Augmented User Manual

for

CSP-Rules V2.1

(release r3)

Denis Berthier

Augmented User Manual
for
CSP-Rules V2.1

(release r3)

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 Press, May 2007.
The Hidden Logic of Sudoku (Second Edition), Lulu Press, November 2007.
Constraint Resolution Theories, Lulu Press, November 2011.
Pattern-Based Constraint Satisfaction and Logic Puzzles (First Edition), Lulu Press, November 2012.
Pattern-Based Constraint Satisfaction and Logic Puzzles (Second Edition), Lulu Press, July 2015.
Basic User Manual for CSP-Rules-V2.1 (First Edition), Lulu Press, August 2020.
Basic User Manual for CSP-Rules-V2.1 (Second Edition), Lulu Press, November 2021.
Pattern-Based Constraint Satisfaction and Logic Puzzles (Third Edition), Lulu Press, November 2021.

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

This is the third, updated release of the book referenced below:
Dépôt légal : November 2022

ISBN: 978-1-4710-2687-4

9 781471 026874

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 can only deal 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 ones (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]). Part III of this Manual will show that several alternative "advanced" 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 resolution 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 (as of now) in the third edition of [PBCS] ([PBCS3], November 2021).

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 "Augmented User Manual for CSP-Rules V2.1" is designed as a supplement to [PBCS], although only a general understanding of [PBCS] is required*. It is called "Augmented" because it is an extended (and revised) version of the previous "Basic User Manual" ([BUM]) that it replaces.

"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.]

As I added new features to CSP-Rules, many of them being first applied to Sudoku, the SudoRules chapter grew too big in the updated versions of the [BUM], with parts that were no longer basic at all. That's why I decided to split this chapter into the original one and a new Third Part for more advanced topics.

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; the recently discovered Tridagons. 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 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 strateg*y (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 universal but 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 $B^{*......}$-braids – unless some exotic pattern allows to reduce the puzzle to a lower T&E (see chapter 14).

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 at the boundaries of the visible universe. Without using T&E or DFS, 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.

As for CSP-Rules being a research tool, see Part III for a detailed description of its recently added features and how it participated in the research about the newly found Tridagon pattern and associated Sudoku puzzles in T&E(3).

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 $C_{i,j}$'s.* 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.

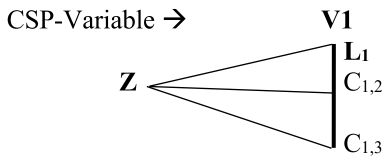


Figure 1.1. *Symbolic representation of a whip[1]*

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: $V_1\{L_1\ .\}$ => 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: $r_3n_4\{c_5\ .\}$ => $r_2c_6 \neq 4$. 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 = $n_4r_2c_6$.

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].

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.*

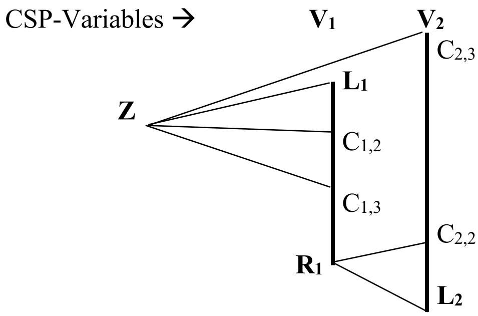


Figure 1.2. *Symbolic representation of a whip[2]. A partial-whip[2] would have in addition one pending candidate for V_2: R_2 (as in Figure 1.3)*

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.

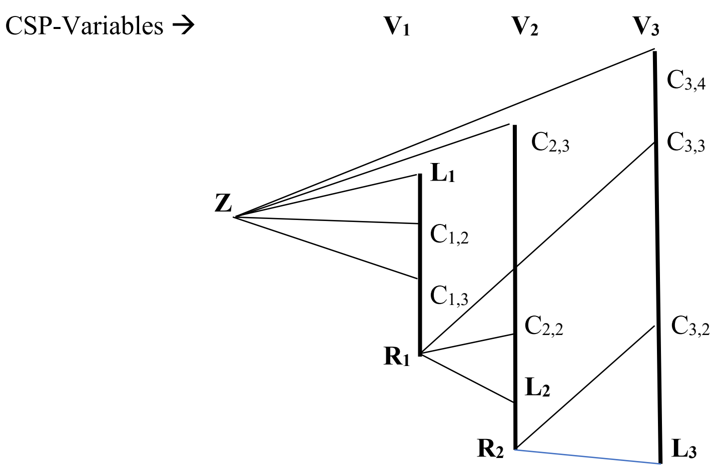


Figure 1.3. *Symbolic representation of a whip[3]*

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.

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].

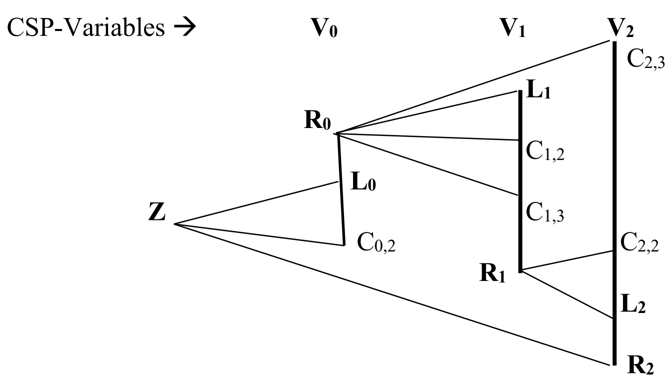


Figure 1.4. *Symbolic representation of a t-whip[3]*

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.

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 covid lockdown in 2020 in France to concentrate on writing the necessary additions – e.g. powerful but easy to use configuration files with related loaders – and on writing the first edition of this Manual to make it usable by others than me. Every bad has its good counterpart.) Notice that "research tool" doesn't mean lower quality; on the contrary I claim much higher quality than many "professional" softwares; CSP-Rules has not been designed for requiring maintenance fees.
- 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), much depending on their inherent branching factors.

- 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.
- Since its first publication in June 2020, CSP-Rules-V2.1 has had a good number of updates. However, the version number (2.1) hasn't changed because the core of the system hasn't changed (it had been developed and largely tested on millions of puzzles long before being made public). The changes are mainly additions, some generic ones (Oddagons, OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips) that are conceptually trivial extensions of the rules defined in [PBCS], and some application-specific (e.g. Pandiagonals in Latin squares or all the Sudoku-specific parts related to strategies other than simplest-first or to the newly discovered tridagon pattern.)

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.
- *Names of variables always start with a "?"*. Names of global variables 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 a real argument, the size of the grid
   Hidato ; I am a real argument, 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 (apparently) 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 from a rational only point of view). Just for fun, you can play with the difference between the following two commands:

```
(printout t “; I am a string” crlf)
(printout t “; I am a string
” crlf)
```

By the way, if you are blocked after typing something like the above first command (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 when batch solving full files of puzzles.

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 (e.g. Xcode on a Mac, VS Code on Windows or Emacs or BBEdit on any platform).

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 four 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. delete the semi-colon here:

```
; (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 r823 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-r823")
```

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 not too old version of MacOS, you will have to recompile your own version of the CLIPS core, due to the strict MacOS security rules. You may have do do this also on some other versions of Unix. This is why the source of the CLIPS core is now included in the CLIPS folder. If not yet done, install XCode (plus the "additional tools" it requires on first launch). In a Terminal, go to the CLIPS/clips-core/ directory inside CSP-Rules-V2.1 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-do the same installation procedure – unless you keep aside a copy of clips (or clips.exe) and of your configuration files.

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 on 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 it 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-r823, 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***. In my own practice, this has never been a problem.

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

CSP-Rules V2.1 includes the generic resolution rules defined in [CRT] and [PBCS], plus (some generic form of) the more specific rules (“2D-chains”) previously defined in [HLS], plus new rules allowing to deal with OR_k relations between candidates (see section 3.5). 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:

1) Reversible ones (with no t-candidate):

- 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 (see definition in section 3.4);
- g-bivalue chains, up to length 20;

2) Non reversible (with t-candidates) ones (from [HLS], [CRT] or [PBCS1]):

- 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;
- 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;

3) Non reversible, recently added ones that require the definition of some application-specific OR_k relation for doing any effective work; they are intended to deal with exotic patterns that can be used to deduce some OR_k relations, e.g. exotic patterns based on some impossible pattern:

- OR_k-forcing-whips, k = 2, 3, … 6, up to length 36 (see definition in section 3.5.1);
- OR_k-contrad-whips, k = 2, 3, … 6, up to length 36 (see definition in section 3.5.2).
- OR_k-whips, k = 2, 3, … 6, up to length 36 (see definition in section 3.5.3).
- OR_k-forcing-g-whips, k = 2, 3, … 6, up to length 36 (see remarks in section 3.5.4).
- OR_k-contrad-g-whips, k = 2, 3, … 6, up to length 36 (see remarks in section 3.5.4).
- OR_k-g-whips, k = 2, 3, … 6, up to length 36 (see remarks in section 3.5.4).

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 bivalue-chains, oddagons 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.

Independently of their speed or memory versions, some of the resolution rules have two slightly different behaviours:

- the old behaviour that works and prints the output as described in [PBCS1] and [PBCS2] and as in CSP-Rules-V2.0 – i.e. only one elimination at a time;
- and a more recent "blocked" behaviour 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 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 you to revert to the previous ([PBCS1], [PBCS2] and CSP-Rules-V2.0) one if that is what you prefer.

Rules amenable to this new "blocked" behaviour are

- whips[1],
- (typed or not) bivalue-chains, z-chains and t-whips,
- oddagons
- Subset rules in any of the coded applications.

These four groups of rules 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.

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 non-typed 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 Oddagons

Oddagons were not discussed in [PBCS1] or [PBCS2]. They are an oddity among generic chains. They first appeared in the Sudoku world. They can be defined as follows (according to Tarek, translated into the [PBCS] formalism): for any candidate Z (the target of the oddagon), for any odd integer n≥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 possibly not linked to Z.

In such conditions, the *oddagon chain rule* states that Z can be eliminated. The proof is very simple: if Z was TRUE, all the z-candidates would be FALSE and:

– if Z in not one of c_1, c_2, …, c_n, then by circulating clockwise along the chain, c_1 TRUE would imply c_1 FALSE whereas c_1 FALSE would imply c_1 TRUE.

– if Z is e.g. c_1 (still supposed to be TRUE), circulating clockwise along the chain from c_1 as before would imply that c_1 is FALSE, a contradiction.

Notice that nothing in the above definition or proof prevents the target Z from being one of c_1, c_2, … or c_n. In such a case, one will have an *autophage elimination*.

The candidates linked to Z (z-candidates) are also called guardians: one of them must be True in order to prevent the impossible pattern that would result from their absence.

Notice that Oddagons are not very brittle: if a right-linking candidate is deleted, the pattern degenerates into a z-chain for Z; if it is asserted, it degenerates into a whip[1] for Z. It is therefore recommended to activate z-chains when oddagons are activated. Notice that this is not automatically ensured by CSP-Rules loading rules.

3.5 OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips

The chain rules introduced in this section are easy generalisations of the rules defined in [HLS], [CRT] or [PBCS]. They are intended for use with rare exotic patterns that lead to logical OR_k relations, in particular with patterns intended to avoid known contradictory patterns. They could also be used with patterns related to uniqueness in some CSPs, if one accepts the assumption of uniqueness.

3.5.1 OR_k-forcing-whips

OR_k-forcing-whips are a straightforward generalisation of forcing-whips.

OR_k-forcing-whip rules: if an OR relation between k different candidates (Z_1, Z_2 ... Z_k, $k > 1$) has been proven, and if there are k partial-whips (of respective lengths p_1, p_2, ... p_k), with respective targets Z_1, Z_2 ... Z_k, then:

– if the last (right-linking) candidate of all the k partial-whips is the same, then one can assert it as True;

– if a candidate is linked to the ends (i.e. the last right-linking candidates) of all the k partial-whips, then it can be eliminated.

The proof is a trivial application of “reasoning by cases”. (For simplicity of the above formulation, a direct contradiction link between two candidates is considered as a partial-whip[0]).

As k streams of reasoning must be followed in parallel and complexity is more or less exponential in the length of chains, the global length of an OR_k-Forcing-Whip can be defined consistently in two and only two ways:

1) either consider that the OR_k relation doesn't lead to any elimination in and of itself and that it must therefore be counted in the complexity of the OR_k-forcing-whip, in which case the total length must be defined as: (length of the OR_k relation) + p_1 + p_2 + … + p_k; this is the view adopted in standard forcing-whips (based on the simple OR_2 bivalue relation of size 1);

2) or consider that the OR_k relation has been proven independently and doesn't have to be counted in the complexity of the forcing-whip, in which case the total length is $1 + p_1 + p_2 + … + p_k$ (the +1 is for consistency with the basic forcing-whips).

In the definition of the general OR_k-forcing-whips, I have adopted the second view, which corresponds to an extended resolution model, where OR_k relations can be asserted as intermediate results (in addition to only values and candidates).

The reason is, *OR_k-forcing-whips are only intended for use with rare exotic patterns that can be expressed as OR_k relations*. Typically, in such cases, one wants to give some priority to the exotic pattern and to what can be deduced from it (while maintaining some consistency with the complexities of the ingredients, i.e. partial-whips). For examples, see chapter 14.

3.5.2 OR_k-contrad-whips

When an application-specific OR_k relation between k candidates has been proven, the most natural way to take advantage of it is via the above-defined generic OR_k-Forcing-Whips. The OR_k relation serves as a natural starting point for the forcing whip. However, there is also another way of using an OR_k relation to eliminate candidates. As far as I know, the first time a Tridagon OR_k relation has been used in a similar way is by DEFISE, in this puzzle: (http://forum.enjoysudoku.com/23427-more-anti-tridagon-guardians-t40295.html)

The formal definition is almost obvious: ***given an OR_k relation between candidates C_1, C_2, ... C_k, an OR_k-contradiction-whip[n] (or OR_k-contrad-whip[n] for short) based on the OR_k relation, with target Z, is an ordinary partial-whip[n-1] W with the same target Z, such that each of the C_k's is linked to (at least) a right-linking candidate of W (and at least one of the C_k's is linked to the last right-linking candidate of W – continuity condition).***

OR_k-contrad-whip rule: in such a situation, the target Z can be eliminated.

Proof: obvious. If Z was True, all the right-linking candidates would be True; which implies that none of the C_i's could be True. Note that the OR_k relation plays here the same role as an n-th CSP-Variable for reaching a final impossibility (which also justifies the definition of length for the OR_k-contrad-whip).

3.5.3 OR_k-whips

OR_k-whips are an easy generalisation of whips or of contrad-whips. An OR_k-whip is a pattern more general than the previously defined OR_k-contrad-whips: the OR_k part may appear at another place than the final (contradiction) one.

Informally speaking, ***an OR_k-whip[n] (based on some OR_k-relation) is much like a whip[n], except that one (and only one) of its CSP-Variables is replaced by an OR_k relation, where k-1 (and only k-1) candidates are linked to the target or to previous right-linking candidates, the continuity condition of the chain is ensured at this step also, and the remaining one is used as a right linking-candidate for the next step; as a special case, it can also be an OR_k-contrad-whip[n].***

OR_k-whip rule: in such a situation, the target can be eliminated.

Proof: obvious; copy the proof for the OR_k-contrad-whips

3.5.4 OR_kFW_n, OR_kCW_n and OR_kW_n parametric resolution theories

For every $k \geq 1$, one can obviously define "parametric" OR_kFW_n, OR_kCW_n, OR_kW_n and OR_kCH_n resolution theories by:

OR_kFW_n = all the OR_j-forcing-whip[p] resolution rules for $j \leq k$ and $p \leq n$;

OR_kCW_n = all the OR_j-contrad-whip[p] resolution rules for $j \leq k$ and $p \leq n$;

OR_kW_n = all the OR_j-whip[p] resolution rules for $j \leq k$ and $p \leq n$ (which includes all the OR_j-contrad-whip[p] resolution rules for $j \leq k$ and $p \leq n$);

$OR_kCH_n = OR_kFW_n + OR_kW_n$; (which includes OR_kCW_n)

In practice, they will generally be combined with SFin, Trid (the elementary tridagon elimination rule) and some W_q.

Notes:

– OR_k-forcing-whips[1], OR_k-contrad-whips[1] and OR_k-whips[1] are the “same thing” (but considered from different points of view);

– the above theories are said *parametric* because they depend on the resolution rule(s) that produce the OR_k relations; generally, these rules will be clear from the context; but the parameter will always be written in the OR_k rule names when they appear in a resolution path, such as Trid-OR3-whip[n] when the OR3 relation is justified by an anti-tridagon (see chapter 14 for details and for the notation);

OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips are collectively called ***OR_k-chains***. They *are intended for use only with* ***exotic*** *OR_k relations* – which is the only possible justification for not counting the real OR_k length in their total length).

Note that in CSP-Rules, OR_k-forcing-whips are restricted in two complementary ways:

– directly by their maximal allowed length (defined by ?OR_k-forcing-whips-max-length*);*

*– and indirectly by the maximal lengths of their inner partial-whips (deduced from ?*whips-max-length* or ?*all-chains-max-length* or ?*OR_k-whips-max-length*);*

*But OR_k-contrad-whips and OR_k-whips are only restricted by their respective ?*ORk-contrad-whips-max-length* and ?*OR_k-whips-max-length*.*

The reason in all cases is to maximise the use of partial-whips once they are found.

3.5.5 Extensions to g-candidates and to braided continuity and braided-g-continuity

The above definitions of OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips have straightforward extensions to accommodate 1) g-candidates as possible right-linking objects, and 2) braided or g-braided continuity. Notice that, as OR_k relations are not based on CSP-Variables, the OR_k steps in these new chains can only have candidates as their right-linking elements (OR_k relations have no g-candidates).

Corresponding resolution theories can also be defined in obviously similar ways.

3.5.6 Managing the brittleness of the OR_k relations: ultra-persistency

Exotic patterns are often *brittle*: if some of their candidates are deleted or asserted as TRUE, the pattern may degenerate into something that one may no longer be able to identify – whence possible problems on non-confluence for resolution theories based on them. This remark will apply to Tridagons in chapter 14. However:

– the above resolution theories are defined in such a way as to minimise problems on non-confluence that are not directly due to the OR_k relation;

– CSP-Rules has generic rules for *systematically making an OR_k-relation ultra-persistent* once it has been found: if one or more of the OR_k-related candidates (“guardians”) is eliminated – which makes the relation *a priori* unusable for the OR_k-chain rules (e.g. because this candidate was the target of a required partial-whip and there can be no partial-whip with no target) –, an ORj-relation for some j smaller than k will automatically be asserted for the remaining guardians, even if the original pattern underlying the OR_k-relation has become degenerated (and could therefore not be found directly). If k=2, the only remaining candidate is asserted. (See section 14.16 for an explanation and illustration of this.)

3.5.7 Splitting the OR_k relations

In some situations, potentially interesting OR_k relations appear between large numbers of candidates (i.e. k is large). However, the larger k, the more difficult it can be to find useful OR_k-chains of “reasonable” lengths. CSP-Rules has a set of generic rules to manage the simplest cases of this situation.

Let us first define a ***c-chain[n]*** as a sequence of n+1 candidates $C_1, \ldots, C_{n+1}$ such that, for each $1 \le p \le n$, there is a bivalue relation between C_p and C_{p+1} for some CSP-Variable csp_p (with all the csp_i’s different). It is obvious (exercise for the reader) that in such a case, if n is even, C_1 and C_{n+1} must have the same truth value.

It is also obvious that, if there is some OR_k relation between k candidates $Z_1, \ldots Z_k$, and if there is some c-chain[n] between two of the Z_i’s, say Z_a and Z_b, then the OR_k relation is equivalent two OR_{k-1} relations, with respective lists of candidates: $\{Z_1, \ldots Z_k\}-\{Z_a\}$ and $\{Z_1, \ldots Z_k\}-\{Z_b\}$.

It is finally obvious that for any OR_k-chain based on this OR_k relation, allowing some elimination Z, there will be at at least one shorter OR_{k-1} chain based on either of the two OR_{k-1} relations and also allowing to eliminate Z.

CSP-Rules has generic “OR_k splitting rules” implementing this idea, for $k \le 12$ and $n \le 10$.

Notice that this mechanism doesn’t claim to be an exhaustive way of reducing the number of guardians in OR_k-relations (in addition to OR_k ultra-persistency). The only purpose here is to have generic and easy rules – much simpler than the OR_k-chain resolution rules.

Together, the OR_k ultra-persistency rules and the OR_k splitting rules are called OR_k-relations management rules or OR_k management rules.

3.6 Rules for simulating T&E (Trial and Error)

CSP-Rules V2.1 also provides a rule-based version of T&E (as defined in [PBCS]) 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. Roughly speaking, T&E(T, 2) will 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 a 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.

3.6.1 Restricting the candidates tried in T&E (at any depth)

Global variable ?*restrict-TE-targets* is FALSE by default and, at any depth, T&E tries all the candidates present in the current resolution state. But when this variable is set to TRUE by some function, only candidates for which predicate (is-restricted-TE-target 0 ?candidate) has been asserted as TRUE will be tried (at any depth of T&E). Obviously, this is only an incomplete form of T&E.

See chapter 14 for an example of how this allows to ***use SudoRules as an assistant theorem prover*** to show that most contradictory 3-digit patterns don't need more than a given depth of T&E to be proven impossible. This will also allow to single out (very rare) patterns that may require T&E-depth 3 in Sudoku. Using such restrictions on T&E may be justified when some abstract pattern makes assumptions only on a limited number of cells; one may tentatively assume that all the other cells do not to participate in any general proof about this pattern and that trying their candidates would be a pure waste of time. (Notice that this is computationally extremely efficient as a filter, but a negative answer has to be confirmed by the full T&E procedure).

3.7 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.

The candidates used as hypotheses in DFS can now be restricted in the same way as in T&E.

3.8 Rules for finding backdoors and anti-backdoors

If T_0 is a resolution theory with the confluence property, one can define a T_0-backdoor [respectively a T_0-anti-backdoor] as a candidate that leads to a solution in T_0 if it is asserted as a value [resp. deleted from the remaining candidates]. Similarly, a T_0-anti-backdoor-pair is a couple of candidates leading to a solution in T_0 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 anti-backdoor-pair found must contain at least one of them), work in the current resolution state and with T_0 chosen as the current set of ordinary resolution rules.

T_0-anti-backdoors and T_0-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. See section 6.10 for reasonable choices of T_0.

3.9 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 partial-whips (resp. partial-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. More extensive data don't fundamentally 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 definitions and results to any theory T_0 with the confluence property.

I will not insist on this, because all three points are obvious consequences or extensions of my various definitions and of the 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 is some recent 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.

Also contrary to forcing-whips/braids, Forcing-T&E is not submitted to the simplest-first strategy. But it allows some (often drastic) reduction of the number of steps in the resolution path. You can see an example of application in section 13.10. 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 reduced 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", of the *steepest-descent* kind, with the number of candidates as the evaluation function, is purely local and, like any steepest-descent method, it isn't guaranteed to produce the shortest resolution path.

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.

The following generic functions (with no argument) apply respectively Forcing-T&E and Forcing{3}-T&E once (and only once) in the current resolution state: ***"apply-F2TE"***, ***"apply-F3TE"***. You can apply any of them any number of times. Notice that the set of rules (T_0) loaded into CLIPS must be restricted to those you want to consider as no-step (see section 13.1.1).

Being a combination of T&E and reasoning by cases, Forcing-T&E and Forcing{3}-T&E can be considered as the nukes of solving and the summum of inelegance.

3.10 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).

3.11 The most general generic functions in CSP-Rules

The following functions have a generic definition, according to which they do nothing; each application has to provide 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***"),

- "***solve-w-preferences***" (abbreviated as "***solve-w-prefs***"); see section 6.8.1 for the meaning,
- "***try-to-eliminate-candidates***" uses the rules loaded into CLIPS and tries to eliminate only the chosen candidates. This is a general focusing feature, available in all the applications without any further coding (see section 6.9 for details).
- "***mute-print-options***" sets to FALSE all the global variables that control what's printed (i.e. ?*print-actions*, ?*print-solution*, ?*print-RS-after-Singles*, ?*print-RS-after-whips[1]*, ?*print-final-RS*, ?*print-main-levels *, ?*print-levels * – see section 4.4.1 for the definition of these variables – plus other control variables related to T&E).
- As expected, "***restore-print-options***" restores their original values (selected in the configuration file). This may be very convenient in order to avoid too long printings when you're only interested in the final results. Notice that if you change any of the above variables manually instead of with "mute-print-options", you cannot use "restore-print-options" to restore its original value; you must do it manually.

3.12 Generic global variables

When an application (such as Sudoku or Latin Squares) has (properly coded) functions for dealing with a sequence of instances (e.g. stored in a file, one per line), there are two global variables of interest at the end of applying them. They are particularly useful when sorting large sets of instances according to some rating/classification system. At the end of calculations, just type the name of one of these variables (and carriage return / line feed) to get the corresponding list:

- ***?*solved-list**** is the list of all the instances (by their place number in the sequence) that have been solved with the current set of rules;
- ***?*not-solved-list**** is the (complementary) list of all the instances (by their place number in the sequence) that have not been solved with the current set of rules;
- ***?*no-sol-list**** is the list of all the instances that have been proven contradictory with the current set of rules (normally empty if the instances are consistent).

There is also a global variable that can be used to define a list of candidates that will automatically be eliminated at the start of resolution, i.e. before any resolution rule is applied:

- ***?*simulated-eliminations****

It must be defined before any "init" or "solve" function is applied, e.g. by:
(bind ?*simulated-eliminations* (create$ $cand_1$ $cand_2$ …)), where the $cand_i$'s are candidates ("create$" is the list creation function in CLIPS).
It is very convenient when one wants to study the consequences of a new, not yet coded rule – or just the effect of deleting some candidate(s) for some reason.

3.13 Appendix 3A: The T&E-depth of a resolution rule

The T&E procedure provided with CSP-Rules can be used to compute more than the solution of a puzzle. Section 14.7, will show how to use it for analysing k-digit patterns. Here is a more general use case that will be useful to understand how some techniques or rules can lower the complexity of a puzzle.

3.13.1 Definitions:

Definition: ***the T&E-depth of a resolution rule R*** is the depth n of the T&E(BRT, n) procedure required to prove R, i.e. to prove all the expected eliminations/assertions of R in any resolution state where the pattern P and only the pattern P of R is assumed to be satisfied.

It is also informally the T&-depth you need to use in order to replace the use of R in an instance of the CSP in the most general situation.

Remarks:
1) as all my definitions, this is invariant under Sudoku isomorphisms; this is essential, as it will allow drastic simplifications in the computation of the T&E-depth of any particular resolution rule;
2) some particular cases of the rule (i.e. of its pattern) may require fewer levels of T&E, but that doesn't change the depth of the rule itself, as the definitions says: "in any resolution state";
3) some particular eliminations/assertions may also require only fewer levels, but that doesn't change the depth of the rule itself, as the definition says: "all the expected eliminations/assertions";
(about remarks 2 and 3, see the Naked Quads example below);
4) in a given puzzle, the applicability of a resolution rule R of T&E-depth n doesn't imply that the T&E-depth of the puzzle is (at least) n. There are always many resolution paths and some of them may totally avoid using R or may use only a special case of R;
5) a similar definition, with similar remarks could be based on gT&E(n) = T&E(W1, n) instead of T&E(n).

Definition: ***the T&E-depth of a contradictory pattern P*** is the depth n of the T&E(Singles, n) procedure required to prove that R is contradictory.

It is also the T&E-depth of the resolution rule: P => FALSE (where FALSE can be considered as an abbreviation for any formula "CSP1 = a AND CSP1 ≠ a" in the language of the CSP).

3.13.2 Elementary Examples

A Single has T&E-depth 0 (by definition of T&E).

A braid has T&E-depth 1 (independent of its length). All the special cases of braids (bivalue-chains, z-chains, t-whips, whips) have T&E-depth 1.

A B-braid has T&E-depth 2 (independent of its total length and of the lengths of its inner braids).

Note: a g-braid or g-whip has T&E-depth 2, but g-T&E-depth 1.

3.13.3 The Naked-Quads-in-a-row example

Before dealing with more complex patterns, let's see how this applies to the most familiar ones: Subsets in Sudoku.

a) Method for computing the T&E-depth of a resolution rule:

The following technique can be applied in many cases, but it should be enough to illustrate it with the Naked Quads example. First, and this is a situation that will occur most of the time: we know in advance what the targets can be. As a result, we can assume that the first level of T&E consists of supposing that one of the targets is True (proof by contradiction). The most general resolution state is:

```
+----------------+----------------+----------------+
! 1234 1234 #    ! 1234 #    #    ! 1234 #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
+----------------+----------------+----------------+
! #    #    #    ! #    #    #    ! #    #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
+----------------+----------------+----------------+
! #    #    #    ! #    #    #    ! #    #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
! #    #    #    ! #    #    #    ! #    #    #    !
+----------------+----------------+----------------+
```

where # is just a notation for the presence of all the candidates 123456789 in a cell and is equivalent to nothing being supposed about it.

Notice that:
1) we suppose that the 4 candidates are present in the 4 cells (this is the hardest possible case, because T&E is monotonic with respect to eliminations);
2) in order to be complete, we should also analyse the different possible positions for the 4 cells of the Quad in the row; but I'll let the other cases as an easy exercise for the reader.

As for the target, by digit symmetries, we can assume it is number 1 in some of the other cells of the row. Again, there are several possibilities for the cell, but I shall analyse only one.

So, at the first level of T&E, we assume the target (we don't repeat the lower two bands):

```
+-------------+-------------+-------------+
! 234 234 1   ! 234 #   #   ! 234 #   #   !
! #   #   #   ! #   #   #   ! #   #   #   !
! #   #   #   ! #   #   #   ! #   #   #   !
+-------------+-------------+-------------+
```

Level 2 of T&E: assume that r1c1=2 (it can't be 1); by digit symmetries, there's no loss of generality:

```
+----------+----------+----------+
! 2  34 1  ! 34 #  #  ! 34 #  #  !
! #  #  #  ! #  #  #  ! #  #  #  !
! #  #  #  ! #  #  #  ! #  #  #  !
+----------+----------+----------+
```

Level 3 of T&E: assume that r1c2=3 (it can't be 1 or 2); again, by digit symmetries, there's no loss of generality:

```
+-------+-------+-------+
! 2 3 1 ! 4 # # ! 4 # # !
! # # # ! # # # ! # # # !
! # # # ! # # # ! # # # !
+-------+-------+-------+
```

The contradiction appears: number 4 has to be in two cells of row r1.

Conclusion (after checking all the other possibilities): The T&E-depth of the Naked-Quads rule is 3.

b) Special case:

As an extreme example of remark 2 in section 3.13.1, consider the cyclic special case of the NQ:

```
+----------+----------+----------+
! 12 23 #  ! 34 #  #  ! 41 #  #  !
! #  #  #  ! #  #  #  ! #  #  #  !
! #  #  #  ! #  #  #  ! #  #  #  !
+----------+----------+----------+
```

It is obvious that assigning 1 to any of the other cells in the row will immediately result in a contradiction in BRT. So, this special case has T&E-depth 1. This is also consistent with the possible view of this pattern has a bivalue-chain[4].

But this special case doesn't change the T&E-depth of the general NQ.

Exercises for the reader:
1) Prove that the T&E-depth of the Naked Triplets rule is 2.
2) Prove that the T&E-depth of the Naked Pairs rule is 1.

Remarks:
1) all the above results can be extended by super-symmetry to Hidden and Super-Hidden Subsets (Fish patterns);
2) for an example of a Swordfish that brings down a puzzle from T&E(2) to T&E(0), i.e. to Singles, see section 8.8.2 of [PBCS]. Notice that the Swordfish resolution rule has T&E-depth 2, but most puzzles that have its pattern are indeed only in T&E(1): as I've shown in [PBCS], most cases of Subsets are subsumed by whips.

3.13.4 The Tridagon elimination rule example:

The "tridagon elimination rule" will be introduced in section 14.2. If you follow the proof of eliminations given there (reviewing all the possible cases modulo isomorphisms, you will see that it also proves that the tridagon elimination rule has T&E-depth 4.

Tridagons were my original motivation for defining the T&E-depth of a resolution rule.

Notice that the Tridagon elimination rule has T&E-depth 4, but all the known puzzles that have its pattern are in T&E(3).

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 following 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 drastically circumscribed in various ways by using functions or combinations of functions more elaborate than "solve". For instance, it is possible to focus the eliminations on particular candidates and/or to prefer groups of rules (e.g. Subsets) over others. 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 in practice 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 in SudoRules, but most of it could easily be re-written in generic form or adapted to any other application.

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-Rules 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 L_n being defined by the number n of CSP-Variables involved in the defining pattern (i.e. the conditions of the rule).

Within each level, there are indeed not many different priorities; such 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).

4.1.1 The basic ordering within level L_n

For every n > 1, one has the basic ordering, which is the only thing you need to know unless you deal with OR_k-chains.

oddagon[n] > biv-chain[n] > z-chain[n] > t-whip[n] > whip[n] >
> g-bivalue-chains[n] > g-whip[n] >
> braid[n] > g-braid[n] >
> forcing-whip[n] > forcing-gwhip[n] > forcing-braid[n] > forcing-gbraid[n].

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, forcing-gwhips, forcing-braids and forcing-gbraids was not mentioned in [PBCS], but at each level, they naturally come at the end of this hierarchy.

Being a rare generic pattern, oddagons[n] (when defined, i.e. for odd n) are placed before any other generic chain of same length. (One may debate whether they should be placed after bivalue-chains[n] or even after z-chains[n]; in practice, it doesn't change fundamentally the results.)

Notice that z-chains[n] are placed before t-whips[n], 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 (and they are reversible).

Notice also that g-whips[n] are placed before braids[n], 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.

4.1.2 The extended ordering within level L_n

With the addition of OR_k chain rules, new priorities have been defined, making each level L_n slightly more complex:

oddagon[n] > biv-chain[n] > z-chain[n] > t-whip[n] > whip[n] >

> c-chain[n] (n even) >

> OR_2-forcing-whip[n] > OR_2-contrad-whip[n] > OR_2-whip[n] >
> OR_3-forcing-whip[n] > OR_3-contrad-whip[n] > OR_3-whip[n] > …
> OR_6-forcing-whip[n] > OR_6-contrad-whip[n] > OR_6-whip[n] >

> g-bivalue-chains[n] > g-whip[n] >

> OR_2-contrad-gwhip[n] > OR_2-gwhip[n] >
> OR_3-contrad-gwhip[n] > OR_3-gwhip[n] >…
> OR_6-forcing-whip[n] > OR_6-contrad-gwhip[n] > OR_6-gwhip[n] >

> braid[n] > g-braid[n] >

> forcing-whip[n] > forcing-gwhip[n] > forcing-braid[n] > forcing-gbraid[n].

Notice that all the OR_k-chains[n] come after whips[n]; g-whips[n] come after all these and OR_k-g-chains[n] come after g-whips[n]. Again, this order may be debated; but this is how it is currently.

Finally, notice that for any k (k=1,2…6), OR_k-forcing-whips[n] are placed before OR_k-contrad-whips[n], in order to conform to my idea that OR_k-forcing-whips are more natural and to my experimental result that they are more powerful (for the same values of k and n). For similarly reasons, OR_k-contrad-whips[n] are placed before their generalisation: OR_k-whips[n]. See section 14.18 for how to place OR_k-forcing-whips[n] after OR_k-whips[n] (and OR_k-forcing-g-whips[n] after OR_k-g-whips[n]). Notice also that all the types of OR_k chains are placed before ordinary forcing-whips, in order to conform to my other idea that they are intended for use in conjunction with OR_k relations derived from exotic patterns.

4.1.3 General remarks

CSP-Rules has a mechanism for allowing an easy addition of application-specific saliences at each level, before or after the generic saliences at the same level. In general, application-specific rules based on n 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 by CSP-Rules 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 extremely hard puzzles (that you never see in any magazine). 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. The fact is, it is currently the only systematic strategy that has ever been formulated (except possibly some exceptions to be reviewed in chapter 13).

Part Two

RUNNING SPECIFIC APPLICATIONS

5. General structure of the configuration files

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

The user functions are totally independent of the choices of rules or settings in the configuration file; they work the same way for any choice. These functions 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 application-specific chapters for more information.

As for the configuration files, they all have the same global 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). 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 add or delete a few leading 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-r823");             <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<


;;; 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 the declarations of 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"))

(defglobal ?*CSP-Rules-Examples-Dir* = (str-cat ?*CSP-Rules* "CSP-Rules-Examples"
?*Directory-symbol*))

;;; 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 (such as the Pandiagonal variant in Latin Squares), 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 (a: Subsets; b: oddagons; c: Whips[1]; d: bivalue-chains, z-chains and t-whips and their typed versions) is now the default behaviour in CSP-Rules and in all its applications: when a pattern in the above 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 four families of rules a, b, c or d), to revert to the old behaviour (until V2.0 and as in [PBCS1] and [PBCS2]), 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 details of the output. The names of the control variables are self-explanatory, except maybe ?*print-main-levels* and ?*print-levels*. They allow to print a line each time the solver enters a new level of the solving process (a level being defined by the number of CSP-Variables involved in the rules), respectively each time it activates a new type of rules or a longer length for a type already activated. All the rules selected by the user in the configuration file (plus implied rules) are loaded at the start, but for efficiency reasons and for limiting memory requirements, 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.) For the very hardest puzzles, it may be useful to display tracking information about these activations when nothing seems to happen (i.e. 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. ?*print-main-levels* and ?*print-levels* are now set to FALSE by default, because most users said they 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],
;;; - for Subsets,
;;; - for bivalue-chains, z-chains and t-Whips (whether typed or not),
;;; - for Oddagons.
;;; However, CSP-Rules allows to revert to the previous behaviour,
;;; independently for each of the above four groups of rules.
;;; Un-comment the relevant line(s) below if you want these rules to be "interrupted"
as all the other rules.
;;; Notice that ?*blocked-Subsets* = TRUE, ?*blocked-chains* = TRUE or ?*blocked-
oddagons* = TRUE
;;; will imply ?*blocked-Whips[1]* = TRUE
; (bind ?*blocked-Whips[1]* FALSE)
; (bind ?*blocked-Subsets* FALSE)
; (bind ?*blocked-chains* FALSE) ; i.e. bivalue-chains, z-chains and t-Whips (whether
typed or not)
; (bind ?*blocked-oddagons* FALSE)
;;; The old interrupted behaviour can be globally selected by ?*unblocked-behaviour*
to TRUE;
;;; (equivalent to setting the above four values to FALSE):
; (bind ?*unblocked-behaviour* TRUE)

;;; 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-main-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 generic resolution rules (or quasi-generic ones – like Subsets), where several groups of options appear (my personal most usual choice is selected by default so that you can start solving puzzles without modifying the configuration – except the installation part);
2) Application-specific resolution rules;
3) T&E-related configurations, where different typical options are proposed;
4) DFS (depth-first-search) options (with or without whips[1 or 2]);
5) Rules based on indirect binary contradictions (not in this public V2.1 version).

The five 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 from sections 3, 4 or 5, you must first comment out my default choices.

5.4.3 Choose ordinary generic and (quasi-generic) resolution rules

First, let me state the rule dependencies (mainly for consistency-preserving reasons) 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 must 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 required 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 required 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).

- For any pattern P and any n > 0: P[n+1] => P[n]
- Subsets => Subsets[4] => Subsets[3] => Subsets[2] => Whips[1]
- FinnedFish => FinnedFish[4] => FinnedFish[3] => FinnedFish[2]
- FinnedFish[k] => Subset[k]
- z-chains => bivalue-chains
- typed-z-chains => typed-bivalue-chains

- 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

- For any OR_k chain, OR_k g-chain, OR_k forcing chain or OR_k forcing g-chain pattern P: OR_6-P => OR_5-P => OR_4-P => OR_3-P => OR_2-P
- For any k, OR_k-whips => OR_k-contrad-whips, which are a special case (note: not => whips)
- For any k, OR_k-contrad-g-whips => OR_k-contrad-whips, which are a special case (note: not => g-whips or whips)
- For any k, OR_k-g-whips => OR_k-whips and OR_k-contrad-g-whips, which are a special case (note: not => g-whips or whips)
- For any k, OR_k-forcing-g-whips => OR_k-forcing-whips, which are a special case (note: not => forcing-whips or forcing-g-whips)

The reason why using OR_k chains[n] does not imply using whips[n] is to allow more freedom for the user when exotic patterns come into play.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 1) Choose ordinary generic or quasi-generic 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)
 (bind ?*z-Chains* TRUE)
 (bind ?*t-Whips* TRUE)

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

;;; Some additional rules I use occasionally:
; (bind ?*Oddagons* 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 almost never use:
; (bind ?*G2-Whips* TRUE)
; (bind ?*Braids* TRUE)
; (bind ?*G-Bivalue-Chains* TRUE)
; (bind ?*G-Braids* TRUE)
```

```
;;; Forcingk-whips are based on k-value cells in any of the 2D-spaces.
;;; Their maximal length is determined by ?*forcing-whips-max-length*.
; (bind ?*Forcing2-Whips* TRUE)
; (bind ?*Forcing3-Whips* TRUE)
; (bind ?*Forcing4-Whips* TRUE)
; (bind ?*Forcing5-Whips* TRUE)
;;; Forcing-Whips are older and simpler than Forcing2-Whips, but they  are the same
thing
; (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)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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.
;;; 36 is the absolute maximum, never reached in practice.
;;; Notice that this global restriction will prevail on any of the individual
restrictions below.
; (bind ?*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 Choose application-specific resolution rules

This part will obviously change from one application to another. It is very developed in SudoRules and almost inexistent in some other applications. Here is part of the Sudoku version. Details will be explained in part III of this Manual.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 2) Choose application-specific resolution rules (besides Subsets)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2.1 Sudoku-specific rules : 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)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2.2 Sudoku-specific rules (besides Subsets): sk-loops and J-Exocets
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

;;; 2.2.1) sk-loops:
; (bind ?*Belt4* TRUE)
; (bind ?*Belt6* TRUE)

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2.3 Sudoku-specific rules : Tridagons and patterns for puzzles in T&E(3)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

This part is much too specific to Sudoku to be described here. See section 14.18.

