Chapter 5

Dual, Primal, and Full approach

One never goes further than his dreams.
Folklore

People understand and remember concepts easier when these concepts can be associated with graphic illustrations. The successful illustrations are also responsible of how the people interpret and use the corresponding concepts. A simplistic image will help many to easily use the represented features, but will impede others from fully taking advantage of the features that are striped. Too complex representations should be reserved to a few advanced users. This applies to CSP representations. In this chapter we will learn the main interpretations of the CSP formalism. Complex ones open the way for several contributions in this thesis. Several recent techniques for distributed problems, and especially the Asynchronous Aggregation Search, are based on the Full approach described in this chapter. The chapter also details several techniques that are sequential ancestors of the Replica-based Multiply Asynchronous Search.

5.1 Graphical representations and their drawbacks

Three graphic representations are extensively used for displaying CSPs. They are: the primal graph (leading to the primal approach), the dual graph (for the dual approach), and the hyper-graph.

5.1.1 Primal Approach

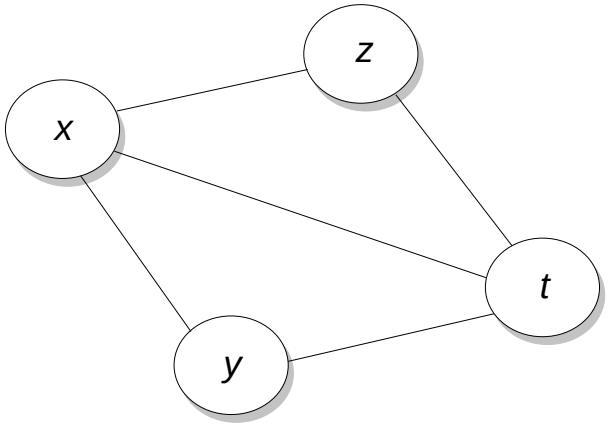

Figure 5.1: Primal graph representation for binary CSPs.

The primal approach to constraint satisfaction consists in focusing most of the attention on variables and restricting the interest on constraints to their indication of directly related variables.

A CSP is represented as a graph where the variables label nodes. The constraints are represented as arcs between nodes and are not labeled. The fact that two predicates may link the same variables is not given attention, but only the total constraint is drawn (Faltings 1994). Within this type of approach, n-ary constraints can be represented as hypergraphs, illustrated in Figure 5.2 (Tsang 1993), but the primal graph is associated with binary CSPs (Figure 5.1). According to (Tsang 1993), the primal graph of a problem whose hypergraph corresponds to Figure 5.2, only illustrates adjacent nodes as shown in Figure 5.3.

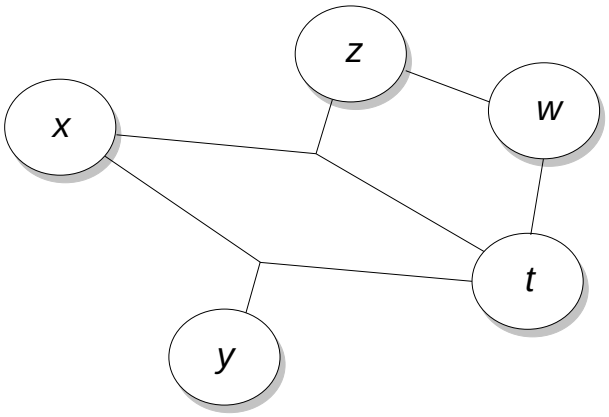

Figure 5.2: Constraint hypergraph for n-ary CSPs.

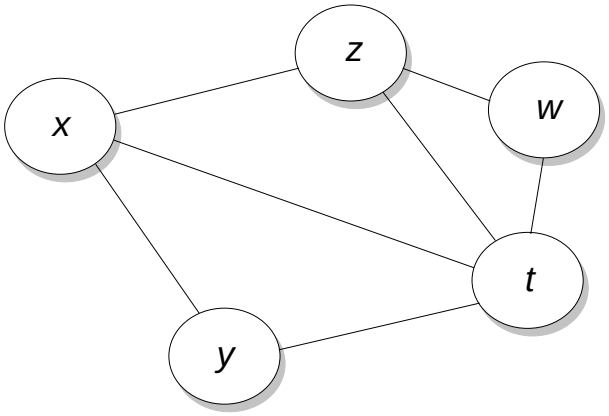

Figure 5.3: Primal graph representation for n-ary CSPs.

Many algorithms have been designed within the philosophy of the primal representation. These algorithms pay special attention to variables, while the constraints are indexed only using the pairs of variables that they involve. No special attention is given to the pattern of n-ary constraints.

5.1.2 Dual Approach

The dual representations reduce non-binary problems to binary ones. The idea is to transform each initial constraint, c, into a variable whose domain consist of the feasible tuples of c. The resulting problem can be modeled with binary constraints, ensuring that the initial variables take the same values in the tuples assigned to each pair of new variables. For the problem depicted in Figure 5.2, the dual graph representation is depicted in Figure 5.4. While within the dualism defined for linear programs, the dual of the dual is the primal (see Annex A), this doesn't hold any longer for the dualism of CSPs.

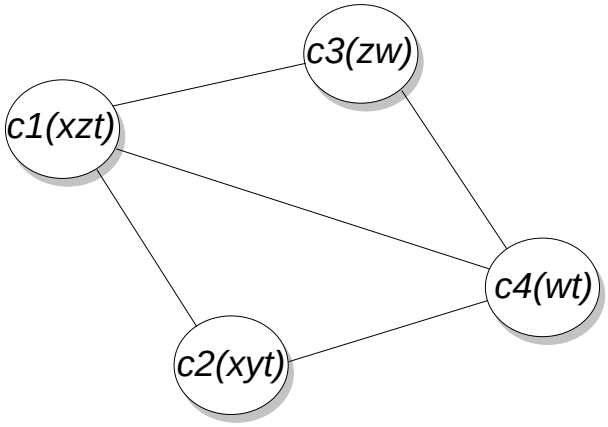

Figure 5.4: Dual graph representation for CSPs.

Once a CSP, P, has been transformed into its dual representation, P', any algorithm for binary CSPs can be directly applied to P'. (Bacchus & Van Beek 1998) has analyzed whether it is better to run an algorithm on the primal or on the dual representation of a problem, and has found that no clear rule can be established.

5.1.3 Hidden variable

The hidden variable encoding (Bacchus & Van Beek 1998; Rossi *et al.* 1989; Dechter 1990) is a technique for transforming n-ary to binary CSPs. It is related to the hypergraph (Tsang 1993) formulation. The hidden variable representation offers a simple solution for representing uniformly both: variables and constraints. It transforms the initial constraints into variables, as the dual representation does, but also keeps the initial variables. The constraints of the hidden variable encoding only link initial variables to new variables, defining a bipartite graph. The obtained graph of the problem in Figure 5.2 is given in Figure 5.5.

