UNIVERSITY OF CATANIA
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR'S DEGREE IN COMPUTER SCIENCE

Matteo Galletta

Minimum Weight Vertex Cover Problem

ARTIFICIAL INTELLIGENCE PROJECT

Professors: Mario Francesco Pavone
Vincenzo Cutello

Academic Year 2024 - 2025

Contents

Chapter 1

Introduction

1.1 Problem

Given a problem instance (G, ω), where G is a undirected graph $G(V, E)$ and $\omega : V \to \mathbb{R}^+$ a function that associates a positive weight value $\omega(v)$ to each vertex $v \in V$, the Minimum Weight Vertex Cover can formally be defined as follows:

$$\textbf{minimize} \quad \omega(S) = \sum_{v \in S} \omega(v), \quad S \in V$$

such that $\forall (v_i, v_j) \in E, v_i \in S \lor v_j \in S$.

Note that the MWVC is a NP-complete problem.

1.2 Proposed Solution and Motivation

This project implements a solution to the MWVC problem using a *Genetic Algorithm (GA) with one-point crossover and k-tournament selection.*

The chosen algorithm facilitates parallel computing compared to Tabu Search, as well as allowing to reduce the risk of the local optima stagnation. Also, Branch-and-Bound has limitations concerning the problem size, while Genetic Algorithms are suitable for NP-Hard problems such as MWVC.

Regarding the chosen Genetic Algorithm variation, even though it is not immediate to state whether one-point crossover will achieve better results over uniform crossover without prior testing, it has been widely proved that k-tournament selection outperforms roulette wheel in most scenarios [1].

Chapter 2

Genetic Algorithms

2.1 Behaviour and Structure

Genetic Algorithms (GAs) are a class of evolutionary algorithms inspired by the principles of natural selection. They operate by iteratively evolving a population of potential solutions towards an optimal or near-optimal state. The process unfolds as follows:

Initialization An initial population of candidate solutions is randomly generated.

Evaluation Each individual in the population is evaluated based on a predefined fitness function, which quantifies its quality or suitability with respect to the problem at hand.

Selection A subset of individuals are selected from the populations. The chosen individuals are selected as parents of the following population. Selection is based on fitness, with fitter individuals having a higher probability of being chosen.

Crossover Using crossover, selected parents genetic material are combined to create offspring.

Mutation With a certain probability, random mutations are introduced into the offspring. This is called mutation and it helps to maintain diversity within the population and prevents premature convergence towards suboptimal solutions.

Replacement This newly generated population replaces the old one.

The cycles of evaluation, selection, crossover, mutation, and replacement are repeated until a halting criteria is satisfied.

2.2 k-Tournament Selection

The selection algorithm for the proposed solution is *k-tournament*. This algorithm involves running several "tournaments" among k individuals chosen at random from the population, until the desired amount of population is reached. Each tournament selects the best amongst the k selected individuals.

The tournament size k can be adjusted to balance exploration and exploitation. Smaller k introduces more diversity, while larger k focuses more on exploiting fittest individuals. Given a population of n individuals:

$k = 1:$ selection is random, there is no preference based on fitness

$k = n:$ the fittest individuals are always selected

Algorithm 2.1: k-tournament selection

```
1 function k-tournament(population, k, n)
2     new_population = []
3     while new_population.size < n
4         selected = []
5         while selected.size < k
6             individual = random element from population
7             selected.push(individual)
8         end
9         new_population.push(best(selected))
10    end
11    return new_population
12 end
```

2.3   Single-Point Crossover

The simplest form of crossover is the single-point crossover, where a random crossover point is selected and the genetic material is exchanged between the parents at that point.

Algorithm 2.2: Single-point crossover

```
1 function single-point-crossover(parent1, parent2)
2     i = random integer in [0..n-1]
3     offspring1 = parent1[0..i] + parent2[i+1..n-1]
4     offspring2 = parent2[0..i] + parent1[i+1..n-1]
5     return offspring1, offspring2
6 end
```

2.4   Bit-Flip Mutation

Mutation is a genetic operator that introduces random changes in the offspring. The simplest form of mutation is the bit-flip mutation, where a bit has a probability p of being flipped.

