CSP: Definition, Creation, and Algorithms

Tianbing Lin, Scott Goodwin
University of Windsor
t.lin@ieee.org, sgoodwin@uwindsor.ca

Abstract

CSP problem is NP-complete and many problems are derivation of CSP. In this paper 4 different algorithms of CSP are introduced and implemented to compare with each other using Random CSP problems. Two kind of performance are compared: to find one solution and to find all solutions of the problem.

1. Introduction

A CSP problem includes some variables, and valid values for those variables (we call it domain of the variables) and conflict tables. We must find a solution to assign values to all the variables and those values must satisfy the conflict tables.

CSP problem is known as NP-complete problem.[2][3] We can't find a polynomial time algorithm until we can prove P=NP, but we've developed some algorithm to accelerate the process to find the solution of CSP.

2. Random CSP Creator and N-Queen Creator

To verify the performance of algorithms, we must have some data set to test. Random CSP Generator should be able to create different CSP problems using random values. N-Queen problem creator is also implemented to compare the performance.

3. Different Algorithms

According to [1][3][4], there're many algorithms to solve CSP problem. We introduced these 4 algorithms:

3.1 Algorithm 1: BackTracking

BackTracking is the basic algorithm to solve CSP. In every step, find a valid value to assign to current variable. If a valid value is found, assign it to current variable and go to next step. If there's no any valid value, back-track to the last variable to assign another value, expect another value of the last variable can lead to success of finding valid value for current variable. A valid value for current variable is a value that is not conflict with any assigned variable.

For example, in the Chart 1, Variable C can't find valid value, then we back-track to Variable B to change the value, then check Variable C again.

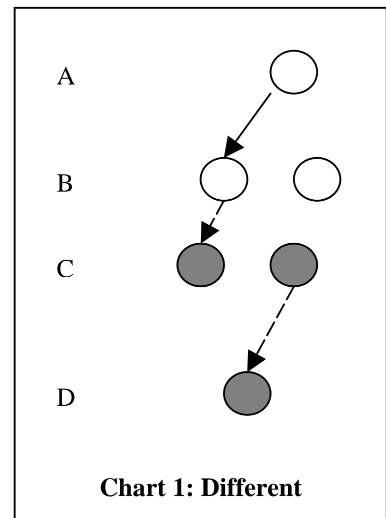


Chart 1: Different

BackTracking is a Depth-First Search algorithm in search-space. Time complexity: O(bd), when b is the average tightness of constraints, which is the branching factor of the searching tree, and d is the number of variables, the depth of the searching tree.

3.2 Algorithm 2: Forward Check

Forward Check sign an invalid value in the search space for future variables when current variable is assigned a valid value. If there's no any valid value for current variable, back-track to the last variable to assign another value. A valid value for the current variable is a value that is not signed invalid from the assigned variables.

For example, in the Chart 1, Values of Variable C is signed invalid and Values of variable D is signed invalid too because of the assignment of A and B, then we back-track to Variable B to change the value, change the related value of C and D again, then check Variable C again.

Forward Check has the same time complexity as BackTracking, and it's slower than Backtracking.

Because Forward Check exam the same nodes as BackTracking, but waste time to assign future nodes that may never been used. For example, because all the value of Variable C is invalid, it doesn't matter if values of Variable D are valid or not, but the values are already checked and signed when assign values for Variable A and B.

3.3 Algorithm 3: Back Jump

Forward Check is slower than BackTracking, but the idea is good. Actually, if we find all the values of Variable C are invalid because they are conflict with the value of Variable A, then we don't need to change the value of B to check C again. We can directly jump to another value of Variable A, skip other values of B (under A1 value). After we assigned another valid value for A, we check B1 again. This is the Back Jump algorithm.

Back Jump skipped some nodes, so theoretically it's faster than BackTracking and Forward Check.

3.4 Algorithm 4: Dynamic Backtracking

Dynamic Backtracking's idea is: When we back jump from C to A, if variable B is not conflict with A, then it's unnecessary to change B's value after A is re-assigned. In color-mapping problem, assume we assigned color of E1, E2 cities in east, then assigned color of W1, W2, W3 cities in west and return to assign color for E3, E4, E5 cities in east. When working in east cities E3, E4, E5, we found we have to change color of E2, using Back Jump we will have to change the color of the western cities too; But deploying Dynamic Backtracking algorithm, we can leave W1, W2, W3 alone, since they're not conflict with any of the east city.

This algorithm is faster than above algorithms, especially for CSPs has many variables.

4. Implementation

4.1 Implementation: Random CSP Problem

RandomCSP.java deploys binary CSPs, because every CSP problem can convert to binary CSP. Universal domain size is used, which means that all the variables have the same domain size.

