Appendix A

Problem Solving

COMMUTATIVE LAW
No cow's like a horse, and no horse like a cow.
That's one similarity anyhow.
PH

One of the hottest current interests in the distributed community is to design asynchronous algorithms for optimizing multiple criteria, namely criteria required by distinct participating agents. Distributed optimization (e.g. the algorithms presented in Chapters 14 and 17) is more general and complex than distributed constraint satisfaction. In this appendix we introduce general notions and examples from centralized optimization, which throw some light on the current trends in distributed optimization. After improvements, these centralized algorithms allow for parallelizations exploited for efficiency by distributed approaches.

After reading this appendix you will have a general knowledge about optimization and constraint satisfaction techniques. Section A.3 promotes a new version (GOB, of the Generalized Partial Order Dynamic Backtracking (GPB) (Bliek 1998a). GOB saves few additional reusable partial results without modifying the space requirements of GPB (Silaghi *et al.* 1999b).

A.1 Optimization

Many general and well known (especially numeric) algorithms are designed for optimization. In the most general settings, a multi-criteria optimization problem consists of a set of n variables, $x_1, x_2, ..., x_n$, taking their values from the domains $D_1, D_2, ...$, respectively D_n, a set of predicates, $p_1, p_2, ..., p_n$, that have to be satisfied and a set of objective functions, $f_1, f_2, ..., f_n$, whose evaluations have to be optimized over their domains D_i^f. These objective functions are combined using some obscure mechanism g.

Definition A.1 *A centralized multi-criteria optimization problem is defined by:*

$$\{x_i\}, i \in [1..(n_x)], \qquad x_i \in D_i \tag{A.1}$$

$$\{p_i\}, i \in [1..(n_p)], \qquad p_i : D_{i_1^p} \times D_{i_2^p} \times ... \times D_{i^p_{a^i}} \rightarrow boolean \tag{A.2}$$

$$\{f_i\}, i \in [1..(n_f)], \qquad f_i : D_{i_1^f} \times D_{i_2^f} \times ... \times D_{i^f_{b^i}} \rightarrow D_i^f \tag{A.3}$$

$$g, \qquad g : D_1^f \times D_2^f \times ... \times D_{n_f}^f \rightarrow D^g \tag{A.4}$$

Considering that D^g is ordered according to some relation "$\preceq$", the goal is to evaluate:

$$\underset{\{x_1,...,x_{(n_x)}\}}{\text{argmin}} (g(f_1(x_{1_1^f}, ..., x_{1^f_{b1}}), ..., f_{(n_f)}(x_{(n_f)_1^f}, ..., x_{(n_f)^f_{(b^{(n_f)})}}))) | p_i(x_{i_1^p}, ..., x_{i^p_{(a^i)}}), \forall i \in [1..(n_p)]$$

argmin *is interpreted in different algorithms as returning the arguments corresponding to either: a global minima, a local minima, or a heuristically small value.*

As typical example, g can transform its parameters in elements of a tuple. Then "$\preceq$" orders tuples lexicographically.

Example 1.28 *In what concerns markets for cars, some customers have the mechanics as hobby and like to have simple cars that they are able to mend themselves. This is just like those of us that prefer buying compatible PCs or install Linux that we can reconfigure at midnight, as we want. This does not mean that the system should have low quality components. Experienced manufacturers are preferred. Let us consider the example of finding the cheapest and most simple car that is robust to bad roads and whose manufacturer has an experience of more than 50 years. We can model it as:*

$n_x = 1$, x_1 models the car choice, and D_1 =\{ARO-10,FIAT-Panda,Honda-HRV,JEEP-Wrangler,Opel-Frontera,Ford-Explorer\}[1].

$n_p = 2$, $a^1 = 1, i_1^p = 1, p_1 := robust(x_1)$ checks robustness to bad roads, and $a^2 = 1, i_2^p = 1, p_2 := (manufacturer \circ experience)(x_1) > 50$ years.

$n_f = 2$, $b^1 = 1, i_1^f = 1, f_1 := price(x_1)$ models the price of the car in $D_1^f = \mathbb{N} \times \{\$\}$ and $b^2 = 1, i_2^f = 1, f_2 := user{-}maintainability(x_1)$ models the end user maintainability (e.g. the measure in which the user can understand how the car functions: D_i^f = {transparent, partial, fuzzy, indecipherable}).

The function $g(p, c) := \langle p, c \rangle$ models here the order on objective functions. $g(p_1, c_1) \prec g(p_2, c_2)$ iif either $p_1 < p_2$, or $p_1 = p_2$ and c_1 more comfortable than c_2.

Let us consider that the predicates and functions p_i and f_i for a certain user are given by:

car	p_1	p_2	f_1	f_2
ARO	*yes*	*yes*	*10000$*	*transparent*
FIAT	*no*	*no*	*15000$*	*fuzzy*
JEEP	*yes*	*yes*	*30000$*	*fuzzy*
Opel	*yes*	*yes*	*25000$*	*partial*
Ford	*no*	*yes*	*20000$*	*indecipherable*
Honda	*yes*	*no*	*10000$*	*indecipherable*

An algorithm for solving this problem may first find out using constraint satisfaction techniques that only {ARO,JEEP,Opel} satisfy the predicates. Then, using an enumerative technique, one could discover that ARO-10 is the only global optimum solution.

Alternatively, as typically done in practice for most problems, one can use a local optimization algorithm that is quicker but may get stuck in a local minima (e.g. it could return Opel-Frontera if no gradient points from Opel-Frontera directly to ARO-10).

Yet another typical simplification is defined for over-constrained problems but is used in practice for all hard and real-time problems. It replaces all predicates by a new objective function that allows for some heuristic maximization of the number of satisfied predicated. The solution returned by such an incomplete search could be Honda-HRV or even Ford-Explorer.

Centralized multi-criteria optimization problems can be very complex and the seldom existing systematic approaches tend to use techniques for analyzing solution spaces (Benjamin 2000), described in Chapter 5 (e.g. interval programming aka MHC (Silaghi *et al.* 1999a; 2000c)). Several results are available for a simple framework, namely the centralized optimization problems. A centralized optimization problem is a centralized multi-criteria optimization problem where $n_f = 1$. Then, without any loss of generality, g can be considered as being the identity function.

Definition A.2 *A centralized optimization problem is defined by:*

$$\{x_i\}, i \in [1..(n_x)], \qquad x_i \in D_i \tag{A.5}$$

$$\{p_i\}, i \in [1..(n_p)], \qquad p_i : D_{i_1^p} \times D_{i_2^p} \times ... \times D_{i_{a^i}^p} \rightarrow boolean \tag{A.6}$$

$$f, \qquad f : D_{f_1} \times D_{f_2} \times ... \times D_{f_b} \rightarrow D^f \tag{A.7}$$

[1] There is no relation between the real prices and capabilities of existing cars and the data used in this example.

Considering that D^f is ordered according to $\preceq$, the goal is to evaluate:

$$\underset{\{x_i|i\in[1..(n_x)]\}}{\operatorname{argmin}} (f(x_{f_1},...,x_{f_b}))|p_i(x_{i_1^p},...,x_{i_{a^i}^p}), i \in [1..(n_p)]$$

One can argue that the multi-criteria optimization problems are always one criterion optimization for an objective function based on vectors. However, especially in distributed problems where the objective functions optimized by distinct agents is private, it is more natural to reason about multiple criteria. Well known subclasses of one criterion optimization are the Valued Constraint Satisfaction Problems (Valued CSPs) and MAX-CSPs (Schiex *et al.* 1995; Larrosa 1998) which try to reduce the number or importance of unsatisfied predicates from a given set of predicates. In the numeric community, a large amount of studies have concentrated on linear problems, but effort is being made for convex functions, least square minimization, infinite dimensional spaces, infinite number of predicates, etc. (Butnariu & Iusem 1999).

A.1.1 Newton Method

Some simple techniques for numeric optimization derive from the Newton iterative approximation technique (Nocedal & Wright 1999).[2] The Newton Method for the unidimensional case is illustrated in Figure A.1. Given the equation $f(x) = 0$, and a starting value x_0, the Newton Method searches for a solution as limit of an infinite sequence $\{x_i\}_{i>0}$. This sequence is built using the recurrence $x_{i+1} = x_i - f(x_i)/f'(x_i)$.

