by Max
February 1, 2020

Backtracking

Problem Modelling

At present, problems are modelled after an abstract Problem class. This class contains the variables of the problem in the form of an int array with a size one greater than the total number of variables in the problem (variable 0 is assumed a fake variable (if the solver ends up checking this variable then no consistent solution was found).

Backtracking Solver

The BacktrackSolver class inherits the basic search procedure from Solver - an abstract class which also implements the SolverMethods label, unlabel, check and solution - since most binary constraint satisfaction solvers derive from a similar search procedure with differences in their labelling and unlabelling methods.

Search Method

The search algorithm for the solvers involved stem from the following algorithm as discussed in [1]: n

Algorithm 1: Search

```
search(n, status)
begin
    consistent := true
    status := UNKNOWN
    i := 1
    while status is UNKNOWN do
        if consistent then
            i := label(i, consistent)
        else
            i := unlabel(i, consistent)
    if i > n then
        status := SOLUTION
    else if i == 0 then
        status := IMPOSSIBLE
```

represents an integer value corresponding to the number of variables in the Problem instance. *consistent* represents the current state of the solution: i.e., whether or not it meets all of the current required constraints with its current variables's values. *status* too is associated with a Problem instance: if the current search has found a solution, know's if a solution is not possible or if either has yet to be determined. i is the index for the current variable being checked: as *variables* contains a dummy value at index 0 and contains n+1 elements, the search starts with index at 1. If the search loop ever reaches this dummy variable then there are no more possibilities to search and therefore, the search must terminate. On the other hand, if the search is consistent on the choice of a final variable then a solution is found.

Label

The label method listed prior has its algorithm given below. Here, square brackets are used to subscript indices such that, for example, variable[i] is the ith variable present. In this example, we assume that consistent is a global value, possibly tied to a Problem object the Solver interacts with. We initially assume that the problem is not yet consistent. Then, while we do not have a consistent solution, each variable value remaining to be checked in the current variable's domain is investigated. The value picked from this domain is checked against all other prior variable values set in the search run and if it is inconsistent with any such variables as per the given constraint, it is excluded from the current variable's domain. At the end of this search, if the current variable's value is consistent with all prior variables, we advance to the next variable - otherwise, we stay on the current variable for another iteration of search() to "unlabel" it.

Algorithm 2: Backtrack Label

```
label(i)
begin
    consistent := false
    foreach variable in current-domain[i] while not consistent do
        consistent := true
        for h := 1 to i - 1 and consistent do
            consistent := check(i, h)
            if not consistent then
                remove variable[i] from current-domain[i]
        if consistent then
            return i + 1
        else
            return i
```

Unlabel

Where i is the current variable's index, h is the previous variable's index and the square brackets denote subscripting such that, for example, current-domain[h] specifies the current consistent domain set for the previous variable. The symbol := is used in this context to mean "assigned the value of". We first begin with clearing and resetting the current variable's domain: no suitable variable value was found so we reset the domain to the original no consistent value was found given a previous variable's so we must assume any value may be correct if we decide to search this variable again for a given history. We then **backtrack** one variable to h and remove its current value from its domain, given that it just failed to find a value for the variable after it (an assumption in backtracking search). Our problem is then consistent providing that variable h still has values in its domain to choose from. We then finally backtrack by returning h as the current variable index.

The check(i,j) method mentioned in 2 relates to checking the constraint relationship bewtween variable i and variable j: if there are no constraints between them or the constraint between them holds for their current values, then check returns true. Otherwise, check returns false.

