Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Λογικης και Επιστημης Υπολογισμων

Small Vertex Cut on the PRAM model

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλειος-Χρήστος Ξανθόπουλος

Επιβλέπων: Αριστείδης Παγουρτζής
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Small Vertex Cut on the PRAM model

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλειος-Χρήστος Ξανθόπουλος

Επιβλέπων: Αριστείδης Παγουρτζής
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16/10/2024.

.....	.....	.....
Αριστείδης Παγουρτζής	Δημήτριος Φωτάκης	Νικόλαος Λεονάρδος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2024

.....
Βασίλειος-Χρήστος Ξανθόπουλος
(Διπλωματούχος Ηλεκτρολόγος Μηχανικός & Μηχανικός Υπολογιστών Ε.Μ.Π.)

Οι απόψεις που εκφράζονται σε αυτό το κείμενο είναι αποκλειστικά του συγγραφέα και δεν αντιπροσωπεύουν απαραίτητα την επίσημη θέση του Εθνικού Μετσόβιου Πολυτεχνείου.

Απαγορεύεται η χρήση της παρούσας εργασίας για εμπορικούς σκοπούς.

Περίληψη

Το πρόβλημα Vertex Cut αφορά την εύρεση του ελάχιστου συνόλου κορυφών, η αφαίρεση των οποίων αποσυνδέει ένα γράφημα σε πολλαπλές συνεκτικές συνιστώσες. Είναι ένα θεμελιώδες πρόβλημα στην επιστήμη υπολογιστών καθώς έχει πολυάριθμες εφαρμογές, όπως ο σχεδιασμός ανθεκτικών σε σφάλματα δικτύων, οι παράλληλοι υπολογισμοί και τα κατανεμημένα συστήματα. Καθώς το μέγεθος και η πολυπλοκότητα των δεδομένων αυξάνονται εκθετικά, η αντιμετώπιση του προβλήματος κοπής κορυφών σε ένα παράλληλο υπολογιστικό περιβάλλον καθίσταται απαραίτητη για την επίτευξη κλιμακούμενων και αποδοτικών λύσεων.

Σε αυτή τη διπλωματική εργασία εξετάζουμε το πρόβλημα Vertex Cut στο πλαίσιο των παράλληλων υπολογισμών, και συγκεκριμένα στο μοντέλο PRAM. Ξεκινάμε παρουσιάζοντας διάφορες προαπαιτούμενες έννοιες, όπως το μοντέλο PRAM, την ανάλυση πολυπλοκότητας παράλληλων αλγορίθμων στο μοντέλο work/depth και θεμελιώδη αποτελέσματα στο μοντέλο PRAM και το πρόβλημα vertex connectivity. Στη συνέχεια, παρουσιάζουμε τους κλασικούς αλγόριθμους για την επίλυση του προβλήματος στο σειριακό μοντέλο RAM, και ακολουθούμε την πορεία εργασιών των Nanongkai, Thatchaphol και Yingchareonthawornchai [1], οι οποίοι ήταν οι πρώτοι που έσπασαν το τετραγωνικό φράγμα για την πολυπλοκότητα του πρόβληματος.

Για να μεταφέρουμε αυτές τις ιδέες στο παράλληλο μοντέλο, αντιμετωπίζουμε το εμπόδιο της παράλληλης προσπελασιμότητας, το οποίο επιλύουμε χρησιμοποιώντας τεχνικές από την εργασία των Sidford, Jambulapati και Liu [2]. Συγκεκριμένα, κατασκευάζουμε hopsets για να μειώσουμε τη διάμετρο του γραφήματος, καθιστώντας το προσπελάσιμο με μικρό depth. Τέλος, συνδυάζουμε αυτές τις ιδέες και παρουσιάζουμε δύο αλγορίθμους με work/depth tradeoff για αυτό το πρόβλημα και εξετάζουμε πιθανές κατευθύνσεις για μελλοντική έρευνα.

Λέξεις κλειδιά: Πρόβλημα Κοπής Κορυφών, Μοντέλο PRAM, Παράλληλοι Αλγόριθμοι, Θεωρία Γραφημάτων, Παράλληλη Προσπελασιμότητα, Συνδεσιμότητα Γραφημάτων, Τυχαιοποιημένοι Αλγόριθμοι, Σχεδίαση Αλγορίθμων, Αλγόριθμοι Γραφημάτων, Δίκτυα Ανθεκτικά σε Σφάλματα

Abstract

The Vertex Cut Problem involves identifying the minimum set of vertices whose removal disconnects a given graph into multiple connected components. Its significance stems from its wide-ranging applications, including fault-tolerant network design, parallel computation, and distributed systems. As the size and complexity of datasets continue to grow exponentially, addressing the Vertex Cut Problem in a parallel computing environment becomes imperative for achieving scalable and efficient solutions.

In this thesis, we examine the Vertex Cut Problem in the context of parallel computation, specifically within the PRAM model. We begin by introducing various preliminaries, including the PRAM model itself, work/depth complexity analysis, and fundamental results in PRAM and vertex connectivity. We then present the folklore algorithms for solving this problem in the sequential setting, before building upon the work of Nanongkai, Thatchaphol and Yingchareonthawornchai [1], whose paper was the first to break the quadratic barrier for this problem.

To transfer these ideas into the parallel regime, we face the challenge of parallel reachability, which we address using techniques from Sidford, Jambulapati και Liu's work [2]. Specifically, we construct hopsets to reduce the graph's diameter, enabling efficient traversal in low depth. Finally, we combine these ideas to present two algorithms that offer a work-depth tradeoff for this problem. The thesis concludes by exploring potential research directions, further improving our understanding of parallel algorithms for vertex cuts.

Keywords: Vertex Cut Problem, PRAM Model, Parallel Computing, Work-Depth Tradeoff, Parallel Algorithms, Graph Theory, Parallel Reachability, Hopsets, Graph Connectivity, Randomized Algorithms, Algorithmic Design, Graph Algorithms, Fault-tolerant Networks

Ευχαριστίες

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο πλαίσιο ενός εξαμηνιαίου προγράμματος έρευνας στο ερευνητικό κέντρο Max Planck Institute for Informatics, από τον Φεβρουάριο έως τον Αύγουστο του 2023, υπό την επίβλεψη των Danupon Nanongkai και Yonggang Jiang.

Ολοκληρώνοντας αυτή την εργασία και, κατ' επέκταση, τις σπουδές μου στο ΕΜΠ, θα ήθελα να ευχαριστήσω τους ανθρώπους που με υποστήριξαν και με βοήθησαν να φτάσω ως εδώ.

Αρχικά, ευχαριστώ τον επιβλέποντα καθηγητή της διπλωματικής μου εργασίας, κ. Παγουρτζή, ο οποίος, μαζί με τον κ. Φωτάκη και τον κ. Παπασπύρου, μέσα απο τη διδασκαλία τους μου καλλιέργησαν την αγάπη για τον κλάδο της θεωρητικής επιστήμης υπολογιστών. Επιπλέον θα ήθελα να ευχαριστήσω τους Danupon Nanongkai και Yonngang Jiang, χωρίς τη βοήθεια των οποίων η ολοκλήρωση αυτής της διπλωματικής εργασίας δεν θα ήταν εφικτή.

Θα ήθελα, επίσης, να εκφράσω την ευγνωμοσύνη μου στους ανθρώπους που μοιράστηκαν μαζί μου όλη ή έστω ένα μέρος αυτής της διαδρομής. Ένα μεγάλο ευχαριστώ στους φίλους και την οικογένεια μου, που με στηρίζουν ανιδιοτελώς σε κάθε μου προσπάθεια και στόχο.

Στην οικογένεια μου

Contents

List of Figures

List of Algorithms

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

Σε αυτό το κεφάλαιο παρουσιάζουμε περιληπτικά τα περιεχόμενα αυτή της διπλωματικής εργασίας στα ελληνικά. Εισάγουμε όλες τις βασικές έννοιες που εμφανίζονται στο αγγλικό κείμενο. Ωστόσο, δεν δίνουμε ούτε αποδείξεις ούτε τεχνικές λεπτομέρειες. Αυτές δίνονται εκτενώς στα επόμενα κεφάλαια.

1.1 Το Μοντέλο PRAM

Σε αυτή την ενότητα παρουσιάζονται κάποιες γενικές ιδέες που διέπουν την περιοχή των Παράλληλων Αλγορίθμων. Το μοντέλο PRAM (Parallel Random Access Machine) είναι ένα από τα πιο εκτενώς μελετημένα, και το επικρατέστερο, μοντέλο στον τομέα της παράλληλης επεξεργασίας. Αποτελεί το αντίστοιχο του παραδοσιακού μοντέλου σειριακής επεξεργασίας, αυτό της μηχανής τυχαίας πρόσβασης (RAM), σχεδιασμένο να προσομοιώνει τη συμπεριφορά πολλαπλών επεξεργαστών που μπορούν να έχουν ταυτόχρονη πρόσβαση σε έναν κοινό χώρο μνήμης. Στο μοντέλο PRAM, οι επεξεργαστές λειτουργούν συγχρονισμένα, δηλαδή εκτελούν λειτουργίες ταυτόχρονα, και η επικοινωνία μεταξύ τους επιτυγχάνεται μέσω της κοινής μνήμης. Αυτό καθιστά το PRAM ένα ιδανικό μοντέλο για την ανάλυση στρατηγικών παραλληλοποίησης για διάφορες υπολογιστικές εργασίες.

Το μοντέλο PRAM χωρίζεται σε τέσσερα υπομοντέλα, ανάλογα με το πώς αντιμετωπίζονται οι συγκρούσεις κατά την πρόσβαση στη μνήμη:

- **Exclusive read exclusive write (EREW):** Σε αυτό το μοντέλο, μόνο ένας επεξεργαστής επιτρέπεται να διαβάζει από ή να γράφει σε μια θέση μνήμης κάθε φορά, καθιστώντας το το πιο περιοριστικό μοντέλο PRAM. Το μοντέλο EREW

είναι δύσκολο να υλοποιηθεί, αλλά εξασφαλίζει ότι δεν υπάρχουν συγκρούσεις πρόσβασης στη μνήμη.

- **Concurrent read exclusive write (CREW):** Πολλοί επεξεργαστές επιτρέπεται να διαβάζουν ταυτόχρονα από την ίδια θέση μνήμης, αλλά μόνο ένας επεξεργαστής μπορεί να γράφει σε αυτήν ανά πάσα στιγμή. Αυτό το μοντέλο επιτρέπει την παράλληλη ανάγνωση, η οποία είναι πιο ευέλικτη από την EREW, αλλά εξακολουθεί να περιορίζει τις λειτουργίες εγγραφής.
- **Exclusive read concurrent write (ERCW):** Αν και αυτό το μοντέλο επιτρέπει σε πολλούς επεξεργαστές να γράφουν στην ίδια θέση μνήμης, σπάνια εξετάζεται, καθώς δεν προσθέτει σημαντική υπολογιστική ισχύ ή αποδοτικότητα στις περισσότερες εφαρμογές, καθώς δεν επιτρέπει την ταυτόχρονη ανάγνωση.
- **Concurrent read concurrent write (CRCW):** Αυτό είναι το πιο γενικό μοντέλο, όπου πολλαπλοί επεξεργαστές μπορούν να διαβάζουν και να γράφουν ταυτόχρονα στην ίδια θέση μνήμης. Διαφορετικές παραλλαγές του μοντέλου CRCW καθορίζουν τον τρόπο χειρισμού των συγκρούσεων εγγραφής (π.χ., το αποτέλεσμα μιας σύγκρουσης εγγραφής μπορεί να καθορίζεται από την ελάχιστη ή τη μέγιστη τιμή που γράφεται ή μπορεί να καθορίζεται τυχαία). Μια CRCW PRAM αναφέρεται μερικές φορές ως Concurrent Random Access Machine (μηχανή ταυτόχρονης τυχαίας προσπέλασης) λόγω της ικανότητάς της να χειρίζεται ταυτόχρονες λειτουργίες τόσο κατά την ανάγνωση όσο και κατά την εγγραφή.

Το μοντέλο που θα χρησιμοποιήσουμε σ' αυτή την εργασία είναι το CREW.

1.1.1 Ανάλυση Πολυπλοκότητας Work/Depth

Στην επιστήμη των υπολογιστών, η ανάλυση των παράλληλων αλγορίθμων περιλαμβάνει τον προσδιορισμό της υπολογιστικής πολυπλοκότητας των αλγορίθμων, αξιολογώντας τους πόρους που απαιτούνται για την εκτέλεση τους, συμπεριλαμβανομένου του χρόνου, της μνήμης ή άλλων πόρων. Ενώ η ανάλυση είναι παρόμοια με αυτή των σειριακών αλγορίθμων, η διαδικασία είναι πιο πολύπλοκη επειδή απαιτεί την εξέταση πολλαπλών κλάδων εκτέλεσης.

Ένας βασικός στόχος στην ανάλυση παράλληλων αλγορίθμων είναι να εκτιμηθεί πώς η χρήση των πόρων του αλγορίθμου, όπως η ταχύτητα και ο χώρος, επηρεάζεται από διαφορετικές τιμές του αριθμού των επεξεργαστών. Έστω ότι εκτελούμε υπολογισμούς σε μια μηχανή με p επεξεργαστές και έστω T_p ο χρόνος που απαιτείται από την έναρξη έως το τέλος του υπολογισμού. Η ανάλυση του χρόνου εκτέλεσης τέτοιων υπολογισμών περιλαμβάνει τις ακόλουθες βασικές έννοιες:

- **Work**: Το work ενός αλγορίθμου που εκτελείται από p επεξεργαστές είναι ο συνολικός αριθμός των θεμελιωδών πράξεων που εκτελούνται από τους επεξεργαστές,

κατ' αντιστοιχία με τη χρονική πολυπλοκότητα στην ανάλυση σειριακών αλγορίθμων. Χωρίς να ληφθεί υπόψη η επιβάρυνση της επικοινωνίας λόγω συγχρονισμού, αυτό ισοδυναμεί με τον χρόνο που απαιτείται για να εκτελεστεί ο υπολογισμός σε έναν μόνο επεξεργαστή, T_1.

- **Depth or Span**: Το depth ή span αναφέρεται στην πιο μακριά ακολουθία λειτουργιών που πρέπει να εκτελεστούν σειριακά λόγω εξαρτήσεων (συχνά αναφέρεται ως κρίσιμο μονοπάτι). Η ελαχιστοποίηση του βάθους είναι κρίσιμη κατά το σχεδιασμό παραλλήλων αλγορίθμων, καθώς το βάθος καθορίζει τον ελάχιστο χρόνο που απαιτείται για την εκτέλεση. Εναλλακτικά, το βάθος είναι ο χρόνος T_∞ που απαιτείται από μια ιδανική μηχανή με άπειρο αριθμό επεξεργαστών.
- **Cost**: Το συνολικό κόστος ενός υπολογισμού δίνεται από το pT_p, το οποίο αντιπροσωπεύει το συνολικό χρονικό διάστημα που αφιερώνουν όλοι οι επεξεργαστές, συμπεριλαμβανομένου του χρόνου υπολογισμού και του χρόνου αδράνειας.

Theorem 1.1.1. *(Work Law): Το συνολικό κόστος είναι πάντα τουλάχιστον ίσο με το Work: $pT_p \geq T_1$. Αυτό προκύπτει από το γεγονός ότι οι p επεξεργαστές μπορούν να εκτελέσουν το πολύ p λειτουργίες ταυτόχρονα.*

Theorem 1.1.2. *(Span Law): Ένας πεπερασμένος αριθμός p επεξεργαστών δεν μπορεί να υπερβεί έναν άπειρο αριθμό επεξεργαστών, άρα $T_p \geq T_\infty$.*

Βάσει των παραπάνω, μπορούμε να ορισουμε τις εξής μετρικές απόδοσης:

Definition 1.1.1 (Speedup S_p)**.** Το speedup είναι ο λόγος του χρόνου που απαιτείται για τη σειριακή εκτέλεση του αλγορίθμου προς τον χρόνο που απαιτείται για την παράλληλη εκτέλεση: $S_p = \frac{T_1}{T_p}$.

Αν $S_p = \Omega(p)$ για p επεξεργαστές, θεωρούμε ότι ο αλγόριθμος έχει γραμμική επιτάχυνση, η οποία είναι βέλτιστη, διότι, βάσει του work law, $S_p \leq p$. Η περίπτωση όπου $S_p = p$ είναι γνωστή ως τέλεια γραμμική επιτάχυνση.

Definition 1.1.2 (Efficiency E_p)**.** Το efficiency ορίζεται ως το speedup ανά επεξεργαστή, δηλαδή $E_p = \frac{S_p}{p}$.

Definition 1.1.3 (Parallelism)**.** Το parallelism είναι ο λόγος $\frac{T_1}{T_\infty}$, που αντιπροσωπεύει τη μέγιστη δυνατή επιτάχυνση με οποιονδήποτε αριθμό επεξεργαστών.

Βάσει του span law, το speedup περιορίζεται από το parallelism: αν $p > \frac{T_1}{T_\infty}$, τότε:

$$\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} < p.$$

Definition 1.1.4 (Slackness)**.** Το slackness ορίζεται ως $\frac{T_1}{pT_\infty}$.

Αν η αδράνεια είναι μικρότερη από ένα, η τέλεια γραμμική επιτάχυνση είναι αδύνατη, σύμφωνα με το span law.

Στο πλαίσιο του μοντέλου PRAM, ένας αποδοτικός παράλληλος αλγόριθμος στοχεύει στην ελαχιστοποίηση τόσο της εργασίας όσο και του βάθους για την επίτευξη βέλτιστης απόδοσης όταν εκτελείται σε πεπερασμένο αριθμό επεξεργαστών.

1.2 Τυχαιοποιημένοι Αλγόριθμοι

Οι τυχαίοι αλγόριθμοι είναι αλγόριθμοι που κάνουν τυχαίες επιλογές κατά την εκτέλεσή τους για να επιτύχουν καλή απόδοση κατά μέσο όρο ή με μεγάλη πιθανότητα. Αποτελούν ένα ισχυρό εργαλείο στη σχεδίαση αλγορίθμων, οδηγώντας συχνά σε απλούστερες, ταχύτερες ή αποδοτικότερες λύσεις σε σύγκριση με τους αντίστοιχους ντετερμινιστικούς αλγορίθμους. Μερικές φορές αυτοί οι αλγόριθμοι μπορούν να «αποτυχαιοποιηθούν», σε κλασικούς ντετερμινιστικούς αλγορίθμους αν και συνήθως οι αλγόριθμοι που προκύπτουν είναι συχνά πιο περίπλοκοι. Σε ορισμένα μοντέλα υπολογισμού, μπορεί κανείς ακόμη και να αποδείξει ότι οι τυχαιοποιημένοι αλγόριθμοι είναι πιο αποδοτικοί από τον καλύτερο δυνατό ντετερμινιστικό αλγόριθμο.

1.2.1 Μοντέλο Υπολογισμού

Πρώτα πρέπει να ορίσουμε το μοντέλο υπολογισμού που θα χρησιμοποιήσουμε. Το μοντέλο μας επεκτείνει ένα ντετερμινιστικό μοντέλο υπολογισμού όπως μια μηχανή Turing με μια επιπρόσθετη είσοδο που αποτελείται απο μια ακολουθία απόλυτα τυχαίων bit (δίκαιες ρίξεις νομισμάτων). Αυτό σημαίνει ότι για κάθε είσοδο, η έξοδος είναι μια τυχαία μεταβλητή των bit της εισόδου, και εμείς μπορούμε να μιλήσουμε για την πιθανότητα γεγονότων όπως $Pr[$«ο αλγόριθμος επιστρέφει σωστό αποτέλεσμα»$]$ ή
$Pr[$«ο αλγόριθμος αποτυγχάνει»$]$.

Χωρίζουμε γενικά τους τυχαιοποιημένους αλγορίθμους σε δύο κατηγορίες: Αλγόριθμοι Monte Calro και Las Vegas.

Οι αλγόριθμοι Monte Carlo τερματίζουν πάντα σε πεπερασμένο χρόνο, αλλά μπορεί να δώσουν λάθος απάντηση. Αυτοί οι αλγόριθμοι διακρίνονται περαιτέρω σε εκείνους με μονόπλευρο και αμφίπλευρο σφάλμα. Οι αλγόριθμοι μονόπλευρου σφάλματος κάνουν σφάλματα μόνο προς μία κατεύθυνση. Για παράδειγμα, εάν η πραγματική απάντηση είναι «ναι», τότε $Pr[$«ναι»$] \geq \epsilon > 0$, ενώ αν η πραγματική απάντηση είναι «όχι», τότε $Pr[$«οχι»$] = 1$. Με άλλα λόγια, η απάντηση «ναι» είναι εγγυημένα ακριβής, ενώ η απάντηση «όχι» είναι αβέβαιη. Μπορούμε να αυξήσουμε την εμπιστοσύνη μας σε μια απάντηση «όχι» εκτελώντας τον αλγόριθμο πολλές φορές και στη συνέχεια απαντάμε «ναι» αν δούμε οποιαδήποτε απάντηση «ναι» μετά από t επαναλήψεις αλλίως απαντάμε «όχι».

Η πιθανότητα σφάλματος σε αυτή την περίπτωση είναι το πολύ $(1-\epsilon)^t$, οπότε αν θέλουμε να την κάνουμε μικρότερη από οποιαδήποτε επιθυμητή τιμή $\delta > 0$, αρκεί να πάρουμε τον αριθμό των δοκιμών t να είναι $O\left(\log \frac{1}{\delta}\right)$ (όπου η σταθερά δ εξαρτάται από το ϵ)

Στους αλγορίθμους Las Vegas, η έξοδος είναι πάντα σωστή, αλλά ο χρόνος εκτέλεσης μπορεί να είναι άπειρος. Ωστόσο, ο αναμενόμενος χρόνος εκτέλεσης απαιτείται να είναι πεπερασμένος. Μπορούμε πάντα να μετατρέψουμε έναν αλγόριθμο Las Vegas σε αλγόριθμο Monte Calro, εκτελώντας τον για ένα σταθερό χρονικό διάστημα και επιστρέφοντας μια αυθαίρετη απάντηση αν αποτύχει να τερματίσει. Αυτό το τέχνασμα λειτουργεί επειδή μπορούμε να φράξουμε την πιθανότητα να τρέξει ο αλγόριθμος πέρα από το ένα σταθερό όριο (χρησιμοποιώντας την ανισότητα του Markov και το γεγονός ότι η αναμενόμενη τιμή του χρόνου εκτέλεσης είναι πεπερασμένη.

1.3 Συνδεσιμότητα Κορυφών

Το πρόβλημα της συνδεσιμότητας κορυφών, που συχνά αναφέρεται και ως πρόβλημα αποκοπής κορυφών (vertex cut) είναι ένα κλασικό πρόβλημα στη θεωρία γραφημάτων που έχει μελετηθεί εκτενώς τις τελευταίες δεκαετίες. Σε ένα γράφημα $G = (V, E)$, το πρόβλημα αποκοπής κορυφών περιλαμβάνει την εύρεση του ελάχιστου συνόλου κορυφών του οποίου η αφαίρεση αποσυνδέει το γράφημα σε περισσότερες απο μια συνεκτικές συνιστώσες. Αυτό το σύνολο κορυφών ονομάζεται σύνολο αποκοπής ή διαχωριστής. Το πρόβλημα είναι θεμελιώδους σημασίας για την αξιοπιστία των δικτύων, καθώς η συνδεσιμότητα των κορυφών ενός γράφου σχετίζεται άμεσα με την ανθεκτικότητά του σε αποτυχίες.

1.3.1 Επισκόπηση

Η συνδεσιμότητα κορυφών αποτελεί ένα θεμελιώδες θέμα στη θεωρία γραφημάτων. Η συνδεσιμότητα κορυφών κ_G ενός γράφου G ορίζεται ως ο ελάχιστος αριθμός κορυφών που πρέπει να αφαιρεθούν ώστε να αποσυνδεθεί ένα ζευγάρι από τις εναπομείνασες κορυφές. Στην περίπτωση των κατευθυνόμενων γραφημάτων, αυτό σημαίνει ότι δεν υπάρχει κατευθυνόμενη διαδρομή μεταξύ δύο κόμβων u και v στο υπόλοιπο υπογράφημα.

Από το 1969, σημαντική έρευνα έχει επικεντρωθεί στην ανάπτυξη αποδοτικών αλγορίθμων [3]–[15] για τον προσδιορισμό της k-συνδεσιμότητας (δηλαδή, για να καθοριστεί εάν $\kappa_G \geq k$) ή για τον ακριβή υπολογισμό της κ_G. Για τα μη κατευθυνόμενα γραφήματα, οι Aho, Hopcroft και Ullman διατύπωσαν το 1974 την εικασία ότι υπάρχει αλγόριθμος χρόνου $O(m)$ για τον υπολογισμό της κ_G για έναν γράφο με n κορυφές και m ακμές. Ωστόσο, παρά την εικασία αυτή, δεν υπάρχει αλγόριθμος ταχύτερος από τον $O(n^2)$, ακόμη και για την περίπτωση που $k = 4$.

1.3.2 Το Θεώρημα του Menger

Το θεώρημα του Menger παρέχει μια θεμελιώδη σχέση μεταξύ της συνδεσιμότητας κορυφών και του αριθμού ανεξάρτητων διαδρομών μεταξύ δύο κορυφών. Συγκεκριμένα, αναφέρει ότι ο ελάχιστος αριθμός κορυφών που πρέπει να αφαιρεθούν για να αποσυνδεθούν δύο κορυφές u και v είναι ίσος με τον μέγιστο αριθμό από ανεξάρτητες διαδρομές κορυφών μεταξύ τους.

Theorem 1.3.1. *(Θεώρημα του Menger): Έστω $G = (V, E)$ ένας γράφος και $u, v \in V$. Ο ελάχιστος αριθμός κορυφών που πρέπει να αφαιρεθούν για να αποσυνδεθούν οι κορυφές u και v είναι ίσος με τον μέγιστο αριθμό από ανεξάρτητες διαδρομές κορυφών μεταξύ τους.*

Αυτό το θεώρημα είναι θεμελιώδες και αποτελεί τη βάση για πολλούς αλγόριθμους κοπής κορυφών.

1.3.3 Αλγόριθμος Ford-Fulkerson

Ο αλγόριθμος Ford-Fulkerson, αρχικά αναπτυγμένος για την επίλυση του προβλήματος της μέγιστης ροής σε δίκτυα, έχει σημαντικές εφαρμογές και στις κοπές κορυφών. Μέσω του θεωρήματος της μέγιστης ροής και ελάχιστης κοπής, το πρόβλημα της κοπής κορυφών μπορεί να μειωθεί σε πρόβλημα ροής.

Theorem 1.3.2. *(Θεώρημα Μέγιστης Ροής - Ελάχιστης Κοπής): Σε ένα δίκτυο ροής, η μέγιστη τιμή της ροής είναι ίση με την χωρητικότητα της ελάχιστης κοπής που διαχωρίζει την πηγή από την καταβόθρα.*

Ο αλγόριθμος Ford-Fulkerson μπορεί να προσαρμοστεί για το πρόβλημα κοπής κορυφών χρησιμοποιώντας έναν μετασχηματισμό του γραφήματος στον οποίο κάθε κορυφή να χωρίζεται σε δύο κορυφές συνδεδεμένες με μια ακμή. Αυτός ο μετασχηματισμός επιτρέπει να ανάγουμε το πρόβλημα κοπής κορυφών σε πρόβλημα κοπής ακμών στο τροποποιημένο δίκτυο και να εφαρμόσουμε τον αλγόριθμο Ford-FUlkerson.

