ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ουρές προτεραιότητας για την αποδοτικότερη υλοποίηση του Αλγορίθμου του Dijkstra

Ονοματεπώνυμο: Καλλιφείδας Νικόλαος

Α.Μ: 1054279

Επιβλέπων καθηγητής: Χρήστος Ζαρολιάγκης

Πίνακας περιεχομένων

Ευχαριστίες

Προτού ξεκινήσω την περιγραφή της διπλωματικής μου εργασίας θα ήθελα να ευχαριστήσω όσους με βοήθησαν ώστε να υλοποιηθεί και να εκπονηθεί. Συγκεκριμένα ευχαριστώ τον διδάσκοντα και επιβλέποντα καθηγητή κ. Χρηστό Ζαρολιάγκη αφενός για την ανάθεση μιας τόσο σημαντικής και κυρίως ενδιαφέρουσας εργασίας και αφετέρου για την καθοδήγηση και την παροχή βοηθητικού υλικού όπως διάφορα επιστημονικά papers και έτοιμες βιβλιοθήκες. Επίσης θέλω να ευχαριστήσω τον κύριο Ανδρέα Παρασκευόπουλο που μου παρείχε πολύ σημαντική βοήθεια όσον αφορά την υλοποίηση της εργασίας και τον κώδικα για τον προγραμματισμό και την εκτέλεση των Αλγορίθμων και ουρών προτεραιότητας.

ΚΕΦΑΛΑΙΟ 1.ΕΙΣΑΓΩΓΗ

Σε αυτό το κεφάλαιο θα γίνει μια εισαγωγή στον Αλγόριθμο του Dijkstra [25] -που χρησιμοποιείται για την εύρεση των συντομότερων διαδρομών σε γραφήματα και πιο συγκεκριμένα σε οδικά δίκτυα μεγαλουπόλεων- αλλά και στις ουρές προτεραιότητας, την σημασία. Επίσης, θα γίνει μια ανάλυση των στόχων της διπλωματικής εργασίας, των ενοτήτων που καλύπτει και της συνεισφοράς στην πραγματοποίησή της. Στην αρχή του κεφαλαίου (σελίδες 2-3) παραθέτω μια συνοπτική λίστα με τα περιεχόμενα και τα κεφάλαια που καλύπτει η εργασία.

1.1.Εισαγωγή στο Πρόβλημα

Η έννοια του προβλήματος των συντομότερων διαδρομών μας απασχολεί αρκετά συχνά στις μέρες μας, κυρίως στον τομέα της επιστήμης των υπολογιστών αλλά και στην καθημερινή μας ζωή στα οδικά δίκτυα και στις καθημερινές διαδρομές μας. Ένας άλλος τομέας που παρατηρούμε αυτό το πρόβλημα είναι στην τηλεπικοινωνίες και στα τηλεπικοινωνιακά δίκτυα μιας και το κύριο ζήτημα της εποχής μας είναι η πιο γρήγορη μετάδοση δεδομένων για μεγαλύτερο εύρος δικτύων με όσο το δυνατόν μικρότερο κόστος υλοποίησης.

Βασισμένη στην εύρεση της συντομότερης διαδρομής είναι και η σχεδίαση ειδικών εφαρμογών που χρησιμοποιούμε όλοι στην καθημερινότητά μας. Η κυριότερη εξ αυτών είναι ο χάρτης της Google (GoogleMaps) που υπάρχει σχεδόν σε κάθε κινητό τηλέφωνο του πλανήτη. Η χρησιμότητά της είναι η αναζήτηση της πιο γρήγορης διαδρομής μεταξύ δύο σημείων του χάρτη ή από την τοποθεσία του χρήστη σε κάποιον προορισμό. Άλλες εφαρμογές, λιγότερο διαδεδομένες είναι το HereWeGo και το εργαλείο του GPS (Global Positioning System) που λειτουργεί βάσει της τοποθεσίας του χρήστη και τον βοηθά να προσανατολιστεί και να ανακαλύψει νέες τοποθεσίες και μέρη.

Εξίσου σημαντική είναι η έννοια αυτή για τα υπολογιστικά συστήματα και την μετάδοση. Βασικός σκοπός των δικτύων είναι η μετάδοση σήματος και δεδομένων από μια πηγή (server) σε πολλούς προορισμούς (clients-πελάτες). Για να είναι πιο αποτελεσματική η μετάδοση του σήματος πρέπει το σύστημα δρομολόγησης να είναι κατάλληλα σχεδιασμένο ώστε να απαιτεί όσο το δυνατόν μικρότερο κόστος σχεδίασης και παράλληλα να εξυπηρετεί όλους τους πελάτες του δικτύου αποτελεσματικά. Έτσι εύκολα καταλαβαίνει

κανείς, ότι η σχεδίαση βάσει ενός αλγορίθμου εύρεσης συντομότερων διαδρομών είναι κάτι παραπάνω από απαραίτητη.

Αυτά τα προβλήματα καλείται να επιλύσει ο Αλγόριθμος του Dijkstra που είναι υπεύθυνος για την εύρεση της πιο σύντομης διαδρομής μεταξύ δύο ή περισσότερων σημείων ενός δικτύου ή γραφήματος. Ο αλγόριθμος αυτός πετυχαίνει πάντοτε τον σκοπό του μιας και σε κάθε βήμα του βρίσκει την πιο σύντομη απόσταση ακολουθώντας την μέθοδο της άπληστης αναζήτησης (greedy) που θα αναλύσουμε παρακάτω. Το πρόβλημα αυτό μπορεί να αναπαρασταθεί με την χρήση γραφημάτων που αναπαριστούν τα οδικά δίκτυα ή τα υπολογιστικά συστήματα που αναφέραμε και να επιλυθεί αποτελεσματικά μέσω του Αλγορίθμου μας.

Σημαντικό ρόλο στην επίλυση έχουν και κάποιες ειδικές δομές δεδομένων, οι ουρές προτεραιότητας που βοηθάνε στην αποδοτικότερη υλοποίηση του αλγορίθμου Dijkstra μειώνοντας το κόστος υλοποίησης (τελική απόσταση) του.

Έτσι με την εφαρμογή των ουρών προτεραιότητας στον Αλγόριθμο του Dijkstra βελτιώνεται κι άλλο η απόδοση του αλγορίθμου και πλέον μπορεί ο οποιοσδήποτε να βρει την βέλτιστη διαδρομή από την τοποθεσία του σε κάποιο μέρος που θέλει να πάει, όπως την δουλειά του ή κάποιο σημείο ενδιαφέροντος όπως ένα αξιοθέατο ή έναν χώρο διασκέδασης.

1.2.Στόχος της Διπλωματικής

Στόχος της διπλωματικής εργασίας είναι η υλοποίηση του Αλγορίθμου Συντομότερων διαδρομών (Dijkstra) με χρήση ουρών προτεραιότητας ώστε να ελαχιστοποιηθεί η χρονική πολυπλοκότητα του προβλήματος των συντομότερων διαδρομών και να ερευνηθεί ποια ουρά είναι πιο αποτελεσματική για κάθε περίπτωση που θα εξετάσουμε.

Μελετήθηκαν διάφορες ουρές προτεραιότητας στον Αλγόριθμο και έγινε εφαρμογή τους σε πραγματικά δεδομένα, δηλαδή πραγματικές συντεταγμένες σημείων. Τα δεδομένα αυτά αφορούν οδικά δίκτυα μεγαλουπόλεων των Ηνωμένων Πολιτειών της Αμερικής που αναπαρίστανται ως γραφήματα με πραγματικά σημεία για κορυφές και είναι βασισμένα σε πραγματικές συντεταγμένες.

Στην συνέχεια θα πραγματοποιηθούν αξιολογήσεις αναφορικά με τον χρόνο εκτέλεσης του Αλγορίθμου για διαφορετικές συνθήκες (μέγεθος γραφήματος, αποστάσεις) ώστε να βρούμε την πιο αποτελεσματική ουρά για κάθε περίπτωση. Επίσης θα καταγράψουμε και το πλήθος των ταξινομήσεων κάθε ουράς για τις παραπάνω περιπτώσεις ώστε να βρούμε την ουρά με το μικρότερο δυνατό κόστος υλοποίησης. Τέλος θα παρουσιαστούν αναλυτικά γραφήματα και διαγράμματα που θα περιέχονται οι χρόνοι εκτέλεσης.

1.3. Συνεισφορά

Σε αυτό το σημείο θα παρουσιαστεί η συνεισφορά στην υλοποίηση και εκπόνηση της διπλωματικής εργασίας. Αρχικά έγινε μελέτη και ανάλυση βασικών εννοιών όπως ο Αλγόριθμος Συντομότερων διαδρομών (Dijkstra) και οι ουρές προτεραιότητας μαζί με τους χρόνους υλοποίησής τους.

Ύστερα μελετήθηκαν οι εξής εργασίες: [1] και [2]. Η μελέτη των συγκεκριμένων διαλέξεων είχε ως σκοπό την κατανόηση των ουρών προτεραιότητας για την αποδοτικότερη υλοποίηση του αλγορίθμου του Dijkstra και την εφαρμογή του σε πραγματικά γραφήματα όπως τα οδικά δίκτυα μεγαλουπόλεων.

Μεταφορτώθηκαν κάποια δεδομένα που αναπαριστούν γραφήματα σε μορφή πραγματικών οδικών δικτύων από τον ιστότοπο του DIMACS9. Τα γραφήματα περιέχουν πραγματικά σημεία μεγαλουπόλεων των ΗΠΑ με τις πραγματικές τους συντεταγμένες στον χάρτη (γεωγραφικό μήκος και πλάτος) ως κόμβους και τις αποστάσεις μεταξύ των σημείων αυτών ως ακμές των γραφημάτων. Πιο συγκεκριμένα τα γραφήματα στα οποία εργάστηκα αφορούσαν τα οδικά δίκτυα των πολιτειών της Νέας Υόρκης (NY- μεγέθους 270 χιλιάδων κορυφών και 730 χιλιάδων ακμών) και της πολιτείας της Florida (FLA-1 εκατομμυρίου κορυφών και 2,7 εκατομμυρίων ακμών).

Έπειτα υλοποιήθηκε ο Αλγόριθμος του Dijkstra [4],[5],[6],[7],[10] με χρήση τεσσάρων διαφορετικών ουρών προτεραιότητας (Binary Heap-Standard [17] και [19], Binary Heap No-Dec, Pairing Heap, Sequence Heap [3]) ώστε να βελτιωθεί ο χρόνος υλοποίησης του αλγορίθμου. Ο αλγόριθμος εκτελέστηκε έχοντας για δεδομένα τα παραπάνω γραφήματα που αναφέρθηκαν. Για κάθε μια ουρά προτεραιότητας του αλγορίθμου εκτελέστηκαν 20.000 ερωτήματα (queries) που θεωρείται επαρκής αριθμός για συμπεράσματα αναφορικά με την συμπεριφορά της κάθε ουράς στον αλγόριθμο σε μεγάλα γραφήματα. Για κάθε ουρά προτεραιότητας έγινε εκτέλεση του κανονικού αλγόριθμου του Dijkstra και μια βελτιωμένη εκδοχή του τον A* που βασίζεται στην χρήση ευρετικών μεθόδων που θα αναλύσουμε σε επόμενο κεφάλαιο.

Αφού εκτελέστηκε ο Αλγόριθμος , έγινε καταγραφή των αποτελεσμάτων αναφορικά με τον μέσο χρόνο εκτέλεσης και το μέσο πλήθος ταξινομήσεων για κάθε query που εκτελείται. Για την καλύτερη κατανόηση της συμπεριφοράς των ουρών πραγματοποιήθηκε ομαδοποίησή σύμφωνα με την απόσταση των queries. Οι αποστάσεις αυτές χωρίζονται στις μικρές (short), μεσαίες (medium) και τις μεγάλες (long). Επιπλέον σχεδιάστηκαν και κάποια ενδεικτικά γραφήματα για κάθε τύπο απόσταση.

Σε αυτό το σημείο το συμπέρασμα από τα γραφήματα και τις μετρήσεις μας ήταν ,ότι η πιο αποδοτική υλοποίηση του Αλγορίθμου γίνεται με την Sequence Heap η οποία παρουσιάζει πολύ καλύτερους χρόνους εκτέλεσης για κάθε υλοποίηση του Αλγορίθμου αλλά και κάθε τύπο απόστασης. Δεύτερη πιο αποδοτική ουρά ήταν ο Binary Heap No-Dec με τις υπόλοιπες δύο ουρές (Standard Binary Heap, Pairing Heap) να ακολουθούν.

1.4. Δομή της Διπλωματικής

Η δομή της διπλωματικής εργασίας είναι χωρισμένη σε 7 κεφάλαια όπου το καθένα καλύπτει και από ένα διαφορετικό κομμάτι.

- Κεφάλαιο 1: Στο υπάρχον κεφάλαιο γίνεται μια συνοπτική ανάλυση των απαιτήσεων του προβλήματος μας του στόχου της εργασίας που θα υλοποιήσουμε.
- Κεφάλαιο 2: Εδώ θα εξηγήσουμε κάποιες βασικές ορολογίες που θα μας χρησιμεύσουν στην ανάλυση και την επίλυση του προβλήματος. Κάποιοι από τους όρους είναι τα γραφήματα και οι εφαρμογές τους μαζί με χαρακτηριστικά παραδείγματα, οι ουρές προτεραιότητας και ο αντίστοιχος ψευδοκώδικας τους (επιπλέον παραθέτω και τον κώδικα σε γλώσσα προγραμματισμού C++) για την υλοποίηση τους. Επίσης θα δούμε πως υλοποιείται ο αλγόριθμος του Dijkstra βήμα προς βήμα μαζί και με απλά παραδείγματα. Τέλος θα γίνει μια προσπάθεια επεξήγησης της λειτουργίας της κρυφής μνήμης (cache) και του τρόπου αποθήκευσης των δεδομένων.
- Κεφάλαιο 3: Θα παρουσιαστούν κάποιες ενδεικτικές ουρές προτεραιότητας και ο τρόπος λειτουργίας και ταξινόμησης των στοιχείων σε κάθε περίπτωση. Θα δούμε παραδείγματα υλοποιήσεων τους όπως και τα αντίστοιχα δέντρα ταξινόμησης. Τέλος έχουμε καταγράψει τους χρόνους εκτέλεσης των βασικών πράξεων μεταξύ ουρών και στοιχείων για την ανάλυση του κόστους και της πολυπλοκότητας τους.
- Κεφάλαιο 4: Θα αναλυθούν κάποια βασικά ζητήματα υλοποίησης της εργασίας. Τα ζητήματα αυτά αφορούν τις προδιαγραφές του υπολογιστή που εργαστήκαμε και των βασικών εργαλείων που διευκόλυναν την δουλειά μας, όπως αυτά της Microsoft (word, excel Κλπ.). Επιπλέον θα ασχοληθούμε με τους editors και τους compilers (μεταγλωττιστές) που χρησιμοποιήθηκαν για την επεξεργασία και την αποσφαλμάτωση αλλά και τις γλώσσες προγραμματισμού. Τέλος θα αναφερθούμε και στις έτοιμες βιβλιοθήκες που χρησιμοποιήθηκαν και περιλαμβάνουν άφθονο υλικό από κώδικες (include) και αρχεία που έκαναν ευκολότερη την υλοποίηση.
- Κεφάλαιο 5: Στο ίσως σημαντικότερο κεφάλαιο της διπλωματικής θα παρουσιαστούν τις πειραματικές αξιολογήσεις που έγιναν. Θα δείξουμε τους χρόνους εκτέλεσης της κάθε ουράς για διαφορετικές συνθήκες αλλά και το σύνολο των ταξινομήσεων που πραγματοποιήθηκαν σε κάθε περίπτωση. Θα παρουσιαστούν τα δεδομένα τα οποία μας βοήθησαν στην εκτέλεση των προγραμμάτων μας και τις διαφορετικές παραμέτρους για το καθένα. Θα αναλυθούν και οι περιπτώσεις για τους τρόπους υλοποίησης του Αλγορίθμου σχετικά με τον τρόπο επίσκεψης των κορυφών των γραφημάτων μας. Στο τέλος θα υπάρχει κι ένας αναλυτικός πίνακας με την κατάταξη της κάθε ουράς όσον αφορά τους χρόνους και τις ταξινομήσεις της κάθε ουράς

- Κεφάλαιο 6: Θα αναλυθούν τα συμπεράσματα και οι παρατηρήσεις αναφορικά με τις εκτελέσεις των ουρών στον Αλγόριθμο μας. Θα απαντηθούν σημαντικά ερωτήματα όπως για το ποια ουρά είναι η αποδοτικότερη ανάλογα με τις συνθήκες εκτέλεσης του Αλγορίθμου μας. Επίσης θα σημειωθούν και οι προοπτικές της διπλωματικής εργασίας, ο τρόπος δηλαδή με τον οποίο μπορεί να επεκταθεί και να εκμεταλλευτεί από άλλους χρήστες όπως φοιτητές ή και διδάσκοντες.
- Κεφάλαιο 7: Καταγράφονται σε αυτό το σημείο οι πηγές και η βιβλιογραφία που χρησιμοποιήθηκε για την συγγραφή της εργασίας. Αυτές οι πηγές σχετίζονται με την θεωρία, τα παραδείγματα αλλά και τα εργαλεία που χρησιμοποιήσαμε για την δημιουργία των γραφημάτων μας.

ΚΕΦΑΛΑΙΟ 2. ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ

Σε αυτό το κεφάλαιο θα γίνει μια επεξήγηση των βασικών όρων του προβλήματος που θα μας είναι πολύ χρήσιμοι για την κατανόηση και την επίλυσή του.

2.1.Γράφημα

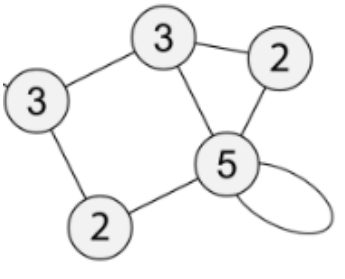
2.1.1. Ορισμός

Ένας απλός ορισμός για το γράφημα είναι η αναπαράσταση των σχέσεων που αναπτύσσονται μεταξύ ορισμένων οντοτήτων ή ποσοτήτων [9]. Ένα γράφημα αποτελείται από τις κορυφές (κόμβους), δηλαδή τις οντότητες που αναπαρίστανται και τις ακμές, δηλαδή τις σχέσεις μεταξύ των οντοτήτων αυτών. Αν οι ακμές, έχουν προσανατολισμό (κατεύθυνση), τότε το γράφημα μας ονομάζεται κατευθυνόμενο ενώ αν δεν έχουν θα ονομάζεται μη-κατευθυνόμενο. Αναφορικά με το πλήθος των ακμών του ένα γράφημα χωρίζεται σε δύο κατηγορίες:

- Πυκνό: αν έχει μεγάλο αριθμό ακμών
- Αραιό: αν το πλήθος ακμών του δεν είναι πολύ μεγάλο

Βαθμός κορυφής (deg) ονομάζεται ο αριθμός των ακμών που καταλήγουν σε μια κορυφή του γραφήματος.

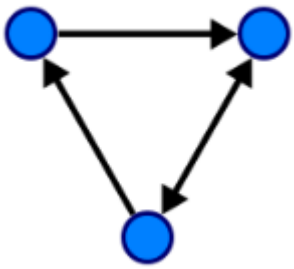
2.1.2 Παραδείγματα γραφημάτων



Εικόνα 1. Παράδειγμα γραφήματος που είναι μη-κατευθυνόμενο και έχει 5 κορυφές και 7 ακμές

Το παραπάνω γράφημα(εικόνα 1.) είναι ένα μη-κατευθυνόμενο γράφημα μιας και οι ακμές του δεν έχουν κατεύθυνση-προσανατολισμό.

- V(G)={v1,v2...v_n} το σύνολο των κορυφών, στην συγκεκριμένη περίπτωση είναι :{2,2,3,3,5}
- E(G)={e1,e2...e_m} το σύνολο των ακμών, στην περίπτωση μας : {(3,2), (3,3), (2,3), (3,5), (5,3), (2,5), (5,5), (5,2)}
  - Βαθμός κορυφής: deg(5)=4 μιας και καταλήγουν σε αυτό οι ακμές (3,5), (2,5), (2,5), (5,5)
  - Διαδρομή: ένα υποσύνολο κορυφών που ενώνονται μεταξύ τους μέσω των ακμών (path)



Εικόνα 2. Παράδειγμα κατευθυνόμενου γραφήματος 3 κορυφών και 3 ακμών

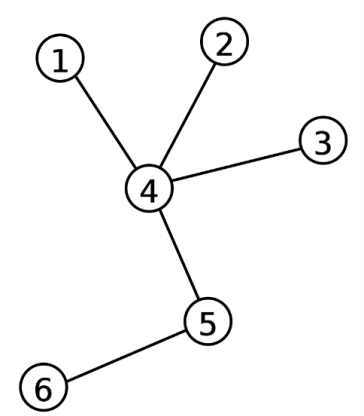
Το γράφημα που φαίνεται (εικόνα 2.) είναι ένα κατευθυνόμενο γράφημα με 3 ακμές και 3 κορυφές. Όπως βλέπουμε οι ακμές του έχουν κατεύθυνση και για αυτό θεωρείται κατευθυνόμενο.

2.1.3. Κατηγοριοποίηση των γραφημάτων

1. G(n, m): Ονομάζουμε τα τυχαία γραφήματα που είναι μη-κατευθυνόμενα γραφήματα αποτελούμενα από n: κορυφές και m: ακμές όπου η επιλογή των ακμών έχει γίνει με τυχαίο τρόπο και με κάθε ακμή να έχει την ίδια πιθανότητα επιλογής με κάποια άλλη κατά την κατασκευή του γραφήματος
2. Real-Life Graphs: Σε αυτή την κατηγορία ανήκουν τα πραγματικά γραφήματα όπως οδικά δίκτυα πόλεων ή και χωρών. Σύμφωνα με τις παραδοχές DIMACS μπορούμε κάθε οδικό δίκτυο να το θεωρήσουμε ως ένα γράφημα όπου οι κόμβοι θα είναι πραγματικά σημεία με πραγματικές συντεταγμένες, οι ακμές τα μονοπάτια (paths) μεταξύ των σημείων και τα βάρη των ακμών οι χιλιομετρικές αποστάσεις μεταξύ των σημείων. Ως επί των πλείστων τα γραφήματα αυτά είναι αραιά. Αυτού του τύπου γραφήματα θα αναλύσουμε περαιτέρω στις επόμενες ενότητες.

