Basic User Manual

for

CSP-Rules V2.1

(Eighth update, August 2021)

Denis Berthier

Basic User Manual
for
CSP-Rules V2.1

(Eighth update, August 2021)

Books by Denis Berthier:
Le Savoir et l'Ordinateur, Éditions L'Harmattan, Paris, November 2002.
Méditations sur le Réel et le Virtuel, Éditions L'Harmattan, Paris, May 2004.
The Hidden Logic of Sudoku (First Edition), Lulu.com, May 2007.
The Hidden Logic of Sudoku (Second Edition), Lulu.com, November 2007.
Constraint Resolution Theories, Lulu.com, November 2011.
Pattern-Based Constraint Satisfaction and Logic Puzzles (First Edition), Lulu.com, November 2012.
Pattern-Based Constraint Satisfaction and Logic Puzzles (Second Edition), Lulu.com, July 2015.

Keywords: Constraint Satisfaction, Artificial Intelligence, Knowledge Based Systems, Constructive Logic, simplest-first search, Logic Puzzles, Sudoku, Futoshiki, Kakuro, Numbrix®, Hidato®, Slitherlink.

9 8 7 6 5 4 3 2 1

The following data was for the original release of this Manual:
Dépôt légal : Septembre 2020

ISBN: 978-1-71664-654-6

9781716646546

Table of Contents

1. Introduction

1.1 What is CSP-Rules?

A finite binary Constraint Satisfaction Problem (CSP) is defined by a finite set of variables (hereafter called the CSP-Variables), each with a finite domain; the problem is to find a value in each of their domains, in such a way that these values are compatible with a set of pre-defined binary contradictions (the constraints).

CSP-Rules is a *pattern-based* (or *rule-based*) solver of finite binary CSPs. In the CSP-Rules approach, *a possible value of a CSP-Variable is called a candidate and binary constraints express in term of "links" all the direct contradictions between two candidates*.

CSP-Rules is inherently associated with the approach to CSP solving defined and largely illustrated in my book "Pattern-Based Constraint Satisfaction and Logic Puzzles" ([PBCS]). This approach can be traced back, in the particular Sudoku context, to my much older book "The Hidden Logic of Sudoku" ([HLS]). CSP-Rules is best considered as a companion to [PBCS] and [HLS]. The books provide the precise definitions and develop the theoretical part, while the software is the proof that the theories are widely applicable.

In the [PBCS] approach, pertaining to the very general "progressive domain restriction" family of CSP solving techniques, the domain of each CSP-Variable is represented by a monotonously decreasing set of candidates and the fundamental concept is that of a *resolution rule*, i.e. a rule of the form "pattern => elimination of a candidate" or (more rarely in practice) "pattern => assertion of a value for a CSP-Variable". Here "pattern" must be a clearly defined set of logical conditions, based only on the fixed set of CSP-Variables, the fixed links (i.e. direct binary contradictions) between candidates and on the current situation (i.e. the remaining candidates for each variable). This pattern must be precisely defined so as to imply the conclusion in an otherwise context-independent way (i.e. in a way that does not depend on any CSP-Variable, link, candidate or value not explicitly mentioned in the pattern). Of course, the rule itself must be provable from the CSP axioms.

Considering this definition, an implementation in terms of rules of a forward-chaining inference engine seemed relevant (in spite of possible *a priori* performance issues). Both the oldest and the current implementations of CSP-Rules are therefore based on CLIPS, the most widely used (and tested) inference engine. CLIPS is written in C, is highly portable, has been public since 1996; it implements the RETE algorithm with drastic performance improvements – and it is open source and free.

One essential aspect of the [PBCS] approach is the introduction of additional CSP-Variables: the set of CSP-Variables is extended beyond the natural, originally given ones. As a simple illustration of this idea in the Sudoku case, in addition to the "natural" rc CSP-Variables (represented by the cells of the standard Sudoku grid), I introduced (in [HLS1]) new rn, cn and bn variables, represented by the cells of the three additional spaces of my "Extended Sudoku Board" (a copy of which is present in the "Publications" folder of CSP-Rules). This "super-symmetric" view allows to consider the three classical Sudoku constraints (only one instance in each row, each column, each block) the same way as the implicit one: only one value in each cell; and relations that were considered different (bivalue/bilocal) appear to be the same.

A priori, CSP-Rules deals only with binary constraints, but the applications studied in [PBCS] and included in the CSP-Rules-V2.1 package (namely: solvers for Latin Squares (and its Pandiagonal variant), Sudoku, Futoshiki, Kakuro, Map Colouring, Numbrix, Hidato and Slithering) show that many types of non-binary constraints can be efficiently transformed into binary ones by adding problem-specific CSP-Variables, thus making them amenable to the CSP-Rules treatment.

Patterns in CSP-Rules can take many forms, but the most powerful generic ones are various kinds of chains (bivalue-chains, z-chains, t-whips, whips, g-whips…). *A chain is defined as a continuous sequence of candidates, where continuity means that each candidate is linked to the previous one*. In the context of logic puzzles, these chains can be considered as logical abstractions of the universal, spontaneous practice of human solvers wondering "can this candidate be true?" and checking for a possible contradiction implied by such a hypothesis; but they also suggest much more constrained and useful ways of proceeding. In particular, the continuity condition is a very strong guide for a human solver looking for chain patterns. More generally, *the absence of OR-branching in any of the CSP-Rules chain patterns means that each of them supports a single stream of reasoning*.

Instead of having zillions of application-specific rules (like e.g. most existing rule-based Sudoku solvers), the resolution backbone of CSP-Rules consists of only a few types of universal rules – though it remains perfectly compatible with the addition of any number of application-specific rules (see the Slitherlink chapter).

One more essential aspect of the CSP-Rules resolution paradigm is its insistence on using the "simplest-first strategy". Indeed, much of my approach can be considered as a detailed study into possible meanings of "*simplest-first search*". At each step of the resolution process, the simplest available rule is applied. Here, "simplest" is not to be understood as it is generally done in the world of "symbolic" AI, i.e. in an abstract logical way that has never had any real application. "Simplest" is defined precisely in terms of the patterns making the conditions of the rules. In case of chains, simplicity is easy to define: a chain (of any type) is simpler than another chain (of any type) if it is shorter, where the length of a chain is defined as

the number of CSP-Variables its definition involves. For chains of same lengths but of different types, it is also easy to define their relative simplicity (see chapter 4). As a result, for each family of rules, an intrinsic rating of the difficulty of instances can be defined and, what's most remarkable, it can often be obtained at the end of *a single resolution path*. The simplest-first strategy is intimately related to the *confluence property* (see [PBCS]). The advanced part of the Sudoku chapter will show that several alternative strategies can be defined based on the same sets of rules, but simplest-first remains the central strategy.

1.2 The contents of CSP-Rules

CSP-Rules consists of a generic part (in the folder "CSP-Rules-Generic") together with a few application-specific parts aimed at solving some familiar logic puzzles. The application-specific parts are integral members of CSP-Rules. They were chosen in order to illustrate how, by the proper choice of additional CSP-Variables, the generic concepts can be used in so different CSPs as the above-mentioned ones, including some non-binary ones.

The generic part consists of powerful generic resolution rules together with a general mechanism for managing the whole system, including the outputting of solutions in an easy-to-understand, universal notation. The generic part cannot be run alone. It requires an application-specific part to feed it with problem instances in the proper format.

Each application-specific part consists of a specific interfacing with the generic part of CSP-Rules, specific commands for launching the resolution process (they will be reviewed in the application-specific chapters in Part II of this Manual), plus a configuration file allowing the user to select general settings and which families of rules he wants to apply. Often, it also contains application-specific resolution rules (a few of which may be rewritings of some basic generic rules, e.g. more focused forms of solution detection, for better performance).

1.3 Scope of this Manual

All the general ideas, the precise concepts and the generic resolution rules, plus the specific applications and associated specific rules that materialised into CSP-Rules were introduced many years ago in a series of books and articles starting with the first edition of [HLS] ([HLS1], May 2007) and ending in the second edition of [PBCS] ([PBCS2], July 2015).

All my publications related to the pattern-based resolution of finite CSPs are an integral part of CSP-Rules. You can find them in the "Publications" folder, except for this Manual (which is in the Docs folder). At least some general understanding

of [PBCS] is a pre-requisite for a deep understanding of CSP-Rules and of this Manual; [PBCS] is also enough, in the sense that there is no other prerequisite.

It must thus be clear that *this "Basic User Manual for CSP-Rules V2.1" is designed as a supplement to [PBCS], although only a general understanding of [PBCS] is required*.

"Basic" means that it explains how to install CSP-Rules and how to use it for solving instances of the specific types of puzzles that are already part of CSP-Rules. It does not explain the CSP-Rules technical design, i.e. how the concepts introduced in [HLS] or [PBCS] are implemented in CSP-Rules, using the CLIPS language. [A possible future "Advanced User Manual" will explain how to add new rules to an existing application and a possible future "Programmer Manual" will explain how to create new applications or variants of the existing ones. Adding new rules is already easily possible by copying and adapting the existing ones.]

V2.0 was the stable version of CSP-Rules used to solve all the puzzles presented in [PBCS2]. It has also solved millions of Sudoku puzzles and thousands of puzzles of each of the other types defined in [PBCS2]. No software can ever be claimed to be bug-free, but V2.0 was probably as bug-free as it could be (in particular, bugs in rules that would illegitimately eliminate a candidate are in general found after trying less than ten puzzles). This is no legal guarantee and there will be no reward other than eternal gratitude for reporting a bug ;) Version V2.1 is the first public distribution and it is only a small variant of V2.0. It has an easier installation procedure; easier selection of rules in the configuration files; optionally, more compact output thanks to "blocked" variants of some existing rules; zero change in the existing rules themselves but addition of a few typed-chains in order to comply with continued requests from readers of [HLS] to re-introduce my older "2D-chains" (as special cases of the general chain rules); addition of "oddagons". Recent updates have a few more additions: means for finding backdoors, anti-backdoors, anti-backdoor-pairs, 1-step, 2-step and fewer-steps solutions; Forcing-T&E. Recent internal changes also include simpler coding (especially for printing the solution), after I definitively gave up the compatibility with the Jess inference engine.

"Supplement" means that this Manual is not fully self-contained; it does not repeat the general resolution paradigm defined and explained in [PBCS]. Nor does it repeat the detailed definitions of the different patterns or their resolution power. General concepts of the resolution paradigm or of whips, g-whips, braids, … are defined in [PBCS] and that is where you should look for full information about them. However, in order to avoid any ambiguity about CSP-Rules, I repeat (in the next two sections) the general motivations for the approach adopted in [PBCS] and some remarks about the rating of instances of a CSP. For an *informal* description of the main chain-pattern (whips…), I also give a graphico-symbolic representation of partial-whips, t-whips and whips in section 1.7, showing how they are related.

1.4 Motivations for the [PBCS] approach (adapted from [PBCS])

Since the 1970s, when it was identified as a class of problems with its own specificities, Constraint Satisfaction has quickly evolved into a major area of Artificial Intelligence (AI). Two broad families of very efficient algorithms (with many freely available implementations) have become widely used for solving its instances: general purpose structured search of the "problem space" (e.g. depth-first DFS, breadth-first BFS) and more specialised "constraint propagation" techniques (that must generally be combined with structured search according to various recipes). SAT solvers are also publicly available.

You may therefore wonder why you would want to use the computationally much harder techniques inherent in the approach introduced in [PBCS]. It should be clear from the start that there is no reason at all if speed is your first or only criterion. In particular, if your goal is to generate instances of a CSP (which requires a very fast solver), CSP-Rules is not a viable choice.

But, instead of just wanting a final result obtained by any available and/or efficient method, you can easily imagine additional requirements of various types and you may thus be interested in *how* the solution was reached, i.e. in the *resolution path*. Whatever meaning is associated with the quoted words below, there are several inter-related families of requirements you can consider:

– the solution must be built by "constructive" methods, with no "guessing";
– the solution must be obtained by "pure logic";
– the solution must be "pattern-based", "rule-based";
– the solution must be "understandable", "explainable";
– the solution must be the "simplest" one satisfying the above requirements.

Vague as they may be, such requirements are quite natural for logic puzzles and in many other conceivable situations, e.g. when you want to ask explanations about the solution or parts of it.

Starting from the above vague requirements, Part I of [PBCS] (generalising the [HLS] approach) elaborated a formal interpretation of the first three, leading to a very general, pattern-based resolution paradigm belonging to the classical "progressive domain restriction" family and resting on the notions of a *resolution rule* and a *resolution theory* (i.e. a set of resolution rules).

Then, in relation with the last purpose of finding the "simplest" solution, [PBCS] introduced ideas that, if read in an algorithmic perspective, should be considered as defining a new kind of search, "*simplest-first search*" – indeed various versions of it, based on different notions of logical simplicity. However, instead of such an algorithmic view (or at least before it), a pure logic one was systematically adopted, because:

– it is consistent with the previous purposes,

– it conveys clear non-ambiguous semantics (and it therefore includes a unique complete specification for possibly multiple types of implementations),

– it allows a deeper understanding of the general idea of "simplest-first search", in particular of how there can be various underlying concrete notions of logical simplicity and how these have to be defined by different kinds of resolution rules associated with different types of chain patterns.

1.5 Simplest-first search and the rating of instances (adapted from [PBCS])

In the [PBCS] context and in conjunction with the "simplest solution" requirement, there appeared the question of rating and/or classifying the instances of a (fixed size) CSP according to their "difficulty" – a much more difficult topic than just solving them. The main families of resolution rules introduced in [PBCS] are various kinds of chains with no OR-branching [which would amount to adding hidden levels of Trial-and-Error] and they go by couples, corresponding to two different linking properties, namely "T-whips" and "T-braids", where the T parameter refers to the "elementary" patterns allowed to appear in the chain as its building blocks (i.e. mere candidates for whips, g-candidates for g-whips, Subsets for S-whips, inner whips for W-whips…). For each T, there are two ratings, defined in pure logic ways:

– one based on T-braids, allowing a smooth theoretical development and having good abstract computational properties; much time was devoted to prove the *confluence property* of all the T-braid resolution theories introduced in [PBCS] (when T itself has the confluence property), because it justifies a "*simplest-first" resolution strategy* (and the associated "simplest-first search" algorithms that may implement it) and it allows to find the "simplest" resolution path and the corresponding rating by *following only one resolution path*, a computationally noticeable advantage; the T-braid rating has the fundamental property of being invariant under CSP isomorphisms.

– one based on T-whips, providing in practice a good, much easier to compute approximation of the first when it is combined with the "simplest-first" strategy. (The quality of the approximation was studied in detail and precisely quantified in the Sudoku case, but it also appeared in intuitive form in all our other examples.)

[PBCS] explained in which restricted sense all the above ratings are compatible. But it also showed that each of them corresponds to a different legitimate "pure logic view" of simplicity – so that defining logical simplicity is not straightforward.

It must also be noted that all of the above ratings are ratings of the hardest-step. One might want to rate full resolution paths instead, but nobody has ever made any consistent proposal of how to approach such a problem. In particular, the number of "steps", i.e. of rule applications, cannot be formulated in pure logic terms.

1.6 What can one do with CSP-Rules?

Basically, CSP-Rules can solve and rate instances of a finite CSP (provided that this type of CSP has been interfaced to the generic part of CSP-Rules) – with some mild restrictions on their complexity. It will display the full resolution path, in an easy-to-read formal syntax clearly stating the justification of each step.

In chapter 11 of [PBCS], the scope of various resolution rules was analysed in terms of a search procedure with no "guessing" – Trial-and-Error (T&E) – and of the T&E-depth necessary to solve an instance. In and of itself (and contrary e.g. to the maximum depth reached by a DFS procedure), *this T&E-depth defines a broad, intrinsic and universal classification of the difficulty of all the instances of any finite binary CSP of any size* (and a – slow – T&E simulator is therefore included in CSP-Rules).

For instances in T&E(1) and T&E(2) (i.e. requiring no more than one or two levels of Trial-and-Error), there are universal "pure logic" ratings based on two T-braids families, respectively the B and the $B_?B$ ratings. [In the present case, universality must be understood in the sense that these two ratings assign a finite value to all of the corresponding instances, but not in the sense that they could provide a unique notion of simplicity.]

The above-mentioned universal T&E classification is an intrinsic property of all the instances of any finite binary CSP. CSP-Rules cannot be blamed for being unable to solve in "pure logic" ways some exceptionally hard instances that are intrinsically beyond the capabilities of its relatively simple chain patterns.

Being in T&E(0) means being solvable by the most elementary resolution rules. Being in T&E(1) amounts to being solvable by braids (and very often by whips). Being in T&E(2) requires not only braids but also much more complex B-braids (not coded in CSP-Rules) or, alternatively, bi-braid contradictions and B*-Braids. However, at the lower end of T&E(2), there is a small layer of puzzles in gT&E(1) = T&E(whips[1], 1) that is worth mentioning. Being in T&E(n) for larger n requires still more complex tri-braid (quadri-braid…) contradictions and $B^{*\cdots\cdots}$-braids.

As a result of this intrinsic T&E stratification, the solution of some exceptionally hard instances cannot be found if the resolution rules one has selected are not powerful enough. For the hardest instances, CSP-Rules may be able to find only a broad classification – e.g. membership in B_pB for instances in T&E(2) – instead of a pattern-based solution or a more precise rating. This can nevertheless be very useful if one wants to check the power of a new resolution rule R and asks: can R simplify the problem enough to make it belong to a lower B_qB after being applied? In case a puzzle cannot be solved by the chain-patterns, CSP-Rules also includes a simulated DFS procedure (very slow for that kind of algorithm) that can solve almost anything (if you allow it enough time).

In order to give a general idea of the power of whips and braids, it has been found that only one minimal 9×9 Sudoku puzzle in about 70,000,000 is in T&E(2). This means that almost all of them are in T&E(0 or 1) and can therefore be solved by braids (and, most of the time, by whips).

A more intuitive way of considering the limits of CSP-Rules is as follows: the various degrees of difficulty of puzzles you can find in newspapers correspond to walking to your garden versus to your nearest neighbour's home. The most extreme known instances correspond to going to far-away galaxies. CSP-Rules lets you travel this galaxy.

It may also be the case that some problem instance (considered as an ill-posed problem in the context of logic games) has several solutions. This is not a real problem for CSP-Rules. But as it can only do (constructive) pure logic deductions, it can only prove properties that are common to all the solutions (i.e. eliminate candidates that are false in all the solutions or find values that are true in all the solutions). Inconsistency of an instance is not a problem either: if it can be proven with the selected rules, CSP-Rules will prove it. Notice that proving inconsistency can be as hard as finding a solution, as shown in [PBCS].

Finally, remember that CSP-Rules was designed as a software tool for research, with the purpose of providing the "simplest" pattern-based solution as defined in [PBCS]. This is *per se* a much harder problem than just finding a solution and computation times cannot be expected to match those of the simpler problem (typically small fractions of a second). But I must also recall that my project was not a programming one, the only purpose of CSP-Rules was to validate the wide range of applicability and the resolution power of the rules defined in [PBCS] (plus alternative ones that I did not keep). Definitely, these rules could be implemented much more efficiently than they currently are; CLIPS has too many restrictions when it comes to using advanced data structures. Efficient programming has never been my purpose in writing CSP-Rules, even though I have done a few speed and memory optimisations within the Rete algorithm paradigm underlying CLIPS.

1.7 A quick graphical introduction to the most basic chain rules

This section is not intended to be a summary of [PBCS], but only a graphical introduction to the most elementary chain rules. The central pattern in CSP-Rules is that of a whip. Except possibly bivalue-chains (that can be better understood without any reference to whips), all the chain rules in CSP-Rules can be considered as variants (either special cases or generalisations) of whips. Whence the necessity to understand whips. Whips have a very important property: they are continuous sequences of candidates, where "continuous" means that each candidate is linked to the previous one in the sequence.

First, we shall need some graphical conventions:

- I represent a CSP-Variable as a thick vertical straight line with a subscripted name above it, starting with V, e.g. V_1, V_2… You must understand that this is a purely abstract representation, that I could as well decide to represent a CSP-Variable by an oval surface, and that it is not supposed to represent real lines (rows or columns) in puzzles residing in a square or rectangular grid. To be concrete, in Sudoku, such a thick vertical line can represent a CSP-Variable of any of the four types (rc, rn, cn, or bn). In other terms, if you prefer, it represents any of the 324 (rc, rn, cn, or bn) cells of the Extended Sudoku Board.
- I represent candidates for V_i as subscripted letters near this line (the first subscript being equal to the subscript of the CSP-Variable, i). Left-linking and right-linking candidates for V_i are named respectively L_i and R_i; other candidates for V_i are named $C_{i,j}$. Note that the $C_{i,j}$'s are not part of the whip. Candidates in bold are part of the whip, while other candidates (the z- and t-candidates, named $C_{i,j}$) are not part of it and are represented in clearer grey. Z, the name I usually choose for the target of a chain, is a candidate for any CSP-Variable(s) other than the V_i's.
- I represent any non-CSP link between two candidates as a thin line joining them. Remember that *a link means a direct contradiction between two candidates. It is a structural ("physical") property of the CSP; it does not require any "inference"; it is independent of any particular instance of the same CSP. A link pertains to the general background of the CSP and it is independent of any particular instance of this CSP (e.g. of any puzzle in Sudoku) or of any resolution state for it.*

Warning: one problem with the following representations is, they may be misleading. *The z and t candidates (the $C_{i,j}$'s in the graphics) are not part of the whip: they could disappear from the resolution state without making any damage to the whip* (remember that only two things can happen to a candidate: disappear, i.e. be proven to be impossible, or be asserted as a value, in which case both the L_i candidate and therefore the whip itself will merely disappear). *Moreover, when establishing the whip property in any particular instance of a whip, naturally proceeding from left to right, the $C_{i,j}$ candidates only need to be taken into account momentarily, at step i. Establishing this property at step i involves no logical reasoning, no "inference"…; it only involves checking the presence of a structural link for each of the $C_{i,j}$.* Once the whip property is established up to V_i, the $C_{i,j}$'s for this i or the previous ones play no role in establishing the whip property in the next steps; they can be totally forgotten. This is why I represent them in lighter grey. Forgetting the $C_{i,j}$'s and the associated links, a whip[4] can be represented graphically as a single continuous line going through Z—L_1—R_1—L_2—R_2—L_3—R_3—L_4— and ending in no candidate (represented by a dot) and this is why only the left-linking and the right-linking candidates need be represented in the linear symbolic representations appearing in the CSP-Rules resolution paths. (The left-linking candidates must be represented as they ensure the continuity condition.)

1.7.1 Whips[1]

Simple as they may seem (they are the simplest pattern one can imagine, apart from Singles), whips[1] are an extremely powerful pattern. Whips[1] involve a single CSP-Variable.

Whips[1] can be considered as the first step of almost all the chain rules defined in [PBCS]. I strongly recommend that you look for whips[1] in different puzzles, not only in the most common one, Sudoku. The more examples you see from different perspectives, the more you will understand the magic of whips[1].

In Sudoku, rules called Intersections, Interactions, Alignments, or also Pointing and Claiming, were known much before I introduced whips[1] (indeed much before I ever heard of Sudoku). As I have proven in detail in [HLS] (with all the necessary graphics), it is easy to check that whips[1] are equivalent to these rules; i.e. they allow the same eliminations. (If you are not yet familiar with whips, I suggest that you prove this equivalence for yourself and you work on whips[1] as long as needed to understand them perfectly before trying to understand longer ones. A good test for this is whether you understand all of this section.)

However, there are three major differences between the usual viewpoint in Sudoku and the one introduced by whips[1]:

- whips[1] are defined in a universal way, meaningful for any binary CSP (and I have shown in [PBCS] that they are indeed very useful in many different types of CSPs, independently of having an underlying grid structure or not); try the various applications included in CSP-Rules to see how they appear in them;
- whips[1] make a strict difference between CSP-Variables and links (a difference totally blurred in Sudoku); but they treat all the types of CSP-Variables the same way and they treat all the kinds of links the same way; for instance, in Sudoku, they don't make any difference between numbers, rows, columns and blocks (more precisely, between CSP-Variables of types rc, rn, cn or bn);
- whips[1] can easily be generalised to longer chains, based on the same reasoning, as shown in the next subsection; it is therefore necessary to understand them perfectly in the precise context of any specific application before looking for longer ones.

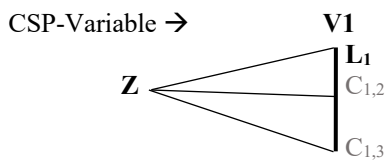


Figure 1.1. *Symbolic representation of a whip[1]*

The whip[1] rule*: if a candidate Z is linked to all the remaining candidates for a CSP-Variable V_1 (which implies that Z is not a candidate for V_1), then Z can be deleted*. The proof is obvious: if Z was True, V_1 could have no value.

The elimination described in the whip[1] rule will be written symbolically as: $V_1\{L_1 .\} \Rightarrow$ not Z. Here, L_1 is called the left-linking candidate and $C_{1,2}$ or $C_{1,3}$ the z-candidates (because they are linked to Z); in the present case, their roles could be interchanged.

In practice, when there is an underlying rectangular grid, a more specific notation (the nrc notation) will be applied and the eliminations will appear in still a more readable form, such as: $r_3n_4\{c_5 .\} \Rightarrow r_4c_6 \neq 4$. Here CSP-Variable V_1 is r_3n_4, it is supposed to have candidates only from the set $\{c_4, c_5, c_6\}$ ("in the intersection of row r_3 with block b_2"), which implies that any candidate "with" number 4 in block b_2 but not in row r_3 can be eliminated. This is the case for target $Z = n_4r_4c_6$.

1.7.2 A partial whip[1] and a whip[2]

It often happens that a candidate Z is linked to all the candidates for a CSP-Variable V_1, except to one of them. In and of itself, this does not allow any elimination; but it can be used to build whips[2].

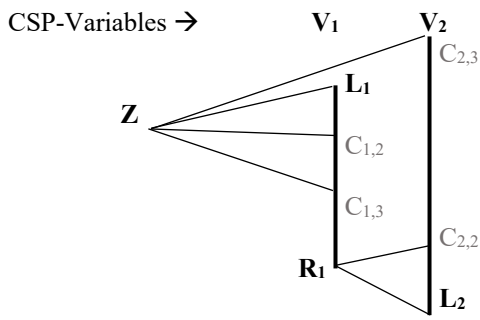


Figure 1.2. *Symbolic representation of a whip[2]. A partial-whip[2] would have in addition one pending candidate for V_2: R_2 (as in Figure 1.3)*

In Figure 1.2, CSP-Variable V_1 has four remaining candidates, the first three of which (the same L_1, $C_{1,2}$, $C_{1,3}$ as before in Figure 1.1) are linked to Z and the fourth of which (R_1) remains pending. This makes a pattern for what I called a partial-whip[1] in [PBCS], where R_1 is called a right-linking candidate. CSP-Variable V_2 (different from V_1) is supposed to have three remaining candidates (L_2, $C_{2,2}$ and $C_{2,3}$), the first two of which are linked to R_1 and the last of which is linked to Z.

It is easy to see that if Z was True, V_1 would have only one remaining possible value (R_1), which would imply that L_2 and $C_{2,2}$ would be eliminated, leaving $C_{2,3}$ as the only possible value for V_2. But $C_{2,3}$ is made impossible by Z. This proves that candidate Z is impossible and can be eliminated. This can be written symbolically as: whip[2]: $V_1\{L_1\ R_1\}$ — $V_2\{L_2\ .\}$ => not Z, where L_2 is chosen as the second left-linking candidate. $C_{2,2}$ is called a t-candidate, because it is linked to a previous right-linking candidate. In this case, the roles of L_2 and $C_{2,2}$ could be interchanged. *The dot in the final cell represents the impossibility for this cell to have a value.*

This is exactly what the whip[2] pattern is: a simple extension of the whip[1].

1.7.3 A partial-whip[2] and a whip[3]

It's now easy to generalise this to longer whips.

Consider first the partial-whip[2] obtained from Figure 1.2 by adding a pending candidate R_2 to its last CSP-Variable, V_2. Figure 1.3 gives a symbolic representation for a whip[3]. A new CSP-Variable is added to the right and all its candidates are linked either to Z or to the right-linking candidates for the previous CSP-Variables, i.e. to R_1 or to R_2. L_3 is chosen as the left-linking candidate (linked to R_2), $C_{3,2}$ and $C_{3,3}$ are t-candidates (linked to a previous right-linking one). $C_{3,4}$ is a z-candidate (linked to the target Z). There are none in this graphics, but V_2 could also have z-candidates and V_3 could have more than one. It might also be the case that the last CSP-Variable V_3 has no candidate linked to Z, if all its candidates are linked to a previous right-linking candidate.

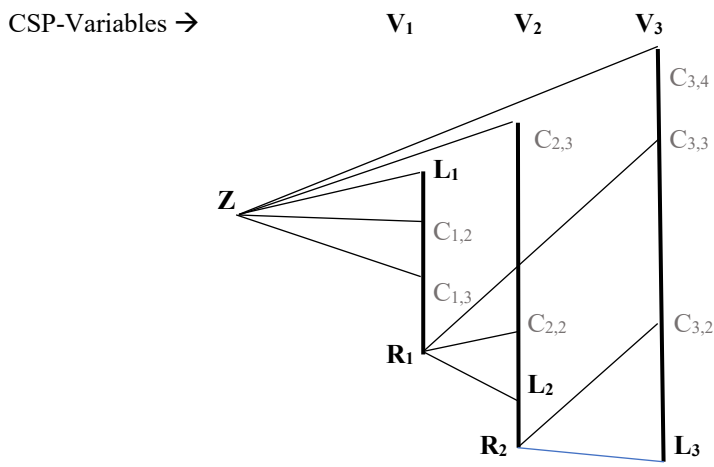


Figure 1.3. *Symbolic representation of a whip[3]*

The whip[3] in Figure 1.3 can be written symbolically as:
whip[3]: $V_1\{L_1\ R_1\}$ — $V_2\{L_2\ R_2\}$ — $V_3\{L_3\ .\}$ => not Z
and this can be seen as a symbolic representation of a first order formula (precisely written in [PBCS]) involving only the mentioned CSP-Variables and candidates.

1.7.4 A partial-whip[2] and a t-whip[3]

There is a second way a partial-whip[n-1] can be used for eliminations. The first way we have seen in section 1.7.3 is by extending its tail so as to obtain a whip[n]. The other way is by adding it a head so as to turn it into a t-whip[3]; and the head has to be a partial-whip[1].

In Figure 1.4, the whip[2] of Figure 1.2 has first been transformed into the same partial-whip[2] as in Figure 1.3, by adding a pending candidate R_2 to CSP-Variable V_2. Its ex-target Z has been renamed R_0 (because it will not be the target of the t-whip[3]). A partial-whip[1] for some target Z and CSP-Variable V_0 has been added to the left. The pending candidate for the unique CSP-Variable V_0 of this new partial-whip[1] must be R_0. And, in order to have a complete t-whip[3], its target Z must be linked to the last pending candidate of the previous partial-whip[2], namely R_2. I leave it as an easy exercise for the reader to prove that in this situation, Z can be eliminated.

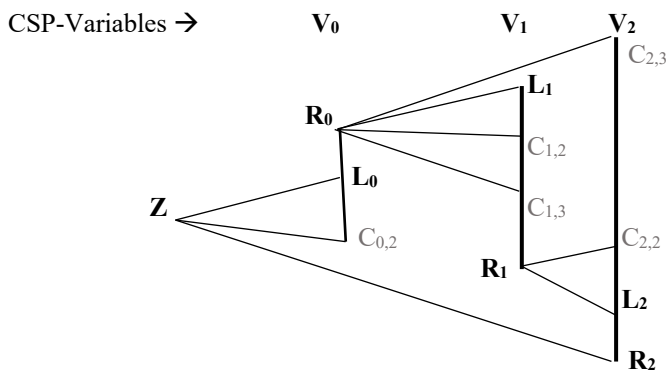


Figure 1.4. *Symbolic representation of a t-whip[3]*

The t-whip[3] in Figure 1.3 can be written symbolically as:
t-whip[3]: $V_0\{L_0\ R_0\}$ — $V_1\{L_1\ R_1\}$ — $V_2\{L_2\ .\}$ => not Z

Some users prefer t-whips because the main part of their construction does not depend on their final target(s). Unfortunately, they are less powerful than whips (as is obvious from their construction). Anyway, understanding the *simple relationship between partial-whips[n-1], whips[n] and t-whips[n]* is important for a human

solver, because it means that, when he has found a partial-whip, he can try to extend it both ways. It is technically important also in the implementation of CSP-Rules, because the partial-whip structures are indeed shared between whips and t-whips.

1.7.5 Partial g-whips[1], g-whips[2], braids, g-braids

Whips[1] allow another kind of generalisation. In Figure 1.2, suppose that, instead of being a candidate, R_1 is a g-candidate for V_1, i.e. a particular group of candidates for V_1 such that each of these candidates is linked to both L_2 and $C_{2,2}$. It is clear that the same reasoning as above still applies (using reasoning by cases for $C_{1,4}$) and the same conclusion allows to eliminate Z. This is what a g-whip[2] is. The same remarks apply to R_2 and R_3. Again, I will not dwell on details or on extension to longer g-whips, but I'll refer you instead to [PBCS]. The existence of g-whips in a particular CSP is conditioned on the existence of whips[1].

Finally, braids and g-braids allow another type of extension, where the continuity condition is relaxed: in Figure 1.3, imagine that, instead of being linked to the pending candidate R_1 for the previous CSP-Variable V_1, "left linking candidate" L_2 for V_2 (and its possible substitute $C_{2,2}$) were linked to Z; or that, instead of being linked to the pending candidate R_2 for the previous CSP-Variable V_2, "left linking candidate" L_3 for V_3 (and its possible substitute $C_{3,2}$) were linked to Z or to a more distant pending candidate, e.g. R_1. It is easy to see that the proof for the elimination of Z remains valid. We have lost the simple form of continuity between the candidates making a single-threaded whip, but we have some "braided" continuity instead (still with no OR branching). However, this easy modification drastically increases the computational complexity, making the statistical approximation results of braids by whips invaluable in practice.

1.7.6 How to start looking for a chain pattern

In [PBCS], all my definitions of chain patterns (except bivalue-chains) start from their target Z. There is a good reason for this: it makes the definitions simpler. One shouldn't conclude from this that the starting point of finding a chain is trying each candidate in turn as a possible target. Nothing could be farther from the real practice of solving puzzles with whips or other chains. [PBCS] is a logical presentation of a general resolution framework, of various types of resolution rules and of their resolution power. Even if part of the book consists of applying this to a lot of different logical puzzles, by showing how to define them so that they fit in the general framework, it doesn't say much about how to find the chain patterns in real cases: [PBCS] is not a tutorial on how to find these chains in a real CSP instance.

As the above graphico-symbolic representations should clearly show, *the real starting point for a whip/braid... is a partial-whip[1] and the real starting point for a g-whip/g-braid... is a partial-g-whip[1]*. These two patterns considerably restrict the number of candidates that could be used as potential targets. This should also

make it clear that *whips, g-whips, braids... are patterns in exactly the same sense as Subsets are. They do not make any assumption; they do not start with any hypothesis that some candidate might be a target. They start by observing that there is a well-defined pattern (a partial-whip[1] or a partial-g-whip[1]) in the current resolution state.* And these two starting patterns are as simple as the simplest two patterns after Singles, namely bivalue cells and whips[1] (a pattern known under various names in Sudoku: row-block interaction / intersection / pointing + claiming / alignments...).

1.8 General warnings

- CSP-Rules interface is text-only and it is not interactive. One types a simple command line in a simple Lisp-like syntax asking for some solution and the output is the full resolution path in plain text. No blinking, no colours, no frills, no nothing. If you cannot do without a graphical interface, you would be better inspired to look for something else or to propose your own graphical extensions via GitHub, because I am not going to write one. CSP-Rules was designed as a research and teaching tool; several times, I gave my AI students a part of SudoRules plus some (generally vague and incomplete) definitions of rules found on websites and I asked them to formalise them, first in English or French, then in First Order Logic and then to code them in CLIPS. I made it public in July 2020 because of an increasing number of readers of [PBCS] and [HLS] insisting on giving it a try. (I took advantage of the first lockdown in France to concentrate on writing the necessary additions and write the first edition of this Manual to make it usable by others than me. Every bad has its good counterpart.)
- CSP-Rules hasn't been designed for fast solving but for ease of implementation (for me). If what you need is a fast solver, you are not looking at the right place. Even the included DFS or T&E simulators are very slow compared to ones programmed in an optimal way in C. Thinking specifically of Constraint Satisfaction, there are much faster (and free) generic CSP solvers – though none (as far as I know) that will output a readable resolution path. Although they all use the same rules, there are also differences of speed between the various existing CSP-Rules applications, e.g. between KakuRules (fast) and SlitherRules (slow).
- As any knowledge-based system, CSP-Rules may require a fair amount of memory for hard puzzles. However, all the examples appearing in [PBCS] have been solved with only 16 GB of RAM and most of them require much less.
- If you are a "normal" puzzle solver, i.e. you are not submitting to CSP-Rules puzzles that any player would not even consider solving himself, you can completely forget the above-mentioned limitations of CSP-Rules in terms of speed or memory. Instead, you can play with the various combinations of rules and see what they allow. The possibilities are almost unlimited.

1.9 Selected generalities about CLIPS

Although the reader of this Manual does not need to know much about CLIPS, it may be useful when typing a user function to remember a few basic facts. Don't worry if you have never used any programming language: you don't have to understand this section in order to use CSP-Rules at the ordinary user level; you can totally skip it. The only two things you will have to do is to use a text editor to delete semi-colons in the configuration files and to type something like:

(solve "4...3.....6..8.....1....5..9..8....6...7.2.....1.27..5.3....4.9.....");

where the string is some more or less standard representation of a puzzle. (See the application-specific chapters for details.)

What follows is more for programmers of classical languages, so that they don't get lost with the specificities of CLIPS.

- If you have a look at the resolution rules coded in CSP-Rules, you will see that the CLIPS rules syntax is an elaborated version of First Order Logic. The main part of CSP-Rules is made of rules: ***the rules are the program***; it is the role of the CLIPS inference engine to apply properly these rules. The parts of CSP-Rules that are not made of rules are of two kinds: output functions and interfaces of the various applications that feed the givens of a problem instance to the rules of CSP-Rules.
- CLIPS general syntax, as you will mainly see it in the user functions, is Lisp-like. Lisp is a functional language. A function is not called as f(x) but as (f x): opening parentheses are before the function symbol.
- Variables names always start with "?". Global variables names always start with "?*" and always end with "*".
- The last argument of a function may be a list; in the function definition it has to be written with leading characters "$?" as in "$?rest" (the rest of the arguments); in a call to the function, all the arguments remaining after those that are associated with individually named variables are automatically made into a list and assigned to variable $?rest. It is important to remember this, because many user functions in CSP-Rules will use this very convenient possibility. There can be only one list variable in the definition of a function and it can only be the last argument.
- Any positive number of consecutive spaces, tabs, carriage returns or line feeds is equivalent to a single space; this has the advantage of allowing nice formatting of the programs or the arguments to functions, at no cost for the programmer.
- A semi-colon is a reserved character. Anything in a line after a semi-colon is a comment and can be considered as non-existent as far as the effective code is concerned. It is common usage (but not mandatory) to use three semi-colons for comments at the start of a line. Comments can appear at almost any place inside a function:

```
;;; Use this function to solve a puzzle
(solve
   ;;; I am a general comment about function "solve"
   9 ; I am the size of the grid
```

```
   Hidato ; I am the type of game (Numbrix vs Hidato)
   ; argument3 ; in this version, I am no longer an argument
   1 5 6 12 ; these final arguments make a list variable
)
```

There are very strange consequences of this rule when a semi-colon appears inside double quotes; but this will never happen in CSP-Rules (because it leads to what I consider unpredictable behaviour). Just for fun, you can play with this:

```
(printout t "; I am a string" crlf)
```

By the way, if you are blocked after typing something like that (i.e. if the CLIPS prompt does not reappear), try typing several closing parentheses. If it does not work, try typing a double quote (only one) and then as many closing parentheses as necessary. This will avoid to quit CLIPS in a savage way – i.e. with a control-c in Unix –, the proper way being by typing "(quit)".

As a general recommendation, if you have to type anything in the CLIPS interpreter, it is much better to type it first in any text editor and then to paste it into CLIPS: it is very easy to miss or add some double quote or some semi-colon or some parenthesis and it may lead you to a blocked situation similar to the above-mentioned one; once a line is entered into CLIPS, there is no way to go back.

- There are six CLIPS commands that it may occasionally be useful to know:

1) (rules) will output the set of current rules in CLIPS (the knowledge base);

2) (facts) will output the set of current facts in CLIPS (the facts base);

3) (clear) will empty CLIPS of all its facts and rules. You can use this instead of quitting CLIPS before loading another set of rules. It is equivalent to quitting and re-launching, but in practice I usually prefer quitting and re-launching, especially after an exceptionally hard puzzle that required large amounts of memory.

4) (reset) will clear all the facts but not the rules. In CSP-Rules, the behaviour of (reset) is fixed so that it does not change the values of the global variables and it is important not to change this. A reset is done automatically by CSP-Rules every time a "solve" function is called and it involves releasing memory no longer used. Memory is managed in a very safe way in CLIPS: many times, I have left a single instance of CLIPS run for full days to solve hundreds of thousands after hundreds of thousands of puzzles without ever needing to quit and restart it.

