Artificial Intelligence Report 3rd Week-6th Problem

Le Kien Trung
03-120291, Department of Mechano-Informatics

November 17th, 2012

- This is the right version of 6th problem. The report I sent on October 29 was a wrong one. Please ignore that!!!
- The cryptarithmetic program also solves the 12th problem (when * and digit are taken into account)
- The 1st week report file was AI_120291_01.pdf, but in fact, it was about the 2nd problem. Sorry for inconvenience.
- Source files: nqueen.hpp, nqueen.cpp, cryptarithmetic.hpp, cryptarithmetic.cpp, backtrack.hpp, Makefile
- Programs are tested on Ubuntu 32/64 bit.

1 Backtrack class

In the previous report, I wrote about Truth Maintenance System (TMS) and the theoretical procedure to use Backtracking, an implementation of TMS, to solve Constraints Satisfaction Problems(CSPs). The pseudo code for Backtracking algorithm can be written as following 1.

```
function BACKTRACKING-SEARCH(CSP)
        return RECURSIVE-BACKTRACKING ({ }, csp)

function RECURSIVE-BACKTRACKING(csp)
        if assignment is complete then return assignment
        var <— SELECT-UNASSIGNED-VARIABLE(VARIABLE[csp], assignment, csp)
        for_each value in ORDER-DOMAIN-VALUES (var, assignment, csp) do
                if value is consistent with assignment according to CONSTRAINT[csp] then
                        add (var = value) to assignment
                        result <- RECURSIVE-BACKTRACKING(assignment, csp)
                        if result is not failure then return result
                        remove (var = value) from assignment
        return failure
```

Listing 1: Backtrack Pseudo Code

In order to solve CSPs such as Cryptarithmetic and N Queens, I created the Backtrack class in C++ as following.

```
class Backtrack{
protected:
    int DOMAIN_SIZE; // Size of the domain
    int NUM_VAR;     // Number of variables
    int numSolution; // number of solution
    int numAssignment; // number of assigments
    /***** Interface for implementation*****/
    // Check if an assignment is valid
    virtual bool IsValid(int index, int value) = 0;
    // Make an assignment and update state
    virtual void Update(int index, int value) = 0;
    // Undo the assignment and reset state
```

```
    virtual void Restore(int index, int value) = 0;
    // Print solutions
    virtual void PrintSolution() = 0;
public:
    virtual ~Backtrack(){}
    Backtrack():DOMAIN_SIZE(0), NUM_VAR(0), numSolution(0), numAssignment(0){}
    // Sovle by backtracking algorithm
    void Solve(int index){
        // A solution is reached when all variables are assigned
        if(index>=NUM_VAR){
            numSolution++;
            PrintSolution();  // Print the solution
            return;
        }
        // Loop all over domain of var[k]
        for(int value = 0; value < DOMAIN_SIZE; value++){
            // If an assigment is possible
            if(IsValid(index,value)){
                numAssignment++;
                // Make an assignment and update state
                Update(index,value);
                // Continue with next variable's assignment
                Solve(index+1);
                // Undo the assignment and reset state
                Restore(index,value);
            }
        }
    }
    int NumSolution() const{
        return numSolution;
    }
    int NumAssigment() const{
        return numAssignment;
    }
};
```

Listing 2: Backtrack Interface (backtrack.hpp)

Backtracking algorithm can be used to solve all CSPs by rewriting the four virtual methods.

2 NQueen class

The N Queens problem is represented as following:

```
int data[MAXSIZE]; // board data
int size; // board size
bool column[MAXSIZE]; // column available state
bool leftDiagonal[2*MAXSIZE-1]; // left diagonal available state
bool rightDiagonal[2*MAXSIZE-1]; // right diagonal available state
```

Listing 3: Data structure for N Queens (nqueen.hpp)

