A Simple Automation of a Peircean Decision Procedure

Richard B. White

Peirce's graphical sentential logic, the system of alpha graphs, is much more than an historical curiosity. Although a large alpha graph is difficult to manipulate "by hand," the alpha graphs are admirably suited to automation. In particular, a simple decision procedure for the alpha graphs is tailor-made for the programming language LISP. This decision procedure is described in section I below and is implemented by a LISP program CSProve in section II. Section III gives examples of CSProve as it ran on a desktop computer and indicates how the program is easily adapted to parallel processing.

The decision procedure differs from tableau methods and the widely used Robinson resolution method in at least two interesting ways: unlike resolution it does not require that a formula be put into a normal form before the method can be applied, and unlike both tableau methods and resolution it does not require any branching tree constructions. If the method were formulated as a proof procedure in the usual notation it would have only four one-premise inference rules of the form:

$$\frac{A_1 \wedge \ldots \wedge A_m}{B_1 \wedge \ldots \wedge B_n},\ m,n \geq 1.$$

Don D. Roberts discovered that by 1903 Peirce had a decision procedure for the alpha graphs.[1] Peirce's method is surprisingly complex in comparison with the method of this paper, a method that now seems almost obvious; perhaps, to use Peirce's own words, this "is a curious example of how late a development simplicity is."[2]

The reader is assumed to be familiar with the alpha graphs but not with LISP. A brief exposition of the little fragment of LISP needed for CSProve is given in section II. Readers familiar with LISP may skip that and go directly to the program, which I hope they can improve.

I

Alpha graphs will be written with parentheses in place of the closed curves Peirce used for "cuts." (Peirce himself sometimes made this replacement.[3]) This has two advantages: it is typographically convenient, in fact necessary for a complicated graph, but more importantly it turns an alpha graph immediately into a LISP list. (The name LISP was given to this language to suggest "list processing.")

The decision procedure uses four operations for transforming graphs on the sheet of assertion:

O1. A graph (- - - (g_1 . . . g_n) - - -) may be erased if graphs of type g_1, . . ., g_n occur, in any order, on the area indicated by the dashes. For example, the graph ((A) (B (A) C) C (A) B) may be erased from the sheet. Any graph erased by O1 could be returned to the sheet by using Peirce's rules. For instance, the graph of the example could be written on the sheet by first writing (( )), then ((A) ( ) C (A) B) by insertion on an odd area, and then ((A) (B (A) C) C (A) B) by iteration into an even area. Therefore O1 is an equivalence transformation; the content of the sheet after an application of O1 is logically equivalent to the content of the sheet before that application. The other three rules also are equivalence transformations in that sense.

O2. If O1 is not applicable and a graph ((g)) is on the sheet, then it is replaced with g.

O3. If neither O1 nor O2 is applicable and there is a graph (- - - ((g)) - - -) on the sheet, then that graph is replaced with (- - - g - - -).

O4. If none of O1, O2, O3 is applicable and there is a graph (- - - (g_1 . . . g_n) - - -) on the sheet, $n \geq 2$, replace that graph with (- - - (g_1) - - -) (- - - (g_2 . . . g_n) - - -). Again this is an equivalence transformation. Given (- - - (g_1 . . . g_n) - - -), it may be repeated to get (- - - (g_1 . . . g_n) - - -) (- - - (g_1 . . . g_n) - - -), and then (- - - (g_1) - - -)(- - -(g_2 . . . g_n) - - -) by erasures on an even area. On the other hand, from (- - - (g_1) - - -)(- - - (g_2 . . . g_n) - - -) one gets (- - -(g_1 (- - - (g_2 . . . g_n) - - -)) - - -) by iteration into an even area and erasure on the sheet. From this follows (- - - (g_1((g_2 . . . g_n))) - - -) by deiteration from an odd area of graphs on the area indicated by the dashes. Then (- - -(g_1 . . . g_n) - - -) by removal of a double cut.

To decide whether or not the graph on the sheet is valid, i.e. tautologous, apply O1–O4 until either the sheet is emptied, in which case the graph with which the procedure was begun is valid, or else at some point at least one graph on the sheet will either be atomic or of the form (g_1 . . . g_n), $n \geq 1$, to which O1 is not applicable and with each g_m either atomic or of the form (B) for atomic B. From this graph one can read off a valuation that falsifies the original graph on the sheet. Two examples illustrate the method.

Let the sheet contain only the graph ([(A [B]) (A)] [A]). (Here, as elsewhere, brackets may be substituted for some parentheses to improve readability.) Only O4 is applicable. Applying it replaces the graph with ([(A [B]

)] [A]) ([(A)] [A]). By O1 the right-hand graph is erased to leave ([(A [B])] [A]). Then an application of O3 replaces that graph with (A [B] [A]) which is erased by O1. The sheet has been emptied, showing that the initial graph is a tautology.

Let the graph ([([A (C)])([(C) B])] [(A) (B)] [C]) be the only graph on the sheet. Only O4 is applicable. It can be applied in two ways. Applying it via the left-most subgraph in virtue of which O4 is applicable replaces the graph with:

([([A (C)])] [(A) (B)] [C]) ([([(C) B])] [(A) (B)] [C]).

