Backtracking without Trailing in CLP($\Re_{\mathrm{Lin}}$)

Pascal Van Hentenryck and Viswanath Ramachandran

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-51
November 1993

Backtracking without Trailing in $\mathtt{CLP}(\Re_{Lin})$[1]

Pascal Van Hentenryck and Viswanath Ramachandran
Brown University, Box 1910, Providence, RI 02912 (USA)
Email:{pvh,vr}@cs.brown.edu

Abstract

Constraint logic programming (CLP) is a generalization of logic programming where unification is replaced by constraint solving as the basic operation of the language. The combination of constraint solving and nondeterminism (approximated by backtracking) makes these languages appealing for a variety of combinatorial search problems. Existing CLP languages support backtracking by generalizing traditional Prolog implementations: modifications to the constraint system are trailed and restored on backtracking. Although simple and efficient, trailing may be very demanding in memory space, since the constraint system may potentially be saved at each choice point.

This paper proposes a fundamentally new implementation scheme for backtracking in CLP languages over linear (rational or real) arithmetic. The new scheme, called *semantic backtracking*, does not use trailing but rather exploits the semantics of the constraints to undo the effect of newly added constraints. Semantic backtracking reduces the space complexity by an order of magnitude compared to implementations based on trailing and makes space complexity essentially independent of the number of choice points. In addition, semantic backtracking introduces negligible space and time overhead on deterministic programs. The price for this improvement is an increase in backtracking time, although constraint-solving time may actually decrease. The scheme has been implemented as part of a complete CLP system $\mathtt{CLP}(\Re_{Lin})$ (about 20,000 lines of C) and compared analytically and experimentally with an optimized trailing implementation. Experimental results indicate that semantic backtracking produces significant reduction in memory space, while keeping the time overhead reasonably small.

1 Introduction

Constraint logic programming (CLP) [6] is a generalization of logic programming where unification is replaced by constraint solving as the basic operation of the language. The combination of constraint solving and nondeterminism (approximated by backtracking) makes these languages appealing for a variety of combinatorial search problems. Existing CLP languages support backtracking by generalizing traditional Prolog implementations: modifications to the constraint system are trailed (i.e. saved on a special stack which contains pairs $\langle address, value \rangle$) and restored on backtracking. In addition, techniques such as timestamps (e.g. [1]) are used to avoid trailing the same object several times in between two choice points. Although reasonably simple and efficient, trailing may be very demanding in memory space, since the constraint system can potentially be saved

[1]This research was partly supported by the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 and the National Science Foundation under grant numbers CCR-910832 and a National Young Investigator Award.

entirely in between each two choice points. As a consequence, it seems worthwhile investigating other backtracking schemes that would depart from the syntactic nature of trailing.

In this paper, we consider an instance of the CLP scheme over the constraint system $\Re_{Lin}$. $\Re_{Lin}$ consists of linear equations, inequalities, and disequations over real numbers[2] and is included in several languages, including CHIP [4, 12], CLP($\Re$) [8, 7, 10], and Prolog III [2]. Its implementation maintains the current set of constraints (which is known to be satisfiable) in a solved form and the fundamental operation of these languages is an incremental satisfiability test which checks if the conjunction of a new constraint c and the existing constraints is satisfiable. This satisfiability test in $\Re_{Lin}$ may fundamentally change the nature of the solved form, implying that trailing can be very expensive in space and that alternative schemes are important to investigate.

The main contribution of this paper is to present a new backtracking scheme for $\Re_{Lin}$. The new scheme, called *semantic backtracking* does not use trailing but rather exploits the semantics of the constraints to undo their effect on the solved form during backtracking. Semantic backtracking reduces the memory requirements of `CLP(`$\Re_{Lin}$`)` by an order of magnitude asymptotically, keeping the space and time overhead of nondeterminism to a minimum and ensuring that space complexity is independent from the number of choice points for numbers. In addition, semantic backtracking introduces negligible space and time overhead on deterministic programs. The price to pay for this functionality is an increase in backtracking time, since backtracking now needs to compute inverse linear transformations.

To demonstrate the practicability of the scheme, semantic backtracking has been implemented as part of a complete system `CLP(`$\Re_{Lin}$`)` (about 20,000 lines of C) and analysed both experimentally and theoretically. It has also been compared with a trailing implementation using exactly the same solver to quantify precisely the loss in efficiency. The experimental results indicate that semantic backtracking achieve a promising space/time tradeoff for $\Re_{Lin}$, since the time overhead of semantic backtracking is kept small while the space requirements are reduced significantly.

The rest of this paper is organized as follows. Section 2 illustrates `CLP(`$\Re_{Lin}$`)` on a simple example to illustrate some of the functionalities of the language and the implementation problems raised by the language. Section 3 provides some background about the constraint system while Section 4 presents the general architecture of the implementation. Sections 5, 6, and 7 discuss the implementation of the new scheme. Section 8 presents some experimental results, while Section 9 contains the conclusion of this research.

2 A Motivating Example

To illustrate the functionalities of `CLP(`$\Re_{Lin}$`)`, we use an example from [2]. Consider an infinite sequence of numbers $x_1, x_2, \ldots$ defined by

$$x_{i+2} \ = \mid x_{i+1} \mid - x_i \quad (i \geq 1)$$

where x_1 and x_2 are arbitrary real numbers. The problem consists in showing that the sequence is always periodic and has a period of 9 or, in other words, that the sequences $x_1, x_2, \ldots$ and $x_{10}, x_{11}, \ldots$ are always identical.

[2]All results described in this paper hold for real and rational numbers since the two structures are elementary equivalent [9]. Our implementation uses infinite-precision numbers to guarantee numerical stability. We also give experimental results on finite precision numbers.

```
invalidate :-
   [ X1, X2 ] ≠ [ X10, X11 ],
   sequence([X11,X10,X9,X8,X7,X6,X5,X4,X3,X2,X1]).

sequence([X2,X1]).
sequence([X11,X10,X9|Xs]) :- abs(X10,X10a), X11 = X10a - X9, sequence([X10,X9|Xs]).

abs(X,X) :- X ≥ 0.
abs(X,-X) :- X < 0.
```

Figure 1: The Periodic Sequence Program

The problem can be solved indirectly by using the following idea: since the two sequences are completely determined as soon as their first two elements are fixed, it is sufficient to show that any sequence $x_1, x_2, \ldots, x_{10}, x_{11}$ implies $x_1 = x_{10}$ and $x_2 = x_{11}$. The program then consists in searching for a solution such that $\langle x_1, x_2 \rangle \neq \langle x_{10}, x_{11} \rangle$. The absence of solutions proves the conjecture. The complete program is shown in Figure 1.

