IN-63-TM

91570

P-27

The Min-Conflicts Heuristic: Experimental and Theoretical Results

STEVE MINTON
ANDREW B. PHILIPS
MARK D. JOHNSTON
PHILIP LAIRD

AI RESEARCH BRANCH, MAIL STOP 244-17
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94035

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA–91–25

September 1991

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE Dates attached	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Titles/Authors - Attached	5. FUNDING NUMBERS
6. AUTHOR(S)	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Code FIA - Artificial Intelligence Research Branch Information Sciences Division	8. PERFORMING ORGANIZATION REPORT NUMBER Attached
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Nasa/Ames Research Center Moffett Field, CA. 94035-1000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Available for Public Distribution [illegible] 5/14/92 BRANCH CHIEF	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstracts ATTACHED

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

The Min-Conflicts Heuristic: Experimental and Theoretical Results

Steven Minton[1] **Andrew B. Philips**[1] **Mark D. Johnston**[2] **Philip Laird**[3]

[1]Sterling Federal Systems
NASA Ames Research Center
AI Research Branch
Mail Stop: 244-17
Moffett Field, CA 94035 USA

[2]Space Telescope Science Institute
3700 San Martin Drive,
Baltimore, MD 21218 USA

[3]NASA Ames Research Center
AI Research Branch
Mail Stop: 244-17
Moffett Field, CA 94035 USA

Abstract

This paper describes a simple heuristic method for solving large-scale constraint satisfaction and scheduling problems. Given an initial assignment for the variables in a problem, the method operates by searching though the space of possible repairs. The search is guided by an ordering heuristic, the *min-conflicts heuristic*, that attempts to minimize the number of constraint violations after each step. We demonstrate empirically that the method performs orders of magnitude better than traditional backtracking techniques on certain standard problems. For example, the one million queens problem can be solved rapidly using our approach. We also describe practical scheduling applications where the method has been successfully applied. A theoretical analysis is presented to explain why the method works so well on certain types of problems and to predict when it is likely to be most effective.

1 Introduction

One of the most promising general approaches for solving combinatorial search problems is to generate an initial, suboptimal solution and then to apply local *repair* heuristics. Techniques based on this approach have met with empirical success on many problems, including the traveling salesman and graph partitioning problems[12]. Such techniques also have a long tradition in AI, most notably in problem-solving systems that operate by debugging initial solutions [22, 24]. This idea can be extended to constraint satisfication problems in a straightforward manner. Our method takes an initial, inconsistent assignment for the variables in a constraint satisfaction problem (CSP) and incrementally repairs constraint violations until a consistent assignment is achieved. The method is guided by a simple ordering heuristic for repairing constraint violations: select a variable that is currently participating in a constraint violation, and choose a new value that minimizes the number of outstanding constraint violations.

The work described in this paper was inspired by a surprisingly effective neural network developed by Adorf and Johnston for scheduling the use of the Hubble Space Telescope[2, 14]. Our heuristic CSP method was distilled from an analysis of the network, and has the virtue of being extremely simple. It can be implemented very efficiently within a symbolic CSP framework, and combined with various search strategies. This paper includes empirical studies showing that the method performs extremely well on some standard problems, such as the n-queens problem, to the extent that the method can quickly find solutions to the one million queens problem. We also describe initial work on large-scale scheduling applications which suggests that the method has important practical implications as well. The final contribution of this paper is a theoretical analysis that describes how various problem characteristics affect the performance of the method.

2 Previous Work: The GDS Network

By almost any measure, the Hubble Space Telescope scheduling problem is a complex task [26, 21, 13]. Between ten thousand and thirty thousand astronomical observations per year must be scheduled, subject to a vast variety of constraints involving time-dependent orbital characteristics, power restrictions, priorities, movement of astronomical bodies, stray light sources, etc. Because the telescope is an extremely valuable resource with a limited lifetime, efficient scheduling is a critical concern. An initial scheduling system, developed in FORTRAN using traditional programming methods, highlighted the difficulty of the problem; it was estimated that it would take over three weeks for the system to schedule one week of observations. A more successful constraint-based system was then developed to augment the original system. At its heart is a neural network developed by Johnston and Adorf, the Guarded Discrete Stochastic (GDS) network, which searches for a schedule[2, 14].

From a computational point of view, the network is interesting because Johnston and Adorf found that it performs well on a variety of tasks, in addition to the space telescope scheduling problem. For example, the network performs significantly better on the n-queens problem than previous heuristic methods. The n-queens problem requires placing n queens on an $n \times n$ chessboard so that no two queens share a row, column or diagonal. The network has been used to solve problems of up to 1024 queens, whereas previous methods discussed in the literature[23] encounter difficulties with problems that are ten times smaller. Later in this paper we describe how our analysis of the GDS network enabled us to build a simple heuristic algorithm that performs even better.

The GDS network is a modified Hopfield network[10]. The most significant modification is that the main network is coupled asymmetrically to an auxiliary network of *guard neurons* which restricts the configurations that the network can assume. This modification enables the network to rapidly find a solution for many problems, even when the network is simulated on a serial machine. The disadvantage is that convergence to a stable configuration is no longer guaranteed. Thus, the network can fall into a local minimum involving a group of unstable states among which it will oscillate. In practice, however, if the network fails to converge after some number of neuron state transitions, it can simply be stopped and started over. [1]

To illustrate the network architecture and updating scheme, let us consider how the network is used to solve binary constraint satisfaction problems. A problem consists of n variables, $X_1 \dots X_n$, with domains $D_1 \dots D_n$, and a set of binary constraints. Each constraint $C_\alpha(X_j, X_k)$ is a subset of $D_j \times D_k$ specifying incompatible values for a pair of variables.[2] To solve a CSP using the network, each variable is represented by a separate set of neurons, one neuron for each of the variable's possible values. Each neuron is either "on" or "off", and in a solution state, every variable will have exactly one of its corresponding neurons "on", representing the value of that variable. Constraints are represented by inhibitory (i.e., negatively weighted) connections between the neurons. To insure that every variable is assigned a value, there is a guard neuron for each set of neurons representing a variable; if no neuron in the set is on, the guard neuron will provide an excitatory input that is large enough to turn one on. (Due to the way the connection weights are set up, it is unlikely that the guard neuron will turn on more than one neuron.) The network is updated on each cycle by randomly picking a set of neurons that represents a variable, and flipping the state of the neuron in that set whose input is *most inconsistent* with its current output (if any). When all neurons' states are consistent with their input, a solution is achieved.

