Artificial Intelligence

University of Catania

Minimum weight vertex cover(MWVC) with
Iterated local search

Locicero Giorgio
locicero.giorgio@studium.unict.it

February 18, 2022

1 Introduction

In this research, some methods to estimate the optimal solution to the **Minimum weight vertex cover** problem will be seen. The method used to compute the estimation is **Iterated Local Search**, a method that uses a **Local search** algorithm under the hood in conjunction with additional parameters, a **perturbation operator** that perturbs the solution to get out of local optimum, an **acceptance criterion** that substitutes the current solution if some conditions are met, and a **termination criterion** used to stop the execution of the algorithm.

The choice of Iterated Local Search is quite fitting for the problem at hand because the perturbation operator after the local search could kick out of the candidate solution some non optimal nodes that are full of constraints and insert some nodes of the global solution because some nodes in the candidate solutions, found during the computation, will probably never leave the candidate otherwise.

2 Algorithm

The algorithm itself builds a complex pipeline around : local search; the definition of **neighborhood** for local search and in general for candidates solutions; the introduction of a **perturbation** to modify the current solution and get out of local optimum and a series of parameters used in this steps to tune the algorithm during execution.

A template pseudocode for ILS is seen in 1

Algorithm 1: ILS

```
Data: Graph G, power, rarity, worstSolAccept
Result: Candidate solution to MWVC
solution ← GetInitialSolution(G) ;                    /* valid solution creation */
solution ← LocalSearch(G, solution);
solutionWeight ← Cost(G, solution);
iterWithImprov ← 0;
iterWithoutImprov ← 0;
numberWorstSol ← 0;
while ¬(termination conditions) do
    perturbedSolution ← Perturbation(G, solution,power,rarity);
    perturbedSolution' ← LocalSearch(G, perturbedSolution);
    if acceptance criterion then
        if Cost(G, solution) < Cost(G, perturbedSolution') then
            numberWorstSol ← numberWorstSol + 1;
            iterWithoutImprov ← iterWithoutImprov + 1;
        else
            iterWithImprov ← iterWithImprov
        end
        solution ← perturbedSolution;
        solutionWeight ← cost(G, perturbedSolution);
    else
        iterWithoutImprov ← iterWithoutImprov + 1;
    end
    power ← adjustPower(G, power, iterWithoutImprov, iterWithImprov);
    rarity ← adjustRarity(G, rarity, iterWithoutImprov, iterWithImprov);
    worstSolAccept ← adjustProb(G, worstSolAccept,
                                iterWithoutImprov,
                                iterWithImprov, numberWorstSol);
end
return solution
```

The algorithm first gets a candidate solution with a greedy algorithm(more than 2 versions were implemented, but only 2 will be seen), the algorithm can be seen in 2 and 5. After getting the initial solution (that is also updated with a local search), the solution weight is assigned and this weight will be used in the **acceptance condition**.

The candidate solution is implemented as a Boolean array where:

- **true**: the node identified by the index in the array is in the candidate solution;
- **false**: the node identified by the index in the array is not in the candidate solution.

After the initial initialization, the algorithm starts with the iterative nature of the method. **Termination conditions** used to iterate over candidates solutions are:

- the number of iteration of ILS is lesser than the max number of iteration established as a parameter;
- the number of iterations without improvement is lesser than an upper bound.

Another termination condition defined by the assignment is that the number of objective function eval should not exceed 20000, that is also used as a termination condition of the algorithm.

The next thing to do is to perturb the current optimal solution to get a new candidate, this is done by the *Perturbation* operator defined in 4. After obtaining the candidate from the perturbation, a run of local search is done to get a better solution and reach the local optimum. The local search algorithm is described roughly in 8.

After getting the local optimum of the perturbed solution with local search, the acceptance condition is applied. The **acceptance criterion** is defined on these cases:

- If the perturbed solution after local search is better than the current one, accept the solution and substitute it to the current solution.
- If the perturbed solution after local search is worst, choose at random with probability *probAcceptSolution* to throw it away or substitute it to the solution.

