Problems on Temporal Graphs

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Vishnu Vardhan V M

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2019

Supervisor: Soumen Maity

Certificate

This is to certify that this dissertation entitled Problems on Temporal Graphs towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Vishnu Vardhan V M at Indian Institute of Science Education and Research and The Institute of Mathematical Sciences, Chennai under the supervision of Soumen Maity, Associate Professor, Department of Mathematics, and Saket Saurabh, Professor, Theoretical Computer Science, respectively during the academic year 2018-2019.

Soumen Maity

Saket Saurabh

Committee:

Soumen Maity

Saket Saurabh

Neeldhara Mishra

This thesis is dedicated to my parents, friends and mentors

Declaration

I hereby declare that the matter embodied in the report entitled Problems on Temporal Graphs are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science Education and Research, Pune, under the supervision of Soumen Maity and Saket Saurabh, and the same has not been submitted elsewhere for any other degree.

Vishnu Vardhan V M

Acknowledgments

I would first like to express my deep gratitude to Prof.Soumen Maity. His course in Algorithms was one of the earliest events that motivated me to explore the topic in more detail and pursue this project. He has also been a supportive mentor whose guidance was invaluable in completing this project.
I would like to thank Prof.Saket Saurabh for sharing his extensive knowledge which provided direction to the project, and for his boundless energy which never failed to motivate me. I am indebted to Prof.Neeldhara Mishra for spending her valuable time on the project, and for her helpful suggestions and comments.
Finally, I would like to express my appreciation for my friends who provided me with opportunities of discussion and support.

Abstract

The idea of temporal graphs can be thought of as a recent addition to the extensively researched concept of graph theory. The advent of cheap wireless communication devices and need of efficient communication protocols, in addition to distributed computation networks has motivated progress in this field. However, the temporal analogues to polynomial time problem on static graphs are very often NP-hard. Thus, there seem to a lot of problems which can be parameterized and explored under various constraints. The study of temporal graphs by parameterized algorithms is a field which is still very much in its infancy, and only a few basic properties and problems have been defined in literature. In this document, we will explore the problem of finding temporal paths parameterized by their length, and a few temporal walk-related problems.

Contents

Chapter 1

Introduction

Graphs are mathematical objects which were conceptualized as early as 1900s, and has been instrumental in the development of discrete Mathematics. They are expressed as (V, E) where V denotes a set of vertices, and E denotes the edges that exists between the vertices. Thus, each element of E can be though of as a tuple (u, v) where u and v represents the vertices connected by this edge.

Graphs are often used for expressing the relationship between objects, where vertices of graphs represents the objects and the edges represents a relationship between them. In this way, we can represent various networks, such as transportation networks and social networks using graphs. However, graphs are insufficient to model time varying networks.

The idea of temporal graphs is to overcome this limitation of graphs. Temporal graphs can encode time dependent data into a graph like structure. Temporal Graphs can be defined as $(V, E, \boldsymbol{L})$. Here, V represents a set of vertices and E, a set of edges defined on V. Thus, (V, E) represents a standard graph upon which we add the temporal element in the form of $\boldsymbol{L}$. $\boldsymbol{L}$ is a set of functions from E to a set of timestamps when the edges exist [9]. This new structure is capable of encoding time based information and is a useful tool in analyzing such information. The advent of cheap wireless communication devices and need of efficient communication protocols, in addition to distributed computation networks has motivated progress in this field.

Parameterized algorithms are a class of algorithms which employ one or more *parameters* along with the input which encapsulates some property of the input instance. Algorithms with running times $f(k).n^{\mathcal{O}(1)}$, where k is a parameter, are called *fixed-parameter tractable* in

parameterized complexity. Parametrizing problems in this way allows one to solve NP hard significantly faster subject to some constraints on the parameters. In this project, we have tried to solve a few problems pertaining to temporal graph by the parameterized complexity approach. Scientific literature related to the subject has already defined concepts of walks and paths in temporal graphs. The first problem is about finding a path involving k vertices in a given temporal graph. While this is a problem which is well-explored in static graphs, it remains unsolved in temporal graphs. After completing this problem, another viable problem had to be found. Thus, considerable amount of time was spent on exploring the present literature on temporal graphs. Two new problems were created, based on problems from existing literature.

The first problem, which we shall refer to as *foremost k-covering walk* requires us to find the earliest ending temporal walk which 'covers' at least k vertices, in a given temporal graph G and an integer k. The second problem is a more restricted version of the same, where the input contains a temporal graph G and a subset S of vertices of G. S has at most k elements. We need to find the foremost temporal walk which covers S. Note that there are two versions of temporal paths/walks depending upon the transversal at the same time stamp; strict temporal path/walk and non-strict temporal walk/path. All of the above mentioned questions needs to be solved for both of these cases.

The first problems pertaining to k-paths are solved using randomized techniques, which construct algorithms which can solve problems with a fixed, constant amount of probability which is reasonably large. One of the most prominent randomized technique is colour coding, which was originally developed in 1994 to discover predefined motifs in graphs [2]. It can be used to discover structures such as paths, cycles, and cliques. To my best knowledge, this is the first time that colour coding is used to discover k-paths in temporal graphs. Furthermore, another algorithm is proposed which has better running time than the original colour coding algorithm.

Most of the techniques for this thesis was adopted from the book *Parameterized Algorithms* [5], which provides a comprehensive understanding of ideas and techniques in parameterized complexity. The "covering walk" problems were motivated by a paper on Traveling salesman problem in Temporal graphs [11]. This paper also proves the NP-hardness of the non-parameterized version of these problems. This thesis covers all the progress that are made throughout the course of the project, in answering the problems defined above. Additionally, it describes a few problems in present literature of temporal graphs. This will be done more in breadth than depth. The goal in undertaking this particular problem can be attributed

to the intrigue of exploring this unexplored field, and the practical relevance of it.

Chapter 2

Preliminaries

2.1 Computational Complexity

The origins of parameterized complexity is closely tied to the famous P vs NP problem. Presently, we mostly accept that NP problems cannot be solved in polynomial time. Parameterized Complexity opens an avenue to tackle this problems and increase their efficiency without violating *NP-hardness*. In this section, we will define the commonly used terms and concepts in this thesis. They can broadly be grouped under Computational complexity and temporal graphs. Less frequently used terms will be explained in their respective sections. Let us first define P and $NP-hard$ complexity classes.

Definition 2.1.1. *NP (Non-deterministic polynomial time) is a complexity class. A decision problem Q is said to be in NP if given an instance of Q and a potential solution S, it is possible to verify whether S is a solution or not in polynomial time.*

Definition 2.1.2. *A problem Q is said to be NP-hard if every problem in NP class can be reduced to Q is polynomial time. i.e, given an instance of any NP problem, we can create an instance of Q such that if we have the solution of Q, we can compute the solution of the original problem in polynomial time.*

The third class we define is called *NP-complete*. A problem is said to be NP-complete if it is both NP, and NP-hard. We can think of NP-complete as being the set of hardest

problems in NP.

Definition 2.1.3. *Algorithms with runnning time $f(k).n^{\mathcal{O}(1)}$ are called fixed-parameter tractable algorithms, where n is the size of the input instance, and k is a parameter provided with the input.*

The reader might notice that parameter has not been defined very well in this definition. This is because parameterized complexity does not impose many restrictions on the parameter. k is a number which encapsulates some aspect of the input instance, usually the solution size required, or some structural property of the input instance. Problems need not be parameterized using exactly one parameter either. A problem parameterized using multiple parameters is described later in this thesis.

Definition 2.1.4. *Algorithms with runnning time $f(k).n^{f(k)}$ are called XP algorithms (for slice-wise polynomial), where n is the size of the input instance, and k is a parameter provided with the input.*

Notice that when k is a fixed quantity, both FPT and XP algorithms are polynomial in n. However, as as k increases, XP algorithms can grow much more rapidly than FPT algorithms. Thus, our goal is to find FPT algorithms for given problems, if possible.

There are a few other concepts in parameterized complexity which are general to the field, but not to this project. These include *Kernalization, Parameterized Reduction, and W-hierarchy*. These are essential knowledge for any researcher in Parameterized Complexity. However, they are not used in this project and won't be elaborated here.

2.2 Temporal Graphs and related concepts

All of the problems in this project were based on *Temporal Graphs*. It is defined as follows:

Definition 2.2.1. *Temporal Graphs is defined as an ordered tuple $(V, E, \boldsymbol{L})$. Here, V is a non-empty set of vertices, $E \subseteq \{(u, v) : \forall u, v \in V\}$, and $\boldsymbol{L}$ is a function $\boldsymbol{L}$: $E \mapsto \{S : S \subset \mathbb{N}\}$.*

Here, V denotes the set of vertices, and E is the set of all edges present in the graph irrespective of time. Thus, (V, E) forms a static graph which is also the *underlying graph* of

the given temporal graph. $\boldsymbol{L}$ is a set of labels, associated with edges given in E. These labels specify the time points at which the edge exists. We will use $\boldsymbol{L}(a, b)$, for $a, b \in V$ to notate the set of time points at which the edge between a and b exists. $\boldsymbol{L}^{-1}(t)$ shall represent the set of edges existing at time point t. We require two more definitions:
Define $\Delta(\boldsymbol{L}) := \{t : \boldsymbol{L}^{-1}(t) \neq \emptyset \}$
Define $|\boldsymbol{L}| := |\{t : \boldsymbol{L}^{-1}(t) \neq \emptyset\}|$.
Here, $|\boldsymbol{L}|$ denotes the number of "valid" time points in the temporal graph and $\Delta(\boldsymbol{L})$ enlists these time point. Temporal graphs can also be denoted in a slightly different manner as $(V, \mathbf{E}, \tau)$. Here, V is the constant set of vertices of the temporal graph and τ is the last time index. Let ϵ be the set of all edges between the vertices specified by V. Then, $\mathbf{E}$ is a subset of $\epsilon \times [1, 2, ...\tau]$. We will use $\mathbf{E}_t$ to represent the set of edges present at time point t.
In this definition, we have restricted the time stamps of temporal graphs to be a subset of natural number. This need to be the case in general. However, we will be using this definition of temporal graphs throughout this document.

