# Chess AI

Junjie Guan < *gjj@cs.dartmouth.edu* >

February 27, 2014

## Contents

**1 Introduction** **1**

**2 Solver design** **2**
2.1 Problem defination . . . 2
2.2 Design overview . . . 2

**3 CSP solver** **3**
3.1 back-tracking . . . 3
3.2 pickMRV . . . 4
3.3 sortLCV . . . 4
3.4 MAC3Inference . . . 5

**4 Problem definition** **6**
4.1 Variable . . . 6
4.2 Domain . . . 6
4.3 Constraints . . . 7

**5 Map coloring problem** **8**
5.1 Important methods . . . 8

**6 Circuit board layout problem** **8**

**7 CS 1 section assignment problem** **8**
7.0.1 minimaxIDS . . . 8
7.0.2 maxMinValue . . . 8
7.0.3 handleTerminal . . . 9
7.0.4 helper class . . . 9

**8 Results demonstration** **10**

**9 Some related work** **12**

## 1 Introduction

*'Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations.'*[1]

[1] Wikipedia: `http://en.wikipedia.org/wiki/Constraint_satisfaction_problem`

## 2 Solver design

### 2.1 Problem defination

CSP is usually defined with a tuple $< X, D, C >$, as following:

$$X = \{X_1, ..., X_n\}$$

$$D = \{D_1, ..., D_k\}$$

$$C = \{C_1, ..., C_m\}$$

where $X$, $D$ and $C$ is a set of variables, domains and constraints. Our goal is to assign each variable a non-empty domain value from $D$, while satisfying all the constraints in $C$.

### 2.2 Design overview

Figure 1 is an overview my codes design. Basically it can be devided into 3 parts.

- The drivers contains the main function that read the input dataset and present the results/solutions.
- The second part is problem defination, theasf the asdf. It contains the basic definition of CSP problem. For example, the Variable, Constraints represent the the set I mentioned above, with necessary methods inside. Such as building constraints, validating the assignment, etc. I implement most of the method in a generic class, while I also extend the basic class for the needs of different problem. Despite of this, solver do not care about the specific problem. It only deal with those lower generic classes.
- The third part the problem solver, where I implement the CSP algorihtm that solves the problem, as well as some helper methods such as heuristic compuation.

