2009 21st IEEE International Conference on Tools with Artificial Intelligence

Using graphs to derive CSP heuristics and its application to Sudoku

Pablo San Segundo[1], Agustín Jiménez
Intelligent Control Group, Universidad Politécnica de Madrid
pablo.sansegundo@upm.es

Abstract

This paper presents a general purpose methodology to derive new heuristics for constraint satisfaction problems using the information contained in an auxiliary graph. The method is then applied to implement a powerful heuristic for Sudoku which is 'almost perfect' in the sense that a depth first search procedure needs very few backtracks to solve the 16x16 puzzle. The heuristic uses a least-constraining tie break for variable ordering, something very unusual in CSP domains.

The paper also presents a minimal set of logical rules which the authors conjecture to solve all 9x9 Sudoku grids with unique solution with just one level of recursion at most.

Keywords: Sudoku, heuristics, constraint satisfaction, graphs, variable ordering, value ordering.

1. Introduction

Sudoku is a popular logic-based number placement problem. The term *Sudoku* is Japanese word which means 'single number'. The modern puzzle is attributed to Howard Garns, an American architect who never saw his creation as a worldwide phenomenon. It first appeared in print in [Garns 1979], and quickly became popular in Japan in the 1980s turning into an international hit in 2005.

Theoretical work on Sudoku falls into two main areas: complete grids and puzzles. In the former the interest lies in enumerating the number of possible solutions with or without symmetries [Russel & Jarvis 2006]. Puzzle analysis centers on the initial given values and is the main concern of this paper.

The class of Sudoku puzzles consists of filling a partially completed square grid of NxN cells partitioned into K blocks (also called boxes or regions) to be filled in using a prescribed set of S distinct symbols (typically the numbers {1, ..., S}), so that each row, column and region contains exactly one of each element in S. There are many Sudoku variants characterized by the size (N) and the shape of their regions. This paper focuses in classic Sudoku where there are exactly N square regions each containing N cells of the NxN grid. The most popular variant in classic Sudoku is a 9x9 grid (N=9), but larger grids (i.e. N=16, 25) also appear in puzzle magazines regularly. The problem is known to be NP-complete [Takayuki & Takahiro 2002].

In this paper we approach Sudoku from a constraint satisfaction perspective. Formally, a constraint satisfaction problem (CSP) is defined by a set of variables X, and set of constraints C. Each variable $x_i \in X$ has a non empty domain D_i of possible values. A constraint specifies the allowable combinations of values for a subset of the variable set. An assignment of variables associates a subset of variables with some value. The aim of the CSP is to find a complete assignment, i.e. an assignment compatible with every constraint in C (cf. [Apt 2003]). From a CSP point of view, each cell in a Sudoku grid is a variable and each candidate a value of the domain of the cell.

The motivation of this research is twofold. In a recent work (at the moment under review) we employed successfully a graph-based technique to find a leading, to our knowledge, heuristic for the N-Queens problem (i.e. that needed almost no backtracking). We present here a general method to derive heuristics in a similar way as we did for N-Queens. Other related work on CSP heuristics can be found in [Lecoutre et al. 2007] [Argyropulos & Stergiou 2008] which use a SAT-based approach instead of a graph-based one. A second aim is to study Sudoku from an AI perspective. We are interested in finding the smallest possible set of logical rules which solves the 9x9 puzzle with minimum trial an error. We are also interesting in applying our methodology to derive powerful heuristics for this domain.

[1] Pablo San Segundo is also a chess GrandMaster and currently a member of the Spanish Olympic team

1082-3409/09 $26.00 © 2009 IEEE
DOI 10.1109/ICTAI.2009.78

The paper is structured as follows: Section 2 presents a new general purpose graph-based method for deriving heuristics for a CSP. Section 3 models Sudoku as a CSP and describes how to derive new variable and value ordering heuristics in the domain. Section 4 presents two sets of experiments: In first the logical rules to filter candidate values are tested. In the second place results for a number of new heuristics are recorded. Finally Section 5 presents conclusions and analyses possible future lines of research.

2. A new graph-based method for finding decision heuristics for CSP

Based on the results obtained for N-Queens we propose the following empirical method for finding good decision heuristics for a generic CSP. For simplicity we will assume the CSP to be binary since it is well known that binarization is always possible in polynomial time (c.f. [Dechter and Pearl 1989]).

