2025

Δυναμική Δισυνεκτικότητα Γραφημάτων

Γρύλλιας Γεράσιμος 1084651

Υλοποίηση Δομής Bridge-Block-Forest (BBF)

Αρχείο bridge-blocks.cpp:

- maketree(char c):

Η maketree() δημιουργεί έναν νέο κόμβο στο γράφημα, του ορίζει γονέα το nil (δεν έχει γονέα) και αν ο δοσμένος χαρακτήρας δεν είναι '\0', τότε αποθηκεύεται ως ετικέτα του κόμβου. Τέλος, επιστρέφει τον κόμβο.

- evert(node n):

Η evert() παίρνει έναν κόμβο και αντιστρέφει τους δείκτες γονέων από αυτόν και προς τα πάνω. Ξεκινώντας από τον κόμβο n, αλλάζει κάθε φορά τον γονέα του ώστε να δείχνει στον προηγούμενο κόμβο, μέχρι να φτάσει στη ρίζα, με αποτέλεσμα το δέντρο να «γυρίζει ανάποδα» και ο n να γίνεται νέα ρίζα.

- link(node x, node y):

Η link() συνδέει τον κόμβο x με τον κόμβο y δημιουργώντας μια ακμή στο γράφημα και θέτοντας τον y ως γονέα του x, οπότε ο x «κρεμιέται» κάτω από τον y.

- findpath(node u, node v):

Η findpath() βρίσκει και επιστρέφει τη διαδρομή από τον κόμβο u στον κόμβο v. Συγκεκριμένα, κατασκευάζει τις διαδρομές από το u προς τη ρίζα και από το v προς τη ρίζα. Ύστερα, τις αντιστρέφει ώστε να ξεκινούν από τη ρίζα, εντοπίζει τον κοινό πρόγονο, συνθέτει το μονοπάτι από τον u έως τον κοινό πρόγονο, και από εκεί έως το v. Τέλος, αφαιρεί τα άκρα (u, v) ώστε να μείνουν μόνο οι ενδιάμεσοι κόμβοι της διαδρομής.

- condensepath(vector<node> condpath, char c):

Η condensepath() παίρνει μια διαδρομή κόμβων (από την findpath()) και έναν χαρακτήρα, δημιουργεί έναν νέο κόμβο με αυτόν τον χαρακτήρα σαν Label, συνδέει όλα τα παιδιά των κόμβων της διαδρομής στον νέο κόμβο και μετά διαγράφει τους παλιούς κόμβους, δηλαδή συμπτύσσει τη διαδρομή σε έναν κόμβο.

Bridge-Block Operations:

- make_vertex(char ch)

Η make_vertex() δημιουργεί έναν νέο κόμβο χωρίς ετικέτα (square node), συνδέει σε αυτόν έναν κόμβο με τον χαρακτήρα ch (round node) και επιστρέφει τον πρώτο κόμβο, δηλαδή φτιάχνει έναν κόμβο τύπου square, που έχει σαν πατέρα έναν κόμβο τύπου round με την ετικέτα ch. Με λίγα λόγια, φτιάχνει ένα bridge-block tree, που αποτελείται από δύο κόμβους: round node με παιδί το square node.

- find_block(node u):

Η find_block() παίρνει έναν square κόμβο u, βρίσκει τη διαδρομή από τον u στον εαυτό του (δηλαδή τον πατέρα του) και επιστρέφει την ετικέτα label του πατέρα του, δηλαδή το bridge-block στο οποίο ανήκει.

