




14 ΙΑΝΟΥΑΡΙΟΥ 2019

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ
ΕΡΓΑΣΙΑ 3ΟΥ ΕΞΑΜΗΝΟΥ

ΝΙΚΟΛΑΟΣ ΔΗΜΗΤΡΑΚΟΠΟΥΛΟΣ
ΜΙΧΑΗΛ ΒΛΑΣΟΠΟΥΛΟΣ
ΓΙΩΡΓΟΣ ΑΘΑΝΑΣΟΠΟΥΛΟΣ



Περιεχόμενα

Εισαγωγή.....	3
Κατανόηση του προβλήματος.....	3
Μορφή του γραφήματος.....	3
Σχετικά με τα αρχεία των γραφημάτων	3
Διαδικασία μετατροπής του αρχείου σε γράφημα	4
Περισσότερα για τις λίστες γειτνίασης.....	4
Περισσότερα για το CSR (Compressed Sparse Rows)	5
Διάσχιση γράφου μέσω του αλγορίθμου BFS (Breadth-First Search)	6
Bidirectional BFS και τι κερδίζουμε από αυτό	6
Υλοποίηση του προβλήματος.....	7
Βήμα 1 ^ο : Από αρχείο σε λίστα γειτνίασης (AdjacencyListGraph)	7
Ανάγνωση του αρχείου	7
Υλοποίηση της λίστας γειτνίασης.....	8
Παρατηρήσεις	8
Βήμα 2 ^ο : Από λίστα γειτνίασης σε CSR (CSRGraph).....	8
Προαπαιτούμενα.....	9
Διαδικασία μετατροπής	9
Η συνάρτηση ανάκτησης γειτόνων μίας κορυφής getNeighbors().....	9
Η συνάρτηση για το ερώτημα ύπαρξης κορυφής vertexExists()	9
Παρατηρήσεις	9
Βήμα 3 ^ο : Υλοποίηση του Bidirectional BFS (BidirectionalBFS)	10
Απαιτήσεις της άσκησης.....	10
Πριν το αμφίδρομο ας πάμε στο απλό.....	10
Αμφίδρομο BFS	12
Προαπαιτούμενα υλοποίησης	12
Ο συγχρονισμός.....	13
Βελτιστοποιήσεις.....	13
Βήμα 4 ^ο : Διεπαφή με τον χρήστη	14
Επιλογή τρόπου εισόδου μέσω Terminal	14
Η πολυμορφική προσέγγιση με την InputInterface.....	14
Είσοδος με πληκτρολόγιο (KeyboardHandler)	14
Είσοδος με αρχείο (QueryFileHandler).....	14
Ελαττώματα του αλγορίθμου	15
Ένας εχθρός της μνήμης.....	15

Συντήρηση κώδικα	15
Παραδείγματα εκτέλεσης	16
Ένα ολοκληρωμένο παράδειγμα	16
Λίστες γειτνίασης:	17
CSR.....	17
Στιγμιότυπο εκτέλεσης.....	18
Ενδεικτική εκτέλεση soc-epinions1.txt.....	19

Εισαγωγή

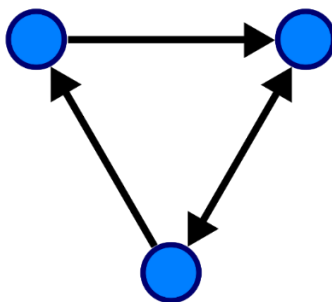
Η εργασία αυτή καλείται να λύσει το πρόβλημα των ερωτημάτων σύνδεσης μεταξύ 2 κορυφών ενός γράφου, ένα πρόβλημα που συναντιέται συχνά στον κόσμο της επιστήμης της πληροφορικής. Αν το καλοσκεφτούμε, πολλές φορές μέσα στην καθημερινότητα μας κάνουμε τέτοιου είδους ερωτήματα «φιλίας». Ίσως το πιο κλασσικό παράδειγμα είναι όταν θέλουμε να δούμε ποιος μας ακολουθεί σε κάποιο κοινωνικό δίκτυο. Με αυτήν την εργασία, μελετάμε την λειτουργία ενός γράφου και πως μπορούμε να υλοποιήσουμε μια λύση αναζήτησης σε αυτόν χρησιμοποιώντας Java.

Κατανόηση του προβλήματος

Μορφή του γραφήματος

Η εργασία μας ασχολείται αποκλειστικά με μη σταθμισμένους κατευθυνόμενους γράφους. Αυτό σημαίνει ότι ο ακμή μεταξύ των κόμβων είναι κατευθυνόμενες. Έτσι λοιπόν μια ακμή $\{x,y\}$ δεν είναι ίδια με μια ακμή $\{y,x\}$ σε αντίθεση με τους μη κατευθυνόμενους γράφους.

Το βάρος σε έναν γράφο χρησιμοποιείται για να δώσει ένα μέτρο σύγκρισης σε μία ακμή σε σχέση με τις άλλες. Η ερμηνεία του βάρους αφήνεται ξεκάθαρα στον δημιουργό του γράφου. Το γράφημα μας δεν έχει βάρη, επομένως κάθε ακμή σαν απόσταση θεωρείται ισάξια έναντι μιας άλλης.



Εικόνα 1.1: Ένας μη σταθμισμένος κατευθυνόμενος γράφος

Σχετικά με τα αρχεία των γραφημάτων

Για την εργασία χρησιμοποιήθηκαν αρχεία από τη βάση γράφων του [Stanford](#). Τα αρχεία περιλαμβάνουν ακμές, των οποίων οι κορυφές είναι συμβολισμένες με κάποιον unsigned ακέραιο που είναι το μοναδικό αναγνωριστικό (ID) τους. Όπως θα δούμε και παρακάτω, η υλοποίηση μας αντέχει κορυφές με ID από 0 μέχρι $2^{63}-1$ (long primitive type).

Η ιστοσελίδα περιλαμβάνει και γράφους με κορυφές οι οποίες ξεπερνούν το όριο μας άλλα έτσι κι αλλιώς είναι εκτός πραγματικότητας για αυτήν την εργασία, συνήθως λόγω του μεγέθους τους, και για λόγους απλότητας στην υλοποίηση που θα αναλυθούν παρακάτω. Ωστόσο τα προτεινόμενα αρχεία (wiki.txt και soc-erinions1.txt) είναι απόλυτα συμβατά μαζί με κάποιους πιο βαριούς γράφους που τηρούν αυτή τη σύμβαση στα IDs τους (ως και 15.000.000 ακμών πιθανότατα σε ένα μηχάνημα με 8GB RAM).

