Algorithms for Constraint Satisfaction Problems (CSPs)

by

Zhe Liu

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1998

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Many problems in AI can be modeled as constraint satisfaction problems (CSPs). Hence the development of effective solution techniques for CSPs is an important research problem. Forward checking (FC) with some other heuristics has been traditionally considered to be the best algorithm for solving CSPs while recently there have been a number of claims that maintaining arc consistency (MAC) is more efficient on large and hard CSPs. In this thesis, we provide a systematic comparison empirically of the performances of the MAC and FC algorithms on large and hard CSPs. In particular, we compare their performance with regard to the size, constraint density and constraint tightness of the problems. Though there is a trend that MAC eventually outperforms FC on hard problems as we increase the problem size, we found that the superiority of MAC over FC would not be revealed on the hard problems with low constraint tightness and high constraint density until the size of these problems is quite large. We also devised a new FC algorithm — FC4, which shows good performance on the hard problems with low constraint tightness and high constraint density.

Acknowledgements

This work was carried out under the supervision of Dr. Fahiem Bacchus. I am greatly indebted to Dr. Bacchus for giving me the opportunity and subsequently providing support and guidance. I would greatly appreciate all his patience, understanding, encouragement and inspiration.

Special acknowledgment is due to my readers Dr. Nick Cercone and Dr. Ian Munro for sparing their precious time to read my thesis and providing valuable comments and suggestions.

I would also like to thank Jean-Charles Regin of ILOG for providing his programs, and his assistance in my understanding his algorithms.

Contents

List of Figures

Chapter 1

Introduction

Many problems in AI can be modeled as constraint satisfaction problems (CSPs). Hence the development of effective solution techniques for CSPs is an important research problem. In this thesis, we first review some algorithms for CSP solving, and then provide an empirical analysis of some recently suggested techniques using randomly generated problems.

1.1 What is a CSP?

Examples and applications of CSPs can be found in many areas, such as resource allocation in scheduling, temporal reasoning, natural language processing, query optimization in database, etc.

In general, a CSP is a problem composed of a finite set of variables, each of which has a finite domain of values, and a set of constraints. Each constraint is defined over some subset of the original set of variables and restricts the values these variables can simultaneously take. The task is to find an assignment of a value for each variable such that the assignments satisfy all the constraints. In some problems, the goal is to find all such assignments.

A great many "real-world" problems can be formulated as CSPs. For example, we can take a look at the area of resource allocation. One application is examination scheduling. Examinations are to be scheduled in a number of given time slots. With a limited number of classrooms, each examination needs a classroom.

Different classrooms are of different capacity and an examination can only be scheduled in a classroom that has enough seats for students who are going to take that examination. Some students may take part in several examinations and these examinations cannot be scheduled in the same time slot. To model this problem, we can make each examination a variable, the possible time slots and classrooms are its domain, and the constraints are that certain examinations cannot be held together. A more complicated but more realistic example of resource allocation is airport gate allocation. Usually both the physical constraints (e.g., certain jet-ways can only accommodate certain types of aircraft) and user preferences (e.g., different airlines prefer to park in certain parts of an airport) need to be considered. This problem can be modeled as a CSP so that a solution to the CSP would be a solution to the airport gate allocation problem. In all areas of industry and business, resource allocation is a key factor to making a profit and a loss. Hence modeling these problems as CSPs to find an effective solution is an interesting research topic. In this thesis, we will focus on how to solve a given CSP.

Since modeling of realistic problems as CSPs is not the topic of this thesis, we illustrate the formalization of a CSP by two simple examples.

The N-queens problem can be modeled as a CSP. Given an integer N, the problem is to place N queens on N distinct squares in an $N \times N$ chess board, satisfying the constraint that no two queens should threaten each other. Two queens threaten each other if and only if they are on the same row, column or diagonal. Figure 1.1 gives one possible solution to the 4-queens problem.

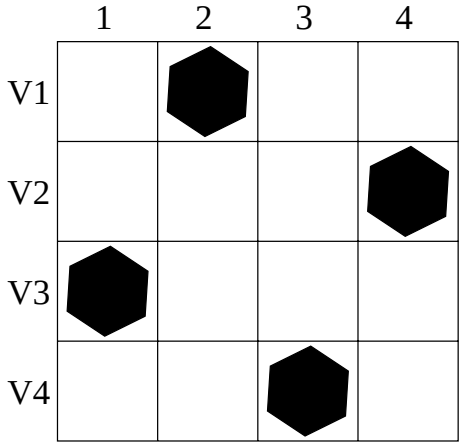

Figure 1.1: A possible solution to the 4-queens problem

Another example of a CSP is the map-coloring problem [6]. In this problem, we need to color each region of the map with one of a given set of colors such that no two adjacent regions have the same color. Figure 1.2 shows a simple map-coloring problem. The map has four regions that are to be colored red, green, or blue.

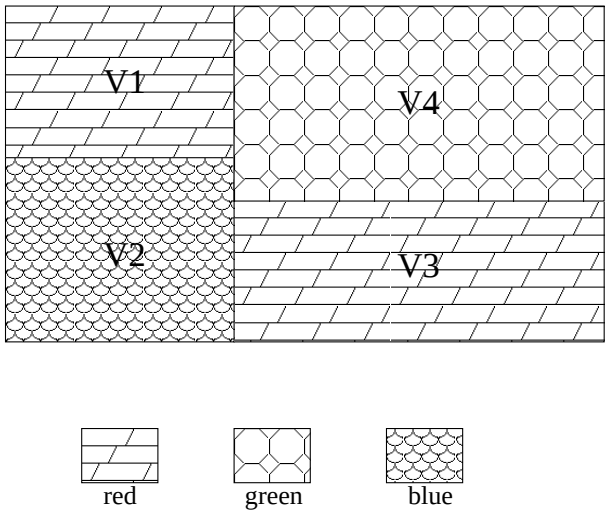

Figure 1.2: A map-coloring problem

A CSP is called an n-ary CSP when n is the maximum number of distinct variables over which a constraint is specified. For instance, a binary CSP only has constraints involving two variables. It is well known that any n-ary CSP can be converted to a binary CSP [8] [4] and in this thesis we restrict our attention to binary CSPs. Both the N-queens and map-coloring problems are binary CSPs.

More formally, a binary CSP can be defined to consist of a triple (V, D, R) where:

- V is a set $\{V_1, ..., V_i, ..., V_n\}$ of n variables;
- D is a set $\{D_1, ..., D_i, ..., D_n\}$ of domains, such that, $\forall i$, $1 \leq i \leq n$, $D_i = \{v_1^i, ..., v_j^i, ..., v_{k_i}^i\}$ is the finite set of possible values for V_i;

- R is a sequence $\{..., R_{ij}, ...\}$ of binary constraint relations such that $\forall R_{ij} \in R$, R_{ij} constrains the two variables V_i and V_j and is defined by a subset of the Cartesian product $D_i \times D_j$. The set of pairs of values in R specifies the allowed pairs of values for variables V_i and V_j. If $(v_l^i, v_m^j) \in R_{i,j}$, we say that the assignment $\{V_i \leftarrow v_l^i, V_j \leftarrow v_m^j\}$ is *consistent.*

Thus, we can encode the 4-queens problem as a CSP as follows:

- Make each of the 4 rows a variable: $V = \{V_1, V_2, V_3, V_4\}$.
  The value of each variable will represent the column in which the queen in row_i $(1 \leq i \leq 4)$ is placed;
- Domains: $D = \{D_1, D_2, D_3, D_4\}$
  Each of these 4 variables can take one of the 4 columns as its value, with labels 1 to 4. The domains of the 4 variables are:
  $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$;
- Constraints: $R = \{R_{ij} | i < j \text{ and } 1 \leq i, j \leq 4\}$
  For each constraint R_{ij}:
  1. No two queens on the same row: This constraint becomes trivial given the variable encoding;
  2. No two queens on the same column : $V_i \neq V_j$;
  3. No two queens on the same diagonal: $|i - j| \neq |V_i - V_j|$.

In a similar manner, the map-coloring problem in Figure 1.2 can be represented as a CSP as follows:

- Variables: V
  Each variable represents a region in the map: $V = \{V_1, V_2, V_3, V_4\}$.
  The value of each variable will represent the color assigned to that region;
- Domains: $D = \{D_1, D_2, D_3, D_4\}$
  The domain D_i of variable V_i will be the set of legal colors for $region_i (1 \leq i \leq 4)$. Assuming that we have three possible colors r (red), b (blue) and g (green) for each region, the domains of the four variables become:
  $D_1 = D_2 = D_3 = D_4 = \{r, g, b\}$;

- Constraints: $R = \{R_{12}, R_{14}, R_{23}, R_{24}, R_{34}\}$
  There is a constraint between two adjacent regions.
  For each constraint R_{ij}: $V_i \neq V_j$.

Associated with every binary CSP is a constraint graph. The constraint graph contains a node for each variable and an edge between each pair of nodes for which there is a constraint between the corresponding two variables.

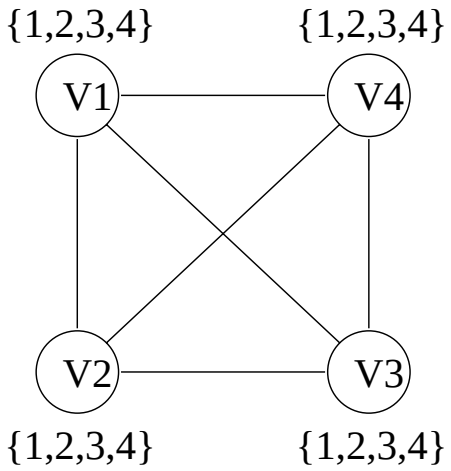

Figure 1.3: Graph representation of the 4-queens problem

Figure 1.3 is the constraint graph for the 4-queens problem which is a complete graph because there is a constraint for each pair of variables. Figure 1.4 shows the constraint graph for the map-coloring problem in Figure 1.2. Each edge represents two adjacent regions in the map.

Finding a solution to a CSP is an NP-complete problem. On one hand, we can guess the assignments of all variables and it is not difficult to check whether all the constraints are satisfied given these assignments. So it is in NP. On the other hand, the satisfiability problem (SAT), which is known to be NP-hard, can be encoded as a CSP problem.

Therefore, it is unlikely to have a polynomial time solution for CSPs. Nevertheless, there is great interest in finding algorithms for solving CSPs that perform well empirically.

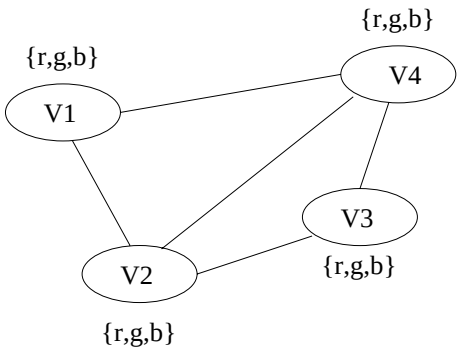

Figure 1.4: Graph representation of the map-coloring problem in Figure 1.2

1.2 General approaches for CSP solving

A most naive approach to solving a CSP is the "generate-and-test" method. Each possible assignment of values to variables is systematically generated and then tested to see if it satisfies all the constraints. The first assignment that satisfies all the constraints is the solution. In the worst case (or when we are trying to find all the solutions for a CSP), the number of assignments to be considered is the size of the Cartesian product of all the variable domains. Thus the time complexity of this approach is exponential in the number of variables. Empirically this method performs very poorly.

Randomized "generate-and-test" algorithms that select the assignments to test at random in accord with some biased distribution (e.g., the distribution might be biased by the most recently tested assignments as in randomized hill-climbing [11]) can sometimes perform extremely well, but unfortunately, lose systematicity. That is, these randomized methods are unable to prove that no solution exists since they do not necessarily test all assignments.

In general, there are three standard approaches for CSP solving.

1.2.1 Tree Search

Tree search is a standard technique for solving CSPs. The basic algorithm is simple backtracking (BT) [12], a general search strategy which has been widely used in problem solving. In solving CSPs, it also serves as the basis for many other algorithms.

In BT, variables are instantiated one by one. When a variable is instantiated, a value from its domain is picked and assigned to it. Then constraint checks are performed to make sure that the new instantiation is compatible with all the instantiations made so far. If all the completed constraints are satisfied, this variable has been instantiated successfully and we can move on to instantiate the next variable. If it violates certain constraints, then an alternative value, when available, is picked. If all the variables have a value, noting that all the assignments are consistent, the problem is solved. If at any stage no value can be assigned to a variable without violating a constraint, backtracking occurs. That means, the most recent instantiated variable has its instantiation revised and a new value, if available, is assigned to it. At this point we continue on to try instantiating the other variables, or we backtrack farther. This carries on until either a solution is found or all the combinations of instantiation have been tried and have failed which means that there is no solution.

Here is the BT algorithm specified in pseudo code:

```
01  function CONSISTENT(V_i, v_l^i)
        % Check against past variables
02      for each (V_j, v_m^j) ∈ Solution
03          if R_ij ∈ R and (v_l^i, v_m^j) ∉ R_ij
04              return FALSE
05      return TRUE
```

```
01 function SEARCH_BT(Vars, Level)
       % Try to instantiate V_i, then recurse
02     select a variable V_i ∈ Vars
03     for each value v_l^i ∈ D_i
04         if CONSISTENT(V_i, v_l^i)
05             Solution ← Solution + (V_i, v_l^i)
06             if V_i is the only variable in Vars
                   % Found a solution
07                 return TRUE
08             else
09                 if SEARCH_BT(Vars\{V_i}, Level + 1)
10                     return TRUE
       % No solution down this branch
11     return FALSE

01 function BT
02     Solution ← ∅
03     return SEARCH_BT(V, 1)
```

The CSP to be solved is given as (V, D, R) and we assume that if $R_{ij} \in R$, then we also have $R_{ji} \in R$. The BT algorithm is made up of three functions. BT is the main function. It calls $SEARCH\_BT(V, 1)$ to solve the problem. It returns $TRUE$ if it finds a solution and returns $FALSE$ if there is no solution. $SEARCH\_BT$ calls itself recursively to explore the search tree. Each invocation of $SEARCH\_BT$ corresponds to a node in the search tree, except the root. $SEARCH\_BT$ has two parameters. $Vars$ is a subset of V. It contains all the uninstantiated variables. $Level$ indicates the recursive level of the current invocation of the function $SEARCH\_BT$. $Level$ is 1 when it is first invoked by BT. Each invocation of $SEARCH\_BT$ tries to assign a value to an uninstantiated variable. At this point, we call the $(Level - 1)$ variables which have already been successfully instantiated the *past* variables, the variable currently being instantiated the *current* variable, and the remaining variables the *future* variables. If the assignment is successful,

it calls itself with increasing *Level* to search for a consistent assignment of the remaining variables. Otherwise, it tries the next value of the current variable. If all of the values of the current variable are tried and fail, the current invocation of $SEARCH\_BT$ exits and returns $FALSE$. This returns as to the previous invocation of $SEARCH\_BT$ at $Level - 1$, where the next value of the previous variable is tried. If the return is from the the first invocation of $SEARCH\_BT$, we return to the main function BT with value $FALSE$ and we know that no solution to the CSP exists. On the other hand, if all the variables are instantiated successfully, a solution is found and the current $SEARCH\_BT$ which has the deepest *Level* exits and returns $TRUE$. It will keep on going back to previous invocations of $SEARCH\_BT$ until it returns $TRUE$ to the main function BT. $CONSISTENT$ is called by $SEARCH\_BT$ to check whether the current instantiation is consistent with the instantiation of all the past variables. Only those constraints between the current variable and the past variables need to be checked. (Constraints that involve only past variables were checked in the previous stages and constraints that involve any future variables cannot be checked now because these variables have not yet been instantiated.) If any of the constraints fail, it returns $FALSE$. Otherwise, it returns $TRUE$. *Solution* is a global variable used to remember the current partial assignment. Initially it is empty and it will contain the solution in the end if one is found.

Figure 1.5 shows the tree searched by BT on the 4-queens problem. The node in a circle is the solution found by BT: $V_1 = 2, V_2 = 4, V_3 = 1, V_4 = 3$. The number of constraint checks performed is often used as a measurement of the efficiency of a CSP algorithm as this corresponds quite closely to the number of atomic operations performed by the algorithm. In the graph, the number beside each node indicates the number of constraint checks for that instantiation and total number of constraint checks and total number of nodes visited are also calculated.

BT is strictly better than generate-and-test in that it is able to eliminate a subspace from the Cartesian product of all the variable domains whenever a partial assignment of variables violates any of the constraints. However, since it essentially performs a depth-first search of the space of potential CSP solutions, its time complexity for most problems is still exponential.

Previous studies [6] have shown that there are two main reasons for the poor performance of BT: thrashing and redundant constraint checks. Some refined algorithms of BT have been developed to avoid these problems.

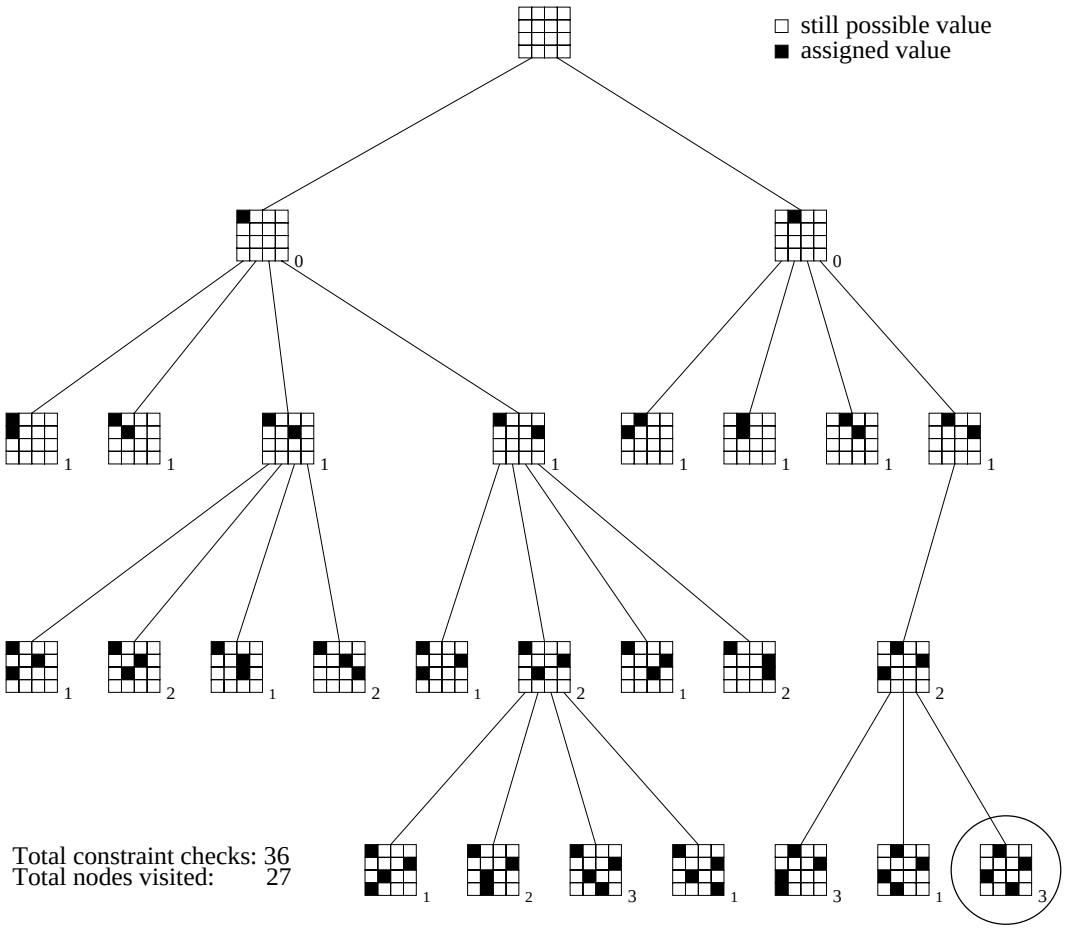

Figure 1.5: Search tree of the 4-queens problem using BT

BT suffers from thrashing. That is, search in different parts of the space keeps failing for the same reasons. Thrashing can be avoided by using some strategies so that backtracking is done directly to the variable that caused the failure. Backjumping (BJ) is an algorithm developed by Gaschnig [7] that jumps back multiple levels, directly to the cause of a conflict to avoid thrashing. Conflict-directed backjumping (CBJ) is an improvement of BJ that can perform multiple backjumps. In both algorithms, the number of nodes visited in the search tree can be reduced, resulting in a reduction in the number of constraint checks.

Sometimes BT has to perform redundant constraint checks. Backmarking (BM) also developed by Gaschnig [7] is aimed at eliminating redundant constraint checks by preventing the same constraint from being tested repeatedly.

All of these algorithms improve the performance of BT to a certain level but still cannot avoid the problem of thrashing and redundant constraint checks completely.

1.2.2 Constraint propagation

Constraint propagation is aimed at transforming a CSP into an equivalent problem that is hopefully easier to solve. Constraint propagation works by reducing the domain size of the variables in such a manner that no solutions are ruled out. It involves removing redundant values from the domains of the variables and propagating the effects of these removals throughout the constraints. Constraint propagation can be performed to different degrees.

As mentioned earlier, binary CSPs have associated constraint graphs, where the nodes represent variables and the edges binary constraints. Constraint propagation algorithms are best described in terms of these constraint graphs. Hence the related consistency concepts are named using terminology borrowed from graph theory.

As we know, binary CSPs have only two kinds of constraints: unary constraints and binary constraints. The simplest degree of consistency that can be enforced on a CSP is *node consistency* which concerns only the unary constraints. A CSP is node consistent if and only if for all variables all values in its domain satisfy the unary constraints on that variable. If a CSP is not node consistent, then there exists a certain variable V_i, and a certain value a in its domain such that value a does not satisfy the unary constraints on variable V_i. That means, the instantiation of V_i to a always results in an immediate failure. In other words, value a is redundant and will not be in any solution tuples. Hence it can be removed.

In this thesis, we assume that all our CSPs are already node consistent. If a variable has a value in its domain that does not satisfy the unary constraints on it, then that value is regarded to not be in its domain. All the values in the domains of all the variables have to satisfy the unary constraints on these variables.

A stronger degree of consistency is *arc consistency*. Arc consistency concerns the binary constraints in a CSP and considers binary constraints between one pair of variables at a time. An edge (V_i, V_j) in the constraint graph can be seen as a pair of arcs (V_i, V_j) and (V_j, V_i). We say arc (V_i, V_j) is arc consistent if and only if for every value a in the current domain of V_i, there exists some value b in the

domain of V_j such that $V_i = a$ and $V_j = b$ are permitted by the binary constraint between V_i and V_j. The concept of arc consistency is directional; that is, if an arc (V_i, V_j) is consistent, then it does not automatically mean that (V_j, V_i) is also arc consistent. For example, given the map-coloring problem in Figure 1.6, (V_1, V_3) is arc consistent, because b is the only value in the domain of V_1 and there exists a value g in the domain of V_3 that $V_1 = b$ and $V_3 = g$ are compatible. However, (V_3, V_1) is not arc consistent. For $V_3 = b$, $V_1 = b$ is not compatible with it and there is no other value in the domain of V_1.

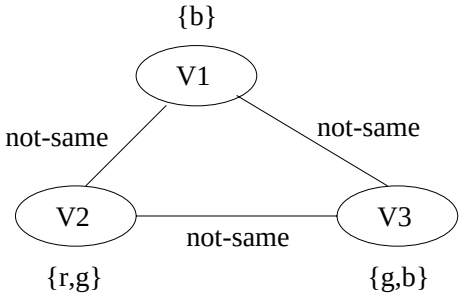

Figure 1.6: An example of arc consistency and inconsistency

An arc (V_i, V_j) can be made consistent by simply deleting those values from the domain of V_i for which there is no value in the domain of V_j compatible with them. Those values will not be in any solutions and therefore are redundant.

Function $REVISE(V_i, V_j)$ makes arc (V_i, V_j) consistent. It deletes all the values in the domain of V_i that are redundant and returns $TRUE$. Otherwise, if (V_i, V_j) is initially arc consistent and no value needs to be removed, it returns $FALSE$.

```
01 function REVISE(V_i, V_j)
02     Deleted ← FALSE
03     for each v_l^i ∈ D_i
04         Found ← FALSE
05         for each v_m^j ∈ D_j
06             if (v_l^i, v_m^j) ∈ R_{i,j}
07                 Found ← TRUE
08                 break
09         if not Found
10             D_i ← D_i − v_l^i
11             Deleted ← TRUE
12     return (Deleted)
```

A CSP is arc consistent if and only if every arc in the constraint graph is arc consistent. This can be done by executing $REVISE$ for each arc. But because the removal of some values of one domain may affect the consistency of other arcs and make more values redundant, it is not sufficient to execute $REVISE$ just once for each arc. Again consider the example in Figure 1.6, (V_2, V_3) is initially arc consistent. After arc (V_3, V_1) is made consistent by deleting b from the domain of V_3, arc (V_2, V_3) is no longer arc consistent and g in the domain of V_2 becomes redundant. In order to make the whole graph fully arc consistent, we have to propagate the removal of values throughout the graph until no more values can be removed.

A naive algorithm AC1 [12] for achieving arc consistency is given below:

```
01  function AC1
        % First part: initialization
02      Q ← ∅
03      for each variable V_i ∈ V
04          for each variable V_j ∈ V
05              if R_ij ∈ R
06                  Q ← Q ∪ (V_i, V_j)
        % Second part: examination and propagation
07      repeat
08          Changed ← FALSE
09          for each (V_i, V_j) ∈ Q
10              if REVISE(V_i, V_j)
11                  if D_i = ∅
12                      return FALSE
13                  Changed ← TRUE
14      until not (Changed)
15      return TRUE
```

Q is a list of arcs to be examined. For each constraint R_{ij} in the problem, both arc (V_i, V_j) and arc (V_j, V_i) are put into Q. $AC1$ examines every arc (V_i, V_j) in Q and calls $REVISE$ to delete from the domain of V_i all those values that do not satisfy R_{ij}. If any value is removed, all the arcs will be examined again. The loop will be repeated until no arc is revised.

Applying AC1 on the example in Figure 1.6, we get:

Arcs	REVISE		
	1st Iteration	2nd Iteration	3rd Iteration
(V_1, V_2)	FALSE	FALSE	FALSE
(V_1, V_3)	FALSE	FALSE	FALSE
(V_2, V_1)	FALSE	FALSE	FALSE
(V_2, V_3)	FALSE	TRUE (V_2, g) is removed	FALSE
(V_3, V_1)	TRUE (V_3, b) is removed	FALSE	FALSE
(V_3, V_2)	FALSE	FALSE	FALSE
Changed?	TRUE	TRUE	FALSE

The reduced problem which is made arc consistent is given in Figure 1.7. Clearly, we have a solution for this problem with $V_1 = b$, $V_2 = r$ and $V_3 = g$.

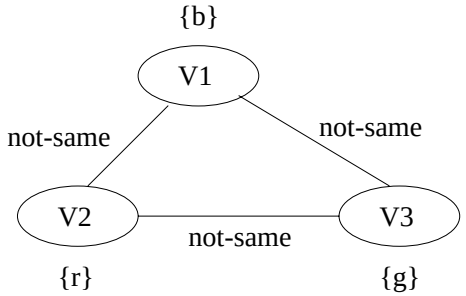

Figure 1.7: After obtaining arc consistency for the example in Figure 1.6

AC1 could be very inefficient because the removal of any value from any domain causes all the elements of Q to be re-examined. A simple examining shows that the removal of a value will not affect all the arcs and only the possibly affected arcs need to be re-examined. In the above example, the removal of (V_2, g) has no effect on arc (V_1, V_3), thus arc (V_1, V_3) need not to be re-examined. An improved

algorithm AC3 [12] is given below:

```
01  function AC3
        % First part: initialization
02      Q ← ∅
03      for each variable V_i ∈ V
04          for each variable V_j ∈ V
05              if R_ij ∈ R
06                  Q ← Q ∪ (V_i, V_j)
        % Second part: examination and propagation
07      while Q ≠ ∅
08          select and remove (V_i, V_j) from Q
09          if REVISE(V_i, V_j)
10              if D_i = ∅
11                  return FALSE
12              else
13                  for each variable V_k ∈ V such that k ≠ j
14                      if R_ki ∈ R
15                          Q ← Q ∪ (V_k, V_i)
16      return TRUE
```

In this algorithm, if $REVISE(V_i, V_j)$ removes any value from the domain of V_i, then only the domain of any variable V_k that is constrained with V_i has to be re-examined. This is because the removed value in the domain of V_i may be the only one that is compatible with some value of V_k. Therefore arcs (V_k, V_i) are added to Q to be re-examined for all k such that there is an arc from V_k to V_i. However, arc (V_j, V_i) does not need to be re-examined as the removed values of V_i have no compatible values in the domains of V_j. In fact, this is the reason that they are removed. Their removal will not cause any values of V_j to lose compatible value in the domain of V_i. It is clear that we do not need to worry about the domains of other variables that are not constrained with V_i.

Applying AC3 on the example in Figure 1.6, we get:

Arcs	REVISE
(V_1, V_2)	FALSE
(V_1, V_3)	FALSE
(V_2, V_1)	FALSE
(V_2, V_3)	FALSE
(V_3, V_1)	TRUE (V_3, b) is removed Arcs (V_1, V_3), (V_2, V_3) are added
(V_3, V_2)	FALSE
(V_1, V_3)	FALSE
(V_2, V_3)	TRUE (V_2, g) is removed Arcs (V_1, V_2), (V_3, V_2) are added
(V_1, V_2)	FALSE
(V_3, V_2)	FALSE

We have the same result as using AC1. However, the number of arcs we need to check for revising is reduced.

In general, achieving arc consistency alone rarely generates solutions except for three special cases:

1. If any of the domains is wiped out during the execution, then no solution exists.
2. If the domain size of each variable becomes exactly one after obtaining arc consistency, then there is one solution.
3. If the domain size of $N-1$ variables becomes one (N is the total number of variables) and the other domain has $k(>1)$ values, then there are k solutions.

We have already seen an example of case 2. Figure 1.8 and Figure 1.9 are examples of case 1 and case 3 respectively.

On the other hand, in some cases, even after obtaining arc consistency, we still do not know the solution(s). Figure 1.10, Figure 1.11 and Figure 1.12 show three

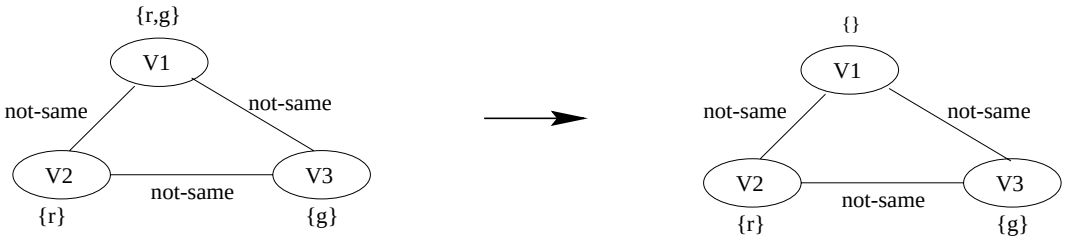

Figure 1.8: No solution after obtaining arc consistency

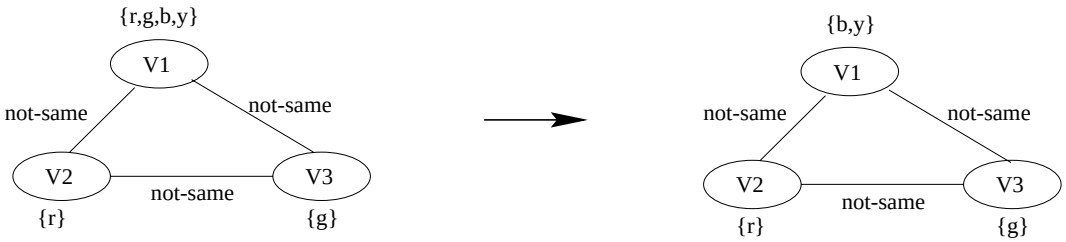

Figure 1.9: Two solutions after obtaining arc consistency: (b, r, g) and (y, r, g)

examples that are all arc consistent but have no solution, one solution and two solutions respectively.

