To appear, *Proceedings of Sixth International Symposium on Artificial Intelligenc*e, Monterrey, Mexico, September 1993.

Improving the Performance of AI Software: Payoffs and Pitfalls in Using Automatic Memoization

Marty Hall
Artificial Intelligence Laboratory
AAI Corporation
PO Box 126
Hunt Valley, MD 21030
(410) 792-6000 x3440
hall@aplcenmp.apl.jhu.edu

James Mayfield
Computer Science Department
University of Maryland Baltimore County
5401 Wilkens Ave.
Baltimore, MD 21228
(410) 455-3099
mayfield@cs.umbc.edu

ABSTRACT

Many functions perform redundant calculations. Within a single function invocation, several sub-functions may be invoked with exactly the same arguments, or, over the course of time in a system run, a function may be invoked by different users or routines with the same or similar inputs. This observation leads to the obvious conclusion that in some cases it is beneficial to store the previously calculated values and only perform a calculation in situations that have not been seen previously. This technique is called memoization, and the "manual" version of this generally involves building lookup tables. This however, is often a tedious and time-consuming process, and requires significant modification and debugging of existing code. This is frequently inappropriate in the dynamic, rapid-prototyping context of AI software development. An "automatic" memoization facility is one in which existing functions can be programmatically changed to cache previously-seen results in a hash table. These results will then be returned when the functions are invoked with arguments they have seen previously. This can be done without changing the code, thus providing a simple, modular, and transparent way to dramatically accelerate certain types of functions.

This paper presents an overview of automatic memoization and discusses the types of applications that benefit from it, illustrated with experiences on the ARPA Submarine Signature Management System, a large LISP-based decision aiding system.

1. Overview

The following section (2) outlines the concept of automatic memoization, and gives a simplified implementation in Common LISP. Section 3 looks at application areas that benefit from automatic memoization, with timing results from several sample problems. Section 4 looks at potential pitfalls, and Section 5 gives conclusions and areas for future work.

Code for the entire facility (in portable Common LISP) is available for non-commercial purposes via anonymous FTP or electronic mail. Anonymous FTP is available from ftp.cs.umbc.edu (130.85.100.53), in /pub/Memoization. Requests for the sources can also be mailed to the author at hall@aplcenmp.apl.jhu.edu.

2. What is Automatic Memoization?

The term "memoization" was first coined by Donald Michie [5] and refers to the process of tabulating results in order to prevent wasted calculations. *Automatic* memoization refers to a method by which an existing function can be changed into one that memoizes.

For example, consider the simplified version of Memo below, adapted from *Paradigms of AI Programming* [6], which was in turn inspired by [1]. It takes a function as input and returns an "equivalent" function that performs lookup from a hash table. When called, this new function compares the argument to ones that it has recorded previously. If the argument has been seen before, the corresponding value in the hash table is returned. If it has not been seen previously, the original function is called with that argument, the return value is stored in the hash table with the argument as key, and then that value is returned.

As a typographical convention throughout the paper, LISP code will be in constant-width font. System functions and variables will be in lower case, and user-defined ones will have the first character of each word capitalized.

```
(defun Memo (Function)
  (let ((Hash-Table
          (make-hash-table :test #'equal)))
    #'(lambda (&rest Args)
        (multiple-value-bind (Value Found?)
            (gethash Args Hash-Table)
          (if Found?
            Value
            (setf
              (gethash Args Hash-Table)
              (apply Function Args)))))
))
```

For those not familiar with LISP, `#'(lambda ...)` indicates that a lexical closure (function with associated binding environment) is being created. This

This work was sponsored in part by the Advanced Research Projects Agency under JHU/APL subcontract 605089-L.