Since temporal graphs and static graphs share many properties, we can try to find analogues of static graph concepts in temporal graphs. It is possible to define things such as paths, walks, and connectivity in temporal graphs. The following is a definition for *strict temporal paths*:

Definition 2.2.2. *A strict temporal k-path, also called a time-respecting k-path is a sequence $P = ((\{v_0, v_1\}, t_1), (\{v_1, v_2\}, t_2), ...(\{v_{k-1}, v_k\}, t_{k-1}))$, where $t_{i-1} < t_i$, $\forall (\{v_i, v_{i+1}\}, t_i) \in P$, $(v_i, v_{i+1}) \in \boldsymbol{E}_{t_i}$ and, $v_i \neq v_j \; \forall \; i \neq j$.*

This path is "strict" because it does not accommodate multiple edges from the same time stamp. Consequently, the maximum length of a strict temporal path $\leq minimum\{|V| - 1, \Delta(L)\}$. Without this restriction, we obtain *non-strict temporal paths.*

Definition 2.2.3. *A non-strict temporal k-path, is a sequence $P = ((\{v_0, v_1\}, t_1), (\{v_1, v_2\}, t_2), ...$ $.., (\{v_{k-1}, v_k\}, t_{k-1}))$, where $t_{i-1} \leq t_i$, $\forall (\{v_i, v_{i+1}\}, t_i) \in P$, $(v_i, v_{i+1}) \in \boldsymbol{E}_{t_i}$ and, $v_i \neq v_j \; \forall \; i \neq j$.*

Non-strict temporal paths can accommodate multiple edges from the same time point, as long as they form a path at that time index. This means that the path length can exceed the time bound, and thus the maximum path length $= |V| - 1$. We can also observe that

the problem of finding the longest non-strict temporal paths in temporal graphs is at least as hard as finding the longest path in static graphs. This is because static graphs can be considered as temporal graphs with only one time point. Let us prove this:

Lemma 2.2.1. *Finding non-strict temporal paths in temporal graphs is at least as hard as finding paths in static graphs. i.e, if there is an algorithm Q which can find the longest non-strict temporal path given any temporal graph $(V, E, \boldsymbol{L})$ in time $\lambda(|V|)$, it is possible to find the longest path in any instance of static graph in time at most $\lambda(n)$, where n is the size of the input instance.*

Proof. Given any input instance $G = (V, E)$ of a static graph, consider the temporal graph instance $T = (V, E, \boldsymbol{L})$, where $\boldsymbol{L}(e) = \{1\} \forall e \in E$. We can now solve this using algorithm Q to find the longest non-strict temporal graph. Any solution, S returned by Q would work for G. Let the solution be $P = ((\{v_0, v_1\}, 1), ...(\{v_{k-1}, v_k\}, 1))$. All time indices will have to be 1, because it is the only time point available. Then, all (v_i, v_{i+1}) edges in the path will belong to E as well. Thus, the same sequence of vertices will form a solution in G. □

This simple proof demonstrates that temporal graph problems are most likely harder than static problems when we allow the selection of multiple edges at each stage. Notice that it is possible to do similar reduction in case of strict paths as well. This can be done by creating a temporal graph instance which has $|V| - 1$ time points each of which has all the edges of E. It is possible to define other temporal paths. For example, *continuous temporal paths* are temporal paths where "staying" in one vertex is not allowed. This means that the time indices $t_1, t_2, ..., t_k$ have to be a continuous subsequence of ordered $\Delta(\boldsymbol{L})$. However, these are not relevant to the project directly and thus, will be omitted.
Similar to paths, we can define temporal walks as well.

Definition 2.2.4. *A strict temporal k-walk, also called a time-respecting k-walk is a sequence $P = ((\{v_0, v_1\}, t_1), (\{v_1, v_2\}, t_2), ...(\{v_{k-1}, v_k\}, t_{k-1}))$, where $t_{i-1} < t_i$, $\forall(\{v_i, v_{i+1}\}, t_i) \in P$, $(v_i, v_{i+1}) \in \boldsymbol{E}_{t_i}$.*

This is a straightforward definition, which is analogous to the case of static graphs. We can define *non-strict temporal walks* similarly, by permitting $t_{i-1} \leq t_i$, as with the case of paths.
Having a temporal aspect allows us to define modified versions of paths and walks on temporary restrictions. *Foremost temporal walk* is defined as follows:

Definition 2.2.5. *A Foremost temporal walk W satisfying condition Ω is a temporal walk such that it satisfies condition Omega, and no other walk in the graph which satisfies Ω ends earlier than W.*

We can similarly define Foremost temporal paths as well. Note that these definition can apply for strict and non-strict versions with slight modifications.

Chapter 3

Randomized Algorithms

3.1 Randomized Algorithms

Randomized FPT algorithms are a category of algorithms which uses randomness in its working. Given an input instance, each runthrough of a randomized algorithm will produce different result. Thus, randomized algorithms are inherently probabilistic, and should be characterized on those terms. Any given random algorithm should have a sufficiently large probability of finding the correct solution.

For describing these algorithms better, we may consider the input to these algorithm to consist of a stream of random bits, in addition to the graph (and parameters). This string provides data to the algorithm to make choices, and is the only source of randomness in the algorithm. If this random bit is of maximum length b, then the probability space is of size 2^b. We can now talk about the probability of success or failure of the algorithm in terms of this probability space. In this project, we have mainly dealt with one-sided error Monte Carlo algorithms. One-sided error Monte Carlo algorithms will always return false if the the instance is a no-instance, while if the instance is a yes-instance, the algorithm will return True with probability $p \in (0, 1)$.

In many of the algorithms we will see, the success probability, p is a function of k. This is common in a parameterized setting. Thus, we have bounds like $p = 1/2^{\mathcal{O}(k)}$, or more generally, $p = 1/f(k)$. We can see that by repeating the algorithm, we can improve the success probability. More specifically, let us call our randomized algorithm as RA, and

define a new algorithm by repeating RA t times. This new algorithm, $t - RA$ returns $False$ if every iteration of RA returns $False$. If any instance of RA in the $t - RA$ returns $True$, then $t - RA$ returns $True$. Let us now try to evaluate the probability of success of this new algorithm. $t - RA$ fails only when each instance of RA fails. The probability of RA failing, i.e, RA returning $False$ for a yes-instance is $(1 - p)$ Thus, the probability of $t - RA$ failing is $(1 - p)^t$.
Now,

$$(1 - p)^t \leq (e^{-p})^t = 1/e^{pt} \tag{3.1}$$

The above hold true because $(1 + x) \leq e^x$. We can now see that the new success probability is at least $1 - 1/e^{pt}$, and the probability of failure decreases exponentially with the number of repetitions. In particular, by repeating the algorithm $\lceil \frac{1}{p} \rceil$ times, we can obtain a success probability which is a constant independent of k. When working with polynomial time algorithms, it is important to bound p by a polynomial of the input size. Otherwise, the running time of the algorithm will not be polynomial for getting a constant, fixed probability of success. However, when dealing with FPT algorithm, we are allowed a bit more leeway. The success probability of the algorithms here are allowed to be bound by $1/f(k).n^{\mathcal{O}(1)}$. This is because we are allowed to repeat the algorithm $f(k).n^{\mathcal{O}(1)}$ times to obtain a constant probability. Note that these bounds on probability are not absolute, but are considered conventions because of their usefulness. Randomized algorithms are now recognized to be one of the important techniques for constructing efficient algorithms. Randomized algorithms are often much more faster than deterministic algorithms, and they are often simpler to describe as well.

Let us now discuss two different random algorithms. The first algorithm will be to solve the Feedback Vertex Set. The second problem is k -Path Problem. We will describe the colour coding technique here, which we will later use in solving the k -Path problem in Temporal graphs.

3.2 Feedback Vertex Set

We will describe a simple randomized algorithm for finding Feedback vertex set of multi-graphs here. Note that this is not going to be complete proof of the algorithm. This section is only meant to convey an idea of creating randomized algorithms. The next section will

provide the complete proof of another randomized algorithm.

Let G be a given multigraph. We say that $X \subseteq V(G)$ is a *feedback vertex set* of G, if $G[V(G) \setminus X]$ is an acyclic graph. In the FEEDBACK VERTEX SET problem, we are given an undirected multigraph G, and an integer l. The problem is to find a feedback vertex set of size at most k in G.

First, we will use a few "reduction rules" to eliminate some redundancy from the graph and obtain an equivalent instance with the property that the original instance is true if and only if the new equivalent instance is true. We will apply these rules repeatedly till none of them can be applied to the instance.

Reduction Rule 1 : If a vertex v has a loop, remove v from the graph and reduce k by one.

Since all vertices with loops form cycles, they have to be added to the feedback vertex set trivially. If there exists a feedback vertex set of size $k-1$ in the reduced instance, we can add v to this feedback vertex set to get a solution to the whole graph.

Reduction Rule 2: If there is a vertex in the graph with degree at most 1, delete it.

No vertex with degree 1 or 0 can be part of a cycle. Thus, we can safely remove them without affecting the solution.

Reduction Rule 3: If there is any edge in the graph with multiplicity greater than 2, reduce it to 2.

Notice that any edge of the the graph with multiplicity greater than or equal to two will necessitate at least one of its vertices to be in the feedback vertex set. Adding either of the vertices into the feedback vertex set will eliminate all the cycles associated with the multi-edge as well. So, we can reduce the size of the multi-edge which preserving the solution.

Reduction Rule 4: If v is a vertex with degree two adjacent to vertices u and w, eliminate v from the graph, and connect u and w with an edge. If u and w are already connected, increase the multiplicity of that edge by one. If u and w are the same, add a loop of the vertex.

The first point here is that v cannot be its own neighbor. If it did, Reduction Rule 1 would apply and it would be eliminated. This rule has a few cases. Let us first evaluate the case where there is a vertex v of degree 2 which is connected to two disconnected vertices u and w. In this case, any cycle which involves v should involve both u and w. Thus, if v is in the feedback vertex set, we could replace it with u or w in the feedback vertex cover. Furthermore, it better to choose a vertex out of u and w as both of them have degree ≥ 2, as we have implemented Reduction rule 2. Thus, eliminating v here would work. If v has

only one neighbor, say, u, it implies the presence of a multiedge with u and thus, either v or u have to be added to the feedback vertex set. By a similar line of reasoning as before, we can see that eliminating v and adding u to the feedback vertex set is the optimal strategy here. Applying Reduction Rule 4 followed by Reduction Rule 1 would result in the same. If the neighbours of v are already connected, then the three vertices form a triangle, and we select either of the neighbours into the feedback vertex set.

