Optimal Offline Fully Dynamic 2-Edge Connectivity

National Technical University of Athens

Ioannis Rizas

Loukas Georgiadis, Dimitrios Fotakis

<MONTH> 2025

Examination Committee

-
-
-

Dedication

To my beloved family—Leonidas, Eleni, Dimitris, and Stergianni— whose love and encouragement have been my constant guide. And to the friends who have enriched my journey.

Acknowledgements

I would like to first express my gratitude to Professor Loukas Georgiadis for his unwavering support and keen interest in dynamic graphs. His valuable insights have greatly enhanced my understanding and expanded my perspective.

I also wish to thank Dimitrios Fotakis, whose exceptional teaching in Discrete Mathematics, Algorithms, and Advanced Algorithms has significantly inspired and motivated me to deepen my passion for Algorithms and Data Structures.

Abstract

Consider an undirected, unweighted graph denoted as $G = (V, E)$, where V represents the set of vertices and E represents the set of edges. We are given a queue with operations of three different types: insertions, deletions, and queries. A query asks whether a given pair of vertices are 2-edge connected. Given all the operations in advance, our objective is to answer all the queries optimally.

Our goal is to implement an algorithmic model that can answer all queries in $O(t \log t)$ time, where $t = \max\{|V| + |E|, |\text{operations}|\}$. This model employs a divide-and-conquer strategy on the queue of operations, allowing us to sparsify the graph information at each recursion. When the size of the remaining operations is $O(1)$, we can answer them in constant time.

Abstract in Greek

Έστω ένα μη κατευθυνόμενο, αβαρές γράφημα, το οποίο συμβολίζεται ως $G = (V, E)$, όπου V είναι το σύνολο των κορυφών και E το σύνολο των ακμών. Επίσης ενυπάρχει μια ουρά με λειτουργίες τριών διαφορετικών τύπων: εισαγωγές, διαγραφές και ερωτήματα (queries). Ένα ερώτημα αφορά το αν ένα ζεύγος κορυφών είναι 2-ακμών συνεκτικό. Δεδομένων όλων των λειτουργιών που θα εφαρμοστούν σε συγκεκριμένη χρονική σειρά εκ των προτέρων, χρειάζεται απαντηθούν όλα τα ερωτήματα στον βέλτιστο χρόνο.

Στόχος είναι η υλοποίηση ενός αλγοριθμικού μοντέλου που να μπορεί να απαντήσει σε χρόνο $O(t \log t)$, όπου $t = \max\{|V| + |E|, |\text{λειτουργίες}|\}$. Αυτό το μοντέλο εφαρμόζει μια στρατηγική διαίρει και βασίλευε στην ουρά των λειτουργιών, επιτρέποντας τη συμπύκνωση της πληροφορίας του γραφήματος σε κάθε επανάληψη. Όταν το μέγεθος των εναπομεινάντων λειτουργιών είναι τετριμμένο, θα μπορούν να απαντηθούν σε σταθερό χρόνο.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Στο κεφάλαιο αυτό θα εξεταστούν οι βασικές έννοιες στατικών και δυναμικών γραφημάτων, καθώς και έννοιες συνεκτικότητας κορυφών και ακμών. Τέλος, θα οριστεί το πλήρως δυναμικό, offline βέλτιστο 2-ακμών συνεκτικό πρόβλημα που θα απασχολήσει την παρούσα εργασία, συνοδευόμενη με μια υλοποίηση στη γλώσσα C++, με πειράματα και αντίστοιχα αποτελέσματα.

Για λόγους απλοποίησης, κάθε γράφημα θεωρείται απλό και μη κατευθυνόμενο.

1.1 Δυναμικά Γραφήματα

Τα γραφήματα αποτελούν ένα βασικό εργαλείο στη θεωρία γράφων και στην υλοποίηση βασικών πρακτικών προβλημάτων. Ένα γράφημα αποτελείται από ένα σύνολο κορυφών που συνδέονται μεταξύ τους με ακμές. Οι κορυφές αναπαρίστανται από σημεία, ενώ οι ακμές από γραμμές ή τμήματα γραμμών που συνδέουν τις κορυφές. Τα γραφήματα μπορούν να είναι κατευθυνόμενα ή μη κατευθυνόμενα, συνδεδεμένα ή ασύνδετα, με βάρη ή ακμές απλές. Ένα γράφημα ορίζεται ως ένα ζεύγος $G = (V, E)$ όπου:

- V: ένα σύνολο κορυφών ή κόμβων.
- E: ένα σύνολο ακμών, που είναι ζευγάρια κορυφών από το V.

Αν κάθε ακμή συνδέεται με ένα βάρος ή κάποιο κόστος, μπορεί να υπάρχει ένας επιπλέον χώρος W που περιέχει τα βάρη.

Τα γραφήματα χωρίζονται σε στατικά και δυναμικά. Ένα στατικό γράφημα παραμένει σταθερό στον χρόνο. Οι κορυφές και οι ακμές του παραμένουν αμετάβλητες καθ' όλη τη διάρκεια. Από την άλλη, ένα δυναμικό γράφημα εξελίσσεται με την πάροδο του χρόνου. Κορυφές και ακμές μπορεί να προστίθενται ή να αφαιρούνται, είτε τα βάρη των ακμών να

μεταβάλλονται. Αυτή η δυνατότητα επιτρέπει την προσέγγιση, την ανάλυση και την επίλυση διαφόρων προβλημάτων της προγαματικής ζωής, όπως προβλήματα διαδικτυακών κοινωνικών δικτύων είτε συστημάτων τηλεπικοινωνιών.

Δύο κατηγορίες ενδιαφέροντος των δυναμικών γραφημάτων είναι τα πλήρως και τα μερικώς δυναμικά γραφήματα. Στα μερικώς δυναμικά (partially dynamic) επιτρέπεται μόνο η προσθήκη κορυφών ή ακμών, ενώ στα πλήρως δυναμικά (fully dynamic) η προσθήκη και η διαγραφή κορυφών ή ακμών.

Επιπλέον, τα προβλήματα πάνω στα δυναμικά γραφήματα μπορούν να χωριστούν σε δύο κύριες κατηγορίες: τα online και τα offline προβλήματα. Στα online προβλήματα, οι λειτουργίες αποκαλύπτονται σειριακά, με την πάροδο του χρόνου, επομένως οι απαντήσεις λαμβάνονται άμεσα, με βάση τις πληροφορίες που είναι διαθέσιμες στην εκάστοτε χρονική στιγμή, χωρίς να υπάρχει εκ των προτέρων γνώση για αυτές. Αντιθέτως, στα offline, το γράφημα και οι λειτουργίες του για ένα χρονικό διάστημα είναι γνωστά εξ'αρχής και ο στόχος είναι να ληφθούν απαντήσεις στα ερωτήματα σε βέλτιστο χρόνο χωρίς την ανάγκη για άμεση ανταπόκριση σε δυναμικές αλλαγές.

Μια εύλογη παρατήρηση είναι ότι μια βέλτιστη offline λύση ενός προβλήματος μπορεί να λειτουργήσει ως μια μετρική για την αποδοτικότητα των online προβλημάτων, καθώς μπορεί να θεωρηθεί μια πολύ καλή προσέγγιση για το online πρόβλημα. Επίσης, για ένα k-ακμών ή κορυφών συνεκτικό πρόβλημα, όσο το k μεγαλώνει, η διαφορά των γνωστών αλγορίθμων offline και online μεγαλώνει ραγδαία. Για τον ορισμό αυτών των προβλημάτων όπως βλέπετε στην επόμενη παράγραφο.

