Computer Science Department
VU Amsterdam, 1081HV Amsterdam

Bachelor Thesis

Evolving Graph Variations in the ST-Path Connectivity Problem

Author: Dovydas Vadišius (2744980)

1st supervisor: Ali Mehrabi
2nd reader: Femke van Raamsdonk

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

October 26, 2025

Contents

1 Abstract **3**

2 Introduction **4**
- 2.1 Related Work . . . 4
- 2.2 Contributions . . . 5

3 Evolving Graphs **6**
- 3.1 Model . . . 6
- 3.2 ST-Path Connectivity Problem . . . 6
- 3.3 Path-Finding Algorithms . . . 6
  - 3.3.1 One-Path Algorithm . . . 7
  - 3.3.2 Two-Path Algorithm . . . 8
- 3.4 New Theoretical Bounds . . . 9
  - 3.4.1 New Change Type: Edge Removal . . . 9
  - 3.4.2 New Change Type: Vertex Removal . . . 10

4 Experimental Results **12**
- 4.1 Framework . . . 12
  - 4.1.1 Model . . . 12
  - 4.1.2 Algorithm . . . 13
  - 4.1.3 Runner . . . 13
  - 4.1.4 Datasets . . . 14
  - 4.1.5 Testing Workflow . . . 15
- 4.2 Results . . . 15
  - 4.2.1 Model with Only Edge Swap . . . 15
  - 4.2.2 Models with Edge and Vertex Removals . . . 18
  - 4.2.3 Experiments on Datasets . . . 21

5 Conclusion **24**

A Appendix: Evolving Graph Model Implementation **26**
- A.1 Basic Model . . . 26
- A.2 Model Extensions . . . 28

B Appendix: Algorithm Implementations **30**
- B.1 One-path Algorithm . . . 30
- B.2 Two-path Algorithm . . . 31

C Appendix: Scripts for Running Experiments **36**
- C.1 Runner Class . . . 36
- C.2 run_experiment.py - Running a Single Experiment . . . 37

1 Abstract

Given a graph G and a graph-related problem, in the *evolving model of computation*, the graph G changes over time and the algorithm is unaware of the changes. The task is to design an algorithm that solves the problem on such an evolving graph G with a high probability. The algorithm can learn the changes by performing a small number of probes during each time step. In the ST-path connectivity problem, given an undirected graph G and two vertices S and T in G, the algorithm has to keep track of the path between S and T as the graph G evolves. The ST-path connectivity problem was first studied in 2012 by Anagnostopoulos, Kumar, Mahdian, Upfal, and Vandin, in a setting where an evolution step consists of removing a randomly selected edge in the graph and adding an edge between two vertices that were not connected before. Such a change type is called a random edge swap. Anagnostopoulos et al. give one-path and two-path algorithms, that solve the ST-path connectivity problem on such evolving graph, and present theoretical bounds on the probability that the returned ST-path is invalid. In this thesis, we present two natural extensions to this evolving graph model. More precisely, we introduce random edge removal, and random vertex removal change types and derive theoretical bounds for one-path and two-path algorithms with these model extensions. Finally, we provide implementations of the one-path and two-path algorithms and perform a comprehensive experimental evaluation of these algorithms on random evolving graphs with new change types, and on Contact Network and Wikipedia Link real-world datasets.

Keywords: evolving graphs, algorithms, path connectivity

2 Introduction

Graphs are widely used to represent complex real-life structures, including social and computer networks, recommendation systems, schedulers, web graphs, and many other relationships. Each of these applications has a particular set of problems that are solved using graph algorithms.

Multiple existing computing algorithms on graphs perform assuming the Classical Computation Paradigm, in which the algorithm is supplied with the fixed data, performs calculations, returns a result, and terminates. However, this computation paradigm is not always realistic, as it does not reflect the common characteristics of graphs: evolving, decentralized nature, and scale; thus, it is insufficient for modern data processing needs.

In certain applications, the graph is dynamically changing and the algorithm is unaware of any changes in the graph and has to find them by probing the graph. For example, as the web graph is constantly changing, search engines can learn about the changes by periodically crawling the web pages. Moreover, this setting applies to social networks (e.g., Facebook or X (formally Twitter)), in which the network is constantly evolving, and the vendor can query the graph for the changes by using a rate-limited API.

2.1 Related Work

In our setting, the algorithm is not aware of the changes in the dynamic graph, and we do not impose any restrictions on running time or memory usage, but rather the number of probes performed on the graph. The different algorithms are compared by the probability of outputting the correct answer or by outputting an answer which is closest to the correct solution for the current version of the dynamic graph. The dynamic graphs have been widely studied and various models have been developed for them. However, such models either inform the algorithm of the changes or try to solve a different kind of problem, and some of them impose particular restrictions.

In the Data Stream model [8], input arrives very rapidly. In this setting, it is hard to transmit the entire input to the program, compute sophisticated functions on large pieces of the input in the rate it is presented, and store all of the data in the long term. Due to the limited memory to store the input, algorithms have to work with only one or a few passes over the data. However, in this model, the algorithm assumes the knowledge of all the changes, and the limited resource memory is different since in our setting the limited resource is the number of queries used.

The Dynamic model [4] involves answering queries on dynamic graphs that include insertions and deletions of edges and vertices, while the algorithm must be faster than naive recomputation. Nevertheless, the algorithm is notified about the changes, and minimization of time and space complexity is one of the objectives in this model, while we do not limit these resources.

In Property Testing [9], the algorithm has to distinguish whether a graph has a particular property (e.g., whether a graph is bipartite or k-colorable) by performing a limited amount of probes on the graph. The model is useful when the property in question is NP-hard or when the input database is huge. However, property testing is limited only to the decision problems, and it imposes restrictions on the input graph, as graphs not satisfying the property are far different from those graphs that satisfy it. Otherwise, if the graph has

the property but is not far from the graph not having the property, it is allowed to decide in either way.

In the Multi-Armed Bandit model [6], there are K slot machines, and the algorithm can perform actions that involve pulling a lever from a selected slot machine. The slot machine produces a random reward which is sampled from the standard distribution, and the distributions vary between different slot machines. After each of these actions, the algorithm observes the given award, and its task is to maximize the total reward over a fixed amount of actions. Although the algorithms aim to discover an optimal solution under the uncertainty of the model and the algorithms do not have limitations on computational resources, they have a different goal that involves maximizing the expected payoff. Additionally, the model does not work on an underlying graph structure.

In Parametric Optimization Model [1], parameters of a graph such as edge weights are continuous functions of a real parameter λ called "time", and the algorithms aim to identify how the solution changes as the time λ varies, for instance, by computing a sequence of minimum spanning trees over time. The kinetic version allows changes of the graph that include edge insertions, deletions, and modifications of weight functions as time progresses, and the algorithms should efficiently maintain the optimal solution at each point in time. Such a model informs the algorithm of all the changes and puts time and space limitations, making it not suitable for our setting.

2.2 Contributions

None of these models reflect the requirements of the defined setting on evolving graphs with a limited amount of probes allowed. Therefore, Anagnostopoulos et al. [3] have proposed a new computational framework for this particular purpose. The framework captures the trade-off between the complexity of maintaining an up-to-date view and the quality of results with the available view. The algorithms in the framework are unaware of the changes and can only observe them by probing the graph.

One of the focuses of the framework is the path connectivity problem in unweighted graphs, where the set of edges of the graph changes over time without altering its size, and the algorithm should output the correct path with a high probability. This work is an extension of the work of Anagnostopoulos et al. [3], which provides an implementation of two path connectivity algorithms *one-path* and *two-path*, and augments the model by allowing the removal of edges and vertices in the graph. Such an extension would be relevant to multiple applications, including social networks for modeling broken connections or leaving users; or network reliability for analyzing the impact of node or edge failures. Extending the model with the removal of edges would keep the same theoretical error probability of invalid path of $O((n \ln n)^{-1/2})$ for the *one-path* algorithm and $O(\ln n/n)$ for the *two-path* algorithm. Furthermore, allowing the removal of vertices would result in a higher error probability $O((n/\ln n)^{-1/2})$ and $O(\ln^2 n/n)$ for the *one-path* and *two-path* algorithms, respectively. These theoretical bounds were validated by performing experiments on random graphs.

3 Evolving Graphs

3.1 Model

The unweighted graph model is followed as defined in [2] and was used in the work of Anagnostopoulos et al. [3]. In the model, time $t \in \mathbb{N}$ proceeds in discrete steps, and the graph during this step is G_t. The graph is gradually changing: the graph G_{t+1} at the next time moment is obtained by making a number of random changes to the graph G_t. At time moment t, let V_t denote the edge set of G_t, E_t denote the edge set of G_t, $n_t = |V_t|$, $m_t = |E_t|$. There are three types of allowed changes, and only the first type of change out of the following list was present in the original paper and is considered in Section 3.3:

- Moving an edge by a random swap. This change is performed by taking a uniformly chosen edge $(u, v) \in E_t$, a randomly chosen non-edge $(u', v') \notin E_t$, and setting $E_{t+1} = (E_t \setminus \{(u, v)\}) \cup \{(u', v')\}$.
- Removing a random edge. The change is performed by removing a uniformly chosen edge $(u, v) \in E_t$, and setting $E_{t+1} = E_t \setminus \{(u, v)\}$.
- Removing a random vertex. The change is performed by removing a randomly chosen vertex $v \in V_t$, setting $V_{t+1} = V_t \setminus \{v\}$, $E_{t+1} = E_t \setminus \{(u', v') \in E_t \mid v \in (u', v')\}$.

At each time step t, the algorithm can probe a small portion of the graph G_t. Each probe is a vertex probe: a vertex v is queried, and the algorithm learns about the set of neighbors of v.

After probing the graph, the algorithm must output a solution for the task. For the ST-path connectivity problem defined below, the quality of the algorithm is measured by the probability that it is a valid solution for graph G_t. There is no restriction on the running time or memory that the algorithm uses.

3.2 ST-Path Connectivity Problem

In this paper, we consider ST-path connectivity problem in unweighted connected graphs as defined in [3]: maintain a path between two fixed vertices $S, T \in V$, assuming one exists. At each time moment t, the algorithm must output a path $P_t \subseteq V$ such that the probability that P_t is not a valid ST-path in G_t is negligible.

For this problem, the interesting range of the parameters is assumed to be between $m = \Omega(n \ln n)$ and $m = O(n^{3/2})$. As $t \to \infty$, the distribution of the graph G_t approaches a uniformly randomly chosen graph chosen from n_t vertices and m_t edges. If $m = o(n \ln n)$, then there is a non-trivial probability that there is no ST-path in the graph, and if $m = w(n^{3/2})$, then the problem is simple, as either S and T are connected, or they have shared neighbors with a high probability.

3.3 Path-Finding Algorithms

A path-finding algorithm involves growing balls B_S and B_T around vertices S and T. Initially, $B_S = \{S\}$, and all vertices are marked as unvisited. In every time step, the algorithm picks an unvisited vertex $v \in B_S$ closest to S, marks v as visited, and performs

a probe on the graph that returns neighbors of v, which are added to B_S. The ball is grown for $R = \lceil \sqrt{c_0 n / \ln n} \rceil$ steps for a constant c_0. The similar construction applies to B_T, with $B_T = \{T\}$ initially. If the algorithm finds a vertex that is both in B_S and B_T, it outputs the path defined through this vertex, otherwise, it returns that there is no path between S and T.

3.3.1 One-Path Algorithm

In the *one-path* algorithm [3], the time is divided into phases of length $2R$. In each phase, the algorithm is run as described previously to find a path between S and T, and at any time step during the phase, the path computed in the last phase is outputted.