Η μορφή του αρχείου είναι η εξής: Κάθε γραμμή μπορεί να είναι **είτε** ακμή , **είτε** σχόλιο. Αν είναι σχόλιο, η γραμμή του ξεκινάει με τον χαρακτήρα '#'. Αν είναι ακμή, τότε η γραμμή αποτελείται από 2 αριθμούς οι οποίοι είναι χωρισμένοι από έναν whitespace χαρακτήρα για delimiter. Αυτή η παρατήρηση θα μας φανεί πολύ χρήσιμη όταν θα χρειαστεί να υλοποιήσουμε parser για το αρχείο.

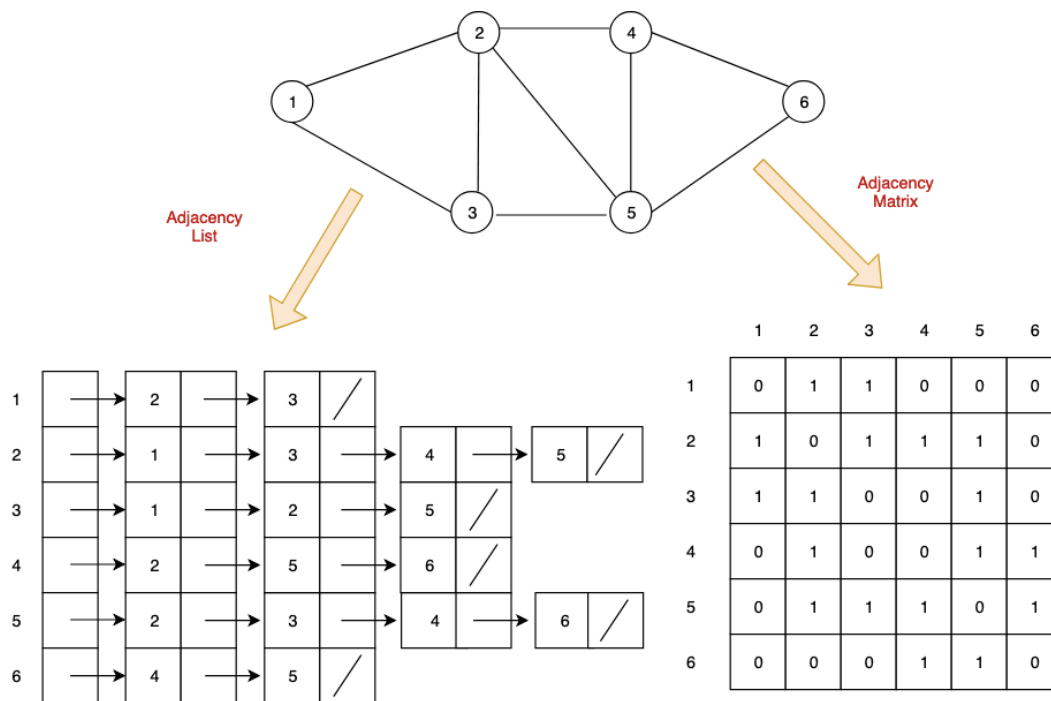
Διαδικασία μετατροπής του αρχείου σε γράφημα

Μία από τις πιο κρίσιμες αποφάσεις που πρέπει να πάρει ο σχεδιαστής του αλγορίθμου είναι με ποιον τρόπο θα μετατρέψει τις ακμές του αρχείου σε ένα πλήρως λειτουργικό γράφημα υλοποιημένο σε κάποια μορφή δομής δεδομένων. Ο τελικός σκοπός της άσκησης είναι να εξάγουμε το γράφημα σε μορφή CSR αλλά επειδή αυτή η διαδικασία απαιτεί εκ των προτέρων μια μήτρα ή λίστα γειτνίασης του γραφήματος, θα πρέπει πρώτα η μετατροπή να περάσει από ένα μεταβατικό σχέδιο. Αποφασίστηκε λοιπόν το μεταβατικό στάδιο αυτό να είναι μία ελαφρώς τροποποιημένη εκδοχή της λίστας γειτνίασης όπως την παρουσιάζει ο Robert Sedgewick. Αφού αποκτήσουμε τη λίστα γειτνίασης είναι εύκολο μετά ο γράφος να μετατραπεί σε μορφή CSR και να αποδεσμευτεί από την συνέχεια της άσκησης για λόγους εξοικονόμησης μνήμης.

Περισσότερα για τις λίστες γειτνίασης

Στη θεωρία των γράφων και την επιστήμη των υπολογιστών, μια λίστα γειτνίασης είναι μια συλλογή συνδεδεμένων λιστών που χρησιμοποιούνται για να αναπαραστήσουν ένα γράφο. Κάθε λίστα περιγράφει το σύνολο των γειτόνων μια κορυφής στον γράφο αυτόν.

Υπάρχουν πολλές εναλλαγές στην υλοποίηση μιας λίστας γειτνίασης. Ο Sedgewick στο βιβλίο του «Αλγόριθμοι σε C Μέρη 1-4» , λέει ότι πρέπει να διατηρηθεί μία συνδεδεμένη λίστα για κάθε κορυφή, ακόμα και αν αυτή δεν έχει γείτονες. Εμείς επιλέξαμε να αναπαραστήσουμε την λίστα γειτνίασης με **HashMap<Long, LinkedList<Long>>** ώστε να κρατήσουμε χαμηλούς τους χρόνους αναζήτησης της λίστας μίας συγκεκριμένης κορυφής και να γλυτώσουμε από περιττές καταχωρήσεις κορυφών που δεν έχουν γείτονες.



Εικόνα 1.2: Στα αριστερά βλέπουμε την λίστα γειτνίασης ενός γράφου. Στα δεξιά βλέπουμε την επίσης ισοδύναμη μορφή του πίνακα γειτνίασης.

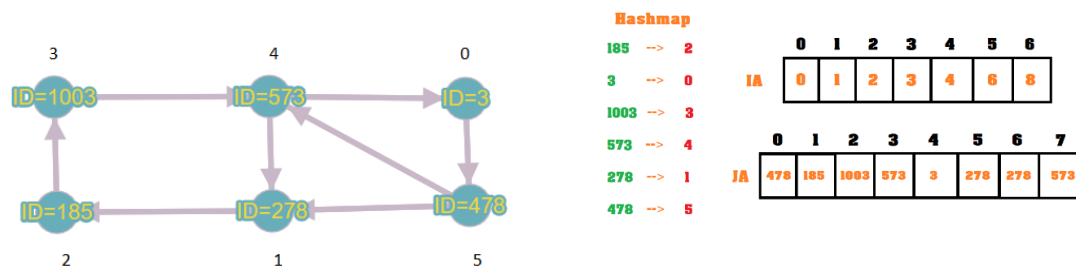
Περισσότερα για το CSR (Compressed Sparse Rows)