In this process, if k drops below 0 at any step, it means that it is not possible to remove all of the cycles by removing only k vertices. Thus, it is a no-instance. Otherwise, we get an equivalent instance G' with the following properties:

- There are no loops
- Maximum multiplicity of edges is 2. ie, there are only single and double edges
- Minimum degree of any vertex is at least 3

Now, we can move on to the randomized algorithm. We will first describe a lemma, which we will later use in defining the algorithm.

Lemma 3.2.1. *Let G be a multigraph on n vertices, with minimum degree at least 3. If X is a feedback vertex set of G, then more than half of the edges have at least one end point in X.*

Proof. Consider $H = G - X$. H is a forest. Now, we only need to show that edges within H is less than other edges, because every edge not within H is incident to some vertex in X. i.e, $|E(G) \setminus E(H)| \geq E(H)$. We also know that H is a forest. Thus, $E(H) \leq V(H)$. It is enough to show that $|E(G) \setminus E(H)| \geq V(H)$. Now, we define 3 sets, $V_{\leq 1}$, V_2, and $V_{\geq 3}$. These denote vertices in $V(H)$ with degree less than or equal to 1, exactly two, and greater than or equal to three, respectively. Note than this classification is entirely based on the edges in $V(H)$ and not $V(G)$. Let us also define Q, as the set of all edges which have one endpoint in X and the other end point in $V(H)$. Now, every vertex in $V_{\leq 1}$ contributes at least 2 edges to Q, as they are of minimum degree 3 with at most of neighbour in $V(H)$. Similarly, every vertex in V_2 contributes at least one edge to Q. Since H is a forest, number of vertices of degree at least 3 is strictly less than the number of vertices with degree at most

1.i.e, $|V_{\leq 1}| \geq |V_{\geq 3}|$. Now, we can put all of these together to obtain the following:

□

The above lemma says that if we pick any arbitrary edge from the graph, there is at least a 50% chance that the edge is connected to a vertex from the feedback vertex set. Thus, if we pick one end point of this edge randomly, there is at least 1/4 probability that it is in the solution. We will use this idea in the following algorithm.

Theorem 3.2.2. *There exists a polynomial time algorithm that, given a* Feedback Vertex Set *instance* (G, k)*, either finds a feedback vertex set of size at most* k*, or reports the failure to find one. If the algorithm is given a yes-instance, it will find the solution with probability at least* 4^{-k}*.*

Proof. Given a feedback vertex set instance, we will apply the Reduction rules mentioned in 3.2 repeatedly till none of them can be applied. If the rules return a no-instance, we are done. Otherwise, we have a reduced instance, and let us call it (G', k'). Here, $0 \leq k' \leq k$, and G' has minimum degree 3. While implementing Reduction rule 1, we have removed several vertices from the graph, and reduced the parameter. Consider X_l to be the set of all such vertices. We have, $|X_l| = k - k'$. Also, if X' is a feedback vertex set of G', $X' \bigcup X_l$ is a solution to the whole graph. Thus, we only need to find an X' such that $|X'| \leq k'$. Note that if G' is empty, or is a forest, we have $X' = \emptyset$. Thus, X_l forms a solution of acceptable size.
Otherwise, we pick an edge e from G' uniformly at random and select one of the endpoints of e uniformly at random, independently. Let us call this vertex u. From 3.2.1, we know that there is a probability of 1/4 that this vertex belongs in the solution. Now we recurse on $(G' - \{u\}, k' - 1)$. i.e, we try to find a feedback vertex cover in this new instance using the same methodology we used before. If the recursion returns a feedback vertex set X', we return $X' \bigcup X_l$ as the solution. Otherwise, recursive step returns a failure and we return a $False$ as the output. This describes the entire algorithm. Now, we will discuss a bound on the probability of the algorithm. Note that this algorithm can proceed only till we have selected k' vertices. Furthermore, we have already established that the chance that a vertex v we have selected belongs to the feedback vertex cover is at least 1/4. Thus, the probability of selecting the correct set of vertices is $1/4^{k'}$, and $1/4^{k'} \geq 1/4^k$. As we discussed earlier, we can repeat this algorithm 4^k times to get a constant probability. □

Note here that we could have branched on the vertices in the above algorithm to obtain a bounded search tree algorithm. This algorithm would be deterministic, with comparable running time. Thus, this particular example is not a great example of the power of randomized algorithms. However, it does illustrate how randomization could be used in creating algorithms.

3.3 Colour Coding

Colour coding was introduced as a technique to detect the presence of a k-vertex "pattern" graph, H on a given n-vertex graph G. i.e, to detect the existence of a subgraph of G which is isomorphic to graph H. This makes the technique very versatile, allowing it to detect cliques, paths of length k, and cycles. This technique reduces the brute force time consumption of $\mathcal{O}(n^k)$ to $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ for forests and when H is of constant treewidth. Since clique problem is conjured to be hard and Colour coding allows us to solve it, we do not expect the running time to be significantly better.
Colour coding algorithm works by coloring vertices of the graph randomly in such a way that if the H is present in the graph G, it can be detected efficiently. Let us define a k-path as a path in a graph which involves exactly k vertices (and consequently, $k-1$ edges).
Finding *walks* on graphs is quite simple. There are algorithms which can do this in polynomial time. One of the toughest parts of the k-path is ensuring that vertices are not repeated. If we take a brute force approach and keep track of every vertex visited, the complexity becomes $\binom{n}{k}$, which is undesirable.

A k-path which has all of its vertices coloured with different colours is called a "colourful k-path". In the path problem for static graphs, we colour the vertices randomly with k colours and hope that if there exist a k-path in the graph, it coloured in a colourful manner. We detect the existence of a colourful path in a graph, if it exists through dynamic programming. We will see that a related approach can be taken to find k-path in temporal graphs later on.
These algorithms can be derandomized in the exact same manner as the colour coding algorithm for longest path in static graphs.
First, we will prove a lemma which we will use for the algorithm.

Lemma 3.3.1. *Let V(G) be the set of vertices of graph G and $S \subseteq$ V(G) such that $|S| = k$.*

If the elements of V(G) are coloured at random in a uniform and independent manner, then all the elements of S will be coloured with pairwise distinct colours with probability at least e^{-k}.

Proof. $V(G)$ can be coloured in k^n ways. The number of distinct ways of colouring $V(G)$ while colouring S in a colourfully is $k!k^{n-k}$. This includes $k!$ ways of colouring S and k^{n-k} ways of colouring $V(G) \setminus S$.
Now, we can use the known inequality $k! > (k/e)^k$ to derive that $e^{-k} < k!/k^k = k!k^{n-k}/k^n$. This proves the lemma.

□

We will now show that if there is a colourful path in the graph, it can be found in FPT time. This is a dynamic programming algorithm.

Lemma 3.3.2. *Let G be a graph, and let $\chi : V(G) \mapsto [k]$ be a colouring of its vertices with k colours. It is possible to check if a colourful k-path exists in the graph deterministically, in $2^k n^{\mathcal{O}(1)}$ time, and return one such path if it exists.*

Proof. Let $V_1, V_2, ..., V_k$ be a partitioning of $V(G)$ such that all vertices in V_i are coloured i. Now, we can use dynamic programming. Consider S to be a non-empty subset of $\{1, ..., k\}$(the possible set of colours), and let u be a vertex in $V(G)$. Let us define a function $PATH(S, u)$, such that $PATH(S, u)$ is True if there is a colourful path with all colours in S ending at u, and colour of u is in S. It is false otherwise. When $|S| = 1$, $PATH(S, u)$ is True if and only if $S = \{\chi(u)\}$. If $|S| > 1$, The following recurrence is valid.

$$\text{PATH(S, u)} = \begin{cases} \vee\{PATH(S \setminus \{\chi(u)\}, v) : uv \in E(G)\}, & if \chi(u) \in S \\ False, & \text{otherwise} \end{cases} \tag{3.2}$$

If there is a colourful path of colours in S ending at u, then at least one neighbour v of u should have a colourful path of colours in $S \setminus \{\chi(u)\}$ ending at v. This is how the recursion is defined. Now, let us compute the time complexity of the algorithm. There are 2^k possible subsets of colours possible, and n vertices in the graph. Thus, $2^k n^{\mathcal{O}(1)}$ is the time to compute all the $PATH(S, u)$ problems possible, and is the time complexity. For checking if there is a colourful path in the graph, we only need to check if $PATH([k], v)$ is true for some vertex v in the graph. □

We can obtain the complete algorithm by putting together Lemma 3.3.1 and Lemma 3.3.2.

Theorem 3.3.3. *There exists a randomized algorithm of time complexity* $(2e)^k n^{\mathcal{O}(1)}$ *that takes a* (G, k) *instance as the input, and returns false if there is no* k*-path in the graph. If there is a* k*-path in the input instance, it returns a solution with a constant probability.*

Proof. As we discussed at the beginning of this chapter, we will first show an algorithm which runs in time $(2)^k n^{\mathcal{O}(1)}$ and find any k-paths in the graph with a success probability e^{-k}. Then, we can repeat the algorithm e^k times to obtain a constant probability of success. Given an input instance (G, k), we colour vertices of the graph with k colours in a uniformly random manner. Let $\chi : V(G) \mapsto [k]$ denote this colouring. We now run the algorithm of Lemma 3.3.2 on the graph G with colouring χ. If it detects a colourful path, it returns this path as the solution. Otherwise, it fails to find a path, and reports this. This is the entire algorithm. It is clear that if the algorithm returns a k-path, it is a solution to the problem. Thus, we will be done if we can bound the probability of finding the solution when a yes-instance is given. Assume that the input instance is a yes-instance. Then, there is a k-path in the graph. By Lemma 3.3.1, all of these vertices will be coloured colourfully by a random colouring with probability at least $1/e^k$. Then, algorithm described in Lemma 3.3.2 can deterministically find the k-path. This concludes the proof. □

This concludes this section. We will be using the techniques mentioned here later on, in temporal graphs a number of times to solve the central problems of this project.

