TÉCNICO LISBOA

# Computing Geodesic Tree Distances with the GTP Algorithm

**Miguel Caires**

Thesis to obtain the Master of Science Degree in

## Mathematics and Applications

Supervisor: Prof. Alexandre Francisco

Co-Supervisor: Prof. Miguel Dionísio

### Examination Committee

Chairperson: Prof. Paulo Mateus

Supervisor: Prof. Alexandre Francisco

Members of the Committee: Prof. Luís Russo

Prof. Alexandre Francisco

**December 2021**

# Acknowledgments

I am very thankful for the dedication and availability of Professor Alexandre Francisco and Professor Miguel Dionísio in providing guidance and support vital to this work. I would also like to express my gratitude towards the University of Lisbon and Instituto Superior Técnico for providing an environment to learn invaluable skills without which this thesis would not be possible. My sincerest thanks goes to all the teachers that accompanied my academic journey and helped me develop those skills. They deserve enormous praise for the work they do. Finally, I cannot overstate my gratitude towards friends and family for the innumerable ways in which they have helped me.

# Abstract

The geometry of tree space $\mathcal{T}_n$ introduced by Billera, Holmes, and Vogtmann provides an effective way of comparing phylogenetic trees in the form of the geodesic distance, i.e. the length of the shortest path from one tree to another. We study and implement the first polynomial-time algorithm for finding geodesic paths in tree space $\mathcal{T}_n$, proposed by Owen and Provan. We focus on other graph problems and algorithms related to the Geodesic Treepath Problem, namely the maximum flow problem. Our results show that the geodesic distance can be efficiently computed in practice with the correct choice of algorithms and data structures, confirming the theoretical results we derived.

# Keywords

Geodesic distance, phylogenetic trees, Geodesic Treepath Problem, Complexity, Graph Theory

# Resumo

A geometria do espaço de árvores $\mathcal{T}_n$ introduzida por Billera, Holmes e Vogtmann possibilita a comparação de árvores filogenéticas através da distância geodésica, i.e. o comprimento do caminho mais curto entre duas árvores. Nesta tese estudamos e implementamos o primeiro algoritmo de tempo polinomial para encontrar geodésicas em $\mathcal{T}_n$, que foi proposto por Owen e Provan. Estudamos outros problemas e algoritmos relacionados com o problema de encontrar geodésicas em $\mathcal{T}_n$ (o *Geodesic Treepath Problem*), nomeadamente no problema do fluxo máximo numa rede. Os nossos resultados demonstram que a distância geodésica pode ser eficientemente calculada em tempo útil com as escolhas corretas de algoritmos e estruturas de dados, confirmando a nossa análise teórica.

# Palavras Chave

Distância geodésica, Árvore filogenética, Geodesic Treepath Problem, Complexidade, Teoria de Grafos

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Listings

# 1

# Introduction

## 1.1 Motivation

Phylogenetic trees are trees which depict evolutionary relationships between entities, which are typically biological species or strains. These trees are often constructed by algorithms which compare DNA sequences from selected parts of the genome using some distance measure (e.g.: the Hamming distance). However, the resulting trees change depending not only on the selection of genes or coding regions for the DNA sequences but also on the choice of the tree-building algorithm. From this uncertainty arises the need for comparing phylogenetic trees, and to this effect several measures have been proposed [1]. The geometry of tree spaces proposed by Billera et al. [2] indeed provides us with one such distance measure, which is the length of a geodesic path in a defined tree space $\mathcal{T}_n$, a quantity that shall henceforth be called geodesic distance. The geodesic distance seems to be the most appropriate quantitative comparison, since it incorporates aspects of tree topology and numerical edge lengths in a single measure, whereas other measures often lose information by focusing exclusively on tree topology and cannot be computed efficiently.

## 1.2 Problem

The Geodesic Treepath Problem (GTP) is the problem of finding the geodesic path between two trees in tree space $\mathcal{T}_n$. The first polynomial-time algorithm able to solve this problem was presented by Owen and Provan [3], seeing as two previous algorithms by Owen [4] and by Kupczok et al. [5], had exponential time complexity. In this thesis we intend to examine and implement this polynomial-time algorithm, to be referred to as the GTP algorithm. We will also perform experimental analysis of the time and space complexity of different versions of the algorithm, and verify whether our implementation is consistent with the theoretical analysis.

## 1.3 Organization

This thesis is divided into 5 chapters, including the current introductory chapter. The second chapter aims to provide the necessary knowledge to understand the workings of the GTP algorithm, focusing on the maximum-flow problem and its relevance to the problem of finding geodesic paths in $\mathcal{T}_n$. Chapter 3 presents the GTP algorithm itself and discusses its time complexity as well as the concrete choices made in our Python implementation to ensure consistency with the theoretical algorithm. Experimental analysis of this implementation was done in Chapter 4 alongside other useful simulations which intend

to interpret and explain the results. The concluding chapter sums up the results obtained and offers suggestions for further research.

# 2

# Background

In this chapter we outline the main characteristics of tree space $\mathcal{T}_n$, and the conditions which describe geodesic paths. It also includes further explanation on how these conditions relate to solving the minimum weight vertex cover problem in bipartite graphs, which itself is efficiently computed by performing a reduction to the maximum flow problem. We refer to [2] for the detailed construction of tree space $\mathcal{T}_n$ and the study of its properties.

## 2.1 Tree Space and Geodesic Paths

Let $\mathcal{T}_n$ be the set of trees which have exactly $n$ leaves. Any tree $T \in \mathcal{T}_n$ has at maximum $n-2$ internal edges, i.e. edges whose endpoints are not leaves. Therefore, a given tree with variable internal edge lengths can be represented as a point in $n-2$ dimensional space with positive coordinates, which we call an orthant of $\mathbf{R}^{n-2}$.

**Definition 1.** *Given an edge $e$ belonging to a tree $T$ a split $\sigma_e = X_e|\overline{X_e}$ is defined as a partition of the tree's leafset into two disjoint subsets $X_e$ and its complement $\overline{X_e}$, resulting from the removal of edge $e$ from $T$. Given two edges $e, f$ (from the same or from two different trees with $n$ leaves each), the splits $\sigma_e$ and $\sigma_f$ are said to be compatible if at least one of the following*

$$X_e \cap X_f, X_e \cap \overline{X_f}, \overline{X_e} \cap X_f, \overline{X_e} \cap \overline{X_f}$$

*is the empty set. Let $\mathcal{X}, \mathcal{Y}$ be distinct sets of splits. $\mathcal{X}$ is said to be a compatible set of splits if any two splits in $\mathcal{X}$ are compatible. $\mathcal{X}$ is said to be compatible with $\mathcal{Y}$ if $x$ is compatible with $y$ for any $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.*

It can be shown that there are exactly $(2n-3)!!$ non-identical trees in $\mathcal{T}_n$, [1, 6] non-identical trees being trees which do not have the exact same set of splits.

$\mathcal{T}_n$ is path-connected [2], which means that through continuous contraction and expansion of these unique edges one can transform any $T_1 \in \mathcal{T}_n$ into a different $T_2 \in \mathcal{T}_n$, along a continuous path $\Gamma = \{T(\lambda) \in \mathcal{T}_n : 0 \leq \lambda \leq 1\}$. For any given tree $T \in \mathcal{T}_n$ the set of its $n-2$ splits is compatible, and any set of $n-2$ compatible splits (of a leafset of $n$ elements) defines a valid tree. In other words, two splits are compatible if and only if they can coexist in the same tree. If two trees each have an internal edge corresponding to the same split we say that this edge is common between the two trees. If there are no common edges we say the trees are disjoint. Geometrically, the tree space $\mathcal{T}_n$ can be seen as a collection of $(2n-3)!!$ orthants of dimension $n-2$.

**Example 1.** *Given $T_1, T_2 \in \mathcal{T}_n$, the cone path between these trees corresponds to uniformly contracting*

*all edges in $T_1$ until their lengths are zero and then uniformly expanding them until arriving at tree $T_2$. This path may or may not be the geodesic path, as evidenced in Figure 2.1, adapted from [3].*

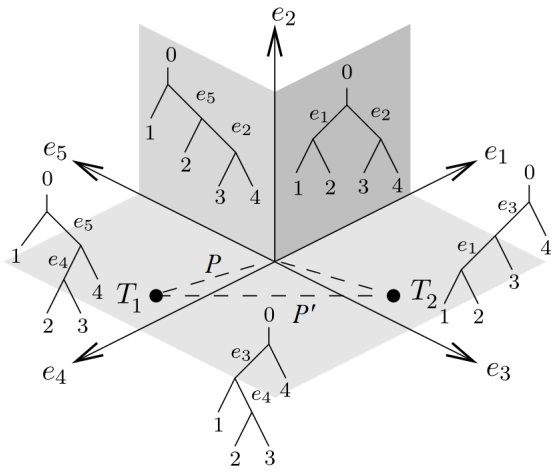



**Figure 2.1:** Embedding of $\mathcal{T}_4$ on $\mathbf{R}^3$. Path $P$ is the cone path between $T_1$ and $T_2$ while $P'$ is the geodesic path.

### 2.1.1 Geodesic Path properties

We now give a rundown of the conditions for determining a geodesic path between two trees. These conditions were introduced and explained in [4] by formulating the problem of finding the geodesic path as a touring problem, i.e. a problem of finding a shortest path in Euclidean space that visits an ordered sequence of regions. They were subsequently summarized in [3], where the Extension Problem and its formulation as a bipartite graph problem were first made known. We start by defining the first of the three conditions, and by formalizing the notion of a path space.

**Definition 2.** *Given $T = (L, \mathcal{E}), T' = (L', \mathcal{E}') \in \mathcal{T}_n$ with disjoint set of splits and $\mathcal{A} = (A_1, ..., A_k)$ and $\mathcal{B} = (B_1, ..., B_k)$ partitions of $\mathcal{E}$ and $\mathcal{E}'$ respectively, assume that*

***(P1)** For each $i > j$, $A_i$ and $B_j$ are compatible sets.*

*Then $B_1 \cup ... \cup B_i \cup A_{i+1} \cup A_k$ is a compatible set for all $1 \leq i \leq k$, and therefore defines a tree $T_i$ belonging to the orthant generated by this set, denoted by $\mathcal{O}_i = \mathcal{O}(B_1 \cup ... \cup B_i \cup A_{i+1} \cup ... \cup A_k)$. $\mathcal{P} = \cup_{i=1}^{k} \mathcal{O}_i$ forms a connected space and we call $\mathcal{P}$ a path space with support $(\mathcal{A}, \mathcal{B})$. The shortest path from $T$ to $T'$ in $\mathcal{P}$ is called a path space geodesic for $\mathcal{P}$.*

It is known that the geodesic in $\mathcal{T}_n$ between any two trees $T, T' \in \mathcal{T}_n$ is a path space geodesic

for some path space between $T$ and $T'$. [3] We now present a necessary condition for a path space geodesic to be the geodesic between $T$ and $T'$.

**Theorem 1.** *[3] Given $T = (L, \mathcal{E}), T' = (L', \mathcal{E}') \in \mathcal{T}_n$ and the geodesic $\Gamma$ between them, $\Gamma$ can be represented as a path space geodesic with support $\mathcal{A} = (A_1, ..., A_k)$ of $\mathcal{E}$ and $\mathcal{B} = (B_1, ..., B_k)$ of $\mathcal{E}'$ if $(\mathcal{A}, \mathcal{B})$ satisfies **(P1)** and the following additional property:*

$$\textbf{(P2)}\ \frac{||A_1||}{||B_1||} \leq \frac{||A_2||}{||B_2||} \leq ... \leq \frac{||A_k||}{||B_k||}.$$

*where $||A_i|| = \sqrt{\sum_{e \in A_i} |e|}$. A path space satisfying (P1) and (P2) is called a proper path space, and the respective path space geodesic is called a proper path.*

Notice that if any of the inequalities in (P2) is not strict the sets in question can be merged to form a new proper path, which means that there exists a support pair such that all inequalities are strict. The following theorem is needed in order to provide a necessary and sufficient set of conditions for a proper path to be a geodesic between two trees.