1) *Ground the input CSP to a constraint graph* G_c *such that the vertices are the values of variable domains and the edges represent inconsistent pairs of values. If necessary, change the structure by adding new variables conveniently*. More formally, for a $CSP \equiv (X, D, C)$ where $X = \{x_1, x_2, \cdots, x_n\}$ is a set of variables, $D = D_1 \cup D_2 \cup \cdots \cup D_n$ the superset containing the domains of all variables and C a set of binary constraints, the constraint graph $G_c = (V_c, E_c)$ is such that $V_c = V_c(D)$ and

$$E_c = \{(v_x, v_y) / \exists(< x_i, v_x >, < x_j, v_y >) \in c,\ c \in C\} \quad (i \neq j, v_x \in D_i, v_y \in D_j) \tag{1}$$

2) *Obtain the complement graph* $G_{CSP} = \overline{G_c}$. The way the constraint graph has been defined, any k-clique in G_{CSP} is a consistent labeling for k variables of the CSP. Therefore all maximum cliques in *G* belong to the set of solutions of the CSP iff the solution exists.

3) *Choose any general CSP backtrack search algorithm with a consistency checking procedure* (i.e arc-consistency) *as the basic solving procedure.* The search procedure should employ some form of look forward technique (i.e. full look-ahead; relax to partial look ahead or forward checking if necessary). We note that the better the prediction of conflicts the more informed the heuristic obtained is likely to be.

4) *Whenever a new value v for a variable is chosen, expand graph* G_{CSP} *as in a maximum clique search*. More formally, if U is the graph to be updated when selecting vertex $v \in U$, the new graph U' is $U' = U \cap N_G(v)$ (the induced graph over G_{CSP} by all the vertices in U with an endpoint in *v*).

5) *Use graph U to determine a variable and (most often) a value ordering heuristic at the current node*. Employ graph invariants (i.e. degree, density etc.) and decision heuristics derived from maximum clique to adapt well known general purpose variable and value ordering CSP heuristics with special attention to breaking of ties. Here are some examples of reasoning with graph *U*:

- *Most-constrained variable*: Select the variable with minimum number of vertices in *U*. We note that this does not necessarily imply the variable with minimum domain in the original CSP since the structure may have changed.
- *Most-constraining variable*: For every variable minimize the number of edges in U with an endpoint in vertices which to refer ti a value from its domain. a value belonging to its domain.
- *Least constraining value*: For every value $v \in U$ of a previously selected variable, propagate consistency with full look-ahead (relax to partial look-ahead or forward search if this is impractical) and compute the corresponding graph as explained in step 4. Select the value which *maximizes* the number of edges in the graph.

We underline the empirical nature of the method based on the results obtained for N-Queens and Sudoku. As such, it is also very flexible. For example, it is possible to use semantics to filter edges in *U* which are known to be unimportant or which map with a particular distribution of constraints in the original CSP. It is also possible to employ other graph invariants instead of degree to reason over *U* (i.e maximize graph density for value ordering). It is well known that efficiency in exact maximum clique search depends heavily on the way vertices are presented to the algorithm [Tomita & Seki 2003]. Therefore initial or dynamic vertex reordering in *U* could be used to break tiebreaks for variable or value selection at the current node.

The proposed method may introduce heavy overhead w.r.t. a typical backtracking search procedure because graph *U* has to be computed explicitly at every node. However the additional time consumed can have an adequate counterpart in the reduction of the search space obtained because of the use of more informed heuristics. Also many well known difficult problems (i.e. NP-Hard problems such as N-Queens or Sudoku) are formulated in a general context. In these domains our method could be applied over small instances to

obtain good heuristics which could then be used in the larger ones.

3. Sudoku: a case study

We will use letters *r*, *c*, *s*, *b, n* to refer to a generic row, column, square, block, cell or number respectively of an NxN Sudoku grid. Optionally a suffix may be added to refer to more than one entity. Symbols for numbers are digits form 1 to N. The term *unit* will be employed to refer to any row, column or block in the grid. For concrete units the symbolic notation most frequently employed in Sudoku literature will be used, i.e. a capital letter followed by number (i.e.R1, C1, B1 indicates the first row, column and block respectively). All entities are numbered from left to right (columns) and from top to bottom (rows) in the grid, so that B1 block is in the upper left hand corner and BN at the bottom right in the grid. Cells inside a block are also numbered from 1 to N in a similar fashion. A cell in the grid $s \in D(r)xD(c)$ is any row-column pair. Alternatively a cell can also be referred to by a block-number pair ($s \in D(b)xD(n)$).

