Optimize a Data Center

Artificial Intelligence 21/22

Beatriz Aguiar, João Marinho & Margarida Vieira

Problem Specification

Given a schema of a data center and a list of available servers, the goal is to assign servers to slots within the **rows** and to logical **pools** so that the **lowest guaranteed capacity** of all pools is **maximized**.

Servers are **physically** divided into **rows**. Rows share resources, therefore if a resource fails in a row, all servers in that row become unavailable.

Servers are also **logically** divided into **pools**. Each server belongs to exactly one pool, and provides it with some amount of computing resources, called capacity, c. The capacity of a pool is the sum of the capacities, c_i, of the available servers in that pool.

To ensure reliability of a pool, it is therefore desirable to distribute its servers between different rows. The **guaranteed capacity** of a pool is the minimum capacity it will have when at most one data center row goes down.

Input data

R - number of rows in the data center

S - number of slots in each row of the data center

U - number of unavailable slots

P - number of pools to be created

M - number of servers to be allocated

Output data

OR - number of the row in the output file

OS - number of the slot in the output file

OP - number of the pool in the output file

X - server not allocated

Objective Function

$for\ i\ (0\ \leq i < P), the\ guaranteed\ capacity, gc_i, can\ be\ defined\ as$

$$gc_i = min_{0 \leq i \leq P}\left(\sum_{k=0, server\ k\ in\ pool\ i, server\ k\ in\ row\ r}^{M-1} c_i\right) - \sum_{k=0, server\ k\ in\ pool\ i, server\ k\ in\ row\ r}^{M-1} gc_i$$

$$f(x) = min_{0 \leq i \leq P} gc_i$$

Optimization Problem

1 Neighbourhood and crossover functions

Neighbourhood function

- Change pool
- Change slot
- Change row
- Deallocate server

Crossover function

- Mix two parent solutions

2 Evaluation function

Given our goal is to maximize the guaranteed capacity, our evaluation function will calculate the maximum guaranteed capacity of the solution, i.e., the maximum value for all the possible lowest guaranteed capacities of the data center.

3 Solution representation

0 1 0 1 0 1 1 3 0 0 4 1 x	Server 0 placed in row 0 at slot 1 and assigned to pool 0. Server 1 placed in row 1 at slot 0 and assigned to pool 1. Server 2 placed in row 1 at slot 3 and assigned to pool 0. Server 3 placed in row 0 at slot 4 and assigned to pool 1. Server 4 not allocated.

4 Hard constraints

- Each slot of the data center must be occupied by at most one server
- No server occupies any unavailable slot of the data center
- No server extends beyond the slots of the row
- Each server belongs to exactly one pool and one row

Implementation work

Data structures

- Python classes for the various problem entities
  - **DataCenter**, main class that contains all global data – servers, rows, pools
  - **Server**, with size, capacity and associated pool
  - **Row**, with list where –1 is empty, -2 is unavailable and x, x >= 0, represents the number of the server
- Lists for storing the problem domain and the solution

Programming Language

IDE

Developed Approach

Solution construction

A solution is itself an object class, holding the rows, pools and servers, as well as the evaluation, value calculated upon initialization. For the initial solution we opted for a first fit approach, i.e., we cycle through the servers, attempting to place each one in the first row that fits. The following solutions are constructed by applying one of the four neighbor functions listed below:

- ***change_pool***: changes the pool of a randomly selected server
- ***change_row***: changes the row of a randomly selected server
- ***allocate_server***: allocates a randomly selected non-allocated server
- ***swap_allocation***: deallocates an allocated server and allocates a non-allocated server, both randomly selected
- ***swap_allocated_servers***: swaps the position of two randomly selected allocated servers

