Draft, Implementation and Testing of Genetic Algorithm to find Defensive Alliances in Undirected Graphs

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Universität Trier
FB IV - Informatikwissenschaften
Professur Theoretische Informatik

Gutachter:	Prof. Dr. Henning Fernau
	Kevin Mann
Betreuer:	Prof. Dr. Henning Fernau

Vorgelegt am 15.09.2025 von:

Jannik Khan
Addresse
54292 Trier
JannikKhan@gmail.com
Matr.-Nr. 1532911

Abstract

The problem of finding defensive alliances in graphs is a computationally challenging task with significant applications, particularly in the field of network analysis. Although the theoretical properties of defensive alliances alliances have been studied extensively over the past years, there is a notable lack of practical algorithms for finding defensive alliances, even though the value of identifying such is well known in the field.

This thesis addresses the gap by developing a novel hybrid genetic algorithm that is designed to specifically discover defensive alliances in simple undirected graphs. The proposed algorithms enhance the standard genetic framework, by integrating various problem specific knowledge. The algorithm divides the problem into two parts, finding a defensive alliance and a defensive alliance of size k, by using a two phased fitness function with different features depending on whether the chromosome is already a defensive alliance. The proposed algorithm efficiently discovers defensive alliances that contain only nodes that can be in a defensive alliance of size k, but has been proven to be ineffective in discovering defensive alliances of a given size k.

The algorithm uses embedded elitism in the form of an integrated plus strategy, as well as learning functions that improve the candidate solution with a greedy approach.

The implementation is highly configurable, allowing for parameter tuning to optimize performance across various different graph structures. We showcase the effectiveness of the optimizations made with empirical evaluation on graphs with different properties.

Contents

1. Introduction and Motivation

Since the advent of social media, social networks of ordinary people have grown to an immense size. We receive a lot of information from many different sources. We colloquially describe networks we receive our information from and share with as the "bubble" we live in. In social network graphs, identifying such groups is particularly important. Groups that are internally cohesive and have many connections within themselves relative to outside connections are more resistant to external information. In the context of misinformation or adversarial propagation of content, such groups may act as echo chambers. They are susceptible to reinforcement of beliefs from within, but are resistant or less responsive to the influence of outside information. Recognizing alliances in social networks can help identify communities that might be more vulnerable to misinformation, or maintain the belief in false narratives longer due to internal reinforcements.

Alliances in graphs were first introduced by Kristiansen et al. in [1]. In this article, they were first classified into offensive, powerful, and defensive alliances. Since then, extensive research has been conducted in this field. Several surveys summarize the key definitions and results [2]. In particular, the literature establishes that finding small defensive alliances is computationally hard. Computing a minimum defensive k-alliance is NP-complete even for fixed k [2, 3]. More recent research such as Gaikwad and Maity shows that finding maximumsize minimal alliances or defensive alliances under various constraints is also NPhard [4]. From a parameterized perspective, the defensivealliance problem is W[1]hard when parameterized by measures like treewidth or feedback vertex set. This holds even when considering certain graph classes [5]. Researchers have collected many results across offensive, defensive, and powerful alliances [2] over time. However, there is relatively little on practical algorithms, suggesting that metaheuristics could be well suited for solving problems of this domain.

For members of the NP-complete problems [6], finding exact solutions is often infeasible for large instances; thus the development of efficient heuristic and metaheuristic approaches is necessary. One of these approaches is the Genetic Algorithm (GA), which has proven to be a powerful and versatile methodology for navigating complex solution spaces where traditional algorithms struggle [7]. Inspired by the principles of natural selection and evolution, GAs maintain a population of candidate solutions that evolve over generations through the application of selection, crossover, and mutation operators. The nature makes them particularly well suited for exploring the vast search spaces inherent to NP-complete problems, without much knowledge about their specific domain. This very property is also the reason they are less efficient, when compared to an algorithm designed to solve a specific problem. According to the *no free lunch* theorem [8] there cannot exist a general

algorithm that can compete with a specialized crafted solution to a specific problem. Genetic algorithms serve the purpose of being a tradeoff in time to develop.

Basic GAs can suffer from premature convergence towards suboptimal candidate solutions or fail to exploit valuable genetic material effectively. As a countermeasure, hybrid approaches were developed, which use problem-specific information to improve performance. Through the implementation of any knowledge about the problem, a better exploitation of genetic material and reduced convergence towards suboptimal regions should occur [9].

1.1 Contribution

Considering the amount of results in the field of alliances and the lack of practical algorithms in the field. We view the approach of developing a hybrid genetic algorithm to find alliances in graphs as suitable. Furthermore, we specify the task to find defensive alliances, since they fit the description of said echo chambers and groups resistant to information from the outside. The development of such algorithm offers a tool that opens up the possibility to search for defensive alliances in graphs, that has not been seen before. It enriches the research field of defensive alliances by contributing to where it lacks.

Main contribution of this thesis is the development and implementation of a hybrid genetic algorithm, which in case of existence reliably discovers defensive alliances of size k on any simple undirected graph $G = (V, E)$.

The proposed algorithm can be fine-tuned for performance through its parameters, offering a versatile and powerful tool for this NP-hard problem. We demonstrate its effectiveness through a comprehensive experimental analysis.

2. Preliminaries and Notations

Throughout this thesis, we adhere to the following graph-theoretic conventions:

- $G = (V, E)$ denotes a finite, simple, and undirected graph.
- The order (number of vertices) of G is denoted by $n = |V|$.
- Let $S \subseteq V$ be a non-empty vertex set and let $v \in V$ be an arbitrary vertex.
- The **open neighborhood of v in S** is the set of neighbors of v that belong to S:
$$N_S(v) = \{u \in S \mid (u, v) \in E\}$$
- The **closed neighborhood of v in S** is its open neighborhood plus the vertex itself:
$$N_S[v] = N_S(v) \cup \{v\}$$
- The **degree of v in S**, denoted $\deg_S(v)$, is the number of edges from v to vertices in S: $\deg_S(v) = |N_S(v)|$.
- The **complement** of the vertex set S is denoted by $S^c = V \setminus S$.
- If S is a defensive alliance, then the size of the defensive alliance is denoted as its cardinality.

For exact time complexity, we use $T(...)$ and to describe exact space complexity, we use $\mathscr{S}(...)$.

2.1 Basics and Notations for Defensive Alliances

Definition 2.1 (r-Defensive Alliance) *Given $r \in \mathbb{Z}$, a non-empty set $S \subseteq V$ is a* r-defensive alliance *in G if for each vertex $v \in S$,*

$$\deg_S(v) \geq \deg_{S^c}(v) + r.$$

Definition 2.2 (Defensive Alliance) *A non-empty set $S \subseteq V$ is a* defensive alliance *(DA) in G if for each vertex $v \in S$,*

$$|N[v] \cap S| \geq |N(v) \setminus S|$$

or equivalently,

$$\deg_S(v) + 1 \geq \deg_{S^c}(v).$$

A vertex $v \in S$ for which $\deg_S(v) + 1 \geq \deg_{S^c}(v)$ holds true is called protected*; otherwise, it is called* unprotected.

Remark 1 *The defensive alliance, as in 2.2 is also called* (-1)*-defensive alliance in literature, due to being a special case of a* r*-defensive alliance with* $r = -1$*. A* (0)*-defensive alliance is called a strong defensive alliance. When referring to a defensive alliance in this thesis, we always refer to the definition of a* (-1)*-defensive alliance 2.2.*

Proposition 1

$$2\deg_S(v) + 1 - \deg(v) \geq 0$$

Proof 1

$$\deg_{S^c}(v) = \deg(v) - \deg_S(v)$$

It follows:

$$\begin{aligned} &\deg_S(v) + 1 \geq \deg(v) - \deg_S(v) \\ \Leftrightarrow\ &2\deg_S(v) + 1 \geq \deg(v) \\ \Leftrightarrow\ &2\deg_S(v) + 1 - \deg(v) \geq 0 \end{aligned}$$

□

Lemma 1 *V is a trivial defensive alliance.*

Proof 2 *For $v \in V$ and $S = V$, it holds $\deg_S(v) = \deg(v) \geq 0$. It follows:*

$$\begin{aligned} &\deg_S(v) + 1 \geq \deg(v) - \deg_S(v) \\ \Leftrightarrow\ &\deg(v) + 1 \geq \deg(v) - \deg(v) \\ \Leftrightarrow\ &\deg(v) + 1 \geq 0 \end{aligned}$$

□

Lemma 2 *Let $v \in V$, if $\deg(v) \geq 2k+1$ then v cannot be in any defensive alliance of size at most k in G.*

Proof 3 *Suppose for the sake of contradiction that v is in a defensive alliance S of size at most k. Then $deg_S(v) \leq k-1$, since the highest possible degree that a vertex in a simple finite undirected graph $G = (V, E)$ can have is $|V| - 1$.*

With Proposition 1 we get

$$\begin{aligned} (k-1) + 1 &\geq \deg(v) - (k-1) \\ \Leftrightarrow\ k &\geq \deg(v) - (k-1) \\ \Leftrightarrow\ 2k + 1 &\geq \deg(v) \end{aligned}$$

As $|S| \leq k$, resulting in $\deg_S(v) < k$ and $\deg(v) \geq 2k+1$, we have

$$\begin{aligned} &\deg(v) - \deg_S(v) = \deg_{S^c} \\ =\ &(2k+1) - (k-1) = \deg_{S^c} \\ \Longrightarrow\ &\deg_{S^c}(v) > k \end{aligned}$$

Therefore $v \in S$ is not protected in S, contradicting the fact that S is a defensive alliance. □

Definition 2.3 (Marginally and Strongly Protected) *Let $S \subseteq V$ be a defensive alliance. A vertex $v \in S$ is called* marginally protected *if for every neighbor $u \in N_S(v)$, the set $S \setminus \{u\}$ is not a defensive alliance for v (i.e., v becomes unprotected in $S \setminus \{u\}$). Conversely, v is* strongly protected *if for every $u \in N_S(v)$, the set $S \setminus \{u\}$ remains a defensive alliance. [10]*

Definition 2.4 (Locally Minimal Defensive Alliance) *A defensive alliance S is* locally minimal *if for every vertex $v \in S$, the set $S \setminus \{v\}$ is not a defensive alliance. [10]*

Definition 2.5 (Globally Minimal Defensive Alliance) *A defensive alliance S is* globally minimal *if there does not exist a defensive alliance S' such that $S' \subset S$. [10]*

Definition 2.6 (Dominating Set) *Given an undirected graph $G = (V, E)$ a subset $S \subseteq G$, is called a* dominating set*, if for every vertex $u \in V \setminus S$ there is a vertex $v \in V$, such that $u, v \in E$.*

Definition 2.7 (Global Defensive Alliance) *A defensive alliance is called* global *if it is a dominating set.*

Definition 2.8 (Subset Sum Problem (SSP)) *Given a set X and a target k does there exist a subset $X' \subseteq X$ such that $k = \sum_{x \in X} x$.*

2.2 Basics of Genetic Algorithms

Genetic algorithms (GA) are heuristic algorithms inspired by natural selection. They make use of operators inspired by natural selection and biology, such as *selection*, *recombination*, and *mutation*, to find good solutions to optimization problems and search problems. Generally, they present a framework that makes it possible to solve any problem without much theorizing. The property of being a solution to problems of all kinds makes them less efficient in general. According to the *no free lunch* theorem [8] there cannot exist a general algorithm that can compete with a specialized crafted solution to a specific problem. Genetic algorithms serve the purpose of being a tradeoff in time to develop. One can write a relatively quick inefficient genetic algorithm that solves a problem unreliably compared to a specific optimized solution for the same problem. It should be noted that genetic algorithms may also make use of any information the given problem offers to solve it more reliably and with better solutions on average.

To understand the design decisions of the later proposed algorithm, it is important to be aware of the following definitions in context of genetic algorithms:

Definition 2.9 (Chromosome) *A* chromosome *is a data structure representing a potential solution to the problem the genetic algorithm tries to solve. A chromosome is composed of a smaller unit called genes.*

Definition 2.10 (Gene) *A* gene *consists of one or more chromosomes. It serves the role of being a candidate solution.*

Definition 2.11 (Allele) *The value of a gene is called* allele. *For instance, if a gene represents the eye color of a human, the alleles could be "blue", "brown", or "green".*

Definition 2.12 (Locus) *The* locus *describes a fixed position on a chromosome where a specific gene is located.*

Definition 2.13 (Population) *A* population *is a set of candidate solutions that interact with each other and are modified over generations to find an optimal solution. The initial population is generally created randomly.*

Definition 2.14 (Genome) *A* genome *is a candidate solution that consists of one or more chromosomes. If a genome only consists of one chromosome, one can use these words interchangeably.*

Definition 2.15 (Generation) *The* generation *refers to the current iteration the genetic algorithm is in.*

Definition 2.16 (Hamming distance) *For two binary vectors, the* Hamming distance *is the number of different positions.*

Definition 2.17 (Fitness) Fitness *refers to a numerical score, assigned by a fitness function, to a potential solution. The fitness of a chromosome determines the distance to an optimal solution. It also predicts the likelihood of a chromosome being selected for reproduction or to be part of the next generation depending on the selection methods used.*

Definition 2.18 (Fitness function) *A* Fitness function *takes a potential solution as argument and assigns a value to it, measuring its distance to the optimal solution for a given problem.*

Definition 2.19 (Selection) *The* selection *one of the main operators in genetic algorithms. It is inspired by the principle of* natural selection.

The operator performs two key roles:

- *It selects parent individuals from the current generation based on their fitness. These parents are then used in the* recombination *(crossover) operator to create new offspring for the next generation.*
- *It can also choose high-fitness individuals from the current population to be passed directly to the next generation.*

In essence, selection applies evolutionary pressure by probabilistically promoting better solutions, thereby driving the population toward improved fitness over successive generations.

Definition 2.20 (Selection Pressure) *Selection pressure describes the tendency to favor individuals with higher fitness, allowing them to reproduce more often.*

Definition 2.21 (Crossover/Recombination) *A* crossover *or* recombination *is an information exchange of genetic material between two or more chromosomes, resulting in the creation of new genetic material to create an offspring.*

Definition 2.22 (Mutation) Mutations *refer to a genetic operator that introduces random changes in the genes of a chromosome to maintain genetic diversity within the population and prevent the algorithm from converging on a suboptimal solution.*

Definition 2.23 (Baldwin effect) *The* Baldwin effect *describes the evolutionary mechanism in which behavior learned by an individual is passed down through generations, eventually becoming encoded and innate through natural selection. In the context of genetic algorithm, it can be interpreted as the integration of a learn function for individuals. The application of said function should improve the fitness of the individuals, resulting in a better convergence reliability and a faster convergence velocity. It should be noted that the effect of learning is most prevalent in the earliest generations, when individuals are furthest from the global optimum.*

Definition 2.24 (Convergence reliability) Convergence reliability *describes the ability of the genetic algorithm to produce reasonably good results. It is closely related to the concept of global convergence with probability one, which states that given infinite running time, the algorithm finds a global optimum with a probability of one.*

Definition 2.25 (Convergence velocity) Convergence velocity *is defined as the change in distance towards an optimum between two subsequent generations or as the change in the average distance of a population between two subsequent generations. When convergence velocity is measured in the first mentioned way, it is common practice to choose the best individual of a population to define convergence velocity.*

2.2.1 Genetic Algorithm Framework

The genetic algorithm starts by initializing the generation t with 0. The first population is initialized $P(t)$ with a fixed number N of potential solutions. Potential solutions within this initial population $P(0)$ are generated randomly. Afterwards, the fitness of each individual $v \in P_0$ is evaluated by applying the fitness function on them separately. After this step, the initialization is complete and the genetic algorithm loop starts.