Η εκφώνηση της άσκησης όμως θέλει την αναπαράσταση του γραφήματος σε μορφή Compressed Sparse Rows η αλλιώς CSR. Το CSR είναι μία μήτρα *M* η οποία έχει 3 μονοδιάστατους πίνακες:

- Έναν πίνακα που περιέχει όλες τις μη μηδενικές τιμές.
- Έναν πίνακα με τις εκτάσεις των γραμμών
- Και έναν πίνακα με τους δείκτες των στηλών

Εφόσον το γράφημα μας όμως είναι μη σταθμισμένο, δεν είναι ανάγκη να κρατήσουμε και τους 3 πίνακες. Όλες οι μη-μηδενικές τιμές εδώ έτσι κι αλλιώς είναι 1. Αρκεί να κρατήσουμε μόνο τις εκτάσεις των γραμμών καθώς και τους δείκτες των στηλών. Θα αναφερόμαστε σε αυτούς τους πίνακες ως *IA* και *JA* αντίστοιχα από δω και πέρα. Όπως θα φανεί όμως και στην ενότητα της υλοποίησης, τελικά θα χρειαστεί και μία 3^η δομή δεδομένων της οποίας δουλειά θα είναι να αντιστοιχίζει τα *IDs* των κορυφών με τον αριθμοδείκτη στην οποία ο *IA* κρατάει το εύρος των θέσεων του *JA* που εκφράζουν τη συγκεκριμένη κορυφή.

Το κέρδος του CSR ουσιαστικά είναι η συμπίεση δεδομένων. Για παράδειγμα, για *N* κορυφές θα χρειαζόταν μια μήτρα γειτνίασης *N*×*N* θέσεων στη μνήμη. Με το CSR, ανάλογα με τον γράφο πάντα, μπορεί να επιτευχθεί συμπίεση τεράστιας κλίμακας.

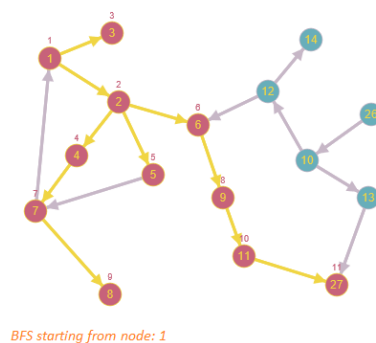


Εικόνα 1.3: Η τελική εικόνα της μετατροπής ενός γράφου σε CSR.

Διάσχιση γράφου μέσω του αλγορίθμου BFS (Breadth-First Search)

Η αναζήτηση με προτεραιότητα εύρους (η αλλιώς BFS από το Breadth-First Search) είναι ένας αλγόριθμος για διάσχιση ή αναζήτηση σε δέντρο ή γράφο. Ξεκινάει από έναν προκαθορισμένο κόμβο και εξερευνάει όλους τους γείτονες του πριν προχωρήσει στο επόμενο επίπεδο (χρησιμοποιεί level-order διάσχιση δηλαδή).

Η χρονική πολυπλοκότητα του αλγορίθμου αυτού είναι $O(|V|+|E|)$ (όπου $|V|$ και $|E|$ ο αριθμός των κορυφών και των ακμών αντίστοιχα) καθώς στην χειρότερη περίπτωση κάθε κορυφή και ακμή θα εξεταστεί. Ως ενδιαφέρουσα λεπτομέρεια , να σημειωθεί ότι το $O(|E|)$ μπορεί να κυμανθεί από $O(1)$ μέχρι και $O(|V|^2)$ ανάλογα με το πόσο αραιός είναι ο γράφος.



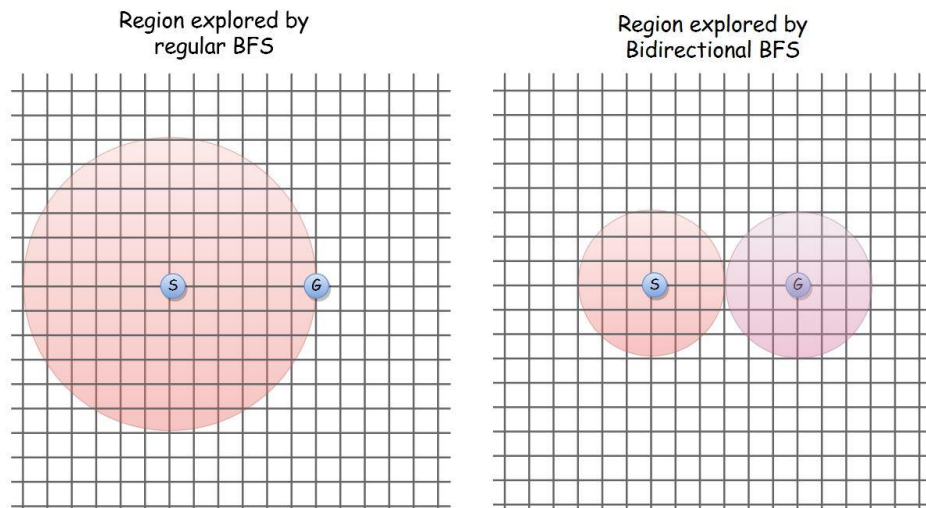
Εικόνα 1.4: Εκτέλεση του αλγορίθμου BFS.

Bidirectional BFS και τι κερδίζουμε από αυτό

Ως μπόνους στην άσκηση ζητήθηκε να υλοποιηθεί bidirectional BFS στην αναζήτηση. Η αμφίδρομη αναζήτηση είναι ένας αλγόριθμος που βρίσκει το συντομότερο μονοπάτι από μια αρχική κορυφή, σε μια κορυφή-στόχο. Τρέχει δύο ταυτόχρονες αναζητήσεις: Μία από το source προσπαθώντας να βρει το target και μία από το target που προσπαθεί να βρει το source. Ο αλγόριθμος τερματίζεται όταν βρεθεί σημείο τομής ανάμεσα στις 2 αναζητήσεις, δηλαδή όταν βρεθεί ένας κόμβος που τον έχουν επισκεφτεί και οι 2 αναζητήσεις. Μπορεί να χρησιμοποιηθεί οποιοσδήποτε αλγόριθμος αναζήτησης αλλά στα γραφήματα χωρίς βάρη προτιμάται ο BFS.

Ως προς την πολυπλοκότητα, αν ο παράγοντας διακλάδωσης είναι b και η απόσταση του source από το target είναι d τότε ένα απλό BFS θα έπαιρνε $O(b^d)$ χρόνο. Με την αμφίδρομη

αναζήτηση όμως θα πάρει $O(b^{d/2} + b^{d/2}) \Rightarrow O(b^{d/2})$ χρόνο που στην ουσία είναι πολύ πιο αποδοτικό.



