Learning a Fair Policy for the Influence Maximization Problem

Rutten, Bart J.G.

Award date:
2022

Eindhoven University of Technology
Department of Mathematics and Computer Science

Learning a Fair Policy for the Influence Maximization Problem

MSc Thesis

Bart Rutten, MSc

Supervisors:
prof. dr. Mykola Pechenizkiy, TU/e
dr. Akrati Saxena, TU/e
dr. Alexander Boer, KPMG Nederland

version 1.0

Eindhoven, 2022

Abstract

The Influence Maximization (IM) problem aims to find a subset of nodes in a social network represented by a graph, such that the influence propagating from that subset reaches the largest number of nodes in the network. Algorithms for the IM problem have been extensively studied, but only little work focuses on developing a fair solution. Fairness issues arise, for example, when certain minority communities are excluded from the set of influenced nodes. In this work, we will consider a community as a set of nodes (people) that share the same characteristic for some sensitive attributes like race, sex and religion. We propose a novel Reinforcement Learning (RL) method, called DQ4FairIM (Deep *Q*-learning for Fair Influence Maximization), that aims to maximize the expected number of influenced nodes, while taking into account that minority groups are not disproportionately excluded. We provide a formulation of the problem as a Markov Decision Process (MDP) and use deep Q-learning together with Structure2Vec node-embeddings to solve the MDP. The main benefit of using RL for this problem is its ability to learn a policy based on fixed problem instances and give the solution for a new problem instance without the need to re-train the model from start. By means of experimental results on synthetically generated networks, we show that our proposed method achieves a higher level of fairness, while keeping a high influence spread. Moreover, it outperforms state-of-the-art methods in terms of fairness.

Preface

First and foremost, my gratitude goes out to dr. Akrati Saxena, my daily supervisor, who introduced me to the topic of Influence Maximization and let me write my thesis about this topic. She guided me in the right direction by providing me with relevant literature and useful suggestions. The time she invested, her quick responses to my messages and useful intermediate feedback really encouraged me along the way. Her guidance and support have proven invaluable to me, and I am sure that without her, my thesis would have been on a lower level.

Of course, I would also like to thank my other supervisors: prof.dr. Mykola Pechenizkiy for giving me the freedom in doing this research and dr. Alexander Boer from KPMG for his sharp eye. He really let me think about the subject from a different perspective and encouraged me to think critically about the purpose of my research. His expertise in AI and law and his enthusiasm about this topic has taught me a lot, not only for my specific research, but also in a much broader sense.

Finally, I would like to thank my other colleagues from KPMG's department of Trusted Analytics for the welcoming atmosphere and making my internship an unforgettable experience—and my partner Quỳnh for giving me the mental support to push through at times I needed it. Without her, this project would not have been possible. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Algorithms are increasingly being used in all aspects of life and replacing human-decision making on a large scale. With this increased dependence on algorithms, there has also been a boom of related ethical dilemmas, which gave rise to a new research field: algorithmic fairness. Fairness is a very open area in the field of data mining and although the topic has got more attention in recent years, there are still a lot of unexplored directions. The difficulty of fairness-aware data mining, and with it the diversity, lies in defining what fairness is. In general, fairness can be quite subjective and its definition is problem-specific, there is no one-size-fits-all notion of fairness that can be used for any kind of problem [15].

Generally speaking, the goal of fairness-aware data mining is to analyse data while taking into account potential issues of fairness, discrimination, neutrality, and/or independence, where a distinction is made between detection and prevention of such issues [24]. In previous research fairness is, for example, seen as non-discriminatory for people based on their protected class status such as race, sex, religion, etc., also known as a sensitive attribute [15]. One notable example of why fairness in data mining is important, is the well-known Dutch childcare benefits scandal (Dutch: *toeslagenaffaire*) where around 26,000 parents – mainly those with an immigration background - were wrongly accused of making fraudulent benefit claims based on algorithmic decision-making [19]. Usually there exists a trade-off between fairness and model accuracy, and hence fairness-aware data mining can be seen as a kind of cost-sensitive learning where the cost is the enhancement of fairness [14]. In this research, we will focus on enhancing fairness in a specific social problem: the Influence Maximization (IM) problem.

Briefly stated, the problem of IM is to find a set of nodes to activate in a directed or undirected graph, so that the expected number of nodes that will eventually get activated in the network is maximized [31]. To put it more concrete, the IM problem is about answering the question: If we can try to convince a subset of individuals to adopt a new product or innovation, and the goal is to trigger a large cascade of further adoptions, which set of individuals should we target? Some example applications of IM are advertisement campaigns, viral online content, news propagation and social intervention [48]. Unfairness can arise when certain groups within a social network are disproportionately excluded from the information. Closely related to influence maximization is influence minimization where the goal is to minimize the nodes that will eventually get activated, a well-known example of this problem is fake-news mitigation [45]. Some research has been done on fairness in social influence maximization by, for example, Stoica et al. [48], Becker et al. [4]

and Tsang et al. [51]. Stoica et al. showed the interplay between being fair and strategic with a basic greedy heuristic and in later work proposed a seeding strategy to prevent unfairness in IM [49]. Becker et al. proposed to use randomization as a mean for achieving fairness and Tsang et al. defined an algorithmic framework to find a fair solution by means of a multi-objective submodular maximization problem. We will elaborate more on these methods in Section 1.2. Nevertheless, there are still a lot of unexplored directions, such as scalable fairness-aware approaches or enhancing fairness in more complicated solution methods like reinforcement learning.

1.1 Background

Influence maximization is a famous and well-studied problem in social networks, of which a known application is fake-news mitigation [45]. Other famous applications are advertisement campaigns, viral online content, news propagation and many others [48]. Given an undirected $G = (V, E)$, the goal is to identify a starting set $S \subset V$ of k nodes to activate that maximizes the expected number of nodes $\sigma(S)$ that will eventually get activated in the network. In the context of fake news, the goal is to find a set S of k nodes to block/immunize so that $\sigma(S)$ is minimized. In addition to the graph, a model of diffusion in the network is given. Two of the most popular models are the *Independent Cascade* (IC) and *Linear Threshold* models. In the IC model, the graph G is weighted: for each edge (u, v) there is a weight $p_{u,v} \in [0, 1]$ to be interpreted as the probability that this edge will propagate the information. In the LT model, for any node, all its neighbours that are activated just at the previous time-stamp, together make a try to activate that node. This activation process will be successful, if the sum of the incoming active neighbour's probability becomes either greater than or equal to the node's threshold.

Several solution methodologies have been proposed over the years, and they can be classified in roughly four categories: approximation algorithms, heuristic solutions, metaheuristic solutions and community-based solutions [2]. One of the most famous approximation algorithms is the greedy algorithm proposed by Kampe et al. [26]. Starting with the empty set, this algorithm (see Algorithm 1) iteratively selects a node that is not in S, so that the social influence function σ for the union of S with this node is maximized. A major drawback of this method is its unscalability. For example, applying this algorithm to a medium-sized network of only 15,000 nodes is already unrealistic [8], whilst real-life social networks could contain millions of different nodes. Nevertheless, Kempe et al.'s algorithm is considered as the foundation for solving the IM problem. Many other approximation algorithms are based on the greedy heuristic, of which examples are the static greedy algorithm proposed by Cheng et al. [9] and the Cost Effective Lazy Forward (CELF) scheme introduced by

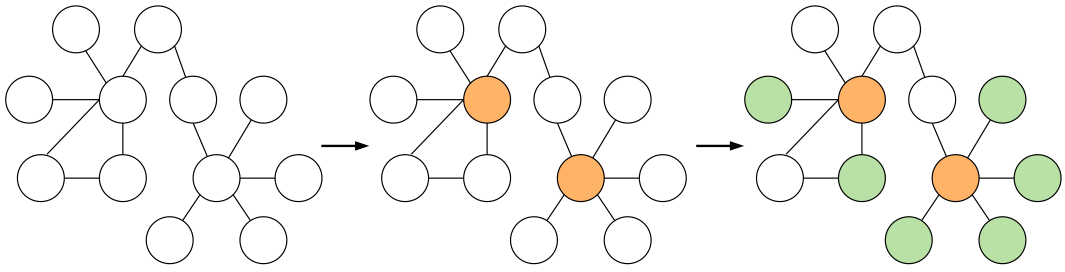


Figure 1.1: The goal of Influence Maximization is to pick an initial set of nodes in the graph (orange) such that the number of influenced nodes is maximized (green).

Leskovec et al. [28]. Algorithms belonging to the set of approximation algorithms give a worst-case bound on the influence spread.

Algorithms belonging to the set of heuristic solutions do not give any approximation bound on the worst-case influence spread, but have generally a better running time and scalability. Over the years, many different heuristics have been proposed. An example is the method proposed by Cordasco et al. [10], their heuristic gives an optimal solution for trees, cycles and complete graphs and performs significantly better than state-of-the-art methods for real-life social networks. Another example is the method of Wu et al. [57], they developed a two-stage stochastic programming approach. As regards metaheuristics, all famous algorithms like genetic algorithms, simulated annealing, and Tabu search have been applied to the IM problem. Lastly, community-based solutions basically involve detecting communities first and influential nodes second. Communities can be seen as clusters in the graph which are densely connected among themselves and sparsely connected with other nodes in the graph. They can be based on co-interest, co-activities, or co-work, but a community could also be considered as a set of nodes (people) that share the same value for some sensitive attributes like race, sex and/or religion.

Most of the discussed algorithms are designed for networks where the whole structure is known beforehand. However, in many real world applications of influence maximization, practitioners intervene in a population whose social structure is initially unknown [56]. A recent, novel approach to this problem was introduced by Kamarthi et al. [23]. They proposed a deep Reinforcement Learning (RL) approach for effective graph sampling. RL is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. Important for RL is to formulate the problem as a Markov Decision Process (MDP). Kamarthi et al. formalized their network discovery problem as an MDP as follows: The current state is the graph discovered so far, the actions that the agent could take are querying any of the nodes in the graph that are not yet queried, and the rewards the agent receives is the total number of influenced nodes in the entire graph. To solve this MDP, they proposed a neural network architecture called Geometric-DQN and a training algorithm that uses Deep Q-learning to learn policies for network discovery by extracting relevant graph properties from the training dataset.

Another reinforcement learning approach was proposed by Chen et al. [7]. They used RL to solve the Contingency-Aware Influence Maximization problem. This is a special type of the IM problem where the willingness of the nodes to be selected is uncertain. For example, in an HIV-prevention campaign, when a person (node) is invited to become a campaigner (seed node), there is uncertainty in whether it is willing to accept the invitation. Wang et al. [53] proposed another RL model for IM. They formulated it as a classification problem where the goal is to identify influenced nodes based on historical observations, which is different from common approaches where a diffusion process is used to measure influence.

Algorithm 1 Kempe et al.'s greedy algorithm [26]

Input: Graph $G = (V, E)$ and k
Output: Maximum influence set $S \subseteq V$

$S \leftarrow \emptyset$
for $i = 1$ to k **do**
  $u = \arg\max_{u \in V \setminus S} \sigma(S \cup u) - \sigma(S)$
  $S \leftarrow S \cup u$
end for
return S

1.2 Related Work

Fairness is a quite new topic in IM and other related graph-mining problems [42]. Link prediction, for example, is a well-studied problem for many social, biological, and information networks [32]. Given is an undirected graph $G = (V, E)$ with a set of nodes V and a set of edges E. In the most general setting, multiple links and self-connections are not allowed. This means that the universal set of all possible links U has $\frac{|V| \cdot (|V|-1)}{2}$ elements. We assume that there are some missing links or links that will pop up in the future in the set $U \setminus E$. The goal of link prediction is to find these links. In the context of social networks, fairness-aware link prediction has been studied [43, 41, 33]. Masrour et al. [33]. They presented a novel Fairness-Aware LInk Prediction (FLIP) framework to mitigate the filter bubble problem, which is the reinforced segregation and narrowing the diversity of information exposed to users. They consider a link prediction algorithm to be unfair if it is biased towards promoting certain types of links (e.g., those between users with similar gender or other protected attributes) For example, Hofstra et al. [20] examined the ethnic and gender diversity of social relationships on Facebook and found high levels of ethnic segregation. Averaged over all respondents, they found that approximately three-quarters of respondents' Facebook friends were of a similar ethnic background. Saxena et al. [43] proposed a link prediction method, called NodeSim, that considers both the similarity of nodes and their community information to predict intra-community links as well as inter-community links, with higher accuracy.

Another problem related to graph mining where fairness has been studied is centrality-ranking [44]. The goal is to assign numbers, or rankings to nodes within a graph corresponding to their network position. As a matter of fact, the IM can be seen as a type of centrality ranking, where we try to identify or rank the most influential people in the network. One of the most famous and well-studied algorithms related to centrality ranking is the Pagerank Algorithm, introduced by Brin et al. [5]. PageRank was originally designed to rank academic webpages (nodes) according to their citations (links). It is, among other algorithms, used by Google Search to rank websites in their search engine results. Fairness aware pagerank has been studied by Tsioutsiouliklis et al. [52]. They defined a protected group based on the value of some sensitive attribute such as gender or race, and say that a link analysis algorithm is ϕ-fair, if the fraction of total weight allocated to the members of the protected group is ϕ.

Stoica et al. [48] studied the IM problem for the purpose of designing fair algorithms for diffusion, aiming to understand the effect of communities in the creation of unequal impact among network participants based on attributes like gender and race. They defined two notions of fairness: Fairness for early-adopters and Fairness in outreach. In the case of early-adopters, they consider a social influence maximization algorithm to be fair if the proportion of all communities $C_1, C_2, \ldots, C_c$ in the

starting set S is equal. That is:

$$\frac{\mathbb{E}\{|u \in S | u \in C_i|\}}{|C_i|} = \frac{\mathbb{E}\{|u \in S | u \in C_j|\}}{|C_j|} \quad \forall i, j. \tag{1.1}$$

In the case of outreach, they consider an algorithm to be fair if in the outreach it achieves, the cascade (information flow) reaches all communities in a calibrated way. In other words, if the information being spread in the network reaches the same percentage of each community. Denote the set of nodes that are influenced by I, then we get:

$$\frac{\mathbb{E}\{|u \in I | u \in C_i|\}}{|C_i|} = \frac{\mathbb{E}\{|u \in I | u \in C_j|\}}{|C_j|} \quad \forall i, j. \tag{1.2}$$

Becker et al. [4] proposed to use randomization in order to achieve a higher level of fairness. They considered group fairness, which is similar to Stoica et al., where they consider group fairness for different communities that are induced by some sensitivity attributes like religion, ethnicity, and gender. By making use of randomized strategies, they achieved a higher level of fairness compared to previous studies, indicating that randomness as source of fairness in influence maximization could be promising to be further explored. Stoica et al. [49] showed in later work that promoting better parity in the seed selection process leads to better parity in the outreach as well. Another recent work by Tsang et al. [51] studied group-fairness in IM, and they provided an algorithmic framework to find solutions which satisfy fairness constraints, making sure that influence is fairly distributed across different groups in the population.

1.3 Research problem and questions

The main goal of this project is to explore the possibility of using RL to find a fair solution to the IM problem, while keeping a high influence spread. We think this direction is particularly interesting since RL is a quite novel approach to solve the IM problem and other graph combinatorial optimization problems in general. Previous work on related topics showed that RL can be a good approach for optimization problems over graphs and, under certain circumstances, gives good results compared to other methods. Moreover, to the best of our knowledge, there is no solution that uses RL to achieve fairness in IM. Therefore, we think that a fairness-aware RL approach for IM could be an inspiration for future work on combining some optimization objective with fairness in a reinforcement learning setting.

As regards fairness, existing notions of fairness in IM are mostly applied for different communities such as Stoica et al. [48] and Becker et al. [4]. We will continue in this direction and argue which notion fits best in our research. As stated in the introduction, fairness can be quite subjective and its definition is problem-specific, there is no one-size-fits-all notion of fairness that can be used for any problem. Hence, we are interested in evaluating different notions of fairness in order to find out which one fits best in the context of our problem. To be more concrete, this project will focus on answering the following research questions:

1.3.1 Research Questions

- **RQ1**: *Can we train an RL agent to solve the IM problem?*
  It is not trivial to use RL for the influence maximization problem. The first research question

focuses on how we can actually make RL work in order to solve the IM. In particular, we aim to answer the following sub questions:
1) What are the pros and cons of using RL compared to other algorithms?
2) How can we formulate the classical IM problem as an MDP?
3) How scalable is the RL approach?

- **RQ2**: *How can RL be used to find a fairer solution to the IM problem?*
  The first research question focuses on how we can design a RL framework in the context of the IM problem. The second research question builds upon on this question and asks how we can use this framework to find a fairer solution to the IM problem. In particular, we aim to answer the following subquestions:
  1) How do we define fairness in the context of IM? More specifically, are there any other fairness criteria that are suitable for the IM problem besides the community-based fairness notions (Equations 1.1 and 1.2)?
  2) How can we enhance this fairness notion in the RL framework?
  3) What is the loss incurred by enhancing fairness compared to a RL model that is not restricted to fairness? In other words, what is the price of fairness?
  4) How does the method compare to other baseline methods?
  5) If the algorithm that accounts for fairness gives different solutions, can we explain, for example by evaluating the choices of the agent, why this is the case?

The main goal of the research will be to to answer these (sub)questions in as much detail as possible. In the next section, we will give a little bit more motivation on why we think fairness plays a relevant role in social networks and why we think reinforcement learning could be a promising solution to tackle the problem.

1.3.2 Motivation

Why fairness in IM?
One can argue if and how fairness should play a role in the IM problem. In the example of fake news mitigation, you could argue that the main goal is to reduce the spread of harmful content as much as possible, regardless of any sensitive attributes of the nodes in the network. This is in principle a valid argument. However, fake news is a political sensitive topic and as a social media company you should prevent the situation that only nodes from a specific side of the political spectrum are affected. For example, a social media company might run into trouble when it only eliminates right wing people from its platform. Our research could give a solution for this issue where both the influence is maximized/minimized and fairness is maximized. Note that in this example, we are interested in fairness of the seed nodes rather than fairness in outreach. Considering fairness in outreach, one could think of any application where it is desirable that the information spread within the network is proportionally divided among the communities. An appealing example is the HIV prevention program, where the goal is to spread awareness among the homeless youth. As discussed by Tsang et al. [51], here we wish to ensure that members of racial minorities or the LGBTQ community are not disproportionately excluded. Another recent example in a similar domain is a COVID-prevention program. In these kinds of programs it is not only important to reach as many people as possible, but also to make sure that certain communities (especially minority groups) are not excluded from access to the information.

Yet another recent example shows the importance of fairness in social networks from a legal per-

spective. Facebook's parent company Meta Platforms Inc settled a lawsuit over a housing advertising system that illegally discriminated against users based on race and other characteristics in June 2022 [21]. The US Department of Housing complained that housing ads on Facebook had been discriminatory. For example, they may or may not be shown to people with a certain skin colour, origin, religion, gender or disability. Facebook's artificial intelligence must ensure that certain population groups are not excluded. The changes should also ensure that job and loan advertisements are non-discriminatory. Facebook earns billions from advertisements and can determine very precisely who sees which advertisements based on user data. At the same time, this leads to concerns about privacy and discrimination.

Why reinforcement learning?
Reinforcement learning for graph problems and Combinatorial Optimization (CO) problems in general, has gotten more attention in recent years [17]. The key for any problem when applying RL to solve it, is to translate it to a Markov decision process first. RL has been applied to several famous CO problems like the Traveling Salesman Problem, The Maximum Cut Problem and The Minimum Vertex Cover Problem [34]. These problems have been studied extensively in the past decades and a tremendous amount of different heuristics have been designed to solve these problems, which one wonder why RL should be used for CO problems in the first place. The main advantage of RL is that it can learn a policy based on previously seen problem instances and give the solution for a new problem instance without re-training the model. As discussed by Zheng et al. [58], the rationale behind using RL for graphs is that graphs from the same application domain or similar types are not totally different from each other; they may have similar structures and are often solved repeatedly. An agent can learn a policy based on previously seen graphs and can give a solution for a new graph immediately. This is also the main motivation for the work by Chen [7] et al.: non-profits usually do not have the high performance computing resources available to recalculate the solutions for time-consuming heuristics. By using RL, there is no need to recompute the solution whenever the social network changes and high performance computing resources are not necessary.

1.4 Potential use cases

As discussed in Section 1.3.2, our work focuses on applications of social influence maximization where the aim is to find a fair solution while keeping a high influence spread and RL can potentially be a good approach to tackle the problem. As mentioned, RL is mainly applicable for combinatorial optimization problems where there is a pool of problem instances that are closely related, and it might be computationally more efficient to train one model that can solve all these problem instances at once. This pre-trained model can be used to find solutions for new graphs in a short amount of time. An appealing domain where RL might be useful in particular, is networks of social media websites. Consider a social network like Facebook, new users enter the network, existing users leave the network and users connect with other users every minute. Such a network is very dynamic and changes over time very fast, but probably the core structure or underlying distribution remains the same over time. A domain where fairness specifically plays a more important and where our method could be useful are social intervention campaigns. We will discuss these potential use cases in more detail in Chapter 6.

1.5 Contributions

This work studies the problem of finding a fair solution for the social influence maximization problem. It builds upon previous research on RL for IM and fairness for IM, and combines the two topics. Our main contribution is a novel deep Q-learning approach to solve the influence maximization problem in a *fair* way. Here, with fair we mean that certain communities sharing some sensitive attribute are not disproportionately excluded in the final outreach. We motivate why fairness should get more attention in data science in general and why it is important specifically in the context of social influence maximization. We reach a higher level of fairness by not only letting the reward function depend on the expected number of influenced nodes, but also on a fairness measure. We show that our algorithm performs well by performing an empirical study on different kind of synthetically generated social networks. We hope that our work could be an inspiration for more research about reinforcement learning for social networks, fairness in influence maximization and fairness in data science in general.

1.6 Outline

Chapter 2 gives more detail about the Influence Maximization problem. It provides a mathematical formulation along with some properties that are relevant to our work. Moreover, it discusses diffusion models to measure influence in a network in extensive detail. Chapter 3 provides the basic theory about reinforcement learning. The goal of this chapter is to provide an understanding about the concepts of RL. It first gives a short introduction and explains the concepts of an MDP. All relevant building blocks that we will use in our proposed method are explained: including standard Q-learning, deep Q-learning and neural networks. In Chapter 4 we propose our RL framework to solve the IM problem fairly: DQ4FairIM. Here, we formulate the IM problem as an MDP and explain how we guide the agent to find a fair solution. Chapter 5 is all about experiments and gives empirical evidence that DQ4FairIM is a suitable method to find a fair solution. Finally, in Chapter 6 we draw conclusions and focus attention on promising research directions.

Chapter 2

Influence Maximization

As mentioned in the introduction, the problem of influence maximization is to find a set of nodes to activate in an undirected graph, such that the expected number of nodes that will eventually get activated in the network is maximized. This chapter will dive a bit deeper into the problem by formalizing the concept and providing some mathematical properties that are useful for understanding the topic more thoroughly. Moreover, the two most famous diffusion models, Independent Cascade and Linear Threshold are discussed in more detail and explained by means of some small examples.

2.1 Definitions and properties

The previous chapter focused on introducing the influence maximization problem and giving some background. Here, we will introduce some common terminology and properties of the influence maximization problem as also discussed by Shakarian et al. [47]. Note that some of these terms have been introduced in the previous sections already, but we will formalize them here so that we can refer to them later on.

