Chapter 3
Fundamental concepts in the CSP

3.1 Introduction

In the last chapter we explained that problem reduction serves two purposes: to reduce the problem to one which is hopefully easier to solve, and to recognize insoluble problems. The whole idea of problem reduction is about removing redundant values and redundant compound labels — values and compound labels which appear in no solution tuples. The question is how to identify such values and compound labels.

Over the years, a number of *consistency* concepts have been developed to help in identifying redundant values and compound labels. These concepts are defined in such a way that if the presence of a value in a domain or a compound label in a constraint falsifies them, then it can be deduced to be redundant. In this chapter we shall look at these consistency concepts.

As mentioned in the last chapter, "consistency" in the CSP literature is neither a necessary nor a sufficient condition for a problem to be solvable. In other words, a problem can be inconsistent and yet have valid solutions. It can also be consistent but insoluble. In CSP, "a CSP being consistent with regard to a certain property" should be interpreted as "values and compound labels whose presence would cause certain properties to be false have been removed from their corresponding domains and constraints". Different types of consistency guarantee different properties.

We continue to define concepts both verbally and in First Order Predicate Calculus (FOPC). The former is easier to read, and the latter is unambiguous. Defining these concepts with FOPC allows one to interpret them more precisely.

3.2 Concepts Concerning Satisfiability and Consistency

In this section, we shall first extend the satisfiability concepts introduced in the last two chapters. Then we shall introduce *k-consistency*, which are a concept in general CSPs. Finally, we shall introduce some important consistency concepts for binary CSPs.

3.2.1 Definition of satisfiability

In Chapter 1, we defined the *satisfiability* relationship between compound labels and constraints when the variables of the compound label is a superset of the variables of the constraint (Definition 1-11). In Chapter 2, we introduced constraint expressions, and defined the satisfiability relationship between compound label and constraint expressions (Definitions 2-9 to 2-11). Here we extend these concepts to ***k*-satisfiability**, which is a relationship between a *k*-compound label (Definition 1-4) and a constraint expression.

Definition 3-1:

A *k*-compound label *CL* ***k*-satisfies** a constraint expression *CE* if and only if *CL* satisfies all the constraints in *CE*[1]:

$$\forall\ \text{csp}((Z, D, C)): \forall\ X \subseteq Z:$$
$$(\forall\ x_1, x_2, ..., x_k \in Z: (\forall\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_k \in D_{x_k}:$$
$$k\text{-satisfies}((<x_1,v_1>...<x_k,v_k>), \text{CE}(X)) \equiv$$
$$(\forall\ S: (S \subseteq \{x_1, x_2, ..., x_k\} \cap X \wedge C_S \in \text{CE}(X)) \Rightarrow$$
$$\text{satisfies}((<x_1,v_1> ... <x_k,v_k>), C_S))) \blacksquare$$

Definition 3-2:

A CSP (Z, D, C) is ***k*-satisfiable** if and only if for all subsets of k variables in Z there exists a set of labels for them which satisfies all the relevant constraints in $\text{CE}(Z, (Z, D, C))$:

$$\forall\ \text{csp}((Z, D, C)):$$
$$k\text{-satisfiable}(\ (Z, D, C)\ ) \equiv$$

---

1. Note that the *k* in the definition of *k-satisfies* is actually treated as an argument of the predicate. A more accurate syntax in first order logic would be to put *k* between the brackets, which makes *satisfies(k, Compound_label, C_s)*. The present syntax is adopted for both simplicity and conformation with the CSP literature. The same arrangement applies to the definition of *k*-satisfiable (Definition 3-2), *k*-unsatisfiable (Definition 3-3), (*i, j*)-consistent (Definition 3-14), strong-(*i, j*)-consistency (Definition 3-15), *k*-tree (Definition 3-26), partial-*k*-tree (Definition 3-29) and weak partial-*k*-tree (Definition 3-30) in this chapter.

$(\forall\, x_1, x_2, ..., x_k \in Z: (\exists\, v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_k \in D_{x_k}:$
k-satisfies((<x_1,v_1> ... <x_k,v_k>), CE({x_1,x_2,...,x_k}, (Z, D, C))))) ■

For convenience, we define the *satisfiability of a CSP* below.

Definition 3-2(a):

A CSP which has n variables is **satisfiable** if it is n-satisfiable:

$\forall$ csp((Z, D, C)): $|Z| = n$:
satisfiable((Z, D, C)) ≡ n-satisfiable((Z, D, C)) ■

Definition 3-3:

A CSP is called ***k*-unsatisfiable** if it is not k-satisfiable:

$\forall$ csp(P): k-unsatisfiable(P) ≡ ¬ k-satisfiable(P) ■

3.2.2 Definition of *k*-consistency

In this section, we define the concept of k-consistency in CSPs. If a CSP has n nodes, then k-consistency is defined when k is less than or equal to n.

Definition 3-4:

A CSP is **1-consistent** if and only if every value in every domain satisfies the unary constraints on the subject variable. A CSP is ***k*-consistent** for k greater than 1 if and only if all $(k-1)$-compound labels which satisfy all the relevant constraints can be extended to include any additional variable to form a k-compound label that satisfies all the relevant constraints:

When k = 1:
1-consistent((Z, D, C)) ≡ ($\forall\, x \in Z$: ($\forall\, v \in D_x$: satisfies((<x,v>), C_x))

When $k \geq 2$:
k-consistent((Z, D, C)) ≡
($\forall\, x_1, ..., x_{k-1} \in Z$: ($\forall\, v_1 \in D_{x_1}, ..., v_{k-1} \in D_{x_{k-1}}$:
(k–1)-satisfies((<x_1,v_1>...<x_{k-1},v_{k-1}>), CE({x_1,...,x_{k-1}}, (Z, D, C)))
⇒ ($\forall\, x_k \in Z$: ($\exists\, v_k \in D_{x_k}$:
k-satisfies((<x_1,v_1>...<x_k,v_k>), CE({x_1, ..., x_k}, (Z, D, C)))
)))) ■

Trivial though it may be, it is worth emphasizing that a 1-satisfiable problem needs

not be 1-consistent. This will be the case when some values in some domains violate the constraint on that variable. A 1-consistent problem can also be 1-unsatisfiable. This will be the case when some domains are empty.

If for all variables x in a CSP we remove from D_x all the values which do not satisfy C_x, then the resulting CSP must be equivalent to the original problem. This is because we can be sure that no solutions will be added or deleted (any value that does not appear in C_x cannot appear in the solution tuple). The resulting CSP is 1-consistent by definition.

Definition 3-5:

A CSP which is not k-consistent is called ***k*-inconsistent**:

$\forall$ csp(P): k-inconsistent(P) $\equiv \neg$ k-consistent(P) ■

It may be tempting to believe that k-consistency implies $(k - 1)$-consistency. However, Freuder [1982] points out that a CSP which is k-consistent needs not be $(k - 1)$-consistent. Consider the problem CSP-1 shown in Figure 3.1. A counter-example will show that CSP-1 is 2-inconsistent. The label <*B*,r> 1-satisfies C_B, but no label for *A* is compatible with <*B*,r> (i.e. no 2-compound labels for *A* and *B* which contains <*A*,r> will 2-satisfy CE({*A*,*B*})). Therefore CSP-1 is 2-inconsistent by definition. However, CSP-1 is 3-consistent. This can be seen by observing that the only compound labels that 2-satisfy the constraints are (<*A*,r><*B*,b>), (<*A*,r><*C*,r>) and (<*B*,b><*C*,r>). They are all projections of (<*A*,r><*B*,b><*C*,r>), which 3-satisfies CE({*A*,*B*,*C*}). Therefore, they can all be extended to include the missing variable to form a 3-compound-label which 3-satisfies all the constraints; hence CSP-1 is 3-consistent.

