Declarative Implementations of Search Strategies for Solving CSPs in Control Network Programming

EMILIA GOLEMANOVA
Department of Computer Systems and Technologies
Ruse University
8 Studentska Street, Ruse
BULGARIA
EGolemanova@ecs.uni-ruse.bg

Abstract: - The paper describes one of the most researched techniques in solving Constraint Satisfaction Problems (CSPs) - searching which is well-suited for declarative (non-procedural) implementation in a new programming paradigm named Control Network Programming, and how this can be achieved using the tools for dynamic computation control. Some heuristics for variable and value ordering in backtracking algorithm, lookahead strategies, stochastic strategies and local search strategies are subjects of interest. The 8-queens problem is used to help in illustrating how these algorithms work, and how they can be implemented in Control Network Programming.

Key-Words: - Control Network Programming; graphical programming; declarative programming; Constraint Satisfaction Problem; MRV, degree, LCV and minimum-conflicts heuristics.

1 Introduction

Control Network Programming (CNP) is a relatively new programming paradigm developed by a team in which the author is involved in and is especially effective for solving problems with natural graph-like representation. There are two major CNP implementation techniques. In the first approach, the classical algorithms are essentially simulated in CNP – we refer to such implementations as **procedural implementations**. The other approach makes use of the built-in in CNP search mechanism which is an extended version of backtracking and the CN program has a descriptive not an algorithmic character. These are **non-procedural or declarative implementations**. The resulting programs are easier to read, modify, and extend, which is important in AI, where efficient algorithms are, in general, difficult to implement and require considerable experimentation.

In addition to the built-in search mechanism, CNP and more specifically, the **SPIDER** language supports powerful means for its dynamic control [1]. They turn out to be very a convenient tool for realization of various heuristic strategies in Problem Solving [2, 3, 4]. This paper expands the application area of these tools describing their usage for solving Constraint Satisfaction Problems (CSPs). On another point of view (the CSPs researcher's view), the aim of this work is to promote a new programming paradigm - CNP, as a convenient tool for illustration of the basic techniques in constraint satisfaction. The question of how to easily model various types of heuristics in a declarative way is addressed. The result is non-procedural implementations which are "natural", i.e. similar to the manner in which people think of and specify problems. Being so intuitive, these implementations and respectively **WinSpider IDE** (which is the last CNP IDE), can be used as an excellent approach for teaching and learning search in constraint satisfaction.

The 8-queens problem will be used as the illustrative example in this paper. Finding all solutions to the 8-queens puzzle is a good example of a simple but nontrivial problem. For this reason, it is often used as an example problem for various programming techniques, including nontraditional approaches such as constraint programming [5], logic programming or genetic algorithms. But, while the n-queens problem is a wonderful problem to study backtracking systems and is intensively used in benchmarks to test these systems, there are real problems that can be modeled and solved as n-queens problems. For instance, it has been used for parallel memory storage schemes, VLSI testing, traffic control and deadlock prevention [6].

2 CNP

Programming through control networks, or **Control Network Programming**, or just **CNP**, is a

style of high-level programming that has been inspired by the idea to create a convenient and effective way for solving problems that can be naturally visualized using graphs. In CNP, what corresponds to a program in a conventional programming language is a description of the problem in the form of a graph, called **Control Network** (**CN**). The CN is a finite set of subnets, one of which is the main subnet. The subnets can call each other, potentially recursively. Each subnet consists of labeled nodes (also routinely called states), and arrows between nodes. A chain of **primitives** is assigned to each arrow. The primitives are elementary actions, and if a parallel is to be drawn with the traditional languages, they correspond to user-defined functions in some imperative language. The complete program consists of two main components - the CN and the definitions of the primitives. The CN may actually be nondeterministic. The system "executes" the CN by implementing a backtracking-like search strategy for traversing the CN. Similarly to Prolog and other declarative languages, SPIDER provides means for static and dynamic control of the search process. Two groups of tools for dynamic search control (which is a subject of interest here) are available: **system options** and **control states**.

For more details on the structure and syntax of a CN program the reader is referred to [7], as well as to the web site [8] especially devoted to CNP. Representative examples of using CNP for solving various types of problems have been considered in [9]. CN programs and their behavior were more formally defined in [10], and the basics of their execution introduced.

3 8-Queens problem as a Constraint Satisfaction Problem

At first, we need to model the 8-queens problem as a CSP problem. To formalize a problem as a CSP, we must identify a set of variables, a set of domains and a set of constraints [11, 12, 16, 19]. For the 8 queens problem let :

- **variables** $\{Q_1, Q_2, \ldots, Q_8\}$ represent the queens,
- **domains** $Q_i \in \{1, 2, \ldots, 8\}$, $\forall i \in \{1, 2, \ldots, 8\}$,

where equality $Q_i=j$ determines the i-th queen is placed on the i-th row and j-th column (note, that each queen strictly determines the row where it is placed),

- **constraints**

$Q_i \neq Q_j$, $\forall i, j \in \{1, 2, \ldots, 8\}, i \neq j$ condition for columns,

$|Q_i-Q_j| \neq |i-j|$, $\forall i, j \in \{1, 2, \ldots, 8\}, i \neq j$ condition for diagonals.

