POLITECNICO
MILANO 1863

Election Manipulation with uncertainty on users' preferences

Advanced Programming for Scientific Programming - 8 cfu

Carlo Augusto Vitellio - 920561

Contents

1 Introduction

An election is a group decision making process by which a population chooses an individual or a group of them to hold public office. Nowadays it seems possible to interfere with the results of an election via social media [AG17].

In the literature the problem has been discussed in various facets [CCG19], [Cas+20], [WV17] and [Wu+19]. None of the aforementioned works analyse the problem with unknown voters' voting preferences.

The purpose of this project is to give a plausible mathematical framework around the latter problem and try to provide some tool to analyse the actual possibilities available in this framework.

1.1 Model

To model an election we imagine to have a proper knowledge of the population called to cast a ballot and we hypothesize it can be represented with a unique undirected graph. Each node of the graph represents a voter. Each voter has a vote preference profile we represent as probabilities she votes for each candidate. Indeed, we consider $\boldsymbol{\pi_j^*} = (\pi_j^*(c_1), ..., \pi_j^*(c_K))$ with $\pi_j^*(c_k)$ representing the probability user j votes for candidate c_k ($\pi_j^*(c_k) \in [0,1]$ and $\sum_k \pi_j^*(c_k) = 1$).

An external agent (called *manipulator*) tries to manipulate the election. In order to do that, she can buy a user (called from now on *seed* s) with a message in favour of a candidate c_k. This message affects all users directly connected to the seed (*i.e.* all users $j \in \delta^+(s)$). These may adopt the message and activate themselves with a certain probability $p_j(c_k)$ (*i.e.* probability node j accepts a message pro candidate c_k). Once a user adopts a message, she becomes a seed herself and affects her neighborhood. This scheme goes under the name of *independent cascade model* [DGW09]. We assume an adopted message causes a change in the probability $\pi_j^*(c_k)$, but a refused message doesn't have any implication.

We assume the probability $p_j(c_k)$ of user j adopting the message pro candidate c_k is equal to the probability she votes for her (*i.e.* $p_j(c_k) = \pi_j^*(c_k)$). This implies the probability of adopting a message doesn't depend on who the sender of the message is, but depends only on the receiver.

We reduce to a two candidates scenario, motivated by two main reasons: first, many elections have two main parties running for the win; secondly, we recall that heuristics in an influence

maximization problem usually try to influence only in favour of the same thing/person. Indeed, the latter interpretation may be interpreted as the candidate most preferred by the *manipulator* (c_0) and the one encapsulating any other candidate, independently on who she is. Then, we hypothesize the *manipulator* sends messages only in favour of her preferred candidate c_0 and that it is not possible to send messages against other candidates.
In this scenario we define:

- $\boldsymbol{\pi_j^*(t)} = (\pi_j^*(t), 1 - \pi_j^*(t))$ (with π_j^* probability of voting c_0) preference profile of user j at time t
- X_j^* Bernoulli variable that is one if voter j casts a ballot for candidate c_0

$$X_j^* \sim Bernoulli(\pi_j^*) \tag{1.1}$$

We introduce another hypothesis on the characteristics of each person in the network: the *resistance* (indicated in the following with w) to change her probability of voting. In fact, we are motivated by the fact that each voter may not react at the same way accepting a new message [AW12]. We assume the *resistance* is known by the *manipulator*.
So, we assume the adoption of a message affects the user j as reported below

$$\pi_j^*(t+1) = \frac{\pi_j^*(t) * w + 1}{w + 1} \tag{1.2}$$

It means that the greater the *resistance* w, the less will be the change in probability of voting the selected candidate. In other words, the user results difficult to be influenced.

For what concerns the *manipulator*, she has both to learn and to elicit users' preferences. The only tool she has is to pick a seed and to learn on the basis of the cascade that takes place. In order to that, she assumes a uniform prior knowledge on the probability of voting of each voter ($\hat{\pi}_j(t)$) and tries to improve its estimate. Her estimates change when a user is solicited by a message following

$$\hat{\pi}_j(t+1) = \begin{cases} \min\left(1, \dfrac{\hat{\pi}_j(t) * w + 2}{w+1}\right) & \text{if user } j \text{ adopts the message} \\ \max\left(0, \dfrac{\hat{\pi}_j(t) * w - 1}{w+1}\right) & \text{if user } j \text{ refuses the message} \end{cases} \tag{1.3}$$

