ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Εφαρμογή Βασικών Αλγορίθμων Εύρεσης Βέλτιστης Διαδρομής σε Δίκτυα Μεταφορών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΣΑΤΣΑΝΗΣ ΑΘΑΝΑΣΙΟΣ

Επιβλέπων : Ηλίας Κουκούτσης
Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Εφαρμογή Βασικών Αλγορίθμων Εύρεσης Βέλτιστης Διαδρομής σε Δίκτυα Μεταφορών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΣΑΤΣΑΝΗΣ ΑΘΑΝΑΣΙΟΣ

Επιβλέπων : Ηλίας Κουκούτσης
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Νοεμβρίου 2014.

.....	.....	.....
Ηλίας Κουκούτσης	Κωνσταντίνος Παπαοδυσσεύς	Αθανάσιος Μπαλλής
Επ. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2014

.....

ΤΣΑΤΣΑΝΗΣ ΑΘΑΝΑΣΙΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright ©
Τσατσάνης Αθανάσιος, 2014
Ηλίας Κουκούτσης, 2014

Με επιφύλαξη παντός δικαιώματος – All rights reserved

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν στη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Το πρόβλημα της εύρεσης μιας βέλτιστης διαδρομής σε ένα δίκτυο είναι από τα πιο διαδεδομένα προβλήματα για εφαρμογές που απαιτούν την χρήση δικτύων και κατά συνέπεια τα τελευταία χρόνια έχει γίνει και εξακολουθεί να γίνεται σημαντική έρευνα πάνω σε αλγορίθμους επίλυσης τέτοιων προβλημάτων. Ο στόχος της παρούσας εργασίας είναι η επιλογή αλγορίθμου για την επίλυση προβλημάτων του τύπου αυτού. Με τα αποτελέσματα της βιβλιογραφικής αναζήτησης επιλέχθηκε ο αλγόριθμος Dijkstra, επειδή συνδυάζει μεγάλη γενικότητα και εξαιρετικές επιδόσεις. Στην συνέχεια της εργασίας μελετήθηκε περισσότερο και υλοποιήθηκε με παραπάνω από έναν τρόπους ο αλγόριθμος αυτός. Δευτερεύον στόχος της εργασίας είναι η μελέτη του αλγόριθμου Κ συντομότερων διαδρομών, ως μια επέκταση του αλγορίθμου Dijkstra με σκοπό την εύρεση των Κ αμέσως καλύτερων διαδρομών σε ένα δίκτυο. Η εφαρμογή των αλγορίθμων αυτών γίνεται πάνω στα μεταφορικά δίκτυα και έχει στόχο την υλοποίηση παραλλαγών τους σε βαθμό τέτοιο ώστε να μπορούν να διαχειριστούν βασικές δομές που χρησιμοποιούνται στην παράσταση μεταφορικών δικτύων. Στην αρχή της παρούσας εργασίας δίνονται κάποιοι βασικοί ορισμοί της θεωρίας γράφων και στην συνέχεια παρουσιάζεται το πρόβλημα του συντομότερου μονοπατιού. Παρουσιάζονται επίσης οι πιο βασικοί αλγόριθμοι που επιλύουν το πρόβλημα στις διάφορες μορφές του. Στο επόμενο τμήμα της εργασίας παρουσιάζεται εκτενέστερα ο αλγόριθμος Dijkstra και K-Best καθώς και ο υλοποιημένος κώδικας για την εφαρμογή τους στις προγραμματιστικές γλώσσες Python και C. Τέλος χρησιμοποιήθηκε ένα εμπορικό πρόγραμμα για την εύρεση των βέλτιστων διαδρομών σε πραγματικά δίκτυα μεταφορών προκειμένου να επαληθευτεί η ορθότητα των αποτελεσμάτων των υλοποιήσεων που έγιναν στα πλαίσια της παρούσας εργασίας.

Λέξεις Κλειδιά: <<Πρόβλημα Εύρεσης Βέλτιστων Διαδρομών σε Γράφους, Αλγόριθμος Dijkstra, Αλγόριθμος Εύρεσης Κ Συντομότερων Μονοπατιών>>

Abstract

The problem of finding the shortest path in a network is one of the most common problems for application that require the usage of networks and as a result during the last years there has been an ongoing research on algorithms which solve such problems. The purpose of this project is the selection of the appropriate algorithm for solving these types of problems. After a bibliographic research, the algorithm that was chosen is Dijkstra because of both its generic nature and high performance. This project also focuses on the implementation of the Dijkstra algorithm in various ways. The secondary part of this project is the study of the algorithm for K shortest paths as an extension of the Dijkstra algorithm for finding the K additional shortest paths in a network. The application of these algorithms focuses on transportation networks and aims in building different variations that will be able to manage basic structures used in the representation of such networks. In the beginning of this project there is a compilation of basic definitions of graph theory which is then followed by an analysis of the shortest path problem. Moving on, there is an introduction to the most basic algorithms that solve this problem and its different variations. From these solutions, the Dijkstra and K-best algorithms are thoroughly analyzed in the next part where their implementations in both C and Python are also included. Finally, in order to confirm the accuracy of these implementations, a commercial software was used for finding the shortest paths in real transportation networks.

Keywords: <<Shortest Path Problem, Dijkstra Algorithm, K Shortest Paths Algorithm>>

Θέλω να ευχαριστήσω θερμά την οικογένεια μου για
την στήριξη που μου παρείχε όλα αυτά τα χρόνια

Ευχαριστώ όλους όσους με βοήθησαν και
με στήριξαν κατά την διάρκεια των
σπουδών μου.

Επίσης ευχαριστώ τον κ. Ηλία Κουκούτση
που μου επέτρεψε να ασχοληθώ
με την παρούσα εργασία

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

1 Εισαγωγή 15

1.1 Εισαγωγή 15

1.2 Αντικείμενο Διπλωματικής 15

1.3 Οργάνωση Κειμένου 16

2 Βασικοί Ορισμοί Θεωρίας Γράφων 17

2.1 Ορισμός του Γράφου 17

2.2 Εφαρμογές Γράφων 19

2.3 Δομές Παράστασης Γράφων σε Υπολογιστικά Συστήματα 19

3 Το Πρόβλημα του Συντομότερου Μονοπατιού σε Γράφο 21

3.1 Ορισμός Προβλήματος 21

3.2 Το Πρόβλημα στα Οδικά Δίκτυα 22

3.3 Το Πρόβλημα σε Άλλες Εφαρμογές 23

3.4 Συναφή Προβλήματα 23

4 Αλγόριθμοι Επίλυσης του Προβλήματος 25

4.1 Αναζήτηση κατά Πλάτος 25

4.1.1 Γενικά 25

4.1.2 Βήματα 25

4.1.3 Ψευδοκώδικας 26

4.1.4 Πολυπλοκότητα 26

4.1.5 Εφαρμογές 26

4.2 Αναζήτηση κατά Βάθος 27

4.2.1 Γενικά 27

4.2.2 Ψευδοκώδικας 27

4.2.3 Πολυπλοκότητα 27

4.2.4 Εφαρμογές 28

4.3 Ο Αλγόριθμος Dijkstra 28

4.3.1 Γενικά 28

4.4 Ο Αλγόριθμος Bellman – Ford 29

4.4.1 Γενικά 29

4.4.2 Περιγραφή Αλγορίθμου *29*
4.4.3 Ψευδοκώδικας *30*
4.4.4 Πολυπλοκότητα *30*
4.5 Ο Αλγόριθμος A* 31
4.5.1 Γενικά *31*
4.5.2 Περιγραφή Αλγορίθμου *32*
4.5.3 Βήματα Αλγορίθμου *32*
4.5.4 Ψευδοκώδικας *32*
4.5.5 Πολυπλοκότητα *33*
4.5.6 Εφαρμογές *34*
4.6 Ο Αλγόριθμος Floyd – Warshall 34
4.6.1 Γενικά *34*
4.6.2 Περιγραφή Αλγορίθμου *34*
4.6.3 Ψευδοκώδικας *35*
4.6.4 Πολυπλοκότητα *35*
4.6.5 Εφαρμογές *36*
4.7 Ο Αλγόριθμος Johnson 36
4.7.1 Γενικά *36*
4.7.2 Περιγραφή Αλγορίθμου *36*
4.7.3 Πολυπλοκότητα *37*
4.8 Ο Αλγόριθμος Viterbi 37
4.8.1 Γενικά *37*
4.8.2 Περιγραφή Αλγορίθμου *37*
4.8.3 Πολυπλοκότητα *38*

5 Εκτενέστερη Μελέτη του Αλγορίθμου Dijkstra **39**

5.1 Περιγραφή Αλγορίθμου 39
5.2 Βήματα Αλγορίθμου 40
5.3 Ψευδοκώδικας 41
5.4 Παράδειγμα Εκτέλεσης 41
5.5 Απόδειξη Ορθότητας Αλγορίθμου 46

6 Ο Αλγόριθμος K Συντομότερων Μονοπατιών **49**

6.1 Γενικά 49
6.2 Περιγραφή Αλγορίθμου 50

6.3 Ψευδοκώδικας 51
6.4 Παράδειγμα Εκτέλεσης 52
6.5 Εφαρμογές 56

7 Υλοποίηση Αλγορίθμου Dijkstra σε Python 57

7.1 Δημιουργία Αρχείου Λίστας Γειτνίασης (Adjacency List) 57
7.2 Δημιουργία Αρχείου Δομής Γράφου 60
7.3 Αρχείο Δημιουργίας ενός Γράφου από ένα αρχείο Λίστας Γειτνίασης 64
7.4 Αρχείο Εκτέλεσης Αλγορίθμου Dijkstra 65
7.4.1 Κώδικας 65
7.4.2 Παράδειγμα Εκτέλεσης Κώδικα 69
7.5 Αρχείο Κλήσης και Εκτύπωσης του Αλγορίθμου Dijkstra για έναν Γράφο δοσμένο σε αρχείο μορφής λίστας γειτνίασης 73
7.6 Εφαρμογή του Υλοποιημένου σε Python Αλγορίθμου Dijkstra σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών 74

8 Υλοποίηση Αλγορίθμου Dijkstra σε GNU C 77

8.1 Κώδικας 78
8.2 Επεξήγηση Κώδικα 82
8.3 Παράδειγμα εκτέλεσης της συνάρτησης Dijkstra 83
8.4 Εφαρμογή του Υλοποιημένου σε GNU C Αλγορίθμου Dijkstra σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών με έμφαση στην ταχύτητα λειτουργίας 87

9 Υλοποίηση Αλγορίθμου K - Συντομότερων Μονοπατιών σε Python 89

9.1 Αρχείο εκτέλεσης του Αλγορίθμου K - Συντομότερων Μονοπατιών 89
9.2 Αρχείο Κλήσης και εκτύπωσης του Αλγορίθμου K – Συντομότερων Μονοπατιών για έναν Γράφο δοσμένο σε αρχείο μορφής Λίστας Γειτνίασης 92
9.3 Εφαρμογή του Υλοποιημένου σε Python Αλγορίθμου K-Best σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών 94

10 Συμπεράσματα - Προτάσεις για συνέχιση της εργασίας 97

10.1 Συμπεράσματα 97
10.2 Μελλοντικές επεκτάσεις 99

Βιβλιογραφία 101

1

Εισαγωγή

1.1 Εισαγωγή

Το πρόβλημα της δρομολόγησης είναι από τα πλέον μελετημένα προβλήματα που έχουν στόχο την βελτιστοποίηση ροής φορτίου σε εφαρμογές που εμπλέκονται δίκτυα. Οι διαφορετικές τεχνικές σχεδίασης αλγορίθμων έχουν οδηγήσει στην ανάπτυξη πολλών αλγορίθμων δρομολόγησης, οι οποίοι έχουν μελετηθεί τόσο θεωρητικά όσο και πειραματικά. Το συμπέρασμα μετά από τις παραπάνω μελέτες ήταν πως δεν υπάρχει γενικός αλγόριθμος ο οποίος να εκτελείται αποδοτικά στα πλαίσια όλων των πιθανών απαιτήσεων που μπορεί να έχει μια εφαρμογή, για κάθε εφαρμογή. Ανάλογα με την μορφή του δικτύου και τις απαιτήσεις χρόνου και μνήμης υπάρχουν διαφορετικοί αλγόριθμοι που είναι κατάλληλοι στην μία περίπτωση από ότι στην άλλη.
Μερικά από τα δίκτυα στα οποία απαντώνται συχνά προβλήματα δρομολόγησης με ιδιαίτερο ενδιαφέρον, είναι τα δίκτυα υπολογιστών, τα τηλεπικοινωνιακά δίκτυα, τα συγκοινωνιακά δίκτυα και τα δίκτυα βιομηχανικής παραγωγής.

1.2 Αντικείμενο Διπλωματικής

Στην παρούσα διπλωματική εργασία ζητήθηκε η υλοποίηση του αλγορίθμου Dijkstra στις προγραμματιστικές γλώσσες Python και C καθώς και η υλοποίηση του αλγορίθμου K συντομότερων μονοπατιών στην προγραμματιστική γλώσσα Python. Απώτερος στόχος των παραπάνω είναι η μελέτη των αλγορίθμων αυτών και προσπάθεια παραλλαγής τους ώστε να μπορούν να ικανοποιούν τις απαιτήσεις που απαιτούνται στα δίκτυα μεταφορών. Η διπλωματική αυτή εργασία ολοκληρώνεται με την επιβεβαίωση της λειτουργίας των υλοποιημένων αλγορίθμων μέσω σύγκρισης τους με ήδη υλοποιημένους αλγορίθμους στο πρόγραμμα TransCAD.

1.3 Οργάνωση Κειμένου

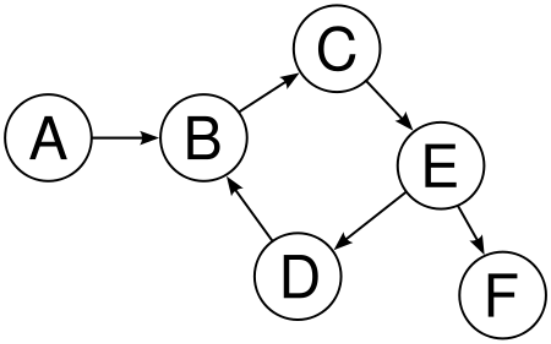
Η διάρθρωση της παρούσας διπλωματικής εργασίας ακολουθεί την εξής σειρά:
Αρχικά παρουσιάζονται οι πολύ βασικές έννοιες της θεωρίας γράφων καθώς θα χρησιμοποιηθούν για την επεξήγηση των διάφορων αλγορίθμων στην συνέχεια. Επίσης αναφέρονται διάφορες εφαρμογές που χρησιμοποιούνται γράφοι. Στην συνέχεια παρουσιάζεται το πρόβλημα του συντομότερου μονοπατιού ανάμεσα σε δυο σημεία ενός δικτύου και επίσης αναφέρονται διάφορες εφαρμογές που απαιτούν την λύση αυτού του προβλήματος. Σε επόμενες ενότητες παρουσιάζονται οι διάφοροι βασικοί αλγόριθμοι, οι οποίοι εφαρμόζονται πάνω σε δίκτυα για την επίλυση του προβλήματος του συντομότερου μονοπατιού και των διάφορων μορφών του. Στην συνέχεια παρουσιάζεται εκτενέστερα ο αλγόριθμος Dijkstra και K-Best καθώς και παραδείγματα εκτέλεσης τους. Τέλος παρουσιάζεται ο υλοποιημένος κώδικας του αλγορίθμου Dijkstra στις προγραμματιστικές γλώσσες Python και C καθώς και ο υλοποιημένος κώδικάς του αλγορίθμου K-Best στην προγραμματιστική γλώσσα Python.

