CSP Solver

Carter J. Bastian

February 21, 2016

Introduction

In this assignment, a framework for solving Constraint Satisfaction Problems (CSPs) was implemented and tested on a series of challenges. The methods used include backtracking, AC-3 arc consistency enforcement, MRV and LCV value-ordering heuristics, and Forward checking / MAC-3 inference.

Backtracking

The concept behind backtracking is to select a variable assignment, complete it (and run an inference function on it), and then continue until a dead-end is reached. When such a dead-end is found, the state of the CSP is reverted to a higher level and a different value is attempted.

In order to implement this type of backtracking, I kept track of which assignments were made and domains were altered throughout each level of recursive depth in such a way that I could undo each alteration at that depth before returning.

My full listing of this implementation is as follows:

```
private Map<Integer, Integer> backtracking(Map<Integer, Integer> partialSolution, int
  if (depth >= MAX_CAPACITY) {
    System.err.println("Maximum␣recursion␣depth␣exceeded");
    return null;
  }

  /* Check if this level of depth has a rLog and aLog yet */
  Map<Integer, Set<Integer>> removed = removedLogs.get(depth);
  Map<Integer, Integer> added = addLogs.get(depth);

  Integer unassignedVar = selectUnassignedVariable(partialSolution);

  if (unassignedVar == -1)
    return partialSolution; // All variables have been adequately assigned

  List<Integer> domainVals = orderDomainValues(unassignedVar, partialSolution);

  Integer[] domainArray = new Integer[domainVals.size()];
  int i = 0;
  for (Integer x : domainVals) {
    domainArray[i] = x;
    i++;
  }
```

```
int domainCount = domainArray.length;

Map<Integer, Integer> result;

// Loop through each potential value in the unassigned variable's domain
for (i = 0; i < domainCount; i++) {
  Integer x = domainArray[i];


  // Make the assigment
  partialSolution.put(unassignedVar, x);
  added.put(unassignedVar, x); /* Add this entry to the aLog */
  incrementNodeCount();

  // Do the inference (and ensure this won't cause directly awful issues)
  if (inference(unassignedVar, x, partialSolution, removed, added)) { // Changes par
    removedLogs.add(new HashMap<Integer, Set<Integer>>());
    addLogs.add(new HashMap<Integer, Integer>());
    result = backtracking(partialSolution, depth + 1); // Recurse on the updated sol

    // Check for success
    if (result != null) {
      /* We've succeeded! Don't undo anything */
      return result;
    }
  }
  // Undo all of the changes we just did so we can try again at this depth
  /* Remove entries based on the aLog */
  Set<Integer> addedSet = new HashSet<Integer>(added.keySet());

  // Loop through each variable added to the partial solution
  for (Integer r : addedSet) {
    partialSolution.remove(r); // Remove it from the partial solution
    //added.remove(r); // Clear that entry in the aLog
  }

  /* Remove inference's changes based on the rLog */
  Set<Integer> removedSet = new HashSet<Integer>(removed.keySet());
  // Loop through all variable's whose domains were modified
  for (Integer r : removedSet) {
    Set<Integer> fromDomainX = removed.get(r);
    // Loop through each value removed from X's domain
    for (Integer gone : fromDomainX) {
      this.Variables.get(r).add(gone); // Add it back
    }
    removed.remove(r); // Get rid of the whole entry in the rLog
  }


  removed = new HashMap<Integer, Set<Integer>>();
  added = new HashMap<Integer, Integer>();
```

```
    /* Keep the removal of the value we just tried from this var's domain? */
  }

  // All the changes from this depth level have been undone
  if (depth >= 1) {
    removedLogs.remove(removedLogs.size() - 1);
    addLogs.remove(removedLogs.size() - 1);
  }

  return null;
}
```

MRV

In addition to backtracking, the minimum remaining values heuristic was implemented when selecting unassigned variables such that the variable which would be assigned to next will always be the one with the fewest available values.

