UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Graph-aware Evolutionary Algorithms for Influence Maximization in Social Networks

Supervisor

Prof. Giovanni Iacca

Student

Kateryna Konotopska

Academic year 2018/2019

Thanks

I am thankful for the opportunity of working with Prof. Giovanni Iacca. His guidance was priceless and his energy and engagement inspired me during the whole process of the thesis work.
I am so much grateful to my mother, Halyna, who made many sacrifices in life to give me the opportunity to study in such a great country as Italy. The Computer Science degree means for me not only professional growth, but, first of all, personal. It refined my individual qualities and set up my way of thinking, so I can not thank my mother enough for the gift she made for me.
I also want to thank my family and friends, Vinicio, Liliana, Mauro and my brother Borys, for their support, which was always positive and happily available, despite my shortcomings.
Finally I want to show gratitude to my treasured boyfriend Jacopo, who is my sweetheart but also my best friend, who I can trust for any decision or advice. Thank you for being patient with me during these last months and for sharing with me dreams about future, which give me strength in the most desperate moments.

Contents

List of Acronyms

GA	Genetic Algorithm
EA	Evolutionary Algorithm
IM	Influence Maximization
IC	Independent Cascade
WC	Weighted Cascade
MAB	Multi Armed Bandit
UCB	Upper Confidence Bound

Abstract

The relevance of social networks in modern society needs no introduction. We spend a vast amount of time while scrolling the news feed and checking the updates of people we are connected with, their opinions and actions, their way of thinking, their goods are a source of inspiration and reflection for us, they affect our thoughts and actions. A generic advertisement is way less powerful when compared to a suggestion from one of our friends or people we are connected with.

Therefore, in the last years social networks became an important source of information transmission. The information diffusion success hugely depends on the influence of people who disseminates this information. Modern marketing often involves paying some targeted users for advertising products or ideas. Here is our problem: how can we find people who lead to the highest information spread?

In this thesis we work on the influence maximization problem in social networks, known to be NP-hard, using a genetic algorithm, extended with graph domain knowledge. We deal with two different aspects of the problem: scalability and networks structure diversity. GAs are known to scale poorly with the graph size and we tried to tackle this aspect by using approximated fitness functions and search space reduction, while for the further aspect we introduced a combination of GA improvements, such as smart initialization and a combination of domain-aware mutation operators with an adaptation technique, so that the most successful mutation is used in different evolution phases and for datasets having different connectivity.

1 Introduction

Online social networks became an integrative part of the daily life of the majority of people. They are used to maintain contact with persons, reading news, search a job or buying new things. In particular all of the listed things can be done by looking what other people say about the products, jobs and events. This is the key difference between searching information on the web versus doing the same thing by means of social networks.

When the activity of a user **a** implies a probability of a supporting action of the user **b**, there is an edge **a** $\rightarrow$ **b** in the graph representing the social network. This may be, for example, one user supporting another during elections, or when buying product by the user **a** may imply buying the same product by the user **b**. This model can be also used for virus spread modeling: in this case we have the probability of the user **a** to infect the user **b**.

In general, **a** $\rightarrow$ **b** means information spread from the node **a** to the node **b** with a certain probability. In case of success, the node **b** is said to be activated by the node **a**. Different spread probability models have been studied by social scientists, called *propagation models*.

When we need to know which set of nodes has more chances of having the maximum possible number of activations we have an Influence Maximization (IM) problem. Usually, we have a budget on the number of nodes we can initially activate, this may correspond to the number of users we can afford to pay for advertising a product. It is also common to have a limit on the runtime for finding a solution, but for now let's assume we just want it to be as fast as possible.

Being proved to be NP-Complete, IM is still an open problem. It is a combinatorial problem, so we may need to evaluate all the possible solutions in order to find the optimal one, which makes this problem particularly tough for large real-world graphs.

We applied genetic algorithms (GAs) to the IM problem, with the aim to better adapt the algorithm to the problem domain, by introducing domain specific operators. We have also preliminarly studied different ways of computing influence spread (our fitness function) which can easily become a bottleneck when the algorithm requires a high number of individuals evaluations. We show that with the additional convergence speedup improvements, GA is competitive with the state-of-the-art heuristics, while providing a number of different solutions with equivalent influence spread.

This work is organized as follows: it starts with the background to provide the information needed to have a "common ground" when later discussing the problem's details, the following second section provides the problem statement with an overview of the state of the art approximation algorithms with provable guarantee and some heuristics, the next two sections contain descriptions of the EA design and the experiments we have done. Finally, a conclusions section summarises the results of our experiments and possible future work directions.

2 Background

This chapter serves as an introduction containing the necessary prior knowledge essential to understanding the problem we try to solve and the proposed solution. We will start by introducing intractable problems, the following sections will help us to figure out how to deal with them: how to define them in a way that is comprehensive to a machine, and which are the algorithms that can find an approximate solutions in a reasonable time.

2.1 Intractable problems

Let's start by defining what does a problem mean: we have a question, with some assumptions, which can be codified as *input variables*, and we want to know the answer, the *output*. The *algorithm* is a sequence of steps for solving problems, which, in our almost common knowledge, can be written in a specific programming language and executed by a computer. Obviously, the time or resources at our disposal are limited, and we would like to have the most *efficient* algorithm that solves our problem. The efficiency of an algorithm is usually measured by the resources it requires, e.g. computation time or space.

Generally, when we change the input variables, we would also like to have the same algorithm solving different *instances* of the problem while involving a limited amount of resources. The *size* of the problem can be described as the amount of input data used to define the problem. The *time complexity function* describes how the required computation time scales with respect to the input size.

When the relation between the problem size and the computation time can be described with a polynomial, we are talking about a *polynomial time algorithm*, otherwise we have an *exponential time algorithm*.

The problem is said to be *intractable*, when no possible polynomial time algorithm can solve it, so we can also say that there is no *efficient* solution for the problem.

2.1.1 NP-complete and NP-hard problems

For a big number of the most common real-life problems no one has been able to find a polynomial-time algorithm, nor proved that they are intractable. What else do we know about them and do all these problems have the same complexity?

Let's first see which problems are *provably intractable*. We can divide them in three different categories. The first one includes the problems with an exponential output, so they are intractable by construction since we would not ever be able to read the unbounded length output.

In the second class we find problems proved to be not solvable by *any* possible algorithm, they are known as *undecidable* problems.

The last group contains all the decidable problems proved to be impossible to solve with the use of a *non-deterministic computer*. This is a kind of machine which can use an unbounded number of parallel computations.

A large amount of previously mentioned apparently intractable problems can be solved by a non-deterministic machine (and thus are decidable), nevertheless there still doesn't exist any proof technique able to confirm their intractability.

We can, however, distinguish some relations between these problems: by solving some of them we automatically have solutions for some others. A problem **A** is said to be *reducible* to the problem **B**, if we can construct from any instance of problem **A** a correspondent instance of problem **B**, such that once we have solutions for the problem **B**, by using this construction we can also have solutions for the problem **A**. If the construction algorithm is polynomial time, the polynomial solutions for **B** would be still polynomial for **A**.

Problems which can be solved by a *N*on-deterministic computer in *P*olynomial time, are also known as *NP*-class problems. We already anticipated that some NP problems can be reduced to others, but is there a problem such that we can reduce *any* problem in NP to it?

The answer is yes, actually, there exist a family of them, these problems are known as *NP-complete* problems. Once we efficiently solve at least one of them, we will find the solutions for all the others. That's why

if we prove a problem to be NP-complete, we know that it would be probably very hard to find an efficient algorithm that solves it, since it would be the solution for a plethora of very well known tough problems.

"I can't find an efficient algorithm, but neither can all these famous people."

Figure 2.1: Illustration from [8] gives an insight on the relevance of the NP-complete problems.

There also exists a more wide class of problems to which we can reduce any problem in NP, but not necessary belonging to NP themselves, they are called *NP-hard* problems. NP-hard problems are at least as hard as NP-complete problems.

Another interesting example of problems we will discuss later in this work are enumeration problems: how many solutions exist for a given problem? Enumeration problems which can be efficiently solved with a non-deterministic computer belong to a class of problems called *#P*, said "number-P" or "sharp-P". Some of these problems can be easily solved in polynomial time, while the others, for example those involving counting solutions of NP-complete problems, are NP-hard. The notion of completeness in this case is again referred to the hardest problems in the class. A #P problem is called #P-complete, when every other problem in #P can be reduced to this problem in polynomial time.

2.2 Optimization

If we are not only interested in any admissible answer, but would like to have the best solution with respect to some measure, we are most likely looking for an *optimal* solution in an optimization problem.

Usually this measure can be described by a function ***f(x)*** that assigns a score to each solution **x**, called the *objective function* or *fitness function*, and we want to find the global minima or global maxima of this function.

Based on the kind of input/solution that we are working with, we can distinguish different types of optimization, specifically, when **x** are discrete variables we will be doing *discrete optimization*, while when the **x** variables are continuous we use *continuous optimization*.

If we have no constraints on the values assumed by the variables we are working with, we are talking about *unconstrained optimization*. And, on the contrary, *constrained optimization* deals with the optimization of a fitness function while respecting some constraints on **x**.

When talking about the number of functions to optimize, we can have single and multiple functions, corresponding to *single-objective* and *multi-objective* optimization.

We can also distinguish two kinds of optimization according to the runtime requirements. If we need solutions to be provided in a quick and on demand manner we should employ *online* optimization, where we trade optimality for velocity, while for situations where we can wait for hours or days for a solution the so called *offline* optimization can be adopted.

Generally we have a function or a set of functions to optimize and a set of equality and/or inequality constraints:

$$\begin{aligned} & max_x f(x) \\ subject\ to\ & h_i(x) <= 0, i = 1..n \\ & g_j(x) = 0, j = 1..m \end{aligned}$$

The $max_x f(x)$ corresponds to the point which has the highest objective function in the specified domain. Points surrounded only by areas with lower objective function values are called local maxima. Each local maxima is a candidate solution for our optimization function.

When we can make use of calculus to find the exact solution, in cases when the function is not too complicated, finding a global minima or a global maxima can be an easy problem, whereas when we are constrained to involve iterative methods, the upper bound on the time to find the global optima cannot be guaranteed in the general case.

As we will see, when we deal with intractable problems, we are generally bound to make use of *approximated algorithms*, which can, hopefully, provide some guarantees on the maximum error on the provided solution. Thus finding the best approximation algorithm can be seen as an optimization problem itself, where we want to minimize the distance from the optimal solution (and of course the running time).

2.3 Heuristics and metaheuristics

When finding the optimal solution is intractable, or it is a hard problem (for example due to the extreme complexity of the objective function), it may be practically efficient to use methods which would provide a reasonable result according to some logic. These kinds of algorithms are usually called *heuristics*, they generally use the information previously gathered by the algorithm to produce the next candidate solution, and they usually do not provide any guarantees about the optimality of their solutions. Heuristics can be broadly defined as ad-hoc and non-optimal algorithms for specific classes of problems.

A *metaheurisitic* is a combination of heuristics in one common procedure, useful for solving a more general class of problems. Metaheuristics usually perform *black-box* optimization, when we don't have an algebraic expression of the objective function or, more generally, we do not make any assumption on the function to optimize.

2.4 Bio-inspired solutions

Bio-inspired metaheuristics are probabilistic algorithms that mimic biological evolution mechanisms adapted from Darwinian evolution theory. According to this theory, population evolves across different generations, and, due to natural selection, only the fittest individuals (or more adapted to the environment) survive and are capable to reproduce and transmit their genes to the next generations.

This biological thesis stands on the four pillars: population, diversity, heredity and selection. That basically means that by using an evolutionary algorithm we must have at least two individuals (population), which must be different among them (diversity), some individual's genes can be transmitted to the offspring (heredity) and only a part of the offspring can reproduce (selection).

In practice, evolutionary algorithms are population-based algorithms which have multiple search directions, where each *individual* is a candidate solution, which should be described by it's *genotype* (genetic information). The genes are randomly changed by mutations, the fittest individuals are selected by the means of *fitness function*, which determines their suitability for the environment, which can be seen as a representation of the objective function we want to optimize (and vice-versa). The strongest individuals reproduce by recombination of their genes (crossover), producing new candidate solutions.

2.4.1 Components of Evolutionary algorithms

In this subsection we will discuss different aspects of evolutionary algorithms, which must be specified/chosen and which together concur to form the final EA algorithm.

Individuals representation

In order to encode a candidate solution or a real-world object, called *phenotype*, to use it in the EA, we must define its genetic information, the *genotype*. The mutations and recombinations are thus performed on genotype, while the fitness of the individual is evaluated by considering its corresponding phenotype.

