QDOOCS, **A C++ Class Library for Solving Binary Constraint Satisfaction Problems**

Kay Wiese, Shiv Nagarajan
and Scott D. Goodwin

Technical Report CS-96-08
September 1996

Department of Computer Science
University of Regina
Regina, Saskatchewan
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0332-5

Contents

List of Tables

List of Figures

1 Introduction

Many problems in Artificial Intelligence (AI) involve assigning values to domain variables so that the constraints between those variables are not violated. Such problems can be expressed in a general form as a *Constraint Satisfaction Problem* (CSP). CSPs are among the most investigated and well understood concepts in AI. Techniques for solving CSPs can be developed independently from the given “real world problem”. There are also specialized techniques that use problem specific characteristics to find solutions to CSPs. Research is ongoing in the field of constraint representation. This research deals with questions like: “How can constraints be represented so that it is space efficient?” “How can we represent constraints, so that we can access them quickly?” “What is a good representation for constraints if our solving algorithm involves merging constraints?”

There are different approaches to implementing software for CSP solving. On the one hand there are solvers implemented in languages such as Prolog or *constraint programming languages* (CPLs). The latter provide built-in functions or predicates for describing commonly encountered constraints and solve CSPs by applying techniques which have been developed in CSP research. The advantage of CPLs is the ease of implementation, because the user does not have to worry about how the actual solving algorithms work. The disadvantage is slow execution and inflexibility. Other languages that have been used for implementing CSP solvers are C and C++ (see [vBe92] and [jac92]). These languages allow greater flexibility because the user can develop different search strategies for solving CSPs and test their performance on different problems. Much interest has been focused on C++ lately because it is used widely in academic research as well as in the industry and combines concepts of efficient procedural C with object oriented design (OOD). It also allows the writing of safe and reusable code (see section 2). Here we report on our design and implementation of a C++ library for solving binary CSPs. A basis for our implementation was Peter van Beek‘s C constraint library ([vBe92]).
Section 2 introduces the basic concepts of object oriented design (OOD) and

object oriented programming (OOP). In section 3 we give a brief overview of the terminology in Constraint Solving. Our implementation including the classes we defined is described in section 4, 5, and 6. Some practical results are given in section 7. After our conclusion in section 8 we discuss some further research in section 9.

2 Concepts of Object Oriented Design

There is no unique definition of what OOD or OOP is. The literature agrees that it consists of *abstract data types (ADTs)*, *overloading of function names and operators*, *data hiding*, *inheritance and code reuse*. In C++ an ADT is usually implemented as a class. A class is an extension of a C struct. As with a struct, a class has a data part. In addition a class has so-called *member functions* that act on the data part. The data part is usually *private* and therefore hidden from reference or access from outside the class. A client can only access or manipulate the internal data of a class through member functions. This makes the member functions public. Client functions outside the class scope can use them. In comparison to procedural programming an object oriented program consists of a collection of objects (in C++ an object is an instantiation of a class) and methods that describe how these objects interact. In C++ these methods are member functions. Inheritance allows code reuse because some objects might share some common characteristics. This is implemented in C++ as *base classes* and *derived classes*. For example if *student* is a base class then *graduate_student* could be a derived class using some of the code for *student* and having its own code for special characteristics of graduate students such as thesis topic. Overloading allows functions of the same name in different classes (these can also be in the same hierarchy) or with different signatures. There is a great variety of books on OOD and OOP. For a quick introduction into OOP using C++ see [Poh94].