Algorithm 3: Backtrack Unlabel

```
unlabel(i)
begin
    h := i - 1
    current-domain[i] reset to original domain[i]
    remove variable[h] from current-domain[h]
    consistent := false if current-domain[h] is empty otherwise true
    return h
```

Trailing Method

Copying Method

Code Listings

The Problem of NQueens

```java
package uk.ac.gla.confound.problem;


import uk.ac.gla.confound.constraint.AlwaysTrueConstraint;
import uk.ac.gla.confound.constraint.Constraint;
import uk.ac.gla.confound.constraint.IndexPair;

import java.util.*;

public abstract class Problem {
    public boolean consistent;
    public int numVariables;

    public Variable[] variables; // For each variables[i] := queen row |-> col, of size numVar
        +1 as variables[0] is always null
    public Domain domain; // Assume all integer variables have the same domain

    public ArrayList<Integer>[] current;


    public Constraint[][] constraints; // Constraint for each variable pair

    public List<int[]> solutions;

    public Problem(int numVariables) {
        // Initialise the domain
        domain = new Domain(numVariables);

        current = new ArrayList[numVariables+1];
        current[0] = new ArrayList<Integer>();
        for (int i = 1; i < numVariables+1; i++)
            current[i] = domain.copy();

        this.numVariables = numVariables;
        variables = new Variable[this.numVariables + 1];

        for (int i = 0; i < this.numVariables+1; i++)
            variables[i] = new Variable(domain);


        // Initialise a constraint mapping such that the pair of indices of the concerned
            variables map to the
        // binary constraint they both share
        constraints = new Constraint[numVariables][numVariables];

        solutions = new ArrayList<>();
    }

```

```java

    public void print(int x)
    {
        StringBuilder s = new StringBuilder();
        for (int i = 0; i < numVariables; i++) {
            for (int j = 0; j < numVariables; j++) {
                if (j==solutions.get(x)[i])
                    s.append("*");
                else
                    s.append("  ");
            }
            s.append("\n");
        }
        for (int i = 0; i < numVariables; i++)
            s.append("==");
        s.append("\n");
        System.out.print(s.toString());
    }

    public boolean check(int i, int j)
    {
        //System.out.println("v["+i+"] v["+j+"]\t"+this.variables[i].value+" "+this.variables[
            j].value+"\t"+this.constraints[i-1][j-1].check());
        return constraints[i-1][j-1].check();
    }

    public void printAll()
    {
        for (int i = 0; i < solutions.size(); i++)
            print(i);
    }
}
```

```java
package uk.ac.gla.confound;


import uk.ac.gla.confound.constraint.AlwaysTrueConstraint;
import uk.ac.gla.confound.constraint.Constraint;
import uk.ac.gla.confound.problem.Problem;
import uk.ac.gla.confound.solver.*;


public class NQueens extends Problem {
    public NQueens(int numVars) {
        super(numVars);

        for (int i = 0; i < this.numVariables; i++) {
            for (int j = 0; j < this.numVariables; j++) {
                if (i != j) {
                    constraints[i][j] = new QueenConstraint(this, i, j);
                } else {
                    constraints[i][j] = new AlwaysTrueConstraint();
                }
            }
        }

    }



    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("USAGE: NQueens [Solver] [Num Queens] ");
            System.out.println("BacktrackSolver, ForwardCheckSolver, BackjumpSolver,
                ConflictBackjumpSolver, DynamicBacktrackSolver");
        } else {
            Problem nQueens = new NQueens(Integer.parseInt(args[1]));
            Solver s;
            switch (args[0]) {
                case "ForwardCheckSolver":
                    s = new ForwardCheckSolver(nQueens);
                    break;
                case "BackjumpSolver":
                    s = new BackjumpSolver(nQueens);
```

```
                    break;
                case "ConflictBackjumpSolver":
                    s = new ConflictBackjumpSolver(nQueens);
                    break;
                /*case "DynamicBacktrackSolver":
                    s = new DynamicBacktrackSolver();
                    break;

                 */
                case "BacktrackSolver":
                    // FALL-THROUGH
                default:
                    s = new BacktrackSolver(nQueens);

            }
            s.solve();
            s.report(nQueens);
        }
    }

    public static class QueenConstraint extends Constraint {
        int ix0;
        int ix1;

        public QueenConstraint(Problem p, int i, int j) {
            super(p, i, j);
            ix0 = i;
            ix1 = j;
        }

        @Override
        public boolean check() {
            return !var0.equals(var1)
                    && Math.abs(var0.value - var1.value) != Math.abs(ix0 - ix1);
        }
    }
}
```