**Theorem 2.** *[3] A proper path $\Gamma$ from $T$ to $T'$ with support $(\mathcal{A}, \mathcal{B})$ satisfying (P1) and (P2) is a geodesic if and only if the following condition also applies:*

***(P3)** For each pair $(A_i, B_i)$ there is no partition $C_1 \cup C_2$ of $A_i$ and $D_1 \cup D_2$ of $B_i$ such that $C_1, C_2, D_1, D_2$ are all non-empty, $C_2$ is compatible with $D_1$ and $\frac{||C_1||}{||D_1||} < \frac{||C_2||}{||D_2||}$.*

### 2.1.2 The Extension Problem

Of the necessary and sufficient conditions for finding the geodesic path between two trees, (P3) suggests a procedure for iteratively improving upon a starting proper path, such as the cone path. The condition is only satisfied if for each support pair no such partition exists in the conditions described by (P3). In order to explain how this can be formulated as a graph problem, let us provide the definition of the incompatibility graph $G(A, B)$.

**Definition 3.** *The incompatibility graph $G(A, B)$ is a bipartite graph $G = (A \cup B, E)$ such that $A \subseteq \mathcal{E}$, $B \subseteq \mathcal{E}'$ correspond to node sets on the left and right sides of $G(A, B)$ respectively, and an edge $(a, b) \in E$ exists if $a \in A$ and $b \in B$ are incompatible edges, i.e. the splits they induce in their respective tree are incompatible.*

Recall also that an independent set in a graph $G = (V, E)$ is a set $U \subseteq V$ such that $(u_1, u_2) \notin E$ for any $u_1, u_2 \in U$. The Extension Problem restates (P3) as a problem to be solved in the context of the incompatibility graph.

**Definition 4.** *The Extension Problem in an incompatibility graph $G = (A \cup B, E)$ is to find non-trivial partitions $C_1 \cup C_2$ of $A$ and $D_1 \cup D_2$ of $B$ such that:*

1. *$C_2 \cup D_1$ corresponds to an independent set in $G(A, B)$;*

2. *$\frac{||C_1||}{||D_1||} < \frac{||C_2||}{||D_2||}$.*

Consider the following lemma, which will allow us to recast the Extension Problem as a minimum weight vertex cover problem.

**Lemma 1.** *A vertex cover of a graph $G = (V, E)$ is a set $C \subset V$ such that for every $e = (u, v) \in E$ either $u \in C$ or $v \in C$. In a graph $G = (V, E)$, $I \subseteq V$ is an independent set in $G$ if and only if $V \setminus I$ is a vertex cover of $G$.*

*Proof.* Let $I$ be an independent set in $G$. By definition, if $u \in I$ and $v \in I$ then it must be that $(u, v) \notin E$. The equivalent contrapositive is that if $(u, v) \in E$ then either $u \notin I$ or $v \notin I$ i.e., $u \in V \setminus I$ or $v \in V \setminus I$, meaning $V \setminus I$ is a vertex cover. □

If weights are assigned to the vertices, the minimum weight vertex cover problem is to determine a vertex cover with the smallest possible weight. The previous result illustrates the relationship between an independent set and a vertex cover in a graph, and it leads us to the following theorem (adapted from [3]), which reinterprets the Extension Problem as a minimum weight vertex cover problem.

**Theorem 3.** *Let weights be assigned to vertices in $G(A, B)$ according to the following:*

$$w_e = \begin{cases} \frac{|e|^2}{\|A\|^2} & \text{if } e \in A \\ \frac{|e|^2}{\|B\|^2} & \text{if } e \in B \end{cases}$$

*Then, a solution to the Extension Problem in $G(A, B)$ exists if and only if the minimum weight vertex cover has weight less than $1$.*

*Proof.* Consider the following equivalent expressions to the second condition in the definition of the

Extension Problem:

$$\begin{aligned}
\frac{||C_1||}{||D_1||} < \frac{||C_2||}{||D_2||} &\iff \frac{||C_1||/||A||}{||D_1||/||B||} < \frac{||C_2||/||A||}{||D_2||/||B||} \\
&\iff \frac{(||C_1||/||A||)^2}{(||D_1||/||B||)^2} < \frac{(||C_2||/||A||)^2}{(||D_2||/||B||)^2} \\
&\iff \frac{(||C_1||/||A||)^2}{1-(||D_2||/||B||)^2} < \frac{1-(||C_1||/||A||)^2}{(||D_2||/||B||)^2} \\
&\iff \frac{||C_1||^2}{||A||^2} + \frac{||D_2||^2}{||B||^2} < 1 \\
&\iff \sum_{e\in C_1} \frac{|e|^2}{||A||^2} + \sum_{e\in D_2} \frac{|e|^2}{||B||^2} < 1
\end{aligned}$$

Since $C_2 \cup D_1$ must be an independent set, the last expression simply states that the total weight of the vertex cover $C_1 \cup D_2$ is less than $1$. In other words, a solution to the problem exists if and only if the minimum weight vertex cover satisfies this requirement. □

## 2.2 Max flow, min-cut and vertex cover

We will now demonstrate how the minimum weight vertex cover problem in a bipartite graph can be solved by converting it into a maximum flow problem, with the crucial intermediate step supported by the max-flow min-cut theorem. We start by defining the maximum flow problem and its related concepts:

**Definition 5.** *Let $G = (V, E)$ be a directed network with $s, t \in V$ chosen to be source and sink nodes respectively. It can be assumed without loss of generality that $s$ has only outgoing edges and $t$ has only incoming edges. Define $c : V \times V \to \mathbf{R}_0^+$ as the capacity function. Let $c(u, v) = 0$ for any $(u, v) \notin E$. A flow $f : V \times V \to \mathbf{R}$ is a map such that:*

- $f(u, v) \leq c(u, v)$ *for all* $(u, v) \in E$;
- $\sum_{u:(u,v)\in E} f(u, v) = \sum_{u:(v,u)\in E}; f(v, u)$ *for any* $v \in V \setminus \{s, t\}$;
- $f(v, u) = -f(u, v)$ *for any* $(u, v) \in E$.

*The value of this flow is given by $|f| = \sum_{v:(s,v)\in E} f(s, v)$. The max-flow problem is to find a flow $g$ for the network $G$ such that $|g| \geq |f|$ for any other flow $f$. Given a flow $f$ in the network $G$ the residual network $G_f$ is defined as $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$ where the residual capacity $c_f(u, v)$ is given by $c_f(u, v) = c(u, v) - f(u, v)$ for any $(u, v) \in V \times V$.*

Note that the previous definition of the residual network $G_f$ makes it possible for $G_f$ to have edges which were not present in $G$, if these edges are the reverse edge of some edge in $G$. The notion of a

minimum capacity cut presented below will be needed in order to establish the relationship between the minimum weight vertex cover and the solution to the maximum flow problem.

**Definition 6.** *Given a directed graph $G = (V, E)$, $s, t \in V$ source and sink nodes respectively, capacity $c : E \to \mathbf{R}_0^+$ and $C \subset V$ we define an $s - t$ cut $(S, T)$ to be a partition of $V$ such that $s \in S$ and $t \in T$. The capacity of this cut is given by*

$$c(S, T) = \sum \{c(u, v) : (u, v) \in E, u \in S, v \in T\};$$

*Let $X_S \subset E$ be the edge set induced by the cut $(S, V \setminus S)$ i.e:*

$$X_S = ((S \times (V \setminus S)) \cup ((V \setminus S) \times S)) \cap E;$$

*The min-cut problem is to find an $s - t$ cut $(S, T)$ such that $c(S, T) \leq c(S', T')$ for any other $s - t$ cut $(S', T')$.*

For introducing the max-flow min-cut theorem we now give our proof of a statement relating the value of a given flow in a network $G = (V, E)$ to the flow of a certain cut in the same network.

**Lemma 2.** *Define the flow of a cut $(S, T)$ of $V$ to be:*

$$f(S, T) = \sum_{(x,y) \in (S \times T) \cap E} f(x, y) - \sum_{(y,x) \in (T \times S) \cap E} f(x, y)$$

*If $f$ is any flow in graph $G = (V, E)$ with value $|f|$ then $f(S, V \setminus S) = |f|$, for any cut $(S, V \setminus S)$ of $V$.*

*Proof.* If $S = \{s\}$ then the statement is obvious from the definition of $|f|$. We now proceed by induction on $S$. Assuming the statement is true for any cut $(A, V \setminus A)$ such that $|A| \leq k$, pick some set $S$ with $|S| = k$. Let $S' \subset V$ be such that $|S'| = k + 1$ and $S' = S \cup v$ for some $v \in V$. We now only need to consider the flow contribution of outgoing and ingoing edges from/to $v$. Notice that to obtain $f(S', V \setminus S')$ outgoing edges flow is added to $f(S, V \setminus S)$ and ingoing edges flow is subtracted from $f(S, V \setminus S)$. By conservation of flow this means that the flow of the cut remains unchanged, and using the induction hypothesis we get $f(S', V \setminus S') = f(S, V \setminus S) = |f|$. □

An adapted proof of the max-flow min-cut theorem is given ahead, for an alternative proof please see Section 6.5 of [7]. Having provided that proof it will finally be possible to demonstrate the relationship between the maximum flow problem and the minimum weight vertex cover problem.

**Theorem 4** (max-flow min-cut theorem)**.** *The maximum value of an $s - t$ flow is equal to the minimum capacity over all $s - t$ cuts.*

*Proof.* Suppose $f$ is the maximum flow for a given graph $G = (V \cup \{s, t\}, E)$, and the corresponding residual graph is $G_f = (V, E_f)$. Let $S \subset V$ be set of vertices that are reachable from the source $s \in V$ in the residual graph $G_f$. From the previous lemma we know that $f(S, V \setminus S) = |f|$ in $G$. For $|f| = f(S, V \setminus S) = c(S, V \setminus S)$ to be true it is necessary that all outgoing edges from $S$ are fully saturated and all ingoing edges have zero flow, by definition of flow and capacity of a cut. We prove that this is the case by way of contradiction.

Suppose there is an outgoing edge $e = (u, v) \in E$ with $u \in S$, $v \notin S$ such that $f(u, v) < c(u, v)$. Then $c_f(u, v) > 0$ which implies $v$ is also reachable from $s$ in $G_f$, i.e. $v \in S$ which contradicts our assumption.

Identically, if there is an ingoing edge $e = (v, u) \in E$ with $u \in S$, $v \notin S$ such that $f(v, u) > 0$ then this means $f(u, v) = -f(v, u) < 0$ which implies $c_f(u, v) = c(u, v) - f(u, v) > 0$ since $c(u, v) \geq 0$. Therefore $(u, v) \in E_f$, which means that $v$ is reachable from $s$ in $G_f$, contradicting $v \notin S$.