1.2 Συνεκτικότητα

Μια θεμελιώδες έννοια στη θεωρία γράφων είναι η έννοια της συνεκτικότητας. Ένα γράφημα θεωρείται συνεκτικό όταν κάθε ζεύγος κορυφών συνδέεται μέσω τουλάχιστον ενός μονοπατιού. Συνεπώς, όλες οι κορυφές ενός συνεκτικού γραφήματος είναι προσβάσιμες ανά μεταξύ τους, δηλαδή ότι δεν υπάρχουν απομονωμένες ομάδες κορυφών.

Οι έννοιες της k-ακμών και k-κορυφών συνεκτικότητας, αναφέρονται στην ικανότητα ενός γραφήματος να παραμένει συνεκτικό ακόμα και μετά την αφαίρεση $k-1$ ακμών ή κορυφών αντίστοιχα. Κατά τον ορισμό, ένα γράφημα είναι k-ακμών συνεκτικό εάν για οποιοδήποτε ζεύγος κορυφών, αν αφαιρεθεί οποιοδήποτε σύνολο ακμών λιγότερων από k, οι κορυφές παραμένουν συνδεδεμένες μεταξύ τους μέσω τουλάχιστον ενός μονοπατιού. Αντίστοιχα και για την k-κορυφών συνεκτικότητα, αν αφαιρεθεί οποιοδήποτε σύνολο κορυφών λιγότερων από k, οι κορυφές παραμένουν συνδεδεμένες μεταξύ τους μέσω τουλάχιστον ενός μονοπατιού.

Οι πλήρως δυναμικοί αλγόριθμοι γραφημάτων είναι συνήθως δομές δεδομένων που υπο-

στηρίζουν το τρίπτυχο των λειτουργιών: προσθήκη, διαγραφή ακμών και δίνουν απαντήσεις για συγκεκριμένο είδος ερωτημάτων (queries). Για παράδειγμα, για ένα απλό μη-κατευθυνόμενο γράφημα $G(V, E)$, για το πλήρες k-ακμών συνεκτικό πρόβλημα υπάρχουν οι εξής λειτουργίες που επιτρέπονται στην πάροδο του χρόνου:

- $insert(u, v)$: εισαγωγή ακμής (u, v)
- $delete(u, v)$: διαγραφή ακμής (u, v)
- $query(u, v)$: Αν τα u, v είναι k-ακμών συνδεδεμένα

1.3 Υπολογισμός δυσυνεκτικών συνιστωσών

Σε αυτό το κεφάλαιο θα αναλυθούν περιλιπτικά οι μέχρι σήμερα τρόποι επίλυσης βασικών συνεκτικών προβλημάτων για στατικά και δυναμικά γραφήματα.

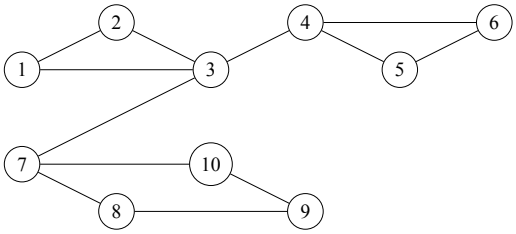
1.3.1 Στατικά Γραφήματα

Για στατικά γραφήματα, είναι γνωστοί οι γραμμικοί αλγόριθμοι για το πρόβλημα της απλής συνεκτικότητας και της δυσυνεκτικότητας, βασισμένοι στην ιδέα της εύρεσης των *γεφυρών* ενός γραφήματος. *Γέφυρα* χαρακτηρίζεται μία ακμή που αν διαγραφεί αποσυνδέεται ένα μέρος του γραφήματος από το υπόλοιπο. Αν σε ένα μη-κατευθυνόμενο γράφημα δεν υπάρχουν γέφυρες, τότε το γράφημα μπορεί να συμπυκνωθεί σε μια *2-ακμών συνεκτική συνιστώσα*, δηλαδή μια συνιστώσα κορυφών όπου κάθε κορυφή του γραφήματος συνδέεται με όλες τις υπόλοιπες με τουλάχιστον δύο διαφορετικά μονοπάτια.

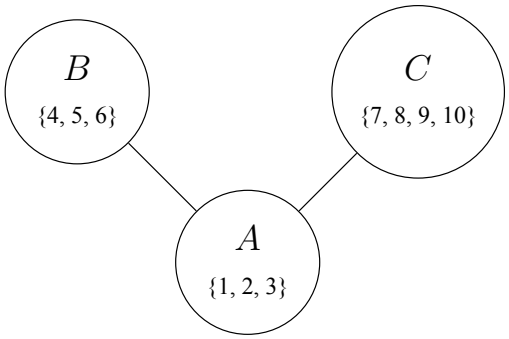
Από την άλλη, αν υπάρχουν γέφυρες, τότε το γράφημα μπορεί νοητικά να μεταφραστεί σε ένα δενδρικό γράφημα και να χωριστεί σε 2-ακμών συνεκτικές συνιστώσες που ενώνονται μεταξύ τους μέσω των γεφυρών. Για να απαντηθεί ένα ερώτημα δυσυνεκτικότητας μεταξύ δύο κορυφών σε στατικό γράφημα, αρκεί να υπολογιστεί το δεντρικό γράφημα με την εύρεση των δυσυνεκτικών συνιστωσών και ο αντιπρόσωπος των δύο κορυφών να είναι κοινός. Δηλαδή να βρίσκονται στην ίδια δυσυνεκτική συνιστώσα. .

Η εύρεση του δένδρου γεφυρών, δηλαδή της εύρεσης των γεφυρών και του μετασχηματισμού του γραφήματος στο ισοδύναμό δενδρικό του γίνεται με δύο DFS περάσματα του γραφήματος, βασισμένα στον αλγόριθμο του Tarjan [9, 10, 12]. Ένα παράδειγμα φαίνεται στο Σχήμα 1.1 -1.2.

Το 2-κορυφών συνεκτικό προβλήμα, από την άλλη, βασίζεται στην ύπαρξη ενός χαρακτηρισμού κορυφών, ως *αρθριτικα σημεία* (articulate point), όπου η αφαίρεσή τους προκαλεί αποσύνδεση του γραφήματος, παρόμοια έννοια με των γεφυρών.

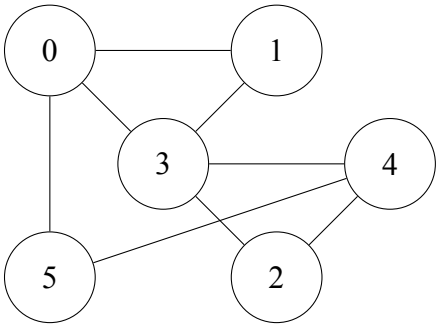


Σχήμα 1.1: Παράδειγμα ενός απλού μη-κατευθυνόμενου γραφήματος

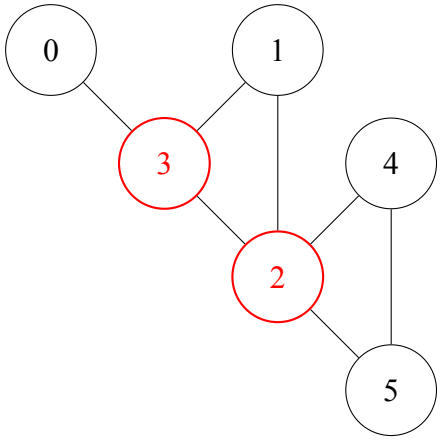


Σχήμα 1.2: Αναπαράσταση του Δεντρικού γραφήματος από το αρχικό

Για παράδειγμα, στον συνεκτικό γράφο που παρουσιάζεται στο Σχήμα 1.1, αν αφαιρεθούν οι κορυφές 3 ή 4, ο γράφος θα διασπαστεί σε δύο συνεκτικές συνιστώσες. Επομένως, οι κορυφές 3 και 4 είναι αρθριτικά σημειά και ο γράφος δεν είναι 2-κορυφών συνεκτικός. Εάν προστεθούν όμως οι ακμές $(0, 1)$, $(0, 5)$ και $(2, 4)$, ο γράφος καθίσταται ως 2-κορυφών συνεκτικός, αφού δεν υπάρχει αρθριτικά σημεία, δηλαδή κορυφές που αν αφαιρεθούν το γράφημα θα σπάσει τη συνεκτικότητά του.



