Chapter 2

General Labeling: Label-Number Maximization

The problem of label placement is usually divided into point, line, and area labeling, depending on the kind of features to be labeled. However, the problem can be formulated independently of the shape of features. Two interesting subproblems have been studied. In both cases, an instance consists of a set of features and a set of label candidates for each feature.

1. *The Label-Size Maximization Problem*: Find the maximum factor σ such that each feature gets a label stretched by this factor and no two labels overlap. Compute the corresponding *complete* label placement.

2. *The Label-Number Maximization Problem*: Find a maximum subset of the features, and for each of these features a label from its set of candidates, such that no two labels overlap.

The decision versions of both problems are NP-hard in general [FW91, FPT81]. The label-size maximization problem can be solved in polynomial time if all features have at most two label candidates. Then the problem can be encoded as a 2-SAT formula and tested for satisfiability in time linear in the number of pairs of intersecting candidates [EIS76]. This was already observed in [FW91]. If the label candidates of a feature overlap in a certain manner, polynomial time algorithms are known for any constant number of label candidates per feature [PZC98, SvK99], and even for an infinite number of label candidates per feature [KSY99].

In recent years, especially the point-labeling problem has achieved some attention in the algorithms community. For maximizing the number of points that are labeled with axis-parallel rectangles, the current status of the problem is described in Chapter 3. For problems related to maximizing the size of rectangular or circular labels for point features, refer to Chapter 4. In this chapter we investigate the general label-number maximization problem.

Methods that have been used for label-number maximization so far are heuristical; they include simulated annealing [CMS95, ECMS97, Zor97] and an algorithm that uses maximum-cardinality bipartite matching between features and cliques of intersecting label candidates [KT98]. Both approaches will be discussed in more detail in Section 3.2.

We propose a new framework for the general label-number maximization problem. It leads to a heuristical algorithm that is easy to implement, and, for point labeling, yields better results than the matching heuristic of [KT98] and similarly good results as simulated annealing, but obtains them much faster, see Section 3.2. Our framework is related to a concept suggested in the artificial intelligence community under the name *constraint satisfaction*, which was independently introduced into the discrete mathematics community by Knuth and Raghunathan under the name *problem of compatible representatives* [KR92]. The difference of our approach to that of the artificial intelligence community is that we try to maximize the number of variables (features) with a conflict-free assignment, while their objective is either to list *all* assignment tuples without conflicts [MF85], to minimize the number of conflicts [FW92], or to find the maximum weighted subset of constraints that still allows an assignment.

Since constraint satisfaction is NP-hard in general, the artificial intelligence community invented so-called *network-consistency algorithms*. These algorithms establish a form of consistency; i.e. they use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network-consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

We develop the notion of *r-irreducibility*, a new form of local consistency that is comparable to consistency in classical constraint satisfaction. We give an algorithm, EI-1, that achieves 2-irreducibility in $O(d^3e)$ time using $O(de)$ space, where d is the maximum domain size and e the number of pairs of variables whose values are in conflict with each other. The domain of a variable corresponds to the set of label candidates of a feature in label placement. The value of a variable is nothing but a label candidate, and for us, two values are in conflict with each other if the corresponding candidates intersect.

While d is considered to be a small constant in point labeling (usually four or eight), there are many applications in artificial intelligence where d can be very large. Thus we take the size of d into account in this chapter. Note that k, the number of pairs of intersecting label candidates, is of $O(d^2e)$.

In addition, we present an algorithm, EI-1*, for general label-number maximization that is based on EI-1. This algorithm first establishes 2-irreducibility. Then it repeatedly makes a heuristical decision and restores 2-irreducibility until each feature is either labeled or known to constrain too many other features and therefore not labeled at all. Given the value conflict graph, EI-1* requires $O(d^3e)$ time and $O(de)$ space like EI-1.

Our new framework is called *maximum variable-subset constraint satisfaction*. Our hope is that EI-1* or other efficient algorithms based on higher

degrees of irreducibility will substitute simulated annealing for the wide variety of problems that fit into our framework. Experiments in the context of point labeling indicate that EI-1* is not only fast but also very effective in practice, see Section 3.2.

This chapter is structured as follows. In Section 2.1 we give a quick introduction into the issues relevant for label placement that have been investigated by the artificial intelligence community. We consider classical constraint satisfaction problems (CSP) and a generalization, namely Max-CSP. In Section 2.2 we extend classical CSP to maximum variable-subset CSP where the label-number maximization problem can easily be formulated. In Section 2.3 and 2.4 we define irreducibility and describe our 2-irreducibility algorithm EI-1. Finally, in Section 2.5 we present our algorithm EI-1* for the general label-number maximization problem.

2.1 Label Placement and CSP

A constraint satisfaction problem (CSP) is defined as follows. Given a set of n variables $v_1, \ldots, v_n$, each associated with a domain D_i and a set of relations constraining the assignment of subsets of the variables, find all possible n-tuples of variable assignments that satisfy the relations [MF85]. Variable domains are restricted to discrete finite sets, and often only binary relations are considered.

Graph coloring is a special case of a CSP where the variables are nodes, the domains a given set of colors, and binary relations express the fact that a node cannot have the same color as any of its neighbors. Since graph coloring, i.e. deciding whether the nodes of a graph can be colored with the given set of colors, is NP-complete, one cannot expect to solve general CSPs in polynomial time [MF85]. For this reason, the class of *network-consistency algorithms* has been invented. These algorithms use local arguments to exclude values from the domain of a variable that cannot be part of a global solution. Network-consistency algorithms can be seen as a preprocessing step to backtracking since they often reduce the search space very effectively.

An m-consistency algorithm removes all inconsistencies among all subsets of m of the given n variables. For the special cases of $m = 1$, 2, and 3, polynomial-time algorithms have been suggested. They are called node-, arc-, and path-consistency algorithms, respectively.

This framework can be used nearly one-to-one for attacking the label *size* maximization problem. When maximizing simultaneously the sizes of all labels, one can do a binary search on *conflict sizes*, i.e. label sizes for which label candidates start to touch. For each conflict size, one then tries to find a complete labeling. Obviously, a feature can be seen as a variable, the set of label candidates of a feature then corresponds to the variable domain and intersections between label candidates are the constraining binary relations. Instead of computing *all* satisfying variable assignments, finding *one* is usually sufficient in the map-labeling context. This allows to reduce the search space dramati-

cally since a variable can immediately be assigned an unconstrained value from its domain if there is such a value. The algorithm for label size maximization suggested in [WW97] exploits this simplification.

When maximizing the number of labeled features, label sizes are fixed and one cannot give up and try a smaller label size as soon as it turns out that there is no complete labeling for the current label size. Systems where one cannot expect to find a *complete solution*, i.e. a non-conflicting variable assignment, are called *over-constrained systems*. In such systems one has to be content with imperfect solutions. Most effort in the CSP community has been directed to finding solutions that violate as few constraints as possible [FW92, Jam96, JFM96]. When labeling maps, such violations would result in label over-plots and thus poor legibility. It would be possible to take the output of an algorithm that minimizes the number of violated constraints and then do some post-processing. In order to get rid of the violations, one could drop a subset of the variables and resign from labeling the corresponding features. Unfortunately the problem of finding the largest violation-free subset of variables corresponds to the maximum independent set problem and is thus NP-hard.

