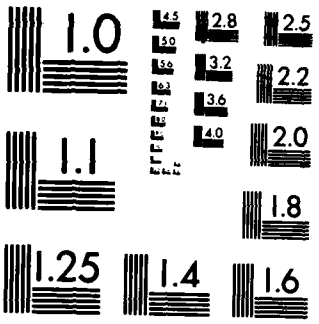
AD-A147 568 EFFICIENT COMPUTATION OF OPTIMAL TIME FOR TRANSACTION PROCESSING IN A DATABASE SYSTEM(U) ELECTRONIC SYSTEMS DIV HANSCOM AFB MA D A VARVEL ET AL. 31 AUG 84 1/1

UNCLASSIFIED ESD-TR-84-193 F/G 12/1 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

ESD-TR-84-193

(12)

Efficient Computation of Optimal Time for Transaction Processing in a Database System

DONALD A. VARVEL
WILLIAM PERRIZO

31 August 1984

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

Prepared for

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR DEVELOPMENT PLANS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

THOMAS SELINKA, Capt, USAF
Project Officer

SILVIO V. D'ARCO, Lt Col, USAF
Deputy Director, Tactical C³I Systems Planning
Deputy for Development Plans

FOR THE COMMANDER

DONALD L. MILLER, Colonel, USAF
Assistant Deputy for Development Plans

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AD-A147568

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): ESD-TR-84-193	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Hq Electronic Systems Division, Deputy for Development Plans; 6b. OFFICE SYMBOL (If applicable): ESD/XR	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Hanscom AFB, Bedford, MA 01731	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO.; PROJECT NO.; TASK NO.; WORK UNIT NO.
11. TITLE (Include Security Classification): Efficient Computation of Optimal Time (Cont.)	
12. PERSONAL AUTHOR(S): Donald A. Varvel and William Perrizo	
13a. TYPE OF REPORT: Final; 13b. TIME COVERED: FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day): 1984 August 31; 15. PAGE COUNT: 28
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Backtracking, Computational complexity, Concurrency control, Database Management Systems DBMS, Decision trees, Depth-first search, Locking, Nondeterministic (Cont.)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Most database systems use locking for concurrency control. Responsiveness is degraded when transactions spend much time waiting for locks. In those situations in which the lockable units need not be processed in a particular order, differences in the order of processing can make large differences in the durations of the transactions, i.e., responsiveness. A "backtracking" or "tree searching" subprogram is used to determine the optimal order of processing. The subprogram uses an interesting method of tree pruning in order to perform the computation in reasonable time.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Capt Thomas Sollinka	22b. TELEPHONE NUMBER (Include Area Code): (617) 271-8400; 22c. OFFICE SYMBOL: ESD/XRTF

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

Block 11 Continued

for Transaction Processing in a Database System. (Unclassified)

Block 18 Continued

algorithms,Pascal,Performance,Pruning,Recursion,Tactical Air Control System, Transactions, Tree search

TABLE OF CONTENTS

TABLE OF CONTENTS

1 Introduction

A search of a decision tree can be used to solve complicated problems involving multiple decisions. It is the only known effective approach to computer chess, for example. The method typically operates on the brink of combinatorial explosion, however. In its basic form its computation time increases exponentially or factorially, depending on the number of choices at each level of the tree. For that reason, considerable effort is often spent in reducing the size of the tree.

Donald Knuth has pointed out in [KNUT75] that the efficiency of backtracking algorithms, to which decision-tree search belongs, is very sensitive to small modifications. We have devised an efficient backtracking method for one application, and feel it should be recorded.

We encountered a tree search problem while investigating a database performance question.

Database management systems (DBMS) control and facilitate access to a collection of data that is integrated and shared [DATE81]. Concurrency is the simultaneous or interleaved execution of more than one process. A transaction is the logical unit of execution in a database system; examples would include displaying the balance in one account, posting interest to all accounts, or finding all those items from a particular supplier where the inventory has fallen below the reorder point. Concurrency control is the handling of concurrent transactions so as to produce the same results as nonconcurrent execution [DATE83, BERN81].

Sequential machines perform one computation at a time. Most existing machines are sequential. The second and third transactions described above, however, involve performing the same operation on many data items. Making such transactions into sequences involves choosing an order. If some of the individual data items are unavailable at certain times because of being used by other transactions, the choice of order is important to performance.