3 The Min-Conflicts Heuristic

Why does the GDS network perform so much better than traditional backtracking methods on tasks such as the n-queens? In addressing this question, we began with a number of competing hypotheses (some of which were suggested in [2]). For instance, one hypothesis

[1]The emphasis in Johnston and Adorf's work is to produce a computational architecture that can efficiently solve CSP problems, as opposed to modeling cognitive or neural behavior. Our discussion necessarily ignores many aspects of Johnston and Adorf's work, which is described in detail elsewhere[14, 2].

[2]This paper only considers the task of finding a single solution, that is, finding an assignment for each of the variables which satisfies the constraints.

was that the systematic nature of the search carried out by backtracking is the source of its problems, as compared to the stochastic nature of the search carried out by the network. Specifically, if solutions in the backtracking space are clustered together (with correspondingly high inter-cluster distances), then a completely randomized search of the space can be more effective than systematic backtracking. However, although tasks such as n-queens are in fact solved more efficiently using randomized algorithms (such as Las Vegas algorithms [4]), our studies indicate that the performance of the GDS network is far too good to be explained by this hypothesis.

As it turns out, the key to the network's performance appears to be that when it chooses a neuron to update, it chooses the neuron whose state is most inconsistent with its input. Thus, from a constraint satisfaction perspective, the network will "deassign" a variable's current value only if it is inconsistent with other variables. Furthermore, when a new value is later assigned, the network will choose the value that minimizes the number of other variables with which it is inconsistent. This idea is captured by the following heuristic:

> **Min-Conflicts Heuristic:**
> *Given:* A set of variables, a set of binary constraints, and an assignment specifying a value for each variable. Two variables *conflict* if their values violate a constraint.
> *Procedure:* Select a variable that is in conflict, and assign it a value that minimizes the number of conflicts.[3] (Break ties randomly.)

We have found that the network's behavior can be approximated by a symbolic system that uses the min-conflicts heuristic for hill-climbing. The hill-climbing system starts with an initial assignment generated in a preprocessing phase. At each choice point, the heuristic chooses a variable that is currently in conflict and reassigns its value, until a solution is found. The system thus searches the space of possible assignments, favoring assignments with fewer total conflicts. Of course, the hill-climbing system can become "stuck" in a local maximum, in the same way that the network may become "stuck" in a local minimum. In the next section we present empirical evidence to support our claim that the min-conflicts heuristic is responsible for the network's effectiveness.

One of the virtues of extracting the heuristic from the network is that the heuristic can be used with a variety of different search strategies in addition to hill-climbing. For example, we have found that informed backtracking can be an effective strategy when used in the following manner. Given an initial assignment generated in a preprocessing phase (as described below), an informed backtracking program employs the min-conflicts heuristic to order the choice of variables and values to consider, as described in figure 1. Initially the variables are all on a list of VARS-LEFT, and as they are repaired, they are pushed onto a list of VARS-DONE. The program attempts to find a sequence of repairs, such that no variable

[3]In general, the heuristic attempts to minimize the number of other variables that will need to be repaired. For binary CSPs, this corresponds to minimizing the number of conflicting variables. For general CSPs, where a single constraint may involve several variables, the exact method of counting the number of variables that will need to be repaired depends on the particular constraint. The space telescope scheduling problem is a general CSP, whereas most of the other tasks described in this paper are binary CSPs.

```
 Procedure INFORMED-BACKTRACK (VARS-LEFT VARS-DONE)
  If all variables are consistent,
    then solution found, STOP.
  Let VAR = a variable in VARS-LEFT
    that is in conflict.
  Remove VAR from VARS-LEFT.
  Push VAR onto VARS-DONE.
  Let VALUES = list of possible values for VAR
    ordered in ascending order according to number
    of conflicts with variables in VARS-LEFT.
  For each VALUE in VALUES, until solution found:
    If VALUE does not conflict with any variable
         that is in VARS-DONE, then
      Assign VALUE to VAR.
      Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
    end if
  end for
 end procedure

Begin program
 Let VARS-LEFT = list of all variables,
                 each assigned an initial value.
 Let VARS-DONE = nil
 Call INFORMED-BACKTRACK(VARS-LEFT VARS-DONE)
End program
```

Figure 1: Informed Backtracking Using the Min-Conflicts Heuristic

is repaired more than once. If there is no way to repair a variable in VARS-LEFT without violating a previously repaired variable (a variable in VARS-DONE), the program backtracks.

It should be clear that the informed backtracking algorithm is simply a basic backtracking algorithm augmented with the min-conflicts heuristic to order its choice of value bindings for a variable. This illustrates an important point. The informed backtracking program incrementally extends a consistent partial assignment (i.e., VARS-DONE), in the same manner as a basic backtracking program, but in addition, uses information from the initial assignment (i.e., VARS-LEFT) to bias its search. The next section documents the degree to which this information is useful.

4 Experimental Results

This section has two purposes. First, we evaluate the performance of the min-conflicts heuristic on some standard tasks using a variety of search strategies. Second, we show that the heuristic, when used with a hill-climbing strategy, approximates the behavior of the GDS network.

We have employed the following search strategies with the min-conflicts heuristic:

1. Hill-climbing: This strategy most closely replicates the behavior of the GDS network. The disadvantage is that a hill-climbing program can get caught in local maxima, in which case it will not terminate.
2. Informed backtracking: As described earlier, this strategy is a basic backtracking strategy, augmented with the min-conflicts heuristic for ordering the assignment of variables and values. Because the min-conflicts heuristic repairs the initial assignment, it can also be viewed as backtracking in the space of possible repairs. One advantage of this strategy is that it is complete – if there is a solution, it will eventually be found; if not, failure will be reported. Unfortunately, this is of limited significance for large-scale problems because terminating in a failure can take a very long time. A second advantage is that the strategy can be augmented with pruning heuristics which cut off unpromising branches. This can be very useful, as documented in the next section.
3. Best-first search: This strategy keeps track of multiple assignments (each corresponding to a leaf in the search tree). On each cycle it picks the assignment with the fewest constraint violations and considers the set of repairs that can be applied to that assignment. We have found that best-first search (of which A^* is one variation) is generally expensive to employ on large-scale problems due to the cost of maintaining multiple assignments.