A related problem, Max-CSP, has also been investigated. There, one is interested in finding a maximum (weighted) subset of the constraints such that there is an assignment that satisfies them all. In order to reduce label-number maximization to Max-CSP, one adds a new value $\triangle$ to the domain of each variable. $\triangle$ has a unary constraint of low weight; i.e. it only constrains itself. A variable that is assigned $\triangle$ then corresponds to an unlabeled feature in our setting. For general Max-CSP, however, even arc consistency is NP-hard [SFV95].

Therefore we take a different approach. We first extend classical CSP in order to be able to express the label-number maximization problem within this new framework, see Section 2.2. Then, in Section 2.3 we develop a new form of local consistency, namely *r-irreducibility*. In Section 2.4, we present an algorithm, EI-1, that achieves 2-irreducibility in $O(d^2e)$ time using $O(de)$ space, where d is the size of the variable domains and e the number of binary relations. Finally, in Section 2.5 we give a simple algorithm that finds near-optimal solutions for problems within our framework by combining EI-1 with a heuristic. This algorithm has proven to be very effective in practice, see Section 3.2, where we apply it to the point-labeling problem.

2.2 Maximum Variable-Subset CSP

Let us start by giving a formal definition of classical CSP [SFV95].

Definition 2.1 (CSP) *An instance of a* constraint-satisfaction problem (CSP) *is a triple* $(V, \mathcal{D}, \mathcal{C})$ *where* V *is a set of* n *variables,* $\mathcal{D}$ *a collection of* domains*, one for each variable, and* $\mathcal{C}$ *a set of* constraints*. A domain* D_v *of a variable* v *is a (finite) set of* values *of* v*. A constraint* $C \in \mathcal{C}$ *is given by a pair*

(V_C, R_C) where $V_C \subseteq V$ is a subset of the variables and $R_C \subseteq \Pi_{v \in V_C} D_v$ is a relation on the variables in V_C.

A solution *of a CSP is a function π that maps each variable to a value of its domain such that all constraints are satisfied, i.e. $\Pi_{v \in V_C} \pi(v) \in R_C$ for all $C \in \mathcal{C}$.*

In classical CSP one is either interested in finding one or in listing all solutions. We extend classical CSP in order to be able to better formulate label-number maximization. In the following definition we assume that no variable domain contains an element 0.

Definition 2.2 (MVS-CSP) *A* solution *of a* maximum variable-subset CSP (MVS-CSP) *$(V, \mathcal{D}, \mathcal{C})$ is a function π that assigns every variable v in V to a value of its domain D_v or to 0 such that all relevant constraints are satisfied, i.e. for all $C \in \mathcal{C}$ if $0 \notin \pi(V_C)$ then $\Pi_{w \in V_C} \pi(w) \in R_C$.*

The size *$|\pi|$ of a solution π is the number of variables v in V that π assigns a value $\pi(v) \in D_v$. In MVS-CSP an* optimal *solution is a solution of maximum size. A solution of size $|V|$ is called a* complete *solution,*

In our definition we drop a constraint $C = (V_C, R_C)$ completely if any of the variables v in V_C is mapped to 0. The reason for this part of our definition is that the restriction of C imposed on the variables in $V_C \setminus \{v\}$ depends on the value of v, thus we cannot make any assumption about which combination of values of $V_C \setminus \{v\}$ is excluded by C. It makes sense to require that V_C is in a sense minimal, in other words that there is no $v \in V_C$ such that the projection of R_C to $\{x\} \times \Pi_{w \in V_C \setminus \{v\}} D_w$ is identical for all values $x \in D_v$.

Definition 2.2 transfers the decision or enumeration problem of classical CSP into an optimization problem.

For label placement only binary constraints are relevant, i.e. $|V_C| = 2$ for all $C \in \mathcal{C}$. Given two features f and g of a label-placement instance, these binary constraints encode which pairs of label candidates b and c of f and g intersect, respectively. Thus we can use a simpler definition.

Definition 2.3 (binary MVS-CSP) *An instance of a* binary MVS-CSP *is a triple $(V, \mathcal{D}, \mathcal{R})$ where $\mathcal{R}$ is a set of predicates R_{vw} on $(D_v \cup \{0\}) \times (D_w \cup \{0\})$, one for each pair (v, w) of variables. For $x = 0$ or $y = 0$ $R_{vw}(x, y)$ is always true. A solution π must fulfill $R_{vw}(\pi(v), \pi(w))$ for all $v \neq w \in V$.*

Given a binary CSP, constraint information can be encoded conveniently by any of the graphs that we define in the following.

Definition 2.4 (variable/value constraint/conflict graph) *We say that a value x of a variable v* constrains *a value y of a variable w if $R_{vw}(x, y)$ is false. Let $R^*_{vw}(x, y)$ be the symmetric predicate that is true if $R_{vw}(x, y)$ and*

$R_{wv}(y,x)$ *are true. Then* x *and* y *are* in conflict *if* $R^*_{vw}(x,y)$ *is false. We say that* w excludes *a value* x *of* v *if all values of* w *constrain* x*.*

We say that a variable v *constrains (is in conflict with) a variable* w *if there is a value in the domain of* v *that constrains (is in conflict with) a value in the domain of* w*. The* variable constraint graph $\overrightarrow{G}(V,\overrightarrow{E})$ *has an arc for each pair* (v,w) *where* v *constrains* w*; the* variable conflict graph $G(V,E)$ *has an edge for each pair* $\{v,w\}$ *where* v *is in conflict with* w*. In the* value constraint (conflict) graph $\overrightarrow{G_{\mathcal{D}}}$ *(*$G_{\mathcal{D}}$*) there is a vertex for each variable-value pair* $[v,x]$ *with* $v \in V$ *and* $x \in D_v$*, and an arc (edge) between two such pairs* $[v,x]$ *and* $[w,y]$ *iff* $R_{vw}(x,y)$ *(*$R^*_{vw}(x,y)$*) false.*

The question whether an instance of MVS-CSP has a complete solution corresponds to classical CSP. Thus the decision version of MVS-CSP, namely *Is there a solution of size* s*?*, is NP-hard as well. Note that MVS-CSP corresponds to maximum independent set on $G_{\mathcal{D}}$ if we make the values of each variable into cliques, i.e. if we add edges between $[v,x]$ and $[v,y]$ for all $v \in V$ and $x,y \in D_v$ with $x \neq y$.

