Shi Xingyue

Dr. Rao Yanghui

Artificial Intelligence

28 November 2018

Lab 9: Constraint Satisfaction Problem – N-Queen Problem

[**Abstract**] In this experiment, I implemented backtracking algorithm and its extension, forward checking algorithm, on N-Queen problem with C++. I then compare the time and space cost of these two algorithms and analyzed the improvement forward checking has beyond backtracking.

1 Methods

1.1 Algorithm

1.1.1 Backtracking Algorithm

Backtracking is a specialized version of depth first search. It reaches a solution by searching through the space of partial assignments. All variables are assigned after a solution is found so the order of assignments does not matter. A sub search tree is pruned and stopped extending when a constraint is violated.

1.1.2 Forward Checking Algorithm

Forward checking is an extension of backtracking search that employs "modest" amount of propagation by looking ahead. When a variable is instantiated, we check all constraints which have only one un-instantiated variables remaining (here is the difference between FC and BT, BT checks all constraints without un-instantiated variables). For that un-instantiated variable, we check all of its values in domain, pruning those value violating a constraint.

1.1.3 N-Queen Problem

This problem asks us to place N Queens on an N*N board so that no Queen can attack any other Queen.

Represented in a CSP, the "variables" and "constraints" of this problem are as below:

- Variables
  - N variables, one per row.
  - Denoted as "Q_i" whose value, "V_i", is the column the Queen in row is placed.
- Constraints
  - $V_i \neq V_j$ for all i ≠ j, namely cannot put two Queens in the same column.
  - $|V_i - V_j| \neq |i - j|$, namely cannot put two Queens in the same diagonal line.

1.2 Pseudo-code

1.2.1 Backtracking

Backtracking algorithm is extremely simple. The only difference between it and regular DFS is that it prunes the subtrees where a variable violates constraints.

```
 1  Backtracking(level)
 2          if all variables have been assigned
 3                  insert into solution set
 4                  return
 5          else
 6                  variable=PickUnassignedVariable()
 7                  for each d in domain[variable]
 8                          assign(level, variable, d)
 9                          if(isSatisfyConstraints())
10                                  Backtracking(level+1)
11                          else
12                                  Undo(level)
13          return
```

Code 1 Pseudo-code of Backtracking algorithm

1.2.2 Forward Checking

Based on Backtracking algorithm, Forward checking looks ahead to prune more unnecessary search. Backtracking only prunes the subtrees where constraints HAVE BEEN violated, while forward checking looks ahead to tell whether this assignment WILL make other variable's domain empty.

So here, we need implement function to find out the CURRENT domains of variables.

```
ForwardChecking(level)
        if all variables have been assigned
                insert into solution set
                return
        else
                variable=PickUnassignedVariable()
                for each d in curDomains[variable]
                        assign(level, variable, d)
                        //curDomains[] may change after this
assignment
                        //so refresh it
                        refreshCurDomains()
                        if (reachDWO() || !isSatisfyConstraints())
                                Undo(level)
                                //curDomains[] may change after this
undoing
                                //so refresh it
                                refreshCurDomains()
                                continue
                        else
                                ForwardChecking(level+1)
                return
```

Code 2 Pseudo-code of Forward Checking algorithm

1.3 Key Codes with Comments

1.3.1 Game Class

I pack the state of N-queens problem, searching functions and other utility functions all in the class "Game". By looking through the member data and functions of this class, you can quickly find out how I solved this problem 😊

```
1  class Game {
2  private:
3        //the size of board
4        int size;
5        //whether a variable has been assigned
6        bool* assigned;
7        //valid values of each variable
8        //Each variable in N-queen has same domain, so only needs one array
9        //seems unnecessary in this problem actually
10       //used in backtracking algorithm
11       int* domain;
12       //current domain of each variable
13       //used in forward checking
14       vector<vector<int>> curDomains;
15       //assigned value of each variable
16       int* values;
17       //the ORDER in which variables are assigned
18       vector<int> variables;
19       set<vector<int>> solutions;
20
21 public:
22       Game(int s):size(8){}
23       ~Game(){}
24
25       /*utillity functions*/
26       int PickUnassignedVariable(){}
27       bool allAssigned() {}
28       void printSolutions() {}
29       //undo the last assignment
30       void Undo(int level) {}
31       void assign(int level, int variable, int value) {}
32
33       /*pruning*/
34       bool columnConstraint() {}
35       bool diagonalConstraint() {}
36       //combine column constraint and diagonal constraint
37       bool satisfyConstraint() {}
38       bool reachDWO(){}
39
40       /*refresh current domains*/
41       void refreshCurDomains(){}
42       //refresh the current domain of selected variable
43       void refreshCurDomain(int variable){}
44
45       /*search algorithms*/
46       void Backtracking(int level) {}
47       void ForwardChecking(int level) {}
48
49
50 };//Game class
```

Code 3 Class Game.