4.1 The N-Queens Problem

The n-queens problem, originally posed in the 19th century, has become a standard benchmark for testing backtracking and CSP algorithms. In a sense, the problem of finding a single solution was "solved" in the 1950's by the discovery of a pair of patterns that can be instantiated in linear time to yield a solution[1]. Nevertheless, the problem has remained relatively "hard" for heuristic search methods. Several studies of the n-queens problem [23, 9, 16] have compared heuristic backtracking methods such as search rearrangement backtracking (e.g., most-constrained first), forward checking, dependency-directed backtracking, etc. However, no previously identified heuristic search method has been able to consistently solve problems involving hundreds of queens within a reasonable time limit.[4]

On the n-queens problem, Adorf and Johnston [2] reported that the probability of the GDS network converging increases with the size of the problem. For large problems, e.g., $n > 100$ (where n is the number of queens), the network almost certainly converges. Moreover, the median number of cycles required for convergence is only about $1.16n$. Since it takes $O(n)$ time to execute a transition (i.e., picking a neuron and updating its connections), the expected time to solve a problem is (empirically) approximately $O(n^2)$. The network has been used to solve problems with as many as 1024 queens, which takes approximately

[4]In a study that was published independently, subsequent to the submission of the original paper on this work, Kale [15] reports on a very different heuristic specifically designed for N-Queens that also works extremely well.

11 minutes in Lisp on a TI Explorer II. For larger problems, memory becomes a limiting factor because the the network requires approximately $O(n^2)$ space. (Although the number of non-zero connections is $O(n^3)$, some connections are computed dynamically rather than stored).

To compare the network with our min-conflicts approach, we constructed a hill-climbing program that operates as follows. An initial assignment is created in a preprocessing phase using a greedy algorithm that iterates through the rows, placing each queen on the column where it conflicts with the fewest previously placed queens (breaking ties randomly). In the subsequent repair phase, the program keeps repairing the assignment until a solution is found. To make a repair, the program selects a queen that is in conflict and moves it to a different column in the same row where it conflicts with the fewest other queens (breaking ties randomly). Interestingly, we found that this program performs significantly better than the network. For $n \geq 100$ the program has never failed to find a solution. Moreover, the required number of repairs appears to remain *constant* as n increases. After further analysis, however, we found the hill-climbing program performs better than the network because the hill-climbing program's preprocessing phase invariably produces an initial assignment that is "close" to a solution, in that the number of conflicting queens in the initial assignment grows extremely slowly (from a mean of 3.1 for $n = 10$ to a mean of 12.8 for $n = 10^6$). Once this difference was eliminated, by starting the network in an initial state produced by our preprocessing algorithm, the network and the hill-climbing program performed quite similarly. We note, however, that the network requires $O(n^2)$ space, as compared to the $O(n)$ space required by the hill-climbing program, which prevented us from running very large problems on the network.

Because the initial assignment had an effect on the number of steps necessary to solve n-queens problems on the net, we decided to explore how different preprocessing methods changed the mean number of conflicts created in the initial assignment. Several methods for producing an initial assignment were examined, and three of these methods are compared below. All of the initialization methods assign one queen per row. Consequently, only columns and diagonals can contain more than one queen after intialization.

	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Conflicts after initialization	3.11	7.35	9.75	10.96	12.02	12.80

Table 1: Number of Conflicts for N-Queens Algorithms

In the first method each queen is placed in a randomly chosen column. Experimentally, the number of queens involved in conflicts was found to be approximately .9N. This can be attributed to the geometric properties of the problem. Repair strategies employing the min-conflicts heuristic then took about .6N steps to find a solution. Note that even with 90% of the variables in conflict, the heuristic works extremely well; a solution is found in a linear number of steps.

In the second initialization method the system puts one queen in each column, with preference given to columns that have no conflicts. To accomplish this, the system maintains a list of empty columns. When placing a queen in a given row, the system examines the list of empty columns, looking for a position with no conflicts (i.e., a position with no conflicts along the diagonals). If more than one column is found, the system selects among them randomly. If none are found, the system randomly selects a position from the list of empty columns. This method performs significantly better than the first initialization method because the number of conflicts produced grows very slowly. However, it does a bit worse than the initialization method used in the experiments described above, which involves a slight variation. In this third method, an initial assignment is created using a greedy algorithm that iterates through the rows, placing each queen on the column where it conflicts with the fewest previously placed queens (breaking ties randomly). Table 1 shows number of conflicts as N becomes large.

Strategy	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Basic Backtrack†	53.8	4473 (70%)	88650 (13%)	*	*	*
Most Constrained Backtrack†	17.4	687 (96%)	22150 (81%)	*	*	*
MinConflicts Hill-Climbing‡	57.0	55.6	48.8	48.5	52.8	48.3
MinConflicts Backtrack‡	46.8	25.0	30.7	27.5	27.8	26.4

† = number of backtracks, ‡ = number of repairs
$*$ = exceeded computational resources (100 runs required > 12 hours on a SPARCstation1)

Table 2: Number of Backtracks/Repairs for N-Queens Algorithms

Table 2 compares the efficiency of our hill-climbing program and several backtracking programs. Each program was run 100 times for n increasing from 10 to one million. Each entry in the table shows the mean number of queens moved, where each move is either a backtrack or a repair, depending on the program. A bound of $n \times 100$ queen movements was employed so that the experiments could be conducted in a reasonable amount of time; If the program did not find a solution after moving $n \times 100$ queens, it was terminated and credited with $n \times 100$ queen movements. For the cases when this occurred, the corresponding table entry indicates in parentheses the percentage of times the program completed successfully. The first row shows the results for a basic backtracking program. For $n \geq 1000$, the program was completely swamped. The second row in the table shows the results for informed backtracking using the "most-constrained first" heuristic. This program is a basic backtracking program that selects the row that is most constrained when choosing the next row on which to place a queen. In an empirical study of the n-queens problem, Stone and Stone [23] found that this was by far the most powerful heuristic for the n-queens problem out of several described earlier by Bitner and Reingold[3]. The program exhibited highly variable behavior. At $n = 1000$, the program found a solution on only 81% of the runs, but three-quarters of these successful runs required fewer than 100 backtracks. Unfortunately, for $n > 1000$, one hundred runs of the program required considerably more than 12 hours on a SPARCstation1, both because the mean number of backtracks grows rapidly and because the "most-constrained first" heuristic takes $O(n)$ time to select the next row after each backtrack. Thus we were prevented from generating sufficient data for $n > 1000$. The next row in the table shows the results for hill-climbing using the min-conflicts heuristic. As discussed above, this algorithm performed extremely well, requiring only about 50 repairs regardless of problem size. The final row shows the results for an informed backtracking program that used the min-conflicts heuristic as described in the previous section. We augmented this program with a pruning heuristic that would initiate backtracking when the number of constraint violations along a path began to increase significantly. However, for $n \geq 100$, this program never backtracked (i.e., no queen had to be repaired more than once). The results are better than those for the hill-climbing program (although there is little room for improvement) primarily because the hill-climbing program tends to repair the same queen again and again.