Representation specification is the first design step, and the next steps will depend on how the genotype is encoded. A good representation should be *complete* and *accurate*: the genotype space should contain the optimal solution and all the feasible candidates, which permit to find the optimum; different solutions should have an opportune accuracy and encode only relevant parameters.

The concrete representation structure depends on the nature of the problem: it could be, for example, a sequence of discrete or real-valued values, a tree, or whatever structure you need to better encode the candidate solutions.

Fitness function

The evaluation function should represent the task to solve, by assigning a quality measure to the genetic candidate representation, preferring individuals that are evaluated better should lead to individuals that are more likely to solve the task, or that are better at it. It usually involves the conversion of the individual from the genotypic to phenotypic space and evaluates the ability of the phenotype to adapt to the environment.

Population

The population of a EA contains a set of possible solutions, or, more precisely, their genotypic representations. Given the individual representation, to define the population it may be sufficient to set the number of individuals, or, in some more sophisticated approaches it may have some spatial structure, with the concept of distance represented by the using distance functions among individuals.

The population is usually defined by randomly sampling the search space, but it may happen that defining the initial population in a smart way could lead to substantial improvements in the end results, especially in terms of convergence and thus computation time.

Such informed initializations are based on domain specific knowledge that allows to obtain somewhat good individuals without having to go through costly optimization steps. The initialization must obviously not introduce a strong bias in our population (e.g. by only considering a very niche neighbour of samples), which would most likely incur in not so optimal results.

The *diversity* of a population is an important factor that should be considered, as it might signal a premature convergence and thus the need for a restart or some kind of stimuli to our population. Different heuristics to measure diversity exist and, as usual, the right one depends on the problem at hand and our needs.

Selection (of parents)

The selection is used is to assign the largest part of offspring to the fittest individuals. The percentage of the individuals which are discarded during this process is called selection pressure. A high selection pressure may produce rapid fitness increase, but it can also lead to a high risk of diversity loss and premature convergence.

Mutation and crossover

Genetic operations catch the impacts of biological transformations on the genotype.

Crossover is a stochastic operator that ensures that the offspring captures the genotypic information from both parent individuals. It is applied to a randomly paired couple of candidates with a given probability p_c. Usually it involves swapping parts of the genotypes in randomly selected points. Recombination of individuals permits to exploit the search space by combining parents genetic components, usually locals search is performed by means of crossover.

Mutation is applied to the offspring, and it is usually characterized by a certain probability p_m of changing (a part of) the genotype. It makes small changes to the individual in order to reach new solutions. The mutation operator permits the exploration of the search space, so its design should permit to reach all the possible candidates in the genotype space. The mutation operator is associated with global search.

3 Influence maximization

In this chapter we will introduce the problem of influence maximization in social networks and some of the most popular approaches used to solve it, also a quick overview of the problem's variations will be provided.

3.1 Problem statement

The Influence maximization problem is a problem of finding the set of nodes that will lead to the maximal number of nodes activations according to some model.

Definition 3.1. (Influence maximization) Given a network, represented with a graph *G*, an information propagation model *m* and a budget *k*, find a set of k initially active nodes (users), such that the influence spread is maximized.

3.1.1 Diffusion models

Influence propagation in a graph representing a social network is usually modeled according to a *diffusion model*. In this work we used the Independent Cascade (IC) and the Weighted Cascade (WC) models for the aforementioned purpose.

Independent cascade

In this model, at each time t a node can be either in an active or inactive state, a node can transit from inactive to active, but never the other way. At the beginning, only the nodes of the initial seed set S are in the active state, and at each next step each active node *u* has a chance of activating its neighbouring nodes *v* (if the edge $u \rightarrow v$ exists) with a probability p, which is the same for all the nodes in the graph and is given as a parameter. At a step t', if no more nodes are activated, the simulation stops. Influence spread is thus given by the number of active nodes.

Weighted cascade

Influence spread in this case is calculated in the same way as in the IC, the difference stands in the probability of node activation. The probability of activating a node v by one of its neighbouring nodes u is given by $\frac{1}{in-degree(v)}$.

3.2 Existing solutions and research directions

We can divide different existing approaches for the given problem into two main categories: approximation algorithms with provable guarantees and heuristics (and/or metaheuristics). The former solutions provide good results but their long runtime leads to the impracticality of their application. For this reason different solutions of the latter category have been designed. We will provide a short overview of some important works from the both approaches in order to have a better idea about what are the main key concepts that make these algorithms archive good results.

3.2.1 Approximation algorithms with provable guarantees

The problem was for the first time formulated as a combinatorial optimization problem under Independent Cascade and Linear Threshold models in [17] and proven to be NP-hard. The authors also proposed a greedy algorithm that yields $(1 - 1/e - \epsilon)$ - approximation.

The submodular property of the objective function was used in [20], the CELF algorithm proposed in this work is up to 700 times faster then naive greedy. This property was further explored in [13], their CELF++ has 35-55% gain in running time. The authors proposed also a heuristic advancement of CELF algorithm using path enumeration as objective function [12].

In [27] and [26] a new approach to calculate the nodes with the biggest spread was used. All TIM, TIM+ [27] and IMM [26] algorithms have the same concept in common : instead of computing the nodes influenced by the seed set in a traditional way, a function is used to find the nodes that have the biggest potential influence spread, which makes use of Reverse Reachability sets. Each Reverse Reachable set is obtained by taking the set of nodes reached by the information spread propagation from one node (one for each RR set) and by putting them in an hypergraph as an hyperedge. The solution is then calculated iteratively by selecting the node with the highest degree in the hypergraph and removing then that node with all its incident edges. The difference between the three algorithms consists in different ways of calculating the sufficient number of RR sets and the method used to sample nodes for their generation. All three algorithms outperform CELF [20], CELF++ [13]; IMM [26] outperforms both TIM and TIM+ [27].

BCT [23] is a further improvement of IMM applied for cost-aware targeted viral marketing, where each node has a benefit and a cost associated to it. It uses this information in order to first sample nodes with higher benefit for creation of RR sets. BCT is significantly faster then the IMM algorithm.

Table 3.1 contains a more detailed information about the experimental setup of the described papers and a short summary of each of them.

Paper	**Introduced concepts**	**Social model**	**Graphs**	**Experiments**	**Results**
[17]	Social influence maximization as a combinatorial problem, greedy algorithm proposed solution	WC, IC (p = {0.1, 0.01}), LT	Collaboration graph obtained from co-authorships in physics publications, 11K nodes, 53K edges	Comparison of the greedy algorithm with heuristics such as high degree, central and random; Monte Carlo simulation with 10.000 repetitions is used as spread function approximation	Greedy algorithm outperforms the other proposed methods, though in the case of IC with p = 0.1 the improvement is less evident
[20]	Greedy algorithm improvement, uses the submodular property of the objective for a faster choice of the next node to insert in the solution at each step	IC, LT	Blogs graph, 45K nodes and 1M edges; Water networks, 21K nodes and25K edges	The CELF's running time is compared to the other heuristics such as naive greedy	CELF is up to 700 times faster then naive greedy
[13]	Further exploiting of submodularity property of influence function,improved version of CELF	IC, LT	NetHEPT, 15K nodes and 32K edges; NetPHY, 37K nodes and 174K edges	Comparison with the CELF algorithm	CELF++ has the gain in running time w.r.t. CELF of 35 - 55%
[12]	CELF improved version, uses path enumeration as objective function	LT	NetHEPT 15K nodes and 62K directed edges; Last.fm 61K nodes and 584K directed edges; Flixster 99K nodes and 978K directed edges; DBLP 914K nodes and 6.6M directed edges	Comparison with high-degree, pagerank, CELF, LDAG	Simpath outperforms other heuristics in terms of running time

Table 3.1 continued from previous page

Paper	Introduced concepts	Social model	Graphs	Experiments	Results
[27], [26]	All TIM, TIM+ and IMM algorithms have in common the same concept: instead of computing in the traditional way nodes influenced by the seed set, they use another function to calculate nodes that have the biggest potential influence spread, which makes use of Reverse Reachability sets; each of those is obtained by taking the set of nodes reached by the information spread propagation from one node (one for each RR set) and by putting them in an hyper graph as an hyper edge. The solution is then calculated iteratively by selecting the node with the highest degree in the hypergraph and removing then that node with all its incident edges. The difference between the three algorithms consists in different ways of calculating the sufficient number of RR sets and the method used to sample nodes for their generation.	IC, LT	[27]: NetHEPT, 15K nodes and 32K edges; Epinions, 76K nodes and 509K edges; DBLP 655K nodes and 2M edges; LiveJournal 4.8M nodes and 69M edges; Twitter 41.6M nodes and 1.5G edges; [26]: NetHEPT 15.2K nodes and 31.4K edges; Pokec 1.6M nodes and 30.6M edges; LiveJournal 4.8M nodes and 69.0M edges; Orkut 3.1M nodes and 117.2M edges; Twitter 41.7M nodes and 1.5G edges	[27] Comparison with greedy, RIS [4], CELF++, IRIE, SIMPATH [26] Comparison with TIM, TIM+ and IRIE	TIM has the running time about two times smaller w.r.t. CELF++; TIM+ runs two times faster then TIM; IMM outperforms both TIM and TIM+. All three algorithms archive comparable influence spread
[23]	Improved version of IMM for cost-aware targeted viral marketing; improves computation time of the number of RR sets and samples first nodes with higher benefit	IC, LT, CTVM	NetHEPT 15K nodes and 59K edges; NetPHY 37K nodes and 181K edges; Enron 37K nodes and 184K edges; Epinions 132K nodes and 841K edges; DBLP 655K nodes and 2M edges; Twitter 41.7M nodes and 1.5G edges	Comparison with IMM , TIM/TIM+, CELF++ and SIMPATH on both CTVM and IM problem	BCT running time is up to 10 and 50times smaller then IMM/TIM/TIM+;

Table 3.1: Approximation algorithms with provable guarantee and some heuristics

3.2.2 Metaheuristics

The first application of evolutionary algorithms to the IM problem was proposed in [5], in which a simple GA without any domain knowledge outperformed the greedy algorithm [17] in run-time while obtaining comparable influence spread results. Since that a plethora of improvements were done in this direction.

A GPU parallelized GA version [28] outperformed the performance of the greedy algorithm given the same

execution time, taking up to 34 times less execution time to archive the same result.

In [30] different initialization strategies were compared, PageRank initialization gave the worst results. High PageRank was not correlated to the relevant according to GA nodes as well according to [24], where PageRank of the frequently selected by the GA nodes was studied. [10] also adopted a smart initialization approach, a similarity based high degree method was proposed; unlike standard high degree initialization it can guarantee a higher diversity of the individuals. The authors used two-hop spread [19] as approximated fitness function and similarity-based local search. Their results were quite promising, the proposed solution was up to 10 times faster then the CELF [20] algorithm while maintaining comparative influence results. The authors also proposed a discrete particle swarm optimization algorithm [11], which made use of two-hop influence spread approximation and a smart initialization, degree discount, was used in this case. DPSO outperformed CELF++ [13] algorithm in running time while the influence results were similar.

Smart initialization techniques were also studied in [18], where each node has a probability to be inserted into the initial population proportional to its out degree. [7] tried to insert "smart" individual into the initial population while choosing randomly the rest of the population. In both cases GA with smart initialization obtained better results w.r.t. standard GA.

The impact of the GA parameters on the results were studied in [29], the authors suggest recommended parameters values according to their findings.

[22] makes use of an informed mutation operator, a neural network is used to predict which nodes to change in a solution, using centrality metrics of nodes as inputs.

In [31] a premature convergence problem is handled, the diversity is guaranteed by using GA with multi-population competition with specific crossover and mutation operations among populations, their solution offers a variety of results.