5) (release-mem) allows to release the memory claimed by CLIPS during resolution. This may be useful after a very hard puzzle has been solved. But this is also done automatically before each new puzzle is solved.

6) (dribble-on "filename") allows to redirect all the output (normally sent to the Terminal / Command Window) to the file named "filename". (dribble-off) stops this behaviour and sends again the output to the standard output.

Part One

GENERALITIES

2. Installing and launching CSP-Rules

CSP-Rules is coded as a knowledge-based system and it requires an independent inference engine to run its sets of rules. The "programming" language used by CSP-Rules is that of the CLIPS inference engine, with a Lisp-like syntax. The present Manual will not explain how to use CLIPS beyond launching it – the only thing strictly necessary to run CSP-Rules. Indeed, in order to make CSP-Rules easy to use for anyone without requiring preliminary compilations (which can be quite challenging for non-programmers on Windows), executable versions of CLIPS for Unix (including MacOS) or Windows environments are provided with CSP-Rules-V2.1 (in the "CLIPS" folder). The relevant CLIPS source code is also provided.

2.1 Installing CSP-Rules (machine independent)

Create a "CSP-Rules" folder at any reasonable place in your file system, e.g. within a "Tools" directory inside your home directory. Download CSP-Rules-V2.1.zip from the GitHub repository: go to https://github.com/denis-berthier/CSP-Rules-V2.1/master, click the "Code" button to open its menu and then select "Download ZIP"; move the downloaded file into your newly created "CSP-Rules" folder, unzip it (by double-clicking it) and rename it as "CSP-Rules-V2.1" (in particular, delete the final "master" part, if any). At this point, the absolute path to CSP-Rules-V2.1 should look like this:

/<absolute_path_to_your_home_directory>/Tools/CSP-Rules/CSP-Rules-V2.1/

Now, you will need your preferred text editor to modify a few files.

Modify as follows each of the configuration files of the predefined applications (i.e. all the files inside the "CSP-Rules-V2.1" folder with names ending in "config.clp"). The three modifications listed below are in the upper block of each configuration file, under the local banner "INSTALLATION ONLY". The four lines are marked with an ending "<<<<<<<<<<<" sign. You can modify one of the configuration files and then copy/paste this whole upper block onto the corresponding blocks of the other files so as to replace them.

1 – mandatory in odd cases) If you are using any odd system (other than MacOS, Unix or recent Windows) that doesn't use "/" as its directory symbol, then define it here, e.g.

```
; (bind ?*Directory-symbol* "\")
```

2 – absolutely mandatory) Change the definition of the ?*CSP-Rules* global variable, so that it coincides with the absolute path to your previously defined CSP-Rules (not CSP-Rules-V2.1) folder (including the final directory symbol "/", otherwise it will not work); for instance:

- on a Unix system:

```
(defglobal ?*CSP-Rules* =
"/Users/<your_home_directory>/Tools/CSP-Rules/")
```

Beware: CLIPS does NOT understand the Unix "~" as a shortcut to your home directory.

- on a Windows system:

```
; (defglobal ?*CSP-Rules* =
"c:/Users/<your_home_directory>/Tools/CSP-Rules/")
```

With this way of proceeding, CSP-Rules can be run from any place.

3 – optional) If necessary (i.e. if you want to use a release of CLIPS different from the r790 one provided with CSP-Rules), change the CLIPS version number (this is only used for keeping track of which release of CLIPS you have used during resolution and it will appear only in the CSP-Rules banners); it will not change anything else if you forget to do this:

```
(defglobal ?*Clips-version* = "6.32-r790")
```

4 – optional) Describe, with all the details you like, the machine you will use to run CSP-Rules. This will only be used for keeping track of this information and for printing it at the end of each resolution path, in the CSP-Rules banner:

```
(defglobal ?*Computer-description* =
"MacBookPro Retina Mid-2012 i7 2.7GHz 16GB, MacOS 10.5.4")
```

5 – additionally, if you are using a Mac with a recent version of MacOS, you will have to recompile your own version of the CLIPS core, due to the strict MacOS security rules. This is why the source of the CLIPS core is now included in the CLIPS folder. In a Terminal, go to the CLIPS/clips-core/ directory and type "make". You will get a "clips" executable file in this directory; move it up to the CLIPS folder of CSP-Rules-V2.1 (thus crushing the previously existing clips file).

Et voilà. CSP-Rules is ready to run.

Notice that, if you download an updated version of CSP-Rules-V2.1 from GitHub, you will have to re-start the same installation procedure.

2.2 Launching a CSP-Rules application

Each application *App* consists of two parts: a directory named *App*Rules-V2.1 plus a configuration file named *App*Rules-V2.1-config.clp. If you frequently use several different configurations for the same application, there is no restriction to

having several different configuration files for it. My recommendation is to always name them according to the pattern *App*Rules-V2.1-config-xxx.clp, where xxx can be any mnemonic for your configuration.

To launch and use an application, there are four steps:

Step 1) Launch CLIPS by double-clicking the proper application in the CLIPS subdirectory of CSP-Rules-V2.1, i.e. "clips" on a Unix (including MacOS) system and "clips.exe" on a Windows system. The starting of CLIPS is immediate. This is indicated by the standard CLIPS prompt: "CLIPS>".

Step 2) In the relevant config file, specify which options and which resolution rules you want to use [how to make such choices consistently will be seen in the relevant chapters of Part II]. A choice by default is already made in the included configuration files, so that you can start using CSP-Rules without any delay.

Step 3) Inject the relevant configuration file into CLIPS. There are two ways to "inject" it:

- either copy the full content of the file and paste it directly into CLIPS (this works perfectly on MacOS, but I did not try on other systems);
- or type in CLIPS the following command line (with the relevant changes for the parts in italics):

```
(batch
"/Users/<your_home_directory>/Tools/CSP-Rules/CSP-Rules-V2.1/AppRules-config.clp")
```

Both ways amount to applying successively in CLIPS all the "constructs" and commands included in the configuration file. The configuration file sets a few global parameters and then loads all the files corresponding to the application and the chosen resolution rules, with the chosen options.

At this point, after some amount of useless messages (including normal warnings about functions and rules being redefined), the CSP-Rules banner should appear, with the first line recalling which application you are using, based on which version of CSP-Rules, which version of CLIPS (if you have configured it during installation) and which resolution theory. The banner is followed by the standard CLIPS prompt ("CLIPS>"). Here you can see my standard – and the default – configuration for SudoRules: whips + Subsets + Finned Fish (the ".s" means that the default option for rules, "speed", has been kept unchanged):

```
*****************************************************************************
*** SudoRules 20.1.s based on CSP-Rules 2.1.s, using CLIPS 6.32-r790, config = W+SFin
*** Copyright Denis Berthier
*** Running on MacBookPro Retina Mid-2012 i7 2.7GHz 16GB 1600MHz DDR3, MacOS 10.15.4
*****************************************************************************
CLIPS>
```

The information thus displayed may seem useless to you now, but if you solve hundreds or thousands or (as I have done) millions of puzzles, keeping track of it together with the actual resolution paths may spare you the necessity to re-run them in case you don't remember.

CSP-Rules is now ready to solve puzzles for you. If you want to check which rules have been loaded, you can type in the CLIPS command: "(rules)". The last line of the output will be the total number of rules. Notice that what is called a resolution rule in [PBCS] is generally implemented via several CLIPS rules (generally an activation rule, a tracking rule, one or more pattern definition/extension rules and one or more rule-application rules), so that what you get here is the number of CLIPS rules, not the number of resolution rules in the sense of [PBCS].

Step 4) Decide which puzzle P you want to solve and type something like:
`(solve "……the_puzzle……")`
where "……the_puzzle……" must be the proper representation of P for this type of puzzle in CSP-Rules. Depending on the application, one or more additional parameters may be needed, such as size of the problem, variant of the puzzle… More application-specific commands will be described in Part II, but function "solve", or some form of it, is the basic command for all the applications. Go directly to the application-specific chapter for details about its syntax and a direct use of CSP-Rules.

The result will be the full resolution path for the chosen puzzle. Note that the computation times can be extremely different for different types of puzzles and for different instances of a given type. A good way of getting used to this is to try progressively harder puzzles.

Step 4 can be repeated as many times as you want, as long as you don't change the type of puzzle, the options or the set of resolution rules you want to apply. The "cleaning" of the CSP-Rules "memory" between the resolution of two puzzles is automatic, but you can apply it yourself by typing "(reset)" and then "(release-mem)" when the CLIPS prompt re-appears.

However, ***if you want to change the set of resolution rules or the options or the type of puzzle (and for some applications such as Sudoku, the size of puzzles), you have to re-start from step 1*** – because, ***for efficiency reasons, only the rules selected in the configuration file are loaded when you "inject" this file into CLIPS at step 3***.

3. The generic resolution rules and functions in CSP-Rules

CSP-Rules V2.1 includes most of the generic resolution rules mentioned in [PBCS], plus (some generic form of) the more specific rules ("2D-chains") previously defined in [HLS]. It also includes rules specific to some applications, that will be discussed in the relevant chapters, in Part Two of this Manual.

3.1 Basic generic resolution rules

The first generic resolution rules present in CSP-Rules V2.1 constitute the universal Basic Resolution Theory (BRT) defined in [PBCS] and, as such, they cannot be turned off. They are:

- ECP (Elementary Constraints Propagation from decided values to candidates); their conclusions are so obvious that they are generally not printed (unless one explicitly sets global variable ?*print-details* to TRUE in the configuration file);
- Singles;
- Rules for managing the resolution process, such as contradiction detection, solution detection, … (see [PBCS] for details). They act unnoticed by the ordinary user.

Note also that each application may (and some do effectively) overwrite the generic version of ECP and/or Singles or even whips[1], for efficiency reasons. All this is transparent to the ordinary user.

3.2 Ordinary generic resolution rules

First, note that Subset rules (and their variants, such as Finned Fish in Sudoku) are not provided in a generic form in CSP-Rules, although they could be programmed as set covering rules. But such a general form would be very inefficient (and it would cause quick memory overload). Instead, they have application-specific versions, but there are generic hooks to deal with them (see the relevant chapters).

As a result, the main types of resolution rules introduced in [HLS] and [PBCS] that are present in generic form in CSP-Rules V2.1 are various kinds of chains (or "braided chains"). They are:

- bivalue-chains, up to length 20;
- typed bivalue-chains, up to length 20;
- z-chains, up to length 20;
- typed-z-chains, up to length 20;
- oddagons, up to length 15;
- t-whips, up to length 36; remember that t-whips are a special case of whips, they are less powerful, but they can be built independently of the target z;
- typed t-whips, up to length 36;
- whips, up to length 36;
- typed whips, up to length 36;
- g-bivalue chains, up to length 20;
- g2-whips, up to length 36;
- g-whips, up to length 36;
- braids, up to length 36;
- g-braids, up to length 36;
- forcing-whips and forcing-g-whips, up to length 36;
- forcing-braids and forcing-g-braids, up to length 36.

I set the above-mentioned maximum lengths for each kind of chains because I never needed anything close to so long chains when solving millions of puzzles of different types. In practice, you will never see so long chains.

All of the above chain rules, except oddagons (see below) and g2-whips, come in two independent versions: speed and memory, corresponding to different optimisations. g2-whips have only the memory version, because they are supposed to be used mainly when g-links haven't yet been defined. Forcing-whips, forcing-g-whips, forcing-braids and forcing-g-braids have apparently only one version, but they are built on the chosen speed or memory versions of whips, g-whips, braids and g-braids, so that, in practice, they do have two versions also.

A small part of the resolution rules also come in two slightly different versions (which may make a total of four different versions for some of them): an ordinary one that works and prints the output as described in [PBCS] (and as in V2.0 – i.e. only one elimination at a time) and a new "blocked" one that finds all the potential targets for an instantiation of the rule before eliminating all of them in a single sweep. In this new behaviour, rules will no longer seem to be "interrupted" by simpler ones (and sometimes to re-start later). After experimenting with this

behaviour, I made it the default one in version V2.1, but each configuration file allows to revert to the previous [PBCS] (and V2.0) behaviour if that is what you prefer. Rules amenable to this new "blocked" behaviour are whips[1], all the (typed or not) bivalue-chains, all the (typed or not) t-whips and all the Subset rules in any of the coded applications; they can be independently reset to the old behaviour. Application-specific rules for Slitherlink were coded from the start with this new behaviour and they have no other version.

Note: *oddagons* were not discussed in [PBCS]. They are an oddity among chains. They first appeared in the Sudoku world. They can be defined as follows: for any candidate Z (the target of the oddagon), for any odd integer $n \geq 3$, an oddagon[n] is defined by:

- a sequence of n different CSP-Variables csp_1, csp_2, …, csp_n,
- a continuous sequence of different candidates c_1, c_2, …, c_n,

such that, setting $c_{n+1} = c_1$ and $csp_{n+1} = csp_1$, one has for every $1 \leq k \leq n$:
c_k and c_{k+1} are candidates for csp_k and they are the only two candidates for csp_k that are not linked to Z.

In such conditions, the oddagon chain rules states that Z can be eliminated. The proof is very simple, by circulating along the chain: if Z was True, c_1 True would imply c_1 False and c_1 False would imply c_1 True.

3.3 A word on typed chains

Typed bivalue-chains, typed z-chains, typed t-whips, typed whips and typed g-whips were not discussed in [PBCS], but they were introduced in [HLS] under other names. In Sudoku, they are the "2D" counterparts of bivalue-chains, z-chains, t-whips, whips and g-whips (see section 6.4).

The difference of any of these typed chains with its untyped version is, all its CSP-Variables must have the same type (e.g. rc, rn, cn or bn in Sudoku); however, typed-chains of different types can appear in a resolution path – unless this possibility is further restricted, using a simple generic mechanism (see the application-specific chapters, in particular the SudoRules chapter, for details). Obviously, the notion of a typed-chain is less general than that of a chain of the same kind (i.e. bivalue-chain, z-chain, t-whip, whip or g-whip), but it may be used for many different purposes, depending on the applications and on one's goals.

In a resolution path, a typed chain (e.g. a typed t-whip) is printed as a chain of the same kind (a t-whip), with its type appended between its kind and its length (e.g. t-whip-rc[5]); the word "typed" does not appear at the start, because it would be redundant with the explicit naming of the type at the end.

3.4 Resolution rules for simulating T&E (Trial and Error)

CSP-Rules V2.1 also provides a rule-based version of T&E at depths 1, 2 and 3. (Notice that it is very inefficient, compared to what an optimised C version could do.) This is mainly designed:

- to check (e.g. before launching the actual resolution) whether a puzzle is in T&E(1), i.e. whether it can be solved by braids (and then, probably by whips),
- to check (e.g. before computing its B?B classification) whether a puzzle is in T&E(2), i.e. whether it could be solved by B-braids,
- to find the B?B classification of a puzzle in T&E(2),
- to find the S?B classification of a puzzle in T&E(S, 1), when application-specific Subset rules are defined in some application.

In principle, T&E can be used in combination with any resolution theory T (i.e. any set of ordinary resolution rules), provided that T has the confluence property, in order to check whether a puzzle is in T&E(T, 1), T&E(T, 2)… Notice however that using T&E(T, 1) amounts to solving hundreds of puzzles and it can take much time if T is complex. T&E(T, 2) will roughly take almost the squared time of T&E(T, 1).

In practice, the configuration files consider using T&E for only three purposes:

- finding the T&E-depth of a puzzle P, i.e. its membership in T&E(BRT, k) = T&E(k), the useful result being when k = 1 or k = 2 (k = 0 is trivial). k = 1 ensures the existence of a solution with braids (thanks to the T&E vs braids theorem). Most of the time in this case, there is also a solution using only whips. But sometimes, braids are really necessary; such puzzles are generally very hard and you can expect very long resolution times and memory requirements with braids; it is often faster to look for a solution with g-whips – though being in T&E(1) is not an absolute guarantee for a g-whip solution;
- finding the smallest k such that P is in T&E(S_k), i.e. it is solvable by S_k-braids;
- finding the smallest k such that P is in T&E(B_k), i.e. it is solvable by B_k-braids.

3.5 Resolution rules for simulating DFS (depth-first-search)

DFS is known to be the fastest method for solving 9×9 Sudoku puzzles. CSP-Rules V2.1 also provides a (very inefficient) rule-based version of DFS. Originally, this was mainly designed for obtaining solutions faster than with the true rule-based methods. DFS can be combined with any set of resolution rules (having the confluence property), but it is generally faster if one only allows whips[1] and possibly whips[2] or whips[3], depending on the application. Notice that the computation time of DFS increases very fast (i.e. exponentially, for all practical purposes) with the size of the puzzle and it does not depend in any measurable way on its W, B, gW, gB… complexity.

3.6 Resolution rules for finding backdoors and anti-backdoors

If T is a resolution theory with the confluence property, one can define a T-backdoor [respectively a T-anti-backdoor] as a candidate that leads to a solution in T if it is asserted as a value [resp. deleted]. Similarly, a T-anti-backdoor-pair is a couple of candidates leading to a solution in T if both are deleted. CSP-Rules contains the rules necessary for finding them. The generic functions for launching these rules, ***find-backdoors***, ***find-anti-backdoors*** and ***find-anti-backdoor-pairs*** (with no argument), ***find-anti-backdoor-pairs-with-one-cand-in-list*** (with one argument: a list of candidates such that each pair must contain at least one of them), work in the current resolution state and with T chosen as the current set of ordinary resolution rules.

Anti-backdoors and anti-backdoor-pairs can be used to find 1-step or 2-step solutions. While the necessary rules and functions are written in generic form, at this point, I have tested them only on Sudoku and they will be explained in detail in the SudoRules chapter.

3.7 Resolution rules for simulating Forcing-T&E and Forcing{3}-T&E

In [CRT] and [PBCS], I introduced two patterns I called forcing-whips (resp. forcing-braids), made up of a (rc, rn, cn or bn) bivalue cell and of two whips (resp. two braids) with respective targets the two candidates in the bivalue cell.

I noticed that the associated resolution rules (eliminating common left-linking candidates and asserting common right-linking candidates) were of little use in practice, because in all the cases I tried, a solution with shorter whips/braids was available. I have no new data allowing to change my conclusions about this result.

Three things I didn't consider at the time of the above publications were:

- a definition of Forcing-T&E(cand1, cand2): start from a bivalue pair of candidates (cand1, cand2), develop independently the T&E branches starting from each candidate, eliminate all the candidates eliminated in both branches and assert all the candidates asserted in both branches.
- a proof that any elimination/assertion done by Forcing-T&E(cand1, cand2) can be done by some Forcing-braid(cand1, cand2) – the converse being obvious as usual.
- an extension of these definition and result to any theory T with the confluence property.

I will not insist on this, because all three points are obvious consequences or extensions of my various definitions and T&E vs braids theorems.

Forcing-whips/braids are not very interesting as long as they respect the natural notion of length and as long as one uses the simplest-first strategy that rely on it.

However, in the Sudoku world, there has been recently some interest for various forcing-things of uncontrolled length. I have therefore coded Forcing-T&E in a generic form in CSP-Rules. Contrary to Forcing-whips or Forcing-braids, there is no restriction on length and Forcing-T&E does not require that whips or braids be activated.

Contrary to Forcing-whips/braids, Forcing-T&E is not submitted to the simplest-first strategy. But it allows some optimisation of the number of steps in the resolution path. You can see an example of application in section 6.18. In some restricted sense, Forcing-braids and Forcing-T&E are submitted to opposite optimisation strategies: minimum length of chains and uncontrolled number of steps versus (partially) minimised number of steps and uncontrolled length of chains.

The principle for choosing a pair of candidates as a starting point for the next application of Forcing-T&E is the maximum number of eliminations common to the two T&E branches. Notice that this "optimisation" is purely local.

Forcing{3}-T&E is the generalisation of Forcing-T&E where one starts from trivalue triplets of candidates and develops three independent paths instead of bivalue pairs and two independent paths. Of course, anything (i.e. assertion of a value or elimination of a candidate) common to the three paths is globally valid. This is merely reasoning by cases, with three branches instead of two.

3.8 Hooks for introducing new resolution rules

Although CSP-Rules-V2.1 does not have generic Subset rules, it provides hooks for dealing with CSPs defined on rectangular grids with rows and columns, as is often the case for logic puzzles. In particular, it defines the *nrc-notation* for the output of every rule, as a specialisation of the general CSP-Rules output mechanism. All the applications discussed in this Manual use this nrc-notation (or some direct variant of it when there is no underlying rectangular grid, such as in MapRules).

3.9 Generic functions in CSP-Rules

The following functions have a generic definition, according to which they do nothing; each application provides a specific implementation: init, solve, compute-current-resolution-state (abbreviated as compute-RS), print-current-resolution-state (abbreviated as print-RS), pretty-print-current-resolution-state (abbreviated as pretty-print-RS), init-resolution-state (abbreviated as init-RS), solve-resolution-state (abbreviated as solve-RS).

4. Simplest-first strategy, saliences, ratings

As recalled in the introduction, given a fixed set of resolution rules, the standard resolution process of CSP-Rules is guided by the ***"simplest-first" strategy***: at any stage of the resolution process, the next rule that will be applied is the simplest available one (or some arbitrarily selected one among the simplest available ones in case several rules of same minimum complexity can be applied). The main advantage is, ***for any resolution theory T with the confluence property, this strategy guarantees that a single resolution path is enough to provide a solution with the smallest rating with respect to T***).

However, CSP-Rules is not intended to be a normative system. As an automatic solver, it must have some strategy; it finds a resolution path according to the rules selected in the configuration file and to the priorities defined by its simplest-first strategy (and to the arbitrary choices made at points where several equally complex rules could be applied). It doesn't mean that you are wrong if, as a manual solver, you find a different path.

In addition, the Sudoku chapter will show in detail how this strict simplest-first strategy can be partly circumscribed in various ways by using functions or combinations of functions more elaborate than "solve". For instance, it is now possible to focus the eliminations on particular candidates. It is also possible to look for resolution paths leading to a solution with a single rule application (1-steppers) or with two rule applications (2-steppers). Of course, this will work only for relatively easy puzzles. More generally, there is also a possibility of looking for resolution paths with fewer steps (within a given resolution theory). This is currently available only for Sudoku, but most of it can easily be re-written in generic form.

4.1 The simplest-first strategy

The simplest-first strategy could be implemented in many different ways in CSP-Rules, but I have chosen to use CLIPS "saliences" (i.e. priorities between rules) in a systematic way. The use of too many saliences in the knowledge base (i.e. the set of rules) of a knowledge-based system is generally not very well considered in the AI community, but this is the most versatile way of allowing additional (application-specific or user-specific) rules to be finely inserted at the proper level of the complexity hierarchy if an occasional user wants to write some rule(s) of his

own, without requiring a deep understanding of the system. Moreover, from a more theoretical point of view, it allows to make this hierarchy explicit and thus to keep totally separate:

- the "*knowledge level*" (in the sense of [Newell 1982]), defined by the textual and/or logical formulations of the resolution rules, as written in [HLS] or [PBCS];
- the "*symbol level*" (also in the sense of [Newell 1982]), defined by their implementation as a set of CLIPS rules;
- the "*strategic meta-level*" (introduced in all the modern methodologies of Rule-Based systems development) of how the rules defined at the knowledge level are to be used in a resolution process; as said before, the meta-level corresponding to the default simplest-first strategy is implemented in CLIPS as a set of saliences.

As the main interest of this point is for the user who wants to extend some existing CSP-Rule application or to add a new one, details will only be given in a more advanced manual. What I'll say in this chapter intends mainly to explain why some patterns appear before other ones in the resolution paths, in (the frequent) cases when different possibilities are available.

General priorities between existing resolution rules have been described in [PBCS]. They are organised by levels, each level being defined by the number of CSP-Variables involved in the defining pattern (i.e. the conditions of the rule).

Within each level, priorities are mainly based on the generalisation relation, where a special case is given higher priority (otherwise, it would never appear in the resolution paths). For every k > 1, one has:

oddagon[k] > biv-chain[k] > z-chain[k] > t-whip[k] > whip[k] > g-whip[k] > braid[k] > g-braid[k] > forcing-whip[k] > forcing-braid[k]

For the typed-chains, which were not considered in [PBCS], the most natural place is to insert them just before their untyped version. The place of forcing-whips and forcing-braids was not mentioned in [PBCS], but they naturally come at the end of this hierarchy.

Notice that z-chains[k] are placed before t-whips[k], though none is more general than the other. The reason is, although they depend on their target, z-chains are structurally simpler than t-whips.

Being an exotic and rare pattern, oddagons (when defined, i.e. for odd k) are placed before any other chain.

Notice also that g-whips[k] are placed before braids[k], though none is more general than the other. There are three reasons: they respect the structural continuity condition, they are computationally simpler patterns and they are statistically more likely to produce results not reachable by whips.

CSP-Rules allows an easy addition of application-specific saliences at each level, before or after the generic saliences. In general, application-specific rules based on k CSP-Variables should be given higher priority than the generic rules with the same number of CSP-Variables. Otherwise, they might never appear in the resolution paths, because most application-specific rules turn out to be particular cases of the generic chain rules. This assertion is not strictly true for Subsets, but it is true in most of the cases (for details, see the subsumption theorems for Subsets in [PBCS]).

If two rules with the same priorities are available, which of the two is used first is decided arbitrarily. This refers to conceptual arbitrariness, i.e. you cannot rely on any assumption about how the choice will be made, but if you run the same example with the same rules, you will get the same resolution path; if you want different paths, you may try to vary the rules, e.g. add or delete some types of special cases. Often, I do this by activating or de-activating z-chains and/or t-whips. In case the application has typed-chains, they can also be used that way.

Notice that having many different kinds of chains activated simultaneously has a cost in terms of memory and computation time and it is not recommended for the very hard puzzles. Also, the simplest-first strategy tends to produce steps that are not strictly necessary to reach the solution, the more so as more different types of rules are activated. Whether this should be considered as a secondary advantage (showing the user more opportunities) or as a disadvantage depends on one’s goals.

Part Two

RUNNING SPECIFIC APPLICATIONS

5. General structure of the configuration files

A user has two independent means of interacting with an application of CSP-Rules: its configuration file and its user functions (or commands).

The functions available to the user may differ from application to application, but there's always some form of a "solve" function (with its own list of arguments); it is the most basic function for solving an instance of the CSP under consideration. See the relevant chapter for each application.

As for the configuration files, they all have the same structure and they are almost self-explanatory. For definiteness, I take here the SudoRules configuration file as an example and I only add a few comments at the relevant places. For clarity, parts of this chapter copied from the SudoRules configuration file use a different, machine-like, font.

In the configuration file, the user chooses general options and which rules he wants to use; at the end of the configuration file, CLIPS is asked to load the rules corresponding to the user choices. For efficiency reasons, non-selected rules are not loaded, except the rules implied by the selected ones (such as braids => whips). ***The user functions are totally independent of the choice of rules or settings; they work the same way for any choice***; they will be described in each application-specific chapter. If you have no experience in programming, don't let it impress you: the only thing you may have to do is to use a text editor to delete a few semi-colons.

5.1 The first part of the configuration file is for installation

The first part of the configuration file is for installation and needs to be changed once and only once, as explained in chapter 2. If you have already done the installation, you can proceed directly to section 5.3 below. Otherwise, check the four lines marked with the ending signs "   <<<<<<<<<"). This part is identical for all the applications and can be copied from the one to the next.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; INSTALLATION ONLY:
;;; Define environment variables: OS, installation directory
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(clear) ; clean CLIPS of anything it may have had before.
;;; Default setting is for Unix and MacOS,
;;; but it should also work for recent versions of Windows:
(defglobal ?*Directory-symbol* = "/")

;;; Define your general CSP-Rules installation directory (including the ending
directory symbol /).
;;; This is the directory in which the CSP-Rules-V2.1 version is installed, not the
CSP-Rules-V2.1 directory.
;;; By defining the path in an absolute way, you will be able to launch CSP-Rules-V2.1
from anywhere.
;;; You need to write something as follows.
;;; For Unix (including MacOS):
 (defglobal ?*CSP-Rules* = "/Users/berthier/Documents/Projets/CSP-Rules/") ; <<<<<<<<
;;; For Windows:
; (defglobal ?*CSP-Rules* = "c:/Users/berthier/Documents/Projets/CSP-Rules/") ; <<<<<<

;;; CLIPS is the underlying inference engine.
;;; The version of CLIPS used may be defined here (used only for displaying it in the
banner)
(defglobal ?*Clips-version* = "6.32-r790");            <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;;; Description of the computer used for the resolution
(defglobal ?*Computer-description* =
  "MacBookPro Retina Mid-2012 i7 2.7GHz 16GB 1600MHz DDR3, MacOS 10.15.4")     <<<<<<<
```

5.2 The second part of the configuration file must not be changed

The next part of a configuration file is application specific in its details but has the same structure in all the applications. It just elaborates the user data defined in the first part and it pre-loads some global variables necessary for the user's upcoming choices. It should not be modified. For Sudoku, it looks like this:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Define the application
;;; Define useful directories and load all the globals
;;; (they must be available before choosing the configuration of rules)
;;;
;;; Do NOT change any of the following
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; an ?*Application* must be defined as the name of the application (here,
SudoRules), not as the name of the puzzle (Sudoku)
;;; this name must coincide with the leading part of the name of the directory for the
application inside the CSP-Rules-V2.1 directory
;;; the version number of the ?*Application* must also be defined
```

```
;;; this allows to have several versions of the same application based on the same
version of CSP-Rules
;;; for historical reasons, SudoRules version number inside CSP-Rules-V2.1 is not 2.1
but 20.1
;;; (there were many versions of SudoRules before the development of a generic CSP-
Rules core)
(defglobal ?*Application* = "SudoRules")
(defglobal ?*Application-VersionNumber* = 20.1)
(defglobal ?*CSP-Rules-current-version* = (str-cat ?*CSP-Rules* "CSP-Rules-V2.1"
?*Directory-symbol*))
(defglobal ?*CSP-Rules-Generic-Dir* = (str-cat ?*CSP-Rules-current-version* "CSP-
Rules-Generic" ?*Directory-symbol*))
(defglobal ?*CSP-Rules-Generic-Loader* = (str-cat ?*CSP-Rules-Generic-Dir* "CSP-Rules-
Generic-Loader.clp"))

(defglobal ?*Application-Dir* = (str-cat ?*CSP-Rules-current-version* ?*Application*
"-V" ?*Application-VersionNumber* ?*Directory-symbol*))
(defglobal ?*Application-Loader* = (str-cat ?*Application-Dir* ?*Application* "-
Loader.clp"))

;;; load declarations for the global variables necessary for the upcoming choices
(load (str-cat ?*CSP-Rules-Generic-Dir* "GENERAL" ?*Directory-symbol* "globals.clp"))
(load (str-cat ?*Application-Dir* "GENERAL" ?*Directory-symbol* "globals.clp"))
```

5.3 The third part of the configuration file is application specific

The next part of a configuration file is application specific and most of the time, it will be non-existent. Variants, if any, will be chosen in this part. In SudoRules, it looks like this:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; General application-specific choices
;;; Define grid size and related parameters
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Because ordinary Sudoku puzzles are 9x9 (i.e. segment-size = 3),
;;; SudoRules has its own way of dealing with the size of puzzles
;;; and this can only be done here; maximum segment size is 9
; (bind ?*segment-size* 4);                                    <<<<<<<<<<<<<<<<<<<<<<<
```

5.4 The fourth part of the configuration file defines the configuration

The next part of a configuration file is the most important one for the user. It allows you to define precisely which sets of rules you want to apply and with which options. With all the comments they include, most subparts are self-explanatory. As recalled in the banner of this part of the configuration file, it is important to have semi-colons deleted in only one section or sub-section of rules at a time.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;                         DEFINE YOUR CONFIGURATION HERE:
;;;                             Choose general settings
;;;        Define your resolution theory (i.e. your set of resolution rules)
;;;            Simply delete the leading semicolon of the proper line(s)
;;;                              IN ORDER TO AVOID ERRORS,
;;;              DELETE SEMI-COLONS IN ONLY ONE SECTION OF RULES AT A TIME
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4.1 Choose general settings

There are four kinds of settings:

- The rule optimisation type: speed or memory. Default is "speed" and should not be changed unless you face a memory overflow problem for some extremely hard puzzle. Notice that resolution will be slower if "memory" is chosen.
- The "blocked" behaviour for some rules (Whips[1], Subsets, bivalue-chains, t-whips and their typed versions) is now the default behaviour in CSP-Rules and in all the applications: when a pattern in the above families of rules is found, all its targets are eliminated as a group before other, simpler rules, are allowed to fire. However, the user may choose, independently for each of the above families of rules, to revert to the old behaviour (until V2.0 and as in [PBCS]), where candidates were eliminated one by one. The same choice applies to the typed and untyped versions of the rules when they both exist.
- The detail of the output. The names of the control variables are self-explanatory, except maybe ?*print-levels*: it allows to print a line each time the solver enters a new level of the solving process, i.e. when it activates a new type of rules or a longer length for a type already activated. For efficiency reasons, all the rules selected by the user in this configuration file (plus implied rules) are loaded at the start, but they are effectively activated only when necessary, i.e. when all the simpler rules have produced all that they can. (Once activated, they remain activated until the end.) This has the advantage of limiting memory requirements. For the hardest puzzles, it may be useful to display tracking information about these activations when nothing seems to happen (no elimination) for a long time; it shows that CSP-Rules is still working and (for those who want to know about such "technical details"), it gives an idea of how many CLIPS facts it is dealing with. By default, ?*print-levels* is now set to FALSE, because most users don't want to see such information.
- The printing of the resolution state at three points: a) after application of all the obvious steps, i.e. of all the rules in BRT (direct contradictions and Singles); b) after whips[1]; c) at the end if the puzzle is not solved by the current set of rules.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 0) Choose general settings
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; DON'T CHANGE ANYTHING IN THIS SECTION UNLESS YOU HAVE SOME REASON

;;; Possibly change the type of optimisation for the chain rules.
;;; Default is pre-defined as SPEED.
;;; Don't change this unless you meet a memory overflow problem.
; (bind ?*chain-rules-optimisation-type* MEMORY)

;;; In the previous standard behaviour of CSP-Rules, when a pattern could have
produced more than one elimination,
;;; the activation of a simpler rule by the first elimination could prevent further
potential eliminations.
;;; This default behaviour is now changed for Whips[1], bivalue-chains (typed or not),
t-Whips (typed or not) and Subsets.
;;; But CSP-Rules allows to revert to the previous behaviour,
;;; independently for Whips[1], for bivalue-chains and t-Whips of any length and for
Subsets.
;;; Un-comment the relevant line(s) below if you want these rules to be "interrupted"
as the other chain rules:
; (bind ?*blocked-Whips[1]* FALSE)
; (bind ?*blocked-bivalue-chains* FALSE)
; (bind ?*blocked-t-Whips* FALSE)
; (bind ?*blocked-Subsets* FALSE)

;;; Choose what's printed as the output.
;;; The default combination is what has been used in [PBCS].
;;; Changes below will print more or less details.
; (bind ?*print-init-details* TRUE)
; (bind ?*print-ECP-details* TRUE)
; (bind ?*print-actions* FALSE)
; (bind ?*print-levels* TRUE)
; (bind ?*print-solution* FALSE)

;;; The resolution state after BRT is printed by default.
;;; Un-comment this if you do not want to print it.
; (bind ?*print-RS-after-Singles* FALSE)

;;; The resolution state after Singles and whips[1] is printed by default.
;;; Un-comment this if you do not want to print it:
; (bind ?*print-RS-after-whips[1]* FALSE)

;;; The resolution state is printed by default at the end of resolution
;;; if the solution has not been found.
;;; Un-comment this if you do not want to print it:
; (bind ?*print-final-RS* FALSE)
```

5.4.2 Choose which resolution method you want to apply

There are four kinds of choices that should not be mixed, i.e. un-commented lines should appear in only one of the following four sub-sections:

1) Ordinary resolution rules, where several groups of options appear (my personal most usual choice is selected by default);
2) T&E-related configurations, where different typical options are proposed;
3) DFS (depth-first-search) options (with or without whips[1 or 2]);
4) Rules based on indirect binary contradictions (not in this public V2.1 version).

The four possibilities are described in sections 5.4.3 to 5.4.6 below. Remember that *selection is done by deleting the semi-colon at the start of a line*, as in the "my standard config" part in section 5.4.3 below. Remember also that, if you want to use rules in another section, you must first comment out these default choices.

5.4.3 Choose ordinary resolution rules

First, let me state the rule dependencies that will be automatically implemented at load time. Here, "A => B" means: "if A is loaded, then B will be loaded". For chain rules, these implications can be read length by length (e.g. if braids are loaded up to length 10, whips will be loaded up to length 10 – or more if they have been separately asked to be loaded with a longer maximum length). There is no implication other than those listed here; in particular, other special cases of the rules selected by the user (such as e.g. bivalue-chains, z-chains, t-whips, typed versions of the selected rules…) are not loaded unless explicitly asked to. In addition, for each specific application, you will be able to load only the rules meaningful for it (e.g. you will not be able to load g-whips in LatinRules or Subsets in MapRules).

