Algorithms on Evolving Graphs

Sangram Kapre

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

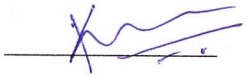
The Degree of Master of Technology

Department of Computer Science and Engineering

July 2013

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.



(Signature)

(Sangram Kapre)

CS11M07

(Roll No.)

Approval Sheet

This Thesis entitled Algorithms on Evolving Graphs by Sangram Kapre is approved for the degree of Master of Technology from IIT Hyderabad

M.V. Panchu 05/07/13

(CSE) Examiner

A Shudeb Sinha 05/07/13

(EE) Examiner

5/7/2013

(Dr. Subramanyam Kalyansundaram) Adviser
Dept. of Computer Science and Engineering
IITH

C. Krishna Mohan 5/7/2013
chairman

Acknowledgements

The final success and outcome of this thesis owes great many thanks to great many people who guided and assisted me in completing this thesis. I am extremely fortunate to have got all this during the course of the thesis.

I would like to express my deepest gratitude to my supervisor **Dr. Subrahmanyam Kalyanasundaram**, who has attitude and a substance of genius. I consider myself extremely lucky to get a guide as supportive as he was. I will always admire the freedom he gave to me to work only on the topics I was interested in. Whenever I approached him with any doubt, he always made sure that the doubt has been cleared. Without his guidance and persistent help the completion of this thesis would not have been possible.

My sincere thanks to the **entire faculty of the CSE department** who inspired all the students towards computer science. I learned the computer science the way it was supposed to be learned, under their guidance. The lectures of **Dr. Naveen Sivadasan** always inspired me to learn new concepts and their applications. His teaching of the subjects 'Advanced Data Structures and Algorithms' and 'Topics in Theoretical Computer Science' had a long lasting effect.

I would not forget to remember my colleagues, who always assisted me in successful completion of the coursework and thesis. Without their cooperation this thesis would not have been possible. Timely completion of this thesis owes many thanks to them. I also take this opportunity to thank all the people at Indian Institute of Technology, Hyderabad, for their help and encouragement. I also thank my family members and friends for their unceasing support.

Dedication

- to my idol *Pappa*, love *Aai*, and inspiration *Sagar* and *Vidya*.

Abstract

Today's applications process large scale graphs which are evolving in nature. We study new computational and data model to study such graphs. In this framework, the algorithms are unaware of the changes happening in the evolving graphs. The algorithms are restricted to probe only limited portion of graph data and are expected to produce a solution close to the optimal one and that too at each time step. This frameworks assumes no constraints on resources like memory and computation time. The limited resource for such algorithms is the limited portion of graph that is allowed to probe (e.g. the number of queries an algorithm can make in order to learn about the graph). We apply this framework to two classical graph theory problems: Shortest Path problem and Maximum Flow problem. We study the way algorithm behaves under evolving model and how does the evolving nature of the graph affects the solution given by the algorithm.

Contents

Chapter 1

Introduction

Graphs are used everywhere today. Graphs are the best ways to represent and process today's social networks (also known as *Social Graphs*). These social graphs depict personal relations of internet users. It has been referred to as *the global mapping of everybody and how they are related.* Graphs are also used to model communication networks (e.g. Internet, telephone networks). Other applications of graphs include hyper-linked induced web graphs where the directed links between pages of World Wide Web are described. Web graphs are directed graphs in which vertices represents the pages of WWW and a directed edge connects page X to page Y if there exists a hyperlink on page X, referring to page Y. All these and other graphs (e.g. recommendation networks) are somewhat similar in two aspects: these graphs are mostly decentralized and are evolving (they keep changing gradually) and these graphs are often large in scale. These two characteristics make computation on such graphs very difficult.

In modern data processing era, the classical computational paradigm in which input is fixed, and algorithm terminates after processing the input, is not adequate to capture the properties of dynamic data. So several computational and data models have been proposed such as data stream models, dynamic models, property testing models, etc. The data stream models take data stream as input and it is expected to output an approximate solution forcing constraints on the memory required to store the data. In dynamic model, the algorithm is aware of the changes happening in the data and computation in such models are expected to be faster than the naive recomputation. In property testing models, algorithm can access a limited portion of data and then produces the output. But today's challenges posed by large scaled and evolving graphs are not being captured by these models. For example, the algorithms in dynamic models and data streaming models must be aware of the changes happening in the data in order to produce an output, thus these models cannot capture the characteristics of most of the graphs being used today (e.g. In web graph, the algorithm is not expected to be notified every time a new hyperlink is added from one page to the other).

1.1 Related Work

Graphs were and are very popular to represent data with relationships. Graphs are even used to effectively represent dynamic data. Lot of work has been done in algorithms to study models which

deals with dynamic data represented as graphs. Unfortunately, the models that were proposed before [1] do not capture the three important aspects: graphs evolves slowly and gradually, gaining information about graph by limited probing, and the observation of the system for infinite (very long) amount of time. Following are the models that are most related to those explained later.

1. **Evolving graph model:** [2, 1, 3] This model was introduced in [1] which in turn followed the framework for dynamic data model proposed in [2]. In this work, unweighed graphs with edge addition and deletion, and weighted graphs with ordinal model were considered. The problems attempted were path-connectivity and minimum spanning tree in dynamic graphs which evolves as per model described in these papers.

2. **Dynamic graph model:** [4] In this model, the goal is to keep track of changes in graphs over time and answering the queries efficiently. But whenever a graph undergo any change, if is notified to the algorithm. But the model we consider involves algorithms that are unaware of any changes happening in the graph. Also the expensive resource in our setting is number of queries but there are no additional constraints on the amount of memory used by the algorithm or the time taken by the algorithm to finish.

3. **Data streams:** In this model, the sequence of events (e.g. addition or deletion of edges) is the input to the algorithm and an approximate solution is expected to be maintained. The algorithm has limited computational resources (typically, the amount of memory that algorithm can use) and it has to maintain an approximate solution while observing an entire sequence of changes. However, in out setting, the limited resource is the number of data probes the algorithm make in order to gain information about evolving graph. We do not really consider the constraints of space and time complexity.

4. **Property testing:** [5, 6] In this model, the goal is to find whether a graph satisfies particular property. The limited resources here is the number of queries that algorithm makes. But this model is static as opposed to evolving graph model which captures dynamic data which changes gradually. Also, the problems considered in evolving graph model need not to be a decision problem (e.g. Maximum Flow problem) and there are no restrictions placed on the input data.

5. **Parametric Optimization and Kinetic Problems:** [7] The continuous functions define the weights of edges in such models. The solution is expected to found out as the real parameter to these functions (often referred to as *time*) varies. In such problems, the initial value of is known to an algorithm. The algorithms are on-line meaning an optimal solution is expected at each time step. However, this model also is inadequate to capture the characteristic that algorithms are unaware of the changes happening in the data as algorithms are needed to be notified about the changes.

1.2 Our Contribution

The only model that considered the constraints on algorithms such as algorithm being unaware of changes happened in the graph and algorithm, without been able to traverse the entire graph at one time, has to produce a solution at each time step is given in [1]. The paper considers two

types of graphs: weighted and unweighted. But for unweighted graphs, only ordinal model has been discussed where only order between the edges changes, but not the actual weights. Also, for unweighted graphs, only Path Connectivity problem is considered.

