Université Catholique de Louvain
Faculté des sciences appliquées
Département d'ingénierie informatique

Viral marketing and Community detection algorithms

Promoteur: Pr. Pierre Dupont

Mémoire présenté en vue
de l'obtention du grade de
ingénieur civil en informatique

par Sébastien Combéfis
et Jean Miller

Louvain-la-Neuve
Année académique 2006-2007

Contents

Introduction

Viral marketing is a field which has much to do with the study of complex networks. It is based on the idea that the propagation of information through a social network looks like a viral propagation. Viral marketing thus develops different propagation models in order to simulate propagation and eventually understand it better. Some of those models can be found in [KKvT03] and [KS06].

One of the main open problems in the viral marketing field is to find a k-set with maximal influence. This problem consists in predicting which group of k persons would offer the best propagation in the network. Being an advertiser with a limited budget allowing you to advertise only k persons, who should be advertised in order to maximize the effect on the social network ? Should it be popular people ? People with few but very reliable relations ? Which criteria could be used in order to find those people ? This problem is called *the influence maximization problem* and is studied in [RD02] and [KKvT03].

Community detection is a different field whose goal is to detect communities within networks. It tries to answer some other questions such as what is a community ? When should people be considered close enough to be in the same community ? What does being close mean ? Some of the most famous concepts used in community detection such as *random walks* [LP05] or *edge betweenness* [GN02, MPM06] try to answer these questions.

Both viral marketing and community detection are based on the observation of graphs in order to retrieve some underlying information about them. Both communities and influence are based on the relational structure people have developed with their neighbours, friends or any other acquaintances. Both communities and influence are real life concepts and have a very concrete meaning in social networks. Though community detection and viral marketing share so many aspects, algorithms developed within those two fields do not have anything in common.

This contrast is the starting point of this work. Once concepts of both viral marketing and community detection are introduced along with their existing models and algorithms, this work looks for ways for both fields to help each other by using techniques from one field into the other. All solutions, existing ones as well as new suggestions, are then thoroughly tested and compared. Conclusions are eventually drawn considering the results of combining both fields in order to improve their existing solutions.

1

The influence maximization problem

In order to find the most influent entities in a social network, the network must be represented in a convenient way. These networks are represented as graphs. The first section recalls some definitions and notations about graphs. An algorithm to find the most influent entities is then defined in the second section while the third section presents several models of diffusion. A model of diffusion is a hypothesis set defining a step-by-step dynamic of propagation of the information through the network. The fourth and last section summarize all that was introduced in a more convenient way. The goal of this chapter is to draw the frame of the influence maximization problem.

1.1 Definition

1.1.1 Graph theory

A *social network* is represented as a directed graph $G = (V, E)$ with V the set of vertices representing the individuals of the social network and $E \subseteq \{(v, w) \mid v, w \in V\}$ the set of edges, each edge representing a relation between two individuals. The number of vertices of the graph is denoted $n = |V|$ and the number of edges is denoted $m = |E|$. Last but not least, only simple directed graphs are considered, meaning that multiple edges and loops should not appear.

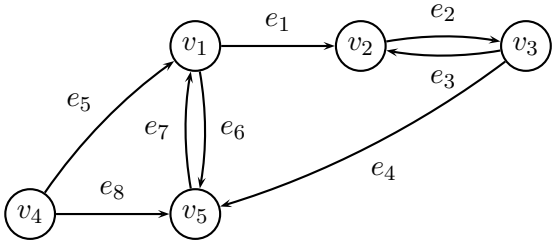


Figure 1.1: A example of a directed graph

Figure 1.1 shows a small directed graph made of 5 vertices and 8 edges. The graph is therefore defined by the sets $V = \{v_1, \cdots, v_5\}$ and $E = \{e_1, \cdots, e8\}$. An edge is a link between a pair of vertices (v, w) with $v, w \in V$, going from v to w. Vertex w is called *the head* and v *the tail* of the edge. Edges (v, w) and (w, v) are considered as different entities as graphs are directed. If there exists an edge between two vertices u and v, these vertices are said to be *adjacent*.

The number of edges leaving a given vertex $v \in V$ is called *the out-degree* of the vertex and is denoted $d^+(v) = |\{(v, w) \in E \mid w \in V\}|$. Similarly, the *in-degree* of a vertex v is the number of edges arriving at the vertex and is denoted $d^-(v) = |\{(w, v) \in E \mid w \in V\}|$. For example, $d^+(v_5) = 1$ and $d^-(v_5) = 3$ in figure 1.1.

Two notions remain to be defined. The *successors*, or children, of a given vertex $v \in V$ are the vertices toward which an edge coming from v exists. The set of *successors* is denoted $N^+(v) = \{w \in V \mid (v, w) \in E\}$. The *predecessors*, or parents, of a given vertex v form the set $N^-(v) = \{w \in V \mid (w, v) \in E\}$. For example, $N^+(v_5) = \{v_1\}$ and $N^-(v_5) = \{v_1, v_3, v_4\}$ in figure 1.1.

1.1.2 Information diffusion

Some vocabulary specific to the context of information diffusion in a network also exists. A vertex $v \in V$ is said to be *active* if the information has reached the vertex and was accepted by it. A synonym of active is *contaminated*, especially if the context involves some viruses instead of information propagation. On the other side, a vertex which information has not reached or convinced so far is called *inactive* or *non-contaminated*. During the process of diffusion of the information, a vertex can switch from inactive to active but not vice-versa. This one-way transition is a restriction imposed in order to simplify diffusion models. In order to diffuse information, there must be some initial source. Some vertices must be initially activated. The set $A_0 \subseteq V$ denotes initially activated vertices. Vertices from this set are said to be *targeted for initial activation.*

A *diffusion model* is the whole definition of the propagation process. It determines how propagation takes place : when should a contamination attempt be made, how the success of such an attempt is determined, and so on ... Given a network $G = (V, E)$ and a diffusion model M, the *influence* of a set of vertices $A \subseteq V$, denoted $\sigma_M(A)$, is the expected number of active vertices once the diffusion process is over. Since information diffusion is a random process, the influence is no more than an expectation of the amount of active vertices once it is over. Denoting $\varphi_M(A)$, with model M and initial set A, the set of active vertices once diffusion is over, influence is expressed as

$$\sigma_M(A) = E[\mid \varphi_M(A) \mid]$$

Given a social network $G = (V, E)$ and a diffusion model M, the *influence maximization problem* consists in finding a k-set A of maximum influence. A k-set is a set of size k, and k is a parameter of the influence maximization problem. The goal of the influence maximization problem is thus to find out which initial set A of given size k would give rise to the best propagation through the network. The problem is simply formulated in equation 1.1.

$$\underset{\substack{A \subseteq V \\ |A| = k}}{\operatorname{argmax}} \sigma_M(A) \tag{1.1}$$

The problem of finding the best set is known though to be $\mathcal{NP}$-hard for all diffusion models

presented in the section 1.3. However, a greedy algorithm makes the problem tractable and gives an approximation guarantee of 0.63. This means the k-set found using the greedy algorithm has an influence at least as good as 0.63 times the influence of the optimal k-set. The next section gives the details about the greedy algorithm.

1.2 Approximation theory

A rather classical way to approximate a $\mathcal{NP}$-hard problem is to adapt it in a greedy way. For the influence maximization problem, the idea would consist in working step by step and adding at each step the vertex maximizing the influence increase locally instead of looking at once for the best k-set overall. This greedy approximation has been treated by Nemhauser, Wolsey and Fisher [NWF78]. They defined a *natural hill-climbing algorithm* presented in listing 1.1.

```
1  A ← ∅
2  for i ← 1 to k do
3      v_i ← argmax_{v∈V\A} ( σ(A ∪ {v}) − σ(A) )
4      A ← A ∪ {v_i}
5  end for
```

Listing 1.1: Natural Hill-Climbing Algorithm for finding the k-set of maximum influence

The algorithm starts with an empty set A. A vertex v maximizing the marginal gain $\sigma(A\cup\{v\})-\sigma(A)$ is added to A at each iteration, in fact the vertex chosen is one maximizing $\sigma(A \cup \{v\})$ but the marginal gain is used to stress on the submodular function property presented next.

If the function $\sigma(\cdot)$ is *submodular* and monotone, Nemhauser et al. proved that the natural hill-climbing algorithm provides a $(1 - 1/e)$-approximation to the problem of finding a k-set A maximizing the value of $\sigma(A)$, e being the base of the natural logarithm. Thus, if A^* is an optimal set maximizing the function $\sigma(\cdot)$, the approximation guarantee is expressed as

$$\sigma(A) \geq (1 - 1/e) \cdot \sigma(A^*)$$

The guarantee to reach a $\sigma(A)$ slightly better than $1 - 1/e = 63\%$ of the optimal $\sigma(A^*)$ is thus achieved. To use this algorithm and be assured to have the approximation guarantee, the influence function must be submodular, and this function depends on the chosen diffusion model.

A *submodular function* is a function that maps subsets of a finite ground set U to non-negative real numbers $f(\cdot) : U \mapsto \mathbb{R}^+$ and satisfies a natural "diminishing returns" property : the marginal gain from adding an element to a set S is at least as high as the marginal gain from adding the same element to a superset of S.

$$f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T) \qquad \forall v \in U \text{ and } \forall (S, T) : S \subseteq T \subseteq U$$

The section 1.4 shows intuitively that the influence for the models of diffusion presented in the

next section are in fact submodular and than the greedy algorithm can thus be used with these models in order to find a k-set of maximum influence.

1.3 Models of diffusion

There are many ways of studying the diffusion of information in a network. The focus here is set on *operational models*, which analyze the step-by-step dynamics of propagation. Two well-known models of diffusion are the *Linear Threshold Model (LTM)* and the *Independent Cascade Model (ICM)*, both presented in [KKvT03].

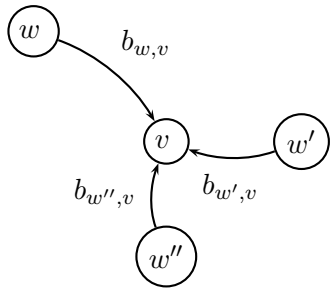
In both models, a vertex can only switch from the inactive state to the active one, and then remains active forever. The intuitive idea of LTM is that the probability for a vertex to become active will increase as more of its neighbours become active, and thus as time goes on. Here is a small real-life example : a new product comes on the market and several friends buy it. As more of those friends use it, they will eventually convince you to buy it as well. This is how the LTM model works. The second model, ICM, gives a single chance to a newly-activated vertex to try activating each of its inactive neighbours. The same example, taken from another point of view, illustrates ICM. This time, you were just convinced by the new product and bought it. You will thus talk to your friends about it immediately, but you will only have one chance to convince them. Whatever the result, you never try again. The probability that you succeed convincing each of your friends depends on your relation with each specific friend and is thus awarded a specific probability.

However, both these models suffer a critical issue. There is no way to compute the influence function $\sigma(\cdot)$ with an explicit formula. The only way to find a suitable value for $\sigma(\cdot)$, in order to solve the influence maximization problem using the natural greedy hill-climbing algorithm, is to approximate it by simulating the diffusion process several times and sampling the resulting active sets. This issue is discussed in the section 1.4. These simulations can take a very long time to achieve fine results, especially with large networks as shown by theoretical time complexities presented in chapter 2 and experimental results provided in chapter 4. A third model, *Shortest Path Model (SPM)*, developed in [KS06] is a specialization of the ICM model and provides a new and efficient way to compute the value of $\sigma(\cdot)$. It achieves a very important speedup at the expense of some precision. This issue is covered in chapter 4.

1.3.1 Linear Threshold Model (LTM)

In this model, the tendency for a vertex to become active increases monotonically as more of its predecessors become active. A vertex v is influenced by each of its predecessors w according to a weight $b_{w,v}$ such that $\sum_{w \in N^-(v)} b_{w,v} \leq 1$.

First, each vertex v of the network chooses a threshold θ_v uniformly at random in the interval $[0, 1]$. The diffusion process then unfolds deterministically in discrete steps.



At step t all active vertices remain active and an inactive vertex v becomes active if

$$\sum_{w \in N_a^-(v)} b_{w,v} \geq \theta_v$$

where $N_a^-(v)$ denotes the set of active predecessors of v. So a vertex becomes active when enough of its predecessors are themselves active. The threshold θ_v intuitively represents the different latent time before a vertex adopts the information. The values of θ_v are not fixed input, they are randomly chosen each time the process of diffusion is run in order to fill the lack of information about the network and the relations between its vertices. However, in some approach, this threshold is simply fixed to a given value for all the vertices of the graph, for example 0.5. The listing 1.2 summarizes the diffusion process for the LTM diffusion model.

```
1  Given
2     (a) a random choice of thresholds θ_v
3     (b) an initial set of active vertices A_0 ⊆ V
4
5  In step t
6     All vertices active in step t−1 remain active
7     Any inactive vertex v becomes active if
8
9           Σ_{w∈N_a^-(v)} b_{w,v} ≥ θ_v
```

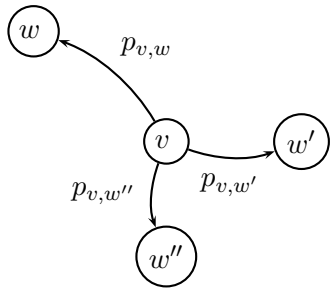
Listing 1.2: The Linear Threshold Model of diffusion (LTM)

The diffusion is a discrete step-by-step process. At each time t, there is a set of active vertices and according to this set of currently active vertices A_t, some inactive vertices can become active at next step A_{t+1}. The process of diffusion ends when, at a step t, there is no more inactive vertices that can become active.

1.3.2 Independent Cascade Model (ICM)

The Independent Cascade Model is quite different. The process also unfolds in discrete steps but according to a randomized rule stating that when a vertex v switches from the inactive state to the active one, this vertex is given a single chance to activate each of its inactive successor w and it will succeed with probability $p_{v,w}$.

The probabilities $p_{v,w}$ are constant input parameters linked to the network. If a vertex w has multiple newly activated predecessors, their attempts to activate w are sequenced in an arbitrary order.



If v succeeds to activate w at step t, the vertex w will become active at step $t+1$ and will try to activate its inactive successors at step $t+1$ so that they might become active at step $t+2$.

It is important to remember that each newly activated vertex v is given only one chance to activate each of its inactive successor. The diffusion process is summarized in the listing 1.3.

```
1 Given
2    (a) an initial set of active vertices A_0
3
4 In step t
5    If v first becomes active
6       v is given a single chance to activate each of it inactive successors
         w and succeeds with probability p_{v,w}
7       If v succeeds, w become active at step t+1
```

Listing 1.3: Independent Cascade Model (ICM)

As with LTM, the process of diffusion ends when, at a step t, no more vertices switches from the inactive state to the active one, that is no more vertices have to try to activate their successors. The probabilities $p_{v,w}$ represents characterize the strength of the relation between two vertices of the network, in the field of marketing, that is the force of persuasion but other interpretation can be given for other fields.

1.3.3 Shortest Path Model (SPM)

The main drawback of ICM diffusion model is that there is no explicit formula to compute efficiently the value of the influence $\sigma(\cdot)$ for a set of vertices, making it inefficient for large networks. Kimura et al. developed a new model of diffusion [KS06] that is an approximation of the ICM diffusion model. The model proposed is the *Shortest Path Model (SPM)*. An efficient way to compute the value of the influence with this model is also proposed.

The model of diffusion is exactly the same as ICM except that a vertex is eligible to become active only at a specified time in the process of diffusion. Each vertex v has the chance to become active only at step $t = d(A_0, v)$ with

$$d(A_0, v) = \min_{u \in A_0} d(u, v)$$

where A_0 denotes the set of initial active vertices and $d(u, v)$ denotes the length of the shortest directed path between the vertices u and v. A vertex v in the network can thus only receive information from the set A_0 through a shortest directed path from the set A_0 to the vertex v.

So, while with ICM a vertex v becoming active tries to activate all of its successors, with SPM, a vertex becoming active at time t will only be allowed to activate his successors w whom a shortest directed path from the initial set to w has a length of exactly $t + 1$.

With this new assumption, the influence of a set can be easily computed and so the greedy algorithm is applicable for large graphs using this diffusion model. The interesting question is obviously related to the quality of the approximation of the ICM model obtained with this model and this question is tackled in chapter 4.

The influence is easily expressed in the term of a function $P_t(v; A)$ and can be efficiently computed. The function $P_t(v; A)$ represents the probability that the vertex v becomes active at step t of the diffusion process given the initial set A. This function knows several special cases which require definite values. Let's denote $V_A = \{v \in V \mid d(A, v) < \infty\}$ the set of vertices reachable from A, then $P_t(v; A) = 0$ for any $t \geq 0$ if $v \notin V_A$. On the opposite, $\forall v \in A_0$, $P_t(v; A) = 1$ if $t = 0$ and 0 for $t > 0$. For the other vertices, the value of $P_t(v; A)$ is obtained using the following recursive definition :

$$P_t(v; A) = 1 - \left(\prod_{u \in N^-(v)} 1 - p_{u,v} P_{t-1}(u; A) \right)$$

with $N^-(v)$ denoting the predecessors of v, that is $N^-(v) = \{u \in V \mid (u, v) \in E\}$. This definition is quite intuitive : the probability that a vertex v becomes active at step t is computed as 1 minus the probability that the vertex v does not become active at step t. The latter one is simply the probability that none of the predecessors of v becomes active at step $t - 1$ and manages to activate v, which happens with probability $p_{u,v}$.

The influence of a set of vertices for the SPM model is expressed with this function as

$$\sigma_{SPM}(A) = \sum_{v \in V_A} P_{d(A,v)}(v; A) \tag{1.2}$$

which is simply the sum for each vertex $v \in V_A$ of the probabilities that the vertex becomes active at the time $d(A; v)$, that is the probability for the vertex v to be activated through the shortest directed path from the initial set A.

SP1 Model (SP1M)

A small variant of the SPM model is also proposed in [KS06]. In this variant, a vertex v can become active at time $t = d(A_0, v)$ as with the SPM model, but also at time $t = d(A_0, v) + 1$. A vertex can thus be activated at two times in the diffusion process. This new model aims

to better approximate the ICM model, in the sense that more activation paths are taken into account than with the SPM model. The influence can also be computed efficiently as

$$\sigma_{SP1M}(A) = \sum_{v \in V_A} \Big(P_{d(A,v)}(v; A) + P_{d(A,v)+1}(v; A) \Big) \tag{1.3}$$

Once again, the formula for computing the value of the influence is quite intuitive, it is the sum over the vertices $v \in V_A$ that the vertex v becomes active at the time $d(A; v)$ or $d(A; v) + 1$. The probability that a vertex v first becomes active at time t given an set targeted for initial activation A is simply given by

$$P_t(v; A) = \big(1 - P_{t-1}(v; A)\big) \left(1 - \prod_{u \in V^-(v)} (1 - p_{u,v} P_{t-1}(u; A)) \right)$$

The formula is quite intuitive, in order for a vertex v to first becomes active at time t given the initial targeted set A, the vertex must not become active at time $t+1$ and one of its predecessors must have become active at time $t - 1$.

1.4 Finding a k-set of maximum influence

The objective is, given a network $G = (V, E)$ and a diffusion model M, to find a k-set of maximum influence, that is a set $A \subseteq V$ of k vertices from the network that are the most influent, that is with a maximal value of $\sigma_M(A)$. Given that the function $\sigma_M(\cdot)$ is submodular, the greedy algorithm of Nemhauser et al. (listing 1.1) can be used to solve the problem and moreover, there is an approximation guarantee. The influence is a submodular function for the three models, see [KKvT03, KS06] for the formal proofs.

However the submodularity of the influence can be easily understood. Let's take a network and two sets of vertices $S \subseteq T \subseteq V$. Adding a vertex v to the set S will give a marginal gain $\sigma(S \cup \{v\}) - \sigma(S)$ greater than the marginal gain obtained by adding the same vertex v to the set T, indeed the set T is a superset of S and the information reached more vertices with T as initial set, and there is a great probability that the vertex v is among the vertices activated using T as initial set but that are not activated if S is used. And so, the marginal gain obtained by adding v to T is smaller than the marginal gain obtained if v is added to S.

For the two first models of diffusion presented in the previous section (LTM and ICM), there is no efficient way to compute the influence of a set A, and so, the only way to do this is to compute an estimate through several simulations. The process of diffusion is repeated R times and the influence is estimated as the mean of the size of the active set obtained for each run.

The value of the influence is thus given by

$$\hat{\sigma}_M(A) = \frac{1}{R} \sum_{i=1}^{R} \mid \varphi_M^i(A) \mid$$

with $\varphi_M^i(A)$ the set of active vertices for the i^{th} run given the initial set A using the model of diffusion M. The issue of how to choose R is addressed in the chapter 4.

The result of Nemhauser et al. has been extended in [KKvT03] to show that for any $\varepsilon > 0$, there is a $\gamma > 0$ such that by using $(1+\gamma)$-approximate values for $\sigma(\cdot)$, the result obtained for the influence maximization problem is a $(1 - 1/e - \varepsilon)$-approximation. So, the approximation guarantee remains even though the value of $\sigma(\cdot)$ is not known but only γ-approximated.

For the two last models (SPM and SP1M), there is an explicit formula (1.2 and 1.3) that is used to compute the influence in order to use the greedy algorithm.

It is important to note the difference between the two things implied in the problem of finding a k-set of maximal influence. The first thing is to define a model of diffusion of information M, this is the rules that explain how the information propagates. Next, an algorithm is used to choose the k vertices that gives a maximal influence for the considered model, that is a set A with a maximal value of $\sigma_M(A)$. This algorithm is called *the solver*.

Next section presents the implementation choices and the time and spatial complexities associated to the problems discussed for several models of diffusion.

2

Implementation and complexities

In order to better understand the properties and behaviours of the solutions presented in the previous chapter, they first had to be implemented. This chapter briefly presents what has been done during the process of implementing these solutions. The implementation has been done with the object-oriented model and the architecture is introduced in the first section with some classes hierarchies diagrams. The second part of the chapter gives more details about the implementation of each model and the greedy solver separately. It also goes through the effort of expressing the time and space complexities of all interesting solutions presented so far.

The implementation has been done using the Java programming language with the JUNG framework [OFN07] for all stuff related to the representation and manipulation of graphs. The appendix A gives more informations about the implementation, the programs and the light GUI provided.

2.1 Architecture

As introduced in chapter 1, there are two main dimensions in the influence maximization problem. First one is the algorithm used for finding a k-set of maximum influence in a network, which is called *the solver*. This mission has been achieved so far using the greedy algorithm presented in listing 1.1, but there are other options. The other dimension is the diffusion model, and its associated way of computing influence for a given set. These two dimensions have been implemented independently in order to be as generic as possible, and this section will explain how it all has been organized.

2.1.1 Influence Maximization Problem

The influence maximization problem is the problem of actually finding a k-set of maximum influence, based on some knowledge of the influence of any set of vertices. The figure 2.1 shows the class hierarchy of this first part. Two main solvers are considered. The first one has been introduced in chapter 1 and is the greedy algorithm suggested by Nemhauser, Wolsey and Fisher. The other main solution implemented is suggested in chapter 3 and is based on splitting the problem into subproblems using communities. A third solution is the brute-force, though it has no interest.

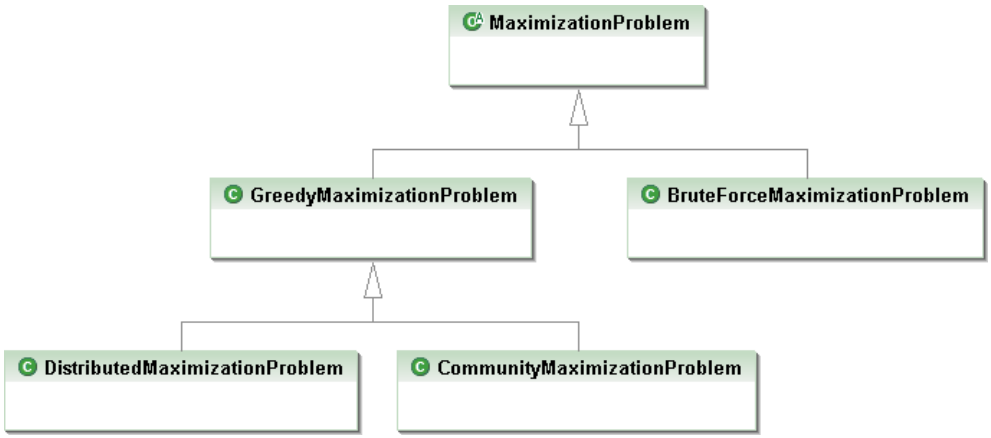


Figure 2.1: Class diagram for the maximization problem

The top most class, `MaximizationProblem`, is an abstract class representing a way to solve the influence maximization problem. A graph has to be specified in order to construct a new maximization problem. All constructors thus take a `Graph` object as single parameter. The most important method in this class allows to solve the maximization problem and its signature is

```
public void solve (DiffusionModel model, int k);
```

The method `solve` takes two parameters. The first one is the diffusion model that must be considered for solving the influence maximization problem and is presented in section 2.1.2 and the second parameter is the size of the k-set that must be found. When this method is called on an instance of `MaximizationProblem`, all the four main informations are specified : the graph G through the constructor, the solver through the concrete subclass used and the model M and k as parameters of the `solve` method.

The concrete subclasses of the `MaximizationProblem` are thus the several solvers that can be used for solving the influence maximization problem. The `BruteForceMaximizationProblem` class can be found at the first level. It will look for the k-set maximizing the influence by going through evaluating all possible k-sets. This method definitely is not efficient, and it shall not be considered.