The 4-queens problem we discussed earlier is initially arc consistent, so the arc consistency algorithms have no effect.

Besides node consistency and arc consistency, there are even stronger degrees of consistency. In general, we say a constraint graph is K-*consistent* if the following is true: Choose values of any $(K-1)$ variables that satisfy all the constraints among these variables, then choose any K'th variable. A value for this variable exists that satisfies all of the constraints among these K variables. Furthermore, we can achieve K-consistency by pruning away any values of the K's variable that fail to satisfy this condition (doing the pruning in a repeated manner as when achieving arc consistency). A constraint graph is strongly K consistent if it is J consistent for all $J \leq K$. Node consistency and arc consistency are actually equivalent to strong 1-consistency and strong 2-consistency respectively.

An N-variable CSP can be solved by achieving N-consistency. However, this

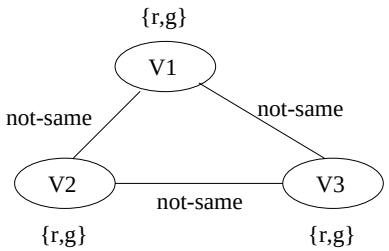

Figure 1.10: No solution

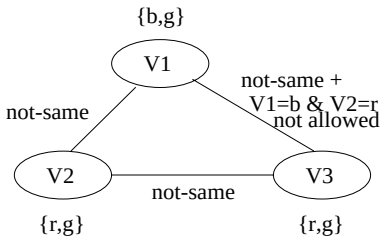

Figure 1.11: One solution: (b, r, g)

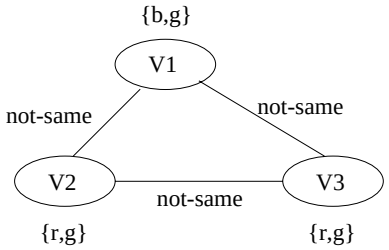

Figure 1.12: Two solutions: (b, r, g) and (b, g, r)

approach is seldom used due to its high cost (considerably more expensive than simple backtracking). Usually it is only useful to achieve arc consistency when performing constraint propagation. As pointed out above, however, this does not guarantee we will find the solution(s). To find solution(s) when employing arc consistency we must combine constraint propagation with backtracking search.

1.2.3 Combining backtracking search and constraint propagation

In the previous two sections, two rather different approaches are discussed for solving CSPs: backtracking search and constraint propagation. Backtracking search guarantees that a solution will be found if one exists, but it suffers from thrashing and redundant constraint checks. Constraint propagation may simplify the problem, but such simplification is usually insufficient to actually solve the problem. A third approach is to combine these two approaches by embedding a constraint propagation algorithm inside a backtracking search algorithm.

The basic idea is as follows. In the search tree of the backtracking algorithm, whenever a node is visited, a constraint propagation algorithm is performed to attain a desired level of consistency by removing inconsistent values from the domains of the as yet uninstantiated variables. If in the process of constraint propagation at the node, the domain of any variable becomes empty, then the node is pruned. The purpose of doing this is to detect a "dead end" as early as possible. This way, potential thrashing can be reduced, and the size of the search tree is reduced.

This approach turns out to be very effective and quite a few important CSP algorithms, such as forward checking (FC) and maintaining arc consistency (MAC) are in fact of this type. FC with some other heuristics has been traditionally considered to be the best algorithm for solving CSPs while recently there have been a number of claims that MAC is more efficient on large and hard CSPs. The difference between them lies in the extent of constraint propagation each algorithm performs. In FC, only partial arc consistency is achieved at each node during search while in MAC, full arc consistency is guaranteed. Furthermore, in MAC, the arc consistency algorithm is also performed as preprocessing before search.

Pseudo code for the FC (called FC3 [1]) and MAC (called MAC3 [1]) algorithms is given below. These two functions are used in both the FC3 and MAC3 algorithms:

```
01 function DWO(V_i)
02     for each value v_l^i ∈ D_i
03         if Domain_l^i is not marked
04             return TRUE
05     return FALSE
```

```
01 function RESTORE(Vars, Level)
       % Restore Domain to previous state
02     for each variable V_i ∈ Vars
03         for each value v_l^i ∈ D_i
04             if Domain_l^i is marked at Level
05                 Domain_l^i ← unMarked
```

These functions are used in the FC3 algorithm:

```
01 function CHECK_FORWARD3(Vars, Level, V_i, v_l^i)
02   for each variable V_j ∈ Vars
03     if R_ij ∈ R
04       for each value v_m^j ∈ D_j such that
           Domain_m^j is not marked
05         if (v_l^i, v_m^j) ∉ R_ij
06           Domain_m^j ← Marked at Level
07       if DWO(V_j)
08         return FALSE
09   return TRUE
```

[1] To make them distinct from other improved FC and MAC algorithms which we will discuss later, we call them FC3 and MAC3 respectively.

```
01 function SEARCH_FC3(Vars, Level)
02     select a variable V_i ∈ Vars
03     for each value v_l^i ∈ D_i such that Domain_l^i is not marked
04         Solution ← Solution + (V_i, v_l^i)
05         if V_i is the only variable in Vars
               % Found a solution
06             return TRUE
07         else
               % Try to achieve partial arc-consistency
08             if CHECK_FORWARD3(Vars\{V_i}, Level, V_i, v_l^i)
                   and SEARCH_FC3(Vars\{V_i}, Level + 1)
09                 return TRUE
10             else
11                 Solution ← Solution − (V_i, v_l^i)
12                 RESTORE(Vars\{V_i}, Level)
       % No solution down this branch
13     return FALSE
```

```
01 function FC3
02     for each variable V_i ∈ V
03         for each value v_l^i ∈ D_i
04             Domain_l^i ← unMarked
05     Solution ← ∅
06     return SEARCH_FC3(V, 1)
```

These functions are used in the MAC3 algorithm (assuming AC3 is the algorithm used to maintain arc consistency):

```
01 function REVISE(V_i, V_j, Level)
02     Deleted ← FALSE
03     for each v_l^i ∈ D_i such that Domain_l^i is not marked
04         Found ← FALSE
05         for each v_m^j ∈ D_j such that Domain_m^j is not marked
06             if (v_l^i, v_m^j) ∈ R_ij
07                 Found ← TRUE
08                 break
09         if not Found
10             Domain_l^i ← Marked at Level
11             Deleted ← TRUE
12     return (Deleted)
```

```
01 function PROPAGATE_AC3(Vars, Level)
02     while Q ≠ ∅
03         select and remove (V_i, V_j) from Q
04         if REVISE(V_i, V_j, Level)
05             if DWO(V_i)
06                 return FALSE
07             else
08                 for each variable V_k ∈ Vars such that k ≠ j
09                     if R_ki ∈ R
10                         Q ← Q ∪ (V_k, V_i)
11     return TRUE
```

```
01 function SEARCH_MAC3(Vars, Level)
02     select a variable V_i ∈ Vars
03     for each value v_l^i ∈ D_i such that Domain_l^i is not marked
04         Solution ← Solution + (V_i, v_l^i)
05         if V_i is the only variable in Vars
               % Found a solution
06             return TRUE
07         else
               % Eliminate all the other values of V_i
08             for each value v_l'^i ∈ D_i\{v_l^i} such that Domain_l'^i
                   is not marked
09                 Domain_l'^i ← Marked at Level
10             for each variable V_j ∈ Vars
11                 if R_ji ∈ R
12                     Q ← Q ∪ (V_j, V_i)
13             if PROPAGATE_AC3(Vars\{V_i}, Level) and
                   SEARCH_MAC3(Vars\{V_i}, Level + 1)
14                 return TURE
15             else
16                 Solution ← Solution − (V_i, v_l^i)
17                 RESTORE(Vars, Level)
       % No solution down this branch
18     return FALSE
```

```
01 function MAC3
02     for each variable V_i ∈ V
03         for each value v_l^i ∈ D_i
04             Domain_l^i ← unMarked
05     Solution ← ∅
06     Q ← ∅
07     for each variable V_i ∈ V
08         for each variable V_j ∈ V
09             if R_ij ∈ R
10                 Q ← Q ∪ (V_i, V_j)
11     if PROPAGATE_AC3(V, 0)
12         return SEARCH_MAC3(V, 1)
13     else
           % No solution due to arc inconsistency
14         return FALSE
```

Since we use BT as the algorithm for backtracking search and embed a constraint propagation algorithm inside it, the frameworks of the FC and MAC algorithms are similar to that of BT. All of the three algorithms are made up of three parts: the main routine, the search routine and the constraint satisfaction routines. The corresponding functions in the three algorithms are:

Algorithm	Main Routine	Search Routine	Constraint Satisfaction Routine
BT	*BT*	*SEARCH_BT*	*CONSISTENT*
FC	*FC3*	*SEARCH_FC3*	*CHECK_FORWARD3*
MAC	*MAC3*	*SEARCH_MAC3*	*PROPAGATE_AC3*

The main routine makes the necessary initializations and starts the search. The search routine explores the search tree. It tries to instantiate one of the variables and determines the success of the instantiation by employing the constraint satisfaction routine: if the instantiation is successful, it calls itself recursively to search further; otherwise, it backtracks. The constraint satisfaction routine maintains the

constraints at each node of the search tree in a certain way according to different algorithms. In fact this is the part where the constraint propagation algorithms are embedded for the FC and MAC algorithms respectively.

The constraint satisfaction parts of the FC and MAC algorithms are different from that of the BT algorithm. Instead of checking the constraints between current variable and all past variables which *CONSISTENT* (in BT) does, in FC and MAC, constraints are checked forward against future variables. In FC, partial arc consistency is achieved in such a way that all values incompatible with the current instantiation are removed from the domains of the future variables. In other words, each future variable is made arc consistent with the current variable. In MAC, full arc consistency is achieved among the newly instantiated variable and all future variables.

When a variable is going to be instantiated, both the FC and MAC algorithms guarantee that any value remaining in its domain is compatible with all the instantiations made so far, no backward consistency check is required. However, *RESTORE* has to be performed to regain the previous state when an instantiation fails. An instantiation fails if the domain of a future variable becomes empty (called domain wipe-out or DWO) as a result of constraint propagation.

In order to restore those values pruned due to constraint propagation after a certain instantiation once that instantiation fails, both the FC and MAC algorithms use an extra data structure *Domain* which is a two dimensional array that keeps track of the status of all the values in the original domains of the variables. Initially all entries of *Domain* are *unMarked* which means all the values in the domains are possible in some solutions. Later on, whenever a value is removed from its domain, the relevant entry of *Domain* will be labeled as *Marked*. Moreover, we can mark it using different flags (e.g., numbers) according to the level at which it is pruned. By this way, when we need to perform *RESTORE*, we can determine whether a pruned value is removed due to the current instantiation and should be put back to its domain, by the flag of its relevant entry in *Domain*. A utility function *DWO* is also given to check whether all the values in a domain are marked — in which case a domain wipe-out occurs.

Since we use AC3 as the algorithm to achieve arc consistency in MAC3, relevant code is borrowed from AC3 with some minor changes (mainly to make it work with *Domain*). Line 6-10 in function *MAC3* corresponds to the first part of *AC3* in AC3,

which initializes Q. Function $PROPAGATE\_AC3$ corresponds to the second part of function $AC3$ in AC3. It is used to restore full arc consistency. It is called not only in function $SEARCH\_MAC3$, but also in the main function $MAC3$ to make sure that the problem is made arc consistent before search starts. Line 10-12 in function $SEARCH\_MAC3$ initializes Q for arc consistency propagation when the current variable is revised, since all the other values are removed except the assigned one.

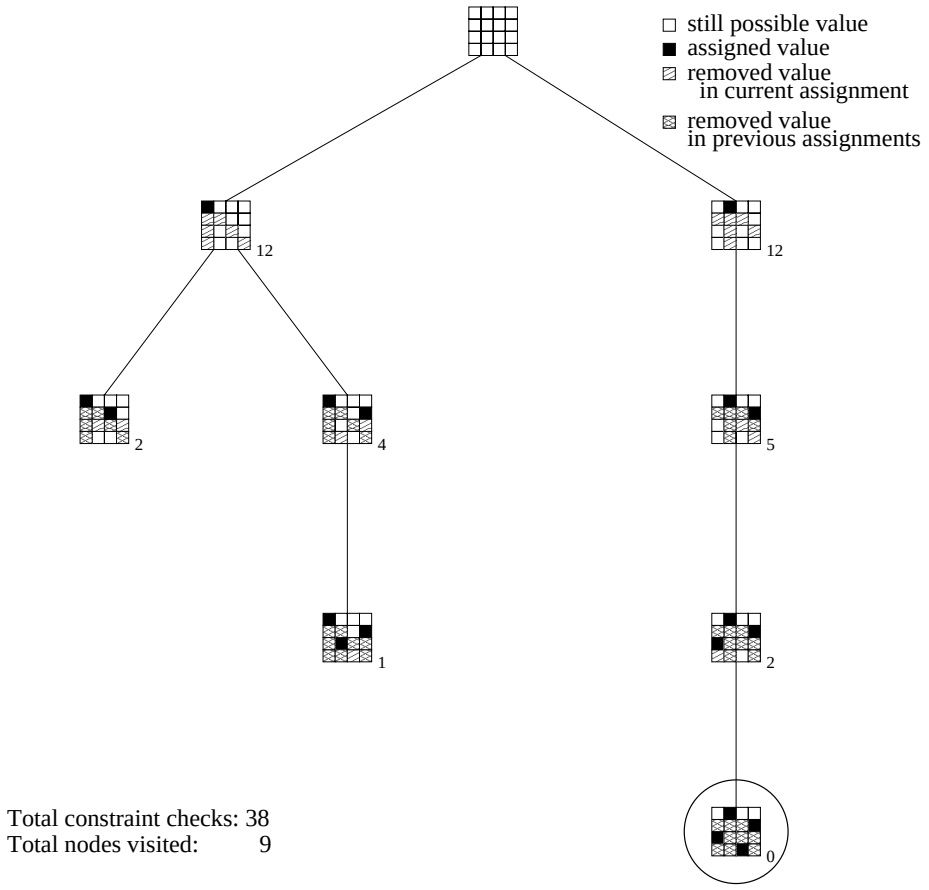

Figure 1.13: Search tree of the 4-queens problem using FC

We have seen the performance of applying BT on the 4-queens problem and as we have pointed out, since it is initially arc consistent, arc consistency algorithms cannot solve it. Figure 1.13 and Figure 1.14 show solving the 4-queens problem using FC and MAC.

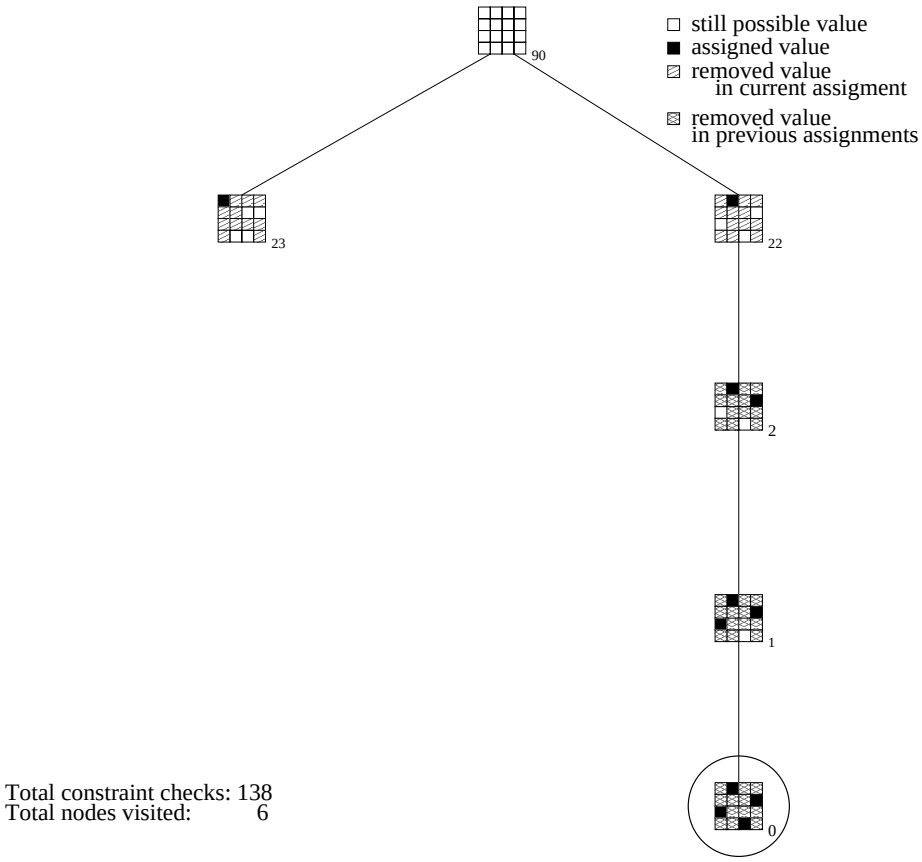

Figure 1.14: Search tree of the 4-queens problem using MAC

The following table is a summary of the performance of the three algorithms.

Algorithm	Constraint Checks	Nodes Visited	Checks per Node
BT	36	27	1.33
FC	38	9	4.22
MAC	138	6	23.00

From this table, we see that FC visits more nodes while MAC does more work at each node. In fact, it is not hard to prove that these conditions hold in general. In particular, if FC detects a future variable with an empty domain, or a DWO, MAC will also detect a DWO, and may in fact detect a DWO at some parent of the node.

1.3 Summary

Three classes of algorithms have been discussed. In practice, there is considerable evidence that the third approach which combines backtracking search and constraint propagation is the most effective solution technique for CSPs.

The question now is how much constraint propagation need to do at each node. More constraint propagation at each node will result in the search tree containing fewer nodes but the overall effort can be higher because the processing at each node will be more expensive. There is a trade-off between the number of nodes visited and the work at each node. The rest of this thesis will concentrate on this question and will address the specific question of comparing the FC and MAC algorithms. In Chapter 2, we will review some recently suggested improved implementations of the MAC algorithm. In Chapter 3, we will propose an improved implementation of the FC algorithm. Empirical results of comparing FC and MAC are given in Chapter 4. The last chapter is on conclusions and future work.

Chapter 2

Previous Work on MAC

Recently, claims have appeared in the literature that MAC is the most efficient general CSP algorithm for solving large and hard problems [3] [9]. To support this claim, empirical evidence is supplied using improved implementations of MAC. These improved versions of MAC rely on improved arc consistency algorithms. In this chapter, we review several of these improved arc consistency algorithms and discuss their implementations in MAC.

2.1 Improved arc consistency algorithms

2.1.1 AC4

AC4 [12] is an arc consistency algorithm that improves on AC3. AC4 is based on the notion of *support*. Let a be a value in the domain of V_i, and V_j be a variable constrained with V_i. Value a is *supported* by V_j if there is at least one value b in the domain of V_j such that $V_i = a$ and $V_j = b$ are compatible. Clearly, if each value in the domain of V_i is supported by V_j, then arc (V_i, V_j) is consistent. Values that are not supported are redundant and can be removed.

AC4 keeps track of *support* explicitly, by maintaining two additional data structures.

1. For each value of every variable there is a *Counter* for each arc starting from that variable representing the number of values in the domain of the variable at the other end of the arc that this value is compatible with.

For example, $Counter[(V_i, V_j), a]$ represents the number of values in the domain of V_j supporting the assignment $V_i = a$.

Whenever a *Counter* for some assignment becomes zero, that domain value can be removed.

2. For each value of every variable, there is a support set S, that contains all of the variable-value pairs that it supports.

   For example, for value a in the domain of V_i, a set

   $$S_{V_i a} = \{(V_j, b) | V_i = a \ supports \ V_j = b\}$$

   is constructed. This data structure will be used to update *Counter* efficiently.

Again consider the example in Figure 1.6. The domain of V_3 has two values, g and b. For V_3, two support sets are constructed:

$$S_{V_3 g} = \{(V_1, b), (V_2, r)\}$$
$$S_{V_3 b} = \{(V_2, r), (V_2, g)\}$$

The support set $S_{V_3 g}$ records the fact that the value g in the domain of variable V_3 supports the assignment $V_1 = b$ and $V_2 = r$. This set helps to identify those assignments that need to be re-examined should the value g be removed from the domain of V_3. In this case, only b of V_1 and r of V_2 need to be considered. No other value will be affected by the removal of g from the domain of V_3.

Similarly, the support sets for V_1 and V_2 are:

$$S_{V_1 b} = \{(V_2, r), (V_2, g), (V_3, g)\}$$
$$S_{V_2 r} = \{(V_1, b), (V_3, g), (V_3, b)\}$$
$$S_{V_2 g} = \{(V_1, b), (V_3, b)\}.$$

A *Counter* is maintained for each arc-value pair. For variable V_1, there are two arcs starting from it: arc (V_1, V_2) and arc (V_1, V_3). For value b in the domain of variable V_1, the arc (V_1, V_2) provides two supports from V_2 — namely $V_2 = r$ and $V_2 = g$; the arc (V_1, V_3) provides one supports from V_3 — $V_3 = g$. Therefore,

$$Counter[(V_1, V_2), b] = 2$$
$$Counter[(V_1, V_3), b] = 1.$$

In general, $Counter[(V_i, V_j), a]$ records the number of values in the domain of V_j supporting the assignment $V_i = a$.

According to this principle, other $Counter$s will be initialized to be the following values:

$$Counter[(V_2, V_1), r] = 1$$
$$Counter[(V_2, V_3), r] = 2$$
$$Counter[(V_2, V_1), g] = 1$$
$$Counter[(V_2, V_3), g] = 1$$
$$Counter[(V_3, V_1), g] = 1$$
$$Counter[(V_3, V_2), g] = 1$$
$$Counter[(V_3, V_1), b] = 0$$
$$Counter[(V_3, V_2), b] = 2$$

We can notice that one of the $Counter$s is zero. As we mentioned earlier, if $Counter[(V_i, V_j), a]$ is zero, value a can be removed from its domain because there is no value in the domain of V_j that supports it. This is equivalent to our definition of arc consistency: (V_i, V_j) is not arc consistent because of the presence of a. Therefore, a has to be removed. The removal of a will also affect the consistency of other arcs. In the context of AC4, it will cause some counters to be updated. For example, since $Counter[(V_3, V_1), b] = 0$, we remove b from V_3. But remember that b in V_3 is one of the two components that make $Counter[(V_2, V_3), r] = 2$, now $Counter[(V_2, V_3), r]$ has to be decremented by 1. In fact, support set $S_{V_3 b}$ keeps track of all the counters that need to be updated. In general, when a value b is removed from the domain of variable V_j, $Counter[(V_i, V_j), a]$ must be decremented for each (V_i, a) in $S_{V_j b}$. If more counters become zero, more values need to be removed and more removals need to be processed. Just like other arc consistency algorithms, this carries on until no more values are removed and the remaining reduced problem is arc consistent.

The algorithm of AC4 has two parts. In the first part, support sets are constructed and counters are initialized; redundant domain values are removed. In the

second part, the value removals are processed to update the relevant counters. This may generate an additional set of redundant values which then have to be removed.

The pseudo code for AC4 is shown below:

```
01 function AC4
   % First part: construction and initialization of
   % support sets: S and Counter
02    for each variable V_i ∈ V
03       for each value v_l^i ∈ D_i
04          S_{V_i v_l^i} ← ∅
05    DeletionStream ← ∅
06    for each variable V_i ∈ V
07       for each variable V_j ∈ V
08          if R_ij ∈ R
09             for each value v_l^i ∈ D_i
10                total ← 0
11                for each value v_m^j ∈ D_j
12                   if (v_l^i, v_m^j) ∈ R_ij
13                      total ← total + 1
14                      S_{V_j v_m^j} ← S_{V_j v_m^j} + (V_i, v_l^i)
15                if total = 0
16                   D_i ← D_i − v_l^i
17                   DeletionStream ← DeletionStream + (V_i, v_l^i)
18                   if D_i = ∅
19                      return FALSE
20                else
21                   Counter[(V_i, V_j), v_l^i] ← total
```

```
    % Second part: propagation of removed values
22  while DeletionStream ≠ ∅
23      select and remove (V_i, v_l^i) from DeletionStream
24      for each (V_j, v_m^j) ∈ S_{V_i v_l^i}
25          Counter[(V_j, V_i), v_m^j] ← Counter[(V_j, V_i), v_m^j] − 1
26          if Counter[(V_j, V_i), v_m^j] = 0 and v_m^j ∈ D_j
27              D_j ← D_j − v_m^j
28              DeletionStream ← DeletionStream + (V_j, v_m^j)
29              if D_j = ∅
30                  return FALSE
31  return TRUE
```

In the above algorithm, *DeletionStream* is a list to maintain all variable-value pairs (V_i, a), where value a has been removed from domain D_i, but the effect of the removal has not yet been propagated. Initially, it contains the values removed in the first part. If there are more counters zeroed during the propagation, more values are added to *DeletionStream*. The process terminates when no more elements remain in *DeletionStream*.

Applying AC4 on the example in Figure 1.6, we get:

1. After initialization:

Arc		Value	Counter	Support Set
V_1	V_2	b	2	$(V_2, r), (V_2, g)$
	V_3		1	(V_3, g)
V_2	V_1	r	1	(V_1, b)
	V_3		2	(V_3, g)
V_2	V_1	g	1	(V_1, b)
	V_3		1	
V_3	V_1	g	1	(V_1, b)
	V_2		1	(V_2, r)
V_3	V_1	b	0	
	V_2		-	$(V_2, r), (V_2, g)$

DeletionStream: (V_3, b).

2. Process the deletion of (V_3, b)

Traverse its support set: $\{(V_2, r), (V_2, g)\}$, we need to check whether (V_2, r) and (V_2, g) still have support in the domain of V_3:

$$Counter[(V_2, V_3), r] = Counter[(V_2, V_3), r] - 1 = 1$$

$$Counter[(V_2, V_3), g] = Counter[(V_2, V_3), g] - 1 = 0$$

Arc		Value	Counter	Support Set
V_1	V_2	b	2	$(V_2, r), (V_2, g)$
	V_3		1	(V_3, g)
V_2	V_1	r	1	(V_1, b)
	V_3		1	(V_3, g)
V_2	V_1	g	1	(V_1, b)
	V_3		0	
V_3	V_1	g	1	(V_1, b)
	V_2		1	(V_2, r)
V_3	V_1	*b	0	
	V_2		-	$(V_2, r), (V_2, g)$

* removed value

DeletionStream: (V_2, g).

3. Process the deletion of (V_2, g)

   Traverse its support set: $\{(V_1, b)\}$, we need to check whether (V_1, b) still has support in the domain of V_2:

   $$Counter[(V_1, V_2), b] = Counter[(V_1, V_2), b] - 1 = 1$$

Arc		Value	Counter	Support Set
V_1	V_2	b	1	$(V_2, r), (V_2, g)$
	V_3		1	(V_3, g)
V_2	V_1	r	1	(V_1, b)
	V_3		1	(V_3, g)
V_2	V_1	*g	1	(V_1, b)
	V_3		0	
V_3	V_1	g	1	(V_1, b)
	V_2		1	(V_2, r)
V_3	V_1	*b	0	
	V_2		-	$(V_2, r), (V_2, g)$

   * removed value

   DeletionStream: $\emptyset$.

In AC3, when we process the deletion of a value, say, (V_3, b), we have to re-examine arcs (V_1, V_3) and (V_2, V_3). In more detail, function $REVISE$ is called twice with parameters (V_1, V_3) and (V_2, V_3) respectively and each value in the domain of V_1 and V_2 is checked to see whether we can still find some compatible value in the domain of V_3 after b is deleted. In other words, we have to seek a support for all the values in the domains of V_1 and V_2 again. However, in AC4, we only need to check those variable-value pairs in (V_3, b)'s support set, in this case, only (V_2, r) and (V_2, g). We do not need to worry about (V_1, b) since (V_3, b) is not a support for it (in fact, they are not compatible). Moreover, we only need to decrement the relevant $Counter$s by 1 and check whether any of them becomes zero instead of explicitly traversing the values in the domain of V_3 one by one until we find a support.

2.1.2 AC6

AC4 can be further improved. In AC4, for each value a in variable V_i and each arc (V_i, V_j), all the values in V_j are checked to compute $Counter[(V_i, V_j), a]$. This is more work than necessary to show that $V_i = a$ is *supported* by V_j. *Support* is guaranteed as long as $Counter[(V_i, V_j), a] > 0$. Hence, one support in V_j is enough. Therefore, instead of maintaining $Counter[(V_i, V_j), a]$ and decrementing it whenever a support in V_j is removed, we can find one support in V_j first and try to look for another one only when necessary, i.e., when the current support is removed from the domain of V_j. Based on this idea, Bessiere proposed the algorithm AC6 in [1].

The pseudo code for AC6 is shown below:

```
01 function SEEK_SUPPORT(V_i, V_j, v_l^i, LastInd)
02     for each value v_m^j ∈ D_j
03         if m ≤ LastInd
04             continue
05         else
06             if (v_l^i, v_m^j) ∈ R_ij
07                 S_{V_j v_m^j} ← S_{V_j v_m^j} + (V_i, v_l^i)
08                 return TRUE
09     return FALSE
```

```
01 function AC6
   % First part: construction and initialization of
   % support sets: S
02    for each variable V_i ∈ V
03       for each value v_l^i ∈ D_i
04          S_{V_i v_l^i} ← ∅
05    DeletionStream ← ∅
06    for each variable V_i ∈ V
07       for each variable V_j ∈ V
08          if R_ij ∈ R
09             for each value v_l^i ∈ D_i
10                if not SEEK_SUPPORT(V_i, V_j, v_l^i, 0)
11                   D_i ← D_i − v_l^i
12                   DeletionStream ← DeletionStream + (V_i, v_l^i)
13                   if D_i = ∅
14                      return FALSE
   % Second part: propagation of removed values
15    while DeletionStream ≠ ∅
16       select and remove (V_j, v_m^j) from DeletionStream
17       for each (V_i, v_l^i) ∈ S_{V_j v_m^j}
18          if v_l^i ∈ D_i
19             if not SEEK_SUPPORT(V_i, V_j, v_l^i, m)
20                D_i ← D_i − v_l^i
21                DeletionStream ← DeletionStream + (V_i, v_l^i)
22                if D_i = ∅
23                   return FALSE
24    return TRUE
```

Applying AC6 on the example in Figure 1.6, we get:

1. After initialization:

Arc		Value	Supported?	Support Set
V_1	V_2	b	TRUE	$(V_2, r), (V_2, g)$
	V_3		TRUE	(V_3, g)
V_2	V_1	r	TRUE	(V_1, b)
	V_3		TRUE	(V_3, g)
V_2	V_1	g	TRUE	
	V_3		TRUE	
V_3	V_1	g	TRUE	(V_1, b)
	V_2		TRUE	(V_2, r)
V_3	V_1	b	FALSE	
	V_2		-	(V_2, g)

DeletionStream: (V_3, b).

2. Process the deletion of (V_3, b)

Traverse its support set: $\{(V_2, g)\}$, we need to check whether (V_2, g) still has support in the domain of V_3:

$SEEK\_SUPPORT(V_2, V_3, g)$: $FALSE$

Arc		Value	Supported?	Support Set
V_1	V_2	b	TRUE	$(V_2, r), (V_2, g)$
	V_3		TRUE	(V_3, g)
V_2	V_1	r	TRUE	(V_1, b)
	V_3		TRUE	(V_3, g)
V_2	V_1	g	TRUE	
	V_3		FALSE	
V_3	V_1	g	TRUE	(V_1, b)
	V_2		TRUE	(V_2, r)
V_3	V_1	*b	FALSE	
	V_2		-	(V_2, g)

* removed value

DeletionStream: (V_2, g).

3. Process the deletion of (V_2, g)

   Traverse its support set: $\emptyset$ — nothing need to do

Arc		Value	Supported?	Support Set
V_1	V_2	b	TRUE	$(V_2, r), (V_2, g)$
	V_3		TRUE	(V_3, g)
V_2	V_1	r	TRUE	(V_1, b)
	V_3		TRUE	(V_3, g)
V_2	V_1	*g	TRUE	
	V_3		FALSE	
V_3	V_1	g	TRUE	(V_1, b)
	V_2		TRUE	(V_2, r)
V_3	V_1	*b	FALSE	
	V_2		-	(V_2, g)

   * removed value

   DeletionStream: $\emptyset$.