Σχήμα 1.3: Παράδειγμα 2-κορυφών συνεκτικού γράφου



Σχήμα 1.4: Παράδειγμα μη 2-κορυφών συνεκτικού γράφου. Με κόκκινο φαίνονται τα αρθριτικά σημεία, που αν αφαιρεθούν *σπάει* η 2-κορυφών συνεκτικότητα του γράφου

Ο τρόπος επίλυσης του 2-κορυφών συνεκτικού προβλήματος είναι αρχικά η εύρεση των articulate points χρησιμοποιώντας γραμμική αναζήτηση με DFS. Ένας γράφος είναι 2-κορυφών συνεκτικός αν και μόνο αν για κάθε κορυφή του γράφου, υπάρχει τουλάχιστον μία πίσω ακμή που βγαίνει από το υποδέντρο που ριζώνει στην κορυφή u προς κάποιον πρόγονο της κορυφής u. Όταν γίνεται αναφορά σε υποδέντρο ριζωμένο στην κορυφή u, εννοούνται όλοι οι απογόνοι της u (εκτός της ίδιας της κορυφής u). Με άλλα λόγια, όταν γίνει επιστροφή από μια κορυφή u, πρέπει να διασφαλιστεί ότι υπάρχει μια πίσω ακμή από

κάποιον απόγονο της u προς κάποιον πρόγονο (γονέα ή ανώτερο) της u. Υπάρχει μια εξαίρεση για τον ριζικό κόμβο του δέντρου. Εάν ο ριζικός κόμβος έχει περισσότερους από έναν απογόνους, τότε είναι articulate point, αλλιώς όχι. [13]

1.3.2 Δυναμικά Γραφήματα

Στο πλαίσιο της συνεκτικότητας γραφημάτων, για απλή, 2-ακμών/κορυφών συνεκτικότητα, ενώ στα στατικά γραφήματα υπάρχουν πολλές γνωστές βέλτιστες υλοποιήσεις, βασισμένες στο DFS, για τα δυναμικά γραφήματα η κατάσταση διαφοροποιείται.

Αρχικά, στην *online* εκδοχή του προβλήματος, όπου οι λειτουργίες των αλλαγών και των ερωτημάτων πάνω στο γράφημα αποκαλύπτονται σειριακά σε σχέση με τον χρόνο (online) έχουν γίνει κάποιες καλές προσεγγίσεις του προβλήματος οι οποίες προσπαθούν να πλησιάσουν το γνωστό κάτω φράγμα $\Omega(\log n)$ ανά λειτουργεία (per operation), όπου ανέδειξαν οι Patrascu και Demaine [1], χωρίς όμως καμία να μπορεί να λύσει βέλτισα το πρόβλημα, κάτι που για την φύση των online προβλήματων είναι φυσικό επακόλουθο. Το πρώτο μητετριμμένο αποτέλεσμα για απλή συνεκτικότητα πάνω στις ακμές και τις κορυφές, για 2-ακμών/κορυφών συνεκτικότητα είναι: $O(\sqrt{n})$ ανά λειτουργία [2--4].

Μέχρι σήμερα, ο πιο γρήγορος αλγόριθμος για το online συνεκτικό πρόβλημα πάνω στις ακμές έχει χρονική πολυπλοκότητα $O(\log n(\log\log n)^2)$ αποσβεσμένος (amortized) χρόνος ανά λειτουργία [**?**]. Καθώς, για την 2-ακμών/κορυφών συνεκτικότητα, οι χρονικές πολυπλοκότητες είναι $O((\log n)^4)$ και $O((\log n)^5)$ αντίστοιχα [6].

Από την άλλη, για την *offline* εκδοχή του προβλήματος, όπου η σειρά των λειτουργιών είναι γνωστή εκ των προτέρων ως στοιχείο εισόδου του προβλήματος μαζί με το αρχικό γράφο που αυτή εφαρμόζεται, υπάρχουν βέλτιστες αλγοριθμικές λύσεις με $O(\log n)$ ανά λειτουργία. Μία από αυτές είναι και το αλγοριθμικό μοντέλο που απασχολεί αυτή την εργασία και αναλύεται στα επόμενα κεφάλαια.

Κεφάλαιο 2

Εισαγωγή στο Αλγοριθμικό Μοντέλο - Βασικές Υπορουτίνες

2.1 Ορισμοί

Η εργασία θα απασχοληθεί με το offline 2-ακμών συνεκτικό πρόβλημα σε δυναμικά γραφήματα, στο οποίο δίνεται ένα αρχικό γράφημα και μία σειρά από λειτουργίες που εκτελούνται με τη σειρά. Οι λειτουργίες που επιτρέπονται στο πλήρως δυναμικό 2-ακμών συνεκτικό πρόβλημα για ένα ζεύγος κορυφών (u, v) είναι οι εξής:

- insert(u, v): εισαγωγή ακμής (u, v)
- delete(u, v): διαγραφή ακμής (u, v)
- query(u, v): αν οι u, v είναι 2-ακμών συνδεδεμένες

Πιο αναλυτικά, για γράφημα $G = (V, E)$ τη χρονική στιγμή t, θα ερωτάται το πρόγραμμα αν στο γράφημα για ένα ζεύγος κορυφών (u, v), οι κορυφές αυτές είναι 2-ακμών συνδεδεμένες, δηλαδή αν συνδέονται με τουλάχιστον 2 διαφορετικά μονοπάτια μεταξύ τους.

Δεδομένης, λοιπόν, μιας offline σειράς λειτουργιών (operations) και ενός αρχικού γραφήματος $G(V, E)$, ο σκοπός είναι να απαντηθούν όλα τα 2-ακμών ερωτήματα (queries) συνεκτικότητας που υπάρχουν σε βέλτιστο χρόνο.

Τα δεδομένα εισόδου είναι ένα αρχικό απλό γράφημα G και μια σειρά από λειτουργίες με συγκεκριμένη σειρά εκ των προτέρων γνωστή. Για λόγους απλοποίησης, μια ισοδύναμη είσοδος είναι να δίνεται ένα κενό γράφημα με γνωστό το μέγεθος των κορυφών του και μια σειρά από λειτουργίες, όπου στην αρχή υπάρχουν όλες οι ακμές του αρχικού γραφήματος.

Κάθε ακμή έχει κάποια περίοδο ζωής. Για παράδειγμα, μία ακμή e με $I_e = 5$ και $D_e = 100$. Τότε η περίοδος ζωής της είναι το διάστημα $[5, 100]$. Έστω διάστημα $[l, r]$, που αντιπροσωπεύει ένα χρονικό διάστημα πάνω στη σειρά των λειτουργιών, και έστω ακμές όπου

$I_e \leq l \leq r \leq D_e$, δηλαδή που είναι ζωντανές σε αυτό το διάστημα, χωρίς να αλλάξουν, καλούνται *μόνιμες ακμές* (*permanent*) για το διάστημα $[l, r]$. Μη μόνιμες ακμές (*non-permanent*) καλούνται οι ακμές που αλλάζουν στη διάρκεια ενός δοθέντος διαστήματος, δηλαδή υπάρχει τουλάχιστον μία λειτουργία εισαγωγής ή αφαίρεσης ακμής μέσα σε αυτό.

Έστω ένα διάστημα από λειτουργίες και μια σειρά από μόνιμες ακμές που αντιστοιχούν σε αυτό, τότε όλες οι κορυφές που συμμετέχουν έστω και σε μία λειτουργία του δοθέντος διαστήματος καλούνται *ενεργές* (*active*), ενώ αυτές που δεν έχουν καμία συμμετοχή καλούνται *ανενεργές* (*inactive*).