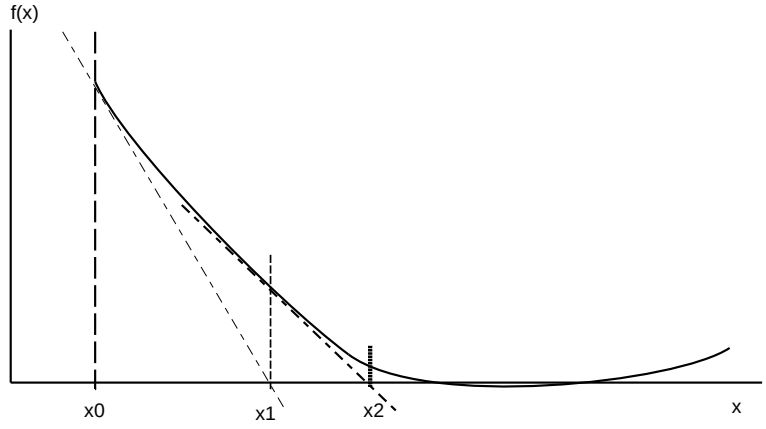


Figure A.1: The Newton iterative approximation technique.

Let us assume that there exists a solution x^*, $f(x^*) = 0$, such that f' exists, being finite and monotonic on $[x_0, x^*)$ for $f(x_0)f'(x_0) < 0$ (respectively on $(x^*, x_0]$ when $f(x_0)f'(x_0) > 0$). When $f(x_0)f''(x)\geq 0$ on $[x_0, x^*)$ (respectively on $(x^*, x_0]$), the sequence $\{x_i\}_{i>0}$ converges monotonically towards x^*. The proof of convergence (Cauchy sequence) follows from the fact that $f(x_i)f''(x)\geq 0$ between x_i and x^* implies that x_{i+1} lies between x_i and x^*, $x_{i+1} \neq x_i$ when $x_i \neq x^*$. The limit corresponds to $x_i = x_{i+1}$ (in Banach spaces), which requires $f(x_i) = 0$, since $f'(x_i)$ is finite.

A.1.1.1 Local optimization

Most algorithms do not attempt to really solve optimization problems in the sense of search for global optima, but try to find local optima (Figure A.2).

Definition A.3 *Let B be the space satisfying the predicates of a given optimization problem P. A local optimum for P is a vector $\vec{x} \in B$ such that there exists a sphere S with the center in $\vec{x}$ and radius $r > 0$ where $f(\vec{x'}) \geq f(\vec{x}), \forall \vec{x'} \in S \cap B$.*

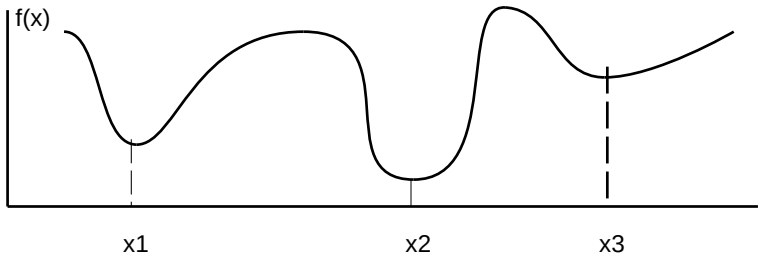


Figure A.2: Local (x_1, x_3) vs. Global (x_2) optima.

The search for a local optimum is called *local optimization*. Often in real applications, a local optimum is considered to be reached in $\vec{x^*}$ when the directional derivate is smaller than a threshold

$$D(f : \vec{d})(\vec{x^*}) < \varepsilon, \forall \vec{d},$$

where ε is a predefined threshold and $\vec{d}$ a direction. It is easy to notice that the Newton Method can be adapted to search a minima of the function $f(x)$. Taking again the unidirectional case, the recursion $x_{i+1} = x_i - f(x_i)/f'(x_i)$ can be written $x_{i+1} = x_i + l$ where $l = -f(x_i)/f'(x_i)$. l was previously chosen in order to estimate the intersection of $f(x)$ with the x-axis. For the minimization problem we search for a minima, x^*, but we do not know the value of $f(x^*)$. All we know is that $f(x^*) \leq f(x_0)$. Searching the solution for $f'(x) = 0, f''(x) > 0$ suffices when any local optimum is acceptable (inflection points can often be detected by studying $f''(x)$).

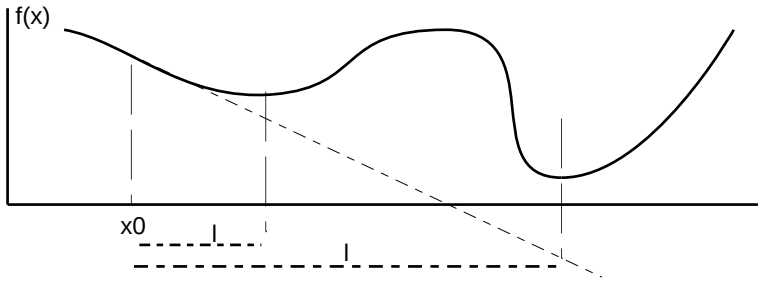


Figure A.3: Heuristics for line search. 'Visually' we can choose the values of l shown in the figure. It is much harder for an automatic system to make a good choice for a general problem.

The *steepest descent* algorithms are variations of the Newton Method where l is optimized according to different heuristics (Figure A.3). The most simple heuristics builds a sequence of values for l such that $l_{i+1} = \alpha^k l_i, 0 < \alpha < 1$, and k is the minimal natural number for which $f(x_{i+1}) < f(x_i)$. More complex heuristics (e.g. Goldberg, Wolfe) put constraints on $|f(x_{i+1}) - f(x_i)|$.

In multi-dimensional settings, the Newton and the steepest descent methods build a sequence of vectors $\{\vec{l_i}\}_{i \geq 0}$. The direction of each vector $\vec{l_i} = l_i \frac{\nabla f(\vec{x_i})}{||\nabla f(\vec{x_i})||}$, where $\vec{x_i} = \vec{x_0} + \sum_{j=0}^{i-1} \vec{l_j}$, and the value of l_i is computed with heuristics like the ones used for the unidimensional case. The *line search algorithms* are variations of steepest descent where the direction of the segments is not necessarily given by the gradient (e.g. *coordinate search* follows parallels to coordinates).

Trust-region methods are an alternation to line search algorithms. They use an approximation $f'(x)$ of $f(x)$. Trusted region methods do not follow the gradient in each point but rather optimize $f'(\vec{l_i} + \vec{x_i})$ over the values of $\vec{l_i}$ contained a sphere having radius r_i. r_i is given by heuristics.

When solving a multi-criteria optimization problem where several functions have to approach corresponding target values, a heuristic used in practice for finding an efficient gradient consists in minimizing the sum of squares of distances to the target. This technique is referred to as least squares minimization.

[2]This old technique has the same nature as techniques used for local search in discrete problems.

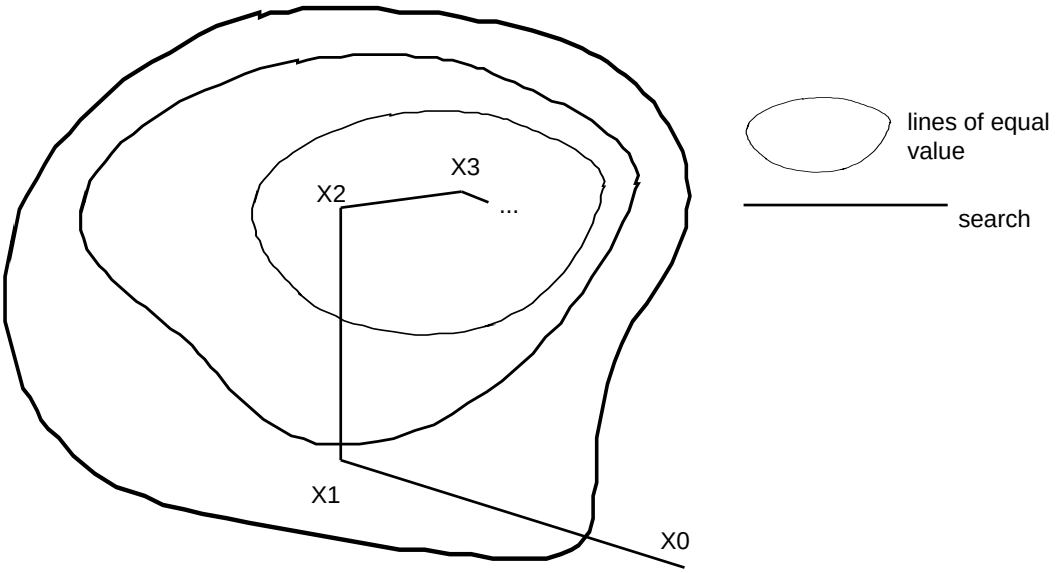


Figure A.4: In the multi-dimensional case, the steepest descent search follows a set of segments which in certain conditions converge toward a local or global optimum. The next direction at each segment end is given by the maximum gradient, while the length of the next segment is given by some heuristic.