If a bad solution is accepted, two counters are incremented (iterations without improvement and the number of worst solutions accepted), otherwise another counter is incremented(iterations with improvement), this counters will be used to modify some parameters used for the perturbation and the probability of accepting worst candidates. After this, the perturbed candidate solution is assigned to the current solution and the algorithm proceeds(omitting the assignment and the different conditional branch because they are obvious).

After controlling the acceptance condition, the parameters used for the perturbation are updated as shown in (9,10,11). After the termination conditions are met, the algorithm returns the candidate solution found.

Algorithm 2: GetInitialSolution

Data: Graph G
Result: Candidate solution to MWVC
partialSolution $\leftarrow \{\}$;
while $G.edges$ *not covered* **do**
  heap $\leftarrow MaxPriorityQueue()$;
  foreach $edge \in notCovered(G.edges)$ **do**
    countSource $\leftarrow$ heap.$getValue(edge.source) + \frac{1}{weight(edge.source)}$;
    heap.$insertOrUpdate(edge.source,$ countSource$)$;
    countTarget $\leftarrow$ heap.$getValue(edge.target) + \frac{1}{weight(edge.target)}$;
    heap.$insertOrUpdate(edge.target,$ countTarget$)$;
  end
  candidateNode $\leftarrow$ heap.$extractMax().key$;
  partialSolution $\leftarrow$ partialSolution $\cup$ {candidateNode};
end
return *partialSolution*

The algorithm seen in 2 builds a new solution(or uses a partial candidate to create a valid candidate since the method will be almost the same) by taking the nodes with higher $\frac{degree}{weight}$ and adding them to the current solution. It also updates the ratio by not considering covered edges dinamically during the algorithm. This algorithm is not perfect since it will also add some nodes that will eventually have all linked edges covered by other nodes at the end of the algorithm.

Other algorithms other than the one illustrated in 2 have been implemented(especially a more random algorithm) but the concept is almost the same, since the only difference among the implementation is not taking nodes by the ratio of $\frac{degreeNotCovered(\text{node})}{weight(\text{node})}$, but by selecting random edges and adding to the partial solution one or both the nodes of the edge (this algorithm described is the same as 5, when the partial candidate is the empty set).

Algorithm 3: Cost

Data: Graph G,solution
Result: weight of the solution
sum $\leftarrow$ 0;
foreach *node* $\in$ *solution* **do**
  sum $\leftarrow$ sum + $weight$(node);
end
return *sum*

The cost is computed by the algorithm described in 3, where all the weights of the nodes in the solution are summed to obtain the cost. Obviously, the solution should be controlled before with the validity algorithm(6,7) to be a valid solution.

When controlling for a candidate solution, the cost could be easily estimated by subtracting the removed nodes weight and adding the weight of inserted nodes; every time this is done to evaluate the cost, the number of objective evaluation increases.

Algorithm 4: Perturbation

Data: Graph G,solution,power,rarity
Result: perturbed solution
perturbedSolution $\leftarrow$ solution;
nodesNotInSolution $\leftarrow$ $G.nodes$ $-$ solution;
maximumNodesChanged $\leftarrow$ $\frac{|G.nodes|}{100} power$;
nodesChanged $\leftarrow$ 0;
while $nodesChanged < maximumNodesChanged$ **do**
  `/* choose a random node to remove or add to the perturbed solution */`
  ⋮
  nodesChanged $\leftarrow$ nodesChanged + 1;
end
return *CompletePartialSolution(perturbedSolution)*

The algorithm to perturb a candidate solution is presented in 4 and it takes nodes from the solution and throw them out of the solution; viceversa takes nodes not in the solution and it adds them to the candidate solution. The implementation is done by creating two vectors of nodes in and not in the candidate solution and choosing at random which node to throw in or out. Additional considerations on the perturbation are described in 2.2.

After this, the solution could not be valid yet, so the algorithm 5 is used to create a valid solution.