//

//

//ΚΑΠΩΣ ΕΔΩ Η ΚΑΤΑΣΤΑΣΗ ΑΛΛΑΖΕΙ ΦΟΥΛ ΥΦΟΣ

//

//

Λόγω της συνεχούς αλλαγής του γραφήματος, σε κάθε *κατάσταση* (*state*) υπάρχουν κορυφές *αντιπρόσωποι* στις οποίες αντιστοιχούν διάφορες κορυφές του αρχικού γραφήματος. Μια χρήσιμη παρατήρηση που θα σχολιαστεί και αργότερα είναι ότι όταν δύο ή παραπάνω αρχικές κορυφές του γραφήματος έχουν τον ίδιο αντιπρόσωπο, τότε θεωρείται ότι βρίσκονται στην ίδια δυσυνεκτική συνιστώσα. Πολύ παρόμοια λογική με την κατασκευή του δυσυνεκτικού δέντρου του Tarjan που αναφέρθηκε νωρίτερα.

Ο *πίνακας αντιπροσώπων* είναι ένας πίνακας που αποθηκεύει την πληροφορία του αντιπροσώπου κάθε αρχικής κορυφής του γραφήματος. Δηλαδή, για κάθε κορυφή του αρχικού γραφήματος, ο πίνακας περιέχει την κορυφή αντιπρόσωπο που αντιστοιχεί στη 2-ακμών συνεκτική συνιστώσα στην οποία ανήκει.

Σημειώνεται ότι, αν το γράφημα αλλάξει κατά τη διάρκεια της εκτέλεσης του αλγορίθμου, ενδέχεται να χρειαστεί να ανανεωθούν οι κορυφές αντιπρόσωποι. Για να διασφαλιστεί ότι ο πίνακας αντιπροσώπων παραμένει έγκυρος, είναι απαραίτητο να ενημερώνονται οι αντιπρόσωποι των κορυφών ενδιαφέροντος, δηλαδή εκείνων που είναι ενεργές. Η κατάλληλη ανανέωση του πίνακα αντιπροσώπων εξασφαλίζει την ορθή αντιστοίχιση των κορυφών στις νέες 2-ακμών συνεκτικές συνιστώσες που μπορεί να προκύψουν από την τροποποίηση του γραφήματος.

2.2 Θεωρητικό Μοντέλο

R.Peng, B.Sandlund, D.Sleator

Στην μελέτη αυτή θα αναλυθεί ένα μοντέλο (*framework*) [1] επίλυσης πάνω στα συνεκτικά προβλήματα, το οποίο βρίσκει εφαρμογή μέχρι στιγμής από 2-4-ακμών και κορυφών προβλήματα συνεκτικότητας. Η προσέγγιση του προβλήματος με την εφαρμογή αυτού του μοντέλου, μιας λογικής *διαίρει και βασίλευε*, όπου το γράφημα διαιρείται σε μικρότερα ισοδύναμα γραφήματα, με μια τεχνική αραιοποίησης (*sparsification*), όπου μπορεί να μειώνει αρκετώς σε κάθε αναδρομική κλήση την πληροφορία του γραφήματος ώστε να πετυχαίνεται συνολγικά το βέλτιστο κάτω φράγμα των Patrascu και Demaine, που αναφέρθηκε νωρίτερα.

Στην ανάλυση του μοντέλου, όπου t είναι στην ίδια τάξη μεγέθους με το άθροισμα των μεγεθών N, M (αριθμός κορυφών και ακμών του αρχικού γραφήματος) και Z (των λειτουργιών που θα εφαρμοστούν στο γράφο), επιτυγχάνεται χρόνος εκτέλεσης:

$$T(t) = T\left(\frac{t}{2}\right) + O(t) \Rightarrow T(t) = O(t \log t), \quad order(t) = O(N + M + Z)$$

2.3 Βασικές Υπορουτίνες

Στο κεφάλαιο αυτό παρουσιάζονται βασικοί αλγόριθμοι που αποτελούν τα δομικά στοιχεία του αλγοριθμικού μοντέλου για την επίλυση του offline 2-ακμών συνεκτικού προβλήματος. Οι υπορουτίνες αυτές απασχολούνται αρχικά με την διαχείρηση της σειράς των λειτουργιών (operations), καθορίζοντας τον τρόπο με τον οποίο θα εκτελεστούν τα βήματα της επίλυσης. Και κατά δεύτερον, στην διαχείρηση της ανάλυσης και της επεξεργασίας του δοθέντος γραφήματος, εξασφαλίζοντας την κατάλληλη προετοιμασία της δομής του για την εφαρμογή της λύσης.

Όσοι απαραίτητοι αλγόριθμοι περιγράφονται παραπάνω, υπάρχει ο κώδικάς τους στο Κεφάλαιο 5 - παράρτημα - κώδικας αλγορίθμων

2.3.1 Ενεργές κορυφές (active vertices)

Έστω μία σειρά από λειτουργίες, τότε μπορεί να απαντηθεί σε γραμμικό χρόνο για κάθε κορυφή ποιές κορυφές είναι *ενεργές* (active). Μία κουρφή είναι *ενεργή* σε ένα υπο-διάστημα της σειράς των λειτουργίων όταν χρησιμοποιείται από τουλάχιστον μία λειτουργία. Οπως φαίνεται στον παρακάτω Πίνακα 2.1.

Χρόνος	$\cdots$	5	6	7	8	9	10	$\cdots$
Διάστημα	$\cdots$	$I(1,3)$	$D(3,9)$	$Q(2,3)$	$D(9,2)$	$Q(1,9)$	$I(1,2)$	$\cdots$

Πίνακας 2.1: Ενεργές κορυφές: $1, 2, 3, 9$

Ο πιο απλός και αποδοτικός τρόπος είναι ένα γραμμικό πέρασμα στο δοθέν διάστημα των λειτουργιών όπου όποια κορυφή βρίσκεται σημειώνεται ως ενεργή. Όλες οι υπόλοιπες που σημειώθηκαν ως τέτοιες μετά το πέρασμα των λειτουργιών θεωρούνται ανενεργές για αυτό το διάστημα.

2.3.2 Μόνιμες ακμές (permanent edges)

Μόνιμες ακμές σε ένα διάστημα λειτουργιών είναι οι ακμές για τις οποίες δεν υπάρχει κανενός είδους λειτουργία στο διάστημα αυτό (προσθήκης ή αφαίρεσής). Με λίγα λόγια, για ένα συγκεκριμένο διάστημα λειτουργιών αυτές θεωρούνται σταθερές, αφού δεν αλλάζουν σε αυτό.

Για τον υπολογισμό τους χρειάζεται πρώτα να γίνει μια προ-επεξεργασία της σειράς λειτουργιών και για κάθε λειτουργία προσθήκης ή αφαίρεσης ακμής να υπάρχει μαζί και το πότε αυτές αφαιρούνται ή προστίθενται από το γράφο αντίστοιχα. Δηλαδή χρειάζεται για κάθε ακμή να αποθηκευτεί το *διάστημα ζωής* κάθε ακμής, στις δύο λειτουργίες που το καθορίζουν. Πχ. αν μία ακμή (u, v) προστεθεί την *17η* χρονική στιγμή και αφαιρεθεί την *42η*, τότε το διάστημα ζωής που της αναλογεί είναι το $[17, 42]$.