Evaluation function

```
class Solution():
    ...
 def evaluate(self):
        guaranteed_capacity = [0 for _ in range(self.pools)]
        max_row_capacity = [0 for _ in range(self.pools)]

        for row in self.rows:
            idx = 0
            row_capacity = [0 for _ in range(self.pools)]
            slots = row.slots

            while idx < len(slots):
                server_idx = slots[idx]
                if server_idx < 0:
                    idx += 1
                    continue

                server = self.servers[server_idx]
                row_capacity[server.pool] += server.capacity
                idx += server.size

            for idx in range(self.pools):
                max_row_capacity[idx] = max(max_row_capacity[idx], row_capacity[idx])
                guaranteed_capacity[idx] += row_capacity[idx]

        evaluation = inf
        for idx in range(self.pools):
            guaranteed_cap = guaranteed_capacity[idx] - max_row_capacity[idx]
            evaluation = min(evaluation, guaranteed_cap)

        self.evaluation = evaluation
```

Search Algorithm

Random Hill Climbing	Simulated Annealing	Tabu Search
Description and solution construction		
Finds a random current solution's neighbor and keeps it if it is better evaluated than the current one.	Finds a random current solution's neighbor and keeps it if it is better evaluated than the current one. If it is worse, keeps it based on a probability inversely proportional to the temperature, updated at each iteration.	Generates the current solution's neighbors and selects the best evaluated, if not in tabu memory. At each iteration, updates the tabu tenure of each tabu memory's entry.
Stop criteria		
Total number of iterations or total number of iterations with no solution improvement.		
Parameters		
- initial solution - maximum number of iterations - maximum number of iteration with no solution improvement		
	- initial temperature - temperature decrease factor, percentage of the temperature to be decreased in each iteration	- tabu tenure, to determine how many iteration a solution remains on the tabu memory

Genetic Algorithm

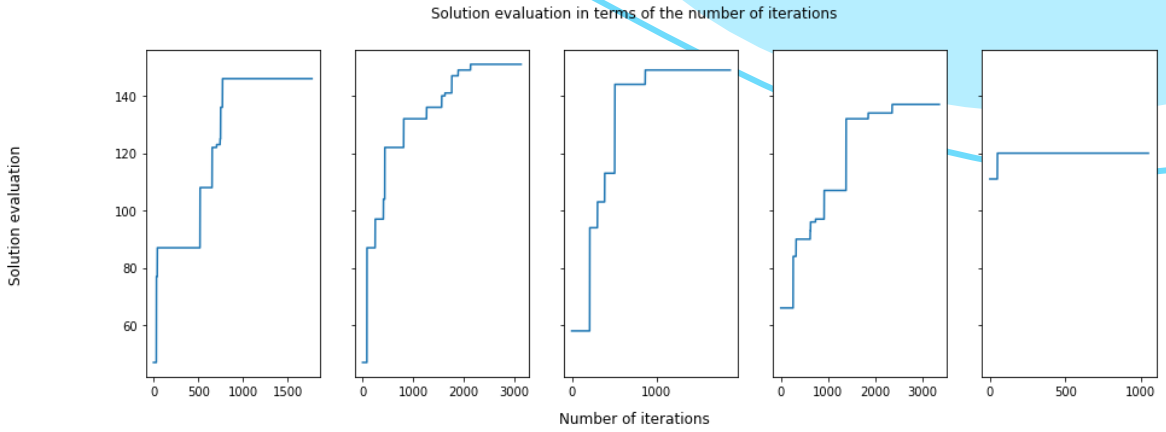
Description	Generate an initial population and evolve it by performing crossover and mutations on certain chromosomes. Of the new generation, are selected the surviving chromosomes, based on a configurable selection method.		
Stop criteria	Total number of iterations or total number of iterations with no solution improvement.		
Parameters	- initial population - maximum number of iterations - maximum number of iteration with no solution improvement - population size - mutation threshold, value below which a mutation occurs - selection and crossover methods to be applied		
Solution construction	**Crossover methods**	**Servers**	Combines the servers from both parents so that half of the servers from offspring1 remain the same as parent1 while trying to allocate the other half of servers from parent2, and vice-versa for offspring2.
		Pools	Combines half of the pools from both parents to procreate offsprings.
	Selection methods	**Tournament**	Selects two chromosomes by creating two random groups and selecting the best chromosome in both.
		Roulette	Associates the chromosomes' fitness level with a probability of selection. Randomly selects two chromosomes, considering that probability.