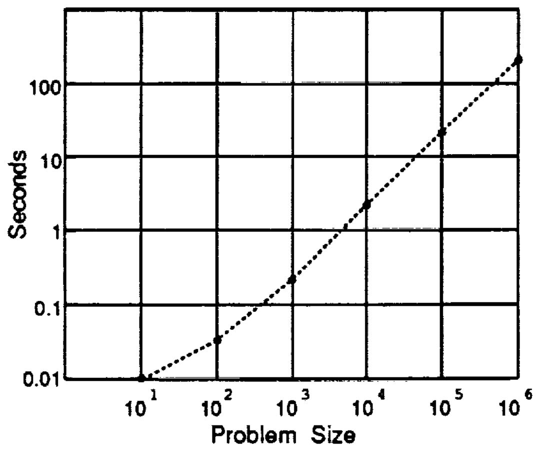

Figure 2: Mean Solution Time for Hill-Climbing Program on N-Queens Problem

We note that for the two programs using the min-conflicts heuristic, each repair requires $O(n)$ time in the worst case. However, this is a relatively minor price to pay. Since the number of repairs remains approximately constant as n grows, the average runtime of the program is approximately linear. This is illustrated by figure 2, which shows the average runtime for the hill-climbing program. In terms of realtime performance, this program solves the million queens problem in less than four minutes on a SPARCstation1.

This program can also be optimized for large problems, in which case the solution time is less than a minute and a half. Specifically, in the repair phase a two step process is used to find the position with the fewest conflicts. The first step checks to see if there are any positions with zero-conflicts. To accomplish this, a set of empty columns is maintained (note that we already have this set of empty columns from the initialization phase of the algorithm). With this set, it is no longer necessary to search the entire row for a zero-conflict position. Only positions that are in an empty column could possibly qualify. This set of empty columns is very small compared to the total number of columns and can be scanned for a suitable candidate in much less time.

Second, if there are no positions with zero conflicts, the program searches for a position that has only one conflict. If we assume a homogeneous distribution of queens, the probability of a column being a one-conflict position is roughly the same as the probability that both diagonals are open, which is $1/2 \times 1/2$ or $1/4$. Since we expect approximately $1/4$ of the columns to be one-conflict positions, the program can randomly test positions in the row (with replacement) until a position is encountered that has only a single constraint violation. Since it is very likely we will quickly find such a position, the amount of work here is small compared to the $O(n)$ work involved in identifying all such positions and then randomly choosing one from that set. In practice, the program generally finds a one-conflict position after just a few tries. Since, theoretically, the possibility exists that the program could not terminate, we could stop this random search after a constant number of steps and then search the row completely for a minimum conflicting position. We never bothered to add this to the program, however, since in practice a one-conflict position is quickly found.

4.2 Scheduling Applications

A scheduling problem involves placing a set of tasks on a time line, subject to temporal constraints, resource constraints, preferences, etc. The space telescope scheduling problem, as discussed earlier, is a complex problem on which traditional backtracking and operations research techniques are either inapplicable or perform poorly. The problem can be considered a constraint optimization problem where we must maximize both the number and the importance of the constraints that are satisfied. As mentioned earlier, an initial scheduling system developed without the use of AI techniques highlighted the difficulty of the problem; it was estimated that the system, called SPSS, would take over three weeks to schedule one week of observations. The constraint-based system, SPIKE, that was developed to augment (and partially replace) the initial system has performed quite well using a relatively simple approach.

The input to SPIKE is a set of detailed specifications for exposures that are to be scheduled on the telescope. These exposures are submitted by astronomers whose proposals have been approved by a peer review process. An exposure specification includes a potentially large number of configuration parameters describing how the data is to be taken. Johnston [13] outlines the problem:

> There are a variety of properties and relationships among these exposures that may be specified by the proposer [astronomer]. Their relative order and time separation may be important. Some exposures are designed as calibrations or target acquisitions for others. Some must be executed at specific times, or at specific phases in the case of periodic phenomena. Some are especially sensitive to stray or scattered light. Exposure durations may vary depending on background light intensity. Some exposures must be executed without interruption while others

> can be broken up as needed. In some cases a specific orientation of an instrument aperture is required. Some exposures are conditional on the results of other exposures.
>
> In addition to proposer-specified constraints, there are a large number of other constraints that must be considered when scheduling HST operations. They range from "strict" constraints that cannot be violated under any circumstances, to "good operating practices" that represent scheduling goals. HST is not allowed to point closer than 50° to the sun and 15° to the bright moon. Slewing the telescope is relatively slow (90° in $\sim$ 15 minutes) so it is important to minimize the time spent in maneuvers. Many constraints are a direct result of HST's low orbital altitude (500 km) and consequent 95 minute orbital period. A typical target is occulted by the earth for $\sim$ 40 minutes of each orbit. Up to half the orbits in a day are contaminated for up to $\sim$ 20 minutes by HST's passage through the South Atlantic Anomaly, a high particle density region during which data cannot be collected. Scattered earthlight changes dramatically over the course of an orbit...

The scheduling team at the Space Telescope Science Institute made the problem considerably more tractable by breaking it into two parts: the long-term scheduling problem and the short-term scheduling problem. The long-term problem consists of taking approximately one year's worth of exposures, and dividing them up into "bins" or time segments of a few days length. The short-term problem consists of coming up with a very detailed schedule for a time segment, which can be translated into commands that the telescope can then directly execute. Currently SPIKE handles only the long-term problem. The short-term problem has a quite different nature, because it involves both planning and scheduling. (We use the term planning to refer to the *generation* of a partially-ordered set of activities to achieve a set of goals, and the term scheduling to refer to the process of placing activities on a time line.) The short-term problem requires planning because an exposure may require activities such as warming up or cooling down different instruments on the telescope, pointing maneuvers, communication of data, etc. Currently, the short-term problem is handled by the original SPSS system, however, Muscettola et al. [21] are developing AI planning techniques that will hopefully do a better job. Another possibility is the extension of the SPIKE system so that it can generate a schedule for significantly smaller time buckets. The research reported here may contribute to this goal, by improving the speed of the SPIKE system.

SPIKE operates by taking the exposure specifications prepared by astronomers and validating that they are internally consistent. It then compiles the specifications into a set of constraints, represented as relative temporal relations and "suitability functions". The relative temporal relations specify the relative before/after ordering of tasks, and the maximal/minimal amount of time between tasks. Each suitability function is a function of time whose value represents the desirability of starting an activity at a specified time, as given by the constraint in question. For example, one suitability function may represent the constraint that the telescope should not point near the moon. Thus, the suitability of scheduling an exposure when the target is close to the moon will be low (perhaps zero). Suitability functions are represented internally as piecewise constant functions, enabling combinations of multiple suitabilities to be calculated efficiently.