Είναι χρήσιμο η πληροφορία αυτή να μπορεί να είναι διαθέσιμη πάνω στην προσθήκη και την αφαίρεση της ακμής και όχι κάπου αλλού. Συνεπώς χρειάζεται ένας αποδοτικός τρόπος όπου θα παίρνει μία σειρά από λειτουργίες, $3 : (1, 6)$, και θα επιστρέψει ένα ζεύγος από την λειτουργία και έναν αριθμό που αντιπροσωπεύει το άλλο άκρο του διαστήματος ζωής της ακμής, $3 : ((1, 6), 10)$. Δίνεται παράδειγμα

$\cdots$	5	6	7	8	9	10	$\cdots$
$\cdots$	$I(1,3)$	$I(3,9)$	$Q(2,3)$	$D(9,3)$	$I(1,2)$	$D(1,3)$	$\cdots$

Πίνακας 2.2: Παράδειγμα αρχικής σειράς λειτουργιών

$\cdots$	5	6	7	8	9	10	$\cdots$
$\cdots$	$(I(1,3),10)$	$(I(3,9),8)$	$(Q(2,3),-)$	$(D(9,3),5)$	$(I(1,2),24)$	$(D(1,3),5)$	$\cdots$

Πίνακας 2.3: Επαυξημένη σειρά λειτουργιών

Υπάρχουν δύο βασικοί τρόποι επίλυσης του προβλήματος με πολυπλοκότητα ίδιας τάξης με αυτή του μοντέλου υλοποίησης, $O(zlogz)$, όπου z το μέγεθος των λειτουργιών. Ο πρώτος τρόπος είναι με μία απλή ταξινόμηση της σειράς με βάση τα ζεύγη κορυφών, όπου πετυχαίνει ακριβώς η επιθυμητή χρονική πολυπλοκότητα. Ο δεύτερος τρόπος είναι με χρήση πίνακα κατακερματισμού (map) όπου με ένα γραμμικό πέρασμα μπορεί να κατασκευαστεί η επαυξημένη σειρά, σε χρονική πολυπλοκότητα $O(z)$ αποσβεσμένος χρόνος.

Μπορεί η χρήση map να μην συμφωνεί απόλυτα με την θεωρητική πολυπλοκότητα που χρειάζεται να ακολουθηθεί, ωστόσο στην πράξη είναι πολύ πιο ευέλικτος και γρήγορος τρόπος επίλυσης του προβλήματος και ο οποίος χρησιμοποιήθηκε στην υλοποίηση του μοντέλου.

Με δεδομένη την επαυξημένη σειρά λειτουργιών, για ένα υπο-διάστημα ενός αρχικού διαστήματος λειτουργιών, μπορεί να βρεθεί σε γραμμικό χρόνο ποιες ακμές είναι μόνιμες. Στο παράδειγμα του Πίνακα 1.3 για το υπο-διάστημα $[6, 8]$ φαίνεται ότι η μόνιμη ακμή είναι η $(1, 3)$, αφού η προσθήκη είναι πριν από το υπο-διάστημα και η αφαίρεση μετά. Η χρήση του αλγορίθμου υπολογισμού των μόνιμων ακμών θα γίνεται πάντα για ένα υπο-διάστημα ενός αρχικού διαστήματος που ανήκει είτε στο αριστερό είτε στο δεξιό μέρος του αρχικού γράφου. Δηλαδή, αν υπάρχει διάστημα λειτουργιών $[l, r]$, τότε το υπο-διάστημα στο οποίο θα βρεθούν οι μόνιμες ακμές θα είναι:

- δεξιά: $[x, r]$, όπου $x > l$
- αριστερά: $[l, x]$, όπου $x < l$

Οπότε, ο υπολογισμός τους χωρίζεται σε δύο βασικές περιπτώσεις. Αν το υπο-διάστημα είναι στην εξής πλευρά:

- δεξιά, δηλαδή $[x, r]$, τότε γίνεται μία γραμμική αναζήτηση από το $l \rightarrow x$ όπου αν υπάρχει προσθήκη ακμής όπου η χρονική στιγμή της αφαίρεσής της, έστω d, ισχύει ότι: $d > r$, τότε σημειώνεται ως μόνιμη
- αριστερά, δηλαδή $[l, x]$, τότε γίνεται μία γραμμική αναζήτηση από το $x \rightarrow r$ όπου αν υπάρχει αφαίρεση ακμής όπου η χρονική στιγμή της πρσθήκης της, έστω i, ισχύει ότι: $i < l$, τότε σημειώνεται ως μόνιμη

2.3.3 Δυσυνεκτικές Συνιστώσες - Δεντρικός Γράφημα

Αρχικά, είναι απαραίτητη η περιγραφή του αλγορίθμου που χρησιμοποιείται για την εύρεση των γεφυρών σε ένα γράφημα [10]. Βασίζεται στην αναδρομική γραμμική αναζήτηση κατά βάθος (DFS) και εκμεταλλεύεται δύο βασικές έννοιες:

1. Χρόνος Ανακάλυψης (Discovery Time): Ο χρόνος που μια κορυφή ανακαλύπτεται κατά τη διάρκεια της DFS.

2. Χρόνος Χαμηλότερης Επίσκεψης (Low Time): Η μικρότερη τιμή του χρόνου ανακάλυψης που είναι προσβάσιμη από την κορυφή μέσω των γειτονικών της κορυφών.

Algorithm 1 Bridges

```
1:  function compute_bridges(graphG)
2:      Initialize visited, disc, low, and parent arrays
3:      for each vertex i in graphG do
4:          if i is not visited then
5:              bridgeUtil(i, visited, parent, low, disc, graphG)
6:          end if
7:      end for
8:      return results
9:  end function
10:
11: function bridgeUtil(u, visited, parent, low, disc, graphG)
12:     visited[u] ← True
13:     disc[u] ← Time
14:     low[u] ← Time
15:     Time ← Time + 1
16:     for each vertex v adjacent to u do
17:         if v is not visited then
18:             parent[v] ← u
19:             bridgeUtil(v, visited, parent, low, disc, graphG)
20:             low[u] ← min(low[u], low[v])
21:             if low[v] > disc[u] then
22:                 mark edge(u, v) and edge(v, u) as a bridge
23:             end if
24:         else
25:             if v ≠ parent[u] then
26:                 low[u] ← min(low[u], disc[v])
27:             end if
28:         end if
29:     end for
30: end function
```

Ο κύριος τρόπος εύρεσης των δυσυνεκτικών συνιστωσών γίνεται μέσω της εύρεσης του *δέντρου γεφυρών*, όπου καταγράφεται σε ποια δυσυνεκτική συνιστώσα ανήκει κάθε κορυφή. Η διαδικασία αυτή διασφαλίζει ότι το γράφημα χωρίζεται σε δυσυνεκτικές συνιστώσες που δεν περιλαμβάνουν γέφυρες μεταξύ τους, κάνοντας την ανάλυση του γραφήματος πιο αποτελεσματική [11]. Ο γραμμικός αλγόριθμος έχει τα εξής βήματα:

- **Εντοπισμός Γεφυρών**: Στο πρώτο βήμα, ο αλγόριθμος αναγνωρίζει όλες τις γέφυρες του γράφου. Οι γέφυρες είναι κρίσιμες ακμές που συνδέουν μέρη του γράφου, και η αφαίρεσή τους θα προκαλέσει διαχωρισμό του γράφου σε περισσότερα μέρη.

Η ανίχνευση αυτών των γεφυρών είναι απαραίτητη για να διασφαλιστεί ότι οι συνιστώσες του γράφου που θα αναγνωριστούν στη συνέχεια δεν περιλαμβάνουν αυτές τις κρίσιμες ακμές.