1.3.4 Τυχαιοποιημένος Αλγόριθμος του Karger

Το 1993, ο David Karger εισήγαγε έναν τυχαίο αλγόριθμο για το πρόβλημα της ελάχιστης κοπής, ο οποίος μπορεί να προσαρμοστεί και για τις κοπές κορυφών. Ο αλγόριθμος του Karger επιλέγει επανειλημμένα τυχαίες ακμές και τις συρρικνώνει μέχρι να παραμείνουν δύο μόνο κορυφές. Οι εναπομείνασες ακμές μεταξύ των δύο κορυφών αντιπροσωπεύουν μια κοπή.

Ο αλγόριθμος έχει χρόνο εκτέλεσης $O(n^2)$ και παρέχει μια λύση για την ελάχιστη κοπή με πιθανότητα επιτυχίας που μπορεί να αυξηθεί τρέχοντας τον αλγόριθμο πολλές φορές.

1.3.5 Κοπή Κορυφών με Τοπικές Μεθόδους Υπολογισμού

Στην πρόσφατη εργασία των Nanongkai, Saranurak και Yingchareonthawornchai [1], παρουσιάζονται νέοι τυχαιοποιημένοι αλγόριθμοι για το πρόβλημα της μικρής συνδεσιμότητας κορυφών. Οι αλγόριθμοι αυτοί ξεπερνούν το παλιό φράγμα πολυπλοκότητας $O(n^2)$, που ίσχυε για πάνω από 49 χρόνια, για την περίπτωση που η συνδεσιμότητα k είναι μικρή ($k = O(1)$).

Ο αλγόριθμος που προτείνουν είναι ένας τυχαιοποιημένος αλγόριθμος τύπου Monte Carlo με χρόνο εκτέλεσης $\tilde{O}(mk^2)$, ο οποίος για μικρές τιμές του $k = O(\sqrt{n})$ μπορεί να αποφασίσει με υψηλή πιθανότητα ($w.h.p$) εάν $\kappa_G \geq k$. Αν $\kappa_G < k$, τότε ο αλγόριθμος επιστρέφει ένα σύνολο κοπής κορυφών μεγέθους μικρότερου από το k.

Μια βασική συμβολή της εργασίας αυτής είναι η ανάπτυξη ενός τοπικού αλγορίθμου για τον υπολογισμό της συνδεσιμότητας κορυφών. Σε αντίθεση με τις προηγούμενες προσεγγίσεις που βασίζονταν στον υπολογισμό της συνδεσιμότητας από μία μόνο πηγή, οι οποίες είχαν κατώτερο όριο χρόνου εκτέλεσης τετραγωνικής τάξης, ο τοπικός αλγόριθμος αυτός αποφεύγει την ανάγκη να διαβαστεί ολόκληρο το γράφημα. Αντίθετα, επικεντρώνεται στην εύρεση ενός διαχωριστή μεγέθους το πολύ k, ή στην πιστοποίηση ότι δεν υπάρχει τέτοιος διαχωριστής "κοντά" σε έναν δεδομένο κόμβο εκκίνησης.

Το κύριο αποτέλεσμα της εργασίας είναι το εξής:

Theorem 1.3.3. *Δεδομένου ενός κατευθυνόμενου γράφου $G = (V, E)$ και $k = O(\sqrt{n})$, εάν ο G δεν είναι k-συνεκτικός τότε με υψηλή πιθανότητα ($w.h.p$) ο Αλγόριθμος 6 βρίσκει έναν διαχωριστή S με μέγεθος μικρότερο από k. Εάν ο G είναι k-συνεκτικός, τότε ο Αλγόριθμος 6 επιστρέφει πάντα $\perp$. Ο αλγόριθμος απαιτεί χρόνο $\tilde{O}(mk^2)$.*

1.4 Παράλληλη Προσπελασιμότητα

1.4.1 Επισκόπηση

Το πρόβλημα της προσπελασιμότητας μεμονωμένης πηγής (single-source reachability) σε ένα κατευθυνόμενο γράφημα $G = (V, E)$ με n κορυφές και m ακμές, αφορά τον υπολογισμό του συνόλου των κορυφών $T \subseteq V$ που είναι προσβάσιμες από μια δεδομένη κορυφή $s \in V$. Το πρόβλημα αυτό μπορεί να λυθεί σε γραμμικό χρόνο $O(n + m)$ με αλγορίθμους όπως η αναζήτηση κατά πλάτος (BFS) ή κατά βάθος (DFS). Παρά την ευκολία του προβλήματος στα σειριακά μοντέλα υπολογισμού, η βελτιστοποίησή του σε παράλληλα μοντέλα παραμένει απαιτητική.

Η εργασία των Sidford et al. [2] εισάγει νέες τεχνικές βελτιστοποίησης της παράλληλης και κατανεμημένης προσπελασιμότητας μέσω των λεγόμενων *hopsets* και *shortcuts*, μειώ-

νοντας σημαντικά τη διάμετρο του γραφήματος, επιτρέποντας ταχύτερους υπολογισμούς. Η μείωση της διαμέτρου ενός γραφήματος είναι κρίσιμη για την παράλληλη προσπελασιμότητα, καθώς η διάμετρος αντιπροσωπεύει το μέγιστο μήκος μονοπατιού στο γράφημα.

1.4.2 Δημιουργία Hopsets για Μείωση της Διαμέτρου

Η κλασική μέθοδος μείωσης της διαμέτρου βασίζεται στη δημιουργία hopsets μέσω μιας τυχαιοποιημένης κατασκευής. Ένα hopset H είναι ένα σύνολο ακμών που, όταν προστεθούν στο γράφημα, μειώνουν τη διάμετρό του διατηρώντας τις προσβασιμότητες. Η διαδικασία κατασκευής hopsets περιλαμβάνει την τυχαία επιλογή κορυφών και την προσθήκη ακμών μεταξύ αυτών, με αποτέλεσμα τη μείωση της διαμέτρου σε $O(\sqrt{n})$ με προσθήκη $\tilde{O}(n)$ ακμών.

Theorem 1.4.1. *Δεδομένου ενός συνδεδεμένου γραφήματος $G = (V, E)$ με n κορυφές και m ακμές, μπορούμε να προσθέσουμε $\tilde{O}(n)$ ακμές ώστε η διάμετρος του νέου γραφήματος G' να είναι $O(\sqrt{n})$.*

1.4.3 Παράλληλη Προσπελασιμότητα μέσω Hopsets

Η εργασία από [2] προτείνει έναν παράλληλο αλγόριθμο για το πρόβλημα της προσπελασιμότητας σε μοντέλο PRAM, με αποτέλεσμα τον υπολογισμό προσπελασιμότητας με $\tilde{O}(m)$ εργασία και βάθος $O(n^{1/2})$. Ο αλγόριθμος αυτός βασίζεται στην προσθήκη $\tilde{O}(n)$ shortcuts, τα οποία μειώνουν τη διάμετρο του γραφήματος και επιτρέπουν τον παράλληλο υπολογισμό προσπελασιμότητας.

Theorem 1.4.2 (Προσπελασιμότητα σε PRAM)**.** *Υπάρχει ένας παράλληλος αλγόριθμος ο οποίος, δεδομένου οποιουδήποτε κατευθυνόμενου γράφου με n κορυφές και m ακμές, εκτελεί εργασία $\widetilde{O}(m)$ με βάθος $n^{1/2+o(1)}$ και υπολογίζει ένα σύνολο από $\widetilde{O}(n)$ shortcuts, τέτοιο ώστε η προσθήκη αυτών των ακμών στο γράφημα να μειώνει τη διάμετρο σε $n^{1/2+o(1)}$ με υψηλή πιθανότητα.*

1.4.4 Εισαγωγή

Για να κατανοήσουμε καλύτερα τη μέθοδο, ορίζουμε έννοιες όπως οι *πρόγονοι* και οι *απόγονοι* μιας κορυφής σε ένα κατευθυνόμενο γράφημα G. Για κάθε κορυφή v, τα σύνολα $R_G^{\text{Anc}}(v)$ και $R_G^{\text{Des}}(v)$ ορίζονται ως οι κορυφές που μπορούν να φτάσουν στην v και οι κορυφές που μπορεί να φτάσει η v, αντίστοιχα. Τα σύνολα αυτά είναι χρήσιμα για την κατασκευή των hopsets και την εφαρμογή των shortcuts.

1.4.5 Αλγόριθμος Παράλληλης Μείωσης της Διαμέτρου

Η μέθοδος παράλληλης μείωσης της διαμέτρου ξεκινά επιλέγοντας τυχαίες κορυφές S από το γράφημα και προσθέτοντας ακμές ώστε να συνδέονται οι απόγονοι και οι πρόγονοι των κορυφών αυτών. Στη συνέχεια, η διαδικασία επαναλαμβάνεται σε μικρότερες ομάδες κορυφών που ορίζονται από την προσπελασιμότητα στις επιλεγμένες κορυφές, μειώνοντας σταδιακά τη διάμετρο του γραφήματος.

Ο αλγόριθμος λειτουργεί ως εξής:

1. Επιλέγουμε τυχαία ένα υποσύνολο κορυφών S και για κάθε κορυφή $v \in S$, προσθέτουμε ακμές που συνδέουν την v με τους απογόνους και τους προγόνους της.
2. Ορίζουμε τις κορυφές που είναι σχετικές με την v, δηλαδή αυτές που μπορούν να φτάσουν στην v ή στις οποίες μπορεί να φτάσει η v, και προσθέτουμε shortcuts.
3. Χωρίζουμε τις κορυφές του γραφήματος σε υποσύνολα με βάση τη σχέση τους με τις κορυφές του S και εφαρμόζουμε τον ίδιο αλγόριθμο αναδρομικά στα υποσύνολα αυτά.

1.4.6 Βελτιώσεις μέσω Περιορισμένων Αναζητήσεων

Για να παραλληλοποιηθεί αποτελεσματικά η μέθοδος, αντί για πλήρεις αναζητήσεις (π.χ., BFS ή DFS) σε όλο το γράφημα, χρησιμοποιούνται *περιορισμένες σε βάθος αναζητήσεις* (depth-restricted searches). Αυτές οι αναζητήσεις περιορίζονται σε απόσταση D.

Η κεντρική ιδέα είναι ότι προσδιορίζουμε σύνολα κορυφών που απέχουν το πολύ D από κάθε κορυφή v και προσθέτουμε shortcuts για να μειώσουμε την απόσταση μεταξύ αυτών των κορυφών. Η διαδικασία αυτή επαναλαμβάνεται αρκετές φορές, με το D να μειώνεται συνεχώς, έως ότου οι αποστάσεις μεταξύ των κορυφών να μειωθούν σημαντικά.

1.4.7 Ανάλυση Πολυπλοκότητας

Η πολυπλοκότητα του αλγορίθμου παραμένει σχεδόν γραμμική, δηλαδή $\tilde{O}(m)$ εργασία, ενώ το βάθος του υπολογισμού μειώνεται σε $O(n^{1/2+o(1)})$. Ο αλγόριθμος προσθέτει $\tilde{O}(n)$ shortcuts, τα οποία μειώνουν τη διάμετρο του γραφήματος και επιτρέπουν ταχύτερους παράλληλους υπολογισμούς προσπελασιμότητας.

1.5 Παράλληλη Συνδεσιμότητα κορυφών

1.5.1 Επισκόπηση

Σε αυτό το κεφάλαιο, παρουσιάζουμε δυο αλγορίθμους για το πρόβλημα της παράλληλης τομής κορυφών συνδυάζοντας ιδέες από δύο προσεγγίσεις που αναλύθηκαν στα προηγούμενα κεφάλαια: τον αλγόριθμο τομής κορυφών των Nanongkai et al. [1] και τον παράλληλο αλγόριθμο προσπελασιμότητας των Sidford et al. [2]. Ο ένας αλγόριθμος εξετάζει τη γενική περίπτωση που $k = O(\sqrt{n}$ ενω ο δεύτερος αφορά την περίπτωση που το k είναι σταθερό, δηλαδή $O(1)$.

Τα αποτελέσματα μας συνοψίζονται στα ακόλουθα θεωρήματα:

Theorem 1.5.1 (Παράλληλη Τομή Κορυφών)**.** *Υπάρχει ένας παράλληλος τυχαίος (Monte Carlo) αλγόριθμος που, δεδομένου ενός κατευθυνόμενου γραφήματος $G = (V, E)$, $k = O(\sqrt{n})$, και $\alpha \in [\sqrt{n}, n]$, με μεγάλη πιθανότητα (w.h.p), βρίσκει έναν διαχωριστή S με μέγεθος μικρότερο από k αν G δεν είναι k-συνδεδεμένος. Αν G είναι k-συνδεδεμένος, τότε ο αλγόριθμος πάντα επιστρέφει $\perp$. Ο αλγόριθμος έχει πολυπλοκότητα έργου $\tilde{O}(mk^2 + \frac{m^2}{\alpha})$ και βάθος $\max\{n^{1/2+o(1)}, k\alpha\}$.*

Theorem 1.5.2 (Παράλληλη Τομή Κορυφών με σταθερό k)**.** *Υπάρχει ένας παράλληλος τυχαίος (Monte Carlo) αλγόριθμος που, δεδομένου ενός κατευθυνόμενου γραφήματος $G = (V, E)$, $k = O(1)$, και $\alpha \in [\sqrt{n}, n]$, με μεγάλη πιθανότητα (w.h.p), βρίσκει έναν διαχωριστή S με μέγεθος μικρότερο από k αν G δεν είναι k-συνδεδεμένος. Αν G είναι k-συνδεδεμένος, τότε ο αλγόριθμος πάντα επιστρέφει $\perp$. Ο αλγόριθμος έχει πολυπλοκότητα έργου $\tilde{O}(\frac{mn}{\alpha})$ και βάθος $O(\alpha)$.*

1.5.2 Ο Αλγόριθμος

Υποθέτουμε ότι ο γράφος G δεν είναι k-συνεκτικός, δηλαδή υπάρχει ένας διαχωρισμός του γράφου σε τρία σύνολα (L, S, R) με $|S| < k$. Ο αλγόριθμος εξετάζει δύο περιπτώσεις, ανάλογα με το μέγεθος των περιοχών L και R.

- **Ισορροπημένη Περίπτωση**: Όταν οι περιοχές L και R έχουν παρόμοιο μέγεθος, δειγματοληπτούμε αρκετές ακμές ώστε *με υψηλή πιθανότητα* να συμπεριλαμβάνονται κορυφές από τις δύο περιοχές. Εκτελούμε μια τροποποιημένη έκδοση του αλγορίθμου Ford-Fulkerson για να ανιχνεύσουμε τη μικρή τομή κορυφών S.
- **Μη Ισορροπημένη Περίπτωση**: Όταν το L είναι αρκετά μικρότερο, χρησιμοποιούμε πάλι δειγματοληψία για να βρούμε κορυφή στη μικρή περιοχή και έπειτα εκτελούμε τοπική εξερεύνηση για τον εντοπισμό της μικρής τομής.

1.5.3 Τροποποιημένος Αλγόριθμος Ford-Fulkerson

Χρησιμοποιούμε την παράλληλη προσέγγιση του αλγορίθμου προσπελασιμότητας των Sidford et al. για να μειώσουμε το βάθος της εκτέλεσης του Ford-Fulkerson. Αυτό μας επιτρέπει να υπολογίζουμε το μονοπάτι επαύξησης με έργο $\tilde{O}(m)$ και βάθος $n^{1/2+o(1)}$.

1.5.4 Τροποποιημένος Τοπικός Αλγόριθμος Τομής Κορυφών

Για την περίπτωση όπου το L είναι αρκετά μικρότερο, κάνουμε κάποιες προσαρμογές. Χρησιμοποιούμε τη μείωση στη συνδεσιμότητα ακμών και εκτελούμε DFS ή τον παράλληλο αλγόριθμο προσπελασιμότητας ανάλογα με το μέγεθος της περιοχής L.

1.5.5 Ανάλυση

Η ανάλυση του αλγορίθμου μας χωρίζεται στις δύο περιπτώσεις:

- **Ισορροπημένη Περίπτωση**: Ο αλγόριθμος εκτελείται με έργο $\tilde{O}(mk^2)$ και βάθος $n^{1/2+o(1)}$.
- **Μη Ισορροπημένη Περίπτωση**: Για την πρώτη υποπερίπτωση (όταν $\text{vol}^*(L)$ είναι μικρότερο από την παράμετρο α), χρησιμοποιούμε DFS. Στη δεύτερη περίπτωση, εφαρμόζουμε την παράλληλη προσπελασιμότητα, επιτυγχάνοντας έργο $O\left(\frac{m^2}{\alpha}\right)$ και βάθος $n^{1/2+o(1)}$.

Συνολικά, το έργο και το βάθος του αλγορίθμου καθορίζονται από τις παραμέτρους k και α, εξασφαλίζοντας έναν αποτελεσματικό αλγόριθμο για την παράλληλη τομή κορυφών.

1.6 Παραλλαγή Δειγματοληψίας Κορυφών

Σε αυτήν την ενότητα, εξετάζουμε μια παραλλαγή του παράλληλου αλγορίθμου χρησιμοποιώντας δειγματοληψία κορυφών.

1.6.1 Ανάλυση

Η ανάλυση του αλγορίθμου μας χωρίζεται πάλι σε δυο περιπτώσεις:

- **Ισορροπημένη Περίπτωση**: Ο αλγόριθμος χρησιμοποιεί τον αλγόριθμο παράλληλης προσπελασιμότητας για να βρει διαχωριστές με έργο $W = \tilde{O}(mk^2)$ και βάθος $D = n^{1/2+o(1)}$.
- **Μη Ισορροπημένη Περίπτωση**: Όταν οι δύο συνιστώσες είναι μη ισορροπημένες, διαχωρίζουμε τις περιπτώσεις ανάλογα με τον όγκο της συνιστώσας L και το tradeoff parameter α. Στην πρώτη περίπτωση, χρησιμοποιούμε δειγματοληψία κορυφών, ενώ στη δεύτερη περίπτωση χρησιμοποιούμε τον τροποποιημένο αλγόριθμο τοπικής κοπής κορυφών που χρησιμοποιεί σαν υπορουτίνα τον αλγόριτθμο παράλληλης προσπελασιμότητας.

Ο συνολικός υπολογισμός του έργου και του βάθους για τα βήματα του αλγορίθμου μας είναι:

$$W = \tilde{O}\left(mk^2 + n\alpha k^3 + \frac{mn}{\alpha}\right)$$

$$D = \max\{n^{1/2+o(1)}, \alpha k\}$$

όπου $k = O(\sqrt{n})$ και $\alpha \in [1, n]$.

1.7 Όταν το k είναι σταθερό

Όταν $k = O(1)$, η ανάλυση έργου και βάθους για τις δύο παραλλαγές δειγματοληψίας έχει ως εξής:

1.7.1 LocalVC με Δειγματοληψία Ακμών

Για την περίπτωση δειγματοληψίας ακμών, το έργο είναι $W_{edge} = \tilde{O}\left(\frac{m^2}{\alpha}\right)$ και το βάθος είναι $D_{edge} = \alpha$ όταν επιλέγουμε α στο διάστημα $[\sqrt{n}, n]$. Αυτή η επιλογή μειώνει το έργο χωρίς να αυξάνει το βάθος.

1.7.2 LocalVC με Δειγματοληψία Κορυφών

Για τη δειγματοληψία κορυφών, το έργο είναι $W_{vertex} = \tilde{O}\left(\frac{mn}{\alpha}\right)$ και το βάθος είναι $D_{vertex} = \alpha$ όταν επιλέγουμε $\alpha \in [\sqrt{n}, n]$. Και εδώ, αυτή η επιλογή μειώνει το έργο χωρίς να αυξάνει το βάθος.

Συμπεραίνουμε ότι η μέθοδος δειγματοληψίας κορυφών δίνει καλύτερα αποτελέσματα σε αυτή την περίπτωση.

Chapter 2

Introduction

The aim of this chapter is to motivate the next by providing an overview of the areas that have inspired the topic of this thesis. Specifically, for the most part, our work focuses on Graph Theory, the Vertex Cut problem and Parallel Computing within the PRAM Model. We provide a short introoduction for each topic and refer to relevant previous work.

2.1 Motivation

Graph theory is a fundamental area of discrete mathematics that deals with the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph is composed of vertices (or nodes) connected by edges (or arcs), which may represent various types of relationships or interactions in real-world scenarios, such as social networks, computer networks, or molecular structures. In computational contexts, graph theory plays a crucial role in optimizing problems like shortest path determination, network flow, and various partitioning techniques.

One of the fundamental concerns in graph theory is the analysis of connectivity—how vertices are linked together and how their relationships impact the overall structure. Vertex cuts, also known as node separators, are critical in understanding the vulnerability and robustness of networks. A vertex cut is a subset of vertices whose removal causes the graph to become disconnected, dividing it into two or more independent subgraphs. Identifying small vertex cuts is particularly significant in optimizing network design, improving fault tolerance, and analyzing communication networks, where it is crucial to minimize the impact of node failures.

In computational graph theory, efficiently solving problems related to vertex cuts is challenging, particularly when dealing with large graphs. The Parallel Random Access

Machine (PRAM) model, which enables multiple processors to work simultaneously, provides a framework for tackling such problems in parallel. The PRAM model allows for the development of parallel algorithms that can significantly reduce computational time by distributing tasks across processors. This thesis focuses on the Small Vertex Cut problem within the PRAM model, aiming to explore algorithms that identify minimal sets of vertices whose removal disconnects specific parts of a graph. The research examines how parallel computing can be leveraged to solve this problem more efficiently compared to traditional sequential methods.

2.2 Problem Statement

This thesis focuses on the small vertex cut problem within the PRAM model. The primary goal is to develop efficient parallel algorithms that can compute small vertex cuts in large graphs, with a particular emphasis on minimizing the work and depth of these algorithms. The small vertex cut problem is formally defined as finding a minimal set of vertices whose removal results in the disconnection of the graph into multiple components.

2.3 Contributions

In this thesis, we begin by presenting the necessary preliminaries that form the foundation of our work. This includes a detailed examination of the PRAM model, which is crucial for understanding parallel algorithms and their efficiency in terms of work and depth. We also discuss randomized algorithms, which play a key role in our approach. Additionally, we cover existing algorithms for the vertex cut problem, providing the context for the specific challenge addressed in this work. The reachability problem is another important area we examine, focusing on parallel algorithms designed to solve it and how these approaches relate to the vertex cut problem.

The primary contribution of this thesis is the development of a novel parallel algorithm for computing small vertex cuts in the PRAM model. This algorithm is designed with an emphasis on optimizing both work and depth, which are critical measures of efficiency in parallel computing. We also present a variant of this algorithm for when the size of the separator is constant.

2.4 Organization of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 3** explores the PRAM model, discussing its relevance and importance in parallel algorithm design.
- **Chapter 4** reviews randomized algorithms, explaining their role and how they are applied in our approach.
- **Chapter 5** focuses on vertex connectivity, covering existing algorithms and their relationship to the vertex cut problem, with special focus given to the local vertex cut algorithm by Nanongkai et al. [1].
- **Chapter 6** addresses parallel reachability, reviewing algorithms in this domain and their connection to our work on vertex cuts, with special focus given to the parallel reachability algorithm of Sidford et al. [2].
- **Chapter 7** presents the core contribution of this thesis, the novel parallel algorithm for computing small vertex cuts in the PRAM model. This chapter includes the algorithm's design, its theoretical analysis, and a comprehensive evaluation of its work and depth complexity. It also includes the variant for constant separator size.
- **Chapter 8** includes a conclusion of the thesis and presents directions for future work in this area.

Chapter 3

The PRAM Model

3.1 Introduction

The Parallel Random Access Machine (PRAM) model is one of the most extensively studied frameworks for parallel computing. It serves as the parallel-computing counterpart to the traditional random-access machine (RAM), designed to simulate the behavior of multiple processors that can access a shared memory space simultaneously. In the PRAM model, the processors operate synchronously, meaning they perform operations in lockstep, and communication between them is achieved through shared memory. This makes PRAM an ideal model for analyzing parallelization strategies for various computational tasks, especially for graph-based algorithms where efficient parallelism can significantly reduce time complexity.

The PRAM model is categorized into four sub-models based on how memory access conflicts are handled:

- **Exclusive read exclusive write (EREW):** In this model, only one processor is allowed to read from or write to a memory location at a time, making it the most restrictive PRAM model. The EREW model is challenging to implement but ensures no memory access conflicts.
- **Concurrent read exclusive write (CREW):** Multiple processors are allowed to read from the same memory location concurrently, but only one processor can write to it at any given time. This model allows parallel reading, which is more flexible than EREW but still restricts write operations.
- **Exclusive read concurrent write (ERCW):** Although this model allows multiple processors to write to the same memory location, it is rarely considered as it does not add significant computational power or efficiency in most

applications, since it doesn't allow concurrent read.

- **Concurrent read concurrent write (CRCW):** This is the most general model, where multiple processors can read from and write to the same memory location simultaneously. Different variations of the CRCW model define how write conflicts are handled (e.g., the result of a write conflict could be determined by the minimum or maximum value being written, or it could allow random overwrites). A CRCW PRAM is sometimes referred to as a Concurrent Random Access Machine due to its ability to handle simultaneous operations in both reading and writing.

The model used in this thesis, and in general the most popular PRAM moded is CREW.

3.2 Work/Depth Model of Complexity

In computer science, analyzing parallel algorithms involves determining the computational complexity of algorithms that run in parallel, including the time, memory, or other resources required for execution. While similar to the analysis of sequential algorithms, the process is more complex because it requires consideration of multiple threads working together. A key objective in parallel algorithm analysis is to assess how the algorithm's resource utilization (such as speed and space) is affected by varying the number of processors.

Consider computations that are executed on a machine with p processors. Let T_p represent the time taken from the start to the end of the computation. The analysis of the running time of such computations involves the following key concepts:

- **Work**: The work of a computation, executed by p processors, is the total number of elementary operations performed by the processors. Disregarding communication overhead from processor synchronization, this is equivalent to the time needed for the computation to run on a single processor, denoted by T_1.
- **Depth or Span**: The depth (or span) refers to the longest sequence of operations that must be executed sequentially due to dependencies (often called the critical path). Minimizing the depth is crucial when designing parallel algorithms, as the depth determines the minimum time required for execution. Alternatively, depth is the time T_∞ taken by an idealized machine with an infinite number of processors.
- **Cost**: The total cost of a computation is given by pT_p, representing the total amount of time spent by all processors, including both computation and idle time.

Several important results can be derived from these definitions:

Theorem 3.2.1. *(Work Law): The total cost is always at least the work: $pT_p \geq T_1$. This stems from the fact that p processors can perform at most p operations in parallel.*

Theorem 3.2.2. *(Span Law): A finite number p of processors cannot outperform an infinite number, so $T_p \geq T_\infty$.*

Based on these definitions and laws, the following performance measures can be defined:

Definition 3.2.1 (Speedup S_p)**.** Speedup is the ratio of the time taken for sequential execution to the time taken in parallel execution: $S_p = \frac{T_1}{T_p}$.

If $S_p = \Omega(p)$ for p processors, this is considered linear speedup, which is optimal because, based on the work law, $S_p \leq p$. In some cases, super-linear speedup can occur due to factors like memory hierarchy effects. The case $S_p = p$ is known as perfect linear speedup, and algorithms with this property are considered scalable.

Definition 3.2.2 (Efficiency E_p)**.** Efficiency is defined as the speedup per processor, given by $E_p = \frac{S_p}{p}$.