Applying O3 to this gives ([A (C)] [(A) (B)] [C]) g, where g is the right-hand graph in the preceding.

Another application of O4 yields ([A] [(A) (B)] [C]) ([(C)] [(A) (B)] [C]) g. O1 then results in ([A] [(A) (B)] [C]) g. O4 replaces this with ([A] [(A)] [C]) ([A] [(B)] [C]) g. O1 then gives ([A] [(B)] [C]) g. By O3 this is replaced by ([A] B [C]) g. But no rule can erase ([A] B [C]), so the procedure halts and we can read off a counterexample to the original graph: make A false, B true, and C false. It is easily verified that this valuation falsifies the graph with which the procedure began, which is equivalent in meaning to: $((A \vee B) \wedge \neg C) \rightarrow ((A \rightarrow C) \wedge (B \rightarrow C))$.

The decision method given by O1-O4 is evidently sound and complete. The procedure starts with a graph $g_1 \ldots g_n$, $n \geq 1$. O1, O2, and O3 replace a graph with a simpler graph. O4 replaces a graph $h_1 \ldots h_m$ with a longer graph, e. g. $h_{11}h_{12}h_2 \ldots h_m$, but the graphs h_{11} and h_{12} are simpler than the graph h_1 which they replace. Therefore the procedure must eventually terminate. If a graph $j_1 \ldots j_m$ is reached with a j_k that is either atomic or of the form $(f_1 \ldots f_p)$ where each f_i is either atomic or (B) for some atomic B and O1 cannot erase j_k, then j_k can obviously be falsified, thereby falsifying $j_1 \ldots j_m$ and the initial graph $g_1 \ldots g_n$ to which $j_1 \ldots j_m$ is logically equivalent.

Although O1–O4 are sufficient to decide any alpha graph, for an efficient automation of the decision procedure it is desirable to modify O4 and to add a rule of contraction which is a special case of Peirce's deiteration rule. An example will illustrate how O4 may be refined. Suppose a graph (1) (A [B C] E [B D] F) is on the sheet. This graph is equivalent to (2) (A [B] E F) (A [C] E [D] F). For (1) is transformable to (A [B] E [B] F) (A [C] E [D] F) by repeating (1) and erasing on even areas, and that graph is equivalent to (2). On the other hand, from (2) follows (A [(A [B] E F) C] E [(A [B] E F) D] F) by iterating (A [B] E F) onto even areas. Then deiteration of A, E and F from odd areas transforms this to (A [([B]) C] E [([B]) D], and (1) follows from this by removing double cuts. Generalizing from this example, the modified O4 becomes O4*: a graph $(X\ (g_1 \ldots g_n\ )Y)$, $n \geq 2$, with not both areas X and Y empty, is replaced with $(X_1\ (g_1)\ Y_1)(X_2\ (g_2 \ldots g_n\ )\ Y_2)$ in which X_1 and Y_1 are the results of removing all graphs $(W\ g_1\ Z)$, for any W and Z, from X and Y, and X_2 and Y_2 are the results of replacing each graph (W g_I Z) in X or Y with (W Z). An application of O4* will be called an *expansion* of the sheet *via* (g_I . . . g_n).

The contraction operation (Con) is a special case of Peirce's deiteration: a graph (X) on the sheet, X not empty, is replaced with (X_I), in which X_I results from X by removing from any graph (g_I . . . g_n) in X all g_m's that are also in X. For example. Con replaces the graph ([A] [B (A)] C [C D]) on the sheet with ([A] [B] C [D]). This accomplishes in one step what OI–O4 would do in four:

([A] [B (A)] C [C D])
([A] [B] C [C D]) ([A] [(A)] C [C D]) by O4
([A] [B] C [C D]) by OI
([A] [B] C [C])([A] [B] C [D]) by O4
([A] [B] C [D]) by OI

OI–O4*, Con are designed to minimize the size of the graph on the sheet during the decision process; as LISP programs the simpler OI–O4 generally require the execution of fewer instructions than OI–O4*, Con.

A graph is *prime* if it is either atomic or of the form (g) for atomic g. A graph is *irreducible* if it is prime or of the form (g_I . . . g_n), $n \geq 2$, with each g_m prime. The OI–O4, Con decision procedure may now be presented in a form that is straightforwardly convertible to a LISP program. A line of the form Q? m; n means "if the answer to question Q is affirmative, then go to step m; otherwise go to step n."

1. Is the sheet empty? 2; 3
2. Output "valid" and halt.
3. Is OI applicable? 4; 5
4. Apply OI and go to I.
5. Is there an irreducible graph on the sheet? 6; 7
6. Output an irreducible graph from the sheet and halt.
7. Is O2 applicable? 8; 9
8. Apply O2 and go to I.
9. Is O3 applicable? I0; II
10. Apply O3 and go to I.
11. Is Con applicable? I2; I3
12. Apply Con and go to I.
13. O4* must be applicable. Apply O4* and go to I.

An irreducible graph may be erasable by OI, but such a graph will never be an output resulting from step 6 because it will be erased at step 4.