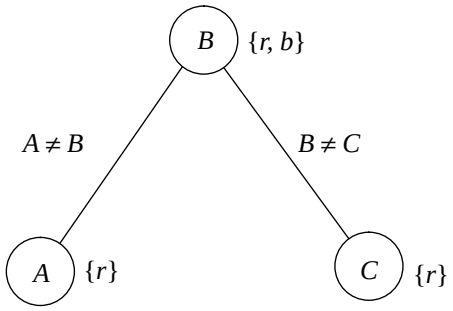


Figure 3.1 CSP-1: example of a 3-consistent CSP which is not 2-consistent (from Freuder [1982])

In view of the weakness of *k*-consistency, Freuder [1982] introduces the concept of **strong *k*-consistency**.

Definition 3-6:

A CSP is **strong *k*-consistent** if it is 1-, 2-, ..., up to *k*-consistent:

$\forall$ csp(P): strong *k*-consistent(P) $\equiv$ ($\forall$ *j*: $1 \le j \le k$: *j*-consistent(P)) ■

By definition, strong *k*-consistency entails strong $(k-1)$-consistency.

3.2.3 Definition of node- and arc-consistency

In Chapter 1, we pointed out that associated to each binary constraint problem is an undirected graph, where the nodes represent the variables and the edges represent the binary constraints. Because of the importance of binary constraint problems, a set of consistency concepts has been defined for them. Borrowing terminology from graph theory, these concepts are called node-, arc- and path-consistency.

Definition 3-7:

A CSP is **node-consistent** (NC) if and only if for all variables all values in its domain satisfy the constraints on that variable. We use *NC(P)* to denote that P is node-consistent:

$\forall$ csp((*Z*, *D*, *C*)):
node-consistent((*Z*, *D*, *C*)) $\equiv$ ($\forall$ $x \in Z$: ($\forall$ $v \in D_x$: satisfies(<*x*,*v*>, C_x)) ■

The formal definition of node-consistency (NC) is exactly the same as 1-consistency.

Recall that we take an arc as a pair of variables, and denote it with (*a*, *b*), where *a* and *b* are the nodes joined by this arc. For undirected graphs, (*a*, *b*) is the same object as (*b*, *a*). (An edge (*x*, *y*) can be seen as a pair of arcs (*x*, *y*) and (*y*, *x*) in a directed graph.)

Definition 3-8:

An arc (*x*, *y*) in the constraint graph of a CSP (*Z*, *D*, *C*) is **arc-consistent** (AC) if and only if for every value *a* in the domain of *x* which satisfies the constraint on *x*, there exists a value in the domain of *y* which is compatible with <*x*,*a*>:

$\forall$ csp((*Z*, *D*, *C*)): $\forall$ *x*, *y* $\in$ *Z*:
AC((*x*, *y*), (*Z*, *D*, *C*)) $\equiv$ ($\forall$ $a \in D_x$: satisfies((<*x*,*a*>), C_x) $\Rightarrow$
$\exists$ $b \in D_y$: (satisfies((<*y*,*b*>), C_y)) $\wedge$ satisfies((<*x*,*a*><*y*,*b*>), $C_{x,y}$))) ■

Definition 3-9:

A CSP is **arc-consistent** (AC) if and only if every arc in its constraint graph is arc-consistent:

$$\forall \text{ csp}((Z, D, C)): \text{AC}((Z, D, C)) \equiv (\forall x, y \in Z: \text{AC}((x, y), (Z, D, C))) \blacksquare$$

In other words, a CSP is arc-consistent if and only if for every variable *x*, for every label <*x*,*a*> that satisfies the constraints on *x*, there exists a value *b* for every variable *y* such that the compound label (<*x*,*a*> <*y*,*b*>) satisfies all the constraints on *x* and *y*. This is exactly the same as the definition of 2-consistency defined in Definition 3-4.

The concept of arc-consistency is useful in searching. Freuder [1982] points out that in any binary CSP which constraint graph forms a tree, a search can be made backtrack-free if both node and arc-consistency are achieved in the problem. The *Waltz filtering* algorithm that we mentioned in Chapter 1 is basically an algorithm which maintains AC throughout the search. The Waltz algorithm and other algorithms for maintaining AC will be discussed in Chapter 4. Here we shall formally state Freuder's theorem.

Theorem 3-1 (mainly due to Freuder, 1982)

A search in a CSP is backtrack-free if the constraint graph of a problem forms a tree and both node- and arc-consistency are achieved in the problem:

$$\forall \text{ csp}(P): P = (Z, D, C) \Rightarrow$$
$$((\text{tree}(\text{G}(P)) \wedge \text{NC}(P) \wedge \text{AC}(P)) \Rightarrow$$
$$\exists <: \text{total\_ordering}(Z, <): \text{backtrack-free}(P, <))$$

Proof

(1) assume that $P = (Z, D, C)$ is a binary CSP which constraint graph G(P) forms a tree. Assume further that both NC(P) and AC(P) are true.

(2) Since G(P) forms a tree, and every node has at most one parent node in a tree, there exists an ordering < such that every node *x* in G(P) except the first node has exactly one node *y* such that $y < x$ and (*x*,*y*) is an edge in G(P).

(3) Let the variables be labelled according to the ordering specified in (2). When a variable *x* is to be labelled, there exists at most one variable *y* which has already been labelled which label could possibly be in con-

flict with x's. But since P is arc-consistent, there is always a value v_x which x may take that is compatible with the label y has taken. Furthermore, since P is NC and v_x is in the domain of x, the label $<x,v_x>$ must satisfy C_x. Therefore, the search is backtrack-free.

(Q.E.D.)

3.2.4 Definition of path-consistency

Definition 3-10:

A path $(x_0, x_1, ..., x_m)$ in the constraint graph for a CSP is **path-consistent** (PC) if and only if for any 2-compound label $(<x_0,v_0> <x_m, v_m>)$ that satisfies all the constraints on x_0 and x_m there exists a label for each of the variables x_1 to x_{m-1} such that every binary constraint on the adjacent variables in the path is satisfied:

$$\forall\ \text{csp}((Z, D, C)): \forall x_0, x_1, x_2, ..., x_m \in Z:$$
$$\text{PC}((x_0, x_1, x_2, ..., x_m), (Z, D, C)) \equiv$$
$$(\forall\ v_0 \in D_{x_0}\ , v_m \in D_{x_m}\ :$$
$$(\text{satisfies}((<x_0,v_0>), C_{x_0}) \wedge \text{satisfies}((<x_m,v_m>), C_{x_m}) \wedge$$
$$\text{satisfies}((<x_0,v_0><x_m,v_m>), C_{x_0, x_m}) \Rightarrow$$
$$(\exists\ v_1 \in D_{x_1}\ , v_2 \in D_{x_2}\ , ... , v_{m-1} \in D_{x_{m-1}}\ :$$
$$\text{satisfies}((<x_1,v_1>), C_{x_1}\}) \wedge ... \wedge$$
$$\text{satisfies}((<x_{m-1},v_{m-1}>), C_{x_{m-1}}) \wedge$$
$$\text{satisfies}((<x_0,v_0><x_1,v_1>), C_{x_0, x_1}) \wedge$$
$$\text{satisfies}((<x_1,v_1><x_2,v_2>), C_{x_1, x_2}) \wedge ... \wedge$$
$$\text{satisfies}((<x_{m-1},v_{m-1}><x_m,v_m>), C_{x_{m-1}, x_m})))) \blacksquare$$

Note carefully that the definition of path-consistency for the path $(x_0, x_1, ..., x_m)$ does not require the values $v_0, v_1, ..., v_m$ to satisfy all the constraints in the constraint expression $\text{CE}(\{x_0, x_1, ..., x_m\}, (Z, D, C))$. For example, since x_3 and x_5 are not adjacent variables in the path, $(<x_3,v_3> <x_5,v_5>)$ needs not satisfy the constraint C_{x_3, x_5}.

