Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα
"Travelling Salesman Problem"

Αναγνώστου Στέφανος - 1115201100024
Μαντζάνα Ελένη - 1115201300091

Δομές Δεδομένων

Όλες οι δομές δεδομένων που χρησιμοποιούνται στην άσκηση είναι υλοποιημένες με πίνακες. Οποιαδήποτε άλλου είδους μορφή υλοποίησης (π.χ. με κόμβους) καθιστά αργή την εκτέλεση του προγράμματος. Η δυναμικότητά τους βασίζεται στη λογική του διπλασιασμού του μεγέθους τους. Άλλο ένα βασικό χαρακτηριστικό τους είναι η συνάρτηση clear() που υλοποιούν όλες οι δομές, η οποία «καθαρίζει» τη δομή, απλά μηδενίζοντας και αρχικοποιώντας τις μεταβλητές, χωρίς όμως να αποδεσμεύει τη μέχρι τώρα δεσμευμένη μνήμη, για μικρότερη χρονική πολυπλοκότητα. Με τη χρήση templates της C++, οι δομές επαναχρησιμοποιούνται όταν χρειάζονται σε διαφορετικά κομμάτια της άσκησης.

Ιδανικά, οι δομές όσο μεγαλύτερες είναι, τόσο καλύτερο για τα μεγάλα αρχεία. Επίσης, μετά από δοκιμές καταλήξαμε για τα list_node πως το κατάλληλο N είναι το 15. Ωστόσο, για τα μικρά (small, tiny) επειδή ο χρόνος εκτέλεσής τους είναι ελάχιστος, η προσπάθεια εύρεσης μεγάλου κομματιού μνήμης από τον allocator καθυστερεί τις εκτελέσεις τους. Οι βασικές δομές που έχουν υλοποιηθεί, εκτός από αυτές τις εκφώνησης, είναι οι: Stack, Queue, HashTable, Explored (index για τους επισκεπτόμενους κόμβους).

Εισαγωγή

Στην εισαγωγή χρησιμοποιείται ένα HashTable αρκετά μεγάλου μεγέθους για να χωρέσει όλες τις ακμές και η αναζήτηση να τείνει σε πολυπλοκότητα O(1). Αυτό χρειάζεται για να γίνεται εύκολος και γρήγορος έλεγχος για διπλότυπα πριν την εισαγωγή κάθε ακμής και έτσι να μπορεί η κάθε νέα ακμή να εισάγεται μέσα στο Buffer σε χρόνο O(1). Για την άμεση εισαγωγή στο Buffer, το NodeIndex κρατάει το τελευταίο διαθέσιμο list_node του κόμβου, και έτσι επιτυγχάνεται εισαγωγή σε O(1).

Bidirectional BFS

Η αναζήτηση χρησιμοποιεί για το κάθε thread δύο ουρές και δύο Explored δομές (αρχικά ήταν HashTable αλλά το Explored εξασφαλίζει O(1)), από ένα ζευγάρι για

τον καθεμία από τις δύο ταυτόχρονες αναζητήσεις που γίνονται. Η ευρετική που αποφασίζει για το ποιά «μεριά» της αναζήτησης θα επεκταθεί κάθε φορά είναι το άθροισμα των γειτόνων όλων των κόμβων που υπάρχουν τη δεδομένη στιγμή μέσα στην ουρά. Ουσιαστικά θα μπορούσε κανείς να πει πως δεν κοιτάμε τα «παιδιά», αλλά τα «εγγόνια» του κάθε συνόρου.

Στατικό - SCC - Grail Index

Αρκετές από τις συναρτήσεις που δόθηκαν στην εκφώνηση για τη δημιουργία των Strongly Connected Components υλοποιήθηκαν αλλά δεν χρησιμοποιήθηκαν, όπως π.χ. το Cursor. Μετά τη δημιουργία των SCC με τη βοήθεια του αλγορίθμου του Tarjan, η δομή SCC δημιουργεί έναν Υπεργράφο με τη χρήση των δομών των κανονικών γράφων (NodeIndex, Buffer). Το πλήθος τον Grail Indexes που θα δημιουργηθούν ορίζεται με define. Το ιδανικό πλήθος τους είναι το 5, αφού με περισσότερα χάνεται περισσότερος χρόνος στο medium που δεν βοηθά στην ουσία. Για το κάθε Grail ανακατεύονται και οι κόμβοι αλλά και οι γείτονες τους με τον αλγόριθμο των Fisher – Yates. Εφόσον οι κόμβοι είναι ανακατεμένοι, για να υπάρχει γρήγορη πρόσβαση (O(1)) σε αυτούς χρησιμοποιούνται Inverted Indexes. Η ερώτηση και στα 5 Grail μας δίνει περισσότερες πιθανότητες να πάρουμε απάντηση «No» και να απορρίψουμε τον κόμβο. Κατά την εκτέλεση του Query, για κάθε κόμβο που μπαίνει στην ουρά γίνεται ερώτηση στα Grail για το αν «φτάνει» τον κόμβο στόχο και αν η απάντηση είναι όχι, τότε απορρίπτεται. Επίσης, αν κάποιος κόμβος ανήκει στο ίδιο SCC με τον στόχο, τότε από αυτόν τον κόμβο και για όλα τα παιδία του απορρίπτονται όλοι οι κόμβοι που δεν ανήκουν στο ίδιο SCC. Ωστόσο, στη δική μας περίπτωση δεν φαίνεται μεγάλη η διαφορά, καθώς τα SCC ακόμα και στο large αποτελούνται από το πολύ 2 κόμβους τα περισσότερα.

