Local and Systematic Search

Author: Aleix Segura Paz
Major: Computing
Date: 07-03-2024

Assignment 1
Advanced Programming in Artificial Intelligence
Universitat de Lleida

Contents

List of Figures

1 Introduction

In this report it's explained both local search and systematic search algorithms. In particular, a basic local search algorithm and simulated annealing algorithm are explained in the part of local search, and a Horn formula solver algorithm for the systematic search part.

Also, plots are provided for some algorithms in order to analyze their solutions and complexities.

1.1 Goal

The goal of implementing from scratch these algorithms is to acquire a better understanding of how do the algorithms work, their weaknesses, the modifications that can be applied and to learn to decide about in which problems we can apply them.

2 Local Search

2.1 Basic Local Search

For the basic local search algorithm the strategy is based in given a number of tries, creating new random problems (states) and check if the cost function is better for the new generated. In our case, the cost function returns the number of conflicts of a given board state.

This approach has a good time complexity but the quality of the solution is not the best if we compare it with another variations of local search.

2.2 Neighborhood Local Search

The neighborhood local search introduces the concept of visiting the neighbors of a state in order to find local and global maximums. With this algorithm the time complexity increases due to visiting neighbors, but the quality of the solutions is quite better than the first approach.

2.3 Simulated Annealing

The simulated annealing algorithm is a more sophisticated one. It has the idea of having a value 'temperature' which is being decreased during the iterations of the algorithm and it's used to increase the strictness of selecting new worse states as the algorithm advances.

If a new random move is better than the current state we simply update the current solution, but if it's not, a probability of acceptance based on the difference of the cost between current state and neighbor state is calculated. Then a random in the range of [0, 1] is generated and if the probability is higher than the random and the temperature is also higher than the random we explore this new path. This can help in order to explore some paths that would not be considered with a different and more basic approach.

The Simulated Annealing is a good algorithm compared to other search algorithms, because it combines hill-climbing technique and a random factor to avoid local maximums. The algorithm has a similar (in some cases faster, in other cases bit slower) time complexity than the neighborhood local search but the solution quality is much better.

2.4 Solution quality and time complexity comparison

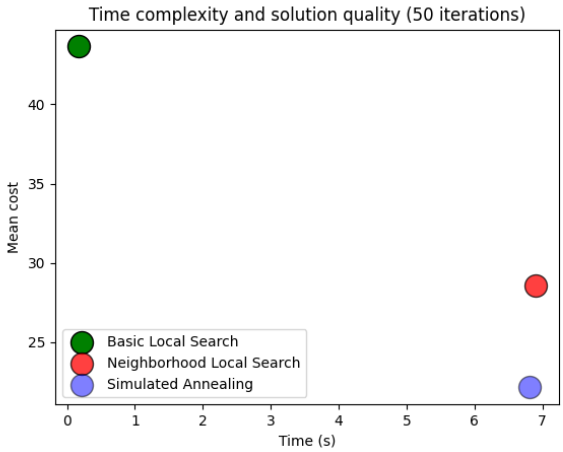


Figure 1: Time complexity and solution quality plot related to the named algorithms.
Board size of 50 and 20 tries.

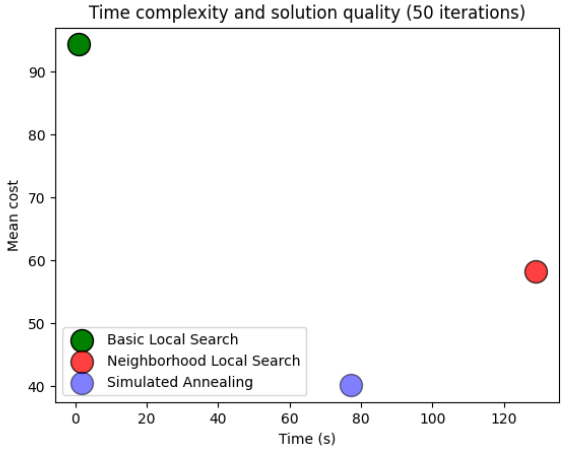


Figure 2: Time complexity and solution quality plot related to the named algorithms.
Board size of 100 and 30 tries.

2.5 Genetic algorithm

For the Genetic N-Queens algorithm firstly the implementation was made following the code provided by the professor. While developing the functions, some little variations we're made because it was, personally, easier to understand the next steps of the implementation.

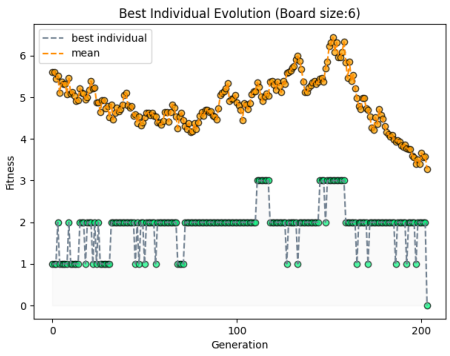
The first solution was okay, it was a genetic algorithm, but it was easy to saw that it could be optimized easily. For this reason, two ways of creating the next iteration of population we're provided, the original and the optimized version.

The optimized version it's based in selecting for a new iteration of population, a group (50% approx.) of elite individuals from the previous population. We consider that an elite individual is that individual that has a cost lower than the mean cost of the population. Also, a function argument called *'elitism degree'* is provided in order to provide more strictness in selecting the elite individuals.