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 3) 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.
;;; Less printing can make T&E faster.
; (bind ?*print-actions* FALSE)
; (bind ?*print-levels* TRUE)
; (bind ?*print-ECP-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)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3a) 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 1 is enough for almost all the 9x9 Sudokus
;;; - depth 2 is enough except for extremely rare 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 additional next line:
; (bind ?*Whips[1]* TRUE)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3b) For computing the SpB classification
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Choose one of the following forms of T&E(Sp or SpFin, 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)

;;; 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) ; same as ?*Subsets[4]*

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3c) 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)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3d) for computing the BpBB classification for puzzles in T&E(3)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Added after the discovery that there are (extremely rare) Sudoku puzzles in
T&E(3).
;;; Beware that computation times can be very long for puzzles not in T&E(Bp, 2).
;;; Choose one of the following forms of T&E(2):
; (bind ?*TE2* TRUE) ;;; for T&E at level 2
;;; For T&E at level 2, with priority for bivalue variables, add the following:
```

```
; (bind ?*special-TE* TRUE)

;;; Choose p (here p = 2) (larger values of p should not be necessary for 9x9 Sudoku
; (bind ?*Whips* TRUE)
; (bind ?*Braids* TRUE)
; (bind ?*whips-max-length* 2)
; (bind ?*braids-max-length* 2)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3e) for looking for backdoors, anti-backdoors or anti-backdoor pairs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Choose one or several of backdoors, anti-backdoors and 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)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3f) for solving with Forcing-T&E
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; For Forcing-T&E OR Forcing{3}-T&E OR Forcing{4}-T&E, activate any of the following
;;; (Several can be activated at the same time.)
; (bind ?*Forcing{2}-TE* TRUE)
; (bind ?*Forcing{3}-TE* TRUE)
; (bind ?*Forcing{4}-TE* TRUE)

;;; for Forcing-T&E(W1) or Forcing{3}-T&E(W1) or Forcing{4}-T&E(W1), add:
; (bind ?*Whips[1]* TRUE)
;;; for Forcing-T&E(S) or Forcing{3}-T&E(S) or Forcing{4}-T&E(S), add:
; (bind ?*Subsets* TRUE)
```

5.4.6 Decide to try DFS

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 4) Choose DFS (depth-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, also activate this 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. 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 [CRT] and [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 the version of 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 application-specific user functions than the other applications. One thing to remember 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, Unique Rectangles and BUGs, plus exotic patterns: sk-loops, J-Exocets, Tridagons.) Most of these rules (except the uniqueness and exotic ones) 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 – which is anyway irrelevant to the usual user.)

Notice that none of the application-specific 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 (i.e. grid-size>16 or segment-size>4) 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.

This chapter is for basic level Sudoku solving and for some goodies, whereas more advanced techniques (most of them for dealing with additional requirements on the resolution path) and techniques related to recently discovered puzzles in T&E(3) are to be found in chapters 13 and 14, respectively.

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 a line are supposed to be numbered from left to right, the whole numbering going from the first line to the last one. 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 function names, “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........")
```

Beware: when you copy the string from e.g. some Sudoku application or website, be careful not to insert a leading space after the first double quote, as in:

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

On the other hand, you can put anything (e.g. the name of the puzzle, its SER…) after the 81st character; it will not be taken into account.

“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 (which has only values). 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 using 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..12345678.123456.8912345678912345678.123.56.89123456..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..91234567.9123456789123456.89123
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-sukaku-string”):

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

will output the starting resolution state:

```
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: until recently, all the known Sudoku puzzles were at worst in T&E(2) [we now know a few exceptional cases in T&E(3), see chapter 14], 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, the set of all the arguments given to it is 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).
For SudoRules, the following two forms are exactly the same:

```
(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 . .
)
```

```
(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; technically, "solve-sudoku-grid" is just the combination of "solve-sudoku-list" with another SudoRules function: "clean-grid-list".

6.2.4 The grid format for Sukaku

For Sukakus, function ***"solve-sukaku-grid*"** allows something 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 is 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)
```

When solving collections of puzzles, it may be useful to know the existence of a few global variables: ***?*solved-list*, ?*no-sol-list*, ?*belt-list*, ?*J-exocet-list*, ?*exotic-list****. They keep track respectively of the puzzles that have been solved, that have no solution (i.e. have contradictory givens), that have used some sk-loop, some J-Exocet, or some other exotic pattern. In order to display the contents of a list, just type its name, followed by enter/return.

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.

```
+-------+-------+-------+
! . . . ! 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
```

The puzzle is cbg#109 from the controlled-bias collection.

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

```
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, dealing with pretty-printing and permutations.

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"*** (abbreviated as ***"pretty-print"***) and ***"pretty-print-sukaku-list"***, with respective syntax:

```
(pretty-print-sudoku-string ?sudoku-string)
(pretty-print ?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, the size of the printed grid is adapted to the cell with the largest number of candidates.

6.6.2 Applying isomorphisms

There are a few functions for creating 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 such that the new 1 2 3 floors are the ?f1 ?f2 ?f3 floors of the given puzzle. The function is also named “permute-bands”. There is a similar function “permute-towers” or “permute-stacks” for vertical permutations.

```
(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 the new 1 2 3 rows of this floor are the ?f1 ?f2 ?f3 rows of the given floor. The function is also named “permute-rows-in-band”. There is a similar function “permute-columns-in-towers” or “permute-columns-in-stacks” for vertical permutations.

```
(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. There is a similar function “vertical-random-shuffle” for vertical permutations.

```
(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.

They describe features that are currently available only for Sudoku but that could easily be extended to other CSPs.

****

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 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 below.

6.7.1 Init functions

Knowing the independent 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 examples.

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; sets of preferences

```
+-------+-------+-------+
! . . . ! . . . ! 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 (among other things) 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 plus 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 with the first release of CSP-Rules. 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
```

6.8.1 Automating the process by using sets of preferences

While the above "manual" approach remains the most versatile one, CSP-Rules now has a way of making easier the whole process of solving in stages, in the most likely to occur cases: function ***"solve-w-preferences"***, with syntax:

```
(solve-w-preferences <sudoku-string> P1 P2 ...)
```

As usual, the first argument is the sudoku puzzle in the standard string format. The next arguments make a (possibly empty) sequence of preferences, each selected among the following list of resolution theories (they are here listed on two lines, one line for the generic resolution theories and one for the Sudoku-specific ones):

- BRT, W1, BIVALUE-CHAINS, REVERSIBLE-CHAINS
- S2, S3, S4 (or S), S2Fin, S3Fin, S4Fin (or SFin), REVERSIBLE-PATTERNS, TRID, TRID-FW, TRID-ORk-FW, TRID-ORk-CW, TRID-ORk-W and TRID-ORk-CHAINS.

BRT, W_1, S_n and S_nFin have already been defined; BIVALUE-CHAINS means W_1 plus the bivalue-chains of maximum length as defined in the configuration file; REVERSIBLE-CHAINS means BIVALUE-CHAINS plus z-chains and oddagons with similar restriction on lengths. REVERSIBLE-PATTERNS means REVERSIBLE-CHAINS plus SFin. Typed-chains are included in the above resolution theories if and only if they were selected in the configuration file (they don't change their resolution power). For TRIDAGONS (TRID-xxx), see chapter 14.

What happens when function "solve-w-preferences" is called is: CSP-Rules first tries to solve the puzzle using only the rules in P1; then, if the puzzle is not solved and starting from the final resolution state thus obtained, it tries again, using only the rules in P2; and so on. At the end, if the puzzle is not yet solved, CSP-Rules tries with all the rules chosen in the configuration file (considered as an implicit preference MAIN at the end of the list). Several remarks are in order:

- before the function does anything, it checks all the names in the list of preferences; if any name is not strictly as in the above list, the function mentions it, recalls the allowed list and stops without doing anything (even if the rest correct);
- the list of preferences may be empty, in which case "solve-w-preferences" is equivalent to "solve";
- within each phase of resolution in one of the P_i's, the simplest-first strategy keeps driving the resolution process; everything happens as if only the rules in P_i were loaded;
- at the end of each phase, if the solution is not yet found and global variable ?*print-final-RS* is set to TRUE (the default value), the resolution state is printed;
- each preference virtually sets a new "process" running in parallel (conceptually) with all the other ones and the default one; as a result, some pattern-matching work may be duplicated and performance may be reduced in terms of resolution time and memory used; (technical note: all these "processes" use the same global set of facts, but each of them has its own Rete network for its own pattern-matching; pattern-matching structures and processes is what's duplicated);
- when a resolution theory is mentioned for the first time in a call to "solve-w-preferences" after loading time, the associated rules are loaded again; this is the new "process" setting up its independent pattern-matching network;
- "solve-w-preferences" does not impose any restriction on the order of preferences, or the possible repetition of some of them; but it is obvious that putting S4 after S4Fin will not be very useful; one could imagine a sequence like "S4Fin REVERSIBLE-CHAINS" being repeated, but it seems more natural to use "S4Fin REVERSIBLE-PATTERNS".

In the above example, function "solve-w-preferences" could be used as follows in order to obtain the second resolution path with a single command:

```
(solve-w-preferences
   "......1....12....3.4..5..2..5..6..4...73....6.....73..2..........87....1.6..4..5."
   S4Fin BIVALUE-CHAINS)
```

However, if you don't know in advance that this is enough to solve the puzzle, you may want to go further in restricting the increase in rules complexity before allowing all the loaded rules to apply, e.g.:

```
(solve-w-preferences
   "......1....12....3.4..5..2..5..6..4...73....6.....73..2..........87....1.6..4..5."
   S4Fin BIVALUE-CHAINS REVERSIBLE-CHAINS REVERSIBLE-PATTERNS)
```

This will not change anything in the present case, but in many harder puzzles, it will allow to try e.g. all the reversible patterns before any non-reversible ones.

If you prefer avoiding fins, you can also be more specific about it:

```
(solve-w-preferences
   "......1....12....3.4..5..2..5..6..4...73....6.....73..2.........87....1.6..4..5."
   S4 S4Fin BIVALUE-CHAINS REVERSIBLE-CHAINS REVERSIBLE-PATTERNS)
```

As should be expected from the previous sections, "solve-w-preferences" (further abbreviated as "***solve-w-prefs***") is an abbreviation for ***"solve-sudoku-string-w-preferences"*** and there are similar functions: ***"solve-sudoku-list-w-preferences"***, ***"solve-sukaku-list-w-preferences"***, ***"solve-sudoku-grid-w-preferences"***, ***"solve-sukaku-grid-w-preferences"***. Their syntax is the same as with the corresponding function without preferences, but with an additional sequence of preferences at the end. CSP-Rules does the job of deciding what pertains to the puzzle and what is a preference. Example for the same puzzle as above:

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

S4Fin BIVALUE-CHAINS REVERSIBLE-CHAINS REVERSIBLE-PATTERNS)
```

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 chapter 13) can be that a candidate is a T_0-anti-backdoor and eliminating it would lead to a single-step solution (considering rules from some elementary resolution theory 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: $n_i r_j c_k$. However, if any such rule is activated, it is mandatory to use the "nrc-to-label" function for each candidate. 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 non-exotic 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. Subsets, bivalue-chains, z-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 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.
- 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 Eleven's replacement technique

This section is about a very smart and very powerful "algebraic" technique first introduced by "eleven" in 2010 (http://forum.enjoysudoku.com/an-alternative-way-to-solve-the-hard-17-clue-t30022.html#p200126), with a recent revival.

6.10.1 Original definition of eleven's replacement technique

If, in any resolution state RS, there is a line L (i.e. a row, a column or a block) with three cells having exactly the same remaining three candidates, say 1, 2, 3, then:

– step 1: in RS, replace any instance of any of these numbers (including the givens and decided values) by the three of them: 123; (in this step, some information is obviously lost);

– step 2: in L, replace the content of the three cells by different variables, respectively x, y, z;

– step 3: solve as if x, y, z were digits;

– step 4: identify the values of x, y, z based on the givens.

In many cases, this will lead to a solution simpler than for the original puzzle. Notice that, even if (Naked, Hidden and Super-Hidden) Triplets rules are allowed, it might be interesting to extend the technique to triplets of cells in the rn, cn or bn spaces; but this is currently not considered in SudoRules.

However powerful it can be, this technique has a few drawbacks:

– solving with x, y, z instead of real digits may not be so easy for the human solver in all the cases;

– it is not doable by the current computer-based solvers;

– in general, there are many possibilities for the three cells, but no general rule for choosing which is best; in some cases, a choice will lead to an easy (or at least easier) solution while another choice will lead nowhere;

– as some information is lost in the transformation process, no solution of the modified puzzle can be guaranteed *a priori*, even in the best case of the original one having a single solution (though, in practice, this doesn't seem to be very restrictive);

– this technique cannot be defined as a resolution rule (it is fundamentally a form of educated T&E – see sub-section 6.10.4);

– the computational complexity cannot be defined (more on this in sub-section 6.10.4).

6.10.2 SudoRules variant of eleven's replacement technique

Here is my own variant of eleven's technique, as implemented in SudoRules. It allows to discard the first two of the above-mentioned drawbacks, while still capturing the essence of the technique:

– step 1: in RS, replace any instance of any of the three numbers (including the givens and decided values) by the three of them: 123; (here, some information is obviously lost);

– step 2: in line L, replace the content of the three cells by arbitrarily assigned different values from the set {1, 2, 3}; we may now be dealing with an isomorph of the original puzzle; (note that this assignation of values amounts to introducing two levels of T&E);

– step 3: solve the sukaku puzzle thus obtained;

– step 4: make the right permutation of digits 1, 2, 3 so that they coincide with the givens (which, in the T&E view, amounts to rejecting the wrong T&E assignations of pairs made at the second step).

6.10.3 Example of eleven's replacement technique

To make it completely clear, consider the following puzzle (from eleven).

```
+-------+-------+-------+
| . 4 . | . . 8 | 7 . . |
| 5 . 9 | . . . | . . . |
| . 7 . | 4 . . | 6 . 9 |
+-------+-------+-------+
| . . 5 | . . . | . . . |
| 7 . . | 8 6 . | . . . |
| 6 . . | . . . | 5 8 4 |
+-------+-------+-------+
| . 6 . | . . 9 | 8 . . |
| 9 . . | . 7 6 | . 1 2 |
| . . . | 5 . . | . 6 . |
+-------+-------+-------+
.4...87..5.9.......7.4..6.9..5......7..86....6.....584.6...98..9...76.12...5...6.
SER = 9.0
```

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

```
+-------------------+-------------------+-------------------+
! 123   4     6     ! 129   12359 8     ! 7     235   135   !
! 5     123   9     ! 6     123   7     ! 13    4     8     !
! 8     7     123   ! 4     1235  123   ! 6     235   9     !
+-------------------+-------------------+-------------------+
! 123   8     5     ! 129   12349 1234  ! 123   7     6     !
! 7     123   4     ! 8     6     5     ! 123   9     13    !
! 6     9     123   ! 7     123   123   ! 5     8     4     !
+-------------------+-------------------+-------------------+
! 1234  6     1237  ! 12    124   9     ! 8     35    357   !
! 9     5     8     ! 3     7     6     ! 4     1     2     !
! 1234  123   1237  ! 5     8     124   ! 9     6     37    !
+-------------------+-------------------+-------------------+
107 candidates.
```

This puzzle has a solution in W7 (see it in [CSP-RULES-EXAMPLES], file Sudoku/eleven-replacement/123-pattern.clp). But we can also apply eleven's technique: suppose that, in the above resolution state, we replace every occurrence of 1 2 3 (including the givens) by the 3 digits (step 1). We get the following sukaku grid:

```
+-------------------+-------------------+-------------------+
! 123   4     6     ! 1239  12359 8     ! 7     1235  1235  !
! 5     123   9     ! 6     123   7     ! 123   4     8     !
! 8     7     123   ! 4     1235  123   ! 6     1235  9     !
+-------------------+-------------------+-------------------+
```

```
! 123   8     5     ! 1239  12349 1234  ! 123   7     6     !
! 7     123   4     ! 8     6     5     ! 123   9     123   !
! 6     9     123   ! 7     123   123   ! 5     8     4     !
+-------------------+-------------------+-------------------+
! 1234  6     1237  ! 123   1234  9     ! 8     1235  12357 !
! 9     5     8     ! 123   7     6     ! 4     123   123   !
! 1234  123   1237  ! 5     8     1234  ! 9     6     1237  !
+-------------------+-------------------+-------------------+
```

We can now choose any arrangement of values among {1, 2, 3} for r4c1, r5c2, r6c3 (step 2). If we get a solution, it will be a solution of the original puzzle modulo some renaming of 1, 2, 3:

```
+-------------------+-------------------+-------------------+
! 123   4     6     ! 1239  12359 8     ! 7     1235  1235  !
! 5     123   9     ! 6     123   7     ! 123   4     8     !
! 8     7     123   ! 4     1235  123   ! 6     1235  9     !
+-------------------+-------------------+-------------------+
! 1     8     5     ! 1239  12349 1234  ! 123   7     6     !
! 7     2     4     ! 8     6     5     ! 123   9     123   !
! 6     9     3     ! 7     123   123   ! 5     8     4     !
+-------------------+-------------------+-------------------+
! 1234  6     1237  ! 123   1234  9     ! 8     1235  12357 !
! 9     5     8     ! 123   7     6     ! 4     123   123   !
! 1234  123   1237  ! 5     8     1234  ! 9     6     1237  !
+-------------------+-------------------+-------------------+
```

And we can apply the standard SudoRules function for solving this sukaku (step 3):

```
(solve-sukaku-grid
+-------------------+-------------------+-------------------+
! 123   4     6     ! 1239  12359 8     ! 7     1235  1235  !
! 5     123   9     ! 6     123   7     ! 123   4     8     !
! 8     7     123   ! 4     1235  123   ! 6     1235  9     !
+-------------------+-------------------+-------------------+
! 1     8     5     ! 1239  12349 1234  ! 123   7     6     !
! 7     2     4     ! 8     6     5     ! 123   9     123   !
! 6     9     3     ! 7     123   123   ! 5     8     4     !
+-------------------+-------------------+-------------------+
! 1234  6     1237  ! 123   1234  9     ! 8     1235  12357 !
! 9     5     8     ! 123   7     6     ! 4     123   123   !
! 1234  123   1237  ! 5     8     1234  ! 9     6     1237  !
+-------------------+-------------------+-------------------+
)
singles ==> r4c7=2, r2c5=2, r6c5=1, r6c6=2
Resolution state after Singles and whips[1]:

+-------------------+-------------------+-------------------+
! 23    4     6     ! 139   359   8     ! 7     1235  1235  !
! 5     13    9     ! 6     2     7     ! 13    4     8     !
! 8     7     12    ! 4     35    13    ! 6     1235  9     !
+-------------------+-------------------+-------------------+
```

```
! 1     8     5     ! 39    349   34    ! 2     7     6     !
! 7     2     4     ! 8     6     5     ! 13    9     13    !
! 6     9     3     ! 7     1     2     ! 5     8     4     !
+-------------------+-------------------+-------------------+
! 234   6     127   ! 123   34    9     ! 8     1235  12357 !
! 9     5     8     ! 123   7     6     ! 4     123   123   !
! 234   13    127   ! 5     8     134   ! 9     6     1237  !
+-------------------+-------------------+-------------------+
85 candidates.
```

```
finned-x-wing-in-columns: n1{c6 c2}{r9 r3} ==> r3c3≠1
singles ==> r3c3=2, r1c1=3, r2c2=1, r2c7=3, r5c7=1, r5c9=3, r9c2=3
biv-chain[3]: r3c6{n3 n1} - r9c6{n1 n4} - r7c5{n4 n3} ==> r3c5≠3
stte
```

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

By observing cells r8c8 and r8c9, one can see that this is not a solution to the original puzzle; but (step 4) replace 3 by 1 and 1 by 2 (and therefore 2 by 3) and you'll get the solution.

Notice that the sukaku puzzle obtained after eleven's replacement also has a 1-step solution, but it is in Z5 (still better than W7):

```
z-chain[5]: b1n1{r3c3 r2c2} - r9c2{n1 n3} - c6n3{r9 r4} - r4c4{n3 n9} - r1c4{n9 .} ==>
r3c6≠1
stte
```

The point here is, ***a puzzle that was in W7 is now solvable in the much simpler BC3***. Which immediately raises a question: ***how is it possible that, after losing information (in the first replacement step), we get a simpler puzzle?***

6.10.4 The complexity of eleven's replacement technique

As noticed at the end of the example in the previous sub-section, eleven's technique has allowed an apparent reduction of complexity from W7 to BC3, even though it had lost some information in its first step. We shall see in chapter 14 that the technique can even drastically downgrade a puzzle from T&E(3) to Z5 (or less), a relatively low part of T&E(1). This obviously requires some explanation.

Notice that one could even raise the restriction that each of the three cells has the three candidates (it is enough that each cell has two candidates and that each candidate appears at least twice in the three cells – the conditions of a cyclic Naked-Triplet. And, in, the replacement, one can even choose for a cell one of the three candidates that do not appear in it – which will of course require some isomorphism at the end.)

Subsets provide a very illuminating example. Suppose there is e.g. some Naked Triplet in a row and, instead of applying it, you apply eleven's replacement to its three digits and cells. The result is, you can now make by Singles all the eliminations that would have been made by the Triplet. In the present case, the complexity of the method appears to be the same as that of the Triplet: finding the three digits and cells that allow them to apply. Notice that eleven's technique could be applied to Pairs or Quadruplets of digits and cells in a line (though it seems to be less useful in such cases) and this conclusion could also be extended to Pairs and Quads. In such cases, the method has not simplified the puzzle.

Consider now the general case. What is there underneath the technique? It is easy to see that step 2 amounts to trying to plug two of the selected candidates in two of the selected cells (which decides the third); then checking if, using the selected resolution theory T, this implies a solution or not. Which is a form of T&E(T, 2). Obviously, the technique is much smarter than general T&E, let's say it's educated T&E, but complexity-wise it is T&E(T, 2), which provides the explanation required at the start.

6.10.5 The SudoRules functions for dealing with eleven's replacement technique

In any resolution state, there are two complementary parts in eleven's technique: choosing the relevant cells and applying the replacements. As of now, SudoRules provides only the functions for doing the second, boring mechanical part, leaving the choice of the three cells to the user. (See however chapter 14 for a case of automatic choice.)

SudoRules has two main functions for applying eleven's technique: ***solve-sukaku-list-by-eleven-replacement*** and ***solve-sukaku-grid-by-eleven-replacement***, with respective syntax:

```
(solve-sukaku-list-by-eleven-replacement
   ?nb1 ?nb2 ?nb3 ?row1 ?col1 ?row2 ?col2 ?row3 ?col3 $?sukaku-list)

(solve-sukaku-grid-by-eleven-replacement
   ?nb1 ?nb2 ?nb3 ?row1 ?col1 ?row2 ?col2 ?row3 ?col3 $?sukaku-grid)
```

As expected, ?nb1, ?nb2 and ?nb3 are for the three digits; (?rowi, ?coli) are the rc-coordinates of the ith cell and $?sukaku-list or $?sukaku-grid are the usual SudoRules list/grid representations of the resolution state RS where the transformation must be applied. What these functions do is very simple: they apply eleven's replacement, they plug each ?nbi into the corresponding (?rowi, ?coli), and then they re-start a resolution path from the resolution state thus obtained. They also print the resolution state before and after the replacement.

Beware that these functions do not check that the conditions for the replacement are satisfied. In the current implementation, they also leave step 4 to the user: check the final solution (if any) and make the proper digit isomorphism of ?nb1, ?nb2 and ?nb3 if needed, based on their occurrences in the initial givens.

Those are very basic functions, letting the user totally free (but unaided) in his choice of the three digits and cells. It doesn't seem desirable to have such a replacement procedure automatically started based only on the overly general condition of three digits in three cells in a line. But chapter 14 will show how, in the very special context of Tridagon related rules, it can be started automatically.

If you're only interested in doing eleven's replacements in some resolution state, without anything else, there are two utilitarian functions: ***replace-digits-in-sukaku-list*** and ***fix-digits-in-3-rc-cells-of-sukaku-list***, doing respectively the first and second step of the technique, with syntax:

```
(replace-digits-in-sukaku-list ?nb1 ?nb2 ?nb3 $?sukaku-list)
```

```
(fix-digits-in-3-rc-cells-of-sukaku-list ?nb1 ?nb2 ?nb3 ?row1 ?col1
?row2 ?col2 ?row3 ?col3 $?sukaku-list)
```

plus two similar functions: ***replace-digits-in-sukaku-grid*** and ***fix-digits-in-3-rc-cells-of-sukaku-grid***, allowing to take into account the sukaku grid format.

Notice that eleven's method could be extended to cells in rn, cn and bn spaces and similar functions could easily be written, but this is not currently done in SudoRules.

For completeness, SudoRules also has two functions for applying eleven's technique to 4 digits: ***solve-sukaku-list-by-eleven-replacement4*** and ***solve-sukaku-grid-by-eleven-replacement4***, with respective syntax:

```
(solve-sukaku-list-by-eleven-replacement4
   ?nb1 ?nb2 ?nb3 ?row1 ?col1 ?row2 ?col2 ?row3 ?col3 ?row4 ?col4
$?sukaku-list)
```

```
(solve-sukaku-grid-by-eleven-replacement4
   ?nb1 ?nb2 ?nb3 ?row1 ?col1 ?row2 ?col2 ?row3 ?col3 ?row4 ?col4
$?sukaku-grid)
```

But I have no example where this could be useful in practice.

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 adding a variant to 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 g-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 remaining 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 LatinSquares/Pandiagonals folder in [CSP-RULES-EXAMPLES] 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 LatinSquares folder in
[CSP-RULES-EXAMPLES]
```

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 [CSP-RULES-EXAMPLES].

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):

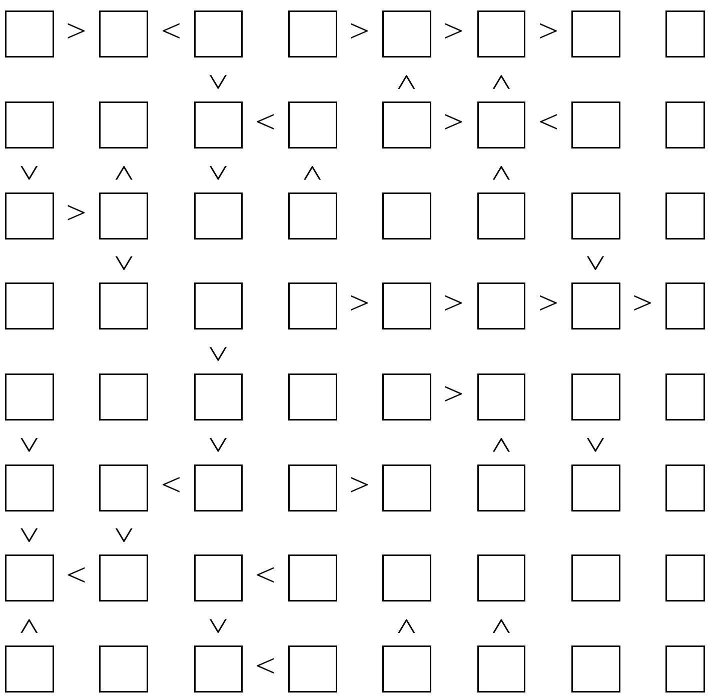


Figure 8 .1. Futoshiki puzzle H9305 from ATK

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). 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.

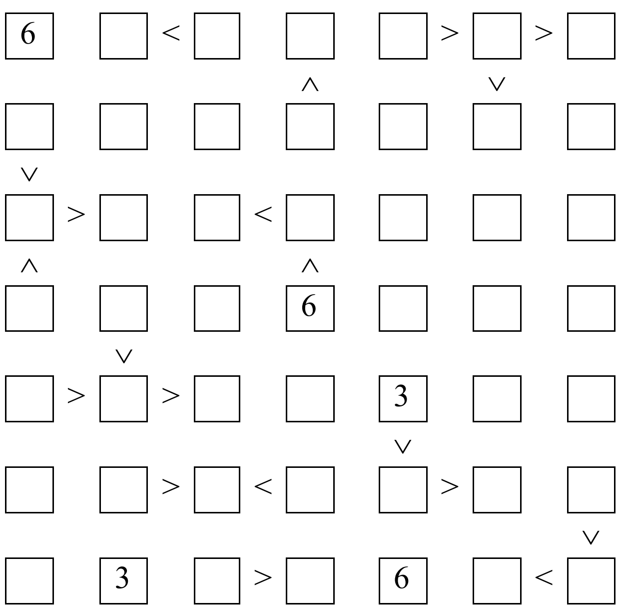


Figure 8.2. "Tricky"Futoshiki puzzle from Tatham

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

Basically, KakuRules 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.

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

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). To make it easier to try it, 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 . . //
)
```

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

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)
- $r_ic_{k'}$ for any white rc-cell $r_ic_{k'}$ in the sector
- $r_in_{k'}$ for any rn-cell $r_in_{k'}$ 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).

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 . . //
)
```

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*

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-r823
***  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 Slitherlink folder in [CSP-RULES-EXAMPLES] 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:

```
.   .———.   .   .———.———.———.———.———.———.
    | 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 |
.................———.———.———.   .———.   .
```

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 "Slitherlink/Mebane" folder in [CSP-RULES-EXAMPLES] 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).

```
.   .___.   .   .___.___.___.___.___.___.
    | 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 |
.___.___.___.___.___.___.___.   .___.   .
```

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

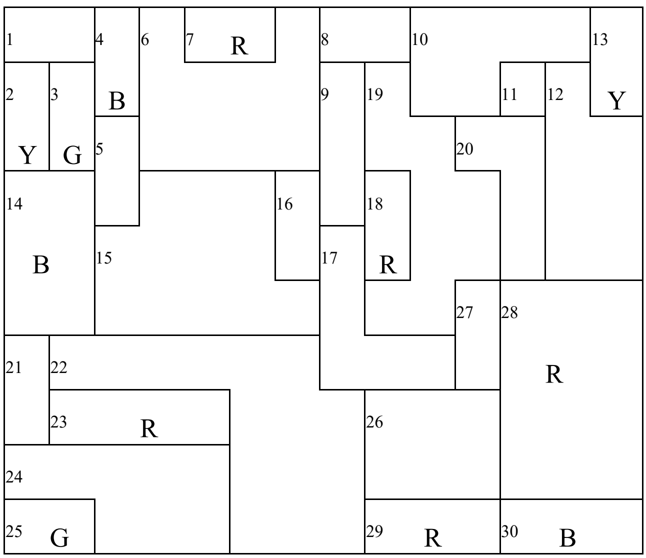
MapRules has a single user function: ***"solve"*** with syntax:

```
(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 as in [PBCS]), an "unreasonable" one from the tatham website: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/map.html, 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.

Part Three

ADVANCED TOPICS in SUDORULES

13. Requirements on the number of steps

The default resolution strategy in CSP-Rules is the simplest-first strategy introduced in my first works on the topic of pattern-based CSP solving (see [HLS], [CRT] or [PBCS]). However, I've always insisted on it: the resolution rules themselves are totally independent from this strategy.

This chapter is mainly about alternative ways of using the rules, in particular about dealing with additional requirements on the number of steps in the resolution paths. One thing it will illustrate is the strong independence between the resolution rules and the simplest-first strategy.

As a general background, it should be recalled that the number of steps of a proof in a logic theory T is irrelevant to what can be proven in T and there is thus no pure logic way of defining a rating based on the number of steps, let alone a rating of a full resolution path (combining the number of steps with the complexity of each step). And, as of now, no consistent proposal has ever been made for defining any other basis for such a rating. However, this doesn't exclude any possibility of dealing with some requirements on the number of steps.

Many of the advanced functions described in this chapter are written in (almost) generic form, but they are currently tested and available only for Sudoku.

It will be useful to keep in mind the remarks made in section 6.8 about focused eliminations. These remarks will not be repeated later, but they will fully apply to the ways we shall look for 1-step, 2-step or fewer-steps solutions – which are fundamentally based on focused eliminations:

- all the generic non-exotic 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. Subsets, bivalue-chains, z-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 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.

Finally: don't try to use the material in this chapter before you properly master the workings of the configuration file and you can confidently use the elementary functions.

13.1 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 (such as Singles or Whips[1]) are trivial and should not be counted as real steps.

13.1.1 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 (and it should be specified in the configuration file); T_0 is the set of rules we want to 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 not depend on the order of their application (and will therefore be a unique and well-defined resolution state). Typical examples for T_0 are:

- BRT (Basic Resolution Theory) = Singles + ECP (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. But you can experiment with the other allowed choices.

13.1.2 Example

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; notice that t-whips are not activated):

```
(defglobal ?*all-chains-max-length* = xx)
(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 of solving a puzzle.) It is true 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 this rating +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. In practice, these remarks mean that, in every case this framework is applied to a puzzle, one should first compute its T-rating.

13.2 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 much 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 13.1.1.

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 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 3e) of the SudoRules configuration file (don't forget to de-activate everything 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.

13.3 Looking for one-step solutions

The puzzle at the start of section 13.2:
`..9......8....97...7..6..5..5..4..6.9....38....2.......6..5.17.2....84......1....`
has a simplest-first 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 13.1).

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 13.2 (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.

13.3.1 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-grid
    +-------------------------+-------------------------+-------------------------+
    ! 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 for the only purpose of illustrating 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.

13.3.2 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 13.5 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.

13.3.3 The fully-automated approach

The semi-automated approach still 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 13.1. I personally consider that only BRT and W1 really make sense: considering a Subset, even a Pair, as a no-step is nonsense because it generally produces many eliminations.

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).

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"*** (further abbreviated as ***"find-1-steppers"***), ***"find-sukaku-1-steppers-wrt-BRT"*** and ***"find-sukaku-1-steppers-wrt-W1"***, with the same syntax except the first ($?T_0$) input variable.

13.4 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 be 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 a way that can prevent them from being enabled, even if loaded. Until recently, there had been no opportunity to effectively use this feature. Since the 8th update of CSP-Rules-V2.1, a more dynamical way of disabling and re-enabling rules is possible. 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 13.3 for function find-sudoku-1-steppers-wrt-resolution-theory.

13.4.1 Functions for disabling and re-enabling rules

Functions allowing to 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 ?RT0)
```

where ?cont is an integer for a context (0 for the starting context) and $?RT_0$ is a resolution theory as in section 13.1. 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 $?RT_0$ will be disabled in context ?cont. In this function, $?RT_0$ can only be BRT, W1, S2, S3, S4 or S, as in section 13.1. What's new since the 8th update is, resolution rules can be disabled even if they have already been automatically enabled.

Function ***"re-enable-rules-not-in-RT0***" with syntax:

```
(re-enable-rules-not-in-RT0 ?cont ?RT0)
```

(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;

– for any family of rules (e.g. z-chains or whips or g-whips or …), 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.

13.4.2 Functions for reconstructing a resolution path

When some rules are considered as no-step, as in section 13.1, 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 – see section 13.5.2).

```
(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)
```

13.5 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, 2-step or fewer-step solutions.

13.5.1 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 into 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, for technical reasons, 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*)
```

13.5.2 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 13.2 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
))
```

13.6 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 13.2.)

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.

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

Consider the following (relatively hard) puzzle (proposed by Urhegyi) and let's start solving it from its resolution state (call it again ?*RS*) after Singles and whips[1]:

```
+-------+-------+-------+
! 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

(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 13.2, 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
OR:
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
OR:
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], as above), 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 MacBookPro 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.

13.8 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$ ...as before...))
(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 (6m 33s instead of 24m 11s).

```
(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$ ... ))
(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 manually 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.

13.9 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 13.8, with only the relevant rules loaded at each step), we shall restrict the lengths of all the chains to 5 in this example.

◊ **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 function ***"find-sudoku-2-steppers-wrt-resolution-theory"***, or its abbreviation when T_0 = W1 ***"find-2-steppers"***, 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"
)
```

or:

```
(find-2-steppers
"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 (depending on your brand of tea), 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 including 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 $?RT_0$ 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"*** (further abbreviated as ***"find-2-steppers"***). 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.

13.10 Reducing the number of steps

This section is about a method 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 that the goal is not to "find the smallest number of steps" in T; this would be meaningless without stating in which resolution theory and unachievable in most cases even with stating it. As previously, rules in some chosen T_0 are considered as no-step.

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 with respect 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; but 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.*

Remark: one general problem about trying to "optimise" 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 intrinsic 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 in this section almost totally avoids such problems. By first carefully selecting the allowed maximum length L of all the chains (based on the simplest-first solution and the rating of the puzzle thus obtained), only the number of steps of length ≤ L has to be taken into account.

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

Notice that my current implementation in CSP-Rules is experimental and very slow. Speed-wise, there remains many open possibilities for improving it.

13.9.1 The heuristics used for evaluating the candidates

The strategy used by SudoRules to reduce the number of steps[1] is of the ***steepest-descent*** kind and 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);
3) define variable step = 0;
4) bind step = step + 1;
5) find all the candidates in RS that could be deleted by the application of a single rule in T (this does not change RS);
6) evaluate (according to the method defined below) each of these "erasable candidates" (this does not change RS);
7) randomly choose one of these erasable candidates among those with the best score, say C (this does not change RS);
8) apply the simplest rule in T that allows to eliminate C (as appears here, the simplest-first strategy is not completely discarded); this changes RS;
9) apply all the rules in T_0 until none can be applied; this changes RS (unless the C score was only 1);
10) if the puzzle is not solved, then GOTO 4, else report it as solved, report the value of variable "step" and stop.

The above procedure 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, in the hope that some

[1] As he pointed it out, Defise's strategy (although of the same steppest-descent kind) is slightly different: instead of evaluating candidates, he evaluates what, in the CLIPS parlance would be called rule activations; this implies that the same candidate can have several scores depending on how it is eliminated. In SudoRules, only the simplest rule activation is considered.

of the combined random choices will lead to a path with still fewer steps. As in any steepest-decent method, no guarantee can be given that the smallest number of steps will ever be found, even if many tries are made.

There remains to define the *evaluation function* for an erasable candidate C in a resolution state RS during step 6. The most natural way of doing it is very simple; the score of C is defined as the total number of deletions that result from applying in RS the simplest rule that allows to delete C and then applying all the rules in T_0.

For ease of understanding, one can imagine this evaluation process is done for each candidate C in a copy RS' of the current resolution state RS and it starts by applying in RS' the simplest rule that deletes C (together with possibly other targets of the same rule instantiation, when the same rule can eliminate more than one candidate, i.e. when the "blocked" version of resolution rules is selected) and then applying the rules in T_0, resulting in a final state RS''. The score is the difference in the numbers of candidates between RS and the final RS''. 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 this new decided value implies by direct contradictions are also counted; as a result, there is no reason to make any special proviso for values that are decided in the process.

The obvious heuristic justification for defining the score this way is, if many candidates are eliminated by applying one step that has to be counted, plus all the 0-step rules, there will remain fewer candidates to eliminate, hopefully requiring fewer additional steps. Unfortunately, this hope is not always verified and this is the main limitation of the approach, *a limitation typical of any steepest descent algorithm*.

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 bivalue cells is rarely useful and I can see no reason why generalizing this to possibly more companion candidates would be very useful.

13.9.2 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 hard puzzle below, from my own collection, Pisces-9.0 (SER = 9.0, W = 7). (See the "pisces2#523-9.0-W7.clp" file in [CSP-RULES-EXAMPLES].)