An example illustrates the program: Let the sheet contain only ([A B (C)] [A (B) D] [A (B) (C) (D)] [(A) D] [(A) (C) (D)] [C]). Given this, the program will reach step II and find that the graph is contractible. Applying Con gives ([A B] [A (B) D] [A (B) (D)] [(A) D] [(A) (D)] [C]). With this graph the program will reach step I3. Applying O4* via [A B] (which is what the LISP version will do) yields ([A] [(A) D] [(A)(D)] [C]) ([B] [(B) D] [(B) (D)] [(A) (D)] [C]). From this by Con comes ([A] [D] [(D)] [C]) ([B] [D] [(D)] [(A) (D)] [C]), and then the program will apply O1 twice to empty the sheet and give the output "valid."

II

LISP was invented in the late 1950's and has steadily evolved into a very powerful general-purpose programming language. However, for the program CSProve only a very small fragment of LISP is needed.

For our purposes a LISP expression is either an *atom* or a *list*. Typical atoms are A, B, C, AB, a, aB, and numerals (not numbers) 1, 2, 3, etc. A list is a sequence of LISP expressions enclosed in parentheses. For example, (C (A B) ((D) A)) is a list with three members: the atom C, the list (A B) whose members are the atoms A and B, and the list ((D) A) whose members are the one-membered list (D) and the atom A. One must take care to separate atoms in a list by spaces. For example, the two-membered list (A B) contains the atoms A and B, but (AB) has just one member, the atom AB. A list may be empty. The empty list is NIL or ( ). NIL does double duty in LISP; it is the empty list but it also serves as the truth-value "false."

In LISP a function f of n inputs is applied to $e_1, \ldots, e_n$ by executing the expression (f $e_1 \ldots e_n$). For example, executing (+ 2 3) will return 5, or briefly (+ 2 3) *is* 5. The program CSProve uses several basic functions that are built in to LISP, along with some functions that will be defined in terms of these functions. The following functions are available in any current dialect of LISP.

first. First applied to a list L returns the first, i.e. the leftmost, member of L. For example, (first '((A B) C ((D) A))) is (A B). The apostrophe before the list input to first, as to other functions that operate on lists, is crucial. It tells LISP to regard what follows the apostrophe simply as a list of expressions, not as a function to be evaluated. For instance, (first '(+ 2 2)) will return +, but (first (+ 2 2)) will return an error message such as "Error. Invalid list 4." This is because given (first (+ 2 2)) LISP first evaluates (+ 2 2) by adding 2 to 2 and then tries to take the first member of the result 4, which is nonsensical.

rest.[4] Rest applied to a list x returns x minus its first member. For example, (rest '((AB) C (D))) is (C (D)). (rest NIL) is NIL.

second. Second applied to a list x is the second member of x, if there is such; otherwise it is NIL (second '(A (B C) D) is (B C).

third. Third given a list x returns the third member of x, if there is such; otherwise it is NIL. (third '(A B (C)(D E))) is (C).

The above functions can of course be composed. For example, (first (rest '(A B C))) is B, and (third (second (second '(A (B (C D E)))))) is E.

The functions cons, list, and append are used to form lists.

cons. Given an expression e and a list x, (cons e x) gives the list whose first member is e and whose rest is x. (cons '(A B) '(C D)) is ((A B) C D). (Cons 'A NIL) is (A).

list. Given expressions $e_1, \ldots, e_n$, n ≥ 1, list forms the list whose members are $e_1, \ldots .e_n$. (list '(A) 'C 'D) is ((A) C D).

append. Append unites two lists by appending the second to the first. (append '(A B) '(C (D))) is (A B C (D)). (append '(A B) NIL) is (A B).

mapcar. Given a function f and a list x, (mapcar #'f x) gives the list which results from x by replacing each member e of x with (f e). (mapcar #'list '(A B C) is ((A) (B) (C)). (mapcar #'first '((A B) (C D) (E F G))) is (A C E).

A LISP predicate is a function that outputs either t ("true") or NIL ("false"). Traditionally the names given to predicates end with p, but there are exceptions to this, as in *atom, null,* and *equal.*

atom. (atom x) is t if x is an atom, NIL if x is not an atom. (atom '(A B)) is NIL. (atom 'A) is t.

listp. (listp x) is t if x is a list, NIL if x is not a list. (listp '(A B C)) is t. (listp 'A) is NIL. (Listp NIL) is t.

null. (null x) is t if x is the empty list; otherwise it is NIL. (null '( )) is t. (null '(A B)) and (null 'A) are NIL.

equal. (equal x y) is t if x and y are the same LISP expression; it is NIL otherwise. (equal 'a 'A) is t. (LISP does not distinguish between upper and lower cases.)

every. For a predicate P and a list x, (every #'P x) is t if every member of x satisfies P; otherwise it is NIL. (every #'atom '(A Aa B)) is t. (every #'listp '((A) B)) is NIL.

subsetp. Given two lists x and y, (subsetp x y :test 'equal) is t if every member of x is also a member of y. (subsetp '(A (B C) A) '((D) (BC) A)) :test 'equal) is t. (subsetp '(A B (C)) '(B (C)) :test 'equal) is NIL. (subsetp '( ) '(A B C) :test 'equal) is t. (The empty list is a subset of every list.) The part ":test 'equal" in subsetp tells LISP to test x and y to see if every member of x is equal to some member of y. It is necessary because there are identity predicates other than equal in LISP. (For example, = is equality between numbers.)

member. (member x y :test 'equal) is t if and only if y is a list of which x is a member. (member '(A) '(B (A) C (A)) :test 'equal) is t. (member '(A) '(B D)) :test 'equal) is NIL.