Algorithm 5: CompletePartialSolution

```
Data: Graph G,partialSolution
Result: valid solution to MWVC
solution ← partialSolution;
while uncovered(G.edges, solution) ≠ ∅ do
    /* Select edge at random                                    */
    ⋮
    /* Choose one of the nodes of the edge or both              */
    ⋮
    solution ← solution ∪ {candidateNodes};
end
return solution
```

In the algorithm illustrated in 5, the partial solution is passed as a parameter and uncovered edges are controlled by seeing if one end of an edge is in the partial solution. If both edge ends are not in the solution, there are 3 possibilities:

- choose the source of the edge and add it to the solution.
- choose the target of the edge and add it to the solution.
- choose both source and target and add them to the solution.

The graphs considered are all undirected, that means that *source* and *target* are only names used to differentiate the two nodes that compose an edge.

Algorithm 6: Validity

```
Data: Graph G,solution
Result: True if the solution is valid, false otherwise
foreach edge ∈ G.edges do
    if notCovered(G, solution, edge) then
        return false
    end
end
return true
```

The *notCovered* operator controls if at least one node of the edge passed as a parameter is in the solution. The graph considered for this algorithm are all **undirected**, if the graphs were **directed** the check over the edge is of the single source(if the source is in the solution or not).

An optimized version of the algorithm presented in 6 to check validity is presented in 7, this variant takes a set of nodes removed from the solution and the solution Itself and it controls if all the edges that had a source or a target removed are covered.

Algorithm 7: ValidityNodesRemoved

```
Data: Graph G,solution,nodesRemoved
Result: True if the solution is valid, false otherwise
foreach node ∈ nodesRemoved do
    foreach target ∈ G.adjList(node) do
        if notCovered(G,solution,(node,target)) then
            return false
        end
    end
end
return true
```

Algorithm 8: LocalSearch

```
Data: Graph G, solution
Result: Candidate solution to MWVC
currentMinimumWeight ← Cost(G, solution);
for numIter ← 0; numIter < MAX_LOCAL_SEARCH_ITERATIONS; Incr(numIter) do
    previousMinimumWeight ← currentMinimumWeight;
    /* Neighbourhood of solution search                                   */
    ⋮
    sampledNeighbourhood ← sample(neighbourhood);
    foreach candidateSolution ∈ sampledNeighbourhood do
        candidateWeight ← Cost(G, candidateSolution);
        if candidateWeight<currentMinimumWeight then
            currentMinimumWeight ← Cost(G, candidateWeight);
            solution ← Cost(G, candidateSolution);
        end
    end
    if previousMinimumWeight≤currentMinimumWeight then
        /* Local optimum                                                  */
        return solution
    end
end
return solution
```

The local search algorithm in 8 implements the best improvement strategy, other strategies (first improvement, random improvement) were implemented but best improvement had the best performance overall.

This algorithm explores the neighbourhood of the current candidate solution with minimum weight to see if there is any other solution that is better than the current one. The neighbourhood is defined in 2.1. The neighbourhood is also sampled to limit the number of iterations(and especially function evaluations) done for every run of the algorithm.

The algorithm ends if the number of iterations is greater than the maximum established or when it encounters a local optimum, that is when the neighbourhood has no candidates that are better than the current one. It also stops execution if the number of objective function evaluation is greater than the maximum established by the assignment.

Algorithm 9: adjustPower

Data: G, power, iterWithouthImprov, iterWithImprov
Result: adjusted power of perturbation
adjPower ← *power*;
adjPower ← adjPower$(1 + \frac{(iterWithoutImprov - iterWithImprov)getMaxDegree(G)}{maxNumberOfIterationsWithouthImprovement * |G.nodes|})$;
return *adjPower*

The function in 9 only computes the correction of the power in relation to iterations of ILS with or without improvement.