Now observe that $|f| \leq c(S, V \setminus S)$ for any flow $f$ and any cut $(S, V \setminus S)$:

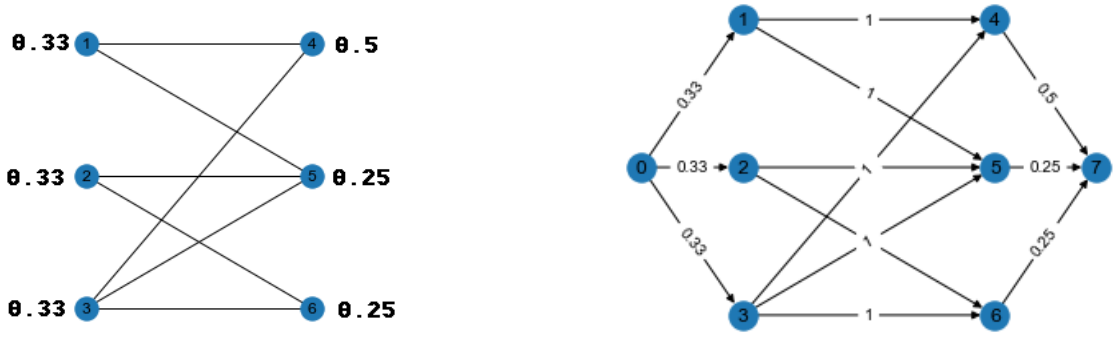
$$\begin{aligned} |f| = f(S, V \setminus S) &= \sum_{(x,y) \in (S \times (V \setminus S)) \cap E} f(x, y) - \sum_{(y,x) \in ((V \setminus S) \times S) \cap E} f(x, y) \\ &\leq \sum_{(x,y) \in (S \times (V \setminus S)) \cap E} f(x, y) \\ &\leq \sum_{(x,y) \in (S \times (V \setminus S)) \cap E} c(x, y) \\ &= c(S, V \setminus S). \end{aligned}$$

This means that the previously described cut $S$ of all reachable nodes from $s$ in $G_f$ must be the min-cut if $f$ corresponds to the maximum possible flow value. □

**Theorem 5.** *Suppose $G = (L \cup R, E)$ is an undirected bipartite graph with weighted vertices. Then the minimum weight weight vertex cover can be found by solving a maximum flow problem in a related flow network.*

*Proof.* To construct the related flow network $G'$ a source node $s$ is added which links to all nodes in $L$, and a sink node $t$ to which all nodes in $R$ link to. Capacities for outgoing edges from $s$ are set to the weights of corresponding vertices in $L$ and similarly for the sink $t$ and vertices in $R$. All original edges of $G$ which link nodes in $L$ to nodes in $R$ are oriented from $L$ to $R$ and given infinite capacity. This construction for bipartite graphs will be referred to as the flow-equivalent network of $G$, an example of which can be seen in Figure 2.2. Consider the residual graph $G_f$ produced by the max flow computation in $G'$. The set of reachable nodes from $s$ in $G_f$ form a min-cut $C$ and the minimum weight vertex cover is then given by $S = ((L \setminus C) \cup (R \cap C))$, as shall now be seen: First note that the weight of this vertex

**Figure 2.2:** Flow network reduction of a bipartite graph with weighted vertices.




cover is equal to the capacity of the corresponding cut because, given $i \in L$ and $j \in R$:

$$i \in S \iff i \notin C \iff (s,i) \in X_C; \quad j \in S \iff j \in C \iff (j,t) \in X_C$$

Now assume that $(i,j) \in E_f$ but $i \notin S$,$j \notin S$. This means that $i \in C$ and $j \notin C$ which contradicts the fact that $(i,j) \in E_f$, i.e. $j$ is reachable from $s$ in $G_f$ so it should be the case that $j \in C$. By contradiction $S$ must then be a vertex cover. □

## 2.3 Maximum flow algorithms

Several algorithms exist for solving the max-flow problem in polynomial time, and for now we shall restrict our attention to two such algorithms: the Edmonds-Karp algorithm [8] and Dinitz's algorithm [9]. These two algorithms incorporate the principles of the Ford-Fulkerson algorithm, but unlike Ford-Fulkerson are guaranteed to terminate in networks with irrational capacities [10]. These algorithms were also chosen because they are relatively efficient, namely Dinitz's algorithm, itself an improvement of the Edmonds-Karp algorithm. Other maximum flow algorithms include the push-relabel method and the method introduced by Orlin that builds upon the work of King et al. to find an $O(|V||E|)$-time maximum flow algorithm. These methods are less practical to implement, and their time complexity in the general case does not provide improvements over the execution of Dinitz's algorithm or the Edmonds-Karp algorithm on flow-equivalent networks of bipartite graphs, as we will demonstrate. For an overview of maximum flow algorithms, including those mentioned here, we refer to [11]. To close off this chapter we will also go over some approximation algorithms that attempt to improve the time complexity of the maximum flow problem at the expense of a relative error.

### 2.3.1 Edmonds-Karp algorithm

**Definition 7.** *Given a flow network $G = (V, E)$ with corresponding flow $f$, an augmenting path in $G_f$ is a path from source $s$ to sink $t$ in $G_f$ that contains no cycles and along which minimum capacity is strictly positive. A shortest augmenting path is an augmenting path of minimum possible length.*

The Edmonds-Karp algorithm [8] works by finding a shortest augmenting path on which to send additional flow at each iteration, until this path no longer exists. The path is found by performing a breadth-first search starting on the source node, sending flow through the augmenting path and then repeating the procedure until the sink is no longer accessible in the residual graph.

**Algorithm 2.1:** Edmonds-Karp max-flow algorithm.

**Input:** $G = (V, E)$ with capacity $c : E \to \mathbf{R}_0^+$,$s, t \in V$
**Output:** $R$ residual network;
F := $n \times n$ zero flow matrix;
C := $n \times n$ capacity matrix;
**while** True **do**
  Use BFS to find shortest augmenting path $P$ with flow $m$;
  **if** m=0 **then**
    **return** $C - F$
  **else**
    **for (** $i < |P|$ **)** {
      $F[P[i]][P[i+1]] += m$;
      $F[P[i+1]][P[i]] -= m$;

In order to find the time complexity of this algorithm consider the following results.

**Theorem 6.** *[12] The total number of flow augmentations performed on a general flow network $G = (V, E)$ by the Edmonds-Karp algorithm is $O(|V||E|)$.*

This bound can be improved in flow networks arising from the minimum weight vertex cover problem, whose construction was detailed in the proof of 5. Our proof of this result is provided in the following theorem.

**Theorem 7.** *Let $G = (L \cup R \cup \{s, t\}, E)$ be the flow network constructed from a bipartite graph, following the procedure detailed in the proof of Theorem 5. Then, given input $G$, the total number of flow augmentations performed by the Edmonds-Karp algorithm is $O(|V|)$.*

*Proof.* Let $G = (L \cup R \cup \{s, t\}, E)$ be a flow network in the conditions described. An edge $(u, v)$ in a residual network $G_f$ of $G$ is said to be critical on an augmenting path $p$ if the residual capacity of $(u, v)$ is the minimum residual capacity of edges in path $p$. An augmenting path has at least one critical edge,

and any critical edge belonging to an augmenting path disappears from the residual network after a flow augmentation along that path. Since we have restricted our attention to networks arising from the min-weight vertex cover reduction to max-flow, it shall be seen that $O(|V|)$ edges can become critical, as opposed to $O(|E|)$ in the general case. These edges are precisely those which link $s$ to nodes in $L$ and $t$ to nodes in $R$. Suppose that $f_1, ..., f_n$ is the sequence of flow augmentations performed up to and including the $n$th iteration of the outer loop of 1. The sum of these flows is obviously also a flow which is valid in the original network $G$. It is known that $c(s,u) < c(u,v)$ for any $(u,v) \in E$ and we assert that additionally $f(u,v) \leq f(s,u)$, where $f = \sum_{i=1}^{n} f_i$. The latter assertion is supported by the flow conservation property, because any $u \in L$ has only one incoming edge in the initial flow network $G$, and may have more than one outgoing edge. It follows that $c(s,u) - f(s,u) < c(u,v) - f(u,v)$, i.e. at no point in the execution of the algorithm can $(u,v)$ become a critical edge along an augmenting path. Note also that originally $G$ does not contain reverse edges, so only edges of the form $(s,u)$ or $(v,t)$ can become critical, which amounts to $|V| - 2$ edges.

As previously stated, for each augmenting path that is found, at least one critical edge is also found. In addition, once an edge is found to be critical it is reversed in the residual graph. Since only edges containing source or sink can be critical, no augmenting path can be found containing the reverse of these edges in the subsequent residual graph, because the search for this path is done by a breadth-first search. Reversing an edge $(u,s)$ would mean revisiting $s$, the starting node of the BFS. Similarly, the BFS cannot reverse any edge of the form $(t,v)$ because this would mean visiting the sink $t$ first, at which point the BFS would end. □

Because augmenting paths are found using BFS, each iteration of the outer loop takes time $O(|E|)$. If $|E| = \Theta(|V|^2)$, time complexity is $O(|V||E|) = O(|V|^3)$ for flow networks resulting from the reduction of the minimum weight vertex cover problem to the maximum flow problem. By the same token, the time complexity of Edmonds-Karp for general flow networks is $O(|V||E|^2)$, because in this case there are $O(|V||E|)$ augmenting paths (as stated in Theorem 6) and each corresponds to one iteration of the outer loop.

### 2.3.2 Dinitz's algorithm

Dinitz's algorithm uses a depth-first search to find a flow which disconnects the source from the sink in a graph constructed from the residual graph at the beginning of each iteration [9]. This graph is called a level graph and its definition is as follows:

**Definition 8.** *Given the residual flow network $G_f = (V, E_f)$ of some flow network $G = (V, E)$ with source*

*$s$, sink $t$ and flow function $f$, the level function $level : V \to \mathbf{N}_0$ of $G_f$ is defined for each $v \in V$ as the length of the shortest path from $s$ to $v$ in $G_f$, i.e. the number of edges in a path from $s$ to $v$ with the minimum number of edges. The level graph $G_L = (V, E_L)$ of $G_f$ is a flow network where*

$$E_L = \{(u, v) \in E_f : level(v) = level(u) + 1\}$$

*A blocking flow is a flow $f'$ in $G_L$ such that the residual graph of $G_L$ with $f'$ contains no path from $s$ to $t$.*

**Algorithm 2.2:** Dinitz's max-flow algorithm.

**Input:** $G = (V, E)$ with capacity matrix $C$
**Output:** $R$ residual network;
$f := 0$ for all edges in $E$; `// flow function`
$G_L :=$ level graph of residual graph $G_f$;
**while** sink is reachable in $G_f$ **do**
    $f :=$ blocking flow of $G_L$ found using DFS;
    Update level graph $G_L$ of new $G_f$;
**return** $C - F$

In order to find the time complexity of this algorithm consider the following result about the number of iterations performed by Dinitz's algorithm, adapted from [13].