This example illustrates nicely the combination of constraint solving (including strict inequalities and disequations) and nondeterminism. The predicate `sequence` defines the sequence in reverse order. It is defined recursively and uses the predicate `abs` to compute the absolute value. `abs` is nondeterministic (since the values are not known originally) and first enforces the constraint that its argument is positive. On backtracking, `abs` enforces the constraint that its argument is strictly negative. Note that the program never gives any value to the variables, yet it fails in all cases thanks to the complete constraint solver. Any implementation of CLP($\Re_{Lin}$) needs to address two issues: incremental constraint solving and backtracking. Incremental constraint solving amounts to deciding dynamically the satisfiability of linear constraints. Backtracking amounts to undoing the effect of some constraint addition on the constraints accumulated so far. For instance, in our example, backtracking needs to undo the effect of constraints of the form $y \geq 0$ and $y < 0$ on previously encountered constraints. As mentioned already, traditional approaches simply undo the effect of new constraints by saving all modifications to the solved form in a purely syntactic manner. This paper proposes a new solution to implement backtracking.

3 The Constraint System

The constraint system of CLP($\Re_{Lin}$) deals with linear constraints.

Definition 1 Let t_1 and t_2 two linear expressions constructed with variables, rational numbers, and the operations $+, *, -$ and $/$. A constraint is a relation $t_1 \; \delta \; t_2$ with $\delta \in \{>, \geq, =, \neq, \leq, <\}$.

The constraint system is a generalization of linear programming [3] due to the presence of strict inequalities ($>$ and $<$) and disequations ($\neq$). Implementations of this constraint system use a set of rewriting rules (usually applied dynamically) to consider only equations and disequations. In particular, we have $t_1 \geq t_2$ is rewritten as $t_1 - s = t_2$, $t_1 > t_2$ as $t_1 - s = t_2$ & $s \neq 0$, and $t_1 \leq t_2$ as $t_1 + s = t_2$, $t_1 < t_2$ as $t_1 + s = t_2$ & $s \neq 0$, where s is a slack variable, i.e. a variable constrained to take only nonnegative values. In the following, arbitrary variables will be denoted by the letter x, slack variables by the letters s and a, and rational numbers by the letter b, all of them possibly subscripted.

4 The Architecture of the Implementation

The implementation of `CLP(`$\Re_{Lin}$`)` is organized around four modules, the engine, the interface, the G-system (Gauss system), and the S-system (Simplex system), with a left-to-right communication pattern. The engine is responsible for the control part of the system (e.g. clause and goal selection) while the G- and S-systems are responsible for constraint solving. The interface module converts the engine representation of constraints into their solver representation. The engine and the G- and S-systems have their own choice point stacks, since there is no reason to save information in the constraint system if no activity takes place. Design decisions similar in spirit were adopted in CLP($\Re$) [8]. The interface and the G- and S-systems can be seen as abstract data types providing seven operations: `SolveEqual, SolveGreatereq, SolveGreater, SolveNotEqual` and `Try, Retry, Trust`.

The first four operations are used to insert a new constraint and returns a Boolean value. The main differences between these functions in the different modules is the constraints they manipulate. The interface receives constraints in the engine representation and transmits them to the G-system in the solver representation. The G-system receives constraints containing only arbitrary variables, while the S-system receives constraints containing only slack variables. The second set of operations corresponds to the traditional or-level instructions of WAM-based implementations of Prolog [13]. `Try` signals the creation of a choice point, `Retry` signals the selection of the next (but not the last) alternative in the choice point, while `Trust` signals the selection of the last alternative in the choice point. Backtracking in the solver is initiated by the instructions `Retry` and `Trust`.

In the G- and S-systems, inequalities are transformed immediately into equations and disequations (by adding slack variables) when they are not trivially satisfiable or unsatisfiable. Hence, in the rest of the presentation, we only consider equations and disequations.

The rest of the implementation will be described as follows. Section 5 discusses the G-system while Section 6 discusses the S-system. To ease the presentation, we assume in Section 5 that the G-system contains only arbitrary variables (i.e. in other words, we ignore inequalities). Section 7 shows how to integrate inequalities in a simple way. Both Sections 5 and 6 are organized in five sections: the solved form, constraint solving, backtracking with trailing, semantic backtracking, and analysis. In the following, we use `Trail` to refer to the `CLP(`$\Re_{Lin}$`)` implementation using trailing and `Semantic` to refer to the implementation using semantic backtracking.

5 The Gauss System

In this section, we consider constraint solving and backtracking for systems of equations and disequations over arbitrary variables. Constraint solving and semantic backtracking are relatively simple in the G-system but they introduce the main ideas used in the more complex S-system.

5.1 The Solved Form

Informally, the solved form in the G-system consists in isolating one variable per equation and making sure that the disequations do not reduce to $0 \neq 0$.

Definition 2 An equation is in solved form iff it is of the form $x_i = b_0 + b_1x_1 + \ldots + b_{i-1}x_{i-1} + b_{i+1}x_{i+1} + \ldots + b_nx_n$. x_i is called a *basic* variable.

Definition 3 A system of equations $\mathcal{E}$ is in solved form iff (1) each equation of $\mathcal{E}$ is in solved form; (2) all basic variables are distinct; (3) the right-hand side of the equations do not contain any basic variable.

Definition 4 A system of equations and disequations $\mathcal{E} \cup \mathcal{D}$ is in solved form iff $\mathcal{E}$ is in solved form and each disequation in $\mathcal{D}$ is of the form $0 \neq b_0 + b_1x_1 + \ldots + b_nx_n$, does not contain any basic variable, and has at least one non-zero coefficient b_i $(0 \leq i \leq n)$.

5.2 Constraint Solving

Constraint solving in the G-system is based on Gaussian elimination and backpropagation.

Insertion of a Disequation Adding a disequation requires to dereference (i.e. to remove the basic variables of) the constraint first. If the dereferenced constraint is of the form $0 \neq b$, it is trivially satisfiable or unsatisfiable. Otherwise, it contains at least one variable and must be added to the disequations.

Insertion of an Equation Adding a new equation e to a system $\mathcal{E} \cup \mathcal{D}$ consists of three steps: (1) dereferencing e; (2) choosing a basic variable for e; and (3) backpropagation. Dereferencing amounts to replacing each basic variable of $\mathcal{E}$ in e by its right-hand side producing a new constraint e' of the form $0 = b_0 + b_1x_1 + \ldots + b_nx_n$. If the b_i $(1 \leq b_i \leq n)$ are all zeros, then the constraint is either trivially satisfiable or unsatisfiable. Otherwise, a variable with a non-zero coefficient is selected to become the new basic variable and the constraint is rewritten accordingly. Backpropagation then replaces each occurrence of this new basic variable by its right-hand side in the old system $\mathcal{E} \cup \mathcal{D}$ and makes sure that no disequation reduces to $0 \neq 0$, in which case the system would be unsatisfiable. The new system is then obtained by adding the new constraint.

5.3 Backtracking with Trailing

In this section, we give an overview of the trail implementation of `CLP(`$\Re_{Lin}$`)` which will be used in the comparison with semantic backtracking. Although it uses some ideas well-known in the folklore of CLP, algorithm `Trail` improves on many existing implementations in its memory usage and will be used in the comparison with semantic backtracking.

