

Foundation of Artificial Intelligence

Assignement 2

Davide Vitagliano

Academic Year 2023/2024

Sommario

[1. Requirements 1](#_Toc149483564)

[2. Introduction 2](#_Toc149483565)

[2.1. Constraint Satisfaction Problem (CSP) 2](#_Toc149483566)

[2.1.1. Definition 2](#_Toc149483567)

[2.1.2. Sudoku as a CSP 2](#_Toc149483568)

[3. Designing the solution 3](#_Toc149483569)

[3.1. Constraint Propagation and Backtracking 3](#_Toc149483570)

[3.1.1. Formalization 3](#_Toc149483571)

[3.1.2. Code explanation 4](#_Toc149483572)

[3.1.3. Conclusion 6](#_Toc149483573)

[3.2. Simulated Annealing 6](#_Toc149483574)

[3.2.1. Formalization 6](#_Toc149483575)

[3.2.2. Code explanation 7](#_Toc149483576)

[3.2.3. Conclusion 9](#_Toc149483577)

[4. Comparison 10](#_Toc149483578)

# Requirements

Write a handwritten digit classifier for the MNIST database. These are composed of 70000 28x28 pixel gray-scale images of handwritten digits divided into 60000 training set and 10000 test set.

Train the following classifiers on the dataset and use 10 fold cross validation to optimize the parameters:

* SVM using linear, polynomial of degree 2, and RBF kernels;
* Random forests;
* Naive Bayes classifier where each pixel is distributed according to a Beta distribution of parameters:

With

* K-NN;

# Introduction

## Constraint Satisfaction Problem (CSP)

### Definition

A constraint satisfaction problem is defined by three components: X, C, D:

* X is a set of variables {X1, X2, …, Xn};
* C is a set of constraints specifying allowed combination of values;
* D is a set of domains {D1, D2, …, Dn} consisting of a set of allowed values {v1, v2, …, vk} for each variable Xi.

### Sudoku as a CSP

The game of Sudoku can be formalized as a CSP having:

* X = cells of the board of size 9x9, which means 81 variables in total;
* C = each digit must appear only one time per each row, column and box; then logically, each row, column and box must contain every digit;
* D = every cell can contain a digit {1, …, 9} or a “.” That indicates the empty cell.

The problem can be solved using different approaches:

* Backtracking Algorithm: a depth first search (DFS) algorithm for finding solutions to CSPs. It incrementally instantiates variables and extends a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable. If it finds an inconsistency, it backtracks to a new instantiation;
* Constraint Propagation: This technique iteratively enforces constraints locally. It’s used to reduce the size of the search space by eliminating values from the domains of variables that cannot be part of any valid solution;
* Simulated Annealing: it’s a metaheuristic approach used to approximate global optimization that performs better in a discrete domain. It slowly decreases the probability of accepting worse solutions for the CSPs while exploring the solution space;
* Genetic Algorithms: they are a type of evolutionary algorithms used to generate high quality solutions to CSPs. It iteratively performs a stochastic selection for fitter individuals and applies a mutation to each of them to produce a new generation. Then, it restarts the cycle until either a maximum number of generations has been produced or a certain fitness level of the population is satisfied;
* Gradient Projection algorithm: it’s a method used to navigate the search space while respecting constraints. It starts from an initial point in the optimization function and iteratively updates this point by moving in the opposite direction of the gradient until either a maximum number of iterations is reached or a certain level of precision is satisfied.

# Designing the solution

## Constraint Propagation and Backtracking

### Formalization

Let’s assume the board is *n* x *n*. If a digit *k* is placed in cell *(i, j*), then:

* Row constraint:
* Column constraint:
* Box constraint:

This reduces the possible steps that can be made after the current one. Indeed, the constraint propagation phase consists in trying and removing, from the domain of each variable, the value that violate the constraints. Whether this results in a variable with domain D = {}, then backtracking is applied to try a different assignment.

During Constraint Propagation phase, there are two ordering heuristics that can be applied to the algorithm:

* Most constrained variable: pick a variable with fewest legal values that can be assigned;
* Least constrained variable: pick a value that removes fewest values from neighboring domain.

In this case, the first one is chosen to prune more often the search space and it applies the Only Choice strategy with more efficacy.

In each epoch, after the constraint propagation phase, Only Choice strategy is applied: for each cell with only one value admitted, that value is assigned to the cell itself. This can considerably improve performances and drastically reduce the chances of backtracking to a previous state of the board.

### Code explanation

The rows of the Sudoku board are labeled by the letters {A, B, C, D, E, F, G, H, I}. The columns are labeled by the numbers {1, 2, 3, 4, 5, 6, 7, 8, 9}. So each cell has label {A1, A2, …, I9}. These labels are used to address each cell in the code.

