Performance of Arc Consistency Algorithms on the CRAY[1]

Ashok Samal and Tom Henderson

UUCS-87-017

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

July 23, 1987

Abstract

The consistent labeling problem arises in high level computer vision when assigning semantic meaning to the regions of an image. One of the drawbacks of this method is that it is rather slow. By using the consistency tests, node, arc and path consistency [9], the search space is drastically reduced. However, for large problems it takes a fair amount of time.

To use these algorithms more efficiently, one can take two approaches. First, is to design special purpose hardware to specifically run these algorithms. Second is to use faster computers. Here again, one can either take advantage of the multiprocessors, which are becoming very widely available, or use supercomputers like the CRAY, CDC, etc. Here, we present results of the performance of these algorithms in the CRAY supercomputer.

[1] This work was supported in part by NSF Grant DMC 85

Contents

1 Introduction **1**

2 Consistent Labeling Problem **1**

2.1 Solutions to CLP 2

3 The CRAY Supercomputer **3**

3.1 Programming Environment 4

4 Arc Consistency Algorithms **5**

4.1 REVISE Procedure 5

4.2 AC-1 6

4.3 AC-2 7

4.4 AC-3 8

4.5 AC-4 8

4.6 Boolean Formulation 10

5 Implementation **11**

5.1 Test Problems 12

6 Results **13**

7 Conclusion and Future Research **26**

1 Introduction

The consistent labeling problem arises in high level computer vision when assigning semantic meaning to the regions of an image [6]. One of the drawbacks of this method is that it is rather slow. In fact, this problem has been proved to be NP-complete [8]. Mackworth [9] took a different approach to the problem. Instead of finding a solution which is consistent, he devised a set of algorithms which remove the inconsistencies in the network. These algorithms, node consistency, arc consistency and path consistency algorithms, are polynomial time algorithms, but they don't find the solutions. One still has to go through backtracking to find the solution(s), but by using these algorithms first, the search space is drastically reduced, if the problem is well constrained. However, for large problems it takes a fair amount of time. Mohr and Henderson [10] gave an optimal algorithm for arc consistency and a better algorithm for path consistency.

Even the new algorithm is not fast enough. Since it is an optimal algorithm, there is not much hope for drastic improvement. To use these algorithms more efficiently, one can take two approaches. First, is to design special purpose hardware to specifically run these algorithms. Second is to use faster computers. Here again, one can either take advantage of multiprocessors, which are becoming very widely available, or use supercomputers like the CRAY, CDC, etc. Here, we present result of the performance of these algorithms in the CRAY supercomputer.

Although path consistency algorithms help in pruning the search space, it has been our experience that they don't help much after the arc consistency algorithms have been applied first. Also, path consistency algorithms are much more expensive. Due to these two reasons they are not very widely used. We also restrict ourselves to the arc consistency algorithms.

The rest of the report is organized as follows. In section 2 we briefly describe the consistent labeling problem. Section 3 some of the features of the Cray supercomputer and their effect on the performance of the algorithms are discussed. In section 4 the four arc consistency algorithms are described. In section 5 some of the implementation issues are considered and the test problems which are used for comparing the performance of the algorithms are discussed. The results are presented and analyzed in section 6. A brief summary and some directions for future research is given in section 7.

2 Consistent Labeling Problem

Although it has variously been called the satisfycing assignment problem [1], the constraint satisfaction problem [9], and Waltz filtering [13], the consistent labeling problem can be formulated as follows, given

- a set of items or units, $\mathbf{U} = \{u_1, u_2, \ldots, u_n\}$.
- that each unit u_i has a domain D_i, which is the set of acceptable labels; often the units all have the same domain, in which case $D_1 = D_2 = \ldots = D_n = D$, $\mathbf{D} = \{D_1, D_2, \ldots, D_n\}$.
- a labeling $\mathbf{L} = (L_1, L_2, ..., L_k)$, where k $\leq$ n, $L_i = (u_i, l_i), l_i \in D_i$; L_is are called unit labels.

If $k < n$ the labeling is a partial labeling, and if $k = n$, it is a complete labeling. A unit can have any label which is in its domain. However, there are restrictions on the labels a set of units can have simultaneously in order to be consistent. These constraints are expressed by a constraint relation **R**. Potentially **R** can be an n-ary relation.

A pair of unit labels $L_i = (u_i, l_i), L_j = (u_j, l_j)$ are consistent if and only if $(u_i, l_i, u_j, l_j) \in \mathbf{R}$. A given labeling $\mathbf{L} = (L_1, L_2, \ldots, L_k)$ is consistent if unit labels L_i and L_j are consistent for all $i,j \leq k$.

The labeling problem is to find a complete consistent labeling, given a set of units **U**, the domains of these units **D** and the constraint relation **R**. Other formulations of the problem ask for all solutions, or the largest (or *best*) partial labeling if a complete solution can not be found. The nature of the problem still remains unchanged.

2.1 Solutions to CLP

The consistent labeling problem can be solved in many ways. The simplest method is the *generate-and-test* method. Here all the possible solutions are enumerated and the consistent solutions are selected. Clearly this method is extremely slow and wasteful. In many cases the labels assigned to the first few units make the whole labeling inconsistent, and this can be detected early, during the configuration process. This observation saves a lot of time and is incorporated in the *standard backtracking* method.

In standard backtracking a single unit is assigned a label from its domain to start with. Then another unit is selected from the rest of the units, and a label is assigned to it in such a way that the partial labeling derived so far still remains consistent. Thus, attempt is made to incrementally get a labeling which is consistent. If at some point such a label can't be found, this process is backed up to the last unit which was assigned a label and the next possible label which keeps the labeling consistent is tried. Normally this process goes back and forth until a consistent labeling is found. If we run out of labels, there is no solution that is complete and consistent.

Clearly this is much more efficient than the generate-and-test method. However, it is not good enough for large practical applications. There have been several approaches to overcome this shortcoming. Gaschnig [1] attempted to solve it within the framework

of backtracking. He gave two new backtrack-type algorithms: *Backmark*, where all the redundant pair-tests are eliminated, and *Backjump*, where it is possible to backtrack across multiple levels, instead of just one level as in standard backtracking. These algorithms indeed perform better. Haralick and Elliot [3] gave another algorithm: *forward checking* to improve backtracking and it performs better under certain circumstances. Haralick *et al* [2,4,5] have described two 'look-ahead' operators, Φ and Ψ, to reduce the computation during backtracking.

Waltz [13] took another approach to solve this problem. The basic idea is to assign all possible labels to all the units, and then remove a label from a unit, if it is not compatible with the labels of the other units. This in turn would make some labels of some units inconsistent. This process continues until no label can be removed from any of the nodes or all the labels from all the nodes. In the former case it is still necessary to search for a solution, while in the later case there is no solution.

Mackworth [9] and others use yet another approach to reduce computation time during the backtracking process. For binary constraints the problem can be formulated in terms of graphs, where nodes correspond to the units, and the arcs represent the constraints between them. Each node has an associated label set, which is the set of possible labels for the unit. The fundamental properties of such networks are studied in [11]. Three consistency tests are given in [9] to prevent the 'thrashing behavior' of the backtracking algorithms: *node* consistency, *arc* consistency, and *path* consistency tests. Several algorithms are also described to achieve these three consistencies in the network. These consistency tests are applied first to remove inconsistent labels and then the backtracking process is applied to obtain a solution. Mohr and Henderson [10] have given an optimal algorithm for arc consistency and an improved algorithm for path consistency. In [7] Henderson gave a unified view of relaxation, both as a numerical and as a symbolic technique. The solution to the later case is formulated as an iterative procedure of boolean operations.

3 The CRAY Supercomputer

The CRAY series of computers: CRAY-1, CRAY X-MP, CRAY-2 are designed primarily for large scale scientific applications. The core of these systems is a high speed vector unit, which can be very effectively used in large scientific applications. Since, the system used for our implementation is a CRAY X-MP/48, we will describe its architecture in detail.

The CRAY X-MP series of computers combine high-speed scalar and vector processing with multiple processors, large and fast memories and high performance I/O. The scalar performance of each processor is attributable to its fast clock cycle (9.5 nsec), short memory access times and large instruction buffers. The CRAY X-MP/48 has four processors and has a maximum combined potential speed of the system of 750Mflops. The vector performance is supported by the fast clock, parallel memory ports, and flexible

hardware chaining. These features allow simultaneous execution of memory fetches, arithmetic operations and memory stores in a series of linked vector operations. This results in high-speed vector processing capabilities for both short and long vectors, characterized by heavy register-to-register or heavy memory-to-memory vector operations.

The CRAY X-MP/48 computer has 8 million words of directly accessible memory and each word is 64 bits long. The four processors share the central memory organized in interleaved memory banks that can be accessed independently and in parallel during each machine clock period. Each processor has four parallel memory ports connected to central memory; two for vector fetches, one for store and one for independent I/O operations. This coupled with the short memory access time provides a high-performance memory system with a high bandwidth to support high-speed CPU and I/O operations in parallel. The memory access time is 14 clock periods for a scalar and 17 clock periods for a vector quantity.

3.1 Programming Environment

CRAY X-MP can run under two operating systems: COS, a batch oriented system, and CTSS, a time shared system. Although several languages are available on the X-MPs, (e.g., Pascal, PSL, etc.) Fortran (called CFT) remains the most widely used. Our implementation used the CFT and hence we describe the features of the CFT compiler in some detail below.

CFT complier translates the Fortran code into CAL (Cray Assembly Language) that make effective use of the machine architecture. CFT itself is an extended version of the ANSI Fortran 77. The most important aspect of the compiler is its ability to auto-vectorize the CFT code. Thus the programmer does not have to modify the code to run on the CRAY. However, to take real advantage of the architecture one must know how the compiler vectorizes. When the code is properly structured, it may reduce the computation time dramatically.

CFT analyzes the inner most DO loops of the Fortran program to determine if the vector processing methods can be applied to improve the program efficiency. If it leads to improvements in the performance, the compiler produces vector instructions to drive the high-speed vector and floating-point functional units and the vector registers (eight). Clearly, it is not possible to vectorize all DO loops and whether a DO loop is vectorizable depends on the statements in the loop and their relationships.

A DO loop must manipulate on the contents of at least one array in order to be vectorized. Also, only the innermost DO loop can be vectorized. Further several conditions inhibit the vectorization. Some of them are:

- CALL, ELSEIF, RETURN, STOP, PAUSE, or I/O statements inside the loop.

