University of Padua

Department of Mathematics Tullio Levi-Civita

Master Thesis in Data Science

Direct Search Methods for Influence Maximization problems

Supervisor
Prof. Francesco Rinaldi
University of Padua

Co-supervisor
PhD Sara Venturini
University of Padua

Master Candidate
Matteo Bergamaschi

Student ID
2041591

Academic Year
2022-2023

"Philosophy is written in this grand book - I mean the universe - which stands continually open to our gaze, but it cannot be understood unless one first learns to comprehend the language in which it is written. It is written in the language of mathematics, and its characters are triangles, circles, and other geometric figures, without which it is humanly impossible to understand a single word of it; without these, one is wandering about in a dark labyrinth."

— Galileo Galilei

Abstract

Networks serve as powerful tools for capturing interdependencies, resource flows, and the dynamics of influence within systems. Analyzing network properties such as connectivity and centrality provides insights into how information, resources, or influence propagate. Networks aid in uncovering patterns, identifying key nodes, and predicting the spread of information, diseases, or innovation. Among the many network analysis challenges, the Influence Maximization problem stands out. The Influence Maximization problem focuses on identifying a subset of individuals within a network. Targeting this subset with specific actions or information is expected to maximize the overall influence or reach of these actions throughout the network. This problem holds significant implications for domains like marketing, public health, social media, and viral marketing campaigns. By addressing the Influence Maximization problem, organizations gain the ability to strategically identify and engage with influential individuals or nodes. This facilitates the promotion of products, ideas, or initiatives effectively, requiring a deep understanding of information diffusion dynamics and leveraging the underlying network structures. In this thesis, we present two novel direct search methods: Neighbors Search and Nonmonotone Neighbors Search. These methods are designed to harness network structures, significantly improving influence maximization efficiency. We conduct comprehensive evaluations on diverse network types, providing valuable insights and practical solutions for organizations seeking to strategically identify and engage influential nodes for targeted initiatives. Our work not only advances the understanding of information diffusion dynamics within networks but also enhances the optimization of influence spread, offering actionable solutions for real-world applications.

Contents

Listing of acronyms

IC ..... Information Cascade
LT ..... Linear Threshold
IM ..... Influence Maximization
EIC ..... Extended Independent Cascade
ELT ..... Extended Linear Threshold
MLT ..... Multivalued Linear Threshold
GIP ..... General Information Propagation
MINLP ..... Mixed-integer Nonlinear Programming
DFM ..... Derivative-free Method
MV ..... Mixed Variables
CDS ..... Customized Direct Search
NS ..... Neighbors Search
NMNS ..... Nonmonotone Neighbors Search
RS ..... Random Search

1 Introduction

The Influence Maximization (IM) problem is a challenging optimization task focused on maximizing the influence function within a network structure. Due to its inherently combinatorial nature, this problem has yet to find a fully satisfying solution. Currently, direct search methods represent the state-of-the-art approach, although they are still considered too slow for large-scale networks. In this thesis, we propose innovative direct search methods that incorporate the network's structure into their operation. These methods are rigorously compared through experiments conducted on a variety of artificial networks.

Networks are integral to understanding the intricate interdependencies and interactions among entities. They offer insights into information flow, resource allocation, and influence propagation within complex systems. Analyzing network properties, such as connectivity and centrality, aids in identifying key nodes and predicting the spread of information, diseases, or innovations. This is particularly relevant to the IM problem.

The IM problem revolves around identifying a subset of individuals within a network. Targeting this subset with specific actions or information aims to maximize the overall influence or spread of these actions or information throughout the entire network. This problem holds immense significance in various domains, including marketing, public health, social media, and viral marketing campaigns.

By effectively solving the IM problem, organizations can strategically pinpoint influential nodes to promote their products, ideas, or initiatives. It involves a deep understanding of information diffusion dynamics and the strategic leveraging of network structures to optimize influence spread.

At the present time, the leading methods for tackling the IM problem are direct search methods. In this thesis, we begin by introducing and implementing the Customized Direct Search (CDS) method, which stands as one of the most established direct search methods specially designed for addressing the IM problem. However, it is worth noting that the field of direct search methods in this context still holds vast unexplored potential, with many avenues for improvement, as evidenced by recent developments in the literature.

The primary contribution of this thesis lies in the creation and implementation of two entirely novel direct search methods, both built upon the foundation of the CDS method. These new algorithms are engineered to

significantly enhance their capability to account for the structural characteristics of the network. This ability is notably absent in the earlier mentioned algorithm, and it represents a crucial advancement in our pursuit of solving the IM problem more efficiently.

Chapter 2 serves as the theoretical foundation of our work. We delve into crucial theoretical concepts essential for IM problem-solving. Drawing insights from prior research [1], we introduce the Independent Cascade (IC) model and the Linear Threshold (LT) model. Building upon these foundational models, we present the General Information Propagation (GIP) model, which unifies elements from IC and LT. Throughout this thesis, the GIP model becomes our primary information propagation model. We then introduce an initial formulation of the IM problem, the algorithm for calculating influence propagation, and lay the groundwork for exploring IM problems. Pertinent theoretical results are established, including an alternative formulation valuable for practical optimization methods.

Chapter 3 transitions into methodological development. Here, we detail the structure of three key methods essential to this thesis: the CDS method, previously implemented in [1], and two novel methods, NS and NMNS, developed by us for the first time.

In Chapter 4, we introduce comprehensive implementations essential for the rigorous evaluation of the methods within the context of the IM problem. Our focus narrows on graph structures, and all implementations are conducted using the Python programming language. The choice of Google Colab as the testing environment is made due to the substantial computational demands inherent to such problems.

Chapter 5 is dedicated to empirical experimentation. We begin by conducting a concise numerical analysis of the computational cost associated with the GIP model function call, a critical aspect as this function underpins the operation of each method. Subsequently, we introduce a set of artificially generated graphs that form the basis of our experiments. Before diving into empirical evaluations, we explore the concept of data profiles, a pivotal metric for comparing various derivative-free optimization methods. Finally, we present the empirical results of our experiments, including a comparative assessment of the methods under varying budget parameters, a critical determinant of IM problem complexity.

2
Related works

In this chapter, we embark on an exploration of the theoretical concepts crucial to the development of methods for solving the Influence Maximization problem. Many of these theoretical foundations are drawn from the insights provided in [1], and we refer to that for more comprehensive explanations of these theoretical foundations.

Section 2.1 serves as our starting point, where we introduce the two classical models: the Independent Cascade model and the Linear Threshold model. Extending upon these foundational models, we introduce a novel framework known as the General Information Propagation model in the subsequent section. The GIP model incorporates elements of both the IC and LT models, and we elucidate its unifying properties. Throughout this thesis, we adopt the GIP model as the primary information propagation model.

In Section 2.2, we present an initial formulation of the IM problem. This section not only introduces the algorithm for calculating influence propagation, a pivotal concept for the entirety of our subsequent work but also lays the groundwork for our exploration of the IM problem. Subsequently, after establishing a set of pertinent theoretical results, we unveil an equivalent formulation of the IM problem. This alternative formulation proves highly valuable in the practical implementation of optimization methods.

2.1 Information propagation models

2.1.1 The classic models

We first describe two classic information propagation models, the IC model and the LT model, in more detail in [2] [3].

Throughout our study, we focus on a social network, denoted as $G(V, E)$, which is directed, connected, and weighted. Here, $V = v_1, v_2, ..., v_n$ represents the set of nodes, and $E = \{(v_i, v_j)$: there exists a directed edge from node v_i to node $v_j\}$ represents the set of edges. Each edge (v_i, v_j) acts as a conduit for information flow between

nodes, with a corresponding weight $W_{ij} > 0$ signifying the strength or level of trust between these agents. When an edge (v_i, v_j) does not exist in E, its weight is set to 0. In this context, we employ the notation $x_i(t) \in \{0, 1\}$ to denote the state of node v_i at discrete time step t, where 0 represents inactivity and 1 signifies activity. It is important to note that in both the IC and LT models, nodes can transition from an inactive state to an active one $(0 \to 1)$, but not the other way around, establishing a progressive propagation process. As a result, feedback loops are absent; a node cannot be influenced by others it has previously influenced.

In the case of the LT model, further specifications are required. Specifically, it is mandated that $W_{ij} \leq 1$ holds true for all $v_j \in V$. Additionally, each node v_j randomly selects a threshold θ_j from the uniform distribution within the range $[0, 1]$. This threshold θ_j delineates the minimum cumulative influence weight required from active neighbors for node v_j to activate. Given these randomly chosen thresholds and an initial set of active nodes denoted as A_0, the propagation process unfolds in a deterministic manner through discrete time steps. At each time step $t > 0$, all nodes that were active in the previous step $(t-1)$ remain active. An inactive node v_j transitions to an active state if the sum of influence weights from its active neighbors surpasses its threshold θ_j, expressed as:

$$\sum_{v_i \in A_{t-1}} W_{ij} \geq \theta_j. \tag{2.1}$$

In contrast, in the IC model, each value of W_{ij} represents the probability that an active node v_i can influence an inactive neighbor v_j within a single step. Moreover, each node possesses only one opportunity to influence others when it initially becomes active. Should node v_j be successfully activated, it assumes a value of 1 in the subsequent time step. However, whether or not node v_i succeeds in influencing others, it cannot make further attempts to activate additional nodes in the following rounds. Variants of both cascade and threshold models, including those incorporating feedback, have been investigated [2] [4].

In these classic models, the spread of influence is defined as the count of active nodes when the process concludes, formally expressed as $\lim_{t\to\infty} \sum_{i=1}^{n} x_i(t)$. Consequently, the IM problem aims to maximize this spread while adhering to a constraint on the initial number of nodes that can be activated, denoted as $|A_0|$. Solving the IM problem under both the IC and LT models is known to be NP-hard. Notably, the field witnessed a pivotal algorithmic advancement with the introduction of approximation guarantees for the greedy hill-climbing algorithms. Numerous subsequent methods have been proposed to enhance the efficiency of these greedy algorithms while preserving the same approximation guarantees [5] [6], albeit not always exactly [7] [8].

It is worth noting that although models incorporating continuous variables and feedback between nodes exist within the realm of opinion dynamics, the associated IM problem is not a central focus in that context. Additionally, while continuous models such as the fully linear models have been analyzed within the scope of IM, their well-established connections with classic models and their underlying mechanisms remain less defined. Furthermore, variants of the constraint in the IM problem emerge when nodes assume continuous values [9], such as constraints based on the sum of initial state values rather than the count of activated nodes. Nevertheless, our primary interest in this paper lies in scenarios where, for instance, companies possess finite resources to persuade individuals to adopt products, thereby retaining the original constraint.

2.1.2 Extending the classic models

We commence by extending the two classic models, namely the IC model and the LT model, to operate within a continuous state space and a deterministic context. In this extension, we employ a continuous variable $x_j(t) \in \mathbb{R}$ to denote the state value of node v_j at each discrete time step $t \geq 0$. This variable can be understood as representing the influence on node v_j at time t. By incorporating continuous variables, we assume that influence is additive; for instance, people may become more persuaded by news if more friends believe it, or they may purchase more products if more friends make purchases, either at each time step or over time. Consequently, a node v_j is deemed influenced or active at time step t when $x_j(t) > 0$. We characterize the overall influence on each node v_j as:

$$s_j = \sum_{t=1}^{\infty} (1-\gamma)^t x_j(t), \tag{2.2}$$

where $\gamma \in [0, 1)$ is a time-discounting factor ensuring convergence. The vector $x(t) = (x_j(t))$ represents the state values.

Starting with the IC model, we make two key assumptions: (i) the expected value equals the actual influence on each node at each time step, and (ii) the state values exhibit the no-memory property. This property implies that a node's ability to either be influenced or influence others in the current time step t is independent of its previous states. Consequently, $x_j(t) = \sum_i W_{ij} x_i(t-1)$ holds for all $v_j \in V$ and $t > 0$. As a result, we introduce the Extended Independent Cascade (EIC) model, characterized by the updating function:

$$x(t) = W^T x(t-1), \quad \text{for all } t > 0, \tag{2.3}$$

where W is the weighted adjacency matrix of the network. It is worth noting that this model falls under the category of linear dynamics on networks. To ensure the convergence of overall influence, the following condition on the time-discounting factor γ and the spectral radius $\rho(W)$ is needed:

$$\gamma > 1 - \frac{1}{\rho(W)}. \tag{2.4}$$

Moving on to the LT model extension, we maintain the linear activation strategy akin to equation (2.1), where the linear product of the state values of each node's neighbors and the edge weights is computed. With the introduction of continuous variables, we set the activated state value to be the threshold value, effectively controlling the source of nonlinearity as activation. Importantly, the state values can vary in magnitude over time, allowing us to impose time-dependent thresholds $\{\theta_{j,t}\}$. Additionally, we continue to assume the no-memory property. Consequently, the Extended Linear Threshold (ELT) model takes the form:

$$x_j(t) = \begin{cases} \theta_{j,t}, & \sum_i W_{ij} x_i(t-1) \geq \theta_{j,t}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

for all $t > 0$ and $v_j \in V$.