A Partial implementation of this is as follows:

```
Map<Integer, Integer> constraintAmount = new HashMap<Integer, Integer>();

// Iterate through each potential variable in the problem
for (Integer x : Variables.keySet()) {
  // If it hasn't been assigned to yet, let's track it
  if (partialSolution.get(x) == null) {
    constraintAmount.put(x, Variables.get(x).size());
  }
}

int maxVal = -1;
Integer maxInt = new Integer(-1);

for (Integer x : constraintAmount.keySet()) {
  if (constraintAmount.get(x) > maxVal) {
    maxVal = constraintAmount.get(x);
    maxInt = x;
  }
}

return maxInt;
```

LCV

Similarly, the Least Constraining Value heuristic was used to choose the order in which values would be assigned to the selected variable. This was implemented by looping through the unassigned variables and summing the number of active constraints with the each potential value for assignment. The value with the fewest said constraints is selected.

A partial listing of this implementation is as follows:

```
Set<Integer> vals = Variables.get(var);
List<Integer> result = new LinkedList<Integer>();
Map<Integer, Integer> constraintAmount = new HashMap<Integer, Integer>();

// Loop through each value in the variable's domain
for (Integer v : vals) {
  int constraintCount = 0;

  // Loop through the remaining variables (that aren't var)
  for (Integer x : this.Variables.keySet()) {
      // Ensure this is an unassigned variable that's not var
      if (x != var && (! partialSolution.containsKey(x))) {
        if (this.Variables.get(x).contains(v))
          constraintCount += this.Variables.get(x).size() - 1;
        else
          constraintCount += this.Variables.get(x).size();
      }
  }
  constraintAmount.put(v, constraintCount);
}

// Sort the List appropriately
Set<Integer> toList = new HashSet<Integer>(constraintAmount.keySet());
for (int i = 0; i < toList.size(); i++) {
  int leastConstraint = Integer.MAX_VALUE;
  int leastInt = Integer.MAX_VALUE;
  // Loop through the constraintAmount keys
  for (Integer val : constraintAmount.keySet()) {
    if (constraintAmount.get(val) < leastConstraint) {
      leastConstraint = constraintAmount.get(val);
      leastInt = val;
    }
  }

  result.add(leastInt);
  constraintAmount.remove(leastInt);
}

return result;
```

AC-3

The AC-3 algorithm was implemented to enforce arc consistency. In this algorithm, every constraint (or arc) is checked against the domains of said arc's variables so as to enforce the potential consistency of every value in the domain of a variable.

A partial listing of this implementation is as follows:

```
/* AC-3 Implementation */
Queue<hashPair> arcQ = new LinkedList<hashPair>();

// Make the arc Queue
```

```
        for (hashPair arc : Constraints.keySet())
          arcQ.add(arc);

        // Loop while the queue is not empty
        while (! arcQ.isEmpty()) {
          hashPair arc = arcQ.remove();

          if (revise(arc, removed)) {
            if (this.Variables.get(arc.getX()).size() == 0)
              return false;
            for (hashPair modded : Constraints.keySet()) {
              if ((modded.getX() == arc.getX() && modded.getY() != arc.getY()) ||
                  (modded.getY() == arc.getX()))
                arcQ.add(modded);
            }
          }

          if (reviseBackward(arc, removed)) {
            if (this.Variables.get(arc.getY()).size() == 0)
              return false;
            for (hashPair modded : Constraints.keySet()) {
              if ((modded.getY() == arc.getY() && modded.getX() != arc.getX()) ||
                  (modded.getX() == arc.getY()))
                arcQ.add(modded);
            }
          }

        }

        return true;
```

As can be seen in the listing above, this implementation relies heavily on both backward and forward versions of the `revise`method. This method checks each value in the domain of a variable against the possible values it can take on given a specific constraint. If no valid result is available, the value is removed from the domain.