3 Concepts of Constraint Solving

In this section we will only give a brief introduction and overview of some concepts, algorithms and the methodology of constraint solving. For a comprehensive overview we refer the reader to [Tsa93]. A CSP (in its finite domain formulation) is a problem composed of a finite set of *variables*, each of which has a finite *domain*, and a set of *constraints* that restrict the values that the variables can simultaneously take. Examples of CSPs are the N-Queens problem, the graph or map coloring problem, resource allocation and scheduling. Formalizing a given problem as a CSP involves representing the constraints. This can be done by 0-1 entries in a matrix, where the rows are the domain values of one variable and the columns are the domain values of another variable. To check if there is a constraint between variable i and j if i takes a value a and j takes a value b we simply have to check the entry [i][j][a][b] in the *constraint network*. A constraint network is a network whose nodes are the variables of the CSP with their domains and if a variable i constrains a variable j then there is an edge between i and j annotated with the constraint. Note that the above remarks are only true for binary constraints. For general constraints the constraint network is usually a hypergraph and the methods of representing constraints are more sophisticated.

To solve a given CSP there are three different approaches:

1. Problem Reduction. This involves preprocessing the given constraint network to achieve some consistency property like node-, arc-, path- or k-consistency. The reduced problem is hopefully, but not necessarily, easier to solve. Problem reduction removes redundant values from domains and redundant compound labels from constraints. It can also prune the search space during searches for solutions as well as help avoiding repeated futile search of subtrees. If a problem has no solution because it is over-constrained, then problem reduction can detect this and no effort has to be spent on finding solutions.

2. Search. Search-based techniques have been fairly successful in constraint solving on sequential computers. According to [Tsa93] these techniques fall into three categories: (1) general search strategies, such

as chronological backtracking BT [Tsa93] and iterative broadening [Gin90], (2) lookahead strategies, such as forward checking [Har80], and (3) gather-information-while-searching strategies, such as back-jumping [Gas79, Pro92], dynamic backtracking [Gin93], back-checking, and back-marking [Har80]. The given list of example search techniques is not exhaustive, there are other specialized algorithms that fall into one of the three categories, as well as there are hybrid algorithms that combine some features of algorithms from different categories.

3. Solution Synthesis. This technique constructively generates legal compound labels rather than eliminating redundant values or compound labels. When synthesizing the n-constraint for a problem from the (n-1)-constraints, then all the n-compound labels which violate some constraints are removed.

4 NETWORK and CONSTRAINT: Two Classes Used for Constraint Solving

As we have already mentioned in section 2 binary constraints can be represented by a four dimensional array structure. In [vBe92] this was done by a type definition for constraints and network.

```
typedef char CONSTRAINT[K][K];
typedef CONSTRAINT NETWORK[N][N];
```

The basis for our object oriented design was to implement CONSTRAINT and NETWORK as classes. The functions for generating CSPs such as *generate_Q*() to generate an N-Queens problem as well as the preprocessing routines and the search routines were implemented as member functions. In addition to rewriting code from C to C++ we also had to write constructors, destructors, public access to the NETWORK and the CONSTRAINT classes (overloaded) as well as overloaded print functions. However, we did not change the internal structure of the algorithms so that we can make a fair comparison between the procedural and the object oriented implementation.

CONSTRAINT is a pointer to pointer structure that allocates a $k \times k$ array if the constructor is called with a parameter k. Otherwise the default constructor is invoked and allocates a $VARS \times VARS$ array, where VARS is a global constant defined in the file *global.h*. Let us look at the declarations of the class CONSTRAINT and the definitions of the constructors and the destructor:

```
// Implementation of binary constraints as a class
// CLASS CONSTRAINT


class CONSTRAINT {
public:
  // default constructor
  CONSTRAINT();
  // constructor defining the size of the constraint
  CONSTRAINT(int size);
  // destructor
  ~CONSTRAINT();
  // return the size of the constraint (i.e. k)
  int size();
  // access the entry in the constraint matrix for value a and b
  char access(int a, int b);
  // print the whole constraint
  void print();
  // assign a value 0 or 1 to C[a][b]
  void assign(int a, int b, char value);


private:
  char **C;    // CONSTRAINT c as a 2D array structure
  int k;      // size of the CONSTRAINT
};
```

```
// default constructor
CONSTRAINT::CONSTRAINT(): k(DOMS)
{
  C = new (char*)[k];
  assert (C != 0);
  for(int i=0; i<k; i++)
    {
      C[i] = new char[k];
      assert(C[i] != 0);
    }
}

// constructor defining the size of the constraint
CONSTRAINT::CONSTRAINT(int size): k(size)
{
  if (size < 1)
    {
      cerr << "Illegal CONSTRAINT size" << size << endl;
      exit(1);
    }

  C = new (char*)[k];
  assert (C != 0);
  for(int i=0; i<k; i++)
    {
      C[i] = new char[k];
      assert(C[i] != 0);
    }
}
```

```
// destructor
CONSTRAINT::~CONSTRAINT()
{
  for (int i=0; i<k; i++)
    delete [](C[i]);
  delete []C;
}
```