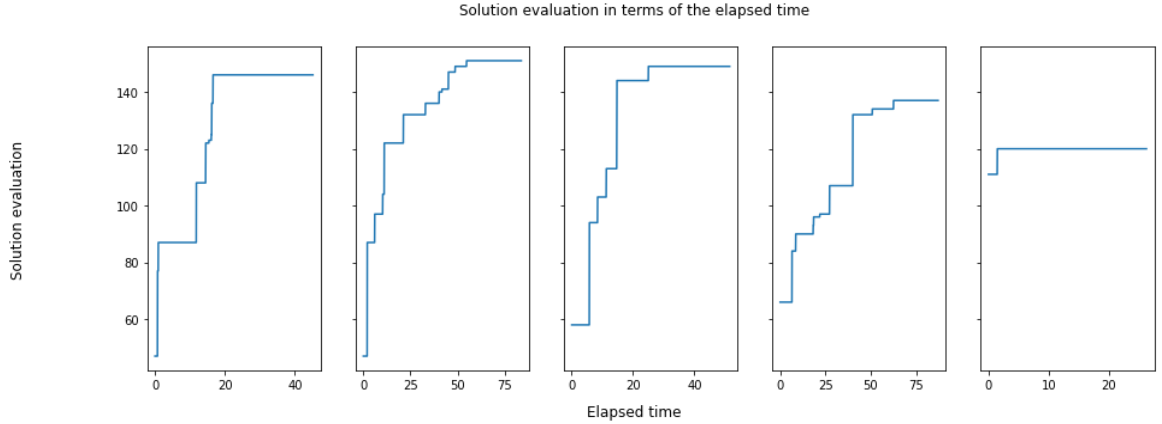
Experimental Results

Random Hill Climbing

As expected, the algorithm consistently found a better solution than the original random solution.

This way, the final solution is replaced by successive improvements to the current solution, represented by the instantaneous vertical jumps in the graphics.





Time and iteration graphics are extremely similiar to each other, hence only presenting one from now on.

Experimental Results

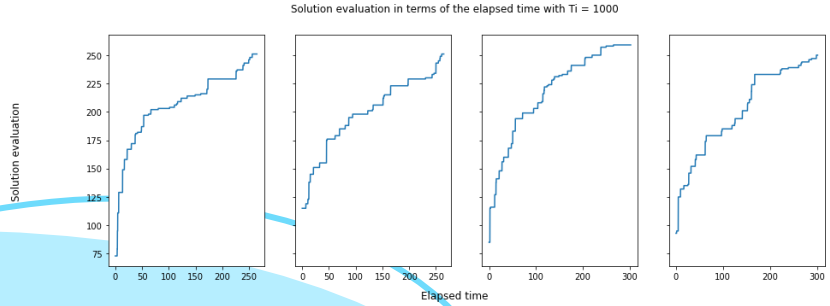
Simulated Annealing

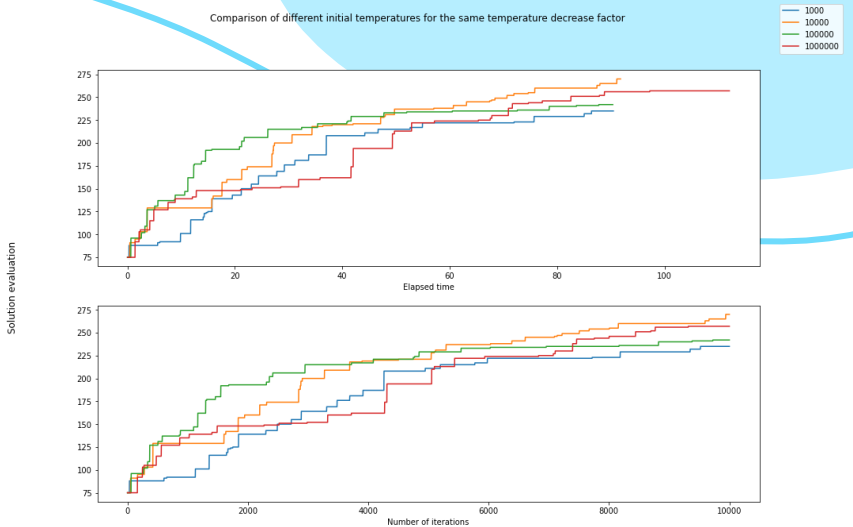
Two studies were conducted in relation to the Simulated Anneling algorithm.

