Chess AI

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

February 27, 2014

Contents

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 Overview

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 Layout

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.

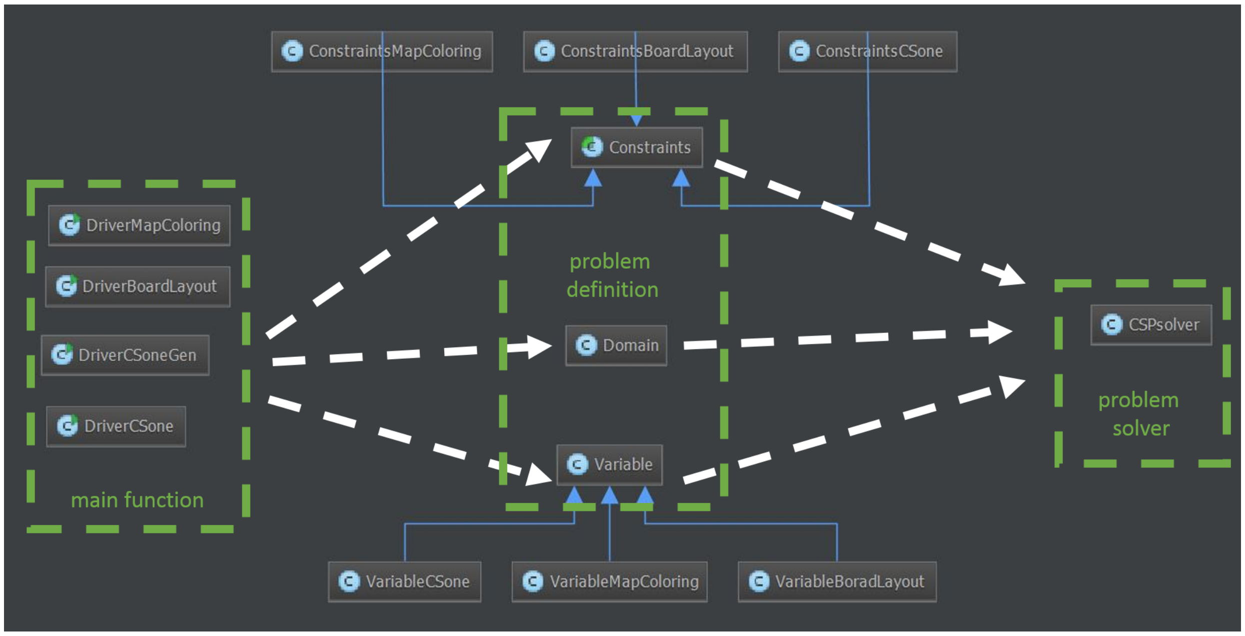


Figure 1: Overview of the design

2.2.1 minimaxIDS

minimaxIDS initializes the search using iterative-depending strategy.

```java
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;
}
```

2.2.2 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;
  }
}
```

2.2.3 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;
}
```

2.2.4 helper class

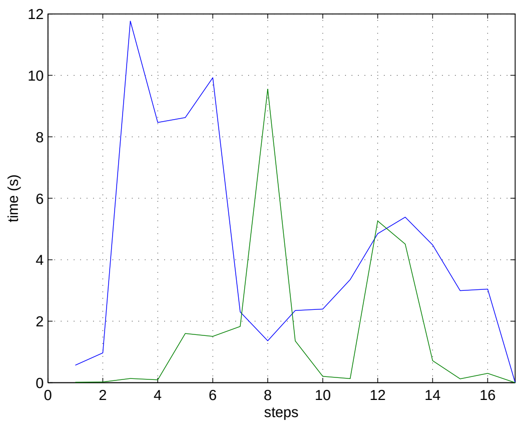
MoveValuePair help me to store the move and corresponding evaluation value. Also it has a generalized method that help me to find the max value for maximum search, or vise versa.

```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;
}
```

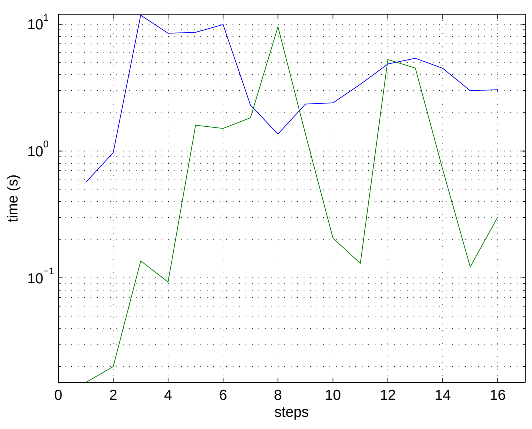
3 Results demonstration

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.

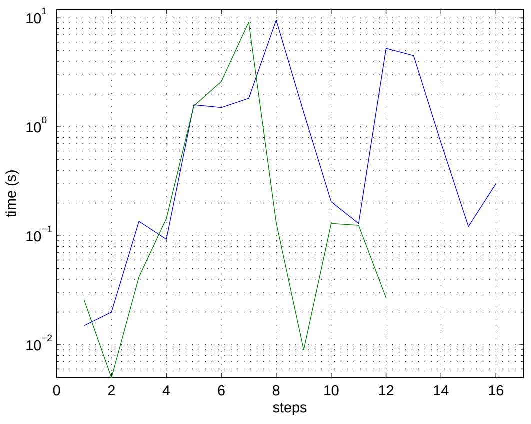


Figure 3: step and time curve of $\alpha\beta$ pruning and$\alpha\beta$ enhanced with transposition table

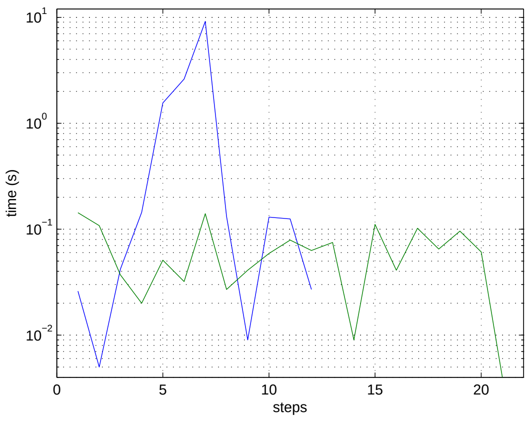


Figure 4: step and time curve of transposition table and$\alpha\beta$ enhanced with re-ordering

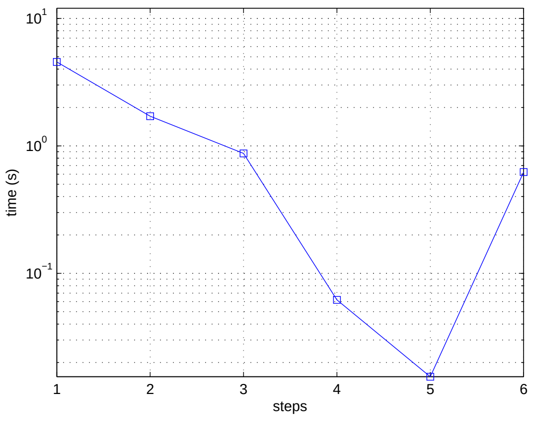


Figure 5: mean time regards to Minimax, Alpha-beta pruning, transposition table, moves reodering, quiescence search and null move heuristic repectively

4 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.