Another concern was to hide the internal representation of the data, so that it could not be manipulated in an uncontrolled manner. That is why C and k are declared private. Note that the constructors are safe in a way that they check whether or not enough memory is available by calling the $assert()$ function. This is a C++ feature. *New* and *delete* are new C++ operators comparable to *alloc*, *malloc* and *free* in traditional C. We also wanted to make sure that no client (user) of our classes could access a memory location in the class NETWORK or CONSTRAINT that was not allocated and would therefore lead to a segmentation violation. The original C version did not take care of out of range memory access. This of course introduces additional overhead which will lead to an increase in execution time. We will look at the NETWORK class to illustrate safe array access:

```
// Access the constraint on variables i and j for values a and b
char NETWORK::access(char i, char j, char a, char b)
{
  if ( i < 1 || i > n || j < 1 || j > n )
    {
      cerr << " illegal index for network when calling NETWORK::access"
        << endl;
      exit(1);
    }
  return(N[i][j].access(a, b));
}
```

If M is an object of type NETWORK then we could access the constraint on variable i and j for values a and b by calling "M.access(i, j, a, b)". The member function NETWORK::access would now check if i and j are within the allocated index range. What about a and b? How are they checked? Note that NETWORK::access calls CONSTRAINT::access on the constraint N[i][j]. As CONSTRAINT::access checks whether a and b are out of range we can be sure that nothing within the NETWORK class can be accessed if it was not allocated. We reuse code here by simply calling a member function of CONSTRAINT instead of writing the checks for the boundaries on a and b explicitly. Something similar is done for the NETWORK::print() function which calls the CONSTRAINT::print() function.

The search algorithms are in the files "BT.cc", "BJ.cc", etc. They are implemented as member functions, because they are the functions that operate on the data part of NETWORK and are hence part of the ADT NETWORK. Because they are declared inside the class scope the scope resolution operator "::" must be used for their definition outside the class scope.

5 STACK and STACK_2: Two Classes Used for Preprocessing

In order to use preprocessing routines to achieve arc consistency and path consistency, two distinct versions of a stack are required. The first version, *class STACK*, is a simple stack. The *class STACK* has as its public member functions functions to pop an element, push an element, test for emptiness and initialise the stack.

```
class STACK
{
public:
  // default constructor
  STACK();
  // constructor defining the size of the stack
  STACK(int size);
```

```
  // destructor
  ~STACK();
  // Pushes a value onto the stack
  void push(int a);
  // Pops a value from the stack (reference argument)
  void pop(int& a);
  // returns true if the stack is empty
  int stack_empty();
  // initializes the stack to an empty stack
  void init_stack();
};
```

The stack is essentially used to push variables and hence need be at most the size of the number of variables (which is *VARS* in our case). This is therefore the default size for the stack in its constructor. But since this class is being provided external to the *NETWORK* and *CONSTRAINT* classes we also provide a constructor with the size specifiable.

```
// Constructor (default)
STACK::STACK(): n(VARS+1)
{
  stack = (int *)new int[n];
  assert(stack != 0);
  on_stack = (int *)new int[n];
  assert(on_stack != 0);
  top = 0;
}
```