Because of the uncertainty inherent in some constraints, and also because the grain-size of the time segments may be relatively large, suitability functions are often used to represent the statistical or aggregate desirability of scheduling an exposure during a certain time segment. For example, a particular orbital constraint might state that an exposure must be taken when the telescope is pointing more than 5° from the earth's limb and is in the earth's shadow. The resulting suitability function might indicate, for each time segment, the average amount of time these conditions are satisfied over that segment (which could encompass many orbits). In other words, it would be preferable to schedule the exposure in a time segment in which a relatively high number of such viewing opportunities occur.

Once SPIKE has compiled the astronomers' proposals into a set of constraints, it must search for a good schedule. SPIKE employs a neural network to carry out this search, the Guarded Discrete Stochastic (GDS) network[2, 14]. The GDS network is a modified Hopfield network[10]. The most significant modification is that the main network is coupled asymmetrically to an auxiliary network of *guard neurons* which restricts the configurations that the network can assume. This modification enables the network to rapidly find a solution for many problems, even when the network is simulated on a serial machine. The disadvantage is that convergence to a stable configuration is no longer guaranteed, in which case the network can fall into a local minimum involving a group of unstable states among which it will oscillate. In practice, however, if the network fails to converge after some number of neuron state transitions, it is simply stopped and started over.

To illustrate the network architecture and updating scheme, let us consider how the network is used to solve the HST scheduling problem. Each task to be scheduled (an exposure or block of exposures) is represented by a separate set of neurons, one neuron for each possible time segment in the schedule. Each neuron is either "on" or "off"; if a neuron is "on" it means the task is currently scheduled for that time segment. Inhibitory (i.e., negatively weighted) connections between the neurons are used to indicate hard constraints between tasks, where the suitability of placing two tasks in a certain configuration is zero. To insure that each task is eventually assigned a time segment there is a guard neuron for each set of neurons representing a task; if no neuron in the set is on, the guard neuron will provide an excitatory input that is large enough to turn one on. (Due to the way the connection weights are set up, it is unlikely that the guard neuron will turn on more than one neuron.) The network is updated on each cycle by randomly selecting a set of neurons that represents a task, and flipping the state of the neuron in that set whose input is *most inconsistent* with its current output (if any). When all neurons' states are consistent with their input, a solution is achieved.

The network updating scheme roughly accomplishes the following: If the task is currently in conflict then it is removed from the schedule, and if the task is currently unscheduled then the network schedules it for the time segment that has the fewest constraint violations. Note that the network only represents hard constraints (i.e. it treats suitabilities as zero or one). Soft constraints (where the suitability is between zero and one) are only consulted when there are two or more "least conflicted" places to move a task.

The min-conflicts algorithm has been shown to be at least as effective as the GDS network on representative data sets provided by the Space Telescope Sciences Institute. In effect, the min-conflicts algorithm mimics the behavior of the GDS network. In fact, the algorithm was developed from an analysis of the network's performance. (The two approaches can be parallelized in a similar manner, but currently both are run on serial machines.) In the HST application, the min-conflicts algorithm operates by constructing an initial schedule in a preprocessing phase, and iteratively repairs the schedule until a conflict-free schedule is found (or the process is terminated by a preset iteration bound). Because our analysis of the min-conflicts algorithm showed that a good initial assignment could greatly improve the solution time, we use a greedy algorithm to create an initial schedule, rather than randomly assigning tasks.[5] The greedy algorithm places each task on the schedule, at each point trying to minimize the number of conflicts.

One advantage in using the min-conflicts algorithm, as compared to the GDS network, is that much of the overhead of using the network can be eliminated (particularly the space overhead). Moreover, because the min-conflicts heuristic is so simple, the scheduling program could be quickly coded in C and is extremely efficient. (The scheduling program runs at least an order of magnitude faster than the network, although some of the improvement is due to factors such as programming language differences, which makes a precise comparison difficult.) While this may be regarded as just an implementation issue, we believe that the clear and simple formulation of the method was a significant enabling factor. We are currently experimenting with a variety of different search strategies that can be combined with the min-conflicts heuristic. Although this study is not yet complete, we expect that the improvements in speed we have observed will eventually translate into better schedules, since the search process can explore a larger number of acceptable schedules.

Several minor issues arose when implementing the HST application. First, the algorithm, as specified in section 3, deals with binary constraints. The HST scheduling problem includes non-binary constraints, i.e., constraints that may involve several variables. For example, one constraint bounds the number of tasks that may be scheduled during a given time segment. For general CSPs, the exact method of counting the number of conflicts for an assignment may depend on the particular constraint in question. As it turned out, for the HST application it sufficed to count each violated constraint as a single conflict, even though multiple tasks might be involved in the violation.

A second issue concerns a difference between the GDS network and the min-conflicts algorithm. As described earlier, the network will remove a conflicted task from the schedule and then reschedule the task in two separate steps, which may not occur consecutively. In contrast, the min-conflicts algorithm rearranges tasks on the schedule, rather than removing them and reinserting them later. It appears that this difference is not significant, except perhaps when the schedule is over-constrained (as discussed below).

[5]We discovered the importance of a good initial assignment by analyzing the min-conflicts algorithm, but it has also been shown to hold for the network as well.

4.2.1 The Over-Subscription Problem

The HST scheduling problem can be considered a constraint optimization problem where we must maximize both the number and the importance of the constraints that are satisfied [8, 20]. We note that the telescope is expected to remain highly over-subscribed, in that many more proposals will be submitted than can be accommodated by any schedule. Unfortunately, one difficulty in analyzing the performance of the scheduler is that no clear objective exists for determining the best schedule in such cases. In particular, we would like to maximize both the overall suitability of the schedule and the number of proposals that can be accommodated – no clear policy for evaluating the tradeoff between these two goals has yet been established by the Space Telescope Science Institute.