3.1. Rules for filtering candidates

We consider cells in the Sudoku grid as variables of an equivalent CSP. The rules of Sudoku restrict these values at rows, columns and blocks. The aim of the search is to find a compatible assignment to the variables.

Much of the research in Sudoku (as in many other logical problems) is concerned with backtrack-free search, as in [Simonis 2005][Berthier 2007]. In [Berthier 2007] a set of first order logic groups of rules are enumerated and classified with the aim of solving the 9x9 puzzle deterministically. A more informal approach appears in many websites of the s*udokian community* where an open set of more than a hundred different rules (with exotic names such as Swordfish, X-Wing, Chains etc.) can be found [e.g. Sudoku (www)].

In this paper, the approach is the opposite. We are interested in finding the smallest possible set of domain filtering rules which minimizes the need of backtrack search. For these purpose we have selected a small subset of the rules commonly used by the sudokian community). For simplicity we use natural language to describe the rules (a more formal definition in first order predicate logic can be found in [Berthier 2007]).

- **Naked Single** (NS): if there is only one candidate *k* for a cell→ assert *k* as the value of the cell.
- **Hidden Single** (HS): if there is a unit and a number *n* such that the number is a candidate for just one cell in the unit → assert *n* as the value of the cell.
- **Block-Row/Column interaction** (BRC): if there is a block *b* and a number *n* such that the number is a candidate for only cells in the block which belong to the same row/Column → remove *n* from all the cells in the row/column outside *b*.
- **Row/Column-Block interaction** (RCB): if there is a row/column *r/c* and a number *n* such that *n* is a candidate for only cells in *r/c* which belong to the same block *b* → remove *n* from all the cells in *b* outside *r/c*.
- **Naked Pair** (NP): if there is a unit, two different numbers n_1 and n_2 and two different cells s_1 and s_2 such that the candidates of s_1 and s_2 are exactly n_1 and n_2 → remove n_1 and n_2 from the all other candidates in the unit.
- **Hidden Pair** (HP): if there is a unit, two different numbers n_1 and n_2 and two different cells s_1 and s_2 such that n_1 and n_2 only appear as candidates in only those two cells in the unit → remove all other candidates from the two cells.
- **Naked Triplet** (NT): if are three different numbers n_1, n_2, n_3 and three different cells c_1, c_2, c_3 in a unit such that none of the cells has other candidates apart from n_1, n_2 and n_3 → remove all other candidates from the cells.
- **Hidden Triplet** (HT): if there are three different numbers n_1, n_2, n_3 and three different cells c_1, c_2, c_3 in a unit such that none of the cells has other candidates apart from n_1, n_2 and n_3 → remove all other candidates from the three cells.
- **Naked Quadruplet** (NQ): Similar to NS but extended to four numbers and squares.
- **Hidden Quadruplet** (HQ): Similar to HS but extended to four numbers and squares.

The first two rules (NS and HS) are the basic constraint propagation rules in Sudoku, i.e. the only ones which place a number in a particular cell. The rest of the rules filter domain values in each cell.

The filtering process is iterative. It employs the above set of rules and applies them in order until no more changes are found in the grid, so that consistency for the implicit constraints modelled by the rule is guaranteed. In the process, no rule is ever fired without exhausting filtering by previous rules. This allows classifying Sudokus by difficulty according to the highest ordered rule which needed to be fired to fill the

grid. We will refer to the ordering of the set of rules as the *standard order*.

We note that there is redundancy in the constraints modelled by the rules (i.e. rules for quadruplets subsume triplets which, in turn, subsume pairs). To avoid classifying mistakes the relative firing order of these rules should be NP→NT→NQ and HP→HT→HQ. In the actual implementation, care has been taken so that domains of antecedents do not overlap.

3.2. Graph-based heuristics for Sudoku

Given an NxN Sudoku grid, the proposed auxiliary graph $G_{CSP} = (V, E)$ is such that $|V| = N^3$. Its vertices are pairs (s, N) (i.e. a cell and an admissible candidate for the cell after applying the filtering procedure to the grid) and there exists an edge between two number-cell pairs iff they are compatible according to the basic rules of the game. Initially if a number is already given in the grid all vertices which refer to that cell are removed from V.