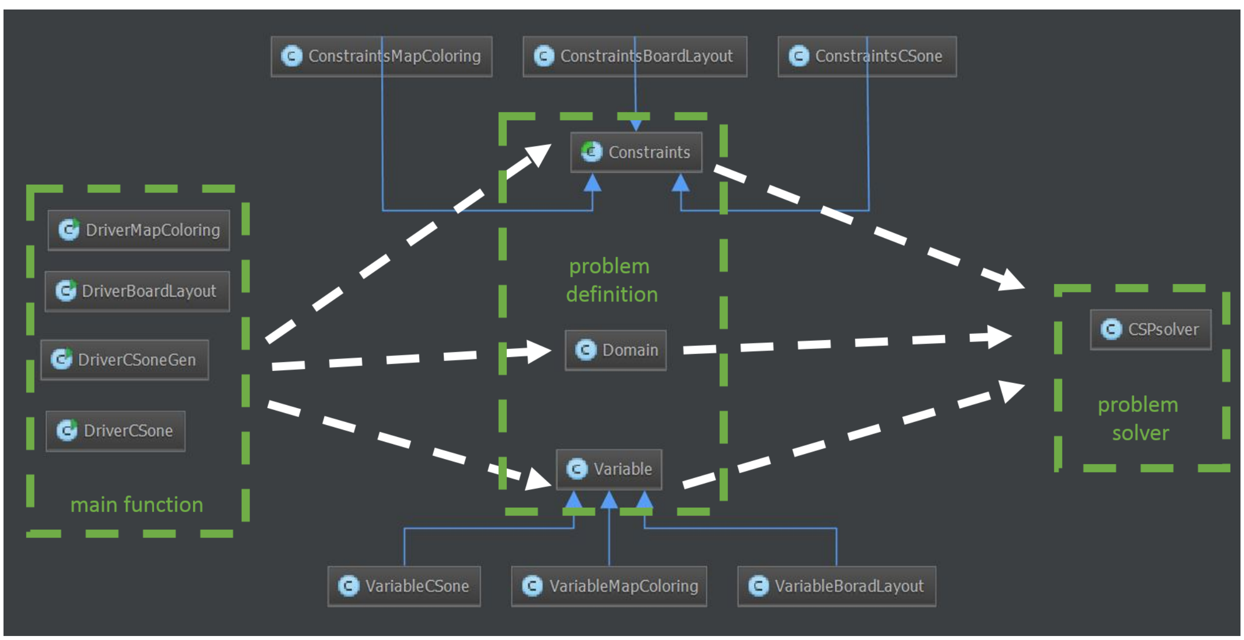



Figure 1: Overview of the design

# 3 CSP solver

## 3.1 back-tracking

**cspDFS** is a recursive style DFS method, which is the outline of the whole back-tracking. The basic procedure is,

1. Pick a *variable* from the unassigned variable set;
2. And then update all its valid *domain* values;
3. After preparation, it pick a *domain* value, and keep recursion of the remaining *variables*. Noted that if its successor report failure of DFS, it will iteratively choose the next *domain* values;
4. If it runs out of *domain* values, it will undo the assignemnt and report fail of search to its parent.

You may already notice some method names such as **pickMRV**, **sortLCV** and **MAC3Inference**. They are optimizations of the back-tracking algorithm, which will be explained in details later.

```
/**
 * @param remain: list of variables that is not assigned yet
 * @return: whether the search is successful or not
 */
protected boolean cspDFS(LinkedList<Variable> remain) {
  // base case of recursion
  if (remain.size() == 0) return true;
  // pick a Minimum Remaining Variable
  Variable var = pickMRV(remain);
  // update domain and sort the values based on Least Constraining
  sortLCV(var, remain);
  // iterate all the valid domain
  for (Domain domain : var.getDomains()) {
    var.assign(domain);
    // try to assign the next variable in the remain
    if (MAC3Inference(var, remain)) {
      if (cspDFS(remain)) {
        return true; // solution found!
      }
    }
  }
  // not found, reset assignment and put variable back to remain
  undoAssignment(var, remain);
  return false;
}
```

## 3.2 pickMRV

The body of **pickMRV** is fairly simple: it returns the variable with min heuristic based on MRV heuristic.

```
protected Variable pickMRV(LinkedList<Variable> remain) {
  Variable var = Collections.min(remain);
  remain.remove(var);
  return var;
}
```

The Main idea of MRV heuristic (minimum remaining values), is to find a variable with minimum remaining values, because these kind of value is most likely to cause a fail search. If two variable has the same amount of domain values, we pick the one with larger *degree*, because potentially it also more easy to fail.

```
@Override
public int compareTo(Variable o) {
  int compared = (int) Math.signum(domainSize() - o.domainSize());
  if(compared != 0)
    // return the one with minimum remaining values
    return compared;
  else
    // return the one with maximum degree
    return (int) Math.signum(o.getDegree() - getDegree());
}
```

## 3.3 sortLCV

**sortLCV** is bascially sort all the valid domain values based LCV heuristic(least constraint value). The key idea is to left domains for the remainning variables as much as possible, so that the search is more likely to succeed earlier.

```
protected void sortLCV(Variable var, LinkedList<Variable> remain) {
  // update the remaining domains based on constraints
  updateDomains(var);
  // compute value heuristic by trying
  for (Domain domain : var.getDomains()) {
    var.assign(domain);
    // try to assign the next variable in the remain
    domain.h = 0.;
    // compute its effect on the remaining varaibles
    for (Variable v : remain)
      domain.h += remainingDomains(v);
  }
  var.undoAssign();
  Collections.sort(var.getDomains());
}
```

## 3.4 MAC3Inference

The body of **MAC3Inference** prunes the domains and forecast failure by checking arc consistency recursively. Supposed current variable is $X_i$. Line 5 means getting all the arcs $< X_i, X_j >$, where $\{X_j\}$ is the neighbors of $X_i$. Line 13 means getting all the $< X_k, X_i >$, where $\{X_k\}$ is the neighbors of $X_i$ except $X_j$.