<. The predicate < applies to numbers. (< m n) is t if m is less than n; otherwise it is NIL. (< 2 3) is t; (< 3 2) and (< 2 2) are NIL.

CSProve uses one function whose value is a number when it is applied to a list.

length. Given a list L, (length 'L) is the number of members of L. (length '(A (B C) D)) is 3. (length '( )) is 0.

The next four functions perform useful operations on lists. Two of them, like member and subsetp, use equality as a test.

find-if. For any predicate P and list x, (find-if #'P x) is the first member of x to which P applies; if there is no such member then (find-if #'P x) is NIL. (find-if #'listp '(A (BC) D (E F))) is (B C). (find-if #'listp '(A B C)) is NIL.

remove. Given x and a list y, (remove x y :test 'equal) is the result of removing all occurrences of x from y. If x is not a member of y then (remove x y :test 'equal) is y' (remove '(A) '(B (A) ((A) C) D (A))) :test 'equal) is (B ((A) C) D). (remove 'A '(B C) :test 'equal) is (B C).

remove-if. For a predicate P and a list x, (remove-if #'P x) is the list that results from x by removing all members that have P. If P applies to no member of x then (remove-if #'P x) is x. (remove-if #'atom '(A (B C) D (E)) is ((B C) (E)). (remove-if #'listp '(A B C)) is (A B C).

remove-duplicates. This function removes repetitions of items in lists. (remove-duplicates '(A (B C) D (B C) A A) :test 'equal) is (A (B C) D). (remove-duplicates '(A (B) C)) :test 'equal) is (A (B) C).

There are several conditionals in LISP. CSProve uses only one:

cond. A cond instruction takes the form

```
(cond (T_1 R_1)
      (T_2 R_2)
        .
        .
        .
      (T_n R_n ))
```

Cond works by running through the T_i's until it finds the first T_m that is not NIL. Then it gives the output R_m . If every T_i is NIL, cond outputs NIL. For example,

```
(cond ((not(listp x)) x)
      ((listp x) (first x)))
```

defines a function that is x if x is not a list and is the first member of x if x is a list. A convenient way of insuring that cond gives a non-NIL result R_n is to make the last clause have the form (t R_n). This has the effect of saying "if none of the above, then R_n", since t is not NIL. In a cond definition n must be at least 2.

The usual truth functions are built into LISP.

not. (not x) is NIL if x is not NIL and t if x is NIL. (not (atom 'A)) is NIL. (not (listp 'A)) is t.

or. (or $x_1 \dots x_n$), $n \geq 2$, gives the first non-nil x_i if at least one of $x_1 \dots x_n$ is not NIL; otherwise it is NIL. (or (atom 'A) (listp 'A)) is t. (or '(A B) (listp 'A)) is (A B). (or (atom '(A B)) (listp 'A)) is NIL.

and. (and x_I . . . x_n) is x_n if all the x_i's are non-NIL; it is NIL if at least one x_i is NIL. (and (atom 'A) '(A B)) is (A B). (and (listp 'A) (atom 'B)) is NIL.

If the inputs to and, or, and not are restricted to NIL and t, then these functions are just the usual two-valued truth functions.

format. The *format* function is used to print expressions in LISP. For example, executing (format t "valid") causes "valid" to appear on the computer monitor's screen, hence the "t" (for computer terminal) in the format instruction. A more elaborate version of the format function uses a place-marker ~a inside the quotation and accepts an input. For example, (format t "~a is a Deke" (second x)) when x is assigned a list as its value will print the quoted expression with the second member of the list in place of ~a. (format t "~a is a Deke"(second '(George W Bush)) prints "W is a Deke."

defun. The function defun ("define a function") can be used to define the functions that make up a program. For example, (defun atomfirstp (x) (and (listp x)(atom (first x)))) may be read, "define a function, to be called "atomfirstp," of one input x as follows: its value for input x is (and(listp x)(atom(first x)))." Thus atomfirstp is a predicate which is t if its input is a list whose first member is an atom and NIL otherwise.

lambda. In LISP lambda abstracts may be used to name functions. For instance, the thirteenth function in CSProve is "remifmem" which when given x and a list y removes from y all lists of which x is a member. Clearly one wants to use remove-if to define this function, but remove-if requires a predicate and LISP has no predicate that applies to the lists of which x is a member, for a variable x. But such a predicate can be defined by a lambda expression: (lambda (z) (and (listp z) (member x z :test 'equal))), which may be read "the property of being a z such that z is a list and x is a member of z." (The lambda comes from Church's notation for functions in his lambda calculi, where for example (λx. x(xx)) is the function such that (λx. x(xx))a = a(aa) for any a.) The desired function remifmem is then defined by (defun remifmem (x y) (remove-if #'(lambda (z) (and(listp z)(member x z :test 'equal))) y)). In a lambda expression (lambda (z) . . .z . . .) the variable z is bound, so (lambda (w) . . .w . . .) is the same function as (lambda (z) . . . z . . .), for any variable w that does not occur in . . . z . . . .