During search, the graph is updated so that only vertices representing the remaining empty cells and valid candidates are left at the current node. Figure 1 depicts an example of G_{CSP}. The 4x4 Sudoku grid above has 4 empty cells with candidates {1,2} in all cases. Below, the corresponding graph (in this case regular) has 8 vertices, each vertex an admissible cell-candidate pair. The value inside each vertex is the candidate and the labels (r, c) indicate the concrete cell in the grid.

We have used simple edge-based heuristics for variable and value ordering which, in combination with a good filtering algorithm, have performed well in our experiments. These are as follows:

- **Variable ordering:** *Select the cells with the minimum number of candidates* (the most-constrained variable). If tie breaks are needed select the variable with minimum number of edges in G_{CSP} at the current node (a *most-constraining* strategy).
- **Value ordering**: *Select the candidate in the previously selected cell which maximizes the number of edges in the updated* G_{CSP} (i.e. which results from placing the candidate value in the grid). Intuitively, the greater the number of edges in G_{CSP} the higher the probability that there is a maximum clique of size the number of empty cells in the grid (a solution to the Sudoku).

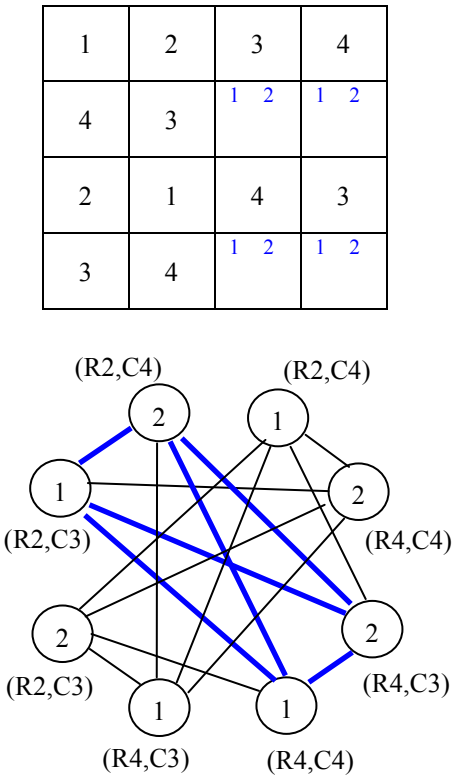

Figure 1. *Above*: A Sudoku grid S. *Below*: G_{CSP} for S. The thicker edges denote a clique of size the number of empty cells in the grid, one of the possible solutions.

Additionally we have implemented a probabilistic tie breaking strategy P for value ordering. Let S_g be the set of all possible NxN Sudokus. Given a Sudoku grid $s_g \in S_g$, $P(s_g): S_g \to \mathbb{N}$ is the product of the number of remaining candidates in each empty cell. Intuitively, P is a measure of the different ways to fill in the grid. The higher the value of P the more difficult the remaining part of the search is expected to be, so grids with a lower P should be preferred.

We note that the way the graph-based heuristic has been defined, it is not necessary in practice to explicitly compute G_{CSP}. Evaluating value-ordering for a particular Sudoku grid can be done by incrementing a counter for every two cell-candidate pairs which are valid partial assignments in the grid (e.g. the graph in figure 1 evaluates to 32, the number of valid assignments of two different cells).

Besides the classical variable-value decision heuristics, it is also possible to implement more informed graph-based heuristics considering all possible cell-candidate pairs in the decision. This increases the branching factor in each node (the recursive procedure must consider all possible candidates for each cell) but also improves the pruning of the search space. This type of heuristics might

result adequate in problems were it is known that a solution exists or were the filtering procedure is expected to prune large parts of the space, as is the case in Sudoku. We have implemented variants of these heuristics considering value ordering over the full set of candidates.

3.3. The search algorithm

A well known basic CSP chronological backtracking search algorithm (BACTRACK-CSP) is described in table 1. We use BACTRACK-CSP with full look ahead maintaining consistency w.r.t. the 10 rules listed in section 3.1. This is done in the FILTER function (step 4). The functions SEL-UNASSIG-VAR and ORDER-VAL are used to implement the general-purpose variable and value ordering heuristics respectively. In heuristics which make decisions taking into consideration all possible assignments in empty cells step 2 is skipped and the scope of ORDER-VAL is modified to include all candidates in the grid.

