Artificial Intelligence - Knowledge Representation and Planning - Assignment 1

Lorenzo Soligo, 875566 – Project made with Mauro Noris and Emanuele Motto

Academic Year 2018-2019

Contents

1 Requirements

The n-queen problem is that of finding a disposition of n queens on a nxn chess board such that no two queens threaten one-another, i.e., there is never more than one queen for every row, column and diagonal on the board.

The goal of the assignment is create solvers for the n-queen problem for any n.

You should provide solves using three different methods:

- Constraint Propagation and Backtracking
- Local optimization (hill climbing)
- Global optimization (simulated annealing or genetic algorithms)

Turn in the code and a report describing problem and approaches, the representation/modeling choices you have made, and a comparison of the behavior, pros and cons of the various approaches.

Extra credit: if the dimension n can be factor into the product of l and m, then with high probability you can construct a solution for the n-queen problem starting from the solutions for the l- and m-queen problems by taking the Kronecker product of the matrices representing the solutions (where 1 represents a queen and 0 and empty aquare). Even if the result is not a solution, it is in general a very good initialization for local search approaches. Reduce the complexity of the algorithms through this factorization approach and analyze and comment on the improvement.

2 Introduction

2.1 Constraint Satisfaction Problems (CSP)

2.1.1 Definition

A constraint propagation problem consists of three components: X, D, and C [1]:

- X is a set of variables $\{X_1, X_2, ..., X_n\}$
- D is a set of domains, $\{D_1, D_2, ...D_n\}$, one for each variable
- C is a set of constraints that specify allowable combinations of values

Each domain D_i consists of a set of allowable values $\{v_1, ..., v_k\}$ for variable X_i.

2.1.2 N-queens as a CSP

In our case, the **n-queens** problem can be formalized as a CPB, having:

- X = cells of a chessboard of size $n \times n$, or simply n variables containing the position on the i-th column of the queen
- D = set of values that each cell can take. A cell can either be *empty* ($value = 0$) or contain a *queen* ($value = 1$).
- C = set of constraints. There must be n queens on a $n \times n$ board and they must not attack each other, i.e. if $Board(i, j)$ *is a queen*:
  - each column must contain exactly one queen
  - $\forall k \in \{1, ..., n\},\ k \neq i\ \rightarrow Board(k, j)$ *must be empty*
  - each row must contain exactly one queen
  - $\forall k \in \{1, ..., n\},\ k \neq j\ \rightarrow Board(i, k)$ *must be empty*
  - each diagonal (also "smaller" ones) must contain exactly one queen
  - $\forall (k, m),\ k \neq i \wedge m \neq j,\ if\ abs(k - i) == abs(m - j)\ then\ Board(k, m)$ *is not a queen*

The problem can be approached in many different ways. Of course, the key point here is that in the huge space of all the possible configurations of a board, only few are actual solutions to our problem. In other words, our main need is to find a smart way to place/move queens. As a matter of fact, given an $n \times n$ board, we have n^2 cells in which to put the first queen, $n^2 - 1$ cells in which to put the second one and so on until the n-th queen, for whom we have $n^2 - n + 1$ cells available. You can see where this is going: bruteforcing is not an option.