As it is well known CSPs are commutative [12, 16]. This means that the order of any given set of actions has no effect on the outcome. As the consequence all CSP systematic search algorithms can generate successors by considering assignments for only a single variable at each node in the search tree.

There are two basic approaches how to solve problems defined by means of constraints: **backtracking** based search that extends a partial solution to a complete solution and **local search** that decreases the number of violations in a complete solution. These two approaches and their corresponding heuristics are implemented and discussed in the paper.

4 Backtracking approach: General-purpose heuristics for solving CSPs efficiently

The classic approach to solve CSPs is to use a backtracking search algorithm [5, 11, 12, 16]. This is a depth-first search that picks one variable at a time and chooses a value for this variable. The choice for a variable or value is called a **choice point** and the assignment of a value to a variable is called **labeling** [11].

Plain backtracking is an uninformed algorithm, so it is not very effective for large problems [12]. Informed search algorithms, such as A*, have better performance due to exploitation of domain-specific heuristics derived from the knowledge of the problem. But it turns out that CSPs can be efficiently solved without such domain-specific knowledge. Instead, there are **general-purpose heuristics** that do with the choice points and answer the following questions:

1. Which variable should be labeled next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. Can we detect inevitable failure early?

One important technique is the propagation of the consequences of an assignment on the other variables through the constraints (**lookahead strategies** [5, 11]). **Forward Checking (FC)** is the improved backtracking by lookahead technique. Another method of enhancing the search is by using heuristics that involve the **variable and value order**. Instead of doing this at random the sequences of variables and their instantiations can

be ordered. This can either be done **globally (static ordering)** before the search starts or **locally (dynamic ordering)** at every node [13-17]. The popular variable ordering heuristics - **Minimum Remaining Values (MRV)** and **degree** [12, 16], and a value ordering heuristic - **Least-Constraining-Value (LCV)** [12, 16] are the subject of the following CNP implementations.

4.1 CNP implementation of the general-purpose heuristics

We start with a discussion of a CNP implementation of the FC algorithm with MRV, degree and LCV heuristics on the example of the 8-queens problem (see Fig.1). Based on this solution we can easily simulate other strategies (simple and stochastic variant of FC) which will be described in Section 6. Figure 1 and all other figures depicting CNP implementations are generated from the WinSpider IDE.

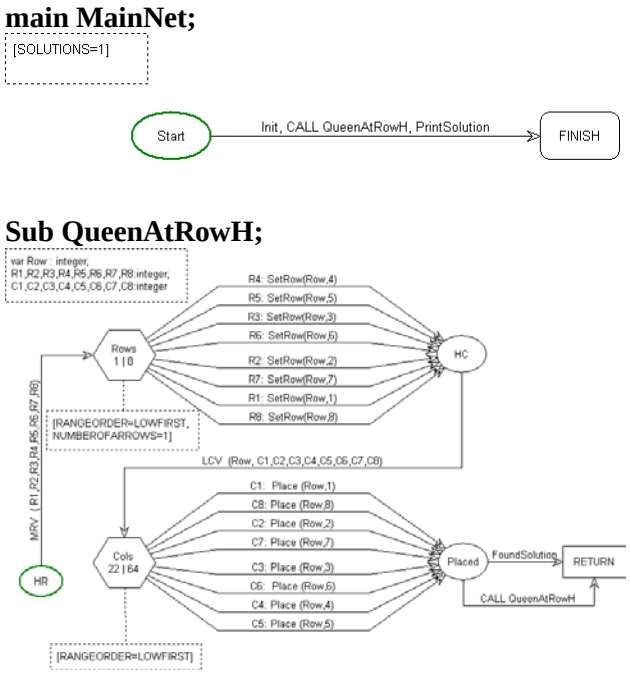


Fig.1 CNP implementation of the general-purpose heuristics for 8-queens problem

The control network consists of two subnetworks - main (**MainNet)** and subnet **QueenAtRowH. MainNet** invokes **QueenAtRowH** (CALL QueenAtRowH) and prints the found solution (primitive **PrintSolution).** The main job is accomplished by the recursive subnet **QueenAtRowH.** A recursive level corresponds to a search tree level (picks one variable and chooses a value for this variable). The two choice points (variable and value choices) are modeled by system control states of type RANGE **(Rows, Cols).**

4.2 Variable ordering

The definition of 8-queens problem as a CSP in Section 3 makes it quite clear that the variable choice corresponds to a choice of a row.