The loop usually has some sort of break condition to ensure termination. Common termination conditions include the maximum number of generations reached, the

satisfactory fitness level achieved, the convergence of the population, or the computational time limit that has been reached. The first operation of the loop is the selection process. As explained in the definitions, it searches for individuals who participate in the upcoming recombination process based on their fitness. The recombination process then uses the genetic material of the selected individuals to produce offspring for the next generation. After the creation of said offspring the offspring will become part of a mutation process, in which a probabilistic approach is being used, to alternate their genes. Later, the fitness of all the new offspring is evaluated. Furthermore, the selection for the next generation begins. Individuals of the previous population $P(t)$ and the now evaluated offspring $C'(t+1)$ are chosen to create the next generation of the population $P(t+1)$. The last operation of the loop is the increment of the generation counter $t \leftarrow t+1$. From there onward, the loop repeats itself until the break condition is fulfilled.

Algorithm 1 General Framework of a Genetic Algorithm

1: $t \leftarrow 0$
2: Initialize population $P(t)$ with N genomes
3: Evaluate fitness of each individual $v \in P(t)$
4: **while** termination condition not satisfied **do**
5: Select parents $S(t+1)$ from $P(t)$ based on fitness
6: Create offspring $C(t+1)$ through crossover/recombination on $S(t+1)$
7: Apply mutation to offspring $C(t)$ creating $C'(t+1)$
8: Evaluate fitness of each individual in $C'(t+1)$
9: Form new population P_{i+1} from offspring $C'(t+1)$ and/or elites from $P(t)$
10: $t \leftarrow t+1$
11: **end while**

Definition 2.26 (Hybrid Genetic Algorithms) *The term* hybrid genetic algorithm *is being used when the generic genetic algorithm framework 1 is enhanced by the integration and incorporation of additional problem specific domain knowledge, heuristics or existing algorithms.*

2.3 Selection methods

In order to understand the generation change in genetic algorithms it is important to have a grasp of the main operators, selection, recombination and mutation. The most general of these is the selection. Crossover and Mutation leave more space to being modified through knowledge for a specific problem, to improve the performance of the genetic algorithm. Therefore, these two are going to be inspected further when drafting the later proposed algorithm.

2.3.1 Roulette Wheel Selection

Roulette Wheel Selection, also known as Fitness Proportionate Selection, chooses individuals based on their fitness. Imagine a roulette wheel where each individual gets a pocket sized according to their fitness. Higher fitness means a larger pocket. The wheel is spun and the individual where the ball lands is selected. This process is repeated until the desired number of parents is chosen. The time complexity is

$O(N)$ per selection, or $O(N^2)$ for the entire new population. The main upside is its direct proportionality: fitter individuals have a clearly higher chance of being selected. The major downside is that it performs poorly when fitness values are very similar or when a single super-individual dominates the wheel, making it hard to maintain population diversity later in a run.

Algorithm 2 Roulette Wheel Selection

Require: Population P, population size N, fitness function f

$F \leftarrow \sum_{i=1}^{N} f(i)$ {Calculate total fitness}
$r \leftarrow$ random number in $[0, F)$ {Spin the wheel}
$current\_sum \leftarrow 0$
for $i = 1$ to N **do**
  $current\_sum \leftarrow current\_sum + f(P[i])$
  if $current\_sum \geq r$ **then**
    return $P[i]$ {Individual found}
  end if
end for

2.3.2 Stochastic Universal Sampling

Stochastic Universal Sampling (SUS) is a selection method designed to be a fairer and more efficient version of roulette wheel selection. Instead of spinning the wheel multiple times, which can lead to bias, N equally spaced pointers are placed around the wheel and then the wheel is spun once. Every individual in whose slot a pointer lands is selected. This guarantees a low spread without bias, meaning that the selection perfectly reflects the fitness distribution.

Algorithm 3 Stochastic Universal Sampling (SUS)

Require: Population P, population size N, fitness function f

$F \leftarrow \sum_{i=1}^{N} f(i)$ {Calculate total fitness}
$P_{distance} \leftarrow F/N$ {Distance between pointers}
$P_{start} \leftarrow$ random number in $[0, P_{distance})$ {Start position of first pointer}
$parents \leftarrow [\,]$
$current\_sum \leftarrow 0$
$i \leftarrow 1$
for ptr in $[P_{start}, P_{start} + P_{distance}, ..., P_{start} + (N-1)P_{distance}]$ **do**
  while $current\_sum < ptr$ **do**
    $current\_sum \leftarrow current\_sum + f(P[i])$
    $i \leftarrow i + 1$
  end while
  $parents$.append($P[i-1]$) {Add individual whose segment the pointer landed on}
end for
return $parents$

2.3.3 Rank-based Selection

Linear Rank Selection

Linear Rank Selection addresses the problems of roulette wheel by basing selection on rank rather than absolute fitness. First, all individuals in the population are sorted from worst to best. Each individual is then assigned a selection probability based on its rank in the population, essentially protecting diversity.

Exponential Rank Selection

Exponential Rank Selection is a more aggressive variant of rank selection. The probability assigned to each rank follows an exponential distribution. The best individual is assigned a probability that is exponentially higher than the second best, and so on. This heavily favors the highest ranked individuals. The time complexity is also $O(N \log N)$ due to the sorting step. The main upside is the very strong selection pressure it provides, which can be useful for problems where exploiting the best solutions is critical. The downside is its tendency to reduce population diversity very quickly, which can lead to premature convergence towards local optima.

Algorithm 4 Rank-Based Selection (Linear or Exponential)

Require: Population P, population size N

1: Sort P by fitness (ascending: worst to best) $\{O(N \log N)$ step$\}$
2: Assign a probability p_i to each individual based on its rank i. {For Linear: $p_i \propto i$} {For Exponential: $p_i \propto c^i$ for some $c \in (0, 1)$}
3: Build a cumulative probability distribution C from the p_i
4: **return** individuals selected using Roulette Wheel Selection on C {Use Algorithm 2 with the cumulative distribution C}

2.3.4 Tournament Selection

In tournament selection, individuals from a population are randomly chosen to compete in a tournament. The chosen individuals are divided into groups of fixed size. The individual with the highest fitness of this group is selected as a winner. This process is repeated until the desired number of parents is chosen. The selection pressure can be easily tuned by adjusting the tournament size k; a larger k increases the pressure, while a lower k reduced it.

Algorithm 5 Tournament Selection

Require: Population P, population size N, tournament size k

1: $best \leftarrow \emptyset$
2: **for** $i = 1$ to k **do**
3: $candidate \leftarrow P$[random integer in $[1, N]$]
4: **if** $best = \emptyset$ OR fitness($candidate$) $>$ fitness($best$) **then**
5: $best \leftarrow candidate$
6: **end if**
7: **end for**
8: **return** $best$

2.3.5 Elitism

Elitism is a deterministic selection method that always ensures that the best solution found so far are chosen as parents or survivors to continue living in the next generation.

2.3.6 Plus strategy

The Plus strategy is often denoted as $(\mu + \lambda)$. It deterministically selects the best individuals from the combined pool of μ parents and λ offspring to form the next generation. This strategy guarantees that the best solutions are always preserved. It allows for a continuous improvement of the population fitness but can also increase the risk of premature convergence if diversity is not managed through other means.

3. Analysis

The goal of this chapter is a comprehensive inspection of the known results and properties of defensive alliances and genetic algorithms that need to be taken into account for the development of a hybrid GA that effectively solves the proposed task 1.1 . We start by stating the exact problem.

3.1 Problem Statement

Development and implementation of a hybrid genetic algorithm that in case of existence reliably discovers defensive alliances of size k on any simple undirected graph $G = (V, E)$.

3.1.1 Two Phases

The problem of finding any defensive alliance of size at most k is NP-complete [2]. In conclusion, we divide the task into two phases:

1. Finding a non trivial defensive alliance S inside G
2. Finding a defensive alliance of size exactly k inside G

Both phases try to solve an NP-complete problem. Finding a non trivial defensive alliance S within G is interchangeable with finding any defensive alliance of size at most $|V| - 1$ that is NP-complete [2]. The second phase is an even harsher requirement and consequently also NP-hard.

3.2 Identifying Pollution

Taking Lemma 2 into account, all vertices $v \in V$ with $\deg(v) \geq 2k + 1$ should be identified and restricted from being part of any chromosome in the population. It has be noted that this restriction reduces the effectiveness of the algorithm in identifying defensive alliances, directly excluding any potential defensive alliance of size greater than k.

3.3 Connected Components of Defensive Alliances

Any defensive alliance S can be decomposed into its connected components $\{C_1, C_2, \ldots, C_n\}$, with $C_i \subset S$ $(i \in \{1, 2, ..., n\})$ and each connected component C_i itself is a defensive alliance.

Proof 4 *This is proven by examining the defensive criterion for any vertex $v \in C_i$. If C_i is a connected component of the subgraph induced by S, then*

$$N_S(v) = N_S(v) \cap N_{C_i}(v) = N_{C_i}(v)$$

Therefore, we have the equality:

$$N_{C_i}(v) = N_S(v)$$

and consequently

$$\deg_{C_i}(v) = \deg_S(v).$$

Since S is a defensive alliance by assumption, $\deg_S(v) \geq \deg_{S^c}(v)$ holds for all $v \in S$. It follows immediately that for any $v \in C_i \subseteq S$:

$$\deg_{C_i}(v) = \deg_S(v) \geq \deg_{S^c}(v) = \deg_{C_i^c}(v).$$

□

This information needs to be considered for the second phase of the algorithm, which starts when the first defensive alliances become part of the population, for the choice of an efficient and effective crossover operator. This result opens up the possibility of finding a defensive alliance of cardinality k by solving the subset sum problem 2.8.

Observation 1 *The definition of defensive alliance 2.2 allows defensive alliances to be of cardinality* 1*. This is exactly the case for every vertex $v \in V$ with* $\deg(v) = 1$

Proof 5 *With* $\deg(v) = 1$ *and therefore* $\deg_S(v) \leq \deg(v)$

$$\deg_S(v) + 1 \geq \deg(v)$$
$$\deg_S(v) + 1 \geq 1$$

□

Observation 2 *When a graph is given as an adjacency list and $k \leq \sum_{v \in V} \deg(v) =$ 1, then the task of finding a defensive alliance of size k is trivial.*

3.4 Parameter

Each parameter of the genetic algorithm has to be carefully selected to ensure efficiency and effectiveness. Parameters need to be chosen dependent on each other and the graph in which the search is going to occur. As a result, this implementation was chosen instead of hard coding the parameters, which would ultimately result in an algorithm that is not worth deploying on certain graphs. Genetic algorithms are designed with the philosophy of general application in mind. To fit this idea, the proposed hybrid approach to find defensive alliances of size k was made as flexible and, in conclusion, as broadly employable as possible, without the cost of losing performance.

3.5 Handling Diversity

All evolutionary algorithms converge to local optima. In the best case scenario these local optima are global optima. To increase the chance of finding a global optimum, the management of diversity inside a population is necessary. If the diversity of the population is low, the algorithm converges faster to a local optimum. Once a population contains many chromosomes with small symmetrical distances to each other, caused through the convergence towards a shared local optima, these individuals will be one of the fittest individuals in the population. This makes them persistent in surviving, when applying the *plus strategy*, while simultaneously improving their chances of becoming part of the list of parents that create the next generation. When applying *one-point crossover* to pairs of parents with small symmetric difference, the chromosomes of the offspring will also have a smaller symmetrical difference to its parents. Once duplicates are inside the population they will converge exponentially towards a local optima.

There are a couple of ways to manage diversity. For example, when the mutation rate is increased, which is possible in this implementation, the diversity of the population in each generation after initialization is going to be higher compared to applying a lower mutation rate. This is because the parents chosen by the selection process are going to be more divers, which results in a higher symmetrical difference between them. As a result, the one-point crossover produces an offspring with a lower symmetrical distance towards its dominant parent on average.

In addition, a possibility is to restrict the selection to only choose an individual to become part of the recombination process once. Even though this violates the spirit of many selection methods, such as the roulette selection, it will slow down the convergence towards a local optima, with the cost of passing lesser high quality genetic material to the next generation.

Another way is to search for duplicates and remove them manually. The freed spaces are then occupied by new randomly generated genomes. The problem here is that randomly generated genomes are much more likely to have a worse fitness assigned to them compared to other individuals of the population. Therefore, they are sorted out rather quickly by the natural selection. The higher the generation value, the more likely the randomly generated genomes are to have a worse fitness score compared to their peers. This is due to the convergence towards a local optima. Keeping in mind the fact that a worse fitness score indicates lesser valuable genetic material. Considering the limited and short termed effect and additional computation cost removing duplicates manually causes, we conclude that no justification can be made to implement this form of handling duplicates.

Increasing the population size will promote more diversity initially, but has no long lasting effect, when combined with high offspring count and integration of the plus strategy.

One can also handle duplicates simply letting them stay in the population, e.g. not handling them, since the faster convergence towards a local optima could result in a faster convergence towards a global optima, if the chromosome representing the global optima has a small symmetrical difference to the chromosome representing the local optima.

3.6 Available Frameworks

There are some frameworks available for genetic algorithms, for example Jenetics for Java and PyGAD for Python. We decided not to use any given framework and build our own because the basic framework of a genetic algorithm is relatively easy to implement. Building our own algorithm from scratch makes it easier to alter the genetic algorithm by incorporating research results when implementing a hybrid variant. Additionally, a deeper understanding is won this way which could benefit the continuous research in this field.

4. Hybridization Draft

The goal of the proposed algorithm will be to reliably identify defensive alliances of a specific size k on undirected simple connected graphs. It has to work on every graph of this sort and find them in a reasonable time.

In this chapter, each part of the proposed genetic algorithm will be exemplified. Each section will give a close inspection on the overall idea and structure of a component of the genetic algorithm. The basic qualities and design decisions will be explained and justified. The proposed algorithm is not a simple genetic algorithm. It should be categorized as a hybrid genetic algorithm. Reason is the implementation of approximation algorithms presented in the Learning section 4.6.1 and the mutation strategy 4.6.

4.1 Preparation

Most practical graphs to search on for defensive alliances are social networks. Although some nodes in these types of graphs might have degrees almost as high as the highest degree possible for undirected simple graphs $|V| - 1$ degree, most nodes have degrees much lower than that. Consequently, it makes sense to use **adjacency lists** as a representation of the given graph. The adjacency list stores for each vertex $v \in V$ a list containing all its neighbors $N(v)$. It has a space complexity of $O(|V| + |E|)$ and returns all neighbors $N(v)$ in $T(v) = \deg(v)$ thus resulting in a much better implementation compared to an adjacency matrix, which has a space complexity of $O(|V|^2)$ and a time complexity to return all neighbors $N(v)$ of $O(|V|)$. These two properties are the only ones that will be needed by the algorithm, hence why the adjacency list is being chosen as a representation of the graph.

To check if a chromosome is a defensive alliance, information about the degree for each node $v \in S$ is needed, $S \subseteq V$. The degree of node inside the graph G never changes. Therefore, it is only logical to store $\deg(v)$ in a single array and have fast access at all times.

Respecting 3.2, a restriction will be created on all subsets to remove any unnecessary haze.

4.2 Chromosome Representation

When a graph $G = (V, E)$ is given, each node $v \in V$ has its own unique ID. This ID is a fixed value between 0 and $|V| - 1$. As a result, one can represent the existence of a node in a subset $v \in S$ using a binary one-dimensional array of size $|V|$.

Each array index is assigned to the respective value of the ID. The same assignment strategy will also be used for the integer array storing the respective degrees $\deg(v)$

mentioned in the previous section 4.1. The IDs are also going to be used in the adjacency list. This implementation ensures $O(1)$ access time for both $\deg(v)$ and $N(v)$.

Let $G = (V, E)$ be a graph with $n = |V|$ vertices, and let $f : V \to \{0, 1, \ldots, n-1\}$ be a bijection that maps vertices to array indices. For a subset $S \subseteq V$, define a binary array arr of size n such that for each vertex $v \in V$:

$$\text{arr}[f(v)] = \begin{cases} 1 & \text{if } v \in S \\ 0 & \text{if } v \notin S \end{cases}$$

Then the following logical equivalences hold:

$$(\forall i \in \{0, 1, \ldots, n-1\}, \text{arr}[i] = 0) \iff S = \emptyset$$
$$(\forall i \in \{0, 1, \ldots, n-1\}, \text{arr}[i] = 1) \iff S = V$$

Now using the terminology of genetic Algorithms, such subset $S \subseteq V$ is called a chromosome. The elements of the chromosomes are called genes and the values of those elements $f(v)$ are called allele. For all $v \in S : f(v) = 1$ and for all $v \notin S : f(v) = 0$. The index of a gene is called locus. The term genome describes the structure that holds the chromosome and any additional information about it. Each genome presents a potential solution. A population consists of genomes. The size of the population is going to be fixed and determined by the number of genomes within the population.