In order to approach classical CSP in spite of its NP-hardness, the notion of consistency has been developed. An instance is *m-consistent* if all m-element subsets $W \subseteq V$ are consistent, i.e. if for each value x in the domain of any variable w in W there is a complete solution for W that maps w to x. m-consistency introduces a scale between totally inconsistent and perfectly consistent. Node-, arc-, and path-consistency algorithms achieve 1-, 2-, and 3-consistency in polynomial time [Mac77]. For backtracking algorithms, achieving arc- or path-consistency is an important preprocessing step that reduces checking the same inconsistent variable assignment repeatedly. In the extreme, for each $v \in V$ and each $x \in D_v$ a $|V|$-consistent instance yields a complete solution that maps v to x.

The input to network-consistency algorithms comprises usually the variable constraint graph, the domain of each variable, and for each arc (v,w) of the graph a method that returns the value of $R_{vw}(x,y)$ for all pairs $(x,y) \in D_v \times D_w$. The variable constraint graph can be transformed into a value constraint graph, but the latter might need up to a factor of $O(d^2)$ more storage, where d is the (maximum) size of the variable domains.

2.3 Irreducibility

The potential of network-consistency algorithms is our motivation for transferring the concept of consistency to MVS-CSP. In our setting, we refer to it as *irreducibility*, which we define as follows.

Definition 2.5 (reducible, redundant) *Given a* binary MVS-CSP $(V,\mathcal{D},\mathcal{R})$ *and a subset* $W \subseteq V$*, a variable* $v \in W$ *is* W-reducible *iff there is a value* $x \in D_v$ *such that for all solutions* π *of* W *with* $\pi(v) = x$ *there*

is a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. For all $w \in W$, $\pi'(w)$ must be either equal to $\pi(w)$ or not in conflict with any values of variables in $V \setminus W$, i.e. $R^*_{vw}(y, \pi'(w))$ for all $v \in V \setminus W$ and all $y \in D_v$. If such solutions π' exist, x is called W-redundant.

$W \subseteq V$ is irreducible *iff there is no $v \in W$ that is W-reducible. V is* r-irreducible *iff all r-element subsets $W \subseteq V$ are irreducible.*

Note that r-irreducibility implies i-irreducibility for all $i < r$. Node-, arc-, and path-irreducible will be used as synonyms for 1-, 2-, and 3-irreducible. If all constraints are symmetric, we will use edge-irreducible instead of arc-irreducible. The notion of reducibility helps us to remove redundant values from variable domains and thus reduce the search space for an optimal solution.

Lemma 2.6 *Let π be an optimal solution of a binary MVS-CSP $(V, \mathcal{D}, \mathcal{R})$. If there is a subset $W \subseteq V$ and a variable $v \in W$ that is W-reducible, then there is an x in the domain D_v of v such that $(V, \mathcal{D}', \mathcal{R})$ has a solution of size $|\pi|$, where $\mathcal{D}' = \{D_v \setminus \{x\} \mid v \in V\}$.*

Proof. We assume that $(V, \mathcal{D}', \mathcal{R})$ has only solutions strictly smaller than π. If $\pi(v) \neq x$ then π would also be a solution to the reduced instance, contradicting our assumption. Thus $\pi(v) = x$. Then, by definition of reducibility, there must be a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi^W|$ where π^W is the restriction of π to W. For each $w \in W$, π' must either fulfill $\pi'(w) = \pi(w)$ or $R^*_{vw}(y, \pi'(w))$ for all $v \in V \setminus W$ and all $y \in D_v$. Let ρ be the following function on V.

$$\rho(u) = \begin{cases} \pi(u) & \text{for all } u \in V \setminus W; \\ \pi'(u) & \text{otherwise.} \end{cases}$$

We show that ρ is a solution of the reduced instance. Let v, $w \in V$ with $\rho(v) = y \neq 0$ and $\rho(w) = z \neq 0$. We must show that y and z are not in conflict. This is clear if $\{v, w\} \subseteq V \setminus W$ and if $\{v, w\} \subseteq W$ since ρ equals π and π' on the respective subsets of V, and π and π' are solutions on $V \setminus W$ and W, respectively. Thus it is enough to consider the case $v \in V \setminus W$ and $w \in W$. On the one hand this implies $\pi(v) = \rho(v) = y$. On the other hand, we have either $\pi(w) = \pi'(w) = \rho(v) = z$ or $R^*_{uw}(a, \pi'(w))$ for all $u \in V \setminus W$ and all $a \in D_u$. In the first case y and z are both part of solution π and therefore cannot be in conflict. In the second case, too, y and z are not in conflict since $v \notin W$ and thus $R^*_{vw}(y, \pi'(w))$.

Since $|\rho^W| = |\pi'| \geq |\pi^W|$ and $\rho^{V \setminus W} = \pi^{V \setminus W}$ we have $|\rho| \geq |\pi|$, which contradicts our assumption. □

$|V|$-irreducibility gives us direct access to an optimal solution.

Lemma 2.7 *In a $|V|$-irreducible binary MVS-CSP $(V, \mathcal{D}, \mathcal{R})$ all variable domains contain at most one value, i.e. $|D_v| \leq 1$ for all variables $v \in V$.*

Proof. Suppose there is a variable $v \in V$ with $|D_v| > 1$ and v is not V-reducible. There are two possibilities. Either there is an optimal solution π_{opt} of V that maps v to a $y \in D_v$ or all optimal solutions map v to 0. In the second case let π_{opt} be one of these solutions, and set y to 0.

In either case there is a value $x \in D_v \setminus \{y\}$ and for all solutions π of V with $\pi(v) = x$ there is a solution π' of V (namely π_{opt}) with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. Thus v is V-reducible since the additional condition, namely $R^*_{vw}(y, \pi'(w))$ for all $v \in V \setminus V$ and all $y \in D_v$, is trivial for V-reducibility. Hence our assumption is contradicted. □

2.4 An Edge-Irreducibility Algorithm

Mackworth [Mac77] proposed an algorithm, AC-3, that achieves arc-consistency in polynomial time, see Figure 2.2. With Freuder [MF85] he showed later that AC-3 requires at least $\Omega(d^2e)$ and at most $O(d^3e)$ time, where e is the size of the variable conflict graph and d the size of the variable domains, which are assumed to be of equal size for all variables. AC-3 does not assume that constraints are symmetric. It uses $O(de)$ storage.

The heart of AC-3 is a procedure REVISE that, given a pair (v, w) of variables, eliminates all values from the domain of v that are excluded by w, see Figure 2.1. AC-3 uses a stack to keep track of all pairs of variables that potentially need revision. Initially the stack is filled with all arcs of the variable constraint graph. Until the stack (or a variable domain) is empty, AC-3 repeatedly draws a pair (v, w) from the stack, calls REVISE(v, w), and, if REVISE removed a value from the domain of v, adds all arcs (u, v) to the stack.

The task of REVISE is simple. It makes the arc (v, w)-consistent by removing all values of v that are excluded by w and therefore cannot be part of any complete solution. Without any additional data structures REVISE requires $O(d^2)$ time. The time complexity of AC-3 follows from the fact that REVISE is called at most d times for each of the e edges of the variable constraint graph.