Algorithm `Trail` follows the traditional approach of CLP languages which is an extension of the Prolog backtracking strategy. The basic idea is to store on a special stack, called the trail, the address and current value of each object about to be modified. The trail thus contains pairs $< address, value >$ and, in `CLP`($\Re_{Lin}$), the value is often a pointer to an infinite precision number. Backtracking simply unwinds the trail and restores the values as the contents of the addresses.

To avoid trailing several times the same object, algorithm `Trail` uses time stamps to determine if the object has been modified since the last choice point, in which case it does not need to be trailed again. Time stamps have been used in several implementations of CLP languages, including `CHIP` and CLP($\Re$), and lead to the following pattern to update an object

```
IF "it is necessary to trail" THEN BEGIN
      trail; update the time stamp;
END
update the object;
```

Since `CLP`($\Re_{Lin}$) deals with infinite precision numbers, algorithm `Trail` devotes extra care (which is often omitted in existing implementations which are based on a stack architecture) to release the numbers which are no longer necessary. This arises in two situations: (1) when a coefficient is updated several times in between two choice points and (2) on backtracking. In between two choice points, it is only important to have access to two numbers for each coefficient: its first and its last values. All intermediary numbers can be freed since they are not needed for backtracking. This leads to the following pattern for updating coefficients

```
IF "it is necessary to trail" THEN BEGIN
      trail; update the time stamp;
END ELSE
      release the old number;
store the new number;
```

On backtracking, all numbers allocated since the last choice point can be released. Once again, using time stamps enables us to perform this efficiently. Note that the first situation defined above precludes the use of a stack to allocate the numbers which would allow the second step to be performed in constant time. Such a stack organization would not enable the reuse of intermediary numbers. Algorithm `Trail` uses the number of active choice points to update the time stamps. This makes sure to recover all unnecessary numbers during the computation at the cost of having to restore the time stamps. A simple scheme using a counter incremented by the `Try, Retry` and `Trust` operations would be suboptimal.

Finally, since the solved form is generally organized using linked-list data structures, algorithm `Trail` devotes extra care to recover cells which are no longer needed. This happens when a new coefficient is created (i.e. it becomes different from zero) and subsequently deleted (i.e. it becomes zero) in between two choice points. Algorithm `Trail` detects this situation using time stamps once again and is able to reuse the space. Note that once again this precludes stack allocation for cells.

Our trailing implementation preserves the minimal amount of numbers and cells (contrary to most implementations we are aware of). Note that trailing can be further reduced by several techniques such as the reconstruction of some of the data structures at backtracking time (e.g. cross-reference tables in the `CLP`($\Re$) system [8]). These techniques may increase the backtracking

```
p(x1,x2,x3) :-
     x1 + 2 x2 + 3 x3 ≠ 1, x1 + x2 + x3 = 4, q(x1,x2,x3), r(x1,x2,x3), ...

q(x1,x2,x3) :- x1 - 2 x2 + 4 x3 = 7 .
q(x1,x2,x3) :- ....

r(x1,x2,x3) :- x1 + 3 x2 - 3 x3 = 4.
r(x1,x2,x3) :- ....
```

Figure 2: A simple CLP program for the G-system

time. For space reasons, we do not discuss these techniques in this abstract. We are now in position to summarize the various actions performed by the operations.

Constraint Solving Constraint solving trails any modification to the solved form.

Operation `Try` This operation saves the dimensions of the solved form and the top of the trail.

Backtracking Backtracking unwinds the trail to restore the old values at the appropriate addresses. It also releases the numbers allocated since the last choice point and resets the top of the trail.

Operations `Retry` and `Trust` Operation `Retry` simply performs backtracking while operation `Trust` performs backtracking and pops the choice point.

5.4 Semantic Backtracking

This section considers the implementation of semantic backtracking for the G-system. Its main motivation is to avoid the drawback of algorithm `Trail` which may potentially save the whole solved form in between two choice points. It starts with a motivating example and then describes the operations precisely.

An Example Consider the (partial) CLP program depicted in Figure 2 and using the same convention as for the constraint system for simplicity. Assuming a goal `p`(x_1, x_2, x_3), the solved form before calling `q/3` is $\{0 \neq 3 + x_2 + 2x_3 \ , \ x_1 = 4 - x_2 - x_3\}$. Before calling `r/3`, the solved form becomes $\{0 \neq 2 + 3x_3 \ , \ x_1 = 5 - 2x_3 \ , \ x_2 = -1 + x_3\}$. The solved form after `r/3` is simply $\{0 \neq -1 \ , \ x_1 = 7 \ , \ x_2 = -2 \ , \ x_3 = -1\}$.

Observe now that backtracking to select the second clause of `r/3` consists in removing the effect of last constraint which can be achieved in a simple way by adding to the first three constraints the last constraint after multiplying it by -3, 2, and -1 respectively. Note also that 3, -2, and 1 are in fact the coefficients of x_3 in the previous solved form. Similarly, backtracking over the third constraint to select the second clause of `q/3` amounts to adding the third constraint to the first two after multiplying it by -1 and 1. Now observe the coefficients of x_2 and x_3 in the first two solved form. It becomes clear that, to backtrack in the G-system, it is sufficient to remember the coefficients of the new basic variable in the old solved form. These coefficients can be used at backtracking time to restore the previous states by adding multiples of the last constraint. We now consider how to implement this idea.

Constraint Solving Equation solving needs to be modified slightly. The modification occurs when the new basic variable is present in the old solved form. In this case, backpropagation should save the coefficients of the new variable for subsequent backtracking, when in a nondeterministic state. Note that no saving is required when the new basic variable is a new variable, since it does not appear in the old system. The handling of disequations is left unchanged.

Instruction `Try` Instruction `Try` is postponed until some activity occurs in the solver. At this point, it simply saves the dimension of the constraint system.

Backtracking Backtracking undoes the effect of all constraints inserted since the last choice point one at a time. No work is necessary when the constraint is a disequation or an equation which contained a new variable at insertion time. Otherwise, multiples of the removed constraints are added to the solved form, using the coefficients saved during equation solving. Backtracking also restores the dimensions of the solved form.

Instruction `Retry` and `Trust` Instruction `Retry` simply performs backtracking. Instruction `Trust` performs backtracking and pops the choice point.

5.5 Analysis

In this section, we compare the space and time complexity of the two algorithms. In the complexity analysis, we distinguish between numbers and words for space complexity and arithmetic and word operations for time complexity. This distinction is useful, since several implementations of the constraint system use infinite precision numbers. The complexity results are expressed in terms of the computation state of the system which can be described for our purposes by three numbers, c, v and b, representing the number of constraints c and variables v of the constraint system in the computation state and the number of choice points which are active in the state. Note also that c is necessarily smaller or equal to v in the analysis, as the system would be overconstrained otherwise.

We first consider space complexity. Algorithm `Trail` may need to trail almost the whole constraint system in between two choice points, since backpropagation may affect the coefficients of all non-basic variables. Algorithm `Semantic` requires saving the coefficients of the new basic variable in the old solved form when a new equation is added to the solved form. Hence, Algorithm `Semantic` reduces the space overhead and complexity by an order of magnitude for numbers and words.