When an element is pushed onto the stack, care is taken to see that the element is not already on the stack. This is essential because of the size limit of the stack which would permit only a limited number of values to be pushed onto it. This does in a way limit the stack implementation but as visible from the code it is easily changeable.

```
void STACK::push(int a)
{
  if (!on_stack[a])
     {
       top++;
       stack[top] = a;
       on_stack[a] = 1;
     }
  return;
}
```

In order to pop a value from the stack we used a reference argument for the function *pop*. The utility of reference arguments is easily visible here, with the value being returned. In traditional C, one would have to pass a reference pointer, which has to be dereferenced to be used in the function. This does in a way render the client code less readable. The reference argument notation in C++, is similar to the *var* notation in Pascal.

```
void STACK::pop(int& a)
{
  a = stack[top];
  on_stack[a] = 0;
  top--;
  return;
}
```

The second version of the stack is *class STACK2*. This stack is more specialised with the push and pop referring to specific edges in the constraint network. Since the number of edges in the constraint network is at most $\frac{VARS*(VARS-1)}{2}$, this is the value used by the default constructor to construct the stack. Again another constructor is provided to allow stacks of different sizes to be created.

```
// Constructor (default)
STACK2::STACK2(): n(VARS+1)
{
  int k;
  k = n*(n+1)/2;
  stack2 = (int **)new (int *)[k];
  assert(stack2 != 0);
  for (int i=0; i < k ; i++)
  {
    stack2[i] = (int *)new int[2];
    assert(stack2[i] != 0);
  }
  on_stack2 = (int **)new (int *)[n];
  assert(on_stack2 != 0);
  for (int i=0; i < n ; i++)
  {
    on_stack2[i] = (int *)new int[n];
    assert(on_stack2[i] != 0);
  }
  top = 0;
}
```

The stack has public member functions *pop*, *push*, *test for emptiness*, and also for initialisation. One also has a marker which ensures that an edge which is used onto the stack is not already on the stack.

```
class STACK2
{
public:
  // default constructor
  STACK2();
  // constructor defining the size of the stack
  STACK2(int size);
  // destructor
  STACK2();
```

```
  // Pushes an edge onto the stack
  void push(int a, int b);
  // Pops an edge from the stack (reference arguments)
  void pop(int& a, int& b);
  // returns true if the stack is empty
  int stack_empty();
  // initializes the stack to an empty state
  void init_stack();
};
```

Again the *pop* function takes reference arguments to ensure clean client code.

```
void STACK2::pop(int& a, int& b)
{
  a = stack2[top][0];
  b = stack2[top][1];
  on_stack2[a][b] = 0;
  on_stack2[b][a] = 0;
  top--;
  return;
}
```

Arc consistency routines use STACK while path consistency routines use STACK2. The path consistency routine ensures arc consistency since any k-path consistent network is also arc consistent for $k \geq 2$.

6 Other Functions and Classes

Besides our main classes that we described in the last two sections there are also a few other functions that were used. The file "solve.cc" contains a general function to solve a CSP. It is also a member function of NETWORK and takes as one argument the search method that one wants to use.

The solution that is found by one of the search algorithms is always verified within the search algorithm by a call of "process_solution(S)", which checks if a found solution S is consistent with the original constraint network. This function is contained in the file "process.cc". The function "show_solution(N,S)" from file "show.cc" verifies the solution S and then outputs it. N is the current network and has to be passed as a parameter since show_solution is not a member function of NETWORK. The search algorithms have a parameter that indicates if a single solution is desired or if all solutions need to be found. In the latter case we store the first solution that was found into a global variable SOL_1 using the function "sfs(S)" from file "sfs.cc". For memory considerations we do not store all solutions that are found.

We use two different timers. The functions in the files "limit.cc" and "timer.cc" implement a stopwatch. The user can specify a certain time limit for the stopwatch. One can either specify real time or user time. After the stopwatch expires the search algorithms terminate. The code for these function was taken from [vBe92]. We also implemented our own timer in "timer_2.cc". This timer allows the user to test how much time a function or program takes to execute. Again one has the choice of real and user time. This timer is implemented as a class TIME_KEEPER and has the public member functions "start_time(void)" to set a current TIME_KEEPER to 0. The member function "print_time(void)" outputs the real and user time that has passed after start_time() was called.