This recursion of inference is meant to converge / terminate at some point. Because it keeping pruning down the domains of the whole variables set, and stops recursion when no more domains can be pruned down. Since domain is a finite set, this inference is expected to termiante.

```
1  protected boolean MAC3Inference(Variable thisVar,LinkedList<Variable> remain){
2    // back up the properties of this variable
3    Variable var = thisVar.snapshot();
4    int backup = thisVar.getAssignment();
5    LinkedList<Constraints.ArcPair> arcs = cons.getAdjArcs(var, remain);
6
7    while (arcs != null && !arcs.isEmpty()) {
8      Constraints.ArcPair arc = arcs.removeFirst();
9      if (revise(arc.first, arc.second)) {
10       if (var.getDomains().isEmpty())
11         return false;
12       arcs.addAll(cons.getAdjArcsInvert(arc.first, arc.second, remain));
13     }
14   }
15   // recover properties of this variable
16   thisVar.assign(new Domain(backup));
17   return true;
18 }
```

The revise function prunes the domain when inconsistency occurs, and report the change of domains.

```
protected boolean revise(Variable var, Variable adj) {
  for (Iterator<Domain> it = var.getDomains().iterator(); it.hasNext(); ) {
    // try to assign a domain and test if conflict exists
    var.assign(it.next());
    if (!cons.consistentTest(var, adj)) {
      it.remove();
      var.undoAssign();
      return true;
    }
  }
  var.undoAssign();
  return false;
}
```

# 4 Problem definition

Here I am presenting the outlines of two major classes that define the problem.

## 4.1 Variable

Here you can see the outline of **Variable**. Since most of the methods are very trivial, I am not presenting the details here.

| Modifier and Type | Field and Description |
|---:|---|
| protected int | assignment |
| protected Constraints | cons |
| protected int | degree |
| protected java.util.LinkedList<Domain> | domains |
| protected java.util.LinkedList<Domain> | domainsBackup |
| protected int | id |

Table 1: Field Summary

| Modifier and Type | Field and Description |
|---:|---|
| protected int | assignment |
| boolean | assign(Domain domain) |
| int | compareTo(Variable o) |
| int | domainSize() |
| void | domainsRecover() |
| boolean | equals(java.lang.Object other) |
| int | getAssignment() |
| int | getDegree() |
| java.util.LinkedList<Domain> | getDomains() |
| int | getId() |
| java.util.ArrayList<java.lang.Integer> | getStates() |
| int | hashCode() |
| void | setDegree(int d) |
| void | setDomainsBackup() |
| Variable | snapshot() |
| java.lang.String | toString() |
| void | undoAssign() |

Table 2: Method Summary

## 4.2 Domain

Domains is very simple, basically just domain value along with a heuristic value.

## 4.3 Constraints

Here you can see the outline of **Variable**. There are two basic members in this class. **binaryAdjs** is used to store all the binary relationship, whole **binaryAdjs** is for global relationships.

I don't store the relationship such as equal, greater, etc. Assuming that all the arcs/sets have the same relationship. Of course you can extend members base on your need of problem. For example, if different binary constraints have different relationship, you should implement a hashmap from collection of variables to their relationship.

| Modifier and Type | Field and Description |
|---:|---|
| static java.util.HashMap<Variable,java.util.LinkedList<Variable>> | binaryAdjs |
| static java.util.HashMap<java.lang.Integer,java.util.HashSet<Variable>> | globalAdjs |