```
+-------+-------+-------+
! 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
```

13.9.3 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:

```
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . 1 ! 2 . . ! . . 3 !
! . 4 . ! . 5 . ! . 6 . !
+-------+-------+-------+
! . 5 . ! . 6 . ! . 7 . !
! . . 3 ! 8 . . ! . . 2 !
! . . . ! . . 7 ! . . . !
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . 2 ! 3 . . ! . 9 8 !
! . 6 . ! . 7 . ! . 5 . !
+-------+-------+-------+
..........12....3.4..5..6..5..6..7...38....2.....7.............23...98.6..7..5.
SER = 4.0
```

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
```

13.9.4 Syntax

The most general function for launching the above calculations is:

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

It has two specialized versions for the most natural resolution theories ?RT$_0$:

```
(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 ?RT0 $?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 slightly 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 the first found among those with the fewest number of steps that also have the smallest effective max-chain-length.

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)
```

13.10 Forcing T&E

Forcing-T&E and Forcing{3}-T&E were defined in section 3.9. Here is an extreme example of application. The puzzle ("Gata de Mar") is a relatively hard one (with SER = 8.3), proposed by eleven:

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

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

```
+---------------------+---------------------+---------------------+
! 1      8      4     ! 239    29     6     ! 5      2379   237   !
! 7      6      5     ! 4      1289   123   ! 12389  1239   1238  !
! 2      9      3     ! 18     7      5     ! 18     6      4     !
+---------------------+---------------------+---------------------+
! 5      7      8     ! 6      123    4     ! 123    123    9     !
! 9      4      126   ! 7      123    8     ! 1236   5      1236  !
! 36     123    126   ! 12     5      9     ! 12468  1247   12678 !
+---------------------+---------------------+---------------------+
! 36     5      7     ! 1239   12469  123   ! 123469 8      1236  !
! 8      123    126   ! 5      12469  123   ! 7      12349  1236  !
! 4      123    9     ! 1238   1268   7     ! 1236   123    5     !
+---------------------+---------------------+---------------------+
148 candidates.
```

The puzzle has a simplest-first solution in SFin+Z5 (in 30 non-W1 steps):

```
hidden-pairs-in-a-row: r8{n4 n9}{c5 c8} ==> r8c8≠3, r8c8≠2, r8c8≠1, r8c5≠6, r8c5≠2,
r8c5≠1
hidden-pairs-in-a-block: b9{n4 n9}{r7c7 r8c8} ==> r7c7≠6, r7c7≠3, r7c7≠2, r7c7≠1
finned-x-wing-in-rows: n9{r8 r1}{c8 c5} ==> r2c5≠9
whip[1]: r2n9{c8 .} ==> r1c8≠9
finned-x-wing-in-rows: n6{r8 r5}{c3 c9} ==> r6c9≠6
hidden-triplets-in-a-row: r6{n4 n7 n8}{c7 c8 c9} ==> r6c9≠2, r6c9≠1, r6c8≠2, r6c8≠1,
r6c7≠6, r6c7≠2, r6c7≠1
whip[1]: r6n6{c3 .} ==> r5c3≠6
biv-chain[3]: c7n9{r2 r7} - c4n9{r7 r1} - b2n3{r1c4 r2c6} ==> r2c7≠3
z-chain[3]: c4n9{r7 r1} - b2n3{r1c4 r2c6} - c6n1{r2 .} ==> r7c4≠1
z-chain[3]: c4n9{r7 r1} - b2n3{r1c4 r2c6} - c6n2{r2 .} ==> r7c4≠2
z-chain[3]: c5n8{r9 r2} - r3c4{n8 n1} - b5n1{r6c4 .} ==> r9c5≠1
biv-chain[4]: r1c5{n2 n9} - r8c5{n9 n4} - c8n4{r8 r6} - c8n7{r6 r1} ==> r1c8≠2
biv-chain[3]: r1c8{n3 n7} - c9n7{r1 r6} - c9n8{r6 r2} ==> r2c9≠3
z-chain[3]: c6n2{r8 r2} - r1n2{c4 c9} - r7n2{c9 .} ==> r9c5≠2
z-chain[3]: c6n2{r8 r2} - r1n2{c5 c9} - r7n2{c9 .} ==> r9c4≠2
biv-chain[4]: r3c7{n1 n8} - b2n8{r3c4 r2c5} - r9c5{n8 n6} - c7n6{r9 r5} ==> r5c7≠1
biv-chain[4]: c5n8{r2 r9} - b8n6{r9c5 r7c5} - r7n4{c5 c7} - c7n9{r7 r2} ==> r2c7≠8
biv-chain[4]: c8n9{r2 r8} - b9n4{r8c8 r7c7} - r6c7{n4 n8} - r3c7{n8 n1} ==> r2c8≠1
z-chain[4]: b5n1{r5c5 r6c4} - c4n2{r6 r1} - b2n3{r1c4 r2c6} - c6n1{r2 .} ==> r7c5≠1
biv-chain[5]: c5n6{r7 r9} - c5n8{r9 r2} - b3n8{r2c9 r3c7} - r6c7{n8 n4} - r7c7{n4 n9}
==> r7c5≠9
biv-chain[5]: c5n6{r7 r9} - c5n8{r9 r2} - b3n8{r2c9 r3c7} - r6c7{n8 n4} - r7n4{c7 c5}
==> r7c5≠2
whip[1]: b8n2{r8c6 .} ==> r2c6≠2
hidden-pairs-in-a-row: r7{n1 n2}{c6 c9} ==> r7c9≠6, r7c9≠3, r7c6≠3
z-chain[3]: b9n3{r9c8 r8c9} - r8n6{c9 c3} - r7c1{n6 .} ==> r9c2≠3
biv-chain[4]: c2n3{r6 r8} - r7n3{c1 c4} - c4n9{r7 r1} - c4n2{r1 r6} ==> r6c2≠2
whip[1]: c2n2{r9 .} ==> r8c3≠2
biv-chain-cn[3]: c3n2{r5 r6} - c3n6{r6 r8} - c9n6{r8 r5} ==> r5c9≠2
biv-chain[3]: r9c2{n2 n1} - r8c3{n1 n6} - b9n6{r8c9 r9c7} ==> r9c7≠2
z-chain[3]: b7n1{r8c3 r9c2} - r9n2{c2 c8} - r7c9{n2 .} ==> r8c9≠1
z-chain[4]: r9c2{n1 n2} - r9c8{n2 n3} - r2n3{c8 c6} - c6n1{r2 .} ==> r9c4≠1
whip[1]: b8n1{r8c6 .} ==> r2c6≠1
naked-single ==> r2c6=3
```

```
naked-pairs-in-a-block: b2{r1c4 r1c5}{n2 n9} ==> r2c5≠2
whip[1]: r2n2{c9 .} ==> r1c9≠2
biv-chain[3]: r8n3{c9 c2} - r7c1{n3 n6} - r8n6{c3 c9} ==> r8c9≠2
z-chain[4]: r7c9{n1 n2} - r2c9{n2 n8} - r2c5{n8 n1} - r4n1{c5 .} ==> r5c9≠1
whip[1]: b6n1{r4c8 .} ==> r4c5≠1
naked-pairs-in-a-column: c9{r5 r8}{n3 n6} ==> r1c9≠3
stte
```

This puzzle has no 1-step or 2-step solution based on whips of length no more than 8 – which would already be very complicated patterns for a puzzle in SFin+Z5. Using the method of section 13.9, I found a resolution path with 9 steps in S+W8, after 3 tries; based on a path obtained with T_0 = S, Defise found one with only 6 steps in S+W8, after 4 tries.

From all this, it appears that this puzzle has no easy solution with few steps of restricted length. However, it has a "1-step" solution using Forcing{3}-T&E, obtained with function ***"apply-F3TE"*** (to be called with no argument).

```
FORCING[3]-T&E(W1) applied to trivalue candidates n2r1c9, n3r1c9 and n7r1c9 :
===> 3 values decided in the three cases: n6r7c1 n3r6c1 n6r6c3
===> 63 candidates eliminated in the three cases: n2r1c4 n2r1c8 n9r1c8 n2r2c5 n9r2c5
n2r2c6 n1r2c7 n3r2c7 n8r2c7 n1r2c8 n2r2c9 n3r2c9 n2r4c5 n2r4c7 n3r4c8 n6r5c3 n2r5c5
n1r5c7 n3r5c7 n1r5c9 n2r5c9 n6r6c1 n3r6c2 n1r6c3 n2r6c3 n1r6c7 n2r6c7 n6r6c7 n1r6c8
n2r6c8 n1r6c9 n2r6c9 n6r6c9 n3r7c1 n1r7c4 n2r7c4 n1r7c5 n6r7c5 n2r7c6 n3r7c6 n1r7c7
n2r7c7 n3r7c7 n6r7c7 n1r7c9 n6r7c9 n1r8c2 n2r8c2 n6r8c3 n1r8c5 n2r8c5 n4r8c5 n1r8c6
n1r8c8 n2r8c8 n3r8c8 n2r8c9 n3r8c9 n3r9c4 n1r9c5 n2r9c5 n1r9c7 n2r9c7
stte
```

Such a carnage among the candidates is why I call Forcing-T&E and Forcing{3}-T&E the nukes of solving. Remember however that it doesn't always work so smoothly and that it is among the most inelegant solving techniques as it cumulates T&E with reasoning by cases (i.e. using several streams of reasoning in parallel).

The corresponding function for Forcing-T&E is ***"apply-F2TE"***.

14. Puzzles in T&E(3) and the Tridagon family of rules

This chapter is all about the recently found "trivalue oddagon" impossible pattern in 9×9 Sudoku, the first known pattern that requires T&E(3) to be proven contradictory. A consistent puzzle that "almost" have this pattern must indeed have some "guardians" preventing it to fall into inconsistency and this translates into the existence of an OR_k relation between them and into the possibility of using generic OR_k chains based on it.

This new pattern was indeed the original reason why I added to CSP-Rules new generic resolution rules based on OR_k relations (see section 3.5 for definitions).

This anti-tridagon pattern can be considered as "exotic" because it is very rare (almost impossible to be found by the random generation of puzzles); in my view, this allows to grant its detection rule some special "rights" to break the natural classification based on the number of CSP-Variables it involves.

In addition to the results themselves, this chapter is a detailed illustration of how CSP-Rules can be used to participate in research on the most recent topics.

14.1 The trivalue oddagon and the tridagon patterns

The pattens and resolution rules introduced in the first sections of this chapter are related to a recently discovered Sudoku-specific pattern that is impossible in any consistent puzzle, known as a "***trivalue oddagon***" (or "Thor's Hammer" in some circles, due to the shape formed by the "+" signs in the following picture). It is based on four blocks forming a rectangle, each with three cells disposed as shown by the "+" signs (modulo Sudoku isomorphisms), with the twelve cells having the same three candidates (and only them) – and no conditions at all on any other cell.

```
+-------+-------+
! + . . ! + . . !
! . + . ! . + . !
! . . + ! . . + !
+-------+-------+
! + . . ! . . + !
! . + . ! . + . !
! . . + ! + . . !
+-------+-------+
```

This beautifully simple pattern has been made famous in the Sudoku community via a puzzle ("***Loki***") created by mith, which also happened to be the 10th known one with the highest known SER: 11.9. Of course, Loki doesn't have the trivalue-oddagon pattern itself: it has one more candidate in one of the twelve cells – allowing the application of an obvious Tridagon elimination rule.

Although this is not strictly the topic of this chapter, it is worth noting the ***role played by Loki in totally re-orienting the decades-old search for the hardest Sudoku puzzles***. Ever since I defined my BpB sub-classification of T&E(2) in [CRT, 2011] and its application to the hardest known puzzles in [PBCS1, 2012], at a time when only three known puzzles were in B7B, I had kept computing the BpB classification of all the puzzles with high SER as soon as they were published. During those eleven years, in spite of many conjugated efforts by many experts for finding new puzzles qualifying as hardest, including many recent SER 11.8 ones created by mith, no new one was found to be in B7B or above.

Imagine my surprise when I found that Loki was not in B7B and my still bigger surprise when I also found that ***Loki was the first ever known puzzle not in T&E(2)*** (a result I first reported here:
http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-1048.html).
Two of my old conjectures (T&E(2) and the stronger B7B) that had resisted eleven years of intensive research were suddenly disproved by a single counter-example.

The positive counterpart was, now that puzzles not in T&E(2) were known to exist, the T&E-depth of a puzzle (much faster to compute than its SER) could become the criterion for looking for the hardest puzzles by the usual vicinity search methods. This suggestion quickly led to a sudden surge in the number of hard puzzles (now produced in millions by mith, and a few more by Hendrik Monard).

14.1.1 Historical note

Contrary to many familiar resolution rules, it is relatively easy to track the origins of the those defined in this section:

– first mention of the name "Thor's Hammer" on sudoku forums:
http://forum.enjoysudoku.com/the-hardest-sudokus-t4212-407.html

– first mention of the name "trivalue oddagon" on sudoku forums:
http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-811.html

– eleven's first explanation of the pattern:
http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-816.html

– eleven's proof that the pattern is contradictory:
http://forum.enjoysudoku.com/post318379.html#p318379

– coloin's identification of 6 cases:
http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-817.html

– mith's first 11.9 "Loki", the 10th known puzzle with SER = 11.9:
http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-1030.html

– my report that Loki is the first ever puzzle not in T&E(2) and my suggestion for a new way of searching for the hardest puzzles:

http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-1048.html

– see also all the posts following the previous two and the discussion about “trivalue oddagons”;

– various resolution paths of Loki:

http://forum.enjoysudoku.com/loki-ser-11-9-t39840.html

– a YouTube presentation of the impossible pattern (unfortunately on a trivial puzzle) by rangsk: https://youtu.be/V7RC1hJ8vZ8

– my view of how to use the pattern in a resolution rule (see the whole thread):

http://forum.enjoysudoku.com/the-tridagon-rule-t39859.html

Notice that, in and of itself, an impossible pattern doesn’t define any resolution rules. In order to avoid any ambiguities, I’ll continue to use the more or less standard name “trivalue oddagon” for the above-mentioned impossible pattern. The next sections will consist of precisely defining two types of resolution rules based on it: the Tridagon elimination rule and the family of Tridagon-forcing-whips[n]. See section 14.9 to 14.11 for three more general families: Trid-OR_k-forcing-whips[n], Trid-OR_k-contrad-whips[n] and Trid-OR_k-whips[n], respectively based on the generic OR_k-forcing-whips[n], OR_k-contrad-whips[n] and OR_k-whips[n].

Notice also that the patterns defined in this chapter can have drastic effects on the difficulty of a puzzle, but puzzles with this pattern, in spite of existing by millions, are extremely rare (no chance of finding one by random generation).

14.2 The Tridagon elimination rule

The first versions of the Tridagon rule started by supposing that there is a trivalue oddagon plus a unique additional candidate in one of its cells and they concluded that the additional candidate could be asserted as True. But this rule should better be seen as an elimination of the three candidates (the further assertion of the additional one being left to the Naked-Singles rule). It then becomes obvious that the rule will work the same way (with possibly no Naked Single after it) with any number of additional candidates in a single of the pattern cells.

14.2.1 The Tridagon elimination rule

Our starting point will therefore be the following set of conditions. Suppose there are four blocks forming a rectangle in two bands (floors) and two stacks (towers):

b11 b12
b21 b22

and that there are three digits (the target digits), say 1 2 3, such that:

– in each of the four blocks, there are three cells in different rows and different columns such that:
--- each of these 4×3 cells contains the three digits as candidates,
--- eleven of these cells (the 123-cells) do not contain any other candidate,
--- the twelfth cell (the target cell) contains at least one more candidate,

– some additional conditions to be found below are satisfied.

Tridagon elimination rule: In the above conditions, the three target digits can be eliminated from the target cell.

My purpose here is to find the necessary and sufficient "additional conditions" for the rule to be valid. I'll propose a T&E-ish direct proof (involving four levels of T&E, first published here: http://forum.enjoysudoku.com/the-tridagon-rule-t39859.html). It will be very inelegant but very elementary, and it will not use any uniqueness argument.

First, notice that in the above conditions, it is easy to:
- make permutations of stacks and bands such that the target cell is in block b11;
- make permutations of rows in the first band such that the target cell is in row r1;
- make permutations of the columns in the first stack such that the target cell is in column r1;
- make further permutations of rows and columns within each band or stack such that we have the following pattern, with the target cell in r1c1 and with the 123-cells of the first three blocks in their anti-diagonal.

```
+-------------------------------+-------------------------------+
! 123456789 .         .         ! 123       .         .         !
! .         123       .         ! .         123       .         !
! .         .         123       ! .         .         123       !
+-------------------------------+-------------------------------+
! 123       .         .         ! *         *         *         !
! .         123       .         ! *         *         *         !
! .         .         123       ! *         *         *         !
+-------------------------------+-------------------------------+
```

- blocks not concerned by the pattern are not represented;
- "456789" means that each of these candidates may be in the cell and at least one of them must be in it;
- "." means a cell that does not participate in the pattern (or in the proof that it is contradictory) and that it can contain anything;
- "*" means a cell for which we are trying to decide whether it is part of the pattern.

At this point, there is nothing we can say about where the three cells of interest in b22 are. We can easily show that, given the conditions at the start, they can form only six non-isomorphic patterns of cells in b22.

Let us now try to prove the elimination rule. Suppose one of the target digits is true in the target cell. By symmetry of the conditions with respect to the three target digits, we can always suppose it is number 1. Again, by symmetry of the resulting situation with respect to the remaining two target digits, we can always suppose that r2c2 = 2, leading to the following situation in the four blocks:

```
+-------------------------------+-------------------------------+
! 1        .        .        ! 23       .        .        !
! .        2        .        ! .        13       .        !
! .        .        3        ! .        .        12       !
+-------------------------------+-------------------------------+
! 23       .        .        ! *        *        *        !
! .        13       .        ! *        *        *        !
! .        .        12       ! *        *        *        !
+-------------------------------+-------------------------------+
```

At this point, we can have either r4c1 = 2 or r4c1 = 3.

If r4c1 = 2, we can have either r1c4 = 2 or r1c4 = 3.

If r1c4 = 2, we have the situation represented below:

```
+-------------------------------+-------------------------------+
! 1        .        .        ! 2        .        .        !
! .        2        .        ! .        3        .        !
! .        .        3        ! .        .        1        !
+-------------------------------+-------------------------------+
! 2        .        .        ! 13*      1*       3*       !
! .        3        .        ! 1*       12*      2*       !
! .        .        1        ! 3*       2*       23*      !
+-------------------------------+-------------------------------+
```

We can now check how a contradiction could arise in b22, depending on where the three cells with no other candidates (no stars) are placed. There are 3 cases:
- cells 3 5 7 (anti-diagonal): 3 would appear twice in b22
- cells 2 4 9: 1 would appear twice in b22
- cells 1 6 8: 2 would appear twice in b22

Exercise for the reader: try all the possible values for r4c1 and r1c4 and show that all of them lead to the same three cell combinations in b22 for a contradiction to be obtained: (3 5 7), (2 4 9) and (1 6 8). It is boring but all the cases rely on the same reasoning. (Or see my full proof here:
http://forum.enjoysudoku.com/the-tridagon-rule-t39859.html.)

Conclusion: given the conditions at the start of this section, the Tridagon elimination rule is valid if and only if the pattern of 123-cells in block b22 is one of the following three:

```
. . +          . + .          + . .
. + .          + . .          . . +
+ . .          . . +          . + .
```

Remark: it has been noted that the first and second patterns are isomorphic (using isomorphisms that preserve r1c1 and the anti-diagonals in b11, b12 and b22) and that this makes only two non-isomorphic cases (with the above conditions) for the rule to apply. This is true, but I prefer to keep it in this form in anticipation of the forthcoming Tridagons-Forcing-Whip rules. With no condition on r1c1 fixed the three patterns are isomorphic.

A visual interpretation that can be useful for manual solvers has been given by eleven: one can draw one and only one rectangle with all its corners on the + cells.

As should be expected, the Tridagon elimination rule is enabled in SudoRules by setting ***?*Tridagons**** to TRUE in the configuration file.

14.2.2 Example: Loki

The best natural example of applying the tridagon elimination rule is the first mith puzzle I found not to be in T&E(2) and which made me interested in all this, the above-mentioned ***Loki***. If we apply this rule before any whips (using a TRIDAGONS set of preferences to be precisely defined in section 14.5), a puzzle in T&E(3) is simplified to a puzzle in T&E(1), more precisely in W3, at the low end of T&E(1):

```
+-------+-------+-------+
! 5 7 . ! . . . ! 9 . . !
! . . . ! . . . ! . . 8 !
! . 1 . ! . . . ! . . . !
+-------+-------+-------+
! . . 1 ! 6 8 . ! . 4 . !
! . . . ! . . 2 ! 8 . 9 !
! . . 2 ! . 9 4 ! 1 6 . !
+-------+-------+-------+
! . . . ! . 2 . ! . . . !
! . 6 . ! 9 . 8 ! 2 . 4 !
! . . . ! 4 1 . ! 6 . . !
+-------+-------+-------+
57....9.........8.1.........168..4......28.9..2.9416.....2.....6.9.82.4...41.6..
SER = 11.9

(solve-w-preferences
   "57....9.........8.1.........168..4......28.9..2.9416.....2.....6.9.82.4...41.6.."
   TRIDAGONS)
Resolution state after Singles and whips[1]:

   +----------------------+----------------------+----------------------+
   ! 5      7      3468   ! 238    346    13     ! 9      123    136    !
   ! 23469  2349   3469   ! 2357   34567  13579  ! 3457   12357  8      !
   ! 234689 1      34689  ! 23578  34567  3579   ! 3457   2357   3567   !
```

```
+----------------------+----------------------+----------------------+
! 379    359    1      ! 6      8      357    ! 357    4      2      !
! 3467   345    34567  ! 1      357    2      ! 8      357    9      !
! 378    358    2      ! 357    9      4      ! 1      6      357    !
+----------------------+----------------------+----------------------+
! 134789 34589  345789 ! 357    2      6      ! 357    135789 1357   !
! 137    6      357    ! 9      357    8      ! 2      1357   4      !
! 23789  23589  35789  ! 4      1      357    ! 6      35789  357    !
+----------------------+----------------------+----------------------+
205 candidates
```

```
hidden-pairs-in-a-column: c8{n8 n9}{r7 r9} ==> r9c8≠7, r9c8≠5, r9c8≠3, r7c8≠7, r7c8≠5,
r7c8≠3, r7c8≠1
```

```
+----------------------+----------------------+----------------------+
! 5      7      3468   ! 238    346    13     ! 9      123    136    !
! 23469  2349   3469   ! 2357   34567  13579  ! 3457   12357  8      !
! 234689 1      34689  ! 23578  34567  3579   ! 3457   2357   3567   !
+----------------------+----------------------+----------------------+
! 379    359    1      ! 6      8      357    ! 357    4      2      !
! 3467   345    34567  ! 1      357    2      ! 8      357    9      !
! 378    358    2      ! 357    9      4      ! 1      6      357    !
+----------------------+----------------------+----------------------+
! 134789 34589  345789 ! 357    2      6      ! 357    89     1357   !
! 137    6      357    ! 9      357    8      ! 2      1357   4      !
! 23789  23589  35789  ! 4      1      357    ! 6      89     357    !
+----------------------+----------------------+----------------------+
```

tridagon for digits 3, 5 and 7 in blocks:
b9, with cells: r8c8 (target cell), r7c7, r9c9
b8, with cells: r8c5, r7c4, r9c6
b6, with cells: r5c8, r4c7, r6c9
b5, with cells: r5c5, r4c6, r6c4
==> r8c8≠3,5,7

```
singles ==> r8c8=1, r1c9=1, r1c6=3, r1c8=2, r1c4=8, r2c6=1, r3c6=9, r3c9=6, r7c1=1
naked-triplets-in-a-row: r7{c4 c7 c9}{n3 n5 n7} ==> r7c3≠7, r7c3≠5, r7c3≠3, r7c2≠5,
r7c2≠3
Resolution state at the end of the TRIDAGONS set of preferences:
```

```
+--------------------+--------------------+--------------------+
! 5     7     46     ! 8     46    3      ! 9     2     1      !
! 23469 2349  3469   ! 257   4567  1      ! 3457  357   8      !
! 2348  1     348    ! 257   457   9      ! 3457  357   6      !
+--------------------+--------------------+--------------------+
! 379   359   1      ! 6     8     57     ! 357   4     2      !
! 3467  345   34567  ! 1     357   2      ! 8     357   9      !
! 378   358   2      ! 357   9     4      ! 1     6     357    !
+--------------------+--------------------+--------------------+
! 1     489   489    ! 357   2     6      ! 357   89    357    !
! 37    6     357    ! 9     357   8      ! 2     1     4      !
! 23789 23589 35789  ! 4     1     57     ! 6     89    357    !
+--------------------+--------------------+--------------------+
```

The end is easy solving in W3:

```
finned-x-wing-in-columns: n3{c4 c9}{r6 r7} ==> r7c7≠3
whip[1]: b9n3{r9c9 .} ==> r6c9≠3
z-chain[3]: r1c3{n4 n6} - c1n6{r2 r5} - c1n4{r5 .} ==> r2c2≠4, r3c3≠4, r2c3≠4
whip[3]: r1c3{n6 n4} - c1n4{r3 r5} - r5n6{c1 .} ==> r2c3≠6
whip[3]: r8c1{n3 n7} - c3n7{r9 r5} - r5n6{c3 .} ==> r5c1≠3
whip[3]: c5n3{r5 r8} - r8c1{n3 n7} - b4n7{r4c1 .} ==> r5c3≠3
whip[3]: c5n3{r5 r8} - r8c1{n3 n7} - b4n7{r4c1 .} ==> r5c5≠7
biv-chain[3]: c4n3{r7 r6} - r5c5{n3 n5} - c6n5{r4 r9} ==> r7c4≠5
whip[1]: r7n5{c9 .} ==> r9c9≠5
biv-chain[3]: b6n3{r4c7 r5c8} - r5c5{n3 n5} - r4c6{n5 n7} ==> r4c7≠7
whip[3]: r4n7{c1 c6} - r9n7{c6 c9} - r6n7{c9 .} ==> r8c1≠7
singles ==> r8c1=3, r7c4=3, r5c5=3, r4c7=3, r6c2=3, r6c1=8, r3c3=8, r2c3=3, r3c8=3,
r9c9=3
whip[1]: c3n9{r9 .} ==> r7c2≠9, r9c1≠9, r9c2≠9
biv-chain[2]: c8n5{r2 r5} - r6n5{c9 c4} ==> r2c4≠5
biv-chain[2]: c8n7{r2 r5} - r6n7{c9 c4} ==> r2c4≠7
stte
```

As will appear from further examples in this chapter, this makes Loki one of the easiest puzzles in the Tridagon family.

14.3 The complexity of the contradictory trivalue-oddagon pattern

The complexity to be considered here is about proving that the pattern is contradictory. This section will also illustrate a way of ***using CSP-Rules as an assistant theorem prover***, similarly to what was done for Slitherlink in [PBCS].

14.3.1 Proving the impossibility of the trivalue oddagon pattern

First of all, let's consider the "trivalue oddagon" impossible pattern in four blocks forming a rectangle, as at the beginning of section 14.1. Without making any other assumption, the most general possible resolution state is (modulo Sudoku isomorphisms):

```
+-------------------------------+-------------------------------+-------------------------------+
! 123       123456789 123456789 ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123       123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123       ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123       123456789 123456789 ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
! 123456789 123       123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123       ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
```

Theorem 14.1: the trivalue oddagon pattern can be proven contradictory in T&E(Singles, 4).

Proof: along the same lines as the proof in section 14.2.1. Whichever way four values are plugged into r1c1, r2c2, r1c4 and r4c1 (with no direct contradictions between them), one of the 123-cells in b4 has no possible value.

Notice that each of the following theorems also proves that *T&E(Singles, 3) is indeed enough*. See also section 14.6.2 for a direct proof of this result.

Theorem 14.2: after using eleven's replacement technique, the trivalue oddagon pattern can be proven contradictory in T&E(Singles, 1)

More precisely and somehow surprisingly:

Theorem 14.3: after using eleven's replacement technique, the trivalue oddagon pattern can be proven contradictory in Z5.

Proof: applying eleven's replacement to block b1 gives the following resolution state:

```
+-------------------------------+-------------------------------+-------------------------------+
! 1         123456789 123456789 ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 2         123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 3         ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123       123456789 123456789 ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
! 123456789 123       123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123       ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
```

Select resolution theory Z5 in the configuration file and apply function solve-sukaku-grid to the above resolution state:

```
biv-chain[3]: r1c4{n2 n3} - r2c5{n3 n1} - r3c6{n1 n2} ==> r1c5≠2, r1c6≠2, r3c4≠2,
r3c5≠2
biv-chain[3]: r1c4{n3 n2} - r3c6{n2 n1} - r2c5{n1 n3} ==> r1c5≠3, r1c6≠3, r2c4≠3,
r2c6≠3
biv-chain[3]: r2c5{n1 n3} - r1c4{n3 n2} - r3c6{n2 n1} ==> r2c4≠1, r2c6≠1, r3c4≠1,
r3c5≠1
biv-chain[3]: r4c1{n2 n3} - r5c2{n3 n1} - r6c3{n1 n2} ==> r4c3≠2, r5c1≠2, r5c3≠2,
r6c1≠2
biv-chain[3]: r4c1{n3 n2} - r6c3{n2 n1} - r5c2{n1 n3} ==> r4c2≠3, r5c1≠3, r6c1≠3,
r6c2≠3
biv-chain[3]: r5c2{n1 n3} - r4c1{n3 n2} - r6c3{n2 n1} ==> r4c2≠1, r4c3≠1, r5c3≠1,
r6c2≠1
z-chain[4]: b4n1{r6c3 r5c2} - b4n3{r5c2 r4c1} - r4c6{n3 n2} - r3c6{n2 .} ==> r6c6≠1
```

```
z-chain[4]: b4n3{r4c1 r5c2} - b4n1{r5c2 r6c3} - r6c4{n1 n2} - r1c4{n2 .} ==> r4c4≠3
z-chain[4]: r4c1{n2 n3} - r4c6{n3 n1} - r6c4{n1 n3} - r1c4{n3 .} ==> r4c4≠2
z-chain[4]: r6c3{n2 n1} - r6c4{n1 n3} - r4c6{n3 n1} - r3c6{n1 .} ==> r6c6≠2
z-chain[5]: b4n1{r6c3 r5c2} - b4n3{r5c2 r4c1} - r4c6{n3 n2} - r5c5{n2 n3} - r2c5{n3 .}
==> r6c5≠1
z-chain[5]: b4n3{r4c1 r5c2} - b4n1{r5c2 r6c3} - r6c4{n1 n2} - r5c5{n2 n1} - r2c5{n1 .}
==> r4c5≠3
z-chain[5]: b4n2{r6c3 r4c1} - b4n3{r4c1 r5c2} - r5c5{n3 n1} - b2n1{r2c5 r3c6} -
b2n2{r3c6 .} ==> r6c4≠2
biv-chain[3]: r6c4{n1 n3} - r1c4{n3 n2} - r3c6{n2 n1} ==> r4c6≠1, r5c6≠1
biv-chain[2]: r4c6{n3 n2} - r4c1{n2 n3} ==> r4c7≠3, r4c8≠3, r4c9≠3
biv-chain[2]: r4c1{n2 n3} - r4c6{n3 n2} ==> r4c7≠2, r4c8≠2, r4c9≠2, r4c5≠2
biv-chain[3]: b4n2{r6c3 r4c1} - r4c6{n2 n3} - r6c4{n3 n1} ==> r6c3≠1
singles ==> r6c3=2, r4c1=3, r4c6=2, r3c6=1, r2c5=3, r1c4=2, r5c5=1
GRID 0 HAS NO SOLUTION : NO CANDIDATE FOR FOR BN-CELL b4n1
```

Note: a shorter resolution path can be obtained if Subsets[3] are allowed (three Triplets start by cleaning blocks b2, b4, b5 instead of all the bivalue-chains[3]) – block b1 is in any case cleaned by ECP), but this is not my point here, and anyway it still requires Z5. I give it here nevertheless because it shows more clearly the essential point about the z-chain[5] immediately after the Triplets:

```
naked-triplets-in-a-block: b5{r4c6 r5c5 r6c4}{n3 n2 n1} ==> r6c6≠3, r6c6≠2, r6c6≠1,
r6c5≠3, r6c5≠2, r6c5≠1, r5c6≠3, r5c6≠2, r5c6≠1, r5c4≠3, r5c4≠2, r5c4≠1, r4c5≠3,
r4c5≠2, r4c5≠1, r4c4≠3, r4c4≠2, r4c4≠1
naked-triplets-in-a-block: b4{r4c1 r5c2 r6c3}{n2 n3 n1} ==> r6c2≠3, r6c2≠1, r6c1≠3,
r6c1≠2, r5c3≠2, r5c3≠1, r5c1≠3, r5c1≠2, r4c3≠2, r4c3≠1, r4c2≠3, r4c2≠1
naked-triplets-in-a-block: b2{r1c4 r2c5 r3c6}{n2 n3 n1} ==> r3c5≠2, r3c5≠1, r3c4≠2,
r3c4≠1, r2c6≠3, r2c6≠1, r2c4≠3, r2c4≠1, r1c6≠3, r1c6≠2, r1c5≠3, r1c5≠2
z-chain[5]: b2n1{r2c5 r3c6} - b2n2{r3c6 r1c4} - b5n2{r6c4 r4c6} - b4n2{r4c1 r6c3} -
b4n1{r6c3 .} ==> r5c5≠1
biv-chain[3]: r5c5{n2 n3} - r2c5{n3 n1} - r3c6{n1 n2} ==> r4c6≠2
biv-chain[3]: b5n2{r5c5 r6c4} - r6c3{n2 n1} - r5c2{n1 n3} ==> r5c5≠3
naked-single ==> r5c5=2
biv-chain[3]: r6c4{n1 n3} - r1c4{n3 n2} - r3c6{n2 n1} ==> r4c6≠1
singles ==> r4c6=3, r4c1=2, r6c3=1
GRID 0 HAS NO SOLUTION : NO CANDIDATE FOR FOR BN-CELL b5n1
```

14.3.2 Degenerated forms of the trivalue oddagon pattern

By using T&E restricted to cells in the pattern, I have checked that degenerated forms of the contradictory trivalue oddagon pattern, i.e. forms that have a missing 123-candidate in one of the twelve tridagon cells or forms that have a decided candidate in one of these cells, are still contradictory (obviously) but they no longer require T&E(3) to be proven so. For the proofs, see Sudoku/Tridagons/Proofs in [CSP-RULES-EXAMPLES]. Not only is the trivalue oddagon pattern brittle, its position in T&E(3) (as a contradictory pattern) is also brittle.

14.4 Tridagon links

Other applications of the impossible trivalue oddagon pattern are known. My goal in this section is to review only the simplest case (after the basic Tridagon rule), where two and only two different cells of the impossible pattern have one and only one additional candidate each. In this case, the conclusion is that one of the additional candidates must be True – which doesn't define a resolution rule in and of itself. But it allows to conclude on the existence of an OR relation between these two candidates, which can then be used to build new types of Forcing Chains.

Based on the analysis of section 14.2, it is easy to see that there are the following cases (and only them). By the same isomorphisms as before, we can always suppose one of the two additional candidates is in r1c1 and the pattern of cells is as before:

```
# . . | + . .
. + . | . + .
. . + | . . +
-------------
+ . . | x x x
. + . | x x x
. . + | x x x
```

with the same conditions on the pattern in the fourth block, i.e. one of:

```
. . +       . + .       + . .
. + .       + . .       . . +
+ . .       . . +       . + .
```

Now, the question is: where can the second candidate be? Notice that the following four cases are possible and have real examples.

<> First case: in the same block. Only one possible place modulo isomorphisms:

```
# . . | + . .
. # . | . + .
. . + | . . +
-------------
+ . . | x x x
. + . | x x x
. . + | x x x
```

<> Second case: in a different block but the same band. Only two possible places modulo isomorphisms:

same row:

```
# . . | # . .
. + . | . + .
. . + | . . +
-------------
+ . . | x x x
. + . | x x x
. . + | x x x
```

different row:

```
# . . | + . .
. + . | . # .
. . + | . . +
-------------
+ . . | x x x
. + . | x x x
. . + | x x x
```

<> Third case: in a different block but the same stack. Only two possible places modulo isomorphisms (deduced from the previous two by row/column symmetry).

<> Fourth case: in opposite blocks. It can be almost anywhere in the 4th block (which must still satisfy one of the three patterns found in the section 14.2).

```
# . . | + . .
. + . | . + .
. . + | . . +
-------------
+ . . | x x x
. + . | x x x
. . + | x x x
```

14.5 Tridagon-forcing-whips

The chain pattern discussed in this section are special cases of the Tridagon-OR_k-Forcing-Whips to be introduced later. I defined them before I introduced the notion of an OR_k relation and the generic OR_k chains that can use them. I keep them here as a step towards the more general ones.

14.5.1 General remarks on forcing chains

Whenever an OR relation OR(C1, C2) between two candidates C1 and C2 has been proven, assertions and eliminations can easily be obtained by resorting to reasoning by cases. This is not very elegant as it relies on two different streams of reasoning, but it can be useful. More precisely, consider two different chains, one based on C1 and one based on C2. It can be concluded in an obvious way that:

– any candidate X that is linked to some candidate with positive valence in each chain can be eliminated,

– and any candidate that has positive valence in both chains can be asserted as a decided value.

As a special case of X having negative valence, one or two of the chains can have length 0 (which means that the corresponding Ci is linked to X).

As for the complexity assigned to such OR-forcing-chains in my usual approach of chains: the OR(C1, C2) relation isn't supposed to be proven with the use of a crystal ball. It involves some particular pattern with its own length. Similarly, each of the partial-chains has its own length, resp. p_1 and p_2. The length of the OR-forcing-chain is thus naturally assigned value $n = k + p_1 + p_2$. Indeed, there is no choice in my standard approach: no other possible value would be consistent with my other ratings for the hardest step in a resolution path. This view also reflects the idea that ***every elimination is self-contained***.

One might object that if the same OR relation is used several times in the resolution path, this is "unfair". But this would rely on a misunderstanding of the fundamental nature of all my ratings: they are ratings of the hardest elimination/ assertion step and how many times this highest rating is reached in the resolution path is totally irrelevant. One might prefer a rating of a full resolution path and I understand this, but as of now, any such proposed rating is totally inconsistent.

In my approach, the typical OR-forcing-chain[n] is a forcing-whip[n], the OR relation being a mere bivalue relation (length 1) and I have already shown long ago that the length of this forcing-whip must be defined as $n = 1 + p_1 + p_2$.

Note: see sections 14.9 to 14.11 for a different view of counting complexity in Tridagon-OR_k-chains – or more generally in cases the OR_k relation is based on an exotic pattern. The way complexity is defined is the main difference between the Tridagon-forcing-whips defined in this section and the more general OR_k-chains to be analysed later. The fact is, due to this different way of defining complexity, solutions with the more general chains will be simpler because the Tridagon pattern will be detected sooner and the associated OR_k-chain rules will be applied sooner.

14.5.2 Tridagon-forcing-whips

Definition: a Tridagon-forcing-whip[n] is a particular type of OR forcing-chain where:
- the partial-chains involved are partial-whips, of respective lengths p_1 and p_2;
- the OR relation involved is a Tridagon-link[12];
- $n = 12 + p_1 + p_2$.

Remarks:
- in order to avoid considering special cases when p_1 and/or p_2 is 0, a direct link will be considered as a partial-whip[0];
- having a well-defined length, these application-specific forcing chains have a natural place in the CSP-Rules hierarchy. However, as for any other rules, various strategies allow to change this default hierarchy. More on this later.

Notation: see the example below. The "–" symbol at the end of the lines for the partial-whips means as usual a direct contradiction link.

As should be expected, the Tridagon-forcing-whip rules are enabled in SudoRules by setting ***?*Tridagon-Forcing-Whips**** to TRUE in the configuration file. In such a case, it is suggested to restrict both the maximum length of all the chains and the maximum length of Tridagon-Forcing-Whips, and to increase them only progressively in order to avoid potential memory overflows, e.g. to start with:

```
(bind ?*all-chains-max-length* 7)
(bind ?*tridagon-forcing-whips-max-length* 15)
```

Example (#147 in mith's database of 246 T&E(3) puzzles), showing two Tridagon-forcing-whips of length 14. Here whips-max-length has been restricted to 4, in order to allow an earlier detection of the tridagon-link.

```
+-------+-------+-------+
! . . . ! . . . ! . 1 2 !
! . . . ! . . . ! 3 . 4 !
! . . . ! . 1 5 ! 6 7 . !
+-------+-------+-------+
! . . 8 ! . 9 . ! . . 6 !
! . 5 1 ! . 7 6 ! . . . !
! 6 9 . ! . . 8 ! . . . !
+-------+-------+-------+
! . 7 5 ! . 8 9 ! . 6 . !
! 8 . . ! 6 5 . ! . 9 . !
! 9 . 6 ! 7 . 1 ! . . . !
+-------+-------+-------+
.......12......3.4....1567...8.9...6.51.76...69...8....75.89.6.8..65..9.9.67.1...;31
SER = 11.7

Resolution state after Singles and whips[1]:

+-------------------+-------------------+-------------------+
! 3457  3468  3479  ! 3489  346   347   ! 589   1     2     !
! 1257  1268  279   ! 289   26    27    ! 3     58    4     !
! 234   2348  2349  ! 23489 1     5     ! 6     7     89    !
+-------------------+-------------------+-------------------+
! 2347  234   8     ! 12345 9     234   ! 12457 2345  6     !
! 234   5     1     ! 234   7     6     ! 2489  2348  389   !
! 6     9     2347  ! 12345 234   8     ! 12457 2345  1357  !
```

```
              +-------------------+-------------------+-------------------+
              ! 1234  7     5     ! 234   8     9     ! 124   6     13    !
              ! 8     1234  234   ! 6     5     234   ! 1247  9     137   !
              ! 9     234   6     ! 7     234   1     ! 2458  23458 358   !
              +-------------------+-------------------+-------------------+
              176 candidates
hidden-pairs-in-a-column: c4{n1 n5}{r4 r6} ==> r6c4≠4, r6c4≠3, r6c4≠2, r4c4≠4, r4c4≠3,
r4c4≠2
whip[4]: r4n7{c1 c7} - r4n1{c7 c4} - r4n5{c4 c8} - r2n5{c8 .} ==> r2c1≠7

              +-------------------+-------------------+-------------------+
              ! 3457  3468  3479  ! 3489  346   347   ! 589   1     2     !
              ! 125   1268  279   ! 289   26    27    ! 3     58    4     !
              ! 234   2348  2349  ! 23489 1     5     ! 6     7     89    !
              +-------------------+-------------------+-------------------+
              ! 2347  234   8     ! 15    9     234   ! 12457 2345  6     !
              ! 234   5     1     ! 234   7     6     ! 2489  2348  389   !
              ! 6     9     2347  ! 15    234   8     ! 12457 2345  1357  !
              +-------------------+-------------------+-------------------+
              ! 1234  7     5     ! 234   8     9     ! 124   6     13    !
              ! 8     1234  234   ! 6     5     234   ! 1247  9     137   !
              ! 9     234   6     ! 7     234   1     ! 2458  23458 358   !
              +-------------------+-------------------+-------------------+

tridagon-link (verti, different column) for digits 2, 3 and 4 in blocks:
        b7, with cells: r7c1 (link cell), r8c3, r9c2
        b8, with cells: r7c4, r8c6, r9c5
        b4, with cells: r5c1, r6c3 (link cell), r4c2
        b5, with cells: r5c4, r6c5, r4c6
 ==> tridagon-link[12](n1r7c1, n7r6c3)

tridagon-forcing-whip-elim[14] based on tridagon-link(n7r6c3, n1r7c1)
   || n7r6c3: partial-whip[1]: c9n7{r6 r8} -
   || n1r7c1: partial-whip[1]: r7c9{n1 n3} -
 ==> r8c9≠3

tridagon-forcing-whip-elim[14] based on tridagon-link(n1r7c1, n7r6c3)
   || n1r7c1: -
   || n7r6c3: partial-whip[2]: c1n7{r4 r1} - c1n5{r1 r2} -
 ==> r2c1≠1

singles ==> r2c2=1, r1c2=6, r3c2=8, r3c9=9, r5c7=9, r2c5=6, r7c1=1, r7c9=3, r5c9=8,
r9c9=5
hidden-pairs-in-a-block: b9{n1 n7}{r8c7 r8c9} ==> r8c7≠4, r8c7≠2
hidden-pairs-in-a-block: b2{n8 n9}{r1c4 r2c4} ==> r2c4≠2, r1c4≠4, r1c4≠3
finned-x-wing-in-columns: n2{c5 c8}{r9 r6} ==> r6c7≠2
z-chain[4]: r1c5{n3 n4} - r1c6{n4 n7} - b1n7{r1c1 r2c3} - c3n9{r2 .} ==> r1c3≠3
z-chain[2]: b1n3{r3c1 r3c3} - c4n3{r3 .} ==> r5c1≠3
z-chain[3]: c4n3{r5 r3} - c3n3{r3 r8} - b8n3{r8c6 .} ==> r6c5≠3
biv-chain[3]: r6c5{n2 n4} - r1c5{n4 n3} - c4n3{r3 r5} ==> r5c4≠2
z-chain[4]: c5n2{r6 r9} - r9n3{c5 c2} - b4n3{r4c2 r4c1} - b4n7{r4c1 .} ==> r6c3≠2
z-chain[3]: c3n2{r3 r8} - c6n2{r8 r4} - b4n2{r4c2 .} ==> r2c1≠2
```

```
singles ==> r2c1=5, r2c8=8, r1c7=5, r2c4=9, r1c4=8, r1c3=9, r9c7=8
finned-swordfish-in-columns: n2{c4 c7 c1}{r3 r7 r4} ==> r4c2≠2
whip[1]: c2n2{r9 .} ==> r8c3≠2
whip[1]: c3n2{r3 .} ==> r3c1≠2
biv-chain[3]: r4c2{n4 n3} - r6n3{c3 c8} - c8n5{r6 r4} ==> r4c8≠4
biv-chain[4]: b5n2{r6c5 r4c6} - r2c6{n2 n7} - c3n7{r2 r6} - r6n3{c3 c8} ==> r6c8≠2
hidden-single-in-a-row ==> r6c5=2
naked-pairs-in-a-row: r4{c2 c6}{n3 n4} ==> r4c8≠3, r4c7≠4, r4c1≠4, r4c1≠3
whip[1]: c1n3{r3 .} ==> r3c3≠3
finned-x-wing-in-rows: n3{r4 r9}{c2 c6} ==> r8c6≠3
singles ==> r9c5=3, r1c5=4
finned-x-wing-in-columns: n4{c6 c2}{r4 r8} ==> r8c3≠4
stte
```

14.5.3 The TRIDAGONS and TRIDAGON-FW sets of preferences

In order to allow applying Tridagons before whips or other resolution rules, a ***TRID*** set of preferences is defined, made of:
- all the Subset and Finned Fish resolution rules;
- the Tridagon elimination rule.

