Artificial Intelligence

Azamon

Package distribution optimization using local search algorithms.

Authors

Dànae Canillas

Miquel Escobar

Arnau Soler

Professor

Javier Béjar

Github Repository

`https://github.com/ArnauSoler/IA`

November, 2020

Contents

1 Problem Description

We are facing a problem raised by the fictitious company Azamon. Azamon, like Amazon, is a product distribution company that has to deal with daily package distributions working side by side with transport companies, the ones who are responsible for making shipments to customers. This company needs a logistics optimization system for its shipments in order to be efficient, minimizing its storage needs and shipping costs.

What is the secret of the success of these companies? The satisfaction of the customers, since they have the security that all their orders will be received in the estimated time and sometimes even earlier than expected, which increases their happiness.

1.1 Main goal

The main objective of the problem is to obtain for all the given packages to be sent in one day, an optimized shipping offers assignation for each of the packages. In the design of the solution we should take into account the costs associated with both transport and storage, and also the customers happiness, which comes from receiving a package earlier than expected.

We can solve this problem using local search since we are not interested in the way to reach the solution: we are only interested in arriving to the best possible solution achievable in a reasonable time, without the need to reach the optimum since the solution space is very large.

To solve it, we will start from an initial solution, and from there on we will try to improve it by using different operators that will iteratively discover neighboring solutions.

1.2 Problem Identification

- What elements are involved in the problem?

  It is an optimization problem where many elements are involved: we have a set **packages** that need to be distributed. Packages are sent to the customers by transport companies through a delivery **offer** agreement. The selection of the offer depends on the price and the maximum **weight** of packages that can be transported for each offer. Customers select their shipping **priority**: if the package arrives in advance the **happiness** value increases. Azamon wants to minimize the **price** of the transport plus the cost of having it stored waiting for it to be collected. In addition, it will also be considered as a criterion to maximize the happiness of the clients.

- Which is the search space?

  The search space groups all possible solutions that meet the requirements of the problem. More specifically, it corresponds to all the possible assignments of packages that not exceed the maximum weight for each of the offers and satisfies the shipping priority of all deliveries.

- How big is the search space?

  Given the number of packages N and the number of offers M, the search space has a maximum size of M^N, given the case that all possible solutions are valid, as those are all the possible combinations of package-offer assignations.

- What is an initial state?

  An initial state is the starting solution, which meets with all the requirements of the problem, and that does not have why to be a good one: it can be either a little bit more costly and complex to generate an initial solution, obtaining what is expectedly a better start, or it can be completely randomized completely ignoring the optimization criterion.

- What conditions does a final state meet?

  The final state must meet all the requirements and constraints of the problem, just like the initial solution and each of the intermediate states. Its peculiarity is that it contains the best value of the parameter that has been set to optimize. Thus, the final solution corresponds to the lowest price found (if only the first heuristic is used) or to a balance between the lowest price and the best happiness obtained (if the second heuristic is used).

- What operators allow modifying the states?

  The operators that allow us to modify the states are: add, remove, move and swap. Out of these, the valid ones to move inside the solution space are move and swap, which are also the ones we will use to define our sets of operators.

1.3 State implementation

The Azamon State has the following attributes:

- **packages** : *Paquetes* object class, contains all the packages (*Paquete* objects class) that must be assigned to an offer.
- **offers** : *Transporte* object class, contains all the offers (*Oferta* objects class) that the transport companies offer.
- **packageAssignments** : Vector of integers, represents each package offer assignment. It takes value -1 if the package has no assigned offer, and any other value corresponds to an offer in offers.
  - Data structure used:
    We have created the vector of assignments since this structure allows an efficient modification of the offer assignments for each package. In the worst case (depending on the search algorithm for an element in a vector) we get $O(n)$.
- **offersLoad** : Vector of doubles, contains the total weight of all packages that have been assigned to an offer of offers.
  - Data structure used:
    Due to the same reasons than **packageAssignments**
- **price** : Double, sum of storage and transport pricec.
- **happiness** : Integer, calculated according to the day the package is delivered and its priority.

Some design considerations:
- To gain spatial efficiency we have declared static the parts of the representation that are the same and thus can be shared among all the instances.
- We have decided to include the price and happiness variables, in the state so that we can keep the values at each state, and only have to calculate the delta at each state change (and not having to recalculate them all over again every time we use them).

1.4 Operators choice

The operators that allow us to obtain different solutions that we have considered have been:

`add package, remove package, move package, swap package.`

We have discarded the first two as they need to be used in a complementary way. If we only use one or the other, we can reach solutions that do not meet the requirements of the problem, solutions that are not valid. Using the *add* operator can lead to assign the same package to several offers. The same goes for the remove operator, it can lead to solutions where there exist unassigned packages. As it has been said before, these two operators must be used simultaneously to guarantee the creation of valid solutions.
An operator that guarantees this combined operation is: *move*, at any time a package is out of assignment. The same happens with the swap operator. These two operators will be explained in more detail below.

1.4.1 Move

`movePackage(int p_idx, int o_idx)`

- Applicability conditions: Moves the package `p` to the offer `o` if the priority of the package allows it and if the maximum transport weight is not exceeded once the package is added.
- Branching factor: Given the number of packages N and the number of offers M, the branching factor is $O(N \times M)$.
- Return: If the conditions to carry out the operation are achieved, the result of the operator will be the movement of the package `p` to the offer `o`.

1.4.2 Swap

`swapPackage(int p1_idx, int p2_idx)`

- Applicability conditions: Swaps the assignments of packages `p1` and `p2` if after the permutation of each package satisfies the delivery priorities and the transport weights sums are not exceeded due to this change.
- Branching factor: Given the number of packages N, the branching factor is $O(\frac{N \times (N+1)}{2})$.
- Return: If the conditions to carry out the operation are achieved, the result of the operator will be the swapping between the package with `p1` and `p2` indexes.

1.5 Initial solution strategies

We have generated three different strategies that obtain an initial solution with a reduced computational cost. The fact of generating different initial solutions will allow us to observe what it happens if a simple or a more elaborate initialization is used. These different solutions have been generated by modifying the ordering of packages and offers.
The packages are assigned sequentially to the offers, checking at all times that the maximum weight is not exceeded and that the delivery priority is correct. Remember we will use the following criteria and restrictions:

- All packages of the day must be assigned to a transport offer
- The weight of the packages assigned to an offer cannot exceed its capacity
- The weight of the packages assigned to an offer does not need to fill its capacity
- Packages must arrive within the period indicated by their priority, never later.

1.5.1 Generator A: Price optimization

The algorithm that generates this initial solution is the following:

```
packages := sort(packages) by priority (lowest to highest)
offers := sort(offers) by price (lowest to highest)
for package in packages:
    for offer in offers:
        if compatible(package.priority, offer.deliveryDays):
            if space_left(offer) > package.weight:
                assign(package, offer)
```

Note that a lower priority value means the package must be delivered before.