- Subsets => Subsets[4] => Subsets[3] => Subsets[2] => Whips[1]
- FinnedFish => FinnedFish[4] => FinnedFish[3] => FinnedFish[2]
- FinnedFish[k] => Subset[k]

- z-Chains[k] => Bivalue-Chains[k]
- Typed-z-Chains[k] => Typed-Bivalue-Chains[k]

- G-Whips (or G2-Whips) => Whips
- Braids => Whips
- G-Braids => Braids and G-Whips

- Forcing-Whips => Whips
- Forcing-G-Whips => G-Whips and Forcing-Whips
- Forcing-Braids => Braids and Forcing-Whips
- Forcing-G-Braids => G-Braids and Forcing-G-Whips

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 1) ordinary resolution rules
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; My standard config and its usual variants
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; My most usual rules (config = W+S+Fin), with unrestricted lengths:
;;; Sudoku-specific:
 (bind ?*Subsets* TRUE)
 (bind ?*FinnedFish* TRUE)
;;; generic:
; (bind ?*Whips[1]* TRUE) ; allows to more easily activate only whips[1]
 (bind ?*Bivalue-Chains* TRUE)
 (bind ?*Whips* TRUE)

;;; Some additional rules I use frequently:
; (bind ?*z-Chains* TRUE)
; (bind ?*t-Whips* TRUE)
; (bind ?*G-Whips* TRUE)

;;; Some optional intermediary Typed Chains, allowing more varied resolution paths:
;;; (remember that whips[1] cannot be type-restricted)
; (bind ?*Typed-Bivalue-Chains* TRUE)
; (bind ?*Typed-z-Chains* TRUE)
; (bind ?*Typed-t-Whips* TRUE)
; (bind ?*Typed-Whips* TRUE)
;;; Choose stricter type restrictions in the above Typed Chains.
;;; The same type restrictions will apply to all the typed-chains.
;;; Type restrictions correspond to working in only some of the four 2D-spaces,
;;; i.e. using only part of the Extended Sudoku Board.
;;; BEWARE: type restrictions defined by global variable ?*allowed-csp-types*
;;; will apply only if ?*restrict-csp-types-in-typed-chains* is set to TRUE.
; (bind ?*restrict-csp-types-in-typed-chains* TRUE)
; (bind ?*allowed-csp-types* (create$ rc))

;;; Some additional rules I use occasionally:
; (bind ?*G2-Whips* TRUE)
```

```
; (bind ?*G-Bivalue-Chains* TRUE)
; (bind ?*Braids* TRUE)
; (bind ?*G-Braids* TRUE)

; (bind ?*Oddagons* TRUE)

;;; Some additional rules I almost never use:
; (bind ?*Forcing-Whips* TRUE)
; (bind ?*Forcing-G-Whips* TRUE)
; (bind ?*Forcing-Braids* TRUE)
; (bind ?*Forcing-G-Braids* TRUE)

;;; Setting ?*All-generic-chain-rules* to TRUE will activate all the generic chain
rules listed above,
;;; (which doesn't include the Subset rules),
;;; with the max-lengths as specified below (but automatically modified for
consistency).
;;; It is NOT RECOMMENDED to use this possibility, unless you know what you are doing
;;; Many complex rules are loaded and memory overflow problems may appear.
; (bind ?*All-generic-chain-rules* TRUE)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Sudoku-specific rules: uniqueness and "exotic" patterns
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; U-resolution rules for uniqueness.
;;; BEWARE: don't activate the following uniqueness rules,
;;; if you are not sure that the puzzle has a unique solution.
;;; The result would be undefined.
; (bind ?*Unique-Rectangles* TRUE)
; (bind ?*BUG* TRUE)

;;; Exotic patterns
;;; Belt (sk-loop) and J-Exocet rules fall under the category of what I called exotic
patterns,
;;; because they are very specialised and rarely present in a puzzle -
;;; a name that has immediately been adopted on the Sudoku forums

; (bind ?*Belt4* TRUE)
; (bind ?*Belt6* TRUE)

; (bind ?*J-Exocet* TRUE)
; (bind ?*J2-Exocet* TRUE)
; (bind ?*J3-Exocet* TRUE)
; (bind ?*J4-Exocet* TRUE)
; (bind ?*J5-Exocet* TRUE)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Change the default maximal lengths of the chain patterns
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Don't change these lengths unless you have some reason:

;;; The maximum length of all the generic chains can be lowered at once:
;  (defglobal ?*all-chains-max-length* = 36)

;;; Maximum lengths can also be lowered individually
; (bind ?*bivalue-chains-max-length* 20)
; (bind ?*z-chains-max-length* 20)
; (bind ?*t-whips-max-length* 36)
; (bind ?*whips-max-length* 36)
; (bind ?*g2whips-max-length* 36)
; (bind ?*g-bivalue-chains-max-length* 20)
; (bind ?*gwhips-max-length* 36)
; (bind ?*braids-max-length* 36)
; (bind ?*gbraids-max-length* 36)
; (bind ?*oddagons-max-length* 15)

; (bind ?*typed-bivalue-chains-max-length* 20)
; (bind ?*typed-z-chains-max-length* 20)
; (bind ?*typed-t-whips-max-length* 36)
; (bind ?*typed-whips-max-length* 36)

; (bind ?*forcing-whips-max-length* 36)
; (bind ?*forcing-gwhips-max-length* 36)
; (bind ?*forcing-braids-max-length* 36)
; (bind ?*forcing-gbraids-max-length* 36)
```

5.4.4 Decide to use some form of T&E for various purposes

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 2) Choose typical T&E config options, for various predefined purposes
;;;
;;; DO NOT FORGET TO DISABLE ALL THE RULES IN THE OTHER SECTIONS
;;; BEFORE ACTIVATING T&E
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Un-comment the proper line(s) below to change the level of details you want to be
printed:
; (bind ?*print-actions* FALSE)
; (bind ?*print-levels* TRUE)
; (bind ?*print-details* TRUE)
; (bind ?*print-solution* FALSE)
```

```
; (bind ?*print-hypothesis* FALSE)
; (bind ?*print-phase* TRUE)
; (bind ?*print-RS-after-Singles* FALSE)
; (bind ?*print-RS-after-whips[1]* FALSE)
; (bind ?*print-final-RS* FALSE)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2a) for checking membership in T&E(k) or gT&E(k), k = 1,2,3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Choose one of the following 3 depths of T&E:
;;; - depth 2 is enough for all the 9x9 Sudokus
;;; - but deeper T&E is often required for larger Sudokus or for Sukakus

; (bind ?*TE1* TRUE) ;;; for T&E at level 1
; (bind ?*TE2* TRUE) ;;; for T&E at level 2
; (bind ?*TE3* TRUE) ;;; for T&E at level 3

;;; In addition to the previous choice, you can give priority to bivalue candidates:
; (bind ?*special-TE* TRUE)

;;; For gT&E(k) instead of T&E(k), activate the next line:
; (bind ?*Whips[1]* TRUE)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2b) For computing the SpB classification
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; choose one of the following forms of T&E(1, Sp or SpFin)
; (bind ?*TE1* TRUE) ;;; for T&E at level 1
;;; For T&E at level 1, with priority for bivalue variables, add the following:
; (bind ?*special-TE* TRUE)

;;; Remember that whips[1] are always activated before Subsets,
;;; even if you don't activate them explicitly here.
;;; But you can choose to activate only them, to get gT&E (as in 2a)
; (bind ?*Whips[1]* TRUE)

;;; choose which Subsets[p] and FinnedFish[p] are activated:
; (bind ?*Subsets[2]* TRUE)
; (bind ?*Subsets[3]* TRUE)
; (bind ?*Subsets[4]* TRUE)
; (bind ?*Subsets* TRUE)

; (bind ?*FinnedFish[2]* TRUE)
; (bind ?*FinnedFish[3]* TRUE)
; (bind ?*FinnedFish[4]* TRUE)
; (bind ?*FinnedFish* TRUE)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2c) for computing the BpB classification
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; choose one of the following forms of T&E(1)
; (bind ?*TE1* TRUE) ;;; for T&E at level 1
;;; For T&E at level 1, with priority for bivalue variables, add the following:
; (bind ?*special-TE* TRUE)

;;; choose p (here p = 3):
; (bind ?*Whips* TRUE)
; (bind ?*Braids* TRUE)
; (bind ?*whips-max-length* 3)
; (bind ?*braids-max-length* 3)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2d) for looking for backdoors or anti-backdoors
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; choose one or several of backdoors, anti-backdoors or anti-backdoor pairs:
; (bind ?*Backdoors* TRUE)
; (bind ?*Anti-backdoors* TRUE)
; (bind ?*Anti-backdoor-pairs* TRUE)

;;; for W1-backdoors, W1-anti-backdoors or W1-anti-backdoor pairs, add the following:
; (bind ?*Whips[1]* TRUE)
;;; for S-backdoors, S-anti-backdoors or S-anti-backdoor pairs, add the following:
; (bind ?*Subsets* TRUE)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2e) for solving with Forcing-T&E
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; For Forcing T&E OR Forcing{3} T&E, activate only one of the following
;;; The possibility of activating both together is not yet available
; (bind ?*Forcing-TE* TRUE)
; (bind ?*Forcing{3}-TE* TRUE)

;;; for Forcing-T&E(W1) or Forcing{3}-T&E(W1), add:
; (bind ?*Whips[1]* TRUE)
;;; for Forcing-T&E(S) or Forcing{3}-T&E(S), add:
; (bind ?*Subsets* TRUE)
```

5.4.5 Decide to try DFS

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 3) Choose DFS (dept-first search) options
```

```
;;;
;;; DO NOT FORGET TO DISABLE ALL THE RULES IN THE OTHER SECTIONS
;;; BEFORE ACTIVATING DFS
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; DFS can be used to provide a relatively fast solution

;;; To block all output:
; (bind ?*print-actions* FALSE)
; (bind ?*print-levels* TRUE)
; (bind ?*print-details* TRUE)
; (bind ?*print-solution* FALSE)
; (bind ?*print-hypothesis* FALSE)
; (bind ?*print-solution* FALSE)
; (bind ?*print-phase* TRUE)
; (bind ?*print-RS-after-Singles* FALSE)
; (bind ?*print-RS-after-whips[1]* FALSE)
; (bind ?*print-final-RS* FALSE)

;;; To activate DFS:
; (bind ?*DFS* TRUE)
;;; To activate priority for bivalue candidates, activate this line, in addition to
the above line:
; (bind ?*special-DFS* TRUE)

;;; Activate short whips for combining whips[1] or whips[2] with DFS:
;;; this often gives a faster result (but not with larger whips)
; (bind ?*Whips* TRUE)
; (bind ?*whips-max-length* 1)
; (bind ?*whips-max-length* 2)
```

At the end of this sub-section, the following banner marks the end of what the user may change:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;                            end of allowed changes
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.5 The final part of the configuration file must not be changed

After the configuration has been defined by the user, the final part ensures the proper loading of the relevant rules. In Sudoku, it looks like this (the redefine-internal-factors part is Sudoku-specific)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; AUTOMATIC CONFIGURATION AND LOADING (DON'T CHANGE ANYTHING BELOW THIS LINE)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Because grid size may have been changed in this file,
;;; redefine the associated internal factors;
;;; this has to be done BEFORE loading
(redefine-internal-factors)

;;; now, load all
;;; Notice that the generic loader also loads the application-specific files
(if (and (or ?*G-Bivalue-Chains* ?*G-Whips* ?*G-Braids*) (> ?*segment-size* 4))
    then (printout t
       "BEWARE: g-labels, g-bivalue-chains, g-whips and g-braids are not managed" crlf
       "for segment size larger than 4, i.e. grid size larger than 16" crlf)
    else (batch ?*CSP-Rules-Generic-Loader*)
)
```

6. SudoRules

SudoRules is the pattern-based solver of Sudoku puzzles based on CSP-Rules. The pattern-based approach of a CSP was first described in [HLS] for Sudoku and then in [PBCS] for the general CSP. I have written several versions of SudoRules before I started a total re-writing of all its general rules into a generic form, thus making what is now the generic part of CSP-Rules. [As several users have asked about it: that's why SudoRules in CSP-Rules-V2.1 is not SudoRules-V2.1 but SudoRules V20.1.]

As a result of this historical development, together with a much broader and deeper study of the Sudoku puzzles than any other application, SudoRules is the most developed application of CSP-Rules. It has been used to solve and rate millions of puzzles. It has many more user functions than the other applications. One thing to be reminded is, SudoRules does not include the zillion Sudoku-specific rules one can find in Sudoku forums (most of the time in very imprecise and ambiguous forms) or in a typical Sudoku solver. (It does include the most familiar of them however, e.g. Subsets, Finned-Fish, sk-loops, J-Exocets.) Most of these rules are subsumed by the generic ones. My initial goals were, and remain, to prove the real applicability and to study the range of the generic resolution rules introduced in [PBCS]. (And, needless to repeat it, these goals included ease of programming for me; they did not include coding for the highest possible speed of resolution.)

Notice that none of the functions defined below takes any argument for grid size. ***The default puzzle size in SudoRules is set to 9***. But this size can be changed in the SudoRules configuration file. The main reasons for this choice are history and the fact that almost all the Sudokus one meets in nature are 9×9. There is also a technical reason: g-whips don't work for large grid sizes and the corresponding rules should not be loaded when one tries to solve such puzzles. The same remarks (except the technical one) apply to LatinRules.

The first six sections of this chapter are for basic level Sudoku solving and for some goodies, whereas the last ones are about advanced techniques, most of them for dealing with additional requirements on the resolution path.

In this chapter, "number" or "digit" means:

- a member of {1, 2, …, 9} if grid size is 9×9,

- a member of {1, 2, …, 9, A, B, C, D, E, F, G} if grid size is 16×16,
- a member of {1, 2, …, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P} if grid size is 25×25,
- a member of {1, 2, …, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0} if grid size is 36×36 – but this large size may be in practice beyond the upper limit for SudoRules, depending on how much RAM you have.

6.1 Solving a puzzle given in the standard "line" (or "string") format

"***solve***" is the first and simplest user function, in SudoRules as in all the CSP-Rules applications. In a first contact with SudoRules, this is indeed the only user function you need to know. The syntax is very simple:

```
(solve ?string)
```

where "?string" is a string of 81 characters (between double quotes) representing the puzzle in the classical "line format". The rc-cells in the grid are supposed to be numbered from left to right and from top to bottom. A digit at the nth place in the string represents a given for the nth cell; a dot (sometimes a 0 for 9×9 puzzles) represents the absence of a given (it is there to mark a place in the string). Same example, with dots and with 0s:

```
(solve
"4...3.......6..8..........1....5..9..8....6...7.2........1.27..5.3....4.9........")
(solve
"400030000000600800000000001000050090080000600070200000000102700503000040900000000")
```

For compatibility with other functions, "solve" is also named "***solve-sudoku-string***"

```
(solve-sudoku-string
"4...3.......6..8..........1....5..9..8....6...7.2........1.27..5.3....4.9........")
```

"***solve-knowing-solution***" is a variant of "solve" that can be used when the solution is already known:

```
(solve-knowing-solution ?puzzle-string ?sol-string)
```

where both "?puzzle-string" and "?sol-string" are strings of 81 characters representing the puzzle and the solution, as before. Example:

```
(solve-knowing-solution
"4...3.......6..8..........1....5..9..8....6...7.2........1.27..5.3....4.9........"
"468931527751624839392578461134756298289413675675289314846192753513867942927345186")
```

The presence of this function may be surprising at first: why should one want to solve a puzzle if he already knows the solution? The reason is, the solution may be known (e.g. after applying a depth-first-search program) without a readable resolution path and/or without the rating being known.

In this function, the solution is only used to avoid building chains (whips,…) that are known in advance to be unable to lead to any elimination: candidates known to be in the solution are not tried as targets for building such chains on them.

The resolution path and the final result are exactly the same as those of "solve-sudoku-string", except that useless calculations are avoided. This may be occasionally useful for the hardest puzzles, to save computation time and/or memory.

"***solve-sukaku-string***" is a variant of "solve-sudoku-string" that can be used when a resolution state (a state including both values and candidates) is given instead of an original puzzle (with values only). The name "Sukaku" has been introduced recently in the Sudoku world for naming such situations. Syntax:

```
(solve-sukaku-string ?string)
```

where "?string" is a sequence of 729 digits or dots, such that the nth group of nine characters represents the candidates present in the nth rc-cell. There is much redundancy in this representation of a resolution state, because the presence of a digit in an rc-cell is doubly represented: by its value and by its position in the sequence. A sequence of 0s and 1s would be enough but it would also be less readable. Anyway, that is the form in which it came to existence and I kept it as is.

Example of an extremely difficult Sukaku puzzle found by Tarek, a famous puzzle creator (SudoRules will not solve it without DFS):

```
(solve-sukaku-string
"..3.5678.1.3456789123456789.23456789...4.678912.4567891234567891.34567891..45.789123.
567891.3...7..12345.7..123....891234.678912.4...8.123..6.891........12345.789123456789
1234567..12345678.123456.891234567891234567 8.123.56.89123456..9123456789.23456789...4.
678912.4567891.3.567891.34567891..45.789.234567891..4.678912.456789.23..6.891234.67891
23456789123.567891.3...7.912345.789123.567891234.678912.4..78.123456.891234.6...123456
789123.56..91234567.91234..7..123456.891234.6...12345678.1234567891...5678912.45678912
345678912345678.12.45678..23.567891.3456789123456789123.567891.....7.912.45.789123..6.
891234.678912.4..78..23.567891.3...7.912345.789123.56..91234567.91234567891234567.89123
4.6...12345678.1234567891234567.912345.7..")
```

As this format is totally illegible, SudoRules has a function for initializing and printing it in a more readable form (which will also be output at the start of function "solve-kukaku-string"):

```
(init-sukaku-string ?sukaku-string)
```

will give:

```
35678     13456789 123456789 23456789  46789     12456789 23456789 3456789  45789
2356789   37       23457     2389      2346789   248      23689    1        2345789
123456789 1234567  12345678  12345689  123456789 12345678 235689   234569   23456789
23456789  46789    12456789  1356789   13456789  145789   23456789 46789    12456789
23689     12346789 123456789 12356789  1379      12345789 12356789 2346789  12478
12345689  12346    123456789 123569    12345679  12347    12345689 2346     12345678
123456789 156789   12456789  123456789 12345678  1245678  2356789  3456789  123456789
12356789  179      1245789   123689    12346789  12478    2356789   379     12345789
123569    12345679 123456789 12345689  12346     12345678 123456789 2345679 123457
```

Notice that a Sukaku not originating from a real Sudoku puzzle (contrary to the above example) can be much harder than a Sudoku puzzle: all the known Sudoku puzzles are at worst in T&E(2), but some Sukakus are known to be in T&E(4).

"solve-sukaku-string" can typically be used when a resolution state of a Sudoku puzzle is obtained after the application of some basic rules and one asks: "what comes next?".

6.2 Solving Sudoku puzzles given in other formats

6.2.1 The list format for Sudoku

Often, a puzzle is given in a more user-friendly format than the line format, such as:

```
. . 1  2 . .  . . 3
. . .  . . 4  2 . .
5 . .  . . .  . 6 .
7 . .  6 . .  . 1 .
. . .  . 3 .  . . .
. 8 .  . . 9  . . 4
. 3 .  . . .  . . 6
. . 7  1 . .  . 2 .
4 . .  . . 7  5 . .
```

"***solve-sudoku-list***" allows to solve puzzles given in this format. This function has a unique argument $?rest; as explained in the introduction, all the arguments given to it are interpreted as a single list. In this list format, make sure to keep at least one space between any two symbols (digits or dots): in a list, "7 5" is two elements, but "75" is only one (and a meaningless one in Sudoku). Example:

```
(solve-sudoku-list
   . . 1  2 . .  . . 3
   . . .  . . 4  2 . .
   5 . .  . . .  . 6 .
   7 . .  6 . .  . 1 .
   . . .  . 3 .  . . .
   . 8 .  . . 9  . . 4
   . 3 .  . . .  . . 6
   . . 7  1 . .  . 2 .
   4 . .  . . 7  5 . .
)
```

6.2.2 The list format for Sukaku

Most of the time, a Sukaku puzzle is given in a much more user-friendly format than the line format. Function ***"solve-sukaku-list"*** allows to deal with it; it has a single argument $?list representing the list of possible candidate sets for each cell. For each cell, all the possible candidate-values from {1, 2, …, Z, 0} are glued into a single symbol. The syntax should be clear from the following example:

```
(solve-sukaku-list
      1     9     6      2      4     7      3     5     8
      8     2     7      3      5     1      4     6     9
      3     4     5      68     9     68     27    27    1
      7     8     24     1      6     49     29    3     5
      5     6     1      89     2     3      89    4     7
      9     3     24     578    78    458    1     28    6
      6     7     89     589    3     2      58    1     4
      4     5     3      6789   1     689    678   789   2
      2     1     89     4      78    56     56    789   3
)
```

[For the technically-minded readers, notice that an entry like 6789 in the above list of arguments is technically (i.e. in CLIPS) neither a string nor a number (in 16×16 or larger puzzles, it could contain letters, e.g. 6789BD); it is a symbol.]

6.2.3 The grid format for Sudoku

A still more user-friendly format (for the same puzzle as in §6.2.1) found on the Sudoku forums would be as shown below. This is the original reason why I introduced the list format. Unfortunately, there can be no way of giving the puzzle in this form directly to SudoRules, because "|" is a reserved symbol in CLIPS (it means logical "or" in some situations); in particular, it cannot appear as an argument of any function. However, the user can easily transform this format into the previous one, either manually or with any text editor.

```
+-------+-------+-------+
| . . 1 | 2 . . | . . 3 |
| . . . | . . 4 | 2 . . |
| 5 . . | . . . | . 6 . |
+-------+-------+-------+
| 7 . . | 6 . . | . 1 . |
| . . . | . 3 . | . . . |
| . 8 . | . . 9 | . . 4 |
+-------+-------+-------+
| . 3 . | . . . | . . 6 |
| . . 7 | 1 . . | . 2 . |
| 4 . . | . . 7 | 5 . . |
+-------+-------+-------+
```

Alternatively, function "***solve-sudoku-grid***" will allow something very close (where symbol "!" can also be ":" and "+" can also be "*"). Again, the syntax should be clear from an example:

```
(solve-sudoku-grid
   +-------+-------+-------+
   ! . . 1 ! 2 . . ! . . 3 !
   ! . . . ! . . 4 ! 2 . . !
   ! 5 . . ! . . . ! . 6 . !
   +-------+-------+-------+
   ! 7 . . ! 6 . . ! . 1 . !
   ! . . . ! . 3 . ! . . . !
   ! . 8 . ! . . 9 ! . . 4 !
   +-------+-------+-------+
   ! . 3 . ! . . . ! . . 6 !
   ! . . 7 ! 1 . . ! . 2 . !
   ! 4 . . ! . . 7 ! 5 . . !
   +-------+-------+-------+
)
```

Notice that "***solve-sudoku-grid***" is more general than solve-sudoku-list and the latter could be forgotten.

6.2.4 The grid format for Sukaku

For Sukakus, function "***solve-sukaku-grid***" allows something quite similar to solve-sudoku-grid. The Sukaku in section 6.2.2 can be solved as follows (where symbol "!" can also be ":" and "+" can also be "*"). Notice that the number of spaces between different cells can be arbitrary; here, it is fitted for visual purposes – which is all there is to the solve-sudoku-grid and solve-sukaku-grid functions.

Notice also that "***solve-sukaku-grid***" is more general than solve-sukaku-list and the latter could be forgotten.

```
(solve-sukaku-grid
                +----------------+----------------+----------------+
                ! 1    9    6    ! 2    4    7    ! 3    5    8    !
                ! 8    2    7    ! 3    5    1    ! 4    6    9    !
                ! 3    4    5    ! 68   9    68   ! 27   27   1    !
                +----------------+----------------+----------------+
                ! 7    8    24   ! 1    6    49   ! 29   3    5    !
                ! 5    6    1    ! 89   2    3    ! 89   4    7    !
                ! 9    3    24   ! 578  78   458  ! 1    28   6    !
                +----------------+----------------+----------------+
                ! 6    7    89   ! 589  3    2    ! 58   1    4    !
                ! 4    5    3    ! 6789 1    689  ! 678  789  2    !
                ! 2    1    89   ! 4    78   56   ! 56   789  3    !
                +----------------+----------------+----------------+
)
```

6.2.5 The tatham format

A slightly different string format is used by Tatham, a famous creator of many types of puzzles (see https://www.chiark.greenend.org.uk/~sgtatham/puzzles/). His puzzles are generally relatively easy but they may be interesting for beginners and

beyond. The “Tatham format” for Sudoku is more compact for printing than the standard line format and as such, it is interesting, though it’s harder to read: every sequence of zeros or dots in the line format is replaced by a lower case letter: “a” means one dot, “b” means two dots, … It is meaningful only for 9×9 puzzles.

“***solve-tatham-string***” is a variant of “solve-sudoku-string” that can deal with the Tatham string format:

```
(solve-tatham-string ?tatham-str)
```

As you may have guessed, “?tatham-str” is the Tatham representation of a puzzle as a string, i.e. between double quotes. Example:

```
(solve-tatham-string
"2a9a15a4c58d9a3i1e4c127c7e9i3a4d17c2a54a6a8")
```

The underscores (“_”) that, for some unknown reason, appear in the format when the puzzle is copied from the Tatham website don’t add anything to the string and could as well be deleted. The following is equivalent to the above:

```
(solve-tatham-string
"2a9a1_5a4c5_8d9a3i1e4c1_2_7c7e9i3a4d1_7c2a5_4a6a8")
```

Both forms are equivalent to:

```
(solve-sudoku-string
  "2.9.15.4...58....9.3.........1.....4...127...7.....9.........3.4....17...2.54.6.8")
```

If you want to see how a puzzle given in Tatham format looks like in the familiar line format, the function “tatham-to-sudoku-string” allows this:

```
(tatham-to-sudoku-string "2a9a1_5a4c5_8d9a3i1e4c1_2_7c7e9i3a4d1_7c2a5_4a6a8")
```

gives:

```
"2.9.15.4...58....9.3.........1.....4...127...7.....9.........3.4....17...2.54.6.8"
```

Indeed, (solve-tatham-string ?tatham-str) is defined as:
(solve-sudoku-string (tatham-to-sudoku-string ?tatham-str))

6.2.6 The sdk format for files

It may sometimes be useful to have a puzzle displayed on 9 lines at the top of a file and then to write information about it and its resolution path(s) in the same file. The nine-line format is also easier to visualise than the one-line format. Function “***solve-sdk-grid***” reads the first nine symbols of the first nine lines of a file and solves the corresponding puzzle (its argument ?file-name is the path to the file; see section 6.3 for an example of path):

```
(solve-sdk-grid ?file-name)
```

For the puzzle in the previous example, the nine-line format at the top of the file is similar to the list format, but with all the spaces in each line deleted (including any spaces or tabs at the start of the lines); it’s also similar to the line format, with line feeds / carriage returns added after each group of nine symbols. It does not accept anything before the nine lines or before the content of each of them.

```
..12....3
.....42..
5......6.
7..6...1.
....3....
.8...9..4
.3......6
..71...2.
4....75..
```

6.3 Solving collections of puzzles

When studying general properties of Sudoku puzzles or of resolutions rules, one often has to deal with a long collection of puzzles written in a text file, with one puzzle in the line format per line. The functions in this section allow to deal with such cases. The first 81 characters of each line must be digits or dots; whatever comes after them is not taken into account. This allows to keep in the same file additional information about each puzzle, such as their "name" if they have one, their author, the date they were discovered, their SER (Sudoku Explainer Rating, …). Fundamentally, each of the following functions repeatedly calls the "solve" function on puzzles in the list, as many times as specified by the argument(s). It also keeps track of the total solving time and it prints it at the end.

"***solve-nth-grid-from-text-file***" allows to pick one puzzle from the list and to solve it:

```
(solve-nth-grid-from-text-file ?file-name ?nb)
```

where "?file-name" is the path to the file and "?nb" is the place of the puzzle of interest in the list. ?nb may not be larger than the number of lines in the file. Example:

```
(solve-nth-grid-from-text-file "/Users/My-User-name/Documents/MySudokuPuzzles.txt" 56)
```

It may also be useful to check what there's on a particular line in a file. This can be done with function display-nth-line-from-text-file:

```
(display-nth-line-from-text-file ?file-name ?nb)
```

The name of function "***solve-n-grids-after-first-p-from-text-file***" is sufficiently explicit to need no explanation (however, beware the order of the arguments):

```
(solve-n-grids-after-first-p-from-text-file ?file-name ?p ?n)
```

where "?file-name" is as before, ?p is the number of lines that must be skipped and "?n" is the number of puzzles that must be solved. Example:

```
(solve-n-grids-after-first-p-from-text-file
"/Users/User-name/Documents/Sudoku/MySudokuPuzzles.txt" 12 56)
```

If you want to solve all the puzzles in the file, set ?p to 0 and ?n to the length of the file.

It is very convenient to be able to keep track of global information about a collection of puzzles, as an additional first line in the same file. For this reason, SudoRules includes variants of the previous three functions that allow to work as if the first line of a file did not exist; they take the same arguments are before:

```
(display-nth-effective-line-from-titled-text-file ?file-name ?nb)
(solve-nth-grid-from-titled-text-file ?file-name ?nb)
(solve-n-grids-after-first-p-from-titled-text-file ?file-name ?p ?n)
```

6.4 For the readers of [HLS]

All the chain names used in [HLS] have been changed in [PBCS] and in this Manual. The naming convention in [HLS] was too dependent on the Sudoku grid structure and could not be made generic. The new naming convention is obviously simpler, as shown by the following table (where lengths of chains are arbitrary).

Name in [HLS]	**Name in [PBCS]**	Remarks
xy5-chain	**biv-chain-rc[5]**	
hxy-uu5-chain	**biv-chain-uu[5]**	for uu = rn, cn or bn
nrc5-chain	**biv-chain[5]**	
xyz6-chain	**z-chain-rc[6]**	
hxyz-uu7-chain	**z-chain-uu[6]**	for uu = rn, cn or bn
xyt-rc6-chain	**t-whip-rc[6]**	
hxyt-uu6-chain	**t-whip-uu[6]**	for uu = rn, cn or bn
nrct6-chain	**t-whip[6]**	t-whips are slightly more general than nrct-chains
xyzt-rc8-chain	**whip-rc[8]**	whips-rc are slightly more general than xyzt-chains
hxyzt-uu8-chain	**whip-uu[8]**	for uu = rn, cn or bn whips-uu are slightly more general than xyzt-uu-chains
nrczt8-chain	**whip[8]**	whips are slightly more general than nrczt-chains

6.5 How different selections of rules change the resolution path

I now use a moderately difficult puzzle to show how the various selections of rules made possible in the configuration file can give very different resolution paths.

The puzzle (cbg#109) is:

```
+-------+-------+-------+
! . . . ! 4 . . ! . 8 9 !
! . 5 . ! . . . ! 1 . . !
! . . 9 ! . 3 . ! . . 6 !
+-------+-------+-------+
! . 7 1 ! . 6 . ! 9 . 3 !
! . . . ! . . 2 ! 4 . . !
! 9 . . ! . . 5 ! . . 1 !
+-------+-------+-------+
! 3 . . ! . . . ! 6 . . !
! . 9 . ! . 4 . ! . 1 . !
! 8 1 5 ! . . 3 ! . . . !
+-------+-------+-------+
```

```
...4...89.5....1....9.3...6.71.6.9.3.....24..9....5..13.....6...9..4..1.815..3...
SER = 8.3
```

All the forthcoming resolution paths have the same straightforward start, ending in the same resolution state RS1 (represented here as a Sukaku list):

```
singles ==> r4c4=8, r4c6=4, r6c5=7, r6c4=3, r9c4=6, r8c7=3, r2c8=3, r6c7=8
whip[1]: r9n7{c9 .} ==> r8c9≠7, r7c8≠7, r7c9≠7
whip[1]: r9n4{c9 .} ==> r7c9≠4, r7c8≠4
whip[1]: c7n5{r3 .} ==> r3c8≠5
whip[1]: c1n4{r3 .} ==> r3c2≠4, r2c3≠4
whip[1]: b6n2{r6c8 .} ==> r9c8≠2, r3c8≠2, r7c8≠2
hidden-pairs-in-a-row: r5{n3 n8}{c2 c3} ==> r5c3≠6, r5c2≠6
;;; Resolution state RS1:
+--------------------+--------------------+--------------------+
! 1267  236   2367  ! 4     125   167   ! 257   8     9     !
! 2467  5     2678  ! 279   289   6789  ! 1     3     247   !
! 1247  28    9     ! 1257  3     178   ! 257   47    6     !
+--------------------+--------------------+--------------------+
! 25    7     1     ! 8     6     4     ! 9     25    3     !
! 56    38    38    ! 19    19    2     ! 4     567   57    !
! 9     246   246   ! 3     7     5     ! 8     26    1     !
+--------------------+--------------------+--------------------+
! 3     24    247   ! 12579 12589 1789  ! 6     59    258   !
! 267   9     267   ! 257   4     78    ! 3     1     258   !
! 8     1     5     ! 6     29    3     ! 27    479   247   !
+--------------------+--------------------+--------------------+
```

This part of the resolution paths will not be repeated; instead, I'll write only the part after resolution state RS1. "stte" means "singles to the end".

All the set of rules I'll consider will have Subsets and Finned Fish, but this is not really relevant. All the presentations will follow the same schema: first, the selected set of options from the SudoRules configuration file (any option not listed is supposed to be inactive) and then the resolution path. Notice that all the patterns that can be "blocked" are, i.e. they allow several eliminations at a time (which is now the default option).

6.5.1 Using whips

Let us first use the following configuration with Whips:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Bivalue-Chains* TRUE)
(bind ?*Whips* TRUE)
```

We get a resolution path in W5:

```
;;; Resolution state RS1
biv-chain[3]: r3c2{n2 n8} - r5c2{n8 n3} - b1n3{r1c2 r1c3} ==> r1c3≠2
biv-chain[3]: r5n7{c9 c8} - r3c8{n7 n4} - b9n4{r9c8 r9c9} ==> r9c9≠7
biv-chain[4]: r5c5{n1 n9} - r9n9{c5 c8} - r7c8{n9 n5} - c5n5{r7 r1} ==> r1c5≠1
whip[4]: c6n9{r2 r7} - r7c8{n9 n5} - c9n5{r8 r5} - c9n7{r5 .} ==> r2c6≠7
biv-chain[5]: r5c1{n5 n6} - c2n6{r6 r1} - b2n6{r1c6 r2c6} - c6n9{r2 r7} - r7c8{n9 n5}
==> r5c8≠5
biv-chain[4]: r2n4{c1 c9} - c9n7{r2 r5} - r5n5{c9 c1} - r4c1{n5 n2} ==> r2c1≠2
biv-chain[5]: c6n6{r1 r2} - c6n9{r2 r7} - r7c8{n9 n5} - b6n5{r4c8 r5c9} - r5c1{n5 n6}
==> r1c1≠6
whip[5]: c1n7{r3 r8} - r8c6{n7 n8} - c5n8{r7 r2} - c3n8{r2 r5} - c3n3{r5 .} ==> r1c3≠7
whip[5]: r1n6{c3 c6} - r1n1{c6 c1} - r1n7{c1 c7} - r3c8{n7 n4} - c1n4{r3 .} ==> r2c1≠6
whip[4]: r9c5{n2 n9} - c6n9{r7 r2} - r2n8{c6 c3} - r2n6{c3 .} ==> r2c5≠2
whip[4]: c3n7{r8 r2} - c9n7{r2 r5} - r5n5{c9 c1} - c1n6{r5 .} ==> r8c1≠7
whip[1]: b7n7{r8c3 .} ==> r2c3≠7
naked-triplets-in-a-column: c1{r4 r5 r8}{n2 n5 n6} ==> r3c1≠2, r1c1≠2
biv-chain[3]: c1n2{r4 r8} - r7c2{n2 n4} - b4n4{r6c2 r6c3} ==> r6c3≠2
whip[4]: c9n8{r8 r7} - c9n5{r7 r5} - r4c8{n5 n2} - c1n2{r4 .} ==> r8c9≠2
biv-chain[5]: c6n6{r1 r2} - c6n9{r2 r7} - r7c8{n9 n5} - r8c9{n5 n8} - r8c6{n8 n7} ==>
r1c6≠7
biv-chain[3]: r2c1{n4 n7} - r1n7{c1 c7} - r3c8{n7 n4} ==> r3c1≠4
singles ==> r2c1=4, r3c8=4, r9c9=4
finned-x-wing-in-columns: n2{c9 c3}{r2 r7} ==> r7c2≠2
singles ==> r7c2=4, r6c3=4
biv-chain[3]: r9n2{c7 c5} - r1c5{n2 n5} - b3n5{r1c7 r3c7} ==> r3c7≠2
biv-chain[3]: r2n7{c4 c9} - r5c9{n7 n5} - r8n5{c9 c4} ==> r8c4≠7
biv-chain[3]: r2n7{c4 c9} - c9n2{r2 r7} - r7c3{n2 n7} ==> r7c4≠7
whip[1]: b8n7{r8c6 .} ==> r3c6≠7
hidden-triplets-in-a-block: b2{r2c4 r1c5 r3c4}{n2 n5 n7} ==> r3c4≠1, r2c4≠9
whip[1]: b2n1{r3c6 .} ==> r7c6≠1
naked-pairs-in-a-row: r2{c4 c9}{n2 n7} ==> r2c3≠2
whip[1]: c3n2{r8 .} ==> r8c1≠2
singles to the end
```

6.5.2 Using t-whips

Let us now replace whips by the *a priori* less powerful t-whips:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Bivalue-Chains* TRUE)
(bind ?*t-Whips* TRUE)
```

We get a resolution path in tW5; this is a (probably rare) case where using t-whips instead of whips does not require longer chains:

```
;;; Resolution state RS1
biv-chain[3]: r3c2{n2 n8} - r5c2{n8 n3} - b1n3{r1c2 r1c3} ==> r1c3≠2
biv-chain[3]: r5n7{c9 c8} - r3c8{n7 n4} - b9n4{r9c8 r9c9} ==> r9c9≠7
biv-chain[4]: r5c5{n1 n9} - r9n9{c5 c8} - r7c8{n9 n5} - c5n5{r7 r1} ==> r1c5≠1
t-whip[4]: c6n9{r2 r7} - r7c8{n9 n5} - c9n5{r8 r5} - c9n7{r5 .} ==> r2c6≠7
biv-chain[5]: r5c1{n5 n6} - c2n6{r6 r1} - b2n6{r1c6 r2c6} - c6n9{r2 r7} - r7c8{n9 n5}
==> r5c8≠5
biv-chain[4]: r2n4{c1 c9} - c9n7{r2 r5} - r5n5{c9 c1} - r4c1{n5 n2} ==> r2c1≠2
biv-chain[5]: c6n6{r1 r2} - c6n9{r2 r7} - r7c8{n9 n5} - b6n5{r4c8 r5c9} - r5c1{n5 n6}
==> r1c1≠6
t-whip[5]: c3n3{r1 r5} - r5c2{n3 n8} - r3n8{c2 c6} - r8c6{n8 n7} - r7n7{c6 .} ==>
r1c3≠7
t-whip[5]: r9c5{n2 n9} - c6n9{r7 r2} - c6n6{r2 r1} - r1c3{n6 n3} - r1c2{n3 .} ==>
r1c5≠2
singles ==> r1c5=5, r3c7=5
biv-chain[4]: r1c7{n2 n7} - c9n7{r2 r5} - r5n5{c9 c1} - r4c1{n5 n2} ==> r1c1≠2
t-whip[4]: c5n2{r9 r2} - c5n8{r2 r7} - r8n8{c6 c9} - r8n5{c9 .} ==> r8c4≠2
t-whip[5]: r1n6{c3 c6} - r1n1{c6 c1} - r1n7{c1 c7} - b3n2{r1c7 r2c9} - r2n4{c9 .} ==>
r2c1≠6
t-whip[4]: r9c5{n2 n9} - c6n9{r7 r2} - r2n6{c6 c3} - r2n8{c3 .} ==> r2c5≠2
whip[1]: c5n2{r9 .} ==> r7c4≠2
t-whip[4]: c1n7{r3 r8} - c1n6{r8 r5} - r5c8{n6 n7} - c9n7{r5 .} ==> r2c3≠7
whip[1]: c3n7{r8 .} ==> r8c1≠7
naked-triplets-in-a-column: c1{r4 r5 r8}{n2 n5 n6} ==> r3c1≠2
biv-chain[3]: c1n2{r4 r8} - r7c2{n2 n4} - b4n4{r6c2 r6c3} ==> r6c3≠2
biv-chain[5]: r6n2{c2 c8} - r4c8{n2 n5} - r7c8{n5 n9} - r9n9{c8 c5} - c5n2{r9 r7} ==>
r7c2≠2
singles ==> r7c2=4
hidden-single-in-a-block ==> r6c3=4
biv-chain[5]: c1n2{r8 r4} - r4c8{n2 n5} - r7c8{n5 n9} - r9n9{c8 c5} - c5n2{r9 r7} ==>
r7c3≠2
naked-single ==> r7c3=7
whip[1]: b7n2{r8c3 .} ==> r8c9≠2
t-whip[4]: c6n9{r2 r7} - r7c8{n9 n5} - r7c4{n5 n1} - r5c4{n1 .} ==> r2c4≠9
naked-triplets-in-a-row: r2{c1 c4 c9}{n4 n7 n2} ==> r2c3≠2
singles to the end
```

6.5.3 Using typed-whips

Let us now replace the originally chosen whips by typed-whips and (of course) also bivalue-chains by typed-bivalue-chains:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Typed-Bivalue-Chains* TRUE)
(bind ?*Typed-Whips* TRUE)
```

We get a resolution path in TyW9; i.e. this time, we need longer chains. This is a general property of solving in the 2D spaces. In [PBCS], I mentioned that 97% of the puzzles can be solved with 2D-chains instead of the fully super-symmetric whips, but that generally implies having longer chains.

