Κωνσταντίνος Τσιάρας AM: 1115201500165

Λειτουργικά Συστήματα

1η Προγραμματιστική Εργασία

Μεταγλώττιση: $make$

Εκτέλεση: $./graph\ [-i\ inputfile]\ [-o\ outputfile]$

(το inputfile ή/και το ouputfile μπορούν να παραλειφθούν)

Έχουν υλοποιηθεί όλα τα ζητούμενα της άσκησης και έχουν ακολουθηθεί ***αυστηρά*** οι δοθέντες Κανόνες Μορφοποίησης Εξόδου.

Χρησιμοποιούμενες Δομές:

❖ Nodes:

Για την αποθήκευση των κόμβων κάνω χρήση **hash table** το οποίο αυξάνει σε μέγεθος δυναμικά. Πιο συγκεκριμένα, ξεκινάω δημιουργώντας έναν πίνακα *INITIAL_BUCKETS_NUM* θέσεων (hash-defined ως 16) και κάθε φορά που το πλήθος των αποθηκευμένων κόμβων υπερβεί το όριο *bucketsNum*HASHING_LOAD_FACTOR* (hash-defined ως 0.7) τότε το hash table επαναδημιουργείται με διπλάσιο πλήθος buckets. Εννοείται πως οι κόμβοι που υπήρχαν στο παλιό hash table μεταφέρονται στις κατάλληλες θέσεις τους στο καινούργιο χωρίς να επηρεαστούν με οποιονδήποτε τρόπο.

Αν και προφανώς ιδανικά θα θέλαμε κάθε bucket να φυλάσσει έναν και μόνο κόμβο, όταν τα δεδομένα μας είναι εκ των προτέρων άγνωστα τα collisions (εισαγωγές σε μη κενά buckets) είναι αναπόφευκτα ακόμα και με καλή hash function. Για την αντιμετώπιση αυτών, κάθε κόμβος διατηρεί κόμβο στον «επόμενό» του (στο bucket) και άρα κάθε bucket ουσιαστικά διατηρεί μια αλυσίδα από κόμβους.

Σε ότι αφορά τις πολυπλοκότητες των διάφορων ενεργειών, ισχύουν τα συνηθισμένα για τα hash tables: Λόγω των ενδεχόμενων collisions, στη χείριστη περίπτωση έχει δημιουργηθεί λίστα με όλους τους κόμβους σε κάποιο ένα bucket οπότε και η αναζήτηση, η εισαγωγή και η διαγραφή κόμβου έχουν worst-case πολυπλοκότητα $O(n)$. Φυσικά, on average και τα τρία αυτά έχουν πολυπλοκότητα $O(1)$ το οποίο είναι και ο λόγος για τον οποίον επέλεξα τη συγκεκριμένη δομή.

Σε ό,τι αφορά τη hash function με την οποία αποφασίζεται το bucket στο οποίο θα τοποθετηθεί ο κάθε κόμβος πειραματίστηκα με τις συναρτήσεις που βρήκα εδώ και εδώ οι οποίες προορίζονται ακριβώς για χρήση με τυχαία αλφαριθμητικά strings, όπως είναι εδώ τα ονόματα των κόμβων. Χρησιμοποιώντας *INITIAL_BUCKETS_NUM=16* και *HASHING_LOAD_FACTOR=0.7* το πλήθος collisions για τα ενδεικτικά αρχεία που μας δόθηκαν εμφανίζεται στον παρακάτω πίνακα:

	djb2	*sdbm*	*fnv-1 32-bit*	*fnv-1 64-bit*
WriteUp	1	5	4	4
Small	5	5	3	3
Medium	21	17	20	18
Big	205	202	204	217

Αν και, όπως φαίνεται, οι αποκλίσεις μεταξύ των συναρτήσεων είναι μικρές, επέλεξα να χρησιμοποιήσω την *sdbm* για βέλτιστη ελαχιστοποίηση collisions σε μεσαίους και μεγαλύτερους γράφους. Για κάθε περίπτωση πάντως, κράτησα και τις υπόλοιπες συναρτήσεις στο *util.cpp* .