For what concerns the choice of the seed, the *manipulator* needs a metric for evaluating the utility of each voter. A reasonable choice is to consider to possible change in probability caused by the acceptance of the message. It means that the marginal utility u_j of the user j is

$$u_j = \pi_j^*(t+1) - \pi_j^*(t) = \frac{1 - \pi_j^*(t)}{w + 1} \tag{1.4}$$

Proceeding further, the actual utility u_j^* may be more complex that the marginal one. In fact, it may take in consideration also the voter's neighbourhood. In particular, including also the users directly connected with j and their expected utility

$$u_j^* = u_j + \sum_{i \in \delta^+(j)} \hat{\pi}_i(t) * u_i \tag{1.5}$$

The same argument can be repeated with users at distance maximum equal to a parameter (*steps*). To do that is important to add a reasoning about the possible paths bringing to the next nodes. To tackle this problem we propose the following algorithm 1.

Algorithm 1 Computing each voter's utility

```
Require: $seed$
  $u^*_{seed} \leftarrow 0$
  add $seed$ to $queue$
  while $queue$ not empty do
      take first $node$ in $queue$
      if $node$ has already been visited or $d(seed, node) \geq steps$ then
          Do nothing
      else
          $q \leftarrow$ prob. activation w.r.t. the path from $seed$        ▷ if $node = seed$, $q = 1$
          $u^*_{seed} \leftarrow u^*_{seed} + u_{node} * q$
          add $neighbours$ of $node$ to $queue$
      end if
  end while
```

Here, the path from *seed* is considered as the first lexicographic path analyzed during the process.

Once determined the best node, it is picked as seed and an information cascade is observed.

Eventually, the *manipulator* has to evaluate the manipulation process according to some metric and this has to be analysed w.r.t. the graph properties of the social network.

2 Instructions

2.1 External tools used

They have been used some external tools and they are listed below

- Boost Graph Library (`https://www.boost.org/`)
  - Mainly, it was used the template only part of the library. This means that no compilation was needed for most of the code.
  - For implementing the *GraphCreatorInputFile* class two other libraries were also used. In particular, "boost_graph" and "boost_regex" were used.

  For installing the Boost library it suggested to follow the documentation.

- GetPot `http://getpot.sourceforge.net/`
  Easily downloadable from the link and there is no need for compilation.

- MuParser `https://beltoforion.de/en/muparser/`
  For installing it, follow the build instructions.

- Utilities implemented by Prof. Luca Formaggia
  Those utilities have been proposed as examples in his APSC classes at Politecnico di Milano and they are available at `https://github.com/pacs-course/pacs-examples`. In the repository of the project they are included in the include/ExternalUtilities folder.

2.2 Make instructions

Please find the repository on GitHub and download it. Before being able to build everything, the above listed have to be already available on your machine.
Then, the user should modify some variable in the *Makefile.inc* file. In particular, the variables *ROOT*, *mkBoostInc*, *MUPARSER_INC_DIR* and *MUPARSER_LIB_DIR* have to be adapted with the directories where the libraries reside in your machine.

Once correctly configured *Makefile.inc* and moved to the directory of the project, it is possible to write *make all* in the terminal to build both the dynamic libraries and the executable.

3 Implementation analysis

3.1 Graph Creators

In order to being able to analyse various graph structures, we conceive a plugin architecture that enables to make at disposal of the code new creator methods just loading a shared library dynamically. In fact, we expect that different graph structures lead different results in the manipulation process and we don't want the user to recompile the whole code every time.
This section of the project is freely inspired by an example proposed by Prof. Luca Formaggia in his classes and his implementation of a generic factory is used. A factory is a collection of objects that normally belongs to the same polymorphic family. The basic idea is to map an identifier to a object in such a way the object can be obtained from the factory by indicating its identifier. In particular, we use a *std::string* as identifier enabling to read it from an input text file.
In order to correctly load the Creators to the factory, an automatic registration is made as soon as the corresponding dynamic library is loaded. This is achieved using a function with the *constructor* attribute. It may be found in *src/GraphCreators/BoostCreators.cpp* and *src/-GraphCreators/GraphCreatorInputFile.cpp*.