- **Αρχικοποίηση Συνιστωσών**: Ο αλγόριθμος αρχίζει με την αναγνώριση των συνιστωσών του γραφήματος, χρησιμοποιώντας DFS. Για κάθε κορυφή που δεν έχει επισκεφθεί, η DFS καλείται με έναν μοναδικό αριθμό για την παρούσα συνιστώσα, όπου ανατίθεται σε όλες τις κορυφές που συνδέονται μέσω αυτής της διαδικασίας. Αυτό βοηθά στη δημιουργία των συνιστωσών χωρίς γέφυρες.
- **Εκτέλεση DFS**: Η DFS αναλαμβάνει την επισκόπηση όλων των γειτονικών κορυφών που δεν έχουν ήδη επισκεφθεί. Κάθε κορυφή που συνδέεται μέσω μιας ακμής που δεν είναι γέφυρα θεωρείται μέρος της ίδιας συνιστώσας. Όλες οι κορυφές που ανήκουν στην ίδια συνιστώσα ενσωματώνονται στην αναγνώριση αυτής της συνιστώσας, και οι συνιστώσες αριθμούνται με μοναδικό τρόπο.
- **Αύξηση Αριθμού Συνιστώσας**: Αφού ολοκληρωθεί η DFS για κάποια ακολουθία κορυφών, που συνδέονται με μονοπάτι χωρίς γέφυρες, ο αριθμός της συνιστώσας αυξάνεται κατά 1, που σημαίνει ότι δεν υπάρχουν άλλες κορυφές στην ίδια δυσυνεκτική συνιστώσα. Αυτό εξασφαλίζει ότι οι επόμενες που θα εξεταστούν θα αναγνωριστούν με διαφορετικό αριθμό συνιστώσας, επιτρέποντας την αναγνώριση ξεχωριστών και διακριτών δυσυνεκτικών συνιστωσών στο γράφημα.

2.3.4 Δυσυνεκτικές Συνιστώσες με Ενεργές Κορυφές

Μια παραλλαγή του προβλήματος εύρεσης δυσυνεκτικών συνιστωσών σε ένα γράφημα αφορά την περίπτωση όπου κάθε κορυφή έχει μία επιπλέον πληροφορία: αν είναι ενεργή (active). Σε αυτή την παραλλαγή, όταν δημιουργείται μια δυσυνεκτική συνιστώσα, αυτή χαρακτηρίζεται ως ενεργή αν τουλάχιστον μία από τις κορυφές της είναι ενεργή. Ο αλγόριθμος για την ανίχνευση αυτών των συνιστωσών είναι μια επέκταση της DFS. Συγκεκριμένα, καθώς η DFS εξερευνά ένα γράφημα, ελέγχει αν μια κορυφή είναι ενεργή. Αν εντοπιστεί τουλάχιστον μία ενεργή κορυφή στη δυσυνεκτική συνιστώσα που δημιουργείται, τότε χαρακτηρίζεται ως ενεργή.

Ο αλγόριθμος επιστρέφει ένα δεντρικό γράφημα με τις κορυφές αντιπροσώπους, που αποτελούν όλες τις δυσυνεκτικές συνιστώσες του γραφήματος, ως καινούργιες κορυφές, για ακμές τις γέφυρες και έναν πίνακα αντιπροσώπων (bridge tree component map), για την αντιστοίχηση των αρχικών κορυφών με τις καινούργιες.

Algorithm 2 Two Edge Connected Components - Bridge Tree

```
1:  function compute_bridge_tree(graphG)
2:      n ← vertices size of the graph
3:      unique_component_number ← 0
4:      Initialize visited arrays, all set to false
5:      //mark the bridges in the graphG
6:      compute_bridges(graphG)
7:      for each vertex i = 1, 2, 3, ..., n do
8:          if visited[i] = false then
9:              bridge_tree_util(i, unique_component_number, visited, graphG)
10:             unique_component_number ← unique_component_number + 1
11:         end if
12:     end for
13: end function
14:
15: function bridge_tree_util(vertex, visited, component_number, graphG)
16:     component[vertex] ← component_number
17:     visited[vertex] ← true
18:     for each next adjacent to vertex do
19:         if edge(next,vertex) is not a bridge and visited[next] = false then
20:             bridge_tree_util(next, visited, component_number, graphG)
21:         end if
22:     end for
23: end function
```

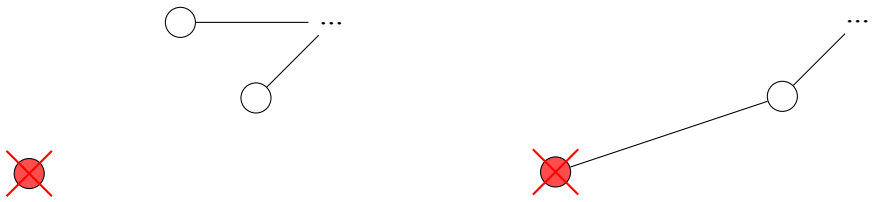
2.3.5 Σύντμηση Δεντρικού Γραφήματος

Η διαδικασία της σύντμησης εφαρμόζεται σε ένα δεντρικό γράφημα που έχει προκύψει από τη διαδικασία της εύρεσης των δυσυνεκτικών συνιστωσών. Στόχος είναι να μειωθεί το γράφημα με γραμμικό τρόπο ως προς το παρόν του μέγεθος. Η σύντμηση του γραφήματος έχει ως αποτέλεσμα ένα μειωμένο γράφημα, στο οποίο οι νέες κορυφές είναι οι αντιπρόσωποι των αρχικών κορυφών του βασικού γραφήματος. Η εκκαθάριση του γραφήματος γίνεται με βάση τους εξής κανόνες:

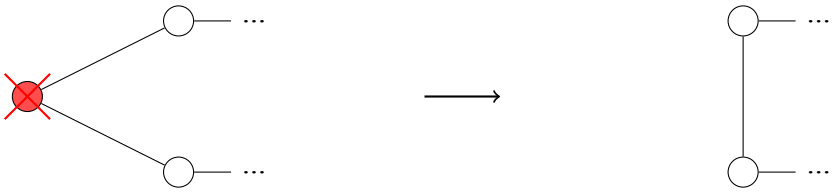
Αν μία κορυφή είναι ανενεργή και έχει βαθμό ≤ 2, τότε αν ο βαθμός είναι:

1. 0 ή 1, αφαιρείται από το γράφημα,
2. 2, ενώνονται οι γείτονες της κορυφής με ακμή και αφαιρείται από το γράφημα

Ο αλγόριθμος επιστρέφει το νέο "κλαδεμένο" δεντρικό γράφημα με τις καινούργιες κορυφές αντιπροσώπους των παλιών κορυφών του και ένα νέο πίνακα αντιπροσώπων (pruning component map), για την αντιστοίχηση αυτή.



Σχήμα 2.1: αφαίρεση ανενεργής κορυφής βαθμού 0 και 1



Σχήμα 2.2: αφαίρεση ανενεργής κορυφής βαθμού 2

Algorithm 3 Tree Pruning

```
1:  function treePruning(graphG, degArray, activeVertices)
2:      size ← size of activeVertices
3:      Initialize visited array with size elements, all set to false
4:      Initialize existedVertices array with size elements, all set to true
5:      for i = 0 to size − 1 do
6:          if not visited[i] then
7:              call treePruningUtil(
8:                  i, graphG, degArray, visited, activeVertices, existedVertices
9:              )
10:         end if
11:     end for
12: end function
13:
14: function treePruningUtil(v,graphG,degArray,visited,activeVertices,existedVertices)
15:     visited[v] ← true
16:     call pruneVertex(v,graphG,degArray, activeVertices,existedVertices)
17:     for each vertex u adjacent to v do
18:         if visited[u] = false and existedVertices[u] = true then
19:             call treePruningUtil(
20:                 u, graphG, degArray, visited, activeVertices, existedVertices
21:             )
22:         else if visited[u] = true and existedVertices[u] = true then
23:             call pruneVertex(v,graphG,degArray, activeVertices,existedVertices)
24:         end if
25:     end for
26:     call pruneVertex(v,graphG,degArray, activeVertices,existedVertices)
27: end function
28:
29: function pruneVertex(v, graphG, degArray, activeVertices, existedVertices)
30:     if degArray[v] = 1 and activeVertices[node] = false then
31:         existedVertices[v] ← false
32:         for each neighbor u of v do
33:             degArray[u] ← degArray[u] - 1
34:         end for
35:     else if degArray[v] = 0 and activeVertices[v] = false then
36:         existedVertices[v] ← false
37:     else if degArray[v] = 2 and activeVertices[v] = false then
38:         existedVertices[v] ← false
39:         for the neighbors u, w of v, add edge(u, w) in graphG
40:     end if
41:     degArray[v] ← 0
42: end function
```

