Minimum Weight Vertex Cover with Iterated Local Search.

Laboratorio Intelligenza Artificiale (LM-18)
Università degli Studi di Catania - A.A 2021/2022

Martucci Giovanni

September 14, 2022

1 Introduction

The aim of the project was to study and implement the Iterated Local Search to solve the problem of the Minimum Weight Vertex Cover belonging to the group of problems NP-Complete, therefore unsolvable with deterministic algorithms. A local search starts from a valid solution and iteratively builds a new solution trying to improve it until reaching a global optimal. During the execution of the algorithm the solution may end up in a very good local. Thanks to the perturbation operators and the acceptance criteria, the solution of local optimal is then removed by exploring new areas of the research space.

The entire project is available at the following Github repository: https://github.com/Giovanni-Martucci/MWVC-ILS.

2 Implementing rules

2.1 Generation of initial solution

The first phase of the algorithm involves the generation of the initial solution following a Greedy logic. The first version of the solution provided for the insertion of the vertex with lower weight for each arc; subsequently, as shown in the algorithm in figure 1, a queue is used to track nodes that must be entered to meet the MWVC constraint. This priority queue is sorted by the degree ratio of each node divided by its weight. (degree[i]/weight[i]) so as to get the nodes with large degree and low weight above the queue.

Algorithm 1 `Greedy Initial Solution`

```
Require: graph, solution
  E = outsideEdges(solution)
  queue = buildingQueue(E, graph)
  while E until 0 do
     topNode = queue.pop()
     Add topNode to solution
     Update E \edges(topNode)
     Update queue
  end while
```

Then some useful parameters have been initialized and set for the convergence of the algorithm: the *Epsilon*, which will be used in the Acceptance Criteria discussed later and the *maximum number of bits* that will be perturbed in the current solution. In both cases the two parameters have an initial value and as the algorithm refines the solution they gradually decrease in such a way as to not allow a large change of the solution when you are close to the global optimal. Several solutions have been tried regarding the calibration of these parameters. In the source code it is possible to

view the different implementations made before arriving at the final solution for both parameters, shown in paragraph 2.2.

The solution is represented as a list of values 1 or 0 indicating the presence or not of the i-th node in the final solution.

2.2 Parameters scheduling

Below are the final solutions adopted for the parameters just discussed.

2.2.1 Eps-scheduling

Algorithm 2 Eps-scheduling

```
Require: currentWeight, bestWeight, eps, minEps, iterWithoutImprovs,
  maxIterWithoutImprovs
  if currentWeight < bestWeight then
     epsNew = epsNew - ((bestWeights - currentWeights) * 2)
     if epsNew < minEps then return minEps
     end if
  else
     epsNew = eps
     if    currentIter mod 6000 == 0 and iterWithoutImprovs > maxIterWithoutImprovs
  then
        less = epsNew / 3
        epsNew = epsNew - less
     end if
     if epsNew < 20 then
        epsNew = 100 * rand(1,4)
     end if
  end if return epsNew
```

As shown in figure 2 the value of Epsilon decreases with every improvement of the solution in such a way as to decrease the range of acceptance of the future iterations. Subsequently, with each number of fixed iterations X (in this case 6000), if the number of iterations without improvement exceeds the maximum limit (fixed at a certain arbitrary value), the value of Epsilon decreases by 1/3; Finally, if Epsilon becomes smaller than 20 it's returned to a value equal to 100 * random value from 1 to 4.

2.2.2 Num-elem-to-change

This parameter is used to iteratively decide how many values of the current solution you need to change. As the solution improves the following parameter decreases. Finally, if the number of iterations without improvements reaches a large arbitrary value (fixed at 150000) and the current number of iterations is less than 80% of the maximum of available iterations, the number of items to change increases, so try to get out of a great local situation in case you are there.