At the same level, there is the `GreedyMaximizationProblem` class which consists in the implementation of the greedy algorithm presented in listing 1.1. This solver itself has two other subclasses. The first one, `DistributedMaximizationProblem`, is a distributed implementation used to reduce the running time by using a distributed version of the greedy algorithm using several computers across a network and the second one, `CommunityMaximizationProblem`, is the solver suggested in chapter 3 which goal is to speed up the greedy algorithm by splitting the problem into smaller subproblems using communities found in the graph.

These classes have been organized so that a new solver could easily be added and used in this environment. All that has to be done is to create a new class extending `MaximizationProblem` and implementing one method whose specification is given in appendix A. Moreover, an extension

of an existing solver can also be introduced quite easily.

2.1.2 Diffusion models

There are two dimensions in the influence maximization problem : the solver and the model. The previous section is about the solver and this one is about the models. Each kind of diffusion model has its own class and the class hierarchy diagram naturally looks like figure 2.2.

As explained in chapter 1, SPM and SP1M are special cases of ICM. They thus both are subclasses of ICM while ICM and LTM are the only two direct subclasses of `DiffusionModel` which is the abstract class at the top of the class hierarchy. `DiffusionModel` being an abstract class is not its only common point with the `MaximizationProblem` class. Once again, the graph is needed in order to construct a new instance of a diffusion model. It is thus required as a parameter of the constructor.

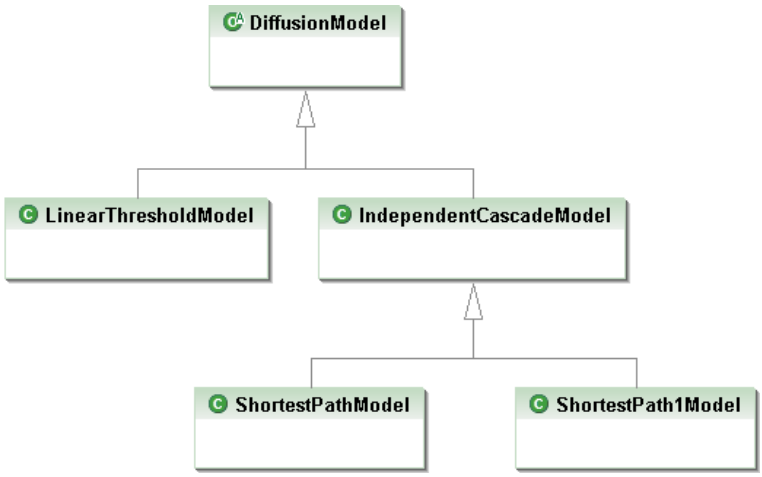


Figure 2.2: Class diagram for the models

There are two important methods in the `DiffusionModel` class. The first one computes the influence of a given set of vertices A and is used many times by the greedy search for a k-set with maximal influence.

```
public RandomDouble getInfluence (Set<Vertex> A);
```

The method returns a `RandomDouble` object, that is a mean and a standard deviation. Indeed, the influence of a set cannot be computed exactly with the ICM and LTM model and its thus an approximation that is returned. For the SPM and SP1M models, it is an exact value and the standard deviation is simply set to 0.

The second important method is used to run the process of diffusion starting from a specified active set A_0.

```
public Set<Vertex> runProcess (Set<Vertex> A0);
```

This method is used to estimate the influence of a set for the ICM and LTM models. It is not its only application though as simulating some diffusions appears useful in several other circumstances. One of these will be introduced in the chapter 5.

2.2 Complexities

Complexities are a very important matter to keep in mind because they can affect the choice of which method to use with which model of diffusion while a specific instance of the influence maximization problem has to be solved. They also have the power to turn a marvellous idea into an useless one. There are two different complexities, though people tend to focus on time complexity.

Time complexity of an algorithm is the number of steps that it takes to solve an instance of the problem as a function of the size of the input.

Space complexity of an algorithm measures the amount of memory space required by the algorithm as a function of the size of the input.

The exact number of steps not being of much interest, complexities are usually expressed using the big-oh notation. This means, looking at the formula expressing the number of steps in terms of the input size or other parameters, only the term with highest degree for each parameter is taken into account.

The goal of this section is to express complexities of the greedy solver according to model of diffusion used. This requires a "divide and conquer" approach combined with good approximations. Once the complexity of each piece of the algorithm has been estimated, they are combined to give the complexity of the algorithm as a whole.

The complexity of a process does not only depend on the underlying algorithm. The implementation often lowers time complexity by using good data structures, thus increasing space complexity. The incoming analyses takes into account some characteristics of the implementation as well as the underlying algorithms.

2.2.1 Greedy Solver

The solver part is common to every model. The greedy solver is the only one presented so far. The listing 2.1 shows again the greedy solver algorithm.

The implementation does not change anything crucial about this algorithm and it can thus be used to draw conclusions about complexities. It is quite obvious that there are two levels of iteration in this algorithm. The outer one consists of choosing k times a best vertex, while choosing a best vertex once consists in evaluating marginal influence of each vertex that has

not been selected yet. The time complexity is thus $\mathcal{O}(k \cdot (n-k) \cdot getInfluence)$. This can be reduced to $\mathcal{O}(kn \cdot getInfluence)$ as $n \gg k$.

```
1  A ← ∅
2  for i ← 1 to k do
3      v_i ← argmax_{v∈V\A} (σ(A ∪ {v}) − σ(A))
4      A ← A ∪ {v_i}
5  end for
```

Listing 2.1: Natural Hill-Climbing Algorithm for finding the k-set of maximum influence

The space complexity simply consists in a copy of the list of vertices selected so far, and in several stand alone variables, which thus yields a space complexity of $\mathcal{O}(k)$.

2.2.2 Evaluating Influence

Next step consists in considering the complexity of evaluating the influence associated with the selection of a vertex. This consists, given a current initial set, in evaluating the influence of this set to which another vertex has been added. This process of evaluating influence is specific to the model of diffusion and is thus analysed separately for each model of diffusion.

ICM

The process of finding out an influence value with ICM goes through repeating numerous times the simulation of propagation, and eventually processing an average on their influence results as explained in the previous chapter at section 1.4. This means that the process of simulating a propagation will a certain number of times R. The process of simulating one propagation has a time complexity of $\mathcal{O}(n+m)$. It is thus considered as $\mathcal{O}(m)$ as $m \gg n$ is expected for real networks. The time complexity for estimating the influence of a set A is thus $\mathcal{O}(Rm)$. The reason why R is considered, though it is not a variable linked to the size of the problem, is that it usually comes to very high values such as $10,000$ for example. It therefore plays a big role in the actual running-time and should be taken into account. The listing 2.2 summarizes how the influence is computed.

```
1  mean ← 0
2
3  for i ← 1 to R
4      call runprocess (ICM, A)
5      mean ← mean + number of activated vertices
6  end for
7
8  σ_ICM(A) ← mean/R
```

Listing 2.2: Computing the influence for a set A with the ICM model of diffusion

The only required data structure is the graph itself since the mean is computed incrementally, and the space complexity is thus considered as $\mathcal{O}(n+m)$.

SPM

The process is far more complex for SPM, though it is actually much faster. The influence for the SPM model is not computed by a simulation as for the ICM model but by using an explicit formula explained in the previous chapter (equation 1.2) which is the sum for each vertex v of the graph of the probability for that vertex to be activated at a certain time which corresponds to the length of the shortest directed path from the initial set A to the vertex v.

The distances $d(A, v)$ must thus be computed and this computation is done in an efficient way by using special data structures. The first step is thus to initialize these data structures. Note that this is a perfect example of reducing time complexity by increasing space complexity. The initialization consists in processing distances between every pair of vertices, which is done by launching a *Dijkstra* from each vertex of the graph. The Dijkstra's algorithm is known to have a $\mathcal{O}(nD \log n)$ time complexity, with D being the average out-degree of a vertex which achieves a time complexity of $\mathcal{O}(n^2 D \log n)$ for this initialization phase.

Once the data structures have been initialized, the core of the process is to evaluate the probability for each vertex to end up activated which is computed using the recursive formula for the $P_t(v; A)$ function. This requires the propagation of probabilities through the graph and appears to be managed in time $\mathcal{O}(n+m)$.

$$P_t(v; A) = 1 - \left(\prod_{u \in N^-(v)} 1 - p_{u,v} P_{t-1}(u; A) \right)$$

Against all odds, the time complexity of those two parts should not be added. In fact, the complexity which is evaluated here is the complexity of finding a best vertex for one step of the greedy solver. The initialization occurs only once for the whole greedy process. It is thus at a higher level and should be added to the complexity of the greedy.

The space complexity is more relevant for SPM than for ICM. The data structure is a $n \times n$ matrix containing distances between each directed pair of vertices, plus two rows for ease of use. The space complexity is thus in $\mathcal{O}(n^2)$.

SP1M

There is no real difference for the complexities between SP1M and SPM. SP1M is a bit longer to process in practice because the propagation of probabilities is more complex but it does not affect the complexities though.

Summary

The table 2.1 gives a summary of the time and space complexities for the greedy solver used with the ICM, SPM and SP1M model of diffusion. The complexities are also given for real cases, indeed in practice graphs are sparse, that is $m = \mathcal{O}(n)e$.

Model	Theoretical		In practice	
	Time	Space	Time	Space
ICM	$\mathcal{O}(knRm)$	$\mathcal{O}(n+m)$	$\mathcal{O}(kn^2R)$	$\mathcal{O}(n)$
SPM	$\mathcal{O}(n^2D\log n + knm)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2D\log n + kn^2)$	$\mathcal{O}(n^2)$
SP1M	$\mathcal{O}(n^2D\log n + knm)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2D\log n + kn^2)$	$\mathcal{O}(n^2)$

Table 2.1: Time and space complexities for finding a k-set of maximum influence for several diffusion models on a graph with n vertices and m edges. R is the number of runs for the ICM model. D is the average out-degree of the vertices. In practice the graphs are sparse and $m = \mathcal{O}(n)$

The best theoretical time and space complexity are obtained for the ICM model, especially if R is not taken into account but that is only theoretical as shown in chapter 4. Considering the high value of R, it can be considered that SPM and SP1M actually achieve a better time complexity than ICM. As explained earlier, this improved time complexity comes with an increase in space complexity. This space complexity in $O(n^2)$ remains fully manageable though and should be of no big concern, even for problems with a great size.

The next chapter explains in details how a better speedup in time complexity can yet be achieved by dividing the influence maximization problem into subproblems with smaller size. The idea is quite simple, it consists of splitting the graph into smaller subgraphs and this partition is done using communities detected in the graph.

3

Communities

This third chapter is the core of this work. While the two first chapters introduce the field and its most popular techniques and models, this third chapter shall suggest new opportunities to improve, or actually speed-up, influence maximization techniques by using communities.

The concepts of community and community detection are first introduced. Their interest for influence maximization is then explained. A community-based solver is eventually suggested and criticized, as well as several available improvements.

3.1 Definition

A *community* can intuitively be viewed as a dense subgraph within a bigger but sparser graph. A more formal definition [Pon06] can be given if the considered communities are disjoints. Only disjoint communities are used for the new solvers suggested in this chapter.

From an undirected graph $G = (V, E)$, a *community detection algorithm* finds a partition $\mathcal{P} = \{\mathcal{C}_1, \cdots, \mathcal{C}_k \mid \mathcal{C}_i \cap \mathcal{C}_j = \emptyset \text{ for } i \neq j \text{ and } \bigcup_i \mathcal{C}_i = V\}$ of the vertices of the graph, such that a given quality function $Q(\mathcal{P})$ on the partition is maximized.

Each element of the partition $\mathcal{P}$ is considered as a *community*. This partition induces two kinds of edges. Those linking two vertices from the same community are called *intra-community edges* while others, joining two vertices from different communities, are called *inter-community edges*. If c_u denotes the community of the vertex u, an edge $\{u, v\}$ is either an intra-community edge if $c_u = c_v$ or an inter-community edge if $c_u \neq c_v$.

Figure 3.1 shows an example of a randomly generated graph with 1000 vertices and 10 communities. The graph shown is directed but the communities have been extracted on the corresponding undirected graph using the algorithm presented in section 3.2.

3.1.1 Modularity

The *modularity*, defined in [New06], is a function evaluating the quality of a partition as well as the community structure of an undirected graph. This means that a partition made on a graph

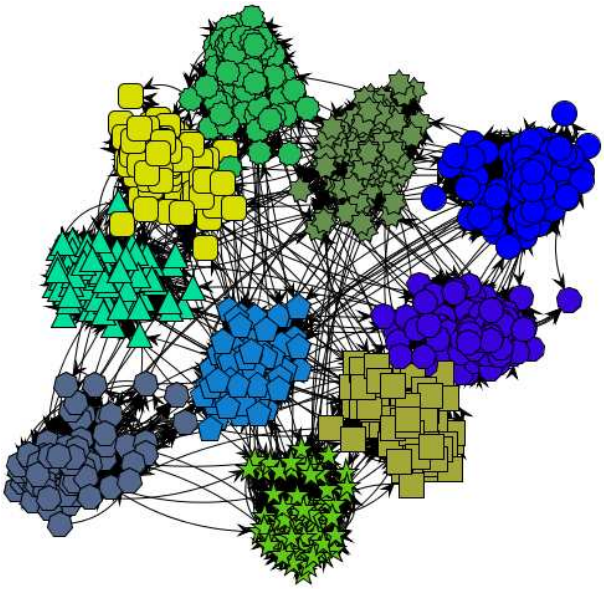


Figure 3.1: Community graph example (1000 vertices, 10 communities)

without any community structure will be considered as a bad partition even if it was the best available on this graph.

The modularity can be written as

$$Q^M = \frac{1}{2m} \sum_{u,v} \left(A_{uv} - \frac{d(u)d(v)}{2m} \right) \delta(c_u, c_v) \tag{3.1}$$

where A is the adjacency matrix of the graph ($A_{uv} = 1$ if vertices u and v are adjacent, and 0 otherwise), $d(u)$ is the degree of the vertex u and $\delta(\cdot, \cdot)$ is the Kronecker's delta.

More intuitively, the modularity of a partition $\mathcal{P}$ compares the number of intra-community edges for the partition $\mathcal{P}$ with the number of intra-community edges that would be found by keeping the same communities and vertex degrees but redistributing randomly all edges throughout the graph. The second term of the sum $\frac{d(u)d(v)}{2m}$ actually is the probability of having an edge between the vertices u and v after redistributing, and is thus subtracted from A_{uv} whose value is 1 if there is an edge between the vertices u and v. The Kronecker's delta is used to count the intra-community edges. A partition is good if there is more intra-community edges than in an equivalent graph whose edges are randomly distributed.

An alternative form of the modularity is given in [CNM04] and seems more intuitive and explicit. Let's first define two quantities :

$$e_{ij} = \frac{1}{2m} \sum_{uv} A_{uv} \delta(c_u, i) \delta(c_v, j)$$

is the fraction of edges that join vertices in community i to vertices in community j, and

$$a_i = \frac{1}{2m} \sum_v d_v \delta(c_v, i)$$

is the fraction of ends of edges that are attached to vertices in community i. Given these two quantities, the modularity can be expressed as

$$Q^M = \sum_i (e_{ii} - a_i^2) \tag{3.2}$$

The modularity of a given partition is comprised between -1 and 1. As the modularity increases, better is the partition and so the community structure. To conclude this introduction to modularity, Newman et al. pretend in [New06] that a graph has a community structure when there is a partition whose modularity is greater than 0.3.

3.2 Community detection

Many community detection algorithms have been developed so far for undirected graphs and a small summary is available in [DDDGA05]. One of those interesting methods is developed in [LP05]. It defines a distance between (groups of) vertices based on random walks while applying a hierarchical clustering based on those distances to find the communities. The proposed community detection algorithm based on the diffusion process, developed in chapter 5, works in a similar manner. Several concepts introduced in this section while presenting the approach used in [LP05] are thus used again in chapter 5.

3.2.1 Random walk

A *random walk* in a graph is a random process of exploration. A walker, starting on a vertex, can travel through the graph going from a vertex to another simply by following the edges. Once the walker is on a vertex u, he may either stay on the same vertex or go to a neighbour. Note that the opportunity for the walker to stay on the same vertex implies the presence of a loop edge on each vertex. At each time step, the walker thus chooses randomly and uniformly which neighbour (itself included) to go to. In other words, each neighbour (itself included) will be joined with a $1/d(u)$ probability, with $d(u) = \sum_v A_{uv}$.

A random walk can be seen as a Markov process with the *transition matrix* P given by $P_{uv} = \frac{A_{uv}}{d_u}$. The entry P_{uv} thus gives the probability for the walker to go from vertex u to vertex v at some step in the walk. The powers of the matrix P give interesting informations. The entry P_{uv}^t actually gives the probability for the walker to go from vertex u to vertex v through a walk of t steps.

Two other properties linked to the matrix P are important :

1. When t tends to infinity, for a random walk starting at vertex u, the probability to end at a vertex v only depends on the degree of v, this is vertices with high degree attract walkers to them.

$$\forall u \in V \ : \ \lim_{t \to \infty} P_{uv}^t = \frac{d(v)}{\sum_k d(k)}$$

2. The probabilities to go from the vertex u to the vertex v and from the vertex v to the vertex u during a random walk of fixed length t have a ratio that only depends on the degrees of the two vertices.

$$\forall u, v \in V \ : \ d(u) P_{uv}^t = d(v) P_{vu}^t$$

3.2.2 Defining a distance

A distance between the vertices of the graph can be defined using the random walks of a fixed length t and the matrix P_{uv}^t. The main goal of this distance is to capture the community structure of the graph. It should therefore tend to be larger for two vertices coming from different communities and smaller for vertices from the same community.

The first important thing is the choice of the length t of the random walks. It should be large enough to gather enough information about the graph topology but should, on the other side, avoid being too large because of the effect predicted by first property of random walks, described above.

Before using the transition matrix to compare two vertices, some observations should be pointed out :

1. If two vertices u and v are in the same community, the probability P_{uv}^t will surely be high. But the fact that P_{uv}^t is high does not necessarily imply that u and v are in the same community.
2. The probability P_{uv}^t is influenced by the degree $d(v)$ because the walker is attracted by high degree vertices as explained earlier.
3. Two vertices u and v of a same community tend to "see" all other vertices in the same way, that is $\forall k \ : \ P_{uk}^t \simeq P_{vk}^t$

The distance between two vertices is thus given this definition, with the numerator expressing the third observation and the denominator expressing the second :

$$r_{uv} = \sqrt{\sum_{k=1}^{n} \frac{(P_{uk}^t - P_{vk}^t)^2}{d(k)}} = \left\| D^{-\frac{1}{2}} P_{i\bullet}^t - D^{-\frac{1}{2}} P_{j\bullet}^t \right\| \tag{3.3}$$

One can notice that this distance can also be seen as the L^2 distance between the two probability distributions $P^t_{u\bullet}$ and $P^t_{v\bullet}$.

In order to use a hierarchical clustering algorithm though, a distance must also be defined between two clusters. An intermediate step is to define the probability P^t_{Cv} to go from a vertex in cluster C to the vertex v during a random walk of length t :$P^t_{Cv} = \frac{1}{|C|}\sum_{u\in C} P^t_{uv}$. Once this is done, the distance between two clusters can easily be defined :

$$r_{C_1C_2} = \sqrt{\sum_{k=1}^{n} \frac{(P^t_{C_1k} - P^t_{C_2k})^2}{d(k)}} \tag{3.4}$$

3.2.3 Hierarchical clustering

Using the distance between clusters, the communities can be detected in a graph using a simple hierarchical clustering. The global structure of a hierarchical clustering is given in listing 3.1. It starts with n clusters, each containing a single vertex, and iteratively gathers the two nearest clusters to eventually obtain a single cluster containing all the vertices of the graph.

```
1  HIERARCHICAL-CLUSTERING (G)
2
3  P ← V
4  while |P| != 1 do
5      find a pair (C1, C2) with smaller distance d(C1,C2)
6      C3 ←C1 ∪ C2
7      P ← P \{C1, C2} ∪ {C3}
8  end while
```

Listing 3.1: Hierarchical clustering

In order to get connected communities, only adjacent clusters (having at least one edge connecting them) are eligible for merging. Clusters that should be merged are chosen according to Ward's method : the pair of clusters minimizing the mean σ_k of squared distances between each vertex and its community

$$\sigma_k = \frac{1}{n}\sum_{C\in\mathcal{P}_k}\sum_{i\in C} r^2_{iC}$$

However, the problem of minimizing σ_k for each k is known to be $\mathcal{NP}$-hard. To avoid this complexity dead-end, the algorithm chooses the two clusters minimizing the variation $\Delta\sigma(C_1, C_2)$ occurring if they were to be merged. Because the distance defined between vertices based on the random walks is an Euclidean distance, this variation can be directly computed as

$$\Delta\sigma(C_1, C_2) = \frac{1}{n}\frac{|C_1||C_2|}{|C_1| + |C_2|} r^2_{C_1C_2} \tag{3.5}$$

When two clusters are gathered together in order to form a new cluster, the distances between this new cluster and all the other clusters must be computed so that the clustering can continue. These distances can be computed efficiently thanks to the fact that the distance is euclidean

$$P^t_{C_1 \cup C_2} = \frac{|C_1| P^t_{C_1} + |C_2| P^t_{C_2}}{|C_1| + |C_2|}$$

3.2.4 Complexity

This algorithm is quite efficient, the time complexity is $\mathcal{O}(mnH)$ qui m the number of edges, n the number of vertices and H is the height of the dendrogram. In practice, the graph is sparse ($m = \mathcal{O}(n)$) and the dendrogram is balanced ($H = \mathcal{O}(\log n)$) and the time complexity reduces to $\mathcal{O}(n^2 \log n)$.

3.3 Using communities to improve speed

As shown in chapter 2, currently existing solutions to find the best k-set all have high complexities relative to the graph's size. As those solutions can hardly be used on real-size problems, either the complexities of algorithms themselves or the size of the graphs have to be reduced. Using communities as suggested in this chapter fits in the second category, though it achieves reducing complexity.

The method proposed here is a new solver for the influence maximization problem. It thus achieves the same functions as the greedy algorithm presented in listing 1.1. This means that all models remain the same, though they are used differently. This new solver and its evolutions are implemented in the `CommunityMaximizationProblem` class (figure 2.1).

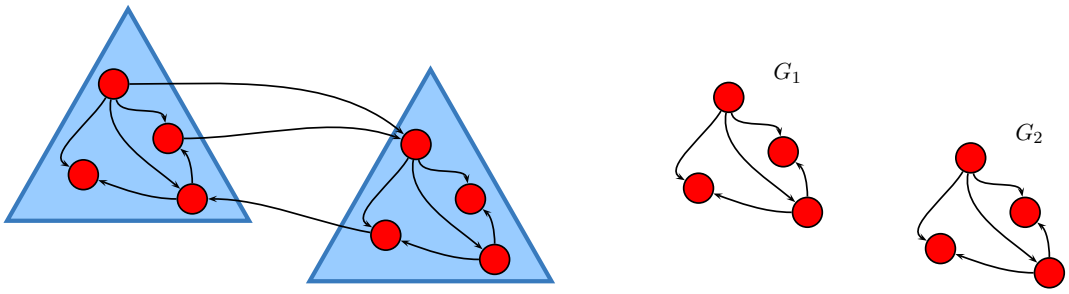
3.3.1 Naive algorithm

The first naive idea is to split the graph into subgraphs, each subgraph corresponding to a community. All inter-community edges are fully forgotten so that only c totally independent subgraphs remain. The removal of inter-community edges is the reason why communities should be detected carefully, as to minimize data losses. Let G_j be the graph corresponding to the j^{th} community.

$$G_j = (V_j, E_j) \quad \text{with } V_j = \{v \in V \mid c_v = j\} \text{ and } E_j = \{(u, v) \in E \mid c_u = c_v = j\}$$

Figure 3.2 shows a graph with two communities and some inter-community edges. The graph is split into two smaller ones and all inter-community edges disappear.

The goal of the community-based solver remains the same, finding the k-set of maximal influence, and the greedy approach also remains, thus adding at each step the vertex giving the greatest



(a) The graph G with 2 communities

(b) The two graphs G_1 and G_2

Figure 3.2: Using communities in a naive way

marginal gain. The difference brought by communities lies in the evaluation of the marginal influence gain. While the former solver evaluated it on the whole graph, the new one evaluates it only on the vertex's associated subgraph. This leads to an important speedup as the sizes of considered graphs are smaller, but it might also imply an important accuracy loss.

Let's define the intra-community influence of a vertex as the influence that a vertex has in his own community, using only edges from this same community to propagate information. The intra-community influence of a set of vertices $A \subseteq V$ in the j^{th} community is denoted $\sigma_j^i(A)$.

The solver's goal still is to find a k-set A of maximal influence on the graph G. Let A_j be the current set of selected vertices from the j^{th} community. At each step of the algorithm, a vertex v_j maximizing the marginal influence $\sigma_j^i(A_j \cup \{v_j\}) - \sigma_j^i(A_j)$ in subgraph j is considered, and so for each subgraph. These vertices are grouped to form a set of candidates $L = \{\langle v_j, \sigma_j^i(A_j \cup \{v_j\}) - \sigma_j^i(A_j)\rangle \mid 1 \leq j \leq c\}$. The next selected vertex to be added to set A will be the one offering the greatest marginal gain among set L, say v. Once v is added to A, the algorithm must update A_{c_v} and find the next best vertex in the c_v community. The algorithm is presented in a more formal way in listing 3.2.