Theorem 1. *The one-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most $O((n \ln n)^{-1/2})$.*

Proof. Since $m = \Omega(n \ln n)$, by Chernoff and union bounds, each vertex will have a degree of at least $c \ln n$ for a sufficiently large constant c during the entire algorithm execution, with a probability of at least $1 - n^{-3}$. Similarly, with high probability, the degree of each vertex is $O(m/n) = O(\sqrt{n})$.

Once the set B_S of size $R = \Theta(\sqrt{n / \ln n})$ is constructed, its size is $|B_S| = O(R\sqrt{n}) = O(n/\sqrt{\ln n})$, since each degree is $O(\sqrt{n})$.

During the construction of B_S, an unvisited vertex $v \in B_S$ is selected and its neighbors are added to B_S. Since v has at least $c \ln n$ neighbors, uniformly distributed in V, the expected number of neighbors that are not currently in B_S is at least

$$c \ln n \cdot \frac{n - O(n/\sqrt{\ln n})}{n}.$$

By a Chernoff bound, the number of neighbors not in B_S is at least $c' \ln n$, for some constant c', thus, adding neighbors of v will increase the size of B_S by $c' \ln n$ vertices. Once the process finishes after R steps, $|B_S| \geq c'' R \ln n$, for some constant c''. The similar analysis for B_T also gives the size $|B_T|$ of at least $\Omega(R \ln n)$.

For each visited vertex v in B_T, the probability that no edge exists between v and B_S is upper bounded by

$$\left(1 - \frac{c'' R \ln n}{n}\right)^{c' \ln n} \leq e^{-c' c'' R \ln^2 n / n} = e^{-c' c'' \ln n},$$

where $c' \ln n$ is the number of neighbors of each visited node v in B_T.

Therefore, with high probability, the algorithm succeeds in finding a plausible path.

Additionally, we should check that the path computed at the end of the path-finding algorithm is still valid, as some edges can be removed during the execution. Since both B_S and B_T have diameter $O(\ln n)$, the length of the path is also $O(\ln n)$. The resulting path is invalid if, during the execution of $2R$ time steps, at least one of the existing $O(\ln n)$ edges is removed. The probability of removing the edge during the graph change is $1/m$, thus the probability of an invalid path is $O(2R \ln n / m) = O(\sqrt{n \ln n}/m) = O((n \ln n)^{-1/2})$. □

3.3.2 Two-Path Algorithm

The *two-path* path-finding algorithm [3] is similar to the *one-path* algorithm, except that firstly, two neighbors u_1 and u_2 of S and two neighbors v_1 and v_2 of T are picked. Then, four balls are grown around each of the vertices u_1, u_2, v_1, v_2, each for R steps. The balls around u_1 and v_1 are kept disjoint from balls around u_2 and v_2, by rejecting any neighbor that is already included in those balls. After $4R$ steps, the path defined through a vertex at the intersection of B_{u_1} and B_{v_1} is set to be a primary path, and the path defined through a vertex at the intersection of B_{u_2} and B_{v_2} is the secondary path.

In the *two-path* algorithm, the time is divided into phases of length $8R$. In the even time steps, the *two-path* path-finding algorithm is run to find a primary and a secondary path between S and T. In the odd time steps, the algorithm probes the vertices on the primary path in a round-robin fashion to discover missing edges. In every step, if there are no missing edges detected, the algorithm outputs the primary path, otherwise, the secondary path is outputted.

Lemma 2. *With probability of at least $1 - O((n \ln n)^{-1/2})$, there exists at least one path between S and T at the end of the execution of the algorithm, and the algorithm described succeeds in finding it.*

Proof. Since the degree of each vertex is $\sqrt{n}$ with a high probability, there would be at most $O(R\sqrt{n}) = O(n/\sqrt{\ln n})$ vertices that would not be added to a currently constructed ball due to disjointness of the balls. Therefore, the argumentation of Theorem 1 could be used to show that balls B_{u_1}, B_{v_1} and balls B_{u_2}, B_{v_2} intersect with a high probability, and the probability of getting an invalid path remains $O((n \ln n)^{-1/2})$. □

Theorem 3. *The two-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most $O\left(\frac{\ln n}{n}\right)$.*

Proof. The probability the outputted path by the algorithm is invalid at time step t is the sum of (1) the probability the primary path is outputted and is invalid at time step t, and (2) the probability that the secondary path is outputted and is invalid at time step t.

(1): Since the constructed primary path has length $O(\log n)$, and every edge is probed during the odd time steps of the current phase, the primary path is invalid and outputted only if any of the edges became removed in the last $O(\log n)$ steps. Since the probability of removing a single edge is $1/m$, by the union bound, the probability that the primary path is outputted and is invalid at time step t is

$$O\left(\ln^2 n \cdot \frac{1}{m}\right) = O\left(\frac{\ln n}{n}\right).$$

(2): If the secondary path is outputted and it is invalid, it must mean that at least one edge was removed from both primary and secondary paths in one of the time steps after the edge has been discovered in the previous phase, and before time moment t of the current phase. Thus, there are at most $16R$ relevant time steps. As each path has $O(\ln n)$ edges, the probability that the edge becomes removed from each of the paths is $O(16R \ln n/m) = O((n \ln n)^{-1/2})$. Since the two paths are disjoint, these events are negatively correlated, and the probability that the secondary path is outputted and is invalid is

$$O(((n \ln n)^{-1/2})^2) = O\left(\frac{1}{n \ln n}\right).$$

Adding these two bounds together, we get that the probability that the algorithm outputs an invalid path at time t is at most

$$O\left(\frac{\ln n}{n} + \frac{1}{n \ln n}\right) = O\left(\frac{\ln n}{n}\right).$$

□

3.4 New Theoretical Bounds

In the extension of Anagnostopoulos et al. [3] work, two new types of graph changes were added: removing an edge and removing a vertex, in addition to the existing change type of moving an edge by a random swap. These types of changes were formally defined in Section 3.1.

When performing a change, the model is set to randomly choose a change type between allowed possible change types. Since the changes should keep the interesting range of parameters of $m = \Omega(n \ln n)$ and $m = O(n^{3/2})$, the removal of the edge is allowed only if after an edge removal, the bound $m = \Omega(n \ln n)$ with constant $c \geq 1$ would remain valid. Similarly, vertex removal is allowed if a vertex whose removal would comply with the interesting range of parameters exists. Note that the compliance to the interesting range of parameters depends on degree of the removed vertex.

3.4.1 New Change Type: Edge Removal

The addition of edge removal change preserves the theoretical bounds of *one-path* and *two-path* algorithms given in Theorem 1 and Theorem 3, and these bounds are proved in the following two theorems.

Theorem 4. *By including the removal of a random edge into the list of possible model changes, the one-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most* $O((n \ln n)^{-1/2})$.

Proof. Since the new change type preserves the interesting range of parameters, we can apply the reasoning of the first part of Theorem 1 that relies on this fact to confirm that the *one-path* algorithm succeeds in finding a plausible path with a high probability.

In the reasoning that the computed path at the end of the algorithm is still valid, we need to reconsider the probability of removal of the edge during the graph change at the end of Theorem 1 proof. The edge removal change type can only remove a single edge at a time, thus the general probability of removing an edge remains $1/m$, and the probability of the invalid path is still $O((n \ln n)^{-1/2})$. □

Theorem 5. *By including the removal of a random edge into the list of possible model changes, the two-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most* $O\left(\frac{\ln n}{n}\right)$.

Proof. Firstly, the reasoning of Lemma 2 applies to show that the probability of getting an invalid path remains $O((n \ln n)^{-1/2})$, since the new change type preserves the interesting range of parameters.

Furthermore, since the probability of removing a single edge is still $1/m$, the reasoning of Theorem 3 can be applied in the same way to show that the probability that the algorithm outputs an invalid path at time t is at most $O\left(\frac{\ln n}{n}\right)$. □

3.4.2 New Change Type: Vertex Removal

Introducing a random vertex removal as a new change type worsens the theoretical bounds established in Theorems 1 and 3, as shown in the two following theorems.

Theorem 6. *By including the removal of a random vertex into the list of possible model changes, the one-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most $O((n/\ln n)^{-1/2})$.*

Proof. Since the new change type preserves the interesting range of parameters, the reasoning of Theorem 1 can be applied to confirm that the *one-path* algorithm succeeds in finding a plausible path with a high probability.

In the reasoning that the computed path at the end of the algorithm is still valid, the probability of edge removal during the graph change is altered. Considering vertex removal, since each edge is connected to two vertices, the probability of removing a particular edge becomes $2/n$ (larger than the previous probability of $1/m$), thus the probability of invalid path becomes $O(2R \ln n \cdot (2/n)) = O(\sqrt{n \ln n}/n) = O((n/\ln n)^{-1/2})$. □

Theorem 7. *By including the removal of a random vertex into the list of possible model changes, the two-path ST-path connectivity algorithm guarantees that for every t, the probability that the path outputted by the algorithm at time t is invalid is at most $O\left(\frac{\ln^2 n}{n}\right)$.*

Proof. The reasoning of Lemma 2 applies to show that the probability of getting an invalid path remains $O((n \ln n)^{-1/2})$, since the new change type preserves the interesting range of parameters.

To assess the probability that the outputted path by the algorithm is invalid at time step t, as in the proof of Theorem 3, we have to calculate the sum of (1) the probability the primary path is outputted and is invalid at time step t, and (2) the probability that the secondary path is outputted and is invalid at time step t.

(1): Following the argumentation of Theorem 3, since the probability of removing a single edge changes from $1/m$ to $2/n$, by the union bound, the probability that the primary path is outputted and is invalid at time step t becomes

$$O\left(\ln^2 n \cdot \frac{2}{n}\right) = O\left(\frac{\ln^2 n}{n}\right).$$

(2): Taking the reasoning from Theorem 3, there are at most $16R$ relevant time steps relevant to the calculation of the secondary outputted path being invalid. As each path has $O(\ln n)$ edges, the probability that the edge becomes removed from each of the paths is $O(16R \ln n \cdot (2/n)) = O((n/\ln n)^{-1/2})$. Since the two paths are disjoint, these events are negatively correlated, and the probability that the second path is outputted and is invalid is

$$O(((n/\ln n)^{-1/2})^2) = O\left(\frac{\ln n}{n}\right).$$

Adding these two bounds together, we get that the probability that the algorithm outputs an invalid path at time t is at most

$$O\left(\frac{\ln^2 n}{n} + \frac{\ln n}{n}\right) = O\left(\frac{\ln^2 n}{n}\right).$$

□

4 Experimental Results

4.1 Framework

In addition to setting theoretical bounds for the variations of the original model, the framework to run ST-path connectivity algorithm experiments on various models was developed, and the implementation is publicly available online.[1] The framework includes implementation of evolving unweighted graph models, *one-path* and *two-path* ST-connectivity algorithms, runner that performs interactions between a model and the algorithm, and two datasets.

4.1.1 Model

In the framework, a model stores the underlying evolving graph structure, and has a list of methods described below. Appendix A contains implementation of the evolving graph model.