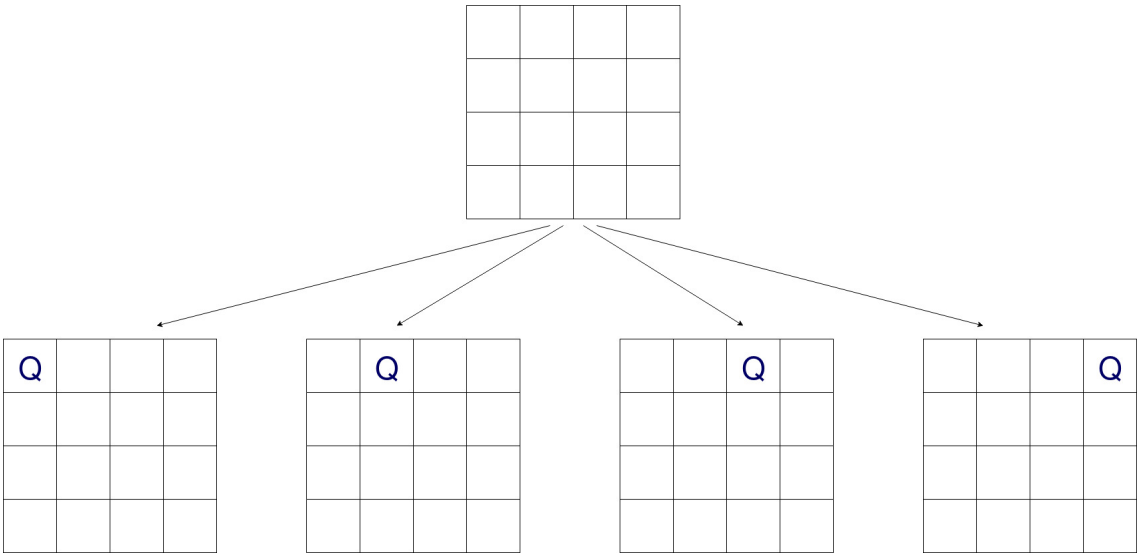

Figure 1: Part of the tree of all the possible configurations of a board

Some intuitive and some less-intuitive solutions to the problem come to mind. Firstly, one might begin with an empty board and add queens one after another, "backtracking" and re-adding queens in different positions until a solution is found. This approach will be shown in the **Constraint Propagation and Backtracking** chapter. This is indeed quite a natural way to approach the problem, even though no person would literally backtrack. One would usually move some queens around trying to improve the situation, moving only queens that reduce the total number of constraints. This approach is *greedy*, and represents the core idea behind Local Search.

The Local Search approach consists in placing the n queens randomly and then moving them following a *greedy* reasoning, until a solution is found. It is worth noting that this approach might lead to a situation in which no moves would improve the board's state, but the problem has not been solved yet: in other words, in a problem such as the one considered, we will get to a local minimum, but it might not be a global one (i.e. an actual solution). The **Local Optimization** chapter will provide further details on this topic.

In order to find an actual solution, hill climbing (the approach used by Local Search) is often not enough, but other search approaches not relying on backtracking will do the job. The two proposed are *simulated annealing* -which allows some randomness in hill-climbing to get out of local optima that are not global ones- and genetic algorithms, that, again, rely on randomness (i.e. crossover and mutation) to get closer and closer to a solution. Simulated annealing will be presented in the **Global Optimization** chapter.

3 Designing the solution

3.1 Programming language and paradigm

Our code is built in a OOP fashion. No particular design patterns have been applied, but having structured the code in different classes has greatly helped us in extending and fixing the code, and also made us focus on the purpose of the particular thing we were working on in a certain moment. Also, we chose to use Python since it makes writing code surprisingly easy and fast for us, letting us focus more on the actual logic than on technical details. In particular, instruments like list comprehension and a complete and well documented standard library have been of great help. The main downside lies in its performance, greatly inferior to the one of the same program written in (also JIT-)compiled languages such as C++ and Java.

3.2 Classes involved

We wrote:

- a class per type of solver (`GlobalSearchSolver`, `LocalSearchSolver`, `ConstraintPropagationSolver`)
- an underlying `Board` class, used to actually represent the queens on the chessboard
- a `Factorizer` class, which is used to factorize a number and return the factorized number in a suitable representation
- a class to implement the solution using the `Kronecker` product, which relies on the `Factorizer` and on some simple methods from the SciPy library
- a `Runner` and a `Benchmark` class, to run and take some measures of the code

The three Solvers will be explained in the following chapters; here I'll focus on the other classes.

3.2.1 Board

The `Board` class is definitely the one "auxiliary" class on which we spent more time, and on which I'll spend more words.

3.2.1.1 First approach At first, our `Board` contained a $n \times n$ matrix (from now on, a matrix will be a list of lists), in which every element was a `Cell`. A `Cell` object was an object with four properties: its `x` coordinate, its `y` coordinate, a set of `constraints`, a `queen` flag.