Algorithm 3 `changeNumberElem`

```
Require: eps, minEps, maxEps, pertMax, pertMin
  improvs = 1.0 - ((eps - minEps) / (maxEps - minEps))
  newNumElemsToChange = pertMax - (improvs * (pertMax - pertMin))
  if iterWithoutImprovements mod 15000 == 0 and currentIter < (maxIter - (maxIter
  / 20)) then
     newNumElemsToChange = numTotalNodes / 2
  end if return newNumElemsToChange
```

2.3 Perturbation of the solution

The general idea of perturbation is to modify the bits present in the solution at each iteration by applying a perturbation on the selected nodes. This procedure will allow you to move between the space of the solutions. It should be noted that a large perturbation creates a large displacement in the space of the solutions, whereas a small perturbation shifts the current solution into the solutions close to them. At first, in the first strategy developed only one random node is subtracted from the solution and another one is added, while if all nodes are present in the solution, only one random node is subtracted. Below is the pseudo-code of the final logic adopted for the perturbation, in which it comes prefixed the number of bits to change:

Algorithm 4 `Perturbation`

```
Require: elemToChange, solution
  for 1 to elemToChange do
     v = randomNode(V)
     Insert or Remove v inside solution
  end for return solution
```

The number of changes that will be made is defined by the parameter taken in input "numElemToChange". This parameter gradually decreases when you are close to the global optimum. After having carried out the perturbation, we move on to the Local Search phase.

2.4 Validity of the solution

After the solution has been perturbated, the solution is checked for validity. This is because a solution is valid if for each arc of the graph there is at least one of the two vertices inside the final solution. This check is shown in Figure 5:

Algorithm 5 `Valid Solution`

```
Require: solution
  for u in V do
     for (u,v) in adjList(u) do
        if  u ∉ Solution and v ∉ solution then return False
        end if
     end for
  end for return True
```

As the figure shows, scrolling through all the adjacency lists for each node, at least one of the vertex nodes for each arc must be contained in the final solution.

2.5 Local search

Before starting with this phase, the validity of the solution is always checked, using the algorithm discussed above. If the solution is not valid, scroll through the list of arcs whose both vertices do not belong to the solution and insert one of the two nodes inside the solution, by comparing the two based on the highest degree number and the lowest weight number. You can see the code in image 6:

Algorithm 6 `Local Search`

```
for 1,2,..., numSwaps do
    v1 = randomNode(solution)
    if v1 ∈ Solution then
        v2 = randomNode(N(v1) \solution)
        if weight(v2) < weight(v1) then
            neighbor = solution - v1 + v2)
        end if
    else
        v2 = randomNode(N(v1) ∈ solution)
        if weight(v1) < weight(v2) then
            neighbor = solution - v2 + v1
        end if
    end if
    if neighbor is valid solution and weight(neighbor) < weight(currentSolution)
then
        currentSolution = neighbor
    end if
end for return currentSolution
```

The implemented local search allows to complete the perturbed solution. You select a random node from the current solution and then a random neighbor is selected from its list of adjacencies that will replace it. If the neighbour will not bring any improvement then it will be passed to the selection of another node to replace, proceeding so iteratively. If instead the randomly selected node to be replaced is not part of the solution then you will search from its list of neighbors a node contained in the solution to be deleted to make room for the one initially selected.

2.6 Acceptance Criteria

Acceptance criteria are necessary to assess the quality of the solution obtained from a local search before it is perturbed and used in the next iteration. In detail a solution is accepted if:

- is better than the previous solution;

- the weight of the current solution is less than the optimal solution added to an Epsilon value and the number of iterations without improvement exceeds the maximum threshold of iterations without improvement;

