In Chap. 3, we saw how useful the theory of equality logic with uninterpreted-function (EUF) is. In this chapter, we concentrate on decision procedures for EUF and on algorithms for simplifying EUF formulas. Recall that we are solving the satisfiability problem for formulas in negation normal form (NNF – see Definition 1.10) without constants, as those can be removed with, for example, Algorithm 3.1.1. With the exception of Sect. 4.1, we handle equality logic without uninterpreted functions, assuming that these are eliminated by one of the reduction methods introduced in Chap. 3.

4.1 Deciding a Conjunction of Equalities and Uninterpreted Functions with Congruence Closure

We begin by showing a method for solving a conjunction of equalities and uninterpreted functions. As is the case for most of the theories that we consider in this book, the satisfiability problem for conjunctions of predicates can be solved in polynomial time.

Note that a decision procedure for a conjunction of equality predicates is not sufficient to support uninterpreted functions as well, as both Ackermann’s and Bryant’s reductions (Chap. 3) introduce disjunctions into the formula.

As an alternative, Shostak proposed in 1978 a method for handling uninterpreted functions directly. Starting from a conjunction $\varphi^{uf}$ of equalities and disequalities over variables and uninterpreted functions, he proposed a two-stage algorithm (see Algorithm 4.1.1), which is based on computing equivalence classes. The version of the algorithm that is presented here assumes that the uninterpreted functions have a single argument. The extension to the general case is left as an exercise (Problem 4.3).
Algorithm 4.1.1: Congruence-Closure

Input: A conjunction $\varphi^{UF}$ of equality predicates over variables and uninterpreted functions
Output: “Satisfiable” if $\varphi^{UF}$ is satisfiable, and “Unsatisfiable” otherwise

1. Build congruence-closed equivalence classes.
   (a) Initially, put two terms $t_1, t_2$ (either variables or uninterpreted-function instances) in their own equivalence class if $(t_1 = t_2)$ is a predicate in $\varphi^{UF}$. All other variables form singleton equivalence classes.
   (b) Given two equivalence classes with a shared term, merge them. Repeat until there are no more classes to be merged.
   (c) Compute the congruence closure: given two terms $t_i, t_j$ that are in the same class and that $F(t_i)$ and $F(t_j)$ are terms in $\varphi^{UF}$ for some uninterpreted function $F$, merge the classes of $F(t_i)$ and $F(t_j)$. Repeat until there are no more such instances.

2. If there exists a disequality $t_i \neq t_j$ in $\varphi^{UF}$ such that $t_i$ and $t_j$ are in the same equivalence class, return “Unsatisfiable”. Otherwise return “Satisfiable”.

Example 4.1. Consider the conjunction

$$\varphi^{UF} := x_1 = x_2 \land x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1 \land F(x_1) \neq F(x_3).$$ (4.1)

Initially, the equivalence classes are

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}.$$ (4.2)

Step 1(b) of Algorithm 4.1.1 merges the first two classes:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}.$$ (4.3)

The next step also merges the classes containing $F(x_1)$ and $F(x_3)$, because $x_1$ and $x_2$ are in the same class:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\}.$$ (4.4)

In step 2, we note that $F(x_1) \neq F(x_3)$ is a predicate in $\varphi^{UF}$, but that $F(x_1)$ and $F(x_3)$ are in the same class. Hence, $\varphi^{UF}$ is unsatisfiable.

Variants of Algorithm 4.1.1 can be implemented efficiently with a union–find data structure, which results in a time complexity of $O(n \log n)$ (see, for example, [141]).
In the original presentation of his method, Shostak implemented support for disjunctions by means of case-splitting, which is the bottleneck in this method. For example, given the formula

\[ \varphi_{UF} := x_1 = x_2 \lor (x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1 \land F(x_1) \neq F(x_3)) , \]  

(4.5)

he considered separately the two cases corresponding to the left and right parts of the disjunction. This can work well as long as there are not too many cases to consider.

The more interesting question is how to solve the general case efficiently, where the given formula has an arbitrary Boolean structure. This problem arises with all the theories that we study in this book. There are two main approaches. A highly efficient method is to combine a SAT solver with an algorithm such as Algorithm 4.1.1, where the former searches for a satisfying assignment to the Boolean skeleton of the formula (an abstraction of the formula where each unique predicate is replaced with a new Boolean variable), and the latter is used to check whether this assignment corresponds to a satisfying assignment to the equality predicates – we dedicate Chap. 11 to this technique. A second approach is based on a full reduction to propositional logic, and is the subject of the rest of this chapter.

4.2 Basic Concepts

In this section, we present several basic terms that are used later in the chapter. We assume from here on that uninterpreted functions have already been eliminated, i.e., that we are solving the satisfiability problem for equality logic without uninterpreted functions. Recall that we are also assuming that the formula is given to us in NNF and without constants. Recall further that an atom in such formulas is an equality predicate, and a literal is either an atom or its negation (see Definition 1.11). Given an equality logic formula \( \varphi^E \), we denote the set of atoms of \( \varphi^E \) by \( \text{At}(\varphi^E) \).

**Definition 4.2 (equality and disequality literals sets).** The equality literals set \( E_\equiv \) of an equality logic formula \( \varphi^E \) is the set of positive literals in \( \varphi^E \). The disequality literals set \( E_\neq \) of an equality logic formula \( \varphi^E \) is the set of disequality literals in \( \varphi^E \).

It is possible, of course, that an equality may appear in the equality literals set and its negation (see Definition 1.11). Given an equality logic formula \( \varphi^E \), we denote the set of atoms of \( \varphi^E \) by \( \text{At}(\varphi^E) \).

**Example 4.3.** Consider the negation normal form of \( \neg \varphi^E \) in (3.43):

\[
\neg \varphi^E := \left( (x_1 \neq x_2 \lor y_1 \neq y_2 \lor f_1 = f_2) \land (u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land (u_1 = f_1 \land u_2 = f_2 \land z = g_1) \right) \land z \neq g_2 .
\]  

(4.6)
We therefore have
\[ E_\simeq := \{(f_1 = f_2), \ (g_1 = g_2), \ (u_1 = f_1), \ (u_2 = f_2), \ (z = g_1)\} \]
\[ E_{\not\simeq} := \{(x_1 \not= x_2), \ (y_1 \not= y_2), \ (u_1 \not= f_1), \ (u_2 \not= f_2), \ (z \not= g_2)\} . \] (4.7)

**Definition 4.4 (equality graph).** Given an equality logic formula \( \varphi^E \) in NNF, the equality graph that corresponds to \( \varphi^E \), denoted by \( G^E(\varphi^E) \), is an undirected graph \( (V, E_\simeq, E_{\not\simeq}) \) where the nodes in \( V \) correspond to the variables in \( \varphi^E \), the edges in \( E_\simeq \) correspond to the predicates in the equality literals set of \( \varphi^E \) and the edges in \( E_{\not\simeq} \) correspond to the predicates in the disequality literals set of \( \varphi^E \).

Note that we overload the symbols \( E_\simeq \) and \( E_{\not\simeq} \) so that each represents both the literals sets and the edges that represent them in the equality graph. Similarly, when we say that an assignment “satisfies an edge”, we mean that it satisfies the literal represented by that edge.

We may write simply \( G^E \) for an equality graph when the formula it corresponds to is clear from the context. Graphically, equality literals are represented as dashed edges and disequality literals as solid edges, as illustrated in Fig. 4.1.

It is important to note that the equality graph \( G^E(\varphi^E) \) represents an abstraction of \( \varphi^E \): more specifically, it represents all the equality logic formulas that have the same literals sets as \( \varphi^E \). Since it disregards the Boolean connectives, it can represent both a satisfiable and an unsatisfiable formula. For example, although \( x_1 = x_2 \land x_1 \not= x_2 \) is unsatisfiable and \( x_1 = x_2 \lor x_1 \not= x_2 \) is satisfiable, both formulas are represented by the same equality graph.

**Definition 4.5 (equality path).** An equality path in an equality graph \( G^E \) is a path consisting of \( E_\simeq \) edges. We denote by \( x =^* y \) the fact that there exists an equality path from \( x \) to \( y \) in \( G^E \), where \( x, y \in V \).

**Definition 4.6 (disequality path).** A disequality path in an equality graph \( G^E \) is a path consisting of \( E_\simeq \) edges and a single \( E_{\not\simeq} \) edge. We denote by \( x \not=^* y \) the fact that there exists a disequality path from \( x \) to \( y \) in \( G^E \), where \( x, y \in V \).
Similarly, we use the terms simple equality path and simple disequality path when the path is required to be loop-free.