Similarly, in order to allow applying Tridagons and Tridagon-forcing-whips before whips or other resolution rules, a ***TRID-FW*** set of preferences is defined, made of:
- the same rules as in the TRIDAGONS module;
- the rules for finding the tridagon-links;
- the rules for the Tridagon-Forcing-Whips (up to the length specified in the configuration file by ?*tridagon-forcing-whips-max-length*).

Example of application (again a puzzle created by mith, #2 in his database of 246 not in T&E(3)):

```
+-------+-------+-------+
! . . . ! . . . ! . . 1 !
! . . . ! . . 2 ! . 3 . !
! . . . ! . 4 . ! 5 6 . !
+-------+-------+-------+
! . . . ! . . 7 ! . . . !
! . . 4 ! 8 1 . ! 2 . . !
! 1 9 . ! . 2 4 ! 8 . . !
+-------+-------+-------+
! . 8 9 ! . . 1 ! . . 7 !
! 4 2 . ! . 7 8 ! . . 9 !
! 7 . 1 ! . 9 . ! . . . !
+-------+-------+-------+
........1.....2.3.....4.56......7.....481.2..19..248...89..1..742..78..97.1.9....;28

(solve-w-preferences
"........1.....2.3.....4.56......7.....481.2..19..248...89..1..742..78..97.1.9....;28"
TRIDAGON-FW)
```

```
Resolution state after Singles and whips[1]:
        +---------------------+---------------------+---------------------+
        ! 235689 34567  235678 ! 35679  3568   3569   ! 479    2489   1      !
        ! 5689   14567  5678   ! 15679  568    2      ! 479    3      48     !
        ! 2389   137    2378   ! 1379   4      39     ! 5      6      28     !
        +---------------------+---------------------+---------------------+
        ! 23568  356    23568  ! 3569   356    7      ! 13469  1459   3456   !
        ! 356    3567   4      ! 8      1      3569   ! 2      579    356    !
        ! 1      9      3567   ! 356    2      4      ! 8      57     356    !
        +---------------------+---------------------+---------------------+
        ! 356    8      9      ! 23456  356    1      ! 346    245    7      !
        ! 4      2      356    ! 356    7      8      ! 136    15     9      !
        ! 7      356    1      ! 23456  9      356    ! 346    2458   234568 !
        +---------------------+---------------------+---------------------+
        196 candidates

tridagon-link (horiz, different row) for digits 3, 5 and 6 in blocks:
        b4, with cells: r6c3 (link cell), r5c1, r4c2
        b5, with cells: r6c4, r5c6 (link cell), r4c5
        b7, with cells: r8c3, r7c1, r9c2
        b8, with cells: r8c4, r7c5, r9c6
 ==> tridagon-link[12](n7r6c3, n9r5c6)

tridagon-forcing-whip-elim[13] based on tridagon-link(n9r5c6, n7r6c3)
   || n9r5c6: -
   || n7r6c3: partial-whip[1]: r5n7{c2 c8} -
 ==> r5c8≠9

singles ==> r5c6=9, r3c6=3
naked-pairs-in-a-block: b6{r5c8 r6c8}{n5 n7} ==> r6c9≠5, r5c9≠5, r4c9≠5, r4c8≠5
singles ==> r9c9=5, r8c8=1, r9c6=6, r1c6=5,r9c2=3, r9c7=4, r7c8=2, r7c4=4, r9c8=8,
r9c4=2, r3c9=2, r2c9=8, r2c5=6, r1c5=8, r1c8=4, r4c8=9, r2c2=4, r3c2=1, r2c4=1,
r4c9=4, r4c7=1
whip[1]: r4n3{c5 .} ==> r6c4≠3
whip[1]: c2n5{r5 .} ==> r5c1≠5, r6c3≠5
naked-pairs-in-a-row: r5{c1 c9}{n3 n6} ==> r5c2≠6
naked-pairs-in-a-row: r1{c4 c7}{n7 n9} ==> r1c3≠7, r1c2≠7, r1c1≠9
stte
```

After the Tridagon-forcing-whip[13] has been applied, nothing more complex than Naked-Pairs is required.

This is an example where not having whips intermingled with Tridagon-forcing-whips, as they would be in the default simplest-first strategy, simplifies the resolution path. However, be aware that this will not always be the case and that this may on the contrary lead to much more complicated paths.

14.5.4 Where to find more examples

As puzzles relying on the trivalue oddagon impossible pattern are extremely rare and can only be found by specifically searching for them, it seems necessary to provide some reference of where one can find a few. Mith is the main creator of such puzzles. Check the various editions of his database, starting e.g. from the version with 246 puzzles here: http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539-1190.html. In [CSP-RULES-EXAMPLES], see the Sudoku/Tridagons folder.

14.5.5 The effectiveness of Tridagons and Tridagon-forcing-whips

As of the writing of this section , the largest database of puzzles related to trivalue-oddagons consisted of 972 puzzles in T&E(3) found by mith after he changed his search criterion from max SER to T&E-depth:
(https://docs.google.com/spreadsheets/d/1t-PsJT-pKGQEWjSbbNBXzLcxb5Inmoosztu9ZVCW_M/edit#gid=0). (See section 14.8 for a much larger database.) See also Sudoku/mith-972-TE3 in [CSP-RULES-EXAMPLES] for more details.

These puzzles may not be minimal, but they are expanded forms of minimal puzzles (i.e. all the Singles have been applied). This avoids part of the redundancy (several minimal puzzles having the same expanded form), although much redundancy remains. Only what I've called the ***min-expands*** (the puzzles in this database that don't have super-puzzles in it) are of real interest (they play among expanded forms the same role as the classical minimals do in random collections).

Considering the way this database was produced (neighbourhood search, expansion by Singles, re-search…), it is obviously very biased; don't take the following results in any statistical sense. It can nevertheless be interesting to note that, of the 972 puzzles:
- 216 can be solved using only Subsets + Finned Fish + the Tridagon elimination rule;
- 505 more can be solved if one adds whips[≤12] to the previous rules;
- 41 more can be solved if one adds Tridagon-Forcing-Whips[≤15];
which leaves 210 of them unsolved (section 14.6 will show how to solve them).

14.6 Automating eleven's method in the context of trivalue oddagons

The attentive reader will have noticed that the tridagon related situations offer large opportunities for applying eleven's very general replacement technique.

In section 6.10.5, I defined user functions for applying the replacement technique and I said there was currently no general way to start it automatically in SudoRules. However, there is an exception when some anti-tridagon pattern (see section 14.8) is present in some resolution state. This behaviour can be activated by setting global variable ?*Eleven-Replacement-in-Tridagons* to True in the configuration file.

The conditions are very general: there must be a block with three cells having exactly the three digits and three other blocks with three cells each, satisfying the same pattern as in the trivalue oddagon, but with the cells being allowed to have more candidates than the 123 ones. One may consider it as restrictive to have a mandatory block with exactly the three digits in the three cells and to have each of the three digits in all the other nine cells. But this is an experimental set of conditions and it may change in the future; in any case, those are the conditions the results reported in the next two subsections rely on.

Notice that nothing forbids to apply the technique several times to the same digits on the same puzzle.

14.6.1 Example of an automatic start of the replacement

Consider puzzle #396 in the 972 database (chosen here because it's the hardest one I found in this collection with automatic replacement applied – see the next subsection). In the configuration file, chains are active up to length 9 and ?*Eleven-Replacement-in-Tridagons* is True.

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

```
...4.....4...89...68.37..4..68.479..73.96.4..9.48.36..3......52....3..9.87....3..
SER = 11.6

Resolution state after Singles and whips[1]:
   +----------------------+----------------------+----------------------+
   ! 125     1259    123579 ! 4       125     1256   ! 12578   13678   136789 !
   ! 4       125     12357  ! 1256    8       9      ! 1257    1367    1367   !
   ! 6       8       1259   ! 3       7       125    ! 125     4       19     !
   +----------------------+----------------------+----------------------+
   ! 125     6       8      ! 125     4       7      ! 9       123     135    !
   ! 7       3       125    ! 9       6       125    ! 4       128     158    !
   ! 9       125     4      ! 8       125     3      ! 6       127     157    !
   +----------------------+----------------------+----------------------+
   ! 3       149     169    ! 167     19      1468   ! 178     5       2      !
   ! 125     1245    1256   ! 12567   3       124568 ! 178     9       14678  !
   ! 8       7       12569  ! 1256    1259    12456  ! 3       16      146    !
   +----------------------+----------------------+----------------------+
   179 candidates.

hidden-pairs-in-a-column: c3{n3 n7}{r1 r2} ==> r2c3≠5, r2c3≠2, r2c3≠1, r1c3≠9, r1c3≠5,
r1c3≠2, r1c3≠1
```

```
whip[7]: b9n6{r9c9 r8c9} - r1n6{c9 c8} - c8n8{r1 r5} - c9n8{r5 r1} - r1n3{c9 c3} -
r1n7{c3 c7} - b9n7{r7c7 .} ==> r9c6≠6

***** STARTING ELEVEN'S REPLACEMENT TECHNIQUE FOR GENERAL TRIDAGON in resolution
state: *****
        +----------------------+----------------------+----------------------+
        ! 125    1259   37     ! 4      125    1256   ! 12578  13678  136789 !
        ! 4      125    37     ! 1256   8      9      ! 1257   1367   1367   !
        ! 6      8      1259   ! 3      7      125    ! 125    4      19     !
        +----------------------+----------------------+----------------------+
        ! 125    6      8      ! 125    4      7      ! 9      123    135    !
        ! 7      3      125    ! 9      6      125    ! 4      128    158    !
        ! 9      125    4      ! 8      125    3      ! 6      127    157    !
        +----------------------+----------------------+----------------------+
        ! 3      149    169    ! 167    19     1468   ! 178    5      2      !
        ! 125    1245   1256   ! 12567  3      124568 ! 178    9      14678  !
        ! 8      7      12569  ! 1256   1259   1245   ! 3      16     146    !
        +----------------------+----------------------+----------------------+
Trying in block 5:

      +---------------------------+---------------------------+---------------------------+
      ! 125      1259     37       ! 4        125      1256     ! 12578    1235678  12356789 !
      ! 4        125      37       ! 1256     8        9        ! 1257     123567   123567   !
      ! 6        8        1259     ! 3        7        125      ! 125      4        1259     !
      +---------------------------+---------------------------+---------------------------+
      ! 125      6        8        ! 5        4        7        ! 9        1235     1235     !
      ! 7        3        125      ! 9        6        2        ! 4        1258     1258     !
      ! 9        125      4        ! 8        1        3        ! 6        1257     1257     !
      +---------------------------+---------------------------+---------------------------+
      ! 3        12459    12569    ! 12567    1259     124568   ! 12578    125      125      !
      ! 125      1245     1256     ! 12567    3        124568   ! 12578    9        1245678  !
      ! 8        7        12569    ! 1256     1259     1245     ! 3        1256     12456    !
      +---------------------------+---------------------------+---------------------------+

finned-x-wing-in-columns: n5{c1 c5}{r1 r8} ==> r8c6≠5
z-chain[4]: b4n5{r6c2 r5c3} - b4n1{r5c3 r4c1} - r1c1{n1 n2} - r1c5{n2 .} ==> r1c2≠5
z-chain[4]: r6c2{n2 n5} - r2c2{n5 n1} - r1c1{n1 n5} - r1c5{n5 .} ==> r1c2≠2
z-chain[5]: r1c5{n5 n2} - r1c1{n2 n1} - r4c1{n1 n2} - r6c2{n2 n5} - r2n5{c2 .} ==>
r1c9≠5, r1c8≠5, r1c7≠5
z-chain[5]: b4n1{r5c3 r4c1} - b4n2{r4c1 r6c2} - b1n2{r2c2 r1c1} - r1c5{n2 n5} -
r3c6{n5 .} ==> r3c3≠1
whip[5]: c6n8{r8 r7} - r7n4{c6 c2} - c2n9{r7 r1} - c2n1{r1 r2} - c4n1{r2 .} ==> r8c6≠1
whip[6]: r1c5{n2 n5} - c1n5{r1 r8} - c1n2{r8 r4} - r6c2{n2 n5} - b1n5{r2c2 r3c3} -
r3n2{c3 .} ==> r1c9≠2
whip[6]: r1c5{n2 n5} - c1n5{r1 r8} - c1n2{r8 r4} - r6c2{n2 n5} - b1n5{r2c2 r3c3} -
r3n2{c3 .} ==> r1c8≠2
whip[6]: r1c5{n2 n5} - c1n5{r1 r8} - c1n2{r8 r4} - r6c2{n2 n5} - b1n5{r2c2 r3c3} -
r3n2{c3 .} ==> r1c7≠2
whip[5]: r1n2{c5 c1} - r4c1{n2 n1} - r5c3{n1 n5} - r3c3{n5 n9} - r9n9{c3 .} ==> r9c5≠2
whip[4]: c5n5{r7 r1} - c5n2{r1 r7} - r7c8{n2 n1} - r7c9{n1 .} ==> r7c6≠5
whip[6]: b1n1{r1c2 r1c1} - r1n2{c1 c5} - r1n5{c5 c6} - r3c6{n5 n1} - r9c6{n1 n4} -
r7n4{c6 .} ==> r7c2≠1
```

```
whip[6]: c4n1{r7 r2} - r3c6{n1 n5} - r1n5{c5 c1} - b1n1{r1c1 r1c2} - c2n9{r1 r7} -
r7n4{c2 .} ==> r7c6≠1
z-chain[7]: r3n9{c9 c3} - c2n9{r1 r7} - r7n4{c2 c6} - r7n8{c6 c7} - c7n5{r7 r8} -
c1n5{r8 r1} - r2n5{c2 .} ==> r3c9≠5
whip[8]: r3c6{n1 n5} - r3c7{n5 n2} - r3c3{n2 n9} - r9n9{c3 c5} - b8n5{r9c5 r7c5} -
r7c9{n5 n2} - r7c8{n2 n1} - b6n1{r5c8 .} ==> r3c9≠1
whip[8]: r6c2{n2 n5} - r2c2{n5 n1} - r8c2{n1 n4} - r7n4{c2 c6} - c6n8{r7 r8} -
c6n6{r8 r1} - r2c4{n6 n2} - b8n2{r7c4 .} ==> r7c2≠2
whip[8]: c1n5{r8 r1} - b2n5{r1c5 r3c6} - c3n5{r3 r5} - r6c2{n5 n2} - r2c2{n2 n1} -
b2n1{r2c4 r1c6} - r9c6{n1 n4} - r7n4{c6 .} ==> r7c2≠5
whip[9]: c1n5{r8 r1} - b2n5{r1c5 r3c6} - r3n1{c6 c7} - c7n5{r3 r2} - c2n5{r2 r6} -
c8n5{r6 r5} - c8n8{r5 r1} - r1c7{n8 n7} - b9n7{r7c7 .} ==> r8c9≠5
whip[9]: c6n8{r8 r7} - r7n4{c6 c2} - r8n4{c2 c9} - r8n8{c9 c7} - b9n7{r8c7 r7c7} -
r1c7{n7 n1} - r1c6{n1 n5} - r1c1{n5 n2} - r1c5{n2 .} ==> r8c6≠6
whip[7]: r7n7{c4 c7} - r7n8{c7 c6} - c6n6{r7 r1} - r2c4{n6 n2} - b8n2{r7c4 r7c5} -
r7c8{n2 n5} - r7c9{n5 .} ==> r7c4≠1
whip[7]: r3c6{n1 n5} - r9c6{n5 n4} - r8c6{n4 n8} - b9n8{r8c7 r7c7} - r1c7{n8 n7} -
b9n7{r7c7 r8c9} - c9n4{r8 .} ==> r1c6≠1
whip[5]: c1n5{r8 r1} - b2n5{r1c5 r3c6} - b2n1{r3c6 r2c4} - r2c2{n1 n2} - b4n2{r6c2 .}
==> r8c1≠2
biv-chain[3]: r8c1{n5 n1} - r4c1{n1 n2} - r6c2{n2 n5} ==> r8c2≠5
whip[5]: c2n5{r2 r6} - b4n2{r6c2 r4c1} - r1c1{n2 n5} - b2n5{r1c5 r3c6} - b2n1{r3c6 .}
==> r2c2≠1
whip[1]: b1n1{r1c1 .} ==> r1c9≠1, r1c7≠1, r1c8≠1
naked-pairs-in-a-column: c2{r2 r6}{n2 n5} ==> r8c2≠2
whip[1]: b7n2{r9c3 .} ==> r3c3≠2
whip[1]: r3n2{c9 .} ==> r2c9≠2, r2c8≠2, r2c7≠2
z-chain[3]: r7n1{c9 c3} - r8c2{n1 n4} - c9n4{r8 .} ==> r9c9≠1
whip[3]: r1c7{n7 n8} - b9n8{r7c7 r8c9} - b9n7{r8c9 .} ==> r2c7≠7
biv-chain[4]: r3c9{n2 n9} - r1n9{c9 c2} - r1n1{c2 c1} - c1n2{r1 r4} ==> r4c9≠2
z-chain[4]: r2c7{n5 n1} - b2n1{r2c4 r3c6} - r3n5{c6 c3} - c1n5{r1 .} ==> r8c7≠5
whip[1]: r8n5{c1 .} ==> r7c3≠5, r9c3≠5
z-chain[4]: r2c7{n1 n5} - c2n5{r2 r6} - r5c3{n5 n1} - r7n1{c3 .} ==> r8c7≠1
whip[4]: r8c6{n4 n8} - b9n8{r8c7 r7c7} - b9n7{r7c7 r8c7} - r1c7{n7 .} ==> r8c9≠4
hidden-single-in-a-block ==> r9c9=4
naked-pairs-in-a-column: c6{r3 r9}{n1 n5} ==> r1c6≠5
naked-single ==> r1c6=6
hidden-pairs-in-a-row: r1{n2 n5}{c1 c5} ==> r1c1≠1
stte
```

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

By comparing block 9 in this solution with the original puzzle, we see that digits 1 and 2 must be permuted.

14.6.2 How powerful is eleven's method in the context of trivalue oddagons?

There are two answers to the question.

1) Considering mith's database of 972 puzzles in T&E(3), it appeared at the end of section 14.4 that 210 of them remained unsolved by Tridagons + whips[≤12] + Tridagon-forcing-whips[≤15]. All of them can be solved by applying the above defined "eleven replacement in tridagons" technique.

2) Forgetting any tridagon resolution rule and considering the full 972 database, all of the puzzles in it can be solved by whips (of length ≤ 9) plus eleven replacement in tridagons.

Now, considering theorem 14.3, there might appear another question: beyond the theoretical interest of this theorem, does it imply anything for real puzzles? A valid puzzle can't have the trivalue oddagon pattern; it must have at least one additional candidate in one of the 123 cells. In such conditions, theorem 14.3 doesn't apply, but does it imply anything useful in practice? The only way to test this is to use existing puzzles and as of the writing of this section the largest relevant collection was mith's 972 one. So, the question will now be: in this 972 database, if not using any tridagon related resolution rule, but using eleven's replacement technique, can all the puzzles be solved in Z5 and how far must one go otherwise? The results are clear.

Only 78 (i.e. 8%) of the puzzles are not solved in Z5: 8 12 20 76 77 80 89 90 128 129 155 156 164 165 168 169 173 226 232 234 235 236 237 265 266 267 269 271 396 442 459 517 518 522 524 563 564 565 567 570 595 596 597 677 776 827 828 836 837 846 884 915 932 933 937 942 944 945 947 948 952 953 956 957 958 959 960 962 963 964 965 966 967 968 969 970 971 972

Of those 78:
- 67 are solved in W6: 20 76 77 80 89 90 129 155 164 165 168 169 173 226 232 234 235 236 237 266 267 269 271 442 518 522 524 563 564 565 567 570 595 596 597 677 776 837 846 884 915 932 933 937 942 944 945 947 948 952 953 956 957 958 959 960 962 963 964 965 966 967 968 969 970 971 972
- 2 are solved in W7: 156 265
- 8 are solved in W8: 8 12 128 459 517 827 828 836
- the last one is solved in W9: 396

Notice that nothing can guarantee that a different replacement wouldn't have produced a simpler puzzle.

The answer to the last question is therefore clear, at least with this 972 database: in most cases (more than 98.6%), whips of length ≤ 6 are enough.

14.7 General k-digit patterns

Shortly after I found that mith's Loki puzzle was not in T&E(2), I showed that the trivalue oddagon pattern can be proven contradictory in T&E(3). I have already given short indirect proofs in section 14.3 but the direct proof in T&E(3) takes an extremely long time. This section will show how a much shorter direct proof can be obtained in T&E(3) by using the restricted (incomplete) form of T&E described in section 3.6.1.

14.7.1 k-digit patterns

Let us define a k-digit pattern as a set of rc-cells that can only have candidates from the {1 2 … k} subset of digits. A k-digit pattern can be represented by a string of 81 characters, with "0" or "." for a cell not in the pattern and "1" or "X" for a cell in the pattern. As an example with k=3, the general resolution state corresponding to the trivalue oddagon pattern was introduced at the start of section 14.3.1:

```
+-------------------------------+-------------------------------+-------------------------------+
! 123       123456789 123456789 ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123       123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123       ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123       123456789 123456789 ! 123456789 123456789 123       ! 123456789 123456789 123456789 !
! 123456789 123       123456789 ! 123456789 123       123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123       ! 123       123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
! 123456789 123456789 123456789 ! 123456789 123456789 123456789 ! 123456789 123456789 123456789 !
+-------------------------------+-------------------------------+-------------------------------+
```

This can be represented as an 81-character string:

```
100100000010010000001001000100001000010010000001100000000000000000000000000000000
```

where "0" stand for a cell not in the pattern and "1" for a 123-cell.

This can also be represented as a 54-character string:

```
100100000010010000001001000100001000010010000001100000
```

where it is assumed that all the missing final characters are "0".

14.7.2 Proving in T&E(3) that the trivalue oddagon pattern is contradictory

Select only T&E(3) in the SudoRules configuration file – i.e. (bind ?*TE3* TRUE) – and type:

```
(solve-k-digit-pattern-string 3
  "100100000010010000001001000100001000010010000001100000000000000000000000000000000"
)
```

After a few combinations of 3 candidates (the whole taking a fraction of a second), the following message will appear:

GRID HAS NO SOLUTION : NO CANDIDATE FOR RC-CELL r3c3
meaning that the pattern is contradictory in restricted T&E(3). For details, you can find the full T&E resolution path leading to this conclusion in the “Sudoku/Tridagons/Proofs/Proofs-of-Tridagon-pattern.clp” file of [CSP-RULES-EXAMPLES].

14.7.3 SudoRules functions for analysing k-digit patterns

SudoRules has several general functions for dealing with k-digit patterns in general, with syntax given below:

```
(solve-k-digit-pattern-string ?k ?string)

(solve-n-grids-after-first-p-from-k-digit-pattern-string-file
?k ?file-name ?p ?n)

(solve-n-grids-after-first-p-from-k-digit-pattern-string-file-
excluding ?k ?file-name ?p ?n $?l-out)
```

where ?k is the number of digits in the pattern, ?file-name, ?p and ?n are as in similar functions and $?l-out is the list of puzzles that are excluded from consideration – possibly because they have already been dealt with by another set of rules/techniques.

14.7.4 Finding other contradictory 3-digit patterns

The trivalue oddagon pattern is far from being the only known contradictory 3-digit pattern. Eleven has produced a list of 380 of them involving 10 to 16 cells. It would have been impossible to manually analyse them all, but the above functions allowed to filter those that potentially require T&E(3) for proving they are contradictory. Indeed, only one of them could be found, the following 15-cell pattern:

```
+-------+-------+-------+
! . . . ! . . X ! . . X !
! . . X ! . . . ! . X . !
! . X . ! . X . ! X . . !
+-------+-------+-------+
! . . . ! . X . ! . . X !
! . . X ! . . X ! X . . !
! . X . ! X . . ! . X . !
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . . ! . . . ! . . . !
! . . . ! . . . ! . . . !
+-------+-------+-------+
```

Or :

```
000001001001000010010010100000010001001001100010100010000000000000000000000000000
```

Or, in “sauce pan with lid” representation:

```
+-------+-------+-------+
! . . . ! . . X ! . . X !
! . . . ! . X . ! . X . !
! . . . ! X . . ! X . . !
+-------+-------+-------+
! . . . ! . . X ! X . . !
! . . . ! . X . ! . . . !
! . . . ! X . . ! . . X !
+-------+-------+-------+
! . . . ! X . . ! . X . !
! . . . ! . X . ! X . . !
! . . . ! . . . ! . . . !
+-------+-------+-------+
```

The whole analysis can be found in the "Sudoku/eleven-impossible-3-digit-patterns" folder of [CSP-RULES-EXAMPLES]. Note that the proof that the pattern effectively requires T&E(3) still requires to try it (and fails to find a conclusion) in the full T&E(2). But now, we need to do it for only one puzzle instead of 380. As the restricted form is orders of magnitude faster, this is a clear advantage. It can also be interesting in and of itself to know that the full T&E(2) is required: this means that candidates other than those in the pattern must be considered in the proof.

As of the updating of this section, the original set of 380 puzzles has been extended to 630, among which 10 (instead one 1 previously) cannot be proven contradictory in restricted T&E(2) [but they can in restricted T&E(3)]:

```
000001001000010010000100100000000100100001010000010001000000000000000000000000000 #38
12-cells
000000000001001001001010010000000101000001000100110010000000000000000000000000000 #171
13-cells
000000000001001001001010010001000000010001010010010001000000000000000000000000000 #180
13-cells
000000000001001001001010010001000000010001010100010001000000000000000000000000000 #181
13-cells
000000001000011010001100100001000001010100001100100010000000000000000000000000000 #56
15-cells
000000001001001010010010100001001000010100001100010001000000000000000000000000000 #66
15-cells
000001001001000010010010100000001000100100110001010001000000000000000000000000000 #97
15-cells (the previously mentioned one))
000001001001010010001010010001000001010001100010001100000000000000000000000000000 #5
16-cells
000001001001010010001010010001000001010001100100001100000000000000000000000000000 #6
16-cells
000001001001010010001010100001001000010100001100100001000000000000000000000000000 #9
16-cells
```

The question arises which among them cannot be proven contradictory in the full T&E(2). There is now a function for launching the full test in T&E(2) – or in whatever set of rules or techniques is selected in the configuration file; the only difference is, no focusing is applied:

```
(fully-solve-k-digit-pattern-string ?k ?string)
```

The result is only the first pattern cannot be proven contradictory in full T&E(2). In the following representation, it is one the isomorphic forms of the trivalue oddagon pattern.

```
+-------+-------+-------+
! . . . ! . . 1 ! . . 1 !
! . . . ! . 1 . ! . 1 . !
! . . . ! 1 . . ! 1 . . !
+-------+-------+-------+
! . . . ! . . 1 ! . . 1 !
! . . . ! 1 . . ! . 1 . !
! . . . ! . 1 . ! 1 . . !
+-------+-------+-------+
! . . . ! . . . ! . . . !
! . . . ! . . . ! . . . !
! . . . ! . . . ! . . . !
+-------+-------+-------+
```

Finally, remember that our goal here was to find patterns that would *potentially* require T&E(3) to prove their contradiction. If our goal was only to prove that they are contradictory, one should use the much faster DFS instead of T&E(3). DFS had been updated to work with the above focused functions. However, DFS will be of no use for suggesting a smart proof of the contradiction.

14.7.5 SudoRules functions for analysing kl-digit patterns

It seemed useful in some cases to have a generalisation of the previous functions. Given two digits k and l, a kl-digit pattern is defined by a sequence of 81 characters, all k, l or ".", with the intended meaning that cells with symbol k must have candidates 1,…k, cells with symbol l must have candidate 1,…l, and cells with a dot are unrestricted. SudoRules has a function with syntax given below:

```
(solve-kl-digit-pattern-string ?k ?l ?string)
```

Example of use:
(***solve-kl-digit-pattern-string*** 2 3 ".3...3...3..3.......2.2....3....3.......2.....3.3.....")

14.8 The anti-tridagon pattern and mith's 63,137 puzzles in T&E(3)

We are now starting the more general approach hinted to at the beginning of this chapter.

While I use the expression "trivalue oddagon pattern" for (and only for) the precise contradictory pattern defined in section 14.1 (a pattern that can appear in no well-formed puzzle), let me call anti-tridagon pattern the following very broad set of conditions (which, in and of itself, allows no assertion or elimination, but only a big OR conclusion). It is defined to cover all the cases of any number of additional candidates in any number of cells of the "trivalue oddagon pattern". (It doesn't cover any case of "degenerated anti-tridagon", i.e. of any missing 123-candidate(s) in any of the 123-cells – but I haven't seen any such case as yet and I have shown that such cases are degenerated when there are no guardians.)

This pattern covers the basic tridagon pattern as well as the patterns at the start of the Tridagon-forcing-whips. But it covers many more cases and it will allow to define Tridagon-OR_k-relations and to use generic OR_k-chains and based on them: Tridagon-OR_k-forcing-whips, Tridagon-OR_k-contrad-whips, Tridagon-OR_k-whips…

14.8.1 Definition of the general anti-tridagon pattern

Definition of the anti-tridagon pattern: let there be four blocks forming a rectangle in two bands and two stacks:

b11 b12

b21 b22

Let there be three digits, say 1 2 3, such that: in each of the four blocks, there are three cells in different rows and different columns such that:

– the same additional conditions on the 4x3 cells as in section 14.1.2 are satisfied;

– each of these 4×3 cells contains the three digits (notice the only difference with the contradictory trivalue oddagon pattern of section 14.1.1: each cell may contain any number of additional candidates);

– at least one of the 4×3 cells contains at least one additional digit (anti-contradiction condition, which is automatically satisfied in a consistent puzzle).

Additional candidates in the 4×3 cells are called "***guardians***" because they guard against the contradiction inherent in the trivalue oddagon pattern.

Essential remark (due to mith): in any such case, none of the three digits can be a given (or a decided value) in the two bands and two stacks of the four blocks. *As a result, at least two of the three digits must be a given or a decided value in the uniquely defined block that is neither in the two stacks nor in the two bands*. This defines an additional criterion (first mentioned by mith) that drastically simplifies the detection of the pattern (be it by a manual solver or a computer program). This may also justify lowering the complexity assigned to the detection of the anti-tridagon pattern, if one considers that this additional criterion is indeed the main detection one and that the other conditions are mere checking.

14.8.2 Mith's database of 63,137 T&E(3) min-expand puzzles

As of the writing of this section, the largest available collection of T&E(3) min-expand puzzles is mith's extension of the above-mentioned 972 one. It now has 63,137 puzzles. Let's state three basic results about it:

⊗ First result: all the T&E(3) puzzles in mith's collection are indeed in T&E(W2, 2), i.e. at the low end of T&E(3). ***T&E(W2, 2) thus seems to be the new frontier for the hardest puzzles***. Note: this result can be extended to the more recent collection of 158,276 min-expands and corresponding collection of 845,781 minimal puzzles.

⊗ Second result: ***each of the 63,137 min-expand puzzles in mith's database has at least one anti-tridagon pattern*** (and obviously at most one possibility for the broader pattern of blocks and digits). Thanks to mith for pointing out a rare example where there are two possible combinations of cells, namely puzzle "Tanngrisnir and Tanngnjóstr" (SER 11.7, see section 14.9.4 below):

```
........1....23.4..56...237....82.....53.6....8.45.....43...6.226..348.55.826....
```

⊗ ***Third result***: ***classification of the number of guardians*** when Subsets, Finned-Fish, whips[≤ 7], Tridagons and Tridagon-forcing-whips(≤ 22) are active:

```
1:  33,579 (including 29,356 solved in SFin+Trid+W7)
2:  19,016 (including 9089 solved in SFin+Trid+W7+TFW22 but not in
SFin+Trid+W7)
3:  7,443
4:  2,201
5:  647
6:  191
7:  42
8:  7
9:  7
10: 2
11: 1
12: 1
Total: 63,137
```

The above results will be better appreciated if compared to the results after allowing only Subsets and Finned Fish:

```
1:  23,279
2:  16,830
3:  10,851
4:  6,165
5:  3,254
6:  1,718
7:  562
8:  233
9:  121
10: 59
11: 40
```

```
12: 20
13: 1
14: 0
15: 4
Total: 63,137
```

The above results justify trying to use OR_k-chains based on a OR_k-anti-tridagon relation for $k > 2$. They also suggest more precisely that, in rare cases, OR_k-chains with $k > 5$ may be useful.

14.9 Tridagon-OR_k-forcing-whips

When an anti-tridagon pattern with k guardians C_1, C_2, … C_k has been found in a puzzle, one can conclude that there is an OR_k relation between them: $OR_k(C_1, C_2, \dots C_k)$. In turn, this allows to use the various types of generic OR_k-chains defined in section 3.5. This can easily be done in SudoRules, where several sections of the configuration file have been added for this purpose (see section 14.18 below for full details).

It is important to understand that:

1) originally, the anti-tridagon rule that detects the anti-tridagons and asserts the OR_k relations was defined at its proper level of complexity (12, because it involves 12 CSP-Variables), after all the non-forcing generic rules at level 12 but before all the forcing-chains at this level. This choice entails that all the non-OR_k chains of length 12 or less will be tried before this rule, if they are activated in the configuration file. The reason was, it allows to give the other rules a chance to ***minimise the number of guardians*** (too many guardians may make it hard to find common eliminations). It pertains to the user to choose how much chance he wants to let to the "normal" rules. For a more recent and smarter behaviour (allowed in part by the introduction of ultra-persistency rules for the Tridagon-OR_k relations and now set to always be used by default), see section 14.18.

2) however, once an OR_k relation has been found, its use as a starting point for OR_k-forcing-whips is not counted as 12, but only as 1 (as is the bivalue relation in standard forcing-whips), which provides a good balance between the standard chains and those exotic OR_k-forcing-whips. This is the standard way for OR_k-forcing-whips, as defined in section 3.5, which is justified by their intended use in conjunction with exotic patterns only. The same remarks apply to OR_k-contrad-whips and OR_k-whips to appear in the next sections, and to the corresponding g-chains.

14.9.2 An example with OR_4-forcing-whips

Consider puzzle #339 in mith's 63,137 min-expand collection, a puzzle that still has 4 guardians, even after applying W7 to it:

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

```
...4.678.......2...8.27....2.834...737..6.....468.7....6....194.34...52......4.73;111;
44367
```

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

```
+----------------------+----------------------+----------------------+
! 159    1259   12359  ! 4      1359   6      ! 7      8      159    !
! 145679 159    13579  ! 159    13589  13589  ! 2      13456  1569   !
! 14569  8      1359   ! 2      7      1359   ! 3469   13456  1569   !
+----------------------+----------------------+----------------------+
! 2      159    8      ! 3      4      159    ! 69     156    7      !
! 3      7      159    ! 159    6      1259   ! 489    145    12589  !
! 159    4      6      ! 8      1259   7      ! 39     135    1259   !
+----------------------+----------------------+----------------------+
! 578    6      257    ! 57     2358   2358   ! 1      9      4      !
! 1789   3      4      ! 1679   189    189    ! 5      2      68     !
! 1589   1259   1259   ! 1569   12589  4      ! 68     7      3      !
+----------------------+----------------------+----------------------+
184 candidates
```

```
hidden-pairs-in-a-column: c1{n4 n6}{r2 r3} ==> r3c1≠9, r3c1≠5, r3c1≠1, r2c1≠9, r2c1≠7,
r2c1≠5, r2c1≠1
hidden-single-in-a-block ==> r2c3=7
biv-chain[3]: r4c7{n9 n6} - b9n6{r9c7 r8c9} - c9n8{r8 r5} ==> r5c9≠9
z-chain[3]: c4n6{r9 r8} - r8n7{c4 c1} - r8n1{c1 .} ==> r9c4≠1
z-chain[3]: c4n6{r9 r8} - r8n7{c4 c1} - r8n9{c1 .} ==> r9c4≠9
biv-chain[4]: r4c7{n9 n6} - b9n6{r9c7 r8c9} - c9n8{r8 r5} - b6n2{r5c9 r6c9} ==> r6c9≠9
whip[1]: c9n9{r3 .} ==> r3c7≠9
biv-chain[4]: r8c9{n6 n8} - b6n8{r5c9 r5c7} - c7n4{r5 r3} - r3c1{n4 n6} ==> r3c9≠6
z-chain[5]: c1n7{r8 r7} - c1n8{r7 r9} - b9n8{r9c7 r8c9} - r8n6{c9 c4} - r8n7{c4 .} ==>
r8c1≠1, r8c1≠9
whip[1]: r8n9{c6 .} ==> r9c5≠9
whip[1]: r8n1{c6 .} ==> r9c5≠1
```

```
+-------------------+-------------------+-------------------+
! 159    1259   12359 ! 4      1359   6      ! 7      8      159    !
! 46     159    7      ! 159    13589 13589 ! 2      13456 1569   !
! 46     8      1359   ! 2      7      1359   ! 346    13456 159    !
+-------------------+-------------------+-------------------+
```

```
! 2     159   8     ! 3     4     159   ! 69    156   7     !
! 3     7     159   ! 159   6     1259  ! 489   145   1258  !
! 159   4     6     ! 8     1259  7     ! 39    135   125   !
+-------------------+-------------------+-------------------+
! 578   6     25    ! 57    2358  2358  ! 1     9     4     !
! 78    3     4     ! 1679  189   189   ! 5     2     68    !
! 1589  1259  1259  ! 56    258   4     ! 68    7     3     !
+-------------------+-------------------+-------------------+
```

```
OR4-anti-tridagon[12] for digits 1, 5 and 9 in blocks:
        b1, with cells: r1c1, r2c2, r3c3
        b2, with cells: r1c5, r2c4, r3c6
        b4, with cells: r6c1, r4c2, r5c3
        b5, with cells: r6c5, r4c6, r5c4
with 4 guardians: n3r1c5 n3r3c3 n3r3c6 n2r6c5
```

Based on this OR4 relation, four different OR4-forcing-whips will allow an easy solution.

```
Trid-OR4-forcing-whip-elim[4]:
   || n3r3c6 -
   || n3r1c5 -
   || n3r3c3 - partial-whip[1]: b3n3{r3c8 r2c8} -
   || n2r6c5 - partial-whip[2]: c6n2{r5 r7} - r7n3{c6 c5} -
 ==> r2c5≠3

t-whip[5]: c6n8{r8 r2} - r2n3{c6 c8} - r2n4{c8 c1} - r2n6{c1 c9} - r8c9{n6 .} ==>
r8c5≠8

Trid-OR4-forcing-whip-elim[5]:
   || n3r3c6 -
   || n3r3c3 -
   || n3r1c5 - partial-whip[1]: c3n3{r1 r3} -
   || n2r6c5 - partial-whip[3]: r5n2{c6 c9} - r5n8{c9 c7} - c7n4{r5 r3} -
 ==> r3c7≠3

hidden-single-in-a-column ==> r6c7=3
naked-pairs-in-a-row: r3{c1 c7}{n4 n6} ==> r3c8≠6, r3c8≠4
z-chain[5]: c6n2{r5 r7} - r7c3{n2 n5} - r5c3{n5 n9} - r6n9{c1 c5} - b5n2{r6c5 .} ==>
r5c6≠1
t-whip[6]: c5n3{r7 r1} - r2n3{c6 c8} - r2n4{c8 c1} - r2n6{c1 c9} - r8n6{c9 c4} -
r9c4{n6 .} ==> r7c5≠5

Trid-OR4-forcing-whip-elim[7]:
   || n3r3c6 - partial-whip[1]: r2n3{c6 c8} -
   || n3r3c3 - partial-whip[1]: b3n3{r3c8 r2c8} -
   || n3r1c5 - partial-whip[1]: r2n3{c6 c8} -
   || n2r6c5 - partial-whip[3]: r5n2{c6 c9} - c9n8{r5 r8} - c9n6{r8 r2} -
 ==> r2c8≠6

hidden-single-in-a-column ==> r4c8=6
naked-single ==> r4c7=9
```

```
z-chain[7]: c6n2{r5 r7} - r7c3{n2 n5} - r5c3{n5 n1} - r4n1{c2 c6} - r8c6{n1 n8} -
c9n8{r8 r5} - r5n2{c9 .} ==> r5c6≠9
whip[5]: r8c5{n1 n9} - b5n9{r6c5 r5c4} - c4n1{r5 r8} - r8c6{n1 n8} - r2n8{c6 .} ==>
r2c5≠1
whip[7]: b3n4{r2c8 r3c7} - c7n6{r3 r9} - c4n6{r9 r8} - c4n1{r8 r5} - b6n1{r5c8 r6c9} -
r6n2{c9 c5} - b5n9{r6c5 .} ==> r2c8≠1
whip[7]: b5n9{r6c5 r5c4} - r8n9{c4 c6} - r8c5{n9 n1} - c4n1{r8 r2} - r2c2{n1 n5} -
r4n5{c2 c6}- c6n1{r4 .} ==> r2c5≠9

Trid-OR4-forcing-whip-elim[7]:
   || n3r3c6 - partial-whip[1]: r2n3{c6 c8} -
   || n3r3c3 - partial-whip[1]: b3n3{r3c8 r2c8} -
   || n3r1c5 - partial-whip[1]: r2n3{c6 c8} -
   || n2r6c5 - partial-whip[3]: r5n2{c6 c9} - r5n8{c9 c7} - c7n4{r5 r3} -
 ==> r2c8≠4
```

The end is easy:

```
singles ==> r3c7=4, r3c1=6, r2c1=4, r5c7=8, r9c7=6, r8c9=8, r8c1=7, r9c4=5, r7c4=7,
r2c9=6, r8c4=6, r5c8=4
finned-x-wing-in-rows: n1{r4 r2}{c2 c6} ==> r3c6≠1
biv-chain[2]: r4n1{c2 c6} - c4n1{r5 r2} ==> r2c2≠1
whip[1]: r2n1{c6 .} ==> r1c5≠1
biv-chain[3]: r4n1{c2 c6} - r5c4{n1 n9} - b4n9{r5c3 r6c1} ==> r6c1≠1
biv-chain[4]: c2n2{r1 r9} - r9c5{n2 n8} - r2c5{n8 n5} - r2c2{n5 n9} ==> r1c2≠9
biv-chain[3]: c2n9{r9 r2} - c4n9{r2 r5} - b4n9{r5c3 r6c1} ==> r9c1≠9
biv-chain[4]: b7n9{r9c3 r9c2} - r2c2{n9 n5} - r2c5{n5 n8} - r9c5{n8 n2} ==> r9c3≠2
biv-chain[3]: c1n1{r1 r9} - r9c3{n1 n9} - b4n9{r5c3 r6c1} ==> r1c1≠9
singles ==> r6c1=9, r5c4=9, r2c4=1
finned-x-wing-in-rows: n1{r5 r3}{c3 c9} ==> r1c9≠1
whip[1]: b3n1{r3c9 .} ==> r3c3≠1
biv-chain[3]: r5c3{n1 n5} - r7n5{c3 c1} - r1c1{n5 n1} ==> r1c3≠1
biv-chain[3]: c1n5{r1 r7} - r7c3{n5 n2} - b1n2{r1c3 r1c2} ==> r1c2≠5
biv-chain[3]: c3n2{r7 r1} - r1n3{c3 c5} - b8n3{r7c5 r7c6} ==> r7c6≠2
singles ==> r5c6=2, r6c9=2
finned-x-wing-in-columns: n5{c2 c6}{r4 r2} ==> r2c5≠5
singles ==> r2c5=8, r9c5=2, r7c5=3, r7c6=8, r7c1=5, r1c1=1, r1c2=2, r9c1=8, r7c3=2,
r1c3=3
finned-x-wing-in-columns: n5{c3 c9}{r5 r3} ==> r3c8≠5
finned-x-wing-in-columns: n5{c8 c5}{r6 r2} ==> r2c6≠5
finned-x-wing-in-columns: n5{c6 c3}{r3 r4} ==> r4c2≠5
stte
```

Notice that currently OR_k-forcing-whips don't have a "blocked" version and they may be interrupted by simpler rules; but this is quite irrelevant to the purposes of this chapter.

