Λιούμη Κυριακή

ΑΜ 1067410

Τεχνολογίες Υλοποίησης Αλγορίθμων

Τελική Εργασία

Dial’s Algorithm

Radix-Heap in Dijkstra

Θέμα:

Υλοποίηση του αλγορίθμου Dijkstra κατά Dial [AMO93] και με χρήση radix-heap [AMO93,AMOT90], καθώς και πειραματική αξιολόγησή τους με τις αντίστοιχες υλοποιήσεις στην LEDA σε τυχαία και μη-τυχαία γραφήματα.

[AMO93] R. Ahuja, T. Magnanti, J. Orlin, "Network Flows", Prentice-Hall, 1993. Κεφ. 4.
[AMOT90] R. Ahuja, K. Mehlhorn, J. Orlin, and R.E. Tarjan, Faster algorithms for the shortest path problem, *Journal of the ACM*, 37(2):213-222, 1990.

Ανάλυση Κώδικα:

Αρχείο dial.cpp:

Ο κώδικας αποτελείται από 3 συναρτήσεις. Αρχικά, η συνάρτηση:

```
void addEdge(vector <pair<int, int> > adj[], int u, int v, int wt)
```

Στην οποία δημιουργούμε ακμές και τις ενώνουμε με τις κορυφές και προς τις δύο κατευθύνσεις, ώστε να μην είναι το γράφημα μας κατευθυνόμενο. Ακόμα, σε κάθε ακμή προσθέτουμε και το βάρος της.

Η δεύτερη συνάρτηση αφορά στον αλγόριθμο του Dial.

```
void Dial(int src, int W, int V,vector <pair<int, int> > adj[] ){
```

Αρχικά, δημιουργούμε τη λίστα dist, η οποία αποθηκεύει τις αποστάσεις μεταξύ των κόμβων. Αρχικοποιούμε όλες τις τιμές στη λίστα αυτή με το μηδέν. Δημιουργούμε τα buckets με την μορφή λίστας και εισάγουμε την κορυφή πηγή στο πρώτο bucket, όπως και στον πίνακα dist. Ύστερα, ψάχνουμε για κάποιο bucket, το οποίο δε θα είναι άδειο. Αν όλα τα bucket είναι άδεια τότε ο αλγόριθμος μας τερματίζει. Έχοντας βρει, λοιπόν, ένα μη άδειο bucket, βγάζουμε το πρώτο του στοιχείο και ενημερώνουμε τις αλλαγές που έγιναν στο γράφημα. Αν βρούμε μικρότερο μονοπάτι από την κορυφή πηγής στην κορυφή προορισμού τότε κρατάμε αυτό και διαγράφουμε την προηγούμενη απόσταση από τη λίστα dist. Αυτό θα γίνει μόνο αν η μεγάλη διαδρομή δεν είναι άπειρη. Τέλος, ανανεώνουμε τις αποστάσεις και εισάγουμε την κορυφή σε ένα άλλο bucket.

Τέλος, έχουμε την main συνάρτηση.

```
int main()
```

Εκεί δημιουργούμε το γράφημα, θέτοντας τυχαίες τιμές τόσο στους κόμβους, όσο και στα βάρη , με την βοήθεια της συνάρτησης rand(). Ύστερα βρίσκουμε το μέγιστο βάρος που έχει μια ακμή στο γράφημα και την κρατάμε καθώς χρειάζεται για τον αλγόριθμο του Dial. Τέλος, καλούμε την συνάρτηση Dial() και μετράμε τον χρόνο εκτέλεσης της, καθώς και τη συνάρτηση Dijkstra της LEDA

Αρχείο radix.cpp:

Ο κώδικας αποτελείται από 2 συναρτήσεις. Αρχικά, η συνάρτηση:

```
void dijkstra(graph& G, node s, edge_array<int>&  cost, node_array<int>&  dist, node_array<edge>& pred, p_queue<int,node,r_impl>& R)
```

Η οποία είναι ο αλγόριθμος Dijkstra με την χρήση radix heap. Κάθε θέση του heap, αποτελείται από έναν πίνακα στον οποίο εμπεριέχονται στοιχεία με συγκεκριμένες ετικέτες, ανάλογα με το range που έχει ο κάθε πίνακας.