2

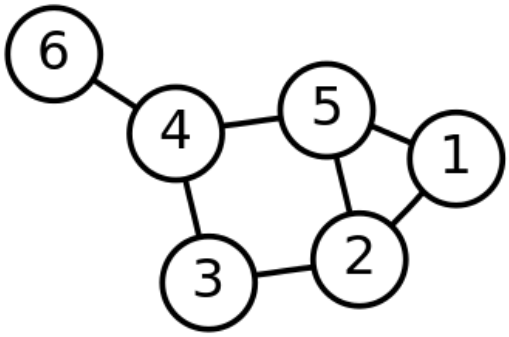
Βασικοί ορισμοί θεωρίας γράφων

2.1 Ορισμός του Γράφου

Ένα σύνολο από οντότητες συνδεδεμένες μεταξύ τους μπορεί να παρασταθεί σχηματικά με την παρακάτω μορφή:



Σχήμα 1



Σχήμα 2

Η παραπάνω σχηματική διάταξη ονομάζεται Γράφος ή Δίκτυο. Οι γράφοι είναι ένα από τα βασικά αντικείμενα της μελέτης των διακριτών μαθηματικών. Η Θεωρία Γράφων είναι η επιστήμη που χρησιμοποιεί μαθηματικά εργαλεία ώστε να μελετά και να αναλύει γράφους.

Ένας πιο τυπικός ορισμός ορίζει τον γράφο ως μια οπτική αναπαράσταση των σχέσεων που υπάρχουν μεταξύ ορισμένων οντοτήτων. Οι οντότητες αυτές αναπαριστούνται ως σημεία και οι σχέσεις ως καμπύλες, όλα σε σχέση με ένα σύστημα. Ένας άλλος ορισμός που κινείται στο ίδιο εννοιολογικό πλαίσιο της οπτικής αναπαράστασης αναγνωρίζει τον γράφο ως απεικόνιση αποτελούμενη από ένα σύνολο σημείων (κορυφών ή κόμβων) που συνδέονται με γραμμές (ακμές).

Στην επιστήμη των μαθηματικών, ένας γράφος θεωρείται ένα ζεύγος G = (V,E) αποτελούμενο από ένα σύνολο V(G) κόμβων και ένα σύνολο E(G) ακμών, με το E(G) είναι σύνολο ζευγών με στοιχεία από το σύνολο V (μία ακμή μπορεί να θεωρηθεί σαν ένα ζεύγος κόμβων).

Οι κόμβοι που ανήκουν σε μία ακμή θεωρούνται ως τα άκρα της ακμής. Ένας κόμβος μπορεί να υπάρχει στο γράφο αλλά να μην ανήκει σε κάποια ακμή. Ως τάξη του γράφου ορίζουμε το |V(G)| (αριθμός των κόμβων). Ως μέγεθος του γράφου ορίζεται το |E(G)| (αριθμός των ακμών). Τάξη ενός κόμβου ορίζεται ως ο αριθμός των ακμών που ανήκει ο κόμβος αυτός, και μία ακμή που έχει ως άκρα της τον ίδιο κόμβο (βρόχος) μετράται δύο φορές. Μία ακμή (u,v) θεωρείται ότι έχει ως άκρα τους κόμβους u,v και εν συντομία μπορεί να γραφτεί uv. Ένας γράφος G λοιπόν ορίζεται αυστηρά μέσω των V(G) και E(G) συνόλων αλλά και μιας συνάρτησης f_G μέσω της οποίας ένα ζευγάρι κόμβων του G, σχετίζεται με μια ακμή του συνόλου E(G). Έστω μια ακμή e και δύο κόμβοι v_1,v_2. Εφόσον ισχύει ότι $f_G(e)= v_1v_2$ τότε λέμε ότι η e ενώνει τα v_1,v_2 τα οποία ορίζονται ως άκρα της e. Υπάρχουν δύο κατηγορίες γράφων, ο κατευθυνόμενος ή προσανατολισμένος και ο μη-κατευθυνόμενος γράφος. Σε έναν κατευθυνόμενο γράφο (*Σχήμα 1*) οι ακμές του είναι διανύσματα που σημαίνει ότι έχουν κατεύθυνση. Αυτή η κατεύθυνση υποδηλώνει ότι για μία ακμή e που έχει ως άκρα τους κόμβους v_1 και v_2 θα έχει έναν από τους δύο κόμβους ως την αρχή του διανύσματος και τον άλλο ως το τέλος. Στην περίπτωση λοιπόν ενός κατευθυνόμενου γράφου εν γένει θα θεωρείται ότι ($f_G(e_1) = v_1v_2$) $\neq$ ($f_G(e_2) = v_2v_1$). Σε έναν μη-κατευθυνόμενο γράφο (*Σχήμα 2*) οι ακμές δεν έχουν κατεύθυνση και ισχύει ότι ($f_G(e) = v_1v_2 = v_2v_1$). Πλήρης αποκαλείται ένας γράφος που έχει ακμές για κάθε πιθανό ζεύγος μέσα από το σύνολο των κόμβων του. Αραιός γράφος αποκαλείται ένας γράφος που περιέχει λίγες ακμές ενώ πυκνός όταν περιέχει πάρα πολλές. Εάν σε έναν γράφο υπάρχουν δύο ακμές με έναν κοινό κόμβο τότε θεωρείται ότι οι μη κοινοί κόμβοι συνδέονται μέσω του κοινού κόμβου. Για παράδειγμα έστω μια ακμή αβ και μια ακμή βγ. Ο κοινός κόμβος είναι ο β επομένως ο κόμβος α συνδέεται με τον κόμβο γ μέσω του κόμβου β. Προεκτείνοντας τον παραπάνω ορισμό, μονοπάτι ή αλλιώς διαδρομή ανάμεσα σε δύο κόμβους ενός γράφου υπάρχει εάν οι δύο αυτοί κόμβοι συνδέονται μέσω άλλων κόμβων.

Για έναν μη προσανατολισμένο γράφο όταν δεν υπάρχει μονοπάτι για τουλάχιστον έναν κόμβο προς τουλάχιστον έναν άλλον λέμε ότι ο γράφος είναι μη συνεκτικό ή μη συνδεδεμένος. Για προσανατολισμένο γράφο διακρίνουμε τις εξής περιπτώσεις:

- Εάν για ένα τουλάχιστον ζεύγος κόμβων δεν υπάρχει μονοπάτι είτε από τον έναν προς το άλλον είτε αντίστροφα λέμε ότι ο γράφος είναι μη συνεκτικός ή μη συνδεδεμένος.
- Εάν για κάθε κόμβο του γράφου υπάρχει μονοπάτι προς κάθε άλλο κόμβο του γράφου λέμε ότι ο γράφος είναι ισχυρά συνεκτικός ή ισχυρά συνδεδεμένος.
- Εάν ο γράφος δεν είναι μη συνεκτικός αλλά για τουλάχιστον ένα ζεύγος κόμβων δεν υπάρχει μονοπάτι από τον έναν κόμβο στον άλλον λέμε ότι ο γράφος είναι ασθενώς συνεκτικός ή ασθενώς συνδεδεμένος.

Σε έναν γράφο οι ακμές του μπορεί να φέρουν τιμές (βάρη ή κόστη) οι οποίες υποδηλώνουν κάποιο φυσικό ή τεχνικό μέγεθος σχετιζόμενο με το πρόβλημα στο οποίο ο γράφος αναπαριστά κάποια πραγματική οντότητα, όπως για παράδειγμα ένα οδικό δίκτυο ή ένα δίκτυο υπολογιστών όπου τα βάρη πιθανόν αντιστοιχούν σε μήκη δρόμων ή ταχύτητες ζεύξεων αντίστοιχα.

2.2 Εφαρμογές Γράφων

Οι γράφοι μπορούν να χρησιμοποιηθούν για την μοντελοποίηση πολλών ειδών σχέσεων και διαδικασιών σε φυσικά, βιολογικά [MRK04], κοινωνικά και πληροφοριακά συστήματα. Πολλά πρακτικά προβλήματα μπορούν να αναπαρασταθούν από γράφους.
Στην επιστήμη των υπολογιστών γράφοι χρησιμοποιούνται για την αναπαράσταση δικτύων επικοινωνιών, δομές δεδομένων, υπολογιστικές μηχανές, διαγράμματα ροής, κ.λπ. Για παράδειγμα, μία ιστοσελίδα μπορεί να αναπαρασταθεί ως ένας κατευθυνόμενος γράφος, στον οποίο οι κόμβοι θεωρούνται οι σελίδες και οι κατευθυνόμενες ακμές θεωρούνται τα links από μία σελίδα σε μία άλλη. Μία παρόμοια προσέγγιση μπορεί να υπάρξει και για προβλήματα στις μεταφορές, στη βιολογία, στον σχεδιασμό ηλεκτρονικών κυκλωμάτων και σε πολλά άλλα πεδία. Η ανάπτυξη αλγορίθμων που διαχειρίζονται αυτούς τους γράφους παρουσιάζουν τεράστιο ενδιαφέρον για την επιστήμη των υπολογιστών. Ο μετασχηματισμός γράφων χρησιμοποιείται συχνά και αναπαρίσταται από τα graph rewrite systems.

Μέθοδοι θεωρίας γράφων σε διάφορες μορφές έχει αποδειχθεί ότι είναι πολύ χρήσιμες και στη γλωσσολογία καθώς στις φυσικές γλώσσες χρησιμοποιούνται διακριτές δομές.
Η θεωρία γράφων χρησιμοποιείται επίσης για τη μελέτη των μορίων και ατόμων στην επιστήμη της χημείας και της φυσικής. Οι γράφοι εκεί είναι ικανοί να αναπαραστήσουν πολύπλοκες μοριακές και ατομικές δομές και η μελέτη αυτών μπορεί στατιστικά να συλλέξει ιδιότητες σχετικές με την τοπολογία των ατόμων.

Η θεωρία γράφων χρησιμοποιείται επίσης και στην κοινωνιολογία ως ένας τρόπος για την παράσταση σχέσεων μεταξύ συνόλων που είναι στατιστικά δείγματα από διάφορες μελέτες. [PK11]
Στη βιολογία, οι κόμβοι των γράφων μπορούν να αναπαραστήσουν περιοχές όπου υπάρχουν συγκεκριμένα είδη ζώων και οι ακμές δυνατές μεταναστεύσεις ή κινήσεις μεταξύ αυτών των περιοχών. Τέτοιες πληροφορίες είναι χρήσιμες για την εύρεση μοτίβων εξάπλωσης ασθενειών ή μεταναστεύσεων ανάλογα με το κάθε είδος.

Στα μαθηματικά, οι γράφοι είναι χρήσιμοι στη γεωμετρία και στην τοπολογία. Στην άλγεβρα η θεωρία γράφων συνδέεται στενά με την θεωρία συνόλων.

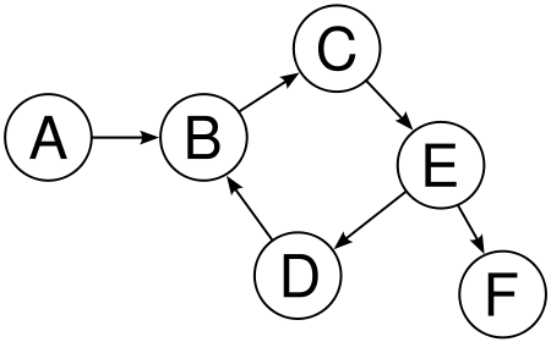
2.3 Δομές Παράστασης Γράφων σε Υπολογιστικά Συστήματα

Υπάρχουν διάφοροι τρόποι για να αποθηκευτεί ένας γράφος σε ένα υπολογιστικό σύστημα. Η δομή δεδομένων που χρησιμοποιείται εξαρτάται και από τον γράφο αυτόν κάθε αυτόν αλλά και από τον αλγόριθμο που θα χρησιμοποιηθεί για την επεξεργασία του γράφου αυτού. Θεωρητικά οι δύο βασικές δομές παράστασης ενός γράφου είναι η λίστα και ο πίνακας αλλά για κατάλληλες εφαρμογές μπορεί να χρησιμοποιηθεί δομή που συνδυάζει και τα δύο. Δομές λιστών συνήθως χρησιμοποιούνται όταν ο γράφος είναι αραιός καθώς απαιτούν λιγότερη μνήμη για την αποθήκευσή του. Οι δομές πινάκων καταναλώνουν πολλή μνήμη αλλά είναι χρήσιμες για εφαρμογές που απαιτούν την γρήγορη προσπέλαση του γράφου.

Οι δομές λίστας περιλαμβάνουν τον πίνακα συνδέσεων (connectivity table) ο οποίος είναι μία λίστα από ζεύγη κόμβων, δηλαδή, μια λίστα με τις ακμές του γράφου, και την λίστα γειτνίασης (adjacency list) η οποία περιλαμβάνει για κάθε κόμβο τους κόμβους που συνδέονται με αυτόν (γείτονες).

Οι δομές πινάκων περιλαμβάνουν τον πίνακα γειτνίασης (adjacency matrix) στον οποίο οι γραμμές και οι στήλες είναι δείκτες του κάθε κόμβου και κάθε στοιχείο ενός πίνακα περιλαμβάνει 0 ή 1 ανάλογα με το αν υπάρχει ακμή που συνδέει τους κόμβους i,j αν υπάρχει (κόμβος i στην γραμμή i, κόμβος j στην στήλη j). Μια άλλη παρόμοια δομή με τον πίνακα γειτνίασης είναι ο πίνακας κόστους (distance matrix) όπου τα στοιχεία του (i,j) αντί για 0 ή 1 έχουν τα αντίστοιχα βάρη των ακμών που .

παρακάτω παρουσιάζεται ένα παράδειγμα προσανατολισμένου γράφου και των δομών αναπαράστασης του από ένα υπολογιστικό σύστημα.



Ο πίνακας γειτνίασης του παραπάνω γράφου θα είναι:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Η λίστα συνδέσεων είναι:

```
[(A,B),(B,C),(C,E),(E,F),(E,D),(D,B)]
```

και η λίστα γειτνίασης είναι:

```
[(A,(B)),(B,(C)),(C,(E)),(E,(D,F)),(D,(B))]
```

Για γράφους με βάρη στις ακμές τους ανάλογη πληροφορία μπορεί να προστίθεται στις παραπάνω δομές ώστε να είναι πλήρεις.

3

Το Πρόβλημα του Συντομότερου Μονοπατιού – Shortest Path Problem

Στην θεωρία γράφων, το πρόβλημα του συντομότερου μονοπατιού είναι ένα πρόβλημα στο οποίο απαιτείται η εύρεση ενός μονοπατιού μεταξύ δύο κόμβων σε ένα γράφο όπου το άθροισμα των βαρών των ακμών που αποτελούν το μονοπάτι να είναι το ελάχιστο δυνατό.