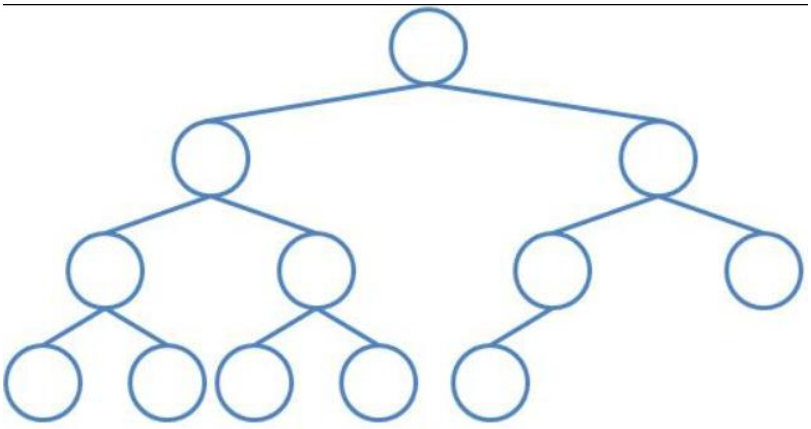
Μια εξίσου σημαντική έννοια στην θεωρία γραφημάτων είναι το δέντρο (εικόνα 3.), μια ξεχωριστή κατηγορία γραφήματος. Το δέντρο ορίζεται ως ένα μη κατευθυνόμενο γράφημα στο οποίο δυο κορυφές συνδέονται μόνο από ένα μονοπάτι. Αποτελείται από την ρίζα που είναι η κορυφή που τοποθετείται στο πιο ψηλό σημείο του δέντρου, τους εσωτερικούς κόμβους και τα φύλλα που είναι οι κόμβοι του πιο κάτω επιπέδου του δέντρου μας.

Κάποιες εξίσου σημαντικές έννοιες των δέντρων είναι οι κόμβοι πατέρα και παιδιού. Αν ξεκινήσουμε μια διαδρομή από την ρίζα θα έχουμε ένα μονοπάτι προς μία κορυφή. Σε αυτό το μονοπάτι θεωρούμε δυο διαφορετικές κορυφές τις α και β. Αν η α βρίσκεται πριν από την β στο μονοπάτι που δημιουργείται τότε λέμε ότι η α είναι ο πατέρας ή πρόγονος της κορυφής β και αντίστοιχα η β παιδί ή απόγονος της α μιας και βρίσκεται μετά από αυτήν στο μονοπάτι μας. Οι κόμβοι που βρίσκονται στο πιο χαμηλό επίπεδο του δέντρου είναι τα φύλλα του και δεν έχουν κανένα παιδί.



Εικόνα 3. Ένα απλό παράδειγμα δέντρου με 6 κορυφές και 5 ακμές. Ως ρίζα μπορεί να θεωρηθεί η κορυφή 3 και σαν φύλλα οι κορυφές 1,2 και 6. Αν θεωρήσουμε μονοπάτι από την ρίζα 3 στο παιδί 6 τότε για τις ενδιάμεσες κορυφές η 4 είναι ο πατέρας της 5 και αντίστοιχα η 5 είναι το παιδί της κορυφής 4.

Μια πολύ σημαντική κατηγορία δέντρου είναι το δυαδικό δέντρο (εικόνα 4.) το οποίο είναι σχεδιασμένο έτσι ώστε κάθε κορυφή του να έχει το πολύ δυο παιδιά, ενώ αν έχει ακριβώς δυο παιδιά τότε ονομάζεται πλήρες δυαδικό δέντρο. Αυτός ο τρόπος ταξινόμησης των κορυφών χρησιμοποιείται από αρκετές ουρές προτεραιότητας και ειδικότερα από τον δυαδικό σωρό (Binary Heap).



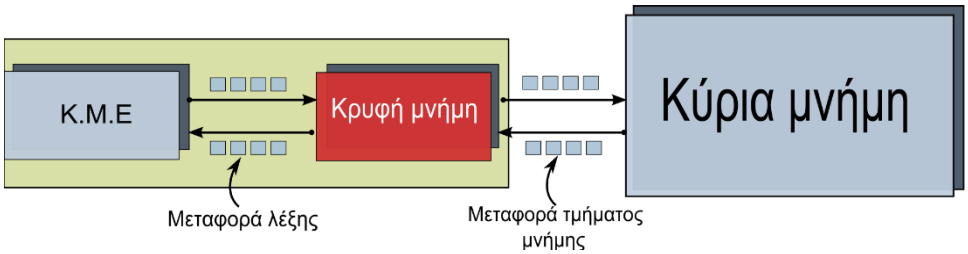
Εικόνα 4. Παράδειγμα δυαδικού δέντρου αποτελούμενο από 12 κορυφές. Όπως παρατηρούμε δεν είναι πλήρες γιατί στο δεξιά υποδέντρο κάποιες κορυφές του έχουν ένα ή και κανένα παιδί.

2.2. Μνήμες

Μια πολύ σημαντική ενότητα της διπλωματικής αλλά και της επιστήμης γενικότερα είναι οι μνήμες [14]. Ως μνήμες ονομάζουμε τον αποθηκευτικό χώρο του υπολογιστή όπου αποθηκεύονται τα ψηφιακά δεδομένα που απαιτούνται για την εκτέλεση των λειτουργιών.

2.2.1 Δομή μνήμης

Πιο αναλυτικά βλέπουμε την σχεδίαση της μνήμης στο παρακάτω σχήμα. Για την μεταφορά μιας λέξης δεδομένων απαιτείται η αλληλεπίδραση μεταξύ της Κεντρικής Μονάδας Επεξεργασίας (ΚΜΕ) και της Κρυφής μνήμης ενώ για την μεταφορά ενός τμήματος (block) μνήμης βοηθάει και η Κύρια μνήμη του υπολογιστή (εικόνα 5.).



Εικόνα 5. Παράδειγμα μνήμης υλοποιημένη με κύρια και κρυφή μνήμη για την μεταφορά και την αποθήκευση λέξεων

2.2.2. Κρυφή μνήμη

Ως κρυφή μνήμη ορίζουμε το μέρος της μνήμης όπου αποθηκεύονται κάποια δεδομένα τα οποία αποτελούν αντίγραφα δεδομένων που βρίσκονται αλλού [15]. Η χρησιμότητα της είναι η διασύνδεση της ΚΜΕ με την Κύρια μνήμη αλλά και η αποθήκευση δεδομένων συχνά χρησιμοποιούμενων από την Μονάδα Επεξεργασίας που πιθανόν να χρειαστεί πάλι η ΚΜΕ. Έτσι όταν τα χρειαστεί θα γίνει μια αναζήτηση πρώτα στην κρυφή μνήμη που είναι και μικρότερου μεγέθους , με σκοπό την πιο γρήγορη πρόσβαση σε αυτά. Αν η αναζήτηση των δεδομένων είναι επιτυχής τότε εξοικονομείται χρόνος μιας και σε διαφορετική περίπτωση θα έπρεπε να γίνει προσπέλαση και της Κύριας μνήμης. Μια τέτοια αναζήτηση επιβαρύνει το σύστημα με επιπλέον χρόνο για προσπελάσεις.

2.2.3. Προσωρινή μνήμη

Η προσωρινή μνήμη είναι άλλο ένα είδος μνήμης που χρησιμεύει στην προσωρινή αποθήκευση δεδομένων , με σκοπό την εξοικονόμηση χρόνου προσπέλασης [16]. Η διαφορά της με την Cache memory (Κρυφή μνήμη) είναι , ότι αποτελεί μέρος ενός άλλου τμήματος όπως του σκληρού δίσκου η της RAM και ορίζεται από το εκάστοτε Λειτουργικό σύστημα. Σε αντίθεση με την κρυφή μνήμη , που αποτελεί από μόνη της ένα ξεχωριστό κομμάτι της Μονάδας επεξεργασίας.

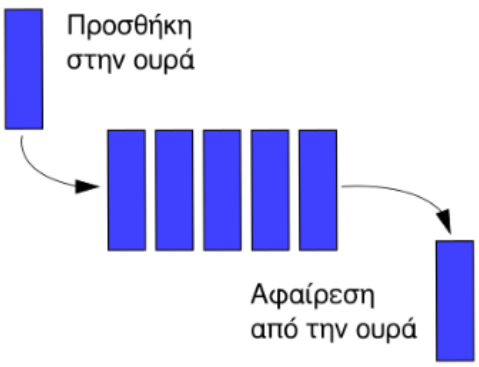
2.3. Βασικές δομές δεδομένων

Τώρα θα κάνουμε μια ανάλυση κάποιων βασικών δομών δεδομένων που είναι σημαντικές για την εργασία μας. Ως δομή δεδομένων ορίζεται η διαδικασία εισαγωγής και απομάκρυνσης στοιχείων με τρόπο ώστε όλη η δομή να μην αλλοιώνεται. Κάθε δομή δεδομένων έχει ως αφηρημένη έννοια συγκεκριμένο ορισμό, δηλαδή διαδικασία εισαγωγής/απομάκρυνσης στοιχείων, αλλά μπορεί να υλοποιείται σε έναν υπολογιστή με διαφορετικούς τρόπους.

Οι πιο σημαντικές δομές δεδομένων είναι η ουρά (queue) και η στοίβα (heap).

2.3.1.Ουρές

Είναι μια δομή δεδομένων που έχει την μορφή παρατεταμένης συλλογής στοιχείων (εικόνα 6.) του τύπου First In First Out (FIFO) [11]. Σε μια τέτοια δομή, το πρώτο στοιχείο που θα εισαχθεί θα είναι και το πρώτο που θα αφαιρεθεί.



Εικόνα 6. παράδειγμα ουράς με τις λειτουργίες εισαγωγής και εξαγωγής στοιχείων

Βασικές Λειτουργίες:

- Εισαγωγή στοιχείου (Insert) →εισάγει ένα νέο στοιχείο στην ουρά μας
- Εξαγωγή στοιχείου (Delete) → όταν η ουρά είναι πλήρης στοιχείων θα έχουμε υπερχείλιση, άρα πρέπει να αφαιρεθεί κάποιο στοιχείο. Επειδή είναι της μορφής FIFO (first in first out) η επιλογή του στοιχείου θα γίνει σύμφωνα με το ποιο εισάγαμε πρώτο στην ουρά μας. Αφού επιλεγεί θα διαγραφεί επιτόπου από την ουρά.

enqueue

```
1.  Αλγόριθμος en_queue
2.  Δεδομένα // rear, size, item //
3.  Αν rear < size τότε
4.      rear ← rear + 1
5.      queue[rear] ← item
6.      done ← Αληθής
7.  αλλιώς
8.      done ← Ψευδής
9.  Τέλος_αν
10. Αποτελέσματα // rear, done //
11. Τέλος en_queue
```

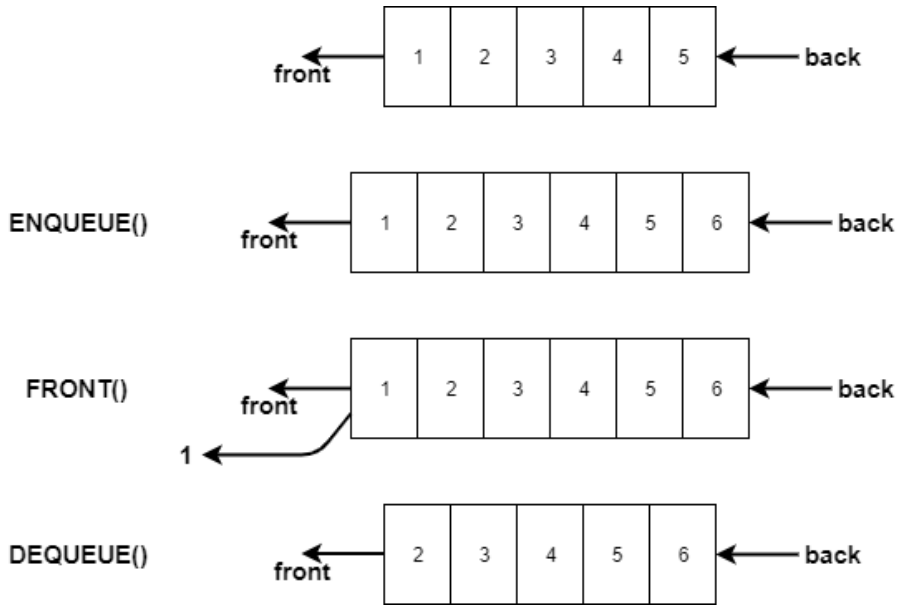
dequeue

```
1.  Αλγόριθμος de_Queue
2.  Δεδομένα // rear, front, queue//
3.  Αν front <= rear τότε
4.      item ← queue[front]
5.      front ← front + 1
6.      done ← Αληθής
7.  αλλιώς
8.      done ← Ψευδής
9.      item ← null
10. Τέλος_αν
11. Αποτελέσματα // item, rear, done /
12. Τέλος de_Queue
```

Εικόνα 7. Ενδεικτικές εικόνες των δυο βασικών λειτουργιών των ουρών. Στην λειτουργία εισαγωγής (enqueue) βλέπουμε πως εισάγεται ένα νέο στοιχείο στην ουρά και στην λειτουργία εξαγωής (dequeue) πως αφαιρούμε ένα στοιχείο αν είναι γεμάτη η ουρά

Στο παραπάνω παράδειγμα (εικόνα 7.) βλέπουμε την υλοποίηση βασικών λειτουργιών των ουρών με την χρήση ψευδοκώδικα. Στο αριστερά μέρος πρώτα ελέγχεται το πλήθος των στοιχείων και αν αυτό είναι μικρότερο από το μέγεθος της ουράς και αν ο ισχυρισμός είναι αληθής τότε εισάγουμε ένα νέο στοιχείο.

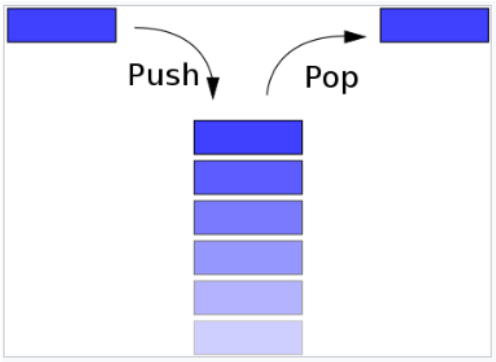
Στην δεύτερη εικόνα ελέγχουμε αν είναι πλήρης η ουρά μας. Αν είναι αληθής ο ισχυρισμός ότι είναι πλήρης και θέλουμε να εισάγουμε νέο στοιχείο θα πρέπει να διαγράψουμε κάποιο προτού γίνει η διαδικασία εισαγωγής. Επειδή η ουρά είναι της μορφής First In First Out, θα εξάγουμε το στοιχείο που εισήχθη πρώτο στην ουρά. Ενδεικτικό παράδειγμα της συγκεκριμένης λειτουργίας παρατίθεται και στην συνέχεια (εικόνα 8.).



Εικόνα 8. Επεξήγηση της εισαγωγής και εξαγωγής στοιχείου από την ουρά. Αρχικά η ουρά έχει 5 στοιχεία και η μέγιστη χωρητικότητά της είναι 5. Όταν γίνει η εισαγωγή (enqueue) του στοιχείου 6 έχουμε υπερχείλιση οπότε πρέπει να διαγραφεί κάποιο από τα στοιχεία (dequeue). Για την καλύτερη ανάλυση έχουμε τοποθετήσεις τους δείκτες front στο στοιχείο που βρίσκεται αριστερά και back σε αυτό που βρίσκεται πιο δεξιά. Έτσι κατά την διαγραφή θα έχουμε την LIFO λειτουργία και θα διαγράψουμε το πιο αριστερά στοιχείο που εισάγαμε πρώτο στην ουρά με δείκτη front ώστε να επιλυθεί το πρόβλημα της υπερχείλισης.

2.3.2. Στοίβες

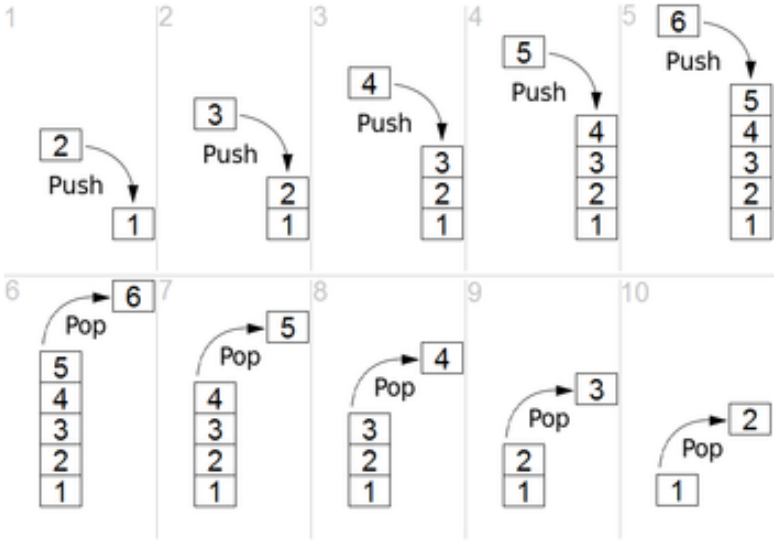
Μια αφηρημένη δομή δεδομένων που είναι της μορφής LIFO (Last In First Out) [12] όπου το τελευταίο στοιχείο που εισάγεται στην ουρά είναι το πρώτο που θα διαγραφεί (εικόνα 9.).



Εικόνα 9. Παράδειγμα στοίβας με τις βασικές τις λειτουργίες (push, pop)

Βασικές Λειτουργίες:

- Εισαγωγή στοιχείου στην στοίβα (Push)
- Αφαίρεση στοιχείου από την στοίβα (Pop) --> όταν η στοίβα είναι πλήρης έχουμε υπερχείλιση (overflow) και πρέπει να αφαιρεθεί κάποιο στοιχείο. Η δομή είναι σχεδιασμένη βάσει της LIFO (Last In First Out) , άρα το στοιχείο που θα αφαιρεθεί θα είναι το αυτό που εισήχθη τελευταίο στην στοίβα.



Εικόνα 10. Παράδειγμα εκτέλεσης της εντολής εισαγωγής (push) για μια στοίβα μεγέθους 6 στοιχείων που εισάγονται κατά αύξουσα αριθμητική τιμή (πάνω εικόνα)

Στην κάτω εικόνα (εικόνα 10.) γίνεται εξαγωγή των στοιχείων (pop) βάσει της LIFO λειτουργίας. Το στοιχείο που εισάγαμε τελευταίο είναι το 6 και όπως βλέπουμε είναι το πρώτο κατά σειρά που θα εξάγουμε. Έπειτα ακολουθεί το 5 που είχε μπει πριν το 6 και διαδικασία συνεχίζεται με τον ίδιο τρόπο μέχρι να απομείνει το στοιχείο 1 στην στοίβα.

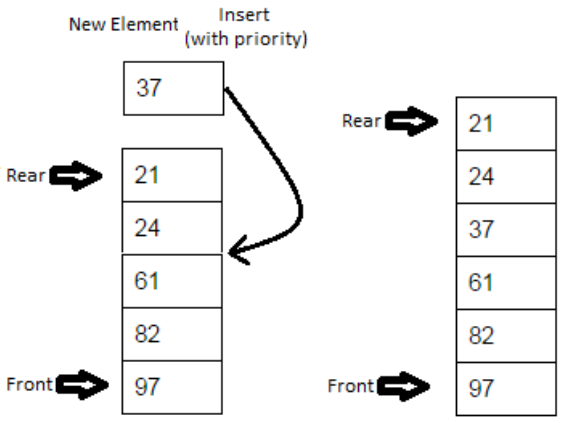
2.4. Ουρές προτεραιότητας

Η ουρά προτεραιότητας ανήκει σε μια διαφορετική κατηγορία ουρών διότι η προτεραιότητα του κάθε στοιχείου της δεν εξαρτάται από την σειρά που εισήχθη αυτό στην ουρά. Κάθε στοιχείο διαθέτει ένα χαρακτηριστικό γνώρισμα, το κλειδί (key) που ανάλογα με την τιμή που θα έχει καθορίζει την προτεραιότητα του στοιχείου στην ουρά. Αν δύο ή παραπάνω στοιχεία έχουν την ίδια τιμή κλειδιού τότε μας αφορά η σειρά με την οποία έχουν εισαχθεί στην ουρά και θα τηρείται η διάταξη (FIFO) που αναφέραμε. [13]

Μια ουρά προτεραιότητας χωρίζεται σε 2 κατηγορίες ανάλογα με την προτεραιότητα. Στην ουρά προτεραιότητας ελάχιστου αν το στοιχείο με την μεγαλύτερη προτεραιότητα είναι αυτό με το μικρότερο (ελάχιστο) κλειδί και στην ουρά μεγίστου αν το στοιχείο με την μεγαλύτερη προτεραιότητα είναι αυτό με το μεγαλύτερο κλειδί (μέγιστο).

Βασικές Λειτουργίες:

- Insert: εισάγει ένα νέο στοιχείο στην ουρά προτεραιότητας
- Delete: διαγράφει ένα στοιχείο της ουράς
- Delete-min: διαγράφει το στοιχείο που έχει τον μικρότερο αριθμό κλειδιού
- Decrease-key: μειώνει κατά κάποιες μονάδες τον αριθμό του κλειδιού που ανήκει σε ένα στοιχείο της ουράς (κάποιες ουρές δεν την εκτελούν). Η λειτουργία αυτή αποτελείται από τρεις επιμέρους λειτουργίες. Αρχικά βρίσκουμε το στοιχείο, μειώνουμε την τιμή του κλειδιού του και ύστερα επιδιορθώνουμε το δέντρο ώστε να τηρείται η ουρά προτεραιότητας.