Algorithm 2.3: Bit-flip mutation

```
1 function bit-flip-mutation(offspring, p)
2     for i in 0..n-1
3         if random() < p
4             offspring[i] = 1 - offspring[i]
5         end
6     end
7     return offspring
8 end
```

Chapter 3

Implementation

3.1 DEAP Framework

The chosen framework for the implementation is DEAP [2]. DEAP (Distributed Evolutionary Algorithms in Python) is a Python library that excels at rapid prototyping and testing of ideas, making the tool ideal for the project.

3.2 Gene Representation

The gene representation is a binary string, where each bit represents the presence of a vertex in the solution. The length of the string is equal to the number of vertices in the graph. The weight of the vertex is stored in a separate list, where the index corresponds to the vertex index.

1	1	0	1	0	0	1	0	1	1

Figure 3.1: Example of a MWVC gene representation

3.3 Fitness Function

The fitness function for the Minimum Weight Vertex Cover problem calculates the weight of the solution by summing the weights of the vertices present in the solution. Additionally, it imposes a penalty for each edge that is not covered by the vertices in the solution, ensuring that the fitness value is higher for incomplete solutions. The penalty function (or a similar alternative) is mandatory since we're using a direct gene representation that could create invalid configurations of uncovered vertices.

Algorithm 3.1: Fitness function

```
1 function fitness(individual)
2     f = 0
3     for i in 0..n-1
4         if individual[i] == 1
5             f = f + vertex_weight[i]
6         end
7     end
8
```

```
9        for edge in graph.edges
10            if individual[edge.from] == 0 and individual[edge.to] == 0
11                f = f + penalty
12            end
13        end
14        return f
15 end
```

The objective function is the same but with a penalty of zero, and it returns NULL if the solution is invalid (i.e. there are uncovered edges).

3.4 Performance Metrics

In order to evaluate the performance of the algorithm, it is necessary to define a set of metrics to evaluate the quality of the solution and the performance of the algorithm. The considered metrics are the following:

- **Convergence rate**: the rate at which the algorithm converges to the optimal solution.

- **Average fitness**: the average fitness of the population over time.

- **Best fitness**: the best fitness of the population over time.

- **Number of generations**: the number of generations required to find the solution.

- **Number of evaluations**: the number of fitness evaluations required to find the solution.

- **Stagnation**: the number of generations without improvement.

The execution time is not considered as a metric as it is highly dependent on the hardware and software environment.

It would be useful to include an optimality metric representing the difference between the best known solution and the solution found by the algorithm, but this information is not easily computable as the problem is NP-hard.

3.5 Parameters

Genetic Algorithms have a set of parameters that need to be tuned in order to achieve the best performance.

The following is the list of the meta-parameters configurable in the implementation:

- **Population size** (POPULATION_SIZE): the number of individuals in the population.

- **Crossover probability** (CXPB): the probability of crossover.

- **Mutation probability** (MUTPB): the probability of mutation.

- **k-tournament selection size** (K_TOURNAMENT): the size k of the tournament in the k-tournament selection.

- **Number of generations** (NUMBER_OF_GENERATIONS): the number of generations the algorithm will run.

The problem of finding the best parameters is an optimization problem itself, and it is possible to use a meta-heuristic algorithm to find the best parameters, such as a Genetic Algorithm [3]. This is out of the scope of this project even though it could be an interesting future development.

3.6 Benchmarking Flow

This paragraph describes the flow of the benchmarking process. The goal of the benchmarking process is to test the algorithm with a fixed parameters combination over different graph instances to extract the metrics and evaluate the performance of the algorithm. The parameters are chosen based on the literature and are discuess in the chapter 4.

The process is divided into the following steps:

1. **Load the graphs**: the problem instances for testing are loaded from the files.

2. **Compute graphs properties**: for each graph a set of properties are computed, such as the number of vertices, the number of edges and the density.

3. **Create the DEAP structures**: the DEAP structures are created, including the individuals, the population and the evaluation function.

4. **Run the algorithm**: the algorithm is executed with the set parameters over the graph instances.

5. **Save the results**: the results are saved to a file in a JSON format for further analysis.

The benchmarking process is repeated iteratively over different parameters combinations to find the best configuration for the algorithm.