- Backward branches in the loop.
- Statement numbers with reference from outside the loop.
- IF statements which may not execute due to the effects of previous IF statements.
- References to character variables, arrays, or functions.

The compiler also inhibits vectorizations of *DO* loops with dependencies. A dependency exists of the following conditions are met.

- An array is referenced and defined in the *DO* loop.
- An array element defined in a previous pass of the DO loop is referenced.

The implications of these will be discussed in the next section and while discussing the implementation details in section 5.

4 Arc Consistency Algorithms

In this section the four arc consistency algorithms, usually referred to as AC-1, AC-2, AC-3, and AC-4 are briefly described. The method given in [7] will also be described. We will refer to it as DR. They are also analyzed for their suitability for vectorizing on the CRAY supercomputer. The algorithms are described in detail in [9] and [10]. It is assumed that node consistency is already achieved.

Before the algorithms are described, the *REVISE* procedure which is used by AC-1, AC-2, and AC-3 is discussed. Although the algorithms are given in the above references, we include them here for two reasons. First, the control structures in Fortran are different and the algorithms have to be slightly modified to portray the computation flow more accurately. They are also included for the sake of completeness.

4.1 REVISE Procedure

This procedure is used by AC-1, AC-2, and AC-3, to enforce consistency along a single arc, say (i,j). It removes all the labels at node i, which doesn't have any support from labels at node j. It also returns *true* if one or more labels are deleted, and returns *false* otherwise. It is sketched in Figure 1.

There is a major implementational issue here, as far the CRAY is concerned. Clearly REVISE is the innermost subroutine and only its code can be vectorized. Also, the innermost loop in the procedure can't be vectorized as it is, since, it doesn't manipulate an array. It just looks to see if there is any support for a label. So, the code has to be slightly modified to take advantage of vector processing.

```
procedure REVISE(i,j)
begin
    DELETE := false;
    for each x ∈ D_i do
    begin
        support = false ;
        for each y ∈ D_j do
        begin
            if (support) then continue;
            if (P_ij(x,y)) then support := true ;
        end
        if ~ support then
        begin
            delete x from D_i ;
            DELETE := true ;
        end
    end
    return DELETE ;
end
```

Figure 1 : REVISE Procedure

4.2 AC-1

AC-1 is essentially a brute force algorithm. During each iteration arc consistency is enforced for every arc in some order. If there is any change in the label sets of any one of the nodes this process is repeated. The algorithm terminates when there is no change in the label sets of any of the nodes. AC-1 is given in Figure 2.

```
procedure AC-1()
begin
    repeat
        CHANGE := false ;
        for each (i,j)∈ ARCS do
            CHANGE := CHANGE ∨ REVISE(i,j) ;
    until ~ CHANGE ;
end
```

Figure 2: Structure of AC-1

From the implementation standpoint it is the easiest. It also is the most inefficient of the four AC algorithms. Its time complexity is $\mathbf{O}(ea^3n)$.

4.3 AC-2

AC-2 is similar to Waltz filtering in spirit. It makes the network consistent in a single pass through the nodes. Although its structure is more complex than AC-3 it is actually a special case of the later. Q and Q' are two Boolean arrays used to maintain the two queues.

```
procedure AC-2()
begin
    for i := 1 to n do
    begin
        Q := {(i,j) | (i,j) ∈ ARCS, j < i}
        Q' := {(j,i) | (j,i) ∈ ARCS, j < i}
        while ~ Qempty do
        begin
            while ~ Qempty do
            begin
                pop (k,m) from Q ;
                if REVISE(k,m) then
                  Q' = Q ∪ {(p,k) | (p,k) ∈ ARCS, p ≤ i, p ≠ m}
            end
            Q := Q' ;
            Q := nil ;
        end
    end
end
```

Figure 3: Structure of AC-2

One of the problems of implementing this in Fortran 77 is the lack of dynamic data structures. So, the queues Q and Q' have to be simulated using arrays. This leads to some amount of inefficiency. The time complexity of AC-2 is $\mathbf{O}(ea^3)$.

4.4 AC-3

The structure of AC-3 is similar to AC-1. However, in AC-3 only the arcs associated with the nodes whose label sets have changed during the previous iteration are added to the pool of arcs to be checked for consistency. Thus, in general, it does less computation than the AC-1.

```
procedure AC-3()
begin
    Qempty := true ;
    Q := All the Arcs in the graph;
    repeat
        for each (k,m) ∈ ARCS do
        begin
            if (i,j) ∉ Q then continue ;
            Remove (k,m) from Q ;
            Qempty := false ;
            if REVISE(k,m) then
              Q := Q ∪ {(i,k) | (i,k) ∈ ARCS, i ≠ k, i ≠ m }
        end
    until Qempty ;
end
```

Figure 4: Structure of AC-3

Again, the lack of dynamic data structures make a Fortran implementation somewhat less efficient. The time complexity of AC-3 is $\mathbf{O}(ea^3)$.

4.5 AC-4

In AC-4 several data structures are used to reduce the time complexity. The algorithm consists of three parts: building the appropriate data structures: *M*, *S*, and *Counter*, initialization of *List*, and pruning the inconsistent labels. We sketch each of the stages separately for ease of understanding.

In the first stage, very detailed information about which label supports which other labels at other nodes, how many supports each label at each node has, etc., are computed and stored in the several data structures. The storing of this information explicitly, helps the later phase where the labels without support are deleted.

```
procedure BuildDataStructures()
begin
      for each (i,j) ∈ ARCS do
         for each b ∈ D_i do
         begin
               total := 0 ;
               for each c ∈ D_j do
                  if R(i,b,j,c) then
                  begin
                        total := total + 1 ;
                        Append(S_jc,(i,b)) ;
                  end
               if(total = 0) then
                 M[i,b] = 1;
                 Delete b from D_i ;
               else
                 Counter[(i,j),b] := total ;
               endif
         end
end
```

Figure 5: Building Data Structures for AC-4

In the next stage, which actually can be part of the first one, the *List* is initialized to all the node-label pair which do not have any support and can be deleted. The pruning process starts off with these pairs to prune other labels if possible.

```
procedure InitList()
begin
      for i:= 1 to n do
         for b:= 1 to a do
            if (M[i,b] = 1) then Append(List,(i,b)) ;
end
```

Figure 6: Initialization of *List* in AC-4

In the last stage, the labels which don't have any remaining support are deleted. We start with the labels which have no support and hence are stored in *List*. These are deleted and the support-count for the labels which are supported by these labels are decremented. If during this process a support-count of a label goes to zero, then it is deleted and is added to the *List*. This continues until *List* is empty.

```
procedure Prune()
begin
      while List not empty do
      begin
            Pop (j,c) from List ;
            for (i,b) ∈ S_jc do
            begin
                  Counter[(i,j),b] := Counter[(i,j),b] - 1 ;
                  if (Counter[(i,j),b] = 0) ∧ (M[i,b] = 0) then
                  begin
                        Append(List,(i,b)) ;
                        M[i,b] := 1 ;
                        Delete b from D_i ;
                  end
            end
      end
end
```

Figure 7: Pruning in AC-4

The main problem with AC-4 is the space. Generally it takes up more space than the other three algorithms, because it has to store lots of information explicitly. This of course reduces the time complexity of the algorithm. It has been proved that AC-4 is optimal and is $\mathbf{O}(ea^2)$.

The structure of both *BuildDataStructrures* and *Prune* procedures had to be changed to make it vectorizable. *InitList* is directly vectorizable without any modification.

4.6 Boolean Formulation

Here the approach given in [7] is briefly discussed and then a simple algorithm to realize it is described.

Let Λ_{ij} be the m by m compatibility matrix between units i and j. $\Lambda_{ij}(k,p)$ is the element in the k'th row and p'th column of Λ_{ij}. L is a m by a labeling matrix, where $L_{ik} =$ 1 means the k'th label of i'th unit is still a viable label. If it is 0, then the label is deleted. The iterative solution can be written as:

$$L_{ik} \leq L_{ik} * \prod_{j=1}^{n} \left[\sum_{p=1}^{m} (L_{jp} * \Lambda_{ij}(k,p)) \right]$$

The above formula says how to update the label sets of the units iteratively. The

updating is done until no label is removed during one iteration. The algorithm is described below. We refer to this algorithm as DR.

```
procedure DR()
begin
    repeat
        Change := false ;
        for i := 1 to n do
            for k := 1 to a do
                begin
                    temp := ComputeL_ik
                            (* use the above formula *)
                    if (temp ≠ L_ik) Change := true ;
                end
    until ~ Change ;
end
```

Figure 8: Algorithm for Using Boolean Formulation

5 Implementation

All the algorithms (AC-1, AC-2, AC-3, AC-4, and DR) are implemented on the CRAY and the Vax (VAX-8600). On CRAY they are implemented in Fortran since it is the most efficient language. To ensure fair comparison, the Vax implementations are also in Fortran 77.

On the CRAY, strong effort was made to vectorize the code where ever possible. This meant that the structure of some algorithms had to be changed. However, the Vax version didn't include these changes. The idea was to compare the best implementation on the CRAY with the best implementation on the Vax.

One of the problems we faced was scarcity of memory in the AC-4 and DR algorithms. On CRAY a whole word (64bits) is used for storing a 'LOGICAL' variable. On the Vax, also, a whole word is allocated for a boolean variable, although one can reduce it to 2 bytes, by setting some compiler flags. To make the implementations efficient, these two algorithms need large boolean arrays and this wastage of space prevents very large problems being solved. An alternative is to do your own memory management. However, this prevents the vectorization on the CRAY, since we are not manipulating arrays any more.

A problem with the Vax implementation is the paging. The Vax has virtual memory and hence one can do big problems. However, once the memory requirement is large, the time taken to swap in and out is a major concern. Since, the pattern of access in these problems is very unpredictable, the overhead is very high.