Algorithm 10: adjustRarity

Data: G, rarity, iterWithouthImprov, iterWithImprov
Result: adjust rarity of perturbation
adjRarity ← *rarity*;
adjRarity ← adjRarity$(1 + \frac{(iterWithoutImprov - iterWithImprov)getAverageDegree(G)}{maxNumberOfIterationsWithouthImprovement * |G.nodes|})$;
return *adjRarity*

The function in 10 computes the correction of the rarity in relation to iterations of ILS with or without improvement just like the adjustment of power, but with a little difference. In the fraction, the average degree of the graph is used, opposed to the maximum degree used in the adjustment of power, that is because the power parameter should depend on how many nodes should be thrown out and in of the candidate solution to get a better candidate, while the rarity should be related to the expected value of edges for a node.

Algorithm 11: adjustProb

Data: G, probability, iterWithouthImprov, iterWithImprov, worstSolAccepted
Result: adjusted probability of accepting worst solutions
adjProb ← *probability*;
adjProb ← adjProb$(1 + \frac{(iterWithoutImprov - iterWithImprov - worstSolAccepted) getMaxDegree(G)}{maxNumberOfIterationsWithouthImprovement * |G.nodes|})$;
return *adjProb*

The function in 11 is similar to the other adjustments but an additional parameter is used to contain the velocity of this adjustment, that is how many bad solutions were already accepted, used to limit further candidate solutions accepted when too many worst candidates are accepted.

The adjustments presented in (9,10,11) were not very effective, so an additional improvement of the algorithm could come by fixing these functions to better adjust the parameter values.

2.1 Neighbourhood for local search

The neighbourhood definition for this type of problem depends heavily on the structure behind the problem Itself. In this instance in particular, the neighbourhood have a lot of importance to define the right algorithm and performance-wise.

The neighborhood for this problem is defined in the following way:

Starting from a valid solution, the operations done to build the neighbourhood are the following

- Removal of a node
- Substitution of a node in the solution
- Insertion of a node not in the solution

All three operations could be generalized as a substitution of 1 node not in the solution with 0 or N nodes in the solution(Insertion of a node) and the substitution of a node in the solution with nothing(remove a node).

Other possible operations to build a better and deeper neighbourhood are p removals of nodes from the solution and m insertions of nodes not in the solution, but this instance was not considered since the number of **Objective function evaluation** was limited and it is already enough to control the operations seen before.

A heuristic to build a better neighbourhood is that when a node is removed, **all the nodes in Its adjacency list should be inserted in the candidate** since every edge must be covered. The same logic(with the opposite operations, that is for an insertion, n removals) could be applied to insertions of nodes not in the solution but, in that instance, not every node in Its adjacency list should be removed from the candidate since some nodes could be covering for other edges.

An additional control could be done when a node is inserted: if a node in the adjacency list of the node inserted has all the nodes adjacent in the candidate solution, then the node could be removed with no issues. This control was not implemented since it is an optimization and has no consequences on the final solution(It could be implemented in future versions).

The control of the nodes to substitute in or out is quite a toll on the algorithm, but additional optimization could be done especially for single removals and insertions. For removals especially, to remove a node there is no need to control the objective function since it can be removed and the solution will be better than the current one. This approach is too bounding sometime since it will only remove nodes from the candidate solution without considering other operations for the neighbourhood that could be better than removing a node, but if the general operation is implemented(p removals and m insertions), this approach could save both running time, objective function evaluations and find better solutions.

2.2 Perturbation

The perturbation operator defined in 4 builds a new candidate solution from the one passed as a parameter, The number of nodes inserted or removed is limited to a number dependent on the power of the perturbation.

A choice was also made to distribute the perturbation on nodes in the solution and nodes not in the solution because the random algorithm that was making no difference between the two cases(by bit flipping an entry of the candidate solution boolean array) was not performing well. In this way, the possibility of kicking out bounding nodes that block the local search algorithm in a local optimum and inserting nodes from the global optimum becomes more probable.

The perturbation is done with the following parameters:

- **Power**: Power $\in [\mathbf{MinPowerValue}, \mathbf{MaxPowerValue}] \subseteq \mathbb{R}$, this parameter is the one that controls how far the perturbed solution should be from the current solution, this parameter changes as the algorithm fails or succeeds to find better solutions in each iteration. By default, minimum value is 0 and max value is 100. A value of 100 means that the solution will be completely different(according to how the perturbation is built with the **Rarity**), a value of 0 means that the solution will be the same.