Εικόνα 11. Αρχικά η ουρά έχει 5 στοιχεία με τα αντίστοιχα κλειδιά τους ταξινομημένα κατά αύξουσα σειρά από το πάνω (rear) μέχρι το κάτω στοιχείο (front). Κατά την εισαγωγή ενός νέου στοιχείου με αριθμό κλειδιού ίσο με 37 το στοιχείο θα τοποθετηθεί ανάλογα με την τιμή του κλειδιού και όχι στην θέση rear όπως στις προηγούμενες δυο περιπτώσεις δομών που αναλύσαμε.

Στο παραπάνω παράδειγμα (εικόνα 11.) αν θέλουμε να κάνουμε delete-min δηλαδή διαγραφή του στοιχείου με την ελάχιστη τιμή κλειδιού πρώτα θα το βρούμε (find) και θα το διαγράψουμε. Αν έπειτα θέλουμε να κάνουμε decrease-key θα κάνουμε πρώτα την μείωση της τιμής του κλειδιού του και μετά θα κάνουμε συγκρίσεις με τα στοιχεία που ήταν πριν από αυτό στην ουρά. Αν μετά την πρώτη σύγκριση το η τιμή του νέου κλειδιού είναι μεγαλύτερη από το προηγούμενο κατά σειρά στοιχείο του τότε δεν θα χρειαστεί να κάνουμε κάποια άλλη αλλαγή θέσης. Αντιθέτως αν είναι μικρότερη η τιμή του κλειδιού από του προηγούμενου στοιχείου τότε θα αλλάξουν θέση τα δυο στοιχεία στην ουρά. Οι συγκρίσεις θα συνεχιστούν έως ότου βρεθεί στοιχείο με τιμή κλειδιού μικρότερη από την «αλλαγμένη» τιμή του κλειδιού του στοιχείου που εκτελέσαμε την πράξη (decrease-key).

2.5. Αλγόριθμος του Dijkstra

Αρχικά θα εξηγήσουμε την γενική έννοια του αλγορίθμου και πώς χρησιμεύει στην επίλυση προβλημάτων όπως αυτό της διπλωματικής μας.

ΑΛΓΟΡΙΘΜΟΣ: ορίζεται ως μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος

Πολυπλοκότητα αλγορίθμου: Με την έννοια πολυπλοκότητα ορίζουμε την κοστολόγηση των συνολικών πόρων που απαιτούνται για την επίλυση ενός προβλήματος. Η πολυπλοκότητα χωρίζεται σε δύο κατηγορίες. Αρχικά έχουμε την χρονική πολυπλοκότητα, δηλαδή τον χρόνο που απαιτείται ώστε να επιλυθεί ένα πρόβλημα, δεδομένου ότι για κάθε βήμα του απαιτείται ένα συγκεκριμένο χρονικό διάστημα για να εκτελεστεί.

Η χρονική Πολυπλοκότητα ενός αλγορίθμου χωρίζεται σε 3 κατηγορίες όσον αφορά τον χρόνο της συνολικής εκτέλεσης. Στην καλύτερη ΧΠ (best case), στην μέση ΧΠ και στην χειρότερη ΧΠ (worst case)

i. Best case: ο ελάχιστος χρόνος που απαιτείται για οποιαδήποτε είσοδο με συγκεκριμένο μέγεθος n. (κάτω φράγμα ΧΠ)
ii. Μέση ΧΠ: είναι ανάμεσα στην best case και την worst case και είναι δύσκολος ο ακριβής προσδιορισμός της.
iii. Worst case: Ο μέγιστος χρόνος που απαιτείται για οποιαδήποτε είσοδο με συγκεκριμένο μέγεθος n. (άνω φράγμα ΧΠ)

Η δεύτερη κατηγορία πολυπλοκότητας είναι η χωρική, η οποία αφορά τον χώρο που καταλαμβάνει ο αλγόριθμος στην μνήμη ώστε να αποθηκεύσει τα δεδομένα που απαιτούνται για την επίλυση του προβλήματος.

2.5.1. Περιγραφή

Αρχικά θα γίνει μια επεξήγηση του προβλήματος των συντομότερων διαδρομών που καλείται να επιλύσει ο αλγόριθμος [25]. Έστω G= (V, E) ένα κατευθυνόμενο γράφημα, του οποίου οι ακμές έχουν συσχετιστεί με μία συνάρτηση βάρους len(u,v)= E → R. Ονομάζουμε len(u,v) το βάρος της ακμής (u, v) του συνόλου E. Για δύο κορυφές του γραφήματος s, t , το βάρος της κατευθυνόμενης διαδρομής από τον κόμβο s στον κόμβο t (στο εξής, s-t διαδρομή), είναι το άθροισμα από τα βάρη των ακμών της. Συντομότερη s-t διαδρομή (ΣΔ) είναι εκείνη η οποία έχει το ελάχιστο βάρος ανάμεσα σε όλες τις s-t διαδρομές. Η απόσταση από το s στο t συμβολίζεται dist(s, t) -και εν συντομία dist(t) παραλείποντας την αρχική κορυφή s από τον συμβολισμό- και είναι το βάρος της συντομότερης s-t διαδρομής στο γράφημα G. Το πρόβλημα της συντομότερης διαδρομής αφορά την εύρεση μίας ΣΔ ανάμεσα σε ένα δοθέν ζεύγος κόμβων του G [23]. Υπάρχουν τέσσερις εκδοχές του προβλήματος:

- Μονό ζεύγος: εύρεση της συντομότερης διαδρομής για μια μόνο αρχική κορυφή s και μία μόνο τελική t.
- Μονή πηγή: αφορά την εύρεση της συντομότερης διαδρομής από μια κορυφή s προς όλες τις υπόλοιπες κορυφές του γραφήματος
- Μονός προορισμός: εύρεση όλων των διαδρομών για κάθε κορυφή του γραφήματος προς μια μοναδική κορυφή t.
- Όλα τα ζεύγη: εύρεση όλων των συντομότερων διαδρομών μεταξύ όλων των κορυφών του γραφήματος.

ΑΛΓΟΡΙΘΜΟΣ DIJKSTRA: Είναι ένας αλγόριθμος που χρησιμοποιείται για την επίλυση τέτοιου είδους προβλημάτων σε γραφήματα με μη-αρνητικά βάρη. Ο αλγόριθμος ξεκινάει από μια αρχική κορυφή s και αναζητά την πιο κοντινή εξετάζοντας όλες τις γειτονικές του κορυφές μέχρι να καταλήξει στην τελική κορυφή [4], [5],[6],[7],[8],[10].

Αρχικά ο αλγόριθμος δεν γνωρίζει τις αποστάσεις των κορυφών. Όταν ξεκινήσει την διαδικασία εκτέλεσής του από την κορυφή s Γνωρίζει μόνο την απόσταση των γειτονικών της κορυφών της s . Ο αλγόριθμος σε κάθε βήμα θα αναζητά τις κοντινότερες αποστάσεις μεταξύ των κορυφών που έχει ήδη επισκεφθεί και στην περίπτωση που βρίσκει κάποια απόσταση προς μια κορυφή θα εκτελεί την χαλάρωση ακμών.

Ο αλγόριθμος αυτός είναι άπληστος (greedy) κάτι που σημαίνει ότι σε κάθε βήμα του ψάχνει την καλύτερη λύση αναφορικά με την απόσταση και μέχρι το τελευταίο βήμα έχει συνθέσει την τελική βέλτιστη λύση, δηλαδή την συντομότερη διαδρομή μεταξύ s-t.

Παρακάτω παρατίθεται ένας ψευδοκώδικας για την εκτέλεση του Αλγορίθμου βήμα προς βήμα και πώς εκτελούνται οι βασικές του λειτουργίες, όπως το χαλάρωμα ακμών, δηλαδή decrease key (Εικόνα 12.). Στον παρακάτω ψευδοκώδικα έχουμε ορίσει ως v την τελική κορυφή μια διαδρομής. Ομοίως και η συνάρτηση έχει οριστεί ως dist (v).

Algorithm 2.1 DIJKSTRA(G, s, t)

Require: graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and Vertices $s, t\mathbf{V}$
Ensure: $\text{dist}(s, v) = \text{dist}(v) \forall v \in \mathbf{V}$

```
1:  for all v ∈ V do
2:      parent(v) ← ⊥
3:      state(v) ← unreached
4:      dist(v) ← ∞
5:  dist(s) ← 0
6:  state(s) ← reached
7:  while vertex v with state(v) = reached exists and state(t) ≠ reached do
8:      Select v ∈ V with state(v) = reached and minimal dist(v)
9:      for all u ∈ V with (v, u) ∈ E do
10:         if dist(v) + len(v, u) < dist(u) then
11:             dist(u) ← dist(v) + len(v, u)
12:             parent(u) ← v
13:             state(u) ← reached
14:     state(v) ← settled
```

Εικόνα 12- Ψευδοκώδικας για την υλοποίηση του Αλγορίθμου του Dijkstra Με την χρήση λίστας

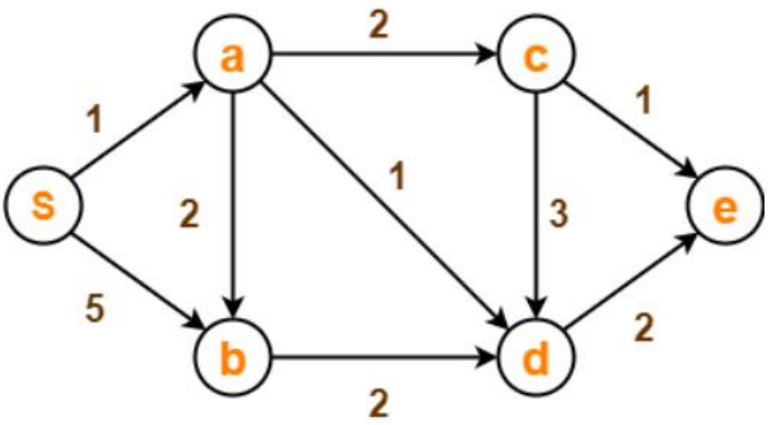
Αρχικά η ετικέτα της κάθε κορυφής είναι ίση με άπειρο μιας και δεν γνωρίζει ο αλγόριθμος καμία από τις αποστάσεις από την αρχική s. Για κάθε κορυφή V του γραφήματος έχουμε 3 πιθανές καταστάσεις. Αυτές είναι η εξερευνημένη (reached) αν γνωρίζουμε ένα πιθανό μονοπάτι από την s στην v (όχι απαραίτητα το συντομότερο), μη-εξερευνημένη (unreached) αν δεν έχει επισκεφθεί από τον αλγόριθμο καθόλου και ταξινομημένη (settled) αν έχει ανακαλυφθεί η συντομότερη διαδρομή από την s στην v.

Για κάθε ακμή από μια κορυφή v σε μια άλλη u (εκτός της s) η ετικέτα dist(u) θα ανανεώνεται σύμφωνα με την συνάρτηση dist(v)+ len(v, u) αν αυτή η απόσταση είναι μικρότερη της dist(u). Σε αυτή την περίπτωση έχουμε το χαλάρωμα ακμών μιας και ο Αλγόριθμος ανακαλύπτει μια συντομότερη διαδρομή από την αρχική κορυφή s στην u.

Για κάθε κόμβο v τέτοιο ώστε η κατάσταση του να είναι settled, η απόσταση από τον s στον v είναι dist(s, v) = dist(v). Η συντομότερη διαδρομή από τον s σε έναν διευθετημένο κόμβο v βρίσκεται κωδικοποιημένη στην ετικέτα γονέα, parent(.). Ο αλγόριθμος τερματίζει όταν επισκεφτεί και την τελική κορυφή t.

2.5.2. Παραδείγματα

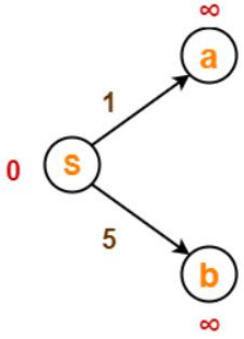
Στις παρακάτω εικόνες θα εξηγήσουμε τον αλγόριθμο του Dijkstra βήμα προς βήμα αναλυτικά (Εικόνα 12.) με ενδεικτικά σχήματα και πίνακες:



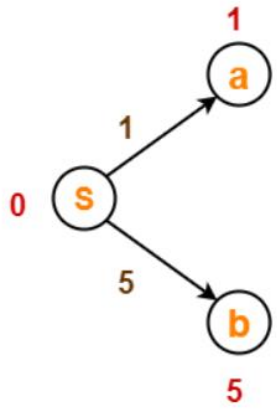
Εικόνα 13. Στο παράδειγμά μας ξεκινάμε από την αρχική κορυφή (s) και ψάχνουμε το πιο σύντομο μονοπάτι ώστε να φτάσουμε στον προορισμό μας την κορυφή e (target)

Βήμα 1^{ο}: Αρχικά τοποθετούμε ετικέτες που προσδιορίζουν την απόσταση μεταξύ των δυο κορυφών σε κάθε κορυφή του γραφήματός μας, όπου η αρχική θα έχει ετικέτα 0 μιας και είναι η κορυφή από την οποία θα ξεκινήσουμε και όλες οι υπόλοιπες θα έχουν ετικέτα το άπειρο μιας και δεν γνωρίζει αρχικά ο αλγόριθμος την απόσταση της αρχικής κορυφής από τις υπόλοιπες.

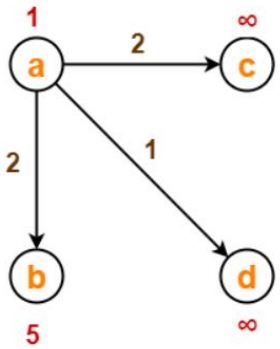
Σε κάθε βήμα του αλγορίθμου θα έχουμε μια λίστα ονόματι S θα περιέχει τις κορυφές που έχει επισκεφτεί ο Αλγόριθμος μας. Αρχικά η λίστα μας είναι άδεια και όταν ξεκινήσουμε την εκτέλεση του Αλγορίθμου θα έχει μόνο την κορυφή s. Άρα S= {s} στο πρώτο βήμα.



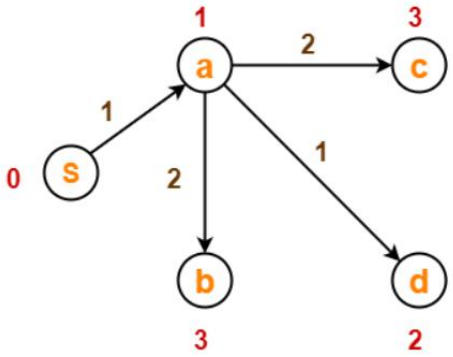
Βήμα 2ο: Σε αυτό το σημείο ο Αλγόριθμος Dijkstra θα εξετάσει τις γειτονικές του κορυφές μόνο και θα εκτελέσει μια λειτουργία decrease-key μειώνοντας την ετικέτα (key) τους μιας και γνωρίζει πλέον την απόσταση κάθε μιας από την αρχική κορυφή s. Τώρα ο Αλγόριθμος θα επισκεφθεί την κορυφή με την μικρότερη ετικέτα (μικρότερη απόσταση). Στην προκειμένη περίπτωση είναι η κορυφή (a) που έχει απόσταση ίση με 1.



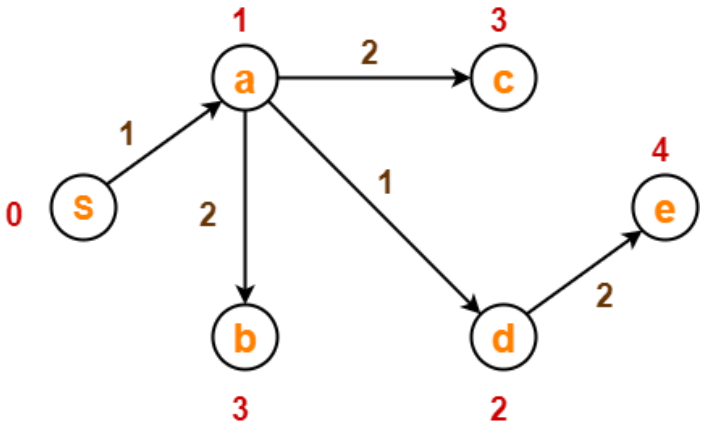
Βήμα 3ο: Ο αλγόριθμος επισκέπτεται την κορυφή 1 αφού έχει την μικρότερη ετικέτα από τις υπόλοιπες γειτονικές και η κορυφή θα προστεθεί στην λίστα S επειδή την έχουμε επισκεφθεί. S= {s, a} πλέον. Τώρα θα ασχοληθούμε με τις γειτονικές κορυφές μόνο της a. Αυτές είναι οι b, c, d με τις δυο τελευταίες να έχουν ως ετικέτα το άπειρο μιας και δεν τις γνωρίζουμε ακόμη.



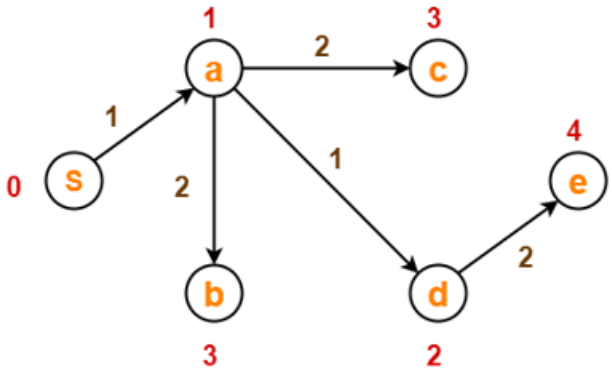
Βήμα 4ο: Πλέον θα έχουμε χαλάρωμα ακμών για τις γειτονικές κορυφές της a. Σε αυτό το σημείο η a θα επισκεφτεί την κορυφή με την μικρότερη απόσταση δηλαδή την d. Αφού την επισκεφτεί θα προστεθεί και αυτή στην λίστα S ={s,a ,d}



Βήμα 5ο: Η διαδικασία συνεχίζεται για κάθε επανάληψη του αλγορίθμου με την ίδια λογική όσον αφορά το χαλάρωμα ακμών και την λίστα S. Τώρα S= {s, a, d, b}.



Βήμα 6ο: Η διαδικασία επαναλαμβάνεται με τον ίδιο τρόπο και στα επόμενα βήματα μέχρι να καταλήξουμε στην τελική κορυφή (e)



Στο παραπάνω σχήμα φαίνεται και η τελική μορφή της συντομότερης διαδρομής που δημιουργείται μίας και έχουν επισκεφτεί όλες οι κορυφές από τον Αλγόριθμό μας και έχουμε καταλήξει στην τελική κορυφή e.

Όπως είδαμε και στις παραπάνω εικόνες η υλοποίηση του Αλγορίθμου έχει γίνει με την χρήση λιστών για την αποθήκευση των κορυφών του γραφήματος. Αυτό έχει ως αποτέλεσμα η συνολική χρονική πολυπλοκότητα να είναι της τάξης του O(n^2), όπου n=το πλήθος των κορυφών του γραφήματος. Αυτός ο χρόνος προκύπτει αν για κάθε κορυφή που επισκέπτεται ο Αλγόριθμος κατά την αναζήτηση του Shortest Path απαιτείται χαλάρωμα ακμών, δηλαδή να μειώνεται η ετικέτα του κόμβου μας. Αυτή η διαδικασία χρειάζεται χρόνο ίσο με O (n) για κάθε μια από τις n κορυφές του γραφήματος.

Άρα time = n* O(n)= O(n^2)

2.4.3. A*

Άλλος ένας τρόπος επίλυσης του προβλήματος συντομότερων διαδρομών είναι με την υλοποίηση ενός Αλγορίθμου αναζήτησης συντομότερων διαδρομών. Ο αλγόριθμος αναζήτησης A* είναι ένας τέτοιος αλγόριθμος ο οποίος χρησιμοποιείται συχνά για προβλήματα συντομότερων διαδρομών και βασίζεται στην χρήση ευρετικών μεθόδων (ευρετική αναζήτηση). Η ευρετική αναζήτηση βασίζεται στην καλύτερη γνώση του γραφήματος στο οποίο θέλουμε να επιλύσουμε το πρόβλημα επειδή γίνεται αναζήτηση πολλών μονοπατιών ταυτόχρονα. Με λίγα λόγια ο A* αναζητά όλα τα μονοπάτια μέχρι την τελική κορυφή (όχι μόνο γειτονικές κορυφές) και εξετάζει αυτά που θεωρεί ότι είναι τα βέλτιστα μονοπάτια. [23]

Σε μορφή γραφήματος μπορεί να αναπαρασταθεί και ως δέντρο όπου στην ρίζα του είναι το μονοπάτι που ο Αλγόριθμος ήδη γνωρίζει και στους κόμβους παιδία του είναι τα πιθανά μονοπάτια μέχρι την τελική κορυφή (t).

Algorithm 2.2 $A^*(G, s, t, \pi_t)$

Require: graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, Vertices s and t
Ensure: $\text{parent}(t)$ encodes the shortest path from s to t

1: **for all** $v \in \mathbf{V}$ **do**
2: $\quad \text{parent}(v) \leftarrow \perp$
3: $\quad \text{state}(v) \leftarrow$ unreached
4: $\quad \text{dist}(v) \leftarrow \infty$
5: $\text{dist}(s) \leftarrow 0$
6: $\text{state}(s) \leftarrow$ reached
7: **while** vertex v with $\text{state}(v) =$ reached exists and $\text{state}(t) \neq$ reached **do**
8: $\quad$ Select $v \in \mathbf{V}$ with $\text{state}(v) =$ reached and minimal $\widehat{\text{cost}}(v) = \text{dist}(v) + \pi_t(v)$
9: $\quad$ **for all** $u \in \mathbf{V}$ with $(v, u) \in \mathbf{E}$ **do**
10: $\quad\quad$ **if** $\text{dist}(v) + \text{len}(v, u) + \pi_t(u) < \text{dist}(u) + \pi_t(u)$ **then**
11: $\quad\quad\quad \text{parent}(u) \leftarrow v$
12: $\quad\quad\quad \text{dist}(u) \leftarrow \text{dist}(v) + \text{len}(v, u)$
13: $\quad\quad\quad \text{state}(u) \leftarrow$ reached
14: $\quad \text{state}(v) \leftarrow$ settled

Εικόνα 14. Ψευδοκώδικας για τον αλγόριθμο A*

Αναλυτική περιγραφή του αλγορίθμου βλέπουμε στην παραπάνω εικόνα (εικόνα 14.) και θα αναλύσουμε την διαδικασία επίσκεψης των κορυφών. Η διαφορά με τον Αλγόριθμο του Dijkstra είναι ότι πραγματοποιείται εκτίμηση από την κορυφή u προς την τελική κορυφή t [24]. Η συνάρτηση απόστασης σε αυτή την περίπτωση είναι ίση με cost(v) = dist(v) + πt(v) ώστε να λαμβάνεται υπόψη και η εκτίμηση για την συντομότερη απόσταση προς τον προορισμό , από την ευρετική συνάρτηση π. Ακολουθεί ανάλυση των συναρτήσεων που χρησιμοποιήσαμε:

- Η συνάρτηση dist() εκφράζει το κόστος μετακίνησης από τον αρχικό κόμβο στον τρέχων. Κοινώς, είναι το άθροισμα όλων των κόμβων που έχει επισκεφθεί ο αλγόριθμος έως εκείνη την στιγμή.
- Η συνάρτηση πt() είναι η γνωστή ευρετική συνάρτηση που καθορίζει τον A*. Πρακτικά είναι το εκτιμώμενο κόστος μετάβασης από τον αρχικό στον τελικό κόμβο. Το πραγματικό κόστος ωστόσο, δεν μπορεί να υπολογισθεί πριν τρέξει ολοκληρωμένα ο αλγόριθμος. Με την πt() διασφαλίζουμε ότι δεν υπάρχει υπερεκτίμηση κόστους.
- Η συνάρτηση cost() είναι το άθροισμα της dist() και της πt().