Compared with AC4, the gain of AC6 is two-fold:

- We can hopefully find a solution (or prove that there is no solution) before the full support sets (which is used in AC4) are setup. Some constraint checks can be saved this way.

  In the above example, we never check the compatibility of (V_2, r) and (V_3, b), because we already know that (V_3, g) is a support for (V_2, r) on arc (V_2, V_3) and this is enough to make (V_2, V_3) arc consistent with regard to (V_2, r).

- Because for each value and each arc there is at most one support, the support sets are minimal in size and fewer values need to be re-examined during the propagation of a removed value.

  In the above example, (V_2, r) is not in (V_3, b)'s support set, though in fact (V_3, b) supports it. When we process the removal of (V_3, b), we do not need to re-examine (V_2, r).

2.1.3 AC7

There is further property of *support* that can be utilized. Support is *bidirectional*. Value a of variable V_i supports value b of variable V_j if and only if value b of variable V_j supports value a of variable V_i. In AC4 or AC6, when we establish that support for value a of variable V_i is provided by value b of variable V_j, we have to check the constraint between $V_i = a$ and $V_j = b$. Later on, if we need to establish support for value b of variable V_j by value a of variable V_i, we would check the same constraint again. Clearly, this constraint check is redundant. AC7 proposed by Bessiere, Freuder and Regin in [2] is an algorithm based on AC6 that makes use of this property to further reduce constraint checks.

To implement this algorithm, we only need to make some small modifications on AC6. More specifically, in function $SEEK\_SUPPORT$ of AC6, whenever we need to seek a support for a value (V_i, a) in the domain of V_j, instead of checking every value of V_j, we first traverse the support set of (V_i, a) to see whether (V_i, a) supports any value in the current domain of V_j. If there is, then we do not need to make any further constraint checks against the values of V_j.

2.2 MAC

To embed these improved AC algorithms in backtracking search is not straightforward. The main difficulty comes from the cost of backtracking. In backtracking search, when we try to instantiate a variable, we have to temporary remove all the other values of the variable except the instantiated one. Furthermore, during the procedure to achieve full arc consistency, more values of different variables are (temporary) removed. Once the instantiation fails, backtrack has to occur and we have to restore all those removed values. As we pointed out in Chapter 1, we solve this problem using an extra data structure called *Domain*. But the problem arising from it is: due to the removal and restoring of the values, the support sets and the counters used in these AC algorithms are affected accordingly. We also have to find a way to restore the relevant support sets and counters during backtracking. How to handle this problem will be the main issue discussed in this section.

2.2.1 MAC4 - embedding AC4 in backtracking search

The effectiveness of MAC on large and hard problems was first pointed out by Sabin and Freuder in [9], where they used AC4 as the algorithm to achieve arc consistency.

In AC4, *Counter*s are used to keep track of the number of supports a variable-value pair has from a certain arc. We need to update all the relevant *Counter*s after an instantiation, and of course need to restore them back to their previous values once the instantiation fails.

In the algorithm proposed by Sabin and Freuder in [9], they save the previous version of *Counter*s elsewhere and restore it when needed. This is a good method with reasonable time efficiency. But a lot of space is required because we have to store a different version of *Counter*s at each level of instantiation. If there are n variables, c constraints defined on them and the domain size of each variable is k, the space complexity is $O(nck)$. The space concern can be a problem which makes it almost impractical.

The purpose of using *Counter*s is to make it easier to know whether the affected values are still supported or not after some values are removed. In fact, we can always find this out by explicitly seeking support. Moreover, since we have full support sets for all values in AC4, by utilizing the bidirection property of support, we

can seek a support for a value by traversing its own support set without constraint checks. That is, if we want to seek a support for $V_i = a$ in the domain of V_j, we only need to check whether there is any un-removed value of V_j in the support set of (V_i, a) instead of checking the compatibility of (V_i, a) and every value of V_j. This is the method used in the following MAC4 algorithm. Compared to the method in [9], it is trading off time for space.

```
01 function SEEK_SUPPORT4(V_i, V_j, v_l^i)
02     for each v_m^j ∈ S_{V_i V_j v_l^i}
03         if Domain_m^j is not marked
04             return TRUE
05     return FALSE
```

```
01 function PROPAGATE_AC4(Vars, Level)
02     while DeletionStream ≠ ∅
03         select and remove (V_j, v_m^j) from DeletionStream
04         for each variable V_i ∈ Vars
05             if R_ji ∈ R
06                 for each v_l^i ∈ S_{V_j V_i v_m^j} such that
                       Domain_l^i is not marked
07                     if not SEEK_SUPPORT4(V_i, V_j, v_l^i)
08                         Domain_l^i ← Marked at Level
09                         DeletionStream ← DeletionStream + (V_i, v_l^i)
10                         if DWO(V_i)
11                             return FALSE
12     return TRUE
```

```
01  function SEARCH_MAC4(Vars, Level)
02      select a variable V_i ∈ Vars
03      for each value v_l^i ∈ D_i such that Domain_l^i is not marked
04          Solution ← Solution + (V_i, v_l^i)
05          if V_i is the only variable in Vars
                % Found a solution
06              return TRUE
07          else
                % Eliminate all the other values of V_i
08              for each value v_l'^i ∈ D_i\{v_l^i} such that Domain_l'^i
                    is not marked
09                  Domain_l'^i ← Marked at Level
N1                  DeletionStream ← DeletionStream + (V_i, v_l'^i)
13              if PROPAGATE_AC4(Vars\{V_i}, Level) and
                    SEARCH_MAC4(Vars\{V_i}, Level + 1)
14                  return TURE
15              else
16                  Solution ← Solution − (V_i, v_l^i)
17                  RESTORE(Vars, Level)
            % No solution down this branch
18      return FALSE
```

```
01 function MAC4
02    for each variable V_i ∈ V
03       for each value v_l^i ∈ D_i
04          Domain_l^i ← unMarked
05          for each variable V_j ∈ V
06             if R_ij ∈ R
07                S_{V_i V_j v_l^i} ← ∅
08    Solution ← ∅
09    DeletionStream ← ∅
10    for each variable V_i ∈ V
11       for each variable V_j ∈ V
12          if R_ij ∈ R
13             for each value v_l^i ∈ D_i such that
                  Domain_l^i is not marked
14                total ← 0
15                for each value v_m^j ∈ D_j such that
                     Domain_m^j is not marked
16                   if (v_l^i, v_m^j) ∈ R_ij
17                      total ← total + 1
18                      S_{V_j V_i v_m^j} ← S_{V_j V_i v_m^j} + v_l^i
19                if total = 0
20                   Domain_l^i ← Marked at 0
21                   DeletionStream ← DeletionStream + (V_i, v_l^i)
22                   if DWO(V_i)
23                      return FALSE
24    if PROPAGATE_AC4(V, 0)
25       return SEARCH_MAC4(V, 1)
26    else
         % No solution due to arc inconsistency
27       return FALSE
```

MAC4 uses a similar framework to the MAC3 algorithm presented in Chapter 1. The main function $MAC4$ initializes data structures including the support sets and starts search. Some of the codes are borrowed from the first part of AC4. All support sets only need to be set up once and will not change during the search, though some of the values in some of the support sets may be removed (and may be put back again later). We will not explicitly keep track of the status of a value in the support sets. We can get that information from $Domain$. In order to make it more efficient to seek a support for a value, we also split support sets $S_{V_i a}$ into $S_{V_i V_j a}$, so that each value has a support set for each constraint. This way, in the function $SEEK\_SUPPORT4$, when we need to seek a support for (V_i, a) in the domain of V_j, we can go directly to the support set of (V_i, a) for the arc (V_j, V_i). Function $PROPAGATE\_AC4$ is borrowed from the second part of AC4. The difference is: instead of updating $Counters$, it calls the function $SEEK\_SUPPORT4$ to check whether the affected values are still supported. The function $SEARCH\_MAC4$ is modified from the function $SEARCH\_MAC3$ in MAC such that instead of putting affected arcs into Q, it puts deleted values into $DeletionStream$ to start the arc consistency propagation.

2.2.2 MAC6 and MAC7 - embedding AC6 and AC7 in backtracking search

Since AC6 keeps minimal support sets, for each value there is only one support per arc. After an instantiation, if a support for a value is removed, we have to find a new support for that value. This makes supports sets different from level to level. How to update and restore these support sets?

In fact, there are several ways to do so. Suppose we have a CSP; V_i and V_j are two of the variables; the domain of V_i is $\{a, b, c\}$; the domain of V_j is $\{d, e\}$; V_i and V_j are constrained; and $\{(a, d), (a, e), (b, d), (c, e)\}$ are the only compatible value pairs defined by the constraint between V_i and V_j. Initially, full arc consistency is achieved by AC6 and a gets d as its support. Later on, as the result of an instantiation, d is pruned, and a needs to find another support in the domain of V_j. e is found and a is put in the support set of $S_{V_j e}$. At this point, we have several choices:

1. Remove a from the support set $S_{V_j d}$.

   If this instantiation fails, during the restore stage, we will remove a from $S_{V_j,e}$ and put it back to $S_{V_j,d}$. (Restore everything just as before the instantiation).

2. Remove a from the support set $S_{V_j d}$.

   But if this instantiation fails, we still keep e as the current support of a, a is still in the support set $S_{V_j e}$, but not in the support set $S_{V_j d}$, even though d is available now.

3. Keep a in both the support sets $S_{V_j d}$ and $S_{V_j e}$.

   If this instantiation fails and d is put back to the domain of V_j, a will have two supports in the domain of V_j and a is in both support sets.

4. Use two kinds of support sets.

   One is for the minimal support sets (called $MinS$) and the other (still called S) for all the supporting information we obtain during search. Initially, a is in both $S_{V_j d}$ and $MinS_{V_j d}$. After d is removed and a finds a new support e, a is put in both $S_{V_j e}$ and $MinS_{V_j e}$. But we remove a from $MinS_{V_j d}$ only. It is not removed from $S_{V_j d}$. If this instantiation fails, we do not put a back into $MinS_{V_j d}$, nor do we remove it from $MinS_{V_j e}$.

Method 1 and 2 reflect the idea of AC6: for each value there is only one support per constraint and the support sets are kept minimal. Method 1 is not a good idea due to the high cost of restoring previous state. In fact as long as there is a support, we do not care what it is. Method 2 keeps an equivalent state though it is not exactly the previous state. Using method 2, we have the following algorithm, called MAC6.

```
01 function SEEK_SUPPORT6(V_i, V_j, v_l^i)
02     for each value v_m^j ∈ D_j such that
           Domain_m^j is not marked
03         if (v_l^i, v_m^j) ∈ R_ij
04             S_{V_j v_m^j} ← S_{V_j v_m^j} + (V_i, v_l^i)
05             return TRUE
06     return FALSE
```

```
01 function PROPAGATE_AC6(Vars, Level)
02     while DeletionStream ≠ ∅
03         select and remove (V_j, v_m^j) from DeletionStream
04         for each (V_i, v_l^i) ∈ S_{V_j v_m^j} such that
               V_i ∈ Vars and Domain_l^i is not marked
05             if SEEK_SUPPORT6(V_i, V_j, v_l^i)
06                 S_{V_j v_m^j} = S_{V_j v_m^j} − (V_i, v_l^i)
07             else
08                 Domain_l^i ← Marked at Level
09                 DeletionStream ← DeletionStream + (V_i, v_l^i)
10                 if DWO(V_i)
11                     return FALSE
12     return TRUE
```

```
01 function MAC6
02    for each variable V_i ∈ V
03       for each value v_l^i ∈ D_i
04          Domain_l^i ← unMarked
05          S_{V_i v_l^i} ← ∅
06    Solution ← ∅
07    DeletionStream ← ∅
08    for each variable V_i ∈ V
09       for each variable V_j ∈ V
10          if R_ij ∈ R
11             for each value v_l^i ∈ D_i such that
                  Domain_l^i is not marked
12                if not SEEK_SUPPORT6(V_i, V_j, v_l^i)
13                   Domain_l^i ← Marked at 0
14                   DeletionStream ← DeletionStream + (V_i, v_l^i)
15                   if DWO(V_i)
16                      return FALSE
17    if PROPAGATE_AC6(V, 0)
18       return SEARCH_MAC6(V, 1)
19    else
         % No solution due to arc inconsistency
20       return FALSE
```

All of the code is borrowed either from AC6 or MAC4, with minor changes. The function $SEARCH\_MAC6$ is not listed because it is exactly the same as $SEARCH\_MAC4$ except that it calls $PROPAGATE\_AC6$ instead of $PROPAGATE\_AC4$. Line 08-16 of $MAC6$ is similar to the first part of AC6, $PROPAGATE\_AC6$ is similar to the second part of AC6. One difference to point out is in the function $SEEK\_SUPPORT6$. Here we cannot guarantee the order the values are checked. So we have to check from the beginning every time. For example, suppose e is the current support for a, we cannot guarantee that the values before e cannot be supports for a. One of them may have been previously used as a support only to have been removed temporarily. d is such a value. If e is pruned later but d is available at that time, we should be able to find d as a support for a.

As you can see, one problem of method 2 is that we used to know that d could be a support for a. But that information is lost once we get a new support e for a. If in the future e is pruned and we need to find another support, we need redundant constraint checks to once again detect that d can be a support. Method 3 tries to solve this problem by keeping the results of all constraint checks in the support sets. However, the support sets get larger and larger and some values will have more than one support for each arc which is contrary to the principle of AC6. One alternative is to use more space to maintain two kinds of support sets. One is the minimal support sets as in method 2 and the other is equivalent to those in method 3. This is what method 4 does. Following is the algorithm based on the idea of method 4. Since it also makes use of the bidirection property of support, we call it MAC7.

```
01 function SEEK_SUPPORT7(V_i, V_j, v_l^i)
02     for each v_m^j ∈ S_{V_i V_j v_l^i}
03         if Domain_m^j is not marked
04             MinS_{V_j v_m^j} ← MinS_{V_j v_m^j} + (V_i, v_l^i)
05             return TRUE
06     for each value v_m^j ∈ UnChecked_{V_i V_j v_l^i} such that
           Domain_m^j is not marked
07         UnChecked_{V_i V_j v_l^i} ← UnChecked_{V_i V_j v_l^i} − v_m^j
08         if (v_l^i, v_m^j) ∈ R_ij
09             S_{V_j V_i v_m^j} ← S_{V_j V_i v_m^j} + v_l^i
10             S_{V_i V_j v_l^i} ← S_{V_i V_j v_l^i} + v_m^j
11             MinS_{V_j v_m^j} ← MinS_{V_j v_m^j} + (V_i, v_l^i)
12             return TRUE
13     return FALSE
```

```
01 function PROPAGATE_AC7(Vars, Level)
02     while DeletionStream ≠ ∅
03         select and remove (V_j, v_m^j) from DeletionStream
04         for each (V_i, v_l^i) ∈ MinS_{V_j v_m^j} such that
               V_i ∈ Vars and Domain_l^i is not marked
05             if SEEK_SUPPORT7(V_i, V_j, v_l^i)
06                 MinS_{V_j v_m^j} = MinS_{V_j v_m^j} − (V_i, v_l^i)
06             else
07                 Domain_l^i ← Marked at Level
08                 DeletionStream ← DeletionStream + (V_i, v_l^i)
09                 if DWO(V_i)
10                     return FALSE
11     return TRUE
```

```
01 function MAC7
02    for each variable V_i ∈ V
03       for each value v_l^i ∈ D_i
04          Domain_l^i ← unMarked
05          MinS_{V_i v_l^i} ← ∅
06          for each variable V_j ∈ V
07             if R_ij ∈ R
08                S_{V_i V_j v_l^i} ← ∅
09                UnChecked_{V_i V_j v_l^i} ← D_j
10    Solution ← ∅
11    DeletionStream ← ∅
12    for each variable V_i ∈ V
13       for each variable V_j ∈ V
14          if R_ij ∈ R
15             for each value v_l^i ∈ D_i such that
                  Domain_l^i is not marked
16                if not SEEK_SUPPORT7(V_i, V_j, v_l^i)
17                   Domain_l^i ← Marked at 0
18                   DeletionStream ← DeletionStream + (V_i, v_l^i)
19                   if DWO(V_i)
20                      return FALSE
21    if PROPAGATE_AC7(V, 0)
22       return SEARCH_MAC7(V, 1)
23    else
         % No solution due to arc inconsistency
24       return FALSE
```

MAC7 is an improvement on MAC6. It has two kinds of support sets: S and $MinS$. $MinS$ is kept minimal while S contains all the supports that are obtained

from constraint checks during search. Moreover, if we find a support a for a value b, we will not only put b in the support set S of a, but also put a in the support set S of b, because b also can be a support for a. As in MAC4, we also split S for each arc and when we need to seek a support for a value, we check its support set S first. But since S may not be complete, if we cannot get a support, we also need to check those unchecked values. A data structure $UnChecked$ is used to maintain those unchecked values. Initially it contains all the values in the domains. Once a value is checked, it is removed from $UnChecked$. $MinS$ is used to get those values affected by removals. If a new support is found for a value, the old support has to be removed from $MinS$. Again $SEARCH\_MAC7$ is not listed because it is exactly the same as $SEARCH\_MAC4$ except that it calls $PROPAGATE\_AC7$.

2.2.3 Summary

All the algorithms of MAC we discussed so far will visit the same number of nodes in a search tree given the same problem. The difference is the work done at each node, i.e., the work to maintain arc consistency after the instantiation at each node. These arc consistency algorithms are triggered by value removals and arc consistency is further achieved by propagating these removals. In MAC3, in the processing of the removal of a value of a variable, all values in the domains of all variables that have an arc going to that variable are processed to find another support in that variable. In MAC4, MAC6 and MAC7, only those values in that variable-value pair's support set need to find a new support. In MAC4, all the support sets are full support sets. In MAC6 or MAC7, for each value, we only keep one support per arc. The support sets are kept minimal. Therefore there are a fewer number of values for which we need to seek a new support.

2.3 Improving MAC: using heuristics

Heuristics can be incorporated in MAC to reduce search costs. Generally, a heuristic can be viewed as a "rule of thumb" that improves the average-case performance on a problem-solving task, but offers no guarantees for improving the worst case.

2.3.1 Variable ordering

In all the algorithms presented in Chapter 1 and the previous section, we assume that the variables of a CSP are given in an order specified by their indices and they are instantiated chronologically during the search with no regard to the properties of each variable. For example, if we have $V = \{V_1, ..., V_i, ..., V_n\}$, the instantiation order will be V_1,...,V_i,...,V_n.

However, it has been known for a long time that the order in which the variables are instantiated strongly affects the size of the search space explored by backtracking based algorithms [13] [3], and heuristics utilizing certain properties of the variables can be incorporated to speed up search.

Variable ordering heuristics can be divided into two classes: static variable ordering (*SVO*) and dynamic variable ordering (*DVO*). A *SVO* orders the variables before search starts while a *DVO* orders the variables at each node, that is, the order in which the variables are instantiated can vary from branch to branch in the search tree. *DVO* is generally better than *SVO* because it uses more updated information, despite the fact that it costs more to compute.

The minimum domain (*dom*) heuristic which selects as the next variable to be instantiated a variable that has a minimal number of remaining values in its domain, has been considered as the best variable ordering heuristic [3]. It is a *DVO* when incorporated in FC or MAC in that the sizes of domains vary from node to node due to value removals.

Another widely used variable ordering heuristic is called the maximum degree heuristic (*deg*). It orders the variables by decreasing number of neighbors in the constraint graph in a sense that the most constrained variables are chosen first. It can be used both as a *SVO* and as a *DVO*. If it is used as a *SVO*, the degrees are calculated based on the initial constraint graph. If it is used as a *DVO*, the degrees

are calculated at each node based on the current graph with all variables that are already instantiated disconnected.

Variable ordering heuristics can be applied on both MAC and FC. To implement these heuristics, we only need to replace line 02 in all those $SEARCH\_*$ functions with:

01 function $SEARCH\_*\,(Vars, Level)$
02 $\quad V_i = SELECT(Vars)$
...

where $SELECT$ is a function to return a variable choosing from the current available variables according to a certain heuristic (or a combination of heuristics). For example, if we want to use the *dom* heuristic, we can get the current domain size for each variable from the array $Domain$ and select the variable with the smallest domain size. If we want to use *deg* as a *SVO*, we can compute the degrees for each variable before search starts (possibly in the main part initialization stage) and select the variable with the largest degree each time $SELECT$ is called. However, more work needs to be done if we want to use *deg* as a *DVO*.

Removing refuted value with the *dom* heuristic

There is another possible improvement that can be made to MAC: whenever an instantiation fails, we can remove the refuted value from the domain and restore arc consistency [10].

There are two expectations in doing so:

1. We might detect an unsolvable branch earlier.

   If a DWO occurs during the arc consistency propagation after the refuted value is removed, we can determine that there is no solution down the branch of the previous upper level instantiation.

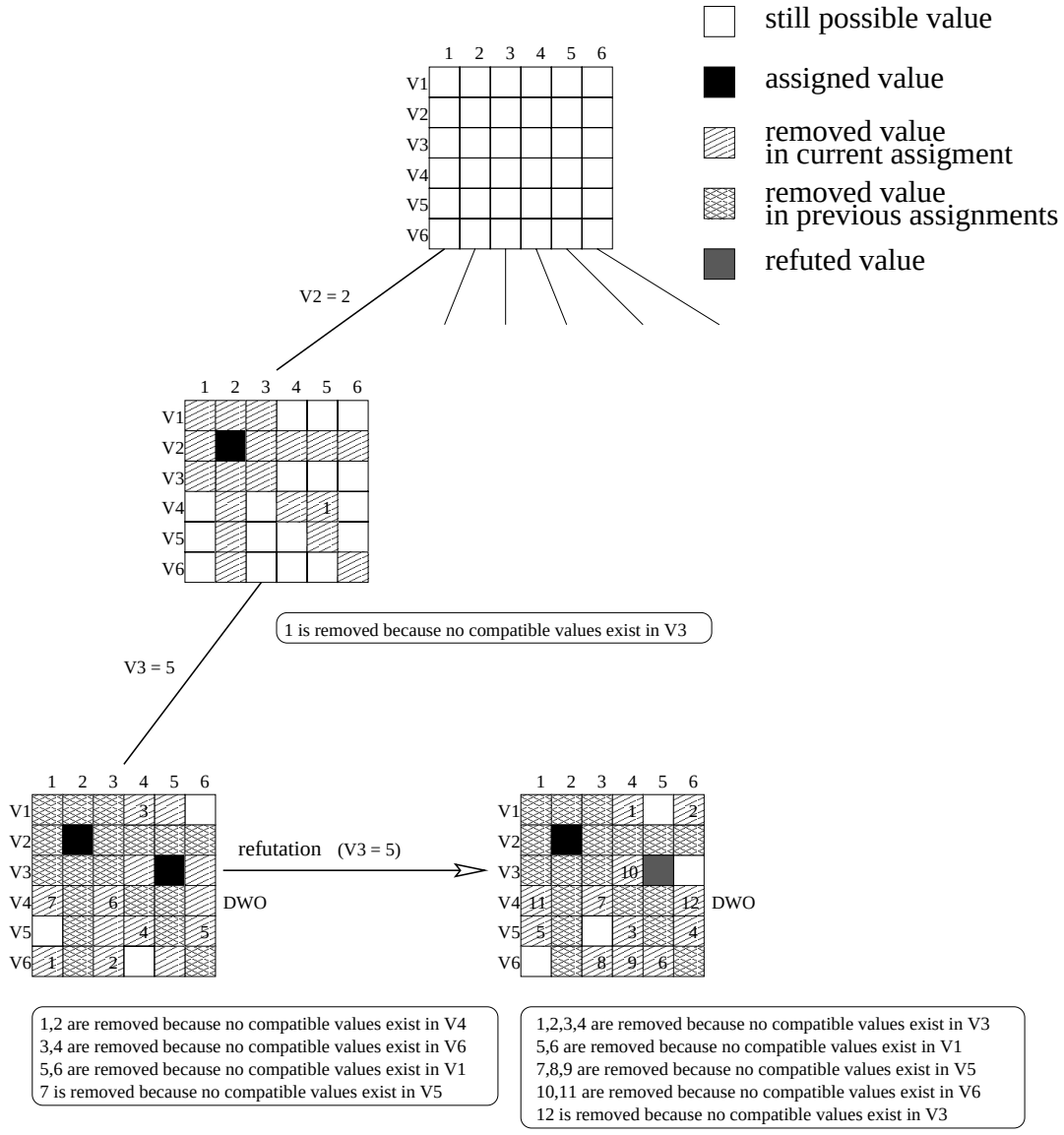

Figure 2.1: Solving the 6-queens problem using MAC with refutation

Part of the search tree for solving the 6-queens problem using MAC with removed refuted values is presented in Figure 2.1. We use the same problem modeling method as for the 4-queens problem in Chapter 1. We start search by putting the first queen in the second square of the second row. Then arc consistency is achieved with no DWO. We continue search by putting the second queen in the fifth square of the third row. Unfortunately, this instantiation is unsuccessful because it causes a DWO. Moreover, during the refutation of this value, DWO occurs again. That means, all the rest of the squares at the third row are no good for a queen. We do not need to examine them one by one, as we would do if we did not have a refutation. We can conclude that the first queen is put in the wrong place at the second row and no solution exists if a queen is in that square.

2. We might choose a new variable after a refutation if searching with *dom*.

   We can apply the heuristic *dom* both after a successful instantiation and after a refutation. In other words, instead of selecting a variable by *dom* and trying to instantiate every available value of it one after another until success or we exhaust its domain, we can also use *dom* to choose a new variable every time after a refutation. Since we are going to achieve arc consistency during a refutation, more values might be removed and some domain sizes might be changed. At this point, the variable with the minimal domain size might be different from the variable instantiated and refuted just now. If we use *dom* again, we can perhaps select a better variable according to the new situation.

We can implement this method in MAC by changing the code for the $SEARCH\_MAC*$ functions.

```
01  function SEARCH_MAC4(Vars, Level)
R1      while TRUE
02          V_i = SELECT(Vars)
03          select a value v_l^i ∈ D_i such that Domain_l^i is not marked
...
15              else
16                  Solution ← Solution − (V_i, v_l^i)
17                  RESTORE(Vars, Level)
R2                  Domain_l^i ← Marked at Level − 1
R3                  DeletionStream ← DeletionStream + (V_i, v_l^i)
R4                  if not PROPAGATE_AC4(Vars, Level − 1)
R5                      break
        % No solution down this branch
18      return FALSE
```

We change line 03 from a loop to one statement, but insert a larger loop at line $R1$ to include line 02 inside it. This way, we are able to select a new variable after every refutation. Line $R2$-$R5$ are inserted to process refutations. Note that we take the refuted value and any other values that are pruned during a refutation as removed at the upper level — $Level - 1$. This is to guarantee that these removed values can be restored during the restore procedure of the instantiation at an upper level. It makes sense because their removal is actually due to the instantiation at the upper level.

We can make similar changes to $SEARCH\_MAC3$, except that instead of putting the refuted value into $DeletionStream$, we have to put the affected arcs into Q.

2.3.2 Singleton variables

A singleton variable is a variable with domain size of 1. MAC can be improved by special treatment of singleton variables [10].

Singleton variables are specially treated in several senses:

1. After restoring arc consistency for all arcs from other variables to a singleton, we can temporarily disconnect the singleton variable from the constraint graph and thus avoid studying the constraints connecting other variables to this singleton in further arc consistency propagation and deeper search. This follows from the fact that: since all the values remaining in the domains of other variables are compatible with the only value of the singleton variable, for arcs from other variables to the singleton, each value of the other variables can find the single value of the singleton as a support; for arcs from the singleton variable to other variables, the single value can always find a support from all the other variables unless there is a DWO.

2. To utilize the above advantage, we need to achieve arc consistency on the arcs related to the singleton first, that is, before processing other arcs. Moreover, we can check directly the values of other variables that are compatible with the singleton value for all these arcs.

3. We can defer the instantiation of singleton variables to the end of the search to reduce the number of nodes visited.

Let's consider the example in Figure 2.2. One special case of singletons are those instantiated variables. They are made singleton explicitly by being assigned a value, which removes all other values in their domains. V_1 is such a singleton. When V_1 is instantiated, according to 2, we should check directly the values of V_2 and V_3 that are compatible with $V_1 = r$ instead of propagating the removal of g and b. This way is likely to be more efficient when the number of deleted values that need to be processed is more than one.

Other singletons can be obtained during the arc consistency propagation. An example of this case is V_2. After full arc consistency is achieved, we are ready to select another variable to instantiate. However, according to 3, V_2 should not be instantiated. In fact, after all other variables are instantiated, we can instantiate V_2 and other singleton variables easily with no backtracking. If we instantiate V_2 now,

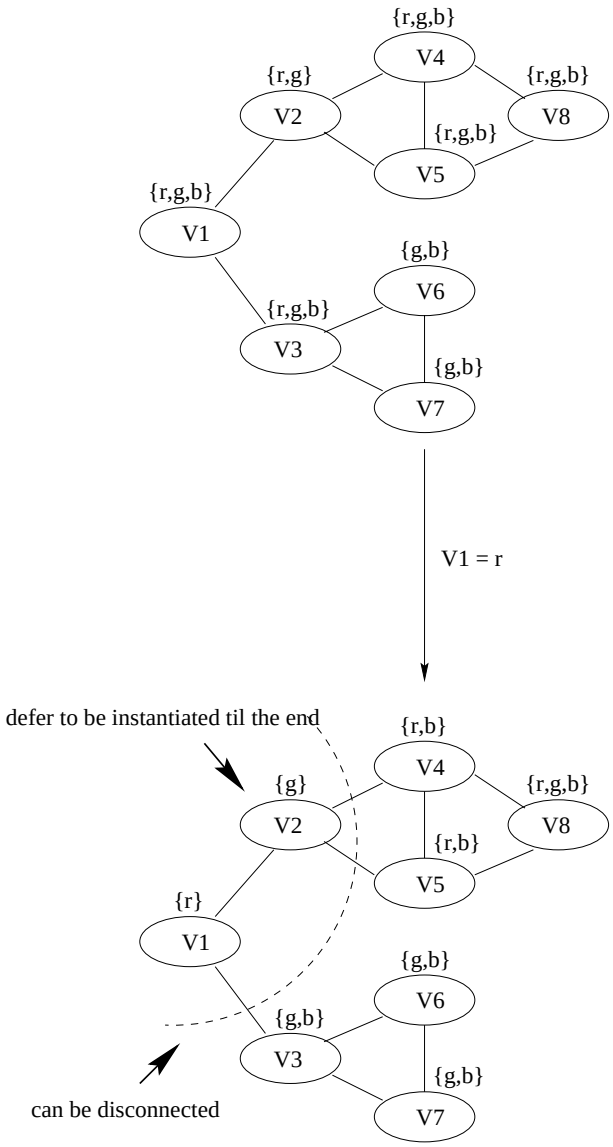

Figure 2.2: An example of singleton variables

backtrack may occur later due to previous instantiations and the instantiation of V_2 has to be restored. Actually this is the case for the example. No solution exists with $V_1 = r$ and we can detect this without instantiating V_2.

Chapter 3

FC reconsidered

The FC algorithm we introduced in Chapter 1, in a sense, uses "partial" AC3 as the algorithm for arc consistency. Can any of the methods in Chapter 2 also be applied to forward checking in order to improve FC? In this chapter, we discuss a new FC algorithm.

3.1 Using non-support sets

Inspired by the AC4 algorithm, when we try to achieve partial arc consistency in the FC algorithm, instead of checking the constraints between the instantiated value of the current variable and all values of all future variables that are constrained with the current variable to prune away incompatible values of future variables, we can also get these incompatible values by accessing already setup information.