The only *crucial* decision that has to be made at compile-time is the actual implementation of the Graph. In fact, this decision is taken in the *include/EMTraits.hpp* file defining an alias *Graph* for a BGL adjacency_list. The choice was guided by the kind of algorithms needed in the Election Manipulation process and led to the implementation of both vertices and edges with a *boost::vecS* which means that they are stored in *std::vector*. If new algorithms are introduced the choice of the Graph implementation may be re-evaluated.

3.1.1 GraphCreatorBase class

This class represents the blueprint for all Graph Creators.
The class is clonable in order to write copy constructors and assignment operators for classes that aggregate object of the GraphCreator hierarchy by composition. The main method is *create()* that generate the Graph with the corresponding attributes.

```
//! The basis class for all graph creators
class GraphCreatorBase
{
```

```
public:

  //! The class is clonable
  virtual std::unique_ptr<GraphCreatorBase> clone() const=0;

  virtual ~GraphCreatorBase()=default;
  //! String that identifies the type of Graph Creator
  virtual std::string name() const=0;
  //! Method to set the Random Generator (source of randomness for creating
    the graph)
  virtual void set_gen(const RandomGenerator&) {}
  //! Method to read the parameters needed by each GraphCreator. It needs a
    GetPot parameter where every information may be found
  virtual void read_params(GetPot)=0;
  //! Method that returns a boolean indicating whether the attributes of
    each vertex should be read from a file or whether they should be randomly
     generated
  virtual bool get_read_attributes() const=0;
  //! The main method used to actually create the Graph
  virtual Graph create()=0;
};
```

All Graph Creators implemented inherits from this base class, as it is clear in figure 3.1.

3.1.2 Boost Creators

Most of the Graph Creators take advantages of the generators provided by the BGL (Boost Graph Library). For this reason, they have been grouped in a unique shared library called libBoostCreators.so. The library contains the following creators:

- *GraphCreatorErdosRenyi*: it exploits the erdos_renyi_generator implemented in the BGL. A Erdös-Rényi graph is a random graph chosen uniformly at random from the collection of all graphs with N vertices and E edges. Erdös-Renyi graphs typically exhibit very little structure. No self-loops is allowed within this implementation.

- *GraphCreatorSmallWorld*: it exploits the small_world_generator implemented in the BGL. A small-world graph consists of a ring graph (i.e. each vertex is connected to its k nearest neighbours). Edges in the graph are randomly rewired to different vertices with a probability p.

- *GraphCreatorRMAT*: it exploits the rmat_graph_generator implemented in the BGL. An R-MAT graph has a scale-free distribution w.r.t. vertex degree and is Implemented using Recursive-MATrix partitioning [CZF04]. A R-MAT graph has N vertices and E edges. The parameters a, b, c, and d represent the probability that a generated edge is placed of each of the 4 quadrants of the partitioned adjacency matrix.