4.3 Fitness Function

In genetic algorithms each individual of a population presents a candidate solution. As a result, each candidate solution needs to have their fitness score evaluated. Fitness scores are evaluated by the fitness function. A good fitness score should measure the distance between the evaluated genome and the global optima. It needs to give enough information to drive the selective pressure of the genetic algorithm. To drive the selective pressure, it is crucial to have as much distinction as possible between the fitness score without disrupting accuracy. Therefore, the fitness score needs to reflect all relevant in measurements as precise as possible. It should weigh each measurement independently in relation to the specific problem it tries to solve.

The goal of the algorithm is to identify defensive alliances of size k. As a result, the fitness function will be divided into two parts. First, the check and measurement of a genome being a defensive alliance. And secondly, measuring the size of the genome in relation to the searched size k. The second measurement will only occur if a genome is identified as a defensive alliance.

If we deploy the second part of the fitness function on chromosomes that are not defensive alliances, the selection pressure would result in a higher discovery of said chromosomes closer to the size k. It is unclear if this might be more efficient in finding connected defensive alliances of size k. It would depend on many factors, such as the graph given as input itself. The original task of this thesis was to find defensive alliances using the genetic algorithm approach, which is why the decision has been made to deploy the second part of the fitness function after defensive alliances have been found.

If the second part of the fitness function were to be called on chromosomes that are not defensive alliances, then chromosomes closer to the desired size k would have been assigned a higher fitness score. This would ultimately result in selection methods that prefer those chromosomes closer to the size of k. Consequently, we would need to weigh both parts of the fitness function differently. Defensive alliances found using the first part of the fitness function, without ever using the second part, are most likely not connected, since a connected defensive alliance is a harsher criterion due to them being a subset of the defensive alliances.

That being said, presenting the first part of the fitness function, which gives a measurement to find defensive alliances. The most important measurement for finding a defensive alliance in graphs are the respective degrees $\deg_S(v)$ of each node in a subset in relation to $\deg(v)$. The definition of defensive alliances 2.2 in conjunction with Proposition 1 can be used to create a measurement function.

$$\sum_{v \in S} \min\{0, 2\deg_S(v) + 1 - \deg(v)\}$$

The function can be interpreted as follows :

- If a vertex $v \in S$ **protected** in S, then 0 is added to the fitness score of the chromosome S
- If a vertex $v \in S$ is **unprotected** in S, then $deg_S(v) + 1 - \deg(v)$ is added to the fitness score of the chromosome S

For each gene $v \in S$ that is unprotected, a negative value will be added to the fitness of the chromosome. This negative value gives the exact difference between $\deg(v)$ and $\deg_S(v)$. It will be used as estimate how far $\deg_S(v)$ is from being part of a potential defensive alliance S. This value will then be decremented each time another vertex $v \in S$ is unprotected in S. The fitness score of a chromosome itself is the total differences between $\deg(v)$ and $\deg_S(v)$ for all unprotected genes $v \in S$. As a result, the optimal fitness score is 0. This score is achieved by every defensive alliance, since for every defensive alliance

$$deg_S(v) + 1 - \deg(v) \geq 0$$

holds truth for every vertex $v \in S$. Note that this fitness function does not provide any information on how many vertices $v \in S$ violate the definition of being part of a DA.

In general, this fitness function would be enough to find a defensive alliance, but our goal is to find defensive alliances of a specific size k. Hence, another measurement is needed to assign a better fitness score to defensive alliances closer to size k. In addition, the optimal fitness score needs to be changed from 0 to a different value. This will be done in the second part of the fitness function.

Let $G = (V, E)$; $S \in V$; $0 < k < |V|$, then

$$|V| - |(k - |S|)|$$

describes the second part of the fitness function. The calculated value is always positive and gives a measurement of the distance between the size a genome S and the optimal size k. The maximum fitness score that can be assigned to a genome is $|V|$. This score will be assigned to a genome if and only if $|S| = k$. As a result, the closer the individual fitness score is towards the maximum fitness score $|V|$ the more fit the individual is.

The complete fitness function is as follows.

Algorithm 6 Fitness Function

1: $fitness \leftarrow \sum_{v \in S} \min\{0, 2\deg_S(v) + 1 - \deg(v)\}$
2: **if** $fitness = 0$ **then**
3: $\quad fitness \leftarrow |V| - |(k - |S|)|$
4: **end if**
5: **return** $fitness$

4.4 Selection

For the selection process of genetic algorithms, the most common selection methods used are the once presented in the introduction (2.3). When trying to solve a given problem, one should choose out of these selection methods with regard to convergence reliability and convergence velocity. The algorithm quickly converging to a local optimum is unwanted. Also, just as unwanted would be the algorithm not finding any global optimum at all. In the context of finding defensive alliances, a local optimum will be quickly reached if the population is swarmed by similar chromosomes. If this chromosome is far away from becoming a defensive alliance, then the convergence reliability will be worse. If, on the other hand, the chromosome is close to becoming a defensive alliance, then the convergence reliability would be quite high. Now respecting the criteria of finding alliances of a desired size k, per the definition of the algorithm as it is implemented right now, one would first have to find defensive alliances as explained in the previous section 4.3. In conclusion, the selection methods should prefer higher fitness individuals, while also considering populating the next generations with individuals similar to these, without prematurely converging towards local optima.

Due to the nature of recombination methods, offspring are going to have many genes resembling those of their parents. Therefore, the chromosomes of the offspring created by their parents are similar by definition. As a result of this, the next generation that will be populated by the children of the previous population is going to have many similar chromosomes. This process then cascades to a local optima over time. To slow down this process and keep the diversity of the population higher, in order to increase the probability of converging towards a global optimum, the option will be given for selection methods to choose a parent out of the current population only once. Duplicated parents are not allowed when this option is enabled.

Using this option alone would most likely result in a really low convergence velocity. To compensate for this, the **plus strategy** (2.3.6) will be implemented. There are two ways to measure convergence velocity (2.25), the first way is to look at the highest fitness chromosome of the previous generation and compare its fitness to

the highest fitness individual of the current population and the second way is to take the overall fitness score of the current generation into account and measuring it against the previous generation. The plus strategy confirms that the overall fitness of the next generation will at least be as high as the overall fitness of the previous generation, since worse fitness individuals out of the produced offspring will never be eligible to being part of the next generation, while all the highest fitness individuals survive. To increase the likely hood of creating a overall more fit generation each iteration it is suggested to produce a number of children that far exceeds to population size, to compensate for all the rejected offspring. Later in the learning section 4.6.1, a approximation algorithm is going to presented, that results in an increment for the fittest individual of the current generation can be ensured.

Instead of enabling this option, one should consider choosing selection method with a lower selection pressure first, such as SUS or linear rank selection. Also, this option should only be used in combination with selection methods that have a low selection pressure, to avoid unnecessary selection that are discarded right away.

4.5 Crossover

The implemented recombination method should keep the average size of a chromosome generated by it, closely to the average size of the previous generation. A fast trend towards either size increment or size decrement is undesired. If one would choose such method, the chromosomes sizes $|S|$ would trend heavily towards either one $|V|$ or 0, depending on if it is a increment or decrement. As a result the **one-point crossover** was chosen to be implemented. To perform a one-point crossover two parents are needed. Then a locus l with $0 \leq l \leq |V|$ is chosen. The child chromosome will inherit all genes from one parent up towards the locus l and inherit all the remaining genes from the other parent. To ensure that no child is going to be a clone of either parent, the locus l is random integer with $0 < l < |V|$. This ensures that at least one gene will be inherited from the non dominant parent.

4.5.1 One-point Crossover and the Subset Sum Problem

In the analysis, we mentioned a different approach in which one would try to solve the subset sum problem 2.8 with the union operator. The one-point crossover has a probability to have the same effect. This probability is heavily amplified if two smaller distinct defensive alliances are chosen for recombination. It is less efficient when it comes to solving the subset sum problem on a population that only consists of defensive alliances, when there is a union of defensive alliance within the population that has size k, because it can create in offspring that are not defensive alliances. In a population that only consists of defensive alliances, this is not tragic, since the plus strategy is deployed and therefore individuals with worse fitness than the inhabitants of the previous generation or fitter offspring will be disposed of. That being said, it also produces offspring that could not be created by the union operator. These offspring could theoretically satisfy the condition of being a global optimum, that would never be found by the subset sum problem, when there is no union of defensive alliance within the population that add up to the size k. Considering all this, the one-point crossover will be used even when the population is mainly occupied by defensive alliances and it will continue to have a positive effect, because of the use of plus strategy.

4.6 Mutation

Mutations are applied in genetic algorithm to encourage diversity and escape the convergence towards

The most basic form of mutation is a single bit flip. This bit flip is combined with a mutation rate. The mutation rate is a probability that is applied on each locus of the chromosome. It states how likely each gene in a given chromosome is to invert the value of its allele.

Adding nodes with high degrees to a subset will more likely have a stronger impact on the fitness value of the chromosome. When the degree $\deg(v)$ of a node is higher in general, the distance $|\deg_S(v) - \deg(v)|$ is more likely to be higher, because $v \in S$ has more neighbors in general that need to be part of S to decrease the distance between $\deg(v)$ and $\deg_S(v)$. Similarly, when removing nodes with high degrees from a subset, it is more likely to result in a larger distance change between $\deg(v)$ and $\deg_S(v)$. In this case, the reason is that removing a node with a high degree $v \in S$ from a subset will more likely impact the degree $\deg_S(u)$ of nodes $u, v \in S, u \neq v$, since v is more likely to be part of the neighborhoods of the other nodes in S.

As result, a multiplier $m = \deg(v)$ will be applied to the mutation rate r.

A restriction prohibits the inclusion of the vertices $v \in V$ with $\deg(v) \geq 2k + 1$.

4.6.1 Learning

Learning is one of the properties that a hybrid genetic algorithm can have. Learning methods take a greedy approach in by exploiting knowledge of a problem specific domain. Two methods are implemented in this style. First, a method that removes harmful genes and later one that tries to add genes to improve the fitness of an individual. The choice of having one method removing nodes and one adding nodes was made to balance the average size of chromosomes. Otherwise there would be a trend in increasing or decreasing the size of a genome.

As explained in the selection section 4.4, the plus size strategy is going to be employed. Learning will be given as a reward to survivors of the past generation. This should ensure in an increase of the fitness value for each survivor that is not a defensive alliance. As a result, the convergence velocity, when measured by the first criteria, is also going to be monotonically increasing.

Learning by removing harmful Genes

Applying the equation $2\deg_S(v) + 1 - \deg(v) \geq 0$ on each node $v \in S$, returns gives information on the fitness score change each node is responsible for. All protected nodes $v \in S$ do not change the fitness score and. For all unprotected nodes $v \in S$, $2\deg_S(v) + 1 - \deg(v)$ results in a negative value that decrements the fitness score. When calculating the fitness for each genome, a list of the most harmful nodes, e.g. the nodes for which the equation $2\deg_S(v) + 1 - \deg(v)$ has the lowest values, can be created. This list serves as the basis of the learning function, that removes nodes. Since the calculations used to determine harmful nodes are the very same the fitness function makes, a list containing all harmful nodes is also going to be provided as a result of the calculations done by the fitness function.

Now that the harmful genes inside a chromosome have been identified, a chromosome can improve its fitness by simply removing those harmful genes. We provide a fixed limit that gives the maximum amount of genes to remove in one learning process, is being used. The genes removed by this methods will always be those that decreased the fitness the most.

This method has no effect if it is used on a defensive alliances, because there are no unprotected nodes inside a defensive alliance.

Learning by adding Genes with High Degree

Learning by adding genes is realized by using the list of the highest degree nodes that were made beforehand 4.1 and the list all vertices that can not be part of a defensive alliance of size k, e.g all $v \in V$ with $\deg(v) \geq 2k + 1$.

The learning method works as follows:

- store the fitness value of S
- Add the highest degree node $v \in V \setminus S$, that is not part of the restricted list nor has already been tested.
- recalculate the fitness value
- if the fitness increased, accept
  else reject and restore fitness value

With this strategy, it is possible to give a maximum of vertices $v \in V \setminus S$ that are going to be added to S. It has to be noted that this process is quite expensive, because the fitness function needs to be used every iteration that has a time complexity $O(|V|)(4.3)$.

With having both a removing and an adding method, it is possible to let a genome learn while keeping the size mostly the same.

This strategy will also not be deployed on defensive alliances. Due to its nature of always adding a node to a subset combined with the uncertainty of the existence of a single node that can be added, this function would try to add every single vertex $v \in V$, which has a time complexity of $O(|V|^2)$ in worst case. The cost outweighs the benefit in the context of a genetic algorithm.

4.7 Handling Defensive Alliances

Once a defensive alliance has been found, it will be decomposed into its connected components. This is achieved using a depth-first search (DFS) in $O(|V| + |E|)$ time. Each connected component $C_i \subseteq S$ is itself a defensive alliance 4.

The fitness of each connected component is then calculated by applying only the second part of the fitness function 5.4. This calculation requires only $O(1)$ time per component.

Finally, the population is updated. Each connected component C_i is compared to the existing genomes in the population. Any genome with a worse fitness score

than C_i is replaced by C_i, provided C_i is not already present inside the population. This iterative replacement strategy seeds the population with smaller, connected defensive alliances. As a result, the population now consists of only the highest fitness individuals.

As mentioned in 4.6.1, defensive alliances, each node of a defensive alliance is protected by definition. Therefore, no nodes can be removed from them through the proposed learning method. Learning by adding, on the other hand, could still work, but is not applied due to the high computational cost expected in the worst case.

4.8 Execution

The proposed evolutionary algorithm will implement the generic framework for genetic algorithms 1, with minor amendments, to effectively find a defensive alliance of size k in a given graph $G = (V, E)$. The break condition will be having found the defensive alliance of desired size k and a maximum number of generation max that has been exceeded, to terminate the algorithm if it runs for to long without finding a defensive alliance of size k.

1. Set current generation t to 0
2. The algorithm will be given a integer N that determines the fixed size of the Population P.
3. N random chromosomes will be created using a probability $existence$, $0 < existence < 1$, that determines the likelihood for each vertex v to be part of a chromosome S. The higher $existence$ is, higher the average order of $S \in P$ will be.
4. Create list that stores all $\deg(v), v \in V$ for fast access and create a list with all $v \in V$ with $\deg(v) \geq 2k + 1$, that can not be part of any defensive alliance of size k. The later will serve as restriction not allowing the addition of any vertices that are part of this list.
5. For each $S \in P$ calculate the fitness apply the proposed fitness function.
6. **WHILE** (DA of size k not found or $t < max$)
   - 6.1 Select parents $Parents(t + 1)$ out of $P(t)$
   - 6.2 Use the one-point crossover to create offspring $C(t+1)$ from the selected parents $Parents(t + 1)$
   - 6.3 Mutate offspring $C(t+1)$ to create $C'(t+1)$, with the optimized mutation method, prioritizing higher degree nodes.
   - 6.4 Calculate the fitness of all offspring $C'(t + 1)$
   - 6.5 Perform *plus strategy* accepting only the fittest individuals out of $\{P(t) \cup C'(t + 1)\}$ to be part of $P(t + 1)$
   - 6.6 Apply learning survivors $P(t + 1) \cap P(t)$
   - 6.7 Discover all connected components of defensive alliances in $P(t + 1)$ and calculate their fitness.