Every instance of CSP includes:

```
ConsTable[] CTs;
int varNum;         // how many variables.
int iDomainSize;  // universal domain size.
```

And

```
int[] solution;
int[][] relation;
ConsTable[] CCTs; //ConflictConstraint
```

CCTs is the inverse of CTs. It is used to mark invalid values in Forwar Check, Back Jump and Dynamic BackTracking.

The varNum and iDomainsize are given by CSP creator, and the Constraint Tables are created in RandomCSP.java. relation[] and CCTs are initialized after the CTs are created. The structure of a constraint table is:

```
class ConsTable
{
        int varList[];
        int details[][];
}
```

The varList[2] gives the name of the two variables related to this constraint table.

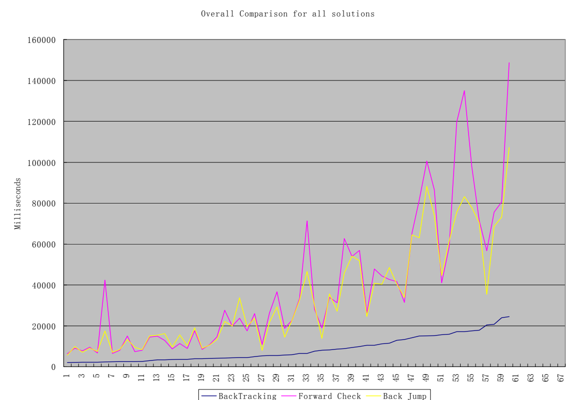
4.2 Average tightness of the constraints vs. Tightness of the CSP problem

Average tightness of the constraints must be given to create Random CSP. It is a value between 1-100, and the constraint density of every constraint is created by this average tightness and a random value.

5. Experimental Result Analysis

5.1 Overall Comparison.

From Chart 2, Overall Comparison for all solution, sorted by consumed time of BackTracking, we can see that the Forward Check is the slowest one to find all solutions, while the Back Tracking is the fastest one.



In Chart 3, Overall Comparison for one solution, we can get the same conclusion too. It's hard to see the performance of the Dynamic Backtracking, because it's lower than the Back Tracking.

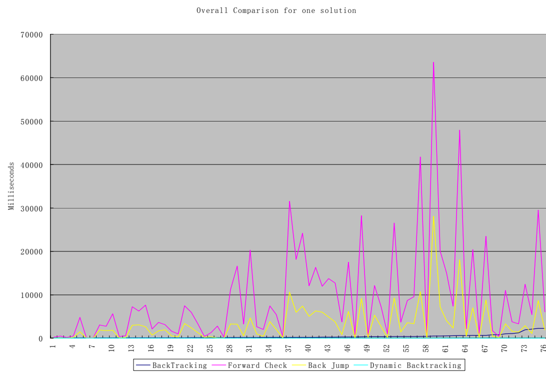
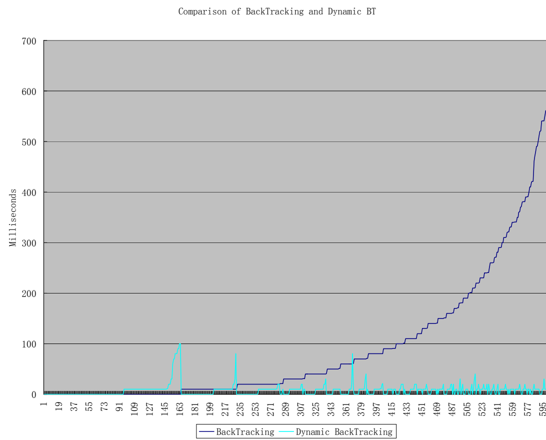
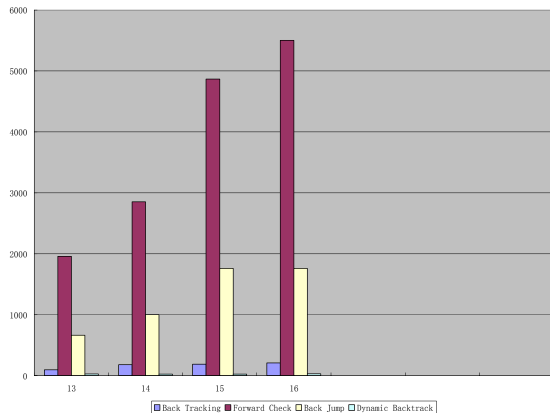


Chart 4 showed the comparison of Dynamic Backtracking and Backtracking. Dynamic Backtracking's performance is much better than all other algorithms.