Later, Mohr and Henderson introduced the notion of *support* [MH86]. A variable-value pair $[v, x]$ supports the value y of a variable w if $R_{vw}(x, y)$. (We will switch between value and variable-value pair depending on which is more convenient.) As soon as $[v, x]$ loses its last support from a variable w that constrains v, x must be removed from the domain of v. Mohr and Henderson gave an algorithm, AC-4, that is based on this idea. For each variable-value pair $[v, x]$, AC-4 keeps track of the number $k_{v,x}$ of values that support $[v, x]$ and maintains a list $S_{v,x}$ with all values that $[v, x]$ supports. Using these data structures yields AC-4's optimal time complexity of $O(d^2e)$. However, they are also responsible for the fact that AC-4 requires $O(d^2e)$ storage. In addition, average and worst case runtime behavior of AC-4 do not differ much. These disadvantages made AC-3 in spite of its inferior time complexity favorable in many applications [Bes94].

```
REVISE(v, w)

deleted ← false
for each x ∈ D_v do
    for each y ∈ D_w do
        if R_vw(x, y) then exit inner loop end
    end
    if ¬R_vw(x, y) then
        D_v ← D_v \ {x}
        deleted ← true
    end
end
return deleted
```

Figure 2.1: The procedure REVISE makes the arc (v, w) consistent.

```
AC-3(V, D, R)

E ← { (v, w) | v, w ∈ V, ∃ x ∈ D_v, y ∈ D_w : ¬R_vw(x, y) }
Q ← E
while Q ≠ ∅ do
    (v, w) ← Q.pop()
    if REVISE(v, w) = true then
        for each u ∈ V such that (u, v) ∈ E do
            Q.push((u, v))
        end
    end
end
```

Figure 2.2: The third arc-consistency algorithm AC-3.

Bessière found out that it is not necessary to maintain counters and that it is enough to keep *one* support for each of the $O(de)$ arc-value pairs [Bes94]. His algorithm AC-6 exploits these observations and takes advantage of a total order on the values in each domain. AC-6 needs less storage than AC-4, namely $O(de)$. Although it shares the time complexity of $O(d^2e)$ with AC-4, it needs less predicate evaluations than both its predecessors AC-3 and AC-4. Shortly after, Bessière, Freuder, and Régin suggested improvements of AC-6 that led to AC-7. This algorithm requires even less predicate evaluations than AC-6 while keeping the asymptotic space and time complexity of its predecessor [BFR95].

Unfortunately the concept of support does not work in the context of MVS-CSP. If a variable-value pair $[b, 1]$ loses support from a variable c, this only means that not both $[b, 1]$ and a value $y \in D_c$ can be part of a solution. However, it does not imply that an optimal solution will not map b to 1. It does not even imply that there is an optimal solution that does not map b to 1 as the example

in Figure 2.3 demonstrates. There, variables are represented by boxes and their values by circles. Conflicting values are connected by edges; all values have degree 3 in the *value conflict graph*, except the values of c that have degree 4. While the optimal solution (indicated by bold circles) has size 4, all solutions that map b to 2 (or 0) have size at most 3.

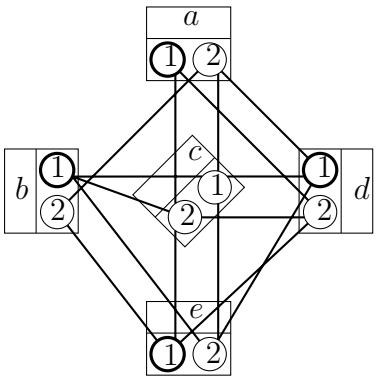

Figure 2.3: Example where the only value $(b, 1)$ that is lacking support on an edge (namely $\{b, c\}$) is in the only optimal solution (indicated by bold circles).

From now on we will only consider CSPs with symmetric constraints, i.e. for all $v, w \in V$ and $x \in D_v, y \in D_w$ we have $R_{vw}(x, y) = R_{wv}(y, x) = R^*_{wv}(y, x)$. For this reason we will avoid saying $[v, x]$ *constrains* $[w, y]$ since this induces a direction, but rather say $[v, x]$ *and* $[w, y]$ *are in conflict*. Since constraints are assumed to be symmetric, arc-irreducibility becomes edge-irreducibility according to our notation.

Since AC-3 is not based on the concept of support, we can rewrite Revise and use AC-3 to achieve edge-irreducibility. For classical CSP, Revise takes $O(d^2)$ time. For our purpose, however, its task becomes more involved. Given two variables v, w, Revise must check whether v is $\{v, w\}$-reducible. To decide whether we can remove a value x from D_v, for each solution π of $\{v, w\}$ with $\pi(v) = x$ we must find a solution π' of $\{v, w\}$ with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. $\pi'(w)$ must either equal $\pi(w)$ or not be in conflict with any values of variables in $V \setminus \{v, w\}$. For $\pi'(v)$ the latter condition must hold.

Using brute force, we could do the following. For each of the $O(d)$ values x of v, we enumerate each of the $O(d)$ possible solutions π_y that map v to x and w to some $y \in D_w \cup \{0\}$. For each π_y we search for a solution π'_y that maps v to a value $x' \neq x$ and fulfills the conditions stated above. To find such a solution π'_y we go through all $O(d^2)$ pairs of values (x', y') with $x' \in (D_v \cup \{0\}) \setminus \{x\}$ and $y' \in D_w$. For each pair we check $R^*_{vw}(x', y')$ and whether x' and y' (if $y' \neq y$) are not in conflict with any values of variables in $V \setminus \{v, w\}$. If this test can be done in constant time Revise requires at most $O(d^4)$ steps. Then AC-3 achieves edge-irreducibility in $O(d^5 e)$ time.

Clearly this rough estimate can only serve as an upper bound. We can definitively do better. Our approach is as follows. We give a list of three rules,

each of which consists of the description of a certain conflict situation and a recipe of how to resolve it. We show that (a) only redundant values, i.e. values that prove the reducibility of a variable, are removed, (b) if all rules are applied exhaustively, the remaining instance is edge-irreducible, and (c) the application of each rule takes $O(d^2)$ time. Given Lemma 2.6, (a) implies that the size of the optimal solution remains the same until arc-consistency is achieved.

Let v and w be two variables in V, $v \neq w$. For each of the three rules below there is a figure depicting a typical situation in which the rule applies. In Figures 2.4 to 2.6 variables are represented by boxes and their values by circles. Conflicting values are connected by edges. Short line segments not ending in a circle indicate that the value from which they emanate might constrain further values possibly of other variables. The values that are removed after applying a rule are indicated by dotted circles.