While this greatly helped us by giving us a very intuitive abstraction on the board, the overhead was way too significant to keep the board that way. Too many operations required $\mathcal{O}(n^2)$ and lots of memory accesses, being n the length (i.e. height or width) of the board (also, the number of queens).

3.2.1.2 Final implementation After having implemented the Constraint Propagation algorithm, we decided to take a step back and re-think the implementation of the `Board` class before going further. While this might be seen as a waste of time -and, in a certain sense, it was- I am really glad we proceeded this way: the first implementation let us write code quickly using an intuitive representation, providing more focus on the thinking process and getting a working solution for Constraint Propagation in a reasonable time. If we hadn't started with that inefficient class, I am not sure we would have asked ourselves the many questions we actually asked.

Of course, having understood the problem better, we were now capable of optimizing the class. In particular, a $n \times n$ board containing 0's and 1's (with 0 being an empty cell and 1 being a queen) can be represented as a list having length n, in which the i-th value represents the row in which the queen is put in column i. For example, the list `queens = [1, 2, 0]` resembles this matrix

```
0 0 1
1 0 0
0 1 0
```

Using the notation (column, row), the above list implies that we have queens in (0, 1), (1, 2) and (3, 0).

This simple optimization actually carries lots of implications with it: with respect to the previous implementation, we lost the `constraints` set of each cell, and browsing the matrix is now harder.

The first thing we had to do was, then, to find a way to keep track of the constraints. We decided to do this by using a list containing n dictionaries. Each of these dictionaries, resembling a column, contains n keys $(0, ..., n-1)$, and the value associated with each key represents the number of constraints (i.e. attacks received) of that cell, regardless of *who* is attacking. To make things easier, in this implementation a queen attacks herself, i.e. in the case of a solutions we have that for each queen, she has only one queen attacking her, which is herself.

Even though the implementation is now way faster than before, I still think some improvements are possible, since in the worst case (i.e. all queens are attacked by all the others) we have n^2 lists of length $\mathcal{O}(n)$, for a total used memory of $\mathcal{O}(n^3)$. As a matter of fact, when running any algorithm on a big board (say 3000×3000), I think that too much time is taken to initialize the constraints on all the board.

This implementation also allows us to "easily" propagate/remove the constraints when we add/remove one queen from the board in $\mathcal{O}(n)$ instead of the previous, unoptimized $\mathcal{O}(n^2)$.

3.2.2 Factorizer

The highlight of the `Factorizer` class is the `factorize` method. This returns a dictionary containing as keys the factors of the number, and as values how many times they appear in the number. Notice that we only look for factors ≥ 4 because we don't have any solutions for the 2×2 and 3×3 problems, and the solution to the 1×1 problem would be pretty useless. As an example, the number 80 will be factorized into $4 * 4 * 5$, so the `factorize` method will return `{4:2, 5:1}`.

3.2.3 Kronecker

This class takes as input the size n of the board and an optional parameter. The `solve` method uses the `Factorizer` to factor n and then solves the "subproblems" using either Constraint Propagation (by default) or Global Search (if the user calls `Kronecker`'s constructor with an additional parameter `'GS'`). After having solved the subproblems, the class performs the Kronecker product between the solutions and returns the resulting board.

Notice that the obtained board isn't necessarily a solution: in the case it isn't, we can use it as a starting point for a Global Search (or Local, if we accept a local minimum).

3.2.4 Benchmark

Class used by `Runner` to run the different algorithms and take some basic measures.

3.2.5 Runner

This class simply runs the program.

4 Constraint Propagation and Backtracking

4.1 Background

The key idea in constraint propagation is (surprise, surprise) *propagating* constraints in order to not having to search the whole, huge search space of a problem.