**Theorem 8.** *Given any flow network $G = (V, E)$ the number of iterations of Dinitz's algorithm is at most $|V| - 1$.*

*Proof.* Consider an arbitrary iteration $i$ and a vertex $v \neq s$, and let $R_i = (V, E_i)$ be the residual graph obtained after the $i$th iteration, $L_i = (V, E_i')$ the level graph of $R_i$ and $level_i : V \to \mathbf{N}$ the corresponding level function. If $P$ is any shortest path from $s$ to $v$ in $R_{i+1}$, then $level_{i+1}(v)$ denotes the length of $P$, by definition of the level function itself. Note also that any edge in $R_{i+1}$ either belongs to $R_i$ or is the reverse of some edge in $R_i$. If all edges in the path $P$ already belong to $R_i$, then $P$ is also a path from $s$ to $v$ in $R_i$, but not necessarily the shortest one, i.e. $level_{i+1}(v) \geq level_i(v)$. Alternatively, if $P$ contains an edge $(u, w) \notin E_i$ then surely $(w, u) \in E_i$. Assuming $(u, w)$ is the first such edge occurring in the traversal of path $P$ let $P'$ be the path from $s$ to $u$ contained in $P$. Since every edge in $P'$ belongs to $R_i$ the previous case applies and $level_{i+1}(u) \geq level_i(u)$. Having $(u, w) \notin E_i$ and $(u, w) \in E_{i+1}$ allows us to state that the algorithm sent some flow through $(w, u)$ between iterations $i$ and $i+1$, meaning that $(w, u)$ belonged to $L_i$ and $level_i(u) = level_i(w) + 1$ from the definition of the level graph. Similarly, it is also the case that $level_{i+1}(w) = level_{i+1}(u) + 1$, because $(u, w)$ belongs to $P$, the shortest path from $s$ to $v$ in $R_{i+1}$. From the last two equations and $level_{i+1}(u) \geq level_i(u)$ we can infer that $level_{i+1}(w) \geq level_i(w) + 2 > level_i(w)$. We have thus showed that $level_{i+1}(v) \geq level_i(v)$ for any $v \in V \setminus \{s\}$. In particular we have $level_{i+1}(t) \geq level_i(t)$ and it is straightforward to see that this inequality is strict, otherwise this would mean that there must be a shortest path from $s$ to $t$ which is shared between

$R_i$ and $R_{i+1}$. This contradicts the fact that a blocking flow was found at iteration $i$ because then at least one of the edges in this path would have been reversed. The fact that $level_{i+1}(t) > level_i(t)$ for any $i$ implies that there can be at most $|V| - 1$ iterations, because the level of $t$ is upper bounded by this number. □

The time complexity of Dinitz's algorithm for a general network $G = (V, E)$ is $O(|V|^2|E|)$. [9] As in the case of the Edmonds-Karp algorithm, this bound can be improved for the types of networks which are of interest in our case. Our proof of this improved time complexity relies on the fact that while the Edmonds-Karp algorithm only finds a single shortest augmenting path at each iteration, Dinitz's algorithm finds a blocking flow which consists of all current shortest augmenting paths.

**Theorem 9.** *Let $G = (L \cup R \cup \{s, t\}, E)$ be the flow network constructed from a bipartite graph, following the procedure detailed in the proof of Theorem 5. Then, given input $G$, time complexity of Dinitz's algorithm is $O(|V||E|)$ if $|E| = \Omega(|V|)$.*

*Proof.* At each of the $N = \mathcal{O}(|V|)$ phases of Dinitz's algorithm, the level graph is constructed with a BFS in $\mathcal{O}(|E|)$ time. We have seen that there are $\mathcal{O}|V|$ augmenting paths in total found by executing the Edmonds-Karp algorithm with input $G$. Augmenting paths of the same length found by Edmonds-Karp form a blocking flow, which is found in each phase for Dinitz's algorithm. For each path that is found by a depth-first search in the level graph, there exists at least one critical edge of the form $(s, u)$ or $(v, t)$ for $u, v \in V$ which remains critical until the algorithm terminates, as previously shown in the proof of Theorem 7. Suppose there are $n_i$ such paths found by the blocking flow at phase $1 \leq i \leq N$, and each of them has length $\mathcal{O}(|V|)$. Hence each path takes time $\mathcal{O}(|V|)$ to find and therefore the time for finding a blocking flow is $\mathcal{O}(|V|n_i)$. Because $\sum_{i=1}^{N} n_i = \mathcal{O}(|V|)$, the total time complexity of Dinitz's algorithm becomes $\mathcal{O}(|V||E| + |V|^2)) = \mathcal{O}(|V||E|)$ if $|E| = \Omega(|V|)$. □

### 2.3.3 Maximum flow approximation algorithms

Algorithms for approximating the maximum flow of a network use various techniques. One such algorithm uses the approach suggested by Christiano et al. [14] based on electrical flows and Laplacian linear system solvers. This approach covers undirected graphs, but it is possible to reduce the directed maximum flow problem into the undirected maximum flow problem to obtain a $(1 - \varepsilon)$-approximation of maximum flow in $0 < \varepsilon < 1/2$ in $\tilde{O}(|E|^{4/3}\varepsilon^{-3})$ time [15], where $\tilde{O}(f(n))$ is equivalent to $O(f(n)\log^c(n))$ for some constant $c$. The authors of these methods also suggest using smoothing and sampling techniques to obtain an $\tilde{O}(|E||V|^{1/3}\varepsilon^{-11/3})$-time algorithm for directed maximum flow. If $|E| = O(|V|^2)$ then

these bounds become $\tilde{O}(|V|^{8/3}\varepsilon^{-3})$ and $\tilde{O}(|V|^{7/3}\varepsilon^{-11/3})$ respectively, which in either case is asymptotically faster than the $O(|V||E|) = O(|V|^3)$ time complexity of Dinitz's algorithm and the Edmonds-Karp algorithm for the flow-equivalent networks of bipartite graphs. However, it is important to recall that the time complexities of the latter exact algorithms are of $O(|V||E|^2) = O(|V|^5)$ and $O(|V|^2|E|) = O(|V|^4)$ for general networks, and so it is entirely feasible that there exist better worst-case bounds for the time complexity of the $(1-\varepsilon)$-approximation algorithms mentioned. Later we will theoretically explore how these approximation algorithms can be integrated in the GTP algorithm.

# 3

# The GTP Algorithm

In this chapter we present the Geodesic Tree Path algorithm, [3] an algorithm which given two trees is meant to determine the length of the geodesic path between them. We begin by describing the whole procedure and its components and then move on to analyse the worst-case time and space complexities theoretically achievable. Finally we explain some important aspects of our implementation of this algorithm in Python and the choices made in order to make our implementation comply with the theoretical boundaries that resulted from our analysis.

## 3.1 Description

### 3.1.1 The GTP algorithm for disjoint trees

As explained in the section relating to the Extension problem, the (P3) condition hints at a procedure for iteratively improving upon a chosen initial proper path, until the length is minimal. Provided that two given trees are disjoint (the corresponding set of splits is disjoint), the procedure detailed in Algorithm **??** does precisely this.

Given input $T_1, T_2 \in \mathcal{T}_n$ such that $T_1 = (L, \mathcal{E})$ and $T_2 = (R, \mathcal{E}')$, the procedure begins by building a simple path between the two trees, represented as a support pair $(\mathcal{A}, \mathcal{B})$ where $\mathcal{A}$ and $\mathcal{B}$ are ordered vectors representing partitions of $\mathcal{E}$ and $\mathcal{E}'$ respectively. Initially $\mathcal{A} = (\mathcal{E})$ and $\mathcal{B} = (\mathcal{E}')$ represent the cone path, and the incompatibility graph $G$ of $T_1$ and $T_2$ is built.

The algorithm enters a loop that will partition the sets $\mathcal{E}$ and $\mathcal{E}'$ depending on the existence of solutions to the Extension Problem, modifying the contents of the support pair $(\mathcal{A}, \mathcal{B})$ in the process. Suppose that $A$ and $B$ are elements of $\mathcal{A}$ and $\mathcal{B}$ indexed by the same position at a given point in the execution of the algorithm. The algorithm builds the subgraph $G(A, B)$ of $G$ induced by $A \subset \mathcal{E}$ and $B \subset \mathcal{E}'$. It then tries to find a solution to the Extension Problem, in the way described by Theorem 5. In summary, this implies:

1. Building a flow-equivalent network $G'$ of $G(A, B)$;
2. Applying a maximum flow algorithm to obtain a residual network $R$ of $G'$;
3. Determining the set of reachable nodes from the source that forms cut $C$;
4. Computing the vertex cover $C_1 \cup D_2$ where $C_1 = L \setminus C$ and $D_2 = R \cap C$.

Assuming $C_2 = L \setminus C_1$ and $D_1 = R \setminus D_2$, if neither of the sets $C_1, C_2, D_1, D_2$ are empty then we have

non-trivial partitions of $A$ and $B$ forming a solution to the Extension Problem. If no such partition exists, the algorithm simply moves on to a next pair that is yet to visit. This is repeated until we have reached $\mathcal{A} = (A_1, ..., A_k)$ and $\mathcal{B} = (B_1, ..., B_k)$ where there does not exist a solution to Extension Problem for any pair $(A_i, B_i)$.

**Example 2.** *This example illustrates the computations performed by the disjoint GTP algorithm when given trees $T_1, T_2 \in \mathcal{T}_4$ of Figure 2.1 as input. Assume that $|e_4|_{T_1} = 2, |e_5|_{T_1} = 1, |e_3|_{T_2} = 2, |e_1|_{T_2} = 1$ , where $|e|_T$ denotes the length of edge $e$ in tree $T$. The splits associated with these edges are as follows:*

$$\sigma_{e_4} = \{2, 3|0, 1, 4\}; \quad \sigma_{e_5} = \{0, 1|2, 3, 4\};$$
$$\sigma_{e_3} = \{1, 2, 3|0, 4\}; \quad \sigma_{e_1} = \{1, 2|0, 3, 4\}.$$

*It can easily be seen that the only split compatibility that exists is between $\sigma_{e_4}$ and $\sigma_{e_3}$ i.e. only $e_3$ and $e_4$ are simultaneously present in some orthant, and the incompatibility graph will therefore have edges $(e_4, e_1), (e_5, e_3), (e_5, e_1)$. Vertex weights will be given by:*

$$w_{e_4} = w_{e_3} = \frac{2^2}{2^2 + 1^2} = \frac{4}{5}; \quad w_{e_5} = w_{e_1} = \frac{1^2}{2^2 + 1^2} = \frac{1}{5}.$$

*Translating these weights to capacities in the incompatibility graph gives the following flow-equivalent network:*

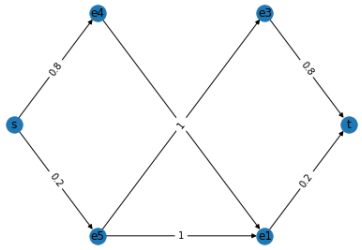



**Figure 3.1:** Flow-equivalent network to the incompatibility graph of $T_1$ and $T_2$

*A maximum flow of $0.4$ can be achieved if we send $0.2$ flow units along the path $s \to e_4 \to e_1 \to t$ and another $0.2$ units of flow along the path $s \to e_5 \to e_3 \to t$. The set of nodes reachable from $s$ in the residual network will then be $C = \{e_4, e_1\}$, producing a vertex cover $S = \{e_5, e_1\}$ that is of minimum weight in the incompatibility graph. Therefore we would obtain the path support pair $(\mathcal{A}, \mathcal{B})$ such that $\mathcal{A} = (\{e_5\}, \{e_4\})$ and $\mathcal{B} = (\{e_3\}, \{e_1\})$. Since the sets in $\mathcal{A}$ and $\mathcal{B}$ cannot be partitioned further we have arrived at the geodesic path. This path traverses the orthants formed by the sets $\{e_4, e_5\}, \{e_3, e_4\}, \{e_1, e_3\}$ in that order, according to Definition 2. The length of this path would then be:*

$$||(|e_5|_{T_1} + |e_3|_{T_2}, |e_4|_{T_1} + |e_1|_{T_2})|| = \sqrt{(1+2)^2 + (2+1)^2} = 3\sqrt{2}$$

**Algorithm 3.1:** The GTP Algorithm for disjoint trees.

**Input:** $T_1 = (L, \mathcal{E}), T_2 = (R, \mathcal{E}') \in \mathcal{T}_n$ disjoint $n$-trees

**Output:** Geodesic distance between $T_1$ and $T_2$

$G :=$ incompatibility graph of $T_1$ and $T_2$;

$w :=$ dictionary of vertex weights;

$\mathcal{A} := (\mathcal{E}), \mathcal{B} := (\mathcal{E}')$;

**while** True **do**

  **for (** $i = 0; i < |\mathcal{A}|; i++$ **)** {

    $A := i$'th element of $\mathcal{A}$;

    $B := i$'th element of $\mathcal{B}$;

    $G(A, B) :=$ subgraph of $G$ induced by $A$ and $B$;