Detailed definition of member functions are omitted here and will be presented in later sections.

1.3.2 Backtracking

Final codes of backtracking algorithm is quite similar to its pseudo-code after my finishing necessary utility functions and call them here, so I don not bother to explain it in detail.

```
void Backtracking(int level) {
        if (this->allAssigned()) {
                vector<int>solution;
                for (int i=0; i<this->size; i++){
                        solution.push_back(values[i]);
                }
                this->solutions.insert(solution);
        }

        else {
                int variable = this->PickUnassignedVariable();
                int d;
                for (int i = 0; i < this->size; i++) {
                        d = this->domain[i];
                        this->assign(level, variable, d);

                        //violate constraint
                        //cancel this assignment and try another value
                        if (!this->satisfyConstraint()) {
                                this->Undo(level);
                                continue;
                        }

                        //keep assigning for other variables
                        else {
                                this->Backtracking(level + 1);
                        }

                }
        }
        return;
}
```

Code 4 Codes of Backtracking algorithm

1.3.3 Forward Checking

```
1  void ForwardChecking(int level) {
2        if (this->allAssigned()) {
3              vector<int>solution;
4              for (int i=0; i<this->size; i++){
5                    solution.push_back(values[i]);
6              }
7              this->solutions.insert(solution);
8        }
9        else{
10             int variable = this->PickUnassignedVariable();
11
12
13             for (int i=0; i<(curDomains[variable]).size(); i++){
14                   int d=(curDomains[variable]).at(i);
15                   assign(level,variable,d);
16
17                   //refresh current domains after assignment
18                   for (int v=0; v<this->size; v++){
19                         if (assigned[v]==false && v!=variable){
20                               this->refreshCurDomain(v);
21                         }
22                   }
23
24                   if (reachDWO()||!satisfyConstraint()){
25                         this->Undo(level);
26                         //refreshment current domains after undoing
27                         for (int v=0; v<this->size; v++){
28                               if (assigned[v]==false && v!=variable){
29                                     this->refreshCurDomain(v);
30                               }
31                         }
32                         continue;
33                   }
34                   else {
35                         ForwardChecking(level+1);
36                   }
37
38             }
39             return;
40       }
   }
```

Code 5 Codes of forward checking algorithm

1.3.4 Utility Functions

1.3.4.1 Check Whether Constraints Are Satisfied

In N-queens problem. we need to satisfy 2 constraints, column constraint and diagonal constraint. They can be check be these 2 functions.

```
bool columnConstraint() {
	int* columns;
	columns = (int*)malloc(sizeof(int)*this->size);
	for (int i = 0; i < this->size; i++) columns[i] = 0;
	//how many queens in each column
	for (int i = 0; i < this->size; i++) {
		if (assigned[i] == true) {
			columns[values[i]]++;
		}
	}
	for (int i = 0; i < this->size; i++) {
		//more than 1 queens in this column
		//violate column constraint
		if (columns[i] > 1) {
			//cout << "columns " <<i<<" "<<columns[i]<< endl;
			//cout << "Violate column constraint!" << endl;
			return false;
		}
	}
	//satisfy constraint
	return true;
}
```

Code 6 Function to check whether column constraint is satisfied

```
bool diagonalConstraint() {
	vector<int> assignedrow;
	vector<int> assignedcol;
	assignedcol.clear();
	assignedcol.clear();
	for (int i = 0; i < this->size; i++) {
		if (assigned[i] == true) {
			assignedrow.push_back(i);
			assignedcol.push_back(values[i]);
		}
	}
	for (unsigned int i = 0; i < assignedrow.size(); i++) {
		for (unsigned int j = i + 1; j < assignedcol.size(); j++) {
			if (abs(assignedrow[i] - assignedrow[j]) == abs(assignedcol[i] - assignedcol[j])) {
				//cout << "Violate diagonal constraint!" << endl;
				return false;
			}
		}
	}
	return true;
}
```