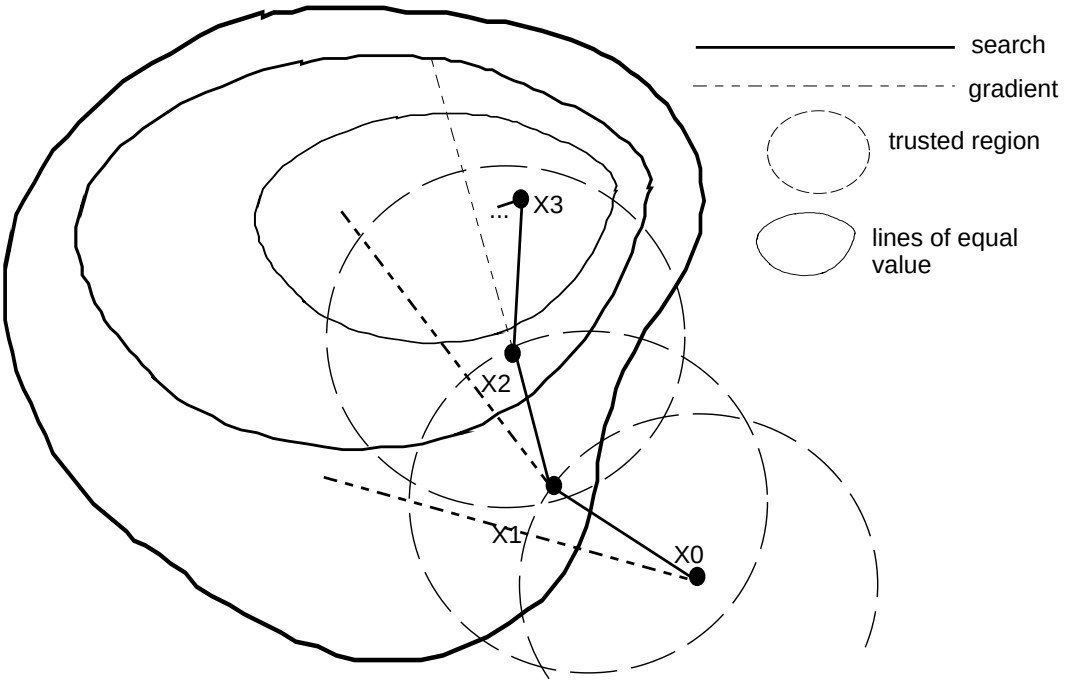


Figure A.5: The trust-region methods also follow a set of segments. Each segment leads to a global optimum of a function $f'(x)$ approximating $f(x)$ on a sphere that is centered in the starting end of the segment. The radius of the sphere is chosen using heuristics.

A.1.1.2 Locally local optimization

In this subsection we illustrate the first conflict in the notion of locality. Locality is defined in three dimensions:

- The first dimension was illustrated in this section. It is defined by the size of the neighborhood of a solution x, where x is guaranteed to be optimal. *Global* corresponds here to an infinite dimension while *local* corresponds to any finite dimension lower than the diameter of the search space. This sense is typically used for computations related to some kind of optimization, and we refer to it with LO.
- A second dimension consists of the amount of specifications of the problem that are analyzed in order to take a decision. The line search and trust-region methods compute gradients or approximate optima for the whole initial problem. In this respect they are *global*. When only parts of the initial problems are considered at once, as shown in this subsection, another kind of *local* methods are obtained. Such local computation is often used in constraint satisfaction, and we refer to it with LP.
- A third dimension consists of the physical space where the computations are done. Local in this sense shows that the computations are done in the extreme case on a processor, or more largely on a set of closer related processors, compared with all the processors involved in the algorithm. In this case *locality* is opposed by the term "distributed", and we refer to it with LS.

Since in this thesis we use all three meanings of locality, we will try to disambiguate it whenever the context is not sufficient. To show the difference between the first two notions of locality, we discuss now an algorithm that mixes both of them. In (Silaghi *et al.* 1998) is presented a technique for reconstructing an optimal skeleton for a recorded motion of an articulated object.

Example 1.29 *In order to animate virtual a character, real motion of a real object is recorded. A transformation is then performed from the motion of the articulations of the real object to the articulations of the virtual one. Most real objects, such like humans, do not have easily representable articulations with few degrees of freedom. Articulations such as shoulders and hands are very complex but are usually approximated with simpler sets of articulations. Moreover, even for simple real articulations, sensors cannot be used to measure exactly their motion. For humans, such sensors either entangle natural movements (e.g. exo-skeletons) or are complex and unhealthy (e.g. X-rays).*

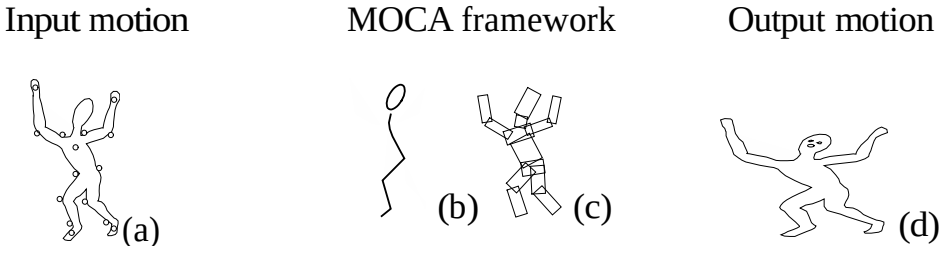
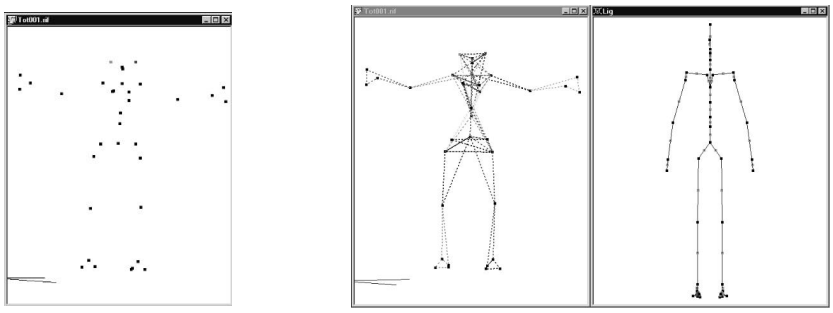


Figure A.6: Motion Capture process: a) Human performer wearing optical markers (little reflective spheres) b) Reconstructed Skeleton of the human being c) Anatomic Envelope for skeleton visibility assessment d) End user virtual character.

The nowadays solution (Boulic et al. *1998) consists in attaching a set of optical or magnetic markers to the moving articulated objects (Figure A.6a). The recorded motion of the markers (Figure A.7a) is transformed into size and movement for the a skeleton template (Figure A.7b). The obtained movement is then applied to new skeletons (Figure A.7d).*

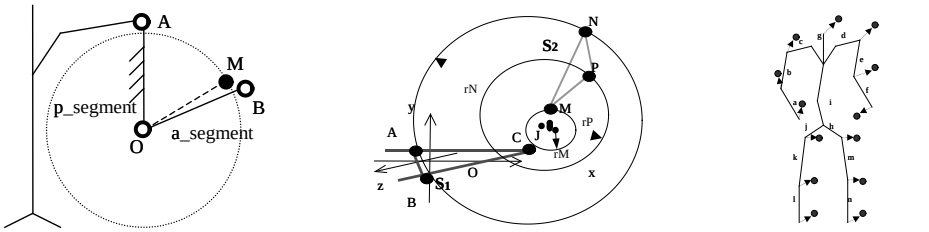
The locally (LP) local (LO) optimization algorithm presented in (Silaghi et al. *1998) is a very quick technique for fitting the dimensions and motion of a template skeleton to recorded special gymnastic motion of a set of markers. It approaches the problem in a hierarchical way, solving first local problems. The smallest subproblems (Figure A.8a) consists in optimizing the position of each joint given a trajectory for a marker on an adjacent segment (here M on OB) and the trajectory of all the markers on the second segment (here OA). It is solved by minimizing the error of the model (least squares) by a trust-region method. The obtained estimations of a joint*



a) Input motion b) Association of markers (right) to skeleton model (left)

Figure A.7: The input is a sequence of scenes with 3D marker positions.

(averaged) are optimized over the whole motion of the markers on the segments adjacent to the joint (Figure A.8b). While the optimization of the global problem (global fitting, Figure A.8c) allows for using more complex models of marker attachment, it is 10.000 times slower on the real application (seconds vs. hours). Whenever a complex model of marker attachment is useful, the locally local technique offers cheaply a very good initialization. Both global and local (LP) techniques are based on local optimization with trust-regions.



a) First Local subproblem b) Second Local Subproblem c) Global Problem

Figure A.8: Hierarchical Local (LP) approaches vs. global approach.

This example illustrates the efficiency of local (LP) reasoning. Local reasoning is extensively used in constraint satisfaction (Section A.2). The local optimization techniques yield suboptimal solutions but are acceptable in many problems. For the previous example, the template model is approximative and therefore even the definition of an optima is fuzzy. This is typical for multi-criteria optimization.