From another perspective in extending the LT model, we can retain the magnitude of the state value at each time step. However, instead of a single binary value (1 for active), each node v_j can adopt values within a range $[1, m_j]$ based on the strength of influence attempts from its neighbors, $\sum_i W_{ij} x_i(t-1)$. To elaborate, akin to

Equation (2.1), a node v_j commences adopting a positive state value if the cumulative sum surpasses a threshold l'_j. Furthermore, the state value escalates from 1 to the maximum state value m_j as the sum increases from l'_j to a higher threshold h'_j. This augmentation constitutes a more direct extension of the LT model, termed the Multivalued Linear Threshold (MLT) model. Specifically, it features the updating function:

$$x_j(t) = f_j\left[\sum_i W_{ij}x_i(t-1)\right], \quad \forall t > 0, \quad v_j \in V, \tag{2.6}$$

where the function $f_j(x)$ is defined as follows:

$$f_j(x) = \begin{cases} 0, & \text{if } x < l'_j, \\ \frac{m_j-1}{h'_j-l'_j}(x-l'_j)+1, & \text{if } l'_j \le x < h'_j, \\ m_j, & \text{if } x \ge h'_j. \end{cases} \tag{2.7}$$

Here, the time-independent bound function $f_j(x)$ models the range within which v_j can take values. Moreover, $x_j(0) \in \{0\} \cup [1, h'_{j,0}]$, with $h'_{j,0}$ signifying the upper bound of the initial state value for node v_j. It is worth noting that we will later demonstrate the equivalence of these two extensions to the LT model through their relationships with the model we will subsequently introduce.

2.1.3 General class of information propagation model

In this section, we introduce a comprehensive model called the GIP model, which unifies the fundamental mechanisms underlying the two classic propagation models. This model encompasses two primary aspects:

(i) Each node v_i independently attempts to influence its neighbors, proportional to the edge weight and its current state value $x_i(t)$. This characteristic aligns with the principles of the IC model.

(ii) The actual impact on each node v_j stems from the collective behavior of its entire neighborhood. This effect is achieved by applying a nonlinear transformation to

$$y_j(t) = \sum_i W_{ij}x_i(t-1), \tag{2.8}$$

enabling us to capture how the cumulative influence attempts from all neighbors translate into a state change for node v_j. This concept draws parallels not only with the LT model but also with nonlinear models applied to opinion dynamics. Specifically, at each time step $t > 0$, we introduce a lower bound $l_{j,t}$ representing the threshold required to initiate propagation. This implies that $x_j(t) = 0$ if $y_j(t) < l_{j,t}$. Additionally, an upper bound $h_{j,t}$ accounts for the saturation effect, signifying that $x_j(t) = h_{j,t}$ if $y_j(t) \ge h_{j,t}$. In essence, the GIP model can be described as a bounded-linear dynamic system:

$$x_j(t) = f_{j,t}\left[\sum_i W_{ij}x_i(t-1)\right], \quad \text{for all } t > 0,\ v_j \in V, \tag{2.9}$$

where $f_{j,t}(x)$ takes the form:

$$f_{j,t}(x) = \begin{cases} 0, & x < l_{j,t}, \\ x, & l_{j,t} \le x < h_{j,t}, \\ h_{j,t}, & x \ge h_{j,t}, \end{cases} \tag{2.10}$$

representing the time-dependent bounds for each node v_j. The matrix $W = (W_{ij})$ with $W_{ij} \ge 0$ is the weighted adjacency matrix of the network, and $\{l_{j,t}\}$ and $\{h_{j,t}\}$ denote the time-dependent lower and upper bounds for each node v_j respectively, where $0 \le l_{j,t} \le h_{j,t}$. These bound values offer flexibility in characterizing the underlying population. Furthermore, it is worth noting that the GIP model can replicate the classic models by setting specific bound values. The initial states $x(0)$ are predefined, with $x_j(0) \in \{0\} \cup [l_{j,0}, h_{j,0}]$ and $l_{j,0} > 0$.

In conclusion, these extensions establish a continuous state space and deterministic foundation for the IC and LT models, enriching the representation and examination of influence dynamics within networks.

2.1.4 Unifying Features of the GIP Model

In this section, we elucidate the unifying aspect of the GIP model, which encompasses the EIC and ELT models as distinctive limiting scenarios. At one end of the spectrum, the GIP model aligns with the EIC model when all upper bounds are adequately large, while simultaneously ensuring that all lower bounds are sufficiently small.

Lemma 1 If $l_{j,t} = l_{\min,0} w^t \le h_{j,t}, \forall t > 0, v_j \in V$, where $l_{\min,0} = \min_j l_{j,0}$ and $w = \min_{i,j:W_{ij}>0} W_{i,j}$, in the GIP model, then there is no threshold effect from the lower bounds, $\forall t > 0, v_j \in V$, s.t. $\sum_i W_{ij} x_i(t-1) > 0$,

$$\sum_i W_{ij} x_i(t-1) > l_{j,t}. \tag{2.11}$$

Theorem 1 If $l_{j,t} \le l_{\min,0} w^t \le h_0^t W^t_{:,j} \le h_{j,t}, \forall t > 0, v_j \in V$, where $h_0 = (h_{j,0})$, $W^t_{:,j}$ is the j-th column of W_t, and $l_{\min,0}$, w are the same as in Lemma 1, the GIP model is equivalent to the EIC model.

Conversely, at the other end of the spectrum, the GIP model aligns with the ELT model when upper bounds are set equal to their corresponding lower bounds, i.e., $l_{j,t} = h_{j,t} = \theta_{j,t}$ for all $t > 0$ and $v_j \in V$. To delineate this relationship more explicitly, we introduce the notion of time-independent upper and lower bound thresholds, $\theta_{l,j}$ and $\theta_{h,j}$, similar to those found in the classic LT model. We propose the following threshold-type bounds for $t > 0$ and $v_j \in V$:

$$l_{j,t} = (\theta_{l,j}\alpha)^t l_{j,0} h_{j,t} = \theta_{h,j} \theta_{l,j}^{t-1} \alpha^t h_{j,0} \tag{2.12}$$

The evolution of these bounds hinges on varying powers of the mean weight $\alpha = \frac{1}{|E|} \sum_{(v_i,v_j)\in E} W_{ij}$ and $\theta_{l,j}$. Consequently, under this condition, the GIP model closely mirrors the ELT model when $l_{j,0} = h_{j,0}$ and $\theta_{l,j} = \theta_{h,j}$ for all $v_j \in V$.

Furthermore, leveraging these threshold-type bounds, the GIP model can also exhibit equivalence with the MLT model. Specifically, if uniform thresholds are assigned to the threshold-type bounds for all $v_j \in V$, such

as $\theta_{l,j} = \theta_l$ and $\theta_{h,j} = \theta_h$, the GIP model (with $l_{j,0} = 1$ for all $v_j \in V$) aligns with the MLT model. This correspondence is marked by $l_j = \theta_l\alpha$, $h_j = \theta_h\alpha h_{j,0}$, $m_j = \frac{\theta_h h_{j,0}}{\theta_l}$, and $h_{j,0} = h_{j,0}$ in terms of the overall influence. The relationship between time-discounting factors for the GIP model and the MLT model can be expressed as $\gamma' = 1 - (1-\gamma)\theta_l\alpha$. Consequently, if we denote the state values from the GIP model as $x_j(t)$ and those from the MLT model as $x'_j(t)$, then $x_j(t) = (\theta_l\alpha)^t x'_j(t)$ for all $t > 0$ and $v_j \in V$.

Moreover, when the network possesses a uniform weight α, setting $h_j = l_j = \theta_l\alpha$ in the MLT model is tantamount to stipulating that θ_l neighbors must exhibit positive state values for activations. This resembles the concept of the constant threshold model . This equivalence extends to both the GIP model and the ELT model. Notably, when $\theta_l = \theta_h = 1$, it signifies simple contagion, wherein a single active neighbor can influence a node. Conversely, when $\theta_l = \theta_h > 1$, it characterizes complex contagion, necessitating collective effort from the neighborhood to influence a node. Given the elucidation of these relationships, we will exclusively focus on threshold-type bounds in our subsequent discussions.

2.2 Influence Maximization

2.2.1 Introduction

Now we transition into a crucial algorithmic challenge closely tied to information propagation: the Influence Maximization problem. This problem revolves around the objective of maximizing the collective impact on network nodes at the culmination of the propagation process. Our particular focus lies on an essential constraint: the initiation of a finite number of nodes, determined by a budget size. This constraint echoes the notion of allocating limited resources to influence a wider audience.

Given a defined information propagation process and an associated function $s_j(\cdot)$ gauging the overall influence on each node v_j, the cumulative influence across the network emerges naturally as:

$$s(x(0)) = \sum_j s_j(x(0)), \tag{2.13}$$

where $x(0)$ represents the initial state vector. This function $s_j(\cdot)$ can encapsulate various elements, including nodes' states at the process's culmination ($\lim_{t\to\infty} x_j(t)$), thus encompassing both classic models and our proposed GIP model from previous section. Consequently, the IM problem revolves around maximizing $s(x(0))$, while ensuring adherence to the constraint of activating a limited number of nodes:

$$|\{v_j : x_j(0) > 0\}| \leq k, \tag{2.14}$$

where $k \in \mathbb{Z}^+$ represents the budget size.

Based on the objective function (2.13) and constraint (2.14), we formulate the IM problem as a mixed-integer

nonlinear programming (MINLP) task:

$$\begin{aligned} \max_{x,z} \; & s(x) \\ \text{s.t.} \quad & x_j \le h_{j,0} z_j \\ & x_j \ge l_{j,0} z_j \\ & \sum_j z_j \le k \\ & x_j \in \mathbb{R}, \quad z_j \in \{0,1\}, \quad \forall j, \end{aligned} \tag{2.15}$$

where $0 < l_{j,0} \le h_{j,0}$ define the initial influence level of node v_j, and $k \in \mathbb{Z}^+$ represents the budget size. The objective function $s(\cdot)$ measures the aggregate influence across the network, as illustrated in (2.13). The vector x comprises initial state values, while vector z, of the same dimensions, signifies the decision to set positive initial state values ($z_j = 1$) or not ($z_j = 0$), thus enforcing the constraint (2.14).

The intricacy of this optimization problem arises from the nature of the objective function $s(x)$. For instance, in the case of the GIP model along with function (2.2) for individual influence:

(i) $s(x)$ might lack smoothness or even be discontinuous due to potential nonsmoothness of $f_{j,t}(x)$ in (2.9) at $h_{j,t}$ and discontinuity at $l_{j,t}$.

(ii) In general, a closed-form expression for $s(x)$ is elusive, except when $f_{j,t}(x) = x$ for all $t > 0$ and $v_j \in V$ in (2.9).

(iii) Derivative information is often unhelpful in locating the maximum point. Nonetheless, even in the latter scenario, evaluating the objective function can be efficiently solved, as shown in Theorem 2. This efficiency becomes especially beneficial in the deterministic setting.

Thus, it becomes imperative to treat the objective function as a black-box system, prompting the use of Derivative-Free Methods (DFMs) for general solutions, an aspect we delve into in Chapter 3.

Theorem 2 Given a network $G(V, E)$ with weight matrix W and initial state $x(0)$, along with the GIP model governing the information propagation process and Equation (2.2) as the function for individual influence, computing the objective function $s(x(0))$ in the MINLP (2.15) can be achieved in $O(|E|t)$ time. Here, t denotes the number of time steps needed for convergence with a specified tolerance $\varepsilon > 0$, ensuring $|(1-\gamma)^t x(t)| < \varepsilon$ for all $t \ge t$.

Proof. The time complexity follows from Algorithm 2.1. In each iteration t, each nonzero element of weight matrix W has only one opportunity to be used to potentially adjust state value $x(t)$. There exist $O(|E|)$ such elements in total. Thus, the time complexity of each iteration is $O(|E|)$, resulting in an overall evaluation time complexity of $O(|E|t)$, contingent on the number of steps towards convergence, t.

2.2.2 General solution methods

There are two main classes of methods in DFMs: model-based methods and direct-search methods. Since we cannot assume that the objective function falls into a simple family, such as polynomials, model-based methods are not appropriate for this problem. Among the direct-search algorithms, the Mesh Adaptive Direct Search

Algorithm 2.1 Propagation Algorithm for the GIP Model

Require: A network $G(V, E)$ with weight matrix W where $W_{ij} > 0$ if $(v_i, v_j) \in E$, parameters $\{l_{j,t}, h_{j,t}\}$ in the GIP model, time-discounting factor γ, initial state $x(0)$ where $x_j(0) \in [l_{j,0}, h_{j,0}]$ if and only if $v_j \in A_0$ (0 otherwise), and the tolerance ε.

Ensure: The value of the objective function in the MINLP, s.

Set $t \leftarrow 0$, $x(0) \leftarrow x(0)$, and $s \leftarrow 0$.
Mark all out-neighbors of A_0 as potentially activated nodes, $N_0 \leftarrow \bigcup_{v_j \in A_0} N_{\text{out}}(v_j)$.
while $|(1-\gamma)^t x^t| > \varepsilon$
  $A_{t+1}, N_{t+1} \leftarrow \varnothing$, and $x(t+1) \leftarrow 0$.
  for each potentially activated node $v_j \in N_t$
    $x_j^{t+1} \leftarrow f_{j,t}\left(\sum_{v_i \in A_t} W_{ij} x(t)_i\right)$.
    if $x(t+1)_j > 0$
      $A_{t+1} \leftarrow A_{t+1} \cup \{v_j\}$.
      $N_{t+1} \leftarrow N_{t+1} \cup N^{\text{out}}(v_j)$.
      $s \leftarrow s + (1-\gamma)^{t+1} x_j^{t+1}$.
    end if
  end for
  $t \leftarrow t + 1$.
end while

(MADS) [10] method is one of the few that has local convergence analysis even when the objective function is not necessarily Lipschitz continuous. Therefore, we consider MADS for Mixed Variables (MV) as a general solution to the IM problem, which can be implemented using software like NOMAD [11] [12].