Definition 3-11:

A CSP is said to be **path-consistent** if and only if every path in its graph is consistent:

$\forall$ csp((Z, D, C)):
$\quad PC(\ (Z, D, C)\ ) \equiv \forall x_0, x_1, ..., x_m \in Z$: PC($(x_0, x_1, ..., x_m)$, (Z, D, C)) ■

This implies that if a CSP is path-consistent, then for all variables *x* and *y*, whenever a compound label (<*x*,*a*> <*y*,*b*>) satisfies the constraints on both *x* and *y*, there exists a label <*z*,*c*> for every variable *z* such that (<*x*,*a*> <*y*,*b*> <*z*,*c*>) satisfies all the constraints on *x*, *y* and *z*.

3.2.5 Refinement of PC

Montanari [1974] points out that if every path of length 2 of a complete constraint graph is path consistent then the graph is path consistent. We shall prove this theorem under the definitions given above.

Theorem 3-2 (due to Montanari, 1974)

A CSP is path-consistent if and only if all paths of length 2 are path-consistent:

$\forall$ csp(P): $P = (Z, D, C) \Rightarrow$
$\quad (\forall z_1, z_2, z_3 \in Z$: PC($(z_1, z_2, z_3)$, P)) $\Leftrightarrow$
$\qquad (\forall x_1, x_2, ..., x_k \in Z$: PC($(x_1, x_2, ..., x_k)$, P)))

Proof

PC((z_1, z_2, z_3), P) is just a special case of PC($(x_1, x_2, ..., x_k)$, P). So it is trivially true that:

$(\forall z_1, z_2, z_3 \in Z$: PC($(z_1, z_2, z_3)$, P)) $\Leftarrow$
$\qquad (\forall x_1, x_2, ..., x_k \in Z$: PC($(x_1, x_2, ..., x_k)$, P)).

To prove the $\Rightarrow$ aspect of the theorem, let us first assume that:

$(\forall z_1 \in Z \wedge z_2 \in Z \wedge z_3 \in Z$: PC($(z_1, z_2, z_3)$, P). (3.1)

Then we shall prove that all paths are path-consistent using strong induction on the length of the path:

Base Step

When a path has length = 2, the above theorem holds (trivial).

Induction step (by strong induction)

(1) Assume that (3.1) is true for all paths with length between 2 and some integer m:

$(\forall\ 2 \le k \le m: \forall\ x_0, x_1, ..., x_k \in Z: \text{PC}((x_0, x_1, ..., x_k), (Z, D, C))$

(2) Pick any two variables x_0 and x_{m+1}. Assume that v_0 and v_{m+1} are two values such that:

$v_0 \in D_{x_0} \wedge v_{m+1} \in D_{x_{m+1}} \wedge$
$(\text{satisfies}((<x_0,v_0>), C_{x_0}) \wedge \text{satisfies}((<x_{m+1},v_{m+1}>), C_{x_{m+1}}) \wedge$
$\text{satisfies}((<x_0,v_0><x_{m+1},v_{m+1}>), C_{x_0, x_{m+1}}))$

(3) Now pick any m variables $x_1, x_2, ..., x_m$. It must be the case that:

$\exists\ v_m \in D_{x_m}: (\text{satisfies}((<x_m,v_m>), C_{x_m}) \wedge$
$\text{satisfies}((<x_0,v_0><x_m,v_m>), C_{x_0, x_m}) \wedge$
$\text{satisfies}((<x_m,v_m><x_{m+1},v_{m+1}>), C_{x_m, x_{m+1}})))$

(the length of the path (x_0, x_m, x_{m+1}) is 2; by the assumption made in step (1), $\text{PC}((x_0, x_m, x_{m+1}), (Z, D, C))$ holds)

(4) $\text{PC}((x_0, x_1, ..., x_m), (Z, D, C))$ (by assumption in step (1))

(5) $\exists\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_{m-1} \in D_{x_{m-1}}$:

$(\text{satisfies}((<x_1,v_1>), C_{x_1}) \wedge ... \wedge \text{satisfies}((<x_{m-1},v_{m-1}>), D_{x_{m-1}}) \wedge$
$\text{satisfies}((<x_0,v_0><x_1,v_1>), C_{x_0, x_1}) \wedge ... \wedge$
$\text{satisfies}((<x_{m-1},v_{m-1}><x_m,v_m>), C_{x_{m-1}, x_m})$

(by step (4) and definition of PC)

(6) The compound label $(<x_0,v_0><x_1,v_1>...<x_{m+1},v_{m+1}>)$ satisfies C_{x_0}, C_{x_1}, ..., $C_{x_{m+1}}$ and $C_{x_0, x_{m+1}}$, C_{x_0, x_1}, ..., C_{x_{m-1}, x_m}, $C_{x_m, x_{m+1}}$.

(by steps (2), (3) and (5))

(7) $\text{PC}((x_0,x_1,...,x_{m+1}), (Z, D, C))$ (by step (6) and definition of PC)

(Q.E.D.)

Therefore, we can redefine PC as follows.

Definition 3-10(R):

$\forall$ csp((*Z*, *D*, *C*)): $\forall$ *x*, *y*, *z* $\in$ *Z*:
PC((*x*, *y*, *z*), (*Z*, *D*, *C*)) $\equiv$
($\forall$ $v_x \in D_x$, $v_z \in D_z$:
(satisfies((<*x*,v_x>), C_x) $\wedge$ satisfies((<*z*,v_z>), C_z) $\wedge$
satisfies((<*x*,v_x><*z*,v_z>), $C_{x,z}$) $\Rightarrow$
($\exists$ $v_y \in D_y$: satisfies((<*y*,v_y>), C_y) $\wedge$
satisfies((<*x*,v_x><*y*,v_y>), $C_{x,y}$) $\wedge$
satisfies((<*y*,v_y><*z*,v_z>), $C_{y,z}$))) ■

Definition 3-11(R):

$\forall$ csp((*Z*, *D*, *C*)): PC((*Z*, *D*, *C*)) $\equiv$ $\forall$ *x*, *y*, *z* $\in$ *Z*: PC((*x*, *y*, *z*), (*Z*, *D*, *C*)) ■

Freuder [1982] points out that path-consistency is equivalent to 3-consistency in binary CSPs. This is not too difficult to realize under the above definitions. According to our definition of *k*-consistency:

3-consistent((*Z*, *D*, *C*)) $\equiv$ (3.2)
($\forall$ *x*, *z* $\in$ *Z*: ($\forall$ $v_x \in D_x$, $v_z \in D_z$:
(2-satisfies((<*x*,v_x><*z*,v_z>), CE({*x*, *z*}, (*Z*, *D*, *C*)))) $\Rightarrow$
($\forall$ *y* $\in$ *Z*: ($\exists$ $v_y \in D_y$:
3-satisfies((<*x*,v_x><*y*,v_y><*z*,v_z>), CE({*x*, *y*, *z*}, (*Z*, *D*, *C*)))))))

We shall show that this is equivalent to PC((*Z*, *D*, *C*)) for binary CSPs. Firstly, the universal quantifier for y in the definition of 3-consistency (3.2) can be moved to the outmost level to make it comparable with the *z* in the definition in PC in Definition 3-11(R). Secondly, by definition, 2-satisfies((<*x*,v_x><*z*,v_z>), CE({*x*, *z*}, (*Z*, *D*, *C*))) in the definition of 3-consistency is equivalent to:

satisfies((<*x*,v_x>),C_x) $\wedge$ (3.3)
satisfies((<*z*,v_z>),C_z) $\wedge$
satisfies((<*x*,v_x><*z*,v_z>),$C_{x,z}$).