Proposition 5 [Space Complexity of `Trail`] In the worst case, algorithm `Trail` requires $O(cvb)$ numbers and $O(cvb)$ words for some computation state to support nondeterminism. The space complexity of algorithm `Trail` is thus $O(cvb)$ numbers + $O(cvb)$ words in the worst case.[3]

Proposition 6 [Space Complexity of `Semantic`] In the worst case, algorithm `Semantic` requires $O(c^2)$ numbers and $O(b)$ words for some computation state to support nondeterminism. The space complexity of algorithm `Semantic` is thus $O(cv)$ numbers + $O(max(cv, b))$ words in the worst case.

We now consider time complexity. The time overhead of `Trail` consists in trailing the (addresses of) modified coefficients. The time overhead of `Semantic` comes from the saving of the pointers to the modified coefficients. Interestingly, the time overhead of `Semantic` is an order of magnitude lower than `Trail`.

Proposition 7 [Time Complexity of Constraint Solving in `Trail`] In the worst case, the overhead of algorithm `Trail` is $O(cv)$ word operations per constraint insertion. Its time complexity for constraint insertion is thus $O(cv)$ word and arithmetic operations in the worst case.

Proposition 8 [Time Complexity of Constraint Solving in `Semantic`] In the worst case, the overhead of algorithm `Semantic` is $O(c)$ word operations per constraint insertion. Its time complexity for constraint insertion is thus $O(cv)$ word and arithmetic operations in the worst case.

The price to pay for the space improvement of `Semantic` is in fact in the backtracking time. Algorithm `Trail` only needs to restore the previous coefficients which can be done by manipulating addresses. Algorithm `Semantic` needs to perform arithmetic operations and cannot amortize the cost over several constraint deletions.

Proposition 9 [Time Complexity of Backtracking in `Trail`] In the worst case, algorithm `Trail` performs $O(cv)$ word operations to backtrack over any number of constraints.

Proposition 10 [Time Complexity of Backtracking in `Semantic`] In the worst case, algorithm `Semantic` performs $O(cv)$ arithmetic operations to backtrack over a constraint. It performs $O(ncv)$ arithmetic operations to backtrack over n constraints.

6 The Simplex System

We now consider consider equations and disequations over slack variables. Constraint solving and backtracking are substantially more complex this case.

6.1 The Solved Form

The only difference between the solved forms of the G-system and S-system is a lexicographic requirement imposed on the solved form of the equations. The lexicographic requirement, proposed in [12], assumes a total ordering on the variables (e.g. $s_1 < s_2 < \ldots$) and generalizes the positivity requirement of the simplex algorithm. The positivity requirement makes sure that the equations are satisfiable, while the lexicographic requirement guarantees that all fixed variables (i.e. variables constrained to take a single value) are made explicit. The guarantee that all fixed variables are

[3] Theoretically, it is possible to reduce the space complexity to take only O(max(cv,b)) words. However, this would entail a time penalty for constraint solving which would need to store coefficients on the trail instead of pointers.

made explicit makes it easy to check the disequations, since it is sufficient to check if a disequation does not reduce to $0 \neq 0$. See [12] for complete account on the solved form. Semantic backtracking can also be applied to systems not based on the lexicographic solved form such as `CLP(`$\Re$`)` and `Prolog III`.

Definition 11 An equation over slack variables is in solved form iff it is of the form

$$s_i = b_0 + b_1 s_1 + \ldots + b_{i-1} s_{i-1} + b_{i+1} s_{i+1} + \ldots + b_n s_n$$

and either the first non-zero coefficient in the sequence $\langle b_0, b_1, \ldots, b_{i-1}, -1\rangle$ is positive or all b_i (i.e. $b_0, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n$) are zeros.

The definitions for systems of equations and disequations are the same as for the G-system.

6.2 Constraint Solving

Insertion of a Disequation Disequations are handled in the same way as in the G-system. The disequation is dereferenced, tested for satisfiability, and inserted in the solved form if necessary.

Insertion of an Equation The addition of a new equation is based on the simplex algorithm. The main idea is to dereference the constraint and to make a case analysis considering at least three cases:

1. *New Variable Case:* the constraint contains a new variable that can be put in basis while satisfying the lexicographic requirement;
2. *Fixed Variables Case:* the constraint is of the form $0 = b_0 + b_1 s_1 + \ldots + b_n s_n \quad (b_i \geq 0$ for $0 \leq i \leq n)$;
3. *General Case:* all other cases.

The case *new variable* is simple, since a new solved form is obtained easily and the disequations are clearly satisfiable. The *general* case is handled through the simplex algorithm. The key idea is to introduce an artificial variable a to obtain a new constraint $a = b_0 + b_1 s_1 + \ldots + b_n s_n$. The simplex algorithm is then used to minimize the value of a in an extended system where the above constraint has been added with a in basis. If the minimization returns a strictly positive value, the system of equations is inconsistent. Otherwise, the system is consistent and the artificial variable can be removed (perhaps after some preliminary work if the artificial variable is still in basis). The final step consists in dereferencing all disequations to make sure that they do not reduce to $0 \neq 0$. Recall as well that the simplex algorithm performs a sequencing of operations, each of them consisting of three steps: (1) choosing a variable to enter the basis; (2) choosing a variable to leave the basis; and (3) pivoting the system to introduce the entering variable in basis in place of the leaving variable. Pivoting is the main operation and consists in adding multiples of the constraint c whose basic variable is the leaving variable to all other constraints to make sure than the entering variable occurs only in c. The constraint c itself is also divided by the coefficient of the entering variable to produce the new solved form. Pivoting preserves the solved form provided that a lexicographic pivoting rule is used [12].

The case *fixed variable* is more subtle but does not require the full simplex algorithm. If b_0 is greater than zero, the constraint is unsatisfiable. Otherwise, all slack variables with a non-zero coefficient must be equal to zero and this fact needs to be propagated in the solved form. This can be achieved by simply removing these variables from the solved form. However, the resulting system may not be in solved form anymore due to the lexicographic requirement. Two main cases must be distinguished when this happens for a constraint:

1. the constraint is of the form $0 = a_1x_1 + \ldots + a_nx_n$ with $a_i < 0$ for all i;
2. the constraint is of the form $0 = a_1x_1 + \ldots + a_nx_n$ with some $a_i > 0$ for $2 \leq i \leq n$.

In the first case, the algorithm has discovered new fixed variables and they can be handled as the initial fixed variables. In the second case, pivoting takes place to introduce into the basis a variable with a positive coefficient. Note that this last step may entail that some other constraints need to be reconsidered but the algorithm terminates due to the lexicographic ordering. The final step consists of course in checking the disequations.

We now consider backtracking in the S-system and focus on semantic backtracking since the trailing implementation uses the same techniques as in the G-system.