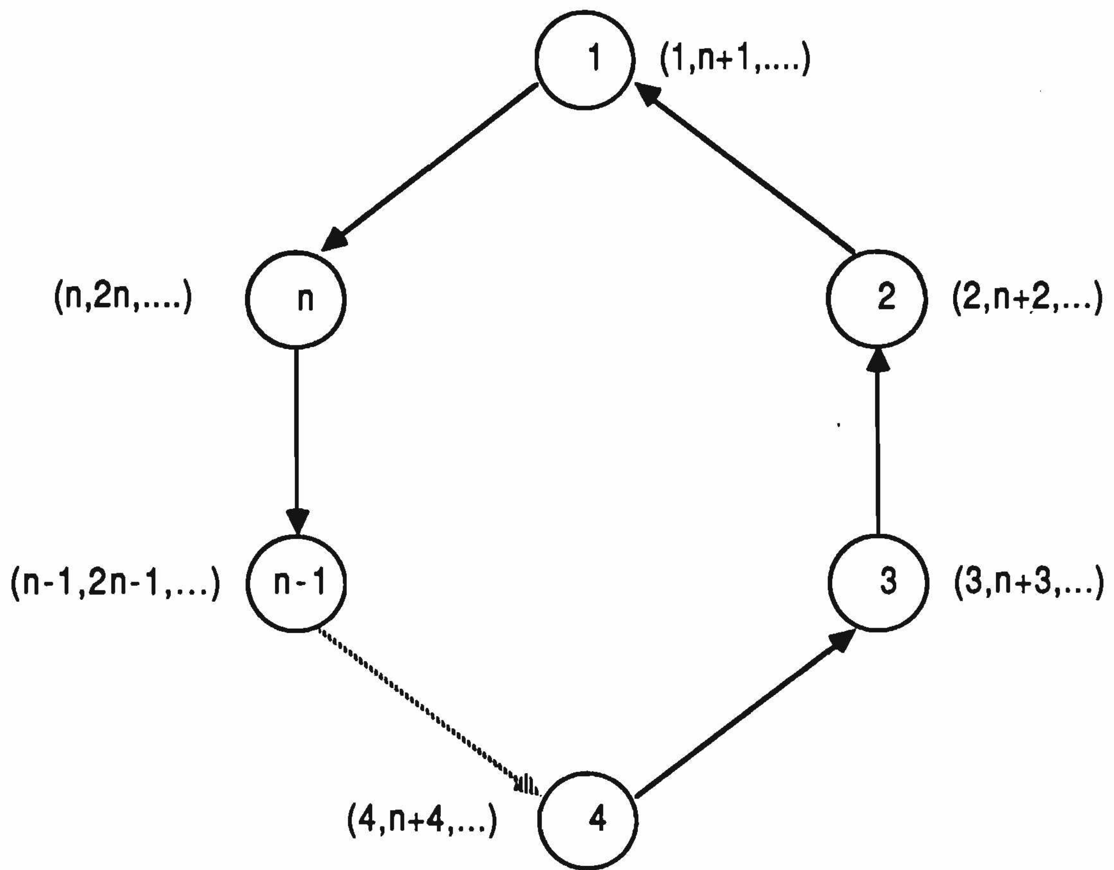

Figure 9: A Network Of Constraints

Lack of dynamic data structures in Fortran is a problem which has been somewhat discussed. Clearly, in AC2, AC3, and AC4, the performance would improve with such facilities in the implementation language.

5.1 Test Problems

We used three problems to test and compare the performance of the algorithms. They are the cyclic graph problem, n-queens problem and the graph coloring problem.

The cyclic graph problem is described in [12]. The nodes of the graph are connected to form a cycle, see Figure 9. The label set of node i is given by $\{ x : jn + i, 1 \leq j \leq a \}$, where there are **a** labels and **n** nodes. The constraint used in the *greater-than* relation. In this problem, only one label is removed from the network in one iteration and hence it takes **na** iterations to converge. There is no solution to this problem and hence, the label sets all go to nil.

In the n-queens problem, we use the row placement strategy and place a queen on each row. The possible labels for each queen are 1, 2, ..., **n**. The consraint is that no two queens should attack each other. If we don't constraint the search space further, nothing will happen since, for every position of queen i there is at least one position for queen j which is compatible. We chose to fix the position of two queens (n'th and n-1'st) such that they conflict and hence there is no solution. This forces the label sets of all units to go to nil.

The underlying graph in the n-queens problem is a complete graph; hence the number of arcs is $\mathbf{O}(n^2)$. In the case of the cyclic graph the number of arcs is $\mathbf{O}(n)$. So, we decided to generate some graphs where the number of arcs is halfway between the above two. Two nodes of a graph were connected (or left unconnected) depending on the outcome of a random event (e.g., a uniform random number generator produces a number greater than 0.5). We also, added a K-clique to the graph. Overall number of arcs remained halfway (approximately) between n and n^2. We used the graph to color the nodes in such a way that no two adjacent nodes have the same color. Number of colors allowed is K-1, so that no coloring possible. So, in this case also the label sets of all the nodes go to nil.

All these problems were tried for several problem sizes (number of units). The results only show for $\mathbf{n}$ = 10, 20, 30, 40 and 50. The number of labels in n-queens is automatically fixed by **n**. In the cyclic graph problem we chose it to be the same as **n**. In the graph coloring the number of colors is taken to be $(\mathbf{n} \div 8)+2$ to keep the number of colors reasonable.

6 Results

Figure 10 compares the performance of the five algorithms on the Cray for the Cyclic Graph problem. Figure 11 shows the performance of the algorithms for the same problem, but on the Vax. In Figure 12 the times taken by AC-1 on the two machines is plotted as a function of the problem size. Figures 13 to 16 do the same for the other four algorithms: AC-2, AC-3, AC-4, and DR. Similarly, Figures 17 to 23 show the performance of the algorithms for the N-Queens problem. Figures 24 to 30 display the same for the Graph Coloring problem on the random graphs.