```
;;; Resolution state RS1
whip-cn[5]: c3n3{r1 r5} - c2n3{r5 r1} - c2n6{r1 r6} - c2n4{r6 r7} - c2n2{r7 .} ==>
r1c3≠2
whip-rc[6]: r9c7{n7 n2} - r9c5{n2 n9} - r9c8{n9 n4} - r3c8{n4 n7} - r3c7{n7 n5} -
r1c7{n5 .} ==> r9c9≠7
whip-cn[5]: c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c9n7{r5 .} ==>
r2c6≠7
whip-cn[7]: c2n6{r6 r1} - c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} -
c8n7{r3 r5} - c8n6{r5 .} ==> r6c3≠6
whip-cn[7]: c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c8n6{r5 r6} - c2n6{r6 r1} -
c6n6{r1 r2} - c6n9{r2 .} ==> r7c5≠9
whip-cn[7]: c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c8n6{r5 r6} - c2n6{r6 r1} -
c6n6{r1 r2} - c6n9{r2 .} ==> r7c4≠9
whip-bn[4]: b5n1{r5c5 r5c4} - b5n9{r5c4 r5c5} - b8n9{r9c5 r7c6} - b8n1{r7c6 .} ==>
r1c5≠1
whip-cn[7]: c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} -
c8n6{r5 r6} - c2n6{r6 .} ==> r1c3≠6
whip-cn[7]: c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} -
c8n6{r5 r6} - c2n6{r6 .} ==> r1c1≠6
whip-cn[7]: c8n6{r5 r6} - c2n6{r6 r1} - c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} -
c8n7{r9 r3} - c8n4{r3 .} ==> r5c8≠5
whip-rc[7]: r4c1{n2 n5} - r5c1{n5 n6} - r8c1{n6 n7} - r1c1{n7 n1} - r3c1{n1 n4} -
r3c8{n4 n7} - r5c8{n7 .} ==> r2c1≠2
whip-cn[7]: c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c8n6{r5 r6} -
c2n6{r6 r1} - c6n6{r1 .} ==> r2c6≠8
whip-rn[8]: r3n5{c7 c4} - r8n5{c4 c9} - r5n5{c9 c1} - r4n5{c1 c8} - r4n2{c8 c1} -
r3n2{c1 c2} - r3n8{c2 c6} - r8n8{c6 .} ==> r3c7≠7
whip-rc[9]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r2c3{n6 n7} - r1c1{n7 n1} -
r1c6{n1 n7} - r3c6{n7 n1} - r3c4{n1 n5} - r3c7{n5 .} ==> r3c1≠2
whip-cn[8]: c9n8{r8 r7} - c9n5{r7 r5} - c1n5{r5 r4} - c8n5{r4 r7} - c5n5{r7 r1} -
c7n5{r1 r3} - c7n2{r3 r1} - c1n2{r1 .} ==> r8c9≠2
whip-rc[5]: r8c6{n7 n8} - r3c6{n8 n1} - r7c6{n1 n9} - r7c8{n9 n5} - r8c9{n5 .} ==>
r1c6≠7
whip-rn[4]: r2n4{c1 c9} - r9n4{c9 c8} - r9n7{c8 c7} - r1n7{c7 .} ==> r2c1≠7
whip-rn[6]: r7n7{c6 c3} - r2n7{c3 c9} - r5n7{c9 c8} - r5n6{c8 c1} - r5n5{c1 c9} -
r8n5{c9 .} ==> r8c4≠7
```

```
whip-rc[6]: r9c5{n9 n2} - r8c4{n2 n5} - r8c9{n5 n8} - r8c6{n8 n7} - r7c4{n7 n1} -
r5c4{n1 .} ==> r5c5≠9
singles ==> r5c5=1, r5c4=9
biv-chain-rc[4]: r2c4{n7 n2} - r8c4{n2 n5} - r8c9{n5 n8} - r8c6{n8 n7} ==> r3c6≠7,
r7c4≠7
whip-rc[5]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r1c6{n6 n1} - r3c6{n1 .} ==>
r6c2≠2
biv-chain-rc[4]: r4c1{n2 n5} - r5c1{n5 n6} - r6c2{n6 n4} - r7c2{n4 n2} ==> r8c1≠2
biv-chain-rc[4]: r8c9{n5 n8} - r8c6{n8 n7} - r8c1{n7 n6} - r5c1{n6 n5} ==> r5c9≠5
singles ==> r5c9=7, r5c8=6, r5c1=5, r4c1=2, r4c8=5, r7c8=9, r6c3=4, r6c2=6, r6c8=2,
r1c6=6, r2c6=9, r9c5=9, r1c1=1, r7c2=4
whip[1]: b7n2{r8c3 .} ==> r2c3≠2
whip[1]: r9n2{c9 .} ==> r7c9≠2
naked-pairs-in-a-row: r3{c1 c8}{n4 n7} ==> r3c4≠7
hidden-single-in-a-block ==> r2c4=7
whip-rc[3]: r8c4{n5 n2} - r7c5{n2 n8} - r7c9{n8 .} ==> r7c4≠5
biv-chain-rc[4]: r3c2{n2 n8} - r2c3{n8 n6} - r2c1{n6 n4} - r2c9{n4 n2} ==> r3c7≠2
singles to the end
```

6.5.4 Using typed-t-whips

Let us restrict still more the patterns we allow, replacing the initial whips by typed-t-whips and (of course) also bivalue-chains by typed-bivalue-chains:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Typed-Bivalue-Chains* TRUE)
(bind ?*Typed-t-Whips* TRUE)
```

We get a resolution path in TytW9; this might have been conjectured in view of the W and tW ratings being identical, but the equality of the TyW and TytW ratings does not follow *a priori* from the equality of the W and tW ratings:

```
;;; Resolution state RS1
t-whip-cn[5]: c3n3{r1 r5} - c2n3{r5 r1} - c2n6{r1 r6} - c2n4{r6 r7} - c2n2{r7 .} ==>
r1c3≠2
t-whip-cn[7]: c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c8n6{r5 r6} -
c2n6{r6 r1} - c6n6{r1 .} ==> r2c6≠7, r2c6≠8
t-whip-cn[7]: c8n6{r5 r6} - c2n6{r6 r1} - c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} -
c8n4{r9 r3} - c8n7{r3 .} ==> r5c8≠5
t-whip-rc[7]: r5c9{n7 n5} - r5c1{n5 n6} - r5c8{n6 n7} - r3c8{n7 n4} - r9c8{n4 n9} -
r9c5{n9 n2} - r9c7{n2 .} ==> r9c9≠7
t-whip-cn[7]: c2n6{r6 r1} - c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} -
c8n7{r3 r5} - c8n6{r5 .} ==> r6c3≠6
t-whip-cn[7]: c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} - c8n6{r5 r6} - c2n6{r6 r1} -
c6n6{r1 r2} - c6n9{r2 .} ==> r7c4≠9, r7c5≠9
t-whip-cn[4]: c6n1{r3 r7} - c6n9{r7 r2} - c4n9{r2 r5} - c4n1{r5 .} ==> r1c5≠1
t-whip-cn[7]: c6n6{r1 r2} - c6n9{r2 r7} - c8n9{r7 r9} - c8n4{r9 r3} - c8n7{r3 r5} -
c8n6{r5 r6} - c2n6{r6 .} ==> r1c1≠6, r1c3≠6
t-whip-bn[9]: b1n4{r2c1 r3c1} - b3n4{r3c8 r2c9} - b9n4{r9c9 r9c8} - b9n7{r9c8 r9c7} -
b3n7{r1c7 r3c8} - b6n7{r5c8 r5c9} - b6n5{r5c9 r4c8} - b6n2{r4c8 r6c8} - b4n2{r6c3 .}
==> r2c1≠2
```

```
t-whip-rc[9]: r3c2{n8 n2} - r7c2{n2 n4} - r6c2{n4 n6} - r1c2{n6 n3} - r1c3{n3 n7} -
r7c3{n7 n2} - r8c3{n2 n6} - r8c1{n6 n7} - r8c6{n7 .} ==> r3c6≠8
singles ==> r2c5=8, r3c2=8, r5c2=3, r5c3=8, r1c3=3
t-whip-rn[4]: r9n2{c9 c5} - r9n9{c5 c8} - r7n9{c8 c6} - r7n8{c6 .} ==> r7c9≠2
t-whip-cn[5]: c5n2{r9 r1} - c5n5{r1 r7} - c8n5{r7 r4} - c8n2{r4 r6} - c2n2{r6 .} ==>
r7c4≠2
t-whip-rn[6]: r1n6{c2 c6} - r1n1{c6 c1} - r1n7{c1 c7} - r9n7{c7 c8} - r9n4{c8 c9} -
r2n4{c9 .} ==> r2c1≠6
t-whip-cn[6]: c1n7{r3 r8} - c1n6{r8 r5} - c1n5{r5 r4} - c8n5{r4 r7} - c9n5{r7 r5} -
c9n7{r5 .} ==> r2c3≠7
whip[1]: c3n7{r8 .} ==> r8c1≠7
naked-pairs-in-a-block: b1{r1c2 r2c3}{n2 n6} ==> r3c1≠2, r1c1≠2
hidden-pairs-in-a-row: r3{n2 n5}{c4 c7} ==> r3c7≠7, r3c4≠7, r3c4≠1
whip[1]: b2n1{r3c6 .} ==> r7c6≠1
naked-pairs-in-a-block: b2{r1c5 r3c4}{n2 n5} ==> r2c4≠2
biv-chain-bn[4]: b4n6{r6c2 r5c1} - b7n6{r8c1 r8c3} - b7n7{r8c3 r7c3} - b7n4{r7c3 r7c2}
==> r6c2≠4
hidden-single-in-a-block ==> r6c3=4
hidden-single-in-a-block ==> r7c2=4
biv-chain-rn[3]: r7n2{c5 c3} - r2n2{c3 c9} - r3n2{c7 c4} ==> r8c4≠2, r1c5≠2
singles ==> r1c5=5, r3c4=2, r3c7=5
finned-x-wing-in-rows: n2{r2 r8}{c9 c3} ==> r7c3≠2
singles to the end
```

6.5.5 Using type-restricted typed-whips

You thought it was the end of it? It is not! With respect to the choices of section 6.5.3, we can also restrict the types of the typed-whips and (of course) also those of the typed-bivalue-chains. In Sudoku, the natural restriction is to allow only type rc, i.e. patterns that lie totally in the "natural" rc-space. This amounts to not taking bilocality into consideration. For some players, bilocality, i.e. a number being present in only two places in a row (or column or block), is more difficult to spot than bivalue rc-cells. When additional t- or z- candidates need to be taken into account, as in whips, the difference in difficulty may be still greater. It is therefore a reasonable idea to start looking for whips or t-whips in the rc-space.

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Typed-Bivalue-Chains* TRUE)
(bind ?*Typed-Whips* TRUE)
(bind ?*restrict-csp-types-in-typed-chains* TRUE)
(bind ?*allowed-csp-types* (create$ rc))
```

We get a resolution path in rc-W12, i.e. we need significantly longer chains:

```
;;; Resolution state RS1
whip-rc[6]: r9c7{n7 n2} - r9c5{n2 n9} - r9c8{n9 n4} - r3c8{n4 n7} - r3c7{n7 n5} -
r1c7{n5 .} ==> r9c9≠7
whip-rc[7]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r2c3{n6 n7} - r2c1{n7 n4} -
r3c1{n4 n1} - r1c1{n1 .} ==> r1c3≠2
```

```
whip-rc[7]: r8c6{n7 n8} - r3c6{n8 n1} - r7c6{n1 n9} - r7c8{n9 n5} - r8c9{n5 n2} -
r9c9{n2 n4} - r2c9{n4 .} ==> r2c6≠7
whip-rc[10]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r2c3{n6 n7} - r2c9{n7 n4} -
r9c9{n4 n2} - r9c5{n2 n9} - r5c5{n9 n1} - r5c4{n1 n9} - r2c4{n9 .} ==> r2c1≠2
whip-rc[10]: r5c5{n1 n9} - r9c5{n9 n2} - r2c5{n2 n8} - r3c6{n8 n7} - r8c6{n7 n8} -
r7c5{n8 n5} - r7c8{n5 n9} - r7c6{n9 n1} - r7c4{n1 n7} - r8c4{n7 .} ==> r1c5≠1
whip-rc[12]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r2c3{n6 n7} - r1c1{n7 n1} -
r1c6{n1 n7} - r3c6{n7 n1} - r3c4{n1 n5} - r1c5{n5 n2} - r9c5{n2 n9} - r7c6{n9 n8} -
r8c6{n8 .} ==> r3c1≠2
whip-rc[8]: r3c8{n7 n4} - r2c9{n4 n2} - r1c7{n2 n5} - r1c5{n5 n2} - r9c5{n2 n9} -
r2c5{n9 n8} - r3c6{n8 n1} - r3c1{n1 .} ==> r3c7≠7
whip-rc[10]: r9c9{n2 n4} - r2c9{n4 n7} - r5c9{n7 n5} - r5c1{n5 n6} - r8c1{n6 n7} -
r2c1{n7 n4} - r3c1{n4 n1} - r1c1{n1 n2} - r1c5{n2 n5} - r1c7{n5 .} ==> r8c9≠2
whip-rc[5]: r8c6{n7 n8} - r3c6{n8 n1} - r7c6{n1 n9} - r7c8{n9 n5} - r8c9{n5 .} ==>
r1c6≠7
whip-rc[7]: r8c6{n7 n8} - r8c9{n8 n5} - r8c4{n5 n2} - r2c4{n2 n9} - r5c4{n9 n1} -
r5c5{n1 n9} - r9c5{n9 .} ==> r7c4≠7
whip-rc[9]: r3c2{n2 n8} - r5c2{n8 n3} - r1c2{n3 n6} - r1c6{n6 n1} - r3c6{n1 n7} -
r8c6{n7 n8} - r7c6{n8 n9} - r7c8{n9 n5} - r8c9{n5 .} ==> r6c2≠2
biv-chain-rc[4]: r4c1{n2 n5} - r5c1{n5 n6} - r6c2{n6 n4} - r7c2{n4 n2} ==> r8c1≠2
biv-chain-rc[4]: r8c9{n5 n8} - r8c6{n8 n7} - r8c1{n7 n6} - r5c1{n6 n5} ==> r5c9≠5
naked-single ==> r5c9=7
whip[1]: c9n5{r8 .} ==> r7c8≠5
singles ==> r7c8=9, r9c5=9, r5c5=1, r5c4=9, r2c6=9, r1c6=6, r6c2=6, r5c1=5, r4c1=2,
r4c8=5, r6c3=4, r5c8=6, r6c8=2, r7c2=4, r1c1=1
whip[1]: b7n2{r8c3 .} ==> r2c3≠2
whip[1]: r9n2{c9 .} ==> r7c9≠2
naked-pairs-in-a-row: r3{c1 c8}{n4 n7} ==> r3c6≠7, r3c4≠7
hidden-single-in-a-block ==> r2c4=7
whip-rc[3]: r7c9{n5 n8} - r7c5{n8 n2} - r8c4{n2 .} ==> r7c4≠5
biv-chain-rc[4]: r3c2{n2 n8} - r2c3{n8 n6} - r2c1{n6 n4} - r2c9{n4 n2} ==> r3c7≠2
singles to the end
```

Now, you can ask: can we go still further and restrict the allowed patterns to t-whips in rc-space only? We can, but not in a very useful way for this puzzle! The only thing CSP-Rules finds in rc-space after RS1 is this t-whip:

```
t-whip-rc[7]: r5c9{n7 n5} - r5c1{n5 n6} - r5c8{n6 n7} - r3c8{n7 n4} - r9c8{n4 n9} -
r9c5{n9 n2} - r9c7{n2 .} ==> r9c9≠7
```

Nevertheless, I am sure that, after playing with SudoRules and the various rule combinations it allows, you will be able to find examples where a t-whip solution in rc-space does not require chains much longer than when you allow the full power of all the 3D-chains. Puzzles of medium difficulty are the best candidates.

6.6 Goodies

SudoRules has currently two kinds of goodies: pretty-printing and shuffling.

6.6.1 Pretty printing of puzzles and resolution states

There are two functions for pretty printing a puzzle or a resolution state: ***pretty-print-sudoku-string*** and ***pretty-print-sukaku-list***, with respective syntax:

```
(pretty-print-sudoku-string ?sudoku-string)
(pretty-print-sukaku-list $?sukaku-list)
```

Examples should make it clear how they work. Command:

```
(pretty-print-sudoku-string
"1..5....2.2.7...3...3.6.1..........8.4..5..9.6.2........6...4...5...3.7.8....4..9")
```

will give:

```
     +-------+-------+-------+
     ! 1 . . ! 5 . . ! . . 2 !
     ! . 2 . ! 7 . . ! . 3 . !
     ! . . 3 ! . 6 . ! 1 . . !
     +-------+-------+-------+
     ! . . . ! . . . ! . . 8 !
     ! . 4 . ! . 5 . ! . 9 . !
     ! 6 . 2 ! . . . ! . . . !
     +-------+-------+-------+
     ! . . 6 ! . . . ! 4 . . !
     ! . 5 . ! . . 3 ! . 7 . !
     ! 8 . . ! . . 4 ! . . 9 !
     +-------+-------+-------+
```

and command:

```
(pretty-print-sukaku-list
   1 6 4789 5 3 89 789 48 2 459 2 4589 7 1489 189 5689 3 456 4579 789 3 2489 6 289 1
458 457 3579 1379 1579 12349 1249 12679 2367 1246 8 37 4 178 1238 5 12678 2367 9 1367
6 13789 2 13489 1489 1789 357 145 13457 2379 1379 6 1289 12789 5 4 128 13 249 5 149
12689 1289 3 268 7 16 8 137 17 126 127 4 2356 1256 9
)
```

will give:

```
   +-------------------+-------------------+-------------------+
   ! 1     6     4789  ! 5     3     89    ! 789   48    2     !
   ! 459   2     4589  ! 7     1489  189   ! 5689  3     456   !
   ! 4579  789   3     ! 2489  6     289   ! 1     458   457   !
   +-------------------+-------------------+-------------------+
   ! 3579  1379  1579  ! 12349 1249  12679 ! 2367  1246  8     !
   ! 37    4     178   ! 1238  5     12678 ! 2367  9     1367  !
   ! 6     13789 2     ! 13489 1489  1789  ! 357   145   13457 !
   +-------------------+-------------------+-------------------+
   ! 2379  1379  6     ! 1289  12789 5     ! 4     128   13    !
   ! 249   5     149   ! 12689 1289  3     ! 268   7     16    !
   ! 8     137   17    ! 126   127   4     ! 2356  1256  9     !
   +-------------------+-------------------+-------------------+
```

In the Sukaku case, grid size is adapted to the cell with the largest number of candidates.

6.6.2 Applying isomorphisms

There are a few functions for finding isomorphic variants of a puzzle (currently restricted to 9×9 puzzles:

```
(permute-floors-9x9 ?puzzle-string ?f1 ?f2 ?f3)
```
where ?puzzle-string is the usual string format for a puzzle and (?f1, ?f2, ?f3) is supposed to be a permutation of (1, 2, 3). This will permute the three floors according to the given permutation.

```
(permute-rows-in-floor-9x9 ?floor ?r1 ?r2 ?r3)
```
where ?floor is supposed to be given as a string of 27 characters and (?r1, ?r2, ?r3) is supposed to be a permutation of (1, 2, 3). This will permute the three rows in the floor according to the given permutation.

```
(horizontal-random-shuffle-9x9 ?puzzle-string)
```
where ?puzzle-string is the usual string format for a puzzle. This will randomly permute the three floors and the three rows in each floor.

```
(diagonal-symmetry-9x9 ?puzzle-string)
```
will apply a diagonal symmetry, as the name suggests. Notice that, in combination with horizontal-random-shuffle-9x9, this allows to make vertical random shuffles.

Finally,
```
(random-shuffle-9x9-puzzle ?puzzle-string)
```
will produce a random isomorph of the original puzzle.

There is currently no independent shuffle-digits function. It's an easy exercise for the reader to write one.

****

WARNING: the next sections of this chapter are for more advanced users.

For the most part, they are additions to the original release of CSP-Rules-V2.1.

****

One thing the following sections will illustrate in particular is the independence of the resolution rules and the simplest-first strategy, by introducing alternative ways of using the rules. Don't try to use this material before you master the workings of the configuration file and you can confidently use the previous elementary functions. Many of the advanced functions are written in (almost) generic form, but they are currently tested and available only for Sudoku.

6.7 Bases for more elaborated functions and techniques

This section is for users who want to push some rules as far as possible before trying different ones – thus partly changing the default simplest-first strategy of CSP-Rules.

First three points to notice are: 1) when the set of resolution rules selected by the user is not enough to completely solve a puzzle, the final resolution state is now printed; 2) it is printed in a form fully compatible with the input of functions solve-sukaku-list or solve-sukaku-grid; 3) it is also fully compatible with the input of the init-sukaku-list or init-sukaku-grid functions defined in the first subsection.

6.7.1 Init functions

Knowing the existence of initialisation functions was not necessary until now. However, they may be useful in conjunction with the advanced features of SudoRules described in the forthcoming sections.

To make it simple, each of the 2×3 solve-xxx functions defined in sections 6.1 to 6.2.4 has a corresponding init-xxx function, with the same syntax, and with its name obtained by replacing "solve" by "init", namely: ***init-sudoku-string, init-sukaku-string, init-sudoku-list, init-sukaku-list, init-sudoku-grid, init-sukaku-grid***. For consistency with "solve", "init-sudoku-string" can be abbreviated as "***init***".

Whereas the solve-xxx functions completely solve a puzzle (if the selected rules are powerful enough for this), starting from their data and applying all the active resolution rules, the init-xxx functions only read the given data and initialise CLIPS without applying any resolution rule. They don't output anything. Apart from some recording of the init and solve times, solve-xxx is basically the combination of init-xxx with CLIPS function "run" (which starts applying the resolution rules).

6.7.2 Printing the current resolution state and the four 2D-views

At any point of the resolution process, you can print the current resolution state, by calling function "***print-current-resolution-state***" with no argument, i.e. as
`(print-current-resolution-state).`

Moreover, this is done systematically at the end of resolution if a puzzle is not fully solved by the chosen set of rules. The output is in a form readable by function “init-sukaku-grid”, as shown in the following section.

There are also functions to print any of the other views of the same resolution state, i.e. the views in the rn, cn and bn spaces, in the current resolution state: “***print-current-resolution-state-rn-view***”, “***print-current-resolution-state-cn-view***”, “***print-current-resolution-state-bn-view***”. Moreover, there is a function to print all four views: “***print-current-resolution-state-all-views***”. All these functions have the same syntax as “print-current-resolution-state”.

After some time of use, it appeared to be convenient to have abbreviations: ***print-RS-rc*** (also ***print-RS***), ***print-RS-rn***, ***print-RS-cn***, ***print-RS-bn***, ***print-RS-all***. I let you guess which abbreviates which.

6.7.3 Examples

As a first elementary example of how these functions and the associated rn, cn and bn representations can be used, in line with what was described in [HLS, 2007], consider the following puzzle (Royle17#21), already studied in section XV.3.1 of [HLS]:

```
+-------+-------+-------+
! . . . ! . . . ! . 3 1 !
! . 8 . ! . 4 . ! . . . !
! . 7 . ! . . . ! . . . !
+-------+-------+-------+
! 1 . 6 ! 3 . . ! . 7 . !
! 3 . . ! . . . ! . . . !
! . . . ! . 8 . ! . . . !
+-------+-------+-------+
! 5 4 . ! . . . ! 8 . . !
! . . . ! 6 . . ! 2 . . !
! . . . ! 1 . . ! . . . !
+-------+-------+-------+

.......31.8..4.....7.......1.63...7.3............8....54....8.....6..2.....1.....
SER = 7.1
```

After 38 Singles and 3 whips[1], the resolution state is as follows. For the player not yet used to spot conjugated digits but knowing the most basic chains, i.e. the xy-chains, there's no obvious chain.

```
+-------------+-------------+-------------+
! 4   6   5   ! 8   279 29  ! 79  3   1   !
! 9   8   1   ! 57  4   3   ! 6   25  257 !
! 2   7   3   ! 59  1   6   ! 59  8   4   !
+-------------+-------------+-------------+
! 1   29  6   ! 3   29  5   ! 4   7   8   !
! 3   259 8   ! 4   6   7   ! 1   259 259 !
! 7   259 4   ! 29  8   1   ! 3   6   259 !
+-------------+-------------+-------------+
! 5   4   279 ! 279 3   29  ! 8   1   6   !
! 8   1   79  ! 6   579 4   ! 2   59  3   !
! 6   3   29  ! 1   259 8   ! 57  4   579 !
+-------------+-------------+-------------+
```

However, the cn-view, in which physical rows are columns, physical columns are digits and data are rows:

```
cn-view:
  4       3       5       1       7       9       6       8       2
  8       456     9       7       56      1       3       2       456
  2       79      3       6       1       4       78      5       789
  9       67      4       5       23      8       27      1       367
  3       149     7       2       89      5       18      6       1489
  6       17      2       8       4       3       5       9       17
  5       8       6       4       39      2       19      7       13
  7       25      1       9       258     6       4       3       58
  1       256     8       3       2569    7       29      4       569
```

shows what looks like an xy-chain[3] (lying on the cells in bold) and is therefore a bivalue-chain-cn[3] (and is enough to solve the puzzle):

```
biv-chain-cn[3]: c7n7{r9 r1} - c5n7{r1 r8} - c5n5{r8 r9} ==> r9c7≠5
stte
```

As a different use case, consider the puzzle below:

```
+-------+-------+-------+
! . 8 6 ! . 7 9 ! . . 5 !
! 4 . 9 ! 8 5 . ! 7 6 . !
! 7 5 . ! . . 6 ! 9 8 . !
+-------+-------+-------+
! . 9 . ! 5 . . ! 8 7 6 !
! 5 . 7 ! 6 8 . ! . 2 9 !
! 8 6 . ! . 9 7 ! 5 . . !
+-------+-------+-------+
! 9 . 8 ! 7 1 . ! 6 5 . !
! 6 . 5 ! 9 . 8 ! . . 7 !
! . 7 . ! . 6 5 ! . 9 8 !
+-------+-------+-------+
```

```
.86.79..54.985.76.75...698..9.5..8765.768..2986..975..9.871.65.6.59.8..7.7..65.98
```

First initialise SudoRules with it, e.g.:

```
(init
  “.86.79..54.985.76.75...698..9.5..8765.768..2986..975..9.871.65.6.59.8..7.7..65.98")
```

and type the command (print-RS-all). The four 2D views will be displayed.

```
standard rc-view:
Physical rows are rows, physical columns are columns. Data are digits.
   123      8        6        1234     7        9        1234     134      5
   4        123      9        8        5        123      7        6        123
   7        5        123      1234     234      6        9        8        1234
   123      9        1234     5        234      1234     8        7        6
   5        134      7        6        8        134      134      2        9
   8        6        1234     1234     9        7        5        134      134
   9        234      8        7        1        234      6        5        234
   6        1234     5        9        234      8        1234     134      7
   123      7        1234     234      6        5        1234     9        8



The following representations may be used e.g. to more easily spot
rn-, cn- or bn- bivalue pairs (also named bilocal pairs), mono-typed-chains,
Hidden Subsets and Fishes (which will appear as Naked Subsets in the proper space.

rn-view:
Physical rows are rows, physical columns are digits. Data are columns.
   1478     147      1478     478      9        3        5        2        6
   269      269      269      1        5        8        7        4        3
   349      3459     3459     459      2        6        1        8        7
   136      1356     1356     356      4        9        8        7        2
   267      8        267      267      1        4        3        5        9
   3489     34       3489     3489     7        2        6        1        5
   5        269      269      269      8        7        4        3        1
   278      257      2578     2578     3        1        9        6        4
   137      1347     1347     347      6        5        2        9        8

cn-view:
Physical rows are columns, physical columns are digits. Data are rows.
   149      149      149      2        5        8        3        6        7
   258      278      2578     578      3        6        9        1        4
   3469     3469     3469     469      8        1        5        7        2
   136      1369     1369     1369     4        5        7        2        8
   7        348      348      348      2        9        1        5        6
   245      247      2457     457      9        3        6        8        1
   1589     189      1589     1589     6        7        2        4        3
   168      5        168      168      7        2        4        3        9
   236      237      2367     367      1        4        8        9        5
```

```
bn-view:
Physical rows are blocks, physical columns are digits. Data are positions in a block.
   159       159       159       4         8         3         7         2         6
   167       1678      1678      178       5         9         2         4         3
   1269      169       1269      129       3         5         4         8         7
   1359      139       1359      359       4         8         6         7         2
   367       237       2367      2367      1         4         9         5         8
   489       5         489       489       7         3         2         1         6
   579       2579      2579      259       6         4         8         3         1
   2         357       357       357       9         8         1         6         4
   457       347       3457      3457      2         1         6         9         8
```

Notice how the rn, cn and bn spaces are represented: all the n coordinates correspond to physical columns. In a sequential presentation as here, this proved to be easier to use than the representation given by the Extended Sudoku Board of [HLS] or [PBCS] (where the c and n coordinates are exchanged in the cn-space).

Now, let's show another elementary way how this can be used. This puzzle was proposed as a candidate for having no bivalue cell (an extremely rare property for a Sudoku puzzle). This seems to be true in the standard rc-space: no rc-cell has only two candidates. But, if you consider the rn-space, you can see that there is an rn-cell, namely r6n2 (in bold), with only two values (34, i.e. c3 and c4), meaning that on row 6, number 2 can only be in columns c3 and c4 (rn-cell r6c2 is rn-bivalue; or: it is bilocal in the usual Sudoku jargon); still another way of saying this is: candidates n2r6c3 and n2r6c4 make a (rn-)bivalue pair. *This puzzle has only one bivalue pair*.

If you want a puzzle with no bivalue pair after Singles have been applied, it's very rare, but the first one has been found by Nazaz (exercise: check this property visually, using the above-mentioned functions):

```
.3.21.45.41..5..235.23.41.61.5..234.72.43.5.1.431.5..225..41.3...15.32.43.482..15
```

6.8 Solving a puzzle in stages

```
+-------+-------+-------+
! . . . ! . . . ! 1 . . !
! . . 1 ! 2 . . ! . . 3 !
! . 4 . ! . 5 . ! . 2 . !
+-------+-------+-------+
! . 5 . ! . 6 . ! . 4 . !
! . . 7 ! 3 . . ! . . 6 !
! . . . ! . . 7 ! 3 . . !
+-------+-------+-------+
! 2 . . ! . . . ! . . . !
! . . 8 ! 7 . . ! . . 1 !
! . 6 . ! . 4 . ! . 5 . !
+-------+-------+-------+
```

```
......1....12....3.4..5..2..5..6..4...73....6.....73..2.........87....1.6..4..5.
SER = 7.1
```

Suppose you want to solve the above puzzle, created by Mith. As Mith is famous for proposing the best ever puzzles involving (Naked, Hidden and Super-Hidden) Subsets, it is natural to try to find as many of them as possible. However, Subsets and Finned Fish are not enough for this puzzle. In addition to them, it requires at least bivalue-chains. As a result, a natural choice of rules is:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Bivalue-Chains* TRUE)
```

We get the following resolution path, using an extraordinarily large number of Subsets (15) intermingled with a few short bivalue-chains:

```
243 candidates, 1790 csp-links and 1790 links. Density = 6.09%
naked-pairs-in-a-block: b7{r8c2 r9c3}{n3 n9} ==> r9c1≠9, r9c1≠3, r8c1≠9, r8c1≠3,
r7c3≠9, r7c3≠3, r7c2≠9, r7c2≠3
hidden-pairs-in-a-block: b5{r5c6 r6c4}{n4 n5} ==> r6c4≠9, r6c4≠8, r6c4≠1, r5c6≠9,
r5c6≠8, r5c6≠2, r5c6≠1
hidden-pairs-in-a-block: b3{r1c9 r2c7}{n4 n5} ==> r2c7≠9, r2c7≠8, r2c7≠7, r2c7≠6,
r1c9≠9, r1c9≠8, r1c9≠7
finned-x-wing-in-rows: n3{r9 r3}{c6 c3} ==> r1c3≠3
finned-x-wing-in-columns: n3{c2 c5}{r1 r8} ==> r8c6≠3
;;; Resolution state RS1
biv-chain[2]: c2n3{r1 r8} - r9n3{c3 c6} ==> r1c6≠3
swordfish-in-columns: n3{c2 c5 c8}{r8 r1 r7} ==> r7c6≠3, r1c1≠3
swordfish-in-columns: n4{c3 c4 c9}{r7 r6 r1} ==> r7c7≠4, r6c1≠4, r1c6≠4
swordfish-in-columns: n7{c2 c5 c8}{r7 r2 r1} ==> r7c9≠7, r7c7≠7, r2c1≠7, r1c1≠7
swordfish-in-columns: n1{c2 c5 c8}{r6 r7 r5} ==> r7c6≠1, r7c4≠1, r6c1≠1, r5c1≠1
hidden-triplets-in-a-column: c1{n1 n3 n7}{r9 r4 r3} ==> r4c1≠9, r4c1≠8, r3c1≠9, r3c1≠8,
r3c1≠6
hidden-triplets-in-a-row: r7{n1 n3 n7}{c2 c5 c8} ==> r7c8≠9, r7c8≠8, r7c8≠6, r7c5≠9,
r7c5≠8
swordfish-in-rows: n5{r2 r5 r8}{c1 c7 c6} ==> r7c6≠5, r1c1≠5
biv-chain[3]: r9c3{n9 n3} - c2n3{r8 r1} - b1n2{r1c2 r1c3} ==> r1c3≠9
biv-chain[3]: c2n3{r1 r8} - b9n3{r8c8 r7c8} - r7n7{c8 c2} ==> r1c2≠7
biv-chain[3]: r1n7{c8 c5} - b2n3{r1c5 r3c6} - r3c1{n3 n7} ==> r3c7≠7, r3c9≠7
singles ==> r3c1=7, r9c1=1, r4c1=3, r7c2=7,  r7c8=3, r7c5=1
biv-chain[3]: r1n2{c2 c3} - r4c3{n2 n9} - b7n9{r9c3 r8c2} ==> r1c2≠9
biv-chain[3]: r4c3{n9 n2} - r1n2{c3 c2} - b1n3{r1c2 r3c3} ==> r3c3≠9
jellyfish-in-columns: n8{c1 c8 c2 c5}{r6 r5 r2 r1} ==> r6c9≠8, r5c7≠8, r2c6≠8, r1c6≠8,
r1c4≠8
naked-triplets-in-a-block: b2{r1c4 r1c6 r2c6}{n4 n6 n9} ==> r3c6≠9, r3c6≠6, r3c4≠9,
r3c4≠6, r2c5≠9, r1c5≠9
whip[1]: r3n9{c9 .} ==> r1c8≠9, r2c8≠9
naked-triplets-in-a-column: c4{r3 r4 r9}{n8 n1 n9} ==> r7c4≠9, r7c4≠8, r1c4≠9
whip[1]: b2n9{r2c6 .} ==> r4c6≠9, r7c6≠9, r8c6≠9, r9c6≠9
whip[1]: r7n9{c9 .} ==> r8c7≠9, r8c8≠9, r9c7≠9, r9c9≠9
singles to the end
```

Now, suppose you are a patented fisherman and you want to find all the possible fishes before you try anything else. The simplest-first strategy, the default strategy in CSP-Rules, doesn't allow this if other simpler patterns are loaded, as shown in the above path. But there is a way to do it nevertheless. It will take two steps:

First step: load SudoRules with the following settings (no bivalue-chains):

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
```

and run it for our puzzle:

```
;;; Same path up to resolution state RS1
swordfish-in-columns: n3{c2 c5 c8}{r8 r1 r7} ==> r7c6≠3, r1c6≠3, r1c1≠3
swordfish-in-columns: n4{c3 c4 c9}{r7 r6 r1} ==> r7c7≠4, r6c1≠4, r1c6≠4
swordfish-in-columns: n7{c2 c5 c8}{r7 r2 r1} ==> r7c9≠7, r7c7≠7, r2c1≠7, r1c1≠7
swordfish-in-columns: n1{c2 c5 c8}{r6 r7 r5} ==> r7c6≠1, r7c4≠1, r6c1≠1, r5c1≠1
hidden-triplets-in-a-column: c1{n1 n3 n7}{r9 r4 r3} ==> r4c1≠9, r4c1≠8, r3c1≠9, r3c1≠8,
r3c1≠6
hidden-triplets-in-a-row: r7{n1 n3 n7}{c2 c5 c8} ==> r7c8≠9, r7c8≠8, r7c8≠6, r7c5≠9,
r7c5≠8
swordfish-in-rows: n5{r2 r5 r8}{c1 c7 c6} ==> r7c6≠5, r1c1≠5
jellyfish-in-columns: n8{c1 c8 c2 c5}{r6 r5 r2 r1} ==> r6c9≠8, r5c7≠8, r2c6≠8, r1c6≠8,
r1c4≠8
naked-triplets-in-a-block: b2{r1c4 r1c6 r2c6}{n4 n6 n9} ==> r3c6≠9, r3c6≠6, r3c4≠9,
r3c4≠6, r2c5≠9, r1c5≠9
naked-triplets-in-a-column: c4{r3 r4 r9}{n1 n8 n9} ==> r7c4≠9, r7c4≠8, r1c4≠9
whip[1]: b2n9{r2c6 .} ==> r4c6≠9, r7c6≠9, r8c6≠9, r9c6≠9
whip[1]: r7n9{c9 .} ==> r8c7≠9, r8c8≠9, r9c7≠9, r9c9≠9
jellyfish-in-rows: n9{r3 r9 r4 r7}{c9 c3 c4 c7} ==> r6c9≠9, r6c3≠9,
r5c7≠9, r1c3≠9
naked-pairs-in-a-block: b6{r5c7 r6c9}{n2 n5} ==> r4c9≠2, r4c7≠2
PUZZLE 0 NOT SOLVED. 59 VALUES MISSING.
```

Hurrah! The puzzle is not solved (this is what "Subsets and Finned Fish are not enough" means) but we have found one more fish – and not a small one: a big and rare Jellyfish (in bold). In many other cases, the difference in the numbers of Subsets will be much larger.

But now, what? Indeed, the above resolution path is what you would have gotten without the new feature. There was no documented way of going further. Now, as the puzzle is not fully solved, you get automatically the following additional output:

```
FINAL RESOLUTION STATE:
   689      23789    256      46       378      69       1        6789     45
   5689     789      1        2        78       469      45       6789     3
   37       4        369      18       5        138      6789     2        789
   13       5        239      189      6        128      789      4        789
   489      1289     7        3        1289     45       25       189      6
   689      1289     246      45       1289     7        3        189      25
   2        17       45       56       13       68       689      37       489
   45       39       8        7        239      256      246      36       1
   17       6        39       189      4        1238     278      5        278
```

We can now make the second step. Load another instance of SudoRules, with the following settings:

```
(bind ?*Subsets* TRUE)
(bind ?*FinnedFish* TRUE)
(bind ?*Bivalue-Chains* TRUE)
```

and, instead of re-starting from the beginning, start from the above resolution state:

```
(solve-sukaku-grid
   689      23789    256      46       378      69       1        6789     45
   5689     789      1        2        78       469      45       6789     3
   37       4        369      18       5        138      6789     2        789
   13       5        239      189      6        128      789      4        789
   489      1289     7        3        1289     45       25       189      6
   689      1289     246      45       1289     7        3        189      25
   2        17       45       56       13       68       689      37       489
   45       39       8        7        239      256      246      36       1
   17       6        39       189      4        1238     278      5        278
)
```

You can see that Subsets were not very far from solving the puzzle. Only two short bivalue-chains[3] and two more Subsets will do it:

```
165 candidates, 672 csp-links and 672 links. Density = 4.97%
biv-chain[3]: r1c4{n6 n4} - r6n4{c4 c3} - b4n6{r6c3 r6c1} ==> r1c1≠6
biv-chain[3]: r3c1{n7 n3} - b2n3{r3c6 r1c5} - b2n7{r1c5 r2c5} ==> r2c2≠7
naked-pairs-in-a-block: b1{r1c1 r2c2}{n8 n9} ==> r3c3≠9, r2c1≠9, r2c1≠8, r1c2≠9, r1c2≠8
whip[1]: r3n9{c9 .} ==> r1c8≠9, r2c8≠9
whip[1]: c8n9{r6 .} ==> r4c7≠9, r4c9≠9
naked-pairs-in-a-block: b6{r4c7 r4c9}{n7 n8} ==> r6c8≠8, r5c8≠8
singles to the end
```

6.9 Focusing on some candidates for eliminations

This section is for users who, for some reason of their own, want to focus their attention on some candidate(s).

One of those reasons (that will be considered in detail in the next sections) can be that a candidate is a T_0-anti-backdoor and eliminating it would lead to a single-step solution (considering rules from T_0 as obvious steps not to be counted as real steps); the same remarks apply to two-step solutions and anti-backdoor-pairs.

Generally speaking, generic function ***"try-to-eliminate-candidates***" takes any number of candidates (which may be a single candidate in spite of the final "s" in the name) as arguments and it applies to them – and only to them – the rules selected in the configuration file. During such focused eliminations, the simplest-first strategy still applies; it is only restricted to apply to candidates in the list.

Function “try-to-eliminate-candidates” can be applied repeatedly to any lists of candidates; each time, the starting point is the current resolution state. Syntax:

`(try-to-eliminate-candidates cand1 cand2 cand3 ...)`
where $cand_i$ is defined by its label – which, in SudoRules, is given by function: (nrc-to-label n_i r_j c_k). In practice, if we are dealing with a 9×9 Sudoku and no rule involving g-candidates (e.g. g-whips, g-braids, …) is loaded, this can merely be: $n_ir_jc_k$. However, if any such rule is activated, it is mandatory to use the nrc-to-label function. The result of this function is the elimination of all the candidates in the list that the active rules can eliminate.

Remarks:
- All the generic chain rules can be used for focused eliminations, except t-whips:
- for technical reasons (because they use the same partial-whips[1] as whips), ***t-whips (typed or not) MAY NOT be loaded for focusing to work****. The “try-to-eliminate-candidates” function will merely halt if t-whips are loaded.*
- For chain rules that have a blocked version (e.g. bivalue-chains), not only the candidates focused on are eliminated, but also the other targets of the same *chain. This is a deliberate choice. If you want to eliminate only the candidates on focus, select (in the configuration file) the non-blocked version of these rules.*
- In all the applications delivered with CSP-Rules, Subsets now allow focused eliminations (with the same remarks as above about their blocked vs non-blocked versions).
- With focused eliminations, the simplest-first strategy still applies, restricted to the candidates on focus.
- All the previous remarks also apply to the search for 1-step, 2-step or fewer-steps solutions (which are based on focused eliminations).
- In forcing chains, focusing is done on the starting candidates, not on the potentially asserted or eliminated candidates; as a result, it is not recommended to explicitly activate other rules (apart from those they automatically imply) when any forcing chains are active – or conversely.

6.10 Considering some rules as “no-step”

When one is interested in the number of steps of a resolution path, this will generally be associated with the idea that some steps are trivial and should not be counted as real steps.

6.10.1 The general framework

Let us make this idea slightly more formal. Consider two resolution theories T and T_0, with $T_0 \subset T$ (as will appear in the 1-step, 2-step and fewer-steps cases below). T is the set of rules we want to use to solve a puzzle P; T_0 is the set of rules we consider as no-step.

T_0 is supposed to have the confluence property. This condition ensures that the final result of applying rules in T_0, starting from any resolution state, will be independent of the order of their application (and therefore a unique and well-defined resolution state). Typical examples for T_0 are:

- BRT (Basic Resolution Theory) = Singles + eliminations by direct contradictions between a decided value and a candidate
- W_1 = BRT + whips[1]
- S_2 = W_1 + Subsets[2] (i.e. + Naked, Hidden and Supper-Hidden Pairs)
- S_3 = S_2 + Subsets[3] (i.e. + Naked, Hidden and Supper-Hidden Triplets)
- S = S_4 = S_3 + Subsets[4] (i.e. + Naked, Hidden and Supper-Hidden Quads)

Notice that I consider that only BRT and W_1 can rationally be considered as no-step, especially as Subsets of any size tend to eliminate lots of candidates.

Typical examples for T will be defined by a set of rule types and their maximal length. To make it concrete, in CSP-Rules this means the following choices in the configuration file (the semi-colons mean rules that I don't usually consider, but that could be):

```
(defglobal ?*all-chains-max-length* = xx)

; (bind ?*Subsets[2]* TRUE)
; (bind ?*Subsets[3]* TRUE)
; (bind ?*Subsets[4]* TRUE)
 (bind ?*Subsets* TRUE)
 (bind ?*Bivalue-Chains* TRUE)
 (bind ?*z-Chains* TRUE)
 (bind ?*Whips* TRUE)
 (bind ?*Typed-Bivalue-Chains* TRUE)
 (bind ?*Typed-z-Chains* TRUE)
 (bind ?*Typed-Whips* TRUE)
; (bind ?*G-Whips* TRUE)
; (bind ?*Oddagons* TRUE)
; (bind ?*Braids* TRUE)
; (bind ?*Typed-Braids* TRUE)
; (bind ?*G-Braids* TRUE)
```

The maximum allowed value for all the chains (?*all-chains-max-length*), here written as xx must be selected carefully. If a puzzle has a solution using only bivalue-chains[3], it would be totally absurd to look for a resolution path with fewer steps but involving any chains of length much larger than 3. (Unfortunately for naive observers of some Sudoku forums, such nonsense is often implicitly proposed as a normal way to solve a puzzle.) It is trie that a human solver doesn't care too much about the length of the chains he uses, but there are limits.

As a rule of thumb, ?*all-chains-max-length* should be taken as close as possible to the maximum length reached in the simplest-first solution with the same set of rules (i.e. equal to the T rating in the best case and not more than it +1 or +2 in any case). Beware also that, if you choose a too high value for ?*all-chains-max-length*, computation times may become very long. This means, in every case this framework is applied to a puzzle, one should first computed its T-rating.

6.11 Finding backdoors, anti-backdoors or anti-backdoor-pairs

If T_0 is a resolution theory with the confluence property, one can easily define the notions of a *T_0-backdoor* and a *T_0-anti-backdoor*: a candidate Z is a ***T_0-backdoor*** (respectively a ***T_0-anti-backdoor***) if adding Z as a decided value (respectively eliminating Z) allows to obtain a solution in T_0. Backdoors are mainly related to techniques of guessing, but anti-backdoors can be used in more interesting ways. The reason for T_0 being required to have the confluence property in these definitions is the usual one (as developed in [PBCS]): it is a necessary condition for the final result of applying rules in T_0 to be independent of the order of their application.

Here, T_0 can be any resolution theory with the confluence property. But in practice, it will satisfy the conditions defined in section 6.10.

Although they have very different use cases, backdoors and anti-backdoors can be found in SudoRules with totally similar commands. I shall therefore illustrate here only anti-backdoors (for backdoors, merely delete the “anti-” part of the function names). The puzzle I’ll use for this purpose (below) is again one from Mith.