- Random graph generation using Erdős–Rényi model [5]. Given generation parameters number of vertices n and number of edges m, the method selects n random distinct pairs of vertices and adds an edge between them. Each edge has the probability of $m/\binom{N}{2}$ to appear in the graph.
- Probing vertex v. The method returns vertices adjacent to v in the current state of the graph.
- Performing a change. The method selects randomly between allowed types of changes for the model from these three:
  - Swapping a random edge. Firstly, a random edge is selected and removed from the list of current edges. Then, a pair of random vertices without a connecting edge is picked, and an edge is added between these vertices.
  - Removing a random edge. This is performed by selecting and removing a random edge from the list of current edges. The action is allowed only if after removal of the edge bound $n \ln n \leq m$ holds.
  - Removing a random vertex. The action is allowed only if there exists a vertex after whose removal the bound $n \ln n \leq m \leq n^{3/2}$ holds. Since the existence of such vertices depends on their degree in the current state of the graph, the method up to 10 times picks a random vertex and checks whether its removal would retain the interesting range of parameters. If it does, the randomly picked vertex and its incident edges are removed from the graph. Otherwise, if removal of all 10 randomly picked vertices would violate the stated inequality, then the model performs a different change type.
- Validating a path to be an answer to the ST-path connectivity problem. If an empty path is provided, the validation succeeds if vertices S and T are not connected in the current state of the graph. Otherwise, if the path is non-empty, then it is valid if every edge of the path exists in the current graph.

[1] Implementation link: https://github.com/DovydasVad/evolving-graphs

- Importing edges. The method replaces the current edges with the list of edges given as the argument. This method is used when simulating an evolving graph from a dataset.
- Updating edges. The method removes and adds specific edges for the current time step that are given as arguments. This method is used when simulating an evolving graph from a dataset.

4.1.2 Algorithm

The *one-path* and *two-path* ST-path connectivity algorithms (Appendix B) in the framework follow the descriptions in Section 3.3. The Algorithm class supports the following methods:

- Getting input vertex for probing the current graph.
- Receiving result of the last probe. After this method call the algorithm advances to a new step of the current phase and prepares the next vertex to be probed.
- Outputting answer to the ST-path connectivity algorithm. If this method is called multiple times without calling methods described above, the algorithm should output identical path every time, as the outputted path is the path computed in the last phase of the algorithm.

4.1.3 Runner

The Runner class (Appendix C.1) initiates the interaction between the algorithm and the model. Initially, it either generates a random graph using the Erdős–Rényi model, or imports starting edges from the dataset. Afterwards, the runner performs a number of iterations, and each iteration consists of the following steps:

1. `probe_rate` probes are performed. In a single probe, the runner asks the algorithm for probe input v_P, provides v_P to the graph for the probe, and sends the result of the probe back to the algorithm.
2. Algorithm is asked to provide an answer to the ST-path connectivity problem. Then, the answer is validated by the algorithm.
3. The graph evolves: if the algorithm is performing on a random graph, the graph performs `change_rate` changes. Otherwise, if a dataset is simulated, the graph is updated by adding and removing edges as stated in the following step of the dataset configuration.

Throughout the execution of iterations, the runner stores the ratio of correct answers over the number of iterations, which corresponds to the probability of getting a correct answer from the algorithm.

To ensure the correctness of the main methods (graph generation, probing, change, path outputting, and validation), these methods were tested with unit tests against expected outcomes. The unit tests can be found in `text/` directory in the implementation repository.

4.1.4 Datasets

The two selected datasets portray real-world evolving graphs. Each dataset consists of an initial configuration with starting edges and vertices, and a number of time steps, indicating the evolution of the dataset. Each time step consists of a list of edges to be removed or added compared to the previous time step. In the following descriptions, the evolution rate of a dataset is set to be $\frac{|AE_t| + |RE_t|}{|E_{t-1}|}$, where AE_t and RE_t specify a list of added and removed edges at time moment t, respectively, and E_{t-1} is the list of edges in the graph at time moment $t - 1$.

Contact Network [10]. The dataset collected close proximity interactions between 789 volunteers in an American high school. The volunteers were asked to wear wireless TelosB motes around their necks for a day. Every 20 seconds (= 1 CPR), the motes broadcasted 'Hello' messages, with nearby nodes recording the messages. The contact network is defined at each time step having volunteers as vertices and edges as the interactions during time interval of 10 minutes. The starting points of two adjacent intervals differ by 20 seconds. The entire dataset consists of 762 868 interactions with a mean duration of 2.8 CPRs, and 1 334 time steps are used in the experiments. The majority of interactions and contacts are very short (80th percentile of interactions at 3 CPRs, 80th percentile of contacts at 15 CPRs), and even though about 80% of the total time is spent in interactions that are shorter than 5 min, short contacts (up to 5 min) represent only about 10% of the total time. The evolution rate is 2-5% (5th and 95th percentiles).

Wikipedia Links [11]. The dataset was collected by The Koblenz Network Collection [7] and includes the editing history (additions and removals of links to other articles) of 100 312 simple English Wikipedia articles over the duration of 10 years from September 2001 to September 2011. The Wikipedia Link network is defined at each time step as having articles as vertices and links in the time interval of 5 days as edges, resulting in 730 time steps. The graph is undirected for consistency with the Contact Network dataset and the random graphs. The evolution rate is 0-7.5% (5th and 95th percentiles, 20th percentile is 0.4%). The lower bound of the evolution rate of 0% is explained by the low popularity of simple English Wikipedia articles during the first year of the time range. At the end of the data collection period (September 2011), the average degree of the network is 32.5, and the diameter is 12.

The answer to the ST-path connectivity problem depends on the choice of vertices S and T: if these vertices have a high average degree throughout the execution of the algorithm, S and T are likely to be directly connected, or connected through a shared neighboring vertex; otherwise, if S and T have a low average degree, it is likely that during the algorithm execution, these vertices would not be connected at all. Therefore, S and T vertices are picked to satisfy the interesting range of parameters, that is, if D is the duration of the experiment in time steps, the sum of degrees over all time steps is at least $\frac{n \ln n}{n} \cdot D = D \ln n$, and at most $\frac{n^{3/2}}{n} \cdot D = D\sqrt{n}$. Due to the nature of the Contact Network dataset with multiple disconnected components, the S and T vertices

were picked such that their average degree is closer to the upper range of aforementioned range to increase the number of time steps when S and T are connected.

4.1.5 Testing Workflow

All experiments were performed by executing Python scripts that initiated Runner with different parameters for model and algorithm configurations. Appendix C.2 contains the script for running a single experiment, and scripts for running multiple experiments at once can be found in the framework repository. The experiment results were saved as a Python dictionary into a pickle format file with `.pkl` file extension, and another script was used to convert this file into a `.csv` format suitable for constructing figures. In this way, to get a different representation of the results it was not needed to rerun the experiments. Additionally, rerunning only a part of the experiment required only loading and modifying a part of the results dictionary.

4.2 Results

4.2.1 Model with Only Edge Swap

Firstly, the experiments on the models with only edge swap change type were performed to find a suitable parameter c_0, which determines the number of time steps R for growing the balls for the *one-path* and *two-path* algorithms. The experiments were performed on graphs generated with Erdős–Rényi model, with five different values for the number of vertices (100, 300, 1 000, 3 000, 10 000), three different values for the number of edges ($m = n \ln n$, $m = n^{3/2}$ for the boundary values of the interesting range of parameters, and $m = n^{4/3} + 0.6n$ for the middle number of vertices[2]). Each experiment was run for 10 000 iterations, in each iteration, the algorithm performs a single probe, then outputs the answer to the ST-path connectivity program, and the graph performs a single change of an edge swap. In the end, the probability of algorithm correctness is the ratio of correct answers over the number of 10 000 iterations, averaged across 100 different initial random graphs (200 random graphs for $n = 100$).

Figure 1 contains 6 subfigures, each with a different combination of the number of edges, and algorithm. In general, the probability of algorithm correctness increases with the number of vertices in the graph, which is consistent with the decreasing probability of invalid outputted path from the theoretical bound. When comparing different algorithms, *two-path* algorithm has lower correctness probability with smaller values of n, but the probability tends to increase quicker with greater values of n and overcomes *one-path* algorithm, which has a worse theoretical correctness bound.

Different values of constant c_0 can significantly impact algorithm correctness. When the number of edges is close to $m = n^{3/2}$ (bottom subfigures of Figure 1), the lower value of c_0 always results in a better algorithm correctness probability. With this number of edges, vertices S and T have a high likelihood of being directly connected or having a shared neighbor vertex, thus a smaller ball and a small phase length is enough to detect a short ST-path. With a smaller number of edges (middle and upper subfigures), the length of ST-path tends to be longer, thus a smaller ball size is not always enough to

[2]The addition of term $0.6n$ in $m = n^{4/3} + 0.6n$ ensures that the number of edges is in the middle of interesting range of parameters with a small value of vertices (e.g., $n = 100$).

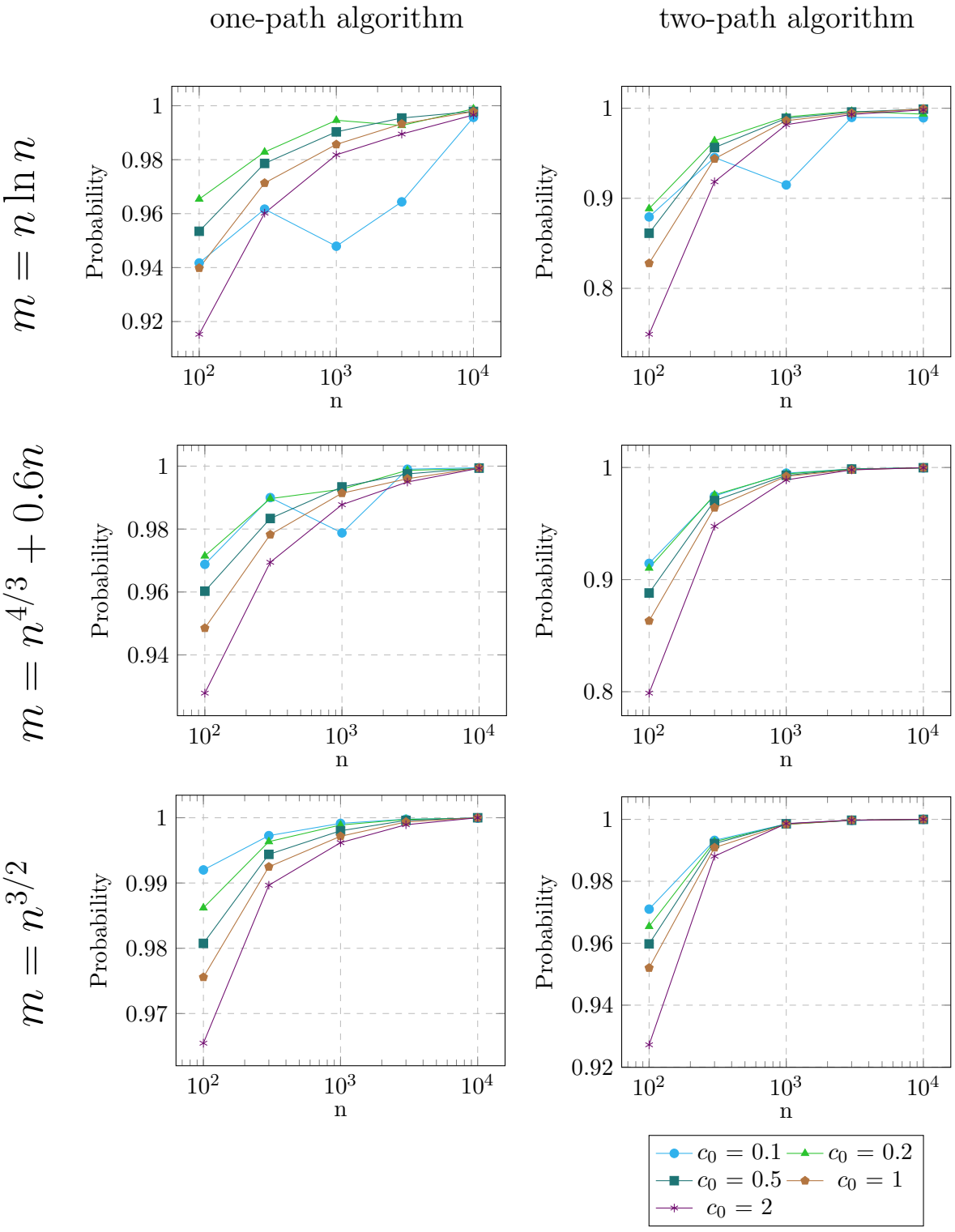

Figure 1: Correct output probability dependence on n, m and c_0.