In AC4, we have a support set for each value of each variable recording all the values it supports in the domains of all variables that variable is constrained with. However, in FC, support sets do not help a lot. When we instantiate a variable, we need to look ahead to prune away all the values of future variables that are not compatible with the current instantiated variable. Because all values of those future variables (that are constrained with the current variable) are either compatible or incompatible with the current instantiated value, i.e., are either supported or not supported by it, if we still use support sets, for those future variables that are constrained with the current variable, we need to prune away all the values that are *not* in the support set of the current instantiated value. In other words, we need the complement of the support set of the current instantiated value (for the

constrained future variables) that records all the values that are *not* supported by the current instantiated value, but are in the domains of variables that are constrained with the current variable. We call this a *non-support set*. For example, in Figure 3.1, the non-support set of $(V1, r)$ is $\{(V2, r)\}$.

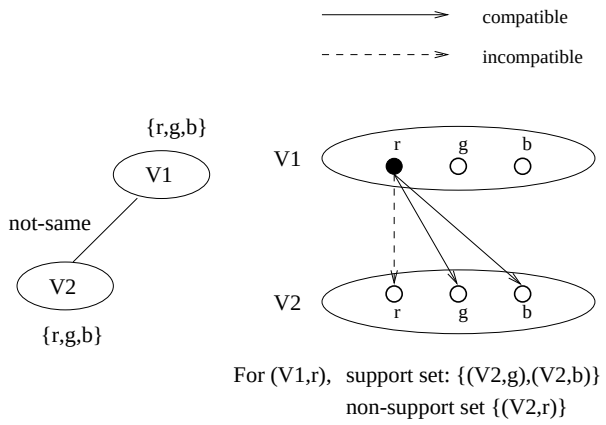

Figure 3.1: Using non-support sets

We need to set up a non-support set for each value of each variable. Like in MAC4, we can set up all the non-support sets before search starts. During the search, after a variable is instantiated, it traverses its non-support set to prune away all the values in it that are not pruned yet. Since only part of the values in a domain will be in the non-support sets, the number of values that need to be examined will be reduced. For example, in Figure 3.1, when we instantiate $V1$ and assign r to it, from its non-support set, we only need to prune one value r in the domain of $V2$. We don't need to check all three values of $V2$.

3.2 Algorithm

The algorithm of the new FC is listed below. We call it FC4.

```
01  function CHECK_FORWARD4(Vars, Level, V_i, v_l^i)
02      for each (V_j, v_m^j) ∈ ¬S_{V_i v_l^i} such that
            V_j ∈ Vars and Domain_m^j is not marked
03          Domain_m^j ← Marked at Level
04          if DWO(V_j)
05              return FALSE
06      return TRUE
```

```
01  function FC4
02      for each variable V_i ∈ V
03          for each value v_l^i ∈ D_i
04              Domain_l^i ← unMarked
N1              ¬S_{V_i v_l^i} ← ∅
05      Solution ← ∅
N2      for each variable V_i ∈ V
N3          for each variable V_j ∈ V such that j > i
N4              if R_ij ∈ R
N5                  for each value v_l^i ∈ D_i
N6                      for each value v_m^j ∈ D_j
N7                          if (v_l^i, v_m^j) ∉ R_ij
N8                              ¬S_{V_j v_m^j} ← ¬S_{V_j v_m^j} + (V_i, v_l^i)
N9                              ¬S_{V_i v_l^i} ← ¬S_{V_i v_l^i} + (V_j, v_m^j)
06      return SEARCH_FC4(V, 1)
```

Not surprisingly, the structure of FC4 is similar to that of FC3. $FC4$ replaces $FC3$ in that non-support sets are initialized and set up. The function $SEARCH\_FC4$ is not listed since it is exactly the same as $SEARCH\_FC3$ in Chapter 1 except that in line 08 it calls $CHECK\_FORWARD4$ and $SEARCH\_FC4$ instead of $CHECK\_FORWARD3$ and $SEARCH\_FC3$. In the function $CHECK\_FORWARD4$, non-support sets are used to remove incompatible values of future variables without involving any constraint checks.

3.3 Analysis

Figure 3.2 is the search tree of applying FC4 to a map-coloring problem. Assuming non-support sets are set up already before search starts. The non-support sets that will be used for each instantiation are listed below the relevant instantiations. FC4 will visit exactly the same number of nodes as FC3 does. If we use FC3, we will get the same search tree for the same problem. The saving of FC4 is obtained at each node. For example, assume we have successfully instantiated $V1$ to g and now we are going to instantiate $V2$ to r. At this point, $V4$ is the only future variable that is constrained with $V2$. If we use FC3, we have to consider all the values r, g and b in the domain of $V4$ one by one to see whether any of them are incompatible with $V2 = r$ and have to be pruned. As a result, we will remove r since g and b are both compatible with $V2 = r$. However, if we use FC4, we know right-away that only $(V4, r)$ is not supported by the assignment $V2 = r$. All need to do is to prune r from $V4$.

Compared to FC3, extra work needs to be done to initialize and set up non-support sets. But the complexity is polynomial in the size of the problem. (The complexity of search might be exponential in the size of the problem.) If there are c constraints and the domain size of each variable is k, then the total number of constraint checks is ck^2. When the problem is large and hard, the search tree is deep and wide, the cost of setting up non-support sets will be negligible compared to the cost of search. Moreover, we only need to set up this information once. We can use it again and again during search when backtracks occur. (If the problem is hard, we should always expect many backtracks). Consider the example in Figure 3.2 again, $V2$ is instantiated to r twice, one is after g is assigned to $V1$, the other is after b is assigned to $V1$. We also benefit from the non-support set of $(V2, r)$ twice. For hard and large problems, there will be many more backtracks and the more nodes visited, the more savings we can get with regard to the search cost using the original FC3.

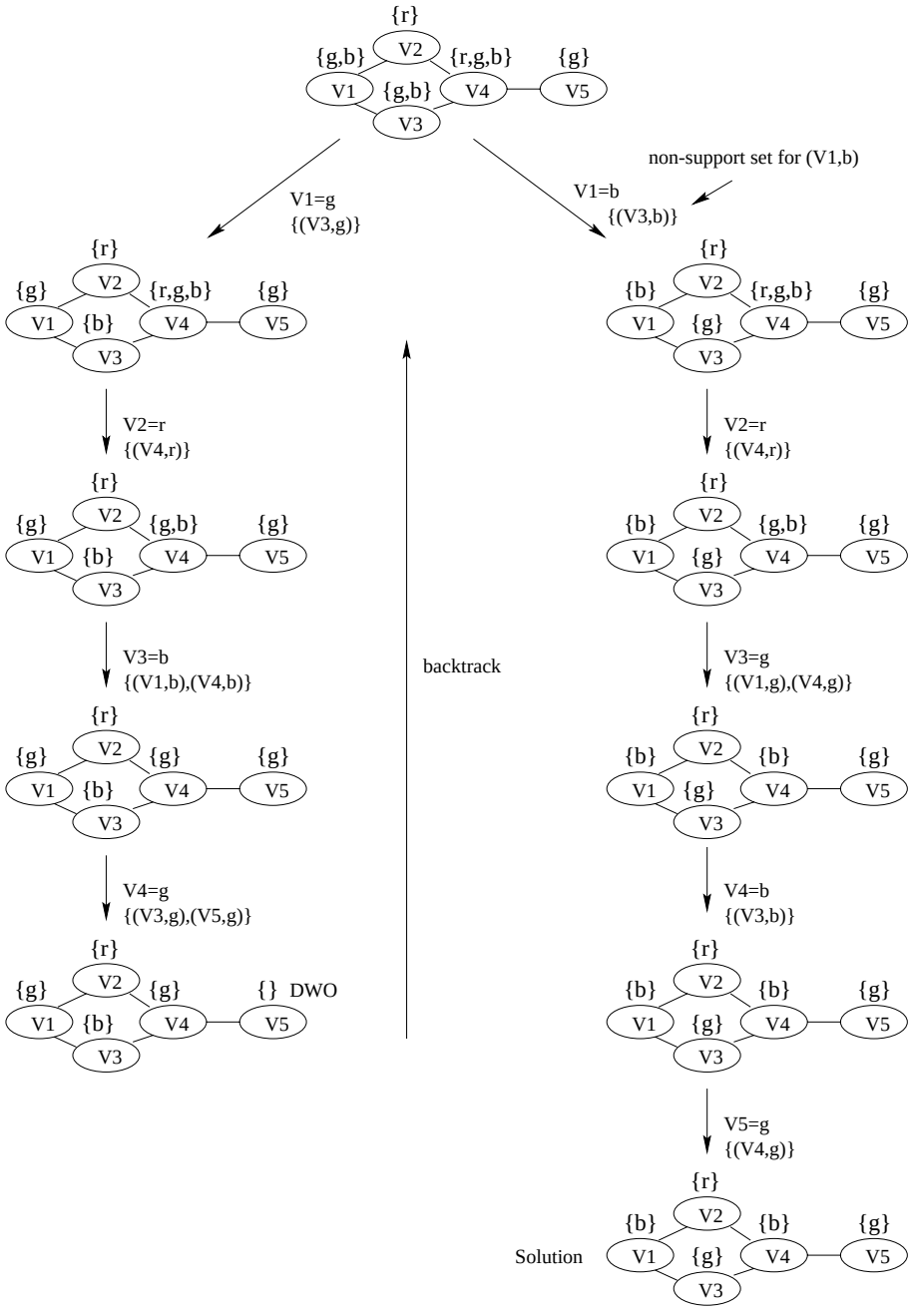

Figure 3.2: Solving a map-coloring problem using FC4

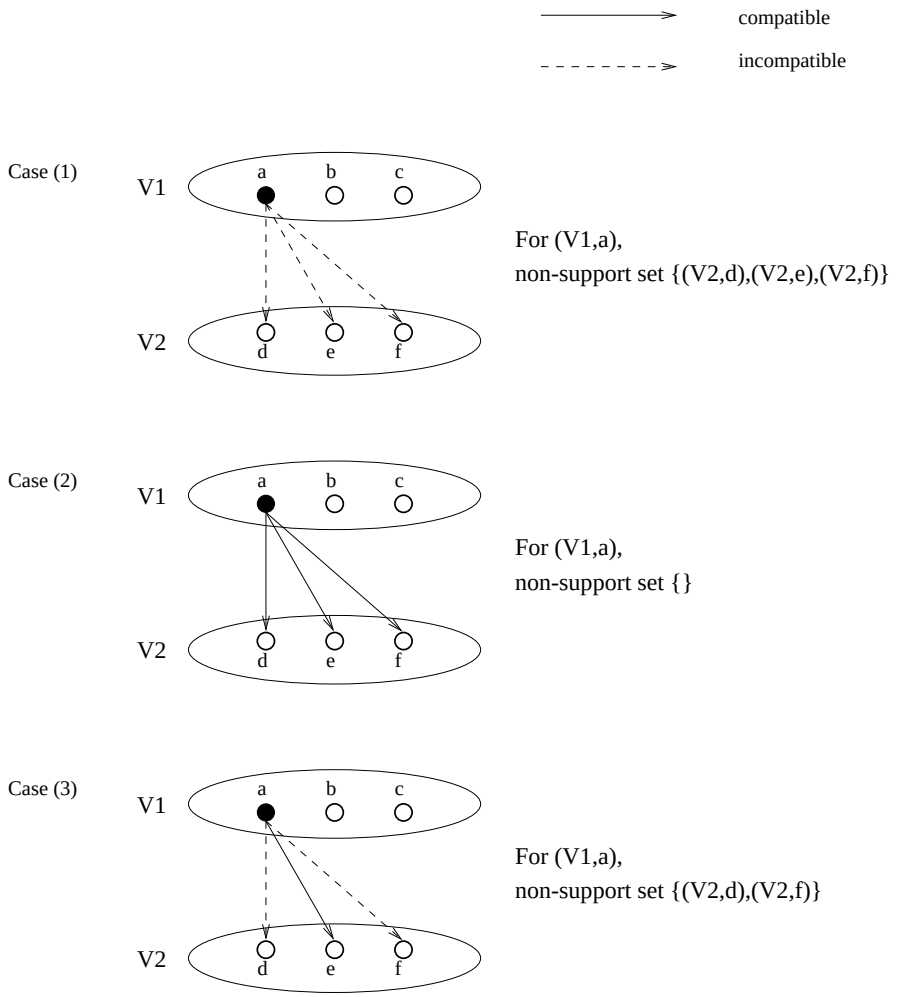

Figure 3.3: Non-support sets analysis

The performance of FC4 depends on the sizes of the non-support sets. Since we only need to process those values in the relevant non-support set at each node, the smaller the size of the non-support set, the more savings we can achieve. Suppose a non-support set $\neg S_{V_1 a}$ is defined for (V_1, a), and V_2 is the only variable that is constrained with V_1. In the worst case, the size of $\neg S_{V_1 a}$ is equal to the domain size of V_2. This is case(1) in Figure 3.3. In this case, since all values of V_2 are incompatible with a, an instantiation of V_1 to a will cause the removal of all values from V_2. In the best case, the size of $\neg S_{V_1 a}$ is zero. This is case(2) in Figure 3.3. In this case, all values are compatible and nothing need be pruned. These are two extreme cases. In most cases, however, the size of $\neg S_{V_1 a}$ divided by the domain size

of V_2 is a fraction between $(0, 1)$, as in case(3) in Figure 3.3. This factor is related to the *tightness* of the constraint between V_1 and V_2, that is, the fraction of all possible pairs of values from the domains of these two constrained variables, that are not allowed by the constraint. Tightness will be used as a property of a CSP and we will discuss it in more detail in the next chapter. Suppose the domain sizes of two constrained variables are both k, the tightness of the constraint is t, then the number of all possible pairs of values from the two domains that are not allowed by the constraint is $t \times (k \times k)$. In other words, for each value of one variable, the expected number of values of the other variable that are incompatible with it is

$$t \times (k \times k) \div k = t \times k \ .$$

Therefore, we expect FC4 works well with CSPs with low tightness.

Chapter 4

Empirical Results

Since finding a solution to a CSP is an NP-hard problem as noted in Chapter 1, worst-case analysis does not necessarily reflect actual performance. Furthermore, theoretical estimation of performance, e.g., expected case analysis is not always possible. Therefore, empirical evaluation of the various algorithms is necessary. In this chapter, we use empirical results to address the distribution of problem classes on which FC or MAC works better.

4.1 Experimental design

We use randomly generated binary CSPs to do our experiments and we only look for the first solution to these problems.

4.1.1 Problem generation

A problem is generated using four parameters: N, K, C and T.

- N is the number of variables.
- K is the domain size of each variable, i.e., the number of values for each variable. Initially the domain size for all variables is equal.
- C is the number of binary constraints.

  This number is related to N. Since a binary constraint is defined on two variables, the maximum possible number of constraints on N variables is

$\frac{N(N-1)}{2}$. Therefore, the range of C is from 0 to $\frac{N(N-1)}{2}$. If C is 0, none of the variables are constrained, and there are no links in the constraint graph. If C is $\frac{N(N-1)}{2}$, each variable is constrained with all other variables and the constraint graph is a complete graph. The fraction of C over $\frac{N(N-1)}{2}$ is called the *constraint density* of the CSP problem.

- T is the number of incompatible pairs of values for each constraint.

  This number is related to K. For each constraint, the number of all possible pairs of values from the domains of the two related variables is $K \times K$. Therefore, the range of T is from 0 to $K \times K$. If T is 0, any pair of values is allowed and the constraint is actually a trivial constraint that is always satisfied. If T is $K \times K$, the relevant constraint can never be satisfied and no solution exists. The fraction of T over $K \times K$ is called the *constraint tightness* of the CSP problem.

When discussing problem size, we are referring to the parameters: N and K of the problem. All the problems generated using the same set of parameters: N, K, C and T are said to be of the same *problem class*.

4.1.2 The algorithms

The main purpose of our experiments is to compare the performance of FC and MAC. We will use both FC3 and FC4 as the algorithms of FC and both MAC3 and MAC7 as the algorithms of MAC. The pure FC and MAC algorithms do not perform well as the problems become larger and harder, so we also use some heuristics and optimizations discussed in Chapter 1 and Chapter 2. For all of these algorithms we use dynamic variable ordering with *dom* (choose the variable with the smallest remaining domain size) used as the ordering heuristic and *deg* (choose the variable with largest out degree in the initial constraint graph) used to break ties (see section 2.3 for details). For the FC algorithm, we also use the look-back scheme CBJ (conflict-directed backjumping, see section 1.2), denoted as FC-CBJ. We will not use CBJ for MAC; there is some empirical evidence that CBJ is not very effective with MAC [3]. We implemented various versions of the MAC algorithm. However, we found that the implementation of MAC7 discussed in [3] and made available by its authors had superior performance. It utilizes the singleton variable optimization discussed in Chapter 2. Finally, we also run the FC4 algorithm with initial arc consistency preprocessing, denoted as FC4-AC.

4.1.3 The empirical studies

We generated and solved 100 problem instances for each problem class. For different algorithms, we guaranteed that they solved exactly the same problem instances of a problem class by generating these problem instances from the same integer seed value.

We always report the median performance of an algorithm on the 100 instances solved for a problem class. We do not use the mean cost because of the existence of *exceptionally hard problems* (*ehps*) [5]. As a matter of fact, individual instances of a problem class may be very hard to solve with a particular algorithm while most other instances of the same problem class are easy to solve. In this case, the cost of these ehps significantly affects the value of the mean cost, making the median a better indication of average difficulty. Individual ehps are highly dependent on the algorithm being used [5] and their hardness doesn't reflect the hardness of the whole problem class. The incidence of ehps is not the interest of this thesis. We focus on the performance of different algorithms on a whole problem class.

Since the number of constraint checks is not an appropriate measure for FC4 or MAC7 (the only constraint checks are done to set up support or non-support sets), we use CPU time to measure their performances and make comparisons. We still use the number of constraint checks when applicable.

It is claimed in both [3] and [5] that MAC outperforms FC in solving large and hard problems. However, in [3], experiments were done on individual problem classes with no apparent relationship among them. In [5], experiments were only done on problems of a certain constraint density. In this thesis, we want to provide a more systematic study. That is, we want to show the performances of FC and MAC in relation to the size, constraint density and constraint tightness of the problems solved.

The problems we use for our experiments must meet the following requirements:

- These problems are hard enough to show the difference among the performances using different algorithms;
- These problems are easy enough to be solved in reasonable time.

It would not be difficult to find individual problem classes having the above properties. However, if we want to have a group of problem classes with relating size, constraint density and constraint tightness, the answer is not trivial.

We start our experiments with CSPs with $N = 30$ and $K = 10$. A wide range of C and T's are experimented to give an overall view of the distribution and hardness of CSPs with varying C and T when the problem size is fixed, and the performances of FC and MAC on them. Then N is increased to show how the problem size will affect the relative performances of FC and MAC, in particular, with regard to C and T.

Experiments are conducted by generating groups of problem classes with varying C whilst holding N, K and T constant and using different algorithms to solve them. We can plot the median behavior of these algorithms with regard to C. However, we will not use C as the x-axis. Instead, we will use the local constraint density: γ, which is defined as $(2 \times C)/N$. In fact, γ is the average degree of all nodes in the constraint graph of a CSP which represents the average number of constraints each variable has. It is indicated in [5] that the effect of increasing N is independent if we maintain the problem topology by holding γ constant. In other words, we can expect that if we increase N whilst holding K, γ and T constant, the difference of the performances of the same algorithm is only caused by N. We will verify this conclusion later in our experiments. Like C, the range of γ is also dependent on the value of N. It is from 0 to $N - 1$. When γ is $N - 1$, every variable is constrained with all other variables in the problem.

4.2 Results

We only use the FC3 and MAC3 algorithms for our first set of experiments. Since they both use AC3 as the algorithm to achieve arc consistency, we are able to use the number of constraint checks as the measurement for their performances. Figure 4.1–4.9 show their median behaviors in terms of number of constraint checks on the $N = 30$ and $K = 10$ problem classes.