- **Rarity**: Rarity $\in [\mathbf{MinRarity}, \mathbf{MaxRarity}] \subseteq \mathbb{R}$, this parameter controls the distribution of the perturbations displacements(distance from the solution), an high value ($75 \leq \text{Rarity} \leq 100$) means that every perturbation will be at max power(the furthest possible for the current power value), a lower value($0 \leq \text{Rarity} \leq 25$). This value should be seen as the parameter(probability of success) of a **Binomial distribution**.

The **acceptance condition** introduced in 1 is described in the following formal definition:

Definition 2.1. (**acceptance criterion**). Given a candidate solution s and a perturbed solution after a local search *s'*, the acceptance condition is defined as follows:

- if $f(s) > f(s')$, accept *s'* as the current solution and iterate another pass of ILS(until termination condition is met).

- if $f(s) \leq f(s')$, accept *s'* with a probability p, this probability changes as the algorithm fails or is successful to find better solutions in the search space.

Worst performance were noticed when the probability of accepting solutions was set even to small values, so the acceptance condition should be considered only when the candidate solution is better than the current solution, as the algorithm depends heavily on nodes in the current solution.

2.3 Parameters tuning

Values for parameters were meticulously selected to get the best result possible for the current implementation, these values are listed as follows:

- **Power** , this parameter controls the power of perturbation. As a starting value it is assigned a 5, that means that around 5% of the solution will be perturbed. it changes during execution as shown in 9;

- **MaxPowerValue** , this parameter is the upper bound of the power parameter, it has a default value of 20;

- **MinPowerValue** , this parameter is the lower bound of the power parameter, it has a default value of 0.1(the solution will remain almost the same);

- **Rarity** , this parameter controls the rarity of perturbation, that is the probability of a single node to be chosen as a candidate or to be thrown out of a candidate solution. As a starting value it is assigned a 50, that means that a node will be thrown out or in of the solution when selected around the 50% of times. It changes during execution as shown in 10;

- **MaxRarity** , this parameter is the upper bound of the rarity parameter, it has a default value of 100, that means that when a node is selected to be thrown out or in of the candidate solution during a perturbation, it will always happen;
- **MinRarity** , this parameter is the lower bound of the rarity parameter, it has a default value of 0.1(almost impossible for a node selected to be thrown out or in of the candidate solution);
- **ProbabilityOfAcceptingSolutions** , this parameter controls the probability of accepting a solution during execution(either better or worst than the current one selected as the best solution). As a starting value it is assigned a 0.1, higher starting values lead to worst performance overall. It changes during execution as shown in 11;
- **MaxProbabilityOfAcceptingSolutions** , this parameter is the upper bound of the ProbabilityOfAcceptingSolutions parameter, it has a default value of 10;
- **MinProbabilityOfAcceptingSolutions** , this parameter is the lower bound of the ProbabilityOfAcceptingSolutions parameter, it has a default value of 0;
- **samplingFactor** , this parameter controls how many candidate solutions for operation(remove, substitute, insert) will be tested in local search for every iteration, an higher value lead to better performance for smaller problems, but with large graphs, it will eventually exhaust the objective function evaluations available. Lower values leads to better performance overall but slower convergence;
- **depthOfRemoves** , this parameter controls the maximum depth of removals when inserting a node (how many node will be thrown out of the solution), it has a default value of 50, that means that at max 50 nodes will be removed from the solution when inserting a node to create a neighbourhood candidate;
- **recursiveDepth** , this parameter controls if the depth of removals should be recursive, that is neighbourhood candidates will be generated by removing 1,2,...,depthOfRemoves nodes. It has a default value of *true*, if it is *false*, the neighbourhood will be generated by considering the maximum amount of nodes removable from the candidate solution(at a maximum of *depthOfRemoves* removals from the solution);
- **localSearchMaximumNumberOfIterations** , this parameter controls the max number of iterations possible for a run oflocal search .It has a default value of 500, the value is high enough for all the graph at hand, an higher value should be used if the graphs are bigger, if they are too big this parameter will be used as a limitation to not get too deep in local optimum.
- **ILSMaximumNumberOfIterations** , the maximum number of ILS iteration, it has a default value of 10000;
- **numberofIterationsWithouthImprovement** , the maximum number of ILS iteration without improvement, it has a default value of 10000;
- **propagationProportion** , this parameter controls the proportion of the propagation between nodes in the candidate solution and nodes not in the candidate solution. It has a default value of 5, that means that around 5% of the maximum nodes removed or inserted during a perturbation will be removed from the candidate solution. An higher value raises the possibility of throwing out optimal nodes while throwing in higher bounding nodes;
- **maximumNumberObjEval** , this parameter is the upper bound of objective function evaluation done. It has a default value of 20000, given by the problem.