4.2.1 MRV heuristic

One of the most popular dynamic heuristic that decides how to choose the next variable is MRV, which comes from the fail-first principle. The MRV heuristic selects from the set of unassigned variables the variable with the fewest remaining values in its domain [16]. That's why it also has been called the "most constrained variable" heuristic [12]. It allows discovering a dead end sooner and thereby prunes the search tree.

This idea can be simulated in CNP using the control state **Rows** and primitive **MRV**. Primitive **MRV** calculates the heuristic evaluations $\mathbf{R_i}$, $i \in \{1, 2, \ldots, 8\}$ of all the 8 variables, i.e. the number of legal positions on the rows. They are used as evaluations of the outgoing arrows from **Rows.** State **Rows** is a RANGE type control state with low selector 1 and high selector 8, determining which emanating arrows will be attempted - only those whose evaluations are in the range [1; 8]. Already assigned variables will have zero remaining values in its domain, because there are no permitted positions on the rows with already placed queens. Therefore their corresponding arrows will be cut off. System option [RANGEORDER=LOWFIRST] states that the "survived" emanating arrows will be attempted in ascending order of their evaluations which means that the row with minimum remaining legal positions is chosen first. The other system option [NUMBEROFARROWS=1] is used because, as it was mentioned in Section 3, CSPs are commutative, i.e. it's only needed to consider assignments to a single variable at each step [12]. Primitive **SetRow(Row, Number)** assigns the parameter **Number** to the subnetwork variable **Row.**

4.2.2 Degree heuristic

Another heuristic is to choose the variable that is involved in the largest number of constraints, causing the largest reduction in the domains of the remaining variables [12, 16]. It attempts to reduce the branching factor on future choices. This is called the **degree heuristic**. The MRV heuristic is usually

a more powerful guide, but the degree heuristic can be useful as a tie-breaker [12, 16, 17].

The MRV heuristic for the 8-queens problem doesn't help at all in choosing the first row, because initially every row has 8 legal columns [12]. In this case, a static version of the degree heuristic comes in handy. It would lead to an ordering from the middle rows outward, since a queen in the middle row restricts the search more than one on the top or bottom of the board [13, 14, 18]. This can be done globally before the search, ordering rows from "inside out". In the CNP implementation this idea is realized setting the order of definition of outgoing arrows from state **Rows** as follows: 4, 5, 3, 6, 2, 7, 1, 8. Therefore, when the control is in the control state **Rows** the row with the smallest number of unthreatened squares is chosen due to the system option [RANGEORDER=LOWFIRST], but in case of more than one row have equal results, the row closer to the middle of the board is preferred.

The result of incorporating these two heuristics (MRV and degree) is definitely positive for n-queens problem and according to Cheadle [14] for example, for the 16-queens instance, the number of backtracks goes down to zero, and more difficult instances become solvable.

4.3 Lookahead technique

Analyzing the situation after placing a queen on the board, it's possible to detect the failure early, i.e. some values can be rejected at earlier stages [5, 11]. FC algorithm improves chronological backtracking by incorporating such a lookahead strategy. When the variable is labeled to a value *L* it checks the remaining domains of unassigned variables. If there is a domain reduced to an empty set, then *L* will be rejected.

The **MRV** primitive implements this idea detecting the rows still without a queen, but with all beaten squares. In this case the primitive is unsuccessfully executed and backtrack is forced. This causes a new value choice which is accomplished in the previous recursive level of the subnetwork **QueenAtRowH.**

4.4 Value ordering (LCV)

Once a variable has been selected, the algorithm must decide on the order in which to examine its values [12]. The way in which we choose values is important in case of looking for just one solution (other way all the values must be tried). The most popular heuristic for choosing a value is **LCV**. The idea is to choose the value that would eliminate the fewest values in the domains of other variables and thus leaving the most choices open for subsequent assignments to unassigned variables. Again it can be realized globally (static version) or locally (dynamic version).

The dynamic ordering is implemented by the primitive **LCV** and the control state **Cols**, which is of the type RANGE again. Primitive **Cols** calculates the heuristic evaluations $\mathbf{C_i}$, $i \in \{1, 2, \ldots, 8\}$ of all values of the already chosen variable, i.e. the heuristic evaluations of the positions (columns) in the chosen row. The attacked positions have the evaluation 0. The heuristic evaluation of an unattacked position is the number of attacked squares on the board after placing the queen on that square. The minimum number of attacked squares positions is 22 and the maximum - 64. These are the selectors of the control state **Cols**, therefore the illegal variable values (those with heuristic value 0) are rejected for examination. Option [RANGEORDER=LOWFIRST] must be used for that control state, too – as a result, the column causing the minimum attacks will be attempted first. Placing a queen on the board is performed by the primitive **Place(Row, Col)** on the attempted arrow.

In case of equal heuristic evaluations the static version of LCV heuristic is used as tie-breaker. The outer columns are preferred because they defeat the board less than the inside ones. Consequently the default order of the outgoing arrows from **Cols** is 1, 8, 2, 7, 3, 6, 4, 5.