3.1 Ορισμός Προβλήματος

Το πρόβλημα του συντομότερου μονοπατιού μπορεί να οριστεί για γράφους που είναι είτε προσανατολισμένοι είτε μη-προσανατολισμένοι. Παρακάτω θα οριστεί για μη προσανατολισμένο γράφο. Για προσανατολισμένο γράφο ο ορισμός του μονοπατιού προϋποθέτει ότι δύο κόμβοι συνδέονται με μία ακμή η οποία έχει κατεύθυνση.

Δύο κόμβοι θεωρούνται γειτονικοί όταν είναι τα δύο άκρα μιας ακμής του γράφου. Ένα μονοπάτι ορίζεται ως μια ακολουθία από κόμβους:
$P = (v_1,v_2,...,v_n)$ E V x V x … x V με τον κόμβο v_i να είναι γειτονικός του v_{i+1} για $1 \leq i < n$. Ένα τέτοιο μονοπάτι P θεωρείται ότι έχει μήκος n από τον κόμβο v_1 έως τον v_n.
Έστω $e_{i,j}$ μια ακμή με άκρα τους κόμβους v_i και v_j δοσμένης μιας συνάρτησης βάρους f : E -> R, και ενός μη κατευθυνόμενου γράφου G, το συντομότερο μονοπάτι από τον v έως τον v' είναι το μονοπάτι $P = (v_1,v_2,...,v_n)$ (με v_1=v και v_n= v') όπου για όλα τα πιθανά n το άθροισμα $\Sigma f(e_{i,i+1})$: i = 1 εως n-1 ελαχιστοποιείται. Όταν το βάρος κάθε ακμής θεωρείται μονάδα, το πρόβλημα είναι ισοδύναμο με το να βρεθεί το μονοπάτι με τις λιγότερες ακμές.

Το πρόβλημα καλείται μερικές φορές και ως πρόβλημα συντομότερου μονοπατιού ενός ζεύγους ώστε να διαφοροποιηθεί από τις παρακάτω περιπτώσεις άλλων προβλημάτων:

- Το πρόβλημα συντομότερου μονοπατιού μονής πηγής (single-source), στο οποίο απαιτείται να βρεθούν τα συντομότερα μονοπάτια από έναν κόμβο πηγή προς όλους τους υπόλοιπους κόμβους στο γράφο.
- Το πρόβλημα συντομότερου μονοπατιού μονού προορισμού (single-destination), στο οποίο απαιτείται να βρεθούν τα συντομότερα μονοπάτια από όλους τους κόμβους σε έναν κατευθυνόμενο γράφο προς έναν μοναδικό κόμβο-προορισμό.
- Το πρόβλημα συντομότερου μονοπατιού όλων των ζευγαριών (all-pairs), στο οποίο απαιτείται να βρεθούν τα συντομότερα μονοπάτια μεταξύ κάθε ζεύγους κόμβων στο γράφο.

Οι παραπάνω γενικεύσεις προβλημάτων επιλύονται μέσω κατάλληλων αλγορίθμων η αποδοτικότητα των οποίων εξαρτάται κάθε φορά από το εκάστοτε πρόβλημα.

3.2 Το Πρόβλημα στα Οδικά Δίκτυα

Ένα οδικό δίκτυο μπορεί να θεωρηθεί ως ένας γράφος με θετικά μέρη στις ακμές του. Οι κόμβοι αναπαριστούν διασταυρώσεις και οι ακμές αναπαριστούν κομμάτια ενός δρόμου μεταξύ δύο διασταυρώσεων. Το βάρος μιας ακμής θεωρείται το μήκος του δρόμου ή ο χρόνος που χρειάζεται για να διασχιστεί ο συγκεκριμένος δρόμος. Χρησιμοποιώντας κατευθυνόμενες ακμές μπορούν να αναπαρασταθούν δρόμοι μονής κατεύθυνσης. Αυτή η κατηγορία γράφων χρησιμοποιείται για την αναπαράσταση οδικών δικτύων όπου υπάρχουν οδοί μονής κατεύθυνσης αλλά και διπλής όπως οι αυτοκινητόδρομοι. Υπάρχουν αρκετοί αλγόριθμοι οι οποίοι εκμεταλλεύονται την παραπάνω ιδιότητα ώστε να υπολογίζουν το συντομότερο μονοπάτι πολύ γρηγορότερα από ότι θα το υπολόγιζαν σε ένα γενικό γράφο.

Όλοι αυτοί οι αλγόριθμοι αποτελούν σε δύο φάσεις. Στην πρώτη φάση, ο γράφος προεπεξεργάζεται χωρίς να είναι γνωστός ο κόμβος-πηγή ή ο κόμβος-στόχος. Αυτή η φάση μπορεί να πάρει μέρες για πραγματικά δεδομένα. Η δεύτερη φάση είναι η φάση υποβολής ερωτημάτων. Σε αυτή τη φάση ο κόμβος-πηγή και ο κόμβος-στόχος είναι γνωστοί. Ο χρόνος εκτέλεσης της δεύτερης φάσης είναι γενικά μικρότερος του ενός δευτερολέπτου. Η ιδέα βασίζεται στο ότι το οδικό δίκτυο θεωρείται στατικό, έτσι ώστε η φάση προεπεξεργασίας να γίνει μόνο μία φορά και στη συνέχεια όταν θα απαιτείται η εύρεση του ελάχιστου μονοπατιού μεταξύ δύο κόμβων, ο χρόνος εκτέλεσης να είναι πολύ μικρός. Ο αλγόριθμος (with the fastest known query time) καλείται hub labeling και είναι ικανός να υπολογίσει το συντομότερο μονοπάτι σε οδικά δίκτυα της Ευρώπης ή της Αμερικής σε κλάσματα μικροδευτερολέπτου[ADG+10]. Άλλες τεχνικές που χρησιμοποιούνται ALT, Arc Flags, Contraction Hierarchies, Transit Node Routing, Rich Based Pruning, Labeling.

3.3 Το Πρόβλημα σε Άλλες Εφαρμογές

Οι αλγόριθμοι εύρεσης συντομότερου μονοπατιού εφαρμόζονται ώστε να βρίσκουν αυτόματα κατευθύνσεις μεταξύ φυσικών τοποθεσιών, όπως οδηγίες πλοήγησης σε ιστοσελίδες δικτυακής χαρτογράφησης όπως το Mapquest και Google Maps. Γι' αυτές τις εφαρμογές υπάρχουν ειδικοί ταχύτατοι αλγόριθμοι. [SP09]

Άλλες εφαρμογές των αλγορίθμων αυτών είναι πάνω σε γράφους όπου οι κόμβοι περιγράφουν καταστάσεις και οι ακμές περιγράφουν πιθανές μεταβάσεις, και ζητείται να βρεθεί η βέλτιστη ακολουθία καταστάσεων από μία αρχική προς μίας τελική κατάσταση. Για παράδειγμα, αν οι κόμβοι αναπαριστούν καταστάσεις ενός πάζλ, όπως ο κύβος του Rubik, και κάθε ακμή αναπαριστά μία κίνηση, αλγόριθμοι συντομότερου μονοπατιού μπορούν να χρησιμοποιηθούν για να βρεθεί η λύση του πάζλ με τον μικρότερο αριθμό κινήσεων.

Στον τομέα των δικτύων ή των τηλεπικοινωνιών τέτοιοι αλγόριθμοι πολλές φορές καλούνται αλγόριθμοι μονοπατιού ελάχιστης καθυστέρησης και συνήθως συνδέονται με το πρόβλημα του ευρύτερου μονοπατιού. Για παράδειγμα, ένας αλγόριθμος μπορεί να ψάχνει the shortest (min-delay) widest path or the widest shortest (min-delay) path.

Άλλες εφαρμογές είναι plant and facility layout, ρομποτική, μεταφορές και σχεδιασμός VLSI. [CD96]

3.4 Συναφή Προβλήματα

- Ευκλείδειο συντομότερο μονοπάτι για προβλήματα συντομότερου μονοπατιού σε υπολογιστική γεωμετρία.
- Το πρόβλημα του πλανόδιου πωλητή είναι ένα πρόβλημα που απαιτείται να βρεθεί το συντομότερο μονοπάτι που διασχίζει κάθε κόμβο του γράφου μόνο μία φορά και επιστρέφει στην αρχή. Σε αντίθεση με το πρόβλημα συντομότερου μονοπατιού το οποίο μπορεί να λυθεί σε πολυωνυμικό χρόνο για γράφους χωρίς αρνητικούς κύκλους, το πρόβλημα του πλανόδιου πωλητή είναι NP complete και όπως μέχρι πιστεύεται δεν μπορεί να λυθεί αποδοτικά για μεγάλα σύνολα δεδομένων. Το πρόβλημα της εύρεσης του μακρύτερου μονοπατιού σε ένα γράφο είναι επίσης NP complete.
- Το πρόβλημα του καναδού ταξιδιώτη και το πρόβλημα του στοχαστικού συντομότερου μονοπατιού είναι γενικεύσεις όπου είτε ο γράφος δεν είναι απολύτως γνωστός για αυτόν που κινείται είτε ο γράφος αλλάζει στο χρόνο είτε οι μεταβάσεις μεταξύ των καταστάσεων είναι πιθανοτικές.
- Το πρόβλημα του ευρύτερου μονοπατιού αναζητά το μονοπάτι ώστε το ελάχιστο βάρος για κάθε ακμή να είναι όσο μεγαλύτερο γίνεται.

4

Αλγόριθμοι επίλυσης του προβλήματος

4.1 Αναζήτηση κατά πλάτος (Breadth-First Search - BFS)

4.1.1 Γενικά

Στη θεωρία γράφων αναζήτηση κατά πλάτος (Breadth-First Search - BFS) είναι μια στρατηγική αναζήτησης σε γράφο. Ο αλγόριθμος αυτός δεν εντοπίζει το συντομότερο μονοπάτι από μόνος του αλλά χρησιμοποιείται ευρέως σε άλλους αλγορίθμους που στόχο έχουν την εύρεση αυτού του μονοπατιού. Η αναζήτηση με BFS γίνεται με τον εξής τρόπο: Έχοντας ήδη επισκεφθεί έναν κόμβο του γράφου η μόνη προσπέλαση που μπορεί να γίνει είναι σε γειτονικούς κόμβους. Το BFS ξεκινά από έναν κόμβο-πηγή (root node) και επισκέπτεται όλους τους γειτονικούς κόμβους του. Στη συνέχεια, για κάθε γειτονικό κόμβο επισκέπτεται τους δικούς του γειτονικούς κόμβους που δεν έχει επισκεφτεί στο παρελθόν.

4.1.2 Βήματα

Ο αλγόριθμος χρησιμοποιεί μια δομή ουράς ώστε να αποθηκεύει ενδιάμεσα αποτελέσματα καθώς διασχίζει τον γράφο:
Βήμα 1: Πρόσθεσε το κόμβο αρχή στην ουρά
Βήμα 2: Αφαίρεσε το κόμβο που έχει σειρά στην ουρά και εξέτασε τον.

- Αν ο κόμβος προορισμός εντοπιστεί ο αλγόριθμος τερματίζει
- Διαφορετικά προστίθενται στην ουρά όλοι οι γειτονικοί κόμβοι του κόμβου που εξετάζεται και δεν έχουν εξεταστεί στο παρελθόν

Βήμα 3: Αν η ουρά είναι άδεια τότε σημαίνει πως όλοι οι κόμβοι του γράφου έχουν εξεταστεί και ο ζητούμενος κόμβος δεν βρέθηκε. Ο αλγόριθμος τερματίζει και επιστρέφει ότι ο κόμβος δεν βρέθηκε.
Βήμα 4: Επαναλαμβάνεται το Βήμα 2 του αλγορίθμου

4.1.3 Ψευδοκώδικας

```
procedure BFS(G,v) is:
   create a queue Q
   create a set V
   add v to V
   enqueue v onto Q
   while Q is not empty loop
        t = Q.dequeue()
        if t is what we are looking for then
             return t
        end if
        for all edges e in G.adjacentEdges(t)
             u = G.adjacentEdges(t,e)
             if u is not in V then
                  add u to V
                  enqueue u onto V
             end if
        end loop
   end loop
   return  none
end BFS
```

4.1.4 Πολυπλοκότητα

Η πολυπλοκότητα του αλγορίθμου αυτού είναι O(V+E). [CLR09 p.597] Εάν ο γράφος αναπαρίσταται από λίστα γειτνίασης τότε καταλαμβάνει Θ(V+E) [CLR09 p.590] χώρο στην μνήμη, ενώ αν ο γράφος αναπαρίσταται από πίνακα γειτνίασης καταλαμβάνει $\Theta(V^2)$[CLR09 p.591].

4.1.5 Εφαρμογές

Ο αλγόριθμος BFS χρησιμοποιείται ευρέως σε πάρα πολλές και διαφορετικές εφαρμογές. Κάποιες από αυτές τις εφαρμογές αναφέρονται παρακάτω σαν παραδείγματα.

- Εύρεση όλων των κόμβων εντός ενός συνδεδεμένου γράφου
- Αλγόριθμος Cheney [CHE70]
- Εύρεση του συντομότερου μονοπατιού μεταξύ δυο κόμβων σε έναν γράφο όταν το κόστος ενός μονοπατιού υπολογίζεται μόνο από τον αριθμό των ακμών.
- Έλεγχος ενός γράφου αν είναι διμερής.
- Μέθοδος Ford – Fulkerson για τον υπολογισμό της μέγιστης ροής σε ένα δίκτυο ροής

4.2 Αναζήτηση κατά βάθος (Depth-First Search - DFS)

4.2.1 Γενικά

Όπως και ο BFS, έτσι και ο DFS είναι ένας αλγόριθμος αναζήτησης κάποιου κόμβου σε έναν γράφο. Αρχίζοντας από έναν κόμβο-πηγή (root node) διασχίζει κάθε κλαδί μέχρι το τέλος και στην συνέχεια οπισθοδρομεί. Μία εκδοχή του DFS χρησιμοποιήθηκε το 19ο αιώνα από τον Γάλλο μαθηματικό Charles Pierre Tremaux ως στρατηγική για την επίλυση λαβυρίνθων[ES11].

4.2.2 Ψευδοκώδικας

Μια αναδρομική υλοποίηση του αλγορίθμου είναι η παρακάτω:

```
procedure DFS(G,v):
   label v as discovered
   for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
             recursively call DFS(G,w)
```

Χωρίς χρήση αναδρομής είναι απαραίτητη η χρήση στοίβας όπως φαίνεται παρακάτω:

```
procedure DFS-iterative(G,v):
   let S be a stack
   S.push(v)
   while S is not empty
        v ← S.pop()
        if v is not labeled as discovered:
             label v as discovered
             for all edges from v to w in G.adjacentEdges(v) do
                   S.push(w)
```

4.2.3 Πολυπλοκότητα

Για την ανάλυση χρόνου και χώρου ο DFS διαφέρει ανάλογα με το πεδίο εφαρμογής. Στην θεωρητική επιστήμη υπολογιστών, ο DFS για να διασχίσει ολόκληρο τον γράφο κάνει χρόνο O(V+E) [CLR09 p.597], και καταλαμβάνει χώρο O(V).