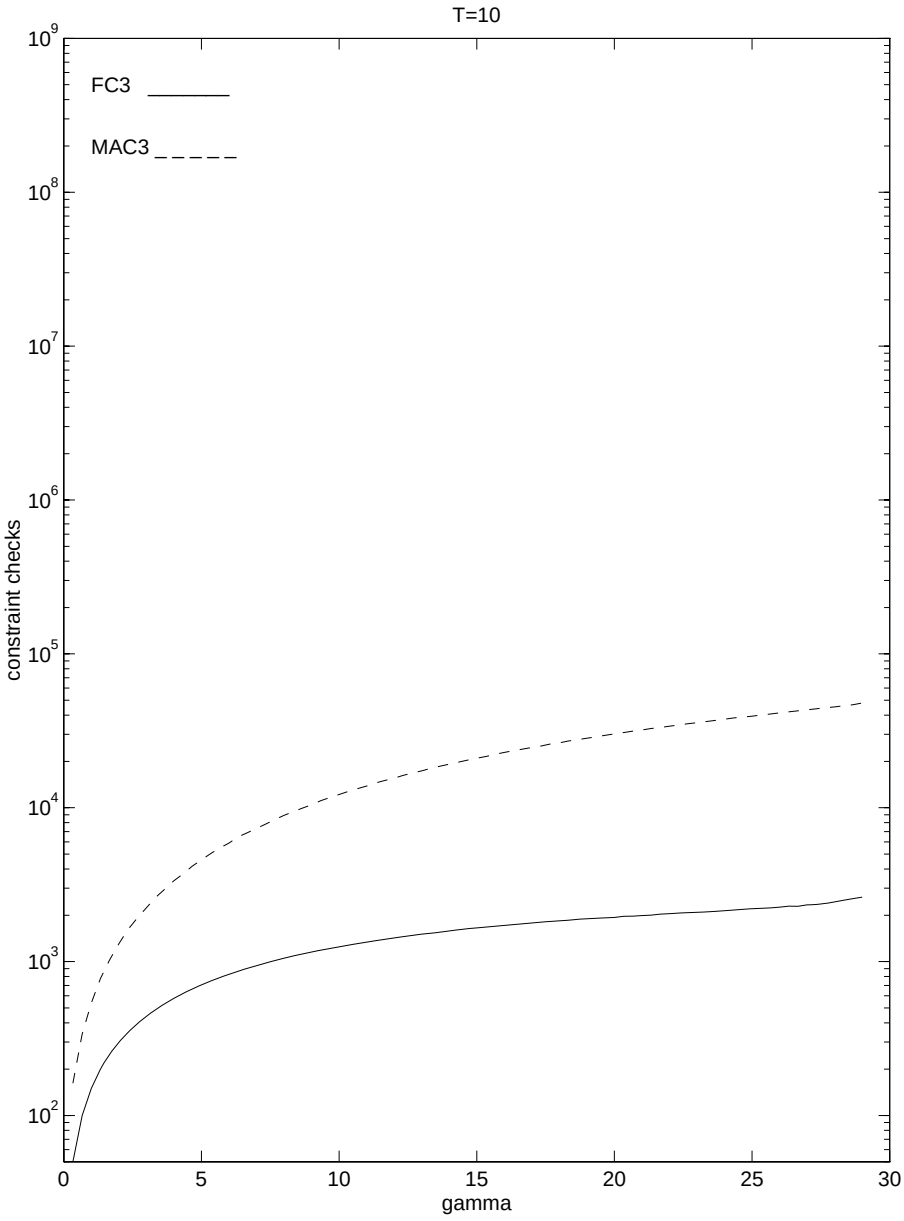

Figure 4.1: N=30, K=10

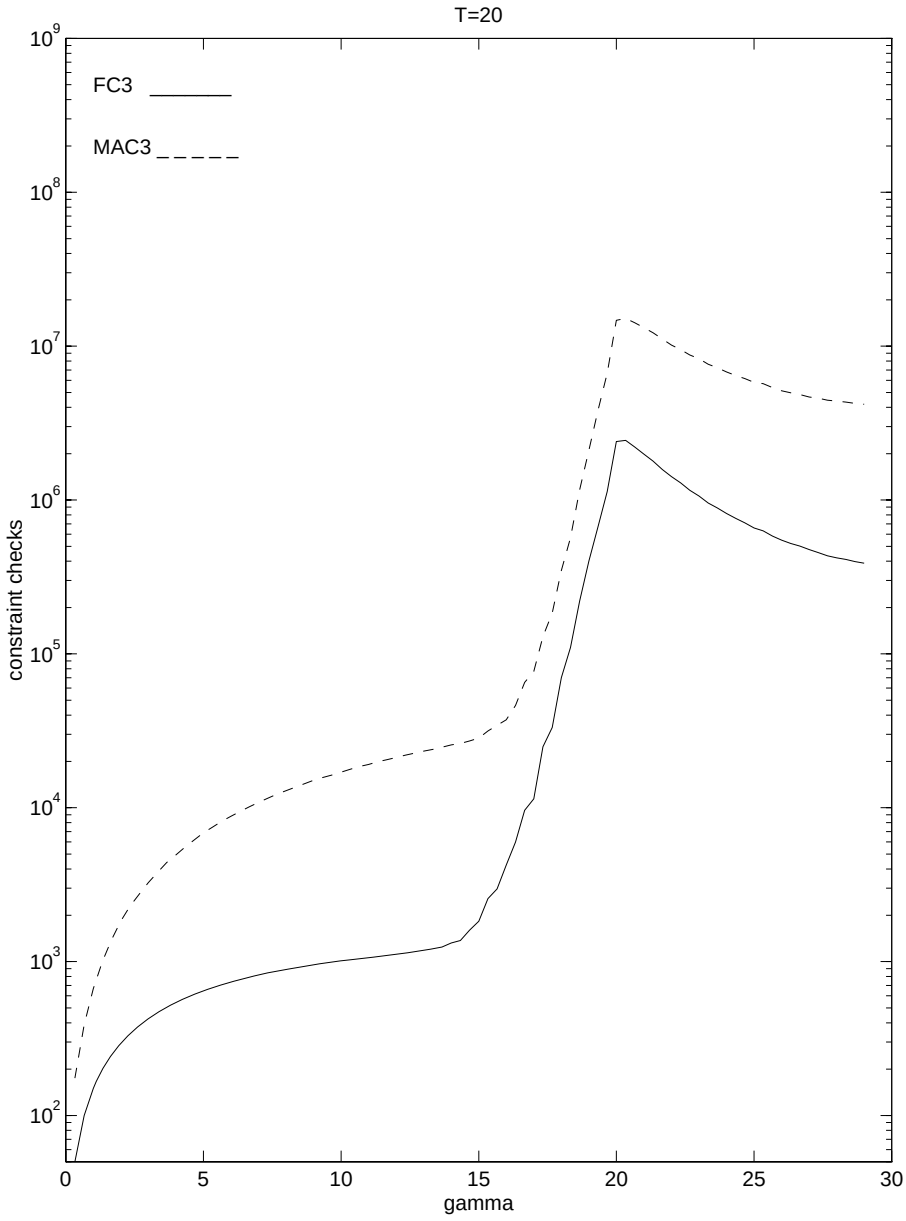

Figure 4.2: N=30, K=10

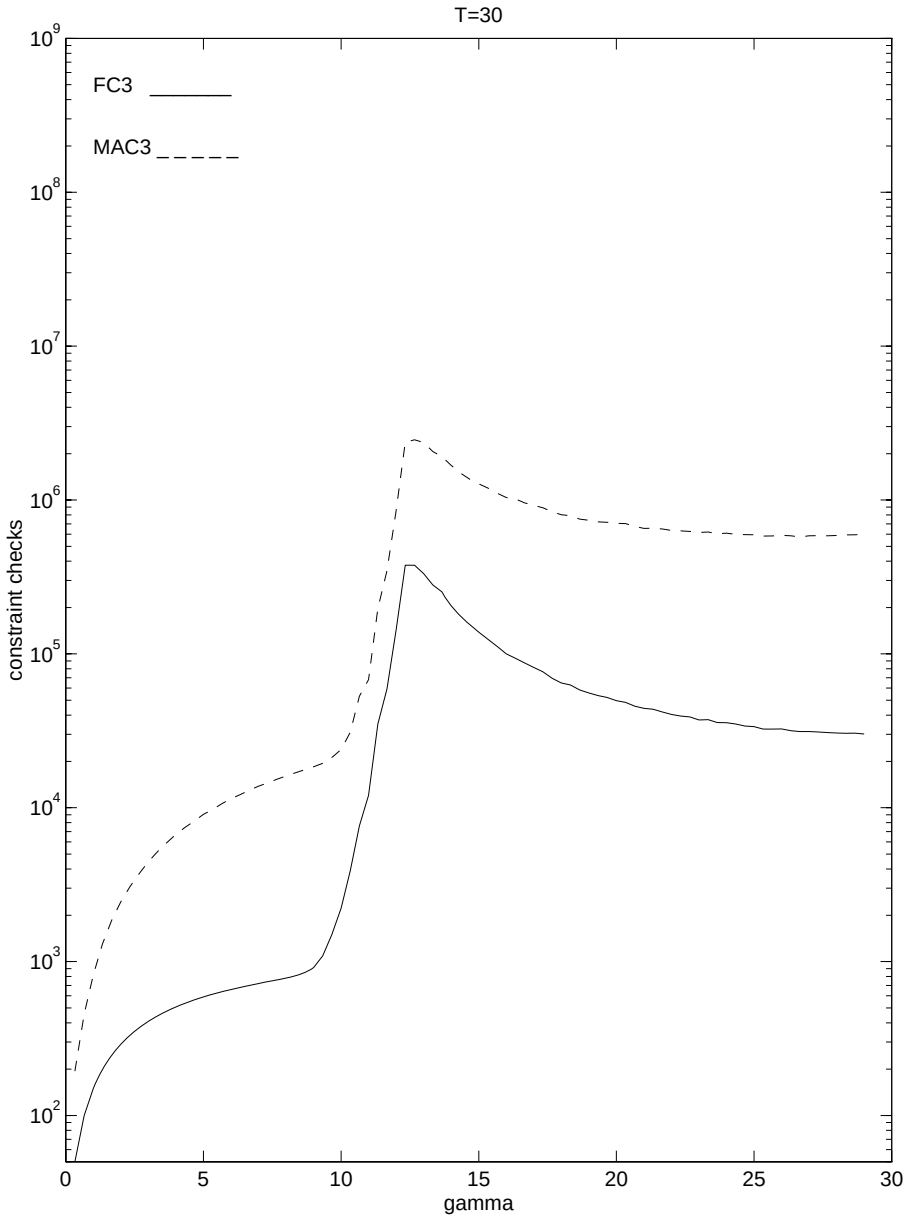

Figure 4.3: N=30, K=10

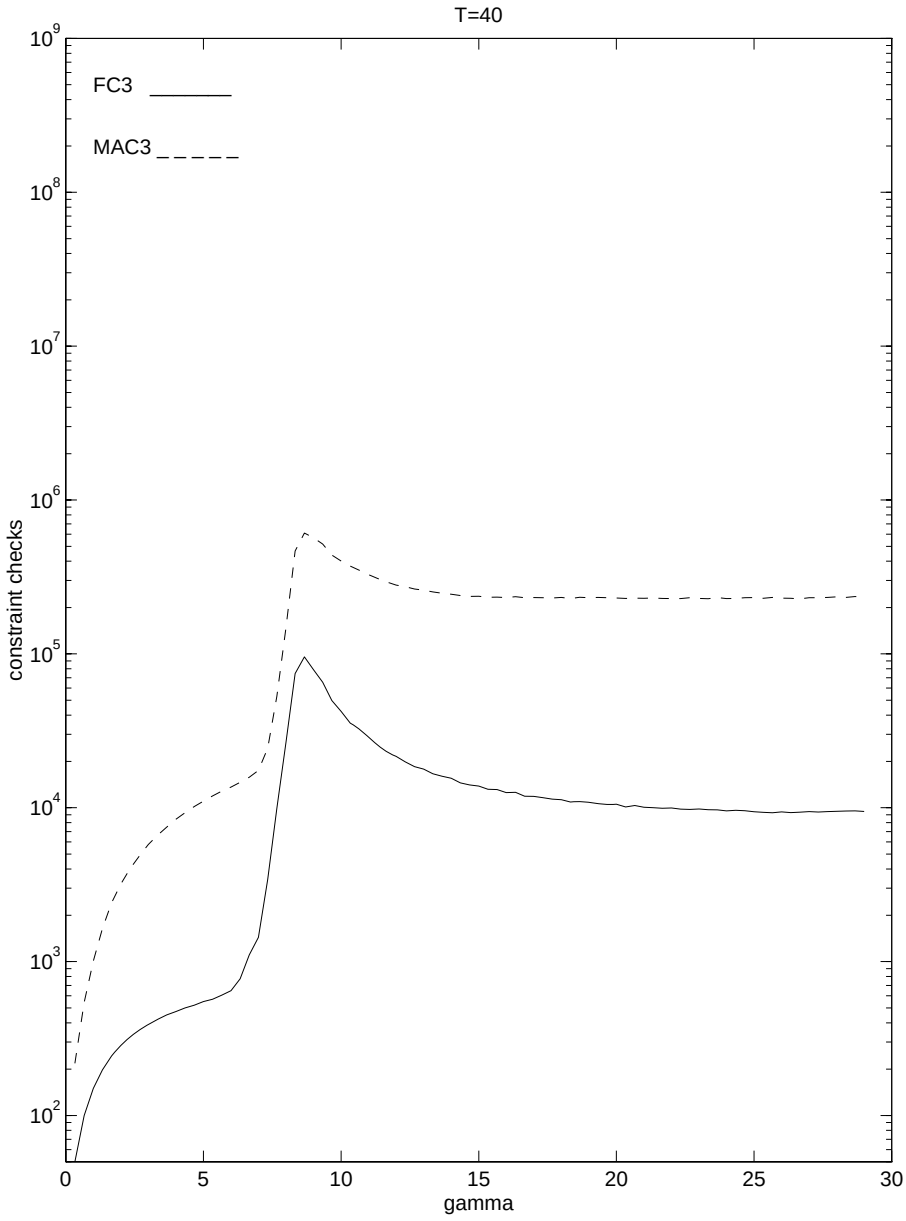

Figure 4.4: N=30, K=10

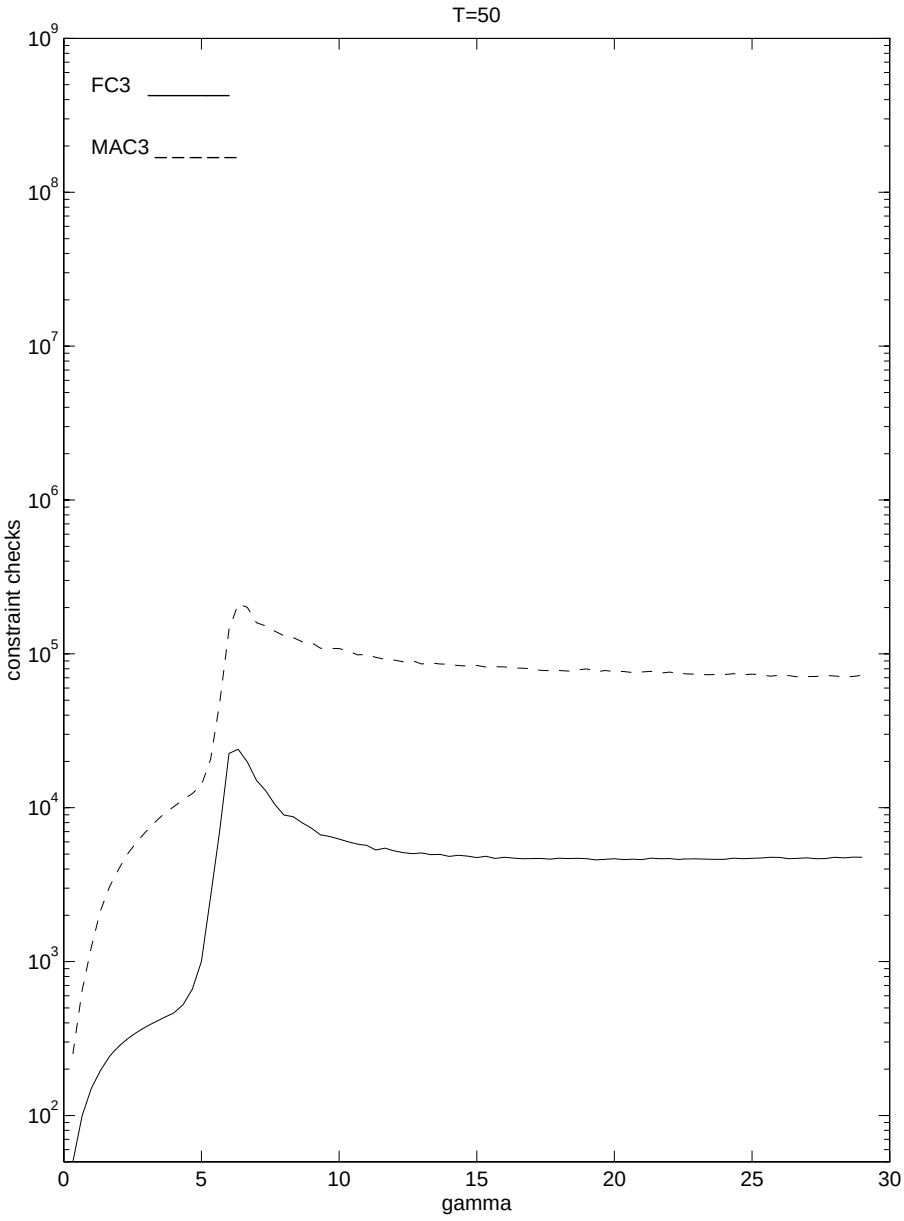

Figure 4.5: N=30, K=10

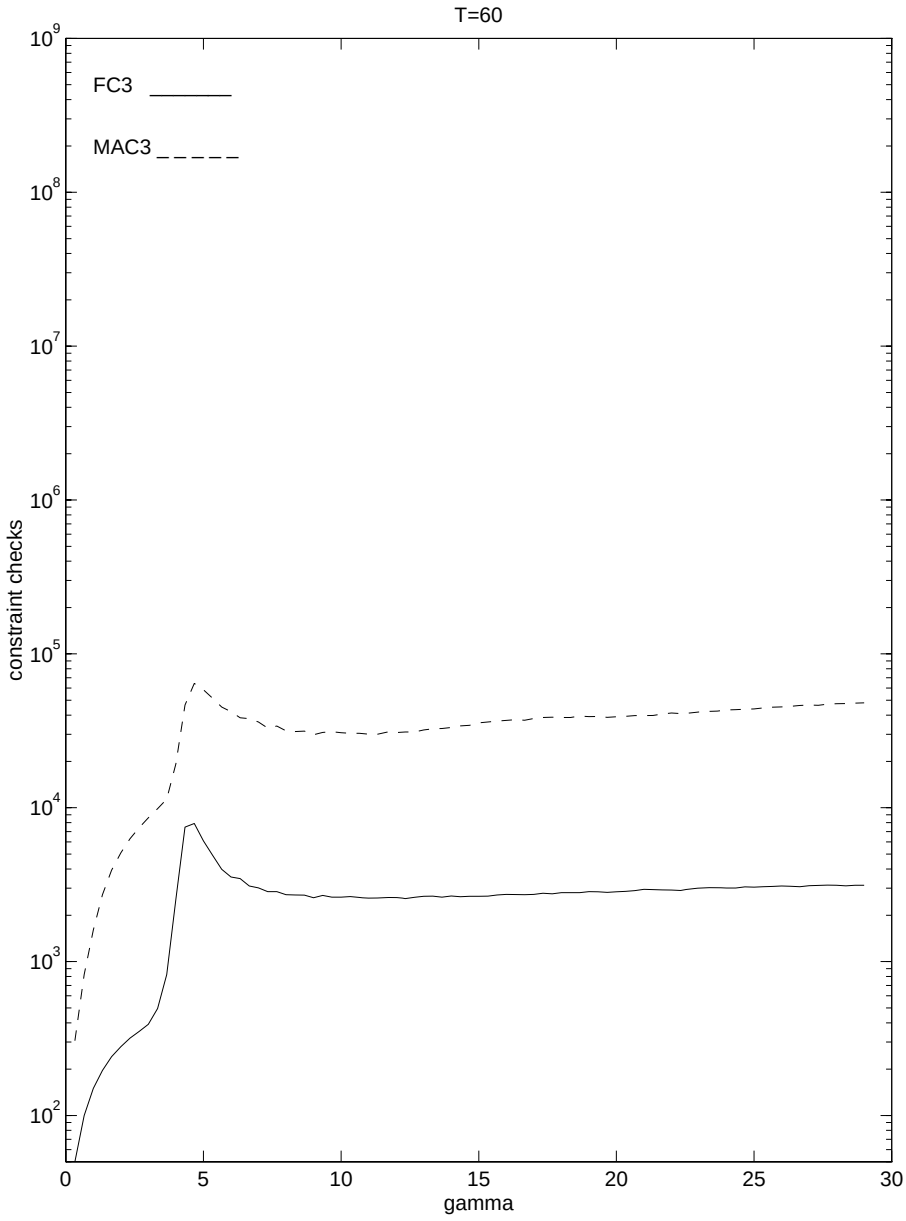

Figure 4.6: N=30, K=10

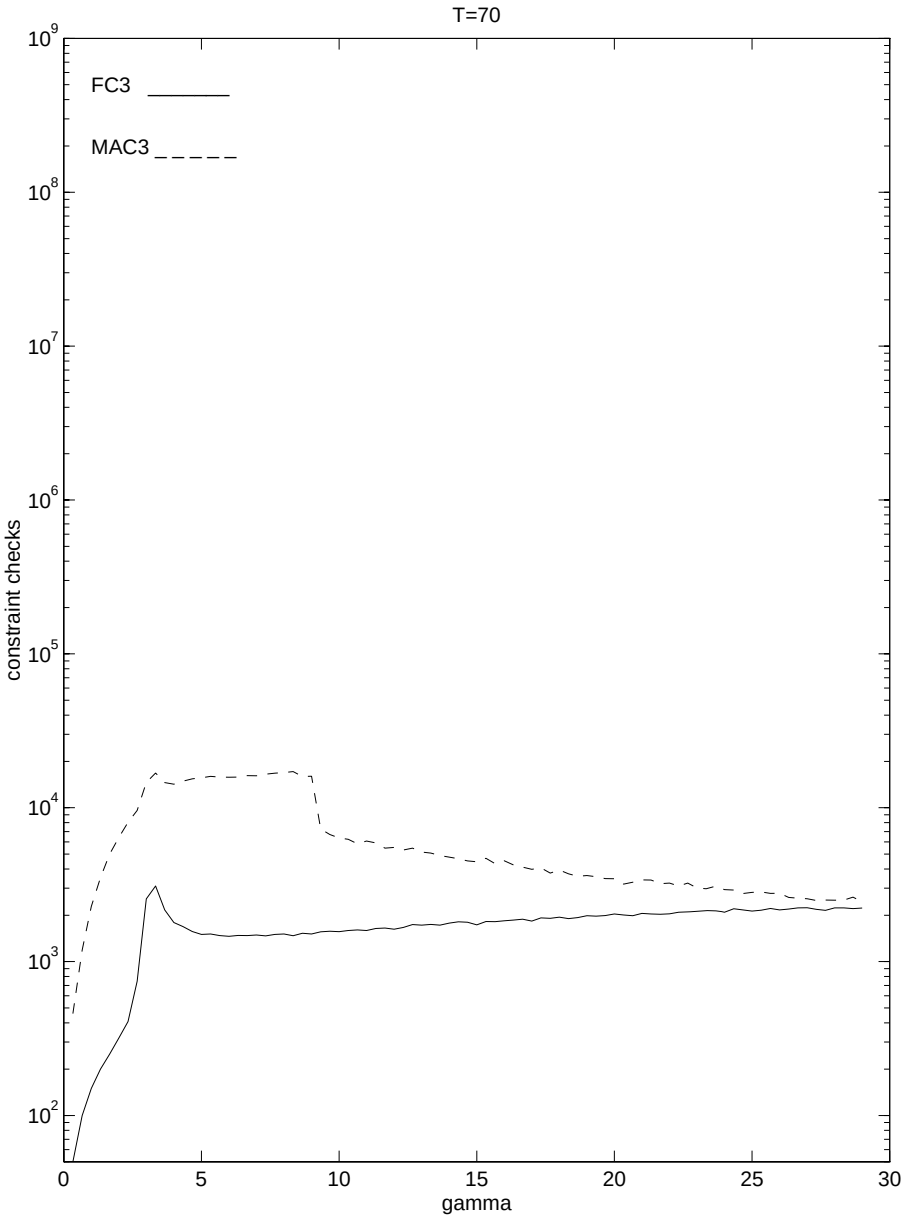

Figure 4.7: N=30, K=10

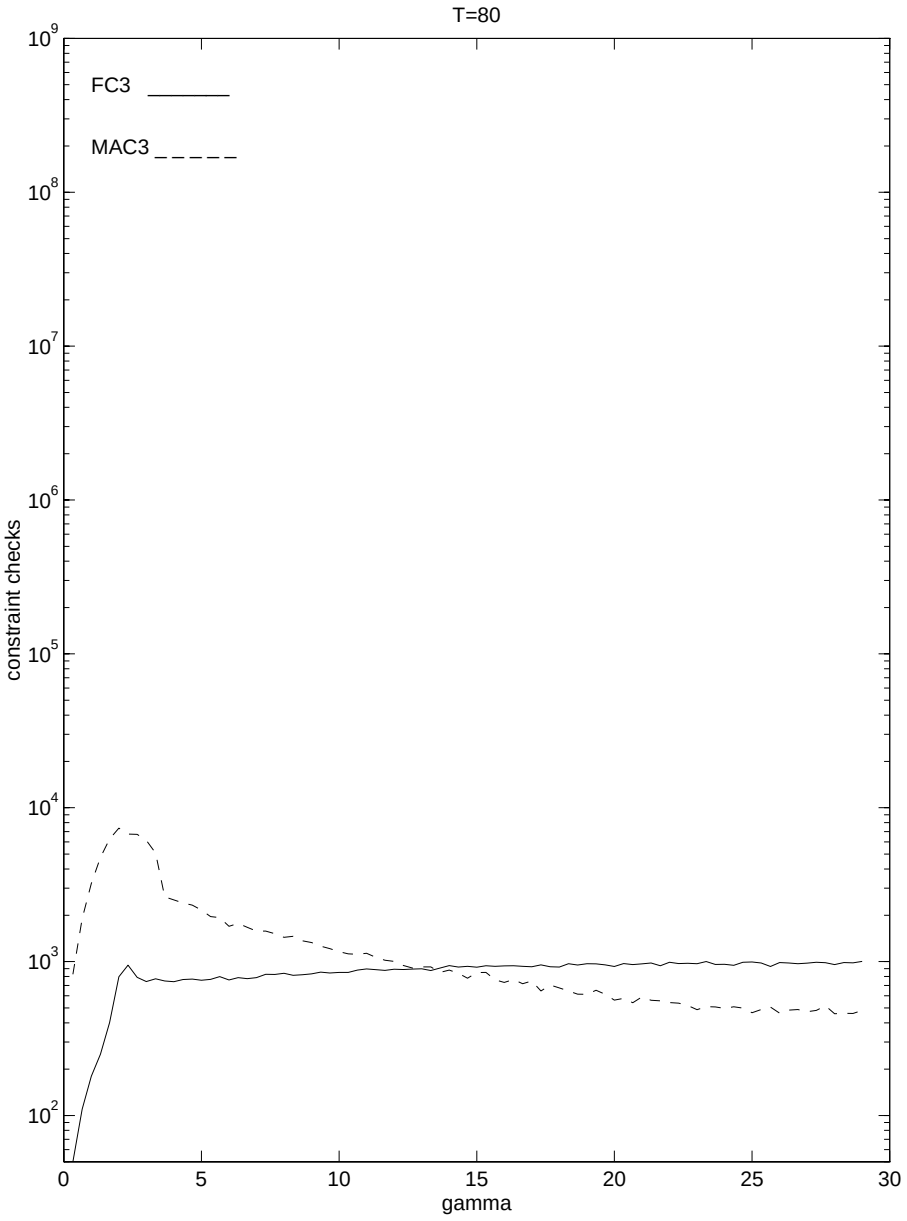

Figure 4.8: N=30, K=10

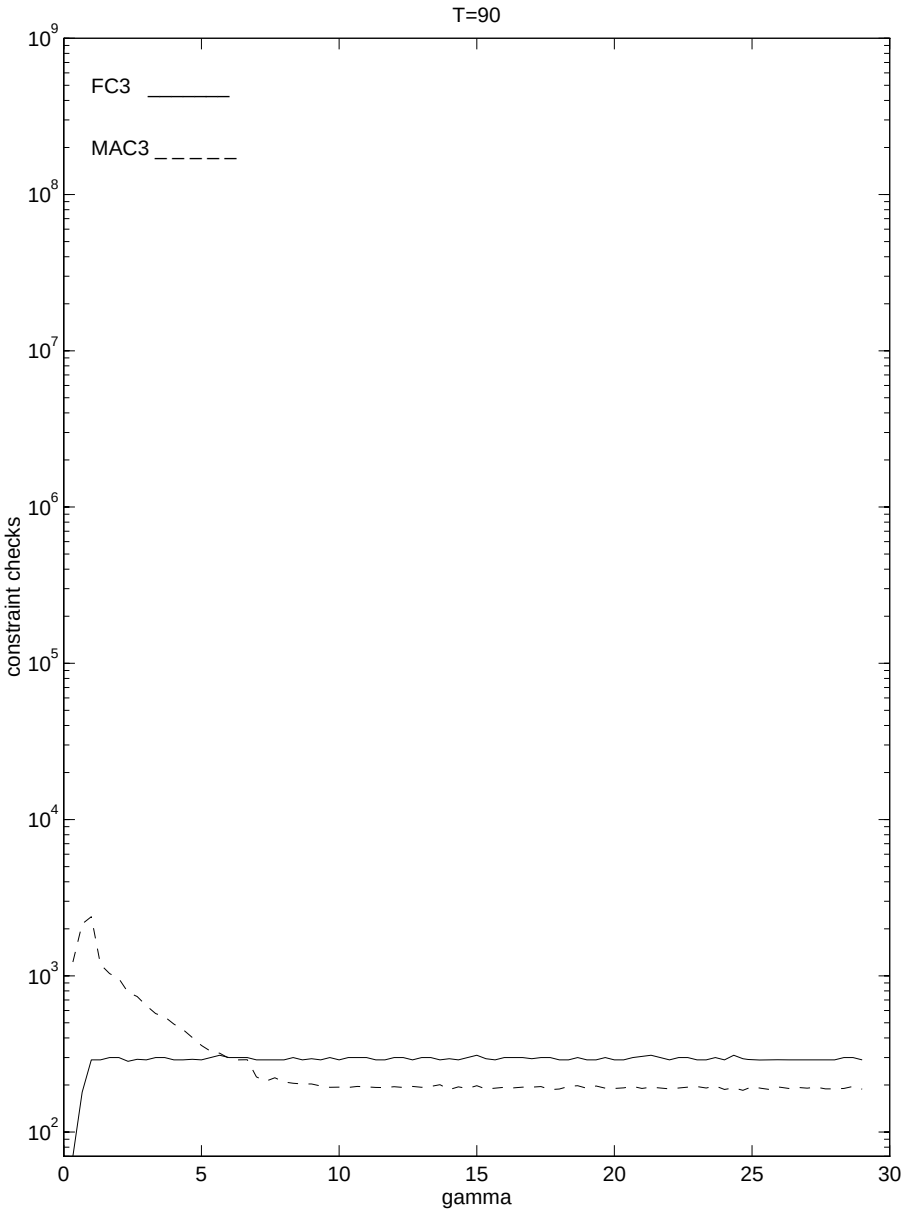

Figure 4.9: N=30, K=10

From these graphs, we can see that:

- The qualitative behavior of FC and MAC is similar. In particular, the peaks of both algorithms almost always appear at the same value of C for a certain T. This suggests that in general, a hard problem class for FC is also a hard problem class for MAC, and *vice versa*.

- When the constraint tightness is low, for both algorithms, the hardest problem class appears where the constraint density is high. When the constraint tightness is high, the hardest problem class appears where the constraint density is low.

  This is not difficult to understand. If both constraint density and tightness are low, a variable is unlikely to be constrained with many other variables and even if it is, most of their values will be compatible. The problems of this class are likely to have many solutions and it is easy to find one. On the other hand, if both constraint density and tightness are high, it usually only takes a few variable assignments to generate a contradiction. Thus these algorithms fail near the top of the backtracking tree, and they do so quite quickly. In other words, the problems of this class are likely to have no solution and their insolubility is relatively easy to prove.

  Only when the constraint density and tightness maintain a certain kind of balance such that some of the problems are soluble and some are insoluble, is the cost of search high. In this case, for the soluble problems, solutions are not easy to find and for the insoluble problems, it is difficult to detect a dead end. In fact, it has generally been observed [5] that the hardest problem classes are those containing about 50% soluble problems and 50% insoluble problems.

- For both algorithms, of all the peaks (the hardest problem classes) with regard to different T, the highest one (Figure 4.2) appears when the constraint tightness is relatively low ($T = 20$ out of 100) and the constraint density is relatively high ($C = 305$ out of 435 or $\gamma = 20.33$ out of 29). In fact, the peak values are getting larger and larger with decreasing T until T is very small (< 20).

- FC3 almost always outperforms MAC3 (because these problems are not large and hard enough). The only exceptions occur when T is very high ($T = 80, 90$) and most of the problems are initially arc inconsistent provided that C is reasonably large. In this case, MAC can detect the insolubility of a problem without any variable assignments.

Since MAC is supposed to work better on large problems, we expect that MAC will outperform FC if we increase N, i.e., increase the size of the problems. Moreover, since MAC is supposed to work better on hard problems, we expect that with increasing N, MAC will start to outperform FC first on the problem class of the value C that makes it the hardest given N, K and T. The following experiments are performed to examine how MAC starts to outperform FC. Figure 4.10–4.16 give the results.