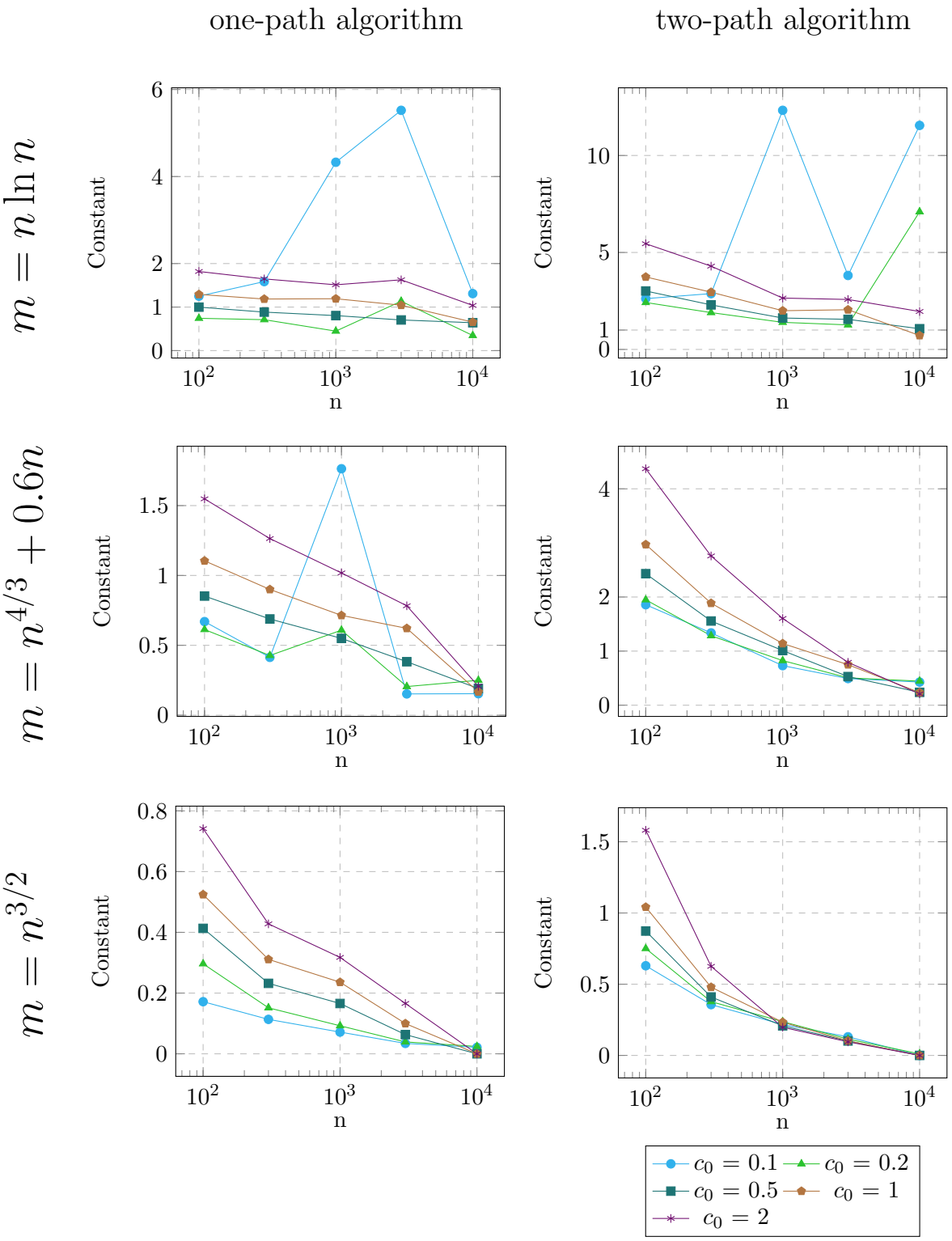

Figure 2: Theorethical bound constant dependence on n, m and c_0.

detect it. This is portrayed with $c_0 = 0.1$, in which an algorithm has the lowest algorithm correctness probability with $m = n \ln n$. While larger values of constant c_0 can help to detect longer ST-paths, this results in a larger phase length of the algorithm, due to which the probability of one edges of ST-path found in the previous phase becoming invalid increases. This depicts the trade-off between higher and lower constant c_0 values, and the algorithms have the highest correctness probability with c_0 value of 0.5-1.

A better picture with higher values of n can be seen in Figure 2, which compares the experimental probabilities against the theoretical bounds of invalid paths established in Section 3.3. The bounds are calculated by taking the ratio of number of incorrect answers given during the experiments and the expected number of incorrect answers as given by theoretical bounds ($(n \ln n)^{-1/2}$ and $\frac{\ln n}{n}$ for *one-path* and *two-path* algorithms, respectively). If the resulting constant is smaller than 1, then the algorithm performs better than the theoretical bound.

In all configurations with a larger number of graph vertices n, the constants gathered from the experiments tend to be smaller than 1. This is expected, as the bounds are defined for graphs with a large number of vertices. The jumps in constants with $c_0 = 0.1$ in the middle and upper subfigures of Figure 2 are connected with the fact that whether ST-path has a short distance and is detected by having a small ball depends on the graph configuration. A much larger number of experiments would be needed to determine the exact constant, but this would consume too many computational resources and is not too relevant to this work.

4.2.2 Models with Edge and Vertex Removals

The experimental framework includes implementations of 4 models, which differ only by allowed change types: Basic model (only edge swap), E model (edge swap and edge removal), V model (edge swap and vertex removal), EV model (edge swap, edge removal, and vertex removal). For the experiments, a fixed constant $c_0 = 0.5$ of *one-path* and *two-path* algorithms was selected, as it results in best algorithm correctness probability in a setting of Section 4.2.1. All other experiment parameters remained the same as in the previous section. During each of 10 000 iterations, the algorithm performs a single probe and then outputs the answer to the ST-path connectivity problem. At the end of an iteration, the graph evolves by performing a single change from the allowed change types defined in the model.

The experiment results are depicted in Figure 3. When $m = n \ln n$, the algorithm correctness probabilities in Basic and E models are identical, as the number of edges is at the interesting range of parameters in which edge removal is not possible. A similar reasoning applies to V and EV models, as the random order in which changes are performed (edge swap or vertex removal) makes a small impact on the correctness probability.

By including an edge removal into a set of allowed change types to the Basic model, the E model has a decreased algorithm correctness probability, this is especially noticeable when $m = n^{3/2}$. This trend has been observed and discussed in Section 4.2.1, as there are more edge removals that increase the average length of the ST-path.

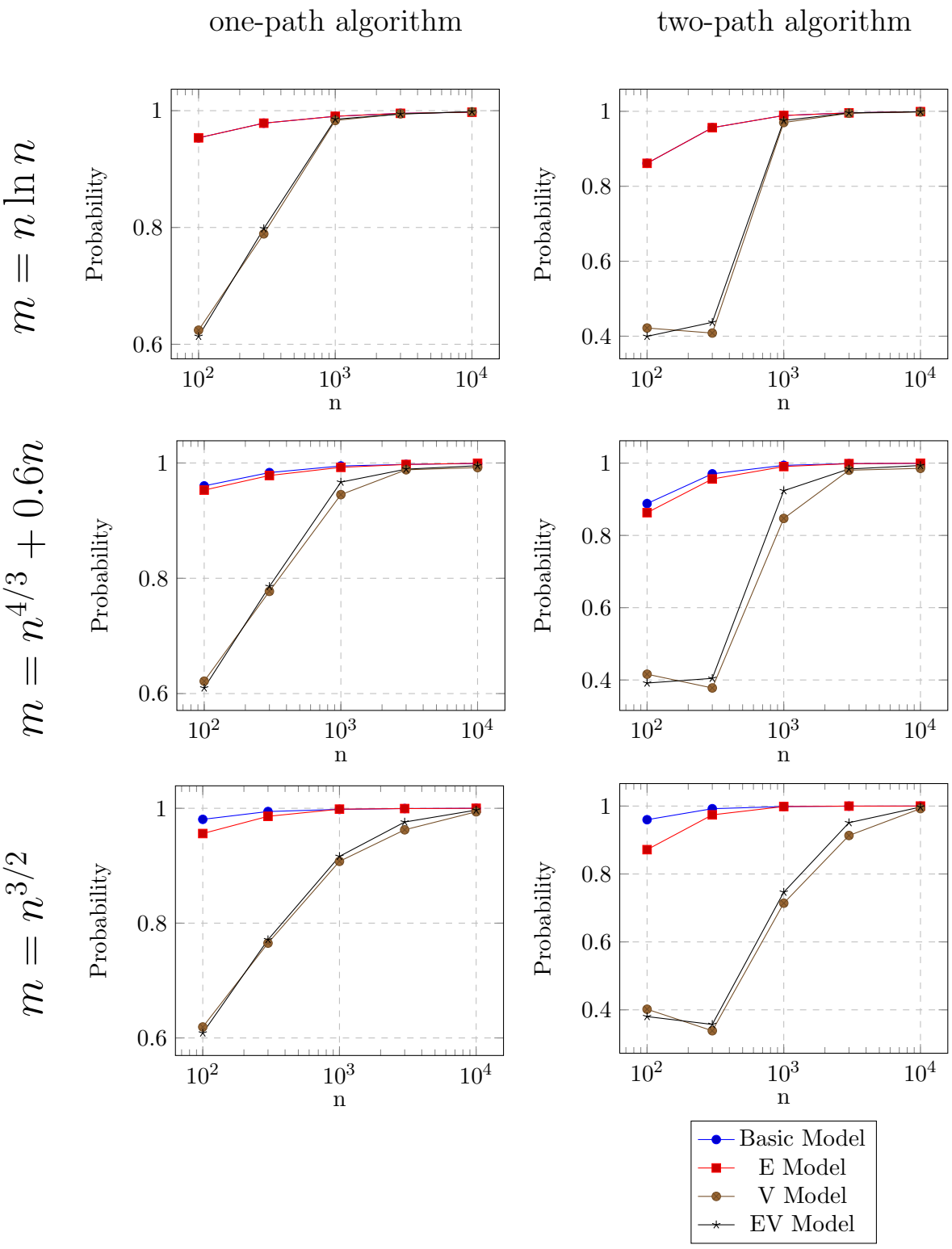

Figure 3: Correct output probability dependence based on available model change types.