Chapter 4

k-Path on Temporal graphs

4.1 Colour coding algorithm for longest strict path in temporal graph

Colour coding was introduced as a technique to detect the presence of a k-vertex "pattern" graph, H on a given n-vertex graph G. i.e, to detect the existence of a subgraph of G which is isomorphic to graph H. This makes the technique very versatile, allowing it to detect cliques, paths of length k, and cycles. This technique reduces the brute force time consumption of $\mathcal{O}(n^k)$ to $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ for forests and when H is of constant treewidth. Since clique problem is conjured to be hard and Colour coding allows us to solve it, we do not expect the running time to be significantly better.

Colour coding algorithm works by coloring vertices of the graph randomly in such a way that if the H is present in the graph G, it can be detected efficiently. Let us define a k-path as a path in a graph which involves exactly k vertices (and consequently, $k-1$ edges).

A k-path which has all of its vertices coloured with different colours is called a "colourful k-path". In the path problem for static graphs, we colour the vertices randomly with k colours and hope that if there exist a k-path in the graph, it coloured in a colourful manner. We detect the existence of a colourful path in a graph, if it exists through dynamic programming. We take a similar approach here in solving the k-strict path problem in temporal graphs.

Since the vertex set for a temporal graph is fixed, we may still colour it randomly and expect for the path to be coloured colourfully. However, identifying the path through dynamic

programming becomes tricky as we have to incorporate the definition of temporal paths into it, and account for the time-varying edges. These algorithms can be derandomized in the exact same manner as the colour coding algorithm for longest path in static graphs.

Lemma 4.1.1. *Let V(G) be the set of vertices of graph G and $S \subseteq V(G)$ such that $|S| = k$. If the elements of V(G) are coloured at random in a uniform and independent manner, then all the elements of S will be coloured with pairwise distinct colours with probability at least e^{-k}.*

Proof. V(G) can be coloured in k^n ways. The number of distinct ways of colouring V(G) while colouring S in a colourfully is $k!k^{n-k}$. This includes $k!$ ways of colouring S and k^{n-k} ways of colouring $V(G) \setminus S$.
Now, we can use the known inequality $k! > (k/e)^k$ to derive that $e^{-k} < k!/k^k = k!k^{n-k}/k^n$. This proves the lemma.

□

Lemma 4.1.2. *Let (V, $\boldsymbol{E}$, τ) be a temporal graph. Let $\chi : V \to [1, 2, ..., k]$ be a colouring of V into k colours. There is a deterministic algorithm to detect the presence or absence of a k-colourful strict temporal path in the temporal graph, in FPT time.*

Proof. This is a dynamic programming algorithm. Let S be a non-empty subset of [1, 2,...., k]. Define STRICT_PATH(S, v, t) as follows:

$$\text{STRICT\_PATH}(S, v, t) = \begin{cases} True, & \text{if there is colourful strict temporal path of size } |S| \\ & \text{ending at time } t, \text{ at the vertex } v \text{ with all colours in} \\ & S. \\ False, & \text{otherwise} \end{cases} \tag{4.1}$$

By definition, v has to be coloured with some colour from S. If $S = \{i\}$, then STRICT_PATH(S, v, t) = True if and only if v is coloured with the colour i.
It can be defined by the following recursive relation. We shall define a recurrence relation to compute STRICT_PATH.

$$\text{STRICT\_PATH}(S, v, t) = \begin{cases} \bigvee\{\text{STRICT\_PATH}(S \setminus \{\chi(v)\}, v, t') : \text{uv} \in E_{t'}(G), t' < t\} & \\ & \text{if } \chi(v) \in S. \\ false, & \text{otherwise} \end{cases} \tag{4.2}$$

If a k-path ends at time t, and it has all the colours in S, then there should be at least one k-1 path, which is colourful in $S \setminus \{\chi(v)\}$. It should also end at a vertex which is a neighbor of v at some time before t. The reason that we restrict t' to strictly less than t is to ensure the strictness of the path. This is precisely the idea captured in the above recurrence relation. We'll see that if $t' \leq t$, the recurrence will find non-strict temporal path, in section 4.2.1. However, note that paths which does have an edge in the last layer will not be found by this algorithm because the restriction on t'. Thus, we need to add an extra, empty layer to the temporal graph at the time index $\tau + 1$. This layer does not change the longest path as it contains no edges.
All possible values of STRICT_PATH can be calculated in time $2^k(n\tau)^{\mathcal{O}(1)}$. There exists a colourful strict temporal k-path in the graph if and only if STRICT_PATH$([1, 2, .., k], v, \tau)$ is true for some $v \in V$. □

Lemma 4.1.3. *There is a randomized algorithm that takes a temporal graph, and a non-negative integer k as input and returns a k-strict temporal path in the graph or the failure to find one. If a yes-instance is given as the input, it will return the correct solution with a constant probability.*

Proof. The algorithm that we propose has two components; The first part, which adds the element of randomness to the algorithm and the second part, which is deterministic and runs in $2^k(n\tau)^{\mathcal{O}(1)}$ time. This algorithm has a time complexity of $2^k(n\tau)^{\mathcal{O}(1)}$ and given a yes-instance, return the correct solution with probability e^-k. Thus, upon running the algorithm e^k times repeatedly, we shall achieve success with a constant probability stated as above.
Given an input instance (V, **E**, τ), we shall first colour it with colors [1, 2, 3, ..., k] in an independent, uniformly random manner. We shall then use the algorithm mentioned in Lemma 4.1.3 to detect if there is a path in the randomly coloured graph. If there is a strict temporal path, and it coloured colourfully, then it shall be detected, and will be provided as

the output. If the algorithm fails to find a path, it shall report a failure to do so.
If there is strict temporal k-path in the graph, it shall be coloured colourfully with a probability of e^{-k}, as shown by Lemma 4.1.1. Any colourful paths shall be detected by the algorithm and returned as output. □

4.2 Colour coding algorithm for longest non-strict path in temporal graph

The Colour coding algorithm for longest strict path in temporal graph can be tweaked slightly to find non-strict temporal paths. Specifically, we can change the the part of the algorithm that detects k-colourful path in a randomly coloured graph, which is described in Lemma 4.1.2.

Lemma 4.2.1. *Let (V, **E**, τ) be a temporal graph. Let $\chi : V \to [1, 2, ..., k]$ be a colouring of V into k colours. There is a deterministic algorithm to detect the presence or absence of a k-colourful non-strict temporal path in the temporal graph, in FPT time.*

Proof. We shall define NON-STRICT_PATH as follows:

$$\text{NON-STRICT\_PATH}(S, v, t) = \begin{cases} True, & \text{if there is colourful strict temporal path of size } S \text{ ending at time } t, \text{ at the vertex } v \text{ with all colours in } S. \\ False, & \text{otherwise} \end{cases} \tag{4.3}$$

Now, we can define the recurrence relation for computing NON-STRICT_PATH(S, v, t), as follows:

$$\text{NON-STRICT\_PATH}(S, v, t) = \begin{cases} \bigvee\{\text{NON-STRICT\_PATH}(S \setminus \{\chi(v)\}, v, t') : \text{uv} \in E_{t'}(G), t' \leq t\} & \text{if } \chi(v) \in S. \\ false, & \text{otherwise} \end{cases} \tag{4.4}$$

If a k-path ends at time t, and it has all the colours in S, then there should be at least one

$k-1$ path, which is colourful in $S \setminus \{\chi(v)\}$. It should also end at a vertex which is a neighbor of v at t or some time before t. This is the reason that have changed the recurrence relation to allow $t' \leq t$.
All possible values of STRICT_PATH can be calculated in time $2^k(n\tau)^{\mathcal{O}(1)}$. There exists a colourful strict temporal k-path in the graph if and only if STRICT_PATH$([1, 2, .., k], v, \tau)$ is true for some $v \in V$.

□

Lemma 4.2.1. *There is a randomized algorithm that takes a temporal graph, and a non-negative integer k as input and returns a non-strict temporal k-path in the graph or the failure to find one. If a yes-instance is given as the input, it will return the correct solution with a constant probability.*

Proof. The structure of the proof is extremely similar to the that of Lemma 4.1.3. We shall first show that there is an algorithm which, if given a yes-instance, will run in a time of $2^k(n\tau)^{\mathcal{O}(1)}$ and return the correct answer with a probability of e^{-k}. Upon repeating this algorithm e^k times, we get the required result.
First, we shall colour the vertices of the temporal graph with colours $[1, 2, 3, .., k]$, in a uniformly random manner. Then, we shall run the algorithm described in Lemma 4.2.1. If the algorithm finds a k-path, it shall provide a positive result. If it fails to find one, it shall report the same.
If there is a non-strict temporal k-path in the graph, the vertices involved in that will coloured with pairwise distinct colours with a probability e^{-k}, as shown by Lemma 4.1.1.

□

4.3 Divide and colour algorithm for k-temporal Path

We have already established that temporal path problem is NP-hard and solved it using color coding method, showing that it is fixed parameter tractable. However, the algorithm had quite a large time complexity, at $(2e)^k(nk)^{\mathcal{O}(1)} = 5.43^k(nk)^{\mathcal{O}(1)}$. We will be attempting to reduce the time complexity by a small margin with this algorithm. The idea behind the algorithm is that if we split the set of vertices into two partitions in a uniformly random way, there is a chance that first half of the path will be fall into one partition and the second

half of the path into the other partition. Now, we can keep repeating this process with each of the partitions and given that we do the whole process repeatedly, it is possible to split the path into partitions continuously and as we do so, the path will get divided as described above with some probability. Therefore, if we can somehow identify all the paths of the appropriate length at a level in both partitions of a parent partition, we can then check how they connect with each other to identify all the paths in the parent partition. However, since we are dealing with Temporal graphs, the paths have to temporal in nature and we need to check if we can join two temporal paths, which is a bit trickier than checking if two static paths can be joined. This algorithm can be defined in two ways. After defining the original algorithm, we shall see that it can run much faster with a slight modification.