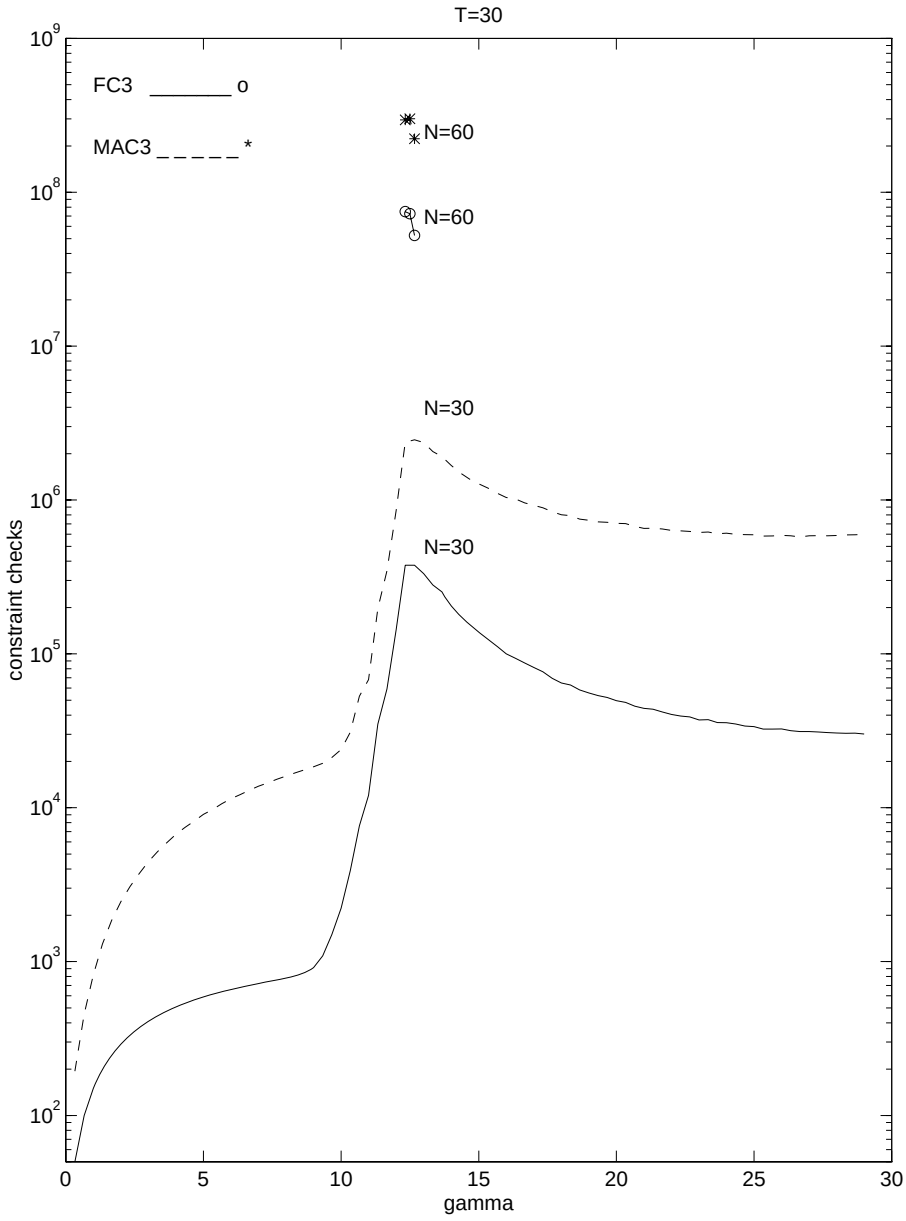

Figure 4.10: K=10 with increasing N

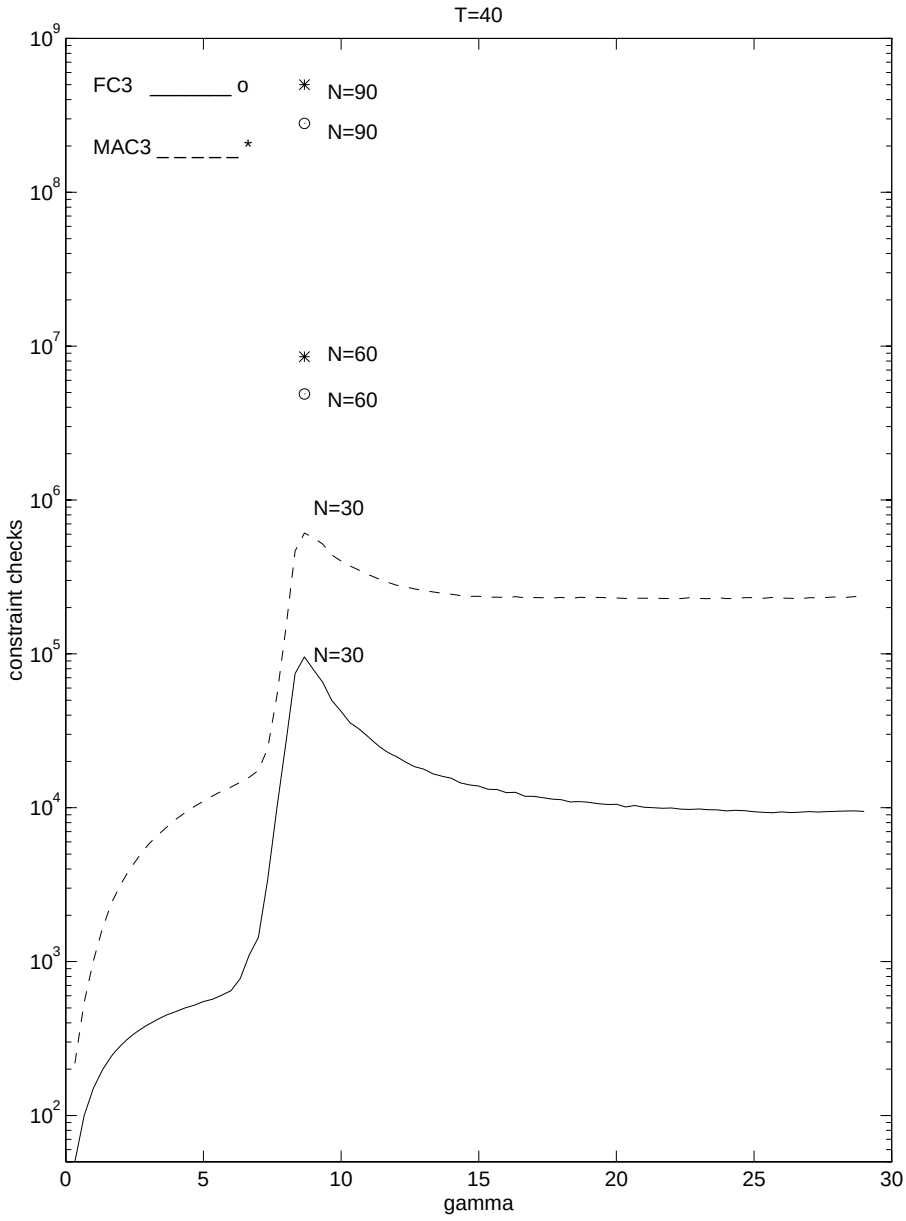

Figure 4.11: K=10 with increasing N

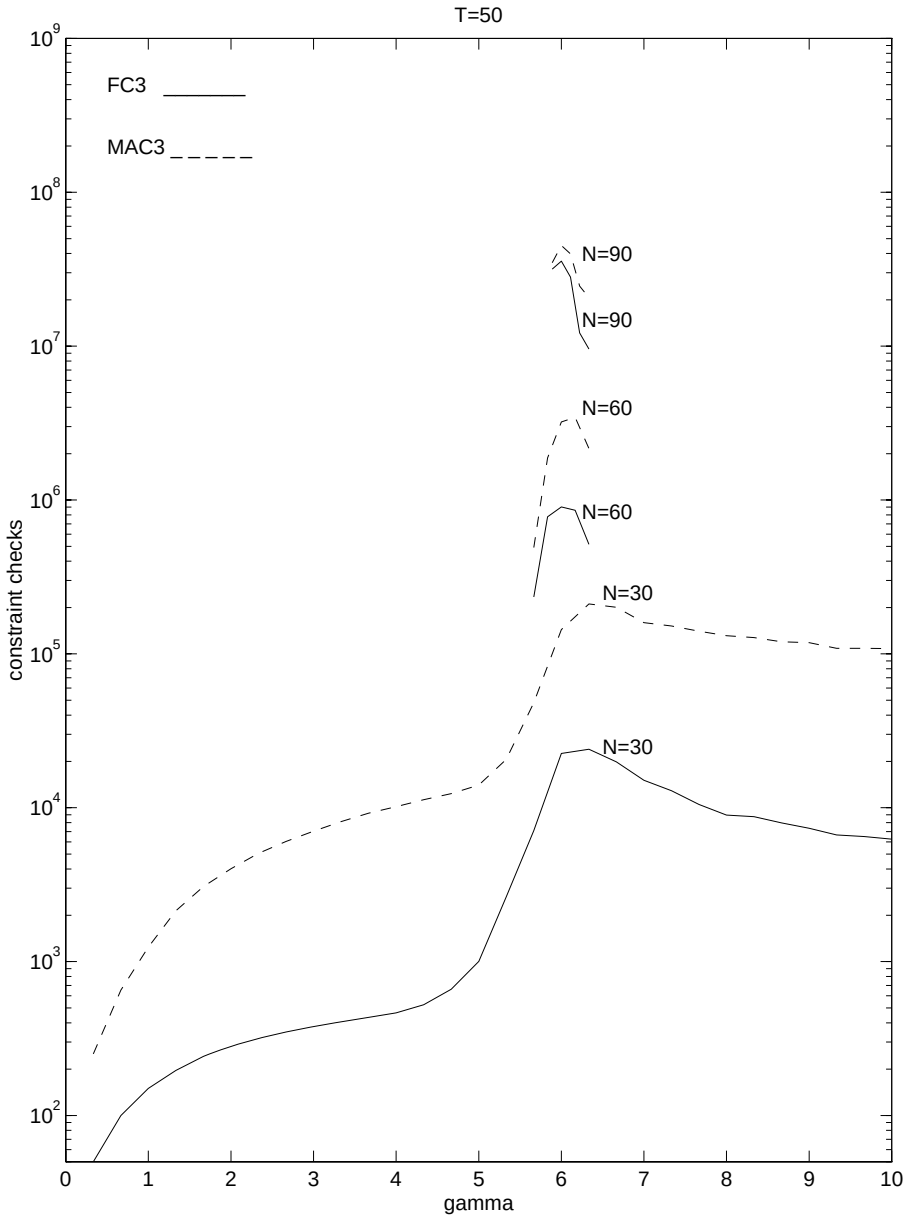

Figure 4.12: K=10 with increasing N

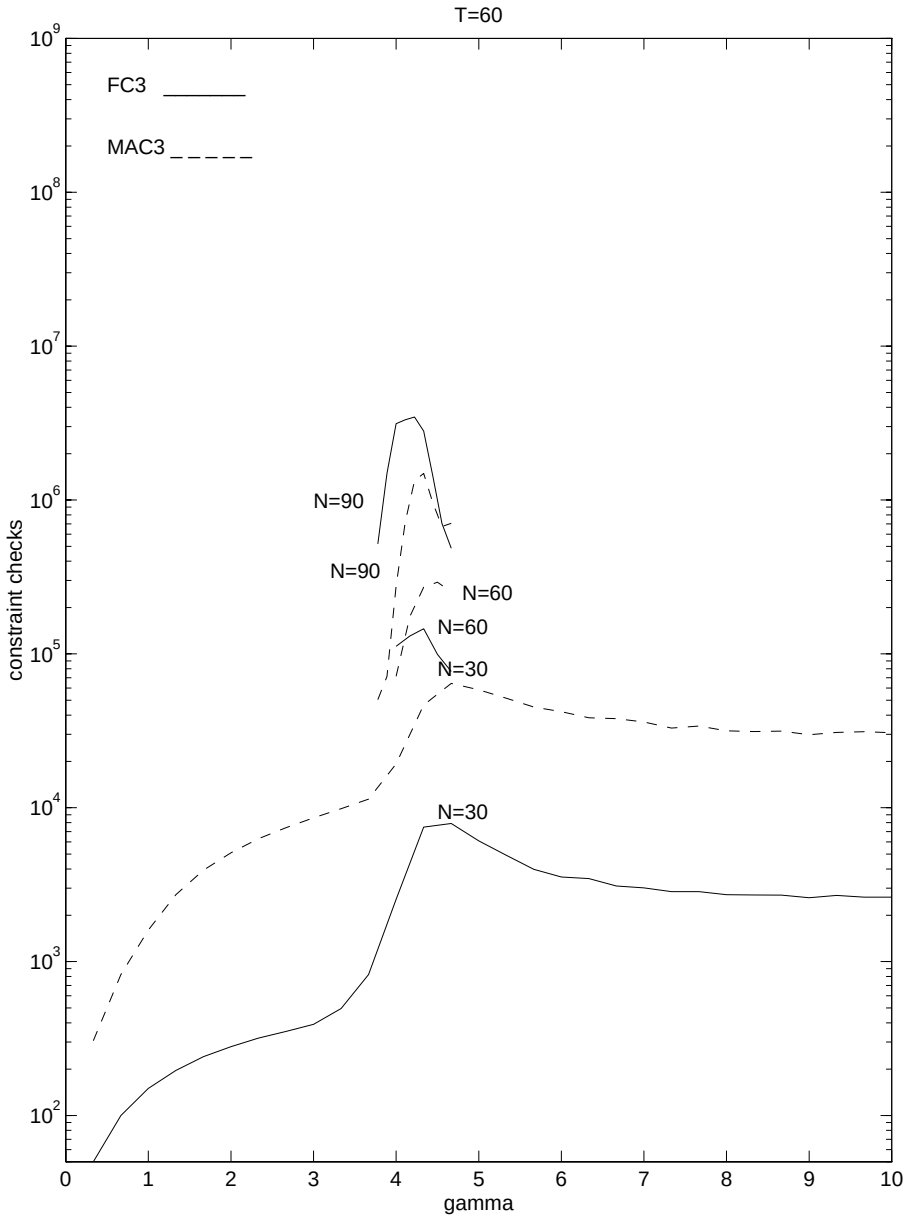

Figure 4.13: K=10 with increasing N

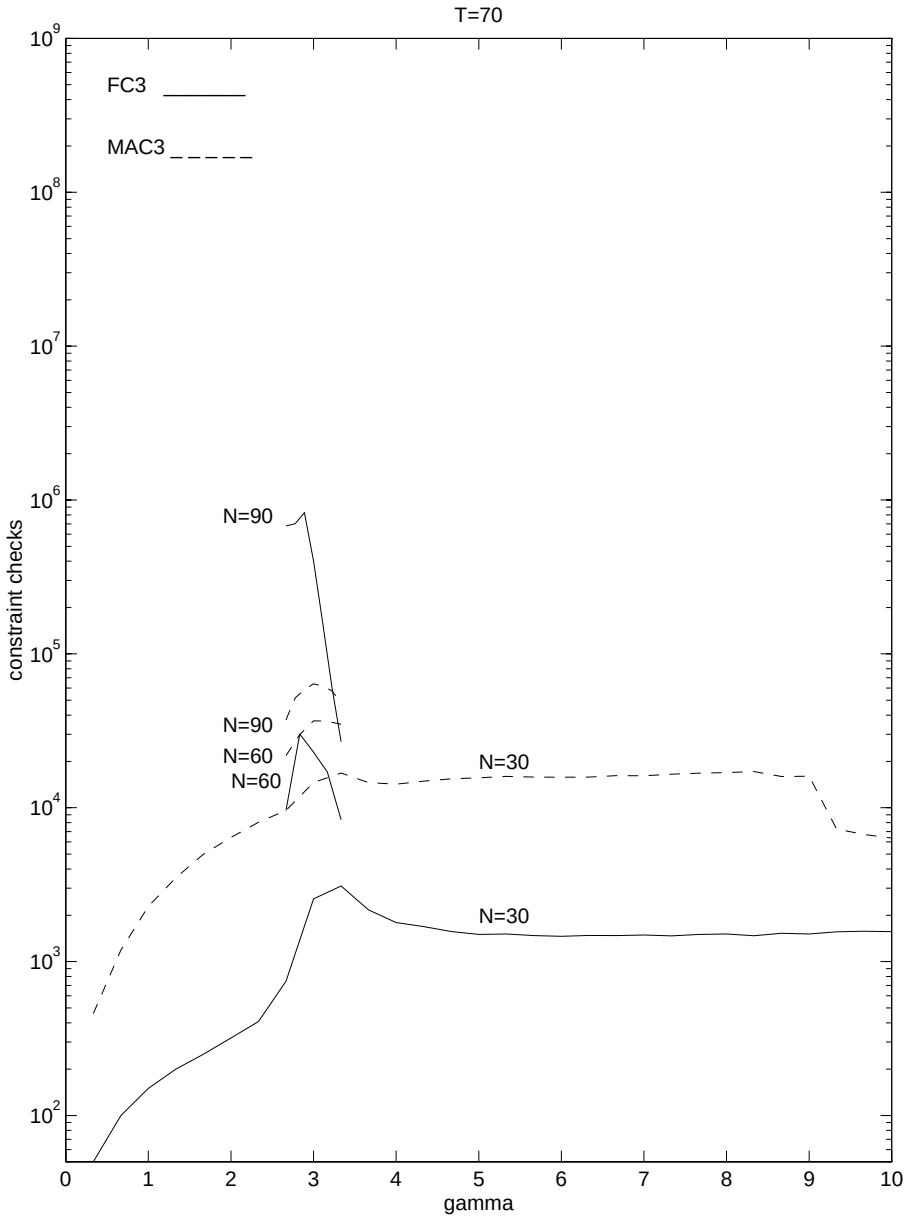

Figure 4.14: K=10 with increasing N

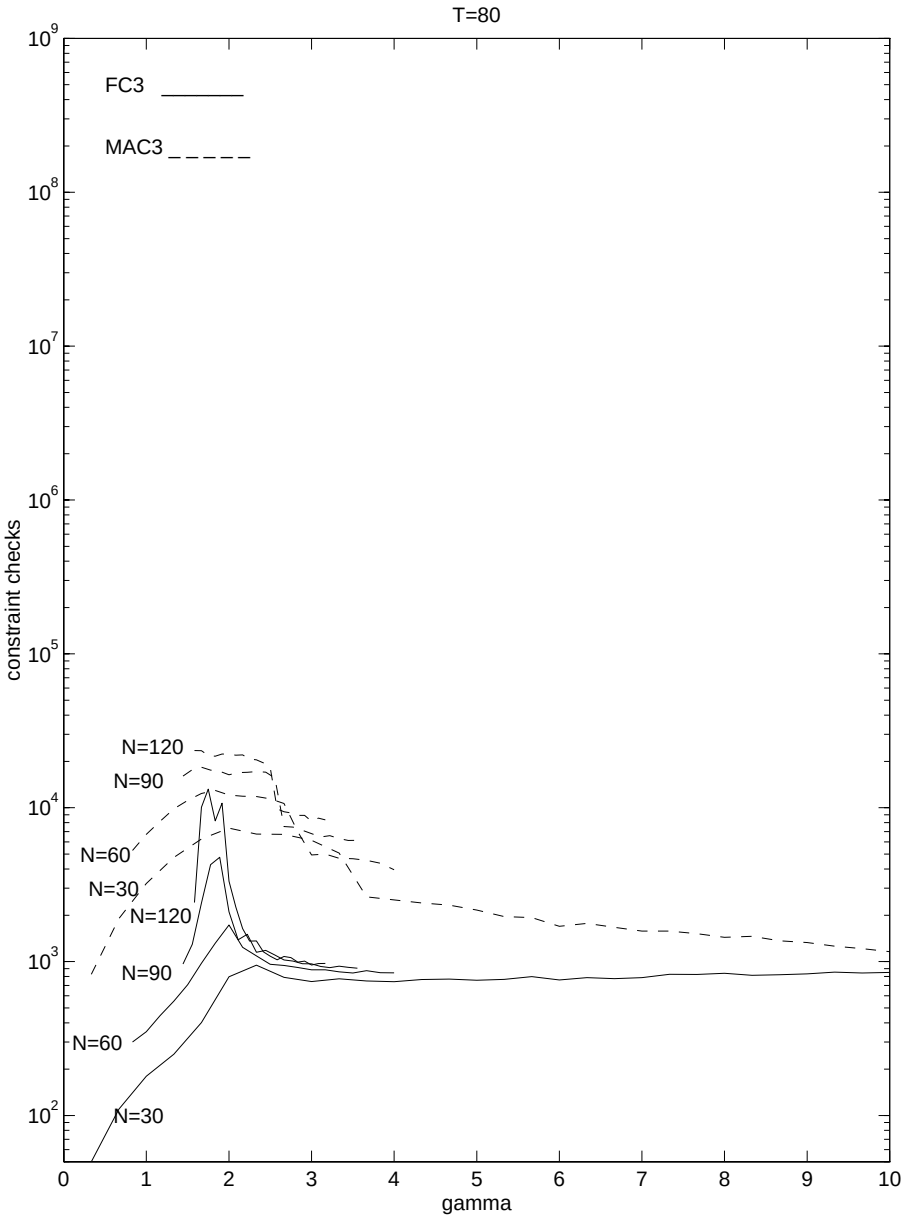

Figure 4.15: K=10 with increasing N

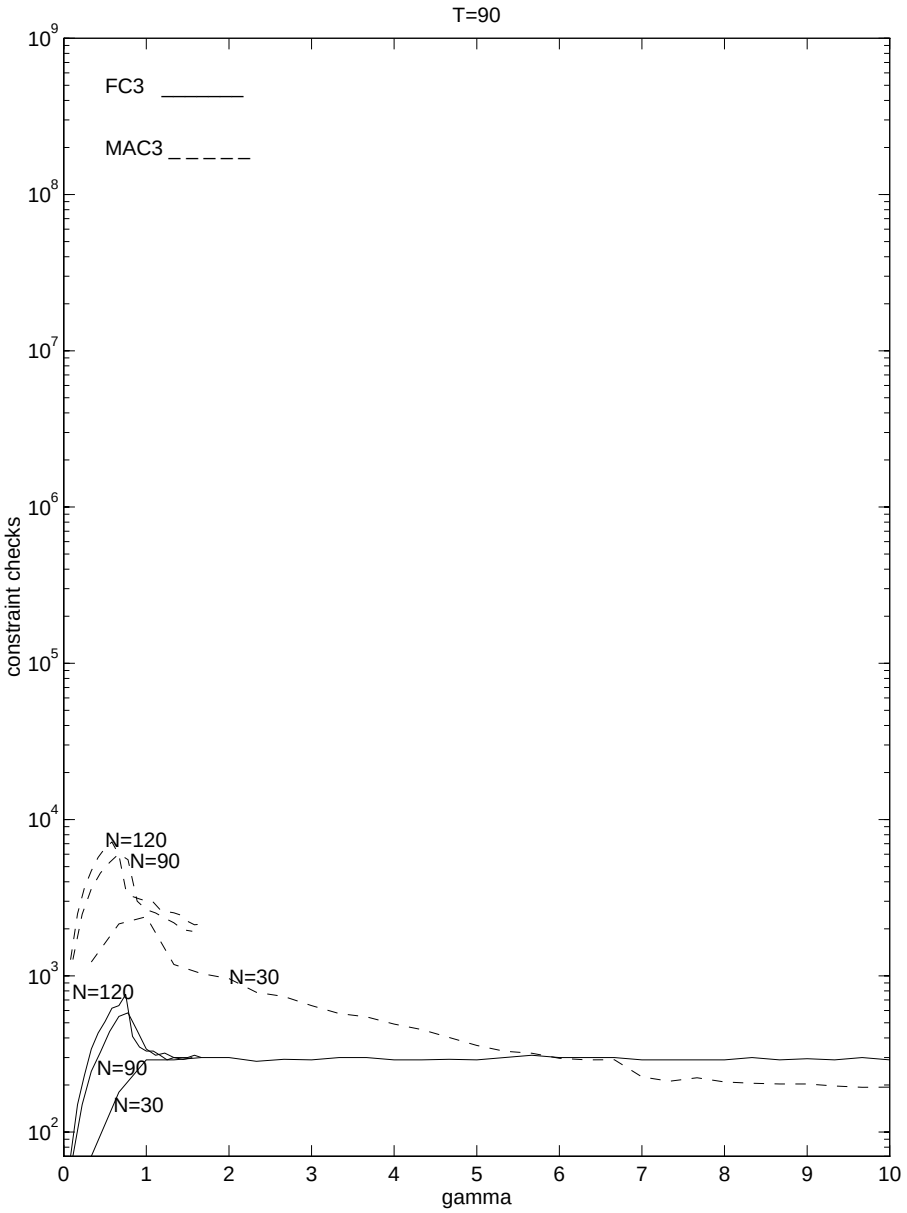

Figure 4.16: K=10 with increasing N

We enlarge the graphs for $T = 50, 60, 70, 80, 90$ to make them more clear to be read. The x-axis for these graphs is [0,10]. Though there is a general trend that as N increases there is a point where MAC starts to outperform FC on hard problem classes, however, we were only able to obtain this result for some values of T; as for other values of T, the value of N at which MAC might outperform FC becomes too large and the problems become too hard for us to complete the experiments. For example, it takes thousands of seconds of CPU time to solve one hard problem as $N = 90, K = 10$ and $T = 40$.

For $T = 60, 70$, we did get the result that MAC outperforms FC as N increases. For $T = 70$, FC always outperforms MAC as $N = 30$. When we increase N to 60, their performances on hard problem classes are close. When we increase N to 90, MAC is more than an order of magnitude better than FC on hard problem classes. For $T = 60$, we have the same result, though the growth rate of the superiority of MAC over FC on hard problem classes is less than that for $T = 70$. These two results clearly show that for a certain value of T, MAC performs better on large and hard problems.

In both of the two graphs, the fact that all those peaks for different values of N appear at almost the same value of γ also suggests that for these two values of T, the effect of increasing N is independent if we hold γ, instead of holding C, constant. In fact, this is general for all the values of T from 30 to 90 for which we have graphs with increasing N. This is the reason why we use γ as the x-axis for these graphs.

For $T = 50, 80$, though we did not obtain the result that MAC outperforms FC, we can see such a trend. When $N = 90$, in both the graph of $T = 50$ and the graph of $T = 80$, the performances of MAC and FC get so close that it can be expected to occur with further increase of N.

The fact that as we increase N to a certain value, say, $N = 90$, for some values of T, say, $T = 60, 70$, MAC outperforms FC on hard problems, while for other values of T, say, $T = 50, 80$, MAC does not outperform FC, suggests that the point of the value of N at which MAC might start to outperform FC is different on different problem classes with different values of T.

For $T = 40$, we were only able to test one problem class for both $N = 60$ and $N = 90$. Though we do not know whether this problem class is the hardest since we have no other problem classes to compare with it, it is of the same value of γ which is used to generate the hardest problem class as $N = 30$. Therefore, this problem class is at least a hard problem class. Following is the data we obtained:

N	30	60	90
γ	8.67	8.67	8.67
C	130	260	390
Constraint Checks (MAC3)	610274	8530290	500002000
Constraint Checks (FC3)	95591	4888620	280530000
Ratio of Constraint Checks	6.3842	1.7449	1.7823

We can see that from $N = 30$ to $N = 60$ there is a big increase of the performance of MAC compared with FC. But from $N = 60$ to $N = 90$, their relative performances are almost the same.

For $T = 30$, we were only able to test several problem classes as $N = 60$. We cannot see many changes in the relative performances of FC and MAC.

For $T = 90$, we tested up to $N = 120$, but still could not get our expected result. In general, as we mentioned earlier, given a problem size, the problem classes of high constraint tightness are relatively easy. Therefore, the problem classes with $T = 90$ will not be very hard with increasing N until N is very large.

It is near $T = 70$ that MAC is most likely to outperform FC. In general, MAC tends to perform better on hard problems of relatively high constraint tightness. This phenomenon can be explained as follows:

- If tightness is high, more values are likely to be pruned due to arc inconsistency. Thus arc consistency processing is more useful in reducing the size of the search tree.
- The hard problem classes with high constraint tightness must have low constraint density. Thus, fewer arcs need to be checked during the propagation of a removed value.

As we mentioned earlier, for a certain value of T, e.g., $T = 70$, compared to FC, MAC really works better on large and hard problems. But how about if we compare the performance of MAC and FC on problems with different values of T? Following are the results we got on three problem classes with different values of T.

1. $N = 90, K = 10, T = 70, C = 135$, see Figure 4.14

Algorithm	Constraint checks
FC3	401114
MAC3	63878

2. $N = 90, K = 10, T = 40, C = 390$, see Figure 4.11

Algorithm	Constraint checks
FC3	280530000
MAC3	500002000

3. $N = 120, K = 10, T = 90, C = 45$, see Figure 4.16

Algorithm	Constraint checks
FC3	760
MAC3	3574

If we compare between problem class1 and problem class2, they are of the same problem size and, clearly, problem class2 is much *harder* than problem class1. However, MAC outperforms FC on problem class1, which is easier, but does not on problem class2, which is harder. This is an example that MAC does not always outperform FC on hard problems.

If we compare between problem class1 and problem class3, problem class3 is *larger* than problem class1. However, MAC outperforms FC on problem class1, which is smaller, but does not on problem class3, which is larger. This is an example that MAC does not always outperform FC on large problems.

Therefore we cannot say that MAC always works better than FC on large and hard problems. The performance of these algorithms is also related to the constraint density and tightness of the problems.

Here is a summary:

- For hard problems of a certain tightness, MAC tends to work better compared to FC as the problem size increases.
- For hard problems of different tightness, the point at which MAC starts to outperform FC with increasing problem size is different.
- The superiority of MAC over FC is most likely to be revealed on hard problems with relatively high tightness (around $T = 70$ out of 100).

The purpose of all the above experiments is to give a general trend of the performance of both the FC and MAC algorithms with regard to the changes of N, C and T. However, since these experiments are done using the FC3 and MAC3 algorithms, which are not supposed to be the best algorithms of FC and MAC respectively, now we repeat these experiments using the improved versions of FC and MAC. For the FC algorithm, we use FC3-CBJ, FC4-CBJ and FC4-CBJ-AC. For the MAC algorithm, we use MAC7. Since these algorithms are supposed to be faster, we can hopefully have more results that MAC outperforms FC.