Definition 2.1 (Influence Spread). *Given a graph $G = (V, E)$ and an initial seed set S of nodes, the influence spread is the expected number of active nodes at the end of the diffusion process, denoted by $\sigma(G, S)$.*

As mentioned previously, the goal of influence maximization is to maximize the influence spread:

Definition 2.2 (Influence Maximization Problem). *Given a natural number k, called the budget, and a graph $G = (V, E)$ find an initial seed set S, where $|S| \leq k$, such that $\sigma(G, S)$ is maximized.*

In general, the influence spread function $\sigma(\dot{})$ satisfies the properties normality, monotonicity and submodularity, which are defined below. Normalization is a straightforward property, it just states that if no seed nodes are selected, there is also no influence spread in the model.

Definition 2.3 (Normality). *If there are no initial seed nodes, there is no influence spread, i.e. $\sigma(G, \emptyset) = 0$.*

Monotonicity states that the influence spread of a set of nodes S is always greater than or equal to the influence spread of a subset of S:

Definition 2.4 (Monotonicity). *For $S' \subseteq S$, $\sigma(G, S') \leq \sigma(G, S)$*

The intuition behind submodalarity is as follows. Consider a group of people S_1, which is a subset of nodes of S_2. If you add a person s to group S_1, then its added value is always at least as much as if you would add this person to group S_2:

Definition 2.5 (Submodularity). *The influence spread function $\sigma : S \longrightarrow \mathcal{R}$ is submodular if and only if for all $S', S'' \subseteq S$, it holds that if $S' \subseteq S''$, then $\sigma(G, S' \cup \{s\}) - \sigma(G, S') \geq \sigma(G, S'' \cup \{s\}) - \sigma(G, S')$. Intuitively, a submodular function has a diminishing returns property.*

2.2 Diffusion models

We will shortly discuss the two most popular diffusion models used in Influence Maximization. First, we will discuss the Independent Cascade (IC) model that we will use for our work as well. Secondly, we will explain an other famous diffusion model, the Linear Threshold (LT) model.

2.2.1 Independent Cascade Model

In the IC model, the graph G is weighted: for each edge (u, v) there is a weight $p_{u,v} \in [0, 1]$ to be interpreted as the probability that this edge will propagate the information. As discussed in the book by Shakarian et al. [46], $p_{u,v}$ can be assigned based on frequency of interactions, geographic proximity, or historical infection traces. This model is the most common in the existing literature to measure influence. Given a graph $G = (V, E)$, the probabilities $p_{u,v}$ for all edges $(u, v) \in E$ and the set of outgoing edges for a node $v \in V$ defined as $\delta^+(v)$, we will follow the definition by Shakarian et al. [47]:

Definition 2.6 (Independent Cascade). *Under the independent cascade model, at each time step t where A_{t-1}^{new} is the set of newly activated nodes at time $t-1$, each $v \in A_{t-1}^{new}$ infects the inactive neighbours $u \in \delta^+(v)$ with a probability $p_{u,v}$. The process terminates at time step T if no new nodes are activated.*

An example of one simulation of the IC model is given in Figure 2.1. The graph consists of 9 nodes, labelled A to I. The numbers on the edges indicate the probabilities, note that here the probabilities are just fixed arbitrary numbers from the set $\{0.1, 0.2, 0.3, 0.4\}$. The initial seed set consisting of $\{A, H\}$ at $t = 0$ is indicated by yellow. Then at time step $t = 1$, nodes A and H have a single chance to influence their neighbouring nodes according to the edge probabilities. That is, node A could influence B with probability 0.1, C with probability 0.3 and E with probability 0.2. Node H could influence F with probability 0.2, G with probability 0.1 and I with probability 0.1. In this simulation, node A influences nodes C and E and node H influences node I. The newly influenced nodes at $t = 1$ are indicated by yellow and have a single chance to influence their neighbouring nodes at the next time step $t = 2$. Node C will influence node B and either node E or I will influence node F. Then at the last time step $t = 3$, nodes E and I do not influence any neighbouring nodes anymore and the model terminates. The final set of influenced nodes is indicated by blue and consists of $\{A, B, C, E, F, H, I\}$.

Note that the sequence illustrated in this figure is only one simulation of the model and that the actual probability of this sequence is very low. What we are mainly interested in, is the *expected* number of influenced nodes under the IC-model. We can calculate the expected number of influenced nodes by running (Monte-Carlo) simulations m times. The results are illustrated in Figure 2.2. The fractions at the nodes represent the percentage of times this node was influenced. We can just

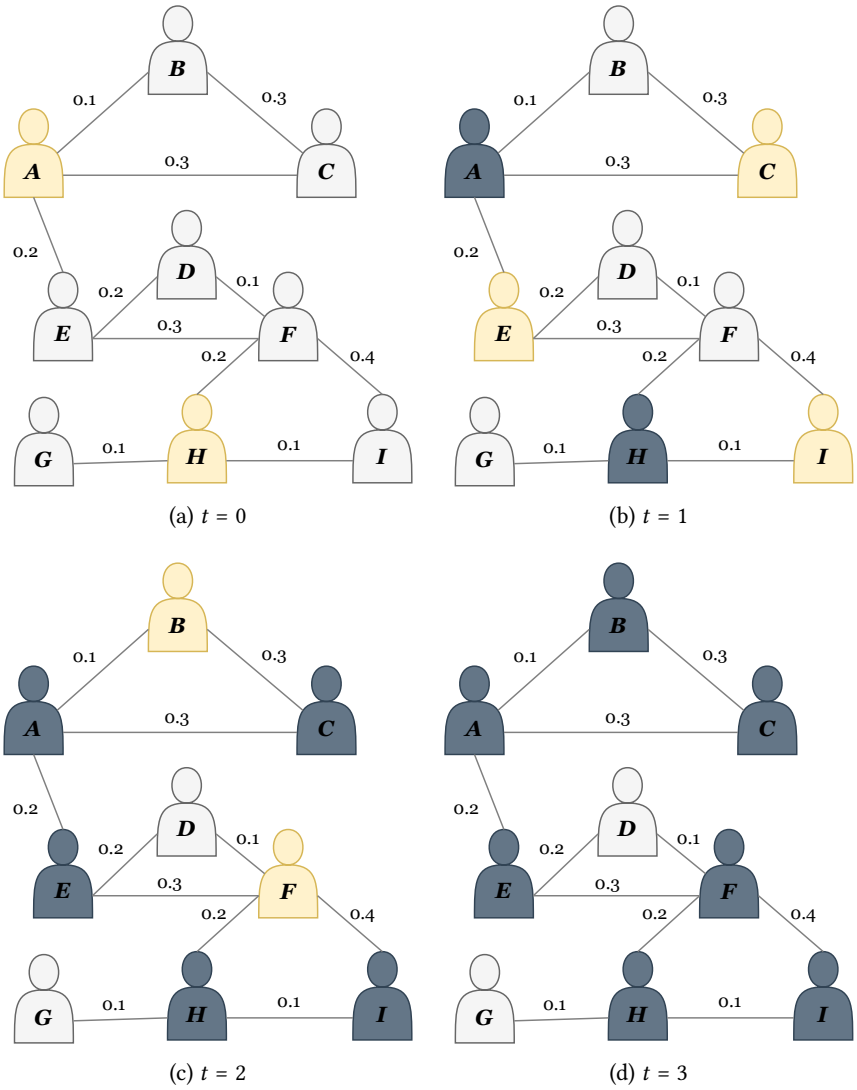


Figure 2.1: A possible simulation of the Independent Cascade model.

sum over all the numbers and end up with an expected number of 3.41 influenced nodes (including the nodes of the seed set). Now for the running time complexity of calculating the influence spread under the IC model we get:

Algorithm 2 Independent Cascade

Input: Graph $G = (V, E)$, seed set S_0 and probability p
Output: Set of influenced nodes $S \subseteq V$

```
t ← 0
A_0 ← S_0
while S_t \ S_{t-1} ≠ ∅ do
    t ← t + 1
    A_t ← {}
    for each v ∈ A_{t-1} do
        for each u ∈ N(v) do
            A_t ← A_t ∪ v with probability p
        end for
    end for
    S_t ← S_t ∪ A_t
end while
return S
```

Theorem 2.7 (**Time complexity of IC**). *The time complexity of evaluating the influence spread under the Independent Cascade model (of a single run) is $O(|E|)$.*

Proof. To see this, we refer to Algorithm 2 and Definition 2.6. Note that each infected node has a *single* change of activating all of its neighbours. This means that in the worst case, we have to evaluate all neighbours for all nodes, which is at most $|E|$. Adding a node to the seed set is just constant time, hence we have $O(|E|)$.

Theorem 2.8 (**Time complexity of simulating IC**). *The time complexity of retrieving the expected number of influenced nodes under the Independent Cascade by running m Monte-Carlo simulations is $O(m|E|)$.*

Proof. The time complexity of evaluating a single run of the IC model is $O(|E|)$ (see Theorem 2.7). We just have to evaluate the influence spread under the IC model m times, hence we get $O(m|E|)$.

As stated by Shakarian et al. [46], influence maximization in the independent cascade model is NP-hard. Moreover, the influence function is normalized, monotone and submodular.

2.2.2 Linear Threshold Model

In the Linear Threshold (LT) model, the graph $G = (V, E)$ is also weighted with weights $b(u, v)$ for each edge $(u, v) \in E$, but the weights do not indicate probabilities. In this model, a node gets activated if the sum of the activated incoming nodes exceeds a certain threshold. Moreover, for each node $v \in V$ the total incoming edge weights cannot exceed 1. We will again follow the definition by Shakarian et al. [46]:

Definition 2.9 (**Linear Threshold**). *Under the linear threshold model dynamics, each node v selects*

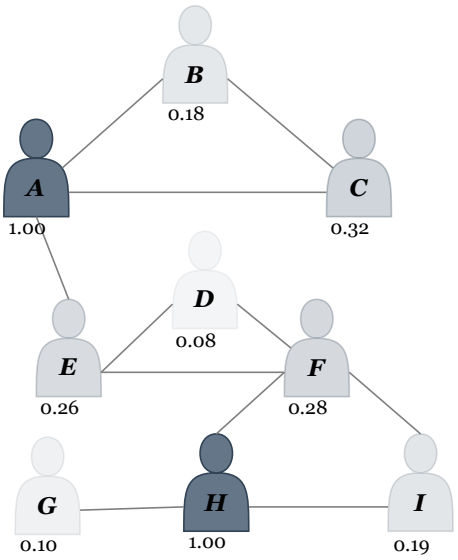


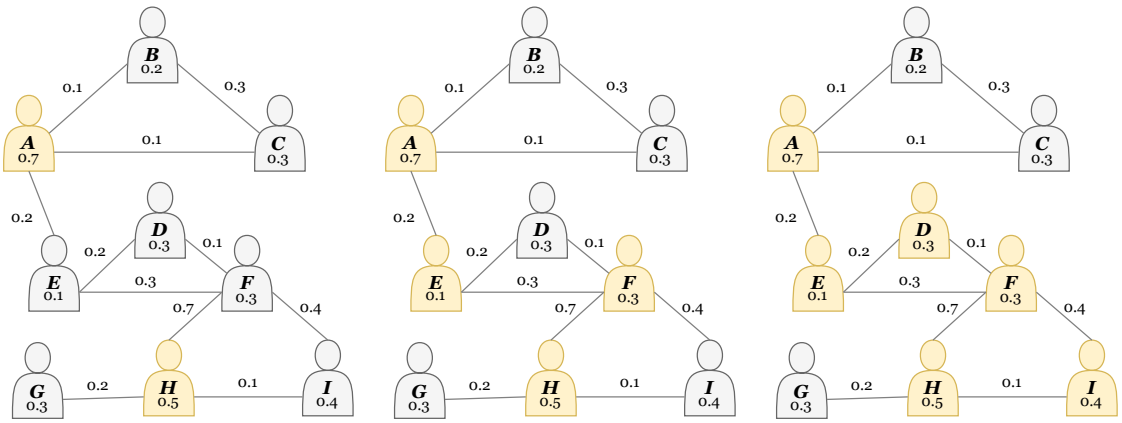
Figure 2.2: Expected number of influenced after running simulations for the graph with seed set as indicated in Figure 2.1(a)

.

a threshold θ_v *in the interval* $[0, 1]$ *uniformly at random. Then at each time step* t *where* H_{t-1} *is the set of nodes activated at time* $t-1$ *or earlier, each inactive node becomes active if* $\sum_{u \in \eta^{in}(v) \cap H_{t-1}} b(u, v) \geq \theta_v$.

Note that the thresholds θ_v do not necessarily have to be random, but can be chosen based on specific node attributes as well, and this usually depend on the problem at hand. However, these thresholds are usually selected randomly due to lack of knowledge of the tendencies of nodes, and express the different levels of tendency of nodes to adopt or believe a message. At each step, an inactive node becomes active if the total weight of its incoming neighbours is at least θ_v.

An example of this model is shown in Figure 2.3. This is the same graph as in Figure 2.1, but with different weights on the edges, and each node is assigned a random threshold in $[0, 1]$. The initial seed set is again picked as $\{A, H\}$. Node A cannot activate nodes B and C since the influence weight is not larger or equal to the thresholds of these nodes ($b(A, B) < \theta_B$ and $b(A, C) < \theta_C$). It will activate node E since the influence weight ($b(A, E) = 0.2$) is larger than the threshold of node E ($\theta_E = 0.1$). Node H cannot activate nodes G and I since the influence weight is not larger or equal to the thresholds of these nodes ($b(H, G) < \theta_G$ and $b(H, I) < \theta_I$). It will activate node F since the influence weight ($b(A, E) = 0.7$) is larger than the threshold of node F ($\theta_F = 0.3$). Hence, at time step $t = 1$ the set of activated nodes is denoted by $\{A, E, F, H\}$. Then, at the next time step, node D gets activated since $b(E, D) + b(F, D) = 0.2 + 0.3 \geq \theta_D = 0.3$ and node I gets activated since $b(F, I) + b(H, I) = 0.4 + 0.1 \geq \theta_I = 0.4$. In the next time step, no new node gets activated and the diffusion process stops. The final set of influenced nodes consists of $\{A, D, E, F, H, I\}$.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

Figure 2.3: An example of the linear threshold model.

Chapter 3

Reinforcement Learning

In this chapter we will give an introduction to the core concepts of Reinforcement Learning (RL). They form the base for our solution method proposed in Chapter 4. The concept of RL explained in this chapter comes from multiple valuable sources, in particular from the book by Sutton & Barto [50] and a lecture about Deep Reinforcement Learning by Li et al. (Stanford University) [29]. First we will give a short introduction to RL without going into theoretical detail along with some famous examples in Section 3.1. Then we will explain the theory in more detail in Sections 3.2-3.6. Finally, we will end this chapter with an example application in Section 3.7.

3.1 Introduction and background

As stated by Sutton & Barto: "Reinforcement learning is learning what to - do how to map situations to actions - so as to maximize a numerical reward signal." Here the learning refers to an *agent* that interacts with an *environment*. The agent observes a certain *state* of the environment and performs an *action*. After performing the action, the agent receives a *reward* from the environment. These concepts are illustrated in figure 3.1 where t indicates the time step. At every discrete time step t, the agent interacts with the environment by observing the current state s_t and performing an action a_t from the set of available actions. After performing an action a_t the environment moves to a new state s_{t+1} and the agent observes a reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}). The ultimate goal of the agent is to maximize the future reward by learning from the impact of its actions on the environment. The agent is not told which actions to take, but must discover itself which actions yield the most reward by trying them. Ultimately, by trial-and-error, the agent knows

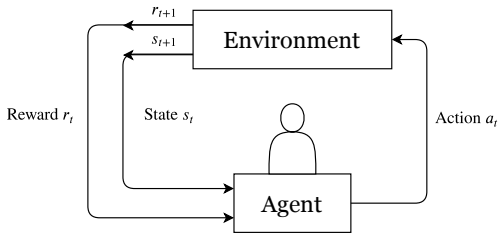


Figure 3.1: Basic concepts of reinforcement learning illustrated.

the best action to take, given the current state of the environment. It is crucial for an agent to trade-off between exploration and exploitation. To receive a high reward, the agent has to exploit what it already knows to maximize reward, but also has to explore the environment by trying actions that it has not selected before.

In contrast to general machine learning algorithms, "RL is defined not by characterizing learning methods, but by characterizing a learning problem. Any method that is well suited to solving that problem, we consider to be a reinforcement learning method." Generally, such a problem is defined as a Markov decision process, which we will explain in the next section (Section 3.2). RL is usually considered as one of the three basic machine learning paradigms, alongside supervised learning and unsupervised learning. It differs obviously from supervised learning since it does not need labeled input data. It differs from unsupervised learning, since the aim is not to explore underlying data patterns. As a matter of fact, it is generally assumed that in RL there is no predefined data at all. This is basically the case with *online learning*, where the the agent is trained as the data comes in. In *offline learning*, there is a static datasets or pool of datasets and the agent is trained in one go.

As with machine learning in general, RL is a hot topic in the field of Data Science and has been applied widely in real-life. In fact, a major part of the world population has to do with RL daily one way or another. One of the most appealing examples is self-driving cars. Other famous applications of RL are playing games like Go or chess, different kinds of recommender systems and customized action in video games.

3.2 Markov Decision Process

As discussed in Section 3.1, RL is characterized by defining a learning problem. Typically, such a problem is defined as a Markov decision process (MDP). This is a discrete-time stochastic control process, that satisfies the Markov property. That is, given the current state and action, the probability of the next state only depends on the current state and is conditionally independent of all previous states and actions:

$$P[s_{t+1} = s' | s_t, a_t] = P[s_{t+1} = s' | s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0]. \tag{3.1}$$

In the context of the agent and its environment, this means that the current state provides the agent with all the information needed in order to make a decision about which action to take next. The representation of the state basically summarizes past experiences, such that all information of the past remains. Or to put it differently, the current state completely characterizes the state of the environment. An MDP can be denoted by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$, where:

- $\mathcal{S}$ is the state-space;
- $\mathcal{A}$ is the action-space, or $\mathcal{A}(s)$ if the possible actions depend on the current state;
- $\mathcal{P}$ is the set of state-action-transition probabilities, i.e. the probabilities of transitioning to state s', given current state s and action a;
- $\mathcal{R}$ is the reward function or distribution of reward, given some state and action;
- $\gamma \in [0, 1]$ is the discount parameter (more on this in the next sections).

The state and action spaces may be finite or infinite, depending on the problem at hand. At time step $t = 0$, the environment samples initial states (usually the environment has a predefined starting state). Then at every time step until the end of the process, the agent selects an action a_t from action-space A, the environment samples a reward r_t from R and a next state s_{t+1} according to P. A policy π is a mapping from S to A that specifies which action to take in each state. The goal of the agent is to find the *optimal policy* π^* that maximizes cumulative discounted reward.

3.3 Policy

Recall that a policy π is a mapping from each state $s \in \mathcal{S}$ to each action $a \in \mathcal{A}(s)$, denoting the probability $\pi(a|s)$ of taking action a while being in state s. The policy can either be deterministic, where it returns a single action,

$$\pi(a|s) = \begin{cases} 1 & \text{for some } a^* \in \mathcal{A}, \\ 0 & \text{for all } a \in \mathcal{A} \setminus \{a^*\} \end{cases} \qquad \forall s \in S, \tag{3.2}$$

or it can be *stochastic*, where it assigns a probability to each action,

$$\pi(a|s) = P_\pi[a|s], \quad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{3.3}$$

In the first case, the agent will learn a policy where it selects one action with probability 1, in the latter case, the agent will learn probabilities where the action with a higher probability is preferred over an action with a lower probability.

3.4 Reward and value function

The goal of the agent is to find an optimal policy π^* that maximizes the expected total cumulative reward. In the simplest case, where there is a finite time horizon from 0 to T and no discount parameter, the cumulative future reward R_t at time step t is just the sum of future rewards:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T. \tag{3.4}$$

This approach is suitable for applications where there is a natural end point and the problem breaks down in episodes, such as plays of a game or trips through a maze. There are many applications, however, where there is not such a natural end point and the time horizon could be infinite. If this is the case, the above approach is not suitable, since the cumulative reward can become infinite. This is where the discount factor from Section 3.2 comes in. Instead of maximizing the expected cumulative reward, we maximize the expected discounted cumulative reward:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{3.5}$$

The discount value essentially determines how much the agent cares about rewards in the distant future relative to those in the immediate future. A reward received k steps from now, is only worth γ^{k-1} times what it would be if it were received now. In general, γ is chosen close to 1, since the access to future rewards is important for obtaining a high reward. Moreover, by introducing the

discount factor, we can prove convergence of certain algorithms (we elaborate more on this in the next sections).

By defining the cumulative reward, we can also define the value function. The value function is defined under a policy π and is defined as the expected cumulative reward at state s and following policy π thereafter:

$$V_\pi(s) = \mathbb{E}_\pi[R_t|s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right], \quad \forall s \in \mathcal{S}. \tag{3.6}$$

The value function basically assigns a value to every state $s \in \mathcal{S}$ and tells loosely speaking how good it is to be in state s. We can rewrite Equation 3.6 to the well-known *Bellman equation*:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_\pi(s')\right]. \end{aligned} \tag{3.7}$$

Similarly, we can also define the state-action value function. This is the value of taking action a in state s under policy π and denoted by $Q_\pi(a, s)$:

$$Q_\pi(a, s) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a\right]. \tag{3.8}$$

Now for finite MDPs, i.e. MDPs with a terminating state and finite state, action and reward sets, we can define the optimal state-value function. A policy π is considered to be optimal if for all other policies π', it holds that $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There might be more than one optimal policy and we define the set of all optimal policies by π_*. These policies have the same state-value function, namely the *optimal state-value function*, denoted by V_*:

$$V_*(s) = \max_\pi V_\pi(s), \quad \forall s \in \mathcal{S}. \tag{3.9}$$

Similarly, we can define the optimal state-action-value function, denoted by Q_*:

$$Q_*(s, a) = \max_\pi Q_\pi(s, a). \tag{3.10}$$

The Bellman optimality equation for the optimal state-value function is given by:

$$V_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma V_*(s')]. \tag{3.11}$$

And the Bellman equation for the optimal state-action-value function is given by

$$Q_*(s, a) = \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} Q_*(s', a')\right] \tag{3.12}$$

3.5 Q-learning

The most famous algorithm to solve the problem is Q-learning, introduced by Chris Watkins in 1989 [55]. The algorithm has a function that calculates a quality measure for every possible state action combination: $Q : S \times A \longrightarrow \mathcal{R}$. Q-learning is a model-free RL algorithm that learns the value of an action in a particular state. The value $Q(s_t, a_t)$ tells, loosely speaking, how good it is to take action a_t while being in state s_t. In the simplest setting, Q-learning iteratively updates the Q-values to obtain the final Q-table with Q-values. From this Q-table, one can read the policy of the agent by taking action a_t in every state s_t that yields the highest values. Updating is done according to the following rule:

$$Q_{t+1}(s_t, a_t) \longleftarrow (1 - \alpha)Q_t(s_t, a_t) + \alpha \left(r_t + \gamma \cdot \max_{a_t \in A} Q(s_{t+1}, a_t) \right). \tag{3.13}$$

The learned state-action function Q directly approximates the optimal function q_*. The algorithm uses several parameters/variables. Usually, the initial Q-values are set to 0. However, if we want to stimulate exploration we should pick relatively high initial Q-values. Exploration means to select a move that does not necessarily lead to the best known reward, but which gives more information about the environment. This could be a random action or an action according to some probability. The learning rate α determines how much the agent learns. A learning factor of 0 means that the agent learns nothing and Q-values are not updated at all. A value of 1 means that the agent only looks at the most recent information and forgets the old information. The discount factor γ indicates how much the agent looks at the future reward. If $\gamma = 0$, the agent only looks at the current reward. Algorithm 3 shows the pseudocode of the algorithm.