Definition 3.2.3 (Parallelism)**.** Parallelism is the ratio $\frac{T_1}{T_\infty}$, representing the maximum possible speedup on any number of processors.

Based on the span law, the speedup is limited by parallelism: if $p > \frac{T_1}{T_\infty}$, then:

$$\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} < p.$$

Definition 3.2.4 (Slackness)**.** The slackness is defined as $\frac{T_1}{pT_\infty}$.

If slackness is less than one, perfect linear speedup is impossible, according to the span law.

In the context of PRAM, an efficient parallel algorithm aims to minimize both work and depth to achieve optimal performance when running on a finite number of processors.

3.3 Examples

3.3.1 Minimum Value in a Tree

Consider finding the minimum value in a tree structure. The PRAM model provides an efficient framework for parallelizing this task, allowing multiple processors to explore the tree concurrently and reducing the overall computational time compared to

a sequential approach. Below, we describe how a parallel algorithm can be applied to find the minimum value in a tree, using the PRAM model and its work/depth analysis to evaluate its efficiency.

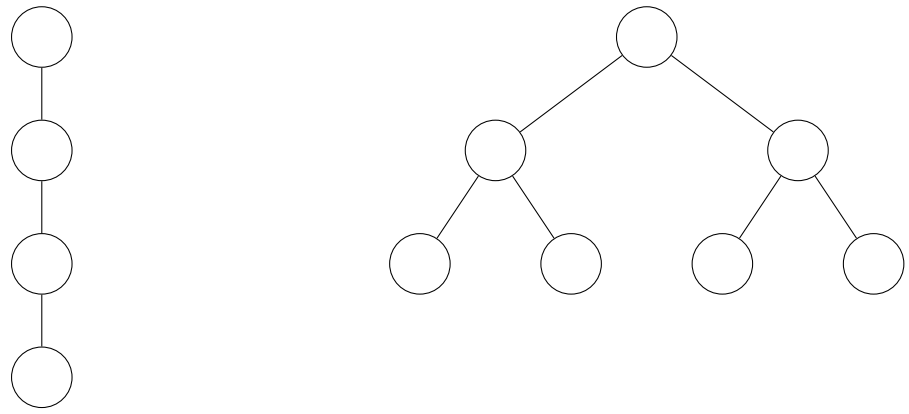
Consider a tree where each node contains a value, and the goal is to find the minimum value across all the nodes. The following algorithm can be applied:

Algorithm 1: Parallel Minimum Value in a Tree

```
1 Input: A binary tree T with root r
2 Output: The minimum value in the tree
3 Function findMin(vode current):
4     if current is a leaf then
5         return current.data;
6     end
7     leftMin ← findMin(current.left);
8     rightMin ← findMin(current.right);
9     return min(current.data, leftMin, rightMin);
```

In this algorithm, the function checks whether the current node is a leaf. If it is, the algorithm simply returns the value of that leaf. Otherwise, the algorithm computes the minimum value by recursively calling `findMin` on both the left and right subtrees. These two recursive calls can be executed in parallel, making this algorithm highly suitable for the PRAM model.

Now consider the following two types of trees:



Degenerate Tree Perfect Binary Tree

Figure 3.1: PRAM example

Case 1: Degenerate Tree

A degenerate tree is essentially a sequence of nodes where each node has only one child, making it similar to a linked list. In terms of computational work, this structure leads to the recurrence:

$$W(n) = W(n-1) + \mathcal{O}(1), \quad W(1) = \mathcal{O}(1).$$

Each step adds constant work, so expanding the recurrence shows that the total work across all n nodes is linear, $W(n) = \mathcal{O}(n)$.

In terms of depth, we assume all calls are in parallel and look for the longest dependency chain. This gives us the recurence:

$$D(n) = D(n-1) + 1, \quad D(1) = 0.$$

This solves to: $D(n) = n$, which means that the depth grows linearly with the number of nodes, making the degenerate tree highly inefficient for parallel processing, as there is no opportunity to divide the work.

Case 2: Perfect Binary Tree

In contrast, a perfect binary tree is highly balanced. Each internal node has exactly two children, and all leaves are at the same level. The recurrence relation for the work is:

$$W(n) = 2W\left(\frac{n}{2}\right) + \mathcal{O}(1), \quad W(1) = \mathcal{O}(1).$$

At each level, the work is split evenly across the two subtrees, and the constant overhead at each level leads to a total work of $W(n) = \mathcal{O}(n)$, which, like the degenerate tree, is linear.

The major advantage of a perfect binary tree comes in terms of depth. To calculate the depth of the computation, we take the max o the recursive calls. The recurrence relation for depth is:

$$D(n) = \max\left(D\left(n/2\right), D\left(n/2\right)\right) + 1, \quad D(1) = 1,$$

which solves to $D(n) = \mathcal{O}(\log n)$. This logarithmic depth is what makes the perfect binary tree much more efficient for parallel processing, as it minimizes the critical path and allows for greater concurrency.

Comparison

The degenerate tree and the perfect binary tree illustrate two extremes in tree structure. Both trees perform the same amount of total work, with $W(n) = \mathcal{O}(n)$. However, the depth differs significantly: the degenerate tree has a depth of $D(n) = n$, meaning that its work cannot be parallelized efficiently, while the perfect binary tree has a depth of $D(n) = \mathcal{O}(\log n)$, which allows the work to be divided across multiple processors with minimal sequential steps. Therefore, while both structures compute the same amount of total work, the perfect binary tree is far more efficient in scenarios that benefit from parallelism.

3.3.2 Parallel Breadth-First Search (BFS)

Problem: Given a graph $G = (V, E)$ and a source vertex s, perform a breadth-first traversal to determine the shortest path distances from s to all other vertices.

The PRAM BFS algorithm works by maintaining the frontier of vertices at each level and updating their neighbors in parallel.

Algorithm 2: Parallel BFS

```
1  Input: A graph G = (V, E), a source vertex s
2  Output: Levels of each vertex from the source vertex s
3  Function ParallelBFS(Graph G, Vertex s):
4      Initialize an array level with all values set to −1, and set level[s] = 0;
5      Initialize a queue Q containing only the source vertex s;
6      while Q is not empty do
7          foreach vertex v ∈ Q in parallel do
8              foreach neighbor u of v do
9                  if level[u] = −1 then
10                     level[u] ← level[v] + 1;
11                     Q ← Q ∪ {u};
12                 end
13             end
14         end
15         Replace Q with the next-level queue;
16     end
```

3.3.2.1 Complexity Analysis

In this section, we analyze the PRAM BFS algorithm using key performance metrics such as work, depth, cost, speedup, efficiency, parallelism, and slackness.

Work: The work of the PRAM BFS algorithm is the total number of elementary operations performed by all processors. In this case, the work is equivalent to the time required to execute the algorithm sequentially. The work complexity for BFS is:

$$T_1 = O(|V| + |E|)$$

where $|V|$ is the number of vertices, and $|E|$ is the number of edges.

Depth or Span: The depth of the PRAM BFS algorithm is the length of the longest chain of dependent operations. In this case, the depth corresponds to the diameter of the graph, which we denote by D. The depth complexity is:

$$T_\infty = O(D)$$

where D is the diameter of the graph.

Cost: The cost of the PRAM BFS algorithm is the total work performed by all processors, taking into account both computation and idle time. If the algorithm runs on p processors and takes T_p time, the cost is:

$$\text{Cost} = pT_p$$

To achieve cost efficiency, we want the total cost to be close to the sequential work T_1, which implies:

$$pT_p \leq O(|V| + |E|)$$

For the PRAM BFS algorithm, assuming that the parallel time is dominated by the depth $T_p = O(D)$, the speedup is:

$$S_p = \frac{O(|V| + |E|)}{O(D)} = O\left(\frac{|V| + |E|}{D}\right).$$

This represents the speedup achieved with p processors. To achieve linear speedup, $S_p = \Omega(p)$, we require $p \leq (|V| + |E|)/D$.

Efficiency is defined as the speedup per processor:

$$E_p = \frac{S_p}{p}.$$

For the PRAM BFS algorithm, this becomes:

$$E_p = \frac{O\left(\frac{|V|+|E|}{D}\right)}{p} = O\left(\frac{|V| + |E|}{pD}\right).$$

This shows that efficiency decreases as the number of processors p increases beyond the parallelism limit $(|V| + |E|)/D$.

Parallelism is the ratio of the total work to the depth:

$$\text{Parallelism} = \frac{T_1}{T_\infty} = \frac{O(|V| + |E|)}{O(D)} = O\left(\frac{|V| + |E|}{D}\right).$$

This represents the maximum possible speedup with any number of processors.

Based on the span law, the speedup is limited by parallelism. If the number of processors p exceeds the parallelism limit $(|V| + |E|)/D$, then:

$$S_p \leq O\left(\frac{|V| + |E|}{D}\right) < p.$$

Slackness is defined as:

$$\text{Slackness} = \frac{T_1}{pT_\infty} = \frac{O(|V| + |E|)}{p \cdot O(D)} = O\left(\frac{|V| + |E|}{pD}\right).$$

For efficient parallelism, slackness should be greater than or equal to 1. If slackness is less than 1, i.e., if $p > (|V| + |E|)/D$, perfect linear speedup is impossible. Therefore, slackness serves as a guideline for choosing an appropriate number of processors for the algorithm.

Conclusion: For graphs with small diameters D, the algorithm scales well. However, as the diameter increases, the depth $O(D)$ becomes the limiting factor, reducing the benefits of parallelism and lowering efficiency.

Chapter 4

Randomized Algorithms

4.1 Introduction

Randomized algorithms are algorithms that make random choices during their execution to achieve good performance on average or with high probability. They are a powerful tool in algorithm design, often leading to simpler, faster, or more efficient solutions compared to their deterministic counterparts. Sometimes these algorithms can be "derandomized," although the resulting deterministic algorithms are often rather obscure. In certain restricted models of computation, one can even prove that randomized algorithms are more efficient than the best possible deterministic algorithm.

4.2 Model of Computation

We first need to formalize the model of computation we will use. Our machine extends a standard model (e.g., a Turing Machine or random-access machine), with an additional input consisting of a stream of perfectly random bits (fair coin flips). This means that for a fixed input, the output is a random variable of the input bit stream, and we can talk about the probability of events such as Pr[algorithm returns correct result] or Pr[algorithm fails].

In the above model, we can view a computation of a randomized algorithm as a path in a binary tree. At each step of the algorithm, we branch right or left with probability $\frac{1}{2}$ based on the random bit we receive. For decision problems, we can label each leaf of this tree with a "yes" or a "no" output. Obviously, we derive no benefit from randomization if all outputs agree.

We generally divide randomized algorithms into two classes: **Monte Carlo** algorithms

and **Las Vegas** algorithms.

4.3 Monte Carlo Algorithms

Monte Carlo algorithms always halt in finite time but may output the wrong answer, i.e., Pr[error] is larger than 0. These algorithms are further divided into those with one-sided and two-sided error.

One-sided error algorithms only make errors in one direction. For example, if the true answer is "yes," then $\Pr[\text{yes}] \geq \epsilon > 0$, while if the true answer is "no," then $\Pr[\text{no}] = 1$. In other words, a "yes" answer is guaranteed to be accurate, while a "no" answer is uncertain. We can increase our confidence in a "no" answer by running the algorithm many times (using independent random bits each time); we then output "yes" if we see any "yes" answer after t repetitions and output "no" otherwise. The probability of error in this scheme is at most $(1-\epsilon)^t$, so if we want to make this smaller than any desired $\delta > 0$, it suffices to take the number of trials t to be $O\left(\log \frac{1}{\delta}\right)$ (where the constant in the O depends on ϵ).

4.3.1 Two-Sided Error Algorithms

Two-sided error algorithms make errors in both directions; thus, e.g., if the true answer is "yes," then $\Pr[\text{yes}] \geq \frac{1}{2} + \epsilon$, and if the true answer is "no," then $\Pr[\text{no}] \geq \frac{1}{2} + \epsilon$. In this case, we can increase our confidence by running the algorithm many times and then taking the majority vote. The following standard calculation shows that the number of trials t required to ensure an error of at most δ is again $O\left(\log \frac{1}{\delta}\right)$.

4.3.2 Proof

The probability that the majority vote algorithm yields an error is equal to the probability that we obtain at most $t/2$ correct outputs in t trials, which is bounded above by

$$\begin{aligned}\sum_{i=0}^{\lfloor t/2\rfloor}\binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{i}\left(\frac{1}{2}-\epsilon\right)^{t-i} &\le \sum_{i=0}^{\lfloor t/2\rfloor}\binom{t}{i}\left(\frac{1}{2}+\epsilon\right)^{t/2}\left(\frac{1}{2}-\epsilon\right)^{t/2}\\ &\le \left(\frac{1}{4}-\epsilon^2\right)^{t/2}\sum_{i=0}^{\lfloor t/2\rfloor}\binom{t}{i}\\ &\le \left(\frac{1}{4}-\epsilon^2\right)^{t/2}2^t\\ &= (1-4\epsilon^2)^{t/2}.\end{aligned}$$

Taking

$$t=\frac{2\log\frac{1}{\delta}}{\log(1-4\epsilon^2)^{-1}}=C\log\frac{1}{\delta}$$

where C is a constant depending on ϵ, makes this at most δ.

4.4 Las Vegas Algorithms

In a Las Vegas algorithm, the output is always correct, but the running time may be unbounded. However, the *expected* running time is required to be bounded. Equivalently, we require the running time to be bounded but allow the algorithm to output either a correct answer or a special symbol $\perp$ so that the probability of outputting $\perp$ is at most $1/2$.

Note that we can always turn a Las Vegas algorithm into a Monte Carlo algorithm by running it for a fixed amount of time and outputting an arbitrary answer if it fails to halt. This works because we can bound the probability that the algorithm will run past a fixed limit (using Markov's inequality and the fact that the expectation is bounded). The reverse is apparently not true because of the strict "correct output" requirement of Las Vegas algorithms. Thus, there is no general scheme for converting a Monte Carlo algorithm into a Las Vegas one.

4.5 Complexity Classes

Randomized algorithms introduce new complexity classes that extend classical deterministic complexity theory by allowing bounded randomness. These classes help characterize the power of randomized algorithms compared to deterministic ones. The

most important complexity classes associated with randomized algorithms are **RP** (Randomized Polynomial time), **ZPP** (Zero-error Probabilistic Polynomial time), and **BPP** (Bounded-error Probabilistic Polynomial time).

4.5.1 RP (Randomized Polynomial time with One-Sided Error)

The class **RP** contains decision problems where a randomized algorithm runs in polynomial time and never makes a false positive error. If the answer is "no," the algorithm is guaranteed to return "no," but if the answer is "yes," the algorithm will return "yes" with probability at least $\frac{1}{2}$. Formally, a language L belongs to RP if there is a probabilistic polynomial-time algorithm A such that:

- If $x \in L$, then $\Pr[A(x) = 1] \geq \frac{1}{2}$.
- If $x \notin L$, then $\Pr[A(x) = 0] = 1$.

It is easy to amplify the success probability of an RP algorithm by running it multiple times and using majority voting, similarly to BPP algorithms.

The relationship P $\subseteq$ RP $\subseteq$ BPP $\subseteq$ NP is clear, though RP is weaker than BPP because it only allows one-sided error.

4.5.2 BPP (Bounded-error Probabilistic Polynomial time)

The class of decision problems solvable by a polynomial-time Monte Carlo algorithm with two-sided error is known as **BPP (Bounded-error Probabilistic Polynomial time)**. Specifically, for a problem in BPP, there exists an algorithm that, for all inputs, returns the correct answer with a probability of at least $2/3$, and the probability of an incorrect answer is at most $1/3$.

Formally, a language L belongs to BPP if there exists a probabilistic polynomial-time algorithm A such that for all x:

- If $x \in L$, then $\Pr[A(x) = 1] \geq \frac{2}{3}$.
- If $x \notin L$, then $\Pr[A(x) = 0] \geq \frac{2}{3}$.

This class is often viewed as a practical class of problems because most real-world applications tolerate a small error, especially when the error probability can be made arbitrarily small.

4.5.3 ZPP (Zero-error Probabilistic Polynomial time)

The class of decision problems solvable by a polynomial-time Las Vegas algorithm is known as **ZPP (Zero-error Probabilistic Polynomial time)**. A problem is in ZPP if it can be solved without any errors, but the algorithm may take a variable amount of time to complete, with the expected time being polynomial.

Formally, a language L belongs to ZPP if there is a probabilistic polynomial-time algorithm A such that for all x:

- The expected runtime of $A(x)$ is polynomial.
- $A(x)$ returns the correct answer (no errors are allowed).

It is known that ZPP = RP $\cap$ coRP, meaning ZPP consists of problems for which both the problem and its complement are in RP.

4.5.4 Containments and Relationships

The class BPP sits between the deterministic class P (problems solvable in polynomial time without randomness) and the class NP (nondeterministic polynomial time), but its exact relationship with these classes is still not fully resolved. It is known that:

- P $\subseteq$ RP $\subseteq$ BPP: Any problem that can be solved deterministically in polynomial time can trivially be solved using randomness by simply not utilizing the random bits. Thus, P $\subseteq$ RP $\subseteq$ BPP is clear. Randomized algorithms in RP can only make errors on "yes" instances, but the use of randomness in BPP algorithms introduces two-sided error, allowing errors on both "yes" and "no" instances. Hence, RP $\subseteq$ BPP.
- RP $\subseteq$ NP: Since RP algorithms accept "yes" instances with probability at least $1/2$, they can be thought of as nondeterministic algorithms. In the case of RP, if a problem has a "yes" solution, the randomized algorithm has a chance of finding it, thus mimicking the guessing process of nondeterminism. Therefore, RP $\subseteq$ NP is obvious.

However, the relationship between BPP and NP is more complex:

- BPP $\subseteq$ NP? It is not known whether BPP is contained within NP, because BPP allows two-sided error, and it's unclear whether there is a nondeterministic algorithm that can efficiently handle both types of errors. While RP algorithms can simulate a nondeterministic "guess," BPP algorithms involve more uncertainty

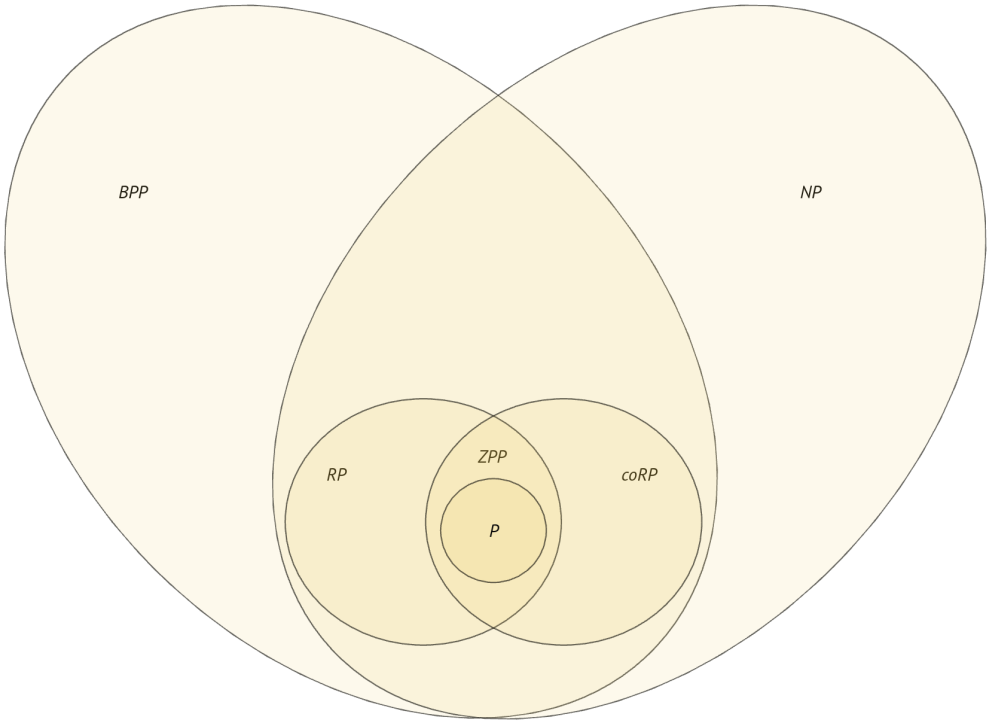


Figure 4.1: Relations of complexity classes in case BPP and NP are discrete

due to their allowance of errors in both directions. This two-sided error complicates the simulation of BPP within NP. Hence, the inclusion of BPP $\subseteq$ NP remains an open question in complexity theory.

- NP $\subseteq$ BPP? It is very unlikely that NP $\subseteq$ BPP. If NP $\subseteq$ BPP, it would imply that every problem in NP, such as the satisfiability problem (SAT), could be solved probabilistically in polynomial time with bounded error. This would further imply that all NP problems have polynomial-size circuits, a result that would cause the polynomial hierarchy to collapse to its second level, contradicting widely-held beliefs in complexity theory. Thus, most researchers believe that NP $\not\subseteq$ BPP.

The general consensus in complexity theory is that BPP is likely distinct from NP, and resolving this question would have profound implications on our understanding of the relationships between probabilistic and nondeterministic computations.

4.5.5 Derandomization and the Question of P = BPP

A central question in complexity theory is whether randomness truly gives more power to polynomial-time algorithms. Specifically, it is an open question whether P = BPP, meaning that every problem solvable by a randomized polynomial-time algorithm with bounded error can also be solved deterministically in polynomial time. If P = BPP, it would imply that randomness does not provide additional power beyond what deterministic algorithms can achieve.

There has been significant progress in the study of derandomization, where researchers attempt to simulate randomized algorithms using deterministic methods. Techniques like pseudorandom generators and hardness vs. randomness results suggest that it might be possible to derandomize BPP algorithms under certain assumptions. For example, if there exist strong enough pseudorandom number generators, then P = BPP. However, proving this equivalence in general remains an open problem.

In the context of parallel computation, particularly the PRAM model, randomness can often lead to more efficient algorithms. Many problems that are difficult to parallelize deterministically can be solved efficiently using randomized techniques. For example, load balancing, random sampling, and graph problems such as finding small vertex cuts can all benefit from randomized approaches.

Chapter 5

Vertex Connectivity

5.1 Introduction

The vertex connectivity problem, often referred to as vertex cut, is a classical problem in graph theory that has been studied extensively over the past several decades. In a graph $G = (V, E)$, the vertex cut problem involves finding the minimum set of vertices whose removal disconnects the graph. This set of vertices is called a *vertex cut* or *separator*. The problem is fundamental in network reliability, as the vertex connectivity of a graph is directly related to its robustness to failures. In this chapter, we provide a historical analysis of the vertex cut problem, focusing on key developments, algorithms, and advancements.

5.2 Overview

Vertex connectivity is a fundamental topic in graph theory. The vertex connectivity κ_G of a graph G is defined as the minimum number of vertices that need to be removed to disconnect some pair of remaining nodes. In the case of directed graphs, this means that there is no directed path between a node u and a node v in the remaining subgraph.

Since 1969, significant research has focused on developing efficient algorithms [3]–[15] to decide k-connectivity (i.e., determining if $\kappa_G \geq k$) or to compute the exact value of κ_G. Table 1 provides further details on these results. For undirected graphs, Aho, Hopcroft, and Ullman conjectured in 1974 that there exists an $O(m)$-time algorithm to compute κ_G for a graph with n nodes and m edges. However, despite this conjecture, no algorithm faster than $O(n^2)$ exists, even for the case where $k = 4$.

For undirected graphs, the earliest known $O(n^2)$ algorithm for the case $m = O(n)$ and

$k = O(1)$ dates back over five decades to the work of Kleitman [3], who introduced an algorithm for deciding k-connectivity with a runtime of $O(kn \cdot \mathrm{VC}_k(n, m))$, where $\mathrm{VC}_k(n, m)$ represents the time required to decide whether the smallest s-t vertex-cut has size at least κ for fixed s and t. While not stated explicitly, it was known that $\mathrm{VC}_k(n, m) = O(mk)$ using the Ford-Fulkerson algorithm [16], leading to a time complexity of $O(k^2nm)$, which simplifies to $O(n^2)$ when $m = O(n)$ and $k = O(1)$, based on the 1992 results of Nagamochi and Ibaraki [11]. Subsequent work by Tarjan [17] and Hopcroft and Tarjan [18] provided $O(m)$-time algorithms for the cases where $k = 2$ and $k = 3$, respectively.

Over the years, various works have improved upon Kleitman's bound for larger values of k and m, but none have managed to surpass the $O(n^2)$ barrier. Kanevsky and Ramachandran [19] were the first to achieve an $O(n^2)$ time bound for $k = 4$, while Nagamochi and Ibaraki [11] achieved the same for any constant k. For general k and m, the fastest known running times are $\tilde{O}(n^\omega + nk^\omega)$ from Linial, Lovász, and Wigderson [9], and $\tilde{O}(kn^2)$ by Henzinger, Rao, and Gabow [14], where $\tilde{O}$ suppresses polylogarithmic factors and ω is the exponent for matrix multiplication, currently known to satisfy $\omega < 2.371552$ [20].

For directed graphs, an $O(m)$-time algorithm exists only for $k \leq 2$, as shown by Georgiadis [21]. For larger k and general m, the best running times are $\tilde{O}(n^\omega + nk^\omega)$ from Cheriyan and Reif [12], and $\tilde{O}(mn)$ from Henzinger et al. [14]. All of these state-of-the-art algorithms for general k and m, both for undirected and directed graphs [9], [12], [14], are randomized and have a high probability of being correct. The fastest deterministic algorithm is due to Gabow [15], though it runs more slowly. Approximation algorithms have also been developed, including a deterministic 2-approximation algorithm by Henzinger [13] with time complexity $O(\min\{\sqrt{n}, k\}n^2)$, and a more recent randomized $O(\log n)$-approximation algorithm by Censor-Hillel, Ghaffari, and Kuhn [10] with time complexity $\tilde{O}(m)$. These approximation algorithms apply only to undirected graphs.

The main challenge faced by previous exact algorithms, aside from a few $O(m)$-time algorithms for $k \leq 3$, is their reliance on solving the following problem: for two nodes s and t, let $\kappa(s, t)$ represent the minimum number of vertices (excluding s and t) that must be removed to ensure there is no path from s to t. Many previous algorithms require solving for $\kappa(s, t) \geq k$ for all t given a fixed node s. This is known as the single-source k-connectivity problem, for which no algorithm achieving better than $O(n^2)$ time exists, even when $k = O(1)$ and $m = O(n)$.

Reference	Directed	Undirected	Note
Folklore	$O(n^2 \cdot \mathrm{VC}(n,m))$		
[3]	$O(kn \cdot \mathrm{VC}_k(n,m))$		
[4], [22]	$O(kn \cdot \mathrm{VC}(n,m))$		
[4] (cf. [5]–[7])	$O((k^2+n) \cdot \mathrm{VC}_k(n,m))$		
[8]	$\tilde{O}(n \cdot \mathrm{VC}(n,m))$		Monte Carlo
[9] ([12] for directed)	$O((n^\omega + nk^\omega)\log n)$		Monte Carlo
	$O((n^\omega + nk^\omega)k)$		Las Vegas
[11], [12]	-	$O(k^3 n^{1.5} + k^2 n^2)$	
[13]	-	$O(\min\{\sqrt{n}, k\}n^2)$	2-approx.
[14]	$O(mn \log n)$	$O(kn^2 \log n)$	Monte Carlo
	$O(\min\{n, k^2\}km + mn)$	$O(\min\{n, k^2\}k^2 n + \kappa n^2)$	
[15]	$O(\min\{n^{3/4}, k^{1.5}\}km + mn)$	$O(\min\{n^{3/4}, k^{1.5}\}k^2 n + kn^2)$	
[10]	-	$\tilde{O}(m)$	Monte Carlo, $O(\log n)$-approx.
[1]	$\tilde{O}(\min\{km^{2/3}n, km^{4/3}\})$	$\tilde{O}(m + k^{7/3}n^{4/3})$	Monte Carlo, for $k \leq \sqrt{n}$
	$\tilde{O}(t_{\text{directed}})$	$\tilde{O}(t_{\text{undirected}})$	Monte Carlo, $(1+\epsilon)$-approx.