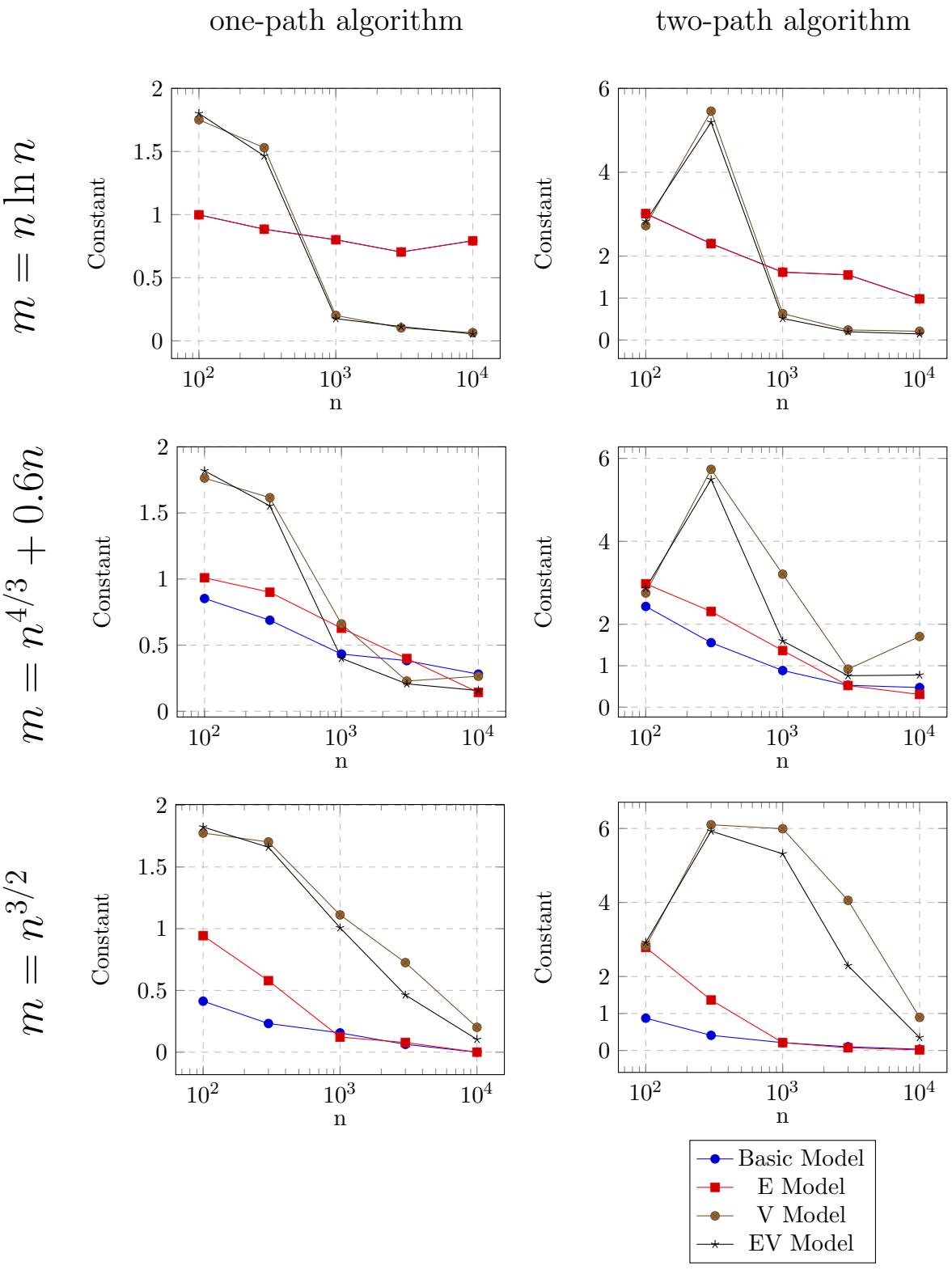

Figure 4: Theorethical bound constant dependence on model change types.

The introduction of a vertex removal into a set of allowed change types to the Basic model significantly reduces correctness probability with smaller values of n. However, such a probability of a V model is expected, as Section 3.4.2 proved a worse bound of an invalid output for both *one-path* and *two-path* algorithms.

Adding an edge removal type of change to the V model slightly improves the algorithm correctness probability, as vertex removals happen less often in EV model compared to V model, which decreases the probability of found ST-path invalidation.

Figure 4 compares the experimental probabilities with different models against the theoretical bounds of invalid paths proven in Section 3.3. With a larger number of vertices n, all models tend to approach constant close or smaller than 1, showing the validity of the theoretical bounds.

4.2.3 Experiments on Datasets

Lastly, *one-path* and *two-path* ST-path connectivity algorithms were run on Wikipedia and Contact datasets defined in Section 4.1.4. The experiments were performed with different algorithm constants c_0. As before, during every time step, the algorithm performs a number of probes (they vary between different experiments from 1 to 128), then it has to provide an answer to the ST-path connectivity algorithm, and afterwards the graph evolves by adding and removing multiple edges. The experiment results are displayed in Figure 5.

Contact Network In general, the algorithm correctness probability is lower than probability on random graphs, as the graph has a small number of vertices ($n = 789$), it evolves by performing more than a single change, and as vertices S and T are not connected during 9 continuous time intervals (in total during 229 out of 1078 time steps). Since connectivity of S and T is changing 16 times, ST-paths found in the previous phases are quite frequently invalidated.

Considering *one-path* algorithm, with a small number of allowed probes in each time step, algorithms with a smaller constant c_0 tend to perform better. Even with a probe rate of 1 algorithm with $c_0 = 0.05$ detects 75% of short paths (that could be detected with the corresponding ball size R if the algorithm had an infinite number of probes). In contrast, an algorithm with $c_0 = 5$ and a probe rate of 1 detects one-third of paths detected by an algorithm of $c_0 = 0.05$. This is due to the large R value with which the path found in the previous phase is quickly invalidated.

With a higher number of allowed probes during each time step, algorithms with a small constant c_0 tend to reach a plateau in the correctness probability. In this case, since the probe rate is higher than the value of $2R$ (phase length) the algorithm recalculates ST-path during every time step, thus the found path is never invalidated and the probability does not vary when the probe rate is at least 32. The correctness probability is bounded by the ball size, with which the algorithm does not detect longer ST-paths when they exist. On the other hand, algorithms with higher c_0 value start to perform better, as they can detect longer ST-paths, and with a higher probe rate a phase lasts for fewer time steps, thus fewer paths are invalidated.

Focusing on *two-path* algorithm, the algorithm correctness probability values seem to be shifted right in the X axis by a factor of 4 when compared to *one-path* algorithm - for

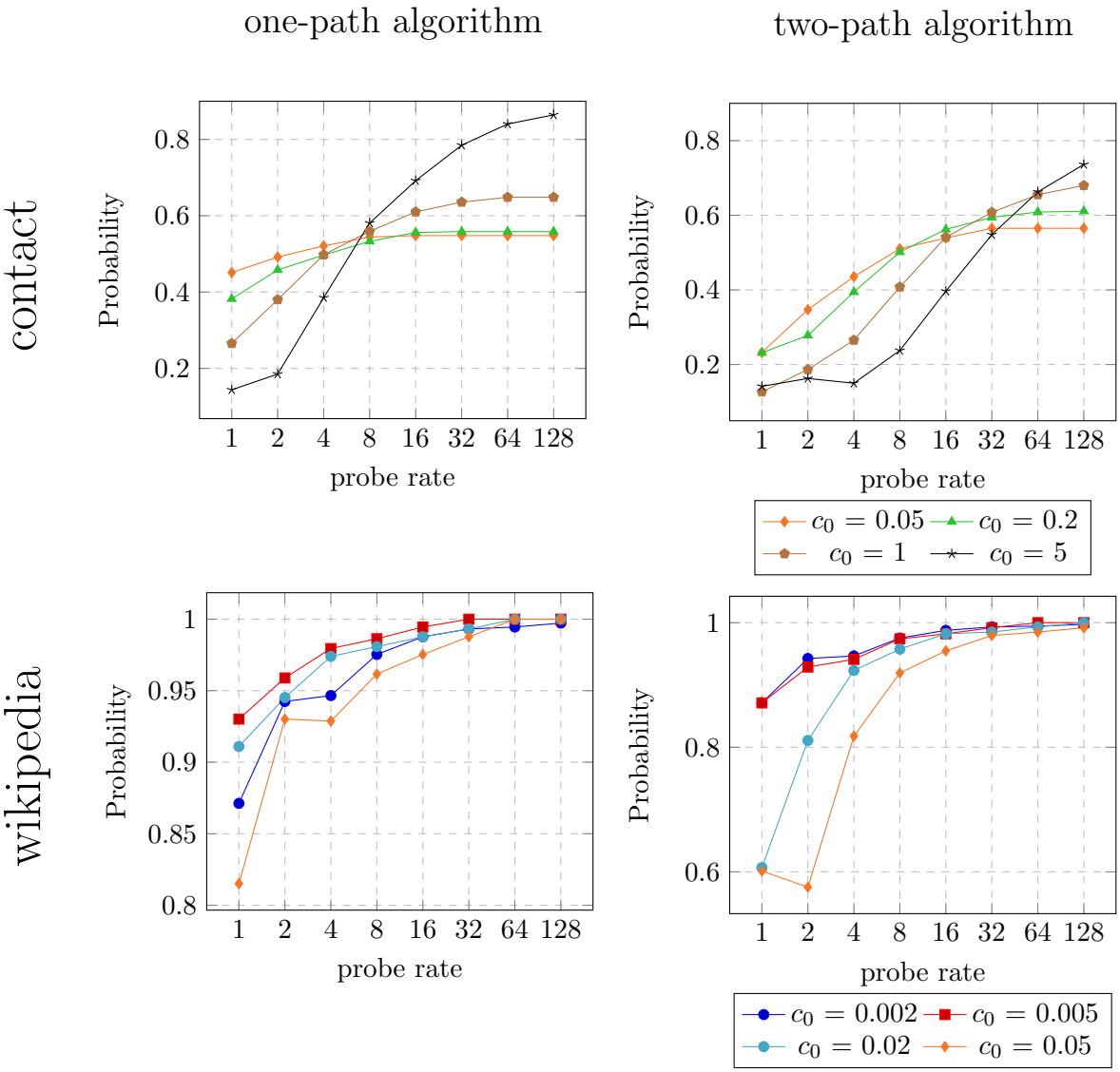

Figure 5: Algorithm correctness probability in Wikipedia and Contact datasets.

instance, all variations of *one-path* algorithm with different c_0 constants perform similarly when the probe rate is 8, while variations of *two-path* algorithm perform similarly with probe rate of 32. This is explained by the fact that the phase length of *two-path* algorithm is 4 times larger than in *one-path* algorithm, thus there are more time steps in which the path can be invalidated. Therefore, the left sides of the subfigures contain algorithms with lower correctness probability values than in *one-path* algorithm, and *two-path* algorithms with higher constant c_0 values still overtake algorithms with lower c_0 values.

However, the algorithms tend to have higher probability values when the probe rate is sufficiently high (when the probe rate is 64, *two-path* algorithm with $c_0 = 0.2$ performs better than equivalent *one-path* algorithm), this is since *two-path* algorithm has two pairs

(instead of one pair) of disjoint balls with size R that can cover more vertices and find longer ST-paths.

Wikipedia Links The Simple English Wikipedia dataset reassembles a real-world network, in which there are few vertices with a high average degree and many vertices with a low average degree. In the selected configuration of S and T vertices, S and T are disconnected in the first 272 out of 730 time steps, and during any later time step, there exists an ST-path. Even though the Simple English Wikipedia dataset contains 100 000 vertices and almost 700 000 edges at the end of the experiment (around 154 000 edges after 400 time steps), performing 32 or 64 probes during every time step is enough to reach correctness probabilities close to 1. When comparing results to experiments on random graphs, experiments on Wikipedia seem to perform worse, since the evolution rate is much higher than the probe rate.