1.5.2 Generator B: Happiness optimization

The algorithm that generates this initial solution is the following:

```
packages := sort(packages) by priority (lowest to highest)
offers := sort(offers) by date (lowest to highest)
for package in packages:
    for offer in offers:
        if compatible(package.priority, offer.deliveryDays):
            if space_left(offer) > package.weight:
                assign(package, offer)
```

Note that a lower priority value means the package must be delivered before, and a lower date means that the package will be delivered earlier.

1.5.3 Generator C: Bad initialization

The algorithm that generates this initial solution is the following:

```
packages := sort(packages) by weight (highest to lowest)
offers := sort(offers) by date (highest to lowest)
for package in packages:
    for offer in offers:
        if compatible(package.priority, offer.deliveryDays):
            if space_left(offer) > package.weight:
                assign(package, offer)
```

In all three cases we are firstly sorting two vectors, one of length N (number of packages) and the other one of length M (number of offers), and the two ordering operations have a combined cost of $O(N \log N + O(M \log M) = O(max(N, M) \log max(N, M)$. Secondly, we are assigning each package to an offer, iterating through all packages and offers, which has a cost of $O(M \times N)$. This leaves us with a total cost of:

$$O(max(N, M) \log max(N, M) + O(M \times N) = O(M \times N)$$

Once we have set an initial solution, our software aims to optimize it using the Hill Climbing and Simulated Annealing algorithms considering a heuristic that will help them to obtain an adequate solution. Two heuristics have been designed for our purpose, which are explained in the section below.

1.6 Heuristic functions

1.6.1 Heuristic Cost

The main goal from this heuristic is to minimize the transport and storage costs as indicated in the first criterion of the project statement. The value that drives the function is the state variable `price` that corresponds to the cost for an assignment.

1.6.2 Heuristic Cost & Happiness

It uses the combination of the two criteria:

- Minimization of transport and storage costs
- Maximization of the customer happiness.

```
state.getPrice() - epsilon * state.getHappiness();
```

It is an heuristic that combines two attributes of the state: `price` and the `happiness` values, it has to balance them in such a way that the solution is good for the company.

The heuristic subtracts the `happiness` attribute from the `price` one. As the difference between the factors could assign more weight to one factor than the other, we have decided to include a weight factor, called epsilon. We will define its value in the Section 2.7.
Another solution would have been to divide the `happiness` attribute, so the larger the denominator, the smaller the result will be, in order to maximize and minimize at the same time.

1.7 Successor Generating functions

We have implemented two successor generating functions: one corresponding to the Hill Climbing algorithm, and the other one corresponding to the Simulated Annealing algorithm.

When we implemented theses functions we had to choose a set of operators to explore the search space, as explained in more detail in section **1.4. Operators choice**. When carrying out the different experiments we have evaluated all the different alternatives of sets of operators in order to justify the selection (see section 2.2).

1.7.1 Hill Climbing

The successor function for Hill Climbing generates, given an state, all the accessible states after applying the move operator for each package with all the offers, the swap operator for every combination of packages, or both operators.

1.7.2 Simulated Annealing

The successor function for Simulated Annealing for a state generates only one successor state of the accessible states after choosing a random operator and applying this operator to a random package and offer, or two packages.
See section 2.4 for a more detailed explanation on the parameter selection.

1.8 Goal State Function

As we are solving a problem using local search, it is not possible to know if the final state has been reached, so the function that implements this class always returns `false`.

2 Experiments

The problem has been implemented in such a way that random problems can be generated. In this case, the elements that vary are the number of packages to be delivered and the transport capacity of the offers.

An array with 10 different seeds has been created. Each seed will be tested in a specific iteration for all the experiments. We show the seeds values used below.

```
[1234, 2345, 3456, 4567, 5678, 6789, 7890, 8901, 9012, 123]
```

2.1 Experiment 0: Special Experiment

Problem Statement: Obtain the total transport and storage cost on stage with 100 packages and a portion 1.2 for the weight transportable by the offers. How long it takes to find the solution in milliseconds (approximately)? Set the seed to 1234, to generate packages and offers. Use the operators, initialization and heuristics chosen with experiments 1 and 2.

Experiment Configuration

```
# Packages: 100, seed: 1234
Proportion: 1.2, seed: 1234
Algorithm: Hill Climbing
Heuristic: Cost
Operators: Move & Swap
Initial Solution Generator: C
```

Results:

Initial price: 807.09
Average final price: 792.8
Average execution time: 42 milliseconds

2.2 Experiment 1: Set of operators

Problem Statement: Which set of operators works best for a heuristic function that optimizes the first criterion with a scenario in which the number of packets to send is 100 and the proportion of the transportable weight for bids is 1.2. Use the Hill Climbing algorithm. Choose one of the initialization strategies that you propose.

Experiment Configuration:

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Hill Climbing
Heuristic: Cost Heuristic Function
```

Approach:

All the combinations of operators are tested and the best of them is selected.

Observation:

Exist better combinations of operators.

Hypothesis:

- H_0: All combinations of operators behave the same way.
- H_1: Some combinations of operators behave better than others.

Results:

iter	Initial Sol.	Price			Time		
		Move	Swap	Move & Swap	Move	Swap	Move & Swap
0	796.64	794.9	794.78	794.28	26.0	20.0	35.0
1	932.52	932.52	929.42	929.42	0.0	22.0	35.0
2	864.49	860.21	862.54	862.34	11.0	19.0	32.0
3	892.27	886.02	889.92	884.64	12.0	20.0	70.0
4	1065.6	1051.53	1063.0	1050.75	19.0	21.0	58.0
5	865.43	860.63	863.34	860.49	15.0	13.0	51.0
6	982.63	982.62	981.88	981.88	0.0	12.0	20.0
7	917.76	917.09	915.95	915.52	3.0	12.0	26.0
8	1100.57	1100.49	1099.35	1098.87	2.0	14.0	34.0
9	969.77	969.77	968.0	967.52	1.0	21.0	41.0
Mean	938.77	935.57	936.82	934.57	8.0	17.0	40.0
Stdev	93.62	93.05	93.76	92.54	9.12	4.11	15.24

Conclusions: The aim of this experiment is to select the set of operators that works best for optimizing the first criterion. We observe that, out of the three possible combinations of our two operators (move and swap), the best results when it comes to the final state price are obtained by the combination of both ("Move & Swap"). This was expected, given that the other two sets of operators ("Move" and "Swap") are subsets of "Move & Swap", and thus every successor state generated in any of the two operators, can also be generated by "Move & Swap", but not the other way around. It is trivial to see that if a set of operators can generate the same and more successor states than another, it is expected that the algorithm finds an equal or better solution. It is also important to consider the differences in execution time, as a slightly worse optimization with a much lower execution time could be a better choice depending on the application. In this regard we observe that, indeed, the best operators in terms of the final price is also the worst in terms of the execution time. Even so, we consider that the difference in time are not that substantial.
Thus, we conclude that H_1 is correct and we select **"Move & Swap"** as our operator for the experiments from now onwards.

2.3 Experiment 2: Initial solution strategy

Problem Statement: Which initial solution build strategy works best for the heuristic function used in the previous section, with the same scenario and using the algorithm by Hill Climbing?

Experiment Configuration

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Hill Climbing
Heuristic: Cost
Operators: Move & Swap
```

Approach:
Generate different initial solutions and compare the results.