Code 7 Function to check whether diagonal constraint is satisfied

1.3.4.2 Refresh Current Domains

After assigning a variable or undoing an assignment, the domain of each variable may change. this function is used to refresh current domains so that all the value in current domains will not violate constraints.

```
1  void refreshCurDomains(){
2          //for each curDomain in curDomains
3          for (unsigned int variable=0; variable<curDomains.size(); variable++){
4                  vector<int> refreshedDomain;
5                  refreshedDomain.clear();
6                  //assigned variable has empty domain
7                  if (assigned[variable]==true){
8                          refreshedDomain.clear();
9                  }
10                 //un-assigned variable
11                 else
12                         for (unsigned int j=0; j<this->size; j++){
13                                 //try assigning this variable with this value
14                                 int value=this->domain[j];
15                                 variables.push_back(variable);
16                                 assigned[variable]=true;
17                                 values[variable]=value;
18
19                                 if (this->satisfyConstraint())
20 {refreshedDomain.push_back(value);}
21
22                                 //undo this attempt
23                                 assigned[variable]=false;
24                                 variables.pop_back();
25                         }
26                 curDomains[variable]=refreshedDomain;
27         }
   }
```

Code 8 Function to refresh current domains

so that all the value in current domains will not violate constraints

2 Results and Analysis

2.1 Result Examples

2.1.1 Solution to 8-Queen Problem

An example solution get is shown in Figure.1.

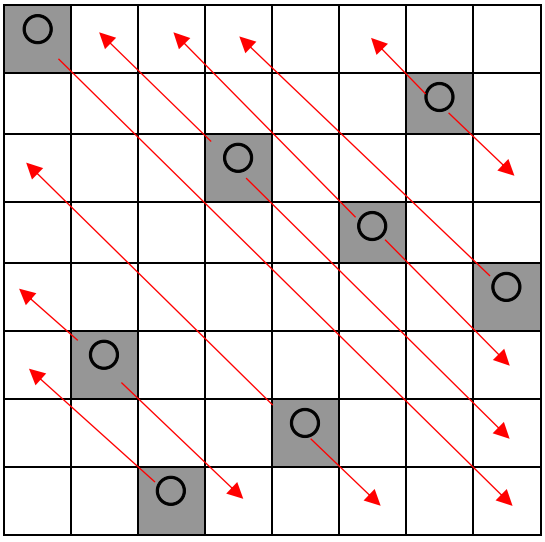

Figure 1 An example solution of 8-Queens Problem

Running results of BT and FC are shown in Figure.2 and Figure.3.

```
Please choose the size of board.
8
Please choose the search alogorithm:
BT-Backtracking
FC-Forward Checking
bt
Time Elapsed:0.002618
The search has found 18 solution(s).
```

Figure 2 Running result of 8-Queens using backtracking

```
Please choose the size of board.
8
Please choose the search alogorithm:
BT-Backtracking
FC-Forward Checking
fc
Time Elapsed:0.001391
The search has found 2 solution(s).
 0 6 4 7 1 3 5 2
 1 6 4 7 0 3 5 2
```

Figure 3 Running result of 8-Queens using forward checking

2.1.2 Solution to 10-Queen Problem

An example solution get is shown in Figure.4.