The combination of FC algorithm, general purpose heuristics - MRV and LCV and problem-dependant heuristics as tie-breakers is turn out to be very effective approach. In [18] Wallac claims that 70-queens problem is solved within a second, and the algorithm scales up easily to 200 queens. The presented above CNP implementation finds the first solution of 8-queens problem in 8 steps which corroborates the result of Kalé in [22] that these heuristics appear to be almost perfect in the sense that they finds a first solution without any backtracks in most cases of n-queens problem for *n* from 4 to 1000.

5 Local Search approach: Constraint-Based Local Search

Local Search (**LS**) algorithms turn out to be very effective in solving many CSPs [12]. For example, they solve even the million-queens problem in an average of 50 steps. Local search also works well for real problems. It has been used to solve scheduling problems (observations for the Hubble

Space Telescope) even when these problems are dynamically changeable (airline schedules).

Local search takes a fundamentally different approach to solving CSPs than the systematic tree search of constraint programming (i.e. backtracking approach), whose main ideas were described above. In essence it uses a complete-state problem formulation and explores a graph, moving from solution to neighboring solution in the hope of improving it. Local search is, in contrast to constraint programming, not complete. There is no guarantee that an optimal solution will be found.

Constraint-Based Local Search (CBLS) uses constraints to describe and control local search. Although the constraints are the same as in constraint programming, the way in which they are used is not. They are not used to prune the search space, but to maintain a number of properties which can be used to guide the local search.

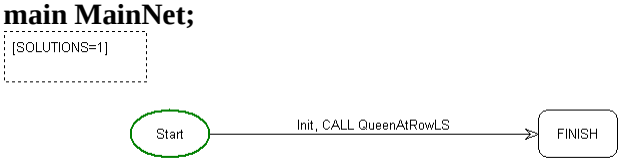
function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen, conflicted variable from VARIABLES[csp]
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return $failure$

Fig. 2 The MIN-CONFLICTS algorithm for solving CSPs by local search from [12]

The presented at Fig. 3 CNP implementation uses a slightly modified variant of the search procedure from Fig.2 as it is defined in [12]. It iterates **max_steps** number of times *(failure)* or until all constraints are satisfied *(success*). Checks for these two situations in the CNP implementation are performed by the primitives **GetToMaxIter** and **NoSol**. The initial state of local search assigns a value (randomly or greedy generated) to every variable. As it has been mentioned, the main operation is moving from one solution to a neighboring solution. Typically a move in CBLS consists of a simple reassignment of a value to a variable, but other moves are possible, such as multiple reassignments, swapping the value of two or more variables [12]. For example, in the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 rows, and the successor function picks one queen and considers moving it elsewhere in its row. Another possibility would be start with the 8 queens, one per row in a permutation of the 8 columns, and to generate a successor by having two queens swap columns. In the presented CNP solution the first approach is adopted.

main MainNet;



Sub QueenAtRowLS;

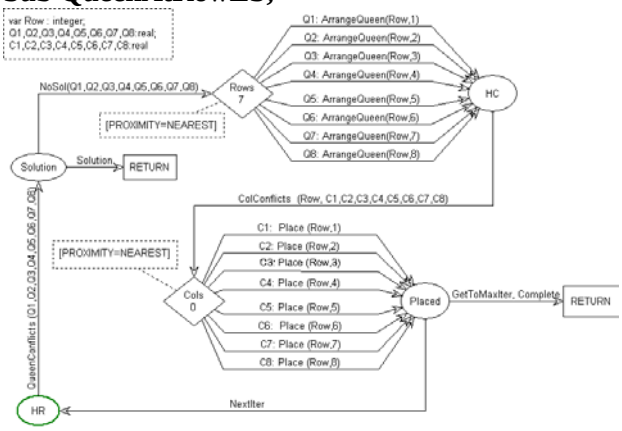


Fig. 3 CNP implementation of CBLS for 8-queens problem

At each iteration the queen for rearrangement must be chosen. Unlike the algorithm from Fig. 2 where the queen is randomly chosen from all the attacked queens, the CNP implementation (see Fig. 3) selects the queen which contributes to the most violations [19]. This is determined using the control state **Rows** of type SELECT with selector 7 (maximum number of conflicts which a queen is involved in) and the system option [PROXIMITY=NEAREST]. The control state uses as arrow evaluations (**Q_1, …, Q_8**) the number of violations that queens are involved in, calculated by the primitive **QueenConflicts**.

When a variable (queen) is selected, the algorithm selects a new value (column) for this variable. In choosing it, the most obvious and popular heuristic is to select the value that results in the minimum number of conflicts with other variables - the **min-conflicts heuristic** [12]. The primitive **ColConflicts(Row, C1, C2, C3, C4, C5, C6, C7, C8)** evaluates the effect of the placement of the selected queen (at **Row**) on the 8 columns. The control state **Cols** of type SELECT with selector 0 and the system option [PROXIMITY=NEAREST] causes the column with minimum number of conflicts to be chosen. As a tie-breaker in both control states the random choice is used. When both a queen and a value have been selected, the

primitive **Place** performs the assignment and thereby executes the actual move.