In the thesis, we consider the shortest path on unweighted evolving graphs. Also, we propose two different models for weighted evolving graphs in which actual weights of edges are allowed to change in some predefined manner. We give theoretical bound for shortest path problem on unweighted evolving graphs. We also discuss two more problems: shortest path problem and maximum flow problem on weighted evolving graphs.

Chapter 2

Model

2.1 General Framework

The general framework defined in [2] for algorithms on dynamic data is followed for evolving graphs. The model for evolving is as follows. The time is assumed to proceed in discrete steps. Each time step is a positive integer. G_t denotes the graph representation of data at time t. Graph can be weighted or unweighted depending on the problem under consideration. The changes in graph occur gradually (*i.e.* on step-by-step basis). A small *random* change in G_t at time t leads to a graph G_{t+1} at time step t+1. The reason for assuming changes to be random is if changes are adversarial, it is easy to show lower bounds. Thus assumption of random changes makes model more interesting.

Algorithm cannot traverse the whole graph at a time. But at each time step, it is allowed to probe a small portion of a graph G_t. A probe can be checking a particular node, for example. Also, algorithm must output a solution for the problem at each time step. The solution is expected to be *close* to the correct solution for the graph G_t. Throughout this thesis, no constraints are imposed on the amount of memory the algorithm maintains or the amount of time the algorithm takes to produce solution, although all of the algorithms that are presented are quite efficient with respect to those factors.

2.2 Change, Probe and Quality Measure

We have seen the general framework of an evolving model for a graph. But when we consider a specific problem, we need to fix these three aspects: the way the graph evolves, the way the algorithm probes a graph under consideration, and the way the quality of the solution is measured. We describe these aspect below in more detail.

(i) Change: For simplicity, we assume that the number of nodes, the number of edges and also the labels of the graph remain unchanged. For a graph G_t, let V be the set of vertices and E_t is the set of edges at time t. The reason for denoting edge set with subscript t is that it keeps changing at each time step, but the vertex set V remain fixed. Let n be the number of vertices and m be the number of edges in the graph. Now we consider how the edge set E_t changes for both weighted and unweighted graphs.

For unweighted graphs, at each time step, a random swap occurs in order to change edge set

from E_t to E_{t+1}. At each time step, a random edge is removed from the graph and a new edge is added between a random pair of nodes. i.e. let $(u, v) \in E_t$ be a uniformly chosen edge and let $u, v \in V$ be uniformly chosen nodes such that $(u', v') \notin E_t \backslash (u, v)$. Thus (u', v') is a randomly selected pair of nodes which are not connected. Then the edge set at time $t + 1$ is given as: $E_{t+1} = (E_t \backslash \{u, v\}) \cup \{(u, v)\}$.

For weighted graphs, we consider two different models, one for shortest path problem and one for maximum flow problem. For shortest path problem, let w_i^t be the weight for an edge ei at time t. Then weight update for edge e_i is as: $w_i^{t+1} = w_i^t * \alpha_i$ where α_i is the random variable generated using normal distribution with mean 0 and variance v. All the edges undergo a change as described above at each time step. For maximum flow problem, at each time step, one random edge $(u, v) \in E_t$ is chosen uniformly at random and it's weight is changed by 1 (either increased or decreased).

(ii) Probe: For shortest path problems on both weighted and unweighted, a node probe is performed. i.e. we query a node $u \in V$ and learn about the neighbors of u . For the problem of finding maximum flow of a graph, we allow the algorithm to find one shortest path from the *source* node to *sink* node before next update happens.

(iii) Quality Measure: We follow the similar benchmarks used in [1] to measure the quality of solution. If a problem is an optimization problem, one can be interested in the difference between the optimal value and the value obtained by the algorithm. For non-optimization problems (i.e. decision problems), the probability of solution given by the algorithm being a valid feasible solution is considered to be a quality measure.

Chapter 3

Shortest Path on unweighted evolving graphs

In this section we consider the basic shortest path problem on unweighted and undirected graph.

3.1 Problem Statement

Given two fixed nodes $S, T \in V$, designated as *source* and *sink* respectively, maintain a shortest path between S and T, assuming one exists.

Here we consider unweighted and undirected graph which keeps on evolving gradually. The model for evolving graph is as explained in chapter 3. As graph under consideration is unweighted, the way the graph evolves per time step is explained below.

Consider G_t to be the graph at time t, with set of vertices as V and set of edges as E_t. We get the graph G_{t+1} at time $t+1$ as follows :

1. $V_{t+1} = V_t$: Set of nodes remain unchanged.
2. E_t is changed to E_{t+1} as : select a pair (u, v) where $u, v \in V$, uniformly at random. If (u, v) is already an edge in E_t then remove it, else add edge (u, v) to the graph.

Now for an evolving graph like this, the problem becomes to output a shortest path from source to destination at every time step with high probability that the path given is a valid shortest path. We will see the algorithm for this problem below and then we will discuss the guarantee that the solution given by the algorithm is correct. The algorithm is simple *Breadth First Search* applied on unweighted graph from source S till we encounter destination T.

3.2 Algorithm

1. Push a node S into Queue.
2. STEP
   (a) remove one node from Queue (*node probe*)

(b) If it is T (destination node), STOP.

(c) Visit all of it's neighbors.

(d) The graph evolves as described before.

3. Repeat the STEP again.

3.3 Discussion

As pointed in [1], as $t \to \infty$, the distribution of the graph G_t approaches the distribution of a *random graph* $\mathcal{G}(n, m)$, a graph chosen uniformly at random from all graphs with n nodes and m edges. Thus if $m = o(n \ln n)$, then there is a non-trivial probability that there is no path between S and T. Thus the number of edges should be $m = \Omega(n \ln n)$. Also we will consider only those graphs for which $m = O(n \ln n)$. This makes the interesting range for number of edges as $m = \Theta(n \ln n)$. Also for simplicity of exposition, we will assume that our initial graph G_0 is a *random graph G(n,m)* and also, at time $t = 1$, our algorithm knows a *valid* shortest path between S and T.

We will consider the algorithm that works in phases. It finds the shortest path between S and T iteratively and at any moment, the output given by the algorithm is the output that was found in the last finished phase of finding shortest path between source and destination.

In every phase, we execute the shortest path finding algorithm explained before. The algorithm grows a ball B_s around S as follows. Initially B_s=$\{S\}$ and S is marked *unvisited*. In every step, the algorithm picks up an unvisited node in B_s closest to S, marks it as visited, and adds all of its neighbours to B_s using a node probe. Also any new node added to the set will be marked as unvisited. As suggested in [1] we will also terminate the algorithm after $R = \lceil \sqrt{c_0 n / \ln n} \rceil$ steps, where c_0 is a constant. Note that after R steps, the ball B_s contains R visited nodes.

3.4 Valid shortest path

Now we give the theoretical bound on the probability of the shortest path given by the algorithm discussed being valid. Note that we are considering the graphs for which the number of edges $m = \Theta(n \ln n)$.

3.4.1 Lemma 1

The degree of each node of a graph is $\Theta(n \ln n)$ during the entire execution of an algorithm.