Table 1. Basic backtrack CSP search procedure

INPUT: 1) *csp*, 2) empty assignment

RETURNS: fail or solution assignment

function BACTRACK-CSP (*assig* , *csp*)

1. **if** *assig* is complete **return** *assig*
2. *var* ← SEL-UNASSIG-VAR(*csp*)
3. **for each** *value* ← ORDER-VAL(*var*, *csp*)
4. *f* ← FILTER (*var*, *value*, *csp*)
5. **if** *f* is not fail **then**
6. add {*var* = *value*} to *assig*
7. *r* ← BACTRACK-CSP (*assig*, *csp*)
8. **if** *r* ≠ fail **then return** (r)
9. delete {*var* = *value*} from *assig*
10. **return** *fail*

4. Experiments

For the experiments we used the well known Gordon Royle benchmark [Royle 2009] which contains a set of minimal Sudokus, i.e. Sudokus that have a unique solution with the minimum possible number of givens (at the moment 17, and it is strongly conjectured that this is the real lower bound). The benchmark is updated regularly with new instances; at the time of the experiments 48072 were available.

We run two different types of tests: tests to validate the proposed set of rules and tests with different heuristics. Tests with rules were implemented both in C (running on a P4, 2.6GHz on WinXP) and in MATLAB (Quad, 2.4 GHz on WinXP). Results were then crosschecked. As expected, CPU times for the C application were faster and only these are recorded in table 2. Experiments with heuristics were implemented only in MATLAB and run using the Quad processor.

4.1. Rules

The rules employed in FILTER were evaluated by applying the filtering algorithm to the benchmark using 5 different rule orders and recording the number of instances solved. Table 2 records the results obtained. In the first column the filtering algorithm uses the standard order of rules. In the rest of the columns two rules are swapped in each case as indicated by the column header. For each cell, the row indicates the highest level rule needed to solve a Sudoku. The value inside the cell is the number of Sudokus solved.

The total number of Sudokus solved is very high (40771, an 84.81% of the benchmark) and obviously independent of the ordering of the rules. This validates our subset of rules and is consistent with best results for backtrack free search found elsewhere (i.e. [Berthier 07]; [Simonis 05] gets an 85.11% for a subset of 7611 instances of Gordon Royle using hyper arc consistency of the alldifferent constraint applied to rows, columns and blocks). The table also shows that the hidden rules (HP, HT, NT) find more solutions when ordered in front of the corresponding 'naked' variants.

We further carried out a specific test to find the minimum set of rules needed to solve the benchmark with only one level of trial-and-error, i.e. after an initial filtering, we test all possible candidates in each cell lexicographically and apply the filtering procedure once more. Interestingly, *only the basic propagation rules (NS, HS), the interaction rules (BRC, RCB) and just one of the pair rules (NP or HP) are needed to solve all Sudokus from the benchmark*. We strongly conjecture that this result can be extended to all 9x9 Sudokus with a unique solution.

4.2. Heuristics

Results in table 2 show that 9x9 Sudoku grids are too simple to test heuristics in searches where at every step consistency for the 10 selected rules is ensured. Since no public benchmark for 16x16 Sudoku grids is available (to our knowledge) we have implemented our own random benchmark.

We generate Sudoku grids starting from a random complete puzzle and removing givens until it becomes unsolvable with the filter algorithm (i.e. applying the proposed 10 rules iteratively). We then empty a further *k* cells to ensure that backtrack search is needed to solve the puzzle, so that the bigger *k* the more solutions are likely to exist. The procedure guarantees that all instances in the benchmark are solvable. The notation used is <sizeofgrid>_<k>_<size> (i.e. *16_1_500* are 500 instances of 16x16 grids with *k*=1).

A number of heuristics have been implemented both graph and non graph-based with a maximum of one level use of FILTER to make decisions. Some heuristics involve several levels of tie-breaks so we have used a specific notation to indicate how decisions at each level are made:

- **L**: Lexicographical
- **D**: Select cells with minimum number of candidates (considered not graph-based since it is a direct application of *most-constrained* variable ordering)
- $\mathbf{G_{max}}$: A graph-based heuristic which maximizes the number of edges in the proposed G_{CSP} graph associated with the grid.
- $\mathbf{G_{min}}$: Similar to G_{max} but minimizing the number of edges of G_{CSP}.
- **r1r2()**: Filter procedure using the first two rules (in the standard order): NS and HS. Strategies inside the parentheses are applied over the filtered grid.
- **P**: minimizes the result of multiplying the number of candidate values in every empty cell.