Let $X \subseteq V$. Then, define X_L and X_R as two mutually exclusive subsets of X where each vertex is assigned to X_L with probability 0.5 and to X_L and to X_R with probability 0.5. Furthermore, each vertex belongs to exactly one of X_L and X_R. Now, let $D_{X,l}$ be a $|X| \times |X|$ matrix and i, j be two vertices of X. Then, $D_{X,l}[i,j]$ is define as follows:

$$D_{X,l}[i,j] = \begin{cases} \phi & : \text{if there is no } l\text{-temporal path in } X \\ \{(t,t') : & \text{there is a temporal } i-j \text{ path with } l \text{ vertices} \quad : otherwise \\ & \text{starting at time } t \text{ and ending at time } t'\} \end{cases} \tag{4.5}$$

Note that in the above definition, temporal paths may be strict or non-strict. There are only minor differences in the algorithms for both and the above definition. If we can find $D_{V,k}$, then we are done as we now know all of the temporal k-paths existing in the temporal graph. Let us now define a matrix $\hat{D}_{X,l}$ such that if $\hat{D}_{X,l}[i,j] \neq \phi$, $\hat{D}_{X,l}[i,j] \subseteq D_{X,l}[i,j]$. i.e, Any path included in $\hat{D}_{X,l}$ is a "valid" path. Our algorithm will ensure that if $(t,t') \in D_{X,l}[i,j]$, then $(t,t') \in \hat{D}_{X,l}$ with a sufficiently high probability.
Let us now define a method to combine two given matrices of the type $D_{X,l}$ or $\hat{D}_{X,l}$. Let $\hat{D}_{L,p} \bowtie \hat{D}_{R,q} = \hat{D}_{V,r}$where $V = L \cup R$ and $r = p + q$ Then,

$$\hat{D}_{V,r}[i,j] = \left\{ \begin{matrix} (t,t') : \text{If } \exists\ v \in \text{ L, } w \in \text{ R, and } t < t_1 < t_2 < t_3 < t', \\ \text{such that } (t,t_1) \in \hat{D}_{L,p}[i,v],\ (t_3,t') \in \hat{D}_{R,q}[w,j], \text{ and } (v,w) \in \mathbf{E}_{t_2} \end{matrix} \right\} \tag{4.6}$$

The above definition might look slightly convoluted. The essence of it is that if there is

(t, t_1) strict temporal p-path in L and (t_3, t') strict temporal path of length q in R, such that $t < t_1 < t_3 < t'$. Then, we need only check for an edge between the ending point of one of these paths and the starting point of the other, at a time point strictly between t_1 and t_3. If there exists such an edge as well, then we can see that there is a strict temporal $(p+q)$ path starting at t and ending at t' in the combined vertex set of L and R. Note that $\forall v \in X$, $D_{X,1}[v, v] = \hat{D}_{X,1}[v, v] = \{(i, i) : i \in \{t : \boldsymbol{L}^{-1}(t) \neq \emptyset\}\}$. i.e, every vertex can be considered as a path of length 1 at all time points. The algorithm is described as follows:

Algorithm 1: Divide_and_Colour(T, k) algorithm for find a strict k-temporal walk in a temporal graph

Input : A set $X \subseteq V$, an integer l, $1 \leq l \leq k$
Output: $\hat{D}_{X,l}$

if $l = 1$ **then**
  Initialize $\hat{D}_{X,l}[v, u] = \emptyset \ \forall v, u \in X$;
  Set $\hat{D}_{X,l}[v, v] = \{(i, i) : i \in \{t : \boldsymbol{L}^{-1}(t) \neq \emptyset\}\} \ \forall v \in S$;
  Return $\hat{D}_{X,l}$;
else
  Partition the vertices of X into two sets L and R uniformly at random;
  $\hat{D}_{L,\lceil \frac{l}{2} \rceil}$:= Divide_and_Colour $(L, \lceil \frac{l}{2} \rceil)$;
  $\hat{D}_{R,\lfloor \frac{l}{2} \rfloor}$:= Divide_and_Colour $(R, \lfloor \frac{l}{2} \rfloor)$;
  return $\hat{D}_{X,l} := \hat{D}_{L,\lceil \frac{l}{2} \rceil} \bowtie \hat{D}_{R,\lfloor \frac{l}{2} \rfloor}$;
end

If there is a k-path in X, Divide_and_Colour(X, k) will find if the path is split into two section from the mid-way point, into L and R, and Divide_and_Colour detects these sections as well. Let us try to evaluate the probability that the algorithm finds a solution if the input instance is a yes-instance. This boils down to finding the probability with which $\hat{D}_{V,k}[u, v]$ is true if $\hat{D}_{V,k}[u, v]$ is true, as $\hat{D}_{V,k}[u, v]$ contains information about every path between u and v.

Let p_k denote the infinimum of this probability for a path of length k, over the space of all input instances $T = (V, \mathbf{E}, \tau)$, $X \subseteq V$, and $u, v \in X$. If $l = 1$, then we have $D_{X,l} = \hat{D}_{X,l}$ from the definition.

Otherwise, we partition X into L and R uniformly, and run DIVIDE_AND_COLOUR on those instances, with parameter as $\lceil \frac{l}{2} \rceil$ and $\lfloor \frac{l}{2} \rfloor$ respectively. But this requires that first $\lceil \frac{l}{2} \rceil$ vertices of the solution go into L and the remaining into R. The probability of this happening is 2^{-l}. By construction, the subpath on L will be identified with probability $p_{\lceil \frac{l}{2} \rceil}$ and the subpath

on R will be identified with probablity $p_{\lfloor \frac{l}{2} \rfloor}$. From this, we obtain the recursive bound:

$$p_l \geq 2^{-l} p_{\lceil \frac{l}{2} \rceil} p_{\lfloor \frac{l}{2} \rfloor} \tag{4.7}$$

Let us solve this recurrence relation. Notice that in the recurrence relation, the parameter l drops by half at each stage. This means that there are roughly $log(k)$ levels possible for this recursion. Also, there are two sub-problems at each stage. For each $i = 0, 1, ..., \lfloor log(l) \rfloor - 1$, the algorithm needs to make rougly 2^i times a correct partition of roughly $k/2^i$ vertices. Thus, the probability is:

$$p_l \approx \prod_{i=0}^{log(l)-1} \left(\frac{1}{2^{l/2^i}} \right)^{2^i} = 2^{\mathcal{O}(l \log(l))}$$

To obtain a constant probability, we need to run the algorithm $2^{\mathcal{O}(llog(l))}$ times. This is significantly worse than our previous colour coding algorithm. However, we can improve it considerably by modifying it slightly. For this, consider the relationship $p_l \geq 2^{-l} p_{\lceil \frac{l}{2} \rceil} p_{\lfloor \frac{l}{2} \rfloor}$. This means that if we can improve the probabilities $p_{\lceil \frac{l}{2} \rceil}$, and $p_{\lfloor \frac{l}{2} \rfloor}$, we improve the probablity p_l as well. Furthermore, the parameter in L and R is roughly half that of X. Thus, computational expense is significantly lower. This means that we can repeat lower levels in the recursion more times to obtain higher probabilities of success, and increase the success probability of the overall algorithm. So, instead of repeating DIVIDE_AND_COLOUR(X, k) $2^{\mathcal{O}(llog(l))}$ times, we select a function $f(l, k)$ for calculating how many times the algorithm repeats DIVIDE_AND_COLOUR(B, l), $B \subseteq X$ in the computation of recurrence.

Given the input instance (T, k), we define our new algorithm as follows:

This algorithm is more time consuming because there is a larger search tree involved. However, the probability increase considerably. In fact, $f(k, l) = 2^l log(4k)$ will do the job. Proof of the same can be found in section 5.4 of *Parameterized Algorithms* [5].

Algorithm 2: FASTER_DIVIDE_AND_COLOUR (T, k) algorithm for find a strict k temporal walk in a temporal graph faster.

Input : A set $X \subseteq V$, an integer l, $1 \leq l \leq k$
Output: $\hat{D}_{X,l}$

if $l = 1$ **then**
  Initialize $\hat{D}_{X,l}[v, u] = \emptyset \ \forall v, u \in X$;
  Set $\hat{D}_{X,l}[v, v] = \{(i, i) : i \in \{t : \boldsymbol{L}^{-1}(t) \neq \emptyset\}\} \ \forall v \in S$;
  Return $\hat{D}_{X,l}$;
else
  Initialize $\hat{D}_{X,l}[v, u] = \emptyset \ \forall v, u \in X$;
  repeat $f(k, l)$ **times**
    Partition the vertices of X into two sets L and R uniformly at random;
    $\hat{D}_{L,\lceil \frac{l}{2} \rceil}$:= Divide_and_Colour $(L, \lceil \frac{l}{2} \rceil)$;
    $\hat{D}_{R,\lfloor \frac{l}{2} \rfloor}$:= Divide_and_Colour $(R, \lfloor \frac{l}{2} \rfloor)$;
    $D'_{X,l} := \hat{D}_{L,\lceil \frac{l}{2} \rceil} \bowtie \hat{D}_{R,\lfloor \frac{l}{2} \rfloor}$;
    for $u, v \in X$ **do**
      $\hat{D}_{X,l}[u, v] = \hat{D}_{X,l}[u, v] \cup D'_{X,l}[u, v]$;
    end
  end
  return $\hat{D}_{X,l}$;
end

Chapter 5

Temporal Walks and related problems

5.1 Temporal Walks

A strict k-temporal walk is a sequence $X = ((\{v_1, v_2\}, t_1), ..., (\{v_{k-1}, v_k\}, t_{k-1}))$ such that $t_i < t_j, \forall i < j$. In this sequence, vertices may be repeated. The relationship between a path and walk in a temporal graph might seem extremely similar to the one between paths and walks in static graphs. However, there are notable differences between the two. There is an inherent sense of directionality in temporal graphs for paths and walks. This is because we are limited on how we can move with respect to the time indices. For example, existence of a (u, v) walk in the time frame (t_1, t_2) in a temporal graph provides no evidence to the existence of a (v, u) path. This is not the case in static graphs. Foremost walks are temporal walks which ends the earliest while satisfying some property. i.e, X is a foremost temporal walk satisfying condition Q if there is no other temporal walk which ends before X, and satisfies Q. In this section, we will aim to solve a few problems related to temporal graphs. The class of problems addressed here are Covering Walk Problems. The first problem is defined as follows:

Given a temporal graph, and an integer k, find the foremost walk which covers at least k vertices.