3.4.2 Proof

Let $X_1, X_2, .., X_k$ be indicator random variables, where $k = n(n-1)/2$, for each (u, v) pair, $u, v \in V$. As the total number of edges, m, is fixed,

$$X_1 + X_2 + .. + X_k = m.$$

Now $Pr.(X_i = 1) = m/k$, for all $i = 1, 2, .., k$, because G_t is from $\mathcal{G}(n, m)$ is a graph chosen uniformly at random from a family of graphs with n nodes and m edges.

Now consider a node $v \in V$. The maximum number of edges incident on v is $n-1$. Thus *degree* of node v can be given as :

$deg(v) = X_{e_1} + X_{e_2} + .. + X_{e_{n-1}}$, where $e_1, e_2, .., e_{n-1}$ are from set $\{1, 2, .., k\}$.

Thus $deg(v)$ is also a random variable.

$E(deg(v)) = E(X_{e_1} + X_{e_2} + .. + X_{e_{n-1}}) = E(Xe_1) + E(Xe_1) + .. + E(X_{e_{n-1}})$.

But, $E(X_i) = m/k$, for $i = 1, 2, .., k$

So, $E(deg(v)) = (n-1)m/k = 2m/n$.

First we will prove the lower bound on degree of all the vertices of a graph. As $m = \Omega(n \ln n)$, $m = c_1.n \ln n$, for some constant c_1. By chernoff bound,

$Pr.(deg(v) < (1-\delta)(2m/n)) \leq e^{\frac{-\delta^2(2m/n)}{2}}$, where $0 < \delta < 1$

$= e^{\frac{-\delta^2 m}{n}}$

$= e^{-\delta^2 c_1 \ln n}$

$= n^{-\delta^2.c_1}$

Chose δ such that $\delta^2.c_1 \geq 5$.

This implies that for some node $v \in V$, the probability that degree of v being less than $c_1 \ln n$ is at most n^{-5}. Thus the probability that there exists at least one node in V for which degree is less than $c_1 \ln n$ is at most n^{-5} (by union bound). So, at a given time point of algorithm every node will have degree at least $c_1 \ln n$ with probability at least $1 - n^{-4}$, for a sufficiently large constant c_1. By applying a union bound, we obtain that with probability at least $1 - n^{-3}$ each node will have degree at least $c_1 \ln n$ during the entire algorithm execution. Similarly, by considering $m = O(n \ln n)$, we can show that the degree of each node, through out the execution of an algorithm, is at most $c_2 \ln n$ for large constant c_2.

Hence, with high probability, the degree of each node of a graph is $\Theta(\ln n)$, for entire execution of an algorithm.

3.4.3 Lemma 2

The path given by the algorithm described above, at the end of a single phase, is a valid shortest path between S ans T, with high probability.

3.4.4 Proof

We consider the situations under which the algorithm fails. Note that there are three such cases.

1. There is no $S - T$ path in the graph
2. Algorithm fails because of edge deletion
3. Algorithm fails because of edge addtion

Note that the probability of first case for the graph we are considering is $O((n \ln n)^{-1/2})$ [1].

The second case is when one or more edges get deleted from the shortest path found while algorithm is running. The probability that the path computed at the end of the execution of the algorithm is still valid is at least the probability that no edge from the path is removed till algorithm is running. Algorithm runs for at most R time steps. Also the length of the shortest path is at most

$O(\ln n)$. Thus the probability that path becomes invalid because of removal of an edge from it at time step t is at least one of the $O(R \ln n) = O(\sqrt{n \ln n})$ bad events of the form *edge e is removed at time x* occurs. The probability of such edge removal is $1/m$, and hence by the union bound, the probability of path becoming invalid because of an edge removal event is $O((n \ln n)^{-1/2})$.

For the third case where shortest path becomes invalid because of edge addition, consider the following. Let B_s denotes the ball around s and contains all the vertices that were visited while algorithm was running. Thus t is on the circumference of that circle. Similarly imagine a ball of same size centered around vertex t. Again note that s must be present on the circumference of ball B_t. Let d denotes the length of the shortest path between s and t. Thus d is the radius of both B_s and B_t. Any new path created because of addition of edge (u, v) will not be a shorter than the shortest path given by the algorithm if $u, v \notin B_s \cup B_t$. As both u and v are at a distance more than d from both s and t the path length will always remain more than d. Now we will analyse the probability of the shortest path given by the algorithm becoming invalid because of newly added edge creates even a shorter path. Clearly it is bounded by an event that at lest one edge (x, y), where at least one of the vertices x and $y \in B_s \cup B_t$, gets added in one or more of the R steps for which algorithm runs. We call this probability P.

The probability of such event at time step t is $\frac{|B_s|*|B_t|}{n^2-|E|}$ and there are R such steps. Thus, by union bound, the probability is bounded by $\frac{R*|B_s|*|B_t|}{n^2-|E|}$.

$R = \Theta(\sqrt{n/\ln n})$. We need to analyse the maximum size of sets B_s and B_t. For graphs under consideration, $m = O(n \ln n)$. By Chernoff bound, at a given point of time of the algorithm every node will have degree at most $c \ln n$ with probability at least $1 - n^{-4}$ for a sufficiently large constant c, as proved in Lemma 3.4.1. By applying a union bound, we obtain that with probability at least $1 - n^{-3}$ each node will have a degree at most $c \ln n$ during the entire execution of the algorithm. Thus the degree of each node is at most $O(\ln n)$ with high probability.

As the algorithm stops after $R = \Theta(\sqrt{n/\ln n})$ steps, and since degree of each node is $O(\ln n)$ with high probability, we have that with high probability,

$$|B_s| = O(R \ln n) = O(\sqrt{n \ln n}).$$

Similar analysis follows for a ball B_t giving $|B_t| = O(R \ln n) = O(\sqrt{n \ln n})$.

Thus with probability at least $1- O(\frac{\sqrt{n \ln n}}{n-\ln n})$ the shortest path remains valid even after graph evolves as edges get added to the graph.

Hence combining all the cases which may lead to an invalid shortest path, the algorithm outputs a valid shortest path with at least $1- O(\frac{\sqrt{n \ln n}}{n-\ln n})$ probability.

3.4.5 Theorem

The path given by the algorithm described above, at every time step t, is a valid shortest path between S ans T, with high probability.

3.4.6 Proof

The algorithm described works in phases and each phase last for at most R time steps. Thus ith phase is from $Ri + 1$ to $R(i + 1)$. At any time step $t \in [Ri + 1, R(i + 1)]$, the output produced at the end of the last phase (i.e. at time Ri) is given as output. Now, by Lemma 2 given in 3.4.3,

the output produced by an algorithm is valid with high probability. Thus applying the exact same analysis, we can deduce that the probability of the shortest path given by an algorithm at each time step t being valid is at least $1- O(\frac{\sqrt{n \ln n}}{n - \ln n})$.

3.5 Experimental results and conclusion

Consider following parameters.