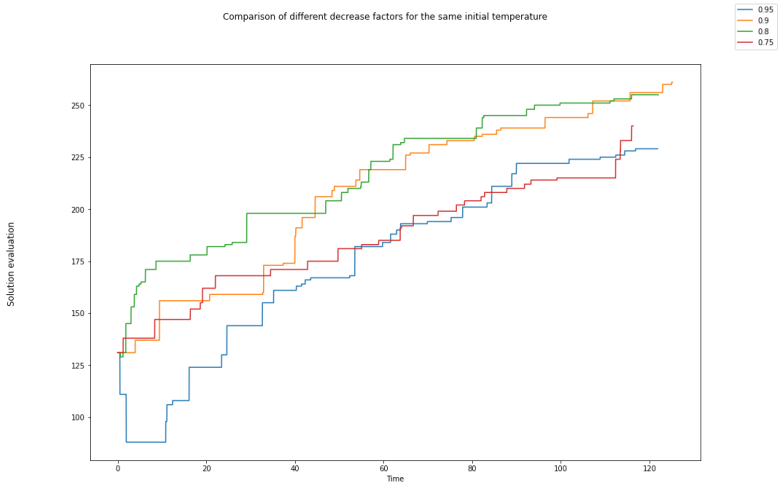
The first one focuses on how different initial temperature values affect the final solution, or the time required to obtain it. The second focuses on the use of different temperature decrease factors.

The initial temperature is not directly related to the solution evaluation. Nevertheless, it is evident that smaller initial temperatures lead to worse final solutions.

The acceptance of worse new solutions is very rare – probability increases for higher initial temperatures and smaller decrease factors – and does not always lead to better solutions.





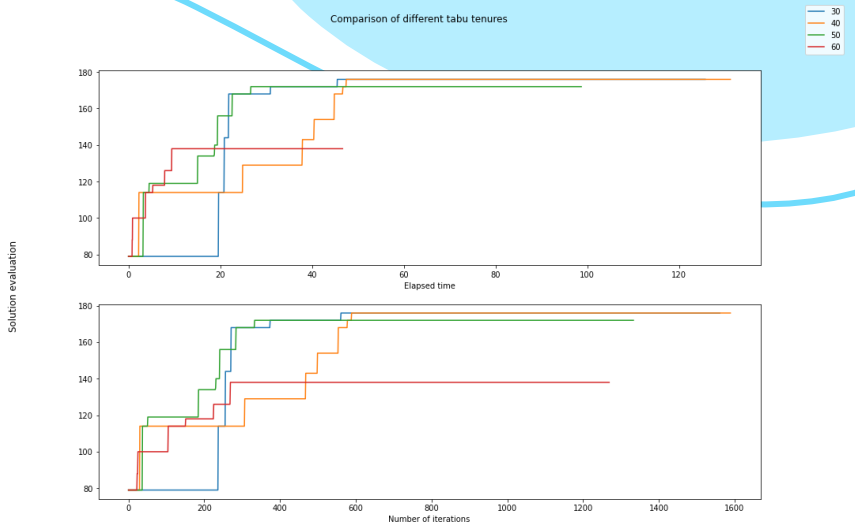


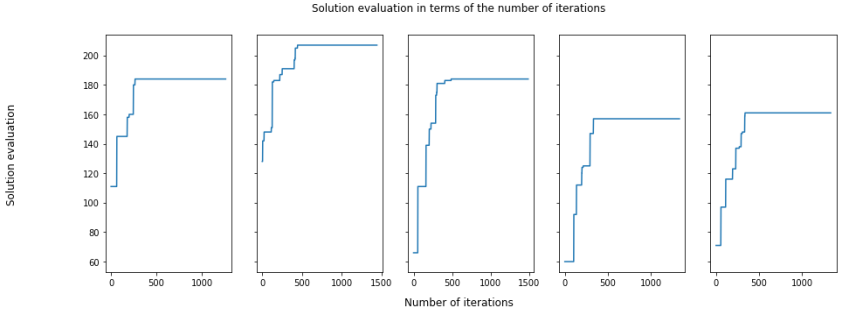
Experimental Results

Tabu Search

In this experiment, for the same initial solution, we changed the tabu tenure values, which indicate the number of iterations that a given solution remains in the Tabu Memory, list of solutions that cannot be visited during the current search.