The proposition 3-satisfies((<*x*,v_x><*y*,v_y><*z*,v_z>), CE({*x*, *y*, *z*}, (*Z*, *D*, *C*))) on the right hand side of $\Rightarrow$ of (3.2) is equivalent to:

satisfies((<*x*,v_x>), C_x) $\wedge$ (3.4)
satisfies((<*y*,v_y>), C_y) $\wedge$
satisfies((<*z*,v_z>), C_z) $\wedge$
satisfies((<*x*,v_x><*y*,v_y>),$C_{x,y}$) $\wedge$

satisfies((<x,v_x><z,v_z>),$C_{x,z}$) ∧
satisfies((<y,v_y><z,v_z>),$C_{y,z}$)

in binary CSPs. (Three of the terms in (3.4) appear in (3.3), or appear on the left hand side of ⇒ in (3.2).) By comparing the two well form formulae 3-11(R) and (3.2) after elaborating the definitions, it is not difficult to see that PC in Definition 3-11(R) is equivalent to 3-consistency.

3.2.6 Directional arc- and path-consistency

Dechter & Pearl [1988a] observe that node- plus arc-consistency is stronger than necessary for enabling backtrack-free search in CSPs which constraints form trees. They propose the concept of *directional arc-consistency*, which is a sufficient condition for backtrack-free search in trees. Directional-arc-consistency is defined under total ordering of the variables.

Definition 3-12:

A CSP is **directional arc-consistent** (DAC) under an ordering of the variables if and only if for every label <x,a> which satisfies the constraints on x, there exists a compatible label <y,b> for every variable y which is after x according to the ordering:

∀ csp((*Z*, *D*, *C*)): (∀<: total_ordering(*Z*, <):
DAC((*Z*, *D*, *C*), <) ≡ (∀ *x*, *y* ∈ *Z*: *x* < *y* ⇒ AC((*x*, *y*), (*Z*, *D*, *C*)))) ■

Here AC((*x*, *y*), (*Z*, *D*, *C*)) is defined in Definition 3-8 above. Notice that the difference between AC and DAC is in the qualification of *y*: all *y*'s in *Z* are considered in AC, but only those *y*'s which satisfy *x* < *y* are considered in DAC. Similarly, we can define directional path-consistent.

Definition 3-13:

A CSP P is **directional path-consistent** (DPC) under an ordering of the variables if and only if for every 2-compound label on variables x and z, PC((*x*, *y*, *z*), P) holds for all variables y which is ordered after both x and z:

∀ csp((*Z*, *D*, *C*)):
(∀<: total_ordering(*Z*, <):
DPC((*Z*, *D*, *C*), <) ≡
(∀ *x*, *y*, *z* ∈ *Z*: (*x* < *y* ∧ *z* < *y*) ⇒ PC((*x*, *y*, *z*), (*Z*, *D*, *C*))) ■

The use of NC, AC, DAC, PC and DPC concepts will be elaborated further in Chapter 5.

3.3 Relating Consistency to Satisfiability

Before we continue, let us examine the relationship between the satisfiability and consistency concepts that we have introduced so far. In particular, is *k*-consistency, or strong *k*-consistency, a sufficient or necessary condition for *k*-satisfiability? Is *k*-consistency, or strong *k*-consistency, a sufficient or necessary condition for the satisfiability of a problem? These questions will be answered in this section.

It is not difficult to show that *k*-consistency is insufficient to guarantee satisfiability of a CSP which has more than *k* variables. For example, the colouring problem CSP-2 shown in Figure 3.2 is a 3-consistent but unsatisfiable CSP.

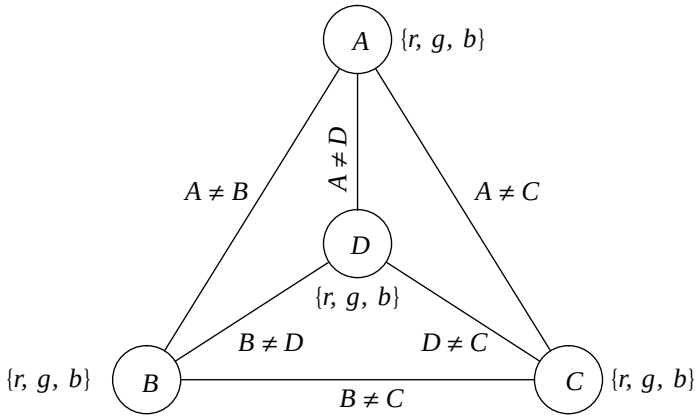


Figure 3.2 CSP-2: example of a 3-consistent but unsatisfiable CSP constraint: no adjacent nodes should take the same value (from Freuder, 1978)

The domains of the variables are shown in curly brackets next to the variables in Figure 3.2. On the edges, the compound labels allowed for the joined nodes are shown. CSP-2 is 3-consistent because whatever combination of three variables that we pick, assigning two of them any two different values from "*r*", "*g*" and "*b*" would allow one to assign the remaining value to the remaining variable without violating any of the constraints on the three variables. But this problem is unsatisfiable because one needs four values to label all the variables without having any adjacent variables taking the same value.

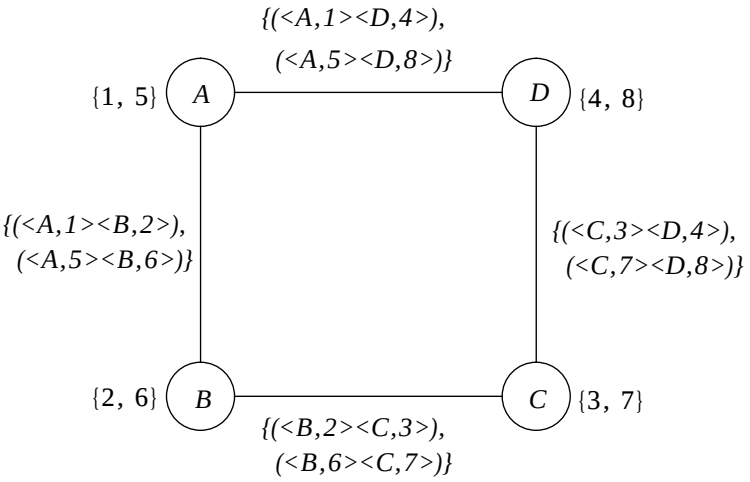


Figure 3.3 CSP-3: a problem which is satisfiable but not path-consistent. The variables are *A*, *B*, *C* and *D*; their domains are shown next to the nodes which represent them. The labels on the edges show the sets of all compatible relations between the variables of the adjacent nodes