Solver

```
package uk.ac.gla.confound.solver;

import uk.ac.gla.confound.problem.Problem;

/**
 * Abstract base class Solver defines common variables and methods between all extending
    solvers such as
 * Problem p, the problem class being solved
 * Status status, the current status of the search undertaken by the solver
 * int numIterations, the number of search iterations undertaken
 * int numSolutions, the number of consistent solutions found
 * ArrayList previousValues, a list of the previous values found
 */
public abstract class Solver implements SolverMethods {
    public static String NAME = "Base Solver";
    public Problem p;
    public Status status;

    int numIterations;
    int numSolutions;
    int backtracks;
    double duration;

    public Solver(Problem p) {
        numIterations = 0;
        numSolutions = 0;
        backtracks = 0;
        this.p = p;
    }

    public void solve()
    {
```

```java
        duration = System.currentTimeMillis();
        status = Status.UNKNOWN;
        p.consistent = true;

        int i = 1;

        while (status == Status.UNKNOWN) {

            for (int x = 1; x < p.numVariables+1; x++)
                if (x==i)
                    System.out.print("\033[1m"+p.variables[x].value+"\033[0m, ");
                else
                    System.out.print(p.variables[x].value+", ");
            System.out.println("\n"+p.consistent);

            if (p.consistent) {
                i = label(i);
            } else {
                i = unlabel(i);
                ++backtracks;
            }

            if (i > p.numVariables) {
                ++numSolutions;     // Now we've found one iteration, we try to find another
                    until there is none

                int[] solution = new int[p.variables.length-1];
                for (int j = 1; j < p.variables.length; j++) {
                    solution[j-1] = p.variables[j].value;
                }
                p.solutions.add(solution);

                i -= 1;
                p.consistent = false;
            } else if (i == 0)
                status = Status.IMPOSSIBLE;


            ++numIterations;
        }
        duration = System.currentTimeMillis() - duration;
    }

    public void report(Problem p)
    {

        System.out.println("Status report for "+NAME);
        System.out.println("#Iterations: "+numIterations);
        System.out.println("Duration: " + 0.001*duration + "(s)");
        System.out.println("Backtracks: " + backtracks);
        System.out.println("#Solutions: "+numSolutions);
        System.out.println("Solutions are as follows\n==========================");
        for (int[] arr: p.solutions) {
            System.out.print("[");
            for (int x : arr) {
                System.out.print(x + ", ");
            }
            System.out.println("\b\b]");
        }
        System.out.println("==========================");

    }

    public String solution()
    {
        StringBuilder s = new StringBuilder();
        s.append("Solution[");
        if (this.status == Status.IMPOSSIBLE){
            s.append("]");
            return s.toString();
        }
        for (int i = 1; i < this.p.variables.length-1; i++)
            s.append(this.p.variables[i] + ", ");
        s.append(this.p.variables[this.p.variables.length-1]+"]");
        return s.toString();
```

```
    }

}
```

```
package uk.ac.gla.confound.solver;

public interface SolverMethods {
    int label(int i);
    int unlabel(int i);
}
```