A.1.2 Searching for global optima

Many technical problems have clearly defined objective functions and require a global optima. Similarly, even for negotiation problems, the observance of an agreed objective function is required to ensure fairness. Local optimization has to be used when no efficient technique for computing a global optima of the given problem is known. Several types of problems allow for algorithms finding global optima: e.g. branch and bound, linear programs, maximal flow, shortest path. The section ends presenting successful classical results in linear programming which we find to foreshadow the nature of some contributions in this thesis.

A.1.2.1 Branch and Bound

Problems with linear predicates and objective functions (linear programs) can be optimally solved both with local search or using the simplex method. However, non-linear numeric problems in general need an exhaustive rummage of the search space for ensuring that no better solution is lost. Many techniques behaving according to this principle are referred to as Branch and Bound methods. Branch and Bound acts by maintaining a current estimation of the optima and removing

out of the search space all subspaces where it can be proved that no better solution can be found. The whole search space is split into regions until they become enough small to be eliminated (after eventually updating the estimated optima).

Function of the strategy used for splitting the search space, one distinguishes several versions of Branch and Bound. Cutting planes methods use hyper-planes to split the search space. When the cuts are chosen in order to isolate subspaces with convex objective function, the technique is called Back-Boxing (Van Iwaarden 1996). For a convex objective function, the local optima can be approached with quick local search techniques, variants of Newton Methods. These local search algorithms are stopped when they reach values that reduce the scanned search space to an acceptable precision. Concave spaces not bounded by discontinuities can be discarded. This technique avoids excessive splitting around local optima.

Branch and Bound can approach mixed integer linear programs (MIPs) as well as non-linear problems. Next subsection illustrate some algorithms guaranteed to reach global optima for network related problems.

A.1.2.2 Dynamic Programming and Iterating Dynamic Programming

Let us consider a network (graph) where each node and each arc are associated with costs. Typical network related problems, search paths with given properties (more exactly: paths, flows, or cuts). Let us associate to each path, P, a cost denoted $cost(P)$. The path between two given nodes, A and B, that optimizes (minimizes or maximizes) the cost function is denoted cost(A,B).

Definition A.4 *The plain sum cost of a path is given by the sum of the node and arc costs along it (excepting the cost of the starting node).*

For example, in Figure A.9, the optimal path from A to C is $P = (A, X_1, C)$. $cost(P) = cost(A, C) = cost(A, X_1) + cost(X_1) + cost(X_1, C) + cost(C)$.

One of most largely used family of optimization techniques for various best-path network optimization problems is called *Dynamic Programming*. Dynamic Programming allows to simultaneously compute the optimal paths and the minimal costs between any two nodes of a network. These optima are reached when an iteration for the re-estimation of the costs based on data from neighbors, converges. The convergence needs a number of iterations equal to the diameter of the network. Simplifications of Dynamic Programming are used to find the best path between 2 given nodes.

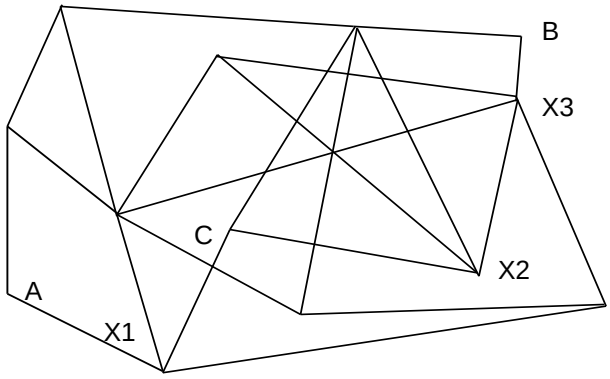


Figure A.9: Optimality condition for Dynamic Programming: if C belongs to the optimal path from A to B then we must have cost(A,B)=cost(A,C)+cost(C,B)

The optimality condition for the cost function with Dynamic Programming, illustrated in Figure A.9, requires that:

Condition A.1 *Given any two nodes A,B and given any node C on a path with optimal cost linking A to B, the optimal cost from A to B equals the sum of the optimal costs from A to C and from C to B.*

This condition is respected by the plain sum cost function. However, extensions of Dynamic Programming can often deal even with problems that do not respect Condition A.1. Let us consider the case of another cost function:

Definition A.5 *The averaged-sum cost of a path is given by the sum of all node and arc costs along the path, averaged over the number of nodes.*

For example, $cost(A, X_1, C) = cost(A, C) = (cost(A, X_1)+cost(X_1)+cost(X_1, C)+cost(C))/2$. (Silaghi & Bourlard 1999b) proves that the averaged sum cost function does not respect the Condition A.1. A general solution to this problem is given by Iterating Dynamic Programming (IDP). IDP is a variant of Iterating Viterbi Decoding (IVD), presented first in (Silaghi 2000a; Silaghi & Bourlard 1999a; Silaghi & Berinde 1999). IVD is a decoding algorithm for keyword spotting in speech recognition (Silaghi 2000a). A related algorithm is found in (Rosenknop & Silaghi 2001) where IDP is applied to optimizing normalized costs in CYK tables for syntactic parsing in Natural Language Processing. Given two nodes A and B and a value λ, IDP iteratively applies Dynamic Programming to find the path from A to B which optimizes the plain sum cost function. Before each iteration step, λ is subtracted from the initial cost of each node of the network. After each iteration step, the value of λ is updated with the average cost along the path computed by Dynamic Programming at that step.

A.1.2.3 Duality in Linear Programs

In this section we present some classic work in linear programming that illustrate a contribution of this thesis showing the interest of simultaneously using several representations of a problem. Besides locality, a second notion that we encounter in various contexts in problem solving is the "duality". In this section we shortly introduce the duality for linear programs. In Chapter 5 we will discuss some differences between duality for linear programs and duality for constraint satisfaction problems in order to better understand their nature.

Definition A.6 *A linear program is defined by:*

$$Ax \quad = \quad b, x \geq 0 \tag{A.8}$$

When b, c, and x are vectors, $c, x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and A is a $m \times n$ matrix, the goal is to find:

$$\min c^T x. \tag{A.9}$$

When c contains only 0, the linear program becomes a constraint satisfaction problem.

Definition A.7 *The dual of a linear program (referred to as primal) is defined by:*

$$\max b^T \pi, \quad when \quad A^T \pi \leq c \tag{A.10}$$

The dual of a linear program, P, can be easily reformulated as a linear program, the dual linear program of P. The primal and the dual linear programs are equivalent in the sense that $\forall x^*$, solution of the primal problem and $\forall \pi^*$, solution of the dual problem, $c^T x^* = b^T \pi^*$. While the primal version searches a minimum in a n_c dimensional polytope, the dual linear program searches a minimum in a n_b dimensional polytope. n_c and n_b are the numbers of elements of c, respectively of b that are not 0

It is worthy to note that:

Property A.1 *The primal-dual relationship is symmetric; by taking the dual of the dual, we recover the primal.*

The proof is direct (details can be found in (Nocedal & Wright 1999)).

Primal-Dual Algorithms Line search and trust-region methods find a global optimum for any linear program, P, if the followed segments are constrained to remain in the feasible space of P (interior point methods). While simplex methods visit an exponential number of vertices in the worst case, efficient polynomial techniques for linear programming are found among the interior point methods. The most used interior point methods are the primal-dual algorithms which use simultaneously both primal and dual representations of the initial problem.

The primal-dual algorithms solve variants of the Karush-Kuhn-Tucker system, defined as:

$$\begin{aligned} A^T\lambda + s &= c, && \text{(A.11)}\\ Ax &= b, && \text{(A.12)}\\ x_i s_i &= \tau; \tau \geq 0; i = 1, 2, ..., n, && \text{(A.13)}\\ (x, s) &> 0, && \text{(A.14)} \end{aligned}$$

where an optimum is reached for $\tau = 0$.

For example, primal-dual algorithms solve the system A.11-A.14 using Newton Methods (e.g. iteratively for decreasing τ) and often visiting only points where the conditions A.11,A.12,A.14 are respected. This gives them the name of *interior point methods*.

These algorithms illustrate a point that is at the center of some contributions in this thesis, namely that the simultaneous use of several representations of a problem enriches flexibility and can improve efficiency. Related approaches to boolean linear programs (linear programs with domains {-1,+1}) are detailed in (Schuurmans *et al.* 2001).

A.2 Constraint Satisfaction

A constraint satisfaction problem is an optimization problem (Definition A.2) where the objective function f is a constant.

Remark A.1 *Alternatively, constraint satisfaction can be seen as an optimization problem without predicates where the objective function in ∞ on infeasible tuples and 0 on feasible ones.*

Definition A.8 *A constraint satisfaction problem (CSP) is defined by:*

$$\begin{aligned} &\{x_i\}, i \in [1..(n_x)], && x_i \in D_i && \text{(A.15)}\\ &\{p_i\}, i \in [1..(n_p)], && p_i : D_{i_1^p} \times D_{i_2^p} \times ... \times D_{i_{a^i}^p} \rightarrow boolean && \text{(A.16)} \end{aligned}$$

A solution is any point in the space defined by the variables $\{x_i\}, i \in [1..(n_x)]$, and which satisfies all predicates $\{p_i\}, i \in [1..(n_p)]$. Such a point is defined by a vector, also called tuple.