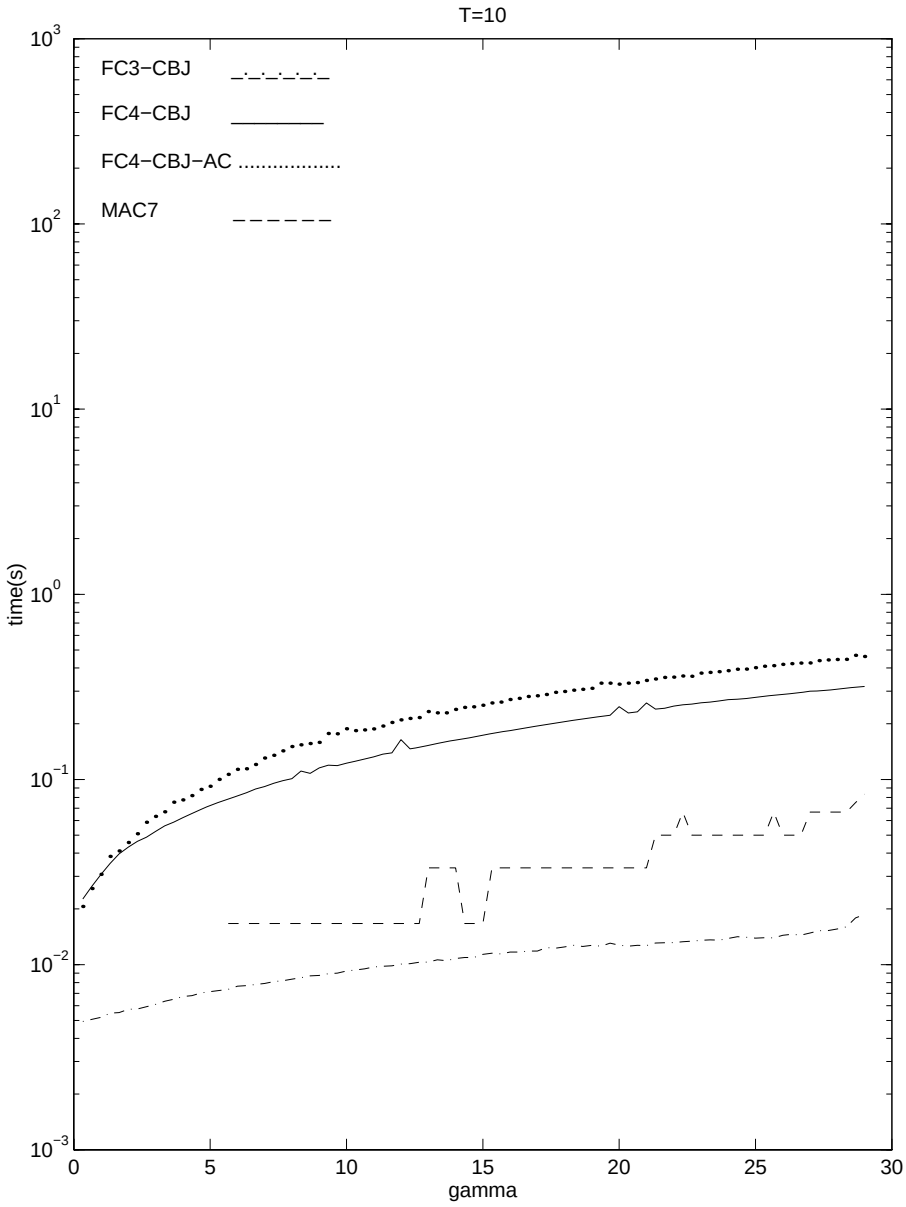

Figure 4.17: N=30, K=10

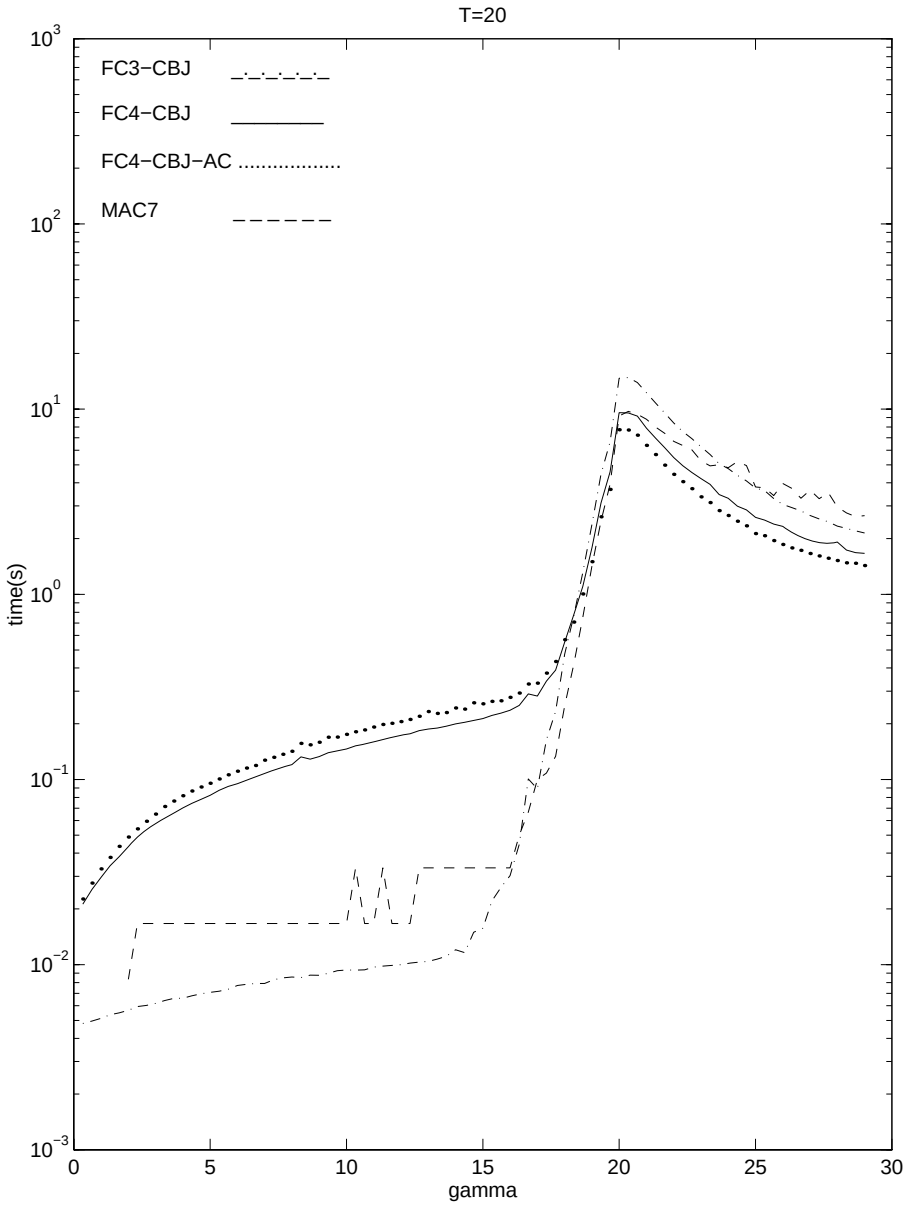

Figure 4.18: N=30, K=10

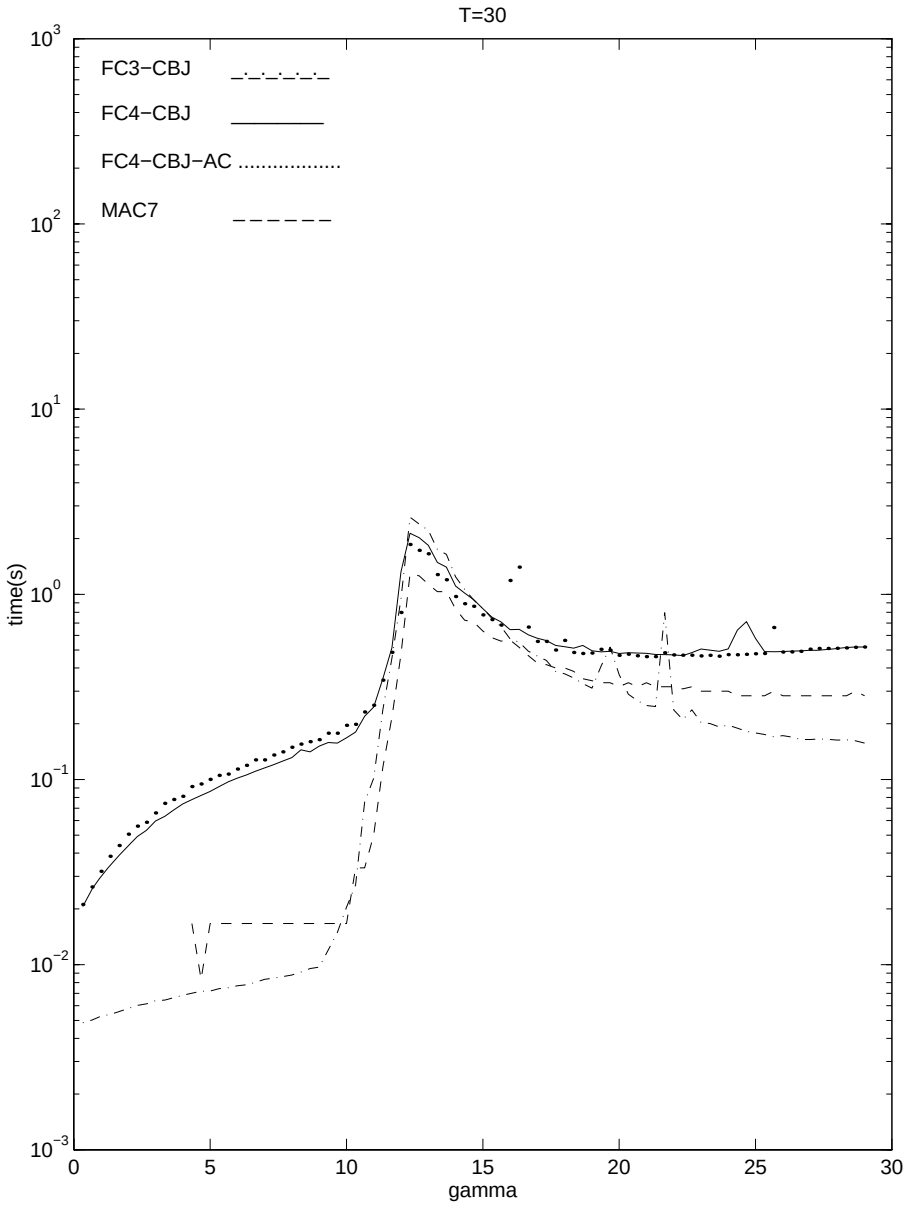

Figure 4.19: N=30, K=10

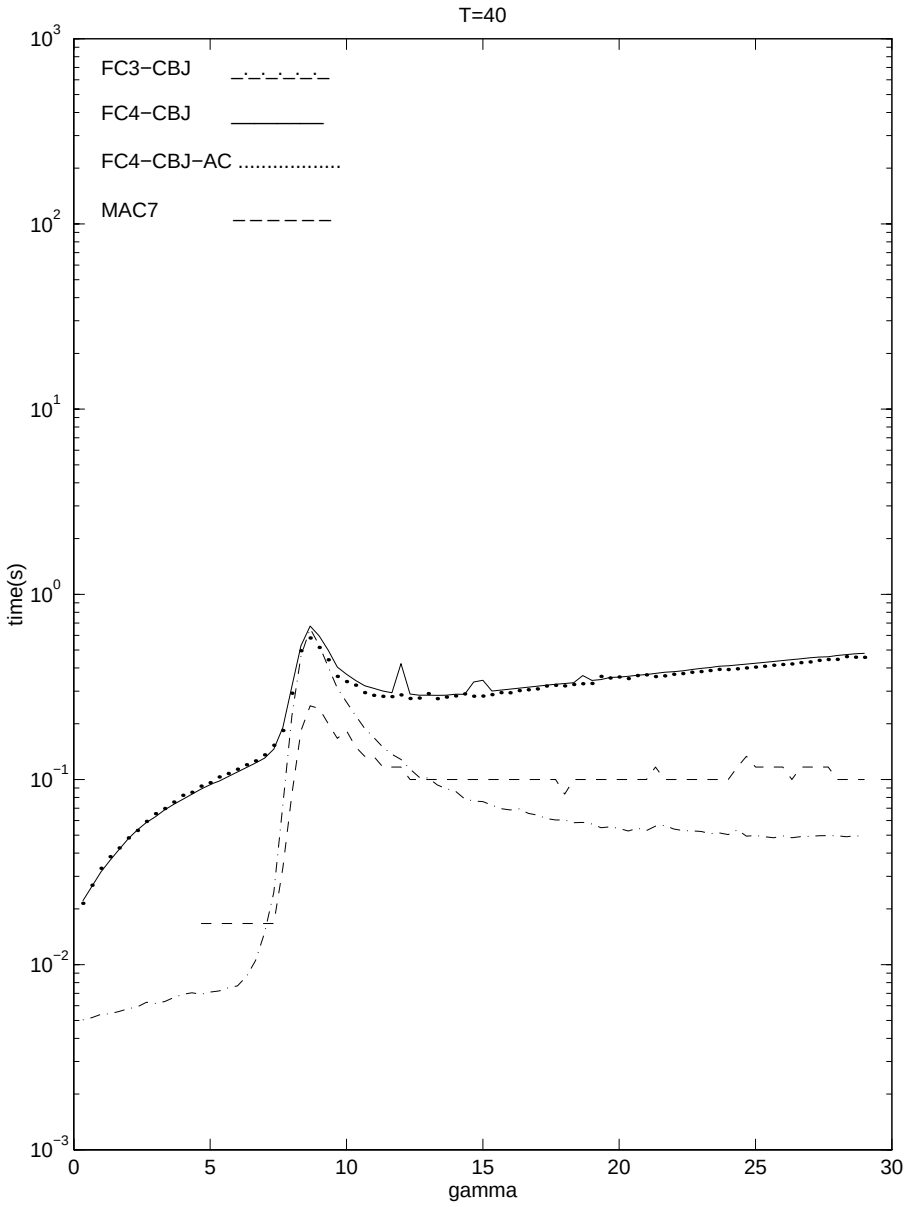

Figure 4.20: N=30, K=10

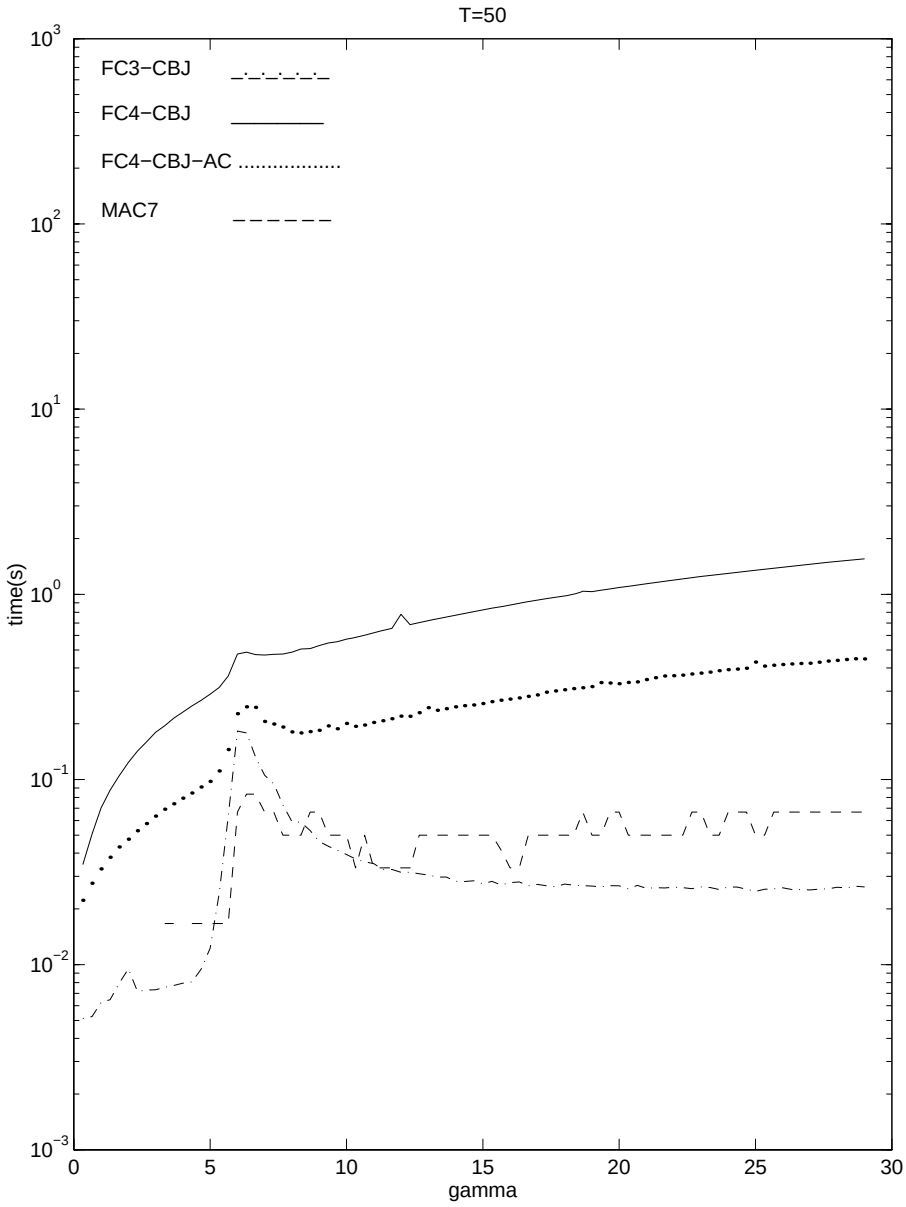

Figure 4.21: N=30, K=10

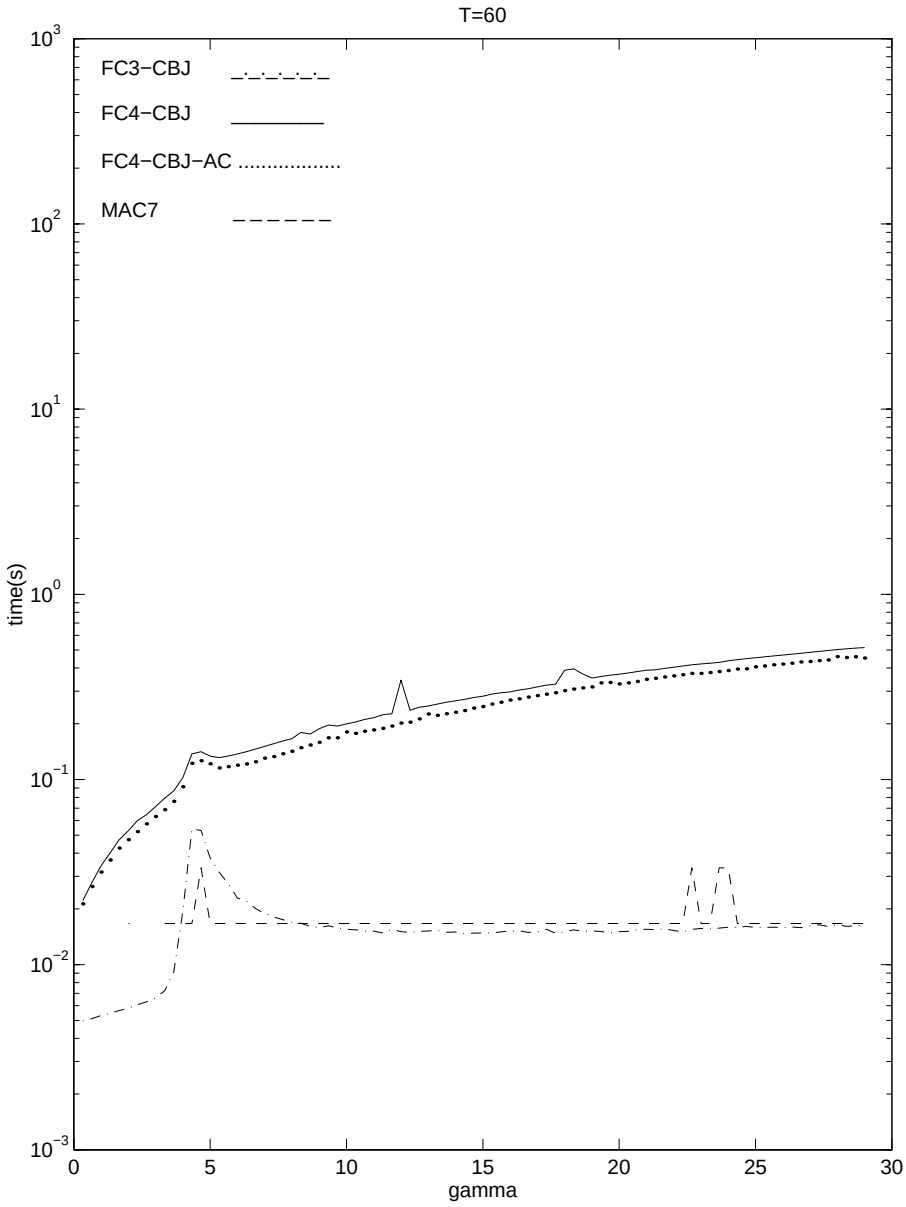

Figure 4.22: N=30, K=10

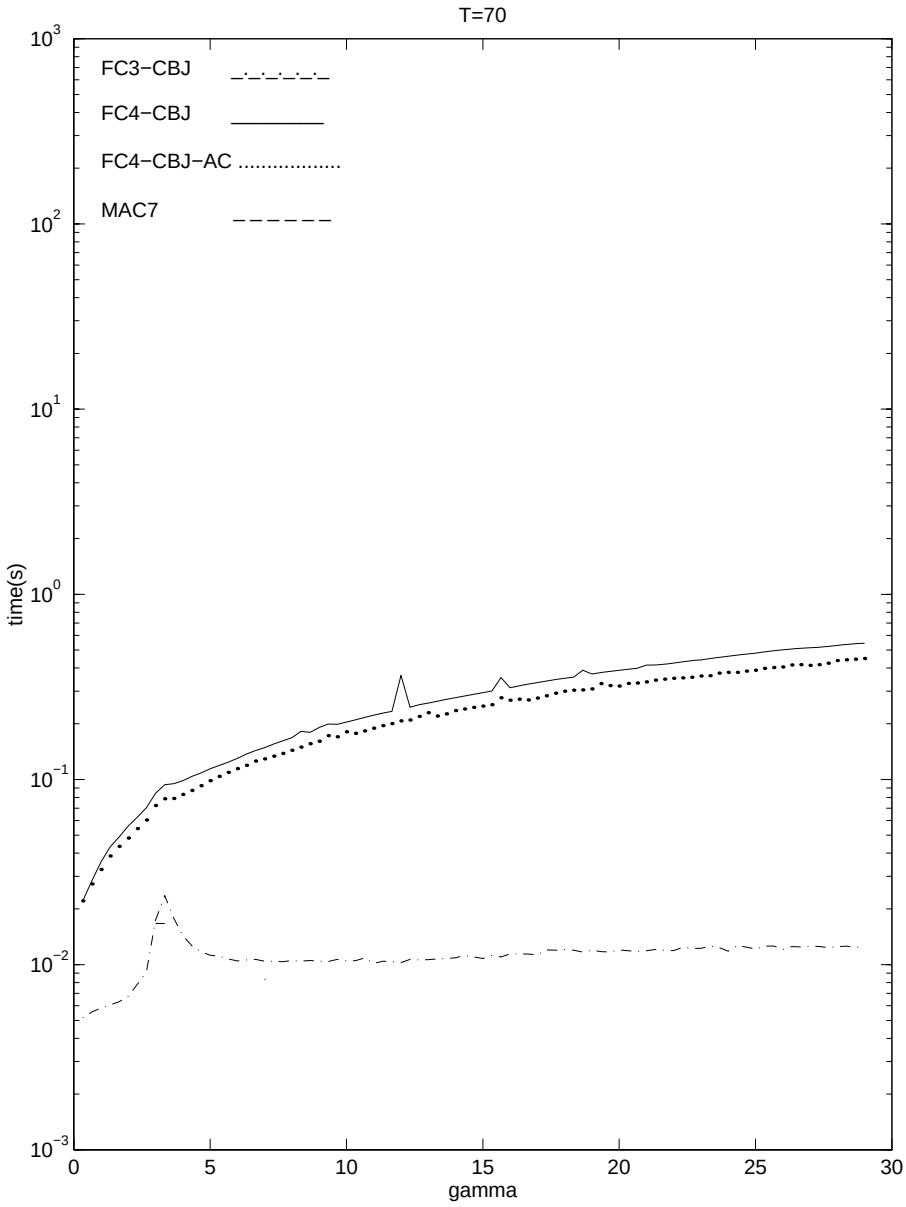

Figure 4.23: N=30, K=10

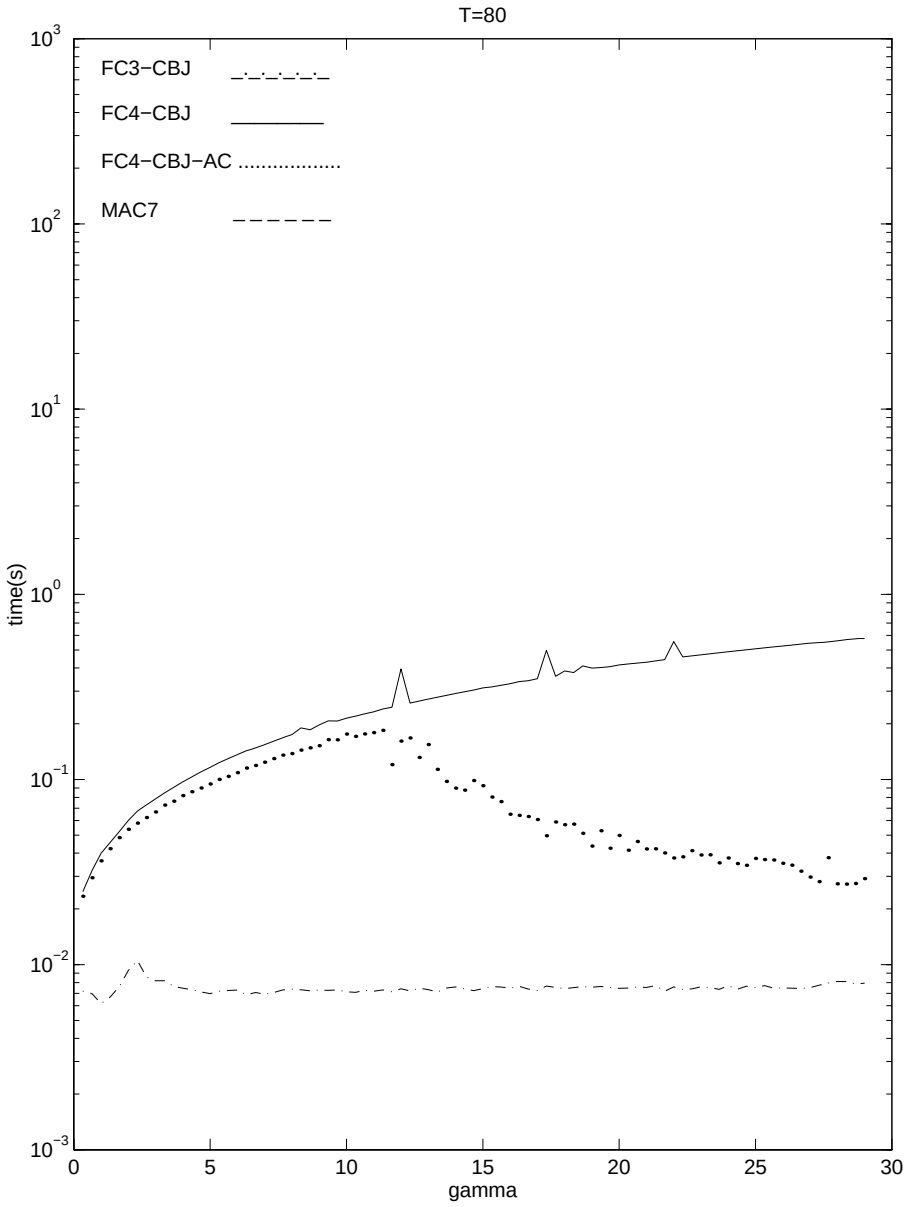

Figure 4.24: N=30, K=10

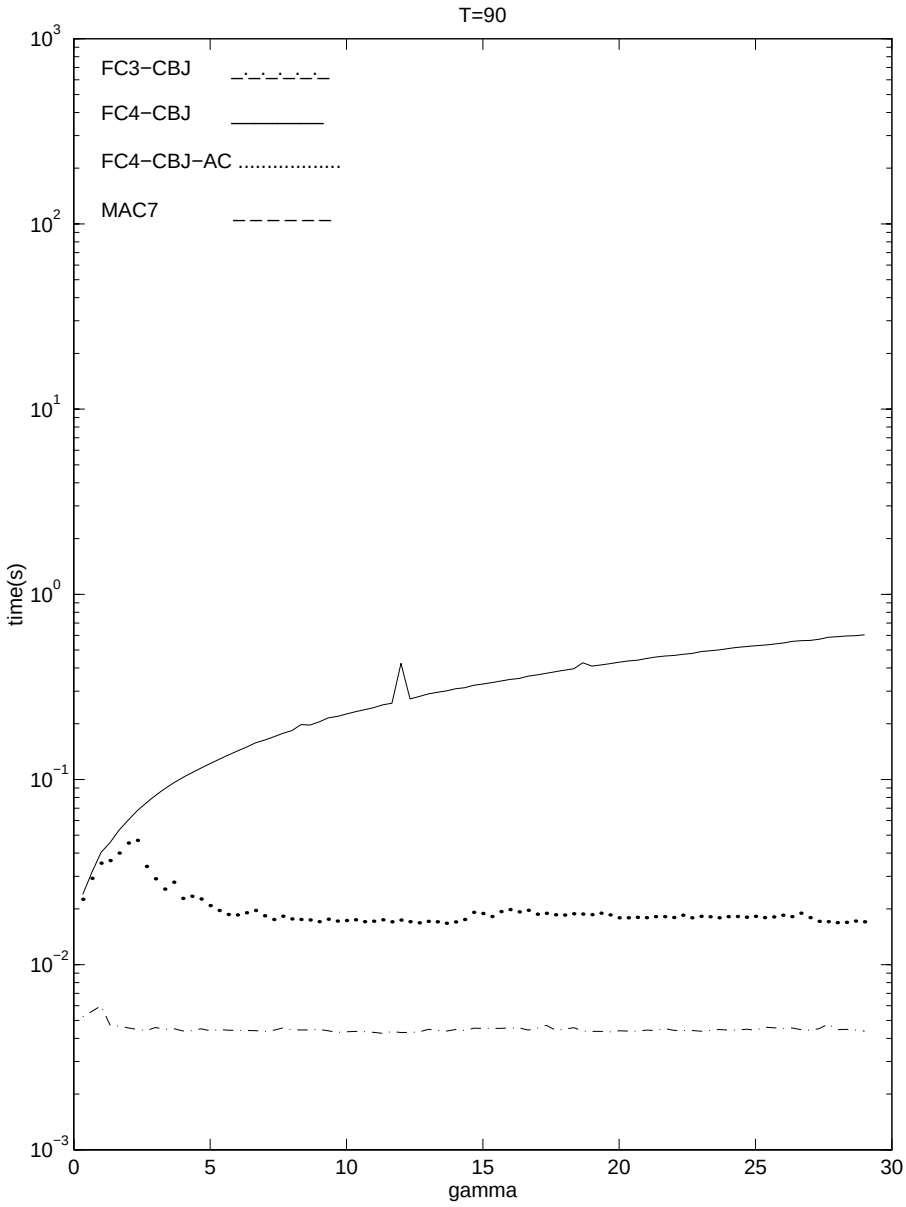

Figure 4.25: N=30, K=10

Figure 4.17–4.25 are the results of $N = 30$ and $K = 10$ problem classes. We have to use CPU time as the measure of performance here.

The qualitative behavior of these improved algorithms are similar to those unimproved algorithms for which we can use constraint checks as the measure of performance. This suggests that CPU time is a reasonably good measure, which is the prerequisite to make comparisons.

In some of the graphs, the plotting of MAC7 is not smoothly continuous and some of the data are even zero (in the graphs of $T = 70, 80, 90$, almost all the results of MAC7 are zero). This is due to the precision of the results of MAC7. That is why almost all of these cases occur when the problems are not hard (median performance is around or below 0.1s). However, its performance on hard problems which we are interested in will not be affected.

The MAC7 algorithm used here is very time efficient and it beats almost all the other FC algorithms at all the peaks except for some values of T that are of low constraint tightness. This coincides with our conclusion that MAC is usually not cost effective on problems with low constraint tightness.

For $T = 10$ (Figure 4.17), like FC3 outperforms MAC3, FC3-CBJ outperforms MAC7. The performances of FC4-CBJ and FC4-CBJ-AC are not good here because these problems are too easy and the overhead for initializing non-support sets becomes the main part of the total cost. For FC4-CBJ-AC, more extra overhead is required to achieve arc consistency before search starts though most of the problems are initially arc consistent when T is of such a small value.

For $T = 20$ (Figure 4.18), the corresponding peak values of the four algorithms are given below:

Algorithm	CPU time (sec)
FC3-CBJ	14.8081
FC4-CBJ	9.5643
FC4-CBJ-AC	7.7405
MAC7	9.7250

The superiority of our new FC algorithm — FC4, is shown here. Both of the two FC4 algorithms (with or without AC) perform better than MAC7 at the peak, especially FC4-AC. All these three algorithms beat the original FC3 algorithm.

For $T = 30$ (Figure 4.19), the corresponding peak values of the four algorithms are given below:

Algorithm	CPU time (sec)
FC3-CBJ	2.6101
FC4-CBJ	2.1362
FC4-CBJ-AC	1.8582
MAC7	1.2917

MAC7 is the best algorithm for hard problem classes of $T = 30$. Our improved algorithms of FC — FC4 (with or without AC) both beat the original FC3 algorithm.

In fact, for $T = 30$ to 90 (Figure 4.19-4.25), MAC7 outperforms all the FC algorithms at the peaks. As the performance of the FC4 algorithms (with or without AC), when $T = 30$, they still have some advantage on hard problem classes compared to FC3. But as T gets larger, their performance is getting worse compared to FC3. This coincides with our expectation that FC4 works well on problems with low constraint tightness. On the other hand, as T gets larger for the problems of fixed size ($N = 30$ and $K = 10$), the problems are getting easier and the overhead part of the cost of FC4 becomes dominant. Furthermore, this overhead is getting larger as T increases. The effect of arc consistency preprocessing is not apparent expect in the two graphs of $T = 80, 90$ where FC4-AC performs better on arc inconsistent problems compared with FC4.

We also did some experiments to show what happens as N increases. For $T = 10, 20, 30$, we increased N to 40. For $T = 40, 50, 60$, we increased N to 60. Figure 4.26-4.31 are the results.