Similarly as in Section 4.2.1, there is a tradeoff between low or high constant c_0 values, *one-path* algorithm performs best when $c_0 = 0.005$, and the best constant c_0 value for *two-path* algorithm is between 0.002 and 0.005.

The *two-path* algorithm performs only slightly worse, this shows that *two-path* algorithm is more suitable for graphs with a large number of vertices.

There is an instance in *two-path* algorithm when $c_0 = 0.05$ where an increase of the probe rate from 1 to 2 did not result in any algorithm correctness probability improvements. In this case, the algorithm detects the same number of continuous time intervals when vertices S and T are connected, regardless of whether the probe rate is 1 or 2.

Concluding experimental results, *one-path* ST-path connectivity algorithm performs better than *two-path* algorithm for small datasets, and the optimal value of constant c_0 depends on the properties of the graph, its size, and the probe rate.

5 Conclusion

In this work, new edge and vertex removal change types were considered. Theoretical bounds for evolving graph models including these change types were derived on *one-path* and *two-path* ST-path connectivity algorithms, and they were validated by performing experiments on random graphs. Moreover, these results were compared to experiments on Contact Network and Wikipedia Links datasets.

There are several possible future work directions. Firstly, other variations of model change types (e.g., not allowing edge swap, or having addition of an edge or a vertex) could be considered both by examining theoretical bounds and conducting experiments. Secondly, since the choice of constant c_0 impacts the probability that an algorithm produces a correct answer, a dynamic algorithm that detects a suitable value of c_0 on the fly could be developed. Lastly, as the algorithms were developed exactly as described in Anagnostopoulos et al. work [3], it would be interesting to examine whether there are heuristics in *one-path* and *two-path* algorithm implementations, and how those would improve the probability of the algorithms being correct experimentally. For instance, *two-path* algorithm in the odd time steps probes every vertex in a round-robin fashion to discover failures in the primary path, but it is enough to probe every second vertex.

References

[1] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. "Parametric and kinetic minimum spanning trees". In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. IEEE. 1998, pp. 596–605. DOI: 10.1109/SFCS.1998.743510.

[2] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. "Sorting and selection on dynamic data". In: *Theoretical Computer Science* 412.24 (2011), pp. 2564–2576. DOI: 10.1016/j.tcs.2010.10.003.

[3] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, Eli Upfal, and Fabio Vandin. "Algorithms on evolving graphs". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 2012, pp. 149–160. DOI: 10.1145/2090236.2090249.

[4] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. "Dynamic graph algorithms". In: *Algorithms and theory of computation handbook* 1 (1999), pp. 9–1.

[5] Paul Erdos and Alfréd Rényi. "On the evolution of random graphs". In: *Publ. math. inst. Hung. acad. sci* 5.1 (1960), pp. 17–60. DOI: 10.1515/9781400841356.38.

[6] Robert David Kleinberg. "Online decision problems with large strategy sets". PhD thesis. Massachusetts Institute of Technology, 2005. DOI: 1721.1/33092.

[7] Jérôme Kunegis. "KONECT – The Koblenz Network Collection". In: *Proc. Int. Conf. on World Wide Web Companion*. 2013, pp. 1343–1350. DOI: 10.1145/2487788.2488173.

[8] Shanmugavelayutham Muthukrishnan. "Data streams: Algorithms and applications". In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236. DOI: 10.1561/0400000002.

[9] Dana Ron. "Property testing: A learning theory perspective". In: *Foundations and Trends® in Machine Learning* 1.3 (2008), pp. 307–402. DOI: 10.1561/2200000004.

[10] Marcel Salathé et al. "A high-resolution human contact network for infectious disease transmission". In: *Proceedings of the national academy of sciences* 107.51 (2010), pp. 22020–22025. DOI: 10.1073/pnas.1009094108.

[11] *Wikipedia dynamic (simple) network dataset – KONECT*. Oct. 2017. URL: http://konect.cc/networks/link-dynamic-simplewiki.

A Appendix: Evolving Graph Model Implementation

Project repository: https://github.com/DovydasVad/evolving-graphs

A.1 Basic Model

Implementation of basic evolving graph model which has a random edge swap as the only change type.

```
import math
import random
from models.model import Graph

class UnweightedGraph(Graph):

    def init_specific(self, m, initialize = True):
        self.m = m
        self.adjacency_list = [set() for _ in range(self.n)]
        if initialize:
            self.interesting_range_check()
        self.start_vertex = 0
        self.end_vertex = self.n - 1
        self.edges = ListDict()
        self.initialize = initialize

    def interesting_range_check(self):
        if self.m < self.n * math.log(self.n):
            print("The range of parameters is not interesting: Number of edges m is lower than n log n")
            return 1
        if self.m > math.pow(self.n, 3/2):
            print("The range of parameters is not interesting: Number of edges m is higher than n^(3/2)")
            return 2
        return 0

    def construct_random_graph(self):
        """ Randomly creates m edges. Each edge has an equal probability to appear in the graph. """
        if not self.initialize:
            return
        for edge in range(self.m):
            non_edge_found = False
            while not non_edge_found:
                v = random.randint(0, self.n - 1)
                v2 = random.randint(0, self.n - 1)
                if v == v2 or v2 in self.adjacency_list[v]:
                    continue
                non_edge_found = True
                self.adjacency_list[v].add(v2)
                self.adjacency_list[v2].add(v)
                self.edges.add_item((min(v, v2), max(v, v2)))

    def probe(self, v):
        """ Vertex probe - returns the list of neighbors of v. """
        return self.adjacency_list[v]

    def change(self):
        """ The basic version of the algorithm has only edge swap as the possible change type. """
        self.change_swap_edge()

    def change_swap_edge(self):
        """ Random edge swap: remove a random edge, and insert an edge between two disconnected vertices. """
        removed_edge = self.edges.choose_random_item()
        v1 = removed_edge[0]
        v2 = removed_edge[1]
```

```
        if v1 not in self.adjacency_list[v2]:
            return RuntimeError
        self.edges.remove_item(removed_edge)

        v3 = v4 = -1
        nonedge_found = False
        while not nonedge_found:
            v3 = random.randint(0, self.n-1)
            v4 = random.randint(0, self.n-1)
            if v4 not in self.adjacency_list[v3] and v3 != v4:
                nonedge_found = True
        self.adjacency_list[v1].remove(v2)
        self.adjacency_list[v2].remove(v1)
        self.adjacency_list[v3].add(v4)
        self.adjacency_list[v4].add(v3)
        self.edges.add_item((min(v3, v4), max(v3, v4)))

    def validate(self, path):
        """
        Checks whether the answer of path from path_v1 to path_v2 is valid in the current state of the graph.
        If path does not follow the right structure, returns -1
        If the answer is invalid, returns 1
        If the answer is valid, returns 0

        The answer is valid if one of the conditions holds:
            1) path = [], and start and end vertices are not connected
            2) path != [], and the path between start and end vertices is valid """
        if len(path) > 0:
            if path[0] != self.start_vertex:
                print("Algorithm error: Returned Path does not start with start_vertex!")
                return -1
            if path[len(path)-1] != self.end_vertex:
                print("Algorithm error: Returned Path does not end with end_vertex!")
                return -1
            path_vertices = set()
            for v in path:
                if v < 0 or v >= self.n:
                    print("Algorithm error: Vertex out of range!")
                    return -1
                if v in path_vertices:
                    print("Algorithm error: Vertex appears twice!")
                    return -1
                path_vertices.add(v)

        if len(path) == 0: # Case 1: path = []
            visited = [0 for i in range(self.n)]
            visited[self.start_vertex] = 1
            new_vertices = [self.start_vertex]
            for v in new_vertices:
                for v2 in self.adjacency_list[v]:
                    if visited[v2] == 0:
                        visited[v2] = 1
                        new_vertices.append(v2)
                        if v2 == self.end_vertex:
                            return 1
        else: # Case 2: path != []
            for i in range(0, len(path)-1):
                if path[i+1] not in self.adjacency_list[path[i]]:
                    return 1
        return 0

    def import_edges(self, edges):
        """ Replaces the current edges with a list of edges. """
        for edge in self.edges.items:
            self.adjacency_list[edge[0]].remove(edge[1])
```

```
            if edge[0] != edge[1]:
                self.adjacency_list[edge[1]].remove(edge[0])
        self.edges = ListDict()
        for edge in edges:
            if edge[1] not in self.adjacency_list[edge[0]]:
                self.adjacency_list[edge[0]].add(edge[1])
            if edge[0] not in self.adjacency_list[edge[1]]:
                self.adjacency_list[edge[1]].add(edge[0])
            self.edges.add_item((min(edge[0], edge[1]), max(edge[0], edge[1])))

    def update_edges(self, new_edges, removed_edges):
        """ Updates the current edges by adding a list of new edges and removing another list of edges. """
        for edge in removed_edges:
            self.adjacency_list[edge[0]].remove(edge[1])
            self.edges.remove_item((min(edge[0], edge[1]), max(edge[0], edge[1])))
            if edge[0] != edge[1]:
                self.adjacency_list[edge[1]].remove(edge[0])
        for edge in new_edges:
            if edge[1] not in self.adjacency_list[edge[0]]:
                self.adjacency_list[edge[0]].add(edge[1])
            if edge[0] not in self.adjacency_list[edge[1]]:
                self.adjacency_list[edge[1]].add(edge[0])
            self.edges.add_item((min(edge[0], edge[1]), max(edge[0], edge[1])))

    def set_start_vertex(self, v):
        self.start_vertex = v

    def set_end_vertex(self, v):
        self.end_vertex = v

# Data structure that supports addition, removal, random selection in constant time
class ListDict(object):
    def __init__(self):
        self.item_to_position = {}
        self.items = []

    def add_item(self, item):
        if item in self.item_to_position:
            return
        self.items.append(item)
        self.item_to_position[item] = len(self.items) - 1

    def remove_item(self, item):
        position = self.item_to_position.pop(item)
        last_item = self.items.pop()
        if position != len(self.items):
            self.items[position] = last_item
            self.item_to_position[last_item] = position

    def choose_random_item(self):
        return random.choice(self.items)
```

Listing 1: Basic model implementation

A.2 Model Extensions

Evolving graph models with extended change types only differ in `change()` function implementation, and have specific methods for random edge and random edge removals. These implementations are presented in the listings below.

```
    def change(self):
        """ Picks an allowed model change type. """
```

```
        possible_actions = ["swap-edge"]
        # vertex removal is only allowed if
        # 1) the interesting range of parameters would be retained,
        # 2) at least one non-edge would still exist (corner case for small n)
        # 3) current vertex count >= 0.1 * original vertex count
        if self.n >= 0.1 * self.initial_n:
            for i in range(10):
                v_id = random.randint(0, self.initial_n - 1)
                if v_id not in self.active_vertices or v_id == self.start_vertex or v_id == self.end_vertex:
                    continue
                if (self.n - 1) * (self.n - 2) / 2 != self.m - len(self.adjacency_list[v_id]) and self.
                    ↪ interesting_range_check(m_subtraction = len(self.adjacency_list[v_id]), n_subtraction =
                    ↪ 1) == 0:
                    possible_actions.append("remove-vertex")
                    break
        if self.interesting_range_check(m_subtraction = 1) == 0:
            possible_actions.append("remove-edge")

        action = possible_actions[random.randint(0, len(possible_actions) - 1)]
        if action == "swap-edge":
            self.change_swap_edge()
        elif action == "remove-edge":
            self.change_remove_edge()
        elif action == "remove-vertex":
            self.change_remove_vertex()
```