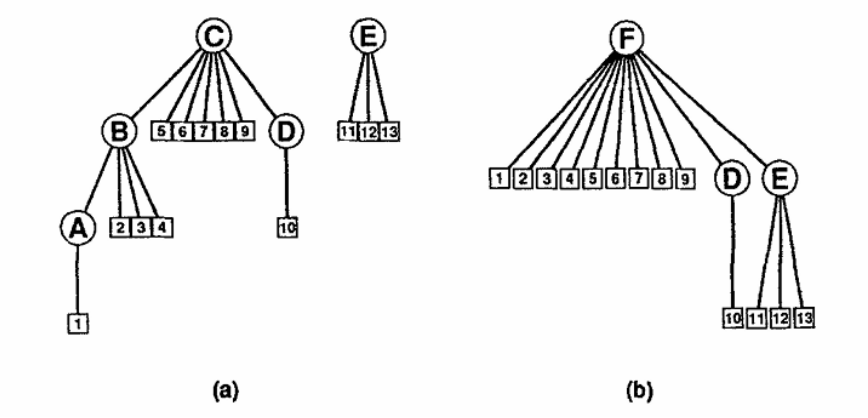
- insert_edge(node u, node v, char c):

Η insert_edge() προσθέτει «ακμή» μεταξύ u και v: αν είναι σε διαφορετικά components, τα συνδέει σωστά ανάλογα με το μέγεθός τους, και τώρα τα u, v βρίσκονται στο ίδιο bridge-block-forest (BBF). Αν είναι στο ίδιο, συμπτύσσει τη διαδρομή μεταξύ τους με τη βοήθεια της condensepath() σε έναν νέο round κόμβο με τον δοσμένο χαρακτήρα c. Σε αυτή την περίπτωση, τώρα τα u,v βρίσκονται στο ίδιο bridge-block.

Συνάρτηση main():

Σε ένα μη κατευθυνόμενο, αρχικά άδειο, γράφημα, ξεκινάμε και σταδιακά εισάγουμε ακμές με την make_vertex(). Δηλαδή φτιάχνουμε ξεχωριστά bridge-block trees. Έπειτα, προσθέτουμε ακμές με την insert_edge() και δημιουργούμε bridge-block forest (BBF). Κάθε φορά που κάνουμε τέτοιες αλλαγές με την insert_edge(), μπορούμε να εκτελούμε και να αξιολογούμε ερωτήματα find_block() για όποιο square node θέλουμε (ή και τυχαία).

Για το 1ο πείραμα αξιολόγησης επιλέχθηκε το παράδειγμα BBF του άρθρου (Fig. 2):



Υλοποίηση Δομής Block-Forest (BF)

Αρχείο blocks.cpp:

Η νέα δομή έχει παρόμοιο βασικό κώδικα με την προηγούμενη, καθώς διατηρεί γενικά τις ίδιες βασικές λειτουργίες για δημιουργία κόμβων, σύνδεση, εύρεση διαδρομών και εισαγωγή ακμών. Ωστόσο, λόγω της διαφορετικής φύσης του Block-Forest (εδώ ένας square κόμβος μπορεί να είναι και πατέρας ενός round), υπάρχουν μερικές σημαντικές διαφορές:

1. Η make_vertex() τώρα δημιουργεί απλά ένα square node χωρίς γονέα (nil).
2. Η find_block() δουλεύει με δύο κόμβους και επιστρέφει το μοναδικό round node που παριστάνει το block που ανήκουν αυτοί οι δύο κόμβοι (αν δεν ανήκουν σε κοινό block επιστρέφει '\0').
3. η insert_edge() εκτός από το να κάνει condense path, είναι ικανή να δημιουργεί νέο round node για να ενώσει διαφορετικά components αντί να συνδέει απευθείας τους υπάρχοντες κόμβους.