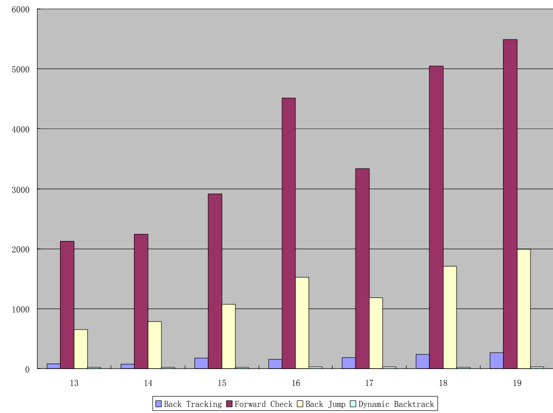
5.2. Different Size

5.2.1 Sort by Variable Number



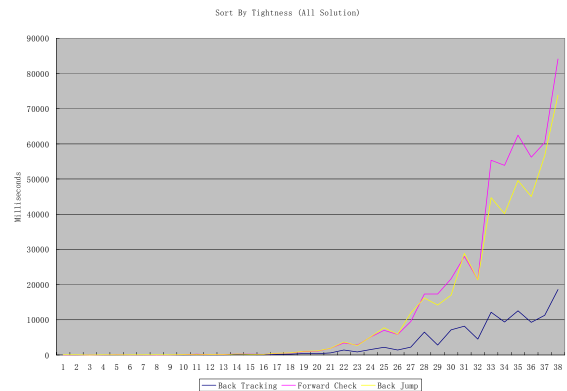
Apparently the average time to calculate CSPs is increasing with the number of variables increasing.

5.2.2 Sort by Domain Size

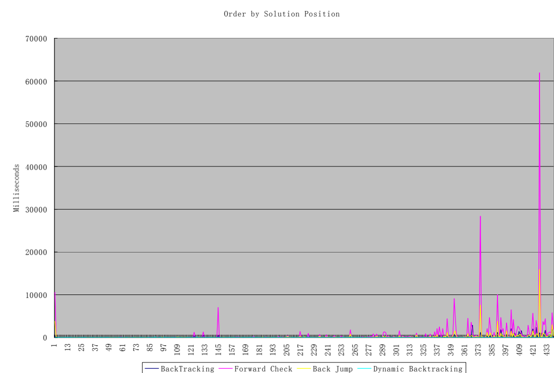


The average time to calculate CSPs is also increasing with the domain size increasing.

5.3 Sort by tightness



The tightness of the CSPs has great impact in finding all solutions. And when the tightness is greater, the time consumed to find all solutions is increasing faster.



All the four algorithms are based on Depth-First Search algorithm. A solution in the problem space has

its position in the DFS tree, and the position has impact in these four algorithms. Higher position leads to more branch and backtrack. From the chart above, we can see that the higher solution position is, more time is used to solve the CSP problem.

6. Conclusion

Constraint density can't assure the tightness of the CSP[6][7]. For example, we have CSP A and CSP B (domain size: 3, 3 variables):

CSP A			
Constraint 1		**Constraint 2**	
V1	V2	V2	V3
1	1	2	1
2	1	2	2
3	1	2	3

Density of Constraint 1 and Density of Constraint 2 are 33.3%. Solution is empty, so the tightness is 0/27

CSP B			
Constraint 1		**Constraint 2**	
V1	V2	V2	V3
1	1	1	1
2	1	1	2
3	1	1	3

Density of Constraint 1 and Density of Constraint 2 are 33.3%, Can have 9 solutions{(1, 1, 1) (1, 1, 2) (1, 1, 3) (2, 1, 1) (2, 1, 2) (2, 1, 3) (3, 1, 1) (3, 1, 2) (3, 1, 3)}, which means the tightness is 9/27=33%.

From the analysis we can have this conclusion:

The smallest constraint density of all the constraints is the maximum value of tightness the CSP can have. The exact value of the tightness depends on the relation of the constraint tables.

According to the analysis before, we can see that:

1. The Forward Check is the slowest one to find all solutions, followed by Back Jumping, while the Dynamic Back Tracking is the fastest one.
2. The average time to calculate CSPs is increasing with the number of variables increasing.
3. The average time to calculate CSPs is also increasing with the domain size increasing.
4. When the tightness is greater, the time consumed to find all solutions is increasing faster.
5. The higher the position of the solution in DFS tree is, more time is used to solve the CSP problem.

7. References

[1] Barták, R., "Constraint Programming: In Pursuit of the Holy Grail", *in Proceedings of the Week of Doctoral Students (WDS99),* Part IV, MatFyzPress, Prague, June 1999, pp. 555-564.

[2] Malek Mouhoub, *class notes of Artificial Intelligence*.

[3] Prosser, P., *Binary constraint satisfaction problems: Some are harder than others,* Proceedings ECAI-94 (11th European Conference on Artificial Intelligence)

[4] White, S, *Enhancing Knowledge Acquisition with Constraint Technology*, PhD Thesis, University of Aberdeen

[5] Pedro Meseguer, *CSP: Constraint Programming*

[6] Joe Culberson and Toby Walsh , *Tightness of Constraint Satisfaction Problems*

[7] Peter van Beek and Rina Dechter, *Constraint Tightness and Looseness versus Local and Global Consistency*