Table 5.1: List of running time $T(k,n,m)$ of previous algorithms on a graph G with n nodes and m edges for deciding if the vertex connectivity $\kappa_G \geq k$. $\mathrm{VC}(n,m)$ is the time needed for finding the minimum size s-t vertex cut for fixed s,t. $\mathrm{VC}_k(n,m)$ is the time needed for either certifying that the minimum size s-t vertex cut is of size at least k, or return such cut. This table is taken from [1].

5.3 Important Milestones

5.3.1 Menger's Theorem

Menger's theorem provides a fundamental relationship between vertex connectivity and the number of independent paths between two vertices. Specifically, it states that the minimum number of vertices that must be removed to disconnect two vertices u and v is equal to the maximum number of vertex-disjoint paths between u and v.

Theorem 5.3.1. *(Menger's Theorem): Let $G = (V, E)$ be a graph and $u, v \in V$. The minimum number of vertices that must be removed to separate u from v is equal to the maximum number of vertex-disjoint paths between u and v.*

This result is foundational, as it forms the basis of many vertex cut algorithms and provides a duality between vertex cuts and independent paths in a graph.

5.3.2 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm, originally developed for solving the maximum flow problem in networks, has important implications for vertex cuts. By applying the max-flow min-cut theorem, one can reduce the vertex cut problem to a flow problem. The algorithm finds the maximum flow in a flow network and, by the max-flow min-cut theorem, identifies the minimum cut.

Theorem 5.3.2. *Max-Flow Min-Cut Theorem: In a flow network, the maximum value of the flow is equal to the capacity of the minimum cut separating the source and sink.*

While the Ford-Fulkerson algorithm primarily addresses edge cuts, it can be adapted for vertex cuts by transforming the graph so that each vertex is split into two vertices connected by an edge. This transformation allows vertex cuts to be treated as edge cuts in a modified network. We will describe this in more detail in the next section.

So, first we transform the graph by replacing each vertex v with two vertices v_{in} and v_{out}, connected by an edge with unit capacity. Then we apply the Ford-Fulkerson algorithm to find the maximum flow between two specified vertices and by the max-flow min-cut theorem, the vertices involved in the minimum cut correspond to the vertex cut in the original graph.

Algorithm 3: Ford–Fulkerson Algorithm

Input: A network $G = (V, E)$ with flow capacity c, a source node s, and a sink node t
Output: Maximum flow f from s to t
Function `FordFulkerson(`*Graph G, Source s, Sink t*`):`
    Initialize $f(u, v) \leftarrow 0$ for all edges $(u, v) \in E$;
    while *there is a path p from s to t in the residual graph G_f* **do**
        Find $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$;
        foreach *edge* $(u, v) \in p$ **do**
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
            $f(v, u) \leftarrow f(v, u) - c_f(p)$
        end
    end
    return *the flow f*;

The complexity of the Ford-Fulkerson algorithm depends on the flow augmentation method used. In its basic form, it runs in $O(m \cdot f)$, where m is the number of edges and f is the maximum flow. More efficient implementations, such as those using the Edmonds-Karp algorithm, run in $O(n^2 \cdot m)$.

We can use the above algorithm to detect if the vertex cut of a graph is at least k in the following way:

Algorithm 4: Ford–Fulkerson for Checking Min Cut at Least k

Input: A network $G = (V, E)$ with flow capacity c, a source node s, a sink node t, and an integer k
Output: Returns true if the min cut is at least k, `false` otherwise
Function `FordFulkersonCheck(`*Graph G, Source s, Sink t, Integer k*`):`
    Initialize $f(u, v) \leftarrow 0$ for all edges $(u, v) \in E$;
    for $i \leftarrow 1$ ***to*** k **do**
        Run BFS to find an augmenting path p from s to t in the residual graph G_f;
        if *no such path exists* **then**
            return `false // Min cut is less than` k
        end
        Find $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$;
        foreach *edge* $(u, v) \in p$ **do**
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
            $f(v, u) \leftarrow f(v, u) - c_f(p)$
        end
    end
    return *true* ; `// Min cut is at least` k

5.3.3 Reduction to Directed Edge Connectivity

In this section we'll discuss how to reduce a vertex connectivity problem into a corresponding directed edge connectivity problem. This uses a standard transformation used in [22], [14] to construct a so-called split graph.

Definition 5.3.1 (Vertex Cut)**.** Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges, and let $s, t \in V$ be two distinct vertices. A set of vertices $S \subseteq V \setminus \{s, t\}$ is called a *vertex cut* if its removal disconnects s from t in G, i.e., after removing S, there is no path between s and t in the resulting graph.

Definition 5.3.2 (Edge Cut)**.** Let $G' = (V', E')$ be a directed graph, where V' is the set of vertices and E' is the set of edges, and let $s', t' \in V'$ be two distinct vertices. A set of edges $E_C \subseteq E'$ is called an *edge cut* if its removal disconnects s' from t' in G', i.e., after removing all edges in E_C, there is no path from s' to t' in the residual graph.

To reduce the vertex cut connectivity problem to a directed edge connectivity problem, we transform the original graph by splitting each vertex, except s and v. Given an undirected graph $G = (V, E)$ with two specified vertices s and t, we construct a directed graph $G' = (V', E')$ as follows:

For each vertex $v \in V$, we create two vertices in V': v_{in} and v_{out}. This effectively splits each original vertex into an inbound node and an outbound node. For each undirected edge $\{u, v\} \in E$, we add two directed edges to E': one from u_{out} to v_{in}, and another from v_{out} to u_{in}. This replacement of each undirected edge with two directed edges allows traversal in both directions through the split nodes. Additionally, for each $v \in V$, we add a directed edge from v_{in} to v_{out} in E'. This edge represents the ability to pass through the original vertex v.

We set the source vertex s' in G' as s_{in} and the sink vertex t' as t_{out}.

5.3.3.1 Proof of Correctness

Let $S \subseteq V \setminus \{s, t\}$ be a vertex cut in G separating s and t. We construct an edge cut E_S in G' by including the edge $(v_{\text{in}}, v_{\text{out}})$ in E_S for each $v \in S$.

Lemma 5.3.3. *E_S separates s' from t' in G'.*

Proof: Assume for contradiction that there exists a path P' from s' to t' in G' that does not use any edge in E_S. This path corresponds to a path P from s to t in G that does not pass through any vertex in S. Each traversal from v_{out} to w_{in} in G' corresponds to traversing edge $\{v, w\}$ in G. The internal edge $(v_{\text{in}}, v_{\text{out}})$ is omitted for $v \in S$, effectively removing v from the traversal. Since S is a vertex cut in G, no such path P can exist, leading to a contradiction. Therefore, E_S separates s' from t' in G'.

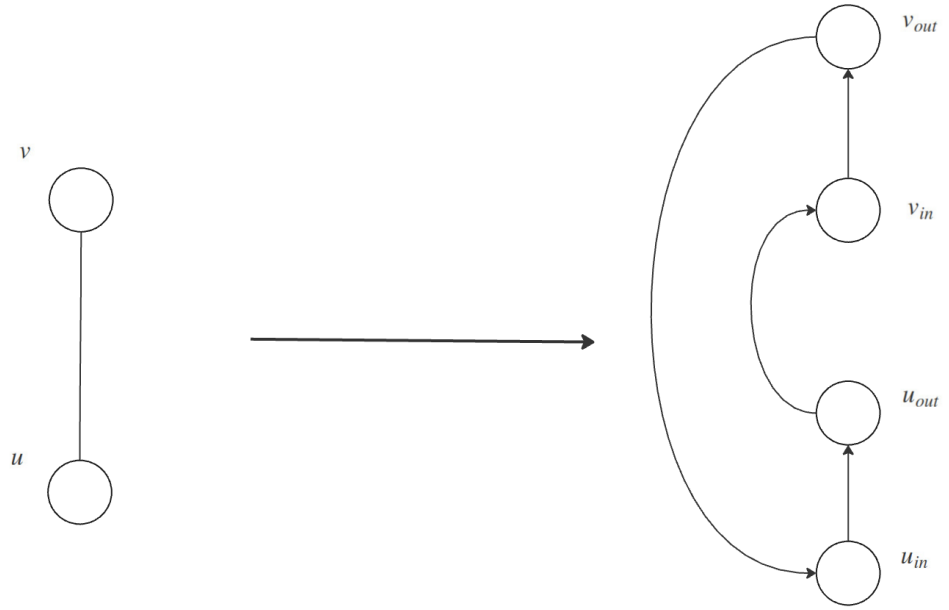


Figure 5.1: Reduction to Directed Edge Connectivity

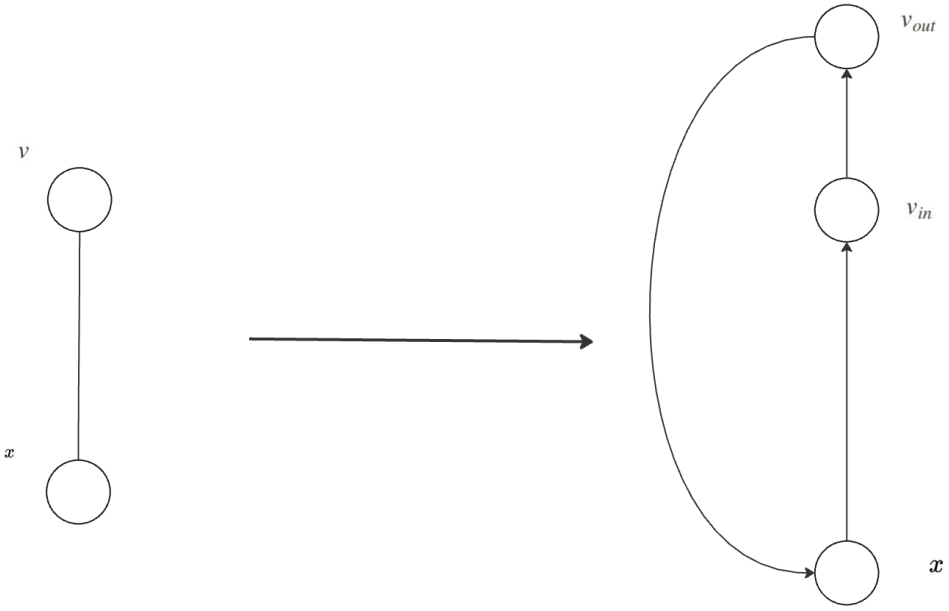


Figure 5.2: Reduction to Directed Edge Connectivity for source

Let E_C be an edge cut in G' separating s' and t'. Define $S = \{v \in V \setminus \{s,t\} \mid (v_{\text{in}}, v_{\text{out}}) \in E_C\}$.

Lemma 5.3.4. *S is a vertex cut in G separating s and t.*

Proof: Assume for contradiction that there exists a path P from s to t in G that does not pass through any vertex in S. This path corresponds to a path P' from s' to t' in G' that does not use any edge in E_C, because for each edge $\{u, v\}$ in P, we traverse from u_{out} to v_{in} or from v_{out} to u_{in} in G'. Since $v \notin S$, the internal edge $(v_{\text{in}}, v_{\text{out}})$ is available for all v in the path.

This contradicts the assumption that E_C separates s' from t' in G'. Therefore, S is a vertex cut in G separating s and t.

The size of the vertex cut S is equal to the size of the edge cut E_C since there is a one-to-one correspondence between vertices in S and edges $(v_{\text{in}}, v_{\text{out}})$ in E_C.

We have established a bijection between vertex cuts in G and edge cuts in G', preserving the size of the cuts. Therefore, finding a minimum vertex cut between s and t in G reduces to finding a minimum edge cut between s' and t' in G'.

5.3.4 Karger's Randomized Contraction Algorithm

In 1993, David Karger introduced a randomized algorithm for the minimum cut problem, which can also be adapted to vertex cuts. Karger's algorithm repeatedly contracts random edges until only two vertices remain. The remaining edges between the two vertices represent a cut. By running the algorithm multiple times, the probability of finding the minimum cut increases.

Algorithm 5: Karger's Randomized Contraction Algorithm

Input: A connected, undirected graph $G = (V, E)$
Output: A pair of vertices that form a minimum cut of G

```
1 Function KargerMinCut(Graph G = (V, E)):
2     while |V| > 2 do
3         Randomly select an edge (u, v) ∈ E;
4         Contract(G, u, v);
5     end
6     return The two remaining vertices;
7 Function Contract(Graph G, Vertex u, Vertex v):
8     Merge vertex v into vertex u;
9     Remove self-loops created during contraction;
```

5.3.4.1 Analysis

Karger's Randomized Contraction Algorithm works by repeatedly contracting randomly chosen edges until only two vertices remain. The probability that this process yields a minimum cut can be computed as follows.

At each step of the algorithm, an edge is selected randomly. For the algorithm to succeed, no edge from the minimum cut must be contracted. Let C be the size of the minimum cut, and let $|V| = n$ be the number of vertices. The probability of choosing an edge that does *not* belong to the minimum cut in the first step is:

$$\frac{|E| - C}{|E|} \geq 1 - \frac{2}{n}$$

This reasoning continues for subsequent contractions. The overall probability of success (i.e., not contracting any edge from the minimum cut) over all iterations of the algorithm is:

$$P_{\text{success}} = \prod_{i=0}^{n-3} \left(1 - \frac{1}{n-i}\right) = \frac{2}{n(n-1)}$$

Thus, the probability of finding the minimum cut in a single execution of Karger's algorithm is $O\left(\frac{1}{n^2}\right)$.

5.3.4.2 Boosting the Probability of Success

Since the probability of success is low, we can apply a technique known as boosting, where the algorithm is run multiple times to increase the probability of finding the minimum cut. If we run the algorithm r times, the probability of failure in each run is $1 - P_{\text{success}}$. Therefore, the probability that all r runs fail is:

$$P_{\text{failure}} = \left(1 - \frac{2}{n(n-1)}\right)^r$$

In order to boost the probability of success, we simply run the algorithm $r\binom{n}{2}$ times. The probability that at least one run succeeds is at least

$$1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{r\binom{n}{2}} \geq 1 - e^{-r}$$

Setting $r = c \ln n$ we have error probability $\leq 1/n^c$, which means the algorithm succeedes with high probability (w.h.p). Therefore we have an $O(n^4 \log n)$ time randomized algorithm with error probability $1/poly(n)$. Specifically, for a success probability of at least $1 - \epsilon$, we require $r = O(n^2 \log \frac{1}{\epsilon})$.

5.3.4.3 Complexity

The time complexity of Karger's algorithm in a single run is dominated by the cost of performing $n - 2$ edge contractions. Each contraction involves merging two vertices and removing self-loops, which can be done in $O(n)$ time. Therefore, the complexity of a single run of Karger's algorithm is:

$$T(n) = O(n^2)$$

5.3.4.4 Karger-Stein Improvement

A faster version of this algorithm was introduced by Karger and Stein in [23]. The key improvement comes from observing the telescoping product behavior during contractions. At the beginning of the process, the probability of contracting an edge in the minimum cut is low. However, as the algorithm progresses, the likelihood of cutting through the minimum cut increases.

From earlier analysis, we know the following: for a given minimum cut $\delta(S)$, the probability that this cut survives until the graph is reduced to ℓ vertices is at least $\left(\frac{\ell}{n}\right)^2$. Therefore, setting $\ell = \frac{n}{\sqrt{2}}$, the probability of success is at least $1/2$. In expectation, two trials should suffice.

The improved version of the algorithm is as follows: Given a multigraph G, if G has at least 6 vertices, repeat the following steps twice:

1. Run Karger's original algorithm until the graph is reduced to $\lceil |V|/\sqrt{2} \rceil$ vertices.
2. Recur on the resulting graph.

Finally, return the minimum of the cuts found in the two recursive calls.

The choice of 6 as the cutoff number of vertices is somewhat arbitrary; changing it would only affect the running time by a constant factor. The running time can be determined via the following recurrence which is straightforward to solve using the standard Master theorem:

$$T(n) = 2\left(n^2 + T\left(\frac{n}{\sqrt{2}}\right)\right) = O(n^2 \log n)$$

Since we preserve the minimum cut with probability $\geq 1/2$, we can express the recurrence for the probability of success, denoted by $P(n)$, as:

$$P(n) \geq 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

This recurrence solves to $P(n) = \Omega\left(\frac{1}{\log n}\right)$. Hence, similar to the argument used for Karger's original algorithm, running the algorithm $O(\log^2 n)$ times ensures that the probability of success is at least $1 - \frac{1}{\text{poly}(n)}$.

Thus, in $O(n^2 \log^3 n)$ total time, the algorithm can find the minimum cut with high probability.

5.4 Local Vertex Cut

5.4.1 Overview

This chapter outlines the key results and techniques from the work of Nanongkai, Saranurak, and Yingchareonthawornchai in [1]. Their work presents new randomized algorithms for small vertex connectivity. In the simplest case where $m = O(n)$ and, hence $k = O(1)$, this algorithm breaks the 49-year-old standing $O(n^2)$ quadratic time barriers as discussed in the introduction. The algorithm is combinatorial, meaning that they do not rely on fast matrix multiplication.

Specifically this paper presents a randomized Monte Carlo algorithm that runs in $\tilde{O}(mk^2)$,time and for small $k = O(\sqrt{n})$, which can decide *w.h.p* if $k_G \geq k$. If $k_G < k$ then the algorithm returns a vertex cut of size less than k.

A core contribution is the development of a local algorithm for computing vertex connectivity. Unlike previous approaches that relied on computing single-source connectivity, which had a quadratic-time lower bound, this local algorithm avoids reading the entire graph. Instead, it focuses on finding a separator of size at most k or certifying that no such separator exists "near" a given seed node. The idea is based on constructing a subgraph around the seed node and exploring only relevant parts of the graph.

The local vertex connectivity algorithm runs in $\tilde{O}(\nu k^2)$ time for a given subgraph of volume ν and connectivity k. It adapts and speeds up the Goldberg-Rao [24] max-flow

algorithm, making it faster when run on a specific structure known as the "augmented graph."

These algorithms improve upon the classical approaches to vertex connectivity and provide efficient solutions to both the exact and approximate problems.

5.4.2 Preliminaries

Definition 5.4.1. A vertex partition (L, S, R) is called a separation triple (or triplet) if $L, R \neq \emptyset$ and there is no edge between L and R, i.e., $N(L) = S = N(R)$. We define the size of (L, S, R) to be $|S|$.

Examples of two different separation triplets in the same graph:

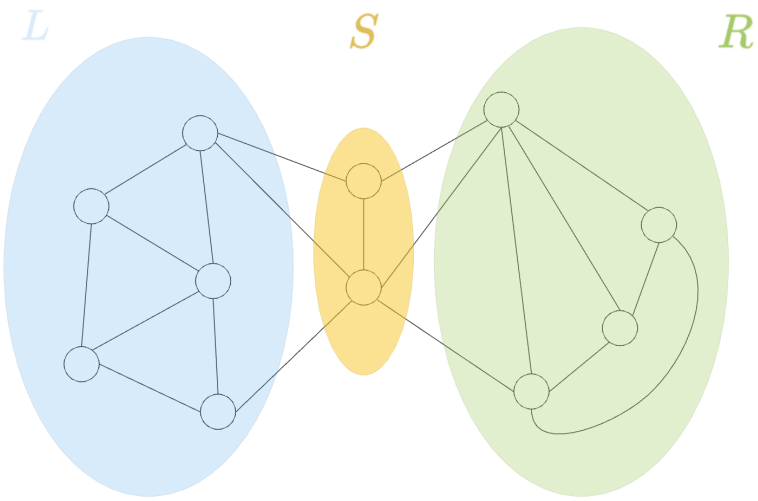


Figure 5.3: Separation triplet

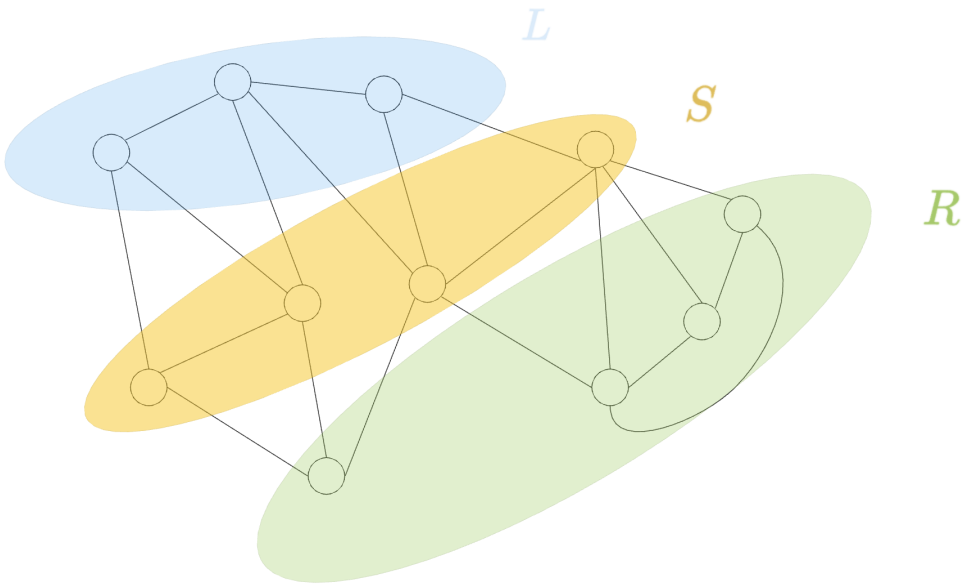


Figure 5.4: Different separation triplet in the same graph

It's important to note that, if a graph G is not k-connected, it means that is has a separation triplet (L, S, R) with $|S| < k$.

Definition 5.4.2 (Edge Set)**.** We define $E(S, T)$ as the set of edges $\{(u, v) : u \in S, v \in T\}$.

Definition 5.4.3 (Separation Triple)**.** A vertex partition (L, S, R) is called a separation triple if $L, R \neq \emptyset$ and $N(L) = N(R) = S$.

Definition 5.4.4 (s-t Connectivity)**.** Let $\kappa(s, t)$ be the minimum number of vertices that must be removed to disconnect s from t, where $s \in L$ and $t \in R$, and (L, S, R) is the partition of the vertices.

Definition 5.4.5 (With High Probability)**.** We say that an event occurs with high probability $(w.h.p)$ if it holds with probability at least $1-1/n^c$, where c is an arbitrarily large constant.

Definition 5.4.6 (Volume)**.** We define the volume of a vertex set X to be $vol(X) = \sum_{x \in X} deg(x)$

Definition 5.4.7 (Edge Sets and L, R-volumes)**.** For a separation triple (L, S, R), we denote

- $E^*(L, S) = E(L, L) \cup E(S, L) \cup E(L, S)$
- $E^*(R, S) = E(R, R) \cup E(S, R) \cup E(R, S)$

- $vol^*_G(L) = \sum_{v \in L} deg^{out}_G(v) + |E(S, L)|$
- $vol^*_G(R) = \sum_{v \in R} deg^{out}_G(v) + |E(S, R)|$

We observe that $m = vol^*_G(L) + |E(S, S)| + vol^*_G(R)$

5.4.3 The Algorithm

The vertex connectivity of a graph G, denoted as k_G, is defined as the minimum number of vertices that need to be removed to disconnect the graph. The paper from [1] presents a randomized Monte Carlo algorithm that runs in $\tilde{O}(mk^2)$ time and for small $k = O(\sqrt{n})$, can decide with high probability (*w.h.p*) if $k_G \geq k$. If $k_G \geq k$ then the algorithm returns a vertex cut of size less than k.

The algorithm is divided into two main phases. Given that the graph G is not k-connected, it means that there must be a separation triplet (L, S, R) with $|S| < k$.

- The first phase samples random edge pairs and checks if there is a small vertex cut using Ford-Fulkerson. This detects the vertex cut w.h.p in the case where L and R are roughly the same size.
- The second phase uses local exploration to detect cuts efficiently when the volume of one side of the partition is small.

If the two phases don't detect a vertex cut, we can conclude *w.h.p.* that G is k-connected.

Algorithm 6: Vetrex Cut(Sample, LocalVC, FordFulkersonCheck, BFS)

Input: G, k
Output: A vertex cut S with $|S| < k$ if it exists, else $\perp$

$n \leftarrow |V|$
$m \leftarrow |E|$
for $i \leftarrow 1$ **to** $k \log n$ **do**
  Sample a random pair of edges $e = (x, x'), f = (y, y')$
  for $(s, t) \leftarrow \{(x, y), (x', y'), (x', y), (x, y')\}$ **do**
    if FordFulkersonCheck(G, s, t, k) **then**
      $T_{BFS} \leftarrow \text{BFS}(G_f, s)$
      return $N_G^{out}(T_{BFS})$
    end
  end
end
for $i \leftarrow 1$ **to** $\log(m/k)$ **do**
  for $j \leftarrow 1$ **to** $m \log(n)/2^i$ **do**
    Sample a random edge $e = (x, y)$
    for $v \in \{x, y\}$ **do**
      $L' \leftarrow \text{LocalVC}(G, v, 2^i, k)$
      if $L' \neq \perp$ **then**
        return $N_G^{out}(L')$
      end
    end
  end
end
return $\perp$

The Ford-Fulkerson subroutine used in this algorithm is modified to just check if $\kappa(s, t) \leq k$ as described in Algorithm 4.

The main theorem that we'll prove in this section is stated as:

Theorem 5.4.1. *Given a directed graph $G = (V, E)$ and $k = O(\sqrt{n})$, if G is not k-connected then w.h.p Algorithm 6 finds a separator S with size less than k. If G is k-connected then Algorithm 6 always returns $\perp$. The algorithm takes $\tilde{O}(mk^2)$ time.*

To simplify the analysis we assume without loss of generality that $vol(L) < vol(R)$.

Algorithm 7: LocalVC(Sample, DFS)

Input: G, x, ν, k
Output: A set L' containing x with $|N(L')| < k$ and $vol(L') < \nu$ if it exists, else $\perp$

```
for j ← 0 to log(n) do
    for i ← 1 to k do
        T_DFS ← DFS(G, x) exploring exactly kν edges
        if |T_DFS| < kν then
            return N_G^out(T_DFS)
        end
        Sample a random edge e = (y', y) from T_DFS
        Reverse the path P_xy
    end
end
return ⊥
```

5.4.4 Analysis of Balanced Partition: $\mathbf{vol}^*(L) \geq m/k$

In this case, we analyze the scenario where the smaller partition L is sufficiently large, such that its volume is approximately equal to the volume of the larger partition R. In this case: $vol^*_G(L) \geq m/k$.

Lemma 5.4.2. *If G has a separation triple (L, S, R) with $|S| < k$ and $vol^*_G(L) \geq m/k$, then w.h.p Algorithm 6 outputs a separator of size less than k.*

Proof. The algorithm proceeds by randomly sampling pairs of edges (x, x') and (y, y') from the edge set E, and then testing the connectivity between pairs of vertices formed from these edges, i.e., $(x, y), (x', y'), (x', y), (x, y')$. The goal is to sample one vertex in L and one in R so we can detect a small cut S, if that exists.