Let us assume we have a $n \times n$ board. If we place a queen in cell (i, j), then:

- all the cells in row i except for cell (i, j) cannot have a queen
- all the cells in column j except for cell (i, j) cannot have a queen
- all the cells in a diagonal with respect to cell (i, j) cannot have a queen

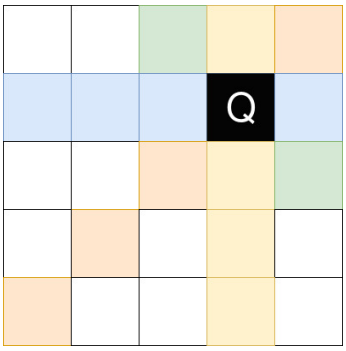

Figure 2: Propagated constraints from a queen placed in a 5x5 board

This heavily reduces the possible steps to be made after the current one. The key idea of constraint propagation is indeed to try and remove all the configurations that could never result in an actual solution, if the current configuration was part of it. In other words: let's assume that the queen I already placed is part of the solution. If it is, then having a queen in the constrained cells is not possible, and we can avoid searching that "subspace".

4.2 Example

Let's consider a 5×5 matrix as an example, and let's put a queen in cell $(0, 0)$. All the assignment such as this one are referred to as *partial* assignments: we are setting some variable to a "tentative" value [3].

Then, the matrix is the following:

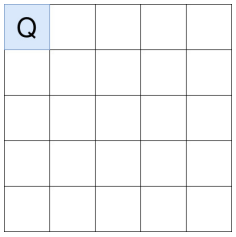

Figure 3: First queen placed

But as we said, we cannot have two queens attacking themselves. Therefore, if we *propagate* the constraints generated by the queen in (0, 0), we get to this situation:

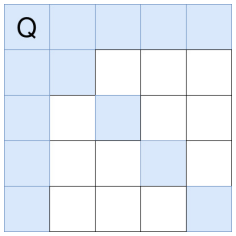

Figure 4: Constraints of the first queen are propagated

Hence the following queen can only be (ideally) placed in one of 12 cells, which is much less than the previous 24 cells! In our case, since we move by column (from left to right), 3 assignments are possible for the next queen instead of 5. We have no additional knowledge, so we need to pick (randomly, in the simplest case) one of the 3 cells and place the queen there. Of course, proceeding in this way will not always lead to a solution. For example, in two moves we might find ourselves in the following situation:

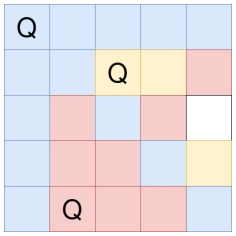

Figure 5: What to do now?

We have placed the first three queens, but we have no legit places in which to put the fourth one.

What to do, then? We need to **backtrack** and try different solutions. In other words, one of our *guesses* (for instance, the row in which we put the second queen) was "wrong", and we need to change it. Pure backtracking goes back of only one move (column) and tries different configurations. If we try all the possible positions in that column but they all lead to a situation in which we can't solve the problem, we need to go one more step (i.e. column) back, choose another placement for the queen and reiterate the process. If we are really unlucky, we might even try all the possible configurations of the board (i.e. search the whole space) and never find a solution! But luckily, this case only happens when $n \in 2, 3$. In all the other cases, at least a solution exists. Too bad it might take way too long to discover it.

Pure backtracking is a quite simple algorithm, but it is not really efficient in terms of time and space complexity. However, constraint propagation results in some kind of *pruning* that heavily reduces the states yet to be analyzed at a certain point, as per the aforementioned example. Notice, however, that especially in the cases of bigger boards, misplacing the "first" queens might result in a huge delay to get the solution, because of the great number of backtracking iterations.

Finally, it is worth saying that another approach could be to implement backjumping, i.e. going back to the latest column that attacks the current queen instead of going back of only one column [1].

4.3 Implementation of the solver

The solver is implemented in the `ConstraintPropagationSolver` class. Every solver keeps a `board`, its size `n` and a list of lists called `backtracked_queens`, which contains, for each column, the rows in which it has tried to put a queen.