```
+-------+-------+-------+
! . . 9 ! . . . ! . . . !
! 8 . . ! . . 9 ! 7 . . !
! . 7 . ! . 6 . ! . 5 . !
+-------+-------+-------+
! . 5 . ! . 4 . ! . 6 . !
! 9 . . ! . . 3 ! 8 . . !
! . . 2 ! . . . ! . . . !
+-------+-------+-------+
! . 6 . ! . 5 . ! 1 7 . !
! 2 . . ! . . 8 ! 4 . . !
! . . . ! . 1 . ! . . . !
+-------+-------+-------+
```

```
..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1....
SER = 7.1
```

Firstly, choose the following settings in the T&E-related part (i.e. section 2e) of the SudoRules configuration file (don’t forget to de-activate anything in the other sections of that file):

```
  (bind ?*Anti-Backdoors* TRUE)
  (bind ?*Whips[1]* TRUE)
```

Secondly, ask SudoRules to find the anti-backdoors:
- either this Sudoku-specific way:

```
(find-sudoku-anti-backdoors
   "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
)
```

- or, more generally, using the generic "***find-anti-backdoors***" function, after some Sudoku-specific initialisation:

```
(init-sudoku-string
  "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
)
(find-anti-backdoors)
```

Depending on whether you have activated nothing more than anti-backdoors or also whips[1] or also Subsets in the configuration file, you'll get different results, as shown below:

```
1 BRT-ANTI-BACKDOOR FOUND:n1r8c2

13 W1-ANTI-BACKDOORS FOUND: n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8
n8r6c5 n9r4c4 n8r4c3 n8r3c4 n8r1c8

38 S-ANTI-BACKDOORS FOUND: n9r9c8 n5r9c7 n6r9c6 n4r9c3 n8r9c2 n6r8c9 n3r8c8 n9r8c5
n7r8c4 n5r8c3 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n4r7c6 n3r7c1 n7r6c9 n9r6c8 n3r6c7 n5r6c6
n8r6c5 n3r6c2 n6r6c1 n5r5c9 n6r5c4 n3r4c9 n9r4c4 n8r4c3 n2r3c6 n8r3c4 n4r3c1 n5r2c4
n6r2c3 n8r1c8 n6r1c7 n7r1c5 n4r1c4 n5r1c1
```

If T_0 has the confluence property, the notion of a T_0-anti-backdoor can be generalised to any number of candidates. For instance, a ***T_0-anti-backdoor-pair*** is a couple of different candidates Z1 and Z2 such that eliminating both would lead to a solution in T_0 (which *a priori* eliminates from consideration any bivalue pair). After selecting the proper choices in the configuration file:

```
  (bind ?*Anti-Backdoor-pairs* TRUE)
  (bind ?*Whips[1]* TRUE)
```

the functions for finding T_0-anti-backdoor-pairs are:

- either this Sudoku-specific one:

```
(find-sudoku-anti-backdoor-pairs
   "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
)
```

- or, more generally, the generic one "***find-anti-backdoor-pairs***", after some Sudoku-specific initialisation:

```
(init-sudoku-string
  "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1....")
```

```
(find-anti-backdoor-pairs)
```

The next sections will show examples of how anti-backdoors and anti-backdoor-pairs can be combined with another new feature of CSP-Rules to produce “single-step” solutions or “two-step” solutions.

6.12 Looking for one-step solutions

The puzzle at the start of section 6.11:

```
..9.....8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1....
```

has a solution using only Subsets and bivalue-chains of size no more than 3. For comparison with the forthcoming alternative solutions, it will be very useful to keep the resolution path in mind.

The resolution state after Singles (and whips[1]) is:

```
+-------------------------+-------------------------+-------------------------+
! 13456   1234    9       ! 1234578 2378    12457   ! 236     12348   123468  !
! 8       1234    13456   ! 12345   23      9       ! 7       1234    12346   !
! 134     7       134     ! 12348   6       124     ! 239     5       123489  !
+-------------------------+-------------------------+-------------------------+
! 137     5       1378    ! 12789   4       127     ! 239     6       12379   !
! 9       14      1467    ! 12567   27      3       ! 8       124     12457   !
! 13467   1348    2       ! 156789  789     1567    ! 359     1349    134579  !
+-------------------------+-------------------------+-------------------------+
! 34      6       348     ! 2349    5       24      ! 1       7       2389    !
! 2       139     1357    ! 3679    379     8       ! 4       39      3569    !
! 3457    3489    34578   ! 234679  1       2467    ! 23569   2389    235689  !
+-------------------------+-------------------------+-------------------------+
```

```
hidden-pairs-in-a-block: b1{n5 n6}{r1c1 r2c3} ==> r2c3≠4, r2c3≠3, r2c3≠1, r1c1≠4,
r1c1≠3, r1c1≠1
finned-x-wing-in-rows: n9{r7 r4}{c4 c9} ==> r6c9≠9
finned-x-wing-in-columns: n9{c5 c8}{r6 r8} ==> r8c9≠9
t-whip[2]: r7n9{c4 c9} - c8n9{r9 .} ==> r6c4≠9
swordfish-in-columns: n9{c2 c5 c8}{r9 r8 r6} ==> r9c9≠9, r9c7≠9, r9c4≠9, r8c4≠9, r6c7≠9
swordfish-in-columns: n8{c2 c5 c8}{r9 r6 r1} ==> r9c9≠8, r9c3≠8, r6c4≠8, r1c9≠8, r1c4≠8
hidden-pairs-in-a-block: b5{n8 n9}{r4c4 r6c5} ==> r6c5≠7, r4c4≠7, r4c4≠2, r4c4≠1
hidden-pairs-in-a-row: r9{n8 n9}{c2 c8} ==> r9c8≠3, r9c8≠2, r9c2≠4, r9c2≠3
swordfish-in-columns: n2{c2 c5 c8}{r1 r2 r5} ==> r5c9≠2, r5c4≠2, r2c9≠2, r2c4≠2,
r1c9≠2, r1c7≠2, r1c6≠2, r1c4≠2
swordfish-in-columns: n5{c1 c6 c7}{r9 r1 r6} ==> r9c9≠5, r9c3≠5, r6c9≠5, r6c4≠5, r1c4≠5
swordfish-in-rows: n6{r2 r5 r8}{c9 c3 c4} ==> r9c9≠6, r9c4≠6, r6c4≠6, r1c9≠6
hidden-pairs-in-a-block: b5{n5 n6}{r5c4 r6c6} ==> r6c6≠7, r6c6≠1, r5c4≠7, r5c4≠1
hidden-pairs-in-a-block: b9{n5 n6}{r8c9 r9c7} ==> r9c7≠3, r9c7≠2, r8c9≠3
whip[1]: b9n2{r9c9 .} ==> r3c9≠2, r4c9≠2
hidden-pairs-in-a-column: c7{n2 n9}{r3 r4} ==> r4c7≠3, r3c7≠3
biv-chain[3]: r3n9{c7 c9} - r3n8{c9 c4} - r4c4{n8 n9} ==> r4c7≠9
singles ==> r4c7=2, r3c7=9, r5c5=2, r2c5=3
naked-pairs-in-a-row: r5{c2 c8}{n1 n4} ==> r5c9≠4, r5c9≠1, r5c3≠4, r5c3≠1
swordfish-in-columns: n3{c2 c7 c8}{r8 r1 r6} ==> r8c4≠3, r8c3≠3, r6c9≠3, r6c1≠3, r1c9≠3
```

```
biv-chain[3]: r7n9{c9 c4} - c4n3{r7 r9} - r9c9{n3 n2} ==> r7c9≠2
hidden-single-in-a-block ==> r9c9=2
hidden-triplets-in-a-column: c4{n2 n8 n9}{r7 r3 r4} ==> r7c4≠4, r7c4≠3, r3c4≠4, r3c4≠1
hidden-single-in-a-block ==> r9c4=3
whip[1]: b8n4{r9c6 .} ==> r1c6≠4, r3c6≠4
hidden-triplets-in-a-column: c9{n3 n8 n9}{r4 r3 r7} ==> r4c9≠7, r4c9≠1, r3c9≠4, r3c9≠1
whip[1]: r3n4{c3 .} ==> r1c2≠4, r2c2≠4
whip[1]: c2n4{r6 .} ==> r6c1≠4
biv-chain[3]: r5n1{c8 c2} - r2c2{n1 n2} - c8n2{r2 r1} ==> r1c8≠1
biv-chain[3]: r3c9{n3 n8} - c4n8{r3 r4} - r4n9{c4 c9} ==> r4c9≠3
singles to the end
```

One can see that each step is easy but there are many steps (some of which are probably unnecessary). Now, suppose we want a single-step solution – modulo steps in some resolution theory T_0 made of easy rules (W1 for instance) that one would consider as no-steps (as explained in section 6.10).

The first step (pun not intended) for doing this in SudoRules is to look for the W1-anti-backdoors (because a 1-step solution will necessarily eliminate one of the W1-anti-backdoors). As we have already found the 13 W1-anti-backdoors for this puzzle in section 6.11 (n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3 n8r3c4 n8r1c8), we now only have to check which of them can be eliminated by some pattern in CSP-Rules.

Here again, we can choose what type of elimination we are looking for, which leaves quite a large number of possibilities, considering all the rules that allow focused eliminations (which excludes t-whips). And this choice is almost totally independent of the choice made for T_0 in the search for T_0-anti-backdoors.

The next subsections will describe the general procedure for looking for 1-step solutions and two approaches for automating it or part of it.

6.12.1 The "manual" approach

After we have the list of T_0-anti-backdoors, how to check each of them individually as a basis for a one-step solution? Launch a new instance of CLIPS and select in the configuration file the rules you want to use (including the rules in T_0, for consistency purposes). Whatever rules you choose, the way of checking if a chosen candidate can be eliminated by them will be the same: first initialise the puzzle with its resolution state after T_0, then try-to-eliminate the candidate (say nrc):

```
(init-sukaku-list
   13456    1234     9        1234578  2378     12457    236      12348    123468
   8        1234     13456    12345    23       9        7        1234     12346
   134      7        134      12348    6        124      239      5        123489
   137      5        1378     12789    4        127      239      6        12379
   9        14       1467     12567    27       3        8        124      12457
```

```
   13467    1348     2        156789   789      1567     359      1349     134579
   34       6        348      2349     5        24       1        7        2389
   2        139      1357     3679     379      8        4        39       3569
   3457     3489     34578    234679   1        2467     23569    2389     235689
)
(try-to-eliminate-candidates nrc)
```

We can do this independently for each of the 13 W1-anti-backdoors and for 3 choices of rules:

1) Let's first assume we allow only bivalue-chains. Among the above 13 W1-anti-backdoors, we find that 7 lead to a single step solution (modulo steps in W1):

```
◊◊◊ n8r9c2: biv-chain[4]: c8n8{r9 r1} - c5n8{r1 r6} - c5n9{r6 r8} - b7n9{r8c2 r9c2} ==>
r9c2≠8, r9c8≠9
◊◊◊ n8r7c9: biv-chain[4]: r3n8{c9 c4} - c5n8{r1 r6} - c5n9{r6 r8} - r7n9{c4 c9} ==>
r7c9≠8, r3c9≠9
◊◊◊ n8r6c5: biv-chain[3]: c2n8{r6 r9} - b7n9{r9c2 r8c2} - c5n9{r8 r6} ==> r6c5≠8
◊◊◊ n9r4c4: biv-chain[3]: r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} ==> r4c4≠9
◊◊◊ n8r4c3: biv-chain[6]: r7n8{c3 c9} - r7n9{c9 c4} - c5n9{r8 r6} - c5n8{r6 r1} -
c8n8{r1 r9} - c2n8{r9 r6} ==> r4c3≠8, r9c2≠8
◊◊◊ n8r3c4: biv-chain[5]: r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} - c5n9{r8 r6} -
c5n8{r6 r1} ==> r3c4≠8, r6c5≠8
◊◊◊ n8r1c8: biv-chain[6]: r3n8{c9 c4} - r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} -
c5n9{r8 r6} - c5n8{r6 r1} ==> r1c8≠8, r1c9≠8, r3c4≠8
```

Notice that any of the two bivalue-chains[3] might have occurred in the simplest-first resolution path in S3Fin+W3, but SudoRules followed a different path that didn't use them.

2) Let's now assume we allow only z-chains (and the particular case of bivalue-chains). Among the above 13 W1-anti-backdoors, we find that 10 lead to a single step solution (modulo steps in W1):

```
◊◊◊ n8r9c2: biv-chain[4]: c8n8{r9 r1} - c5n8{r1 r6} - c5n9{r6 r8} - b7n9{r8c2 r9c2} ==>
r9c2≠8
◊◊◊ n9r8c5: z-chain[6]: c2n9{r8 r9} - c2n8{r9 r6} - r4n8{c3 c4} - c4n9{r4 r6} - c8n9{r6
r9} - r7n9{c9 .} ==> r8c5≠9
◊◊◊ n9r7c9: z-chain[4]: r7n8{c9 c3} - r4n8{c3 c4} - r4n9{c4 c7} - r3n9{c7 .} ==> r7c9≠9
◊◊◊ n8r7c9: biv-chain[4]: r3n8{c9 c4} - c5n8{r1 r6} - c5n9{r6 r8} - r7n9{c4 c9} ==>
r7c9≠8
◊◊◊ n9r6c8: z-chain[5]: c5n9{r6 r8} - c2n9{r8 r9} - c2n8{r9 r6} - r4n8{c3 c4} - r4n9{c4
.} ==> r6c8≠9
◊◊◊ n8r6c5: biv-chain[3]: c2n8{r6 r9} - b7n9{r9c2 r8c2} - c5n9{r8 r6} ==> r6c5≠8
◊◊◊ n9r4c4: biv-chain[3]: r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} ==> r4c4≠9
◊◊◊ n8r4c3: biv-chain[6]: r7n8{c3 c9} - r7n9{c9 c4} - c5n9{r8 r6} - c5n8{r6 r1} -
c8n8{r1 r9} - c2n8{r9 r6} ==> r4c3≠8
◊◊◊ n8r3c4: biv-chain[5]: r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} - c5n9{r8 r6} -
c5n8{r6 r1} ==> r3c4≠8
◊◊◊ n8r1c8: biv-chain[6]: r3n8{c9 c4} - r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} -
c5n9{r8 r6} - c5n8{r6 r1} ==> r1c8≠8
```

Notice that the previously obtained chains are unchanged (but some of their eliminations might *a priori* have been obtained by shorter z-chains). A larger change will appear with our third set of rules.

3) Let's finally assume we allow whips (plus the special cases of z-chains and bivalue-chains). Among the above 13 W1-anti-backdoors, we find that 11 lead to a single step solution (modulo steps in W1), i.e. only one more than without whips. But we can also see that some of the eliminations with whips are shorter than the previous eliminations with bivalue-chains or z-chains:

```
◊◊◊ n8r9c2: biv-chain[4]: c8n8{r9 r1} - c5n8{r1 r6} - c5n9{r6 r8} - b7n9{r8c2 r9c2} ==>
r9c2≠8, r9c8≠9
◊◊◊ n9r8c5: whip[5]: r7n9{c4 c9} - c8n9{r9 r6} - r4n9{c9 c4} - r4n8{c4 c3} - r7n8{c3 .}
==> r8c5≠9
◊◊◊ n9r8c2: whip[4]: c5n9{r8 r6} - c8n9{r6 r9} - c8n8{r9 r1} - c5n8{r1 .} ==> r8c2≠9
◊◊◊ n97c9: z-chain[4]: r7n8{c9 c3} - r4n8{c3 c4} - r4n9{c4 c7} - r3n9{c7 .} ==> r7c9≠9
◊◊◊ n8r7c9: biv-chain[4]: r3n8{c9 c4} - c5n8{r1 r6} - c5n9{r6 r8} - r7n9{c4 c9} ==>
r7c9≠8, r3c9≠9
◊◊◊ n9r6c8: whip[4]: r4n9{c9 c4} - r7n9{c4 c9} - r7n8{c9 c3} - r4n8{c3 .} ==> r6c8≠9
◊◊◊ n8r6c5: biv-chain[3]: c2n8{r6 r9} - b7n9{r9c2 r8c2} - c5n9{r8 r6} ==> r6c5≠8
◊◊◊ n9r4c4: biv-chain[3]: r4n8{c4 c3} - r7n8{c3 c9} - r7n9{c9 c4} ==> r4c4≠9
◊◊◊ n8r4c3: whip[5]: c2n8{r6 r9} - c8n8{r9 r1} - c5n8{r1 r6} - c5n9{r6 r8} - c2n9{r8 .}
==> r4c3≠8
◊◊◊ n8r3c4: whip[4]: c5n8{r1 r6} - c2n8{r6 r9} - c2n9{r9 r8} - c5n9{r8 .} ==> r3c4≠8
◊◊◊ n8r1c8: whip[4]: c5n8{r1 r6} - c2n8{r6 r9} - c2n9{r9 r8} - c5n9{r8 .} ==> r1c8≠8
```

We can consider that our search has been successful: instead of a long series of Subsets and chains of size no more than 3, we have several possibilities for a solution with a single step of size no more than 4, including two that require a single bivalue-chain[3] with a single elimination. Needless to say, this example is an extreme case. As for the other solutions, based on longer chains, we shall talk about them later. Notice also that we could have stopped this search for 1-step solutions as soon as we found one with a bivalue-chain[3], because the puzzle is in W3 and we know *a priori* there can be no simpler solution with whips or any simpler chains. Here, we continued only for illustrative purposes of the various types of solutions one can find. Finally, notice also that we could have restricted *a priori* (in the configuration file) the lengths of all the chains used in the search of 1-step solutions.

6.12.2 The semi-automated approach

Trying manually each anti-backdoor in turn for checking which of them give rise to a 1-step solution can be very tedious, especially if there are many of them. SudoRules has two functions for automating this part of the search for 1-step solutions: "***find-sudoku-1-steppers-among-cands***" and "***find-sukaku-1-steppers-among-cands***". Their respective syntax is:

```
(find-sudoku-1-steppers-among-cands ?sudoku-string $?cand-list)
```

```
(find-sukaku-1-steppers-among-cands ?sukaku-list $?cand-list)
```

where $?cand-list is the pre-calculated list of the T_0-anti-backdoors, as the list of 13 candidates in the previous section.

```
(find-sudoku-1-steppers-among-cands
   "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
   (list-of-nirjck-to-list-of-labels
      n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3
      n8r3c4 n8r1c8
   )
)
```

where we have used function "list-of-nirjck-to-list-of-labels" to turn this external representation to internal labels (see section 6.14.1 for technical details).

Currently, these two functions only execute the successive tries of each T-anti-backdoor, leaving to the user the job of scanning their output to find which of them have led to a solution.

6.12.3 The fully-automated approach

The semi-automated approach leaves much work to the user. The approach described in this section is much simpler for the user, but (for technical reasons) it also takes more computation time (which is in my opinion a very good trade for less user time). One can use a single function, ***find-sudoku-1-steppers-wrt-resolution-theory***, to launch all the necessary calculations. The syntax is:

```
(find-sudoku-1-steppers-wrt-resolution-theory ?T0 ?sudoku-string)
```

where $?T_0$ is the (simple) resolution theory whose rules one decides to count as no steps, and ?sudoku-string is the usual string representation of a puzzle. $?T_0$ can only be BRT, W1, S2, S3, S4 or S, as in section 6.10. I personally consider that only BRT and W1 really make sense: considering a Subset, even a Pair, as a no-step is nonsense.

What this function will do is:
- firstly, find the resolution state after the rules in $?T_0$ are applied,
- secondly, find the $?T_0$-anti-backdoors,
- finally, check each of them as a potential basis for a 1-step solution (modulo $?T_0$) with the current set of rules (chosen in the configuration file).

As before, currently, this function leaves to the user the job of scanning the output to find which of the $?T_0$-anti-backdoors have led to a solution.

A similar function "***find-sukaku-1-steppers-wrt-resolution-theory***" with syntax:
`(find-sukaku-1-steppers-wrt-resolution-theory ?T0 $?sukaku-list)`
does the same job, starting from a resolution state defined by $?sukaku-list.

Notice that these two automatic functions rely on the technical possibility of disabling and re-enabling rules, as described in the next section.

Considering that BRT and W1 are the most common values for $?T_0$ in these functions, there are two abbreviations for each of the above two functions: ***find-sudoku-1-steppers-wrt-BRT***, ***find-sudoku-1-steppers-wrt-W1***, ***find-sukaku-1-steppers-wrt-BRT*** and ***find-sukaku-1-steppers-wrt-W1***, with the same syntax except the first ($?T_0$) input variable.

6.13 Disabling and re-enabling rules in CSP-Rules

In the standard working of CSP-Rules, resolution rules are selected in the configuration file, which defines what will loaded by the system and made available to the user. However, for efficiency reason, that doesn't make all the rules immediately "enabled". The simplest-first strategy automatically enables them when needed and they remain enabled until some (reset) or (init-…) command is issued.

However, since the inception of SudoRules, all the rules in it have been coded in such a way that can prevent them from being enabled, even if loaded. Until recently, there had been no opportunity to effectively use this feature. The 8th update of CSP-Rules-V2.1 allows a more dynamical disabling and re-enabling of rules. Such technical details are transparent to the standard user and irrelevant in a Basic User Manual, but the next sections will take advantage of this to take care of several types of additional requirements about the number of steps in the resolution path.

Notice that the good way of restricting rules has always been and remains as described above, at load time via the configuration file. Any other ways imply an increase in computation time. However, there are cases when such an increase becomes acceptable by requiring less user time, as was the case in section 6.12.2 for function find-sudoku-1-steppers-wrt-resolution-theory.

6.13.1 Functions for disabling and re-enabling rules

Functions allowing disable and re-enable resolution rules are not intended for being applied as such by the user, but they will appear in all the functions dealing with the number of steps. An advanced user may find other ways of using them.

Function "***disable-rules-not-in-RT0***" has syntax:

```
(disable-rules-not-in-RT0 ?cont ?T0)
```

where ?cont is an integer for a context (0 for the starting context) and $?T_0$ is a resolution theory as in section 6.10. A context is the technical counterpart of a resolution state; as its content evolves with time, it represents a resolution path. After this function is called, all the generic rules that do not belong to $?T_0$ will be

disabled in context ?cont. In this function, $?T_0$ can only be BRT, W1, S2, S3, S4 or S, as in section 6.10. What's new in the 8th update is, resolution rules can be disabled even if they have already been enabled.

Function "***re-enable-rules-not-in-RT0***" with syntax:
```
(re-enable-rules-not-in-RT0 ?cont ?T0)
```
(where $?RT_0$ can again only be BRT, W1, S2, S3, S4 or S) does the converse.

Notice that:
- only generic rules and Subset rules can be disabled and re-enabled;
- disabling and re-enabling are global in a context, i.e. it is not currently possible to do it in a way that would depend on the length of chains (and no known reason for allowing it);
- these functions are essential to the full automation of 1-step, 2-step and fewer-steps solutions.

6.13.2 Functions for reconstructing a resolution path

When some rules are considered as no-step, as in section 6.10, most of the specific functions for dealing with the number of steps output both a full resolution path and a summary of their main steps (i.e. those not in RT0), as a list of candidates e.g. (n1r4c5 n8r7c3 n6r9c2). Knowing only the original puzzle, theory RT0 and this list, the following functions allow to reconstruct a full resolution path with the same eliminations. The arguments are as usual (notice that ?sukaku-list must first have been turned into a single argument, as usual).

```
(reconstruct-sudoku-resolution-path-wrt-RT0 ?RT0 ?sudoku-string
$?sequence-of-eliminations)
(reconstruct-sukaku-resolution-path-wrt-RT0 ?RT0 ?sukaku-list
$?sequence-of-eliminations)
```

As expected, they have abbreviations for the most current cases:

```
(reconstruct-sudoku-resolution-path-wrt-BRT ?sudoku-string $?sequence-
of-eliminations)
(reconstruct-sudoku-resolution-path-wrt-W1 ?sudoku-string $?sequence-
of-eliminations)
(reconstruct-sukaku-resolution-path-wrt-BRT ?sukaku-list $?sequence-
of-eliminations)
(reconstruct-sukaku-resolution-path-wrt-W1 ?sukaku-list $?sequence-of-
eliminations)
```

6.14 Two technical details

Before going further, we need to evoke two more "technical details". You don't really need to know them if you use only the fully automated functions for finding 1-step or 2-step solutions.

6.14.1 Internal vs external representations of candidates

The first "technical detail" is about the difference between internal and external representations of candidates in SudoRules (or CSP-Rules in general). The n8r9c2, n3r8c8, … appearing in the lists of anti-backdoors found by SudoRules use their external "nrc" representations – which have the advantage of being totally independent of any choice for their internal representations. However (because they need to be able to be combined with other functions), all the CSP-Rules functions work on internal representations. Function "***nirjck-to-label***" allows to transform a candidate from its external to its internal representations. More useful in practice, function "***list-of-nirjck-to-list-of-labels***" allows to transform a list of candidates in external representation to its internal representation. For instance, the command:

```
(list-of-nirjck-to-list-of-labels n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2
n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3 n8r3c4 n8r1c8)
```

will give the following result if no rules involving g-labels is activated:

```
(892 388 985 982 182 979 879 968 865 944 843 834 818)
```

But it will give a different result, if such rules are activated (because the internal representations are different).

As an obvious consequence, both functions must be used in the environment where their result will be applied.

This first "technical detail" allows to understand why function "find-sudoku-1-steppers-among-cands" must be used this way:

```
(find-sudoku-1-steppers-among-cands
   "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
   (list-of-nirjck-to-list-of-labels
         n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3
n8r3c4 n8r1c8
))
```

Another possibility, generally cleaner and particularly useful when the list of candidates to be considered is large, consists of first defining a global variable, e.g ?*my-cands*:

```
(defglobal ?*my-cands* = (create$
   (list-of-nirjck-to-list-of-labels
n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3
n8r3c4 n8r1c8
 )))
(find-sudoku-1-steppers-among-cands
   "..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1...."
   ?*my-cands*
)
```

6.14.2 List arguments in CLIPS

The second "technical detail" is about a CLIPS function being allowed only one list argument, which can only be the last one. The apparent problem in e.g. function "find-sukaku-1-steppers-wrt-resolution-theory" introduced in section 6.10 is, both sukaku-list and cand-list are lists. The (trivial) solution is to first turn the first argument (i.e. sukaku-list) into a single argument. The simplest way of doing this is to define a global variable, e.g. ?*RS* (beware of not using a name already used by the system), for representing the sukaku-list:

```
(defglobal ?*RS* = (create$
   13456    1234     9        1234578  2378     12457    236      12348    123468
   8        1234     13456    12345    23       9        7        1234     12346
   134      7        134      12348    6        124      239      5        123489
   137      5        1378     12789    4        127      239      6        12379
   9        14       1467     12567    27       3        8        124      12457
   13467    1348     2        156789   789      1567     359      1349     134579
   34       6        348      2349     5        24       1        7        2389
   2        139      1357     3679     379      8        4        39       3569
   3457     3489     34578    234679   1        2467     23569    2389     235689
))
```

As we're interested in W1-anti-backdoors and single-step solutions modulo whips[1], the resolution state chosen here is naturally the one obtained after applying Singles and whips[1] (indeed, there's none applicable for this puzzle). One can now write:

```
(find-sukaku-1-steppers-among-cands ?*RS* ?*my-cands*)
```

where I have also used the previous global variable ?*my-cands*. We could also have kept the list of candidates explicit, as in:

```
(find-sukaku-1-steppers-among-cands ?*RS*
   (list-of-nirjck-to-list-of-labels
n8r9c2 n3r8c8 n9r8c5 n9r8c2 n1r8c2 n9r7c9 n8r7c9 n9r6c8 n8r6c5 n9r4c4 n8r4c3
n8r3c4 n8r1c8
))
```

6.15 Looking for two-step solutions – 1st part (first example)

Consider now the following puzzle (from Denksport Magazine). It is moderately difficult, with a solution in gW4 or in W5 (or in Z7 if one wants to use only reversible chains). Let us assign the resolution state after Singles and whips[1] to a global variable ?*RS* and consider for instance the resolution path in W5:

```
+-------+-------+-------+
! 8 7 . ! . 9 6 ! 2 1 . !
! 2 5 1 ! . . 8 ! . 6 9 !
! 9 . . ! 1 2 . ! . . 7 !
+-------+-------+-------+
! . 3 . ! . . 2 ! . . 1 !
! . . . ! . . . ! . 2 . !
! 4 . 2 ! 8 . . ! . 3 . !
+-------+-------+-------+
! 3 . . ! 2 . 1 ! 6 . 8 !
! 6 2 . ! 9 . . ! 1 4 . !
! 1 . 5 ! 6 . . ! . 7 2 !
+-------+-------+-------+
```

```
(defglobal ?*RS* = (create$
   8        7        34       345      9        6        2        1        345
   2        5        1        347      347      8        34       6        9
   9        46       346      1        2        345      3458     58       7
   57       3        689      457      4567     2        45789    589      1
   57       1689     689      3457     134567   34579    45789    2        456
   4        169      2        8        1567     579      579      3        56
   3        49       47       2        457      1        6        59       8
   6        2        78       9        3578     357      1        4        35
   1        489      5        6        348      34       39       7        2
))
;;; 129 candidates
```

the standard simplest-first strategy gives the following resolution path in W5:

```
biv-chain[4]: r5n1{c5 c2} - c2n8{r5 r9} - r8c3{n8 n7} - r7n7{c3 c5} ==> r5c5≠7
z-chain[4]: c9n4{r5 r1} - r1n5{c9 c4} - r3c6{n5 n3} - r9c6{n3 .} ==> r5c6≠4
z-chain[4]: c9n3{r8 r1} - r1n5{c9 c4} - r3c6{n5 n4} - r9c6{n4 .} ==> r8c6≠3
whip[4]: c9n4{r5 r1} - r1n5{c9 c4} - c4n3{r1 r2} - r2c7{n3 .} ==> r5c4≠4
t-whip[5]: r9c7{n9 n3} - c9n3{r8 r1} - c3n3{r1 r3} - c6n3{r3 r5} - c6n9{r5 .} ==>
r6c7≠9
t-whip[4]: r6n9{c6 c2} - c3n9{r5 r7} - r7n7{c3 c5} - r8c6{n7 .} ==> r6c6≠5
finned-x-wing-in-rows: n5{r7 r6}{c5 c8} ==> r4c8≠5
t-whip[4]: r6n5{c9 c5} - c4n5{r5 r1} - c6n5{r3 r8} - c9n5{r8 .} ==> r5c7≠5, r4c7≠5
t-whip[4]: r8c6{n7 n5} - r7n5{c5 c8} - r3n5{c8 c7} - r6c7{n5 .} ==> r6c6≠7
naked-single ==> r6c6=9
naked-triplets-in-a-row: r5{c1 c4 c6}{n7 n5 n3} ==> r5c9≠5, r5c7≠7, r5c5≠5, r5c5≠3
whip[1]: b6n5{r6c9 .} ==> r6c5≠5
biv-chain[3]: r5n1{c5 c2} - r6c2{n1 n6} - b6n6{r6c9 r5c9} ==> r5c5≠6
biv-chain[4]: r9c7{n9 n3} - r8c9{n3 n5} - r6n5{c9 c7} - b6n7{r6c7 r4c7} ==> r4c7≠9
biv-chain[4]: r1n5{c4 c9} - c7n5{r3 r6} - c7n7{r6 r4} - r4c1{n7 n5} ==> r4c4≠5
biv-chain[4]: r4c4{n4 n7} - b6n7{r4c7 r6c7} - c7n5{r6 r3} - b2n5{r3c6 r1c4} ==> r1c4≠4
biv-chain[3]: r3c2{n6 n4} - r1n4{c3 c9} - r5c9{n4 n6} ==> r5c2≠6
biv-chain[3]: c3n3{r3 r1} - r1n4{c3 c9} - r2c7{n4 n3} ==> r3c7≠3
finned-x-wing-in-columns: n3{c7 c5}{r2 r9} ==> r9c6≠3
naked-single ==> r9c6=4
whip[1]: b2n4{r2c5 .} ==> r2c7≠4
singles to the end
```

Can one find a single-step solution for it? It has only 6 (BRT-, W1- or S-) anti-backdoors, but none of them can be eliminated by a whip of any length. (Exercise for the reader: check these claims, applying the techniques of section 6.12.)

As there's no one-step solution with whips, can one find a two-step one? The surprise appears when we look for anti-backdoor-pairs as a first step in the search for such solutions (because a two-step solution necessarily eliminates an anti-backdoor pair). While there were limited numbers of anti-backdoors, there are very large numbers of anti-backdoor-pairs: 615 BRT-anti-backdoor-pairs, 622 W1-anti-backdoor-pairs and 626 S-anti-backdoor-pairs. 85 of the 615 BRT ones give rise to a two-step solution, but many of these solutions involve long whips. Here is the simplest and only one in W5:

```
whip[5]: c3n3{r1 r3} - c7n3{r3 r9} - r9c6{n3 n4} - r3c6{n4 n5} - b3n5{r3c7 .} ==> r1c9≠3
singles ==> r8c9=3, r9c7=9, r7c8=5, r3c8=8, r4c8=9
whip[5]: r3n5{c7 c6} - r8c6{n5 n7} - r7c5{n7 n4} - c6n4{r9 r5} - c9n4{r5 .} ==> r1c9≠5
singles to the end
```

Notice that this solution makes sense with respect to the one obtained above by the simplest-first CSP-Rules strategy: it is also in W5. The large number of steps in the original solution is replaced by only two steps, without increasing the complexity of the hardest step (indeed increasing it only slightly form t-whip[5] to whip[5]).

Unfortunately, this is a very rare situation. As the next subsection will illustrate (and this is very far from being the worst case), most of the time, if one requires a 1- or 2- step solution, one has to accept much larger chains than in the original simplest-first solution, often to the point of total absurdity. Such solutions can also be computationally much heavier than the standard simplest-first solution.

6.16 Looking for two-step solutions – 2nd part (second example)

Consider the following (hard) puzzle (proposed by Urhegyi).

```
+-------+-------+-------+
! 1 . . ! 5 . . ! . . 2 !
! . 2 . ! 7 . . ! . 3 . !
! . . 3 ! . 6 . ! 1 . . !
+-------+-------+-------+
! . . . ! . . . ! . . 8 !
! . 4 . ! . 5 . ! . 9 . !
! 6 . 2 ! . . . ! . . . !
+-------+-------+-------+
! . . 6 ! . . . ! 4 . . !
! . 5 . ! . . 3 ! . 7 . !
! 8 . . ! . . 4 ! . . 9 !
+-------+-------+-------+
```

```
1..5....2.2.7...3...3.6.1.........8.4..5..9.6.2........6...4...5...3.7.8....4..9
SER = 8.5
```

Let's start solving it from its resolution state (call it again ?*RS*) after Singles and whips[1]:

```
(solve-sukaku-grid
            +-------------------+-------------------+-------------------+
            ! 1     6     4789  ! 5     3     89    ! 789   48    2     !
            ! 459   2     4589  ! 7     1489  189   ! 5689  3     456   !
            ! 4579  789   3     ! 2489  6     289   ! 1     458   457   !
            +-------------------+-------------------+-------------------+
            ! 3579  1379  1579  ! 12349 1249  12679 ! 2367  1246  8     !
            ! 37    4     178   ! 1238  5     12678 ! 2367  9     1367  !
            ! 6     13789 2     ! 13489 1489  1789  ! 357   145   13457 !
            +-------------------+-------------------+-------------------+
            ! 2379  1379  6     ! 1289  12789 5     ! 4     128   13    !
            ! 249   5     149   ! 12689 1289  3     ! 268   7     16    !
            ! 8     137   17    ! 126   127   4     ! 2356  1256  9     !
            +-------------------+-------------------+-------------------+
))
;;; 198 candidates
```

the standard simplest-first strategy gives the following resolution path in W4:

```
z-chain[3]: r9n5{c7 c8} - c8n6{r9 r4} - c8n2{r4 .} ==> r9c7≠2
biv-chain[4]: r1c8{n4 n8} - r1c6{n8 n9} - b3n9{r1c7 r2c7} - b3n6{r2c7 r2c9} ==> r2c9≠4
finned-x-wing-in-columns: n4{c9 c4}{r3 r6} ==> r6c5≠4
z-chain[4]: c9n5{r3 r6} - c9n4{r6 r3} - b3n7{r3c9 r1c7} - c7n9{r1 .} ==> r2c7≠5
t-whip[4]: c9n4{r6 r3} - r1c8{n4 n8} - r3c8{n8 n5} - c9n5{r3 .} ==> r6c9≠7, r6c9≠3,
r6c9≠1
biv-chain[2]: c9n7{r5 r3} - r1n7{c7 c3} ==> r5c3≠7
z-chain[4]: r9c3{n1 n7} - r1n7{c3 c7} - c9n7{r3 r5} - c9n1{r5 .} ==> r9c8≠1
whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r7c8≠8
hidden-single-in-a-block ==> r8c7=8
whip[1]: b9n2{r9c8 .} ==> r4c8≠2
t-whip[3]: c7n9{r2 r1} - r1c6{n9 n8} - r2n8{c6 .} ==> r2c3≠9
biv-chain[4]: r7n8{c4 c5} - b8n7{r7c5 r9c5} - r9c3{n7 n1} - r5c3{n1 n8} ==> r5c4≠8
finned-x-wing-in-rows: n8{r5 r2}{c3 c6} ==> r3c6≠8, r1c6≠8
stte
init-time = 0.2s, solve-time = 0.87s, total-time = 1.06s
```

Applying the approach described in section 6.12, we find that this puzzle has 6 W1-anti-backdoors (n1r9c3 n7r3c9 n6r2c7 n9r1c7 n8r1c6 n7r1c3) and that 3 of them give rise to 1-step solutions, but with rather long whips (compared to the maximum length of the above solution in W4, the best solution is in W7):

```
whip[7]: c7n9{r2 r1} - r1c6{n9 n8} - r2n8{c6 c3} - c3n5{r2 r4} - c3n9{r4 r8} - c3n4{r8
r1} - r1n7{c3 .} ==> r2c7≠6
stte
```

```
whip[8]: r1n7{c7 c3} - r1n4{c3 c8} - c9n4{r3 r6} - c9n5{r6 r2} - r3c8{n5 n8} - c2n8{r3
r6} - r5c3{n8 n1} - r9c3{n1 .} ==> r3c9≠7
stte

whip[9]: r1c6{n9 n8} - r1c8{n8 n4} - r1c3{n4 n7} - r9c3{n7 n1} - r5c3{n1 n8} - c2n8{r6
r3} - r3c8{n8 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r1c7≠9
stte
```

I leave it to the reader to decide for himself if such solutions look better to him than the original one in W4. For me, they do not: don't forget that complexity increases "exponentially" with the W rating. I don't mean the resolution path in W4 is perfect; it could probably be simplified, because the simplest-first strategy tends to include unnecessary steps. (However, apart from allowing to compute the rating at the same time as finding the solution, the good point of this strategy is, it shows several patterns a manual user can naturally find.)

At this point, the natural question is, can one find a 2-step solution with shorter chains? In the rest of this section, I shall state the results of the search for such solutions, which will allow in the next subsection to justify some choices in the recommended procedure.

In resolution state ?*RS* (obtained after Singles and whips[1]) , there are 198 candidates. An easy computation shows that 198*(198-1)/2 = 19,503 pairs will have to be checked as potential W1-anti-backdoor-pairs. The "-1" is for eliminating pairs composed of identical candidates and the "/2" is for eliminating redundancies due to symmetry: (cand1 cand2) vs (cand2 cand1).

After loading SudoRules with the following settings:

```
(bind ?*Anti-backdoor-pairs* TRUE)
(bind ?*Whips[1]* TRUE),
```

the following commands:

```
(defglobal ?*RS* = (create$
   1        6        4789     5        3        89       789      48       2
   459      2        4589     7        1489     189      5689     3        456
   4579     789      3        2489     6        289      1        458      457
   3579     1379     1579     12349    1249     12679    236      1246     8
   37       4        178      1238     5        12678    2367     9        1367
   6        13789    2        13489    1489     1789     357      145      13457
   2379     1379     6        1289     12789    5        4        128      13
   249      5        149      12689    1289     3        268      7        16
   8        137      17       126      127      4        2356     1256     9
))
(init-sukaku-list ?*RS*)
(find-sukaku-anti-backdoor-pairs ?*RS*)
```

will find 1972 W1-anti-backdoor-pairs, after 24m 11s of computation. This is a very long computation time (on an old Mac from 2012) and the result is a very large number of pairs to test for potential 2-step solutions. The next subsection will therefore explain a smarter procedure.

6.17 Looking for two-step solutions – 3rd part (the method)

Let us start by noticing that, for a 2-step solution eliminating candidates cand1 and cand2, not only does the (cand1 cand2) pair need to be an anti-backdoor-pair, but one must also be able to directly eliminate at least one of cand1 and cand2 from resolution state ?*RS* by the rules allowed for the 2-step solution.

SudoRules has a function for finding such candidates: "***find-erasable-candidates-sukaku-list***", with syntax:

```
(find-erasable-candidates-sukaku-list $?sukaku-list)
```

Notice that this function is based on focused eliminations (and it has therefore the same restrictions about t-whips) and that it doesn't change the resolution state. Here is how to use it in the example of the previous subsection and how to simplify the search for anti-backdoor-pairs by taking our goals into account since the beginning of the search.

◊ **First step**: decide the type of generic patterns you will accept in a 2-step solution (say bivalue-chains, z-chains, whips…; remember that t-whips are not allowed). Also decide the maximum length you will accept for these chains. Considering that this puzzle has a multi-step solution in W4 (and no one-step solution), it would be natural to put an upper bound of 5 or 6. For illustration purposes, we shall keep it much larger (8).

Load a first instance of CLIPS with the following settings (notice that, at this point, as whips are activated, it is not necessary to activate rules that are subsumed by them).