2.5. Αλγόριθμος του Dijkstra και ουρές προτεραιότητας

Στα προηγούμενα παραδείγματα είδαμε πως λειτουργεί ο Αλγόριθμος συντομότερων διαδρομών του Dijsktra για κατανομή των κόμβων σε λίστες. Αυτή η υλοποίηση μπορεί να εκτελεστεί σε καλύτερο χρόνο από αυτόν που είδαμε αν χρησιμοποιήσουμε τις κατάλληλες δομές δεδομένων [1],[2]. Η λύση είναι η ταξινόμηση των στοιχείων (κόμβων) σε ουρές προτεραιότητας προτού ξεκινήσει η εκτέλεση του Dijkstra.

Αρχικά δημιουργούμε την ουρά προτεραιότητας και εισάγουμε τα στοιχεία – κόμβους μαζί με τις αποστάσεις τους από την αρχική κορυφή (s). Οι αποστάσεις αυτές θα είναι και τα κλειδιά για κάθε στοιχείο της ουράς προτεραιότητας. Έτσι το στοιχείο που θα απέχει το λιγότερο από την αρχική κορυφή θα είναι και το πρώτο στην ουρά μας.

Τώρα θα εξάγουμε το στοιχείο με την μικρότερη απόσταση ώστε να μπορεί να το επισκεφθεί ο αλγόριθμος. Αφού το επισκεφτεί αυτό θα πάει σε έναν πινακα s που θα περιέχει τα στοιχεία – κόμβους που έχουμε επισκεφτεί με τον αλγόριθμο (αναλυτική λειτουργία στην εικόνα 16.). Ανάλογα με τον τύπο της ουράς θα μειώσουμε την ετικέτα κλειδί της κορυφής για κάθε βήμα του αλγορίθμου.

Διακρίνουμε 2 κατηγορίες υλοποιήσεων με αυτή τη μέθοδο:

1. Dijkstra-Dec: Υλοποιούμε τον Αλγόριθμο με τη χρήση των ουρών προτεραιότητας, που υποστηρίζουν την λειτουργία decrease-key. Σε αυτή την περίπτωση για κάθε βήμα του αλγορίθμου το κλειδί (ετικέτα) του κόμβου θα μειώνεται αφού την επισκεφτεί ο Dijkstra. Κάποιες ενδεικτικές ουρές της κατηγορίας αυτής είναι οι παρακάτω:
    i. Binary Heap (Κανονικός δυαδικός σωρός)
    ii. Buffer Heap
    iii. Pairing Heap (Σωρός ταιριασμάτων)

2. Dijkstra-NoDec: Υλοποίηση με ουρές που δεν υποστηρίζουν την λειτουργία Decrease-key -άρα δεν αλλάζουν την τιμή του κλειδιού των κορυφών που επισκέπτεται ο αλγόριθμος - , αλλά μόνο τις λειτουργίες εισαγωγής (Insert)και διαγραφής στοιχείων (Delete, Delete-min). Ενδεικτικά παραδείγματα τέτοιων ουρών είναι οι εξής :
    i. Sequence Heap
    ii. Binary Heap (δίχως decrease-key)
    iii. Bottom-Up Binary Heap (bottom-up insert, delete)

3. Dijkstra-Ext: Στις προηγούμενες δυο κατηγορίες μιλήσαμε για τους αλγόριθμους που αναφέρονται σε πράξεις που εκτελούνται εντός του πυρήνα της μνήμης. Όμως μπορούμε να εκτελούμε και πράξεις εκτός του πυρήνα με τον Dijkstra-Ext. Η συγκεκριμένη κατηγορία έχει και αυτή δυο εκδοχές αναφορικά με την εκτέλεση της λειτουργίας Decrease-key και μπορεί να εφαρμοστεί και σε μη-κατευθυνόμενα γραφήματα. Στις πράξεις εκτός του πυρήνα είναι πιο αποδοτικός και πιο γρήγορος από τους άλλους 2 ενώ εντός του πυρήνα η αποδοτικότητα του μειώνεται αισθητά και είναι πολύ πιο αργός από τους άλλους δυο αλγορίθμους.

Αν συγκρίνουμε τις 2 υλοποιήσεις του Αλγορίθμου θα συμπεράνουμε ότι ο Dijkstra-NoDec είναι πιο γρήγορος και με μικρότερο κόστος υλοποίησης μιας και δεν απαιτείται χρόνος για υλοποίηση των λειτουργιών decrease-key σε αντίθεση με τον Dijkstra-Dec, όμως παρόλα αυτά στον Dijkstra-Dec εκτελούνται λιγότερες λειτουργίες σωρού.

Input: Graph $G = (V, E)$, directed or undirected; positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u.
procedure DIJKSTRA(G, l, s)

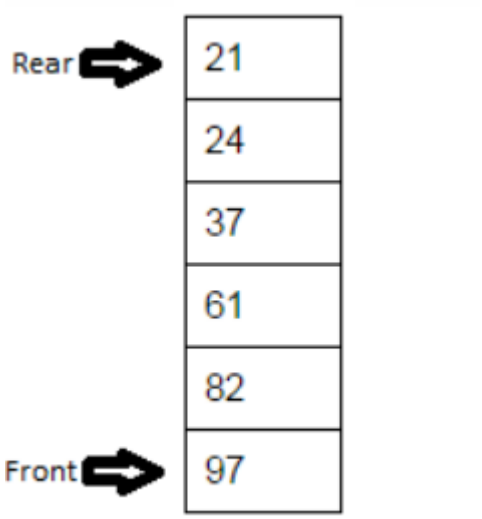
```
for all u ∈ V do
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = MAKEQUEUE(V)                          ▷ using dist-values as keys
while H is not empty do
    u = DELETEMIN(H)
    for all edges (u, v) ∈ E do
        if dist(v) > dist(u) + l(u, v) then
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            DECREASEKEY(H, v)
```

Εικόνα 15. Ψευδοκώδικας για την υλοποίηση του Αλγορίθμου του Dijkstra με χρήση ουράς προτεραιότητας.

Η αναλυτική υλοποίηση του Αλγορίθμου του Dijkstra φαίνεται στο παραπάνω σχήμα (Εικόνα 15.) όπου έχουμε ένα γράφημα G =(V, E) και δημιουργείται μια ουρά προτεραιότητας H που περιέχει όλες τις κορυφές του γραφήματος ταξινομημένες κατά

ελάχιστη απόσταση. Επιπλέον για κάθε χαλάρωμα ακμών κατά την εκτέλεση του αλγορίθμου γίνεται ανανέωση των αποστάσεων των ακμών.

Για την καλύτερη κατανόηση μπορούμε να δούμε πως θα λειτουργούσε αυτή η τεχνική με χρήση ουράς προτεραιότητας παρόμοιας με αυτήν που αναφέραμε ως παράδειγμα στο προηγούμενο κεφάλαιο (εικόνα 16.):



Εικόνα 16. Ενδεικτική ουρά προτεραιότητας που υλοποιείται για βελτιωμένη απόδοση του Αλγορίθμου του Dijkstra.

Έστω ότι στην ουρά η τιμή του κλειδιού του κάθε στοιχείου αναπαριστά την απόσταση των κορυφών από την αρχική (s). Έτσι ο αλγόριθμος γνωρίζει ποια κορυφή είναι πιο κοντά στην s και την επισκέπτεται.

Όταν επισκέπτεται μια κορυφή της ουράς εκτελεί παράλληλα και μια λειτουργία delete-min μιας και την εξάγει από την ουρά. Προτού επισκεφτεί την επόμενη εκτελεί και μια λειτουργία decrease-key αν υπάρχει χαλάρωμα ακμών και κυρίως αν η ουρά που επιλέξαμε μπορεί να εκτελέσει την παραπάνω λειτουργία. Η ίδια διαδικασία θα εκτελείται για κάθε βήμα του Αλγορίθμου μέχρι να φτάσουμε στην τελική κορυφή (s)

Όπως βλέπουμε ο χρόνος εκτέλεσης μειώνεται σε σχέση με την εκτέλεση του Dijkstra με λίστα ή πίνακα μιας και ο χρόνος για την επίσκεψη της κάθε κορυφής του γραφήματος μειώνεται στο ελάχιστο.

ΚΕΦΑΛΑΙΟ 3.ΟΥΡΈΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ ΠΟΥ ΜΕΛΕΤΗΘΗΚΑΝ

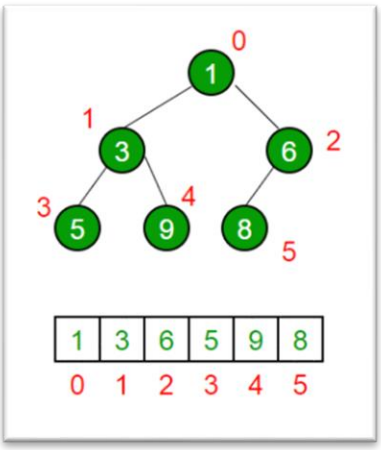
3.1. Binary Heap

Ο δυαδικός σωρός (Binary Heap) ταξινομεί τα στοιχεία σε ένα πλήρες δυαδικό δέντρο (Complete Binary Tree- εικόνα 17.) όπου όλα τα επίπεδα του να είναι πλήρη από κόμβους εκτός του τελευταίου και του πρώτου [2],[17],[18].Ο σωρός αυτός εμφανίζεται σε τρεις διαφορετικές παραλλαγές:

1.)STANDARD: Είναι ο κανονικός σωρός και περιλαμβάνει όλες τις βασικές λειτουργίες όπως εισαγωγή (Insert), διαγραφή (Delete, Delete-Min) και αφαίρεση κλειδιού (Decrease-key). Η Χρονική πολυπλοκότητα είναι η εξής: Insert, Delete, Delete-Min, Decrease-key: O(logn), όπου n= το πλήθος των στοιχείων του σωρού

2.)BINARY HEAP without Decrease-key: Απλούστερη εκδοχή του κανονικού δυαδικού Σωρού με την διαφορά ότι δεν υποστηρίζει την λειτουργία decrease-key, αλλά μόνο λειτουργίες εισαγωγής και διαγραφής στοιχείων (insert, delete).

3.)BOTTOM-UP BINARY HEAP: Δυαδικός σωρός όπου οι λειτουργίες Delete-Min εκτελούνται από το τελευταίο στοιχείο (bottom) του σωρού στο πρώτο (up) με αποτέλεσμα την μείωση των συγκρίσεων των στοιχείων στο μισό. Η εκδοχή αυτή εκτελεί μόνο λειτουργίες εισαγωγής και διαγραφής.



Εικόνα 17. Δυαδικό δέντρο και ο αντίστοιχος πίνακας των στοιχείων του ταξινομημένα βάσει της τιμής τους

Βασικές Λειτουργίες:

- Insert: Εισαγωγή στοιχείου που εκτελείται ως εξής:
    - Αρχικά εισάγουμε το νέο στοιχείο μας στο κάτω αριστερά φύλλο του δυαδικού δέντρου
    - Για να δούμε ποια θα είναι η τελική του θέση θα κάνουμε συγκρίσεις με τον γονέα του, δηλαδή τον κόμβο που βρίσκεται ακριβώς από πάνω του στο δέντρο.
        - Αν στην πρώτη σύγκριση το νέο κλειδί του νέου στοιχείου είναι μικρότερο από τον γονέα το, τότε παραμένει εκεί (do-nothing)
        - Αν είναι μεγαλύτερο από το φύλλο γονέα του τότε θα αλλάξουν θέσεις οι δύο κόμβοι. Η αλλαγή θέσεων θα συνεχιστεί μέχρι να βρεθεί γονέας όπου το κλειδί του να είναι μεγαλύτερο από του νέου στοιχείου. Τότε δεν θα χρειάζεται άλλη αλλαγή στο δέντρο μας μιας και είναι έτοιμο.

Αναφορικά με την πολυπλοκότητα της διαδικασίας υπάρχουν 2 περιπτώσεις:

- Αν το κλειδί του νέου στοιχείου είναι μικρότερο από τον γονέα του, τότε απαιτείται 1 σύγκριση, άρα O(1)
- Αν το κλειδί είναι μεγαλύτερο από του γονέα και κάνουμε συγκρίσεις τότε μπορεί να καταλήξουμε στην χειρότερη περίπτωση. Στην περίπτωση αυτή το κλειδί του νέου στοιχείου είναι μεγαλύτερο από κάθε κλειδί ενός γονέα του και γίνεται ρίζα του δέντρου. Αυτό σημαίνει ότι θα χρειαστεί να κάνουμε συγκρίσεις όπου το πλήθος τους ισούται με το ύψος του δέντρου, δηλαδή την απόσταση της ρίζας από το τελευταίο φύλλο, δηλαδή το νέο μας στοιχείο. Το ύψος αυτό ισούται με O(logn), όπου n: το πλήθος των στοιχείων κόμβων.

- Delete: Η διαδικασία διαγραφής ενός στοιχείου απαιτεί τις παρακάτω ενέργειες:
    - Αρχικά αναζητούμε το στοιχείο που θέλουμε να διαγράψουμε
    - Έπειτα το διαγράφουμε και αν δεν είναι φύλλο, δηλαδή να βρίσκεται στο πιο κάτω επίπεδο του δέντρου θα πρέπει να επιδιορθώσουμε το δέντρο μας
    - Κατά την επιδιόρθωση στην θέση του στοιχείου θα πάει το παιδί του (στοιχείο κάτω από αυτό) που θα έχει το μεγαλύτερο κλειδί
    - Και αυτή η διαδικασία απαιτεί το πολύ O(logn) χρόνο μιας και στην χειρότερη περίπτωση (worst case) η ρίζα είναι το στοιχείο που θα διαγραφεί

- Delete-Min: Η διαδικασία είναι ίδια με την κανονική διαγραφή, με μόνη διαφορά το ότι το στοιχείο που θέλουμε να διαγράψουμε είναι αυτό με το μικρότερο κλειδί

  Και σε αυτή την περίπτωση η χρονική πολυπλοκότητα είναι της τάξης του O(logn)

- Decrease-key: Κατά την εκτέλεση αλγορίθμων με χρήση του δυαδικού σωρού απαιτείται πολλές φορές η παρέμβαση στα κλειδιά των στοιχείων του δέντρου και συγκεκριμένα η μείωση τους. Η διαδικασία μείωσης κλειδιού έχει ως εξής:
  - Find: διαδικασία αναζήτησης και εύρεσης του ζητούμενου στοιχείου που θέλουμε να αλλάξουμε την τιμή του κλειδιού
  - Decrease: μειώνουμε την τιμή του κλειδιού του στοιχείου
  - Έπειτα στο δέντρο που έχει δημιουργηθεί θα κάνουμε τις απαραίτητες αλλαγές ώστε να το επιδιορθώσουμε και να ισχύει η σειρά προτεραιότητας των στοιχείων βάσει των τιμών των κλειδιών τους

  Το συνολικό κόστος της λειτουργίας αυτής είναι της τάξης του O(logn)

3.2.Pairing Heap

Για τον Pairing Heap (Σωρός ταιριάσματος) διακρίνουμε 4 διαφορετικές παραλλαγές του: τον σωρό 2 περασμάτων (2-pass) και πολλών περασμάτων (multi-pass) μαζί με τους βοηθητικούς τους σωρούς (Auxiliary) - οι οποίοι παρουσιάζουν ελάχιστα καλύτερους χρόνους εκτέλεσης από τους κανονικούς - με τον σωρό 2-εισοδων (2-pass) να είναι λίγο πιο αποδοτικός, εκδοχή της οποίας θα αναλύσουμε και τα αποτελέσματα για τους χρόνους των λειτουργιών [2], [19], [20]. Οι χρόνοι εκτέλεσης των λειτουργιών του Pairing Heap είναι πολύ κοντά με τους αντίστοιχους για τον Binary Heap.

Κύριες λειτουργίες:

Insert: $O(2^{\sqrt{\log \log n}})$

Delete, Delete-min: O(logn)

Decrease-key: $O(2^{\sqrt{\log \log n}})$

Meld (συγχώνευση): O (1)

Ο σωρός αυτός βασίζεται στην συγχώνευση επιμέρους σωρών για καλύτερη υλοποίηση.

Επεξήγηση βασικών λειτουργιών:

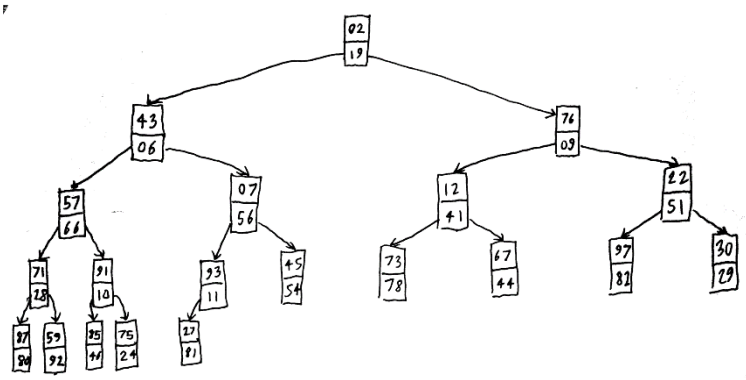
- Meld: Αρχικά θα εξηγήσουμε πως γίνεται η συγχώνευση 2 ή περισσότερων σωρών ανάλογα με τον τύπο του σωρού (2-pass ή Multi-pass). Για να γίνει αυτή η διαδικασία κοιτάζουμε μόνο τις ρίζες του κάθε σωρού και αυτή με την μικρότερη τιμή κλειδιού, άρα και μεγαλύτερη προτεραιότητα θα είναι η ρίζα του τελικού σωρού. Η ρίζα ή ρίζες των υπολοίπων σωρών θα είναι τα παιδιά της ρίζας στο τελικό δέντρο.
- Insert: η εισαγωγή ενός νέου στοιχείου στον σωρό θα γίνεται σαν μια διαδικασία συγχώνευσης του αρχικού σωρού με έναν δεύτερο που θα αποτελείται από 1 στοιχείο, αυτό που θέλουμε να εισάγουμε. Το νέο στοιχείο θα τοποθετηθεί ως παιδί της ρίζας του σωρού και αν η τιμή του κλειδιού του είναι μικρότερη από της ρίζας του σωρού θα γίνει αλλαγή και θα τοποθετηθεί εκείνο ως ρίζα. Η πολυπλοκότητα της λειτουργίας αυτής θα είναι $O(2^{\sqrt{\log \log N}})$ ενώ αν χρειάζεται να γίνει μόνο μια σύγκριση θα είναι O(1).
- Delete: η διαγραφή ενός στοιχείου θα γίνει όπως και στον Binary Heap. Θα βρούμε το στοιχείο που θέλουμε να διαγράψουμε (find) και όταν γίνει η διαγραφή θα κάνουμε swaps (αλλαγές θέσεων) μεταξύ των στοιχείων του σωρού μας μέχρι να επιδιορθωθεί το δέντρο και να τηρείται η προτεραιότητα των στοιχείων. Η

διαδικασία αυτή έχει ίδια χρονική πολυπλοκότητα με την αντίστοιχη διαδικασία του Binary Heap.

- Delete-min: η διαδικασία διαγραφής του στοιχείου με το μικρότερη κλειδί συνεπάγεται την διαγραφή της ρίζας του δέντρου του σωρού. Αφού διαγραφεί η ρίζα του δέντρου αυτό θα χωριστεί σε n δέντρα, όσα και τα παιδιά της αρχικής ρίζας με τα παιδιά να είναι οι ρίζες των επιμέρους υποδέντρων. Έπειτα θα κάνουμε συγχωνεύσεις μεταξύ των επιμέρους δέντρων και ως ρίζα του τελικού δέντρου θα θέσουμε την ρίζα των επιμέρους δέντρων με το μικρότερο κλειδί. Οι υπόλοιπες ρίζες θα τοποθετηθούν κάτω από την ρίζα μαζί με τα υποδέντρα τους. Η χρονική πολυπλοκότητα της συγκεκριμένης λειτουργίας απαιτεί χρόνο της τάξεως του O (logn), όπου n= το πλήθος των στοιχείων του σωρού.
- Decrease-key: Η διαδικασία μείωσης κλειδιού αποτελείται από τις τρεις επιμέρους λειτουργίες που αναφέραμε και στα προηγούμενα κεφάλαια. Αρχικά βρίσκουμε το στοιχείο στο οποίο θα εκτελέσουμε την διαδικασία, κάνουμε την μείωση και έπειτα επιδιορθώνουμε το δέντρο ώστε να τηρείται η προτεραιότητα που έχουμε ορίσει. Η διαδικασία αυτή απαιτεί O (logn) χρόνο.