Chapter 4

Results

4.1 Introduction

As previously stated, this chapter describes the process of tuning the parameters of the genetic algorithm and the results obtained from the experiments.

4.2 Iteration 1: Default Parameters

This first iterations consists of the most basic configuration of the genetic algorithm with literature defaults [4].

Parameter	Value
POPULATION_SIZE	100
CXPB	0.5
MUTPB	0.2
K_TOURNAMENT	3
NUMBER_OF_GENERATIONS	1 000 000

Table 4.1: Parameters for Iteration 1

After grouping and plotting the results by the problem class it's immediately clear that the genetic algorithm converges depending on the problem size (amount of vertices). This will be useful for later when limiting the number of generations. Since we still need to try different configurations, we will change the number of generations for last.

(a) SPI Problem Class   (b) MDI Problem Class   (c) LPI Problem Class
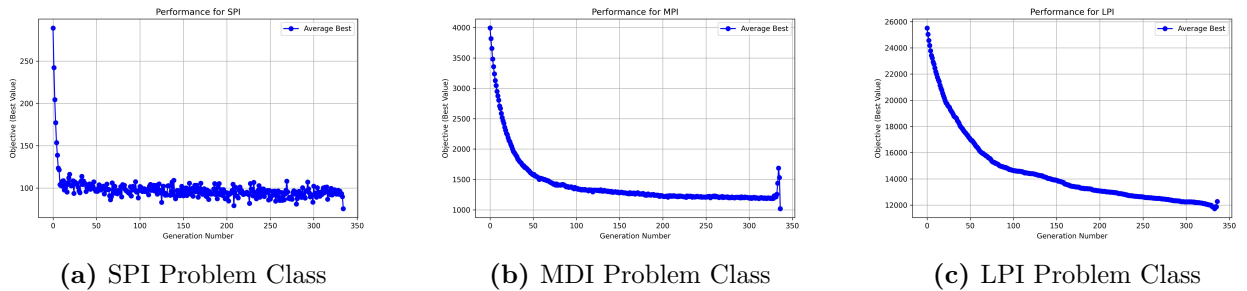
Figure 4.1: Objective function value by generation for each problem class (Iteration 1)

The results for the initial iteration are as follows.

Problem Class	Test Instance	Average Objective	Average Evaluations
LPI	10000	12317.3	20000.0
MPI	3000	1027.2	20000.0
MPI	2000	223.9	20000.0
MPI	750	2773.6	20000.0
MPI	500	903.5	20000.0
SPI	150	113.7	20000.0
SPI	120	92.2	20000.0
SPI	60	178.3	20000.0

Table 4.2: Results for Iteration 1

4.3   Iteration 2: Population Size

The first parameter to be tuned is the population size. The test is initially performed using a population size of 50, 75, 100, 150, 200 and 400. The results are compared using the average objective function value since the average number of evaluations would still be capped at 20000.

After performing the tests, it looks like the population size of 150 is the best for every test instance. The improvement is not significant, but it's still better than the other population sizes. Other than the original population sizes, different ones have been tested as well, but none of these have shown any improvement.

Pop. size	50	75	100	150	200	400
Avg obj	2229.7875	2196.2875	2203.7125	2126.1125	2259.25	2156.4375

Table 4.3: Average Objective for different Population Sizes

The results are as follows.

Problem Class	Test Instance	Average Objective	Average Evaluations
LPI	10000	11865.3	20000.0
MPI	3000	988.7	20000.0
MPI	2000	209.9	20000.0
MPI	750	2731.7	20000.0
MPI	500	822.7	20000.0
SPI	150	112.5	20000.0
SPI	120	84.3	20000.0
SPI	60	177.5	20000.0

Table 4.4: Results for Iteration 2

Conclusion

Bibliography

[1] Saneh Lata, Saneh Yadav, and Asha Sohal. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering Science*, 07 2017.

[2] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[3] Saman Almufti, Awaz Shaban, Rasan Ali, and Jayson Fuente. Overview of metaheuristic algorithms. *Polaris Global Journal of Scholarly Research and Trends*, 2:10–32, 04 2023.

[4] DEAP: Evolutionary Algorithms Made Easy. Examples: One max problem, 2012.