1. n : number of nodes in a graph
2. α : number of edges updated (added / deleted) per step when graph evolves
3. p : probability for every (u, v) pair that (u, v) is an edge
4. i : number of iterations for which algorithm runs (this was 1000 for all the experiments)

Now this problem is non-optimization problem. So as we discussed, we consider the number of times the solution given by the algorithm was *not valid* out of the total number of times the algorithm was executed.

Following figures show the number of mismatches as α varies. Here mismatch means that the shortest path given by the algorithm and actual shortest path at the end of iteration are different.

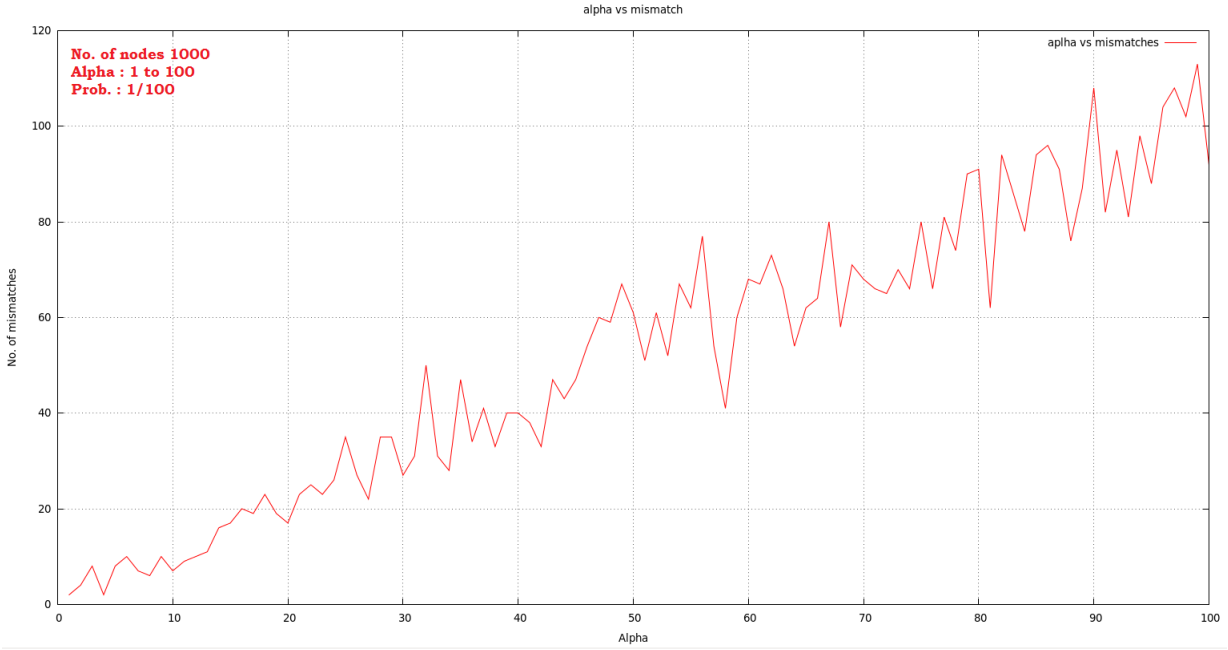


Figure 3.1: $p = 1/100$

The graph 3.1 shows alpha vs. total number of iterations for which algorithm failed to find a shortest path out of 1000 iterations. As probability for each (u, v) pair to be an edge was $1/100$, the expected number of edges in this graph were roughly around 5000. As this was the densest amongst all the graphs, the number of mismatches were low (less than 120) even for $\alpha = 100$ (i.e. when 100 updates were made to the graph at each time step). This was expected because more number of

edges means there is a good chance of existence of $S-T$ path. Also more number of edges means the length of the shortest path was expected to be less, so the addition of an edge to get even shorter path or removal of an edge from the path found by an algorithm had comparatively less chance that the graphs with fewer edges.

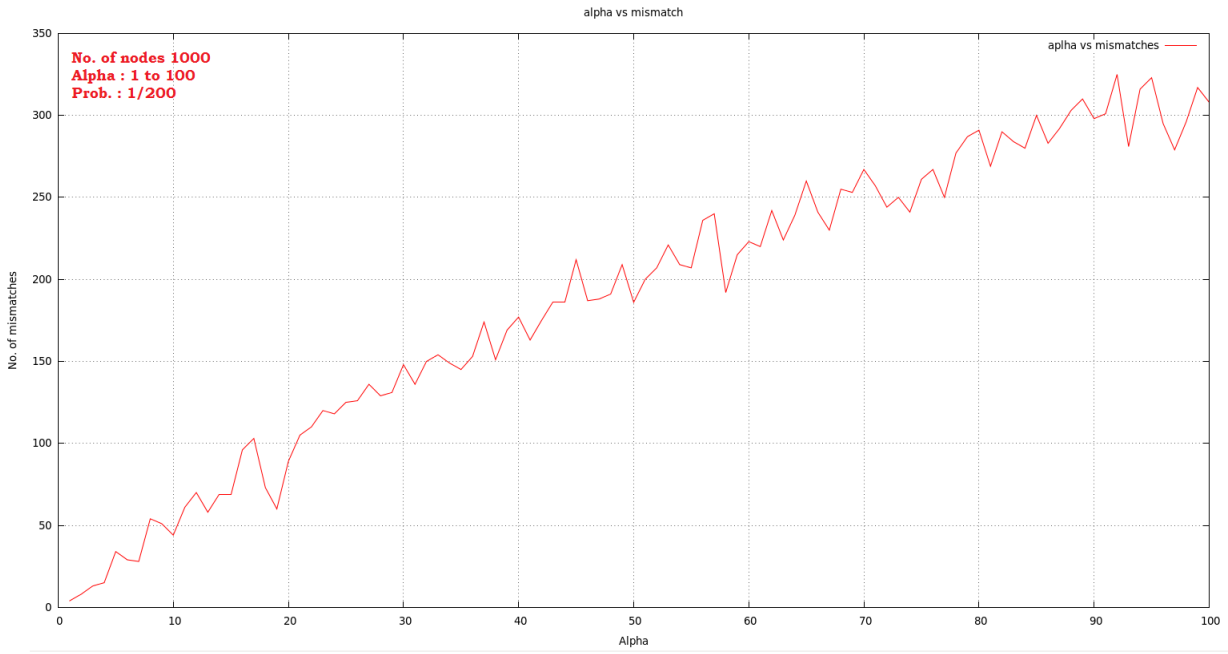


Figure 3.2: $p = 1/200$

The figure 3.2 shows alpha vs. total number of mismatches for a graph where the probability of each (u, v) pair to be an edge was $1/200$. This graph was less dense than the previous one and thus the maximum number of mismatches were roughly around 325 (as expected) when $\alpha = 100$ (for maximum value of α in the graph).

The figure 3.3 shown above was for the graph for which the probability for a pair (u, v) to be an edge was $1/500$. This also was the sparsest graph of all. Thus the number of mismatches were very high (close to 700 out of 1000 iterations). As there were comparatively fewer number of edges, the length of the shortest path was expected to be long. Hence the chance of that path becoming invalid was relatively high because addition of new edges leading to even shorter path or deletion of edges from that path itself. Thus for graphs with fewer edges, the algorithm fails most of the time.