3.3. Sequence Heap

Μέχρι τώρα είχαμε δει ουρές προτεραιότητας μιας διάστασης. Υπάρχουν όμως και οι πολυδιάστατες ουρές προτεραιότητας (δομές δεδομένων) που θα αναλύσουμε σε αυτή την ενότητα [3], [22]. Η πιο χαρακτηριστική είναι η K-N heap ή αλλιώς K-D heap (tree). Για μια K-N heap ορίζουμε ως n: το πλήθος των στοιχείων του σωρού και k: το πλήθος των κλειδιών του κάθε στοιχείου. Ο σωρός αυτός περιέχει μεγαλύτερο πλήθος κλειδιών από ότι ένας κανονικός σωρός μίας διάστασης με ίδιο πλήθος στοιχείων. Αυτή η ιδιότητα τον κάνει να είναι κατάλληλος για γραφήματα μεγάλου μεγέθους μιας και το k μπορεί να είναι πολύ μεγάλο.



Εικόνα 18. Ακολουθιακός σωρός ως δέντρο με 20 στοιχεία εκ των οποίων καθένα περιέχει από 2 κλειδιά

Στο παραπάνω παράδειγμα (εικόνα 18.) διακρίνουμε έναν σωρό 2 διαστάσεων (2-D) όπου στην σχεδίαση του δέντρου της υπάρχουν 2 στοιχεία ανά φύλλο. Μπορούμε να εκτελέσουμε μια λειτουργία σε πάνω από ένα στοιχείο κάθε φορά, όπως να εισάγουμε 2 και περισσότερα στοιχεία σε ένα φύλλο του δέντρου ή αντίστοιχα να τα διαγράψουμε.

Η σχεδίαση της ουράς είναι ως ένα πλήρες δυαδικό δέντρο n: στοιχείων όπου καθένα από αυτά έχει k: κλειδιά (keys) και στην ρίζα του δέντρου είναι το σύνολο στοιχείων που περιέχει το στοιχείο με το μικρότερο κλειδί. Όπως και στην εικόνα πιο πάνω βλέπουμε ότι η ελάχιστη τιμή κλειδιού είναι 2 και για αυτό τον λόγο έχει τοποθετηθεί ως ρίζα του δέντρου

Ενδεικτικές λειτουργίες και οι αντίστοιχοι Χρόνοι Εκτέλεσης:

- Insert: αρχικά εισάγουμε το στοιχείο με τα k: κλειδιά του στο κάτω αριστερά μέρος του δέντρου μας. Θεωρητικά σε εκείνη την θέση βρίσκεται το μικρότερο στοιχείο του σωρού. Μετά θα το συγκρίνουμε με τον γονέα του και αν η τιμή του κλειδιού του είναι η μικρότερη από τις τιμές των κλειδιών των δυο στοιχείων θα γίνει αλλαγή της θέσης τους στο δέντρο. Οι συγκρίσεις θα συνεχιστούν μέχρι να βρεθεί στοιχείο του σωρού όπου η τιμή κάποιου κλειδιού του θα είναι μικρότερη από την τιμή του κλειδιού του στοιχείου που έχουμε εισάγει τελευταίο. Η χειρότερη χρονική πολυπλοκότητα της πράξης αυτής εμφανίζεται στο σενάριο που το στοιχείο που εισήχθη στον σωρό μας έχει κάποιο κλειδί με την μικρότερη τιμή από όλα τα υπόλοιπα. Σε αυτή την περίπτωση το καινούργιο στοιχείο θα είναι η ρίζα του καινούργιου σωρού που έχει δημιουργηθεί. Σε αυτή την περίπτωση θα έχουμε τόσες συγκρίσεις όσες και το ύψος του δέντρου (σωρού), άρα O(logn).
- Delete: η διαδικασία διαγραφής δεν διαφέρει από την διαγραφή των προηγούμενων δύο ουρών προτεραιότητας. Αρχικά αφού βρούμε το στοιχείο θα το διαγράψουμε και έπειτα θα επιδιορθώσουμε τον καινούργιο δέντρο που δημιουργείται ώστε να τηρεί την προτεραιότητα που έχουμε επιβάλλει. Άρα θα κάνουμε συνεχόμενες αλλαγές (swaps) μέχρι να επιτευχθεί η προτεραιότητα. Αυτή η διαδικασία θα έχει μέγιστη χρονική πολυπλοκότητα O(logn)
- Delete-min: η διαδικασία διαγραφής του στοιχείου με το μικρότερη κλειδί (delete-min) είναι ουσιαστικά η διαγραφή της ρίζας του δέντρου. Αφού διαγραφεί η ρίζα του δέντρου αυτό θα χωριστεί σε 2 δέντρα, μιας και η δομή του K-N είναι ένα πλήρες δυαδικό δέντρο. Έπειτα θα κάνουμε συγχώνευση των δύο επιμέρους δέντρων που δημιουργούνται και θα ορίσουμε ως ρίζα του τελικού δέντρου την ρίζα των επιμέρους δέντρων που θα περιέχει το στοιχείο του οποίου κάποιο από τα κλειδιά θα έχει την μικρότερη τιμή από τα υπόλοιπα. Η ρίζα του δεύτερου δέντρου θα τοποθετηθεί κάτω από την ρίζα μαζί με τα αντίστοιχο υποδέντρα τους. Η χρονική πολυπλοκότητα της συγκεκριμένης λειτουργίας απαιτεί χρόνο της τάξεως του O (logn), όπου n= το πλήθος των στοιχείων του σωρού
- Όπως αναφέραμε και στην εκφώνηση οι δομές του τύπου K-N δεν εκτελούν την λειτουργία μείωσης κλειδιού (decrease-key) πάρα μόνο τις παραπάνω λειτουργίες εισαγωγής και διαγραφής στοιχείων.

ΚΕΦΑΛΑΙΟ 4. ΖΗΤΗΜΑΤΑ ΥΛΟΠΟΙΗΣΗΣ

Για να υλοποιηθεί πιο αποτελεσματικά η διπλωματική εργασία χρησιμοποιήθηκαν κάποιες τεχνολογίες και open-source εργαλεία και εφαρμογές.

4.1. Υπολογιστής

Ο υπολογιστής που χρησιμοποιήθηκε για την υλοποίηση και την εκπόνηση της διπλωματικής εργασίας ήταν το Laptop μου μάρκας DELL , μοντέλου Dell Vostro 15 3000, επεξεργαστή Intel core i3 (3.00 GHz) και λογισμικού Windows

Τα προγράμματα υλοποιήθηκαν και εκτελέστηκαν σε λειτουργικό σύστημα linux μέσω εικονικής μηχανής (Virtual machine) μιας και το συγκεκριμένο λειτουργικό σύστημα παρείχε όλες τις απαραίτητες εντολές. Κατέβασα την εικονική μηχανή από την επίσημη ιστοσελίδα της Oracle (https://www.virtualbox.org/)

Ως εικονική μηχανή (virtual machine) ορίζουμε μια “εφαρμογή” που μπορεί να λειτουργεί με διαφορετικό λειτουργικό σύστημα από αυτό που τρέχει ο υπολογιστής μας. Για παράδειγμα ο υπολογιστής ενός χρήστη μπορεί να έχει λειτουργικό σύστημα Windows αλλά με την virtual machine να εκτελεί προγράμματα σε Linux. Το μόνο που πρέπει να κάνει είναι να κατεβάσει το αντίστοιχο εκτελέσιμο αρχείο .iso και να το εκτελέσει κατά την δημιουργία της εικονικής μηχανής.

4.2. Εργαλεία Microsoft

Για την συγγραφή του κειμένου της διπλωματικής εργασίας χρησιμοποιήθηκαν τα εξής εργαλεία της Microsoft:

1.)Word για την συγγραφή του πλήρους κειμένου της διπλωματικής εργασίας

2.)PowerPoint για την παρουσίαση των papers και για την τελική παρουσίαση του ολοκληρωμένου κειμένου

3.)Excel για την καταγραφή των αποτελεσμάτων των υλοποιήσεων σε πίνακες και για την καλύτερη ταξινόμηση

4.3. Γλώσσα υλοποίησης και editors

Η γλώσσα προγραμματισμού στην οποία υλοποιήθηκαν τα προγράμματα είναι η γλώσσα C++ για αρχεία τύπου .cpp και .h (C++ και C/object files) ενώ για την επεξεργασία του πηγαίου κώδικα χρησιμοποιήθηκε ο Text Editor που μας παρείχε το Linux. Η C++ είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου που χρησιμεύει στην ανάπτυξη και υλοποίηση εφαρμογών και αποτελεί εξέλιξη της αρχικής γλώσσας C, ενώ παράλληλα έχει και χαρακτηριστικά άλλων γλωσσών. Μια εκ των γλωσσών είναι η JAVA που ασχολείται με τον αντικειμενοστραφή προγραμματισμό.

Τα αρχεία στα οποία έχουμε εργαστεί είναι της μορφής .c ή .cpp αλλά και header files (.h) που είναι αρχεία που έχουν έτοιμες εντολές και συναρτήσεις και εισάγονται σε ένα αρχείο .cpp μέσω include. Έτσι αν θέλουμε το αρχείο του dijksta με όνομα results.cpp να περιέχει τις συναρτήσεις της Sequence Heap θα κάνουμε include το αρχείο sequence_heap.h (#include /path/<sequence_heap.h>) με το πλήρες μονοπάτι του, ώστε να μην χρειάζεται η επανάληψη του κώδικα της Sequence heap, με αποτέλεσμα την εξοικονόμηση χώρου και χρόνου.

4.4. Βιβλιοθήκες

Πολλοί κώδικες χρησιμοποιήθηκαν από την βιβλιοθήκη pgl-master που μου δόθηκε από τους διδάσκοντες και υπεύθυνους για την εργασία. Η εν λόγω βιβλιοθήκη παρέχει πολλούς κώδικες και δομές δεδομένων που ήταν χρήσιμες ώστε να υλοποιηθούν πιο γρήγορα και αποτελεσματικά οι ουρές προτεραιότητας στον αλγόριθμο. Μια ακόμη βιβλιοθήκη που χρησιμοποιήθηκε είναι η STL που παρέχει πρόσβαση σε πολλούς κώδικες και έτοιμες υλοποιήσεις. Τέλος για επιπλέον αριθμό ουρών προτεραιότητας χρησιμοποιήθηκε η βιβλιοθήκη priority-queue-testing που ήταν έτοιμη και ανοιχτής πρόσβασης (open-source) από το google και παρείχε πλήθος ετοίμων υλοποιήσεων ουρών προτεραιότητας.

4.5. Εργαλεία και εφαρμογές

Για την δημιουργία διαγραμμάτων των ουρών για τους χρόνους και το πλήθος των settled_nodes ανά query χρησιμοποιήθηκε η open-source εφαρμογή του canvas από τον ιστότοπο (https://www.canva.com/graphs/) που έχει πληθώρα templates για την δημιουργία γραφημάτων και σχημάτων.

ΚΕΦΑΛΑΙΟ 5:ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ

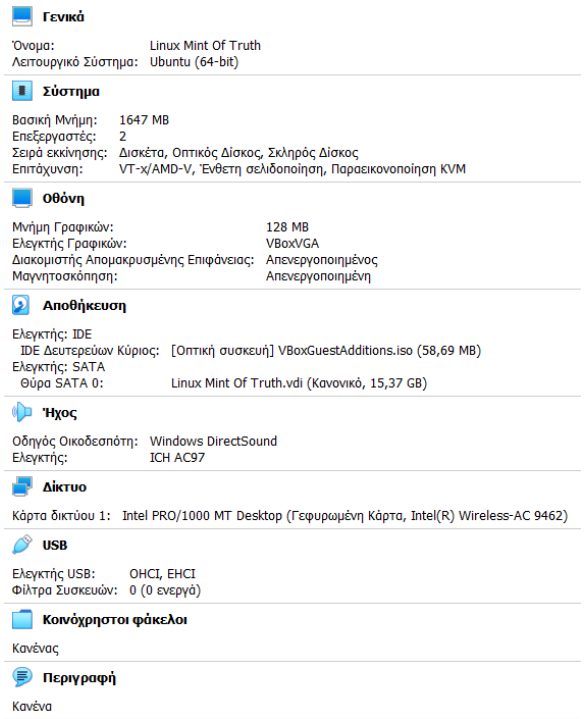
5.1. Περιβάλλον υλοποίησης

5.1.1. Λειτουργικό σύστημα

Για την κατάλληλη εκτέλεση και καταγραφή των αποτελεσμάτων χρησιμοποιήθηκε το λειτουργικό σύστημα των Linux και πιο συγκεκριμένα το Linux mint το οποίο μεταφορτώθηκε από την επίσημη ιστοσελίδα του Linux: https://www.linuxmint.com/download.php

(linuxmint-20.3-cinnamon-64bit.iso).

Για την εγκατάσταση του χρησιμοποιήθηκε μια εικονική μηχανή της Oracle VM VirtualBox η οποία είχε τις παρακάτω προδιαγραφές (εικόνα 20.):



Εικόνα 19. Ενδεικτική εικόνα των προδιαγραφών και των χαρακτηριστικών της εικονικής μηχανής

Το κατέβασμα και η υλοποίηση της μηχανής έγιναν από την επίσημη ιστοσελίδα της Oracle στο κεφάλαιο του Virtual Box.

Όπως βλέπουμε και στην παραπάνω εικόνα το σύστημα μας έχει τις εξής προδιαγραφές:

Για το σύστημα μνήμης έχουμε:

- Βασική μνήμη αποθήκευσης μεγέθους 1647 MegaBytes
- Δυο επεξεργαστές

Για την οθόνη μας:

- Μνήμη γραφικών μεγέθους 128 MegaBytes
- Ελεγκτή γραφικών τύπου VBoxVGA

Για την αποθήκευση δεδομένων:

- Ελεγκτή IDE: ο δευτερεύων IDE Κύριος ελήφθη από το αρχείο VBoxGuestAdditions.iso
- Ελεγκτή SATA: θύρα SATA 0 μεγέθους 15,37 GigaBytes

Οι παραπάνω προδιαγραφές έχουν επιλεχθεί βάσει των προτεινόμενων από τον Υπολογιστή μου ώστε να μπορεί να λειτουργεί η εικονική μηχανή με τον πιο αποτελεσματικό τρόπο χωρίς προβλήματα μνήμης.

5.1.2. Μεταγλώττιση

Προτού το αρχείο μας εκτελεστεί πρέπει να περάσει πρώτα από το στάδιο της μεταγλώττισης και αποσφαλμάτωσης του. Σε αυτήν την περίπτωση έχουμε τον g++ compiler του εργαλείου του Terminal. Πιο συγκεκριμένα οι εντολές μεταγλώττισης ήταν της παρακάτω μορφής:

- g++ file.cpp -std=c++17 -O3 -I$(INCLUDEDIR) -I$(BOOSTINCLUDEDIR) -L$(BOOSTLIBDIR) -DNDEBUG –lboost_program_options, όπου file.cpp είναι ο ολοκληρωμένος κώδικας υλοποίησης του Αλγορίθμου με την εφαρμογή της ουράς προτεραιότητας σε αυτό.
- INCLUDEDIR: η βιβλιοθήκη των include όπου περιέχει κάποιες δομές δεδομένων για την υλοποίηση των προγραμμάτων μας

Η παραπάνω εντολή μεταγλώττισης υπάρχει έτοιμη σε ένα αρχείο τύπου makefile το οποίο μπορούμε να καλέσουμε με την εντολή make του Terminal. Έπειτα από αυτό το βήμα και στην περίπτωση που δεν υπάρχει κανένα σφάλμα μεταγλώττισης (error) τότε το πρόγραμμα μας είναι έτοιμο να εκτελεστεί (run).

Στην περίπτωση μας έχουμε ένα αρχικό αρχείο όπου γίνεται δημιουργία του τυχαίου πλήθους από queries (queryGen.cpp) και ένα δεύτερο αρχείο που εκτελεί τον Αλγόριθμο (results.cpp) με τα queries που έχουν δημιουργηθεί από το προηγούμενο.

5.1.3. Εκτέλεση

Αφού ολοκληρωθεί η μεταγλώττιση και διόρθωση του κώδικα του κάθε αρχείου, δημιουργείται ένα executable file με όνομα a.out. Για την εκτέλεση του αρχείου χρησιμοποιήθηκε η εντολή των Linux ./a.out με τις κατάλληλες παραμέτρους εκτέλεσης.

Για το αρχείο queryGen.cpp έχουμε τις εξής παραμέτρους

- -q: Καθορίζει τον αριθμό των queries που ζητούνται σε κάθε περίπτωση εκτέλεσης του αλγορίθμου
- -f: Η μορφή του γραφήματος δεδομένων όπου θα εκτελεστεί ο αλγόριθμος (αν είναι της μορφής DIMACS9 ή DIMACS10.
- -m: άνοιγμα του αρχείου των δεδομένων μας, απαιτείται απλά το όνομα του χάρτη που περιέχει τα δεδομένα , όπως για παράδειγμα “NY” δεδομένου ότι παίρνουμε το πακέτο δεδομένων NY (θα μιλήσουμε πιο αναλυτικά για αυτά σε επόμενο κεφάλαιο- πειραματικά δεδομένα) ή FLA για το πακέτο δεδομένων FLA.

Για το αρχείο results.cpp έχουμε τις παρακάτω παραμέτρους:

- -a: καθορίζει τον τύπο του Αλγορίθμου που θα εκτελέσει. Εδώ διακρίνουμε 2 περιπτώσεις του Αλγορίθμου:
  - Plain Dijkstra (Single-Source)
  - A*(Astar)
- -f: Η μορφή του γραφήματος δεδομένων όπου θα εκτελεστεί ο αλγόριθμος (αν είναι της μορφής DIMACS9 ή DIMACS10.
- -g: Ορίζει τον τύπο γραφήματος των δεδομένων που θα πάρουμε. Η επιλογή που έκανα είναι το Forward Star (προτεινόμενη)
- -m: άνοιγμα του αρχείου των δεδομένων μας, απαιτείται απλά το όνομα του χάρτη που περιέχει τα δεδομένα , όπως για παράδειγμα “NY” δεδομένου ότι παίρνουμε το πακέτο δεδομένων NY (θα μιλήσουμε πιο αναλυτικά για αυτά σε επόμενο κεφάλαιο- πειραματικά δεδομένα) ή FLA για το πακέτο δεδομένων FLA.

5.1.4. Διαδικασία εκτέλεσης

Αρχικά πρέπει να κάνουμε compile το πρώτο αρχείο για την δημιουργία των queries που θα δημιουργηθούν. Αυτό θα γίνει μέσω της εντολής του τερματικού (Terminal) :

make file=queryGen.cpp.

```
nikos@nikos-VirtualBox:~/pgl/ResultGenerators/ShortestPathAlgResults$ make file=queryGen.cpp
g++ queryGen.cpp -O3 -march=native -I'/home/nikos/pgl/include/' -I'/usr/local/include' -L'/usr/loc
al/lib' -DNDEBUG -lboost_program_options
```

Από την στιγμή που δεν υπάρχει κάποιο σφάλμα κατά την μεταγλώττιση μπορούμε να προχωρήσουμε στην εκτέλεση του αρχείου με τις κατάλληλες παραμέτρους που μπορούμε να δούμε με την εντολή ./a.out

```
nikos@nikos-VirtualBox:~/pgl/ResultGenerators/ShortestPathAlgResults$ ./a.out
Supported options:
  -q [ --queries ] arg  number of queries.Default:1000
  -n [ --nodes ] arg    number of nodes. The format-map parameters can be used
                        instead.
  -f [ --format ] arg   map format. DIMACS10[0], DIMACS9[1]. Default:0
  -m [ --map ] arg      input map. The name of the map to read.
                        Default:'luxembourg'. Maps should reside in
                        '$HOME/Projects/Graphs/' and should consist of 2 files,
                        both with the same map name prefix, and suffixes
                        'osm.graph' and 'osm.xyz' in the case of DIMACS10 and,
                        '.gr' and '.co' in the case of DIMACS9.
```

Έπειτα μπορούμε να τρέξουμε το αρχείο μας επιλέγοντας τις παραμέτρους από την λίστα που βλέπουμε παραπάνω. Για παράδειγμα για πλήθος 20.000 queries και μορφή δεδομένων DIMACS9 από το διάγραμμα με όνομα NY η επόμενη εντολή θα έχει ως εξής:

```
nikos@nikos-VirtualBox:~/pgl/ResultGenerators/ShortestPathAlgResults$ ./a.out -q 20000 -f 1 -m NY

num nodes:264346
```

Όπως βλέπουμε έχουν δημιουργηθεί τα τυχαία queries και ως έξοδο έχουμε απλά το πλήθος των κόμβων του διαγράμματος μας. Τώρα θα κάνουμε compile το αρχείο που περιέχει τον κώδικα του αλγορίθμου με την ουρά προτεραιότητας σε αυτόν. Το αρχείο έχει όνομα results.cpp άρα η εντολή θα είναι make file=results.cpp

```
nikos@nikos-VirtualBox:~/pgl/ResultGenerators/ShortestPathAlgResults$ make file=results.cpp
g++ results.cpp -O3 -march=native -I'/home/nikos/pgl/include/' -I'/usr/local/include' -L'/usr/loca
l/lib' -DNDEBUG -lboost_program_options
```

Πλέον είναι έτοιμο το αρχείο για να εκτελεστεί σύμφωνα με τις παραμέτρους που θα επιλέξουμε από την λίστα μας που θα την δούμε ξανά με την εντολή ./a.out:

```
Allowed options:
  -a [ --shortest path algorithm ] arg  Choose: All[0], plain-dijkstra[1],
                                        plain-bidirectional[2], uni-A*-ecl[3],
                                        bid-A*-ecl-sym[4], bid-A*-ecl-max[5],
                                        bid-A*-ecl-ave[6], uni-A*-lmk[7],
                                        bid-A*-lmk-sym[8], bid-A*-lmk-max[9],
                                        bid-A*-lmk-ave[10], uni-A*-{lmk+ecl}[11
                                        ], bid-A*-{lmk+ecl}-sym[12],
                                        bid-A*-{lmk+ecl}-max[13],
                                        bid-A*-{lmk+ecl}-ave[14]. Default:0
  -g [ --graphtype ] arg                graph type. All[0], Adjacency List[1],
                                        Packed Memory Graph[2], forward
                                        Star[3]. Default:0
  -f [ --format ] arg                   map format. DIMACS10[0], DIMACS9[1].
                                        Default:0
  -m [ --map ] arg                      input map. The name of the map to read.
                                        Default:'luxembourg'. Maps should
                                        reside in '$HOME/Projects/Graphs/DIMACS
                                        {9,10}/' and should consist of 2 files,
                                        both with the same map name prefix, and
                                        suffixes 'osm.graph' and 'osm.xyz' in
                                        the case of DIMACS10 and, '.gr' and
                                        '.co' in the case of DIMACS9.
```

Τώρα για να εκτελέσουμε το αρχείο μας με τις κατάλληλες παραμέτρους, δηλαδή για την εκδοχή του Αλγορίθμου τον κανονικό Dijkstra, τύπο γραφήματος το Forward Star, μορφή δεδομένων DIMACS9 από το διάγραμμα με όνομα NY η επόμενη εντολή θα έχει ως εξής:

```
nikos@nikos-VirtualBox:~/pgl/ResultGenerators/ShortestPathAlgResults$ ./a.out -a 1 -g 3 -f 1 -m NY
Reading DIMACS9 from /home/nikos/Projects/Graphs/DIMACS9/NY/NY.gr
```

Έτσι το αρχείο είναι έτοιμο να εκτελεστεί και τα αποτελέσματα που θα έχουμε στην έξοδο είναι τα εξής:

Statistics	time(ms)	efficiency	spreading	distance(m)	SPNodes	settledNodes	visitedNodes
min:	0.004	0.0654704	0.0113488	9740	3	17	30
max:	499.396	23.0769	112.709	2.11394e+06	903	297935	297942
mean:	48.3607	0.344538	56.6859	669316	345.012	148843	149847
std:	31.0453	0.511604	32.7391	317268	159.548	86559.3	86544.6

Στην έξοδο βλέπουμε τα στατιστικά δεδομένα για τους χρόνους εκτέλεσης κάθε query, τις αποστάσεις τους αλλά και για τις ταξινομήσεις που γίνονται σε καθένα από τα queries. Επίσης δημιουργείται και ένα αρχείο με όνομα stats που έχει αναλυτικά τα παραπάνω στατιστικά για κάθε query Που εκτελείται. Με αυτό το αρχείο θα ασχοληθούμε κιόλας για την ομαδοποίηση των queries ανάλογα με την απόσταση τους

5.2. Πειραματικά δεδομένα

Για την εκτέλεση του Αλγορίθμου με τις ουρές προτεραιότητας έχουμε χρησιμοποιήσει πραγματικά δεδομένα από τον ιστότοπο του DIMACS9 http://www.diag.uniroma1.it//challenge9/download.shtml για να τρέξουμε τα προγράμματα μας και να καταγράψουμε τα στατιστικά που χρειαζόμαστε. Στην συγκεκριμένη ιστοσελίδα υπάρχει μια λίστα από φακέλους δεδομένων οδικών δικτύων μεγαλουπόλεων των Ηνωμένων Πολιτειών της Αμερικής με τις πραγματικές τους συντεταγμένες.