The predicates of a CSP are often referred to as constraints. While a CSP is a special optimization problem, the CSP community usually approaches more general types of predicates than those involved in usual optimization problems. Even with the simplex method for general linear programs, (Nocedal & Wright 1999) mentions that the problem of finding an initialization has approximatively the same complexity as the subsequent optimization. Many CSPs deal with discrete domains, which seem less complex than continuous ones. However, most CSP techniques for discrete domains are only usable for very small domains.

Research around CSPs can be classified into two main streams, even if most often they are compactly interleaved. One of the main interests consists in reducing an initial CSP to an equivalent one, but which can be easier to handle further. The other class of approaches tempts explicitly to find solutions, tuples satisfying the constraints.

A.2.1 Reduction

An important operation that can be effectuated on a CSP is to reduce it to a (hopefully) simpler, equivalent CSP.

Definition A.9 *Two CSPs are equivalent if the sets of their solutions are equal.*

There are various reasons for resorting to reductions. One of them is based on the consideration that a CSP is a compact representation of a set of solutions and a comfortable way to send it over the Internet (Torrens & Faltings 1999; Willmott *et al.* 2000). Another reason to reduce a CSP is to enhance efficiency of further processing and solving. In this case, the reduction was traditionally referred to as a preprocessing step. Nowadays, the reduction is interleaved at all levels of a CSP solving process (Sabin & Freuder 1994). In the following, several kinds of reductions are discussed. The reformulation is one type of reduction. It applies to a whole CSP, or to a set of constraints.

Definition A.10 *The reformulation of a CSP, P, consists in reducing P to another equivalent CSP based on a different set of variables and constraints.*

Some reformulations integrate additional constraints that improve the efficiency of many solving algorithms for that problem.
Example 1.30

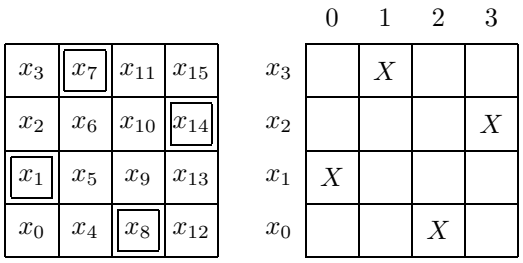


Figure A.10: Two formulations for the four queens problem. a) There are 16 boolean variables, one for each square. b) There are four variables, one for each row. The domains have four values each.

Freuder has often illustrated the importance of the formulation choice by comparing two formulations for the four queen problem with 16 respectively 4 variables. The first formulation, illustrated in Figure A.10a, associates a boolean variable to each chess-board square. The constraint that does not allow two variable to be simultaneously true (e.g. on a row, column or diagonal) can be expressed by logical operators, $x_3 \vee \neg x_6$, and has to be specified for all conflicting pairs of variables (there are 60 such constraints). The fact that there are 4 queens is formulated by the constraint $\sum_{i,x_i=true} 1 = 4$. A solution is $x_1 = x_7 = x_8 = x_{14} = true$ when all the other variables are false.

The second formulation, Figure A.10b, integrates implicitly the a priori knowledge that a solution can be found only when exactly one queen is placed on each row. Each row is modeled with a variable which can take as value the number of a column (0,1,2, or 3). The number of constraints is 12. The search space has now only $4^4 = 256$ points instead of the $2^{16} = 65536$ points of the first formulation.

Similar tradeoffs are noticed when we consider two other formulations where variables are associated with queens. Representing each queen by 2 variables, one for the row and one for the column, the search space is $4^8 = 65536$ and the number of constraints is 18. When implicitly each queen is in a different column, the same representation is obtained as in Figure A.10b.

Since a set of constraints defines a CSP, the reformulation of a set of constraints, C, consists in replacing C with an equivalent set of constraints on the same variables.

For example, six constraints of type $x_3 \vee \neg x_6$, for the formulation in Figure A.10a can be represented by $\sum_{i\in\{3,6,9,12\},x_i=true} 1 = 1$. A successful approach of this type makes use of *alldif* and *cardinality* predicates to constrain sets of variables to different values (Régin 1996; 1999; Puget 1998).

Definition A.11 *The domain reduction for a CSP consist in eliminating values from domains of some of its variables.*

Definition A.12 *The constraint reduction for a CSP consist in eliminating feasible tuples from the definition of existing predicates (adding new predicates).*

The domain and constraint reduction typically compresses the representation of CSPs, but it can also break an interval based representation. Most solving algorithms, among which the baseline Chronological Backtracking (Golomb & Baumert 1965), are quicker after any domain reduction. Exceptions can appear for interval-based and for Cartesian-representation-based algorithms (see Chapter 5), which exploit abstractions.

Example 1.31 *An example of domain reduction, reformulates the CSP in Figure A.10b as* $x_0 \in \{1,2\}$, $x_1 \in \{0,3\}$, $x_2 \in \{0,3\}$, $x_3 \in \{1,2\}$ *while all the constraints remain identical.*

Alternatively, by constraint reduction, the domains would not be modified, but one can discard the feasible tuples containing assignments eliminated in the previous example for domain reduction.

Actually domain reduction can be seen as constraint reduction when the domains are perceived as unary predicates. Later we show that constraint reduction is nothing but domain reduction in a dual representation.

Definition A.13 *A redundant constraint for a CSP, P, is a predicated which can be added to P without modifying its solutions.*

A CSP can be reduced by adding or removing redundant constraints. The removal of redundant constraints always reduces the representation requirements. Instead, adding new redundant constraints may improve efficiency (Yokoo 1997), if this is not overcome by the additional effort for constraint management. Actually, it induces overhead with linear cost and exponential improvements (Van Beek 2001).

A.2.1.1 Local Consistencies

Let us see how domain reduction can be obtained. Much research has been directed towards defining algorithms for efficient domain reduction. Typically, different algorithms lead to different results. Often, the intuition that more complex and expensive algorithms leads to stronger reductions is verified. However, many algorithms can be guaranteed to always achieve the same result. In order to classify the algorithms according to their results, the notion of *order of consistency* was defined. The *order of consistency* describes the obtained result, independently from the algorithm itself.

The most general orders of consistency for discrete domains are the global consistency, (i,j)-consistency (Freuder 1985; Mackworth & Freuder 1985) and the k-bound consistency (Lhomme & Rueher 1997). Real domains also allow the definition of kB(ε)-consistency (Lhomme & Rueher 1997) which is an adaptation of kB-consistency allowing for approximations quantified by the vector of errors ε. Annex D also describes a distinct notion called $\varepsilon_1\varepsilon_2\Phi$-consistency.

Definition A.14 *A CSP is* (i,j)*-consistent if any consistent assignment of* i *variables can be extended with* j *new assignments such that the obtained assignment is consistent.*

For given classes of problems, combinations of (m,n)-consistency are equivalent to global consistency, where each instantiation can be extended to a solution. k-bound consistencies for CSPs with ordered domains are obtained from $(1,k)$-consistency when the condition is only put for assignments of x_i to $inf(D_i)$ and $sup(D_i)$. $\varepsilon_1\varepsilon_2\Phi$-consistency and kB($\varepsilon$)-consistency are equivalents of global respectively of k-bound-consistency for continuous domains with bounded approximation errors.

The most used instances of (i,j)-consistency are the $(1,1)$-consistency aka Arc Consistency (AC), $(2,1)$-consistency aka Path Consistency (PC), and $(1,2)$-consistency aka singleton consistency. Existing general algorithms for achieving Arc Consistent CSPs are denoted ACx (so far AC1 to AC7, AC3.1, AC2000), respectively PCx (so far PC1 to PC5).

Several algorithms yield results that are dependent on very specific functions, on initial conditions or on used heuristics for ordering their operations. The most used notions for describing the obtained results are the k-Box-consistency for continuous domains and the one path AC for discrete domains.

A.2.1.2 Waltz Algorithm

The first widely known Arc Consistency algorithm was developed and used by Waltz (Waltz 1975). The famous application, that is described in most books on Constraint Satisfaction, was tempting to label scene edges in computer vision (see Figure A.11). Waltz has modeled this problem as a

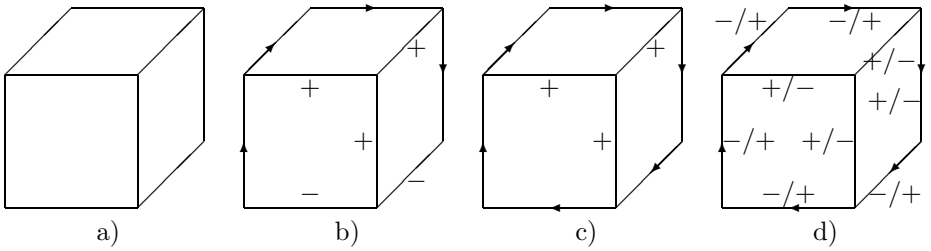


Figure A.11: a) A scene edges labeling problem; b) Labeling as a cube laying on a surface; c) Labeling as a flying cube; d) Possible labels after node-consistency.