To understand local convergence, we introduce the crucial notion of local optimality for mixed variables, and subsequently, the concept of the local neighborhood. We partition each vector into its continuous and discrete components, denoted as $y = (y^c, y^d) \in \mathbb{R}^n$, where n is the dimension of the domain. For the MINLP (2.15), $y^c = x$ and $y^d = z$. For the continuous variables with maximum dimension n^c, the neighborhood is well-defined as the open ball $B_\varepsilon(y^c) = \{y_1^c \in \mathbb{R}^{n^c} : \|y_1^c - y^c\| < \varepsilon\}$ with $\varepsilon > 0$. However, there are different ways to define the discrete neighborhood.

A common choice for integer variables is $N(y) = \{y_1 \in \Omega : y_1^c = y^c, \|y_1^d - y^d\| \leq 1\}$. With a user-defined discrete neighborhood, the classical definition of local optimality can be extended to mixed variable domains as follows.

Definition 1 A point $y = (y^c; y^d) \in \Omega$ is considered a local maximizer of a function f on Ω with respect to the set of neighbors $N(y) \subset \Omega$ if there exists $\varepsilon > 0$ such that $f(y) \geq f(y_2)$ for all y_2 in the set

$$\Omega \cap \left(\bigcup_{y_1 \in N(y)} B_\varepsilon(y_1^d) \times (y_1^d) \right), \tag{2.16}$$

were $B_\varepsilon(y^c) = \{y_1^c \in \mathbb{R}^{n^c} : \|y_1^c - y^c\| < \varepsilon\}$ is an open ball, and $N(y)$ is a user-defined discrete neighborhood.

As mentioned earlier, MADS is one of the algorithms that can relax the assumptions for convergence analysis to include discontinuous functions. To conclude the overview of applicable DFMs, it is worth noting that there are also many heuristic algorithms [13], but they lack theoretical performance guarantees.

2.2.3 Special Cases

In the preceding sections, we extensively analyzed the general features of the IM problem and provided corresponding general solution methods. Now, we shift our focus to the IM problem with the GIP model governing the information propagation process and Equation (2.2) serving as the function for individual influence. In this section, we explore two distinct special cases of the GIP model to illuminate potential insights for even more general scenarios.

The first special case arises when the lower bounds $\{l_{j,t}\}$ are significantly small in the GIP model. In this situation, we demonstrate that the objective function becomes both continuous and concave concerning the continuous variables x, as stated in Theorem 3. Consequently, any local maximum attained also becomes a global maximum. As a result, the MADS method exhibits global convergence in this case, although it pertains solely to the continuous variables, considering that the optimality of the integer part remains local, as per Definition 1.

Theorem 3 If $\{l_{j,t}\}$ and $\{h_{j,t}\}$ are in accordance with Lemma 1, then the objective function $s(\cdot)$ in the MINLP 2.15 is continuous and concave with respect to the continuous variables x.

The other distinctive scenario emerges when not only $\{l_{j,t}\}$ but also $\{h_{j,t}\}$ are sufficiently large in the GIP model, essentially representing the extreme of the EIC model. In this context, $x(t) = W^T x(t-1) = (W^T)^t x(0)$. Consequently, the objective function takes the form:

$$s(x(0)) = \sum_{t=1}^{\infty}(1-\gamma)^t x_j(t) = \sum_{t=1}^{\infty}(1-\gamma)^t (W^T)^t x(0) = c^T x(0), \tag{2.17}$$

where $c = \{[I-(1-\gamma)W]^{-1} - I\}1$ embodies the Katz centrality with a factor of $(1-\gamma)$, I denotes the identity matrix, and the penultimate equation is established under the condition (2.4) within this extreme scenario. Therefore, the objective function becomes linear, thus Lipschitz continuous, concave, and smooth. The exact solution(s) for this case, as elucidated in Theorem 4, can be achieved.

Theorem 4 If $\{l_{j,t}\}$ and $\{h_{j,t}\}$ follow the conditions outlined in Theorem 1, then the exact solution(s) to the MINLP (2.15) is given by:

$$x_j^* = \begin{cases} h_{j,0}, & \text{if } j \in A, \\ 0, & \text{otherwise,} \end{cases} \quad z_j^* = \begin{cases} 1, & \text{if } j \in A, \\ 0, & \text{otherwise,} \end{cases} \tag{2.18}$$

where $A = \{j_1, \ldots, j_k\}$ such that $h_{i,0}c_i \geq h_{j,0}c_j$ for all $i \notin A$ and $j \in A$, $c = \{[I-(1-\gamma)W]^{-1} - I\}1$ is the Katz centrality with factor $(1-\gamma)$, and the uniqueness of the solution relies on the uniqueness of set A.

Hence, in the scenario where the GIP model resides at the extreme of the EIC model (i.e., linear dynamics), the exact solution(s) entails activating k nodes with the highest product of their Katz centrality and their maximum

initial value. This connection aligns the IM problem with a widely studied network centrality measure, the Katz centrality. Additionally, this solution can serve as a promising starting point for the ensuing search algorithm for the MINLP (2.15). The search depth could potentially be proportional to the distance between the underlying propagation and linear dynamics.

2.2.4 Problem Formulation

Here we leverage a specific property of the objective function, namely the non-decreasing nature of $s(x)$, which is established in the proof of Theorem 4. Building upon this property, we introduce a customized direct search method tailored for the MINLP (2.15).

Due to this characteristic, the task of maximizing the objective $s(x)$ concerning both x and z in the MINLP (2.15) can be simplified to the task of maximizing $s(x)$ while x and z are set to their highest attainable values. Specifically, we set $x_j = h_{j,0} z_j$ and $\sum_j z_j = k$, effectively reducing the problem to the following form with respect to the binary vector z:

$$\begin{aligned} &\max_z s(h_0 \odot z) \\ \text{s.t.} \quad &\sum_j z_j = k, \\ &z_j \in \{0,1\}, \quad \forall j, \end{aligned} \tag{2.19}$$

where $h_0 = (h_{j,0})$ and $\odot$ denotes the element-wise Hadamard product. As a result, the domain Ω^d becomes a natural mesh for searching at each iteration r:

$$\mathcal{M}_r = \Omega^D = \{z \in \{0,1\}^n : \sum_j z_j = k\}. \tag{2.20}$$

The constraints are inherently embedded within the domain, and they are managed using the extreme barrier approach s_{Ω^d}, where $s_{\Omega^d}(z) = s(h_0 \odot z)$ if $z \in \Omega^d$ and $-\infty$ otherwise. We define the neighborhood function for binary variables z as:

$$N(z) = \{y \in \{0,1\}^n : \|y - z\|_1 \leq d\}, \tag{2.21}$$

where $d \in \mathbb{Z}^+ \setminus \{1\}$, as the L_1 distance between y and z can reach a minimum of 2 when $y \neq z$ and $y, z \in \Omega^d$. This minimal distance of 2 is achieved when only one element of value 1 is exchanged with an element of value 0.

3 Methods

3.1 Customized Direct Search

In this section we present the Customized Direct Search [1] algorithm tailored for the revised problem (2.19). The algorithm starts from an exact solution (as given in Theorem 4) when the GIP model is at the extreme of the EIC model. At each iteration r in the poll step, we explore the local neighborhood of the current candidate $z(r)$ until a point with a sufficient improvement in the objective value is found or all points have been examined. In the termination check, if an improved point is identified, the algorithm will revert to the optional search step, but will decrease the required improvement if a sufficiently improved point has not been discovered. If no improvement is found, the algorithm outputs the current iterate and terminates. Refer to Algorithm 3.1 for detailed steps.

Hence, the termination step directly guarantees local convergence by Definition 1. While global convergence could be achieved through a carefully developed search step and a better understanding of the objective function's landscape to avoid poor local optima, the drawback of global methods is their time complexity. Thus, we maintain the search step as an optional component. The CDS method takes advantage of the problem's characteristics and mitigates worst-case complexity by initializing with an exact solution when the GIP model reaches the extreme of the EIC model. It is conjectured that the local optima near this specific solution are sufficiently favorable.

From the current CDS method, two avenues for enhancing output quality emerge. The problem is equivalent to selecting a set of nodes with values 1 (and others with 0). Hence, two known methods for global convergence can be applied: (i) brute force, where all node sets of size k are evaluated to select the optimal one, and (ii) random sampling, which exhibits asymptotic global convergence if it samples densely enough. The two improvement dimensions are inspired by these methods. Firstly, by expanding the neighborhood definition, more points are searched within the domain. If the neighborhood encompasses the entire domain, it resembles the brute-force method. Secondly, the search process (steps 2, 3, and 4 in Algorithm 3.1) can be restarted from other unexplored points randomly, mirroring the logic of the search step. This strategy achieves asymptotic global convergence, akin

to the random sampling approach.

Algorithm 3.1 Customized Direct Search (CDS)

1: Initialization: Set $0 < \zeta, \delta < 1, d > 1$. Let $z(0) \in \Omega^d$ such that $z(0)_j = 1$ if node $j \in A = \{j_1, \ldots, j_k\}$ where $h_{i,0}c_i = h_{j,0}c_j, \forall i \notin A, j \in A$, and $c = \{[I-(1-\gamma)W]^{-1} - I\}1$ is the Katz centrality. Set iteration $r = 0$.

2: SEARCH step (optional): Evaluate s_{Ω^d} on a finite subset of trial points on the mesh M_r, until a sufficiently improved mesh point z is found, where $s_{\Omega^d}(z) > (1+\zeta)s_{\Omega^d}(z^{(r)})$, or all points have been exhausted. If an improved point is found, then the SEARCH step may terminate, skip the next POLL step and go directly to step 4.

3: POLL step: Evaluate s_{Ω^d} on the set $\Omega^d \cap N(z^{(r)}) \subset M_r$ as in 2.21, with distance d, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > (1+\zeta)s(\Omega^d)(z^{(r)})$, or all points have been exhausted.

4: Termination check: If an improvement is found, set $z^{(r+1)}$ as the improved solution, while decreasing $\zeta \leftarrow \delta\zeta$ if a sufficient improvement has not been found, increment $r \leftarrow r+1$, and go to step 2. Otherwise, output the solution $z^{(r)}$.

3.2 Neighbors Search

In the realm of direct search methods, a central challenge revolves around the selection of an appropriate mesh, a pivotal factor that significantly influences how the solution space is explored. In the preceding section, we delved into the CDS method, which employs a rather straightforward mesh strategy. Specifically, it populates the mesh with all points located at a specified L_1 distance, commonly set at 2. Moreover, the CDS method incorporates the graph structure by utilizing the Katz centrality to guide the initial point selection. However, from our standpoint, these measures seemed insufficient. Hence, in this section, we introduce our first alternative approach: the Neighbor Search (NS) method.

Distinguishing itself from the CDS method, the NS method deviates primarily in its mesh formulation. This new methodology aims to better leverage the underlying graph structure during mesh construction. The core notion is centered around a more judicious selection of mesh points, emphasizing connections within the graph.

The fundamental premise of the NS method is to exclusively consider points that are interconnected through graph edges. Unlike the CDS method's inclusive approach of all points at a set L_1 distance, the NS method narrows down its focus to only those points that are directly linked through edges in the graph.

In summary, the NS method seeks to enhance the mesh construction process by incorporating the graph's topological structure more effectively. Unlike the CDS method, which indiscriminately covers a designated $L1$ distance, the NS method opts for a more graph-aware approach by exclusively considering neighboring points connected through edges. This approach aims to provide a refined exploration of the solution space, accounting for the inherent relationships within the underlying graph structure.

Specifically, the NS method operates within a more constrained mesh, leading to a reduction in the number of function evaluations required at each iteration. Importantly, this reduction in mesh size does not compromise the quality of the search in our view. This is due to the fact that the NS method hones in solely on the most significant and meaningful points within the solution space.

This approach ensures a more focused exploration by considering points that are directly linked in the graph, rather than exhaustively covering a broader L_1 distance as done by the CDS method. By deliberately selecting only interconnected points, the NS method strikes a balance between efficiency and search.

Given that this novel approach doesn't inherently ensure local convergence due to the unique nature of the refined mesh, we've taken this into careful consideration. In circumstances where a search within the smaller mesh fails, an additional search is conducted across the entire neighborhood, a procedure analogous to the one employed by the CDS method. This ensures a guarantee of local convergence, a critical aspect that we've incorporated.

It is conjectured that this supplementary step will be infrequently invoked throughout the entire search process. As a result, it is anticipated that this occasional occurrence won't significantly impact the overall speed of the algorithm. We'll delve into this hypothesis further in Chapter 5, where we'll see that initial observations align with this line of thinking.

In Algorithm 3.2, we delve into the inner workings of the NS method, highlighting its distinct SEARCH step, which sets it apart from the CDS method. It is worth noting that the remaining steps in this algorithm remain consistent with those found in the CDS method.

Algorithm 3.2 Neighbors Search (NS)

1: Initialization: Set $0 < \zeta, \delta < 1, d > 1$. Let $z(0) \in \Omega^d$ such that $z(0)_j = 1$ if node $j \in A = \{j_1, \ldots, j_k\}$ where $h_{i,0}c_i = h_{j,0}c_j, \forall i \notin A, j \in A$, and $c = \{[I-(1-\gamma)W]^{-1} - I\}1$ is the Katz centrality. Set iteration $r = 0$.

2: SEARCH step: Evaluate s_{Ω^d} on the set $\Omega^d \cap N(z^{(r)}) \cap C(z^{(r)}) \subset M_r$ as in 2.21 with distance d, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > (1+\zeta)s(\Omega^d)(z^{(r)})$, or all points have been exhausted. $C(z^{(r)})$ is the set of points that are connected by an arc with $z^{(r)}$. If an improved point is found, then the SEARCH step may terminate, skip the next Additional SEARCH step and go directly to step 4.