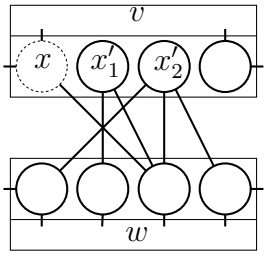

Figure 2.4: rule A1

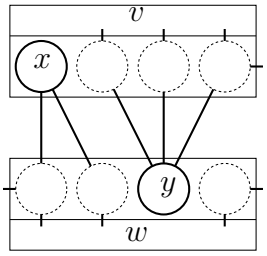

Figure 2.5: rule A2

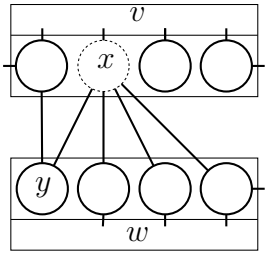

Figure 2.6: rule A3

(A1) If there is a value $x \in D_v$ and a subset $X \neq \emptyset$ of $D_v \setminus \{x\}$ such that all $x' \in X$ are at most in conflict with values of w, and for each value y of w that x does not constrain, there is a value $x' \in X$ that does not constrain y either, then remove x from D_v.

Special case ($X = \{x'\}$): If x' is only in conflict with values of w and those form a subset of the values that are in conflict with x, then remove x from D_v.

Special sub-case ($X = \{x'\}$ and x' has no conflicts): Then remove all values $x \neq x'$ from D_v. (This rule yields node-irreducibility.)

(A2) If there are values x and y of v and w, respectively, that are not in conflict with each other and with values of variables other than v and w, then set $D_v = \{x\}$ and $D_w = \{y\}$.

(A3) If there is a value $x \in D_v$ that is excluded by w, and there is a value $y \in D_w$ that is only in conflict with values of v, then remove x from D_v.

Special case ($D_w = \{y\}$): If y is only in conflict with values of v, then remove all these values from D_v.

Lemma 2.8 *If any of the rules A1 to A3 are applied to two variables v and w, only $\{v, w\}$-redundant values are removed from the domains of D_v and D_w.*

Proof. Given the situation described in rule A1, we have to show that x is $\{v, w\}$-redundant. Let π be any solution for v and w that maps v to x. If A1 is applicable there is a subset $X \neq \emptyset$ of $D_v \setminus \{x\}$ that contains a value x' that is only in conflict with values of w, but not with $\pi(w)$. (For $\pi(w) = 0$ this is true for any $x' \in X$.) Then $\pi'(v, w) = (x', \pi(w))$ is a solution of the same size as π, and x is redundant.

For A2 we can argue as follows. Since $\{x, y\}$ is a complete solution for $\{v, w\}$, it is obvious that all other values of v and w are redundant before applying A2.

Considering A3, a solution π for v and w that maps v to x must map w to 0, hence it cannot be larger than a solution π' that maps w to y. This shows that x is redundant. ❑

Lemma 2.9 *After rules A1 to A3 have been applied exhaustively to an instance $(V, \mathcal{D}, \mathcal{R})$, the resulting instance $(V, \mathcal{D}', \mathcal{R})$ is edge-irreducible.*

Proof. If $|V| < 2$ then there is nothing to show; arc-consistency is defined for pairs of variables. (Still, the special sub-case of A1 would have removed all but one value of the only variable, and the resulting instance would then be (node-)consistent.) Thus let $|V| \geq 2$. We assume that there is a subset $W = \{v, w\} \subseteq V$ and that v is W-reducible in $(V, \mathcal{D}', \mathcal{R})$. Then, due to the definition of reducibility, there is a value $x \in D_v$ such that for all solutions π of W with $\pi(v) = x$ there is a solution π' of W with $\pi'(v) \neq x$ and $|\pi'| \geq |\pi|$. $\pi'(w)$ must be either equal to $\pi(w)$ or not in conflict with any values of variables in $V \setminus W$. For $\pi'(v)$ the latter condition must hold. We have to consider the following three cases.

Case 1: $D_v = \{x\}$

All $y \in D_w$ must be in conflict with x, otherwise there would be a solution π of W with $\pi(v) = x$ and $|\pi| = 2$, and all solutions π' of W with $\pi'(v) \neq x$ would have size at most one, contradicting our assumption.

Thus x is excluded by w, and all $y \in D_w$ must be in conflict with values of variables $u \in V \setminus W$, otherwise we could have applied A3. Then, however, a solution π with $\pi(v) = x$ has size one, while all solutions π' with $\pi'(v) \neq x$ have size zero, since $\pi(w) \neq \emptyset$ would be in conflict with values of variables in $V \setminus W$. Thus x is not W-redundant, which yields the contradiction.

Case 2: $|D_v| \geq 2$ and there is a value $y \in D_w$ that is not in conflict with x.

Then for each such y there is a solution π_y of W with $|\pi_y| = 2$, namely $\pi_y(v, w) = (x, y)$. Due to our assumption for each π_y there must be a solution π'_y of W with $\pi'_y(v) \neq x$ and $|\pi'_y| = 2$. $\pi'_y(w)$ must be y, and y must constrain values of variables in $V \setminus W$, otherwise A2 would have applied and all values of v and w (among them x) would have been removed—except $\pi'_y(v)$ and $\pi'_y(w)$. Let $X = \{\pi'_y(v) \mid y \in$

D_w and y is not in conflict with $x\}$. The condition of case 2 guarantees that $X \neq \emptyset$. All $x' \in X$ are at most in conflict with values of w; this is due to the restrictions imposed on each solution π'_y. In this situation, however, A1 would have been applicable: for each value y of w that is not in conflict with x we have an $x' \in X$ that does not constrain y since x' and y are both part of solution π'_y. Thus x would have been removed from D_v, which contradicts our assumption.

Case 3: $|D_v| \geq 2$ and x is excluded by w.

Then a solution π of W with $\pi(v) = x$ has size one. Due to our assumption there must be a solution π' of W with $\pi(v) \neq x$ and size at least one. Suppose $x' := \pi'(v) \neq 0$. Then x' is at most in conflict with values of w, and those form a subset of the values that are in conflict with x. Thus the special case of A1 would have applied and x would have been removed.

Hence $\pi'(v) = 0$ and $y := \pi'(w) \neq 0$. In this case, however, A3 would have applied and x would have been removed from D_v, contradicting our assumption.

□

Lemma 2.10 *Suppose there is an oracle that answers question of the type* "Given two variables v and w and a value $x \in D_v$, does x constrain at most values of w?" *in $O(d)$ time, and suppose that the predicate $R_{vw}(x, y)$ can be evaluated in constant time for any $x \in D_v$ and $y \in D_w$, then applying any of the rules A1 to A3 to a pair of variables $\{v, w\}$ requires at most $O(d^2)$ time.*

Proof. Let v and w be the two variables under consideration, and let $\alpha_{vw}(x)$ be the answer of the oracle applied to the variables v and w, and to a value x in D_v. For rules A1 and A3, we show that their application to (v, w) is in $O(d^2)$, then obviously the same holds for (w, v).

Our algorithm for A1 is sketched in Figure 2.7. We assume that D_v and D_w are given as lists and that we can store an integer entry $b(y)$ with each $y \in D_w$. We initialize these entries with zero. Let X be a subset of D_v. Initially X is empty.