    **for (** $e \in A$ **)** {

      $w[e] = \frac{|e|^2}{\|A\|^2}$

    **for (** $e \in B$ **)** {

      $w[e] = \frac{|e|^2}{\|\mathcal{E}'\|^2}$;

    $G' :=$ flow-equivalent network of $G$; $R :=$ residual matrix of applying max flow algorithm to $G'$;

    $C_1 \cup D_2 :=$ min. weight vertex cover of $G(A, B)$ computed from $R$;

    // $(C_1 \subseteq A, D_2 \subseteq B)$

    $C_2 = A \setminus C_1, D_1 = B \setminus D_2$;

    $w' :=$ total weight of $C_1 \cup D_2$;

    **if** $w' < 1$ **then**

      Replace $A$ with $C_1, C_2$ in $\mathcal{A}$;

      Replace $B$ with $D_1, D_2$ in $\mathcal{B}$;

      break;

  **if** $w \geq 1$ **then**

    **return** $[\|A_1\| + \|B_1\|, ..., \|A_k\| + \|B_k\|]$;

### 3.1.2 Common edge handling

**Algorithm 3.2:** The GTP Algorithm admitting common splits.

**Input:** $T_1, T_2 \in \mathcal{T}_n$

**Output:** The geodesic distance between $T_1$ and $T_2$

$C := \mathcal{E}_1 \cap \mathcal{E}_2$ //$T_1(i), T_2(i)$ are subtrees of $T_1, T_2$ indexed by the same $e \in C$

$r := |C|$;

$\mathcal{P}_1 := \{T_1(i)\}_{i=1}^{r}$;

$\mathcal{P}_2 := \{T_2(i)\}_{i=1}^{r}$;

$v :=$ empty list;

**for (** $i = 1; i < r; i++$ **)** {

  $p :=$ result of Algorithm **??** applied to $T_1(i)$ and $T_2(i)$;

  Concatenate list $v$ with list $p$;

  **if** $e_i$ is not the root edge of $T_1$ and $T_2$ **then**

    diff := $||e_i|_{T_1} - |e_i|_{T_2}|$;

    Append diff to list $v$;

$L :=$ length of list $v$;

**return** $\sqrt{\sum_{i=1}^{L} v[i]^2}$

The procedure for dealing with pairs of trees whose edge sets are not necessarily disjoint is done by splitting each tree at those common edges to form two forests of disjoint subtrees, where subtrees in each forest are indexed by their parent edge. This parent edge can be the root edge of the respective tree, which is always present in the set of common edges between any two trees. This allows us to form a collection of pairs of disjoint subtrees $\mathcal{P} = \{(T_1(e), T_2(e)) : e \in \mathcal{E} \cap \mathcal{E}'\}$, and we apply the previously explained disjoint trees version of the GTP algorithm for each of these pairs. The support pairs of the geodesic path between each pair of subtrees is then used to form the full geodesic path between $T_1$ and $T_2$, and the corresponding distance. Assuming $|\mathcal{E} \cap \mathcal{E}'| = r$, let $(A_1(l), ..., A_{k_l}(l))$,$(B_1(l), ..., B_{k_l}(l))$ be the support for the geodesic path bewteen a given subtree pair $(T_1(e), T_2(e)) \in \mathcal{P}$. Then the length of the geodesic path between $T_1$ and $T_2$ is given by:

$$\begin{aligned} L(\Gamma) = ||(&||A_1(1)|| + ||B_1(1)||, ..., ||A_{k_1}(1)|| + ||B_{k_1}(1)||, \\ &..., \\ &||A_1(r)|| + ||B_1(r)||, ..., ||A_{k_r}(r)|| + ||B_{k_r}(r)||, \\ &|e_1|_{T_1} - |e_1|_{T_2}, ..., |e_r|_{T_1} - |e_r|_{T_2})|| \end{aligned}$$

where $|e|_T$ denotes the length of edge $e$ in tree $T$. [3] The pseudo-code for this complete version of the GTP algorithm can be seen in Algorithm 3.2.

## 3.2 Theoretical Analysis

We now present the theoretical space and time complexities for each of the steps underlying the execution of the GTP algorithm. A similar proof of this time complexity is presented in [3].

**Theorem 10.** *Taking $T_1, T_2 \in \mathcal{T}_n$ as input, the GTP algorithm has worst-case time complexity in $\mathcal{O}(n^4)$ and space complexity in $\mathcal{O}(n^2)$.*

*Proof.* Taking $T_1, T_2 \in \mathcal{T}_n$ as input, the algorithm first determines the set of splits belonging to both trees. Each tree has $n-2$ internal edges, and so the time for computing each split set is $\mathcal{O}(n)$. Set intersection can then be implemented in linear time using hash tables. Each tree is partitioned according to the common split set, and the version of the GTP algorithm for disjoint trees is applied afterwards to each pair of subtrees indexed by a given split. Assume that the partitions induced by slicing the trees at their common edges are given respectively by $\{T_1(i)\}_{i=1}^{r}$ and $\{T_2(i)\}_{i=1}^{r}$. Let $n_i$ be the number of leaves of $T_1(i)$ and $T_2(i)$, other than a possible root node of degree 1.

Given $T_1(i)$ and $T_2(i)$, the disjoint trees version of the GTP algorithm determines their incompatibility graph. Since they both have $n_i - 2$ internal edges each, the worst case is when $\mathcal{O}(n_i^2)$ pairs of corresponding splits are incompatible. Determining the compatibility of two splits using bitwise operations on the split bitmasks we describe in the implementation details takes time $\mathcal{O}(n_i)$, meaning that the worst case time complexity for the construction of these graphs is of $\mathcal{O}(n_i^3)$. On the other hand, space needed to perform the relevant computations is just $\mathcal{O}(n_i^2)$ in the worst case, which is the space needed to store a representation of the resulting incompatibility graph along with the respective vertex weights. Thus the flow-equivalent network should also take space $\mathcal{O}(n_i^2)$ in the worst case. Since our implementation represents this network as an adjacency list and a matrix of capacities, it is expected that space complexity will be $\mathcal{O}(n_i^2)$.

Entering the main `while` loop in the disjoint version of GTP, given $A \in \mathcal{A}$ and $B \in \mathcal{B}$, the incompatibility graph $G(A, B)$ is a subgraph of $G$, and its adjacency list can be constructed simply by scanning the entries of the adjacency list of $G$ corresponding to the nodes in $A \cup B$, taking $O(n_i^2)$ time.

In the flow-equivalent network, we have that $|V| = \mathcal{O}(n_i)$ and $|E| = \mathcal{O}(n_i^2)$ and thus time complexity of maximum-flow algorithms we studied is $\mathcal{O}(|V||E|) = \mathcal{O}(n_i^3)$ for the case of the flow-equivalent network of the bipartite graphs. However, the space needed is also the space taken by the representation of the flow network given as input. Since we chose to represent edge capacities in a capacity matrix, the space needed in our case is of $\mathcal{O}(n_i^2)$.

The vertex cover is then found by the method detailed in the proof of Theorem 5. Determining the

set of reachable nodes from the source in the residual graph is done with a BFS in time $\mathcal{O}(|E|) = \mathcal{O}(n_i^2)$, and computing the vertex cover from this cut takes time $\mathcal{O}(|V|) = \mathcal{O}(n_i)$.

If this vertex cover forms a valid solution to the Extension Problem the support pair is reconstructed accordingly, and since the number of internal edges of $T_1(i)$ and $T_2(i)$ is $n_i - 2$, there exist $\mathcal{O}(n_i)$ solutions of the Extension Problem throughout the execution of the disjoint trees GTP algorithm in the worst-case.

Therefore each execution of the disjoint trees version of the GTP algorithm should have worst-case time complexity in $\mathcal{O}(n_i^4)$. Observe that $\sum_{i=1}^{r} n_i = n + (r-1) < 2n$, since there are $r-1$ common edges between $T_1$ and $T_2$ and $r < n - 2$. Due to these constraints the complexity of the GTP algorithm remains unaltered when applied to all pairs of subtrees. In other words, if the execution time of the disjoint GTP algorithm with $T_1(i), T_2(i) \in \mathcal{T}_{n_i}$ as input is given by $f(n) = an^4 + \omega(n^4)$ for some $a > 0$ then the complexity of GTP will be given by:

$$\sum_{i=1}^{r} f(n_i) = a \sum_{i=1}^{r} n_i^4 + \sum_{i=1}^{r} \omega(n_i^4) \leq a \left( \sum_{i=1}^{r} n_i \right)^4 + \omega(n^4) = an^4 + \omega(n^4) = \mathcal{O}(n^4)$$

Using a similar argument, space complexity of the GTP algorithm should be of $\mathcal{O}(n^2)$, which is the space needed to store and represent the capacity matrices of the incompatibility graphs needed for computations at any given point in the execution of algorithm. What is meant by this is that even though the total size of data structures built and used can be greater than $\mathcal{O}(n^2)$, they can be written over and replaced as they cease to be useful. □

### 3.2.1 The GTP algorithm with approximate max-flow

If one were to replace Dinitz's algorithm or the Edmonds-Karp algorithm by an $(1-\varepsilon)$-approximate maximum flow algorithm, the complexity of the algorithm would change accordingly. The complexity of the GTP algorithm would become $\tilde{O}(n^{11/3}\varepsilon^{-3})$ or $\tilde{O}(n^{10/3}\varepsilon^{-11/3})$ for the version that employs smoothing and sampling techniques. However, it is not obvious how the error in the maximum flow approximation would affect the final geodesic distance estimate. If $f'$ is the approximation for the maximum flow $f$ then $(1-\varepsilon)|f| \leq |f'| \leq |f|$, and since the existence of a solution to the Extension Problem is contingent on $|f| < 1$, approximation errors may lead to a sub-optimal solution of the Extension Problem and thus an incorrect partition of the support pairs. Furthermore, if at any point in the execution of the GTP algorithm a sub-optimal solution is found, that changes the path space to be searched from there on out, locking in the error. To measure this error propagation we propose simulating the execution of the GTP algorithm with an approximate max-flow algorithm, and calculating the relative error for a set of trees for each $\varepsilon$ in

a given range. In that case, it would be appropriate to use the jackknife resampling technique to obtain confidence intervals for the relative distance error, and study how it behaves in relationship to $\varepsilon$.

## 3.3 Implementation details

The GTP algorithm and the maximum flow algorithms on which it relied were implemented in Python 3. Beyond Python's standard library, packages used include `dendropy` - a library for phylogenetic computing - and `numpy`, a popular scientific computing package.

**Listing 3.1:** GTP algorithm main routine in Python.

```
def GTP(T1,T2,algorithm="EK"):
    T1.encode_bipartitions()
    T2.encode_bipartitions()
    # determine common splits in T1 and T2
    C = part.common_internal_edges(T1, T2)
    vector = []
    #partition trees according to common splits
    T1P = part.tree_partition(T1,C[0])
    T2P = part.tree_partition(T2,C[1])
    for b in T1P:
        #ensure the set of leaf node labels is the same for both trees
        trees = part.taxon_namespace_migration(T1P[b],T2P[b])
        vector += list(disjoint_GTP(*trees,algorithm=algorithm))
        # determine absolute difference of lengths of common edges
        if b:
            diff = T1.split_bitmask_edge_map[b].length
            diff -= T2.split_bitmask_edge_map[b].length
            vector.append(abs(diff))
    dist = lambda x: sqrt(sum(i**2 for i in x))
    return dist(vector)
```

The first thing the `GTP` procedure does is take inpu trees $T_1, T_2 \in \mathcal{T}_n$ and determine which edges are in common. For the `Tree` class in `dendropy` one can use the method `encode_bipartitions`[1]. One of the things this does is encode a split bitmask for each edge, which is useful for determining their compatibility. We partition the trees accordingly, representing each partition as a Python `dict` indexed

by split bitmasks where values are still of the dendropy `Tree` type. The time and space complexity of this sequence of operations is therefore $\mathcal{O}(n)$. The `disjoint_GTP` function is then applied to pairs of trees encoded by the same bitmask.