4.2.4 Εφαρμογές

Ο αλγόριθμος DFS ως δομικό στοιχείο σε διάφορες εφαρμογές μπορεί να χρησιμοποιηθεί τουλάχιστον για τα παρακάτω:

- Εντοπισμός σύνδεσης ανάμεσα σε δύο κόμβους σε έναν γράφο
- Τοπολογική Ταξινόμηση
- Εύρεση γεφυρών σε έναν γράφο
- Εύρεση ισχυρά συνδεδεμένων συνόλων
- Εύρεση λύσεων σε προβλήματα λαβυρίνθων
- Τυχαία δημιουργία λαβυρίνθων

4.3 Αλγόριθμος Dijkstra

4.3.1 Γενικά

Ο αλγόριθμος Dijkstra, ο οποίος δημιουργήθηκε από τον επιστήμονα Edsger Dijkstra το 1956 και εκδόθηκε το 1959 [DT10], είναι ένας αλγόριθμος αναζήτησης γράφου ο οποίος λύνει το πρόβλημα του συντομότερου μονοπατιού μιας πηγής για έναν γράφο με μη αρνητικά κόστη στις ακμές του, δημιουργώντας ένα δέντρο σύντομου μονοπατιού. Αυτός ο αλγόριθμος χρησιμοποιείται συχνά ως υπορουτίνα για την δρομολόγηση σε άλλους αλγόριθμους γράφων.
Για ένα δεδομένο κόμβο-πηγή σε ένα γράφο ο αλγόριθμος βρίσκει το μονοπάτι με το ελάχιστο κόστος μεταξύ αυτού του κόμβου-πηγή και οποιουδήποτε άλλου κόμβου στο γράφο. Μπορεί να χρησιμοποιηθεί, επίσης, για να βρεθούν τα κόστη των συντομότερων μονοπατιών από έναν μονό κόμβο σε ένα μονό κόμβο προορισμού σταματώντας τον αλγόριθμο με το που εντοπιστεί ο κόμβος προορισμού. Για παράδειγμα, εάν οι κόμβοι του γράφου αναπαριστούν πόλεις και τα κόστη των ακμών αναπαριστούν αποστάσεις μεταξύ δύο πόλεων που συνδέονται μέσω δρόμου, ο αλγόριθμος Dijkstra μπορεί να χρησιμοποιηθεί για να βρει τη συντομότερη διαδρομή μεταξύ μιας πόλης και όλων των υπολοίπων πόλεων. Ως αποτέλεσμα, ο αλγόριθμος συντομότερου μονοπατιού χρησιμοποιείται ευρέως σε πρωτόκολλα δρομολόγησης δικτύων όπως IS-IS και OSPF (Open Shortest Path First).
Λεπτομέρειες για τον αλγόριθμο Dijkstra θα αναφερθούν στην συνέχεια της διπλωματικής καθώς ο αλγόριθμος θα υλοποιηθεί μέσω προσομοίωσης και θα αναλυθεί περισσότερο.

4.4 Αλγόριθμος Bellman - Ford

4.4.1 Γενικά

Ο αλγόριθμος Bellman - Ford είναι ένας αλγόριθμος που υπολογίζει τα συντομότερα μονοπάτια από μία μονή κόμβο-πηγή προς όλους τους υπόλοιπους κόμβους σε έναν κατευθυνόμενο γράφο με βάρη [BG00]. Είναι βραδύτερος από τον αλγόριθμο Dijkstra για το ίδιο πρόβλημα αλλά πιο ευέλικτος καθώς είναι ικανός να διαχειριστεί γράφους στους οποίους μερικά από τα βάρη είναι αρνητικοί αριθμοί. Ο αλγόριθμος ονομάστηκε από τους δύο δημιουργούς του, Richard Bellman και Lester Ford Jr. και εκδόθηκε το 1958 και 1956 αντίστοιχα. Παρ' όλα αυτά, ο Edward F. Moore επίσης δημοσίευσε τον ίδιο αλγόριθμο το 1957 και γι' αυτό τον λόγο ο αλγόριθμος μερικές φορές ονομάζεται Bellman-Ford-Moore.
Αρνητικά βάρη μπορούν να βρεθούν σε διάφορες εφαρμογές με γράφους και ως εκ τούτου ο αλγόριθμος είναι αρκετά χρήσιμος [SED02]. Αν ο γράφος περιέχει "αρνητικό κύκλο" (κύκλος στον οποίο το άθροισμα των βαρών των ακμών του έχει αρνητική τιμή) ο οποίος είναι προσβάσιμος από την πηγή, τότε δεν υπάρχει συντομότερο μονοπάτι διότι κάθε μονοπάτι μπορεί να συνεχίσει να γίνεται συντομότερο διασχίζοντας παραπάνω από μία φορές αυτόν τον αρνητικό κύκλο. Σε αυτήν την περίπτωση ο αλγόριθμος Bellman-Ford μπορεί να εντοπίσει αρνητικούς κύκλους και αν αναφέρει την ύπαρξή τους [KT06].

4.4.2 Περιγραφή Αλγορίθμου

Όπως και στον αλγόριθμο Dijkstra, ο αλγόριθμος Bellman-Ford βασίζεται στο αξίωμα της χαλάρωσης (principle of relaxation), στο οποίο μία προσέγγιση της σωστής απόστασης αντικαθίσταται σταδιακά από περισσότερο ακριβείς τιμές μέχρι τελικά να βρεθεί η βέλτιστη λύση. Και στους δύο αλγορίθμους η κατά προσέγγιση απόσταση του κάθε κόμβου είναι πάντα υπερεκτιμημένη σε σχέση με την πραγματική τιμή, και αντικαθίσταται από την ελάχιστη τιμή ενός νέου μονοπατιού κάθε φορά. Η διαφορά μεταξύ των αλγορίθμων αυτών έγκειται στο γεγονός ότι ο αλγόριθμος Dijkstra επιλέγει με άπληστη μέθοδο τον κόμβο με την ελάχιστη απόσταση ο οποίος δεν έχει ακόμα υποστεί επεξεργασία και εφαρμόζει την διαδικασία χαλάρωσης σε όλες τις γειτονικές ακμές. Από την άλλη, ο αλγόριθμος Bellman-Ford απλά εφαρμόζει τη διαδικασία χαλάρωσης σε όλες τις ακμές κάνοντάς το V-1 φορές, όπου V ο αριθμός των κόμβων στο γράφο. Σε κάθε μία από αυτές τις επαναλήψεις ο αριθμός των κόμβων με τις νέες υπολογιζόμενες αποστάσεις μεγαλώνει μέχρι το σημείο όπου θα καλυφθούν όλοι οι κόμβοι οι οποίοι θα έχουν τις σωστές αποστάσεις. Αυτή η μέθοδος επιτρέπει στον αλγόριθμο Bellman-Ford να χρησιμοποιείται σε ευρύτερες εφαρμογές από ότι ο Dijkstra.

Αρχικά, ο αλγόριθμος αρχικοποιεί τις αποστάσεις όλων των κόμβων με την εξής λογική: στον κόμβο-πηγή θεωρεί απόσταση 0 και σε όλους τους υπολοίπους άπειρο (η απόσταση κάθε κόμβου συμβολίζει την απόσταση από τον κόμβο πηγή). Στη συνέχεια, για κάθε ακμή, αν η απόσταση από τον κόμβο προορισμού μπορεί να μικρύνει παίρνοντας αυτή την ακμή, η απόσταση ανανεώνεται στη νέα μικρότερη τιμής της. Για κάθε επανάληψη i όπου όλες οι ακμές ελέγχονται, ο αλγόριθμος βρίσκει τα συντομότερα μονοπάτια με μέγιστο μήκος i ακμές. Εφόσον το μακρύτερο πιθανό μονοπάτι χωρίς κύκλο θα έχει V-1 ακμές, οι ακμές πρέπει να προσπελαστούν V-1 φορές ώστε να επιβεβαιωθεί ότι το συντομότερο μονοπάτι βρέθηκε για όλους τους κόμβους. Πραγματοποιείται μια

τελευταία προσπέλαση σε όλες τις ακμές και αν οποιαδήποτε απόσταση ανανεωθεί τότε σημαίνει ότι έχει βρεθεί ένα μονοπάτι με μήκος V ακμών, κάτι το οποίο δηλώνει ότι μέσα στο γράφο υπάρχει ένας αρνητικός κύκλος.

4.4.3 Ψευδοκώδικας

```
function BellmanFord(list vertices, list edges, vertex source)
   ::weight[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (weight and predecessor) with shortest-path
   // (less cost/weight/metric) information
   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then
          weight[v] := 0
       else
          weight[v] := infinity
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if weight[u] + w < weight[v]:
               weight[v] := weight[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if weight[u] + w < weight[v]:
           error "Graph contains a negative-weight cycle"
   return weight[], predecessor[]
```

4.4.4 Πολυπλοκότητα

Ο αλγόριθμος Bellman-Ford τρέχει σε O(V·E) χρόνο όπου V,E ο αριθμός των κόμβων και των ακμών αντίστοιχα.

*4.5 Αλγόριθμος A**

4.5.1 Γενικά

Στην επιστήμη των υπολογιστών ο αλγόριθμος A* είναι ένας αλγόριθμος ο οποίος χρησιμοποιείται εκτενώς για την εύρεση μονοπατιών και την διάσχιση γράφων. Είναι αξιοσημείωτος για την ακρίβεια και την αποδοτικότητά του. Ωστόσο, σε πρακτικά προβλήματα δρομολόγησης γενικώς υστερεί σε απόδοση σε σχέση με αλγόριθμους οι οποίοι μπορούν να προεπεξεργάζονται τον γράφο για καλύτερη απόδοση [DSS+09], αν και έχει βρεθεί ότι ο A* είναι ανώτερος για άλλες προσεγγίσεις. [ZC09]
Ο Peter Hart, Nils Nilson και Bertram Raphael του Stanford Research Institute (SRI International) περιέγραψαν πρώτοι τον αλγόριθμο το 1968 [HNR68]. Είναι μια επέκταση του Dijkstra αλγορίθμου όπου όμως χρησιμοποιεί ευρετικές, τεχνικές με αποτέλεσμα την καλύτερη απόδοση σε χρόνο.

4.5.2 Περιγραφή Αλγορίθμου

Ο A* χρησιμοποιεί best-first search (BFS) και βρίσκει το μονοπάτι με το ελάχιστο κόστος με δοσμένους τους κόμβους αρχής και τέλους. Καθώς ο A* διασχίζει το γράφο ακολουθεί ένα μονοπάτι με το αναμενόμενο ελάχιστο συνολικό κόστος διατηρώντας μία ταξινομημένη ουρά προτεραιότητας των εναλλακτικών τμημάτων του μονοπατιού.

Χρησιμοποιεί knowledge-plus-heuristic συνάρτηση κόστους του κόμβου x (f(x)) ώστε να καθορίσει τη σειρά με την οποία αναζητά κόμβους στο δέντρο. Η συνάρτηση κόστους είναι το άθροισμα δύο συναρτήσεων:

- η παρελθοντική συνάρτηση κόστους και η μελλοντική συνάρτηση κόστους. Η παρελθοντική συνάρτηση κόστους γνωρίζει τις αποστάσεις των κόμβων από τον αρχικό κόμβο, άρα γνωρίζει και την απόσταση του κόμβου x (g(x)).
- Η μελλοντική συνάρτηση κόστους δίνει μία ευρετική εκτίμηση της απόστασης από τον x μέχρι τον στόχο h(x).

Το h(x), μέρος της συνάρτησης f(x), πρέπει να είναι μία αποδεκτή ευρετική συνάρτηση, δηλαδή δεν πρέπει να υπερεκτιμά την απόσταση προς τον στόχο. Έτσι, σε εφαρμογές, όπως η δρομολόγηση, η h(x) πρέπει να αναπαριστά την απόσταση από το στόχο σε ευθεία γραμμή, αφού η μικρότερη πιθανή απόσταση μεταξύ δύο σημείων είναι η ευθεία. Εάν η ευρετική h(x) ικανοποιεί την επιπλέον συνθήκη h(x) <= d(x,y) + h(y) για κάθε ακμή (x,y) του γράφου (όπου d το κόστος της ακμής), τότε η h λέγεται μονότονη (or consistent). Σε αυτήν την περίπτωση ο A* μπορεί να υλοποιηθεί περισσότερο αποδοτικά και είναι ισοδύναμος με τον αλγόριθμο Dijkstra έχοντας μειωμένα κόστη d'(x,y) = d(x,y) + h(y) - h(x).

4.5.3 Βήματα Αλγορίθμου

Όπως όλοι οι αλγόριθμοι αναζήτησης πληροφορίας στην αρχή ψάχνει για μονοπάτια τα οποία φαίνεται να οδηγούν κατευθείαν στο στόχο. Αυτό που κάνει τον A* να διαφέρει από έναν άπληστο BFS είναι το ότι λαμβάνει υπόψη επίσης την απόσταση που έχει ήδη διανύσει. Η g(x) είναι το κόστος μέχρι τώρα από τον κόμβο πηγή και όχι απλά ένα τοπικό κόστος από προηγούμενους διευρυμένους κόμβους.

Αρχίζοντας από έναν κόμβο αναφοράς, διατηρείται μια ουρά προτεραιότητας των κόμβων που πρέπει να προσπελαστούν, γνωστό και ως ανοιχτό σύνολο (open set, fringe). Όσο πιο μικρή η συνάρτηση f(x) για έναν δεδομένο κόμβο x τόσο μεγαλύτερη είναι η προτεραιότητα. Σε κάθε βήμα του αλγορίθμου ο κόμβος με την μικρότερη τιμή f(x) αφαιρείται από την ουρά, οι τιμές των f και g των γειτόνων ανανεώνονται ανάλογα και αυτοί οι γείτονες προστίθενται στην ουρά. Ο αλγόριθμος συνεχίζει μέχρι ο κόμβος-στόχος να έχει την μικρότερη τιμή f σε σχέση με οποιονδήποτε άλλον κόμβο στην ουρά (ή μέχρι η ουρά να αδειάσει). (Ο κόμβος-στόχος μπορεί να προσπελαστεί πολλαπλές φορές αν υπάρχουν άλλοι κόμβοι με μικρότερη τιμή f, καθώς μέσω αυτών πιθανών να υπάρχει μικρότερο μονοπάτι προς τον στόχο). Η τιμή f του στόχου είναι το μήκος του συντομότερου μονοπατιού καθώς η τιμή h του στόχου είναι πια 0.

Ο αλγόριθμος που περιγράφτηκε παραπάνω δίνει μόνο το μήκος του συντομότερου μονοπατιού. Για να βρεθεί η πραγματική ακολουθία βημάτων από την πηγή μέχρι τον στόχο, ο αλγόριθμος μπορεί εύκολα να επεκταθεί έτσι ώστε σε κάθε κόμβο καθ' όλη την διαδικασία να αποθηκεύεται ο προηγούμενος προς αυτόν κόμβος. Αφού εκτελεστεί ο αλγόριθμος, ο κόμβος στόχος θα δηλώνει ποιος είναι ο προηγούμενος κόμβος και έτσι με οπισθοδρόμηση θα καταλήξουμε στον κόμβο-πηγή. Επιπλέον, αν η ευρετική μέθοδος είναι μονοτονική, όπως αναφέρθηκε παραπάνω, ένα κλειστό σύνολο κόμβων που έχουν ήδη προσπελαστεί μπορεί να χρησιμοποιηθεί ώστε να κάνει την αναζήτηση πιο αποδοτική.