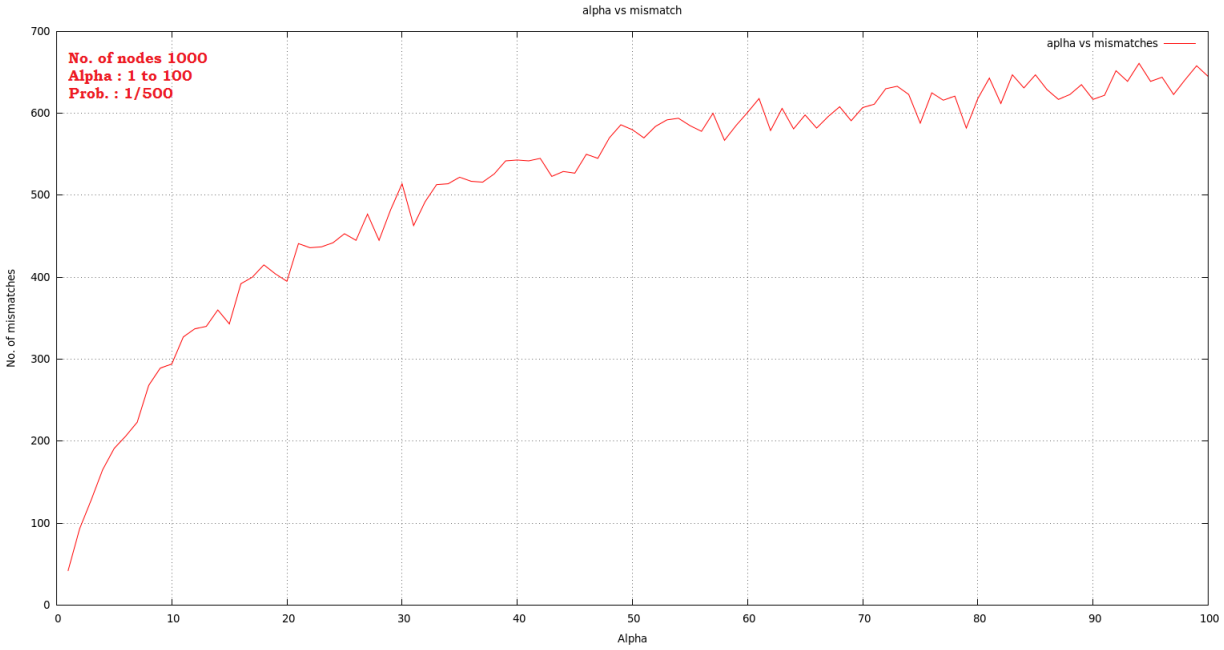


Figure 3.3: $p = 1/500$

Chapter 4

Shortest Path on weighted evolving graphs

This section considers shortest path problem on weighted, undirected evolving graphs.

4.1 Problem Statement

In a weighted undirected graph, from a designated source S to destination T find the shortest path from S to T (the path with the minimum weight between S and T), if one exists.

Again the graph under consideration is assumed to be evolving gradually (i.e. step-by-step). The model for such graph is the one explained for weighted graphs in the chapter 2.

We now will see the model for this problem in more details. Let $G_t(V, E_t)$ be the weighted undirected graph at time t. The graph evolves from G_t at time t to G_{t+1} at time $t+1$ as follows.

1. $V_{t+1} = V_t$: Set of nodes remain unchanged.

2. E_t is changed to E_{t+1} as shown below:
   All the edges in E_t changes their weights with normal distribution having mean 0 and variance v as follows.
   For each e_i let w_i^t be the weight at time t.
   The weight update for the edge e_i is as :
   $$w_i^{t+1} = w_i^t * \alpha_i$$
   where α_i is the random variable generated using normal distribution with mean 0 and variance v.

   The algorithm we consider is simple Dijkstra's Algorithm on weighted undirected graphs.

4.2 Algorithm

1. Push a node S into Queue.

2. STEP : remove one node from Queue (*node probe*).

3. If it is T (destination node), STOP.

4. Visit all of it's neighbors.

5. The graph evolves as described before.

6. Repeat the STEP again.

4.3 Experimental results and conclusion

Now consider following parameters for this problem.

1. n : number of nodes in an undirected graph (we consider graph with 100 nodes)

2. v : the variance of a normal distribution with which all α_i are generated (we vary this parameter to see how it affects the solution given by the algorithm)

3. $max - weight$: the maximum initial weight allowed for each edge

Now this is an optimization problem. The solution must be an optimal shortest path (i.e. the path with minimum weight). So there must be some way to compare the actual solution with the solution given by the algorithm.

We plot $M = \max\{(evolving - sp - weight/static - sp - weight)_i\}$ for all $i = 1, 2, .., n$.

Following figures show the value of M against *variance* with which the random variables α_i were selected. The graphs considered for shortest problems were complete graphs. Each edge had a weight chosen uniformly at random from 1 to maximum weight allowed for that experiment. All graphs had 100 nodes.

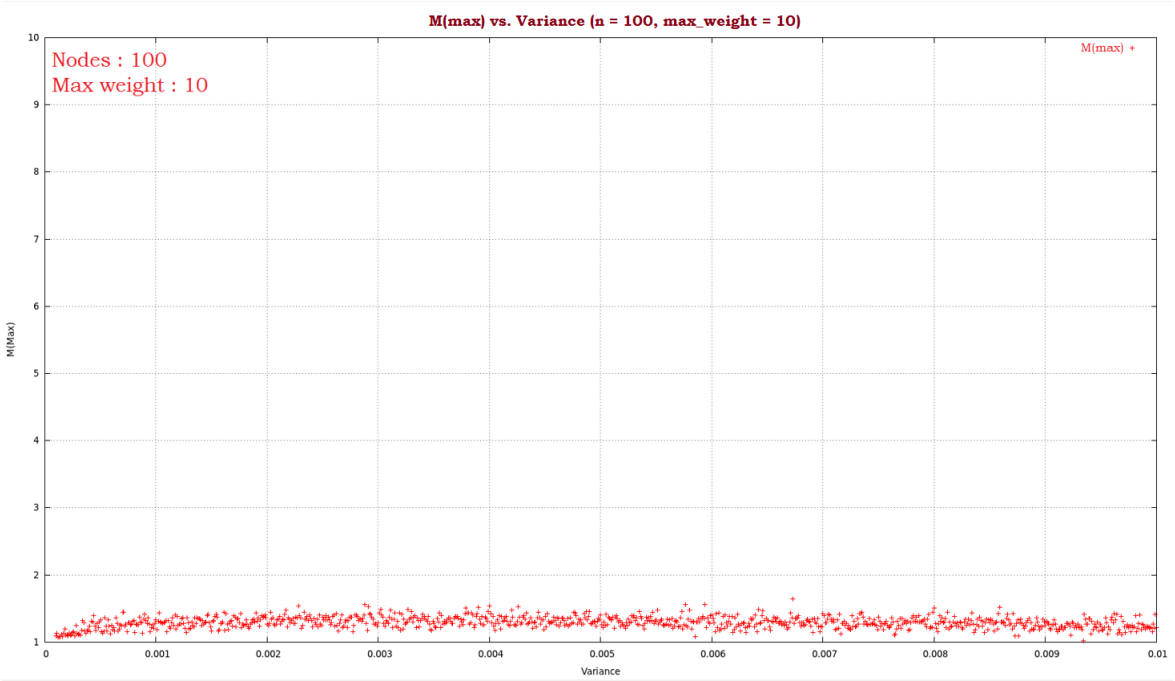


Figure 4.1: $max - weight = 10$

The figure 4.1 shows the variations in the parameter M (which is described above) against the variance v with which we are generating αs to update the weights of all the edges of the graph. For this case, maximum weight allowed was only 10. The variance was changed from 0.0001 to 0.01, each time increasing by 0.00001 only. As it can be seen from the graph, when maximum weight was 10, the value of M remained very low (much less than 2, even for maximum variance). This shows that the model explained work well when the maximum initial weight allowed for each edge in an evolving graph is less.