6.8 replace individuals of $P(t+1)$ with higher fitness connected components

6.9 Increase t by 1

7. **END WHILE**

The presented ideas of this chapter serve as the basis of the implementation. In the following section a closer look will be taken into important implementation details and the overall structure of the algorithm.

4.9 A different Approach

We shortly want to mentioned a different approach, which also can be effective and efficient, depending on $G = (V, E)$. This approach was implemented in context of this thesis. For this approach the proposed chromosome representation 4.2 and the second part of the fitness function 4.3 are considered.

Given a randomly generated population, theoretically one could remove all unprotected nodes $v \in S$, $2\deg_S(v) + 1 - \deg(v)$ in one iteration by using $ALGORITHM$ by Rodolfo Carvajal[11], then calculate all unprotected nodes again and repeat this process to find the largest defensive alliance a subset S contains. $ALGORITHM$ returns the largest defensive alliance contained in a subset given as input and has a time complexity of $O(n^2)$ in the worst case. The larger the population size the higher the probability of finding a global optimum for a given size k, for the sole reason that there will be more genomes.

One can hope to try finding a connected defensive alliance of size k this way by chance. The problem of finding out if a defensive alliance of size k even exists is in NP [12], therefore one would stop this process after some time of not finding one, assuming that there might not be a global optimum.

This approach can also be combined with adding a second phase to the algorithm. We define the first phase as the search for a defensive alliance of size k using $ALGORITHM$, as explained above, while simultaneously preparing a large population set, for the second phase.

The second phase then uses the prepared population, consisting of defensive alliances, to solve the subset sum Problem 2.8 via either one-point crossover or union, in the same fashion as presented and justified in the "Handling Defensive Alliances" section 4.7. Note that the subset sum Problem is a NP-complete problem [6].

Using this approach, we try to find defensive alliances of size k, by skipping the first phase of the algorithm proposed above 4.8. Since $ALGORITHM$ always returns defensive alliances, the first part of the fitness function will not be needed, therefore the fitness of each individual can be directly calculated in $O(1)$. It is a tradeoff in which we replace the first phase of the genetic algorithm, which is the search for any non trivial defensive alliance, by a deterministic variant, that finds defensive alliances for a given subset.

The proposed algorithm of this thesis has been implemented in a way enabling the option of learning upon initialization of the population possible. The effect of learning is more prevailing in for genomes with lower fitness, and therefore when letting

the population learn a lot when initialized, relative to the expected chromosome size upon initialization, we receive a population with a high average fitness score that might as well contain defensive alliances. Considering this determining Which of the two proposed algorithms is more efficient and effective requires further research.

Even though this approach is not implemented yet, it is an idea worth discovering further.

5. Implementations

In this chapter the actual implementation of the most important functions will be explained in pseudo code. The implementation itself is in Java. The overall structure of the implementation will be shown in figure 5.1. Goal of this section is to create a deeper understanding of the complete hybrid genetic algorithm as it is implemented. In the previous section all important overall ideas were given, now a deeper look into the implementation and time and space complexity of the algorithm will be taken.

5.1 Structure

The illustration 5.1, shows a class diagram of the implemented algorithm. It holds the proposed structure of the previous chapter 4. Note that for the purpose of readability, the relations and attributes shown in 5.1 are not complete.

The algorithm is performed by the **Genetic_Algorithm** class. In here the main method is method is called, which then makes use of each component in the hybrid genetic algorithm. The algorithm outputs files storing all relevant information from its run and visualizes them by calling the visualization program **visualize_GeneticAlgorithmLogs.py**. The visualization program is the only program written in Python; all other parts of the genetic algorithm are Java classes. Python was chosen for visualization because of the NumPy package, which makes visualization of data sets easy.

Each class is implemented exactly as described in the previous chapter 4. The **Selection** class holds all selection methods, which are implementations of the previously introduced methods 2.3, with the addition of the option to forbid the selection of duplicates. The **Recombination** class implements the *one-point crossover* using the selected parents. As proposed the **Mutation** class implements both mutations, the standard one and the mutation favoring vertices of higher order.

The **Population** class represents the population and stores all potential solutions of a generation in a single fixed size array. The genetic algorithm only stores the current population, to avoid the allocation of unnecessary space that would be accommodated by previous generations. The method *newGeneration* is used after the selection of the next generation parents has been concluded. *newGeneration* then creates the next generation using recombination and mutation, as well as calling the learning functions, stored inside the **Learning** class, on the survivors. The method call is happening inside the **Genetic_Algorithm** class and therefore said relations are not shown in the illustration5.1, to improve readability.

The **Genome** class consists of the chromosome, implemented as proposed in the chromosome representation section of the previous chapter 4.2. It additionally stores

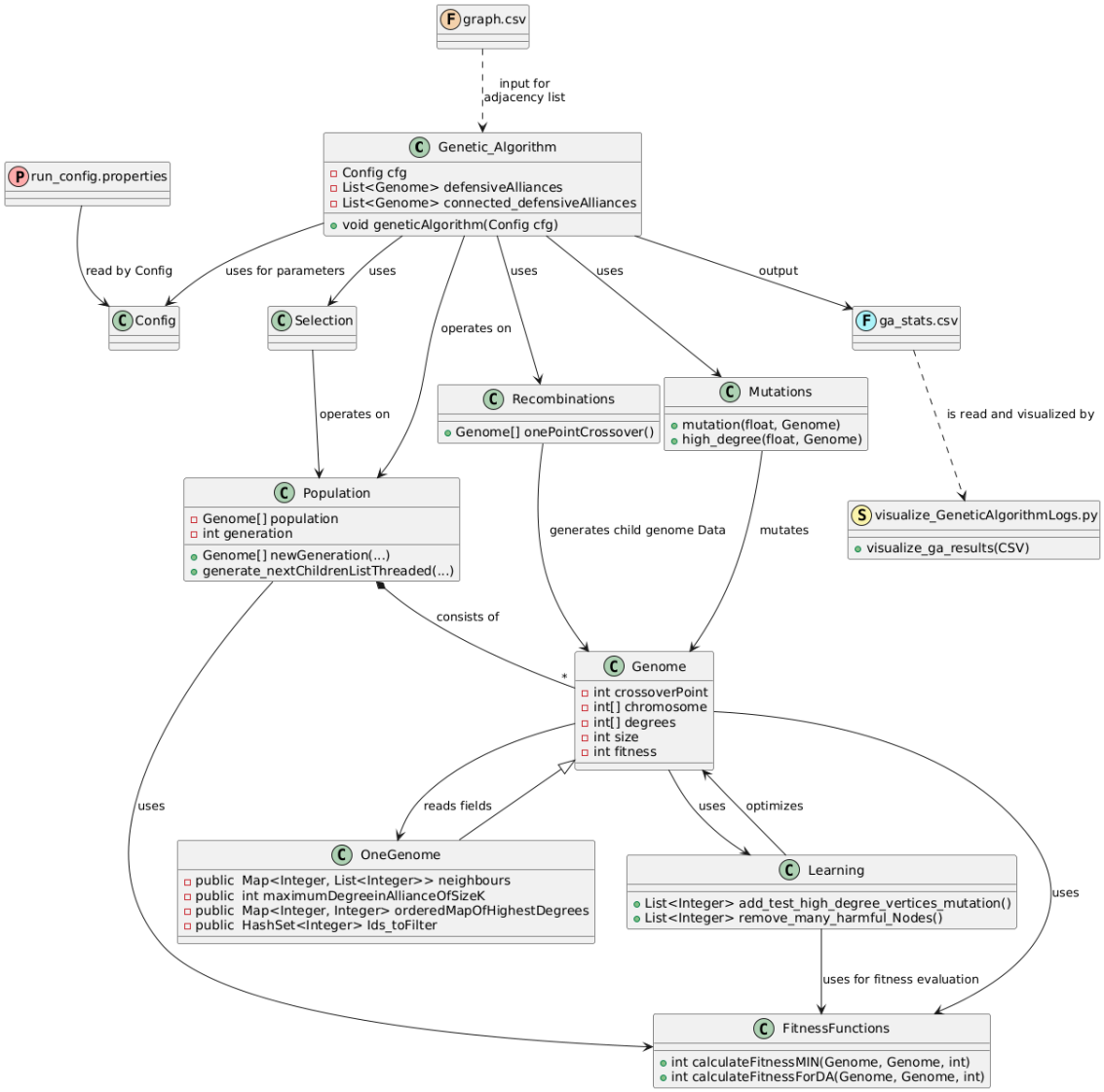

Figure 5.1: Structure of the hybrid genetic algorithm

all relevant meta data for the subset S it represents such as its size $|S|$, length $|V|$, fitness and the crossover point in the case of the chromosome being the direct result of a recombination process. Knowing the crossover point will significantly improve the calculation of $\deg_S(v)$ (see 5.3. The class also holds an array storing $\deg_S(v)$ for every $v \in V$ as previously mentioned 4.2.

The **OneGenome** class stores all important fixed information about $G = (V, E)$. As proposed in the previous chapter 4, the class contains the adjacency lists of G *neighbors* and *orderedMapOfHighestDegrees*. Both are implemented as maps, ensuring $O(1)$ time complexity for accessing the neighborhood $N(v)$ of any vertex $v \in V$ and its degree $\deg(v)$ via the *get(indexOf)* method. This method is always used to access any certain vertex $v \in V$, since the chromosome representation ensures that the locus of any gene is the same as the ID of the vertex it points to in the graph the search for a defensive alliance is happening on. *Ids_toFilter* is implemented as Hash set. The *contains* method of the Collection has a expected time complexity of also $O(1)$ in java, which is way faster than $O(n * log(n))$ when using a sorted list combined binary search. The reason the OneGenome class inherits from the

Genome class, is that both of these classes represent subsets of G, with OneGenome being the subset that equals G. Therefore it makes sense for OneGenome to inherit from Genome, because it is nothing more that a special case of a subset S, that the Genome class represents in general, being the trivial defensive alliance $S = V$.

5.2 The Configuration

All parameters for the genetic algorithm are stored inside the **run_config_properties** file, which is read by the **Config** class. The *Config* class then holds all the arguments of the genetic algorithm.

The implementation of flexibility in a genetic algorithm in the context of finding defensive alliance of size k is important, since the search for them differs depending on the inspected graph. Order, density, size and structure of the graph are all variables that impact the search. To suit the task of implementing a genetic algorithm fitted to search for defensive alliances on any undirected simple finite graph, it is essential to have parameters that are tunable to optimize the search for a given graph. This also gives a framework that makes it possible to test the effectiveness of the previously mentioned implementation ideas 4. The configuration gives the opportunity to test each hybrid feature in conjunction or in singularity. Furthermore, parameters such as `CAPPED_LEARNING` `AMOUNT_OF_LEARNERS` give the freedom to restrict the learning method, to tune the algorithm towards are less approximal approach. Note that by setting `FILTER_NODES_THAT_CANNOT_BE_IN_A_DEFENSIVE_ALLIANCE_OF_SIZE_K` to $false$, the algorithm will search for any defensive alliance regardless of cardinality in the first phase, ultimately increasing the chance of finding one. This will come at the cost of a worse search of defensive alliances of cardinality k, since the defensive alliances found while this option was set $false$, can contain vertices that can not be part of any defensive alliance of size k, and consequently their connected components can also contain these vertices.

Choosing a `public int POPULATION_SIZE` is heavily dependent on the order of the graph and therefore should be a flexible parameter. `NUMBER_OF_PARENTS` and `NUMBER_OF_CHILDS_PER_PARENT` need to be chosen depending on the `POPULATION_SIZE`, with the tendency for convergence towards a local optimum and the embedded plus strategy in mind. More parents result in more diverse children. More children result in less divers chromosomes in $C'(t+1)$, while less children result from more parents result in more divers offspring $C'(t+1)$. Less offspring result in a higher survivor count, which needs to be balanced by deploying `CAPPED_LEARNING` or choosing a lower `AMOUNT_OF_LEARNINGS`, since the learning by adding calls the fitness function each time. Additionally, allowing selection methods to choose certain individuals multiple times as part of the recombination process, due to higher relative fitness, aligns with their design philosophy but also lowers diversity of the created offspring.

- `public String FILEPATH;` // path of $G = (V, E)$
- `public int NUMBER_OF_NODES;` //equals $|V|$
- `public int POPULATION_SIZE;`
- `public float NODE_EXISTENCE_PROBABILITY;` // $0 < existence < 1$ 4.8

- `public int NUMBER_OF_PARENTS;` // $|Parents(t+1)|$ 4.8
  // must be even
- `public int NUMBER_OF_CHILDS_PER_PARENT;` // A parent is a pair
- `public float MUTATION_RATE;` //mutation rate r 4.6
- `public int NUMBER_OF_ITERATIONS;` //break-condition; max generation
- `public int SIZE_OF_DEFENSIVE_ALLIANCE;` // size k
- `public boolean FILTER_NODES_THAT_CANNOT_BE_IN_A_DEFENSIVE_ALLIANCE_OF_SIZE_K;`
- `public int BREAK_FITNESS;` //break-condition, fitness
- `public String SELECTION_METHOD;` // choose from 2.3
- `public boolean ALLOW_DUPLICATE_PARENTS;` // impacts selection methods
- `public String MUTATION_METHOD;` // standard or improved 4.6
- `public boolean ACTIVATE_LEARNING;` // true deploys learning 4.6.1
- `public boolean CAPPED_LEARNING;`
  // true fixes amount of survivors that the learning method is called on
  // ACTIVATE_LEARNING must be true for this to take effect
- `public int AMOUNT_OF_LEARNERS;`
  // amount of survivors the learning method is called
  // CAPPED_LEARNING must be true for this to take effect
- `public int AMOUNT_OF_LEARNINGS;`
  // gives upper bound of nodes that will be removed and added by learning
  // CAPPED_LEARNING must be true for this to take effect
  // ACTIVATE_LEARNING must be true for this to take effect
- `public boolean RANDOMIZE_LEARNERS;`
  // if true survivors the learn method is applied on are chosen randomly
  // CAPPED_LEARNING must be true for this to take effect
  // ACTIVATE_LEARNING must be true for this to take effect
- `public boolean DEPLOY_LEARNING_ON_INITIALIZATION;`
  // if true the learning methods are applied on the population upon initialization
- `public int AMOUNT_OF_LEARNINGS_UPON_INITIALIZATION;`
  // DEPLOY_LEARNING_ON_INITIALIZATION must be true for this to take effect
  // similar to ACTIVATE_LEARNING

5.3 Recalculating the Degree of mutated Chromosomes and Offspring

The calculation of $\deg_S(v)$, for a vertex $v \in S$ can be done in $T(v) = \deg(v)$ when using an adjacency list. This calculation has to be done for each $v \in S$, resulting in a total of $T(v) = \sum_{v \in S} |N_S(v)|$ when not optimized, e.g. not increasing the the degrees of two adjacent vertices $u, v \in S$ simultaneously. This operation will be made only $|P|$ times, with P being defined as the population. It will only be done on the initial randomly created chromosomes, that are part of generation 0.

Every time a chromosome is alternated one of its allele inverts its value. When a allele is inverted its value is either set to 0 or to 1. If it is set to 0 a node was removed decreasing the order of the subset it was in and if it is set to 1 a node was added increasing the order of the subset it is now part of. As a result two methods are implemented: **addNode** and **removeNode**. Both methods are part of the Genome class and are called inside it on their respective chromosome. Both methods *addNode* and *removeNode* first check if the allele of a given locus via *indexOf* indicates that the vertex that should be added in case of *addNode* is already part of the chromosome or already absent from the chromosome in case of *removeNode*, to avoid wrongly updating the genome. After the check, the method terminates in the case of the allele of the given locus *indexOf* showing absence for *removeNode* or presence for *addNode*. When not already terminated, the methods work as follows starting with *addNode*:

- *addNode* adds the vertex v to its subset S, setsdeg$_S(v)$ to 0 and increases the size of the genome, which always needs to be equal $|S|$. After these initialization steps an iteration through the open neighborhood $N(v)$ is started to identify $N_S(v)$ and update all degrees $\deg_S(u)$ by checking for each $v, u \in N(v)$ if $u \in S$. If $u \in S$, then the degrees $\deg_S(u)$ and $\deg_S(v)$ are incremented by 1. When the iteration over the open neighborhood $N(v)$ is over all degrees $\deg_S(v)$ will be correctly updated.
- *removeNode* removes the vertex $v \in S$ from S and decreases the size of the genome, so that the size of the genome equals $|S|$ again. Then the degree of each $\deg_S(u)$, with $u \in S \cap N(v)$ is decremented by 1. For any $v \notin S$: $deg_S(v) = 0$, therefore $\deg_S(v)$ is set to 0.

removeNode and *addNode* have a time complexity of $O(\deg(v))$ for every node $v \in V$ that shall be removed or added. The time complexity has an upper bound of $2k+1$, when all nodes that can not be part of any defensive alliance with size at most k are removed 3.2. Because when searching for a defensive alliance of size k, all $v \in S : \deg(v) < 2k + 1$ for any subset S, thus resulting in neither function ever being called on such node v.