3: Additional SEARCH step: Evaluate s_{Ω^d} on the set $\Omega^d \cap N(z^{(r)}) \subset M_r$ as in 2.21 with distance d, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > (1+\zeta)s(\Omega^d)(z^{(r)})$, or all points have been exhausted.

4: Termination check: If an improvement is found, set $z^{(r+1)}$ as the improved solution, while decreasing $\zeta \leftarrow \delta\zeta$ if a sufficient improvement has not been found, increment $r \leftarrow r+1$, and go to step 2. Otherwise, output the solution $z^{(r)}$.

3.3 Nonmonotone Neighbors Search

The inclusion of a nonmonotone acceptance rule holds significant importance within the realm of optimization problems [14]. It represents as a valuable tool for enhancing algorithm performance. In contrast to monotone direct search methods, which exclusively seek new points that consistently reduce the objective function, nonmonotone approaches offer distinct advantages.

In scenarios where a monotone approach might fall short, nonmonotone methods come to the rescue. For instance, when dealing with an objective function characterized by steep-sided valleys, monotone searches may yield only small movements along the search directions. Similarly, when the objective function exhibits local "flatness", monotone searches might necessitate generating an excessive number of primitive directions to break away from a specific point.

Nonmonotone acceptance rules circumvent these limitations by allowing for more flexible exploration of the search space. Rather than rigidly adhering to strict reductions in the objective function, nonmonotone methods offer the versatility to consider a broader range of possibilities. This adaptability is particularly advantageous when dealing with complex landscapes and can significantly enhance the efficiency and effectiveness of optimization algorithms.

In this section, we introduce a novel method that builds upon the NS method approach, incorporating a nonmonotone acceptance rule, the Nonmonotone Neighbors Search (NMNS) method. Refer to Algorithm 3.3 for a comprehensive algorithm description. Here, we provide a concise overview of the method's functioning.

The initialization, SEARCH, and additional SEARCH steps closely resemble their counterparts in the NS method. However, a key distinction lies in the point acceptance rule. Instead of exclusively accepting points that exhibit improvement by a fixed margin, our method considers every point that surpasses a certain reference value f^{ref}. The reference value, denoted as f^{ref}, represents the minimum among the last n objective function values accepted by the algorithm. These values are stored in a set, which is managed using a first-in-first-out policy.

To further harness the potential of nonmonotonicity, we introduce an additional phase, known as the enriched SEARCH, positioned between the SEARCH and additional SEARCH steps. During this phase, the algorithm explores points within the neighborhood at a distance of $d + 2$. Notably, this neighborhood encompasses a substantially larger cardinality of points compared to other phases. To manage computational resources efficiently, we randomly select a limited number of points from this expanded set. Specifically, the number of points sampled during the enriched SEARCH phase is n_{dim} times the number of points selected in the previous phase.

During this phase, it is important to highlight that we include points that may also be candidates for the additional SEARCH step. There are two primary reasons for not excluding these points. Firstly, iterating through each of these points and verifying their membership in a stricter neighborhood imposes a significant computational burden. Secondly, given the difference in cardinality between the two neighborhoods, randomly selecting points inherently increases the likelihood of including points from the desired set. Moreover, when choosing points that will be considered by the additional SEARCH step, the buffer system effectively manages any additional function calls, making this approach more efficient overall.

Algorithm 3.3 Nonmonotone Neighbors Search (NMNS)

1: Initialization: Set $d > 1, n > 1, n_{\text{dim}} > 0$. Let $z(0) \in \Omega^d$ such that $z(0)_j = 1$ if node $j \in A = \{j_1, \ldots, j_k\}$ where $h_{i,0}c_i = h_{j,0}c_j, \forall i \notin A, j \in A$, and $c = \{[I-(1-\gamma)W]^{-1} - I\}1$ is the Katz centrality. Set iteration $r = 0$.

2: SEARCH step: Evaluate s_{Ω^d} on the set $\Omega^d \cap N(z^{(r)}) \cap C(z^{(r)}) \subset M_r$ as in 2.21, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > \min\limits_{t=r,..,r-n+1} s(\Omega^d)(z^{(t)})$, or all points have been exhausted. $C(z^{(r)})$ is the set of points that are connected by an arc with $z^{(r)}$. If an improved point is found, then the SEARCH step may terminate, and go directly to step 5.

3: Enriched SEARCH step: Evaluate s_{Ω^d} on $n \cdot |\Omega^d \cap N(z^{(r)}) \cap C(z^{(r)}|$ random points of the set $\Omega^d \cap N'(z^{(r)}) \subset M_r$ as in 2.21 with distance $d+2$, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > \min\limits_{t=t=r,..,r-n+1} s(\Omega^d)(z^{(t)})$, or all points have been exhausted. If an improved point is found, then the enriched SEARCH step may terminate, and go directly to step 5.

4: ADDITIONAL SEARCH step: Evaluate s_{Ω^d} on the set $\Omega^d \cap N(z^{(r)}) \subset M_r$ as in 2.21, until a sufficiently improved mesh point z is found, where $s(\Omega^d)(z) > \min\limits_{t=r,..,r-n+1} s(\Omega^d)(z^{(t)})$, or all points have been exhausted.

5: Termination check: If an improvement is found, set $z^{(r+1)}$ as the improved solution, increment $r \leftarrow r+1$, and go to step 2. Otherwise, output the solution $z^{(r)}$.

3.4 Other Methods

In this section, we introduce additional methodologies for experimentation and testing purposes.

These techniques encompass the Brute Force Solver, involving a meticulous exploration of every conceivable combination within the influence maximization problem. Although it is notably constrained in terms of efficiency and only suitable for small-scale scenarios, its utilization in preliminary testing provides valuable metrics regarding the quality of solutions generated by alternative direct search methods.

The second technique, the Random Search (RS) Solver (see, e.g., [15] and references therein) revolves around the random selection of points within the solution space. Subsequently, it identifies the best solution encountered after a predetermined number of function evaluations. This technique serves as a fundamental reference point throughout all tests, helping us identify instances where our direct search methodologies exhibit inferior performance compared to this random search. Such occurrences may indicate potential implementation issues or algorithmic errors requiring further investigation.

4

Implementation

In this chapter, we introduce the comprehensive implementations that have been developed to rigorously evaluate the previously presented methods in the context of the IM problem, with a specific focus on graph structures. All these implementations are conducted using the Python programming language. Moreover, owing to the substantial computational demands inherent to such problems, a deliberate choice was made to leverage the computational resources provided by Google Colab as the testing environment.

4.1 Propagation Model

We developed a function that computes the propagation algorithm as in 2.1. A snippet of the code is presented in Listing 4.2, offering a glimpse into the detailed implementation.

The `Influence_evaluation` function serves the purpose of estimating node influence in a graph iteratively. The core of the computation lies within the `GIP_function` call, where the bulk of influence calculations takes place. To enhance efficiency, a significant portion of the operations has been vectorized, optimizing the code for computational speed.

By iteratively updating influence estimates based on graph structure and predefined parameters, the function provides a comprehensive way to evaluate node influence over a series of iterations. The iterative nature ensures that influence accumulates through graph interactions, making it a valuable tool in studying and solving problems like the IM problem.

The `Influence_evaluation` function takes the following inputs:

- `g`: The input graph.
- `W`: A weight matrix.
- `x0`: The initial influence estimates.

- `params`: A tuple of parameters containing $l0$, $h0$, θ_l, θ_h, γ, and ε.
- `max_t`: Maximum number of iterations (default value is 999).

The function outputs three values:

- The total influence estimate s.
- A list `st` containing influence estimates at each iteration.
- A list `x` containing nodes' state at each iteration.

A pivotal aspect of this function lies within the subroutine `GIP_function`, responsible for implementing the GIP model as described in Eqaution 2.9. A glimpse into the code implementation can be found in Listing 4.1. In the course of our research endeavors, we further extended our work by incorporating the classical IC and LT models. This deliberate exploration aimed to discern and analyze the nuanced behavioral distinctions among these models.

```
1 def GIP_function(x,h,l):
2
3   r = np.multiply(x >= l, x)
4
5   return np.minimum(r, h)
```

Listing 4.1: GIP Function

```
def Influence_evaluation(g, W, x0, params, max_t = 999):

  l0, h0, theta_l, theta_h, gamma, eps = params

  s = 0

  st = [0]

  x = [x0]

  alpha = 0

  for u, v, d in g.edges(data=True):
    alpha += d["weight"]

  alpha = alpha/len(g.edges)

  t = 1

  while np.linalg.norm(x[-1]*((1-gamma)**t)) > eps and t <= max_t:
    l = ((theta_l*alpha)**t)*l0
    h = (theta_h*(theta_l**(t-1))*(alpha**t))*h0

    xt = GIP_function(np.sum(np.multiply(W,x[-1]), axis = 1), h, l)
    s += np.sum(xt)

    x.append(xt)

    st.append(s)
    t += 1

  return s, st, x
```

Listing 4.2: Influence Evaluation Function

4.2 Methods

4.2.1 Customized Direct Search

Here, we present a Python implementation of the CDS method, as introduced in Section 3.1. This implementation directly corresponds to the algorithm outlined in Algorithm 3.1. Additionally, we have introduced a buffer mechanism to enhance efficiency by avoiding redundant propagation calls on the same points.

The `CDS_solver` function accepts the following inputs:

- `g`: The input graph.
- `W`: A weight matrix.
- `x0`: The starting point of the algorithm.
- `params`: A tuple of parameters containing l_0, h_0, θ_l, θ_h, γ, and ε.
- `delta, xi, d`: Additional parameters specific to the CDS method.
- `max_calls`: The maximum number of propagation function calls allowed.
- `buffer_dim`: The dimension of the buffer used for optimization.

The function provides three output values:

- `s`: A list containing the influence scores at each iteration.
- `X`: A list of every iteration point.
- `history`: A history of the best value found along with the number of function calls required to find it.

In the listings below, we illustrate the distinct phases of the algorithm. In Listing 4.3, we showcase the CDS initialization step, where all parameters and variables are appropriately initialized. It is in this phase that the buffer is created. Subsequently, in Listing 4.4, we delve into the CDS POLL step, which constitutes the core of the computation. Lastly, in Listing 4.5, we present the termination check. Notably, if no improvements are made (lines 53-56), the algorithm concludes. Additionally, throughout each step of the method, we include lines that provide real-time updates on the algorithm's progress, estimating the time remaining based on the maximum number of allowed function calls.

```
def CDS_solver(g, W, x0, params, delta, xi, d, max_calls, buffer_dim):
    N = len(x0)
    K = np.count_nonzero(x0)
    X = [x0]
    s = [Influence_evaluation(g, W, x0, params)[0]]
    r = 0
    xi_t = xi

    stop = False
    buffer = collections.deque(maxlen=buffer_dim)
    calls = 1
    start = time.time()
    history = [[s[-1], calls]]

    print("\r" + "Custom direct search... {}/{}. ETA: {} s.".format(calls,
        max_calls, round((time.time()-start)*((max_calls-calls)/calls))),
        end = "")
```

Listing 4.3: CDS_solver Function Inizialization

```
    while (stop == False) and (r < 1000):
        idx = set(np.nonzero(X[-1])[0])
        neighbors = []
        idx_temp = set(range(N)) - set(idx)
        for i in range(d // 2):
            for elem1 in itertools.combinations(idx, len(idx) - 1 - i):
                for elem2 in itertools.combinations(idx_temp, i + 1):
                    neighbors.append(list(set(elem1) | set(elem2)))

        s_temp = s[-1]
        x_temp = X[-1]

        for elem in neighbors:
            x_elem = np.zeros(N)
            x_elem[list(elem)] = theta_l
            if list(x_elem) in buffer:
                continue
            if calls >= max_calls:
                stop = True
                break

            s_elem = Influence_evaluation(g, W, x_elem, params)[0]
            calls += 1
            print("\r" + "Custom direct search... {}/{}. ETA: {}
                s.".format(calls, max_calls,
                round((time.time()-start)*((max_calls-calls)/calls))), end =
                "")

            if s_elem > s_temp:
                history.append([s_elem, calls])
                x_temp = x_elem
                s_temp = s_elem
                if s_temp > (1 + xi_t) * s[-1]:
                    break
```

Listing 4.4: CDS_solver Function POLL step

```
49          X.append(x_temp)
50          s.append(s_temp)
51          r += 1
52
53          if s[-1] == s[-2]:
54              X.pop()
55              s.pop()
56              stop = True
57          elif s[-1] > (1 + xi_t) * s[-2]:
58              continue
59          else:
60              xi_t = xi_t * delta
61
62      history.append([s[-1], calls])
63
64      print("\r" + "Custom direct search... {}/{} Done!".format(calls,
            max_calls))
65      print("s: {}".format(s[-1]))
66
67      return s, X, history
```

Listing 4.5: CDS_solver Function Termination check and Output

4.2.2 Neighbors Search

Now, we present the implementation of the NS method, as introduced in Section 3.2. This marks the introduction of the first completely new algorithm in this paper. Similar to the previous `CDS` method, we have implemented a buffer system to enhance the method's efficiency.

The `NS_solver` function takes the following inputs:

- `g`: The input graph.
- `W`: A weight matrix.
- `x0`: The starting point of the algorithm.
- `params`: A tuple of parameters containing l_0, h_0, θ_l, θ_h, γ, and ε.
- `delta, xi, d`: Additional parameters specific to the NS method.
- `max_calls`: The maximum number of propagation function calls allowed.
- `buffer_dim`: The dimension of the buffer used for optimization.

The function provides three output values:

- `s`: A list containing the influence scores at each iteration.
- `X`: A list of every iteration point.
- `history`: A history of the best value found along with the number of function calls required to find it.