Table 2. Results for the filtering algorithm using the current Gordon Royle benchmark (48072 instances). The first column places rules in standard order. The rest of the column headers indicate the single swap w.r.t. the standard order. CPU time is measured in seconds.

	Std. order	**RCB->BRC**	**HP->NP**	**HT->NT**	**HS->NS**
CPU time	27.762	27.953	28.000	27.766	27.703
% solved	84.81	84.81	84.81	84.81	84.81
NS/HS	21487	21487	21487	21487	21487
BRC	14043	1706	14043	14043	14043
RCB	1097	13434	1097	1097	1097
NP	2313	2313	158	2313	2313
HP	1775	1775	3930	1775	1775
NT	36	36	36	7	36
HT	17	17	17	46	17
NQ	3	3	3	3	0
HQ	0	0	0	0	3

Table 3. Results for different decision heuristics using our own 16x16 benchmark. Notation: **L**-lexicographical, **D**-minimum number of candidates, **E**-minimum number of empty cells, $\mathbf{G_{max}}$-maximum number of edges, $\mathbf{G_{min}}$-minimum number of edges, **P**- minimum product of the number of candidates in every empty cell, **r1r2-** filtering with the first two rules according to the standard order (NS, HS). A suffix *v* refers to variable ordering. One example $G_{max}r1r2(EP)$: Orders all possible assignments to empty cells by maximizing the number of edges in the resulting G_{CSP}. To break ties updates the current grid with the assignment and applies HS and NS rules. It then selects the minimum number of empty cells in the new grid. Breaks further ties by minimizing the product of the number of candidates in the empty cells.

	$\mathbf{L_v}$	$\mathbf{D_v}$r1r2(EP)	$\mathbf{D_vG_{vmax}}$ r1r2($\mathbf{G_{max}}$EP)	$\mathbf{D_vG_{vmin}}$ r1r2($\mathbf{G_{max}}$EP)	Dr1r2 ($\mathbf{G_{max}}$EP)	$\mathbf{G_{max}}$ r1r2(EP)
Av. max depth	8.372	6.898	6.120	6.934	2.316	2.444
Backtracks	5834	2000	542	1066	80	68
CPU time (s)	9.766	5.549	1.866	2.080	2.876	6.422

It becomes necessary to define the scope of the decisions. Suffix *v* indicates variable scope. Heuristics with no suffix make decisions considering all possible assignments in the grid

Results for a number of heuristics for a *16_1_500* set can be found in table 3. The first row of the table records the average maximum depth for the benchmark; the second row records the total number of backtracks and the third the total CPU time measured in seconds. The first four columns show results for variable-value ordering heuristics and the remaining two consider all cell-value pairs from the beginning. Results would seem to indicate that graph-based heuristics perform better than non graph-based ones, although this needs further experimental analysis.

The first column is the lexicographical ('first found') heuristic which performs the worst as expected. We note that the average maximum depth of the lexicographical search trees is not too high because of the very powerful rules in the FILTER procedure.

The variable-value heuristic in the fourth column ($D_vG_{vmin}r1r2(G_{max}EP)$) is the graph-based heuristic which we expected to perform best. It uses minimum number candidates to select a particular empty cell and breaks ties between cells using G_{mim} (a graph-based interpretation of a *most-constraining strategy*). Surprisngly $D_vG_{vmax}r1r2(G_{max}EP)$ performs even better using a graph-based *least-constraining strategy* as variable tie break. This is, in our opinion, an important result since, to our knowledge, we have not found elsewhere any example of CSP with a least-*constraining* variable tie break. This might be explained for two reasons:

1) *The probabilistic nature of the information captured by G_{vmax} in Sudoku*. G_{vmax} minimizes the number of times a candidate in a cell appears in other cells in the same row, column or region, which implies that there is more probability of being correct when choosing a particular value for the cell.

2) *The pruning capability of detecting a wrong decision*. In Sudoku, detecting a wrong decision (i.e. placing an incorrect value in a cell at the current node) implies removing the candidate from the corresponding cell. If the value appears in very few other cells in the same row, column or block this means that it is more probable that the FILTER algorithm will be able to place the deleted value in the remaining cells. Because search is done with full look ahead and the FILTER procedure is very powerful the cost of detecting a wrong decision is not high. The two arguments would seem to justify a *least-constraining* variable tie break in Sudoku, an important contribution in the authors' opinion since this strategy might possibly be extended to other domains of similar characteristics.