14.9.3 An example with OR_5-forcing-whips

Consider puzzle #47008 in mith's 63,137 min-expand collection, a puzzle that still has 5 guardians, even after applying W7 to it:

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

1.345..894.....1.3.9813......9....35..4..581.........4...5.13.88...6....93.278...;98

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

```
+----------------------+----------------------+----------------------+
! 1      267    3      ! 4      5      267    ! 267    8      9      !
! 4      2567   2567   ! 6789   289    2679   ! 1      2567   3      !
! 2567   9      8      ! 1      3      267    ! 24567  24567  267    !
+----------------------+----------------------+----------------------+
! 267    12678  9      ! 678    1248   2467   ! 267    3      5      !
! 2367   267    4      ! 3679   29     5      ! 8      1      267    !
! 23567  125678 12567  ! 3678   128    2367   ! 2679   2679   4      !
+----------------------+----------------------+----------------------+
! 267    2467   267    ! 5      49     1      ! 3      2679   8      !
! 8      12457  1257   ! 39     6      349    ! 2579   2579   127    !
! 9      3      156    ! 2      7      8      ! 456    456    16     !
+----------------------+----------------------+----------------------+
177 candidates.
```

biv-chain[3]: r5c5{n2 n9} - r7c5{n9 n4} - b5n4{r4c5 r4c6} ==> r4c6≠2
biv-chain[3]: r4n1{c2 c5} - b5n4{r4c5 r4c6} - r8n4{c6 c2} ==> r8c2≠1
whip[1]: b7n1{r9c3 .} ==> r6c3≠1
hidden-pairs-in-a-block: b4{n1 n8}{r4c2 r6c2} ==> r6c2≠7, r6c2≠6, r6c2≠5, r6c2≠2, r4c2≠7, r4c2≠6, r4c2≠2
biv-chain[4]: c1n5{r6 r3} - c2n5{r2 r8} - r8n4{c2 c6} - c6n3{r8 r6} ==> r6c1≠3
hidden-single-in-a-block ==> r5c1=3
z-chain[5]: c7n4{r3 r9} - c7n5{r9 r8} - c2n5{r8 r2} - r3n5{c1 c8} - r3n4{c8 .} ==> r3c7≠2, r3c7≠7, r3c7≠6

```
+-------------------+-------------------+-------------------+
! 1      267   3    ! 4      5     267  ! 267    8     9    !
! 4      2567  2567 ! 6789   289   2679 ! 1      2567  3    !
! 2567   9     8    ! 1      3     267  ! 45     24567 267  !
+-------------------+-------------------+-------------------+
! 267    18    9    ! 678    1248  467  ! 267    3     5    !
! 3      267   4    ! 679    29    5    ! 8      1     267  !
! 2567   18    2567 ! 3678   128   2367 ! 2679   2679  4    !
+-------------------+-------------------+-------------------+
```

```
! 267   2467  267   ! 5     49    1     ! 3     2679  8     !
! 8     2457  1257  ! 39    6     349   ! 2579  2579  127   !
! 9     3     156   ! 2     7     8     ! 456   456   16    !
+-------------------+-------------------+-------------------+
```

```
OR5-anti-tridagon[12] for digits 2, 6 and 7 in blocks:
        b1, with cells: r1c2, r2c3, r3c1
        b3, with cells: r1c7, r2c8, r3c9
        b4, with cells: r5c2, r6c3, r4c1
        b6, with cells: r5c9, r6c8, r4c7
with 5 guardians: n5r2c3 n5r2c8 n5r3c1 n5r6c3 n9r6c8

Trid-OR5-forcing-whip-elim[7]:
   || n5r2c3 - partial-whip[1]: b7n5{r9c3 r8c2} -
   || n5r3c1 - partial-whip[1]: c2n5{r2 r8} -
   || n5r6c3 - partial-whip[1]: b7n5{r9c3 r8c2} -
   || n9r6c8 - partial-whip[1]: c7n9{r6 r8} -
   || n5r2c8 - partial-whip[2]: c2n5{r2 r8} - r9n5{c3 c7} -
 ==> r8c7≠5

hidden-pairs-in-a-column: c7{n4 n5}{r3 r9} ==> r9c7≠6

Trid-OR5-forcing-whip-elim[7]:
   || n5r2c3 - partial-whip[1]: b7n5{r9c3 r8c2} -
   || n5r2c8 - partial-whip[1]: c2n5{r2 r8} -
   || n5r3c1 - partial-whip[1]: c2n5{r2 r8} -
   || n5r6c3 - partial-whip[1]: b7n5{r9c3 r8c2} -
   || n9r6c8 - partial-whip[2]: r7n9{c8 c5} - b8n4{r7c5 r8c6} -
 ==> r8c2≠4
```

After these two OR_5 eliminations, the end is quite easy (for a puzzle in T&E(3):

```
singles ==> r7c2=4, r7c5=9, r5c5=2, r2c5=8, r6c5=1, r4c5=4, r6c2=8, r4c2=1, r8c4=3,
r8c6=4, r6c6=3, r4c4=8, r5c4=9, r2c6=9
finned-x-wing-in-columns: n2{c9 c2}{r8 r3} ==> r3c1≠2
t-whip[2]: r5n6{c2 c9} - c7n6{r6 .} ==> r1c2≠6
finned-x-wing-in-columns: n6{c4 c2}{r2 r6} ==> r6c3≠6, r6c1≠6
biv-chain[2]: r1n6{c7 c6} - b5n6{r4c6 r6c4} ==> r6c7≠6
biv-chain[3]: r4c6{n7 n6} - r6n6{c4 c8} - r5c9{n6 n7} ==> r4c7≠7
z-chain[3]: r5n7{c9 c2} - r1n7{c2 c6} - b5n7{r4c6 .} ==> r6c7≠7
biv-chain[3]: c7n7{r8 r1} - c7n6{r1 r4} - r5c9{n6 n7} ==> r8c9≠7
z-chain[3]: c9n7{r5 r3} - r1n7{c7 c6} - r4n7{c6 .} ==> r5c2≠7
singles ==> r5c2=6, r5c9=7
x-wing-in-rows: n6{r1 r4}{c6 c7} ==> r3c6≠6
finned-x-wing-in-columns: n6{c1 c9}{r3 r7} ==> r7c8≠6
whip[1]: b9n6{r9c9 .} ==> r9c3≠6
biv-chain[3]: b2n2{r1c6 r3c6} - r3c9{n2 n6} - r1n6{c7 c6} ==> r1c6≠7
biv-chain[3]: b2n7{r2c4 r3c6} - c6n2{r3 r1} - r1c2{n2 n7} ==> r2c2≠7, r2c3≠7
x-wing-in-columns: n7{c2 c7}{r1 r8} ==> r8c8≠7, r8c3≠7
biv-chain[3]: r3c6{n2 n7} - r2n7{c4 c8} - r7c8{n7 n2} ==> r3c8≠2
biv-chain[3]: r2n7{c8 c4} - r3c6{n7 n2} - r3c9{n2 n6} ==> r2c8≠6
```

```
biv-chain[3]: r2n6{c3 c4} - r6c4{n6 n7} - c3n7{r6 r7} ==> r7c3≠6
stte
```

14.9.4 An example with two different combinations of tridagon cells

Puzzle "Tanngrisnir and Tanngnjóstr" (SER 11.7) was mentioned in section 14.8.2 as a rare one (found by mith) with two different anti-tridagon patterns in the same four blocks for the same three digits:

```
+-------+-------+-------+
! . . . ! . . . ! . . 1 !
! . . . ! . 2 3 ! . 4 . !
! . 5 6 ! . . . ! 2 3 7 !
+-------+-------+-------+
! . . . ! . 8 2 ! . . . !
! . . 5 ! 3 . 6 ! . . . !
! . 8 . ! 4 5 . ! . . . !
+-------+-------+-------+
! . 4 3 ! . . . ! 6 . 2 !
! 2 6 . ! . 3 4 ! 8 . 5 !
! 5 . 8 ! 2 6 . ! . . . !
+-------+-------+-------+
........1....23.4..56...237....82.....53.6....8.45.....43...6.226..348.55.826....
```

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

```
+----------------------+----------------------+----------------------+
! 34789  2379   2479   ! 56789  479    5789   ! 59     5689   1      !
! 1789   179    179    ! 156789 2      3      ! 59     4      689    !
! 1489   5      6      ! 189    149    189    ! 2      3      7      !
+----------------------+----------------------+----------------------+
! 134679 1379   1479   ! 179    8      2      ! 134579 15679  3469   !
! 1479   1279   5      ! 3      179    6      ! 1479   12789  489    !
! 13679  8      1279   ! 4      5      179    ! 1379   12679  369    !
+----------------------+----------------------+----------------------+
! 179    4      3      ! 15789  179    15789  ! 6      179    2      !
! 2      6      179    ! 179    3      4      ! 8      179    5      !
! 5      179    8      ! 2      6      179    ! 13479  179    349    !
+----------------------+----------------------+----------------------+
189 candidates.
```

```
naked-pairs-in-a-column: c7{r1 r2}{n5 n9} ==> r9c7≠9, r6c7≠9, r5c7≠9, r4c7≠9, r4c7≠5
hidden-single-in-a-block ==> r4c8=5
whip[1]: c7n9{r2 .} ==> r1c8≠9, r2c9≠9
hidden-pairs-in-a-row: r9{n3 n4}{c7 c9} ==> r9c9≠9, r9c7≠7, r9c7≠1
whip[1]: b9n1{r9c8 .} ==> r5c8≠1, r6c8≠1
whip[1]: b9n7{r9c8 .} ==> r5c8≠7, r6c8≠7
```

```
whip[1]: c9n9{r6 .} ==> r5c8≠9, r6c8≠9
hidden-pairs-in-a-row: r7{n5 n8}{c4 c6} ==> r7c6≠9, r7c6≠7, r7c6≠1, r7c4≠9, r7c4≠7,
r7c4≠1
biv-chain[3]: r1c7{n9 n5} - r2n5{c7 c4} - b2n6{r2c4 r1c4} ==> r1c4≠9
biv-chain[4]: c3n4{r4 r1} - c3n2{r1 r6} - r6c8{n2 n6} - b4n6{r6c1 r4c1} ==> r4c1≠4
z-chain[5]: c2n2{r1 r5} - r5c8{n2 n8} - r1c8{n8 n6} - r2n6{c9 c4} - r2n7{c4 .} ==>
r1c2≠7
biv-chain[6]: r1c7{n9 n5} - r2n5{c7 c4} - b2n6{r2c4 r1c4} - c8n6{r1 r6} -
b6n2{r6c8 r5c8} - c2n2{r5 r1} ==> r1c2≠9
t-whip[5]: c1n6{r4 r6} - r6c8{n6 n2} - r5n2{c8 c2} - r1c2{n2 n3} - b4n3{r4c2 .} ==>
r4c1≠1, r4c1≠7, r4c1≠9
z-chain[6]: r1n3{c1 c2} - c2n2{r1 r5} - r5c8{n2 n8} - r1c8{n8 n6} - r2n6{c9 c4} -
r2n7{c4 .} ==> r1c1≠7
biv-chain[7]: r1c7{n9 n5} - r2n5{c7 c4} - b2n6{r2c4 r1c4} - c8n6{r1 r6} -
b6n2{r6c8 r5c8} - c2n2{r5 r1} - b1n3{r1c2 r1c1} ==> r1c1≠9
```

```
+----------------------+----------------------+----------------------+
! 348    23     2479   ! 5678   479    5789   ! 59     68     1      !
! 1789   179    179    ! 156789 2      3      ! 59     4      68     !
! 1489   5      6      ! 189    149    189    ! 2      3      7      !
+----------------------+----------------------+----------------------+
! 36     1379   1479   ! 179    8      2      ! 1347   5      3469   !
! 1479   1279   5      ! 3      179    6      ! 147    28     489    !
! 13679  8      1279   ! 4      5      179    ! 137    26     369    !
+----------------------+----------------------+----------------------+
! 179    4      3      ! 58     179    58     ! 6      179    2      !
! 2      6      179    ! 179    3      4      ! 8      179    5      !
! 5      179    8      ! 2      6      179    ! 34     179    34     !
+----------------------+----------------------+----------------------+
```

First anti-tridagon pattern, with 4 guardians, and associated OR_4-forcing-whip elimination:

```
OR4-anti-tridagon[12] for digits 1, 7 and 9 in blocks:
        b4, with cells: r4c3, r5c2, r6c1
        b5, with cells: r4c4, r5c5, r6c6
        b7, with cells: r8c3, r9c2, r7c1
        b8, with cells: r8c4, r9c6, r7c5
with 4 guardians: n4r4c3 n2r5c2 n3r6c1 n6r6c1

Trid-forcing-whip-elim[6]:
   || n4r4c3 -
   || n2r5c2 - partial-whip[1]: c3n2{r6 r1} -
   || n3r6c1 - partial-whip[2]: c2n3{r4 r1} - r1n2{c2 c3} -
   || n6r6c1 - partial-whip[2]: r6c8{n6 n2} - c3n2{r6 r1} -
 ==> r1c3≠4

hidden-single-in-a-column ==> r4c3=4
z-chain[6]: c9n8{r2 r5} - r5c8{n8 n2} - c2n2{r5 r1} - r1n3{c2 c1} - c1n4{r1 r3} -
r3n8{c1 .} ==> r2c4≠8
t-whip[6]: c1n4{r3 r1} - r1n3{c1 c2} - c2n2{r1 r5} - r5c8{n2 n8} - c9n8{r5 r2} -
c1n8{r2 .} ==> r3c1≠9, r3c1≠1
```

```
whip[1]: r3n1{c6 .} ==> r2c4≠1
whip[1]: r3n9{c6 .} ==> r1c5≠9, r1c6≠9, r2c4≠9
whip[7]: r5n2{c2 c8} - r5n8{c8 c9} - r2n8{c9 c1} - r3c1{n8 n4} - c5n4{r3 r1} -
c5n7{r1 r7} - c1n7{r7 .} ==> r5c2≠7
```

```
+-------------------+-------------------+-------------------+
! 348   23    279   ! 5678  47    578   ! 59    68    1     !
! 1789  179   179   ! 567   2     3     ! 59    4     68    !
! 48    5     6     ! 189   149   189   ! 2     3     7     !
+-------------------+-------------------+-------------------+
! 36    1379  4     ! 179   8     2     ! 137   5     369   !
! 179   129   5     ! 3     179   6     ! 147   28    489   !
! 13679 8     1279  ! 4     5     179   ! 137   26    369   !
+-------------------+-------------------+-------------------+
! 179   4     3     ! 58    179   58    ! 6     179   2     !
! 2     6     179   ! 179   3     4     ! 8     179   5     !
! 5     179   8     ! 2     6     179   ! 34    179   34    !
+-------------------+-------------------+-------------------+
```

Second anti-tridagon pattern on different cells, with 2 guardians, and associated OR_2-forcing-whip elimination:

```
OR2-anti-tridagon[12] for digits 1, 7 and 9 in blocks:
        b4, with cells: r4c2, r5c1, r6c3
        b5, with cells: r4c4, r5c5, r6c6
        b7, with cells: r9c2, r7c1, r8c3
        b8, with cells: r9c6, r7c5, r8c4
with 2 guardians: n3r4c2 n2r6c3

Trid-OR2-forcing-whip-elim[2]:
   || n2r6c3 -
   || n3r4c2 - partial-whip[1]: r1c2{n3 n2} -
 ==> r5c2≠2

singles ==> r6c3=2, r6c8=6, r1c8=8, r2c9=6, r5c8=2, r1c4=6, r5c9=8, r5c7=4,r9c7=3,
r9c9=4, r2c1=8, r3c1=4, r1c1=3, r1c2=2, r4c1=6, r4c2=3, r4c9=9, r6c9=3,r1c5=4
whip[1]: b4n7{r6c1 .} ==> r7c1≠7
biv-chain[2]: c2n7{r9 r2} - b2n7{r2c4 r1c6} ==> r9c6≠7
biv-chain[3]: c5n7{r7 r5} - b5n9{r5c5 r6c6} - r9c6{n9 n1} ==> r7c5≠1
biv-chain[3]: r5n7{c1 c5} - r7c5{n7 n9} - r7c1{n9 n1} ==> r5c1≠1
biv-chain[3]: r5n1{c2 c5} - b5n9{r5c5 r6c6} - r9c6{n9 n1} ==> r9c2≠1
biv-chain[3]: b8n1{r9c6 r8c4} - c4n9{r8 r3} - b2n8{r3c4 r3c6} ==> r3c6≠1
t-whip[4]: r1c3{n9 n7} - r2n7{c3 c4} - r4c4{n7 n1} - r8c4{n1 .} ==> r8c3≠9
whip[1]: c3n9{r2 .} ==> r2c2≠9
biv-chain[2]: b7n9{r9c2 r7c1} - r6n9{c1 c6} ==> r9c6≠9
naked-single ==> r9c6=1
biv-chain[3]: r2c2{n7 n1} - r5n1{c2 c5} - r4c4{n1 n7} ==> r2c4≠7
singles ==> r2c4=5, r1c6=7, r1c3=9, r1c7=5, r6c6=9, r3c6=8, r7c6=5, r2c7=9, r7c4=8
biv-chain[3]: c1n9{r7 r5} - r5n7{c1 c5} - r7c5{n7 n9} ==> r7c8≠9
biv-chain[3]: b9n1{r7c8 r8c8} - c8n9{r8 r9} - b7n9{r9c2 r7c1} ==> r7c1≠1
stte
```

14.9.5 Semi-manual use of the anti-tridagon pattern

Whereas the above examples show how to automatically combine the detection of the anti-tridagon pattern with its use in OR_k-forcing-whips, one may prefer to apply a more manual approach. Let's illustrate it with a new example, puzzle #62886 in mith's 63137 min-expands list.

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

```
...4..78.4..1....3......1.4.413.78.5..5.18.......4..173.4...5.8.968.....8.273....;1402
2;367772
SER = 10.1
```

Suppose, in a first instance of CSP-Rules, we activate only Subsets, Finned-Fish and Anti-Tridagons. We'll get the following:

```
Resolution state RS1 after Singles and whips[1]:

+---------------------+---------------------+---------------------+
! 12569  12356  39     ! 4      2569   23569  ! 7      8      269    !
! 4      2568   789    ! 1      256789 2569   ! 269    2569   3      !
! 2569   23568  3789   ! 2569   256789 23569  ! 1      2569   4      !
+---------------------+---------------------+---------------------+
! 269    4      1      ! 3      269    7      ! 8      269    5      !
! 2679   2367   5      ! 269    1      8      ! 23469  23469  269    !
! 269    2368   389    ! 2569   4      2569   ! 2369   1      7      !
+---------------------+---------------------+---------------------+
! 3      17     4      ! 269    269    1269   ! 5      2679   8      !
! 157    9      6      ! 8      25     1245   ! 234    2347   12     !
! 8      15     2      ! 7      3      14569  ! 469    469    169    !
+---------------------+---------------------+---------------------+
185 candidates.

hidden-pairs-in-a-column: c5{n7 n8}{r2 r3} ==> r3c5≠9, r3c5≠6, r3c5≠5, r3c5≠2, r2c5≠9,
r2c5≠6, r2c5≠5, r2c5≠2
finned-x-wing-in-columns: n5{c5 c1}{r8 r1} ==> r1c2≠5
```

```
+-------------------+-------------------+-------------------+
! 12569 1236  39    ! 4     2569  23569 ! 7     8     269   !
! 4     2568  789   ! 1     78    2569  ! 269   2569  3     !
! 2569  23568 3789  ! 2569  78    23569 ! 1     2569  4     !
+-------------------+-------------------+-------------------+
! 269   4     1     ! 3     269   7     ! 8     269   5     !
! 2679  2367  5     ! 269   1     8     ! 23469 23469 269   !
! 269   2368  389   ! 2569  4     2569  ! 2369  1     7     !
+-------------------+-------------------+-------------------+
! 3     17    4     ! 269   269   1269  ! 5     2679  8     !
! 157   9     6     ! 8     25    1245  ! 234   2347  12    !
! 8     15    2     ! 7     3     14569 ! 469   469   169   !
+-------------------+-------------------+-------------------+
```

```
OR6-anti-tridagon[12] for digits 2, 6 and 9 in blocks:
        b2, with cells: r1c5, r2c6, r3c4
        b3, with cells: r1c9, r2c7, r3c8
        b5, with cells: r4c5, r6c6, r5c4
        b6, with cells: r4c8, r6c7, r5c9
with 6 guardians: n5r1c5 n5r2c6 n5r3c4 n5r3c8 n5r6c6 n3r6c7
```

At this point, none of the activated rules can be applied. But we have found an anti-tridagon pattern with 6 guardians. This is quite a lot of guardians, but there is an easy way of checking if some of them can be eliminated. Load CSP-Rules with SFin plus, say whips of length ≤ 8, initialize it with the previous resolution state and try to eliminate these 6 guardians:

```
(init-sukaku-grid
             +-------------------+-------------------+-------------------+
             ! 12569 1236  39    ! 4     2569  23569 ! 7     8     269   !
             ! 4     2568  789   ! 1     78    2569  ! 269   2569  3     !
             ! 2569  23568 3789  ! 2569  78    23569 ! 1     2569  4     !
             +-------------------+-------------------+-------------------+
             ! 269   4     1     ! 3     269   7     ! 8     269   5     !
             ! 2679  2367  5     ! 269   1     8     ! 23469 23469 269   !
             ! 269   2368  389   ! 2569  4     2569  ! 2369  1     7     !
             +-------------------+-------------------+-------------------+
             ! 3     17    4     ! 269   269   1269  ! 5     2679  8     !
             ! 157   9     6     ! 8     25    1245  ! 234   2347  12    !
             ! 8     15    2     ! 7     3     14569 ! 469   469   169   !
             +-------------------+-------------------+-------------------+
)
(try-to-eliminate (list-of-nirjck-to-list-of-labels n5r1c5 n5r2c6 n5r3c4 n5r3c8 n5r6c6
n3r6c7))
```

```
whip[4]: r5n3{c8 c2} - r5n7{c2 c1} - r8n7{c1 c8} - r8n3{c8 .} ==> r6c7≠3
whip[1]: r6n3{c3 .} ==> r5c2≠3
```

Only one of the 6 guardians can be eliminated, but that's enough to fall into what was available in CSP-Rules at the time of this writing (OR$_k$-chains are now defined for k = 2, …, 8, but k = 7 or 8 implies high computational complexity). It requires at least W4. The full solution of this puzzle will finally require W7, but the only purpose here was to illustrate how to focus on eliminating a few guardians. Starting from RS1:

```
hidden-pairs-in-a-column: c5{n7 n8}{r2 r3} ==> r3c5≠9, r3c5≠6, r3c5≠5, r3c5≠2, r2c5≠9,
r2c5≠6, r2c5≠5, r2c5≠2
finned-x-wing-in-columns: n5{c5 c1}{r8 r1} ==> r1c2≠5
biv-chain[3]: r5n7{c2 c1} - r8n7{c1 c8} - c8n3{r8 r5} ==> r5c2≠3
whip[1]: r5n3{c8 .} ==> r6c7≠3
hidden-pairs-in-a-block: b6{n3 n4}{r5c7 r5c8} ==> r5c8≠9, r5c8≠6, r5c8≠2, r5c7≠9,
r5c7≠6, r5c7≠2
hidden-pairs-in-a-row: r6{n3 n8}{c2 c3} ==> r6c3≠9, r6c2≠6, r6c2≠2
whip[1]: c3n9{r3 .} ==> r1c1≠9, r3c1≠9
z-chain[4]: c2n6{r3 r5} - c9n6{r5 r9} - c9n1{r9 r8} - c1n1{r8 .} ==> r1c1≠6
t-whip[4]: c6n4{r8 r9} - r9n5{c6 c2} - r9n1{c2 c9} - r8c9{n1 .} ==> r8c6≠2
t-whip[5]: r8c5{n5 n2} - r7n2{c6 c8} - c8n7{r7 r8} - r8n3{c8 c7} - r8n4{c7 .} ==>
r8c6≠5
```

```
+-------------------+-------------------+-------------------+
! 125   1236  39    ! 4     2569  23569 ! 7     8     269   !
! 4     2568  789   ! 1     78    2569  ! 269   2569  3     !
! 256   23568 3789  ! 2569  78    23569 ! 1     2569  4     !
+-------------------+-------------------+-------------------+
! 269   4     1     ! 3     269   7     ! 8     269   5     !
! 2679  267   5     ! 269   1     8     ! 34    34    269   !
! 269   38    38    ! 2569  4     2569  ! 269   1     7     !
+-------------------+-------------------+-------------------+
! 3     17    4     ! 269   269   1269  ! 5     2679  8     !
! 157   9     6     ! 8     25    14    ! 234   2347  12    !
! 8     15    2     ! 7     3     14569 ! 469   469   169   !
+-------------------+-------------------+-------------------+
```

```
OR5-anti-tridagon[12] for digits 2, 6 and 9 in blocks:
        b2, with cells: r1c5, r2c6, r3c4
        b3, with cells: r1c9, r2c7, r3c8
        b5, with cells: r4c5, r6c6, r5c4
        b6, with cells: r4c8, r6c7, r5c9
with 5 guardians: n5r1c5 n5r2c6 n5r3c4 n5r3c8 n5r6c6
```

As expected, when an anti-tridagon is found, it has fewer than 6 guardians. In the present case (which is also a reason for my choosing this puzzle), one finds the elimination of a candidate that is directly linked to the 5 guardians (an OR_5-whip[1]):

```
Trid-OR5-whip[1]:
   || n5r1c5 -
   || n5r2c6 -
   || n5r3c4 -
   || n5r3c8 -
   || n5r6c6 -
 ==> r3c6≠5
```

```
Trid-OR5-forcing-whip-elim[4]:
   || n5r3c4 -
   || n5r3c8 -
   || n5r1c5 - partial-whip[1]: r8n5{c5 c1} -
   || n5r2c6 - partial-whip[1]: c8n5{r2 r3} -
   || n5r6c6 - partial-whip[1]: c4n5{r6 r3} -
 ==> r3c1≠5

hidden-pairs-in-a-column: c1{n1 n5}{r1 r8} ==> r8c1≠7, r1c1≠2
singles ==> r7c2=7, r5c1=7, r7c6=1, r8c6=4, r8c8=7, r8c7=3
naked-single ==> r5c7=4, r5c8=3, r9c8=4, r1c6≠5
t-whip[6]: c2n1{r1 r9} - c9n1{r9 r8} - b9n2{r8c9 r7c8} - b8n2{r7c4 r8c5} - r4n2{c5 c1}
- r3c1{n2 .} ==> r1c2≠6
z-chain[3]: r1n6{c6 c9} - r5n6{c9 c2} - b1n6{r2c2 .} ==> r3c4≠6
t-whip[7]: b2n6{r3c6 r1c5} - c5n5{r1 r8} - r8n2{c5 c9} - r1c9{n2 n9} - r5n9{c9 c4} -
c5n9{r4 r7} - r9c6{n9 .} ==> r6c6≠6
whip[7]: c7n2{r2 r6} - c8n2{r4 r7} - b8n2{r7c4 r8c5} - r4n2{c5 c1} - c1n9{r4 r6} -
r6c6{n9 n5} - b8n5{r9c6 .} ==> r1c9≠2
z-chain[5]: c7n2{r2 r6} - c9n2{r5 r8} - r8n1{c9 c1} - r1n1{c1 c2} - r1n2{c2 .} ==>
r2c6≠2
whip[6]: r5n9{c4 c9} - r1c9{n9 n6} - r9c9{n6 n1} - r9n9{c9 c7} - r9n6{c7 c6} -
b2n6{r1c6 .} ==> r7c4≠9
whip[5]: r7c4{n2 n6} - b5n6{r5c4 r4c5} - c5n9{r4 r1} - r1c9{n9 n6} - c8n6{r2 .} ==>
r7c5≠2
biv-chain[2]: c9n2{r5 r8} - b8n2{r8c5 r7c4} ==> r5c4≠2
biv-chain[2]: c7n2{r2 r6} - r5n2{c9 c2} ==> r2c2≠2
whip[1]: r2n2{c8 .} ==> r3c8≠2
biv-chain[3]: b8n2{r7c4 r8c5} - c5n5{r8 r1} - c4n5{r3 r6} ==> r6c4≠2
biv-chain[3]: c4n2{r3 r7} - r8n2{c5 c9} - r5n2{c9 c2} ==> r3c2≠2
biv-chain[3]: b5n2{r6c6 r4c5} - r8n2{c5 c9} - r5n2{c9 c2} ==> r6c1≠2
biv-chain[2]: b5n2{r6c6 r4c5} - c1n2{r4 r3} ==> r3c6≠2
biv-chain[4]: c4n5{r6 r3} - b3n5{r3c8 r2c8} - b3n2{r2c8 r2c7} - r6n2{c7 c6} ==> r6c6≠5
hidden-single-in-a-block ==> r6c4=5
biv-chain[3]: r3c4{n9 n2} - r7c4{n2 n6} - r7c5{n6 n9} ==> r1c5≠9
biv-chain[3]: c4n6{r5 r7} - r7n2{c4 c8} - c9n2{r8 r5} ==> r5c9≠6
finned-x-wing-in-columns: n6{c9 c6}{r9 r1} ==> r1c5≠6
whip[1]: b2n6{r3c6 .} ==> r9c6≠6
whip[1]: r9n6{c9 .} ==> r7c8≠6
naked-pairs-in-a-column: c5{r1 r8}{n2 n5} ==> r4c5≠2
singles ==> r6c6=2, r2c7=2
naked-triplets-in-a-row: r1{c3 c6 c9}{n9 n3 n6} ==> r1c2≠3
finned-swordfish-in-columns: n9{c1 c7 c5}{r4 r6 r9} ==> r9c6≠9
stte
```

14.10 Tridagon-OR$_k$-contrad-whips

Generic OR$_k$-contrad-whips have been defined in section 3.5.2. Here, we combine their use with OR$_k$-anti-tridagon relations. An example should be enough to make the idea clear. As for selecting them in the configuration file, the latter has a part similar to the one for Tridagon-OR$_k$-forcing-whips (see section 14.18 for details).

The notation for an OR_k-contrad-whip (or an OR_k-whip in the next section) based on an OR_k-relation is as follows:

– the full name is <ORname>-OR_k-whip[n], with "n" the total length as usual, "k" the size or the OR relation and <OR-name> the short name of this relation (here Trid);

– double curly braces are used for the OR_k part (to recall that it is not a CSP-Variable);

– contrary to CSP-Variables, all the OR_k candidates are written inside the OR_k relation, the first of them is the left-linking candidate;

– a vertical bar is used inside the OR_k-relation to separate the right-linking candidate (possibly none, i.e. a dot, for OR_k-contrad-whips) from the rest.

Consider puzzle #25591 in mith's list of 63,137 min-expands:

```
+-------+-------+-------+
! 1 . 3 ! . . . ! . . . !
! . 5 . ! 1 . 9 ! . . 6 !
! 6 9 . ! 2 3 . ! . . . !
+-------+-------+-------+
! 2 . . ! . . . ! 4 . 8 !
! 3 6 . ! . . . ! 1 2 7 !
! . . . ! . . 2 ! . 9 5 !
+-------+-------+-------+
! 5 . 6 ! . 2 3 ! . . . !
! . . 1 ! 6 9 . ! . 5 . !
! 9 . . ! 5 . 1 ! . . . !
+-------+-------+-------+
1.3.......5.1.9..669.23....2.....4.836....127.....2.955.6.23.....169..5.9..5.1...;5250
;73296
SER = 11.0
```

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

```
+-------------------+-------------------+-------------------+
! 1     2478  3     ! 478   45678 45678 ! 25789 478   249   !
! 478   5     2478  ! 1     478   9     ! 2378  3478  6     !
! 6     9     478   ! 2     3     4578  ! 578   1478  14    !
+-------------------+-------------------+-------------------+
! 2     17    579   ! 379   1567  567   ! 4     36    8     !
! 3     6     4589  ! 489   458   458   ! 1     2     7     !
! 478   1478  478   ! 3478  14678 2     ! 36    9     5     !
+-------------------+-------------------+-------------------+
! 5     478   6     ! 478   2     3     ! 789   1478  149   !
! 478   23478 1     ! 6     9     478   ! 278   5     234   !
! 9     23478 2478  ! 5     478   1     ! 2678  4678  234   !
+-------------------+-------------------+-------------------+
169 candidates.
```

```
hidden-pairs-in-a-column: c3{n5 n9}{r4 r5} ==> r5c3≠8, r5c3≠4, r4c3≠7
whip[1]: r5n4{c6 .} ==> r6c4≠4, r6c5≠4
```

```
whip[1]: r5n8{c6 .} ==> r6c4≠8, r6c5≠8
z-chain[5]: c2n3{r8 r9} - b7n2{r9c2 r9c3} - r2n2{c3 c7} - r8n2{c7 c9} - r8n3{c9 .} ==>
r8c2≠4, r8c2≠8, r8c2≠7
whip[5]: c8n6{r9 r4} - r4n3{c8 c4} - r6c4{n3 n7} - r7n7{c4 c2} - r4n7{c2 .} ==> r9c8≠7
```

```
+-------------------+-------------------+-------------------+
! 1      2478  3     ! 478    45678 45678 ! 25789 478   249   !
! 478    5     2478  ! 1      478   9     ! 2378  3478  6     !
! 6      9     478   ! 2      3     4578  ! 578   1478  14    !
+-------------------+-------------------+-------------------+
! 2      17    59    ! 379    1567  567   ! 4     36    8     !
! 3      6     59    ! 489    458   458   ! 1     2     7     !
! 478    1478  478   ! 37     167   2     ! 36    9     5     !
+-------------------+-------------------+-------------------+
! 5      478   6     ! 478    2     3     ! 789   1478  149   !
! 478    23    1     ! 6      9     478   ! 278   5     234   !
! 9      23478 2478  ! 5      478   1     ! 2678  468   234   !
+-------------------+-------------------+-------------------+
```

At this point, an OR3 anti-tridagon appears and (with a t-whip[5] in between them) it allows 7 direct eliminations with OR3-contrad-whips of lengths 4 and 5, which is enough to solve the puzzle:

```
OR3-anti-tridagon[12] for digits 4, 7 and 8 in blocks:
        b1, with cells: r1c2, r2c1, r3c3
        b2, with cells: r1c4, r2c5, r3c6
        b7, with cells: r7c2, r8c1, r9c3
        b8, with cells: r7c4, r8c6, r9c5
with 3 guardians: n2r1c2 n5r3c6 n2r9c3
```

Trid-OR3-ctr-whip[4]: c3n2{r9 r2} - r1n2{c2 c7} - c7n5{r1 r3} - OR3{{n2r9c3 n5r3c6 n2r1c2 | .}} ==> r9c9≠2

Trid-OR3-ctr-whip[5]: r8c2{n3 n2} - c9n2{r8 r1} - r1n9{c9 c7} - c7n5{r1 r3} - OR3{{n2r1c2 n5r3c6 n2r9c3 | .}} ==> r9c2≠3

```
hidden-single-in-a-block ==> r8c2=3
hidden-single-in-a-block ==> r9c9=3
whip[1]: r8n2{c9 .} ==> r9c7≠2
t-whip[5]: c2n2{r9 r1} - r2n2{c3 c7} - c7n3{r2 r6} - r6c4{n3 n7} - r4n7{c6 .} ==>
r9c2≠7
```

Trid-OR3-ctr-whip[5]: r2n2{c7 c3} - r1n2{c2 c9} - r1n9{c9 c7} - c7n5{r1 r3} - OR3{{n2r9c3 n5r3c6 n2r1c2 | .}} ==> r8c7≠2

hidden-single-in-a-block ==> r8c9=2

Trid-OR3-ctr-whip[4]: b1n2{r1c2 r2c3} - b3n2{r2c7 r1c7} - c7n5{r1 r3} - OR3{{n2r1c2 n5r3c6 n2r9c3 | .}} ==> r1c2≠4

Trid-OR3-ctr-whip[4]: b1n2{r1c2 r2c3} - b3n2{r2c7 r1c7} - c7n5{r1 r3} - OR3{{n2r1c2 n5r3c6 n2r9c3 | .}} ==> r1c2≠7

```
Trid-OR3-ctr-whip[4]: r2n2{c7 c3} - r1c2{n2 n8} - b2n8{r1c4 r3c6} - OR3{{n2r1c2 n5r3c6
n2r9c3 | .}} ==> r2c7≠8

Trid-OR3-ctr-whip[4]: b1n2{r1c2 r2c3} - b3n2{r2c7 r1c7} - c7n5{r1 r3} - OR3{{n2r1c2
n5r3c6 n2r9c3 | .}} ==> r1c2≠8
stte
```

14.11 Tridagon-OR$_k$-whips

Generic OR$_k$-whips have been defined in section 3.5.3. Here, we combine their use with OR$_k$-anti-tridagon relations.

Examples should be enough to make the idea clear.

14.11.2 First example

Take #1359 in mith's list of 63137 min-expand puzzles in T&E(3).