Parameters tuning is done during execution after the checking for the condition of the acceptance criterion with the methods illustrated in 9,10,11 .

Also the *depthOfRemoves* and *recursiveDepth* parameters will become obsolete if the optimization illustrated in 2.1 is implemented, since all the nodes that respect the criterion defined there will be considered removable.

3 Experimental analysis

In the current section, experimental data will be analyzed. The experiments are conducted on the 3 types of graphs given by the problem, all data will be reported in the next subsection.

3.1 Iterated local search

As requested by the assignment, the average cost of the **smaller** and **medium** graphs is reported in the table 1, for the **large** graph of 800 nodes and 10000 edges, 15 experiments were done, and the average of the results have been taken.

input graph	average cost	average number of objective function eval
vc_100_2000	6051.90	5377.10
vc_100_500	4604.30	11374.40
vc_20_120	1038.20	29.60
vc_20_60	861.80	259.80
vc_200_3000	11605.50	3775.30
vc_200_750	8278.10	7378.30
vc_25_150	1264.00	99.30
vc_800_10000	44680.31	16024.88

Table 1: Average cost for the graphs and evaluations to find optimum

For smaller and medium graphs, the performance are really good and weight values are consistent through all the runs. For the larger graphs, things escalate quickly and values become dispersed. This can be seen in the table 2. Even though cost function evaluations for the larger graphs are low, the results were dispersed and quite high for other runs of ILS.

Experiments were done on 15 executions of ILS for every graph to obtain some good measurements. The results reported are quartiles of cost function evaluations, average for the costs and the average number of object evaluation computed during the algorithm to find the optimum.

Input graph	Min.	X1st.Qu.	Median	Mean	X3rd.Qu.	Max.	AvgObjEval
vc_100_2000_01	6057	6057.0	6057	6057.000	6057.0	6057	6080.600000
vc_100_2000_02	5864	5864.0	5864	5864.000	5864.0	5864	6984.000000
vc_100_2000_03	5738	5738.0	5738	5738.000	5738.0	5738	2622.066667
vc_100_2000_04	5959	5959.0	5959	5959.000	5959.0	5959	7840.000000
vc_100_2000_05	6292	6292.0	6292	6298.733	6302.5	6330	7245.666667
vc_100_2000_06	5904	5904.0	5904	5906.533	5904.0	5942	3926.000000
vc_100_2000_07	6297	6297.0	6297	6297.000	6297.0	6297	2425.933333
vc_100_2000_08	6211	6211.0	6211	6211.000	6211.0	6211	2747.466667
vc_100_2000_09	5957	5957.0	5957	5957.000	5957.0	5957	3346.000000
vc_100_2000_10	6240	6240.0	6240	6240.000	6240.0	6240	3.000000
vc_100_500_01	4475	4475.0	4475	4475.000	4475.0	4475	5864.733333
vc_100_500_02	4872	4872.0	4872	4893.667	4923.0	4949	9929.466667
vc_100_500_03	4370	4375.0	4380	4382.933	4392.0	4392	11873.533333
vc_100_500_04	4608	4622.0	4635	4629.267	4637.5	4640	7191.866667
vc_100_500_05	4807	4807.0	4807	4808.667	4812.0	4812	16105.600000
vc_100_500_06	4730	4730.0	4730	4747.333	4762.5	4795	9721.200000
vc_100_500_07	4637	4637.0	4637	4639.000	4641.0	4643	12436.266667
vc_100_500_08	4592	4592.0	4592	4596.333	4592.0	4643	15917.400000
vc_100_500_09	4544	4558.0	4560	4557.200	4562.0	4562	11775.666667
vc_100_500_10	4371	4371.0	4430	4406.467	4430.0	4431	10053.266667