Εικόνα 1.5: Η διαφορά στην περιοχή που εξερευνείται μεταξύ των 2 τακτικών.

Υλοποίηση του προβλήματος

Βήμα 1^ο: Από αρχείο σε λίστα γειτνίασης (AdjacencyListGraph)

Για την διαδικασία μετατροπής του αρχείου σε λίστες γειτνίασης (για τον κανονικό και αντίστροφο γράφο) δημιουργήθηκε η κλάση **AdjacencyListGraph**. Η συγκεκριμένη κλάση έχει ως πεδία 2 **HashMap<Long, LinkedList<Long>>** τα οποία κρατούν τις λίστες γειτνίασης καθώς και ένα πεδίο **int** που κρατάει τις συνολικές ακμές του αρχείου. Η γνώση αυτού του μεγέθους θα μας χαρίσει ένα τεράστιο optimization που θα φανερωθεί στην μετατροπή της λίστας γειτνίασης σε CSR.

Ανάγνωση του αρχείου

Το πρώτο πράγμα που πρέπει να γίνει στο πρόγραμμα είναι η ανάγνωση του αρχείου. Τη δουλειά αυτή αναλαμβάνει ο constructor της **AdjacencyListGraph**. Αυτό είναι ίσως και το πιο κομβικό σημείο για την απόδοση του κώδικα καθώς είναι ένα σημείο που ο βρόχος του θα εκτελεστεί πιθανότατα αρκετές εκατοντάδες χιλιάδες φορές. Χρησιμοποιώντας τις παρατηρήσεις σχετικά με τη μορφή των αρχείων κατασκευάσαμε έναν parser.

Χρησιμοποιήθηκε κωδικοποίηση ASCII επειδή το αρχείο είναι εγγυημένο ότι περιέχει μόνο ASCII χαρακτήρες. Μέσω ενός **BufferedReader** σαρώνουμε το κείμενο γραμμή προς γραμμή και κάνουμε tokenization χρησιμοποιώντας ως delimiters όλα τα whitespace characters ώστε να πάρουμε τους 2 αριθμούς και να τους χρησιμοποιήσουμε για τη συνέχεια.

Έχουν καλυφθεί όλα τα πιθανά σενάρια για exceptions (που έτσι κι αλλιώς είναι απαίτηση της Java), όπως το να δώσει ο χρήστης λανθασμένο path αρχείου γράφου ή το να συμβεί κάποιο απροσδόκητο I/O Error : σε αυτές τις περιπτώσεις πετιέται exception που τερματίζει

την δημιουργία της λίστας. Αντιθέτως, προσέχουμε το αποτέλεσμα του κάθε tokenization προειδοποιώντας τον χρήστη αν πάει κάτι στραβά σε μία γραμμή χωρίς όμως να τερματίσουμε και την εκτέλεση του αλγορίθμου.

Υλοποίηση της λίστας γειτνίασης

Η δομή `HashMap<Long, LinkedList<Long>>` κάνει την διαδικασία δημιουργίας της λίστας γειτνίασης πανεύκολη. Ουσιαστικά είναι μια διαδικασία 2 βημάτων:

- Για το source στο οποίο θέλουμε να αποθηκεύσουμε το target ως γείτονα, πρώτα ελέγχουμε αν υπάρχει στο HashMap. Αν δεν υπάρχει τον τοποθετούμε και κάνουμε initialize μια συνδεδεμένη λίστα γι' αυτόν.
- Κάνουμε add στην λίστα το target που αντιστοιχεί σε αυτό το source.

Αυτή η διαδικασία εφαρμόζεται για να αποθηκευτεί και ο κανονικός και ο αντίστροφος γράφος μέσω της συνάρτησης `addTargetToSource()`.

Παρατηρήσεις

Να σημειωθεί ότι τα keys των HashMap περιλαμβάνουν μόνο τις κορυφές όπου έχουν τουλάχιστον 1 γείτονα. Όποτε μια `containsKey()` μπορεί να μας πει μόνο αν υπάρχει κορυφή η οποία έχει γείτονες και **όχι** αν αυτή υπάρχει γενικότερα στο γράφημα. Αυτό είναι ένα μικρό memory optimization καθώς αν θελήσουμε μπορούμε να κάνουμε τέτοιο ερώτημα χρησιμοποιώντας την ένωση των κλειδιών του κανονικού και του αντίστροφου γράφου.

Για τον έλεγχο εξαιρέσεων σε αυτήν την κλάση δημιουργήθηκε και μια κλάση `AdjacencyListException`.

Βήμα 2^ο: Από λίστα γειτνίασης σε CSR (CSRGraph)

Όπως έχουμε αναφέρει και πιο πάνω, το CSR μας αποτελείται από 2 πίνακες και ένα HashMap που αναλαμβάνει τον ρόλο του «μεταφραστή» ανάμεσα σε IDs και την αντιστοιχία τους στο CSR. Αυτά είναι και τα 3 πεδία της κλάσης αυτής:

- Έναν `long[] JA` που κρατάει όλα τα targets.
- Έναν `int[] IA` που κάνει το prefix-summing και στην ουσία είναι κάτι σαν διευθυνσιοδοτητής για την JA.
- Και ένα `HashMap<Long, Integer> idMap` για να κάνει το mapping.

Απορία μπορεί να δημιουργήσει το γιατί ο IA είναι τύπου `int` και όχι `long`. Αν φανταστούμε τον IA σαν τον διευθυνσιοδοτητή της JA τότε αυτό σημαίνει ότι ο IA πρέπει να δέχεται τιμές μέχρι το μέγιστο αριθμοδείκτη που μπορεί να δεχτεί ο JA. Αν διαβάσει κάποιος το documentation της Java, θα δει ότι το μέγιστο index των πινάκων είναι $2^{31} - 1$ μείον κάτι ακόμα που είναι JVM dependent. Σώζεται αρκετή μνήμη με αυτήν την παρατήρηση.

Τη δυνατότητα να χρησιμοποιήσουμε απλούς πίνακες και όχι κάποια δυναμική δομή δεδομένων που θα μας θυσίαζε και χώρο (επειδή θα απαιτούσε reference τύπους) και απόδοση (επειδή δεν θα είχαμε εγγυημένο $O(1)$) μας τη δίνουν τα sizes που προέρχονται

από τη **AdjacencyListGraph**. Το πλήθος των sources από τη λίστα γειτνίασης φανερώνουν το τελικό μέγεθος των JA και idMap και το πεδίο totalEdges το μέγεθος του JA. Να υπενθυμισθεί ότι στη Java απαγορεύεται το resize στα arrays.