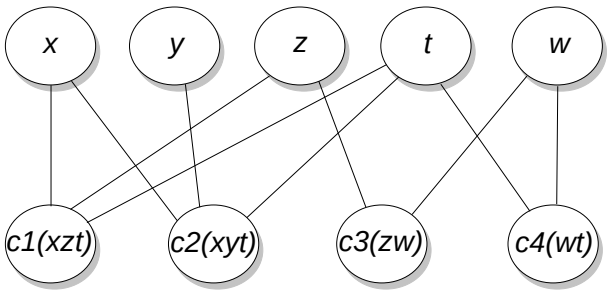

Figure 5.5: Some interpretations of the Hyper graph representation for CSPs. Two constraints on the same tuple of variables cannot be distinguished.

(Bacchus & Van Beek 1998) describes algorithms that theoretically perform exponentially better on the hidden variable encoding. It has to be noted that the hidden variable encoding of a hidden variable encoding representation keeps increasing strictly monotonically the number of variables. It can be inferred that it will not always be beneficial to apply this transformation to a CSP.

In contrast to hidden variable encoding, hypergraphs (Tsang 1993) make a difference between hyperedges (the initial constraints) and nodes (the initial variables). However, (Tsang 1993) defines a hyperedge as a set of nodes, obtaining therefore the representation in Figure 5.2.

5.2 Full Approach

I propose to take in the following an existing but rare approach. In this approach to CSPs, we will be able to address each constraint separately, as it is done in the hidden variable encoding (Section 5.1.3). However, we do not transform constraints into variables. We take an approach similar to (Tsang 1993), with the difference that a hyperedge is defined as a node with an attribute that consists of a set of nodes. This does not modify the space requirements of the primal representation.

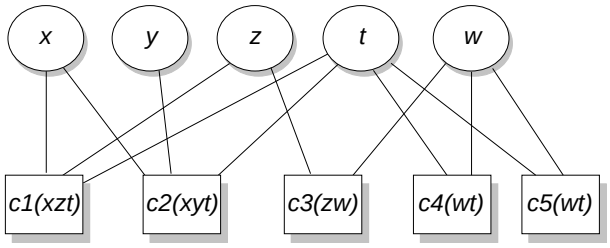

Figure 5.6: Hypergraph in full approach to CSPs. The difference to the Hidden representation is that constraints are not treated as new variables, even if their structure is taken into account. The difference to some Hypergraph interpretations is that constraints on the same variables can be distinguished (with a label).

The next definitions are slightly modified variants of definitions found in the work of Dechter and Pearl.

Definition 5.1 *A hyper-graph (also called full-graph) is defined as a tuple* $(\mathcal{V}, \mathcal{R})$ *where* $\mathcal{V}$ *is a set of nodes and* $\mathcal{R}$ *is a set of ridges (i.e. a set of links, each of them has an attribute,* ends*, that specifies a set of nodes). A pair of nodes found in the ends of the same ridge are called adjacent.*

Definition 5.2 *A constraint hyper-graph (full-graph) of a CSP (X,P) is a full-graph in which each node represents a variable in X and each ridge represents a constraint in P.*

The full-graph for the problem in Figure 5.2 is given in Figure 5.6. This approach to CSPs is already the standard for numeric problems (Benhamou *et al.* 1994; Tanimoto 1993). Its advantage is that in search and reduction, constraints can be treated in special ways according to their peculiarities:

- sum constraints (Régin 1996),
- all-diff constraints (Puget 1998),
- equality/inequality (Silaghi *et al.* 2001j)

Algorithms designed with the constraint full-graph representation in mind can dynamically take decisions (reordering, splitting strategies) about issues related to both variables and remaining unsatisfied constraints.

The generic Algorithm 13, called Full Backtracking (FBT), is the most general algorithm comprising any depth-first search sequential technique described in this thesis. Variants that are not necessarily depth first can be obtained by reformulating FBT to an iterative equivalent. An example is given later in this chapter. Among novel features of Algorithm 13 please note:

- Algorithm 13 treats each node in the search tree as a stand alone problem.
- Algorithm 13 publishes solutions when P is unconstrained. A related instance is described in (Tanimoto 1993).

```
procedure FBT (P, Q) do
    //unconstraint checks that all constraints marked 'relevant' are unary;
    //P is a descriptor of a search node (available values and relevant constraints);
    //Q is the list of parent search nodes descriptors, empty on the first call;
    //[P|Q] (notation inspired from Prolog) denotes a list with head P and tail Q;
    //P.labels is the set of available domains in the search node P;
    if (unconstrained(P)) then
        publish P.labels;
        return;
    //reduction applies some reduction algorithm on the problems;
    //in the set received as parameter;
    reduction([P|Q]);
    while (non-empty(P)) do
        //use strategies to split disjunctive constraints and extract subproblems;
        //not necessarily all subproblems are extracted at once;
        S ← extract-subproblems(P);
        foreach s ∈ S do
            FBT (s, [P|Q]);
            P ← P \ s;
        reduction([P|Q]);
```

Algorithm 13: Generic depth-first Full-graph Backtracking. The 'irrelevant' mark for constraints can be set in either reduction(), extract-subproblems(), or unconstrained().

- Algorithm 13 applies reductions for ancestor super-problems (the reduction procedure reduces a hierarchical set of problems). Namely, a conflict detected deep in the search is applied first for reductions at the closest possible level to the root of the search. This cheaply propagates to further levels avoiding redundant work. We have first developed this for distributed consistency maintenance (Silaghi *et al.* 2001f; 2001l). A related instance is (Baget & Tognetti 2001).
- Algorithm 13 treats enumeration as a set of binary splits (the reduction is also called in the **while** cycle). A related instance is described in (Tel 1999).
- Algorithm 13 allows general types of problem splitting. Related instances appear in (Tanimoto 1993; Benhamou *et al.* 1994; Larrosa 1997; Silaghi *et al.* 1999a).

Definition 5.3 (irrelevant) *A constraint is irrelevant for a problem when it allows all the tuples in the current search space (cross-product of domains).*

A constraint c is marked 'irrelevant' either when it is checked and found irrelevant for the current problem, or after the problem was specially split in order to get subproblems on which c is irrelevant.

Definition 5.4 (instantiation) *When a problem is split for extracting a subproblem for which c is irrelevant we say that we* instantiate *c.*

5.2.1 Aggregation

This section describes the techniques used by agents for local computations in AAS (Chapter 9). More exactly, I exemplify a problem splitting technique for FBT, where the label assigned to a variable is a discrete interval.