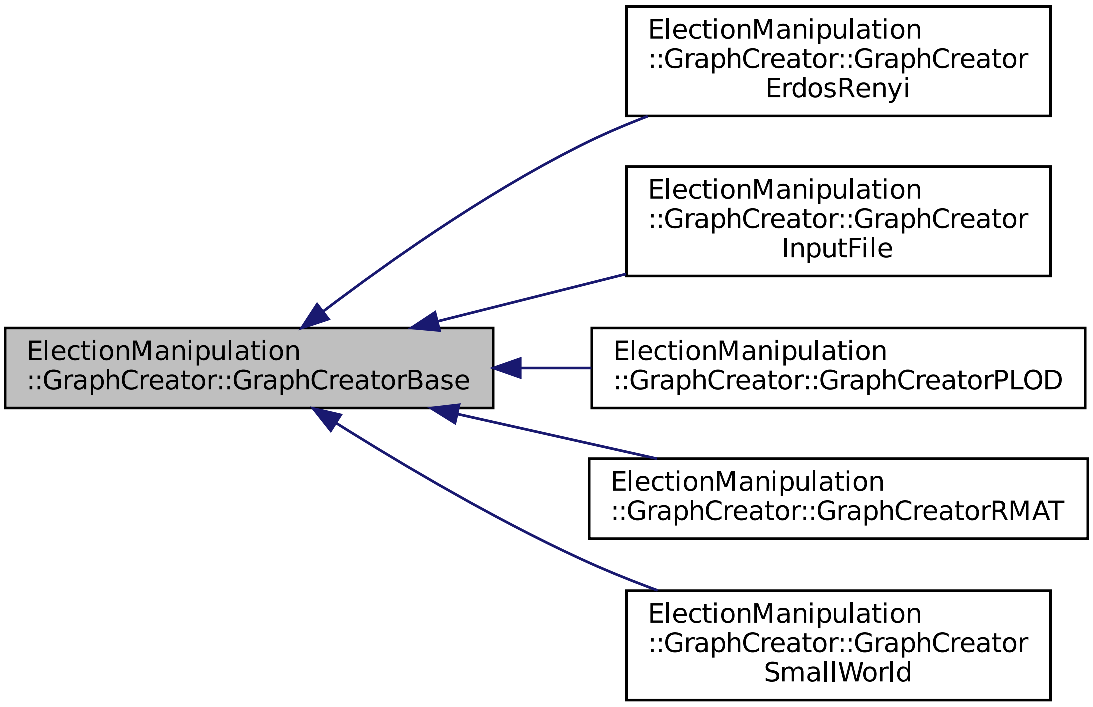
ElectionManipulation ::GraphCreator::GraphCreatorBase
ElectionManipulation ::GraphCreator::GraphCreator ErdosRenyi
ElectionManipulation ::GraphCreator::GraphCreator InputFile
ElectionManipulation ::GraphCreator::GraphCreatorPLOD
ElectionManipulation ::GraphCreator::GraphCreatorRMAT
ElectionManipulation ::GraphCreator::GraphCreator SmallWorld

Figure 3.1: Graph Creators family

- *GraphCreatorPLOD*: it exploits the plod_generator implemented in the BGL. The algorithm used to generate the Graph is thee Power Law Out Degree (PLOD) algorithm [PS00]. A scale-free graph typically has a very skewed degree distribution, where few vertices have a very high degree and a large number of vertices have a very small degree. Self-loops are not allowed within this implementation.

All these classes use the *read_params(GetPot)* method to read from a GetPot class the parameters needed for building the corresponding graph. In this way almost any decision may be made at run-time.

3.1.3 GraphCreatorInputFile class

The *GraphCreatorInputFile* class enables to read a graph from an input file. It is possible to read both graphviz and graphml formats, thanks to the methods available in BGL. In contrast with other GraphCreators, this class doesn't need a generator since nothing is randomly generated, but to use this class, you will need to build and link against the "boost_graph" and "boost_regex" libraries. For more information, please refer to Boost documentation.

This class is the core of the shared library libInputCreator.so.

Interfacing with a graphml format it is possible to read the structure of the graph. Instead, reading a graphviz format it is also possible to read the attributes of each node if it is written with the same attributes needed to construct a *Person* object (*i.e.* the attributes related to the voter's probability of voting c_0 and her resistance).

3.2 Person

3.2.1 Person class

This class represents the properties attached to each vertex in the Social Network. In fact, it is used as a Bundled Property attached to the Boost Graph as it can be seen in the declaration of the alias *Graph* in *include/EMTraits.hpp*. Being the vertex property, it should be default constructable.
It is characterized by a private and a public interface. In particular, private data members are intended to be modified only in the construction of the Social Network or, in case of acceptance of a message, by the voter herself. Instead, the public is intended to be accessible also to the *manipulator*.

```
//! Class that identifies a person in the network
class Person{
public:
  using DefRandEngine=std::default_random_engine;

private:
  std::string name;            //!< The name of the person
  double prob_voting_c0{0.};   //!< Probability of voting for the candidate analyzed
  mutable DefRandEngine engine;        //!< Source of randomness during casting a ballot

public:
  // Accessible also to the manipulator
  std::size_t resistance{10};              //!< The resistance in changing the probability of voting
  double manipulator_estim_prob{0.5};      //!< The manipulator's estimate of prob_voting_c0
  double manipulator_prob_activ{0.};       //!< The manipulator's belief in activation during an information cascade
  double manipulator_utility{0.};          //!< The manipulator's gain in influencing the node
  double manipulator_marginal_utility{0.};//!< The manipulator's expected utility in conditioning other vertices

  //! The Person casts a ballot according to her probability of voting
  bool cast_a_ballot() const;

  //! The Person receives a message and decides whether accepting it or not
  bool receive_message();

  //! The Person has been influenced and changes her probability of voting
   accordingly
  void update_prob();

  //! Function used by the manipulator to update her marginal utility in
   conditioning the current node
  void update_marginal_utility();

};
```