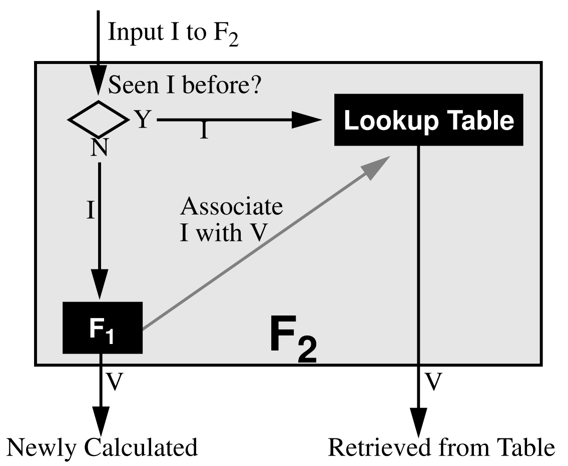
function is returned from Memo. A hash table, created via make-hash-table, is generated each time

Figure 1: Creating a Memoized Function

F1 is input to Memo. F2 is output.

Memo is called, then accessed each time the resultant function is invoked. The first gethash form tests to see if there is an entry in the table corresponding to the argument list Args. That entry, if any, is stored in Value, while Found? is assigned a boolean value indicating whether the lookup was successful. The form (apply Function Args) simply invokes the original function on the argument list, and (setf (gethash ...)) stores the resultant value in the hash table, returning it as a side effect.

Figure 2: Using a Memoized Function



Memo takes a function as input and returns a memoized version. Memoize takes a function *name* as input, memoizes the associated function, then assigns that new function to the associated function name. This supports the memoization of recursive functions, and existing code that referred to the function name will now automatically get the memoized version.

```
(defun Memoize (Function-Name)
  (setf
    (symbol-function Function-Name)
    (Memo (symbol-function
            Function-Name))))
```

Memoize would be used as follows:

```
(defun Expensive (<Args>)    <Long-Body>)
(time (Expensive <Values>)) → [30 seconds]
(Memoize 'Expensive)
(time (Expensive <Values>)) → [≤30 seconds]
(time (Expensive <Values>)) → [0.0001 seconds]
```

The real version of the facility has many more utilities for bookkeeping, memoizing and unmemoizing functions temporarily or permanently, evaluating the benefits of memoization, and saving the hash table to disk for use in a later session. But the core implementation is very similar to the simple version presented here.

3. Applications

There are four basic applications for automatic memoization: Repetitions Within a Function Call, Repetitions Over Time, Off-Line Runs, and Performance Profiling.

3.1 Repetitions Within a Function Call

This case is when a single routine calls some subroutine (or itself recursively) more than is needed, resulting in extra calculations. By means of illustration, consider the following definition of Divided-Difference, which may be used to determine coefficients of interpolated polynomials. The algorithm is a standard one in numerical methods, taken directly from *Numerical Mathematics and Computing* [2]. The application is not particularly important; the point is that the recursive calls form a graph, not a tree, and calculations are repeated. Determining a calling order to avoid these repeated calculations may not appear obvious, and thus it is a ripe candidate for Memoization.

```
(defun Divided-Difference (Function Points)
  "Determines kth coefficient, where
  'Points' contains k entries"
  (if
    (null (rest Points))
    (apply Function Points)
    (/ (- (Divided-Difference
            Function (rest Points))
          (Divided-Difference
            Function (butlast Points)))
       (- (first (last Points))
          (first Points))) ))

(defun Test-Function (N)
  (* pi (cos N)))
```

Figure 3 compares the performance of the memoized and unmemoized versions of Divided-Difference using Test-Function and the first N natural numbers as arguments. Since one function call with N points results in 2 calls with N-1 points, the unmemoized version has $O(2^N)$ time complexity. After memoization, the first invocation takes $O(N^2)$ time, since no subsequence of Points is calculated more than once, and there are $1 + 2 + 3 + \ldots + N = ((N+1)N)/2$ subsequences. Subsequent invocations take near-constant time.