Paper	**Introduced concepts**	**Social model**	**Graphs**	**Experiments**	**Results**
[28]	GA with parallelization in GPU of node representations and for obtaining and evaluating seed sets	LT	UC Irvine messages DiggFacebook wall posts Only a limited sample of the networks was used due to the GPU memory limitation: up to 1400 nodes	Comparison with the greedy algorithm: in terms of quality by fixing the same time for both and in terms of execution time required to reach similar results	The proposed solution has up to 16% quality improvement and is up to 34 times faster
[31]	GA with mutli-population competition, can guarantee diversity when the threshold in LT is fixed, crossover and mutation operations are defined between populations	LT	NETSCIENCE 1.589 nodes and 2.742 edges; EMAIL 167 nodes and 5.784 edges; INFECTIOUS 410 nodes and 5.530 edges; UCSOCIAL 1.899 nodes and 20.296 edges	Comparison with greedy, CELF, SA(simulated annealing)	Influence spread is comparative to KGA algo and superior w.r.t. the others in terms of efficiency GA is slower then CELF and SA but faster then KGA;GA can obtain a variety of results

Table 3.2 continued from previous page

Paper	Introduced concepts	Social model	Graphs	Experiments	Results
[15]	Multi objective differential evolutionary algorithm, where the objectives are the influence spread and the information delivery cost Improved selection operation: if there is a pareto dominance relationship between two solution vectors, a better solution is chosen, otherwise both solutions are chosen to be part of the next generation;when the population grows too much, a selection operation with non-dominated sorting is called to reduce the population size to the original size	IC	ArnetMiner, 2.162 nodes and 19.875 edges	Comparison with NSGA-II	Better results of the paretofront
[7]	GA for Linear Threshold Model	LT	CA-CrQc, 5K nodes and 14K edges; CA-HepPh, 12K nodes and 118K edges; Epinions 75K nodes and 508K edges	Comparison with best solutions according to centrality measure (50 best ranking nodes in each centrality measure), such as Degree, PageRank, Closeness, Betweenness comparison with GA with smart initialization: insertion of the solution with 50 most central nodes. Seed set size was fixed to 50.	GA-pure solutions are worse then the centrality based ones, but there is an important improvement with smart initialization, in this case the most successful centrality metrics are degree and betweenness
[11]	Discrete particle swarm optimization algorithm, smart initialization using degree discount heuristic and approximation of two-hop influence spread	IC, WC	NetInfective 410 nodes and 17K edges; NetScience 1.5K nodes and 3K edges; NetGRQC 5K nodes and 15K edges; NetHEPT 15K nodes and 59K edges	Comparison with other heuristics such as Degree, PageRank, SAEDV, CELF++	Outperforms CELF++ in running time, while maintaining comparable performance

Table 3.2 continued from previous page

Paper	Introduced concepts	Social model	Graphs	Experiments	Results
[25]	Community-based approach, propagates information in two phases: in the first phase the seed nodes are expanded among different communities at the beginning of diffusion, in second phase the influence propagates within the communities, the community influence function depends on the first and second order seeds; the greedy algorithm is applied to find the solution.	WC	NetHEPT 15K nodes and 31K edges; NetPHY 37K nodes and 174K edges; Epinions 76K nodes and 406K edges; Amazon 335K nodes and 926K edges; DBLP 317K nodes and 1M edges; Patent 3.8M nodes and 16.5M edges; Pokec 1.6M nodes and 22.3M edges; LiveJournal 4M nodes and 35M edges; Orkut 3.1M nodes and 117M edges	Comparison with baseline algorithms such as IPA, TIM+, IMM, Single Discount, Degree	CoFIM provides the best influence spread, comparablewith TIM+ and IMM algorithms. In terms of runtime & efficiency CoFIM is significantly faster then TIM+ and IMM on most datasets and it requires less memory
[29]	Study of the impact of GA algorithm parameters on the result	LT	Enron 87K nodes and 1M edges; Digg 30K nodes and 88K edges; Facebook wall posts 47K nodes and 877K edges; Barabási-Albert (BA) model Watts-Strogatz (WS) modelErdős-Rényi (ER) model	All possible combinations of parameters picked from prefixed initial ranges were tried of different networks. The correlation coefficient of the best parameter settings on different datasets were computed in order to confirm the good performance of these parameters all over the datasets	Individuals: a bigger number of individuals led to better results but slower execution times. The authors recommend values of 0.7 and 0.9 for crossover and mutation ratio respectively; the best results had low mutation potency parameter.
[24]	Application of GA for SIR model in small world networks	SIR	Watts-Strogatz (WS) model of different number of nodes: 500, 1K and 2K	Study of the algorithm convergence and the correlation between PageRank and selection frequency of the nodes	The authors archived good results after the first 100 generations. Highly frequent individuals don't always have high PageRank

Table 3.2 continued from previous page

Paper	Introduced concepts	Social model	Graphs	Experiments	Results
[30]	Comparison of GA initialization strategies	SPA - Spreading activation	ca 5K nodes and 29K edges; hep 15K nodes and 59K edges; phy 37K nodes and 232K edges; email 37K nodes and 368K edges	Global initialization strategies:- PageRank (all individuals chosen from top20%)- RandomLocal initialization strategies:- Community 100% (all individuals are chosen from the top 40 communities with higher modularity values and from each community the nodes with top 20% degree values are chosen)- Community 60%(the remaining 40% are chosen randomly)	Pagerank led to the worst results while other initialization strategies had similar performances.
[10]	Influence spread approximation with two-hop influence spread, smart initialization and similarity-based local search	IC	Dolphin 62 nodes and 159 edges; NetGRQC 5K nodes and 14K edges; NetHEPT 15K nodes and 59K edges	Comparison with Random, PageRank, Degree-Centrality, CELF and CMA-IM (proposed algorithm) variations	The proposed algorithm has comparative with CELF influence spread results, while it is up to 10times faster. The algorithm is slower then the other heuristics while it's solution quality is much higher
[18]	Guided genetic algorithm: the probability of a node to be inserted into an individual of the initial population is proportional to it's out degree.	IC	Random graphs with 50, 100, 150 and 200nodes	Comparison with the original GA algorithm	In the majority of cases the proposed algorithm outperforms GA, but in some cases it has a problem of premature convergence

Table 3.2 continued from previous page

Paper	Introduced concepts	Social model	Graphs	Experiments	Results
[5]	The first application of GA for IM problem	IC, WC	Wiki 7115 nodes and 103689 edges; Amazon 262K nodes and 1M edges	The runtime and influence results were compared to the greedy algorithm and heuristics such as HighDegree and Single Discount	On Amazon dataset the computation time of GA is reduced by a half w.r.t.the greedy algorithm while the influence results are comparable. In case of the Wiki dataset the computational times and results were both comparable.
[6]	Multi-objective GA where the objectives are the Influence spread and the "budget" k - size of seed set; MicroGP algorithm was used.	IC, WC	Gong2016Influence	MOEA pareto front results are compared to the heuristics solutions of each seed set size: High Degree and Single Discount	MOEA outperforms the heuristics for the intermediate values of k, while having alternating success for high and low values

Table 3.2: Metaheuristics state-of-the-art solutions

4 GA design

In this chapter we explicate all the steps we have done on the way to our basic GA improvement. We tried to collect all the best ideas we have seen in the literature and put them into practice by adapting them to our GA. Some of them were more successful while some others hardly improved the GA with the basic setup; nevertheless we report all of our results in case it would be helpful for someone who is working on the problem and tries to understand which are the most promising directions and which are not.

The objectives of our work were the following:

1. reduce runtimes of the fitness function, which is the main bottleneck of the algorithm
2. design genetic operators which would produce correct individuals and would not duplicate already present in the population individuals, in order to increment the diversity of the population
3. find a good smart initialization technique, which would boost the algorithm convergence
4. introduce graph-domain knowledge to the mutation operators
5. reduce the GA search space
6. try to reduce the runtimes by introducing early stop and dynamic population size

In the following sections we will discuss all these steps in detail.

4.1 Fitness function approximations

The influence spread computation under the IC and WC models was proven to be #P-complete in [13]. It is one of the main bottlenecks in the IM algorithms, which in the literature is usually calculated by using Monte Carlo sampling with a big number of influence propagation simulations (10.000 is a frequently used number). One approach used to tackle this problem was to avoid the expensive Monte Carlo spread calculation and to use a different way of candidates evaluation, as for example, the usage of Reverse Reachable sets [4].

An alternative solution is to make use of spread function approximations which are faster then the Monte Carlo standard approach. This is the idea we have decided to adopt in this work.

We made use of two different approximations: a 2-hop approximation taken from [19], which was used as well in [10] because of its efficiency and a Monte Carlo simplified variation, where we truncate the information propagation up to a certain degree, or hop. We compared both these methods with the standard Monte Carlo sampling, in order to decide which of them is more convenient then the other in terms of their accuracy vs velocity trade-off.

Then main question we asked ourselves was how the dimension and the connectivity of the graph impacted the results given by each of the approximation functions and in which circumstances one of them can be more appropriate then the others.

Before more formal definitions of the spread functions let's recap the spread function task:

Definition 4.1. (Influence spread) Given a graph G with the influence propagation probabilities $p(u, v)$ associated to each edge (u, v) in the graph, and a given seed set S, calculate the number of active nodes at the end of influence spread propagation.

Monte Carlo

In this procedure the influence spread is calculated by simulating the information propagation in the graph according to a specified model a given number of times $no\_simulations$. The influence spread is then approximated by the simulations sample mean.

The Monte Carlo method is better explained by the pseudo code 1:

Algorithm 1 Monte Carlo simulation

Input seed set *S*, seed set size *k*, graph *G*, social model *model*, number of simulations *no_simulations*
Output spread mean value *avg*, spread standard deviation *std*

procedure MonteCarloSimulation
    $sample \leftarrow$ *initialize_array(k)*
    $i \leftarrow 1$
    for $i \leq$ *no_simulations* **do**
        $A \leftarrow$ *set(S)*
        $B \leftarrow$ *set(S)*
        $converged \leftarrow$ *False*
        while not $converged$ **do**
            $nextB \leftarrow$ *set()*
            for n in B **do**
                $neighbors \leftarrow$ *get_neighbors(n, G)*
                $not\_activated\_neighbors \leftarrow$ *set_difference(set(neighbors), A)*
                for m in $not\_activated\_neighbors$ **do**
                $propagation\_prop \leftarrow$ *get_influence_spread_prop(n, m, model)*
                $p \leftarrow$ *random_real(0, 1)*
                If *p > propagation_prop* **then**: *add_element_to_set(m, nextB)*
            $B \leftarrow nextB$
            If *is_empty(nextB)* **then**: $converged \leftarrow True$
            $A \leftarrow$ *union(A, B)*
        $sample[i] \leftarrow$ *length(A)*
        $i \leftarrow$ *i + 1*
    $avg \leftarrow mean(sample)$
    $std \leftarrow$ *standard_deviation(sample)*

Two-hop spread

This approximation function is taken from [19]. Influence spread is approximated by the following formula:
$\hat{\sigma}_S = \sum_{s \in S} \hat{\sigma}_{\{s\}} - \left(\sum_{s \in S} \sum_{c \in C_s \cap S} p(s,c) \left(\sigma_c^1 - p(c,s)\right)\right) - \chi$,
where
$\chi = \sum_{s \in S} \sum_{c \in C_s \setminus S} \sum_{d \in C_c \cap S \setminus \{s\}} p(s,c)p(c,d)$,
σ_u^1 is the one-hop influence spread of node u, defined as $\sigma_u^1 = 1 + \sum_{c \in C_u} p(u,c)$,
and C_u denotes the set of neighbour nodes of node u.

Monte Carlo max hop

Monte Carlo variation where influence is propagated up to a maximum number of hops max_hops. If, for example, $max\_hops = 2$, the influence is propagated up to neighbours of neighbours of the initial seed set S.

The algorithm becomes as in 2.

Algorithm 2 Monte Carlo max hop simulation

Input seed set *S*, seed set size *k*, graph *G*, social model *model*, number of simulations *no_simulations*, maximum number of hops *max_hops*

Output spread mean value *avg*, spread standard deviation *std*

```
1:  procedure MonteCarloMaxHopSimulation
2:      sample ← initialize_array(k)
3:      i ←1
4:      for i ≤ no_simulations do
5:          A ← set(S)
6:          B ← set(S)
7:          converged ← False
8:          j ← max_hop
9:          while not converged and j ≤ max_hop do
10:             nextB ← set()
11:             for n in B do
12:                 neighbors ← get_neighbors(n, G)
13:                 not_activated_neighbors ← set_difference(set(neighbors), A)
14:                 for m in not_activated_neighbors do
15:                     propagation_prop ← get_influence_spread_prop(n, m, model)
16:                     p ← random_real(0, 1)
17:                     If p > propagation_prop then: add_element_to_set(m, nextB)
18:             B ← nextB
19:             If is_empty(nextB) then: converged ← True
20:             A ← union(A, B)
21:             j ← j + 1
22:         sample[i] ← length(A)
23:         i ← i + 1
24:     avg ← mean(sample)
25:     std ← standard_deviation(sample)
```

4.2 Evolutionary algorithm improvements

In this section we will better frame the core of the algorithm: the EA graph-aware operations. We will start with the description of the basic setup where the overall procedure with common operations is described, followed by each class of the operators, in the end the combination of all the best-working pieces put on trial will be discussed.

4.2.1 Basic GA

All of the following experiments have in common a basic GA setup where one modified component was inserted at a time.

The design of a GA starts from individuals encoding. In our case each individual genotype was composed of a vector of node IDs of the network graph. The fitness function adopted for our further experiments was the Monte Carlo max hop procedure.

The basic initial individuals generator creates an individual by randomly selecting a set of nodes from the graph. During each generation parents are selected by performing tournament selection, which then reproduce by means of crossover operator, followed by mutation.