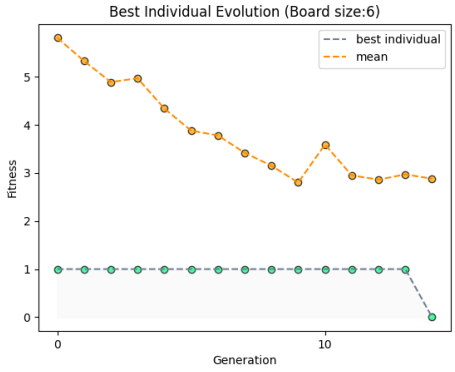
With this optimized version we obtain solutions much faster for bigger boards of queens. This is obviously in consequence of having a good solid base of elite individuals in each iteration, which reduces significantly execution time. In the next section we appreciate the difference between the two implementations.

2.5.1 Comparison between the two versions

For a size of board 5 we can't appreciate significant differences, it both cases finalizes between the range of [1, 3] generations in most of the executions. But as we increase the size of the board we appreciate bigger differences:

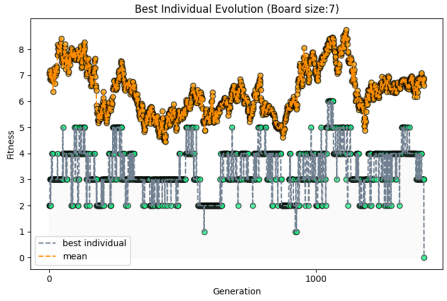


(a) Not optimized execution for a board of size 6.

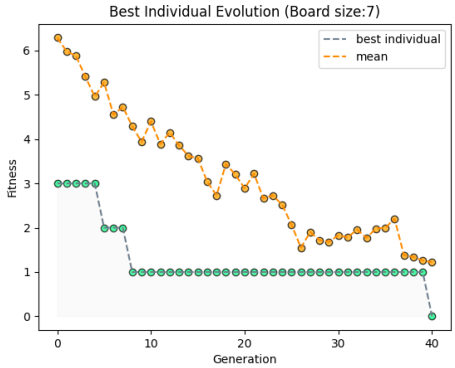


(b) Optimized execution for a board of size 6.

Figure 3: Comparison of execution for a board of size 6.



(a) Not optimized execution for a board of size 7.



(b) Optimized execution for a board of size 7.

Figure 4: Comparison of execution for a board of size 7.

As we see just by the clarity of the plots, we can rapidly see which plots correspond to the optimized version. The plots show how easy is for the optimized version to found a solution, while the original version struggles a lot because has a higher degree of randomness.

This degree of randomness can be appreciated by looking the point that is the solution, which jumps straight to 0 from a fitness of 2 in board size 6 and from a fitness of 3 in board size 7, this straight jump to the solution indicates that you need to have a bit of luck (more luck needed as we grow the board size) in order to have a solution. In the other hand, the optimized version doesn't rely that much on luck an shows a clear descendant linear function.

3 Systematic Search

3.1 Horn formula solver

The algorithm used to build the solver follows the following structure: first, the clauses are parsed from the clauses text file, then we filter the complete clauses that it positive variable is not also contained in unitaries found. Next step is cleaning 'the left part' of the 'complete' clauses by eliminating from each clause those variables that are in the unitaries. Also, in this step, we have to check if we have erased all variables of the left part (the negatives) because if that happens we have a new unitary which is the positive of that clause that now doesn't have negatives. Once the negatives are cleaned, we have to repeat the first step by filtering by the complete clauses that it positive variables is not in, now, the new unitaries found.

When the simplification process is made, we can solve our final Horn formula.

3.1.1 Usage

In order to execute the solver script named `solver.py` the following syntax is accepted:

- `python3 solver.py clauses.txt` (for a normal execution)
- `python3 solver.py -p clauses.txt` (for an execution with phase transition plot)

Note: if plot execution applied, remember to modify the iterations of each size if you want a more or less correct phase transition function.

3.1.2 Phase Transition

In order to compute the Phase Transition, we execute several times (1000 for the plot provided) a determined formula with **N** clauses to compute it's probability of being satisfiable. We can appreciate how the probability falls as the number of clauses goes up. In the other hand, hardness increases as number of clauses goes up.

A plot showing the concept of *Phase Transition* can be found in next page.

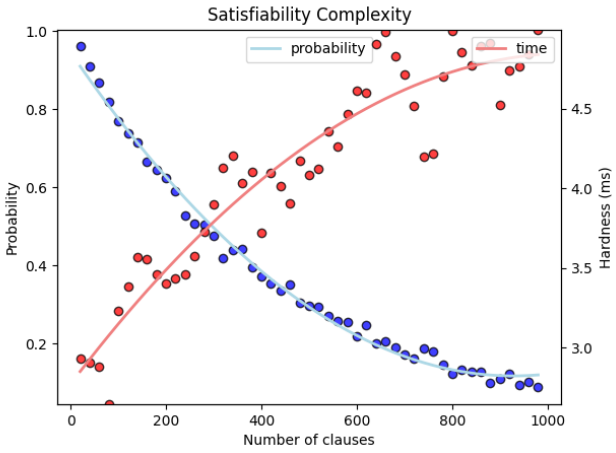


Figure 5: Phase transition for multiple clause sizes with 1000 iterations per size.