Consider Fig. 4.1 and observe, for example, that $x_2 =^* x_4$ owing to the path $x_2, x_5, x_4$, and $x_2 \neq^* x_4$ owing to the path $x_2, x_5, x_1, x_4$. In this case, both paths are simple. Intuitively, if $x =^* y$ in $G^E(\varphi^E)$, then it might be necessary to assign the two variables equal values in order to satisfy $\varphi^E$. We say “might” because, once again, the equality graph obscures details about $\varphi^E$, as it disregards the Boolean structure of $\varphi^E$. The only fact that we know from $x =^* y$ is that there exist formulas whose equality graph is $G^E(\varphi^E)$ and that in any assignment satisfying them, $x = y$. However, we do not know whether $\varphi^E$ is one of them. A disequality path $x \neq^* y$ in $G^E(\varphi^E)$ implies the opposite: it might be necessary to assign different values to $x$ and $y$ in order to satisfy $\varphi^E$.

The case in which both $x =^* y$ and $x \neq^* y$ hold in $G^E(\varphi^E)$ requires special attention. We say that the graph, in this case, contains a contradictory cycle.

**Definition 4.7 (contradictory cycle).** In an equality graph, a contradictory cycle is a cycle with exactly one disequality edge.

For every pair of nodes $x, y$ in a contradictory cycle, it holds that $x =^* y$ and $x \neq^* y$.

Contradictory cycles are of special interest to us because the conjunction of the literals corresponding to their edges is unsatisfiable. Furthermore, since we have assumed that there are no constants in the formula, these are the only topologies that have this property. Consider, for example, a contradictory cycle with nodes $x_1, \ldots, x_k$ in which $(x_1, x_k)$ is the disequality edge. The conjunction

$$x_1 = x_2 \land \ldots \land x_{k-1} = x_k \land x_k \neq x_1$$

(4.8)

is clearly unsatisfiable.

All the decision procedures that we consider refer explicitly or implicitly to contradictory cycles. For most algorithms we can further simplify this definition by considering only simple contradictory cycles. A cycle is simple if it is represented by a path in which none of the vertices is repeated, other than the starting and ending vertices.

### 4.3 Simplifications of the Formula

Regardless of the algorithm that is used for deciding the satisfiability of a given equality logic formula $\varphi^E$, it is almost always the case that $\varphi^E$ can be simplified a great deal before the algorithm is invoked. Algorithm 4.3.1 presents such a simplification.
Algorithm 4.3.1: Simplify-Equality-Formula

**Input:** An equality formula $\varphi^E$.

**Output:** An equality formula $\varphi^{E'}$ equisatisfiable with $\varphi^E$, with length less than or equal to the length of $\varphi^E$.

1. Let $\varphi^{E'} := \varphi^E$.
2. Construct the equality graph $G^E(\varphi^{E'})$.
3. Replace each pure literal in $\varphi^{E'}$ whose corresponding edge is not part of a simple contradictory cycle with TRUE.
4. Simplify $\varphi^{E'}$ with respect to the Boolean constants TRUE and FALSE (e.g., replace $\text{TRUE} \lor \phi$ with $\text{TRUE}$, and $\text{FALSE} \land \phi$ with $\text{FALSE}$).
5. If any rewriting has occurred in the previous two steps, go to step 2.
6. Return $\varphi^{E'}$.

The following example illustrates the steps of Algorithm 4.3.1.

**Example 4.8.** Consider (4.6). Figure 4.2 illustrates $G^E(\varphi)$, the equality graph corresponding to $\varphi^E$.

![Equality Graph](image)

Fig. 4.2. The equality graph corresponding to Example 4.8. The edges $f_1 = f_2$, $x_1 \neq x_2$ and $y_1 \neq y_2$ are not part of any contradictory cycle, and hence their respective predicates in the formula can be replaced with $\text{TRUE}$.

In this case, the edges $f_1 = f_2$, $x_1 \neq x_2$ and $y_1 \neq y_2$ are not part of any simple contradictory cycle and can therefore be substituted by TRUE. This results in

$$
\varphi^{E'} := \begin{cases}
\text{(TRUE } \lor \text{TRUE } \lor \text{TRUE}) \land \\
(u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \land z = g_1 \land z \neq g_2)
\end{cases},
$$

(4.9)

which, after simplification according to step 4, is equal to

$$
\varphi^{E'} := \begin{cases}
(u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \land z = g_1 \land z \neq g_2)
\end{cases}.
$$

(4.10)

Reconstructing the equality graph after this simplification does not yield any more simplifications, and the algorithm terminates.
4.3 Simplifications of the Formula

Now, consider a similar formula in which the predicates \( x_1 \neq x_2 \) and \( u_1 \neq f_1 \) are swapped. This results in the formula

\[
\varphi^E := \begin{cases} 
(u_1 \neq f_1 \lor y_1 \neq y_2 \lor f_1 = f_2) \land \\
(x_1 \neq x_2 \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \land z = g_1 \land z \neq g_2)
\end{cases}.
\] (4.11)

Although we start from exactly the same graph, the simplification algorithm is now much more effective. After the first step we have

\[
\varphi^{E_1} := \begin{cases} 
(u_1 \neq f_1 \lor \text{TRUE} \lor \text{TRUE}) \land \\
(\text{TRUE} \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \land z = g_1 \land z \neq g_2)
\end{cases},
\] (4.12)

which, after step 4, simplifies to

\[
\varphi^{E_2} := \begin{cases} 
(u_1 = f_1 \land u_2 = f_2 \land z = g_1 \land z \neq g_2)
\end{cases}.
\] (4.13)

The graph corresponding to \( \varphi^{E_2} \) after this step appears in Fig. 4.3.

![Fig. 4.3. An equality graph corresponding to (4.13), showing the first iteration of step 4](image)

Clearly, no edges in \( \varphi^{E_2} \) belong to a contradictory cycle after this step, which implies that we can replace all the remaining predicates by \( \text{TRUE} \). Hence, in this case, simplification alone proves that the formula is satisfiable, without invoking a decision procedure.

Although we leave the formal proof of the correctness of Algorithm 4.3.1 as an exercise (Problem 4.5), let us now consider what such a proof may look like. Correctness can be shown by proving that steps 3 and 4 maintain satisfiability (as these are the only steps in which the formula is changed). The simplifications in step 4 trivially maintain satisfiability, so the main problem is step 3.

Let \( \varphi_1^E \) and \( \varphi_2^E \) be the equality formulas before and after step 3, respectively. We need to show that these formulas are equisatisfiable.

\(\Rightarrow\) If \( \varphi_1^E \) is satisfiable, then so is \( \varphi_2^E \). This is implied by the monotonicity of NNF formulas (see Theorem 1.14) and the fact that only pure literals are replaced by \( \text{TRUE} \).
If $\varphi^E_2$ is satisfiable, then so is $\varphi^E_1$. Only a proof sketch and an example will be given here. The idea is to construct a satisfying assignment $\alpha_1$ for $\varphi^E_1$ while relying on the existence of a satisfying assignment $\alpha_2$ for $\varphi^E_2$. Specifically, $\alpha_1$ should satisfy exactly the same predicates as are satisfied by $\alpha_2$, but also satisfy all those predicates that were replaced by TRUE. The following simple observation can be helpful in this construction: given a satisfying assignment to an equality formula, shifting the values in the assignment uniformly maintains satisfaction (because the values of the equality predicates remain the same). The same observation applies to an assignment of some of the variables, as long as none of the predicates that refer to one of these variables becomes FALSE owing to the new assignment.

Consider, for example, (4.11) and (4.12), which correspond to $\varphi^E_1$ and $\varphi^E_2$, respectively, in our argument. An example of a satisfying assignment to the latter is

$$\alpha_2 := \{u_1 \mapsto 0, f_1 \mapsto 0, f_2 \mapsto 1, u_2 \mapsto 1, z \mapsto 0, g_1 \mapsto 0, g_2 \mapsto 1\}.$$ (4.14)

First, $\alpha_1$ is set equal to $\alpha_2$. Second, we need to extend $\alpha_1$ with an assignment of those variables not assigned by $\alpha_2$. The variables in this category are $x_1, x_2, y_1,$ and $y_2$, which can be trivially satisfied because they are not part of any equality predicate. Hence, assigning a unique value to each of them is sufficient. For example, we can now have

$$\alpha_1 := \alpha_1 \cup \{x_1 \mapsto 2, x_2 \mapsto 3, y_1 \mapsto 4, y_2 \mapsto 5\}.$$ (4.15)

Third, we need to consider predicates that are replaced by TRUE in step 3 but are not satisfied by $\alpha_1$. In our example, $f_1 = f_2$ is such a predicate. To solve this problem, we simply shift the assignment to $f_2$ and $u_2$ so that the predicate $f_1 = f_2$ is satisfied (a shift by minus 1 in this case). This clearly maintains the satisfaction of the predicate $u_2 = f_2$. The assignment that satisfies $\varphi^E_1$ is thus

$$\alpha_1 := \{u_1 \mapsto 0, f_1 \mapsto 0, f_2 \mapsto 0, u_2 \mapsto 0, z \mapsto 0, g_1 \mapsto 0, g_2 \mapsto 1, x_1 \mapsto 2, x_2 \mapsto 3, y_1 \mapsto 4, y_2 \mapsto 5\}.$$ (4.16)

A formal proof based on this argument should include a precise definition of these shifts, i.e., which vertices do they apply to, and an argument as to why no circularity can occur. Circularity can affect the termination of the procedure that constructs $\alpha_1$.