6.3 Semantic Backtracking

Backtracking in the S-system is substantially more involved than in the G-system. The simplicity of backtracking in the G-system comes from two facts: on the one hand, older constraints are never added or subtracted from new constraints; on the other hand, new constraints only extend the basis. None of these properties hold in the S-system. However, constraint solving in the S-system still performs linear transformations and hence, with some care, it should be possible to invert them. The rest of this section starts with an informal overview of the approach (including examples) and then specify more precisely the various instructions.

Informal Presentation Algorithm `Semantic` performs backtracking in the S-system in two main steps: (1) undoing the effect of the new constraints, i.e. the constraints introduced since the last choice point; (2) restoring the previous solved form, i.e. the solved form in use when the last choice point was created.

The first step is similar in spirit to backtracking in the G-system but is more subtle for reasons that will become clear shortly. The second step is necessary because the first step may not return to a solved form or may return to a different solved form. The only guarantee provided by step 1 is that it returns to a system which is equivalent to the previous solved form. We now consider each step in detail, focusing mainly on the general case for simplicity.

For the first step, observe that, after the minimization procedure, the coefficients of the artificial variable contains the information necessary to undo the effect of the constraint. Hence, it seems natural to save these coefficients to enable backtracking subsequently. Unfortunately, this simple solution is not correct because of subsequent pivotings which may affect the relationships between the constraints. Consider the following incomplete CLP program (the s_i are thus assumed to be slack variables):

`p(`s_1`,`s_2`,`s_3`,`s_5`,`s_6`,`s_7`) :- `s_1` + 2 `s_2` + `s_3` - `s_5` = 8, q(`s_1`,`s_2`,`s_3`,`s_5`,`s_6`,`s_7`), ...`

```
q(s1,s2,s3,s5,s6,s7) :- 3 s1 + 3 s2 + s3 - s6 = 15, s2 - s7 = 3.
q(s1,s2,s3,s5,s6,s7) :- ...
```

The first solved form before the call to `q/6` is $\{s_3 = 8 - s_1 - 2s_2 - s_5\}$. Insertion of the first constraint in `q/6` leads to the general case producing the system

$$\{s_3 = \tfrac{9}{2} - \tfrac{3}{2}s_2 + \tfrac{3}{2}s_5 - \tfrac{1}{6}s_6 + \tfrac{1}{2}a_0 \ , \ s_1 = \tfrac{7}{2} - \tfrac{1}{2}s_2 - \tfrac{1}{2}s_5 + \tfrac{1}{2}s_6 - \tfrac{1}{2}a_0\}$$

and its associated solved form $\{s_3 = \frac{9}{2} - \frac{3}{2}s_2 + \frac{3}{2}s_5 - \frac{1}{6}s_6 \ , \ s_1 = \frac{7}{2} - \frac{1}{2}s_2 - \frac{1}{2}s_5 + \frac{1}{2}s_6\}$ after removal of the artificial variable. Note also that the coefficients of the artificial variable. Insertion of the second constraint of `q/6` induces a change of basis and leads to the system

$$\{s_2 = 3 + s_7 - a_1 \ , \ s_1 = 2 - \tfrac{1}{3}s_3 + \tfrac{1}{3}s_6 - s_7 + a_1 \ , \ s_5 = 0 + \tfrac{2}{3}s_3 + \tfrac{1}{3}s_6 + s_7 - a_1\}$$

and to the solved form $\{s_2 = 3 + s_7 \ , \ s_1 = 2 - \frac{1}{3}s_3 + \frac{1}{3}s_6 - s_7 \ , \ s_5 = 0 + \frac{2}{3}s_3 + \frac{1}{3}s_6 + s_7\}$. Now assume that we need to backtrack to the second clause of `q/6`. Using the coefficients of a_1, backtracking produces the system $\{s_2 = 3 - \frac{2}{3}s_3 + s_5 - \frac{1}{3}s_6 \ , \ s_1 = 2 + \frac{1}{3}s_3 - s_5 + \frac{2}{3}s_6\}$. Using the coefficients of a_0, backtracking would produce the incorrect system $\{s_1 + s_2 = 5 - \frac{1}{3}s_3 + \frac{1}{3}s_6\}$. The problem comes from the fact that the first constraint has been divided by 2 during pivoting and then added to the other constraints, but this information has not been propagated on the coefficients.

The solution adopted in algorithm `Semantic` is to preserve the artificial variable until the next (solver) choice point and to perform traditional pivoting operations on artificial variables as well. In other words, algorithm `Semantic` works with an extended system which, in addition to the solved form, contains the artificial variables introduced since the last choice point. The solved form can be obtained easily by removing the artificial variables. All the operations (e.g. dereferencing, selection of the entering variables) use the solved form except pivoting which also updates the coefficients of the artificial variables. It is sound to save the coefficients of the artificial variables at the next choice point, since backtracking returns to the same solved form and hence the validity of our coefficients is guaranteed. Consider our example again. The final system, keeping the artificial variables a_0 and a_1 for backtracking purposes, becomes

$$\{s_2 = 3 + s_7 - a_1 \ , \ s_1 = 2 - \tfrac{1}{3}s_3 + \tfrac{1}{3}s_6 - s_7 - \tfrac{1}{3}a_0 + a_1 \ , \ s_5 = 0 + \tfrac{2}{3}s_3 + \tfrac{1}{3}s_6 + s_7 - \tfrac{1}{3}a_0 - a_1\}.$$

The solved form can of course be obtained by removing the artificial variables. Backtracking over the last constraint leads to the system

$$\{s_2 = 3 - \tfrac{2}{3}s_3 + s_5 - \tfrac{1}{3}s_6 + \tfrac{1}{3}a_0 \ , \ s_1 = 2 + \tfrac{1}{3}s_3 - s_5 + \tfrac{2}{3}s_6 - \tfrac{2}{3}a_0\}.$$

Now backtracking over the second constraint leads to the system $\{2s_2 + s_1 = 8 - s_3 + s_5\}$ correctly undoing the effect of the new constraints.

The second step, *restoring the solved form*, is necessary[4] since the first step may not return to a solved form or may return to a solved form which is different from the initial solved form, invalidating the coefficients of the artificial variables saved in the choice point. For instance, the first step for our example may lead to a system $\{2s_2 = 8 - s_1 - s_3 + s_5\}$ which is not in solved form. It is of course easy to restore the solved form in this example by putting s_3 in basis instead of s_2 but, in general, the system obtained by the first step may be far from the old solved form. Algorithm `Semantic` overcomes this problem in two steps: (1) at each choice point, it saves the

[4] It is possible to imagine an implementation returning to a different solved form. See Section 6.4.