For each sampled vertex pair (s, t), the algorithm runs the Ford-Fulkerson max-flow algorithm to check if the minimum vertex cut between s and t is less than k. If the minimum cut is found to be smaller than k, this indicates the existence of a small separator in the graph. The algorithm then performs a Breadth-First Search (BFS) on the residual graph to identify the set of vertices T_{BFS} reachable from s, and returns the out-neighborhood $N^{\text{out}}_G(T_{\text{BFS}})$ as the separator.

If no such separator is found after $k \log n$ iterations, the algorithm concludes that the graph is k-connected and returns $\perp$. The choice of $k \log n$ is to ensure that we detect the small cut *w.h.p* if it exists, while maintaining total work of $\tilde{O}(mk^2)$ for this step.

Now we prove that if $vol(L) \geq m/k$ then by sampling $\tilde{O}(k)$ pairs of edges e_1, e_2 we get

w.h.p one edge in $E^*(L,S)$ and one in $E^*(R,S)$.

$$\begin{aligned} Pr(e1 \in E^*(L,S), e2 \in E^*(R,S)) &= Pr(e_1 \in E^*(L,S))Pr(e_2 \in E^*(R,S)) \\ &= \frac{|E^*(L,S)|}{m}\frac{|E^*(R,S)|}{m} \\ &= \frac{vol^*_G(L)}{m}\frac{vol^*_G(R)}{m} \end{aligned} \tag{5.1}$$

Since $vol^*_G(L) + vol^*_G(R) = \Omega(m)$, either $vol^*_G(L)$ or $vol^*_G(R) = \Omega(m)$

$$vol^*_G(L) \geq m/k, (5.1) \implies Pr(e1 \in E^*(L,S), e2 \in E^*(R,S)) > 1/k$$

The probability that we don't get an edge in each side if we repeat the process T times is $(1-\frac{1}{k})^T \leq e^{-T/k}$. By taking $T = klogn$ we get that the probability of failure is less than $1/n$.

■

5.4.5 Analysis of Imbalanced Partition: Small $\mathbf{vol}^*_G(L) < m/k$

In this case, the partition is highly imbalanced, where the volume of L is significantly smaller than the volume of R. In this case: $vol^*_G(L) < m/k$. For this case we'll use $LocalVC(G, x, \nu, k)$ (Algorithm 7).

First we make the reduction from vertex connectivity to edge connectivity using the split graph as we described in Section 5.3.3.

Definition 5.4.8 (Local Vertex Connectivity)**.** Given a directed graph G, a node x, and integers ν, k, then if there exists a set L' containing x with $|N(L')| < k$ and $vol(L') < \nu$ then Algorithm 7 returns it *w.h.p* Otherwise it returns $\perp$.

Definition 5.4.9 (Local Edge Connectivity)**.** Given a directed graph G, a node x, and integers ν, k, return a set L' containing x with $|\delta_{out}(L')| < k$ and $vol(L') < \nu$ if it exists, otherwise return $\perp$.

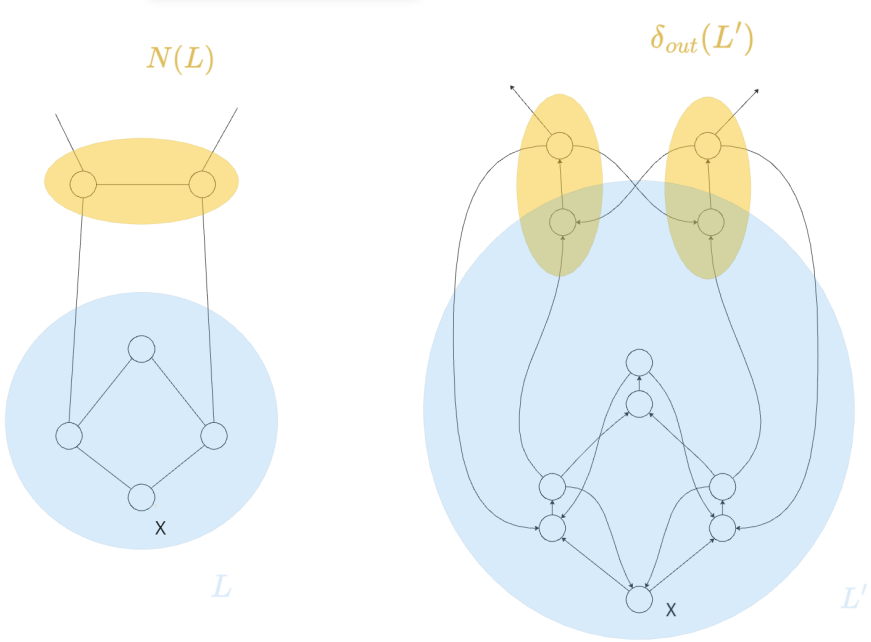


Figure 5.5: Reduction to Directed Edge Connectivity Degree

After the reduction, the size of the neighborhoud of L, $N(L)$ is equal to $|\delta_{out}(L')|$ and $vol^*(L) = \Theta(vol(L')$.

Since $vol^*_G(L) \leq m/k$ there exists an integer $i \in [0, \log(m/k)]$ s.t. $vol^*_G(L) \in [2^{i-1}, 2^i]$. First we prove the following claim.

The goal in this step is, again, to sample enough edges so that *w.h.p.* we get an edge inside this small component L. Then for each sampled vertex we attempt to identify a small vertex cut *locally*, "near" this seed vertex x. To do this more efficiently we bound the size of the small component L between two powers of 2.

Lemma 5.4.3. *If $vol(L) \in [2^{i-1}, 2^i]$ then w.h.p by sampling $\tilde{O}(m/2^i)$ edges we get an edge in $E^*(L, S)$.*

Proof.

$$Pr(e \in E^*(L,S)) = \frac{vol^*_G(L)}{m} \geq \frac{2^{i-i}}{m}$$

The probability that we don't get an edge in $E^*(L, S)$ if we repeat the process T times is $(1 - \frac{2^{i-1}}{m})^T \leq e^{-2^{i-1}T/m}$. By taking $T = mlogn/2^{i-1}$ we get that the probability of failure is less than $1/n$. So we get an edge in $E^*(L, S)$ *w.h.p.*

■

In the algorithm we check for all possible values of $vol(L)$ between 1 and m/k (which is the upper bound for this case).

5.4.6 Warm-up Case: L with $\delta_{\text{out}}(L) = 1$ and $\delta_{\text{in}}(L) = 0$

Before diving into the more general case, let us first consider a simplified scenario where the set L contains a single outgoing edge and no incoming edges, i.e., $\delta_{\text{out}}(L) = 1$ and $\delta_{\text{in}}(L) = 0$. This basic example will help illustrate how the algorithm operates when the out-degree of L is $\leq k$.

The algorithm begins by performing a depth-first search (DFS) starting from the seed node x. DFS in this case needs to explore $\nu + 1$ edges to guarantee that it leaves the small component L and since $\delta_{\text{in}}(L) = 0$ there is no way for DFS to return to L.

Then we take the vertex where the DFS exploration ends y, and reverse the path P_{xy} from x to y. This operation removes the single outgoing edge from L, thus reducing $\delta_{\text{out}}(L)$ to zero.

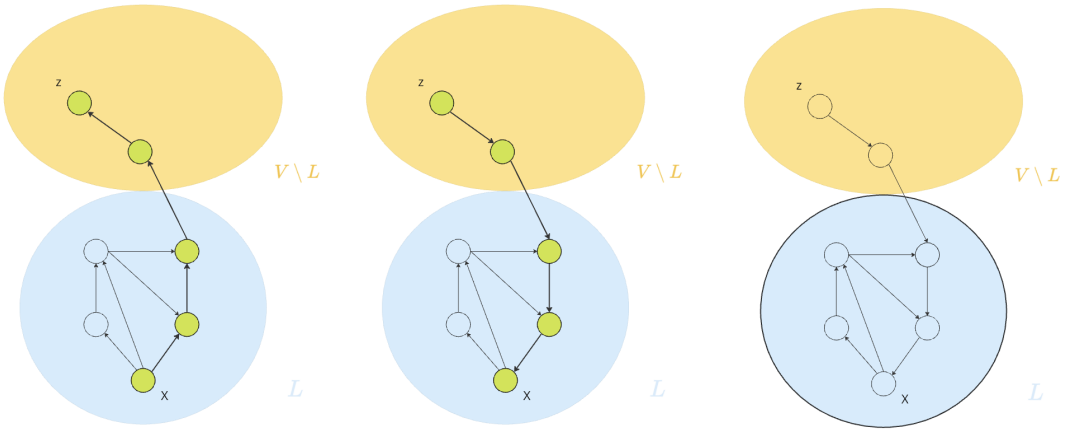


Figure 5.6: Local Vertex Cut Warmup Case

At this point, with no outgoing edges left, the DFS will get stuck inside L during the next iteration, as it will no longer be able to explore nodes outside the set. Consequently, the algorithm identifies a valid cut $N_G^{\text{out}}(T_{\text{DFS}})$ and terminates.

This warm-up case provides a simple demonstration of the key concepts underlying the algorithm: exploration using DFS, path reversal, and reduction of $\delta_{\text{out}}(L)$. These same principles apply in the more general case where $\delta_{\text{out}}(L) \geq 1$ and $\delta_{\text{in}}(L) \neq 0$.

5.4.7 General Case

The LocalVC algorithm takes three parameters: a seed node x, a volume threshold ν, and a cut size k. Its goal is to find a small set L' containing x such that $\delta_{\text{out}}(L') < k$ and $\text{vol}(L') < \nu$, or return $\perp$ if no such set exists.

The volume threshold takes it's value from 2^i as discussed before, and is used to efficiently sample the graph so that *w.h.p.* we get a node inside the small component L. It also plays a role on how many edges we'll explore with DFS. In this case where $\delta_{\text{in}}(L) \neq 0$, there is the chance that DFS ends up inside L again. To mitigate this, we explore enough nodes, $k\nu$ to ensure that the sampled vertex of T_{DFS} is outside L.

The algorithm consists of two nested loops. The outer loop runs from 1 to $\log(n)$ and serves as a boosting mechanism to ensure that the algorithm succeeds *w.h.p.* The inner loop, which runs for k iterations, is responsible for identifying whether the size of the cut is less than k by eliminating outgoing edges one at a time.

During each iteration of the inner loop, a depth-first search (DFS) is performed from the seed node x. The DFS explores up to $k\nu$ nodes to have a high probability that the search extends beyond the boundary of L.

At the end of each DFS, the algorithm samples a random edge (y', y) from the explored tree and reverses the path P_{xy} between x and y. Each path reversal eliminates an outgoing edge from L', progressively reducing $\delta_{\text{out}}(L')$. The inner loop repeats this process for k iterations, each time attempting to reduce the out-degree by one.

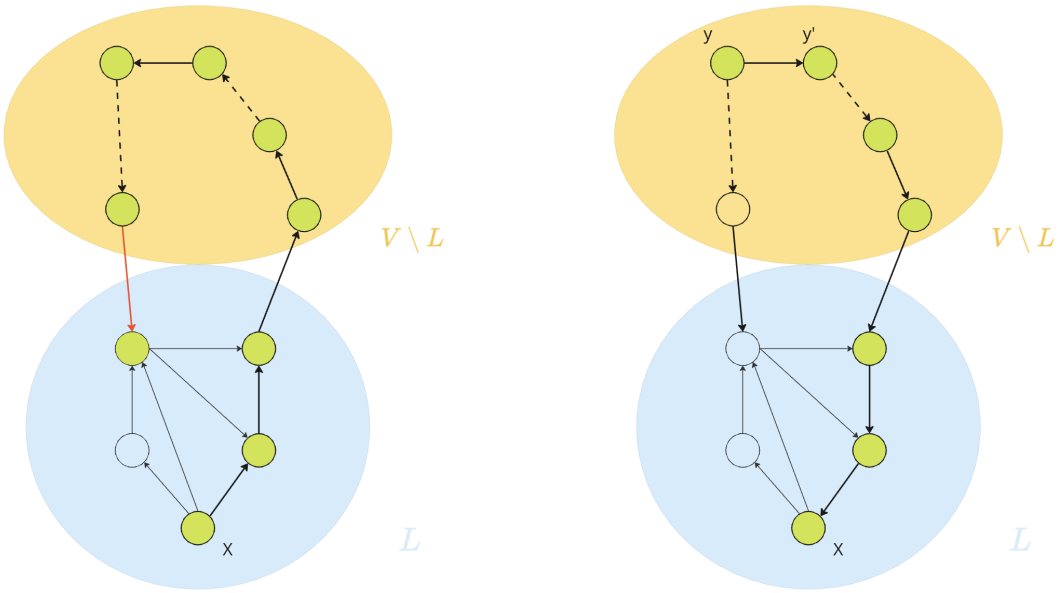


Figure 5.7: Local Vertex Cut General Case

If fewer than $k\nu$ nodes are explored, i.e. DFS gets "stuck", it implies that we found a set L that contains x with volume smaller than ν and $|N_G^{out}(T_{DFS})| < k$, thus identifying a small vertex cut $N_G^{out}(T_{DFS})$.

Lemma 5.4.4. *If G has a separation triple (L, S, R) with $|S| < k$ and $vol_G^*(L) < m/k$, then w.h.p Algorithm 6 outputs a separator of size less than k.*

Lemma 5.4.5. *If Algorithm 7 returns a set L', then $\delta_{out}(L') < k$.*

Proof. As we mentioned, during the execution of the algorithm, a depth-first search (DFS) is performed from the seed node x to explore up to $k\nu$ nodes. If fewer than $k\nu$ nodes are explored, it implies that the volume of the current set L' is less than ν, and the algorithm terminates, returning L'.

Throughout the iterations, the algorithm randomly samples edges from the DFS tree and reverses the paths between x and the sampled node y. If y is not contained in L', reversing the path reduces the out-degree $\delta_{\text{out}}(L')$ by one. If the algorithm returns a set L', this implies that the out-degree $\delta_{\text{out}}(L')$ became zero, and since fewer than k path reversals occurred, the initial out-degree must have been smaller than k. Therefore, if the algorithm returns L', we can conclude that $\delta_{\text{out}}(L') < k$. ■

Lemma 5.4.6. *If the algorithm returns $\perp$, then w.h.p, there is no set L' containing x such that $|\delta_{out}(L')| < k$ and $vol(L') < \nu$.*

Proof. Suppose that there exists a set L' containing x, such that $|\delta_{\text{out}}(L')| < k$ and $\text{vol}(L') < \nu$. The algorithm performs a depth-first search from x, followed by the sampling of a random edge from the explored DFS tree. For each sampled edge, the algorithm attempts to reduce the out-degree $\delta_{\text{out}}(L')$ by reversing the path P_{xy}.

The probability that the node y is not contained within L, i.e. $\delta_{\text{out}}(L')$ is decreased by one, is at least $1 - \frac{\nu}{k\nu} = 1 - \frac{1}{k}$. After $k-1$ iterations, the probability that $\delta_{\text{out}}(L') = 0$ is $\left(1 - \frac{1}{k}\right)^{k-1} \geq \frac{1}{10}$.

Since we have a constant probability of success, we can boost this by running the outer loop $\tilde{O}(1)$ times to guarantee *w.h.p* that if a valid set L' exists, it will be found. We do this in Algorithm 7 in line 1. If the algorithm terminates without returning a set, it concludes that *w.h.p* no such set L' exists that satisfies $|\delta_{\text{out}}(L')| < k$ and $\text{vol}(L') < \nu$. ■

5.4.8 Running Time Analysis

The running time of the algorithm is determined by two main steps: checking the maximum flow between two nodes and the local vertex connectivity procedure.

The Ford-Fulkerson algorithm can check whether the maximum flow between two nodes is less than k in $O(mk)$ time, where m is the number of edges. Additionally, computing the out-neighborhood of a set and running a breadth-first search (BFS) takes linear time, i.e., $O(m)$. Hence, the first part of the algorithm, which involves running Ford-Fulkerson and BFS, takes a total time of $\tilde{O}(mk^2)$.

For the second part of the algorithm, the LocalVC procedure takes $O(\nu k^2 \log n)$ time, where ν represents the volume threshold. We run LocalVC for all $i \in [1, \log m/k]$ sampling $\tilde{O}(m/2^i)$ edges each time. Parameter ν takes value 2^i for each i. The total time for this part of the algorithm can be written as:

$$O\left(\log\left(\frac{m}{k}\right) \cdot \frac{m\log(n)}{\nu}\right) \cdot O(\nu k^2 \log(n)) = \tilde{O}(mk^2)$$

Thus, combining the running time of both steps, the total time complexity of the algorithm is $\tilde{O}(mk^2)$. This concludes the proof of Theorem 5.4.1.

5.5 Local Vertex Cut with vertex Sampling

In this section we present a variant of the algorithm from [1] but sampling vertices instead of edges. Later in Chapter 7 where we attempt to parallelize this algorithm we'll see this gives us improved work in some cases.

The setup is the same. The algorithm is divided into two main phases. Given that the graph G is not k-connected, it means that there must be a separation triplet (L, S, R) with $|S| < k$.

- The first phase samples random vertex pairs and checks if there is a small vertex cut using Ford-Fulkerson. This detects the vertex cut w.h.p in the case where L and R are roughly the same size.
- The second phase uses local exploration to detect cuts efficiently when the volume of one side of the partition is small. This time we sample vertices instead of edges to get a vertex inside the small partition.

If the two phases don't detect a vertex cut, we can conclude *w.h.p.* that G is k-connected. For any subgraph $H \subseteq G$, we define $n_H = |V(H)|$. The main theorem that we'll prove in this section is stated as:

Theorem 5.5.1. *Given a directed graph $G = (V, E)$ and $k = O(\sqrt{n})$, if G is not k-connected then w.h.p Algorithm 8 finds a separator S with size less than k. If G is k-connected then Algorithm 8 always returns $\perp$. The algorithm takes $\tilde{O}(n^2k^2)$ time.*

Algorithm 8: Vetrex Cut with Vertex Sampling(Sample, LocalVC, FordFulkersonCheck, BFS)

Input: G, k
Output: A vertex cut S with $|S| < k$ if it exists, else $\perp$

$n \leftarrow |V|$
$m \leftarrow |E|$
for $i \leftarrow 1$ **to** $k \log n$ **do**
  Sample a random pair of vertices x, y
  if FordFulkersonCheck(G, s, t, k) **then**
    $T_{BFS} \leftarrow \text{BFS}(G_f, s)$
    return $N_G^{out}(T_{BFS})$
  end
end
for $i \leftarrow 1$ **to** $\log(n/k)$ **do**
  for $j \leftarrow 1$ **to** $n \log(n)/2^i$ **do**
    Sample a random vertex x
    $L' \leftarrow \text{LocalVC}(G, x, 2^i, k)$
    if $L' \neq \perp$ **then**
      return $N_G^{out}(L')$
    end
  end
end
return $\perp$

5.5.1 Analysis of Balanced Partition: $n_L \geq n/k$

In this case, we analyze the scenario where the smaller partition L is sufficiently large, such that it has approximately the same amount of vertices as the larger partition R. In this case: $n_L \geq n/k$.

Lemma 5.5.2. *If G has a separation triple (L, S, R) with $|S| < k$ and $n_L \geq n/k$, then w.h.p Algorithm 8 outputs a separator of size less than k.*

Proof. The algorithm proceeds by randomly sampling pairs of vertices (x, y) from the vertex set V, and then testing the connectivity between these vertices. The goal is to sample one vertex in L and one in R so we can detect a small cut S, if that exists.

For each sampled vertex pair (x, y), the algorithm runs the Ford-Fulkerson max-flow algorithm to check if the minimum vertex cut between x and y is less than k. If the minimum cut is found to be smaller than k, this indicates the existence of a small separator in the graph. The algorithm then performs a Breadth-First Search (BFS)

on the residual graph to identify the set of vertices T_{BFS} reachable from x, and returns the out-neighborhood $N_G^{\text{out}}(T_{\text{BFS}})$ as the separator.

If no such separator is found after $k \log n$ iterations, the algorithm concludes that the graph is k-connected and returns $\perp$. The choice of $k \log n$ is to ensure that we detect the small cut *w.h.p* if it exists, while maintaining total work of $\tilde{O}(mk^2)$ for this step.

Now we prove that if $n_L \geq n/k$ then by sampling $\tilde{O}(k)$ pairs of vertices x, y we get *w.h.p* one vertex in L and one in R.

$$Pr(x \in L, y \in R) = Pr(x \in L)Pr(y \in R) = \frac{n_L}{n}\frac{n_R}{n} \tag{5.2}$$

Since $n_L + n_R = \Omega(n)$, either n_L or $n_R = \Omega(n)$

$$n_L \geq n/k, (5.2) \implies Pr(x \in L, y \in R) > 1/k$$

The probability that we don't get a vertex in each side if we repeat the process T times is $(1 - \frac{1}{k})^T \leq e^{-T/k}$. By taking $T = klogn$ we get that the probability of failure is less than $1/n$.

■

5.5.2 Analysis of Imbalanced Partition: $n_L < n/k$

In this case, the partition is highly imbalanced, where the number of vertices of L is significantly smaller than those of R. In this case: $n_L < n/k$. For this case we'll use the variant of *LocalVC* (Algorithm 9) with vertex sampling.

As in the previous case, we use the reduction to directed edge connectivity and we bound the number of nodes in L between powers of two. Since $n_L \leq n/k$ there exists an integer $i \in [0, \log(n/k)]$ s.t. $n_L \in [2^{i-1}, 2^i]$. First we prove the following claim.

Lemma 5.5.3. *If $n_L \in [2^{i-1}, 2^i]$ then w.h.p by sampling $\tilde{O}(n/2^i)$ vertices we get a vertex in L.*

Proof.

$$Pr(x \in L) = \frac{n_L}{n} \geq \frac{2^{i-i}}{n}$$

The probability that we don't get a vertex in L if we repeat the process T times is $(1 - \frac{2^{i-1}}{n})^T \leq e^{-2^{i-1}T/n}$. By taking $T = nlogn/2^{i-1}$ we get that the probability of failure is less than $1/n$. So we get a vertex in L *w.h.p.*

■

In the algorithm we check for all possible values of n_L between 1 and n/k (which is the upper bound for this case).

Algorithm 9: LocalVCVertex(Sample, DFS)

Input: G, x, ν, k
Output: A set L' containing x with $|N(L')| < k$ and $n_{L'} < \nu$ if it exists, else $\perp$

```
for j ← 0 to log(n) do
    for i ← 1 to k do
        T_DFS ← DFS(G, x) exploring exactly kν vertices
        if |T_DFS| < kν then
            return N_G^out(T_DFS)
        end
        Sample a random vertex y from T_DFS
        Reverse the path P_xy
    end
end
return ⊥
```

The setup is quite similar as the edge sampling case, Algorithm 9 takes three parameters: a seed node x, a size threshold ν, and a cut size k. Its goal is to find a small set L' containing x such that $\delta_{\text{out}}(L') < k$ and $n_{L'} < \nu$, or return $\perp$ if no such set exists.

The size threshold takes it's value from 2^i as discussed before, and is used to efficiently sample the graph so that *w.h.p.* we get a node inside the small component L. It also plays a role on how many vertices we'll explore with DFS. In the general case where $\delta_{\text{in}}(L) \neq 0$, there is the chance that DFS ends up inside L again. To mitigate this, we explore enough nodes, $k\nu$ to ensure that the sampled vertex of T_{DFS} is outside L.

The algorithm consists of two nested loops. The outer loop runs from 1 to $\log(n)$ and serves as a boosting mechanism to ensure that the algorithm succeeds *w.h.p.* The inner loop, which runs for k iterations, is responsible for identifying whether the size of the cut is less than k by eliminating outgoing edges one at a time.

During each iteration of the inner loop, a depth-first search (DFS) is performed from the seed node x. DFS explores up to $k\nu$ nodes to have a high probability that when we sample a vertex in the next step, this is beyond the boundary of L.

At the end of each DFS run, the algorithm samples a random vertex y from the explored tree and reverses the path P_{xy} between x and y. Each path reversal eliminates an outgoing edge from L', progressively reducing $\delta_{\text{out}}(L')$. The inner loop repeats this process for k iterations, each time attempting to reduce the out-degree by one.

If fewer than $k\nu$ nodes are explored, i.e. DFS gets "stuck", it implies that we found a set

L that contains x with volume smaller than ν and $|N_G^{out}(T_{DFS})| < k$, thus identifying a small vertex cut $N_G^{out}(T_{DFS})$.

The proof for the following two lemmas is identical to the case where we sample edges so we omit them,

Lemma 5.5.4. *If G has a separation triple (L, S, R) with $|S| < k$ and $n_L < n/k$, then w.h.p Algorithm 8 outputs a separator of size less than k.*

Lemma 5.5.5. *If Algorithm 9 returns a set L', then $\delta_{out}(L') < k$.*

Lemma 5.5.6. *If the algorithm returns $\perp$, then w.h.p, there is no set L' containing x such that $|\delta_{out}(L')| < k$ and $n_{L'} < \nu$.*

Proof. Suppose that there exists a set L' containing x, such that $|\delta_{\text{out}}(L')| < k$ and $n_{L'} < \nu$. The algorithm performs a depth-first search from x, followed by the sampling of a random vertex y from the explored DFS tree. For each sampled vertex, the algorithm attempts to reduce the out-degree $\delta_{\text{out}}(L')$ by reversing the path P_{xy}.

The probability that the node y is not contained within L, i.e. $\delta_{\text{out}}(L')$ is decreased by one, is at least $1 - \frac{\nu}{k\nu} = 1 - \frac{1}{k}$. After $k-1$ iterations, the probability that $\delta_{\text{out}}(L') = 0$ is $\left(1 - \frac{1}{k}\right)^{k-1} \geq \frac{1}{10}$.

Since we have a constant probability of success, we can boost this by running the outer loop $\tilde{O}(1)$ times to guarantee *w.h.p* that if a valid set L' exists, it will be found. We do this in Algorithm 9 in line 1. If the algorithm terminates without returning a set, it concludes that *w.h.p* no such set L' exists that satisfies $|\delta_{\text{out}}(L')| < k$ and $n_{L'} < \nu$. ■

5.5.3 Running Time Analysis

For the second part of the algorithm, LocalVC takes $\tilde{O}(\nu^2 k^3)$ time, where ν is the size threshold. This happens because when exploring νk vertices we can have up to $O(\nu^2 k^2)$ edges. We do this k times and also we boost the probability of success by running the whole thing $\log n$ times, thus the $\tilde{O}(\nu^2 k^3)$ complexity. In total:

$$\frac{n}{\nu}\tilde{O}(\nu^2 k^3) = \tilde{O}(n\nu k^3)$$

We run this for all $\nu = 2^i$ for $i \in [1, \log n/k]$ so the total time for this part of the algorithm can be written as:

$$\sum_{i=1}^{\log n/k} \tilde{O}(n 2^i k^3) = \tilde{O}(n^2 k^2)$$

Thus, combining the running time of both steps, the total time complexity of the algorithm is $\tilde{O}(mk^2 + n^2k^2) = \tilde{O}(n^2k^2)$ since $m = O(n^2)$. This concludes the proof of Theorem 5.5.1.

Chapter 6

Parallel Reachability

6.1 Introduction

Given a directed graph (or digraph) $G = (V, E)$ with n vertices and m edges, and a specific vertex $s \in V$, the *single-source reachability problem* involves determining the set of vertices $T \subseteq V$ that can be reached from s. This means identifying all vertices $t \in V$ for which there exists a path from s to t in G. This problem is one of the most fundamental in graph optimization. It can be efficiently solved in linear time, $O(n + m)$, using classic graph traversal algorithms like breadth-first search (BFS) or depth-first search (DFS). Often introduced in basic algorithms courses, the reachability problem plays a crucial role in both theoretical and practical applications, serving as a foundational step in solving more advanced problems such as finding strongly connected components, shortest paths, maximum flow, and spanning arborescences.