For high tabu tenures, the solution might converge to worse local maximums.





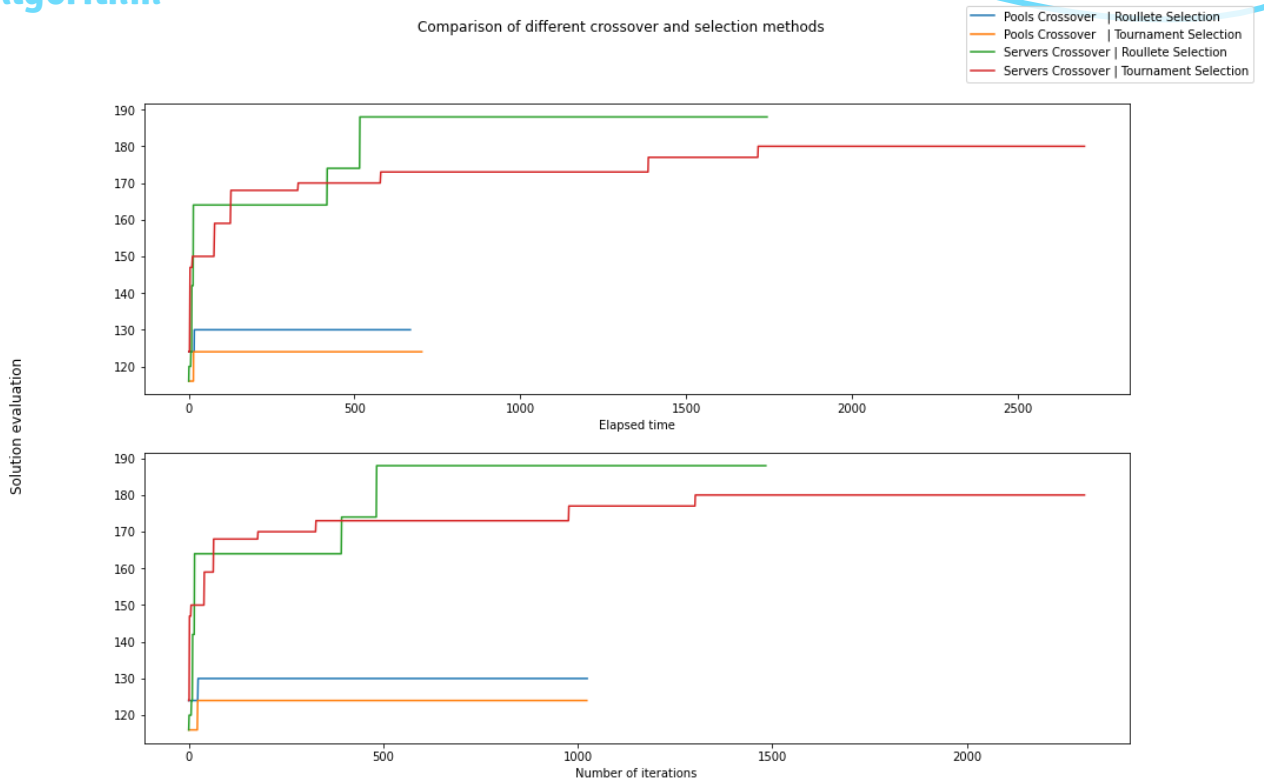
Experimental Results

Genetic Algorithm

The efficiency of this algorithm rests on the crossover and selection methods.

The servers crossover is evidently better at generating new solutions.

The roulette selection method is proven to obtain slightly better results than the tournament method, according to the generated graphs. A possible explanation is the fast convergence of the tournament selection method.

