Ba:
Assume column is i or j.
Rook: row(a_i) != row(a_j)
Bishop: abs(row(a_i) - row(a_j)) != abs(i - j)
Knight: row(a_i) – row(a_j) != 2 && i – j != -1
row(a_i) – row(a_j) != 1 && i – j != -2
row(a_i) – row(a_j) != -1 && i – j != -2
row(a_i) – row(a_j) != -2 && i – j != -1
Combined they form the CSP constraints for the amazon piece
Only need four conditions for knight, if both (a_i) and (a_j) are checked in in swapped order as well

Bb:
see attached program

Bc:

The problem is solved with a min-conflict approach. The board is set diagonally as an initial condition. Then each tile has it amount of possible conflicts calculated. One of the pieces standing on a tile with conflict greater then zero then be moved to the tile with the least amount of conflict in that column. The piece to be moved is chosen randomly from a list of ”conflicted” pieces.
This is repeated for size/5 iterations, where size is the size of the board. Then the (size/100 +1) pieces that has not been moved for the longest time will be randomly placed within their columns, to prevent getting stuck in a local minima.

This is will find a solution if there is one, but it can take a lot of time as the two random elements can work against each other.

Bd:
Solving boards with a width of under 100 is quick and takes less then a second. Increasing board size from 100 and upwards increases the time spent to solve the board considerably. Approximately, doubling the size of the board increases the time spent to find a solution by a factor of 25-30, meaning it grows with n^4 or n^5. This is probably not true though, as it probably gets worse the bigger the problem is due to the random nature of the solver. A more structured approach should have yield better results. One major change that should improve performance noticeably is if instead of throwing around some pieces at random every set number of iterations, a smarter way to detect if the solver is stuck at a local minima might be used. Such as a checksum comparison of the states between moves. Another way to improve performance might be to move the pieces in a not completely random fashion, for example, trying to move pieces with high conflict values first and making sure the same piece is not chosen two moves in a row.

size	100	200	400	**800**
time(mean), s	1,09	23,78	868,5	**26055**

Numbers in black are averages of 50 runs, 25 runs and 5 runs respectively. The figure for size 800 is an estimate based on the assumption that the time needed to solve the problem is increased by 30 every time size is doubled. It should be noted that at 400 steps, 2 of the solutions took less than 5 minutes, and 2 of them close to 25min. This shows the random behavior and really highlights a weak point of this solution. The numbers are from running the program on a dual core 1,6GHz intel Atom and linux 32-bit. Compiled with -O3 and gcc 4.4.1

Performance gains for a particular board size is also possible with some tweaking to the parameters that govern the random movements of pieces that has been sitting still too long.

Another approach might be to keep the random choice of what piece to move, and reset the board to a diagonal state after a certain amount of iterations have passed. This would need some more tests to figure out a reasonable value relating to the board size.

With a non-random way of choosing what piece to move next, it is easy to see while watching the program solve smaller boards stepwise that the last steps are the ones taking the most time to complete. This is due to the systematic way the pieces are picked, a behavior that easily gets them stuck in a local minima. Picking pieces systematically in the beginning and random pieces when the conflict values are lower might be a way around this,