Single-source reachability is a fundamental challenge in parallel computation models, and achieving efficient algorithms for it has been notoriously difficult. Despite the simplicity of solving reachability in optimal sequential time, finding optimal algorithms under parallelism remains a significant obstacle. Until the recent breakthrough by Fineman [25], all known parallel reachability algorithms involved trade-offs between depth and work, with any algorithm achieving linear work requiring the trivial $O(n)$ depth.

In this chapter, we analyze the work of Sidford et al. [2] who propose novel techniques to improve the parallel and distributed computation of reachability by leveraging *hopsets* and *shortcuts*. These techniques significantly reduce graph diameter, allowing for faster computations while maintaining near-optimal work complexity.

The work from [2] builds upon a breakthrough of Fineman [25] and utilizes a powerful decompositional technique for improving reachability, known as *hopsets* or *short-*

cuts. Shortcuts are additional edges introduced into the graph without altering the set of reachable vertex pairs. In a graph with diameter D, representing the maximum shortest-path distance between any two reachable vertices, reachability can be computed with $O(m)$ work and $O(D)$ depth. Thus, a natural direction for enhancing reachability algorithms is to identify a small set of shortcuts that reduce the graph's diameter, ideally computed in nearly linear time.

The use of shortcuts holds great promise for optimizing reachability algorithms. A simple probabilistic argument shows that for any n-node directed graph, there exists a set of $O(t^2 \log^2 n)$ shortcuts, which when added, can reduce the graph's diameter to at most $O(n/t)$. For instance, by adding approximately $O(n \log^2 n)$ shortcuts, it is possible to shrink the diameter to $O(\sqrt{n})$. Finding such shortcuts in nearly linear time and $O(\sqrt{n})$ depth would directly result in reachability algorithms with linear work and $O(\sqrt{n})$ depth. However, constructing such shortcuts in almost linear time with this level of diameter reduction remained an open problem until recently.

Fineman's work [25] made significant progress by providing the first nearly linear time algorithm that computes a nearly linear number of shortcuts, achieving polynomial reductions in graph diameter from the trivial $O(n)$ bound. Specifically, Fineman's algorithm computes a set of shortcuts that reduces the diameter to $O(n^{2/3})$ in nearly linear time and then leverages this to design a parallel reachability algorithm with $O(m)$ work and $O(n^{2/3})$ depth, which succeeds with high probability.

Despite this breakthrough, the question of whether parallel algorithms with almost linear work can achieve the depth bound predicted by hopsets, namely $O(n^{1/2})$, remained unanswered. Furthermore, it was unclear whether these improvements could be extended to other resource-constrained computational problems. The paper from [2] addressed both questions with an algorithm that achieves $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ depth, computing a set of $\tilde{O}(n)$ shortcuts that reduces the diameter to $n^{1/2+o(1)}$. The results are achieved by refining and simplifying aspects of Fineman's algorithm.

6.2 Folklore Hopset Construction for Diameter Reduction

In this chapter, we present a folklore way of reducing the diameter of a given graph G by constructing hopsets using probabilistic arguments. The *diameter* of G, denoted as $\text{diam}(G)$ or D, is the maximum distance between any pair of vertices in G. A hopset H is a set of edges that, when added to the graph, preserve the reachability relationships while reducing the diameter.

We show that via a probabilistic construction, it is possible to reduce D to $O(\sqrt{n})$ by adding $\tilde{O}(n)$ edges, where n is the number of vertices in G, and $\tilde{O}$ hides polylogarithmic

factors.

Theorem 6.2.1. *Given a connected graph $G = (V, E)$ with n vertices and m edges, we can add $\tilde{O}(n)$ shortcut edges to G to obtain a new graph $G' = (V, E')$ that preserves the reachability relations in G and has diam$(G') = O(\sqrt{n})$.*

The algorithm to construct the hopset is as follows:

Algorithm 10: Hopset Construction for Reducing Graph Diameter

Input: Graph $G = (V, E)$ with n vertices and m edges
Output: Graph $G' = (V, E \cup H)$ with additional shortcut edges to reduce the diameter
Function `HopsetConstruction(`*Graph* G`):`
    $S \leftarrow \emptyset$
    $H \leftarrow \emptyset$
    for $i \leftarrow 1$ **to** $\lceil \sqrt{n} \rceil$ **do**
        Randomly select a vertex $v_i \in V$
        $S \leftarrow S \cup \{v_i\}$
    end
    for each pair $(u, v) \in S \times S$ **do**
        if u can reach v in G **then**
            $H \leftarrow H \cup \{(u, v)\}$
        end
    end
    return $G' = (V, E \cup H)$

We now provide a formal proof that the above algorithm reduces the diameter of G to $\tilde{O}(n^{1/2})$ with high probability.

6.2.1 Preliminaries

Definition 6.2.1 (Shortest Path)**.** Let $G = (V, E)$ be an undirected graph. For any pair of vertices $s, t \in V$, let $\pi(s, t)$ denote a shortest path from s to t, and let $L = |\pi(s, t)|$ be the length of this path, defined as the number of edges in $\pi(s, t)$.

Definition 6.2.2 ($\tilde{O}(f(n))$ Notation)**.** For a function $f(n)$, the notation $\tilde{O}(f(n))$ represents the class of functions $O(f(n) \log^k n)$, where $k \geq 0$ is a constant and polylogarithmic factors are hidden.

Definition 6.2.3 (With High Probability)**.** An event A occurs *with high probability (whp)* if there exists a constant $c > 0$ such that $\Pr[A] \geq 1 - \frac{1}{n^c}$, where n is the problem size. This indicates that the probability of the event's failure decays polynomially with increasing n.

Definition 6.2.4 (Chernoff Bound)**.** Let $X = \sum_{i=1}^{n} X_i$ be the sum of independent Bernoulli random variables, where $X_i \in \{0, 1\}$ and $\Pr[X_i = 1] = p$. Let $\mu = \mathbb{E}[X]$ be the expected value of X. Then, for any $\delta > 0$, the Chernoff bound gives an exponentially decreasing bound on the probability that X deviates from its expected value:

$$\Pr[X \geq (1 + \delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2 + \delta}\right).$$

Similarly, the lower tail is bounded by:

$$\Pr[X \leq (1 - \delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{2}\right).$$

6.2.2 Key Lemma

Lemma 6.2.2. *For any pair of vertices $s, t \in V$ with shortest path length $L = |\pi(s, t)| \geq cn^{1/2} \log n$ for some constant $c > 0$, with high probability, there exist vertices $u, v \in S$ such that:*

- *u lies on $\pi(s, t)$ within distance $\tilde{O}(n^{1/2})$ from s.*
- *v lies on $\pi(s, t)$ within distance $\tilde{O}(n^{1/2})$ from t.*

Proof. We partition the path $\pi(s, t)$ into three segments:

- **Prefix:** The first $\ell = c' n^{1/2} \log n$ vertices starting from s, for some constant $c' > 0$.
- **Middle:** The segment between the prefix and suffix.
- **Suffix:** The last $\ell = c' n^{1/2} \log n$ vertices ending at t.

Since $L \geq cn^{1/2} \log n$, the middle segment exists. Each vertex in S is selected uniformly at random from V. The probability that a vertex $v \in V$ is included in S is:

$$p = \frac{|S|}{n} = \frac{n^{1/2}}{n} = n^{-1/2}.$$

The expected number of sampled vertices in the prefix is:

$$\mu = \ell p = c' n^{1/2} \log n \cdot n^{-1/2} = c' \log n.$$

Using the Chernoff bound, the probability that S contains no vertex from the prefix is:

$$\Pr[\text{No vertex from prefix in } S] \leq e^{-\mu} = e^{-c' \log n} = n^{-c'}.$$

The same analysis applies to the suffix. Thus, with high probability, S contains at least one vertex u in the prefix and at least one vertex v in the suffix. Since u can reach v via the subpath of $\pi(s,t)$, a shortcut edge (u,v) is added to G'. Therefore, the distance from s to t in G' is $\tilde{O}(n^{1/2})$, and the diameter of G' is reduced to $\tilde{O}(n^{1/2})$ with high probability. ■

6.2.3 Optimality for Exact Hopsets

After discussing the folklore algorithm for constructing hopsets, it is somewhat surprising that recent work by Bodwin and Hoppenworth [26] has confirmed its near-optimality for *exact hopsets*. They demonstrate that for any graph G, any exact hopset with $O(n)$ edges must have a diameter of at least $\widetilde{\Omega}(n^{1/2})$, improving upon the previous lower bound of $\Omega(n^{1/3})$ by Kogan and Parter [27].

They prove this by constructing graphs on which any exact hopset of $O(n)$ edges has diameter $\widetilde{\Omega}(n^{1/2})$. This result confirms that folklore sampling reaches a fundamental $\sqrt{n}$-size barrier for exact hopsets.

Their work extends to provide lower bounds for exact hopsets and shortcut sets of size $O(p)$, demonstrating that folklore sampling is near-optimal across a wide range of parameters $p \in [1, n^2]$.

6.3 Parallel Reachability via Hopsets

In this chapter we will focus on the key contribution from [2] which addresses the single-source reachability problem in the PRAM model. This result can be summarized as follows:

Theorem 6.3.1 (Reachability in the PRAM Model)**.** *For any parameter k, there exists a parallel algorithm that, given an n-node m-edge digraph, solves the single-source reachability problem with work $\tilde{O}(mk + nk^2)$ and depth $poly(k) \cdot n^{1/2+O(1/\log k)}$ with high probability.*

First we'll present the sequential algorithm for diameter reduction which forms the basis for the improved parallel reachability algorithm.

Theorem 6.3.2 (Sequential Diameter Reduction)**.** *For any parameter k, there is an algorithm that given any n-node and m-edge digraph in $\widetilde{O}(mk)$ time computes $\widetilde{O}(nk)$ shortcuts such that adding these edges to the graph reduces to the diameter to $n^{1/2+O(1/\log k)}$ with high probability.*

Setting $k = O(\log n)$ immediately gives the following corollaries for the sequential and parallel case.

Corollary 6.3.1. *There is a parallel algorithm that given any n-node and m-edge digraph performs $\widetilde{O}(m)$ work in depth $n^{1/2+o(1)}$ and computes a set of $\widetilde{O}(n)$ shortcuts such that adding these edges to the graph reduces the diameter to $n^{1/2+o(1)}$ w.h.p.*

Corollary 6.3.2. *There is an algorithm that given any n-node and m-edge digraph in $\widetilde{O}(m)$ time computes $\widetilde{O}(n)$ shortcuts such that adding these edges to the graph reduces to the diameter to $n^{1/2+o(1)}$ with high probability.*

6.3.1 Preliminaries

We denote vertex set of a graph G by $V(G)$, and the edge set by $E(G)$. We simply write these as V and E when the graph G is clear from context. For a $V' \subseteq V$, we let $G[V']$ denote the induced subgraph on V', i.e. the graph with vertices V' and edges of G that have both endpoints in V'.

6.3.1.1 Digraph relations

Let G be a *directed graph* or *digraph* for short. We say that $u \preceq v$ if there is a directed path from u to v in G. In this case we say that u *can reach* v, or that v *is reachable from* u. We say that $u \not\preceq v$ if there is no directed path from u to v in G. In this case we say that u cannot reach v, or that v is not reachable from u. When $u \preceq v$ and $v \preceq u$ we say that u and v are in the same strongly connected component. We define the *descendants* of v to be $R^{\text{Des}}_G(v) \stackrel{\text{def}}{=} \{u \in V(G) : v \preceq u\}$ and the *ancestors* of v to be $R^{\text{Anc}}_G(v) \stackrel{\text{def}}{=} \{u \in V(G) : u \preceq v\}$. We say that u and v are *related* if $u \preceq v$ or $v \preceq u$. We define the *related vertices* of v as $R_G(v) \stackrel{\text{def}}{=} R^{\text{Des}}_G(v) \cup R^{\text{Anc}}_G(v)$. Throughout, the letter R we use in the notation should be read as "related" or "reachable". We say that u is *unrelated* to vertex v if $u \in V(G) \backslash R_G(v)$.

We extend this notation to subsets $V' \subseteq V$ in the natural way. We define the ancestors, descendants, and related vertices to V' as

$$R^{\text{Des}}_{G'}(V') \stackrel{\text{def}}{=} \bigcup_{v \in V'} R^{\text{Des}}_{G'}(v), \quad R^{\text{Anc}}_{G'}(V') \stackrel{\text{def}}{=} \bigcup_{v \in V'} R^{\text{Anc}}_{G'}(v),$$
$$R_{G'}(V') \stackrel{\text{def}}{=} R^{\text{Anc}}_{G'}(V') \cup R^{\text{Des}}_{G'}(V')$$

We say that a vertex v is related to a subset V' if $v \in R_G(V')$. When the graph G is clear from context, we will often drop the G subscript and simply write (for example) $R^{\text{Des}}(v), R^{\text{Anc}}(v)$, and $R(v)$ instead of $R_G^{\text{Des}}(v), R_G^{\text{Anc}}(v), R_G(v)$.

We further extend this notation to induced subgraphs of G. Let G' be a subgraph of G, possibly with a different set of vertices and edges than G. We say that $u \preceq_{G'} v$ if there is a directed path from u to v in the subgraph G'; we say that v is reachable from u *through* G' in this case. Define

$$R_{G'}^{\text{Des}}(v) \stackrel{\text{def}}{=} \{u \in V(G') : v \preceq_{G'} u\}, \quad R_{G'}^{\text{Anc}}(v) \stackrel{\text{def}}{=} \{u \in V(G') : u \preceq_{G'} v\},$$
$$R_{G'}(v) \stackrel{\text{def}}{=} R_{G'}^{\text{Des}}(v) \cup R_{G'}^{\text{Anc}}(v)$$

We similarly extend this definition to subsets of vertices $V' \subseteq V(G')$:

$$R_{G'}^{\text{Des}}(V') \stackrel{\text{def}}{=} \bigcup_{v \in V'} R_{G'}^{\text{Des}}(v), \quad R_{G'}^{\text{Anc}}(V') \stackrel{\text{def}}{=} \bigcup_{v \in V'} R_{G'}^{\text{Anc}}(v),$$
$$R_{G'}(V') \stackrel{\text{def}}{=} R_{G'}^{\text{Anc}}(V') \cup R_{G'}^{\text{Des}}(V')$$

As our algorithm performs recursion on subgraphs of G, this notation enables us to reference specific subproblems as our algorithm progresses.

A *shortcut* refers to adding an edge (u, v) to a graph G where $u \preceq v$ in G. Adding the edge does not affect the reachability structure of G. A *shortcutter* v is a node we add shortcut edges to and from. A *hopset* refers to a collection of shortcuts.

6.3.1.2 Paths

Our analysis will consider paths in the graph as well as the relations between the vertices on the path and other vertices in the graph. Let G be a digraph. We denote a path $P = \langle v_0, v_1, \ldots, v_\ell \rangle$, where all the v_i are vertices of G and $(v_i, v_{i+1}) \in E(G)$. Here, the length of the path is ℓ, where we have that $v_0 \preceq v_1 \preceq \ldots \preceq v_\ell$. We say that the head of the path is $\text{head}(P) \stackrel{\text{def}}{=} v_0$ and the tail is $\text{tail}(P) \stackrel{\text{def}}{=} v_\ell$. We now make the following definitions.

6.3.1.3 Path-related vertices

We adopt a similar convention as [25]. For a path $P = \langle v_0, v_1, \ldots, v_\ell \rangle$ we say that v is *path-related* if $v \in R_G(P)$. Further, for any path P in digraph G, we define $s(P, G) \stackrel{\text{def}}{=} |R_G(P)|$ as the number of path-related vertices. All path-related vertices are one of the following three types:

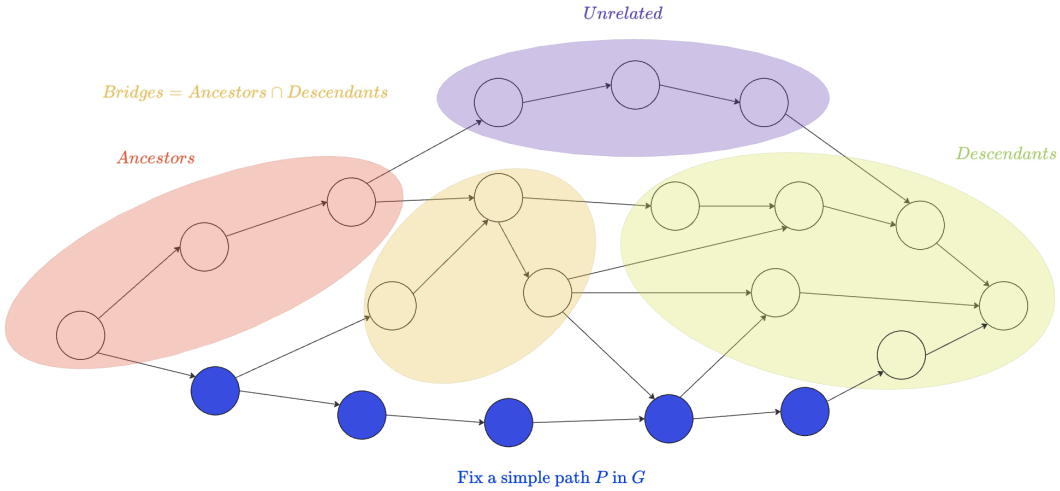


Figure 6.1: Path-related Vertices

- **Descendants:** We say that a vertex v is a descendant of the path P if $v \in R_G^{\text{Des}}(P) \backslash R_G^{\text{Anc}}(P)$. Note that this holds if and only if $v_0 \preceq v$ and $v \not\preceq v_\ell$.
- **Ancestors:** We say that a vertex v is an ancestor of the path P if $v \in R_G^{\text{Anc}}(P) \backslash R_G^{\text{Des}}(P)$. Note that this holds if and only if $v_0 \not\preceq v$ and $v \preceq v_\ell$.
- **Bridges:** We say that a vertex v is a bridge of the path P if $v \in R_G^{\text{Des}}(P) \cap R_G^{\text{Anc}}(P)$. Note that this holds if and only if $v_0 \preceq v$ and $v \preceq v_\ell$.

A vertex which is not a descendant, ancestor, or bridge for a path P is called *unrelated* to P. Later we explain how to extend all these definitions to the distance-limited case.

6.3.1.4 Subproblems

During our algorithms' recursions, we will make reference to the induced recursive calls made. Consider a graph G and a path P in G. During a call to an algorithm on the graph G, we define a *subproblem* to be an induced subgraph $G[V']$ along with a subpath P' of P which lies inside $G[V']$ on which we perform a recursive execution.

6.3.1.5 Miscellaneous

We let $B(n, p)$ be the binomial random variables over n events of probability p. We have the following standard fact about binomial random variables:

Lemma 6.3.3 (Chernoff Bound). *Let $X \sim B(n,p)$ be a binomial random variable. Then*

$$\Pr\left[X > (1+\delta)np\right] \leq \exp\left(-\frac{\delta^2}{2+\delta}np\right).$$

6.3.2 Fineman's Shortcutting Scheme

First, we provide a blueprint for the sequential nearly linear time algorithm for computing diameter reducing hopsets. For the remainder of this section, let $G = (V, E)$ be a digraph for which we wish to efficiently compute diameter-reducing shortcuts. For simplicity we consider the case where G is a directed acyclic graph (DAG); the analysis of the general case is essentially identical and for the purposes of reachability (ignoring parallel computation issues) we can contract every strongly connected component to a single vertex.

The shortcutting algorithm is inspired by the general framework used by Fineman [25] for efficiently computing diameter-reducing shortcuts. In Fineman's approach, at each iteration, a "shortcutter" vertex v is selected uniformly at random from the vertex set V. The algorithm then categorizes the vertices into three groups: v's ancestors $R_G^{\text{Anc}}(v)$, v's descendants $R_G^{\text{Des}}(v)$, and the vertices that are unrelated to v, denoted by $U_G(v) = V \setminus \{R_G^{\text{Des}}(v) \cup R_G^{\text{Anc}}(v)\}$. Once these groups are established, shortcut edges are added from v to all vertices in $R_G^{\text{Des}}(v)$ and from all vertices in $R_G^{\text{Anc}}(v)$ to v. After adding these shortcuts, the algorithm generates three induced subgraphs: $G_D = G[R_G^{\text{Des}}(v)]$, $G_A = G[R_G^{\text{Anc}}(v)]$, and $G_U = G[U_G(v)]$, and recursively applies the same procedure to each of these subgraphs.

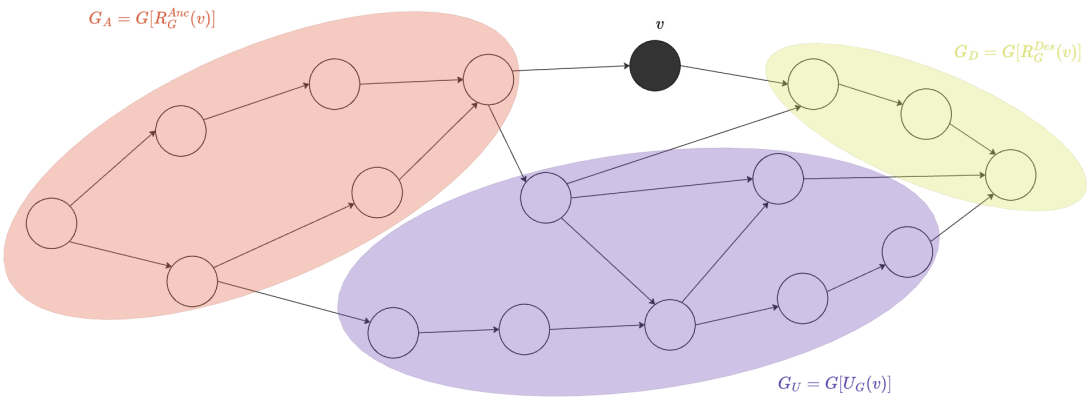


Figure 6.2: Fineman's shortcutting scheme

To analyze this procedure, we consider any path P in G. The work in [25] examines how the shortcuts introduced by the algorithm impact the distance between the endpoints

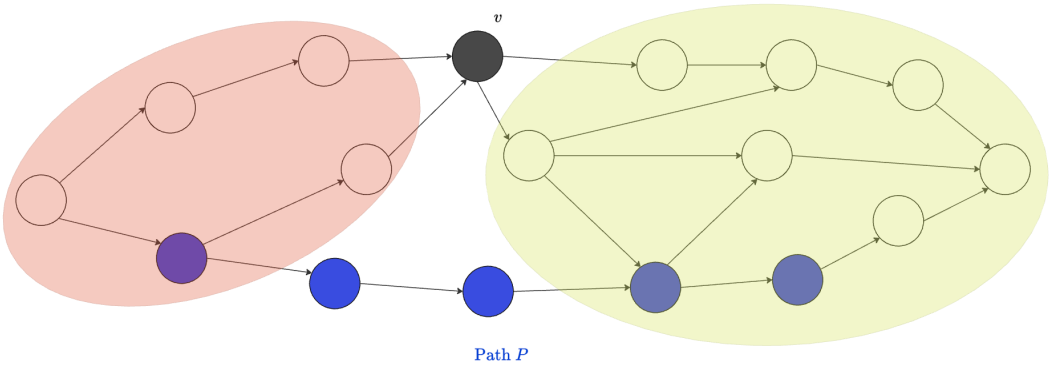


Figure 6.3: Interaction of shortcutter with path P

of P. When a "shortcutter" vertex v is chosen, its interaction with P can fall into one of the following four categories:

1. v is unrelated to any node in P.
2. v is an ancestor of some nodes in P, forming a subpath P_1, and is unrelated to the rest of the path P_2.
3. v is a descendant of some nodes in P, forming a subpath P_2, while being unrelated to the remaining subpath P_1.
4. v is both an ancestor of the tail and a descendant of the head of P.

When applying the recursive steps of the algorithm, one of three outcomes is possible after shortcutting through a vertex v: either the path P remains intact in a subproblem (case 1), the path splits into exactly two subpaths in two different subproblems (cases 2 or 3), or the connectivity between the endpoints of P is resolved via v, meaning the endpoints of P can be connected in just two edges through v (case 4). Thus, the path P is either divided into at most two subpaths or the distance between its endpoints is reduced to 2.

Let P_i represent the subpath of P at some point during the execution of the algorithm, and let V_i denote the vertex set of the subproblem that contains P_i. The key insight of Fineman is to define the following function (which we defined in Section 6.3.1) and to use it to reason about the effect of this random process:

$$s(P_i, G_i) \stackrel{\text{def}}{=} |\{u \in V(G_i) | u \text{ is related to some node in } P_i\}| .$$

Observe that $s(P_i, G_i)$ is an overestimate of the length of P_i, and that $s(P, G) \leq n$. Define

$$L(P) \stackrel{\text{def}}{=} \sum_i s(P_i, G_i)$$

to be the sum over all $s(P_i, G_i)$ at this state of our algorithm. The random variable $L(P)$ represents the path length, and by analyzing the four cases mentioned earlier, we can predict how $L(P)$ changes in expectation. For any subpath P_i, we look at the induced subproblems after shortcutting through a randomly selected node v. If v falls under case 1, nothing changes for P_i. In case 4, the connectivity between the endpoints of P_i is resolved, so we set $s(P_i, G_i) = 0$ as no further subproblem remains.

However, in cases 2 and 3, the path P_i is split into two subpaths, say P_{i_1} and P_{i_2}. Fineman argues that a randomly selected node ensures that the number of nodes related to either P_{i_1} or P_{i_2} decreases by a constant factor c in expectation. Thus, if $f(x)$ is the expected shortcut length of a path P_i where $s(P_i, G_i) = x$, we can use induction to show that:

$$f(x) \leq \max_{a+b=cx} f(a) + f(b)$$

Fineman derives a constant $c = \frac{3}{4}$, leading to a bound of $f(n) \leq O(n^{1/\log(8/3)})$. A more refined version of this argument enables him to achieve the tighter bound of $O(n^{2/3})$.

6.3.3 Sequential Reachability

The algorithm from [2] closely resembles Fineman's but introduces a key difference: instead of selecting just one "shortcutter" node before recursing, they choose an increasing number of random vertices in each step.

By selecting multiple shortcut vertices, they enhance the efficiency of the process. After shortcutting, they group the remaining vertices into subsets, similar to Fineman's partitioning into ancestors, descendants, and unrelated vertices. However, in their partitioning scheme, two vertices will belong to the same subset only if they share the same relationship with each of the k selected shortcutters.

We give some definitions and intuition for the quantities defined in the algorithm.

- **Inputs** k, r: k is a parameter governing the speed that we recurse at. Intuitively, the algorithm picks shortcutters so that graphs at one level deeper in the recursion are "smaller" by a factor of k. $r \leq \log_k n$ is the depth of recursion that the algorithm is currently at, where we start at $r = 0$.
- **Set** S: S is the set of vertices from which we search and build shortcuts from.