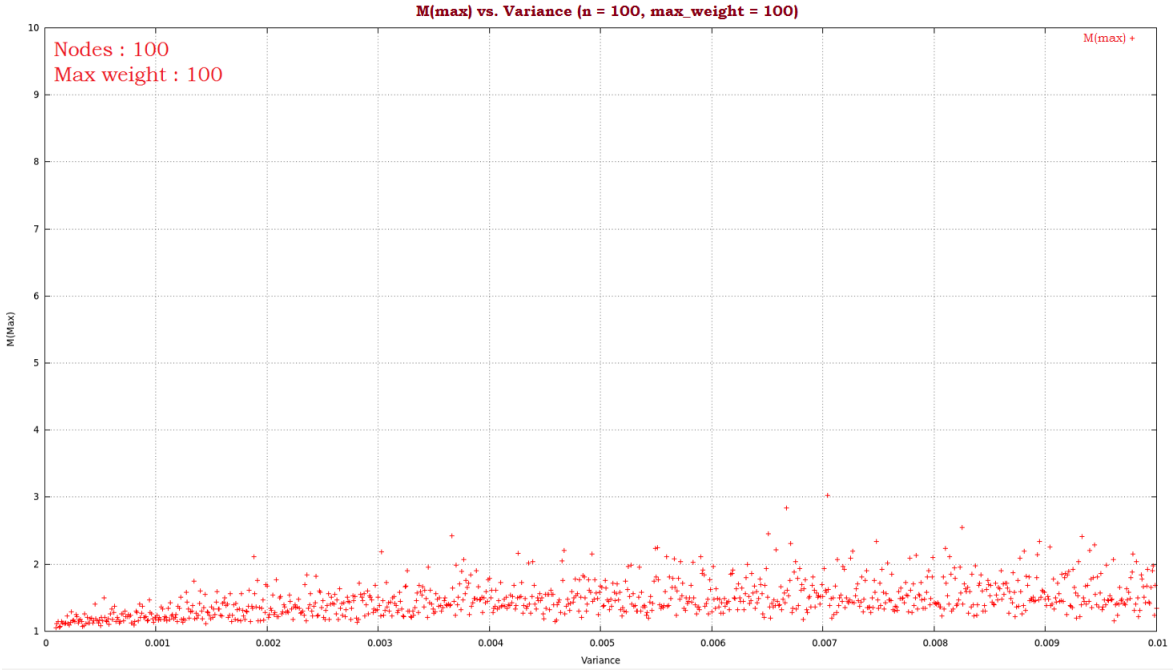


Figure 4.2: $max - weight = 100$

The graph in 4.2 is the plot of M against variance v for a graph where maximum weight allowed was 100. Again, the initial weights for all the edges were chosen uniformly at random from 1 to 100, for every edge. Variance was changed for each iteration as explained before. Here, as the maximum weight allowed was more than the previous one, the maximum value of M was also more (slightly more than 3 for one of the tests).

The figure 4.3 is again the plot of M vs. v, but here the maximum weight allowed for each edge in a graph was 1000. This the initial weight for each edge was chosen uniformly at random from 1 to 1000. As the initial maximum weight allowed was 1000, the value of parameter M was very high for high variance (close to 10 in some cases). Though it remained under 2 for variance below 0.001.

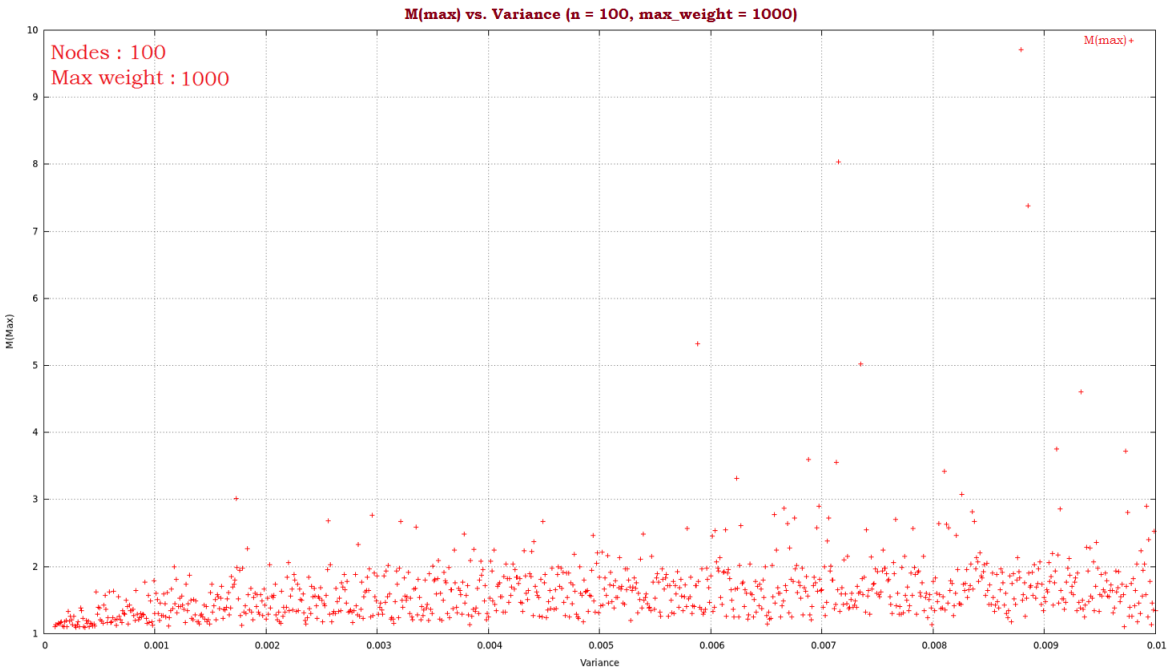


Figure 4.3: $max - weight = 1000$

Chapter 5

Maximum Flow on weighted evolving graphs

In this section we consider the problem of finding maximum flow on weighted evolving graphs. The maximum flow problem is defined below.

5.1 Problem Statement

In an evolving flow network (weighted directed graph), from source S to sink T, find out the maximum amount of flow that flows from S to T. A flow network is defined as $N(V, E)$ with $s, t \in V$ being the source and sink respectively. The *capacity* of an edge c_{uv} or $c(u, v)$ represents the maximum amount of flow that can pass through an edge. A *flow* on edge (u, v), denoted by $f(u, v)$, is subject to following two constraints:

1. $f(u, v) \leq c(u, v)$.