```
(bind ?*Whips* TRUE)
(bind ?*whips-max-length* 8)
```

- initialise it as before:

```
(defglobal ?*RS* = (create$
   1        6        4789     5        3        89       789      48       2
   459      2        4589     7        1489     189      5689     3        456
   4579     789      3        2489     6        289      1        458      457
   3579     1379     1579     12349    1249     12679    236      1246     8
   37       4        178      1238     5        12678    2367     9        1367
   6        13789    2        13489    1489     1789     357      145      13457
   2379     1379     6        1289     12789    5        4        128      13
   249      5        149      12689    1289     3        268      7        16
   8        137      17       126      127      4        2356     1256     9
))
```

```
(init-sukaku-list ?*RS*)
```

- and look for the candidates that can be eliminated:

```
(find-erasable-candidates-sukaku-list ?*RS*)
===> 34 candidates can be eliminated with the current set of rules:
(n4r1c3 n8r1c7 n8r1c8 n9r2c1 n9r2c3 n5r2c7 n6r2c7 n8r2c7 n4r2c9 n5r2c9 n4r3c9 n7r3c9
n2r4c8 n4r4c8 n7r5c3 n1r5c9 n6r5c9 n3r6c2 n4r6c4 n4r6c5 n7r6c7 n1r6c9 n3r6c9 n7r6c9
n8r7c8 n1r8c3 n8r8c4 n8r8c5 n2r8c7 n6r8c7 n6r8c9 n1r9c2 n2r9c7 n1r9c8)
computation time = 1m 11s
```

◊ **Second step**: load a second instance of SudoRules, with the following settings:

```
  (bind ?*Anti-Backdoor-pairs* TRUE)
  (bind ?*Whips[1]* TRUE)
```

- initialise it as before:

```
(defglobal ?*RS* = (create$
   1       6       4789    5       3       89      789     48      2
   459     2       4589    7       1489    189     5689    3       456
   4579    789     3       2489    6       289     1       458     457
   3579    1379    1579    12349   1249    12679   236     1246    8
   37      4       178     1238    5       12678   2367    9       1367
   6       13789   2       13489   1489    1789    357     145     13457
   2379    1379    6       1289    12789   5       4       128     13
   249     5       149     12689   1289    3       268     7       16
   8       137     17      126     127     4       2356    1256    9
))
(init-sukaku-list ?*RS*)
```

- define the list of erasable candidates:

```
(defglobal ?*erasable-cands* = (list-of-nirjck-to-list-of-labels (create$
   n4r1c3 n8r1c7 n8r1c8 n9r2c1 n9r2c3 n5r2c7 n6r2c7 n8r2c7 n4r2c9 n5r2c9 n4r3c9 n7r3c9
n2r4c8 n4r4c8 n7r5c3 n1r5c9 n6r5c9 n3r6c2 n4r6c4 n4r6c5 n7r6c7 n1r6c9 n3r6c9 n7r6c9
n8r7c8 n1r8c3 n8r8c4 n8r8c5 n2r8c7 n6r8c7 n6r8c9 n1r9c2 n2r9c7 n1r9c8
)))
```

- ask SudoRules to find the anti-backdoor-pairs that contain at least one of the above erasable candidates. Notice that the number of pairs now expected to be tried is: 34*(198-34) + 34*33/2 = 6,137. This remains a large number, but significantly smaller than the total number of pairs (19,306) and the computation time is also smaller (6m33s instead of 24m11s).

```
(find-anti-backdoor-pairs-with-one-cand-in-list ?*erasable-cands*)
475 W1-ANTI-BACKDOOR-PAIRS FOUND:
computation time = 6m 33s
```

We now have only 475 relevant anti-backdoor-pairs instead of the full set of 1972 ones.

- as the number of relevant pairs remains large and it would be difficult to copy and paste them to another instance of CLIPS, copy them to a file “W1-anti-backdoor-pairs-for-W8.txt” anywhere in your Documents folder and surround the whole list by

```
(defglobal ?*W1-anti-backdoor-pairs-for-W8* =
    (list-of-nirjck-to-list-of-labels (create$
```

and

```
)))
```

so that this file can later be loaded into CLIPS.

◊ **Third step**: load a third instance of CLIPS with the final settings for the rules you allow, e.g. (notice the global restriction on chain lengths):

```
(bind ?*Bivalue-Chains* TRUE)
(bind ?*z-Chains* TRUE)
(bind ?*Whips* TRUE)
(bind ?*all-chains-max-length* 8)
```

- initialise it as before:

```
(defglobal ?*RS* = (create$
   1        6        4789     5        3        89       789      48       2
   459      2        4589     7        1489     189      5689     3        456
   4579     789      3        2489     6        289      1        458      457
   3579     1379     1579     12349    1249     12679    236      1246     8
   37       4        178      1238     5        12678    2367     9        1367
   6        13789    2        13489    1489     1789     357      145      13457
   2379     1379     6        1289     12789    5        4        128      13
   249      5        149      12689    1289     3        268      7        16
   8        137      17       126      127      4        2356     1256     9    ))
(init-sukaku-list ?*RS*)
```

- initialise global variable ?*W1-anti-backdoor-pairs-for-W8* by loading the file where you have stored the previous results:
(load “xxx/W1-anti-backdoor-pairs-for-W8.txt”)

- and finally look for the possible two-step solutions:

```
(find-sukaku-2-steppers-among-pairs
    ?*RS*
    ?*W1-anti-backdoor-pairs-for-W8*
)
```

This will launch the search for 2-step solutions. When this is finished, you will have to scan the output in order to find which pairs led to a solution. In this example, we get the following four possibilities in W6 (among many more in W8):

```
whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r7c8≠8
```

```
hidden-single-in-a-block ==> r8c7=8
z-chain[5]: r1n4{c3 c8} - b3n8{r1c8 r3c8} - b1n8{r3c2 r2c3} - r5c3{n8 n1} - r9c3{n1 .}
==> r1c3≠7
stte

whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r7c8≠8
hidden-single-in-a-block ==> r8c7=8
whip[6]: r1c6{n9 n8} - r2n8{c6 c3} - c3n5{r2 r4} - c3n9{r4 r8} - c3n4{r8 r1} - r1n7{c3
.} ==> r1c7≠9
stte

whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r7c8≠8
hidden-single-in-a-block ==> r8c7=8
whip[6]: r1n7{c7 c3} - r9c3{n7 n1} - r5c3{n1 n8} - c2n8{r6 r3} - c8n8{r3 r1} - r1n4{c8
.} ==> r3c9≠7
stte

whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==> r2c7≠8
whip[6]: r1c6{n9 n8} - r2n8{c6 c3} - c3n5{r2 r4} - c3n9{r4 r8} - c3n4{r8 r1} - r1n7{c3
.} ==> r1c7≠9
stte
```

The total computation time for the three steps has been 11 minutes, admittedly rather long, but much shorter than if we had dealt with all the anti-backdoor-pairs. Notice that this time could have been drastically reduced if we had stuck to our idea that W6 was the maximum reasonable size for chains for this puzzle:

```
Total computation time = 3m 21s
```

Contrary to the standard simplest-first solution, the above procedure supposes some time investment from the user. A disadvantage is, if you want to try with longer chains, you have to re-start all from the beginning.

6.18 Looking for two-step solutions – 4th part (fully automated search)

SudoRules now has a way for fully automating the search for 2-step solutions. As the fully automated search is slower than the semi-automatic one (following manually the various steps described in section 6.16, with only the relevant rules loaded at each step) , we shall restrict the lengths of all the chains to 5.

◊ **Unique step**: in the configuration file, choose which set of rules you will allow (remember that t-whips are not allowed) and load SudoRules with them, e.g.:

```
(bind ?*Bivalue-Chains* TRUE)
(bind ?*z-Chains* TRUE)
(bind ?*Whips* TRUE)
(bind ?*all-chains-max-length* 5)
```

Use the new function ***find-sudoku-2-steppers-wrt-resolution-theory***, i.e. type the command:

```
(find-sudoku-2-steppers-wrt-resolution-theory
   W1
   "1..5....2.2.7...3...3.6.1..........8.4..5..9.6.2........6...4...5...3.7.8....4..9"
)
```

and prepare some tea while it's running. After 7 minutes or so, you will get the following output:

```
===> There is 1 2-step solution, based on the following pair:
n8r7c8 n7r1c3
```

It is then left to you to scan the full output (e.g. by searching it for the message "PUZZLE 0 IS SOLVED", in order to find the only 2-step solution in W5:

```
whip[4]: r1c8{n8 n4} - r3c8{n4 n5} - c9n5{r3 r6} - c9n4{r6 .} ==>
r7c8≠8
hidden-single-in-a-block ==> r8c7=8
z-chain[5]: r1n4{c3 c8} - b3n8{r1c8 r3c8} - b1n8{r3c2 r2c3} - r5c3{n8
n1} - r9c3{n1 .} ==> r1c3≠7
stte
```

Considering the puzzle is in W4, we have found a 2-step solution with a chain that is only slightly longer than necessary (z-chain[5]). The search can be considered successful.

It is interesting to check the intermediate results, when chain length is restricted to 5:

```
There remains 198 candidates after the rules in W1 have been applied.

===> CHECKING WHICH OF THE 198 CANDIDATES CAN BE ELIMINATED BY THE CURRENT SET OF
RULES:
erasable candidates computation time = 1m 55.08s
===> 15 candidates can be eliminated:
n8r1c7 n5r2c7 n8r2c7 n4r2c9 n4r4c8 n3r6c9 n7r6c9 n8r7c8 n1r8c3 n8r8c4 n8r8c5 n2r8c7
n6r8c7 n2r9c7 n1r9c8
```

Having 15 candidates that can be eliminated from the resolution state after W1 using the chosen set of rules implies that, even with the strict restriction on chain length, 15*(198-15) + 15*14/2 = 2,850 relevant pairs must be checked as potential anti-backdoor-pairs. Of these, only 107 are found to satisfy the conditions:

```
===> LOOKING FOR THE RELEVANT ANTI-BACKDOOR-PAIRS (BE PATIENT):
anti-backdoor-pairs computation time = 4m 22.1s
===> There are 107 W1-anti-backdoor-pairs for the current set of rules
```

```
===> CHECKING WHICH OF THE ANTI-BACKDOOR-PAIRS LEAD TO 2-STEP SOLUTIONS:
Total computation time = 7m 19.81s
There is one 2-step solution for the current set of rules, based on the following
pair:
n8r7c8 n7r1c3
```

Only one pair among the potentially relevant 2,850 ones is found to lead to a 2-step solution. As a result, it seems clear that such a 2-step solution is almost impossible to find by a human solver and any claim to the contrary should be considered with some grain of salt.

Moreover, suppose we had allowed slightly longer chains, e.g. max-length 6 instead of 5 (because we had no *a priori* reason to think 5 would be enough). Then there would be 20 candidates (instead of 15) erasable from the resolution state after W1. This would imply 20*(198-20) + 20*19/2 = 3,750 relevant pairs to check as potential anti-backdoor-pairs. Only 146 of them are effectively anti-backdoor-pairs. And there are finally 5 2-step solutions in W6.

```
===> CHECKING WHICH OF THE 198 CANDIDATES CAN BE ELIMINATED BY THE CURRENT SET OF
RULES:
erasable candidates computation time = 1m 49.83s
===> 20 candidates can be eliminated:
n8r1c7 n5r2c7 n8r2c7 n4r2c9 n2r4c8 n4r4c8 n7r5c3 n4r6c4 n4r6c5 n1r6c9 n3r6c9 n7r6c9
n8r7c8 n1r8c3 n8r8c4 n8r8c5 n2r8c7 n6r8c7 n2r9c7 n1r9c8

===> LOOKING FOR THE RELEVANT ANTI-BACKDOOR-PAIRS (BE PATIENT):
anti-backdoor-pairs computation time = 5m 57.38s
===> There are 146 W1-anti-backdoor-pairs for the current set of rules.
===> CHECKING WHICH OF THE ANTI-BACKDOOR-PAIRS LEAD TO 2-STEP SOLUTIONS:
Total computation time = 9m 11.66s
===> There are 5 2-step solutions for the current set of rules, based on the following
pairs:
n8r2c7 n9r1c7     n7r5c3 n6r2c7     n8r7c8 n7r1c3     n8r7c8 n9r1c7     n8r7c8 n7r3c9
```

Notice that the 4 additional solutions rely on whips[6] and are therefore significantly more complex.

Considering that BRT and W1 are the most common values for $?T_0$ in function ***find-sudoku-2-steppers-wrt-resolution-theory***, there are two abbreviations for these cases: ***find-sudoku-2-steppers-wrt-BRT***, ***find-sudoku-2-steppers-wrt-W1***. In parallel with the 1-step case, there is also a function ***find-sukaku-2-steppers-wrt-resolution-theory***, with two abbreviations: ***find-sukaku-2-steppers-wrt-BRT*** and ***find-sukaku-2-steppers-wrt-W1***. Of course, they take one argument less than their original counterparts.

6.19 Reducing the number of steps

This section is about a technique allowing to reduce the number of steps of a resolution path for a puzzle, with respect to a fixed, properly chosen resolution theory T. Notice the goal is not to "find the smallest number of steps" in T.

Given a family T of resolution rules, the CSP-Rules "standard" simplest-first strategy has the main advantage of computing the rating of a puzzle wrt to T at the same time as it solves it, but it tends to produce many steps, especially if various types of rules are activated at the same time as more general ones (e.g. bivalue-chains and/or z-chains at the same time as whips). This may be seen as a second advantage (as it shows many possibilities a manual player could use) or as a disadvantage, depending on one's goals.

Generally speaking, the simplest-first strategy is for a computer program and a human solver doesn't have to follow it (and in general he will not). But there is no reason why a computer program itself could not have different strategies for using the same rules.

There are two special cases that can (more or less) be excluded from our considerations here: 1-step and 2-step solutions. In such cases, the previous sections have shown that a systematic review of all the possible 1-step or 2-step paths is possible (though somewhat time consuming for 2-step solutions). Unfortunately, in case no such 1 or 2 step solution exists, a systematic exploration of all the resolution paths is impossible in practice, due to the combinatorial explosion of the number of paths to take into consideration. A totally different approach must be taken.

In such a case, any "fewer steps" approach has to choose each step "randomly" from the "most promising ones" among all the currently available ones. Obviously, for this to work, some heuristics for defining "promising steps" must be defined. The negative aspects of such an approach is, being based on random choices, the final result will depend on how many tries we make and we can never be sure that a shorter path doesn't exist.

The following remark may sound a tad bit sarcastic; it is only realistic. In order to find a resolution path with fewer steps, any algorithm or human solver has to explore many more steps than in the simplest-first strategy or in any natural "first-found-first-applied" strategy. The only thing that has fewer steps is the final printed result.

The idea of reducing the number of steps by successive random choices among those selected as "most promising" is due to François Defise.

Remark: one general problem about trying to "optimize" full resolution paths is, no rating of a full path has ever been defined in any rational way. Merely adding the lengths of each step would be total nonsense. As the complexity of a step increases

"exponentially" with its size (i.e. the length of a chain or the size of a Subset), it would be more rational to consider a function such as sum(a^size), where a > 1 is some parameter to be determined. Anyway, that would be totally useless, as it is impossible to use it in practice for any path optimisation.

The (much more modest) approach considered here almost totally avoids such problems. By first carefully selecting the allowed maximum length of all the chains (based on the simplest-first solution and the rating of the puzzle thus obtained), only the number of steps has to be taken into account.

6.19.1 The heuristics used for evaluating the candidates

The global strategy used by SudoRules to reduce the number of steps is given by the following pseudo-code (with an infamous GOTO):
1) Initialise the puzzle with the givens;
2) - apply all the rules in T_0, until none can be applied (as T_0 has the confluence property, this results in a uniquely defined resolution state RS)
- define step = 0
3) - step = step+1;
- find all the candidates in RS that could be deleted by the application of a single rule in T (this doesn't change RS);
- evaluate (according to the method defined below) each of these "erasable candidates" (this doesn't change RS);
- randomly choose one of these erasable candidates among those with the best score, say C (this doesn't change RS);
- apply the simplest rule in T that allows to eliminate C (as you can see, the simplest-first strategy is not completely discarded); this changes RS;
- apply all the rules in T_0 until none can be applied (same remark as before); this changes RS (unless the C score was only 1);
- if the puzzle is not solved, then GOTO 3, else report it as solved, report the value of variable "step" and stop.

This defines one try and produces a single resolution path in T (generally already shorter than the simplest-first one in T); in order to reduce more significantly the number of steps, several tries have to be made.

There remains to say how an erasable candidate C is evaluated in a resolution state RS. In the current implementation, it's very straightforward: the score of a candidate C is defined as the total number of deletions (including itself) its own deletion would imply in T_0. For ease of understanding, you can imagine this evaluation process is done for each candidate in a copy of the current resolution state RS and it starts by deleting C in this copy (score 1). Any subsequent deletion obtained by repeatedly applying a rule in T_0 adds 1 to the score. After the evaluation is done, the copy is merely discarded.

Notice that, if, in this evaluation process, a candidate is asserted as a decided value, it is counted as deleted as a candidate and the other deletions the new decided value implies by direct contradictions are also counted; as a result, there's no reason to make any special proviso for values that are decided in this evaluation process in the copy.

The obvious heuristic justification for defining the score this way is, if many candidates considered as 0-step are eliminated after one that has to be counted as a step, there will remain fewer candidates to eliminate, hopefully requiring fewer steps. Unfortunately, this hope is not always verified and this is the main limitation of the approach.

In his original algorithm, François Defise had a secondary criterion for selection between candidates with the same score: those with minimum number of candidates in one of the (i.e. the four rc, rn, vn and bn) CSP-Variables to which it belongs. I haven't retained such a criterion because my experience is, concentrating on (3D-) bivalue cells is rarely useful and I can see no reason why generalizing this to possibly more companion candidates would be very useful.

6.19.2 Example 1

Application of the above procedure may be more or less effective, depending on the puzzle, on the chosen theories T and T_0 and on luck. In any case, it takes much more time than a simplest-first solution, as it does a second level of "optimisation", "exponentially" more complex than the standard strategy.

Consider the following hard puzzle, Pisces-9.0 (SER = 9.0, W = 7). (See the "pisces2#523-9.0-W7.clp" file in the Examples folder for details not explicited here.)

```
+-------+-------+-------+
! 6 . . ! . . . ! . . 1 !
! . 2 . ! . . . ! . 4 . !
! . . 7 ! . . . ! 8 . . !
+-------+-------+-------+
! . . 4 ! 1 . 5 ! 9 . . !
! 3 8 1 ! 6 9 7 ! 4 2 5 !
! . . 9 ! 3 . 2 ! 6 . . !
+-------+-------+-------+
! . . 6 ! . . . ! 7 . . !
! . 9 . ! . . . ! . 8 . !
! 1 . . ! . . . ! . . 3 !
+-------+-------+-------+

6.......1.2.....4...7...8....41.59..381697425..93.26....6...7...9.....8.1.......3
29 givens, non-minimal, SER = 9.0
```

The resolution state after Singles and whips[1] is as follows:

```
+----------------------+----------------------+----------------------+
! 6      345    358    ! 24589  25     489    ! 235    7      1      !
! 589    2      358    ! 578    1567   168    ! 35     4      69     !
! 459    1      7      ! 245    2356   346    ! 8      569    269    !
+----------------------+----------------------+----------------------+
! 2      6      4      ! 1      8      5      ! 9      3      7      !
! 3      8      1      ! 6      9      7      ! 4      2      5      !
! 57     57     9      ! 3      4      2      ! 6      1      8      !
+----------------------+----------------------+----------------------+
! 458    345    6      ! 24589  1235   13489  ! 7      59     249    !
! 457    9      235    ! 2457   23567  346    ! 1      8      246    !
! 1      457    258    ! 245789 2567   4689   ! 25     569    3      !
+----------------------+----------------------+----------------------+
```

Using the simplest-first strategy, this puzzle has a solution in W7 with 31 non-W1 steps (with all the Subsets, Finned Fish, z-chains, t-whips and whips activated. Using the fewer-steps procedure, I obtained a solution with only 7 non-W1 steps in W8, after 5 tries (the first try had 15 steps); I considered it was good enough and I stopped after one more try. See the above-mentioned file for the details of all these calculations; you will notice in particular that they are very slow (about 10 minutes for each try, to be compared to the 18.3 seconds for the simplest-first solution – which is already slow for SudoRules, due to the inherent difficulty of this puzzle).

```
>>> whip[8]: c3n2{r8 r9} - r9c7{n2 n5} - c8n5{r9 r3} - r3c4{n5 n4} - r1n4{c6 c2} -
r9c2{n4 n7} - r9c5{n7 n6} - c8n6{r9 .} ==> r8c4≠2
>>> whip[7]: r9c7{n5 n2} - r9c3{n2 n8} - c1n8{r7 r2} - r2n9{c1 c9} - r7c9{n9 n4} -
r7c1{n4 n5} - b9n5{r7c8 .} ==> r9c5≠5
>>> whip[8]: r2n1{c5 c6} - r2n6{c6 c9} - c8n6{r3 r9} - r9c5{n6 n2} - r9c7{n2 n5} -
r9c3{n5 n8} - c1n8{r7 r2} - r2n9{c1 .} ==> r2c5≠7
hidden-single-in-a-block ==> r2c4=7
>>> whip-rc[8]: r2c9{n9 n6} - r3c9{n6 n2} - r8c9{n2 n4} - r8c4{n4 n5} - r3c4{n5 n4} -
r3c1{n4 n5} - r8c1{n5 n7} - r6c1{n7 .} ==> r3c8≠9
whip[1]: c8n9{r9 .} ==> r7c9≠9
>>> whip[8]: c9n4{r8 r7} - c1n4{r7 r3} - r3n9{c1 c9} - b3n2{r3c9 r1c7} - r1c5{n2 n5} -
r1c2{n5 n3} - r7c2{n3 n5} - r8n5{c1 .} ==> r8c4≠4
naked-single ==> r8c4=5
>>> whip[7]: b9n6{r9c8 r8c9} - c9n4{r8 r7} - c9n2{r7 r3} - r3c4{n2 n4} - r3c6{n4 n3} -
r8c6{n3 n4} - c1n4{r8 .} ==> r3c8≠6
singles ==> r3c8=5, r2c7=3, r1c7=2, r1c5=5, r9c7=5, r7c8=9, r9c8=6, r2c3=5
>>> whip-rc[5]: r6c1{n5 n7} - r8c1{n7 n4} - r7c2{n4 n3} - r8c3{n3 n2} - r8c9{n2 .} ==>
r6c2≠5
stte
```

6.19.3 Example 2

Here is another example for a much simpler puzzle due to Mith (see details in the "Tatooine-Tosche-Station.clp" file), using only Subsets:

```
...........12....3.4..5..6..5..6..7...38....2.....7.............23...98.6..7..5.
SER = 4.0
```

```
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . 1 ! 2 . . ! . . 3 !
! . 4 . ! . 5 . ! . 6 . !
+-------+-------+-------+
! . 5 . ! . 6 . ! . 7 . !
! . . 3 ! 8 . . ! . . 2 !
! . . . ! . . 7 ! . . . !
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . 2 ! 3 . . ! . 9 8 !
! . 6 . ! . 7 . ! . 5 . !
+-------+-------+-------+
```

The resolution state after Singles and whips[1] is as follows:

```
+-------------------------+-------------------------+-------------------------+
! 2356789 23789   56789   ! 14679   13489   134689  ! 1245789 1248    14579   !
! 56789   789     1       ! 2       489     4689    ! 45789   48      3       !
! 23789   4       789     ! 179     5       1389    ! 12789   6       179     !
+-------------------------+-------------------------+-------------------------+
! 12489   5       489     ! 149     6       12349   ! 13489   7       149     !
! 14679   179     3       ! 8       149     1459    ! 14569   14      2       !
! 124689  1289    4689    ! 1459    12349   7       ! 1345689 1348    14569   !
+-------------------------+-------------------------+-------------------------+
! 1345789 13789   45789   ! 14569   12489   1245689 ! 123467  1234    1467    !
! 1457    17      2       ! 3       14      1456    ! 1467    9       8       !
! 13489   6       489     ! 149     7       12489   ! 1234    5       14      !
+-------------------------+-------------------------+-------------------------+
265 candidates, 2088 csp-links and 2088 links. Density = 5.97%
```

Using the simplest-first strategy, there is a solution with 16 non-W1 steps using only Subsets and Finned Fish:

```
hidden-pairs-in-a-block: b5{n2 n3}{r4c6 r6c5} ==> r6c5≠9, r6c5≠4, r6c5≠1, r4c6≠9,
r4c6≠4, r4c6≠1
swordfish-in-columns: n7{c3 c4 c9}{r7 r3 r1} ==> r7c7≠7, r7c2≠7, r7c1≠7, r3c7≠7,
r3c1≠7, r1c7≠7, r1c2≠7, r1c1≠7
swordfish-in-columns: n3{c2 c5 c8}{r7 r1 r6} ==> r7c7≠3, r7c1≠3, r6c7≠3, r1c6≠3, r1c1≠3
swordfish-in-columns: n6{c3 c4 c9}{r6 r1 r7} ==> r7c7≠6, r7c6≠6, r6c7≠6, r6c1≠6,
r1c6≠6, r1c1≠6
hidden-pairs-in-a-block: b9{n6 n7}{r7c9 r8c7} ==> r8c7≠4, r8c7≠1, r7c9≠4, r7c9≠1
swordfish-in-rows: n2{r3 r4 r9}{c7 c1 c6} ==> r7c7≠2, r7c6≠2, r6c1≠2, r1c7≠2, r1c1≠2
naked-pairs-in-a-block: b9{r7c7 r9c9}{n1 n4} ==> r9c7≠4, r9c7≠1, r7c8≠4, r7c8≠1
hidden-pairs-in-a-block: b1{n2 n3}{r1c2 r3c1} ==> r3c1≠9, r3c1≠8, r1c2≠9, r1c2≠8
swordfish-in-rows: n5{r2 r5 r8}{c1 c7 c6} ==> r7c6≠5, r7c1≠5, r6c7≠5, r1c7≠5, r1c1≠5
hidden-pairs-in-a-block: b1{n5 n6}{r1c3 r2c1} ==> r2c1≠9, r2c1≠8, r2c1≠7, r1c3≠9,
r1c3≠8, r1c3≠7
hidden-pairs-in-a-block: b6{n5 n6}{r5c7 r6c9} ==> r6c9≠9, r6c9≠4, r6c9≠1, r5c7≠9,
r5c7≠4, r5c7≠1
hidden-pairs-in-a-block: b8{n5 n6}{r7c4 r8c6} ==> r8c6≠4, r8c6≠1, r7c4≠9, r7c4≠4,
r7c4≠1
```

```
hidden-triplets-in-a-column: c1{n5 n6 n7}{r8 r2 r5} ==> r8c1≠4, r8c1≠1, r5c1≠9, r5c1≠4,
r5c1≠1
singles ==> r8c5=4, r8c2=1
hidden-pairs-in-a-block: b7{n5 n7}{r7c3 r8c1} ==> r7c3≠9, r7c3≠8, r7c3≠4
finned-x-wing-in-rows: n4{r5 r2}{c6 c8} ==> r1c8≠4
hidden-triplets-in-a-row: r1{n5 n6 n7}{c9 c3 c4} ==> r1c9≠9, r1c9≠4, r1c9≠1, r1c4≠9,
r1c4≠4, r1c4≠1
whip[1]: c4n4{r6 .} ==> r5c6≠4
stte
```

After a single try of the fewer steps function, I obtained a solution with only 8 steps; I didn't make more tries with Subsets only or tries including whips, because it is difficult to beat Subsets in the number of candidates they eliminate.

```
=====> STEP #1
swordfish-in-columns: n7{c3 c4 c9}{r7 r3 r1} ==> r1c2≠7, r7c7≠7, r7c2≠7, r7c1≠7,
r3c7≠7, r3c1≠7, r1c7≠7, r1c1≠7
=====> STEP #2
swordfish-in-columns: n6{c3 c4 c9}{r6 r1 r7} ==> r7c7≠6, r7c6≠6, r6c7≠6, r6c1≠6,
r1c6≠6, r1c1≠6
=====> STEP #3
hidden-pairs-in-a-block: b5{n2 n3}{r4c6 r6c5} ==> r4c6≠4, r6c5≠9, r6c5≠4, r6c5≠1,
r4c6≠9, r4c6≠1
=====> STEP #4
swordfish-in-rows: n5{r2 r5 r8}{c1 c7 c6} ==> r1c1≠5, r7c6≠5, r7c1≠5, r6c7≠5, r1c7≠5
=====> STEP #5
hidden-triplets-in-a-row: r1{n5 n6 n7}{c9 c3 c4} ==> r1c9≠1, r1c9≠9, r1c9≠4, r1c4≠9,
r1c4≠4, r1c4≠1, r1c3≠9, r1c3≠8
=====> STEP #6
hidden-triplets-in-a-column: c7{n5 n6 n7}{r2 r5 r8} ==> r2c7≠4, r8c7≠4, r8c7≠1, r5c7≠9,
r5c7≠4, r5c7≠1, r2c7≠9, r2c7≠8
=====> STEP #7
hidden-triplets-in-a-column: c1{n5 n6 n7}{r8 r2 r5} ==> r5c1≠4, r8c1≠4, r8c1≠1, r5c1≠9,
r5c1≠1, r2c1≠9, r2c1≠8
whip[1]: r8n4{c6 .} ==> r7c4≠4, r7c5≠4, r7c6≠4, r9c4≠4, r9c6≠4
whip[1]: c4n4{r6 .} ==> r5c5≠4, r5c6≠4
hidden-single-in-a-row ==> r5c8=4
naked-single ==> r2c8=8
hidden-single-in-a-block ==> r1c7=4
whip[1]: b3n9{r3c9 .} ==> r3c1≠9, r3c3≠9, r3c4≠9, r3c6≠9
=====> STEP #8
hidden-pairs-in-a-column: c6{n4 n6}{r2 r8} ==> r2c6≠9, r8c6≠5, r8c6≠1
stte
```

6.19.4 Syntax

The most general function for launching the above calculations is:

```
(solve-sudoku-with-fewer-steps-wrt-resolution-theory ?T0 ?sudoku-
string)
```

It has two specialized versions for the most natural resolution theories $?T_0$:

```
(solve-sudoku-with-fewer-steps-wrt-BRT ?sudoku-string)
(solve-sudoku-with-fewer-steps-wrt-W1 ?sudoku-string)
```

There are corresponding functions for puzzles given as a sukaku list:

```
(solve-sukaku-with-fewer-steps-wrt-resolution-theory ?T0 $?sukaku-
list)
(solve-sukaku-with-fewer-steps-wrt-BRT $?sukaku-list)
(solve-sukaku-with-fewer-steps-wrt-W1 $?sukaku-list)
```

Examples for the first puzzle:

```
(solve-sudoku-with-fewer-steps-wrt-W1
  "6.......1.2.....4...7...8....41.59..381697425..93.26....6...7...9.....8.1.......3")
```

Or, starting e.g. from the resolution state after Singles and whips[1]:

```
(solve-sukaku-with-fewer-steps-wrt-W1
    6      345    358     24589  25     489     235    7      1
    589    2      358     578    1567   168     35     4      69
    459    1      7       245    2356   346     8      569    269
    2      6      4       1      8      5       9      3      7
    3      8      1       6      9      7       4      2      5
    57     57     9       3      4      2       6      1      8
    458    345    6       24589  1235   13489   7      59     249
    457    9      235     2457   23567  346     1      8      246
    1      457    258     245789 2567   4689    25     569    3
)
```

In order to make several tries, you have to iterate the previous function or to use the function that does it for you, i.e. :

```
(solve-ntimes-sudoku-with-fewer-steps-wrt-resolution-theory
   ?ntimes ?RT0 ?sudoku-string)
```

or any of the abbreviations you can imagine for the ?RT_0 = BRT or W1 cases, and for the corresponding Sukaku cases.

There is also a smarter function that iterates a specified number of times, while avoiding to pursue resolution paths longer than any path previously obtained, with syntax:

```
(smart-solve-ntimes-sudoku-with-fewer-steps-wrt-resolution-theory
   ?ntimes ?RT0 ?sudoku-string)
```

As expected, ?ntimes is the number of times you want it to iterate. At the end, it will output the best resolution path it has found. Here, “best” means with the fewest steps and with the smallest max-chain-length among those with the same number of steps.

There are also associated functions for the most usual ?RT0 cases, with syntax:

```
(smart-solve-ntimes-sudoku-with-fewer-steps-wrt-BRT
       ?ntimes ?sudoku-string)
(smart-solve-ntimes-sudoku-with-fewer-steps-wrt-W1
       ?ntimes ?sudoku-string)
```

6.20 Forcing T&E

(forthcoming – maybe. I'm still not convinced that this T&E-ish solving method is worth publishing)

7. LatinRules

LatinRules is the pattern-based solver of Latin Square puzzles based on CSP-Rules. Latin Square puzzles are played on a square grid and grid size is unrestricted in general.

The 6th update of CSP-Rules-V2.1 (June 2021) added the Pandiagonal variant of Latin Squares. This variant is very unlikely to ever become a popular logic puzzle, but I finally chose to add it to LatinRules because it illustrates an easy way of defining variants of a CSP already defined in CSP-Rules.

The Pandiagonal variant can be selected in the "General application-specific choices" part of the configuration file, by setting variable ?*Pandiagonal* to TRUE (it is FALSE by default), in which case grid size may not be divisible by 2 or 3.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; General application-specific choices
;;; Definition of grid size and related parameters
;;; Choice of variants
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; By default, grid size is 9, as in Sudoku
;;; If needed, change grid size here (grid-size can be any integer)

; (bind ?*grid-size* 13) ;                                 <<<<<<<<<<<<<<<<<<<<<<<<<

;;; By default, Latin Squares is the classical version. But Pandiagonal constraints
can be added.
;;; Notice that, in this case, ?*grid-size* may not be divisible by 2 or 3.
; (bind ?*Pandiagonal* TRUE);                              <<<<<<<<<<<<<<<<<<<<<<<<<
```

7.1 Latin Squares

In Latin Squares, with respect to Sudoku, the absence of constraints related to blocks implies there are no Subset rules for blocks, no whips[1], no g-labels, no g-whips or g-braids, no g-Subsets (such as Finned Fish), no g-anything. For Sudoku players, the interactions between blocks and rows or columns (e.g. whips[1]) make an important part of the excitement. Blocks are what makes Sudoku a unique logic game, with unmatched success.

LatinRules is a stripped-down version of SudoRules (eliminating any reference to blocks and g-labels) and it hasn't given rise to similar large-scale studies. It has the same basic user functions as Sudoku, with the same syntax, i.e. in essence functions "***init***", "***solve***", "***solve-n-grids-after-first-p-from-text-file***", "***print-current-resolution-state***" and "***solve-list***" (which can take as input the result of the previous function or any resolution state in the same list format as in Sudoku). There is no solve-xxx-grid or init-xxx-grid function, as no blocks need to be separated by vertical or horizontal lines.

I keep LatinRules separate from SudoRules mainly for practical reasons: this way, it is easier to code variants for each, without needing to put (Latin Squares vs Sudoku) conditions everywhere blocks are mentioned.

7.2 Pandiagonal Latin Squares

Pandiagonal Latin Squares is a variant of Latin Squares where the "only one" constraint for each number is required to be true not only for rows and columns but also for all the wrap-around diagonals and anti-diagonals (see 7.2.1 for precise definitions).

These are very strong constraints and it can be shown that there exists no Pandiagonal Latin Square when grid size is divisible by 2 or 3 ([Bartlett 2012]). Moreover, there are only very few non essentially equivalent ones if grid size is less than 13 ([Atkin & al. 1983]): for n = 5, 7 and 11 there are exactly (up to isomorphism) n-3 complete grids and they are cyclic. For n = 13 there are 10 (still n-3) non essentially equivalent complete grids, 2 cyclic, 6 semi-cyclic and 1 acyclic.

Does this imply that non trivial Pandiagonal Latin Square puzzles should be quite large (n > 13)? I don't think so, as far as solving is concerned. The usual requirement of a solution in the pattern-based approach to CSP solving developed in [PBCS] and implemented in CSP-Rules requires that each step of the resolution path be based on a precise pattern. One may explicitly add to this requirement that knowledge about cyclicity of the completed grids be not used. In this view, small sized grids remain interesting from a solver's perspective.

Pandiagonal Latin Squares can be considered as a variant of Sudoku, where the block constraints are replaced by wrap-around diagonal and anti-diagonal constraints. This is justified from an abstract point of view and is totally in line with the approach taken in CSP-Rules. However, when one considers puzzles from a human solver point of view, the two types of puzzles are very different in practice. Block constraints are immediately visible to the player. On the contrary, due to the wrap-around, diagonal and anti-diagonal constraints are very unfriendly in practice, unless some graphical technique allows to visualize them better, e.g. different colours for diagonals and different background pattern for anti-diagonals.

In theory, Pandiagonal constraints could also be added to Sudoku(25) [25 being the smallest integer for which the two constraints are consistent], resulting in some very large and messy puzzles.

Pandiagonal Latin Squares have been the topic of recent interest in the New Sudoku Player's Forum, thanks to several participants (Mathimagics, 1to9only, …). Wouldn't they have introduced it to the forum and generated the first example puzzles of various sizes, this new part of LatinRules wouldn't exist. I owe many thanks in particular to "1to9only" for generating a large collection (> 1000) of 7×7 puzzles.

7.2.1 Wrap-around diagonals and associated CSP-Variables

The definition of wrap-around diagonals is best understood if you consider that the square grid is wrapped around a torus (a tyre), the upper and lower sides being stuck together and the left and right sides being also stuck together, edge to edge. For grid size n, a wrap-around row draws a horizontal circle, a wrap-around column draws a vertical circle and a wrap-around diagonal or anti-diagonal draws a full ellipse on the torus. Each of them has exactly n cells and each digit must appear once and only once in each of them.

In LatinRules, wrap-around diagonals (respectively anti-diagonals) are designated by the letter d (respectively a) and are numbered from 1 to n. To be more specific, suppose grid size is 7:

- d_1 starts from cell r_1c_1 and it moves "upwards and rightwards" (like a /) to cells r_7c_2, r_6c_3… until it finishes on cell r_2c_7;
- d_2 starts from cell r_2c_1 and it moves "upwards and rightwards" to cells r_1c_2, r_7c_3… until it finishes on cell r_3c_7;
- d_7 starts from cell r_7c_1 and it moves "upwards and rightwards" to cells r_6c_2, r_5c_3… until it finishes on cell r_1c_7;

Similarly,

- a_1 starts from cell r_1c_1 and it moves "downwards and leftwards" (like a \) to cells r_2c_7, r_3c_6… until it finishes on cell r_7c_2;
- a_7 starts from cell r_1c_7 and it moves "downwards and leftwards" to cells r_2c_6, r_3c_5… until it finishes on cell r_7c_1.

It is easy to see that wrap-around diagonals and anti-diagonals define a second system of coordinates on the grid. Indeed, any pair of different terms taken among row, column, diagonal and anti-diagonal defines a system of coordinates.