The heuristics in the last two columns of table 3 consider every possible assignment to empty cells and therefore make the 'best' choices. Both are 'almost perfect' in the sense that they make almost no mistakes (80/68 backtracks out of 500 instances). As expected, CPU time for $G_{max}r1r2(DP)$ was much greater than $Dr1r2(G_{max}DP)$ because of the greater cost of evaluating G_{max} for all possible cell-candidate pairs. However the worst case branching factor for a single node is N^3 (all cell-candidate pairs in an empty Sudoku grid) compared to N in the case of typical variable-value heuristics. This makes the former more difficult to apply for bigger grids (i.e. 25x25, 36x36 etc.) Considering CPU time, number of backtracks and grid sizes the best heuristic is $D_vG_{vmax}r1r2(G_{max}EP)$.

5. Conclusions and future work

This paper presents an empirical general purpose methodology to derive heuristics for constraint satisfaction problems. It is based on previous work by the authors on the N-Queens problem, which is currently being reviewed. The technique is used successfully in Sudoku to find a number of graph-based heuristics which need almost no backtracking in a generated benchmark of 16x16 Sudoku grids. The proposed theoretical framework makes them applicable to other domains.

This paper is also a step forward in the analysis of Sudoku from an AI perspective. A small set of logical rules is presented which solves the Gordon Royle benchmark of 48072 minimal puzzles with just one level of recursion (i.e. a search tree of depth 1). We strongly conjecture that this is extensible to all 9x9 Sudoku grids with a unique solution, but this needs to be proved.

In our experiments Sudoku variable ordering heuristics with a *least-constraining* strategy to break ties perform better than its *most-constraining* counterpart. This is, in the opinion of the authors, a very interesting result. We have proposed two plausible explanations for this fact but further work is needed to reach a final conclusion. It is the first example, to our knowledge, that a least-constraining strategy appears as the best option to break ties for general purpose variable ordering.

Finally, we note that all Sudoku instances used in our experiments were locally minimal (no redundant presets) and guaranteed to have a unique solution. Further tests are needed with bigger grids (i.e. 25x25, 36x36) with and without unique solution.

6. Acknowledgements

This work is funded by the Spanish Ministry of Science and Technology (Robonauta: DPI2007-66846-02-01) and supervised by CACSA whose kindness we gratefully acknowledge. The authors would also like to thank Sergio García Pedreira for his collaboration in this research.

7. Bibliography

[Garns 1979] Garns, H. "Number Place." *Dell Pencil Puzzles & Word Games*. (16), pp. 6, May 1979.

[Russel & Jarvis 2006] Russell, E and Jarvis, A. F. *Mathematicsof Sudoku II*, Mathematical Spectrum (39), pp. 54–58, 2006.

[Takayuki & Takahiro 2002] Takayuki, Y and Takahiro, S; *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*; Fundamentals of Electronics, Communications and Computer Sciences Vol.E86-A (5), pp.1052-1060, 2003.

[Apt 2003] Apt, K. A. *Principles of Constraint Programming*. Cambridge University Press 2003.

[Lecoutre et al. 2007] Lecoutre, C., Sais, L., Vion, J. *Using SAT Encodings to Derive CSP Value Ordering Heuristics*. Journal of Satisfiability, Boolean Modeling and Computation 1, pp.169–186, 2007.

[Argyropulos & Stergiou 2008] Argyropulos,Y. Stergiou, K. *A Study of SAT-Based Branching Heuristics for the CSP*, Lecture Notes in Computer Science (2631), pp. 38-50, 2008.

[Dechter and Pearl 1989] Dechter, R. Pearl, J. *Tree clustering for constraint networks*. Artificial Intelligence (38), pp.353–366,1989.

[Tomita & Seki 2003] Tomita, E, Seki, T. *An efficient branch-and-bound algorithm for finding a maximum clique*, Lecture Notes in Computer Science (2631), 278-289, 2003.

[Simonis 2005] Simonis, H. *Sudoku as a Constraint Problem;* Workshop on Modelling and Reformulating Constraint Satisfaction Problems CP 05, pp.13–27 2005.

[Berthier 2007] Berthier, D; *The Hidden Logic of Sudoku*, ISBN: 978-1-84799-214-7, Lulu.com, sec. edition, Nov. 2007.

[Sudoku (www)] http://www.ringsworld.com /sudoku

[Royle 2009] http://people.csse.uwa.edu.au /gordon /sudokumin.php