Προαπαιτούμενα

Πριν ξεκινήσουμε να περιγράψουμε το πως λειτουργεί η CSRGraph, πρέπει να μιλήσουμε συνοπτικά για την κλάση **GraphType**. Η **GraphType** είναι ένα απλό Enum όπου θα επιτρέπει στον πελάτη να επιλέγει αν επιθυμεί να γίνει μετατροπή είτε του κανονικού είτε του αντίστροφου γράφου της **AdjacencyListGraph** σε μορφή CSR.

Διαδικασία μετατροπής

Η μετατροπή λαμβάνει χώρα στον constructor. Μέσω του **GraphType** προσδιορίζεται σε ποιο από τα 2 γραφήματα θα εφαρμοστεί η μετατροπή. Έπειτα γίνεται ένας γρήγορος έλεγχος για το αν το γράφημα αυτό κρατάει όντως δεδομένα. Αν κάτι πήγε στραβά στην δημιουργία της λίστας γειτνίασης τότε πετιέται exception.

Έπειτα γίνεται initialize σε όλες τις δομές του CSR και περνάμε από κάθε entry του HashMap που κρατάει την λίστα γειτνίασης εφαρμόζοντας ταυτόχρονη αντιγραφή των targets στον JA και prefix-summing στον IA. Η διαδικασία της αντιστοίχισης ID με τη θέση στον CSR λαμβάνει χώρα επίσης εδώ.

Αν τελειώσει επιτυχώς ο constructor, τότε έχουμε ένα πλήρως λειτουργικό CSR γράφημα.

Η συνάρτηση ανάκτησης γειτόνων μίας κορυφής getNeighbors()

Όπως απαιτεί και η εκφώνηση της άσκησης, πρέπει να κατασκευαστεί μια συνάρτηση που να επιστρέφει όλους τους γείτονες κάποιας ζητούμενης κορυφής ώστε να μπορούμε να τροφοδοτήσουμε εύκολα το BFS με δεδομένα. Τη δουλειά αυτή την κάνει η getNeighbors() η οποία δέχεται ένα ID και επιστρέφει όλους του γείτονες του σε μορφή ουράς. Σε περίπτωση που μία κορυφή δεν έχει γείτονες, επιστρέφεται μια κενή ουρά.

Η συνάρτηση για το ερώτημα ύπαρξης κορυφής vertexExists()

Υλοποιήθηκε μια έξτρα λειτουργία που θα μας φανεί χρήσιμη στο Bidirectional BFS. Με την συνάρτηση vertexExists() μπορούμε να δούμε αν υπάρχει μια συγκεκριμένη κορυφή στο CSR η οποία είναι source (ισχύουν οι ίδιοι περιορισμοί που έχουν συζητηθεί και στη λίστα γειτνίασης).

Παρατηρήσεις

Η CSRGraph κατασκευάστηκε έτσι ώστε να είναι ανεξάρτητη της AdjacencyListGraph. Αυτό σημαίνει ότι εφόσον μετατραπούν οι 2 ζητούμενοι γράφοι σε CSR τότε μπορούμε να ελπίζουμε ότι ο GC θα καταλάβει γρήγορα ότι η δουλειά του αντικειμένου της

AdjacencyListGraph τελείωσε εδώ καθώς βγαίνει out-of-scope για το υπόλοιπο του αλγορίθμου και έτσι να αποδεσμεύσει ένα (πολύ) μεγάλο ποσοστό της χρησιμοποιούμενης μνήμης.

Βήμα 3^ο: Υλοποίηση του Bidirectional BFS (BidirectionalBFS)

Απαιτήσεις της άσκησης

Η άσκηση μας ζητάει να υπολογίζουμε αν υπάρχει μονοπάτι μεταξύ 2 κορυφών και αν όντως υπάρχει να επιστρέφει το μέγεθος του και πόσο χρόνο χρειάστηκε σε ms για να υπολογίσει το αίτημα. Εκμεταλλευόμενοι την ευελιξία της Java, θα παρουσιάσουμε μία παραλλαγή του αλγορίθμου σε σχέση με αυτόν που παρουσιάστηκε στην τάξη, κάνοντας τον αλγόριθμο περισσότερο δυναμικό.

Πριν το αμφίδρομο ας πάμε στο απλό

Όπως ξέρουμε ο BFS είναι ένας αλγόριθμος που κάνει level-order διάσχιση χρησιμοποιώντας μια FIFO ουρά. Αρχίζει από μία προκαθορισμένη κορυφή, παίρνει τους γείτονες της και μετά τους γείτονες των γειτόνων και κάπως έτσι διατρέχεται ο γράφος μέχρι να επισκεφτεί όλες τις κορυφές που έχουν μονοπάτι από τη συγκεκριμένη κορυφή. Εφόσον εμείς κάνουμε αναζήτηση, δεν μας ενδιαφέρει να διατρέξουμε όλο το γράφημα, αντιθέτως πρέπει να σταματήσουμε με το που βρούμε ότι επισκεφτήκαμε την κορυφή που ψάχνουμε. Αν την βρούμε, αυτό σημαίνει ότι υπάρχει μονοπάτι που τις συνδέει. Η καταγραφή της κατάστασης επισκεψιμότητας σε έναν γράφο, καθώς η απόσταση του από το source σε έναν κλασικό αλγόριθμο καταγράφονται με boolean και int πίνακες αντίστοιχα. Αυτή η μέθοδος όμως παρουσιάζει 2 μειονεκτήματα:

- Απαιτεί από το BFS να ξέρει εκ των προτέρων το μέγεθος όλου του γραφήματος.
- Σπαταλάει πολύ μνήμη καθώς πρέπει να δεσμευτεί χώρος για κάθε κορυφή ακόμα και αν η αναζήτηση δεν φτάσει ποτέ εκεί.

Η λύση σε αυτά τα προβλήματα ήταν και εδώ ένα HashMap <Long,Integer> το οποίο κάνει 2-σε-1 δουλειά:

- Με την containsKey() ελέγχουμε αν έχει γίνει επίσκεψη σε έναν κόμβο
- Το value συμβολίζει την απόσταση του συγκεκριμένου key από την αρχική κορυφή.