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

```
.........4.6.8..237...32..42.8.634..34..7.6...67..4.........958.4.....6....27.41;250;
48833
SER = 11.7
```

The start is quite easy, using only short reversible chains.

```
hidden-pairs-in-a-row: r1{n2 n3}{c2 c3} ==> r1c3≠9, r1c3≠5, r1c3≠1, r1c2≠9, r1c2≠8,
r1c2≠5, r1c2≠1
hidden-single-in-a-block ==> r3c2=8
biv-chain[4]: r1n4{c4 c5} - r7c5{n4 n1} - r7c1{n1 n6} - c6n6{r7 r1} ==> r1c4≠6
z-chain[5]: r8n2{c2 c7} - r8n7{c7 c8} - r4n7{c8 c9} - r4n9{c9 c4} - r9n9{c4 .} ==>
r8c2≠9
whip[1]: r8n9{c6 .} ==> r9c4≠9
```

```
+-------------------+-------------------+-------------------+
! 159   23    23    ! 14579 1459  1569  ! 1579  15678 789   !
! 4     159   6     ! 1579  8     159   ! 1579  2     3     !
! 7     8     159   ! 1569  3     2     ! 159   156   4     !
+-------------------+-------------------+-------------------+
```

```
! 2     159   8     ! 159   6     3     ! 4     157   79    !
! 3     4     159   ! 12589 7     1589  ! 6     158   289   !
! 159   6     7     ! 12589 159   4     ! 1359  1358  289   !
+-------------------+-------------------+-------------------+
! 16    1237  123   ! 13468 14    168   ! 2378  9     5     !
! 8     12357 4     ! 1359  159   159   ! 237   37    6     !
! 569   359   359   ! 3568  2     7     ! 38    4     1     !
+-------------------+-------------------+-------------------+
```

```
OR3-anti-tridagon[12] for digits 1, 5 and 9 in blocks:
        b1, with cells: r1c1, r2c2, r3c3
        b2, with cells: r1c5, r2c6, r3c4
        b4, with cells: r6c1, r4c2, r5c3
        b5, with cells: r6c5, r4c4, r5c6
with 3 guardians: n4r1c5 n6r3c4 n8r5c6[/code]
```

This is where the OR_k-whips appear:

Trid-OR3-whip[4]: r7c1{n6 n1} - r7c6{n1 n8} - OR3{{n8r5c6 n6r3c4 | n4r1c5}} - r7c5{n4 .} ==> r7c4≠6
z-chain[4]: c7n1{r3 r6} - c1n1{r6 r7} - r7n6{c1 c6} - r1n6{c6 .} ==> r1c8≠1
Trid-OR3-whip[5]: r7c1{n1 n6} - r9n6{c1 c4} - r7c6{n6 n8} - OR3{{n8r5c6 n6r3c4 | n4r1c5}} - r7c5{n4 .} ==> r7c4≠1, r7c2≠1, r7c3≠1

The end is easy, in W5:

```
naked-pairs-in-a-column: c3{r1 r7}{n2 n3} ==> r9c3≠3
t-whip[5]: r1n6{c8 c6} - r7n6{c6 c1} - b7n1{r7c1 r8c2} - r8n2{c2 c7} - r8n7{c7 .} ==>
r1c8≠7
biv-chain[4]: b3n8{r1c8 r1c9} - c9n7{r1 r4} - c8n7{r4 r8} - c8n3{r8 r6} ==> r6c8≠8
hidden-pairs-in-a-row: r6{n2 n8}{c4 c9} ==> r6c9≠9, r6c4≠9, r6c4≠5, r6c4≠1
biv-chain[3]: r1n6{c6 c8} - c8n8{r1 r5} - c6n8{r5 r7} ==> r7c6≠6
singles ==> r9c4=6, r1c6=6, r3c8=6, r7c1=6, r8c2=1, r8c7=2, r8c8=7, r4c9=7, r7c2=7,
r7c3=2, r1c3=3, r1c2=2, r9c2=3,r9c7=8, r7c7=3, r6c8=3, r8c4=3
whip[1]: c8n1{r5 .} ==> r6c7≠1
z-chain[3]: c9n9{r5 r1} - r3n9{c7 c4} - r4n9{c4 .} ==> r5c3≠9
biv-chain[3]: c1n1{r1 r6} - r5c3{n1 n5} - b7n5{r9c3 r9c1} ==> r1c1≠5
biv-chain[3]: r2c2{n5 n9} - b4n9{r4c2 r6c1} - r6c7{n9 n5} ==> r2c7≠5
biv-chain[4]: r5c3{n5 n1} - r6n1{c1 c5} - b8n1{r7c5 r7c6} - c6n8{r7 r5} ==> r5c6≠5
biv-chain[4]: b4n9{r6c1 r4c2} - r2c2{n9 n5} - c6n5{r2 r8} - b8n9{r8c6 r8c5} ==> r6c5≠9
biv-chain[2]: r6n9{c7 c1} - c2n9{r4 r2} ==> r2c7≠9
finned-x-wing-in-rows: n9{r4 r2}{c2 c4} ==> r3c4≠9, r1c4≠9
finned-x-wing-in-rows: n9{r6 r3}{c7 c1} ==> r1c1≠9
stte
```

14.11.3 Second example

Here is an easier example, #1167 in mith's list of 63137 min-expand puzzle in T&E(3):

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

```
+-------+-------+-------+
! . . . ! . 3 4 ! 5 . 8 !
! . . . ! 6 . 5 ! . . 2 !
! . . 5 ! 8 2 . ! 6 . . !
+-------+-------+-------+
! . 7 . ! . . . ! . . 5 !
! . 9 . ! . . . ! 8 . 4 !
! . . 2 ! . . 3 ! . . . !
+-------+-------+-------+
...45..8.45....2.368.3.245.....345.8...6.5..2..582.6...7......5.9....8.4..2..3...;234;
17556
SER = 11.7
```

As usual, using the simplest-first strategy, let's first do some easy cleaning:

```
hidden-pairs-in-a-column: c3{n4 n8}{r5 r7} ==> r7c3≠6, r7c3≠3, r7c3≠1, r5c3≠9, r5c3≠7,
r5c3≠3, r5c3≠1
biv-chain[3]: r9n8{c1 c5} - b8n4{r9c5 r7c5} - r7c3{n4 n8} ==> r7c1≠8
biv-chain[3]: r9n8{c5 c1} - r7c3{n8 n4} - b8n4{r7c5 r9c5} ==> r9c5≠1, r9c5≠6, r9c5≠7,
r9c5≠9
z-chain[3]: r9n7{c9 c4} - c4n5{r9 r8} - r8n2{c4 .} ==> r8c8≠7
whip[1]: b9n7{r9c9 .} ==> r9c4≠7
z-chain[3]: r7c1{n1 n3} - b9n3{r7c7 r8c8} - c8n2{r8 .} ==> r7c8≠1
z-chain[4]: r7c1{n1 n3} - r8n3{c3 c8} - r8n2{c8 c4} - r8n5{c4 .} ==> r8c1≠1
```

```
+-------------------+-------------------+-------------------+
! 12379 123   1379  ! 4     5     1679  ! 179   8     1679  !
! 4     5     179   ! 179   16789 16789 ! 2     1679  3     !
! 6     8     179   ! 3     179   2     ! 4     5     179   !
+-------------------+-------------------+-------------------+
! 1279  126   1679  ! 179   3     4     ! 5     179   8     !
! 13789 134   48    ! 6     179   5     ! 1379  13479 2     !
! 1379  134   5     ! 8     2     179   ! 6     13479 179   !
+-------------------+-------------------+-------------------+
! 13    7     48    ! 129   14689 1689  ! 139   2369  5     !
! 35    9     136   ! 1257  167   167   ! 8     1236  4     !
! 158   146   2     ! 159   48    3     ! 179   1679  1679  !
+-------------------+-------------------+-------------------+
```

```
OR3-anti-tridagon[12] for digits 7, 9 and 1 in blocks:
        b2, with cells: r1c6, r2c4, r3c5
        b3, with cells: r1c7, r2c8, r3c9
        b5, with cells: r6c6, r4c4, r5c5
        b6, with cells: r6c9, r4c8, r5c7
with 3 guardians: n6r1c6 n6r2c8 n3r5c7
```

The part with the OR$_k$-whips:

`Trid-OR3-whip[4]: c9n6{r9 r1} - OR3{{n6r2c8 n6r1c6 | n3r5c7}} - b9n3{r7c7 r7c8} - c8n2{r7 .} ==> r8c8≠6, r7c8≠6`

The end is easy, in S3+W3:

```
whip[1]: b9n6{r9c9 .} ==> r9c2≠6
singles ==> r8c3=6, r4c2=6, r4c1=2, r1c2=2, r1c3=3
whip[1]: c2n3{r6 .} ==> r5c1≠3, r6c1≠3
naked-pairs-in-a-block: b8{r8c5 r8c6}{n1 n7} ==> r9c4≠1, r8c4≠7, r8c4≠1, r7c6≠1,
r7c5≠1, r7c4≠1
whip[1]: b8n1{r8c6 .} ==> r8c8≠1
hidden-pairs-in-a-column: c4{n1 n7}{r2 r4} ==> r4c4≠9, r2c4≠9
whip[1]: c4n9{r9 .} ==> r7c5≠9, r7c6≠9
hidden-pairs-in-a-row: r6{n3 n4}{c2 c8} ==> r6c8≠9, r6c8≠7, r6c8≠1, r6c2≠1
biv-chain[2]: r7n1{c7 c1} - c2n1{r9 r5} ==> r5c7≠1
naked-triplets-in-a-column: c5{r3 r5 r8}{n1 n9 n7} ==> r2c5≠9, r2c5≠7, r2c5≠1
biv-chain[3]: c7n3{r5 r7} - r7n1{c7 c1} - c2n1{r9 r5} ==> r5c2≠3
hidden-single-in-a-block ==> r6c2=3
naked-single ==> r6c8=4
z-chain[3]: r4n9{c8 c3} - r2n9{c3 c6} - b5n9{r6c6 .} ==> r5c8≠9
z-chain[3]: r4n9{c8 c3} - r3n9{c3 c5} - b5n9{r5c5 .} ==> r6c9≠9
t-whip[3]: r4n9{c8 c3} - r6n9{c1 c6} - r2n9{c6 .} ==> r9c8≠9, r7c8≠9
naked-pairs-in-a-block: b9{r7c8 r8c8}{n2 n3} ==> r7c7≠3
singles ==> r5c7=3, r4c8=9
whip[1]: b4n9{r6c1 .} ==> r1c1≠9
biv-chain[2]: b6n1{r6c9 r5c8} - c2n1{r5 r9} ==> r9c9≠1
finned-swordfish-in-columns: n1{c8 c2 c4}{r2 r9 r5} ==> r5c5≠1
biv-chain[3]: r4n7{c3 c4} - r5c5{n7 n9} - b4n9{r5c1 r6c1} ==> r6c1≠7
biv-chain[2]: b6n7{r6c9 r5c8} - c1n7{r5 r1} ==> r1c9≠7
biv-chain[3]: c7n7{r9 r1} - c1n7{r1 r5} - b6n7{r5c8 r6c9} ==> r9c9≠7
biv-chain[3]: r9n7{c7 c8} - r5c8{n7 n1} - c2n1{r5 r9} ==> r9c7≠1
biv-chain[3]: r2c4{n7 n1} - c5n1{r3 r8} - b8n7{r8c5 r8c6} ==> r1c6≠7, r2c6≠7
biv-chain[2]: b2n7{r3c5 r2c4} - r4n7{c4 c3} ==> r3c3≠7
x-wing-in-columns: n7{c3 c4}{r2 r4} ==> r2c8≠7
finned-x-wing-in-rows: n7{r3 r6}{c9 c5} ==> r5c5≠7
singles ==> r5c5=9, r6c1=9
naked-pairs-in-a-block: b2{r2c4 r3c5}{n1 n7} ==> r2c6≠1, r1c6≠1
biv-chain[2]: b2n1{r3c5 r2c4} - r4n1{c4 c3} ==> r3c3≠1
stte
```

14.11.4 Third example

Our third example (#1888) has a simplest-first solution using only chains of lengths ≤ 5 and it has a large number (12) of direct eliminations by OR_k-whips based on the anti-tridagon pattern:

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

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

```
12....78.4.7.8.2.6.68.2..142.....84.........7....46.1....7.1..67.5.....81.9.3...;326;
25948 ; SER = 11.6
```

```
Resolution state after Singles and whips[1]:
    +-------------------------+-------------------------+-------------------------+
    ! 1       2       359     ! 346     34569   569     ! 7       8       359     !
    ! 4       359     7       ! 13      8       159     ! 2       359     6       !
    ! 359     6       8       ! 37      2       579     ! 359     1       4       !
    +-------------------------+-------------------------+-------------------------+
    ! 2       359     13569   ! 1367    13569   15679   ! 8       4       3579    !
    ! 359     34589   134569  ! 123678  13569   1256789 ! 359     23579   23579   !
    ! 7       3589    359     ! 238     359     4       ! 6       2359    1       !
    +-------------------------+-------------------------+-------------------------+
    ! 359     3459    23459   ! 2468    7       268     ! 1       23569   23589   !
    ! 6       7       2349    ! 5       14      128     ! 349     239     2389    !
    ! 8       1       245     ! 9       46      3       ! 45      2567    257     !
    +-------------------------+-------------------------+-------------------------+
189 candidates.
```

```
hidden-pairs-in-a-column: c3{n1 n6}{r4 r5} ==> r5c3≠9, r5c3≠5, r5c3≠4, r5c3≠3, r4c3≠9,
r4c3≠5, r4c3≠3
singles ==> r5c2=4, r6c2=8
biv-chain[3]: r9c7{n5 n4} - r9c5{n4 n6} - b9n6{r9c8 r7c8} ==> r7c8≠5
```

```
    +-------------------------+-------------------------+-------------------------+
    ! 1       2       359     ! 346     34569   569     ! 7       8       359     !
    ! 4       359     7       ! 13      8       159     ! 2       359     6       !
    ! 359     6       8       ! 37      2       579     ! 359     1       4       !
    +-------------------------+-------------------------+-------------------------+
    ! 2       359     16      ! 1367    13569   15679   ! 8       4       3579    !
    ! 359     4       16      ! 123678  13569   1256789 ! 359     23579   23579   !
    ! 7       8       359     ! 23      359     4       ! 6       2359    1       !
    +-------------------------+-------------------------+-------------------------+
    ! 359     359     23459   ! 2468    7       268     ! 1       2369    23589   !
    ! 6       7       2349    ! 5       14      128     ! 349     239     2389    !
    ! 8       1       245     ! 9       46      3       ! 45      2567    257     !
    +-------------------------+-------------------------+-------------------------+
```

```
OR2-anti-tridagon[12] for digits 3, 5 and 9 in blocks:
        b1, with cells: r1c3, r2c2, r3c1
        b3, with cells: r1c9, r2c8, r3c7
        b4, with cells: r6c3, r4c2, r5c1
        b6, with cells: r6c8, r4c9, r5c7
with 2 guardians: n7r4c9 n2r6c8
```

This is now the main part:

```
Trid-OR2-whip[2]: OR2{{n2r6c8 | n7r4c9}} - r9n7{c9 .} ==> r9c8≠2
Trid-OR2-whip[3]: OR2{{n7r4c9 | n2r6c8}} - r6c4{n2 n3} - r3c4{n3 .} ==> r4c4≠7
Trid-OR2-whip[3]: OR2{{n2r6c8 | n7r4c9}} - r9n7{c9 c8} - c8n6{r9 .} ==> r7c8≠2
Trid-OR2-whip[4]: r6c4{n3 n2} - OR2{{n2r6c8 | n7r4c9}} - b5n7{r4c6 r5c6} - r5n8{c6 .}
==> r5c4≠3
Trid-OR2-whip[4]: r5n8{c6 c4} - b5n7{r5c4 r4c6} - OR2{{n7r4c9 | n2r6c8}} -
b5n2{r6c4 .} ==> r5c6≠9, r5c6≠6, r5c6≠5, r5c6≠1
Trid-OR2-whip[4]: r5n8{c4 c6} - b5n7{r5c6 r4c6} - OR2{{n7r4c9 | n2r6c8}} -
b5n2{r6c4 .} ==> r5c4≠6
biv-chain[4]: r8c5{n1 n4} - r9c5{n4 n6} - r5n6{c5 c3} - b4n1{r5c3 r4c3} ==> r4c5≠1
Trid-OR2-whip[4]: r5n8{c4 c6} - b5n7{r5c6 r4c6} - OR2{{n7r4c9 | n2r6c8}} -
b5n2{r6c4 .} ==> r5c4≠1
hidden-pairs-in-a-row: r5{n1 n6}{c3 c5} ==> r5c5≠9, r5c5≠5, r5c5≠3
naked-triplets-in-a-column: c5{r5 r8 r9}{n6 n1 n4} ==> r4c5≠6, r1c5≠6, r1c5≠4
singles ==> r1c4=4, r1c6=6, r7c3=4
biv-chain[4]: r5c5{n1 n6} - r9n6{c5 c8} - r9n7{c8 c9} - r4n7{c9 c6} ==> r4c6≠1
hidden-pairs-in-a-block: b5{n1 n6}{r4c4 r5c5} ==> r4c4≠3
Trid-OR2-ctr-whip[6]: r5n8{c6 c4} - b5n2{r5c4 r6c4} - r7c4{n2 n6} - c8n6{r7 r9} -
c8n7{r9 r5} - OR2{{n7r4c9 n2r6c8 | .}} ==> r5c6≠7
naked-pairs-in-a-column: c6{r5 r7}{n2 n8} ==> r8c6≠8, r8c6≠2
singles ==> r8c6=1, r8c5=4, r9c5=6, r5c5=1, r4c4=6, r4c3=1, r5c3=6, r2c4=1, r7c8=6,
r9c7=4, r8c9=8
whip[1]: b8n2{r7c6 .} ==> r7c9≠2
Trid-OR2-whip[2]: OR2{{n7r4c9 | n2r6c8}} - b9n2{r8c8 .} ==> r9c9≠7
```

The rest is easy, in Z4:

```
hidden-single-in-a-block ==> r9c8=7
whip[1]: b9n5{r9c9 .} ==> r1c9≠5, r4c9≠5, r5c9≠5
biv-chain[4]: r4n7{c9 c6} - r3n7{c6 c4} - b2n3{r3c4 r1c5} - r1c9{n3 n9} ==> r4c9≠9
z-chain[4]: c5n3{r6 r1} - c3n3{r1 r8} - r8n2{c3 c8} - r6n2{c8 .} ==> r6c4≠3
singles ==> r6c4=2, r5c6=8, r5c4=7, r3c4=3, r7c6=2, r7c4=8, r3c6=7, r4c9=7
finned-x-wing-in-rows: n3{r1 r7}{c9 c3} ==> r8c3≠3
whip[1]: r8n3{c8 .} ==> r7c9≠3
finned-x-wing-in-columns: n3{c9 c3}{r1 r5} ==> r5c1≠3
hidden-single-in-a-column ==> r7c1=3
whip[1]: r5n3{c9 .} ==> r6c8≠3
x-wing-in-columns: n5{c1 c7}{r3 r5} ==> r5c8≠5
finned-x-wing-in-columns: n5{c8 c6}{r2 r6} ==> r6c5≠5
whip[1]: b5n5{r4c6 .} ==> r4c2≠5
biv-chain[3]: r3n9{c1 c7} - r1c9{n9 n3} - b1n3{r1c3 r2c2} ==> r2c2≠9
biv-chain[3]: r2n9{c6 c8} - r2n3{c8 c2} - r4c2{n3 n9} ==> r4c6≠9
singles ==> r4c6=5, r2c6=9, r1c5=5
finned-x-wing-in-rows: n9{r1 r8}{c3 c9} ==> r7c9≠9
stte
```

14.12 Resolution power of OR_k-chains

The above sections have shown that puzzles in T&E(3) can be brought down to being solvable by short standard chains after a few anti-tridagon rules have been applied. But what is the resolution power of such rules? First of all, it is clear that it doesn't extend beyond puzzles that have a non-degenerated anti-tridagon pattern, because the degenerated versions are very difficult to spot.

Keeping these remarks in mind, this section compares the resolution powers of OR_k-forcing-whips, OR_k-contrad-whips, OR_k-whips and all these OR_k-chains together when they are used in conjunction with with ordinary whips and with the anti-tridagon pattern. Control variable ?*high-Tridagon-salience* was set to FALSE – but this shouldn't make much difference, considering the way the tables were computed: by progressively increasing the lengths of all the chains. The reference database is the already mentioned one, the 63,137 min-expand puzzles in T&E(3), developed by mith.

In addition to their intrinsic interest, and combined with your preferences for one type of OR_k-chain or another, the results below can be used as a guide for choosing the various parameters when you want to solve a puzzle.

There are a priori three parameters to consider: the maximal k used in the OR_k anti-tridagon relation, the maximal length of whips and the maximal length of OR_k-forcing-whips (resp. OR_k-contrad-whips, OR_k-whips). However, all these OR_k-chains are built on the same partial-whips as whips and the (statistically) best way to use all the partial whips underlying these three chain patterns is to take the same maximal lengths for whips, OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips. The systematic comparisons done here will stick to this idea, although SudoRules allows much more varied combinations that may produce more interesting individual solutions; you are strongly encouraged to try them, together with the various sets of preferences that will be defined later.

The first tables in the next four sub-sections will present the number of puzzles solved by SFin + Trid + W_n + respectively OR_kFW_n / OR_kCW_n / OR_kW_n / OR_kCH_n, for different values of k and n.

Values of k are in different lines and values of n in different columns; k=1 means that only the basic Tridagon elimination rule (Trid) has been used, possibly with ordinary whips, but with no OR_k chain. The starting point of the four tables is their first line after the title: 8,196 puzzles can readily be solved using only SFin (i.e. Subsets + Finned Fish) + Trid. All the tables also share the next line, where only SFin + Trid + whips[n] are used.

Each (k, n) cell (except the first column) has three values in it: the main one, in bold, in the lower right corner, is the total number of puzzles solved by e.g. SFin + Trid + W_n + respectively OR_kFW_n , OR_kCW_n , OR_kW_n , OR_kCH_n. The value above it is the difference with the previous line; it shows what's gained by increasing k by one. The value on the left of the main number is the difference with the previous cell (if any) in the same line; it shows what's gained by increasing n.

Some general conclusions can be drawn from all the tables in the next sub-sections:

- for fixed n, as k increases, the difference between two lines decreases quite fast; this shouldn't be too surprising, as larger k means e.g. that more partial-whips (with same total of lengths) have to converge to the same candidate;
- for fixed k, as n increases, the difference between two columns decreases quite fast; this shouldn't be too surprising either, as it already happens with all the "classical" chains (whips…);
- starting from k=2 and n=3, at any point in the table, it is much more fruitful to increase n than to increase k;
- for any k and n, OR_k-forcing-whips[n] have a larger resolution power than OR_k-contrad-whips[n];
- *for any k and n, there is a large overlap between puzzles that can be solved by OR_k-forcing-whips[n] and puzzles that can be solved by OR_k-whips[n];*
- *for any k and n, OR_k-whips[n] (which include OR_k-contrad-whips[n] as a special case) have a larger resolution power than OR_k-forcing-whips[n].*

14.12.1 Resolution power of OR_k-Forcing-Whips

Solved in SFin+Trid+W_n+OR_kFW_n							
	8 196	puzzles solved by SFin+Trid (among **63 137** min-expands)					
	n = 3	n = 5		n = 7		n = 8	
	8 137		17 532		21 160		22 332
k = 1	**16 333**	9 395	**25 728**	3 628	**29 356**	1 172	**30 528**
	3 350		8 713		11 231		12 068
k = 2	**19 683**	14 758	**34 441**	6 146	**40 587**	2 009	**42 596**
	428		2 255		3 471		3 886
k = 3	**20 111**	16 585	**36 696**	7 362	**44 058**	2 424	**46 482**
	67		365		540		722
k = 4	**20 178**	16 883	**37 061**	7 537	**44 598**	2 606	**47 204**

k = 5	2 **20 180**	28 16 909 **37 089**	99 7 608 **44 697**	119 2 626 **47 323**
k = 6	0 **20 180**	4 16 913 **37 093**		

14.12.2 Resolution power of OR_k-Contrad-Whips

Solved in SFin+Trid+W_n+OR_kCW_n				
	8 196	puzzles solved by SFin+Trid (among **63 137** min-expands)		
	n = 3	n = 5	n = 7	n = 8
k = 1	8 137 **16 333**	17 532 9 395 **25 728**	21 160 3 628 **29 356**	22 332 1 172 **30 528**
k = 2	1 700 **18 033**	6 276 13 971 **32 004**	8 863 6 215 **38 219**	9 944 2 253 **40 472**
k = 3	286 **18 319**	1 379 15 064 **33 383**	2 413 7 249 **40 632**	2 849 2 689 **43 321**
k = 4	49 **18 368**	319 15 334 **33 702**	478 7 408 **41 110**	640 2 851 **43 961**
k = 5	6 **18 374**	25 15 353 **33 727**	111 7 494 **41 221**	140 2 880 **44 101**

14.12.3 Resolution power of OR_k-Whips and comparison with OR_k-forcing-whips[n]

The first table below is as the previous ones, but in the second table, while k and n have the same meanings as before, each (k, n) cell has two values:

– the left value is the number of puzzles solved in SFin+Trid+W_n+OR_kFW_n but not in SFin+Trid+W_n+OR_kW_n;

– the right value is the number of puzzles solved in SFin+Trid+W_n+OR_kW_n but not in SFin+Trid+W_n+OR_kFW_n.

The calculations underlying the second table allow one more conclusion:

- *most of the puzzles that have a solution with OR_k-forcing-whips[n] also have one with OR_k-whips[n] but the converse is not true*: for instance, only 96 puzzles can be solved in SFin+Trid+W5+OR5FW5 but not in SFin+Trid+W5+OR5W5; whereas 1894 can be solved in SFin+Trid+W5+OR5W5 but not in SFin+Trid+W5+OR5FW5; and the difference is still larger for n=7.

Underlying calculations also show that:

- the difference between OR_k-forcing-whips and OR_k-whips is not well compensated by increasing the FW lengths: for instance, only 55 puzzles can be solved in SFin+Trid+W5+OR5FW5 but still not in SFin+Trid+W7+OR5W7; whereas 683 can be solved in SFin+Trid+W5+OR5W5 but still not in SFin+Trid+W7+OR5FW7.

Solved in SFin+Trid+W_n+OR_kW_n							
	8 196	puzzles solved by SFin+Trid (among **63 137** min-expands)					
	n = 3	n = 5		n = 7		n = 8	
	8 137		17 532		21 160		22 332
k = 1	**16 333**	9 395	**25 728**	3 628	**29 356**	1 172	**30 528**
	3 422		9 705		12 499		13 457
k = 2	**19 755**	15 678	**35 433**	6 422	**41 855**	2 130	**43 985**
	508		2 818		4 049		4 444
k = 3	**20 263**	17 988	**38 251**	7 653	**45 904**	2 525	**48 429**
	94		566		892		998
k = 4	**20 357**	18460	**38 817**	7 979	**46 796**	2 631	**49 427**
	7		70		179		234
k = 5	**20 364**	18 523	**38 887**	8 088	**46 975**	2 676	**49 651**

Solved in SFin+Trid+W_n+(OR_kFW_n vs OR_kW_n)

	n = 3		n = 5		n = 7		n = 8	
k = 2	66	138	22	1 014	3	1 271	3	1 392
k = 3	72	224	78	1633	81	1 927	70	2017
k = 4	75	254	96	1852	101	2 299	108	2 331
k = 5	77	261	96	1894	129	2 407	134	2 462

14.12.4 Resolution power of all the OR_k-chains together

Solved in SFin+Trid+W_n+OR_kFW_n+OR_kW_n

	8 196	puzzles solved by SFin+Trid (among **63 137** min-expands)					
	n = 3	n = 5		n = 7		n = 8	
	8 137		17 532		21 160		22 332
k = 1	**16 333**	9 395	**25 728**	3 628	**29 356**	1 172	**30 528**
	3 488		9 736		12 503		13 460
k = 2	**19 821**	15 643	**35 464**	6 395	**41 859**	2 129	**43 988**
	514		2 884		4 140		4 530
k = 3	**20 335**	18 013	**38 348**	7 651	**45 999**	2 519	**48 518**
	97		584		929		1 050
k = 4	**20 432**	18 500	**38 932**	7 996	46 928	2 640	**49 568**
	9		72		209		251
k = 5	**20 441**	18 563	**39 004**	8 133	**47 137**	2 682	**49 819**

14.13 General remarks on OR_k-chains

First remark: forcing-chains in general are very inelegant, because they start with a big OR and they suppose one follows two (or more) streams of reasoning in parallel. They amount to reasoning by cases – which mathematicians tend to avoid whenever possible. With respect to T&E, they also amount to having some T&E step right at the start (any OR-branching step in a chain amounts to introducing one level of T&E).

Ordinary forcing whips have no argument to make forgive this inelegance: they are statistically not more powerful than whips of same length (see [PBCS3, section 6.5.1]). Tridagon-based OR_k-forcing-whips are radically different, both in terms of resolution power (as shown in the tables of section 14.12) and in terms of focus: their OR starting point is concentrated on very specific candidates, the guardians – which drastically reduces the risk of combinatorial explosion.

Second remark: using the tridagon elimination rule, the anti-tridagon pattern and any OR_k-forcing-whip, OR_k-contrad-whip or OR_k-whip based on it does NOT suppose that the puzzle is consistent. It may seem so, because the argument is: if the puzzle is consistent, then the anti-tridagon pattern cannot be present; therefore at least one guardian must be True. And the above-mentioned rules are built on this to make eliminations. However, if the original puzzle is inconsistent, applying any of these rules cannot make it *more* inconsistent. Conclusion: the rules can be applied without making any ado about consistency.

Third remark: OR_k-contrad-whips being a special case of OR_k-whips, they could be merged with them. However, one may consider that they are easier to find; that's why CSP-Rules allows to activate them independently and, in any case, grants them a higher priority than OR_k-whips of same length.

Fourth remark: the final table shows that there remains much room for solutions with longer OR_k-chains; there also remains some room for solutions based on larger values of k, but this is less promising. CSP-Rules has OR_k-chains for k up to 8, but the computational complexity is too high for being useful. The value k=6 remains to be explored more extensively.

14.14 Preferences for OR_k-chains

Along with the introduction of OR_k-forcing-whips, OR_k-contrad-whips and OR_k-whips based on anti-tridagon OR_k relations, four application-specific sets of preferences have been defined in SudoRules; they are intended to be used with function "solve-with-preferences": ***TRID-ORk-FW***, ***TRID-ORk-CW***, ***TRID-ORk-W*** and ***TRID-ORk-CHAINS***. They all include SFin+Trid.

They respectively include the OR_k-forcing-whips and/or OR_k-contrad-whips and/or OR_k-whips, with the values of k selected in the configuration file for each of them (automatically extended downwards for k, in order to make the choices consistent) and with the relevant length restrictions defined by the choices in the configuration file for global variables: all-chains-max-length, OR_k-forcing-whips-max-length, OR_k-contrad-whips-max-length and OR_k-whips-max-length (here again adapted to be made consistent). The latter set (TRID-ORk-CHAINS) includes the above three types of Tridagon-OR_k-chains. As is the case with any other set of preferences, some puzzles have better solutions with and some without using them.

Notice that the tables in section 14.12 have been obtained without using preferences. The classifications might be slightly different if one uses them. The purpose of these tables is only to give an idea of what to expect with different values of k and n.

Be aware that using these preferences implies lots of unseen pattern-matching in the MAIN module (the module in which all the rules are loaded at start time) and in all the other selected OR_k chain modules; in particular, lots of partial-whips are found in the MAIN module, but not used in it. If chains lengths are chosen large, this may be very inefficient. The problem is, most puzzles in the known T&E(3) database require some mixing of ordinary chains and OR_k chains. That's why I rarely use the OR_k sets of preferences and, in any case, I don't use them before seeing what the solution without them (with ?*high-Tridagon-salience* set to TRUE) looks like. If it has many ordinary chains before any OR_k chain, it's likely not worth trying.

14.15 OR_k-forcing-g-whips

This section illustrates the combined use of Trid-OR_k-forcing-g-whips with the ultra-persistency of OR_k-relations. Here, ?*high-Tridagon-salience* is set to TRUE. The puzzle is #2423 in the 63,137 list.

```
+-------+-------+-------+
! . . . ! 4 . . ! 7 . 9 !
! . 5 7 ! . 8 9 ! . . . !
! . . 9 ! 7 3 2 ! . . . !
+-------+-------+-------+
! . 4 5 ! 9 . . ! . 1 . !
! 6 7 . ! . . . ! 5 9 . !
! 9 . 1 ! 5 . . ! 4 . . !
+-------+-------+-------+
! . 9 4 ! . . 1 ! 6 7 . !
! . . 6 ! . . . ! 9 . 1 !
! 7 1 . ! . 9 . ! . . . !
+-------+-------+-------+
...4..7.9.57.89.....9732....459...1.67....59.9.15..4...94..167...6...9.171..9....;418;
4232
SER = 11.7

Resolution state after Singles and whips[1]:
```

```
+-------------------+-------------------+-------------------+
! 1238  2368  238   ! 4     156   56    ! 7     2368  9     !
! 1234  5     7     ! 16    8     9     ! 123   2346  2346  !
! 148   68    9     ! 7     3     2     ! 18    4568  4568  !
+-------------------+-------------------+-------------------+
! 238   4     5     ! 9     267   3678  ! 238   1     23678 !
! 6     7     238   ! 1238  124   348   ! 5     9     238   !
! 9     238   1     ! 5     267   3678  ! 4     2368  23678 !
+-------------------+-------------------+-------------------+
! 2358  9     4     ! 238   25    1     ! 6     7     2358  !
! 2358  238   6     ! 238   2457  34578 ! 9     23458 1     !
! 7     1     238   ! 2368  9     34568 ! 238   23458 23458 !
+-------------------+-------------------+-------------------+
167 candidates.
137 g-candidates, 607 csp-glinks and 361 non-csp glinks
```

```
OR5-anti-tridagon[12] for digits 2, 3 and 8 in blocks:
        b4, with cells: r4c1, r5c3, r6c2
        b6, with cells: r4c7, r5c9, r6c8
        b7, with cells: r7c1, r9c3, r8c2
        b9, with cells: r7c9, r9c7, r8c8
with 5 guardians: n6r6c8 n5r7c1 n5r7c9 n4r8c8 n5r8c8

z-chain[3]: c5n4{r8 r5} - r5n1{c5 c4} - c4n2{r5 .} ==> r8c5≠2
biv-chain[4]: r1n5{c6 c5} - c5n1{r1 r5} - c5n4{r5 r8} - b8n7{r8c5 r8c6} ==> r8c6≠5
biv-chain[4]: r5n4{c6 c5} - b5n1{r5c5 r5c4} - r2c4{n1 n6} - b8n6{r9c4 r9c6} ==> r9c6≠4
whip[1]: r9n4{c9 .} ==> r8c8≠4
```

```
+-------------------+-------------------+-------------------+
! 1238  2368  238   ! 4     156   56    ! 7     2368  9     !
! 1234  5     7     ! 16    8     9     ! 123   2346  2346  !
! 148   68    9     ! 7     3     2     ! 18    4568  4568  !
+-------------------+-------------------+-------------------+
! 238   4     5     ! 9     267   3678  ! 238   1     23678 !
! 6     7     238   ! 1238  124   348   ! 5     9     238   !
! 9     238   1     ! 5     267   3678  ! 4     2368  23678 !
+-------------------+-------------------+-------------------+
! 2358  9     4     ! 238   25    1     ! 6     7     2358  !
! 2358  238   6     ! 238   457   3478  ! 9     2358  1     !
! 7     1     238   ! 2368  9     3568  ! 238   23458 23458 !
+-------------------+-------------------+-------------------+
```

;;; OR_k ultra-persistency:

```
At least one candidate of a previous Trid-OR5-relation has just been eliminated.
There remains a Trid-OR4-relation between candidates: n6r6c8 n5r7c1 n5r7c9 n5r8c8

hidden-pairs-in-a-row: r8{n4 n7}{c5 c6} ==> r8c6≠8, r8c6≠3, r8c5≠5
```

```
+-------------------+-------------------+-------------------+
! 1238   2368   238   ! 4      156    56    ! 7      2368   9      !
! 1234   5      7     ! 16     8      9     ! 123    2346   2346   !
! 148    68     9     ! 7      3      2     ! 18     4568   4568   !
+-------------------+-------------------+-------------------+
! 238    4      5     ! 9      267    3678  ! 238    1      23678  !
! 6      7      238   ! 1238   124    348   ! 5      9      238    !
! 9      238    1     ! 5      267    3678  ! 4      2368   23678  !
+-------------------+-------------------+-------------------+
! 2358   9      4     ! 238    25     1     ! 6      7      2358   !
! 2358   238    6     ! 238    47     47    ! 9      2358   1      !
! 7      1      238   ! 2368   9      3568  ! 238    23458  23458  !
+-------------------+-------------------+-------------------+
```

;;; OR_k splitting:

```
Trid-OR4-relation between candidates n6r6c8, n5r7c1, n5r7c9 and n5r8c8
+ same valence for candidates n5r8c8 and n5r7c1 via c-chain[2]: n5r8c8,n5r8c1,n5r7c1
==> Trid-OR4-relation can be split into two Trid-OR3-relations with respective lists
of guardians:
    n6r6c8 n5r7c1 n5r7c9  and n6r6c8 n5r7c9 n5r8c8 .

Trid-OR3-forcing-whip-elim[5]:
   || n6r6c8 -
   || n5r7c9 - partial-whip[1]: c8n5{r9 r3} -
   || n5r7c1 - partial-whip[3]: c5n5{r7 r1} - r1c6{n5 n6} - c2n6{r1 r3} -
 ==> r3c8≠6

Trid-OR3-forcing-whip-elim[5]:
   || n6r6c8 -
   || n5r7c1 - partial-whip[2]: c5n5{r7 r1} - r1c6{n5 n6} -
   || n5r7c9 - partial-whip[2]: c5n5{r7 r1} - r1c6{n5 n6} -
 ==> r6c6≠6

Trid-OR3-forcing-whip-elim[5]:
   || n6r6c8 -
   || n5r7c1 - partial-whip[2]: c5n5{r7 r1} - r1c6{n5 n6} -
   || n5r7c9 - partial-whip[2]: c5n5{r7 r1} - r1c6{n5 n6} -
 ==> r1c8≠6

t-whip[4]: r1n1{c1 c5} - r2c4{n1 n6} - r1n6{c6 c2} - r3c2{n6 .} ==> r1c1≠8
t-whip[4]: c7n1{r3 r2} - r2c4{n1 n6} - r1n6{c6 c2} - r3c2{n6 .} ==> r3c7≠8
naked-single ==> r3c7=1
whip[7]: c4n6{r9 r2} - b3n6{r2c9 r3c9} - r3c2{n6 n8} - b3n8{r3c8 r1c8} - r8n8{c8 c1} -
r8n5{c1 c8} - r3n5{c8 .} ==> r9c4≠8
whip[8]: r5n4{c6 c5} - r5n1{c5 c4} - r2c4{n1 n6} - r1n6{c6 c2} - r3c2{n6 n8} -
b4n8{r6c2 r4c1} - c7n8{r4 r9} - c3n8{r9 .} ==> r5c6≠8
whip[3]: c7n8{r9 r4} - c6n8{r4 r6} - b4n8{r6c2 .} ==> r9c3≠8
whip[4]: c1n5{r8 r7} - b7n8{r7c1 r8c2} - r8c4{n8 n2} - r7c5{n2 .} ==> r8c1≠3
whip[6]: r9n8{c9 c6} - b8n5{r9c6 r7c5} - c1n5{r7 r8} - r8n8{c1 c2} - r6n8{c2 c8} -
b3n8{r1c8 .} ==> r7c9≠8
```

```
whip[7]: c1n5{r8 r7} - r7c5{n5 n2} - c4n2{r9 r5} - c3n2{r5 r1} - c3n8{r1 r5} -
r5c9{n8 n3} - r7c9{n3 .} ==> r8c1≠2
whip[7]: r2c7{n3 n2} - r1c8{n2 n8} - c3n8{r1 r5} - r5c9{n8 n2} - r7c9{n2 n5} -
r7c5{n5 n2} - b5n2{r4c5 .} ==> r2c9≠3
whip[8]: r9c3{n2 n3} - r8c2{n3 n8} - r7n8{c1 c4} - b8n3{r7c4 r8c4} - b8n2{r8c4 r9c4} -
c4n6{r9 r2} - r1n6{c5 c2} - r3c2{n6 .} ==> r7c1≠2
whip[8]: c9n7{r4 r6} - b6n6{r6c9 r6c8} - r6c5{n6 n2} - r7n2{c5 c4} - r7n8{c4 c1} -
r7n3{c1 c9} - b6n3{r4c9 r4c7} - r4c1{n3 .} ==> r4c9≠2
whip[8]: r7n8{c1 c4} - r7n3{c4 c9} - r7n2{c9 c5} - b5n2{r4c5 r5c4} - r8c4{n2 n3} -
b7n3{r8c2 r9c3} - r5c3{n3 n8} - r5c9{n8 .} ==> r7c1≠5
hidden-single-in-a-block ==> r8c1=5
```

```
+-------------------+-------------------+-------------------+
! 123   2368  238   ! 4     156   56    ! 7     238   9     !
! 1234  5     7     ! 16    8     9     ! 23    2346  246   !
! 48    68    9     ! 7     3     2     ! 1     458   4568  !
+-------------------+-------------------+-------------------+
! 238   4     5     ! 9     267   3678  ! 238   1     3678  !
! 6     7     238   ! 1238  124   34    ! 5     9     238   !
! 9     238   1     ! 5     267   378   ! 4     2368  23678 !
+-------------------+-------------------+-------------------+
! 38    9     4     ! 238   25    1     ! 6     7     235   !
! 5     238   6     ! 238   47    47    ! 9     238   1     !
! 7     1     23    ! 236   9     3568  ! 238   23458 23458 !
+-------------------+-------------------+-------------------+
```

;;; Small cascade of ORk ultra-persistency rules, due to the elimination(s) following the Single:

```
At least one candidate of a previous Trid-OR3-relation has just been eliminated.
There remains a Trid-OR2-relation between candidates: n6r6c8 n5r7c9

At least one candidate of a previous Trid-OR3-relation has just been eliminated.
There remains a Trid-OR2-relation between candidates: n6r6c8 n5r7c9

whip[8]: r9c3{n2 n3} - r9c7{n3 n8} - r8c8{n8 n3} - r1c8{n3 n8} - r1c3{n8 n2} -
c1n2{r1 r4} - c7n2{r4 r2} - b3n3{r2c7 .} ==> r9c8≠2
whip[8]: b9n4{r9c9 r9c8} - c8n5{r9 r3} - c9n5{r3 r7} - r7c5{n5 n2} - b5n2{r4c5 r5c4} -
r5c9{n2 n8} - b3n8{r3c9 r1c8} - c3n8{r1 .} ==> r9c9≠3
```

OR2-forcing-gwhip-elim[2]:
|| n6r6c8 -
|| n5r7c9 - partial-gwhip[1]: c9n3{r7 r456} -
==> r6c8≠3

```
whip[8]: b8n3{r9c4 r9c6} - r9c3{n3 n2} - r5c3{n2 n8} - r5c9{n8 n2} - b9n2{r7c9 r8c8} -
r8n3{c8 c2} - r6n3{c2 c9} - r7n3{c9 .} ==> r5c4≠3
whip[1]: c4n3{r9 .} ==> r9c6≠3
g-whip[6]: c9n5{r9 r3} - b3n6{r3c9 r2c789} - c4n6{r2 r9} - r9c6{n6 n8} - c7n8{r9 r4} -
c9n8{r4 .} ==> r9c8≠5
hidden-single-in-a-column ==> r3c8=5
```

```
biv-chain[2]: b3n8{r3c9 r1c8} - c3n8{r1 r5} ==> r5c9≠8
finned-x-wing-in-rows: n8{r7 r5}{c4 c1} ==> r4c1≠8
whip[4]: r5c9{n3 n2} - r7c9{n2 n5} - r7c5{n5 n2} - b5n2{r4c5 .} ==> r6c9≠3
whip[4]: r5c9{n3 n2} - r7c9{n2 n5} - r7c5{n5 n2} - b5n2{r4c5 .} ==> r4c9≠3
t-whip[4]: r5n8{c3 c4} - c6n8{r6 r9} - c7n8{r9 r4} - b6n3{r4c7 .} ==> r5c3≠3
whip[5]: c3n3{r1 r9} - c7n3{r9 r4} - c7n8{r4 r9} - r8c8{n8 n2} - b7n2{r8c2 .} ==>
r1c8≠3
whip[1]: b3n3{r2c8 .} ==> r2c1≠3
whip[5]: r1c8{n8 n2} - r2c7{n2 n3} - r9c7{n3 n2} - r9c3{n2 n3} - c8n3{r9 .} ==> r8c8≠8
whip[1]: b9n8{r9c9 .} ==> r9c6≠8
whip[1]: c6n8{r6 .} ==> r5c4≠8
hidden-single-in-a-row ==> r5c3=8
naked-pairs-in-a-column: c6{r1 r9}{n5 n6} ==> r4c6≠6
whip[1]: b5n6{r6c5 .} ==> r1c5≠6
biv-chain[3]: r1c3{n3 n2} - b7n2{r9c3 r8c2} - r6c2{n2 n3} ==> r1c2≠3
z-chain[3]: r5n2{c5 c9} - b6n3{r5c9 r4c7} - r4c1{n3 .} ==> r4c5≠2
z-chain[4]: r1n8{c8 c2} - r8n8{c2 c4} - r8n2{c4 c2} - c3n2{r9 .} ==> r1c8≠2
naked-single ==> r1c8=8
whip[1]: b3n2{r2c9 .} ==> r2c1≠2
biv-chain[4]: r9n5{c9 c6} - c6n6{r9 r1} - r1c2{n6 n2} - b7n2{r8c2 r9c3} ==> r9c9≠2
biv-chain[4]: r3c9{n4 n6} - b1n6{r3c2 r1c2} - r1c6{n6 n5} - r9n5{c6 c9} ==> r9c9≠4
hidden-single-in-a-block ==> r9c8=4
biv-chain[4]: r4n2{c1 c7} - r2c7{n2 n3} - c8n3{r2 r8} - c2n3{r8 r6} ==> r6c2≠2, r4c1≠3
singles ==> r4c1=2, r6c2=3

OR2-forcing-gwhip-elim[2]:
   || n5r7c9 -
   || n6r6c8 - partial-gwhip[1]: b6n2{r6c8 r456c9} -
 ==> r7c9≠2

whip[1]: r7n2{c5 .} ==> r8c4≠2, r9c4≠2
biv-chain[3]: b7n3{r7c1 r9c3} - r9n2{c3 c7} - r8c8{n2 n3} ==> r7c9≠3
stte
```

14.16 ORk-g-whips

The puzzle is this section (#53056 in the 63,137 list) illustrates the combined use of Trid-ORk-g-whips and ultra-persistency of ORk-relations, with ?*high-Tridagon-salience* again set to TRUE.