Backtrack Solver

```
package uk.ac.gla.confound.solver;


import uk.ac.gla.confound.NQueens;
import uk.ac.gla.confound.constraint.IndexPair;
import uk.ac.gla.confound.problem.Problem;
import uk.ac.gla.confound.problem.Variable;

import java.util.Scanner;

public class BacktrackSolver extends Solver {

    public BacktrackSolver(Problem p) {
        super(p);
        NAME = "BacktrackSolver";
    }

    /**
     * Label takes in a variable index and searches through that variable's current domain until a value consistent with
     * all preceding variables and their constraints is found or all possibilities are exhausted.
     * @param i The index of the variable to check
     * @return The succeeding variable's index if a solution is found, otherwise the current variable index
     */
    public int label(int i)
    {
        p.consistent = false;
        // Check each value variable[i] *could* be until we have a consistent value or we exhaust all current possibilities
        for (int j = 0; j < p.current[i].size() && !p.consistent; j++) {
            p.variables[i].value = p.current[i].get(j);


            p.consistent = true;
            // Run through all previously chosen variables and check if they are all consistent with the current candidate
            // variable[i]
            for (int h = 1; h < i && p.consistent; h++) {
                // Remove value from candidates on constraint failure
                if (!(p.consistent = p.check(i, h))) {
                    p.current[i-1].remove(Integer.valueOf(p.variables[i-1].value));
                }
            }
        }

        if (p.consistent)
            return i + 1;
        else
            return i;
    }

    /**
     * Unlabel is called when the current solution thus far is inconsistent by the introduction of variable i's value.
```

```java
     * The current domain of variable i is reset to the original, full domain and the current domain of the preceding
     * variable has the value of the preceding variable removed from it. If this causes the current domain of the
     * preceding variable to become empty then the overall solution is still inconsistent.
     * @param i The index of the current variable to check
     * @return The index of the preceding variable
     */
    public int unlabel(int i)
    {
        int h = i - 1;
        // Rather than store any domain set for the fake variable, we assign the domain as a null pointer and just
        // check for when we try to unlabel the first possible variable
        p.current[i] = p.domain.copy();
        p.current[h].remove(Integer.valueOf(p.variables[h].value));
        p.consistent = !p.current[h].isEmpty();

        return h;
    }


    public static void main(String... args) {
        int n = 0;

        if (args.length > 1 && args[1].startsWith("-n=")) {
            n = new Scanner(args[1]).nextInt();
        } else {
            System.out.println("Usage: Solver.java -n=[NUM]");
        }


        Problem nQueens = new NQueens(n);
        Solver solver = new BacktrackSolver(nQueens);
        solver.solve();
        solver.report(nQueens);
    }

}
```

Forward-Check Solver

The Forward-Check Solver is an extension to the chronologically backtracking solver discussion in the previous section. In fact, an attempt has been made to generalise the search method body such that the differences between the search algorithms studied are made more prominent. This is intended to suggest precisely where performance improvements and hits come from.

As with Backtrack Solver, the Forward-Check Solver advances on and retreats from instantiating variables through label and unlabel methods. The Forward-Checking Solver, being a modification of the Backtrack Solver, has very similar label and unlabel methods; however, additions have been added to attempt to backtrack less.

When the solver instantiates a value it then "looks forwards" at each variable not yet visited. Looking forward involves modifying the current domain of the future variable such that all values which conflict with the current variable's value are removed. The changes made are recorded in an array of stacks. Each element corresponds to each variable in the problem and stores the indices of the future variable's whose domain was modified by a forward check in order of last modified. This is to ensure that when a forward check clears a variable's entire domain, the changes to that domain can be undone and the current variable re-instantiated with a different value. The elements removed by a forward check are stored in an array of stacks, each corresponding to the variable which had its values

Algorithm 4: Checking forward

```
checkForward(i, j)
begin
    h := i - 1
    current-domain[i] reset to original domain[i]
    remove variable[h] from current-domain[h]
    consistent := false if current-domain[h] is empty otherwise true
    return h
```

Algorithm 5: Undoing Reductions

```
unlabel(i)
begin
    h := i - 1
    current-domain[i] reset to original domain[i]
    remove variable[h] from current-domain[h]
    consistent := false if current-domain[h] is empty otherwise true
    return h
```

removed. A forward check involves a call of checkForward, which returns the inverse of whether or not it has cleared the variable's domain. Updating the current domain to remove domain values that are not permitted is done via updatedCurrentDomain. The changes made to a variable's domain can be undone via undoReductions.

References

[1] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 1993.

Algorithm 6: Updating the current domain using reductions

```
updatedCurrentDomain(i)
begin
```