We performed a modified one point crossover in our EA. The crossover operator adopted in this work has a constraint to produce individuals without nodes repetitions, and, for the diversity sake, we also set a constraint of producing nodes different from those given in input. To satisfy the former condition we swapped only nodes which were not in common between the two parents, while maintaining the common nodes original positions. The figure 4.1 clears up how the crossover is done. Step 1 could be implemented also by simply selecting common nodes and by moving them at the beginning of the individual or by attaching them at the end. But in this case it would be more probable that these nodes would stay attached to each other in the subsequent crossover operations, which can cause diversity loss.

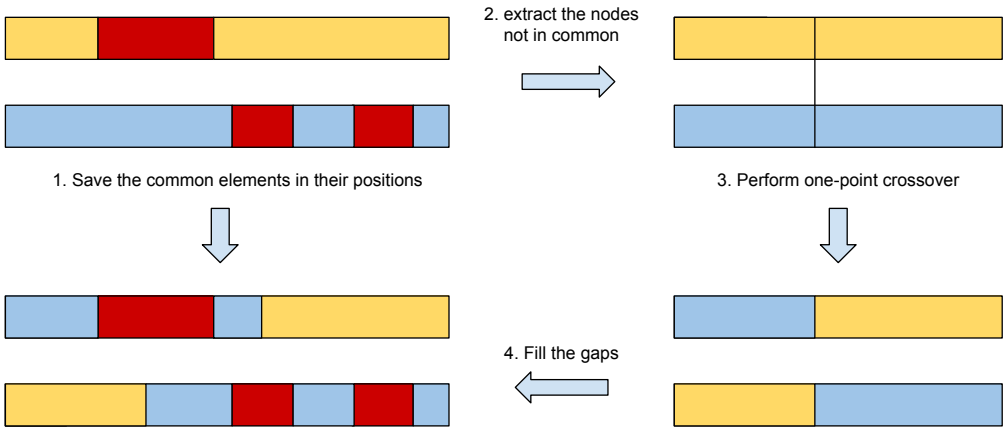

Figure 4.1: Crossover operator

Regarding to the latter property, there may be cases in which we may produce no new individuals. By performing the crossover as described above we still could end up with two children which are symmetrical to their parents. This can happen when the parents have all nodes in common except one. In order to produce two distinct nodes in such situation we forced a mutation of one random gene of one of the two children. This operation is also helpful to increase population nodes diversity when it has particularly low values (we will define this metric later).

The crossover operation is followed by the mutation applied with a certain probability. The basic mutation operator mutates one random gene with one random node of the graph, excluding the already present in the individual nodes, in order to avoid repetitions.

Finally for the new population selection generational replacement with a low number of elites was used.

In our experiments we also made use of the early stop condition to terminate the evolution when there was not any improvement during the last α generations, which was set to 10% of the total generational budget.

The complete list of the default setup can be found in the appendix.

4.2.2 Smart Initialization

Multiple publications report a positive effect on the EA when using smart initialization technique for the IM problem. Here we tried to compare different smart initialization techniques and to see whether some of them are more advantageous then the others.

Insertion of a single fit individual

In this experiment a single "smart" individual was inserted into the initial population. It was the fittest individual w.r.t. different centrality metrics: the individual was composed of the first k nodes which obtained the highest centrality scores. We compared the following centrality metrics:

- betweenness: captures how often the node is on the shortest path among any two nodes of the graph
- closeness: corresponds to the average length of the shortest path between the node and all the other nodes of the graph
- degree: the number of out-going links of the node
- eigenvector: gives high score to nodes neighboring with a few highly connected nodes
- katz: a variant of the eigenvector centrality, the distance between two nodes is measured by considering the number of possible walks among them, and not only the shortest path

We wanted to investigate whether the GA could improve these initial fit seed sets on our datasets. The idea was taken from [7], where the GA obtained a significant improvement of one of the individuals inserted in such way.

Multiple smart individuals

Here a percentage of the initial population is initialized according to one of different initialization strategies described below. Each strategy is defined by the way a single individual is created, which is not conditioned by the other individuals in the population.

- degree random: the probability of a node to be inserted into an individual is proportional to its degree
- degree random ranked: all the nodes were ranked here according to their degree, and the probability of a node of being selected for an individual corresponds to its ranking position
- community degree: the general idea here is that the dataset might have community structure and a good solution would contain the fittest nodes from different communities. First we select for each node in the individual the community we want to pick it from, in our case the probability of a community to be selected was proportional to its size. The next step is the selection of the community "fittest" node: here we selected according to the degree probability.

  Two community detection algorithms were used for comparison: the Louvain algorithm [3] and spectral clustering, our goal was not the ranking of all possible community detection algorithms but rather a simple attempt to see if this kind of smart initialization could be a promising direction for the IM problem.

The rest of the population is initialized randomly.

4.2.3 Graph-aware mutations

We attempted to accelerate the search process of fit individuals by introducing some knowledge about the graph nodes properties and graph structure into the mutation operator.

Global mutation methods

By global mutations we mean mutations in which we mutate the selected gene with any randomly chosen node from the graph, while the gene selection is performed by using some criteria.

Table 4.1 recapitulates the global mutation techniques we tested in our experiments. We investigated whether the application of an additional criterion for the mutation gene selection can lead to a faster convergence. The modified global mutation operators give higher mutation probability to genes which are considered less fit w.r.t. some measure.

Mutation	Description
Global random	The gene to be mutated is selected randomly
Global low degree	The mutation probability of the node corresponds to its out-degree: nodes with lower degree have more probability of being mutated
Global low spread	Here nodes with low influence spread values have higher chance of being mutated
Global low additional spread	To evaluate the mutation probabilities in this case we calculated how each node increase the spread value of the seed set without that node, the low increase corresponds to higher probability to be mutated

Table 4.1: Global mutation functions

Local mutation methods

Here the idea is to perform a local search on the graph: to use different *walks* around the graph while mutating genes instead of jumping to random places and changing drastically the spatial location of the mutated nodes.

This time the gene to be mutated is selected randomly, while the new node is chosen among the closest to the selected nodes ones and according to some selection criteria. The list of all the local mutation variations is reported in table 4.2.

Mutation	Description
Local neighbors random	The new node is randomly selected among the node's neighbors
Local neighbors second degree	The neighbor is chosen according to it's degree probability
Local neighbors approximated spread	A variation of the previous mutation, but this time the spread is approximated
Local neighbors additional spread	To select the neighbor we calculate how it would increase the seed set spread when inserted at the mutated node position
Local embeddings random	The new node is randomly selected among the closest nodes according to its corresponding node2vec [14] embedding

Table 4.2: Local mutation functions

Activation mutations

Activation mutation functions were inspired by the Reverse Reachable sets [4] idea of fit nodes detection. While performing Monte Carlo simulations we track, up to a certain degree, which nodes activated which other nodes and we use this information to reach the "sink" nodes when performing mutation.

Each time the Monte Carlo spread simulation is performed, for each newly activated node we track the flow of influence spread in order to detect which nodes are the most frequent "activators" of other nodes. We save for each node (as its attribute in the graph) which nodes caused theirs activation, limited up to two-hop distance, to avoid large memory usage. Each activator has its own counter, so for each node we can say which activator was the most influent for that node.

So if **a** activated **b**, and **b** activated **c**, the counters are incremented in the following way, illustrated better in the figure 4.2:

$node$ **b** : $[b]\_activated\_by[\mathbf{a}]$+= 1

$node$ **c** : $[c]\_activated\_by[\mathbf{b}]$ += 1, $[c]\_activated\_by[\mathbf{a}]$ += 1

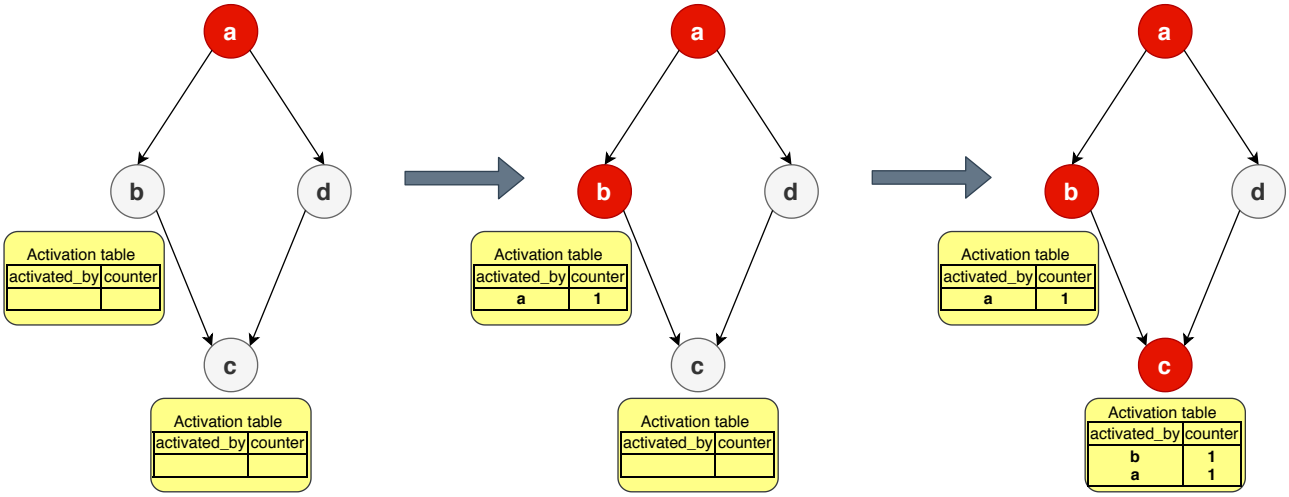

Figure 4.2: Nodes activation tracking

Subsequently we used this information in two different ways:

1. local activation mutation: mutate the selected gene with the node by which it was activated the biggest number of times (with the highest activation counter)

2. global activation mutation: mutate the node a with one node random node b, satisfying the following constraints: 1) a has never activated b and vice versa, all the nodes in the candidate individual have never activated b and vice versa

4.2.4 Mutations combination

We have observed that some mutations are more effective then others when used in different evolution phases and when applied to diverse datasets (with different out-degree distributions). Therefore we decided to use them all: instead of deciding the best mutation to use singularly, use them all with dynamic selection mechanism.

Given n mutation methods to choose from at each mutation call, we don't know the fitness improvement distribution of none of them and the number of trials could be potentially infinite (without deciding a priori the number of generations). This scenario seems to fit perfectly the *multi-armed bandit problem*, where we have n actions to choose from, for each of them the expected reward is unknown thus estimated from the past experience, and at each step we have to choose between exploration and exploitation, between actions we have more accurate estimate about and the less explored methods with lower expected reward.

During different evolution stages the reward model may change, for example when we are stuck in a local minima, the exploration mutation strategies may be more appropriate w.r.t the exploitation methods, thus we have a *non-stationary* MAB problem.

To solve MAB we decided to adopt a widely used algorithm, which promotes exploration of mutations with bigger reward uncertainty, the Upper Confidence Bound (UCB) algorithm. More precisely, since the problem is non-stationary, a sliding-window variation was used.

UCB takes into account the linear combination of the expected reward and the uncertainty of reward estimation of a particular action. This mechanism is a fair trade-off between exploration and exploitation we need in our algorithm.

We adopted the UCB1 algorithm, according to which in order to select the next action the algorithm maximizes the following expression:

$$Q(a) + exploration\_weight \cdot \sqrt{\frac{2 \log t}{N_t(a)}} \tag{4.1}$$

where $Q(a)$ is the cumulative reward of the last n times the action was selected (n is the sliding window size), $N_t(a)$ corresponds to the number of times the action was chosen and t is the counter of the total selections.

To shrink the "learning phase" and to explore less throughout the generations we adopted exponentially decaying exploration weight:

$exploration\_weight(g) = (\frac{1}{g})^3$, where g is the generations counter.

4.2.5 Nodes selection strategies

The search space of the IM problem is combinatorial, therefore the algorithms scale poorly when the seed set size increases. Our attempt in the following experiments was to reduce the search space and/or to keep it constant for multiple graph sizes.

The selected nodes were used by the EA for all its operations and the discarded nodes were completely ignored.

Min degree nodes

The nodes with low out degree have a null or a very little probability of influencing the others, thus it is very unlikely that they would be a part of an optimal solution. In this experiment we tried to consider only nodes having out-degree bigger then a specified threshold to see whether it can speed up the algorithm's convergence.

Best spread nodes

When considering big scale networks, it is very likely that the optimal seed set would be made of nodes placed far enough from each other to not activate the same nodes. According to this assumption it is highly probable that the nodes in the solution would be "isolated" among them, in the sense that their activations would not intersect. So the resulting spread would have value very close to the sum of the spreads of each of the nodes calculated separately; therefore we know that the final solution nodes should have high spread values.