The example CSP-3 in Figure 3.3 shows that 3-consistency is not a *necessary condition* for satisfiability either. In CSP-3, if $A = 1$, then from $C_{A,B}$ we have to make $B = 2$, which by $C_{B,C}$ forces $C = 3$, which by $C_{C,D}$ forces $D = 4$. Similarly, if $A = 5$, then $B = 6$, which forces $C = 7$, which in turn forces $D = 8$. Therefore, two and only two compound labels for the variables in the problem satisfy all the constraints:

(<*A*, 1><*B*, 2><*C*, 3><*D*, 4>)

and (<*A*, 5><*B*, 6><*C*, 7><*D*, 8>)

But consider the compound label (<*A*,1><*C*,7>): it satisfies all the constraints C_A, C_C and $C_{A,C}$ ($C_{A,C}$ is not a constraint stated in the problem, and therefore not shown in Figure 3.3). But no value for *B* is compatible with (<*A*,1><*C*,7>) (<*B*,2> violates the constraint $C_{B,C}$ and <*B*,6> violates the constraint $C_{A,B}$). Therefore PC((*A*, *B*, *C*), CSP-3) is false; in other words, PC does not hold for CSP-3. This example shows that path-consistency, or 3-consistency, is not a necessary condition for satisfiability of a CSP. Therefore, *k*-consistency is neither a necessary nor a sufficient condition for satisfiability.

In fact, we can show that a CSP which is 1-consistent need not be 1-satisfiable. This would be the case if there exist some variables which have empty domains, and all the values in the nonempty domains satisfy the constraints of the corresponding variables. Theorem 3-3 states that a CSP which has all the domains and constraints as empty sets is strong *k*-consistent for all *k*.

Theorem 3-3

A CSP in which all the domains are empty sets is strong k-consistent for all k:

$$\forall\ \text{csp}((Z, D, C)) \Rightarrow (\forall\ D_x \in D: D_x = \{\}) \Rightarrow (\forall\ k \leq |Z| : \text{strong } k\text{-consistent}((Z, D, C))))$$

Proof

Let P = (*Z*, *D*, *C*) be a CSP in which all the domains are empty sets. It is 1-unsatisfiable by definition. It is also *h*-unsatisfiable for all $1 \leq h \leq |Z|$ because no *h*-compound label *h*-satisfies C. However, P is 1-consistent (by definition of 1-consistency, since for all *x*, D_x is empty). For any $k > 1$, there exists no $(k-1)$-compound label which $(k-1)$-satisfies the constraints of P, and therefore the left hand side of the "⇒" in the definition of *k*-consistency (Definition 3-4) is never satisfied. Therefore, the proposition *k*-consistency(P) is always true for all *k*, which means strong *k*-consistency(P) is always true.

(Q.E.D.)

One significant implication of Theorem 3-3 is that strong *n*-consistency itself does not guarantee *n*-satisfiability. Careful analysis shows that 1-satisfiability together with strong *k*-consistency is a sufficient (but not necessary) condition to *k*-satisfiability.

Theorem 3-4 (The Satisfiability Theorem)

A CSP which is 1-satisfiable and strong k-consistent is k-satisfiable for all k:

$$\forall\ \text{csp}(P): \text{1-satisfiable}(P) \wedge \text{strong } k\text{-consistent}(P) \Rightarrow k\text{-satisfiable}(P)$$

Proof

Let P = (*Z*, *D*, *C*) be 1-satisfiable and strong *k*-consistent for some integer *k*. Pick an arbitrary subset of *k* variables $S = \{z_1, z_2, ..., z_k\}$ from *Z*. We shall prove that there exists at least one compound label for all the variables in *S* which satisfies all the relevant constraints (i.e. CE(*S*, P)).

Since P is 1-satisfiable, for any arbitrary element x_1 that we pick from S, we can at least find one value v_1 from the domain of x_1 such that satisfies(<x_1,v_1>, C_{x_1}) holds. Furthermore, since P is 2-consistent, for any other variable x_2 that we pick from S, we would be able to find a compound label (<x_1, v_1><x_2, v_2>) which satisfies CE({x_1,x_2}, P). Since P is strong-k-consistent, it should not be difficult to show by induction that for any 3rd, 4th, ..., kth variables in S that we pick, we shall be able to find 3-, 4-, ..., k-compound labels that satisfy the corresponding constraints CE({x_1, x_2, ..., x_k}, P). Therefore, the subproblem on S is satisfiable, and so P is k-satisfiable.

(Q.E.D.)

We summarize below the results that we have concluded so far:

(1) k-satisfiability subsumes (k – 1)-satisfiability (trivial).

(2) However, k-consistency does not entail (k – 1)-consistency. This is illustrated by example CSP-1, which is 3-consistent but not 2-consistent. But some k-consistent CSPs must be (k – 1)-consistent, and *vice versa*. This leads to the definition of strong k-consistency, which entails strong (k – 1)-consistency.

(3) k-consistency does not guarantee 1-satisfiability. Consequently, k-consistency does not guarantee h-satisfiability for any h. This is true for $k \leq h$, as illustrated in the example CSP-2 which is 3-consistent but not 4-satisfiable. It is also true for $k > h$, as it is illustrated by the colouring problem CSP-4 in Figure 3.4, which is 3-consistent, but not 2-satisfiable.

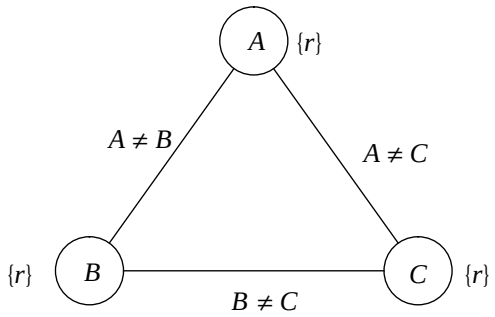


Figure 3.4 CSP-4: a CSP which is 1 satisfiable and 3-consistent, but 2-inconsistent and 2-unsatisfiable (it is 3-consistent because there is no 2-compound label which satisfies any of the binary constraints)

(4) Similarly, *h*-satisfiability does not guarantee *k*-consistency when $k > 1$. We have shown in the CSP-3 example that a 4-satisfiable CSP need not be 3-consistent.
(5) Neither does strong *k*-consistency guarantee *k*-satisfiability: Theorem 3-3 indicates that if the domain of all variables are empty, the problem is 2-consistent.
(6) However (as proved in Theorem 3-4), 1-satisfiability plus strong *k*-consistency guarantees *k*-satisfiability. A little reflection should convince the readers that this means a strong *k*-consistent CSP without any empty domain is *k*-satisfiable.

These results will be summarized in Figure 3.7 at the end of this chapter, after the introduction of more consistency concepts.

3.4 (*i, j*)-consistency

The concept of *k*-consistency is generalized to (*i, j*)-consistency by Freuder.