1.1 Optimality

We wish to determine the time required for the processing of a transaction provided the optimal order of access is used. No actual transaction manager can have sufficient information to achieve optimality in all cases, but optimality provides a useful standard of comparison.

However, even given knowledge of when each data item will be locked, computation of optimal time is not trivial.

2. The algorithm

We decided to develop an algorithm that would always produce the correct result and would nearly always compute it in acceptable time.

2.1 Decision tree search

One way of finding an optimal solution to a complicated problem is to create a decision tree and search it exhaustively.

2.1.1 Creating a decision tree

A tree is a directed acyclic graph in which one node (the root) has no parent and each other node has exactly one parent. Intuitively, it is a diagram of a branching-out from a starting point. A decision tree represents a series of decisions, with branches representing particular choices. Given three data items, A, B, and C, Figure 1 shows a decision tree for order of processing. The leftmost branch represents A, B, C.

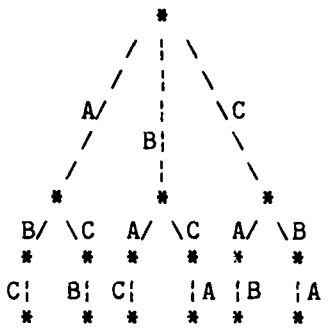


Figure 1

2.1.2 Exhaustive search

Assume each terminal node has a value. In the example, the value would be time required to process in that order. To find the minimum of those, one could use the recursive function Search:

```
Function Search(Tree : Tree_type) : Integer;
Var Val, Min : Integer; Pos : Tree_type;
Begin
   If this is a terminal node then
      Search := Value of this node
   Else begin
      Min := Maxint;
         While Possibilities remain untried do begin
            Select an untried choice, called Pos;
            Val := Search(Pos);
            If Val < Min then Min := Val
         End;
         Search := Min
   End
End;
```

Algorithm 1

This algorithm always produces correct results, but operates in O(N!) time, where N is the number of data items. This is not acceptable performance.

2.1.3 Tree pruning

The performance of tree-searching algorithms may be improved by recognizing branches that do not contribute to the result and not searching those branches. This is called pruning. In the most general case, where only terminal nodes have values and no lower-bound calculations may be made, pruning is not possible. As more knowledge of the actual situation becomes available, however, some pruning becomes possible.

We have found ways to prune away nearly the entire tree.

2.1.4 Lower bound

A valid lower bound on the value of a tree (optimal time, in this case) can greatly reduce the time needed to search the tree. As soon as a way

is discovered to achieve the lower bound, the search may be abandoned since no better result can be achieved. Also, if a path has already been found with a total cost as low as can possibly be achieved in this branch, the search of this branch may be abandoned. Since these considerations apply at any level of the search, we may express them as another recursive function.

```
Var
   Path_cost : Integer;     { Initialize to 0 }
   Cutoff : Integer;        { Initialize to Maxint }
Function Search(Tree : Tree_type) : Integer;
Var
   Unit_cost, Best_Path, Lowerbound, Pathtime : Integer;
   Which : Tree_type;
Begin
   If this is a terminal node then begin
      Unit_cost := cost associated with this node;
      If Path_cost+Unit_cost < Cutoff then
         Cutoff := Path_cost + Unit_cost;
      Search := Unit_cost
   End
   Else begin                { Choices remain }
      Best_path := Maxint;
      Compute Lowerbound;
      If Path_cost + Lowerbound < Cutoff then begin  { Cutoff }
         Arrange the possible choices in a good order;
         While choices remain and (Best_Path>Lowerbound) do begin
            Select next possibility, call it Which;
            Unit_cost := cost associated with Which;
            Path_cost := Path_cost + Unit_cost;
            Pathtime := Search(Which);
            If Pathtime < Best_Path - Unit_cost then
               Best_Path := Pathtime + Unit_cost;
            Path_cost := Path_cost - Unit_cost
         End
      End;
      Search := Best_Path
   End
End;
```

Algorithm 2

2.2 Optimization under Algorithm 2

The performance of Algorithm 2 depends on two factors:

1. The ordering of the choices

2. The accuracy of Lowerbound

2.2.1 Selection order

If only one branch of the tree produces optimal time, we would like to examine that branch early. If we always search the wrong branch, we still have to examine the entire tree! A good a priori ordering is possible in our application. Of course, if we could guarantee selecting the best order beforehand we would not need to search the tree, but we have discovered no method to achieve that.