First we initialize all the relevant parameters and initial values in lines 1-2. Lines 3-11 actually show how the algorithm works. Usually, the algorithm runs for a fixed number of episodes E or until convergence of Q-values. At the start of an episode, we initialize the starting state (line 4). Then, for each time step t of the process until the terminal step T, the agent selects an action. This action is either a random action with probability ϵ or the action with the highest Q-value with probability $1 - \epsilon$ (line 6). The agent performs the chosen action, observes the reward and next state (line 7) and updates the Q-value according to the update rule (3.13) (line 8). Finally, the environment moves to the next state and a new time step starts (line 9).

The Q-learning algorithm possesses some nice property: it converges to the optimal Q-function, given that all state-action pairs are visited infinitely often:

Theorem 3.1 (Convergence of Q-values). *Given a finite MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$, the Q-learning algorithm, given by the update rule*

$$Q_{t+1}(s_t, a_t) \longleftarrow (1 - \alpha)Q_t(s_t, a_t) + \alpha \left(r_t + \gamma \cdot \max_{a_t \in A} Q(s_{t+1}, a_t) \right).$$

converges with probability 1 to the optimal Q-function, given that:

1. *The state and action spaces are finite.*
2. *$\sum_t \alpha = \infty$ and $\sum_t \alpha^2 < \infty$.*

Proof. The proof given by Melo [35] can be found in Appendix A.

In case of a relatively small problem with a finite state and action space, one can maintain a Q-table that stores the current estimate of the values. For many problems however, it becomes impossible

to maintain such a table. In this case you need to use a function approximate. This is where neural networks come in handy due to their expressive power, also referred to as Deep Q learning. In the context of influence maximization, Chen et al. [7] use a multi-layer perceptron and Kamarthi et al. [23] use a Geometric-DQN.

Algorithm 3 Standard Q-learning

Set parameters $\alpha \in (0, 1]$, small $\epsilon > 0$.
Initialize $Q(s, a)\ \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$.
for episode 1 till E **do**
  Initialize s;
  for step $t = 0, \dots, T$ **do**
    $a = \begin{cases} \text{random action } a \in \mathcal{A}(s_t) & \text{w.p. } \epsilon \\ \arg\max_{a \in \mathcal{A}(s_t)} Q(s_t, a; \Theta) & \text{w.p. } 1 - \epsilon \end{cases}$
    Take action a and observe reward r and next state s'
    $Q(s, a) \longleftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \cdot \max_{a \in A} Q(s', a)\right)$
    $s \longleftarrow s'$
  end for
end for

3.6 Deep Q-learning

Q-learning is a suitable method to solve small problems. Small in the sense of problems with relatively few different distinct states and a small action space. It does not work for problems where the state space can be very large, since maintaining a Q-table is just computationally infeasible. For example, suppose we are finding a solution for a problem on a graph with n nodes where we have to select a subset of k nodes. If we are trying to find a solution by iteratively adding a node to the set, a state is represented by the nodes selected so far. However, this means that there are a total of $\sum_{i=0}^{k} \binom{n}{i}$ distinct states and for a graph of 500 nodes and $k = 15$ this means we have $\sum_{i=0}^{15} \binom{500}{i} \approx 1.95\text{e+}28$ distinct states. Memory wise, it is of course impossible to maintain a Q-table of such a size. This is where we can use a Q-function approximator.

The function approximator, with θ denoting the function parameters, estimates the state-action-value:

$$Q(s, a; \theta) \approx Q^*(s, a). \tag{3.14}$$

Usually this function approximator is a neural network, and when it is a deep neural network, i.e. a neural network with more than one layer, the algorithm is called deep Q-learning. We will first shortly explain the concepts of a neural network before specifying the deep Q-learning algorithm. Our explanation is mainly based on the book about neural networks and deep learning by Nielsen [38] and the lecture notes for the course Deep Learning (Eindhoven University) by Menkovksi et al. [36]

3.6.1 Neural networks

In general, a neural network takes as input training data pairs (x_i, y_i), where $i + 1, \dots, N$ and tries to estimate a function $y = f(x)$. A neural network in its most simple form consists of one artificial

neuron. A graphical representation of such a neuron can be found in Figure 3.2. This neuron consists of the following components:

- The input data is represented by x_0, x_1 and x_2, also denoted as the input vector $\mathbf{x}$. These are the independent features, i.e. the features with which we want to predict the outcome y.
- Every feature is assigned a weight, represented by the edges in the figure. The weights together form the weight vector $\mathbf{w}$. The bias term is indicated by b and can be thought of as analogous to the role of a constant in a linear function. The bias and the weights together form the parameters of the neuron, denoted by θ.
- The neuron itself applies some activation function to the weighted sum plus the bias term. We denote this activation function by σ and consequently the output $o_\theta(\mathbf{x})$ will be as follows:

$$o_\theta(\mathbf{x}) = \sigma\left(\sum_i w_i x_i + b\right) = \sigma(\mathbf{w}^\top \mathbf{x} + b). \tag{3.15}$$

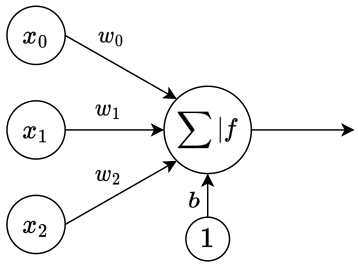


Figure 3.2: Single artificial neuron.

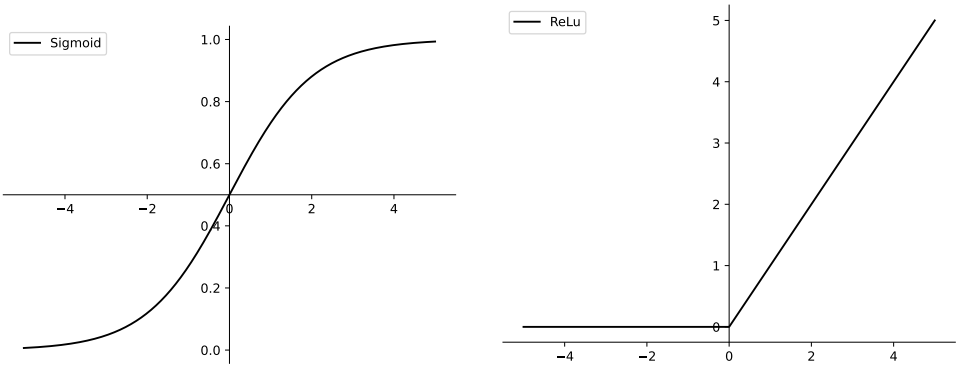
Initially sigmoidal functions such as the logistic sigmoid function (see Figure 3.3) were the most commonly used. However, they suffer from the well-known vanishing gradient problem, making it hard to train a deep model. An activation function that overcomes this problem partly and which we also use in our approach is the Rectified Linear Unit (see Figure 3.3):

$$\text{ReLU} : x \longrightarrow \max(0, x). \tag{3.16}$$

In general, ReLU's allow faster and more effective training of deep neural networks on large and complex datasets than sigmoidal functions.

A single neuron is rather restricted in its ability to capture complex maps between the input and the output of the model. For example, a neural network consisting of a single neuron with a logistic sigmoid activation function, is nothing more than a logistic regression model. To address this limitation and enable the network to represent more complex mappings, we can stack neurons. Such a stacking of neurons is called a hidden layer. Typically, these neurons have the same activation function. A structure like this is called a Multilayer Perceptron (MLP), a graphical representation of an MLP with one hidden layer is given in Figure 3.4.

To allow for even more complex mappings on non-linear data, we can stack multiple layers of neurons on top of each other. The output of one hidden layer then becomes the input to the next. A graphical representation of such an MLP with two hidden layers is given in Figure 3.5.



(a) The graph of the logistic sigmoid function.

(b) The graph of a Rectified Linear Unit.

Figure 3.3: The graphs of two commonly used activation functions.

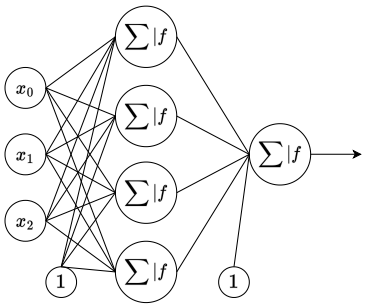


Figure 3.4: A Multi Later Perceptron with a single hidden layer.

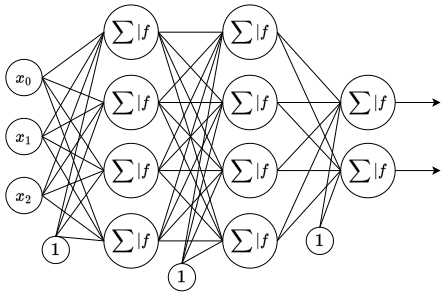


Figure 3.5: A Multilayer Perceptron with two hidden layers and two output values.

In this way, we can make the neural network as complex as possible. The number of hidden layers is called the *depth* of the network and a network with multiple layers is called a *deep neural network*. The benefit of having multiple layers is that a complex boundary between two decisions about the input data can be decomposed as a set of simpler boundaries that are then combined in the next layer. Such a highly complex network can also be achieved by a network with only one layer with a significant amount of neurons. For example, a neural network with two hidden layers of five nodes has in principle the same complexity of a network with one hidden layer consisting of ten nodes. However, this is significantly less efficient in terms of the number of parameters.

3.6.2 Training a Neural Network

In order to train the parameters of a neural network, we first need to define a loss function. Remember that we have data pairs (x_i, y_i) that serve as training data. Now the goal is to train the neural network, so that the predicted value $\hat{y}_i$ is as "close" to the real value y_i. The loss function $L(y_i, \hat{y}_i)$ will tell you how "close" these values are. To determine how good our parameters are, we are interested in the average loss, also called empirical risk:

$$\frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y}_i). \tag{3.17}$$

We want to find the parameters θ that minimize this loss. The objective of training the model then becomes to minimize the empirical risk:

$$\theta^* = \arg\min_{\theta} \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y}_i). \tag{3.18}$$

One of the most famous loss functions, the one that we will also use in our proposed method is called Mean Squared Error (MSE):

$$\text{MSE}(y, \hat{y}) = (y - \hat{y})^2. \tag{3.19}$$

Gradient Descent
In order to minimize the loss function in Equation 3.17, we actually need to use an algorithm. Although there are many more effective ways to optimize a simple linear regression model, we will focus on gradient descent because it enables us to scale this to far more complex and non-linear models, like deep neural networks, that we aim for. Gradient descent is a simple optimization process in principle. Assume we have a continuously differentiable function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ of two variables with a hilly-looking graph. We begin on some slope and want to find a local minimum. We could take a step in that direction by observing which way the hill descends most sharply. This is gradient descent: the gradient of f, $\nabla f(u, v) = (\frac{\partial}{\partial u} f(u, v), \frac{\partial}{\partial v} f(u, v)$ at a point $(u, v) \in \mathbb{R}^2$, points in the direction in which f has the steepest increase, and the negative of the gradient of f points in the direction of the steepest descent.

We consider the empirical risk, or average loss function, as a function of the model parameters in order to apply gradient descent to our machine learning model. We let the model make predictions for the dataset, calculate the loss and calculate the derivatives of that loss with respect to the model parameters. The parameters are then updated.

Gradient descent (GD) optimization is an iterative optimization algorithm for finding minima of differentiable functions. The algorithm iteratively modifies the models parameters until it converges, or until the loss value stops decreasing. Algorithm 4 provides the update rule for the parameters in an arbitrary model.

Intuitively, the gradient of the loss with respect to the model parameters indicates how the loss value changes as the parameter values change. If the loss increases as a parameter is increased, the gradient will be positive. As we want to reduce the loss, we should reduce the value of that parameter, hence the minus sign in the expression and the word 'descent' in the name of the algorithm.

A variant of GD is Stochastic Gradient Descent (SGD). While in GD, you have to run through all the samples in your training set to do a single update for a parameter in a particular iteration, in

SGD, on the other hand, you use only a subset of training samples from your training set to do the update for the parameters in a single iteration. Usually, if the number of training samples is very large, using GD may take too long because in every iteration of updating the parameters, you are running through the complete training set. Using SGD will be faster because we are only using a subset of the training samples.

Algorithm 4 Gradient Descent update rule

1: Given a set of training examples $D : \{(\mathbf{x}, y)\}$ and learning rate α
2: **repeat**
3: $\quad \mathbf{w} \longleftarrow \mathbf{w} - \alpha \nabla_w L(\mathbf{x}, y; \mathbf{w}, b)$
4: $\quad b \longleftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b)$
5: **until** convergence

Backpropagation

Calculating the gradients for a complex function like a multi-layered neural network is not straightforward. In fact, we also need an algorithm for calculating the gradients as well: backpropagation. Backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes the algorithm very scalable and hence very useful to apply it to large-scale neural networks. In order to explain how backpropagation works, we will first introduce some notation. We will use w^l_{jk} to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer. The activation a^l_j of the j^{th} neuron in the l^{th} layer is then denoted as:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{3.20}$$

where the sum goes over all neurons k in the $(l-1)^{\text{th}}$ layer. In matrix notation, where a^l contains the activations a^l_j, w^l the weights w^l_{jk} on row j and column k, and b^l the biases b^l_j, we can rewrite Equation 3.20 as:

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right). \tag{3.21}$$

Moreover, to make notation less cumbersome, we will introduce a variable for the weighted input to the neurons in layer l: $z^l \equiv w^l a^{l-1} + b^l$. As explained by Nielsen [38], backpropagation relies on 4 equations. The first equation is an equation for the error in the output layer L, denoted by δ^L. We will denote the loss (cost) function we want to minimize as C in order to prevent confusing notation. By the chain rule we have:

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{3.22}$$

Where $\nabla_a C$ is a vector whose components consist of the partial derivatives $\frac{\partial C}{\partial a^L_j}$. The equation for the error δ^l in an intermediate layer l in terms of the error in the next layer δ^{l+1} is given by:

$$\delta^l = \left(\left(w^{l+1}\right)^\top \delta^{l+1}\right) \odot s'\left(z^l\right) \tag{3.23}$$

Note that here we can already see why the algorithm is called backpropagation. Intuitively, we are moving the error backwards through the network, giving us some measure of the error at the l^{th}

layer. The equation for the rate of change of the cost function with respect to the bias term denoted as:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{3.24}$$

And lastly, the equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{3.25}$$

Algorithm 5 shows how the backpropagation equations are used to compute the gradient of the cost function.

training examples $D : \{(\mathbf{x}, y)\}$

Algorithm 5 Backpropagation Algoritm [38]

Input: Training examples $D : \{(\mathbf{x}, y)\}$

1: Set the corresponding activation a^1 for the input layer
Feedforward:
2: **for each** $l = 2, 3 \ldots, L$ **do**
3: Compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
4: **end for**
5: **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
Backpropagate the error:
6: **for each** $l = L-1, L-2, \ldots, 2$ **do**
7: Compute $\delta^l = \left(\left(w^{l+1}\right)^\top \delta^{l+1}\right) \odot s'\left(z^l\right)$
8: **end for**

Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

3.6.3 Estimating Q-function with a neural network

Remember that we are trying to estimate the Q-function by means of a neural network (Equation 3.14). Moreover, recall that the optimal Q-function satisfies the Bellman equation (Equation 3.12) and hence we want to find a Q-function that satisfies the Bellman equation. For the sake of convenience, we will rewrite the Bellman equation slightly in terms of expected values to follow the definitions by Mnih et al. [37]. For this, we will denote the (possibly stochastic) environment in which the agent operates by $\mathcal{E}$. Now we can define the Bellman equation as:

$$Q_*(s, a) = \mathbb{E}_{s', r\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a')|s, a\right] \tag{3.26}$$

We can iteratively train a neural network where the loss function is trying to minimize the error of our Bellman equation. Consequently, we can define the target value for iteration i for our neural network as follows:

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\right]. \tag{3.27}$$

The loss function is then defined as:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]. \tag{3.28}$$

Here Mnih et al. [37] call $\rho(s,a)$ the bahaviour distribution, which is a probability distribution over sequences s and actions a. Note that we are iteratively trying to make the Q-value close to the target value it should have, if Q-function corresponds to the optimal Q^*. Then the gradient update with respect to the neural network's parameters θ is given by:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a'\theta_i)\right]. \tag{3.29}$$

3.6.4 Deep Q-learning with Experience Replay

Since we have explained the core concepts of deep Q-learning now, we can explain how the basic algorithm works. It is similar to the standard Q-learning algorithm (Algorithm 3) introduced in Section 3.5, but instead of calculating the Q-values directly, we use a neural network to approximate the values. Consequently, we also have to train the network somewhere within the algorithm. The pseudocode is shown in Algorithm 6. In line 1 we initialize the *experience replay* memory and the initial weights for the neural network. The replay memory $\mathcal{M}$ is used to store transitions (state, action, reward, next state) that are used for calculating state-action value calculations and updating the weights. We will train the algorithm for a total of E episodes (lines 2-11). Similar to standard Q-learning, we will choose an action based on the ϵ-greedy policy and observe the reward and next state for T time steps (lines 4-6). We will add the observation to the replay memory (line 7). Then we sample a random minibatch of transitions of predetermined size b from the replay memory (line 8), calculate for all the transitions in this batch the corresponding target value y_j according to Equation 3.27 (line 9) and update the weights by performing a gradient descent step on the mean squared error (line 10).

The reason why experience replay is often used is that learning from batches of consecutive samples could be problematic. These samples are heavily correlated which could cause inefficient learning. For example, if the goal is to teach the agent some arbitrary game where it either moves right or left and the current best action is to move right, this will bias the upcoming examples to be dominated by samples from the right-hand side of the environment. This can lead to bad feedback loops which is not desired. Experience replay overcomes this issue by sampling across all the transitions that are stored in the memory. Moreover, each transition can also contribute to multiple weight updates which leads to greater data efficiency.

Conjecture 3.2 (Convergence of Deep Q-learning). *There is no guarantee that the function approximator $Q(s,a;\theta)$ converges to the optimal function $Q^*(s,a)$.*

Intuition. As argued by Fu et al. [16], there are no known convergence guarantees for deep Q-learning. The standard Q-learning algorithm however, converges under some assumptions to the optimal Q-function with probability 1. The main difference with deep Q-learning is that in deep Q-learning we use a function approximator since we cannot maintain a table with all the Q-values. This function approximation can be done with any parametrizable function. For example, the function approximator could just be a very simple linear function which can never accurately represent the true Q-function. Moreover, neural networks are *universal* function approximators. This means that if you have a function, you can also build a neural network that is deep or wide enough to arbitrarily accurately simulate the function. Any particular network architecture you choose, unless

it is infinitely broad or indefinitely deep, won't be able to learn all functions, though.

Algorithm 6 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{M}$ and initial Q-function with random weights
for episode $e = 1$ till E **do**
  Initialize state s_1
  for step $t = 1$ till T **do**
    $a_t = \begin{cases} \text{random action } a \in \mathcal{A}(s_t) & \text{w.p. } \epsilon \\ \arg\max_{a \in \mathcal{A}(s_t)} Q(s_t, a; \Theta) & \text{w.p. } 1 - \epsilon \end{cases}$
    Take action a_t and observe reward r_t and next state s_{t+1}
    Add tuple (s_t, a_t, r_t, s_{t+1}) to $\mathcal{M}$
    Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{M}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_a Q(s_{j+1}, a\ \Theta) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j\ \Theta))^2$
  end for
end for
return Θ

3.7 Example

To conclude this chapter, we will illustrate the concepts of RL by means of a famous example: the cart-pole game. The cart-pole and its core concepts are illustrated in Figure 3.6. The goal of the game is to balance the pole of the cart as long as possible by moving the cart to left or right. A cart-pole, also known as an inverted pendulum, is a pendulum that has its centre of mass above its pivot point. It is unstable and without additional help will fall over. There are several factors that play a role in deciding how to balance the cart: the position of the cart, the velocity of the cart, the angle of the pole and the velocity of the pole. The game terminates if one of the following two conditions holds: the pole angle is more than 12 degrees or the car hits the wall, i.e. if for the position of the cart x it holds that $x \geq 2.4$ or $x \leq 2.4$. According to Section 3.2, the first step of solving a problem by means of RL, is to translate the problem as an MDP. We need to define the state space, action space, state-action-transition probabilities and the reward function. We can do this as follows:

- $\mathcal{S}$ is the state-space and is denoted by a tuple (x, V, θ, ω) where x is the position of the cart, V the cart velocity, θ the pole angle and ω the pole velocity;
- $\mathcal{A}$ is the action-space. There are two possible actions: either move right or left (one distance unit);
- $\mathcal{P}$ is the set of state-action-transition probabilities, i.e. the probabilities of transitioning to state s', given current state s and action a. These probabilities just follow from the environment, i.e. the rules of nature;
- $\mathcal{R}$ is the reward function. The agent will receive a reward value of 1.0 for every time step it

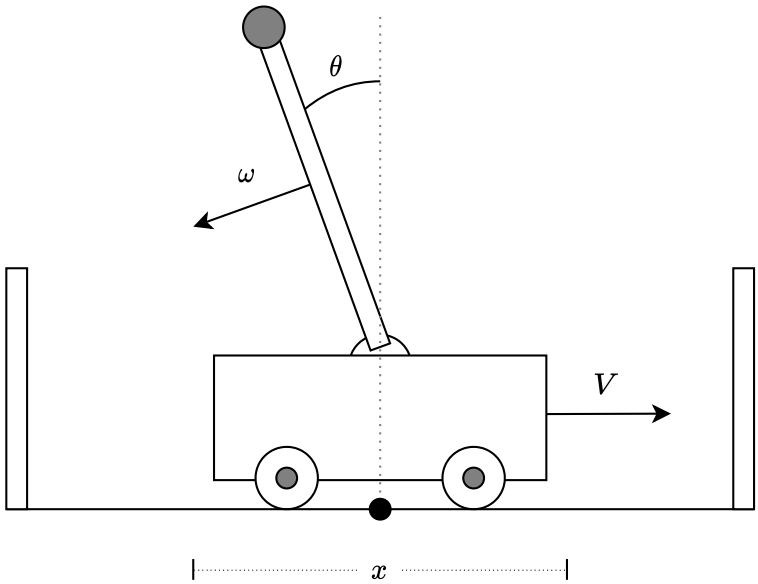


Figure 3.6: Illustration of the carptole game, where x is the position of the cart, V the cart velocity, θ the pole angle and ω the pole velocity.

manages to balance the pole and -10 if the pole falls or the cart hits the wall:

$$r(s, a, s') = \begin{cases} 1 & \text{if } \theta' \in (-12, 12) \text{ and } x' \in (-2.4, 2.4). \\ -10 & \text{otherwise} \end{cases} \qquad \forall s, s' \in S, a \in \mathcal{A}. \tag{3.30}$$

The environment of the cart-pole game and the corresponding markov decision process is implemented by OpenAI [6] and freely available online. Now since we have defined our markov decision process, we can directly train an agent to balance the pole by applying the deep Q-learning algorithm discussed in Section 3.6 to the MDP above. We will set the number of episodes to 1000, the mini-batch size to 32, the discount factor γ to 0.96, the initial ϵ to 1 with a minimum value of 0.01 and an ϵ-decay parameter of 0.001 and the learning rate for stochastic gradient descent η to 0.001. The neural network consists of an input layer of size 4, since a state is represented by 4 values. Then there are two hidden layers of size 24 with a ReLU activation function. Finally, the output layer is of size two with a linear activation function since the are two possible actions.

