# CS542 HW2

**1.** Explain how you solved the problem using simulated annealing. Which temperature schedule did you use and why? How did you pick the initial assignment?

In simulated annealing, I basically designed three parts:

1) Initialization function
   Initialization function takes a Sudoku puzzle as an input, and generates a Sudoku matrix with no blank slot inside. The way that initialization function initializes matrix is a partial random process. For one after another, the function assigns values for each small block in a way that keeping block constrain held. Therefore, after the whole process finished, every conflict comes only from either row constrain or column constrain.
2) Successor function
   Successor function generates a next state from a current state. Since we want to hold the block constrain all the time, we simply choose a block randomly and swap two randomly choose values (not fixed value). In this sense, only row and column situations are changed.
3) Temperature function
   I did not use any of the simulated annealing function given by the ppt, but designed my own function.
   The function is:

   *T = 0.05 * (t - 1) / (MaxIterateTimes - 1) + 0.3*

   This function is in a slightly increasing order. At the beginning, T equals to 0.3, then it goes up slowly to 0.35.
   From observation, at the final phase of running, when delta E is around +/- 1, I am expecting the possibility should be no more than 5%, resulting in the value of T should be below 0.35.
   The reason I made it an increasing function is because I think it is pointless to hang around at the top of the “mountain”. I am hoping to drop down to somewhere near the “bottom” and hang around near “bottom”. I also designed a visualization software to depict the curve of the conflicts change. (See picture in the last page)

**2.** Explain how you solved the problem with A*, including which heuristic function you used.

In A*, I basically designed one part:

1) The way that fn is calculated
   According to the A* algorithm, fn = gn + hn.
   - In my design, gn stands for the number already assigned slots from the total number of possible slots that related to the current slot (means the slots which is in the same row/column/block as the current slot). In some sense, gn stands for the "cost" from starting state to current state. As time goes, gn tend to be larger and larger (because more and more slots are assigned). It makes sense because the current state gets further and further from beginning state, so "cost" should be getting larger.
   - hn is stands for the number of possible values. Returning values in the range of all values (for example 1~9 in 9*9 puzzle) keeps the heuristic function from overestimating the cost.
2) Each step we pick up the the slot with the lowest fn value.
   Intuitively we pick up the slot that is mostly likely to be correct, which is similar to human logic.

**3.** Compare the performance of Simulated Annealing, A*, and CSP in terms of number of seconds to solve a board (average across 5 puzzles).

We use five different 9*9 puzzles with different difficulties.
Table below is the calculating results: (in secs)

| | esay | esay | esay | medium | medium |
|---|---|---|---|---|---|
| Simulated Annealing | 1 | 1 | 9 | 6 | 5 |
| A* | 39 | 116 | 24 | 186 | 10 |
| CSP | 0 | 0 | 0 | 5 | 0 |

Simulated Annealing average: 4.4 secs
A* average: 75 secs
CSP average: 1 secs

**4.** Analyze A* and CSP's performance to determine why CSP is a better solution. What is the source of CSP's power here? Determine how much of the savings comes from each of the components within CSP.

In CSP, there are three heuristic involved:

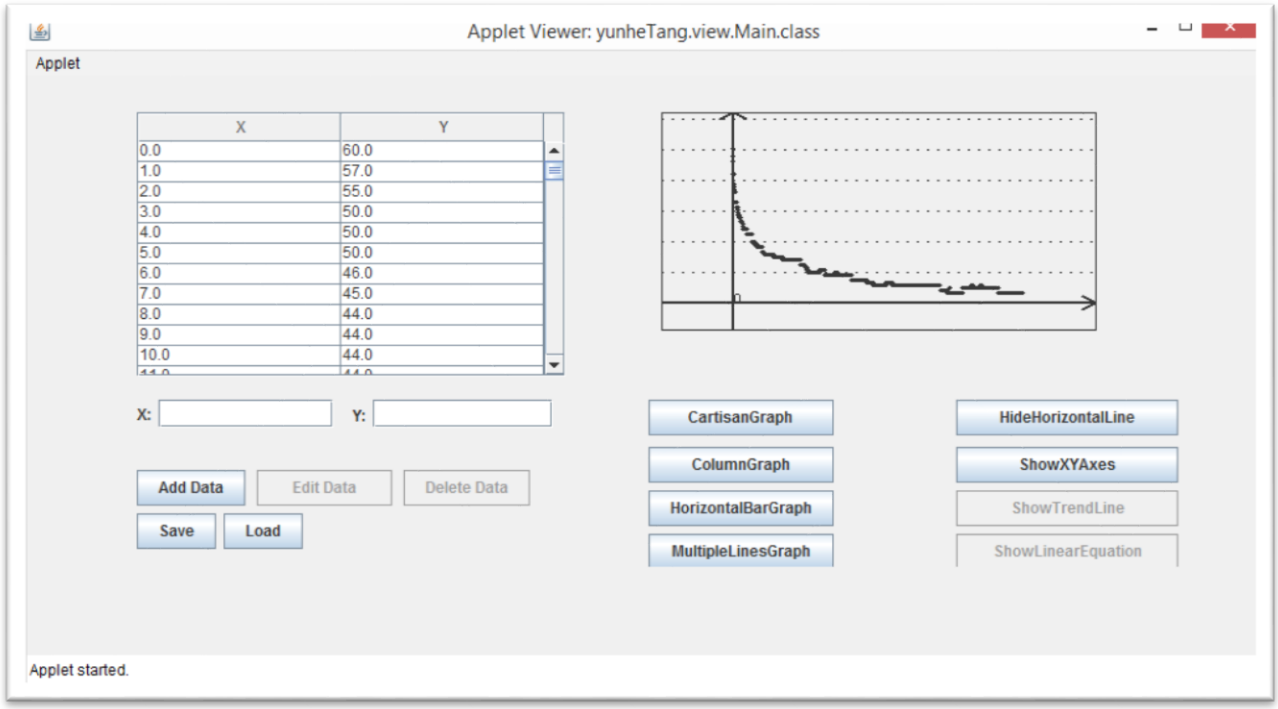
1) MRV
2) Degree Heuristic
3) Least-constraining-value heuristic

The first two heuristics is for choosing a variable to assign, and the last one is for deciding a value to assign.

The CSP is much faster than A* is because of two reasons:

1) Better way of picking up slot
   In each step in A*, we pick up the variable with the lowest fn value. However, since fn is calculated from gn and hn. And gn works as an indicator of guiding the A* to different areas of puzzle based on potential impact.
   CSP simply pick up the variable with the least set of potential value, which does not care about the position of the variable. The smaller the length of potential set is, the greater chance that CSP will be able to correctly guess the value.
   From picking up the smallest remaining value, CSP return the failure faster than usual, therefore avoiding spending to time on search useless space.

2) Choose a value to assign according to heuristic, rather than picking randomly.
   After pick up a variable, we can calculate the available value set.
   Here A* simply pick up a value from available value set randomly or withdraw from queue one by one.
   CSP, instead, takes advantage of Least-constraining-value heuristic to pick up value with priority. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. It proves that it is a effective way of choosing value in Sudoku problem.


Applet Viewer: yunheTang.view.Main.class
Applet
X
Y
0.0 60.0
1.0 57.0
2.0 55.0
3.0 50.0
4.0 50.0
5.0 50.0
6.0 46.0
7.0 45.0
8.0 44.0
9.0 44.0
10.0 44.0
X:
Y:
Add Data
Edit Data
Delete Data
Save
Load
CartisanGraph
ColumnGraph
HorizontalBarGraph
MultipleLinesGraph
HideHorizontalLine
ShowXYAxes
ShowTrendLine
ShowLinearEquation
Applet started.