Δυναμικό - CC - Update Index

Η δημιουργία των Connected Components γίνεται με BFS. Το Update Index αποτελείται από δύο δομές. Η πρώτη είναι ένα index με όλα τα CC όπου το κάθε κελί δείχνει σε έναν πίνακα με τους γείτονες αυτού. Η δεύτερη είναι ένα απλό HashTable που κρατάει όλους τους δυνατούς συνδυασμούς μεταξύ των CC, ώστε τα ερωτήματα στο UpdateIndex να απαντώνται σε σταθερό χρόνο. Κάθε φορά που γίνεται κάποια εισαγωγή μιας ακμής όλοι οι γείτονες αυτής αντιγράφονται σε όλους τους γείτονες της άλλης και συνεπώς όλοι οι νέοι δυνατοί συνδυασμοί που δημιουργήθηκαν προστίθενται στο HashTable.

Η ανακτασκευή (rebuild) των CC γίνεται αδειάζοντας τη δομή UpdateIndex και κάνοντας από την αρχή BFS στους κόμβους, όπως την πρώτη φορά που δημιουργήθηκε. Η κατάλληλη τιμή για τη μετρική που αποφασίζει πότε θα γίνει ανακατασκευή των CC ορίστηκε μετά από πολλές δοκιμές στο 0.52, το οποίο μας προσφέρει μερικά rebuilds στο medium και κανένα (!) στο large dataset.

JobScheduler - Πολυνηματισμός

Ο JobScheduler δημιουργήθηκε στα πρότυπα της εκφώνησης. Είναι αφηρημένος, που σημαίνει πως εκτελεί Jobs, τα οποία είναι υποκλάσης της κλάσης Job (στην πε-

ρίπτωσή μας StaticQueryJob και DynamicQueryJob) οι οποίες υλοποιούν συγκεκριμένες συναρτήσεις που καλούνται μέσα στον JobScheduler. Ο πολυμορφισμός που προσφέρουν οι κλάσεις της C++ βοήθησε πολύ σε αυτό το κομμάτι.

Το πλήθος των νημάτων δίνεται από τη γραμμή εντολών κατά την εκκίνηση του προγράμματος. Ο γράφος δημιουργεί αντίστοιχα το πλήθος των ουρών και explored δομών που θα χρειαστεί το κάθε thread. Επίσης, το πόσα Jobs θα παίρνει κάθε φορά το κάθε thread για να εκτελέσει δίνεται με define, ωστόσο μετά από πολλές δοκιμές καταλήξαμε πως δεν έχει ιδιαίτερη σημασία, αφού είτε 1 Job τη φορά είτε 500, η απόδοση του προγράμματος παραμένει ίδια. Αυτό συμβαίνει διότι όπως φαίνεται το χρονικό κόστος του συγχρονισμού είναι σχεδόν αμελητέο.

Versioning και η περίπτωση που συμφέρει να «χάνεται»