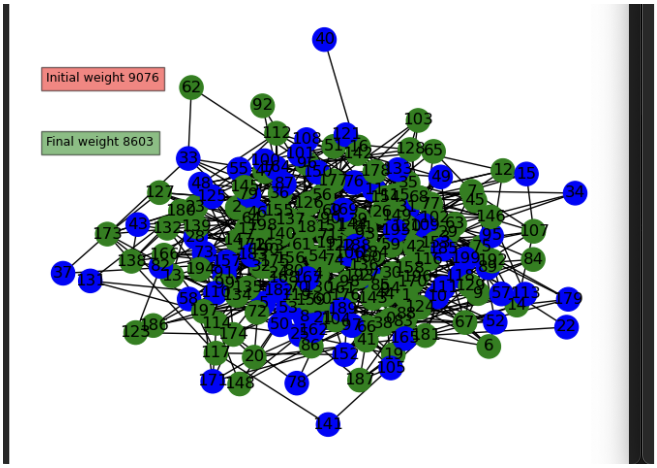
Algorithm 7 `Acceptance Criteria`

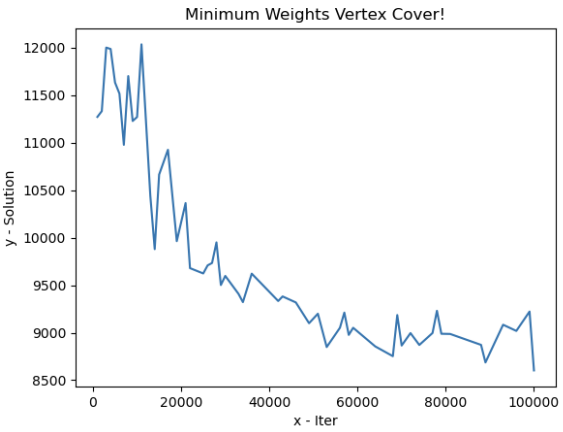
```
if currentWeight < prevWeight or currentWeight < (bestWeight + eps) and
iterWithoutImprovs > maxIterWithoutImprovs then return True
else return False
end if
```

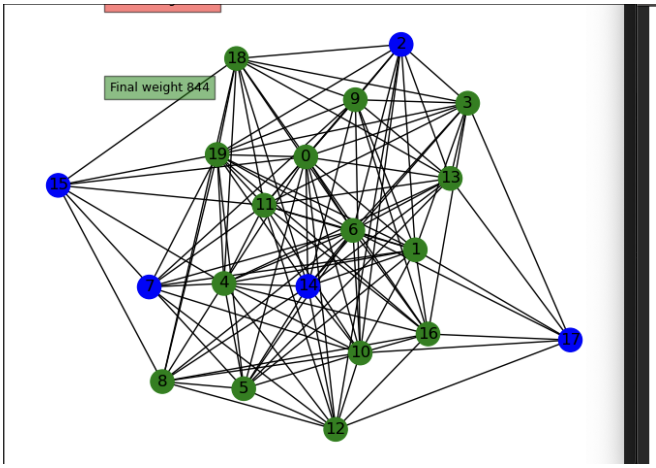
3 Benchmarks

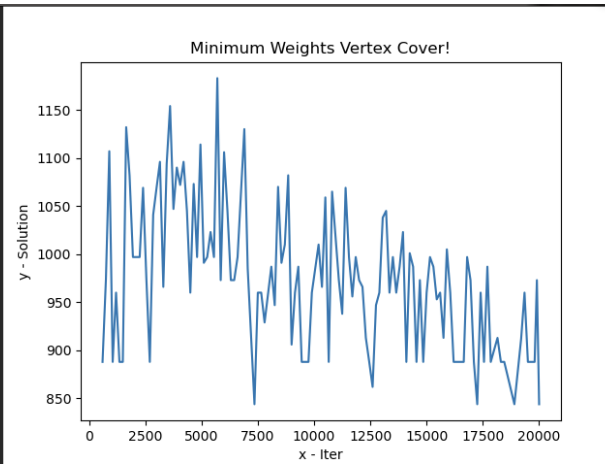
The algorithm developed is able to find an excellent solution for each input, the introduction of the parameter *Epsilon* and *numElemToChange* has helped to achieve a fairly comprehensive neighborhood discovery greatly improving the final results.

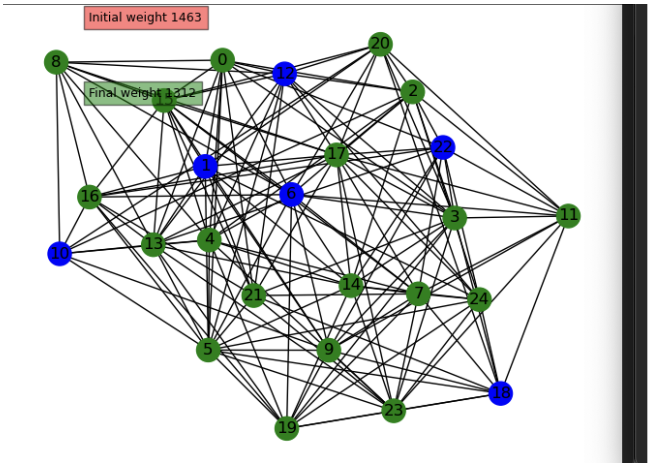
The following are the benchmarks:

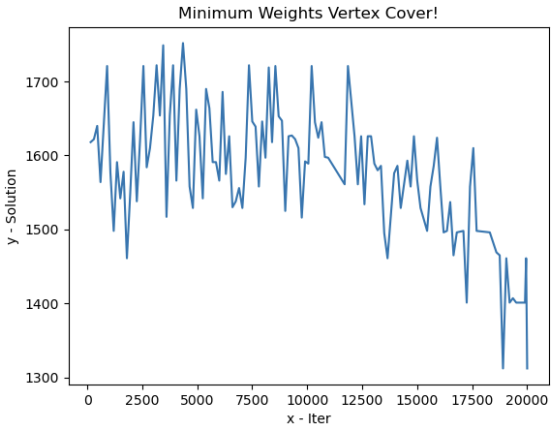












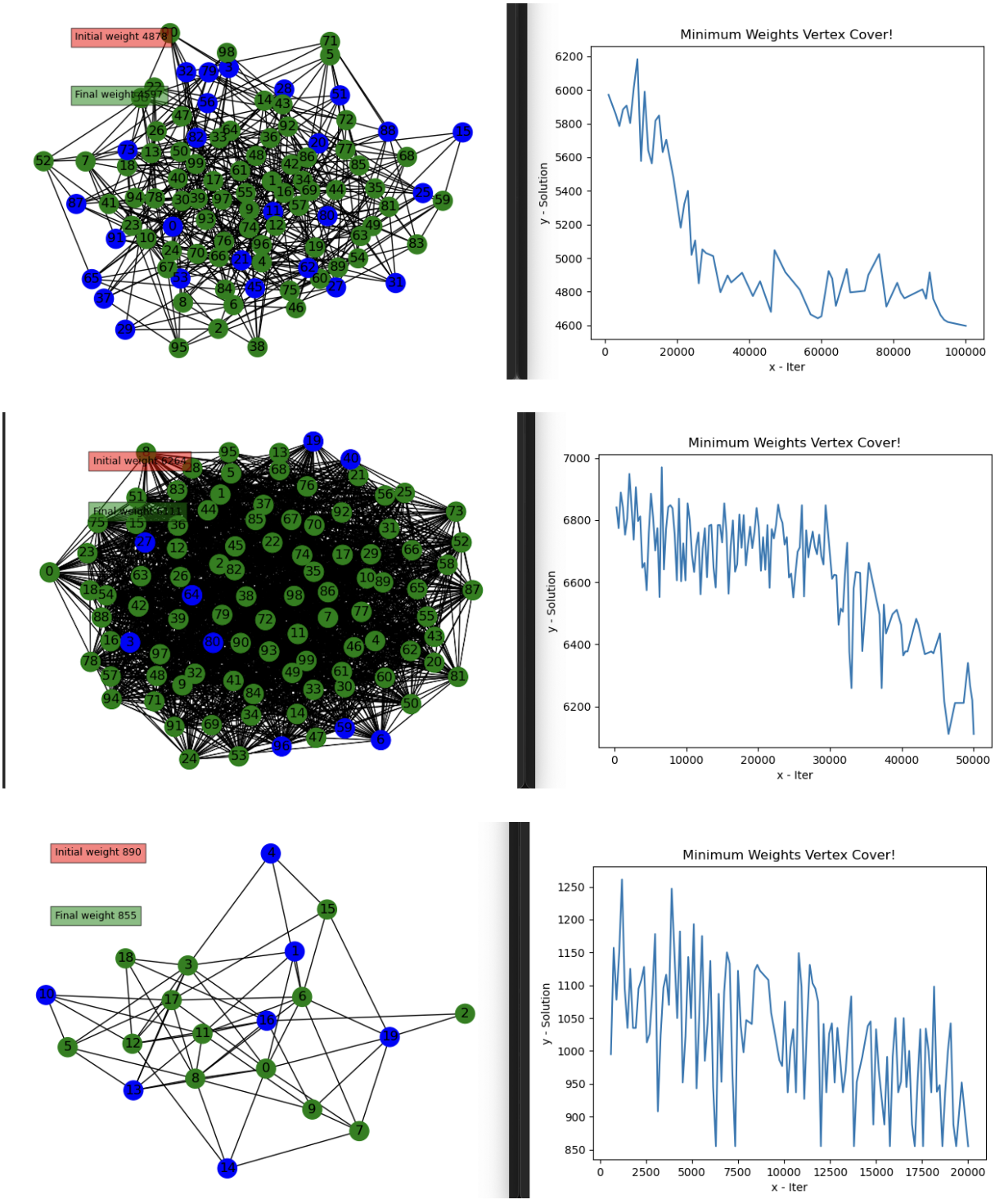


Figure 1: Grafici di convergenza

Input name	Initial Weight	Final Weight	Final Solution	Final Eps	Run Time
vc_20_60_01.txt	861	774	[1, 2, 3,..., 18]	127,5555556	2,669713974
vc_20_60_02.txt	1026	938	[0, 3, 6,...,19]	152	2,684123039
vc_20_60_03.txt	796	730	[0, 3, 5,...,19]	117,9259259	2,296877146
vc_20_60_04.txt	819	769	[0, 1, 2,...,18]	121,3333333	2,56937027
vc_20_60_05.txt	926	871	[1, 3, 4,..., 19]	137,1851852	2,281625748
vc_20_60_06.txt	890	855	[0, 2, 3,...,18]	131,8518519	2,445345163
vc_20_60_07.txt	1006	984	[0, 1, 6,...,18]	149,037037	2,359009027
vc_20_60_08.txt	895	867	[2, 4, 5,...,19]	132,5925926	2,198531866
vc_20_60_09.txt	1002	980	[1, 3, 5,...,19]	148,4444444	2,45902276
vc_20_60_10.txt	875	875	[3, 4, 6,...,19]	129,6296296	2,339637041
vc_20_120_01.txt	910	844	[0, 1, 3,...,19]	134,8148148	3,625911951
vc_20_120_02.txt	1086	1009	[0, 1, 3,...,19]	160,8888889	3,825074911
vc_20_120_03.txt	1039	994	[0, 1, 2,...,19]	153,9259259	3,100630999
vc_20_120_04.txt	1097	1050	[0, 1, 2,...,19]	162,5185185	3,225299835
vc_20_120_05.txt	1029	997	[1, 3, 4,...,19]	152,4444444	3,18448782
vc_20_120_06.txt	961	961	[0, 1, 2,...,19]	142,3703704	3,598522902
vc_20_120_07.txt	991	991	[0, 1, 2,...,19]	146,8148148	4,857724905
vc_20_120_08.txt	1170	1142	[1, 2, 3,...,19]	173,3333333	3,927416086
vc_20_120_09.txt	1296	1261	[0, 1, 2,...,19]	192	3,158335924