4.3.1 Code

```
def solve(self):
    # place a random queen on the first column and go on to the next one
    first_queen = randint(0, self.n-1)
    self.board.add_queen(first_queen)
    col = 1
    # keep going until a solution is found
    while not self.board.is_a_solution():
        # count the number of constrainted cells on the current column
        col_constraints = 0
        for i in range(self.n):
            if(self.board.constraints[col][i]) != 0:
                col_constraints += 1
        # if we cannot try some other move on the current column, backtrack
        if col_constraints + len(self.backtracked_queens[col]) >= self.n:
            r = self.board.get_queen(col-1)
            self.board.remove_queen_constraints(r)
            self.backtracked_queens[col-1].append(r)
            self.backtracked_queens[col] = []
            col -= 1
            self.board.remove_last_queen()
        # if there is some move available on the current column, try it
        else:
            moves = [k for k, v in self.board.constraints[col].items() if v
                == 0 and k not in self.backtracked_queens[col]]
            new_queen = choice(moves)
            self.board.add_queen(new_queen)
            col += 1
    return self.board
```

4.3.2 Explaination

Intuitively, the `solve` method adds a queen to the first column and propagates its constraints.

After this, it loops until a solution to the problem is found. It computes how many constraints the next column is subject to. In other words, it computes the number of non-available rows to put the queen on. If this number + the number of positions in which we have *already* tried to put the queen is (at least) n, then we have no other moves available for this column, and we should backtrack. Therefore, we add the current row to the rows that we tried in the current column and led to a failure, and reset the rows we tried for the next column (since we are changing queen, hence we are in a different configuration). We go back to the previous column and try another position for the queen (if available), or we go another column back until it is possible to try something new.

If, on the other hand, the current column has some not-yet "explored" cells available, we try putting the queen in one of them (choosing randomly) and go on: we might be reaching a solution!

Of course, every time we remove or add a queen, we add/remove constraints to the rest accordingly. Finally, when we can choose between more than one move, we choose a random one in order to try avoiding always choosing the "top-most" move.

The whole algorithm can be summarized as follows:

```
*until the current state is not a solution*
1. add a queen to the leftmost column which does not have a queen
2. propagate the constraints created by the latest queen added to the board
3. if the next column has some "free slots", try putting a queen there and
    go to 2
4. otherwise remove the last queen (and the constraints she had propagated)
     and
   a. if there are other slots that have not been tried yet on the same
       column, try putting the queen there and go to 2
   b. if there are no slots available, go to 4
```

4.4 Performance analysis

4.4.1 Results

The test here reported consisted in 10 runs of Constraint Propagation and Backtracking on a 77×77 matrix. Here are the results.

Time:

- Total time required: 287.9 seconds
- Average time required: 28.8 seconds
- Time taken by the worst iteration: 244.7 seconds
- Standard deviation (corrected) of the results: 76.51 seconds
- Average time required not considering the worst iteration: 4.80 seconds

Steps:

- Total steps: 3784140
- Average steps required: 378414.7
- Steps taken by the worst iteration: 3264101
- Standard deviation (corrected) of the steps 1096394.7 steps
- Average steps required not considering the worst iteration: 57782.111 steps

4.4.2 Considerations

The results highlight that the algorithm's efficiency is quite erradic: in the worst case, it took ~8 times the average number of steps (and seconds), and this behavior tends to get worse and worse increasing the size of the matrix.

Also, we can see that the number of steps greatly varies: the average is ~378.000, but in the worst iteration more than 3 million steps were made!

From the tests made, the average number of steps taken has always been smaller than n^4, which seems like a decent result, even though the exponent is still quite big. We'll see if and how Local/Global optimization will lower it.

All in all, considering that Constraint Propagation and Backtracking always return an actual solution (except for the trivial cases $n \in \{2, 3\}$, for which a solution does not exist), I think this algorithm is pretty solid, especially when running it many times. On an average, it works decently even if the code is written in Python (and by not-so-awesome developers), but for large matrices other methods should be used, for this is too slow when computing the solutions of "big" matrices. Another thing one might do is to use Constraint Propagation and Backtracking to solve "smaller"

matrices and then combine the results with the Kronecker product, finally applying a local/global search to the obtained matrix.