2.2.2 Lowerbound

A valid lower bound for a tree search will never exceed the actual value of the tree. Given an invalid quantity as Lowerbound, Algorithm 2 will return that instead of the correct answer. On the other hand, a valid lower bound which can never be achieved will result in no pruning. The greatest lower bound of the value of the tree is in fact the value of the tree, and this is the only value that results in any pruning. Our computation can be very fast, provided we already know the answer.

This is much more helpful than it sounds, though, because it holds at all levels of the tree. Even if we can do no pruning at the top level, we may prune very severely at the next.

A perfect algorithm for determining Lowerbound would eliminate the need for a tree search.

3 Our application

We wished to apply tree search with pruning to our database application. To explain, we must first tell something about the application itself.

3.1 The simulation

We have simulated algorithms for database lock managers and the routines that use them. The various routines simulate processing simulated transactions. The important features of a transaction in the simulation have to do with the various data items accessed; we call these Lockable Units, or LU's.

A transaction has some number of LU's. This number is specified as a parameter. Each LU has associated with it a processing delay, which represents the time required to process the data in that unit. Each LU may also have some adverse activity. That represents other transactions accessing the unit in a way incompatible with our transaction's planned access. These quantities are generated at random. Some LU's are represented as having fixed-length queues; these are very high-activity data items. There is a minimum time required to acquire any data item, which we have set at one time unit. Table 1 represents an example of a transaction.

Transaction (* = steady-state queue of given length)				
Unit	Delay	Activity	Activity	Activity
1	10	* 86		
2	4	90 - 247	250 - 295	330 - 445
3	8	* 96		
4	4	220 - 282	310 - 393	520 - 579
5	12	150 - 276	570 - 630	

Table 1

4 The method

To find optimal time, we first assume that all queues are entered at transaction initiation, thus changing all asterisks in Table 1 to zeroes, and then perform a tree search.

4.1 Ordering of alternatives

At a given moment, each lockable unit is in one of three states:

1. Available now, but will become unavailable.
2. Available now and at all future times.
3. Not now available.

We select first those in state 1, in ascending order of when they will become unavailable; then state 2, in ascending order of processing delay; and then state 3, in order of when they will become available. That is an optimal order a high percentage of the time. The example transaction would be ordered initially 2-5-4-1-3.

4.2 Computation of Lowerbound

Optimal performance for the example transaction is 105 time units. LU's 2, 5, and 4 are acquired and processed in 23 time units. LU1 is acquired at time 86, and processing is completed at 96. LU3 is then immediately available, so acquiring it takes one time unit; processing

it takes 8, for a total of 105.

We compute Lowerbound as follows:

```
Create a record for each remaining LU, consisting of a field
representing the time necessary to acquire a lock on that item
(LU.When_avail) and a field for its processing delay (LU.Delay).
Sort on When_avail, descending.
Cum_Delay := 0;
Lowerbound := 0;
While records remain do begin
   Get an LU record;
   Cum_Delay := Cum_Delay + LU.Delay;
   If LU.When_avail + Cum_Delay > Lowerbound then
      Lowerbound := LU.When_avail + Cum_Delay;
   Cum_Delay := Cum_Delay + Min_lock_acquisition_time
End
```

Algorithm 3

Testing this with the example transaction shows why the last line in the **While** is necessary.

Unit	Delay	WhenAvail	CumDelay_1	CumDelay_2	Col.2+Col.3	Lowerbnd
3	8	96	8	9	104	104
1	10	86	19	20	105	105
2	4	1	24	25	25	105
4	4	1	29	30	30	105
5	12	1	42	43	43	105

Table 2

B I B L I O G R A P H Y

[BERN81]: Bernstein, P.A., and Goodman, N. "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13,2 (June, 1981), 185-221.

[DATE81]: Date, C.J. An Introduction to Database Systems, vol. I, 3rd Ed., Addison-Wesley, 1981.

[DATE83]: Date, C.J. An Introduction to Database Systems, vol. II, Addison-Wesley, 1983.

[KNUT75]: Knuth, Donald E. "Estimating the Efficiency of Backtrack Programs," Mathematics of Computation 29,129 (January, 1975), 121-136.