Definition 5.5 (discrete interval) *Let I be a set of values taken from a totally ordered discrete domain D. I is a* discrete interval *if its elements are successive in D.*

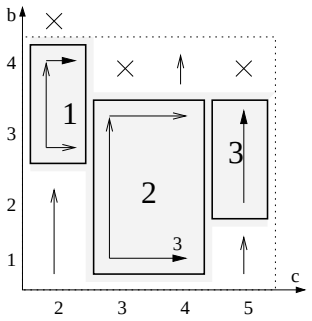

Figure 5.7: *Example of a run of the dynamic aggregation procedure. The arrows show the direction of the scanning. The crosses show the continuation points. The rectangles are the resulting multi-boxes and the big numbers inside show the order in which they are returned. Smaller number in the rectangles show the jump displacement for the scan procedure.*

A discrete interval can be represented extensively by listing all its elements, or intensively using its bounds. The nodes of FBT are characterized by multi-dimensional boxes, called multi-boxes.

Definition 5.6 (multi-box) *Let C be a constraint between a set $V = \{v_1, \ldots v_k\}$ of variables. Let $\{I_1, \ldots I_k\}$ be a set of discrete intervals respectively assigned to the elements of V from their domains. The cross product $I_1 \times \ldots I_k$ is a multi-box of C iff it satisfies C.*

The search technique basically proceeds as follows. A FBT node is created by selecting a constraint, c. It is instantiated to a multi-box of c, computed dynamically. The label of the variables are then restricted to the values allowed by the multi-box. This restriction is further propagated in the problem using a reduction algorithm. If one of the variable domains is emptied, another multi-box is chosen for c. A backtracking is initiated if no new multi-box is found for c. When all the constraints have been instantiated, a set of solutions is obtained. These solutions are given by the cross-product between the interval labels of the variables.

5.2.1.1 Aggregation Procedure

We now present the dynamic aggregation procedure we propose for computing multi-boxes. This procedure scans the extensional representation of a constraint according to an ordering of the variables. This ordering specifies priorities on the directions of aggregation. A progress information structure is used that stores the next starting point for aggregation at each step. Given an order of the variables, at each request for a multi-box, we systematically scan the constraint up to a feasible tuple. The starting point is inherited from the previous call. Once a feasible tuple is found, we try to expand it systematically along all the variables according to the ordering provided by heuristics described in Section 5.2.1.4. The tuples of the obtained multi-box are marked as already covered, so that future scanning and aggregations will skip them. The starting point for the next request is set to the right side of the multi-box, on the direction with the highest priority at scanning. For the scanning procedure to skip wide covered multi-boxes we assign to the left side tuples, the width of the covered multi-box (jump displacement). The ordering heuristic we use tries to maximize the probability that a bloc can be aggregated more on its highest priority direction.

The aggregation procedure is illustrated by the binary example of Figure 5.7. The current domain of the constraint (c, b) is set to the cross-product $[2, 5] \times [1, 4]$. The iterator is set to the position (2,1). Let us consider that the ordering heuristic gives us the order b, c. We scan the constraint as shown by the arrow starting at (2,1) and we discover a feasible tuple at (2,3). We extend it first for b obtaining the first multi-box $[2, 2]\times[3, 4]$. Then we try to extend it for c, operation that fails because of the infeasible tuple (3,4). The continuation point is set to (2,5), and the multi-box 1 is returned.

When a new query for a multi-box is received, the continuation point is read and the scan continues from (3,1), where it also finds a feasible tuple and stops. The obtained multi-box is stretched first for b to $[3,3]\times[1,3]$. Then, it is expanded for c and it becomes $[3,4]\times[1,3]$. We do not try to expand it in the descending direction of a variable. The new continuation point is set to (3,4) and the jump displacement for scan in (4,1) to 3. Then the multi-box 2 is returned. In the same manner, at the third request, the scan begins from (3,4), reads (4,1) and jumps to (4,4). It continues on (5,1) and discovers a feasible tuple at (5,2). The multi-box is expanded for b and the multi-box 3 is delivered. At the fourth request, the continuation point (5,4) is checked and then empty constraint is signaled. The complexity of this algorithm is $O(a * d^a)$, where a is the maximal arity and d is the maximal domain size. The complexity of an alternative algorithm for aggregation (see (Freuder 1991)) was $O(d^a)$ but was requesting more complex data structures, was not depth first, and an additional value reordering would be required in order to provide intervals. A related work performed in the framework of design is described in (D'Ambrosio *et al.* 1996).

5.2.1.2 Cartesian product representation (CPR)

An intuitive way to lower the complexity barrier is to structure the search space so that the exploration algorithm operates on aggregated subsets of data rather than on individual possible instantiations. This is the idea behind the Cartesian product representation (CPR) which aggregates partial solutions during backtracking. The use of CPR was shown to bring improvements, especially to the problem of finding all solutions.

Interchangeability (Freuder 1991) provides a principled approach to simplifying the search space. Values are interchangeable if exchanging one for one another in any solution produces another solution. BT-CPR (Hubbe & Freuder 1992) is one of the first search algorithms using interchangeabilities. It uses a limited form of interchangeability called neighborhood interchangeability (NI) to aggregate partial solutions in the search space. A combination of CPR with FC (FC-CPR) is also described in (Hubbe & Freuder 1992). Two values, a and b of a variable x_i in the CSP (V,C,D) are neighborhood interchangeable iff for any constraint $C_l \in C$ linking x_i and x_{l_j}, the two sets of values from x_{l_j} that satisfy C_l with a, respectively with b are identical. Two values a and b are partially neighborhood interchangeable (PNI) against a set of constraints $S \subset C$ iff they are neighborhood interchangeable in the CSP (V,S,D). We will also say that two values a and b are partially neighborhood interchangeable against a set of variables $N \subset V$ iff they are neighborhood interchangeable in the CSP (N,C',D) where C' is the subset of C linking only variables from N.

FC-CPR maintains at each node a partial solution represented as a Cartesian product. For example, a partial solution involving variables a and b will be represented by a Cartesian product like $(A := \{1,2\}) \times (B := \{3,5,8\})$. The values of the future variables that are compatible with the current partial solution are also structured as Cartesian products. FC-CPR prunes (FC) the domains of the future variables for each possible value of the current one. Based on the obtained active domains for the future variables, it builds a structure called the discrimination tree (DT) (Freuder 1991) that enables a cheap merging of the result into maximal Cartesian products. These Cartesian products correspond to partial neighborhood interchangeable sets of the current variable against all future variables. The children nodes in the search tree are obtained by expanding the current partial solution with any of the obtained interchangeable sets.

We recall that the DT is a tree structure having the values in some of the nodes. The insertion of a new value is performed by following a path from the root of the tree, where each node corresponds to the next feasible tuple containing the value, in the ordered relations of the current variable. If the corresponding node did not previously exist, a new branch (set) is created. The value is placed in the set found in the node at the end of this path. The enumeration of the feasible tuples is not an overhead if it can be reused in the search process, for example when all solutions are looked for.