The training results can be found in Figures 3.7 and 3.8. Note that the this specific environment of the Cartpole game stops after 200 timesteps, so the maximum time to balance the cartpole without hitting the wall is 200. We see that the deep Q-learning algorithm with this relatively simple neural network is able to learn to balance the cartpole pretty well. At the beginning, the agent does not know anything about the environment and we can see that it is only able to balance the pole for around 20 timesteps. But the agent learns quickly and at the end of learning the agent is able to balance the pole for 200 timesteps.

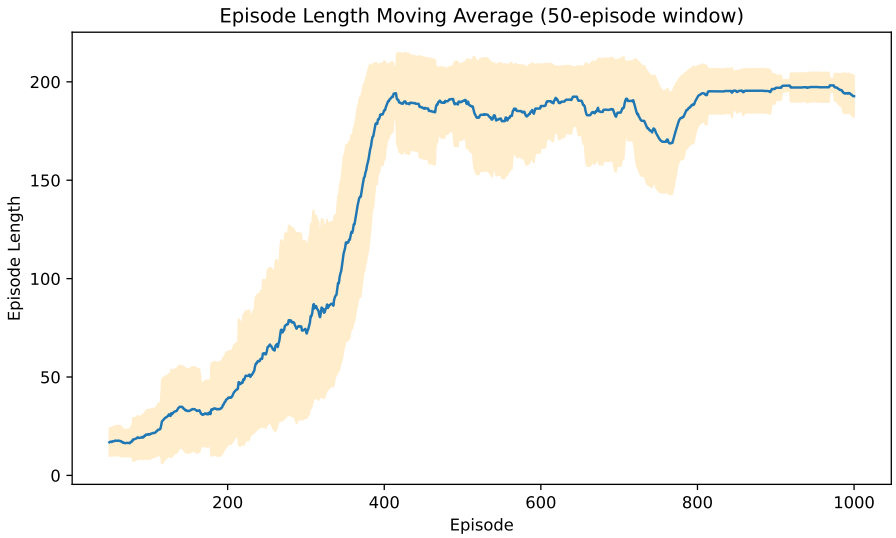


Figure 3.7: Moving average of the episode length (total reward) for the Cartpole game explained in Section 3.7 during training of the agent.

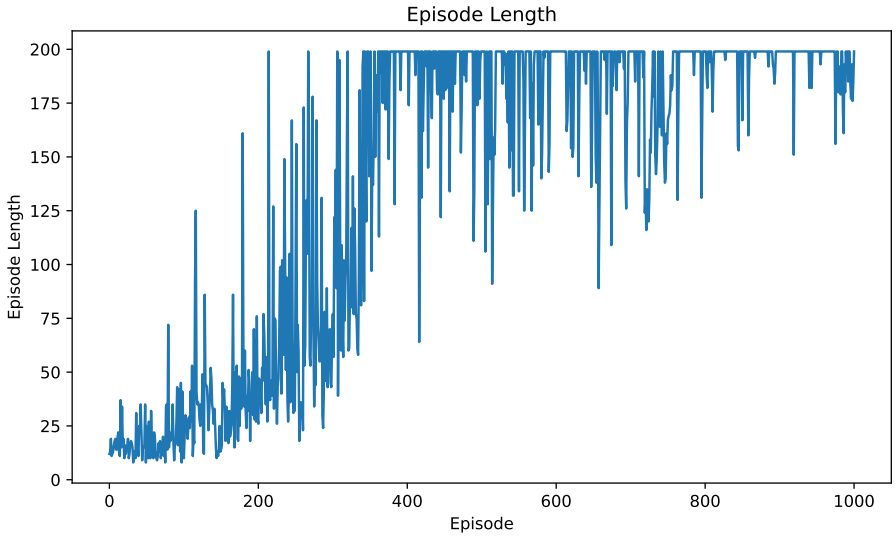


Figure 3.8: Episode length (total reward) for the Cartpole game explained in Section 3.7.

Chapter 4

Proposed Method: DQ4FairIM

In this chapter we will propose and discuss our model to find a fair solution to the Influence Maximization (IM) problem by using Reinforcement Learning (RL): DQ4FairIM. As discussed in the introduction, this algorithm trains an agent to find a general solution for a pool of graphs $\mathcal{G}$. In order to use RL for IM, we first need to formulate the IM problem as a Markov Decision Process (MDP). As mentioned before, there exists some work on RL and IM like the studies by Ali et al. [1], Kamarthi et al. [23] and Chen et al [7]. However, these do all consider a different variant of the IM problem, which makes their formulation not totally suitable for our approach. The formulation of Chen et al. [7] is closest to our formulation since its problem is closest to our problem, except that they have uncertainty about a node's willingness to be a seed node, and they do not include community information. We are looking for a fair solution to the IM problem using RL rather than finding the best greedy algorithm (Ali et al.) or exploring an unknown graph (Kamarthi et al.). We will present our MDP formulation in Section 4.1. In Section 4.2 we elaborate on how we incorporate fairness in our model. In Section 4.3 we discuss different potential node embeddings as well as the one applicable for our method. In Section 4.4 we propose our deep Q-learning algorithm and finally in Section 4.5 we discuss the training time complexity of DQ4FairIM.

4.1 MDP formulation

As discussed in Section 3.2, an MDP can be denoted by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the state-space, $\mathcal{A}$ the action space, $\mathcal{P}$ the state-action-transition probabilities, $\mathcal{R}$ the reward function and γ the discount parameter. Naturally, an MDP consists of time steps. An MDP is a discrete-time stochastic control process where at each time step, the process is in some state, an action is performed and the process goes to a next state in the next time step. Our formulation of IM as an MDP is as follows:

- **Time step:** Note that there are no natural time steps in the IM problem since there are originally no time steps involved in the problem. However, we can consider a single round setting where in each round we select a single node to be activated, a round can be seen as a time step t. Chen et al. [8] also consider a multi-round setting, where in each round B nodes are selected. The time horizon is indicated by $t = 1, \dots, T$ where $T = k$, the number of seed nodes in the starting seed set S. This means that at each time step, the agent can add a node to the seed set, which automatically ends with a seed set of k nodes. Note that this formulation means that

we are dealing with an ending time-horizon and discrete time steps.

- **State:** The current state S_t is represented by the tuple $S_t = (G, \mathcal{C}, X_t)$ where $G = (V, E)$ is a graph randomly sampled from the pool of graphs $\mathcal{G}$ at time step $t = 0$ and $\mathcal{C}$ the set of communities present in the network, which are fixed over time. We assume that G contains all the information about the structure of the graph which includes the nodes and the edges. In practice, this could mean that the edges are either represented by an edge list, an adjacency list or an adjacency matrix. $X_t \in \{0,1\}^{|V|}$ denotes the status of the nodes, i.e. which nodes are selected as seed nodes at time step t. For example, $X_t^v = 1$ means that node v is selected as a seed node at time step t. Initially, none of the nodes are selected and hence the start value $X_1^v = 0$ for all nodes $v \in V$. For implementation purposes, we could also include the community information (or any other node attributes) in the matrix X_t. In this case we could denote $X_t \in \{0,1\}^{|V|+|\mathcal{C}|}$ where $X_t^{v,i+1} = 1$ if node v belongs to community i.
- **Action:** At every time step, the agent adds a node to the seed set. Hence, the action the agent can take at time step t is denoted by the one-hot vector $a_t \in \{0,1\}^{|V|}$ where only one element of a_t corresponds to the selected node ($\sum_{v=1}^{|V|} a_t^v = 1$). Moreover, the action depends on the current state since a agent can only choose from nodes that have not been selected so far. Specifically this means that at time step t the agent can only choose a node $v \in V$ for which it holds that $X_t^v = 0$.
- **State-action-transition probability**: The state transition is deterministic and when a new node is selected at time step t, the state of the next time step is known with probability 1:

$$X_{t+1} = X_t + a_t, \quad t = 1., \dots T.$$

- **Reward:** The total reward at the end of an episode is defined as the total influence that is achieved within the social network G, given the nodes that are selected as seed nodes (represented by S). However, this would mean that the agent does not receive any reward during the intermediate time steps, but only at the terminal time step T. This makes it harder for the agent to learn efficiently, since there is the so-called issue of reward sparseness. To overcome this issue we can use the marginal influence of a node $\Delta\sigma(G, S, v) = \sigma(G, S \cup \{v\}) - \sigma(G, S)$ as the immediate reward at a given time step, denoted by $r(G, X_t, a_t)$. Recall that $\sigma(.)$ denotes the influence spread, i.e. the number of expected influenced nodes. So (without accounting for fairness), the reward at time step t is denoted by

$$r(G, X_t, a_t) = \sigma(G, S \cup v) - \sigma(G, S), \text{where} \tag{4.1}$$

$$v = \{u \in V \mid a_t^u = 1\} \text{ and } S = \{u \in V \mid X_t^u = 1\}.$$

  There is one issue here however, as discussed in Section 2.2.1: we will calculate $\sigma()$ by running m Monte-Carlo simulations. If we have to do this for every step of every episode with a certain accuracy, for example with $m = 1,000$ simulations, this will be very time costly. However, as noted earlier, an MDP is a stochastic-process and the reward can be stochastic as well. So we can reasonably estimate the reward based on a lower number of simulations, resulting in a stochastic reward. The idea is that as the agent learns over time by interacting with the environment, it can also learn the reward function.
- **Enhancing fairness:** To this end, the above formulation is suitable to model the IM problem without accounting for fairness. A straightforward way to enhance fairness in this formulation

is to not only give the agent a reward based on the influence of the selected nodes, but also on how fair the selection of the nodes is. If we want to take fairness for early-adopters into account (Equation 1.1), we can, for example, take the difference of the highest ratio and the lowest ratio among all communities and add this as negative reward. This forces the agent to seek for a fair solution, since in the fairest solution this difference would be equal to 0. Suppose we have c communities, $C_1, C_2, \dots, C_c$, then this difference can be formulated as:

$$f^{\text{ea}}(G, \mathcal{C}, X_t, a_t) = \min_{i=1,\dots,c} \frac{\sum_{v \in C_i} X_t^v + a_t^v}{|C_i|} - \max_{i=1,\dots,c} \frac{\sum_{v \in C_i} X_t^v + a_t^v}{|C_i|}. \tag{4.2}$$

In case of fairness in outreach (Equation 1.2), we should have information about the set of (expected) influenced nodes at time-step t. Denote this set by I_t, then the fairness notion we want to maximize becomes:

$$f^{\text{outreach}}(G, \mathcal{C}, X_t, a_t) = \min_{i=1,\dots,c} \frac{\sum_{v \in C_i} \mathbb{1}\{v \in I_t\}}{|C_i|} - \max_{i=1,\dots,c} \frac{\sum_{v \in C_i} \mathbb{1}\{v \in I_t\}}{|C_i|}. \tag{4.3}$$

Our final method will include a different notion of fairness, which we will discuss in more detail in the next section. Regardless of how we define the fairness measure denoted by $f(\cdot)$, we propose the reward function to be a weighted sum of the level of influence and the level of fairness. Let $\phi \geq 0, \phi \in \mathbb{R}$ be the *fairness weight*, then the total reward function at the end of an episode for a given seed set S is given by:

$$R(G, \mathcal{C}, S) = \sigma(G, S) + \phi \cdot f(G, \mathcal{C}, S). \tag{4.4}$$

This means that a higher ϕ means that we assign more importance to the fairness-measure. Again, we define the reward at time step t as the marginal gain of adding node a_t to the seed set:

$$r(G, X_t, a_t) = \sigma(G, S \cup v) - \sigma(G, S) + \phi \cdot \left(f(G, \mathcal{C}, S \cup v) - f(G, \mathcal{C}, S)\right), \text{where} \tag{4.5}$$
$$v = \{u \in V \mid a_t^u = 1\} \text{ and } S = \{u \in V \mid X_t^u = 1\}.$$

4.2 Accounting for fairness

We are interested in finding a solution that does not disproportionately exclude certain communities from the outreach. Which nodes are selected as seed nodes is not important for this work. In IM, it is about which nodes finally get activated in the network, i.e. which nodes receive the influence or information. We are looking for a fair solution in the sense that we do not want certain communities to be excluded from the information. For example, if we are able to reach 50 % of all nodes in the network, but this means that we reach 100 % of men in the network and 0 % of women, we consider this solution to be extremely unfair. A fair solution would, optimally, achieve the same influence spread of 50 % and reach both men and women equally for 50 %. Note that in the fairness of outreach formulation in Section 4.1, Equation 4.3. we are aiming to reduce the difference between the most influenced community and the least influenced community. In other words, with this fairness measure, the agent will try to find a solution where all communities are proportionally equally influenced. However, suppose that for some reason there is a community in our network of interest where the nodes belonging to this community are very badly connected. Suppose these nodes are that badly connected that given our budget k in the IM problem, in the most optimal solution we cannot reach more than 10 % of these people. Then the agent is trying to reach all communities

for only 10%, while there could be a solution where the other communities are reached for 50 % while this badly connected community is still reached for 10%. This is of course not what we want. Instead, we should use the *maxmin* fairness criterion. This criterion is also used Tsang et al. [51] to measure Group-Fairness in IM. It captures the goal of improving the outcome for the least well-off groups. It aims to maximize the minimum influence received by any of the groups, as proportional to their population. For the sake of convenience, we introduce $\sigma_{C_i}(G, S)$ to be the expected number of influenced nodes for community C_i. The maxmin fairness is then denoted as:

$$f^{\text{maxmin}}(G, \mathcal{C}, S) = \min_{i=1,\dots,c} \frac{\sigma_{C_i}(G, S)}{|C_i|}. \tag{4.6}$$

It is called maxmin, since we are trying to maximize the minimal influence received by any group. This maximizing is achieved by adding the fairness measure to the reward function. The reward at time step t for the MDP defined in the previous section is denoted as:

$$r(G, X_t, a_t) = \frac{\sigma(G, S \cup v) - \sigma(G, S)}{|V|} + \phi \left(\min_{i=1,\dots,c} \frac{\sigma_{C_i}(G, S \cup v)}{|C_i|} - \min_{i=1,\dots,c} \frac{\sigma_{C_i}(G, S)}{|C_i|} \right), \text{where} \tag{4.7}$$

$$v = \{u \in V \mid a_t^u = 1\} \text{ and } S = \{u \in V \mid X_t^u = 1\}.$$

Equation 4.7 is the reward function we will use in our proposed method DQ4FairIM. It is a weighted sum of the expected percentage of influenced nodes in the whole network and the percentage of influenced nodes of the minimally influenced group.

Definition 4.1 (SimFairIC$(G, S, \mathcal{C}, \phi, f, m)$)**.** *Given a graph $G = (V, E)$, seed set S, set of communities $\mathcal{C}$, fairness weight $\phi \geq 0, \phi \in \mathbb{R}$, fairness function f and number of simulations m, the procedure SimFairIC$(G, S, \mathcal{C}, \phi, f)$ calculates the weighted sum of the expected percentage of influenced nodes in network G and a fairness measure $f(\cdot)$ as*

$$\frac{\sigma(G, S)}{|V|} + \phi \cdot f(G, S, \mathcal{C}) \tag{4.8}$$

by running m simulations of the IC-model.

4.3 Node and graph embeddings

In order to use the graph data in our RL framework, it is essential that we can represent the nodes as embeddings so that we can feed them to our Q-function. A node embedding is a relatively low-dimensional vector representation of a node, where similar nodes have similar representations. Two of the most famous and widely used algorithms to calculate node embeddings are DeepWalk and Node2Vec. Deepwalk was introduced by Perozzi et al. [40] in 2014, and they introduced the concept of random walks to generate embeddings. Basically, the algorithm uses random walks to generate sequences of nodes and then feeds them to a skip-gram model (Word2Vec) to generate the embeddings. In 2016, Grover et al. [18] introduced Node2Vec, which uses some of the ideas presented by DeepWalk but goes a step further. The main improvement is that Node2Vec has the ability to learn representations that embed nodes from the same network community closely together, as well as to learn representations where nodes that share similar roles have similar embeddings. It can achieve

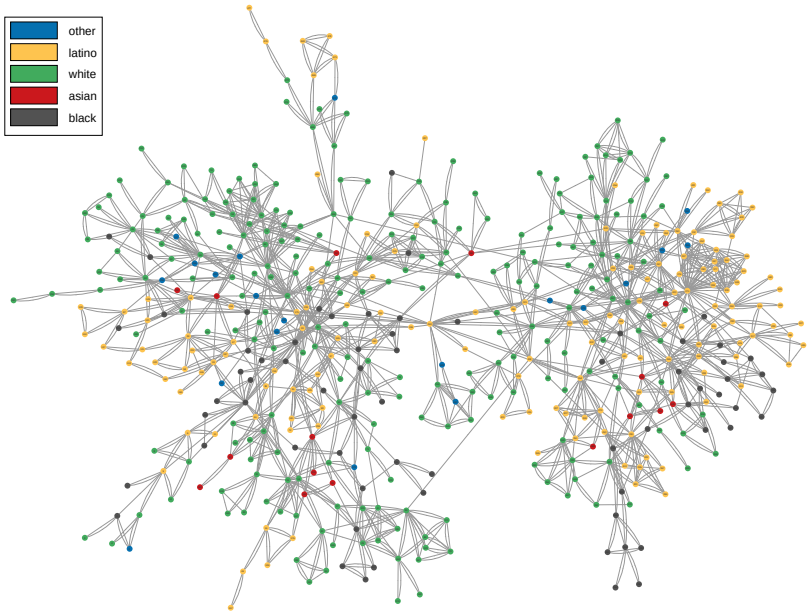


Figure 4.1: An example graph of the obesity prevention dataset.

this by using a combination of depth-first search and breadth-first search for generating the random walks. A two-dimensional projection of the 32-dimensional node embeddings generated using Node2Vec (by using this package) of the graph is shown in Figure 4.2.

However, these algorithms lack one thing for our method, which is also discussed by Li et al. [30]. Both algorithms are namely transductive. That is, we cannot use them to get embeddings for nodes that the algorithm has not seen before. This means that we would still need to execute the algorithms for every new network separately, which contradicts our main argumentation for using RL. Namely, we want to train an agent that can generalize the node embeddings for a pool of graphs. To overcome this issue, we can use another embedding method proposed by Dai et al.: Structure2vec [11]. As stated by the authors, Structure2vec is an effective and scalable approach for structured data representation based on the idea of embedding latent variable models into feature spaces, and learning such feature spaces using discriminative information. Structure2vec has two different variants denoted as DE-MF and DE-LBP, which stands for discriminative embedding using mean field or loopy belief propagation, respectively. We will use DE-MF, which was also used by Dai et al. [12] in their work about RL for combinatorial optimization problems over graphs. We will not discuss the general working of this embedding method here, but we will explain how this embedding method will be incorporated in defining the Q-function in the next section.

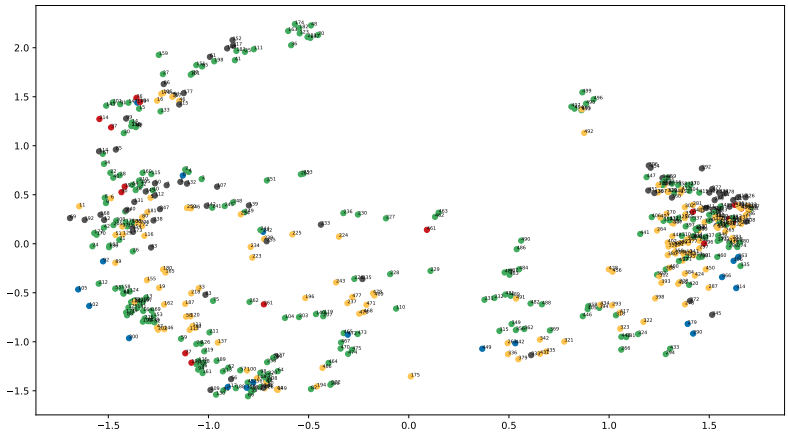


Figure 4.2: The node embeddings generated by node2vec of the graph in Figure 4.1 projected to a two dimensional space using PCA.

4.4 Deep Q-learning: DQ4FairIM

We will use a deep Q-learning algorithm to find a fair solution for the IM problem. The base of the algorithm, namely the Q-network in combination with structure2vec node embeddings, was introduced by Dai et al. [12]. This algorithm also forms the base for the RL methods proposed by Li et al. [30] and Chen et al. [23] for IM. Dai et al. introduced this method for combinatorial optimization problems over graphs in general. They showed that their framework can be applied to a diverse range of optimization problems over graphs, and learns effective algorithms for well-known problems like the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems. It uses a unique combination of the structure2vec graph embedding and rl. We will discuss how we will apply this framework to our problem and in particular to the MDP formulated in section 4.1.

Our algorithm is closely related to the standard deep Q-learning algorithm discussed in section 3.6, as it uses a deep neural network to estimate the Q-values in combination with experience replay. The main difference and complexity is in the structure of the neural network. The neural net is not just a simple feedforward neural network, but it will compute both the node embeddings and the state-action values. The embeddings are a p-dimensional vector μ_v for each node $v \in V$. These embeddings are calculated recursively according to the structure of the input graph G. Node-specific features x_v are aggregated recursively according to G's graph topology. Here x_v consists of the status of the node X_t as defined in section 4.1, and a dummy variable for the d values of the sensitive attribute. We will use the variant of structure2vec that will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as:

$$\mu_v^{(t+1)} \leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u \in \mathcal{N}(v)}; \Theta\right), \tag{4.9}$$

where $\mathcal{N}(v)$ is the set of neighbour nodes of node v and F a nonlinear mapping such as a kernel function or a neural network. The node embeddings are updated for T iterations, which results in $\mu_v^{(T)}$ containing information about the T-hop neighbourhood of node v. Let p be the embedding size, then we design F as follows:

$$\mu_v^{(t+1)} \leftarrow \text{relu}\left(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t)}\right), \tag{4.10}$$

where $\theta_1 \in \mathbb{R}^{(1+d)\times p}$ and $\theta_2 \in \mathbb{R}^{p\times p}$ are the model parameters. To calculate the Q-values for state S and action a, we take the pooled embedding over the entire graph $\sum_{u\in V} \mu_u^{(T)}$ as state representation and $\mu_a^{(T)}$ as representation for the action (node):

$$\hat{Q}(S, a; \Theta) = \theta_3^\top \text{relu}\left(\left[\theta_4 \sum_{u\in V} \mu_u^{(T)}, \theta_5 \mu_a^{(T)}\right]\right), \tag{4.11}$$

where $\theta_3 \in \mathbb{R}^{2p}$ and $\theta_4, \theta_5 \in \mathbb{R}^{p\times p}$.