Our algorithm consists of two phases. In the first phase we collect in X all values $x \in D_v$ for which $\alpha_{vw}(x)$ is true, and set the entries $b(y)$ for each $y \in D_w$ to the number of $x \in X$ with $R_{vw}(x, y)$ true. The fact that each $b(y)$ equals this number is an invariant of our algorithm. In the second phase we actually remove the values from D_v for which A1 applies.

In phase 1 we go once through all values x of v. If $\alpha_{vw}(x)$ is true, we append x to X and go through D_w incrementing all $b(y)$ for which $R_{vw}(x, y)$ holds. If after this procedure $X = \emptyset$, we cannot remove any value and stop.

In phase 2 we go through D_v once more and test which values $x \in D_v$ we can remove given the entries $b(y)$ of each $y \in D_w$. In order to do so, for each

```
ALGO_A1( v, w; D_v, D_w, α_vw, R_vw )

// phase 1: initialize the data structures X, a(D_v), and b(D_w)
X ← ∅
for each y ∈ D_w do b(y) = 0
for each x ∈ D_v do
    a(x) ← α_vw(x)
    if a(x) then
        for each y ∈ D_w do if R_vw(x, y) then b(y) ← b(y) + 1
        X ← X ∪ {x}
    end
end
// phase 2: remove values from the domain of v
for each x ∈ D_v do
    if a(x) then threshold ← 0 else threshold ← 1 end
    can_remove ← true
    for each y ∈ D_w do
        if R_vw(x, y) and b(y) ≤ threshold then can_remove ← false
    end
    if X ≠ {x} and can_remove then
        D_v ← D_v \ {x}
        if x ∈ X then
            for each y ∈ D_w do if R_vw(x, y) then b(y) ← b(y) − 1
            X ← X \ {x}
        end
    end
end
```

Figure 2.7: The algorithm that implements rule A1.

$x \in D_v$ we go through D_w and check whether X *covers* each $y \in D_w$ that is not in conflict with x, i.e. if there is an $x' \in X$ that is not in conflict with y either. If we want to remove a value $x \in X$, we additionally have to make sure that $X \setminus \{x\}$ covers the same subset of D_w as X. The conditions for $x \in X$ ($x \notin X$) are fulfilled if the entries $b(y)$ for all $y \in D_w$ with $R_{vw}(x, y)$ are greater than 1 (0). If this is the case and $X \neq \{x\}$ holds, then we remove x from D_v. If additionally $x \in X$, we decrement the appropriate entries $b(y)$ and remove x from X. The condition $X \neq \{x\}$ ensures that we do not remove the last value x' of X. This can only happen if x' is excluded by w. Keeping x' in X is necessary to remove—in accordance with the special case of A1—all other values in $D_v \setminus X$ that are excluded by w.

Note that we do not attempt to find the set X with minimal cardinality such that X covers all $y \in D_w$ that are not in conflict with x. This would enable us to remove the maximum number of values x from D_v. However, such an attempt would correspond to solving the set-cover problem, which is

NP-complete in general [Kar72]. One cannot even expect that set cover can be approximated within a factor of $\ln N$, where N is the size of the set to be covered [Fei96]. Our objective is only to remove enough values of v such that no $\{v, w\}$-redundant value of v remains.

For a time bound of this algorithm, observe that we need to ask the oracle $O(d)$ times, and for each of the $O(d)$ values of v we have to go through the $O(d)$ values of w at most three times. This yields a time complexity of $O(d^2)$ as desired. (The necessary operations on the set X can be done in constant time each if X is implemented by Boolean entries associated with each $x \in D_v$ and by a counter that keeps track of the current size of X.)

The algorithm for A1 is correct for the following two reasons. First, whenever we remove a candidate x, the current set X and the current Boolean entries of the values of w constitute a proof guaranteeing that A1 applies: the entry of each $y \in D_w$ that x does not constrain is marked true, thus there is a value $x' \in X$ that does not constrain y either. (If $|X| = 1$, and both x and the only element of X constrain all values of W then we can still remove x according to A1.) Note that we append to X only values x for which $\alpha_{vw}(x)$ is true.

Second, if no value is removed, there are two possibilities. If the algorithm terminates with $X = \emptyset$ then all values of v are in conflict with values of variables in $V \setminus \{w\}$ and A1 does not apply.

If $X \neq \emptyset$, suppose there is a value $x \in D_v$ and a subset $X' \neq \emptyset$ of $D_v \setminus \{x\}$ with the properties required for applying A1. We claim that then our algorithm for A1 then would have removed x from D_v. There are two cases.

Case 1: x is in conflict with values of variables in $V \setminus \{w\}$.

> Then $\alpha_{vw}(x)$ is false, and x is only considered during phase 2. Since we assume that x is not removed, there must have been a value $y \in D_w$ with $R_{vw}(x, y)$ true and $b(y) = 0$. Due to our assumption, there must be a value $x' \in X'$ with $R_{vw}(x', y)$ and $\alpha_{vw}(x')$ true. If this is the case, however, $b(y)$ would have been greater than 0 when our algorithm processed x' during phase 2. The entries $b(\cdot)$ are never decreased from 1 to 0. Thus our assumption is contradicted.

Case 2: x is at most in conflict with values of w.

> Then there must be a value $y \in D_w$ with $R_{vw}(x, y)$ and $b(y) = 1$ that prevented x from being removed in the second pass through D_v. Since $b(y)$ corresponds to the number of values x' in X with $R_{vw}(x', y)$ true, X did not contain such a value except x itself. During the second pass, no new values are added to X, and when the algorithm terminates, X contains all values of D_v with $\alpha_{vw}(x)$ true that have not been removed. Thus X' must be a subset of X. Since $x \notin X'$ there is no value in X' with $R_{vw}(x, y)$, which contradicts the assumption.

The algorithm for A2 is simple. We mark each value of v and w with the answer of the oracle. Then for each pair (x, y) of values of v and w with $\alpha_{vw}(x)$

and $\alpha_{wv}(y)$ true we check whether $R_{vw}(x, y)$ holds. If this is the case, we stop and delete all values of D_v and D_w other than x and y.

Applying A3 is also easy. For each value x of v we go through all values y of w and check $\alpha_{wv}(y)$ and $R_{vw}(x, y)$. If $R_{vw}(x, y)$ is false for all $y \in D_w$ and there is one y with $\alpha_{wv}(y)$ true, then we remove x from D_v.

It is clear that the algorithms for A2 and A3 are correct and do not require more than $O(d^2)$ time. □

Lemma 2.11 *There is an algorithm, EI-1, that given an instance $(V, \mathcal{D}, \mathcal{R})$ of a MVS-CSP achieves edge-irreducibility in $O(d^3e)$ time under the conditions stated in Lemma 2.10.*

Proof. The structure of EI-1 is very similar to that of AC-3. First we put all e edges of the variable conflict graph $G(V, E)$ on a stack Q. While Q is not empty we take an edge $\{v, w\}$ from the stack and call Revise(v, w). Revise applies rules A1 to A3 until no further value of v and w can be deleted. Note that our Revise is symmetric; while the procedure of Mackworth makes the arc (v, w) of the (directed) variable constraint graph $\overrightarrow{G}$ consistent, we make the edge $\{v, w\}$ of the (undirected) variable conflict graph G irreducible.

