by Max
March 4, 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

```
1  search(n, status)
2  begin
3      consistent ← true
4      status ← UNKNOWN
5      i ← 1
6      while status is UNKNOWN do
7          if consistent then
               i ← label(i, consistent)
8          else
9              i ← unlabel(i, consistent)
10     if i > n then
11         status ← SOLUTION
12     else if i = 0 then
13         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, knows 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

```
1  label(i)
2  begin
3      consistent ← false
4      foreach variable in current-domain[i] while not consistent do
5          consistent ← true
6          for h ← 1 to i - 1 and consistent do
7              consistent ← check(i, h)
8              if not consistent then
9                  remove variable[i] from current-domain[i]
10         if consistent then
11             return i + 1
           else
12             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. 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

```
1 unlabel(i)
2 begin
3     h ← i - 1
4     current-domain[i] reset to original domain[i]
5     remove variable[h] from current-domain[h]
6     consistent ← false if current-domain[h] is empty otherwise true
7     return h
```

Constraints

How Constraints are Related to Variables

Constraints are represented by classes which extend Constraint, or CompositeConstraint: the base Constraint class holds a pair of Variable object references and a boolean check() method, which returns true. Classes which extend Constraint must override the check() method to return true if and only if the behaviour it encapsulates holds. For example, the NeqConstraint check() method returns true if the Variable values it refers to are inequal. Conversely, the binary EqConstraint class checks if the variable values involved are equal.
Constraints are stored globally in the elements of an (N+1)x(N+1) array (where N equals the number of variables involved)[1] as part of a ConstraintList (a wrapper class for an ArrayList of Constraint objects) When checking if all constraints between variable[i] and variable[j] hold, constraint[i][j].check() is called, which proceeds to check all constraints in that constraint list, returning false immediately if any constraint did not hold and true otherwise. Constraints between variables i and j are added by appending the new constraint to constraint[i][j]/constraint[j][i].

Compound Constraints

Compound Constraints are represented by a CompoundConstraint base class, which represents multiple constraints under one check() method. The base Constraint is a binary constraint, which limits problem representation to pairs of variables. To overcome this, constraints involving more than two variables may be represented as a compound constraint. For example, the constraint *v1 != v2 != v3* may be represented as a compound constraint of *v1 != v2*, *v1 != v3* and *v2 != v3*. The check() of the compound constraint requiring all check() methods of its constituent constraints to hold. That is, for the compound constraint to hold true, *v1 != v2*, *v2 != v3*, **and** *v1 != v3* must all hold true. Compound constraints are added on to the constraint lists for the variables involved, just as binary constraints are. Compound constraints may consist of AndConstraint or OrConstraint where the return value for the check function is a logical and/or (respectively) between all binary constraints listed within. This allows some constraints such as the inequality discussed to short-circuit: if v1 != v2, then we need not check v1 != v3 or v2 != v3, saving time and resources.

Statistics Reporter Class

Statistics regarding a particular solver's search over a given problem are logged within an external statistics class, which is associated with the solver. This class records various pieces of information regarding the search, allowing analysis of search over given problems to be analysed at a later date.

[1] The additional row is present to account for the "nil" Variable at index 0 used within the search algorithm to signify there is no solution. The constraint lists at constraint[0][0..N+1] and constraint[0..N+1][0] are null.

Statistics include the number of nodes visited (i.e., the total number of iterations made in the search); the number of times the search backtracked; the time taken to find a solution (or multiple solutions); the number of times the search algorithm visits each variable (providing an indication as to the profile of the search, where problem variables may lie). The collection and analysis of such data will be discussed in a later section.

Arc-consistency and Search Heuristics

Since the objective of the study is to analyse and compare multiple search algorithms and nothing more, metaheuristics and other measures to simplify a constraint satisfaction problem and reduce the required solve time are not considered or implemented to assure that only the performance of the search algorithm is considered. If AC3 were implemented, it is unsure what effect this would have on solve time and if this effect is predictable or relevant for a comparison between solvers.

Code Listings

The Problem of NQueens

Solver

Backtrack Solver

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 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. It is hoped that this forward-checking allows the solver to be informed of incompatible values in advance, reducing the amount of unnecessary backtracking in exchange for more checking per variable decision.

References

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

Algorithm 4: Checking forward

```
1 checkForward(i, j)
2 begin
3     h ← i - 1
4     current-domain[i] reset to original domain[i]
5     remove variable[h] from current-domain[h]
6     consistent ← false if current-domain[h] is empty otherwise true
7     return h
```

Algorithm 5: Undoing Reductions

```
1 unlabel(i)
2 begin
3     h ← i - 1
4     current-domain[i] reset to original domain[i]
5     remove variable[h] from current-domain[h]
6     consistent ← false if current-domain[h] is empty otherwise true
7     return h
```

Algorithm 6: Updating the current domain using reductions

```
1 updatedCurrentDomain(i)
2 begin
3     current-domain[i] ←
4     current-domain[i] ← domain[i]
5     for reduction ∈ reductions[i] do
6         current-domain[i] ← current-domain[i] - reduction
```