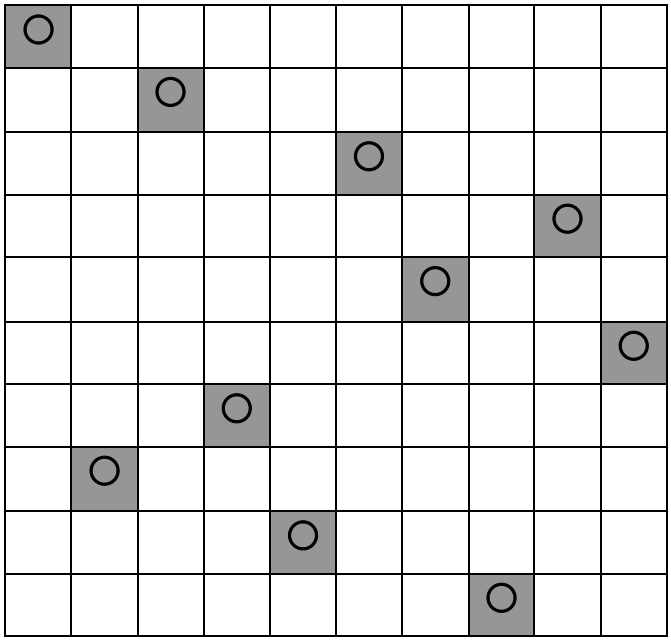

Figure 4 An example solution of 10-Queens Problem

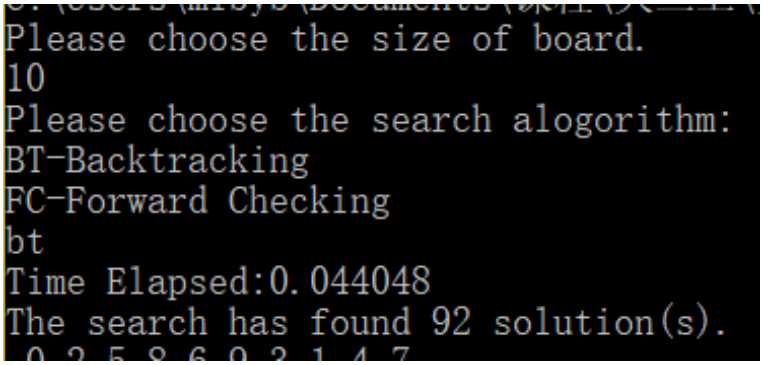

Figure 5 Running result of 10-Queens using backtracking

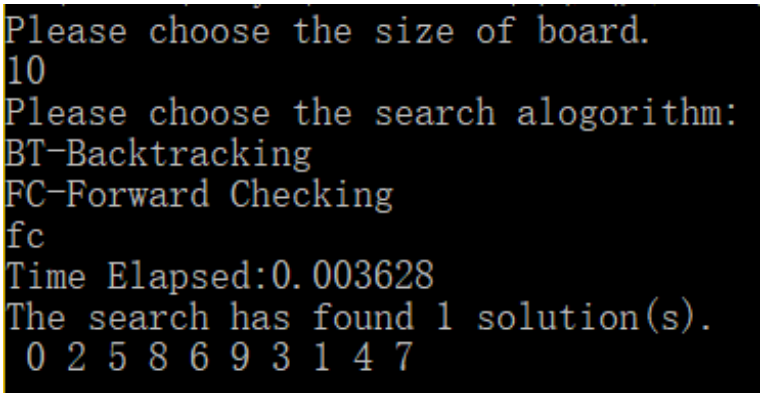

Figure 6 Running result of 10-Queens using forward checking

2.2 Evaluation Indices and Algorithms Comparison

2.2.1 Time Elapsed

I run each algorithm for 10 times and get the average elapsed time. Results show FC get obvious improvement on BT on the account of time.

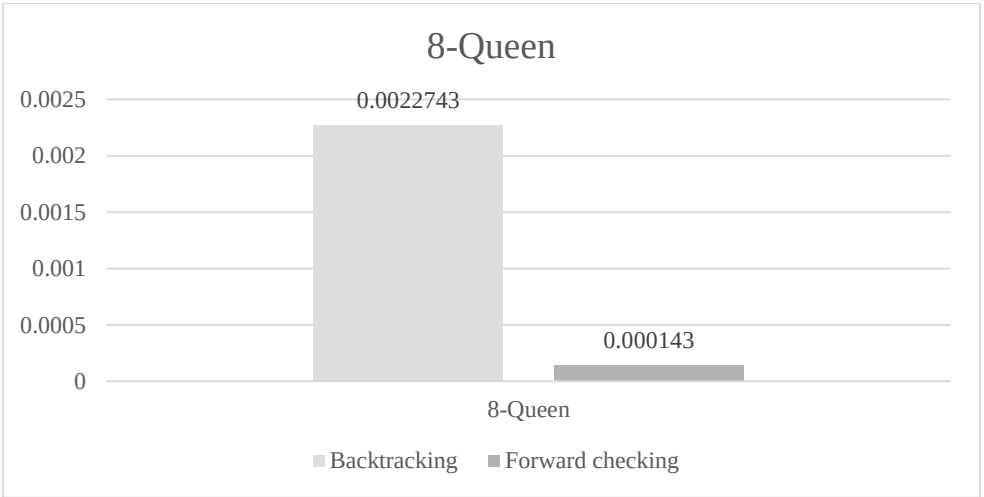

Table 1 Time elapsed comparison of BT and FC.