This algorithm, though being a pretty rough approximation of the initial problem, gives quite good results depending on the graph and model used. It performs especially well when used with the SPM diffusion model on graphs with very good community structure (modularity $Q^M \geq 0.6$). Experiments and results are detailed in chapter 4.

However, the initial goal of this new solving algorithm is to provide an improvement in the required time for finding the k-set of maximal influence. The number of communities in a graph tends to be in $c = \mathcal{O}(\log n)$. Therefore, if communities are roughly of the same size, the initial problem on a graph of size n is roughly reduced to solving c problems of size $\frac{n}{\log n}$.

3.3.2 Drawbacks of the naive algorithm

The drawbacks of the naive algorithm come from its approximations. They come from the fact that a graph is split into smaller subgraphs and a lot of information is lost. This section tries

```
1  A ← ∅
2  A_j ← ∅ for 1 ≤ j ≤ c
3  L ← ∅
4
5  for j ← 1 to c do
6      Choose a vertex v ∈ G_j maximizing σ^i_j(A_j ∪ {v}) − σ^i_j(A_j)
7      L ← L ∪ {⟨v, σ^i_j(A_j ∪ {v}) − σ^i_j(A_j)⟩}
8  end for
9
10 for i ← 1 to k do
11     Choose a couple ⟨v, σ⟩ in L with maximal σ
12     A ← A ∪ {v}
13     A_{c_v} ← A_{c_v} ∪ {v}
14     L ← L \ ⟨v, σ⟩
15
16     Choose a vertex v ∈ G_{c_v} \ A maximizing σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v})
17     L ← L ∪ {⟨v, σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v})⟩}
18 end for
```

Listing 3.2: Naive Community-based Algorithm for finding the k-set of maximum influence

to list the main information losses and next section suggests several improvements in order to compensate these losses.

The first important issue due to removal of inter-community edges appears when a vertex from one community is linked to an influent vertex in another community. The figure 3.3 illustrates such a case.

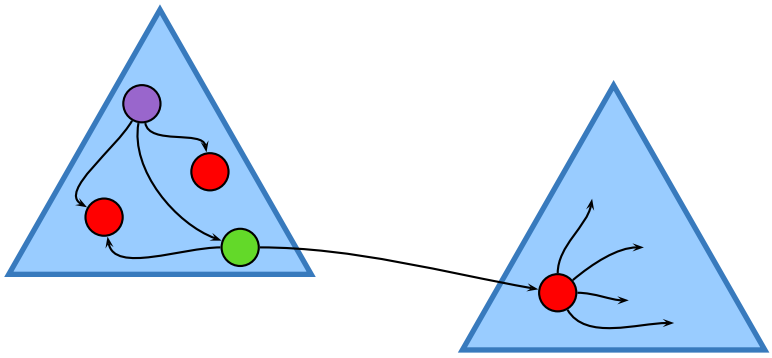


Figure 3.3: Inter-community edge to an influent vertex

The purple vertex in the left community would be selected by the naive algorithm because its intra-community influence is the greatest. The green vertex of the left community might however be a better choice for the global influence because of its link to a high influence vertex from another community. The example taken here implies a link to vertex with a high influence but it is also valid for any number of links to vertices in other communities. All this information is lost and vertices with inter-community links are thus disadvantaged in the new process of selection.

This specific loss of information has a lower impact on high community-structure graphs, but its impact on results still should not be neglected as shown in chapter 4. An improvement trying to take into account these inter-community influences is introduced in section 3.3.3 under the name of *advanced solver*.

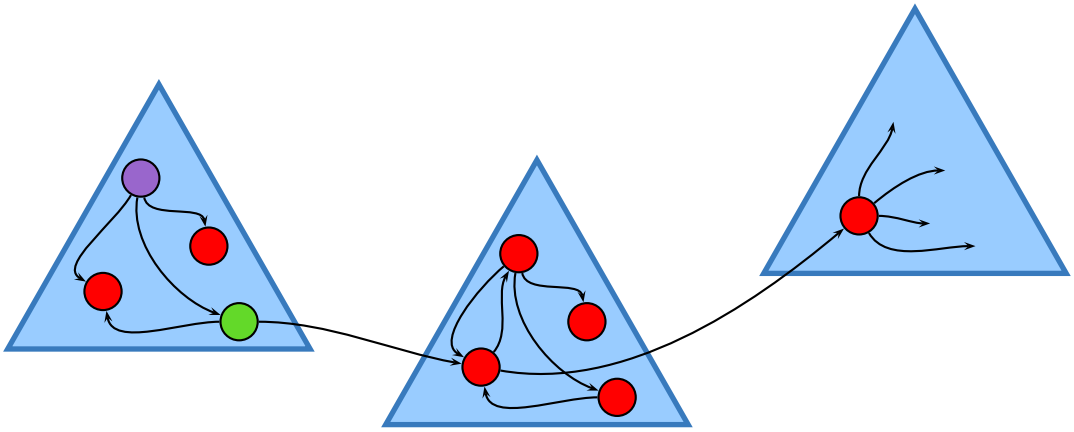


Figure 3.4: Influent path through communities

Another information loss comes from influent paths crossing several communities as shown in figure 3.4. Indeed, the influence of a vertex can come from the fact that it reaches several communities through some path jumping from a community to another. The purple vertex in the left community would have been selected by the naive algorithm. However, choosing the green vertex might appear being a better choice because of its community-jumping path. Such information can also be partially taken into account quite easily, and *iterative solver*, presented in section 3.3.3, tries to.

Yet another kind of issue arises. Both former information losses came from some ignored propagation opportunities, causing underestimation of several vertices influence. This issue, though being highly linked with previous ones, comes from a totally different point of view. When a vertex is added to the k-set A, its influence on vertices from other communities is not remembered. Therefore, the vertices it might reach in other communities will not be disadvantaged later on by his own selection, though they should. The problem is illustrated in figure 3.5. The green vertex is already in the k-set A and the three purple vertices are the candidates.

Let's suppose that the two candidates in the two right communities have exactly the same best intra-community influence σ^i, but one of them is reached by the already selected green vertex. The naive solver will thus chose one of the two vertices at random to add in the set A. A good choice though would be the one on the right because it is unreached yet and thus offers a higher global influence increase. This issue will not be addressed in this work, though it is probably partly solvable by keeping an expensive data structure up-to-date, and though it probably could achieve interesting results improvements.

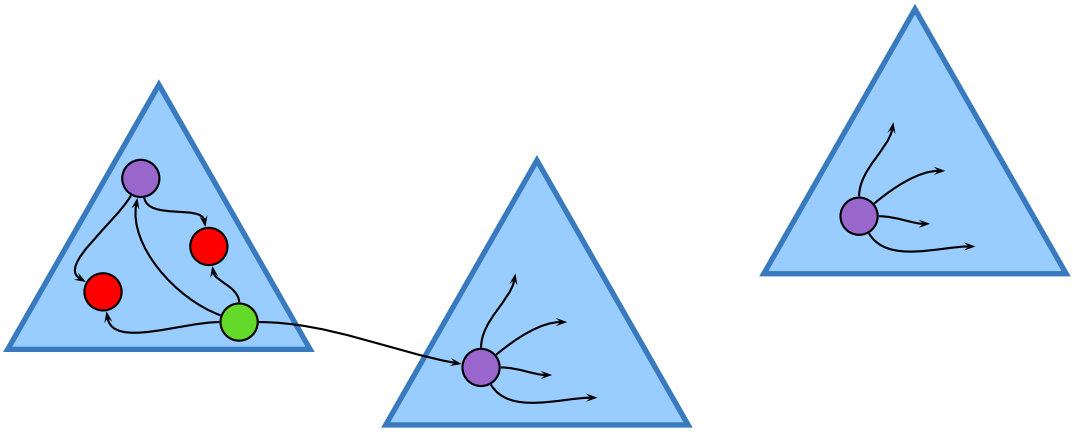


Figure 3.5: Inter-community edge from a selected vertex

3.3.3 Advanced and iterative algorithm

In order to recover partly from the first issue explained in previous section, a modification has been made to the naive algorithm. This modification leads us to a supposedly improved version called *advanced solver* from now on. The new version tries to take into account the influence related to outgoing inter-community edges when evaluating the influence of a vertex. In addition to the intra-community influence, a new quantity called the *inter-community influence* and denoted $\sigma^o(v)$ is used. This influence is defined for a vertex $v \in V$ and is defined as

$$\sigma^o(v) = \sum_{\substack{w \in N^+(v) \\ c_w \neq c_v}} p_{vw}\Big(\sigma^i_{c_w}(A_{c_w} \cup \{w\}) - \sigma^i_{c_w}(A_{c_w})\Big)$$

Using this new quantity, the advanced solver is expected to select vertices which are themselves expected to give a better global influence improvement. Of course, this improvement only takes into account one of the numerous impacts of inter-community edges on propagation and influence and remains an approximation, but it is a first step toward a better approximation. Note that this modification involves mainly the creation and updating of a data structure and very little calculation time. It should thus avoid, as shown in chapter 4, degrading the time complexity improvement brought up by the use of communities. The new algorithm, called *advanced solver* is shown in listing 3.3.

Like the naive algorithm, a list of candidates is built taking the best vertex from each community. The best vertex is the one giving the maximal marginal gain inside the community as well as outside, a vertex v maximizing $\sigma^i_{c_v}(A_{c_v} \cup \{v\}) - \sigma^i_{c_v}(A_{c_v}) + \sigma^o(v)$. The best candidate is then chosen and added to set A. The main difference with the naive algorithm is that, each time a candidate is chosen, all inter-community influences must be quickly recomputed and thus the list of candidates for each community must be reprocessed. This only consists in easy and fast operations though and should be of no concern.

The *iterative solver* goes even a step further. The idea is to propagate the information accessed by the advanced solver, inter-community influence. The iterative solver is thus an attempt

```
1  A ← ∅
2  A_j ← ∅ for 1 ≤ j ≤ c
3
4  for i ← 1 to k do
5      L ← ∅
6      for j ← 1 to c do
7          Choose a vertex v ∈ G_j maximizing σ^i_j(A_j ∪ {v}) − σ^i_j(A_j) + σ^o(v)
8          L ← L ∪ {⟨v, σ^i_j(A_j ∪ {v}) − σ^i_j(A_j) + σ^o(v)⟩}
9      end for
10
11     Choose a couple ⟨v, σ⟩ in L with maximal σ
12     A ← A ∪ {v}
13     A_{c_v} ← A_{c_v} ∪ {v}
14 end for
```

Listing 3.3: Advanced Community-based Solver for finding the k-set of maximum influence

to improve the advanced solver itself. While advanced solver processed the inter-community influence $\sigma^o(v)$ of v only to improve the estimation of the influence of v, the iterative algorithm shall use it to improve other vertices influence estimations too by propagating the information of how many vertices v is able to reach outside its own community.

The algorithm follows paths in the graph by doing a certain number $DEPTH$ of iterations. This value can for example be set to the number of communities in the graph.

```
1  A ← ∅
2  A_j ← ∅ for 1 ≤ j ≤ c
3  X_v ←σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v}) for v ∈ V
4  Y_v ←0 for v ∈ V
5
6  for j ← 1 to DEPTH do
7      for v ∈ V do
8          Y_v ←σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v}) + Σ_{w∈N^+(v), c_w≠c_v} p_vw · X_v
9      end for
10     Y ←X
11 end for
12
13 for i ← 1 to k do
14     Choose a vertex w ∈ V maximizing Y_w
15     A ← A ∪ {w}
16     A_{c_w} ← A_{c_w} ∪ {w}
17     X_v ←σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v}) for v ∈ G_{c_w}
18
19     for j ← 1 to DEPTH do
20         for v ∈ V do
21             Y_v ←σ^i_{c_v}(A_{c_v} ∪ {v}) − σ^i_{c_v}(A_{c_v}) + Σ_{w∈N^+(v), c_w≠c_v} p_vw · X_v
22         end for
23         Y ←X
24     end for
25 end for
26
27 Choose a vertex w ∈ V maximizing Y_w
28 A ← A ∪ {w}
```

Listing 3.4: Iterative Community-based Algorithm for finding the k-set of maximum influence

4

Experiments

The goal of this chapter is to confirm expected results from previous chapters. All interesting techniques are thus tested on several graph families in order to increase our understanding and to determine which techniques are best in which circumstances. Conclusions about the usefulness of the community solvers are also drawn in this chapter.

The two first sections explain the choices made for the testing. The first section presents the graphs sets used for the testing while the second section explains which technique is taken as a reference for the comparisons. The following sections present the several testing phases as well as their corresponding results and conclusions. And the last section eventually draws global conclusions about the testing process as a whole.

4.1 Studied graphs

The tests have been performed on different kinds of randomly generated graphs. Two graph families are especially studied because they reflect several interesting properties, some of which appear in real-world complex networks [New03, RSM+02]. Each type of graph will see 9 different graphs generated with same parameters. The goal is to have a representative set of each kind of graphs, preventing thus later results from being severely biased by the random aspect of generation.

The first class of graphs is made of vertices whose degrees distribution follows a power-law. There is some vertices in the graph that acts as *hubs*, that is vertices with a high degree linked to many other vertices and there is also a lot of vertices with a very small degree. Such graphs are called *scaled-free* graphs [AB02] and are representative because many real-world graphs appear to be scale-free. They have been generated using the method proposed in [VL05] and are thus undirected simple connected graphs. Directed graphs are needed tough for the experiments and they thus have been modified in two different ways. In the first version, each undirected edge is replaced by a single directed edge with random direction. In the second version, each undirected edge is replaced by both directed edges. Figure 4.1 illustrates an example of each kind of graph.

The graphs from the first version are *connected* but not *strongly connected*. They also contain isolated vertices, which cannot join any other vertex, these one having no influence. The graphs from the second version are strongly connected and twice as dense as those from the first version. The set of the graphs from the first version is called `random500_1` and the second set is called

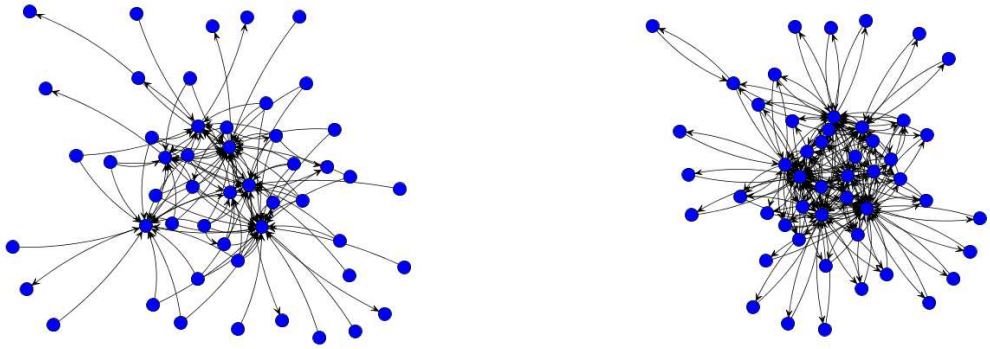


Figure 4.1: Examples of random graphs whose distribution of the degree of the vertices follows a power-law with 50 vertices for the two versions

`random500_2`.

The graphs from these sets do not have any community structure as their average modularity of $Q^M = 0.182194$ and for the `random500_1` set and $Q^M = 0.143180$ for the `random500_2` set testifies.

The second class of graphs is made of graphs with a high community structure, that is *modular* graphs. They are randomly generated using an algorithm created for this work and presented in appendix B. The generator takes several parameters : the number of communities, their size (average and standard deviation), the number of intra-community and inter-community edges (average and standard deviation) and the propagation probability which is common to all edges.

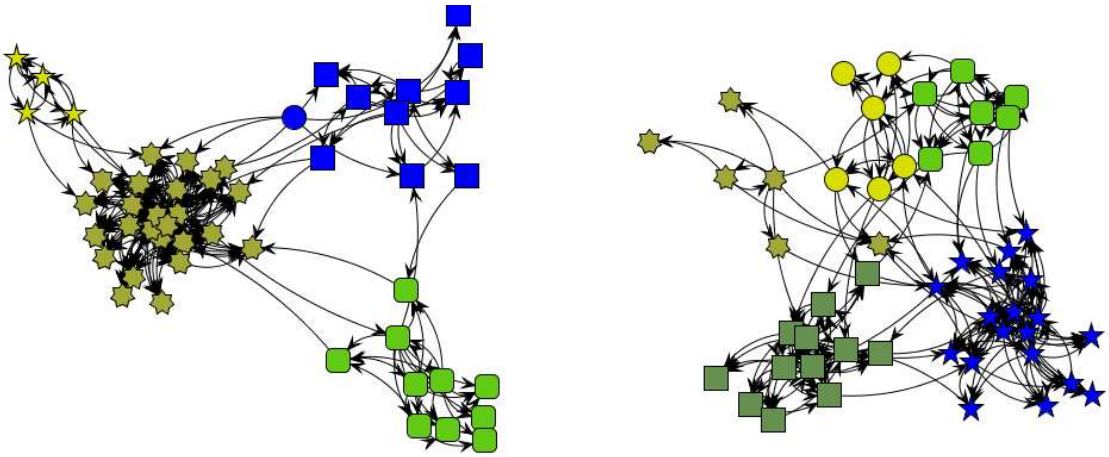


Figure 4.2: Examples of random graphs with community structure. The graphs from the first version have denser communities than those from the second version

Once again, two versions of such graphs are used in the tests. Both versions have a rather small number of inter-community edges. The difference comes from the number of intra-community

edges. The communities are very dense in the first version while they are a bit sparser in the second version. Both lead to a high modularity though, and thus to a pretty high community structure. The first version set of graphs is called `commu500_1` and leads to an average modularity of 0.793291 while the second set, called `commu500_2`, leads to an average modularity of 0.649664. It is no surprise that the second version has a lower modularity average as its communities tend to be sparser. Examples of both versions are shown in figure 4.2.

Table 4.1 gives some more information about the graphs from all four sets. The statistics presented are means obtained on the 9 graphs generated for each test set.

	Edges	Density	Communities	Modularity	Inter Edge	Intra Edge
`random500_1`	1250	0.005	162.111111	0.182194	46.1600%	53.8400%
`random500_2`	2500	0,01	283.888889	0.143180	67.7600%	32.2400%
`commu500_1`	14720.67	0.058883	5	0.793291	0.4826%	99.5174%
`commu500_2`	2697.33	0.010789	5.11	0.645508	15.2260%	84.7740%

Table 4.1: Some statistics about the graphs : the number of edges m, the density (m/n^2), the number of communities, the modularity and the percentage of inter-community and intra-community edges. The number of communities is found using the algorithm based on random walks (see section 3.2) with $t = 4$.

4.2 ICM_{10000} as a reference

This section presents two choices related to the way the tests are carried out and the results evaluated and compared. A model is first chosen to evaluate the quality of the k-sets returned by the several techniques in order to compare them. A technique is then chosen as a reference to evaluate how good are results obtained by other techniques compared to this reference.

All the techniques tested in this chapter have the same output, a k-set. In order to compare these techniques, a value describing the quality of these sets is needed. The influence seems a logical choice, but it raises the issue of knowing which influence should be considered. All the models consider different influences as propagation occurs differently. ICM has been chosen as the reference propagation model to evaluate the influence of sets to compare them. It has been chosen because ICM is the model that seems the closest to the reality of information propagation and that do not require too lot parameters as the LTM model for example. This last model of diffusion is completely ignored in this study.

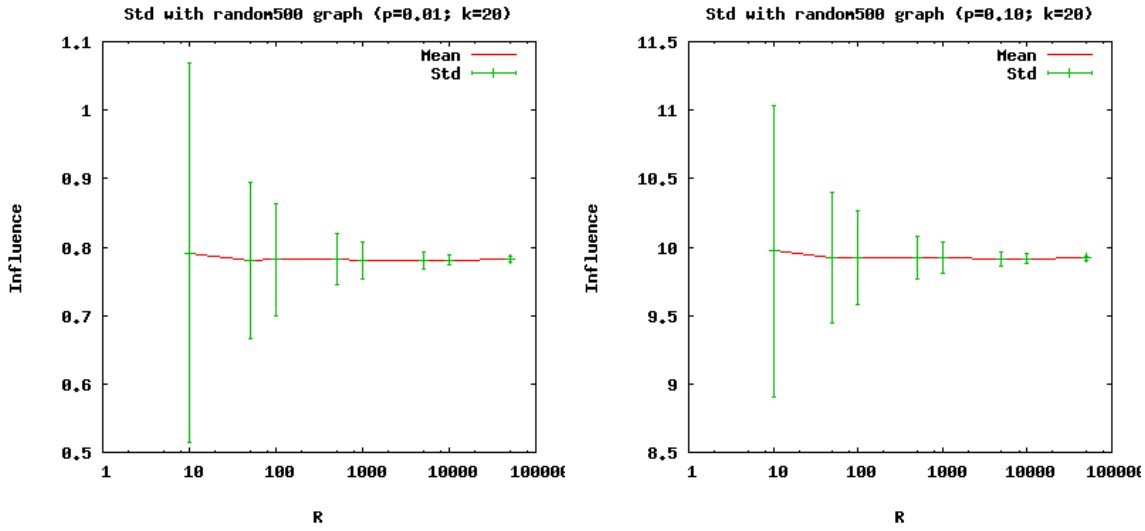
In order to carry out tests and draw conclusions, a reference is needed. The reference in this testing process is ICM_{10000} combined with the classic greedy solver. ICM_{10000} is the technique associated with the ICM model to process the influence. It consists in running $R = 10,000$ times the propagation process with the same initial set and averaging the size of active sets after each run. This average is then considered to be the influence of the initial set.

ICM_{10000} with the greedy solver is the best technique so far in terms of influence obtained. It has a big flaw though which is the computation time. This technique can not be used on real-sized

problems but certainly is an excellent comparison point for faster techniques to evaluate the precision lost through the speedup process.

There is no doubt ICM_{10000} combined with greedy solver is an excellent reference. The real issue is to know if ICM_{5000} or ICM_{50000} for example would be even better references. This is the issue left to consider in this section.

The test consists in running ICM_R several times on different graphs and different initial sets, not only the most influent. More details about the way these tests have been managed can be found in appendix C. The goal is to confirm that increasing R, whatever the circumstances, the graph or initial set, decreases the standard deviation on the obtained influence and thus ensures a more reliable result. Now, there is no real surprise and it does not answer the question which is to know at which R it is not worth increasing R anymore.

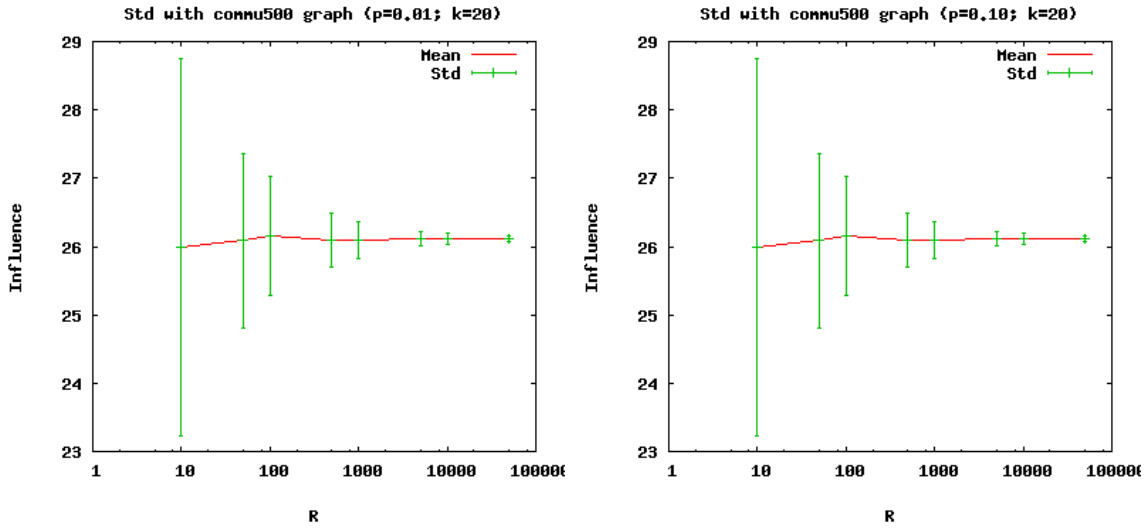


(a) Uniform $p_{v,w} = 0.01$

(b) Uniform $p_{v,w} = 0.10$

Figure 4.3: Mean and standard deviation of the influence computed with ICM_R for the `random500_1` set for a set of 20 vertices.

Looking at the graphs on figures 4.3 and 4.4 , 10,000 seems a pretty good value. It still offers a relatively big difference in standard deviation compared to 5,000 while 50,000 definitely looks like an useless effort. One could argue that the precision obtained by 10,000 is useless and 1,000 for example is far enough as the standard deviation is 0.03. There seems to be no point for $p = 0.01$ to go until 10,000 to reduce it from 0.03 to 0.01. The other graph with $p = 0.1$ presents a behavior that has not been talked about so far : increasing the probability increases not only the influence, but also the standard deviation. The change from 0.03 to 0.01 with $p = 0.01$ thus corresponds to a change from 0.15 to 0.5 for $p = 0.10$ which is less negligible. Taking into account the fact that propagation probabilities can be higher than 0.01 and that the goal here is to choose a precision reference for testing, not taking big care of the processing time as long as it is achievable for the tested graphs, $R = 10,000$ definitely appears as the best choice as a reference.