NY	New York City	264,346	733,846	[40.3; 41.3]	[73.5; 74.5]
FLA	Florida	1,070,376	2,712,798	[24.0; 31.0]	[79; 87.5]

Εικόνα 20. Η λίστα με τα πραγματικά δεδομένα που χρησιμοποιήθηκαν από την ιστοσελίδα του DIMACS9

Όπως είναι διακριτό (εικόνα 21.) οι κόμβοι αναπαριστούν τα πραγματικά σημεία ενώ οι ακμές τα μονοπάτια μεταξύ τους. Στις δυο τελευταίες στήλες φαίνεται το εύρος των συντεταγμένων που καλύπτει το κάθε αρχείο. Στην πρώτη στήλη έχουμε το Longtitude (γεωγραφικό μήκος) ενώ στην δεύτερη έχουμε το Latitude (γεωγραφικό πλάτος) των σημείων- κόμβων του αρχείου μας.

Στις παραπάνω στήλες υπάρχουν τα Files που χρειάζεται να μεταφορτωθούν και περιέχουν τις συντεταγμένες των σημείων (.co.gz) και τον χρόνο που απαιτείται για την (βάρη ακμών .gr.gz). Επόμενο βήμα μας είναι να κατεβάσουμε τα δυο αυτά αρχεία και να τα τοποθετήσουμε σε έναν φάκελο με το όνομα του αρχείου. Αναλυτικά στο παρακάτω παράδειγμα (εικόνα 22.):

NY	New York City	264,346	733,846	[40.3; 41.3]	[73.5; 74.5]	gr.gz file, 3.5 MB	gr.gz file, 3.6 MB	co.gz file, 2.0 MB

Εικόνα 21. Αναλυτικά τα χαρακτηριστικά του οδικού δικτύου της Νέας Υόρκης (NY)

Κατεβάζουμε τα δυο αρχεία NY.gr.gz και NY.co.gz και τα βάζουμε σε έναν φάκελο με όνομα NY. Στην τρίτη στήλη φαίνεται τα γεωγραφικό μήκος που καλύπτει το αρχείο μας, στην επόμενη το γεωγραφικό πλάτος και στις τελευταίες τα αρχεία που περιέχουν τα δεδομένα που πρέπει να μεταφορτωθούν. Το μέγεθος των δεδομένων είναι 264346 κορυφές και 733846 ακμές για το γράφημα της New York State (4η στήλη)

Επίσης χρησιμοποίησα τα αντίστοιχα δεδομένα FLA (FLA.gr.gz και FLA.co.gz) δηλαδή τα πραγματικά δεδομένα των σημείων για την πολιτεία της Florida. Το πλήθος τους είναι 1070376 κορυφές (σημεία) και 2712798 ακμές. Η επιλογή αυτού του πακέτου δεδομένων έγινε μιας και είναι πολύ μεγαλύτερου μεγέθους από το NY και σκοπός μας είναι η καταγραφή της συμπεριφοράς των ουρών προτεραιότητας στον Αλγόριθμο για γραφήματα διαφορετικού μεγέθους

5.3. Αποτελέσματα πειραματικών αξιολογήσεων

Σε αυτή την ενότητα θα παρουσιαστούν αναλυτικά τα αποτελέσματα των πειραματικών εκτελέσεων μας με την εφαρμογή διαφορετικών ουρών προτεραιότητας. Για κάθε ουρά έχουμε πάρει τους μέσους χρόνους 3 εκτελέσεων για διαφορετικές συνθήκες εκτέλεσης. Όσον αφορά την υλοποίηση του αλγορίθμου Dijkstra έχουμε πάρει 2 εκδοχές του.

- Την Single Source- Dijkstra SS που είναι η κανονική εκδοχή του αλγορίθμου Dijkstra με αναζήτηση των συντομότερων διαδρομών ξεκινώντας από μια αρχική κορυφή (s) και ψάχνουμε την απόστασή της από όλες τις υπόλοιπες του γραφήματος
- Τον A* (A-star) Αλγόριθμο (A* Dijkstra) όπου υπολογίζει όλες τις συντομότερες διαδρομές πιο αποτελεσματικά μιας και βασίζεται στην χρήση ευρετικών μεθόδων.

Τα queries (ερωτήματα) που εκτελούνται σε κάθε περίπτωση είναι 20000 ,αριθμός επαρκής για να βγουν συμπεράσματα αναφορικά με την λειτουργία των ουρών

Οι εκτελέσεις που έχουμε κάνει αφορούν τους μέσους χρόνους εκτέλεσης που έχουν καταγραφεί σε μονάδα μέτρησης τα ms (miliseconds) και το πλήθος των ταξινομημένων κόμβων (ταξινομήσεων) για καθένα ερώτημα που εκτελείται (query).

Για την καλύτερη αξιολόγηση των ουρών προτεραιότητας έχουμε χωρίσει τα queries σε 3 είδη ανάλογα με την συνολική τελική απόσταση του καθενός:

1. Short: σε αυτή την κατηγορία ανήκουν τα queries μικρών αποστάσεων. Στο δίκτυο New York State το μέγεθος των αποστάσεων φτάνει μέχρι τα 100 χιλιόμετρα, ενώ στο Florida State τα 500, δεδομένου ότι είναι αρκετά μεγαλύτερο γράφημα και αντίστοιχα τα queries θα είναι μεγαλύτερα
2. Medium: είναι για τις μεσαίες αποστάσεις, δηλαδή στο New York μεταξύ 100 και 250 χιλιομέτρων και στο Florida μεταξύ 1000 και 5000
3. Long: Τέλος έχουμε τις μεγάλες αποστάσεις και σε αυτές ανήκουν τα queries με απόσταση μεταξύ 1000 και 2000 χιλιομέτρων για το New York state ενώ για το Florida State οι αντίστοιχες αποστάσεις είναι μεταξύ 8000 και 15000 χιλιομέτρων.

5.3.1. Αποτελέσματα εκτελέσεων

Στο πρώτο αυτό μέρος θα αναλύσουμε τα αποτελέσματα των πραγματικών δεδομένων μας που έχουν ληφθεί από τον ισότοπο του DIMACS9 και αφορούν πραγματικές συντεταγμένες και αποστάσεις μεταξύ των σημείων. Σε κάθε πίνακα θα έχουμε το πλήθος των ταξινομημένων κόμβων ανά query (settled_nodes) και τον μέσο χρόνο εκτέλεσης (time) ανάλογα με τον τύπο της απόστασης για κάθε ουρά προτεραιότητας.

5.3.1.1. Οδικό δίκτυο Νέας Υόρκης

Σε αυτή την ενότητα θα παραθέσουμε τους πίνακες για τα αποτελέσματα που καταγράψαμε από την εκτέλεσης των διαφόρων ουρών προτεραιότητας για το δίκτυο γράφημα (NY) για τις 2 εκδοχές του αλγορίθμου Dijkstra (Κανονικός - Plain, A*)

Διαχωρισμός των αποστάσεων:

Distance Type	Km(minimum)	Km(maximum)
Short	20	100
Medium	250	600
Long	1.000	2.000

Στις εικόνες 23, 24 παραθέτω τα αποτελέσματα των πειραματικών αξιολογήσεων για τις 2 εκδοχές του Αλγορίθμου

- **PLAIN DIJKSTRA:**

DISTANCE	SHORT		MEDIUM		LONG	
20000 QUERIES	**time**	**settled_nodes**	**time**	**settled_nodes**	**time**	**settled_nodes**
STANDARD BINARY HEAP	0,58	2.096	28,32	92.112	76,67	253.608
BINARY HEAP NO-DEC	0,37	2.004	19,89	91.790	62,23	250.565
SEQUENCE HEAP	0,36	2.063	17,5	91.442	47,06	247.830
PAIRING HEAP	0,63	2.075	29,62	92.876	80,07	255.094

Εικόνα 22. Αναλυτικός πίνακας με τους μέσους χρόνους εκτέλεσης και το μέσο πλήθος ταξινομήσεων του Αλγορίθμου για την εκτέλεση του κανονικού Dijkstra για το οδικό δίκτυο της πολιτείας της Νέας Υόρκης (NY)

- **A* DIJKSTRA:**

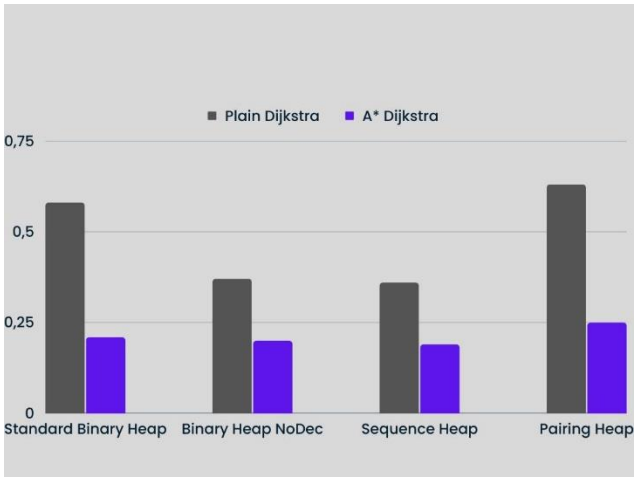
DISTANCE	SHORT		MEDIUM		LONG	
20000 QUERIES	**time**	**settled_nodes**	**time**	**settled_nodes**	**time**	**settled_nodes**
STANDARD BINARY HEAP	0,21	723	8,74	28.658	51,56	153.305
BINARY HEAP NO-DEC	0,2	668	6,19	28.069	35,02	150.815
SEQUENCE HEAP	0,19	650	5,81	28.767	32,16	147.931
PAIRING HEAP	0,25	778	8,77	28.931	52,02	153.854

Εικόνα 23. Αναλυτικός πίνακας με τους μέσους χρόνους εκτέλεσης και το μέσο πλήθος ταξινομήσεων του Αλγορίθμου για την εκτέλεση του A* Dijkstra για το οδικό δίκτυο της πολιτείας της Νέας Υόρκης (NY)

Διαγράμματα χρόνων

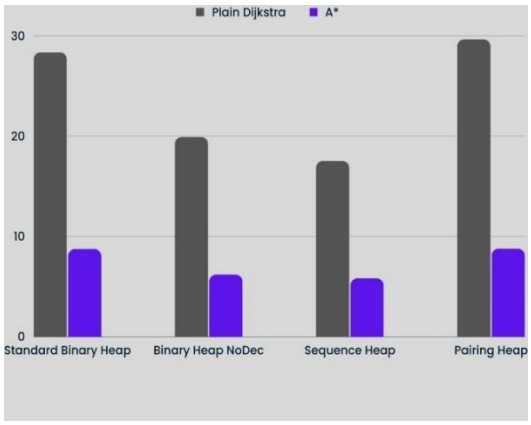
Οι καταγραφές των χρονών παρουσιάζονται σε ms (miliseconds)/query. Σε κάθε διάγραμμα έχουμε από δυο στήλες για κάθε ουρά προτεραιότητας που έχει μελετηθεί, παρουσιάζοντας και συγκρίνοντας τα αποτελέσματα μεταξύ Plain Dijkstra και τον A* (εικόνες 26, 27, 28).

- Short distances:



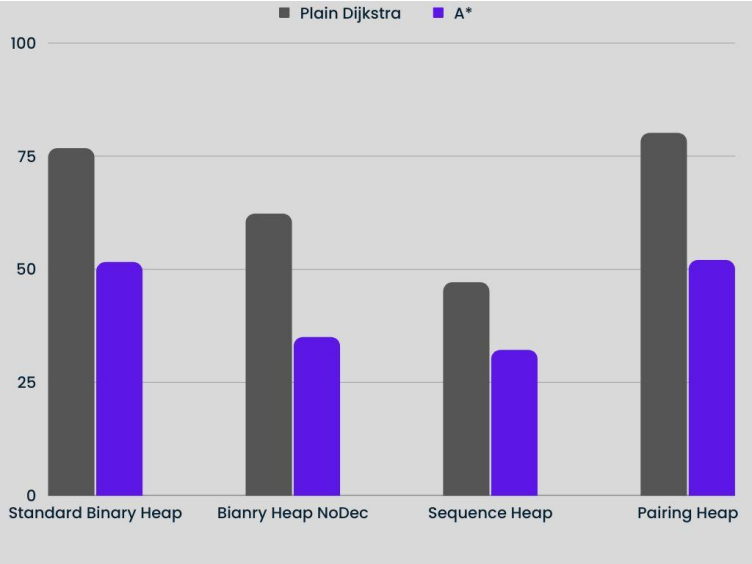
Εικόνα 26. Short distances (NY). Βλέπουμε ότι για τον Plain Dijkstra οι 2 No-dec (Sequence heap, Binary Heap no-dec) εκδοχές παρουσιάζουν πολύ καλύτερους χρόνους εκτέλεσης από τις δυο DijkstraDec (Standard Binary Heap, Pairing Heap) εκδοχές. Η διαφορά τους μειώνεται αισθητά για τον A* μιας και η αναλογία των χρόνων εκτέλεσης είναι αρκετά μικρότερη.

- Medium distances:



Εικόνα 27. Medium distances (NY)- Για μεσαίες (medium) αποστάσεις υπερτερούν και πάλι οι 2 DijkstraNoDec εκδοχές του αλγορίθμου με την διαφορά τους από τους άλλους δυο DijkstraDec να αυξάνεται. Η πιο αποδοτική ουρά είναι η Sequence Heap Με την Binary Heap NoDec να ακολουθεί. Για τον A* ισχύει η ίδια κατάταξη των ουρών απλά η διαφορά μεταξύ τους μειώνεται.

- Long distances:



Εικόνα 28. Long distances (NY): Για μεγάλες αποστάσεις φαίνεται η υλοποίηση με Sequence Heap να μεγαλώνει την διαφορά από τις υπόλοιπες ουρές με την Binary Heap NoDec να ακολουθεί. Η διαφορά αυτή μειώνεται λίγο στην A* αφού και οι χρόνοι εκτέλεσης μειώνονται λόγω των ευρετικών μεθόδων.

Συγκρίσεις των Αλγορίθμων

Στους αναλυτικούς πίνακες είδαμε τις ταξινομήσεις και τους χρόνους εκτέλεσης για διαφορετικές συνθήκες και περιπτώσεις του αλγορίθμου και των διαγραμμάτων μας. Σε αυτό το σημείο θα γίνει μια πλήρης επεξήγηση των πινάκων όσον αφορά το που οφείλονται οι διαφορές μεταξύ χρόνων και ταξινομήσεων.

- Plain Dijkstra-A* Dijkstra: Αρχικά θα συγκρίνουμε και θα εξηγήσουμε τις διάφορες μεταξύ των 2 εκδοχών των εκτελέσεων του αλγορίθμου. Έχουμε τον κανονικό Dijkstra (Plain) που είναι ο αλγόριθμος που είδαμε σε προηγούμενο κεφάλαιο (κεφάλαιο 2) και την βελτιωμένη έκδοση του τον A* Dijkstra (A-star) , που βασίζεται στην ευρέτική αναζήτηση των ακμών. Στην ευρετική αναζήτηση η κάθε κορυφή γνωρίζει εκτός από την απόσταση της με τις γειτονικές την πιθανή απόσταση από την τελική κορυφή (t- target) μέσω εκτιμήσεων που γίνονταιστο δίκτυο NY. Έτσι μειώνεται ο χρόνος αναζήτησης για κάθε ακμή μιας και διαθέτουμε μεγαλύτερη γνώση του προβλήματος.
-
    1. Short distances: Ο χρόνος εκτέλεσης κάθε ερωτήματος μειώνεται κατά ποσοστό της τάξεως του 45-50% ενώ για τις ταξινομήσεις το αντίστοιχο ποσοστό είναι στο 65%
    2. Medium distances: Ο χρόνος εκτέλεσης μειώνεται κατά 65 - 70% ενώ και το πλήθος των ταξινομήσεων μειώνεται σε παρόμοιο ποσοστό.
    3. Long distances: Ο χρόνος εκτέλεσης μειώνεται κατά 30 – 35% ενώ το μέσο πλήθος ταξινομήσεων μειώνεται αντίστοιχα κατά 40%

Ενδεικτικά ποσοστά μείωσης του μέσου χρόνου εκτέλεσης και του μέσου πλήθους ταξινομήσεων ανά ερώτημα (query)

DISTANCE	SHORT		MEDIUM		LONG	
	time	settled_nodes	time	settled_nodes	time	settled_nodes
STANDARD BINARY HEAP	64%	65%	69%	69%	33%	40%
BINARY HEAP NO-DEC	45%	67%	69%	70%	44%	40%
SEQUENCE HEAP	47%	68%	67%	68%	32%	41%
PAIRING HEAP	68%	63%	70%	32%	35%	40%

Σύνοψη: Ο A* φτάνει στο μέγιστο της απόδοσης του για τις μικρές και τις μεσαίες αποστάσεις του δικτύου μας μιας και ο χρόνος εκτέλεσης όπως και οι ταξινομήσεις μειώνονται στο 1/3 τους. Αν όμως η απόσταση των queries είναι αρκετά μεγάλη τότε η απόδοση του αλγορίθμου μειώνεται αισθητά και ο χρόνος εκτέλεσης και το πλήθος ταξινομήσεων μειώνονται στο 2/3.

- **Dijkstra-Dec και Dijkstra-NoDec :** Σε προηγούμενη ενότητα είδαμε ότι ο Αλγόριθμος του Dijkstra με χρήση ουρών προτεραιότητας χωρίζεται σε δυο κατηγορίες: τον Dijkstra-Dec (που υλοποιείται με ουρές προτεραιότητας που εκτελούν την λειτουργία μείωσης κλειδιού - decrease-key) και τον Dijkstra-NoDec (υλοποίηση ουρών που δεν υποστηρίζουν την λειτουργία decrease-key, πάρα μόνο τις λειτουργίες εισαγωγής (insert) και διαγραφής (delete) στοιχείων). Όπως είχε αναφερθεί και στην θεωρία μας η 2η εκδοχή είναι πιο γρήγορη από την 1η λόγω της απουσίας της λειτουργίας της decrease-key. Ο ισχυρισμός αυτός αποδεικνύεται και από τις αξιολογήσεις μας μιας και οι δύο ουρές της κατηγορίας Dijkstra No-dec εκτελούσαν τον αλγόριθμο πολύ καλύτερα. Παρακάτω αναγράφονται και τα σχετικά ποσοστά των διαφορών τους.

  - **Plain Dijkstra:**

    - Short distances: Οι υλοποιήσεις Dijkstra-NoDec παρουσιάζουν καλύτερους χρόνους κατά ποσοστό της τάξεως 40% με ενδεικτικούς χρόνους
      - DijkstraDec: 0,6 ms/query
      - DijkstraNoDec: 0,35 ms/query

Αναφορικά με το μέσο πλήθος των ταξινομήσεων που γίνονται οι διαφορές είναι πολύ μικρές μεγέθους των 50-60 ανά query, κάτι που αντιστοιχεί σε ποσοστό κοντά στο 10%

    - Medium distances: Και σε αυτού του είδους αποστάσεις ο Dijkstra-NoDec παρουσιάζει καλύτερους χρόνους εκτέλεσης κατά 30-35% έχοντας ενδεικτικούς χρόνους:
      - DijkstraDec: 28,32 ms/query
      - DijkstraNoDec: 17,5 ms/query

Για το μέσο πλήθος ταξινομήσεων που εκτελούνται ανά query η διαφορά είναι κατά μέσο όρο 500 queries κάτι που αντιστοιχεί σε πολύ μικρό ποσοστό της τάξης του 5%.