CSP where each edge can take one of the next four values: $\{+, -, \rightarrow, \leftarrow\}$. + means convex visible sides, − means concave visible sides, and $\rightarrow$ interprets the edge as having only one visible side and the arrow encircles it clockwise. 16 types of realistic 2 and 3 edged junctions are possible: 6 of them are V-shaped, 4 are T-shaped, 3 are Y-shaped and 3 are arrow-shaped.

Such a scene may have several possible interpretations. Two interpretations for the scene in Figure A.11a are given in Figure A.11b and in Figure A.11c. In order to reduce the search space, Waltz has designed a preprocessing algorithm which reduces the number of possible labels for each node. For example, the three arrow shaped junctions in Figure A.11a eliminate the possibility that any of the three edges labeled + in Figure A.11b could be an arrow. Also, none of the external edges can be labeled with counterclockwise arrows (see Figure A.11d). By this simple operation, the search space is therefore reduced from the initial $4^9 = 262.144$ to $3^6 2^3 = 5.832$. The operation that we have just performed is called Node-Consistency, since it has checked that any possible label for any edge is allowed by any constraint taken separately.

Assuming now that the lowest edge is known to have the label −. Let us take now an edge, E, in the scene. Let us verify that given the possible labels for each of the connected edges, for each current label for E can be used to build legal junction types at both ends of E. The labels of E that fail this test are removed.

Performing the previous test once for each label in the scene yields an algorithm called one pass AC or Directed Arc Consistency (DirAC). For the scene in Figure A.11a, with some luck in the order in which edges are considered, DirAC can directly yield the labeling in Figure A.11b. When the order in which DirAC verifies the edges is less lucky, one can still obtain the labeling in Figure A.11b by iteratively calling DirAC until no modification is made. The algorithm obtained this way is called *Waltz filtering* and lays the basis for all Arc Consistency algorithms (see Chapter 2). *Waltz filtering* is a generalization of the filtering steps that were already used for boolean variables during search in the Davis-Putnam Procedure (Putnam & Davis 1960). The Davis-Putnam Procedure is given in Section A.2.2.2.

The powerful propagation of the *Waltz filtering* algorithm also amplifies the detection errors that are so common in Computer Vision. Variants for MAX-CSPs were separately developed (Larrosa 1998).

The Waltz algorithm seems easy to distribute since the effort is distributed to domains of variables. However, (Kasif 1990) argues that such distribution (variable-oriented) does not lead to parallelism.

A.2.1.3 Why Reduction?

As pointed out in examples, different types of reductions can be performed with a cost that is polynomial in the size of the predicates used in inferences. For large problems with many small

predicates (e.g. binary discrete problems, ternarized symbolic constraints, 3-SAT) this means polynomial cost in the size of the global problem.

Nevertheless, the gain brought by reduction is typically exponential in the size of the global problem. Polynomial cost with exponential gain make out of reduction the most important technique in constraint satisfaction.

A.2.2 Search

Reduction in general and usually local consistency can make a CSP easier or can even lead to a solution. In general, some splitting of the problem is required before a solution is found. In the example with scene edge labeling, we had to label an edge of Figure A.11d before a solution was returned by Waltz filtering. There are different strategies that can be adopted for problem splitting. The most used ones are dichotomous splitting (Van Hentenryck 1998; ILOG 1999) and value enumeration (Sabin & Freuder 1994; Gaschnig 1977; Kondrak 1994a). A few approaches have tried alternative splitting techniques (Larrosa 1997; Silaghi *et al.* 2000f; 2001j).

The splitting of a problem is often performed in optimization problems as well. In Mixed Integer Linear Programming, which are Linear Programs where some variables are constrained to take integer values, the local optimization is interleaved with problem splitting (Branch and Bound).

The techniques for solving CSPs are usually classified in complete and incomplete algorithms.

Definition A.15 *An algorithm that solves a CSP is complete if it cannot fail (return the answer "no solution") when a solution exists.*

An algorithm is incomplete if the previous property does not hold. Many complete algorithms are extensions of Chronological Backtracking. In general, the problem of deciding the feasibility of a CSP can be proven NP-complete by reduction to SAT (Garey & Johnson 1979). A very expensive procedure for solving CSP that is often mentioned in theory but never used in practice is *generate and test*. *Generate and test* consists in generating the Cartesian-product of the domains for the variables in CSP, and then filtering out those satisfying all predicates.

A.2.2.1 Backtracking

A valuation (a point in the search space of a problem), consists in a set of assignments, exactly one assignment for each variable of the problem. A partial valuation is also a set of assignments, but where not all variables of the problem are represented.

Chronological Backtracking (BT) avoids generating all the search space by filtering out of the not yet enumerated space, the regions containing *the current partial valuations* when they are proved infeasible. The next table compares the runs of *generate and test* and BT for the problem $a \neq b \neq c \neq a, a, b, c \in \{1, 2\}$.

Nr.crt.	a	b	c	Solution	Nr.crt.	a	b	c	Solution
1	1	1	1	no	1	1	1		
2	1	1	2	no	-	-	-		
3	1	2	1	no	2	1	2	1	no
4	1	2	2	no	3	1	2	2	no
5	2	1	1	no	4	2	1	1	no
6	2	1	2	no	5	2	1	2	no
7	2	2	1	no	6	2	2		
8	2	2	2	no	-	-	-		

In this example, BT avoids generating two valuations. The BT algorithms generate the space in a systematic constructive manner, the same that is typically used for combinatorial generation. It has the characteristic that feasibility tests are performed in a defined ordered manner, in order to detect as early as possible a *minimal* partial valuation that is infeasible (found leftmost in the valuation vector). It is this *detected minimal* infeasible partial valuation that prunes the future

```
procedure search do
    result=INTERRUPTED;
    while (!end) do
        UpdateDomains(crtVariable);
        if (!nextInstantiation(crtVariable)) then
            if (crtVariable==0) then
                crtInstantiated = false;
                result=EXAUSTED;
                resetVariableDomain(0);
                end=true;
            else
                crtVariable−−;
            continue;
        crtInstantiated=true;
        crtVariable++;
        crtInstantiated=false;
        if (checkFinished(crtVariable)) then
            crtVariable−−;
            crtInstantiated=true;
            result=SOLUTION;
            end=true;
        else
            resetVariableDomain(crtVariable);
    return result;
```

Algorithm 33: Chronological Backtracking. This procedure can be interrupted and then resumed.

search space. The fact that an order is defined on the variables in current feasible partial valuation helps to save feasibility tests for the prefixes that appear in the next generated partial valuation. It also helps defining a search tree. The search tree for the previous example is given in Figure A.12.

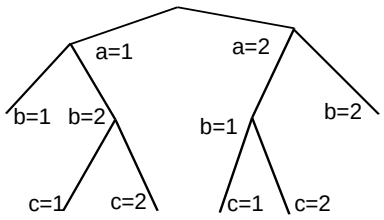


Figure A.12: BT runs can be graphically represented with search trees.

Most often, the BT algorithms are presented using recursive functions. The Algorithm 33 is an iterative version of BT. *crtVariable* stands for the index of the current variable and is initially 0. *UpdateDomains* is the procedure that sets the current domain to the domains allowed after analyzing the variable $crtVariable - 1$. *nextInstantiation* is a procedure that looks for a next valid value for *crtVariable* and returns true on success after updating the allowed domains. The *resetVariableDomain* procedure resets the iterator called by *nextInstantiation*. *checkFinished* checks whether we have checked all the constraints. An interrupt sets the *end* flag.

More complex backtracking schemes (Prosser 1993b; 1993a) avoid generating additional parts of the search space. This is done by filtering out of the not yet enumerated space, regions containing *subsets of the partial valuations* that are proved infeasible. Techniques such as backmarking (Gaschnig 1977) also save computation effort for tests at different partial valuations.

Variable instantiation corresponds to a splitting of the search space into a space conforming to

the instantiation and the space that does not conform to it. Backtracking algorithms resemble tree (and graph) visiting algorithms. The main graph visiting techniques are instances of A (Nilson 1980). We will mention several times such visiting techniques, of which two extrema are the breadth-first and the depth-first search.

A.2.2.2 Davis-Putnam Procedure for SAT

Now we take a short look at an old work whose relevance resisted the proof of time and where many concepts in the modern search techniques were already present. The first problem proven to be NP-complete is the Propositional Satisfiability (SAT).