Table 3: Field Summary

| Modifier and Type | Field and Description |
|---:|---|
| void | addConstraint(Variable var1, Variable var2) |
| void | addGlobalVar(Variable var) |
| boolean | conflictTest(java.util.LinkedList<Variable> vars) |
| boolean | conflictTest(Variable var) |
| boolean | consistentTest(Variable var, Variable adj) |
| LinkedList<Constraints.ArcPair> | getAdjArcs(Variable var, LinkedList<Variable> remain) |
| LinkedList<Constraints.ArcPair> | getAdjArcsInvert(Variable var, Variable exclude, LinkedList<Variable> remain) |
| abstract boolean | isSatisfied(Variable var1, Variable var2) |
| void | rmGlobalVar(Variable var) |

Table 4: Method Summary

One major method is **conflictTest**. It is used to test conflict with the adjacent, after assigning values to a variable. The base method is written for binary constraint, because theoretically all problems can be represented by binary constraint problem. I will extend this for global constraints in some other problems, which makes the implementation and compuation more easy.

```java
/***
 * @param var: the variable to be tested
 * @return: whether the conflict exists
 */
public boolean conflictTest(Variable var) {
  LinkedList<Variable> adjs = binaryAdjs.get(var);
  if (adjs == null)
    return false; // no adjacent in constraint graph, no conflict
  for (Variable adj : adjs) {
    if (!isSatisfied(var, adj) )
      return true;
  }
  return false;
}
```

Since most of the methods are not very crucial, I am not presenting the details here. The most important one is the **isSatisfied** method. It related to specific problem, I will present those extended codes when discussing real problem application later.

# 5 Map coloring problem

Map coloring is a very classic constraint satisfied problem. The basic idea is, any adjacent areas in the map should have different color, which forms a binary contraint.

## 5.1 Important methods

**isSatisfied** is method to determine whether two variables satisfied the bianry constraint. For each assigment, there can be a list of states corresponding to this vairble. The **getStates** is such a genric method in the base class. As you can see for map coloring problem, there is only one states, which is the assigned color.

Depending on the needs of the problem, I will override the **getStates** in **VariableExtendClass**, and wrote corresponding **isSatisfied** in the **ConstraintsExtendClass**

```
@Override
public boolean isSatisfied(Variable var1, Variable var2){
    return var1.getStates().get(0) != var2.getStates().get(0);
}
```

## 5.2 Testing

```
public ArrayList<Integer> getStates() {
    return new ArrayList<>(Arrays.asList(getAssignment()));
}
```

# 6 Circuit board layout problem

# 7 CS 1 section assignment problem

### 7.0.1 minimaxIDS

**minimaxIDS** initializes the search using iterative-depending strategy.

```
private short minimaxIDS(Position position, int maxDepth)
    throws IllegalMoveException{
  this.terminalFound = false;
  MoveValuePair bestMove = new MoveValuePair();
  for (int d = 1; d <= maxDepth && !this.terminalFound; d++) {
    bestMove = maxMinValue(position, maxDepth - 1, MAX_TURN);
  }
  return bestMove.move;
}
```

### 7.0.2 maxMinValue

**maxMinValue** is an recursive funciton that keep searching in the tree. I write it in a compact way by mering min and max procedure in this one method, which turns out to be a big mistake for future when I try to implement more fancy mechansim for the searching. This design makes the codes a little messy.

There is also another better way to implement the search in a compact way, which is called *Nagamax*. However it was too late for me to discover it so I leave it to my future work.

```java
private MoveValuePair maxMinValue(Position position, int depth,
    boolean maxTurn) throws IllegalMoveException{
  if (depth <= 0 || position.isTerminal()) {
    // the base case of recursion
    return handleTerminal(position, maxTurn);
  } else {
    // get all the legal moves
    MoveValuePair bestMove = new MoveValuePair();
    for (short move : position.getAllMoves()) {
      // collect values from further moves by recursion
      position.doMove(move);
      MoveValuePair childMove = maxMinValue(position, depth - 1, !maxTurn);
      bestMove.updateMinMax(move, childMove.eval, maxTurn);
      position.undoMove();
    }
    return bestMove;
  }
}
```