Definition 3-14:

A CSP is **(*i, j*)-consistent** if, given any *i*-compound label that satisfies all the constraints on a set of *i* variables *I*, and given any set of *j* or less variables *K* which does not overlap with *I*, one can always find for the variables in *K* values which are compatible with the compound label for *I*. In other words, the combined compound label for both *I* and *K* satisfies all the constraints on *I* union *K*:

$$
\begin{aligned}
&\forall\ \text{csp}((Z, D, C)): \forall\ i, j: \\
&\quad (i, j)\text{-consistent}((Z, D, C)) \equiv \\
&\qquad (\forall\ x_1, x_2, ..., x_i \in Z: \forall\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ..., v_i \in D_{x_i}: \\
&\qquad\quad (\text{satisfies}((<x_1,v_1>...<x_i,v_i>), \text{CE}(\{x_1, x_2, ..., x_i\}, (Z, D, C))) \Rightarrow \\
&\qquad\qquad (\forall\ x'_1, x'_2, ..., x'_k \in Z: k \le j: \\
&\qquad\qquad\quad ((\{x_1,x_2,...,x_i\} \cap \{x'_1, x'_2, ..., x'_k\} = \{\}) \Rightarrow \\
&\qquad\qquad\qquad (\exists\ v'_1 \in D_{x'_1}, v'_2 \in D_{x'_2}, ..., v'_k \in D_{x'_k}: \\
&\qquad\qquad\qquad\quad \text{satisfies}((<x_1,v_1>...<x_i,v_i><x'_1,v'_1>...<x'_k,v'_k>), \\
&\qquad\qquad\qquad\qquad \text{CE}(\{x_1, ..., x_i, x'_1, ..., x'_k\}, (Z, D, C)))))))) \ \blacksquare
\end{aligned}
$$

It follows that *k*-consistency is equivalent to $(k - 1, 1)$-consistency.

Definition 3-15:

A CSP is **strong (*i, j*)-consistent** if it is (*k, j*)-consistent for all $1 \le k \le i$:

$\forall$ csp(P): $\forall$ i, j:
strong-(i, j)-consistent(P) $\equiv$ ($\forall$ k: $1 \le k \le i$: (k, j)-consistent(P)) ■

It should be pointed out that a CSP which is (i, j)-consistent need not be (i', j')-consistent even though $i + j = i' + j'$ may hold. (i, j)-consistency has interesting properties which is relevant to backtracking search. Interesting properties of (i, j)-consistent CSPs are illustrated in Chapter 7, when we explain search techniques.

3.5 Redundancy of Constraints

In Chapter 2, we defined the concept of redundancy on *values* and *compound labels*. Dechter & Dechter [1987] extend these concepts to the redundancy of *constraints*. These concepts, which could help us to derive algorithms for removing constraints, are defined in this section.

Definition 3-16:

A k-constraint in a CSP is **redundant** if it does not restrict the k-compound labels of the subject variables further than the restrictions imposed by other constraints in that problem. This means that the removal of it does not change (increase) the set of solution tuples in the problem:

$\forall$ csp((Z, D, C)): ($\forall$ $S \subseteq Z$: $C_S \in C$:
redundant(C_S, (Z, D, C)) $\equiv$
($\forall$ T : solution_tuple(T , (Z, D, $C - \{C_S\}$)) $\Leftrightarrow$
solution_tuple(T , (Z, D, C)))) ■

For example, if x, y and z are integer variables, and $x < y$, $x < z$ and $y < z$ are three constraints, then the constraint $x < z$ is redundant because it imposes no more constraints to x and z than $x < y$ and $y < z$ together.

Redundancy is in general difficult to detect. However, some redundant constraints could be detected quite easily. In Dechter & Dechter [1987], which focuses on binary CSPs, a number of concepts for helping to identify redundant binary constraints are introduced.

Definition 3-17:

A 2-compound label *CL* is **path-allowed** by a path *PA* that begins and ends with the variables of *CL* if in addition to the labels in *CL*, one can assign a value to each of the variables in the path satisfying all the binary constraints on adjacent nodes in the path:

$\forall$ csp((*Z*, *D*, *C*)): ($\forall$ x_0, x_1, ..., $x_k \in Z$: ($\forall$ $v_0 \in D_{x_0}$, $v_k \in D_{x_k}$:
path-allowed((<x_0,v_0><x_k,v_k>), (x_1, x_2, ..., x_{k-1}), (*Z*, *D*, *C*)) ≡
($\exists$ $v_1 \in D_{x_1}$, $v_2 \in D_{x_2}$, ..., $v_{k-1} \in D_{x_{k-1}}$:
($\forall$ *p*: $0 \le p < k$: satisfies((<x_p,v_p><x_{p+1},v_{p+1}>), $C_{x_p, x_{p+1}}$)))) ■

Definition 3-18:

A 2-compound label is **path-induced** in a problem if it is path-allowed by every path in the graph which represents the problem:

$\forall$ csp((*Z*, *D*, *C*)): $\forall$ *x*, *y* $\in$ *Z*: $\forall$ *a* $\in$ D_x, *b* $\in$ D_y:
path-induced((<*x*,*a*><*y*,*b*>), (*Z*, *D*, *C*)) ≡
($\forall$ $z_1 \in Z$, $z_2 \in Z$, ..., $z_m \in Z$:
path-allowed((<*x*,*a*><*y*,*b*>), (z_1, z_2, ..., z_m), (*Z*, *D*, *C*)) ■

Definition 3-19:

A binary constraint is path-redundant if no 2-compound label which violates it is path induced. In other words, it does not restrict the choice of compound labels for the subject variables more than the paths have already done so:

$\forall$ csp((*Z*, *D*, *C*)): $\forall$ $C_{x,y} \in C$:
path-redundant($C_{x,y}$, (*Z*, *D*, *C*)) ≡
($\forall$ *a* $\in$ D_x, *b* $\in$ D_y:
((<*x*,*a*><*y*,*b*>) $\notin$ $C_{x,y}$) $\Rightarrow$
$\neg$ path-induced((<*x*,*a*><*y*,*b*>), (*Z*, *D*, *C*))) ■

A binary constraint can be removed if it is path-redundant. Removal of path-redundant constraints would change the topology of the constraint graph. This would be desirable if the resulting topology of the constraint graph enables specialized algorithms to be applied — e.g. when the resulting constraint graphs are unconnected or acyclic. This will be elaborated further in Chapter 7.

3.6 More Graph-related Concepts

Every binary CSP is associated with a constraint graph. Many CSP solving techniques are designed to exploit the topology of the constraint graphs of the problems. To help in illustrating those techniques later in this book, we shall define the relevant concepts in graph theory in this section. Readers may choose to skip this section and refer to it for the relevant definitions when they are encountered in

subsequent chapters.

In this section, we shall continue to denote graphs by G, where $G = (V, E)$, with V being a set of nodes and E being a set of edges (see Definition 1-15). All graphs referred to in this section are undirected graphs without loops.

The first group of definitions are about the width of a graph. These concepts are useful for explaining the ordering of variables in searching, which will be discussed in Chapter 6.

Definition 3-20:

Given a graph (V, E) and a total ordering on its nodes, the **width of a node** v is the number of nodes that are before and adjacent to v:

$\forall$ graph((V, E)): ($\forall$ <: total_ordering(V, <): ($\forall\ x \in V$:
width(x, (V, E), <) $\equiv$ $\left|\ \{y \mid y < x \wedge (x, y) \in E\}\ \right|$)) ■

Definition 3-21:

The **width of a graph under an ordering** is the maximum width of all the nodes in the in the graph under that ordering:

$\forall$ graph((V, E)): ($\forall$ <: total_ordering(V, <):
width((V, E), <) $\equiv$ MAX width(x, (V, E), <): $x \in V$) ■

Definition 3-22:

The **width of a graph** is the minimum width of the graph under all possible orderings of its nodes:

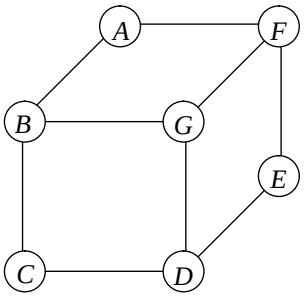
$\forall$ graph((V, E)): width((V, E)) $\equiv$ MIN width((V, E), <): total_ordering(V, <) ■

For example, Figure 3.5(a) shows a graph. If the ordering of the nodes is (A, B, C, D, E, F, G), then the width of the nodes are 0, 1, 1, 1, 1, 2, 3, respectively (Figure 3.5(b)). Therefore the width of this ordering is 3, which is the maximum width among all nodes. Should the ordering be (G, F, E, D, C, B, A), the width of the nodes would be 0, 1, 1, 2, 1, 2, 2 (Figure 3.5(c)). The width of this ordering is 2.