A P P E N D I X A

Optimal Simulator Using Tree Search With Cutoffs

```
{$INCLUDE:'B:LOCGLBLS.DOC'}
{$INCLUDE:'B:LOCTMCAL.DOC'}

      {************************************************}
      {* Module containing code to simulate OPTIMAL. *}
      {************************************************}
Module Locopt;
Uses Globids, Tmcal;

Function Max(A, B : Integer) : Integer; Extern;

Procedure Optimal(LU_Num : Integer);
      {*****************************************************************}
      {* OPTIMAL determines a lower bound on processing the given   *}
      {* transaction using locks.  By assumption it uses only as    *}
      {* many lock requests as there are lockable units and gets    *}
      {* into all queues at initiation time.  It does a search of   *}
      {* the decision tree of orders of lock requests to find one   *}
      {* that results in the least delay.  The tree search selects  *}
      {* a first order of requests that is likely to be good, and   *}
      {* performs forward pruning according to two criteria; its    *}
      {* worst-case performance is O(N!), but is usually O(N).      *}
      {*                                                            *}
      {* OPTIMAL contains the recursive tree search Findbest, which *}
      {* in turn contains Sort.                                     *}
      {*****************************************************************}
Type
   Low_Rec = Record
      When : Integer;
      Proc : 3..15
   End;

Var
   Cutoff : Integer;              { Best time so far.  If it can't }
                                  { be undercut, prune.            }
   I : Integer;
   Remaining : Unit_Range;        { Units remaining to be processed }
   Done : Boolarray;
   Low_Vec : Array[0..Max_Units] of Low_Rec;
                                  { Used in computing Lowerbound }

Function Findbest : Integer;
      {*****************************************************}
      {*  Recursive decision tree search, with cutoffs. *}
      {*****************************************************}
Type
```