In the code listings provided below, we illustrate the distinct phases of the algorithm. In Listing 4.6, you can observe the initialization step of the NS method. During this phase, all parameters and variables are set to their appropriate initial values. It is noteworthy that this phase is identical to the corresponding one in the `CDS_solver` function. Next, in Listing 4.7, we delve into the NS SEARCH step. This part closely resembles the CDS POLL step, except for the neighborhood selection process, which is evident in lines 21-26. Moving forward, Listing 4.8 presents the additional SEARCH step. It bears a strong resemblance to the POLL step in the `CDS_solver` function. Finally, in Listing 4.9, you'll find the termination check. Similar to the `CDS_solver` function, real-time updates on the algorithm's progress are displayed.

```
def NS_solver(g, W, x0, params, delta, xi,d, max_calls, buffer_dim):
    N = len(x0)
    K = np.count_nonzero(x0)
    X = [x0]
    s = [Influence_evaluation(g, W, x0, params)[0]]
    r = 0
    xi_t = xi

    stop = False
    buffer = collections.deque(maxlen=buffer_dim)
    calls = 1
    start = time.time()
    history = [[s[-1],calls]]

    print("\r" + "Neighbors search... {}/{}. ETA: {} s.".format(calls,
        max_calls, round((time.time()-start)*((max_calls-calls)/calls))),
        end = "")
```

Listing 4.6: NS_solver Function Inizialization

```
    while (stop == False) and (r < 1000):
        idx = set(np.nonzero(X[-1])[0])
        neighbors = []

        for elem1 in idx:
            for elem2 in g.neighbors(elem1):
                temp = idx.copy()
                temp.add(elem2)
                temp.remove(elem1)
                neighbors.append(temp)

        s_temp = s[-1]
        x_temp = X[-1]

        for elem in neighbors:
            x_elem = np.zeros(N)
            x_elem[list(elem)]  = theta_l
            if list(x_elem) in buffer:
                continue
            if calls >= max_calls:
                stop = True
                break

            s_elem = Influence_evaluation(g, W, x_elem, params)[0]
            buffer.append(list(x_elem))
            calls +=1
            print("\r" + "Neighbors search... {}/{}. ETA: {}
                s.".format(calls, max_calls,
                round((time.time()-start)*((max_calls-calls)/calls))), end =
                "")

            if s_elem > s_temp:
                history.append([s_elem,calls])
                x_temp = x_elem
                s_temp = s_elem
```

```
                if s_temp > (1+xi_t)*s[-1]:
                    break

        X.append(x_temp)
        s.append(s_temp)
        r += 1

        if s[-1] > (1+xi_t)*s[-2]:
            continue

        elif s[-1] > s[-2]:
            xi_t = xi_t * delta

        else:
            X.pop()
            s.pop()
            idx = set(np.nonzero(X[-1])[0])
            neighbors = []
            idx_temp = set(range(N))-set(idx)
```

Listing 4.7: NS_solver Function SEARCH step

```
            for i in range(d//2):
                for elem1 in itertools.combinations(idx, len(idx)-1-i):
                    for elem2 in itertools.combinations(idx_temp, i+1):
                        neighbors.append(list(set(elem1) | set(elem2)))

            s_temp = s[-1]
            x_temp = X[-1]
            for elem in neighbors:
                x_elem = np.zeros(N)
                x_elem[list(elem)]  = theta_l
                if list(x_elem) in buffer:
                    continue
                if calls >= max_calls:
                    stop = True
                    break

                s_elem = Influence_evaluation(g, W, x_elem, params)[0]
                buffer.append(list(x_elem))
                calls += 1
                print("\r" + "Neighbors search... {}/{}. ETA: {}
                    s.".format(calls, max_calls,
                    round((time.time()-start)*((max_calls-calls)/calls))),
                    end = "")

                if s_elem > s_temp:
                    history.append([s_elem,calls])
                    x_temp = x_elem
                    s_temp = s_elem
                    if s_temp > (1+xi_t)*s[-1]:
                        break

            X.append(x_temp)
            s.append(s_temp)
            r += 1
```

Listing 4.8: NS_solver Function Additional SEARCH step

```
            if s[-1] == s[-2]:
                X.pop()
                s.pop()
                stop = True

            elif s[-1] > (1+xi_t)*s[-2]:
                continue

            else:
                xi_t = xi_t * delta

    history.append([s[-1], calls])

    print("\r" + "Neighbors search... {}/{} Done!".format(calls, max_calls))
    print("s: {}".format(s[-1]))

    return s, X, history
```

Listing 4.9: NS_solver Function Termination Check and Output

4.2.3 Nonmonotone Neighbors Search

Now, we present the implementation of the NMNS method, as introduced in Section 3.3. This marks the introduction of the second completely new algorithm in this paper. Similar to the previous methods, we have implemented a buffer system to enhance the method's efficiency.

The `NMNS_solver` function takes the following inputs:

- `g`: The input graph.
- `W`: A weight matrix.
- `x0`: The starting point of the algorithm.
- `params`: A tuple of parameters containing l_0, h_0, θ_l, θ_h, γ, and ε.
- `d, n, n_d`: Additional parameters specific to the NMNS method.
- `max_calls`: The maximum number of propagation function calls allowed.
- `buffer_dim`: The dimension of the buffer used for optimization.

The function provides three output values:

- `s`: A list containing the influence scores at each iteration.
- `X`: A list of every iteration point.
- `history`: A history of the best value found along with the number of function calls required to find it.

In the subsequent listings, we delve into the distinct phases of the algorithm. In Listing 4.10, we embark on the initialization step, where all parameters and variables are meticulously configured. Notably, in line 8, we initialize the queue that plays a pivotal role in implementing the nonmonotone acceptance rule. Despite this nuance, the code largely mirrors its counterpart in the `NS_solver` function. Moving forward, Listing 4.11 captures the SEARCH step, while Listing 4.13 elucidates the additional SEARCH step. This section of the code closely parallels the corresponding segments in the `NS_solver` function, with minor differentiations primarily stemming from alterations in the acceptance rule. Listing 4.12, on the other hand, offers a glimpse into the distinctive enriched SEARCH step. While akin to the additional SEARCH step, this phase stands apart due to its approach to generating the set of points for exploration, as highlighted in lines 63-69. To terminate the algorithm, Listing 4.14 presents the termination check. As with all other algorithms, real-time progress updates are thoughtfully incorporated.

```
def NM_NS_solver(g, W, x0, params,d, max_calls, buffer_dim, n = 4, n_dim =
    4):
    N = len(x0)
    K = np.count_nonzero(x0)
    X = [x0]
    s = [Influence_evaluation(g, W, x0, params)[0]]
    r = 0
    xi_t = xi
    queue = [s[-1]]
    s_out = s[-1]

    stop = False
    buffer = collections.deque(maxlen=buffer_dim)
    calls = 1
    start = time.time()
    history = [[s[-1], calls]]

    print("\r" + "Nonmonotone Neighbors search... {}/{}. ETA: {}
        s.".format(calls, max_calls,
        round((time.time()-start)*((max_calls-calls)/calls))), end = "")
```

Listing 4.10: NM_NS_solver Function Inizialization

```
    while (stop == False):
        idx = set(np.nonzero(X[-1])[0])
        neighbors = []
        found = False

        for elem1 in idx:
            for elem2 in g.neighbors(elem1):
                temp = idx.copy()
                temp.add(elem2)
                temp.remove(elem1)
                neighbors.append(temp)

        s_ref = min(queue)

        for elem in neighbors:
            x_elem = np.zeros(N)
            x_elem[list(elem)] = theta_l
            if list(x_elem) in buffer:
                continue
            if calls >= max_calls:
                stop = True
                break

            s_elem = Influence_evaluation(g, W, x_elem, params)[0]
            buffer.append(list(x_elem))
            calls +=1
            print("\r" + "Nonmonotone Neighbors search... {}/{}. ETA: {}
                s.".format(calls, max_calls,
                round((time.time()-start)*((max_calls-calls)/calls))), end =
                "")

            if s_elem > s_ref:
                queue = insert(queue, n, s_elem)
                X.append(x_elem)
                s.append(s_elem)
```

```
51                 if s_elem > s_out:
52                     s_out = s_elem
53                     history.append([s_elem,calls])
54                 found = True
55                 break
```

Listing 4.11: NM_NS_solver Function SEARCH STEP

```
        if found == False:
            idx = set(np.nonzero(X[-1])[0])
            n_neighbors = len(neighbors)
            neighbors = []
            idx_temp = set(range(N)) - set(idx)

            for i in range((d // 2) + 1):
                for elem1 in itertools.combinations(idx, len(idx) - 1 - i):
                    for elem2 in itertools.combinations(idx_temp, i + 1):
                        temp = list(set(elem1) | set(elem2))
                        neighbors.append(temp)

            neighbors = random.sample(neighbors, min(len(neighbors), n_dim *
                n_neighbors))

            for elem in neighbors:
                x_elem = np.zeros(N)
                x_elem[list(elem)] = theta_l
                if list(x_elem) in buffer:
                    continue
                if calls >= max_calls:
                    stop = True
                    break

                s_elem = Influence_evaluation(g, W, x_elem, params)[0]
                buffer.append(list(x_elem))
                calls += 1
                print("\r" + "Nonmonotone Neighbors search... {}/{}. ETA: {}
                    s.".format(calls, max_calls,
                    round((time.time()-start)*((max_calls-calls)/calls))),
                    end = "")

                if s_elem > s_ref:
                    queue = insert(queue, n, s_elem)
                    X.append(x_elem)
```

```
88                      s.append(s_elem)
89                      if s_elem > s_out:
90                          s_out = s_elem
91                          history.append([s_elem,calls])
92                      found = True
93                      break
```

Listing 4.12: NM_NS_solver Function Enriched SEARCH STEP

```
        if found == False:
            idx = set(np.nonzero(X[-1])[0])
            neighbors = []
            idx_temp = set(range(N)) - set(idx)

            for i in range(d // 2):
                for elem1 in itertools.combinations(idx, len(idx) - 1 - i):
                    for elem2 in itertools.combinations(idx_temp, i + 1):
                        neighbors.append(list(set(elem1) | set(elem2)))

            for elem in neighbors:
                x_elem = np.zeros(N)
                x_elem[list(elem)] = theta_l
                if list(x_elem) in buffer:
                    continue
                if calls >= max_calls:
                    stop = True
                    break

                s_elem = Influence_evaluation(g, W, x_elem, params)[0]
                buffer.append(list(x_elem))
                calls += 1
                print("\r" + "Nonmonotone Neighbors search... {}/{}. ETA: {}
                    s.".format(calls, max_calls,
                    round((time.time()-start)*((max_calls-calls)/calls))),
                    end = "")
```

Listing 4.13: NM_NS_solver Function additional SEARCH STEP

```
                if s_elem > s_ref:
                    queue = insert(queue, n, s_elem)
                    X.append(x_elem)
                    s.append(s_elem)
                    if s_elem > s_out:
                        s_out = s_elem
                        history.append([s_elem,calls])
                    found = True
                    break

            stop = True

    x_out = X[s.index(s_out)]
    history.append([s_out, calls])

    s.append(s_out)
    X.append(x_out)

    print("\r" + "Nonmonotone Neighbors search... {}/{} Done!".format(calls,
        max_calls))
    print("s: {}".format(s[-1]))

    return s, X, history
```

Listing 4.14: NM_NS_solver Function Termination Check and Output

4.2.4 Other Methods

In this section, we introduce the implementation of two additional algorithms employed as utility tools during our testing procedures.

Listing 4.15 showcases the code for the Brute Force algorithm designed for the Influence Maximization (IM) problem. Notably, the `brute_force_ranker` function not only executes the aforementioned method but also generates a list containing the ranking of all points. This feature proved invaluable, particularly for smaller-scale problems, as it offered insights into the quality of solutions produced by our direct search methods.

On the other hand, Listing 4.16 presents the code for the RS solver. The `RS_solver` function closely resembles a simplified SEARCH step of the CDS method, with the primary distinction being the random selection of points for evaluation.

```
def brute_force_ranker(g, W, params, K):
    N = len(g.nodes)
    combinations = itertools.combinations(range(N), K)
    s = []
    X = []
    for elem in combinations:
        x_temp = np.zeros(N)
        x_temp[list(elem)] = params[2]
        s.append(Influence_evaluation(g, W, x_temp, params)[0])

    indices = list(range(len(s)))
    indices.sort(key=lambda x: s[x], reverse=True)
    rankings = [0] * len(indices)
    for i, x in enumerate(indices):
        rankings[x] = i+1

    return s, rankings
```

Listing 4.15: brute_force_ranker Function

```
def RS_solver(g, W, params, K, max_calls):
    N = len(g.nodes)
    calls = 0
    s = 0
    history = []
    start = time.time()

    while calls < max_calls:
        x_temp = rand_bin_array(K, N, params[2])
        s_temp =  Influence_evaluation(g, W, x_temp, params)[0]
        calls += 1
        print("\r" + "Random search... {}/{}. ETA: {} s.".format(calls,
            max_calls,
            round((time.time()-start)*((max_calls-calls)/calls))), end = "")

        if s_temp > s:
            s = s_temp
            X = x_temp
            history.append([s, calls])

    history.append([s, calls])

    print("\r" + "Random search... {}/{} Done!".format(calls, max_calls))
    print("s: {}".format(s))

    return s, X, history
```

Listing 4.16: RS_solver Function

5 Results

In this chapter, we delve into a series of numerical experiments to assess the methods examined in the preceding chapter.

We start in Section 5.1 with a concise numerical analysis of the computational cost associated with the GIP model function call. This analysis holds great importance as the GIP function is the central subroutine for each of the methods under consideration.

Subsequently, in Section 5.2, we introduce a collection of artificially generated graphs that will serve as the basis for our experiments.