variable in basis for each constraint; (2) during backtracking, it restores the old basis by solving the system of equations in terms of these variables. We illustrate this on an example. Assume that the first step returns the system

$$\{s_5 = \frac{1}{2} - \frac{1}{2}s_1 - \frac{3}{2}s_4 - \frac{1}{2}s_6 \ , \ s_7 = \frac{5}{2} + \frac{1}{2}s_1 - s_2 - \frac{7}{2}s_4 - \frac{1}{2}s_6 \ , \ s_8 = \frac{3}{2} + \frac{5}{2}s_1 - s_3 + 1\frac{1}{2}s_4 + \frac{3}{2}s_6\}$$

and assume that the basic variables saved in the choice point are s_1, s_2, s_3. The initial system is simply obtained by dividing the first constraint by the coefficient of s_1 and removing s_1 from the other constraints to obtain

$$\{2s_5 = 1 - 1s_1 - 3s_4 - 1s_6 \ , \ s_7 = 3 - s_2 - 5s_4 - s_5 - s_6 \ , \ s_8 = 4 - s_3 - 2s_4 - 5s_5 - s_6\}$$

The old solved form is then obtained by putting s_1, s_2, s_3 in basis to produce

$$\{s_1 = 1 - 3s_4 - 2s_5 - 1s_6 \ , \ s_2 = 3 - 5s_4 - s_5 - s_6 - s_7 \ , \ s_3 = 4 - 2s_4 - 5s_5 - s_6 - s_8\}$$

In summary, algorithm `Semantic` stores two kinds of information in between two choice points to perform backtracking: (1) for each equation, it saves a *witness* variable which is essentially the first basic variable of the equation in between the choice points. In particular, it is the basic variable for the constraints existing before the choice point, the artificial variable for the general case of insertion, and the new variable for the *new variable* case; (2) tt also saves the coefficients of the artificial variables when a new choice point is created.

Constraint Solving The disequation solving algorithm is left unchanged. Equation solving is modified differently depending on the various cases. The *new basic variable* case requires only to remember the new basic variable as the witness variable. The *general* case is modified to remember the artificial variable as the witness variable and to avoid removing the artificial variables. Pivoting operations apply to the artificial variables as well but all other operations do not involve them. The *fixed variables* case is more subtle since removing the fixed variables from the solved form would prevent us from undoing correctly later on. The key idea is to consider each fixed variable s at a time and introduce a constraint $s = a$ where a is a new artificial variable which will be the witness variable for the constraint. This constraint is then used to eliminate s from the solved form. The rest of the algorithm is left unchanged except that new fixed variables are handled as just described.

Operation `Try` Recall that this operation is postponed until some activity takes place. Operation `Try` consists of three steps: (1) the saving of the coefficients of the artificial variables of the current system; (2) the saving of the witness variable for each equation of the solved form; (3) the saving of the dimensions of the constraint system.

Backtracking Backtracking in the S-system is performed in four steps: (1) undoing the effect of the new constraints; (2) restoring the old dimensions; (3) restoring the old solved form; (4) dereferencing the disequations. The first step considers each equation at a time and undoes its effect. Undoing a single equation e consists in using the witness variable s to eliminate all occurrences of s in the rest of the constraint system, the only occurrence of s being now in e. This can be done easily by using linear transformations on the constraint system. The third step uses the witness variables of the old equations to restore the old solved form. Assume that $s_1, \ldots, s_n$ are the witness variables. The key idea is to apply linear transformations so that the column of the constraint system associated with $s_1, \ldots, s_n$ make up a unit matrix. This amounts to solving the system of equations for variables $s_1, \ldots, s_n$. Finally, the last step dereferences the disequations to remove any occurrence of a basic variable.

Operation `Retry` and `Trust` Operation `Retry` simply performs backtracking. Operation `Trust` performs backtracking, pops the last choice point, and restores the coefficients of the artificial variables. This last step is necessary, since, on the one hand, future backtracking needs to use these coefficients to undo the effect of the constraints inserted in between the last two choice points and, on the other hand, subsequent pivotings must update these coefficients.

6.4 Analysis

We first consider space complexity. Algorithm `Trail` may trail the whole constraint system in between two choice points, since pivoting may affect the coefficients of all variables. The space complexity of `Semantic` for the S-system is similar to the complexity of `Semantic` for the G-system, except that we need to account for the witness variables.

Proposition 12 [Space Complexity of `Trail`] In the worst case, algorithm `Trail` requires $O(cvb)$ numbers and $O(cvb)$ words for some computation state to support nondeterminism. Its space complexity of algorithm `Trail` is thus $O(cvb)$ numbers + $O(cvb)$ words in the worst case.[5]

Proposition 13 [Space Complexity of `Semantic`] In the worst case, algorithm `Semantic` requires $O(c^2)$ numbers and $O(cb)$ words for some computation state to support nondeterminism. Its space complexity of algorithm `Semantic` is thus $O(cv)$ numbers + $O(max(cv, cb))$ words in the worst case.

Once again, algorithm `Semantic` reduces the space overhead and complexity by an order of magnitude for numbers and words. We now consider the time overhead for constraint solving and restrict attention to the overhead of pivoting since this is the kernel operation of the solver. In each pivoting, algorithm `Trail` checks if the modified coefficients need to be trailed and pushes their addresses on the trail. The time overhead of `Semantic` for pivoting comes from the use of additional variables (i.e. the artificial variables). In the worst case, there are as many additional variables as new constraints.

Proposition 14 [Time Overhead for Pivoting in `Trail`] In the worst case, algorithm `Trail` performs an additional $O(cv)$ word operations per constraint insertion compared to a deterministic algorithm.

Proposition 15 [Time Overhead for Pivoting in `Semantic`] In the worst case, algorithm `Semantic` performs an additional $O(nc)$ operations per constraint insertion compared to a deterministic algorithm, where n is the number of constraints inserted since the last choice point.

It is easy to see that algorithms `Trail` and `Semantic` do not increase the overall time complexity asymptotically. We now consider backtracking time. Backtracking in algorithm `Trail` is simple and consists in freeing some numbers and restoring some coefficients. Backtracing in algorithm `Semantic` requires solving a system of equations in terms of the witness variables.

[5] Once again, in theory, it is possible to reduce the space complexity to take only O(max(cv,cb)) words.

Proposition 16 [Time Complexity of Backtracking in `Trail`] In the worst case, algorithm `Trail` performs $O(cv)$ word operations for backtracking.

Proposition 17 [Time Complexity of Backtracking in `Semantic`] In the worst case, algorithm `Trail` performs $O(c^2v)$ arithmetic operations for backtracking.

The complexity of backtracking increases not only because arithmetic operations are considered but also because the number of operations increases by a factor c. It is interesting to note that there exists a backtracking scheme which is asymptotically better in memory space than `Semantic`. The key idea is to avoid storing the witness variables for the old constraints, reducing to the space complexity to $O(cv)$ numbers + $O(max(cv, b))$ words and the overhead of $O(b)$ words. To achieve this space complexity, the coefficients of the artificial variables need to be preserved for the whole computation to allow the system to return to any solved form. A price to pay is a further increase in backtracking time, since backtracking has no information to come back to a solved form.

7 Integration of the G-system and the S-system

The G-system and the S-system can be integrated by allowing slack variables to appear in G-system as well but never as basic variables. The solved form then contains both arbitrary and slack variables on the right-hand side. All algorithms remain the same except that (1) if dereferencing produces a constraint containing only slack variables, this constraint is transferred from the G-system to the S-system; and (2) if backpropagation leads to a disequation containing only slack variables, it is sent to the S-system. See [5] for the details.