Table 2: Statistics found by ILS for all the graphs, continues 3

vc_20_120_01	844	844.0	844	844.000	844.0	844	273.933333
vc_20_120_02	1009	1009.0	1009	1009.000	1009.0	1009	194.933333
vc_20_120_03	994	994.0	994	994.000	994.0	994	186.466667
vc_20_120_04	1050	1050.0	1050	1050.000	1050.0	1050	2.000000
vc_20_120_05	997	997.0	997	997.000	997.0	997	2.000000
vc_20_120_06	961	961.0	961	961.000	961.0	961	1.000000
vc_20_120_07	991	991.0	991	991.000	991.0	991	1.000000
vc_20_120_08	1142	1142.0	1142	1142.000	1142.0	1142	2.000000
vc_20_120_09	1261	1261.0	1261	1261.000	1261.0	1261	615.133333
vc_20_120_10	1133	1133.0	1133	1133.000	1133.0	1133	336.133333
vc_20_60_01	773	773.0	773	773.000	773.0	773	1468.133333
vc_20_60_02	938	938.0	938	938.000	938.0	938	11.533333
vc_20_60_03	730	730.0	730	730.000	730.0	730	61.600000
vc_20_60_04	757	757.0	757	757.000	757.0	757	529.000000
vc_20_60_05	871	871.0	871	871.000	871.0	871	324.133333
vc_20_60_06	855	855.0	855	855.000	855.0	855	1560.666667
vc_20_60_07	972	972.0	972	972.000	972.0	972	196.466667
vc_20_60_08	867	867.0	867	867.000	867.0	867	1.000000
vc_20_60_09	980	980.0	980	980.000	980.0	980	88.800000
vc_20_60_10	875	875.0	875	875.000	875.0	875	1.000000
vc_200_3000_01	11508	11517.0	11544	11547.533	11553.0	11616	11172.866667
vc_200_3000_02	11176	11176.0	11176	11218.133	11252.0	11345	8111.066667
vc_200_3000_03	11823	11823.0	11854	11841.667	11854.0	11876	6522.200000
vc_200_3000_04	12549	12563.0	12563	12592.467	12639.0	12662	9111.133333
vc_200_3000_05	11515	11515.0	11515	11525.267	11539.5	11550	6901.800000
vc_200_3000_06	11121	11121.0	11121	11121.000	11121.0	11121	3825.400000
vc_200_3000_07	11540	11540.0	11540	11540.000	11540.0	11540	7.466667
vc_200_3000_08	11638	11638.0	11671	11660.133	11671.0	11705	5302.933333
vc_200_3000_09	11681	11681.0	11681	11706.133	11734.0	11770	12418.000000
vc_200_3000_10	11504	11504.0	11504	11504.000	11504.0	11504	1106.666667
vc_200_750_01	8548	8548.0	8548	8548.000	8548.0	8548	3296.200000
vc_200_750_02	8240	8272.0	8272	8268.533	8272.0	8284	1595.600000
vc_200_750_03	8524	8524.0	8524	8526.933	8527.5	8547	7182.733333
vc_200_750_04	7943	7943.0	7943	7946.600	7943.0	7972	3369.933333
vc_200_750_05	8752	8761.0	8761	8760.400	8761.0	8761	2628.533333
vc_200_750_06	8153	8153.0	8153	8153.000	8153.0	8153	2820.666667
vc_200_750_07	7635	7635.0	7635	7637.000	7635.0	7665	2488.733333
vc_200_750_08	8388	8417.5	8427	8422.467	8427.0	8455	6420.933333
vc_200_750_09	8422	8484.0	8484	8471.600	8484.0	8484	4857.933333
vc_200_750_10	8176	8176.0	8204	8196.333	8204.0	8217	12058.333333
vc_25_150_01	1312	1312.0	1312	1312.000	1312.0	1312	626.666667
vc_25_150_02	1132	1132.0	1132	1132.000	1132.0	1132	5.533333
vc_25_150_03	1305	1305.0	1305	1305.000	1305.0	1305	899.000000
vc_25_150_04	1425	1425.0	1425	1425.000	1425.0	1425	1416.866667
vc_25_150_05	1307	1307.0	1307	1307.000	1307.0	1307	856.466667
vc_25_150_06	1249	1249.0	1249	1249.000	1249.0	1249	100.666667
vc_25_150_07	1450	1450.0	1450	1450.000	1450.0	1450	2.000000
vc_25_150_08	1151	1151.0	1151	1151.000	1151.0	1151	1.000000
vc_25_150_09	1248	1248.0	1248	1248.000	1248.0	1248	1.000000
vc_25_150_10	1061	1061.0	1061	1061.000	1061.0	1061	1.000000
vc_800_10000	44362	44653.5	44708.5	44680.312	44764.75	44858	16024.875000