A variant of FC-CPR is proposed in (Lesaint 1994). It allows for cheaply getting certain additional solutions once the first one has been obtained. However that algorithm is not optimal when we need an arbitrary number of additional solutions. The original FC-CPR offers a more general alternative in that case.

In (Choueiry & Noubir 1998) it was shown that the partial interchangeable sets against subsets of the future variables can be organized in a hierarchy of a tree. This proves helpful in one of the

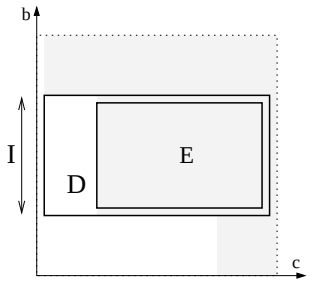

Figure 5.8: *If the variable b was instantiated by an interval I, the domain D where the merging E is performed for c is given by intersecting I with the range of b in the constraint between b and c*

algorithms we present later.

5.2.1.3 CPR in Ordered Domains

We now discuss differences between the approach of (Hubbe & Freuder 1992) using Cartesian product representation (CPR) and the implementation that has just been presented (Section 5.2.1.1). This implementation (without maintaining the consistency) is referred to as FCPR in the rest of the thesis. We first recall how the primal-based CPR works. Suppose that at a given step, $k-1$ variables $x_1, \ldots x_{k-1}$ have already been instantiated and that we want to assign aggregated values to a k^{th} new variable. The current instantiation for $x_1, \ldots x_{k-1}$ is structured as a partial cross-product, b_{k-1}. Each possible value v_i for x_k must be first tested against b_{k-1}, requiring the search for and the representation of cross-products of the type $b^i_{k-1} \times \{v_i\}$. The technique of CPR requires that these representations must be restructured by merging (using a decision tree).

In our case, performing the instantiation directly provides the cross-products. These cross-products are in fact generated one by one and the merging step is implicit. The domains where the implicit merging is performed are determined by simple interval intersection, as shown in Figure 5.8. Moreover, the data management required by explicitly storing all partial cross-products is no more necessary. Performing FCPR in ordered domains also provides a concise representation for a cross-product, which takes the form of a set of intervals.

To summarize, the primal-based implementation of CPR reorders the values in the domains to perform the aggregations. As long as we deal with ordered domains, reordering is not done with FCPR. FMBC1 combines FCPR with the maintenance of Bound-consistency. One can demonstrate that the aggregation obtained by FMBC1 can be stronger than the one performed in a corresponding primal approach (Silaghi *et al.* 1999c).

FCPR offers the possibility of reordering the constraints during search allowing for more flexibility. A full representation can incorporate any order of the values and any order of constraints while in the primal the order of the constraints is restricted by that of the variables.

Let us compare the gain brought by FCPR against BT in the dual (constraint oriented) approach. We note with $C_i(b)$ and $V(b)$ the volume (number of instantiated tuples) of the projection of the cross-product partial solution b on the instantiated variables of the constraint i, respectively the volume of b. The volume of an empty set is 1 by convention. The corresponding gain of FCPR is:

Theorem 5.1 *At each instantiation of a new constraint i, FCPR turns the maximum number of checks for a current cross-product partial solution (cprps) from:*

$$V(cprps)/C_i(cprps) \tag{5.1}$$

to 1 for each active tuple (not yet eliminated by nogoods) in the constraint i.

Proof. With FCPR a tuple is tested once. In the standard tuple by tuple based approaches it would have been tested once for each combination of the values in the unrelated current labels. We mean the labels of all the instantiated variables not involved in the current constraint.

Here we have considered in $V(cprps)$ only the variables that were constrained by the past constraints.

This applies as well for the comparisons with the standard primal-based approaches if the order of testing the constraints is identical. □

From the definition of $C_i(b)$ we see that $V(cprps)/C_i(cprps)$ is always bigger than 1. For finding the first solution there is always a possible overhead since portions of the search space can uselessly be explored.

5.2.1.4 Aggregation Heuristics

The aggregation procedure uses heuristics for assigning priorities to the directions of aggregation.

The heuristic we propose is not designed to produce the multiboxes with the biggest volume. It rather tries to generate the multiboxes that promise the biggest further gain in the number of checks. The orderings on directions of aggregation correspond to orderings for the variables.

Since the efficiency depends on the future constraints, we can use the result of the theorem 5.1 to estimate the gains of each direction of aggregation. We can therefore try to maximize the size of the labels for the variable that promises more.

We estimate that the number of variable values that can be successfully aggregated along one variable v is proportional to the size of the current label of that variable $Size(v)$.

Due to equation 5.1 we can estimate that for any future constraint C_i, for any of the variables v that are not involved in C_i, we will gain with a factor of $Size(v)$. This is the factor with which $V(cprps)$ of the Equation 5.1 increases. If v is constrained by the constraint C_i, then there is no gain at the level of C_i. In our computation we say that for the variable v we have a *loss* of factor $Size(v)$.

One can therefore estimate that the *gain* of expanding the current constraint C_{crt} along direction v is proportional with

$$\sum_{i>crt, v\notin C_i} \frac{Size(v)V(cprps)}{C_i(cprps)}.$$

We have used a cheaper estimation of the *loss* of expanding the variable v as:

$$Size(v)(\sum_{i>crt, v\in C_i} 1). \tag{5.2}$$

Some preliminary tests evaluating the relevance of the technique for pure centralized algorithms are given in Annex D.2.1.

5.2.1.5 Generality of FBT

Let us now analyze the generality of the full approach as compared to the primal approaches.

Theorem 5.2 *For whatever run of a Chronological Backtracking (BT) search on the primal representation with any strategy for variable ordering, there exists a strategy of constraint ordering and value ordering for which FBT performs identically.*

Proof. I show that for each run of a Chronological Backtracking in the primal representation (BT), there exists a run with identical checks of FBT. First we recall that in the primal approach to non-binary constraints(Bacchus & Van Beek 1998), each constraint is checked only when all the involved variables are instantiated. The moment when constraints are verified (marked 'irrelevant') with FBT is completely left to the search strategy.

We will follow now a trace of the BT and see how to build the corresponding FBT algorithm. If, when a variable is chosen to be instantiated in the BT, in FBT we choose to instantiate the constraints in the order in which they are checked in BT, the same checks will always be performed. Indeed, the BT will try combinations in the first constraint until a first one is accepted. The same thing happens in FBT. Then BT passes to the next constraint, and, if the value is not accepted, then it returns to the remaining values for previous one. The same happens in FBT, where after trying to instantiate the next constraint (same check as in BT) we backtrack.