We tested the truthfulness of this assumption on our datasets by applying nodes filtering: we computed the spread of each node and selected the best n nodes for the EA nodes pool, while the other nodes were not considered by our algorithm.

The proposed selection technique requires the spread measures of two nodes to be comparable with each other in order to determine which one is bigger. Also we cannot use a big number of Monte Carlo simulations for this step since we have to evaluate all the nodes of the dataset, which may require a big amount of time. So we have the following problem: how many Monte Carlo simulations do we need in order to have a statistically significant sample and to minimize sample size?

By statistical significance of two sample comparison we mean the comparison of the confidence intervals of the means of two samples without intersections.

We need to compare a spread's sample mean of one node with another node and we want to know with a certain probability whether one is bigger then the other. Let us suppose the spread variable has a normal distribution.

For nodes comparison we used their spread confidence intervals to decide whether one is bigger then another or to conclude whether they are comparable or not. In order to do that we adopted student's T distribution for confidence interval estimate:

$$\bar{X} \pm t\frac{s}{\sqrt{n}} \tag{4.2}$$

where $\bar{X}$ is the sample mean, n the sample size and s the sample's standard deviation and t is the t-value calculated given our sample size and the desired confidence level.

Assuming $\bar{X_1} < \bar{X_2}$, we defined two nodes not comparable if their confidence intervals overlapped: if $\bar{X_1} + t_1\frac{s_1}{\sqrt{n_1}} \geq \bar{X_1} - t_2\frac{s_2}{\sqrt{n_2}}$.

The required number of Monte Carlo simulations was a variable we could not decide a priori for all nodes, so we decided to proceed gradually, by incrementally increasing our spread samples up to a needed precision.

To sum up, to select the best n nodes of the graph we followed the next steps:

1. perform the Monte Carlo sampling of the spread of each individual with a number of simulations needed to ensure a maximum error rate max_erorr_rate on the mean that we are trying to estimate, the error of the mean is estimated by calculating the confidence interval of the student's t distribution.

2. the best n nodes are selected by looking only at their mean values. Then the node with the lowest spread (among the best n nodes) is considered and all the nodes of the graph which are not comparable with this node are selected as well.

3. select the best nodes and the nodes which are not comparable with the lowest best node and go to step 1 after having set the max_error_rate to a lower value.

We repeated this loop until all our best nodes were comparable with the others.

The sample size required to compare some pair of nodes resulted too costly, thus we were content with a number of best nodes contained in a given in input range: between n and m, lower and upper bounds. As soon as the algorithm finds a number of the highest spread nodes n, and the number of the incomparable with those nodes is lower then $m - n$, the algorithm terminates and returns n best nodes and all the nodes which cannot be compared with them.

5 Experiments

5.1 Datasets

Different datasets used in this work will be described in this section, also a quick overview of their connectivity and dimensions is provided.

5.1.1 Real-world graphs

Three real-world graphs with different dimensions and connectivity were used: Epinions, Wiki-Vote, Amazon and CA-GrQc. Epinions is the who-trusts-whom online social network. In Wiki-Vote the edges represent who-voted-whom information in elections for promoting adminship. The Amazon dataset contains co-purchased products relations. The smallest dataset, CA-GrQc, is a network of collaboration in General Relativity and Quantum Cosmology fields, thus it contains co-authorship information between authors. All the graphs can be downloaded from SNAP [21].

Table 5.1 contains detailed description of each graph representing the aforementioned datasets.

Dataset	**Graph size**	**Node out-degree**				
		Avg	**Stddev**	**Min**	**Max**	**Median**
Epinions	75.879 nodes, 508.837 edges	13.41	52.67	1	3079	2
Wiki-vote	7.115 nodes, 103.689 edges	14.57	42.28	0	893	2
Amazon	262.111 nodes, 1.234.877 edges	4.71	0.95	0	5	5
CA-GrQc	5.242 nodes, 14.496 edges	5.52	7.92	1	81	3

Table 5.1: Real-world datasets specifications

5.1.2 Synthetic graphs

We also make use of artificial datasets obtained by generating new graphs by making use of the Barabasi-Albert model [2].

These datasets were created of diverse size and connectivity, a more detailed description of each of them is provided in table 5.2. All the graphs were created using the networkx library [16] with a $random\_seed$ parameter set to 0.

5.2 Experimental setup

All the experiments were implemented using python language and inspyred library [9], the source code can be foun on github [1].

Each experiment used single process and single thread due to hardware limitations.

5.3 Fitness functions comparisons

In order to decide which of the approximation functions to adopt for our EA we compared the runtimes and the quality of results of each approximated spread function.

We computed spread values with the approximation methods and the traditional Monte Carlo method on different datasets under the IC and the WC diffusion models, then their runtimes and correlations were analyzed.

For each dataset the standard Monte Carlo, the Monte Carlo max hop and the two-hop influence spread approximations were calculated on 100 randomly generated seed sets, same for all the procedures. The Pearson

Dataset	Graph type and size	Node out-degree				
		Avg	Stddev	Min	Max	Median
Barabasi_albert_1000_1	1.000 nodes, new_edges=1	1.99	3.51	1	75	1
Barabasi_albert_1000_3	1.000 nodes, new_edges=3	5.98	7.37	3	99	4
Barabasi_albert_1000_5	1.000 nodes, new_edges=5	9.95	10.57	2	114	7
Barabasi_albert_1000_7	1.000 nodes, new_edges=7	13.90	12.80	7	137	10
Barabasi_albert_1000_9	1.000 nodes, new_edges=9	17.84	15.65	9	148	12
Barabasi_albert_1000_11	1.000 nodes, new_edges=11	21.76	19.25	11	211	15
Barabasi_albert_10000_1	10.000 nodes, new_edges=1	1.99	4.13	1	243	1
Barabasi_albert_10000_3	10.000 nodes, new_edges=3	5.99	8.96	3	293	4
Barabasi_albert_10000_5	10.000 nodes, new_edges=5	9.99	13.45	2	365	7
Barabasi_albert_10000_7	10.000 nodes, new_edges=7	13.99	17.00	7	446	10
Barabasi_albert_10000_9	10.000 nodes, new_edges=9	17.98	21.25	9	470	12
Barabasi_albert_10000_11	10.000 nodes, new_edges=11	21.98	25.97	11	695	15

Table 5.2: Synthetic Barabasi-Albert datasets configurations, new_edges parameter indicates the number of edges created from a new node to the existing nodes.

correlations of spread values calculated by the proposed approximations and the standard Monte Carlo technique were then calculated.

We were also interested to see the relation of the runtimes and spread correlations with some variables, such as: the seed set size k, the number of Monte Carlo simulations $no\_simulations$ and the probability p under the IC model. The $no\_simulations$ parameter was applied to both the Monte Carlo standard and its max hop version; the max_hop parameter of the Monte Carlo max hop was set to 2 in all the experiments.

In some circumstances Monte Carlo spread values are high and limited by the graph dimension while the other approximations' results may still have high diversity values because they do not propagate to the entire graph. In those cases the correlation with the standard Monte Carlo could not be high since it's values are almost constant. So in order to understand when the correlation values are more relevant the standard deviation of the standard Monte Carlo influence spread was tracked. Finally the average runtime of each approximation was reported.

We examined the correlations and the runtimes by varying one variable of interest at a time: seed set size k, number of simulations and probability p under the IC model.

5.3.1 Results

Here we will briefly discuss the main trends we observed from all our experiments. We will report only some selected results to be short, the complete list of results of all the experiments and datasets can be found in the appendix.

Generally, we observed that both approximated algorithms under some circumstances had an important gain in runtime and high spread correlations with our original Monte Carlo method.

The correlation of the spread values in this case was high for both functions (>0.9) for the WC model and under the low values of the IC model (p=[0.01, 0.1]). Under the IC model the correlation was not that high, but

it is not relevant for our case, since the IC model is mostly used in literature with low p values, which we are going to adopt for our next experiments as well.

Speaking about runtime, we observed that the results depend on the dimension and the connectivity of the graph we are using, the dimension of the seed set k, and, of course, on the number of simulations we are going to adopt for the Monte Carlo methods.

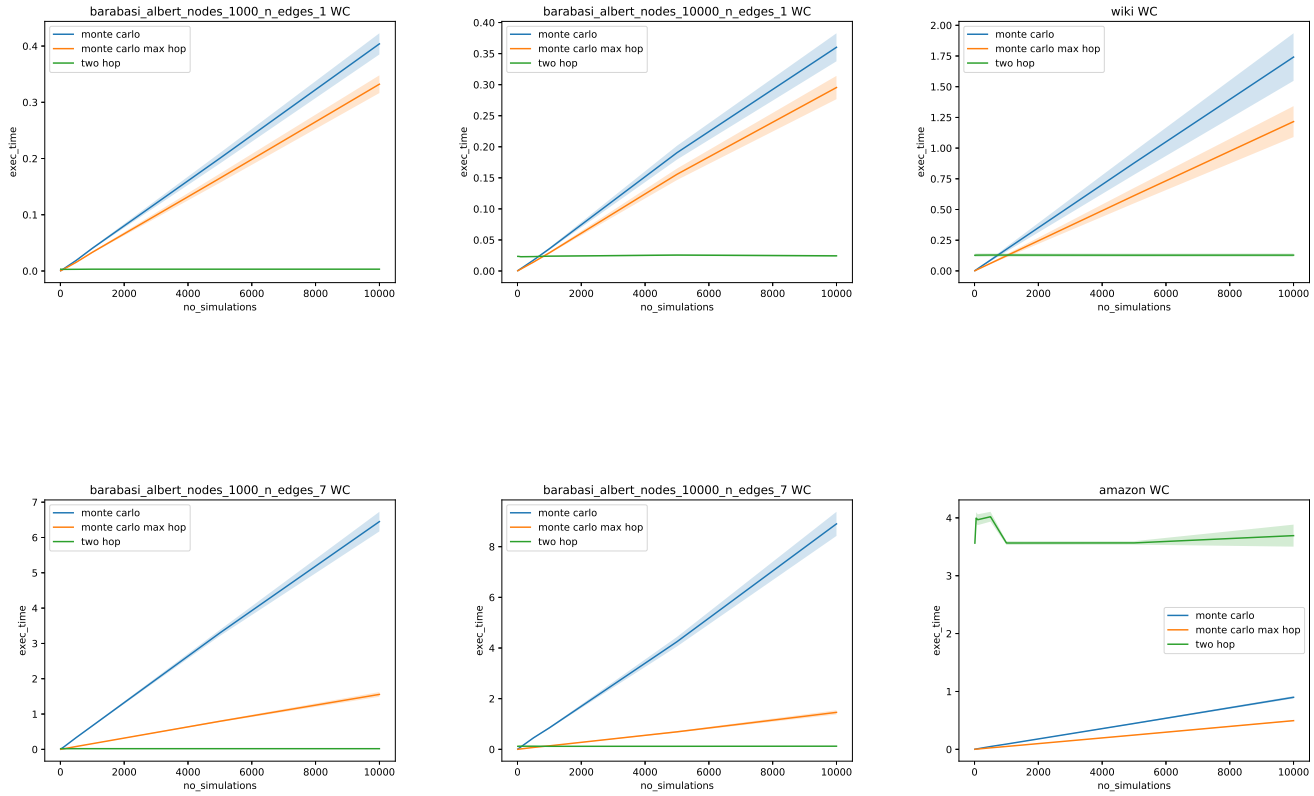

Figure 5.1: Spread functions execution times as a function of number of Monte Carlo simulations

In the figure 5.1 we report the running times of all the examined procedures as the number of Monte Carlo evaluations varies. In this experiment we set the seed set size to 5, and the model to weighted cascade.

Here we can observe the linear growth of the runtime of Monte Carlo methods w.r.t the number of simulations. We can still notice that in some cases the two hop spread is slower. In particular the Monte Carlo method is faster when the number of simulations is low, which is an obvious result, but also when the datasets size grows: as the dataset becomes bigger, the intersection of the Monte Carlo results line and the two hop spread line is moving to the higher number of evaluations and it is no more visible for the amazon dataset, which is the biggest one we used fot this experiment.

Another thing which can be noticed from these plots is that the Monte Carlo max hop approximation is particularly helpful as the datasets connectivity increases: at the top row we have Barabasi-Albert graphs of different dimensions and the same connectivity, while on the second row we can find same more connected datasets; here we can clearly see the difference of performance gain of the Monte Carlo max hop method. This trend was also observed in all the other conducted experiments: more connected graphs can have a longer influence diffusion process since usually more nodes get influenced, so a truncation of this process up to some level (max hop) can lead to bigger runtime gains.