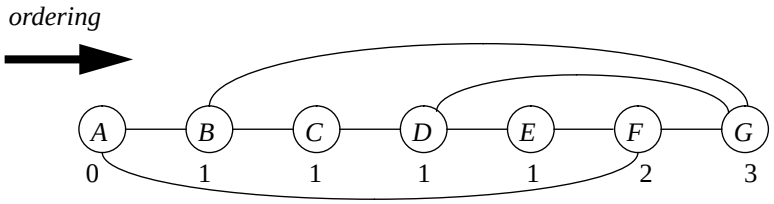
Definition 3-23:

A graph $G' = (V', E')$ is **induced by** another graph $G = (V, E)$ if V' is a subset of V, and E' is the set of all the edges in E which join the nodes in V':

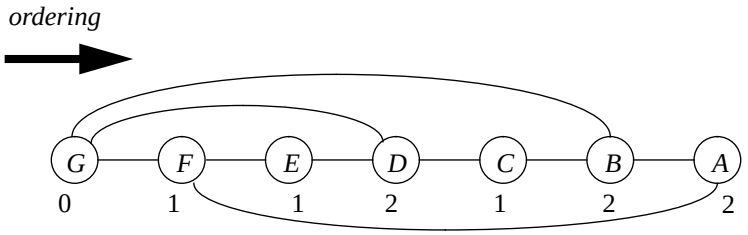
$\forall$ graph((V, E)), graph((V', E')):
induced_by((V', E'), (V, E)) $\equiv$



(a) A constraint graph to be labelled



(b) Width of the nodes given the order *A*, *B*, *C*, *D*, *E*, *F*, *G*



(c) Width of the nodes given the order *G*, *F*, *E*, *D*, *C*, *B*, *A*

Figure 3.5 Example of a constraint graph with the width of different orderings shown

$$(V' \subseteq V) \wedge (E' = \{(a, b) \mid (a, b) \in E \wedge a \in V' \wedge b \in V'\}) \blacksquare$$

Definition 3-24:

The **neighbourhood** of a node v in a graph G is the set of all the nodes in G which are adjacent to v:

$\forall$ graph((V, E)): ($\forall$ $v \in V$:
neighbourhood(v, (V, E)) $\equiv$ $\{w \mid w \in V \wedge (v, w) \in E\}$) $\blacksquare$

Definition 3-25:

The **degree (which is sometimes called valency in the literature) of a node** in a graph is the number of nodes to which this node is adjacent:

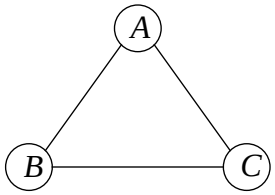
$\forall$ graph((V, E)): ($\forall$ $v \in V$: degree(v, (V, E)) $\equiv$ $\left|\right.$ neighbourhood(v, (V, E)) $\left|\right.$) $\blacksquare$

Definition 3-26:

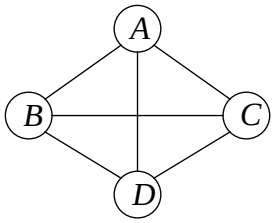
A ***k*-tree** is either a complete graph with k nodes, or a graph in which one can find a node v that satisfies three conditions: (1) that it is adjacent to k nodes; (2) its neighbourhood (which has k nodes) forms a complete graph; and (3) the graph without both v and the edges involving v forms a k-tree. A k-tree which is a complete graph G with k nodes is called a **trivial *k*-tree**, denoted *trivial_k-tree*(G):

$\forall$ graph((V, E)):
k-tree((V, E)) $\equiv$
(($\left|\right.$ V $\left|\right.$ = k $\wedge$ complete_graph((V, E))) $\vee$
$\exists$ $v \in V$: (degree(v, (V, E)) = k $\wedge$
(G' = (neighbourhood(v, (V, E)), E') $\Rightarrow$
induced_by(G', (V, E)) $\wedge$ complete_graph(G')) $\wedge$
(k-tree($(V - \{v\}, E - \{(v, w) \mid (v, w) \in E\})$)))) $\blacksquare$

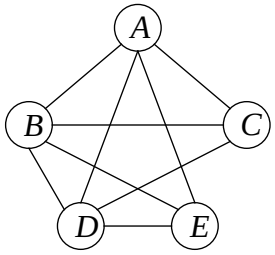
Figure 3.6 shows examples and counter-examples of k-trees. The graph in Figure 3.6(a) is a trivial 3-tree which is a complete graph with 3 nodes. The graph in Figure 3.6(b) is the graph in Figure 3.6(a) with an extra node D added. It is a 3-tree because there exists a node D which has a degree of 3, its neighbourhood $\{A, B, C\}$ forms a complete graph and the graph without D is a (trivial) 3-tree. In each of the graphs in Figures 3.6(c) and (d), one more node is added. They are 3-trees as the added nodes satisfy the above conditions. The graph in Figure 3.6(e) is not a 3-tree because there exists only one node, F, which degree is 3. But the neighbourhood of F, which is the set $\{C, D, E\}$, does not form a complete graph.



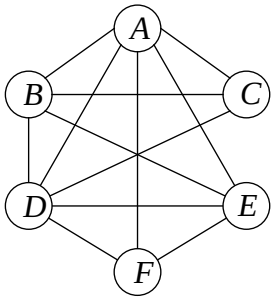
(a) A (trivial) 3-tree



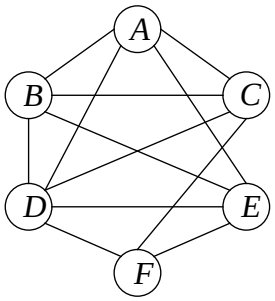
(b) A 3-tree with 4 nodes,
which is also a trivial 4-tree



(c) A 3-tree with 5 nodes



(d) A 3-tree with 6 nodes



(e) A graph which is NOT a 3-tree (*F* is the only node with degree=3,
but the neighbourhood of F, {*C*,*D*,*E*} is not a complete graph)

Figure 3.6 Examples and counter-examples of k-trees

Definition 3-27:

$G' = (V', E')$ is a **partial graph** of $G = (V, E)$ if V' is a subset of V and E' is a subset of the edges in E which join the nodes in V'. *Partial_graph(G', G)* reads "G' is a partial graph of G":

$\forall$ graph((*V*, *E*)), graph((*V'*, *E'*)):
  partial_graph((*V'*, *E'*), (*V*, *E*)) $\equiv$
    $(V' \subseteq V \wedge E' \subseteq \{(a, b) \mid a \in V' \wedge b \in V' \wedge (a, b) \in E\})$ ■

In the above definition, when E' equals the set of *all* the edges in E which join the nodes in V', G' is induced by G.