8 Experimental Results

The Benchmarks Our benchmarks are mainly taken from [2, 8, 11] with a couple of additional programs. The first four programs are cryptarithmetic puzzles with two versions for each problem. The two versions correspond to different representations of the constraint (a natural and a carry representation) which lead to very different search spaces. `SendMory1` is from [8] while `Donald` is from [2]. The programs have the property that backtracking undoes only constraints of the form $s = b$, where s is a slack variable and b is a value. The next two programs, `Investment` and `Laplace`, are deterministic programs. The first program is taken from [2] while the second program is from [8]. `Investment` is a program to compute installments with strong relationships between the same values. `Laplace` computes an approximation method for Laplace's equation. Both programs manipulate large coefficients and produce overflows with finite precision numbers. Note also that, although the programs are deterministic, they produce some choice points (which would lead to failure if tried). The last programs illustrate backtracking over arbitrary constraints. `Magic` and `Permutation` solves two problems from combinatory theory [11]. `Periodic` is the problem described in Section 2. `Square` [2] is a very combinatorial problem which amounts to placing squares, all of different sizes, in a rectangle such that no two squares overlap and no empty space is left. The only input to the program is the number of squares, i.e. the sizes of the squares and of the rectangle are left unspecified. We use various instances of this problem.

The Systems Algorithms `Semantic` and `Trail` are both WAM-based compilers using infinite precision integers to guarantee numerical stability. The solved form in the algorithms uses a linked list representation and the two systems (resp. 20,000 and 19,000 lines of C) only differ in the way backtracking is handled.

Memory Space For each system, we measure the space requirement at the end of the computation and at each backtracking. In the table, we report the space at the end of the computation and the maximum over all backtracking points. The space itself is divided into various categories, including the integers, the rationals, the cells in the linked lists, and the data structures needed to support nondeterminism. Tables 1 and 2 report the results for algorithms `Semantic` and `Trail`; Table 3 depicts the ratios between the algorithms. The experimental results indicate that `Semantic` uses in the average a third of the space used by `Trail` both at the end of the computation and over the backtracking points. The space requirement for `Square 14` is divided by 5 and substantial improvements also occur for deterministic programs, since `Semantic` requires half the space of `Trail` in `Laplace`, which creates choice points not leading to any solution. In general, the space improvement is higher for programs which add general constraints nondeterministically (about a quarter of the space). It is also interesting to identify where the space goes. In the average, `Semantic` produces about a 55% reduction on the integers, a 25% reduction on the cells, and a 90% reduction on the additional data structures, indicating that the trail and the integers that need to be preserved are mainly responsible for the space difference. Table 4 presents the theoretical minimum space for each benchmarks, i.e. the space needed by the solved form, and compares the space consumption of `Semantic` and `Trail` with the theoretical minimum at backtracking points. It appears that, in the average, `Semantic` requires about twice the theoretical minimum, while `Trail` requires about 6.5 times the minimum space. The additional space is taken mainly by additional data structures in both cases. Note also the excellent behaviour of `Semantic` on deterministic programs for which it requires negligible overhead, contrary to `Trail`. Finally, Table 5 depicts the results where the rationals are implemented using finite-precision numbers. Not all programs can be run with this restriction. The overall average is even more impressive than for finite precision numbers, since `Semantic` produces reductions of about 70%.

Computation Time Algorithm `Semantic` trades memory space for computation time and we now quantify experimentally the loss in computation time of `Semantic` compared to `Trail`. The results are reported in Table 6. They indicate that `Trail` is 20% faster than `Semantic` in the average. In the worst case (i.e. `Square 17`), `Trail` is 37% faster while, in the best case (i.e. `Magic`), `Semantic` is 1.20 times faster than `Trail`. This last case is mainly due to the fact that little undoing is performed during backtracking and that the system is extremely sparse. Hence, the time overhead of `Trail` becomes significant. For finite precision, the difference between both algorithms is even lower with an average reduction of 15% for `Trail`.

9 Conclusion

In traditional CLP implementations, backtracking is performed by trailing all modifications to the solved form and restoring them at backtracking time. Although simple and efficient, this scheme is potentially expensive in memory, since the entire solved form may need to be trailed between two choice points. In this paper, we proposed a fundamentally different backtracking scheme for `CLP(`$\Re_{Lin}$`)`. The new scheme, called *semantic backtracking*, avoids trailing and exploits the semantics of the constraints to come back to a previous state. The scheme has been analysed both theoretically and experimentally. On the theoretical side, semantic backtracking produces an order of magnitude reduction in memory space, making the space requirement for numbers independent from the number of choice points. The cost for this improvement is a higher backtracking time, since backtracking now requires solving a set of linear equations. The experimental results show that semantic backtracking requires, in the average, about 1/3 of the space of a traditional trailing implementation, and 1/5 on some benchmarks. The overhead in time is very reasonable, since trailing produces about a 20% reduction in computation time in the average. Moreover, on deterministic programs (possibly leaving some choice points leading to failures), semantic backtracking is superior to trailing, since it exhibits little time and space overhead to support nondeterminism.

This work may also open a new perspective for the implementation of CLP systems in general. In particular, it would be interesting to investigate if semantic backtracking can be applied to other constraint systems.

References

[1] A. Aggoun and N. Beldiceanu. An Overview of the CHIP compiler. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[2] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[3] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[4] M. Dincbas, P. Van Hentenryck, H. Simonis, Ã. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS-88*, Tokyo, Japan, Dec. 1988.

[5] J.L. Imbert and P. Van Hentenryck. Efficient Handling of Disequations in CLP Over Linear Rational Arithmetics. In *Constraint Logic Programming: Selected Research.* MIT Press, 1993.

[6] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, Jan. 1987.

[7] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. An Abstract Machine for CLP($\Re$). In *PLDI'92*, San Francisco, CA, June 1992.

[8] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP($\Re$) Language and System. *ACM Transactions on Programming Languages*, 14(3):339–395, 1992.

[9] J.R. Schoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[10] P. Stuckey. Incremental Linear Constraint Solving and Detection of Implicit Equalities. *ORSA Journal of Computing*, 3:269–274, 1991.

[11] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[12] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 5(2-4), 1992.

[13] D.H.D Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, October 1983.