Theoretically, it would be more efficient for *removeNode* to iterate through $N_S(v)$. To do this, each genome would need to store an adjacency list with space complexity $O(|V| + |E|)$ for its subgraph. Furthermore, this adjacency matrix would need to be calculated that has time complexity $T(S) = |S|^2$. This implementation bears an intolerable cost to benefit ratio. As a result, this approach of removing a node implemented.

Algorithm 7 Add Node to Genome

```
1: procedure addNode(index)
2: if chromosome[index] == 1 then
3:    return
4: end if
5: chromosome[index] ← 1
6: degrees[index] ← 0
7: size ← size + 1
8: neighbours ← OneGenome.neighbours.get(index)
9: for i ← 0 to neighbours.size() − 1 do
10:    neighbourIndex ← neighbours.get(i)
11:    if chromosome[neighbourIndex] == 1 then
12:       degrees[neighbourIndex] ← degrees[neighbourIndex] + 1
13:       degrees[index] ← degrees[index] + 1
14:    end if
15: end for
16: end procedure
```

Algorithm 8 Remove Node from Genome

```
1: procedure removeNode(index)
2: if chromosome[index] == 0 then
3:    return
4: end if
5: chromosome[index] ← 0
6: size ← size − 1
7: neighbours ← OneGenome.neighbours.get(index)
8: for i ← 0 to neighbours.size() − 1 do
9:    neighbourIndex ← neighbours.get(i)
10:    if chromosome[neighbourIndex] == 1 then
11:       degrees[neighbourIndex] ← degrees[neighbourIndex] − 1
12:    end if
13: end for
14: degrees[index] ← 0
15: end procedure
```

Calculating the Degrees after Crossover

A chromosome, that is a subset S, generated via one-point crossover does not require recalculating all vertex degrees $\deg_S(v)$ from scratch. The calculation can be significantly optimized by viewing the offspring as a mutated version of its *dominant parent*.

The **dominant parent** D is defined as the parent from which the offspring S inherited the majority of its genetic material. For parents $Parent_1$ and $Parent_2$:

$$D = \begin{cases} Parent_1 & \text{if } |S \triangle Parent_1| \leq |S \triangle Parent_2| \\ Parent_2 & \text{otherwise} \end{cases}$$

The set of allele that differ between S and D is defined as the **symmetric difference** between S and D, denoted by the operator $\triangle$:

$$S \triangle D = (S \setminus D) \cup (D \setminus S)$$

For one-point crossover, the number of changed vertices is bounded by:

$$|S \triangle D| \leq \frac{|V|}{2}$$

The set of changed vertices C is defined as the symmetric difference between S and D:

$$C = S \triangle D = (S \setminus D) \cup (D \setminus S)$$

For one-point crossover, the number of changed vertices is bounded by:

$$|C| = |S \triangle D| \leq \frac{|V|}{2}$$

This inequality guarantees that by considering S as mutated version of D, we never have to recalculate the degrees $\deg_S(v)$ for more than half of the vertices $v \in S$.

5.3.1 Algorithm

The algorithm for efficient recalculation of the degrees $\deg_S(v)$ is as follows:

1. Identify the dominant parent D
2. For each vertex $v \in (S \triangle D)$, recalculate $\deg_S(v)$ by calling
   - *addNode*, if $v \in (S \setminus D)$
   - *removeNode*, if $v \in (D \setminus S)$
3. For each vertex $v \notin (S \triangle D)$, reuse $\deg_D(v)$

Considering the time complexity of *addNode* and *removeNode*, with respect to the restriction of not adding any vertices $v \in V$ with $\deg(v) \geq 2k + 1$, since they cannot be part of any defensive alliance of size k, the total time complexity of the recalculation adds up to $T(v) = (|S \triangle D|) * (2k + 1))$. This is a heavily optimized version compared to the standard internal degree calculation which has to be done initially 5.3.

5.4 Fitness function

The time complexity of the first part of the fitness function is $O(|V|)$, because the algorithm iterates through the whole chromosome of length $|V|$ and to check whether a vertex v is part of S and applies the equation on it if it is. The second part of it has a time complexity of $O(1)$ and is only called if S is a defensive alliance. As explained explained in the draft 4.3, the list of harmful nodes is also calculated by this function to avoid redundant calculations.

Algorithm 9 Calculate Fitness for Defensive Alliance

Require: Genome *genome*, Chromosome $S \in genome$, Graph $G = (V, E)$, target alliance size k
Ensure: Fitness value f

$sum \leftarrow 0$
$H \leftarrow \emptyset$ {Initialize map for harmful nodes}
for $i \leftarrow 0$ to $|V| - 1$ **do**
  if $S[i] = 1$ **then**
    $x \leftarrow 2 \cdot \deg_S(i) + 1 - \deg_V(i)$
    $sum \leftarrow sum + \min(0, x)$
    if $x < 0$ **then**
      $H[i] \leftarrow x$ {Store harmful node information}
    end if
  end if
end for
$genome.harmfulNodes \leftarrow H$ {Store harmful nodes in genome}
if $sum = 0$ **then**
  $sum \leftarrow |V| - |k - |S||$ {Penalize deviation from target size}
end if
return *sum*

5.5 Connected Components of Defensive Alliances

We previously proofed that each connected component of a defensive alliance is a defensive alliance itself 4. In this section, we discuss the time complexity of handling defensive alliances as previously proposed 4.7. Once a defensive alliances has been found it will be divided into its respective connected components. This is done using *depth-first search (DFS)*, which has a time complexity of $O(|V| + |E|)$. All unique connected components are then added into a list. For each connected component, the fitness value is calculated using the second part of the fitness function only. To save memory we look at the worst individual of the population and remove all connected components from the list that have a lower fitness score. We then combine the population and the list of connected components into one. The list is then sorted by descending fitness order and duplicates are removed. This operation has a time complexity of $O(nlog(n))$. From this combined list, the best $|P|$ individuals are then chosen to form the population of the next generation. This population will contain only unique chromosomes.

All of these steps happen after the next generation has already formed 4.8. It is an additional step that is called when the population contains defensive alliances.

5.6 Time and Space Complexity for a given Configuration

The space and time complexity of the algorithm i heavily dependent on the graph it operates on and the parameters it works with, given by the properties file. It follows a comprehensive review of the time and space complexity.

5.6.1 Space Complexity

For a given graph $G = (V, E)$ the algorithm first transforms the graph into a adjacency list. This has a space complexity of $O(|V| + |E|)$, as mentioned before 4.1. Each genome holds a chromosome array and an array consisting of the respective internal degrees. Both of them are of size $|V|$. The population P consists of $|P|$ genomes. To avoid storing genomes of previous generations, the population is replaced by the next generation each time. Therefore, the space complexity of a population P is always $\mathscr{S}(P, V) = |P| * 2|V|$.

The amount offspring generated by recombination process is dependent on the number of parents chosen by the selection methods $\tau \in \mathbb{N}$ and the amount of offspring $\lambda \in \mathbb{N}$ each pair of parents produces. It holds that for every generation $\frac{\tau\lambda}{2}$ offspring are generated. Each child is genome. As a result the offspring have a total space complexity of $\mathscr{S}(V, \tau, \lambda) = |V| * \tau\lambda$.

Additionally, the *OneGenome* is also a genome which is part of the algorithm, even when not being part of the population. It holds the adjacency list and has a total space complexity of $\mathscr{S}(V, E) = 3|V| + |E|$. Considering all variables, the algorithm has a space complexity of $\mathscr{S}(V, E, P, \tau, \lambda) = (3|V| + |E|) + (|V| * \tau\lambda) + (|P| * 2|V|)$.

5.6.2 Time Complexity

All instances of the Genome class are created in $O(|V|)$ time. This is because of the chromosome they hold. Each chromosome needs to be a new instance and, therefore, needs to be filled with information. With each chromosome having a length of $|V|$ the mentioned time complexity follows. The initial internal degree calculation is called on each genome that has no parents. When the adjacency list is given and therefore all $\deg(v)$, $v \in V$ are known, it has a time complexity of $T(v) = \sum_{v \in S} \deg(v)$. The time complexity of genomes with parents has already been extensively discussed 5.3. When defensive alliance have been found and additional $O(nlog(n))$ time will be added to the iteration for sorting the combined list of connected components and the population 5.5.

The time complexity of a single iteration depends heavily on the order of the graph, the size of the population and the amount of offspring that are generated as candidates to inhabit the next generation. Recalculation of the degrees and calculation of the fitness score are done $\frac{\tau\lambda}{2}$ times each generation, with τ being the amount of parent candidates selected by the selection methods and λ being the amount of children produced per pair of parent. Every time offspring are created, the list containing all $\frac{\tau\lambda}{2}$ offspring needs to be sorted again, to identify the fittest individuals. This process has a time complexity of $O(nlog(n))$, $n = \frac{\tau\lambda}{2}$. Bearing this cost becomes less justifiable, the more offspring get rejected, which happens naturally by employing the plus strategy.

As a result, the total time complexity of each iteration of the proposed hybrid genetic algorithm has a time complexity of $O(\tau\lambda * |V|) + (\frac{\tau\lambda}{2} * log(\frac{\tau\lambda}{2}))$, $n = \frac{\tau\lambda}{2}$.

These results suggest that the parameters of the algorithm should be tuned in a way that keeps the amount of offspring generated moderate while having a larger population size and therefore a low selection pressure, to keep diversity high, when aiming

fast iterations to promote the repeated use of crossover, mutation and learning, ultimately speeding up the genetic algorithm tremendously compared to a configuration of contrary parameters.

6. Evaluation

To classify the relevance of the hybrid genetic algorithm, it needs to undermine an empirical evaluation.

6.1 Conditions and Methodology

We evaluate the proposed algorithm, by its effectiveness and efficiency in finding defensive alliances of certain size k. Furthermore, we test the effectiveness of the single optimizations made, such as the addition of learning, outsourcing of any vertices that cannot be included in a defensive alliance of size k and the promotion of mutating vertices with a higher degree. The tests are done using graphs of different sizes and with a vastly dissimilar transitivity. The transitivity values of the graphs range from 0.047, denoted as low, to 1.7 denoted as high.

The social networks that were used for the evaluation process were found on https://snap.stanford.edu

6.2 Basic Mutation vs Mutations

For the test cases in this section, learning functions were not called. We set the following fixed and uniform parameters on which all graphs will be tested:

- Population size $|P|$ is set to $|V|$
- The initial population will be inhabited by chromosomes that have for each locus a probability of 0.5 to set to 1. The expected average size of a chromosome is therefore $\frac{|V|}{2}$.
- All nodes that can not be inside a defensive alliance of size k are filtered out
- Stochastic Universal Sampling will be used for selection, allowing duplicates
- The amount of parents selected to participate in the recombination process is set to $\frac{|V|}{2}$
- The number of children per pair of parent is set to 4. This results in a total amount of $2|V|$ offspring each iteration
- The standard mutation rate is set to $\frac{1}{|V|}$
- The algorithm stops when a global optimum is found or the generation limit of 200 to 300 is reached, depending on the graph.

Each proposed optimization will be tested on graphs that searches defensive alliances of size $k \in \{\frac{3}{4}|V|; \frac{|V|}{2}; \frac{|V|}{4}\}$.

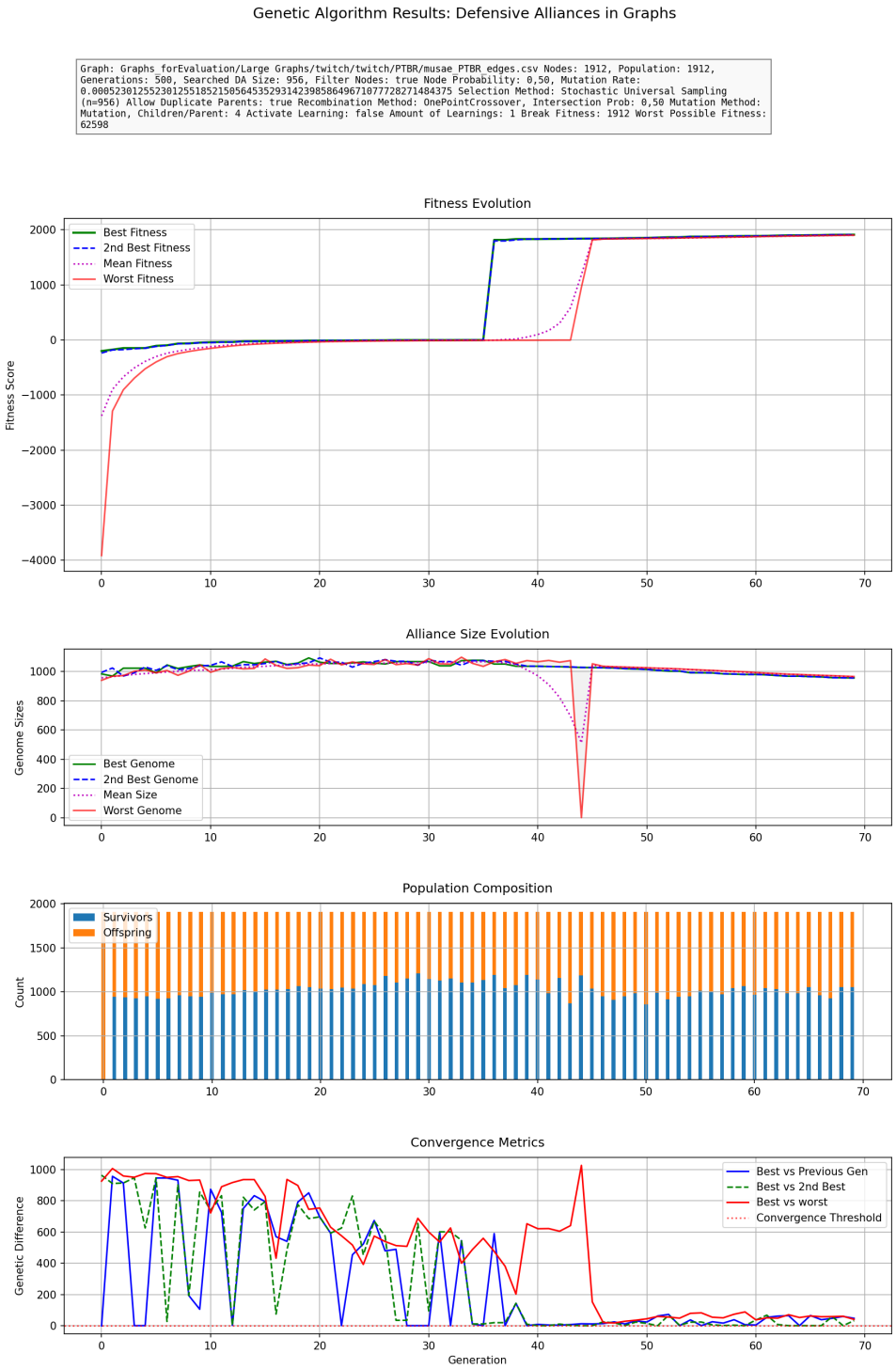

Figure 6.1: basic mutation on the same graph with high transitivity as in figure 6.2