(a) Uniform $p_{v,w} = 0.01$

(b) Uniform $p_{v,w} = 0.10$

Figure 4.4: Mean and standard deviation of the influence computed with ICM_R for the `commu500_2` set for a set of 20 vertices.

4.3 Performance of ICM_{10000}

The first step of the testing process is to know in which circumstances ICM_{10000} is useful. There are in fact other very simple techniques that could maybe do as well in definite situations. There would be no need then to go through the calculations needed by ICM_{10000} or other techniques such as those developed in this work.

The selection of the k-set of maximal influence using the ICM_{10000} algorithm has been compared with the selection of k vertices using two simple ranking methods. The first ranking method, extremely simple, is based on the *degrees* of the vertices while the second one is based on the *betweenness* centrality measure.

4.3.1 Degree

This ranking is fairly simple. All the vertices of the graph are ranked according to their degree. A good question would be which degree to use, the in-degree, the out-degree, the sum or even the difference. Figure 4.5 gives an example of a small network to better understand the situation.

The first idea is to consider only the out-degree. After all, the information propagates from a vertex only through its outgoing edges. The green and purple vertices of the example both have an out-degree of 4 and could thus both be chosen. Choosing the purple one seems a better choice since it can reach more interesting vertices, including the green. This can be seen on two different point of views. The first one would be to say that vertices with low in-degree should

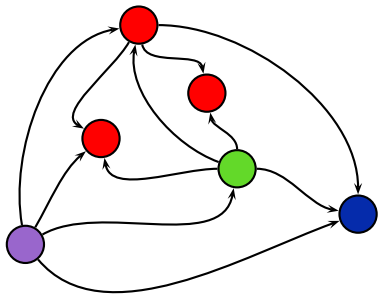


Figure 4.5: Selection of influent vertices based on their degree

be favoured because they have a lower probability to be reached if another vertex is chosen, or because the other vertex has a higher probability to be reached even if it is not chosen. The second point of view would be to take into account the out-degrees of the vertices that are directly reached by green and purple vertices. This would also lead to select the purple vertex.

The first ranking used is only based on the out-degree. All vertices get a rank consisting exactly of their out-degree. The ranks are then normalized by dividing by the sum of ranks of all the vertices of the graph.

$$c_D(u) = \left(\frac{1}{\sum_{v \in V} c_D(v)}\right) d^+(u)$$

A second ranking tries to manage the ties between vertices with the same out-degree. Each vertex is therefore ranked according to both its out-degree and the out-degree of its successors weighted with the probability $p_{v,w}$, as it has been suggested on the example.

$$c_{D_2}(u) = d^+(u) + \sum_{w \in N^+(u)} p_{u,w} \cdot d^+(w)$$

Both those rankings are extremely easy to compute. If they happen to achieve as good results as ICM in some circumstances, there will be no point in studying ICM further in those contexts.

4.3.2 Betweenness

Another interesting ranking method is the betweenness [Fre77]. The betweenness is a measure of how central a vertex is in a graph. The betweenness is evaluated by considering all the (directed) shortest paths between one pair of vertices after the other, and counting how many go through each vertex. Let σ_{st} denote the number of (directed) shortest paths from $s \in V$ to $t \in V$, where $\sigma_{ss} = 1$ by convention, the betweenness of a vertex u is then defined as

$$c_B(u) = \sum_{s \in V} \sum_{t \in V} \delta_{st}(v) \qquad \text{with } \delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Note that the value $\delta_{st}(v)$ is called the *pair-dependency* of a pair (s,t) on an intermediary v, that is the ratio of (directed) shortest paths between s and t on which v lies.

The value of betweenness for all the vertices of the graph can be computed efficiently by using for example the algorithm of Ulrik Brandes [Bra01] which runs in $\mathcal{O}(nm)$. Roughly, the algorithm performs a BFS exploration and stores necessary information. The cost of a BFS is $\mathcal{O}(m)$ and as the exploration is repeated for each vertex of the graph, the global complexity is then $\mathcal{O}(nm)$. This makes betweenness more time consuming than a degree ranking, but still light enough to be compared to ICM_{10000} in the same way as a degree ranking.

4.3.3 Results on the `random500` set

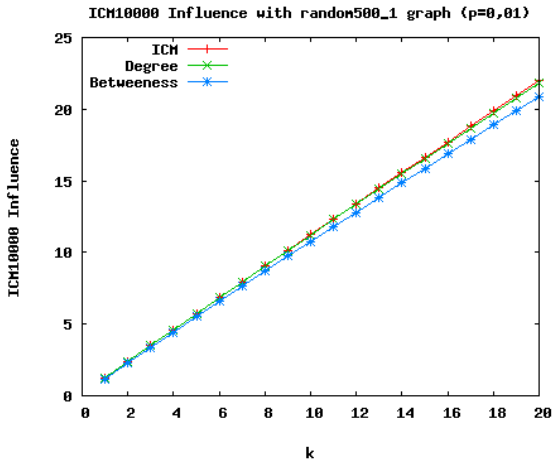
Figure 4.6 on page 39 shows the evolution of the ICM influence σ_{ICM} of k-sets found using the greedy ICM_{10000} solver, the degree and the betweenness, on scale-free graphs. The values are averages obtained on the whole set of graphs and have been computed with a uniform probability $p_{v,w}$ set to 0.01, 0.05 and 0.10.

The first observation that can be done on both sets is that the degree and betweenness rankings give k-sets with ICM influence quite comparable with the greedy ICM_{10000} for low probabilities ($p = 0.01$). There seems thus to be no point in computing the latter in such circumstances. This could be interpreted by thinking again about the propagation process itself. The propagation probability through a single edge being pretty low, a vertex will need lots of outgoing edges in order to contaminate even a single neighbour. It is thus very unlikely that two vertices achieve contaminating a same third vertex, wasting part of their work, ... It seems thus quite reasonable to see a degree ranking fit well though only taking care of choosing the vertices with most outgoing edges, and thus with highest standalone probability of propagating. There is no point in knowing anything about the graph structure or whatever with such a restrained propagation.

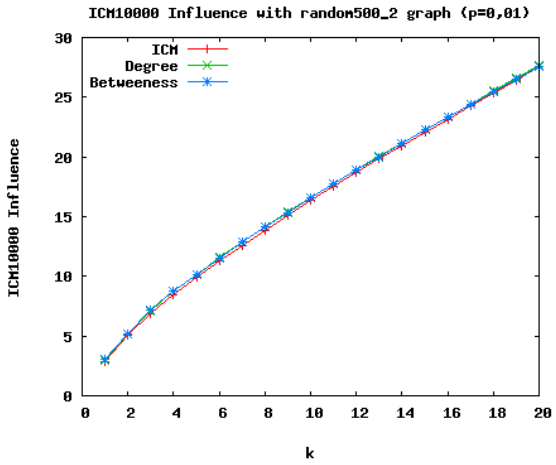
By looking at the graphs with a higher probability, it can be observed that the greedy algorithm becomes quite interesting from $p = 0.05$ on the first version `random_1` while for the second version `random_2` it only becomes interesting at $p = 0.10$, but becomes very interesting at once. No reliable explanation of such a huge difference between $p = 0.05$ and $p = 0.10$ for the second version is available.

The betweenness has a really interesting behaviour. While it does as well as the degree ranking on the second version, it fails miserably on the first version. No concrete explanation is brought up here. The only suggestion is that it might be a consequence of the very low density and special structure of the first version. These two might cause a low number of interesting directed paths, and thus a lack of representative information for the betweenness ranker which could explain the failure.

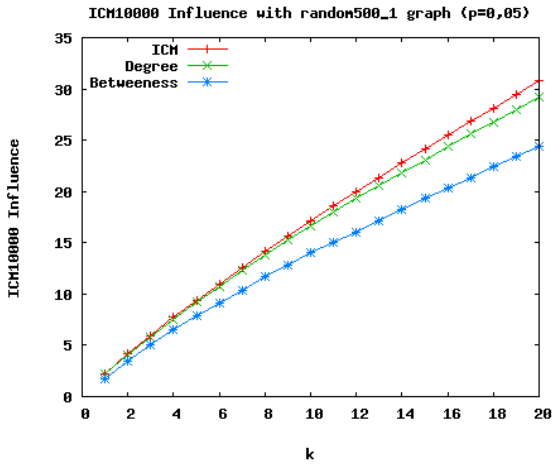
The main conclusion about this first wave of results is that ICM is interesting on scale-free graphs for high enough $p_{v,w}$ values, at least 0.05 it seems. Otherwise, propagation does not go far enough to turn it into a complex problem needing complex solutions, and taking high degree



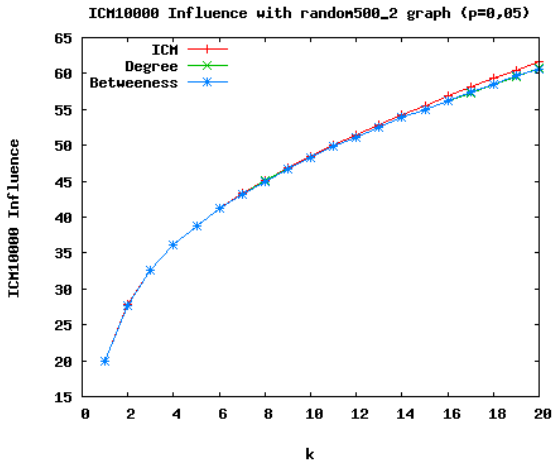
(a) Uniform $p_{v,w} = 0.01$



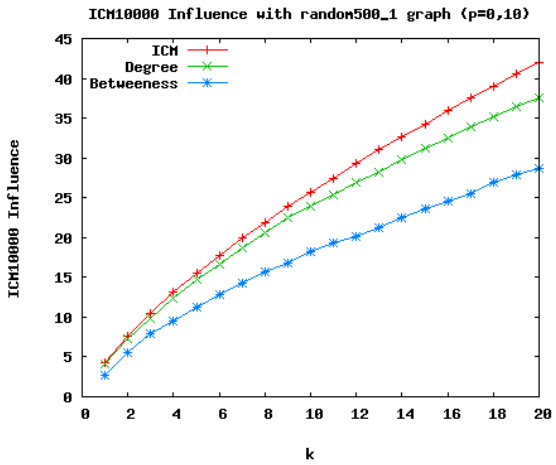
(b) Uniform $p_{v,w} = 0.01$



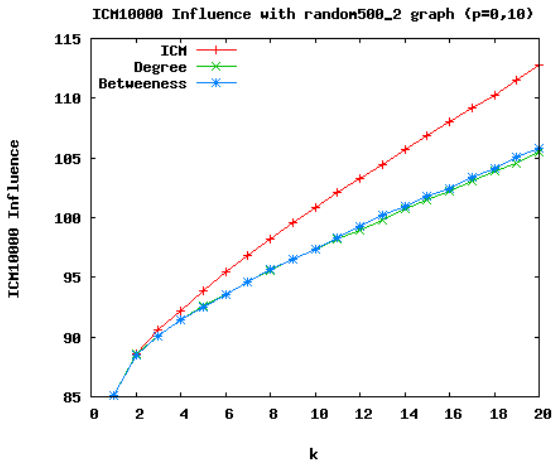
(c) Uniform $p_{v,w} = 0.05$



(d) Uniform $p_{v,w} = 0.05$



(e) Uniform $p_{v,w} = 0.10$



(f) Uniform $p_{v,w} = 0.10$

Figure 4.6: Evolution of the ICM influence for the k most influent vertices obtained with the greedy ICM_{10000}, the degree and the betweenness centrality for the `random500_1` set on the left and the `random500_2` set on the right.

vertices appears to be quite enough.

The previous graphs presented the first version of degree ranking c_{D_1}. Another ranking based on degree c_{D_2} is also suggested in the section 4.3.1 and it occurs to get better results on the first set as it catches back part of the difference between greedy ICM_{10000} and the degree ranking c_{D_1}. This result does not change much of the situation though as it stays behind ICM as the probability $p_{v,w}$ and k increase. On the second set, the c_{D_2} degree ranking does even worse than the basic one, better forget it.

	`random500_1`			`random500_2`		
p	0.01	0.05	0.10	0.01	0.05	0.10
c_D	21.849267	29.164122	37.588478	27.646000	60.720244	105.466844
c_{D_2}	21.866944	29.418978	38.409089	27.576500	60.252778	104.691556

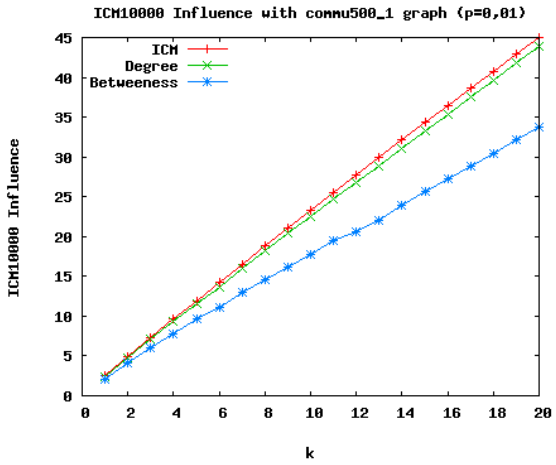
Table 4.2: Comparison of the two degree rankings for the `random500_1` and `random500_2` sets ($k = 20$)

4.3.4 Results on the `commu500` set

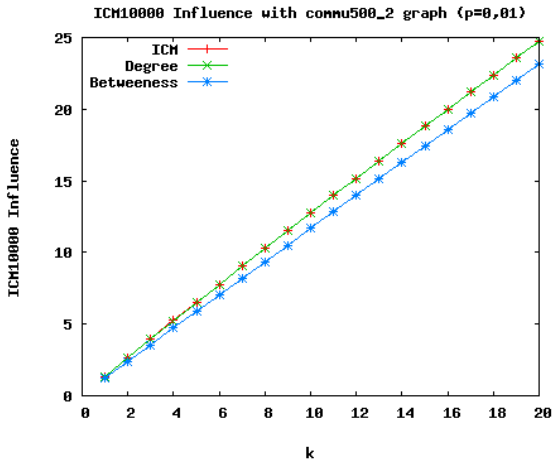
Figure 4.7 on page 41 shows the evolution of the ICM influence σ_{ICM} of k-sets found using the greedy ICM_{10000}, the degree and the betweenness, on modular graphs. The values are averages obtained on the whole set of graphs and have been computed with a uniform probability $p_{v,w}$ set to 0.01, 0.05 and 0.10.

Some observations made on the `random500` set tend to appear again. The fact that ICM does not give anything more than a degree ranking on low probabilities seems confirmed on community structured graphs too. The supposition concerning the propagation saturation making the process more complex, and giving ICM an advantage on degree ranking also tends to be confirmed by these results. The graphs from the set `commu500` are far denser than those form any other sets. This is the reason why choosing the best vertices is a much harder task than taking the high degree vertices. One vertex is enough to propagate through its whole community if $p_{v,w}$ is not too low. It is thus very important to choose next vertices into other communities, and a degree ranking cannot give such a behaviour, except with luck. Such graphs with very dense communities are a perfect example of a situation where finding the most influent k-set requires a good knowledge of the graph structure. A look at the average number of edges for the graphs from the second set (2697.33) shows that they are really far sparser than the graphs from the first set (14720.67). This is the reason why ICM makes less difference compared to degree ranking as the saturation effect inside a community is far less obvious. This saturation effect shall probably gradually reappear though as the probability increases.

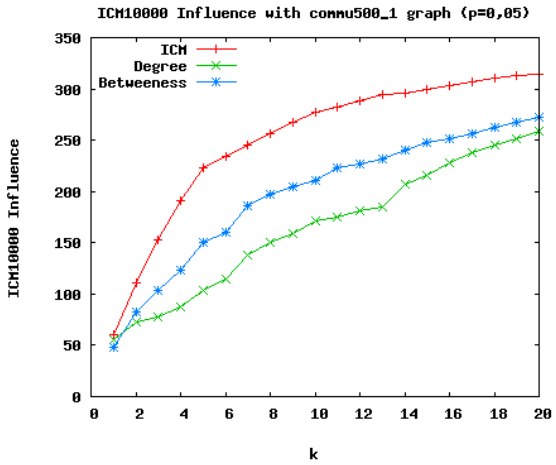
Figure 4.8 shows the evolution of the ICM influence while the probability $p_{v,w}$ is increased. As presumed, the degree ranking gives results that are less good compared to the greedy ICM_{10000} as the probability is growing. But as the probability becomes higher, another effect appear, indeed, with large probabilities, no matter which vertices are chosen, because of the high probability, it will contaminates a lot of other vertices and having a great influence.



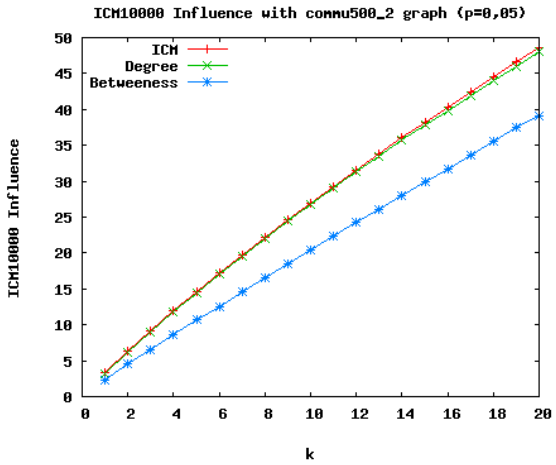
(a) Uniform $p_{v,w} = 0.01$



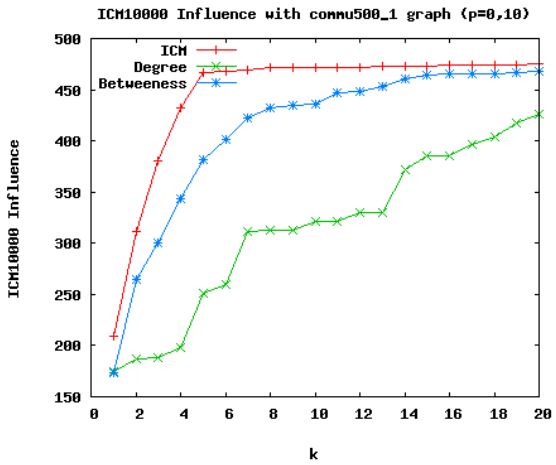
(b) Uniform $p_{v,w} = 0.01$



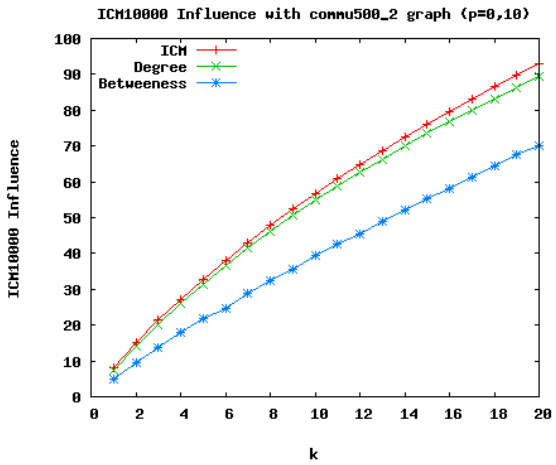
(c) Uniform $p_{v,w} = 0.05$



(d) Uniform $p_{v,w} = 0.05$



(e) Uniform $p_{v,w} = 0.10$



(f) Uniform $p_{v,w} = 0.10$

Figure 4.7: Evolution of the ICM influence for the k most influent vertices obtained with the greedy ICM_{10000}, the degree and the betweenness centrality for the `commu500_1` set on the left and the `commu500_2` set on the right.

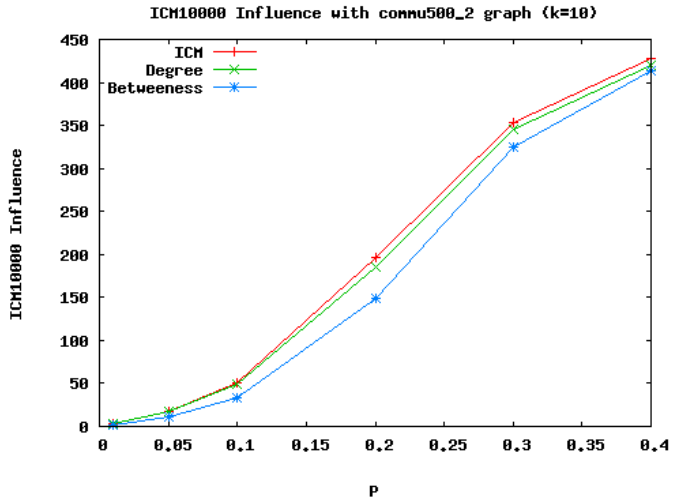


Figure 4.8: Evolution of the ICM influence for the $k = 10$ most influent vertices obtained with the greedy maximization problem with the ICM diffusion model with the probability $p_{v,w}$.

The second version of degree ranking c_{D_2} has also been tested on the community graphs sets. Its results on the first set are even worse than the original ranking, which already was quite bad itself. On the second set, the advanced degree ranking achieves quite the same results as the basic one, and is thus not of much interest.

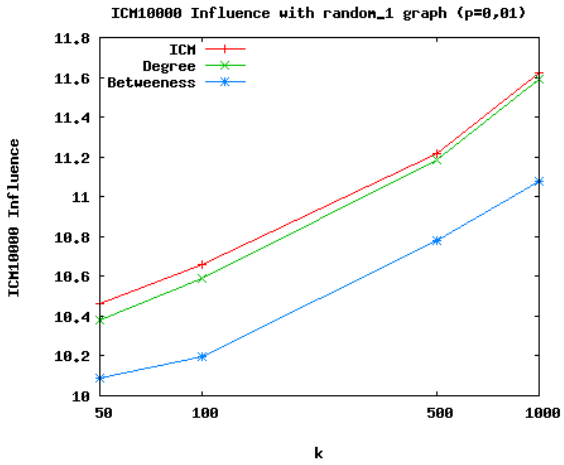
	commu500_1			commu500_2		
p	0.01	0.05	0.10	0.01	0.05	0.10
c_D	43.822767	258.449067	425.794233	24.767211	48.033978	89.520256
c_{D_2}	43.456622	222.156378	339.797744	24.750878	48.008711	89.903811

Table 4.3: Comparison of the two degree rankings for the commu500_1 and commu500_2 sets ($k = 20$)

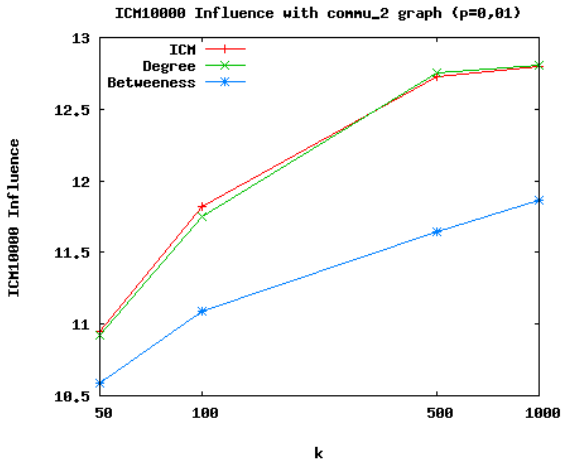
4.3.5 Varying the size of the graphs

The size of the graphs involved is a dimension that should not be forgotten. All tests in this chapter are performed on graphs made of 500 vertices, thus very small compared to real-size problems. Working only on graphs of the same size might induce characteristics that could appear general though being specific to this size of graphs. This subsection thus checks that degree and ICM seem to evolve in similar ways as the size of graphs changes and the results are presented in figure 4.9 on page 43.

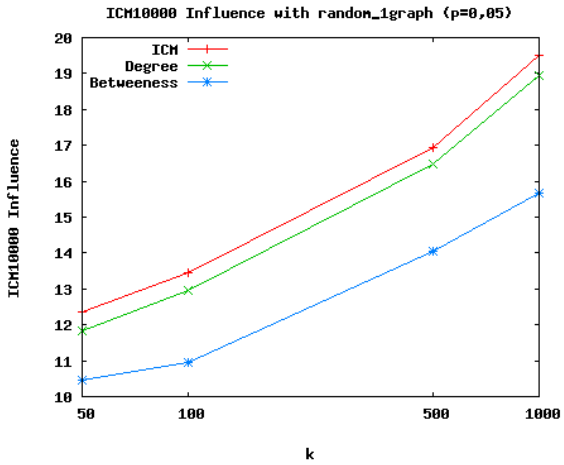
These tests seem to confirm everything is alright as ICM_{10000}, degree ranking and betweenness centrality increase in quite similar patterns for both kinds of graphs. Though further tests would be necessary to ensure there is no big concern with the size of test graphs, they are not conducted in this work.