	Final State					Backtracking Points				
Programs	Int	Rat	Cell	Other	Total	Int	Rat	Cell	Other	Total
Donald	8180	4144	10080	5414	27818	9180	4664	11712	6480	32028
Donald1	9324	4824	10944	5238	30330	9356	4848	10992	5250	30446
SendMory	3540	1792	4056	3276	12664	3596	1840	4224	3302	12774
SendMory1	6396	3240	8064	3546	21246	6428	3264	8112	3608	21042
Investment (200)	50424	9576	19152	5612	84764					
Laplace (12)	14704	7320	19560	1534	43118					
Magic	4408	2232	3672	4156	14468	5076	2560	4608	5120	17004
Permutation						7064	3544	8040	6460	25108
Periodic						2216	1168	2784	932	7100
Square 9 (A)	13976	7056	18600	10360	49992	16476	8306	22536	10716	58032
Square 9 (F)	11344	5744	14664	6698	38450	15692	7928	21408	10162	54862
Square 14 (F)	35084	17648	48216	24176	125124	45368	22808	63984	32222	125124
Square 17 (F)	64788	32520	91296	41072	229676	73752	37000	105048	56880	272642

Table 1: Space Requirement of Algorithm `Semantic` in Bytes

	Final State					Backtracking Points				
Programs	Int	Rat	Cell	Trail	Total	Int	Rat	Cell	Trail	Total
Donald	11296	3528	10976	31096	56896	16764	4184	13792	45140	77680
Donald1	18496	6256	20320	40948	86020	18496	6352	20800	40964	86028
SendMory	7036	1776	5344	20240	34396	7036	1832	5600	20256	34412
SendMory1	7120	2480	7712	21108	38420	7120	2560	8288	21124	38436
Investment (200)	55992	9568	25504	43116	134180					
Laplace (12)	19784	7240	25760	29820	82604					
Magic	13008	2056	4192	35900	54844	11788	2408	5600	37528	58288
Permutation						18044	3896	12160	57500	92672
Periodic						3296	1072	3360	10136	18232
Square 9 (A)	35504	7256	22400	112348	177508	45768	9056	28840	125272	207712
Square 9 (F)	32388	6696	20440	97952	157476	41752	8560	27188	120316	194368
Square 14 (F)	121792	20440	75456	364404	582092	143440	24096	90400	386408	639752
Square 17 (F)	208076	33112	108584	620860	970632	249560	38656	128268	700104	1106920

Table 2: Space Requirement of Algorithm `Trail` in Bytes

	Final State					Backtracking Points				
Programs	Int	Rat	Cell	Other	Total	Int	Rat	Cell	Other	Total
Donald	0.72	1.17	0.92	0.17	0.49	0.55	1.11	0.85	0.14	0.41
Donald1	0.50	0.77	0.54	0.13	0.35	0.51	0.76	0.53	0.13	0.35
SendMory	0.50	1.01	0.76	0.16	0.37	0.51	1.00	0.75	0.16	0.37
SendMory1	0.90	1.31	1.05	0.17	0.55	0.90	1.28	0.98	0.17	0.56
Investment (200)	0.90	1.00	0.75	0.13	0.63					
Laplace (12)	0.74	1.01	0.76	0.05	0.52					
Periodic						0.67	1.09	0.83	0.09	0.39
Magic	0.39	1.09	0.88	0.11	0.26	0.43	1.06	0.82	0.13	0.29
Permutation						0.39	0.91	0.66	0.11	0.27
Square 9 (A)	0.39	0.97	0.83	0.09	0.28	0.36	0.92	0.78	0.09	0.28
Square 9 (F)	0.35	0.86	0.72	0.07	0.24	0.38	0.93	0.79	0.08	0.28
Square 14 (F)	0.29	0.86	0.64	0.07	0.21	0.32	0.95	0.71	0.08	0.20
Square 17 (F)	0.31	0.98	0.84	0.07	0.24	0.30	0.96	0.82	0.08	0.25

Table 3: Space Requirement: Ratio `Semantic/Trail`

	Minimum				Semantic				Trail			
Programs	Int	Rat	Cell	Total	Int	Rat	Cell	Total	Int	Rat	Cell	Total
Donald	4788	2464	5160	12412	1.92	1.89	2.27	2.58	3.50	1.70	2.67	6.26
Donald1	8688	4584	10440	23680	1.08	1.06	1.05	1.29	2.13	1.39	1.99	3.63
SendMory	2800	1480	3288	7568	1.28	1.24	1.28	1.69	2.51	1.24	1.70	4.55
SendMory1	3788	1992	4512	10292	1.70	1.64	1.80	3.05	1.88	1.29	1.84	3.73
Investment (200)	50408	9568	19128	79104	1.00	1.00	1.00	1.07	1.11	1.00	1.33	1.70
Laplace (12)	14544	7240	19320	41104	1.01	1.01	1.01	1.05	1.36	1.00	1.33	2.01
Magic	3748	1912	2616	8272	1.35	1.34	1.76	2.06	3.15	1.126	2.14	7.05
Permutation	5632	2904	6360	14896	1.25	1.22	1.26	1.69	3.20	1.34	1.91	6.22
Periodic	1612	864	1896	4372	1.37	1.35	1.47	1.62	2.04	1.24	1.77	4.17
Square 9 (A)	10336	5248	13416	29000	1.59	1.58	1.68	2.00	4.43	1.73	2.15	7.16
Square 9 (F)	9432	4792	12072	26296	1.66	1.65	1.77	2.09	4.43	1.79	2.25	7.39
Square 14 (F)	25044	12640	33504	71188	1.81	1.80	1.91	1.76	5.73	1.91	2.70	8.99
Square 17 (F)	38744	19496	52536	110776	1.90	1.90	2.00	2.46	6.44	1.98	2.44	9.99

Table 4: Space Requirement: Ratios over Strict Minimum Space

	Final State			Backtracking Points		
Programs	`Semantic`	`Trail`	ratio	`Semantic`	`Trail`	Semantic/Trail
Donald	19638	45600	0.43	22856	60916	0.38
SendMory	9124	27360	0.33	9222	27376	0.34
SendMory1	14850	31300	0.47	14980	31316	0.48
Periodic				4884	14976	0.33
Magic	10060	43012	0.23	12024	45892	0.26
Permutation				18044	73120	0.25
Square 9 (A)	36016	142004	0.25	41556	161944	0.26
Square 9 (F)	27106	125088	0.22	39170	152804	0.26
Square 14 (F)	90040	460300	0.20	119006	496312	0.24
Square 17 (F)	164888	762556	0.22	198890	857360	0.23

Table 5: Space Requirement on Finite Precision Numbers

Programs	Infinite Precision Trail	Infinite Precision Semantic	Infinite Precision ratio	Finite Precision Trail	Finite Precision Semantic	Finite Precision ratio	backtracks
Donald	1350	1570	0.86	970	1030	0.94	772
Donald1	10730	12220	0.88				3625
SendMory	115250	120280	0.96	74940	77320	0.97	51618
SendMory1	250	250	1.00	210	160	1.31	161
Investment (200)	16690	17040	0.98				0
Laplace (12)	490	470	1.04				0
Periodic	310	290	1.07	220	180	1.22	107
Magic	77280	64410	1.20	37850	41910	0.90	63983
Permutation	16280	23030	0.71	9710	11630	0.83	8751
Square 9 (A)	117050	180600	0.65	75590	97850	0.77	14948
Square 9 (F)	11050	16330	0.68	7150	9080	0.79	1511
Square 14 (F)	139030	216060	0.64	83420	120520	0.69	8813
Square 17 (F)	363340	576350	0.63	184950	279360	0.66	12505

Table 6: Computation Time in Milliseconds: Comparison of `Trail` and `Semantic`