5 Local Optimization

5.1 Hill climbing: definition

We define *hill climbing* as the following algorithm [1]:

```
function HILL-CLIMBING (problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
  current = neighbor
```

Hill climbing is a local search algorithm which aims at improving the current configuration following a greedy strategy. If there is a move that would reduce the total number of constraints, we do it; otherwise, we stop. Hill Climbing, conversely to Constraint Propagation, doesn't add queens incrementally, as it begins with an arbitrary configuration (an arbitrary board, in our case).

Notice that the only case in which a local minimum (maximum) is surely a global one is when the considered problem is convex (concave). This is not the case, therefore it is possible to get stuck in local minima.

5.2 Hill climbing for the N-Queens

5.2.1 Intuition and design

In the case of the N-queens problem, it is reasonable to work with configurations having exactly one queen per column (row) and move the queen along the same column (row), in a different row (column) from the current one. This is also very well suited for our implementation, since our `queens` list can be seen as a dictionary where the keys are the columns and the values are the rows (i.e. `queens[1] = 2` means that the queen in column 1 is in row 2), and therefore we cannot have two different queens in the same column. Of course, hill climbing could also be done by simply moving a queen around, but we decided to go with the swapping-rows solution because from our tests it works better than the standard one with respect to the percentage of actual results, with little to no differences on the time taken to solve the problem.

Swapping two rows instead of simply moving one queen seems to be a smart move because the space of the solutions to the problem is only made of boards that have one queen per row and one queen per column. Of course they also need to also satisfy the constraints along the diagonals, but at least we are enforcing the satisfaction of the vertical and horizontal constraints, and we are getting it almost for free. In other words, if we started with a configuration with one queen per row/column and only moved one queen, the following configuration would surely not be a part of the space of solutions, because we would either have two queens in the same row or in the same column; hence, what we thought about the local search as a swap of two rows, so that at least the one queen per row/column property is satisfied.

This swap is done by "forward-checking" that the configuration obtained by this swap will have a strictly lower number of total constraints on the board. In other words: we know that we need to have one queen per row and one per column, therefore instead of moving one queen (which would result in 2 queens in the same row/column), we swap two rows on the chessboard, enforcing

that the following configuration will be better (i.e. have less attacks) than the current one. This guarantees an improvement in the solution because of the forward-checking (which is not a true forward-checking, anyway), and also guarantees that the current state is still a part of the space of the solutions of the problem.

The queen to be moved is chosen in such a way that the total number of queens attacking each other will decrease at each step. If we cannot improve the current situation, we are stuck in a local minimum and the algorithm terminates, returning the current board. From our tests, this happens quite frequently with small boards (for example the 6×6 one, which only has 4 solutions), while the situation gets better with bigger boards, probably because of the huge number of possible solutions.

5.2.2 Considerations

5.2.2.1 Standard algorithm The `HILL-CLIMBING` algorithm could definitely be improved. Firstly, it does not allow "sideways" moves, i.e. moves in which we go from a state with a total of k constraints to another state which also has a total of k constraints [1]. This means that either we are extremely lucky and find a sequence of steps that keep improving the solution, or we will end up in a local minimum.

This is the algorithm we implemented, and it doesn't always solve the board exactly because of the aforementioned problem. Notice that we implemented the algorithm in such a way that between p possible "best" moves, one will be chosen randomly between them.

5.2.2.2 Possible improvements The first improvement that comes to mind consists in allowing *sideways moves* to try and escape local minima. While this could be a reasonable approach because it allows searching a wider space, the algorithm could get stuck in a loop, and therefore would require the user to set a maximum number of iterations of the algorithm, or at least it would require some form of loop-checking.

Many other improvements are possible. For example, we might keep many (say k) configurations at a time, run an iteration of the algorithm and then choose the k best configurations from the previous k plus the new k. This is called **Local Beam Search** [1] and is very useful because it shares knowledge about the current quality of the considered solutions.

5.3 Implementation

5.3.1 Explaination

The implementation relies on two key functions: `find_next_move` and `solve`. `solve` is very much similar to the other ones: it looks for a possible improving move and either performs it or returns the current state if we are in a local minimum, i.e. no improvement is possible following the greedy strategy.