Το BFS μετά ανάλογα το αποτέλεσμα επιστρέφει τις κατάλληλες πληροφορίες μέσω της δημιουργίας ενός BFSResult αντικειμένου το οποίο στο απλό BFS είχε λίγο πιο απλή μορφή από την τελική της στο διπλό BFS. Παρακάτω είναι ο κώδικας (δεν είναι μέρος της τελικής εργασίας):

```

public BFSResult connectionQuery(long source_id, long target_id)
{
    //Start counting BFS query time
    long tStart = System.nanoTime();

    if(source_id == target_id)
        return new
BFSResult(source_id,target_id,true,0,Duration.ofNanos(System.nanoTime()-tStart));

    // Create a queue for BFS
    LinkedList<Long> queue = new LinkedList<>();
    //Create a HashMap that holds the distance level as well as if a node has
been visited
    HashMap<Long,Integer> nodeInfo = new HashMap<>();

    long current_Source = source_id;

    // Mark the current node as visited, set its' distance to 0 and enqueue it
    nodeInfo.put(current_Source,0);
    queue.add(current_Source);

    while (queue.size() != 0)
    {
        // Dequeue a vertex from queue and print it
        current_Source = queue.poll();

        // Get all the children of the source
        LinkedList<Long> children = getChildren(current_Source);

        while(!children.isEmpty())
        {
            long child = children.poll();

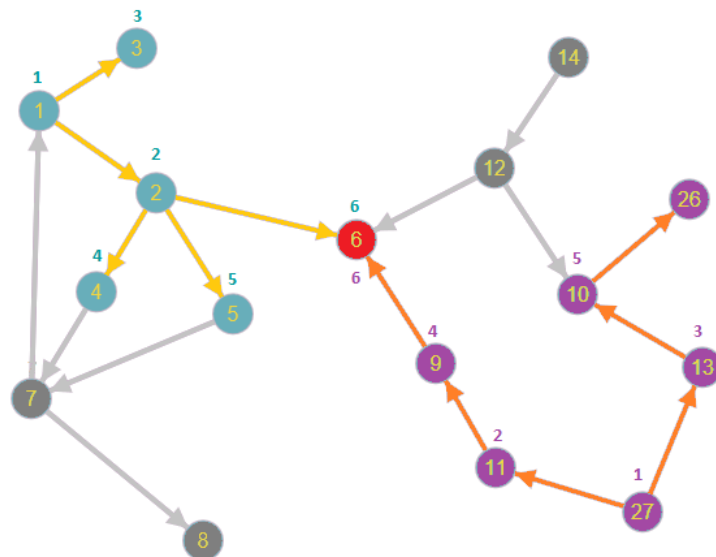
            if ( !nodeInfo.containsKey(child) )
            {
                //Mark the distance of the children as one level deeper than its' parent
                nodeInfo.put(child,nodeInfo.get(current_Source) + 1);
                //Check if any of the children which have not been visited is our target
                if(child == target_id)
                    return new BFSResult(source_id, target_id, true, nodeInfo.get(child),
Duration.ofNanos(System.nanoTime() - tStart));
                //Add the child to the queue
                queue.add(child);
            }
        }
    }
    return new BFSResult(source_id, target_id, false, null,
Duration.ofNanos(System.nanoTime() - tStart));
}

```

Αμφίδρομο BFS

Χρησιμοποιώντας όλες τις προηγούμενες παρατηρήσεις, μπορούμε να προχωρήσουμε στην εύρεση υλοποίησης για το αμφίδρομο BFS. Όπως είπαμε και στην ενότητα της κατανόησης του προβλήματος, το αμφίδρομο BFS δεν είναι τίποτε άλλο από την εκτέλεση 2 απλών BFS εναλλάξ, μέχρι να έχουν επισκεφτεί εκ κοινού κάποια κορυφή. Αυτό το σημείο λέγεται το σημείο τομής και είναι μια κρίσιμη πληροφορία, ειδικά για τον υπολογισμό της απόστασης του μονοπατιού.

Η απόσταση ουσιαστικά είναι το άθροισμα 2 αποστάσεων: της απόστασης από το source μέχρι το σημείο τομής και της απόστασης από το target πάλι μέχρι το σημείο τομής. Επειδή το γράφημα μας είναι κατευθυνόμενο, για την αντίστροφη πλοήγηση θα χρειαστεί ο αντίστροφος γράφος. Έχουμε φροντίσει από τα προηγούμενα βήματα να τον έχουμε ήδη διαθέσιμο.



Εικόνα 3.1: Γραφική αναπαράσταση του αιτήματος 1->27. Το BFS από το source έκανε διάσχιση 2 επιπέδων και του BFS από το target έκανε διάσχιση 3 επιπέδων. Αν αυτό το query εκτελούνταν με αυτό το γράφημα στο πρόγραμμά μας, θα επέστρεφε ότι υπάρχει σύνδεση με απόσταση 5.

Προαπαιτούμενα υλοποίησης

Για την πιο εύκολη υλοποίηση του αμφίδρομου BFS χρειάστηκαν 2 βοηθητικές κλάσεις:

- Την κλάση Intersection η οποία αποθηκεύει αν βρέθηκε σημείο τομής και αν ναι ποιο είναι.
- Την κλάση BFSResult η οποία αποθηκεύει το τελικό αποτέλεσμα του query του χρήστη.

Η δομή τους είναι αρκετά απλή και για λόγους συντομίας δεν θα αναλυθούν περαιτέρω.

Ο συγχρονισμός

Εφόσον έχουμε βρει τακτική για να δουλεύει καλά ένα BFS μόνο του, το μόνο που μένει είναι να συγχρονίσουμε δύο από αυτά ώστε να τρέχουν για λίγο, να ελέγχουν αν βρήκαν κάποιο σημείο τομής με το άλλο BFS και αν όχι να δώσουν τον έλεγχο στο άλλο BFS ώστε να κάνει την ίδια διαδικασία. Ως πηγές δεδομένων αυτά τα BFS χρειάζονται τον κανονικό και τον αντίστροφο γράφο σε μορφή CSR γι' αυτό και δημιουργήθηκε η κλάση BidirectionalBFS με αυτά τα 2 ως πεδία. Εφόσον έχουμε κανονίσει εύκολη πρόσβαση σε αυτά πλέον είναι η ώρα να γράψουμε τον κώδικα που κάνει τη δουλειά που περιγράψαμε παραπάνω.

Θα χρειαστούμε 2 συναρτήσεις:

- Μια συνάρτηση graphBFS() η οποία δέχεται έναν CSRGraph, μια FIFO ουρά για τους σκοπούς του BFS και (εδώ είναι το μυστικό) τα 2 HashMaps που περιέχουν τις πληροφορίες των κορυφών στα 2 BFS που εκτελούνται. Το ένα HashMap διακρίνεται από το άλλο από εναλλαγή θέσης στο κάλεσμά τους. Η συνάρτηση αυτή προχωράει την αναζήτηση ένα επίπεδο πιο πέρα για τον γράφο τον οποίο έχει καλεστεί, κάνει ότι θα έκανε και ο απλός αλγόριθμος BFS και μετά ελέγχει αν σε κάποια από τις κορυφές γείτονες έχει γίνει ήδη επίσκεψη από το άλλο BFS. Αν έχει γίνει τότε επιστρέφει το σημείο Intersection αλλιώς επιστρέφει false στο intersectionExists.
- Η άλλη συνάρτηση connectionQuery() θα παίξει το ρόλο του «τροχονόμου» και θα δίνει εναλλάξ κλήσεις της graphBFS() δίνοντας εναλλάξ πρόοδο στα 2 BFS. Αν κάποια κλήση της graphBFS() επιστρέφει ότι υπάρχει σημείο τομής τότε τα BFS τερματίζονται και γίνεται άμεσα υπολογισμός της συνολικής απόστασης του μονοπατιού. Στο τέλος επιστρέφεται ένα BFSResult με τα τελικά αποτελέσματα. Φυσικά έχει καλυφθεί και το σενάριο του να μην βρεθεί ποτέ σημείο τομής, κάτι που σημαίνει ότι δεν υπάρχει σύνδεση οπότε επιστρέφεται και σε αυτήν την περίπτωση το κατάλληλο BFSResult.