Here, a walk 'covers' a vertex if it passes through that vertex at least once. Most graph problems in literature deal with paths over walks. Additionally, the literature on temporal walks is very little. Most techniques used for paths cannot be used for finding temporal

walks either, because of drastic increases in complexity. This makes this problem a difficult one. Thus, we try an alternate version of this problem, where we fix the set of vertices to be covered. This is discussed in the following section. We will use the notation S-covering walk, where S is a set of vertices, to denote a walk which covers all the vertices of S.
If X is a temporal walk defined on a temporal graph $(V, E, \boldsymbol{L})$, and $W \subset V$ we will also use the notation $X \setminus W$ to represent the sequence obtained by removing all the vertices of W from X. This sequence might represent a collection of disjoint walks.

5.2 Length Bounds on Walks

This section aims at developing a bound on temporal walk length under certain criteria. For this, let us first define S-Covering walk as follows:

Definition 5.2.1. *S-Covering walk takes a temporal graph $(V, E, \boldsymbol{L})$,and a set $S \subset V$. Here, $k = |S|$ is the parameter. The problem is to find a strict temporal walk, X which traverses through all the vertices of S.*

Consider an S-covering walk, W. We can split W into two parts; the sections of W which are inside S, and the the components which are outside S. Let us say $W = W_1, W_2, ..., W_l$ where W_i are walks, and $V(W_i) \in S \forall$ odd i and $V(W_i) \in H \forall$ even i. This split is possible because the walk has to start in S and end in S. We can find a bound for the total walk length if we can find bounds for each W_i. Here, we will show that the total length of the walk inside S is bounded by a function of k. This means that if there is any solution of size greater than the bound size, there exists another solution of size less than the bound size. More specifically, if given a solution of size greater than the bound, we can obtain a new solution of bounded size inside S. This is demonstrated below.

Lemma 5.2.1. *Given an instance (G, k, S) of the Covering Walk problem, and a solution X, we can bound the total length of the walk X in S by $\mathcal{O}(k^2)$.*

Proof. The given solution is a walk of size $|X| = x$. Let us consider the sequence of vertices in X, and eliminate all the vertices of $\overline{S}$ which are in X. This new sequence, $X_s = u_1, u_2, .., u_r$ is a subsequence of X which only contains the vertices S. Notice that vertices in X_s can be repeated multiple times. We also rename the vertices, without loss of generality such that

there is no vertex u_i appearing for the first time in the walk after a vertex u_j has appeared, where $j > i$. For example, vertex u_5 cannot appear in X till vertices u_1 to u_4 has appeared at least once. Notice that this sequence is a collection of separate walks in S. It also has every vertex in S, as X is a solution. Thus, we can construct a sequence $p_1, .., p_k$ indicating the position when any vertex in S is first encountered in X_s. This is an increasing sequence, with p_i denoting the time at which vertex u_i first appears. Now, notice that there can be at most i unique vertices between p_i and p_{i+1}. This is because there can only be i vertices visited before visiting vertex u_{i+1}. If a vertices v is repeated between positions p_i and p_{i+1}, we may eliminate all vertices between the repetitions and consider the vertex only once. This will work equally well because no new vertices are visited and we are allowed to stay in one vertex through multiple time indices. Furthermore, this is not influenced by moving in and out of S. This implies that no repetitions are possible in this sequence. Thus, the maximum size of this sequence is $k - 1$, because we cannot use vertex u_i again. This implies that the length of the sequence X_s, T is:

$$T = \sum_{i=0}^{k-1} i + k = \frac{k(k+1)}{2} = \frac{k^2 + k}{2} = \mathcal{O}(k^2) \tag{5.1}$$

The summation denotes the vertices present in the 'gaps', and the k term represents the first time each vertex appears in the walk. This completes the proof. □

It may be observed that the length of a walk that covers all the vertices of a temporal graph with n vertices is at most $\mathcal{O}(n^2)$. Also, $T = \frac{k^2+k}{2} \leq k^2 \ \forall k \geq 1$

5.3 Foremost S-Covering Walk

S-Covering walk parameterized by the size of S is a difficult problem. Initial attempts at finding an FPT algorithm for this problem yielded no results. However, adding an additional parameter made it possible to find an FPT algorithm. While this changes the parameterization, it does not change the problem itself. Let us discuss this algorithm. Foremost S-Covering walk problem we are going to address is stated as follows:

Definition 5.3.1. *S-Covering walk takes a temporal graph $(V, E, \boldsymbol{L})$, a set $S \subset V$, and an integer $l > 0$ as the input. Here, $k = |S|$, and the integer l are parameters. The problem is to find a strict temporal walk, X which satisfies the following conditions:*

1. *X covers (traverses through) all the vertices in S.*
2. *Each connected component of $X \setminus S$ is of size $\leq l$*

Foremost S-Covering walk requires us to find a S-Covering walk which ends the earliest. i.e, X is a Foremost S-Covering walk if X is a S-Covering walk and,

3. *There is no other S -Covering walk in the graph which ends earlier than X.*

In the definition, Condition 2 describes how we have parameterized the problem using the parameter l. $X \setminus S$ describes the set of vertices of the walk outside S. We use the parameter l to restrict the walk length outside S, thereby preventing the complexity from increasing significantly. Since the size of S is bound by k, all paths and walks in S can be computed in $f(k)$ time. However, there is no limitation on walks or paths outside S, and these can be arbitrarily large. This is why we use parameter l.

We can also make the following observations about the solution X:

1. X begins and ends in S.
2. Every connected component of X outside of S is a path.

These are not difficult to observe. However, let us prove them for the sake of completeness.

Lemma 5.3.1. *Given a solution X for the problem in Definition 5.3.1, X either satifies the following conditions, or we can find a valid solution X' for the instance of the problem which satisfies them.*

1. *X begins and ends in S.*
2. *Every connected component of $X \setminus S$ is a path.*

Proof. Given the solution X if is satisfies conditions 1 and 2, we are done. Otherwise, let us apply a few modifications to it to obtain X' which is also a solution. First of all, notice that X has to end in S. Otherwise, we can remove all the vertices after X visits a vertex in S for the last time to obtain a solution which ends before X. This is a contradiction. Thus, assume that X begins from a vertex outside S. And let v be the first vertex of S visited by X. In this scenario, we may eliminate all the vertices in X before v to obtain another solution X'. Thus, Condition 1 holds true.
Assume that there is a connected component C of X outside S which is not a path. Since we have established that the solution begins and ends in S, and C becomes disjoint from X upon removal of vertices in S. This means that vertices before and after C in X are in S. Thus, C can only be walk if it not a path. Let C be a walk, where without loss of generality, a vertex u appears more than once, say at (u, t_1) and (u, t_2). We can now eliminate all the vertices between these two appearances, and stay at vertex u from t_1 to t_2. We can do this repeatedly till no vertex appears more than once. i.e, we obtain a path. Let's call it P. Now, we claim that replacing C with P in the graph gives an acceptable solution, X_p. Assume that the claim is wrong. Then, our new solution violates one of the three conditions of Definition 5.3.1. This can't be condition 3, as the modification we have performed do not affect the starting or ending point of the walk. Since $V(P) \subset V(C)$, replacing C with P only reduces $|X \setminus S)|$. This means that if X_p violates condition 2, so does X. This is a contradiction. Only condition 1 is left. However, we do not remove any vertex in S from X to obtain X_p. Thus, the new solution is valid. □

So far, we have been working with two different components of the solution X, the part within S, and the one outside. We can compute all walks within S without violating the *fixed-parameter tractability* of the algorithm. We need to bound what we compute outside S. However, we only need to compute paths of length at most l in this scenario. We can use colour coding technique for this. This is intuition behind the following algorithm. Note that a *sub-colourful path* of C, or a *C-sub-colourful path* is a path where each vertex of the path is assigned a unique colour from C, but it is not necessary for all the colours of C to be assigned. Obviously, the number of vertices in the path $\leq |C|$ if such a colouring is possible.

Theorem 5.3.2. *There exists a randomized algorithm which accepts a temporal graph $(V, E, \boldsymbol{L})$, a set of vertices $S \subset V$ such that $|S| \leq k$, and an integer $l > 0$, to solve the S-*Covering walk *problem. If a solution exists, the algorithm finds it with a constant probability. It runs in $\mathcal{O}(e^{l.k^2} 2^{l+k} (n|\boldsymbol{L}|)^3)$ time.*

Proof. Let $H = V(G) \setminus S$. We will use dynamic programming to solve this problem. Let X be a potential solution to this problem. Consider some edge (u, v, t) in X. Here, vertices u and v may or may not belong in S. There are 4 possible options for this edge. We need to find a walk that moves between S and H. Let F be a set of vertices, such that $F \subseteq S$, let χ be a colouring of V, and let $C \subseteq [1, .., l]$. Define $\text{COV\_WALK}_\chi(F, C, v, t)$ as follows:

$$\text{COV\_WALK}_\chi(F, C, v, t) = \begin{cases} True, & \text{if } v \in S \text{ and there is a walk ending at } v \text{ at time } t, \text{ which covers all the vertices in } F \text{ with each of its components in } H \text{ being a sub-colourful path of colours } [1, .., l] \\ True, & \text{if } v \notin S, \text{ and there is a walk ending at } v \text{ at time } t \text{ which covers all the vertices in } F, \text{ with each of its components in } H \text{ being a sub-colourful path of colours } [1, .., l], \text{ and the component containing } (v, t) \text{ is C-sub-colourful on the colouring } \chi \\ False, & \text{otherwise} \end{cases} \tag{5.2}$$

This definition might come off as contrived at the first glance. But notice that if $v \in S$, COV_WALK describes the problem stated in Definition 5.3.1. If $v \notin S$, we need to first move from V to S only using the colours from C. Once we do that, $v \in S$ and we come back to our main problem. Also, $\text{COV\_WALK}_\chi(\emptyset, C, v, t)$ is $True$, as long as $v \in S$ and $t > 0$. Since we have shown that any solution begins and ends in S, we only need to calculate $\bigvee_{v \in S} \text{COV\_WALK}_\chi(S, [1, .., l], v, \tau)$. The recurrence relation for this function is as follows:

$\text{COV\_WALK}_\chi(F, C, v, t) =$
if $v \in S$:

$$\begin{cases} \bigvee\limits_{\substack{t'<t, \\ (u,v)\in E_{t'}(G), \\ u \in S}} \text{COV\_WALK}_\chi(F \setminus \{v\}, [1, .., l], u, t') \bigvee\limits_{\substack{t'<t, \\ (u,v)\in E_{t'}(G), \\ u \notin S}} \text{COV\_WALK}_{\chi'}(F \setminus \{v\}, [1, .., l], u, t') \end{cases} \tag{5.3}$$

if $v \notin S$ and $\chi\{v\} \in C$:

$$\begin{cases} \bigvee_{\substack{t'<t,\\ (u,v)\in E_{t'}(G),\\ u\notin S}} \text{COV\_WALK}_\chi F, C \setminus \chi\{v\}, u, t') & \bigvee_{\substack{t'<t,\\ (u,v)\in E_{t'}(G),\\ u\in S}} \text{COV\_WALK}_\chi F \setminus \{v\}, [1,..,l], u, t') \end{cases} \tag{5.4}$$

In an instance of $\text{COV\_WALK}_\chi(F, C, v, t)$, we have two possibilities; $v \in S$ and $v \in H$. If $v \in S$, then we need to find a walk as described by equation 5.2. Assume that the vertices of H are coloured with l colours uniformly at random. Notice that in this case, v needs to have a neighbour at a time $t' < t$, since we are looking for a strict path. Moreover, this neighbor could be in S or S'. Since we only need to make sure that we visit each vertex of S at least once, we may now look for a walk which only needs to cover $F \setminus \{v\}$. If the neighbour u of v lies in S, we have a new instance of the same problem. Otherwise, if $u \in H$, then we need to make sure that the segment of the walk which lies in H and contains u is a sub-colourful walk of $[1, ..l]$. This is possible in two ways; if u has a neighour w in S at $t' < t$, we can compute $\text{COV\_WALK}_\chi(F, C, w, t')$, and combine with w and the remaining path. Otherwise, consider neighbours of u in H at $t' < t$. There should be a a walk covering all the vertices in $[1, .., l] \setminus \chi\{u\}$, with each of its disconnected component in H being a sub-colourful path, and the disconnected component containing u should be C-sub-colourful. This is the relation when $u, v \in H$. Thus, the recurrence relation mentioned above is consistent with its definition. Moreover, the colouring of the vertices with l colours only matter when we consider a vertex $v \in H$. If $v \in S$, we do not consider the colouring of the graph in any way. This allows us to colour all the vertices in H uniformly at random again when $v \in S$ and $u \in H$. This new colouring is denoted as χ'. Since we only re-colour when $v \in S$, colouring is maintained and COV_WALK remains consistent with its definition.

The algorithm takes the input instance, and colours the vertices of H with l colours uniformly at random. Then we compute $R = \bigvee_{v\in S} \text{COV\_WALK}_\chi(S, [1, .., l], v, \tau)$.

Now, let's calculate the bound on the probability and compute the runtime of this algorithm. Consider a arbitrary solution X. The algorithm will detect X if every connected component in $X \setminus S$ is coloured in a colourful manner. We know, by definition that the length of every segment is less than l. We can now use the Lemma 4.1.1, to show that every connected component of $X \setminus S$ will be coloured in a colourful manner with probability e^{-l}. However, this applies only for one component of the walk outside S. Suppose there are m

such components, then we will succeed only if all of them are coloured colourfully. We colour each component independently during the process of detection. Thus, we can multiply the probabilities to obtain the final success probability to be at least $(e^{-l})^m$. If we use Lemma 5.2.1, we see that there can be at most k^2 vertices of S in X. This implies that there can be at most $k^2/2$ components outside S. Thus, $m \leq k^2$ and $(e^{-l})^m \geq e^{l.k^2}$. Thus, the overall chance of success is $e^{l.k^2}$. Observe that if re-colouring is not carried out, it is not possible to multiply probabilities in the manner we have done because of dependence between them.

There are $(2^l)*(2^k)*n^2*|\boldsymbol{L}|$ instances of COV_WALK(F, C, v, t) possible, and computation of neighbours and comparison of colours at each stage will take at most $2 * n * |\boldsymbol{L}|$ time. Additionally, at most $|V| * k * |\boldsymbol{L}|)$ colourings needs to be computed. However, this is polynomial in complexity. We need to repeat the algorithm $e^{l.k^2}$ to obtain a constant success probability. Thus, the total time complexity is at most $\mathcal{O}(e^{l.k^2}2^{l+k}(n|\boldsymbol{L}|)^3)$. □

To find a foremost S-covering walk, we restrict the temporal graph input to first $k-1$ time stamps, and run the algorithm in Theorem 5.3.2. If we find a path, we are done as no shorter path can exist. If not, we take first k time stamps, and repeat. The shortest time index when we find a walk we be returned as a solution. This is the randomized algorithm we require. A python implementation of the algorithm mentioned in Theorem 5.3.2 was constructed. However, the code is too long to be incorporated into this text. It can be produced on demand.

We have proposed an algorithm which uses parameterization by l to tackle Foremost S-Covering Walk. While this works, it requires a new parameter and has a large time complexity. Runtime of this algorithm can be significantly improved if we could substitute colour-coding with a more efficient method. Attempts were made at doing this. No complete results were obtained. Thus, this point will be briefly discussed.
Literature provides us with a few algorithms to compute Foremost (u, t)-Paths in polynomial time. This means that given a vertex u and time point t, it is possible to compute the foremost paths from (u, t) to every other vertex in polynomial time.
More precisely,

Theorem 5.3.3. *Let $(V, E, \boldsymbol{L})$ be a temporal graph, $u \in S$, and let $t \in \boldsymbol{L}(w)$ for some $w \in V$. There is an algorithm which computes for all $v \in V \setminus \{u\}$, a foremost strict temporal walk*

from (u, v) *in time* $(|L| * |V|)$.

The proof of this can be found in [10] and [7]. Notice that with slight modifications to this algorithm, we can find the *Greatest (u, t)-starting point* for a vertex v. This is the largest time index t' such that a $(v, t') - (u, t)$ path exists. To accomplish this, one only needs to *invert* the temporal graph and run the FOREMOST (u, t)-PATH algorithm. Let τ be the largest time index of G. The inverted temporal graph, $\hat{G}$ has the same vertices as G, and for each edge $(\{u, v\}, t)$ of G, has an edge $(\{u, v\}, \tau - t)$.
Now, consider S_e to be the set of all temporal edges within S. Let H be the graph obtained by removing S_e from G. H has no edges between any two vertices of S. What we may now do is use dynamic programming to compute all the paths within S. But at each time point, we also calculate Greatest (u, t)-starting point for all vertices in S, in the temporal graph H. In this manner, we can identify all the walks involving vertices of S, even when they involve vertices .
The exact formulation of this idea is still a work in progress. But there is a possibility of obtaining an FPT algorithm this way.

Chapter 6

Literature on Temporal Graphs

6.1 Introduction

This chapter outlines a limited amount of recent work on temporal graphs, without going too extensively into details. The only problems explicitly stated to be parameterized in nature relating to Temporal graphs seems to be on temporal separators. However, the temporal analogues to polynomial time problem on static graphs are very often NP-hard. Thus, there seem to a lot of problems which can be parameterized and explored under various constrains.

6.2 Definitions and notations:

In addition to the concepts discussed in chapter 2, the present academic literature uses many other concepts. The following are some of the essential definitions used:
Temporal Connectivity A temporal graph is said to be connected if there exists a strict temporal path from any vertex to any other vertex. An analogous version for non-strict paths also exists, which we shall denote as non-strictly connected.
Temporal r-connectivity: A temporal graph is said to be r-connected if there is a strict temporal path from the r vertex to any other vertex. The analogue for non-strict paths shall be called, somewhat unimaginatively, as non-strictly r-connected. We shall use the notation E_t to denote the set of edges with timestamps t.

Cost of labeling: Let (V, E, $\mathbf{L}$) be a temporal graph. Then, its cost of labeling is defined as $c(L) = \sum_{e \in E} |\mathbf{L}_e|$.
Simple temporal graph: A simple temporal graph is one where each edge has a label of size 1. i.e, any temporal edge exists at only one time point.
Periodic Temporal graphs: These are temporal graphs where each edge repeats itself periodically. The graph is called homogeneous if all edges have the same period and heterogeneous otherwise.
All paths property: A temporal graph is said to have the all-paths property if for every simple path in its underlaying graph, there is at least one corresponding strict temporal path in the temporal graph. This means that there is a strict temporal path involving all the vertices in the simple path in the corresponding order.
reach(G): A temporal graph G is called said to satisfy/have the reach property if there is (u, v)-temporal walk for every pair (u, v) such that v is reachable from u in the underlying graph.

6.3 Explored Problems:

6.3.1 Minimum temporal connectivity

Temporal connectivity is a crucial property of temporal graphs and one of the most explored areas in the literature. The fundamental network (design) problem may stated as follows: *Given an underlaying (di)graph G, assign labels to the edges of G so that the resulting temporal graph* $(V, E, \boldsymbol{L})$ *minimizes some parameter while satisfying some connectivity property [10].* The following are some solved problems relating to connectivity:

Checking if a graph is temporally connected: There exists a *low* polynomial time algorithm to check if a given graph is temporally connected or not [1].

***Tight bound on edges for connected graphs*:** Determine a function $c(n)$ such that any temporally connected graph on n vertices have a connectivity certificate with at most $c(n)$ edges.