SPIKE handles the problem in a manner that is a bit ad-hoc, but apparently quite satisfactory to its users. There is, in effect, a pool of tasks that are either unscheduled or in conflict, and SPIKE's network updating scheme is equally likely to select any of these tasks. (Unscheduled tasks will be moved onto the schedule, and tasks that are in conflict will be moved off the schedule.) Thus, the number of unscheduled tasks are likely to remain approximately equal to the number of tasks in conflict. When the algorithm is eventually interrupted (assuming a conflict-free schedule has not been found) tasks that are in conflict can be removed. One of the advantages of the min-conflicts algorithm is that it is relatively easy to try a variety of schemes for dealing with overconstrained problems. We are currently experimenting with two basic approaches. The first is to follow the approach taken by the network (where tasks are removed and later re-inserted), but vary the procedure for removing and inserting tasks. For example, we can alter the probability of choosing an unscheduled task versus an already scheduled task, or bound the number of unscheduled tasks. (If we set the bound to zero, then tasks will never be removed from the schedule, but simply be moved from place to place on the schedule as in the normal case.) Another approach is to use a more principled method for removing conflicting tasks after coming up with an initial schedule, so that only the minimum number of conflicting tasks need to be removed.

4.3 Other Applications

The min-conflicts and/or GDS network have also been tried on a variety of other problems with good (but preliminary) results, including the randomly generated problems described by Dechter and Pearl [6, 2] and conjunctive precondition matching problems[19]. We are currently cataloging the types of applications for which our method works well.

We have also compared the performance of the GDS network and the min-conflicts heuristic on graph 3-colorability, a well-studied NP-complete problem. In this problem, we are given an undirected graph with n vertices. Each vertex must be assigned one of three colors subject to the constraint that no neighboring vertices be assigned the same color. Adorf and Johnston found that the performance of the network depended greatly on the connectivity of the graph. On sparsely connected graphs (with average vertex degree 4) the network performed poorly, becoming caught in local minima with high probability. On densely connected graphs the network converged rapidly to a solution.

We have repeated Adorf and Johnston's experiments with our hill-climbing program, and found similar results. We have also experimented with variations of informed backtracking using the min-conflicts heuristic. Our most effective program is an informed backtracking program that records the assignment with the least conflicts found so far. When the number of backtracks exceeds a (dynamically adjusted) threshold, the search process is restarted using this best assignment. We have found that performance is further improved by adding heuristics for selecting which vertex to repair, and that, as in the n-queens problem, it helps to have a good initial assignment, which can also be produced using additional heuristics. This illustrates the well-known principle that combining multiple heuristics can improve performance significantly.

In this domain, certain heuristic methods are known to produce excellent results. For instance, Brelaz's k-colorability algorithm [5] employs two strong heuristics (forms of "most-constrained first") and it outperforms our informed backtracking algorithm. Turner [25] has shown that this algorithm will optimally color "almost all" random k-colorable graphs without backtracking, so its dominance is not surprising.

4.4 Summary of Experimental Results

For all of the tasks discussed in the previous section, we have found that the behavior of the GDS network can be approximated by hill-climbing with the min-conflicts heuristic. To this extent, we have a theory that explains the network's behavior. Obviously, there are certain practical advantages to having "extracted" the heuristic from the network. First, the heuristic is very simple, and so can be programmed extremely efficiently, especially if done in a task-specific manner. Second, the heuristic can then be used in combination with different search strategies and task-specific heuristics. This is a significant factor for most practical applications.

Insofar as the power of the heuristic is concerned, our experimental results are encouraging. On the n-queens problem the min-conflicts heuristic clearly outperforms heuristics that have previously been investigated. Furthermore, the heuristic has already been applied successfully to real-world scheduling problems.

We have also considered variations of the min-conflicts heuristic, such as repairing the variable that participates in the *most* conflicts first. In some cases, such variations improve the performanace of the algorithm, and in other cases performance is not significantly changed. As long as the heuristic tends to decrease the number of variables that are inconsistent, it appears that our basic results tend to hold.

5 Analysis

The previous section showed that the min-conflicts heuristic is extremely effective on some tasks, such as placing queens on a chessboard, and less effective on other tasks, such as coloring sparsely connected graphs. In this section, we analyze how the parameters of a task influence the effectiveness of the heuristic. Consider a CSP with n variables, where each variable has k possible values. We restrict our consideration to a simplified model where every variable is subject to exactly c binary constraints, and we assume that there is only a single solution to the problem, that is, exactly one satisfying assignment. We address the following question: What is the probability that the min-conflicts heuristic will make a mistake when it assigns a value to a variable that is in conflict? We define a mistake as choosing a value that will have to be changed before the solution is found. We note that for our informed backtracking program, a mistake of this sort may prove fatal, as it may require an exponential amount of search to recover from a mistake.

For any assignment of values to the variables, there will be a set of d variables whose values will have to be changed to convert the assignment into the solution. We can regard d as a measure of distance to the solution. The key to our analysis is the following observation. Given a variable V to be repaired, only one of its k possible values will be good[6] and the other $k-1$ values will be bad (i.e., mistakes). Whereas the good value may conflict with at most d other variables in the assignment, a bad value may conflict with as many as c other variables. Thus, as d shrinks, the min-conflicts heuristic should be less likely to make a mistake when it repairs V. In fact, if each of the $k-1$ bad values has more than d conflicts, then the min-conflicts heuristic cannot make a mistake - it will select the good value when it repairs this variable, since the good value will have fewer conflicts than any bad value.

We can use this idea to bound the probability that the min-conflicts heuristic will make a mistake when repairing variable V. Let V' be a variable related to V by a constraint. We assume that a bad value for V conflicts with an arbitrary value for V' with probability p, independent of the variables V and V'. Consider an arbitrary bad value for V. Let N_b be the total the number of conflicts between this bad value and the values for the other variables. Given the above assumptions, the expected value of N_b is pc, because there are exactly c variables that share a constraint with V, and the probability of a conflict is p. As mentioned above, the min-conflicts will not make a mistake if the number of conflicts N_b for each bad value is greater than d. We can, therefore, bound the probability of making a mistake by bounding the probability that N_b is less than or equal to d.

To bound N_b, we use Hoeffding's inequality, which states that the sum N of n independent, identically distributed random variables is less than the expected value $\bar{N}$ by more than sn only with probability at most e^{-2s^2n}. In our model, N_b is the sum of c potential conflicts, each of which is either 1 or 0, depending on whether there is a conflict. The expected value of N_b is pc. Thus:

$$\Pr(N_b \leq pc - sc) \leq e^{-2s^2c}$$

Since we are interested in the behavior of the min-conflicts heuristic as d shrinks, let us suppose that d is less than pc. Then, with $s = (pc-d)/c$, we obtain:

$$\Pr(N_b \leq d) \leq e^{-2(pc-d)^2/c}$$

[6]Although a variable is in conflict, its current value may actually be the good value. This can happen when the variable with which it conflicts has a bad value. In this paper we have defined the min-conflicts heuristic so that it can choose *any* possible value for the variable, including its initial value.