Πρώτα απ' όλα, αρχικοποιούμε όλες τις αποστάσεις με το άπειρο και όλους τους προγόνους με 0. Παίρνουμε την κορυφή πηγή και την προσθέτουμε στο heap και θέτουμε την απόσταση της στον πίνακα dist στο 0.

Έπειτα ψάχνουμε να βρούμε ένα bucket (κάποιον πίνακα του heap) που να μην είναι άδειο. Όταν το βρούμε ψάχνουμε για το στοιχείο που έχει την μικρότερη ετικέτα απόστασης. Αφαιρούμε το στοιχείο αυτό από το heap και ενημερώνουμε τις αποστάσεις για τις ετικέτες που είναι μικρότερες από αυτήν που βρήκαμε προηγουμένως. Τέλος, μειώνουμε το κλειδί του heap και ανακατανέμουμε τα στοιχεία με τις μεγαλύτερες ετικέτες σε άλλα buckets.

Ακόμα η συνάρτηση:

```
int main(){
```

Στην οποία δημιουργούμε το γράφημα, βάζοντας τον αριθμό των κορυφών και των ακμών που θέλουμε και προσθέτουμε τα βάρη των ακμών.

Ύστερα, βρίσκουμε το μέγιστο βάρος ακμής που υπάρχει στο γράφημα μας, καθώς μας χρειάζεται για τον αλγόριθμο.

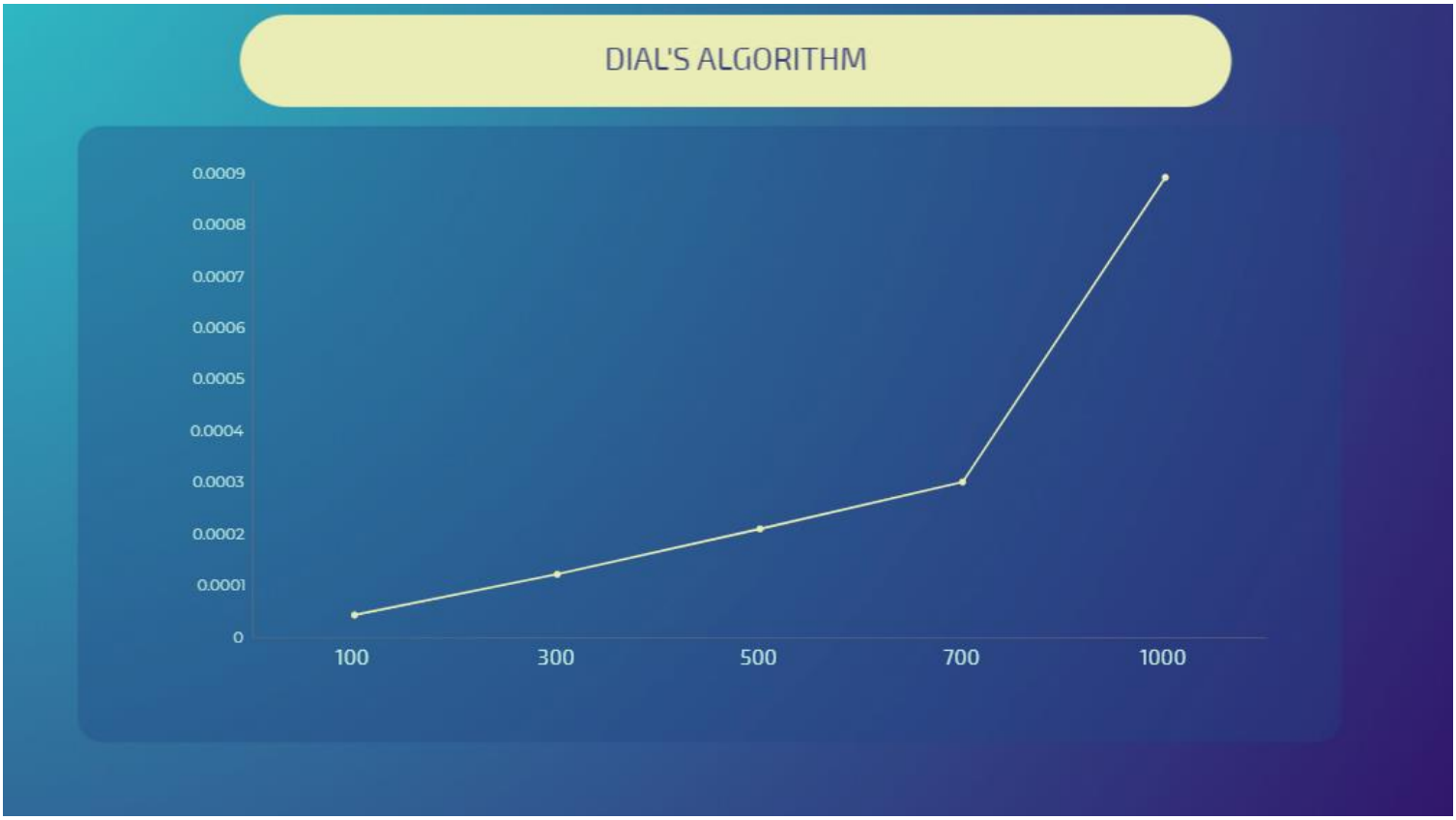
Τέλος, καλούμε την συνάρτηση που αναφέρθηκε παραπάνω, αλλά και την συνάρτηση Dijkstra της LEDA και μετράμε τον χρόνο εκτέλεσης τους.