❖ Edges:

Σε κάθε κόμβο επέλεξα να υλοποιήσω μια **απλά συνδεδεμένη λίστα** στην οποία αποθηκεύονται οι **εξερχόμενες** ακμές αυτού. Κάθε τέτοια λίστα διατηρείται ταξινομημένη (ως προς το όνομα του κόμβου που καταλήγει η κάθε ακμή) καθ' όλη τη διάρκεια του προγράμματος. Το γεγονός αυτό, αν και δίνει worst-case πολυπλοκότητα $O(n)$ σε αναζήτηση, εισαγωγή και διαγραφή ακμής [σε αντίθεση, π.χ., με το να προστίθεται κάθε καινούργια ακμή στο τέλος της λίστας που θα έδινε $O(1)$ τουλάχιστον στην εισαγωγή], έχει εμφανώς καλύτερα αποτελέσματα on average.

Operations 7-9:

Για την εύρεση κύκλων και στα 3 αυτά operations έχει γίνει χρήση DFS, για τον οποίο υλοποίησα επιπροσθέτως μια δομή στοίβας. Τόσο οι κόμβοι όσο και οι ακμές περιέχουν ένα boolean πεδίο *visited* ώστε να εντοπίζεται αποδοτικά το κατά πόσο τον/την έχουμε ήδη επισκεφτεί. Οι συνθήκες τερματισμού για τον DFS σε κάθε περίπτωση, ξεκινώντας πάντα από τον κόμβο *Ni*, έχουν ως εξής:

- *c(irclefind) Ni*: Επίσκεψη ήδη visited κόμβου. Αν ο κόμβος αυτός είναι ο *Ni* τότε βρήκαμε ζητούμενο κύκλο και τον τυπώνουμε, ενώ αν είναι άλλος τότε πέσαμε σε κύκλο που δεν περιέχει τον *Ni* και άρα εγκαταλείπουμε το συγκεκριμένο μονοπάτι.
- *f(indcircles) Ni k*: Επίσκεψη ήδη visited ακμής. Αν η ακμή αυτή καταλήγει στον *Ni* τότε βρήκαμε ζητούμενο κύκλο και τον τυπώνουμε, ενώ αν όχι τότε πέσαμε σε κύκλο που δεν περιέχει τον *Ni* και άρα εγκαταλείπουμε το συγκεκριμένο μονοπάτι.
- *t(raceflow) Ni Nj l*: Επίσκεψη ήδη visited ακμής. Αν η ακμή αυτή καταλήγει στον *Nj* τότε βρήκαμε ζητούμενο flow και το τυπώνουμε, ενώ αν όχι τότε πέσαμε σε άσκοπο κύκλο και άρα εγκαταλείπουμε το συγκεκριμένο μονοπάτι.

Πρόσθετες Λειτουργίες:

- Εντολή *"h(elp)"*: Εκτυπώνει στο τερματικό τις διαθέσιμες εντολές του προγράμματος καθώς και τον τρόπο με τον οποίο αυτές συντάσσονται.
- Εντολή *"p(rint)"*: Εκτυπώνει στο τερματικό τον γράφο όπως ακριβώς θα εκτυπώνονταν στο outputfile εάν τερματίζαμε το πρόγραμμα εκείνη τη στιγμή.
- *SHOW_EXTRA_UI*: Θέλοντας να είναι εύχρηστο το πρόγραμμα, είχα ενσωματώσει ένα στοιχειώδες user interface, το οποίο περιλάμβανε μηνύματα όπως "Loaded graph from inputfile successfully!" και promt "Type a command:". Προκειμένου όμως το output του προγράμματος να ταυτίζεται *ακριβώς* (εκτός βέβαια της σειράς εκτύπωσης ορισμένων ερωτημάτων, καθώς αυτή εξαρτάται από την υλοποίηση) με τα δοθέντα ενδεικτικά, απέκρυψα το interface αυτό μέσω της σταθερά *SHOW_EXTRA_UI*. Μπορεί να εμφανιστεί ξανά θέτοντάς την σε *true* στο *util.h* .
- *ALLOW_NEGATIVE_WEIGHTS*: Δεδομένου ότι το πρόγραμμα αφορά τραπεζικές δοσοληψίες, είναι λογικό να μην επιτρέπονται αρνητικές τιμές για τα βάρη των ακμών. Ωστόσο, για λόγους επεκτασιμότητας η υλοποίησή μου δεν βασίζεται σε αυτό το γεγονός και παρέχεται δυνατότητα υποστήριξης και αρνητικών τιμών αλλάζοντας τη σταθερά *ALLOW_NEGATIVE_WEIGHTS* σε *true* στο *util.h* .