The next observation might be again kind of obvious: the runtime has a linear relationship with the seed set

size, figure 5.2 reports the runtimes as a function of seed set size k, the number of the Monte Carlo evaluations is set to 100, the model used is the weighted cascade. Here we can observe that the two hop spread may be convenient only if the graph size is kept small, while for the real world cases it might be the slowest spread evaluation method.

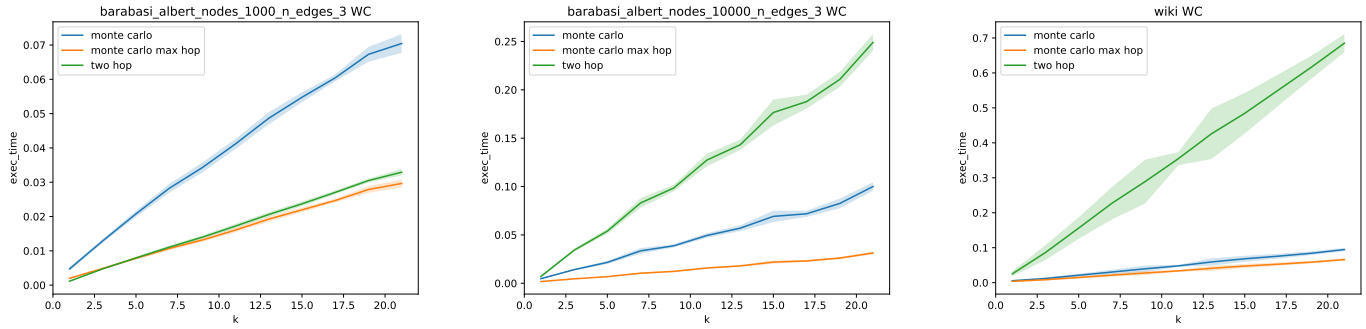

Figure 5.2: Spread functions execution times as a function of seed set size k

These experiments indicate that both methods have high correlations with the Monte Carlo spread, while the Monte Carlo max hop solution is more efficient w.r.t the two-hop spread when the size of the dataset increases and the number of simulations is not very high. When using Monte Carlo max hop approximation we can observe a significant runtime boost in the more connected graphs.

5.4 EA improvements

5.4.1 Smart initialization experiments

Smart initialization experiments were performed only on the real-world datasets: Wiki-vote, Amazon and Epinions. In order to decide which smart initialization technique suits best our needs we run the EA algorithm with common basic parameter settings and different population initializations. The complete list of default parameters can be found in the appendix.

From our experiments we observed that the results of smart initialization strongly depend on the connectivity of the dataset, the more the dataset presents single highly connected nodes, the more the degree-like smart initialization tend to obtain a better influence score.

Insertion of a single fit individual

The insertion of the individuals containing high centrality nodes generally brought a good starting point the EA for the Wiki-Vote and CA-GrQc graphs. We are not able to provide comparisons for the Amazon dataset given their cost in memory and/or runtime (we did not consider centrality metrics with computation time > 12 hours).

The degree centrality is a clear winner for the Wiki-Vote dataset (see fig. 5.3), which is a reasonable result considering its degree distribution. CA-GrQc experiments do not present an apparent difference between various centrality metrics under the IC model, while when considering the WC model, the betweennes centrality obtained a slightly better results with respect to the other methods. The only centrality metric we could compute for the Amazon dataset was the degree centrality; it does not seem to bring any improvement to the results on this dataset.

Multiple smart individuals

In the following section we report the comparison of smart initializations done by inserting diverse "smart" individuals into the population.

Due to the resources constraints we were not be able to test these comparative experiments on a different number of seed set sizes. For the experiment we tested the algorithm with a fixed seed set size $k = 10$, the percentage of "smart" individuals was set to 0.5. The table 5.3 reports the obtained results. The community detection algorithm used was the Louvain algorithm.

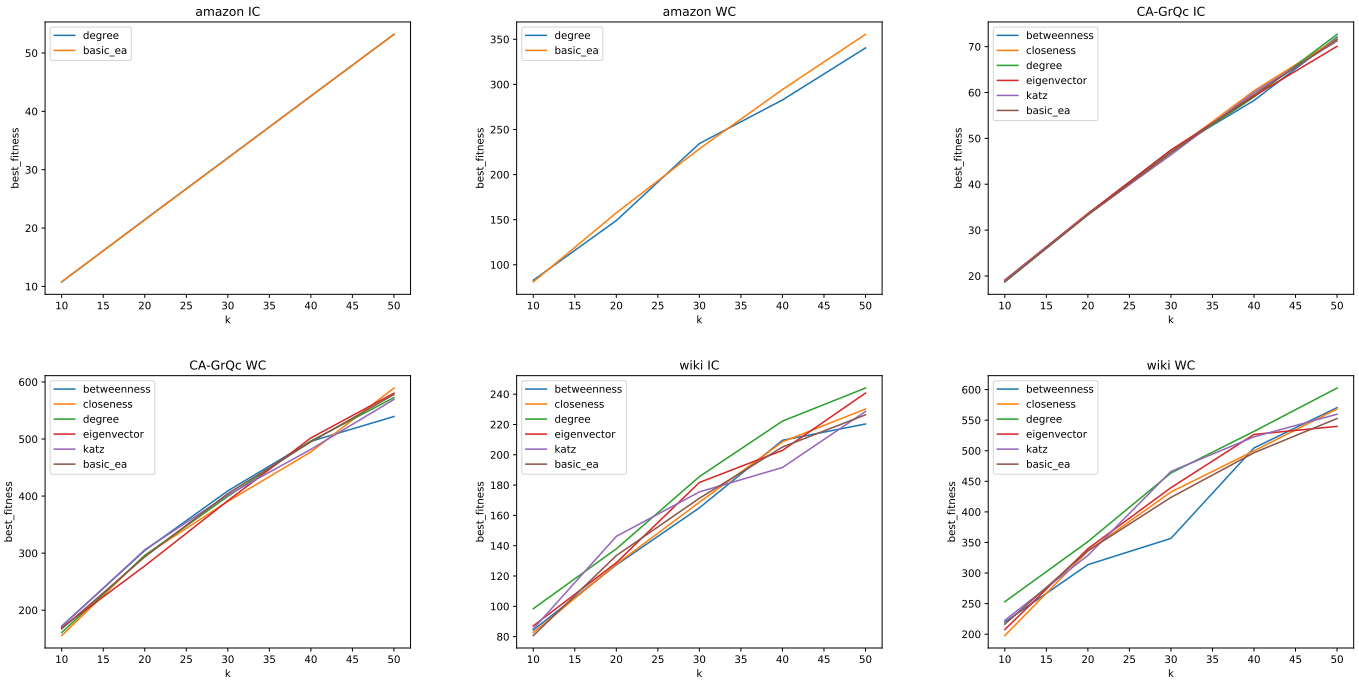

Figure 5.3: Single smart individual insertion comparison

From these results we were not able to detect the best smart initialization method involving multiple individuals introduction. Still since the results of the *degree_random* and *community_degree* strategies are comparable, the *degree_random* technique is a better choice because of much lower computation time requirement and no need to choose and fine tune community detection algorithms.

Dataset	Model	Best fitness			
		Degree_random	Community_degree	Degree	No smart. init.
Wiki-Vote	WC	226.24	223.35	**230.65**	220.75
Wiki-Vote	IC	93.7	93.21	**94.04**	87.92
Ca-GrQc	WC	**147.38**	147.36	146.91	146.13
Ca-GrQc	IC	18.64	18.59	18.59	**18.67**
Amazon	WC	66.55	65.99	**67.97**	67.87
Amazon	IC	10.79	**10.83**	10.80	10.82

Table 5.3: Smart initialization multiple smart individuals comparison

5.4.2 Graph-aware mutations

Global mutations

The results of our experiments are shown in the figure 5.4.

Figure 5.4: Global mutations compared

As we can observe from these results, the degree based global mutation obtained a slightly better results on the graphs presenting a low number of large degree nodes, such as Wiki-Vote and CA-GrQc, while for the Amazon dataset degree-based global search led to worse results. The execution times of these methods were in general comparable, except the $ea\_global\_low\_additional\_spread$ method which execution time grows linearly with the seed set size.

For the Wiki-Vote dataset the execution of methods involving spread calculation were not completed because of the high computation time requirement.

Local mutations

Due to computation resources limitations we could not compute one of the local mutations, the additional spread local mutation, which, like its global variation is particularly slow on the graphs with high degree nodes.

In the figure 5.5 we reported also the global random mutation, which is our reference basic default mutation method.

Figure 5.5: Local mutations compared

Before reviewing the results we should add an important detail regarding the local embeddings mutation: we performed some hyperparameters search for the Amazon node2vec embeddings training, where we tried a range of different parameters for node2vec training and selected the parameters which produced embeddings which better captured the shortest path length between two nodes and nodes' degrees, while for the other datasets we just picked random parameters without any hyperparameter search.

Said that, it is not a surprise that the local embeddings mutation operator performed better on the Amazon dataset, while its results for CA-GrQc and Wiki-Vote were way worse then the others. We can for now conclude that embeddings with some hyperparameter search may lead to promising results as we can see from the Amazon WC plot, but it came at a cost of training numerous embeddings with different hyperparameters settings, which is quite a long process considering real-world graphs dimensions.

The neighbours degree mutation is at least as good as global random mutation for all the datasets, while the approximate spread and neighbours random approach are still valid choices for CA-GrQc and Wiki-Vote, but not in case of the Amazon dataset.

Combination of multiple mutations

We framed the problem of selecting the best mutation from a possible set as a Multi Armed Bandit problem, given that the most useful mutation might be dependant on the graph at hand.

The algorithm considered a sliding window of length 100, so only the last 100 improvement results were taken into consideration for the expected reward evaluation.

Overall, as we can notice from figure 5.6, the results were slightly better then the basic EA algorithm. While the improvement is not very significative, it is, however, positive for all graphs, which confirms that our UCB algorithm adapts to all the datasets.

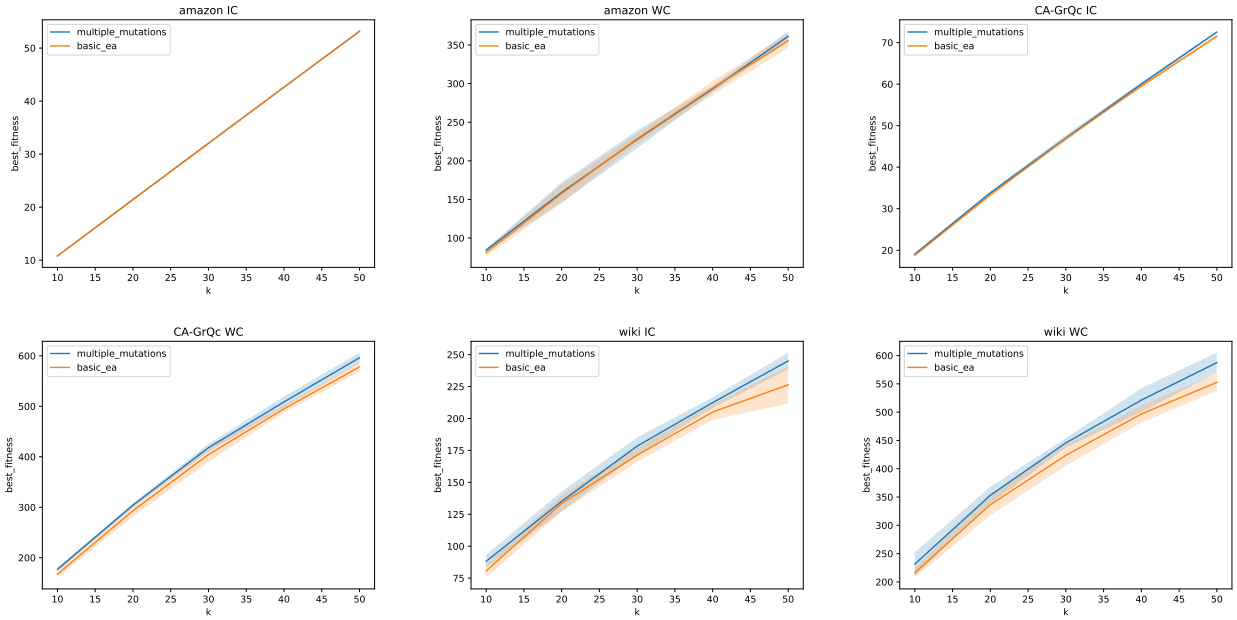

Figure 5.6: Multiple mutations experiments results

5.4.3 Nodes filtering

For these algorithms we kept each other GA setting as simple as possible without any of the above improvements, in order to be able to compare any of these techniques separately with the previous ones.

Min degree nodes

In the following experiment we filtered for the EA nodes which had a given minimum out-degree. Specifically min-degrees values of 0, 1, 2, 3 and 4 were tested.