To account for the fact that a mistake can occur if *any* of the $k-1$ bad values has d or fewer conflicts, we bound the probability of making a mistake on any of them by multiplying by $k-1$:

$$\Pr(mistake) \leq (k-1)e^{-2(pc-d)^2/c}$$

Note that as c (the number of constraints per variable) becomes large, the probability of a mistake approaches zero, if all other parameters remain fixed. This analysis thus offers an explanation as to why 3-coloring densely connected graphs is relatively easy. We also see that as d becomes small, a mistake is also less likely, explaining our empirical observation that having a "good" initial assignment is important. Additionally, we note that the probability of a mistake also depends on p, the probability that a bad value conflicts with another variable's value, and k, the number of values per variable. The probability of a mistake shrinks as p increases or k decreases.

The analysis makes several simplifying assumptions, including the assumption that only a single solution exists. In the n-queens problem, it appears that the number of possible solutions grows rapidly with n [23]. To explain the excellent performance of the min-conflicts heuristic on the n-queens problem, it seems necessary to take this additional fact into account; we note that for n-queens the bounds derived above are relatively weak. (In n-queens, each row is represented by a variable, so that $c = n$, and $p \approx 2.5/n$, since any two rows constrain each other along a column and either one or two diagonals. Therefore, pc remains approximately constant as n grows.)

6 Discussion

The heuristic method described in this paper can be characterized as a *local search* method[12], in that each repair minimizes the number of conflicts for an individual variable. Local search methods have been applied to a variety of important problems, often with impressive results. For example, the Kernighan-Lin method, perhaps the most successful algorithm for solving graph-partitioning problems, repeatedly improves a partitioning by swapping the two vertices that yield the greatest cost differential. The much-publicized simulated annealing method can also be characterized as a form of local search[11]. However, it is well-known that the effectiveness of local search methods depends greatly on the particular task. We are currently comparing the algorithm's performance with alternative techniques on a variety of tasks.

There is also a long history of AI programs that use repair or debugging strategies to solve problems, primarily in the areas of planning and design[24, 22]. These programs have generally been quite successful, although the repair strategies they employ may be domain specific. In comparison, the min-conflicts heuristic is a completely general, domain-independent approach. Of course, any domain-independent heuristic is likely to fail in certain cases, precisely because of its lack of domain-specific expertise.

In fact, it is easy to imagine problems on which the min-conflicts heuristic will fail. The heuristic is poorly suited to problems with a few highly critical constraints and a large number of less important constraints. For example, consider the problem of constructing a four-year course schedule for a university student. We may have an initial schedule which satisfies almost all of the constraints, except that a course scheduled for the first year is not actually offered that year. If this course is a prerequisite for subsequent courses, then many significant changes to the schedule may be required before it is fixed. In general, if repairing a constraint violation requires completely revising the current assignment, then the min-conflicts heuristic will offer little guidance. This intuition is partially captured by the analysis presented in the previous section, which shows how the effectiveness of the heuristic is inversely related to the distance to a solution.

The problems investigated in this paper, especially the n-queens problem, tend to be relatively uniform, in that the likelihood of such critical constraints existing is low. In the space telescope scheduling problem, constraint preprocessing techniques[18] are applied to reduce the likelihood that any particular constraint will be highly critical. For example, by taking the transitive closure of temporal constraints (e.g. the "after" relation) and representing each inferred constraint explicitly, critical constraints can be transformed into sets of constraints. This works well because the min-conflicts heuristic will be less likely to violate a set of constraints than a single constraint. In some cases, we expect that more sophisticated techniques will be necessary to identify critical constraints[7]. To this end, we are currently evaluating abstraction and explanation-based learning techniques that have worked well for planning systems[17, 19].

7 Conclusions

This paper has two primary contributions. First, we have analyzed a very successful neural network algorithm and shown that an extremely simple heuristic is responsible for its effectiveness. Second, we have demonstrated that this heuristic can be incorporated into symbolic CSP programs with excellent results.

8 Acknowledgements

The authors wish to thank Hans-Martin Adorf, Richard Franier, and Don Rosenthal for their assistance on this research project, and Peter Cheeseman, Monte Zweben, John Bresina, Megan Eskey, Mark Drummond, Eric Raymond, Oren Etzioni, Craig Knoblock and Bernadette Kowalski for their comments and advice, and for their company during late-night dinners. The Space Telescope Science Institute is operated by the Association of Universities for Research in Astronomy for NASA.

Appendix A: Hill Climbing with Minimum Conflicts

This piece of Lisp code implements a hill climbing algorithm employing the min-conflicts heuristic. Each queen is assigned to a single row. So, in constraint satisfaction terms, each row is a variable and each column assignment is its value. Note: this program is a simplified, but less efficient, version of the one described in the paper.

Execute the following lisp statements for a sample run of the code solving for the traditional 8 queens problem.

```
 (setq *same-assignment* nil)
 (setq *vars-created* nil)
 (setq *cutoff* 1000)
 (setq *printing* t)

 (minconflicts-hc 8)
 (print-pic)
```