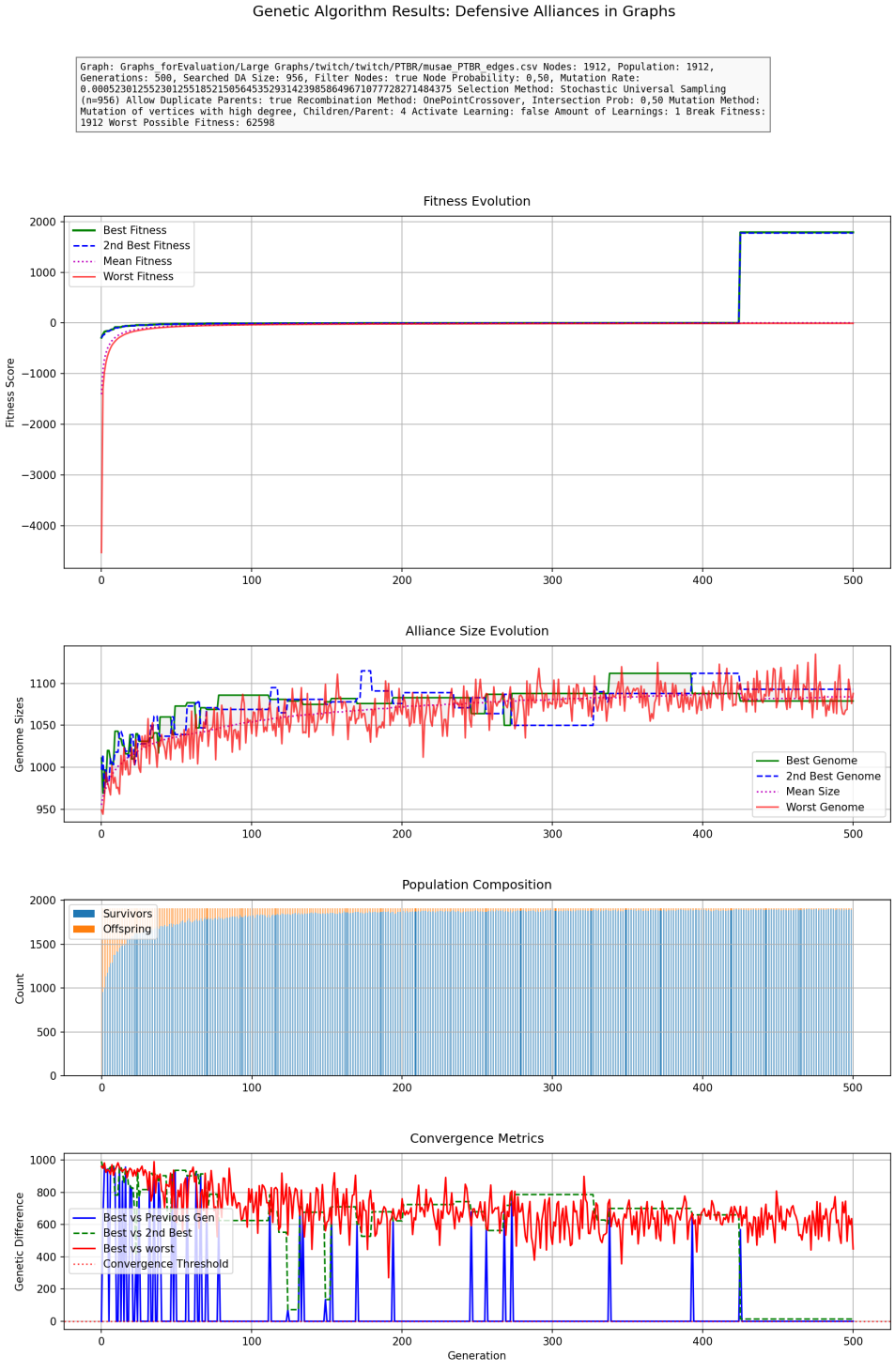

Figure 6.2: high order mutation on the same graph with high transitivity as in figure 6.1

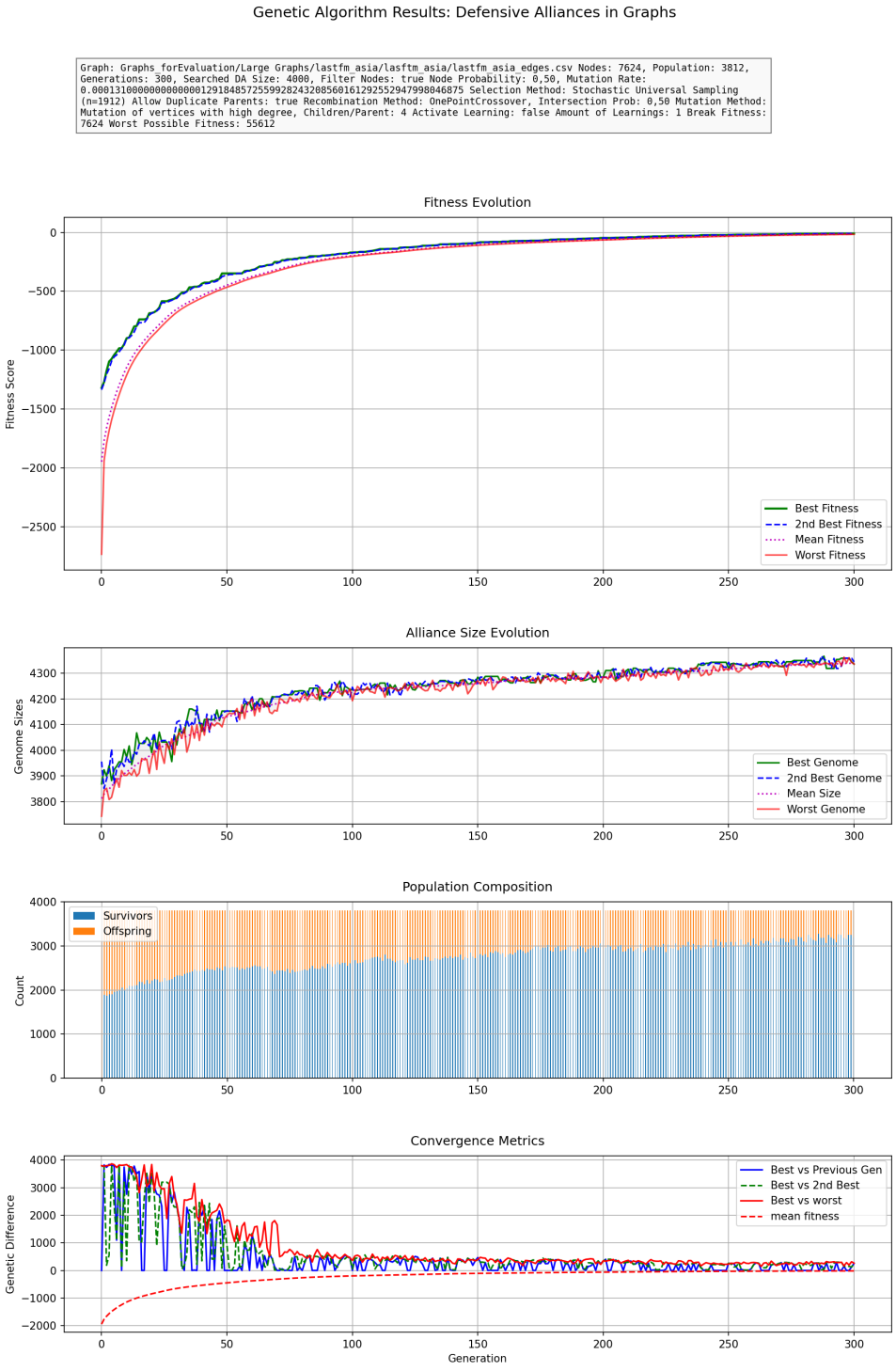

Figure 6.3: Another example for high order mutation on a different graph, same graph as in figure 6.4

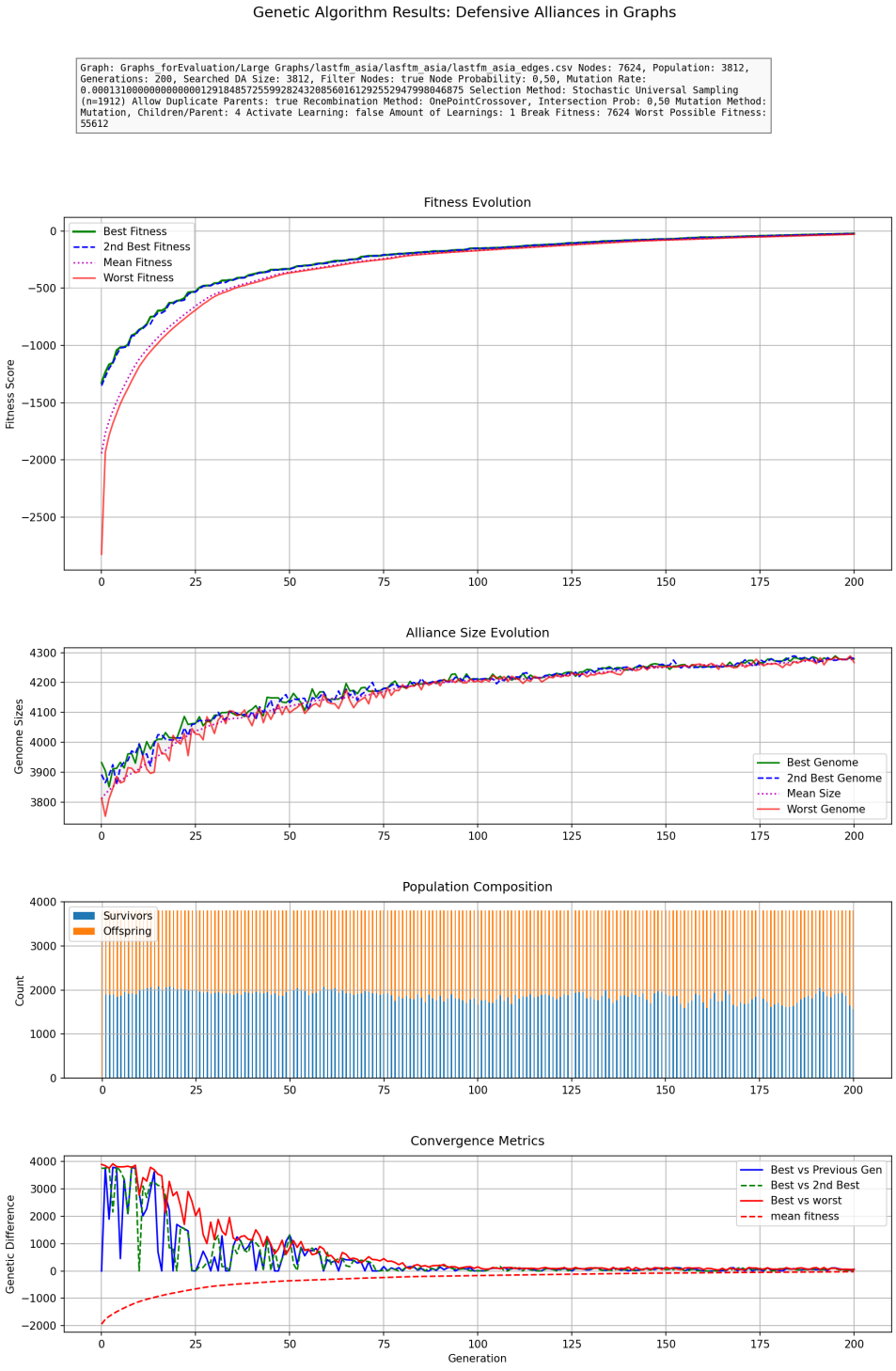

Figure 6.4: Another example for high order mutation on a different graph, same graph as in figure 6.3

6.2.1 Observations

By applying the two different mutation techniques to graphs of different sizes and with vastly dissimilar transitivity, we observe that the proposed mutation operator converges to local optima faster while simultaneously lowering the symmetrical difference of the offspring. This ultimately results in fewer offspring being accepted by the plus size strategy. We therefore evaluate the mutation favoring vertices of higher order as an ineffective and worse option compared to the basic mutation. The premature convergence of the mutation with higher order has in all tests never resulted in the discovery of a defensive alliance after higher generations, unlike the basic mutation (see Figures 6.1 to 6.4).

6.3 Learning

Learning is a process that is given as a reward to survivors of the previous generation. When we gave the time complexity results 5.6.2, we concluded a combination of parameters to maximize the iteration speed. Since learning is only given to survivors, we decided to make use of faster iterations to amplify the effect of learning for evaluation. We will again search for defensive alliance of size $k \in \{\frac{3}{4}|V|; \frac{|V|}{2}; \frac{|V|}{4}\}$ with the learning option enabled.

The computational advantage of learning decreases over time[7, p. 312], as a result, we know that preparing a population by calling the learn function upon initialization is the most effective variant of learning. We will not investigate this further. In our observation, we expect diminishing returns of the learning function as the generation count increases.

The Learning function was tested by uncapping the amount of learners and enabling the learn option, thus resulting in the learning function called on every survivor. Each survivor first removed the l most harmful node, then added a node that increased its fitness score, with $l \in \{1; 10; 100\}$, depending on the algorithm run. Learning upon initialization is disabled. Also note that the learning function is only applicable for the first phase of finding defensive alliances.

The basic parameters are as follows:

- Population size $|P|$ is set to $\frac{|V|}{4}$
- The initial population will be inhabited by chromosomes that have for each locus a probability of 0.5 to set to 1. The expected average size of a chromosome is therefore $\frac{|V|}{2}$.
- All nodes that can not be inside a defensive alliance of size k are filtered out
- Stochastic Universal Sampling will be used for selection, allowing duplicates
- The amount of parents selected to participate in the recombination process is set to $\frac{|P|}{4}$
- The number of children per pair of parent is set to 4. This results in a total amount of $|P|$ offspring each iteration

- The standard mutation rate is set to $\frac{1}{|V|}$
- The standard basic Mutation is used as mutation operator
- The algorithm stops when a global optimum is found or the generation limit of 300

6.3.1 Observation

The referenced Figures figs. 6.5 to 6.8 all show that the learning function has only a small impact, when used with small l relative to $|V|$. The order of the graph given in 6.6 has an order of 7624, and the graph given in 6.5 has an order of 9498. The main difference between these two graphs is the transitivity of 0.047 for 6.5 and 1.787 for 6.6.

Therefore, they act as valuable references for the evaluation of the effect the learning function has. In these example there is no significant difference between the learning amount of 1 and 10 observed. When it comes to $l = 100$, the effectiveness of the learning function increases drastically. A possible reason for this might be that candidate solutions that are close to becoming a defensive alliance are not exploiting the learning function enough, by removing not remaining all remaining harmful nodes or simply failing to add the extra needed nodes.

We observe that the essential last learning of a genome to become a defensive alliance is done more efficiently and more effectively when the learning amount is higher. The tests indicate that when learning to survivors is applied, it should be in relatively high amounts.

6.4 Evaluating the Handling of defensive alliances

The previous sections were more focused on the problem of finding a defensive alliance, now we evaluate their handling. We do not call any additional learning methods on defensive alliances. Once a defensive alliances has been discovered we search for its connected components to add them into the population. If the population size is small, the effect of adding connected components to the population only impacts the search for a defensive alliance marginally. The effect is even worse, when we search for larger defensive alliances of size k. The reason for this lies in the outsourcing of smaller connected defensive alliances that have a worse fitness score assigned to them by default. These defensive alliances fail to join the population and thus it remain populated by mainly larger defensive alliances. As a result, the subset sum problem most likely has no solution in the population.