Definition A.16 *A SAT problem is a decision problem. It questions the existence of an assignment for a set of boolean variables such that a given propositional formula is satisfied.*

It is interesting to note that while, according to definitions, a SAT problem only asks about the existence of a solution, many known algorithms actually try to build such a solution.

Most algorithms developed for SAT resemble Newton Methods in that they iteratively improve a valuation by following some gradient which reduces a measure of the current conflict. Typically such algorithms do not have guaranteed termination.

The most famous complete search algorithm for SAT (Davis & Putnam 1960) was developed by Davis and Putnam in 1960 and already integrates the boolean equivalent of the general *Waltz filtering* formulated in 1975. The procedure in (Davis & Putnam 1960) has visited the search space according to the general A algorithm (see subsection A.2.2.1) that could degenerate into breath-first. The algorithm was restricted to its depth-first version in (Davis *et al.* 1962). The version in Algorithm 34 is taken from (Gent & Walsh 1999). It takes as parameter a formula Σ in conjunctive normal form (CNF), that is a conjunction of clauses. A clause is a disjunction of literals (a variable or a negated variable).

procedure *DP(Σ)* **do**
- *(Sat)* **if**(Σ empty) **then return** satisfiable;
- *(Empty)* **if**(Σ contains an empty clause) **then return** unsatisfiable;
- *(Tautology)* **if**(Σ contains a tautologous clause c) **then return** DP($\Sigma - \{c\}$);
- *(Unit propagation)* **if**(Σ contains a unit clause $\{l\}$) **then**
  return DPΣ simplified by assigning l to *True*;
- *(Pure literal deletion)* **if**(Σ contains a literal l but not the negation of l) **then**
  return DPΣ simplified by assigning l to *True*;
- *(Split)* **if**(DP(Σ simplified by assigning a literal l to *True*) is satisfiable)
  then return satisfiable;
  else return DP(Σ simplified by assigning the negation of l to *True*);

Algorithm 34: Depth-first version of the Davis-Putnam procedure.

The Davis-Putnam procedure performs with priority at each step the *Unit propagation* which can be shown to be the equivalent of Waltz filtering for SAT. Actually, since SAT variables have only two values, each value elimination leads to the instantiation of the corresponding variable to the remaining value. The Waltz filtering is obtained implicitly since each such instantiation is performed explicitly in each clause (this is called *unit subsumption*).

The *Pure literal deletion* corresponds to a value ordering heuristic, with subproblem reuse (Freuder & Hubbe 1995). The elimination of the remaining value does not lead to loss of completeness. Indeed, whenever *Pure literal deletion* can be applied for a literal l, and a solution exists with $l = False$, changing l to $True$ remains a solution and the satisfiability will be proved.

While value ordering such as *Pure literal deletion* are generally not efficient, (Gent & Walsh 1999) mentions that variable ordering heuristics obtained by heuristics for choosing l at *Split* are more successful (e.g. first literals in the shortest clause). This foreshadows the successful CSP

heuristics for variable ordering that are meant to catalyze the efficiency of future Waltz filtering (e.g. MACE in (Sabin & Freuder 1997)).

A.3 Versions of Generalized Partial Order Backtracking

In Chapter 2 we learn about a powerful and general algorithm called Generalized Partial Order Backtracking (GPB). In this section I show two interesting versions: the most well known version, respectively the version used in the tests of (Silaghi *et al.* 2000a).

A.3.1 Dynamic Backtracking

The best known instance of GPB remains the Dynamic Backtracking (Ginsberg 1993a) (DB). DB is an efficient instance where effort is not made for maintaining assignments for all variables. A new value, τ, is added to each variable. A new unary constraint is added for each variable, such that τ is inconsistent. A set I of instantiated variables is defined as the set of variables not instantiated to τ (inspired from Conflict-Based Backtracking). I is constructed using additional rules for GPB and has the property of having assignments that are consistent with all initial constraints and a total order on variables. The additional rules are:

- initially $I = \emptyset$.
- only nogoods γ with conclusion variable y, $y \in I$, are selected in the procedure GPB at line 4.1. When none exists, I is extended (e.g by jumping at line 4.3).
- all the possible conditions $i <_S u$, $\forall i \in I, u \notin I$ are implied. Also, a total order on I is implied (Bliek 1998b).
- a set of ordering conditions $i <_S y$, is added $\forall i \in I \setminus \{y\}$.
- after adding *gamma*, at the end of Backtrack(γ), y is taken out of I, by instantiating it to τ.
- at line 4.3 in the procedure GPB, only one assignment, and namely one initially assigned to τ, is modified. The modified assignment is also inserted in I.

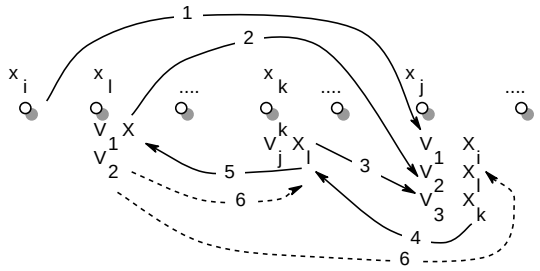
A.3.2 Versions of General Partial Order Dynamic Backtracking (GOB)

I promote now a new version of GPB that is more efficient in nogood storage. It takes some ideas from (McAllester 1993). For a set X of N variables taking their values in the domains D_k, $k = \overline{1..N}$, the inference rules (McAllester 1993) used in no-good based backtracking are:

1. Derive $\neg((x_{j_1}{=}w_{j_1}) \wedge (x_{j_2}{=}w_{j_2}) \wedge ... \wedge (x_{j_n}{=}w_{j_n}))$ whenever the no-good j is implied by a single constraint, where $w_i \in D_i$ and $x_i \in X$
2. Backtracking appears when, for a given variable y, all values v_1 through v_n in its domain are ruled out by no-goods $\sum_i \rightarrow y \neq v_i$. Based on these no-goods, the $\neg(\sum_1 \wedge ... \wedge \sum_n)$ no-good may be generated. $\sum_j$ is the notation for a set of assignments $(x_{j_1}{=}w_{j_1}) \wedge (x_{j_2}{=}w_{j_2}) \wedge ... \wedge (x_{j_n}{=}w_{j_n})$.

Many nogoods discarded with GPB at line 4.4 contain non-redundant information. Actually, the nogood $\rho := \Sigma \rightarrow (z{\neq}a)$ obtained at line 4.5 replaces completely all nogoods γ_i removed after recursion at line 4.4, only when the set of antecedent variables of each γ_i contains Σ. This means when:

1. all the nogoods used for inference at line 4.5 have identical antecedents $\Sigma \wedge (z{\neq}a)$, whenever they have z as antecedent variable,
2. any other nogood used in this inference must have its antecedents contained in Σ,



x_i	x_l	x_k	x_j	new γ	GPB Γ	GOB Γ
v^i	v_1	v_j^k	v_1	$x_i{=}v^i \rightarrow x_j{\neq}v_1$ (1)	1	1
v^i	v_1	v_j^k	v_2	$x_l{=}v_1 \rightarrow x_j{\neq}v_2$ (2)	1,2	1,2
v^i	v_1	v_j^k	v_3	$x_k{=}v_j^k \rightarrow x_j{\neq}v_3$ (3)	1,2,3	1,2,3
v^i	v_1	v_j^k	v_3	$x_i{=}v^i, x_l{=}v_1 \rightarrow x_k{\neq}v_j^k$ (4)	1,2,4	1,2,3,4
v^i	v_2	v_j^k	v_3	$x_i{=}v^i \rightarrow x_l{\neq}v_1$ (5)	1,4,5	1,3,4,5
v^i	v_2	v_j^k	v_2			

Figure A.13: *Sequence in GOB: the dashed arcs show no-goods removal while continuous lines show no-goods generation. The arcs are number in the order of the operation that they represent. In the first four columns of the table are the current assignments. The fifth column shows the no-goods added at each step while the sixth and seventh columns compare the list of no-goods maintained by GPB and GOB.*

3. any nogood in the system that has z as antecedent variable, also has the whole Σ as antecedent (for eliminations at line 4.4).

Property A.2 *Only when the three conditions hold, the eliminations of nogoods that follows the addition of γ does not correspond to a loss of information.*

Proof. We can say that a nogood γ replaces the set of nogoods Γ when:

- the space covered by γ covers the space covered by Γ and whenever
- γ is invalidated, all the nogoods in Γ are invalidated, too.

For solving these, we need that γ is activated whenever any nogood in Γ is activated (the antecedents of each nogood in Γ is a superset of the antecedents of γ) (Rules 1,3).

The Rule 2 is a consequence of the way in which the inference is made. □

You will ask:

2 *Why do we remove the nogoods if they are still useful and not redundant?*

Answer: There is an exponential number of non-redundant nogoods and we cannot store them all. We want to remove the nogoods in a way that ensures completeness with polynomial space requirements.