If Revise eliminates values of v or w, we have to ensure edge-irreducibility of all edges of G that are incident to v and w, respectively—except $\{v, w\}$. Therefore we put these edges on Q and continue by calling Revise for the following edge on Q.

Actually there is another point where EI-1 differs from AC-3 due to the difference between arc-consistency and edge-irreducibility. Edge-irreducibility induces node-irreducibility; but in the algorithm sketched so far we do not take variables without conflicts into account at all. To each of these variables we must apply the special sub-case of rule A1, i.e. we must remove all of its values except one. Obviously this can be done in $O(dn)$ total time given the variable conflict graph.

The algorithm EI-1 is correct for the following reasons. Due to the initialization of Q each edge $\{v, w\}$ is made irreducible at least once. An edge can only become reducible if (a) the domain of v or w changes or (b) a value of v or w loses its last conflict with values of variables other than v and w. Both kinds of changes are triggered by the removal of a value; namely a value of v, w, or of a variable u that is adjacent to v or w in G. In the latter case it is obvious that EI-1 puts $\{v, w\}$ on Q and makes $\{v, w\}$ irreducible again later. If a value of v or w is removed, Revise was called for either $\{v, w\}$, $\{v, s\}$, or $\{t, w\}$, where s is a variable adjacent to v and t a variable adjacent to w in G. Lemma 2.9 guarantees that $\{v, w\}$ is made irreducible since we apply our rules A1 to A3 exhaustively. In the other two cases EI-1 puts $\{v, w\}$ on Q since $\{v, w\}$ is incident to $\{v, s\}$ and $\{t, w\}$, respectively. Later, when EI-1 takes $\{v, w\}$ from Q, the irreducibility of $\{v, w\}$ is reestablished.

The time bound of $O(d^3e)$ that Mackworth and Freuder [MF85] gave for

AC-3 applies to EI-1 as well. In Lemma 2.10 we proved that one application of rules A1 to A3 costs $O(d^2)$ time given the oracle mentioned there. Suppose we had such an oracle. We call an application of A1 to A3 *successful* if it leads to the removal of a value of at least one of the two participating variables. The rules are applied at most dn times successfully and for each edge $\{v, w\}$ at most $2d + 1$ times unsuccessfully, namely once for the edge we put on Q during initialization, and once for each of the $2d$ values we potentially remove from v and w. We can assume that G is connected otherwise we can treat each component separately. Thus $e \geq n - 1$, and the runtime of EI-1 sums up to $O(e+(dn+2de)\cdot d^2) = O(d^3e)$ if we assume the existence of the oracle. □

It would be simple to implement the required oracle to run in $O(d)$ time if we were willing to accept a storage consumption of $O(d^2e)$. In this case we could compute explicitly the value conflict graph $G_{\mathcal{D}}$, see Definition 2.4. Computing $G_{\mathcal{D}}$ from the given variable conflict graph costs $O(d^2e)$ time and space. Recall that the oracle $\alpha_{vw}(x)$ has to tell whether the value x of the variable v is only in conflict with values of the variable w. Given $G_{\mathcal{D}}$ the oracle's answer is "no" if the length of the adjacency list of $[v, x]$ is greater than the size of the domain of w. Otherwise the adjacency list of $[v, x]$ is short. Thus we simply have to check to which variable each entry of the list refers and answer "yes" if each of these variables is w, "no" otherwise. Since the domain of w has at most d elements, the oracle's answer can obviously be determined in $O(d)$ steps.

However, for large values of d (and n) such an approach would consume too much storage. Instead, we take advantage of ideas that Bessière used in order to speed up AC-6 and to lower its space requirements as compared to AC-4 [Bes94]. As mentioned before, AC-6 stores at most *one* support for each arc-value pair, while AC-4 stores *all* supports. Recall that a value x of a variable v has support on an arc $(v, w) \in \overrightarrow{E}$ if there is a $y \in D_w$ with $R_{vw}(x, y)$ true. If x has no support on (v, w), x cannot be in the solution of a classical CSP and is therefore removed from the domain of v.

The data structure that AC-6 uses to keep track of which value has support on which arc works as follows. For each arc-value pair $[(v, w), x]$ with $x \in D_v$ and $(v, w) \in \overrightarrow{E}$, AC-6 keeps a list $S_{v,x}$ of all variable-value pairs that $[v, x]$ supports. If x is removed from the domain of v, all $[w, y]$ in $S_{v,x}$ must get new support. If it turns out that there is none, y must be removed from the domain of w. The other "trick" Bessière introduced is that he does not go through all values of v when looking for new support for $[w, y]$. He observes that it is useless to check values of v that have been checked before. Instead, he assumes that the domains are given as lists, i.e. with an arbitrary but fixed total order, and only checks those values z of v that succeed x in the domain list of v. Thus, for each arc-value pair $[(w, v), y]$, he can bound the time required for searching new support for y by $O(d)$. Since there are $O(de)$ arc-value pairs, his support data structure can be initialized and maintained in $O(d^2e)$ total time using $O(de)$ space. The following lemma shows how we can use these ideas for a space- and time-efficient oracle data structure for EI-1.

Lemma 2.12 *There is a data structure that implements the oracle of Lemma 2.10 for EI-1. It takes $O(de)$ storage and can be maintained in $O(d^2e)$ total time during the execution of EI-1.*

Proof. For each edge-value pair $[\{v, w\}, x]$ with $\{v, w\} \in E$ and $x \in D_v$, we keep a witness (i.e. a kind of support) $[u, z]$ for the answer "no" of the oracle. The witness testifies that the value x of v is in conflict with the value z of a variable $u \neq w$. Like in Bessière's case, it is enough to have one such witness per edge-value pair, and it is useless to check twice whether a value is a witness for a given edge-value pair. Thus we can apply his ideas.

We keep a list $W_{u,z}$ with all variable-value pairs for which z is a witness. In addition, we store a Boolean entry with every edge-value pair $[\{v, w\}, x]$ that encodes $\alpha_{vw}(x)$, the answer of the oracle. Suppose all these entries are correct before we remove the value z of a variable u. After the removal, for each $[v, x]$ in the list $W_{u,z}$ we must find a new witness or change its Boolean entry if no further witness exists. We can do this exactly as Bessière's search for new support, i.e. in $O(d)$ time. The initialization is similar to his as well — except that we do not remove values without witness, but only change their Boolean entry. The Boolean entries require $O(de)$ storage; so do the lists of type $W_{u,z}$. Thus our witness data structure can be maintained in $O(d^2e)$ total time using $O(de)$ space. □

Now it is clear that EI-1 requires no more than $O(de)$ storage. Combining Lemmas 2.11 and 2.12 yields our main result concerning the algorithm EI-1.