Figure 4 reports the experimental results for the CBLS model with SPIDER. It compares the performance of this algorithm according to the maximum allowed iterations (**max_steps**). The algorithm was run 1000 times (because of randomness) for a bound of 20 iterations, 50 iterations and 100 iterations. At **max_steps**=20 it found a solution on 37% of the runs in an average of 10 steps. Using **max_steps**=100 raises success to 75% in an average of 30 steps. But amazingly more than half of the successful runs required fewer than 20 iterations to find the solution and this is true for all the cases of the parameter **max_steps**. This means, firstly, that the frequency of occurrence of a solution with a small number of iterations is bigger. And secondly, the distribution of the successful runs over to the number of iterations required to obtain the solution is roughly independent of the bound of number of iterations. Another interesting characteristic of the graph of Fig. 4 are the spikes near the 6 iteration mark, i.e. CBLS delivers maximum number of successful runs in 5-8 improvements.

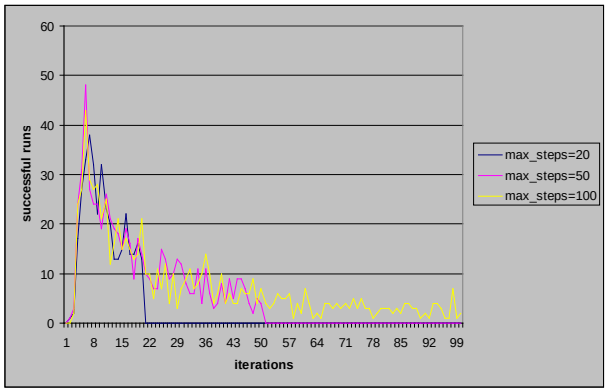


Fig. 4 Distribution of the successful runs over to number of required iterations

6 Other CNP implementations for solving CSPs

Using the CNP implementations presented in Fig. 1 and Fig. 3 and another type of dynamic control (system options and control states) we can easily model other CSP solving algorithms. Table 1 presents their performance on the 8-queens problem and the performance of the already discussed in the previous sections two algorithms (columns 2 and 5).

Table 1

1	2	3	4	5	6
FC	**FCs&d**	**FCs**	**FCrnd**	**CBLSmc**	**CBLSrnd**
88	8	8	(25)	(10)	(13)

The algorithms from left to right, are simple forward checking algorithm (**FC)**, forward checking with static and dynamic variable and value ordering heuristics (**FCs&d**), forward checking with only static variable and value ordering heuristics **(FCs)**, forward checking with random variable and value ordering (**FCrnd)**, CBLS with most conflicted variable chosen **(CBLSmc)** and CBLS with randomly chosen conflicted variable **(CBLSrnd)**. Each cell is the number of consistency checks required to solve the problem. For the algorithms with elements of randomness (the last three in the table) this is the mean number of checks (marked in parentheses) over 1000 runs with a bound of the number of iterations 20.

6.1 Forward Checking algorithm with only static variable and value ordering heuristics (degree and static LCV; Table 1, column 3)

Static versions of the discussed heuristics require variable and value pre-ordering (before the search). With our model from Fig.1, that can only be achieved by deleting the rearranging arrows option RANGEORDER. This way the default order of the outgoing arrows from **Rows** and **Cols** states is only meaningful which lead to ordering rows from “inside out” and columns from “outside-in”. The segment of the modified CN that corresponds to **Rows** and **Cols** is shown in the Fig.5. The experiments show that this implementation finds the solution in 8 steps, i.e. without any backtracks. This result proves the thesis that a dynamic ordering is not necessarily better than a static ordering [13]. In 8-queens problem the static heuristics work a lot better than the dynamic ones because the dynamic heuristics return the same values in most cases.

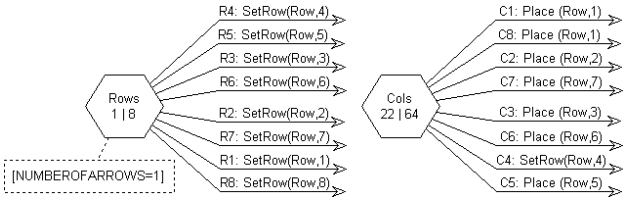


Fig. 5 CNP implementation of FCs for 8-queens problem

6.2 Forward Checking algorithm with random variable and value ordering (Table 1, column 4)

Sometimes it is useful to randomize the variable and value selection procedure [17, 19]. The importance of introducing randomness generally in computation theory is discussed in [20]. The CNP implementation of stochastic variable and value choice could be easily achieved if we modify the CN from Fig.1 replacing the option [RANGEORDER=LOWFIRST] with [RANGEORDER=RANDOM] for both **Rows** and **Cols** control states (see Fig.6). Generated heuristic evaluations won't be used in ordering rows and columns and the outgoing arrows from the corresponding control states will be attempted in random order. The performance of this algorithm is worse than the performance of the previous ones. It finds the solution in an average of 25 steps.