The program CSProve uses nineteen functions defined by defun instructions. The first seventeen functions are enough to automate the decision procedure, which is implemented by the function *test*. Following the program are brief explanations of its functions. The first seventeen definitions in the program are the following:

CSProve

```
(defun primfp (x) (or (atom x)(and (listp x) (atom (first x))(null (rest x)))))
(defun primlp (x) (and (listp x)(every #'primfp x)))
```

```
(defun irreducp (x) (or (primfp x)(primlp x)))
(defun mlistp (x) (and (listp x)(< 1 (length x))))
(defun dnegp (x) (and (listp x)(listp (first x))(null (rest x))))
(defun erasablep (x) (and (listp x) (find-if #'(lambda (w) (equal t w))
          (mapcar #'(lambda (z) (and (listp z)(subsetp z x :test 'equal))) x))))
(defun remby (x y) (cond ((not (and (listp x)(listp y))) x)
          (t (remove-if #'(lambda (z) (member z y :test 'equal)) x))))
(defun contract (x) (mapcar #'(lambda (z) (remby z x)) x))
(defun contratiblep (x) (not (equal (contract x) x)))
(defun remdnegs (x) (cond ((not(listp x)) x)
               ((null x) x)
               ((dnegp (first x)) (append (first (first x)) (remdnegs (rest x))))
               (t (cons (first x)) (remdnegs (rest x))))))
(defun remfrom (x y) (cond ((not (listp y)) y)
        (t (remove x y :test 'equal))))
(defun remfrey (x y) (mapcar #'(lambda (z) (remfrom x z)) y))
(defun remifmem (x y) (remove-if #'(lambda (z) (and (listp z) (member x z :test
   'equal))) y))
(defun firstfml (x) (first (find-if #'mlistp (first x))))
(defun expansion (x) (append (list (cons (list (firstfml x)) (remifmem (firstfml
   x) (first x)))
        (remfrey (firstfml x) (first x))) (rest x))))
(defun test (x) (cond ((null x) (format t "valid"))
        ((erasablep x) (test (rest x)))
        ((irreducp (first x)) (format t "~a is a counterexample" (remove-
           duplicates (first x) :test 'equal)))
        ((dnegp (first x)) (test (remdnegs x)))
        ((find-if #'dnegp (first x)) (test (cons (remdnegs (first x)) (rest x))))
        ((contractiblep (first x)) (test (cons (contract (first x)) (rest x))))
        ((find-if #'mlistp (first x)) (test (expansion x)))))
(defun decidegraph (x) (test (list x)))
```

The first function *primfp* ("— is a prime formula") is a predicate such that (primfp x) is t if and only if x is either an atom or a list (A) in which A is an atom; i.e. a list whose first member is an atom and whose rest is the empty list.

The second function *primlp* ("— is a prime list") is also a predicate. (primlp x) is t if and only if x is a list and every member of x is a prime formula.

The next predicate *irreducp* ("— is irreducible") is true of x if and only if x is either a prime formula or a prime list.

The fourth function *mlistp* is a predicate that is true of x if and only if x is a list with more than one member.

The fifth function, *dnegp,* may be read "— is a double negation." (dnegp x) is t if and only if x is a list of the form ((. . .)); i. e., a list whose first is a list and whose rest is empty, hence the definition of dnegp by (and (listp x)(listp (first x))(null (rest x))).

The sixth function is a predicate *erasablep* which, when applied to a list x, is t if and only if x has at least one member that is a subset of x. It works as follows. First mapcar applies the predicate (lambda (z) (and (listp z) (subsetp z x :test 'equal))) ("the property of being a list that is a subset of x") to x. (mapcar #'(lambda (z) (and (listp z)(subsetp z x :test 'equal))) x) is then a list (v_I . . . v_n), when x is (e_I . . . e_n), with each v_i being t or NIL according as e_i is a subset of x or not. If at least one v_i is t then (erasable x) should be t; therefore erasablep is (find-if #'(lambda (w) (equal t w)) applied to (v_I . . . v_n).

The next function *remby* is defined by a conditional. If x and y are not both lists then (remby x y) is x. If x and y are lists then (remby x y) is (remove-if #'(lambda (z) member z y :test 'equal) x), so (remby x y) is the result of removing from x all members that are members of y. (remby x y) may be read "removal from x by way of y."

The eighth function *contract* is easily defined using remby. A list x is contracted by removing from each list in x all members that are also members of x, hence the definition of (contract x) by (mapcar #'(lambda (z) (remby z x)) x).

A list is *contractiblep* if it differs from its contraction, so (contractiblep x) is defined by (not (equal (contract x) x).