The deep Q-learning algorithm, called DQ4FairIM, can be found in Algorithm 7. For notational purposes, we represent X_t here is a set containing the nodes selected in the seed set at time t, rather than a vector of zeros and ones. Note that the algorithm is quite similar to the deep Q-learning algorithm discussed in Section 3.6, but it incorporates some additional features. First of all, we make use of the so-called *Epsilon Decay* method in line 16. Recall that ϵ marks the trade-off between exploration and exploitation. Especially at the beginning of learning, we want to stimulate the agent to explore more, since it does not know anything of the environment yet. In other words, we want to have a high ϵ at the start of the algorithm. However, as the agent learns about future rewards, we

Algorithm 7 DQ4FairIM: Deep Q-learning for fair SIM

1: Initialize $\mathcal{M}$, Θ, E, ϵ
2: **for** episode $e = 1$ till E **do**
3: Draw a random graph G from pool of graphs $\mathcal{G}$, set $R_0 = 0$
4: Initialize state $S_0 = (G, \mathcal{C}, X_0)$, with seed set $X_0 = \{\}$
5: **for** step $t = 1$ till budget k **do**
6: $a_t = \begin{cases} \text{random node } v \in V \setminus S_t & \text{w.p. } \epsilon \\ \arg\max_{v\in V\setminus S_t} Q(S_t, a; \Theta) & \text{w.p. } 1-\epsilon \end{cases}$
7: Add node a_t to solution: $X_{t+1} := X_t \cup \{a_t\}$, $S_{t+1} = (G, \mathcal{C}, X_{t+1})$
8: Calculate $R_t = \textsc{SimFairIC}(G, X_{t+1}, \mathcal{C}, \phi, f^{\text{maxmin}}, m)$
9: Reward is marginal gain: $r_t = R_t - R_{t-1}$
10: Add tuple (S_t, a_t, r_t, S_{t+1}) to $\mathcal{M}$
11: **if** $(e \cdot k + t) \mod K = 0$ **then**
12: Sample random batch of transitions B of size b from $\mathcal{M}$
13: Set $y_j = \begin{cases} r_j & \text{for terminal } S_{j+1} \\ r_j + \gamma \max_a Q(S_{j+1}, a; \Theta) & \text{otherwise} \end{cases}, \quad \forall j \in B$
14: Update Θ by SGD over $(y_j - \hat{Q}(S_j, a_j; \Theta))^2$ for B
15: **end if**
16: Update exploration parameter: $\epsilon \leftarrow \max(\eta \cdot \epsilon, \epsilon_{\min})$
17: **end for**
18: **end for**
19: return Θ

want the agent to exploit the higher Q-values it has found. So we want ϵ to be lower at the end of the algorithm. This is what epsilon decay does, it decays the value of ϵ with a factor η in every episode, till some minimum value ϵ_{min}. Another difference can be found at line 11. This line indicates that we only update the weights of the neural network every K steps. This is commonly used in deep reinforcement and is just a method to speed up training, since updating the parameters in every single step of the algorithm could be quite time-consuming. An overview of all the parameters used in the algorithm can be found in Table 4.1. A graphical overview of DQ4FairIM, including description, can be found in Figure 4.3.

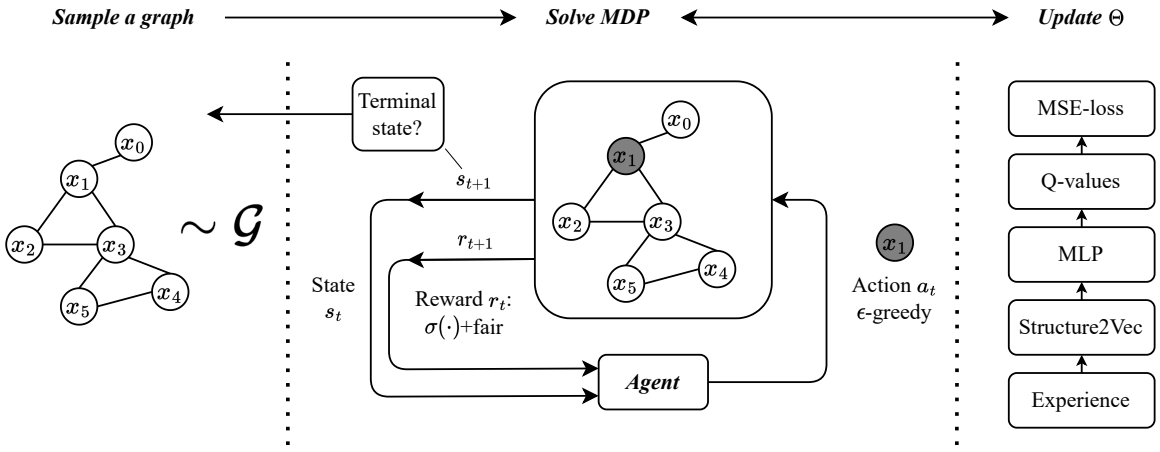


Figure 4.3: Graphical overview of the DQ4FairIM algorithm described in Algorithm 7. The process starts by selecting a graph randomly from the pool of graphs. This selected graph creates a new environment. The agent then interacts with the environment and solves the MDP defined in Section 4.1 for this specific graph. It chooses a new node (action) based on the ϵ-greedy policy: it either selects a random node or a node with the highest Q-value. The reward it receives is both based on the expected number of influenced nodes and the fairness measure. It picks a new graph at random once the terminal state is reached (k nodes are selected) and a new episode begins. Along the way, the parameters of the neural network are updated with the samples stored in the Experience Replay Memory according to the mean squared error loss. First, the current state will be parameterized to an embedding space using the Structure2Vec mechanism. These parameterized state representations will serve as input for the neural network (MLP) that estimates the Q-values.

Parameter	Description
$\mathcal{G}$	Pool of graphs to train the network on. Graphs within this pool are represented by $G_i = (V_i, E_i)$ where $i = 1, \dots \lvert\mathcal{G}\rvert$. Assumption is that these graphs come from the same distribution or at least have a similar structure.
k	Budget size, i.e. the number of nodes to select in the IM problem.
p_{uv}	Propagation probability of node u to node v.
ϕ	The fairness weight. The higher ϕ the higher the reward for the agent when it finds a fair solution (also see Equation 4.4).
E	The number of episodes for training the RL agent.
γ	Discount parameter in the RL framework. See Section 3.2.
ϵ	The probability of selecting a random node as the next action according to the ϵ-greedy policy.
η_ϵ	Cooling down parameter of ϵ.
ϵ_{min}	The lowest value for ϵ. While cooling down, the value of ϵ can never get below this value.
p	Size of the node/graph embeddings (see the beginning of this section).
Θ	The set of parameters to estimate the Q-function. These include the parameters to estimate the values for each node given the state representation, as well as the parameters for calculating the node embeddings.
$\mathcal{M}$	Replay memory. This is where the historical transitions are stored that are used to train the neural network on.
b	Size of the batch on which the parameters are updated in every episode.
α	Learning rate for stochastic gradient descent, see Section 3.6.
m	The number of (Monte-Carlo) simulations for the independent cascade model.
K	Every K steps the parameters of the function approximator are updated.

Table 4.1: Overview of parameters/notation used in DQ4FairIM.

4.5 Training time complexity

In order to get a better understanding of our algorithm and be able to evaluate the bottlenecks, we think it is convenient to do an extensive analysis of DQ4FairIM's running time complexity and propose the following theorem:

Theorem 4.2 (Time complexity DQ4FairIM). *Given that the parameters Θ are updated in every step ($K = 1$), the (training) time complexity of DQ4FairIM for a pool of graphs is $O(kE(b\cdot|V|\cdot|\Theta|+m\cdot|E|))$ where k is the budget, E the number of episodes, b the batch size, $|V|$ the number of nodes in a graph, $|E|$ the number of edges in a graph, $|\Theta|$ the number of weights for $\hat{Q}()$ and m the number of Monte-Carlo simulations for the IC model.*

Proof. First of all, notice the time complexity of DQ4FairIM per line (written between curly brackets) in Algorithm 8. We will skip the lines that take constant running time $O(1)$, since these are trivial, and explain the other lines one-by-one:

- Line 6: Picking a random node is just constant time. However, with probability $1-\epsilon$, we have to calculate the Q-value for all nodes (except the ones that are in the seed set). The number of nodes is denoted by V. The time complexity for a forward pass of the neural network depends on the complexity of the neural network, i.e. the number of weights of the neural network $|\Theta|$. Hence, for calculating the Q-value for all nodes, we get time complexity $O(|V|\cdot|\Theta|)$. From these Q-values, we have to calculate the maximum value, which is done in $O(|V|)$ time. Hence, the time complexity for this line is $O(|V|) + O(|V|\cdot|\Theta|)$.
- Line 8: since we use Monte-Carlo simulations to calculate the influence, this takes $O(m\cdot|E|)$ time (see 2.7).
- Line 13: This explanation is similar to the explanation of line 6. For every transition in B, we have to set the target value r_j, where we again have to the take the maximum value over $|V|$ Q-values. Calculating the Q-values has time complexity $O(|V|\times|\Theta|)$, selecting the max $O(|V|)$ and we have to do this b times which comes down to a time complexity of $b(O(|V|)+(|V|\cdot|\Theta|))$.
- Line 14: first of all, we have to calculate $\hat{Q}$ for every transition in B. For every transition, this is just a single forward pass of time complexity $O(|\Theta|)$. We have b transitions, so the time complexity for this operation is $b(O(|\Theta|))$. Then we have to do a step of gradient descent on these batches. Processing a single transition through backpropagation has time complexity $O(|\Theta|)$ and updating the weights also has time complexity $O(|\Theta|)$. Hence, for this line we get $b(O(\Theta|))$.

  Summing everything up, we get:

$$
\begin{aligned}
\text{Time complexity} &= O(1) + E(O(1) + O(1) + k(O(|V|) + O(|V|\cdot\Theta) + O(m\cdot|E|) + O(1) + O(1) + O(1) \\
&\quad + b(O(|V|) + O(|V|\cdot\Theta)) + bO(\Theta) + bO(\Theta) + O(1) \\
&= E(k(O(|V|) + O(|V|\cdot\Theta) + O(m\cdot|E|) + bO(|V|) + bO(|V|\cdot\Theta) + bO(\Theta) + bO(\Theta))) \\
&= kE(O(|V|(1 + b + \Theta + b\Theta)) + O(m\cdot|E|) + O(b\cdot\Theta)) \\
&= kE(O(|V|\cdot b\cdot\Theta) + O(m\cdot|E|) + O(b\Theta)) \\
&= O(ke(b\cdot|V|\cdot\Theta + m\cdot|E|))
\end{aligned}
$$

Algorithm 8 DQ4FairIM: Time complexity per line

1: Initialize $\mathcal{M}$, Θ, E, ϵ // $O(1)$
2: **for** episode $e = 1$ till E **do**
3: Draw a random graph G from pool of graphs $\mathcal{G}$, set $R_0 = 0$ // $O(1)$
4: Initialize state $S_0 = (G, \mathcal{C}, X_0)$, with seed set $X_0 = \{\}$ // $O(1)$
5: **for** step $t = 1$ till budget k **do**
6: $a_t = \begin{cases} \text{random node } v \in V \setminus S_t & \text{w.p. } \epsilon \\ \arg\max_{v \in V \setminus S_t} Q(S_t, a; \Theta) & \text{w.p. } 1 - \epsilon \end{cases}$ // $O(|V|) + O(|V| \times |\Theta|)$
7: Add node a_t to solution: $X_{t+1} := X_t \cup \{a_t\}$, $S_{t+1} = (G, \mathcal{C}, X_{t+1})$ // $O(1)$
8: Calculate $R_t = \textsc{SimFairIC}(G, X_{t+1}, \mathcal{C}, \phi, f^{\text{maxmin}}, m)$ // $O(m \cdot |E|)$
9: Reward is marginal gain: $r_t = R_t - R_{t-1}$ // $O(1)$
10: Add tuple (S_t, a_t, r_t, S_{t+1}) to $\mathcal{M}$ // $O(1)$
11: **if** $(e \cdot k + t) \mod K = 0$ **then**
12: Sample random batch of transitions B of size b from $\mathcal{M}$ // $O(1)$
13: Set $y_j = \begin{cases} r_j & \text{for terminal } S_{j+1} \\ r_j + \gamma \max_a Q(S_{j+1}, a; \Theta) & \text{otherwise} \end{cases}$, $\forall j \in B$ // $b(O(|V|) + O(|V| \times |\Theta|))$
14: Update Θ by SGD over $(y_j - \hat{Q}(S_j, a_j; \Theta))^2$ for B // $b(O(|\Theta|)) + b(O(|\Theta|))$
15: **end if**
16: Update exploration parameter: $\epsilon \leftarrow \max(\eta \cdot \epsilon, \epsilon_{\min})$ // $O(1)$
17: **end for**
18: **end for**
19: return Θ

Chapter 5

Experiments & Results

In this chapter we will evaluate the performance of DQ4FairIM and compare the results with other methods. In Section 5.1 we explain which existing data we use and how we generate synthetic graphs. In Section 5.2 we discuss some baseline methods that will be used as comparison to DQ4FairIM. The rest of the sections, Section 5.3 till Section 5.7 contain the actual experiments. They include training performance for different levels of ϕ, results on unseen graphs, results on different graph sizes, results of the model on larger graphs that is trained on small graphs, an evaluation of the Q-values and results on different types of graphs.

5.1 Datasets

5.1.1 Synthetic network generation

We will mainly use synthetically generated networks to test our algorithm on. This allows us to generate as much data as needed and give us certainty about the structure of the graphs. Two core concepts of social network generation are *preferential attachment* and *homophily*. In the context of networks, preferential attachment means that the more connected a node is, the more likely it is to receive new links. Or, to put it differently, a new node entering the network is more likely to connect with nodes having a high degree than with nodes having a lower degree. Homophily is something that individuals in social networks tend to exhibit in their social ties, namely, they prefer bonding with others of the same social group. For example, a person that identifies as woman is more likely to connect with other women than with men. These concepts are key-factors to mimic real networks and also play an important role in the study of Wang et al. [54] about information access equality on network generative models. They performed a study on the equality of information access in network models with different growth mechanisms and spreading processes. The generative network models that they used in their study will be used in our study as well. They are called *Homophily BA* and *Diversified Homophily BA*.

Homophily BA
Preferential attachment and homophily are two core concepts for the Homophily BA model. The Barabási-Albert (BA) model [3] is a well-known algorithm for generating random scale-free networks using the preferential attachment mechanism. Karimi et al. [25] combined this model with the homophily mechanism, which resulted in the Homophily BA model. As with all generative network models, nodes are added iteratively (one-by-one) to the graph until the desired number of n

nodes is reached. How new nodes connect to already existing nodes, depends on several parameters. We will assume that there are two groups in our network: the minority group and the majority group. The proportion of minority nodes is indicated by m and the proportion of majority nodes by $1 - m$, for which it holds that $m < 1 - m$. The group of node i is indicated by g_i. Each new node that enters the network, connects to a number of l nodes. The homophily parameter h denotes the probability of connecting to a node of your own group. When $h = 1$, the network is perfectly homophilic, i.e. the network consists of two distinct groups. When $h = 0$ the network is perfectly heterophilic and when $h = 0.5$ there is no homophily/heterophily. The preferential attachment strength is indicated by α, the higher α the more likely a node is to connect with nodes with a high degree. The network starts with a majority node and a minority node with one edge between them. Then, at each time step, the network grows as follows:

- A new node j enters the network. Node j is assigned to the minority group with probability m and to the majority group with probability $1 - m$.
- Node j connects with l nodes in the network according to the probability distribution Π where the probability of connecting to node i is indicated by π_i:

$$\pi_i = \frac{h_{g_j g_i} d_i^\alpha}{\sum_i h_{g_j g_i} d_i^\alpha}, \tag{5.1}$$

  where d_i is the degree of node i and

$$h_{g_j g_i} = \begin{cases} h & \text{if } g_j = g_i \\ 1 - h & \text{if } g_j \neq g_i \end{cases}. \tag{5.2}$$

Diversified Homophily BA

Wang et al. [54] proposed a variant of Homophily BA: Diversified Homophily BA. Their motivation for this model is to encourage inter-group connections while maintaining some degree of homophily. It uses, apart from the Homophily BA model, two extra parameters: l_d is the number of diversified edges for each node and p_d is the diversification probability. The network grows as follows at each time step:

- A new node j enters the network. Node j is assigned to the minority group with probability m and to the majority group with probability $1 - m$.
- Node j forms $l - l_d$ connections according to the Homophily BA mechanism, Equation 5.1. The nodes to which node j connects at this step are denoted by S_j.
- Node j forms l_d diversified links. The connecting probability of two nodes j and k is defined as:

$$p_{jk} = \begin{cases} p_d, & \text{if } g_j \neq g_k \\ 1 - p_d & \text{if } g_j = g_k \end{cases}. \tag{5.3}$$

  Now, for each node $i \in S_j$, we obtain their neighbours denoted as N_{S_j}. Node j connects to l_d nodes from N_{S_j} with the probability of connecting to node $k \in N_{S_j}$ denoted by $\Pi_{jk} \propto p_{jk} \times \frac{1}{|d_k - d_i|}$, where d_i is the degree of node $i \in S_j$ and is node k's neighbour.

Lancichinetti–Fortunato–Radicchi benchmark

The Lancichinetti–Fortunato–Radicchi (LFR) benchmark is an algorithm, introduced by Lancichinetti

et al. [27], to generate benchmark networks for community detection. These networks are designed to have a structure that does reflect the real properties of nodes and communities found in real networks. The algorithm proceeds as follows:

- **Step 1**: Generate a graph of size n and find a degree sequence for the nodes with a power law distribution with exponent γ, minimum value $k_{\min}$ and maximum value $k_{\max}$ to have approximate average degree k. Each node u will have $\mu\deg(u)$ edges joining it to nodes in communities other than its own and $(1-\mu)\deg(u)$ edges joining it to nodes in its own community.

- **Step 2**: Generate communities with sizes according to a power law distribution with exponent τ. The sum of all sizes must be equal to n. The minimal and maximal community sizes are denoted by $s_{\min}$ and $s_{\max}$ respectively.

- **Step 3:** Each node u will be assigned to a community at random, given that the community is large enough for the node's intra-community degree $(1\mu)\deg(u)$. If a community grows too large, a random node will be selected for reassignment to a new community, until all nodes have been assigned a community.

- **Step 4:** Finally, each node u creates $(1-\mu)\deg(u)$ intra-community edges and $\mu\deg(u)$ inter-community edges.

5.1.2 Existing Datasets

Obesity Prevention dataset
This dataset was introduced by Wilder et al. [56] to model an obesity prevention intervention in the Antelope Valley region of California. The data was used by Tsang et al. [51] who studied group fairness in influence maximization and also by Becket et al. to study fairness through randomization [4]. The dataset consists of 24 graphs and each graph has 500 nodes. Each node has several sensitive attributes like a geographic region, ethnicity, and gender, which makes it suitable for fairness evaluation. Moreover, the data is publicly available [here].

5.1.3 Experiment setup

We will generate graphs using the Homophily BA method discussed in Section 5.7. These graphs with two communities will be the main graphs that we use in our experiments to evaluate DQ4FairIM. We will set $m = 0.25$, $l = 4$, $h = 0.8$, $\alpha = 0.2$ and vary n (the number of nodes that enter the network). We will generate datasets consisting of 60 graphs: 50 training graphs and 10 test graphs. Note that these graphs contain the same number of nodes. They do differ in the number of edges, since the implementation of the network generation method by Wang et al. [54] uses sampling with replacement for the distribution of Equation 5.1. This means that a new node connects to at most l nodes instead of exactly l nodes. Note that we initially start with two nodes, one belonging to the minority group and one belonging to the majority group, so we end up with graphs of $n + 2$ nodes. We will refer to these datasets with BA[number_of_nodes]. So for example, BA100 refers to the pool of graphs generated according to the Homophily BA principle with the settings above, and consists of 102 nodes. An example of such a graph can be found in Figure 5.1.

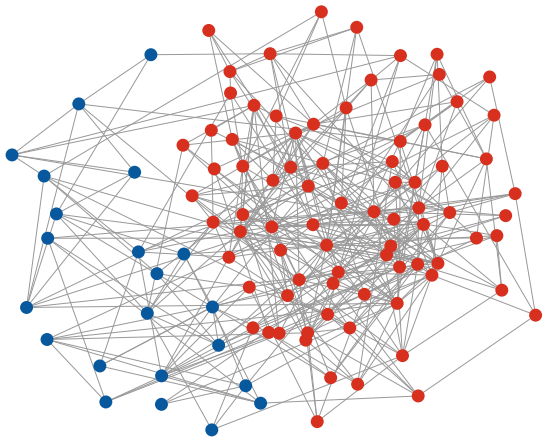


Figure 5.1: Graph generated by Homophily BA mechanism, red nodes are the majority group (75 %) and blue nodes are the minority group (25 %).

Similarly, we generate datasets consisting of 60 graphs using the diversified homophily BA principle, with the same settings as for the homophily BA datasets, and with diversification probability p_d = 0.6. We refer to these datasets as dBA[number_of_nodes].

To generate the LFR benchmark networks, we use the following settings: network size n = 250, γ = 5, τ = 1.1, μ = 0.1, k = 10 ($k_{\min}$ = 0 and $k_{\max}$ = n), $s_{\min}$ = 75 and $s_{\max}$ = 220. Again, we generate 50 train graphs and 10 test graphs. We will refer to these as LFR250.

We will refer to the obesity prevention dataset as *obesity*. We split the 24 graphs up in 20 training graphs and 4 test graphs and train our algorithm on the attribute *gender*.

5.2 Baseline Methods

To be able to evaluate the performance of DQ4FairIM, we will compare the results with other algorithms, referred to as baseline methods. We will implement a variant of the greedy algorithm called CELF, which was discussed in Section 1.1 and the diversity seeding method from Stoica et al. [49].

Cost Effective Lazy Forward (CELF)
The CELF algorithm was shortly mentioned in the introduction and works, in principle, the same as the greedy algorithm. However, it exploits the submodularity property of the influence function, making it much faster than the greedy algorithm. The resulting seed set is the same for both algorithms. Recall that for the greedy algorithm (Algorithm 1) we have to calculate the expected spread for all nodes in every iteration, forcing us to run a lot of simulations, which costs a significant amount of computing time. CELF only calculates the spread for all nodes in the first round and then stores them in a list/heap, which is then sorted. Logically, the top node is added to the seed set in the first iteration, and then removed from the list/heap. In the next iteration, only the spread for the top node is calculated. If, after resorting, that node remains at the top of the list/heap, then it must have the highest marginal gain of all nodes and is added to the seed set. If not, then the marginal spread of the new top node is evaluated and so on, until all k nodes have been selected. The pseudocode of the algorithm can be found is shown in Algorithm 9.