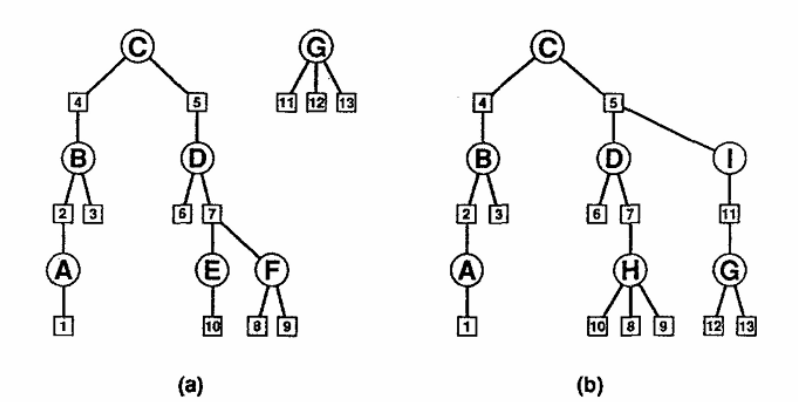
Οι αλλαγές αυτές διατηρούν τη βασική λογική, αλλά βελτιώνουν τον χειρισμό των blocks και των διαδρομών.

Συνάρτηση main():

Η main() παραμένει παρόμοια με την προηγούμενη έκδοση: δημιουργεί έναν άδειο μη κατευθυνόμενο γράφο, φτιάχνει ξεχωριστούς square nodes και σχηματίζει τα bridge-blocks χρησιμοποιώντας την insert_edge(), συμπτύσσοντας διαδρομές όπου χρειάζεται. Στη συνέχεια, εκτυπώνει τον χαρακτήρα του block κάθε ζεύγους κόμβων πριν και μετά

από επιπλέον εισαγωγές ακμών, ώστε να δείξει πώς εξελίσσεται η δομή του Bridge Block Forest με βάση τις αλλαγές.

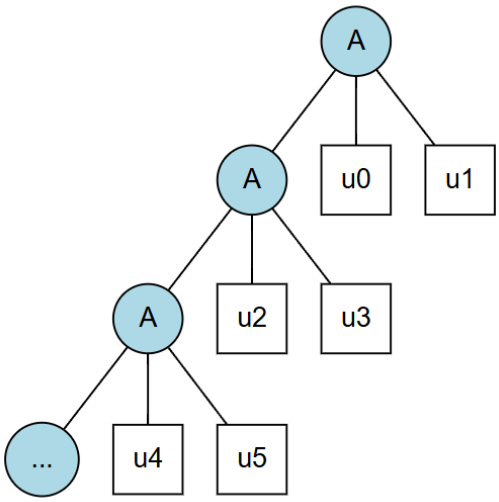
Και σε αυτή τη δομή, για το πείραμα επιλέχθηκε το παράδειγμα BF του άρθρου:



Πειραματική Αξιολόγηση

Για την πειραματική αξιολόγηση, αρχικά δημιουργήθηκαν γραφήματα μεγάλου μεγέθους. Συγκεκριμένα, χρησιμοποιήθηκαν επαναληπτικοί βρόχοι (for loops) ώστε να εκτελείται πολλές φορές η συνάρτηση make_vertex και insert_edge, με σκοπό τη δημιουργία του BBF. Το τελικό γράφημα που προέκυψε είναι συνεκτικό και η μέτρηση του χρόνου εκτελέστηκε αποκλειστικά για τη συνάρτηση find_block, για κάθε node, η οποία εντοπίζει το bridge-block του κόμβου.

Πιο συγκεκριμένα, το BBF θα έχει την εξής μορφή:



Γραφήματα με n = 1000 square nodes

n=1000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	49 ms	50 ms	48 ms	57 ms	52 ms

Μέσος Όρος find_block(): 51.2 ms

Γραφήματα με n = 5000 square nodes

n=5000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	252 ms	234 ms	329 ms	242 ms	229 ms

Μέσος Όρος find_block(): 257.2 ms

Γραφήματα με n = 8000 square nodes

n=8000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	379 ms	381 ms	399 ms	391 ms	384 ms

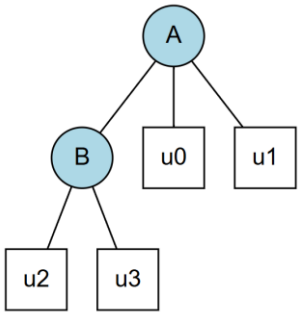
Μέσος Όρος find_block(): 386.8 ms

Παρατήρηση χρονικών αποτελεσμάτων:

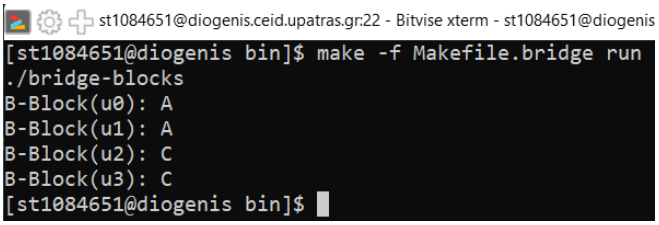
Όταν εκτελέστηκε ο αλγόριθμος για διαφορετικά μεγέθη γραφημάτων, τα αποτελέσματα έδειξαν μια σχεδόν γραμμική αύξηση του χρόνου εκτέλεσης σε σχέση με τον αριθμό κορυφών. Συγκεκριμένα, για 1000 κορυφές ο χρόνος ήταν 51,2 ms, για 5000 κορυφές 257,2 ms και για 8000 κορυφές 386,8 ms. Παρατηρείται ότι ο χρόνος αυξάνεται περίπου ανάλογα με το μέγεθος του γραφήματος. Από τις 1000 στις 5000 κορυφές, όπου το πλήθος αυξάνεται πέντε φορές, ο χρόνος εκτέλεσης επίσης πενταπλασιάζεται, ενώ από τις 5000 στις 8000 κορυφές η αύξηση είναι περίπου 1,5 φορά. Αυτό δείχνει ότι ο αλγόριθμος έχει πολυπλοκότητα κοντά στο O(n) γεγονός που εξηγεί την καλή κλιμάκωση. Επιπλέον, οι απόλυτοι χρόνοι παραμένουν μικροί (κάτω από μισό δευτερόλεπτο για 8000 κορυφές), γεγονός που υποδηλώνει ότι η υλοποίηση είναι αρκετά αποδοτική για πρακτική χρήση.

Το επόμενο στάδιο της πειραματικής αξιολόγησης είναι η «δοκιμή» της δομής BBF σε συγκεκριμένα σενάρια σχεδιασμένα για «άγνωστες» καταστάσεις:

~ *Σενάριο 1:* *Σε ένα γράφημα BBF, δύο κορυφές βρίσκονται στο ίδιο bridge-block, και τις συνδέουμε με την insert_edge. Για παράδειγμα, στο παρακάτω γράφημα, δοκιμάζουμε να κάνουμε insert_edge(u2, u3, 'C') :*

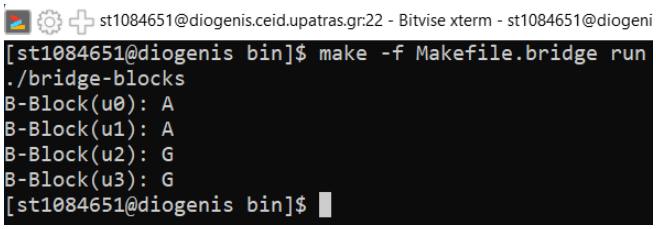


Το αποτέλεσμα σε αυτή την περίπτωση είναι ότι ο round node 'B' θα αντικατασταθεί με τον round node 'C' και πλέον τα bridge-blocks κάθε square node θα είναι:



Το ίδιο θα συμβεί και αν εκτελέσουμε την ίδια εντολή παραπάνω από μία φορά.