7 Practical Results

7.1 The N-Queens Problem

We tested our constraint solver with the N-Queens problem for 15, 20, and 25 queens. We were interested in the time it takes to find a single solution as well as in the number of solutions a search algorithm can find within a given time limit. Although we recorded both the real and the user time to find a single solution we will only report the user time here, since the real time is not representative. All programs were run on an SGI Indy which has a MIPS R4000 CPU and a MIPS R4010 floating point unit, and a 100

MHz clock. The main memory size is 64 Mbytes. For the C++ version we used the g++ compiler. For the C version we used the cc compiler. No optimization options were used. The times shown in table 1 are in seconds. The first number represents the execution time for the C++ version, the second number represents the execution time for the C version. Table 2 shows how many solutions the different search algorithms found given a specific time-limit. The first number represents the number of solutions found by the C++ version, the second number is the number of solutions found by the C version.

The graphs give a better picture of the execution times and the slowdowns of the C++ version compared to the C version. Note that the slow down factor is not a constant. However the graphs are similar in appearance.

Method	**15 Queens**	**20 Queens**	**25 Queens**
BT	0.4/0.1	98.2/19.4	35.0/6.6
BJ	0.2/0.1	44.8/17.8	15.2/5.5
BM	0.1/0.0	13.2/5.6	3.8/1.6
FC	0.2/0.1	36.8/14.6	12.3/5.0
BM-BJ	0.1/0.0	12.0/5.1	3.4/1.4
FC-BJ	0.2/0.1	40.3/17.3	13.6/5.8
CBJ	0.3/0.2	61.4/19.0	16.9/6.2

Table 1: Single Solution

Method	**15 Q (15 Secs)**	**20 Q (300 Secs)**	**25 Q (300 Secs)**
BT	134/1184	40/936	19/169
BJ	389/1413	219/1150	58/200
BM	1294/3086	1697/5666	311/966
FC	488/1691	292/1697	103/283
BM-BJ	1478/3195	2226/5999	352/1120
FC-BJ	452/1288	263/1363	82/225
CBJ	161/1122	126/1006	57/178