Diversity seeding from Stoica et al.
We will compare our method with the different seeding strategies provided by Stoica et al. [49]. The purpose of these different strategies was to find a fairer solution in biased networks, which makes it a suitable method to compare with. We assume that there is a bi-populated network, i.e. a network with two communities: a majority and minority community. Stoica et al. make a distinction between blue and red nodes, but we will refer to nodes belonging to group 0 (V^0) or belonging to group 1 (V^1). The seeding methods are based on degree centrality, where the degree of a node is the number of connections it has with other nodes (neighbours). The baseline agnostic strategy is similar to a greedy method, and selects the top-k nodes with the highest degree. For notation purposes, we denote $t(k)$ as the degree threshold for which we select k nodes:

Definition 5.1 (**Agnostic seeding**). *The baseline agnostic seeding defines the seed set of a bi-populated network $G = (V, E)$ of majority (0) and minority (1) nodes as $S_{t(k)} = \{v \in V | deg(v) \geq t(k)\}$.*

Parity seeding is defined by increasing the threshold for the majority nodes and decreasing it for the minority nodes in order to achieve the same ratio of majority and minority nodes in the seed set as it is in the general population, while preserving the seed set budget. In other words, this method focuses on fairness in the seed set (fairness of early adopters) in order to find a fairer solution regarding total outreach:

Definition 5.2 (**Parity seeding**). *Parity seeding defines the seed set of a bi-populated network $G = (V, E)$ of majority (0) and minority (1) nodes based on two differentiated thresholds $t^0(k)$ and $t^1(k)$ as $S^0_{t^0(k)} \cup S^1_{t^1(k)} = \{v \in V^0 | deg(v) \geq t^0(k)\} \cup \{v \in V^1 | deg(v) \geq t^1(k)\}$ such that $|S_{t(k)}| = |S^0_{t^0(k)} \cup S^1_{t^1(k)}|$ and $\frac{|S^0_{t^0(k)}|}{|S_{t(k)}|} = \frac{|V^0|}{|V|}$.*

Algorithm 9 CELF [28]

Input: Graph $G = (V, E)$, budget k, number of simulations M
Output: Maximum influence set $S \subseteq V$, $|S| = k$

for all $v \in V$ **do** $a_v \leftarrow$ SimulateIC($\{v\}, G, M$)
select $u = \arg\max_{u \in V} a_v$
$S \leftarrow S \cup u$, $I = a_v$
for all $v \in V \setminus S$ **do** Insert (a_v, v) into max-heap H
for $i = 2, \ldots, k$ **do**
    $matched$ = False
    while not $matched$ **do**
        v = top item in H
        $a_v \leftarrow$ SimulateIC($S \cup \{v\}, G, M$) $- I$
        Update the value of v in H to a_v
        if v = still top item in H **then**
            $matched$ = True
        end if
    end while
    $S \leftarrow S \cup v$, $I \leftarrow I + a_v$
end for
return S

5.3 Training for different levels of fairness

The first thing we are actually interested in, is to see how DQ4FairIM performs for different levels of ϕ. By starting with $\phi = 0$ and gradually increasing it, we can see how the agent changes its behaviour based on the fairness measure and if it actually finds a fairer solution for larger ϕ. We will start with small graphs and run this experiment on the BA100 instances. Recall that we train the model first on the 50 test graphs and then test the performance of the agent on the 10 unseen test graphs.

We proceed as follows. We will train five different models for five different levels of fairness: $\phi \in [0, 0.25, 0.5, 0.75, 1]$ using the maxmin fairness defined in Equation 4.6. Except for the value of ϕ we use the same parameters for all models. First of all, the propagation probability for the IC-model is set to $p = 0.1$ and the budget $k = 7$. We train the model for 700 episodes, with a batch size of 32, discount parameter $\gamma = 1$ (since we are dealing with a terminating task), initial $\epsilon = 1$, epsilon decay $\eta_\epsilon = 0.995$, minimum value $\epsilon_{\text{min}} = 0.05$, learning rate for stochastic gradient descent $\eta = 0.001$, embedding size is 64 and number of iterations for the node embeddings is $T = 5$. An overview of all the parameter settings and a motivation of our choice can be found in Table 5.1.

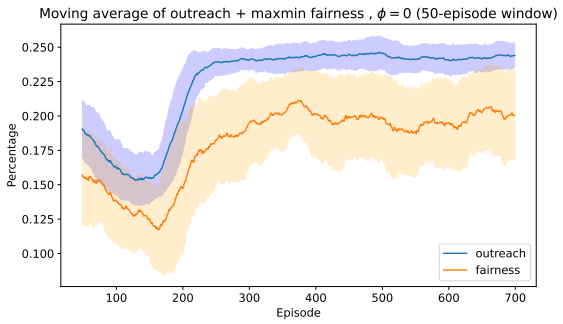
Parameter	Value	Explanation
k	7	The number of seed nodes is usually a very small percentage of the whole population. 7 % of the nodes as seed node is relatively large, but for the purpose of the experiment it does not really matter.
p_{uv}	0.1	Based on literature, this is a common probability to pick for graphs of this size. Usually, when networks are much larger, the probability is smaller.
E	700	The agent did not seem to learn any big things anymore after a while. So training it longer than 700 episodes would only lead to potential overfitting.
γ	1	We are dealing with an episodic task, so $\gamma = 1$ is suitable here.
ϵ	1	At the start, the agent does not know anything and needs to explore the environment. By setting ϵ we stimulate exploration at the beginning of training.
η_ϵ	0.9975	We are doing a lot of steps (4900), so this is a normal cooling down parameter.
ϵ_{min}	0.05	In later stages of training we want the agent to exploit the environment, but still want to do some exploration.
p	64	Size of the node/graph embeddings (see the beginning of this section).
α	0.001	Learning rate for stochastic gradient descent, see Section 3.6.
m	100	For intermediate rewards we do only 10 simulations, but at the end of the episode we run 100 simulations. The number of simulations are quite small, but by keeping it small the training will go faster. Moreover, the idea is that the agent will see a lot of steps and learns over time.
K	1	We update the parameters Θ in every time step.

Table 5.1: Model set up for experiment in Section 5.3.

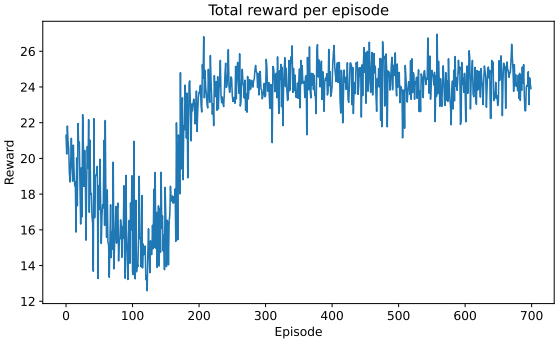
5.3.1 Performance during training

In order to compare the effect of different levels of ϕ on the unseen graphs, we have plotted the outcomes for the different levels in Figure 5.2. With outcome, we refer to the total reward the agent receives at the end of an episode, i.e. when it has selected k nodes for the seed set. Each row contains two figures, on the left side we make a clear distinction between the total spread (expected percentage of influenced nodes) in blue, and the maxmin fairness in orange. In order to make the lines

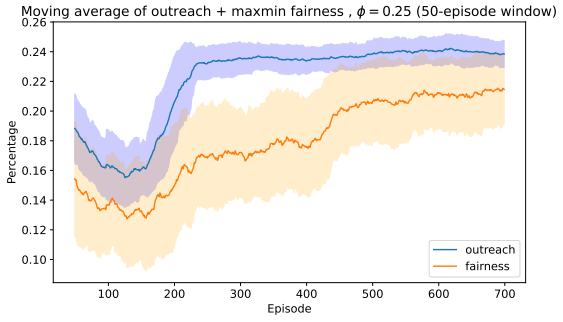
more smooth and readable, we have plotted the rolling mean of a 50-episode window along with the standard deviation, depicted by the shaded area. On the right side, we have plotted the total reward per episode as the sum of the total spread plus ϕ times the maxmin fairness, without averaging over a number of episodes. Note that these graphs are very volatile, since we are dealing with a probabilistic environment where the agent samples a random graph at the beginning of the episode. Basically, it solves the influence maximization problem for a different graph in every episode. Hence, the reward the agent is able to receive heavily depends on the instance of the graph at hand.



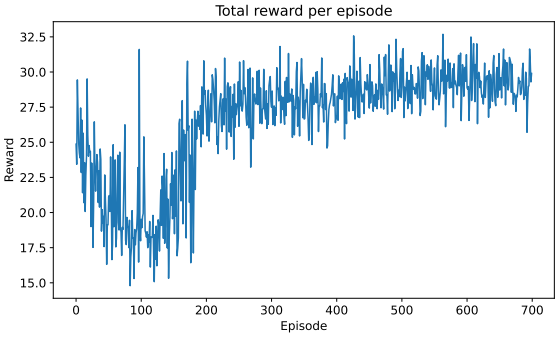
(a) ϕ = 0



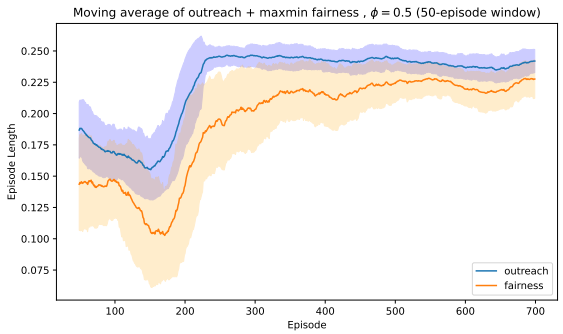
(b) ϕ = 0



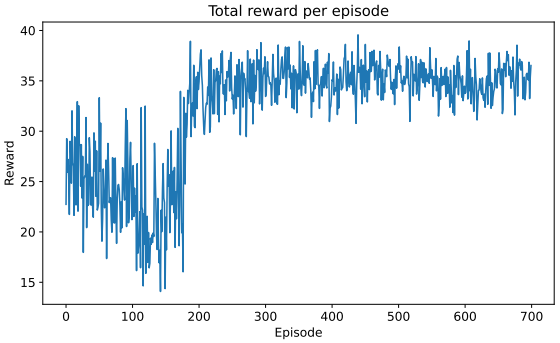
(c) ϕ = 0.25



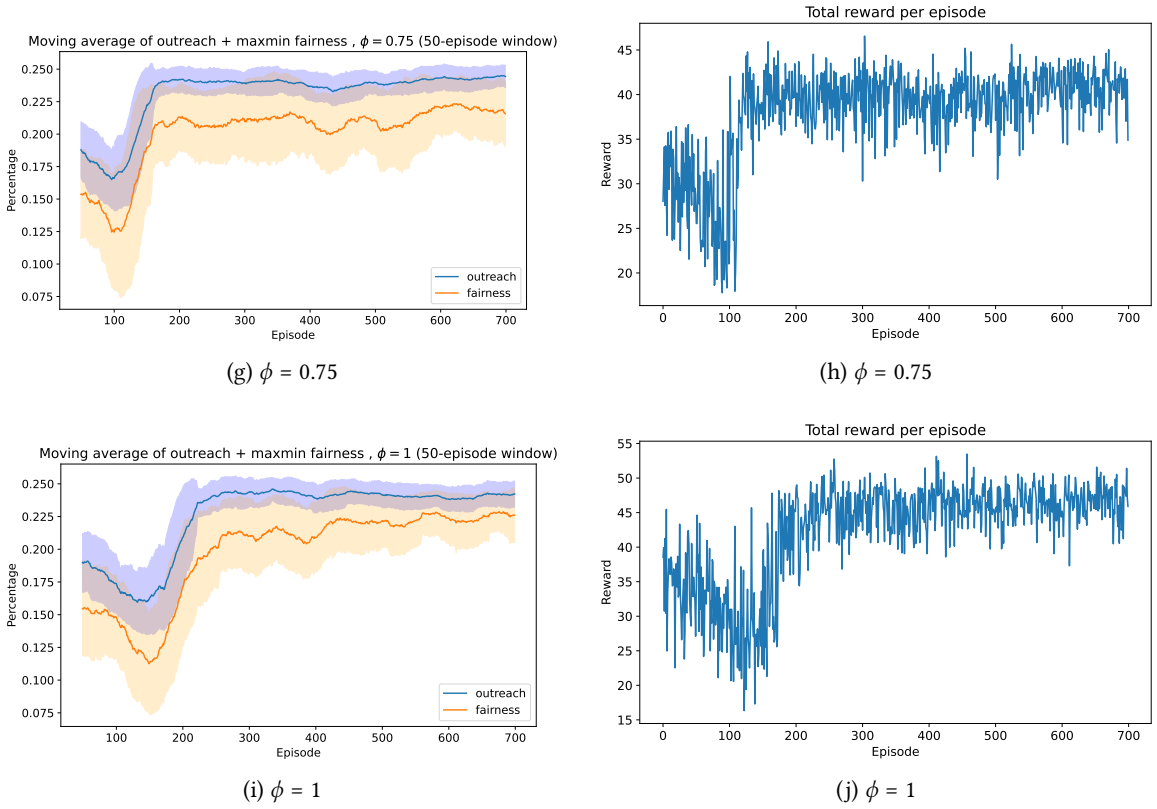
(d) ϕ = 0.25



(e) ϕ = 0.5



(f) ϕ = 0.5



(g) ϕ = 0.75 (h) ϕ = 0.75

(i) ϕ = 1 (j) ϕ = 1

Figure 5.1: Training progress DQ4FairIM for different levels of ϕ.

All the graphs for the different levels of ϕ share a common pattern: in the early episodes the agent does not know anything about the environment and selects nodes mostly random, which to an arbitrary bad reward, but after some time, around episode 150, we see that the reward goes up significantly and after a while it seems to converge to an average reward. This is not surprising, as a matter of fact, this is what we expect from a RL algorithm. If we had observed a different pattern, it would have been an indication that our model was doing something wrong. Moreover, we can also see a clear difference between the training patterns of the different models.

For example, if we look at the averaged reward for $\phi = 0$, we notice that the maxmin fairness increases naturally when the total outreach increases, but it does not seem to get any 'closer' to the total outreach over time and moreover, it stays very volatile over time, i.e. the shaded area is quite big. This is in clear contrast with training progress of the agent in the model where $\phi = 1$ (Figure (i)). Here, the minimum fraction keeps increasing over time, while the total spread keeps stable. Also, the average maxmin fairness and average total outreach are pretty close together at the end of training, and we can clearly see that the minimum fraction is less volatile as well. This indicates that the agent is actually able to learn a fairer solution for the IM problem by adding a fairness objective in its reward function. If we look at the graphs for the other levels of ϕ, we do not observe a remarkable difference between $\phi \in [0.5, 0.75, 1]$, indicating that it does not matter whether we set ϕ to 0.5, 0.75 or 1. There seems to be a slight difference between $\phi = 0$ and $\phi = 0.25$: when $\phi = 0$ the agent does not care about fairness at all, whereas with $\phi = 0.25$, the minimum fraction seems to increase slightly halfway during training (around episode 400). Based on training, we can

clearly see that DQ4FairIM gives better solutions in terms of fairness. In the next subsection, we look at how the model performs on unseen graphs.

5.3.2 Performance on unseen graphs

We trained the model for different levels of ϕ for a pool of 50 graphs. What we are mainly interested in, is to see how the agent performs on the 10 test graphs that it has not seen before. It constructs a solution by just iteratively selecting the node with the highest Q-value based on the function approximator $\hat{Q}(s, a, \Theta)$, where the function parameters Θ have been learned during training. The results for the different models, the greedy algorithm CELF, agnostic seeding and the parity seeding by Stoica et al., averaged over the 10 test graphs together with the standard deviation, can be found in Figure 5.2. What we notice firstly, is that the total influence spread is quite similar for all the algorithms, although CELF and $\phi = 0$ are slightly performing better compared to the other algorithms on these graphs. A reason for this could be that under the IC model, with a relatively high propagation probability of $p = 0.1$, it might be relatively easy to achieve high outreach. This suggestion is strengthened by the fact at the start of training (see Figures 5.2), when the agent selects nodes purely at random, it already is able to get an influence spread close to 0.19, while ending up with an average spread of 0.24. The maxmin fairness, i.e. the fraction of the minimally influenced group, however, differs more for the different algorithms. We observe that $\phi = 1$ performs best in terms of fairness, outperforming $\phi = 0.75$ and parity only slightly. Another remarkable thing to notice is that the volatility for the minimum fraction is very low for parity seeding, $\phi = 0.75$ and $\phi = 1$, while it is quite high for the other algorithms.

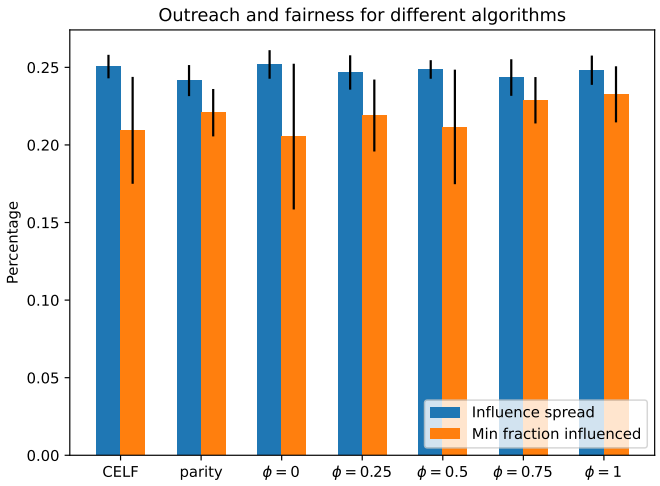
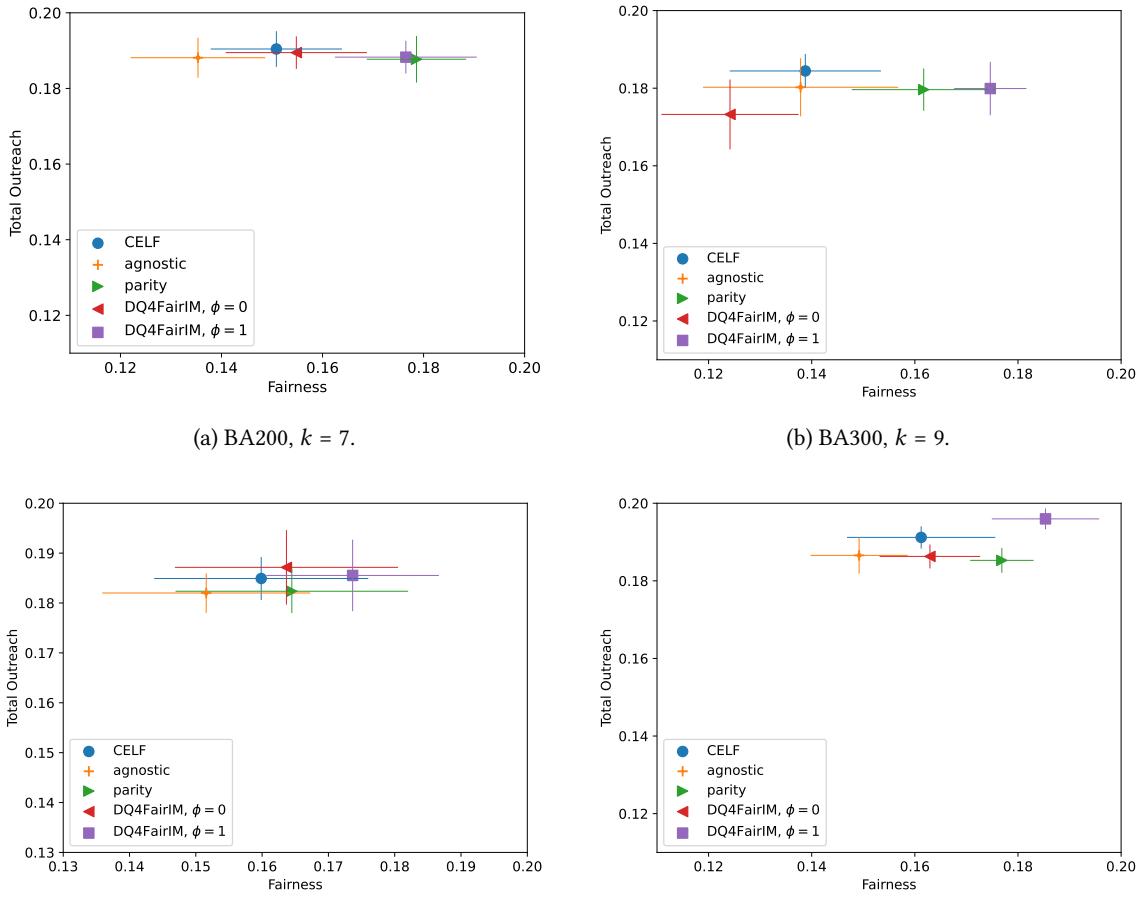


Figure 5.2: DQ4FairIM results for different levels of ϕ averaged over 10 test graphs.

5.4 Increasing Graph Size

We evaluated the performance of DQ4FairIM for different levels of ϕ on graphs of 100 nodes and obtained pretty good results. But what if we increase the size of the graphs? Does our method still perform well? To find out, we perform the same experiment as in the previous section. We have trained models with $\phi = 0$ and $\phi = 1$ on pools of 50 graphs with 200, 300, 400 and 500 nodes respectively, and tested them on test sets of 10 graphs. The results for the different graph pools can be found in Figure 5.3. Note that compared with the parameter settings in Table 5.1, the only difference is the value for k. We observe similar results as for the graphs of 100 nodes, this means that $\phi = 1$ is clearly performing better in terms of fairness than $\phi = 0$, and the influence spread is relatively similar for all methods. If we compare DQ4FairIM with parity seeding, we see that in terms of fairness DQ4FairIM is doing better on all datasets, except for BA200, where Stoica's parity seeding is slightly doing better. Overall, we observe that there is no big difference between the results on the different datasets with different graph sizes.



(a) BA200, $k = 7$.

(b) BA300, $k = 9$.

(c) BA400, $k = 12$.

(d) BA500, $k = 15$.

Figure 5.3: Results of DQ4FairIM for different graph sizes.

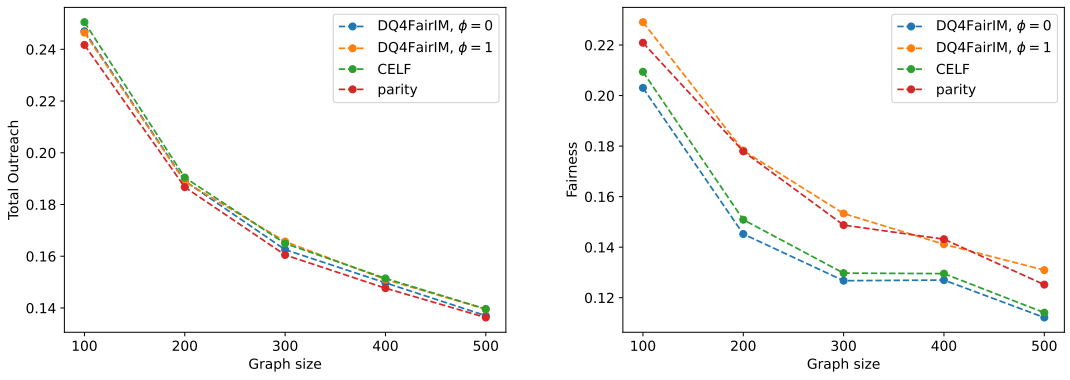
Hence, we can conclude that regardless of the graph size, our algorithm gives good results. Of course, as stated by Theorem 4.2, the training time for DQ4FairIM will increase significantly with a larger

graph size. For example, we can compare the running time complexity of the dataset with graphs of 100 nodes and budget $k = 7$, with graphs of 500 nodes and budget $k = 15$. The graphs with 500 nodes have 5 times more nodes and around 5 times more edges as well. This means that the total training time is around $5 \cdot \frac{15}{7} = 10.7$ times higher.

5.5 Extending to larger graphs

We can possibly reduce training time for DQ4FairIM significantly. Namely, a great advantage of DQ4FairIM is that the trained Q-function approximator does not depend on graph sizes. As discussed, the training speed depends heavily on the graph size as shown by Theorem 4.2, but the learned Q-function approximation can take any graph size as input. This means that in principle, we could train an agent on small graphs of 100 nodes and use the trained Q-network to obtain a solution for graphs with more nodes. In this section, we will investigate whether an agent that is trained on small graphs is able to give good solutions for larger graphs too. We proceed as follows, we use the DQ4FairIM models with $\phi = 0$ and $\phi = 1$ from Section 5.3 that are trained on a pool of graphs of 100 nodes with $k = 7$ to generate a solution for increasing graph sizes with the same budget. We have plotted the results for both influence spread and maxmin fairness in Figure 5.4. Remarkably, all methods considered are performing equally well in terms of influence spread. In terms of fairness, CELF and DQ4FairIM with $\phi = 0$ are performing equally bad, while DQ4FairIM with $\phi = 1$ and Stoica's method are getting similar good results.