An implementation of the revise function is listed in full below:

```
    private boolean revise(hashPair arc, Map<Integer, Set<Integer>> removed) {
        Integer id1 = arc.getX(), id2 = arc.getY();
        boolean revised = false;
        Iterator<Integer> domainIt = this.Variables.get(id1).iterator();

        // Loop through each value in the domain of X_i
        while (domainIt.hasNext()) {
//        for (Integer x : this.Variables.get(id1)) {
          Integer x = domainIt.next();
          boolean useless = true;
          for (Integer y : this.Variables.get(id2)) {
            hashPair attempt = new hashPair(x, y);
            if (Constraints.get(arc).contains(attempt))
                useless = false;
          }
```

```
            if (useless) {
//              System.out.println("Removing a Var in revise forward");
              if (removed.containsKey(id1)) {
                removed.get(id1).add(x);
              } else {
                removed.put(id1, new HashSet<Integer>());
                removed.get(id1).add(x);
              }
              domainIt.remove();
              revised = true;
            }
          }
          return revised;
    }
```

Notice that the AC-3 functionality is built in such a way that it tracks the modifications it causes. As such, it is called directly from the inference function in order to implement the MAC-3 algorithm.

Foward Checking and MAC-3

In forward checking, whenever a variable is assigned, consistency is established for it. In order to do this, each constraint involving the newly-assigned variable is checked against the non-assigned variable in said constraint for consistency.

A partial listing of the Forward-Checking implementation is as follows:

```
    // Loop through each constraint in the CSP
    for (hashPair scope : Constraints.keySet()) {
      incrementConstraintCheck();
      // Check if this constraint involves the newly assigned value
      if (scope.getX() == var) {
        // this var is the first item in the pair
        tempVar = scope.getY();

        if (! partialSolution.containsKey(tempVar)) {
          // Get the possible set of acceptable constrained pairs
          relation = Constraints.get(scope);

          // Loop through each item in the domain of the constraint-connected
          // variable to check for consistency.
          domainIter = this.Variables.get(tempVar).iterator();

          while (domainIter.hasNext()) {
            potentialVal = domainIter.next();
            tempConstraint = new hashPair(value, potentialVal);

            // If this constraint is not in the acceptable set of constraints,
            // remove this value from the domain
            if (!relation.contains(tempConstraint) ) {
              domainIter.remove();
              if (removed.containsKey(tempVar)) {
                removed.get(tempVar).add(potentialVal);
              } else {
                removed.put(tempVar, new HashSet<Integer>());
                removed.get(tempVar).add(potentialVal);
              }
              /* Mark this as removed in the rLog */
            }

          }
        }
      } else if (scope.getY() == var) {
        // This var is the second item in the pair
        tempVar = scope.getX();

        if (! partialSolution.containsKey(tempVar)) {
          // Get the possible set of acceptable constrained pairs
          relation = Constraints.get(scope);

          // Loop through as above
          domainIter = this.Variables.get(tempVar).iterator();

          while (domainIter.hasNext()) {
            potentialVal = domainIter.next();
            tempConstraint = new hashPair(potentialVal, value);

            if (!relation.contains(tempConstraint)) {
              domainIter.remove();
              /* Mark this as removed in the rLog */
              if (removed.containsKey(tempVar)) {
                removed.get(tempVar).add(potentialVal);
              } else {
                removed.put(tempVar, new HashSet<Integer>());
                removed.get(tempVar).add(potentialVal);
              }
            }
          }
        }
      }
    }
```

Notice that, following this, the inference function also has the potential to fail (in the event that an unassigned variable has no values in its domain) or to recurse (in the event that an unassigned variable has only one value in its domain).

Results

Overall, all of the tests including N-Queens (up to 41), Sudoku (up to the difficult board), and the Circuit Board Problem (with the default example).

Furthermore, each of the additional features and heuristics improved the search significantly. To exemplify this, we'll look at the results of running the N-queens problem with 20 queens.