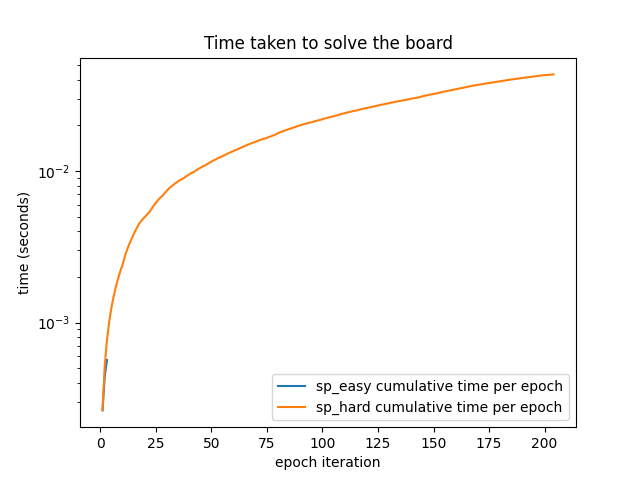
A *unit* consists of a set of 9 cells belonging to the same row, column or 3x3 box, and for each cell, a *peer* is another cell that belongs to a common *unit*.

* The board
  + The board itself will be represented in the form of a dictionary with the keys as labels of each cell (‘E7’) and the values as the digit in the cell.
* The units
  + The 27 units will be stored as a list of lists with each inner list representing a single unit that contains all the labels of cells that belong to the same row or column or 3x3 box;
  + This unit list is used to create a unit dictionary to store the information about which cell is a part of which unit. Its keys are the labels of cells and its values are a list of all the units of which the cell is a part of.
* The peers
  + The peers are stored as a dictionary with the keys representing the labels of cells and the values representing the set of all the peers of the cell.

Follows the detailed explanation of each method of the class:

* Initialization (*\_\_init\_\_* method):
  + The class initializes several attributes, such as *\_ROWS, \_COLS, \_CELLS*, and *exec\_time*.
  + It defines the structure of the Sudoku grid and its units (rows, columns, and 3x3 boxes).
  + It creates a dictionary of units for each cell and a dictionary of peers for each cell.
* Solving Sudoku (*solve* method):
  + This method is used to solve the Sudoku puzzle using an implicit version of backtracking.
  + It initially calls the *\_run\_epoch* method to perform constraint propagation and the Only Choice strategy.
  + If the Sudoku puzzle is unsolvable, it returns *False*.
  + If all cells have only one value, it means the puzzle is solved, and it returns the solved board.
* Constraint Propagation (*\_constraint\_propagation* method):
  + This method eliminates the values of solved cells from their peers.
  + It iterates over cells, identifies peers, and removes the values of the solved peers from the current cell.
* Only Choice Strategy (*\_set\_value* method):
  + This method assigns a value to a cell if it's the only choice for that value in its unit (row, column, or box).
  + It iterates over units, numbers, and identifies cells where a number is the only choice. If found, it assigns the value to that cell.
* Epoch Execution (*\_run\_epoch* method):
  + This method executes a single "epoch" of constraint propagation and Only Choice strategy.
  + It iterates through the board, applying constraint propagation and setting values if possible.
  + The process continues until no more changes can be made to the board.
  + It also tracks the execution time of each epoch.

### Conclusion



From the chart it’s concluded that Constraint propagation and Backtracking approach, with the use of the Only Choice strategy, is very efficient while solving simple Sudoku board: it takes up to 10-3 seconds to solve the board in 3 epochs. Moreover, this approach can solve the world’s hardest Sudoku problem in less than 10-1 seconds varying from 100 to 300 moves.

## Simulated Annealing

### Formalization

Simulated Annealing is an optimization technique which finds the optimal point by running series of moves under different thermodynamic condition. During each move, a neighboring state is determined by randomly changing the current state of the individual. The new state is evaluated via a cost function. If a state with a lower cost is found, then the individual moves to that state. Otherwise, if the neighboring state has a higher cost, then the individual moves to that state according to the Metropolis rule:

* Calculate an acceptance probability;
* Generate a random number between 0 and 1;
* Compare this probability to the random number. If the probability is superior than the random number, then we accept the new solution (neighbor). If not, it is rejected.

The acceptance probability is calculated using an additional parameter known as temperature T, which scales the difference between the new and current solution by a certain amount. This is expressed as:

where:

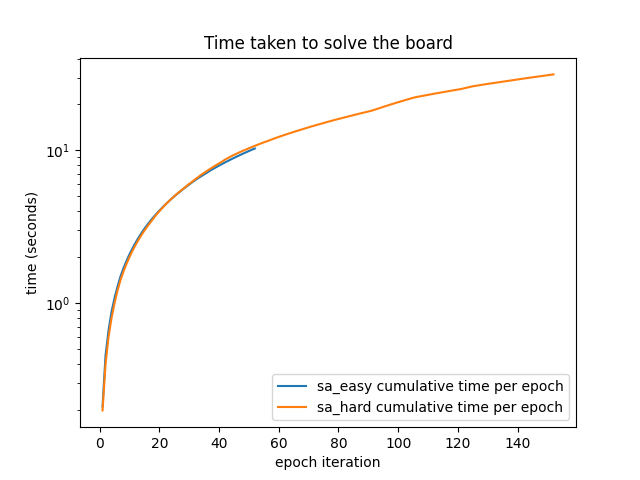
* x and x′ are the current and new solutions respectively;
* f(x) and f(x′) are their corresponding cost function values;
* T is the temperature parameter.