### 4.4 A Graph-Based Reduction to Propositional Logic

We now consider a decision procedure for equality logic that is based on a reduction to propositional logic. This procedure was originally presented by Bryant and Velev in [39](under the name of the sparse method). Several definitions and observations are necessary.
Definition 4.9 (nonpolar equality graph). Given an equality logic formula $\varphi^E$, the nonpolar equality graph corresponding to $\varphi^E$, denoted by $G_{NP}^E(\varphi^E)$, is an undirected graph $(V, E)$ where the nodes in $V$ correspond to the variables in $\varphi^E$, and the edges in $E$ correspond to $At(\varphi^E)$, i.e., the equality predicates in $\varphi^E$.

A nonpolar equality graph represents a degenerate version of an equality graph (Definition 4.4), since it disregards the polarity of the equality predicates.

Given an equality logic formula $\varphi^E$, the procedure generates two propositional formulas $e(\varphi^E)$ and $B_{trans}$, such that

$$\varphi^E \text{ is satisfiable } \iff e(\varphi^E) \land B_{trans} \text{ is satisfiable.}$$

(4.17)

The formulas $e(\varphi^E)$ and $B_{trans}$ are defined as follows:

- The formula $e(\varphi^E)$ is the **propositional skeleton** of $\varphi^E$, which means that every equality predicate of the form $x_i = x_j$ in $\varphi^E$ is replaced with a new Boolean variable $e_{i,j}$.\(^1\)

  For example, let

  $$\varphi^E := x_1 = x_2 \land ((x_2 = x_3) \land (x_1 \neq x_3)) \lor (x_1 \neq x_2).$$

  Then,

  $$e(\varphi^E) := e_{1,2} \land ((e_{2,3} \land \neg e_{1,3}) \lor \neg e_{1,2}).$$

  (4.19)

  It is not hard to see that if $\varphi^E$ is satisfiable, then so is $e(\varphi^E)$. The other direction, however, does not hold. For example, while (4.18) is unsatisfiable, its encoding in (4.19) is satisfiable. To maintain an equisatisfiability relation, we need to add constraints that impose the transitivity of equality, which was lost in the encoding. This is the role of $B_{trans}$.

- The formula $B_{trans}$ is a conjunction of implications, which are called **transitivity constraints**. Each such implication is associated with a cycle in the nonpolar equality graph. For a cycle with $n$ edges, $B_{trans}$ forbids an assignment FALSE to one of the edges when all the other edges are assigned TRUE. Imposing this constraint for each of the edges in each one of the cycles is sufficient to satisfy the condition stated in (4.17).

**Example 4.10.** The atoms $x_1 = x_2, x_2 = x_3, x_1 = x_3$ form a cycle of size 3 in the nonpolar equality graph. The following constraint is sufficient for maintaining the condition stated in (4.17):

$$B_{trans} = \left( \begin{array}{c} (e_{1,2} \land e_{2,3} \implies e_{1,3}) \land \\ (e_{1,2} \land e_{1,3} \implies e_{2,3}) \land \\ (e_{2,3} \land e_{1,3} \implies e_{1,2}) \end{array} \right).$$

(4.20)

\(^1\) To avoid introducing dual variables such as $e_{i,j}$ and $e_{j,i}$, we can assume that all equality predicates in $\varphi^E$ appear in such a way that the left variable precedes the right one in some predefined order.
Adding \( n \) constraints for each cycle is not very practical, however, because there can be an exponential number of cycles in a given undirected graph.

**Definition 4.11 (chord).** A chord of a cycle is an edge connecting two non-adjacent nodes of the cycle. If a cycle has no chords in a given graph, it is called a chord-free cycle.

Bryant and Velev proved the following theorem:

**Theorem 4.12.** It is sufficient to add transitivity constraints over simple chord-free cycles in order to maintain (4.17).

For a formal proof, see [39]. The following example may be helpful for developing an intuition as to why this theorem is correct.

**Example 4.13.** Consider the cycle \((x_3, x_4, x_8, x_7)\) in one of the two graphs in Fig. 4.4. It contains the chord \((x_3, x_8)\) and, hence, is not chord-free. Now assume that we wish to assign \textsc{true} to all edges in this cycle other than \((x_3, x_4)\). If \((x_3, x_8)\) is assigned \textsc{true}, then the assignment to the simple chord-free cycle \((x_3, x_4, x_8)\) contradicts transitivity. If \((x_3, x_8)\) is assigned \textsc{false}, then the assignment to the simple chord-free cycle \((x_3, x_7, x_8)\) contradicts transitivity. Thus, the constraints over the chord-free cycles are sufficient for preventing the transitivity-violating assignment to the cycle that includes a chord.

The number of simple chord-free cycles in a graph can still be exponential in the number of vertices. Hence, building \( \mathcal{B}_{\text{trans}} \) such that it directly constrains every such cycle can make the size of this formula exponential in the number of variables. Luckily, we have:

**Definition 4.14 (chordal graphs).** A chordal graph is an undirected graph in which no cycle of size 4 or more is chord-free.

Every graph can be made chordal in a time polynomial in the number of vertices.\(^2\) Since the only chord-free cycles in a chordal graph are triangles, this implies that applying Theorem 4.12 to such a graph results in a formula of size not more than cubic in the number of variables (three constraints for each triangle in the graph). The newly added chords are represented by new variables that appear in \( \mathcal{B}_{\text{trans}} \) but not in \( \mathcal{e}(\varphi^E) \). Algorithm 4.4.1 summarizes the steps of this method.

**Example 4.15.** Figure 4.4 depicts a nonpolar equality graph before and after making it chordal. We use solid edges, but note that these should not be confused with the solid edges in (polar) equality graphs, where they denote

---

\(^2\) We simply remove all vertices from the graph one by one, each time connecting the neighbors of the eliminated vertex if they were not already connected. The original graph plus the edges added in this process is a chordal graph.
4.4 A Graph-Based Reduction to Propositional Logic

Fig. 4.4. A nonchordal nonpolar equality graph corresponding to $\varphi^E$ (left), and a possible chordal version of it (right)

Algorithm 4.4.1: EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC

**Input:** An equality formula $\varphi^E$

**Output:** A propositional formula equisatisfiable with $\varphi^E$

1. Construct a Boolean formula $e(\varphi^E)$ by replacing each atom of the form $x_i = x_j$ in $\varphi^E$ with a Boolean variable $e_{i,j}$.
2. Construct the nonpolar equality graph $G_{NP}^E(\varphi^E)$.
3. Make $G_{NP}^E(\varphi^E)$ chordal.
4. $B_{trans} :=$ TRUE.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{i,k})$ in $G_{NP}^E(\varphi^E)$,
   \[
   B_{trans} := B_{trans} \land \\
   (e_{i,j} \land e_{j,k} \implies e_{i,k}) \land \\
   (e_{i,j} \land e_{i,k} \implies e_{j,k}) \land \\
   (e_{i,k} \land e_{j,k} \implies e_{i,j}).
   \] (4.21)
6. Return $e(\varphi^E) \land B_{trans}$.

The added edge $e_{2,5}$ corresponds to a new auxiliary variable $e_{2,5}$ that appears in $B_{trans}$ but not in $e(\varphi^E)$.

There exists a version of this algorithm that is based on the (polar) equality graph, and generates a smaller number of transitivity constraints. See Problem 4.6 for more details.
4.5 Equalities and Small-Domain Instantiations

In this section, we show a method for solving equality logic formulas by relying on the small-model property that this logic has. This means that every satisfiable formula in this logic has a model (a satisfying interpretation) of finite size. Furthermore, in equality logic there is a computable bound on the size of such a model. We use the following definitions in the rest of the discussion.

Definition 4.16 (adequacy of a domain for a formula). A domain is adequate for a formula if the formula either is unsatisfiable or has a model within this domain.

Definition 4.17 (adequacy of a domain for a set of formulas). A domain is adequate for a set of formulas if it is adequate for each formula in the set.