- Recursive Backtracking is computationally expensive, so N Queen with N $>$ 15 will not be considered.
- Index of data[] represents #row, value of data[] represents #column. Place a queen in to (4,5) means data[4]=5. This representation is much more efficient than to use NxN matrix, because the it already assumes all the queens must be on different rows, which reduces the number of constraints to check.
- Constraints are represented by three boolean array column[], leftDiagonal[] and rightDiagonal[]. There are N, 2*N-1, 2*N-1 number of columns, left diagonals and right diagonals on a NxN chess board, respectively. The constraint of N Queens problem is that no queen can attack another, namely, two arbitrary pair of queens must lay on different rows, columns, left and right diagonals. That is why methods inherited from Backtrack classes are implemented as following:

```
bool NQueen::IsValid(int row, int col){
    if(row ==0 && col>=(size+1)/2)
        return false;
    return column[col] == AVAILABLE && leftDiagonal[row+col] == AVAILABLE &&
        rightDiagonal[row-col+size-1] == AVAILABLE;
}
// Assign and update consistency
void NQueen::Update(int row, int col){
    data[row] = col;
    column[col] =leftDiagonal[row+col] = rightDiagonal[row-col+size-1] = !AVAILABLE;
}
// Unassign and restore consistency
void NQueen::Restore(int row, int col){
    data[row] = -1;
    column[col] =leftDiagonal[row+col] = rightDiagonal[row-col+size-1] = AVAILABLE;
}
```

Listing 4: Implementation of Backtrack class methods (nqueen.cpp)

  Using those arrays to represent constraints helps the program run faster and the algorithm look cleaner. There is other way to update and restore consistency, such as everytime a new queen is placed, check if it can attack another queen by comparing their positions. That way should work just fine, but slower.

- The last thing I want to mention is about the symmetry of N Queens solutions. In my code, I only use one symmetric property, which makes my code run twice fater than the normal one. Actually, solutions of N Queens problem holds much more symmetric properties, if all of them are used, we can expect faster running time of the program. Details about this topic can be found in 2.

Here is the running result:

```
@:~/backtrack$ make
g++    -c -o nqueen.o nqueen.cpp
g++ -o nqueen nqueen.o -lm
@:~/backtrack$ ./nqueen 10
#1:
Q * * * * * * * * *
* * Q * * * * * * *
* * * * * Q * * * *
* * * * * * * Q * *
* * * * * * * * * Q
* * * * Q * * * * *
* * * * * * * * Q *
* Q * * * * * * * *
* * * Q * * * * * *
* * * * * * Q * * *
#2:
* * * * * * * * * Q
* * * * * * * Q * *
* * * * Q * * * * *
* * Q * * * * * * *
Q * * * * * * * * *
* * * * * Q * * * *
* Q * * * * * * * *
* * * * * * * * Q *
* * * * * * Q * * *
* * * Q * * * * * *

...................

#724:
* * * * * Q * * * *
Q * * * * * * * * *
* * Q * * * * * * *
* * * * * * * * * Q
* * * * * * * Q * *
* Q * * * * * * * *
* * * Q * * * * * *
* * * * * * * * Q *
* * * * * * Q * * *
* * * * Q * * * * *
Execution's time: 0.04 (s)
```

3 Cryptarithmetic class

Cryptarithmetic also known as verbal arithmetic, is a type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters. The goal is to identify the value of each letter 3. Data structure of this problem:

```
    map<char,int> mapChar;  // map characters to values
    string str[3];          // 3 strings
    bool available[10];     // available[i]=true means i is available for assignment
    bool assigned[10];      // E.g (7a+bb=cc) assigned[7]=true
    string var;             // string of unique variables
    int length;             // length of longest string, also number of constraints to check
    int carry[10];          // Carry bit
```

Listing 5: Data structure for Cryptarithmetic (cryptarithmetic.hpp)

The efficiency of Backtracking algorithm can be measured by the number of conducted assignment. While there are 10!=3628800 map values permutations of 10 different characters, backtracking algorithm reduces that number significantly in this problem. Consider our traditional example:

send + more = money

This is a demonstration of how backtracking algorithm works. I started assign variables from left to right, top to down. In this example, the string **var=$msoenrdy**. This proves to be more efficient because the first character, m in this case, has very few possible values. In order to show assigments, please uncomment lines: 88,89,93,101 in cryptarithmetic.cpp.

```
+1: m <<---1       (Assign)
+2: s <<---0
-1: s--->>0    (Unassign)
+3: s <<---2
-2: s--->>2

.................

+46: d <<---4
-40: d--->>4
+47: d <<---7
+48: y <<---2
m --> 1
s --> 9
o --> 0
e --> 5
n --> 6
r --> 8
d --> 7
y --> 2
9567 + 1085 = 10652
-41: y--->>2

.................

-57: s--->>9
-58: m--->>1
```