Table 2: Multiple Solutions

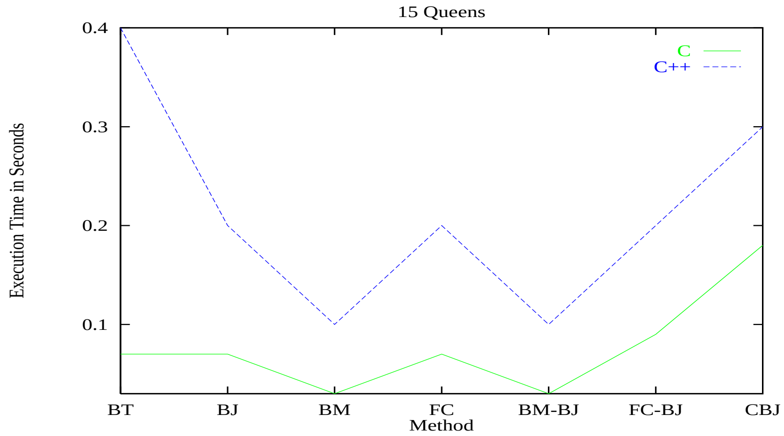

Figure 1: C++ versus C for the 15 Queens Problem

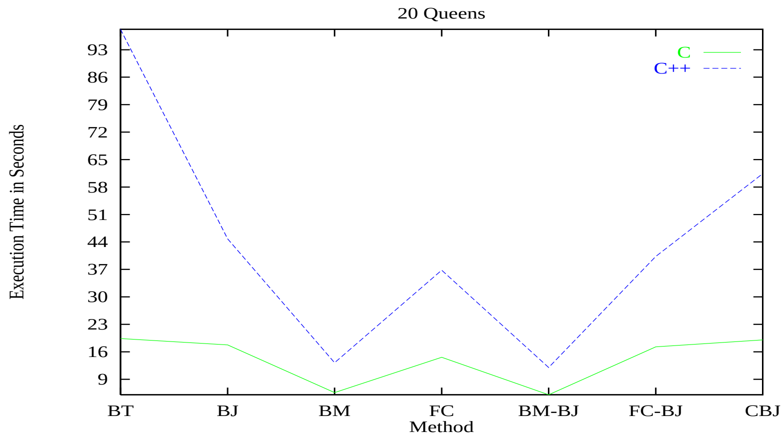

Figure 2: C++ versus C for the 20 Queens Problem

7.2 Other CSPs

The N-Queens problem is a very special CSP. It has certain characteristics that are not found in other CSPs. In fact there are deterministic algorithms for the N-Queens problem ([Sos91] and [Sos90]). In order to get more general results we tested our constraint solver with randomly generated CSPs

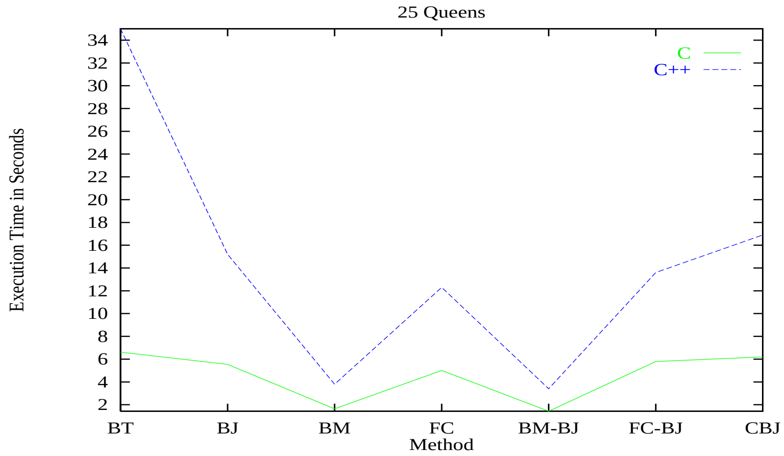

Figure 3: C++ versus C for the 25 Queens Problem

that were guaranteed to have a solution. We also included preprocessing before the actual search techniques were used. The routines for preprocessing achieved arc- and path-consistency. The following tables show the results of the different runs. The first column of each table is labeled V and represents the number of of variables in the CSP. D stands for the number of domain values in the domains of each variable in the CSP. Column 3 and 4 represent the constraint fraction (CF) and the label fraction (LF). CF represents the parameter p such that an edge in the constraint network is labeled with a nonlinear constraint with independent probability $p/100$. The label fraction LF represents the program parameter q such that two values of two given variables are constrained with independent probability $q/100$. In a nutshell, the greater CF is the more variables are constraining each other. The higher LF the more values within the domains of two constrained variables are compatible. The following columns represent the execution times of the programs. $C - Arc$ and $C - Path$ are the execution times in seconds for the C version using preprocessing for arc consistency and path consistency respectively. The other times reported are from the C++ version. Note that for these randomly generated CSPs the overhead is much more a constant than was the case for the N-queens problem.

V	D	CF	LF	C-Arc	C-Path	C++Arc	C++Path
20	15	90	70	0.01	1.74	0.1	6.8
25	10	70	70	0.01	2.22	0.1	8.6
30	10	70	70	0.02	6.32	0.1	23.5
40	5	90	90	0.02	1.26	0.1	4.0
40	10	70	70	0.05	34.64	0.1	123.1
40	15	70	70	0.05	10.59	0.2	40.6