2.3.6 Ανανέωση του πίνακα αντιπροσώπων

Αφού εκτελεστεί ο αλγόριθμος εύρεσης του ισοδύναμου γραφήματος, χρειάζεται να ανανεωθεί κατάλληλα και ο πίνακας των αντιπροσώπων. Σκοπός είναι να ανανεωθεί για κάθε ενεργή αρχική κορυφή ο αρχικός πίνακας των αντιπροσώπων. Έχοντας δηλαδή το:

1. initialComp: ο αρχικός πίνακας αντιστοιχίσεων από τις αρχικές κορυφές, στις κορυφές αντιπροσώπους του παρόντος γραφήματος
2. bridgeTreeComp: ο πίνακας αντιστοιχίσεων μέτα την εύρεση των δυσυνεκτικών συνιστωσών από τις παρούσες κορυφές στις νές κορυφές του δεντρικού γραφήματος
3. pruningComp: ο πίνακας αντιστοιχίσεων μετά το τελικό "κλάδεμα" από τις νές κορυφές αντιπροσώπουε του δεντρικού γραφήματος στις νεότερες και τελικές κορυφές αντιπροσώπους του νέου γραφήματος

Δεν χρειάζεται να ανανεωθεί όλος ο αρχικός πίνακας αντιπροσώπων, παρά μόνο αυτός όπου περιέχει ακόμη ενεργή πληροφορία σε σχέση με τη σειρά λειτουργιών. Αυτός είναι ένας βασικός λόγος για τον οποίο μπορεί να επιτευχθεί η πολυπλοκότητα που αποδεικνύεται στη θεωρητική έρευνα.

Algorithm 4 update components (based on initial active vertices)

```
1: for each vertex v in initialActiveVertices do
2:     initialComp[v] ← pruningComp[bridgeTreeComp[initialComp[v]]]
3: end for
```

Κεφάλαιο 3

Αλγόριθμοι επίλυσης του offline 2-ακμών συνεκτικού προβλήματος

3.1 Εξαντλητική Αναζήτηση (brute force)

Ο αλγόριθμος με τον οποίο θα συγκριθεί το μοντέλο το οποίο αξιολογείται σε αυτή την εργασία έχει χρονική πολυπλοκότητα $O(t^2)$, (λογική είσοδος με $t < 10^4 - 10^5$).Ο αλγόριθμος παίρνει το αρχικό γράφημα και τη σειρά των λειτουργιών και εκτελεί με την σειρά κάθε λειτουργία που του δίνεται. Κάθε λειτουργία έχει την εξής πολυπλοκότητα:

- insert(u, v): $O(1)$
- delete(u, v): $O(t)$
- query(u, v): $O(t)$

Τα ερωτήματα μεταξύ των κορυφών βασίζονται στην εύρεση σε γραμμικό χρόνο του δεντρικού γραφήματος, ή αλλιώς, του δέντρου γεφυρών, όπως περιγράφτηκε νωρίτερα. Πιο συκγκεκριμένα:

- **Εντοπισμός Γεφυρών**: Χρησιμοποιείται ένας αλγόριθμος βασισμένος σε DFS για τον εντοπισμό γεφυρών στο αρχικό γράφημα [10].
- **Κατασκευή του Δεντρικού γραφήματος**: Ένα δέντρο γεφυρών κατασκευάζεται με βάση τις εντοπισμένες γέφυρες. Κάθε κορυφή αντιπρόσωπος αντιστοιχεί σε μία 2-ακμών συνεκτική συνιστώσα του δοθέντος γραφήματος, και οι ακμές του δέντρου αποτελούνται από τις εντοπισμένες γέφυρες. Η κατασκευή του ακολουθεί μία υπάρχουσα υλοποίηση [11].
- **Αποτέλεσμα**: Αν οι δύο κορυφές βρίσκονται στην ίδια 2-ακμών συνεκτική συνιστώσα, δηλαδή ο αντιπρόσωπός τους στο καινούργιο δεντρικό γράφημα είναι ο ίδιος, τότε είναι 2-ακμών συνδεδεμένες. Αλλιώς η απάντηση είναι αρνητική.

Algorithm 5 Brute Force Algorithm

```
1: Input: Initial graph: G, operations sequence: L
2: Output: Query results
3: Initialize: An empty list for the query results: results
4: for each operation in L do
5:     if operation = insert(u, v) then
6:         Insert edge (u, v) into G
7:     else if operation = delete(u, v) then
8:         Delete edge (u, v) from G
9:     else if operation = query(u, v) then
10:        Find bridges in G
11:        Construct the bridge tree
12:        if (u, v) are in the same connected component in the bridge tree then
13:            res ← TRUE
14:        else
15:            res ← FALSE
16:        end if
17:        Add res to results
18:    end if
19: end for
20: return results
```

3.2 Βέλτιστος Αλγόριθμος Επίλυσης 2-ακμών Συνεκτικού Προβλήματος

Δίνεται ως είσοδος ένα γράφημα και μια σειρά από λειτουργίες. Για να εκτελεστεί ο αλγόριθμος με ένα γράφημα κορυφών και ακμών, η ακολουθία των λειτουργιών μπορεί να ενοποιηθεί σε μία ενιαία σειρά εντολών. Αυτό επιτυγχάνεται με την προσθήκη όλων των αρχικών ακμών ως λειτουργίες προσθήκης ακμών στην αρχή της σειράς λειτουργιών, κρατώντας το γράφημα αρχικά κενό, όπως έχει προαναφερθεί.

Συγκεκριμένα, μετά την προ-επεξεργασία, των προσθηκών στη σειρά των εισαγωγών ακμών που αντιπροσωπεύουν 1-1 τις ακμές του αρχικού γραφήματος, με τυχαία σειρά προσθήκης, τα δεδομένα εισόδου είναι:

- ένα κενό αρχικό γράφημα, και
- μία σειρά από λειτουργίες στο χρόνο

Οι κορυφές που συμμετέχουν σε αυτές τις λειτουργίες χαρακτηρίζονται ως ενεργές. Έπειτα, καλείται η συνάρτηση 2-edge-connectivity, σκοπός της οποίας είναι η εύρεση των απαντήσεων όλων των ερωτημάτων για 2-ακμών συνεκτικότητα μεταξύ ζευγαριών κορυφών που συμπεριλαμβάνονται σε όλη την δοθείσα σειρά

1. Ο αλγόριθμος χωρίζει το διάστημα λειτουργιών στη μέση με τη λογική του διαίρει και βασίλευε (divide and conquer)

2. Βρίσκει έπειτα για κάθε μέρος τις ενεργές κορυφές του κάθε διαστήματος
3. Βρίσκει τις μόνιμες ακμές για κάθε μέρος και τις προσθέτει στο αντίστοιχο γράφημα που αντιστοιχεί σε κάθε μέρος
4. Υπολογίζει το ισοδύναμο γράφημα του υπάρχοντος γραφήματος, που περιγράφεται παρακάτω με τη χρήση όλων των υπο-ρουτινών που αναλύθηκαν στο προηγούμενο κεφάλαιο
5. Έχοντας το ισοδύναμο γράφημα και το διάστημα λειτουργιών που του αντιστοιχεί, καλείται η η συνάρτηση πάλι αναδρομικά
6. έως ότου το μέγεθος του διαστήματος των λειτουργιών να γίνει τετριμμένο, έτσι ώστε τα ερωτήματα να μπορούν να απαντηθούν σε σταθερό χρόνο, κοιτώντας αν οι δύο κορυφές ανήκουν στην ίδια κορυφή αντιπρόσωπο του νέου γραφήματος