From figure 5.7 we can see that nodes selection technique led to a visible improvement only in case of the Wiki-Vote dataset. The high standard deviation does not permit to define a clear winner of one min-degree value upon the others.

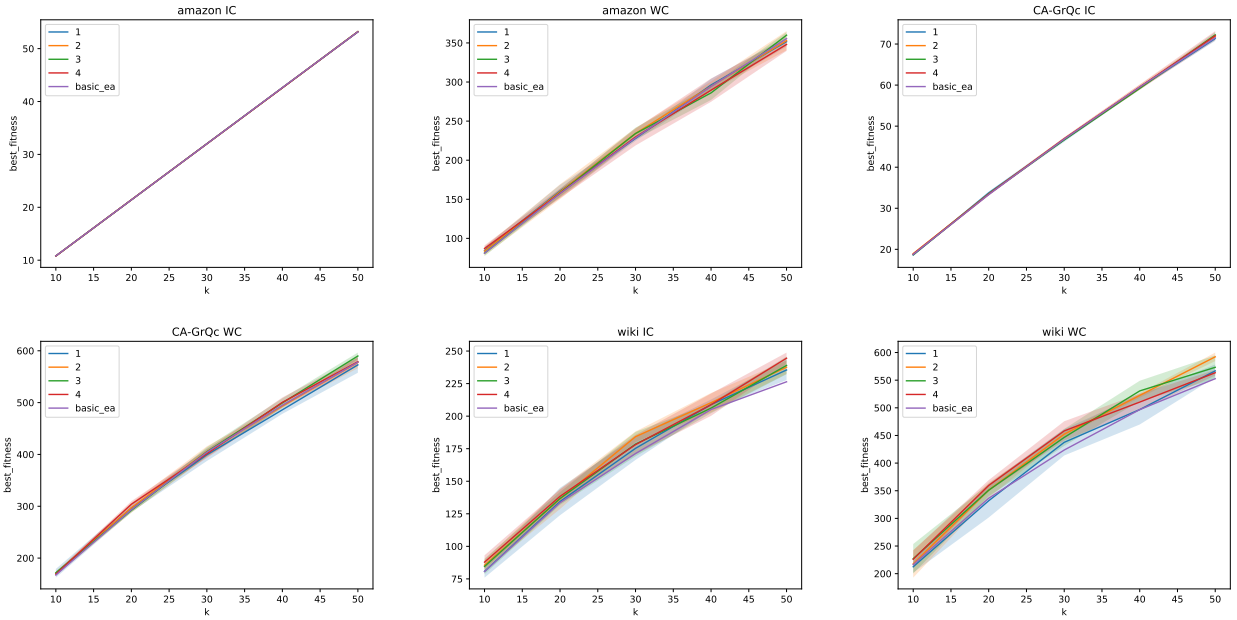

Figure 5.7: Minimum degrees compared

Best spread nodes

In this experiment the nodes were filtered by using a more sophisticated criteria: their spread approximations. For this purpose we decided to use a minimalistic Monte Carlo max hop setup: with $max\_hop = 2$.

In the step 1 of our procedure 4.2.5 the initial minimum precision sample was done with the max_error_rate set to 0.8 and was subsequently decreased of 0.1 at each iteration.

The parameters n and m, which correspond to the lower bound and the upper bounds of the number of the best spread nodes, were chosen in a way to guarantee the same search space size regardless of the seed set length. The search space size is the number of total possible combinations we can do with k nodes given the number of nodes to choose from N. The upper and lower bound on the number of total possible combinations was set to 10^9 and 10^{11}.

From the plots 5.8 we can see that this method significantly improves the results up to 40% w.r.t. the basic genetic algorithm.

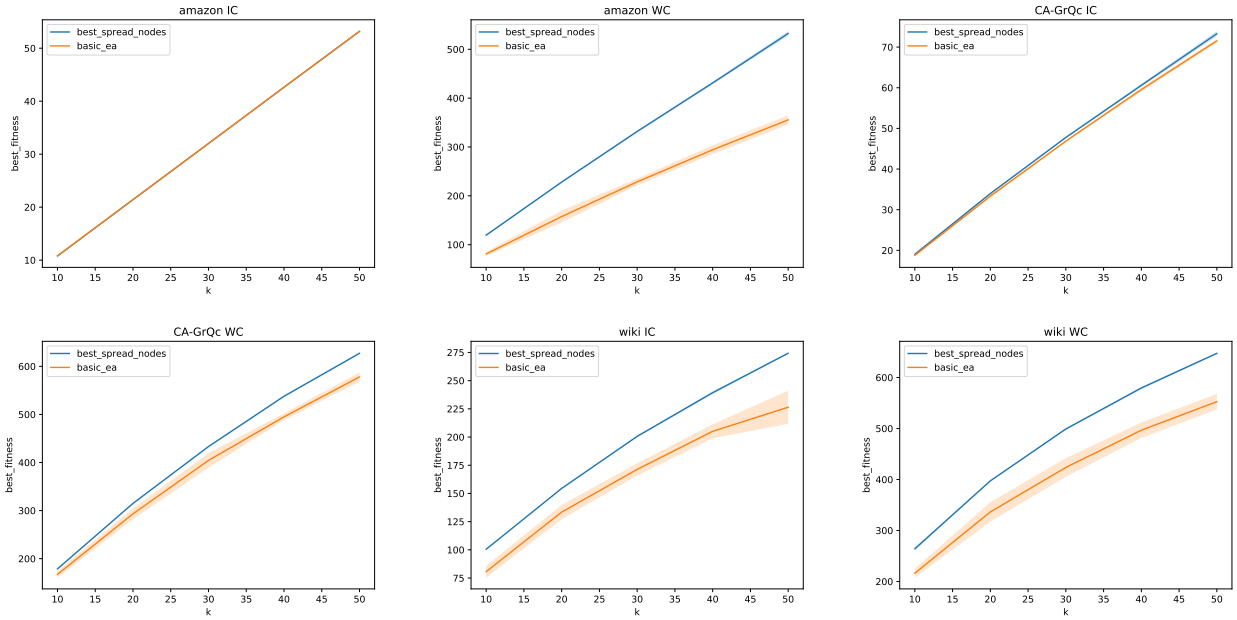

Figure 5.8: Best spread nodes filtering results

Best results combination

In our final comparison we tested the combination of all the improvements with the basic genetic algorithm, and a combination of the most computationally efficient improvements, in order to see whether we could reduce the running times by archiving the same results.

The final combination included best spread nodes selection, random degree smart initialization and multiple mutations combination. In the "selected" setup we eliminated mutations having the highest computational cost.

In both the improved GA setups in order to tackle high fitness evaluation time we made use of *dynamic population size*: the population size increase when no improvement is present for the last q generations, and in case of improvement, the population size decreases. However the number of individuals in the population is bounded with low_bound and $upper\_bound$.

Additionally we compare our results with a simple baseline and the CELF [20] algorithm to see how the final algorithm performs in terms of spread and the execution time. The high degree heuristic was chosen as a baseline, according to which the solution includes nodes with the highest degree in the graph (or out-degree when dealing with directed graphs).

Since heuristics and genetic algorithms performed optimization for different objective functions (Monte Carlo simulations in the case of heuristic and Monte Carlo max hop for the GA), in order to have a common comparison measure we evaluated each final seed set found by the heuristics and the GAs by using Monte Carlo simulation with $no\_simulations = 100$.

Despite the additional techniques tackled to reduce runtimes, the execution of the final combination of all the best-working improvements on the Wiki-Vote dataset was never terminated, it exceeded 100 hours of the execution time limit.

As we can observe from the figure 5.9, the results present an important spread increase of the improved versions of genetic algorithm when compared to it's basic version.

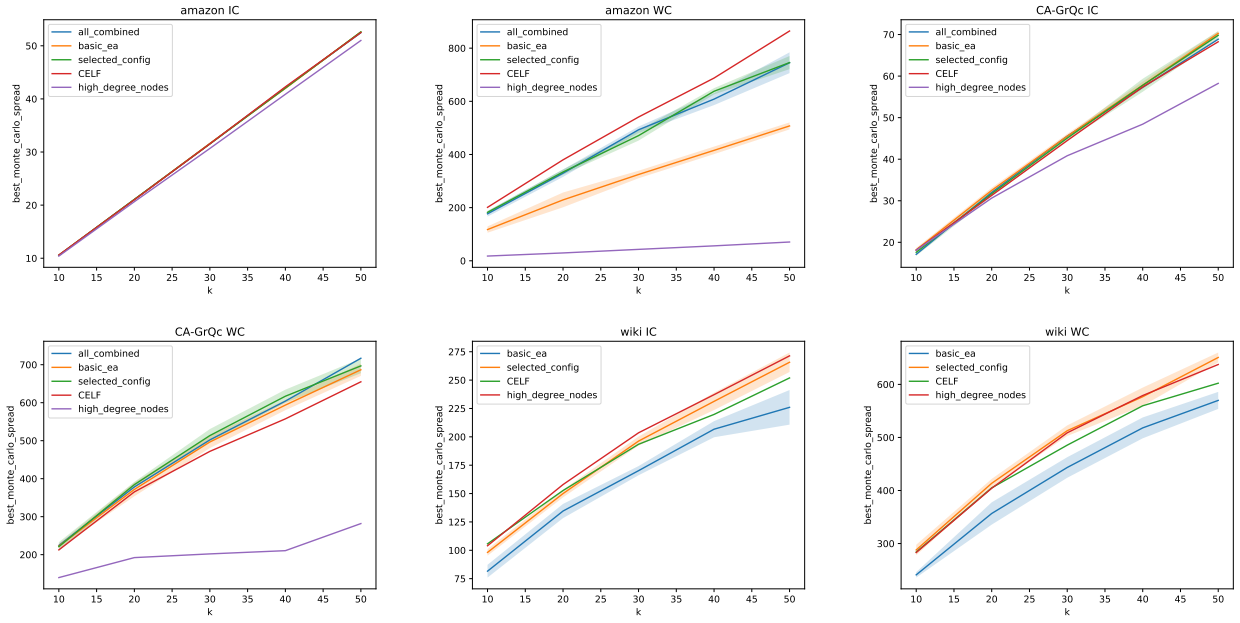

Figure 5.9: Comparison of the final algorithm with baseline heuristics

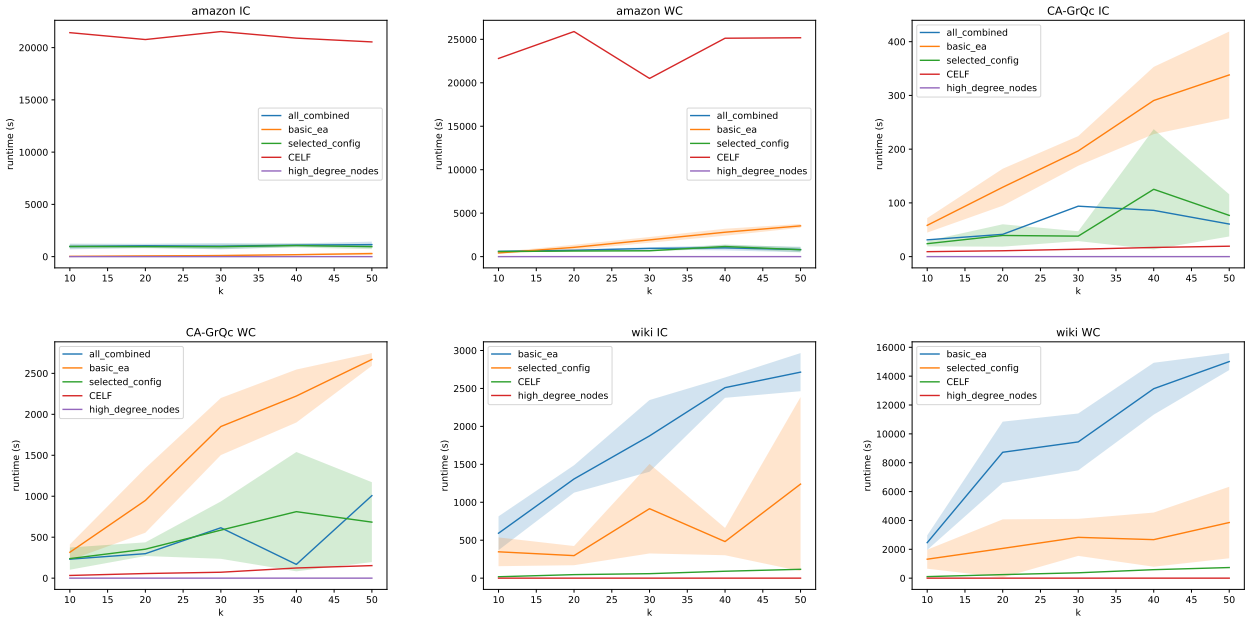

Figure 5.10: Runtimes comparison of the final algorithms

For all the datasets we have influence spread improvements up to 40% with respect to the basic GA. The runtimes 5.10 of the improved version are also significantly reduced, especially with the increase of the seed set size. When comparing with high degree nodes heuristic, only in one dataset, Wiki-Vote, this heuristic is the best solution, while in the other datasets it obtained the worst results. This confirms the fact that due to the various degree distribution of the datasets the same simple heuristic cannot be applied to all of them, while the improved GA obtained good results in all the datasets, including the Wiki-Vote, where its result is comparable to the best-working heuristic.