6.5 Results and Discussion

We determine the learning function to be highly effective for early generations, even when the learning amount is set low. This statement aligns with [7, p. 312]. We also see that the learning function fails to break through and evolve a subset into a defensive alliance when the learning amount is set too low. Our test indicates that the learning function reliably transforms individuals with a higher fitness score in

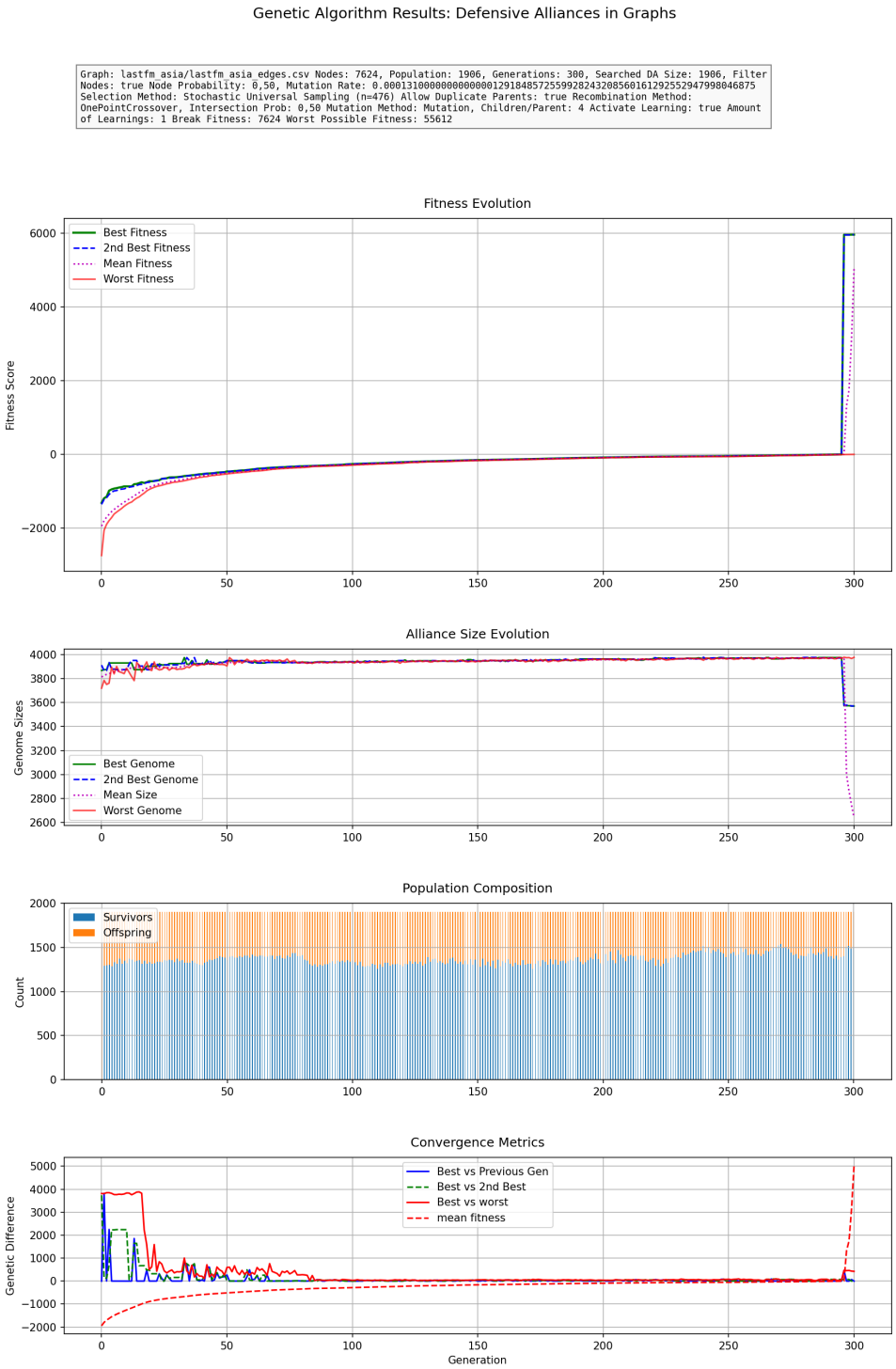

Figure 6.5: Learning amount of 1 on a graph with high transitivity

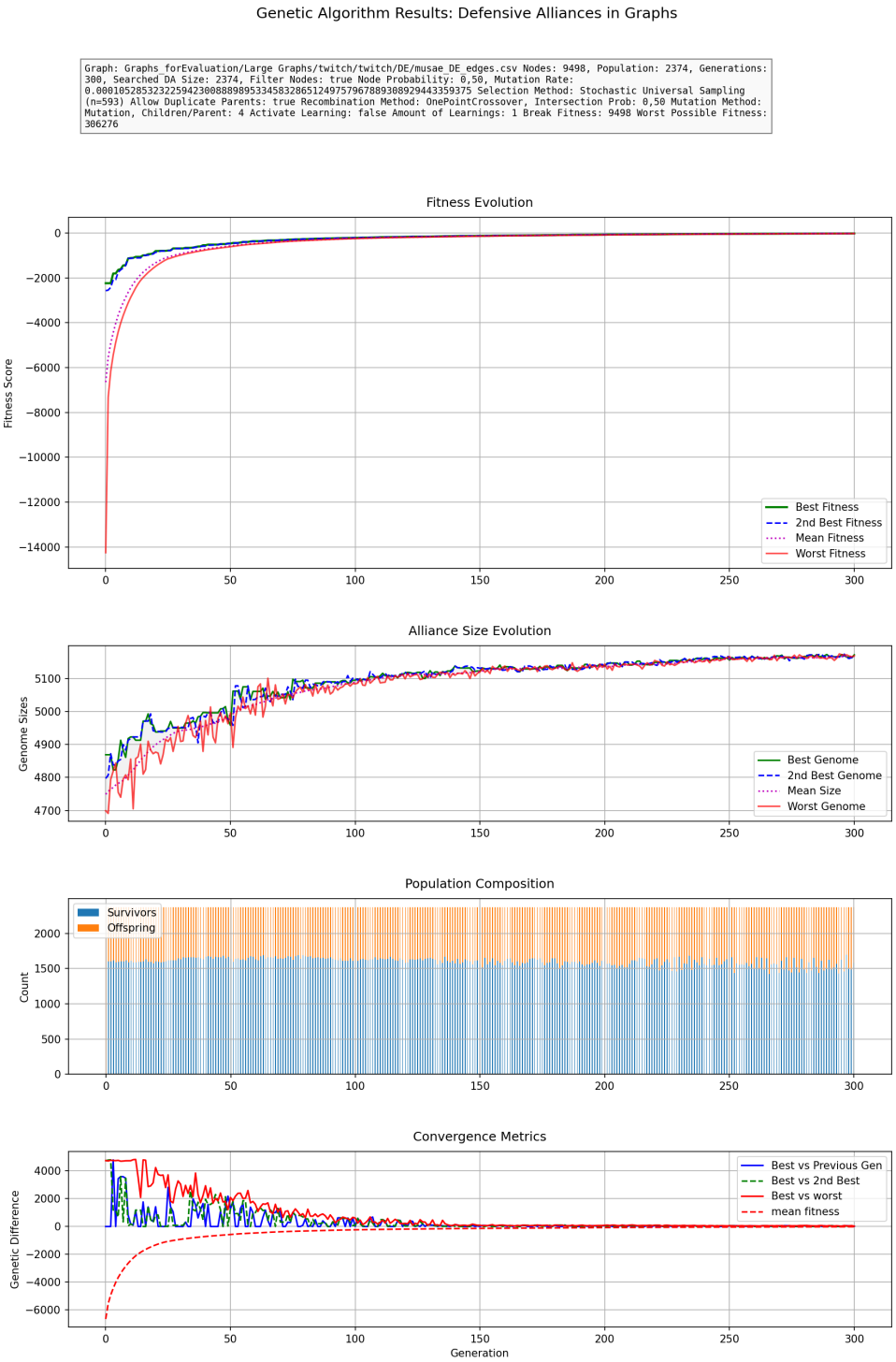

Figure 6.6: Learning amount of 1 on a graph with low transitivity

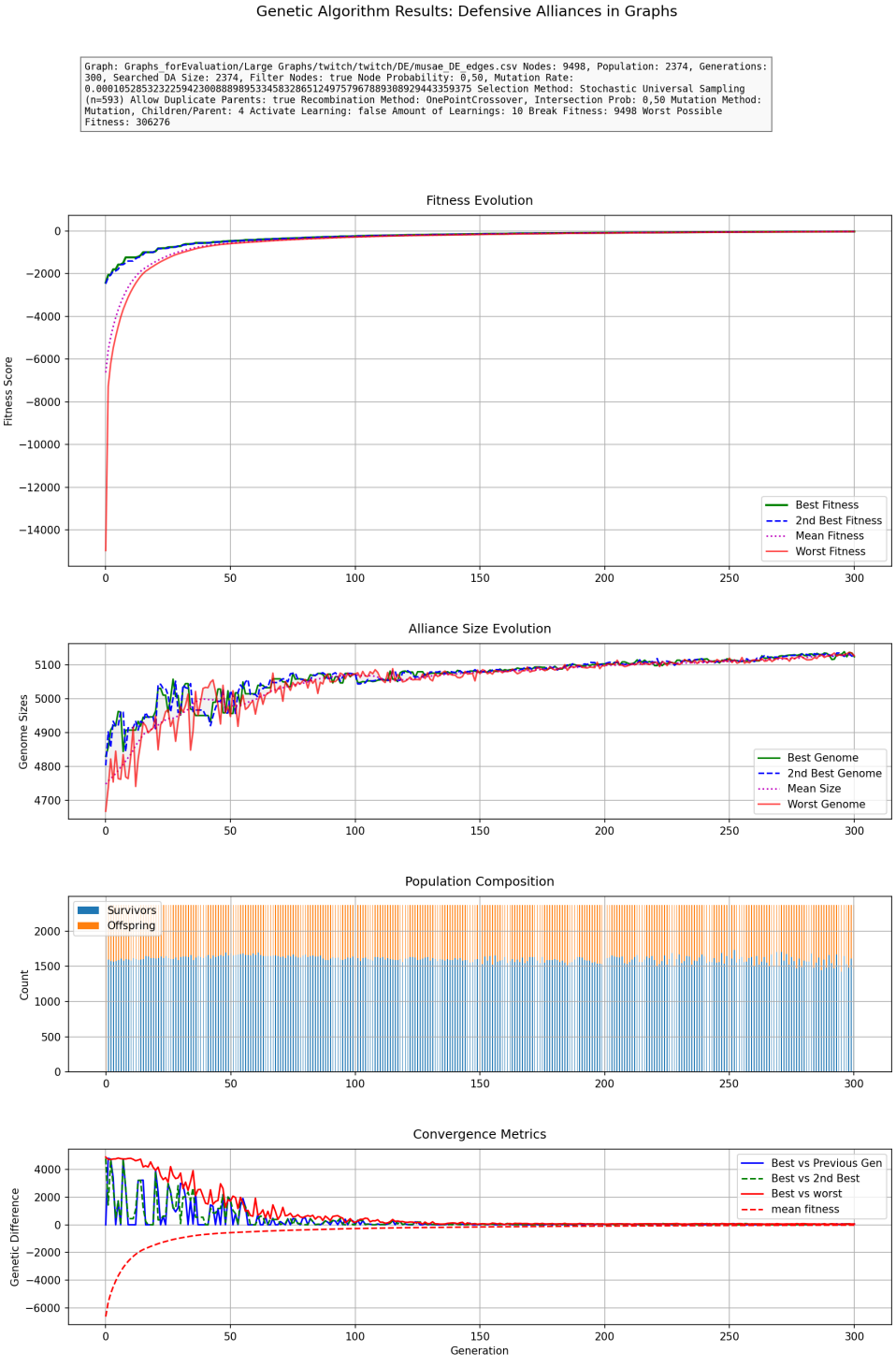

Figure 6.7: Learning amount of 10 on graph with high transitivity (same graph as in 6.6

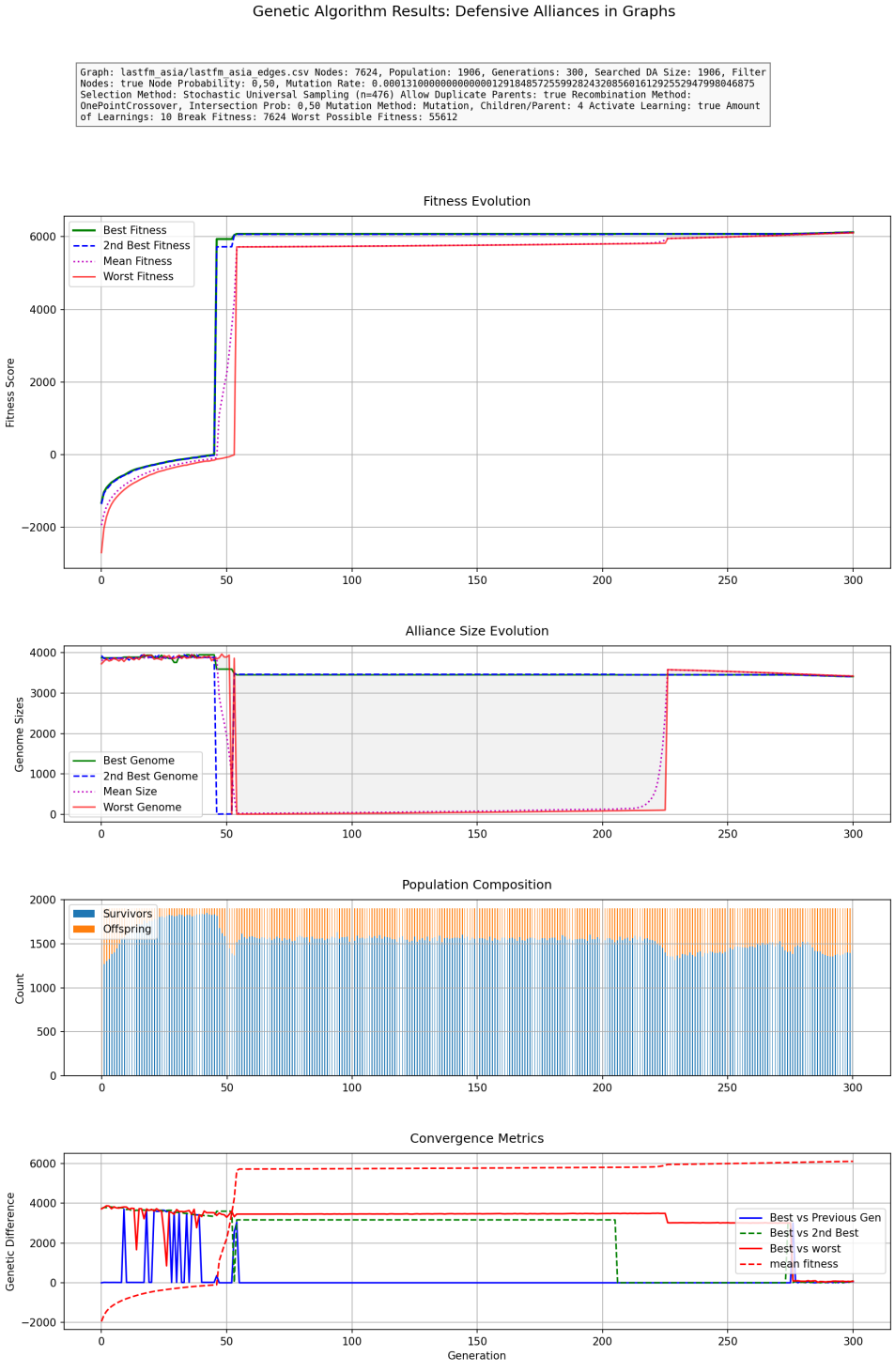

Figure 6.8: Learning amount of 10 on graph with low transitivity (same graph as in 6.5