2. conservation of flows : the sum of the flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes.

The *value of flow* $|f| = \sum_{v:(s,v)\in E} f_{sv}$ where s is the source of N represents the amount of flow passing from the source to the sink. The *maximum flow problem* is to maximize $|f|$, that is, to route as much flow as possible from s to t.

Let $G_t(V, E_t)$ be a flow network at time t. Each edge (u, v) has a non-negative *capacity*. For each node, except for designated *source* and *sink*, the amount of flow coming into the node is same as the amount of flow going out of the node. The graph G_{t+1} is obtained from G_t as follows:

1. $V_{t+1} = V_t$ (set of nodes remain unchanged).

2. E_{t+1} is obtained from E_t as: select a pair $(u, v), u, v \in V$ uniformly at random, and either increase the capacity of an edge by 1 (add an edge with capacity with 0 if (u, v) is not an edge already) or decrease the capacity by 1 (no change if (u, v) is not tan edge already) with probability 1/2.

This describes the single update at each time step. Note that there can be multiple updates where this procedure will be repeated for predefined number of times.

The algorithm we discuss is Edmonds-Karp's algorithm for finding out maximum flow. It is specialization of Ford-Fulkerson. We use the version where the shortest path is found from source to sink using *Breadth First Search* in each iteration of the algorithm and some amount of flow is augmented along it.

5.2 Algorithm

As long as there is a shortest path from S to T with non-zero residual capacity:

1. Find such shortest path from source to sink.
2. Augment the maximum flow possible along this path.
3. The graph is evolved for one time step as explained before.

5.3 Experimental results and conclusion

For maximum flow problem, we consider following parameters. Note that all graphs are complete graphs.

1. n : number of nodes in a flow network
2. α : number of edges that goes under weight change as described in the evolving model for max-flow problem
3. p : probability for every (u, v) pair that (u, v) edge has a positive capacity (this decides the number of edges in a flow network, initially)
4. i : number of iterations for each value of α

This also is an optimization problem. However for this problem, we consider the number of times the solution given by the algorithm was incorrect (i.e. different from the actual maximum amount of flow from source to sink)

Here we plot the number of mismatches out of 1000 iterations of algorithm against α. This gives the idea of how maximum flow changes when a certain predefined amount of graph undergoes edge change. The weights for all edges were assigned randomly. Each edge had a weight chosen uniformly at random from 1 to 10. For all the experiments, the number of updates were varied from 1 to 50, and for each α the algorithm was run for 1000 iterations.

The graph in 5.1 shows the number of mismatches out of 1000 iterations for each α ranging from 1 to 50. For this graph, the total number of nodes were 1000 and the probability that each (u, v) pair is an edge was 1/5. For this case, the number of mismatches varied almost linearly along with the value of α and the maximum value of number of mismatches were close to 850 out of 1000 iterations.

The figure 5.2 is also for the number of mismatches vs. α. The graph considered here had 500 nodes and the probability of each (u, v) pair being an edge was 1/2. Thus this graph was very dense

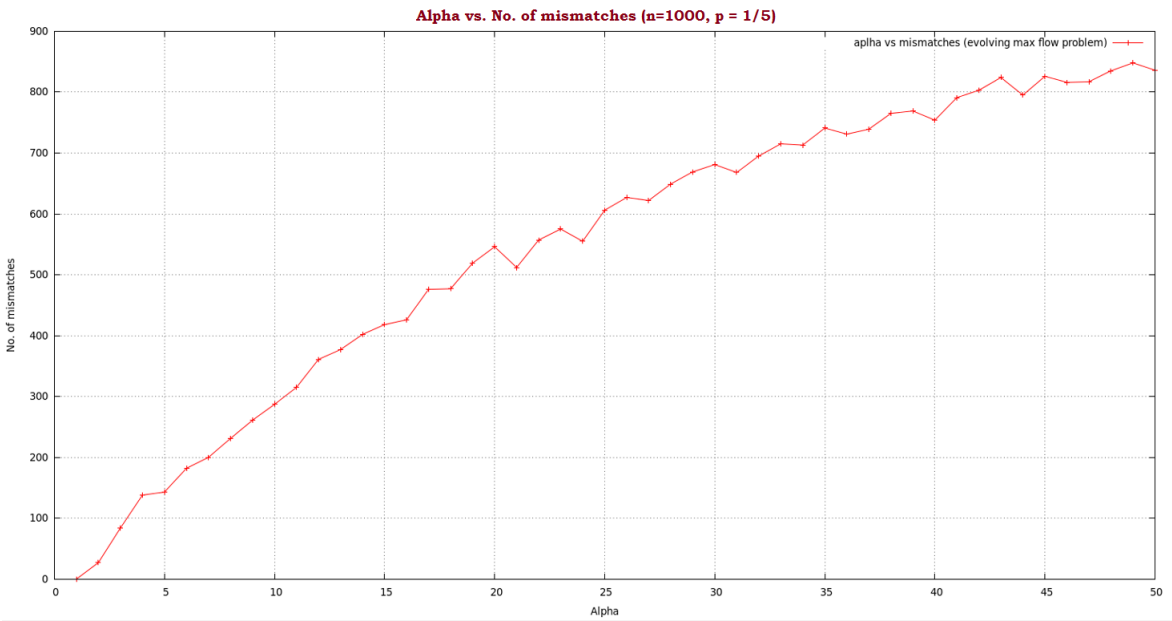


Figure 5.1: $n = 1000, p = 1/5$

graph. The value of number of mismatch increased exponentially against the number of updates that were made at each time step, and it reached very close to 1000 (that is the algorithm gave incorrect value for almost every iteration when α was more than 20). But for a single update per time step, the number of mismatches were still very close to 0, same as in previous case.

The graph shown in 5.3 is very similar to the one in 5.2 but with only one difference. The probability of each (u, v) pair being an edge was $1/3$, so this graph had fewer number of edges. The number of mismatches here also varied exponentially with the number of updates being made to the graph at each time step, but the maximum value reached was less than 1000 for all αs. Similar to both the previous cases, the number of mismatches were very close to 0 for a single update per time step.