Listing 2: Method for picking an allowed change type

```
    def change_remove_edge(self):
        """ Random edge removal. """
        removed_edge = self.edges.choose_random_item()
        v1 = removed_edge[0]
        v2 = removed_edge[1]
        if v1 not in self.adjacency_list[v2]:
            return RuntimeError
        self.m = self.m - 1
        self.edges.remove_item(removed_edge)
        self.adjacency_list[v1].remove(v2)
        self.adjacency_list[v2].remove(v1)
```

Listing 3: Method for removing a random edge

```
    def change_remove_vertex(self):
        """ Random vertex removal. """
        vertex_found = False
        while not vertex_found:
            v_id = random.randint(0, self.initial_n - 1)
            if v_id in self.active_vertices and v_id != self.start_vertex and v_id != self.end_vertex:
                if (self.n - 1) * (self.n - 2) / 2 != self.m - len(self.adjacency_list[v_id]) and self.
                    ↪ interesting_range_check(m_subtraction = len(self.adjacency_list[v_id]), n_subtraction =
                    ↪ 1) == 0:
                    vertex_found = True

        self.n = self.n - 1
        self.m = self.m - len(self.adjacency_list[v_id])
        for v in self.adjacency_list[v_id]:
            self.adjacency_list[v].remove(v_id)
            self.edges.remove_item((min(v, v_id), max(v, v_id)))
        self.adjacency_list[v_id] = []
        self.active_vertices.remove(v_id)
```

Listing 4: Method for removing a random vertex

B Appendix: Algorithm Implementations

B.1 One-path Algorithm

```python
"""
Phase length: 2R.

Steps:
    Two balls around start and end vertices are grown, each for R steps.
    Every step outputs the index of unvisited vertex that is closest to center of a currently grown ball.
    At the last (2R-1)th step, path between start and end vertices is constructed, if intersection of the balls is non-empty.

Output:
    Last found path (possibly empty) of the last phase.
"""

import math
from algorithms.algorithm import Algorithm

class AlgorithmOnePath(Algorithm):
    def __init__(self, c, n):
        self.name = "one_path"
        self.R = math.ceil(math.sqrt((c * n)/math.log(n)))
        self.phase_length = 2 * self.R
        self.start_vertex = 0
        self.end_vertex = n - 1
        self.last_path = []
        self.path = []
        self.phase_position = 0
        self.ball_S = []
        self.ball_T = []
        self.parent_S = [-1 for _ in range(n)]
        self.parent_T = [-1 for _ in range(n)]

    def get_probe_input(self):
        if self.phase_position == 0:
            self.path = []
            self.ball_S = [self.start_vertex]
            self.ball_T = []
            self.curr_ball = self.ball_S
        elif self.phase_position == self.R:
            self.ball_T = [self.end_vertex]
            self.curr_ball = self.ball_T

        # it is intended to repeatedly take the last vertex if the ball is not growing anymore, as this is a simple implementation as described in the paper
        probe_vertex = self.curr_ball[min(self.phase_position % self.R, len(self.curr_ball) - 1)]
        self.last_probe = probe_vertex
        return probe_vertex

    def set_probe_result(self, probe_result):
        for v in probe_result:
            if v not in self.curr_ball:
                self.curr_ball.append(v)
                if self.phase_position < self.R:
                    self.parent_S[v] = self.last_probe
                else:
                    self.parent_T[v] = self.last_probe

        if len(self.path) == 0:
            ball_intersection = list(set(self.ball_S) & set(self.ball_T))
            if len(ball_intersection) >= 1:
                self.path = self.construct_path(ball_intersection[0])
```

```
            elif self.end_vertex in self.ball_S:
                self.path = self.construct_path(self.end_vertex)
            elif self.start_vertex in self.ball_T:
                self.path = self.construct_path(self.start_vertex)

        self.phase_position = self.phase_position + 1
        if self.phase_position == 2 * self.R:
            self.last_path = self.path

            self.phase_position = 0

    def answer(self):
        """ Output answer for ST-path connectivity problem. """
        return self.last_path

    def construct_path(self, middle_vertex):
        """
        Constructs a path between start_vertex and end_vertex through a middle_vertex, which belongs to
            ↪ intersection of two balls around a neighbor of start_vertex and end_vertex.
        This is done by concatenating to paths:
            1) path (start_vertex, middle_vertex)
            2) path (middle_vertex, end_vertex)
        """
        path = []
        # 1) construct path (middle_vertex, start_vertex) by going backwards (through parents) from
            ↪ middle_vertex to start_vertex
        curr_vertex = middle_vertex
        while curr_vertex != self.start_vertex:
            next_vertex = self.parent_S[curr_vertex]
            path.append(next_vertex)
            curr_vertex = next_vertex
        # reverse (middle_vertex, start_vertex) to (start_vertex, middle_vertex)
        path.reverse()
        path.append(middle_vertex)

        # 2) construct path (middle_vertex, end_vertex)
        if middle_vertex != self.end_vertex:
            curr_vertex = middle_vertex
            while curr_vertex != self.end_vertex:
                next_vertex = self.parent_T[curr_vertex]
                path.append(next_vertex)
                curr_vertex = next_vertex

        return path
```

Listing 5: One-path algorithm implementation

B.2 Two-path Algorithm

```
"""
Phase length: 8R.

Even time steps:
    Check validity of the primary path - check whether edges of previously found path still exist.
    The vertices of the path are checked in a round-robin fashion.

    Exception 1: at time step 0, get neighbors u1 and u2 (centers of balls) of starting vertex S.
    Exception 2: at time step 1, get neighbors v1 and v2 of end vertex T.

Odd time steps:
    Grow balls around u1, u2, v1, v2. Each ball is grown for R steps.
    Balls around u2 and v2 are disjoint from the balls around u1 and v1.
```

```
14
15      After last (8R-1)th step:
16          Primary path is set to path (u1, v1)
17          Secondary path is set to path (u2, v2)
18
19      Exception: at time step 2, ball around u1 is grown.
20
21  Output:
22      If primary path is valid: last primary path
23      If primary path has been invalidated: last secondary path
24  """
25
26  import math
27  from algorithms.algorithm import Algorithm
28
29  class AlgorithmTwoPath(Algorithm):
30      def __init__(self, c, n):
31          self.name = "two_path"
32          self.R = math.ceil(math.sqrt((c * n)/math.log(n)))
33          self.phase_length = 8 * self.R
34          self.start_vertex = 0
35          self.end_vertex = n - 1
36          self.last_primary_path = []
37          self.last_secondary_path = []
38          self.primary_path = []
39          self.secondary_path = []
40          self.phase_position = 0
41          self.ball_v1 = []
42          self.ball_v2 = []
43          self.ball_u1 = []
44          self.ball_u2 = []
45          self.parent_v = [-1 for _ in range(n)]
46          self.parent_u = [-1 for _ in range(n)]
47
48
49      def get_probe_input(self):
50          # start of the phase: initiate empty balls, try to get neighbors of the start_vertex to get values of
                ↪ u1 and u2
51          if self.phase_position == 0:
52              self.ball_u1 = []
53              self.ball_u2 = []
54              self.ball_v1 = []
55              self.ball_v2 = []
56              self.primary_path = []
57              self.secondary_path = []
58              self.curr_ball = self.ball_u1
59              self.primary_valid = True
60              self.last_probe = self.start_vertex
61              return self.start_vertex
62          # 2nd probe: get neighbors of end_vertex to get values of v1 and v2
63          elif self.phase_position == 1:
64              self.last_probe = self.end_vertex
65              return self.end_vertex
66
67          # even step: check validity of the primary path by performing round robin on the vertices of primary
                ↪ path
68          if self.phase_position % 2 == 0 and self.phase_position != 2:
69              v_index = (self.phase_position // 2 - 2) % (len(self.last_primary_path) - 1)
70              if len(self.last_primary_path) == 0:
71                  self.last_probe = self.start_vertex
72              else:
73                  self.last_probe = self.last_primary_path[v_index]
74
75          # odd step: grow the current ball by probing nearest unvisited vertex
76          else:
```

```
            if len(self.curr_ball) == 0:
                self.last_probe = 0
            else:
                v_index = ((self.phase_position % (2 * self.R)) + 1) // 2
                self.last_probe = self.curr_ball[min(v_index, len(self.curr_ball) - 1)]

        return self.last_probe

    def set_probe_result(self, probe_result):
        probe_result = list(probe_result)
        # step 0: set values of u1 and u2
        if self.phase_position == 0:
            if len(probe_result) >= 1:
                self.ball_u1.append(probe_result[0])
                self.parent_u[probe_result[0]] = self.start_vertex
                if probe_result[0] == self.end_vertex:
                    self.primary_path = self.construct_path(probe_result[0])
            if len(probe_result) >= 2:
                self.ball_u2.append(probe_result[1])
                self.parent_u[probe_result[1]] = self.start_vertex
                if probe_result[1] == self.end_vertex:
                    self.secondary_path = self.construct_path(probe_result[1])

        # step 1: set values of v1 and v2
        elif self.phase_position == 1:
            ball_v1_set = False
            ball_v2_set = False
            # v1 and v2 should be disjoint from u1 and u2
            for v in probe_result:
                if ball_v1_set == False and v not in self.ball_u2:
                    ball_v1_set = True
                    self.ball_v1.append(v)
                    self.parent_v[v] = self.end_vertex
                    if v == self.end_vertex:
                        self.primary_path = self.construct_path(v)
                elif ball_v2_set == False and v not in self.ball_u1:
                    ball_v2_set = True
                    self.ball_v2.append(v)
                    self.parent_v[v] = self.end_vertex
                    if v == self.start_vertex:
                        self.secondary_path = self.construct_path(v)
                if ball_v1_set and ball_v2_set:
                    break

        # even step: validate primary path of the last phase
        elif self.phase_position % 2 == 0 and self.phase_position != 2:
            if self.primary_valid and len(self.last_primary_path) > 0:
                v_index = (self.phase_position // 2 - 2) % (len(self.last_primary_path) - 1)
                if self.last_primary_path[v_index + 1] not in probe_result:
                    self.primary_valid = False

        # odd step (or step 2): grow the current ball, while not including vertices that would disrupt
            ↪ disjointness condition
        elif len(self.curr_ball) > 0:
            for v in probe_result:
                if v not in self.curr_ball:
                    disjointness_valid = True
                    if self.phase_position < 2 * self.R and (v in self.ball_u2 or v in self.ball_v2):
                        disjointness_valid = False
                    if 2 * self.R <= self.phase_position < 4 * self.R and (v in self.ball_u1 or v in self.
                        ↪ ball_v1):
                        disjointness_valid = False
                    if 4 * self.R <= self.phase_position < 6 * self.R and (v in self.ball_u2 or v in self.
                        ↪ ball_v2):
                        disjointness_valid = False
```

```
                if 6 * self.R <= self.phase_position and (v in self.ball_u1 or v in self.ball_v1):
                    disjointness_valid = False
                if disjointness_valid:
                    self.curr_ball.append(v)
                    if self.phase_position < 4 * self.R:
                        self.parent_u[v] = self.last_probe
                    else:
                        self.parent_v[v] = self.last_probe

        # Try to construct primary_path, this is possible when:
        # 1) intersection of balls around start_vertex and end_vertex is non-empty, or
        # 2) currently grown ball has reached start_vertex or end_vertex
        if len(self.primary_path) == 0:
            ball_intersection = list(set(self.ball_v1) & set(self.ball_u1))
            if len(ball_intersection) >= 1:
                self.primary_path = self.construct_path(ball_intersection[0])
            elif self.end_vertex in self.ball_u1:
                self.primary_path = self.construct_path(self.end_vertex)
            elif self.start_vertex in self.ball_v1:
                self.primary_path = self.construct_path(self.start_vertex)

        # Try to construct secondary_path
        if len(self.secondary_path) == 0:
            ball_intersection = list(set(self.ball_v2) & set(self.ball_u2))
            if len(ball_intersection) >= 1:
                self.secondary_path = self.construct_path(ball_intersection[0])
            elif self.end_vertex in self.ball_u2:
                self.secondary_path = self.construct_path(self.end_vertex)
            elif self.start_vertex in self.ball_v2:
                self.secondary_path = self.construct_path(self.start_vertex)


        self.phase_position = self.phase_position + 1
        # change ball that will be constructed in the new segment of the current phase
        if self.phase_position == 2 * self.R:
            self.curr_ball = self.ball_u2
        if self.phase_position == 4 * self.R:
            self.curr_ball = self.ball_v1
        if self.phase_position == 6 * self.R:
            self.curr_ball = self.ball_v2
        if self.phase_position == 8 * self.R:
            # End of phase: change path validity and primary_path
            self.curr_ball = self.ball_u1
            self.last_primary_path = self.primary_path
            self.last_secondary_path = self.secondary_path
            self.primary_valid = True
            self.phase_position = 0

    def answer(self):
        """ Output answer for ST-path connectivity problem. """
        if self.primary_valid:
            return self.last_primary_path
        else:
            return self.last_secondary_path

    def construct_path(self, middle_vertex):
        """
        Constructs a path between start_vertex and end_vertex through a middle_vertex, which belongs to
            ↪ intersection of two balls around a neighbor of start_vertex and end_vertex.
        This is done by concatenating to paths:
            1) path (start_vertex, middle_vertex)
            2) path (middle_vertex, end_vertex)
        """
        path = []
```

```
202         # 1) construct path (middle_vertex, start_vertex) by going backwards (through parents) from
                ↪ middle_vertex to start_vertex
203         curr_vertex = middle_vertex
204         while curr_vertex != self.start_vertex:
205             next_vertex = self.parent_u[curr_vertex]
206             path.append(next_vertex)
207             curr_vertex = next_vertex
208         # reverse (middle_vertex, start_vertex) to (start_vertex, middle_vertex)
209         path.reverse()
210         path.append(middle_vertex)
211 
212         # 2) construct path (middle_vertex, end_vertex)
213         if middle_vertex != self.end_vertex:
214             curr_vertex = middle_vertex
215             while curr_vertex != self.end_vertex:
216                 next_vertex = self.parent_v[curr_vertex]
217                 path.append(next_vertex)
218                 curr_vertex = next_vertex
219         return path
```