Table 3: Preprocessing

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	40.88	36.35	36.19	103.6	103.1	103.2
25	10	70	70	101.98	102.37	64.0	296.4	296.3	183.8
30	10	70	70	30.88	30.85	15.92	87.2	86.5	44.4
40	5	90	90	396.81	404.96	211.73	1092.4	1095.8	586.5
40	10	70	70	270.14	271.21	63.54	767.5	767.1	178.6
40	15	70	70	285.55	270.21	258.45	768.1	768.0	730.7

Table 4: Backtracking

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	20.31	19.17	18.99	52.3	52.1	52.1
25	10	70	70	17.64	17.66	12.74	48.8	52.4	34.5
30	10	70	70	7.61	7.50	4.82	20.6	20.3	12.8
40	5	90	90	88.19	89.30	44.76	230.7	231.0	118.2
40	10	70	70	74.75	84.61	19.65	205.0	205.0	54.0
40	15	70	70	75.83	75.71	72.78	205.5	205.5	195.5

Table 5: Backjumping

Backmarking

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	19.73	19.77	19.67	46.3	45.8	45.9
25	10	70	70	53.0	52.98	33.53	119.9	120.3	85.7
30	10	70	70	17.92	17.91	9.55	39.2	39.2	20.6
40	5	90	90	203.67	205.23	113.87	441.8	441.4	239.7
40	10	70	70	174.76	176.18	45.50	359.7	360.9	95.9
40	15	70	70	140.99	141.31	134.98	304.8	304.3	290.8

Table 6: Backmarking

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	10.86	10.85	10.77	29.3	28.2	28.4
25	10	70	70	6.47	6.44	5.27	17.1	17.0	13.9
30	10	70	70	2.24	2.29	1.67	5.9	6.8	4.2
40	5	90	90	10.19	10.37	7.82	26.6	27.1	20.1
40	10	70	70	10.07	10.68	2.90	25.9	26.5	7.4
40	15	70	70	17.32	17.38	16.93	45.1	44.7	43.6

Table 7: Forward Checking

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	10.64	10.57	10.47	24.8	24.5	24.5
25	10	70	70	10.58	10.64	7.66	25.3	25.1	18.1
30	10	70	70	4.63	5.27	3.02	10.8	10.6	6.8
40	5	90	90	45.97	46.00	25.07	106.1	105.9	56.4
40	10	70	70	46.28	47.77	13.27	102.6	102.4	29.5
40	15	70	70	39.21	39.11	37.47	89.4	89.4	85.7

Table 8: Backmarking with Backjumping

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	12.52	11.92	11.82	30.4	29.8	30.4
25	10	70	70	6.82	6.78	5.50	17.6	32.8	23.4
30	10	70	70	2.38	2.41	1.79	6.1	6.1	4.3
40	5	90	90	8.19	8.35	12.85	21.2	21.7	16.2
40	10	70	70	44.04	9.91	2.86	23.9	24.4	7.1
40	15	70	70	17.60	17.74	17.27	44.0	43.8	42.6

Table 9: Forward Checking with Backjumping

V	D	CF	LF	C	C-Arc	C-Path	C++	C++Arc	C++Path
20	15	90	70	19.62	19.07	18.91	46.7	46.5	46.3
25	10	70	70	13.66	13.66	9.93	32.8	32.8	23.4
30	10	70	70	6.50	6.44	4.17	15.4	15.2	9.8
40	5	90	90	23.71	23.98	13.97	51.9	51.7	30.1
40	10	70	70	44.71	45.34	11.59	107.9	107.9	27.5
40	15	70	70	50.11	49.98	48.36	121.7	121.7	116.4

Table 10: Conflict directed Backjumping

8 Conclusion

Object oriented design allows the development of software that is safer, easy to use and easily extendable. If one wishes to write another search algorithm for CSP solving one can simply write the code for the new algorithm and "plug" it into the NETWORK class as a member function. If C code already exists one can reuse much of the procedural code for a C++ implementation. However one has to spend a great deal of time for a new object oriented design. C++ has the advantage of combining the efficiency of procedural programming with object oriented design. The drawback is that C++ also allows concepts that violate the OOD Paradigm. For example the concept of *friend functions* allows a function that is not a member function to manipulate the private data of a class. We have avoided friend functions for our implementation.