Βελτιστοποιήσεις

Στην τελική πορεία της ανάπτυξης της εφαρμογής, προστέθηκαν 2 τακτικές που αποτρέπουν την αργοπορημένη εκτέλεσης ενός ερωτήματος φιλίας:

- Αν δοθεί κορυφή η οποία δεν υπάρχει σε κάποιο από τα 2 CSR τότε επιστρέφεται άμεσα αποτέλεσμα ότι δεν υπάρχει σύνδεση μαζί με ξεχωριστά στοιχεία για την ύπαρξη της καθεμιάς κορυφής
- Αν δοθεί source που είναι ίσο με το target τότε γίνεται ένας στιγμιαίος έλεγχος αν στα 2 CSR υπάρχει αυτή η κορυφή και αν ναι επιστρέφεται αληθές αποτέλεσμα με απόσταση 0.

Βήμα 4^ο: Διεπαφή με τον χρήστη

Μόνο ένα πράγμα λείπει από την εφαρμογή για να είναι μία ολοκληρωμένη εφαρμογή: να έχει μία σωστή διεπαφή με τον χρήστη. Επιλέχθηκε να υπάρχουν 2 τρόποι για τους οποίους να μπορεί κάποιος να δώσει είσοδο: με πληκτρολόγιο ή με είσοδο αρχείου.

Επιλογή τρόπου εισόδου μέσω Terminal

Ο χρήστης πρέπει να επιλέξει την είσοδο του από την γραμμή εντολών.

- Αν θέλει εκτέλεση με είσοδο από πληκτρολόγιο αρκεί να δώσει ως όρισμα μόνο το path του αρχείου του γράφου (`/.../graph.txt`)
- Αν θέλει εκτέλεση με είσοδο από αρχείο αρκεί να δώσει ως όρισμα το path του αρχείου του γράφου ακολουθημένο με το flag `-f` και το path του αρχείου που περιέχει τα queries(`/.../graph.txt -f /.../queries.txt`)

Η πολυμορφική προσέγγιση με την `InputInterface`

Για να είναι ο κώδικας πιο ευανάγνωστος και εν τέλει να γραφεί πιο εύκολα προτιμήθηκε μια πολυμορφική προσέγγιση στο θέμα της εισόδου δημιουργώντας έτσι το interface `InputInterface` το οποίο γίνεται implemented από τις κλάσεις `KeyboardHandler` και `QueryFileHandler`. Ανάλογα λοιπόν την επιλογή του χρήστη από το command line θα εκτελεστεί και η κλήση `processQueries()` της αντίστοιχης κλάσης.

Είσοδος με πληκτρολόγιο (`KeyboardHandler`)

Η `processQueries()` της `KeyboardHandler()` εκτελεί έναν ατέρμον βρόχο ο οποίος δέχεται συνεχόμενα ζεύγη κορυφών ώστε να τα κάνει `VertexPair` και να τα στείλει στο `BidirectionalBFS`. Στο τέλος κάθε query ερωτάται ο χρήστης αν θέλει να πληκτρολογήσει κι άλλο ερώτημα με (Yes/No). Μεγάλη προσοχή έχει δοθεί στον έλεγχο εγκυρότητας εισόδου καθώς ο χρήστης υποχρεώνεται να βάλει.

Είσοδος με αρχείο (`QueryFileHandler`)

Η `processQueries()` της `QueryFileHandler()` δέχεται ως όρισμα το path του αρχείου των queries, το οποίο πρέπει να έχει ίδια μορφή με αυτή του αρχείου του γράφου. Τα ερωτήματα που διαβάζονται περνάνε σε μια ουρά η οποία έπειτα προωθεί τα ερωτήματα στο BFS. Έχουν καλυφθεί τα πιθανά exceptions με τη χρήση της `QueryFileHandlerException`.

Ελαττώματα του αλγορίθμου

Ένας εχθρός της μνήμης

Ο αλγόριθμος αυτός είναι σχεδιασμένος ώστε να «νικάει το ρολόι» με καλές επιδόσεις στην ανάγνωση και στην δημιουργία ενός γράφου και στην δημιουργία ερωτημάτων φιλίας με όμως μία πολύ μεγάλη και σημαντική υποχώρηση: πεινάει πραγματικά για μνήμη. Η έλλειψη Serialization/Deserialization τεχνικών ώστε να μπει και ο δίσκος στο παιχνίδι οδηγούν τον GC της Java στο να καταναλώνει μεγάλο μέρος της CPU προσπαθώντας να ελευθερώσει μνήμη καθώς βλέπει ο χώρος του σωρού τελειώνει απειλητικά όταν παρσάρονται αρχεία αρκετών εκατομμυρίων ακμών. Η υλοποίηση του bidirectional search έκανε αυτό το πρόβλημα πιο εμφανές καθώς απαιτεί σχεδόν διπλάσια δεδομένα στη μνήμη. Ευτυχώς όμως, αυτό το φαινόμενο συμβαίνει μέχρι να φορτωθεί επιτυχώς ο γράφος. Τα queries στο BFS έπειτα δεν επηρεάζονται από τέτοιου είδους bottlenecks (αν δεν έχει πεταχτεί πριν από αυτό OutOfMemoryException).

Συντήρηση κώδικα

Ο συγκεκριμένος αλγόριθμος είναι σχεδιασμένος να δουλεύει μόνο με κορυφές των οποίων μοναδικά χαρακτηριστικά είναι το ID τους (long) . Μια πιο προσεκτική υλοποίηση θα χρησιμοποιούσε μια πιο γενική κλάση Vertex η οποία θα χαρακτήριζε τον κόμβο και θα επέτρεπε πιο εύκολο Refactor.