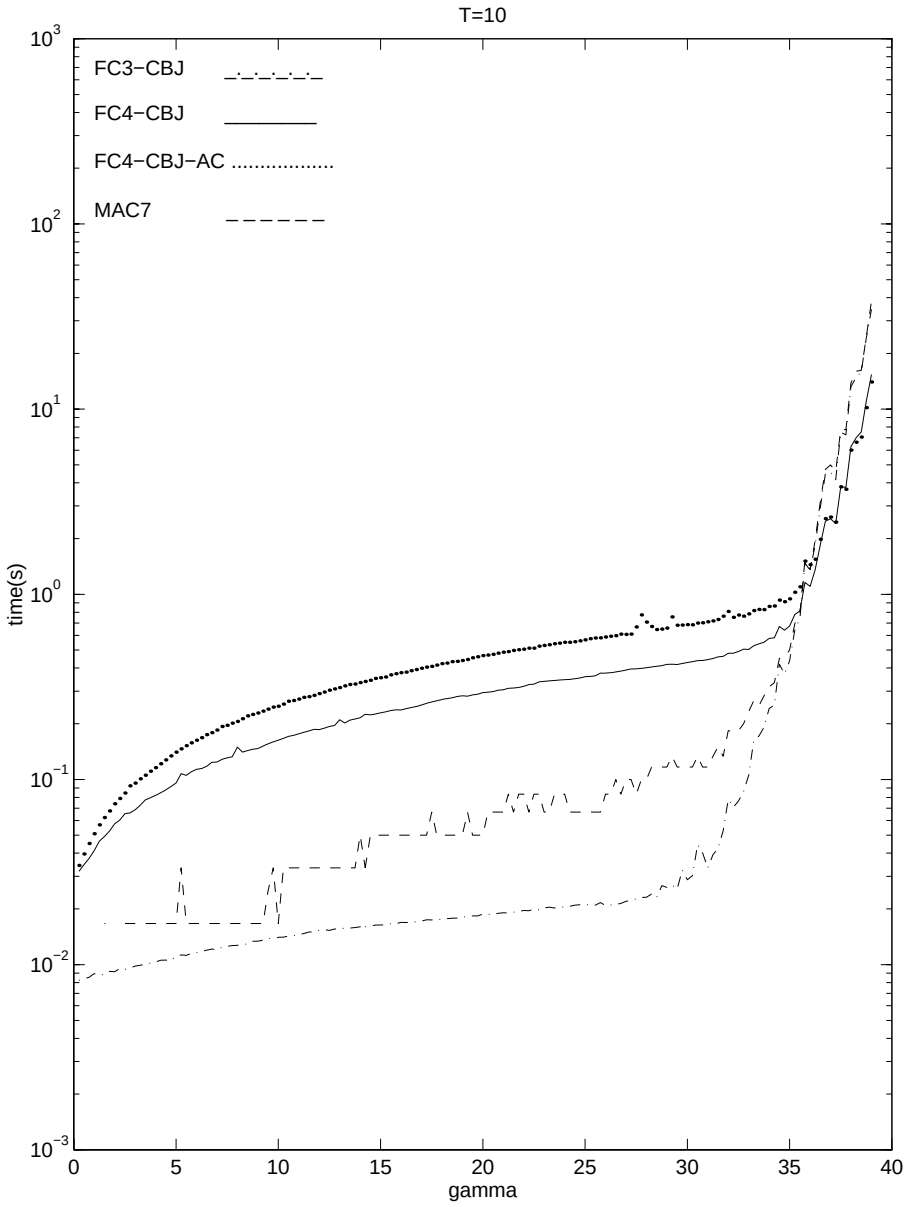

Figure 4.26: N=40, K=10

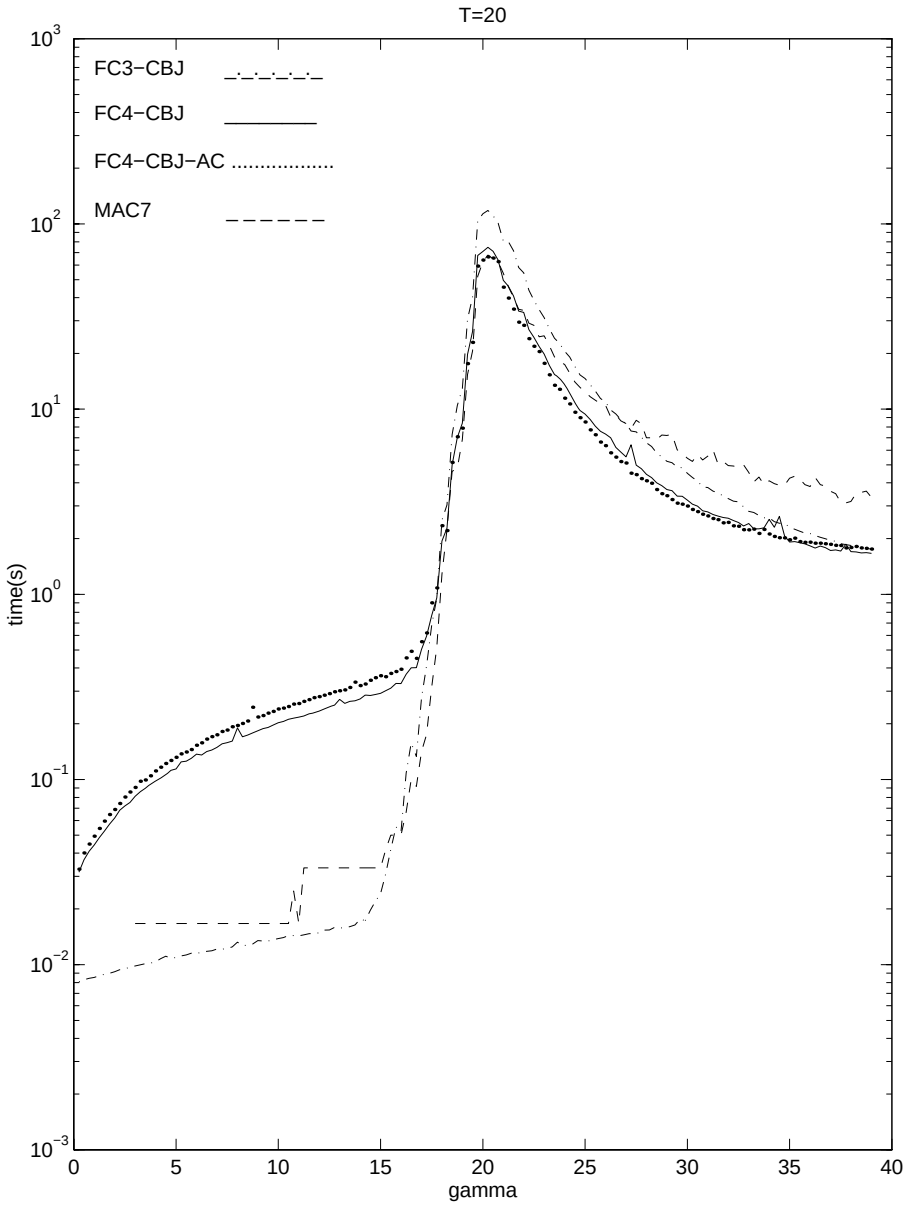

Figure 4.27: N=40, K=10

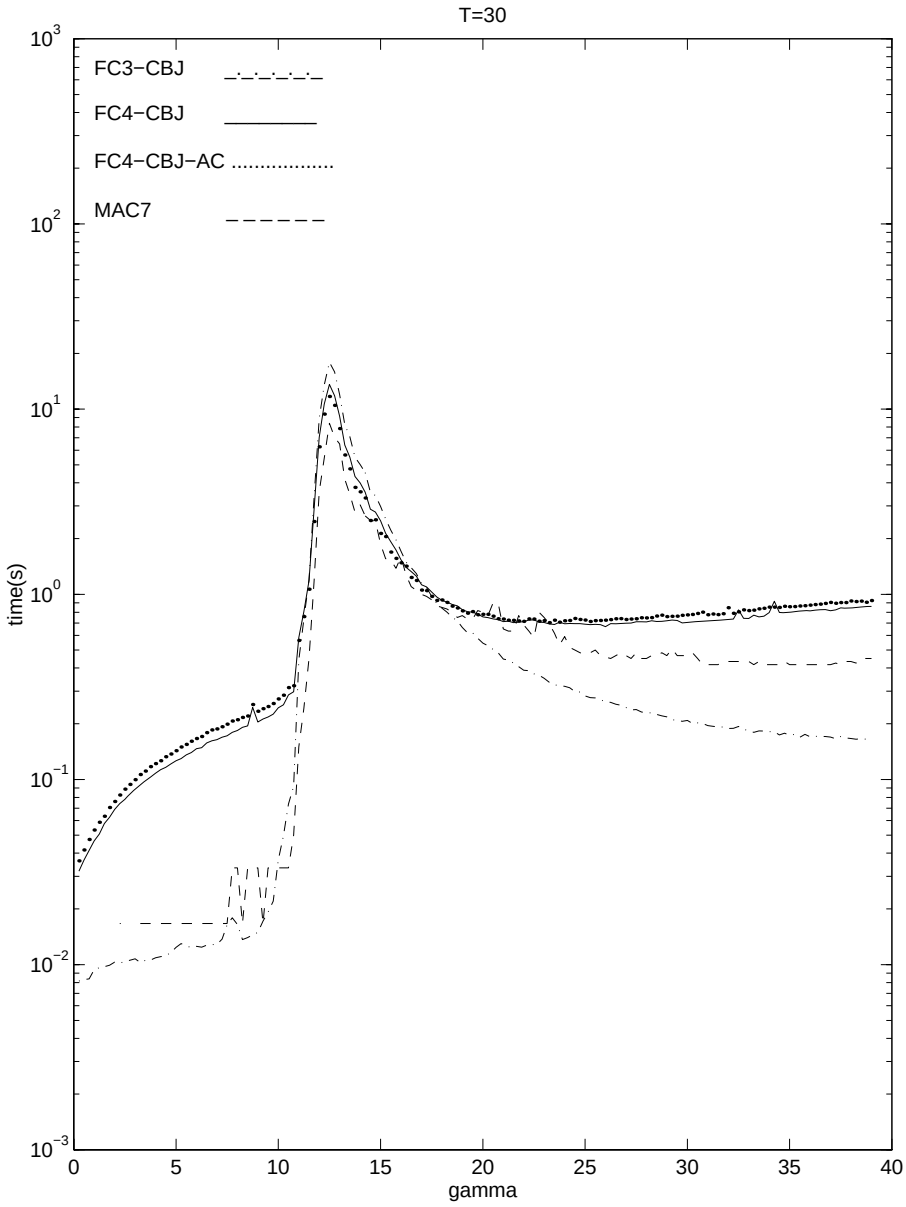

Figure 4.28: N=40, K=10

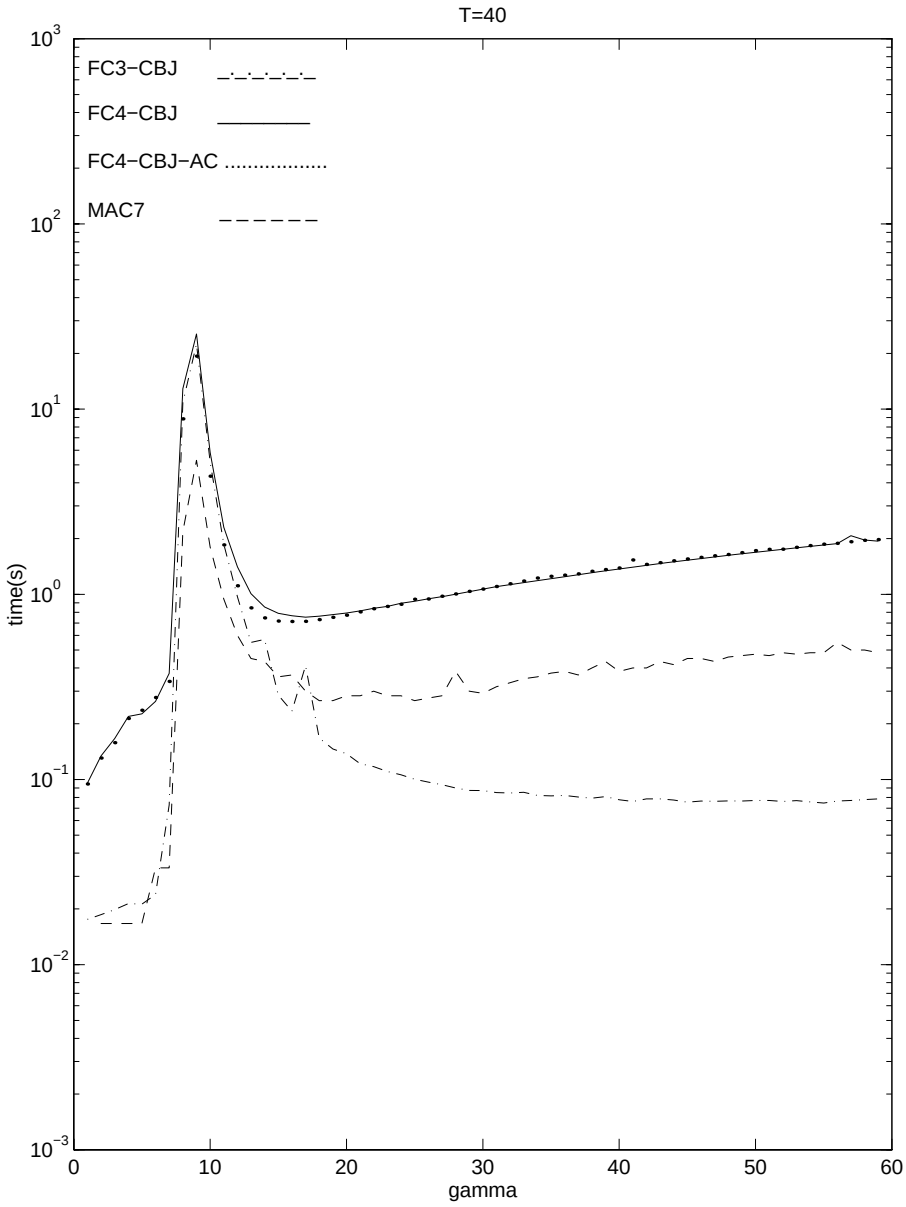

Figure 4.29: N=60, K=10

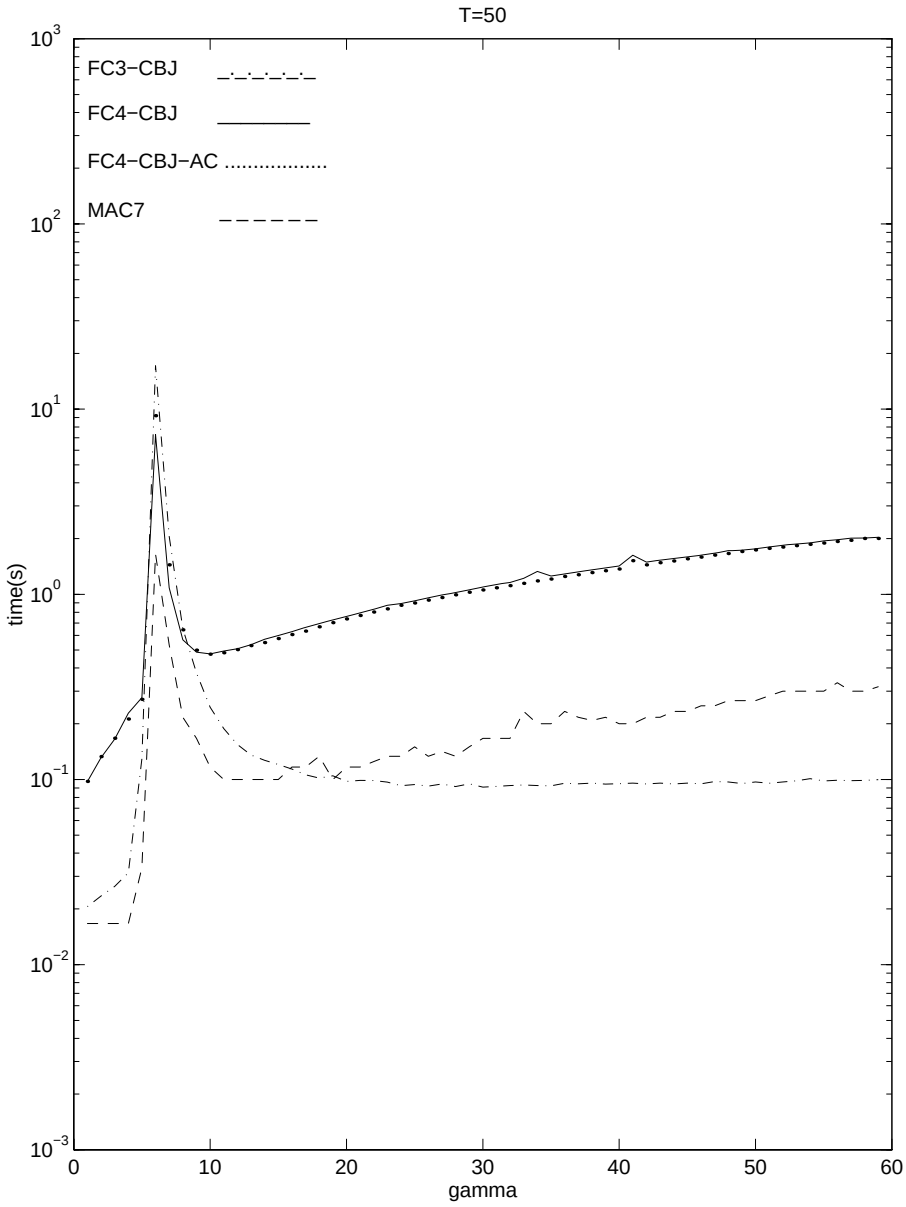

Figure 4.30: N=60, K=10

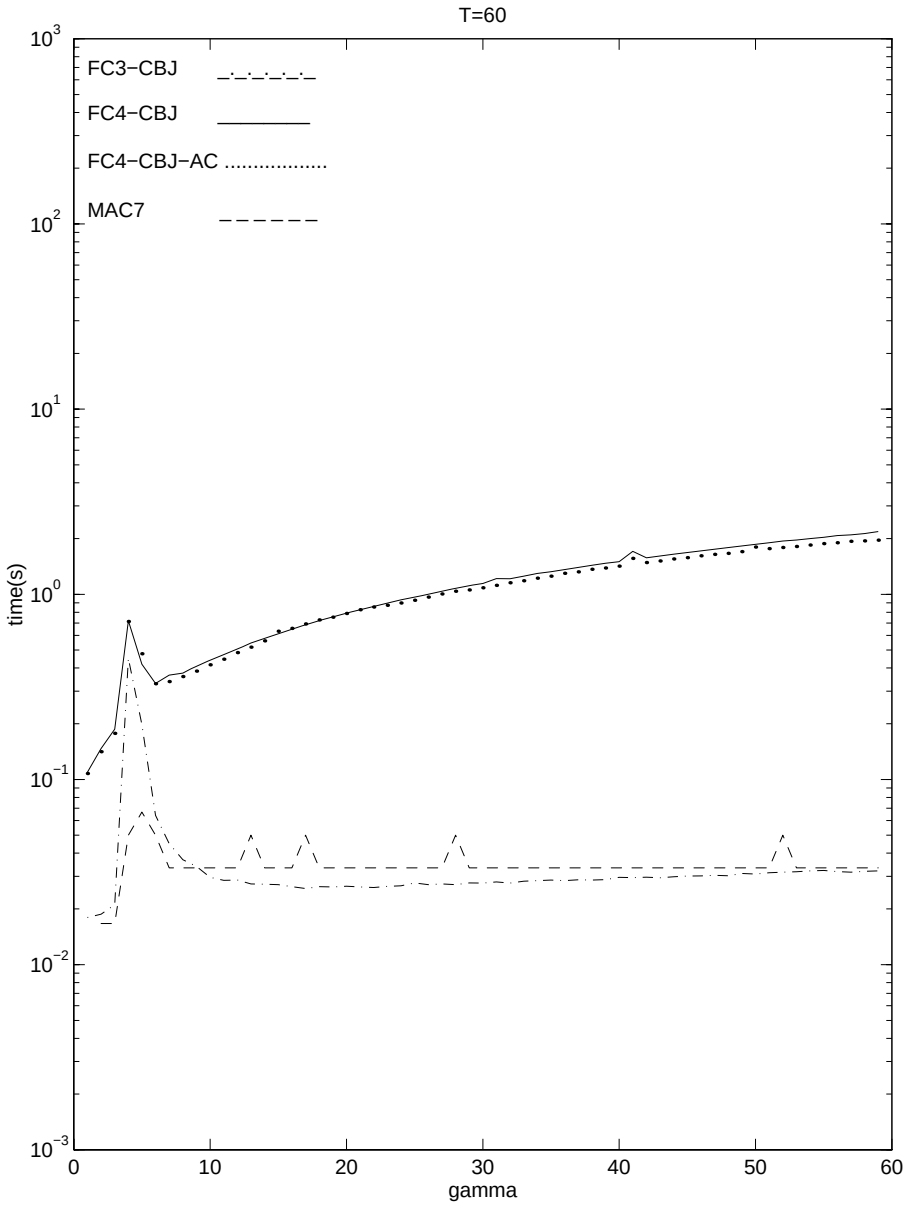

Figure 4.31: N=60, K=10

We look at the graphs for $T = 40, 50, 60$ first. We can see that the superiority of MAC7 on hard problems is enlarged as N is increased to 60. Since the problems become larger and thus harder, the FC4 algorithms have a better performance compared to that when $N = 30$. This suggests that the overhead of initialization do make a large part of the cost when we use FC4 to solve easy problems. But as the problems get harder, FC4 will show its superiority.

For the graph of $T = 20$ (Figure 4.27), the corresponding peak values of the four algorithms are given below:

Algorithm	CPU time (sec)
FC3-CBJ	118.3035
FC4-CBJ	74.8433
FC4-CBJ-AC	66.3708
MAC7	67.9500

The performance of MAC7 catches up with the performance of FC4 on hard problem classes as N is increased to 40, which shows a trend that MAC7 will eventually outperforms all the FC algorithms on hard problems of $T = 20$.

For the graph of $T = 30$ (Figure 4.28), there is no much difference compared to the graph of $T = 30$ as $N = 30$ except that the problems are harder.

The most interesting graph is for $T = 10$ (Figure 4.26). The corresponding peak values of the four algorithms are given below:

Algorithm	CPU time (sec)
FC3-CBJ	34.6245
FC4-CBJ	15.3864
FC4-CBJ-AC	14.0235
MAC7	38.3750

We are unable to obtain the result that MAC7 outperforms FC4. In fact, the performance of MAC7 here is as bad as the performance of the original FC3 algorithm. However, the performances of FC4, both with AC and without AC

are much better than MAC7 and FC3. This means that our new algorithm FC4 performs well on problems with low constraint tightness.

From this graph, we can also see that the hardest problem class appears as the value of C or γ is the largest, where the constraint graphs of these problems are complete graphs. In other words, when T is of such a small value as less than or around 10, the effect of increasing N is dependent on C or γ. We can also notice that the problems are getting hard so quickly as C grows to a certain point.

To explore further on the performance of these algorithms on problems with low constraint tightness, we did another set of experiments with $T = 5$. The results are given in Figure 4.32-4.35.

We have similar results as $T = 10$, except that we need to increase N to larger number to reach the point that MAC7 catches up with FC3. But we still were not able to obtain the result that MAC7 outperforms FC4. N is increased as large as 75, and the corresponding peak values of the four algorithms are given below:

Algorithm	CPU time (sec)
FC3-CBJ	511.4895
FC4-CBJ	177.3990
FC4-CBJ-AC	163.1060
MAC7	523.1250

From this table, we can see that the superiority of FC4 is more clear compared to the results of $T = 10$.

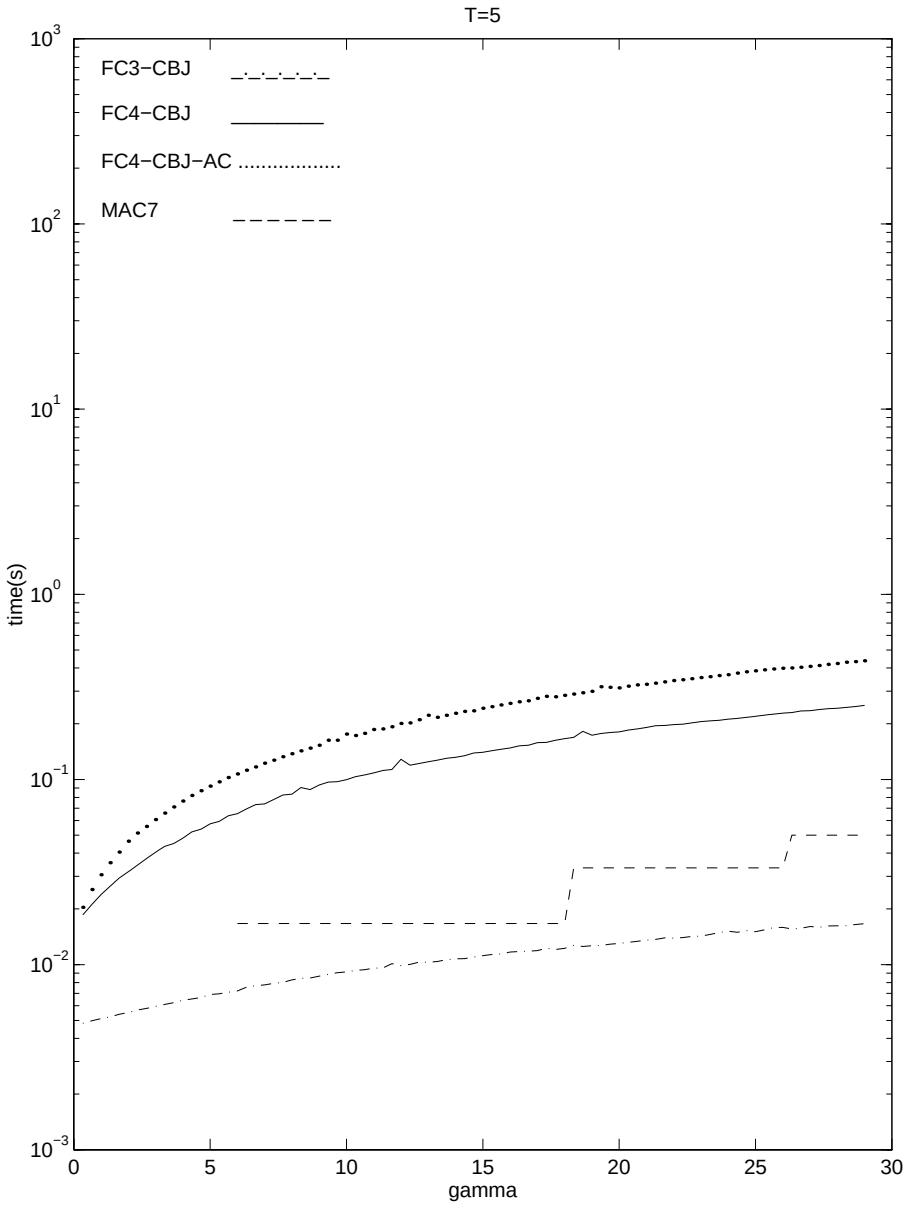

Figure 4.32: N=30, K=10

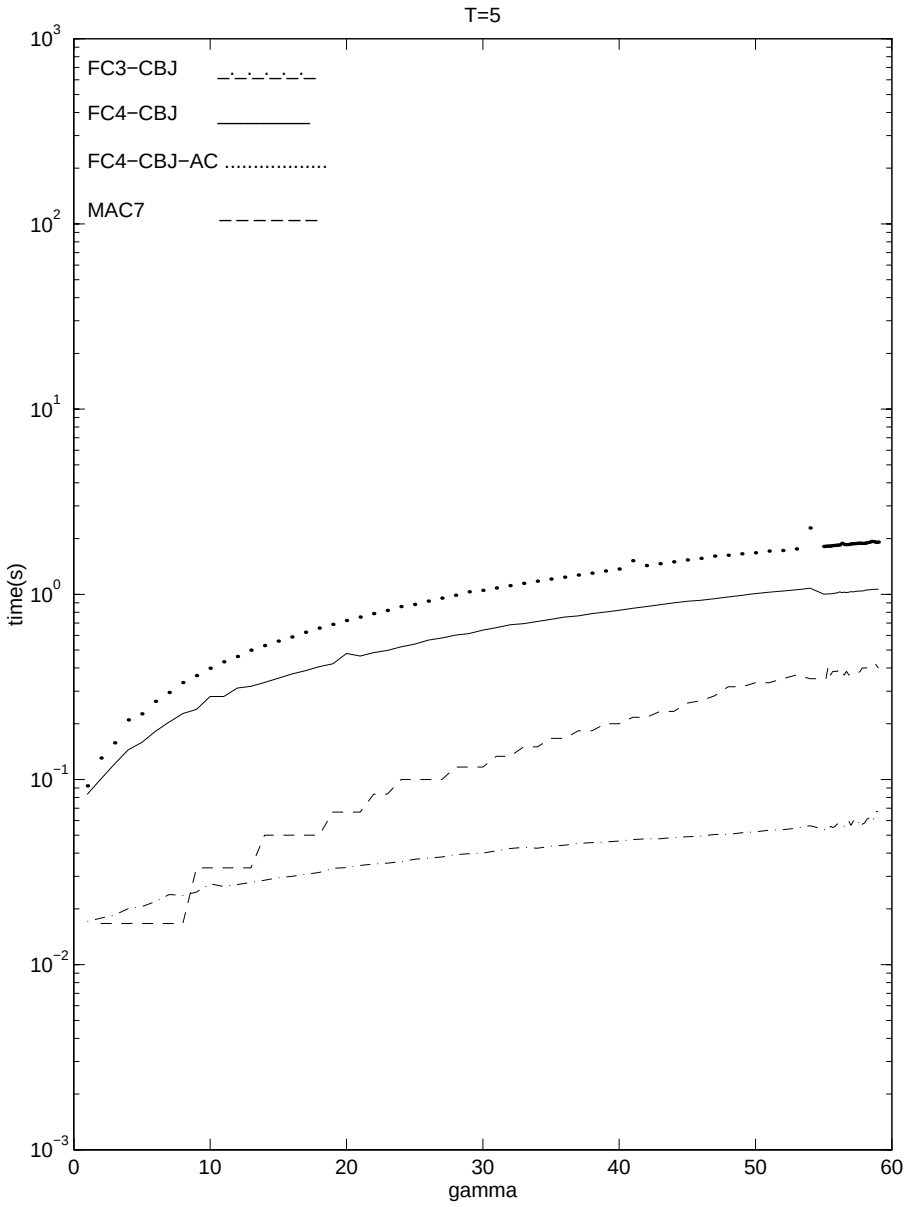

Figure 4.33: N=60, K=10

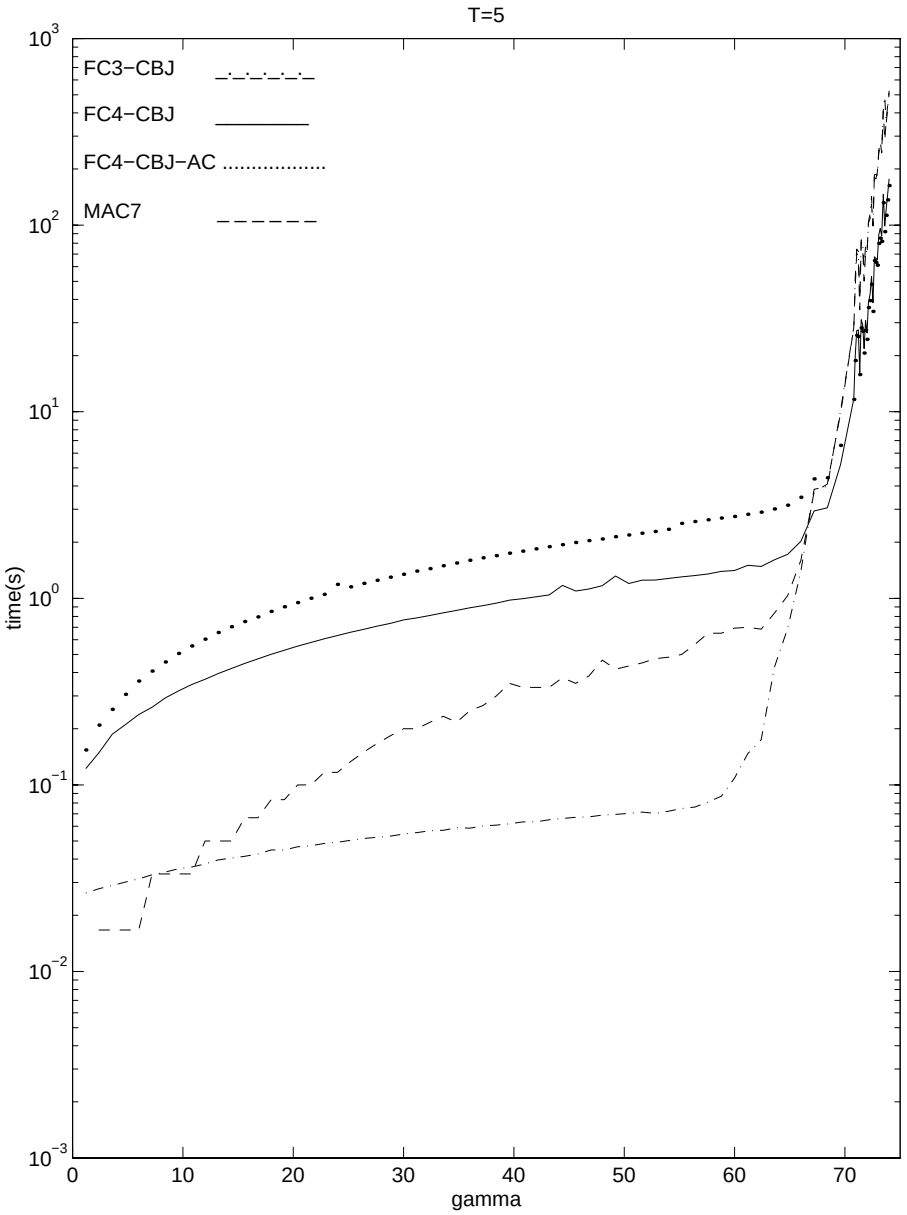

Figure 4.34: N=75, K=10

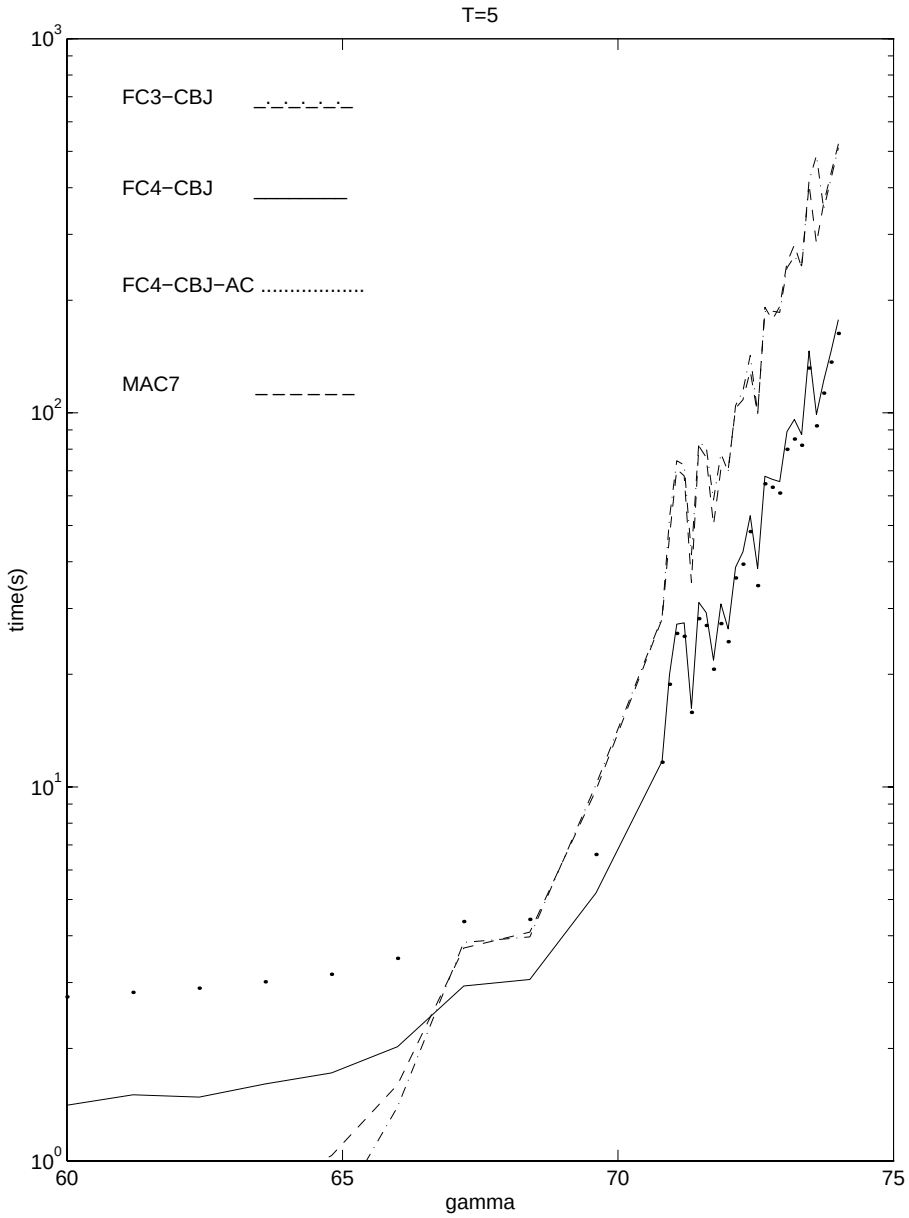

Figure 4.35: N=75, K=10 (enlarged)

Chapter 5

Conclusions and future work

In this thesis, we provide a systematic comparison of the performances of the MAC and FC algorithms on large and hard CSPs. In particular, we compare their performance with regard to the size, constraint density and constraint tightness of the problems.

From the empirical results, we have following conclusions:

- Given a problem size, hard problem classes tend to be those with either low constraint tightness and high constraint density or high constraint tightness and low constraint density. The hardest problem class for both FC and MAC is the one with relatively low constraint tightness and high constraint density.
- Though there is a trend that MAC eventually outperforms FC as we increase the problem size, for hard problem classes of a certain size, MAC performs better on those with high constraint tightness and low constraint density. In other words, the superiority of MAC over FC will not be revealed on the hard problem classes with low constraint tightness and high constraint density (which are harder than the problem classes with high constraint tightness given the same problem size) until the size of these problems is quite large. It could well be that for real problems FC remains the superior algorithm.
- The performance of MAC also depends greatly on its implementation. Improved algorithms of MAC work much better than the naive implementation of MAC that uses AC3 as the algorithm for arc consistency.

- Another contribution of this thesis is that we have devised a new FC algorithm — FC4. It shows good performance on the hard problem classes with low constraint tightness and high constraint density (which are the problem classes that MAC performs relatively poorly), if these problem classes are hard enough to overcome the extra overhead of FC4. It works especially well on problems with low constraint tightness less than or close to 10%, for which we were unable to increase the problem size to the point that MAC7 can beat it.

Future work

It will be interesting to explore further the performance of MAC7 and our new algorithm FC4 on the problems with low constraint tightness and high constraint density with increasing problem size.

All the experiments in this thesis were done on randomly generated problems with the same domain size for all variables and the same constraint tightness for all constraints. Real problems are unlikely to be exactly like this. With the conclusions we have obtained here, we can make some conjectures and try to verify these conjectures on some real problems.

We also only changed N, the number of variables, when we needed to change problem size. The effect of increasing domain size also needs to be studied.

Bibliography

[1] Christian Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[2] Christian Bessiere, Eugene C. Freuder, and Jean-Charles Regin. Using Inference to Reduce Arc Consistency Computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 592–598, Montreal, Canada, 1995.

[3] Christian Bessiere and Jean-Charles Regin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge, Massachusetts, 1996.

[4] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[5] Stuart A. Grant and Barbara M. Smith. The Phase Transition Behaviour of Maintaining Arc Consistency. In *Proceedings of the Twelfth European Conference on Artificial Intelligence*, pages 175 – 179, Budapest, Hungary, 1996.

[6] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[7] B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5:188–224, 1989.

[8] C. S. Peirce. In C. Hartshorne and P. Weiss, editors, *Collected Papers, Vol. III.* Harvard University Press, 1933.

[9] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 10–20, Rosario, Orcas Island, Washington, 1994.

[10] Daniel Sabin and Eugene C. Freuder. Understanding and Improving the MAC Algorithm. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 167–181, Linz, Austria, 1997.

[11] B. Selman, H. A. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 337–343, Seattle, Washington, 1994.

[12] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[13] Paul van Run. Domain Independent Heuristics in Hybrid Algorithms for CSP's. Master's thesis, University of Waterloo, 1994.