The practical results show that for the N-Queens problem back-marking is the superior method, especially when combined with back-jumping. One might wonder why the times for finding a single solution to the 25-Queens problem is smaller than the ones for finding a single solution to the 20-Queens problem. This is due to less backtracking in the part of the search space that is first explored. If we look at the second table we see that given a certain time limit we find many more solutions to the 20-Queens problem than to the 25-Queens problem. So as more of the search space is explored, the more the results mimic our expectations.

The C++ version however is inferior to the C version if the criterion is execution time. In fact C++ is much slower than pure C for the tested problem. The overhead comes from features like safe array access where 8 tests are performed on 4 indices whereas in the C version this is a simple access. Function call overhead is also a factor. For example NETWORK::access calls CONSTRAINT::access. The graphs that we plotted for the C++ and the C version have a similar appearance. But they also show that the overhead factor is not a constant, but varies from search algorithm to search algorithm for the N-queens problem. When other CSPs are investigated the overhead can be regarded as constant (see tables 3 to 10).

We have also implemented preprocessing algorithms to achieve arc and path consistency. These helped to reduce the running time for constraint satisfaction problems other than the N-Queens problem which is already path and arc consistent for $n > 3$.

9 Further Research and Discussion

In the near future we will extend our object oriented design to include a new class CSP. CSP will have a private data part that contains the constraint network of a given CSP as an object of type NETWORK and another object for the solution. SOLUTION will be a new class for solutions. We will also include preprocessing heuristics for domain value ordering and variable ordering.

10 Acknowledgements

The authors would like to thank the Institute for Robotics and Intelligent Systems (IRIS/PRECARN) for funding this research. This research is part of the Intelligent Scheduling Project IC-6.

11 How to get QDOOCS

QDOOCS is still under development. The most recent version is accessible under *ftp://ftp.cs.uregina.ca/pub/QDOOCS*.

References

[Gas79] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, CMU-CS-79-124, Technical Report, Carnegie Mellon University, Pittsburgh, 1979.

[Gin90] M.L. Ginsberg, W. D. Harvey, "Iterative Broadening", *Proceedings National Conference on Artificial Intelligence (AAAI)*, 1990, pp. 216–220.

[Gin93] M. L. Ginsberg, "Dynamic Backtracking", *Journal of Artificial Intelligence Research*, Vol. 1, 1993, pp. 25–46.

[Har80] R. M. Haralick, G. L. Elliot, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", *Artificial Intelligence*, Vol.14, 1980, pp. 263–313.

[jac92] Ken Jackson, *An Object Oriented Constraint Solver using Dynamic Backtracking*, Simon Fraser University, personal communication, see WWW page: http://fas.sfu.ca/cs/research/projects/IC-6/vvertm under the software heading.

[Pro92] Prosser P., *Backjumping Revisited*, Technical Report AISL-47-92, Department of Computer Science, University of Strathclyde, 1992.

[Poh94] Ira Pohl, *C++ for C Programmers*, Second Edition, Benjamin Cummings Publishing Co., 1994.

[Sos91] R. Sosic and J. Gu., *3,000,000 Queens in Less Than One Minute*, SIGART Bulletin, Vol. 2, 2, pp. 22-24, Apr, 1991.

[Sos90] R. Sosic and J. Gu. A Polynomial Time Algorithm for the N-Queens Problem. SIGART Bulletin, Vol. 1, 3, pp. 7-11, Oct, 1990.

[Tsa93] E. Tsang, *Foundations of Constraint Satisfaction*, Computation in Cognitive Science, Academic Press, 1993.

[vBe92] Peter van Beek, *A C Library for Solving Binary CSPs*, University of Alberta, personal communication, see WWW page: http://web.cs.ualberta.ca/ vanbeek/ under Software.