- Long distances:
    - DijkstraDec: 76,67 ms/query
    - DijkstraNoDec: 47,06 ms/query

Η διαφορά μεταξύ των δυο χρόνων εκτελέσεων των αλγορίθμων είναι της τάξεως του 40% και σε αυτή την περίπτωση, ενώ για το πλήθος των settled nodes/query η διαφορά τους είναι περίπου 3.000, κάτι που αντιστοιχεί σε ποσοστό περίπου 2-3%.

- **A*:**
    - Short distances:
        - DijkstraDec: 0,21 ms/query
        - DijkstraNoDec: 0,19 ms/query

Η διαφορά σε αυτόν τον τύπο αποστάσεων είναι πολύ μικρή μιας και οι αντίστοιχοι χρόνοι είναι μικροί, της τάξεως του 0,2 ms/query (κατά προσέγγιση για κάθε ουρά) και έτσι η διαφορά θα είναι εξίσου μικρή, δηλαδή κοντά στο 10%. Για το αντίστοιχο μέσο πλήθος settled_nodes/query θα είναι και εκεί σε ποσοστό που αγγίζει το 10%

- Medium distances:
    - DijkstraDec: 5,81 ms/query
    - DijkstraNoDec:8,74 ms/query

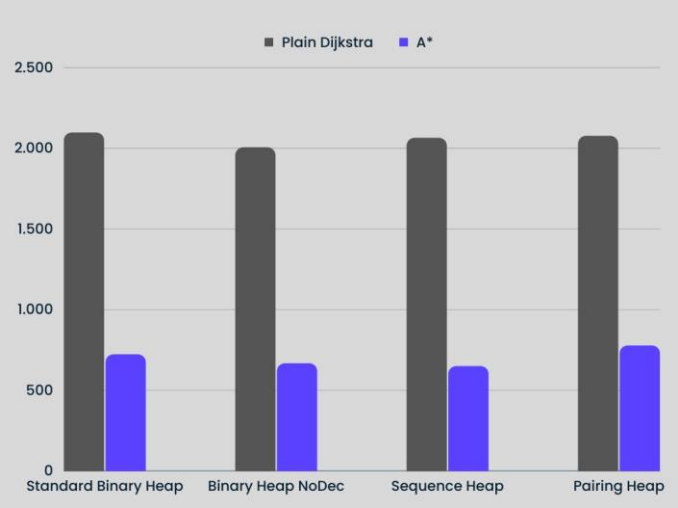
Η διαφορά των χρόνων είναι σε ποσοστό 33% (καλύτερος ο NoDec). Το ποσοστό αυτό είναι μεγαλύτερο από ότι στην κατηγορία των short distances, αλλά παρόμοιο με το ποσοστό για τις medium distances για τον Plain Dijkstra. Όσον αφορά τα settled_nodes/query το αντίστοιχο ποσοστό είναι αρκετά μικρότερο, της τάξεως του 2.5 - 3%.

- Long Distances:
    - DijkstraDec: 32,16 ms/query
    - DijkstraNoDec: 51,56 ms/query

Το ποσοστό διαφοράς ανεβαίνει και σε αυτή την περίπτωση μιας και φτάνει το 37% υπέρ του DijkstraNoDec. Για το πλήθος settled_nodes/query είναι μικρή η διαφορά δηλαδή 5.500 settled_nodes, δηλαδή ποσοστό κοντά στο 3%

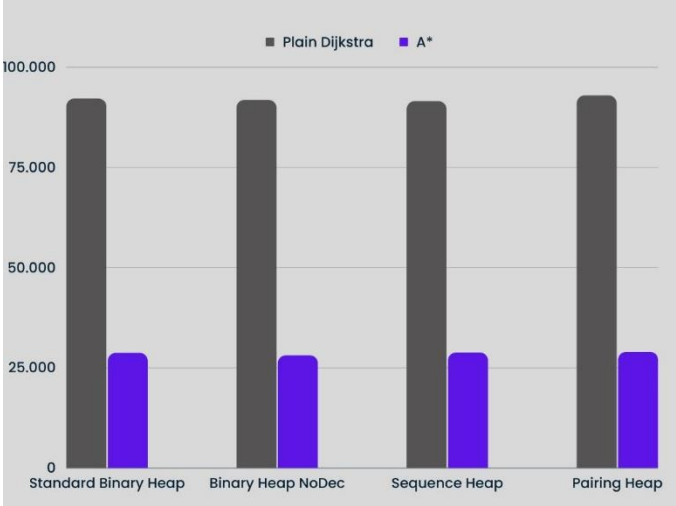
Καταγράφηκε σε διαγράμματα και το πλήθος των ταξινομήσεων που εκτελούνται για κάθε query (εικόνες 34, 35, 36).

- Short distances:



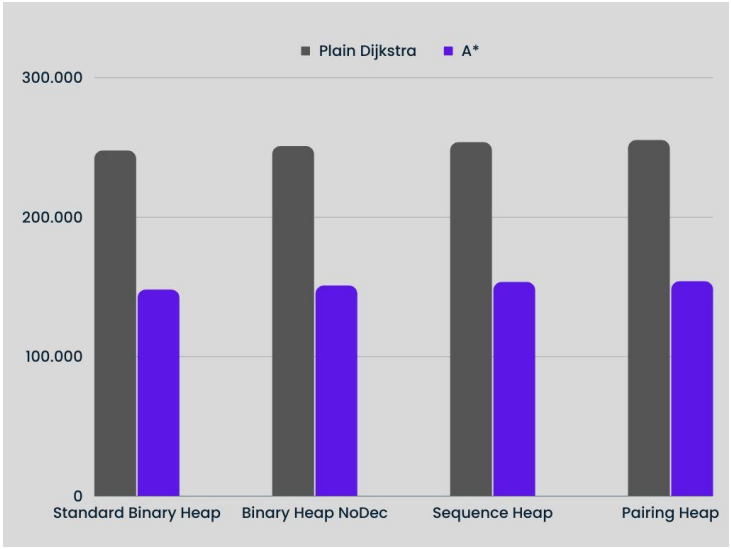
Εικόνα 29. Settled_nodes για Short distances (NY): Στην υλοποίηση του κανονικού αλγορίθμου Dijkstra η Sequence Heap είναι η ουρά που εκτελεί τις λιγότερες ταξινομήσεις ανά query σε σχέση με τις υπόλοιπες που απέχουν με αρκετά μικρή διαφορά πάντως. Η διαφορά αυτή γίνεται ακόμη μικρότερη στον A* με την Sequence να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική όμως το χάσμα μεταξύ τους είναι ελάχιστο δηλαδή της τάξης των 100 ταξινομήσεων το πολύ

- Medium distances:



Εικόνα 30. Settled_nodes για Medium distances (NY): Στην υλοποίηση του κανονικού αλγορίθμου Dijkstra η Sequence Heap είναι η ουρά που και εδώ εκτελεί τις λιγότερες ταξινομήσεις ανά query σε σχέση με τις υπόλοιπες. Η διαφορά αυτή είναι πιο μεγάλη από ότι ήταν στις μικρές αποστάσεις δηλαδή είναι της τάξης των 600 ταξινομήσεων ανά ερώτημα. Η διαφορά αυτή γίνεται ακόμη μικρότερη στον A* με την Sequence να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική όμως το χάσμα μεταξύ τους είναι ελάχιστο δηλαδή της τάξης των 300 ταξινομήσεων το πολύ. Αυτό συμβαίνει μιας και ο A* είναι πιο γρήγορος και οι αποστάσεις μεταξύ των τιμών των ταξινομήσεων μειώνονται.

- Long distances:



Εικόνα 31. Settled_nodes για Long Distances (NY). Στην υλοποίηση του κανονικού αλγορίθμου Dijkstra η Sequence Heap είναι η ουρά που εκτελεί τις λιγότερες ταξινομήσεις ανά query σε σχέση με τις υπόλοιπες που απέχουν με αρκετά μικρή διαφορά πάντως, περίπου 3000 – 6000 ταξινομήσεων, διαφορά αρκετά μεγαλύτερη από ότι στις μικρές και στις μεσαίες αποστάσεις. Για τον A* όπως είναι λογικό η διαφορά αυτή γίνεται μικρότερη με την Sequence να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική όμως το χάσμα μεταξύ τους είναι ελάχιστο δηλαδή της τάξης των 500-600 ταξινομήσεων το πολύ.

- **Settled_nodes :** Ένας άλλος τρόπος να εξηγήσουμε τα αποτελέσματα είναι να συγκρίνουμε το πλήθος των ταξινομήσεων που γίνονται για κάθε ερώτημα (query) που εκτελείται. Άρα όσο λιγότερες είναι οι ταξινομήσεις που γίνονται τόσο λιγότερος θα είναι και ο χρόνος εκτέλεσης του Αλγορίθμου. Στους επόμενους πίνακες θα δούμε τις εν λόγω διάφορες και την κατάταξη όσον αφορά την συγκεκριμένη παράμετρο

	DIJKSTRA			A*		
Distance type	**short**	**medium**	**long**	**short**	**medium**	**long**
Sequence Heap	2.004	91.442	247.565	650	28.069	147.931
Binary Heap NoDec	2.063	91.790	250.830	668	28.658	150.815
Standard Binary Heap	2.075	92.112	253.608	723	28.767	153.305
Pairing Heap	2.096	92.876	255.094	778	28.931	153.854

  - **PLAIN DIJKSTRA:** Όσον αφορά τον κανονικό Αλγόριθμο του Dijkstra η διαφορά στο μέσο πλήθος ταξινομήσεων είναι πολύ μικρή για short distances μιας και η καλύτερη χρονικά ουρά διαφέρει από την χειρότερη κατά 50 ταξινομήσεις. Για medium distances αυτή η διαφορά κυμαίνεται από 350 μέχρι 1.400 μεταξύ των ουρών. Αντιθέτως όσο μεγαλώνουν οι αποστάσεις (long distances) η διαφορά αυξάνεται αισθητά και είναι της τάξεως μεταξύ 3.000 μέχρι και 8.000 για την διαφορά “καλύτερης” και “χειρότερης” ουράς προτεραιότητας.

  - **A*:** Για τον A* αλγόριθμο όπως είναι προφανές θα μειώνεται η διαφορά αυτή μιας και μειώνονται αντίστοιχα και οι μέσοι χρόνοι εκτέλεσης των queries. Μόνο για short distances η μέγιστη διαφορά θα αυξηθεί επειδή στην σύγκριση μεταξύ Sequence και Pairing Heap θα φτάνει τις 120 ταξινομήσεις. Για medium distances θα γίνει μικρότερη της τάξης το πολύ 900 ταξινομήσεων ενώ για long distances θα είναι σχεδόν παρόμοια με την προηγουμένη υλοποίηση, δηλαδή από 3.000 μέχρι 6.000.

  -

5.3.1.2. Οδικό δίκτυο της πολιτείας της Florida

Distance type	Km(minimum)	Km(maximum)
Short	20	500
Medium	1.000	5.000
Long	8.000	15.000

Ακολουθούν οι πίνακες με τα αποτελέσματα των εκτελέσεων για το οδικό δίκτυο της Florida (FLA)- εικόνες 32, 33 όπως και τα ενδεικτικά διαγράμματά τους στις εικόνες 34, 35, 36.

- **PLAIN DIJKSTRA:**

	SHORT		MEDIUM		LONG	
20000 QUERIES	**time**	**settled_nodes**	**time**	**settled_nodes**	**time**	**settled_nodes**
STANDARD BINARY HEAP	8,29	25.677	178,35	495.031	358,01	1.040.074
BINARY HEAP NO-DEC	6,24	25.009	147,69	457.159	269,33	845.673
SEQUENCE HEAP	5,73	24.973	109,09	448.773	229,87	744.265
PAIRING HEAP	9,16	25.540	197,71	496.149	404,62	1.042.043

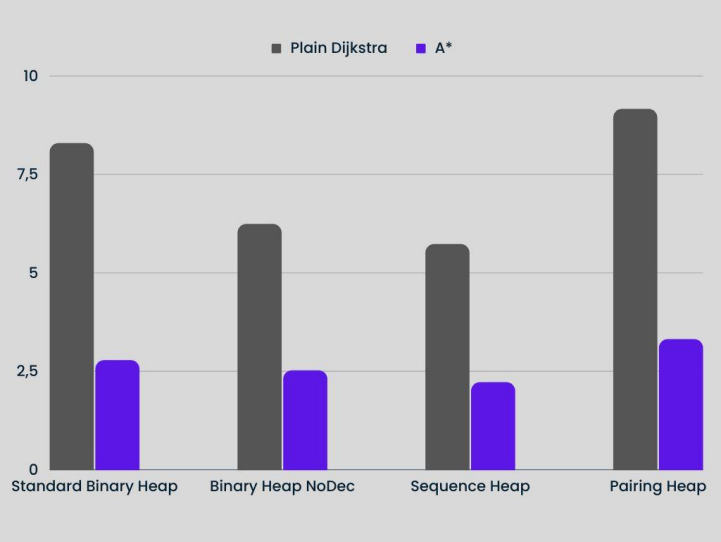
Εικόνα 32. Αναλυτικός πίνακας με τους μέσους χρόνους εκτέλεσης και το μέσο πλήθος ταξινομήσεων του Αλγορίθμου για την εκτέλεση του κανονικού Dijkstra για το οδικό δίκτυο της πολιτείας της Florida (FLA)

- **A* DIJKSTRA:**

DISTANCE	SHORT		MEDIUM		LONG	
20.000 QUERIES	**time**	**settled_nodes**	**time**	**settled_nodes**	**time**	**settled_nodes**
STANDARD BINARY HEAP	2,79	9.114	82,29	199.559	340,73	848.222
BINARY HEAP NO-DEC	2,53	9.040	58,15	190.056	266,91	807.829
SEQUENCE HEAP	2,23	9.002	44,89	189.890	209,27	689.464
PAIRING HEAP	3,32	9.140	76,08	223.662	370,78	868.295

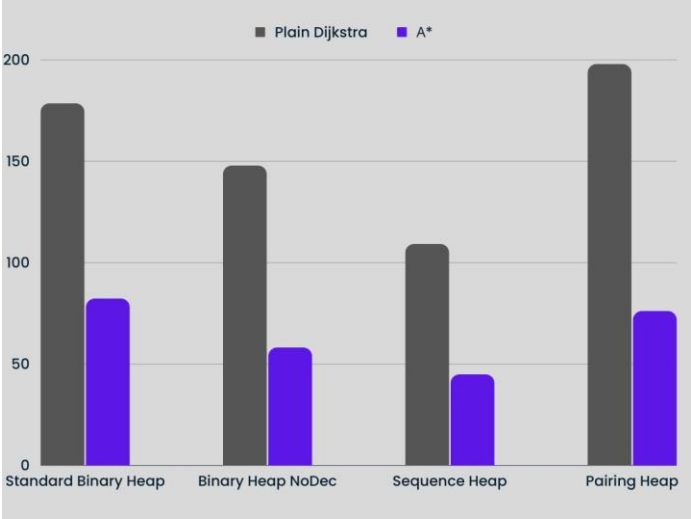
Εικόνα 33. Αναλυτικός πίνακας με τους μέσους χρόνους εκτέλεσης και το μέσο πλήθος ταξινομήσεων του Αλγορίθμου για την εκτέλεση του A* Dijkstra για το οδικό δίκτυο της πολιτείας της Florida (FLA)

- Short distances:



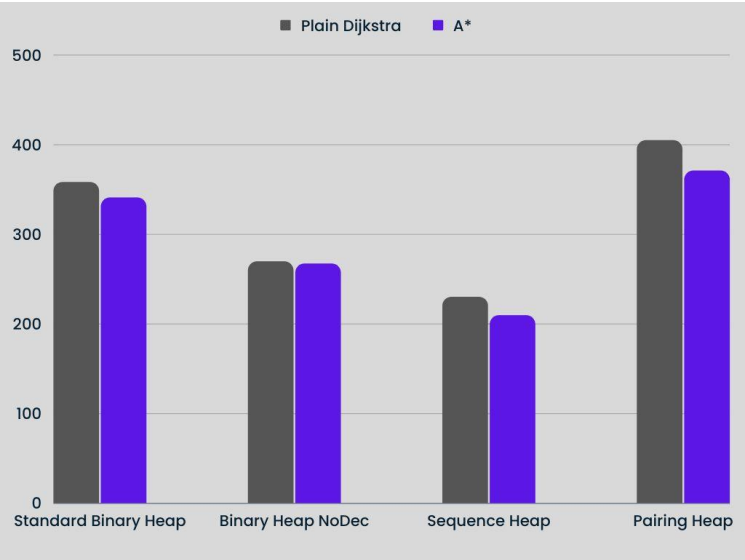
Εικόνα 34. Short distances (FLA). Όπως είναι προφανές οι χρόνοι εκτέλεσης αυξάνονται μιας και εργαζόμαστε σε πολύ μεγαλύτερο γράφημα. Στην υλοποίηση του κανονικού αλγορίθμου Dijkstra η πιο αποδοτική ουρά είναι η Sequence Heap με τις υπόλοιπες να ακολουθούν με μια σχετική διαφορά οι υπόλοιπες. Η διαφορά αυτή είναι μικρότερη στον A* με την Sequence να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική.

- Medium distances:



Εικόνα 35. Medium distances (FLA). Στην υλοποίηση του κανονικού αλγορίθμου Dijkstra για μεσαίες αποστάσεις η πιο αποδοτική ουρά είναι η Sequence Heap με τις υπόλοιπες να ακολουθούν με μια σχετική διαφορά οι υπόλοιπες. Η διαφορά είναι μεγαλύτερη από ότι ήταν στις προηγούμενες αποστάσεις. Για τον A* πάλι βλέπουμε την Sequence Heap να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική. Η διαφορές μεταξύ των ουρών είναι μεγαλύτερες όσο μεγαλώνουν και οι αποστάσεις στις οποίες εργαζόμαστε

- Long distances:



Εικόνα 36. Long distances (FLA): Στις μεγάλες αποστάσεις βλέπουμε ότι η κατάταξη μεταξύ των ουρών δεν αλλάζει με την Sequence Heap να υπερτερεί και τις υπόλοιπες να ακολουθούν. Η μόνη διαφορά με τις προηγούμενες περιπτώσεις (μέγεθος γραφήματος, αποστάσεις) είναι ότι η διαφορά των χρόνων μεταξύ Plain - A* μειώνεται κατά πολύ. Σε προηγούμενες περιπτώσεις η διαφορά μεταξύ των 2 εκδοχών κυμαίνεται στο 30-40% όμως εδώ το ποσοστό πέφτει κατακόρυφα στο εύρος του 1-

- Plain Dijkstra -A*: όπως είδαμε και στο προηγούμενο δίκτυο, ετσι και εδώ θα παρουσιαστούν οι διαφορές μεταξύ των δυο εκδοχών του Αλγορίθμου.

1. Short distances: Όπως και στο προηγούμενο γράφημα έτσι και σε αυτό για τις μικρές αποστάσεις ο μέσος χρόνος εκτέλεσης μειώνεται κατά ποσοστό 60 – 65% ενώ το πλήθος των ταξινομήσεων μειώνεται και αυτό κατά ποσοστό της τάξεως του 65%.
2. Medium distances: Ο μέσος χρόνος εκτέλεσης αλλά και το πλήθος των ταξινομήσεων μειώνονται κατά το ίδιο ποσοστό, που είναι της τάξεως του 55 – 60%
3. Long distances: Ο μέσος χρόνος εκτέλεσης μειώνεται μόλις κατά 10-15% ενώ οι ταξινομήσεις κατά 15-20%, ποσοστό αισθητά μικρότερο σε σχέση με το αντίστοιχο για τους 2 προηγούμενους τύπους αποστάσεων.

DISTANCE	SHORT		MEDIUM		LONG	
	time	settled_nodes	time	settled_nodes	time	settled_nodes
STANDARD BINARY HEAP	65%	65%	54%	55%	5%	19%
BINARY HEAP NO-DEC	60%	64%	60%	58%	1%	5%
SEQUENCE HEAP	61%	65%	60%	58%	8%	8%
PAIRING HEAP	64%	64%	62%	60%	8%	17%

Σύνοψη: Όπως είδαμε και στο προηγούμενο γράφημα ο A* πιάνει το μέγιστο της απόδοσης του για μικρές και μεσαίες αποστάσεις μειώνοντας τους χρόνους εκτελέσεις και τις ταξινομήσεις στο 1/3. Όταν η απόσταση είναι αρκετά μεγάλη (για queries μήκους 10000 χιλιόμετρα και πάνω) η απόδοση του A* πέφτει κατακόρυφα μιας και οι αντίστοιχες μειώσεις πέφτουν στο 10-20% μόλις.

- **Dijkstra-Dec και Dijkstra-NoDec :** Όπως είδαμε και σε προηγούμενη ενότητα ο Αλγόριθμος του Dijkstra με χρήση ουρών προτεραιότητας χωρίζεται σε δυο κατηγορίες: τον Dijkstra-Dec (που υλοποιείται με ουρές προτεραιότητας που εκτελούν την λειτουργία μείωσης κλειδιού - decrease-key) και τον Dijkstra-NoDec (υλοποίηση ουρών που δεν υποστηρίζουν την λειτουργία decrease-key, πάρα μόνο τις λειτουργίες εισαγωγής (insert) και διαγραφής (delete) στοιχείων). Όπως είναι εύκολα κατανοητό από την θεωρία και τα αποτελέσματα των εκτελέσεων μας η 2η εκδοχή του αλγορίθμου εκτελείται σε καλύτερο χρόνο, μιας και απαιτείται χρόνος μόνο για τις 2 λειτουργίες που αναφέραμε προηγουμένως

  - **Plain Dijkstra:**

    - Short distances: Οι υλοποιήσεις Dijkstra-NoDec παρουσιάζουν καλύτερους χρόνους κατά ποσοστό της τάξεως 30% με ενδεικτικούς χρόνους
      - DijkstraDec: 5,73 ms/query
      - DijkstraNoDec: 8,29 ms/query

    - Medium distances: Και σε αυτού του είδους αποστάσεις ο Dijkstra-NoDec παρουσιάζει καλύτερους χρόνους εκτέλεσης κατά 40% περίπου έχοντας ως ενδεικτικούς χρόνους:
      - DijkstraDec: 178,35 ms/query
      - DijkstraNoDec: 109,09 ms/query

    - Long distances:
      - DijkstraDec: 229,87 ms/query
      - DijkstraNoDec: 358,01 ms/query

Η διαφορά μεταξύ των δυο χρόνων εκτελέσεων των αλγορίθμων είναι της τάξεως του 35-40% και σε αυτή την περίπτωση.