With solely Forward Checking Backtracking, the CSP was solved searching 151195 nodes and checking 28727050 constraints. However, with MRV implemented, the number of nodes reduced to 17938 and the constraints checked reduced to 3408220. Further, adding in the LCV heuristic reduced the nodes explored to 6833 and the constraints checked to 1298270. Finally, when AC-3 and MAC-3 are added into the mix, the results were significantly lower, at 1706 explored nodes and only 324140 checked constraints.

These improvements were also seen in the Sudoku challenge, improving the nodes explored on the medium puzzle from 222 to 137, and the constraints checked from 359640 to 221940.

Circuit Board Problem

To implement the Circuit Board problem, I designed the code in `Circuit.java`such that a Variable is a piece (implemented with properties in the `CircuitPiece`class) and its domain is the set of locations (starting with 0 at the top left corner) that its top left corner can be placed in.

As such, to create the domains of all of the pieces, I simply had to loop through the available top-left positions of the board and use some tricky index arithmetic as shown below:

```
// Generate domains of all of the pieces
for (int p = 0; p < pieces.length; p++) {
  Set<Integer> domain = new HashSet<Integer>();

  for (int i = 0; i <= this.cols - pieces[p].w; i++)
    for (int j = 0; j <= this.rows - pieces[p].h; j++)
      domain.add((j * this.cols) + i);

  solver.addVariable(p, domain);
}
```

Further, to generate all of the sets of constraints, I looped through each possible pair of pieces, and added all of the pairs of values in their respective domains that did not overlap. This overlap test was implemented such that each piece got a set of Integer locations it occupies. If the intersection of these sets was the empty set, then this shows that the pieces do not overlap.

A partial implementation of this is as follows:

```
private boolean isConflict(CircuitPiece p1, int val1, CircuitPiece p2, int val2) {
  Set<Integer> p1Claimed = new HashSet<Integer>();
  Set<Integer> p2Claimed = new HashSet<Integer>();

  // Build the set of all places on circuitboard claimed by piece 1
  int xStart = val1 % this.cols;
  int yStart = val1 / this.cols; // Note the integer division

  for (int y = yStart; y < (yStart + p1.h); y++)
    for (int x = xStart; x < (xStart + p1.w); x++)
      p1Claimed.add(y * this.cols + x);

  // Build the set of all places on circuitboard claimed by piece 2
  xStart = val2 % this.cols;
  yStart = val2 / this.cols; // Note the integer division

  for (int y = yStart; y < (yStart + p2.h); y++)
    for (int x = xStart; x < (xStart + p2.w); x++)
      p2Claimed.add(y * this.cols + x);

  // If the intersection of these two sets is nonempty, there is a conflict
  p1Claimed.retainAll(p2Claimed);
  return (! p1Claimed.isEmpty());
}
```

In the example given in the problem, A and B would together have the constraint of

```
[(16, 0), (16, 1), (0, 3), (0, 4), (0, 5), (7, 10), (7, 11), (7, 12), (16, 10),
(16, 11), (0, 13), (2, 5), (0, 14), (0, 15), (11, 4), (11, 5), (2, 15), (6, 0),
(6, 1), (11, 14), (11, 15), (15, 0), (6, 10), (6, 11), (17, 0), (17, 1), (15, 10),
(17, 2), (1, 4), (1, 5), (10, 3), (10, 4), (10, 5), (17, 10), (17, 11), (17, 12),
(1, 14), (1, 15), (5, 0), (10, 13), (12, 5), (10, 14), (10, 15), (7, 0), (7, 1),
(5, 10), (7, 2), (12, 15)]
```

In this way, indexes into the one dimensional grid that corresponds to the way a two-dimensional grid would be layed out contiguously in memory are used to represent both domains, sets of occupied space on the board, and integer constraints.

The full implementation of this is in `Circuit.java`.