- **Set** F: F is the final set of shortcuts we construct.
- **Probability** p_r: At recursion depth r, for each vertex $v \in V(G)$, we put v in S with probability p_r which is roughly $\tilde{O}(k^r/n)$.
- **Labels** $v^{\text{Des}}, v^{\text{Anc}}, \text{✗}$: We want to distinguish vertices by their relations to vertices in S. Therefore, when we search from a vertex v we add a label v^{Des} to add vertices in $R_G^{\text{Des}}(v) \backslash R_G^{\text{Anc}}(v)$, a label v^{Anc} to all vertices in $R_G^{\text{Anc}}(v) \backslash R_G^{\text{Des}}(v)$, and a label ✗ to all vertices in $R_G^{\text{Des}}(v) \cap R_G^{\text{Anc}}(v)$. The label ✗ should be understood as "eliminating" the vertex (since it is in the same strongly connected component as v and we have shortcut through v already).

Algorithm 11: SEQ(G, k, r): Sequential Diameter Reduction Algorithm

Input: Graph G, parameter k, recursion depth $r \leq \log_k n$ (starting with $r = 0$)
Output: A set of shortcut edges to add to G

n represents the number of vertices at the top level of recursion
begin
  $p_r \leftarrow \min\left\{1, \frac{20k^{r+1}\log n}{n}\right\}$;
  $S \leftarrow \emptyset$;
  foreach $v \in V$ **do**
    Add v to S with probability p_r;
  end
  $F \leftarrow \emptyset$;
  foreach $v \in S$ **do**
    foreach $w \in R_G^{\text{Des}}(v)$ **do**
      Add edge (v, w) to F;
    end
    foreach $w \in R_G^{\text{Anc}}(v)$ **do**
      Add edge (w, v) to F;
    end
    foreach $w \in R_G^{\text{Des}}(v) \setminus R_G^{\text{Anc}}(v)$ **do**
      Add label v^{Des} to vertex w;
    end
    foreach $w \in R_G^{\text{Anc}}(v) \setminus R_G^{\text{Des}}(v)$ **do**
      Add label v^{Anc} to vertex w;
    end
    foreach $w \in R_G^{\text{Des}}(v) \cap R_G^{\text{Anc}}(v)$ **do**
      Add label ✗ to vertex w;
    end
  end
  $W \leftarrow \{v \in V \mid v$ has no label of ✗$\}$;
  Partition W into subsets $V_1, V_2, \ldots, V_\ell$ such that vertices x and y belong to the same subset if and only if they share exactly the same labels;
  for $1 \leq i \leq \ell$ **do**
    $F \leftarrow F \cup \text{SEQ}(G[V_i], k, r + 1)$;
  end
  return F;
end

For example, two vertices u_1 and u_2 would not be placed in the same subset if u_1 is an ancestor of a shortcutter v and u_2 is a descendant of that same shortcutter v. By selecting k shortcut nodes from scenarios 2, 3, or 4 and partitioning as described, we ensure

that the number of path-related nodes decreases by a factor of $\frac{2}{k+1}$ in expectation after each recursion.

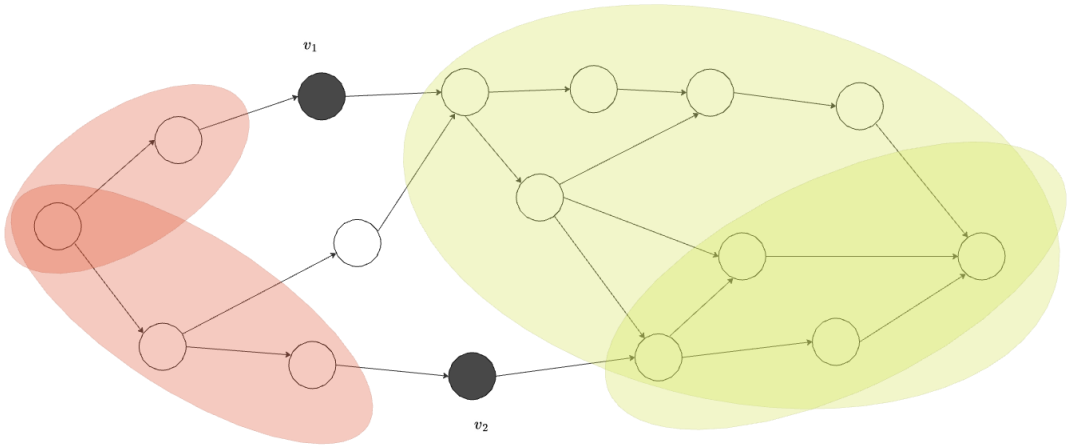


Figure 6.4: Multiple Shortcutters

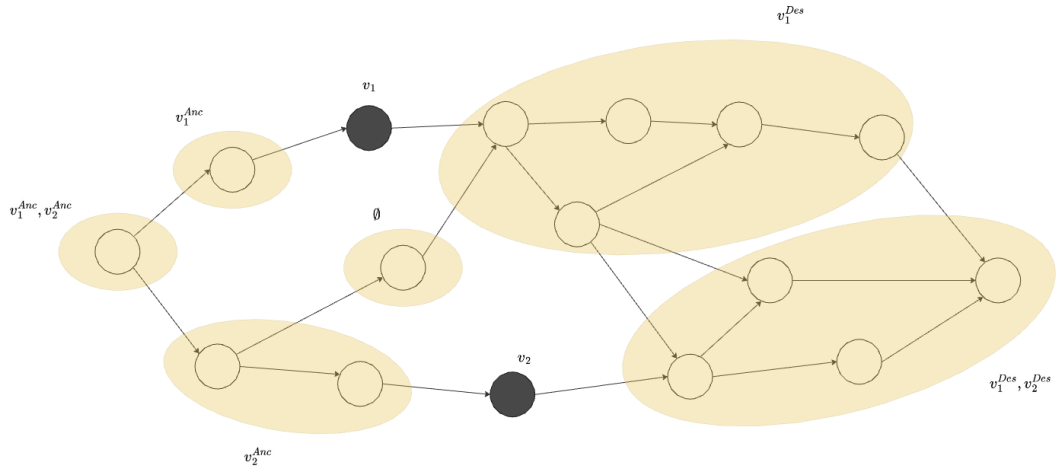


Figure 6.5: Partitioning Scheme According to Labels

The goal is to select as many shortcutter vertices as possible while maintaining the nearly-linear work bound. The more shortcutters that are selected, the greater the likelihood of achieving the desired $\frac{2}{k+1}$ reduction in path-related nodes. However, selecting the same number of shortcutters at every level of recursion is not feasible because the number of path-related nodes decreases rapidly. If k shortcutters are chosen per level of recursion, eventually only a single path-relevant shortcutter can be picked in each round.

Instead, it is shown that after each level of recursion, the structure of the subproblems allows for the selection of k *times more* shortcutters while still adhering to the nearly-linear work requirement.

Thus, in the first iteration we add $\widetilde{O}(k)$ shortcutters w.h.p. and perform $\widetilde{O}(mk)$ work. We then partition the nodes into clusters such that any two nodes x and y which are in the same V_i have exactly the same labels assigned to them by the shortcutters, none of which are ✗. We then recursively apply the algorithm within each cluster with a sampling probability that is a factor of k larger.

6.3.4 Work and Shortcut Bound

In this section, we provide bounds on the work and the number of shortcut edges introduced by Algorithm 11. At any recursion level of the algorithm, the total work comes from two main tasks. The first task involves computing the necessary labels v^{Des}, v^{Anc}, and ✗ for each node v from which we generate shortcuts. The second task is grouping the nodes based on these labels to form the subproblems for the next recursion level. We will bound both of these tasks by leveraging a useful observation about the number of ancestors and descendants each node has within its subproblem at any level.

Lemma 6.3.4. *Consider an execution of* $\textsc{Seq}(G, k, 0)$ *on* n*-node* m*-edge* G*. With probability* $1 - n^{-10}$ *in each recursive execution of* $\textsc{Seq}(G', k, r)$ *of Algorithm 11 the following holds*

$$R^{\text{Des}}_{G'}(v) \leq nk^{-r} \quad \text{and} \quad R^{\text{Anc}}_{G'}(v) \leq nk^{-r} \quad \text{for all } v \in G'.$$

Proof. We prove the statement by induction on r. The claim is trivially true for the base case where there is a single recursive call at $r = 0$. Now, assuming that the claim holds for all recursive calls at $r = j$, we will show that it also holds for calls at $r = j+1$ with probability at least $1 - n^{-11}$. Applying the union bound across all $\widetilde{O}(1)$ values of r encountered in the algorithm will then imply the final result.

Assume that the claim holds for all recursive executions at $r = j$. Let $v \in V$ be any vertex, and let G' be the induced subgraph on which the algorithm is called recursively at $r = j + 1$ and contains v. We will first prove the claim for $R^{\text{Des}}_{G'}(v)$, as the argument for $R^{\text{Anc}}_{G'}(v)$ follows symmetrically.

Note that the recursive call $\textsc{Seq}(G', k, j+1)$ is called during the execution of $\textsc{Seq}(H, k, j)$ for some subgraph $H \subseteq G$. Let Q be the set of vertices in G' that are descendants of v in H. If $|Q| = R^{\text{Des}}_{H}(v))$ is less than nk^{-r}, the claim holds, as recursive calls only reduce the size of the subgraphs. Hence, we now assume that $|Q| \geq nk^{-r}$.

Let $Q_1, Q_2, \ldots$ be the strongly connected components (SCCs) of Q, and consider any topological ordering of these SCCs. In this ordering, Q_i precedes Q_j whenever there is

a path from Q_i to Q_j. Now, consider any two vertices $x, y \in Q$ where y precedes x in this order. We analyze the random choices made during the execution of $\textsc{Seq}(H, k, j)$ that determine the formation of G'.

If y is chosen as a shortcutter for H, then x cannot be in G', because v is in G' and x and v would have received different labels from y. Specifically, v is an ancestor of y, whereas x is either a descendant of or unrelated to y. Additionally, if a shortcut is added from y, any vertex z in the same SCC as y would also be excluded from G' since z would receive an ✗ label.

Therefore, if we select any of the first nk^{-j} nodes in the topological ordering of Q (i.e., those closest to v in H) as shortcutters, we can ensure that at the G' level, v will have at most nk^{-j} descendants. Since each vertex is chosen with probability $\frac{20k^j \log n}{n}$, the probability of failing to do this is at most:

$$\left(1 - \frac{20k^j \log n}{n}\right)^{nk^{-j}} \leq e^{-20 \log n} = n^{-20}.$$

By applying the union bound over all vertices in G' and over all induced subgraphs encountered at level $r = j$, we can conclude that the bound holds for all recursive calls at $r = j$ with probability at least $1 - n^{-11}$. Thus, the result follows. ■

We now bound the number of labels any vertex v receives in any recursive execution which contains it. This will provide us with an elegant way to bound the total work of our procedure.

Lemma 6.3.5. *Consider an execution of* $\textsc{Seq}(G, k, 0)$ *on n-node m-edge G with $k \geq 2$. With probability $1 - 2n^{-10}$, every recursive execution* $\textsc{Seq}(G[V_i], k, r)$ *assigns at most $80k \log n$ labels to every node $w \in V_i$ where ✗ labels assigned by different shortcutters are counted as distinct labels.*

Proof. A node v receives a label from a shortcutter u only if u is related to v. By Lemma 6.3.4, we know that at most $2nk^{-r}$ nodes can be related to v in any execution of Seq, with probability at least $1 - n^{-10}$. Since nodes at the r^{th} recursion level are selected as shortcutters with probability $p_r = \frac{20k^{r+1} \log n}{n}$, the probability that more than $80k \log n$ labels are assigned to v, assuming at most $2nk^{-r}$ nodes are related to it, is at most:

$$\Pr\left[B\left(2nk^{-r}, \frac{20k^{r+1} \log n}{n}\right) > 80k \log n\right] \leq \exp\left(-\frac{40}{3} k \log n\right) \leq n^{-12}$$

by applying Lemma 6.3.3. Therefore, the probability that v receives more than $80k \log n$ labels is at most $1 - n^{-12}$. Applying a union bound over all nodes and all recursive executions of Seq gives the desired result. ■

Finally, we conclude this subsection by bounding the total work of SEQ, as well as the number of shortcut edges it adds.

Lemma 6.3.6. *Consider an execution of* $\textsc{Seq}(G, k, 0)$ *on* n*-node* m*-edge* G *with* $k \geq 2$*. With probability* $1 - 2n^{-10}$ $\textsc{Seq}(G, k, 0)$ *runs in* $\widetilde{O}(mk)$ *time and adds* $\widetilde{O}(nk)$ *shortcuts.*

Proof. By our given probability of failure, we may assume Lemma 6.3.4 and Lemma 6.3.5 hold deterministically.

We begin by considering a single recursive execution $\textsc{Seq}(G[V_i], k, r)$ generated by SEQ. We will bound the number of shortcuts added by this call and amount of work it performs before it recurses. We will then aggregate these bounds over all recursive executions and obtain our final result. For convenience, let $G[V_i]$ have $\hat{n}$ nodes and $\hat{m}$ edges.

First, we bound the number of shortcut edges added by $\textsc{Seq}(G[V_i], k, r)$. According to Lemma 6.3.5, each recursive call assigns $\widetilde{O}(k)$ labels to every vertex $w \in V_i$. Since each label corresponds to a shortcut edge being added, $\textsc{Seq}(G[V_i], k, r)$ adds $\widetilde{O}(k)$ edges for each vertex w, resulting in a total of $\widetilde{O}(\hat{n}k)$ edges.

We now bound the work performed by $\textsc{Seq}(G[V_i], k, r)$. Within a call to SEQ, we perform work in two places: within the loop in line 9 and when generating the partition in line 27. We bound the contributions of these sources in order. First, observe that the loop in line 9 can be implemented by computing breadth-first searches forwards and backwards from every w in the shortcutter set S. The amount of work needed to apply the labels and and the shortcuts themselves is clearly $\widetilde{O}(\hat{n}k)$ by the above argument, so we need only to bound the cost of running these traversals.

Observe that by Lemma 6.3.5 $\textsc{Seq}(G[V_i], k, r)$ assigns $\widetilde{O}(k)$ labels to every $w \in V_i$. Now the number of labels w receives is within a factor of two the number of times it is visited in searches. Thus w is visited $\widetilde{O}(k)$ times in our traversals. Each time we encounter w in a traversal we perform a constant amount of work for each edge incident upon it. Thus if $\delta_i(w)$ is the undirected degree of w in $G[V_i]$, the total work performed by $\textsc{Seq}(G[V_i], k, r)$ is

$$\widetilde{O}\left(\sum_{w \in V_i} k\delta_i(w)\right) = \widetilde{O}(\hat{m}k).$$

We finally bound the cost of generating the partition. We implement this in two parts. First, we check each vertex to see whether it has an ✗ label and discard any vertex which does. Next, we define an order over all possible combinations of labelings a node could receive. We then sort the remaining nodes by this order: we can then trivially read off the partition. To implement this order of labelings, pick an arbitrary ordering on the individual labels we distribute to nodes. To compare two labeling schemes a

and b we internally sort a and b by our arbitrary ordering, and then determine the order amongst a and b lexicographically.

To implement this procedure, we first note that by Lemma 6.3.5 every node receives at most $\widetilde{O}(k)$ labels. Determining which of the $\hat{n}$ nodes have an ✗ label clearly takes $O(\hat{n}k)$ time. It is straightforward to verify that comparing two labelings each with at most $\widetilde{O}(k)$ labels with this scheme requires $\widetilde{O}(k)$ time: thus this partitioning can be found in $\widetilde{O}(\hat{n}k)$ time using a mergesort. Combining this with the previous bound we see that $\textsc{Seq}(G[V_i], k, r)$ requires $\widetilde{O}(\hat{m}k)$ time before recursing.

We now obtain our final work and shortcut bounds by aggregating. If we consider the set of recursive calls $\textsc{Seq}(H, k, r)$ for any fixed value of r, we see that the calls are applied to a disjoint collection of subgraphs of G. Thus, the total number of nodes in all of these subproblems is n, and the total number of edges is at most m. Thus cost of performing all of these calls without recursing is $\widetilde{O}(mk)$, and these calls collectively add $\widetilde{O}(nk)$ shortcuts. As there are at most $\widetilde{O}(1)$ different values of r our claim follows. ■

6.3.5 Parallel Reachability

The method is parallelized similarly to Fineman's approach. The core concept behind Fineman's parallelization involves the use of *depth-restricted* searches, referred to as D^{search}-restricted searches. Instead of performing full graph traversals to compute the ancestor, descendant, and unrelated sets for a vertex v, Fineman computes only the D^{search}-ancestors and D^{search}-descendants—nodes reachable within a distance of D^{search} from v. Although these searches can be executed at low depth, relying solely on them as a substitute for full ancestor and descendant sets no longer guarantees the expected reduction in $L(P)$.

Fineman resolves this issue through a novel approach. Consider a directed graph G. Assume that a set of edges F can be efficiently identified and added to G, such that if two nodes s and t are at distance D, their distance in the modified graph $G \cup F$ is reduced to at most $D/5$ with high probability. For any pair of nodes u and v separated by a distance greater than D, the distance between them in $G \cup F$ is halved with high probability. This process breaks the shortest path between u and v into segments of length D, with each segment's size decreasing by a constant factor with constant probability. Repeating this process on $G \cup F$ and iterating it $O(\log n)$ times ensures that every reachable pair of nodes u and v are within distance D. This reduction incurs only logarithmic costs in terms of total work and parallel depth. Building off of these ideas with some modifications for the new algorithm we can leverage the result for the parallel version Theorem 6.3.1.

The final parallel algorithm we are going to use:

Algorithm 12: ParallelSC($G, k, r, r_{\text{fringe}}$). Takes a digraph G, parameter k, recursion depth $r \leq \log_k n$ (starts at $r = 0$), and inner fringe node recursion depth $r_{\text{fringe}} \leq \log n$. Returns a set of shortcut edges to add to G. n denotes the number of vertices at the top level of recursion, not the number of vertices in G.

Input: $G, k, r, r_{\text{fringe}}$
Output: A set of shortcut edges to add to G

$p_r \leftarrow \min\left(1, \frac{10k^{r+1}\log n}{n}\right)$
$S \leftarrow \emptyset$
$\kappa_{2r+1} \leftarrow 10^6 k^2 \log^5 n \left(1 + \frac{1}{4\log n}\right)^{-2r-1}$ and $\kappa_{2r} \leftarrow 10^6 k^2 \log^5 n \left(1 + \frac{1}{4\log n}\right)^{-2r}$
Choose $\kappa \in [\kappa_{2r+1}, \kappa_{2r}]$ uniformly at random. `// Picking a random search distance`
$D \leftarrow 100 \cdot \sqrt{2}^{\log_k n} \cdot n^{\frac{1}{2}} \cdot \log^2 n$
for $v \in V$ **do**
  With probability p_r, do $S \leftarrow S \cup \{v\}$
end
$F \leftarrow \emptyset$
for $v \in S$ **do**
  for $w \in R_{\text{Des}}(v, (\kappa+1)D)$ **do**
    add edge (v, w) to F
  end
  for $w \in R_{\text{Anc}}(v, (\kappa+1)D)$ **do**
    add edge (w, v) to F
  end
  for $w \in R_{\text{Des}}(v, \kappa D) \setminus R_{\text{Anc}}(v, \kappa D)$ **do**
    add label v^{Anc} to vertex w
  end
  for $w \in R_{\text{Anc}}(v, \kappa D) \setminus R_{\text{Des}}(v, \kappa D)$ **do**
    add label v^{Des} to vertex w
  end
  for $w \in R_{\text{Des}}(v, \kappa D) \cap R_{\text{Anc}}(v, \kappa D)$ **do**
    add label ✗ to vertex w
  end
  $V_u^{\text{ring}} \leftarrow R(v, (\kappa+1)D) \setminus R(v, (\kappa-1)D)$
  $F \leftarrow F \cup \text{ParallelSC}(G[V_v^{\text{ring}}], k, r, r^{\text{fringe}} + 1)$. `// Recursion on fringe nodes`
end
$W \leftarrow \{v \in V \mid v \text{ has no label of } ✗\}$
Partition W into subsets $V_1, V_2, \ldots, V_\ell$ such that $x, y \in V_i$ iff x and y have exactly the same labels. `// Vertices in the` V_i `have no label of ✗.`
for $1 \leq i \leq \ell$ **do**
  $F \leftarrow F \cup \text{ParallelSC}(G[V_i], k, r+1, 0)$
end
return F

Algorithm 13: PARALLELDIAM(G, k). Takes a digraph $G = (V, E)$, parameter k. Modifies digraph G. Parallelizable diameter reduction algorithm.

Input: G, k
Output: Modified digraph G with reduced diameter

```
for i ← 1 to 10 log n do
    for j ← 1 to 10 log n do
        S_j ← ParallelSC(G, k, 0, 0)
    end
    E(G) ← E(G) ∪ (⋃_j S_j)
end
```

We provide a detailed explanation of what each part of Algorithm 12 and Algorithm 13 does skipping some of the technical details. For the complete analysis, we refer the reader to [2].

Algorithm 12 is similar to Algorithm 11. All parts of Algorithm 11 can be implemented in low parallel depth except for the breadth first searches from the vertices in S. To resolve this, a natural idea is to limit the distance of the breadth first searches to D, where D denotes the diameter bound. Running these incomplete searches introduces issues in the analysis though. To get around this, we follow an approach similar to [25] and perform some additional computation on the fringe of our breadth first searches. Specifically, we choose a random integer κ in the range $[\kappa_{2r+1}, \kappa_{2r}]$ (think of these as parameters which are $\text{poly}(\log n, k)$), and search from a vertex v to distance approximately κD. Then, we call the vertices in the set $R(v, (\kappa+1)D) \backslash R(v, (\kappa-1)D)$ the *fringe vertices*. We chose κ randomly to ensure that the expected number of fringe vertices is sufficiently small. We then recurse on the fringe vertices.

In the algorithm, we first initialize the set of vertices from which we will perform our search, denoted as S, starting with an empty set. We then choose a probability p_r, which determines whether each vertex v belongs to S. Following that, we select our search scale D, a value chosen to be a constant factor larger than the bounds previously discussed in relation to the graph's diameter. At this stage, we define the parameters $\kappa \in [\kappa_{2r+1}, \kappa_{2r}]$, which will set our search radius to κD. The parameters $\kappa_0 \geq \kappa_1 \geq \kappa_2 \geq \cdots$ ensure that as the recursion progresses, the search radius decreases with each level, with the guarantee that $\kappa_i \geq \frac{1}{2}\kappa_0$ for all i when $r \leq \log_k n$.

After setting the search radius, we move on to choosing the specific vertices for the breadth-first search, which are added to S. Once the set S is established, we initialize the set of shortcut edges, starting with an empty set. From here, we proceed by adding shortcut edges for each vertex $v \in S$, connecting them to the relevant vertices based on the current search parameters.

Next, we label other vertices based on their relationships with v, applying ancestor, de-

scendant, and "eliminated" labels. Once the labels are assigned, we initiate a recursion for the fringe vertices identified during the search. At this point, the recursion depth for the fringe nodes, r_{fringe}, is incremented by one while the main recursion depth, r, remains the same.

With the labels applied, we move on to processing the vertices. All vertices that have been marked with the "eliminated" label, denoted by ✗, are excluded from further consideration. The remaining vertices are partitioned into groups based on matching labels, ensuring that vertices with the same set of labels are placed together. Finally, we perform a recursive step on each of the groups created, increasing the main recursion depth r by one, while resetting the fringe recursion depth r_{fringe} back to zero.

By modifying this algorithm slightly, we can also retrieve the paths discovered during the depth-limited breadth-first searches from vertices in S. In the graph, we have BFS trees that are of diameter $\sqrt{n}$ at most. This means the total depth of the path retrieval is $O(\sqrt{n})$. We can concat the paths from the BFS trees to get the paths in parallel in $\tilde{O}(\sqrt{n})$ work and $O(\sqrt{n})$ depth, while still performing the main task of diameter reduction. We will use this in the next section.

For Algorithm 13, we are essentially running Algorithm 12 for multiple iterations. Specifically, we can ensure that the expected distance from the head to the tail of any path of original length at most D is reduced to $\frac{D}{10}$. Running this process $O(\log n)$ times guarantees that any path of length D is reduced to $\frac{D}{5}$ with high probability. By repeating this procedure $O(\log n)$ times, we can see that the distance from the head to the tail of any path will be reduced to at most D with high probability. This is done by dividing the path into several subpaths of length at most D and applying the reduction iteratively, so that each subpath's length is reduced by a constant factor, ultimately reducing the length of the original path.

Chapter 7

Parallel Vertex Cut

7.1 Introduction

In this chapter, we present an algorithm for parallel vertex cut by combining ideas from the two approaches we presented in the previous chapters: the vertex cut algorithm of Nanongkai et al. [1] and the parallel reachability algorithm of Sidford et al [2]. Just to reiterate the results from the previous sections:

Theorem 7.1.1 (Reachability in the PRAM Model)**.** *There exists a parallel algorithm that, given an n-node m-edge digraph, solves the single-source reachability problem with work $\tilde{O}(m)$ and depth $n^{1/2+o(1)}$ with high probability in n.*

Theorem 7.1.2 (Vertex Connectivity)**.** *There exists a randomized (Monte Carlo) alorithm that given a directed graph $G = (V, E)$ and $k = O(\sqrt{n})$, if G is not k-connected then w.h.p finds a separator S with size less than k. If G is k-connected then the algorithm always returns $\perp$. The algorithm takes $\tilde{O}(mk^2)$ time.*

Theorem 7.1.3 (Vertex Connectivity with Vertex Sampling)**.** *There exists a randomized (Monte Carlo) alorithm that given a directed graph $G = (V, E)$ and $k = O(\sqrt{n})$, if G is not k-connected then w.h.p finds a separator S with size less than k. If G is k-connected then the algorithm always returns $\perp$. The algorithm takes $\tilde{O}(n^2k^2)$ time.*

Our results can be summarized in the two following theoremes:

Theorem 7.1.4 (Parallel Vertex Cut)**.** *There exists a parallel randomized (Monte Carlo) alorithm that given a directed graph $G = (V, E)$, $k = O(\sqrt{n})$ and $\alpha \in [\sqrt{n}, n]$, if G is not k-connected then w.h.p finds a separator S with size less than k. If G is k-connected then the algorithm always returns $\perp$. The algorithm takes $\tilde{O}(mk^2 + \frac{m^2}{\alpha})$ work and $\max\{n^{1/2+o(1)}, k\alpha\}$ depth.*

Theorem 7.1.5 (Parallel Vertex Cut with constant k)**.** *There exists a parallel randomized (Monte Carlo) alorithm that given a directed graph $G = (V, E)$, $k = O(1)$ and $\alpha \in [\sqrt{n}, n]$, if G is not k-connected then w.h.p finds a separator S with size less than k. If G is k-connected then the algorithm always returns $\perp$. The algorithm takes $\tilde{O}(\frac{mn}{\alpha})$ work and $O(\alpha)$ depth.*

7.2 The Algorithm

Suppose that graph G is not k-connected. This means there exists a separation triplet (L, S, R) with $|S| < k$. The algorithm diferentiates two cases according to the relative sizes of L and R. In each case we employ a subroutine to identify a small vertex cut if such exists. If no cut is found we conclude that *w.h.p.* there is none.

- The first phase of the algorithm deals with the case where L and R are roughly the same size. In this case we sample enough edges to ensure a high probability of landing one vertex in L and one in R. Then we run a modified version of Ford-Fulkerson with each vertex pair as source and sink to detect the small vertex cut S.
- The second phase deals with the case where L is small. Again we sample enough edges such that *w.h.p.* we get a vertex inside the small partition. Then we try to identify a small cut "near" this vertex through local exploration.

7.2.1 Modified Ford Fulkerson

First we present the modified version of Ford-Fulkerson. The original Ford-Fulkerson algorithm uses BFS to find augmenting paths and as we showed in Section 3.3.2. BFS when executed in parallel admits depth equal to the diameter of the graph D, which in the worst case could be n.