vc_20_120_10.txt	1148	1133	[0, 2, 4,...,19]	170,0740741	3,382470131
vc_25_150_01.txt	1463	1312	[0, 2, 3,..., 24]	216,7407407	4,598255157
vc_25_150_02.txt	1234	1132	[1, 2, 3,...,24]	182,8148148	4,611696959
vc_25_150_03.txt	1416	1352	[0, 1, 2,...,24]	209,7777778	4,227385044
vc_25_150_04.txt	1519	1425	[0, 1, 2,...,24]	225,037037	4,095405102
vc_25_150_05.txt	1390	1307	[0, 2, 3,...,24]	205,9259259	4,551862001
vc_25_150_06.txt	1328	1255	[0, 1, 2,...,22]	196,7407407	3,764107227
vc_25_150_07.txt	1524	1511	[0, 1, 4,...,24]	225,7777778	4,488750696
vc_25_150_08.txt	1151	1151	[0, 1, 2,...,24]	170,5185185	4,198129177
vc_25_150_09.txt	1248	1248	[1, 2, 3,...,24]	184,8888889	4,251717091
vc_25_150_10.txt	1061	1061	[0, 1, 2,...,23]	157,1851852	4,223310947
vc_100_500_01.txt	4878	4597	[1, 2, 4,...,99]	68,64529305	133,8012891
vc_100_500_02.txt	5273	5033	[1, 3, 4,...,99]	43,84079923	110,9794841
vc_100_500_03.txt	4592	4424	[1, 3, 4,...,99]	39,26773973	83,4767592
vc_100_500_04.txt	4980	4636	[2, 6, 8,...,99]	67,07962836	91,55720091
vc_100_500_05.txt	5057	4885	[1, 3, 4,...,97]	60,93683859	106,0464609
vc_100_500_06.txt	5069	4860	[0, 2, 3,...,99]	98,89224204	67,52395678
vc_100_500_07.txt	5082	4694	[0, 1, 2,...,99]	99,14586191	41,51749468
vc_100_500_08.txt	4814	4673	[0, 1, 2,...,98]	93,91739064	47,34881878
vc_100_500_09.txt	4722	4657	[1, 2, 3,...,98]	92,1225423	53,81567311
vc_100_500_10.txt	4601	4515	[0, 1, 2,...,98]	89,76192654	46,76840401