In the case of equality logic, each set of formulas with the same number of variables has an easily computable adequate finite domain, as we shall soon see. The existence of such a domain immediately suggests a decision procedure: simply enumerate all assignments within this domain and check whether one of them satisfies the formula. Our solution strategy, therefore, for checking whether a given equality formula \( \varphi^E \) is satisfiable, can be summarized as follows:

1. Determine, in polynomial time, a domain allocation

\[
D : \text{var}(\varphi^E) \mapsto 2^N
\]  

(4.23)

(where \( \text{var}(\varphi^E) \) denotes the set of variables of \( \varphi^E \)), by mapping each variable \( x_i \in \text{var}(\varphi^E) \) into a finite set of integers \( D(x_i) \), such that \( \varphi^E \) is satisfiable if and only if it is satisfiable within \( D \) (i.e., there exists a satisfying assignment in which each variable \( x_i \) is assigned an integer from \( D(x_i) \)).

2. Encode each variable \( x_i \) as an enumerated type over its finite domain \( D(x_i) \). Construct a propositional formula representing \( \varphi^E \) under this finite domain, and use either BDDs or SAT to check if this formula is satisfiable.

This strategy is called small-domain instantiation, since we instantiate the variables with a finite set of values from the domain computed, each time checking whether it satisfies the formula. The number of instantiations in the worst case is what we call the size of the state space spanned by a domain. The size of the state space of a domain \( D \), denoted by \( |D| \) is equal to the product of the numbers of elements in the domains of the individual variables. Clearly, the success of this method depends on its ability to find domain allocations with small state spaces.
4.5 Equalities and Small-Domain Instantiations

4.5.1 Some Simple Bounds

We now show several bounds on the number of elements in an adequate domain. Let $\Phi_n$ be the (infinite) set of all equality logic formulas with $n$ variables and without constants.

**Theorem 4.18 ("folk theorem").** The uniform domain allocation $\{1, \ldots, n\}$ for all $n$ variables is adequate for $\Phi_n$.

**Proof.** Let $\varphi^E \in \Phi_n$ be a satisfiable equality logic formula. Every satisfying assignment $\alpha$ to $\varphi^E$ reflects a partition of its variables into equivalence classes. That is, two variables are in the same equivalence class if and only if they are assigned the same value by $\alpha$. Since there are only equalities and disequalities in $\varphi^E$, every assignment which reflects the same equivalence classes satisfies exactly the same predicates as $\alpha$. Since all partitions into equivalence classes over $n$ variables are possible in the domain $1, \ldots, n$, this domain is adequate for $\varphi^E$.

This bound, although not yet tight, implies that we can encode each variable in a $\Phi_n$ formula with no more than $\lceil \log n \rceil$ bits, and with a total of $n \lceil \log n \rceil$ bits for the entire formula in the worst case. This is very encouraging, because it is already better than the worst-case complexity of Algorithm 4.4.1, which requires $n \cdot (n - 1)/2$ bits (one bit per pair of variables) in the worst case.

**Aside: The Complexity Gap**

Why is there a complexity gap between domain allocation and the encoding method that we described in Sect. 4.4? Where is the wasted work in \textsc{Equality-Logic-to-Propositional-Logic}? Both algorithms merely partition the variables into classes of equal variables, but they do it in a different way. Instead of asking ‘which subset of $\{v_1, \ldots, v_n\}$ is each variable equal to?’, with the domain-allocation technique we ask instead ‘which value in the range $\{1, \ldots, n\}$ is each variable equal to?’. For each variable, rather than exploring the range of subsets of $\{v_1, \ldots, v_n\}$ to which it may be equal, we instead explore the range of values $\{1, \ldots, n\}$. The former requires one bit per element in this set, or a total of $n$ bits, while the latter requires only $\log n$ bits.

The domain $1, \ldots, n$, as suggested above, results in a state space of size $n^n$. We can do better if we do not insist on a uniform domain allocation, which allocates the same domain to all variables.

**Theorem 4.19.** Assume for each formula $\varphi^E \in \Phi_n$, $\text{var}(\varphi^E) = \{x_1, \ldots, x_n\}$. The domain allocation $D := \{x_i \mapsto \{1, \ldots, i\} \mid 1 \leq i \leq n\}$ is adequate for $\Phi_n$. 


Proof. As argued in the proof of Theorem 4.18, every satisfying assignment \( \alpha \) to \( \varphi^E \in \Phi_n \) reflects a partition of the variables to equivalence classes. We construct an assignment \( \alpha' \) as follows.

For each equivalence class \( C \):

- Let \( x_i \) be the variable with the lowest index in \( C \).
- Assign \( i \) to all the variables in \( C \).

Since all the other variables in \( C \) have indices higher than \( i \), \( i \) is in their domain, and hence this assignment is feasible. Since each variable appears in exactly one equivalence class, every class of variables is assigned a different value, which means that \( \alpha' \) satisfies the same equality predicates as \( \alpha \). This implies that \( \alpha' \) satisfies \( \varphi^E \).

The adequate domain suggested in Theorem 4.19 has a smaller state space, of size \( n! \). In fact, it is conjectured that \( n! \) is also a lower bound on the size of domain allocations adequate for this class of formulas.

Let us now consider the case in which the formula contains constants.

**Theorem 4.20.** Let \( \Phi_{n,k} \) be the set of equality logic formulas with \( n \) variables and \( k \) constants. Assume, without loss of generality, that the constants are \( c_1 < \cdots < c_k \). The domain allocation

\[
D := \{ x_i \mapsto \{ c_1, \ldots, c_k, c_k+1, \ldots, c_k+i \} \mid 1 \leq i \leq n \} \tag{4.24}
\]

is adequate for \( \Phi_{n,k} \).

The proof is left as an exercise (Problem 4.8).

The adequate domain suggested in Theorem 4.20 results in a state space of size \((k + n)!/k!\). As stated in Sect. 3.1.3, constants can be eliminated by adding more variables and constraints (\( k \) variables in this case), but note that this would result in a larger state space.

The next few sections are dedicated to an algorithm that reduces the allocated domain further, based on an analysis of the equality graph associated with the input formula.

Sects. 4.5.2, 4.5.3, and 4.5.4 cover advanced topics.

### 4.5.2 Graph-Based Domain Allocation

The formula sets \( \Phi_n \) and \( \Phi_{n,k} \) utilize only a simple structural characteristic common to all of their members, namely, the number of variables and constants. As a result, they group together many formulas of radically different nature. It is not surprising that the best size of adequate domain allocation for the whole set is so high. By paying attention to additional structural similarities of formulas, we can form smaller sets of formulas and obtain much smaller adequate domain allocations.
As before, we assume that \( \varphi^E \) is given in negation normal form. Let \( e \) denote a set of equality literals and \( \Phi(e) \) the set of all equality logic formulas whose literals set is equal to \( e \). Let \( E(\varphi^E) \) denote the set of \( \varphi^E \)'s literals. Thus, \( \Phi(E(\varphi^E)) \) is the set of all equality logic formulas that have the same set of literals as \( \varphi^E \). Obviously, \( \varphi^E \in \Phi(E(\varphi^E)) \). Note that \( \Phi(e) \) can include both satisfiable and unsatisfiable formulas. For example, let \( e \) be the set

\[
\{x_1 = x_2, x_1 \neq x_2\}.
\]

Then \( \Phi(e) \) includes both the satisfiable formula

\[
x_1 = x_2 \lor x_1 \neq x_2
\]

and the unsatisfiable formula

\[
x_1 = x_2 \land x_1 \neq x_2.
\]

An adequate domain, recall, is concerned only with the satisfiable formulas that can be constructed from literals in the set. Thus, we should not worry about (4.27). We should, however, be able to satisfy (4.26), as well as formulas such as \( x_1 = x_2 \land (\text{true} \lor x_1 \neq x_2) \) and \( x_1 \neq x_2 \land (\text{true} \lor x_1 = x_2) \). One adequate domain for the set \( \Phi(e) \) is

\[
D := \{x_1 \mapsto \{0\}, x_2 \mapsto \{0, 1\}\}.
\]

It is not hard to see that this domain is minimal, i.e., there is no adequate domain with a state space smaller than 2 for \( \Phi(e) \).

How do we know, then, which subsets of the literals in \( E(\varphi^E) \) we need to be able to satisfy within the domain \( D \), in order for \( D \) to be adequate for \( \Phi(E(\varphi^E)) \)? The answer is that we need only to be able to satisfy consistent subsets of literals, i.e., subsets for which the conjunction of literals in each of them is satisfiable.

A set \( e \) of equality literals is consistent if and only if it does not contain one of the following two patterns:

1. A chain of the form \( x_1 = x_2, x_2 = x_3, \ldots, x_{r-1} = x_r \) together with the formula \( x_1 \neq x_r \).
2. A chain of the form \( c_1 = x_2, x_2 = x_3, \ldots, x_{r-1} = c_r \) where \( c_1 \) and \( c_r \) represent different constants.