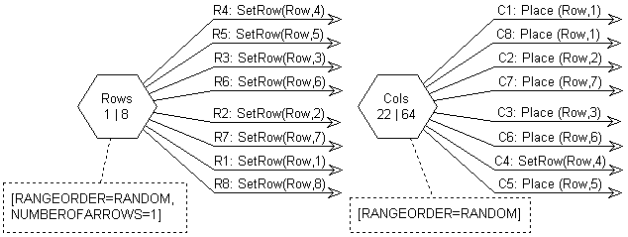


Fig. 6 CNP implementation of FCrnd for 8-queens problem

6.3 Forward Checking algorithm (Table 1, column 1)

A simple backtracking strategy for solving the 8-queens problem can be performed in the following way. Rows and columns are looked at one at a time in numerical order. This algorithm with incorporated lookahead technique (forward checking algorithm) could be simulated in CNP using CN from Fig.1 and deleting the rearranging arrows option RANGEORDER. The outgoing arrows from the control states **Rows** and **Cols** must be set in numerical order, i.e. from 1 to 8. The modified **Rows** and **Cols** are shown in the Fig.7. This algorithm finds the first solution in 88 steps.

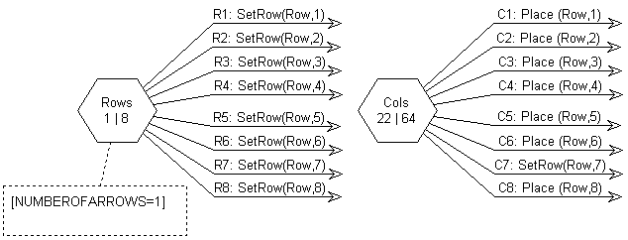


Fig. 7 CNP implementation of FC for 8-queens problem

6.4 CBLS algorithm with randomization (Table 1, column 6)

The algorithm of Fig. 2 where the variable is randomly chosen from all the conflicted variables could be implemented in SPIDER easily using the CNP implementation from Fig.3 which simulates a LS algorithm with most conflicted variable chosen. The only change that must be performed concerns the control state **Rows** of type SELECT. Now it should be stated of type RANGE with low selector 1 and high selector 7, determining minimum and maximum number of conflicts which a queen is involved in. This way the unattacked queens (those with value 0) are rejected for examination. Random choice from the conflicted queens is performed by the system option [RANGEORDER=RANDOM]. As the search process in LS is determined another system option must be used - [NUMBEROFARROWS=1]. These changes are depicted on Fig.8.

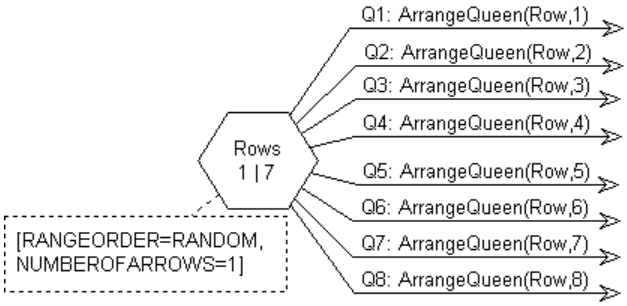


Fig. 8 CNP implementation of CBLSrnd for 8-queens problem

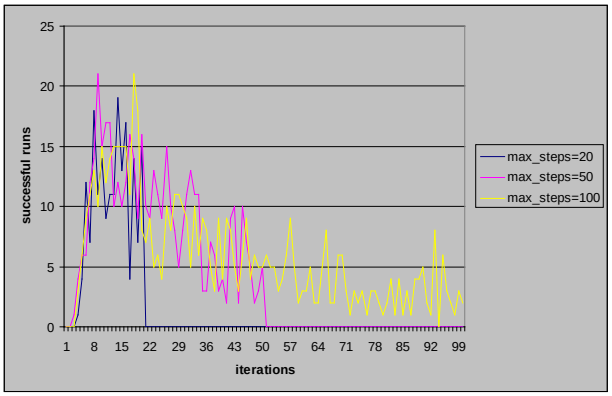


Fig. 9 Distribution of the successful runs over to number of required iterations for the CBLS model with random chosen conflicted variable

The same experiment like that for the previous CBLS model (Section 5) was performed - the algorithm was run 1000 times with a bound of the number of iterations 20, 50 and 100. At

max_steps=20 the successful runs are 19% (in an average of 10 steps), at **max_steps**=50 they are 44%, and at **max_steps**=100 - 59%. Again, the frequency of occurrence of the successful runs decreases with the increment of the number of required iterations (see the Fig.9). But the successful runs for the algorithm discussed are more evenly distributed as it could be seen at Fig.10 which compares the performance of the two algorithms for **max_steps**=20.