To lower the depth of this part of the algorithm we employ the parallel reachability algorithm from Sidford et al. We run this algorithm, placing the source vertex s in the set of shortcutters. From this algorithm, we get a set of shortcut edges and a set of paths from our source vertex s to all the reachable vertices R_x. Thus, getting the augmenting path is possible in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ depth. The total complexity for the modified Ford-Fulkerson is $\tilde{O}(mk)$ work and $n^{1/2+o(1)}$ depth.

Algorithm 14: Parallel Ford–Fulkerson for Checking Min Cut at Least k

Input: A network $G = (V, E)$ with flow capacity c, a source node s, a sink node t, and an integer k
Output: Returns true if the min cut is at least k, `false` otherwise
Function `FordFulkersonReachCheck(`*Graph G, Source s, Sink t, Integer k*`):`
    Initialize $f(u, v) \leftarrow 0$ for all edges $(u, v) \in E$;
    for $i \leftarrow 1$ **to** k **do**
        $R_s, P_s \leftarrow$ `ParallelDiam(`$G_f, \log n, s$`)` `// find augmenting path from` s `to` t `in residual graph` G_f
        if P_{st} does not exist **then**
            return `false` `// Min cut is less than` k
        end
        Find $c_f(p) = \min\{c_f(u, v) \mid (u, v) \in p\}$;
        foreach edge $(u, v) \in p$ **do**
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
            $f(v, u) \leftarrow f(v, u) - c_f(p)$
        end
    end
    return true `// Min cut is at least` k

7.2.2 Modified Local Vertex Cut

For the unbalanced case where $vol^*(L) < m/k$, we have to make some adjustments. Again we will make use of the reduction to directed edge connectivity and try to bound the size of $vol^*(L)$ between two powers of two to efficiently sample edges so that *w.h.p.* we get a vertex x in L.

After that, the method used by Nanongkai et al. uses a DFS traversal to explore edges "near" this sampled vertex x. The problem with DFS is that, like BFS, admits depth equal to the diameter of the graph D which could be equal to n.

Thus, we introduce a tradeoff parameter α such that if the size of $vol^*(L)$ is smaller than α then we opt to use the same technique as in the sequential case, running DFS to explore nodes near our source vertex x, sample an edge (y', y) from this DFS tree and reverse the path P_{xy} attempting to decrease the $\delta_{out}(L')$ by one each time.

If on the other hand the size of the partition is larger than α then we replace the DFS run with the parallel reachability algorithm we presented in the previous chapter. Again, we sample one of the reachable vertices and reverse the path. If the number of explored nodes is less than the size of L we guessed then we return $N_G^{out}(R_x)$ as the cut. In this case we keep the depth at $n^{1/2+o(1)}$ but we spend linear work per sample, thus increasing the total work of the algorithm.

Algorithm 15: LocalVCReach(Sample, DFS, ParallelDiam)

Input: G, x, ν, k
Output: A set L' containing x with $|N(L')| < k$ and $vol(L') < \nu$ if it exists, else $\perp$

```
for j ← 0 to log(n) do
    for i ← 1 to k do
        R_x, P_x ← ParallelDiam(G, log n, x)
        if |R_x| < kν then
            return N_G^out(R_x)
        end
        Sample a random edge e = (y', y) from R_x
        Reverse the path P_xy
    end
end
return ⊥
```

The work for LocalVCReach is $\tilde{O}(mk)$ meaning we spend linear time per sample instead of $\tilde{O}(\nu k^2)$ for LocalVC. The depth on the other hand is $n^{1/2+o(1)}$ for LocalVCReach in contrast with LocalVC where it's $k\nu$ which could be linear for big values of ν. Now we present the whole algorithm:

Algorithm 16: Parallel Vetrex Cut(Sample, LocalVC, FordFulkerson-ReachCheck, BFS)

Input: G, k
Output: A vertex cut S with $|S| < k$ if it exists, else $\perp$

$n \leftarrow |V|$
$m \leftarrow |E|$
for $i \leftarrow 1$ **to** $k \log n$ **do**
  Sample a random pair of edges $e = (x, x'), f = (y, y')$
  for $(s, t) \leftarrow \{(x, y), (x', y'), (x', y), (x, y')\}$ **do**
    if FordFulkersonReachCheck(G, s, t, k) **then**
      $T_{BFS} \leftarrow \text{BFS}(G_f, s)$
      return $N_G^{out}(T_{BFS})$
    end
  end
end
for $i \leftarrow 1$ **to** $\log(m/k)$ **do**
  if $2^i < \alpha$ **then**
    for $j \leftarrow 1$ **to** $m \log n / 2^i$ **do**
      Sample a random edge $e = (x, y)$
      for $v \in \{x, y\}$ **do**
        $L' \leftarrow \text{LocalVC}(G, v, 2^i, k)$
        if $L' \neq \perp$ **then**
          return $N_G^{out}(L')$
        end
      end
    end
  end
  else
    for $j \leftarrow 1$ **to** $m \log n / 2^i$ **do**
      Sample a random edge $e = (x, y)$
      for $v \in \{x, y\}$ **do**
        $L' \leftarrow \text{LocalVCReach}(G, v, 2^i, k)$
        if $L' \neq \perp$ **then**
          return $N_G^{out}(L')$
        end
      end
    end
  end
end
return $\perp$

7.2.3 Analysis

We analyze the two cases of the algorithm separately:

7.2.3.1 Balanced Case

Since *ParallelDiam* runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ depth, our parallel Ford-Fulkerson implementation checking if the min-cut between two vertices is less than k runs in $\tilde{O}(mk)$ work and $n^{1/2+o(1)}$ depth.

Thus the first part of our algorithm runs in $W_1 = \tilde{O}(mk^2)$ work and $D_1 = n^{1/2+o(1)}$ depth.

7.2.3.2 Unbalanced Case

When the two components are unbalanced, again we have two separate cases for when the size of L is smaller or larger than our tradeoff paramaeter α. For the first case we use the same LocalVC algorithm we used for the sequential algorithm, while for the latter case we use LocalVCReach as we discussed in the previous chapter.

Starting with LocalVC. For the work:

$$W_{21} = \sum_{i=1}^{\log \alpha} 2^i k^2 \frac{m}{2^i} \log n = \sum_{i=1}^{\log \alpha} mk^2 \log n = \tilde{O}(mk^2)$$

and for the depth:

$$D_{21} = \max_{0 \le i \le \log \alpha} \left\{ 2^i k \right\} = \alpha k$$

Respectively for LocalVCReach we have:

$$W_{22} = \sum_{\log \alpha}^{\log m/k} \tilde{O}(m) \frac{m}{2^i} \log n = \tilde{O}(m^2) \sum_{i=\log \alpha}^{\log m/k} \frac{1}{2^i}$$

For the sum computation:

$$S = \sum_{i=\log \alpha}^{\log m/k} \frac{1}{2^i}$$

Recognize that:

$$\sum_{i=a}^{b} \frac{1}{2^i} = \sum_{i=a}^{\infty} \frac{1}{2^i} - \sum_{i=b+1}^{\infty} \frac{1}{2^i}$$

Compute the infinite geometric series:

$$\sum_{i=a}^{\infty} \frac{1}{2^i} = \frac{1}{2^a} \cdot \frac{1}{1 - \frac{1}{2}} = \frac{1}{2^a} \cdot 2 = \frac{2}{2^a}$$

$$\sum_{i=b+1}^{\infty} \frac{1}{2^i} = \frac{1}{2^{b+1}} \cdot 2 = \frac{2}{2^{b+1}}$$

Calculating S:

$$\begin{aligned} S &= \left(\frac{2}{2^{\log \alpha}}\right) - \left(\frac{2}{2^{\log m/k+1}}\right) \\ &= \frac{2}{\alpha} - \frac{2}{2 \cdot \frac{m}{k}} = \frac{2}{\alpha} - \frac{k}{m} \end{aligned}$$

Calculating $m^2 S$:

$$m^2 S = m^2 \left(\frac{2}{\alpha} - \frac{k}{m}\right) = \frac{2m^2}{\alpha} - mk$$

Since $k = O(\sqrt{n})$:

$$W_{22} = \tilde{O}(m^2) S = \tilde{O}\left(\frac{m^2}{\alpha}\right)$$

For the depth:

$$D_{22} = n^{1/2 + o(1)}$$

7.2.3.3 Total

As we discussed in Chapter 3, the combined work and depth for the two steps is

$$W = W_1 + W_{21} + W_{22} = \tilde{O}(mk^2) + \tilde{O}(mk^2) + \tilde{O}\left(\frac{m^2}{\alpha}\right) = \tilde{O}\left(mk^2 + \frac{m^2}{\alpha}\right)$$

$$D = \max\{D_1, D_{21}, D_{22}\} = \max\{n^{1/2+o(1)}, \alpha k, n^{1/2+o(1)}\} = \max\{n^{1/2+o(1)}, \alpha k\}$$

where $k = O(\sqrt{n})$ and $\alpha \in [1, n]$.

7.3 Vertex Sampling Variation

In the section we explore a variation of the above parallel algorithm when using vertex sampling in local vertex cut as described in Section 5.5. We reiterate the main theorem from that section:

Theorem 7.3.1. *Given a directed graph $G = (V, E)$ and $k = O(\sqrt{n})$, if G is not k-connected then w.h.p Algorithm 8 finds a separator S with size less than k. If G is k-connected then Algorithm 8 always returns $\perp$. The algorithm takes $\tilde{O}(n^2k^2)$ time.*

The depth of this algorithm is the number of vertices DFS has to visit, thus $k\nu$ whre ν is passed as 2^i for every $i \in [1, \log n/k]$. Thus the above algorithm admits depth equal to $O(n)$.

Introducing the tradeoff parameter in this case we follow the same arguments to calculate work and depth for each part.

7.3.1 Analysis

We analyze the two cases of the algorithm separately:

7.3.1.1 Balanced Case

Since *ParallelDiam* runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ depth, our parallel Ford-Fulkerson implementation checking if the min-cut between two vertices is less than k runs in $\tilde{O}(mk)$ work and $n^{1/2+o(1)}$ depth.

Thus the first part of our algorithm runs in $W_1 = \tilde{O}(mk^2)$ work and $D_1 = n^{1/2+o(1)}$ depth.

7.3.1.2 Unbalanced Case

When the two components are unbalanced, again we have two separate cases for when the volume of L is smaller or larger than our tradeoff paramaeter α. For the first case we use LocalVC with vertex sampling, while for the latter case we use LocalVCReach as we discussed in the previous chapter.

7.3.1.3 LocalVC

LocalVC with vertex sampling takes $\tilde{O}(\nu^2 k^3)$ time, where ν is the size threshold. We run this for n/ν samples, so:

$$\frac{n}{\nu}\tilde{O}(\nu^2 k^3) = \tilde{O}(n\nu k^3)$$

At last we run this for all $\nu = 2^i$ for $i \in [1, \log \alpha]$ so the total time for this part of the algorithm can be written as:

$$W_{21} = \sum_{i=1}^{\log \alpha} \tilde{O}(n 2^i k^3) = \tilde{O}(n\alpha k^3)$$

For the depth:

$$D_{21} = \max_{0 \le i \le \log \alpha} \left\{2^i k\right\} = \alpha k$$

7.3.1.4 LocalVCReach

For the work:

$$W_{22} = \sum_{\log \alpha}^{\log n/k} \tilde{O}(m)\frac{n}{2^i}\log n = \tilde{O}(mn)\sum_{i=\log \alpha}^{\log n/k}\frac{1}{2^i} = \tilde{O}(mn)\left(\frac{2}{\alpha} - \frac{k}{n}\right)$$
$$= \tilde{O}\left(\frac{mn}{\alpha} - km\right)$$

Since $k = O(\sqrt{n})$:

$$W_{22} = \tilde{O}\left(\frac{mn}{\alpha}\right)$$

For the depth:

$$D_{22} = n^{1/2+o(1)}$$

7.3.1.5 Total

As we discussed in Chapter 3, the combined work and depth for the two steps is

$$W = W_1 + W_{21} + W_{22} = \tilde{O}(mk^2) + \tilde{O}(n\alpha k^3) + \tilde{O}\left(\frac{mn}{\alpha}\right)$$
$$= \tilde{O}\left(mk^2 + n\alpha k^3 + \frac{mn}{\alpha}\right)$$

$$D = \max\{D_1, D_{21}, D_{22}\} = \max\{n^{1/2+o(1)}, \alpha k, n^{1/2+o(1)}\}$$
$$= \max\{n^{1/2+o(1)}, \alpha k\}$$

where $k = O(\sqrt{n})$ and $\alpha \in [1, n]$.

7.4 When k is constant

When $k = O(1)$, if we revisit the work depth analysis we did earlier for the two different variations of sampling, we get:

7.4.1 LocalVC with Edge Sampling

$$\begin{cases} W_{edge} = \tilde{O}\left(mk^2 + \frac{m^2}{\alpha}\right) \\ D_{edge} = \max\{n^{1/2+o(1)}, \alpha k\} \end{cases} \xrightarrow{k=O(1)} \begin{cases} W_{edge} = \tilde{O}\left(\frac{m^2}{\alpha}\right) \\ D_{edge} = \max\{n^{1/2+o(1)}, \alpha\} \end{cases}$$

We can afford to only consider values for α that are in the range $[\sqrt{n}, n]$, since it only decreases work without increasing depth, thus making the work, depth:

$$W_{edge} = \tilde{O}\left(\frac{m^2}{\alpha}\right), \quad D_{edge} = \alpha$$

7.4.2 LocalVC with Vertex Sampling

$$\begin{cases} W_{vertex} = \tilde{O}\left(mk^2 + n\alpha k^3 + \frac{mn}{\alpha}\right) \\ D_{vertex} = \max\{n^{1/2+o(1)}, \alpha k\} \end{cases} \xrightarrow{k=O(1)} \begin{cases} W_{vertex} = \tilde{O}\left(\frac{mn}{\alpha}\right) \\ D_{vertex} = \max\{n^{1/2+o(1)}, \alpha\} \end{cases}$$

For this case, again we can afford to only consider values for $\alpha \in \left[n^{1/2}, n\right]$, since it only decreases work without increasing depth, thus making the work, depth:

$$W_{vertex} = \tilde{O}\left(\frac{mn}{\alpha}\right), \quad D_{vertex} = \alpha$$

We see that in this case the vertex sampling method yields a better result, concluding the proof of Theorem 7.1.5.

Chapter 8

Future Work

Building on the contributions of this thesis, there are several potential directions for further research:

1. **Exploring Efficient LocalVC Algorithms**: One open question is whether there exists an $O(\nu k)$-time LocalVC algorithm.
2. **Breaking Existing Time Bounds**: Another challenge is to break the current $O(n^3)$ time bound when $k = \Omega(n)$. This remains difficult, even if we assume the existence of an $O(\nu k)$-time LocalVC algorithm.
3. **Vertex-Weighted Graphs**: Extending the algorithm to handle vertex-weighted graphs efficiently poses another open problem, particularly when $m = O(n)$. The current LocalVC algorithm does not directly generalize to the weighted case, leaving room for further research.
4. **Single-Source Max-Flow Problem**: Identifying an $o(n^2)$-time algorithm for the single-source max-flow problem when $m = O(n)$ could provide insights into improving related graph algorithms.
5. **Dynamic and Distributed Settings**: Investigating the vertex connectivity problem in dynamic settings (e.g., with edge insertions and deletions) and distributed environments (such as the CONGEST model).
6. **Reducing Depth**: Removing the $n^{o(1)}$ term in the depth of parallel reachability algorithm and breaking the $\sqrt{n}$ depth bound with new methods are promising directions for achieving more efficient parallel algorithms.

For the parallel reachability problem, we conjecture the following:

Conjecture 1. There exists a parallel algorithm which takes as inputs x, μ and can explore μ vertices reachable from x in $O(\mu)$ work and $O(\sqrt{\mu})$ depth.

These directions represent exciting opportunities to refine and expand on the work presented here, potentially leading to more powerful and adaptable algorithms.

Bibliography

[1] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai, "Breaking quadratic time for small vertex connectivity and an approximation scheme", in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 241–252, ISBN: 9781450367059 (cit. on pp. 5, 7, 29, 32, 39, 61, 69, 72, 80, 109).

[2] A. Sidford, A. Jambulapati, and Y. P. Liu, "Parallel reachability in almost linear work and square root depth", *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 1664–1686, 2019 (cit. on pp. 5, 7, 29, 30, 32, 39, 87, 88, 91, 97, 106, 109).

[3] D. Kleitman, "Methods for investigating connectivity of large graphs", *IEEE Transactions on Circuit Theory*, vol. 16, no. 2, pp. 232–233, 1969 (cit. on pp. 27, 59–61).

[4] S. Even, "An algorithm for determining whether the connectivity of a graph is at least k", *SIAM Journal on Computing*, vol. 4, no. 3, pp. 393–396, 1975 (cit. on pp. 27, 59, 61).

[5] Z. Galil, "Finding the vertex connectivity of graphs", *SIAM Journal on Computing*, vol. 9, no. 1, pp. 197–199, 1980 (cit. on pp. 27, 59, 61).

[6] A. H. Esfahanian and S. Louis Hakimi, "On computing the connectivities of graphs and digraphs", *Networks*, vol. 14, no. 2, pp. 355–366, 1984 (cit. on pp. 27, 59, 61).

[7] D. W. Matula, "Determining edge connectivity in 0 (nm)", in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, IEEE, 1987, pp. 249–251 (cit. on pp. 27, 59, 61).

[8] M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, W. Keber, K. Mehlhorn, S. Näher, H. Rohnert, and T. Winter, "A probabilistic algorithm for vertex connectivity of graphs", *Information Processing Letters*, vol. 15, no. 3, pp. 135–136, 1982 (cit. on pp. 27, 59, 61).

[9] N. Linial, L. Lovasz, and A. Wigderson, "Rubber bands, convex embeddings and graph connectivity", *Combinatorica*, vol. 8, pp. 91–102, 1988 (cit. on pp. 27, 59–61).

[10] K. Censor-Hillel, M. Ghaffari, and F. Kuhn, "Distributed connectivity decomposition", in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014, pp. 156–165 (cit. on pp. 27, 59–61).

[11] H. Nagamochi and T. Ibaraki, "A linear-time algorithm for finding a sparse k-connected spanning subgraph of ak-connected graph", *Algorithmica*, vol. 7, no. 1, pp. 583–596, 1992 (cit. on pp. 27, 59–61).

[12] J. Cheriyan and J. H. Reif, "Directed st numberings, rubber bands, and testing digraph k-vertex connectivity", *Combinatorica*, vol. 14, no. 4, pp. 435–451, 1994 (cit. on pp. 27, 59–61).

[13] M. R. Henzinger, "A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity", *Journal of Algorithms*, vol. 24, no. 1, pp. 194–220, 1997 (cit. on pp. 27, 59–61).

[14] M. R. Henzinger, S. Rao, and H. N. Gabow, "Computing vertex connectivity: New bounds from old techniques", *Journal of Algorithms*, vol. 34, no. 2, pp. 222–250, 2000 (cit. on pp. 27, 59–61, 64).

[15] H. N. Gabow, "Using expander graphs to find vertex connectivity", *Journal of the ACM (JACM)*, vol. 53, no. 5, pp. 800–844, 2006 (cit. on pp. 27, 59–61).

[16] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network", *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956 (cit. on p. 60).

[17] R. Tarjan, "Depth-first search and linear graph algorithms", *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972 (cit. on p. 60).

[18] J. E. Hopcroft and R. M. Karp, "A n5/2 algorithm for maximum matchings in bipartite", in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, IEEE, 1971, pp. 122–125 (cit. on p. 60).

[19] A. Kanevsky and V. Ramachandran, "Improved algorithms for graph four-connectivity", *Journal of Computer and System Sciences*, vol. 42, no. 3, pp. 288–306, 1991 (cit. on p. 60).

[20] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, "New bounds for matrix multiplication: From alpha to omega", in *ACM-SIAM Symposium on Discrete Algorithms*, 2023 (cit. on p. 60).

[21] L. Georgiadis, "Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs", in *International Colloquium on Automata, Languages, and Programming*, Springer, 2010, pp. 738–749 (cit. on p. 60).

[22] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity", *SIAM journal on computing*, vol. 4, no. 4, pp. 507–518, 1975 (cit. on pp. 61, 64).

[23] D. R. Karger and C. Stein, "A new approach to the minimum cut problem", *J. ACM*, vol. 43, no. 4, pp. 601–640, Jul. 1996, ISSN: 0004-5411 (cit. on p. 68).

[24] A. V. Goldberg and S. Rao, "Beyond the flow decomposition barrier", *Journal of the ACM (JACM)*, vol. 45, no. 5, pp. 783–797, 1998 (cit. on p. 69).

[25] J. T. Fineman, "Nearly work-efficient parallel algorithm for digraph reachability", *SIAM Journal on Computing*, vol. 49, no. 5, STOC18-500-STOC18–539, 2018. eprint: https://doi.org/10.1137/18M1197850 (cit. on pp. 87, 88, 93, 95, 106).

[26] G. Bodwin and G. Hoppenworth, "Folklore sampling is optimal for exact hopsets: Confirming the $\sqrt{n}$ barrier", Nov. 2023, pp. 701–720 (cit. on p. 91).

[27] S. Kogan and M. Parter, "Having hope in hops: New spanners, preservers and lower bounds for hopsets", *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 766–777, 2022 (cit. on p. 91).

[28] T. H. Spencer, "Time-work tradeoffs for parallel algorithms", *J. ACM*, vol. 44, pp. 742–778, 1997.

[29] A. Wigderson, "The complexity of graph connectivity", in *Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science*, ser. MFCS '92, Berlin, Heidelberg: Springer-Verlag, 1992, pp. 112–132, ISBN: 354055808X.

[30] A. Abboud and V. V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems", *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pp. 434–443, 2014.

[31] R. Cole, "Parallel merge sort", *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 511–516, 1988.

[32] R. Becker, A. Karrenbauer, S. Krinninger, and C. Lenzen, "Near-optimal approximate shortest paths and transshipment in distributed and streaming models", *SIAM J. Comput.*, vol. 50, pp. 815–856, 2016.

[33] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd", in *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12, New York, New York, USA: Association for Computing Machinery, 2012, pp. 887–898, ISBN: 9781450312455.

[34] S.-E. Huang and S. Pettie, "Lower bounds on sparse spanners, emulators, and diameter-reducing shortcuts", in *Scandinavian Workshop on Algorithm Theory*, 2018.

[35] J. D. Ullman and M. Yannakakis, "High probability parallel transitive-closure algorithms", *SIAM J. Comput.*, vol. 20, no. 1, pp. 100–125, Feb. 1991, ISSN: 0097-5397.

[36] W. Hesse, "Directed graphs requiring large numbers of shortcuts", in *ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[37] G. Barnes, J. F. Buss, W. L. Ruzzo, and B. Schieber, "A sublinear space, polynomial time algorithm for directed s-t connectivity", *[1992] Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pp. 27–33, 1992.

[38] S. Krinninger and D. Nanongkai, "A faster distributed single-source shortest paths algorithm", *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 686–697, 2017.

[39] J. T. Fineman, "Nearly work-efficient parallel algorithm for digraph reachability", *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, 2017.

[40] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer, "Distributed verification and hardness of distributed approximation", in *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 363–372, ISBN: 9781450306911.

[41] R. Cole, "Correction: Parallel merge sort", *SIAM J. Comput.*, vol. 22, p. 1349, 1993.

[42] M. Ghaffari and R. Udwani, "Brief announcement: Distributed single-source reachability", *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, 2015.

[43] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems", in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, pp. 81–90.

[44] Y. T. Lee and A. Sidford, "Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow", in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, IEEE, 2014, pp. 424–433.

[45] A. Sinclair, *Cs271 lectures: Introduction to randomized algorithms*, https://people.eecs.berkeley.edu/~sinclair/cs271/, Lecture Notes, University of California, Berkeley, 2024.

[46] F. Le Gall, "Powers of tensors and fast matrix multiplication", in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, 2014, pp. 296–303.

[47] F. Le Gall, "Algebraic complexity theory and matrix multiplication.", in *ISSAC*, 2014, p. 23.

[48] L. Orecchia and Z. A. Zhu, "Flow-based algorithms for local graph clustering", in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2014, pp. 1267–1286.

[49] M. Henzinger, S. Rao, and D. Wang, "Local flow partitioning for faster edge connectivity", *SIAM Journal on Computing*, vol. 49, no. 1, pp. 1–36, 2020.

[50] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees", in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, 1981, pp. 114–122.

[51] D. A. Spielman and S.-H. Teng, "A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning", *SIAM Journal on computing*, vol. 42, no. 1, pp. 1–26, 2013.

[52] J. Chuzhoy and S. Khanna, "A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems", in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019, pp. 389–400.

[53] N. Veldt, D. Gleich, and M. Mahoney, "A simple and strongly-local flow-based method for cut improvement", in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1938–1947.

[54] M. Henzinger, S. Krinninger, and D. Nanongkai, "A subquadratic-time algorithm for decremental single-source shortest paths", in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2014, pp. 1053–1072.

[55] M. Ghaffari, "Near-optimal scheduling of distributed algorithms", in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, ser. PODC '15, Donostia-San Sebastián, Spain: Association for Computing Machinery, 2015, pp. 3–12, ISBN: 9781450336178.

[56] M. Ghaffari and J. Li, "Improved distributed algorithms for exact shortest paths", in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2018, Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 431–444, ISBN: 9781450355599.

[57] D. Nanongkai, "Distributed approximation algorithms for weighted shortest paths", in *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '14, New York, New York: Association for Computing Machinery, 2014, pp. 565–573, ISBN: 9781450327107.

[58] J. Cheriyan and R. Thurimella, "Algorithms for parallel k-vertex connectivity and sparse certificates", in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91, New Orleans, Louisiana, USA: Association for Computing Machinery, 1991, pp. 391–401, ISBN: 0897913973.

[59] R. Andersen and K. J. Lang, "An algorithm for improving graph partitions.", in *SODA*, vol. 8, 2008, pp. 651–660.

[60] S. Oveis Gharan and L. Trevisan, "Approximating the expansion profile and almost optimal local graph clustering", Oct. 2012, pp. 187–196, ISBN: 978-1-4673-4383-1.

[61] A. Abboud, V. V. Williams, and J. Wang, "Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs", in *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*, SIAM, 2016, pp. 377–391.

[62] D. Wang, K. Fountoulakis, M. Henzinger, M. W. Mahoney, and S. Rao, "Capacity releasing diffusion for speed and locality", in *International Conference on Machine Learning*, PMLR, 2017, pp. 3598–3607.

[63] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows", in *Soviet Math. Dokl.*, vol. 15, 1974, pp. 434–437.

[64] K.-i. Kawarabayashi and M. Thorup, "Deterministic global minimum cut of a simple graph in near-linear time", in *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, 2015, pp. 665–674.

[65] J. Hopcroft, "Dividing a graph into triconnected components", *Information and Computation*, vol. 79, pp. 43–59, 1988.

[66] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, "Dynamic minimum spanning forest with subpolynomial worst-case update time", in *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2017, pp. 950–961.

[67] T. Saranurak and D. Wang, "Expander decomposition and pruning: Faster, stronger, and simpler", in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2019, pp. 2616–2635.

[68] R. Andersen and Y. Peres, "Finding sparse cuts locally using evolving sets", in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 235–244.

[69] C. Wulff-Nilsen, "Fully-dynamic minimum spanning forest with improved worst-case update time", in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, 2017, pp. 1130–1143.