### 7.0.3 handleTerminal

**handleTerminal** is used to terminate the searching by returning an evalution value, when either reaching the maximun depth or check mate or draw. Noted that getMaterial evalutes the weighted sum of the stones, while getDominant evaluates the distribution of the stones.

```java
private MoveValuePair handleTerminal(Position position, boolean maxTurn) {
  MoveValuePair finalMove = new MoveValuePair();
  if (position.isTerminal() && position.isMate()) {
    this.terminalFound = position.isTerminal();
    finalMove.eval = (maxTurn ? BE_MATED : MATE);
  } else if (position.isTerminal() && position.isStaleMate())
     finalMove.eval = 0;
  else {
    finalMove.eval = (int) ( (maxTurn ? 1 : -1) * (position.getMaterial()
       + position.getDomination()));
  }
  return finalMove;
}
```

### 7.0.4 helper class

**MoveValuePair** help me to store the move and corresponding evaluation value. Also it has a generalized method that help me to find the max value for maximum search, or vise versa.

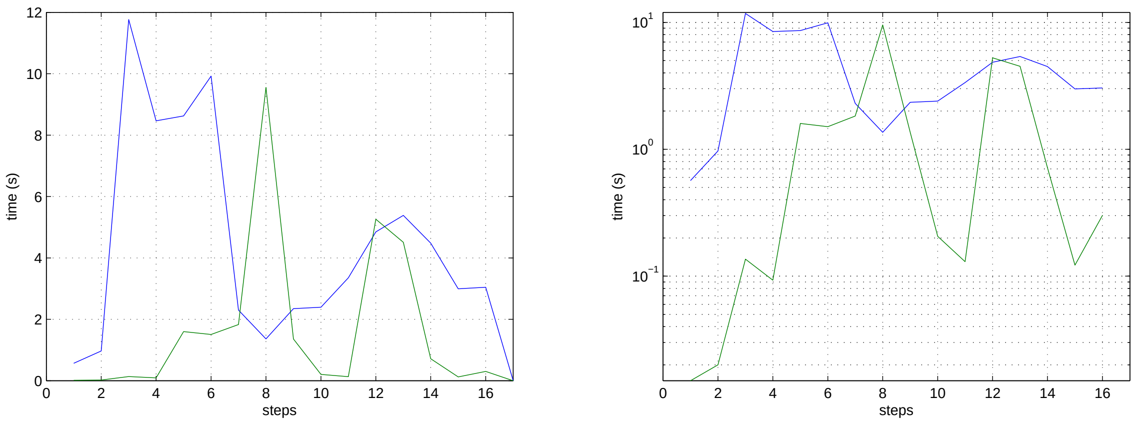
```java
private MoveValuePair handleTerminal(Position position, boolean maxTurn) {
  MoveValuePair finalMove = new MoveValuePair();
  if (position.isTerminal() && position.isMate()) {
    this.terminalFound = position.isTerminal();
    finalMove.eval = (maxTurn ? BE_MATED : MATE);
  } else if (position.isTerminal() && position.isStaleMate())
     finalMove.eval = 0;
  else {
    finalMove.eval = (maxTurn ? 1 : -1) * position.getMaterial();
```

```
  }
  return finalMove;
}
```

# 8 Results demonstration

I did a lot of testing, turn out that I don't leave myself much time to organize how to present them. Here I am going to focus on computation time of each step. I create a fix random seed for the random AI, and let my AI play with it.

Figure 2 demonstrate the step and time curve, with my Minimax against Random AI (blue curve), $\alpha\beta$ pruning against Random AI respectively (green curve). The left figure is a normal plot, while y axis of the right one is set to log scale (Noted that logy scale will shrink the difference on $y$ direction!). Considering sometimes the computation time grows exponentially with depth, this can provide a better observation. As you can see, the $\alpha\beta$ pruning finish the game using exact same amount of steps, while taking much less computation time.




(a) normal

(b) log scale

Figure 2: step and time curve of Random MinimaxAI and $\alpha\beta$ pruning.