Πειραματική Αξιολόγηση:

Η πειραματική αξιολόγηση του αλγορίθμου έχει γίνει για 5 διαφορετικές τιμές κορυφών. Κάθε περίπτωση δοκιμάστηκε 10000 φορές και διαιρέθηκε δια του 10000. Οπότε ο χρόνος αυτός είναι ο μέσος όρος όλων των χρόνων που προέκυψαν από τις εκτελέσεις.

Υλοποίηση για Dial's Algorithm	
Αριθμός Κορυφών	**Χρόνος(sec)**
100	0.000045
300	0.000124
500	0.000212
700	0.000303
1000	0.000893

Γράφημα

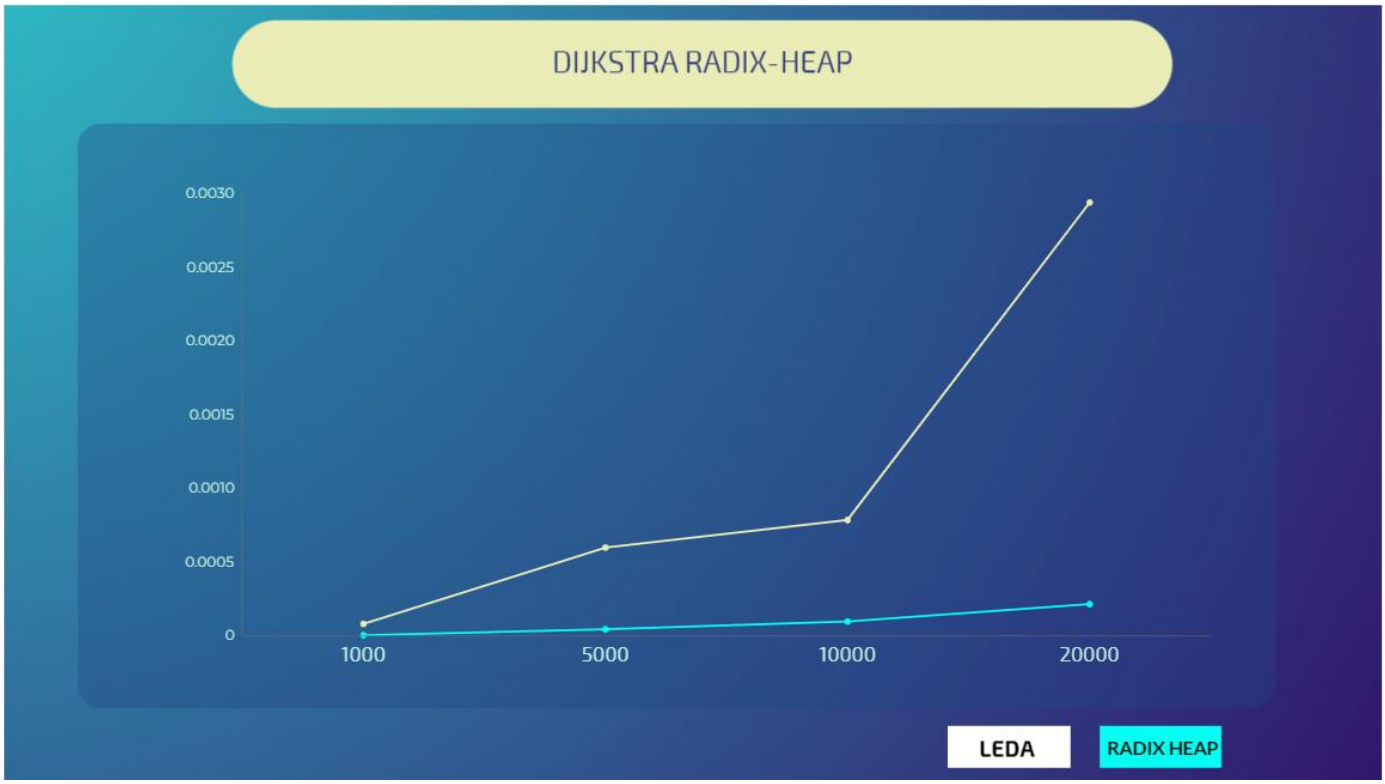


Η πειραματική αξιολόγηση του αλγορίθμου έχει γίνει για 4 διαφορετικές τιμές κορυφών και ακμών. Κάθε περίπτωση δοκιμάστηκε 10000 φορές και διαιρέθηκε δια του 10000. Οπότε ο χρόνος αυτός είναι ο μέσος όρος όλων των χρόνων που προέκυψαν από τις εκτελέσεις.

Υλοποίηση με Radix-Heap		
Αριθμός Κορυφών	**Αριθμός Ακμών**	**Χρόνος(sec)**
1000	2000	0.000007
5000	10000	0.000047
10000	15000	0.0001
20000	40000	0.000218

Υλοποίηση Dijkstra της LEDA		