3.2.2 PersonCreator class

Each *Person* object may be read from an input file or created thanks to the *PersonCreator* class. This latter class is used to characterise a Person in the Graph. In particular, the Person is populated w.r.t. her probability of voting the candidate in consideration, her resistance to change her belief and her name.
Her resistance attribute is drawn from the ResistanceDistribution passed via the constructor.
Her probability of voting is drawn from the VotingDistribution passed via the constructor.
Two constructors are provided:

- the first one takes the distributions as objects and exploits the fact that the base class ProbabilityDistribution is clonable;
- the second one takes the distributions as unique pointers storing them.

Both constructors take also the Generator passed to the Person (used as source of randomness in her lifetime) and, possibly, the Person's name.
The class stores both the ResistanceDistribution and the VotingDistribution in a wrapper around a unique pointer. The implementation of this wrapper is due to Prof. Luca Formaggia.

3.2.3 Distributions

The aforementioned probability distributions are implemented similarly to the GraphCreators, but this time a simpler factory is used. Indeed, it is used a *std::map* that has to be populated at compile-time.
All Distributions implemented derive from the base class *ProbabilityDistribution*. This pure virtual template class offers a common interface for implementing a Probability Distribution.

The class is clonable for helping with compositions. A new sample can be drawn with the extract() method.

```cpp
//! The basis class for a Probability Distribution
template<class Generator, class ResultType>
class ProbabilityDistribution
{
public:
  //! The class is clonable
  virtual std::unique_ptr<ProbabilityDistribution> clone() const=0;

  virtual ~ProbabilityDistribution()=default;
  //! String that identifies the current Distribution in use
  virtual std::string name() const=0;
  //! Set the random engine used to extract new samples from the
    Distribution
  virtual void set_gen(const Generator&) {}
  //! Method used to read the distribution parameters from GetPot
  virtual void read_params(GetPot)=0;
  //! Method used to extract a new sample from the distribution
  virtual ResultType extract()=0;
};
```

In this project two main type of probability distributions are needed: one on $[0, 1]$ for generating the probability of voting a candidate and another one on $\mathbb{N}$ for generating the resistance attribute. For this reason the Beta and Poisson distributions have been implemented. Those may read their respective parameters via the *read_params(GetPot)* method.

Furthermore, two other classes have been implemented using the MuParser library:

- *ParsedDistribution_0_1*: enables to read from a .txt file a probability density function on [0,1] and to simulate from it. In order to extract a new sample, Rejection sampling is used [CRW04].
- *ParsedDistribution_N*: enables to read from a .txt file a probability mass function on the set of natural numbers and to simulate from it. In order to extract a new sample, Inverse transform sampling is used.

3.3 SocialNetworkCreator class

In order to build a Social Network it must be build both the graph structure and then the latter must be populated.
Building the structure of the graph is delegated to the polymorphic object GraphCreator (derived from GraphCreatorBase). Populating each vertex with a Person is delegated to the PersonCreator member that is simply aggregated as an object. Move semantic is also supported.

```cpp
//! Class for creating a Graph with Persons as vertices

template<class Generator>
class SocialNetworkCreator{
public:
  using GCHandler = GraphCreator::GCHandler;
  //! First constructor taking a GraphCreatorBase object
  template<class PCreator>
  SocialNetworkCreator(const GraphCreatorBase & gc, PCreator&& pc,
                       bool rename_=false):
      graph_creator_ptr{gc.clone()}, person_creator{std::forward<PCreator>(
    pc)},
      rename{rename_} {}
  //! Second constructor taking a GraphCreatorBase unique pointer
  template<class GCH, class PCreator>
  SocialNetworkCreator(GCH && gc, PCreator&& pc, bool rename_=false):
      graph_creator_ptr{std::forward<GCH>(gc)},
      person_creator{std::forward<PCreator>(pc)}, rename{rename_} {}

  //! The actual method used to create the social network
  Graph apply();

private:
  GCHandler graph_creator_ptr;                //!< Unique pointer for
    GraphCreatorBase
  PersonCreator<Generator> person_creator;    //!< PersonCreator object for
    creating a Person
  bool rename;                                //!< true if every Person is
    renamed, false otherwise
};
```