`find_next_move`, on the other hand, is the function in which most of the "smart" work happens. I will not report the code here since it is quite a long function, but I'll explain what it does.

What we do at the beginning is to choose the columns whose queens are the *most constrained*. After having gotten them, we go on to analyze, for each of these columns (shuffled to try and search a wider space), the possible improvements that swapping two rows might produce, and we save the best ones. At this point, either there are no possible moves to get closer to the optimum (i.e. we are stuck in a local minimum), or we choose one of the k (which we set to be at most 5) possible row-switches randomly. It is worth to notice that there might be other moves that improve the situation even more, but we kill the search when we reach a maximum length of 5 possible moves in order to avoid the chance of spending too much time searching for a move and getting, in the end, a small improvement compared to the time taken to find that move. From our test, setting

the length of the list of possible moves to be at most 5 works fairly well as a compromise between quality of the moves and time taken to get the move to be chosen.

A similar heuristic will be found in the Global Search. We don't have any theoretical proof of their quality (that's why we call them heuristics in the first place), but in practice, form our tests, they result in a decent time to get the result, and in the end that's what we mostly care about.

5.4 Performance analysis

5.4.1 Results

The test here reported consisted in 10 runs of Hill Climbing on a 77×77 matrix. Here are the results.

Time:

- Total time required: 0.45 seconds
- Average time required: 0.045 seconds
- Time taken by the worst iteration: 0.05 seconds
- Number of actual solutions = 5 out of 10 (50.0%)
- Standard deviation (corrected) of the results 0.01 seconds
- Average time required not considering the worst iteration: 0.044 seconds

Steps:

- Total steps: 204
- Average steps required: 20.4
- Steps taken by the worst iteration: 23
- Standard deviation (corrected) of the steps: 21.5 steps
- Average steps required not considering the worst iteration: 20.1 steps

5.4.2 Considerations

As we can see, the algorithm is *extremely* fast, but it doesn't always find a soution. From our test, it almost always finds a solution with big matrices (e.g. 1000×1000), but it suffers smaller matrices with a smaller number of possible solutions. Anyway, it is so fast that re-iterating it up to k times (say 10) might produce a solution much faster than running Constraint Propagation. For example, in ~38 seconds we get an actual solution of a 1000×1000 board, and most of the time is taken by the initialization of the constraints on the board.

6 Global Optimization

6.1 Simulated annealing

We define simulated annealing (*stochastic hill climbing*) as the following algorithm [1]:

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    current ← MAKE-NODE(problem.INITIAL-STATE)
    for t = 1 to ∞ do
        T ← schedule(t)
        if T == 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.VALUE - current.VALUE
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

For the sake of precision, simulated annealing *approximates* global optimization and it is used mainly when the search space is discrete [2]. Also, while simulated annealing is often used to approximate a global optimum, our implementation actually finds one (even though it might take forever in extremely unlucky cases).

6.2 Implementation

The implementation of the global search is quite similar to the one of the local search. To be honest, our algorithm is kind of different from the standard simulated annealing, since we use some heuristic to choose the next move instead of going completely random (which works terribly) and only apply a completely random transformation if no possible improvement is possible (i.e. in the case in which the local search would return a local minimum). Notice that also in this case our moves consist in swapping two rows in the board.

The key idea is the same as local search: look for some smart moves to be done and do one of them. If there are no possible moves that would improve the current situation, perform a completely random move in the hope of getting out of local minima. From our tests, this is one of the configurations that work best, because we tend to have significant improvements from one state to the next one and we move towards the solution quite quickly. Also, we noticed that even when performing the completely random move, the configuration doesn't get *that* bad, for usually only a few constraints are added. All in all, this heuristic works way better than choosing a randomly selected successor as per the original algorithm, which often leads to unacceptable computational times.