~ *Σενάριο 2:* *Σε ένα ίδιο γράφημα με το παραπάνω, θα δοκιμάσουμε τώρα να δημιουργήσουμε ένα self-loop με την insert_edge(u3, u3, 'G') και να αιτιολογήσουμε την ανταπόκριση του αλγορίθμου:*

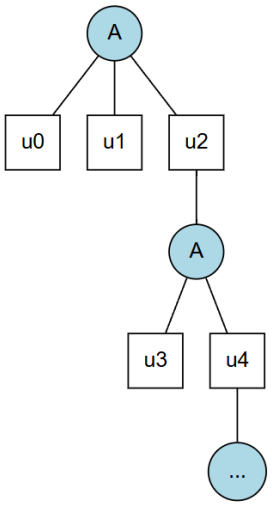


Αιτιολόγηση: Όπως φαίνεται και από την εκτύπωση, τώρα το bridge-block που ανήκει το u3 είναι το G, όπως επίσης και εκείνα τα square nodes (εδώ το u2) που ανήκαν στο ίδιο bridge-block με το u3. Σύμφωνα με τον αλγόριθμο, επειδή το u3 με τον εαυτό του πρακτικά ανήκουν στο ίδιο component, η insert_edge(u3, u3, 'G') θα εκτελέσει findpath(u3, u3) και θα επιστρέψει τον πατέρα του u3. Ύστερα θα εκτελεστεί η condensepath με όρισμα τον πατέρα του u3 και το label 'G'. Οπότε, θα δημιουργηθεί ένα

νέο round node με label 'G', και θα αντικαταστήσει το παλιό round node 'B'. Επομένως, η εντολή αυτή στη ουσία είναι ταυτόσημη με την insert_edge(u2, u3, 'G').

Blocks problem

Με την ίδια λογική που μετρήσαμε το χρόνο εκτέλεσης της συνάρτησης find_block στο πρόβλημα των bridge-blocks, έτσι και εδώ, χρησιμοποιήθηκαν επαναληπτικοί βρόχοι (for loops) ώστε να εκτελείται πολλές φορές η συνάρτηση make_vertex και insert_edge, με σκοπό τη δημιουργία του Block Forest (BF). Το τελικό γράφημα που προέκυψε είναι συνεκτικό και η μέτρηση του χρόνου εκτελέστηκε αποκλειστικά για τη συνάρτηση find_block, η οποία τώρα παίρνει σαν όρισμα κάθε ζεύγος από square nodes και εντοπίζει το κοινό τους block. Το γράφημα που δημιουργείται έχει την εξής μορφή:



Γραφήματα με n = 1000 square nodes

n=1000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	7674 ms	7612 ms	7600 ms	7561 ms	7653 ms

Μέσος Όρος find_block(): 7620ms

Γραφήματα με n = 5000 square nodes

n=5000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	168918ms	178899ms	179904ms	180338ms	180453ms

Μέσος Όρος find_block(): 177702.4ms

Γραφήματα με n = 8000 square nodes

n=8000	Run 1	Run 2	Run 3	Run 4	Run 5
find_block() time:	463072ms	466094ms	466246ms	467850ms	468383ms

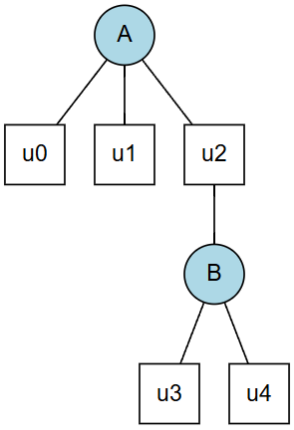
Μέσος Όρος find_block(): 466329ms

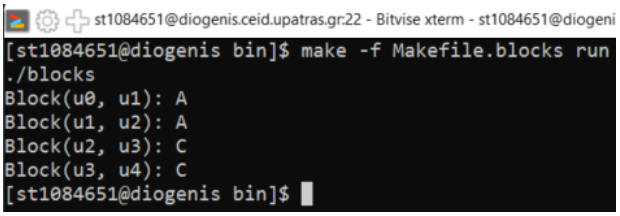
Παρατήρηση χρονικών αποτελεσμάτων:

Στο πρόβλημα των blocks, οι χρόνοι εκτέλεσης ήταν σημαντικά μεγαλύτεροι σε σχέση με τον προηγούμενο αλγόριθμο. Συγκεκριμένα, για 1000 κορυφές ο χρόνος ήταν 7620 ms, για 5000 κορυφές 177.702,4 ms, ενώ για 8000 κορυφές έφτασε τις 466.329 ms. Παρατηρείται ότι η αύξηση του χρόνου είναι πολύ πιο απότομη: από τις 1000 στις 5000 κορυφές ο χρόνος εκτέλεσης αυξάνεται περίπου 23 φορές, ενώ από τις 5000 στις 8000 κορυφές αυξάνεται σχεδόν 2,6 φορές. Αυτό υποδηλώνει ότι ο αλγόριθμος για το πρόβλημα των blocks έχει σαφώς μεγαλύτερη πολυπλοκότητα και επομένως δεν κλιμακώνεται το ίδιο καλά με τον προηγούμενο αλγόριθμο.

Το τελευταίο στάδιο της πειραματικής αξιολόγησης είναι η «δοκιμή» της δομής BF σε συγκεκριμένα σενάρια σχεδιασμένα για «άγνωστες» καταστάσεις:

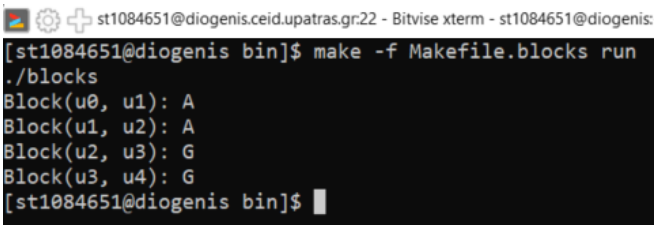
~ *Σενάριο 1:* *Σε ένα γράφημα BF, δύο κορυφές βρίσκονται στο ίδιο block, και τις συνδέουμε με την insert_edge. Ας υποθέσουμε ότι στο παρακάτω γράφημα, δοκιμάζουμε να κάνουμε insert_edge(u3, u4, 'C') :*





Παρατηρούμε ότι, όπως και στην περίπτωση της δομής BBF, έτσι κι εδώ, ο round node B αντικαταστάθηκε από τον round node C χωρίς αλλοίωση της πληροφορίας. Το ίδιο συμβαίνει και στην περίπτωση insert_edge(u2, u3, 'C') που οι u2 είναι articulation point ενώ ο u3 όχι.

~ *Σενάριο 2:* *Σε ένα ίδιο γράφημα με το παραπάνω, θα δοκιμάσουμε τώρα να δημιουργήσουμε όπως και στην BBF ένα self-loop με την insert_edge(u3, u3, 'G') και να αιτιολογήσουμε την ανταπόκριση του αλγορίθμου:*



Όπως ήταν αναμενόμενο, ομοίως και με το BBF, το "self-loop" ορθά δεν λειτουργεί με την βασική έννοια του self-loop δημιουργώντας νέα ακμή από το u3 στον εαυτό του, αλλά αλλάζει/μετονομάζει τον round node που ανήκει, χωρίς να αλλάζει το BF και τη συνολική πληροφορία.

Συμπέρασμα:

Συνοψίζοντας, από τα πειραματικά αποτελέσματα προκύπτει ότι τα προβλήματα των bridge blocks και των blocks έχουν παρόμοια δομή και παρουσιάζουν παρόμοια συμπεριφορά ως προς την αύξηση του χρόνου εκτέλεσης με το μέγεθος του γραφήματος. Ωστόσο, παρατηρείται ότι το πρόβλημα των blocks είναι σημαντικά πιο απαιτητικό υπολογιστικά, με τους χρόνους εκτέλεσης να αυξάνονται πολύ εντονότερα σε σχέση με το bridge blocks, γεγονός που υποδηλώνει υψηλότερο υπολογιστικό κόστος για μεγάλα γραφήματα.