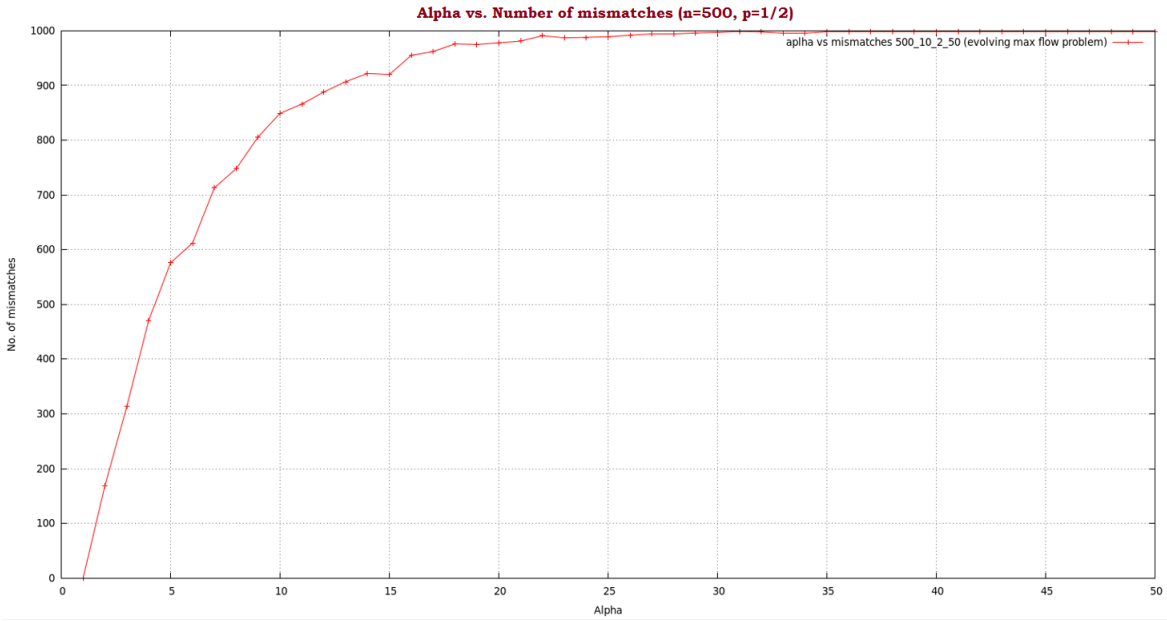


Figure 5.2: $n = 500, p = 1/2$

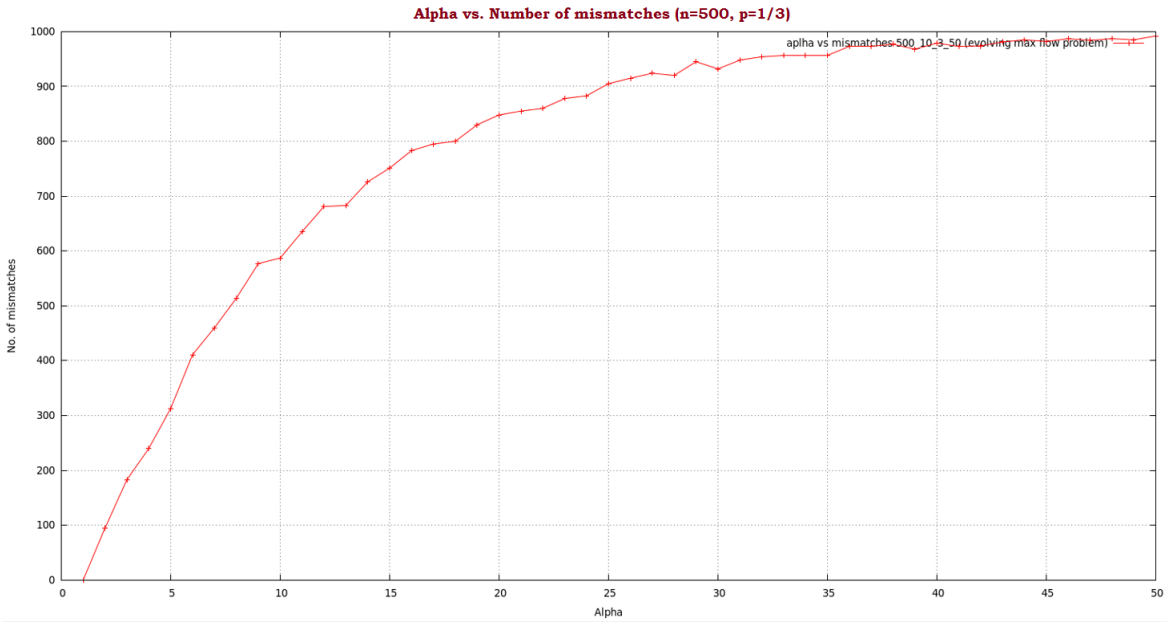


Figure 5.3: $n = 500, p = 1/3$

Chapter 6

Conclusion and Future Work

We considered the evolving graph model to model some of very common real world problems. As we saw, most of the problems today where data is represented as graphs (e.g. social graphs, communication and information networks) have two characteristics in common: those are very large scale graphs, and they always keep on changing gradually. Also it is expected to design frameworks for such problems where the algorithm never gets notified about the changes that are happening in large graphs but still it is expected to produce an outcome that is as close to the actual solution as possible, with only scanning the limited portion of the graph (limited probing). Though this evolution model looks very specific (as we consider restrictions on type of graph we want to process or the way graph evolves), it serves the purpose of showing the possibility of achieving the algorithms for real world graph problems that outputs a solution closer to an optimal one at every time step, while the graph keeps evolving.

We studied the shortest path problem on unweighted graphs and saw that how the number of edges present in the graph affects the guarantee of the path output by an algorithm being the valid shortest path. Also we studied the same problem on weighted evolving graphs where each edge of a graph undergo a small change at every time step. Here we had restrictions on the maximum weight that each edge can have at the start. We found that the variance with which the edges change their weights and the maximum weight an edge (and hence overall expected weights) has an impact on the factor by which a solution given by an algorithm differs from the actual solution.

We also looked at the model for maximum flow problem. Here we studied the effect of the number of updates happening in the graph per time step on the solution given by an algorithm as maximum flow. Here the restriction on the graph was the probability with which each (u, v) pair was an edge. For single update per time step, the number of mismatches were very low. But as the updates per time were increased, the number of times for which algorithm failed was increased too, depending upon how dense the graph was. For a graph with fewer edges, this increase was linear, but for very dense graphs, it showed exponential behavior.

Future work includes the application of this evolving graph model to more graph problems such as maintaining a minimum-weight matching, minimum-flow, etc. It also remains to provide the theoretical basis for the shortest path problem on unweighted evolving graphs for more general cases, such as considering the graphs with no restriction on number of edges as well as the number of updates increased to some fixed value α instead of just single update at each time step. Even

addition and deletion of nodes to graphs can be considered as a way of evolving. It is also worth exploring theoretical bounds on the approximate solution provided by an algorithm for the problems discussed i.e. shortest path problem on weighted graphs and maximum flow problem on evolving graphs.

References

[1] A. Anagnostopoulos, R. Kumar, M. Mahdian, E. Upfal, and F. Vandin. Algorithms on evolving graphs. In Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS '12. ACM, New York, NY, USA, 2012 149–160.

[2] A. Anagnostopoulos, R. Kumar, M. Mahdian, and E. Upfal. Sorting and selection on dynamic data. *Theoretical Computer Science* 412, (2011) 2564 – 2576. Selected Papers from 36th International Colloquium on Automata, Languages and Programming (ICALP 2009).

[3] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. PageRank on an evolving graph. In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '12. ACM, New York, NY, USA, 2012 24–32.

[4] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. Springer, 1998.

[5] D. Ron. Algorithmic and Analysis Techniques in Property Testing.

[6] D. Ron. Property Testing: A Learning Theory Perspective. *Found. Trends Mach. Learn.* 1, (2008) 307–402.

[7] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In JOURNAL OF ALGORITHMS. 1997 747–756.