The CSP-Variables for Pandiagonal Latin Squares are the same as for Latin Squares (the rc, rn and cn Variables), plus similar ones related to wrap-around diagonals and anti-diagonals (the dn and an Variables).

7.2.2 g-labels and Subsets

Pandiagonal Latin Squares have whips[1] and therefore g-labels, g-whips and g-braids. g-labels are of four types (rn, cn, dn and an) and are made of three candidates with the same digit value, with cells respectively in the same row, column, diagonal or anti-diagonal. A g-label G of type xn has its three labels at the intersection of its CSP-Variable with a combination of three CSP-Variables, each of a different type among the other three types and intersecting at some point; this point will be g-linked to G. However, due to combinatorial explosion, g-labels are not coded in LatinRules and the corresponding chains are not available.

Pandiagonal Latin Squares also have Subsets – lots of. As in Latin Squares, Naked and Hidden Subsets rely on a single "line", but a "line" can now be a row, a column, a diagonal or an anti-diagonal. For a fixed grid size, this multiplies by two the number of possibilities for Naked and Hidden Subsets.

The real computational problem arises with Super Hidden Subsets (Fish). Both the base sets and the transversal sets (or cover sets) of a Fish can *a priori* be made of CSP-Variables of different types. Even when one doesn't allow mixing of types neither in the base set nor in the transversal set, there remains twelve (instead of two in Latin Squares) different "homogeneous" combinations of possible Fish patterns of fixed size. Notice that all these homogeneous combinations are coded in LatinRules, up to Fishes of size 4. (it was straightforward to code them by permuting words "row", "column", "diagonal", "anti-diagonal" and associated variables in the original Fish rules of LatinRules).

For puzzles with large grid sizes, this implies a potential combinatorial explosion of the number of Fish patterns to be tested. This is why the Latin Rules configuration file recommends not to activate Subsets of size 4 in general when ?*Pandiagonal* is TRUE and grid size is greater than 11.

7.2.3 Branching factors

One (though not the only) important number in the practical application of the generic chain rules of CSP-Rules is the branching factor of the CSP, i.e. the mean number of candidates that are linked to a given one. For any type xxx of chain, the square of this factor gives a rough estimate of how many partial-xxx-chains[n+1] one will have to consider for each partial-xxx-chain[n]. (Of course, for a better estimate, it has to be combined with the number of actual candidates.)

The branching factors are easy to compute for grid size n: $3n-1-2\sqrt{n}$ in Sudoku and $4(n-1)$ in Pandiagonal Latin Squares – and they are constant (the same for each

candidate). The following tables allow a comparison between the two types of puzzles:

```
Sudoku:
grid size     branching factor
9             20
16            39
25            64

Pandiagonal Latin Squares:
grid size     branching factor
7             24
11            40
13            48
17            64
```

It appears that Pandiags(7) is the closest to standard Sudoku(9). This is also why it is important not to discard the small grid sizes from our considerations. Grid size 13 is closest to Sudoku(16 or 25), which generally implies lots of repetitive and boring easy eliminations before things eventually get exciting.

7.2.4 Example 1: whips[1], generic chains and Naked/Hidden Subsets

Consider the following 7×7 puzzle (created by 1to9only):

```
. 1 . . . . .
4 . . . . . .
. . . 3 . . 5
. . . . . . .
. . . 6 . . .
. . . 2 . . .
. . . . . . .

.1.....4.........3..5..........6......2..........
```

The first point to notice here is the density (after Singles), much larger than in Sudoku; it results from the existence of more (in the mean) direct contradiction links between candidate pairs:

```
Resolution state after Singles:
2357   1      234567 457    23467  4567   237
4      23567  2567   157    12567  12367  137
167    27     12467  3      12467  1247   5
12367  34567  1257   1457   2457   367    123467
12357  247    13457  6      37     2457   12347
1567   34567  1347   2      1357   134567 467
2567   23457  367    147    134567 12357  12467

181 candidates, 1563 csp-links and 1563 links. Density = 9.59%
```

The first part of the resolution path provides an interesting example of all the types of whips[1]; study them and check them carefully on the grid if you want to understand whips[1] in Pandiagonal Latin Squares; this requires a lot of gymnastics with diagonals and anti-diagonals.

```
whip[1]: d1n5{r7c2 .} ==> r7c1≠5
whip[1]: r7n5{c6 .} ==> r4c2≠5
whip[1]: d2n6{r4c6 .} ==> r4c7≠6, r1c3≠6
whip[1]: r1n6{c6 .} ==> r2c6≠6, r4c2≠6, r7c5≠6, r2c5≠6
whip[1]: c5n6{r3 .} ==> r3c3≠6
whip[1]: r3n6{c5 .} ==> r6c1≠6, r7c1≠6
whip[1]: a2n6{r6c7 .} ==> r6c6≠6
whip[1]: a2n4{r4c5 .} ==> r4c7≠4
whip[1]: c7n4{r7 .} ==> r6c6≠4
whip[1]: d1n1{r2c7 .} ==> r2c6≠1
whip[1]: c6n1{r7 .} ==> r3c3≠1
whip[1]: a7n3{r1c7 .} ==> r1c5≠3
whip[1]: c5n3{r7 .} ==> r6c6≠3

Resolution state RS1 after Singles and whips[1]: <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
2357  1     23457 457   2467  4567  237
4     23567 2567  157   1257  237   137
167   27    247   3     12467 1247  5
12367 347   1257  1457  2457  367   1237
12357 247   13457 6     37    2457  12347
157   34567 1347  2     1357  157   467
27    23457 367   147   13457 12357 12467
163 candidates.
```

If Subsets are not activated in the configuration file, a typical sequence of short chains leads to the solution:

```
z-chain[2]: r1n3{c3 c7} - r6n3{c5 .} ==> r7c2≠3
whip[1]: c2n3{r6 .} ==> r4c7≠3
whip[1]: c7n3{r5 .} ==> r5c3≠3
z-chain[2]: c2n5{r2 r6} - d6n5{r2c5 .} ==> r1c3≠5
whip[1]: r1n5{c6 .} ==> r6c6≠5
whip[1]: d4n5{r7c5 .} ==> r2c5≠5
z-chain[2]: d2n7{r4c6 r5c5} - c4n7{r4 .} ==> r1c3≠7
z-chain[2]: r3n4{c3 c6} - a2n4{r5c6 .} ==> r1c5≠4
whip[1]: c5n4{r7 .} ==> r7c2≠4
whip[1]: c2n4{r6 .} ==> r5c3≠4
z-chain[2]: r2n1{c5 c7} - c3n1{r5 .} ==> r6c1≠1
whip[1]: d6n1{r7c7 .} ==> r4c7≠1
whip[1]: c7n1{r7 .} ==> r7c5≠1
whip[2]: d2n7{r4c6 r7c3} - a7n7{r4c3 .} ==> r3c5≠7
z-chain[2]: r3n7{c3 c6} - c4n7{r1 .} ==> r2c2≠7
whip[2]: d2n7{r7c3 r4c6} - r2n7{c6 .} ==> r7c5≠7
z-chain[3]: r3n7{c6 c2} - r1n7{c7 c6} - d2n7{r4c6 .} ==> r5c1≠7
z-chain[3]: r3n7{c6 c1} - a3n7{r6c1 r5c7} - d2n7{r5c5 .} ==> r7c6≠7
z-chain[3]: r6c1{n7 n5} - r7c2{n5 n2} - r3c2{n2 .} ==> r6c2≠7
z-chain[2]: c2n7{r5 r7} - c4n7{r2 .} ==> r4c1≠7
```

```
whip[2]: c1n7{r7 r3} - c2n7{r4 .} ==> r7c7≠7
;;; Resolution state RS2 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
z-chain[3]: c2n7{r7 r4} - a2n7{r4c5 r7c1} - d2n7{r7c3 .} ==> r5c7≠7
z-chain[3]: r6c1{n7 n5} - r7c2{n5 n2} - r3c2{n2 .} ==> r5c2≠7
z-chain[2]: c2n7{r4 r7} - d2n7{r4c6 .} ==> r3c3≠7
z-chain[2]: r3n7{c2 c6} - d6n7{r1c6 .} ==> r6c5≠7
z-chain[2]: a1n7{r5c5 r6c6} - c2n7{r3 .} ==> r4c5≠7
z-chain[2]: a2n7{r6c7 r7c1} - r3n7{c1 .} ==> r6c6≠7
stte
init-time = 0.05s, solve-time = 2.15s, total-time = 2.21s
```

If Subsets and chains are activated, an alternative path after Resolution state RS2 allows to see a Naked Triplets in a diagonal, where (diagonal, anti-diagonal) coordinates are used by the current output function:

```
naked-triplets-in-a-diagonal: d7{a2 a5 a7}{n7 n2 n3} ==> d7a4≠7,
d7a4≠3, d7a6≠7, d7a1≠7, d7a3≠2
whip[1]: r3n2{c6 .} ==> r7c6≠2
whip[1]: r7n2{c7 .} ==> r1c1≠2
whip[1]: c4n7{r7 .} ==> r1c5≠7
whip[1]: d7n7{r7c1 .} ==> r7c6≠7
whip[1]: r6n3{c5 .} ==> r1c3≠3
whip[1]: r1n3{c7 .} ==> r2c7≠3
whip[1]: d1n3{r6c3 .} ==> r4c1≠3
whip[1]: r4n3{c6 .} ==> r7c6≠3
whip[1]: r7n3{c5 .} ==> r5c5≠3
stte
```

If only Subsets (and whips[1]) are activated, the same Naked Triplets in a diagonal can be applied immediately after RS1, short-circuiting all the chains between RS1 and RS2, and the rest of the path is in W1.

7.2.5 Example 2: Fish

Consider now the following 7×7 puzzle (also created by 1to9only). The resolution state after Singles and whips[1] is given immediately after the puzzle (it is a good exercise in whips[1] to try to reach this state by yourself – or see the Examples/LatinSquares/Pandiagonals folder for details of the whips[1], similar to those in the previous example):

```
. . . . . . .
. . . . . . .
. 2 . . 7 . .
. . . . . . .
. . . . . . 4
1 . . . 6 . .
. 3 . . . . .


..............2..7..............41...6...3.....
```

```
Resolution state RS1 after Singles and whips[1]:
2457  14567 25    13567 345   23457 135
3457  1457  1357  256   2345  1235  1567
345   2     13456 345   7     1456  15
3567  1457  3457  12345 1245  256   12357
257   567   12356 157   1235  12357 4
1     45    2457  23457 6     357   2357
2456  3     257   12457 15    1457  2567
```

Because that's what we're interested in in this example, only Subsets have been activated in the configuration file, leading to the following resolution path. After the above resolution state, there appears a sequence of Swordfishes in columns, diagonals or anti-diagonals with transversal cover sets of various types:

```
swordfish-in-columns-w-transv-anti-diags: n2{c1 c5 c7}{a2 a4 a1} ==>
a4c3≠2, a2c6≠2, a1c4≠2
swordfish-in-diags-w-transv-columns: n4{d2 d3 d7}{c4 c1 c2} ==>
d5c2≠4, d6c4≠4, d1c1≠4
swordfish-in-anti-diags-w-transv-rows: n3{a4 a5 a7}{r4 r2 r1} ==>
r4a1≠3, r2a2≠3, r1a6≠3
swordfish-in-anti-diags-w-transv-columns: n1{a4 a5 a7}{c7 c6 c4} ==>
a2c6≠1, a1c4≠1, a6c7≠1
(for the end of the resolution path, see the Examples folder)
```

Notice that both base and cover sets are homogeneous in all the cases (i.e. made of a single type: rn, cn, dn or an). (Currently, fully mixed Fishes are not coded in LatinRules – and they will probably never be, as there are too many possible cases. They are the exotic Fishes of Pandiagonal Latin Squares. An exotic example would be an X-Wing with base sets r_1n_1 and d_3n_1 and with cover sets c_4n_1 and a_6n_1.)

In such conditions, "transversal" cover sets merely means that cover sets are of a different kind than base sets. Given any type x of homogeneous base sets (x = rn, cn, dn or an), a homogeneous transversal cover set can be of any homogeneous type (rn, cn, dn or an) except x.

Notice that, for Fish patterns, LatinRules uses the coordinate system the most appropriate for each of them, according to the types of the base and cover sets.

For much harder Pandiagonal puzzles, see the Examples folder.

8. FutoRules

FutoRules is the pattern-based solver of Futoshiki puzzles based on CSP-Rules. Futoshiki is played on a square grid. Grid size is unrestricted (and FutoRules does not have any *a priori* restriction), but in practice, it will rarely be larger than 9.

8.1 The configuration file

The generic part looks much like the generic Sudoku part described in chapter 5. In particular, it allows to select g-whips and g-braids.

The Futoshiki specific part deals with allowing (or not): 1) the presence of givens in the cells (i.e. dealing with “impure” or “pure” Futoshiki); 2) the rules specifically related to sequences of inequalities and 3) the Subsets rules. By default:

- Futoshiki is pure (i.e. it does not have givens in the cells, it has only inequality symbols); but this can be changed in the configuration file;
- all the rules related to sequences of inequalities are disabled by default; but this can be changed in the configuration file, and this is indeed changed in my standard configuration, because they are as natural to Futoshiki as Subsets are to Sudoku;
- the option for Subsets is necessarily FALSE by default in CSP-Rules; it can be set to TRUE in the configuration file and it is indeed TRUE in my standard config.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; My standard config and its usual variants
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 ;;; change the default type of Futoshiki; pure by default (i.e. there is no given in
the cells)
; (bind ?*pure-Futoshiki* FALSE)                          <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;;; My most usual rules:
(bind ?*asc-chains* TRUE) ; enable ascending chains
(bind ?*str-asc-chains* TRUE) ; enable strong ascending chains
(bind ?*hills-and-valleys* TRUE) ; enable hills and valleys
(bind ?*Subsets* TRUE)
(bind ?*Bivalue-Chains* TRUE)
(bind ?*Whips* TRUE)

;;; Some additional rules I use frequently:
; (bind ?*t-Whips* TRUE)...
```

8.2 The user functions

FutoRules has only two user functions.

1) Function "***solve***" has four arguments, with the following syntax:

```
(solve ?nb-rows ?givens-str ?horiz-ineq-str ?verti-ineq-str)
```

where:

- ?nb-rows is the number of rows (or columns),
- ?givens-str is a string representing the sequence of givens in the cells (with the same conventions as in SudoRules: a dot represents no given):

  "....."
- ?horiz-ineq-str is a string representing the sequence of horizontal inequality signs,
- ?verti-ineq-str is a string representing the sequence of vertical inequality signs.

Example (Figure 8.1, H9305 from ATK:
http://www.atksolutions.com/games/futoshiki.html, solvable in S+W6):

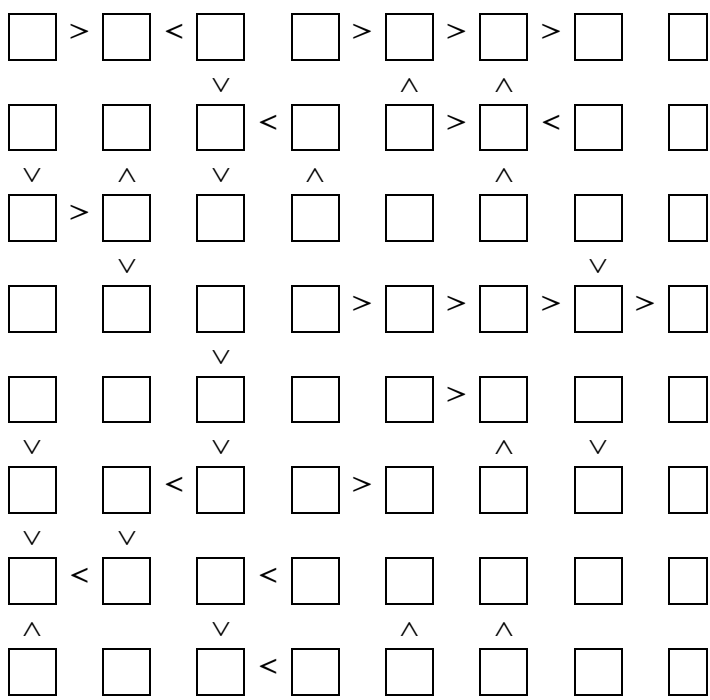


Figure 8 .1. *Futoshiki puzzle H9305 from ATK*

To solve this 8×8 pure Futoshiki puzzle, the call to function “solve” should be:

```
(solve 8
   "................................................................"
   "><->>>---<->-><->----------->>>>----->---<->---<-<------<----"
   "->-->>><-<>-->->>->>->-><-----<-----<<<--<-<-->->---------"
)
```

The first argument is 8, the number of rows and columns of the puzzle. The second argument is as in Sudoku, with a dot representing no given in the corresponding cell; in “pure” Futoshiki (no givens), this string reduces to a sequence of n^2 dots, as in this example.

The last two arguments require more explanations. On each row of the puzzle, there are n cells and there are at most n-1 inequality signs. If “-” is used to represent the absence of an inequality between two cells, then a row can be represented as a sequence of n-1 signs, looking like this (for the first row): ><->>>-

The whole set of horizontal left to right inequalities, from top row to bottom row, will be a set of n similar lines:

```
><->>>-
--<->-<
>------
--->>>>
---->--
-<->---
<-<----
--<----
```

Stick these n lines together (by deleting the carriage returns), put them between double quotes and you get the first argument:

```
“><->>>---<->-><->---------->>>>----->---<->---<-<------<----"
```

Similarly, write the set of top to bottom vertical inequalities, from left column to right column, as n lines of n-1 signs:

```
->-->>><
-<>-->-
>>->>->
-<-----
<-----<
<<--<-<
-->->--
-------
```

Stick these n lines together (by deleting the carriage returns), put them between double quotes and you get the second argument:

```
"->-->>><-<>-->->>->>->-><-----<-----<<<--<-<-->->---------"
```

2) Function "***solve-tatham***" has two arguments, with the following syntax:

```
(solve-tatham ?n ?tatham-str)
```

Suppose you generate a Futoshiki puzzle on the Tatham webpage for Futoshiki (where it's named "unequal"

https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/unequal.html)

You will get a puzzle like in Figure 7.2 (this is a 7×7 "Tricky" one):

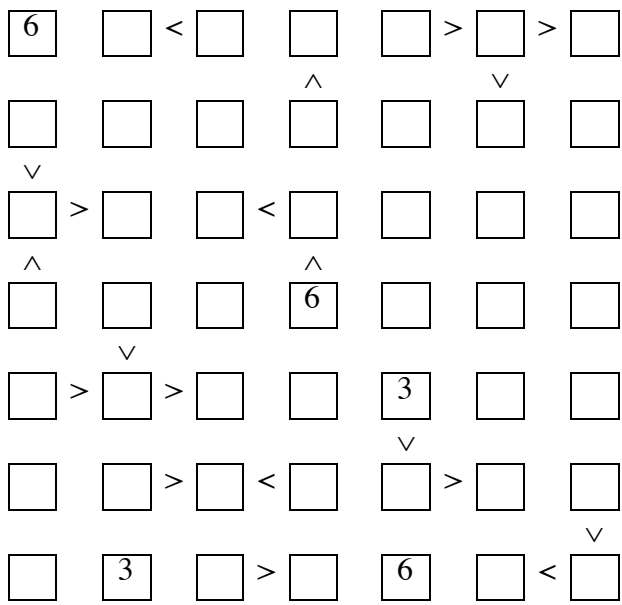


Figure 8.2. *"Tricky"Futoshiki puzzle from Tatham*

Now, suppose you click "link to this puzzle by game ID". Your browser will give you an address that looks like this:

```
https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/unequal.html#7:6,0,0L,0,0R,0RD
,0,0D,0,0,0U,0,0,0,0R,0,0,0L,0,0,0,0U,0D,0,6U,0,0,0,0R,0R,0,0,3D,0,0,0,0R,0,0L,0R,0,0D
,0,3,0R,0,6,0,0L,
```

What I call the tatham string format for a Futoshiki puzzle is the part of this address following the second colon symbol, considered as a string. Notice that the address also encodes the size of the puzzle (between the "#" and the second ":"), but I prefer not to consider it as part of the representation of the puzzle, because grid size information is usually separate in the other CSP-Rules applications. This size

will be given as the value for argument ?n. Thus, the tatham string format for the above 7x7 Futoshiki puzzle is:

```
“6,0,0L,0,0R,0RD,0,0D,0,0,0U,0,0,0,0R,0,0,0L,0,0,0,0U,0D,0,6U,0,0,0,0R,0R,0,0,3D,0,0,0
,0R,0,0L,0R,0,0D,0,3,0R,0,6,0,0L,”
```

This string can be considered as having 7×7 small parts, each followed by a comma. The nth part corresponds to the content of the nth cell (a digit, including 0), plus optionally one U (greater than the cell upwards), one D (greater than the cell downwards), one L (greater than the cell on the left) and/or one R (greater than the cell on the right); the symbols U, D, L, R express inequalities between cells. All four of them may appear in the same small part of the string, which will then be a hill, both horizontally and vertically.

Not surprisingly, function “solve-tatham” for this puzzle is written as:

```
(solve-tatham 7
“6,0,0L,0,0R,0RD,0,0D,0,0,0U,0,0,0,0R,0,0,0L,0,0,0,0U,0D,0,6U,0,0,0,0R,0R,0,0,3D,0,0,0
,0R,0,0L,0R,0,0D,0,3,0R,0,6,0,0L,”)
```

FutoRules has a function “tatham-to-FutoRules-strings” that outputs a list of the three string arguments to function “solve”; “solve-tatham” is the proper combination of those two functions. The command:

```
(tatham-to-FutoRules-strings 7
“6,0,0L,0,0R,0RD,0,0D,0,0,0U,0,0,0,0R,0,0,0L,0,0,0,0U,0D,0,6U,0,0,0,0R,0R,0,0,3D,0,0,0
,0R,0,0L,0R,0,0D,0,3,0R,0,6,0,0L,”)
```

outputs a list of the three strings:

```
("6.....................6.......3.........3..6.."
 "-<-->>------>-<--------->>----->-><->--->--<"
 "->><------>--------<-<------->->---------->")
```

Instead of “solve-tatham”, function “solve” can be directly called using the above three strings, with the same result as the above call to “solve-tatham”:

```
(solve 7
   "6.....................6.......3.........3..6.."
   "-<-->>------>-<--------->>----->-><->--->--<"
   "->><------>--------<-<------->->---------->"
)
```

Remarks:

- the above Tatham puzzle provides an example of Triplets in Futoshiki;
- the Tatham classification of puzzles is unclear to me. I would have thought that “Recursive” meant the puzzles were not solvable without T&E(2); but FutoRules has solved some “Recursive” ones using only whips, which implies they are in T&E(1). On the other hand, most “Extreme” puzzles should be solvable by FutoRules without requiring T&E but a few are not.

8.3 Some specific notations in the resolution path

As it has application-specific rules, FutoRules has some specific notation for them. All the examples in this section are taken from the resolution path of the Tatham puzzle in section 8.2.

Ascending chains (respectively strong ascending chains) appear as follows:

```
asc[3]: r3c3<r2c3<r1c3<r1c2 ==> r3c3≠7, r3c3≠6, r3c3≠5, r2c3≠7, r2c3≠6, r2c3≠1, r1c3≠7,
r1c3≠1, r1c2≠3, r1c2≠1
str-asc[1]: r4c2<r4c1 ==> r4c1≠3
```

The number between square brackets is the length of the chain, measured by the number of < signs. It should not mislead you about the complexity of the pattern. I've shown in [PBCS] that, as far as eliminations are concerned, an ascending chain is equivalent to a series of whips[1]. Ascending chains are an example of an application-specific rule that can be interpreted as a macro-rule, i.e. a sequence of simpler ones. These rules are activated by default in my usual configuration, because they provide more compact, better looking resolution paths, although they add no resolution power.

Hills, valleys, strong hills and strong valleys also have their own notation, similar to that of ascending chains, with the same convention about the pseudo-length.

```
str-hill[3]: r5c3<r5c4>r5c5>r5c6 ==> r5c4≠4
```

But they can only be reduced to much more complex rules (S_p-whips) and they do add to resolution power. However, as they rely on a very natural pattern, they are also activated by default in my usual configuration:

```
;;; My most usual rules:
  (bind ?*asc-chains* TRUE) ; enable ascending chains
  (bind ?*str-asc-chains* TRUE) ; enable strong ascending chains
  (bind ?*hills-and-valleys* TRUE) ; enable hills and valleys
  (bind ?*Subsets* TRUE)
  (bind ?*Bivalue-Chains* TRUE)
  (bind ?*Whips* TRUE)

;;; Some additional rules I use frequently:
; (bind ?*z-Chains* TRUE)
; (bind ?*t-Whips* TRUE)
; (bind ?*G-Whips* TRUE)
```

9. KakuRules

KakuRules is the pattern-based solver of Kakuro puzzles based on CSP-Rules. Kakuro can be played on any rectangular grid, but most of the grids we meet in practice are square. KakuRules requires a square grid (but a rectangular one can be straightforwardly transformed into a square one, by adding dummy black cells with no content).

Kakuro has Subsets and g-whips and both patterns are noteworthy. The theory of g-labels in Kakuro has been developed in full detail in [PBCS], with real examples only in [PBCS2] (due to a stupid bug in the implementation of glabels in KakuRules at the time of [PBCS1]). It is quite complex but, as in Sudoku or even more, it leads to one of the most fascinating applications of g-whips (see examples in [PBCS2]).

Remarks and warnings:

1) KakuRules does not have any version of the rules related to “surface sums”, as they are explained in [PBCS]. It does not mean you cannot use KakuRules in case you notice almost closed areas in the grid. After properly using the surface sums, you can try to solve each resulting small puzzle separately and then inject the partial result(s) into the global puzzle. You can also apply KakuRules directly to the global puzzle, but then it may sometimes require (much) longer chains than if you first use the “surface sums”. I make no statement about whether the presence of surface sums is a good thing or not in a Kakuro puzzle; every player will have his own opinion on them. I’m just warning: don’t expect KakuRules to deal with them as such by itself.

2) In the CSP-Rules approach in general and in KakuRules in particular, additional CSP-Variables are added to the “natural” ones. In KakuRules, this means variables related to the sums in the black cells, i.e. variables intended to keep track of the *combinations* allowed in each horizontal or vertical sector. It is natural to attach these additional CSP-Variables to the black cells and to count the first (totally black) row and the first (totally black) column as parts of the Kakuro grid. This justifies the default numbering I’ve adopted for rows and columns, i.e. 1 for the first (black) row (or column). Similarly, the size or the grid (number of rows or columns) includes the first (black) row or column. The configuration file allows to change the numbering of the rows and columns (i.e. to start at 0 instead of 1) when printing the resolution path, but this does not change the size of the grid parameter required by function “solve”.

3) KakuRules has an interesting application of Typed-Chains (see section 8.3), totally different from the SudoRules one.

9.1 The configuration file

There is nothing much to say about the KakuRules configuration file. As said above, you can change the numbering of the rows and columns (setting 0 instead of the default 1 for the first all-black row and column). You can also disable the initial data consistency check (not recommended).

Kakuro has the most beautiful applications of g-whips. Don't miss trying them.

9.2 The user functions

K	17	42				8	5		
7			13		6 20			41	
9				19 13					13
35							12 11		
13			12 3			24			
	5 6				17	9 8			19
13				11			3 8		
9			12	29 15					
	23					12			
		16					4		

Figure 9.1. *ATK Kakuro puzzle H2340*

Basically, there is only one user function: "***solve***", with syntax:

```
(solve ?n $?givens)
```

?n is grid size (including the first row or column).

?givens is a sequence representing all the row and column data: first all the rows from top to bottom, then all the columns from left to right. The idea for this format, with separate data for the horizontal and vertical parts, was inspired by crosswords.

It has some syntactic sugar (the separate lines, the single and double slashes) for better clarity and for easy checks of consistency of the horizontal data with the vertical ones.

Once again, I'll use an example from the ATK website, (Figure 9.1, puzzle H2340, a puzzle that can be solved in W6), to illustrate the proper syntax. For this puzzle, the proper call to "solve" is as follows.

As size is n, there are n-1 lines of horizontal data, followed by n-1 lines of vertical data. In the horizontal data part, there's no line corresponding to the first (black) row; in the vertical data part, there's no line corresponding to the first (black) column: they would contain no information.

```
(solve 10
;;; horizontal data:
7 . . B B 6 . . B B /
9 . . . 19 . . . . B /
35 . . . . . B 12 . . /
13 . . 12 . . 24 . . . /
B 5 . . B B 9 . . B /
13 . . . 11 . . 3 . . /
9 . . B 29 . . . . . /
B 23 . . . . 12 . . . /
B B 16 . . B B 4 . . //

;;; vertical data:
17 . . . . 6 . . B B /
42 . . . . . . . . B /
B 13 . . 3 . . 12 . . /
B B 13 . . B B 15 . . /
B 20 . . . 17 . . . B /
8 . . B B 8 . . B B /
5 . . 11 . . 8 . . B /
B 41 . . . . . . . . /
B B 13 . . 19 . . . . //
)
```

Notice that the spaces between each entry are mandatory (writing "BB" instead of "B B" will produce an error message).

Each line must have n+1 symbols, each of which can only be:

- a dot, representing a white cell,
- an integer between 3 and 45, representing a sum constraint (horizontal if it's in the horizontal data part, vertical if it's in the vertical data part),
- symbol "B" representing a black cell with no horizontal (respectively vertical) sum constraint if it's in the horizontal (resp. vertical) data part,

- a slash, denoting the end of a line (it must appear at the end of the line and it may only appear there),
- a double slash, denoting the end of the horizontal and vertical parts (it must appear at these places in the sequence and it may only appear there).

What function "solve" deals with what could be called "pure" Kakuro, in the same sense as we spoke of "pure" Futoshiki. KakuRules also has a user function, "***solve-partly-solved-puzzle***", for solving a puzzle with givens in the white cells, with syntax:

```
(solve-partly-solved-puzzle ?n $?givens)
```

This function may be useful when you find an almost closed surface, you apply the surface sums rule, you solve some of the small sub-puzzle(s) and you want to inject the partial results thus obtained into the global puzzle.

"solve-partly-solved-puzzle" seems to have the same syntax as "solve", but the $?givens argument must contain additional information, for expressing the givens in the white cells. This information will appear after the horizontal and vertical parts. For the above puzzle (which has no white cell data), the proper call would be:

```
(solve-partly-solved-puzzle 10
;;; horizontal data:
7 . . B B 6 . . B B /
9 . . . 19 . . . . B /
35 . . . . . B 12 . . /
13 . . 12 . . 24 . . . /
B 5 . . B B 9 . . B /
13 . . . 11 . . 3 . . /
9 . . B 29 . . . . . /
B 23 . . . . 12 . . . /
B B 16 . . B B 4 . . //

;;; vertical data:
17 . . . . 6 . . B B /
42 . . . . . . . . B /
B 13 . . 3 . . 12 . . /
B B 13 . . B B 15 . . /
B 20 . . . 17 . . . B /
8 . . B B 8 . . B B /
5 . . 11 . . 8 . . B /
B 41 . . . . . . . . /
B B 13 . . 19 . . . . //

;;; white cells data:
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
```

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
)
```

The horizontal and vertical parts are the same as in function “solve”. The white cells part contains the already solved cells (none in the present case). The above calls to “solve” and to “solve-partly-solved-puzzle” for the same puzzle are equivalent. Try them to check that they give exactly the same resolution path. Of course, it would be particularly absurd in practice to use “solve-partly-solved-puzzle” in case no white cell is solved.

Notice that the white cells data part has n-1 lines of n-1 symbols each; it totally discards the first (black) row and column. Each line corresponds to a row in the grid (except the first black one) and its content corresponds to the n-1 cells of this row in the puzzle (excluding the first black cell). Each symbol may only be a dot or a non-zero digit. Non-zero digits may only appear at a place corresponding to a white cell.

It you want to try a real partly solved puzzle, you can change the white part in the above puzzle, by adding a single given, into the following. You will notice that the result is now in W5 instead of W6:

```
;;; white cells data
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. 8 . . . . .
. . . . . .
. . . . . .
```

9.2.1 An example from [PBCS2]

In order to end this section with a very good example of whips, g-whips, surface sums, …, you can play with the puzzle discussed in [PBCS2], by trying different sets of rules (showing that it is in W10 but in gW8, comparing the different resolution times and memory requirements), using surface sums, … It’s a hard puzzle from ATK (Figure 9.2).

𝒦	10	16	23	12			33	11	11	15	8	
23					20	33						8
32						23						
	13	11 26				11 13	D		15	9 20		
14			25		C			13 8				
7				26	16 14						24	11
6			17 8				19 23				A	
27		B				19 21				12		
		19						8	25	13		
	3	11 16			12 21			E		4 14		
7			14	17 3		F	21				17	4
22							21					
	27							23				

Figure 9.2. *ATK Kakuro puzzle H83722*

To make it easier to try it for yourself, here is the proper call to function “solve”:

```
(solve 13
;;; horizontal data:
23 . . . . B 33 . . . . . B /
32 . . . . . 23 . . . . . /
B B 11 . . . 11 . . B 9 . . /
14 . . 25 . . . . 13 . . B B /
7 . . B B 16 . . . . . B B /
6 . . 17 . . . 19 . . . . . /
27 . . . . . 19 . . . 12 . . /
B B 19 . . . . . B B 13 . . /
B B 11 . . 12 . . . . 4 . . /
7 . . B 17 . . 21 . . . B B /
22 . . . . . 21 . . . . . /
B 27 . . . . . B 23 . . . . //
```

```
;;; vertical data:
10 . . 13 . . . . B 3 . . B /
16 . . 26 . . . . B 16 . . . /
23 . . . B B 8 . . . 14 . . /
12 . . . . 26 . . . . 3 . . /
B 20 . . . 14 . . . 21 . . . /
B B B 13 . . . 21 . . . . . /
33 . . . . . 23 . . . B B B /
11 . . . 8 . . . 8 . . . B /
11 . . 15 . . . . 25 . . . . /
15 . . 20 . . . B B 14 . . . /
8 . . . B 24 . . . . 17 . . /
B 8 . . B 11 . . . . 4 . . //
)
```

9.3 Typed-Chains in KakuRules

In chapter 6, we have seen how the generic Typed-Chains can be applied to SudoRules, when we adopt a typing system based on the four kinds of CSP-Variables specific to Sudoku, namely the "natural" rc ones and the additional rn, cn and bn ones, as I introduced them in [HLS] and as they are represented by the four 2D spaces of the Extended Sudoku Board. However, such a typing system would not be very interesting in Kakuro, because the most important interactions between CSP-Variables of types hrc (or vrc) and rc (i.e. interactions between the still possible combinations for a sector and the still possible values in the white cells of this sector) would be discarded in the resulting Typed-Chains. Instead of this, I've therefore defined a totally different typing system, where a type corresponds to a fixed (vertical or horizontal) sector.

For each horizontal sector, represented by its controller (black) cell r_ic_k and named correspondingly hr_ic_k, the following three kinds of CSP-Variables are assigned the hr_ic_k type:

- hr_ic_k (i.e. the variable representing the still possible combinations for this sector)
- $r_{i'}c_{k'}$ for any white rc-cell $r_{i'}c_{k'}$ in the sector
- $r_{i'}n_{k'}$ for any rn-cell $r_{i'}n_{k'}$ in the sector and any mandatory number in this sector.

Of course, there is a corresponding definition for each vertical sector.

The attentive reader will have noticed that each CSP-Variable for a white cell is assigned two types, one corresponding to its horizontal sector and one corresponding to its vertical sector. However, this is not a problem for CSP-Rules because the notion of "type" used in the Typed-Chains is not a function but a predicate.

Remark: few Kakuro puzzles are solvable using only Typed-Chains. But those that are may define an interesting level of easy puzzles: the human player does not have to look for complex interactions between values in different sectors. The only interactions between different sectors will appear when candidates in the white cells are effectively deleted or turned into values. Everything harder happens in a single sector.

As such typed rules for Kakuro were not mentioned in [PBCS], time has come for an example. Once more, I'll take one from the ATK website, their puzzle M42942 (see Figure 9.3).

K	7	16	30	33	6	20		23	38
28							12		
39							9 11		
	39	17 23			11 15				
31						23 27			
16			19 13				14 27		
22				15 18					
19					11 13			5	6
5			32						
9			38						

Figure 9.3. *ATK Kakuro puzzle M42942*

The calling function from CSP-Rules is:

```
(solve 10
;;; horizontal data
28 . . . . . 12 . . /
39 . . . . . 9 . . /
B B 17 . . 11 . . . . /
31 . . . . . 23 . . . /
16 . . 19 . . . 14 . . /
22 . . . 15 . . . . . /
19 . . . . 11 . . B B /
```

```
5 . . 32 . . . . . /
9 . . 38 . . . . . //

;;; vertical data
7 . . 39 . . . . . /
16 . . 23 . . . . . /
30 . . . . 13 . . B B /
33 . . . . . 18 . . . /
6 . . 15 . . . 13 . . /
20 . . . 27 . . . . . /
B B 11 . . 27 . . . . /
23 . . . . . 5 . . /
38 . . . . . 6 . . //
)
```

This puzzle has a solution in W3 or in TyW5, as shown in the following two resolution paths. Note that, in the first resolution path, where untyped chains are activated, I've activated both typed and untyped chains, in order to show how they easily live together. Also, this allows to have a common path at the start, until resolution state RS1:

```
*******************************************************************************
***  KakuRules 2.1.s based on CSP-Rules 2.1.s, config = W+S
***  using CLIPS 6.32-r790
***  Running on MacBookPro Retina Mid-2012 i7 2.7GHz 16GB, 1600MHz DDR3, MacOS 10.15.4
*******************************************************************************
       Uppermost (black) row and leftmost (black) column have index 1
singles ==> r10c10=5, r10c9=3, r9c2=4, r6c3=7, r6c2=9, vr4c3=123467, r7c3=6,
hr7c1=679, r7c2=7, r7c4=9, vr6c4=49, r8c4=4, hr9c1=14, r9c3=1, vr8c9=23, r9c9=2,
vr8c10=15, r9c10=1, hr9c4=125789, ==> r4c9=1, r4c8=2, vr3c8=29, r5c8=9
323 candidates, 768 csp-links and 1672 links. Density = 3.22%
naked-pairs-in-verti-sector: c10{r4 r7}{n3 n5} ==> r6c10≠5, r3c10≠5, r3c10≠3, r2c10≠5,
r2c10≠3
biv-chain-vr1c6[2]: vr1c6{n15 n24} - r3c6{n5 n4} ==> r2c6≠4
biv-chain-vr1c6[2]: vr1c6{n24 n15} - r3c6{n4 n5} ==> r2c6≠5
biv-chain-vr1c9[2]: vr1c9{n123458 n123467} - r5c9{n8 n6} ==> r3c9≠6, r6c9≠6
singles ==> vr1c9=123458, r5c9=8, r6c9=5, r5c10=6, r3c10=7, hr3c8=27, r3c9=2,
hr6c8=59, r6c10=9, r2c10=8, hr2c8=48, r2c9=4, r7c9=3, r7c10=5, r7c8=4, r4c10=3,
r4c7=5, vr6c8=4689
z-chain-hr8c1[2]: r8c3{n3 n2} - hr8c1{n3457 .} ==> r8c5≠3
whip-hr8c1[2]: r8c3{n2 n3} - hr8c1{n2458 .} ==> r8c5≠2
naked-single ==> vr7c5=567
whip-vr4c6[2]: r7c6{n2 n1} - vr4c6{n249 .} ==> r6c6≠2, r5c6≠2
whip-hr8c6[2]: hr8c6{n56 n29} - r8c8{n6 .} ==> r8c7≠9
whip-hr8c6[2]: hr8c6{n56 n38} - r8c8{n6 .} ==> r8c7≠8
whip-hr8c6[2]: hr8c6{n38 n56} - r8c8{n8 .} ==> r8c7≠6
z-chain-vr5c7[2]: r8c7{n5 n2} - r7c7{n2 .} ==> vr5c7≠24678
whip-hr8c1[2]: hr8c1{n3457 n2467} - r8c2{n5 .} ==> r8c5≠6
naked-pairs-in-verti-sector: c5{r8 r9}{n5 n7} ==> r10c5≠7
```

```
singles ==> r10c5=6, vr8c6=58, r10c6=8, r10c8=9, r10c7=7, r9c8=8, r8c8=6, r9c6=5,
r9c7=9, r9c5=7, r8c5=5, hr8c6=56, r8c7=5, vr5c7=24579, r7c7=2, r7c6=1, r6c7=4, r8c2=8,
hr8c1=2458, r8c3=2
biv-chain-hr5c1[2]: r5c3{n4 n3} - hr5c1{n45679 n35689} ==> r5c5≠4
biv-chain-hr5c1[2]: hr5c1{n45679 n35689} - r5c3{n4 n3} ==> r5c5≠3
whip-hr6c4[2]: hr6c4{n469 n478} - r6c6{n6 .} ==> r6c5≠8
whip-vr4c6[2]: vr4c6{n168 n159} - r6c6{n6 .} ==> r5c6≠9
x-wing-in-horiz-sectors: n9{r4 r5}{c4 c5} ==> r6c5≠9, r3c5≠9, r3c4≠9, r2c5≠9, r2c4≠9
biv-chain-hr6c4[2]: hr6c4{n478 n469} - r6c5{n7 n6} ==> r6c6≠6
biv-chain-vr4c6[2]: vr4c6{n159 n168} - r6c6{n9 n8} ==> r5c6≠8
naked-pairs-in-horiz-sector: r5{c2 c6}{n5 n6} ==> r5c5≠6, r5c5≠5, r5c4≠6
whip-vr1c2[2]: vr1c2{n34 n16} - r3c2{n4 .} ==> r2c2≠6
whip-vr1c2[2]: vr1c2{n34 n25} - r3c2{n4 .} ==> r2c2≠5
whip-vr1c2[2]: vr1c2{n25 n34} - r3c2{n5 .} ==> r2c2≠4
;;; resolution state RS1
```

Here is now the end of the resolution path after state RS1, in case untyped chains are activated; you will notice an untyped bivalue-chains[3]; in this puzzle (where Subsets are activated) this is the only interaction between two sectors based on a rule involving several sectors (indeed only two, as the first two CSP-Variables are of the same type: vr1c7).

```
;;; resolution state RS1
biv-chain[3]: vr1c7{n569 n578} - r3n9{c7 c3} - r2c3{n9 n7} ==> r2c7≠7 ; <<<<<<<<<<<<<
whip-vr1c7[2]: vr1c7{n569 n578} - r2c7{n9 .} ==> r3c7≠8
x-wing-in-horiz-sectors: n8{r3 r4}{c4 c5} ==> r5c5≠8, r5c4≠8, r2c5≠8, r2c4≠8
singles ==> hr5c1=45679, r5c3=4, r10c3=3, hr10c1=36, r10c2=6, r5c2=5, r5c6=6
naked-single ==> vr4c6=168, r6c6=8, hr6c4=478, r6c5=7, r5c5=9, r5c4=7, r2c4=6, r3c4=8,
r4c4=9, r4c5=8
hidden-pairs-in-horiz-sector: r3{n7 n9}{c3 c7} ==> r3c7≠6
singles to the end
----------
-176428-48
-698547-27
---98-5213
-54796-986
-97-784-59
-769-12435
-8245-56--
-41-759821
-63-687935
```

Finally, here is the alternative end of the resolution path after state RS1, in case only typed chains are activated:

```
;;; resolution state RS1
z-chain-hr2c1[3]: r2c3{n7 n9} - r2c7{n9 n8} - r2c4{n8 .} ==> hr2c1≠123589
z-chain-hr2c1[3]: r2c3{n7 n9} - r2c7{n9 n6} - r2c4{n6 .} ==> hr2c1≠134569
whip-hr2c1[3]: r2c3{n9 n7} - r2c4{n7 n8} - r2c7{n8 .} ==> hr2c1≠134578
whip-hr2c1[3]: r2c4{n8 n7} - r2c3{n7 n9} - r2c7{n9 .} ==> hr2c1≠124579
```

```
ctr-to-horiz-sector  ==> r2c5≠5
whip-hr2c1[4]: r2c3{n7 n9} - hr2c1{n124678 n123679} - r2c4{n8 n6} - r2c7{n6 .} ==>
r2c5≠7
whip-hr2c1[4]: hr2c1{n123679 n124678} - r2c3{n9 n7} - r2c4{n7 n6} - r2c7{n6 .} ==>
r2c5≠8
whip-vr1c5[2]: vr1c5{n36789 n45789} - r2c5{n6 .} ==> r3c5≠4
whip-hr2c1[5]: hr2c1{n124678 n123679} - r2c5{n4 n6} - r2c4{n6 n7} - r2c3{n7 n9} -
r2c7{n9 .} ==> r2c2≠3
singles to the end
```

Which of the two resolution paths is simpler for a human solver is a matter of taste. As should be expected, using only less powerful chain rules (Typed-Chains) requires some form of compensation and this is provided in the present case by using longer chains.