The tenth function *remdnegs* may be read "the result of removing all double negations from —." It is the first function in CSProve defined recursively—remdnegs occurs in some of the clauses of the conditional defining remdnegs. If x is not a list then the result of removing double negations from x is just x, so the first clause in the cond is ((not (listp x)) x). If x is the empty list then the result of removing double negations from x is again x; therefore the second clause in the cond is((null x) x). If x is of the form (((e_I . . . e_n)) f_I . . . f_m), then (dnegp (first x)) is true. in this case appending (first (first x)), which is (e_I . . . e_n) to (- - -), the result of removing all double negations from (f_I . . . f_m), which is (rest x), will give a list (e_I . . .e_n - - -) that contains no double negations. Therefore the third clause in the cond is: ((dnegp (first x)) (append (first (first x))(remdnegs (rest x)))). If x is (e - - -) with e not a double negation, then (cons (first x) (remdegs (rest x))) is (e . . .), where (. . .) is the result of removing all double negations from (- - -). Therefore the final clause in the cond defining remdegs is (t (cons (first x) (remdnegs (rest x)))). Remdnegs removes double negations only from x, not from members of x, members of members of x, and so on. For example, (remdnegs '(A ((B ((C)))) ((D ((E F)))))) is (A B ((C)) D ((E F))).

The eleventh function in the program is *remfrom.* (remfrom x y) is y if y is not a list; otherwise it is (remove x y :test 'equal), the result of removing all occurrences of x from y.

The function *remfrey* uses remfrom to remove x from every member of a list y, so it is defined by (mapcar #'(lambda (z) (remfrom x z)) y).

The thirteenth function, *remifmem,* is such that (remifmem x y) is the result of removing from a list y all members of which x is a member. It was explained above.

The function *firstfml* may be read " the first member of the first mlist in - -." (firstfml x) is therefore defined as (first (find-if #'mlistp x)).

The list that the decision procedure works on will be called *the sheet.* The fifteenth function in CSProve, *expansion,* applies the operation O4* to the sheet. An example illustrates its operation. Suppose the sheet is ((A (B C) (D B) E) F G), and let this list be x. Expanding x via (B C) will change the sheet to (((B) A E) (A (C) (D) E) F G). In this example (A (B C) (D B) E) is (first x), (F G) is (rest x), B is (firstfml x), (B) is (list (firstml x)), (A E) is (remifmem #'(firstfml x) (first x)), ((B) A E) is (cons (list (firstfml x)) (remifmem #'(firstfml x) (first x))), (A (C) (D) E) is (remfrey (firstfml x) (first x)), (((B) A E) (A (C) (D) E)) is (list (cons (list (firstfml x)) (remifmem #' (firstfml x) (first x))) (remfrey (firstfml x) (first x))), and (((B) A E) (A (C) (D) E) E F) is (append (list (cons (list (firstfml x)) (remifmem #' (firstfml x) (first x))) (remfrey (firstfml x) (first x))) (rest x)), which is the definition of (expansion x).

Now the function *test* is defined recursively by a conditional with seven clauses. If the sheet has been emptied then the decision procedure returns "valid", so the first clause is ((null x) (format t "valid")). If the first member of the sheet is erasable, then test moves on to test the rest of x. (This is equivalent to erasing the first member of the sheet.) The second clause in the cond is then ((erasablep (first x)) (test (rest x))). If the first member of the sheet is irreducible [but not erasable if this point has been reached] then that first member is a counterexample, so the third clause is ((irreducp (first x)) (format t "~a is a counterexample" (remove-duplicates (first x) :test 'equal))). (It is not necessary to remove duplicates in this case, but doing so often makes the counterexample easier to read.) If the first member of the sheet is a double negation, then test removes all double negations from the sheet and tests the resulting sheet, so the fourth clause is ((dnegp (first x)) (test (remdnegs x))). If there is a double negation in the first member of the sheet, then test removes all double negations from the first member of the list and tests the resulting sheet. The fifth clause is therefore ((find-if #'dnegp (first x)) (test (cons (remdnegs (first x)) (rest x)))). If the first member of the sheet is contractible, test contracts that first member and tests the resulting list; thus the sixth clause is ((contractible (first x)) (test (cons (contract (first x)) (rest x)))). Finally, if none of the first six clauses was applicable there must be a list with more than one member in the first member of the sheet, so test expands the sheet and tests the result. The last clause in the cond is therefore ((find-if #'mlistp (first x)) (test (expansion x))).

The function *decidegraph* is used to decide a graph g by testing the sheet whose only member is g: (decidegraph x) is therefore (test (list x)).

CSProve can be used at this point by decidegraph. For example, executing (decidegraph '((A (C)) (B (C)) ((((A) (B)) (C))))) will return "valid," and (decidegraph '((((A (C)))((B (C)))) ((((A) (B)) (C))))) will return "(B (C) (A)) is a counterexample."

However, in general it is not feasible to type alpha graphs directly into CSProve. For instance, the simple formula that would usually be written (A

↔ (B ↔ (C ↔ D))) becomes ((A ((B (C D) ((C) (D)))) ((B) (C D) ((C) (D)))))) ((A) ((B ((C D) ((C) (D)))) (B) (((C D) ((C) (D)))))) as an alpha graph.