Figure 3 demonstrate the step and time curve difference when there exists a tranposition table. As expected, the time conusmption is lower when implemented with transposition table. What's more, with transposition table it actually finish the game even fater! Because sometimes the table provider more depth of information than current node, which might lead to a bette decision.

Figure 4 demonstrate the step and time curve difference when there exists a tranposition table. Though the time decrease even more significantly, the takes more steps for some unknown reason. I also test ordering enhancement against pure transposition table, it shows that after reodering it becomes a little more stupid. This leave as my future work.

Figure 5 demonstrate mean time of different methods. It seems that null-move actually takes more time. I think it becuase although it reduce depth of search occassionally, it actually increase searching times on the same depth. May be I haven't tuned it properly.

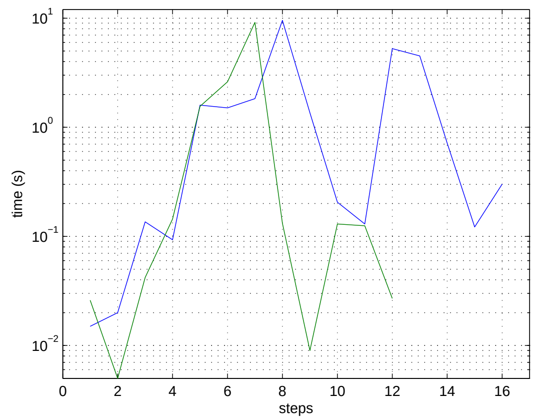



Figure 3: step and time curve of $\alpha\beta$ pruning and$\alpha\beta$ enhanced with transposition table

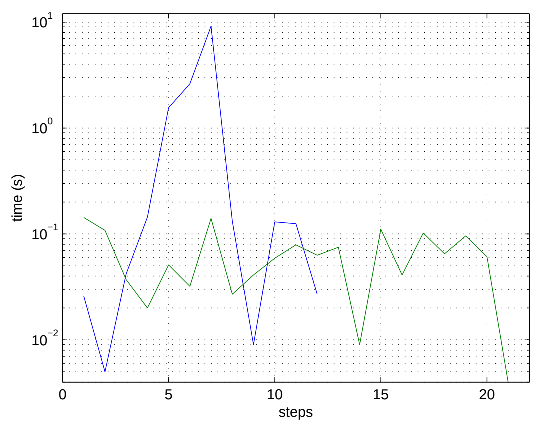



Figure 4: step and time curve of transposition table and$\alpha\beta$ enhanced with re-ordering

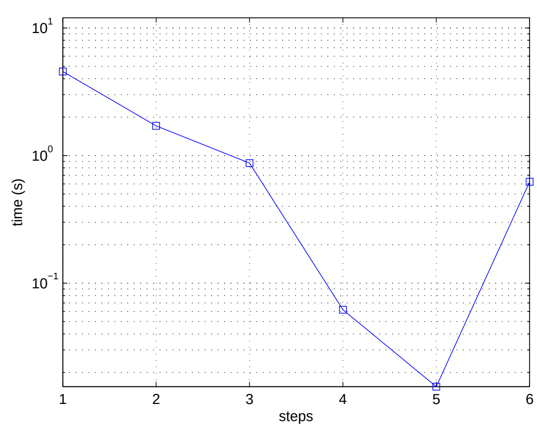



Figure 5: mean time regards to Minimax, Alpha-beta pruning, transposition table, moves reodering, quiescence search and null move heuristic repectively

# 9 Some related work

Null move strategy can be very tricky. Adaptive Null-Move Pruning [2] prose some good suggestions. 1) when depth is less or equal to 6, use $R = 2$. When Depth is larger than 8, use $R = 3$. When depth is 6 or 7, and both sides has more than 3 stones, then $R = 3$. Otherwise, R = 2.

[2]Heinz, Ernst A. "Adaptive null-move pruning." Scalable Search in Computer Chess. Vieweg+ Teubner Verlag, 2000. 29-40.