10. SlitherRules

SlitherRules is the pattern-based solver of Slitherlink puzzles based on CSP-Rules. Slitherlink is played on a rectangular grid and SlitherRules does allow the numbers of rows and columns to be different (and function “solve” will therefore require two size arguments, nb-rows and nb-columns). However, most Slitherlink puzzles found on the web are square.

There are no Subsets in Slitherlink.

Computationally, the main characteristic of Slitherlink is its large fan-out or branching factor: most candidates are linked to many ones. This explains the reputation of Slitherlink as a hard kind of puzzle for automated solvers. For the human player, he has to do many obvious eliminations between each interesting step. (It is worth trying to solve manually some of these puzzles if you want to really understand what I mean here). When such steps are done by an automated solver and each step is printed out, the resolution path seems extremely boring.

As a result of the large fan-out, each resolution state may lead to many different ones via Elementary Constraint Propagation rules. For the same reason, each chain of length n-1 can give rise to many chains of length n and this can lead very fast to memory overload. Therefore, by default, chains are limited to length 5 in the Slitherlink configuration file. For large grids, this may still be too large. For small grids, if you have enough memory, you can try to extend this to 7 if 5 is not enough to solve the puzzle. Be warned, SlitherRules is the slowest part of CSP-Rules.

Slitherlink does have whips[1], indeed lots of them, as you will see in the resolution paths. Therefore, it has g-labels and it could have g-whips. However, due to the already very large branching factor, I haven’t coded g-labels; as a result, g-whips or any g-something cannot be activated. You may want to try g2-whips, a special case of g-whips.

Of all the CSP-Rules applications described in this Manual, SlitherRules has the largest number of application-specific resolution rules, the main purpose being to shorten the resolution paths. A complete and detailed description of each of them is available in [PBCS2]; it is largely inspired by what I found in [slinker www] and in the Wikipedia page (which seems to be a copy of [slinker www], without the proper credit); what I introduced is only some rational organisation and a systematic naming convention for the rules.

Using SlitherRules as an assistant theorem prover, I also showed that most (but not all) of these application-specific rules are equivalent to sequences of Singles and whips[1] (with a few of them needing slightly longer whips).

One change to be noticed with respect to version 1.0 used in [PBCS2] is the higher saliences given to Quasi-Loops and Colour rules. The reason is, these rules are very general and very easy to use. They have been raised to just before the whips[1]. As a result, the resolution paths may be different from those given in [PBCS2].

Other changes are the introduction of isolated-HV-chains and extended Quasi-Loops (see section 10.5) and of pre-computed backgrounds for square grids (making initialisation times much shorter).

10.1 The configuration file

After the generic settings that are the same in all the CSP-Rules applications, you can choose application-specific ones:

H, V, I, P and B singles are elementary rules for Singles applied to the H, V, I, P and B CSP-Variables. Contrary to "normal" Single rules, these are not printed by default, but they can be activated individually. If you set them to TRUE here, long sequences of them will appear:

```
;;; As H/V-singles, I-singles, P-singles and B-singles are trivial rules that
;;; appear very often, their output is not printed by default.
;;; Printing can be enabled here:
 (bind ?*print-HV-single* TRUE)
 (bind ?*print-I-single* TRUE)
 (bind ?*print-P-single* TRUE)
 (bind ?*print-B-single* TRUE)
```

You can also decide how the final output will appear:

```
;;; By default, the final output is not printed in any form.
;;; But you can independently choose to print it in two forms: HV borders and in/out
cells.
;;; (However, IO will be effectively printed only if rules for Colours are activated.)
 (bind ?*print-IO-solution* TRUE)
 (bind ?*print-HV-solution* TRUE)
```

Global variable ?*print-IO-solution* controls whether the solution will be printed in the form of inner and outer cells (if ?*Colours* has been set to TRUE), as in the following, where an "x" stands for an inner cell and an "o" for an outer cell.

```
OXOXO
XXXXX
XXOOX
XXXOO
XOXXO
```

Global variable ?*print-HV-solution* controls whether the solution will be printed in the (standard) form of borders around the cells, as in the following standard form:

```
.   .──.   .──.   .
    | 3 |   | 3 | 2
.──.   .──.   .──.
| 2   0           2 |
.   .   .──.──.   .
|       | 3     |   |
.   .   .──.   .──.
| 1         | 2   1
.   .──.   .──.   .
| 3 | 3 |     3 |
.──.   .──.──.   .
```

When a complete solution is not found, the final partial solution can be printed (default option) or not, depending on the same global variables as for the full solution.

The next choices are about application-specific rules. These rules are separated into four groups: quasi-loops and extended-loops, colours, rules that are equivalent to series of Singles and whips[1], and rules that are not (which is in which group is made clear in [PBCS2]). The four groups are disabled by default in CSP-Rules and they can be separately activated; I did not consider useful to add a possibility for individually activating each rule.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;;; My standard config and its usual variants
 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Without Loops, most puzzles cannot be solved;
;;; they are therefore always activated in my usual configuration.
;;; By default, ?*loops-max-length* is set to 300;
;;; it may have to be increased for very large puzzles (size > 15); change it here:
; (bind ?*loops-max-length* 300)
 (bind ?*Loops* TRUE)
;;; extended-Loops are useful in some circumstances
;;; their max-length is the same as Loops.
; (bind ?*xtd-Loops* TRUE)

;;; Propagation of colours (or in/out positions) is also a very general and useful
method.
 (bind ?*Colours* TRUE)

;;; By default, all the application-specific rules are FALSE
;;; They are enabled in  my standard configuration:
 (bind ?*W1-equiv-patterns* TRUE)
 (bind ?*non-W1-equiv-patterns* TRUE)
```

```
;;; generic:
 (bind ?*Bivalue-Chains* TRUE)
 (bind ?*bivalue-chains-max-length* 5)
 (bind ?*Whips* TRUE)
 (bind ?*whips-max-length* 5)

;;; Some additional rules I use frequently:
; (bind ?*z-Chains* TRUE)
; (bind ?*z-chains-max-length* 5)
; (bind ?*t-Whips* TRUE)
; (bind ?*t-whips-max-length* 5)

;;; Some rules I use occasionally:
; (bind ?*G2-Whips* = TRUE)
; (bind ?*g2whips-max-length* 5)

;;; Some rules I almost never use:
; (bind ?*Braids* TRUE)
; (bind ?*braids-max-length* 5)
```

10.2 The user functions

Slitherlink has only two user functions, “solve” and “solve-tatham”, plus an auxiliary function for changing formats.

Example of using “solve” for a 10×10 puzzle:

```
(solve 10 10
2 . . 3 . 3 2 . . .
. 3 . 3 . . . 3 2 2
. . . 2 2 . 1 2 2 .
. 1 . 2 . . . . . 3
. . 2 2 . . . 2 . .
3 3 . . 3 1 3 . 2 .
2 . . . . . . . . 3
. 3 1 . . 2 . . 1 .
2 1 2 2 1 1 2 . . 3
. . . 1 . . . . 1 .
)
```

The syntax for “***solve***” is:

```
(solve ?nb-rows ?nb-columns $?givens)
```

where ?nb-rows (respectively ?nb-columns) is the number of rows (resp. columns) and $?givens is the sequence of givens in the cells (as usual, a dot denotes no given). $?nb-givens must have exactly nb-rows × nb-columns symbols. As usual, the givens may (but don’t have to) be arranged in an easy to read format.

“***solve-tatham***” is a syntactic variant of “solve”, based on the tatham format. The syntax is:

```
(solve-tatham ?nb-rows ?nb-columns ?tatham-str)
```

where ?nb-rows (respectively ?nb-columns) are as before and ?tatham-str is the string representation of the puzzle in the Tatham format for Slitherlink. For the above puzzle, it would be

```
2b3a32d3a3c322c22a122b1a2e3b22c2b33b313a2a2h3a31b2b1a2122112b3c1d1a
```

In the Tatham format, each letter stands for the corresponding number of dots. Take the Tatham string, replace each letter by the right number of dots, insert a space between any two symbols (re-format it to make it look nicer) and you get the sequence used in the "solve" function. We have that:

```
(solve-tatham 10 10
   "2b3a32d3a3c322c22a122b1a2e3b22c2b33b313a2a2h3a31b2b1a2122112b3c1d1a ")
```

is equivalent to:

```
(solve 10 10
   (tatham-to-csp-rules-list
      "2b3a32d3a3c322c22a122b1a2e3b22c2b33b313a2a2h3a31b2b1a2122112b3c1d1a"))
```

If you want to use the Tatham generator of Slitherlink puzzles, you should be aware that most of its "recursive" puzzles will require the use of T&E or DFS. The most entertaining Slitherlink puzzles I've found on the web are at https://www.puzzle-loop.com; but you have to copy them manually.

10.3 A lot of application-specific rules

Of all the applications of CSP-Rules, SlitherRules is the one for which I've coded the largest number of application-specific rules. Part of this work was somehow frustrating, as most of these rules can be replaced by whips[1] and they did not increase the resolution power of SlitherRules. But it made the output look much nicer, as it involved rules familiar to the players and it shortened the resolution paths by directly asserting several H or V lines. It also allowed to single out a few rules that did add some resolution power.

Slitherlink illustrates on a much larger scale than Futoshiki the idea of a *macro-rule*, i.e. a resolution rule that can be reduced in some resolution theory T to a sequence of rules in T. Although macro-rules don't extend the resolution power of T, they can lead to shorter (and therefore more readable) resolution paths.

With this large number of application-specific rules, there appeared a strong need to test them individually. Slitherlink has therefore a "TEST" directory that reproduces the structure of the "SPECIFIC" directory: for each file for an application-specific rule, there is a file for testing it, with the same name. Load only the W1-equiv and non-W1-equiv rules (no Loops, Colours or Whips), set Final-Fill and Print-HV-solution to TRUE, choose the rule you want to test, go to the corresponding file in "TEST", launch the examples there and check the result.

10.4 An experience in assisted theorem proving

I've shown in [PBCS2] that most of the classical resolution rules specific to Slitherlink can be considered as macro-rules in W_1 (or in W_k, for some k>1, in a few cases), i.e. they can be reduced to sequences of Singles and whips[1] (or longer whips) – highlighting as a result those that are irreducibly Slitherlink-specific. For this purpose, I've used SlitherRules as an assistant theorem prover. This version of CSP-Rules allows you to easily reproduce all these proofs: activate only whips (and none of the Loops, Colours, W1-Equiv or Non-W1-Equiv patterns), open the file in "SlitheRules-V2.1/SPECIFIC/TESTS" corresponding to the rule you want to prove and paste into CSP-Rules one of the "(solve ….)" commands present in this file. You should obtain all the eliminations/assertions of H and V variables done by the rule (plus some for the P and B CSP-Variables).

The general idea of these assisted proofs was to create a pattern representing the situation one wants to deal with, on a sufficiently large grid to ensure that no unwanted corner or border condition creeps in, and then run SlitherRules on it. The result is a resolution path, each step of which is a short whip, with precise rows and columns, but this can easily be turned manually into a general proof: change the values of rows and columns in the given pattern into variable names, and then change accordingly all the row and columns in the resolution path into variables relative to the preceding ones. For instance, if the original pattern involves row 5 and column 6 and a row 6 and column 4 appear somewhere in the resolution path, first change "row 5" into variable ?row, "column 6" into variable ?col and then change "row 6" into (+ ?row 1) and column 4 into (- ?col 2). This is exactly what I've done for generating the proofs that appear in [PBCS2] (where I've also pruned the resolution path of its unnecessary steps). The final step involves checking that the tentative proof thus obtained is valid.

As the essential parts of these proofs have been published in [PBCS2], I leave it to the reader as an easy exercise to reproduce them for himself.

10.5 Isolated-HV-chains and Extended-Loops

In section 17.10.2 of [PBCS2], I mentioned the possibility of extending the definition of Quasi-Loops. I now have added this extension to SlitherRules-V2.1. Here are the relevant definitions:

- two undecided H or V lines with a common Point are said to have an *isolated junction* (at this Point) if the other two lines from this Point are decided and FALSE; as a result, the two given lines can only be TRUE or FALSE at the same time; they behave as a single line.
- a sequence of n H/V lines is called an *isolated-HV-chain[n]* if all these H/V lines are undecided and any two contiguous H/V lines in the sequence have an isolated

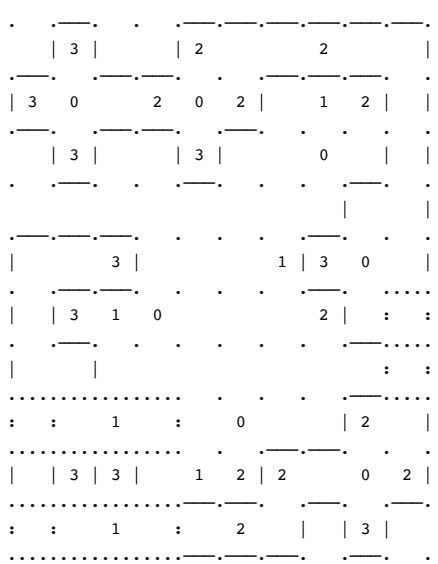
junction; as an obvious result, all the lines of this isolated-HV-chain[n] can only be TRUE or FALSE at the same time;

- an *Extended-Loop[n]* is defined as a Quasi-Loop[p] plus an isolated-HV-chain[q], such that p + q = n and the two chains meet at their two ends, making a closed loop; as a result, depending on the total number of lines touching the given cells, the whole isolated-HV-chain (i.e. all of its HV-lines) can be declared TRUE or FALSE; this is the Extended-Loop resolution rule.

Let's take the following example (puzzle I.4 from Mebane – see the "Examples" folder for details of the resolution paths):

```
. 3 . . 2 . . 2 . .
3 0 . 2 0 2 . 1 2 .
. 3 . . 3 . . 0 . .
. . . . . . . . . .
. . 3 . . . 1 3 0 .
. 3 1 0 . . . 2 . .
. . . . . . . . . .
. . 1 . . 0 . . 2 .
. 3 3 . 1 2 2 . 0 2
. . 1 . . 2 . . 3 .
```

If we apply all the rules in SlitherRules, except the Extended-Loops, we reach the following partial solution:



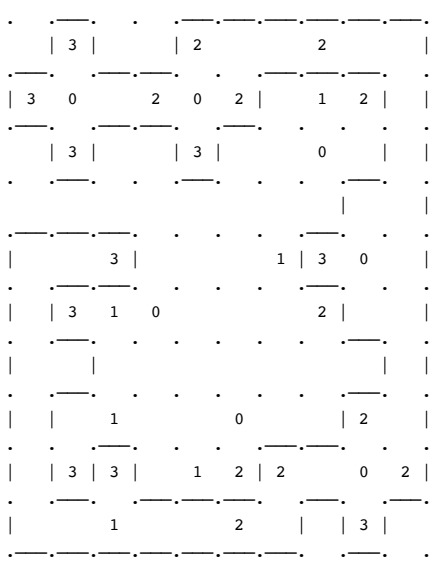
You can see three (and only three) isolated-HV-chains (you could argue there are indeed six chains, as they can be reversed):

- isolated-HV-chains [2]: Hr6c10-Vr6c10
- isolated-HV-chains [2]: Hr11c2-Hr11c3
- isolated-HV-chains [4]: Hr8c3-Hr8c4-Vr8c5-Hr9c4.

If we activate xtd-Loops, the part of the resolution rule up to the previous partial solution is unchanged, but then an Extended-Loop appears and it's enough to allow a full solution using a few more standard Loops (see the "Examples/ Slitherlink/Mebane" folder for details). The partial-loop part has length 42 and the isolated-HV-chain part has length 2 (hence the total length, 44):

```
XTD-LOOP[44]: Hr7c9-Vr6c9-Hr6c8-Vr5c8-Hr5c8-Vr4c9-Hr4c9-Vr3c10-Vr2c10-Hr2c9-Hr2c8-
Hr2c7-Vr2c7-Hr3c6-Vr3c6-Hr4c5-Vr3c5-Hr3c4-Hr3c3-Vr3c3-Hr4c2-Vr3c2-Hr3c1-Vr2c1-Hr2c1-
Vr1c2-Hr1c2-Vr1c3-Hr2c3-Hr2c4-Vr1c5-Hr1c5-Hr1c6-Hr1c7-Hr1c8-Hr1c9-Hr1c10-Vr1c11-
Vr2c11-Vr3c11-Vr4c11-Vr5c11- ==> Hr6c10-Vr6c10 = 0
```

Notice the convention for writing the Extended-Loop: on the left is the partial-loop made of decided TRUE H/V-lines; on the right is the isolated-HV-chain made of undecided H/V-lines; they all become decided by the conclusion of the rule (i.e. they are all set to FALSE in the present case).



11. HidatoRules (Numbrix and Hidato)

HidatoRules is the pattern-based solver of Hidato and Numbrix puzzles based on CSP-Rules. Both games consist of finding a path, using only adjacent white cells for each elementary step. The only difference between Numbrix and Hidato lies in the meaning of "adjacent": in Numbrix two cells are adjacent if and only if they touch each other by one side in the same row or column; in Hidato , they may also touch each other diagonally, i.e. by a corner.

11.1 The configuration file

Numbrix® and Hidato® have Subsets, but their implementation is greedy for memory and I often use only Subsets[2].

They also have whips[1] and therefore g-labels, g-whips,… But these are not coded in this version of HidatoRules, for the same reason they are not coded in SlitherRules: both applications have a large branching factor and g-whips would soon lead to memory overflow. Even g2-whips are not safe from this problem; you can try them, if the puzzle has small size (see the last but one example in this chapter).

11.2 The user functions

HidatoRules has a single user function, "***solve***". The syntax is:

```
(solve
   ?game ?model ?grid-size ?nb-white-cells $?data-sequence)
```

where:

- ?game is either Numbrix or Hidato
- ?model is either topological or geometric (see [PBCS] for the difference)
- ?grid-size is the number of rows (or columns)
- ?nb-white-cells is the number of white cells
- $?data-sequence is a sequence of ?grid-size × ?grid-size symbols; each symbol is either a dot or a "B" (for black cell) or a number between 1 and ?nb-white-cells (both included); no number can be repeated in the sequence. If there are B's, their number must be equal to ?grid-size × ?grid-size - ?nb-white-cells.

Numbrix example, from the https://parade.com/member/marilynvossavant/ website:

```
(solve Numbrix topological 9 81
69 .  67 .  65 .  61 .  57
.  .  .  .  .  .  .  .  .
77 .  .  .  .  .  .  .  55
.  .  .  .  .  .  .  .  .
19 .  .  .  .  .  .  .  45
.  .  .  .  .  .  .  .  .
3  .  .  .  .  .  .  .  39
.  .  .  .  .  .  .  .  .
5  .  7  .  11 .  31 .  33
)
```

Hidato example, from the https://www.smithsonianmag.com/games/ website:

```
(solve Hidato topological 10 78
B  B  .  .  65 .  .  62 B  B
B  .  19 .  .  66 .  .  .  B
.  21 .  B  .  .  B  .  58 59
.  .  12 B  .  .  B  69 .  56
.  10 .  74 .  .  .  1  .  .
.  26 .  5  4  76 .  .  54 52
28 .  B  .  .  .  78 B  .  .
.  .  .  B  B  B  B  48 47 .
B  .  .  .  .  39 .  43 .  B
B  B  33 35 37 40 42 .  B  B
)
```

Even small puzzles can be very hard. Example III.4 from the https://mellowmelon.files.wordpress.com/2012/05/pack03hidato_v3.pdf website (a puzzle in W8 or g2W7).

Generally speaking, the geometric model is more powerful, but it may lead faster to memory overflow for hard puzzles (because it has more links and therefore a larger fan-out). As an example of the difference, the following (hard) puzzle by Evert can only be solved with the geometric model, in W14. It can also illustrate the difference in computation time when you add special case patterns (e.g. z-chains, t-whips…) to only whips.

```
(solve Hidato geometric 9 81
.  .  .  .  77 .  28 .  .
.  13 .  .  .  .  .  21 .
.  .  .  15 16 .  31 .  .
.  6  .  .  73 .  19 .  .
.  58 .  .  .  .  36 .  33
60 .  62 .  71 .  .  .  .
.  .  .  .  64 .  .  .  43
.  54 .  .  .  68 .  .  .
.  .  .  .  .  .  .  .  .
)
```

12. MapRules

MapRules is the pattern-based solver of map colouring problems based on CSP-Rules. It has neither Subsets nor g-labels, which drastically restricts the possibilities for complex patterns. As the least developed application of CSP-Rules, it is intended only to provide one more illustration of the generic rules. A full solver of colouring problems should include pattern representation of graph concepts similar to those necessary for proving the four colour theorem.

12.1 The configuration file

There isn't much to say about the MapRules configuration file. My standard config is as follows.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; my standard config
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (bind ?*Bivalue-Chains* TRUE)
 (bind ?*Whips* TRUE)
; (bind ?*Braids* TRUE)

; (bind ?*bivalue-chains-max-length* 20)
; (bind ?*whips-max-length* 36)
; (bind ?*braids-max-length* 36)
```

12.2 The user functions

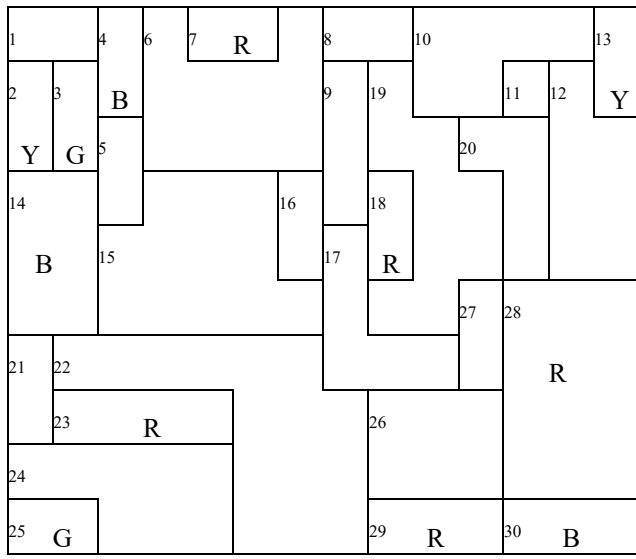
MapRules has a single user function: "***solve***" with syntax:

```
(solve ?nb-colours ?nb-countries ?givens $?list)
```

- ?nb-colours is the number of allowed colours;
- ?nb-countries is the number of countries to be coloured;
- ?givens is a string of given colours for some of the countries; its length must be equal to ?nb-countries; a country with no pre-defined colour appears as a dot;
- $?list is a sequence of adjacency data; it is composed of at most ?nb-count - 1 sub-sequences, separated by colons; each sub-sequence is a sequence of country numbers, meaning that the first country in the sub-sequence has a common border with each of the other countries in the same sub-sequence; if c1 < c2 and a

common border between c1 and c2 has already been declared in the sub-sequence starting with c1, it does not have to be declared again in the sub-sequence starting with c2 (which also implies that there may be no sequence starting with c2).

Example (the same example as in [PBCS], an "unreasonable" one from the https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/map.html website, (apparently the only place where this kind of colouring puzzle can be found); this example requires a whip[7]:




```
(solve 4 30
".YGB..R.....YB...R....R.G..RRB"
1 2 3 4 :
2 3 14 :
3 4 5 14 :
4 5 6 :
5 6 14 15 :
6 7 8 9 15 16 :
8 9 10 19 :
9 16 17 18 19 :
10 11 12 13 19 20 :
11 12 20 :
12 13 20 28 :
14 15 21 22 :
15 16 17 22 :
16 17 :
17 18 19 22 26 27 :
18 19 :
```

```
19 20 27 :
20 28 :
21 22 23 24 :
22 23 24 26 29 :
23 24 25 :
24 25 :
26 27 28 29 : 2
7 28 :
28 30 :
29 30
)
```

There are 30 countries (named 1, …30) and only 4 colours are allowed. The given colours appear in large capital letters in the Figure and they correspond to the sequence appearing as the third argument in the function call.

As for the rest of the arguments (considered as a single list by CLIPS), the first sub-sequence of adjacency data “1 2 3 4” means that country 1 has a common border with each of countries 2, 3 and 4 and none other. You can notice that there is no sub-sequence starting from country 7; the reason is, it has a common border with country 6 only and this has already been declared in the sequence for country 6. Readability aside, all this could be written in a single line (but be careful to keep the “:” separators, whichever way you write these data).

Colours in the data and the solution are displayed as B, Y, R, G, because these are the default names for the first four colours (the full list is B,Y, R, G, O, V, I, 8, 9, 10, …, if you need more than four colours.

I wanted to write a “solve-tatham” function for MapRules, especially as this is the only website I know of that produces those puzzles. But I’ve been unable to understand the Tatham format and I’ve received no answer to an email I sent them. If anyone has information about this, I would be glad to add such a function, which would allow testing many more examples than those one can only copy manually.

13. Miscellanea

13.1 Examples

CSP-Rules includes a set of examples, all of which appear in the "Examples" folder (as expected), which has one sub-folder for each application. The purpose is not to provide an extensive set of cases, but to give you a more concrete idea of the various puzzles and their resolution paths, with more cases than can be mentioned in book form. Nor is the purpose to replace the original websites; it is to direct you to them for more examples of the same kinds.

All these examples have been run with CSP-Rules V2.1 on a MacBookPro from mid-2012 with 16 GB of RAM, so that you can be sure the resolution times you will get on your machines are much smaller than those appearing at the end of the paths (unless your PC is still older). Don't consider these runtimes as absolutes, because some puzzles were solved while I was running other processes and may therefore have taken longer than if they had run alone. Nevertheless, they provide useful approximative indications about the relative times you can expect for each puzzle.

Unless otherwise stated, all the examples have been run with the "standard" configuration of their application.

The "Examples" folder for each application includes several folders, with names corresponding to the origins of the puzzles. Thus, an "ATK" folder means "from the "atksolutions.com" website; a "Puzzle-loop" folder means "from the "puzzle-loop.com" website… Instead of a website, it can be the name of a puzzle creator, e.g. "Tarek". It can also be both, e.g. "Tatham", which means both the person Tatham and the website "tatham.com". If you compare the puzzles from different sources, you will generally observe that they have different characteristics. For instance, the Slitherlink puzzles from puzzle-loop.com slither more than those on grids of same size from kakuro-online.com and they therefore make a more intensive use of the Quasi-loop rules.

See the "README" file of the "Examples" folder of each application for more specific details.

The resolution paths given in all the examples are the raw output from CSP-Rules, with all the printout options on. Of course, if such a path was to be published in a book, some manual cleaning would be necessary. I am thinking of Slitherlink in particular (but not only), where interminable sequence of whips[1] appear. Most of

these could be considered as trivial and merely deleted, keeping only assertions of H, V and P values. Similarly, long sequences of colour rules could be replaced by a single line "colours => xxx in, yyy out, … $H_{ij} = 0$, … $V_{kl} = 1$…"

13.2 Questions and answers

Any question can be asked via GitHub or via any publisher of any of my books.

One interesting question asked by a reader of CSP-Rules is: "In a single sentence, of all the ideas introduced in your books and implemented in CSP-Rules/SudoRules, which would you name as the most innovative?"

As this may help clarify the [PBCS] and CSP-Rules approach, I'll answer it with a single character, after listing possible contenders:

- Contender #1: the idea that solving Constraint Satisfaction Problems could be done in a general pattern-based way (the standard approach to CSP is based on looking for fast algorithms); the whole framework for doing it: epistemic logic (here reducible to intuitionnistic logic), with a formal definition of candidates, resolution rules (with precise conditions sufficient to prove their conclusion generically, once and for all, without any additional ad hoc reasoning in each instantiation), resolution theories, resolution paths within these theories, all this allowing to prove precise results. Not sure it's a main contender, as all the rest can be understood informally without it, but it remains the fundamental theoretical background for all the rest.
- Contender #2: the view of a pattern, especially a chain pattern, as a "static" pattern visible on the current resolution state; this view, based only on CSP-Variables and (direct contradiction) links, was since the beginning and still is in a radical opposition to the classical view of a chain as a "chain of inferences". Of course, a "static" chain pattern can be the support of inferences, but it is not, in and of itself, a chain of inferences. See #3 to #5 for very concrete consequences of this.
- Contender #3: the idea of the z- and t-candidates in chains. However, in isolation, this is not enough to make it the main innovative idea: in a sense, T&E implicitly uses z- and t- candidates; similarly, Sudoku Explainer (SE) implicitly uses z- and t- candidates. The innovative idea is that such candidates are not part of the chains as I've defined them. This is justified on two grounds: firstly, because z- and t-candidates have no impact on the possibilities of extending a partial-chain; secondly, because they can disappear before the chain is applied, without changing it in the least. A practical consequence is, any whip/braid... can be drawn exactly as a bivalue-chain with no added, useless complications; as a result, a very long whip can be written as two lines of text, when it will require more than two pages to write the corresponding chain in SE.

- Contender #4: some specific types of chains: whips (the most important of all the chains), g-whips, braids, g-braids; the precise definition of T&E (a procedure that is neither BFS nor DFS); the clear relationship between T&E, braids, whips or between gT&E, g-braids, g-whips. Such precisely defined chains have very specific properties wrt to the vague "contradiction chains [of inferences]" and considering that they can be reduced to such chains would be utter nonsense.
- Contender #5: the definition of the complexity of a pattern as the number of CSP-Variables its definition involves. This applies consistently to Subsets, any types of chains, any exotic patterns...

  This definition heavily relies on the previous 4 points, it can be formulated in pure logic terms, which guarantees the essential property that the associated ratings are invariant under isomorphism.

  Notice that this point is what distinguishes most radically the various ratings defined in [PBCS] from the "number of nodes" used in Sudoku Explainer and all the other solvers based on "(forcing) contradiction chains [of inferences]". This is also what makes any claim (by people who don't really understand them) that whips or braids can be reduced to such chains or forcing nets utter nonsense.
- Contender #6: the simplest-first strategy. This can be considered as a new search strategy, based on the notion of complexity of a pattern defined in point #5.
- Contender #7: the definition of the confluence property and the various theorems about several resolution theories having it. These theorems allow to find the simplest solution after following a single resolution path. This is the most important results in practice and they justify using the simplest-first strategy. This fundamentally relies on point #5.
- Contender #8: the introduction of additional CSP-Variables. In Sudoku, this appears as the super-symmetric view, in which all the types of CSP-Variables (rc-, rn-, cn, and bn-) are considered on the same footing. This is a very natural idea in the two cases where it is applied: variables "dual" to the natural ones (as the rn, cn and bn variables in Sudoku, dual to the rc ones) or variables allowing to re-write the naturally given constraints as binary ones (e.g. the sum constraints in Kakuro re-written using the hrc and vrc variables).

This is just a quick list of contenders, without thinking too much about it. With more time, I should be able to find more. But let's do with this. Notice that all these ideas were already in [HLS, 2007].

Now my answer to the question in a single character, as promised : 5.

13.3 Reporting bugs

Any bug found in CSP-Rules should preferably be reported via GitHub.

13.3 Improving CSP-Rules

The present manual does not say anything about how to modify CSP-Rules. Until I have written further manuals, you can contact me via GitHub if you want to discuss some improvements, extensions or modifications.

One way of improving the speed of the generic part and of all the applications would be to extend CLIPS. CLIPS lacks complex data structures such as (sparse) binary matrices or low density graphs that would allow to have a more efficient implementation of binary links with a quick access to them.

See the “CONTRIBUTING” file on github.com for other suggestions.

References

Books and articles

Note: the starred publications can be found in the CSP-Rules-V2.1 “Publications” folder.

[Apt 2003]: APT K., *Principles of Constraint Programming*, Cambridge University Press, 2003.

*[Berthier 2007a]: BERTHIER D., *The Hidden Logic of Sudoku*, First Edition, Lulu.com Publishers, May 2007.

*[Berthier 2007b]: BERTHIER D., *The Hidden Logic of Sudoku, Second Edition*, Lulu.com Publishers, November 2007.

*[Berthier 2008a]: BERTHIER D., From Constraints to Resolution Rules, Part I: Conceptual Framework, *International Joint Conferences on Computer, Information, Systems Sciences and Engineering (CISSE 08)*, December 5-13, 2008, Springer. Published as a chapter of *Advanced Techniques in Computing Sciences and Software Engineering*, Khaled Elleithy Editor, pp. 165-170, Springer, 2010.

*[Berthier 2008b]: BERTHIER D., From Constraints to Resolution Rules, Part II: chains, braids, confluence and T&E, *International Joint Conferences on Computer, Information, Systems Sciences and Engineering (CISSE 08)*, December 5-13, 2008, Springer. Published as a chapter of *Advanced Techniques in Computing Sciences and Software Engineering*, Khaled Elleithy Editor, pp. 171-176, Springer, 2010.

*[Berthier 2009]: BERTHIER D., Unbiased Statistics of a CSP - A Controlled-Bias Generator, *International Joint Conferences on Computer, Information, Systems Sciences and Engineering (CISSE 09)*, December 4-12, 2009, Springer. Published as a chapter of *Innovations in Computing Sciences and Software Engineering*, Khaled Elleithy Editor, pp. 11-17, Springer, 2010.

*[Berthier 2011]: BERTHIER D., *Constraint Resolution Theories*, Lulu.com Publishers, October 2011.

*[Berthier 2012]: BERTHIER D., *Pattern-Based Constraint Satisfaction and Logic Puzzles*, Lulu.com Publishers, July 2012.

*[Berthier 2015]: BERTHIER D., *Pattern-Based Constraint Satisfaction and Logic Puzzles (Second Edition)*, Lulu.com Publishers, November 2015.

[CRT]: abbreviation for [Berthier 2011].

[Dechter 2003]: DECHTER R., *Constraint Processing*, Morgan Kaufmann, 2003.

[Freuder et al. 1994]: FREUDER E. & MACKWORTH A., *Constraint-Based Reasoning*, MIT Press, 1994.

[Früwirth et al. 2003]: FRÜWIRTH T. & SLIM A., *Essentials of Constraint Programming*, Springer, 2003.

[HLS1], [HLS2], [HLS]: respectively, abbreviations for [Berthier 2007a], [Berthier 2007b] or for any of the two.

[Guesguen et al. 1992]: GUESGUEN H.W. & HETZBERG J., *A Perspective of Constraint-Based Reasoning*, Lecture Notes in Artificial Intelligence, Springer, 1992.

[Kumar 1992]: KUMAR V., Algorithms for Constraint Satisfaction Problems: a Survey, *AI Magazine*, Vol. 13 n° 1, pp. 32-44, 1992.

[Lecoutre 2009]: LECOUTRE C., *Constraint Networks: Techniques and Algorithms*, ISTE/Wiley, 2009.

[Marriot et al. 1998]: MARRIOT K. & STUCKEY P., *Programming with Constraints: an Introduction*, MIT Press, 1998.

[Newell 1982]: NEWELL A., The Knowledge Level, *Artificial Intelligence*, Vol. 59, pp 87-127, 1982.

[PBCS1], [PBCS2], PBCS]: respectively, abbreviations for [Berthier 2012], [Berthier 2015] or for any of the two.

[Riley 2008]: RILEY G., *CLIPS documentation*, 2008, available at http://clipsrules. sourceforge.net/OnlineDocs.html.

[Rossi et al. 2006]: ROSSI F., VAN BEEK P. & WALSH T., *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, Elsevier, 2006.

[Stuart 2007]: STUART A., *The Logic of Sudoku*, Michael Mepham Publishing, 2007.

[Van Hentenryck 1989]: VAN HENTENRYCK P., *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

Websites

[Angus www]: ANGUS J. (Simple Sudoku), http://www.angusj.com/sudoku/, 2005-2007, the classical reference for the most basic Sudoku techniques.

[askmarilyn www]: http://www.parade.com/askmarilyn/index.html, the “official” place for Numbrix® puzzles.

[Atkin & al. 1983]: https://www.sciencedirect.com/science/article/pii/089812218390130X

[atksolutions www]: http://www.atksolutions.com, the most interesting source for Futoshiki and Kakuro puzzles.

[Bartlett 2012]:

http://web.math.ucsb.edu/~padraic/mathcamp_2012/latin_squares/MC2012_LatinSquares_lecture6.pdf

[Berthier www]: BERTHIER D., https://denis-berthier.pagesperso-orange.fr. This is where supplements to my books can be found.

[Brouwer 2006]: BROUWER A., Solving Sudokus,

https://homepages.cwi.nl/~aeb/games/sudoku/, 2006.

[CLIPS core]: https://sourceforge.net/p/clipsrules/code/HEAD/tree/branches/63x/core/

[edhelper www]: http://www.edhelper.com/puzzles.htm, a website with instances of various difficulty levels for many different logic puzzles.

[Hodoku www]: http://hodoku.sourceforge.net, one of the best Sudoku solvers, with an explanation of the Sudoku specific rules.

[Mebane 2012]: MEBANE P.,

https://mellowmelon.files.wordpress.com/2012/05/pack03hidato_v3.pdf, the hardest and most interesting Hidato® puzzles.

[Nikoli www]: http://www.nikoli.com/, Probably the most famous reference in logic puzzles.

[slinker www]: https://github.com/timhutton/slinker (last version as of this writing: 0.1.1).

[Smithsonian www]: http://www.smithsonianmag.com/games/hidato.html, the "official" place for Hidato® puzzles.

[Sterten www]: STERTEN (alias dukuso), http://magictour.free.fr/sudoku.htm, a famous collection of hard Sudoku puzzles;

[Sterten 2005]: STERTEN (alias dukuso), *suexg*, http://www.setbb.com/phpbb/viewtopic.php?t=206&mforum= sudoku, 2005, a top-down Sudoku generator;

[Tatham www]: http://www.chiark.greenend.org.uk/~sgtatham/puzzles/, one of the classical references in logic puzzles, with generally rather easy instances.

[Werf www]: van der WERF R., Sudocue, Sudoku Solving Guide, http://www.sudocue.net/guide.php, 2005-2007.