Observation:
There are better generators than others.

Hypothesis:

- H_0: With all generators we get the same solutions.
- H_1: There exist one generator that gives better solutions.

Results:

iter	Initial Solution			Final Price			Time		
	A	B	C	A	B	**C**	A	B	**C**
0	796.64	1021.86	807.9	794.28	841.58	792.75	50.0	156.0	36.0
1	932.52	1137.89	959.08	929.42	1002.17	929.72	29.0	102.0	86.0
2	864.49	1061.38	901.83	862.34	888.4	851.53	30.0	133.0	70.0
3	892.27	1094.08	943.95	884.64	864.33	850.62	52.0	170.0	210.0
4	1065.6	1240.5	1108.59	1050.74	966.62	1021.53	77.0	261.0	207.0
5	865.43	1059.36	870.29	860.49	819.36	854.61	56.0	204.0	93.0
6	982.63	1196.35	1005.19	981.88	1055.41	967.93	20.0	120.0	120.0
7	917.76	1120.76	942.13	915.52	982.04	901.11	27.0	136.0	158.0
8	1100.57	1382.06	1128.74	1098.87	1066.24	1079.03	37.0	211.0	198.0
9	969.78	1195.94	992.95	967.52	1031.92	941.56	42.0	119.0	216.0
Mean	938.769	1151.018	966.065	934.57	951.77	919.04	42.0	161.0	139.0
Stdev	93.62	106.93	99.13	92.54	91.45	87.41	17.10	50.32	66.74

Conclusions: The aim of this experiment is to select the initial state generator that works best for optimizing the first criterion. Out of the three implemented generators, which are explained in more detail in section 1.5, we observe that, counterintuitively, the best results when it comes to optimizing the first criterion are obtained by the **Generator C: Bad Initialization**, instead of **Generator A**, which is implemented to optimize the price. In fact, **Generator A**'s initial solution is better, but it does not optimize better from there onwards. One of the reasons that can explain these results is it might be better for the Hill Climbing to optimize starting from a worse state than from a considerabily optimized one, which could be initialized near a local minimum in a biased manner, jeopardazing the optimization.

As in the previous experiment, it is also important to consider the differences in execution time. We observe that the **Generator C** does not have the highest execution times, as the initial solutions obtained by **Generator B**, which are optimized for happiness, are worse, and consequently the algorithm has to walk a longer path to find the solution. And even though the average execution time for **Generator C** is 4 times higher than that of **Generator A**, we consider that this difference is worth trading for the best and most unbiased results, we select **Generator C: Bad initializaion** as our initial state generator for the experiments from now onwards.

2.4 Experiment 3: Simulated Annealing parameters

Problem Statement: Decide the parameters that give the best results for the Simulated Annealing with the same scenario, using the same heuristic function and the operators and the solution generation strategy chosen in the previous experiments.

Experiment Configuration:

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Simulated Annealing
Heuristic: Cost
Operators: Move & Swap
Initial Solution Generator: C
```

Approach: Several solutions are generated and the parameters are adjusted based on the observed values.

Observation: There is a better combination of Simulated annealing parameters that obtain better results.

Hypothesis:

- H_0: With all the parameters combinations we obtain the same results.
- H_1: There exist a parameter combination with which we get better results.

Results:

id	Steps	Stiter	k	λ	price	time
290	10000	500	1	0.010	921.42	390.3
39	10000	2	1	0.001	922.73	251.9
255	10000	200	1	0.001	923.13	387.4
219	10000	100	1	0.001	924.02	250.1
147	10000	20	1	0.001	924.37	324.5
291	10000	500	1	0.001	924.37	464.8
289	10000	500	1	0.100	924.44	95.1
182	10000	50	1	0.010	924.45	468.0
253	10000	200	1	0.100	924.65	99.1
7	10000	1	5	0.100	924.6590	86.4

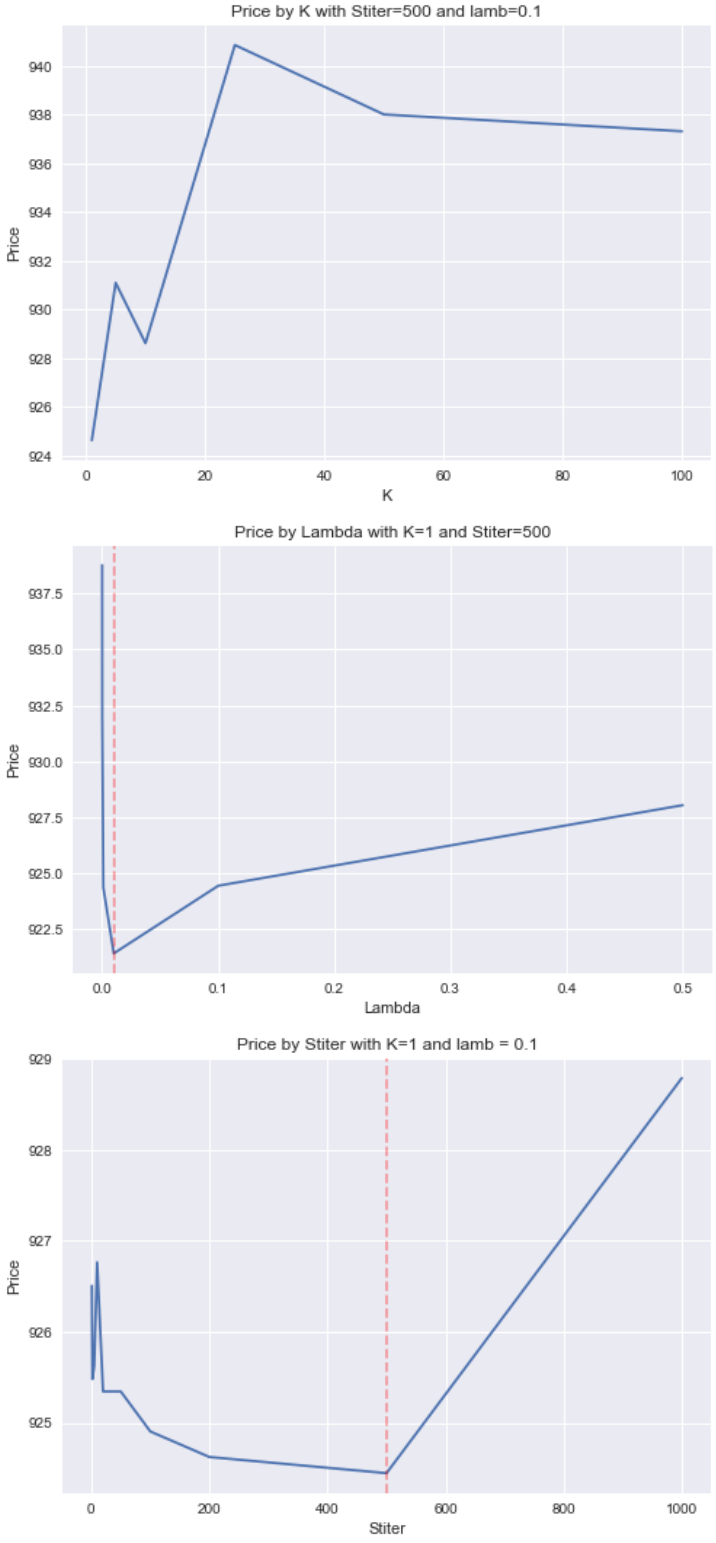


Figure 1: Price values as a function of k, λ and `Stiter`

Conclusions: The aim of this experiment is to select the best parameters for the Simulated Annealing algorithm. In order to obtain the best parameters' values, we have computed the execution time and final price for a large combinatorial sample of the parameters. What we observe is that there exist some combinations of parameters that seem to work generally better than the rest, both considering execution time and final price. In the above table we can see the parameters corresponding the top 10 average final price values of an experiment.
After multiple executions and analyzing their corresponding parameters plots, as the ones in Figure 1, we have decided to use $k = 1$, $\lambda = 1e - 3$ and different values of `Stiter` and `Steps` for each of the different experiments.

2.5 Experiment 4: Runtime evolution

Problem Statement: Given the scenario of the previous sections, study the evolution time to find the solution as a function of the number of packages and the proportion of transportable weight.

- Set the number of packages to 100 and go increasing the proportion of the weight transportable from 1, 2 from 0, 2 to 0, 2 until we see the trend.

Experiment Configuration:

```
Algorithm: Hill Climbing
Heuristic: Cost
Operators: Move & Swap
Initial Solution Generator: C
```

Approach:
Several solutions are computed for different proportion values, and the evolution of execution time is studied based on the observed values.

Observation:
There exists significant positive correlation between the proportion parameter and the execution time.

Hypothesis:

- H_0: The execution time is not correlated to the proportion parameter.
- H_1: The execution time increases is somehow correlated to the proportion parameter.

Results:

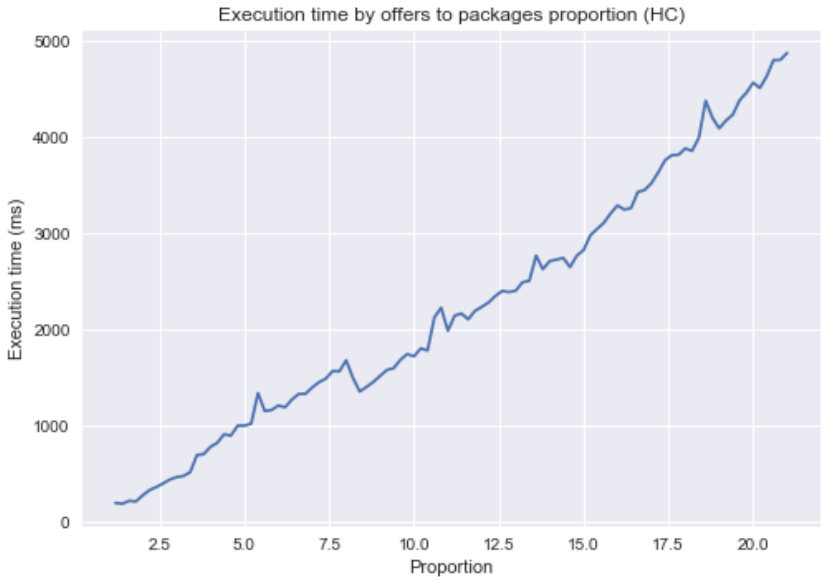


Figure 2: Execution time by the proportion parameter value, for the Hill Climbing algorithm.

Conclusions: As it can be seen in Figure 2, there exists a clear positive correlation between the execution time and the proportion parameter. We can also deduce (with certain confidence, given that the sample space is significant enough), that there exists a linear relationship between both variables, that is, that time can be explained by a linear function on proportion:

$$t = ap + b, \text{where } t \text{ is execution time and } p \text{ is proportion} \tag{1}$$

- Set the weight proportion to 1, 2 and go increasing the number of packages from 100 of 50 in 50 until we see the trend.

Experiment Configuration:

```
Algorithm: Hill Climbing
Heuristic: Cost
Operators: Move & Swap
Initial Solution Generator: C
```

Approach:

Several solutions are computed for different number of packages, and the evolution of execution time is studied based on the observed values.

Observation:

There exists significant positive correlation between the number of packages and the execution time.

Hypothesis:

- H_0: The execution time is not correlated to the number of packages.
- H_1: The execution time increases is somehow correlated to the number of packages.

Results:

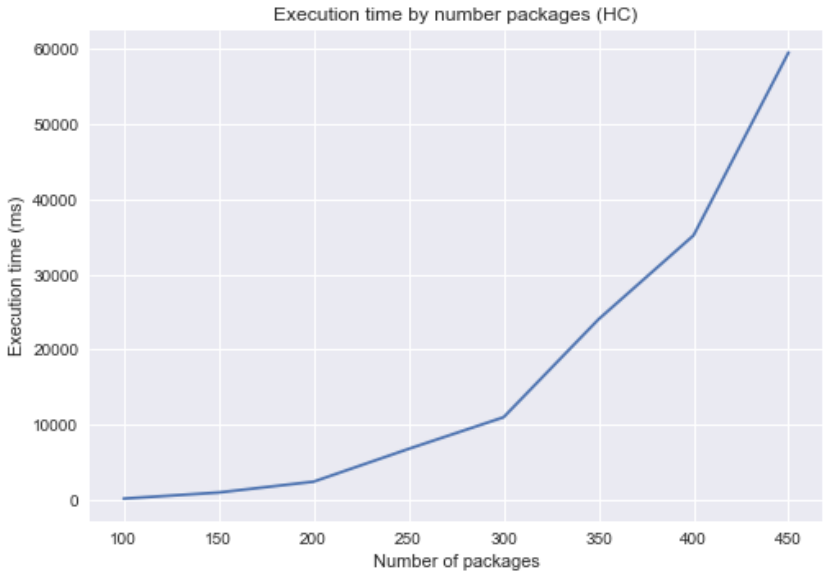


Figure 3: Execution time by the number of packages, for the Hill Climbing algorithm.

Conclusions: As it can be seen in Figure 3, there exists a clear positive correlation between the execution time and the proportion parameter. We can also deduce (with certain confidence, given that the sample space is significant enough), that relationship between both variables is exponential, that is, that execution time can be explained by an exponential function on the number of packages:

$$t = N^a, \text{where } t \text{ is execution time and } N \text{ is the number of packages} \tag{2}$$

2.6 Experiment 5: Price evolution

Problem Statement: Analyzing the results of the experiment in which the proportion of offers is increased, what is the behavior of the cost of transport and storage? Is it worth increasing the number of offers?

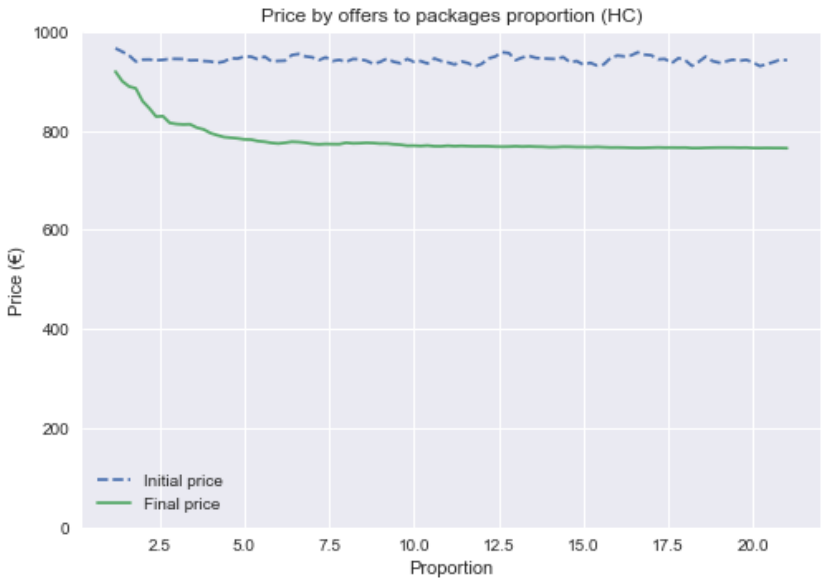


Figure 4: Final price in € by the proportion parameter value, for the Hill Climbing algorithm.

Conclusions: As it can be seen in Figure 4, the initial solution does not get better nor worse with an increment of the proportion parameter, while on the other hand the price seems to get lower (which is obviously better) along with a higher proportion, up to a certain point (in this case it is around $p = 7.5$) where the final solution obtained has reached its maximum even when increasing the proportion value.

This was partially what we could expect. On the one hand, a better solution with a higher proportion value seems logical, as a higher number of offers implies there will be at least the same or more cheap offers, with at least the same or more capacity, consequently reducing the total price of the assignments. But on the other hand, the initial solution does not seem to improve with a higher proportion value: but if you think about it, we are using the **Generator C: Bad initialization**, and thus a larger amount of offers does not improve the initial solution, as there is a larger amount of both good and bad offers and, given that the generator does not distinguish between them (that is why we called it bad initialization, by the way ;)), the initialization does not improve nor worsen consistently.

2.7 Experiment 6: Heuristic happiness function analysis

Problem Statement: Study how the happiness of users affects the cost of transport and storage. We will assume that this happiness is calculated as the sum of the differences between the minimum arrival time of the package indicated by its priority and the number of days it has really taken to arrive when this quantity is positive. Given the scenario of the first section, estimate how transport and storage costs and the execution time vary using Hill Climbing by changing the weighting of the heuristic function. Think and justify how you introduce this element in the heuristic function and how you do the exploration of its weighting in the function.

Experiment Configuration:

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Hill Climbing
Heuristic: Cost & Happiness
Operators: Move & Swap
Initial Solution Generator: C
```

Approach:
Generate different solutions by weighting (using different values) the happiness value of the heuristic.

Observation:
If we assign a greater weight to happiness factor, the packages will be delivered in the shortest possible days, ignoring the offers' prices and consequently it will cause a generalized increase on the cost of the delivery.

Hypothesis:

- H_0: The cost of transportation and storage will not vary as a function of the happiness weight.
- H_1: The price will increase or decrease significantly as a function of the happiness weight.

Results:

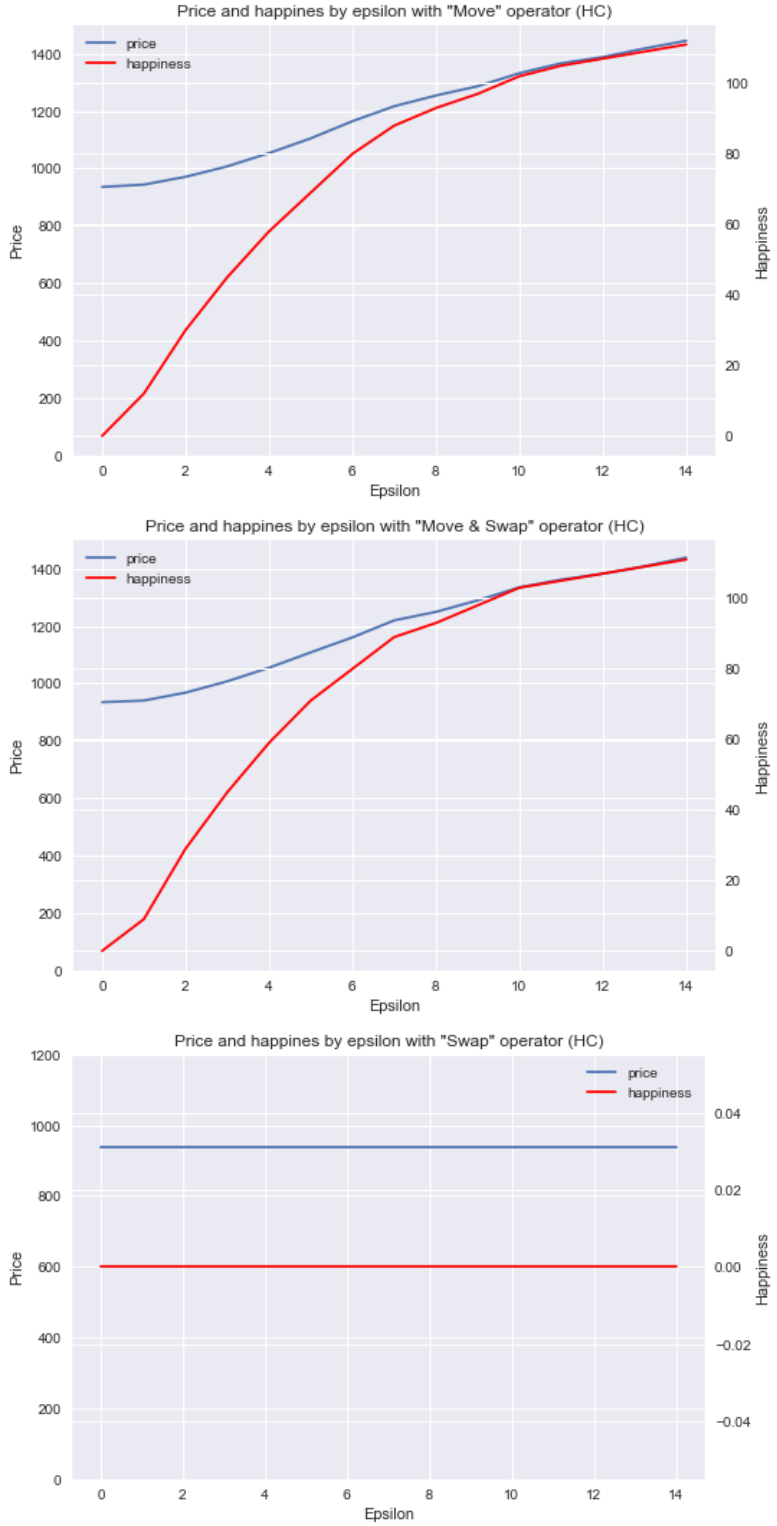


Figure 5: Price (left axis) and happiness (right axis) evolution by epsilon value for each of the operators.

Conclusions: As it can be seen in Figure 5, both happiness and price increase with a higher epsilon value or, in other words, the happiness gets better and the price gets worse. This is what we could expect, as a higher epsilon value implies giving more importance to the happiness results while giving less to the price, and consequently we would expect the solution to move away from the price optimization while approaching the optimal happiness solution.
It is the same case for all operators, as the difference in successor states might affect the exact final solution but has nothing to do with the tendency, which is always guided by the epsilon value.

2.8 Experiment 7: Simulated Annealing

Problem Statement: Repeat the experiments with Simulated Annealing and compare the results. Are there differences in trends that you observe between the two algorithms?

2.8.1 Experiment 7.1

Experiment Configuration:

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Simulated Annealing
Heuristic: Cost
```

Approach:
All the combinations of operators are tested and the best of them is selected.

Observation:
There are combinations of operators better than others.

Hypothesis:

- H_0: All combinations of operators behave in the same way
- H_1: There exist one combination of operators better than the others.

Results:

iter	Initial Sol.	Price			Time		
		Move	Swap	Move & Swap	Move	Swap	Move & Swap
0	796.64	796.46	795.16	796.25	160.0	479.0	176.0
1	932.52	932.52	929.48	931.46	141.0	431.0	229.0
2	864.49	861.92	862.8	862.99	137.0	518.0	231.0
3	892.27	888.5	889.77	887.01	145.0	387.0	216.0
4	1065.6	1054.46	1063.08	1059.22	164.0	536.0	270.0
5	865.43	862.43	863.39	862.03	137.0	281.0	206.0
6	982.62	982.62	982.17	982.62	161.0	777.0	259.0
7	917.765	917.69	915.05	916.43	159.0	718.0	273.0
8	1100.57	1100.57	1099.35	1100.29	155.0	562.0	228.0
9	969.77	969.77	968.18	968.79	176.0	703.0	300.0
Mean	938.76	936.69	936.84	936.70	153.5	539.2	238.8
Stdev	93.62	92.73	93.73	93.42	12.99	156.87	36.68

Conclusions: The aim of this experiment is to select the set of operators that works best for optimizing the first criterion. We observe that, out of the three possible combinations of our two operators (move and swap), the best results when it comes to the final state price are obtained by the combination of both ("Move & Swap"). This was expected, given that the other two sets of operators ("Move" and "Swap") are subsets of "Move & Swap", and thus every successor state generated in any of the two operators, can also be generated by "Move & Swap", but not the other way around. It is trivial to see that if a set of operators can generate the same and more successor states than another, it is expected that the algorithm finds an equal or better solution. It is also important to consider the differences in execution time, as a slightly worse optimization with a much lower execution time could be a better choice depending on the application. In this regard we observe that, indeed, the best operators in terms of the final price is also the second best in terms of the execution time. Even so, we consider that the difference with operator "Move" is not that substantial, and even more considering that the difference with the slowest operator "Swap" is comparatively much higher.

Thus, we conclude that H_1 is correct and we select **"Move & Swap"** as our operator for the experiments from now onwards, the same selection than with the Hill Climbing experiments.

2.8.2 Experiment 7.2

Experiment Configuration

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Simulated Annealing
Heuristic: Cost Heuristic Function
Operators: "Move & Swap"
```

Approach:

Generate different initial solutions and compare the results.

Observation:

There are better generators than others.

Hypothesis:

- H_0: With all generators we get the same solution.
- H_1: There exist one generator better than the others.

Results:

iter	Initial Solution			Final Price			Time		
	A	B	C	A	B	**C**	A	B	**C**
0	796.64	1021.86	807.9	795.28	815.98	797.22	50.0	39.0	30.0
1	932.52	1137.89	959.08	931.27	953.39	927.82	39.0	55.0	22.0
2	864.49	1061.38	901.83	862.95	866.37	858.32	31.0	37.0	36.0
3	892.27	1094.08	943.95	886.94	886.23	854.28	30.0	29.0	19.0
4	1065.6	1240.5	1108.59	1059.33	1075.01	1040.4	35.0	45.0	25.0
5	865.43	1059.36	870.29	862.31	874.18	860.1	25.0	26.0	24.0
6	982.63	1196.35	1005.19	982.62	1007.84	970.95	34.0	35.0	27.0
7	917.76	1120.76	942.13	916.4	932.19	903.02	36.0	39.0	27.0
8	1100.57	1382.06	1128.74	1100.47	1117.72	1088.15	29.0	30.0	22.0
9	969.78	1195.94	992.95	969.08	991.52	953.14	38.0	38.0	24.0
Mean	938.76	1151.01	966.06	936.65	952.04	925.3	34.7	37.3	25.6
Stdev	93.62	106.93	99.13	93.63	96.54	90.25	6.89	8.39	4.78

Conclusions: The aim of this experiment is to select the initial state generator that works best for optimizing the first criterion. Out of the three implemented generators, which are explained in more detail in section 1.5, we observe that, counterintuitively, the best results when it comes to optimizing the first criterion are obtained by the **Generator C**, instead of **Generator A**, which is implemented to optimize the price. In fact, **Generator A**'s initial solution is better, but it does not optimize better from there onwards. One of the reasons that can explain these results is that it might be better for the Simulated Annealing algorithm to optimize starting from a worse state than from a considerabily optimized one, which could be initialized near a local minimum in a biased manner, jeopardazing the optimization.
As in the previous experiment, it is also important to consider the differences in execution time. We observe that the **Generator C** has the lowest execution times, so it is an easy decision to select **Generator C: Bad initializaion** as our initial state generator for the experiments from now onwards, the same that we chose with the Hill Climbing algorithm.