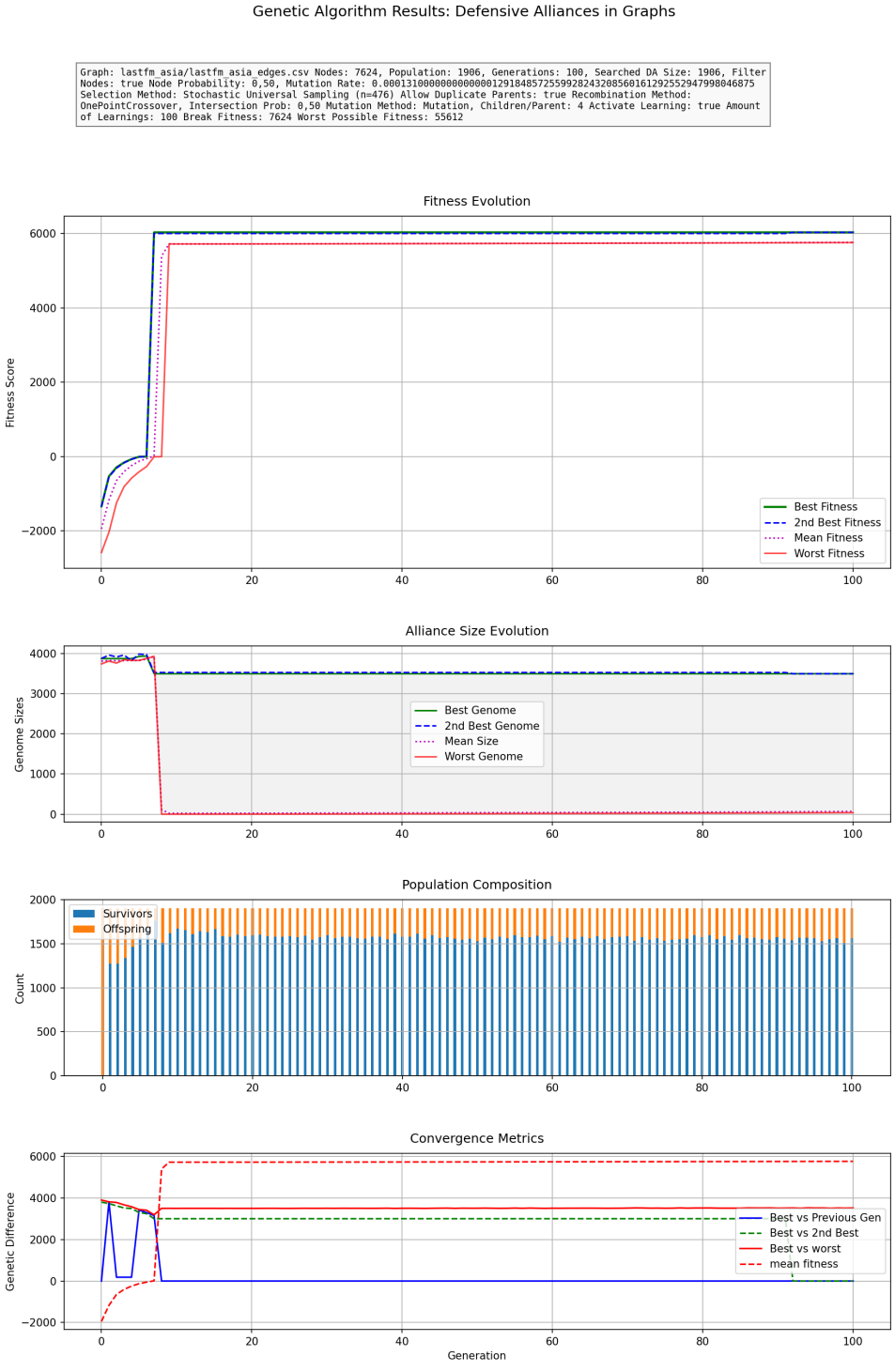

Figure 6.9: Learning amount of 100 on graph with high transitivity (same graph as in 6.5 and 6.8

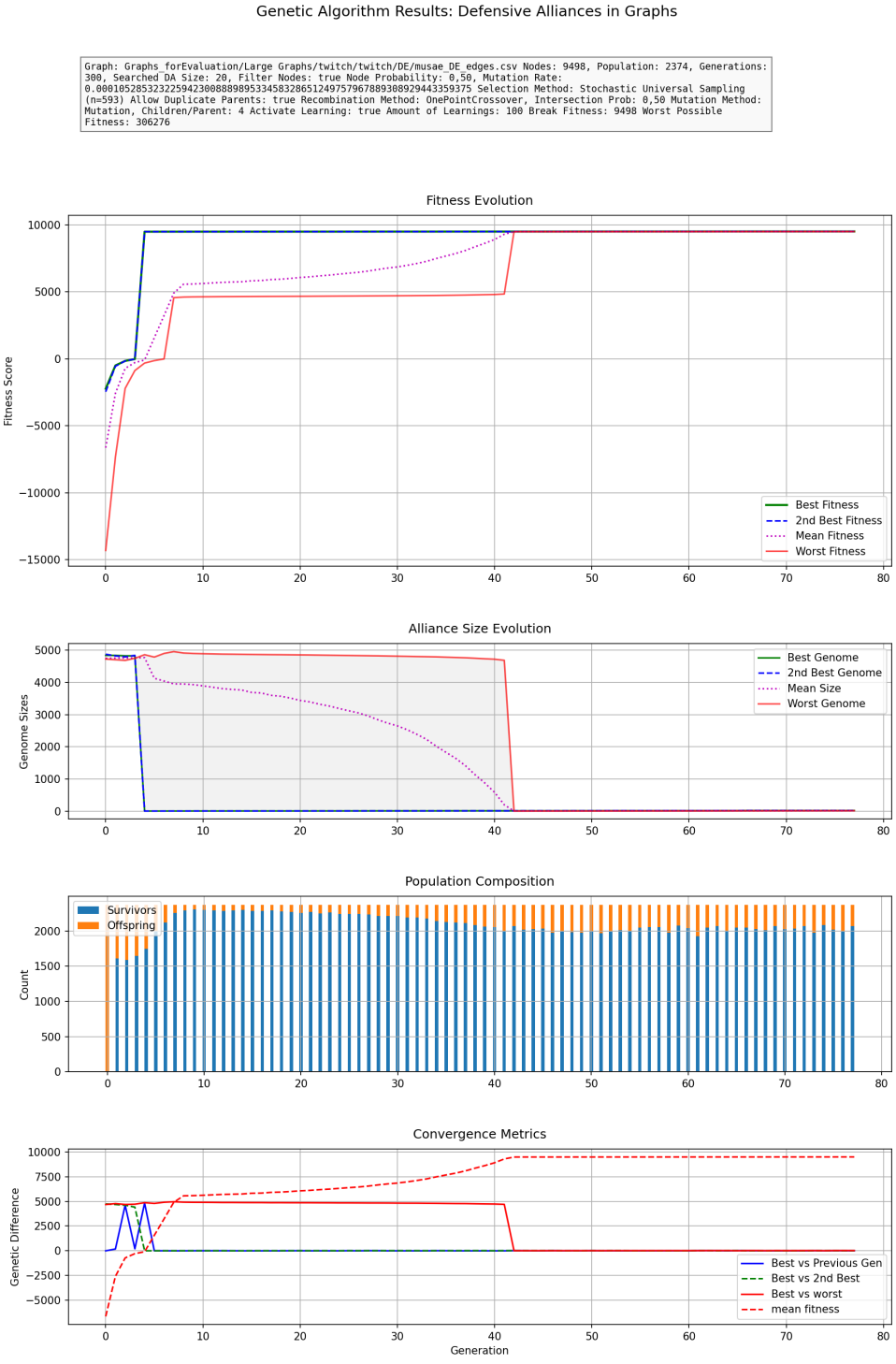

Figure 6.10: Learning amount of 100 on graph with low transitivity (same graph as in 6.6 and 6.7

relation to becoming a defensive alliance to defensive alliances if the learning amount is set high.

Furthermore, the algorithm lacks in solving the subset sum problem for higher k, thus is unsuited for finding defensive alliances of size k (see figs. 6.1, 6.2, 6.7, 6.9 and 6.10). The test cases suggest that the problem of finding a defensive alliance should possibly be separated even more so than it is done here. They should be solved separately. The main problem lies in the opposing needs in parameter selection between the two problems. The efficiency of the search for defensive alliances is heavily dependent on the amount of offspring generated and the order of the graph. The computational time grows quickly when increasing those parameters, thus should be avoided. To find defensive alliances of a certain size k on the other hand, we need a large population size that stores all unique connected components with a cardinality lower k of all defensive alliances in a given generation, and then those have to be recombined quickly and often, until a solution for the subset sum problem with k is discovered. When trying to solve both problems at once, we are left with the worst case scenario for both. Also, the proposed optimized mutation parameter that dynamically changed its mutation rate depending on the degree of a locus, has been identified as failure. It leads to premature convergence in all test cases and reduces diversity, thus being unsuitable in combination with the plus strategy. The restriction made by outsourcing all $\deg(v) \geq 2k+1$ is rational and does not cost any reasonable computation time. Therefore, it should always be made. That being said, it has in many cases no effect when searching for greater k in relation to the order of a graph because there often are no such vertices, especially in graphs with lower transitivity.

Despite all of these negative results we need to encourage the fact that this algorithm is really efficient in discovering defensive alliances consisting of vertices, for that $\deg(v) < 2k + 1$ holds true, when given the right parameters. The algorithm was developed to be flexible and applicable to all simple undirected graphs. This flexibility made room for suboptimal parameter configurations, but the algorithm should be defined by them.

BREAK_FITNESS when set to 0, stops the loop of the genetic algorithm, once a defensive alliance has been found, effectively reducing the algorithm to the first part only. Therefore, the algorithm still is applicable when one only desires to search for defensive alliances with the restriction of not including any nodes that can not be part of any defensive alliances of size k. As a result, it can still be used to build an initial population to solve the subset sum problem.

We conclude that our proposed algorithm efficiently discovers defensive alliances on a diversity of graphs, when the correct parameters are set, but is inefficient in discovering defensive alliances of size k for smaller k and ineffective in discovering them for larger k.

6.6 Future Work

This work only addresses the problem of finding (-1)-defensive alliances. The algorithm can be generalized to solve the r-alliance problem, by only altering certain sections slightly, such as the fitness function. The learn function that removes harmful nodes would be altered as a result. With this algorithm, further research to find

k-defensive alliances in graphs can be promoted. This algorithm can also be applied to investigate the optimal configuration, if there is one for discovering defensive alliances in graphs, due to its adaptability.

Bibliography

[1] Petter Kristiansen, Sandra Hedetniemi, and Stephen Hedetniemi. Alliances in graphs. 48.

[2] Ismael González Yero and Juan A. Rodríguez-Velázquez. Defensive alliances in graphs: a survey. version: 1.

[3] Lindsay H. Jamieson, Stephen T. Hedetniemi, and Alice A. McRae. The algorithmic complexity of alliances in graphs. 68(137).

[4] Ajinkya Gaikwad, Soumen Maity, and Shuvam Kant Tripathi. Parameterized complexity of locally minimal defensive alliances. 372:324–340.

[5] Ajinkya Gaikwad and Soumen Maity. Globally minimal defensive alliances. 177:106253.

[6] Richard M. Karp. Reducibility among combinatorial problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg.

[7] Evolutionary computation 2: Advanced algorithms and operators.

[8] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. 1(1):67–82.

[9] T. El-Mihoub, A. A. Hopgood, L. Nolle, and A. Battersby. Hybrid genetic algorithms - a review. 13(2):124–137. Publisher: Nottingham Trent University.

[10] Ajinkya Gaikwad and Soumen Maity. Globally minimal defensive alliances: A parameterized perspective.

[11] Rodolfo Carvajal, Martín Matamala, Ivan Rapaport, and Nicolas Schabanel. Small alliances in graphs. In Ludk Kuera and Antonín Kuera, editors, *Mathematical Foundations of Computer Science 2007*, volume 4708, pages 218–227. Springer Berlin Heidelberg. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[12] Ajinkya Gaikwad and Soumen Maity. Defensive alliances in graphs. 928:136–150.

[13] J. M. Sigarreta and J. A. Rodríguez. On defensive alliances and line graphs. 19(12):1345–1350.

[14] Hasan Kharazi and Alireza Mosleh Tehrani. On the defensive alliances in graph. 8(1):1–14. Publisher: University of Isfahan.

[15] Kahina Ouazine, Hachem Slimani, and Abdelkamel Tari. Alliances in graphs: Parameters, properties and applicationsa survey. 15(2):115–154.

[16] Juan A. Rodríguez-Velázquez, Ismael G. Yero, and José M. Sigarreta. Global r-alliances and total domination. 22(1):96–100.

[17] Cristina Bazgan, Henning Fernau, and Zsolt Tuza. Aspects of upper defensive alliances. 266:111–120.

[18] Lindsay Jamieson. Algorithms and complexity for alliances and weighted alliances of various types.

[19] Jose M Sigarreta. Upper k-alliances in graphs.

[20] Henning Fernau. Global r-alliances and total domination.

[21] Teresa W. Haynes, Stephen T. Hedetniemi, and Michael A. Henning. Global defensive alliances in graphs. pages R47–R47.

[22] Ahmad Hassanat, Khalid Almohammadi, Esraa Alkafaween, Eman Abunawas, Awni Hammouri, and V. B. Surya Prasath. Choosing mutation and crossover ratios for genetic algorithmsa review with a new dynamic approach. 10(12):390. Publisher: Multidisciplinary Digital Publishing Institute.

[23] J.-M. Renders and S.P. Flasse. Hybrid methods using genetic algorithms for global optimization. 26(2):243–258.

[24] Alan Piszcz and Terence Soule. Genetic programming: optimal population sizes for varying complexity problems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 953–954. ACM.

[25] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. 3(2):124–141.

[26] Tansel Dokeroglu, Ayça Deniz, and Hakan Ezgi Kiziloz. A comprehensive survey on recent metaheuristics for feature selection. 494:269–296.

[27] Christian Blum, Jakob Puchinger, Günther R. Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. 11(6):4135–4151.

[28] Tzung-Pei Hong, Hong-Shung Wang, Wen-Yang Lin, and Wen-Yuan Lee. Evolution of appropriate crossover and mutation operators in a genetic process. 16(1):7–17.

[29] S. Gotshall and B. Rylander. Optimal population size and the genetic algorithm.

[30] T. Weise, Yuezhong Wu, R. Chiong, K. Tang, and Jörg Lässig. Global vs. local search the impact of population sizes on evolutionary algorithm performance.

[31] Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. Global versus local search: the impact of population sizes on evolutionary algorithm performance. 66(3):511–534.

[32] Thomas Baeck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press. Google-Books-ID: 4HMYCq9US78C.

[33] Chun-Wei Tsai, Shih-Pang Tseng, Ming-Chao Chiang, Chu-Sing Yang, and Tzung-Pei Hong. A high-performance genetic algorithm: Using traveling salesman problem as a case. 2014:178621.

[34] Lars Nolle. Hybrid genetic algorithms: A review.

UNIVERSITÄT TRIER

Eigenständigkeitserklärung

Hiermit versichere ich,

Name, Vorname _____

Matrikelnummer _____

dass ich die vorliegende schriftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken und Inhalte als solche kenntlich gemacht habe. Die beigefügte Arbeit habe ich bisher nicht zum Erwerb eines anderen Leistungsnachweises eingereicht und keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Ich versichere, dass ich auf generativer Künstlicher Intelligenz (genKI) basierende text- oder sonstige inhaltgenerierende Hilfsmittel nur auf die durch die Prüferin/den Prüfer schriftlich gestattete Weise verwendet und genKI generierte Passagen explizit gekennzeichnet habe.
Bei Verwendung von genKI-Tools habe ich diese mit ihrem Produktnamen angegeben und in einer Übersicht vollständig aufgeführt.

Ich verantworte die Übernahme der von mir verwendeten genKI generierten Passagen und Inhalte in meiner Arbeit vollumfänglich selbst.

Titel der Arbeit _____

Studienfach _____

Prüferin/Prüfer _____

Es handelt sich um eine Leistung für eine

- ☐ Bachelorarbeit
- ☐ Masterarbeit
- ☐ sonstige Prüfungsleistung für Modul (Hausarbeit etc.) _____

_____ _____

Ort, Datum Unterschrift

Stand: 06/2025