Theorem 2.13 *Given an instance $(V, \mathcal{D}, \mathcal{R})$ of a MVS-CSP, the algorithm EI-1 achieves edge-irreducibility in $O(d^3e)$ time if all predicates R_{vw} in $\mathcal{R}$ can be evaluated in constant time for any $v, w \in V$, $x \in D_v$, and $y \in D_w$. EI-1 requires $O(de)$ storage.*

2.5 A General Label-Placement Algorithm

In this section we suggest a new algorithm for the general label-number maximization problem. Our algorithm is a combination of EI-1 and a heuristic that removes additional candidates. The heuristic chooses a candidate c according to the *conflict number* of c, i.e. the number of candidates of other features that c intersects.

Our algorithm is simple, but has turned out to be very effective. In Section 3.2 we give experimental results obtained in the context of point labeling. Our hope is that EI-1* or other efficient algorithms based on higher degrees of irreducibility will substitute simulated annealing and other iterative methods of gradient descent for the wide variety of problems that fit into the framework of maximum variable-subset constraint satisfaction.

We proceed as follows. Given an instance of a maximum label-number problem $(F, \mathcal{D})$, where F is a set of n features and $\mathcal{D}$ contains a set D_f of at

most d candidates for each feature $f \in F$, we first compute the *candidate conflict graph* G_{cand}, the equivalent to the value conflict graph $G_{\mathcal{D}}$ in Definition 2.4. In G_{cand} there is a vertex for each feature-candidate pair and an edge for each pair of intersecting candidates that belong to different features.

We use G_{cand} to maintain the conflict number for each candidate. Our invariant is that the conflict number of a candidate c is always equal to the degree of c in G_{cand}, i.e. to the number of edges incident to c. If we remove c from our label-placement instance, we decrement the conflict number of all candidates whose vertices are adjacent to that of c. Then we delete the vertex $v(c)$ corresponding to c and the edges incident to $v(c)$ from G_{cand}.

Recall that REVISE needs constant-time access to the relation R_{fg} for each pair of candidates of f and g. If an intersection test for a pair of candidates can be done in constant time, we can compute the candidate conflict graph in $O((dn)^2)$. It requires $O(k)$ space where k is the number of edges in the graph. If we are given the *feature conflict graph* (the equivalent to the variable conflict graph in Definition 2.4) we can use a data structure similar to the witness data structure suggested in the proof of Theorem 2.13. With this data structure we can initialize and maintain the conflict number of all candidates in $O(d^2e)$ time using $O(de)$ space. For large values of d this is better than transforming the feature conflict graph into a candidate conflict graph that requires $O(k) \subseteq O(d^2e)$ space.

This observation is useful for high-quality point labeling if each point has a large set of candidates and each candidate is an axis-parallel rectangle[1]. If for each point the union of its initial label candidates forms an axis-parallel rectangle[1], one can compute the feature conflict graph in $O(e+n\log n)$ independently of d. An alternative would be to allow an infinite number of candidates and use algorithms for so-called slider models, see Section 3.3.

In case label candidates have more complex shapes with at most s edges and an intersection test needs $f(s)$ time for some function f, computing the candidate conflict graph takes $O(f(s)(dn)^2)$ time in general. To ensure fast access to R_{fg}, the graph can be stored in an adjacency matrix. This requires $O((dn)^2)$ space. However, a careful revision of the proof of Lemma 2.10 shows that constant-time access to R_{fg} is only needed in loops over all pairs of candidates $[b, c]$ of f and g, respectively. Thus representing $G_{\mathcal{D}}$ by ordered adjacency lists suffices. Since time and space for constructing and storing the conflict graph are application-dependent, we assume for the following time and space bounds that G_{cand} is given.

Next we interpret the maximum label-number problem $(F, \mathcal{D})$ as a MVS-CSP $(F, \mathcal{D}, \mathcal{R})$ by identifying each feature with a variable and each candidate with a value. We set $R_{fg}(b, c)$ to false if candidate b of feature f overlaps candidate c of feature g ($g \neq f$), or if this combination is not desired due to some other application-dependent restriction.

Given $(F, \mathcal{D}, \mathcal{R})$, we use EI-1 to achieve edge-irreducibility, then remove a

[1] there are similar results for other shapes of constant complexity

candidate c of some feature f by means of a heuristic, and call REVISE(f, g) for each feature g that was in conflict with f. This process is repeated until each feature has at most one candidate left and no candidates are in conflict any more.

We suggest the following two heuristics for determining the candidate c that is to be removed next. Both base their decision on the conflict number of c.

RemoveTroubleMaker removes the candidate with the greatest conflict number, either locally, i.e. among the candidates of the current feature, or globally. For the local version, the next feature either is the successor of the current feature in a list containing all features, or it can be a feature that still has the maximum number of candidates (MaxCandNumber).

TakeGoodChild does in a sense the opposite of what RemoveTroubleMaker does. Among the candidates of the current feature f this heuristic selects the candidate c with the smallest conflict number, puts c in the solution, and removes all other candidates of f and all those that intersect c. Again, the search for c can be local or global. For the local version, the next feature is either the successor of f in F or a feature with the minimum number of candidates.

Let EI-1* be the algorithm that combines EI-1 with RemoveLocalTroubleMakerMaxCandNumber, i.e. the local version of heuristic RemoveTroubleMaker and selection according to MaxCandNumber. EI-1* operates on a given candidate conflict graph.

Using no data structures other than doubly connected lists, EI-1* requires $O(d^2n)$ total time to repeatedly select the next candidate to be removed. Removing all of these candidates can cause at most $O(de)$ unsuccessful applications of the rules A1 to A3, using ideas and terminology of the proof of Theorem 2.13. Since the rules are applied at most $O(nd)$ times successfully, and each application requires $O(d^2)$ time, EI-1* has a time complexity of $O(d^3e)$.

We conclude with the following lemma. It is simple, but important for applying the concept of edge-irreducibility in practice. A formal proof is omitted since the basic ideas have been sketched above. The proof would use Lemma 2.9 and Lemma 2.10.

Lemma 2.14 *Given an instance of a maximum label-number problem $(F, \mathcal{D})$, where F is a set of n features and $\mathcal{D}$ contains a set C_f of at most d candidates for each feature f in F, and given the corresponding candidate conflict graph G_{cand}, there is an algorithm, EI-1*, that finds a solution π of $(F, \mathcal{D})$ in $O(d^3e)$ time and requires $O(de)$ space, where e is the number of pairs of features with conflicting candidates.*

Let the predicate $R_{fg}(b, c)$ be true iff candidate b of feature f does not intersect candidate c of feature g, and let $\mathcal{R}$ be the set of predicates R_{fg}, one for each pair $\{f, g\} \subseteq F$ of features. Then π is optimal if for the MVS-CSP $(F, \mathcal{D}, \mathcal{R})$ edge-irreducibility implies $|F|$-irreducibility.