`disjoint_GTP` (see code in Appendix A.1) begins by calculating the incompatibility graph using the function `incompatibility_graph`, which returns an adjacency list with antiparallel edges included, i.e. for each edge the reverse edge must be present. It proceeds exactly as detailed in the theoretical analysis, determining the compatibility for $n^2$ pairs of edges. Normalized weights are then computed for each node in the incompatibility graph, from the lengths of edges in the respective tree. The construction of this graph only has to be done once for each execution of `disjoint_GTP`, because subsequent incompatibility graphs are subgraphs of the original, resulting in changes in the normalized vertex weights, which are then translated into a capacity matrix. Time complexity for this set of procedures is therefore $\mathcal{O}(n^3)$ and space complexity is $\mathcal{O}(n^2)$.

**Listing 3.2:** Python implementation of Dinitz's algorithm.

```
def Dinic(A,C):
    '''
    A = adjacency list
    C = capacity matrix
    '''
    F = np.zeros_like(C) # F is the flow matrix
    level = levels(A,C,F)
    B,reachable = level_graph(A,C,F,level)
    # while sink is reachable in residual graph
    while(reachable):
        # Find blocking flow F using a DFS
        F = DFS(B,C,F,0,level)
        # Rebuild level graph
        level = levels(A,C,F)
        B,reachable = level_graph(A,C,F,level)
    return C-F
```

[1] The source code for `encode_bipartitions` shows that it uses a post order edge iterator to calculate split bitmasks, i.e. it traverses an edge only after having visited its children. Each edge's bitmask is found by performing bitwise-OR operations on its childrens' bitmasks. Since the size of the bitmask is the number of leaves $n$ and in Python 3 the number of bits for an integer is technically unlimited, the complexity of this operation is $\mathcal{O}(n)$ in theory, but for all practical purposes the cost is negligible as confirmed by separate experiments.

Maximum flow algorithms are then used to find a solution to the Extension Problem on the flow-equivalent network of a given subgraph of the incompatibility graph. Our implementation of Dinitz's and Edmonds-Karp algorithms only takes an adjacency list and a capacity matrix. This adjacency list must have antiparallel edges. The Edmonds-Karp implementation in particular made use of an optional technique which is used if the `select_edges` keyword is set to `True`. This technique simply traverses the edges in the incompatibility graph and successively sends the maximum allowed flow along the paths of length 3 containing those edges in the flow-equivalent network. This is meant to reduce running time, and in some cases it already gives a good approximation of the real maximum flow without having to perform any BFS.

**Listing 3.3:** Python implementation of the Edmonds-Karp algorithm.

```
def EK(A,C,select_edges=True):
    flow = 0
    F = np.zeros_like(C, dtype = float)
    if select_edges:
        visited = np.full(len(C[0]),False)
        for i in range(len(C[0])):
            if C[0][i] > 0:
                for j in A[i]:
                    if not visited[j] and C[i][j]>F[i][j]:
                        m = min(C[0][i],C[i][j],C[j][len(C)-1])
                        F[0][i] += m
                        F[i][0] -= m
                        F[i][j] += m
                        F[j][i] -= m
                        F[j][len(C)-1] += m
                        F[len(C)-1][j] -= m
                        flow += m
                        visited[j] = True
                        break
    while True:
        # find augmenting path P with flow m
        m, P = BFS(A,C,F)
        flow += m
        if m == 0: #no augmenting path is found
            return C-F
        # send flow through P
```

```
27          for node in range(len(P)-1):
28              F[P[node]][P[node+1]] -= m
29              F[P[node+1]][P[node]] += m
```

Capacity matrices were represented as `numpy` arrays, and adjacency lists as lists of `list`. Adjacency lists are much more intuitive here because they allow for easy access to one node's neighbours in breadth-first traversals of the graph, whereas in a capacity matrix neighbours can only be found by scanning non zero elements across a row and a column indexed by the node. Our Edmonds-Karp implementation does not build other adjacency lists other than the one provided as input, but this is done in Dinitz's algorithm implementation, namely for updating the level graph at each iteration.

While the Edmonds-Karp algorithm uses an adapted breadth-first search procedure to find shortest augmenting paths, Dinitz's algorithm uses a BFS-like procedure to construct level graphs. Additionally, Dinitz's algorithm employs an adapted depth-first search procedure to find a blocking flow. Breadth-first and depth-first searches use some kind of data structure to keep track of nodes which are yet to visit or already visited. The use of the `collection.deque` type in all these procedures - exemplified in Listing 3.4 - instead of the native Python `list` type is due to a small but important distinction, which consists in the fact that the `popleft()` operation on a `deque` has complexity $\mathcal{O}(1)$ while the equivalent operation for a `list` named `q` is done with `q.pop(0)`, and this has complexity $\mathcal{O}(\texttt{len(q)})$ [16].

**Listing 3.4:** Auxiliary breadth-first search procedure to the Edmonds-Karp algorithm.

```
1  def BFS(A,C,F):
2      '''
3      A - adjacency list
4      C - capacity matrix
5      F - flow matrix
6      '''
7      q = deque([0]) #queue q of elements yet to visit
8      flow = decimal.Decimal('Infinity') #flow of the path P
9      parent = np.full(len(A),-1)
10     while q:
11         curr = q.popleft()
12         for node in A[curr]:
13             if C[curr][node] > F[curr][node] and\
14                 parent[node] == -1:
15                 parent[node] = curr
16                 if node!=len(A)-1:
```

```
17                      q.append(node)
18                  else:
19                      Path = []
20                      while node != 0:
21                          Path.append(node)
22                          par = parent[node]
23                          flow = min(flow,\
24                                     C[par][node]-F[par][node])
25                          node = parent[node]
26                      Path.append(0)
27                      return flow,Path
28      return 0,None
```

Assuming the `BFS` function used to find augmenting paths in the Edmonds-Karp implementation is correct and has a time complexity of $\mathcal{O}(|E|)$, it follows immediately that the time complexity of `EK` is $\mathcal{O}(|V||E|)$, considering that the number of iterations is given by the number of augmenting paths which is in $\mathcal{O}(|V|)$ (Section 7). Dinitz's algorithm used very similar BFS-like procedures `levels` and `level_graph`, and an additional recursive function `DFS` (presented in Listing 3.5). Assuming all of these are correct and conform to the established theoretical time complexities the correctness and complexity of $\mathcal{O}(|V||E|)$ for Dinitz's algorithm is also evident.

**Listing 3.5:** Auxiliary depth-first search procedure to Dinitz's algorithm.

```
1  def mymin(x,y):
2      return y if x<0 else min(x,y)
3
4  def DFS(A, C, F, k, level, c = -1):
5      tmp = c
6      if k == len(C)-1:
7          return c
8      for i in A[k]:
9          if F[k][i] < C[k][i]:
10             f = DFS(A, C, F, i, level,\
11                 mymin(tmp, C[k][i]-F[k][i]))
12             F[k][i] += f
13             F[i][k] -= f
14             tmp -= f
15     return F if k==0 else c-tmp
```

A non-trivial solution of the Extension Problem, if it exists, is calculated by determining the set of reachable nodes from the source in the residual graph, represented by the residual capacity matrix. This forms a cut $C$ and the minimum weight vertex cover is given by $S = (L \setminus C) \cup (R \cap C)$, as previously explained in Theorem 5. The cut is found using a BFS on the residual capacity matrix in $\mathcal{O}(|V|^2)$ time, and the vertex cover is found in linear time on the size of this cut, which itself is $\mathcal{O}(|V|)$, by representing sets $L, C, R$ as arrays of booleans (similar to an indicator function) and using `numpy` functions `logical_not` and `logical_and`. Finally, the resulting sets which form a non-trivial solution to the Extension Problem are appended to the new path support in order, and the cycle breaks.

# 4

# Experimental Evaluation

In this chapter experiments were done with the intention of validating the results from the previous chapter, as well as demonstrating that all the implemented algorithms perform significantly better than the theoretical bounds, under a reasonable choice of a probability distribution over inputs. Separate experimental tests were carried out for the complete GTP algorithm itself and the maximum flow algorithms on which it depends, so as to demonstrate the relationship between their space and time complexities.

## 4.1 Maximum flow algorithms

### 4.1.1 Random bipartite graphs

We decided to separately test our implementations of the maximum flow algorithms discussed in Chapter 2 in the context of the bipartite minimum weight vertex cover problem (and its corresponding reduction to max-flow). This was done by generating bipartite random graphs according to the Erdős–Rényi model, their corresponding weights, and then constructing the respective flow network. Given $n, m \in \mathbf{N}$ and a probability $p \in (0, 1)$, in the Erdős–Rényi model each edge from the complete bipartite graph $K_{n_1,n_2}$ is chosen to be part of the final graph $G = (L \cup R, E)$ with probability $p$, independently of other edges. Since $K_{n_1,n_2}$ has exactly $n_1 n_2$ edges, the expected value of $|E|$ is $pn_1n_2$ edges. The probability $p$ is therefore the expected value for the graph density of a graph generated according to this model. Assuming $n_1 = n_2 = n$, our graphs were generated fixing $p = 0.2$ or $p = 0.6$ for several values of $n$ and therefore $|E| = \mathcal{O}(|V|^2)$, where $V = L \cup R$. Random weights were assigned to vertices and normalized so that weights of vertices in $L$ would sum to 1, and likewise for $R$. Finally, the appropriate flow network reduction was performed.

### 4.1.2 Results

The results presented in Figure 4.1 show the cube root of the execution time of the maximum flow algorithms taking as input 25 random bipartite graphs with varying average density $p = 0.2, 0.6$ and size $n = 20, 40, ..., 300$, where $n = |L| = |R|$ is the number of nodes on each side of the bipartite graph.

Linear regression was performed on a log-log scale to determine the power law which better describes the results (before applying the cubic root), producing the slopes and $r^2$ coefficients of determination in Table 4.1. These appear to conform to the theoretical bounds we established of $\mathcal{O}(|V||E|)$ for both max-flow algorithms, since we're assuming $|E| = \mathcal{O}(|V|^2)$. It is also important to note that both algorithms are faster for random graphs with smaller average densities of $p = 0.2$.

**Figure 4.1:** Cubic roots of maximum flow algorithms execution time for random bipartite graphs. Top and bottom rows correspond to $p = 0.2$ and $p = 0.6$ respectively.