Για το δυναμικό κομμάτι της άσκησης η δομή Buffer κρατάει για κάθε ακμή και ποιά είναι η έκδοση από την οποία και μετά η συγκεκριμένη ακμή υπάρχει. Εκτός όμως από εκεί, οι εκδόσεις κρατιούνται και στο UpdateIndex των CC, και πιο συγκεκριμένα στη δομή του HashTable. Έτσι λοιπόν και εκεί ελέγχουμε την έκδοση στην οποία ενώθηκαν τα δύο CC. Ωστόσο, στην υλοποίηση μας «χάνεται» μία περίπτωση. Αυτή είναι η περίπτωση που δύο νέοι κόμβοι προστίθενται στο CC και άρα δε χρειάζεται να χρησιμοποιηθεί το Update Index, αλλά παίρνουν κατευθείαν τιμή στο CC index. Σε αυτήν την περίπτωση η ερώτηση που θα γίνει θα επιστρέψει «Maybe», ενώ στην πραγματικότητα η απάντηση είναι αρνητική. Εφόσον δεν μας συμφέρει να κρατάμε εκδόσεις και στο CC index, η προσέγγισή μας σε αυτό το κομμάτι ήταν να προσθέσουμε μια ακόμα μεταβλητή στο NodeIndex για κάθε κόμβο, η οποία κρατάει την έκδοση στην οποία πρωτοεμφανίστηκε ο κόμβος και συνεπώς έχει γείτονες μεγαλύτερους ή ίσους με αυτή την έκδοση. Με αυτόν τον τρόπο πριν την ερώτηση στο CC θα μπορούσαμε να ξέρουμε αν υπήρχε στη συγκεκριμένη έκδοση ο κόμβος και να μη χρειαστεί να ρωτήσουμε καν τα components. Η μέθοδος αυτή όμως, πιθανώς λόγω cache, έχει μεγάλη επίπτωση στην απόδοση του προγράμματος σε σχέση με την περίπτωση του να παίρνουμε λίγα περισσότερα «Maybe» τα οποία ούτος η άλλως θα επιστρέψουν -1 μετά το πρώτο expand, αφού δε θα βρεθούν γείτονες.

Επιπλέον Βελτιστοποιήσεις

- Στην περίπτωσή μας η μετρική αποφασίζει να κάνει rebuild στο medium μετά το τελευταίο Batch από queries, το οποίο δεν έχει ιδιαίτερο νόημα αφού το πρόγραμμά μας αμέσως μετά τερματίζει. Για αυτό το λόγο, ελέγχεται αν βρισκόμαστε στο τελευταίο Batch ώστε να αποφευχθεί η ανακατασκευή των CC.

- Αντικαταστάθηκε το cout με printf και το διάβασμα των αρχείων από istream σε fscanf, κάτι που μας γλίτωσε γύρω στο 1 λεπτό! Λογικό αφού τα προηγούμενα πρέπει να «μαντέψουν» και να μετατρέψουν τον τύπο της μεταβλητής για να διαβαστεί ή να εκτυπωθεί σωστά, ενώ στις printf και fscanf ορίζεται από εμάς ο τύπος της μεταβλητής.

- Στο HashTable η συνάρτηση της εισαγωγής δεν ελέγχει πλέον για διπλότυπα, αλλά προσθέτει τον κόμβο στο τέλος του Bucket κατευθείαν. Πολλές φορές στην άσκηση χρειάστηκε να εισάγουμε στοιχείο που ήμασταν σίγουροι πως δεν υπήρχε ήδη στο HashTable, άρα είναι πιο αποδοτικό. Για όλες τις άλλες περιπτώσεις χρησιμοποιούμε πρώτα την find.

Compile – Run

Για το compile του προγράμματος χρησιμοποιείται η εντολή make. Για την εκτέλεση η εντολή:

./tsp [plithos_threads] [arxikos_grafos.txt] [workload.txt]

Για το unit testing η μεταγλώττιση γίνεται με την εντολή make unit. Χρησιμοποιείται η βιβλιοθήκη cunit. Για την εκτέλεση η εντολή:

./unitTesting

Μετρήσεις Απόδοσης

Για τις παρακάτω μετρήσεις πραγματοποιήθηκαν πολλές εκτελέσεις του προγράμματος και ο χρόνος που γράφεται είναι περίπου ο μέσος από όλες τις μετρήσεις. Χρησιμοποιήθηκε το εξής μηχάνημα: CPU: Intel i5-3317U @ 1,70 GHz, RAM: 4 GB, OS: Ubuntu 16.04 64bit, SSD 128 GB.

	Part 1	Part 2	Part 3
Tiny	1sec	<1sec	<1sec
Small	9sec	<1sec	<1sec
Medium Static	-	5.6sec	**6.2sec**
Medium Dynamic	2min 23sec	5.6sec	**5.9sec**
Large Static	-	2min 54sec	**1min 35sec**
Large Dynamic	-	14min 30sec	**8min 33sec**

Σχήμα 1: Χρονικές Μετρήσεις

	Static	Dynamic
Medium	1,7 GB	1,5 GB
Large	3,7 GB	2,2 GB

Σχήμα 2: Χωρικές Μετρήσεις στο Part 3