Listing 6: Two-path algorithm implementation

C Appendix: Scripts for Running Experiments

C.1 Runner Class

```python
"""
Runner class implementation.
Performs an experiment by performing interactions between algorithm and the model.
"""

from tqdm import tqdm
from algorithms.algorithm import Algorithm

def visualize_result(algorithm: Algorithm, answer_correct: bool):
    print_char = '.'
    if answer_correct:
        if hasattr(algorithm, 'primary_valid') and not algorithm.primary_valid:
            print_char = '_'
    else:
        print_char = 'F'
        if hasattr(algorithm, 'primary_valid') and not algorithm.primary_valid:
            print_char = 'S'
    print(print_char, end = "")


class Runner:
    def __init__(self, probe_rate, change_rate, algorithm, graph, use_dataset = False, dataset = None):
        self.correct_answers = 0
        self.correct_answers_after_1st_phase = 0
        self.total_iterations = 0
        self.probe_rate = probe_rate
        self.change_rate = change_rate
        self.algorithm = algorithm
        self.graph = graph
        self.use_dataset = use_dataset
        self.dataset = dataset
        self.count_correct_empty = 0
        self.count_correct_path = 0
        self.count_incorrect_empty = 0
        self.count_incorrect_path = 0
        if use_dataset:
            algorithm.set_start_vertex(dataset["start_vertex"])
            algorithm.set_end_vertex(dataset["end_vertex"])
            graph.set_start_vertex(dataset["start_vertex"])
            graph.set_end_vertex(dataset["end_vertex"])

    def run(self, iterations, visualization_step):
        self.total_iterations += iterations
        for iteration in tqdm(range(iterations), disable = (not self.use_dataset)):
            # perform probes
            for j in range(self.probe_rate):
                v = self.algorithm.get_probe_input()
                self.algorithm.set_probe_result(self.graph.probe(v))

            # get and validate answers from models
            answer = self.algorithm.answer()
            answer_correct = (self.graph.validate(answer) == 0)
            if answer_correct:
                self.correct_answers += 1
                if iteration >= self.algorithm.phase_length:
                    self.correct_answers_after_1st_phase += 1
                if answer == []:
                    self.count_correct_empty += 1
                else:
                    self.count_correct_path += 1
```

```
            else:
                if answer == []:
                    self.count_incorrect_empty += 1
                else:
                    self.count_incorrect_path += 1
            if visualization_step != -1 and iteration % visualization_step == 0:
                visualize_result(self.algorithm, answer_correct)

            # perform changes in the model
            if self.use_dataset:
                if iteration == 0:
                    self.graph.import_edges(self.dataset["edges"][iteration])
                else:
                    self.graph.update_edges(self.dataset["new_edges"][iteration], self.dataset["removed_edges"][
                        ↪ iteration])
            else:
                for j in range(self.change_rate):
                    self.graph.change()

    def get_total_iterations(self):
        return self.total_iterations

    def get_correct_answers(self):
        return self.correct_answers

    def get_correct_answers_after_1st_phase(self):
        return self.correct_answers_after_1st_phase
```

Listing 7: Runner Class Implementation

C.2 run_experiment.py - Running a Single Experiment

```
"""
Runs a single experiment with a particular algorithm and model.
Multiple iterations are performed, in a single iteration:
    1) algorithm performs probe_rate probes on the model (graph)
    2) algorithm provides the answer
    3) model validates the answer
    4) model makes change_rate changes

Calling example for random graphs:
python3 run_experiment.py --alg=one --n=1000 --m=15000 --c0=0.5 --iterations=10000 --change=1 --probe=1 --
    ↪ rand_seed=0 --model=basic

Calling example for dataset:
python3 run_experiment.py --alg=one --c0=0.5 --change=1 --probe=5 --dataset=wikipedia
"""

import argparse
import os
import pickle

from models.unweighted_model import UnweightedGraph
from models.unweighted_model_e import UnweightedGraphE
from models.unweighted_model_v import UnweightedGraphV
from models.unweighted_model_ev import UnweightedGraphEV
from algorithms.algorithm import Algorithm
from algorithms.one_path import AlgorithmOnePath
from algorithms.two_path import AlgorithmTwoPath
from runner import Runner

def parse_args():
```

```
    parser = argparse.ArgumentParser(description = 'Run the experiments: interaction between changing model and the algorithm.')
    parser.add_argument('--alg', dest = 'alg', type = str, required = True, help = 'Algorithm used for testing , one of: one (one_path), two (two_path)')
    parser.add_argument('--n', dest = 'n', type = int, default = 1000, help = 'Number of vertices in the graph ')
    parser.add_argument('--m', dest = 'm', type = int, default = 5000, help = 'Number of edges in the graph')
    parser.add_argument('--c0', dest = 'c0', type = float, required = True, help = 'Constant c0 used to define  constant R (used for phase_length and ball growth step duration)')
    parser.add_argument('--change', dest = 'change_rate', type = int, default = 1, help = 'Number of changes that graph makes at once (default = 1)')
    parser.add_argument('--probe', dest = 'probe_rate', type = int, default = 1, help = 'Number of probes that  the algorithm is allowed to make at once (default = 1)')
    parser.add_argument('--iterations', dest = 'iterations', type = int, default = 10000, help = 'Number of iterations performed')
    parser.add_argument('--model', dest = 'model', type = str, default = "", help = "Configuration of the model ('e' and 'v' include edge and vertex removals, respectively)")
    parser.add_argument('--dataset', dest = 'dataset', type = str, default = "", help = "Dataset for experiment ('' for a random graph, or 'contact', 'wikipedia')")
    parser.add_argument('--rand_seed', dest = 'rand_seed', type = int, default = 0, help = 'Random seed used for reproducibility (default = 0)')
    parser.add_argument('--visualization', dest = 'visualization_step', type = int, default = -1, help = ' every <visualization_step> iterations, prints a character indicating the validity of answer provided by the algorithm (default = -1 (not active))')
    return parser.parse_args()

args = parse_args()

def print_alg_info(algorithm: Algorithm):
    print("--- {} algorithm ---".format(algorithm.name))
    print('R = {}, phase length = {}'.format(algorithm.R, algorithm.phase_length))

n = args.n
m = args.m
iterations = args.iterations
initialize_graph = True
use_dataset = False
dataset = None
if "contact" in args.dataset:
    use_dataset = True
    with open(os.path.join("datasets", "contact", "dataset.pkl"), "rb") as fin:
        dataset = pickle.load(fin)
        n = 789
        m = 0
    iterations = len(dataset["edges"])
    initialize_graph = False
elif "wikipedia" in args.dataset:
    use_dataset = True
    print("Loading dataset...")
    with open(os.path.join("datasets", "wikipedia", "dataset.pkl"), "rb") as fin:
        dataset = pickle.load(fin)
        n = 100312
        m = 0
    print("Dataset loaded.")
    iterations = len(dataset["edges"])
    initialize_graph = False
elif args.dataset != "":
    print("Wrong dataset name! Use 'contact' or 'wikipedia'.")

if args.alg.startswith("one"):
    algorithm = AlgorithmOnePath(args.c0, n)
    print_alg_info(algorithm)
elif args.alg.startswith("two"):
    algorithm = AlgorithmTwoPath(args.c0, n)
    print_alg_info(algorithm)
```

```
else:
    print("Wrong algorithm name! Use 'one' for one-path algorithm, or 'two' for two-path algorithm.")

if "e" in args.model and "v" in args.model:
    graph = UnweightedGraphEV(args.rand_seed, n, m, initialize_graph)
elif "e" in args.model:
    graph = UnweightedGraphE(args.rand_seed, n, m, initialize_graph)
elif "v" in args.model:
    graph = UnweightedGraphV(args.rand_seed, n, m, initialize_graph)
else:
    graph = UnweightedGraph(args.rand_seed, n, m, initialize_graph)

runner = Runner(args.probe_rate, args.change_rate, algorithm, graph, use_dataset, dataset)
runner.run(iterations, args.visualization_step)

if not use_dataset:
    print()
    print("at the end of execution, m = {}, n = {}".format(graph.m, graph.n))

print()
print("Correct answers: {}/{} ({}%)".format(runner.get_correct_answers(), runner.get_total_iterations(), round
    ↪ (100*runner.get_correct_answers()/runner.get_total_iterations(), 2)))
print("Correct answers (without first phase): {}/{} ({}%)".format(runner.get_correct_answers_after_1st_phase()
    ↪ , runner.get_total_iterations() - algorithm.phase_length, round(100*runner.
    ↪ get_correct_answers_after_1st_phase()/(runner.get_total_iterations() - algorithm.phase_length), 2)))
```

Listing 8: Script running a single experiment