The problems are solve after 58 times of assigments, which is very efficient. I do not use carry as direct variables. The constraints are checked forward from left to right. When the #i constraint is checked, carry[i] and carry[i+1] are assigned. If the assignment of carry[i] with #i constraint is violated with its assigment with #(i-1) constraint, constradictory occurs. The next listing shows how the IsValid(int,int) is implemented in cryptarithmetic:

```
// Check if assigment of value to var[index] is possible or not
bool CRYPTARITHMETIC::IsValid(int index, int value){
    // First characters cannot be Zero
    if(value==0){
        for(int i=0; i<3; i++){
            if(var[index]==str[i][0])
                return false;
        }
    }
    // Assign carry to UNKNOWN
    for(int i =0; i<=length; i++){
        carry[i] = UNKNOWN;
    }
    // If value is already used
    if(!available[value] || assigned[value])
        return false;
    // Temporary assignment
    mapChar[var[index]]= value;
    for(int i =0; i<length; i++){
        // If #i constraint is violated
        if(!IsValidForm(i)){
            mapChar[var[index]]= UNKNOWN;
            return false;
        }
    }
    // Undo assignment
    mapChar[var[index]]= UNKNOWN;
    return true;
//
bool CRYPTARITHMETIC::IsValidForm(int index){
    int tmp[3];
    // Return true if there is an unassigned character
    for(int i =0; i<3; i++){
        tmp[i] = mapChar[str[i][index]];
        if(tmp[i]==UNKNOWN)
            return true;
    }
    // Check 4 possible possibilities
    if (tmp[0]+tmp[1] == tmp[2]){
        if(carry[index]==1)
            return false;
        carry[index] = carry[index+1] = 0;
        return true;
    }
    if(tmp[0]+tmp[1]+1 == tmp[2]){
        if(index==length-1)
            return false;
        if(carry[index]== 1)
            return false;
        carry[index] = 0;
        carry[index+1] = 1;
        return true;
    }
    if(tmp[0]+tmp[1] == tmp[2]+10){
        if(index==0)
            return false;
        if(carry[index]==0)
            return false;
        carry[index]=1;
        carry[index+1] = 0;
        return true;
    }
    if(tmp[0]+tmp[1]+1 == tmp[2]+10){
        if(index == 0)
            return false;
        if(index==length-1)
            return false;
```

```
        if(carry[index]==0)
            return false;
        carry[index]=1;
        carry[index+1] = 1;
        return true;
    }
    return false;
}
```

Listing 6: IsValid(int, int) method implementation (cryptarithmetic.cpp)

Here is the running result.

```
@:~/Backtrack$ make
g++    -c -o cryptarithmetic.o cryptarithmetic.cpp
g++ -o cryptarithmetic cryptarithmetic.o -lm
@:~/Backtrack$ ./cryptarithmetic donald gerald robert
#1:
d --> 5
g --> 1
r --> 7
o --> 2
e --> 9
n --> 6
b --> 3
a --> 4
l --> 8
t --> 0
526485 + 197485 = 723970
Number of solutions 1
Number of assigments:3026
Execution's time: 0.03 (s)
@:~/Backtrack$ ./cryptarithmetic *boc abac 1a7cc
#1:
* --> 9
a --> 4
b --> 8
o --> 6
c --> 0
9860 + 4840 = 14700
#2:
* --> 9
a --> 6
b --> 8
o --> 4
c --> 0
9840 + 6860 = 16700
Number of solutions 2
Number of assigments:67
Execution's time: 0 (s)
```

4 Conclustion

Backtracking is very simple and straight forward algorithm, which can solve most of CSPs problems. I presented two examples, Nqueen and Cryptarithmetic here, but there are a lot more problems can be solved by this method, such as Maze, Sudoku. However, Backtracking has a flaw, its high complexity, because of recursion. For example, in N queens problems, if N=30, Backtrack cannot find any solution. In cases like that, we need more suitable algorithms such as Hill Climbing, Simulated Annealing or Genetic Algorithm.

References

[1] Stuart Russell, Peter Norvig *Aritifical Intelligence A Modern Approach* Prentice Hall 3nd Edition, December 2009

[2] `http://csc.columbusstate.edu/bosworth/SearchProblems/N_Queens.htm`

[3] `http://en.wikipedia.org/wiki/Constraint_satisfaction_problem`
`http://en.wikipedia.org/wiki/Verbal_arithmetic`