The result is $c(n) \in \Theta(n^2)$ [4]

***r-MTC*:** Find the minimum weight subset of temporal edges that preserve connectivity of a given temporal graph from a given vertex r.
In this problem, the input is a temporal graph, with a weight function $w(e, t)$ which assigns a weight to each edge at each time point it exists. The same edge might have different weights at different time points. Weight of a subset of temporal edges is the sum of weights of all the edges in that subset. It is shown that this problem cannot be approximated within a ratio of $\mathcal{O}(log^{2-\epsilon}n)$ for any constant $\epsilon > 0$, unless $NP \subseteq \text{ZTIME}(n^{poly\ logn})$. Using a transfromation from r-MTC to Directed Steiner Tree, Kyriakos Axiotis et al. obtains a polynomial time $\mathcal{O}(n^{\epsilon})$-approximation, for any constant $\epsilon > 0$. The problem can be solved in polynomial time if the underlaying graph has bounded treewidth [4].

***MTC with weights (Minimum Temporal Connectivity with weights)*:** Find a minimum weight subset of temporal edges that preserve connectivity of a given temporal graph between all pairs of vertices in the graph.
It can solved in polynomial time, if the underlying graph is a tree, and 2-aproximable if the underlying graph is a cycle. There is a $\mathcal{O}(n^{1+\epsilon})$ approximation for MTC with weights as well as a polynomial time $\mathcal{O}((\Delta M)^{2/3+\epsilon})$-approximation, where M is the number of temporal edges and Δ is the maximum degree of the underlying graph. All of the above are based on non-strict paths, but can be extended to strict paths with small changes in approximation guarantees and running times. Note the importance of weights in this problem. The version without weights is given below [4].

***MTC*:** What is the largest number of labels which can be removed from a graph while preserving temporal connectivity?
This problem is shown to be APX-hard [1, 10], which means there is no PTAS unless P=NP. A related result is that in a complete or random graph, if the labels are also assigned at random, all but $\mathcal{O}(n)$ can be removed while maintaining connectivity.

***Minimal designs of non-linear cost*:** Are there simple temporal graphs with each edge having a differing time label such that no edge is redundant and the cost of the graph

is non-linear(in the number of vertices)?
The question asks if there is an edge which can be removed without affecting the temporal connectivity for a temporal graph which is given to be connected and have non-linear cost, in the number of vertices. The answer is no[1]. The paper shows this by an example of a complete graph on n vertices.

6.3.2 Temporal Exploration

Temporal exploration is another well-researched topic relating to temporal graphs. The notion of path changes considerably from static to temporal graphs and this has demanded for a reformulation of the associated theory. While graph exploration is well developed for static graphs, it is still in the developing stage for temporal graphs. Some of the related problems are discussed below:

***Random Exploration of temporal graph*:**
A random walk on a graph is a process of visiting the vertices randomly. It may be described as an agent starting at vertex and at each step, moves to a neighbor of the current vertex at random. The walk is called a simple random walk on a graph if the next vertex is chosen from the set of neighboring vertices uniformly at random. For temporal graphs, exactly one movement happens at each time index.The cover time of a graph is the expected time (say, in the number of steps) required to visit all the vertices in the graph. This is known be $\mathcal{O}(mn)$ in static graphs where n is the number of vertices and m is the number of edges, owing to Aleliunas et al. Temporal graphs have a cover time which is exponential in the number of vertices [3], when explored by a simple random walk. In fact, even if we loosen the condition on strict paths and allow $n^{1-\epsilon}$ steps per layer, the cover time still remains exponential. However, a lazy random walk can explore a temporal graph in polynomial time, unlike a simple random walk. A lazy random walk is where the walk moves to a neighboring vertex with probability $1/\Delta$, where Δ is the maximum degree of any vertex in the graph, and stays at the vertex with the remaining probability.

***Exploration of Temporal graph on periodic graphs*:** The problem is defined as visiting all the vertices (or nodes) and exiting the system in finite time by a single entity (agent) with certain properties [6]

This problem can be explored under various constrains and characteristics of the graph. This version is restricted to the case where the temporal graph is periodic in nature. The first result related to the graph is that if the nodes have no identifiers, the exploration is unsolvable if the agent has no knowledge of the largest period of any edge in the graph(p). If the vertices are identifiable, either the number of vertices or system period must be known for the problem to be solvable. The lower bound for exploration is $\Omega(|E|.p)$ for homogeneous graphs and $\Omega(|E|.p^2)$ for heterogeneous graphs, in general case. Other lower bounds have been established as well, in [6].

***Temporal Exploration with the fastest arrival time(TEXP)*:** An unweighted temporal graph is given as the input. The problem is find a strict temporal walk which covers all the vertices and ends the earliest.
The decision version of exploring a temporal graph is NP-complete [11]. There also exists some constant c such that TEXP cannot be approximated within cn unless P=NP.

6.3.3 Path and Separator Problems

Path problems are another fundamental category of problems on Temporal graphs. The idea of temporal path is as old as the idea of temporal graphs and questions relating to temporal paths were among the first asked on the subject. The writer's first problem was also on the same topic.

***Finding the foremost journey*:** Given a temporal graph, a time index t which is present in the graph, and an ordered pair of vertices (s, z), is it possible to compute presence of the foremost journey in the graph?
There is an algorithm which runs in polynomial time which computes the (s, z) foremost journey which starts at t. [10].

***Menger's Theorem*:** Is there an analogue of Menger's theorem for general temporal graphs.
George Mertzios et al. has proved an analogue of Menger's theorem for general temporal

graphs [10]. It is stated as follows: Consider a temporal graph *(V, E,* $\boldsymbol{L}$*)*, with an ordered pair of vertices (s, z). The maximum number of out-disjoint journeys from s to z is equal to the maximum number of node departure time needed to separate s from z.

***Node disjoint s-t Paths*:** Given a temporal graph and ordered pair of vertices (s, t) calculate the number of node-disjoint (s, t) paths.
While this problem is solvable in polynomial time for static graphs, it is NP-complete for temporal graphs w.r.t strict temporal paths [8].

***Temporal separator of bounded size*:** Given a temporal graph, an ordered pair of vertices (s, t) and an integer k, calculate if there exists a temporal (s, t)-separator of size at most k.
An (s, t)-temporal separator is a set of vertices such that every path from s to t passes through at least one vertex in the separator. This problem is shown to be NP-hard for temporal graphs, in general. [13]. This is valid for both strict and non-strict paths.

***Traveling salesman problem for costs one and two (TTSP(1, 2))*:** The input is a temporal graph *(V,* $\boldsymbol{E}$*,* τ*)*, such that each layer is complete and W, where W is a weighing function. It assigns a weight of 1 or 2 to every edge, respecting time. i.e, the same edge can have different weights at different time points. Each layer of the graph should be complete. Find a strict temporal cycle (Strict path that starts and ends at the same vertex) of minimum weight in the path.
This problem is **APX**-hard as Asymmetric TSP(1, 2) is a special case of TTSP(1, 2) [12]. There is a $1.7 + \epsilon$ approximation algorithm for the problem.

***Temporal Path Packing*:** Given a temporal graph, find time and node disjoint strict paths maximizing the number of temporal edges used. Time disjoint means that it is required for them to correspond to distinct intervals which differ by more that one in time.
There is a $\frac{1}{\frac{7}{2}+\epsilon}$-factor approximation algorithm for TPP when the lifetime of the graph (largest time index of any edge in the graph - smallest time index of any edge in the graph $+ 1) = n$.

As we can see, most of the problems are confined into a few categories, and complexity of analogous problems to static graphs are significantly higher.

Chapter 7

Conclusion

Temporal graphs are versatile objects with the capability of encoding dynamic, temporal data in addition to graph data. This comes at a cost, and temporal graph algorithms are likely to be more complex to describe and more time consuming to run. This project explores a few of these problems from a parameterized viewpoint.

We obtained a few results for the first problem of this project, k-path problem in temporal graphs. We have obtained an FPT algorithm for solving this problem in time $\approx 5.44^k(n\tau)^{\mathcal{O}(1)}$ by colour coding technique. We also propose another algorithm which can solve the k-path problem in $4^{k+o(k)}(n\tau)^{\mathcal{O}(1)}$.

Afterwards, literature on temporal graphs was explored for finding another viable problem. It was found that the literature on temporal graphs from a parameterized algorithms perspective is quite limited. Furthermore, the open problems that could be found were unsuitable for the project. Thus, a few new problems were defined based on problems from existing literature.

The first problem is called *Foremost k-covering walk*. Given a temporal graph G and an integer k, it requires to find the foremost temporal walk which 'covers' at least k vertices of G. The second problem is a more restricted version of the same, where the input contains a temporal graph G and a subset S of vertices of G. S has at most k elements. We need to find the foremost temporal walk which covers S. This problem is called *Foremost S-covering walk*. We found an algorithm which can solve *Foremost S-covering walk* with an extra parameter. This algorithm relies on a bound on walk length we have found. This is another parameterization of the same problem, and can solved in $\mathcal{O}(e^{l.k^2}2^{l+k}(n|\boldsymbol{L}|)^3)$ time.

Thus, both *Foremost S-covering walk* and *Foremost k-covering walk* parameterized by k remains open for further research.

Bibliography

[1] Eleni Akrida, Leszek Gasieniec, George Mertzios, and Paul Spirakis. On temporally connected graphs of small cost. In *Approximation and Online Algorithms*, pages 84–96, Cham, 2015. Springer International Publishing.

[2] Noga Alon, Raphy Yuster, and Uri Zwick. Color-coding: A new method for finding simple paths, cycles and other small subgraphs within large graphs. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 326–335, New York, NY, USA, 1994. ACM.

[3] Chen Avin, Michal Koucký, and Zvi Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 121–132, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Kyriakos Axiotis and Dimitris Fotakis. On the size and the approximability of minimum temporally connected subgraphs. *CoRR*, abs/1602.06411, 2016.

[5] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer International Publishing, Cham, 2015.

[6] Paola Flocchini, Bernard Mans, and Nicola Santoro. On the exploration of time-varying networks, 1 2013.

[7] Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, abs/1401.1919, 2014.

[8] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820 – 842, 2002.

[9] Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007 – 1023, 2009.

[10] George B. Mertzios, Othon Michail, and Paul G. Spirakis. Temporal network optimization subject to connectivity constraints. Jul 2018.

[11] Othon Michail and Paul G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016.

[12] Christos H. Papadimitriou and Mihalis Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, 18(1):1–11, 1993.

[13] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The computational complexity of finding separators in temporal graphs. *CoRR*, abs/1711.00963, 2017.