3.4 Manipulation phase

3.4.1 ManipulatorInfluence class

The real manipulation takes place within this class.
The class encapsulates a reference to the social newtork to be manipulated. As described in section 1, the manipulator assigns an utility to each vertex and initiate an information cascade for conditioning the network. The utility represents the expected change in number of votes.
The constructor requests the graph to be manipulated and the number of steps to be used in order to evaluate the utility of a single vertex. Whether or not a user accepts a message, the manipulator improves her estimate on the probability of voting of that node.

```cpp
//! Class to influence the Social Network
class ManipulatorInfluence{
```

```cpp

public:
  ManipulatorInfluence(Graph& my_graph_, std::size_t steps_):
            my_graph{my_graph_}, steps{steps_} {}

  /*!
    This function computes the utility of the seed with respect to the
   expected max_utility
    within its neighours of order lower or equal to the steps member.
    For example, if steps=2 the utility will be computed considering the
   seed's neighbours
    and their adjacent nodes.
    @param seed The node whose utility has to be computed
  */
  void compute_utility(Vertex seed);

  /*!
    This function iterates through all nodes of the graph and call the
    compute_utility method on each of them and pick the one with maximum
   utility
  */
  Vertex max_utility_vertex();

  /*!
    Function to influence the social network
  */
  void influence();

  /*!
    The manipulator updates her estimate on the probability of voting of
    each Person reached in the information cascade.
  */
  void update_estimated_prob(Vertex v, bool accepted);

private:
  Graph& my_graph;   //!< The actual social network to influence
  std::size_t steps; //!< Only vertices at distance lower or equal to this
   parameter will be considered for computing the utility of each vertex
};
```

The method *compute_utility* implements a slightly different algorithm w.r.t. algorithm 1. As a matter of fact, they are equivalent, but the one here implemented considers a node and each edge from it and the edges' target node in order to compute the seed's utility.

3.4.2 PerformanceEvaluator class

This class is used to define and compute metrics for evaluating the influence process. At this moment, two metrics are implemented:

- the MSE of the estimated probabilities
- the number of expected votes for the candidate in consideration

3.5 ReadInformation class

This class is the one used to interface the program with an input file. Various parameters may be changed without recompilation needed since this approach has been adopted. In this class many methods are implemented for retrieving information from command line and input file. Some method takes care of returning the proper pointer to the method chosen by the user at run-time and loaded from a factory (see sections 3.1 and 3.2.3).

```
//! Helper class for reading the information needed from the command line
   and the input file
template<class Generator>
class ReadInfoGetPot{
private:
  GetPot cl;        //!< The GetPot member related to the command line
  GetPot GPfile;    //!< The GetPot member related to the input file to be
   read
  Generator gen;    //!< The RandomGenerator passed to the Creators
public:
  ReadInfoGetPot(const GetPot& cl_, const Generator& gen_): cl{cl_}, gen{
   gen_} {}

  //! Check whether the user is asking for the help of the executable
  void check_help();

  //! Look for the input file where to search every information needed for
  //! the manipulation process
  void read_input_file();

  //! Look for the information needed for constructing the GraphCreator
   requested
  //! \return a string related to the method to use
  std::string readInfoGraphCreator();

  //! Read the plugin libraries to be loaded
  void readPluginLibraries();

  //! Try to instantiate the requested method
```

```
    //! \return Returns a unique pointer to the GraphCreator base class
    std::unique_ptr<GraphCreatorBase> instantiateGraphCreator();

    //! Look for the ProbabilityDistribution needed for the generating the
    //! Resistance attribute of the Person in the social network.
    //! \return Returns a unique pointer to the ResistanceDistribution class
    std::unique_ptr<ResistanceDistribution<Generator>> readInfoResistanceDist
      ();

    //! Look for the ProbabilityDistribution needed for the generating the
    //! probability of voting of each Person in the social network.
    //! \return Returns a unique pointer to the VotingDistribution class
    std::unique_ptr<VotingDistribution<Generator>> readInfoVotingDist();

    //! Read parameters useful for the influence process
    //! \return A pair containing the number of steps used in the estimation
      of
    //!         the utility of each node and the number of influence rounds
    std::pair<std::size_t, std::size_t> readInfluenceOption();

    //! Check whether the user wants the results (in term of metrics)
    //! printed in a file in the out folder
    bool readInfoOutputResults();

    //! Check whether the user wants the graph printed in a file in the out
      folder
    bool readInfoPrintGraph();
};
```

The default input file used is located in the input folder. Most of the options of the manipulation process may be chosen at run-time since they are read from this file (or similar if correctly pointed executing the program).