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

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

```
.2..567.....18.2.....2.7..52..7.58.1...86..72......65.3.5...1.8.125.8...984...5.7;1093
2;206235 ; SER = 10.4
```

```
Resolution state after Singles and whips[1]:
         +----------------------+----------------------+----------------------+
         ! 148     2       1389   ! 349     5       6       ! 7       13489   349     !
         ! 4567    345679  3679   ! 1       8       349     ! 2       3469    3469    !
         ! 1468    3469    13689  ! 2       349     7       ! 349     134689  5       !
         +----------------------+----------------------+----------------------+
         ! 2       3469    369    ! 7       349     5       ! 8       349     1       !
         ! 145     3459    139    ! 8       6       1349    ! 349     7       2       !
         ! 1478    3479    13789  ! 349     12349   12349   ! 6       5       349     !
         +----------------------+----------------------+----------------------+
         ! 3       67      5      ! 469     2479    249     ! 1       2469    8       !
         ! 67      1       2      ! 5       3479    8       ! 349     3469    3469    !
         ! 9       8       4      ! 36      123     123     ! 5       236     7       !
         +----------------------+----------------------+----------------------+
         171 candidates.
```

```
hidden-pairs-in-a-column: c8{n1 n8}{r1 r3} ==> r3c8≠9, r3c8≠6, r3c8≠4, r3c8≠3, r1c8≠9,
r1c8≠4, r1c8≠3
whip[1]: r3n6{c3 .} ==> r2c1≠6, r2c2≠6, r2c3≠6
130 g-candidates, 545 csp-glinks and 326 non-csp glinks
```

```
              +--------------------+--------------------+--------------------+
              ! 148    2      1389  ! 349    5      6     ! 7      18     349    !
              ! 457    34579  379   ! 1      8      349   ! 2      3469   3469   !
              ! 1468   3469   13689 ! 2      349    7     ! 349    18     5      !
              +--------------------+--------------------+--------------------+
              ! 2      3469   369   ! 7      349    5     ! 8      349    1      !
              ! 145    3459   139   ! 8      6      1349  ! 349    7      2      !
              ! 1478   3479   13789 ! 349    12349  12349 ! 6      5      349    !
              +--------------------+--------------------+--------------------+
              ! 3      67     5     ! 469    2479   249   ! 1      2469   8      !
              ! 67     1      2     ! 5      3479   8     ! 349    3469   3469   !
              ! 9      8      4     ! 36     123    123   ! 5      236    7      !
              +--------------------+--------------------+--------------------+
```

```
OR2-anti-tridagon[12] for digits 3, 4 and 9 in blocks:
        b2, with cells: r1c4, r2c6, r3c5
        b3, with cells: r1c9, r2c8, r3c7
        b5, with cells: r6c4, r5c6, r4c5
        b6, with cells: r6c9, r5c7, r4c8
with 2 guardians: n6r2c8 n1r5c6

Trid-OR2-whip[5]: r8c1{n7 n6} - c9n6{r8 r2} - OR2{{n6r2c8 | n1r5c6}} - b4n1{r5c1 r6c3}
- r6n8{c3 .} ==> r6c1≠7
biv-chain[4]: c1n6{r3 r8} - b7n7{r8c1 r7c2} - r6n7{c2 c3} - b4n8{r6c3 r6c1} ==> r3c1≠8
```

```
Trid-OR2-ctr-whip[6]: c5n1{r9 r6} - c5n2{r6 r7} - c5n7{r7 r8} - r8c1{n7 n6} - c9n6{r8
r2} - OR2{{n1r5c6 n6r2c8 | .}} ==> r9c5≠3
Trid-OR2-whip[6]: c1n6{r3 r8} - c9n6{r8 r2} - OR2{{n6r2c8 | n1r5c6}} - b4n1{r5c1 r6c3}
- r6n7{c3 c2} - r7c2{n7 .} ==> r3c1≠1
hidden-pairs-in-a-row: r3{n1 n8}{c3 c8} ==> r3c3≠9, r3c3≠6, r3c3≠3
hidden-single-in-a-column ==> r4c3=6
```

Trid-OR2-gwhip[6]: c3n7{r2 r6} - r6n8{c3 c1} - r6n1{c1 c456} - OR2{{n1r5c6 | n6r2c8}} - b9n6{r7c8 r8c9} - r8c1{n6 .} ==> r2c1≠7

```
singles ==> r8c1=7, r7c2=6, r3c1=6, r9c4=6, r7c5=7
hidden-pairs-in-a-column: c5{n1 n2}{r6 r9} ==> r6c5≠9, r6c5≠4, r6c5≠3
t-whip[5]: r9n3{c6 c8} - r8n3{c9 c5} - r4n3{c5 c2} - r3n3{c2 c7} - r5n3{c7 .} ==>
r6c6≠3, r2c6≠3
whip[5]: c4n3{r6 r1} - r3n3{c5 c7} - r5n3{c7 c6} - r4n3{c5 c8} - r9n3{c8 .} ==> r6c2≠3
Trid-OR2-whip[8]: OR2{{n6r2c8 | n1r5c6}} - c6n3{r5 r9} - r9n1{c6 c5} - b8n2{r9c5 r7c6}
- r7n4{c6 c4} - r1n4{c4 c1} - c1n1{r1 r6} - c1n8{r6 .} ==> r2c8≠4
z-chain[3]: b3n4{r2c9 r3c7} - c5n4{r3 r4} - c8n4{r4 .} ==> r8c9≠4
g-whip[8]: c9n4{r6 r123} - r3n4{c7 c5} - b2n3{r3c5 r1c4} - r6c4{n3 n9} - r4c5{n9 n3} -
r4c2{n3 n9} - b6n9{r4c8 r5c7} - r3n9{c7 .} ==> r6c2≠4
whip[7]: r8n9{c9 c5} - r4n9{c5 c2} - r3n9{c2 c7} - r5n9{c7 c6} - r2n9{c6 c3} - r2n7{c3
c2} - r6c2{n7 .} ==> r7c8≠9
whip[1]: b9n9{r8c9 .} ==> r8c5≠9
biv-chain[3]: b2n3{r1c4 r3c5} - r8c5{n3 n4} - r7c4{n4 n9} ==> r1c4≠9
t-whip[6]: b2n3{r1c4 r3c5} - r8c5{n3 n4} - b9n4{r8c7 r7c8} - r4n4{c8 c2} - r3c2{n4 n9}
- r1n9{c3 .} ==> r1c9≠3
t-whip[6]: c6n3{r5 r9} - r8c5{n3 n4} - b9n4{r8c7 r7c8} - r4n4{c8 c2} - r3n4{c2 c7} -
r5n4{c7 .} ==> r5c6≠9, r5c6≠1
```

```
+-------------------+-------------------+-------------------+
! 148   2     1389  ! 34    5     6     ! 7     18    49    !
! 45    34579 379   ! 1     8     49    ! 2     369   3469  !
! 6     349   18    ! 2     349   7     ! 349   18    5     !
+-------------------+-------------------+-------------------+
! 2     349   6     ! 7     349   5     ! 8     349   1     !
! 145   3459  139   ! 8     6     34    ! 349   7     2     !
! 148   79    13789 ! 349   12    1249  ! 6     5     349   !
+-------------------+-------------------+-------------------+
! 3     6     5     ! 49    7     249   ! 1     24    8     !
! 7     1     2     ! 5     34    8     ! 349   3469  369   !
! 9     8     4     ! 6     12    123   ! 5     23    7     !
+-------------------+-------------------+-------------------+
```

```
At least one candidate of a previous Trid-OR2-relation has just been eliminated.
There remains a Trid-OR1-relation between candidates: n6r2c8
```

Trid-ORk-relation with only one candidate => r2c8=6

```
hidden-single-in-a-column ==> r8c9=6
whip[1]: b5n1{r6c6 .} ==> r6c1≠1, r6c3≠1
hidden-pairs-in-a-row: r6{n1 n2}{c5 c6} ==> r6c6≠9, r6c6≠4
biv-chain[3]: r5c6{n3 n4} - r2c6{n4 n9} - c5n9{r3 r4} ==> r4c5≠3
biv-chain[2]: c9n3{r2 r6} - r4n3{c8 c2} ==> r2c2≠3
biv-chain[2]: r1n3{c3 c4} - b5n3{r6c4 r5c6} ==> r5c3≠3
```

```
biv-chain[3]: r8c5{n3 n4} - r4c5{n4 n9} - c8n9{r4 r8} ==> r8c8≠3
finned-x-wing-in-columns: n3{c6 c8}{r9 r5} ==> r5c7≠3
x-wing-in-columns: n3{c5 c7}{r3 r8} ==> r3c2≠3
whip[1]: c2n3{r5 .} ==> r6c3≠3
biv-chain[3]: r5c7{n9 n4} - r5c6{n4 n3} - r6n3{c4 c9} ==> r6c9≠9
whip[1]: c9n9{r2 .} ==> r3c7≠9
biv-chain[2]: b5n9{r6c4 r4c5} - r3n9{c5 c2} ==> r6c2≠9
stte
```

14.17 Using OR$_k$-splitting rules

The puzzle in this section (#1182 in the 63,137 collection) illustrates the combined use of the ultra-persistency of OR$_k$-relations (defined in section 3.5.6) and of the OR$_k$ splitting rules (defined in section 3.5.7). Again, ?*high-Tridagon-salience* is set to True. Due to this latter choice, it has two anti-tridagons almost at the start, with large numbers of guardians. In what follows, I've kept all the intermediate OR$_k$-relations produced by:

– either ***OR$_k$-reduction*** (due to the elimination of one of the guardians), the concrete implementation of ***OR$_k$ ultra-persistency***;

– or ***OR$_k$-splitting*** (due to the existence of an even c-chain between two guardians, proving that they have the same truth value).

In a normal publication, the useless ones should be discarded. (Note that this can't be done automatically in a no-look-ahead solver, because it can't know in advance what will be useful later.) But the purpose here is to show how these rules work.

```
+-------+-------+-------+
! . 2 . ! 4 . . ! . . . !
! . . 7 ! . . . ! . . 6 !
! 6 . 8 ! . . . ! . 1 5 !
+-------+-------+-------+
! . . . ! 5 . 4 ! . 6 1 !
! . . . ! . 9 . ! . . 2 !
! . 6 . ! . 1 2 ! . 9 . !
+-------+-------+-------+
! . . . ! . . 5 ! 1 . . !
! 5 . . ! . . . ! . 2 4 !
! 9 . 1 ! 2 4 . ! . . . !
+-------+-------+-------+
.2.4.......7.....66.8....15...5.4.61....9...2.6..12.9......51..5......249.124....;236;
29694; SER = 11.7

Resolution state after Singles (and whips[1]):

+----------------------+----------------------+----------------------+
! 13     2      359    ! 4      35678  136789 ! 3789   378    3789   !
! 134    13459  7      ! 1389   2358   1389   ! 23489  348    6      !
! 6      349    8      ! 379    237    379    ! 23479  1      5      !
```

```
+----------------------+----------------------+----------------------+
! 2378   3789   239    ! 5      378    4      ! 378    6      1      !
! 13478  134578 345    ! 3678   9      3678   ! 34578  34578  2      !
! 3478   6      345    ! 378    1      2      ! 34578  9      378    !
+----------------------+----------------------+----------------------+
! 23478  3478   2346   ! 36789  3678   5      ! 1      378    3789   !
! 5      378    36     ! 136789 3678   136789 ! 36789  2      4      !
! 9      378    1      ! 2      4      3678   ! 35678  3578   378    !
+----------------------+----------------------+----------------------+
213 candidates.
```

```
hidden-pairs-in-a-column: c2{n1 n5}{r2 r5} ==> r5c2≠8, r5c2≠7, r5c2≠4, r5c2≠3, r2c2≠9,
r2c2≠4, r2c2≠3
+----------------------+----------------------+----------------------+
! 13     2      359    ! 4      35678  136789 ! 3789   378    3789   !
! 134    15     7      ! 1389   2358   1389   ! 23489  348    6      !
! 6      349    8      ! 379    237    379    ! 23479  1      5      !
+----------------------+----------------------+----------------------+
! 2378   3789   239    ! 5      378    4      ! 378    6      1      !
! 13478  15     345    ! 3678   9      3678   ! 34578  34578  2      !
! 3478   6      345    ! 378    1      2      ! 34578  9      378    !
+----------------------+----------------------+----------------------+
! 23478  3478   2346   ! 36789  3678   5      ! 1      378    3789   !
! 5      378    36     ! 136789 3678   136789 ! 36789  2      4      !
! 9      378    1      ! 2      4      3678   ! 35678  3578   378    !
+----------------------+----------------------+----------------------+
```

The first anti-tridagon, with the splitting rules immediately applied twice to it:

```
OR9-anti-tridagon[12] for digits 7, 8 and 3 in blocks:
        b5, with cells: r4c5, r5c6, r6c4
        b6, with cells: r4c7, r5c8, r6c9
        b8, with cells: r8c5, r9c6, r7c4
        b9, with cells: r8c7, r9c9, r7c8
with 9 guardians: n6r5c6 n4r5c8 n5r5c8 n6r7c4 n9r7c4 n6r8c5 n6r8c7 n9r8c7 n6r9c6

Trid-OR9-relation between candidates n6r5c6, n4r5c8, n5r5c8, n6r7c4, n9r7c4, n6r8c5,
n6r8c7, n9r8c7 and n6r9c6
+ same valence for candidates n6r9c6 and n6r8c7 via c-chain[2]: n6r9c6,n6r9c7,n6r8c7
==> Trid-OR9-relation can be split into two Trid-OR8-relations with respective lists
of guardians:
    n6r5c6 n4r5c8 n5r5c8 n6r7c4 n9r7c4 n6r8c5 n6r8c7 n9r8c7
    and n6r5c6 n4r5c8 n5r5c8 n6r7c4 n9r7c4 n6r8c5 n9r8c7 n6r9c6 .

Trid-OR8-relation between candidates n6r5c6, n4r5c8, n5r5c8, n6r7c4, n9r7c4, n6r8c5,
n9r8c7 and n6r9c6
+ same valence for candidates n9r8c7 and n9r7c4 via c-chain[2]: n9r8c7,n9r7c9,n9r7c4
==> Trid-OR8-relation can be split into two Trid-OR7-relations with respective lists
of guardians:
    n6r5c6 n4r5c8 n5r5c8 n6r7c4 n9r7c4 n6r8c5 n6r9c6
    and n6r5c6 n4r5c8 n5r5c8 n6r7c4 n6r8c5 n9r8c7 n6r9c6.
```

The second anti-tridagon, where no splitting rules can be applied at this point. Notice that it will not be used for any elimination, so that I could have completely deleted it, but this is what SudoRules finds.

```
OR12-anti-tridagon[12] for digits 7, 8 and 3 in blocks:
        b5, with cells: r4c5, r5c6, r6c4
        b6, with cells: r4c7, r5c8, r6c9
        b8, with cells: r7c5, r9c6, r8c4
        b9, with cells: r7c9, r9c8, r8c7
with 12 guardians: n6r5c6 n4r5c8 n5r5c8 n6r7c5 n9r7c9 n1r8c4 n6r8c4 n9r8c4 n6r8c7
n9r8c7 n6r9c6 n5r9c8
```

Now, a few easy "cleaning" steps:

```
biv-chain[3]: r1c1{n3 n1} - r2c2{n1 n5} - b2n5{r2c5 r1c5} ==> r1c5≠3
biv-chain[4]: r1c1{n3 n1} - r2c2{n1 n5} - b2n5{r2c5 r1c5} - b2n6{r1c5 r1c6} ==> r1c6≠3
biv-chain[4]: r1n1{c1 c6} - b2n6{r1c6 r1c5} - b2n5{r1c5 r2c5} - r2c2{n5 n1} ==> r2c1≠1
biv-chain[3]: r2c1{n3 n4} - r3n4{c2 c7} - b3n2{r3c7 r2c7} ==> r2c7≠3
whip[3]: r3n4{c7 c2} - r2c1{n4 n3} - b2n3{r2c4 .} ==> r3c7≠3
biv-chain[4]: r1n1{c6 c1} - r2c2{n1 n5} - b2n5{r2c5 r1c5} - b2n6{r1c5 r1c6} ==>
r1c6≠7, r1c6≠8, r1c6≠9
biv-chain[3]: r8n1{c4 c6} - r1c6{n1 n6} - b5n6{r5c6 r5c4} ==> r8c4≠6
```

```
+----------------------+----------------------+----------------------+
! 13     2      359    ! 4      5678   16     ! 3789   378    3789   !
! 34     15     7      ! 1389   2358   1389   ! 2489   348    6      !
! 6      349    8      ! 379    237    379    ! 2479   1      5      !
+----------------------+----------------------+----------------------+
! 2378   3789   239    ! 5      378    4      ! 378    6      1      !
! 13478  15     345    ! 3678   9      3678   ! 34578  34578  2      !
! 3478   6      345    ! 378    1      2      ! 34578  9      378    !
+----------------------+----------------------+----------------------+
! 23478  3478   2346   ! 36789  3678   5      ! 1      378    3789   !
! 5      378    36     ! 13789  3678   136789 ! 36789  2      4      !
! 9      378    1      ! 2      4      3678   ! 35678  3578   378    !
+----------------------+----------------------+----------------------+
```

The previous OR12 relation can now be reduced (though this will be useless):

```
At least one candidate of a previous Trid-OR12-relation has just been eliminated.
There remains a Trid-OR11-relation between candidates: n6r5c6 n4r5c8 n5r5c8 n6r7c5
n9r7c9 n1r8c4 n9r8c4 n6r8c7 n9r8c7 n6r9c6 n5r9c8
```

One of the two OR7 relations found before (from splitting) can now also be split:

```
Trid-OR7-relation between candidates n6r5c6, n4r5c8, n5r5c8, n6r7c4, n9r7c4, n6r8c5
and n6r9c6
+ same valence for candidates n6r7c4 and n6r5c6 via c-chain[2]: n6r7c4,n6r5c4,n6r5c6
==> Trid-OR7-relation can be split into two Trid-OR6-relations with respective lists
of guardians:
    n6r5c6 n4r5c8 n5r5c8 n9r7c4 n6r8c5 n6r9c6
    and n4r5c8 n5r5c8 n6r7c4 n9r7c4 n6r8c5 n6r9c6.
```

Easy cleaning again:

```
z-chain[3]: r1n8{c9 c5} - c5n5{r1 r2} - r2n2{c5 .} ==> r2c7≠8
biv-chain[4]: r1n5{c5 c3} - r2c2{n5 n1} - r1n1{c1 c6} - b2n6{r1c6 r1c5} ==> r1c5≠7,
r1c5≠8
whip[1]: r1n8{c9 .} ==> r2c8≠8
whip[1]: r1n7{c9 .} ==> r3c7≠7
naked-pairs-in-a-row: r2{c1 c8}{n3 n4} ==> r2c7≠4, r2c6≠3, r2c5≠3, r2c4≠3
whip[1]: b2n3{r3c6 .} ==> r3c2≠3
z-chain[3]: b1n3{r2c1 r1c3} - c3n9{r1 r4} - r4n2{c3 .} ==> r4c1≠3
z-chain[3]: c2n3{r9 r4} - r4n9{c2 c3} - c3n2{r4 .} ==> r7c3≠3
biv-chain[4]: r6n5{c7 c3} - r1n5{c3 c5} - b2n6{r1c5 r1c6} - r9n6{c6 c7} ==> r9c7≠5
hidden-single-in-a-block ==> r9c8=5
```

```
+----------------------+----------------------+----------------------+
! 13     2      359    ! 4      56     16     ! 3789   378    3789   !
! 34     15     7      ! 189    258    189    ! 29     34     6      !
! 6      49     8      ! 379    237    379    ! 249    1      5      !
+----------------------+----------------------+----------------------+
! 278    3789   239    ! 5      378    4      ! 378    6      1      !
! 13478  15     345    ! 3678   9      3678   ! 34578  3478   2      !
! 3478   6      345    ! 378    1      2      ! 34578  9      378    !
+----------------------+----------------------+----------------------+
! 23478  3478   246    ! 36789  3678   5      ! 1      378    3789   !
! 5      378    36     ! 13789  3678   136789 ! 36789  2      4      !
! 9      378    1      ! 2      4      3678   ! 3678   5      378    !
+----------------------+----------------------+----------------------+
```

With the eliminations implied by the last Single comes a new series of reductions for the previously found OR_k relations:

```
At least one candidate of a previous Trid-OR6-relation has just been eliminated.
There remains a Trid-OR5-relation between candidates: n6r5c6 n4r5c8 n9r7c4 n6r8c5
n6r9c6
At least one candidate of a previous Trid-OR6-relation has just been eliminated.
There remains a Trid-OR5-relation between candidates: n4r5c8 n6r7c4 n9r7c4 n6r8c5
n6r9c6

At least one candidate of a previous Trid-OR7-relation has just been eliminated.
There remains a Trid-OR6-relation between candidates: n6r5c6 n4r5c8 n6r7c4 n6r8c5
n9r8c7 n6r9c6

At least one candidate of a previous Trid-OR8-relation has just been eliminated.
There remains a Trid-OR7-relation between candidates: n6r5c6 n4r5c8 n6r7c4 n9r7c4
n6r8c5 n6r8c7 n9r8c7
```

Normal resolution resumes, with a new cascade of reductions due to candidate eliminations after the Singles:

```
z-chain[4]: c2n3{r9 r4} - r4n9{c2 c3} - c3n2{r4 r7} - c3n6{r7 .} ==> r8c3≠3
```

```
naked-single ==> r8c3=6
hidden-single-in-a-block ==> r9c7=6

                  +-------------------+-------------------+-------------------+
                  ! 13    2     359   ! 4     56    16    ! 3789  378   3789  !
                  ! 34    15    7     ! 189   258   189   ! 29    34    6     !
                  ! 6     49    8     ! 379   237   379   ! 249   1     5     !
                  +-------------------+-------------------+-------------------+
                  ! 278   3789  239   ! 5     378   4     ! 378   6     1     !
                  ! 13478 15    345   ! 3678  9     3678  ! 34578 3478  2     !
                  ! 3478  6     345   ! 378   1     2     ! 34578 9     378   !
                  +-------------------+-------------------+-------------------+
                  ! 23478 3478  24    ! 36789 3678  5     ! 1     378   3789  !
                  ! 5     378   6     ! 13789 378   13789 ! 3789  2     4     !
                  ! 9     378   1     ! 2     4     378   ! 6     5     378   !
                  +-------------------+-------------------+-------------------+

At least one candidate of a previous Trid-OR5-relation has just been eliminated.
There remains a Trid-OR3-relation between candidates: n6r5c6 n4r5c8 n9r7c4

At least one candidate of a previous Trid-OR5-relation has just been eliminated.
There remains a Trid-OR3-relation between candidates: n4r5c8 n6r7c4 n9r7c4

At least one candidate of a previous Trid-OR6-relation has just been eliminated.
There remains a Trid-OR4-relation between candidates: n6r5c6 n4r5c8 n6r7c4 n9r8c7

At least one candidate of a previous Trid-OR7-relation has just been eliminated.
There remains a Trid-OR5-relation between candidates: n6r5c6 n4r5c8 n6r7c4 n9r7c4
n9r8c7
```

Notice that until now, not much has been done in terms of resolution proper. Only easy resolution rules and their almost obvious consequences on OR_k-relations have been applied – nothing very harmful to a puzzle in T&E(3).

Here comes the OR_k-chains part, using only the first of the two OR3 relations:

```
z-chain[5]: b6n5{r6c7 r5c7} - b6n4{r5c7 r5c8} - r2n4{c8 c1} - r6n4{c1 c3} - r6n5{c3 .}
==> r6c7≠8, r6c7≠7, r6c7≠3<Fact-48574>
whip[6]: c4n6{r5 r7} - c5n6{r7 r1} - r1n5{c5 c3} - r5c3{n5 n4} - r6n4{c3 c7} -
r6n5{c7 .} ==> r5c4≠3
Trid-OR3-whip[6]: r3c2{n9 n4} - r2c1{n4 n3} - r2c8{n3 n4} - OR3{{n4r5c8 n9r7c4 |
n6r5c6}} - r1c6{n6 n1} - r1c1{n1 .} ==> r3c4≠9
Trid-OR3-whip[6]: b1n4{r2c1 r3c2} - b1n9{r3c2 r1c3} - c9n9{r1 r7} - OR3{{n9r7c4 n4r5c8
| n6r5c6}} - r1c6{n6 n1} - c1n1{r1 .} ==> r5c1≠4
Trid-OR3-whip[6]: r3n4{c7 c2} - b1n9{r3c2 r1c3} - c9n9{r1 r7} - OR3{{n9r7c4 n4r5c8 |
n6r5c6}} - r1n6{c6 c5} - r1n5{c5 .} ==> r5c7≠4
finned-x-wing-in-rows: n4{r2 r5}{c8 c1} ==> r6c1≠4
whip[1]: b4n4{r6c3 .} ==> r7c3≠4
singles ==> r7c3=2, r4c1=2
hidden-pairs-in-a-row: r6{n4 n5}{c3 c7} ==> r6c3≠3
```

```
Trid-OR3-whip[6]: r3n4{c7 c2} - b1n9{r3c2 r1c3} - c9n9{r1 r7} - OR3{{n9r7c4 n4r5c8 |
n6r5c6}} - r1n6{c6 c5} - r1n5{c5 .} ==> r6c7≠4
```

The end is routine solving in W6:

```
singles ==> r6c7=5, r6c3=4, r5c8=4, r2c8=3, r2c1=4, r3c2=9, r4c3=9, r7c2=4, r3c7=4,
r2c7=2, r3c5=2
z-chain[5]: c5n6{r7 r1} - r1c6{n6 n1} - r1c1{n1 n3} - r7c1{n3 n7} - r7c8{n7 .} ==>
r7c5≠8
z-chain[5]: c5n6{r7 r1} - r1c6{n6 n1} - r1c1{n1 n3} - r7c1{n3 n8} - r7c8{n8 .} ==>
r7c5≠7
z-chain[4]: r7c5{n3 n6} - r1n6{c5 c6} - c6n1{r1 r2} - c6n9{r2 .} ==> r8c6≠3
z-chain[5]: r3c4{n7 n3} - r6c4{n3 n8} - r4c5{n8 n3} - r7c5{n3 n6} - c4n6{r7 .} ==>
r5c4≠7
biv-chain[4]: r5c4{n8 n6} - r7n6{c4 c5} - r1c5{n6 n5} - r2c5{n5 n8} ==> r4c5≠8, r2c4≠8
biv-chain[3]: r8n1{c6 c4} - r2c4{n1 n9} - c6n9{r2 r8} ==> r8c6≠7, r8c6≠8
t-whip[3]: r4n8{c7 c2} - c1n8{r6 r7} - c8n8{r7 .} ==> r1c7≠8
biv-chain[3]: c8n7{r7 r1} - r1n8{c8 c9} - c9n9{r1 r7} ==> r7c9≠7
whip[6]: r4n8{c7 c2} - r9n8{c2 c6} - r5n8{c6 c4} - r5n6{c4 c6} - c6n7{r5 r3} -
c6n3{r3 .} ==> r6c9≠8
whip[1]: b6n8{r5c7 .} ==> r8c7≠8
t-whip[4]: r4n8{c7 c2} - r6n8{c1 c4} - r8n8{c4 c5} - c5n7{r8 .} ==> r4c7≠7
z-chain[3]: r4n7{c5 c2} - r9n7{c2 c9} - b6n7{r6c9 .} ==> r5c6≠7
z-chain[3]: c5n3{r8 r4} - b5n7{r4c5 r6c4} - r3c4{n7 .} ==> r8c4≠3, r7c4≠3
whip[4]: b5n7{r4c5 r6c4} - r6c9{n7 n3} - r7n3{c9 c1} - c2n3{r8 .} ==> r4c5≠3
naked-single ==> r4c5=7
whip[1]: c2n7{r9 .} ==> r7c1≠7
whip[1]: c5n3{r8 .} ==> r9c6≠3
finned-x-wing-in-rows: n3{r9 r4}{c2 c9} ==> r6c9≠3
singles ==> r6c9=7, r5c1=7, r5c2=1, r2c2=5, r1c3=3, r1c1=1, r1c6=6, r1c5=5, r5c3=5,
r2c5=8, r8c5=3, r7c5=6, r5c4=6
finned-x-wing-in-rows: n8{r8 r6}{c4 c2} ==> r4c2≠8
stte
```

Note 1: with the introduction of OR_k-splitting rules comes a new control variable in the config file (?*allow-ORk-splitting*), allowing to choose if you want to use these rules or not.

Note 2: without the OR_k-splitting rules, the puzzle can still be solved, but in W7+OR5W8 (instead of W6+OR3W6).

Note 3: the ultra-persistency and the splitting rules have high priority, because one cannot know in advance when the newly OR_k-relations they generate may be useful. In the present puzzle, they could have been postponed until after W4, but there can be no general reason to postpone them.

14.18 The part of the configuration file for OR_k-chains

This section gives all the necessary explanations about the part of the SudoRules configuration file not described in chapter 5, specific to all the rules related to the anti-tridagon pattern. You will see that it is extremely versatile.

It may be useful at this point to repeat that:

- activating the anti-tridagons will compute the Sudoku-specific OR_k-relations (in a format compatible with OR_k-chains);
- activating some of the OR_k-chains will allow these generic chains to use these application-specific OR_k-relations;
- the activation of any type of OR_k-chain CANNOT automatically activate anti-tridagons, because one cannot presume that these generic chains are not intended to be used with a different application-specific OR_k-relation; this activation must therefore be done explicitly.

14.18.1 General anti-tridagon settings

In the original default settings, the basic Tridagon elimination rule and the general anti-tridagon pattern detection rule had low priority (because they rely on 12 CSP-Variables); they were applied only after any ordinary chain of length < 12 (or any ordinary chain of length < ?*all-chains-max-lengths*). This had the advantage of avoiding the detection of anti-tridagons that will not be usable by SudoRules because they have too large sets of guardians. See the tables in section 14.12 for getting an idea of the return on investment when the number of guardians increases.

Alternatively, one can acknowledge the anti-tridagon pattern as an exotic one and a brittle one (it will not be detected if one of the 123-candidates is deleted) – and that it should therefore be applied as soon as possible. One can also argue that mith's pre-filter (three digits and a block such that the three digits are not decided in any of the other blocks, see section 14.8.1) is the essential criterion, the rest being mere verification. In any case, usage proved this choice to be more effective and therefore, in SudoRules default behaviour, global variable ***?*use-high-Tridagon-salience**** is now set to TRUE. The two basic tridagon rules (Tridagon elimination and anti-tridagon detection) are available immediately after Subsets[3] – not earlier, because, when the anti-tridagon pattern is present, Subsets[3] generally allow some significant cleaning of the resolution state.

The configuration file allows to set ?*use-high-Tridagon-salience* to FALSE. With this old setting (FALSE), in mith's database of 63,137 min-expand puzzles in T&E(3), no case has appeared where the anti-tridagon pattern would have been destroyed before being detected.

As previously mentioned, there's also a control variable ?*allow-ORk-splitting* to decide if you want to use splitting or not. Default is FALSE.

Finally, in the current default settings, OR_k-forcing-whips[n] have higher priority than OR_k-contrad-whips[n] and OR_k-whips[n] (and OR_k-forcing-g-whips[n] have higher priority than OR_k-contrad-g-whips[n] and OR_k-g-whips[n]). I think there's no real reason to prefer one over the other. It is therefore left to personal choices: you can change this default behaviour by setting global variable ***?*ORk-Forcing-Whips-before-ORk-Whips**** to FALSE. This variable will also change the way the lchains of ength 1 will be displayed: either as OR_k-forcing-whips[1] or as OR_k-whips[1] – which are fundamentally the "same thing" viewed in different ways. For instance, in the example of section 14.9.5,

```
Trid-OR5-whip[1]:
   || n5r1c5
   || n5r2c6
   || n5r3c4
   || n5r3c8
   || n5r6c6
==> r3c6≠5
```

could as well appear as follows, if OR_k-whips were active (which was not the case in the above section) and of higher priority than OR_k-forcing-whips:

```
Trid-OR5-whip[1]: OR5{{n5r6c6 n5r3c8 n5r3c4 n5r2c6 n5r1c5 | .}} ==> r3c6≠5
```

14.18.2 The part of the configuration file for anti-tridagons

What's shown below is a typical choice of activations: all the types of OR_k-chains are chosen, with k=5 and with all the lengths (of general chains and of ORk-chains) restricted to 8. Notice that, by default, the configuration file has no rule activated at this level. I've added "<<<<<<<<" signs where choices are made here which are not in the default config.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2.3 Sudoku-specific rules : Tridagons and patterns for puzzles in T&E(3)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; BEWARE: in this section 2.3, general rules (Subsets, chains... are supposed to be
activated in section 2.2)
;;; BEWARE: It is strongly recommended not to choose any ORk-chains for k > 6.

;;; 2.3.0) General settings:
;;; By default, the Tridagon elimination rule and the anti-tridagon detection rule
have high priority, alowing their early use (i.e. they will be available immediately
after Subsets[3]).
;;; Give them lower priority (as in the original settings) here:
; (bind ?*use-high-Tridagon-salience* FALSE)
;;; Choose if you want ORk-forcing-whips to be applied before or after ORk-whips
```

```
;;; (and simultaneaously ORk-forcing-g-whips before or after ORk-g-whips).
;;; Default is before (i.e. ?*ORk-Forcing-Whips-before-ORk-Whips* = TRUE); change it
here:
; (bind ?*ORk-Forcing-Whips-before-ORk-Whips* FALSE)

;;; Allow the splitting of ORk relations via conjugacy-chains (FALSE by default):
 (bind ?*allow-ORk-splitting* TRUE) ;;; <<<<<<<<

;;; Use any of the rules in this section, possibly with chains defined in the generic
part, with restricted lengths:
 (bind ?*all-chains-max-length* 8) ;;; <<<<<<<<

;;; 2.3.1) Use the simplest Tridagon elimination rule:
 (bind ?*Tridagons* TRUE) ;;; <<<<<<<<
```

The following is essential. If you want to use any of the OR_k-chain rules, do not forget to explicitly activate anti-tridagons that will feed them with OR_k-relations:

```
;;; IN ALL THE CASES BELOW (ORk-chains and eleven replacement: 2.3.2a, 2.3.2b, 2.3.3,
2.3.4, 2.3.5):
;;; - the anti-tridagons pattern detection rule must be selected explicitly;
;;;   it then automatically implies Tridagons;
;;; - it is highly recommended to restrict the max length of the basic chains rules,
;;;   and that must be done explicitly (it is NOT a consequence of the ORk chain
rules).  See 2.3.0 above.
 (bind ?*Anti-Tridagons* TRUE) ;;; <<<<<<<<
```

Application-specific Tridagon-forcing-whips were defined before I had developed the generic OR2-forcing-whips. They are basically the same thing, with a single difference: instead of being counted as 1, the Tridagon part is counted as 12 in the global length. With usage, this proves to be much too constraining for an efficient use of the Tridagon OR_k-relations. This is why these chains are “deprecated”, meaning they could merely be discarded in a future release.

```
;;; 2.3.2a - deprecated) Use Tridagon-Forcing-Whips (based on Tridagon-links):
;;; (Remember that Tridagon-Forcing-Whips => Tridagons)
; (bind ?*Tridagon-Forcing-Whips* TRUE)
;;; If you use Tridagon-Forcing-Whips,
;;; it is highly recommended to put a strict upper bound also on the lengths of all
the forcing-chains;
;;; 15 is a good starting points;
;;; try to increase these lengths progressively.
; (bind ?*tridagon-forcing-whips-max-length* 12)
```

Now, for each of the three types of OR_k-chains, you can choose very finely which of them you want to use and up to which length. You can make either global or detailed choices for their lengths.

See section 5.4.3 for the implications that will be applied to your choices. Moreover, before applying all these implications, if ?*all-ORk-chains-max-length* is defined, it overrides any other restrictions on any OR_k length. And, if ?*all-ORk-forcing-whips-max-length* is defined, it overrides any definition of the individual values of ?*OR2-forcing-whips-max-length* … ?*OR6-forcing-whips-max-length*. Similarly, if ?*all-ORk-contrad-whips-max-length* is defined, it overrides any definition of the individual values of ?*OR2-contrad-whips-max-length* … ?*OR6-contrad-whips-max-length*. And if ?*all-ORk-whips-max-length* is defined, it overrides any definition of the individual values of ?*OR2-whips-max-length* … ?*OR6-whips-max-length*. Similar rules apply to the OR_k-g-chains.

```
;;; 2.3.2b) Use ORk-Forcing-Whips in combination with Anti-Tridagons:
; (bind ?*OR2-Forcing-Whips* True)
; (bind ?*OR3-Forcing-Whips* True)
; (bind ?*OR4-Forcing-Whips* True)
  (bind ?*OR5-Forcing-Whips* True) ;;; <<<<<<<<
; (bind ?*OR6-Forcing-Whips* True)

;;; 2.3.3) Use ORk-Contrad-Whips in combination with Anti-Tridagons:
; (bind ?*OR2-Contrad-Whips* True)
; (bind ?*OR3-Contrad-Whips* True)
; (bind ?*OR4-Contrad-Whips* True)
  (bind ?*OR5-Contrad-Whips* True) ;;; <<<<<<<<
; (bind ?*OR6-Contrad-Whips* True)

;;; 2.3.4) Use ORk-Whips in combination with Anti-Tridagons:
;;; (Remember that ORk-Whips[n] => ORk-Contrad-Whips[n] => Tridagons)
; (bind ?*OR2-Whips* True)
; (bind ?*OR3-Whips* True)
; (bind ?*OR4-Whips* True)
  (bind ?*OR5-Whips* True) ;;; <<<<<<<<
; (bind ?*OR6-Whips* True)

;;; 2.3.5) Use ORk-Forcing-G-Whips in combination with Anti-Tridagons:
; (bind ?*OR2-Forcing-G-Whips* True)
; (bind ?*OR3-Forcing-G-Whips* True)
; (bind ?*OR4-Forcing-G-Whips* True)
; (bind ?*OR5-Forcing-G-Whips* True)
; (bind ?*OR6-Forcing-G-Whips* True)

;;; 2.3.6) Use ORk-Contrad-G-Whips in combination with Anti-Tridagons:
; (bind ?*OR2-Contrad-G-Whips* True)
; (bind ?*OR3-Contrad-G-Whips* True)
; (bind ?*OR4-Contrad-G-Whips* True)
; (bind ?*OR5-Contrad-G-Whips* True)

;;; 2.3.7) Use ORk-G-Whips in combination with Anti-Tridagons:
; (bind ?*OR2-G-Whips* True)
; (bind ?*OR3-G-Whips* True)
; (bind ?*OR4-G-Whips* True)
```

```
; (bind ?*OR5-G-Whips* True)

;;; 2.3.8) If you use Tridagon-Chains,
;;; it is highly recommended to put a strict upper bound on their lengths;
;;; 5 is a good starting point ; try to increase these lengths progressively.