Αριθμός Κορυφών	**Αριθμός Ακμών**	**Χρόνος(sec)**
1000	2000	0.000084
5000	10000	0.000602
10000	15000	0.000789
20000	40000	0.002944

Γράφημα:



Σημειώσεις:

Πειραματική αξιολόγηση δεν έχει γίνει για τον αντίστοιχο αλγόριθμο της Leda στο Dial's implementation, λόγω πολλών error.

Η εύρεση του μέγιστου κόστους δίνει Segmentation Fault, οπότε στο radix heap.cpp έχω θέσει ως μέγιστο βάρος το 99. Μερικές φορές κατά την εκτέλεση εμφανίζεται το error:

```
LEDA ERROR HANDLER
        r_heap::insert: k= 117 out of range [23,113]
 (_r_heap.cpp:184)
```

υποθέτω επειδή δεν βρίσκει την τιμή αυτή στα κόστη ή επειδή δεν είναι μέγιστη.

Στο dial.cpp το μέγιστο κόστος βρίσκεται, ωστόσο πάλι εμφανίζεται segmentation fault μετά την εκτύπωση του χρόνου που χρειαζόμαστε.

Το αρχείο Makefile έχει δημιουργηθεί για το αρχείο dial.cpp. Για να εκτελέσετε το radix.cpp, θα πρέπει να αλλάξετε στο Makefile: f = radix.cpp.