When the temperature is high, even solutions that are worse than the current one can be accepted, allowing the algorithm to escape from local minima. As the temperature decreases, the probability of accepting worse solutions also decreases.

### Code explanation

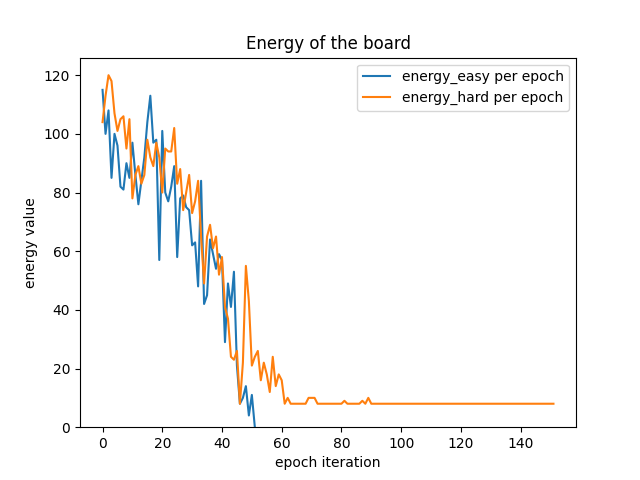
* Initialization (*\_\_init\_\_* method):
  + This method initializes the class. It takes a string board as input, representing the Sudoku board, where "." indicates empty cells. The code converts this string into a 9x9 NumPy array and assigns random values to empty cells as a starting point. It also sets the initial temperature and cooling rate for the simulated annealing algorithm.
* Solving Sudoku (*solve* method):
  + This method is the core of the Sudoku-solving algorithm. It uses simulated annealing to iteratively improve the Sudoku board until a solution is found or until a certain temperature threshold is reached.
  + The algorithm runs a loop while the temperature is below a certain threshold, in this case 10. Within each epoch, it runs the Metropolis rule algorithm 5000 times.
  + The Metropolis rule algorithm is used to evaluate whether a proposed change to the Sudoku board, changing the value of a random cell, should be accepted based on the change in energy, a measure of how close the solution is. If the energy decreases, the change is accepted. However, even if the energy increases, the change can be accepted probabilistically based on the temperature.
* Local Energy (*\_local\_energy* method):
  + This method calculates the energy associated with a specific cell in the Sudoku board.
  + It checks the row, column, and 3x3 box containing the cell and counts how many times each number 1 to 9 appears in these regions. The energy is increased for each number that appears more than once in these regions.
  + The local energy of a cell reflects how many constraints are violated by the current assignment of values.
* Global Energy (*\_global\_energy* method):
  + This method calculates the global energy of the entire Sudoku board by summing up the local energies of all cells.
  + This function is used to determine the overall quality of the Sudoku board.
* Metropolis Algorithm (*\_metropolis* method):
  + This method is the heart of the simulated annealing process. It attempts to make a random change to the Sudoku board and decides whether to accept or reject that change.
  + It selects a random cell with a missing value from *\_guesses*, records the current energy of that cell, and the current value in that cell.
  + Then, it assigns a new random value to the cell and calculates the change in energy due to this assignment.
  + If the change in energy is negative or is accepted with a certain probability based on the temperature, the change is applied.
  + If the change is not accepted, the value of the cell is reverted to its previous value.

### Conclusion



From the chart it’s concluded that the Simulated Annealing approach has similar performance in solving easy and hard Sudoku board: it can takes up to 102 seconds to solve the board in a varying number of epochs, but it’s more likely it stops on a suboptimal solution for harder boards. This depends on the choice of some parameters:

* Starting temperature (inverse): it controls the initial exploration behavior, a higher initial temperature encourages more exploration, while a lower temperature promotes exploitation;
* Maximum temperature: it represents an upper limit on the temperature during the annealing process. It can help preventing the algorithm from running for an excessive number of iterations at high temperatures;
* Cooling rate: it determines how fast the temperature decreases during the annealing process. In this case it’s used an exponential cooling schedule, where the temperature is reduced by a certain factor at each epoch;
* Iterations per level of temperature (epoch): it determines how thoroughly the search space is explored at that temperature.



From this chart it’s deducted that, with a starting temperature=0.5, maximum temperature=10, cooling rate=0.02 and iterations of Metropolis rule per epoch=5000, the easy board is solved after about 50 epochs in about 10 seconds. But the world’s hardest board converges to a suboptimal solution with few cells violating the constraints.

# Comparison

When comparing the two approaches, it’s clearly found that both converge to an optimal solution for easier Sudoku boards, but in case of harder boards, the Simulated Annealing approach could converge to a suboptimal solution, except in the case the above-mentioned parameters are properly set.

Comparing the time charts, it’s concluded that the Constraint Propagation and Backtracking approach is much faster than Simulated Annealing for both easier and harder boards.