Output : here the user indicates whether to output the final social network in a file in the out folder or not (*print_graph*) and may also choose to output the influence metrics in the aforementioned directory. (*output_results*).

Graph_option : all the characteristics of the Graph to be implemented

- *library*: indicates which dynamic libraries to be loaded;
- *graph_input*: chosen graph type;
- *Input_graph*: filename of the graph to be read if *graph_input* = *Input-file*;
- N: number of nodes (if requested by the Creator choosen);
- E: number of edges (if requested by the Creator choosen);

– option related to each specific graph type.

Person_option : the distributions used to generate each Person in the network may be chosen;

InfluenceOption : the number of influence rounds (*rounds*) and the maximum distance used to compute the utility of each node (*steps*) are chosen.

4 Conclusions

At the end of this project an architecture has been built and it allows to run many experiments on the subject without having to recompile the whole project. Instead, interfacing directly with the input file, as described in section 3.5, empower the user to change most of the parameters of the problem without any recompilation needed.

A part from that, there are still some topic that may be explored more in-depth.
In particular, other estimates' update function (eq. 1.3) may be analysed for better emulating the behaviour of users' preferences.
Furthermore, it may be interesting to examine other utility metrics: for instance, it may be useful to find a metric summarising a long-run utility instead of a one-round utility; on the contrary, it may be convenient to penalise a node who is picked several times as seed.
In addition, it may be analysed if huge graphs may be treated as simpler small graphs and algorithms may be parallelised.

Bibliography

[1] Hunt Allcott and Matthew Gentzkow. "Social Media and Fake News in the 2016 Election". In: *Journal of Economic Perspectives* 31.2 (May 2017), pp. 211–36. DOI: 10.1257/jep.31.2.211. URL: https://www.aeaweb.org/articles?id=10.1257/jep.31.2.211.

[2] Matteo Castiglioni, Andrea Celli, and Nicola Gatti. *Persuading Voters: It's Easy to Whisper, It's Hard to Speak Loud.* 2019. arXiv: 1908.10620 [cs.GT].

[3] Matteo Castiglioni et al. *Election Manipulation on Social Networks: Seeding, Edge Removal, Edge Addition.* 2020. arXiv: 1911.06198 [cs.AI].

[4] Bryan Wilder and Yevgeniy Vorobeychik. *Controlling Elections through Social Influence.* 2017. arXiv: 1711.08615 [cs.MA].

[5] Qingyun Wu et al. "Factorization Bandits for Online Influence Maximization". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (July 2019). DOI: 10.1145/3292500.3330874. URL: http://dx.doi.org/10.1145/3292500.3330874.

[6] Wenjing Duan, Bin Gu, and Andrew B. Whinston. "Informational Cascades and Software Adoption on the Internet: An Empirical Investigation". In: *MIS Quarterly* 33.1 (2009), pp. 23–48. ISSN: 02767783. URL: http://www.jstor.org/stable/20650277.

[7] Sinan Aral and Dylan Walker. "Identifying Influential and Susceptible Members of Social Networks". In: *Science* 337.6092 (2012), pp. 337–341. DOI: 10.1126/science.1215842. eprint: https://www.science.org/doi/pdf/10.1126/science.1215842. URL: https://www.science.org/doi/abs/10.1126/science.1215842.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A recursive model for graph mining". In: *SIAM Proceedings Series* 6 (Apr. 2004). DOI: 10.1137/1.9781611972740.43.

[9] C.R. Palmer and J.G. Steffan. "Generating network topologies that obey power laws". In: 1 (2000), 434–438 vol.1. DOI: 10.1109/GLOCOM.2000.892042.

[10] George Casella, Christian P. Robert, and Martin T. Wells. "Generalized Accept-Reject Sampling Schemes". In: *Lecture Notes-Monograph Series* 45 (2004), pp. 342–347. ISSN: 07492170. URL: http://www.jstor.org/stable/4356322.