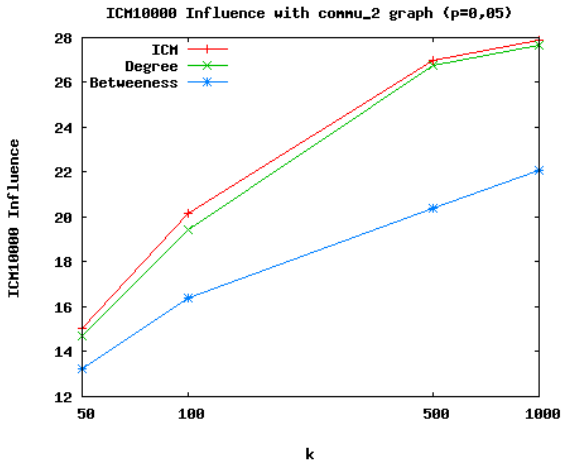
(a) Uniform $p_{v,w} = 0.01$



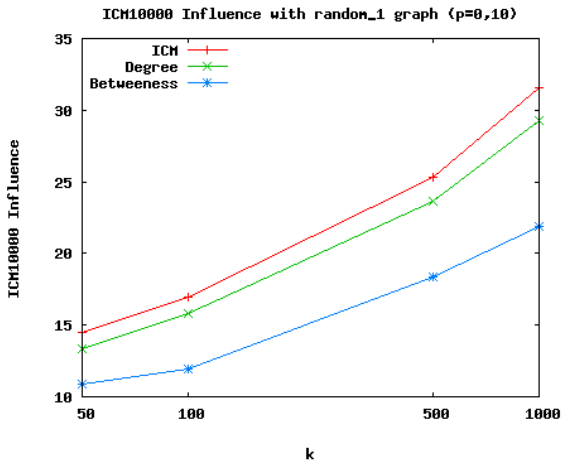
(b) Uniform $p_{v,w} = 0.01$



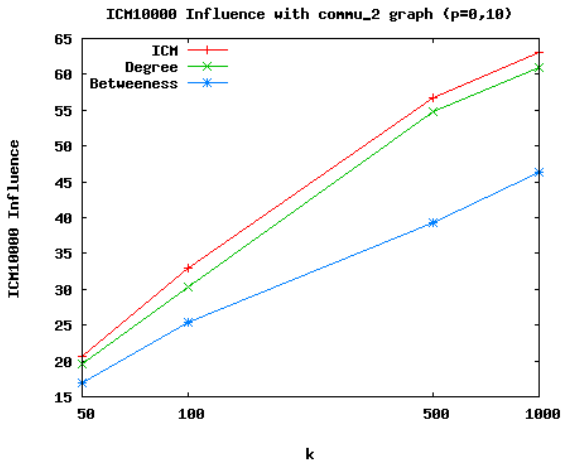
(c) Uniform $p_{v,w} = 0.05$



(d) Uniform $p_{v,w} = 0.05$



(e) Uniform $p_{v,w} = 0.10$



(f) Uniform $p_{v,w} = 0.10$

Figure 4.9: Evolution of the ICM influence with the size of the graph for the 10 most influent vertices obtained with the greedy ICM_{10000}, the degrees and the betweenness centrality for the `random500_1` set on the left and the `commu500_2` set on the right.

4.4 SPM vs ICM

The previous section shows several criteria allowing partly to predict when greedy ICM_{10000} shall prove useful : on dense graphs and on graphs with a specific structure tending to saturate parts of the graph without contaminating other parts. The goal of this section is to go on further with ICM_{10000} on these kind of graphs, and to test greedy SPM and greedy SP1M to see if they manage to approximate greedy ICM_{10000} well. SPM and SP1M are models of diffusion other than ICM, they are based on assumptions that are a simplification of the ICM model assumptions. They also lead to faster ways of selecting a k-set of high influence. The question is to know if those k-sets selected by SPM and SP1M with the greedy solver are as good as those selected by greedy ICM_{10000}, in the eyes of ICM. If this happens to be the case, using greedy SPM and SP1M would be a better solution than using greedy ICM_{10000} as the k-sets are found much faster.

4.4.1 Results on the `random500` set

Figure 4.10 on page 45 shows the comparison between the greedy ICM_{10000}, the greedy SPM and the greedy SP1M for the `random500_1` and `random500_2` sets. Greedy ICM appeared to gain advantage gradually compared to degree ranking on the first set of scale-free graphs, starting at $p = 0.05$. The difference was not gigantic though and SPM should really be very close to ICM to appear interesting.

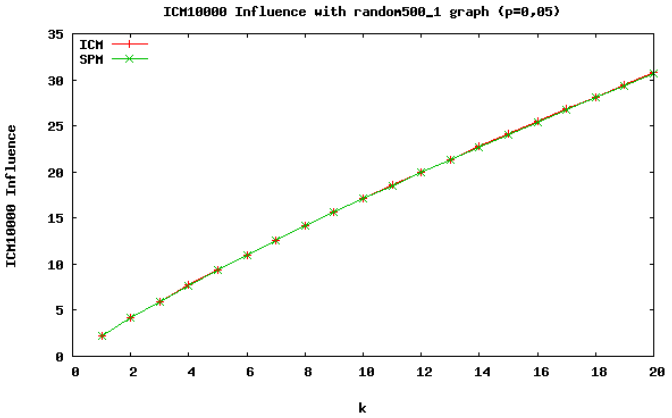
SPM had to be very close, and it is. It matches exactly the performances of greedy ICM_{10000} for $p = 0.05$ and is not far for $p = 0.10$ compared to degree ranking. Keeping a look at SPM, and derivatives, in this context thus seems quite interesting, even if it will probably not lead to anything fabulous.

The situation was quite different with the second set of scale-free graphs. The degree ranking did as well as greedy ICM_{10000} for $p = 0.05$, except near $k = 20$. On the other side, the degree ranking did pretty bad for $p = 0.10$ where greedy ICM_{10000} quickly took a fairly big advantage. SPM has a big opportunity to prove useful for $p = 0.10$.

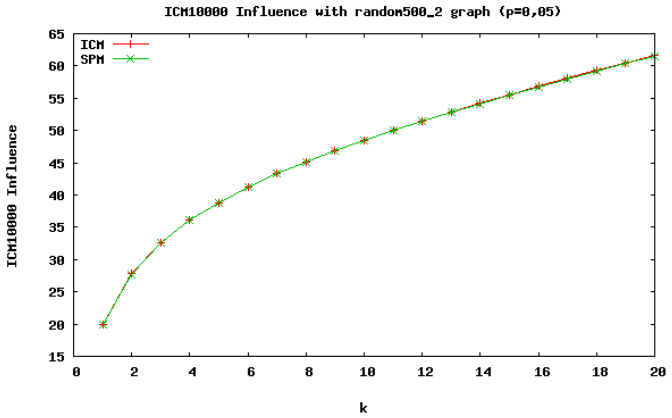
Unfortunately, greedy SPM does not achieve much more than degree ranking and is quite as miserable compared to greedy ICM_{10000}. SP1M on the other side achieves to catch back part of the difference. SP1M is not quite as good as ICM_{10000} yet but it is in the middle between degree ranking and greedy ICM_{10000}. The gap being quite big, greedy SP1M might be worth the calculations it requires while SPM is not.

4.4.2 Results on the `commu500` set

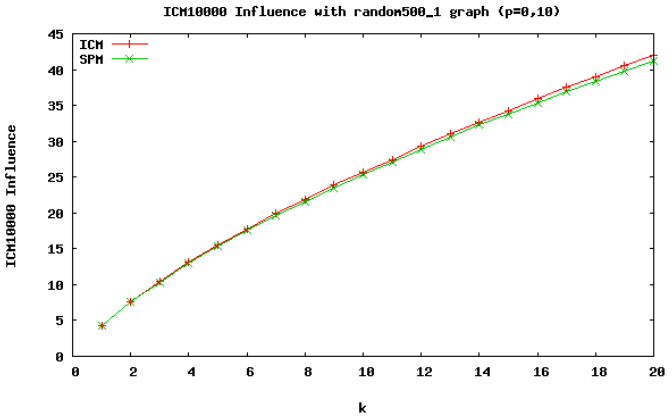
Figure 4.11 on page 47 shows the comparison between the greedy ICM_{10000}, the greedy SPM and the greedy SP1M for the `commu500_1` and `commu500_2` sets. The first set which was very



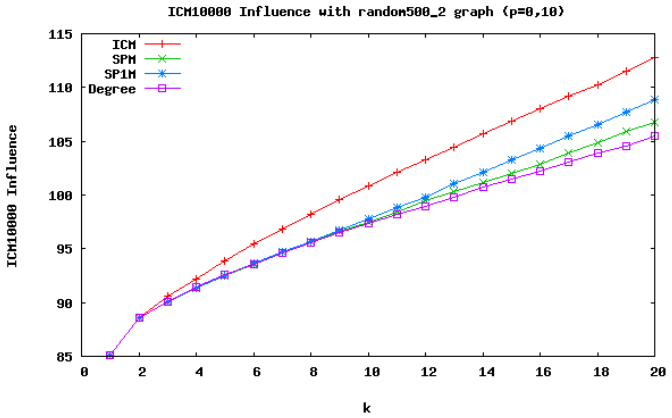
(a) `random500_1` set, uniform $p_{v,w} = 0.05$



(b) `random500_2` set, uniform $p_{v,w} = 0.05$



(c) `random500_1` set, uniform $p_{v,w} = 0.10$



(d) `random500_2` set, uniform $p_{v,w} = 0.10$

Figure 4.10: Evolution of the ICM influence for the k most influent vertices obtained with the greedy maximization problem with ICM_{10000} compared to SPM for the `random500_1` and `random500_2` sets

dense communities gave an enormous advantage to greedy ICM_{10000} on degree or betweenness rankings, even with $p = 0.05$. This context probably is the one where SPM is expected to be the most useful since the gap between greedy ICM_{10000} and degree ranking and betweenness centrality.

And this definitely is the context where SPM totally outperforms degree and betweenness rankings. SPM follows the same behavior as ICM, being just a little bit behind. This context of very dense community graphs appears thus to be very promising for the development of the techniques based on communities proposed in the previous chapter, community SPM for example.

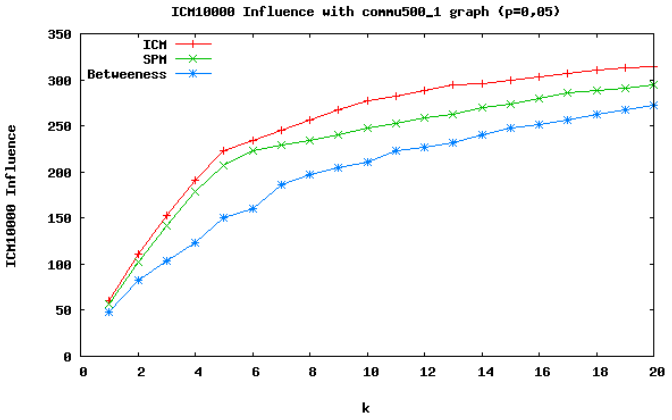
A quite strange behavior appears here though as the main difference between ICM and SPM seems to come from the choice of the first vertex. This is quite unusual and no explanation are given in this work.

The second community set is much sparser and greedy ICM_{10000} did not manage to drop the degree ranking much. There is a small opportunity for greedy SPM with $p = 0.10$ as greedy ICM_{10000} gradually improves with the increase of probabilities, but the gap is still quite small even at $p = 0.10$.

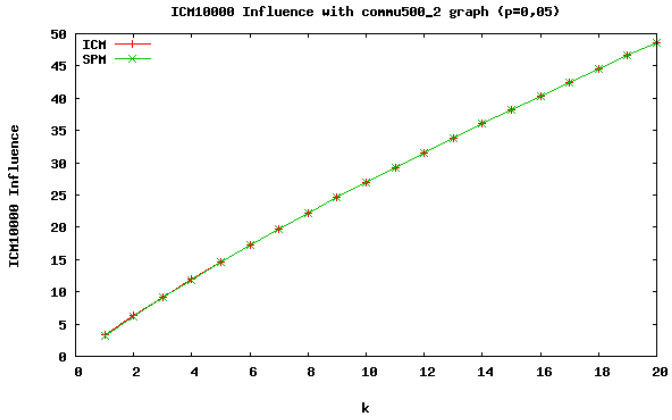
The gap is quite small, and greedy SPM fills it. The same conclusions can be reached as for first set of scale-free graphs. Though it does not seem very promising, greedy SPM still does a bit better than degree ranking and these graphs thus deserve to be further tested.

4.4.3 Varying the size of the graphs

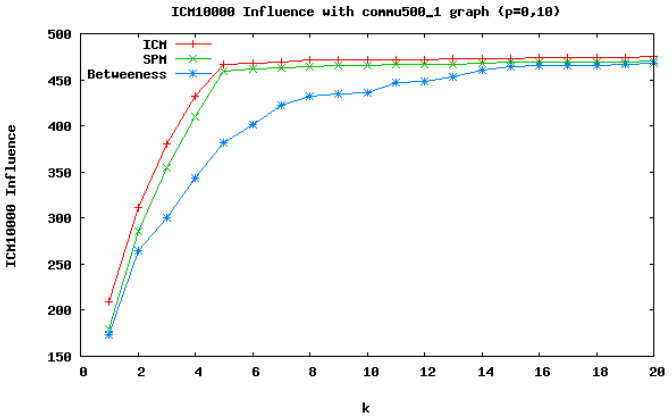
Once again, the size of the graphs involved should not be forgotten. The tests done in order to compare ICM with SPM have been performed on graphs with different sizes and the results are presented on figure 4.12 (page 48). Once again, these tests seem to confirm the observations done previously, though further tests on the other sets would be necessary to increase the confidence in the conclusions done, but they are not conducted in this work.



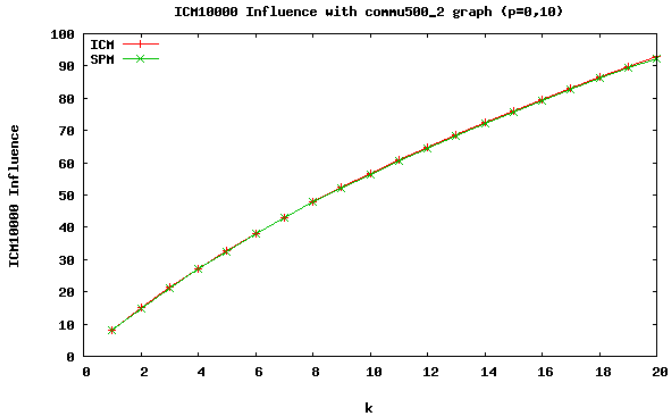
(a) `commu500_1` set, uniform $p_{v,w} = 0.05$



(b) `commu500_2` set, uniform $p_{v,w} = 0.05$

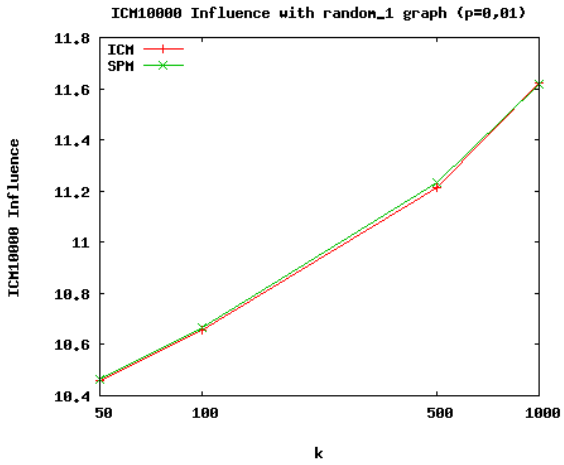


(c) `commu500_1` set, uniform $p_{v,w} = 0.10$

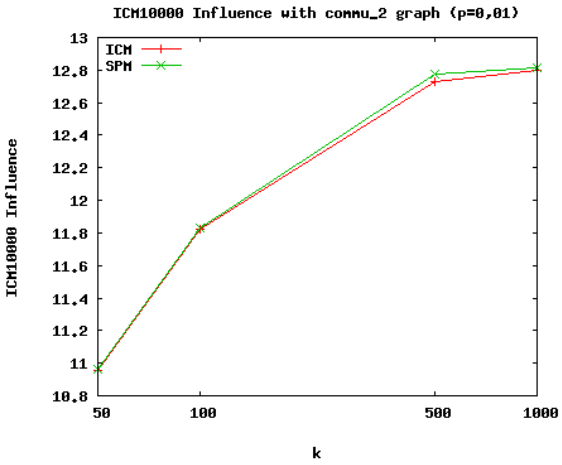


(d) `commu500_2` set, uniform $p_{v,w} = 0.10$

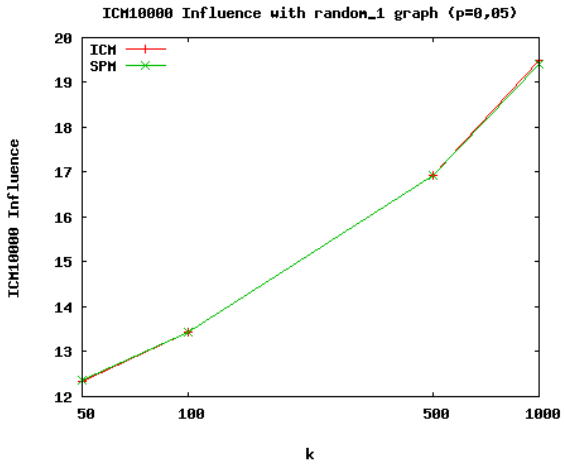
Figure 4.11: Evolution of the ICM influence for the k most influent vertices obtained with the greedy maximization problem with ICM_{10000} compared to SPM for the `commu500_1` and `commu500_2` sets



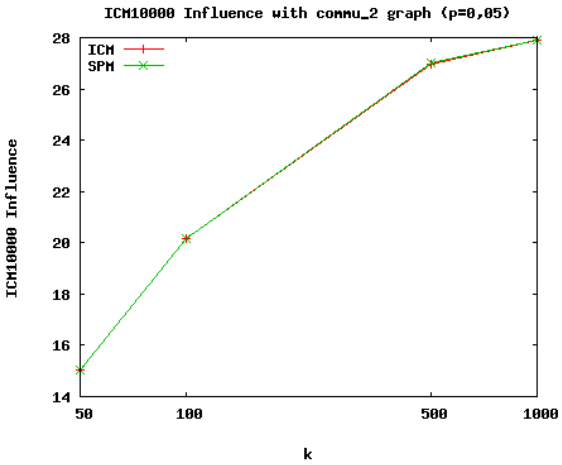
(a) Uniform $p_{v,w} = 0.01$



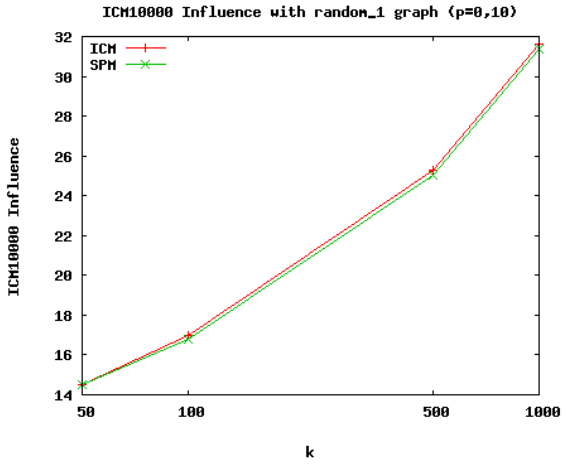
(b) Uniform $p_{v,w} = 0.01$



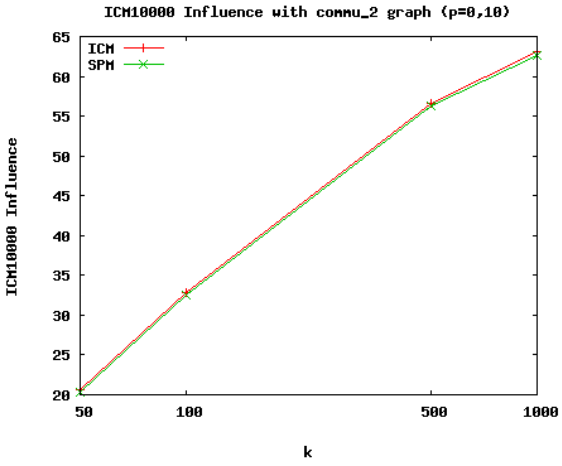
(c) Uniform $p_{v,w} = 0.05$



(d) Uniform $p_{v,w} = 0.05$



(e) Uniform $p_{v,w} = 0.10$



(f) Uniform $p_{v,w} = 0.10$

Figure 4.12: Evolution of the ICM influence with the size of the graph for the 10 most influent vertices obtained with the greedy ICM_{10000}, the degrees and the betweenness centrality for the `random500_1` set on the left and the `commu500_2` set on the right.

4.5 Community SPM

The tests performed with greedy SPM and presented in the previous section shows that greedy SPM is able to approximate greedy ICM_{10000} well enough to be worth the difference of calculations compared to degree betweenness. Next step consists in testing the suggested community SPM in those specific situations. Community SPM is another solver using the SPM model and is expected to run much faster than greedy SPM. The question that shall be answered here is at which cost. There is of course no point in finding a k-set faster than greedy SPM if the set quality is too much compromised.

4.5.1 Results on the `random500` set

The first set remains the first version of scale-free graphs. The degree ranking was not that bad but greedy SPM obtained results comparable to those of greedy ICM_{10000}. It was thus decided to keep on testing this first set for $p = 0.10$. Once again, the suggested versions must remain quite close to greedy SPM and greedy ICM_{10000} if they are to beat the degree ranking.

	Influence	Computation time
Greedy ICM	42.032733	$12710.42\,s$ (≈ 3.5 hours)
Degree ranking	37.588478	$0.05\,s$
Greedy SPM	41.236933	$5.66\,s$
Community SPM	39.680578	$0.93\,s$
Advanced SPM	40.064333	$1.03\,s$

Table 4.4: ICM Influence and computation time for the `random500_1` set for several solvers of the influence maximization problem ($k = 20$ and $p = 0.10$)

The loss in influence by using community SPM is quite important. The influence obtained with the community SPM solver is closer to degree ranking than to greedy ICM. The advanced version catches back only a small part of the loss and does not really change much to the situation. Both community-based algorithms achieve better results than Degree ranking, but is the improvement worth the calculations ? Both are around 20 times slower than the degree ranking and 5 times faster than greedy SPM. The precision gaps are quite important though and, except if computation time really is a big issue, it looks like greedy SPM is the technique offering the best ratio time/precision in this situation.

The second set is the second version of scale-free graphs. This context is even tougher for greedy SPM. Only $p = 0.10$ presented an opportunity for greedy SPM but it was not able to catch it, only SP1M managed to catch back part of the difference and avoid the disaster. It was therefore wondered if a community SP1M could maybe do anything good in here.

This results table offers some surprises. Community SPM achieves better results than greedy

	Influence	Computation time
Greedy ICM	112.782356	58898.85 s ($\approx$ 16.36 hours)
Degree ranking	105.466844	0.05 s
Greedy SPM	106.753233	9.71 s
Greedy SP1M	108.873689	18.1 s
Community SPM	107.131633	0.302 s
Community SP1M	107.572767	0.421889 s

Table 4.5: ICM Influence and computation time for the `random500_2` set for several solvers of the influence maximization problem ($k = 20$ and $p = 0.10$)

SPM while community SP1M does worse than SP1M. Both community-based solutions thus end with comparable results and computing time. Based on this table, the best solution here, if there is no time for greedy ICM_{10000}, seems to be greedy SP1M. This result seems worth some more tests to confirm it though. Those tests are not conducted in this work.

4.5.2 Results on the `commu500` set

The best context for the community-based algorithms are of course the modular graphs. The greedy SPM solver has achieved to do nearly as well as greedy ICM_{10000} on dense community graphs while the degree ranking was sinking. If community SPM and advanced SPM manage to stay close to greedy SPM, this will definitely be a great result.

	Influence	Computation time
Greedy ICM	474.735980	651352.41 s ($\approx$ 7.5 days)
Degree ranking	425.794233	0.05 s
Greedy SPM	470.061356	57.05 s
Community SPM	470.562044	2.57 s
Advanced SPM	469.505467	3.38 s
Iterative SPM	469.592156	3.5 s

Table 4.6: ICM Influence and computation time for the `commu500_1` set for several solvers of the influence maximization problem ($k = 20$ and $p = 0.10$)

And community SPM does it. Its k-set gives as good an influence as greedy SPM. This means that the speedup expected thanks to the community-based SPM comes here, on dense community graphs, at nearly no cost of influence at all. As for the speedup, it brings SPM from 57.05 s down to 2.57 s by using communities, reducing the computing time by a factor of 20. This is thus an excellent result for the community-based solver which achieves an important speedup through nearly no accuracy sacrifice. More tests deserve to be carried to confirm this result on dense community graphs, but it definitely seems promising.

The last set is the second version of community graphs. On those not-so-dense community

graphs, the situation is quite the same as for first version of scale-free graphs. Community SPM has no great margin and must stay very close to greedy SPM to remain an option.

	Influence	Computation time
Greedy ICM	93.047978	25616.79 s ($\approx$ 7.12 hours)
Degree ranking	89.520256	0.05 s
Greedy SPM	92.412433	11.17 s
Community SPM	92.194567	0.64 s
Advanced SPM	92.447378	0.84 s

Table 4.7: ICM Influence and computation time for the `commu500_2` set for several solvers of the influence maximization problem ($k = 20$ and $p = 0.10$)

Unlike scale-free graphs, community SPM manages to stay very close to greedy SPM, more like first set of community graphs. Community SPM therefore manages to keep some good influence advantage on degree ranking. As for the computation time, community SPM, though being 10 times slower than degree ranking, remains around 20 times faster than greedy SPM with nearly no accuracy loss, especially with its advanced version. Community SPM and Advanced SPM both appear to be good options in this context too. The overall quality loss compared to greedy ICM_{10000} seems negligible compared to the speedup acquired.

4.6 Two real examples

All tests so far have been done on randomly generated graphs. The same tests have been performed in this section on two "real-world" graphs, based on data collected in the real world [BM06].

The first graph, called `roget`, is built from a thesaurus of english words. Each vertex of the graph corresponds to one of the 1022 categories in the 1879 edition of Peter Mark Roget's Thesaurus of English Words and Phrases, edited by John Lewis Roget. An arc goes from one category to another if Roget gave a reference to the latter among the words and phrases of the former, or if the two categories were directly related to each other by their positions in Roget's book.