4.5.4 Ψευδοκώδικας

```
function A*(start,goal)

     // The set of nodes already evaluated.
     closedset := the empty set
     // The set of tentative nodes to be evaluated, initially
     // containing the start node
     openset := {start}
     // The map of navigated nodes.
     came_from := the empty map
     // Cost from start along best known path.
     g_score[start] := 0
     // Estimated total cost from start to goal through y.
     f_score[start] := g_score[start] +
     heuristic_cost_estimate(start, goal)
```

```
     while openset is not empty
        current := the node in openset having the lowest f_score[]
value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
          if neighbor in closedset
               continue
          tentative_g_score := g_score[current] +
          dist_between(current,neighbor)

          if neighbor not in openset or tentative_g_score <
          g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
               f_score[neighbor] := g_score[neighbor] +
               heuristic_cost_estimate(neighbor, goal)

               if neighbor not in openset
                    add neighbor to openset

     return failure

function reconstruct_path(came_from, current_node)
    if current_node in came_from
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

4.5.4 Πολυπλοκότητα

Η πολυπλοκότητα χρόνου του αλγορίθμου A* εξαρτάται από την ευρετική συνάρτηση. Στην χειρότερη περίπτωση, ο αριθμός των κόμβων εξελίσσεται εκθετικά κατά την διάρκεια της λύσης, αλλά η εξέλιξη γίνεται πολυωνυμική όταν ο χώρος αναζήτησης είναι δέντρο, όπου υπάρχει μία τελική κατάσταση και η ευρετική συνάρτηση h συναντά την παρακάτω προϋπόθεση:

$|h(x) - h^*(x)| = O(\log h^*(x))$

Όπου $h^*(x)$ είναι η βέλτιστη ευρετική συνάρτηση, δηλαδή το ακριβές κόστος για να φτάσουμε από το x στον στόχο. Με άλλα λόγια, το σφάλμα της h δεν θα μεγαλώνει γρηγορότερα από τον λογάριθμο της βέλτιστης ευρετικής h* ή οποία επιστρέφει την πραγματική απόσταση από τον x προς τον στόχο [RN95]

4.5.5 Εφαρμογές

Ο A* χρησιμοποιείται κατά κόρον για απλά προβλήματα εύρεσης μονοπατιών σε εφαρμογές όπως βιντεοπαιχνίδια, αλλά είχε πρωτοσχεδιαστεί για να χρησιμοποιηθεί ως αλγόριθμος γενικής αναζήτησης σε γράφο [HNR68]. Βρίσκεται, επίσης, σε εφαρμογές ποικίλων προβλημάτων συμπεριλαμβανομένου του προβλήματος της ανάλυσης χρησιμοποιώντας στοχαστικές γραμματικές σε NPL[KM03].

4.6 Αλγόριθμος Floyd - Warshall

4.6.1 Γενικά

Στην επιστήμη των υπολογιστών, ο αλγόριθμος Floyd-Warshall (επίσης γνωστός ως αλγόριθμος του Floyd, Roy Warshall αλγόριθμος, Roy-Floyd αλγόριθμος ή WFI) είναι ένας αλγόριθμος ανάλυσης γράφου για την εύρεση συντομότερων μονοπατιών σε έναν γράφο με θετικά ή αρνητικά βάρη στις ακμές του (αλλά κανέναν αρνητικό κύκλο) και επίσης για την εύρεση μεταβατικής κλειστότητας μιας σχέσης R. Μία απλή εκτέλεση του αλγορίθμου βρίσκει τα μήκη (άθροισμα βαρών) των συντομότερων μονοπατιών μεταξύ όλων των ζευγαριών των κόμβων, δεν επιστρέφει όμως λεπτομέρειες για τα μονοπάτια αυτά καθαυτά.

Ο αλγόριθμος Floyd-Warshall δημοσιεύτηκε το 1962 από τον Robert Floyd. Ωστόσο, είναι ένας αλγόριθμος όμοιος με αλγόριθμους που εκδόθηκαν στο παρελθόν από τους Bernard Roy και Stephen Warshall το 1959,1962 αντίστοιχα για την εύρεση της μεταβατικής κλειστότητας ενός γράφου.
Η σύγχρονη μορφή του αλγόριθμου του Warshall έχει τρεις εμφωλευμένες επαναλήψεις όπως είχε περιγραφεί από τον Peter Ingerman επίσης το 1962. Ο αλγόριθμος είναι ένα παράδειγμα δυναμικού προγραμματισμού.

Ο αλγόριθμος Floyd-Warshall συγκρίνει όλα τα πιθανά μονοπάτια στον γράφο μεταξύ κάθε ζευγαριού κόμβων. Αυτό το κάνει σε $\Theta(V^2)$ συγκρίσεις μέσα στο γράφο. Λαμβάνοντας υπόψη ότι υπάρχουν $\Omega(V^2)$ ακμές στον γράφο και κάθε συνδυασμός ακμών ελέγχεται, ο παραπάνω χρόνος είναι αξιοσημείωτος. Αυτό το πετυχαίνει με την σταδιακή βελτίωση μιας εκτίμησης του συντομότερου μονοπατιού μεταξύ δύο κόμβων μέχρι να βρεθεί η βέλτιστη.

4.6.2 Περιγραφή Αλγορίθμου

Θεωρούμε ένα γράφο G με κόμβους V αριθμημένοι από το 1 έως το N. Θεωρούμε συνάρτηση shortestpath(i,j,k) η οποία επιστρέφει το συντομότερο μονοπάτι μεταξύ του i και του j χρησιμοποιώντας μόνο κόμβους από το σύνολο 1,2,...,k ως ενδιάμεσα σημεία της διαδρομής. Δεδομένης της παραπάνω συνάρτησης ο στόχος είναι να βρεθεί το συντομότερο μονοπάτι μεταξύ του i και του j χρησιμοποιώντας μόνο κόμβους από το 1 έως το k+1.

Για κάθε ένα από αυτά τα ζευγάρια των κόμβων, το πραγματικό συντομότερο μονοπάτι μπορεί να είναι είτε ένα μονοπάτι το οποίο χρησιμοποιεί ενδιάμεσους κόμβους από το σύνολο 1 έως k ή ένα μονοπάτι το οποίο ξεκινά από τον κόμβο i και καταλήγει στον κόμβο j με υποχρεωτικό ενδιάμεσο

κόμβο τον k+1. Είναι γνωστό ότι το συντομότερο μονοπάτι από τον i στον j το οποίο χρησιμοποιεί μόνο κόμβους από τον 1 έως τον k καθορίζεται από την συνάρτηση shortestpath. Άρα εάν υπάρχει συντομότερο μονοπάτι από τον i έως τον j μέσω του k+1 τότε το μήκος αυτού του μονοπατιού θα είναι η συνένωση του συντομότερου μονοπατιού από το i έως το k+1 χρησιμοποιώντας ενδιάμεσους κόμβους από 1 έως k και τους συντομότερου μονοπατιού από k+1 έως j χρησιμοποιώντας ενδιάμεσους κόμβους από 1 έως k. Έστω w(i,j) είναι το βάρος μιας ακμής μεταξύ των κόμβων i,j η συνάρτηση shortestpath(i,j,k+1) μπορεί να οριστεί μέσω της παρακάτω αναδρομικής σχέσης:

shortestpath(i,j,0) = w(i,j)
shortestpath(i,j,k+1) = min(shortestpath(i,j,k), shortestpath(i,k+1,k)+shortestpath(k+1,j,k))

Η παραπάνω φόρμουλα είναι η καρδιά του αλγορίθμου Floyd-Warshall. Ο αλγόριθμος δουλεύει υπολογίζοντας αρχικά το shortestpath(i,j,k) για όλα τα ζευγάρι i,j για k=1,k=2, κ.λ.π. Η διαδικασία αυτή συνεχίζει μέχρι το k=N με αποτέλεσμα να έχουν βρεθεί τα συντομότερα μονοπάτια για όλα τα ζευγάρια i,j χρησιμοποιώντας οποιουσδήποτε ενδιάμεσους κόμβους.

4.6.3 Ψευδοκώδικας

Ο ψευδοκώδικας του αλγορίθμου στην βασική εκδοχή του φαίνεται παρακάτω:

```
let dist be a |V| × |V| array of minimum distances initialized to
∞ (infinity)
for each vertex v
    dist[v][v] ← 0
for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
    for i from 1 to |V|
     for j from 1 to |V|
          if dist[i][j] > dist[i][k] + dist[k][j]
          dist[i][j] ← dist[i][k] + dist[k][j]
```

4.6.4 Πολυπλοκότητα

Έστω n ο αριθμός των κόμβων στον γράφο. Για να βρεθούν όλα τα n^2 των shortestPath(i,j,k) (για όλα τα i και j) από εκείνα των shortestPath(i,j,k-1) απαιτούνται $2n^2$ ενέργειες. Εφόσον ξεκινάμε με την shortestPaths(i,j,0) = edge(i,j) και υπολογίζουμε την ακολουθία των n πινάκων shortestPath(i,j,1), shortestPath(i,j,2), …, shortestPath(i,j,n) ο συνολικός αριθμός των ενεργειών θα είναι $n \cdot 2n^2 = 2n^3$. Επομένως ο πολυπλοκότητα του αλγορίθμου θα είναι $\Theta(n^3)$

4.6.5 Εφαρμογές

Ο αλγόριθμος Floyd-Warshall μπορεί να χρησιμοποιηθεί για να λύσει τα παρακάτω προβλήματα, τα οποία αναφέρονται ενδεικτικά:

- συντομότερα μονοπάτια σε κατευθυνόμενο γράφο
- μεταβατική κλειστότητα κατευθυνόμενων γράφων
- εύρεση κανονικής έκφρασης η οποία υποδηλώνει κανονική γλώσσα αποδεκτή από πεπερασμένο αυτόματο (kleene algorithm) [GY03]
- αντιστροφή πραγματικών πινάκων (gauss algorithm)
- βέλτιστη δρομολόγηση. Σε αυτήν την εφαρμογή απαιτείται η εύρεση ενός μονοπατιού με την μέγιστη ροή μεταξύ δύο κόμβων. Αυτό σημαίνει ότι αντί να λαμβάνονται ελάχιστα, όπως ο αλγόριθμος που περιγράφτηκε παραπάνω, λαμβάνονται μέγιστα. Σε ένα τέτοιο πρόβλημα τα βάρη των γράφων αναπαριστούν ροές μεταξύ κόμβων.
- γρήγορος υπολογισμός δικτύων εύρεσης μονοπατιών
- ευρύτερα μονοπάτια ή μονοπάτια μέγιστου bandwidth
- υπολογισμός κανονικών μορφών των difference bound matrices (DBMs)

4.7 Αλγόριθμος Johnson

4.7.1 Γενικά

Ο αλγόριθμος Johnson βρίσκει τα συντομότερα μονοπάτια μεταξύ όλων των ζευγαριών των κόμβων σε ένα αραιό κατευθυνόμενο γράφο με βάρη στις ακμές του. Στον γράφο αυτό μπορούν να υπάρχουν αρνητικά βάρη αλλά όχι αρνητικοί κύκλοι. Ο αλγόριθμος δουλεύει χρησιμοποιώντας τον αλγόριθμο Bellman-Ford ώστε να υπολογίσει έναν μετασχηματισμό του γράφου ο οποίος αφαιρεί όλα τα αρνητικά βάρη επιτρέποντας να εφαρμοσθεί ο αλγόριθμος Dijkstra στον μετασχηματιζόμενο γράφο [CLR+01]. Ονομάστηκε από τον Donald B. Johnson ο οποίος τον δημοσίευσε πρώτος τον 1977 [JON77].
Μία παρόμοια τεχνική μετασχηματισμού χρησιμοποιείται επίσης στον αλγόριθμο Suurballe ώστε να βρεθούν δύο ασύνδετα μονοπάτια με το ελάχιστο άθροισμα μηκών μεταξύ δύο ίδιων κόμβων σε ένα γράφο με μη αρνητικά βάρη [SUU74].

4.7.2 Περιγραφή Αλγορίθμου

Ο αλγόριθμος Johnson ακολουθεί τα παρακάτω βήματα:

Βήμα 1: Αρχικά ένας νέος κόμβος q προστίθεται στον γράφο και συνδέεται με όλους τους υπόλοιπους κόμβους με ακμές βάρους 0.
Βήμα 2: Στη συνέχεια, εφαρμόζεται ο αλγόριθμος Bellman-Ford, με αρχή τον νέο κόμβο q, ώστε να βρεθεί για κάθε κόμβο v το ελάχιστο βάρος h(v) ενός μονοπατιού από το q μέχρι το v. Εάν σε αυτό το βήμα εντοπιστεί αρνητικός κύκλος, ο αλγόριθμος τερματίζεται.
Βήμα 3: Στη συνέχεια, οι ακμές του γράφου ανανεώνονται χρησιμοποιώντας τις τιμές που υπολογίστηκαν από τον αλγόριθμο Bellman-Ford: μία ακμή από το u στο v η οποία έχει μήκος w(u,v) ανανεώνει το μήκος της σε w(u,v)+h(u)-h(v).

Βήμα 4: Τέλος, ο κόμβος q αφαιρείται και εφαρμόζεται στο νέο γράφο ο αλγόριθμος Dijkstra για κάθε κόμβο s προς όλους τους άλλους κόμβους.

4.7.3 Πολυπλοκότητα

Η πολυπλοκότητα χρόνου του αλγορίθμου χρησιμοποιώντας σωρούς Fibonacci είναι $O(V^2 \log V + V \cdot E)$: Ο αλγόριθμος τρέχει σε $O(VE)$ χρόνο για το στάδιο Bellman-Ford και σε χρόνο $O(V \log V + E)$ για κάθε μία από τις V επαναλήψεις του αλγόριθμου Dijkstra. Επομένως, όταν ο γράφος είναι αραιός, ο συνολικός χρόνος εκτέλεσης είναι μικρότερος από τον χρόνο εκτέλεσης του αλγορίθμου Floyd-Warshall ο οποίος λύνει το ίδιο πρόβλημα σε χρόνο $O(V^3)$ [CLR+01].

4.8 Αλγόριθμος Viterbi

4.8.1 Γενικά

Ο αλγόριθμος Viterbi είναι ένας αλγόριθμος δυναμικού προγραμματισμού που βρίσκει την πιο πιθανή ακολουθία από κρυφά στάδια ονομαζόμενη ως μονοπάτι Viterbi και δίνει ως αποτέλεσμα μια ακολουθία από παρατηρούμενα γεγονότα στο πλαίσιο των Μαρκοβιανών πηγών πληροφορίας (Markov information sources) και των κρυφών Μαρκοβιανών μοντέλων (hidden Markov models). Ο αλγόριθμος εφαρμόζεται παγκοσμίως για την αποκωδικοποίηση συνελλικτικών κωδικών που χρησιμοποιούνται σε CDMA και GSM ψηφιακών κυψελών, dial-up modems, δορυφόρους, deep-space επικοινωνίες και σε ασύρματα 802.11 LANs. Χρησιμοποιείται επίσης σε εφαρμογές αναγνώρισης φωνής, σύνθεση φωνής, diarization [XAV10], εντοπισμού λέξης-κλειδί, υπολογιστικές γλώσσες και βιοπληροφορική. Για παράδειγμα, σε μία εφαρμογή αναγνώρισης φωνής το ακουστικό σήμα θεωρείται ως μία παρατηρούμενη ακολουθία από γεγονότα και μία συμβολοσειρά θεωρείται ως ο "κρυφός στόχος" του ακουστικού σήματος. Ο αλγόριθμος Viterbi βρίσκει την πιο πιθανή συμβολοσειρά δεδομένου αυτού του ακουστικού σήματος.

4.8.2 Περιγραφή Αλγορίθμου

Έστω ένα δοσμένο κρυφό Μαρκοβιανό μοντέλο με χώρο καταστάσεων S, εργοδικές πιθανότητες Pi (η πιθανότητα να βρισκόμαστε στο στάδιο i) και πιθανότητες μετάβασης Ai,j (η πιθανότητα να μεταβούμε από την κατάσταση i στην κατάσταση j). Έστω ότι παρατηρούμε εξόδους $y_1,...,y_T$. Η πιο πιθανή ακολουθία καταστάσεων $x_1,..,x_T$ η οποία δημιουργεί τις παραπάνω παρατηρήσεις δίνεται από τις παρακάτω σχέσεις:

$$V_{1,k} = P(y_1|k) \cdot \pi_k$$

$$V_{t,k} = P(y_t|k) \cdot \max_{x \in S}(a_{x,k} * V_{t-1,x})$$

Όπου $V_{t,k}$ είναι η πιθανότητα της περισσότερο πιθανής κατάστασης για τις πρώτες t επαναλήψεις που έχουν το k ως τελευταία κατάσταση. Το μονοπάτι Viterbi μπορεί να εξαχθεί μέσω της αποθήκευσης των δεικτών οι οποίοι θυμούνται ποια κατάσταση x χρησιμοποιήθηκε στην δεύτερη εξίσωση.

Έστω Ptr(k,t) μια συνάρτηση η οποία επιστρέφει την τιμή του x που χρησιμοποιήθηκε για να υπολογιστεί η $V_{t,k}$ αν t > 1 ή t > k. Αν t = 1 τότε:

$x_T = \arg\max_{x \in S}(V_{T,x})$

$x_{t-1} = Ptr(x_t,t)$

όπου χρησιμοποιείται ο βασικός ορισμός του arg max f(x) = { x | για κάθε y : f(y) ≤ f(x) }

4.8.3 Πολυπλοκότητα

Η πολυπλοκότητα του αλγορίθμου είναι $O(T \cdot |S|^2)$

5

Εκτενέστερη Μελέτη του Αλγορίθμου **Dijkstra**

5.1 Περιγραφή Αλγορίθμου

Σημείωση: Για την ευκολία κατανόησης του αλγορίθμου, στην παρακάτω περιγραφή θα χρησιμοποιηθούν οι όροι διασταύρωση, δρόμος και χάρτης αντί των όρων κόμβος, ακμή και γράφος αντίστοιχα ώστε να συσχετιστεί ο αλγόριθμος με μια πρακτική εφαρμογή.

Ζητείται να βρεθεί η συντομότερη διαδρομή ανάμεσα σε δύο διασταυρώσεις πάνω σε ένα χάρτη μια πόλης, δηλαδή έχουν οριστεί ένα σημείο αφετηρίας και ένα σημείο προορισμού. Αρχικά, μαρκάρουμε την απόσταση κάθε διασταύρωσης στο χάρτη από την αρχή στο άπειρο. Αυτό συμβαίνει όχι για να θεωρηθεί ότι η απόσταση είναι όντως άπειρη, αλλά για να σημειωθεί ότι τη διασταύρωση αυτή δεν την έχουμε επισκεφτεί ακόμα. Σε μερικές άλλες εφαρμογές του αλγορίθμου ανάλογα με το σύστημα που εκτελείται, οι διασταυρώσεις αυτές μπορούν να θεωρηθούν χωρίς ετικέτα (unlabeled). Τώρα, σε κάθε επανάληψη επιλέγουμε την τρέχουσα διασταύρωση. Για την πρώτη επανάληψη η τρέχουσα διασταύρωση είναι το αρχικό σημείο και η απόστασή της είναι μηδέν. Για επόμενες επαναλήψεις, η τρέχουσα διασταύρωση θα είναι η διασταύρωση την οποία δεν έχουμε επισκεφτεί και έχει την μικρότερη απόσταση από το αρχικό σημείο σε σχέση με όλες τις αποστάσεις των διασταυρώσεων που δεν έχουμε επισκεφτεί.

Για την τρέχουσα διασταύρωση θα ενημερώνουμε την απόσταση κάθε γειτονικής διασταύρωσης που δεν έχουμε επισκεφτεί, δηλαδή μιας διασταύρωσης η οποία συνδέεται άμεσα με την τρέχουσα. Αυτό συμβαίνει για να καθοριστεί η απόσταση των γειτονικών διασταυρώσεων που δεν έχουμε ακόμα επισκεφτεί. Με αυτόν τον τρόπο εάν η νέα υπολογιζόμενη απόσταση μιας διασταύρωσης από το αρχικό σημείο βρεθεί μικρότερη από μία απόσταση η οποία είχε υπολογιστεί σε προηγούμενη επανάληψη την αντικαθιστά. Επίσης, όταν μία απόσταση μίας διασταύρωσης αντικαθίσταται με μία νέα θα σημειώνεται και μία επιπλέον πληροφορία για την διασταύρωση αυτή η οποία θα είναι η τρέχουσα διασταύρωση. Αυτό γιατί όταν θα προσπαθήσουμε να διασχίσουμε το

συντομότερο μονοπάτι θέλουμε να γνωρίζουμε για μία διασταύρωση ποια είναι η προηγούμενη από αυτήν. Με άλλα λόγια, αν βρεθεί συντομότερο μονοπάτι για ένα σημείο αντικαθιστά το προηγούμενο μονοπάτι που είχε σημειωθεί. Αφού ενημερωθούν όλες οι αποστάσεις των γειτονικών διασταυρώσεων της τρέχουσας διασταύρωσης σημειώνουμε την τρέχουσα διασταύρωση ότι την έχουμε επισκεφτεί και συνεχίζουμε ορίζοντας την επόμενη τρέχουσα διασταύρωση. Η νέα τρέχουσα διασταύρωση θα είναι μία διασταύρωση την οποία δεν έχουμε επισκεφτεί ακόμα και έχει την μικρότερη σημειωμένη απόσταση από το αρχικό σημείο.