In the equality graph corresponding to \( e \), the first pattern appears as a contradictory cycle (Definition 4.7) and the second as an equality path (Definition 4.5) between two constants.

To summarize, a domain allocation \( D \) is adequate for \( \Phi(E(\varphi^E)) \) if every consistent subset \( e \subseteq E(\varphi^E) \) is satisfiable within \( D \). Hence, finding an adequate domain for \( \Phi(E(\varphi^E)) \) is reduced to the following problem:

Associate with each variable \( x_i \) a set of integers \( D(x_i) \) such that every consistent subset \( e \in E(\varphi^E) \) can be satisfied with an assignment from these sets.
We wish to find sets of this kind that are as small as possible, in polynomial time.

### 4.5.3 The Domain Allocation Algorithm

Let $G_E^E(\varphi^E)$ be the equality graph (see Definition 4.4) corresponding to $\varphi^E$, defined by $G_E^E$ and $G_E^{\neq}$ denote two subgraphs of $G_E^E(\varphi^E)$, defined by $(V, E_\neq)$ and $(V, E_\neq)$, respectively. As before, we use dashed edges to represent $G_E^E$ edges and solid edges to represent $G_E^{\neq}$ edges. A vertex is called a mixed if it is adjacent to edges in both $G_E^E$ and $G_E^{\neq}$.

On the basis of the definitions above, Algorithm 4.5.1 computes an economical domain allocation $D$ for the variables in a given equality formula $\varphi^E$. The algorithm receives as input the equality graph $G_E^E(\varphi^E)$, and returns as output a domain which is adequate for the set $\Phi(E(\varphi^E))$. Since $\varphi^E \in \Phi(E(\varphi^E))$, this domain is adequate for $\varphi^E$.

We refer to the values that were added in steps I.A.2, I.C, II.A.1, and II.B as the characteristic values of these vertices. We write $\text{char}(x_i) = u_i$ and $\text{char}(x_k) = u_{C_{\text{max}}}$. Note that every vertex is assigned a single characteristic value. Vertices that are assigned their characteristic values in steps I.C and II.A.1 are called individually assigned vertices, whereas the vertices assigned characteristic values in step II.B are called communally assigned vertices. We assume that new values are assigned in ascending order, so that $\text{char}(x_i) < \text{char}(x_j)$ implies that $x_i$ was assigned its characteristic value before $x_j$. Consequently, we require that all new values are larger than the largest constant $C_{\text{max}}$. This assumption is necessary only for simplifying the proof in later sections.

The description of the algorithm presented above leaves open the order in which vertices are chosen in step II.A.1. This order has a strong impact on the size of the resulting state space. Since the values given in this step are distributed on the graph $G_E^E$ in step II.A.2, we would like to keep this set as small as possible. Furthermore, we would like to partition the graph quickly, in order to limit this distribution. A rather simple, but effective heuristic for this purpose is to choose vertices according to a greedy criterion, where mixed vertices are chosen in descending order of their degree in $G_E^{\neq}$.

We denote the set of vertices chosen in step II.A.1 by $MV$, to remind ourselves that they are mixed vertices.

**Example 4.21.** We wish to check whether (4.6), copied below, is satisfiable:

$$\neg \varphi^E := \left( (x_1 \neq x_2 \lor y_1 \neq y_2 \lor f_1 = f_2) \land (u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land u_1 = f_1 \land u_2 = f_2 \land z = g_1 \right) \land z \neq g_2. \quad (4.29)$$

The sets $E_\neq$ and $E_\neq$ are:

$$E_\neq := \{(f_1 = f_2), (g_1 = g_2), (u_1 = f_1), (u_2 = f_2), (z = g_1)\}$$

$$E_\neq := \{(x_1 \neq x_2), (y_1 \neq y_2), (u_1 \neq f_1), (u_2 \neq f_2), (z \neq g_2)\}. \quad (4.30)$$
Algorithm 4.5.1: Domain-Allocation-for-Equalities

**Input:** An equality graph $G^E$

**Output:** An adequate domain (in the form of a set of integers for each variable-vertex) for the set of formulas over literals that are represented by $G^E$ edges

I. Eliminating constants and preprocessing
Initially, $D(x_i) = \emptyset$ for all vertices $x_i \in G^E$.

A. For each constant-vertex $c_i$ in $G^E$, do:
   1. (Empty item, for the sake of symmetry with step II.A.)
   2. Assign $D(x_j) := D(x_j) \cup \{c_i\}$ for each vertex $x_j$, such that there is an equality path from $c_i$ to $x_j$ not through any other constant-vertex.
   3. Remove $c_i$ and its adjacent edges from the graph.

B. Remove all $G^E_{\neq}$ edges that do not lie on a contradictory cycle.

C. For every singleton vertex (a vertex comprising a connected component by itself) $x_i$, add to $D(x_i)$ a new value $u_i$. Remove $x_i$ and its adjacent edges from the graph.

II. Value allocation

A. While there are mixed vertices in $G^E$ do:
   1. Choose a mixed vertex $x_i$. Add $u_i$, a new value, to $D(x_i)$.
   2. Assign $D(x_j) := D(x_j) \cup \{u_i\}$ for each vertex $x_j$, such that there is an equality path from $x_i$ to $x_j$.
   3. Remove $x_i$ and its adjacent edges from the graph.

B. For each (remaining) connected $G^E_{=} = C_=$ component $C_=$, add a common new value $u_{C=} = D(x_k)$, for every $x_k \in C_=$.

Return $D$.

and the corresponding equality graph $G^E(\neg \varphi^E)$ reappears in Fig. 4.5.

**Fig. 4.5.** The equality graph $G^E(\neg \varphi^E)$
We refrain in this example from applying preprocessing, in order to make the demonstration of the algorithm more informative and interesting. This example results in a state space of size \(11^{11}\) if we use the domain \(\{1, \ldots, n\}\) as suggested in Theorem 4.18, and a state space of size \(11! \approx 4 \times 10^7\) if we use the domain suggested in Theorem 4.19. Applying Algorithm 4.5.1, on the other hand, results in an adequate domain spanning a state space of size 48, as can be seen in Fig. 4.6.

<table>
<thead>
<tr>
<th>Step</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(y_1)</th>
<th>(y_2)</th>
<th>(u_1)</th>
<th>(f_1)</th>
<th>(u_2)</th>
<th>(g_2)</th>
<th>(z)</th>
<th>(g_1)</th>
<th>Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>edges ((x_1 - x_2),) ((y_1 - y_2))</td>
</tr>
<tr>
<td>I.C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x_1, x_2, y_1, y_2)</td>
</tr>
<tr>
<td>II.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td>(f_1)</td>
</tr>
<tr>
<td>II.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(f_2)</td>
</tr>
<tr>
<td>II.A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
<td>(g_2)</td>
</tr>
<tr>
<td>II.B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II.B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II.B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>State space (= 48)</td>
</tr>
<tr>
<td>(D)-sets</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4, 7</td>
<td>4</td>
<td>4, 5</td>
<td>4, 5, 8</td>
<td>6</td>
<td>6, 9</td>
<td>6, 9</td>
</tr>
</tbody>
</table>

**Fig. 4.6.** Application of Algorithm 4.5.1 to (4.29)

Using a small improvement concerning the new values allocated in step II.A.1, this allocation can be reduced further, down to a domain of size 16. This improvement is the subject of Problem 4.12.

For demonstration purposes, consider a formula \(\varphi^*\) where \(g_1\) is replaced by the constant “3”. In this case the component \((z, g_1, g_2)\) is handled as follows: in step I.A, “3” is added to \(D(g_2)\) and \(D(z)\). The edge \((z, g_2)\), now no longer part of a contradictory cycle, is then removed in step I.B and a distinct new value is added to each of these variables in step I.C.

Algorithm 4.5.1 is polynomial in the size of the input graph: steps I.A and II.A are iterated a number of times not more than the number of vertices in the graph; step I.B is iterated not more than the number of edges in \(G^G_{\neq, p}\); steps I.A.2, I.B, II.A.2 and II.B can be implemented with depth-first search (DFS).

### 4.5.4 A Proof of Soundness

In this section, we argue for the soundness of Algorithm 4.5.1. We begin by describing a procedure which, given the allocation \(D\) produced by this algorithm
and a consistent subset $e$, assigns to each variable $x_i \in G^e$ an integer value $a_e(x_i) \in D(x_i)$. We then continue by proving that this assignment satisfies the literals in $e$.

**An Assignment Procedure**

Given a consistent subset of literals $e$ and its corresponding equality graph $G^e(e)$, assign to each variable-vertex $x_i \in G^e(e)$ a value $a_e(x_i) \in D(x_i)$, according to the following rules:

- **R1** If $x_i$ is connected by a (possibly empty) $G^e(e)$-path to an individually assigned vertex $x_j$, assign to $x_i$ the minimal value of $\text{char}(x_j)$ among such $x_j$’s.
- **R2** Otherwise, assign to $x_i$ its communally assigned value $\text{char}(x_i)$.