The second graph, called `EVA` [NLGC02], is built based on corporate ownership informations. The vertices of the graph represent companies and a directed edge links the vertices u and v if the company u is an owner of the company v.

Table 4.8 gives some statistics about both graphs and about their community structure. The graphs have previously been cleaned in the sense that only the biggest connected component is considered.

Examining the degree distribution (figure 4.13 on page 53) of both graphs shows there are a lot of vertices with a small degree and some vertices with a high degree. The highest degree vertex has a degree of 39 in `roget` graph and of 552 for `EVA` graph. These graphs can thus be approximately considered as scale-free graphs. The `EVA` graph, unlike the randomly generated

	Vertices	Edges	Density	Communities	Modularity	Inter Edge	Intra Edge
`roget`	946	4949	0.005530	105	0.463666	44.5140%	55.4860%
`EVA`	4475	4664	0.000233	124	0.915906	4.3096%	95.6904%

Table 4.8: Some statistics for the two graphs `roget` and `EVA` : number of vertices n, number of edges m, density (m/n^2), number of communities, modularity and percentage of inter-community and intra-community edges. The number of communities is found using the algorithm based on random walks (see section 3.2) with $t = 4$.

scale-free graphs, has a very high community structure though as testified by its modularity ($Q^M = 0.915906$). The `EVA` graph is thus in some sense a new type of graph that has not be tested on yet.

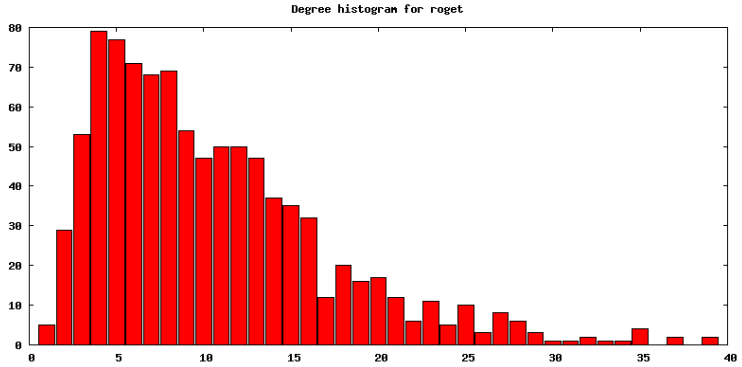
4.6.1 The `roget` graph

The `roget` graph is a scale-free graph. According to the observations done on the randomly generated graphs, greedy ICM_{10000} is only interesting when probability $p_{v,w}$ increases. This behavior is indeed observed on the `roget` graph as shown in the figure 4.14 on page 54.

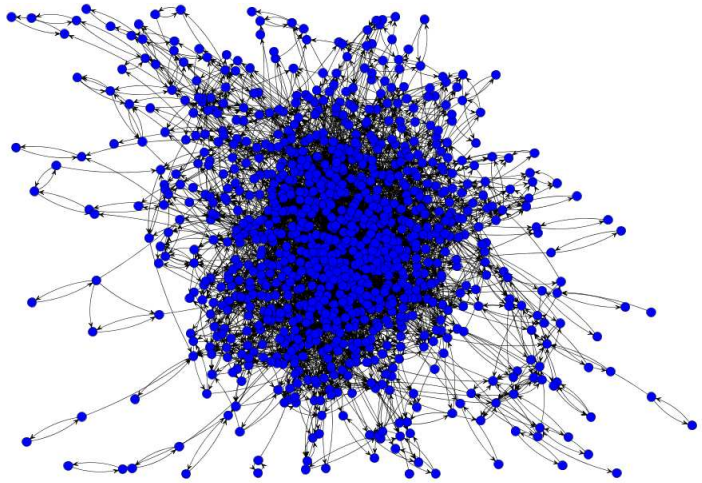
According to the density of the `roget` graph, the graph is like those from the `random500_1` set. Using greedy SPM should thus provide a good approximation of greedy ICM_{10000} for small values of k and gradually lose some accuracy as k increases. This is confirmed by figure 4.15 on page 54. SP1M can recover part of the difference between ICM_{10000} and SPM though as shown by the results listed in table C.6 on appendix C, and as expected by the results obtained on randomly generated graphs.

The last step of the experiments is to test the community-based solver. Results are shown in figure 4.16. If the `roget` graph keeps on behaving like the generated graphs from set `random500_1`, community SPM should be a bit behind SPM while advanced SPM should catch back half of the difference. The `roget` graph does not behave at all as the `random500_1` set though as can be seen, and it partly is a good news. Community SPM indeed yields very disappointing results, probably because of the bad proportion between intra(55%) and inter(45%)-community edges. This allows, for the first time through the testing process, advanced SPM to express itself as it catches back nearly all that was lost by community SPM. This big difference between the `random500_1` set and the `roget` graph probably comes from the different community structure. While the generated graphs had a very bad community structure with most high degree vertices ending up in 2 or 3 big communities containing together nearly half of vertices, `roget` has a better community structure with more significative communities and a better repartition of important vertices between these communities.

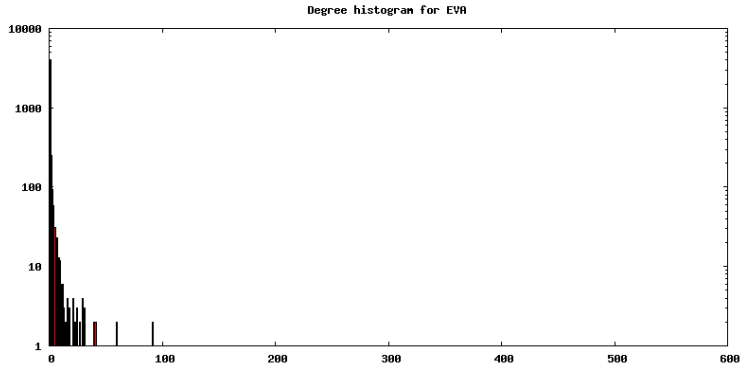
This reasoning might sound illogical as the community SPM is said to perform better on a graph with a lower community structure. It is not quite that though. The community SPM performs pretty well on the generated graphs in spite of its high ratio of lost edges, because the lost edges are those from the other half of vertices who lie in communities of less than 10 vertices and who



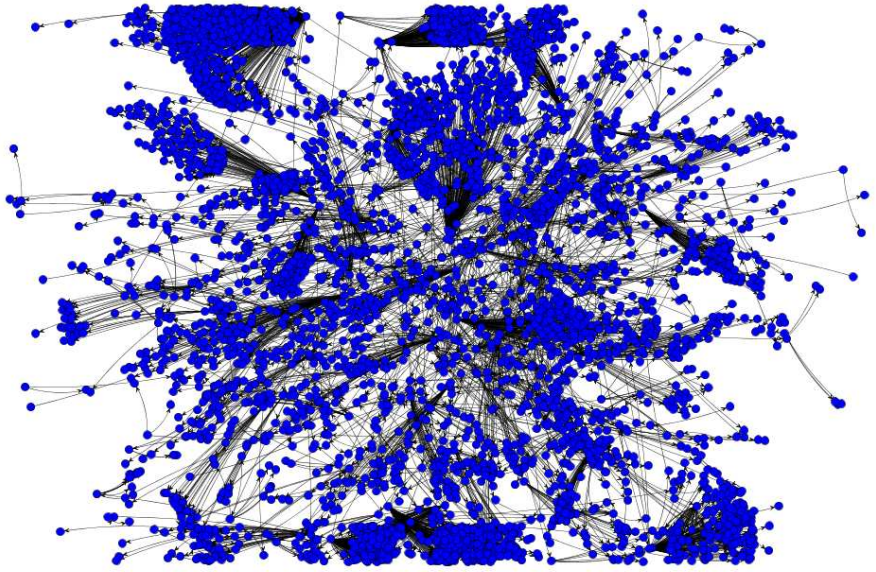
(a) Degree distribution of `roget`



(b) The `roget` graph

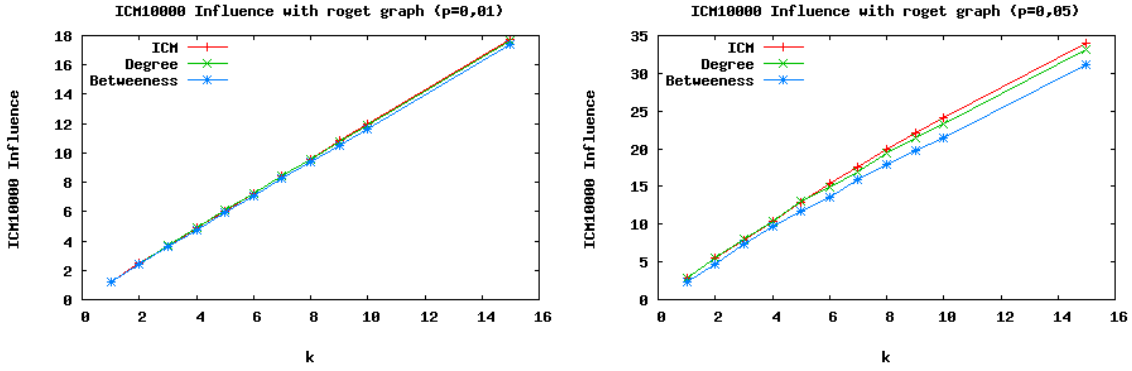


(c) Degree distribution of `EVA`

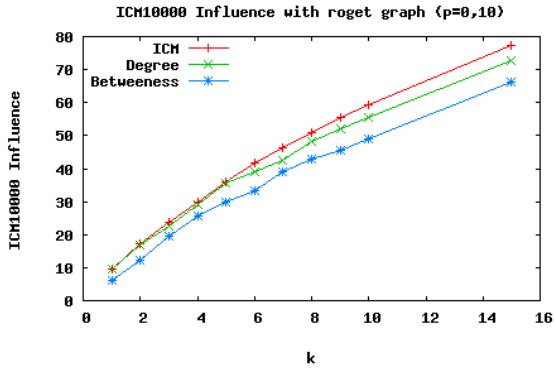


(d) The `EVA` graph

Figure 4.13: The two "real" graphs `roget` and `EVA`



(a) Uniform $p_{v,w} = 0.01$ (b) Uniform $p_{v,w} = 0.05$



(c) Uniform $p_{v,w} = 0.10$

Figure 4.14: Evolution of the ICM influence for the 15 most influent vertices obtained with the greedy maximization problem with ICM10000, degree ranking and betweenness for the `roget` graph as propagation probability of edges increases.

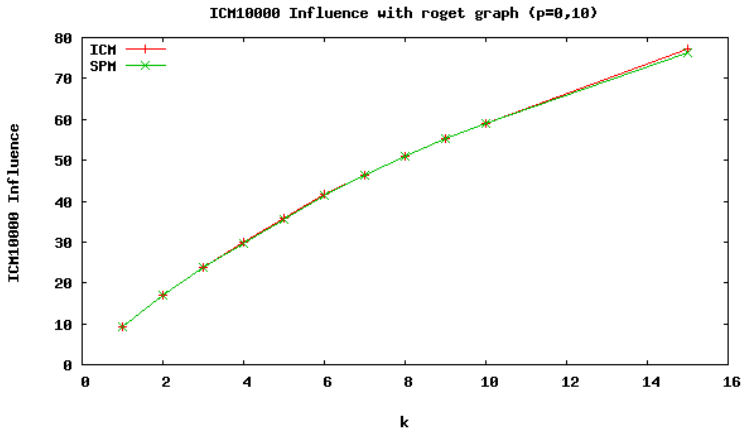
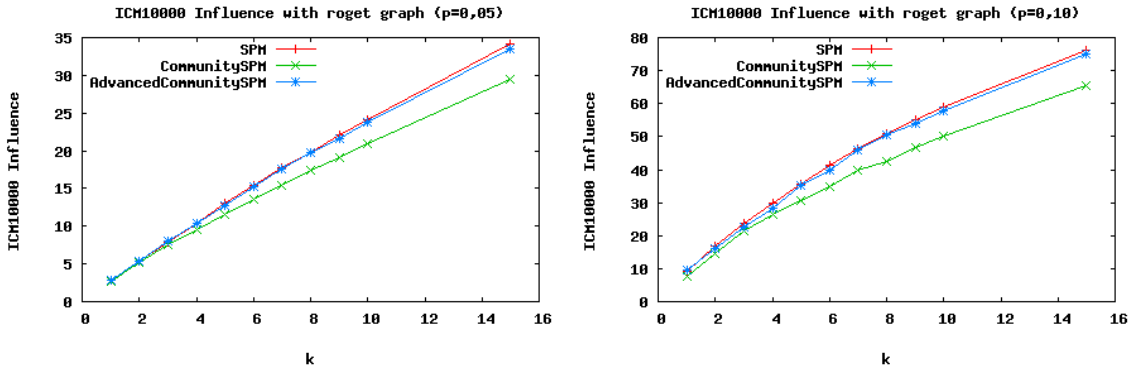


Figure 4.15: Evolution of the ICM influence for the k most influent vertices obtained with the greedy maximization problem with ICM_{10000} compared to SPM for the `roget` graph

	ICM influence	Time
Greedy ICM	77.165000	49312 $s \approx$ 13.7 hours
Degree ranking	72.723700	0.05 s
Betweenness	66.208200	7.55 s
Greedy SPM	76.310800	33.735 s
Community SPM	65.409300	0.594 s
Advanced SPM	74.954900	1.188 s

Table 4.9: ICM Influence and computation time for the `roget` graph for several solvers with $p_{v,w} = 0.01$ and $k = 15$



(a) Uniform $p_{v,w} = 0.05$ (b) Uniform $p_{v,w} = 0.10$

Figure 4.16: Evolution of the ICM influence for the k most influent vertices obtained with the greedy SPM compared to community SPM and advanced SPM for the `roget` graph

are of little interest in looking for the influent vertices. The data from edges in the vicinity of influent vertices are saved as all important vertices are kept within 2 or 3 communities. The relative good behavior of community SPM on the generated graphs thus is a direct consequence of the way community detection algorithm detects the communities, actually only removing all low degree vertices and leaving all high degree vertices in very few communities. While the number of edges lost by communitirising is equivalent in both generated graphs and `roget` graph, the importance of the data contained in those removed edges is far more valuable on `roget` than for the generated ones. This is the reason why the advanced algorithm, getting back part of this information, is far more useful on the `roget` as it recovers valuable information.

4.6.2 The `EVA` graph

The `EVA` graph is a graph with a very high community structure (modularity $Q^M = 0.915906$) and with a very specific structure, though common in real world problems. It does not correspond at all to either the generated scale-free graphs nor the generated community graphs. Its community structure comes from the existence of "fans" of vertices. A fan is a root vertex with many edges going to leaf vertices of degree 1. Such a fan provides a high community structure, though it has

nothing to do with the kind of communities considered in the randomly generated community graphs. A specific graph is expected to give specific results, and they definitely are (table 4.10).

	0.01	0.05	0.10	Time for $p = 0.10$
Greedy ICM	26.592300	-	-	> 4 days
Community ICM	26.507300	93.356500	178.469500	≈ 12 hours
Degree ranking	26.483000	93.180800	178.382700	$0.36\,s$
Betweenness	16.336800	42.701800	78.455600	$20.31\,s$
Greedy SPM	26.5042	93.3669	179.206500	$237.84\,s$ (3.96 min)
Community SPM	26.450700	93.456400	179.206500	$1.33\,s$
Advanced SPM	26.481200	93.198900	179.353500	$1.594\,s$

Table 4.10: ICM Influence and computation time for the `EVA` graph for several solvers while the probability $p_{v,w}$ is set to 0.01, 0.05 and 0.10 ($k = 10$)

All methods give the same results because all methods return the same vertices. Actually, looking at the graph structure, it indeed is not too hard to guess which vertices are the most influent, the roots of fans. The complex techniques developed in the literature seem thus of no interest as a degree ranking is enough to detect which vertices are the fan roots. Considering the computation times, though degree ranking obviously is the best, on can see that community-based solvers perform extremely well as they are nearly 200 times faster than greedy SPM, which is really impressive.

Abstraction could be made of the fans, by putting for example some interest value on their roots and simply removing fans. There would then only remain the backbone of the graph. Such a backbone could be of much interest for experiments as it probably has very specific properties from real world graphs. Such preprocessing of the `EVA` graph is not done in this work though, and thus no analysis of the backbone is given.

4.7 Learnings

The main learnings about this testing process consists in deciding which pairs $\langle solver, model \rangle$ seem best under which circumstances. The first interesting observation is that, whatever graph is involved, a low probability of propagation makes all models and other calculations useless compared to a simple degree ranking.

Another important conclusion is that community-based algorithms tend to generate very small precision losses in community graphs compared to scale-free graphs. This does not mean that they are not worth the effort in scale-free graphs, but they will probably require more algorithmic improvements before they play a role. This leads to the other learning that community SPM seems the best solution for community graphs so far, especially for very dense ones, while greedy SPM, community SPM and degree ranking seem to be quite comparable for scale-free graphs, depending on $p_{v,w}$, and on which ratio time/quality is desired.

One aspect of the community-based solvers is that they require graphs to be split into subgraphs

corresponding to communities in order to achieve a speed-up. It should not be forgotten though that this splitting is time consuming. Detecting communities inside a graph is no easy trick and it is no mystery that many different algorithms exist. There is a difference though between the goals of the community detection field and what is needed to run the community-based solver. No tests have really been done but the solvers are expected to be quite robust for non-perfect community splitting. The data loss shall be a bit more important if communities are not perfectly split, but part of it should be recovered thanks to the advanced solver. Determining how good a community splitting is required and how time consuming it would be is thus a pretty tough question that remains.

5

Influence-based Community Detection

The problem considered so far has been to find the most influential k-set of vertices in a network. Another interesting problem, which has been introduced in chapter 3, is to detect communities in the network. There is a big difference though between community detection as it has been introduced in chapter 3 and what is done in this chapter. The communities considered in chapter 3 are detected on the underlying undirected graph, and so, do not take into account either the direction of the edges nor their associated propagation probabilities $p_{v,w}$. The algorithms presented in this chapter all take into account the direction of edges, and some of them also take into account a weight on the edges.

This chapter first introduces what a community detection algorithm could gives when taking into account new information from the graph such as edge direction. It then presents two existing algorithms. The first one is an extension of another algorithm based on the edge betweenness concept while the other use the notion of random walks. The second part of the chapter suggests a new community detection algorithm based on the concepts of influence and propagation of information.

5.1 Directed Community Detection

Figure 5.1 shows a directed graph with 17 vertices and 46 edges. As mentioned in chapter 3, most of the existing community detection algorithms are suited for undirected graphs and do not take into account either the direction nor the weight of edges. Figure 5.1 also shows the three communities extracted from the corresponding undirected graph with the random walk algorithm [LP05] presented in section 3.2, using the modularity (section 3.1.1) as a stop criterion for the clustering.

The communities found by the algorithm based on random walks are structural communities, meaning they do not take into account the orientation of the edges.

The communities that are to be detected in this chapter are *influence communities*. An *influence community* can be defined as a group of vertices such that the information propagates between two vertices of the community with a high probability.

An example of the consequences of using the corresponding undirected graph to look for communities lies in figure 5.3. Considering vertices 0 and 3, information can propagate from vertex

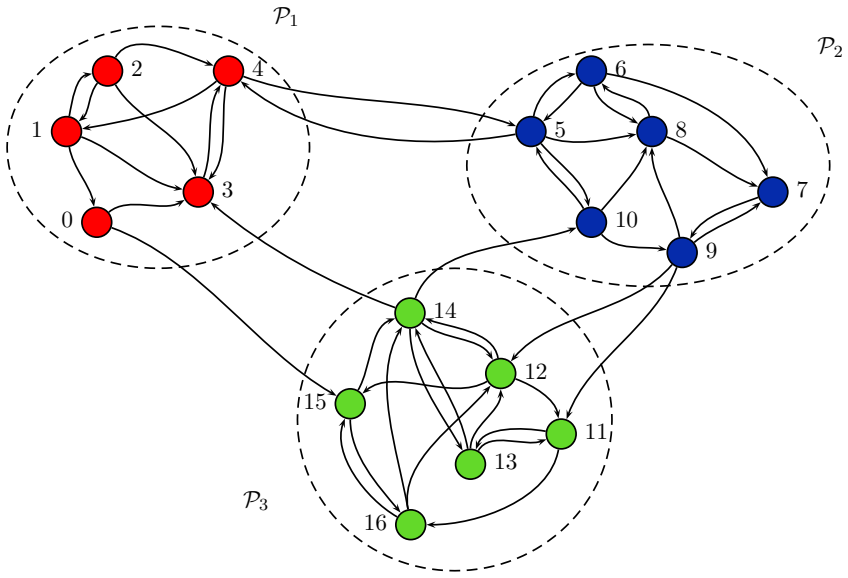


Figure 5.1: Example of a directed graph with uniform probabilities $p = 0.01$ with the three communities detected on the underlying undirected graph using the algorithm based on random walks with $t = 4$.

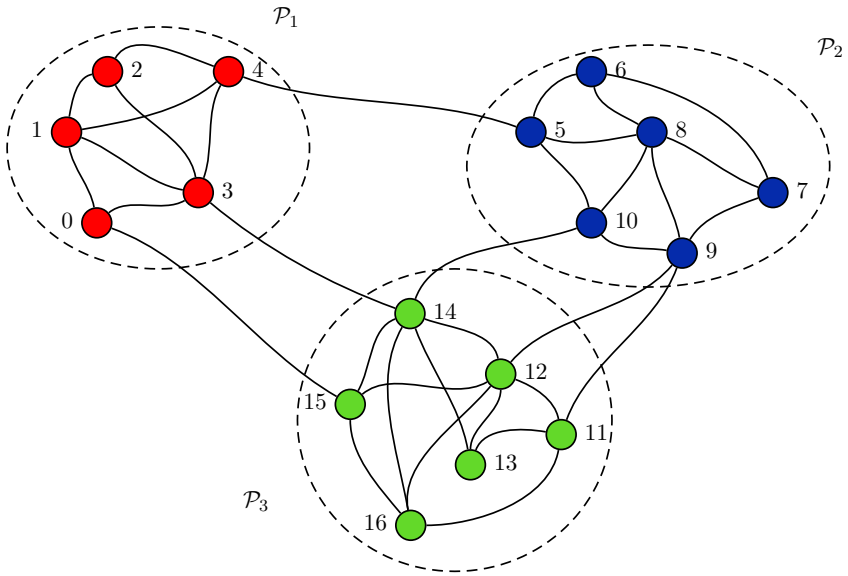


Figure 5.2: The underlying undirected graph of the one shown in figure 5.1

0 to vertex 3 with a probability $p_{0,3} \approx 0.01$ but the probability for vertex 3 to propagate toward vertex 0 is only $p_{3,0} \approx 10^{-6}$. So, those two vertices do not seem so close to each other, though they should probably end up in the same communities if the clustering goes far enough. Actually, vertex 0 looks more like an independent vertex that should stay alone a long time before being clustered with some other vertices. On the other side, looking at the corresponding undirected graph, vertex 0 seems to be fairly connected with the red community for a simple reason : both single directed edges from 1 and toward 3 are now considered as important as

bi-directed edges within the rest of the community. The direct consequence is that 0 and 3 are immediately clustered by the random walk algorithm (figure 5.3) though they should not when looking at the directed graph.

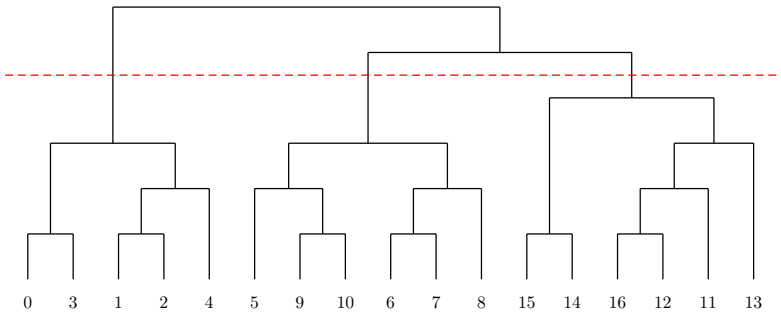


Figure 5.3: Dendrogram of the clustering used in order to detect the communities in the network of figure 5.1 using the algorithm based on random walks. The dashed line represents the partition with greatest modularity.

This example is based on a random walk based community detection algorithm. The conclusion is valid for any other community detection algorithm not taking into account edges directions though. They will not give good influence communities because the graph they think of is too far from the real graph they should be working on. Not having a good knowledge of the graph, they cannot give a good dendrogram. And with no good dendrogram they cannot give good communities, whatever stop criterion is used.

Though the problem seems clear, only a small number of papers considering the case of directed graphs have been found in the community detection literature. Two methods found in these papers follow.

5.1.1 Edge-betweenness Based Algorithm

The first of those methods, presented in [MPM06], tries to extend an existing community detection algorithm in order to take into account the direction of edges, which is a key feature in many situations. The algorithm extended here is the edge betweenness algorithm of Girvan and Newman [GN02]. A very close concept, the betweenness, has been presented earlier in section 4.3.2 and the edge-betweenness us actually an extension of the betweenness concept.