Input name	Initial Weight	Final Weight	Final Solution	Final Eps	Run Time
vc_100_2000_01.txt	6264	6111	[0, 1, 2,...99]	122,2057613	194,4893651
vc_100_2000_02.txt	6023	5992	[0, 1, 2,...99]	117,504039	431,6642709
vc_100_2000_03.txt	5830	5738	[0, 1, 2,...,99]	113,7387593	261,161907
vc_100_2000_04.txt	6145	6058	[0, 1, 2,...,99]	72,10912012	477,8304288
vc_100_2000_05.txt	6471	6330	[0, 1, 3,...,99]	53,45673204	278,957222
vc_100_2000_06.txt	5960	5904	[0, 1, 2,...,99]	174,4124371	145,7328162
vc_100_2000_07.txt	6504	6354	[1, 2, 3,...,99]	126,8879744	136,9901099
vc_100_2000_08.txt	6360	6309	[0, 1, 2,...,99]	124,0786465	168,2622838
vc_100_2000_09.txt	6001	5985	[0, 1, 2,...,99]	117,0748362	157,6364491
vc_100_2000_10.txt	6287	6240	[2, 3, 4,...,99]	122,6544734	239,845624
vc_200_750_01.txt	9076	8603	[0, 1, 2,...198]	229,1767746	223,5511329
vc_200_750_02.txt	8696	8384	[0, 3, 4,...,199]	84,50493346	239,3247552
vc_200_750_03.txt	8952	8654	[0, 1, 2,...,199]	261,9698217	132,4322062
vc_200_750_04.txt	8466	8074	[1, 2, 3,...,199]	69,60856611	211,2878647
vc_200_750_05.txt	9173	9006	[0, 1, 2,...,199]	151,3027671	223,7731762
vc_200_750_06.txt	8594	8265	[0, 1, 3,...,197]	33,11846847	176,7502
vc_200_750_07.txt	8005	7701	[1, 3, 4,...,199]	156,1713153	135,1315081
vc_200_750_08.txt	8712	8528	[0, 3, 4,...,199]	57,91196203	240,9734781
vc_200_750_09.txt	8830	8661	[0, 8, 10,...,199]	157,7677004	248,000886
vc_200_750_10.txt	8708	8459	[1, 2, 4,...,199]	169,8862978	133,300956
vc_200_3000_01.txt	11895	11728	[0, 2, 3,...,199]	197,980836	710,657182
vc_200_3000_02.txt	11576	11346	[0, 1, 2,...,199]	131,1731888	623,197376
vc_200_3000_03.txt	12227	12030	[2, 3, 4,...,199]	20,94171716	678,0509641
vc_200_3000_04.txt	13136	12894	[0, 2, 3,...,199]	175,0668067	690,9683321
vc_200_3000_05.txt	11950	11858	[0, 1, 2,...,199]	202,771632	696,4376061
vc_200_3000_06.txt	11539	11328	[0, 1, 3,...,199]	66,169563	691,5875759
vc_200_3000_07.txt	11836	11583	[1, 2, 3,...,199]	214,2054142	659,8127627
vc_200_3000_08.txt	12083	11868	[1, 2, 3,...,199]	231,7402946	679,6931341
vc_200_3000_09.txt	12118	12008	[0, 1, 2,...,199]	200,4280632	692,4992981
vc_200_3000_10.txt	11915	11551	[0, 1, 2,...,198]	168,2991086	689,6815619
vc_800_10000.txt	46155	44649	[0, 2, 3,...,799]	400,4481024	348,5432

4 Conclusion

This project has made possible the acquisition of several practical skills analyzed during the course of Artificial Intelligence and has allowed to become aware of the use and development of solutions based on these algorithms.

The results obtained are quite good since randomness makes it possible to make locally optimal choices, especially thanks to the greedy algorithm that builds a good starting solution. The values of the parameters that have been chosen after many attempts are also quite good. There is definitely room for improvement in the sampling of neighbors, you could think of new selection criteria.