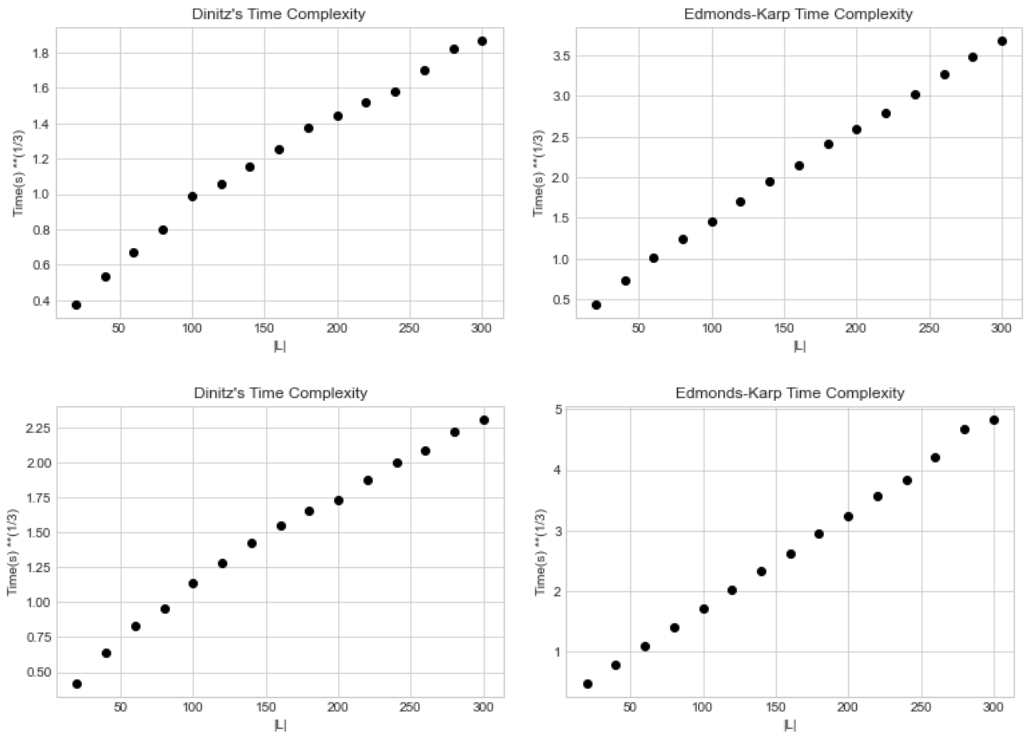


**Figure 4.2:** Elapsed execution time of maximum-flow algorithms versus the average density $p$ of input graphs.

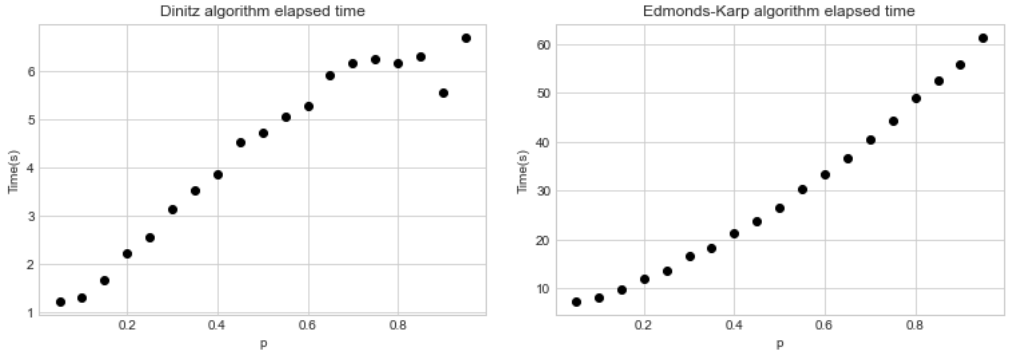



**Table 4.1:** Linear regression performed in log-log scale on time measurements in Figure 4.1, for different maximum flow algorithms and for random bipartite graphs with average densities $p = 0.2$ and $p = 0.6$.

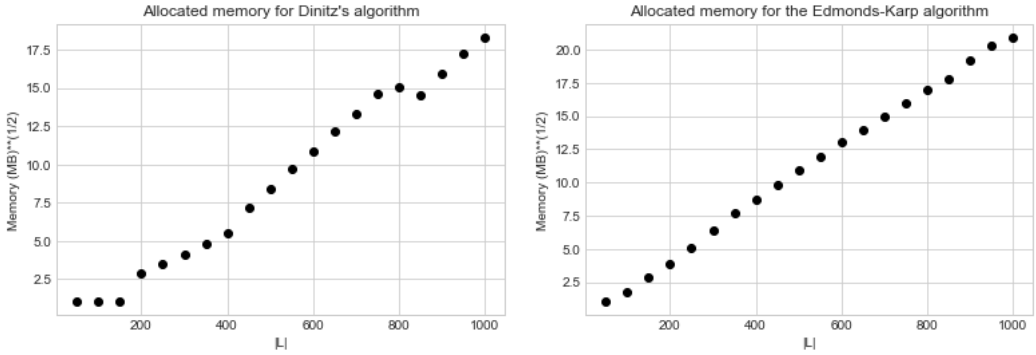
| | Dinitz | | Edmonds-Karp | |
|---|---|---|---|---|
| | slope | $r^2$ | slope | $r^2$ |
| p=0.2 | 1.82 | 0.9974 | 2.37 | 0.9992 |
| p=0.6 | 1.90 | 0.9993 | 2.65 | 0.9973 |

Further measurements were also done to determine how these maximum-flow algorithms respond to increasing densities while maintaining the number of nodes fixed, and they are featured in Figure 4.2. The power-law which best seems to describe the increase in time when compared to density was once again found using log-log scale linear regression, producing slope 1.48 with $r^2 = 0.9027$ for Dinitz's algorithm and slope 1.70 with $r^2 = 0.9923$ for the Edmonds-Karp algorithm.

Tests relating to space complexity of these algorithms were also performed, by measuring allocated memory for their execution while varying the number of nodes. An overhead was observed for both algorithms, which might be attributable to the use of dynamic data structures in our implementation of these algorithms, and the size of imported packages. It is interesting to note as well that the time for experimental evaluation of Dinitz's algorithm seems to better bounded by $\mathcal{O}(|V|^2)$ as opposed to the theoretical bound of $\mathcal{O}(|V||E|)$ (equal to $\mathcal{O}(|V|^3)$ with $|E| = \mathcal{O}(|V|^2)$ ) that was determined in Chapter 3, while the Edmonds-Karp performs closer to that same time complexity of $\mathcal{O}(|V|^3)$.

Results in Figure 4.3 show total allocated memory for 5 executions of each max-flow algorithm, for $|L|$ varying between 50 and 1000 on random bipartite graphs of average density equal to $0.2$. Subtracting the aforementioned overhead of about 400MB from the original results and performing log-log scale linear regression produced slopes of 2.33 and 2.09 and $r^2$ of 0.9504 and 0.9978 for Dinitz's algorithm

**Figure 4.3:** Square root of allocated memory for the execution of maximum flow algorithms.




and the Edmonds-Karp algorithm respectively. These seem to conform to the space complexity bound of $\mathcal{O}(|V|^2)$ established in Chapter 3.

## 4.2 The GTP algorithm

### 4.2.1 The continuous birth-death tree model

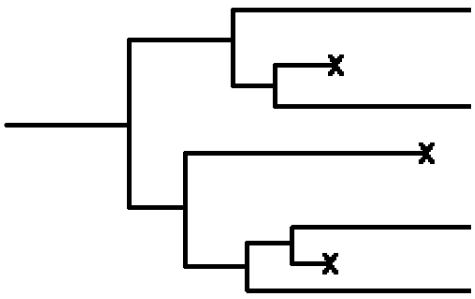
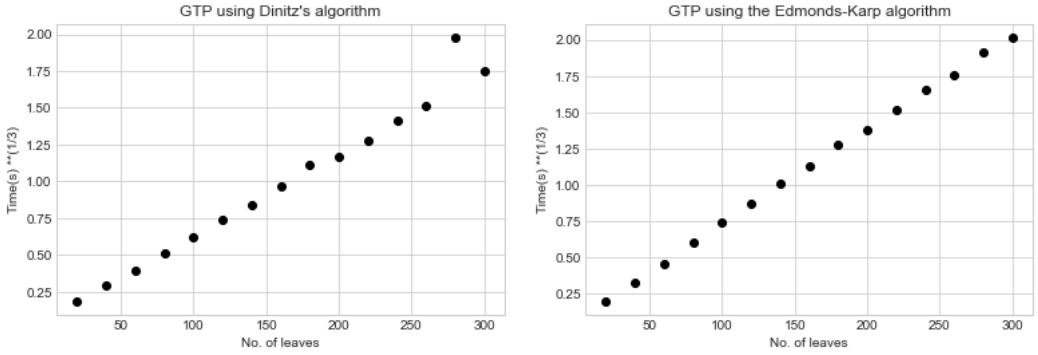


**Figure 4.4:** A birth-death tree with 6 leaves, of which 3 are extinctions.

For our experiments we used trees generated from a continuous time birth-death model. The continuous birth-death model is a continuous time Markov process, which means that given a point in time, probability distributions of future events are exclusively dependent on the current state of the process. In particular, the birth-death process simply starts with a root node and has two possible events, speciation(birth) or extinction(death). The time between two consecutive speciation events is given by an exponential distribution with some parameter $\lambda$, and likewise with parameter $\mu$ for two consecutive ex-

**Figure 4.5:** Cubic roots of execution time of the GTP algorithm using different max-flow algorithms.




tinction events. Here $\lambda$ and $\mu$ are called birth rate and death rate respectively, as they can also be seen as the average number of births or deaths occuring within one time unit. One can view a speciation event as the branching of an edge into two distinct edges, where length denotes time and new edges will comply to a birth-death model, i.e. they will branch or cut off according to the probability distribution of births and deaths respectively. Simulations according to this model were obtained using Python's dendropy package, which allows for the process to stop when there are $n$ tips in tree, where the number of tips is the total number of births. Birth and death rates were fixed and set to $\lambda = 1$ and $\mu = 0$ respectively.

### 4.2.2 Results

All pairwise comparisons were performed between 10 randomly generated birth death trees with $n$ leaves, for $n = 20, 40, ..., 300$. The cubic root of the average execution time for each these pairs is plotted in Figure 4.5. Linear regressions in log-log scale for the data in these plots produces a slope of $2.64$ with $r^2 = 0.9892$ for the version of GTP with Dinitz's algorithm and slope of $2.68$ with $r^2 = 0.9988$ for the Edmonds-Karp version, as expected. If one ignores the obvious outlier for $n = 280$ in the graph corresponding to the Dinitz version of GTP, the linear regression in log-log scale produces slope $2.58$ with $r^2 = 0.9955$.

Allocated memory for the GTP algorithm was also measured for different max-flow algorithms, and the square root of these results are presented in Figure 4.6. Log-log scale linear regression of the original results gave slopes of 1.51 and 1.57 with $r^2 = 0.9798$ and $r^2 = 0.9903$ for the version of GTP using Dinitz's algorithm and the Edmonds-Karp algorithm respectively. Performance of our GTP implementation is once again consistent to the previously established time complexity of $\mathcal{O}(n^4)$ and space complexity of $\mathcal{O}(n^2)$. However, average time complexity seems to conform to $\mathcal{O}(n^3)$, and this may be

**Figure 4.6:** Square root of allocated memory for the GTP algorithm using different max-flow algorithms.

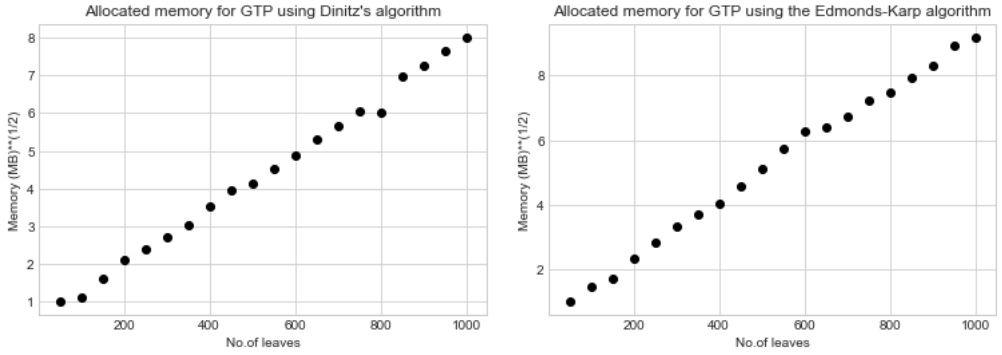
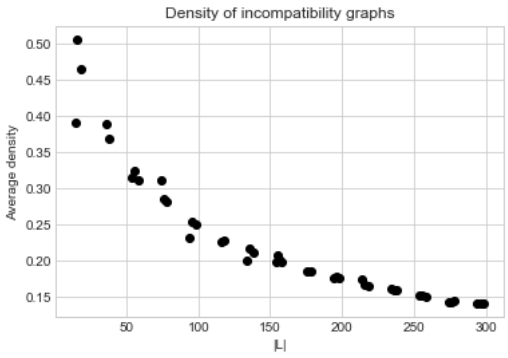



**Figure 4.7:** Average density of incompatibility graphs constructed during the execution of the GTP algorithm.




explained by the number of solutions to the Extension Problem not being as much as assumed in the worst-case analysis done in Chapter 3, among other reasons to be seen ahead.