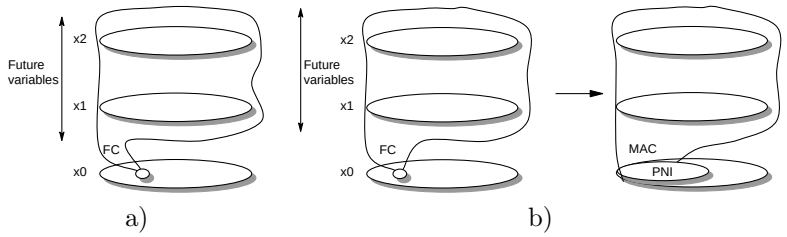

Figure 5.9: a) FC-CPR: Full FC is performed on all future variables for all values in x_0 in order to compute a DT. b) MAC-CPR: Full FC performed on all future variables for all values in x_0 to build a DT, then AC is enforced on all future variables for the chosen PNI set.

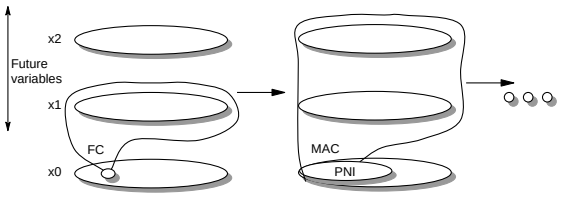

Figure 5.10: QMAC-CPR: FC is performed for all values in x_0 considering only one of the future neighbor variables (here x_1) to build a DT, then AC is enforced for the chosen PNI set on all current and future variables (less the constraints already scanned in order to build the DTs). These two steps are iterated considering one by one, all the future neighbor variables.

When variables in BT can be instantiated with no check, then before passing its first check at the instantiation of a following variable, the order in respect to that variable is insured in FBT by establishing an order of the directions to scan the constraint tables. This way, the checks, as well as their order, are the same in the two algorithms. □

Corollary 5.2.1 *The Chronological Backtracking in Full approach (FBT) is a generalization of the BT in the primal representation.*

Proof. The previous theorem has shown that FBT can emulate at no cost BT. But no standard BT can emulate a FBT that instantiates a new variable before all the constraints between the already instantiated variables are checked. This is possible in FBT. □

5.2.2 Cartesian Product Representation and AC (MAC-CPR)

For using arc-consistency in distributed algorithms, one can use the concepts described in this section. In this section we describe some techniques exploiting the full approach for problems where the domains are not ordered. We now notice two other possible ways of interleaving backtracking on CPR with arc-consistency. Their main characteristics are that they:

- compute the Cartesian products in a fully dynamic way,
- guarantee that the Cartesian products obtained at each step are minimal in number

Figures 5.9 and 5.10, illustrate the main differences between the two studied variants.

It is obvious that the enhancement of FC-CPR with AC should not be done by using AC to prune the future variables for each value of the current variable. The propagations performed by AC will be identical for all the members of a partially neighborhood interchangeable set, and the work would just be replicated. The simplest way to accomplish the task is therefore to enforce AC at each node, i.e. for each interchangeable set computed. The interchangeable sets are computed

dynamically as in FC-CPR. MAC-CPR is obtained by simply performing an AC propagation for each node of FC-CPR before forward check and merging. Once the interchangeable values are obtained, the Arc-Consistent domains of the future variables with any value of an interchangeable set, are consistent with all the other values of that set.

Here I have to specify that during MAC, each time that AC is reinforced there exists an ordering of the AC queue such that a prefix of the process is identical to FC. What one can do in MAC-CPR is to first perform separately this prefix for all values, detecting sets of PNI values. We then perform the rest of the AC with any chosen queue ordering only once for a PNI set.

It is worth mentioning that it may be useful to merge the next levels of nodes immediately after AC is applied to all of them since the pruning can reveal stronger interchangeabilities. QMAC-CPR is an elaborate technique with strong theoretical aggregation power and is detailed in Annex D.2.2.

Remark 5.1 *It is useful to perform arc-consistency at several hierarchical levels for increasingly detailed groupings of PNIs*

- *of a constraint in QMAC-CPR, (resulting algorithm is called QMAC-CPR2) or*
- *of a variable in MAC-CPR, (resulting algorithm is called MAC-CPR2).*

5.2.3 Numerical constraints

An important part of the work described in this thesis is motivated by the need to support numeric constraints in asynchronous techniques for distributed problems with privacy requirements. Now I describe the centralized version of the asynchronous distributed algorithms I propose in Chapter 13.

The finite nature of computers precludes an exact representation of the reals. The set $\mathbb{R}$, extended with the two infinity symbols, and then denoted by $\mathbb{R}^\infty = \mathbb{R} \bigcup \{-\infty, +\infty\}$, is in practice approximated by a finite subset $\mathbb{F}^\infty$ containing $-\infty$, $+\infty$ and 0. In interval-based constraint solvers, $\mathbb{F}^\infty$ usually corresponds to the floating point numbers used in the implementation. Let $<$ be the natural extension to $\mathbb{R}^\infty$ of the order relation $<$ over $\mathbb{R}$. For each l in $\mathbb{F}^\infty$, we denote by l^+ the smallest element in $\mathbb{F}^\infty$ greater than l, and by l^- the greatest element in $\mathbb{F}^\infty$ smaller than l.

A closed interval $[l, u]$ with $l, u \in \mathbb{F}$ is the set of real numbers $\{r \in \mathbb{R} \mid l \leq r \leq u\}$. Similarly, an open/closed interval $[l, u)$ (respectively $(l, u]$) with $l, u \in \mathbb{F}$ is the set of real numbers $\{r \in \mathbb{R} \mid l \leq r < u\}$ (respectively $\{r \in \mathbb{R} \mid l < r \leq u\}$). The set of intervals, denoted by $\mathbb{I}$ is ordered by set inclusion. In the rest of the chapter, intervals are written uppercase, reals or floats are lowercase, vectors in boldface and sets in uppercase calligraphic letters. A *box*, $\mathbf{B} = I_1 \times \ldots I_n$ is a Cartesian product of n intervals. A *canonical interval* is a non-empty interval of the form $[l..l]$ or of the form $[l..l^+]$. A canonical box is a Cartesian product of canonical intervals.

5.2.3.1 Relations and Approximations

Let $c(x_1, \ldots x_n)$ be a real constraint with arity n. The *relation* defined by c, denoted by ρ_c, is the set of tuples satisfying c. The relation defined by the negation, $\neg c$, of c is given by $\mathbb{R}^n \setminus \rho_c$ and is denoted by $\rho_{\overline{c}}$. The global *relation* defined by the conjunction of all the constraints of an NCSP, $\mathcal{C}$ is denoted $\rho_{\mathcal{C}}$. It can be approximated by a computer-representable superset or subset. In the first case the approximation is *complete* but may contain points that are not solutions. Conversely, in the second case, the approximation is *sound* but may lose certain solutions. A relation ρ can be approximated conservatively by the smallest (w.r.t set inclusion) union of boxes, $\mathbf{Union}_\rho$, or more coarsely by the smallest box $\mathbf{Outer}_\rho$, containing it. By using boxes included into ρ, sound (inner) approximations $\mathbf{Inner}_\rho$ can also be defined. In (Benhamou & Goualard 2000), $\mathbf{Inner}_\rho$ is defined as the set $\{r \in \mathbb{R}^n \mid \mathbf{Outer}\{r\} \subseteq \rho\}$.