And, now the program...

```
(proclaim '(special *num-times* *nodes* *same-assignment* *vars-created*
                    *printing*  *cutoff* *num-vars-violated*
                    *board* *num-col-elts* *num-up-diag-elts*
                    *num-dn-diag-elts*))


(defun minconflicts-hc (n)
  (setq *num-times* 0)
  (if (not *vars-created*)
      (create-vars n))
  (if (not *same-assignment*)
      (create-assignment n))
  (find-sol n)
  *num-times*)


(defun find-sol (n)
  (loop with row = nil
        with new-col = nil
        with old-col = nil
        with limit = (1- n)
        while (setq row (find-a-violated-row limit))
        do (setq old-col (elt *board* row))
           (setq new-col (find-least-violated-col row old-col limit))
           (sub-queen row old-col limit)
           (add-queen row new-col limit)
           (if *printing* (print-info row old-col new-col))
           (setq *num-times* (1+ *num-times*))
        until (> *num-times* *cutoff*)))


(defun find-a-violated-row (limit)
  (setq *num-vars-violated* 0)
  (loop with row = 0
        with col = 0
        with vio-rows = nil
        with max-vios = 0
        with num-vios = 0
        do (setq col (elt *board* row))
           (setq num-vios (num-preexisting-vios row col limit))
           (cond ((equal 3 num-vios)) ; three = zero violations
                 (t (push row vio-rows)))
        until (> (incf row) limit)
        finally (progn (setq *num-vars-violated* (length vio-rows))
                       (return (and vio-rows (get-a-random vio-rows))))))


(defun find-least-violated-col (row old-col limit)
  (loop with col = 0
        with least-vios-cols = nil
        with min-vios = limit
        with num-vios = 0
        do (cond ((not (eq old-col col))
                  (setq num-vios (num-preexisting-vios row col limit))
                  (cond ((equal num-vios min-vios)
                         (push col least-vios-cols))
                        ((< num-vios min-vios)
                         (setq least-vios-cols nil)
                         (push col least-vios-cols)
                         (setq min-vios num-vios)))))
        until (> (incf col) limit)
        finally (progn (if (eql min-vios 3)
                           (error "no columns are violated"))
                        (return (get-a-random least-vios-cols)))))


(defun create-vars (n)
   (setq *board* (make-array n :element-type 'fixnum :initial-element 0))
   (setq *num-col-elts*
         (make-array n :element-type 'fixnum :initial-element 0))
   (setq *num-up-diag-elts*
         (make-array (1- (* 2 n)) :element-type 'fixnum :initial-element 0))
   (setq *num-dn-diag-elts*
         (make-array (1- (* 2 n)) :element-type 'fixnum :initial-element 0)))


(defun create-assignment (n)
   (loop with limit = (1- n)
         with row = 0
         with cols-left = (make-n limit)
         do (fill-row row limit n cols-left)
            (setq cols-left (remove (elt *board* row) cols-left))
         until (> (incf row) limit)))


(defun fill-row (row limit n columns)
  (loop with col = nil
        with best-col = (get-a-random columns)
        with best-vios = n
        with num-vios = 0
        while (setq columns (remove (setq col (get-a-random columns)) columns))
        do (setq num-vios (+ (elt *num-col-elts* col)
                             (elt *num-dn-diag-elts* (+ limit (- row col)))
                             (elt *num-up-diag-elts* (+ row col))))
           (cond ((< num-vios best-vios)
                  (setq best-vios num-vios best-col col)))
        until (equal 0 best-vios)
        finally (add-queen row best-col limit)))

(defun add-queen (row col limit)
  (setf (elt *board* row) col)
  (setf (elt *num-col-elts* col) (1+ (elt *num-col-elts* col) ))
  (setf (elt *num-dn-diag-elts* (+ limit (- row col)))
        (1+ (elt *num-dn-diag-elts* (+ limit (- row col)))))
  (setf (elt *num-up-diag-elts* (+ row col))
        (1+ (elt *num-up-diag-elts* (+ row col)))))

(defun sub-queen (row col limit)
  (setf (elt *board* row) 0)
  (setf (elt *num-col-elts* col) (1- (elt *num-col-elts* col) ))
  (setf (elt *num-dn-diag-elts* (+ limit (- row col)))
        (1- (elt *num-dn-diag-elts* (+ limit (- row col)))))
  (setf (elt *num-up-diag-elts* (+ row col))
        (1- (elt *num-up-diag-elts* (+ row col)))))



(defun num-preexisting-vios (row col limit)
  (+ (elt *num-col-elts* col)
     (elt *num-dn-diag-elts* (+ limit (- row col)))
     (elt *num-up-diag-elts* (+ row col))))


(defun get-a-random (x)
   (nth (random (length x)) x))


(defun print-info (var old-val new-val)
  "Prints solution status every iteration"
   (format t " ~A ~%"  *num-vars-violated*)
   (format t "~A ~A ~A ~A Vios: "
           *num-times* var old-val new-val))

(defun print-pic ()
  "Prints a NxN picture of the board"
  (loop with i = 0
        with size = (length *board*)
        while (< i size)
        do (loop with j = 0
                 while (< j size)
                 do (cond ((equal j (elt *board* i))
                           (format t "~A" "|*"))
                          (t (format t "~A" "| ")))
                 (incf j)
                 finally (format t "~A~%" "|"))
        (incf i))
  (terpri))
```

```
(defun make-n (n)
   (r-make-n n nil))


(defun r-make-n (n l)
   (cond ((equal n 0) (cons 0 l))
         (t (r-make-n (1- n) (cons n l)))))
```

References

[1] B. Abramson and M. Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *Journal of Parallel and Distributed Computing*, 61:649–662, 1989.

[2] H.M. Adorf and M.D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA, 1990.

[3] J. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–655, 1975.

[4] G. Brassard and P. Bratley. *Algorithmics - Theory and Practice.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] D. Brelaz. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[6] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[7] M.S. Fox, N. Sadeh, and C. Baykan. Constrained heuristic search. In *Proceedings IJCAI-89*, Detroit, MI, 1989.

[8] E.C. Freuder. Partial constraint satisfaction. In *Proceedings IJCAI-89*, Detroit, MI, 1989.

[9] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[10] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, volume 79, 1982.

[11] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation, Part II. *To appear in Journal of Operations Research*, 1990.

[12] D.S. Johnson, C.H. Papadimitrou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988.

[13] M.D. Johnston. Automated telescope scheduling. In *Proceedings of the Symposium on Coordination of Observational Projects.* Cambridge University Press, 1987.

[14] M.D. Johnston and H.M. Adorf. Learning in stochastic neural networks for constraint satisfaction problems. In *Proceedings of NASA Conference on Space Telerobotics*, Pasadena, CA, January 1989.

[15] L.V. Kale. An almost perfect heuristic for the n nonattacking queens problem. *Information Processing Letters*, 34:173=178, 1990.

[16] N. Keng and D.Y.Y. Yun. A planning/scheduling methodology for the constrained resource problem. In *Proceedings IJCAI-89*, Detroit, MI, 1989.

[17] C.A. Knoblock. Learning hierarchies of abstraction spaces. In *Proceedings of the Sixth International Conference on Machine Learning*, Ithica, N.Y., 1989.

[18] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:98–118, 1977.

[19] S. Minton. Empirical results concerning the utility of explanation-based learning. In *Proceedings AAAI-88*, Minneapolis, MN, 1988.

[20] Fox M.S. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann Publishers, Inc., 1987.

[21] N. Muscettola, S.F. Smith, G. Amiri, and D. Pathak. Generating space telescope observation schedules. Technical Report CMU-RI-TR-89-28, Carnegie Mellon University, Robotics Institute, 1989.

[22] R.G. Simmons. A theory of debugging plans and interpretations. In *Proceedings AAAI-88*, Minneapolis, MN, 1988.

[23] H.S. Stone and J.M. Stone. Efficient search techniques - an empirical study of the n-queens problem. *IBM Journal of Research and Development*, 31:464–474, 1987.

[24] G. J. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier, New York, 1975.

[25] J.S. Turner. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[26] M. Waldrop. Will the Hubble space telescope compute? *Science*, 243:1437–1439, 1989.