Definition 3-28:

Graph G **embeds** graph G' if G' is a partial graph of G and G is a k-tree for some integer k. *Embedding(G, G')* reads "G embeds G'":

$\forall$ graph(G), graph(G'): embedding(G, G') $\equiv$
  (partial_graph(G', G) $\wedge$ ($\exists$ k: k-tree(G))) ■

Definition 3-29:

G is a **partial-*k*-tree** if there exists a k-tree G' of which G is a partial graph:

$\forall$ graph(G): $\forall$ k: partial-k-tree(G) $\equiv$ ($\exists$ G': partial_graph(G, G') $\wedge$ k-tree(G'))
■

According to this definition, any graph is a partial-k-tree for a sufficiently large k.

Definition 3-30:

A **weak-*k*-tree** is either a complete graph with k or less nodes, or a graph in which one can find a node v which satisfies three conditions: (1) that it is adjacent to no more than k nodes; (2) its neighbourhood forms a complete graph; and (3) the graph without v and edges involving v forms a weak-k-tree:

$\forall$ graph((*V*, *E*)):
  weak-k-tree((*V*, *E*)) $\equiv$
    (($|V| \leq k \wedge$ complete((*V*, *E*))) $\vee$
      $\exists$ $v \in V$: (degree(v, (*V*, *E*)) $\leq k \wedge$
        (G'=(neighbourhood(v, (*V*, *E*)), E') $\Rightarrow$
          induced_by(G', (*V*, *E*)) $\wedge$ complete(G')) $\wedge$
        (weak-k-tree($(V - \{v\}, E - \{(v, w) \mid (v, w) \in E\})$)))) ■

The definition of **weak-*k*-tree** is similar to *k*-tree except that all "= *k*" are replaced by "≤ *k*".

3.7 Discussion and Summary

A number of concepts, many of which surround the notion of *consistency*, have been defined in this chapter. Many of these concepts are directly related to problem reduction and search methods, which we shall introduce in the coming chapters.

In this chapter, we first introduced the concept of *k*-satisfiability. Then we introduced a number of consistency concepts that may help in identifying redundant values in the domains and redundant compound labels in the constraints. Node-, arc-, path-, directional arc- and directional path-consistency are some of the best known consistency concepts for binary constraint problems, while *k*-consistency and strong *k*-consistency are concepts for general CSPs. Figure 3.7 summarizes the relationship among the consistency concepts introduced in this chapter. In general, the stronger the level of consistency one achieves, the more computation one requires, but the more redundant values and redundant compound labels one can be expected to remove.

We have pointed out in this chapter that not even strong-*k*-consistency is strong enough to be a necessary condition for *k*-satisfiability. We have shown that 1-satisfiability together with strong *k*-consistency guarantees *k*-satisfiability.

We have also introduced Freuder's (*i, j*)-consistency, which is an extension of *k*-consistency. The concept of redundancy in Chapter 2 is extended to constraints. Finally, we introduced more concepts in graph theory. These concepts will be used in the chapters to come.

3.8 Bibliographical Remarks

Although we suggest that problem reduction has a good chance of reducing the problem to easier problems, Prosser [1992] points out that there are exceptions. The ideas of node-, arc- and path-consistency originate from Montanari [1974]; these terminologies are well summarized by Mackworth [1977]. Freuder [1978] first introduced the more general concept of *k*-consistency, which is later extended to strong-*k*-consistency. Freuder also points out the sufficient condition for backtrack-free search, which lays the foundation for a number of specialized CSP solving techniques which we shall introduce in Chapter 7. Although the maintenance of directional arc-consistency (DAC) has long been proposed and analysed in searching (e.g. see Haralick & Elliott, 1980), the concept was never formally defined until Dechter & Pearl [1988a]. Freuder [1985] introduces the concept of (*i, j*)-consistency and *k*-trees. The use of them in CSP solving will be explored in Chapter 7. Most of

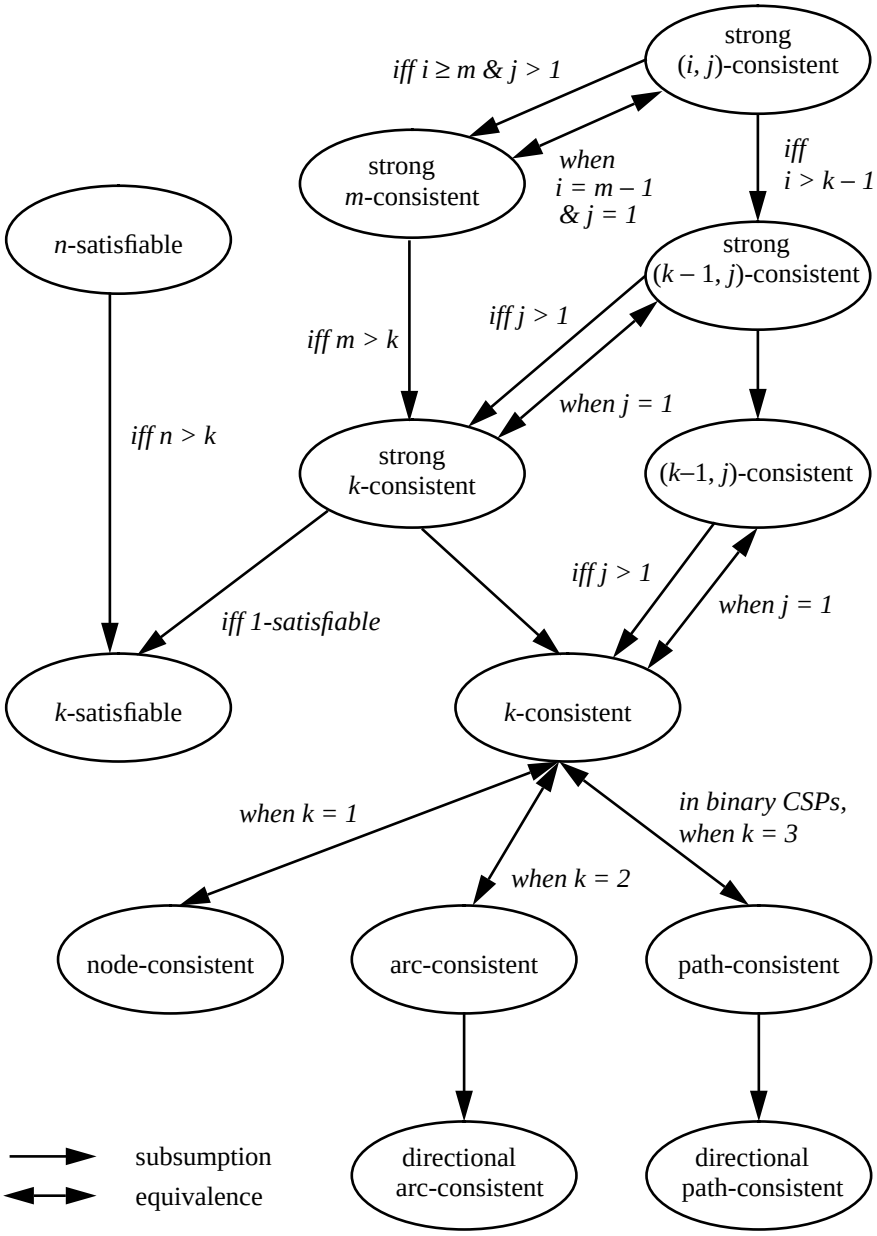


Figure 3.7 Relationship among some consistency and satisfiability properties

the above concepts have been defined verbally in the literature. Tsang [1989] makes an attempt to define them in first order logic, as well as outlining the relationship between consistency and satisfiability concepts.