2.8.3 Experiment 7.4

- Set the number of packages to 100 and increase the proportion parameter by intervals of 0.2 until we see the trend.

Experiment Configuration:

```
Algorithm: Simulated Annealing
Heuristic: Cost Heuristic Function
Operators: "Move & Swap"
Initial Solution Generator: C
```

Approach:
Solutions are generated for each proportion and the trend of the results are observed.

Observation:
There exists a direct relationship between the transportable weight proportion and the execution time.

Hypothesis:

- H_0: The execution time is not correlated to the proportion parameter.
- H_1: The execution time increases is somehow correlated to the proportion

Results:

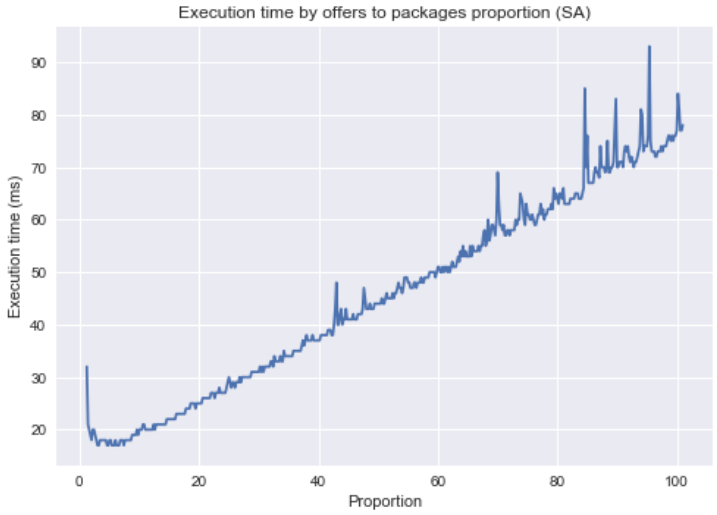


Figure 6: Execution time by the proportion parameter.

Conclusions: As it can be seen in Figure 6, there exists a clear positive correlation between the execution time and the proportion parameter. We can also deduce (with certain confidence, given that the sample space is significant enough), that there exists a linear relationship between both variables, that is, that time can be explained by a linear function on proportion:

$$t = ap + b, \text{where } t \text{ is execution time and } p \text{ is proportion} \tag{3}$$

Even so, there seems to be lots of outliers present in the data. The linear relationship is strongly present, but differently from other parameters where the plot is much smoother, here we can observe quite a few irregularities.

- Set the weight proportion to 1.2 and increase the number of packages from 100 onwards by intervals of 50 until we see the trend.

Experiment Configuration:

```
Algorithm: Simulated Annealing
Heuristic: Cost Heuristic Function
Operators: "Move & Swap"
Initial Solution Generator: C
```

Approach:
Solutions are generated for each proportion and the trend of the results are observed.

Observation:
There exists a direct relationship between the package proportion and the execution time.

Hypothesis:

- H_0: The execution time is not correlated to the number of packages.
- H_1: The execution time increases is somehow correlated to the number of packages.

Results:

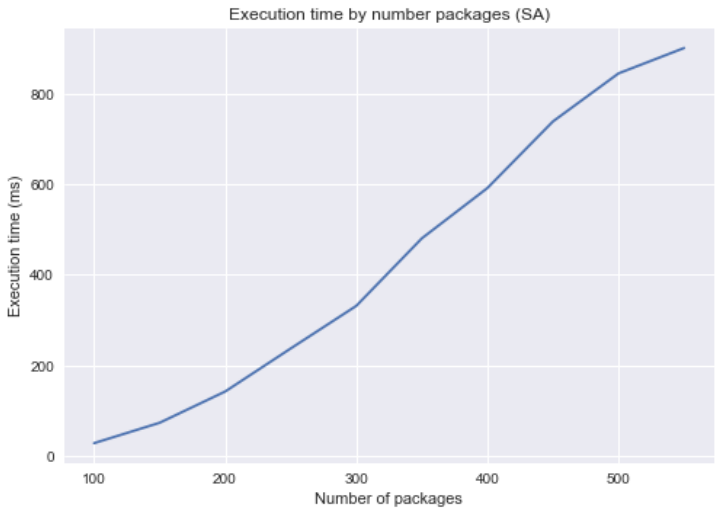


Figure 7

Conclusions: As it can be seen in Figure 7, there exists a clear positive correlation between the execution time and the proportion parameter. It is not clearly distinguishable if the relationship between both variables is linear or exponential, that is, that execution time can be explained by either an exponential function on the number of packages:

$$t = N^a, \text{where } t \text{ is execution time and } N \text{ is the number of packages} \tag{4}$$

Or a linear function:

$$t = aN + b, \text{where } t \text{ is execution time and } N \text{ is the number of packages} \tag{5}$$

2.8.4 Experiment 7.6

Experiment Configuration:

```
# Packages: 100, seed: various
Proportion: 1.2, seed: various
Algorithm: Simulated Annealing
Heuristic: Cost & Happiness
Operators: Move & Swap
Initial Solution Generator: C
```

Approach:
Generate different solutions by weighting (using different values) the happiness value of the heuristic.

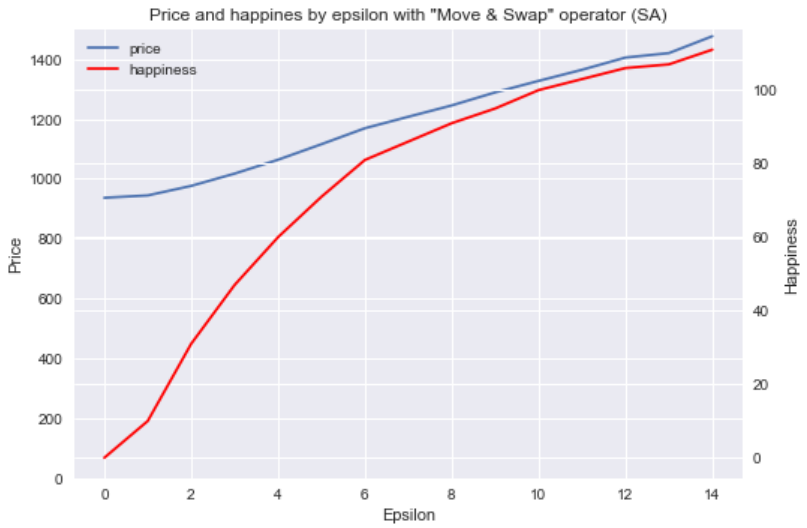
Observation:
If we assign a greater weight to happiness factor, the packages will be delivered in the shortest possible days, which will increase the cost of the delivery.