To see why all vertices are assigned a value by this procedure, observe that every vertex is allocated a characteristic value before it is removed. This can be an individual characteristic value allocated in steps I.C and II.A.1, or a communal value allocated in step II.B. Every vertex $x_i$ that has an individual characteristic value can be assigned a value $a_e(x_i)$ by **R1**, because it has at least the empty equality path leading to an individually allocated vertex, namely itself. All other vertices are allocated a communal value that makes them eligible for a value assignment by **R2**.

**Example 4.22.** Consider the $D$-sets in Fig. 4.6. Let us apply the above assignment procedure to a consistent subset $e$ that contains all edges, excluding the two edges between $u_1$ and $f_1$, the dashed edge between $g_1$ and $g_2$, and the solid edge between $f_2$ and $u_2$ (see Fig. 4.7).

![Fig. 4.7](image)

The assignment is as follows:

- By **R1**, $x_1, x_2, y_1$ and $y_2$ are assigned the characteristic values “0”, “1”, “2”, and “3”, respectively, which they received in step I.C.
- By **R1**, $f_1, f_2$ and $u_2$ are assigned the value $\text{char}(f_1) = “4”$, because $f_1$ was the first mixed vertex in the subgraph $\{f_1, f_2, u_2\}$ that was removed in step II.A, and consequently it has the minimal characteristic value.
• By $R1$, $g_2$ is assigned the value $\text{char}(g_2) = “6”$, which it received in step II.A.

• By $R2$, $z$ and $g_1$ are assigned the value “9”, which they received in step II.B.

• By $R2$, $u_1$ is assigned the value “7”, which it received in step II.B.

Theorem 4.23. The assignment procedure is feasible (i.e., the value assigned to a node by the procedure belongs to its $D$-set).

Proof. Consider first the two classes of vertices that are assigned a value by $R1$. The first class includes vertices that are removed in step I.C. These vertices have only one (empty) $G^e_e(e)$-path to themselves, and are therefore assigned the characteristic value that they received in that step. The second class includes vertices that have a (possibly empty) $G^e_e(e)$-path to a vertex from $MV$. Let $x_i$ denote such a vertex, and let $x_j$ be the vertex with the minimal characteristic value that $x_i$ can reach on $G^e_e(e)$. Since $x_i$ and all the vertices on this path were still part of the graph when $x_j$ was removed in step II.A, then $\text{char}(x_j)$ was added to $D(x_i)$ according to step II.A.2. Thus, the assignment of $\text{char}(x_j)$ to $x_i$ is feasible.

Next, consider the vertices that are assigned a value by $R2$. Every vertex that was removed in step I.C or II.A is clearly assigned a value by $R1$. All the other vertices were communally assigned a value in step II.B. In particular, the vertices that do not have a path to an individually assigned vertex were assigned such a value. Thus, the two steps of the assignment procedure are feasible.

Theorem 4.24. If $e$ is a consistent set, then the assignment $a_e$ satisfies all the literals in $e$.

Proof. Consider first the case of two variables $x_i$ and $x_j$ that are connected by a $G^e_e(e)$-edge. We have to show that $a_e(x_i) = a_e(x_j)$. Since $x_i$ and $x_j$ are $G^e_e(e)$-connected, they belong to the same $G^e_e(e)$-connected component. If they were both assigned a value by $R1$, then they were assigned the minimal value of an individually assigned vertex to which they are both $G^e_e(e)$-connected. If, on the other hand, they were both assigned a value by $R2$, then they were assigned the communal value assigned to the $G^e_e$ component to which they both belong. Thus, in both cases they are assigned the same value.

Next, consider the case of two variables $x_i$ and $x_j$ that are connected by a $G^e_{\not{=}e}(e)$-edge. To show that $a_e(x_i) \neq a_e(x_j)$, we distinguish three cases:

• If both $x_i$ and $x_j$ were assigned values by $R1$, they must have inherited their values from two distinct individually assigned vertices, because, otherwise, they are both connected by a $G^e_e(e)$-path to a common vertex, which together with the $(x_i, x_j)$ $G^e_{\not{=}e}(e)$-edge closes a contradictory cycle, excluded by the assumption that $e$ is consistent.
• If one of $x_i$, $x_j$ was assigned a value by $R1$ and the other acquired its value from $R2$, then since any communal value is distinct from any individually assigned value, $a_e(x_i)$ must differ from $a_e(x_j)$.

• The remaining case is when both $x_i$ and $x_j$ were assigned values by $R2$. The fact that they were not assigned values in $R1$ implies that their characteristic values are not individually allocated, but communally allocated. Assume falsely that $a_e(x_i) = a_e(x_j)$. This means that $x_i$ and $x_j$ were allocated their communal values in the same step, II.B, of the allocation algorithm, which implies that they had an equality path between them (moreover, this path was still part of the graph at the beginning of step II.B). Hence, $x_i$ and $x_j$ belong to a contradictory cycle, and the solid edge $(x_i, x_j)$ was therefore still part of $G_E(e)$ at the beginning of step II.A. According to the loop condition of this step, at the end of this step there are no mixed vertices left, which rules out the possibility that $(x_i, x_j)$ was still part of the graph at that stage. Thus, at least one of these vertices was individually assigned a value in step II.A.1, and, consequently, the component that it belongs to is assigned a value by $R1$, in contradiction to our assumption.

**Theorem 4.25.** The formula $\varphi^E$ is satisfiable if and only if $\varphi^E$ is satisfiable over $D$.

**Proof.** By Theorems 4.23 and 4.24, $D$ is adequate for $E_\varphi \cup E_\chi$. Consequently, $D$ is adequate for $\Phi(At(\varphi^E))$, and in particular $D$ is adequate for $\varphi^E$. Thus, by the definition of adequacy, $\varphi^E$ is satisfiable if and only if $\varphi^E$ is satisfiable over $D$.

**4.5.5 Summary**

To summarize Sect. 4.5, the domain allocation method can be used as the first stage of a decision procedure for equality logic. In the second stage, the allocated domains can be enumerated by a standard BDD or by a SAT-based tool. Domain allocation has the advantage of not changing (in particular, not increasing) the original formula, unlike the algorithm that we studied in Sect. 4.4. Moreover, Algorithm 4.5.1 is highly effective in practice in allocating very small domains.

**4.6 Ackermann’s vs. Bryant’s Reduction: Where Does It Matter?**

We conclude this chapter by demonstrating how the two reductions lead to different equality graphs and hence change the result of applying any of the algorithms studied in this chapter that are based on this equality graph.
Example 4.26. Suppose that we want to check the satisfiability of the following (satisfiable) formula:

\[ \varphi_{\text{UF}} := x_1 = x_2 \lor (F(x_1) \neq F(x_2) \land \text{FALSE}) \]  

(4.31)

With Ackermann’s reduction, we obtain:

\[ \varphi^E := (x_1 = x_2 \implies f_1 = f_2) \land (x_1 = x_2 \lor (f_1 \neq f_2 \land \text{FALSE})) \]  

(4.32)

With Bryant’s reduction, we obtain:

\[ \text{flat}^E := x_1 = x_2 \lor (F_1^* \neq F_2^* \land \text{FALSE}) , \]  

\[ \text{FC}^E := F_1^* = f_1 \land F_2^* = \left( \begin{array}{c} \text{case } x_1 = x_2 : f_1 \\ \text{TRUE} : f_2 \end{array} \right) , \]  

(4.33)

\[ \text{and, as always,} \]  

\[ \varphi^E := \text{FC}^E \land \text{flat}^E . \]  

(4.34)

The equality graphs corresponding to the two reductions appear in Fig. 4.8. Clearly, the allocation for the right graph (due to Bryant’s reduction) is smaller.

Indeed, an adequate range for the graph on the right is

\[ D := \{ x_1 \mapsto \{0\}, x_2 \mapsto \{0, 1\}, f_1 \mapsto \{2\}, f_2 \mapsto \{3\} \} . \]  

(4.35)

These domains are adequate for (4.35), since we can choose the satisfying assignment

\[ \{ x_1 \mapsto 0, x_2 \mapsto 0, f_1 \mapsto 2, f_2 \mapsto 3 \} . \]  

(4.36)

On the other hand, this domain is not adequate for (4.32).

In order to satisfy (4.32), it must hold that \( x_1 = x_2 \), which implies that \( f_1 = f_2 \) must hold as well. But the domains allocated in (4.36) do not allow an assignment in which \( f_1 \) is equal to \( f_2 \), which means that the graph on the right of Fig. 4.8 is not adequate for (4.32).
So what has happened here? Why does Ackermann’s reduction require a larger range?