- **A*:**

  - Short distances:
    - DijkstraDec: 2,79 ms/query
    - DijkstraNoDec: 2,23 ms/query

Η διαφορά σε αυτόν τον τύπο αποστάσεων είναι πολύ μικρή μιας και οι αντίστοιχοι χρόνοι είναι μικροί, της τάξεως του 0,56 ms/query (κατά προσέγγιση για κάθε ουρά) και έτσι η διαφορά θα είναι εξίσου μικρή, δηλαδή κοντά στο 20%.

  - Medium distances:
    - DijkstraDec: 76,08 ms/query
    - DijkstraNoDec: 44,89 ms/query

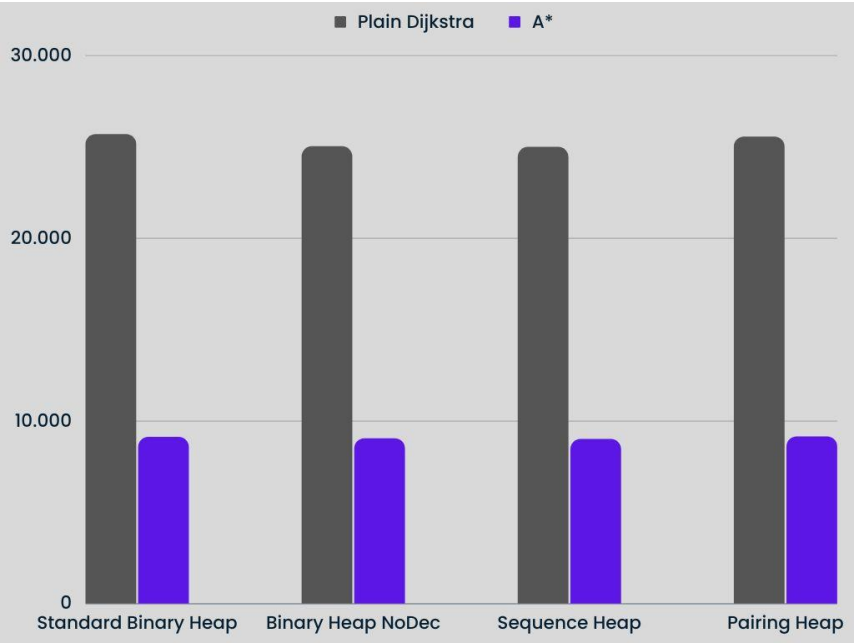
Η διαφορά των χρόνων είναι σε ποσοστό 40% (καλύτερος ο NoDec). Το ποσοστό αυτό είναι μεγαλύτερο (περίπου το διπλάσιο) από ότι στην κατηγορία των short distances, αλλά παρόμοιο με το ποσοστό για τις medium distances για τον Plain Dijkstra.

  - Long Distances:
    - DijkstraDec: 340,73 ms/query
    - DijkstraNoDec: 209,27 ms/query

Το ποσοστό διαφοράς ανεβαίνει και σε αυτή την περίπτωση μιας και φτάνει το 38-40% υπέρ του DijkstraNoDec.

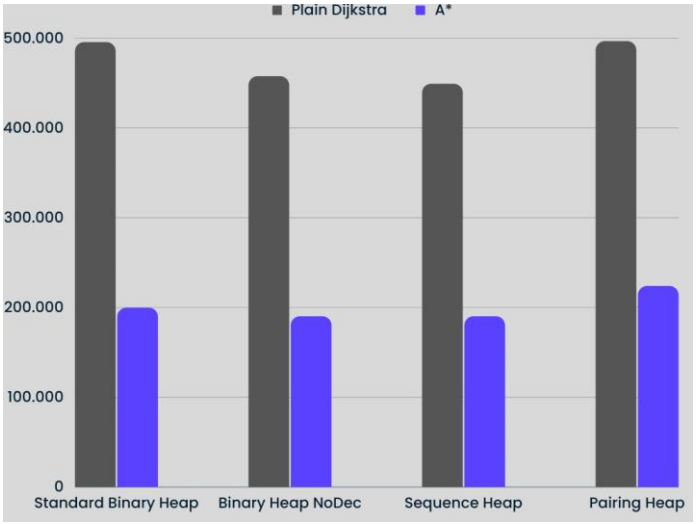
Επιπλέον θα δούμε και τα ενδεικτικά διαγράμματα για το πλήθος των settled_nodes για κάθε query (37, 38, 39).

- Short distances:



Εικόνα 24. Settled_nodes για Short distances (FLA): Για τον κανονικό αλγόριθμο Dijkstra και εδώ η Sequence Heap είναι η ουρά που εκτελεί τις λιγότερες ταξινομήσεις ανά query σε σχέση με τις υπόλοιπες που απέχουν με αρκετά μικρή διαφορά πάντως. Η διαφορά αυτή γίνεται ακόμη μικρότερη στον A*όπως είναι λογικό με την Sequence να υπερτερεί και την Pairing Heap να είναι η λιγότερο αποδοτική όμως το χάσμα μεταξύ τους είναι ελάχιστο δηλαδή της τάξης των 1000 ταξινομήσεων το πολύ

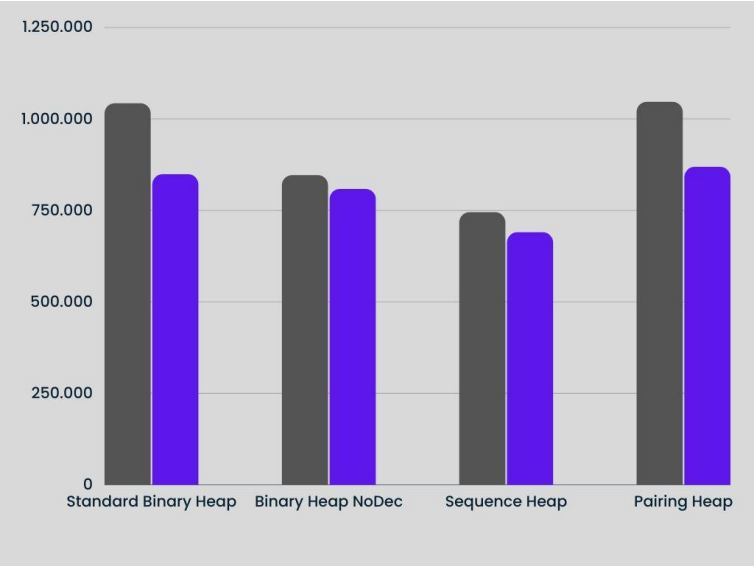
- Medium distances:



Εικόνα 38. Settled_nodes για Medium distances (FLA): Για τον κανονικό αλγόριθμο Dijkstrakαι εδώ η Sequence Heap είναι η ουρά που εκτελεί τις λιγότερες ταξινομήσεις ανά query σε σχέση με τις υπόλοιπες που απέχουν με σχετικά μεγάλη διαφορά μιας και οι αποστάσεις των queries αυξάνονται κατά πολύ. Η διαφορά αυτή ισούται με 15.000 με 50.000 ταξινομήσεις, νούμερο αρκετά μεγαλύτερο από τα νούμερα που είδαμε

στις προηγούμενες αξιολογήσεις. Βέβαια το νούμερο αυτό θα μειωθεί στον A* από την στιγμή που η συγκεκριμένη εκδοχή είναι ταχύτερη από την κανονική. Πλέον η διαφορά αυτή θα κυμαίνεται μεταξύ 10.000 και 35.000 ταξινομήσεων ανά εκτελούμενο ερώτημα.

- Long distances:



Εικόνα 39. Settled_nodes για Long distances (FLA): Σε αυτή την περίπτωση έχουμε τις μεγαλύτερες διαφορές που εμφανίζονται κατά τις αξιολογήσεις μας. Αρχικά βλέπουμε ότι και σε αυτό το διάγραμμα η Sequence Heap είναι αυτή που εκτελεί τις λιγότερες ταξινομήσεις με τις υπόλοιπες ουρές να ακολουθούν. Οι διαφορές στις τιμές τους πλέον φτάνουν μέχρι και τις 100.000 - 300.000 ταξινομήσεις για τον κανονικό αλγόριθμο του Dijkstra. Στον A* το νούμερο αυτό μειώνεται αλλά και πάλι είναι σε υψηλά επίπεδα, δηλαδή 80.000 – 200.000 ταξινομήσεις. Όπως είναι εύκολα κατανοητό οι διαφορές των τιμών είναι πολύ μεγαλύτερες από κάθε άλλον τύπο γραφήματος και απόστασης.

- **Settled_nodes:** Ακολουθεί επεξήγηση αναφορικά με το μέσο πλήθος των ταξινομήσεων

	DIJKSTRA			A*		
Distance type	**short**	**medium**	**long**	**short**	**medium**	**long**
Sequence Heap	24.973	448.730	744.265	9.002	189.890	689.464
Binary Heap NoDec	25.006	457.159	845.673	9.040	190.056	807.829
Standard Binary Heap	25.540	495.031	1.042.043	9.114	199.559	848.222
Pairing Heap	25.677	496.149	1.046.074	9.140	223.662	868.295

  - **PLAIN DIJKSTRA:** Όσον αφορά τον κανονικό Αλγόριθμο του Dijkstra η διαφορά στο μέσο πλήθος ταξινομήσεων είναι πολύ μικρή για short distances μιας και η καλύτερη χρονικά ουρά διαφέρει από την χειρότερη κατά 700 ταξινομήσεις. Για medium distances αυτή η διαφορά κυμαίνεται από 9.000 μέχρι 48.000 μεταξύ των ουρών. Το νούμερο αυτό βλέπουμε ότι είναι αρκετά μεγαλύτερο από το αντίστοιχο για το δίκτυο NY. Όσο μεγαλώνουν οι αποστάσεις (long distances) η διαφορά αυξάνεται κι άλλο και φτάνει να κυμαίνεται μεταξύ 100.000 μέχρι και 300.000 για την διαφορά “καλύτερης” και “χειρότερης” ουράς προτεραιότητας. Για αυτές τις αποστάσεις έχουμε και την μεγαλύτερη απόκλιση για το μέσο πλήθος των ταξινομήσεων.

  - **A*:** Για τον A* αλγόριθμο όπως είναι προφανές και σε αυτήν την περίπτωση θα μειώνεται η διαφορά μιας και μειώνονται αντίστοιχα και οι μέσοι χρόνοι εκτέλεσης των queries. Για short distances η διαφορά θα κυμαίνεται μεταξύ 40 και 140 όπου είναι για την διαφορά των Sequence και Pairing Heap. Για medium distances θα γίνει μικρότερη, δηλαδή περίπου 1000 ταξινομήσεις αλλά για την διαφορά της καλύτερης με την χειρότερη ουρά θα φτάνει και τις 34.000. Τέλος για long distances θα είναι σχεδόν παρεμφερής με την προηγουμένη υλοποίηση, δηλαδή από 110.000 μέχρι και 180.000.

5.4. Επεξήγηση αποτελεσμάτων

Όπως παρατηρήσαμε από τις προηγούμενες συγκρίσεις, αυτά τα τέσσερα κριτήρια καθορίζουν ποια ουρά προτεραιότητας είναι πιο αποδοτική. Τα κυριότερα εξ αυτών είναι προφανώς η δομή της ουράς αλλά και η ύπαρξη της λειτουργίας decrease-key.

Για το πρώτο κριτήριο είδαμε ότι αν η ουρά είναι πολυδιάστατη δομή, τότε θα είναι σιγουρά αποδοτικότερη από μια μονοδιάστατη. Έτσι συμπεραίνουμε ότι η Sequence Heap είναι η πιο αποδοτική από όλες τις ουρές στην επίλυση του προβλήματος συντομότερων διαδρομών.

Αυτό αρχικά οφείλεται στο γεγονός ότι ακολουθεί την συγχώνευση k-τρόπων για τα στοιχεία (k-way merge) με αποτέλεσμα την καλύτερη χρήση της κρυφής μνήμης. Όπως είδαμε στις προηγούμενες ουρές προτεραιότητας, η σύγκριση των κλειδιών των στοιχείων γινόντουσαν μέσω συγκρίσεων μεταξύ των στοιχείων κάθε επιπέδου του δέντρου της ουράς. Στην περίπτωση όμως της Sequence heap έχουμε ομαδοποίηση των κλειδιών των στοιχείων σύμφωνα με το k. Αυτό σημαίνει ότι αντί για ένα στοιχείο – και αντίστοιχα ένα κλειδί- σε κάθε κόμβο του δέντρου υπάρχουν k στοιχεία με αντίστοιχα k κλειδιά (keys). Έτσι γίνονται συγκρίσεις μεταξύ των κλειδιών των στοιχείων κάθε κόμβου και αντίστοιχα ταξινομούνται τα στοιχεία βάσει των κλειδιών τους. Αυτό έχει ως αποτέλεσμα να γίνονται λιγότερες συγκρίσεις μεταξύ των κλειδιών των στοιχείων να γίνεται πιο αποδοτικά η μεταφορά τους στην κρυφή μνήμη.

Πιο συγκεκριμένα στα δεδομένα των οδικών δικτύων του DIMACS9 αφού έχει δημιουργηθεί η ουρά (Sequence heap) θα υπάρχουν k: τιμές αποστάσεων. Το k έχει οριστεί να είναι ίσο με 4. Έπειτα θα γίνει σύγκριση μεταξύ των 4 αυτών αποστάσεων και θα ταξινομηθούν βάσει της μικρότερης απόστασης από την αρχική κορυφή (s) του δικτύου.

Έτσι αποδεικνύεται ότι η ουρά είναι η καταλληλότερη βάσει της δομής της για μεγάλο αριθμό queries αλλά και αντίστοιχα μεγάλο μέγεθος δεδομένων, όπως τα δεδομένα που μεταφορτώθηκαν για τις αξιολογήσεις.

Αναφορικά με το δεύτερο κριτήριο (DijkstraDec - DjikstraNoDec) είδαμε ότι δεν υπήρχε καμία ουρά του τύπου DijkstraDec που να είναι αποδοτικότερη κάποιας DijkstraNoDec. Άρα εύκολα συμπεραίνουμε ότι η μη ύπαρξη της decrease-key μειώνει τον απαιτούμενο χώρο και τις ταξινομήσεις στην μνήμη του συστήματος μας.

ΚΕΦΑΛΑΙΟ 6.ΣΥΜΠΕΡΑΣΜΑΤΑ-ΠΡΟΟΠΤΙΚΕΣ

6.1. Συμπεράσματα – Επίλογος

Όπως είδαμε από τα παραπάνω αποτελέσματα η Sequence Heap είναι η πιο αποτελεσματική από όλες τις ουρές που εξετάσαμε. Η επόμενη πιο αποδοτική είναι η Binary Heap (No-dec) και ακολουθούν οι Standard Binary Heap και η Pairing Heap που είναι οι λιγότερο αποδοτικές ουρές.

6.2. Προοπτικές

Η παρούσα διπλωματική μπορεί να εξελιχθεί περαιτέρω από τον οποιονδήποτε, από προπτυχιακούς φοιτητές μέχρι και διδάσκοντες. Έχοντας ως δεδομένα τα όσα έχουν προαναφερθεί μπορούν να μελετηθούν από άλλους χρήστες και άλλες ουρές προτεραιότητες που εφαρμόζονται στον Αλγόριθμο του Dijkstra. Έτσι μπορούν να υπάρξουν κι αλλά πιο σαφή συμπεράσματα αναφορικά με τις ουρές και την συμπεριφορά τους.

Άλλη μια επέκταση που μπορεί να γίνει, είναι να εφαρμοστούν οι ουρές προτεραιότητας και για πολύ μεγαλύτερα γραφήματα. Εδώ έχουν μελετηθεί για γραφήματα που αποτελούνται από 200 χιλιάδες κορυφές και 700 χιλιάδες ακμές (New York State) αλλά και με 1 εκατομμύριο κορυφές και 2 εκατομμύρια ακμές (Florida State) και με αριθμό queries που ισούται με 20000. Άρα μπορεί κάποιος να δει την λειτουργία των ουρών προτεραιότητας και σε πολύ μεγαλύτερα γραφήματα (περισσότερες από 10 εκατομμύρια κορυφές και αντίστοιχα ακμές), όπως μεγαλύτερα δίκτυα πολιτειών της Αμερικής όπως της California (1,9 εκατομμύρια κορυφές - σημεία, 4,6 εκατομμύρια ακμές) ή και ολόκληρης της Αμερικής (24 εκατομμύρια κορυφές και 59 εκατομμύρια ακμές) αλλά και να εκτελέσει πολύ μεγαλύτερο πλήθος queries, της τάξεως 100 χιλιάδων τουλάχιστον ή και εκατομμυρίων. Έτσι μπορούμε να εξάγουμε και άλλα συμπεράσματα αναφορικά με την συμπεριφορά της κάθε ουράς για τα γραφήματα πολύ μεγάλου μεγέθους σχετικά με τους Χρόνους εκτέλεσης αλλά και το πλήθος των βασικών λειτουργιών.

Τέλος μέσω της διπλωματικής μπορεί να υλοποιηθούν και εφαρμογές τύπου GPS, μιας και υπάρχουν δεδομένα και πληροφορίες αναφορικά με τις συντομότερες διαδρομές μεταξύ σημείων τα οποία αποτελούνται από πραγματικές συντεταγμένες. Έτσι με την κατάλληλη χρήση της διπλωματικής και την δημιουργία Mock-up screens μπορεί να αναπτυχθεί το παρόν εγχείρημα.

ΚΕΦΑΛΑΙΟ 7.ΒΙΒΛΙΟΓΡΑΦΙΑ-ΠΗΓΕΣ

7.1. Πηγές θεωρίας

1. An Empirical Study of Cache-Oblivious Priority Queues and their Application to the Shortest Path Problem των Benjamin Sach and Raphael Clifford, Bristol University , March 2008
2. Priority Queues and Dijkstra's Algorithm των Mo Chen , Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, October 12, 2007
3. Ding Y., Weiss M.A. (1993) The *K*-D heap: An efficient multi-dimensional priority queue. In: Dehne F., Sack JR., Santoro N., Whitesides S. Algorithms and Data Structures. WADS 1993. Lecture Notes in Computer Science
4. Σχεδίαση Αλγορίθμων των J. Kleinberg, E. Tardos (κεφάλαια 2, 3, 4, 5), 2006
5. Εισαγωγή στους Αλγορίθμους, διαφάνειες του μαθήματος «Εισαγωγή στους Αλγορίθμους» του καθηγητή του πανεπιστημίου Πατρών κύριου Χρήστου Ζαρολιάγκη , 2014
6. Εισαγωγή στους Αλγορίθμους, των T. Cormen, C. Leiserson, R. Rivest, and C. Stein,, Πανεπιστημιακές Εκδόσεις Κρήτης, 2012
7. Αλγόριθμοι και Δομές Δεδομένων – Τα βασικά εργαλεία , των K. Mehlhorn Και P. Sanders, 2014
8. Ενότητα 2η: Συντομότερες διαδρομές, διαφάνειες του μαθήματος του κυρίου Χρήστου Ζαρολιάγκη του Πανεπιστημίου Πατρών, με τίτλο «Αλγόριθμοι και Συνδυαστική Βελτιστοποίηση»
9. Θεωρία γραφημάτων: https://el.wikipedia.org/wiki/Γράφος
10. Αλγόριθμος του Dijkstra: https://el.wikipedia.org/wiki/Αλγόριθμος_του_Ντάικστρα
11. Ουρές: https://el.wikipedia.org/wiki/Ουρά_(δομή_δεδομένων)
12. Στοίβες: https://el.wikipedia.org/wiki/Στοίβα_(δομή_δεδομένων)
13. Ουρές προτεραιότητας: https://el.wikipedia.org/wiki/Ουρά_προτεραιότητας_(δομή_δεδομένων)
14. Μνήμες: https://el.wikipedia.org/wiki/Μνήμη_υπολογιστή
15. Κρυφή μνήμη: https://el.wikipedia.org/wiki/Κρυφή_μνήμη
16. Προσωρινή μνήμη: https://el.wikipedia.org/wiki/Προσωρινή_μνήμη_(υπολογιστές)
17. Binary Heap: https://www.geeksforgeeks.org/binary-heap/
18. Binary Heap (operations): https://en.wikipedia.org/wiki/Binary_heap
19. Pairing Heap: https://www.geeksforgeeks.org/pairing-heap/
20. Pairing Heap (operations): https://en.wikipedia.org/wiki/Pairing_heap
21. Binomial Heap: https://en.wikipedia.org/wiki/Binomial_heap
22. Sequence Heap: https://en.wikipedia.org/wiki/K-D_heap
23. Βέλτιστες μετακινήσεις με χρήση συνδυασμένων μέσων μαζικής μεταφοράς, διπλωματική εργασία της Βούλας Μαχαίρα, 2017

24. Αλγόριθμοι Εύρεσης Βέλτιστων Διαδρομών σε Μη Διασυνδεδεμένες Κινητές Συσκευές, διπλωματική εργασία της Δήμητρας Λύρου, 2022
25. Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" Numerische Mathematik. 1: 269–271.

7.2. Βιβλιοθήκες

1) PGL Library: Αρχείο .zip το οποίο μου δόθηκε από τον διδάσκοντα που περιείχε πολλούς κώδικες και έτοιμες υλοποιήσεις για την ευκολότερη εφαρμογή των ουρών προτεραιότητας στον Αλγόριθμο. Περιείχε και άλλα αρχεία σχετιζόμενα με γραφήματα, δέντρα και μνήμες. Τα αρχεία είναι σε γλώσσα C και C++(.c files και header files .h)
2) Priority-queue-testing: Αρχείο Open-source που μου δόθηκε από τους διδάσκοντες και βρίσκεται στο google. Περιέχει κώδικες σε γλώσσα C και C++ που σχετίζονται με τις ουρές προτεραιότητας και τις μνήμες.
3) STL Library: Έτοιμη βιβλιοθήκη που μου δόθηκε από τον διδάσκοντα με έτοιμες υλοποιήσεις αλγορίθμων και δομών δεδομένων.