Preceding the actual numerical experiments, we explore the concept of data profiles in Section 5.3, a pivotal metric for comparing various derivative-free optimization methods.

Finally, Section 5.4 presents the empirical results of our experiments, including a comparative assessment of the methods across different values of the budget parameter K, a critical determinant of the complexity of the IM problem.

Throughout this chapter, we consistently employ the following parameters for the GIP propagation model:

- $l_0 = 1$
- $h_0 = 1$
- $\theta_l = 1$
- $\theta_h = 1000$
- $\gamma = 0$
- $\varepsilon = 0.1$

While for the Direct Search Methods we used the following parameters:

- $\zeta = 0.1$
- $\delta = 0.5$
- $d = 2$
- $n = 4$
- $n\_dim = 4$
- $max\_calls = 5000$
- $buffer\_dim = 500$

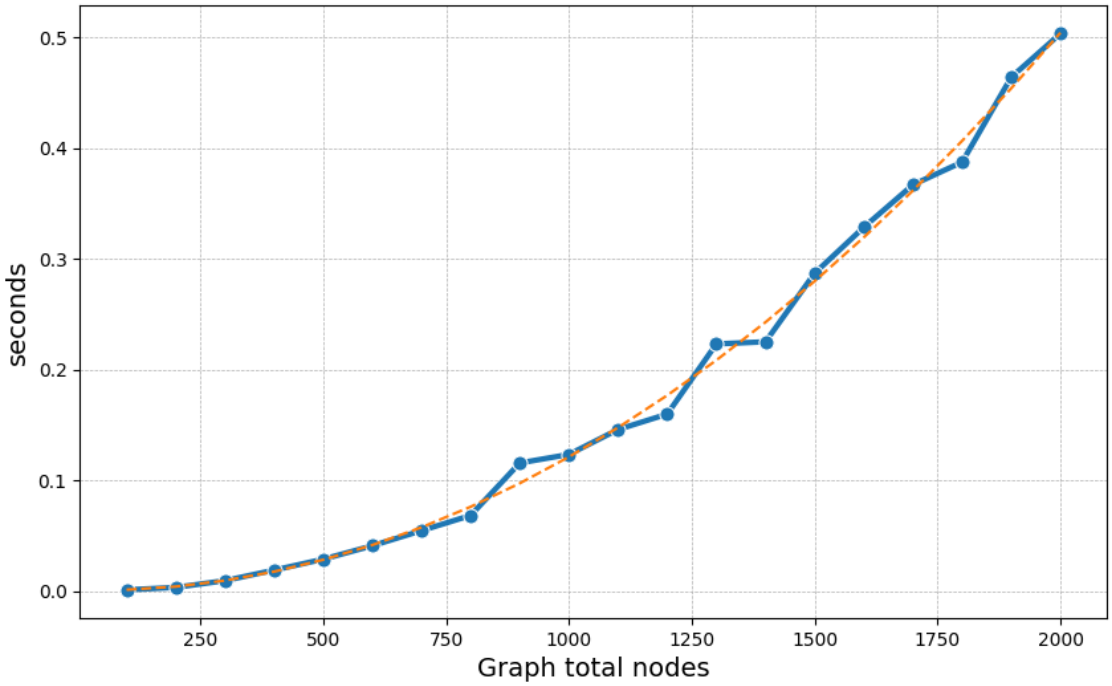
5.1 GIP Model Function

In this section, we analyze the complexity analysis of the GIP model function. This analysis is of paramount significance for our experiments, given that, as we will demonstrate, a substantial portion of the computational burden within our direct search methods is attributed to this function call.

As previously discussed in Chapter 2, the time complexity for calculating the total influence generated is $O(|E|t)$. Notably, for the scale of our experiments and due to our fixed parameter ε, we treat the value of t as constant. Consequently, we can approximate the computational complexity of calculating the total influence for a solution point as quadratic in relation to the total number of nodes. This hypothesis is empirically confirmed by the observations presented in Figure 5.1.As we can see, the blue line representing the observed data can be well approximated by a quadratic formula, as highlighted by the dotted orange line.

This plot was generated by calculating the average time consumed by the `influence_evaluation` function across numerous random points. For each step of 100 nodes increment in graph dimension, we employed 4 different graphs and considered 40 points for every graph, ensuring more robust and reliable statistical insights.

Figure 5.1: The blue line represents the observed cost of GIP calls, while the dotted orange line represents a fitted quadratic function.



5.2 Generated Networks

In this section, we present a collection of artificially generated networks utilized for conducting numerical experiments on the three direct search methods outlined in this thesis. These networks were generated using the Stochastic Block Model (SBM), a model specifically designed to capture the inherent structure of complex networks. The SBM partitions nodes into distinct groups or "blocks" and defines connection probabilities between nodes based on their block assignments. This model assumes that nodes within the same block exhibit similar connectivity patterns, while nodes from different blocks have varying connection probabilities. Key parameters include the number of blocks, block sizes, and edge probabilities conditioned on block assignments.

For our numerical experiments, we aimed to maintain consistent characteristics across the generated network dataset, as we later vary the budget value, significantly impacting problem scale. Consequently, we focused on networks composed of two distinct blocks or communities. To construct these networks, we initially selected five different sizes for the first block. Subsequently, for each of these chosen first block sizes, we selected second block sizes that spanned four distinct scales relative to their corresponding first block sizes. This approach enables the creation of networks with diverse block structures, facilitating comprehensive analysis. For each pair of size values, we randomly generated a symmetric probability matrix. Additionally, we imposed the constraint that the diagonal elements of these probability matrices have higher values than the off-diagonal elements. This constraint mirrors

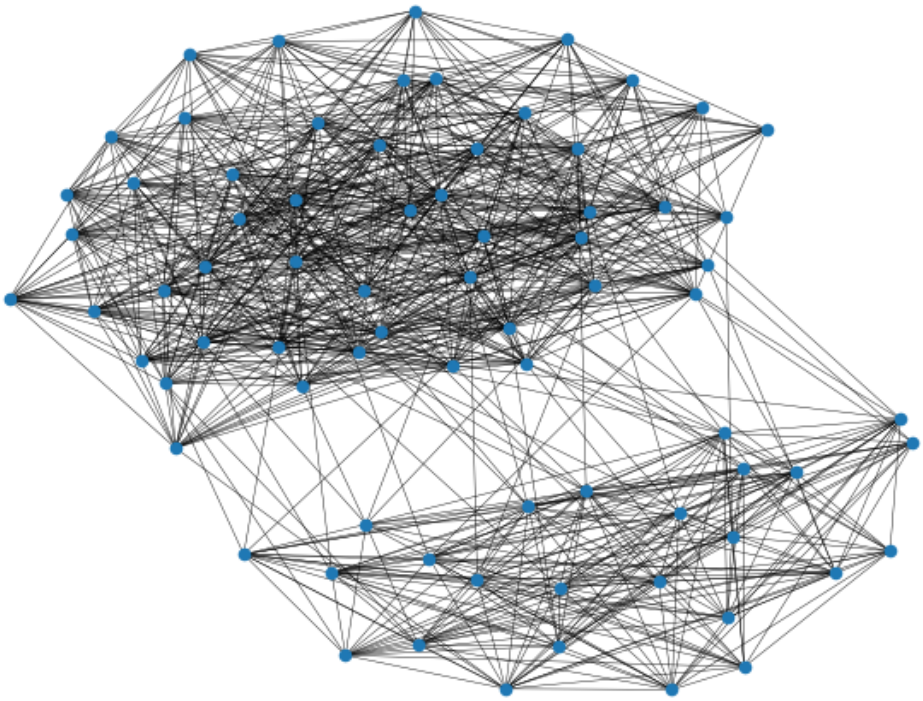
real social graphs where individuals within the same community tend to have stronger connections compared to those in different communities. Additionally, it is worth mentioning that we standardized the connection weight to a uniform value of 0.1.

The complete list of generated graphs is detailed in Table 5.1. We selected dimensions for the first block as 50, 100, 150, 200, and 250. For the ratio between the two block sizes, we used values of 1, 5/6, 2/3, and 1/2. To ensure the total number of nodes is an integer, we rounded the second block sizes to the nearest whole number. Therefore, in the table, we provide only approximate ratios. Moreover, in Figure 5.2, we can see an example of a network used in our test. In particular, it represents Problem 16, which has the smallest number of total nodes.

Table 5.1: List of Problems

Probem ID	First Block Size	Second Block Size	Approximate Ratio
1	50	50	1
2	100	100	1
3	150	150	1
4	200	200	1
5	250	250	1
6	50	41	0.83
7	100	83	0.83
8	150	125	0.83
9	200	166	0.83
10	250	208	0.83
11	50	33	0.67
12	100	66	0.67
13	150	100	0.67
14	200	133	0.67
15	250	166	0.67
16	50	25	0.5
17	100	50	0.5
18	150	75	0.5
19	200	100	0.5
20	250	125	0.5

Figure 5.2: Representation of the problem 16 network using the NetworkX library



5.3 Data Profiling

In this section, we introduce the crucial data profile metric, which holds significant importance when working with derivative-free methods like the ones presented in this thesis. For a more comprehensive explanation, please refer to [16].

Users conducting costly function evaluations often require a convergence test that quantifies the increase in function value. We propose the following convergence test:

$$f(x) - f(x_0) \geq (1 - \tau)(f_L - f(x_0)) \tag{5.1}$$

In this test, $\tau > 0$ represents a user-defined tolerance, x_0 denotes the initial starting point for the problem, and f_L is calculated individually for each problem. f_L corresponds to the highest value of f achieved by any solver within a specified number μ_f of function evaluations. This convergence test is particularly well-suited for derivative-free optimization methods due to its scale-invariant nature. It evaluates the increase in the function value $f(x) - f(x_0)$

achieved by the point x relative to the best possible increase $f_L - f(x_0)$.

We define the data profile of a solver $s \in S$ as follows:

$$d_s(\alpha) = \frac{1}{|P|} \text{size} \left\{ p \in P : \frac{t_{p,s}}{n_p + 1} \leq \alpha \right\} \tag{5.2}$$

In this definition, $|P|$ represents the number of problems, n_p is the number of variables in problem $p \in P$, and $t_{p,s}$ is the number of function evaluations required to satisfy equation (5.1) for a given tolerance τ. With this scaling, the unit of cost is $n_p + 1$ function evaluations. This unit is convenient and can be readily converted into function evaluations. Furthermore, it allows us to interpret $d_s(\alpha)$ as the percentage of problems that can be solved with the equivalent of α function estimates.

5.4 Numerical Experiments

In this section, we present the results of our numerical experiments comparing the three direct search methods implemented in this thesis: the Custom Direct Search method, the Neighbors Search method, and the Non-monotone Neighbors Search method. We also include the results of the Random Search method as a baseline for comparison.

It is essential to emphasize the pivotal role of the budget parameter, denoted as K, in the scalability of the problem. The number of possible points to be evaluated, given a budget K, grows exponentially with both the number of nodes n and the budget itself. In practical applications, n is typically fixed by the nature of the real network under observation, while K allows some flexibility.

Our experiments encompass three different budget values: 2, 3, and 4. For each budget value, we conducted tests following a standardized pipeline:

1. We selected a common maximum number of function calls.
2. We evaluated the performance of each method on every problem in the dataset.
3. For each problem, we recorded the history of improvements concerning the number of function calls for each method, storing this information in a matrix.
4. We aggregated the results from all problems into a single tensor.
5. We computed the data profile on the aggregated results using various gate values.

In the following pages, we present the results of our tests, beginning with an examination of the problem set with budget $K = 2$. This set is the simplest among all the sets considered. From Images 5.3,5.4 and 5.5, it is evident that both the NS and NMNS methods outperformed the CDS method, albeit in different ways. Notably, the NS method consistently exhibits quicker convergence to a good solution, typically aligning with the solution obtained by the CDS method. Conversely, the NMNS method, owing to its nonmonotone nature, initially progresses more slowly. However, with sufficient function calls, it tends to uncover better solutions. A specific instance of this behavior is evident in Image 5.3, where the NS and CDS methods converge to the same point, with the NS method demonstrating faster convergence than the CDS. On the other hand, the NMNS method, after an initially slower start, surpasses both of the previous methods. It is important to emphasize that this first set of

problems is relatively simple for all the methods. This observation becomes apparent when we consider Image 5.5, where, with a low tolerance setting, even the straightforward RS method demonstrates results comparable to the other direct search methods.

However, the situation changes when considering a budget of $k = 3$, as evident from Images 5.6,5.7 and 5.8. While the NS method continues to outperform the CDS method, this difference becomes even more pronounced in this more complex set of problems. Conversely, the NMNS method struggles to maintain performance similar to the CDS method but still achieves comparable results.

This trend is further confirmed in the last and most challenging set of problems, those with a budget of $K = 4$. As seen in Images 5.9,5.10 and 5.11, the NS method clearly outperforms the CDS method. Specifically, Image 5.9 demonstrates that the NS method consistently finds solutions that are generally superior to those of the CDS method. This can be attributed to the fact that both methods do not completely converge on all problems using only 5000 function calls, highlighting the NS method's faster convergence. However, the NMNS method performs poorly and, as shown in Image 5.11, almost as bad as the RS method in finding average solutions. This could be due to two primary reasons: first, the method may not have enough function calls to effectively exploit non-monotonicity, and second, when increasing the value of the budget K, the enriched search step may not explore sufficiently meaningful points.