Finally we compare the obtained results with CELF, an improved version of the general greedy algorithm [17]. For the datasets of modest dimensions, Wiki-Vote and CA-GrQc, the GA has higher spread values, while being more expensive in terms of execution time. When talking about Amazon dataset, which is bigger more

then 30 times then the other two graphs, the GA significantly outperforms CELF in terms of runtime: it is up to 10 times faster then CELF. The spread results are slightly worse but still comparable with CELF. This may be due to the use of truncated Monte Carlo fitness function which may not approximate well the spread in the Amazon dataset, where the connectivity is very low and highly homogeneous.

6 Conclusions

We improved a basic genetic algorithm, GA, applied to the IM problem with graph-aware operators and enhancements aimed to reduce runtimes.

Monte Carlo max hop simulations were used to approximate the influence spread and permitted a significant computational time saving and a possibility to conduct a variety of experiments.

The most important progress was achieved by limiting the search space of the GA by the means of nodes filtering, a method which selected the most promising individuals in terms of information spread in an efficient way . The limited number of combinations permitted the algorithm to better scale with the increasing graph size.

The combination of graph-aware mutations permitted the GA to adapt to graphs with different structure or connectivity. Dynamic population size was useful to reduce fitness evaluation costs and smart mutations computations.

We compared our results with the CELF algorithm and obtained a significant runtime gain with comparable influence spread results for the Amazon dataset, and improved spread results with longer runtimes for the other datasets. The important advantage of the genetic algorithm is the diversity of proposed solutions it presents with comparable fitnesses.

6.1 Limits

The usage of the Monte Carlo max hop procedure as fitness function may be not a good choice in case of networks with scarce connectivity or under the IC model with high probability values. The initial nodes filtering method is valid only under the assumption that the most influential nodes are distant, in the sense that they do not have nodes in common. So this method may perform worse on small scale networks where the optimal seed set may include nodes with common spread influence. The execution time of influence spread or execution of mutation metrics which involve computation of influence spread or degree-like methods is proportional to the fitness of the individual, so the runtime of the algorithm is proportional to the graph connectivity and a big number of fit candidates in the population may imply high computation costs.

6.2 Future work

There is still further research to do on the graph-aware mutations, which would be better adapted to work with a small amount of pre-filtered nodes. The success of the node2vec embeddings mutation with some hyperparameter search suggests that the usage of embeddings may accelerate the search process. The usage of embeddings reduces the genetic algorithm runtime since it does not involve degree-like mutations which make use of expensive genes evaluations.

Bibliography

[1] Influence maximization. https://github.com/katerynak/Influence-Maximization, 2020.

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. URL: https://science.sciencemag.org/content/286/5439/509, arXiv:https://science.sciencemag.org/content/286/5439/509.full.pdf, doi:10.1126/science.286.5439.509.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008. URL: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008, doi:10.1088/1742-5468/2008/10/p10008.

[4] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time, 2012. arXiv:1212.0884.

[5] Doina Bucur and Giovanni Iacca. Influence maximization in social networks with genetic algorithms. pages 379–392, 03 2016. doi:10.1007/978-3-319-31204-0_25.

[6] Doina Bucur, Giovanni Iacca, Andrea Marcelli, Giovanni Squillero, and Alberto Tonda. Multi-objective evolutionary algorithms for influence maximization in social networks. pages 221–233, 01 2017. doi:10.1007/978-3-319-55849-3_15.

[7] Arthur Rodrigues da Silva, Rodrigo Ferreira Rodrigues, Vinícius da Fonseca Vieira, and Carolina Ribeiro Xavier. Influence maximization in network by genetic algorithm on linear threshold model. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Elena Stankova, Carmelo M. Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, Eufemia Tarantino, and Yeonseung Ryu, editors, *Computational Science and Its Applications – ICCSA 2018*, pages 96–109, Cham, 2018. Springer International Publishing.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Co., USA, 1979.

[9] A. Garrett. inspyred: Bio-inspired algorithms in python, June 2017. URL: http://aarongarrett.github.io/inspyred/.

[10] Maoguo Gong, Chao Song, Chao Duan, Lijia Ma, and Bo Shen. An efficient memetic algorithm for influence maximization in social networks. *Comp. Intell. Mag.*, 11(3):22–33, August 2016. URL: https://doi.org/10.1109/MCI.2016.2572538, doi:10.1109/MCI.2016.2572538.

[11] Maoguo Gong, Jianan Yan, Bo Shen, Ma Lijia, and Qing Cai. Influence maximization in social networks based on discrete particle swarm optimization. *Information Sciences*, 367, 07 2016. doi:10.1016/j.ins.2016.07.012.

[12] A. Goyal, W. Lu, and L. V. S. Lakshmanan. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *2011 IEEE 11th International Conference on Data Mining*, pages 211–220, Dec 2011. doi:10.1109/ICDM.2011.132.

[13] Amit Goyal, Wei Lu, and Laks V.S. Lakshmanan. Celf++: Optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th International Conference Companion on*

World Wide Web, WWW '11, pages 47–48, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1963192.1963217, doi:10.1145/1963192.1963217.

[14] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2939672.2939754, doi:10.1145/2939672.2939754.

[15] Jian-bin Guo, Fu-zan Chen, and Min-qiang Li. *A Multi-objective Optimization Approach for Influence Maximization in Social Networks*, pages 706–715. 01 2019. doi:10.1007/978-981-13-3402-3_74.

[16] Aric Hagberg, Pieter Swart, and Daniel Chult. Exploring network structure, dynamics, and function using networkx. 01 2008.

[17] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/956750.956769, doi:10.1145/956750.956769.

[18] Pavel Krömer and Jana Nowaková. Guided genetic algorithm for the influence maximization problem. In *COCOON*, 2017.

[19] Jong-Ryul Lee and Chin-Wan Chung. A fast approximation for influence maximization in large social networks. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 1157–1162, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2567948.2580063, doi:10.1145/2567948.2580063.

[20] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 420–429, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1281192.1281239, doi:10.1145/1281192.1281239.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] Krzysztof Michalak. Informed mutation operator using machine learning for optimization in epidemics prevention. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 1294–1301, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3205455.3205647, doi:10.1145/3205455.3205647.

[23] H. T. Nguyen, M. T. Thai, and T. N. Dinh. A billion-scale approximation algorithm for maximizing benefit in viral marketing. *IEEE/ACM Transactions on Networking*, 25(4):2419–2429, Aug 2017. doi:10.1109/TNET.2017.2691544.

[24] Rodrigo Rodrigues, Arthur Silva, Vinícius Vieira, and Carolina Xavier. *Optimization of the Choice of Individuals to Be Immunized Through the Genetic Algorithm in the SIR Model*, pages 62–75. 07 2018. doi:10.1007/978-3-319-95165-2_5.

[25] Jiaxing Shang, Shangbo Zhou, Xin Li, Lianchen Liu, and Hongchun Wu. Cofim: A community-based framework for influence maximization on large-scale networks. *Knowl.-Based Syst.*, 117:88–100, 2017.

[26] Youze Tang, Yanchen Shi, and Xiaokui Xiao. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1539–1554, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2723372.2723734, doi:10.1145/2723372.2723734.

[27] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. *CoRR*, abs/1404.0900, 2014. URL: http://arxiv.org/abs/1404.0900, arXiv:1404.0900.

[28] MichałWeskida and RadosMichalski. Evolutionary algorithm for seed selection in social influence process. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '16, pages 1189–1196, Piscataway, NJ, USA, 2016. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=3192424.3192645.

[29] Michal Weskida and Radosław Michalski. Finding influentials in social networks using evolutionary algorithm. *Journal of Computational Science*, 31, 12 2018. doi:10.1016/j.jocs.2018.12.010.

[30] Carolina Xavier, Vinícius Vieira, and Alexandre Evsukoff. Populational algorithm for influence maximization. volume 9789, pages 346–357, 07 2016. doi:10.1007/978-3-319-42089-9_25.

[31] Kaiqi Zhang, Haifeng Du, and Marcus Feldman. Maximizing influence in a social network: Improved results using a genetic algorithm. *Physica A: Statistical Mechanics and its Applications*, 478, 02 2017. doi:10.1016/j.physa.2017.02.067.

Appendix A

Here the experimental setup of approximations experiments can be found.

Parameter name	Parameter value	Parameter description
k	5	seed set size
no_simulations	100	number of Monte Carlo simulations
max_hop	2	maximum hop parameter used for the Monte Carlo max hop simulations
p	0.1	probability of information spread diffusion under the IC model
g_seed	0	random seed used to generate the Barabasi-Albert graphs

Table A.1: Fitness functions correlation experimental setup

Appendix B

Here we report the basic hyperparameters setup used used for all the subsequent experiments where one variable at a time was modified.

Parameter name	Parameter value	Parameter description
no_simulations	100	number of Monte Carlo simulations
max_hop	3	maximum hop parameter used for the Monte Carlo max hop simulations
p	0.01	probability of information spread diffusion under the IC model
population_size	100	number of individuals in population
offslpring_size	100	dimention of offspring produced
dynamic_population_size	False	option of the dynamic increase and decrease of the individuals
generational_budget	100	the upper bound on the number of generations
smart_initialization	none	smart initialization technique used
crossover_rate	1.0	crossover probability
mutation_rate	0.1	mutation probability
tournament_size	5	tournament size for tournament parents selection
num_elites	2	number of best individuals to save from last generation
min_degree	0	the minimum degree used to initially filter nodes
nodes_filtering	False	option of initial best spread nodes filtering

Table B.1: Basic GA experimental setup

In the following tables we report the experimental setup for the final comparison, in particular for selected_config and all_combined experimental setups.

Parameter name	Parameter value	Parameter description
no_simulations	100	number of Monte Carlo simulations
max_hop	3	maximum hop parameter used for the Monte Carlo max hop simulations
p	0.01	probability of information spread diffusion under the IC model
dynamic_population_size	True	option of the dynamic increase and decrease of the individuals
dynamic_population_lower_bound	2	the minimum number of inidividuals in the population
dynamic_population_upper_bound	100	the maximum number of individuals in the population
generational_budget	100	the upper bound on the number of generations
smart_initialization	degree_random	smart initialization technique used
crossover_rate	1.0	crossover probability
mutation_rate	0.1	mutation probability
tournament_size	5	tournament size for tournament parents selection
num_elites	1	number of best individuals to save from last generation
min_degree	1	the minimum degree used to initially filter nodes
nodes_filtering	True	option of initial best spread nodes filtering
adaptive_mutations	True	option of adaptively select mutations to apply
mutations_to_alterate	[”ea_local_activation_mutation”, ”ea_global_low_deg_mutation”, ”ea_global_random_mutation”,]	list of mutation operators to choose from

Table B.2: Selected config experimental setup

Parameter name	Parameter value	Parameter description
no_simulations	100	number of Monte Carlo simulations
max_hop	3	maximum hop parameter used for the Monte Carlo max hop simulations
p	0.01	probability of information spread diffusion under the IC model
dynamic_population_size	True	option of the dynamic increase and decrease of the individuals
dynamic_population_lower_bound	2	the minimum number of inidividuals in the population
dynamic_population_upper_bound	100	the maximum number of individuals in the population
generational_budget	100	the upper bound on the number of generations
smart_initialization	degree_random	smart initialization technique used
crossover_rate	1.0	crossover probability
mutation_rate	0.1	mutation probability
tournament_size	5	tournament size for tournament parents selection
num_elites	1	number of best individuals to save from last generation
min_degree	1	the minimum degree used to initially filter nodes
nodes_filtering	True	option of initial best spread nodes filtering
adaptive_mutations	True	option of adaptively select mutations to apply
mutations_to_alterate	[”ea_local_activation_mutation”, ”ea_local_neighbors_second_degree_mutation”, ”ea_local_embeddings_mutation”, ”ea_local_neighbors_random_mutation”, ”ea_local_approx_spread_mutation”, ”ea_global_activation_mutation”, ”ea_global_low_deg_mutation”, ”ea_global_random_mutation”]	list of mutation operators to choose from

Table B.3: All combined experimental setup