This type of repetition is common and is normally addressed by either determining a better calling sequence or building a special purpose data structure to store intermediate results. For instance, in Volume 2 (Seminumerical Algorithms) of *The Art of Computer Programming* [4], Knuth presents a straightforward method for building up the divided differences in the proper order to get the same performance as the first invocation of the memoized version. Similarly, Peter Norvig shows that the performance of chart parsing or

Figure 3: Average Time in Seconds to Calculate Divided-Difference on *N* points

N	Unmemoized	Memoized (First Run)	Memoized (Subsequent Runs)
15	11	0.18	0.0006
16	22	0.21	0.0006
17	43	0.22	0.0006
18	87	0.28	0.0007
19	173	0.4	0.0007
100	Centuries	25.0	0.002

Earley's algorithm can be obtained for parsing context-free languages by memoizing a simple recursive backtracking parser [7].

Given this, memoization can be viewed as a general and straightforward technique for automatic dynamic programming. Rather than tackling the difficult task of determining the proper order in which to build up subpieces, a simple solution can be memoized to get the same performance[3]. The question, then, becomes which approach is better: memoizing a less efficient algorithm or changing to an implementation that either uses a different algorithm or maintains special-purpose data structures?

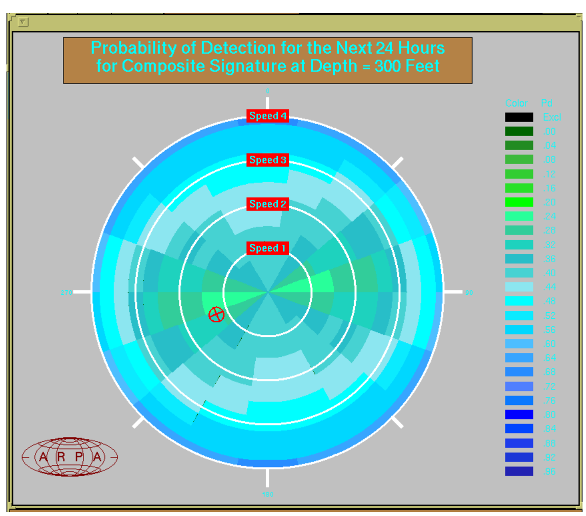
Clearly, automatic memoization is not meant to be a substitute for finding the proper algorithm for the task. However, in cases where the major benefit of the better algorithm is a savings in repeated calculations, memoizing the obvious algorithm has several advantages over an explicit dynamic programming approach. First of all, a different algorithm would not "remember" its results after the top-level function exits, so it would have performance analogous to the third column of Figure 3 rather than the fourth. This is discussed more in the following section (3.2). Secondly, memoization tends to keep the code shorter and clearer, and requires little additional debugging if the straightforward method has already been well tested. On large programs, there is often a reluctance to change routines that have already been tested and verified, especially if that will require changes in multiple places in the code. Furthermore, since it is simple to switch back and forth between the memoized and unmemoized versions, it is easy to compare the performances of the two versions. Finally, and most importantly, there is the practical issue of programmer time and effort. If finding a better algorithm is difficult, programmers will tend not to bother unless there is a very large payoff. So a lot of effort gets placed on a few routines, but others that could benefit from memoization are overlooked altogether. This last point should be stressed, and in fact this was a common occurrence on the Signature Management System (SMS). Places where wasted calculations were suspected or even known to occur were often disregarded, since the effort to quantify the repeats and determine a method to avoid repetition was frequently deemed not worth the effort, given the rapidly changing nature of AI software. However, providing an easy method for memoization made it simple to find the routines that really benefited, and additive speedups from several small routines resulted in greatly increased overall performance.