;;; 2.3.8.a) restrict all the ORk-chains and ORk-g-chains at once;
;;; Notice that this global restriction will prevail on any of the individual
restrictions further below.
 (bind ?*all-ORk-chains-max-length* 8) ;;; <<<<<<<<

;;; 2.3.8.b) restrict all the ORk-chains and ORk-g-chains of each type;
;;; Notice that these semi-global restrictions will prevail on any of the individual
restrictions further below.
; (bind ?*all-ORk-forcing-whips-max-length* 5)
; (bind ?*all-ORk-contrad-whips-max-length* 5)
; (bind ?*all-ORk-whips-max-length* 5)
; (bind ?*all-ORk-contrad-gwhips-max-length* 5)
; (bind ?*all-ORk-gwhips-max-length* 5)

;;; 2.3.8.c) restrict each ORk-chain and ORk-g-chain max lengths individually;
;;; notice that consistency preserving rules will be applied.
; (bind ?*OR2-forcing-whips-max-length* 5)
; (bind ?*OR3-forcing-whips-max-length* 5)
; (bind ?*OR4-forcing-whips-max-length* 5)
; (bind ?*OR5-forcing-whips-max-length* 5)
; (bind ?*OR6-forcing-whips-max-length* 5)

; (bind ?*OR2-contrad-whips-max-length* 5)
; (bind ?*OR3-contrad-whips-max-length* 5)
; (bind ?*OR4-contrad-whips-max-length* 5)
; (bind ?*OR5-contrad-whips-max-length* 5)
; (bind ?*OR6-contrad-whips-max-length* 5)

; (bind ?*OR2-whips-max-length* 5)
; (bind ?*OR3-whips-max-length* 5)
; (bind ?*OR4-whips-max-length* 5)
; (bind ?*OR5-whips-max-length* 5)
; (bind ?*OR6-whips-max-length* 5)

; (bind ?*OR2-contrad-gwhips-max-length* 5)
; (bind ?*OR3-contrad-gwhips-max-length* 5)
; (bind ?*OR4-contrad-gwhips-max-length* 5)
; (bind ?*OR5-contrad-gwhips-max-length* 5)

; (bind ?*OR2-gwhips-max-length* 5)
; (bind ?*OR3-gwhips-max-length* 5)
; (bind ?*OR4-gwhips-max-length* 5)
; (bind ?*OR5-gwhips-max-length* 5)


;;; 2.3.9) Eleven's replacement technique:
```

```
;;; Allow the automatic use of eleven's replacement method based on tridagons.
;;; (Note that the method is much more general;
;;; here, the anti-tridaon structure is only used to define a starting point).
;;; The method will be applied only when no other rule is applicable.
; (bind ?*Anti-Tridagons* TRUE)
; (bind ?*Eleven-Replacement-in-Tridagons* TRUE)

;;; If you use Eleven-Replacement-in-Tridagons,
;;; it is highly recommended to put a strict upper bound on the lengths of all the
chains and ORk-chains;
;;; see above
```

14.18.3 Example with an OR$_k$-whip and with high Tridagons salience

Here is an example where setting ?*use-high-Tridagon-salience* to TRUE provides a simpler solution. The puzzle is #5 in mith's list of 63,137 min-expands.

```
+-------+-------+-------+
! . . . ! 4 . . ! . . . !
! 4 . . ! . 8 9 ! . . . !
! 6 8 . ! 3 7 . ! . 4 . !
+-------+-------+-------+
! . 6 8 ! . 4 7 ! 9 . . !
! 7 3 . ! 9 6 . ! 4 . . !
! 9 . 4 ! 8 . 3 ! 6 . . !
+-------+-------+-------+
! 3 . . ! . . . ! . 5 2 !
! . . . ! . 3 . ! . 9 . !
! 8 7 . ! . . . ! 3 . . !
+-------+-------+-------+
...4.....4...89...68.37..4..68.479..73.96.4..9.48.36..3......52....3..
9.87....3..;3;396
```

```
Resolution state after Singles and whips[1]:

+---------------------+---------------------+---------------------+
! 125    1259   123579 ! 4      125    1256   ! 12578  13678  136789 !
! 4      125    12357  ! 1256   8      9      ! 1257   1367   1367   !
! 6      8      1259   ! 3      7      125    ! 125    4      19     !
+---------------------+---------------------+---------------------+
! 125    6      8      ! 125    4      7      ! 9      123    135    !
! 7      3      125    ! 9      6      125    ! 4      128    158    !
! 9      125    4      ! 8      125    3      ! 6      127    157    !
+---------------------+---------------------+---------------------+
! 3      149    169    ! 167    19     1468   ! 178    5      2      !
! 125    1245   1256   ! 12567  3      124568 ! 178    9      14678  !
! 8      7      12569  ! 1256   1259   12456  ! 3      16     146    !
+---------------------+---------------------+---------------------+
179 candidates.
```

The puzzle can be solved using only whips and OR2-whips. If one sets ?*high-Tridagon-salience* to FALSE, a whip[7] will appear before the tridagon-OR2-relation can be detected. However, with ?*high-Tridagon-salience* set to TRUE, this step is not necessary and the puzzle can be solved using (almost) only anti-tridagon chain rules no longer than 5:

```
hidden-pairs-in-a-column: c3{n3 n7}{r1 r2} ==> r2c3≠5, r2c3≠2, r2c3≠1, r1c3≠9, r1c3≠5,
r1c3≠2, r1c3≠1

+----------------------+----------------------+----------------------+
! 125    1259   37     ! 4      125    1256   ! 12578  13678  136789 !
! 4      125    37     ! 1256   8      9      ! 1257   1367   1367   !
! 6      8      1259   ! 3      7      125    ! 125    4      19     !
+----------------------+----------------------+----------------------+
! 125    6      8      ! 125    4      7      ! 9      123    135    !
! 7      3      125    ! 9      6      125    ! 4      128    158    !
! 9      125    4      ! 8      125    3      ! 6      127    157    !
+----------------------+----------------------+----------------------+
! 3      149    169    ! 167    19     1468   ! 178    5      2      !
! 125    1245   1256   ! 12567  3      124568 ! 178    9      14678  !
! 8      7      12569  ! 1256   1259   12456  ! 3      16     146    !
+----------------------+----------------------+----------------------+

OR2-anti-tridagon[12] for digits 2, 5 and 1 in blocks:
        b1, with cells: r1c1, r2c2, r3c3
        b2, with cells: r1c5, r2c4, r3c6
        b4, with cells: r4c1, r6c2, r5c3
        b5, with cells: r4c4, r6c5, r5c6
with 2 guardians: n6r2c4 n9r3c3

Trid-OR2-whip[4]: r3c9{n1 n9} - OR2{{n9r3c3 | n6r2c4}} - r2n2{c4 c2} - r2n5{c2 .} ==>
r2c7≠1
Trid-OR2-whip[4]: OR2{{n6r2c4 | n9r3c3}} - c2n9{r1 r7} - r7c5{n9 n1} - r7c3{n1 .} ==>
r7c4≠6
Trid-OR2-whip[5]: r1n9{c9 c2} - OR2{{n9r3c3 | n6r2c4}} - r1n6{c6 c8} - r9c8{n6 n1} -
b6n1{r4c8 .} ==> r1c9≠1
Trid-OR2-whip[5]: r3c9{n1 n9} - OR2{{n9r3c3 | n6r2c4}} - r2n1{c4 c2} - r2n2{c2 c7} -
r2n5{c7 .} ==> r1c8≠1, r1c7≠1, r3c7≠1
whip[1]: c7n1{r8 .} ==> r8c9≠1, r9c8≠1, r9c9≠1
singles ==> r9c8=6, r9c9=4
hidden-pairs-in-a-block: b8{n4 n8}{r7c6 r8c6} ==> r8c6≠6, r8c6≠5, r8c6≠2, r8c6≠1,
r7c6≠6, r7c6≠1
singles ==> r8c4=6, r1c6=6, r2c9=6, r7c3=6, r7c4=7
whip[1]: b8n2{r9c6 .} ==> r9c3≠2
whip[1]: b8n5{r9c6 .} ==> r9c3≠5

+----------------+----------------+----------------+
! 125  1259 37   ! 4    125  6    ! 2578 378  3789 !
! 4    125  37   ! 125  8    9    ! 257  137  6    !
! 6    8    1259 ! 3    7    125  ! 25   4    19   !
+----------------+----------------+----------------+
```

```
! 125   6    8    ! 125   4    7    ! 9    123  135  !
! 7     3    125  ! 9     6    125  ! 4    128  158  !
! 9     125  4    ! 8     125  3    ! 6    127  157  !
+-----------------+-----------------+-----------------+
! 3     149  6    ! 7     19   48   ! 18   5    2    !
! 125   1245 125  ! 6     3    48   ! 178  9    78   !
! 8     7    19   ! 125   1259 125  ! 3    6    4    !
+-----------------+-----------------+-----------------+
```

```
tridagon for digits 1, 2 and 5 in blocks:
        b1, with cells: r3c3 (target cell), r1c1, r2c2
        b2, with cells: r3c6, r1c5, r2c4
        b4, with cells: r5c3, r4c1, r6c2
        b5, with cells: r5c6, r4c4, r6c5
 ==> r3c3≠1,2,5

stte
```

14.18.4 Example with an OR$_k$-g-whip and with high Tridagons salience

Here is an example where setting ?*use-high-Tridagon-salience* to TRUE and activating OR$_k$-g-whips provides a short solution. The puzzle is #33,177 in mith's list of 63,137 min-expands.

```
+-------+-------+-------+
! 1 2 . ! . 5 . ! . . 9 !
! . . . ! . . . ! . . . !
! 7 . 9 ! . 3 2 ! . . . !
+-------+-------+-------+
! . . . ! . 7 5 ! 6 . 1 !
! . 7 1 ! . . . ! 9 . . !
! . . . ! . 2 . ! . 7 8 !
+-------+-------+-------+
! 3 1 . ! . . 7 ! . 9 . !
! 5 . 2 ! . 9 3 ! . 1 . !
! . 9 7 ! . 1 . ! . 3 . !
+-------+-------+-------+
12..5...9.........7.9.32.......756.1.71...9......2..7831...7.9.5.2.93.1..97.1..3.;6692
;132339

Resolution state after Singles and whips[1]:
   +----------------------+----------------------+----------------------+
   ! 1      2      3468   ! 4678   5      468    ! 347    468    9      !
   ! 468    34568  34568  ! 146789 468    14689  ! 123457 24568  23457  !
   ! 7      4568   9      ! 1468   3      2      ! 145    4568   45     !
   +----------------------+----------------------+----------------------+
   ! 2489   348    348    ! 3489   7      5      ! 6      24     1      !
   ! 2468   7      1      ! 3468   468    468    ! 9      245    2345   !
   ! 469    3456   3456   ! 13469  2      1469   ! 34     7      8      !
   +----------------------+----------------------+----------------------+
```

```
! 3       1       468    ! 24568  468     7      ! 2458   9       2456   !
! 5       468     2      ! 468    9       3      ! 478    1       467    !
! 468     9       7      ! 24568  1       468    ! 2458   3       2456   !
+----------------------+----------------------+----------------------+
```

```
186 candidates.

hidden-pairs-in-a-column: c4{n2 n5}{r7 r9} ==> r9c4≠8, r9c4≠6, r9c4≠4, r7c4≠8, r7c4≠6,
r7c4≠4
hidden-pairs-in-a-column: c6{n1 n9}{r2 r6} ==> r6c6≠6, r6c6≠4, r2c6≠8, r2c6≠6, r2c6≠4
152 g-candidates, 625 csp-glinks and 373 non-csp glinks
+----------------------+----------------------+----------------------+
! 1       2       3468   ! 4678   5       468    ! 347    468     9      !
! 468     34568   34568  ! 146789 468     19     ! 123457 24568   23457  !
! 7       4568    9      ! 1468   3       2      ! 145    4568    45     !
+----------------------+----------------------+----------------------+
! 2489    348     348    ! 3489   7       5      ! 6      24      1      !
! 2468    7       1      ! 3468   468     468    ! 9      245     2345   !
! 469     3456    3456   ! 13469  2       19     ! 34     7       8      !
+----------------------+----------------------+----------------------+
! 3       1       468    ! 25     468     7      ! 2458   9       2456   !
! 5       468     2      ! 468    9       3      ! 478    1       467    !
! 468     9       7      ! 25     1       468    ! 2458   3       2456   !
+----------------------+----------------------+----------------------+

OR3-anti-tridagon[12] for digits 6, 8 and 4 in blocks:
       b1, with cells: r1c3, r2c1, r3c2
       b2, with cells: r1c6, r2c5, r3c4
       b7, with cells: r7c3, r9c1, r8c2
       b8, with cells: r7c5, r9c6, r8c4
with 3 guardians: n3r1c3 n5r3c2 n1r3c4

z-chain[3]: r4n9{c4 c1} - c1n2{r4 r5} - r5n8{c1 .} ==> r4c4≠8
whip[1]: b5n8{r5c6 .} ==> r5c1≠8
whip[3]: c9n3{r2 r5} - r6c7{n3 n4} - b9n4{r7c7 .} ==> r2c9≠4
g-whip[4]: r3c9{n5 n4} - c8n4{r3 r456} - r6c7{n4 n3} - b3n3{r1c7 .} ==> r2c9≠5
whip[5]: r6c7{n3 n4} - r1c7{n4 n7} - r2n7{c9 c4} - r2n1{c4 c6} - r2n9{c6 .} ==> r2c7≠3
whip[5]: r6c7{n4 n3} - r1c7{n3 n7} - r2n7{c9 c4} - r2n1{c4 c6} - r2n9{c6 .} ==> r2c7≠4
```

Trid-OR3-gwhip[5]: c8n4{r5 r123} - r3c9{n4 n5} - r3c7{n5 n1} - OR3{{n1r3c4 n5r3c2 | n3r1c3}} - c7n3{r1 .} ==> r6c7≠4

```
singles ==> r6c7=3, r2c9=3, r1c3=3, r4c2=3, r5c4=3, r8c9=7
z-chain[2]: r8n6{c4 c2} - b1n6{r3c2 .} ==> r2c4≠6
hidden-triplets-in-a-row: r2{n1 n7 n9}{c6 c7 c4} ==> r2c7≠5, r2c7≠2, r2c4≠8, r2c4≠4
singles ==> r2c8=2, r4c8=4, r4c3=8, r4c4=9, r4c1=2, r6c6=1, r2c6=9, r5c8=5, r5c9=2,
r6c1=9
whip[1]: b3n5{r3c9 .} ==> r3c2≠5
biv-chain[3]: r6c4{n4 n6} - r8n6{c4 c2} - r7c3{n6 n4} ==> r6c3≠4
finned-swordfish-in-rows: n4{r8 r6 r1}{c7 c2 c4} ==> r3c4≠4
finned-swordfish-in-rows: n4{r6 r8 r3}{c2 c4 c7} ==> r1c7≠4
singles ==> r1c7=7, r2c7=1, r2c4=7, r3c4=1
whip[1]: b3n4{r3c9 .} ==> r3c2≠4
whip[1]: b1n4{r2c3 .} ==> r2c5≠4
```

```
biv-chain[3]: b4n4{r6c2 r5c1} - c5n4{r5 r7} - c3n4{r7 r2} ==> r2c2≠4
biv-chain[3]: r5c1{n6 n4} - c2n4{r6 r8} - b7n8{r8c2 r9c1} ==> r9c1≠6
biv-chain[3]: b1n4{r2c3 r2c1} - c1n6{r2 r5} - r6c3{n6 n5} ==> r2c3≠5
singles ==> r2c2=5, r6c3=5
x-wing-in-rows: n6{r6 r8}{c2 c4} ==> r3c2≠6, r1c4≠6
stte
```

14.16 Ultra-persistency of the OR$_k$-relations vs degeneracy

As noticed in section 3.5.6, exotic patterns are often brittle: if some of their defining candidate(s) disappear(s) during resolution, what remains may be much more difficult to spot than the original pattern. This is true in particular of anti-tridagons.

Thanks to mith's additional criterion (section 14.8.1), the full anti-tridagon pattern (with the three defining digits in all of the twelve defining cells, plus possibly other digits in them) is quite easy to find. However, if one allows missing candidates (which would make the corresponding degenerated trivalue-oddagon pattern contradictory in T&E(2)), mith's pre-filter no longer applies and the standard anti-tridagon detection rule no longer works. There's currently nothing in SudoRules to allow finding this degenerated pattern if it is already degenerated at the start of resolution. (I have indeed written a rule to detect degenerated cases, but its complexity is much too high to be useful in any way.)

However, if the degeneracy of an OR$_k$-relation occurs during resolution, CSP-Rules has no problem with this, because the previously found ***OR$_k$-relations are persistent: once proven, they remain TRUE forever***. Moreover, as mentioned in section 3.5.6, ***CSP-Rules has generic rules for making any OR$_k$-relation ultra-persistent***: if a guardian disappears – which makes this OR$_k$-relation unusable for the OR$_k$-chain rules (because this guardian was either the target of a required partial-whip or the end of a required link; and there can be no partial-whip with no target and no link with a missing end) –, an OR$_j$-relation for some j smaller than k will be automatically asserted for the remaining guardians, even if the underlying anti-tridagon pattern is degenerated. The following example shows how this works (it is difficult to find a puzzle where both degeneracy and ultra-persistency appear).

The puzzle is #48950 in mith's list of T&E(3) min-expands. For definiteness, I have set ?*use-high-Tridagon-salience* to TRUE (so that the OR$_k$-relations are found as soon as possible) and ?*ORk-Forcing-Whips-before-ORk-Whips* to FALSE (so that OR$_k$-contrad-whips and OR$_k$-contrad-whips appear before OR$_k$-forcing-whips of same length). Note that OR$_k$-splitting rules are not selected (they would lead to a much simpler solution – see section 14.16.1).

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

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

```
12...678..5....2.6.....2.152.5.7.....1.......87.21..6.5..948...78.36........276.8;1019
0;240260
SER = 10.4
```

```
Resolution state after Singles and whips[1]:

   +----------------------+----------------------+----------------------+
   ! 1      2      349    ! 45     359    6      ! 7      8      349    !
   ! 349    5      34789  ! 1478   389    1349   ! 2      349    6      !
   ! 3469   3469   346789 ! 478    389    2      ! 349    1      5      !
   +----------------------+----------------------+----------------------+
   ! 2      3469   5      ! 468    7      349    ! 13489  349    1349   !
   ! 3469   1      3469   ! 4568   3589   3459   ! 34589  234579 23479  !
   ! 8      7      349    ! 2      1      3459   ! 3459   6      349    !
   +----------------------+----------------------+----------------------+
   ! 5      36     1236   ! 9      4      8      ! 13     237    1237   !
   ! 7      8      1249   ! 3      6      15     ! 1459   2459   1249   !
   ! 349    349    1349   ! 15     2      7      ! 6      3459   8      !
   +----------------------+----------------------+----------------------+
   177 candidates.
```

```
hidden-pairs-in-a-row: r5{n2 n7}{c8 c9} ==> r5c9≠9, r5c9≠4, r5c9≠3, r5c8≠9, r5c8≠5,
r5c8≠4, r5c8≠3
whip[1]: c8n5{r9 .} ==> r8c7≠5
hidden-pairs-in-a-column: c3{n7 n8}{r2 r3} ==> r3c3≠9, r3c3≠6, r3c3≠4, r3c3≠3, r2c3≠9,
r2c3≠4, r2c3≠3
```

Because of our choice of high anti-tridagon salience, the anti-tridagon pattern is found here, after all the chain rules of length ≤ 2 and all the Subset rules of size ≤ 3 have been tried:

```
Resolution state RS1:

   +--------------------+--------------------+--------------------+
   ! 1     2     349    ! 45    359   6      ! 7     8     349    !
   ! 349   5     78     ! 1478  389   1349   ! 2     349   6      !
   ! 3469  3469  78     ! 478   389   2      ! 349   1     5      !
   +--------------------+--------------------+--------------------+
   ! 2     3469  5      ! 468   7     349    ! 13489 349   1349   !
   ! 3469  1     3469   ! 4568  3589  3459   ! 34589 27    27     !
   ! 8     7     349    ! 2     1     3459   ! 3459  6     349    !
   +--------------------+--------------------+--------------------+
```

```
   ! 5     36    1236  ! 9     4     8     ! 13    237   1237  !
   ! 7     8     1249  ! 3     6     15    ! 149   2459  1249  !
   ! 349   349   1349  ! 15    2     7     ! 6     3459  8     !
   +-------------------+-------------------+-------------------+
```

```
OR5-anti-tridagon[12] for digits 3, 4 and 9 in blocks:
        b1, with cells: r1c3, r2c1, r3c2
        b3, with cells: r1c9, r2c8, r3c7
        b4, with cells: r6c3, r5c1, r4c2
        b6, with cells: r6c9, r5c7, r4c8
with 5 guardians: n6r3c2 n6r4c2 n6r5c1 n5r5c7 n8r5c7
```

However, it will not be used in any OR$_k$-chain before other rules of higher salience:

```
biv-chain[3]: r1c4{n4 n5} - b8n5{r9c4 r8c6} - c6n1{r8 r2} ==> r2c6≠4
whip[1]: c6n4{r6 .} ==> r4c4≠4, r5c4≠4
biv-chain[4]: r7c7{n1 n3} - r7c2{n3 n6} - r4n6{c2 c4} - r4n8{c4 c7} ==> r4c7≠1
hidden-single-in-a-block ==> r4c9=1
whip[7]: r3n6{c1 c2} - r7c2{n6 n3} - r9c1{n3 n4} - b1n4{r3c1 r1c3} - r1c4{n4 n5} -
r9n5{c4 c8} - r9n3{c8 .} ==> r3c1≠9
whip[8]: r9n9{c3 c8} - c8n5{r9 r8} - b8n5{r8c6 r9c4} - r1c4{n5 n4} - r1c3{n4 n3} -
r6c3{n3 n4} - c9n4{r6 r8} - r8n2{c9 .} ==> r8c3≠9
whip[1]: r8n9{c9 .} ==> r9c8≠9
whip[8]: r6n5{c7 c6} - b8n5{r8c6 r9c4} - b8n1{r9c4 r8c6} - r8c7{n1 n4} - r9c8{n4 n3} -
r4c8{n3 n4} - b3n4{r2c8 r1c9} - r1c4{n4 .} ==> r6c7≠9
```

Trid-OR5-whip[8]: r7c7{n1 n3} - r7c2{n3 n6} - c3n6{r7 r5} - r4n6{c2 c4} - r4n8{c4 c7} - OR5{{n8r5c7 n6r5c1 n6r4c2 n6r3c2 | n5r5c7}} - b5n5{r5c4 r6c6} - r8c6{n5 .} ==> r8c7≠1

hidden-single-in-a-block ==> r7c7=1

Trid-OR5-whip[7]: c6n5{r6 r8} - r8n1{c6 c3} - c3n2{r8 r7} - c3n6{r7 r5} - c1n6{r5 r3} - OR5{{n6r5c1 n5r5c7 n6r4c2 n6r3c2 | n8r5c7}} - r5c4{n8 .} ==> r5c5≠5

singles ==> r1c5=5, r1c4=4

```
   +-------------------+-------------------+-------------------+
   ! 1     2     39    ! 4     5     6     ! 7     8     39    !
   ! 349   5     78    ! 178   389   139   ! 2     349   6     !
   ! 346   3469  78    ! 78    389   2     ! 349   1     5     !
   +-------------------+-------------------+-------------------+
   ! 2     3469  5     ! 68    7     349   ! 3489  349   1     !
   ! 3469  1     3469  ! 568   389   3459  ! 34589 27    27    !
   ! 8     7     349   ! 2     1     3459  ! 345   6     349   !
   +-------------------+-------------------+-------------------+
   ! 5     36    1236  ! 9     4     8     ! 1     237   237   !
   ! 7     8     124   ! 3     6     15    ! 49    245   249   !
   ! 349   349   1349  ! 15    2     7     ! 6     3459  8     !
   +-------------------+-------------------+-------------------+
```

At this point, the anti-tridagon pattern starts degenerating: it looses candidates n4r1c3 and n4r1c9 (but it still has its 5 original guardians). ***Thanks to ORk-persistency, this will not prevent using it again in the next OR5-chains.***

```
naked-pairs-in-a-row: r3{c3 c4}{n7 n8} ==> r3c5≠8
z-chain[7]: r8c7{n9 n4} - b3n4{r3c7 r2c8} - c8n9{r2 r8} - c8n5{r8 r9} - c4n5{r9 r5} -
c4n6{r5 r4} - r4n8{c4 .} ==> r4c7≠9
whip[7]: r3n6{c1 c2} - r7c2{n6 n3} - r9n3{c3 c8} - r9c1{n3 n9} - c2n9{r9 r4} -
r4c8{n9 n4} - b3n4{r2c8 .} ==> r3c1≠4
```

`Trid-OR5-whip[5]: r3n4{c7 c2} - r3n6{c2 c1} - OR5{{n6r5c1 n8r5c7 n5r5c7 n6r3c2 | n6r4c2}} - r4c4{n6 n8} - c7n8{r4 .} ==> r5c7≠4`

```
whip[6]: r1c3{n9 n3} - r3c1{n3 n6} - b4n6{r5c1 r4c2} - b4n3{r4c2 r5c1} - r5c5{n3 n8} -
r4c4{n8 .} ==> r5c3≠9
```

`Trid-OR5-whip[6]: r8c7{n9 n4} - r3n4{c7 c2} - r3n6{c2 c1} - OR5{{n6r5c1 n8r5c7 n5r5c7 n6r3c2 | n6r4c2}} - r4c4{n6 n8} - c7n8{r4 .} ==> r5c7≠9`

```
biv-chain[3]: b6n9{r4c8 r6c9} - c9n4{r6 r8} - r8c7{n4 n9} ==> r8c8≠9
z-chain[3]: c8n9{r4 r2} - r1c9{n9 n3} - b9n3{r7c9 .} ==> r4c8≠3
whip[6]: r4c8{n9 n4} - r2n4{c8 c1} - c2n4{r3 r9} - c2n9{r9 r3} - r1n9{c3 c9} -
b6n9{r6c9 .} ==> r4c6≠9
whip[6]: c7n8{r4 r5} - c7n5{r5 r6} - c7n3{r6 r3} - r3c5{n3 n9} - r5c5{n9 n3} -
r4c6{n3 .} ==> r4c7≠4
whip[5]: r1n9{c3 c9} - r1n3{c9 c3} - r6c3{n3 n4} - b6n4{r6c9 r4c8} - c8n9{r4 .} ==>
r9c3≠9
whip[5]: r5n4{c3 c6} - r4n4{c6 c8} - c8n9{r4 r2} - c6n9{r2 r6} - b6n9{r6c9 .} ==>
r6c3≠4
naked-pairs-in-a-column: c3{r1 r6}{n3 n9} ==> r9c3≠3, r7c3≠3, r5c3≠3
whip[7]: r3n4{c2 c7} - r2n4{c8 c1} - b1n9{r2c1 r1c3} - b3n9{r1c9 r2c8} - r4n9{c8 c2} -
r9n9{c2 c1} - b7n3{r9c1 .} ==> r3c2≠3
whip[7]: r4c8{n9 n4} - r2n4{c8 c1} - c2n4{r3 r9} - r9c3{n4 n1} - c4n1{r9 r2} -
r2c6{n1 n3} - r4c6{n3 .} ==> r2c8≠9
hidden-single-in-a-column ==> r4c8=9
whip[1]: b6n4{r6c9 .} ==> r6c6≠4
biv-chain[3]: c1n6{r3 r5} - b4n9{r5c1 r6c3} - c3n3{r6 r1} ==> r3c1≠3
naked-single ==> r3c1=6
```

```
+----------------+----------------+----------------+
! 1    2    39   ! 4    5    6    ! 7    8    39   !
! 349  5    78   ! 178  389  139  ! 2    34   6    !
! 6    469  78   ! 78   39   2    ! 349  1    5    !
+----------------+----------------+----------------+
! 2    346  5    ! 68   7    34   ! 38   9    1    !
! 3469 1    46   ! 568  389  3459 ! 358  27   27   !
! 8    7    39   ! 2    1    359  ! 345  6    34   !
+----------------+----------------+----------------+
! 5    36   26   ! 9    4    8    ! 1    237  237  !
! 7    8    124  ! 3    6    15   ! 49   245  249  !
! 349  349  14   ! 15   2    7    ! 6    345  8    !
+----------------+----------------+----------------+
```

At this point, what remains of the original anti-tridagon pattern is a highly degenerated form of it:

– number 3 is missing in r3c2, r4c8

– number 4 is missing in r1c3, r1c9, r6c3, r4c8

– number 9 is missing in r2c8, r4c2, r6c9, r5c7

Ten out of the twelve defining cells have a missing 123-candidate. Actually, cell r4c8 is even decided; i.e. we have the worst possible case of degeneracy. I don't think anyone would be able to find this kind of degenerated pattern, without the knowledge of the preceding non-degenerated form.

The assertion of n6r3c1 implies two applications of ECP that successively delete two guardians; each deletion of a guardian activates the ultra-persistency rules (in spite of the high degeneracy):

```
+----------------+----------------+----------------+
! 1    2    39   ! 4    5    6    ! 7    8    39   !
! 349  5    78   ! 178  389  139  ! 2    34   6    !
! 6    469  78   ! 78   39   2    ! 349  1    5    !
+----------------+----------------+----------------+
! 2    346  5    ! 68   7    34   ! 38   9    1    !
! 349  1    46   ! 568  389  3459 ! 358  27   27   !
! 8    7    39   ! 2    1    359  ! 345  6    34   !
+----------------+----------------+----------------+
! 5    36   26   ! 9    4    8    ! 1    237  237  !
! 7    8    124  ! 3    6    15   ! 49   245  249  !
! 349  349  14   ! 15   2    7    ! 6    345  8    !
+----------------+----------------+----------------+
```

```
At least one candidate of a previous Trid-OR5-relation has just been eliminated.
There remains a Trid-OR4-relation between candidates: n6r3c2 n6r4c2 n5r5c7 n8r5c7

At least one candidate of a previous Trid-OR4-relation has just been eliminated.
There remains a Trid-OR3-relation between candidates: n6r4c2 n5r5c7 n8r5c7
```

`Trid-OR3-whip[3]: OR3{{n8r5c7 n5r5c7 | n6r4c2}} - r4n3{c2 c6} - r4n4{c6 .} ==> r5c7≠3`

```
biv-chain[3]: r5c7{n5 n8} - r4n8{c7 c4} - b5n6{r4c4 r5c4} ==> r5c4≠5
singles ==> r9c4=5, r8c6=1, r2c4=1, r3c4=7, r3c3=8, r2c3=7, r2c5=8, r9c3=1, r8c8=5
naked-pairs-in-a-column: c8{r2 r9}{n3 n4} ==> r7c8≠3
finned-x-wing-in-rows: n9{r2 r5}{c1 c6} ==> r6c6≠9
singles ==> r6c3=9, r1c3=3, r1c9=9, r8c7=9
biv-chain[2]: c8n4{r9 r2} - b1n4{r2c1 r3c2} ==> r9c2≠4
x-wing-in-rows: n4{r2 r9}{c1 c8} ==> r5c1≠4
stte
```

Note: there would still be a solution, but with final steps in W5, if the OR3-whip[3] was not found.

14.16.1 The effect of adding OR$_k$-splitting rules

With OR$_k$-splitting rules, degeneracy of the anti-tridagon pattern is still visible, but the solution becomes much simpler. Starting from resolution state RS1, the OR5 relation can be split twice:

```
Trid-OR5-relation between candidates n6r3c2, n6r4c2, n6r5c1, n5r5c7 and n8r5c7
+ same valence for candidates n6r5c1 and n6r3c2 via c-chain[2]: n6r5c1,n6r3c1,n6r3c2
==> Trid-OR5-relation can be split into two Trid-OR4-relations with respective lists
of guardians:
    n6r3c2 n6r4c2 n5r5c7 n8r5c7  and n6r4c2 n6r5c1 n5r5c7 n8r5c7 .

biv-chain[3]: r1c4{n4 n5} - b8n5{r9c4 r8c6} - c6n1{r8 r2} ==> r2c6≠4
whip[1]: c6n4{r6 .} ==> r4c4≠4, r5c4≠4
biv-chain[4]: r7c7{n1 n3} - r7c2{n3 n6} - r4n6{c2 c4} - r4n8{c4 c7} ==> r4c7≠1
hidden-single-in-a-block ==> r4c9=1

+-------------------+-------------------+-------------------+
! 1     2     349   ! 45    359   6     ! 7     8     349   !
! 349   5     78    ! 1478  389   139   ! 2     349   6     !
! 3469  3469  78    ! 478   389   2     ! 349   1     5     !
+-------------------+-------------------+-------------------+
! 2     3469  5     ! 68    7     349   ! 3489  349   1     !
! 3469  1     3469  ! 568   3589  3459  ! 34589 27    27    !
! 8     7     349   ! 2     1     3459  ! 3459  6     349   !
+-------------------+-------------------+-------------------+
! 5     36    1236  ! 9     4     8     ! 13    237   237   !
! 7     8     1249  ! 3     6     15    ! 149   2459  249   !
! 349   349   1349  ! 15    2     7     ! 6     3459  8     !
+-------------------+-------------------+-------------------+

Trid-OR4-relation between candidates n6r3c2, n6r4c2, n5r5c7 and n8r5c7
+ same valence for candidates n8r5c7 and n6r4c2 via c-chain[4]:
n8r5c7,n8r4c7,n8r4c4,n6r4c4,n6r4c2
==> Trid-OR4-relation can be split into two Trid-OR3-relations with respective lists
of guardians:
    n6r3c2 n6r4c2 n5r5c7  and n6r3c2 n5r5c7 n8r5c7 .
```

Trid-OR3-whip[5]: r7n1{c7 c3} - r7n6{c3 c2} - OR3{{n6r4c2 n6r3c2 | n5r5c7}} - b5n5{r5c4 r6c6} - r8c6{n5 .} ==> r8c7≠1
hidden-single-in-a-block ==> r7c7=1
Trid-OR3-whip[5]: c4n6{r5 r4} - OR3{{n6r4c2 n5r5c7 | n6r3c2}} - r7c2{n6 n3} - b9n3{r7c9 r9c8} - r9n5{c8 .} ==> r5c4≠5
naked-pairs-in-a-block: b5{r4c4 r5c4}{n6 n8} ==> r5c5≠8
whip[1]: c5n8{r3 .} ==> r2c4≠8, r3c4≠8
Trid-OR3-whip[5]: r7n2{c9 c3} - r7n6{c3 c2} - OR3{{n6r4c2 n6r3c2 | n5r5c7}} - b5n5{r5c5 r6c6} - r8n5{c6 .} ==> r8c8≠2
hidden-pairs-in-a-column: c8{n2 n7}{r5 r7} ==> r7c8≠3
Trid-OR3-whip[5]: b9n3{r9c8 r7c9} - r7c2{n3 n6} - OR3{{n6r4c2 n6r3c2 | n5r5c7}} - c5n5{r5 r1} - r1n3{c5 .} ==> r9c3≠3

```
Trid-OR3-whip[5]: b9n3{r9c8 r7c9} - r7c2{n3 n6} - OR3{{n6r4c2 n6r3c2 | n5r5c7}} -
c5n5{r5 r1} - c4n5{r1 .} ==> r9c8≠5

singles ==> r8c8=5, r8c6=1, r9c4=5, r1c4=4, r3c4=7, r2c4=1, r3c3=8, r2c3=7, r2c5=8,
r1c5=5, r9c3=1
z-chain[3]: r2n4{c1 c8} - r9n4{c8 c2} - b1n4{r3c2 .} ==> r5c1≠4
z-chain[4]: r8c7{n9 n4} - c9n4{r8 r6} - r6c3{n4 n3} - r1c3{n3 .} ==> r8c3≠9
whip[1]: r8n9{c9 .} ==> r9c8≠9
biv-chain[3]: c8n9{r4 r2} - r1c9{n9 n3} - b9n3{r7c9 r9c8} ==> r4c8≠3
biv-chain[3]: r4c8{n9 n4} - c9n4{r6 r8} - r8n9{c9 c7} ==> r4c7≠9, r5c7≠9, r6c7≠9
z-chain[4]: c8n9{r4 r2} - r2c6{n9 n3} - r4c6{n3 n4} - r4c8{n4 .} ==> r4c2≠9
z-chain[3]: r5c5{n3 n9} - b4n9{r5c1 r6c3} - r1c3{n9 .} ==> r5c3≠3
t-whip[3]: c5n9{r3 r5} - b4n9{r5c3 r6c3} - r1n9{c3 .} ==> r3c7≠9
hidden-single-in-a-column ==> r8c7=9
biv-chain[3]: r1n3{c3 c9} - r3c7{n3 n4} - r2n4{c8 c1} ==> r2c1≠3
finned-swordfish-in-columns: n3{c1 c5 c8}{r9 r5 r3} ==> r3c7≠3
singles ==> r3c7=4, r2c1=4
whip[1]: c7n3{r6 .} ==> r6c9≠3
x-wing-in-rows: n9{r2 r4}{c6 c8} ==> r6c6≠9, r5c6≠9
x-wing-in-rows: n9{r1 r6}{c3 c9} ==> r5c3≠9
x-wing-in-columns: n4{c2 c8}{r4 r9} ==> r4c6≠4
naked-pairs-in-a-block: b5{r4c6 r5c5}{n3 n9} ==> r6c6≠3, r5c6≠3
biv-chain[3]: r7c2{n6 n3} - r9c1{n3 n9} - c2n9{r9 r3} ==> r3c2≠6
hidden-single-in-a-block ==> r3c1=6

                      +-------------+-------------+-------------+
                      ! 1   2   39  ! 4   5   6   ! 7   8   39  !
                      ! 4   5   7   ! 1   8   39  ! 2   39  6   !
                      ! 6   39  8   ! 7   39  2   ! 4   1   5   !
                      +-------------+-------------+-------------+
                      ! 2   346 5   ! 68  7   39  ! 38  49  1   !
                      ! 39  1   46  ! 68  39  45  ! 358 27  27  !
                      ! 8   7   349 ! 2   1   45  ! 35  6   49  !
                      +-------------+-------------+-------------+
                      ! 5   36  236 ! 9   4   8   ! 1   27  237 !
                      ! 7   8   24  ! 3   6   1   ! 9   5   24  !
                      ! 39  349 1   ! 5   2   7   ! 6   34  8   !
                      +-------------+-------------+-------------+

At least one candidate of a previous Trid-OR3-relation has just been eliminated.
There remains a Trid-OR2-relation between candidates: n6r4c2 n5r5c7

At least one candidate of a previous Trid-OR3-relation has just been eliminated.
There remains a Trid-OR2-relation between candidates: n5r5c7 n8r5c7

At least one candidate of a previous Trid-OR4-relation has just been eliminated.
There remains a Trid-OR3-relation between candidates: n6r4c2 n5r5c7 n8r5c7

Trid-OR2-whip[1]: OR2{{n8r5c7 n5r5c7 | .}} ==> r5c7≠3
finned-x-wing-in-rows: n3{r3 r5}{c5 c2} ==> r4c2≠3
naked-pairs-in-a-block: b4{r4c2 r5c3}{n4 n6} ==> r6c3≠4
naked-pairs-in-a-column: c3{r1 r6}{n3 n9} ==> r7c3≠3
```

```
finned-x-wing-in-rows: n3{r7 r1}{c9 c2} ==> r3c2≠3
stte
```

15. Miscellanea

15.1 Examples and large scale studies

In the original releases of CSP-Rules-V2.1, a set of examples was included, in an "Examples" sub-folder of CSP-Rules-V2.1, itself including one sub-folder for each application. The purpose was not to provide an extensive set of cases, but to give users a more concrete idea of the various puzzles and their resolution paths, with more cases than can be mentioned in book form. Nor was the purpose to replace the original websites; it was to direct the user to them for more examples of the same kinds.

A new independent GitHub repository has now been created ([CSP-RULES-EXAMPLES]), into which the previous set of examples has been moved. This is because examples were starting to take too much place inside the CSP-Rules-V2.1 project and I don't want to consider any addition of an example as a change to CSP-Rules itself. This also allows to have a top-level README.md file specifically dedicated to describing the Examples. Beware: when installed as described on GitHub, and contrary to the previous Examples folder, this CSP-Rules-Examples folder is in the CSP-Rules folder but not in its CSP-Rules/CSP-Rules-V2.1 sub-folder. A new global variable ?*CSP-Rules-Examples-Dir* has been added to CSP-Rules-V2.1, referring to this new folder of examples if it has been installed at the proper place. This allows you to re-run the examples with the commands written with them (generally for full files of examples).

Most of 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.md” file of the “CSP-Rules-Examples” repository on GitHub for more specific details.

The resolution paths given in all the examples are the raw output from CSP-Rules, with some of 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…”

With time, the SudoRules part of the CSP-Rules-Examples repository tends to include much more than mere examples. It also includes full results of large scale studies of puzzle collections with specific properties, such as:

- the cbg-000 controlled-biased collection, containing detailed classification results with respect to many different ratings and their comparisons;
- eleven’s list of 3-digit patterns and SudoRules results on extracting the only one that could potentially requite T&E(3) to be proven contradictory;
- mith’s databse of 63,137 min-expand puzzles in T&E(3), with e.g. proofs of contradiction in T&E(3) and lots of classification results.

While the ordinary examples illustrate the “general CSP solver” side of CSP-Rules, such additions illustrate its “research tool” side.

15.2 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" (i.e. number of inference steps) 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**.

15.3 Reporting bugs

Any bug found in CSP-Rules or in CSP-Rules-Examples should preferably be reported via GitHub.

15.4 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 both in the CSP-Rules-V2.1 “Publications” folder of [CSP-RULES] and on ResearchGate (https://www.researchgate.net/profile/Denis-Berthier/research).

[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.

*[Berthier 2020]: BERTHIER D., *Basic User Manual for CSP-Rules-V2.1* (First Edition), Lulu Press, August 2020.

*[Berthier 2021a]: BERTHIER D., *Basic User Manual for CSP-Rules-V2.1 (Second Edition)*, Lulu Press, November 2021.

*[Berthier 2021b]: BERTHIER D., *Pattern-Based Constraint Satisfaction and Logic Puzzles (Third Edition)*, Lulu Press, November 2021.

[BUM1], [BUM2], [BUM]: respectively, abbreviations for [Berthier 2020], [Berthier 2021a], or for any of the two.

[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], [PBCS3], PBCS]: respectively, abbreviations for [Berthier 2012], [Berthier 2015], [Berthier 2021] or for any of the three.

[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. Permanent address: http://carva.org/denis.berthier.

[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.

Software

[CSP-RULES]: https://github.com/denis-berthier/CSP-Rules-V2.1

[CSP-RULES-EXAMPLES]: https://github.com/denis-berthier/CSP-Rules-Examples

[CBG]: https://github.com/denis-berthier/Controlled-bias_Sudoku_generator_and_collection