Η διαδικασία συνεχίζεται ενημερώνοντας κάθε φορά τις αποστάσεις των γειτόνων της τρέχουσας διασταύρωσης μέχρι το σημείο όπου θα σημειώσουμε ότι έχουμε επισκεφτεί τη διασταύρωση η οποία είναι το σημείο προορισμού. Από αυτό το σημείο και μετά μπορούμε να αναγνωρίσουμε το συντομότερο μονοπάτι από το αρχικό σημείο προς το τελικό με οπισθοδρόμηση. Δηλαδή, ξεκινώντας από τον προορισμό κοιτάμε ποια είναι η προηγούμενη σημειωμένη σε αυτόν διασταύρωση. Μετά κοιτάμε για αυτήν την διασταύρωση την προηγούμενή της κ.ο.κ. μέχρι να φτάσουμε στο αρχικό σημείο.

Είναι αξιοσημείωτο το γεγονός ότι ο αλγόριθμος δεν κατευθύνεται από το αρχικό σημείο προς το τελικό όπως θα περίμενε κανείς. Αντιθέτως, το χαρακτηριστικό εκείνο το οποίο καθορίζει την επόμενη τρέχουσα διασταύρωση είναι η απόστασή της από το αρχικό σημείο. Αυτό σημαίνει ότι ο αλγόριθμος εκτείνεται "γύρω" από το αρχικό σημείο, μέχρι να εντοπίσει το τελικό. Με άλλα λόγια, μεγαλώνοντας την ακτίνα ενός κύκλου με κέντρο το αρχικό σημείο περιμένουμε ο κύκλος να τμήσει το σημείο προορισμού. Βλέποντας τον αλγόριθμο από αυτήν την οπτική είναι ξεκάθαρο ότι ο αλγόριθμος βρίσκει πράγματι το συντομότερο μονοπάτι ανάμεσα σε δύο σημεία. Όμως, αποκαλύπτεται μία από τις αδυναμίες του αλγορίθμου η οποία είναι η σχετική βραδύτητά του για ορισμένες τοπολογίες.

5.2 Βήματα Αλγορίθμου

Αρχικά ορίζουμε έναν κόμβο ως τον κόμβο αρχή ή αλλιώς κόμβο-πηγή από όπου ο αλγόριθμος θα ξεκινάει. Θεωρούμε ως απόσταση ενός κόμβου Α ως την απόσταση από τον αρχικό κόμβο. Ο αλγόριθμος Dijkstra θα αρχικοποιήσει αυτές τις αποστάσεις και στη συνέχεια σε βήματα θα τις ενημερώνει.

Βήμα 1: Θέτουμε την απόσταση του κάθε κόμβου ίση με άπειρο και την απόσταση του κόμβου-πηγή μηδέν.
Βήμα 2: Δημιουργούμε ένα σύνολο το οποίο θα περιέχει τους κόμβους τους οποίους ο αλγόριθμος δεν έχει ακόμα επισκεφτεί. Στο βήμα αυτό δεν έχουμε επισκεφτεί κανέναν κόμβο άρα το σύνολο αυτό θα περιέχει όλους τους κόμβους του γράφου. Θέτουμε τον αρχικό κόμβο ως τρέχον κόμβο.
Βήμα 3: Για τον τρέχον κόμβο, βλέπουμε όλους τους γείτονες του που ο αλγόριθμος δεν έχει ακόμα επισκεφτεί και υπολογίζουμε τις αποστάσεις. Συγκρίνουμε την καινούρια απόσταση με την προηγούμενη και αν είναι μικρότερη αντικαθιστούμε την παλιά με την νέα. Για παράδειγμα, αν σε έναν τρέχον κόμβο Α έχει σημειωθεί ότι η απόστασή του είναι 6 και η ακμή που συνδέει αυτόν με έναν γείτονά του Β έχει μήκος 2 τότε η απόσταση του Β μέσω του Α θα είναι 6+2=8. Εάν ο Β πριν τον υπολογισμό αυτόν είχε απόσταση μεγαλύτερη του 8 τότε η απόσταση του Β αντικαθίσταται και γίνεται 8, αλλιώς διατηρεί την παλιά της τιμή.

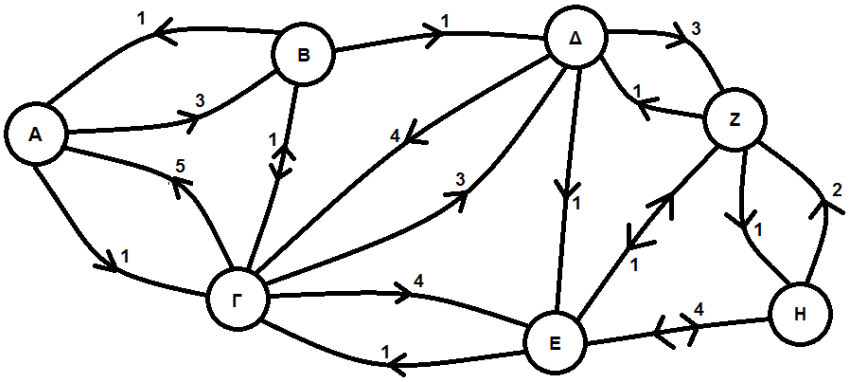
Βήμα 4: Όταν για όλους τους γείτονες του τρέχον κόμβου έχει ενημερωθεί η απόστασή τους, σημειώνουμε τον τρέχον κόμβο ότι ο αλγόριθμο τον έχει επισκεφτεί και τον αφαιρούμε από το σύνολο των κόμβων που δεν έχουμε επισκεφτεί. Έτσι, ένας κόμβος που ο αλγόριθμος έχει επισκεφτεί δεν θα προσπελαστεί ξανά.
Βήμα 5: Ο αλγόριθμος τερματίζει για τις δύο παρακάτω περιπτώσεις:
Περίπτωση 1: αν ο αλγόριθμος εκτελείται για την εύρεση της συντομότερης διαδρομής ανάμεσα σε δύο συγκεκριμένους κόμβους, τερματίζει όταν ο κόμβος προορισμού σημειωθεί ότι τον έχουμε επισκεφτεί.
Περίπτωση 2: αν ο αλγόριθμος εκτελείται για την εύρεση των αποστάσεων όλων των κόμβων του γράφου από τον κόμβο πηγή, τότε ο αλγόριθμος τερματίζει όταν η μικρότερη απόσταση ανάμεσα στους κόμβους του συνόλου των κόμβων που δεν έχουμε επισκεφτεί είναι άπειρη. Αυτό σημαίνει ότι δεν υπάρχει σύνδεση ανάμεσα στον κόμβο πηγή και στους εναπομείναντες κόμβους.
Βήμα 6: Διαλέγουμε από το σύνολο των κόμβων που δεν έχουμε επισκεφτεί τον κόμβο με την μικρότερη απόσταση και τον θέτουμε ως τρέχον κόμβο. Στη συνέχεια, επιστρέφουμε στο βήμα 3.

5.3 Πολυπλοκότητα

Ο πρωτότυπος αλγόριθμος Dijkstra εκτελείται σε $O(V^2)$ (όπου V είναι ο αριθμός των κόμβων). Η ιδέα για αυτόν τον αλγόριθμο δόθηκε επίσης στο (*Leyzorek et al, 1957*). Η υλοποίηση βασίζεται στην ουρά προτεραιότητας υλοποιημένη με Fibonacci Heap και τρέχει σε χρόνο $O(E + V\log V)$ (όπου E ο αριθμός των ακμών) (*Fredman & Tarjan, 1984*). Αυτός είναι ασυμπτωτικά ο γρηγορότερος γνωστός αλγόριθμος εύρεσης συντομότερου μονοπατιού μονής πηγής για αφηρημένους κατευθυνόμενους γράφους με μη αρνητικά βάρη.

5.4 Παράδειγμα Εκτέλεσης

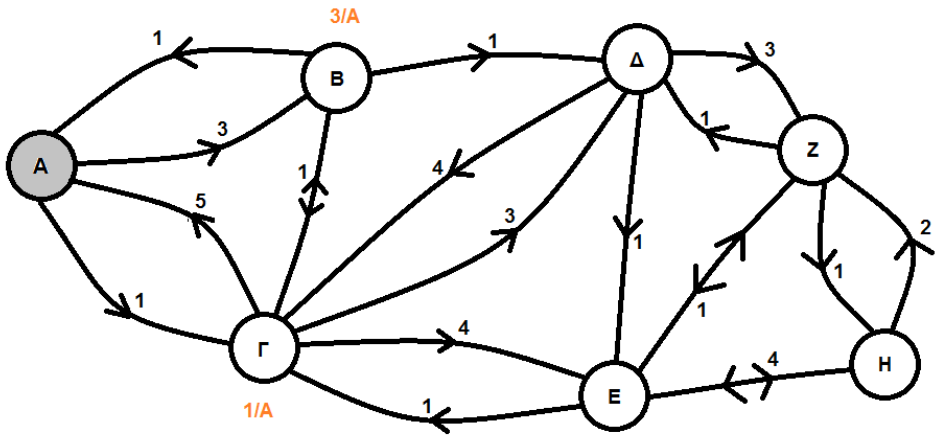
Θα θεωρήσουμε τον παρακάτω κατευθυνόμενο γράφο ως παράδειγμα για την εκτέλεση του αλγορίθμου Dijkstra και θα περιγράψουμε τα βήματα που εξηγήθηκαν παραπάνω μέσα από αυτόν.



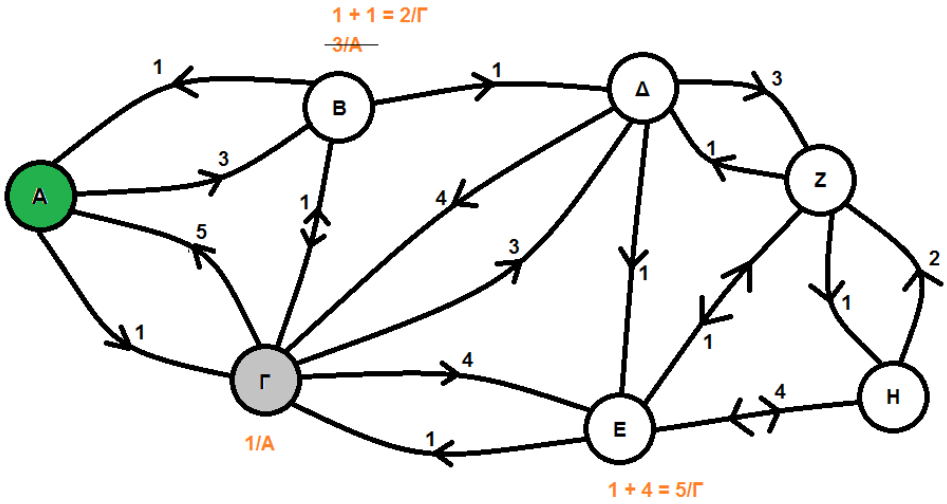
Ζητείται η εύρεση του συντομότερου μονοπατιού στον παραπάνω γράφο με αρχή τον κόμβο Α και τέλος τον κόμβο Η.

Ξεκινώντας από τον κόμβο Α θα κοιτάξουμε τους γειτονικούς κόμβους του. Ως γειτονικός κόμβος σε έναν κατευθυνόμενο γράφο ενός κόμβου Κ, θεωρείται ο κόμβος που συνδέεται άμεσα με μια ακμή που επιτρέπει την μετάβαση από τον κόμβο Κ στον γειτονικό. Εάν η ακμή που συνδέει τον κόμβο Κ με έναν άλλο κόμβο Ζ επιτρέπει μόνο την μετάβαση από τον Ζ στον Κ τότε ο Κ είναι γειτονικός του Ζ αλλά ο Ζ δεν είναι γειτονικός του Κ.

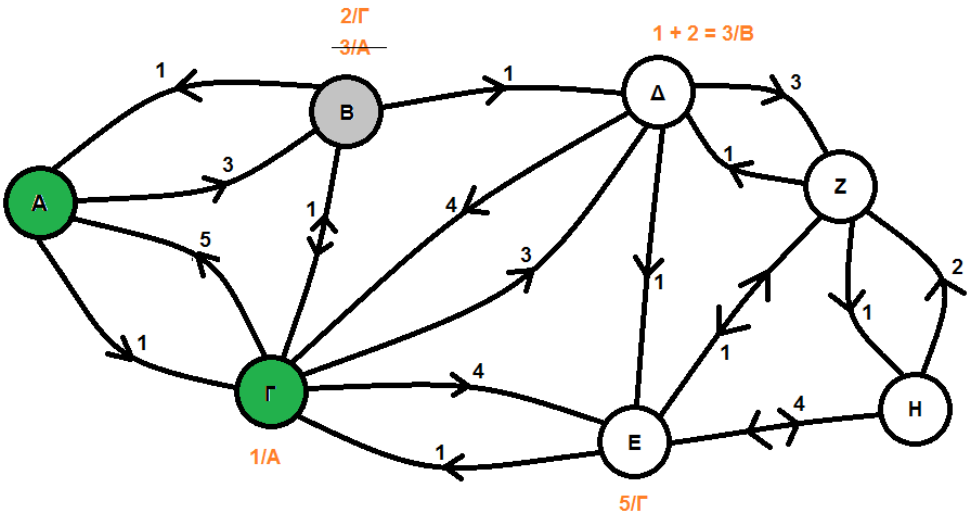
Παρακάτω φαίνεται πώς ο κόμβος Α συνδέεται με τον Β με κόστος 3 και με τον Γ με κόστος 1.
Τα παραπάνω δεδομένα σημειώνονται πάνω στους γειτονικούς κόμβους του Α.



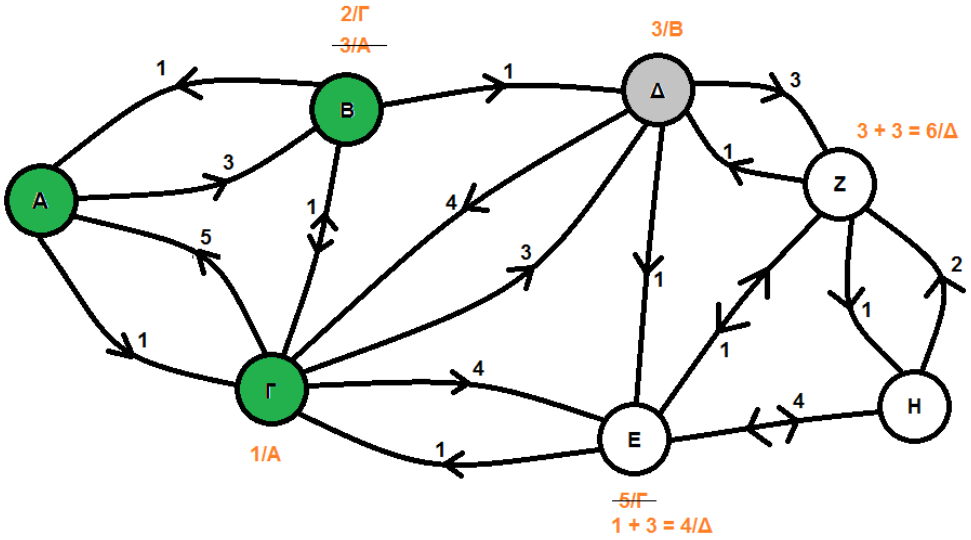
Επίσης σε κάθε γειτονικό κόμβο σημειώνεται ότι προς το παρόν για να φτάσουμε σ' αυτόν ο προηγούμενος κόμβος στο μονοπάτι θα είναι ο Α.
Στο επόμενο βήμα εφόσον εξετάσαμε όλους τους γείτονες του Α θα σημειώσουμε τον Α ως επεξεργασμένο κόμβο (visited) έτσι ώστε να μην χρειαστεί να τον επισκεφθούμε ξανά στο μέλλον.
Ο επόμενος κόμβος για τον οποίο θα εξετασθούν οι γείτονες του θα είναι ο κόμβος με την μικρότερο απόσταση από τον κόμβο Α. Υπενθυμίζουμε πώς για όλους τους κόμβους που δεν έχουμε επισκεφθεί ακόμα η απόσταση από τον Α θεωρείται άπειρη. Εάν κρατήσουμε ένα διάνυσμα με στοιχεία τους κόμβους και την απόσταση τους από τον Α (κόμβο – πηγή) προς το παρόν θα έχουμε το εξής:
`[(A):0/A, B:3/A, Γ:1/A, Δ:άπειρο, E:άπειρο, Z:άπειρο, H:άπειρο]`
Επομένως ο επόμενος κόμβος θα είναι ο Γ αφού έχει την μικρότερη απόσταση από τον Α δηλαδή 1.
Ο Α θεωρείται επεξεργασμένος κόμβος και δεν λαμβάνεται υπόψη στην σύγκριση της απόστασης.



Όπως φαίνεται και στο παραπάνω σχήμα οι μη επεξεργασμένοι γείτονες του Γ είναι ο Β και ο Ε. Τον Ε αφού τον επισκεπτόμαστε πρώτη φορά σημειώνουμε την απόσταση του από τον Α ως το άθροισμα της απόστασης από τον Γ στον Ε συν της απόστασης του Γ από τον Α. Επίσης σημειώνουμε πώς προς το παρόν στον Ε πηγαίνουμε με προηγούμενο κόμβο τον Γ.
Για τον κόμβο Β αφού έχουμε σημειώσει στο παρελθόν κάποια απόσταση (εδώ 3) συγκρίνουμε την νέα απόσταση με την παλιά. Η νέα απόσταση είναι το άθροισμα της απόστασης από τον Γ στον Β συν την απόσταση του Γ από τον Α δηλαδή 1 + 1 = 2. Εφόσον 2 < 3 αντικαθιστούμε την παλιά σημειωμένη απόσταση με την νέα διαφορετικά θα αφήναμε την παλιά όπως έχει. Επίσης σημειώνουμε ότι προς το παρόν για να φτάσουμε στο κόμβο Β ο προηγούμενος κόμβος θα είναι ο Γ αφού αν ήταν ο Α (όπως είχαμε σημειώσει προηγουμένως θα διανύσουμε μεγαλύτερη απόσταση).
Αφού σημειώσουμε τον Γ ως επεξεργασμένο κόμβο το διάνυσμα αποστάσεων θα είναι τώρα:
`[(Α):0/Α, Β:2/Γ, (Γ):1/Α, Δ:άπειρο, Ε:5/Γ, Ζ:άπειρο, Η:άπειρο]`
Άρα ο επόμενος κόμβος θα είναι ο Β ως ο μη επεξεργασμένος κόμβος με την μικρότερη απόσταση από τον Α.

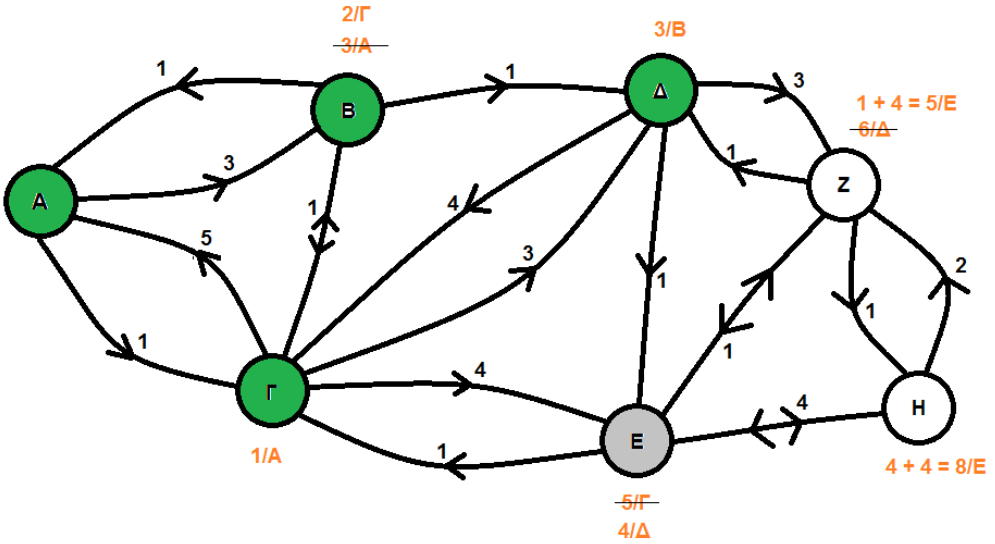


Ο μοναδικός γείτονας του Β είναι ο Δ. Σημειώνουμε την απόσταση του από τον Α ως το άθροισμα της απόστασης από τον Β στον Δ συν την απόσταση του Β από τον Α.
Σημειώνουμε τον Β ως επεξεργασμένο και έτσι το διάνυσμα αποστάσεων θα γίνει:
`[(Α):0, (Β):2/Γ, (Γ):1/Α, Δ:3/Β, Ε:5/Δ, Ζ:άπειρο, Η:άπειρο]`
Άρα ο επόμενος κόμβος θα είναι ο Δ.

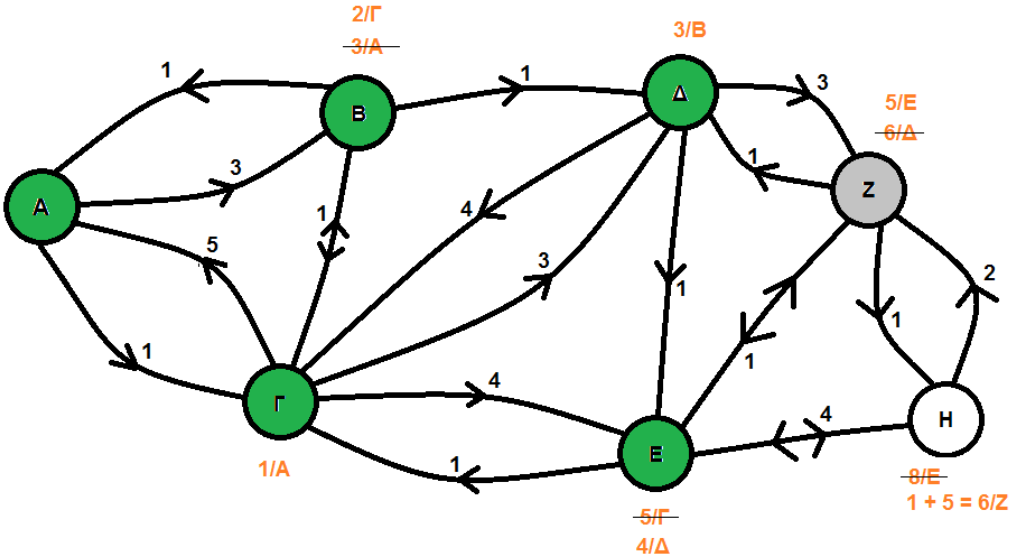


Οι γείτονες του Δ είναι ο Ζ και ο Ε. Καθώς τον Ζ τον επισκεπτόμαστε πρώτη φορά σημειώνουμε την απόσταση του από τον Α κατά τα γνωστά. Για τον Ε συγκρίνουμε την νέα απόσταση με την προηγούμενη του. Εφόσον η νέα είναι μικρότερη αντικαθιστά την προηγούμενη και σημειώνουμε πώς για να πάμε στον Ε ξεκινώντας από τον Α ο προηγούμενος κόμβος θα είναι ο Δ και όχι ο Γ.

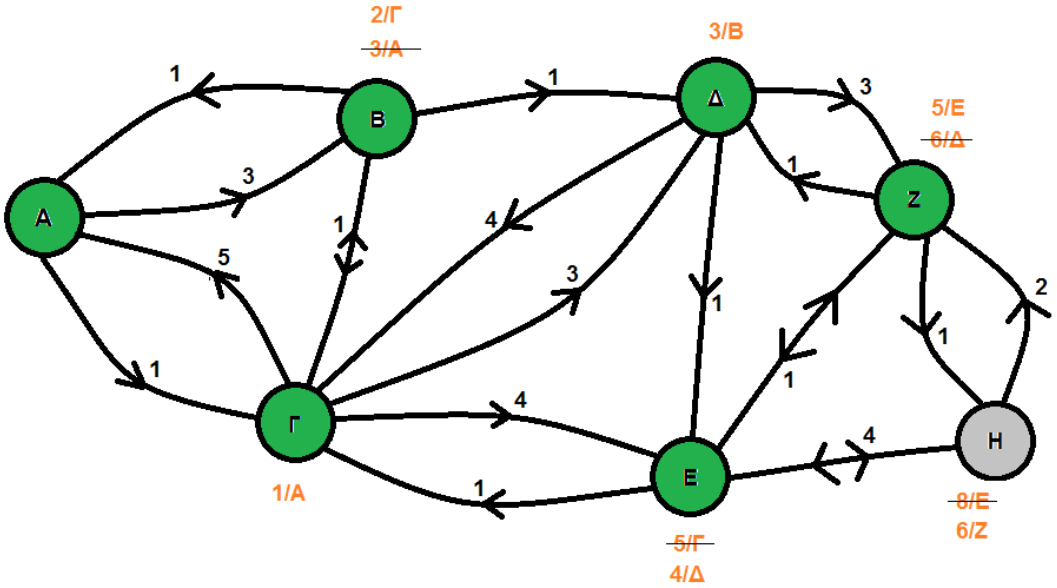
Το διάνυσμα αποστάσεων θα είναι τώρα:
`[(Α):0/Α, (Β):2/Γ, (Γ):1/Α, (Δ):3/Β, Ε:4/Δ, Ζ:6/Δ, Η:άπειρο]`
Επόμενος κόμβος θα είναι ο Ε.



Ενημερώνουμε την απόσταση του Z αφού τυχαίνει να είναι μικρότερη μέσω του E από ότι μέσω του Δ και σημειώνουμε και την απόσταση του H.