3.2 Repetitions Over Time

In Section 3.1, there was a central routine which invokes the lower-level functions repeatedly. Changes to the algorithm at this level, or a data structure local to that central routine could gain many of the same efficiency benefits as automatic memoization, albeit with decreased flexibility and increased effort. However in a team programming environment different sections of the system, written by different programmers, may access the same function. Alternatively, in an interactive system the user may invoke calculations at different times that make use of some of the same pieces. In these cases, there is no central routine which could manage the calling sequence, and the only alternative to automatic memoization is to have the routine in question manage its own global data structure to remember previous results. Memoization provides an efficient and convenient alternative.

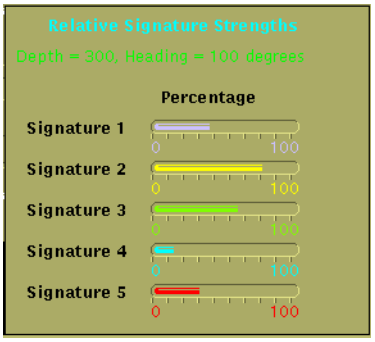
For instance, Figure 4 shows a display used as an aid to planning submarine operations in the SMS system. It shows the predicted probability of detection of the submarine for various choices of heading and speed, drawn on a polar plot with the angle (theta) indicating heading (0 corresponding to due north), and the radius (r) corresponding to speed. Each (r,theta) pair (arc) in the display is coded with a color indicating the cumulative probability of detection for the sub if it were to operate at the course and speed.

Figure 4: SMS Detectability Planning Screen



This display is used as a high-level tool in planning, and thus presents highly summarized information. It presents a single number for probability of detection, a composite of all the potential detection methods (signatures). The user frequently is interested in the contribution of individual signature components to this composite. Since the probability of detection of each component is memoized before it is combined into the composite, any component corresponding to a point on the display can be retrieved almost instantly. Taking advantage of this, the display of Figure 5 can be set up.

Figure 5: Individual Signature Components



Whenever the user moves the mouse over the composite detectability display (Figure 4), the corresponding speed and course for the point under the mouse is calculated. Then, the individual components are calculated, with their relative values shown in the bar charts. Due to the effects of memoization, the component values can be calculated and graphed as quickly as the user can move the mouse.

Accomplishing this was extremely simple, with no special purpose routines or data structures needed. The original code looked something like the following, where PD means “Probability of Detection”:

```
(defun Composite-PD (Course Speed <Args>)
  (Weighted-Average
    (Signature-1-PD Course Speed <Arg-Subset>)
    (Signature-2-PD Course Speed <Arg-Subset>)
    (Signature-3-PD Course Speed <Arg-Subset>)
    (Signature-4-PD Course Speed <Arg-Subset>)
    (Signature-5-PD Course Speed <Arg-Subset>)

(defun Signature-1-PD ...)
(defun Signature-2-PD ...)
(defun Signature-3-PD ...)
(defun Signature-4-PD ...)
(defun Signature-5-PD ...)
```

Now, to make the real-time component display, the routines need to look up the (x,y) position of mouse, convert to (course,speed), then call the individual component functions directly. Since each component has been calculated at each course and speed, all of these calculations will result in simple table lookups after the following simple change:

```
(Def-Memo-Fun Signature-1-PD ...)
(Def-Memo-Fun Signature-2-PD ...)
(Def-Memo-Fun Signature-3-PD ...)
(Def-Memo-Fun Signature-4-PD ...)
(Def-Memo-Fun Signature-5-PD ...)
```