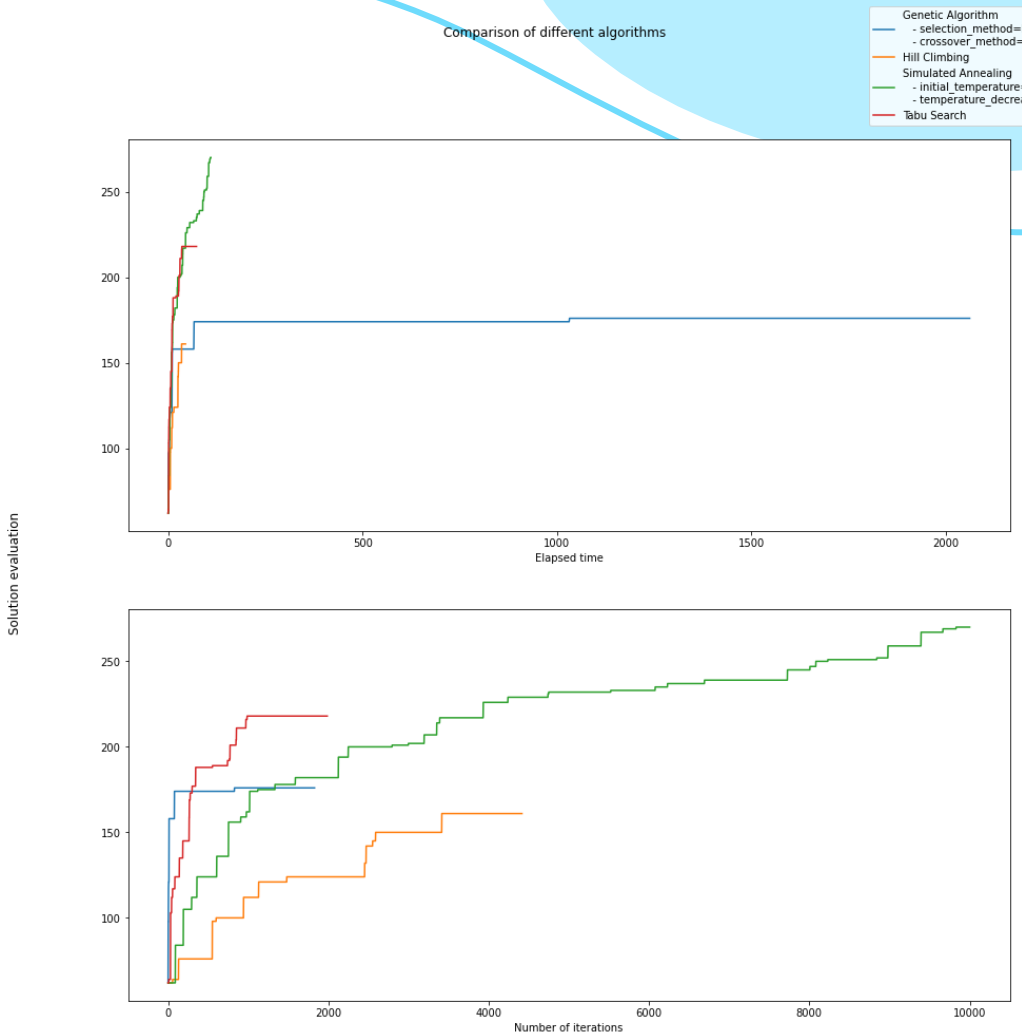


Experimental Results

Algorithms Comparison

The best solution is found by the simulated annealing algorithm within the average elapsed time.

The genetic algorithm is extremely time-consuming when compared to the other algorithms, and the hill climbing reveals itself as the worse, although, fastest algorithm.



Conclusions

The hill climbing algorithm stands out for its time efficiency. However, it finds the worse solutions, when compared to the other algorithms because it only accepts random new best solutions. When combined with additional acceptance criteria (hill climbing and tabu search), the results are significantly better.

The simulated anneling and the tabu search algorithms appear to obtain the best final results, since they are algorithms that seek to diverge from local maxima.

The genetic algorithm has the worst time efficiency, due to the number of operations applied to the whole population and the complexity of the crossover functions. This complexity, however, does not lead to the best new solutions given the current optimization problem.

Overall, the results are consistent and can be expected according to what was taught during the course lessons and researched during the development of the project.