Therefore it is convenient to have a connective notation (CN) that is easily typed on a computer keyboard and to add to CSProve a function that will translate formulas from CN into Peirce's graphical notation (PN). The formulas of CN are defined recursively as follows. The LISP atoms A, B, . . ., Z, 1, 2, 3 etc. are atomic CN formulas. If α and ß are CN formulas then (not α), (α imp ß), and (α iff ß) are CN formulas. If $\alpha_1, \ldots \alpha_n$ are CN formulas, $n \geq 2$, then (or $\alpha_1 \ldots \alpha_n$) and (& $\alpha_1 \ldots \alpha_n$) are CN formulas.

A CN formula is translated into PN by the following algorithm, where $^*\alpha$ is the translation of α: $^*\alpha$ is α if α is atomic. *(not α) is ($^*\alpha$). *(α imp ß) is ($^*\alpha$ (*ß)). *(α iff ß) is (($^*\alpha$ *ß) (($^*\alpha$) (*ß))). *(or $\alpha_1 \ldots \alpha_n$) is (($^*\alpha_1$) . . . ($^*\alpha_n$)). *(& $\alpha_1 \ldots \alpha_n$) is (($^*\alpha_1 \ldots {}^*\alpha_n$)).[5]

This algorithm is easily implemented by a function *trans* to be added to CSProve.

```
(defun trans (x) (cond ((atom x) x)
                    ((equal 'not (first x)) (list(trans(second x))))
                    ((equal 'imp (second x)) (list (trans(first x))(list(trans(third x)))))
                    ((equal 'or (first x)) (mapcar #'list (mapcar #'trans (rest x))))
                    ((equal '& (first x)) (list(mapcar #'trans (rest x))))
                    ((equal 'iff (second x)) (list(list(trans(first x)) (trans(third x)))
                         (list(list(trans(first x))) (list(trans(third x))))))))
```

Finally, the nineteenth function *decide* applies the decision procedure to a CN formula: (defun decide (x) (test(list(trans x)))).

III

CSProve was tested on a PC with a 1.6 gh processor and 512 mb of RAM running common lisp in the form of CormanLisp 2.01.[6] To evaluate an argument with premises P_1, .., P_n and conclusion C in CN, one executes (decide '((& $P_1 \ldots P_n$) imp C)). For example, the argument

$A \leftrightarrow (B \leftrightarrow C)$
$\neg A \leftrightarrow \neg B$
$C \rightarrow ((D \vee E) \rightarrow F)$
$\neg(G \rightarrow \neg D)$
$\underline{F \leftrightarrow H}$
$H \vee I$

was found to be valid in .009 seconds by executing:
(decide '((& (A iff (B iff C))((not A) iff (not B))(C imp ((or D E) imp F))(not (G imp (not D))) (F iff H)) imp (or H I))).

The argument

$A \rightarrow (B \rightarrow \neg C)$
$D \rightarrow B$

$\neg F \rightarrow C$
$\neg A \rightarrow \neg E$
$J \rightarrow \neg H$
$G \rightarrow J$
$\neg I \rightarrow \neg H$
$G \leftrightarrow \neg D$
$B \rightarrow H$
$\underline{\neg D \rightarrow H}$
$E \leftrightarrow F$

was found to be invalid by executing:

(decide '((& (A imp (B imp (not C))) (D imp B)((not F) imp C)((not A) imp (not E))(J imp (not H))(G imp J)((not I) imp (not H)) (G iff (not D))(B iff H)((not D) imp H)) imp (E iff F))) which returned "(I (J) (A) B (G) (C) D H (E) F) is a counterexample" ("make I, B, D, H, F true and J, A, G, C, E false") in .008 seconds. Examples like the above show that CSProve may be pedagogically useful, since it should decide even the most complex sentential arguments in introductory texts within a fraction of a second.

Of course, it is easy to tax CSProve by giving it very large graphs. For instance, 22 seconds were required for the program to announce "valid" when (decide '(A iff (B iff (C iff (D iff (E iff (F iff (G iff (H iff (A iff (B iff (C iff (D iff (E iff (F iff (G iff H))))))))))))))) was executed. In general, CN formulas with lots of nested biconditionals are challenging for CSProve, as they are for the resolution method. An alpha graph for the above CN formula is huge, as may be seen by using trans to translate it into PN.

The "pigeon-hole" tautologies are also hard for CSProve. The simplest of these tautologies is P2: (((A) (B)) ((C) (D)) ((E)(F)) (A C) (A E) (C E) (B D) (B F) (D F)), which decidegraph found to be valid in .0014 seconds. P3, (((A) (B) (C)) ((D) (E) (F)) ((G) (H) (I)) ((J) (K) (L)) (A D) (A G) (A J) (D G) (D J) (G J) (B E) (B H) (B K) (E H) (E K) (H K) (C F) (C I) (C L) (F I) (F L) (I L)), required .022 seconds. P4 was pronounced valid in .33 seconds and P5 in 5 seconds. But P6, which has one hundred and thirty-three members using forty-two atoms, needed 76 seconds.[7] (This is not too surprising, because a truth-table for P6 has over four trillion rows: $2^{42} =$ 4,398,046,511,104.) P7 is surely beyond the capability of the modest computer used for these tests.