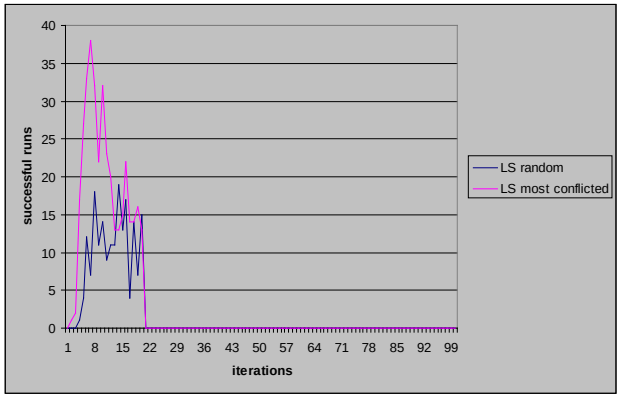


Fig. 10 Comparison of the distributions of the successful runs for the presented CBLS models

6.5 Number of solutions

SPIDER system option - SOLUTIONS is used for setting the solution scope. The value of the option determines the number of solutions that are looked for. The presented FC implementations find just the first one solution to the 8-queens problem, setting the [SOLUTIONS=1]. If we are trying to find all the 92 solutions the [SOLUTIONS=ALL] must be specified. This will cause a full traversing of the control network.

7 CNP (SPIDER language) and Constraint Programming Languages

CNP and constraint programming are fundamentally different but at the same time they have interesting similarities. This section explores (not in depth) the differences and comparing the features the two approaches offer.

1) Constraint satisfaction programming languages are used to encode and solve only constraint satisfaction/optimization problems.

On the other side CNP is a universal programming paradigm and this was illustrated in [9] through solutions to selected representative applications, but it is a style of high-level programming created to be especially convenient for solving problems with natural graph-like representation.

2) The computational model of constraint-programming languages and platforms typically employs the various constraint propagation techniques and handles the backtracking, while the choice for variables and values is left to a user specified search procedure.

SPIDER search engine is based on backtracking too, but it provides a lot of static and dynamic tools to control the search and this way to incorporate various heuristics. As it was shown the choice for variables and values in solving CSPs are easily implemented by the wide set of system options and control states.

3) Constraint programming is a form of declarative programming, because constraints don't specify a step or sequence of steps to execute, but rather the properties of a solution to be found.

CNP is a declarative style of programming too, because the problem is specified in the form of graph and there is a built-in inference, searching a path (solution of a problem) in the graph.

4) Constraint programming is an embedding of constraints in a host language. It has been established that constraints can be mixed with the following programming paradigms: logic programming, functional programming and imperative programming. Constraints are usually integrated into a programming language or provided via separate software libraries. The first approach is implemented in program systems *(Prolog III, ECLiPSe, Oz, Kaleidoscope, Comet)* which unfortunately aren't in the top 50 of the most usable tools for software development, according to The TIOBE Programming Community Index [23] for example. They have insufficiently good maintenance and outdated versions. The second approach (libraries) supposes limited functionality and difficult communication between both paradigms.

Technically, the SPIDER program is integrated in the imperative programming language project even at the level of source code. Hereby, a two-way connection between paradigms and an access to all common recourses are achieved.

5) In many cases, the innate structure of a problem to handle is not linear. For example, the primary, natural description of a problem might take the form of a tree, a graph (network), or a recursive set of networks. It would be a great advantage if there was no need for the 'programmer' to try to translate an inherently graph-like, possibly nondeterministic, possibly declarative description

into a much more complicated and difficult to understand sequential algorithmic model of this description. In fact, because of its naturalness, the human form will be most probably the most consistent and easily verifiable representation. This imposes the requirement that a programming language to be natural, i.e. similar to the manner in which people think of problems and the style in which they specify them informally. As a matter of fact, CNP has been created with exactly that goal in mind. Furthermore, being a declarative description of the problem, the CN is a graphical specification of that declarative representation, which is more natural and intuitive than the declarative representation implemented by instructions.

In solving CSPs, the natural description of choice points is graphical, specifying a limited number of alternatives, but the program must later choose between them. According to this a shortcoming of specifications in constraint programming languages like COMET is that they are less natural than those in CNP. To observe this, compare the equivalent specifications of the 8-queens problem in the two languages. Figure 11 shows part of that COMET specification, which is equivalent to the choice points of variables and its values. While these choice points are specified in CNP graphically through control states with outgoing arrows, in COMET they are implemented as instructions (*forall* and *tryall*)

```
forall(i in Size) by (queen[i].getSize())
tryall<m>(v in Size : queen[i].memberOf(v))
label(queen[i], v);
```

Fig. 11 The search procedure of n-Queens problem in COMET

8 Conclusion

As a new programming paradigm, CNP builds on fundamental research in programming paradigms [7, 10]. It integrates ideas from imperative programming, declarative programming, rule-based systems, nondeterministic programming and graphical programming.