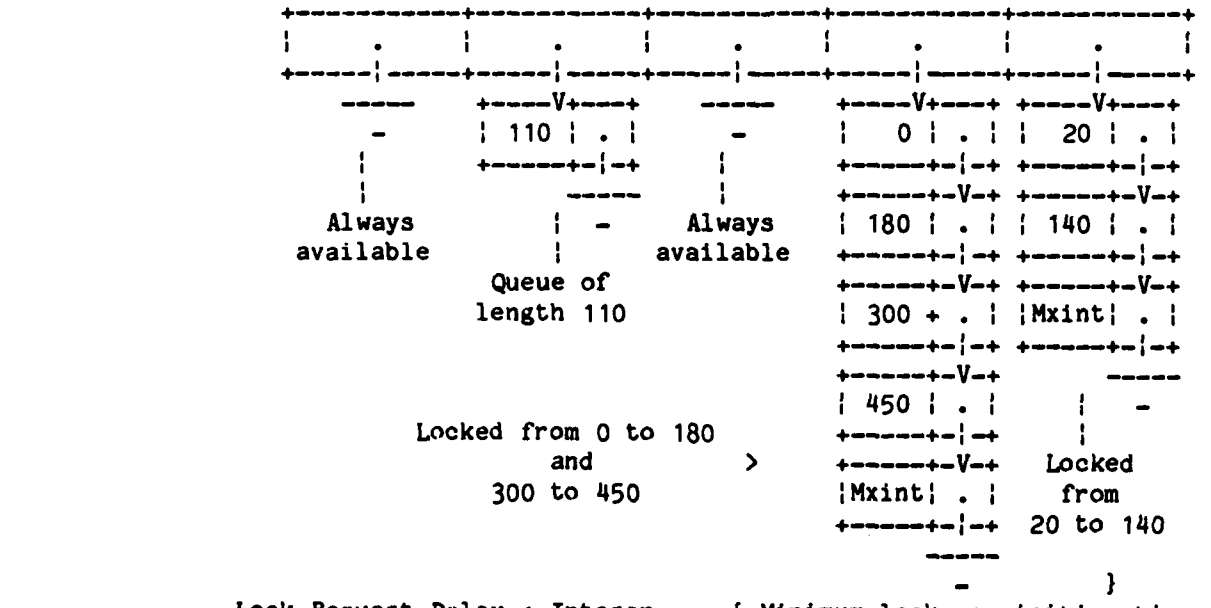
```
   ND_Rec = Record                { Units not yet done, weights }
      U : Unit_Range;
      Val : Integer
   End;
   ND_Vec = Array[Unit_Range] of ND_Rec;

Var
   Getlock, Best_Path, Lowerbound, Pathtime : Integer;
   Cursor : 0..Max_Units;
   I : Unit_Range;
   Notdone : ND_Vec;
   P : T_L_Ptr;
   J, Cum_Delay : Integer;

Procedure Sort(Var Tosort : ND_Vec; N : Unit_Range);
                {*******************************************************}
                {* Linear insertion sort, in place.  An O(N**2) sort *}
                {* makes sense here, since it will be called far     *}
                {* more times with small N than with large.  This    *}
                {* sort beats Shellsort and Quicksort for N less     *}
                {* than about 15, and N will seldom be that large.   *}
                {*******************************************************}
Var
   I, J, TempVal : Integer;
   TempU : Unit_Range;
Begin
   For I := 1 to N-1 do begin     { Elements 1..I are in order }
      TempU := Tosort[I+1].U;
      TempVal := Tosort[I+1].Val;
      J := I;
      While TempVal < Tosort[J].Val do begin
         Tosort[J+1].U := Tosort[J].U;
         Tosort[J+1].Val := Tosort[J].Val;
         J := J - 1;
         If J < 1 then Break      { Nonstandard: Leave innermost loop }
      End;      { While }
      Tosort[J+1].U := TempU;
      Tosort[J+1].Val := TempVal
   End       { For I ... }
End;

{************************************************}
Begin     { Findbest }
   If Remaining = 1 then begin

                { Only one unit remains }
      I := 1; While Done[I] do I := I + 1;
      Getlock := LM0(I) + Delay[I]; { LM0 = time to acquire lock }
      If Present_time + Getlock < Cutoff then
         Cutoff := Present_time + Getlock;
      Findbest := Getlock
   End       { Else if Remaining = 1 ... }
   Else begin
```

```
        { More than one unit remains }
Best_Path := Maxint;
Lowerbound := 0;
Cursor := 0;

        { Compute Lowerbound }
For I := 1 to LU_Num do
   If Not Done[I] then begin
        { Linear insertion according to when available }
      Cum_Delay := LM0(I);
      Low_Vec[0].When := Cum_Delay;
      Low_Vec[0].Proc := Delay[I];
      J := Cursor;
      While Low_Vec[J].When < Cum_Delay do begin
         Low_Vec[J+1] := Low_Vec[J];
         J := J - 1
      End;     { While }
      Cursor := Cursor + 1;
      Low_Vec[J+1] := Low_Vec[0]
   End;  { End linear insertion }
Cum_Delay := 0;
For I := 1 to Cursor do begin
   Cum_Delay := Cum_Delay + Low_Vec[I].Proc;
   Lowerbound := Max(Lowerbound, Low_Vec[I].When + Cum_Delay);
   Cum_Delay := Cum_Delay + Lock_Request_Delay
End;
        { End computation of Lowerbound }

If Present_time + Lowerbound < Cutoff then begin   { A pruning }
        { Arrange those units not processed }
        { Generate weights: Time of next locking for units     }
        {    that are unlocked but which will be locked again  }
        {    (Note crystal ball), Processing-time + 9000       }
        {    for those that are available and will not become  }
        {    unavailable, and Release-time + 10000 for those   }
        {    that are presently locked.                        }
   Cursor := 0;
   For I := 1 to LU_Num do
      If Not Done[I] then begin
         Cursor := Cursor + 1;
         Notdone[Cursor].U := I;
         If Units[I] = Nil then
            Notdone[Cursor].Val := Delay[I] + 9000
         Else if (Units[I]^.Next = Nil)
               and (Units[I]^.Time <= Present_time)
            then Notdone[Cursor].Val := Delay[I] + 9000
         Else if Units[I]^.Next = Nil then
            Notdone[Cursor].Val := Units[I]^.Time + 10000
         Else begin
            P := Units[I];
            While (P^.Next^.Time <= Present_time)
                  and (P^.Next^.Next^.Next <> Nil)
               do P := P^.Next^.Next;
                              { PT () }
```

```
                If P^.Time > Present_time then