3.3 Off-Line Runs

In the Divided-Difference example and the discussion of Section 3.1 it was seen how the use of memoization could be viewed as an automatic dynamic programming facility, remembering the results of subproblems when building up a larger solution. This can result in the reduction of exponential time algorithms to polynomial or linear time on the first invocation, but without time-consuming rewrites or dynamic programming algorithm design. In Section 3.2, it was seen how memoization could save on repeated invocations of expensive calculations, giving a constant factor (but potentially large) speedup. This still leaves the case where even the very first invocation is too expensive. This is normally addressed by building a special purpose data file, and filling the values with an off-line execution of the expensive routine. Then, the function in question is modified to access that file. The automatic memoization facility provides a method to do the same thing while still maintaining the transparency and ease of use of memoization, and without forcing the programmer to know which ranges of values are stored in the data file, and which must be freshly calculated. The idea is that the function is memoized and then run off-line in the normal manner on the cases of interest. The contents of the hash table are then saved to disk in a file with a name associated with the LISP function name. Then, this file is automatically used to seed the hash table for the function when it is reloaded in a later session. For instance, to use a simplified example from the SMS system, suppose that `Magnetic-Parameter` was a very time consuming calculation that only depended on the latitude and longitude:

```
(defun Magnetic-Parameter (Lat Long) <Body>)
```

Now, you could run the following at night or over the weekend:

```
(defun Fill-Magnetic-Table
        (Lat-Min Lat-Max Lat-Step
         Long-Min Long-Max Long-Step)
  (Memoize 'Magnetic-Parameter)
  (loop for Lat from Lat-Min to Lat-Max
              by Lat-Step do
    (loop for Long from Long-Min to Long-Max
                by Long-Step do
      (Magnetic-Parameter Lat Long)))
  (Save-Memo-Table 'Magnetic-Parameter))
```

Once this completes, then the previous definition of `Magnetic-Parameter` would be changed as below:

```
(Def-Memo-Fun Magnetic-Parameter (Lat Long)
  (:Hash-Table-Source :Disk)
  <Body>)
```

This is where the ease of use of memoization particularly pays off. If this were a permanent situation, it might be feasible to build conventional lookup tables. But for temporary conditions (e.g. running multiple simulations in the same environment), the effort to build the tables would likely not be worth the effort.

3.4 Performance Profiling

Rather than using memoization for its own sake, it can also be used as a tool in conventional optimization. Most LISP implementations provide a profiling facility whereby the user can see the time that a top-level function spends in various lower-level routines. This is important since knowing where a routine spends its time tells you where to spend your optimization efforts. However, these profilers generally take quite a bit of overhead. This is certainly worth the effort for important cases, and is an extremely valuable tool. In smaller cases, however, memoization provides a quick but rough method for determining where to spend the effort of optimization. Simplicity is the key: tools that take a long time to be used will be used only occasionally; tools that are simple for programmers to use will be used more often.

Rather than running the full metering system, users would interactively memoize certain functions using the `Memoized-Time` macro. This temporarily memoizes certain functions, and then executes a body of code without memoization, with memoization and an empty cache, and with memoization and a full cache. If the timing for the second memoized case only improved by, for instance 5%, then, for that test case, no amount of optimization in the routines in question would provide more than a 5% speedup at the higher level.

For example, consider the following case:

```
(defun F1 (A B C D)
  (F2 (F3 A B)
      (F4 B C D)
      (F5 A)))

(Memoized-Time (F2 F3) (F1 <Values>))
First Time:  30.0 seconds
Second Time: 29.5 seconds
Third Time:  29.3 seconds
```

This shows that neither F2 nor F3 contribute significantly to the overall time of F1. Again, it is the interactive nature and transparency of the facility that makes it useful here; if memoization required changes to the source code it would never be used for this application.

3.5 Two Case Studies

3.5.1 Magnetics

Figure 6 gives timing statistics for a magnetics module used in the Signature Management System, timed after various uses of memoization were put into effect. Ignoring the benefits when the user asks for the exact same displays at different times (which is in fact quite common), the following is a summary of the time benefits of memoization on the first time invocation of the top-level display, as shown in Figure 4. Times are in seconds, and are conservative approximations. Similar results were obtained with other modules.

Figure 6: TIming of Magnetics Module

Use of Memoization	Time (Seconds)	Relative Speedup (Cumulative)
Original	48	1.0
Conventional Optimization	36	1.33
Repetitions Over Time	24	2.0
Dynamic Programming	2	24.0
Saved Lookup Tables	0.001	48,000.