The most prominent usage of the tools for dynamic search control in CNP is for automatic, declarative implementation of various heuristics in search algorithms. A wider view at the approaches to implementing various search strategies in CNP is the subject of [21]. The questions of what search techniques for solving CSPs are well suited for such an elegant declarative CNP implementation and how to specify (to program) such a strategy have been targeted in this paper.

The resulting programs are intuitive and natural and can be used as an illustration of the main concepts and techniques in solving CSPs. Another important benefit of the proposed approach is its flexibility. It is easy to modify an existing implementation, thus to experiment with different heuristics obtaining a wide variety of search algorithms without affecting the problem modeling.

References:

[1] Kratchanov, K., Golemanov, T., Golemanova, E. and Ercan, T.: Control Network Programming with SPIDER: Dynamic Search Control, *In: 14th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2010)*, Cardiff, UK (2010)

[2] Golemanova, E., Golemanov, T., Kratchanov, K.: "Built-in Features of the SPIDER Language for Implementing Heuristic Algorithms", *In: Proc. CompSysTech 2000*, Sofia, June 2000, II.9-1 – II.9-5 (in Bulgarian). Also published by ACM Press, 2091-2095.

[3] Kratchanov, K., Golemanova, E., Golemanov, T. and Ercan, T., "Nonprocedural Implementation of Local Heuristic Search in Control Network Programming", *In: 14th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2010)*, Cardiff, UK, 2010.

[4] Kratchanov, K., Golemanova, E., Golemanov, T. and Ercan, T.: Procedural and Non-Procedural Implementation of Search Strategies in Control Network Programming. *In: Intl Symposium on Innovations in Intelligent Systems and Applications (INISTA 2010)*, Kayseri & Cappadocia, Turkey (2010)

[5] Tsang, E.: *A Glimpse of Constraint Satisfaction.* Artificial Intelligence Review, Kluwer Academic Publishers, Printed in the Netherland, 13: 215–227, 1999

[6] Bell, J., Stevens, B.: A survey of known results and research areas for n-queens. *Discrete Mathematics*, Volume 309, Issue 1, 2009, pp 1-31

[7] Kratchanov, K., Golemanov, T., Golemanova, E.: Control Network Programming, *In: Proc. 6th IEEE/ACIS Conf. on Computer and Information Science (ICIS 2007)*, July 2007, Melbourne, Australia, 1012-1018.

[8] Control Network Programming web site: http://controlnetworkprogramming.com

[9] Kratchanov, K., Golemanova, E., Golemanov, T.: Control Network Programming Illustrated: Solving Problems With Inherent Graph-Like Structure, *In: Proc. 7th IEEE/ACIS Conf. on Computer and Information Science (ICIS 2008)*, May 2008, Portland, OR, USA, 453-459.

[10] Kratchanov, K., Golemanova, E., Golemanov, T.: Control Network Programs and Their Execution, *In: 8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)*, Cambridge, UK, pp. 417–422. WSEAS Press, 2009.

[11] Tsang, E.: *Foundations of Constraint Satisfaction.* Academic Press, 1996

[12] Russell, S, and Norvig, P.: *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall, Upper Saddle River, NJ, 2010

[13] Run, P.: Domain Independent Heuristics in Hybrid Algorithms for CSP's. Masters thesis. University of Waterloo, Ontario, Canada, 1994.

[14] Cheadle, A., et al.: ECLiPSe: A Tutorial Introduction: http://www.eclipseclp.org/doc/tutorial/tutorial088.html

[15] Simonis, H. ECLiPSe ELearning Website: http://4c.ucc.ie/~hsimonis/ELearning/nqueen/slides.pdf

[16] The Cork Constraint Computation Centre (4C). CSP tutorial: http://4c.ucc.ie/web/outreach/tutorial.html

[17] Rossi, F, Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier. 2006

[18] Wallac, M. Constraint Programming: http://eclipseclp.org/reports/handbook/handbook.html

[19] Müller,T.: Interactive Heuristic Search Algorithm, *In Proceedings of the CP'02 Conference - Doctoral Programme*, 2002

[20] Hromkovic, J.: *Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography*, Springer, Berlin (2004)

[21] Kratchanov, K., Golemanova, E, Golemanov, T. and Gökçen, Y.: Implementing Search Strategies in Winspider II: Declarative, Procedural, and Hybrid Approaches. *In: Stanev, I. and K. Grigorova (eds.): Knowledge-Based Automated Software Engineering*, Cambridge Scholars Publ., pp. 115-135 (2012)

[22] Kalé L.V.: An almost perfect heuristic for the n nonattacking queens problem, *Information processing letters*, vol. 34, no. 4, 1990, pp. 173-178

[23] TIOBE Programming Community Index http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html