Let's summarize how the original algorithm of Girvan and Newman works. The idea is to start from the whole graph as a single cluster and to remove the inter-community edges in order to gradually isolate communities from the rest of the graph. The edge-betweenness of an edge is the number of (directed) shortest paths joining two vertices of the graph containing this edge. The observation done in [GN02] is that the inter-community edges have a great edge-betweenness since the shortest paths between vertices from different communities will all go through these

few inter-community edges.

$$c_B(e) = \sum_{s \in V} \sum_{t \in V} \delta_{st}(e) \qquad \text{with } \delta_{st}(e) = \frac{\sigma_{st}(e)}{\sigma_{st}}$$

First step of the algorithm is to compute the edge-betweenness of all edges of the graph. The edge with greatest value is then removed, edge betweenness are computed again, and so on until no edge is left. The order in which edges are removed is memorized and communities are then detected using a hierarchical clustering based on reading edges in the reverse order and clustering when the added edge joins two vertices from different clusters. The algorithm eventually outputs the partition with highest modularity.

The extension proposed in [MPM06] occurs when adding back an edge (u, v) with $c_u \neq c_v$. When such a case occurs, the algorithm performs a depth-first search from the vertex v and checks if there is a path from vertex v to vertex u. If there exists no such path, the clusters are not merged together. The obtained clusters are thus always strongly connected components of the graph.

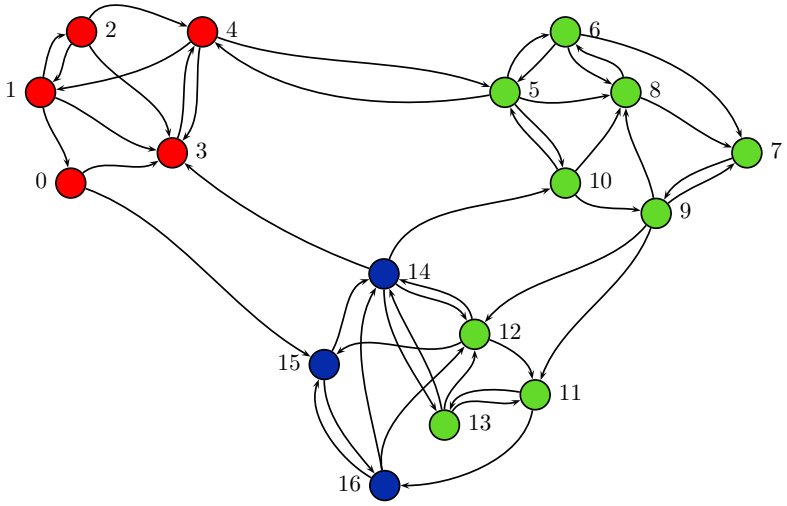


Figure 5.4: Example of a directed graph with uniform probabilities $p = 0.01$ with the three communities detected using the method based on edge-edge-betweenness extended for directed graphs proposed in [MPM06]

Figure 5.4 shows the result of this algorithm on the example used throughout this chapter. The method detects 3 communities. By looking at the dendrogram (figure 5.5), one can notice that vertex 0 stays alone a long time before joining the cluster $1, 2, 3, 4$. This corresponds much more to what would be expected from an algorithm taking into account the direction of edges, as explained earlier when comparing to the dendrogram given by the undirected community detector.

However, this algorithm is not perfect yet. The first issue arises with its choice of grouping 9 and 12. Though the difficulty of judging how appropriate this choice is, it remains a quite doubtful

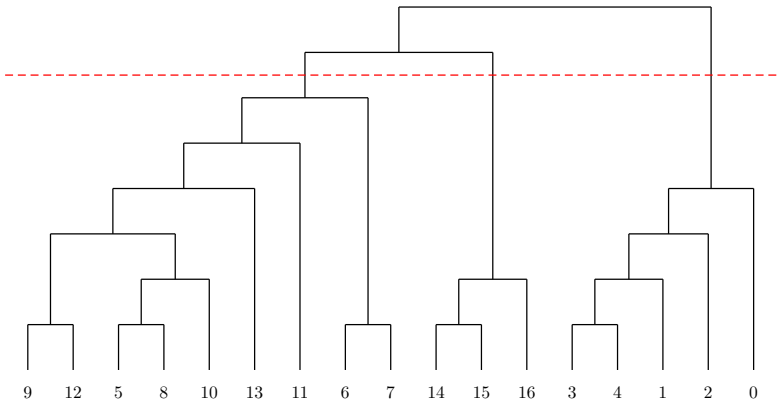


Figure 5.5: Dendrogram of the clustering used in order to detect the communities in the network of figure 5.4 using the extended algorithm based on edge-betweenness. The red dashed line represents the partition with greatest modularity.

choice and does not inspire much confidence. The second issue, which is more a lack than an issue, is that this algorithm does not seem to offer the possibility of an extension taking into account the edges probabilities.

5.1.2 Random Walks Based Algorithm

The other method found in the literature also is the adaptation of a well known community detection concept, the random walks. This algorithm, presented in [YSCH04], takes into account the direction of edges and is easily extensible to take into account their probabilities too.

This method is thus a random walk or agent-based approach. The random walks, of chosen length t, are not totally random though. There is a bias in the sense that an agent, called ant, cannot go on and edge it already went on unless there is no other choice. Another difference with the well known random walk algorithm presented in section 3.2 is the way those random walks are used. Each ant remembers its path and all vertices it went through. They then enter in a voting process to define how to split the graph into communities. For each pair of vertices u and v, the algorithm computes the number of ants that traversed both u and v, this number is denoted A. It also computes the number of ants that traversed one of u and v (or both), this number is B. The vertices u and v are merged if $\frac{A}{B} \geq c$ with c being the voting cut-off, a parameter to be specified to the algorithm. In fact, this is the sets S_u and S_v that are merged together, S_u being the set of vertices already merged with u.

This algorithm has unfortunately not been tested because, as it requires several parameters, it also requires many experiments in order to gain a good understanding of its behaviour even if it could perhaps gives interesting results.

5.2 Influence-based Community Detection

In order to cluster, a distance, or some other criterion, has to exist. The distance defined for this method is based on propagation processes. The whole idea of the method is that two vertices are close if they often propagate information to each other. The distance from vertex u to another vertex v should thus be defined using the results of several propagations starting from u. The distance between two vertices u and v should then be obtained by combining the distances from u to v and from v to u.

The propagation model is chosen to be ICM. Next step is to define a distance from vertex u to vertex v based on propagations denoted $d_u(v)$. Several propagations are run with vertex u as initial set. During each of these runs, some information is grasped in order to evaluate which vertices are joined by u, and after how many propagation steps. At each run, a distance from vertex u to another vertex w is computed as 1 if w is active at the end of the diffusion process or 0 otherwise.

Since a run is a random process, one run is not representative enough. K runs are thus done in order to compute a representative distance. After the K runs, the distance $d_u(v)$ represents thus the number of times the vertex v has been activated with $\{u\}$ as initial set.

The distance defined is an unidirectional distance. In order to obtain a symmetric distance for the hierarchical clustering, a distance is computed given equation 5.1.

$$d(v, w) = d(w, v) = d_v(w) \cdot d_w(v) \tag{5.1}$$

Once these distances are computed for each pair of vertices (u, v), the standard hierarchical clustering can start with all clusters being a single vertex. The distances between two clusters are computed following the concept of average linkage clustering, which yields this formula :

$$d(C_1, C_2) = \frac{\sum_{u \in C_1} \sum_{v \in C_2} d(u, v)}{|C_1| \cdot |C_2|}$$

5.2.1 Discussion

Some observation have been done on the few tests that have been done. The first important observation is that the proposed method in stochastic, that is the dendrogram is not the same at each run of the algorithm. However, while the value of K is increased, it seems that the number of communities found using the modularity as stop criterion tends to stabilize.

As shown in one dendrogram on figure 5.7, the vertex 0 is only gathered with another community very late in the clustering process which is the desired effect as mentioned in the previous section.

The last observation done is the isolation of the $\{7, 9\}$ community and this effect is also desired. There are indeed only two ways of leaving this community. The first possibility is to go in the blue community via the vertices 11 and 12 and the second possibility is to go in the red community via the vertex 8. Going into the blue community is quite easy, but coming back is highly improbable. On the other side, joining other red vertices implies first following the path $\langle 8, 6, 5\rangle$ which implies pretty low probabilities of joining other red vertices than 8.

By increasing the probability of path $\langle 9, 8, 6, 5\rangle$ — increasing the probabilities $p_{9,8}$, $p_{8,6}$ and $p_{6,5}$ — the suggested algorithm gathers the red and purple communities as expected since it makes leaving purple community toward red community as easy as joining it.

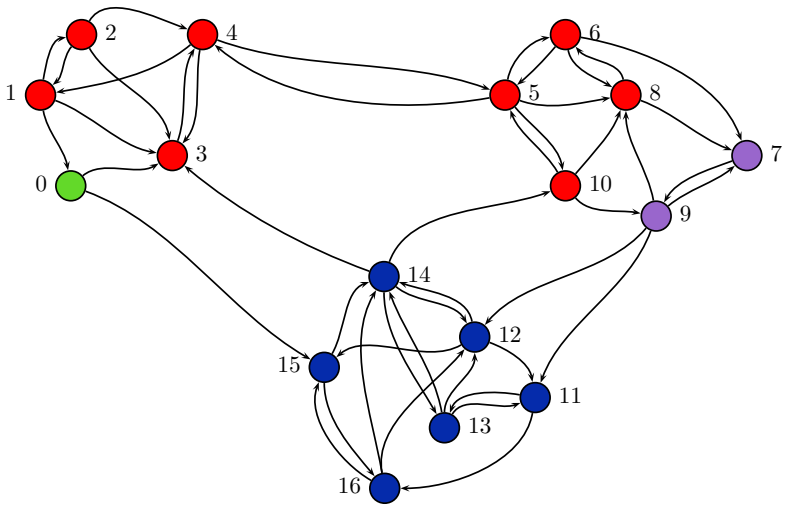


Figure 5.6: Example of a directed graph with uniform probabilities $p_{v,w} = 0.01$ with the four influence communities detected using the suggested method with $K = 10000$.

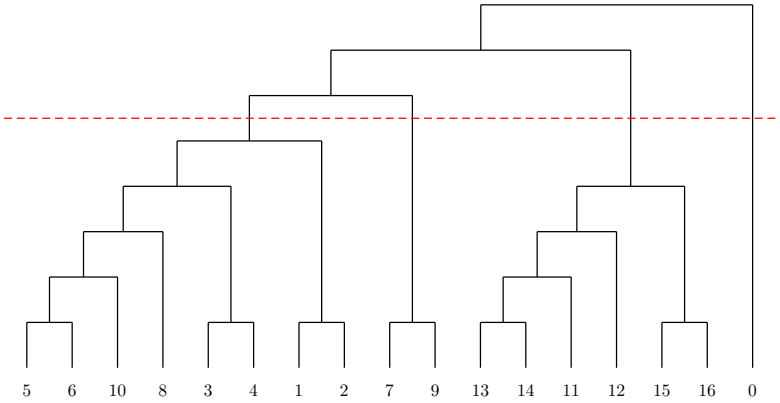


Figure 5.7: Dendrogram of the clustering used in order to detect the communities in the network of figure 5.6 using the suggested algorithm based on influence. The red dashed line represents the partition with greatest modularity.

Conclusion

The main subject of this work is the influence maximization problem — consisting in finding the maximal influence k-set — and its associated techniques.

First thing to keep in mind is the reason why each algorithm has been developed, and the links between them. From the beginning, there are the solvers (greedy and community) on one side and the models of diffusion (ICM, SPM, SP1M) on the other side. Both greedy and ICM are the accuracy references. They give the best results within all these techniques but are the most time consuming too. Greedy ICM_{10000} is indeed not suited for real-size problems. SPM is an approximation of the ICM model of diffusion. Its advantage — though the approximation causes a loss of data itself causing a loss of precision — lies in the fact that it allows a much faster way to compute the influence than ICM_{10000}. SPM can therefore be used on real-size problems, though obtained solutions are not the best one. SP1M is an intermediate between ICM and SPM. It is very close to SPM but makes things a bit more complicated in order to recover part of the data loss. SP1M is therefore around twice as time consuming as SPM, but usually manages to be a bit more accurate, depending on the graphs. On the other side, the community solver suggested in this work is an approximation for the greedy solver. Its goal is to take advantage of some community structure within the graph in order to reduce time complexity by cutting the graph into several smaller graphs, once again losing some accuracy.

The large battery of tests, conducted on both the existing algorithms and the suggested community-based solver, brings up several conclusions regarding these techniques. Tests have been conducted in order to compare all solutions and evaluate their respective worthiness depending on the kind of graph involved. The first observation obtained thanks to the tests it that, whatever the type of graph and the technique used, they are not worth the effort when propagation is highly restrained. A restrained propagation can have many different causes (sparse graph, low propagation probabilities,...) leading to the same effect, which is propagations starting from selected vertices do not interfere because they do not propagate enough. The graph structure does not therefore have enough impact on the propagation process to make it worth analysing the structure, as simply taking the k vertices with highest degree is enough to achieve good results. Algorithms presented in the viral marketing literature are interesting only when the graph structure plays a role in the propagation. Either graphs must have a specific and interesting structure or the propagation probabilities must be high enough for the propagations coming from different vertices to interfere.

The next step is to know for which situations the different solutions and approximations are best suited. The community solver is nearly perfect when combined with SPM on the randomly generated community graphs. It brings nearly no loss of accuracy, compared to greedy SPM, and offers an interesting speedup. It is of course even more impressive on the very dense community graphs than on the sparser communities. On the other side, on scale-free graphs, using the community solver instead of greedy solver implies a significant loss of accuracy. Debate is still open in order to determine if this loss might happen to be worth the computation time gain.

Community SPM, greedy SPM and degree ranking tend all three to be significantly behind greedy ICM_{10000} and it seems hard to determine which one of those three is best, especially as it slightly changes from one graph to another.

The second main problem considered in this work is the directed community detection. The suggested algorithm, based on ICM propagations, has been tested on a small graph example offering several interesting patterns. The obtained dendrogram and communities seem quite promising, but not enough tests have been conducted to confirm anything serious about this new algorithm. Thoroughly testing this algorithm is one of the doors left open by this work.

Many other aspects of this work leave some perspectives for future works. For example, the several patterns detected through the testing process still lacking an explanation.

Another example is the testing process of influence maximization solutions. Though it is far more complete than the testing of the community detection algorithm, many dimensions are still neglected. One of them is the tested graph families as only two kind of graphs have been tested. No tests have been conducted on other graphs such as hierarchical networks [RSM+02] for example which occurs in many reals complex networks. Other dimensions of the testing have also been neglected as for example the variation of the size of the graphs or even the testing on graphs whose edges have different propagation probabilities. The different versions of the community-based solver, advanced and iterative, also lack much testing as most of the attention went to the basic community-based solver. Much is left to be done in testing influence maximization solutions.

One more dimension has not been considered regarding community-based solvers. They allow faster finding of the k-set, but also require that the communities are detected in the graph and this can be time consuming. An open question is thus to know how fast can a graph be partitioned into communities in order for the solver to work properly on it.

Suggested algorithms also are open for future improvements. The community-based solver can suffer many modifications in order to take into account some graph information or another. The advanced and iterative versions only are two examples of such improvement attempts, but many others are left to be found and experimented. Only once most of these improvements have been tested will it really be possible to evaluate what the community-based solver really has to offer.

A

Code documentation

This appendix gives a little more information about the programs that have been developed and how to use them in the first section. The second section gives the specification of the two important abstract classes that are `MaximizationProblem` and `DiffusionModel` in order to be ready to add new solvers or models of diffusion. And eventually, the last section shows the GUI and explain briefly how to use it.

A.1 Programs usage

The most important program furnished is `FindMaxInfluenceSet` whose goal is — as clearly mentioned in its name — to find a set of maximal influence. The program takes 5 mandatory parameters.

```
viralmarketing.FindMaxInfluenceSet file k solver model detail
```

Parameters	
file	is the path to the file containing the graph in the GraphML format [Tea04]
k	is the size of the k-set to find
solver	is the solver to be used
model	is the diffusion model to be used
detail	is `true` or `false`, if set to true, the details are given, this all the intermediary sets are given, this is a 1-set, a 2-set, ..., a k-set of maximal influence
Solvers	
b	the brute force algorithm
g	the greedy algorithm of Nemhauser et al. (listing 1.1 at section 1.2)
r	the distributed version of the greedy algorithm
d	the algorithm based on the degree ranking (section 4.3.1)
b	the algorithm base on the betweenness (section 4.3.2)
c	the naive community based algorithm (section 3.3.1)
a	the advanced community based algorithm (section 3.3.3)
i	the iterative community based algorithm (section 3.3.3)

Diffusion models	
l	Linear Threshold Model (section 1.3.1)
i	Independent Cascade Model (section 1.3.2)
s	Shortest Path Model (section 1.3.3)
p	SP1 Model (section 1.3.3)

The second useful program is used to detect communities in a graph. The program takes 4 mandatory parameters.

```
viralmarketing.communities.FindCommunities clust quality file output
```

Parameters	
clust	is the clusterer used to detect communities
quality	is the quality function to maximize
file	is the path to the file containing the graph in the GraphML format [Tea04]
output	is the path to the file where the partitionned graph must be written with the GraphML format [Tea04]
Clusterers	
r	the algorithm based on random walks (section 3.2)
e	the algorithm based on edge-betweenness (section 5.1.1)
d	the extended version of the algorithm based on edge-betweenness (section 5.1.1)
Quality functions	
m	the modularity (section 3.1.1)
p	the performance (not discussed in this work, but defined in [BE05])

A third program taking only one mandatory parameters provides some informations about a graph such as the number of vertices, edges, the density, the radius and diameter, the degree distribution, ...

```
viralmarketing.utils.GraphStats file
```

Parameters	
file	is the path to the file containing the graph in the GraphML format [Tea04]

A.2 Specification of abstract classes

Two abstract classes are the top classes of the main classes hierarchies that represents a solver and a diffusion model. In order to add a new solver or diffusion model, these classes must be extended. This section gives the specification of the methods from these classes and underlines the abstract methods to implement in order to get concrete classes.

viralmarketing.MaximizationProblem

Fields

An instance of the graph

```
protected Graph<Vertex,Edge> graph;
```

The k-set found

```
protected Set<Vertex> targetset;
```

Constructors

Pre: graph != null

Post: An instance of this is created

```
public MaximizationProblem (Graph<Vertex,Edge> graph);
```

Methods

Get the target set

Pre: The problem must be solved

Post: The returned value contains the target set for this problem

```
public Set<Vertex> getTargetSet();
```

Solve the problem

Pre: model != null

Pre: model is a model on the graph of this problem

Pre: 0 <= k <= the number of nodes in the graph of this problem

Post: The maximization problem is solved

```
public void solve (DiffusionModel model, int k);
```

Solve the problem

Pre: model != null

Pre: model is a model on the graph of this problem

Pre: tset != null

Pre: tset.size() < k <= the number of nodes in the graph of this problem

Post: The maximization problem is solved starting from the specified "tset"

```
public abstract void solve (DiffusionModel model, int k, Set<Vertex> tset);
```

The only abstract method in the `MaximizationProblem` is the method `solve` and its goal is to solve the problem of finding a k-set of maximum influence given a certain model of diffusion. The three parameters of the method is first a model of diffusion (`DiffusionModel`), next the size of the k-set to find and finally a set *tset* of vertex that are pre-selected. Of course, the value of k is between the size of the set *tset* and the number of vertices in the graph.

viralmarketing.models.DiffusionModel

Fields

The default value of R (the number of runtimes for estimating the influence)

```
private static final int RUN_TIMES = 10000;
```

An instance of the graph

```
protected Graph<Vertex,Edge> graph;
```

Constructors

Pre: g != null

Post: An instance of this is created

Post: The model is initialized

```
public DiffusionModel (Graph<Vertex,Edge> g);
```

Methods

Initialize the graph for the model

Pre: -

Post: The graph has been initialized and the diffusion can start

```
protected void init();
```

Run a diffusion process

Pre: A0 != null

Pre: The vertices of A0 are all in the network of the model

Post: The returned value contains a list of set of vertices so that the information has spread from the vertices of A0 to them. The ith element of the list contains the vertices activated at the ith time step

```
public abstract List<Set<Vertex» runStepProcess (Set<Vertex> A0);
```

Run the process of diffusion of informations from an initial vertex

Pre: v != null

Pre: v is in the network of the model

Post: The returned value contains the set of vertices so that the information has spread from the vertex v to them

```
public Set<Vertex> runProcess (Vertex v);
```

viralmarketing.models.DiffusionModel

Methods

Run the process of diffusion of informations from an initial set

Pre: A0 != null

Pre: The vertices of A0 are all in the network of the model

Post: The returned value contains the set of vertices so that the information has spread from the vertices of A0 to them

```
public Set<Vertex> runProcess (Set<Vertex> A0);
```

Get the influence of a set of vertices

Pre: A != null

Pre: The vertices of A are all in the network of the model

Post: The returned value contains the influence of A (mean and standard deviation), that is the number of vertices that have been influenced by vertices from A in the model

```
public RandomDouble getInfluence (Set<Vertex> A);
```

Get the influence of a set of vertices

Pre: A != null

Pre: The vertices of A are all in the network of the model

Pre: runtimes >= 1

Post: The returned value contains the influence of A (mean and standard deviation), that is the number of vertices that have been contaminated by A in the model computed running "runtimes" time the diffusion process

```
public RandomDouble getInfluence (Set<Vertex> A, int runtimes);
```

Get the influence of a vertex

Pre: v != null

Pre: The vertex v is in the network of the model

Post: The returned value contains the influence of v (mean and standard deviation), that is the number of vertices that have been contaminated by v in the model

```
public RandomDouble getInfluence (Vertex v);
```

Get the influence of a vertex

Pre: v != null

Pre: The vertex v is in the network of the model

Pre: runtimes >= 1

Post: The returned value contains the influence of v (mean and standard deviation), that is the number of vertices that have been contaminated by v in the model computed running "runtimes" time the diffusion process

```
public RandomDouble getInfluence (Vertex v, int runtimes);
```

---

viralmarketing.models.DiffusionModel
Methods
Construct a set of active nodes **Pre:** A0 != null **Pre:** The vertices of A0 are all in the network of the model **Post:** The returned value contains a set of actives nodes, this is the nodes from the A0 set **Post:** The nodes of the graph gets a decoration "active" with Boolean value representing whether they are or not in A0 `protected Set<Vertex> setActiveNodes (Set<Vertex> A0);`
Called before starting the maximization process `public void initialize (Set<Vertex> initSet);`
Called when a node is added to the set A `public void nodeAdded (Vertex v);`
Called when a node is removed from the set A `public void nodeRemoved (Vertex v);`

Only one method is abstract and need to be implemented in the concrete subclasses, this is the `runStepProcess` method which run a diffusion process starting from a set A_0 of vertices. The method must returns a list of the vertices activated at each step of the diffusion process.

In this abstract class, the influence is always estimated using by default $10,000$ runs, but the number of runs can be changed by using the specific version of the method `getInfluence`. If a subclass wants to compute the influence with another technique, this class must only overrides the `public RandomDouble getInfluence (Set<Vertex> A, int runtimes);` method, all the other methods `getInfluence` are defined by calling the latter method.

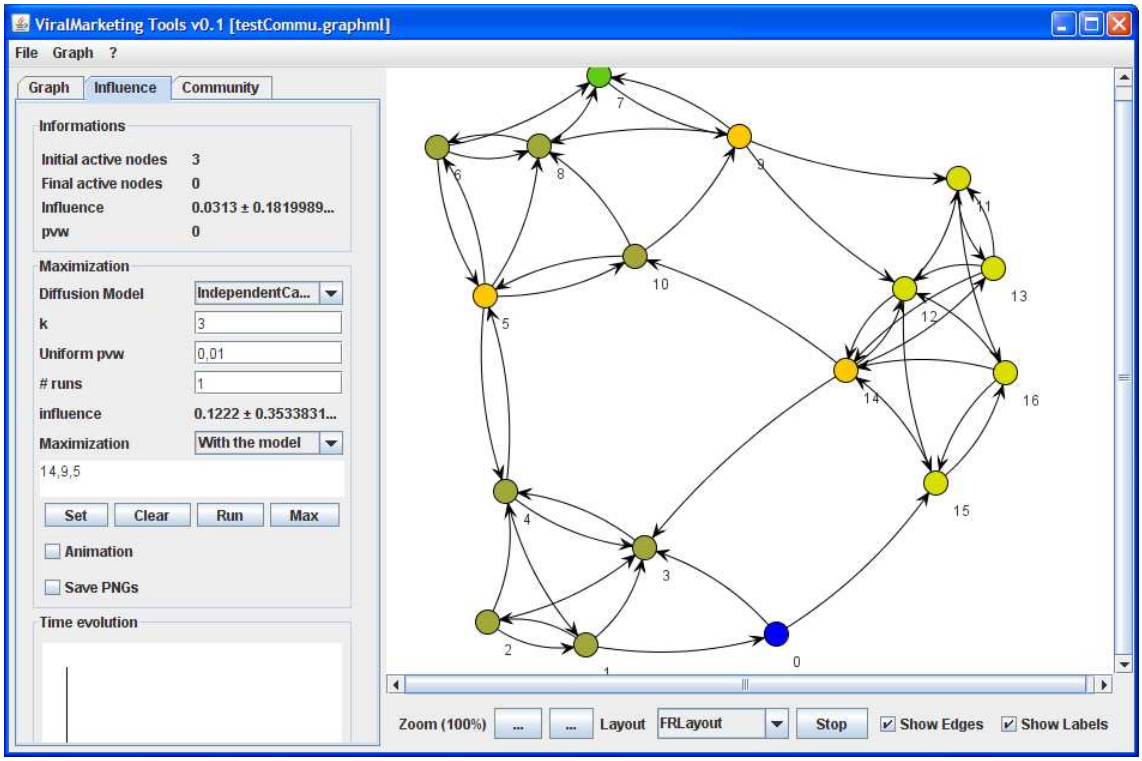
Finally, the three last methods of the class `DiffusionModel` can be overridden in subclasses. In the abstract class `DiffusionModel`, they do nothing, but these methods are called by the greedy solver each time a new vertex is added to the set A before computing the marginal gain and each time the added vertex is removed from this set. These methods are for example used in the SPM model in order to maintain the data structures up-to-date to speedup the computation.