We think these results are very promising. It shows that the agent is able to generalize well to larger graphs by interacting with an environment of small graphs. Of course, we have assumed that the graphs are coming from the same distribution and are similar in number of communities, balance of communities and structure. But nevertheless, the results show that if we want to train an agent in finding a fair solution for a large network, we can train it on small representations of this large network, for example subsets, with the same budget constraint, and are able to obtain good results for these large networks. The biggest advantage of this, is that we can obtain good results in terms of fairness and total outreach, while keeping the training time low.



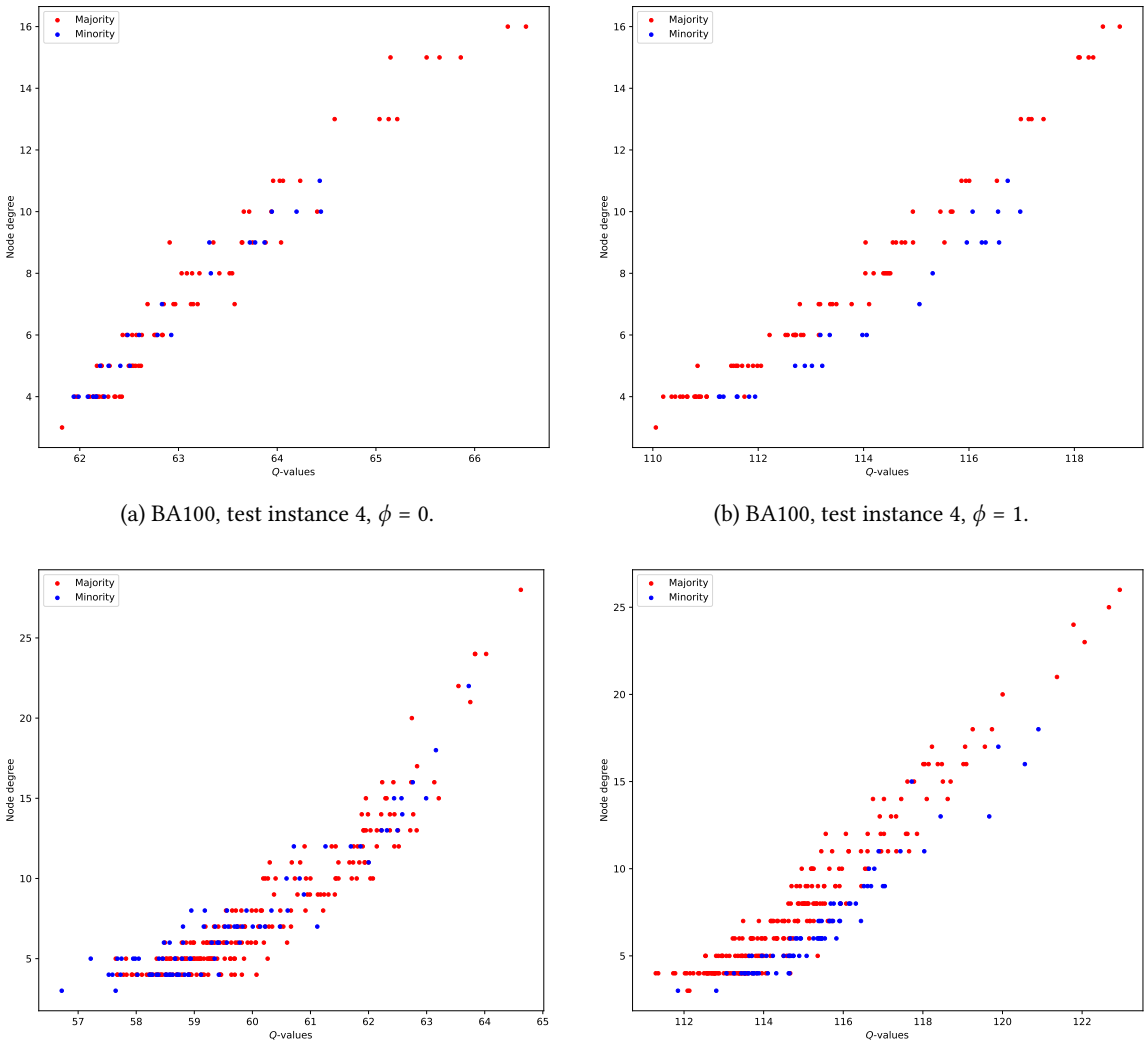
(a) Influence spread.

(b) Fairness.

Figure 5.4: Results of CELF, diversity seeding and the DQ4FairIM model trained on graphs of 100 nodes for different graph-sizes, $k = 7$.

5.6 Evaluating Q-values

So far, we have looked at our model in terms of performance on both training data and test data. In order to do a better examination of what the agent actually learns, we should examine the models in more detail. We can do this, for example, by looking at the Q-values for each node. Note that a higher Q-value for a node, means that the agent 'ranks' this node higher. We proceed as follows, we calculate the Q-values for a given test graph based on the trained models for $\phi = 0$ and $\phi = 1$ at the start state (so when the seed set is empty) and plot these Q-values together with the degree of the node in Figure 5.5. To keep things organized, we have only plotted the results for two graph instances: one for BA100 and one for BA300. What we notice immediately is that there is a clear



(a) BA100, test instance 4, $\phi = 0$.

(b) BA100, test instance 4, $\phi = 1$.

(c) BA300, test instance 1, $\phi = 0$.

(d) BA300, test instance 1, $\phi = 1$.

Figure 5.5: Node degree and Q-value at the start for different graphs and models.

correlation between Q-value and node degree. The agent seems to be able to learn the structure of the graph and ranks nodes with many neighbours higher than nodes with less neighbours. This

seems reasonable: in general, nodes with a lot of connections are good nodes to add to the seed set. Another interesting thing we observe is that when $\phi = 0$ the agent does not really seem to make a distinction between majority (red) and minority (blue) nodes. However, when $\phi = 1$, there is still a strong correlation between node degree and Q-value, but the agent rates minority nodes higher than majority nodes with a similar node degree. This is very interesting to notice. We already saw that by adding fairness in the reward function, the agent is able to learn a fairer solution and now we can see why. It is able to identify the minority nodes and assign them a higher Q-value in order to come to a fairer solution.

5.7 Training on other graph types

All the results we have reported so far are based on one type of graph, namely graphs that were generated according to the Homophily BA principle. These graphs all have two communities that are somewhat isolated in the network. This means that the nodes in the network have more intra-community links than inter-community links, which makes them very interesting to test our method on. Similar to these graphs are the LFR graphs, where the degree distribution and community sizes are generated according to a power law distribution. We have plotted the results for these graphs in Figure 5.6. We actually observe the same results as for the homophily BA graphs: all methods perform similar in terms of outreach, but DQ4FairIM and parity seeding are much better in terms of fairness.

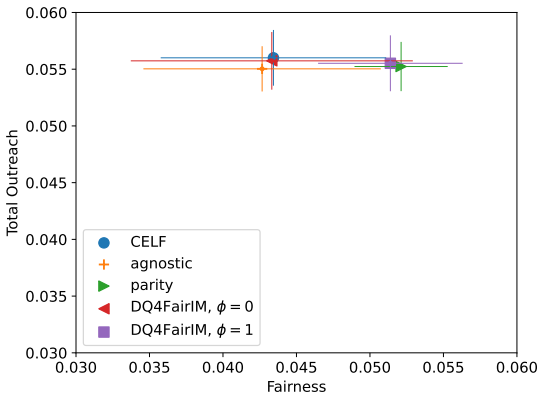
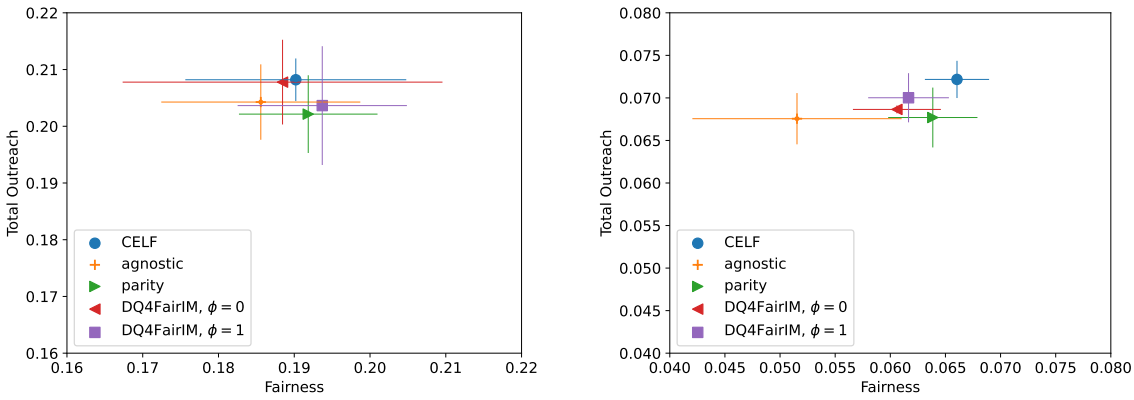


Figure 5.6: Results of different algorithms for LFR250, $k = 8$, $p = 0.1$.

But what about other graph types? For example, graphs where fairness could be less of an issue because of the structure of the graph. If communities are not isolated in a network, meaning that people are as connected with other people in their community as with people outside their community, the choice of seed nodes has probably less effect on the level of fairness. In these networks, we call communities a community because they belong to the same group based on some attribute, but they are not necessary a community because they form a cluster within the graph. Recall that the Diversified Homophily BA mechanism stimulates inter-group communities and hence generates these kinds of graphs. We will test and compare the same methods as before on the diversified BA graphs. DQ4FairIM is again trained on 50 test graphs and tested on 10 test graphs. Next to that, we will also run the same experiment on the obesity prevention graphs with the sensitive attribute gender. More or less half of the people in these graphs are men and the other half are women,

where there is no clear cluster of women and men in the network. DQ4FairIM is trained on 20 training graphs and tested on the other 4 graphs. The results for both datasets can be found in Figure 5.7.



(a) dBA200, $k = 7$, $\phi = 0$.

(b) Obesity prevention, attribute gender, $k = 15$. $\phi = 1$.

Figure 5.7: Results of DQ4FairIM and other methods on diversified homophily BA and obesity prevention graphs.

We indeed observe that there is less difference in results between the considered methods, as indicated by the arguing above. For the diversified BA graphs, all results are very similar in terms of total outreach and fairness, although DQ4FairIM with $\phi = 1$ is (very) slightly doing better than other methods in terms of fairness. For the obesity dataset, the results are again very similar, although remarkably CELF is slightly better than the other methods in terms of outreach and fairness. But, for example, DQ4FairIM with $\phi = 0$ and $\phi = 1$ almost obtain the same results which, is very different compared to the homophily BA graphs discussed in the previous sections.

Chapter 6

Conclusions

In the final chapter of this work we will discuss our main contributions, conclusions and answers to the research questions in Section 6.1. In Section 6.2 we discuss some potential practical problems where we think our method could be beneficial. And finally, in Section 6.3 we discuss the future directions of reasearch.

6.1 Main Contributions

The goal of this research was to investigate if reinforcement learning can be used to find a fair solution for the influence maximization problem, while keeping a high influence spread under the independent cascade model. We formulated the problem of influence maximization as a Markov decision process and proposed a novel method, called DQ4FairIM, that incorporates a fairness objective in the reward function, and uses the deep Q-learning structure together with node embeddings proposed by Dai et al. [12] for other graph optimization problems, to train an agent on a pool of similar graphs. The benefit of such a model is that the reinforcement learning agent can be trained on historical graphs and find a solution for a new graph in short time. We examined the training time complexity of DQ4FairIM and performed several experiments to demonstrate the strength and benefits of our method.

As argued in the introduction, fairness can be quite subjective and there is no one-size-fits-all notion of fairness that can be used for any kind of problem. In our work, we decided to focus on fairness in outreach. That is, we consider a solution to be fair if in the total outreach there are no groups (e.g. people sharing the same characteristic for some sensitive attribute) that are disproportionately excluded. In practice, this means that our model is not only maximizing total spread, but its objective is the sum of the total spread and the minimum influence received by any of the groups in the network, as proportional to their population. We showed our results on different graph types. The majority of those were synthetic graphs that have been generated according to the homophily BA-principle and LFR benchmark networks, where there is a clear distinction between a majority group and a minority group within the graph. These network generation mechanisms have been designed to mimic real world networks as much as possible. The results of our model were quite promising. Firstly, DQ4FairIM was indeed able to find a fairer solution in comparison with greedy approaches and our method without a fairness objective, and a similar solution compared with Stoica's parity seeding. Secondly, in terms of total outreach our model performed similar compared with the other

methods. This means that our method was able to find a fairer solution while keeping a high influence spread under the independent cascade model. To be more concrete, this implies that DQ4FairIM does not disproportionately exclude groups while having no loss in influence spread. Moreover, we showed that we can train DQ4FairIM on small graphs and use the trained model to find good solutions for larger graphs, indicating that we can reduce training time for large graphs significantly. More concretely, we are now able to answer the research questions stated at the beginning of our work (Section 1.3).

RQ1: Can we train an RL agent to solve the IM problem?

1) *What are the pros and cons of using RL compared to other algorithms?*
The main benefit of approaching the problem in this way, i.e. by using a RL approach, is that we can learn a policy based on previously seen problem instances and give the solution for a new problem instance without the need of retraining the model. This is particularly useful when graphs from the same application domain or similar types are not totally different from each other and it is too expensive to solve each of them individually. We do not necessarily recommend our approach if there is not enough training data available or if one wants to solve the problem for only one graph, since in that case the costs of training an RL agent might not outweigh the results. Of course, our method could still be used to find a solution, but there could be other suitable methods that are less computationally expensive and perform equally well.

2) *How can we formulate the classical IM problem as an MDP?*
We have formulated the classical influence maximization problem as a Markov decision process. This is not a straightforward thing to do, since IM is in general formulated as an optimization problem. We can summarize our formulation as follows: the time steps are indicated by 1 till k (budget, i.e. size of the seed set), where at every time step the agent adds a node to the seed set. A state is represented by the graph, the communities and the nodes that have been added to the seed set so far. The reward is the marginal gain in influence of adding a node, and the transition-state probabilities follow trivially from the states and actions. By formulating the problem in this way, we were able to apply deep Q-learning with experience replay in order to solve the IM problem.

3) *How scalable is the RL approach?*
We have shown that the training time complexity of DQ4FairIM is given by $O(kE(b\cdot|V|\cdot|\Theta|+m\cdot|E|))$, this means that training our model strongly depends on the sizes of graphs in terms of number of nodes $|V|$ and number of edges $|E|$. For example, training our model on large sized graphs of thousands of nodes could take several days. However, as we have shown, we can train our model on smaller graphs and obtain good solutions with this model for larger graphs as well, making our model very scalable. This also means that pre-trained models can be used to find solutions for new graph instances, which reduces training time significantly. In that case, the model does not have to be retrained from start and can be updated according to the new data. Since the models do not depend on the size of the graphs, we think it might be possible and useful to make pre-trained models widely available so that they can be used as a starting point to solve new problems.

RQ2: How can RL be used to find a fairer solution to the IM Problem?

1) *How do we define fairness in the context of IM? More specifically, are there any other fairness criteria that are suitable for the IM problem besides the community-based fairness notions (Equations 1.1 and 1.2)?*
As mentioned already, we have decided to focus on fairness in outreach where the goal is that no groups are disproportionally excluded. In IM, it is about which nodes finally get activated in the

network, i.e. which nodes receive the influence or information. We were looking for a fair solution in the sense that we do not want certain communities to be excluded from the information. However, the fairness measure of Equation 1.2 did not seem suitable for our approach, since the agent will try to find a solution where all communities are proportionally equally influenced. Instead, we used the maxmin fairness where we maximize the minimum influence received by any of the groups, as proportional to their population.

2) *How can we enhance this fairness notion in the RL framework?*
We enhanced this fairness notion in our method by not only letting the reward depend on total outreach, but also on the maxmin fairness, i.e. minimum influence received by any of the groups in the network. The reward the agent receives is the expected percentage of influenced nodes plus ϕ (the fairness weight) times the maxmin fairness.

3) *What is the loss incurred by enhancing fairness compared to a RL model that is not restricted to fairness? In other words, what is the price of fairness?*
We obtained good results with DQ4FairIM and the price of fairness was zero. With our method, we were able to obtain fairer solutions while not losing any of the total outreach. The RL method without a fairness objective (DQ4FairIM with $\phi = 0$), obtained similar results in terms of outreach but performed significantly worse in terms of fairness. Important to remark here is that under the independent cascade model with a high propagation probability, it is relatively easy to obtain a high influence spread, as discussed in Section 5.3.2.

4) *How does the method compare to other baseline methods?*
In terms of influence spread, our method was comparable to baseline methods such as CELF, Stoica's seeding and RL without a fairness objective. In terms of fairness it is outperforming baseline methods that are 'blind' for fairness and gave slightly better results than Stoica's parity seeding in most cases.

5) *If the algorithm that accounts for fairness gives different solutions, can we explain, for example by evaluating the choices of the agent, why this is the case?*
We found another interesting element about DQ4FairIM. We compared our model that was trained in only maximizing influence spread (DQ4FairIM with $\phi = 0$) with the model that is trained to find a fair solution (DQ4FairIM with $\phi = 1$) in terms of Q-values at the start of seed node selection. Both models were able to identify nodes with many connections and give them a higher Q-value, but the latter was clearly ranking nodes from the minority group higher than similar nodes from the majority group. This indicates that our model is able to identify the minority nodes and give them a higher Q-value in order to find a fairer solution.

6.2 Potential use cases

We have shown and argued that DQ4FairIM outperforms other methods in terms of fairness. As discussed in the introduction, our work focuses on applications of social influence maximization where the aim is to find a fair solution while keeping a high influence spread and RL can potentially be a good approach to tackle the problem. As mentioned, RL is mainly applicable for optimization problems where there is a pool of problem instances that are closely related, and it might be computationally more efficient to train one model that can solve all these problem instances at once. This pre-trained model can be used to find solutions for new graphs in a short amount of time. An appealing domain where RL might be useful in particular, are social networks. In these networks,

new users enter the network, existing users leave the network and users connect with other users frequently. Such a network is very dynamic and changes over time very fast, but probably the core structure or underlying distribution remains the same over time. Instead of solving the IM problem every time over and over again from start, DQ4FairIM will produce a solution for a changed network very fast. Practical IM applications, where fairness in particular is important, and hence DQ4FairIM can be useful, are social intervention campaigns.

One could think of any practical problem where the goal is to spread awareness among people by means of social intervention. An appealing example that we discussed earlier, is the HIV prevention program, where the goal is to spread awareness about HIV among the homeless youth [51]. In practical, there might be several areas where one wants to run this program and our method could be used to train a model that can give a solution for future campaigns as well. Another, more recent example about social intervention in practice involves the Dutch government. They hired so-called influencers to spread awareness about COVID-prevention measures among their (mainly young) followers on social media platforms [13]. DQ4FairIM can address this problem by not only identifying the most influential people in the network, but also making sure that no groups are disproportionately excluded from the information. Especially in these kind of problems, we think that DQ4FairIM can really contribute something, because they serve a common good. In these problems, that have a social character, we think it is especially crucial that certain minority groups are not disproportionately excluded from the information. This is different from classic viral marketing campaigns, where a company may wish to spread the adoption of a new product from some initially selected adopters through the social connections between users. We can imagine that for such a company it does not matter what kind of people it reaches, as long as it reaches a lot of them. However, DQ4FairIM could still be applied to these kinds of problem as well, where the fairness weight ϕ is set to zero.

Another, more specific example of a social intervention campaign wherein our work could be relevant was studied by Oostenbroek et al. [39]. They studied the influence maximization problem applied to a Dutch health promotion program called Jongeren Op Gezond Gewicht (JOGG), translated as Children At Healthy Weight. This program collaborates with municipalities to influence the environment of children aiming at a healthy lifestyle. To achieve behavioural change, JOGG reaches out to organizations in the proximity of children, such as schools and sport clubs, that can participate in the program. There is a limited budget available to hire local JOGG directors who are responsible for introducing the program. The goal is to appoint these directors in such a way that the number of participating organizations is maximized, and consequently the number of children changing their lifestyle is maximized. The directors are the seed of the diffusion process. There certainly is a fairness goal in this setting, since it is undesirable that children from a specific (ethnic) minority are disproportionately excluded from the program. Moreover, RL might be useful since the structure of the social network changes over time because people are joining and leaving, and JOGG wants to make sure that its model works for future campaigns as well. Or, for example, consider that someone wants to run the campaign in different areas or municipalities and the structure of these networks are similar. DQ4FairIM can be used to address this problem: it can be trained to solve the problem for all these different networks at once and used to get a solution for future campaigns.

6.3 Future work

We have presented a novel reinforcement learning framework to find a fair solution to the IM problem, however there are also some limitations to our research. We will discuss these limitations and subsequent directions for further research in this section. Firstly, we evaluated our model on small-sized graphs of 100-500 nodes. This enabled us to run many different experiments and tweak the models where necessary. The majority of these graphs were generated according to the homophily-BA principle that incorporates preferential attachment and homophily of nodes in order to mimic real-world networks as close as possible. However, real-world networks are usually not this small and have many more nodes, like thousands or even millions. Moreover, the distribution of nodes and communities could be different from the graphs we have used. Although we have shown that we can train our model on graphs of 100 nodes and use it for up to 500 nodes, we have not run experiments for graphs of thousands of nodes. We believe that our model will also work for larger sized graphs, but training time will increase significantly. Hence, our first recommendation for future research would be to scale up experiments for large sized (and potentially differently distributed) graphs and explore techniques to make it even more scalable.

Our second recommendation for future research is about exploring and tweaking the technicalities of our algorithm. Interesting to look at, for example, is how different kinds of node embedding methods affect the behaviour of the agent. For now, we have only used the structure2vec node embeddings and did not incorporate any other methods in our approach. Another possible direction is to look at different structures of the Q-network. It now combines the node embeddings with some simple ReLU-layers, but a more complex Q-network might potentially catch more patterns of the problem and consequently give better solutions for more complex graphs. Regarding the method, one could also look at redefining the Markov decision process and see how it affects performance. For example, we now let the agent construct a solution node by node, but it could also select the top k nodes with the highest value in every iteration.

Our third recommendation is to look at the performance of DQ4FairIM for different information propagation models. This work focused on the independent cascade model with a fixed probability only. In practice, it is hard to find a propagation model that mimics reality. Although the independent cascade model is one of the most used in existing literature about IM, it is often argued whether it is a suitable model for practice. We could think of, for example, to use edge-based probabilities that take node characteristics into account or a totally different propagation model like the linear threshold model that we have discussed in Section 2.2.2.

Our final recommendation for future research is to look at how DQ4FairIM can be extended to other graph problems. Note that the Q-network, in principle, can be used for any graph problem. The MDP however, has been specifically designed for IM. We think there is potential to modify the MDP for other graph problems and use DQ4FairIM as inspiration for finding fairer solutions for these problems. In particular, we think that our method could be extended to the Influence blocking maximization problem, which is related to the IM problem. Here the goal is to select seed nodes that block the information spread, instead of selecting nodes that accelerate it.