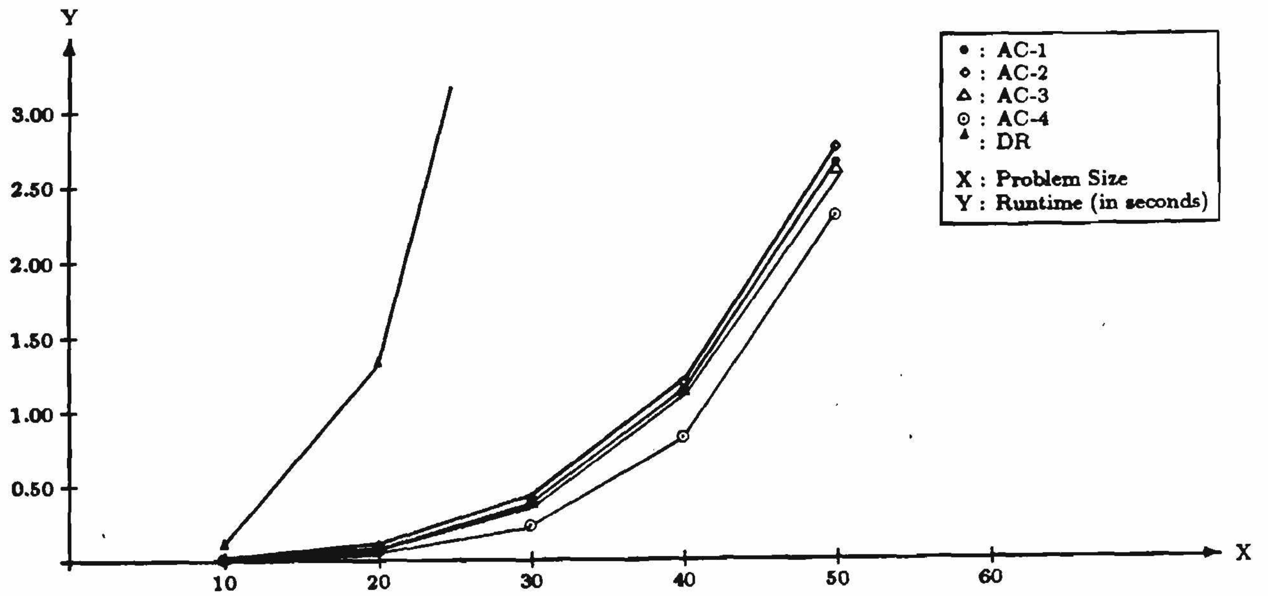

Figure 10 : Execution Times for Cyclic Graph on CRAY X-MP/48

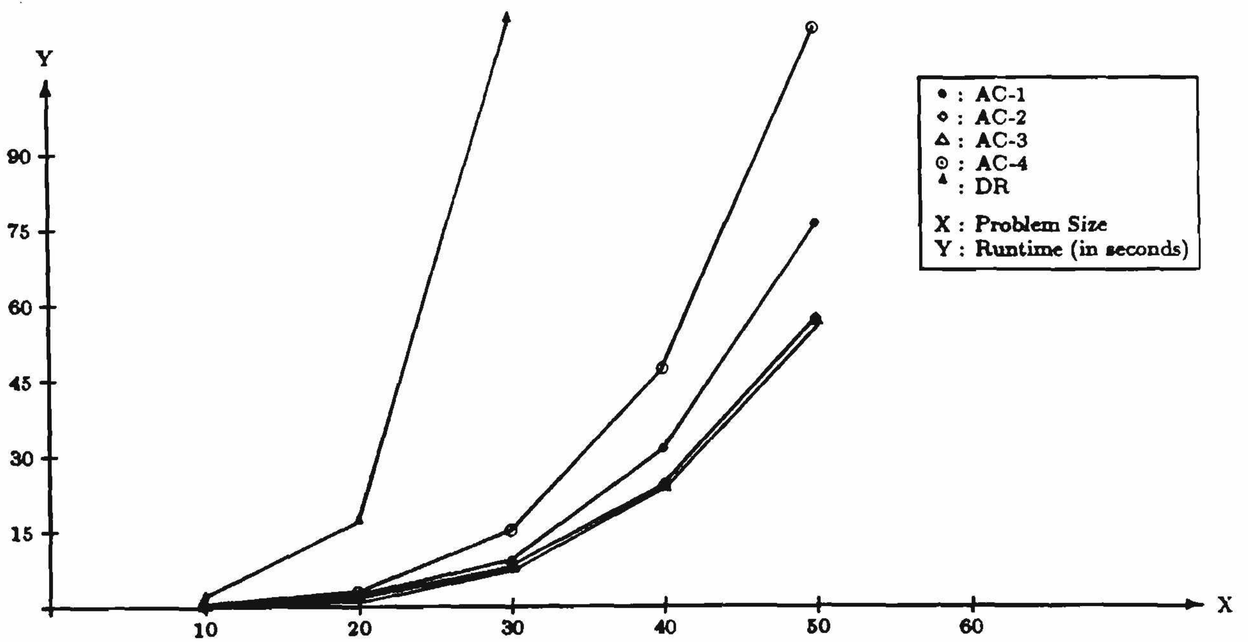

Figure 11 : Execution Times for Cyclic Graph on VAX 8600

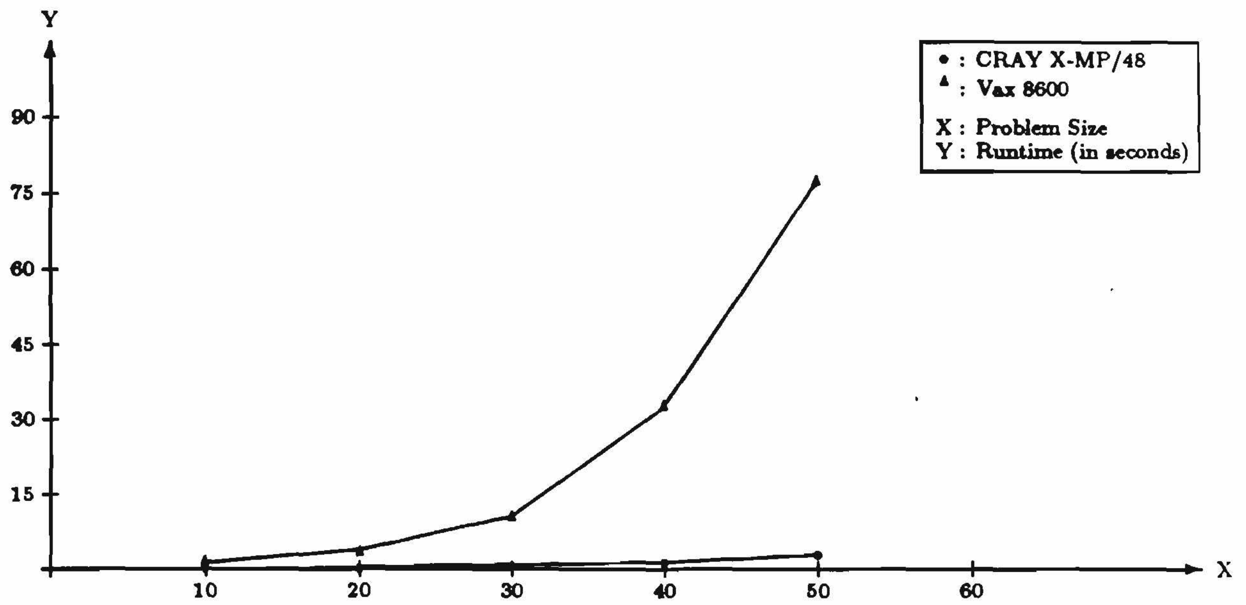

Figure 12 : Execution Times of AC-1 for Cyclic Graph

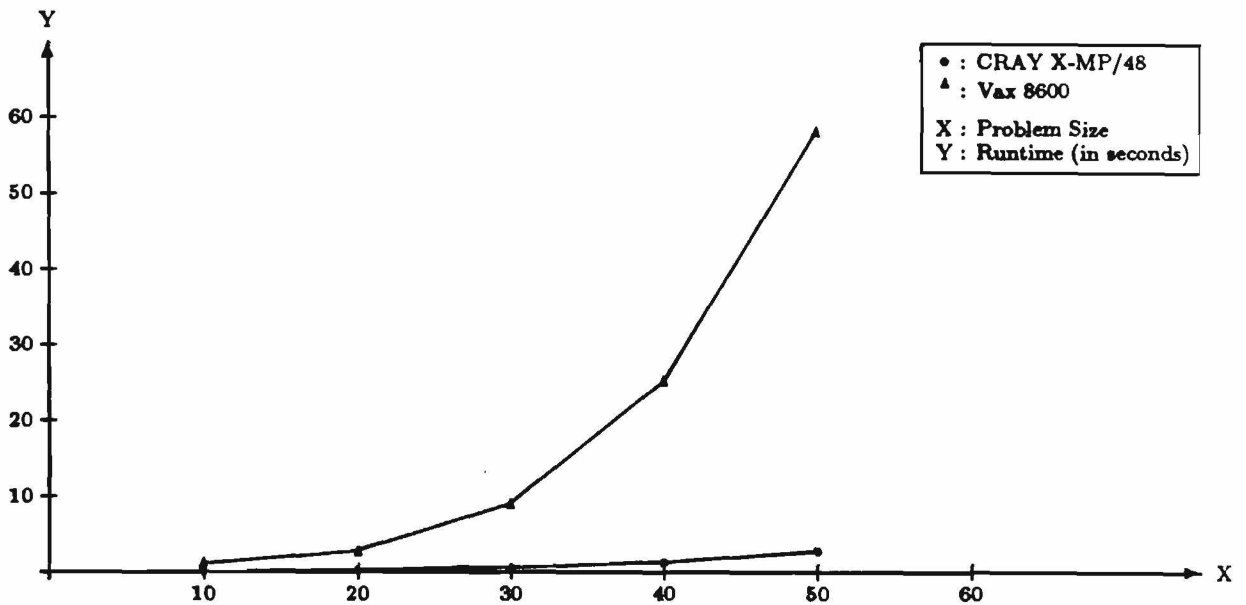

Figure 13 : Execution Times of AC-2 for Cyclic Graph

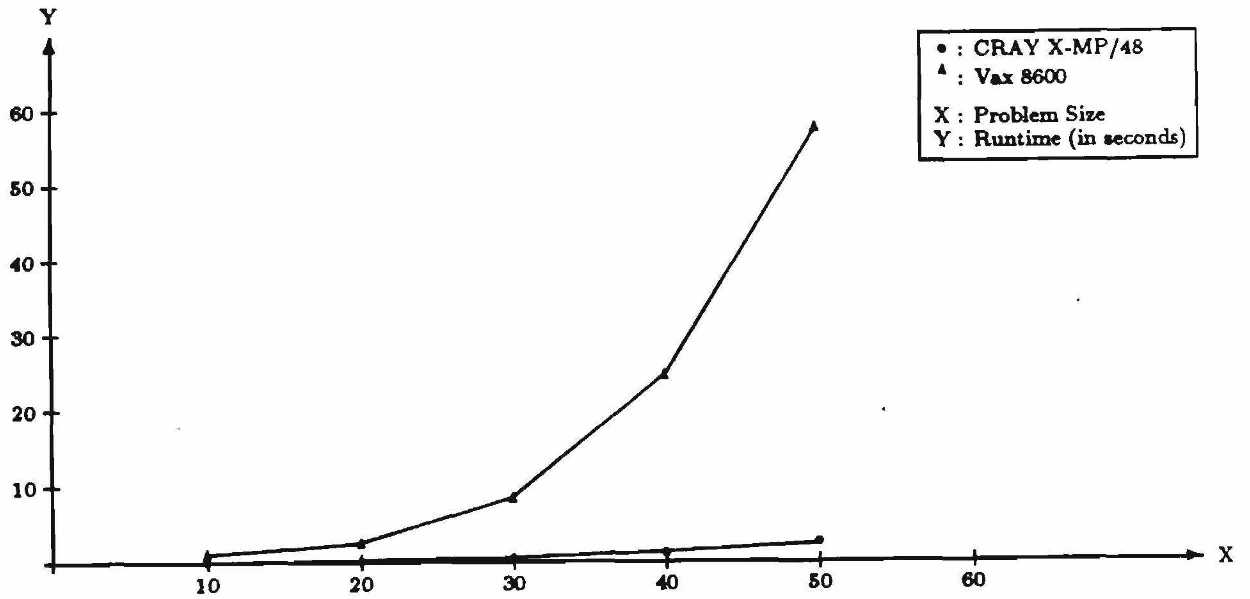

Figure 14 : Execution Times of AC-3 for Cyclic Graph

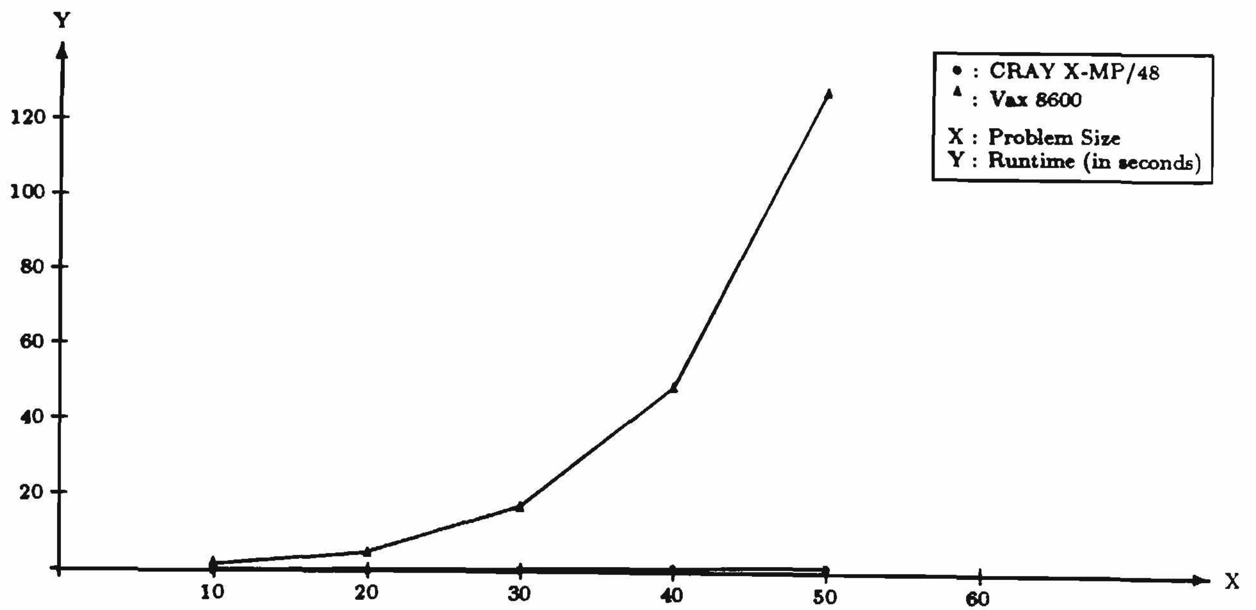

Figure 15 : Execution Times of AC-4 for Cyclic Graph

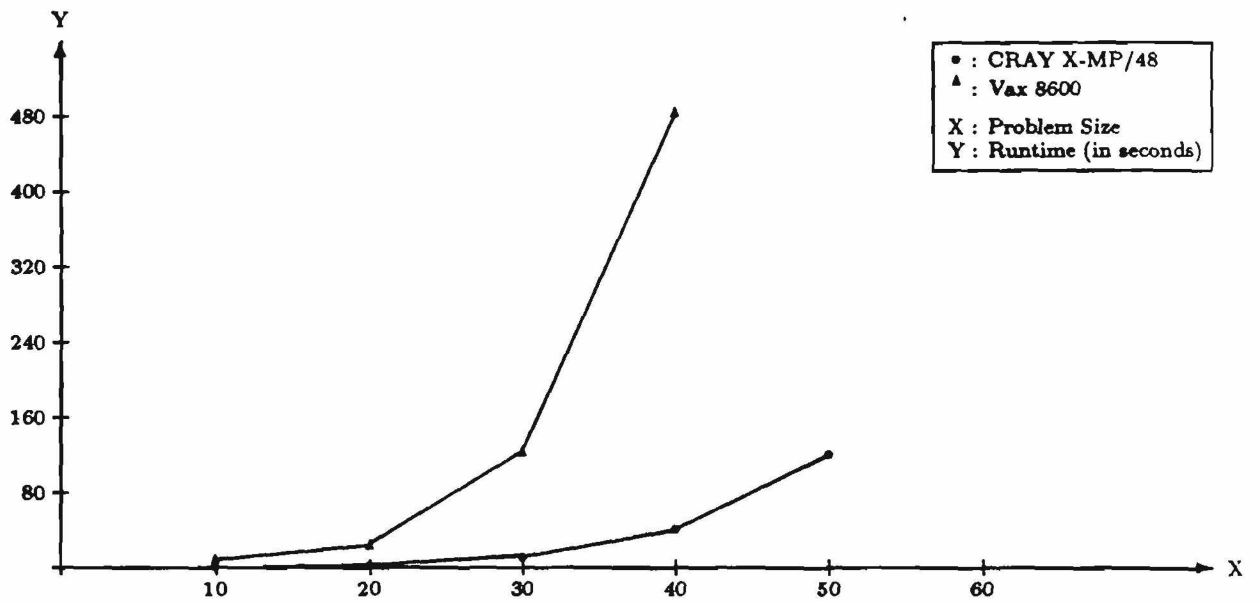

Figure 16 : Execution Times of DR for Cyclic Graph

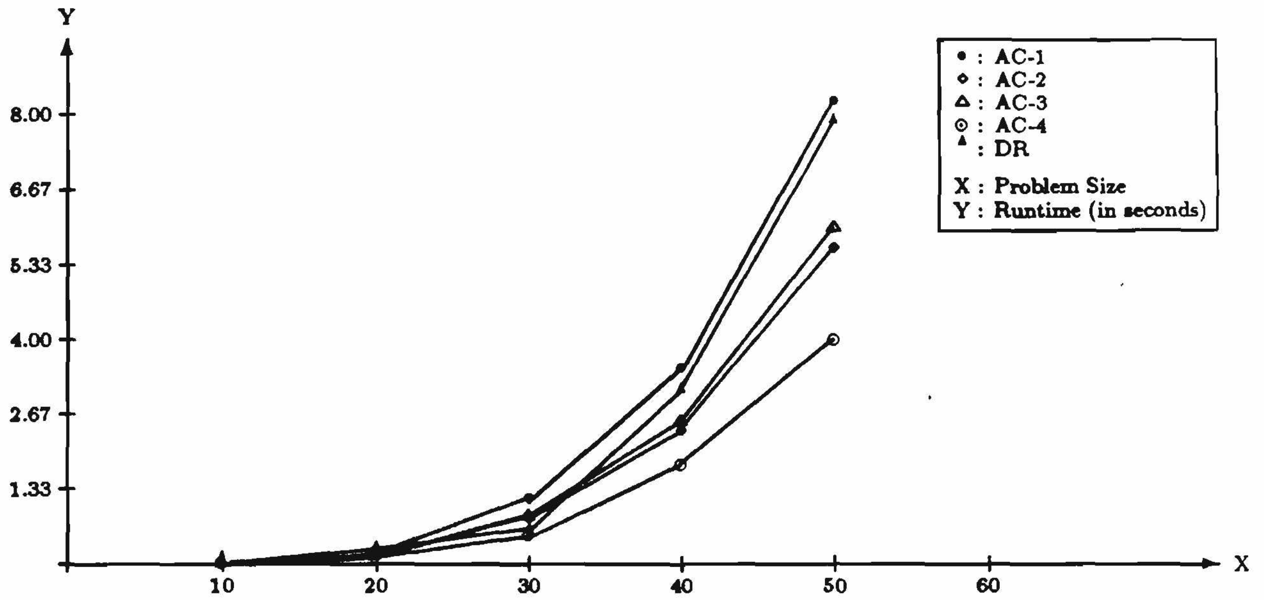

Figure 17 : Execution Times for N-Queens on CRAY X-MP/48

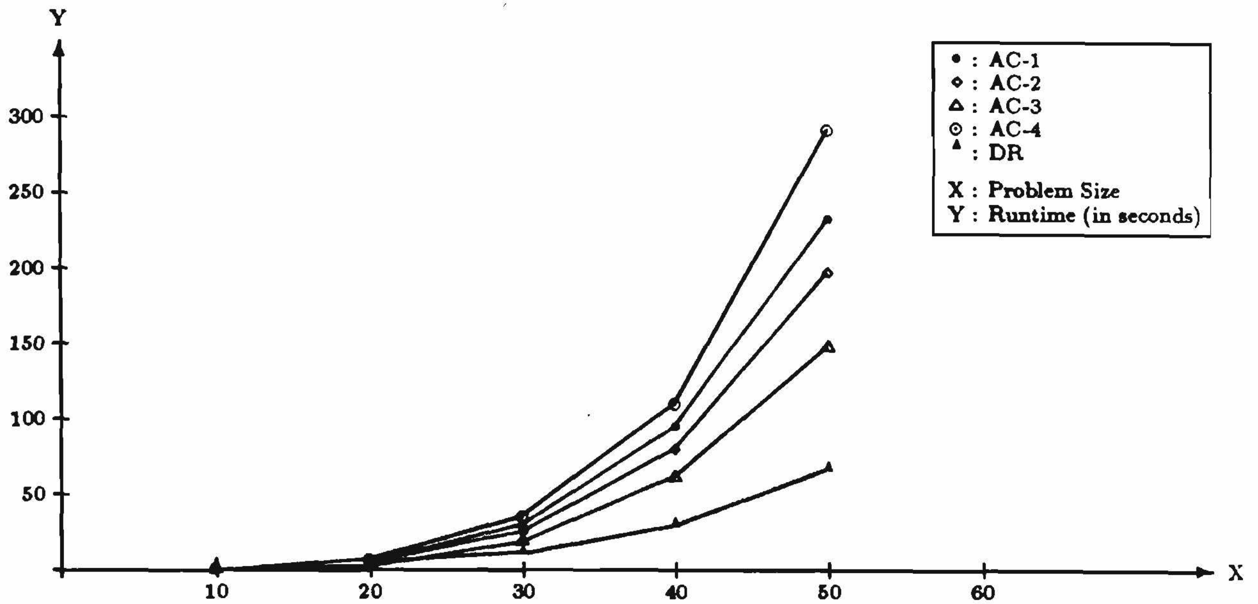

Figure 18 : Execution Times for N-Queens on VAX 8600

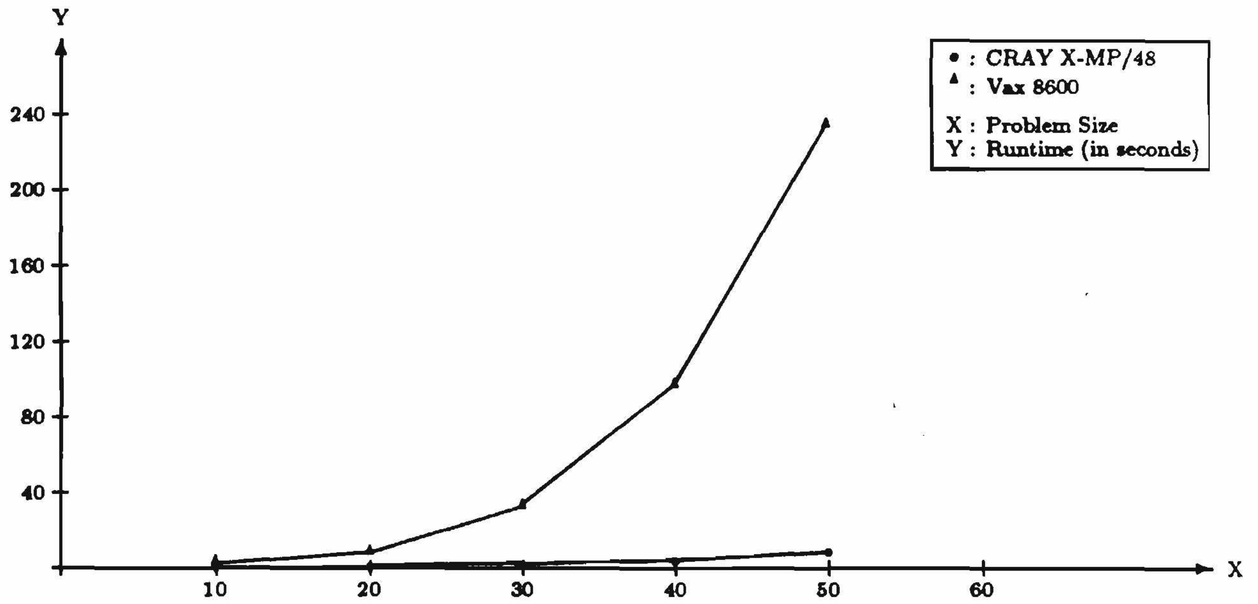

Figure 19 : Execution Times of AC-1 for N-Queens

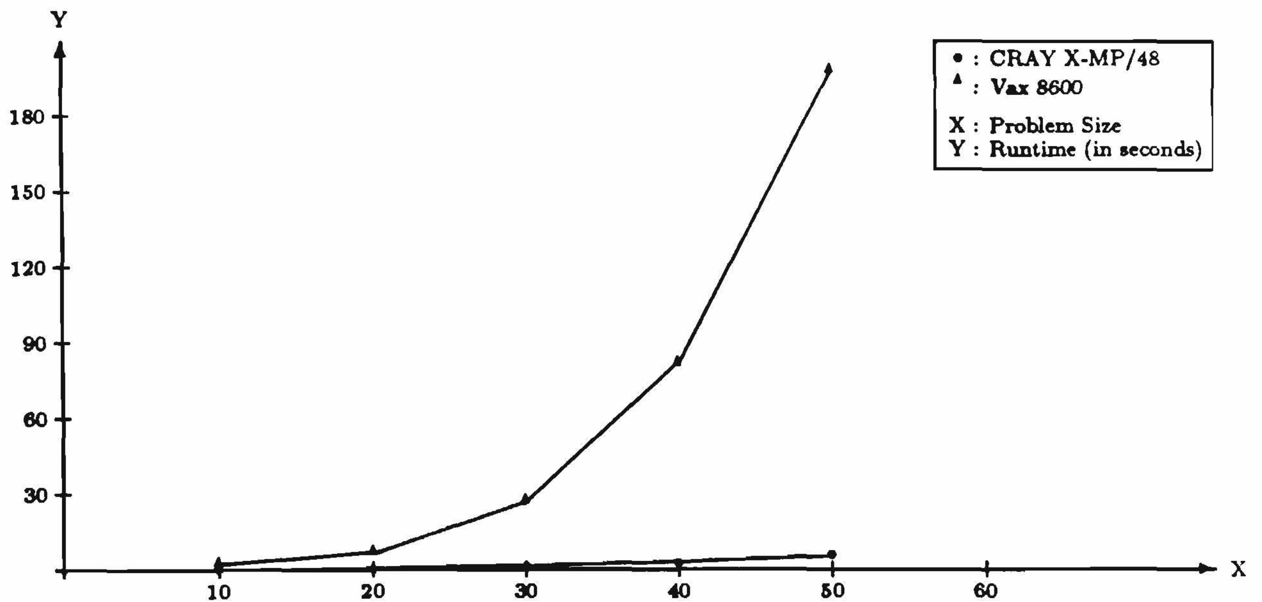

Figure 20 : Execution Times of AC-2 for N-Queens

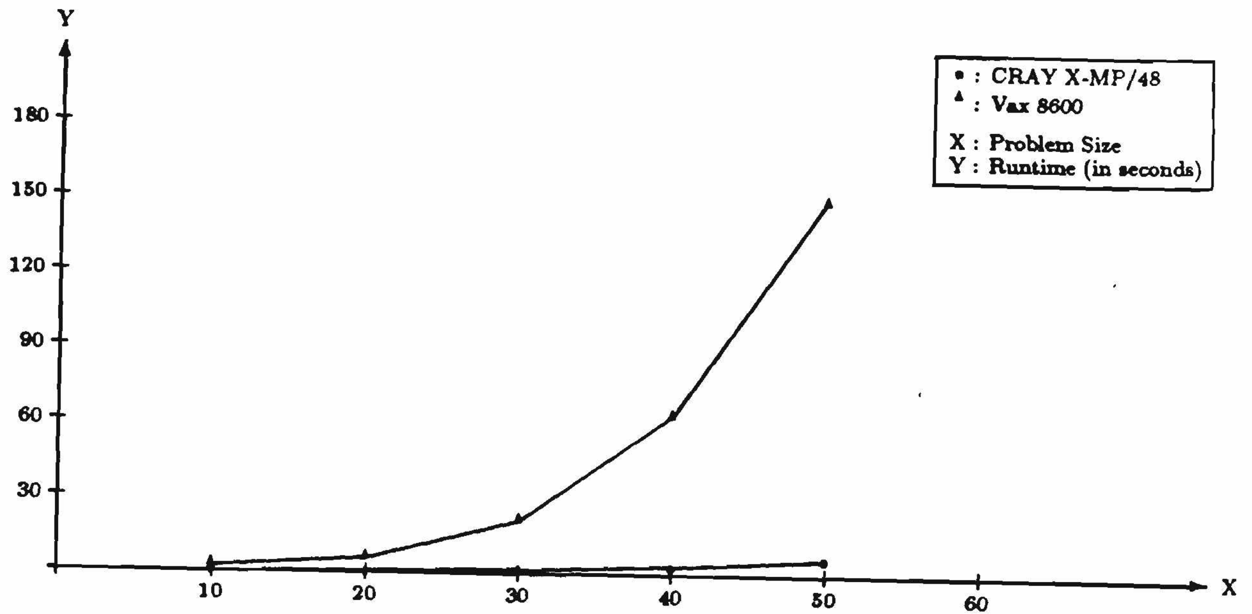

Figure 21 : Execution Times of AC-3 for N-Queens

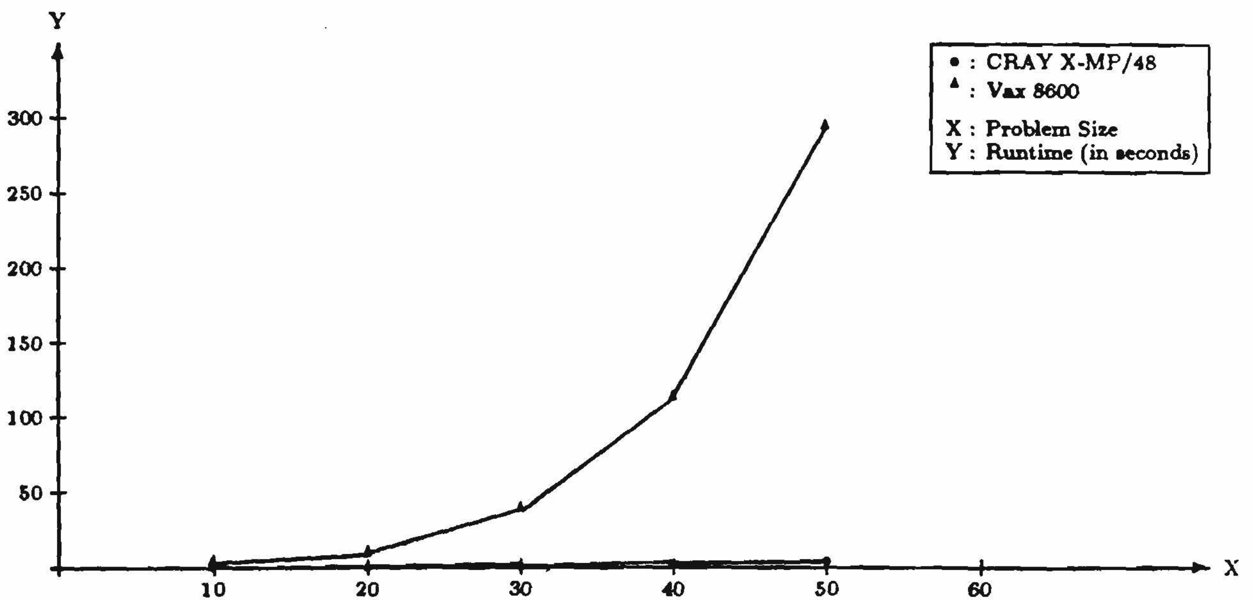

Figure 22 : Execution Times of AC-4 for N-Queens

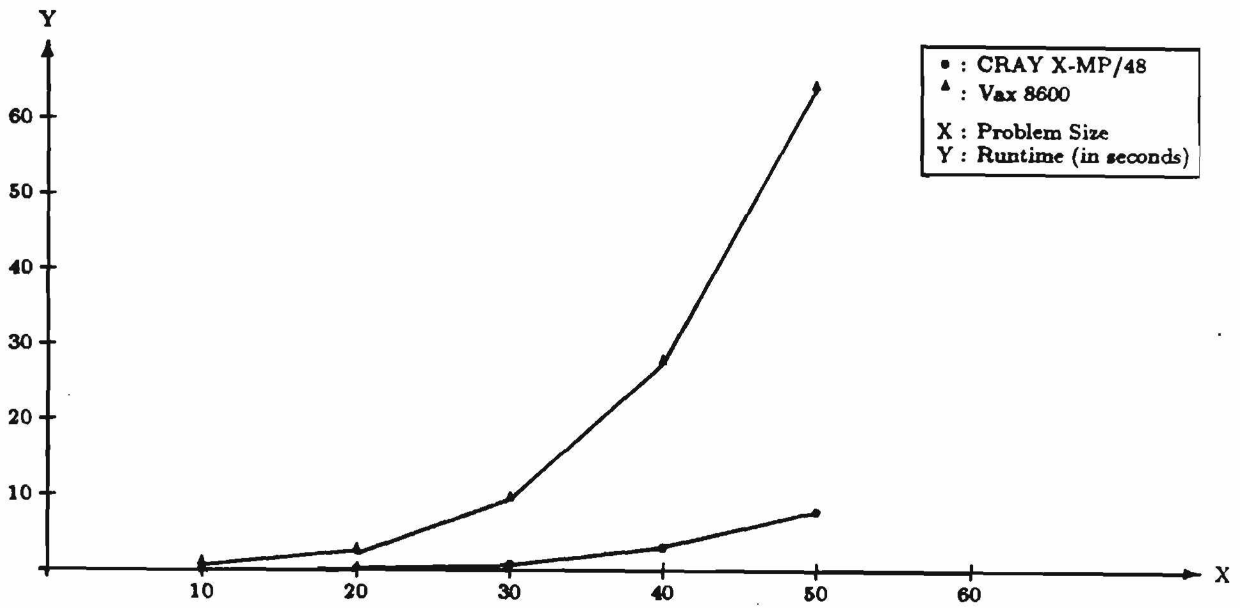