3.5.2 Detectability Planning Display

Given the diverse uses of memoization by various programmers on the SMS program, an attempt was made to estimate the overall contribution of memoization to the system. To do this, the default display (as shown in Figure 4) was run both in the default mode and then with *all* memoization turned off:

```
(time (Make-PD-Planning Display))
Elapsed Time: 4.06 seconds
Ephemeral Bytes Consed: 615,784

(time (Make-PD-Planning Display))
Elapsed Time: 2562.74 seconds (42 min 42 sec)
Ephemeral Bytes Consed: 2,969,392,724
```

This showed a 631x improvement in speed, and a 4,822x improvement in the amount of temporary memory (garbage) allocated. Now, benchmarks are notoriously misleading, and in many places the code would have been written dramatically differently if memoization had not been available. Nevertheless, the results are illuminating.

4. Pitfalls

One of the chief benefits of memoization is its transparency. Since requirements and code tends to change rapidly in AI programs, the user needs to be able to easily switch back and forth between memoized and unmemoized versions, and without making changes in the routines that make use of the function that is to be memoized. However, an overly transparent view can also lead to difficulty, as described in the following sections.

4.1 Non-Functions

Memoization only works for true functions, not procedures. That is, if a function's result is not completely and deterministically specified by its input parameters, using memoization will give incorrect results, since it uses the parameter list to retrieve previous values.

Before memoizing a given routine, the programmer needs to verify that there is no internal dependency on side effects. This is not always simple; despite attempts to encourage a functional programming style, programmers will occasionally discover that some routine their function depended upon had some deeply buried dependence on a global variable or the slot value of a CLOS instance.

4.2 Destructive Operations

In LISP, parameters to functions are normally pointers to the existing objects, not new copies. Most list functions, however, take care not to modify the supplied lists, but make copies of certain sections if needed. Limiting most of their code to these types of operations allows the LISP programmer to have apparent call-by-value semantics without the overhead of copying lists every time they are passed to a function.

Destructive operations, however, actually modify the supplied argument. A common rule of thumb for LISP programmers is to only use destructive operations on lists that are freshly consed (generated), to be sure that the changes do not inadvertently propagate to unexpected places. If, however, the routine that creates the list becomes memoized, then the list is no longer newly created each time, and the destructive operation will change the value that the memoized function will return.

For instance, suppose that function `Raw-Data` returns a newly consed list of numbers, and is only called by the function `Normalized-Data`, which uses `delete` to destructively remove the maximum and minimum entries from the list before returning it. Prior to memoization, this might be perfectly safe. After memoizing `Raw-Data`, however, each subsequent retrieval of supposedly identical data values might in fact get a shorter list.

Before memoizing a routine that returns a list, the programmer needs to check the callers of that routine to verify no destructive operations are performed on it.

4.3 Too-Strict Matches

Memoization is performed by doing an exact match on the argument list, using the LISP function `equal` by default. If you write `(defun Foo (&key (Bar 2) (Baz 3) ...)`, and then memoize `Foo`, all of the following will be treated as distinct, even though the parameters have identical values in all cases:

```
(Foo)
(Foo :Bar 2)
(Foo :Bar 2 :Baz 3)
(Foo :Baz 3)
(Foo :Baz 3 :Bar 2)
```

Similarly, one can have counterintuitive results when the arguments are floating point numbers, forgetting that, for instance, 2 is not `equal` to 2.0, and 1.234567 is not `equal` to 1.23456, even though the function may treat them as identical.

The solution adopted by the SMS program is to introduce “wrapper” functions that take keyword arguments, floating point numbers, etc., canonicalize the arguments into some common form, then pass them on to an internal function that takes only required arguments in the standard format. It is this internal function that would be memoized.