The reason is that when two function instances $F(x_1)$ and $F(x_2)$ have equal arguments, in Ackermann’s reduction the two variables representing the functions, say $f_1$ and $f_2$, are constrained to be equal. But if we force $f_1$ and $f_2$ to be different (by giving them a singleton domain composed of a unique constant), this forces $FC^E$ to be false, and, consequently $\varphi^E$ to be false. On the other hand, in Bryant’s reduction, if the arguments $x_1$ and $x_2$ are equal, the terms $F_1^*$ and $F_2^*$ that represent the two functions are both assigned the value of $f_1$. Thus, even if $f_2 \neq f_1$, this does not necessarily make $FC^E$ false.

In the bibliographic notes of this chapter, we mention several publications that exploit this property of Bryant’s reduction for reducing the allocated range and even constructing smaller equality graphs. It turns out that not all of the edges that are associated with the functional-consistency constraints are necessary, which, in turn, results in a smaller allocated range.

4.7 Problems

4.7.1 Conjunctions of Equalities and Uninterpreted Functions

Problem 4.1 (deciding a conjunction of equalities with equivalence classes). Consider Algorithm 4.7.1. Present details of an efficient implementation of this algorithm, including a data structure. What is the complexity of your implementation?

```
Algorithm 4.7.1: Decide-a-conjunction-of-equalities-with-equivalence-classes

Input: A conjunction $\varphi^E$ of equality predicates
Output: “Satisfiable” if $\varphi^E$ is satisfiable, and “Unsatisfiable” otherwise

1. Define an equivalence class for each variable. For each equality $x = y$ in $\varphi^E$, unite the equivalence classes of $x$ and $y$.
2. For each disequality $u \neq v$ in $\varphi^E$, if $u$ is in the same equivalence class as $v$, return “Unsatisfiable”.
3. Return “Satisfiable”.
```

Problem 4.2 (deciding a conjunction of equality predicates with a graph analysis). Show a graph-based algorithm for deciding whether a given conjunction of equality predicates is satisfiable. What is the complexity of your algorithm?
Problem 4.3 (a generalization of the Congruence-Closure algorithm). Generalize Algorithm 4.1.1 to the case in which the input formula includes uninterpreted functions with multiple arguments.

4.7.2 Reductions

Problem 4.4 (a better way to eliminate constants?). Is the following theorem correct?

Theorem 4.27. An equality formula $\varphi^E$ is satisfiable if and only if the formula $\varphi^{E'}$ generated by Algorithm 4.7.2 (Remove-constants-optimized) is satisfiable.

Prove the theorem or give a counterexample. You may use the result of Problem 3.2 in your proof.

Algorithm 4.7.2: Remove-constants-optimized

**Input:** An equality logic formula $\varphi^E$

**Output:** An equality logic formula $\varphi^{E'}$ such that $\varphi^{E'}$ contains no constants and $\varphi^{E'}$ is satisfiable if and only if $\varphi^E$ is satisfiable

1. $\varphi^{E'} := \varphi^E$.
2. Replace each constant $c$ in $\varphi^{E'}$ with a new variable $C_c$.
3. For each pair of constants $c_i, c_j$ with an equality path between them ($c_i =^* c_j$) not through any other constant, add the constraint $C_{c_i} \neq C_{c_j}$ to $\varphi^{E'}$. (Recall that the equality path is defined over $G^E(\varphi^E)$, where $\varphi^E$ is given in NNF.)

Problem 4.5 (correctness of the simplification step). Prove the correctness of Algorithm 4.3.1. You may use the proof strategy suggested in Sect. 4.3.

Problem 4.6 (reduced transitivity constraints). (Based on [126, 169].) Consider the equality graph in Fig. 4.9. The sparse method generates $B_{\text{trans}}$ with three transitivity constraints (recall that it generates three constraints for each triangle in the graph, regardless of the polarity of the edges). Now consider the following claim: the single transitivity constraint $B_{\text{rtc}} = (e_{0,2} \land e_{1,2} \implies e_{0,1})$ is sufficient (the subscript rtc stands for “reduced transitivity constraints”).

To justify this claim, it is sufficient to show that for every assignment $\alpha_{\text{rtc}}$ that satisfies $e(\varphi^E) \land B_{\text{rtc}}$, there exists an assignment $\alpha_{\text{trans}}$ that satisfies $e(\varphi^E) \land B_{\text{trans}}$. Since this, in turn, implies that $\varphi^E$ is satisfiable as well, we obtain the result that $\varphi^E$ is satisfiable if and only if $e(\varphi^E) \land B_{\text{rtc}}$ is satisfiable.
Fig. 4.9. Taking polarity into account allows us to construct a less constrained formula. For this graph, the constraint $B_{rtc} = (e_{0,2} \land e_{1,2} \implies e_{0,1})$ is sufficient. An assignment $\alpha_{rtc}$ that satisfies $B_{rtc}$ but breaks transitivity can always be “fixed” so that it does satisfy transitivity, while still satisfying the propositional skeleton $e(\varphi^E)$. The assignment $\alpha_{trans}$ demonstrates such a “fixed” version of the satisfying assignment.

We are able to construct such an assignment $\alpha_{trans}$ because of the monotonicity of NNF (see Theorem 1.14, and recall that the polarity of the edges in the equality graph is defined according to their polarity in the NNF representation of $\varphi^E$). There are only two satisfying assignments to $B_{rtc}$ that do not satisfy $B_{trans}$. One of these assignments is shown in the $\alpha_{rtc}$ column in the table to the right of the drawing. The second column shows a corresponding assignment $\alpha_{trans}$, which clearly satisfies $B_{trans}$.

However, we still need to prove that every formula $e(\varphi^E)$ that corresponds to the above graph is still satisfied by $\alpha_{trans}$ if it is satisfied by $\alpha_{rtc}$. For example, for $e(\varphi^E) = (\neg e_{0,1} \lor e_{1,2} \lor e_{0,2})$, both $\alpha_{rtc} \models e(\varphi^E) \land B_{rtc}$ and $\alpha_{trans} \models e(\varphi^E) \land B_{trans}$. Intuitively, this is guaranteed to be true because $\alpha_{trans}$ is derived from $\alpha_{rtc}$ by flipping an assignment of a positive (unnegated) predicate ($e_{0,2}$) from FALSE to TRUE. We can equivalently flip an assignment to a negated predicate ($e_{0,1}$ in this case) from TRUE to FALSE.

1. Generalize this example into a claim: given a (polar) equality graph, which transitivity constraints are necessary and sufficient?
2. Show an algorithm that computes the constraints that you suggest in the item above. What is the complexity of your algorithm? (Hint: there exists a polynomial algorithm, which is hard to find. An exponential algorithm will suffice as an answer to this question).

4.7.3 Complexity

Problem 4.7 (complexity of deciding equality logic). Prove that deciding equality logic is NP-complete.

Note that to show membership in NP, it is not enough to say that every solution can be checked in P-time, because the solution itself can be arbitrarily large, and hence even reading it is not necessarily a P-time operation.
4.7.4 Domain Allocation

**Problem 4.8 (adequate domain for $\Phi_{n,k}$).** Prove Theorem 4.20.

**Problem 4.9 (small-domain allocation).** Prove the following lemma.

**Lemma 4.28.** If a domain $D$ is adequate for $\Phi(e)$ and $e' \subseteq e$, then $D$ is adequate for $\phi(e')$.

**Problem 4.10 (small-domain allocation: an adequate domain).** Prove the following theorem:

**Theorem 4.29.** If all the subsets of $E(\varphi^e)$ are consistent, then there exists an allocation $R$ such that $|R| = 1$.

**Problem 4.11 (formulation of the graph-theoretic problem).** Give a self-contained formal definition of the following decision problem: given an equality graph $G$ and a domain allocation $D$, is $D$ adequate for $G$?

**Problem 4.12 (small-domain allocation: an improvement to the allocation heuristic).** Step II.A.1 of Algorithm 4.5.1 calls for allocation of distinct characteristic values to the mixed vertices. The following example proves that this is not always necessary.

Consider the subgraph $\{u_1, f_1, f_2, u_2\}$ of the graph in Fig. 4.2. Application of the basic algorithm to this subgraph may yield the following allocation, where the characteristic values assigned are underlined: $R_1 : u_1 \mapsto \{0, 2\}, f_1 \mapsto \{0\}, f_2 \mapsto \{0, 1\}, u_2 \mapsto \{0, 1, 3\}$. This allocation leads to a state space complexity of 12. By relaxing the requirement that all individually assigned characteristic values should be distinct, we can obtain the allocation $R_2 : u_1 \mapsto \{0, 2\}, f_1 \mapsto \{0\}, f_2 \mapsto \{0\}, u_2 \mapsto \{0, 1\}$ with a state-space complexity of 4. This reduces the size of the state space of the entire graph from 48 to 16.