Figure 5.3: Data Profile for $k = 2$ and tolerance $\tau = 0.99$

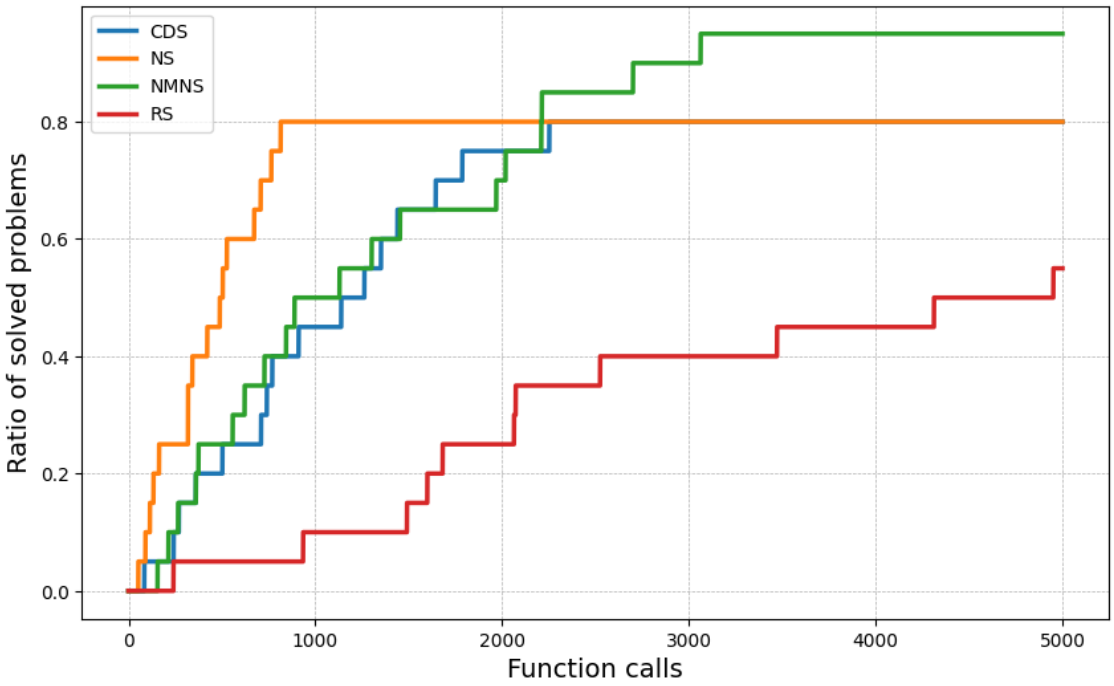


Figure 5.4: Data Profile for $k = 2$ and tolerance $\tau = 0.95$

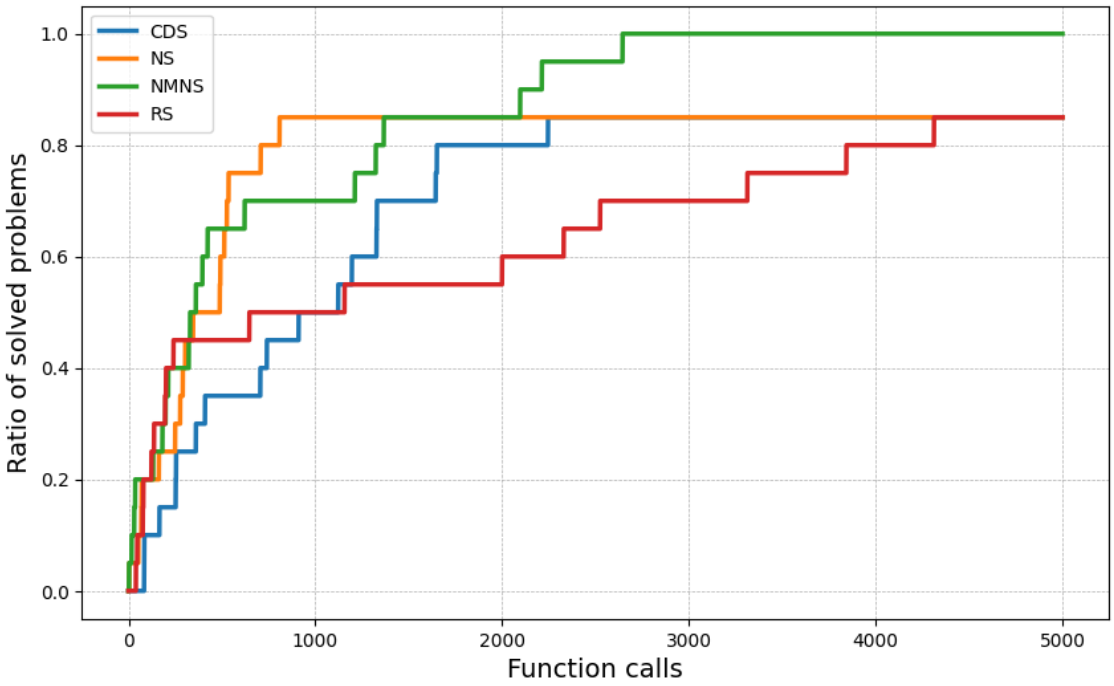


Figure 5.5: Data Profile for $k = 2$ and tolerance $\tau = 0.90$

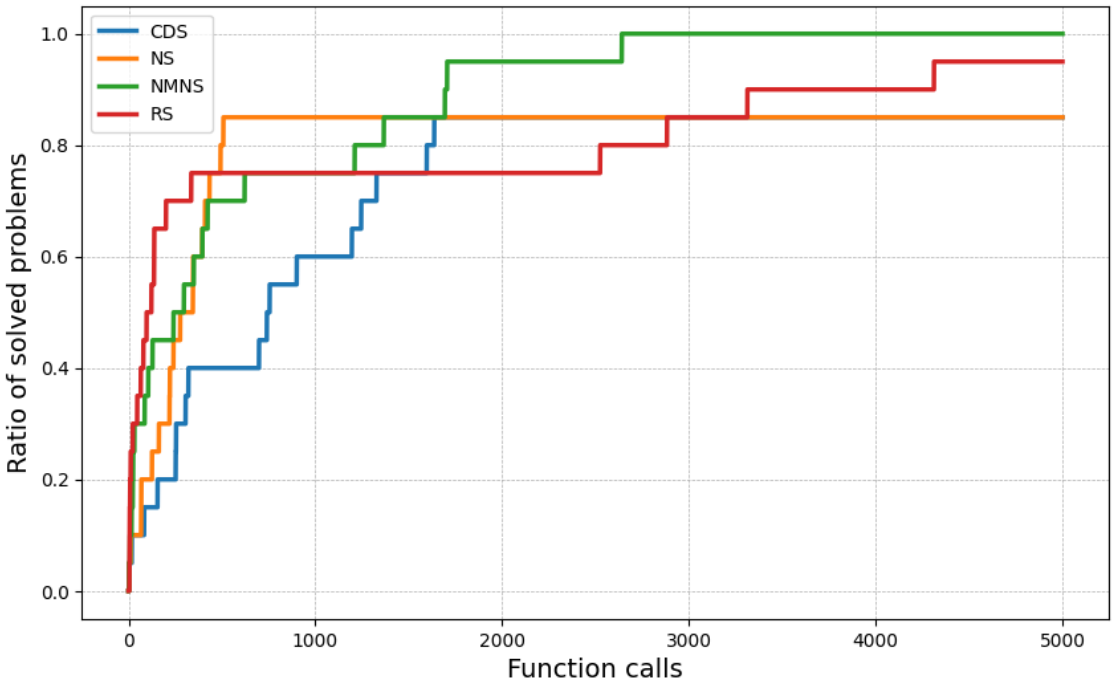


Figure 5.6: Data Profile for $k = 3$ and tolerance $\tau = 0.99$

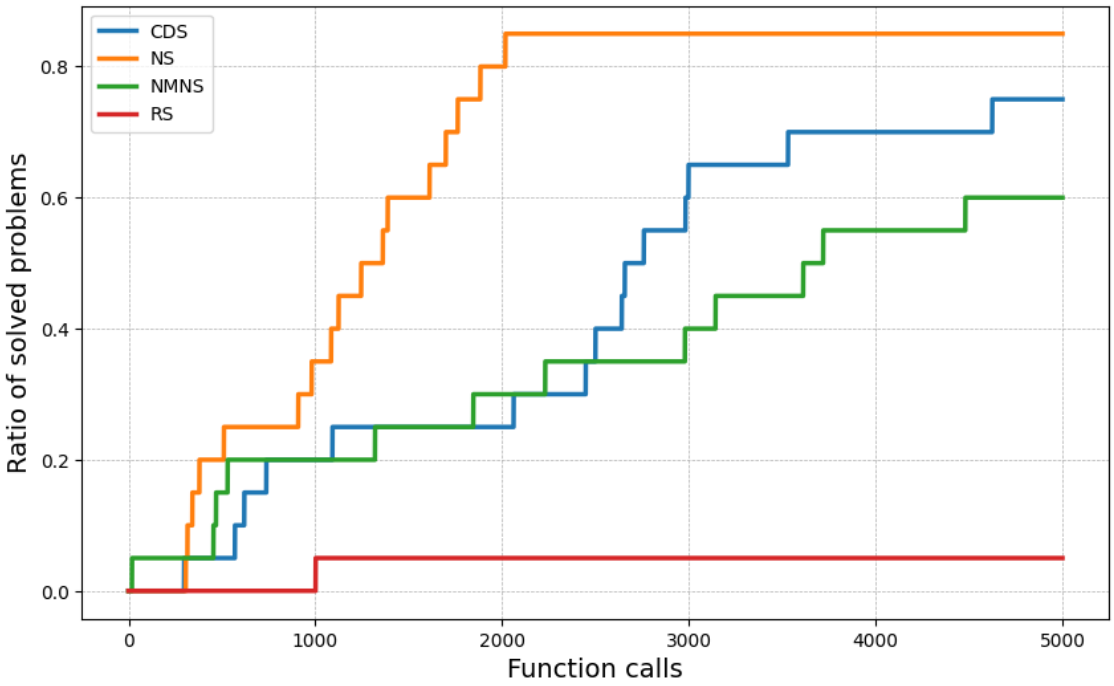


Figure 5.7: Data Profile for $k = 3$ and tolerance $\tau = 0.95$

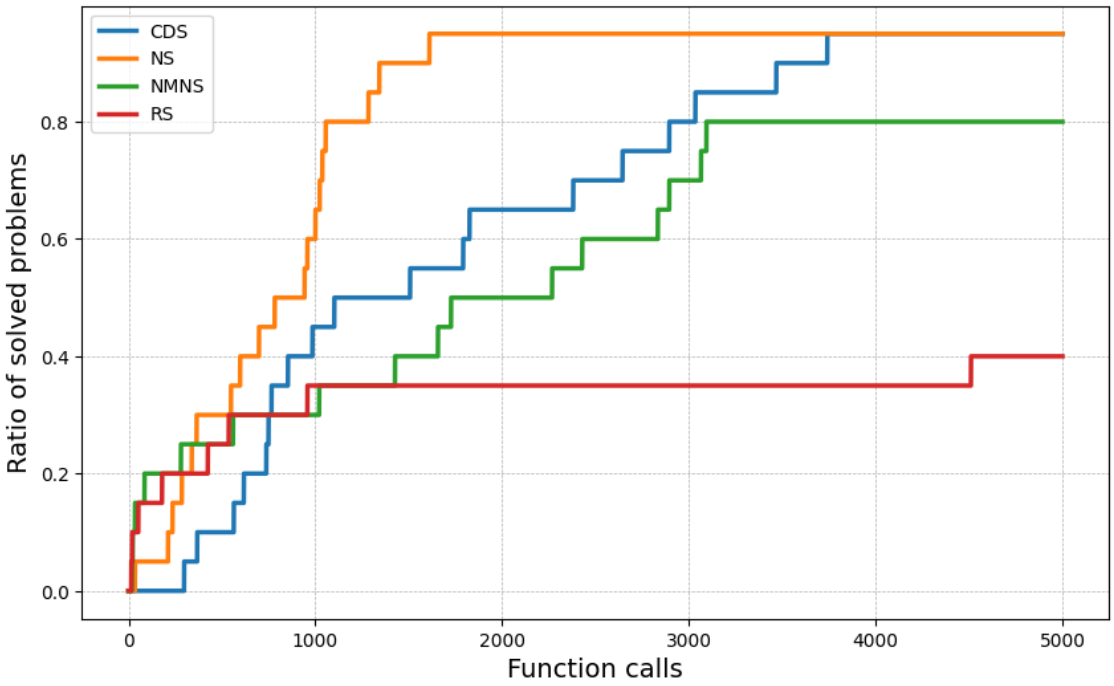


Figure 5.8: Data Profile for $k = 3$ and tolerance $\tau = 0.90$

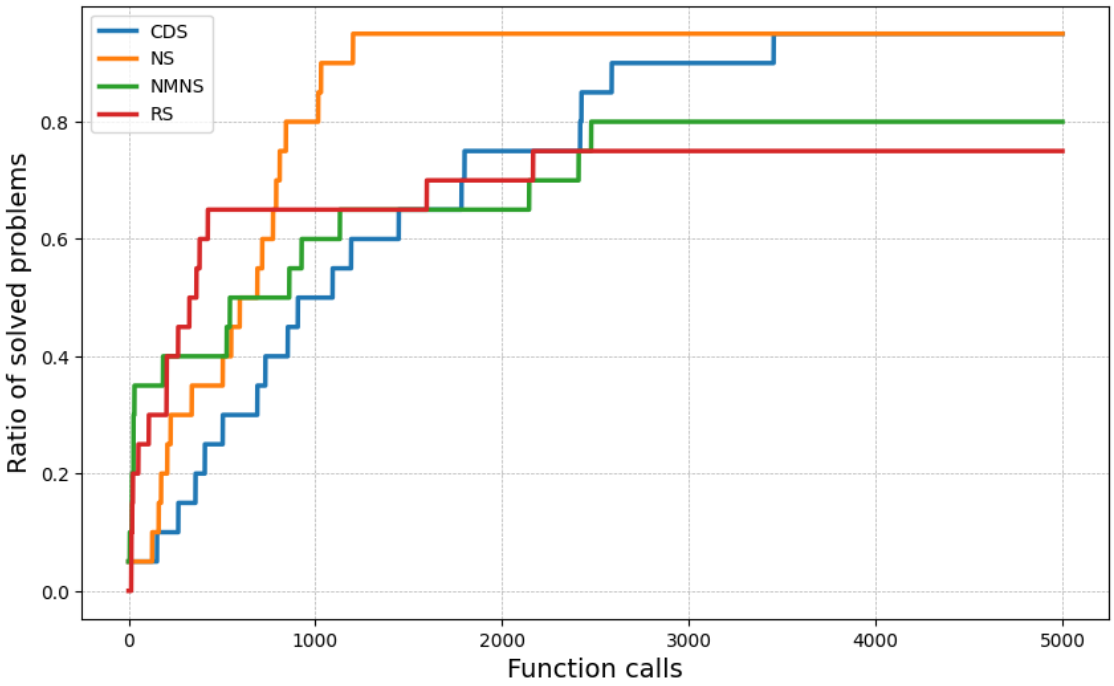


Figure 5.9: Data Profile for $k = 4$ and tolerance $\tau = 0.99$

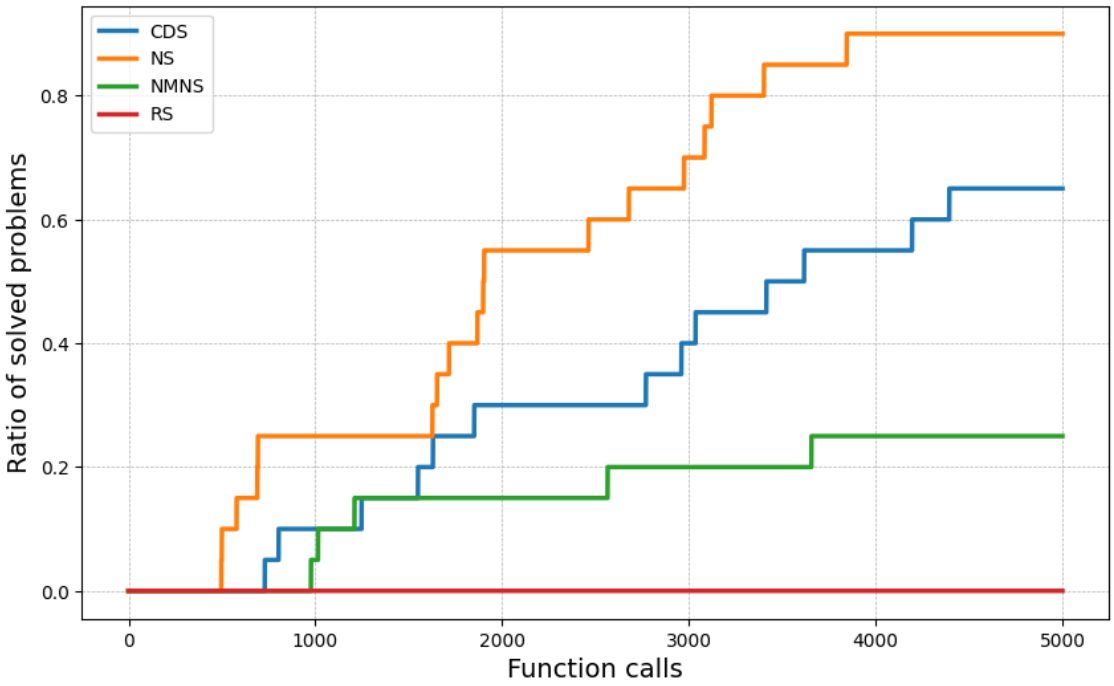


Figure 5.10: Data Profile for $k = 4$ and tolerance $\tau = 0.95$

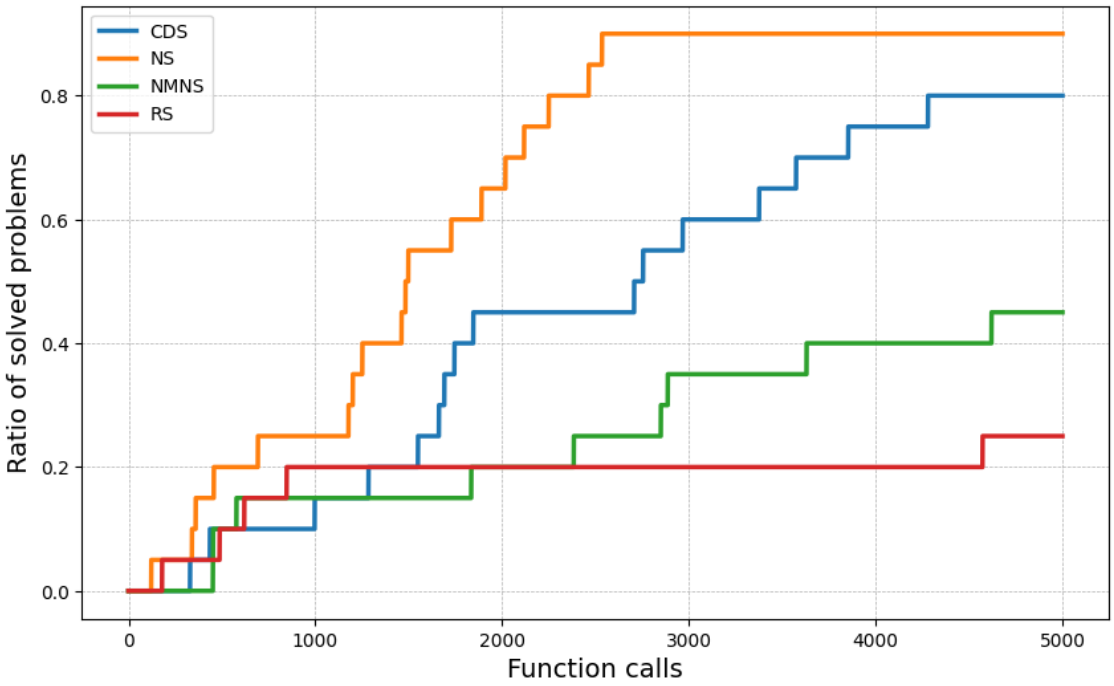
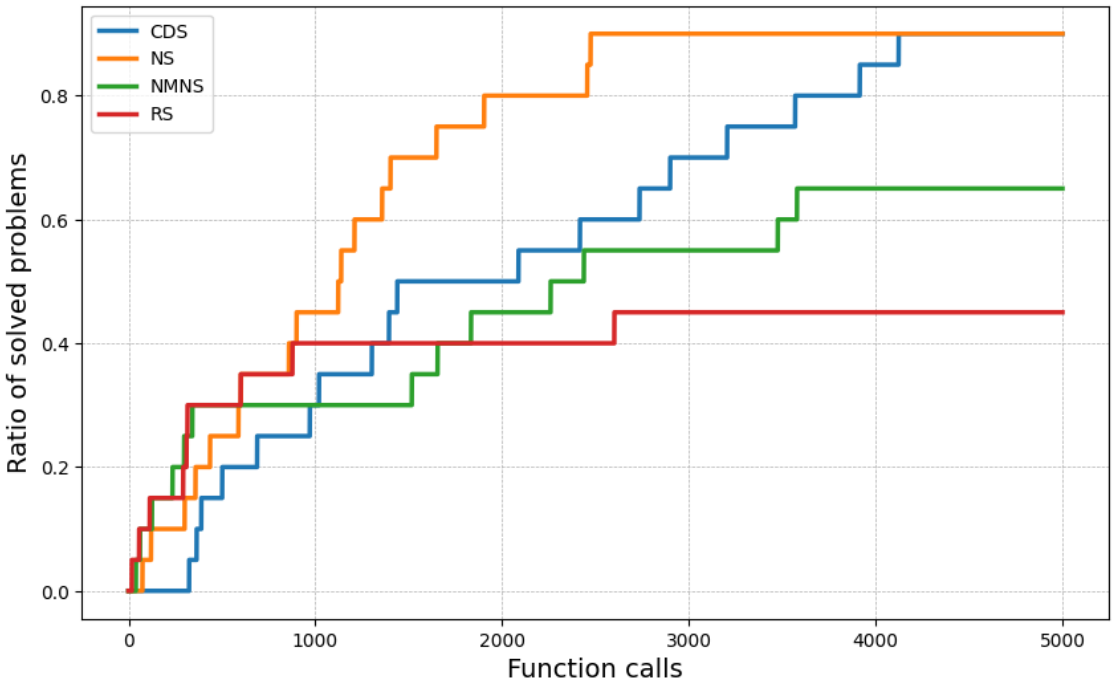


Figure 5.11: Data Profile for $k = 4$ and tolerance $\tau = 0.90$



6
Conclusion

In conclusion, this thesis explored the challenging domain of Influence Maximization, a problem centered around optimizing influence functions within network structures. Networks serve as invaluable tools for comprehending complex systems, making the IM problem highly pertinent across various domains, ranging from marketing strategies to public health interventions.

The primary objective of this thesis was to introduce and implement novel direct search methods tailored to the IM problem, building upon the well-established CDS method. These new algorithms, NS and NMNS, were designed to address the inherent limitations of existing methods by incorporating network structure into the optimization process. This marks a significant departure from traditional approaches, where network characteristics are often disregarded.

Our empirical experiments, provided valuable insights into the performance of these methods. The NS method demonstrated consistent superiority over the CDS method in terms of convergence speed, showcasing its potential for solving IM problems more efficiently. Moreover, the NMNS method, with its nonmonotone nature, exhibited a unique ability to uncover superior solutions with increased computational resources, although it struggled to match the CDS method in some cases.

It is crucial to note that these initial findings indicate promising directions for future research. The field of direct search methods for IM problems remains relatively uncharted, and our work opens doors for further exploration, particularly in handling larger budgets and more complex network structures. Moreover, the NMNS method, in particular, has revealed substantial untapped potential. Its performance, especially on larger budgets, presents a compelling area for further enhancement. One focal point for future research should concentrate on refining the Enriched SEARCH step, as it holds the key to unleashing the method's full capabilities. Furthermore, it is imperative to extend the scope of numerical experiments by incorporating more intricate network structures, as well as transitioning into real-world network scenarios. This expansion will not only validate the applicability of the proposed methods in practical settings but also shed light on their performance in the face of the complexities inherent in real-world data.