Βεβαία πιθανότατα, αν ο συγκεκριμένος κώδικας πέρναγε από production code review είναι πιθανό να βρισκόントυσαν κι άλλα ψεγάδια. Εργασίες σαν αυτή πάντως σε ωθούν να γράψεις όσο το δυνατόν πιο ποιοτικό κώδικα μπορείς.

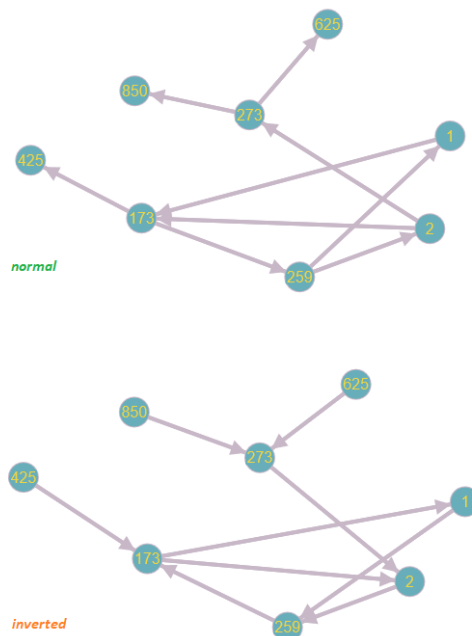
Παραδείγματα εκτέλεσης

Ένα ολοκληρωμένο παράδειγμα

Έχουμε το ακόλουθο γράφημα σε μορφή αρχείου:

273	625
273	850
173	425
173	259
259	2
2	173
1	173
259	1
2	273

Αυτός ο γράφος γραφικά αναπαρίσταται έτσι σε κανονική και αντεστραμμένη μορφή:



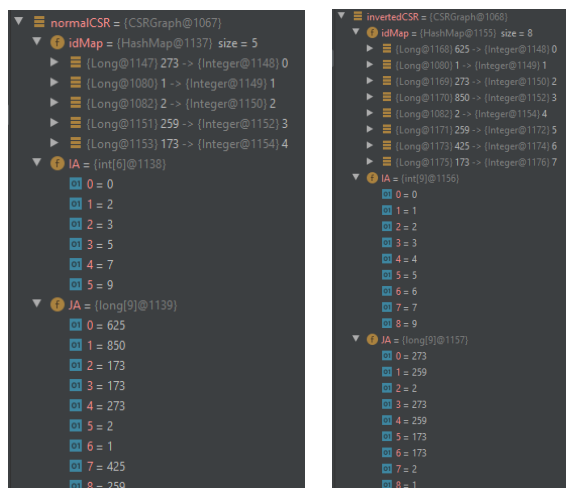
Θα παρακολουθήσουμε τη μνήμη του προγράμματος μέχρι την δημιουργία των 2 CSR.

Λίστες γειτνίασης:



Εικόνα 5.1: Οι 2 λίστες γειτνίασης για τον κανονικό και αντίστροφο γράφο του παραδείγματος.

CSR



Εικόνα 5.2: Οι 2 CSR πίνακες

Στιγμιότυπο εκτέλεσης

Χρησιμοποιήθηκε είσοδος πληκτρολογίου:

```
Parsing file C:\Users\Public\example.txt
Loaded 9 edges in 1ms
Normal graph conversion to CSR took 0ms
Inverted graph conversion to CSR took 0ms
Using keyboard input.
Give a node pair (source,target)
173 850
BFSResult{source_id=173, source_exists=true, target_id=850, target_exists=true, areConnected=true, distance=4, timeElapsed=0ms}
Would you like an another query? (Yes/No)
yes
Give a node pair (source,target)
1 173
BFSResult{source_id=1, source_exists=true, target_id=173, target_exists=true, areConnected=true, distance=1, timeElapsed=0ms}
Would you like an another query? (Yes/No)
yes
Give a node pair (source,target)
625 273
BFSResult{source_id=625, source_exists=false, target_id=273, target_exists=true, areConnected=false, distance=null, timeElapsed=0ms}
Would you like an another query? (Yes/No)
yes
Give a node pair (source,target)
2 2
BFSResult{source_id=2, source_exists=true, target_id=2, target_exists=true, areConnected=true, distance=0, timeElapsed=0ms}
Would you like an another query? (Yes/No)
yes
Give a node pair (source,target)
10000 25000
BFSResult{source_id=10000, source_exists=false, target_id=25000, target_exists=false, areConnected=false, distance=null, timeElapsed=0ms}
Would you like an another query? (Yes/No)
yes
Give a node pair (source,target)
173 625
BFSResult{source_id=173, source_exists=true, target_id=625, target_exists=true, areConnected=true, distance=4, timeElapsed=0ms}
Would you like an another query? (Yes/No)
no
Thank you for using our software!
```

Ενδεικτική εκτέλεση soc-epinions1.txt

Χρησιμοποιήθηκε το αρχείο soc-epinions1.txt μαζί με το ακόλουθο αρχείο εισόδου:

```
#Testfile
273 850
273 625
253 2
2 273
15 7
173 425
1 570
5 7
10 8
8 9
5 13
850 570
625 173
173 253
173 425
253 1
15 10
13 15
10 5
```

Το αποτέλεσμα της εκτέλεσης ήταν το παρακάτω:

```
Parsing file C:\Users\Public\soc-epinions.txt
Loaded 508837 edges in 420ms
Normal graph conversion to CSR took 28ms
Inverted graph conversion to CSR took 18ms
Using file input mode.
Parsing query file C:\Users\Public\test.txt
Loaded 19 queries in 0ms
BFSResult{source_id=273, source_exists=true, target_id=850, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=273, source_exists=true, target_id=625, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=253, source_exists=true, target_id=2, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=2, source_exists=true, target_id=273, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=15, source_exists=true, target_id=7, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=173, source_exists=true, target_id=425, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=1, source_exists=true, target_id=570, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=5, source_exists=true, target_id=7, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=10, source_exists=true, target_id=8, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=8, source_exists=true, target_id=9, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=5, source_exists=true, target_id=13, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=850, source_exists=true, target_id=570, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=625, source_exists=true, target_id=173, target_exists=false, areConnected=false, distance=null, timeElapsed=0ms}
BFSResult{source_id=173, source_exists=true, target_id=253, target_exists=false, areConnected=false, distance=null, timeElapsed=0ms}
BFSResult{source_id=173, source_exists=true, target_id=425, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=253, source_exists=true, target_id=1, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=15, source_exists=true, target_id=10, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
BFSResult{source_id=13, source_exists=true, target_id=15, target_exists=true, areConnected=true, distance=3, timeElapsed=0ms}
BFSResult{source_id=10, source_exists=true, target_id=5, target_exists=true, areConnected=true, distance=2, timeElapsed=0ms}
Thank you for using our software!

Process finished with exit code 0
```