Figure 23 : Execution Times of DR for N-Queens

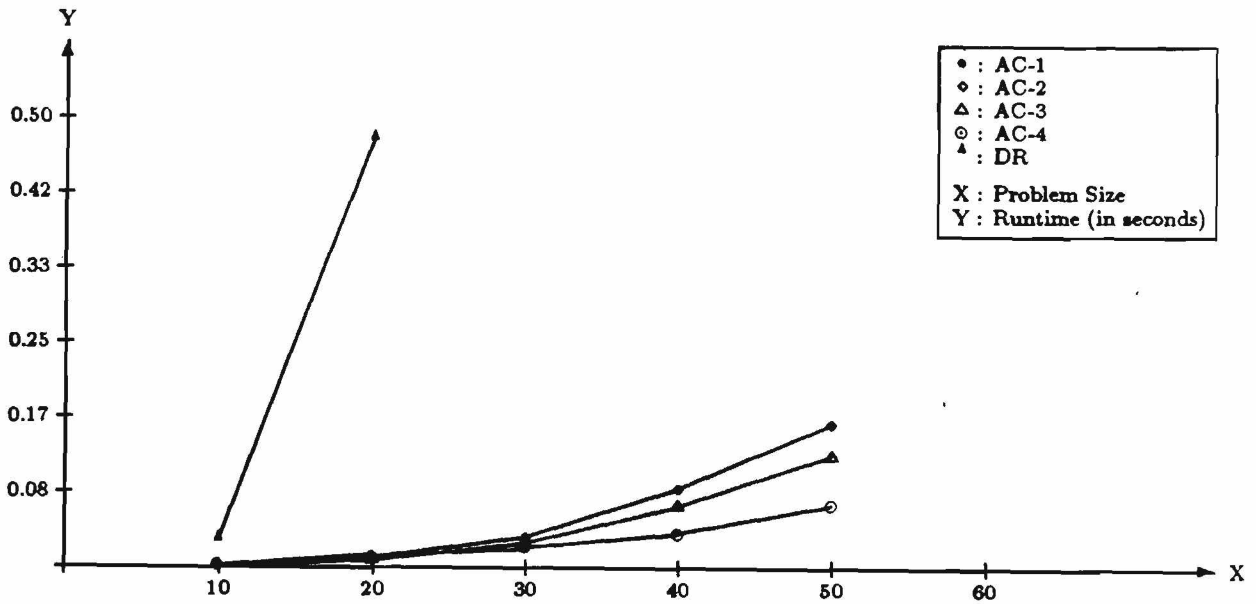

Figure 24 : Execution Times for Graph Coloring on CRAY X-MP/48

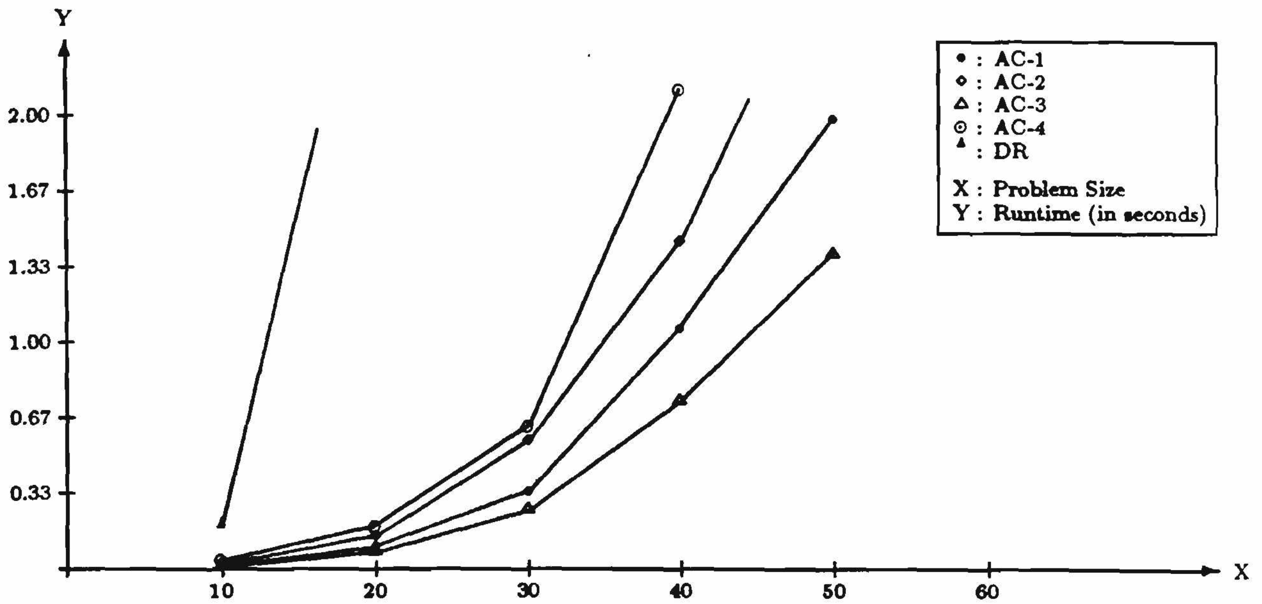

Figure 25 : Execution Times for Graph Coloring on VAX 8600

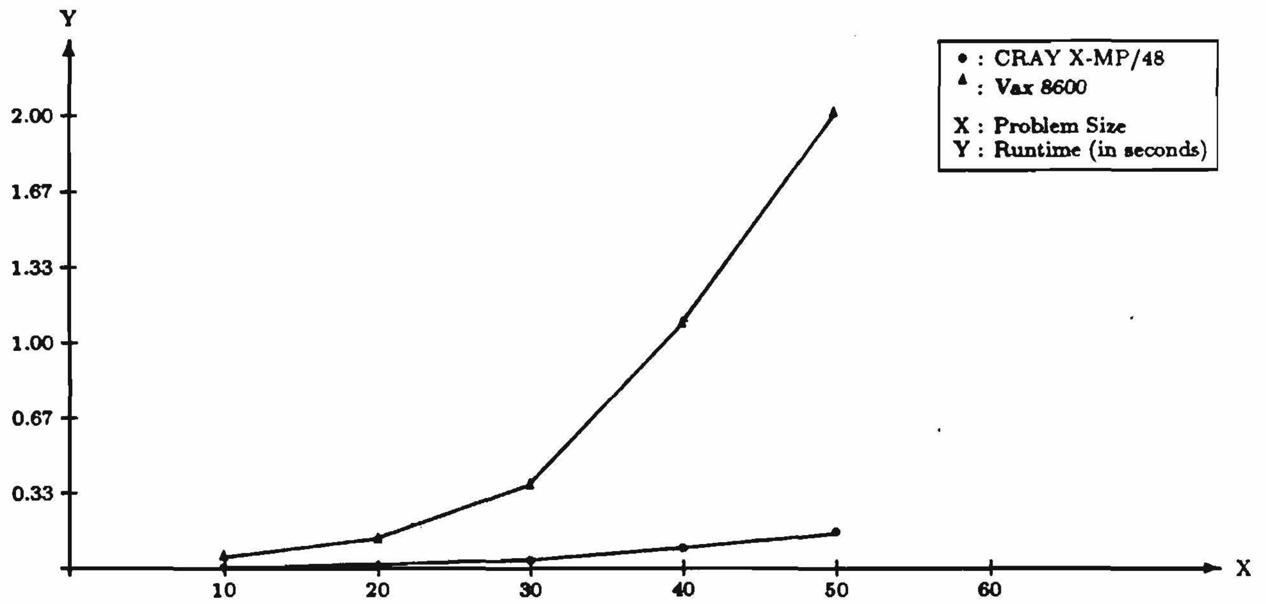

Figure 26 : Execution Times of AC-1 for Graph Coloring

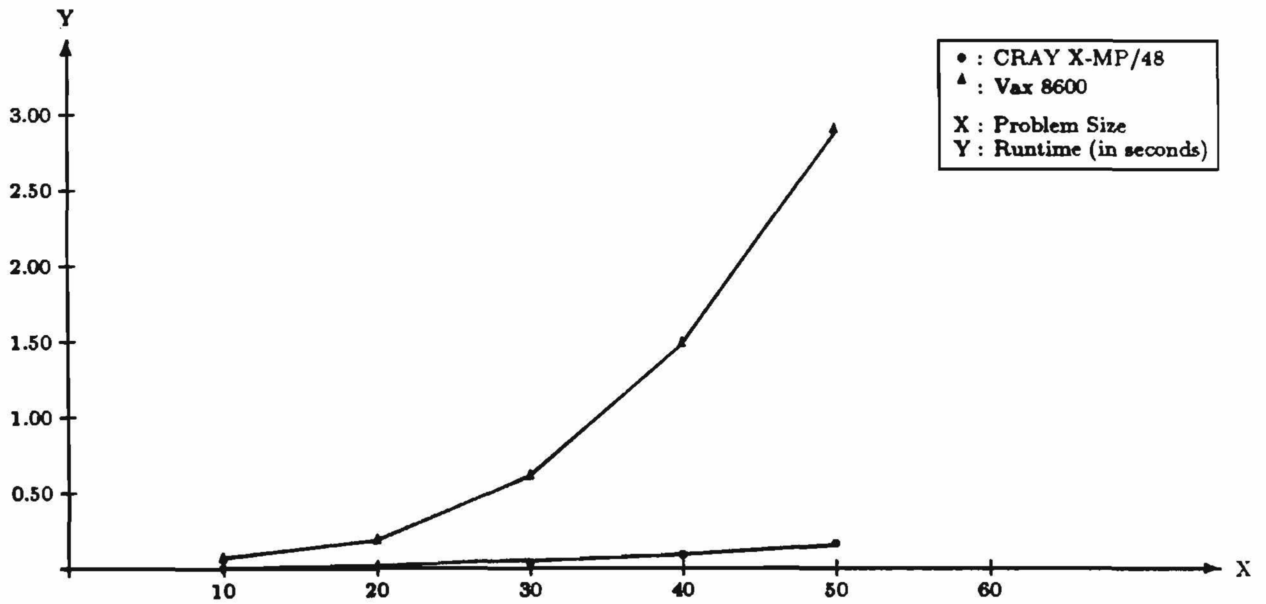

Figure 27 : Execution Times of AC-2 for Graph Coloring

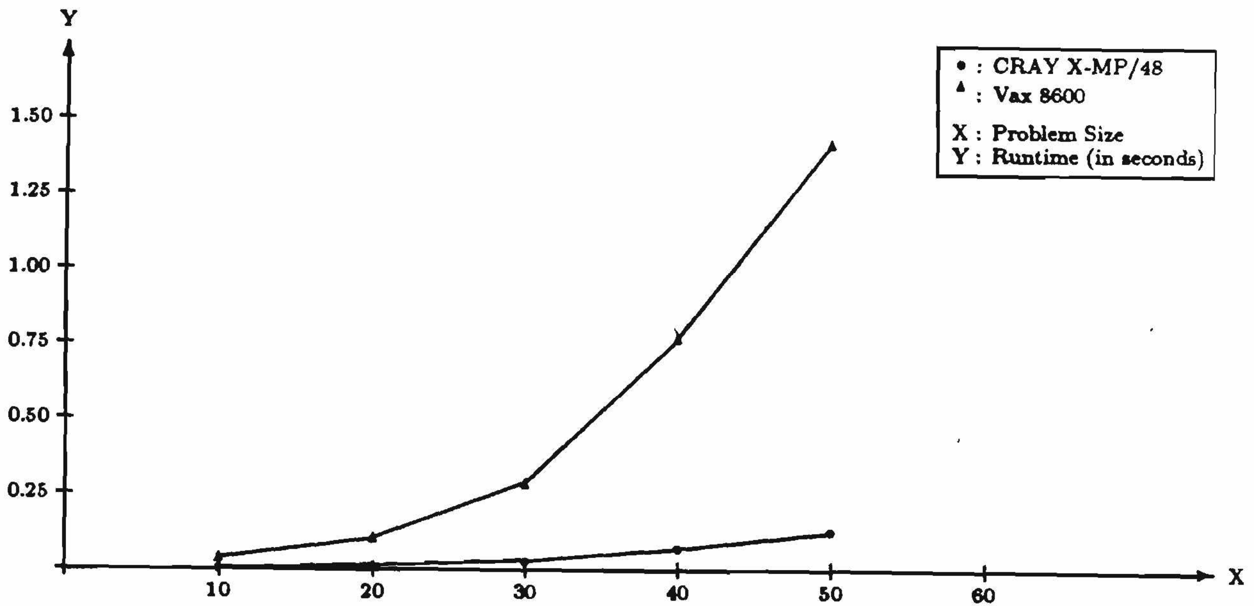

Figure 28 : Execution Times of AC-3 for Graph Coloring

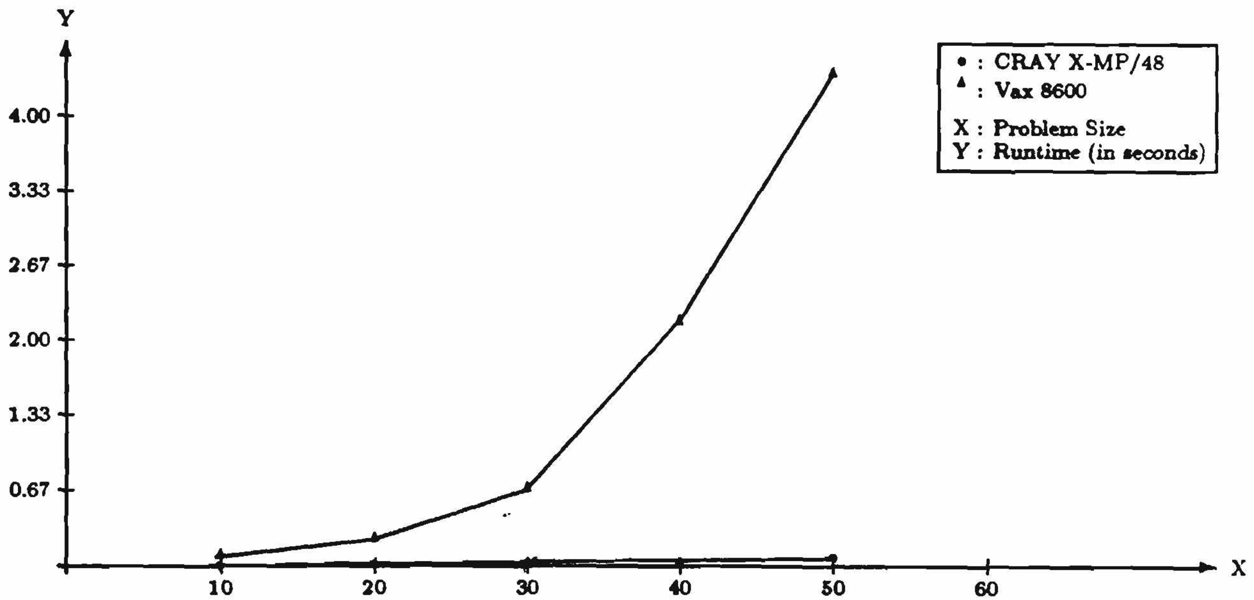

Figure 29 : Execution Times of AC-4 for Graph Coloring

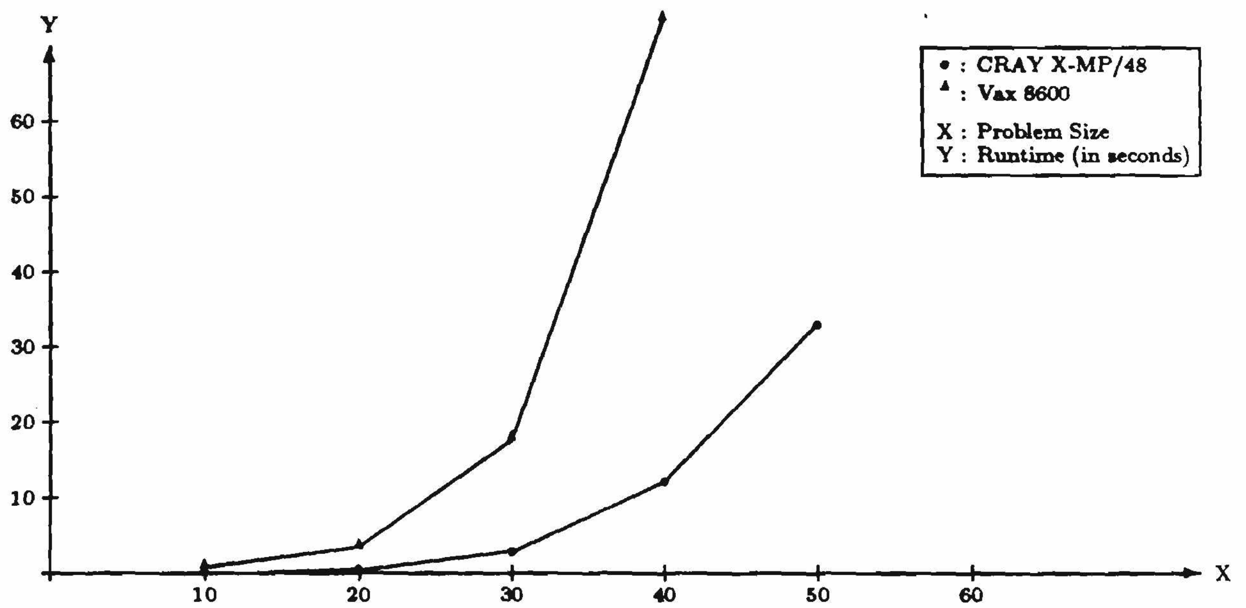

Figure 30 : Execution Times of DR for Graph Coloring

Several observations should be made about the performance of these algorithms. As expected, the performance of the algorithms on the Cray is much superior as compared to the Vax. However, the extent of speedup is different for different algorithms. The speedups obtained are in the range of 10 to 60. Clearly for the smaller problem sets, maximum advantage of vector processing can't be taken advantage of. So, the gain is not very conspicuous in these cases.