The computation of these approximations relies on the notion of *contracting operators*. Basically, a contracting operator narrows down the variable domains by discarding values that are locally inconsistent. This is often done using bound consistency. In this chapter we use the notion of outer contracting operator, defined as follows:

Definition 5.7 (Outer contracting operator) *Let $\mathbb{I}$ be a set of intervals over $\mathbb{R}$ and ρ a real relation. The function $\mathbf{OC}_\rho : \mathbb{I}^n \to \mathbb{I}^n$ is a contracting operator for the relation ρ iff for any box $\mathbf{B}, \mathbf{B} \in \mathbb{I}^n$, the next properties are true:*

$$\begin{array}{llcll} (1) & \mathbf{OC}_\rho(\mathbf{B}) & \subseteq & \mathbf{B} & \textit{(Contractiveness)} \\ (2) & \rho \cap \mathbf{B} & \subseteq & \mathbf{OC}_\rho(\mathbf{B}) & \textit{(Completeness)} \end{array}$$

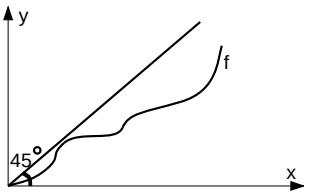

Figure 5.11: A function f in unidimensional cases is *contractive* if for positive arguments it is positive and lies below the identity function.

The contractiveness in unidimensional cases is illustrated in Figure 5.11. Often, a monotonicity condition is also required in some approaches (Granvilliers 1998; Apt 2001). The monotonicity ensures that a *maximal* fix-point is obtained. However, the ideal result is the *minimal* fix-point where the completeness is not lost. Reaching this minimal fix-point (as Hull-consistency (Granvilliers 1998)) is impossible or prohibitive with most common operators (Bound or Box consistency). Figure 5.12 describes this situation. While in general one cannot guarantee that the wished *minimal* fix-point is reached, it is seldom needed to guarantee that the *maximal* fix-point will be obtained.

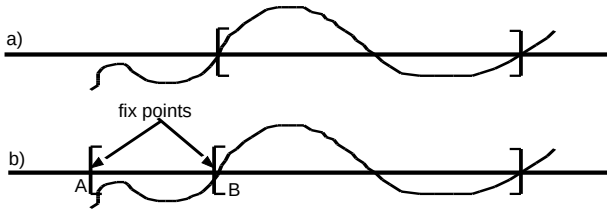

Figure 5.12: a) Monotonicity reaches the unique (wished) fix-point with Hull-consistency. b) With techniques like Box-consistency, monotonicity leads to the worst fixed point, A. Without monotonicity one may reach a smaller complete fix-point (e.g. B).

Let $\mathcal{V}_{\mathbb{R}} = \{x_1 \dots x_n\}$ be a set of variables taking their values over $\mathbb{R}$. Given $\sum_{\mathbb{R}} = \{\mathbb{R}, \mathcal{F}_{\mathbb{R}}, \mathcal{R}_{\mathbb{R}}\}$ a structure where $\mathcal{F}_{\mathbb{R}}$ denotes a set of operators and $\mathcal{R}_{\mathbb{R}}$ a set of relations defined in $\mathbb{R}$, a *real constraint* is defined as a first order formula built from $\sum_{\mathbb{R}}$ and $\mathcal{V}_{\mathbb{R}}$. Interval arithmetic methods (Granvilliers 1998) are the basis of interval constraint solving. They approximate real numbers by intervals and compute conservative enclosures of the solution space of real constraint systems.

5.2.3.2 Algorithms

In the following we describe a couple of algorithms for problems with numeric constraints. Unlike algorithms in Section 2.5, here one deposes more effort at each node for extracting more information at once. This information can be used for generalizing intermediate results. Far related techniques are the Intelligent Backtrackers (Kondrak 1994a; Prosser 1995), the Cross-Product Representation algorithms (Hubbe & Freuder 1992; Haselböck 1993a; Silaghi *et al.* 1999c) (see Section 5.2.2), and the problem splitting (Larrosa 1997).

Actually, we dynamically eliminate constraints that are irrelevant to the current search space. This helps detecting when fully feasible spaces don't need to be split any longer. The algorithms work by saving the detected feasible spaces. There may exist an overhead, but it is quite limited

since we only consider feasible spaces detected as a by-product of one step of the test for constraint relevance given in (Benhamou & Goualard 2000). Using only one step gives the advantage that the result can be simply represented as a single box and is fully reused in further detections.

Due to the philosophy of making effort in advance, as with all intelligent backtrackers and look-ahead algorithms, the algorithms efficiency is improved only for problems above a certain complexity threshold and here only when several backtracks are needed in other competitive algorithms. This situation occurs for hard problems, but also for finding the whole solution space of otherwise simple problems. As argued in (Garey & Johnson 1979), 1.4, finding the whole solution space of a numeric problem can be intractable mostly as a sign that the problem is not realistically defined, since nobody could ever use all the solutions. However, within the settings of the $\varepsilon_1\varepsilon_2\Phi$-consistency defined in Section D.1, a large amount of the search space has to be covered.

```
    procedure UCA6(C = (V, C, D): NCSP) do
    |  P = [[{OC_ρc(D), C, {B_q(OC_ρc(D))}}]];
13.1|  while (getNext(P, C, solution)) do
    |  |  U ← {solution}∪U;
    |  return U;
    function getNext(inout:P = [B = [{B ∈ 𝕀^n, C : NCSP, {B_q ∈ 𝕀^n}} | T_B] | T_P];
    in: C_G ∈ NCSP; out: solution∈ 𝕀^n) → bool
    |  for ever do
13.2|  |  if (B = [ ]) then
13.3|  |  |  if (T_P = [ ]) then
13.4|  |  |  |  return (false);
    |  |  |  else
13.5|  |  |  |  P ← T_P;
13.6|  |  |  continue;
13.7|  |  (C',B',{B_q'}) ← reduc(C, B, {B_q});
13.8|  |  B ← T_B;
13.9|  |  if (B' <> ∅) then
13.10|  |  |  if (C' = ∅) then
13.11|  |  |  |  solution ← B';
13.12|  |  |  |  return (true);
13.13|  |  |  B' ← split(B',C',{B_q'});
13.14|  |  |  P ← [B' | P];
```

Algorithm 14: The Search procedure

We now present an algorithm named UCA6 (Algorithm 14) that computes a **Union** approximation for numerical CSPs with equalities and inequalities.