                   Notdone[Cursor].Val := P^.Time
                                  {  ( PT )  }
                Else if P^.Next^.Time > Present_time then
                   Notdone[Cursor].Val := P^.Next^.Time + 10000
                                  {  ()  PT  }
                Else Notdone[Cursor].Val := Delay[I] + 9000
             End      { Else }
          End;     { Then }

             { Sort according to weights }
       Sort(Notdone, Cursor);

             { Search for optimal order, cutting off if equal to }
             { a previously-computed lower bound or if unable to }
             { better the best previous time.                    }
       I := 1;
       While (I <= Cursor) and (Best_Path > Lowerbound) do begin
          Getlock := LM0(Notdone[I].U) + Delay[Notdone[I].U];
                   { Simulate processing the unit }
          Done[Notdone[I].U] := True;
          Present_time := Present_time + Getlock;
          Remaining := Remaining - 1;
                   { Recurse }
          Pathtime := Findbest;
                   { Record best found so far }
          If Pathtime < Best_Path - Getlock then
             Best_Path := Pathtime + Getlock;
                   { Undo }
          Done[Notdone[I].U] := False;
          Present_time := Present_time - Getlock;
          Remaining := Remaining + 1;
             { Increment loop control }
          I := I + 1
       End      { While }
    End;     { Then }
    Findbest := Best_Path
 End      { Else }
End;     { Findbest }

{*********************************************************}
Begin    { Optimal }
             { Initializations }
 Remaining := LU_Num;
 Cutoff := Maxint;
 Present_Time := 0;
 For I := 1 to LU_Num do begin
    Done[I] := False;
    Avail[I] := Maxint;
             { Start all queues }
    If Units[I] <> Nil then if Units[I]^.Next = Nil then
       Avail[I] := Units[I]^.Time
 End;
```

```
                { Call recursive tree search }
    I := Findbest;

    Opteval(FLOAT(Cutoff), FLOAT(LU_Num)); { Record, for comparison }
    Accumulate(Cutoff, LU_Num, 0);
    Summary_Stats(FLOAT(Cutoff), FLOAT(LU_Num))
End;        { Optimal }


End.        { Module Locopt }
```

```
   {*************************************************************}
   {*  Interface that supplies global identifiers to all those *}
   {*  program units that need them.  Important consts and     *}
   {*  types are included, but also the files, lockable unit   *}
   {*  information Units, availability, processing delay, and  *}
   {*  the scalars Lock_Request_Delay (presently 1), number of *}
   {*  lockable units LU_No, and Present_time.                 *}
   {*************************************************************}
Interface;
Unit Globids(Max_Units, Algos, Unit_Range, T_L_Ptr, Time_list,
   Un_Vec, Boolarray, Intarray, Detailfile, Summaryfile, Units,
   Lock_Request_delay, Present_time, Avail, Delay, LU_No);
Const
   Max_Units = 100;                  { Max lockable units (arbitrary)}
   Algos = 8;                        { Number of algorithms          }
Type
   Unit_Range = 1..Max_Units;
   T_L_Ptr = ^Time_list;             { Time node for linked list }
   Time_list = Record
      Time : Integer;
      Next : T_L_Ptr
   End;
   Un_Vec = Array[Unit_Range] of T_L_Ptr;
                                     { Array of time lists }
   Boolarray = Array[Unit_Range] of Boolean;
                                     { Type for Done, used in subprograms
}
   Intarray = Array[Unit_Range] of Integer;
                                     { Used for Avail and Delay }
Var
   DetailFile, SummaryFile : Text;  { Output files }
   Units : Un_Vec;                   { Availability of lockable units:
```




```
   Lock_Request_Delay : Integer;     { Minimum lock acquisition time
(value:1) }
```

```
    Present_time : Integer;          { Simulated clock }
    Avail, Delay : Intarray;         { When available, Processing delay }
    LU_No : Integer;

Begin
End;
```