It is not difficult to see that $R_2$ is adequate for the subgraph considered.

What are the conditions under which it is possible to assign equal values to mixed variables? Change the basic algorithm so that it includes this optimization.

4.8 Bibliographic Notes

The treatment of equalities and uninterpreted functions can be divided into several eras. In the first era, before the emergence of the first effective theorem provers in the 1970’s, this logic was considered only from the point of view of mathematical logic, most notably by Ackermann [1]. In the same book, he also offered what we have called Ackermann’s reduction in this book. Equalities were typically handled with rewriting rules, for example substituting $x$ with $y$ given that $x = y$. 
The second era started in the mid 1970’s with the work of Downey, Sethi, and Tarjan [69], who showed that the decision problem was a variation on the common-subexpression problem; the work of Nelson and Oppen [136], who applied the union–find algorithm to compute the congruence closure and implemented it in the Stanford Pascal Verifier; and then the work of Shostak, who suggested in [178] the congruence closure method that was briefly presented in Sect. 4.1. All of this work was based on computing the congruence closure, and indicated a shift from the previous era, as it offered complete and relatively efficient methods for deciding equalities and uninterpreted functions. In its original presentation, Shostak’s method relied on syntactic case-splitting (see Sect. 1.3), which is the source of the inefficiency of that algorithm. In Shostak’s words, “it was found that most examples four or five lines long could be handled in just a few seconds”. Even factoring in the fact that this was done on a 1978 computer (a DEC-10 computer), this statement still shows how much progress has been made since then, as nowadays many formulas with tens of thousands of variables are solved in a few seconds. Several variants on Shostak’s method exist, and have been compared and described in a single theoretical framework called abstract congruence closure in [8]. Shostak’s method and its variants are still used in theorem provers, although several improvements have been suggested to combat the practical complexity of case-splitting, namely lazy case-splitting, in which the formula is split only when it is necessary for the proof, and other similar techniques.

The third era of deciding this theory avoided syntactic case-splitting altogether and instead promoted the use of semantic case-splitting, that is, splitting the domain instead of splitting the formula. All of the methods of this type are based on an underlying decision procedure for Boolean formulas, such as a SAT engine or the use of BDDs. We failed to find an original reference for the fact that the range \{1, \ldots, n\} is adequate for formulas with \(n\) variables. This is usually referred to as a “folk theorem” in the literature. The work by Hojati, Kuehlmann, German, and Brayton in [95] and Hojati, Isles, Kirkpatrick, and Brayton in [94] was the first, as far as we know, where anyone tried to decide equalities with finite instantiation, while trying to derive a value \(k\), \(k \leq n\) that was adequate as well, by analyzing the equality graph. The method presented in Sect. 4.5 was the first to consider a different range for each variable and, hence, is much more effective. It is based on work by Pnueli, Rodeh, Siegel, and Strichman in [154, 155]. These papers suggest that Ackermann’s reduction should be used, which results in large formulas, and, consequently, large equality graphs and correspondingly large domains (but much smaller than the range \{1, \ldots, n\}). Bryant, German and Velev suggested in [38] what we refer to as Bryant’s reduction in Sect. 3.3.2. This technique enabled them to exploit what they called the positive equality structure in formulas for assigning unique constants to some of the variables and a full range to the others. Using the terminology of this chapter, these variables are adjacent only to solid edges in the equality graph corresponding to the original formula (a graph built without referring to the functional-consistency
constraints, and hence the problem of a large graph due to Ackermann’s constraints disappears). A more robust version of this technique, in which a larger set of variables can be replaced with constants, was later developed by Lahiri, Bryant, Goel, and Talupur [112].

In [167, 168], Rodeh and Shtrichman presented a generalization of positive equality that enjoys benefits from both worlds: on the one hand, it does not add all the edges that are associated with the functional-consistency constraints (it adds only a small subset of them based on an analysis of the formula), but on the other hand it assigns small ranges to all variables as in [155] and, in particular, a single value to all the terms that would be assigned by the technique of [38]. This method decreases the size of the equality graph in the presence of uninterpreted functions, and consequently the allocated ranges (for example, it allocates a domain with a state space of size 2 for the running example in Sect. 4.5.3). Rodeh showed in his thesis [167] (also see [153]) an extension of range allocation to dynamic range allocation. This means that each variable is assigned not one of several constants, as prescribed by the allocated domain, but rather one of the variables that represent an immediate neighbor in $G_e$, or a unique constant if it has one or more neighbors in $G_e$. The size of the state space is thus proportional to $\log n$, where $n$ is the number of neighbors.

Goel, Sajid, Zhou, Aziz, and Singhal were the first to encode each equality with a new Boolean variable [87]. They built a BDD corresponding to the encoded formula, and then looked for transitivity-preserving paths in the BDD. Bryant and Velev suggested in [39] that the same encoding should be used but added explicit transitivity constraints instead. They considered several translation methods, only the best of which (the sparse method) was presented in this chapter. One of the other alternatives is to add such a constraint for every three variables (regardless of the equality graph). A somewhat similar approach was considered by Zantema and Groote [206]. The sparse method was later superseded by the method of Meir and Strichman [126] and later by that of Rozanov and Strichman [169], where the polar equality graph is considered rather than the nonpolar one, which leads to a smaller number of transitivity constraints. This direction is mentioned in Problem 4.6.

All the methods that we discussed in this chapter, other than congruence closure, belong to the third era. A fourth era, based on an interplay between a SAT solver and a decision procedure for a conjunction of terms (such as congruence closure in the case of EUF formulas), has emerged in the last few years, and is described in detail in Chap. 11. The idea is also explained briefly at the end of Sect. 4.1.

4.9 Glossary

The following symbols were used in this chapter:
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Refers to . . .</th>
<th>First used on page . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_=$, $E_\neq$</td>
<td>Sets of equality and inequality predicates, and also the edges in the equality graph</td>
<td>83</td>
</tr>
<tr>
<td>$At(\varphi^E)$</td>
<td>The set of atoms in the formula $\varphi^E$</td>
<td>83</td>
</tr>
<tr>
<td>$G^E$</td>
<td>Equality graph</td>
<td>84</td>
</tr>
<tr>
<td>$x =^* y$</td>
<td>There exists an equality path between $x$ and $y$ in the equality graph</td>
<td>84</td>
</tr>
<tr>
<td>$x \neq^* y$</td>
<td>There exists a disequality path between $x$ and $y$ in the equality graph</td>
<td>84</td>
</tr>
<tr>
<td>$e(\varphi^E)$</td>
<td>The propositional skeleton of $\varphi^E$</td>
<td>89</td>
</tr>
<tr>
<td>$\mathcal{B}_{\text{trans}}$</td>
<td>The transitivity constraints due to the reduction from $\varphi^E$ to $\mathcal{B}_{\text{sat}}$ by the sparse method</td>
<td>89</td>
</tr>
<tr>
<td>$G^E_{\text{NP}}$</td>
<td>Nonpolar equality graph</td>
<td>89</td>
</tr>
<tr>
<td>$\text{var}(\varphi^E)$</td>
<td>The set of variables in $\varphi^E$</td>
<td>92</td>
</tr>
<tr>
<td>$D$</td>
<td>A domain allocation function. See (4.23)</td>
<td>92</td>
</tr>
<tr>
<td>$</td>
<td>D</td>
<td>$</td>
</tr>
<tr>
<td>$\Phi_n$</td>
<td>The (infinite) set of equality logic formulas with $n$ variables</td>
<td>93</td>
</tr>
<tr>
<td>$\Phi_{n,k}$</td>
<td>The (infinite) set of equality logic formulas with $n$ variables and $k$ constants</td>
<td>94</td>
</tr>
<tr>
<td>$\phi(e)$</td>
<td>The (infinite) set of equality formulas with a set of literals equal to $e$</td>
<td>95</td>
</tr>
<tr>
<td>$E(\varphi^E)$</td>
<td>The set of literals in $\varphi^E$</td>
<td>95</td>
</tr>
<tr>
<td>$G^E_=$, $G^E_\neq$</td>
<td>The projections of the equality graph on the $E_=$ and $E_\neq$ edges, respectively</td>
<td>96</td>
</tr>
<tr>
<td>$\text{char}(v)$</td>
<td>The characteristic value of a node $v$ in the equality graph</td>
<td>96</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Refers to ...</th>
<th>First used on page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MV</td>
<td>The set of mixed vertices that are chosen in step II.A.1 of Algorithm 4.5.1</td>
<td>96</td>
</tr>
<tr>
<td>$a_e(x)$</td>
<td>An assignment to a variable $x$ from its allocated domain $D(x)$</td>
<td>99</td>
</tr>
</tbody>
</table>