Η διαδικασία δημιουργίας του ισοδύναμου γραφήματος από το αρχικό γράφημα περιγράφεται σε τέσσερα στάδια, ως σύνθεση των υπο-ρουτίνων που περιγράφτηκαν στο Κεφάλαιο 2. Ο αλγόριθμος ακολουθεί τα εξής βήματα:

- bridges-stage: Εντοπισμός γεφυρών
- bridge-tree-stage: Κατασκευή του Δεντρικού Γραφήματος
- pruning-stage: Συνέχιση Σύντμησης - "Κλάδεμα" Δέντρου
- update-components-stage: Ανανέωση του πίνακα των αντιπροσώπων (μετά την τελική σύντμηση του γραφήματος)

Algorithm 6 Two Edge Connectivity

```
1:  procedure two_edge_connectivity(operations, graph)
2:      augmented_operations:
3:          compute the augmented operations (graph now is empty)
4:      results:
5:          initialize an empty list for the queries' results
6:      component_map:
7:          define an empty array
8:      // initialize the components
9:      for each vertex v in graph do
10:         component[v] = v
11:     end for
12:     call compute_two_edge_connectivity (
13:         augmented_operations,
14:         graph,
15:         component_map
16:     )
17: end procedure
```

Algorithm 7 compute two edge connectivity

```
1: function compute_two_edge_connectivity(operations, graph, component_map, results)
2:     if |operations| = 1 and operation's type = query then
3:         if component_map[u] = component_map[v] then
4:             res ← true
5:         else
6:             res ← false
7:         end if
8:         Add res to results
9:     else
10:        // split operations into two halves
11:        for each half do
12:            half_graph:
13:                a copy of the graph
14:            half_component_map:
15:                a copy of the component_map
16:            half_operations:
17:                compute half operations
18:            half_active_vertices:
19:                compute active vertices for the half_operations
20:            half_component_active_vertices:
21:                compute component active vertices from half_active_vertices
22:                through component_map
23:            half_permanent_edges:
24:                compute permanent edges
25:            half_component_permanent_edges:
26:                compute component permanent edges from half_permanent_edges
27:                through component_map
28:            add half_component_permanent_edges to graph
29:            call compute_equivalent_graph(
30:                half_graph,
31:                half_active_vertices,
32:                half_component_active_vertices,
33:                half_component_map
34:            )
35:            call compute_two_edge_connectivity (
36:                half_operations,
37:                half_graph,
38:                half_component_map
39:            )
40:        end for
41:    end if
42: end function
```

Algorithm 8 compute the equivalent graph

```
input:
    graph - The initial graph.
    original_active_vertices - List of original active vertices.
    current_active_vertices - List of current active vertices.
    component_map - Mapping of components.
output:
    pruned-graph - The sparse equivalent graph.
    component_map - Updated component map.

procedure compute_equivalent_graph
    function bridges stage
        Find all bridges in graph
    end function
    function 2-edge connected components stage
        Construct the bridge − tree − graph from the bridges
        for each vertex original_vertex in graph do
            bridge_tree_mapping[original_vertex] = new_vertex
        end for
        Find bridge_tree − active_vertices
    end function
    function pruning stage
        Construct the pruned − graph from bridge-tree-graph and
        bridge_tree-active_vertices
        for each vertex v in bridge-tree-graph do
            pruning_mapping[v] = pruned_vertex
        end for
    end function
    function update components stage
        Update component_map
        for each vertex v in original_active_vertices do
            component_map[v] =
              pruning_mapping[bridge_tree_mapping[component_map[v]]]
        end for
    end function
    return pruned-graph, component_map
end procedure
```

Κεφάλαιο 4

Πειράματα και Αποτελέσματα

4.1 Κατασκευή Περιπτώσεων Ελέγχου

Η κατασκευή των περιπτώσεων ελέγχου του μοντέλου υλοποίησης του αλγορίθμου έγινε ως εξής:

- **Αρχικό γράφημα:** Παράχθηκαν απλά γραφήματα τάξης από $[10^1 - 10^6]$.
- **Λίστα λειτουργιών:** Πάνω σε κάθε αρχικό γράφημα δημιουργήθηκε και μια λίστα από λειτουργίες όπου δεν παραβιάζει τους κανόνες του γραφήματος, τάξης ίσης του αρχικού γραφήματος.

Για την αποθήκευση του γραφήματος χρησιμοποιήθηκε πίνακας διανυσμάτων, όπου είναι γνωστό ότι οι διαγραφές είναι της τάξης $O(n)$. Για να μπορέσει το μοντέλο ωστόσο να επιτύχει την πολυπλοκότητα που ζητείται, με την ιδέα που αναφέρθηκε, ο αλγόριθμος απαντά τα 2-ακμών συνεκτικά ερωτήματα μεταξύ δύο κορυφών χωρίς ποτέ να χρειαστεί να εκτελεστεί αφαίρεση κορυφής.

4.2 Πειράματα: TODO

4.3 Αποτελέσματα: TODO

Βιβλιογραφία

[1] Richard Peng, Bryce Sandlund, Daniel D. Sleator, *Optimal Offline Dynamic 2,3-Edge/Vertex Connectivity*, 2019. https://arxiv.org/abs/1708.03812

[2] Patrascu, Demaine, *Lower Bounds for Dynamic Connectivity*, 2004. https://people.csail.mit.edu/mip/docs/enc-algs/connect.pdf

[3] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer, *Separator Based Sparsification: I. Planarity Testing and Minimum Spanning Trees*, J. Comput. Syst. Sci., 52(1):3--27, 1996. https://sciencedirect.com/science/article/pii/S0022000096900096

[4] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer, *Separator-Based Sparsification II: Edge and Vertex Connectivity*, SIAM J. Comput., 28(1):341--381, 1998. https://epubs.siam.org/doi/10.1137/S0097539793244356

[5] Greg N. Frederickson, *Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees*, SIAM J. Comput., 26(2):484--538, 1997. https://epubs.siam.org/doi/10.1137/S0097539791191374

[6] Kathrin Hanauer, Monika Henzinger, Christian Schulz, *Recent Advances in Fully Dynamic Graph Algorithms*, 2022. https://arxiv.org/abs/2102.11169

[7] Jacob Holm, Kristian de Lichtenberg, Mikkel Thorup, *Poly-Logarithmic Deterministic Fully Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity*, SIAM J. Comput., 31(3):720--740, 2001. https://dl.acm.org/doi/10.1145/502090.502095

[8] Mehmet Koyutürk, *Algorithmic and Analytical Methods in Network Biology*, 2010. https://wires.onlinelibrary.wiley.com/doi/10.1002/wsbm.61

[9] Jimmy Wu, Alex Khodaverdian, Benjamin Weitz, Nir Yosef, *Connectivity Problems on Heterogeneous Graphs*, 2019. https://almob.biomedcentral.com/articles/10.1186/s13015-019-0141-z

[10] *Find Bridges in a Graph Using Tarjan's Algorithm*. https://www.geeksforgeeks.org/bridge-in-a-graph/

[11] *Bridge Trees Algorithm - Implementation in C++*. https://codeforces.com/blog/entry/99259

[12] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie, *Fully Dynamic Connectivity in O(log n(log log n)²) Amortized Expected Time*, 2022. https://arxiv.org/pdf/2102.11169.pdf

[13] R. E. Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM Journal on Computing, 1(2):146--159, 1972.