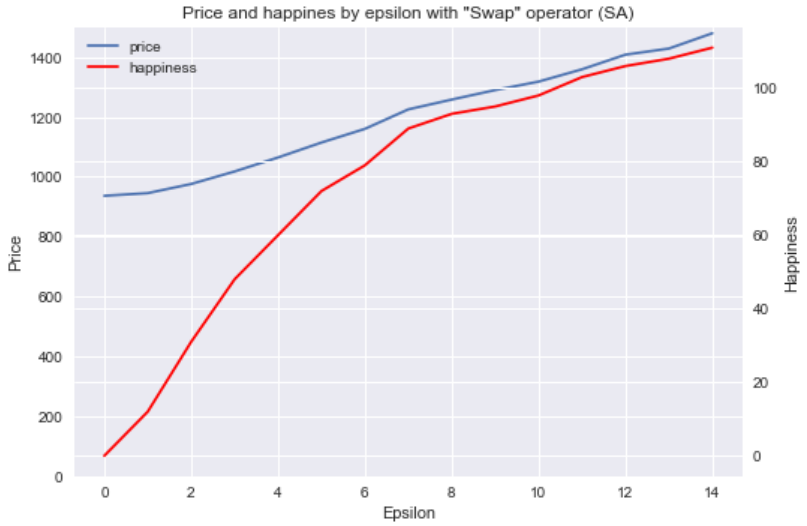
Hypothesis:

- H_0: The cost of transportation and storage will have similar values.
- H_1: The price will increase or decrease considerably.

Results:


Price and happines by epsilon with "Move" operator (SA)
price
happiness
Price
Happiness
Epsilon
0
200
400
600
800
1000
1200
1400
0
20
40
60
80
100
0
2
4
6
8
10
12
14


Price and happines by epsilon with "Move & Swap" operator (SA)
price
happiness
Price
Happiness
Epsilon
0
200
400
600
800
1000
1200
1400
0
20
40
60
80
100
0
2
4
6
8
10
12
14


Price and happines by epsilon with "Swap" operator (SA)
price
happiness
Price
Happiness
Epsilon
0
200
400
600
800
1000
1200
1400
0
20
40
60
80
100
0
2
4
6
8
10
12
14

Conclusions: As it can be seen in Figure 5, both happiness and price increase with a higher epsilon value or, in other words, the happiness gets better and the price gets worse. This is what we could expect, as a higher epsilon value implies giving more importance to the happiness results while giving less to the price, and consequently we would expect the solution to move away from the price optimization while approaching the optimal happiness solution.
It is the same case for all operators, as the difference in successor states might affect the exact final solution but has nothing to do with the tendency, which is always guided by the epsilon value.

2.8.5 Simulated Annealing vs. Hill Climbing

Once the two algorithms have been executed with the same experiments, and taking into consideration that we have used the same seeds to be able to compare results more faithfully, we have reached the following conclusions:

- Simulated Annealing algorithm generally converges faster.

  In the Hill Climbing algorithm we start with an initial solution, then we make a modification and we check if the new solution is better or worse than the previous one. If it is better we accept it, but if it worse we keep the original solution in order to suggest a new change. After a while the algorithm is not able to improve the solution and stops, since it can quickly find a local maximum or local minimum.

  In the Simulated Annealing algorithm the procedure is similar, but we have a certain probability of accepting a solution that is worse than the previous one. Following that idea the algorithm explores some paths even if a priori they do not seem very good. If we follow this method the algorithm keeps jumping all over the place in order to find different solutions, unlike the Hill Climbing algorithm.

- Due to its random nature, the spectrum of obtained solutions is wider.

- A beneficial aspect of Simulated Annealing is that it is much more difficult to get lost in local maximums or minimums (always depending on the used criteria).

In general, the experiments have not shown us much difference in results.

2.9 Experiment 8: Storage cost variation analysis

Problem Statement:

We have assumed a fixed daily storage cost of 0.25€/kg. Without doing any experiments. How would the solutions change if we vary this price up or down?

Resolution:

- > 0.25€/kg: in this case, the algorithm will tend to give preference to the offers that deliver the packages in a shorter amount of time over those with a lower price, since the cost of storage in Azamon is expensive. If the algorithm reduce this delivery time, the happiness would also increase since the packages would often be delivered earlier.
- < 0.25€/kg: in this case, the delivery time would increase given that it is more profitable to have the packages more time in the storage than less time in a more expensive transport, since the storage in Amazon would be cheap. The assignments would tend to ignore the offers' delivery days (up to the point of always fulfilling the restriction of delivering each package on time) while mainly focusing on the offers' prices. This would translate into customers often receiving their packages closer to the estimated time and the company would renounce to the happiness bonus.

In summary, a higher storage price would move the solutions into faster deliveries - thus reducing the time of the packages in the storage and increasing the happiness value - while a lower storage price would allow the algorithm to let the packages rest more days in the storage while choosing the best priced offers - resulting in lower happiness values.

3 Conclusions

Firstly, we must say that it is essential to understand the problem in order to develop an application responsible for its solution: its size, the generation of initial solutions, the requirements and limitations of the problem, etc. These aspects has helped us build the representation of the state of the problem, which we believe is the very base of the project, and afterwards, it has allowed us to define the set of operators that will modify it in order to find alternative solutions. It has also been very important the deep analysis of the behaviour of each of the possible parameter configurations, with the goal of optimizing both the value of the solution itself and others aspects such as the execution time, which is critical for the applications in real cases.

In conclusion, we could say that the best way to understand the performance of an algorithm is by executing and trying it. We have profoundly understood, through trial and error, the Hill Climbing and Simulated Annealing algorithms. In a lower level, we have observed that different initial states lead to different solutions and different ways of arriving to those solutions, and different parameters may lead to faster solutions, or might trade some kind of value for another. It has also been interesting to analyze the obtained results for each of the criterions or for the combination of both.

To sum up, regarding the work organization, it has been beneficial for us to distribute the tasks and establish internal deadlines, with their corresponding meetings, to closely follow the progress in each of the steps, and be able to share the knowledge of the three of us in each of the tasks. On the downside, we regret not having more time to clean some parts of the code and document it, but we believe this was not critical for the purpose of this project, which we believe was learning about the concepts we are treating with and not the implementation of a software application in itself.

Finally, we would like to thank the teachers Sergio Álvarez and Javier Béjar for the effort dedicated in theory and lab classes, correspondingly, which has helped us present this as our final work.