Table 3: Statistics found by ILS for all the graphs

3.2 Convergence

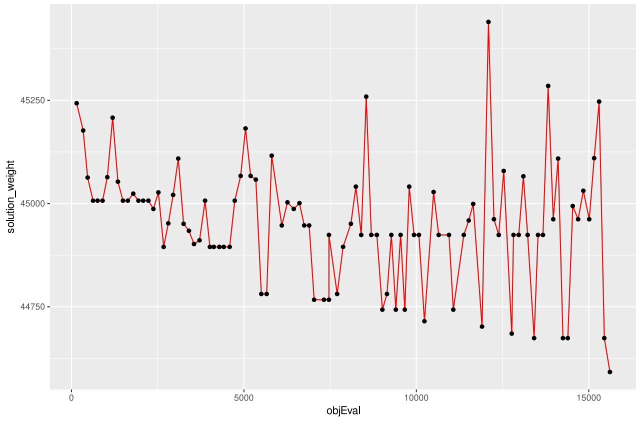

Figure 1: Test run of ILS on the large graph,
cost evaluations and perturbed solution weights plot

Because the algorithm performance depends heavily on random perturbations, the graph in 1 shows a lot of spikes, even though it seems to show a somewhat descending trend. For other graphs, the trend is more defined but more or less the same as the one showed in the figure, so other plots were not shown.

Running times were not included because they depend heavily on the parameters used(if the perturbation is of low power, the running times will be much faster). They were also not included because some variations of the algorithm presented that focus more on neighbourhood exploration and the operations of substitution and insertion are much more quicker and use a lot less objective function evaluation, at the cost of having a slower convergence and higher values in general since removals are performed beforehand without the consideration of the objective function.

An additional consideration should be done when the number of Cost function evaluation is not limited, in that case the algorithm finds better solutions than the ones presented in the table 3

4 Conclusion and future outlooks

The choice of Iterated Local Search for this type of problem is good for a lot of reasons, the most important are:

- Local search uses a neighbourhood that could be expanded to find better solution and eventually the global optimum.
- The perturbation operator allows to visit the search space thoroughly and eludes the possibility of getting stuck in a local optimum.
- As explained in the introduction, the whole concept of ILS is good for this type of problem because parts of the candidate solution are probably in the global optimum, and global optimum nodes tend to remain in the candidate solution during the execution of ILS.

Additional data analysis on results obtained could be conducted to see how the minimum weight vertex problem is characterized , like some comparative analysis on solutions obtained to see which nodes are more frequent in the sets obtained(or which group of nodes is more frequent, this problem is the search for **Frequent Itemsets**).

An additional update to how the neighborhood is searched is to consider a bipartite subgraph (p nodes to remove and m nodes to insert in the solution) and test which combination is valid.