We can actually store an additional number of nogoods that has a size that is related to the number of GPB nogoods by a polynomial function, and we still get a polynomial space algorithm.

3 *Is GPB optimal for its size of the nogood storage?*

As a nice exercise, we now describe an algorithm that maintains a set of nogoods that is bounded by exactly the same size as GPB, and still saves more useful nogoods than GPB. We have noticed that certain no-goods that can be useful immediately may be erased too early with GPB (and with Partial Order Dynamic Backtracking (Ginsberg & McAllester 1994), PDB), when we backtrack due to the elimination of all the values in the domain of a variable. We show that the space requirement remains identical even if those additional no-goods are maintained.

Let us consider the example in Figure A.13 where the domain of variable x_j is exhausted and, based on the aforementioned rule, we generate a no-good that will have as conclusion $x_k \neq$

v_j^k. The no-goods are marked with a cross (x) that is indexed by one of the variables in its elimination explanation. PDB and GPB perform first the elimination of all the no-goods that have as antecedent $x_k = v_j^k$. If it happens now that x_k gets also its whole domain eliminated, then based on the same rule we backtrack to a no-good with conclusion x_l. The modification of the value of x_l may eliminate some no-goods for values of x_j, besides the no-good for $x_k = v_j^k$.

We may decide to re-instantiate x_k to v_j^k. When we return from this backtracking we may be able to instantiate x_j and continue the search without rechecking the no-goods already existing for x_k. Therefore, the elimination of no-goods performed here by PDB and GPB was not appropriate. We have noticed, instead, that the wished behavior can be allowed by the structure of Partial Order Backtracking (POB) (McAllester 1993).

Definition A.17 *An acceptable next assignment of variables X for a set of no-goods Γ and a subset $Y \subset X$ is any assignment for X that supports all those antecedents of the no-goods in Γ that do not depend on the variables in Y. All the assignments of the variables that are not in Y, or that are in Y and have just been modified now, do not figure in any no-good conclusion.*

The algorithm combining the described technique with the flexible ordering rules of GPB is called GOB and has the two steps described in Algorithm 35. The rule required to translate a

procedure *GOB* **do**

1. While no empty no-good is created and if one no-good γ is found, call simp(γ,$\emptyset$)
2. If no empty no-good was inferred then return the current instantiation.

procedure *simp (γ,St)* **do**

1. If γ is empty then stop reporting no solution.
2. Use the translation rules to represent γ as an implication $\sum \rightarrow z \neq v_z$.
3. Add the no-good (represented as an implication) to the current set Γ of no-goods.
4. If the live domain of z is empty, it must be possible to apply the second inference rule for deriving new no-goods to obtain a no-good γ_1 involving variables constrained to precede z. Derive such a no-good and recursively backtrack from it by applying simp(γ_1,$St \cup \{z\}$).
5. Else, if no new no-good was generated in step 4., i.e., if the no-good added in step 3 did not eliminate all live values of z, then for each variable w, which must follow z, and for each x that must be followed by z under the current order constraints, add the the safety condition $x < w$ and delete all safety conditions of the form $y < w$
6. The assignment ρ is now consistent with any no-goods added in step 4. This implies that the live domain for z must now be nonempty. Set ρ to $\rho[z := v]$ where v is in the current live domain of z and remove all no-goods not relevant to the new assignment. Set any acceptable next assignment for Γ and St, and if variables in St are changed, remove all no-goods that are not relevant to the new assignment.

.

Algorithm 35: General Partial Order Dynamic Backtracking (GOB): POB-like version of GPB. GOB can reuse better nogoods than GPB

no-good to an implication will choose the conclusion of the implication as one of the variables in the no-good that is the last in at least a total order that complies with the current no-goods and with the current safety order. The GOB algorithm has the qualities of GPB, while storing more useful nogoods than GPB.

4 *Is GOB optimal from the point of view of the use of allocated space resources?*

The answer is clearly no. The upper bound for the number of stored nogoods in GPB is nv. Besides the structures of GOB, an algorithm that wants to optimally use its resources can store an additional set of promising nogoods, rising the total number of nogoods to nv.

Theorem A.3 *The algorithm GOB is complete, terminates, is additive on disjoint subproblems for the first solution and visits at most as many states as GPB does.*

Proof. The GOB algorithm behaves like GPB with the only difference that the no-goods are erased only after making sure that some variables in their antecedents have changed. GPB guarantees a polynomial space by ensuring that all the no-goods are relevant and there exist at most one no-good per value. This means that we can have at most N^2K no-goods where N is the number of variables and K is the maximum domain size. For the current instantiation there exists at most NK no-goods (one for each value), and each no-good is at most of size N. In the case of GOB, since the additional no-goods we maintain remain relevant, the same reasoning holds. The existence of additional no-goods cannot increase the remaining space of the search at any step, therefore the maximum number of steps for GPB remains valid for GOB, and it terminates. The additivity is due to the same impossibility of generating no-goods between variables in disjunct problems as in GPB. For the comparison with GPB we notice that if we use with GOB the same heuristics for choosing no-goods, their conclusions and the re-instantiations, the only difference remains that some no-goods are not rediscovered in GOB for the case where variables are re-instantiated with the old values after backtracking. This reduces the number of expanded states. □

A difference to GPB is that using this approach, the algorithm cannot be requested to find a next assignment consistent with Γ at the end of each call to *simp* in the same way as GPB. When the set St is not empty, such instantiation cannot be found for the variables in it, which have an empty domain.

A.4 Distributions of random generators

Most systems provide users with a random number generators with uniform density of probability on [0,1], Uniform(0,1). Often, one actually needs a random generator offering a density of probability defined on a domain $[d_{min}, d_{max}] \in \mathbb{R}$ by a function f, $\int_{d_{min}}^{d_{max}} f(y)dy = 1$. Such a generator can be uniquely obtained by applying a monotonically ascending function q to the output of the generator of Uniform(0,1) (see Figure A.14). The probability density function (likelihood) obtained in this way is given by f:

$$f(q(x)) = \frac{1}{q'(x)}$$

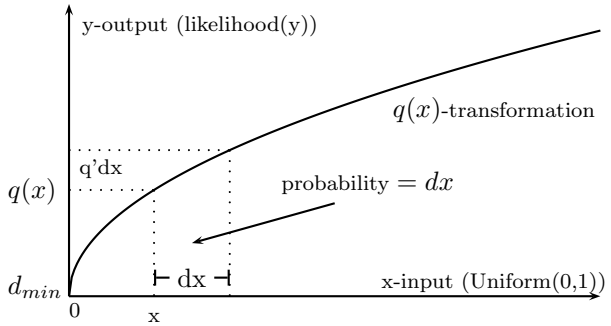


Figure A.14: In order to obtain a generator with likelihood f, a transform q is applied to the input Uniform(0,1) generator.

Therefore, in order to obtain a given probability density function f, q has to be chosen such that:

$$\int_{d_{min}}^{y} f(t)dt = q^{(-1)}(y)$$

Or, by inverting the functions in both terms:

$$q(x) = (\int_{d_{min}}^{x} f(t)dt)^{(-1)}.$$

Since some functions can have several inverses, the chosen one must be monotonically ascending.
Example 1.32 *For $f(y) = y$, the transformation that has to be applied to Uniform(0,1) is $q(x) = \sqrt{2x}$. This is actually the function plotted in Figure A.14.*

For

$$f(y) = \begin{cases} y - 1 & \text{if } 1 \leq y \leq 2 \\ 3 - y & \text{if } 2 < y \leq 3 \end{cases}$$

$$q(x) = \begin{cases} \sqrt{2x} + 1 & \text{if } 0 \leq x \leq \frac{1}{2} \\ 3 - \sqrt{2 - 2x} & \text{if } \frac{1}{2} < x \leq 1 \end{cases}$$

A.5 Summary

This appendix has introduced the optimization and constraint satisfaction generalities.

Many optimization numeric problems are approached with approximate methods. The Newton Method is the a simple local optimization technique that has many variants: steepest descent, line search, trust region, coordinate search, interior point methods. For some problems (e.g. linear programs) local techniques can lead to global optima. There exist algorithms that allow for finding global optima with some problems (e.g. simplex for linear programming, dynamic programming for some graph problems). For guaranteeing global optimality for non-linear numeric problems, one recurses to some kind of branch and bound (e.g. back-boxing).

The constraint satisfaction is a special case of optimization, where the objective function is a constant. While many optimization techniques start with a feasible point, and the technique mainly concentrates on iteratively searching a better one, constraint satisfaction only concentrates on finding the first feasible point. Modern techniques for constraint satisfaction work by interleaving problem reduction with problem splitting.

For assessing algorithms, we often use random problems. Real problems have special patterns. We end showing how problem generators with special distributions can be obtained from uniform random generators.