We simulated the density of the incompatibility graphs arising from the tree set on which the GTP algorithm was executed, so as to provide comparison between these and the random bipartite graphs generated for testing the maximum flow algorithms. It can be inferred from Figure 4.7 that simulation of birth death trees of an increasing number of leaves results in a decrease in the density of the corresponding incompatibility graphs. In fact, log-log scale linear regression shows that the density seems to be inversely proportional to $\sqrt{n}$, resulting in slope approximately equal to -0.52 with corresponding $r^2 = 0.9740$.

This makes sense given that in $\mathcal{T}_n$ there are exactly $(2n-3)!!$ non-identical binary trees, and each has $n-2$ edges. As seen previously, compatibility between two internal edges simply tells us that those edges can coexist in the same tree, i.e. there exists some orthant in $\mathcal{T}_n$ formed by these two edges. Due to the comparatively disproportionate increase in the number of non-identical trees in $\mathcal{T}_n$, the likelihood of any two different edges being compatible increases when $n$ increases. Therefore the

density of incompatibility graphs must decrease, because this density is simply the relative proportion of edges that are incompatible.

The results for the incompatibility graph densities in Figure 4.7 coupled with those from Figure 4.2 put the time measurements of Figure 4.5 in a new light, as they help explain why a theoretical bound of $\mathcal{O}(n^4)$ for the time complexity of the GTP algorithm is too pessimistic. In fact, these results appear to give a stronger guarantee that average-case time complexity is in $\mathcal{O}(n^3)$ for the probability distribution in $\mathcal{T}_n$ given by random birth-death processes. The justification behind this conjecture lies in the theoretical analysis made in Section 3.2 of Chapter 3, which shows that the main contributing factor to the time complexity of the GTP algorithm is the maximum flow algorithm it employs, assuming it is either Dinitz's algorithm or the Edmonds-Karp algorithm.

In other words, let $T_1, T_2 \in \mathcal{T}_n$ and assume that the partitions induced by slicing the trees at their common edges are given respectively by $\{T_1(i)\}_{i=1}^r$ and $\{T_2(i)\}_{i=1}^r$. Let $n_i$ be the number of leaves of $T_1(i)$ and $T_2(i)$. The execution of the GTP algorithm on $T_1$ and $T_2$ leads to the execution of the disjoint GTP algorithm on the pairs of disjoint trees $\{(T_1(i), T_2(i))\}_{i=1}^r$, which in turn performs maximum flow computations on the flow-equivalent networks of corresponding incompatibility graphs. The maximum flow algorithms we studied have general time complexities in $\mathcal{O}(|V||E|)$. The number of internal edges in the trees $T_1(i)$ and $T_2(i)$ is $n_i - 2$, and therefore $|V| = \mathcal{O}(n_i)$ in the flow-equivalent network, ultimately resulting in $|V| = \mathcal{O}(n)$. If one is to believe that $p = \mathcal{O}(1/\sqrt{n})$ where $p$ is the density of the incompatibility graph, then $|E| = \mathcal{O}(\frac{1}{\sqrt{n}} n^2) = \mathcal{O}(n^{3/2})$. Time complexity becomes $\mathcal{O}(n^{5/2})$ for maximum flow algorithms and $\mathcal{O}(n^{7/2})$ for the GTP algorithm, better than the original bound of $\mathcal{O}(n^4)$ of Theorem 10. We also add that the disparity between $7/2$ and the slopes found in log-log scale linear regression - 2.68 and 2.64 for Dinitz and Edmonds-Karp versions of the GTP algorithm respectively - can be explained in part by the disparity between the time complexity of $\mathcal{O}(n^3)$ and the equivalent slopes found for time measurements of the max-flow algorithms (presented in Table 4.1).

The usefulness of computing the geodesic distance is that it allows us to resolve the uncertainty behind reconstructing phylogenetic trees from the genetic code of a set of species. As pointed out in the introduction, this uncertainty is due to the choice of genes/coding regions to represent the species and the choice of algorithm to build the trees based on that information. However, it is reasonable to predict that different trees for a biological dataset will be clustered together, and therefore share a larger set of splits than what would be expected for random birth-death trees. The GTP algorithm benefits from the partition into smaller trees, which means the algorithm may perform better for real phylogenetic trees than for our synthetic datasets.

# 5

# Conclusion

In this thesis we have explored the concept of geodesic paths in tree space and their usefulness in comparing tree structures in the field of phylogenetics. We have also explained how to solve the Geodesic Tree Path problem in polynomial time, and have implemented an efficient algorithm to do so. In particular, this involved understanding the procedure for iterative path finding in tree space and graph problems such as the minimum weight vertex cover, and required the knowledge of the best algorithms and data structures to solve such problems. We theoretically analysed all the chosen algorithms, and proposed original proofs of their time complexity in the given context, while also providing extensive detail and explanation of the most important parts of our code. Throughout all of this, we suggested the incorporation of approximation algorithms to achieve a more desirable time complexity. Finally, we performed exhaustive experiments with synthetic datasets to demonstrate the success of our implementation in comparison with the theoretical analysis. Given the success of our experimental measurements, the existence of better bounds for the time complexity of some of the algorithms discussed is promising. For example, these measurements compare very favourably to the bound of $O(|V||E|)$ we achieved for maximum flow algorithms, particularly in the case of Dinitz's algorithm. Given the similarity of the maximum-flow problem in bipartite flow-equivalent networks and the maximum cardinality bipartite matching problem, the latter of which can be solved with the appropriate reductions by Dinitz's algorithm in time $O(\sqrt{|V|}|E|)$ (see the Hopcroft-Karp algorithm [17]), it is justified to search for a similar bound for the former, since the proof of that time complexity does not apply to all flow-equivalent networks of bipartite graphs. Even more challenging is the implementation of approximate max-flow algorithms such as the ones suggested in [14], due to the considerable theoretical background specific to them, and this is also left as future work.

# Bibliography


[1] B. L. Tavares, “An analysis of the geodesic distance and other comparative metrics for tree-like structures,” 2019.

[2] L. J. Billera, S. P. Holmes, and K. Vogtmann, “Geometry of the space of phylogenetic trees,” *Advances in Applied Mathematics*, vol. 27, no. 4, pp. 733–767, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196885801907596

[3] M. Owen and J. S. Provan, “A fast algorithm for computing geodesic distances in tree space,” 2009.

[4] M. Owen, “Computing geodesic distances in tree space,” 2011.

[5] A. Kupczok, A. von Haeseler, and S. Klaere, “An exact algorithm for the geodesic distance between phylogenetic trees,” *Journal of computational biology : a journal of computational molecular cell biology*, vol. 15, pp. 577–91, 07 2008.

[6] J. Felsenstein, “The Number of Evolutionary Trees,” *Systematic Biology*, vol. 27, no. 1, pp. 27–33, 03 1978. [Online]. Available: https://doi.org/10.2307/2412810

[7] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.

[8] J. Edmonds and R. M. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems,” *J. ACM*, vol. 19, no. 2, p. 248–264, Apr. 1972. [Online]. Available: https://doi.org/10.1145/321694.321699

[9] Y. Dinitz, “Dinitz’ algorithm: The original version and even’s version,” *Lecture Notes in Computer Science*, vol. 3895, pp. 218–240, 01 2006.

[10] U. Zwick, “The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate,” *Theoretical Computer Science*, vol. 148, no. 1, pp. 165–170, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759500022O

[11] A. V. Goldberg and R. E. Tarjan, “Efficient maximum flow algorithms,” *Commun. ACM*, vol. 57, no. 8, p. 82–89, Aug. 2014. [Online]. Available: https://doi.org/10.1145/2628036

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[13] R. E. Tarjan, *Data Structures and Network Algorithms*. USA: Society for Industrial and Applied Mathematics, 1983.

[14] P. F. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S. Teng, “Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs,” *CoRR*, vol. abs/1010.2921, 2010. [Online]. Available: http://arxiv.org/abs/1010.2921

[15] C. Wang, “Faster approximation of max flow for directed graphs,” *ArXiv*, vol. abs/1211.0752, 2012.

[16] “Time complexity of python data types,” https://wiki.python.org/moin/TimeComplexity, accessed: 2021-10.

[17] J. E. Hopcroft and R. M. Karp, “An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs,” *SIAM J. Comput.*, vol. 2, pp. 225–231, 1973.

# Appendix

**Listing A.1:** The disjoint trees version of the GTP algorithm in Python.

```
def disjoint_GTP(T1,T2,algorithm="EK"):
    '''Assuming no common edges'''
    G,weights = incompatibility_graph(T1,T2)
    N = len(T1.taxon_namespace)
    size = len(T1.internal_edges())-1
    nodes = range(2*size+2)
    A, B = nodes[1:size+1], nodes[size+1:-1]
    Al,Bl = [A], [B] #support for cone path
    iteration = 0
    while True:
        iteration += 1
        weight = decimal.Decimal('Infinity')
        new_support = [],[]
        found = False
        for i in range(len(Al)):
            Ai, Bi = Al[i], Bl[i]
            if not found:
                inA = np.full(len(G),False)
                inB = np.full(len(G),False)
                for j in Ai: inA[j] = True
                for j in Bi: inB[j] = True
                if not len(Ai) or not len(Bi):
                    new_support[0].append(Ai)
                    new_support[1].append(Bi)
                    continue
                C, w = capacities(G, Ai, Bi,\
                                  weights, belongsB=inB)
                if algorithm == "EK":
                    F = EK(G, C, vb=verbose)
                elif algorithm == "Dinic":
                    F = Dinic(G, C)
                else:
                    raise Exception("Unknown max flow algorithm")
                cut = reachable(F,0) #characteristic function
                cover, curr = vertex_cover(cut,inA,inB,w)
                weight = min(weight,curr)
                if weight < 1:
```

```
                    C1 = np.nonzero(np.logical_and(inA,cover))[0]
                    C2 = np.nonzero(np.logical_and(inA,np.logical_not(cover)))[0]
                    D2 = np.nonzero(np.logical_and(inB,cover))[0]
                    D1 = np.nonzero(np.logical_and(inB,np.logical_not(cover)))[0]
                    if len(C1)==0 or len(D1)==0 or len(C2)==0 or len(D2)==0:
                        #Solution to Extension Problem not found
                        new_support[0].append(Ai)
                        new_support[1].append(Bi)
                    else:
                        new_support[0].append(C1)
                        new_support[1].append(D1)
                        new_support[0].append(C2)
                        new_support[1].append(D2)
                else:
                    new_support[0].append(Ai)
                    new_support[1].append(Bi)
            else:
                new_support[0].append(Ai)
                new_support[1].append(Bi)
        no_change = (len(Al) == len(new_support[0]))
        Al,Bl = list(new_support[0]),list(new_support[1])
        if weight >= 1 or no_change:
            norm = lambda x: sqrt(sum(weights[i]**2 for i in x))
            return map((lambda x,y: x+y),map(norm,Al),map(norm,Bl))
        Al, Bl = new_support
```