Bibliography

[1] Khurshed Ali, Chih-Yu Wang, and Yi-Shin Chen. Boosting reinforcement learning in competitive influence maximization with transfer learning. In *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 395–400, 2018. 30

[2] Suman Banerjee, Mamata Jenamani, and Dilip Kumar Pratihar. A survey on influence maximization in a social network. *Knowledge and Information Systems*, 62:3417–3455, 2020. 2

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. 41

[4] Ruben Becker, Gianlorenzo D'Angelo, Sajjad Ghobadi, and Hugo Gilbert. Fairness in influence maximization through randomization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17):14684–14692, May 2021. 1, 5, 43

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference. 4

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 28

[7] Haipeng Chen, Wei Qiu, Han-Ching Ou, Bo An, and Milind Tambe. Contingency-aware influence maximization: A reinforcement learning approach. In *Conference on Uncertainty in Artificial Intelligence*, 2021. 3, 7, 20, 30

[8] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, page 199–208, New York, NY, USA, 2009. Association for Computing Machinery. 2, 30

[9] Suqi Cheng, Huawei Shen, Junming Huang, Guoqing Zhang, and Xueqi Cheng. Staticgreedy: Solving the scalability-accuracy dilemma in influence maximization. CIKM '13, New York, NY, USA, 2013. Association for Computing Machinery. 2

[10] Gennaro Cordasco, Luisa Gargano, Marco Mecchia, Adele Rescigno, and Ugo Vaccaro. *A Fast and Effective Heuristic for Discovering Small Target Sets in Social Networks*, pages 193–208. 01 2015. 3

[11] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data, 2016. 34

[12] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665 https://arxiv.org/abs/1704.01665*, 2017. 34, 35, 55

[13] Agnes de Goede. Kabinet huurt influencers in voor coronacampagne: 'hou je aan de regels'. *RTL Nieuws*. 58

[14] Charles Elkan. The foundations of cost-sensitive learning. *Proceedings of the Seventeenth International Conference on Artificial Intelligence: 4-10 August 2001; Seattle*, 1, 05 2001. 1

[15] Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 329–338, 2019. 1

[16] Justin Fu, Aviral Kumar, Matthew Soh, and Sergey Levine. Diagnosing bottlenecks in deep q-learning algorithms. In *International Conference on Machine Learning*, pages 2021–2030. PMLR, 2019. 26

[17] Pratik Gajane, Akrati Saxena, Maryam Tavakol, George Fletcher, and Mykola Pechenizkiy. Survey on fair reinforcement learning: Theory and practice. *arXiv preprint arXiv:2205.10032*, 2022. 7

[18] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016. 33

[19] Jon Henley. Dutch government faces collapse over child benefits scandal. *The Guardian*. 1

[20] Bas Hofstra, Rense Corten, Frank van Tubergen, and Nicole B. Ellison. Sources of segregation in social networks: A novel approach using facebook. *American Sociological Review*, 82(3):625–656, 2017. 4

[21] Mike Isaac. Meta agrees to alter ad technology in settlement with u.s. *The New York Times*. 7

[22] Tommi Jaakkola, Michael Jordan, and Satinder Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 11 1994. 65, 66

[23] Harshavardhan Kamarthi, Priyesh Vijayan, Bryan Wilder, Balaraman Ravindran, and Milind Tambe. Influence maximization in unknown social networks: Learning policies for effective graph sampling. AAMAS '20, page 575–583, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems. 3, 20, 30, 35

[24] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. Considerations on fairness-aware data mining. In *2012 IEEE 12th International Conference on Data Mining Workshops*, pages 378–385, 2012. 1

[25] Fariba Karimi, Mathieu Génois, Claudia Wagner, Philipp Singer, and Markus Strohmaier. Homophily influences ranking of minorities in social networks. *Scientific Reports*, 8, 07 2018. 41

[26] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, page 137–146, New York, NY, USA, 2003. Association for Computing Machinery. 2, 4

[27] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008. 43

[28] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. KDD '07, page 420–429, New York, NY, USA, 2007. Association for Computing Machinery. 3, 45

[29] Fei-Fei Li, Justin Johnson, and Serena Yeung. *Lecture 14: Reinforcement Learning*. Stanford University School of Engineering, May 2017. 15

[30] Hui Li, Mengting Xu, Sourav S Bhowmick, Changsheng Sun, Zhongyuan Jiang, and Jiangtao Cui. Disco: Influence maximization meets network embedding and deep learning, 2019. 34, 35

[31] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 30(10):1852–1872, 2018. 1

[32] Linyuan Lü and Tao Zhou. Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150–1170, Mar 2011. 4

[33] Farzan Masrour, Tyler Wilson, Heng Yan, Pang-Ning Tan, and Abdol Esfahanian. Bursting the filter bubble: Fairness-aware network link prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):841–848, Apr. 2020. 4

[34] Nina Mazyavkina, Sergei Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *CoRR*, abs/2003.03600, 2020. 7

[35] Francisco S Melo. Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4, 2001. 19, 65

[36] Vlado Menkovski and Simon Koop. *Deep Learning Lecture Notes*. Eindhoven University of Technology, May 2021. 20

[37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. 25, 26

[38] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. 20, 24, 25

[39] Maurits H. W. Oostenbroek, Marco J. van der Leij, Quinten A. Meertens, Cees G. H. Diks, and Heleen M. Wortelboer. Link-based influence maximization in networks of health promotion professionals. *PLOS ONE*, 16:1–21, 08 2021. 58

[40] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, aug 2014. 33

[41] Akrati Saxena, George Fletcher, and Mykola Pechenizkiy. Hm-eiict: Fairness-aware link prediction in complex networks using community information. *Journal of Combinatorial Optimization*, pages 1–18, 2021. 4

[42] Akrati Saxena, George Fletcher, and Mykola Pechenizkiy. Fairsna: Algorithmic fairness in social network analysis. *arXiv preprint arXiv:2209.01678*, 2022. 4

[43] Akrati Saxena, George Fletcher, and Mykola Pechenizkiy. Nodesim: node similarity based network embedding for diverse link prediction. *EPJ Data Science*, 11(1):24, 2022. 4

[44] Akrati Saxena and Sudarshan Iyengar. Centrality measures in complex networks: A survey. *arXiv preprint arXiv:2011.07190*, 2020. 4

[45] Akrati Saxena, Pratishtha Saxena, and Harita Reddy. Fake news propagation and mitigation techniques: A survey. In *Principles of Social Networking*, pages 355–386. Springer, 2022. 1, 2

[46] Paulo Shakarian, Abhinav Bhatnagar, Ashkan Aleali, Elham Shaabani, and Ruocheng Guo. *Diffusion in Social Networks*. 06 2015. 10, 12

[47] Paulo Shakarian, Abhinav Bhatnagar, Ashkan Aleali, Elham Shaabani, and Ruocheng Guo. The independent cascade and linear threshold models. In *Diffusion in Social Networks*, pages 35–48. Springer, 2015. 9, 10

[48] Ana-Andreea Stoica and Augustin Chaintreau. Fairness in social influence maximization. *WWW '19: Companion Proceedings of The 2019 World Wide Web Conference*, pages 569–574, 05 2019. 1, 2, 4, 5

[49] Ana-Andreea Stoica, Jessy Xinyi Han, and Augustin Chaintreau. Seeding network influence in biased networks and the benefits of diversity. In *Proceedings of The Web Conference 2020*, pages 2089–2098, 2020. 2, 5, 44, 45

[50] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 15

[51] Alan Tsang, Bryan Wilder, Eric Rice, Milind Tambe, and Yair Zick. Group-fairness in influence maximization. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5997–6005. International Joint Conferences on Artificial Intelligence Organization, 7 2019. 2, 5, 6, 33, 43, 58

[52] Sotiris Tsioutsiouliklis, Evaggelia Pitoura, Panayiotis Tsaparas, Ilias Kleftakis, and Nikos Mamoulis. Fairness-aware pagerank. In *Proceedings of the Web Conference 2021*, WWW '21, page 3815–3826, New York, NY, USA, 2021. Association for Computing Machinery. 4

[53] Chao Wang, Yiming Liu, Xiaofeng Gao, and Guihai Chen. A reinforcement learning model for influence maximization in social networks. In *International Conference on Database Systems for Advanced Applications*, pages 701–709. Springer, 2021. 3

[54] Xindi Wang, Onur Varol, and Tina Eliassi-Rad. Information access equality on network generative models. *ArXiv*, abs/2107.02263, 2021. 41, 42, 43

[55] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989. 19

[56] Bryan Wilder, Nicole Immorlica, Eric Rice, and Milind Tambe. Maximizing influence in an unknown social network. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. 3, 43

[57] H.-H. Wu and S. Küçükyavuz. A two-stage stochastic programming approach for influence maximization in social networks. *Computational Optimization and Applications*, 69(3):563–595, 2018. 3

[58] Weijian Zheng, Dali Wang, and Fengguang Song. Opengraphgym: A parallel reinforcement learning framework for graph optimization problems. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João

Teixeira, editors, *Computational Science – ICCS 2020*, pages 439–452, Cham, 2020. Springer International Publishing. 7

Appendix A

Convergence of Q-values

Melo [35] proved Theorem 3.1 in the following way. He first writes the optimal Q-function as a fixed point of a contraction operator $\mathbf{B}$ for a generic function $q : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ as:

$$(\mathbf{B}q)(s,a) = \sum_{s' \in \mathcal{S}} \mathbb{P}_a(s,s')[r(s,a,s') + \gamma \max_{a' \in \mathcal{A}} q(s',a')]. \tag{A.1}$$

Lemma A.1. *This operator is a contraction in the sup-norm, i.e.,*

$$||\boldsymbol{B}q_1 - \boldsymbol{B}q_2||_\infty \leq \gamma ||q_1 - q_2||_\infty. \tag{A.2}$$

Proof.

$$\begin{aligned}
||\mathbf{B}q_1 - \mathbf{B}q_2|| &= \max_{s,a} \left| \sum_{s' \in \mathcal{S}} \mathbb{P}_a s, s' [r(s,a,s') + \gamma \max_{a' \in \mathcal{A}} q_1(s,a') - r(s,a,s') + \gamma q_2(s,a')] \right| \\
&= \max_{s,a} \gamma \left| \sum_{s' \in \mathcal{S}} \mathbb{P}_a(s,s') [\max_{a' \in \mathcal{A}} q_1(s,a') - \max_{a' \in \mathcal{A}} q_2(s,a')] \right| \\
&\leq \max_{s,a} \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s,s') \left| \max_{a' \in \mathcal{A}} q_1(s,a') - \max_{a' \in \mathcal{A}} q_2(s,a') \right| \\
&\leq \max_{s,a} \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_a(s,s') \max_{z,a'} |q_1(z,a') - q_2(z,a')| \\
&= \max_{s,a} \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_a(s,s') ||q_1 - q_2||_\infty \\
&= \gamma ||q_1 - q_2||_\infty.
\end{aligned}$$

He also introduces a theorem, that was proven by Jaakola et al. [22].

Theorem A.2 (Convergence of random iterative process.). *The random iterative process* $\{\Delta_t\}$ *defined as:*

$$\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \alpha_t(x) F_t(x)$$

converges to zero with probability 1, under the following assumptions:

1. *The state space is finite;*
2. $0 \leq \alpha_t \leq 1, \sum_t \alpha_t(x) = \infty$ *and* $\sum_t \alpha_t^2(x) < \infty$*;*
3. $||\mathbb{E}[F_t(x)|\mathcal{F}_t]||_W \leq \gamma||\Delta_t||_W$*, with* $\gamma < 1$*;*
4. $\boldsymbol{var}[F_t(x)|\mathcal{F}_t] \leq C(1 + ||\Delta_t||_W^2)$*, for* $C > 0$*.*

Where $\mathcal{F} = \{\Delta_t, \Delta_{t-1}, \dots, F_{t-1}, \dots, \alpha_{t-1}, \dots, \beta_{t-1}, \dots\}$ *stands for the past at step t. The notation* $\|\cdot\|_W$ *refers to some weighted maximum norm.*

Proof. See Jaakola et al. [22].

Now, we can proof Theorem 3.1 as follows. The update-rule is given by:

$$Q_{t+1}(s_t, a_t) \longleftarrow (1 - \alpha)Q_t(s_t, a_t) + \alpha\left(r_t + \gamma \cdot \max_{a_t \in A} Q(s_{t+1}, a_t)\right).$$

If we substract $Q^*(s_t, a_t)$ from both sides and let $\Delta_t(s, a) = Q_t(s, a) - Q_t^*(s, a)$, we get:

$$\Delta_t(s_t, a_t) = (1 - \alpha)\Delta_t(s_t, a_t) + \alpha[r_t + \gamma \max_{a' \in \mathcal{A}} Q_t(s_{t+1}, a') - Q^*(s_t, a_t)].$$

We can write:

$$F_t(s, a) = r(s, a, X(s, a)) + \gamma \max_{a' \in \mathcal{A}} Q_t(X(s, a), b) - Q^*(s, a), \tag{A.3}$$

where $X(s, a)$ is a random sample obtained from the Markov chain $(\mathcal{S}, \mathbb{P}_a)$, we have

$$\begin{aligned}\mathbb{E}[F_t(s, a)|\mathcal{F}_t] &= \sum_{s' \in \mathcal{S}} \mathbb{P}_a(s, s')[r(s, a, s') + \gamma_{a' \in \mathcal{A}} Q_t(s', a') - Q^*(s, a)] \\ &= (\mathbf{B}Q_t)(s, a) - Q^*(s, a).\end{aligned}$$

Note that $Q^* = \mathbf{B}Q^*$, so we get:

$$\mathbb{E}[F_t(s, a)|\mathcal{F}_t] = (\mathbf{B}Q_t)(s, a) - (\mathbf{B}Q^*)(s, a)$$

From Equation A.2 it directly follows that:

$$||\mathbb{E}[F_t(s, a)|\mathcal{F}_t]||_\infty \leq \gamma||Q_t - Q^*||_\infty = \gamma||\Delta_t||_\infty.$$

Lastly, we have

$$\begin{aligned}\mathbf{var}[F_t(x)|\mathcal{F}_t] &= \mathbb{E}\left[\left(r(s, a, X(s, a)) + \gamma \max_{a' \in \mathcal{A}} Q_t(X(s, a), a') - Q^*(s, a) - (\mathbf{B}Q_t)(s, a) + Q^*(s, a)\right)^2\right] \\ &= \mathbb{E}\left[\left(r(s, a, X(s, a)) + \gamma \max_{a' \in \mathcal{A}} Q_t(X(s, a), a') - (\mathbf{B}Q_t)(s, a)\right)^2\right] \\ &= \mathbf{var}\left[r(s, a, X(s, a)) + \gamma \max_{a' \in \mathcal{A}} Q_t(X(s, a), a')|\mathcal{F}_t\right]\end{aligned}$$

which, due to the fact that the reward r is bounded, clearly satisfies

$$\mathbf{var}[F_t(x)|\mathcal{F}_t] \leq C(1 + ||\Delta_t||_W^2) \tag{A.4}$$

for some constant C. Then, by Theorem A.2, Δ_t converges to 0 with probability 1, i.e., Q_t converges to the optimal Q^* with probability 1.

Appendix B

Detailed results

This appendix contains the detailed results for the experiments in Chapter 5. In Chapter 5, we average over a number of test graphs, here we present the results for each graph individually in tabular form.

I	CELF		Parity		$\phi = 0$		$\phi = 0.25$		$\phi = 0.5$		$\phi = 0.75$		$\phi = 1$	
0	0.26	0.25	0.26	0.22	0.27	0.24	0.26	0.25	0.26	0.21	0.24	0.24	0.24	0.21
1	0.25	0.15	0.23	0.19	0.25	0.14	0.22	0.21	0.25	0.24	0.24	0.22	0.26	0.22
2	0.26	0.18	0.26	0.25	0.26	0.15	0.25	0.20	0.25	0.25	0.26	0.25	0.27	0.25
3	0.23	0.17	0.23	0.22	0.23	0.17	0.23	0.19	0.24	0.20	0.23	0.22	0.24	0.22
4	0.25	0.24	0.24	0.22	0.26	0.26	0.26	0.25	0.25	0.23	0.25	0.24	0.24	0.23
5	0.25	0.21	0.24	0.22	0.25	0.25	0.25	0.23	0.25	0.21	0.23	0.21	0.25	0.25
6	0.25	0.24	0.24	0.23	0.24	0.23	0.25	0.24	0.24	0.21	0.24	0.21	0.23	0.21
7	0.26	0.22	0.24	0.21	0.25	0.15	0.25	0.21	0.25	0.15	0.25	0.25	0.25	0.24
8	0.25	0.24	0.25	0.24	0.25	0.20	0.24	0.19	0.25	0.16	0.25	0.24	0.26	0.25
9	0.25	0.20	0.23	0.21	0.25	0.25	0.25	0.22	0.24	0.19	0.24	0.22	0.25	0.25

Table B.1: Detailed results in tabular form for BA100 of Figure 5.2. For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.18	0.16	0.18	0.14	0.18	0.18	0.19	0.16	0.18	0.18
1	0.20	0.16	0.19	0.14	0.19	0.18	0.19	0.13	0.19	0.18
2	0.19	0.14	0.19	0.14	0.19	0.18	0.19	0.16	0.20	0.19
3	0.19	0.13	0.19	0.14	0.18	0.18	0.18	0.16	0.19	0.19
4	0.19	0.14	0.19	0.11	0.20	0.16	0.19	0.14	0.19	0.15
5	0.20	0.17	0.19	0.14	0.19	0.19	0.19	0.17	0.19	0.19
6	0.19	0.16	0.18	0.12	0.18	0.17	0.19	0.17	0.19	0.15
7	0.19	0.17	0.18	0.16	0.18	0.18	0.19	0.17	0.19	0.18
8	0.19	0.14	0.19	0.14	0.19	0.19	0.20	0.15	0.19	0.17
9	0.19	0.15	0.18	0.12	0.18	0.16	0.19	0.15	0.19	0.18

Table B.2: Detailed results in tabular form for BA200 of Figure 5.3 (a). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.18	0.11	0.17	0.11	0.18	0.14	0.17	0.11	0.17	0.17
1	0.19	0.15	0.18	0.16	0.18	0.18	0.18	0.13	0.19	0.18
2	0.18	0.13	0.17	0.11	0.17	0.16	0.18	0.11	0.17	0.17
3	0.18	0.15	0.17	0.14	0.17	0.17	0.16	0.14	0.18	0.18
4	0.18	0.12	0.18	0.14	0.18	0.17	0.17	0.12	0.19	0.18
5	0.19	0.15	0.18	0.14	0.19	0.17	0.18	0.15	0.18	0.17
6	0.19	0.13	0.19	0.13	0.18	0.15	0.17	0.12	0.18	0.18
7	0.18	0.16	0.18	0.14	0.18	0.15	0.16	0.11	0.18	0.17
8	0.19	0.14	0.19	0.14	0.19	0.15	0.19	0.13	0.19	0.18
9	0.19	0.16	0.19	0.16	0.18	0.18	0.18	0.12	0.17	0.17

Table B.3: Detailed results in tabular form for BA300 of Figure 5.3 (b). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.19	0.15	0.18	0.13	0.19	0.19	0.19	0.19	0.19	0.17
1	0.19	0.17	0.18	0.17	0.18	0.18	0.18	0.18	0.18	0.16
2	0.18	0.16	0.18	0.16	0.18	0.18	0.19	0.16	0.18	0.17
3	0.19	0.16	0.19	0.16	0.19	0.18	0.20	0.16	0.19	0.19
4	0.18	0.13	0.18	0.13	0.18	0.17	0.18	0.14	0.19	0.16
5	0.18	0.17	0.18	0.18	0.18	0.17	0.18	0.15	0.18	0.17
6	0.18	0.15	0.19	0.14	0.19	0.17	0.20	0.16	0.19	0.19
7	0.19	0.18	0.18	0.16	0.18	0.18	0.19	0.18	0.19	0.19
8	0.18	0.15	0.18	0.14	0.17	0.17	0.18	0.15	0.19	0.18
9	0.19	0.18	0.18	0.16	0.18	0.18	0.18	0.17	0.17	0.15

Table B.4: Detailed results in tabular form for BA400 of Figure 5.3 (c). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.19	0.16	0.18	0.14	0.18	0.17	0.19	0.16	0.20	0.17
1	0.19	0.17	0.19	0.16	0.19	0.19	0.19	0.16	0.20	0.20
2	0.19	0.16	0.18	0.15	0.19	0.17	0.18	0.17	0.19	0.19
3	0.19	0.19	0.18	0.16	0.19	0.19	0.18	0.18	0.19	0.18
4	0.19	0.15	0.19	0.15	0.18	0.17	0.19	0.16	0.19	0.18
5	0.19	0.13	0.19	0.13	0.19	0.17	0.19	0.17	0.20	0.18
6	0.20	0.16	0.20	0.16	0.19	0.17	0.19	0.16	0.20	0.20
7	0.19	0.16	0.19	0.14	0.19	0.18	0.19	0.16	0.19	0.19
8	0.19	0.17	0.18	0.15	0.18	0.18	0.18	0.16	0.20	0.19
9	0.19	0.16	0.19	0.15	0.18	0.18	0.19	0.15	0.19	0.17

Table B.5: Detailed results in tabular form for BA500 of Figure 5.3 (d). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.21	0.19	0.21	0.17	0.21	0.18	0.21	0.19	0.19	0.18
1	0.22	0.21	0.21	0.20	0.22	0.21	0.21	0.20	0.22	0.20
2	0.21	0.18	0.21	0.18	0.20	0.19	0.21	0.19	0.22	0.21
3	0.21	0.20	0.20	0.20	0.20	0.20	0.20	0.19	0.19	0.18
4	0.20	0.19	0.20	0.18	0.19	0.19	0.21	0.21	0.20	0.20
5	0.21	0.19	0.19	0.19	0.19	0.19	0.20	0.19	0.20	0.20
6	0.21	0.19	0.21	0.19	0.21	0.19	0.21	0.18	0.20	0.20
7	0.21	0.20	0.21	0.19	0.20	0.19	0.20	0.15	0.19	0.19
8	0.21	0.21	0.20	0.20	0.20	0.19	0.22	0.22	0.22	0.20
9	0.21	0.16	0.20	0.16	0.20	0.18	0.21	0.17	0.20	0.18

Table B.6: Detailed results in tabular form for dBA200 of Figure 5.7 (a). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.07	0.06	0.07	0.05	0.07	0.06	0.07	0.06	0.07	0.06
1	0.07	0.07	0.06	0.05	0.06	0.06	0.07	0.06	0.07	0.07
2	0.08	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.06
3	0.07	0.06	0.07	0.05	0.07	0.06	0.07	0.06	0.07	0.06

Table B.7: Detailed results in tabular form for obesity of Figure 5.7 (b). For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.

I	CELF		Agnostic		Parity		$\phi = 0$		$\phi = 1$	
0	0.06	0.05	0.05	0.03	0.06	0.06	0.06	0.04	0.06	0.06
1	0.06	0.04	0.05	0.05	0.05	0.05	0.05	0.02	0.05	0.05
2	0.06	0.04	0.06	0.04	0.06	0.05	0.06	0.04	0.06	0.06
3	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
4	0.05	0.03	0.05	0.04	0.05	0.05	0.05	0.05	0.05	0.04
5	0.06	0.03	0.06	0.04	0.06	0.05	0.06	0.05	0.06	0.06
6	0.06	0.05	0.05	0.03	0.05	0.05	0.06	0.03	0.06	0.05
7	0.05	0.05	0.05	0.04	0.05	0.05	0.05	0.05	0.05	0.05
8	0.06	0.04	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
9	0.06	0.05	0.06	0.04	0.06	0.05	0.06	0.05	0.06	0.05

Table B.8: Detailed results in tabular form for LFR250 of Figure 5.6. For every model/algorithm, the first column contains the total outreach and the second column the maxmin fairness.