CSProve operates serially, but it is easy to see how it could be used for parallel computation. Let C_1, C_2, . . . be a network of computers with each C_n running CSProve. A computation begins by starting C_1 with a sheet (g_1). When C_1 performs an expansion of the sheet to get (g_{11} g_{12}), it sends the sheet (g_{11}) to C_2 and continues to work on (g_{12}). Each computer works in the same way: when C_m expands a sheet (h) to (h_1 h_2) it sends (h_1) to some C_n and retains (h_2). This is a considerable idealization, because it assumes that an unoccupied C_n is always available; however, even a network of two computers would reduce the time required to decide P6 from 76 seconds to no more than 62 seconds. This is seen by executing (decidegraph(first(expansion (list '(- - -))))) and (decidegraph(second(expansion (list '(- - -))))), where (- - -) is P6. The first announced "valid" in 12 seconds, and the second reached "valid" in 62 seconds. Actually a two-computer network would decide P6 in less than 62 seconds, because after 12 seconds C_1 would be available to receive inputs from C_2. Similarly, a four-computer network could decide P6 in no more than 49 seconds, because (decidegraph(second(expansion (list (second (expansion (list '(- - -)))))))) produced "valid" in 49 seconds.

The pigeon-hole tautologies are examples of graphs in disjunctive normal form. Such a normal form is easy to describe in PN: it is a list of irreducible lists. An interesting exercise would be to provide an upper bound to the number of steps needed for the decision procedure to decide a disjunctive normal form tautology in serial and parallel modes.[8]

Centre College
whiterob@mikrotec.com

NOTES

1. Don D. Roberts, "A Decision Method for Existential Graphs," in *Studies in the Logic of Charles Sanders Peirce*, ed. Nathan Houser, Don D. Roberts, and James Van Evra, 387–401 (Bloomington, IN: Indiana University Press, 1997)
2. Peirce, CP 4.434
3. For example, in CP 4.378.
4. "First" and "rest" are relatively new names for the functions that were originally called "car" and "cdr" (pronounced "cudder" or "kidder"). These came from references to a long-gone IBM computer; car stood for "contents of address register" and cdr for "contents of decrement register." Car and cdr are still available in LISP, along with many combinations of them, such as caar, cadr, caadr, and caddr. These are handy abbreviations. For example, (caddr x), pronounced "Cudidder x", is equivalent to (car(cdr(cdr x))), or (first(rest(rest x))).
5. Since a CN formula other than an atom must be translated to a list, trans uses a double negation ((*α_1 . . . *α_1)) to translate a conjunction (& α_1 . . . α_n) because *α_1 . . . *α_n is not a list.
6. CormanLisp is available on the internet at *www.cormanlisp.com*. It has a timer function, an efficient "garbage collector", and is relatively inexpensive.
7. Using numerals for atoms, P6 is:
(((1) (2) (3) (4) (5) (6)) ((7) (8) (9) (10) (11) (12)) ((13) (14) (15) (16) (17) (18)) ((19) (20) (21) (22) (23) (24)) ((25) (26) (27) (28) (29) (30)) ((31) (32) (33) (34) (35) (36)) ((37) (38) (39) (40) (41) (42)) (1 7) (1 13) (1 19) (1 25) (1 31) (1 37) (7 13) (7 19) (7 25) (7 31) (7 37) (13 19) (13 25) (13 31) (13 37) (19 25) (19 31)(19 37) (25 31) (25 37) (31 37) (2 8) (2 14) (2 20) (2 26) (2 32) (2 38) (8 14) (8 20) (8 26) (8 32) (8 38) (14 20) (14 26) (14 32) (14 38) (20 26) (20 32)(20 38) (26 32) (26 38) (32 38) (3 9)(3 15) (3 21)(3 27) (3 33)(3 39) (9 15) (9 21) (9 27) (9 33) (9 39) (15 21) (15 27) (15 33) (15 39) (21 27) (21 33) (21 39) (27 33) (27 39) (33 39) (4 10) (4 16) (4 22) (4 28) (4 34) (4 40)(10 16) (10 22)(10 28) (10 34)(10 40) (16 22) (16 28) (16 34) (16 40) (22 28) (22 34) (22 40)

(28 34) (28 40) (34 40) (5 11) (5 17) (5 23) (5 29) (5 35) (5 41) (11 17) (11 23) (11 29) (11 35) (11 41) (17 23) (17 29) (17 35) (17 41) (23 29) (23 35) (23 41) (29 35) (29 41) (35 41) (6 12) (6 18) (6 24) (6 30) (6 36) (6 42) (12 18) (12 24) (12 30) (12 36) (12 42) (18 24) (18 30)(18 36)(18 42) (24 30)(24 36) (24 42) (30 36) (30 42) (36 42)).

In general, Pn has $(n^3 + n^2)/2 + n + 1$ members and $n(n + 1)$ atoms.

8. The tests of CSProve in this section were made on an aging computer with Windows Me. After this paper was written CSProve was run on a much more powerful desktop computer, a dual-processor Power MacIntosh G5 with Digitool Corporation's MacIntosh Common Lisp 5.0 (MCL 5.0). On that computer CSProve decided P6 in 7.6 seconds, exactly ten times faster than the Windows computer. MCL 5.0 is an elegant version of LISP for MacIntosh computers, but even with an educational discount it is quite expensive.