In summary, this thesis represents a significant advancement in the realm of IM problem-solving by introducing novel methods that account for network structure. While the results are promising, they are only the beginning of a broader journey to enhance the efficiency and effectiveness of IM problem solutions. Future research should build upon these foundations, ultimately contributing to more impactful decision-making in various real-world applications.

References

[1] Y. Tian and R. Lambiotte, "Unifying diffusion models on networks and their influence maximisation," *CoRR*, vol. abs/2112.01465, 2021. [Online]. Available: https://arxiv.org/abs/2112.01465

[2] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 137–146. [Online]. Available: https://doi.org/10.1145/956750.956769

[3] P. Shakarian, A. Bhatnagar, A. Aleali, E. Shaabani, and R. Guo, "The independent cascade and linear threshold models," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:123320083

[4] E. Mossel and S. Roch, "On the submodularity of influence in social networks," 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:401879

[5] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. Vanbriesen, and N. S. Glance, "Cost-effective outbreak detection in networks," in *Knowledge Discovery and Data Mining*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:850930

[6] A. Goyal, W. Lu, and L. V. S. Lakshmanan, "Celf++: optimizing the greedy algorithm for influence maximization in social networks," *Proceedings of the 20th international conference companion on World wide web*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:3460004

[7] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier, "Maximizing social influence in nearly optimal time," 2016.

[8] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," *2010 IEEE International Conference on Data Mining*, pp. 88–97, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:14294472

[9] E. D. Demaine, M. Hajiaghayi, H. Mahini, D. L. Malec, S. Raghavan, A. Sawant, and M. Zadimoghadam, "How to influence people with partial incentives," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 937–948. [Online]. Available: https://doi.org/10.1145/2566486.2568039

[10] M. A. Abramson, C. Audet, J. W. Chrissis, and J. G. Walston, "Mesh adaptive direct search algorithms for mixed variable optimization," *Optimization Letters*, vol. 3, pp. 35–47, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:18400572

[11] C. Audet, S. L. Digabel, V. R. Montplaisir, and C. Tribes, "Nomad version 4: Nonlinear optimization with the mads algorithm," 2021.

[12] S. Le Digabel, "Algorithm 909: Nomad: Nonlinear optimization with the mads algorithm," *ACM Trans. Math. Softw.*, vol. 37, no. 4, feb 2011. [Online]. Available: https://doi.org/10.1145/1916461.1916468

[13] M. Laguna, F. Gortázar, M. Gallego, A. Duarte, and R. Martí, "A black-box scatter search for optimization problems with integer variables," *Journal of Global Optimization*, vol. 58, pp. 497–516, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:17421991

[14] G. Liuzzi, S. Lucidi, and F. Rinaldi, "An algorithmic framework based on primitive directions and nonmonotone line searches for black-box optimization problems with integer variables," *Mathematical Programming Computation*, vol. 12, pp. 1–30, 02 2020.

[15] M. Locatelli and F. Schoen, *Global optimization: theory, algorithms, and applications*. SIAM, 2013.

[16] J. Moré and S. Wild, "Benchmarking derivative-free optimization algorithms," *SIAM Journal on Optimization*, vol. 20, pp. 172–191, 01 2009.