DISTRIBUTION LIST

HQ USAF/XOORC
Washington, D.C. 20330 (1)

Air Force Systems Command
Andrews AFB, MD 20332
AFSC/XR (1)
AFSC/XRK (2)

Electronic Systems Division
Hanscom AFB, MA 01731
AFGL/SULR (3)
AFGL/SULL (1)
ESD/CC (1)
ESD/DC (1)
ESD/IN (1)
ESD/OC (1)
ESD/TC (2)
ESD/TO (1)
ESD/YW (1)
ESD/XR (2)
ESD/XRC (2)
ESD/XRX (2)
ESD/XRT (10)
ESD/XRW (1)

ESD DET 9
APO NY 09021 (2)

Rome Air Development Center
Griffiss AFB, NY 13441
RADC/CC (1)
RADC/CA (2)
RADC/DC (2)
RADC/CO (2)
RADC/OC (2)
RADC/XP (2)

HQ ESC/XOX
Kelly AFB
San Antonio, TX 78243 (2)

Tactical Air Command
Langely AFB, VA 23665
TAFIG/II (2)
TAC/DRC (2)

Tactical Air Command Systems Office
Hanscom AFB, MA 01731
TACSO-E (2)

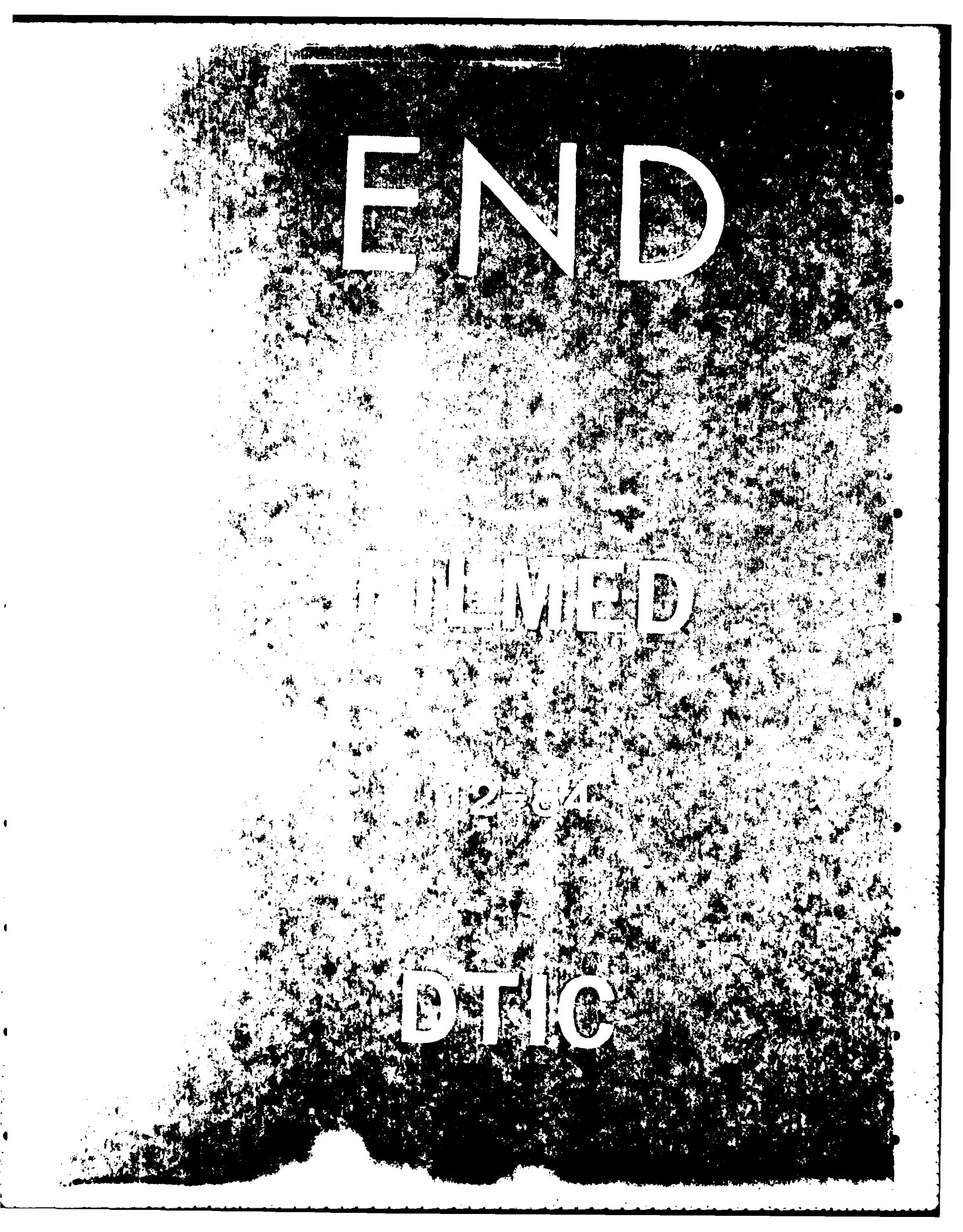
The MITRE Corporation
Bedford Operations
P.O. Box 208
Bedford, MA 01730
ATTN: Mr Norm Briggs (2)

DARPA/ITPO
1400 Wilson Blvd
Arlington, VA 22209 (1)

USA/CECOM
Ft. Monmouth, NJ 07703 (1)

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 (12)

AU Library
Maxwell AFB, AL 36112 (1)


END
LIMED
DTIC