UCA6 is an algorithm introducing two important concepts:

- Feasibility test: In search nodes, constraints are checked for relevance and completely feasible ones are deactivated to avoid redundant splitting and analysis.

- Dynamic splitting strategy: In search nodes, the split variable and the splitting point is chosen according to the next criteria:

  - The ratio of the size of search space of obtained children nodes must be close to $1 \pm \varepsilon$, for some ε.

  - As much as allowed by the previous criteria, the fragmentation should help isolate search spaces where some constraints are completely feasible.

The feasibility test is chosen in order to be as cheap as possible. To avoid overhead, the intermediary results in each node for a constraint q are saved in case of failure in a structure denoted B_q, which is reused:

- as starting point for feasibility tests in children nodes.
- for deciding splitting points.
- for avoiding tests of some search space borders when contracting-operators are applied to the constraint q. This was not present in the initial UCA6 (Silaghi *et al.* 2001j), being first proposed now, and the improved version obtained from UCA6 by adding this optimization is called UCA6$_2$.[1]

B_q's semantic is the *unexplored box*. A name for B_q used in a few recent publications is: Back-Box. To be noted that in optimization, back-boxing is a different well-known technique (Van Iwaarden 1996) where a Back-Box is a search space with proved sub-optimality.

We note lists in the Prolog style $[Head|Tail]$. $\mathcal{B}$ denotes the list of children to be checked for a node, and $\mathcal{P}$ denotes the list of all $\mathcal{B}$. The algorithm presented is depth-first. Breadth-first and other heuristics can be obtained by treating the lists $\mathcal{B}$ and $\mathcal{P}$ as sets, $\mathcal{P}$ becoming respectively the union of all the sets of type $\mathcal{B}$.

The algorithm UCA6 iteratively calls the function **getNext** which delivers a new **Outer** approximation for a subspace in the solution space. By construction, the new **Outer** approximation will not intersect with any previously computed box.

The function **getNext** has two main components: a reduction operator, **reduc** (Algorithm 15), and a splitting operator, **split** (Algorithm 16). These operators are interleaved as in a classical maintaining bound consistency algorithm. Practically, it is preferable to stop the dichotomous split when the precision of the numeric search tool (splitting and contracting operators) can lead to unsafe solutions at a given precision ε. An unsafe solution is a box that may contain no real solution. **reduc**, checks this state using a function called *Indiscernible*(constraint, **Box**, **OC**, ε), which is not discussed here in detail[2].

function reduc(in: $\mathcal{C}$: NCSP, $\mathbf{B}\in \mathbb{I}^n$, $\{\mathbf{B_i}\in \mathbb{I}^n\}$**)** $\rightarrow$**(**NCSP, $\mathbb{I}^n$, $\{\mathbb{I}^n\}$**)**

14.1 **B'** $\leftarrow \mathbf{OC}_{\rho_{\mathcal{C}}}(\mathbf{B})$;
14.2 **for all** (q={inequality}, q$\in \mathcal{C}$, $\mathbf{B_q} \in \{\mathbf{B_i}\}$) **do**
14.3 **if** ($\mathbf{B'}\cap\mathbf{B_q} <> \mathbf{B_q}$) **then**
14.4 $\mathbf{B_q} \leftarrow \mathbf{OC}_{\rho_{\neg q}}(\mathbf{B'}\cap\mathbf{B_q})$;
14.5 **if** (($\mathbf{B_q} = \emptyset$)$\vee$*Indiscernible*(q,$\mathbf{B_q}$)) **then**
14.6 $\mathcal{C} \leftarrow \mathcal{C} \setminus \{q\}$, $\{\mathbf{B_i}\} \leftarrow \{\mathbf{B_i}\} \setminus \mathbf{B_q}$;
14.7 **for all** (q=equality, q$\in \mathcal{C}$) **do**
14.8 **if** (*Indiscernible*(q,**B'**)) **then**
14.9 $\mathcal{C} \leftarrow \mathcal{C} \setminus \{q\}$, $\{\mathbf{B_i}\} \leftarrow \{\mathbf{B_i}\} \setminus \mathbf{B_q}$;
14.10 return ($\mathcal{C}$, **B'**, $\{\mathbf{B_i}\}$);

Algorithm 15: Problem Reduction

Each search node is characterized by the next structures:

* The list $\mathcal{B}$ corresponds to a set of splits for the current search node. It defines the next branches of search. Each split correspond to a new node of the search.
* A box **B** defining the domains of the current NCSP.

[1] Note that another optimization of the constraint propagation algorithm leads to UCA6+, as described in (Vu *et al.* 2002). These two optimizations can be composed in an algorithm that would be denoted UCA6$_2$+.

[2] The simplest strategy consists of checking that all the intervals are smaller than ε, but more sophisticated techniques can be built by estimating computational errors.

* The current NCSP $\mathcal{C}$ containing only the constraints of the initial NCSP that can participate in pruning the search space. The constraints that are indiscernible or entirely feasible in $\mathbf{B}$ are eliminated.

* Each constraint q in a node is associated with a box, $\mathbf{B_q}$, such that all the space in $\mathbf{B} \setminus \mathbf{B_q}$ is feasible.

Each $\mathbf{B_q}$ is initially equal with the projection of the initial search space on the variables in the constraint q, after applying $\mathbf{OC}_{\rho_{\overline{q}}}$.

One of the features of **reduc** is that it removes redundant completely feasible or indiscernible constraints. If the recent domain modifications of some inequality q have modified $\mathbf{B_q}$, q is checked for feasibility at line 14.4, and eventually removed from the current CSP (line 14.6). Equalities are similarly eliminated at line 14.9 when they become indiscernible.