Theoretically, AC-4 is the best algorithm and should take the least amount of time after a certain point. In the Cray implementation it is fairly obvious. It can be seen that as the problem size increases, the difference is even more pronounced. However, such performance is not reflected in the Vax implementation. The reason is as follows. AC-4 needs very large amount of space to be most efficient. Since, VAX-8600 is a time sharing system, there is a lot of thrashing. Also, as the problem size increases, it gets worse.

It should be emphasized here, that this doesn't mean that AC-4 can't run efficiently on the Vax. As mentioned before, this rather large memory requirement is because the Fortran compiler allocates 16 bits for a logical type variable. Clearly one bit would suffice, and one can do that by managing the memory by oneself. The reasons for not doing this was explained in section 5.

In general DR performs very poorly. In fact, it is one order of magnitude slower than the others. However, for N-Queens it is competitive. It is because for N-Queens the graph is complete and hence, DR does no work that would not be done in other algorithms.

Among AC-1, AC-2, and AC-3, AC-3 is generally the fastest, both on the Cray and on the Vax. As pointed out before, AC-2 and AC-3 have the same time complexity and their performances are comparable. AC-3 generally fairs much better than AC-1.

7 Conclusion and Future Research

Several arc consistency algorithms are compared for their performance on Vax and Cray. Three problems with different characteristics are used to measure their efficiency. As expected, the implementation on the Cray is generally much better. Although AC-4 is the most efficient algorithm in theory, it was not reflected in the Vax implementation. However, on the Cray it is clearly visible. DR is generally expensive and should be avoided unless the graph is complete (or close to). Also, AC-3 is generally very competitive and takes less amount of space as compared to AC-4.

In order to be fair to AC-4 algorithm, it should be implemented on the Vax more efficiently as explained in section 6. Also, since multiprocessors are commercially available, it would be interesting to test the performance of these algorithms on them. Some of our current research is along these lines. We intend to implement these algorithms on the Butterfly multiprocessor and study their performance.

References

[1] John Gashnig. *Performance Measurements and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1979.

[2] Robert M. Haralick, Larry S. Davis, Azriel Rosenfeld, and David Milgram. Reduction operations for constraint satisfaction. *Information Sciences*, 14:199–219, 1978.

[3] Robert M. Haralick and Gordon Elliot. *Increasing Tree Search Efficiency for Constraint Satisfaction Problems.* Technical Report, Virginia Polytechnic Institute and State University, March 1979.

[4] Robert M. Haralick and Linda G. Shapiro. The consistent labelling problem: Part I. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, PAMI-1(2):173–184, April 1979.

[5] Robert M. Haralick and Linda G. Shapiro. The consistent labelling problem: Part II. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, PAMI-2(3):193–203, May 1980.

[6] T.C. Henderson and Ashok Samal. Multi-constraint shape analysis. *Image and Vision Computing*, 4(2):84–96, May 1986.

[7] Thomas C. Henderson. A note on discrete relaxation. *Computer Vision, Graphics And Image Processing*, 28:384–388, 1984.

[8] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

[9] Alan K. Mackworth. Consistency in network of relations. *Artificial Intelligence*, 8:99–118, 1977.

[10] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, March 1986.

[11] Ugo Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[12] Ashok Samal. Parallelism in node and arc consistency algorithms. 1986. Unpublished.

[13] David Waltz. *Understanding Line Drawings of Scenes with Shadows*, chapter 2, pages 19–92. McGraw-Hill Book Company, 1975.