A.3 Using the GUI

A light GUI is also available and offers the ability to visualize graphs and to performs operations on it such as running a diffusion process, finding the k-set of maximum influence, detecting the communities in a graph, ...

The interface is divided into three parts. On the left, there is the tools pane and on the right there

is the visualization pane with a control bar. The tools pane is structured with three tabs, the first one gives information about the graph, the second is linked to the influence maximization problem and the last one is linked to communities.



This appendix is not an user manual but only gives some keys in order to use the ligt GUI. However, the GUI is quite intuitive and performing an action is not so difficult.

The first thing to do before working is to load a graph into the application, it is done using the `File > Open` menu or using the keyboard shortcut `CTRL+O`. The graphs must be in the GraphML format [Tea04], see section A.4 for detailed explanation about the format.

After a graph is loaded, the first tab of the tools pane gives some informations about the graph. When a vertex is selected, informations about it is also displayed. Moreover, the distribution of the degrees is shown as an histogram. The control pane allows to control the rendering of the graph. Buttons are available to zoom in or out, to change the layout algorithm used to render the graph and to show or hide the edges and/or the name of the vertices.

The second tab of the tools pane is related to the influence maximization problem. From this tabs, simulation of the propagation of information can be run for several models of diffusion and a k-set of maximal influence can be found. Notice that it seems that there is a small bug for finding a k-set of maximal influence with the GUI. In order to do such simulations, vertices must be selected and added to the initial set, this is done by right clicking on the vertices and selecting `Activate` in the menu to add them to the initial active set or `Desactivate` to remove them.

Finally, the last tab contains tools related to the community detection. From this tab, it is possible to detect and display communities on the graph.

A.4 The graphs file

Listing A.1 gives an example of a small graph written with the GraphML format. There is some important restrictions on this file if it has to be used with the program. The first thing is that the id's of the vertices must be consecutive integers starting from 0. There is no restriction on the names of the vertices nor the id's of the edges. Finally, each edge must have an attribute `pvw` specifying the probability $p_{v,w}$.

In order to use the community-based solvers, an attribute `foundcommunity` must be added to each vertices specifying in which community the vertex is. The communities are identified by consecutive integers starting from 0.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml" xmlns:xsi="
   http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
   graphml.graphdrawing.org/xmlns/graphml">
3    <graph id="G" edgedefault="directed">
4      <node id="0" name="0" foundcommunity="1" />
5      <node id="1" name="1" foundcommunity="0" />
6      <node id="2" name="2" foundcommunity="1" />
7      <node id="3" name="3" foundcommunity="0" />
8      <edge pvw="0.90" id="e0" source="0" target="1" />
9      <edge pvw="0.90" id="e1" source="0" target="2" />
10     <edge pvw="0.90" id="e2" source="1" target="2" />
11     <edge pvw="0.01" id="e3" source="2" target="3" />
12     <edge pvw="0.01" id="e4" source="0" target="3" />
13   </graph>
14 </graphml>
```

Listing A.1: An example of a graph in the GraphML format

B

Community Graph Generation

A community graph generator has been developed for the purpose of creating random graphs sharing some properties to carry on some tests. It can be found in class `CommunityGraphGenerator`. The call parameters are discussed one after the other in order to go through every aspect of the graphs creation. Other choices concerning the created graph that would not appear within the call parameters will be discussed afterwards.

Lots of things can vary in a graph, especially with communities. This is the reason why the generator takes many parameters in order to allow generating as many different patterns as possible.

```
viralmarketing.utils.CommunityGraphGenerator numNodes numCommun
ratioStdCommunSize meanExtDegree stdExtDegree meanIntDegreeRatio
stdIntDegreeRatio filename p
```

Parameters	
numNodes	is the number of vertices constituting the whole graph
numCommun	is the number of communities
ratioStdCommunSize	contributes determining the standard deviation for the size of communities. The sizes of communities are generated one after the other following each time a normal distribution. Its mean is the number of nodes left to distribute on the number of communities left to fill. Its standard deviation is the same as its mean divided by this parameter. This way, the normal distribution will adapt itself to the number of nodes and communities left to assign and should ensure quite realistic sizes. Note that this is a ratio. A high value parameter means a low standard deviation
meanExtDegree & stdExtDegree	are the mean and standard deviation of another normal distribution. This probability distribution is used in generating the number of outgoing edges leaving each community. Once the number of edges leaving a community is decided, those edges sources are randomly chosen within the community and the destinations are randomly chosen within all nodes outside the community. These two parameters thus decide how isolated will be communities

meanIntDegreeRatio & stdIntDegreeRatio	are used to compute the mean and standard deviation of a normal distribution for each community. Those two parameters are ratios such as those used for the community sizes generation. This means that they both will adapt to the size of the community the following way : $mean = commuSize/meanRatio$ and $std = commuSize/stdRatio$. This probability distribution is used in generating nodes internal degree. The degrees generated this way only are minimal values as they might be increased to ensure connectivity. The ensuring of connectivity is discussed below. These two parameters thus decide how dense the communities should be, and if there should be some very high and low degree nodes or if all should have quite the same importance.
filename	is the path to the file where the generated graph must be written in the GraphML format [Tea04]
p	is the propagation probability that should be associated with each edge. It is chosen to be common to all edges. It could easily be improved though

These are the main characteristics that are involved in the graph generation but everything has not been said yet. Let's first have a look at the order in which things are generated, and thus at a high level algorithm presented in listing B.1.

```
1 Determine size of each community
2
3 for each community C_i
4    generate intra-community edges
5    generate extra-community edges
6 end for
```

Listing B.1: High-level algorithm for random community graph generation

There are many mysteries left about the way the intra-community edges are generated. The listing B.2 on page 77 gives a high level algorithm presenting the way the intra-community edges are generated while ensuring connectivity.

This way of ensuring connectivity every time a node is considered is a very good way to do it. First this algorithms leaves as much as it can to randomness. It only interferes when none of th enot yet created nodes is joined, in which case it ensures connectivity at once by making sure one edge goes out toward an already connected node. The hope is of course that the random generation of edges will keep the set *Connected* nonempty so that the algorithm never has to enforce anything introducing a less random behavior.

```
1  NonConnected ← V
2  Connected ← ∅
3  Added ← ∅
4
5  while Added ≠ V
6     d ← generate node degree
7     if Connected ≠ ∅
8        v ← rdm node in Connected
9        Connected ← Connected \ {v}
10       Added ← Added ∪ {v}
11       for i ← 1 to d do
12          choose randomly a destination within all nodes of the community
13          if the edge doesn't exist
14             create it
15             if the destination is in NonConnected
16                remove it and put it in Connected
17             end if
18             i++
19          end if
20       end for
21    else
22       v ← rdm node in NonConnected
23       Connected ← NonConnected \ {v}
24       Added ← Added ∪ {v}
25       choose randomly a destination within all nodes from Added
26       create the edge
27       for i ← 1 to d − 1 do
28          choose randomly a destination within all nodes of the community
29          if the edge doesn't exist
30             create it
31             if the destination is in NonConnected
32                remove it and put it in Connected
33             end if
34             i++
35          end if
36       end for
37    end if
38 end while
```

Listing B.2: High-level algorithm for intra-community edges generation

C

Results of the tests

This appendix gives the results of the several tests performed whose results are presented and analysed in the chapter 4. The full results are not presented such that this appendix in not too overloaded, only the really interesting information is kept.

The graphs from `random500_1` and `random500_2` have been generated with the algorithm from [VL05] with the following parameters

```
> distrib 500 2.5 1 1000 5 > mydegs
> graph -vv  mydegs > mygraph
```

The graphs from `commu_1` and `commu_2` have been generated with the algorithm presented in appendix B with the following parameters

```
> CommunityGraphGenerator 500 5 40 15 5 5 2
> CommunityGraphGenerator 500 5 40 80 25 100 10
```

C.1 Performance

These results have been obtained by searching a best k-set using several solvers. The results are mean done on the 9 graphs from each set. The influence mentionned are computed using $\sigma_{ICM_{10000}}$.

	1	5	10	15	20	p
ICM_{10000}	1.191944	5.736889	11.254011	16.654867	22.016678	
Degree c_{D_1}	1.192222	5.725744	11.217389	16.562011	21.849267	0.01
Betweenness	1.121800	5.519300	10.777333	15.865022	20.884633	
SPM	1.194678	5.733367	11.267300	16.681500	22.037767	
ICM_{10000}	2.212133	9.396756	17.094322	24.106256	30.753600	
Degree c_{D_1}	2.206433	9.196844	16.654156	23.101456	29.164122	0.05
Betweenness	1.706111	7.846267	14.035000	19.365233	24.383122	
SPM	2.199278	9.350444	17.072556	24.030244	30.641589	
ICM_{10000}	4.237333	15.532333	25.705178	34.289522	42.032733	
Degree c_{D_1}	4.165456	14.659678	23.969078	31.159689	37.588478	0.10
Betweenness	2.676356	11.238444	18.212956	23.630456	28.709522	
SPM	4.234967	15.394333	25.318222	33.807089	41.236933	

Table C.1: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `random500_1` set

	1	5	10	15	20	p
ICM_{10000}	2.880167	9.898078	16.344800	22.080800	27.522844	
Degree c_{D_1}	3.002200	10.178033	16.592000	22.267189	27.646000	0.01
Betweenness	3.025867	10.178189	16.550322	22.241389	27.590011	
SPM	3.007444	10.191044	16.600422	22.325911	27.782644	
ICM_{10000}	19.950600	38.829644	48.494211	55.555289	61.667178	
Degree c_{D_1}	19.896678	38.819133	48.361656	54.964067	60.720244	0.05
Betweenness	19.968644	38.789533	48.248089	55.000311	60.685600	
SPM	19.893344	38.837278	48.454900	55.414000	61.544078	
ICM_{10000}	85.145067	93.849356	100.826422	106.861433	112.782356	
Degree c_{D_1}	85.117533	92.601567	97.359600	101.490078	105.466844	0.10
Betweenness	85.153756	92.550011	97.393800	101.775500	105.829056	
SPM	85.073522	92.494956	97.433244	102.023611	106.753233	
SP1M	85.100411	92.548467	97.774078	103.229011	108.873689	

Table C.2: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings, greedy SPM and greedy SP1M for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `random500_2` set

	1	5	10	15	20	p
ICM_{10000}	2.467622	11.956633	23.346533	34.349444	44.932389	
Degree c_{D_1}	2.453456	11.533722	22.557400	33.241144	43.822767	0.01
Betweenness	2.125167	9.732467	17.800567	25.710944	33.720478	
SPM	2.451967	11.920933	23.336822	34.380678	44.940111	
ICM_{10000}	60.035514	222.730457	276.874157	299.420850	314.640525	
Degree c_{D_1}	55.866278	104.116200	171.731033	215.737889	258.449067	0.05
Betweenness	48.665244	149.778733	210.716711	247.496056	272.076444	
SPM	56.538267	206.993100	247.784544	273.353900	294.862333	
ICM_{10000}	208.884633	467.165767	471.702325	473.232529	474.735980	
Degree c_{D_1}	174.359922	250.487978	321.612500	385.310100	425.794233	0.10
Betweenness	173.186889	381.804911	435.637344	464.517989	468.410344	
SPM	180.091200	459.497222	465.962022	468.585489	470.061356	
SP1M	188.064460	464.688460	468.338460	472.258140	473.510360	

Table C.3: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings, greedy SPM and greedy SP1M for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `commu500_1` set

	1	5	10	15	20	p
ICM_{10000}	1.355889	6.516067	12.729467	18.796656	24.759922	
Degree c_{D_1}	1.356800	6.520344	12.736722	18.800778	24.767211	0.01
Betweenness	1.203167	5.887956	11.672244	17.438789	23.171822	
SPM	1.357678	6.517644	12.760711	18.835756	24.838256	
ICM_{10000}	3.284656	14.588911	26.965567	38.209511	48.659644	
Degree c_{D_1}	3.231000	14.413467	26.709656	37.765578	48.033978	0.05
Betweenness	2.305200	10.691567	20.486122	30.007644	39.164278	
SPM	3.253289	14.612178	26.985511	38.185467	48.635078	
ICM_{10000}	8.065400	32.779756	56.621467	76.105644	93.047978	
Degree c_{D_1}	7.507889	31.453178	54.935733	73.565622	89.520256	0.10
Betweenness	4.957244	21.739189	39.370889	55.338633	70.078300	
SPM	8.001200	32.567489	56.358222	75.767922	92.412433	

Table C.4: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `commu500_2` set

This test has been done on the `commu500_2` set and its goal is to see the effect of varying the probability $p_{v,w}$. The influence are the $\sigma_{ICM_{10000}}$ influence and the influence are computed on the 10-set of maximal influence.

	0.01	0.05	0.10	0.20	0.30	0.40
ICM_{10000}	2.805100	17.833200	50.743800	196.846400	352.556700	427.144300
Degree c_{D_1}	2.835300	17.766200	49.903100	186.086900	345.270300	420.661000
Betweenness	1.765500	11.390100	33.132000	149.261900	324.055200	414.014300
SPM	2.855000	17.859100	50.325400	192.594200	346.822500	421.371600
SP1M	2.841600	17.800600	50.550000	196.558000	348.067900	421.585200

Table C.5: ICM_{10000} influence for the k-set found while $k = 10$ using several methods : greedy ICM_{10000}, degree and betweenness rankings, greedy SPM and greedy SP1M for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05, 0.10, 0.20, 0.30 and 0.40 for the `commu500_2` set

These two last tables gives the results for the `roget` and the `EVA` graphs.

	1	5	10	15	p
ICM_{10000}	1.245500	6.066100	11.977000	17.726400	
Degree c_{D_1}	1.238500	6.080200	11.894600	17.688800	0.01
Betweenness	1.190900	5.957400	11.654000	17.363200	
SPM	1.251400	6.080000	11.981200	17.795200	
ICM_{10000}	2.895000	12.953700	24.038900	34.030300	
Degree c_{D_1}	2.893000	13.068300	23.253800	33.238400	0.05
Betweenness	2.298700	11.797900	21.464000	31.084200	
SPM	2.917800	13.006200	24.162500	34.084300	
ICM_{10000}	9.391700	35.907600	59.146200	77.165000	
Degree c_{D_1}	9.477300	35.417100	55.448800	72.723700	0.10
Betweenness	6.011700	29.850300	49.013200	66.208200	
SPM	9.344200	35.579800	58.957200	76.310800	
SP1M	9.483900	35.850000	59.098500	77.864500	

Table C.6: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `roget` graph

	1	5	10	15	p
ICM_{10000}	6.552100	16.333800	26.592300	-	
Community ICM_{10000}	6.491100	16.396500	26.507300	35.225800	0.01
Degree c_{D_1}	6.511900	16.377600	26.483000	35.206100	
Betweenness	1.876900	9.499100	16.336800	23.089500	
SPM	-	-	-	-	
ICM_{10000}	28.558500	62.542200	-	-	
Community ICM_{10000}	28.602100	62.303400	93.356500	117.080100	0.05
Degree c_{D_1}	28.588000	62.345300	93.180800	117.270200	
Betweenness	6.755300	28.791500	42.701800	56.855900	
SPM	-	-	-	-	
ICM_{10000}	56.366000	120.507600	-	-	
Community ICM_{10000}	56.113700	120.192300	178.469500	222.223500	0.10
Degree c_{D_1}	56.300400	120.452300	178.382700	222.210100	
Betweenness	15.943600	55.695400	78.455600	102.280100	
SPM	56.248500	120.344700	179.206500	222.278600	

Table C.7: ICM_{10000} influence for the k-set found while k is set to 1, 5, 10, 15 and 20 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `EVA` graph

C.2 Varying the size of the graph

These two tests have as goal to check if the properties observed are valid while the size of the graph changes. These results are obtained by finding the 10-set of maximum influence for 6 graphs of each size, and the values reported are means on the results for the 6 graphs.

	50	100	500	1000	p
ICM_{10000}	10.939583	11.736017	16.193633	20.468517	
Degree c_{D_1}	10.827917	11.695133	16.581900	20.521117	0.01
Betweenness	10.867400	11.700567	16.539867	20.465767	
SPM	10.951567	11.749417	16.583617	20.522483	
ICM_{10000}	14.684850	18.981017	48.206567	78.114950	
Degree c_{D_1}	14.049367	18.615200	48.144583	77.976133	0.05
Betweenness	14.234333	18.707617	48.013833	77.981533	
SPM	14.662100	18.918000	48.196900	77.996383	
ICM_{10000}	19.415083	28.537783	100.086000	185.082800	
Degree c_{D_1}	17.819733	27.294400	96.877533	168.828783	0.10
Betweenness	18.204633	27.526667	96.918083	178.133917	
SPM	19.017717	27.862533	96.974517	178.188267	

Table C.8: ICM_{10000} influence for the k-set found while the size of the graph is set to 50, 100, 500 and 1000 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `random_2` set

	50	100	500	1000	p
ICM_{10000}	10.951283	11.822200	12.730783		
Degree c_{D_1}	10.919583	11.746017	12.751517	12.809017	0.01
Betweenness	10.591083	11.091767	11.645550	11.867517	
SPM	10.961750	11.828667	12.778500	12.815933	
ICM_{10000}	15.034700	20.169400	26.981467	27.899900	
Degree c_{D_1}	14.709833	19.454900	26.746400	27.653233	0.05
Betweenness	13.226883	16.402833	20.373783	22.062217	
SPM	15.023417	20.178667	26.999433	27.906467	
ICM_{10000}	20.567783	32.871867	56.638367	63.080867	
Degree c_{D_1}	19.504300	30.382467	54.874717	60.989267	0.10
Betweenness	17.010517	25.386467	39.230100	46.260433	
SPM	20.274700	32.543350	56.326083	62.663017	

Table C.9: ICM_{10000} influence for the k-set found while the size of the graph is set to 50, 100, 500 and 1000 using several methods : greedy ICM_{10000}, degree and betweenness rankings and greedy SPM for the uniform probabilities $p_{v,w}$ set to 0.01, 0.05 and 0.10 for the `commu_2` set

C.3 Testing different ICM_R

These tests have as goal to find a good value of R, the number of runtimes to be used in order to get an estimate for the value of the influence under the ICM model of diffusion. The tests have been performed for 3 graphs un each set, for each value of R, 7 20-set have been tested. In those 7 20-set is the 20-set of maximal influence, a 20-set with a very low influence and the 5 remaining 20-set are chosen randomly.

	Mean	StdDev	Mean	StdDev
10	0.790952	0.276879	9.971429	1.062033
50	0.780571	0.113806	9.917333	0.477185
100	0.782048	0.082071	9.918476	0.342388
500	0.782533	0.036671	9.919495	0.158138
1000	0.780295	0.027116	9.922229	0.112683
5000	0.780381	0.012228	9.914416	0.049823
10000	0.781110	0.008048	9, 916920	0.036116
50000	0.781489	0.004052	9.920630	0.016762
	$p = 0.01$		$p = 0.10$	

Table C.10: Mean and standard deviation obtained with ICM_R for R varying between 10 and 50000 for the `random_1` set with the probability $p_{v,w}$ set to 0.01 and 0.10.

	Mean	StdDev	Mean	StdDev
10	1.594286	0.400179	25.989048	2.762259
50	1.604667	0.177300	26.084476	1.285638
100	1.598190	0.120771	26.154667	0.863929
500	1.593600	0.052483	26.098505	0.393452
1000	1.597481	0.038597	26.098871	0.264987
5000	1.595580	0.017255	26.117220	0.108683
10000	1.595657	0.012867	26.118047	0.086716
50000	1.595550	0.005486	26.108867	0.037278
	$p = 0.01$		$p = 0.10$	

Table C.11: Mean and standard deviation obtained with ICM_R for R varying between 10 and 50000 for the `commu_2` set with the probability $p_{v,w}$ set to 0.01 and 0.10.

Glossary

$\sigma(\cdot)$	*See Influence*
Active vertex	A vertex of a network is said to be active if the information has reached the vertex and is accepted by it.
Betweeness	I a graph $G = (V, E)$, the betweeness of a vertex $v \in V$ is the number of shortest paths between any two vertices from the graph that go trough v. The betweeness of a vertex v is noted $c_B(v)$
Community	In a graph, a community is a subgraph whose density is high relative to the density of the graph.
Connected graph	A connected graph $G = (V, E)$ is an undirected graph such that between any pair of vertices (u, v), there exists a path connecting them. (For the case of directed graph, this is tge underlying undirected graph that is considered)
Contaminated vertex	*See Active vertex*
Degree	I a graph $G = (V, E)$, the degree of a vertex $v \in V$ is the number of edges incident to the vertex. The degree of a vertex v is noted $d(v)$
Diffusion model	A diffusion model describes how the information propagates through the vertices of a network, that is how the step-by-step dynamics of the propagation unfolds.
Diffusion process	The diffusion process is the process by which the information propagates through the vertices of the network.
Inactive vertex	A vertex of a network is said to be inactive if the information has not reached the vertex or has not been accepted by it.

Influence	Given a graph $G = (V, E)$ and a model of diffusion, the influence of a set of vertices $A \subseteq V$ is the expectation of the number of active vertices after a diffusion process is run.
Influence maximization problem	Given a graph $G = (V, E)$ and a diffusion model M, the influence maximization problem consists in finding a k-set $A \subseteq V$ of maximum influence, that is with maximum value of $\sigma_M(A)$.
Non-contaminated vertex	*See Inactive vertex*
Scale-free graph	A scale-free graph is complex graph in which some vertices acts as highly connected hubs (high degree) although most vertices are of low degree. Their degree distribution follows a power law relationship.
Social network	A social network is a social structure made of nodes (which are generally individuals or organizations) that are tied by one or more specific types of relations, such as values, visions, idea, financial exchange, friends, kinship, dislike, trade, web links, sexual relations, disease transmission (epidemiology), or airline routes.
Solver	High level algorithm whose goal is to find a k-set of maximal influence
Strongly connected graph	A strongly connected graph $G = (V, E)$ is a directed graph such that between any pair of vertices (u, v), there exists a directed path connecting them.

Bibliography

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, January 2002.

[BE05] Ulrik Brandes and Thomas Erlebach. *Network Analysis : Methodological Foundations*. Springer-Verlag, 2005.

[BM06] Vladimir Batagelj and Andrej Mrvar. *Pajek datasets*. http://vlado.fmf.uni-lj.si/pub/networks/data/, 2006.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[CNM04] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, December 2004.

[DDDGA05] Leon Danon, Jordi Duch, Albert Diaz-Guilera, and Alex Arenas. Comparing community structure identification, 2005.

[Fre77] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, March 1977.

[GN02] Michelle Girvan and M. E. J. Newman. Community structure in social and biological networks. *PNAS*, 99(12):7821–7826, June 2002.

[KKvT03] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. *9-th International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[KKvT05] David Kempe, Jon Kleinberg, and Éva Tardos. Influential nodes in diffusion model for social networks. *International Colloquium on Automata, Languages and Programming*, 3580:1127–1138, July 2005.

[KS06] Masahiro Kimura and Kazumi Saito. Tractable models for information diffusion in social networks. *10-th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 259–271, 2006.

[LP05] Matthieu Latapy and Pascal Pons. Computing communities in large networks using random walks. *LNCS*, 3733:284–293, 2005.

[MPM06] David Meunier and Hélène Paugam-Moisy. Cluster detection algorithm in neural networks, 2006.

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[New06] M. E. J. Newman. Modularity and community structure in networks. *PNAS*, 103:8577, February 2006.

[NLGC02] Kim Norlen, Gabriel Lucas, Mike Gebbie, and John Chuang. Eva: Extraction, visualization and analysis of the telecommunications and media ownership network. *Proceedings of International Telecommunications Society 14th Biennial Conference*, 2002.

[NWF78] George L. Nemhauser, Laurence A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions–i. *Mathematical Programming*, 14(3):265–294, 1978.

[OFN07] The JUNG Team (Joshua O'Madadhain, Danyel Fisher, and Tom Nelson). *Java Universal Network/Graph Framework (JUNG)*. http://jung.sourceforge.net/, 2003–2007.

[Pon06] Pascal Pons. Post-processing hierarchical community structures: Quality improvements and multi-scale view, 2006.

[RD02] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. *8-th International Conference on Knowledge Discovery and Data Mining*, pages 61–70, 2002.

[RSM+02] E. Ravasz, A.L. Somera, D.A. Mongru, Z.N. Oltvai, and A.L. Barabasi. Hierarchical organization of modularity in metabolic networks. *Science*, 297(5586):1551–1555, August 2002.

[Tea04] The GraphML Team. *The GraphML File Format*. http://graphml.graphdrawing.org/, 2001–2004.

[VL05] Fabien Viger and Matthieu Latapy. Fast generation of random connected graphs with prescribed degrees. *LNCS*, 3595:440–449, 2005.

[WN99] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.

[YSCH04] Maxwell Young, Jennifer Sager, Gabor Csardi, and Peter Haga. An agent-based algorithm for detecting community structure in networks, 2004.