5.2.3.3 Splitting operator

procedure splitHistory(in:B,$\mathcal{C}$,{$\mathbf{B_i}$};inout:$\mathcal{B} \in [\{\mathbb{I}^n, \text{NCSP}, \{\mathbb{I}^n\}\} \mid \_]$) do
15.1 q ← choose constraint ∈ $\mathcal{C}$;
if *parent split along* B_q **then**
15.2 x ← choose variable of q given $\mathcal{C}$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{r(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{l(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;
else
split($\mathcal{C}$, {$\mathbf{B_i}$}, $\mathcal{B}$);

function split(in: $\mathbf{B} \in \mathbb{I}^n$, $\mathcal{C}$: NCSP, $\{\mathbf{B_i} \in \mathbb{I}^n\}$) → $[\{\mathbb{I}^n, \text{NCSP}, \{\mathbb{I}^n\}\} \mid \_]$
15.3 fun ← choose appropriate(splitFeasible, splitIneq, splitEq);
$\mathcal{B} \leftarrow [\ ]$;
fun($\mathbf{B}$, $\mathcal{C}$, {$\mathbf{B_i}$}, $\mathcal{B}$);
return $\mathcal{B}$;

procedure splitFeasible(in:B,$\mathcal{C}$,{$\mathbf{B_i}$};inout:$\mathcal{B} \in [\{\mathbb{I}^n, \text{NCSP}, \{\mathbb{I}^n\}\} \mid \_]$) do
15.4 q ← choose {inequality}∈ $\mathcal{C}$, $\mathbf{B_q} \in \{\mathbf{B_i}\}$;
15.5 **foreach** *(bound* b *of some variable x of* q *in* $\mathbf{B_q}$ *(e.g. in descending order of the relative distance* rd *to the corresponding bound in* $\mathbf{B}$*))* **do**
15.6 **if** (rd <*frag*) continue;
B' ← $\mathbf{B}_{\mathbf{f(x,b)}[\mathbf{B_q},\mathbf{B}]}$;
B ← $\mathbf{B}_{\mathbf{u(x,b)}[\mathbf{B_q},\mathbf{B}]}$;
$\mathcal{B} \leftarrow [\{\mathbf{B'}, \mathcal{C} \setminus \{q\}, \{\mathbf{B_i}\} \setminus \mathbf{B_q}\} \mid \mathcal{B}\ ]$;
$\mathcal{B} \leftarrow [\{\mathbf{B}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;

procedure splitIneq(in:B,$\mathcal{C}$,{$\mathbf{B_i}$};inout:$\mathcal{B} \in [\{\mathbb{I}^n, \text{NCSP}, \{\mathbb{I}^n\}\} \mid \_]$) do
15.7 q ← choose {inequality, $\mathbf{B_q} \in \mathbb{I}^n$} ∈ $\mathcal{C}$;
15.8 x ← choose variable of q given $\mathcal{C}$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{r(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{l(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;

procedure splitEq(in:B,$\mathcal{C}$,{$\mathbf{B_i}$};inout:$\mathcal{B} \in [\{\mathbb{I}^n, \text{NCSP}, \{\mathbb{I}^n\}\} \mid \_]$) do
15.9 q ← choose {equality} ∈ $\mathcal{C}$;
15.10 x ← choose variable of q given $\mathcal{C}$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{r(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;
$\mathcal{B} \leftarrow [\{\mathbf{B}_{\frac{1}{2}\mathbf{l(x)}[\mathbf{B}]}, \mathcal{C}, \{\mathbf{B_i}\}\} \mid \mathcal{B}\ ]$;

Algorithm 16: Three splitting operators (splitFeasible,splitIneq,splitEq). Two splitting managers (splitHistory,split).

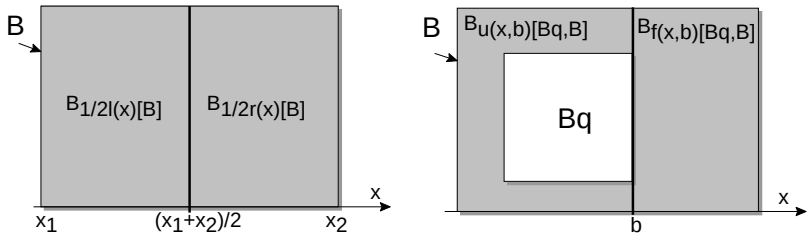

Figure 5.13: Splitting operators. Gray in the second image stands for feasible space.

The example of function **split** given in Algorithm 16 allows for using three splitting strategies. The first one, **splitFeasible**, extracts sound subspaces for some inequality, as long as these subspaces fragment the search space in a ratio limited by a given *fragmentation threshold*, denoted by *frag* (line 15.6). The second and the third strategies (**splitIneq**, respectively **splitEq**), consist of choosing for dichotomous split, a variable involved in an inequality (respectively an equality) of the current NCSP $\mathcal{C}$. The heuristics used at lines 15.7, 15.8, 15.9, and 15.10 in Algorithm 16 can be based on the occurrence of variables in the constraints of $\mathcal{C}$, or according to a round robin technique. The domain of the chosen variable is then split in two halves. Techniques based on the occurrences of variables in constraints can also be used to devise heuristics on ordering the bounds at line 15.5 in **splitFeasible**.

This work has been continued as reported in (Xuan-Vu *et al.* 2002), showing that new splitting heuristics integrated in this technique can lead to even stronger improvements.

The criteria for choosing a constraint at line 15.4 can look for maximizing the size of the search space for which a given constraint is eliminated, minimize the number of children nodes, or maximize the number of constraints that can benefit[3] from the split.

Given two boxes $\mathbf{B}$ and $\mathbf{B_q}$, where $\mathbf{B}$ contains $\mathbf{B_q}$, and given a bound b in $\mathbf{B_q}$ for a variable x, we use the next notations:

* $\mathbf{B_{f(x,b)[B_q,B]}}$ is the (feasible) box not containing $\mathbf{B_q}$ obtained from $\mathbf{B}$ by splitting the variable x in b.
* $\mathbf{B_{u(x,b)[B_q,B]}}$ is the (indiscernible) box containing $\mathbf{B_q}$ obtained from $\mathbf{B}$ by splitting the variable x in b.
* $\mathbf{B_{\frac{1}{2}r(x)[B]}}$ is the (indiscernible) box obtained from $\mathbf{B}$ by splitting the variable x in half and retaining its upper half.
* $\mathbf{B_{\frac{1}{2}l(x)[B]}}$ is the (indiscernible) box obtained from $\mathbf{B}$ by splitting the variable x in half and retaining its lower half.

These concepts are illustrated in the Figure 5.13. In Procedure *splitFeasible*, our described implementation splits the current box q along all borders of B_q that do not fragment the search space with a ratio more than $frag$ (considering the ratio is computed such that it is always less or equal to 1). However, one may choose to split in a step only along the border which offers the highest ratio.

5.3 Summary

In this chapter we have learned about the graph representations for CSPs, as well as about the the binary encodings for n-ary CSPs. The full approach to CSPs is common for numeric problems, but has not been used often for discrete ones. We design a generic backtracking procedure (FBT) that exploits the full approach and show how it generalizes many primal approaches. Some new instantiations of FBT are presented for:

[3]The constraints for which the domains are split may propagate more when **OC** is applied.

- problems with ordered domains (FMBC1),
- extensionally represented discrete problems ((Q)MAC-CPR), and
- numeric problems (UCA6/UCA7)

Part II

Asynchronous Algorithms

In the following part are described most of the new contributions of this thesis. After reading this part you are expected to know:

- The major types of parallelism and examples of applications.
- Several complete asynchronous algorithms.
- Several protocols allowing for families of asynchronous algorithms.