On 8-Queen problem.

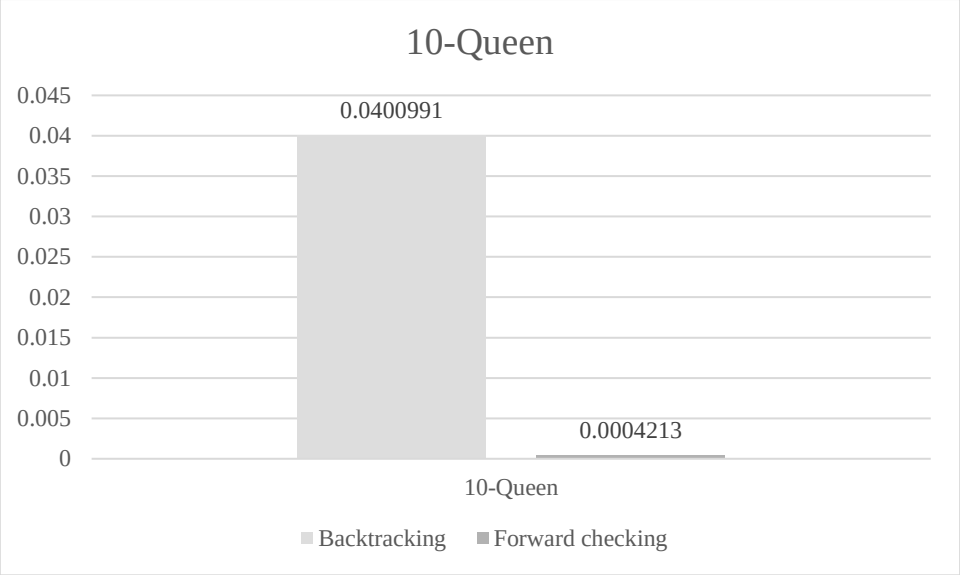

Table 2 Time elapsed comparison of BT and FC.

On 10-Queen problem.

2.2.2 Memory Used

Memory used when solving 10-Queen by BT and FC is shown in Figure.7 and Figure.8. This shows FC uses much less memory than BT because it has pruned many unnecessary subtrees.

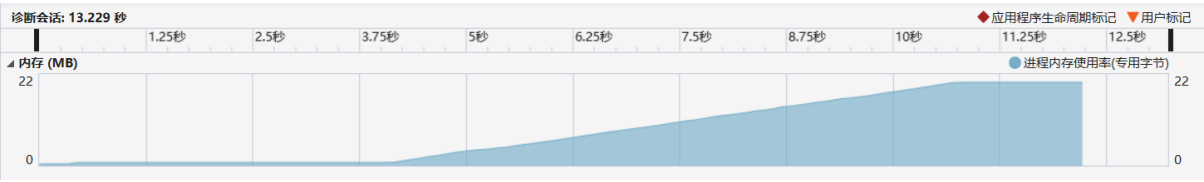

Figure 7 Memory used on 10-Queen using BT (≈22MB)

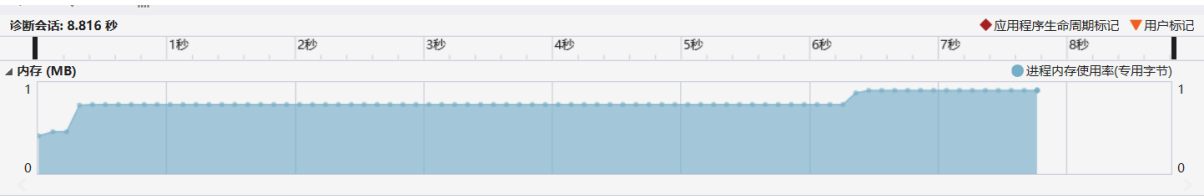

Figure 8 Memory used on 10-Queen using FC (≈1MB)