4.4 Non-Printable Values

The routine that saves data to disk, described in Section 3.3, does so by printing the representation of the object using `format`, then directing that output stream to a file. This means that LISP objects whose print representation cannot be parsed by `read` cannot be saved to disk. Some objects such as CLOS instances and structures allow the definition of a custom print function, and this can sometimes be used to save them to disk. But this is not a general mechanism, and special purpose code will need to be written in those cases.

4.5 Compiler Optimization of Recursive Calls

Some LISP implementations will remove all self-recursive calls when compilation is done at the highest speed optimization. That is, recursive calls will jump directly to the code, rather than going through the symbol-function slot. This means that memoization will only be of benefit for repeated calls to the top-level form, and speedups of the type described in section 3.1 will be eliminated. However, compilers that have this optimization also provide flags that can be set to prevent its use.

4.6 Memory Usage

In most cases, memoization is a time vs. memory trade-off. In some cases, where a frequently repeated function generates large structures or a lot of garbage, memoization may actually save memory by simply reusing the previously created structures. This was the case in the example of section 3.5.2 . However, in most cases you sacrifice space in order to gain speed. These trade-offs should be evaluated carefully to avoid consuming large amounts of memory with little speedup. Several automated tools can assist in this process. `Memoized-Function-Call-Count` reports on how many times a memoized function was called, broken down by whether those calls resulted in new calculations or previous results being returned from the hash table. `With-Memoization` is a macro that allows the user to execute forms in a context where certain functions are temporarily memoized. `Memoized-Time` runs a body of code three times: unmemoized, memoized but with an empty cache, and memoized with the values of the previous run, returning time and space information for each run.

5. Conclusions and Future Work

In summary, an automatic memoization facility provides the tools to easily convert functions to variations that “remember” previous arguments and associated results. This can be used to get behavior similar to dynamic programming, where a solution is built up from subproblems, but without the necessity for determining the proper order for calculation. The facility can also be used for calculations that are repeated in time throughout a system run, and to save expensive calculations during an off-line session. It is also a useful aid for timing and profiling code, even in applications where only conventional optimization is being performed. The simplicity and transparency of use allows the facility to be quickly adopted by programmers, and helps minimize the debugging and verification time.

Tools for determining when cached values are out of date need to be developed. This is particularly true when storing the contents of the cache to disk for use in a later session. Also, there is a lot of area to explore regarding inexact matches for memoization. An organized framework for "fuzzy memoization" might provide utility for applications in planning and/or learning. Finally, methods for limiting the size of the memo tables (either by total entries or time of last access) should be developed.

6. Acknowledgments

People providing valuable feedback on the code and early drafts of the paper were V.J. Benokraitis (AAI Corporation), Lien T. Duong (AAI Corporation), Tim Finin (UMBC), J. Paul McNamee (AAI Corporation), Peter Norvig (Sun Microsystems), and David J. Scheerer (The Johns Hopkins Applied Physics Laboratory). John Aspinall (Symbolics) suggested the use of the Divided Difference example.

7. References

[1] Abelson, Harold, Sussman, Gerald Jay, and Sussman, Julie, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[2] Cheney, Ward, and Kincaid, David, *Numerical Mathematics and Computing*, Brooks/Cole, 1980.

[3] Cormen, Thomas, Leiserson, Charles, and Rivest, Ronald, *Introduction to Algorithms*, McGraw Hill and MIT Press, 1991.

[4][Knuth, Donald E., *The Art of Computer Programming*, Addison-Wesley, 1969.

[5] Michie, Donald, "Memo Functions and Machine Learning," *Nature*, **218** No. 1, April 1968, pp. 19-22.

[6] Norvig, Peter, *Paradigms of AI Programming: Case Studies in Common LISP*, Morgan Kaufmann, 1992.

[7] Norvig, Peter, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, 1991.

[8] [Warren, 1992] Warrren, David S., "Memoing for Logic Programs," *Communications of the ACM*, **35** No 3, March 1992, pp. 93-111.