Another heuristic we use to avoid having too-long computational time is to only check $log^2(n)$ columns to find the next move until we are close to the solution, in which case we start scanning all the columns in the hope of not having to make a completely random move. Also in this case, we have no reason to say that this approach is better than others, but from our tests it reduced significantly the running time of the global search. The key idea behind this is that when we are "far" from the solution, it is easier to improve the configuration than it is when we are really close to the solution and we need to make very careful moves.

The whole algorithm can be summarized as follows:

```
*until the current state is not a solution*
1. choose one of the most constrained queens
2. find another queen such that by swapping the rows of these two queens,
    we get closer to a solution
3. if such a queen does not exist, then choose two completely random queens
4. swap the two chosen queens and go back to 1
```

6.3 Performance analysis

6.3.1 Results

The test here reported consisted in 10 runs of Simulated Annealing on a 77×77 matrix. Here are the results.

Time:

- Total time required: 0.64 seconds
- Average time required: 0.064 seconds
- Time taken by the worst iteration: 0.13 seconds
- Standard deviation (corrected) of the results 0.03 seconds
- Average time required not considering the worst iteration: 0.057 seconds

Steps:

- Total steps: 383
- Average steps required: 38.3
- Steps taken by the worst iteration: 66
- Standard deviation (corrected) of the steps 44.6 steps
- Average steps required not considering the worst iteration: 35.2 steps

6.3.2 Considerations

As we can see, Simulated Annealing is probably the best compromise between performance and optimality of the solution. It always finds an actual solution, and it does so really fast. In our implementation, it takes 5-6 minutes to solve a 5000×5000 board, taking more than 1 minute to initialize the board. All in all, this is pretty satisfying. An even better improvement can be found using the Kronecker product: factorize n, solve the "sub-boards" with Global Search and then, if the board obtained computing the Kronecker product is not an actual solution, re-run Global Search on it. Except for the unlucky case of prime numbers, this method seems to be extremely fast and always returns an actual solution.

7 Comparison

7.1 Measures

A comparison can be done in many ways: we can study the time taken to give a solution, whether that solution is an actual one or only a local minimum, the variance of the time taken between many runs given the same input size n and much more.

The following table summarizes the measures we have taken on the test we have already mentioned (10 runs on a 77×77 board). More conclusion will be drawn afterwards.

Algorithm	Always solves to optimality	Percentage of actual solutions	Average time taken	Standard deviation
Constraint Propagation & Backtracking	Yes	100%	28.8s	76.5s
Hill Climbing	No	50.0%	0.045s	0.01s
Simulated Annealing	Yes	100%	0.064s	0.03s
Hill Climbing from Kronecker	No	70%	0.023s	0.01
Simulated Annealing from Kronecker	Yes	100%	0.012s	0s (too small)

Also, here is the comparison between solving the "smaller" boards in Kronecker with Constraint Propagation vs. Global Search:

Algorithm	Always solves to optimality	Percentage of actual solutions	Average time taken
Kronecker CPB	No	70.0%	0.01s
Kronecker GS	No	80.0%	0.011s

7.2 Drawing some conclusions

Having looked at the results and at the pros and cons of each method, some conclusions can be drawn. In particular, from our test and our implementation, Simulated Annealing seems to be the best compromise between performance and optimality: it is much faster than Constraint Propagation and slightly slower than Hill Climbing, but with respect to the latter it always finds an actual solution. Constraint Propagation seems to be a fairly decent choice to solve "smaller" boards when using Kronecker, but also in that case -especially when factors are big or the number is prime- Simulated Annealing either performs equivalently, or better.

All in all, the best choice seems to be Simulated Annealing. Other, less preferable options are Hill Climbing (many runs might be required) and Constraint Propagation and Backtracking (especially if combined with Kronecker).

It has been very interesting, however, to have the opportunity to implement and see the differences between these algorithms. Even if it looks like there is a "winner", the different implementations

might be suitable for different problems, and having had the chance to implement all of these algorithms, I have surely got a deeper understanding of how these techniques actually work.

8 References

[1] *Russell S., Norvig P.* Artificial Intelligence, A Modern Approach, 3rd edition

[2] https://en.wikipedia.org/wiki/Simulated_annealing

[3] Professor's slides