Προσοχή, ο αλγόριθμος δεν τερματίζει αφού επισκεφθήκαμε τον τελικό κόμβο H και βρήκαμε μια απόσταση του από τον A. Πρέπει στο διάνυσμα απόστασης να έχει και την μικρότερη απόσταση σε σχέση με τους υπόλοιπους μη επεξεργασμένους κόμβους.
Το διάνυσμα απόστασης τώρα είναι:
`[(A):0/A, (B):2/Γ, (Γ):1/A, (Δ):3/B, (E):4/Δ, Z:5/E, H:8/E]`
Άρα ο επόμενος κόμβος θα είναι ο Z.



Ενημερώνουμε την απόσταση του μοναδικού γείτονα του κόμβου Z.
Το διάνυσμα απόστασης θα είναι τώρα:
`[(A):0/A, (B):2/Γ, (Γ):1/A, (Δ):3/B, (E):4/Δ, Z:5/E, H:6/Z]`
Άρα ο επόμενος κόμβος θα είναι ο H και αφού είναι ο τελικός κόμβος ο αλγόριθμος τερματίζει.



Το συντομότερο μονοπάτι από τον Α στον Η βρίσκεται ως εξής:

Ξεκινάμε από τον Η και βλέπουμε ποιος είναι ο προηγούμενος σημειωμένος κόμβος. Εδώ είναι ο Ζ. Πηγαίνουμε στο Ζ και βλέπουμε ποιος είναι ο προηγούμενος από αυτόν σημειωμένος κόμβος. Εδώ είναι ο Ε. Ακολουθώντας την λογική αυτή της οπισθοδρόμησης κάποια στιγμή θα εντοπίσουμε τον κόμβο Α και θα έχουμε σημειώσει το μονοπάτι που ζητείται.
Στο παράδειγμα θα είναι: `Α -> Γ -> Β -> Δ -> Ε -> Ζ -> Η`

5.5 Απόδειξη Ορθότητας Αλγορίθμου Dijkstra

Ο αλγόριθμος του Dijkstra είναι ένας άπληστος (greedy) αλγόριθμος, δηλαδή σε κάθε βήμα υπολογίζει την τοπικά βέλτιστη λύση. Στο τέλος, συνθέτει τις τοπικές λύσεις και επιστρέφει μια συνολική. Όμως, το γεγονός ότι οι τοπικές λύσεις είναι βέλτιστες δεν εγγυάται λογικά ότι και η σύνθεσή τους θα είναι μια συνολικά βέλτιστη λύση. Για το λόγο αυτόν είναι απαραίτητη μια απόδειξη ορθότητας του αλγορίθμου.

Αρχικά ορίζεται το σύνολο S, το οποίο περιέχει τους κόμβους του γράφου που αλγόριθμος έχει ήδη προσπελάσει. Αυτό σημαίνει ότι όσο ο αλγόριθμος εκτελείται το σύνολο S θα μεγαλώνει. Ο αλγόριθμος μόλις προσπελάσει έναν κόμβο u θα γνωρίζει το μονοπάτι από τον αρχικό κόμβο s προς τον κόμβο u.

Ορίζουμε διαδρομή $P_{i,j}$ ως μια διαδρομή (μονοπάτι) του γράφου που ξεκινά από τον κόμβο i και καταλήγει στον κόμβο j. Ως $|P_{i,j}|$ ορίζεται ο αριθμός των κόμβων που αποτελούν το μονοπάτι. Ορίζουμε την απόσταση R_{ij} ως το κόστος της ακμής που ενώνει του κόμβους i και j.

Έστω $P_{s,v}$ η διαδρομή (μονοπάτι) που έχει σημειώσει ο αλγόριθμος από τον αρχικό κόμβο s στον κόμβο v. Θα δείξουμε ότι για κάθε κόμβο v του συνόλου S, σε κάθε στάδιο της εκτέλεσης του αλγόριθμου, η $P_{s.v}$ είναι η ελάχιστη διαδρομή. Αυτό μας δείχνει άμεσα ότι ο αλγόριθμος επιστρέφει τη βέλτιστη λύση, αφού στο τελευταίο βήμα το σύνολο S θα περιέχει τον κόμβο προορισμού t και επομένως θα γνωρίζει την διαδρομή $P_{s,t}$ η οποία θα είναι και η ελάχιστη. Η απόδειξη είναι επαγωγική ως προς το μέγεθος του συνόλου S (των κόμβων που έχουν μέχρι εκείνο το βήμα προσπελαστεί).

Για |S|=1, έχουμε ότι S={s} και το κόστος της διαδρομής $P_{s,s}$ είναι 0. Αφού δεν υπάρχουν αρνητικά βάρη, αυτή είναι όντως η συντομότερη διαδρομή. Έστω ότι για |S|=k, με k>1, ο ισχυρισμός ισχύει. Μένει να δείξουμε ότι ο ισχυρισμός ισχύει και για |S|=k+1, έχοντας αυξήσει το σύνολο των επεξεργασμένων κόμβων κατά 1, με την προσθήκη ενός νέου κόμβου, έστω του v. Έστω ότι ο κόμβος u είναι ο αμέσως προηγούμενος του v στη διαδρομή $P_{s,v}$ που έχει σημειώσει ο αλγόριθμος. Λόγω της υπόθεσης της επαγωγής, η διαδρομή $P_{s,u}$ είναι η συντομότερη από τον αρχικό κόμβο στον u.

Έστω ότι στο σύνολο S οι κόμβοι x_i είναι οι μόνοι κόμβοι όπου έχουν γειτονικούς κόμβους εκτός του συνόλου S. Προφανώς ο κόμβος u θα είναι ένας από τους κόμβους x_i. Τώρα έστω ένα σύνολο Z με κόμβους y_j που είναι εκτός του συνόλου S και είναι γειτονικοί με τουλάχιστον έναν κόμβο x_i του S. Κατά την επανάληψη k+1 του αλγορίθμου, εξετάζονται όλες οι αποστάσεις $R_{xi,yj}$ για κάθε i,j με i e [1,|S|] και j e [1,|Z|]. Ο αλγόριθμος εντοπίζει την μικρότερη $R_{xi,yj}$ (έστω οι κόμβοι u και v αντίστοιχα) και προσθέτει τον κόμβο v στο σύνολο S. Στην συνέχεια αποθηκεύει το μονοπάτι $P_{s,v}$ ως την ένωση $P_{s,u}$ και $P_{u,v}$ με $|P_{u,v}| = 2$. Επομένως αφού η διαδρομή $P_{s,u}$ είναι η συντομότερη από την υπόθεση και η $P_{u,v}$ είναι η συντομότερη σε σχέση με όλες τις άλλες $P'_{u,v}$ αφού ο αλγόριθμος την εντόπισε τοπικά σημαίνει ότι η $P_{s,v}$ είναι η συντομότερη.

6

Αλγόριθμος Δρομολόγησης Κ Συντομότερων Μονοπατιών

6.1 Γενικά

Ο αλγόριθμος δρομολόγησης Κ συντομότερων μονοπατιών είναι μια επέκταση των αλγορίθμων δρομολόγησης συντομότερου μονοπατιού ανάμεσα σε δύο σημεία σε ένα δοσμένο δίκτυο. Σε πολλές εφαρμογές είναι σημαντικό να βρεθούν παραπάνω από ένα μονοπάτια ανάμεσα σε δύο κόμβους ενός δικτύου. Αυτό σημαίνει ότι μπορεί να ζητηθεί να υπολογιστούν μονοπάτια διαφορετικά από το συντομότερο ανάμεσα σε δύο κόμβους. Για να βρεθεί το συντομότερο μονοπάτι μπορούν να χρησιμοποιηθούν αλγόριθμοι όπως ο αλγόριθμος Dijkstra ή ο αλγόριθμος Bellman-Ford και να επεκταθούν έτσι ώστε να βρίσκουν περισσότερα από ένα μονοπάτια. Ο αλγόριθμος δρομολόγησης Κ συντομότερων μονοπατιών είναι μία γενίκευση για το πρόβλημα συντομότερου μονοπατιού (shortest path problem). Ο αλγόριθμος δεν βρίσκει μόνο το συντομότερο μονοπάτι αλλά βρίσκει Κ μονοπάτια με πρώτο το συντομότερο και τα επόμενα με αύξουσα σειρά ως προς το κόστος. Το πρόβλημα μπορεί να περιοριστεί ώστε να έχει τα Κ συντομότερα μονοπάτια χωρίς βρόχους ή με βρόχους.

Από το 1957 έχουν υπάρξει πολλές δημοσιεύσεις πάνω στον αλγόριθμο δρομολόγησης Κ συντομότερων μονοπατιών. Τα περισσότερα από τα θεμελιώδη έργα δεν βρίσκουν μόνο το συντομότερο μονοπάτι ανάμεσα σε ένα ζευγάρι κόμβων, αλλά βρίσκουν μία ακολουθία από Κ συντομότερα μονοπάτια. Τα έργα αυτά ξεκινάνε στη δεκαετία του 60 μέχρι το 2001. Από το σημείο εκείνο και μετά η περισσότερη έρευνα γίνεται πάνω σε εφαρμογές του αλγορίθμου. Το 2010, ο Michael Gunter δημοσίευσε ένα βιβλίο "Symbolic Calculation of K Shortest Paths and Related Measures with the Stochastic Process Algebra Tool KASPA" .

Υπάρχουν δύο βασικές παραλλαγές του αλγόριθμου δρομολόγησης K συντομότερων μονοπατιών όπως αναφέρθηκαν και παραπάνω. Άλλες παραλλαγές στην ουσία εμπίπτουν στις παρακάτω δύο περιπτώσεις. Στην πρώτη περίπτωση επιτρέπονται βρόχοι που σημαίνει ότι ένα μονοπάτι επιτρέπεται να επισκεφτεί τον ίδιο κόμβο πολλαπλές φορές [EPP98]. Στην δεύτερη περίπτωση δεν επιτρέπονται βρόχοι και τα μονοπάτια θεωρούνται απλά, δηλαδή σε κάθε μονοπάτι οι κόμβοι που το αποτελούν εμφανίζονται μόνο μια φορά [YEN71].

6.2 Περιγραφή αλγορίθμου

Ο αλγόριθμος K-best είναι μία γενίκευση του αλγόριθμου Dijkstra γι' αυτό το λόγο ακολουθεί παρόμοια λογική με την λογική του αλγορίθμου Dijkstra. Η βασική διαφορά είναι ότι ενώ στον αλγόριθμο Dijkstra οι πληροφορίες για τους κόμβους που επισκεπτόμαστε είναι μόνο η απόσταση από τον κόμβο αρχή και ο προηγούμενος κόμβος από αυτόν, στον αλγόριθμο K-best αποθηκεύεται ολόκληρο το μονοπάτι.
Την εξέλιξη του αλγορίθμου K-best θα μπορούσε να την δει κανείς και σαν ένα δέντρο με ρίζα τον κόμβο πηγή και κλαδιά μονοπάτια πάνω στον γράφο που δημιουργούνται με μια λογική που θα εξηγηθεί παρακάτω. Ως φύλλα του δέντρου θεωρούνται οι τελευταίοι κόμβοι για το κάθε μονοπάτι μέχρι εκείνο το σημείο που έχει φτάσει ο αλγόριθμος.
Θα συμβολίσουμε P_n ένα μονοπάτι το οποίο ξεκινάει από τον κόμβο πηγή και καταλήγει στον κόμβο n και θα σημειώνουμε το κόστος για αυτό το μονοπάτι το οποίο θα είναι το κόστος για να διασχίσουμε το μονοπάτι μέχρι τον κόμβο n. Επίσης, κατά τη διάρκεια του αλγορίθμου θα σημειώνεται ο αριθμός των συντομότερων μονοπατιών που έχουν βρεθεί για έναν κόμβο, δηλαδή έστω C_n ο αριθμός αυτός για τον κόμβο n.
Ο αλγόριθμος ξεκινάει θέτοντας $C_i = 0$ για κάθε κόμβο i στο γράφο. Επίσης, έστω P_s το μονοπάτι το οποίο θα περιέχει μόνο τον κόμβο πηγή s για το οποίο θα σημειωθεί ότι έχει κόστος ίσο με το μηδέν. Επιπλέον, θα θεωρήσουμε ένα σύνολο S το οποίο θα περιέχει όλα τα μονοπάτια που έχουν βρεθεί μέχρι τώρα. Κατά τη διάρκεια του αλγορίθμου θα εντοπίζουμε το μονοπάτι μέσα από το σύνολο S που θα έχει το ελάχιστο κόστος και μόλις το βρούμε θα αυξάνουμε το μετρητή C_i για τον τελευταίο κόμβο αυτού του μονοπατιού. Στη συνέχεια, εάν ο μετρητής του τελευταίου κόμβου αυτού του μονοπατιού είναι μικρότερος ή ίσος του K τότε για τον τελευταίο κόμβο θα δημιουργούμε νέα μονοπάτια, επεκτάσεις αυτού τόσα όσα και οι γείτονες του τελευταίου κόμβου σημειώνοντας παράλληλα και τα αντίστοιχα κόστη. Κατόπιν, θα προσθέτουμε όλα τα μονοπάτια στο σύνολο S και ο αλγόριθμος θα επαναλαμβάνεται. Ο αλγόριθμος θα τερματίσει εάν ο μετρητής του τελικού κόμβου γίνει ίσος με K γιατί αυτό θα σημαίνει ότι έχουμε βρει K συντομότερα μονοπάτια μέχρι τον τελικό κόμβο. Μόλις τερματίσει ο αλγόριθμος θα συλλέξουμε τα K μονοπάτια P_t από το σύνολο S.
Ο παραπάνω αλγόριθμος θα γίνει πιο κατανοητός στη συνέχεια με την παρουσίαση ψευδοκώδικα και με ένα παράδειγμα που θα εκτελείται βήμα προς βήμα.
Σημείωση: Για τον παραπάνω αλγόριθμο θεωρήθηκε ότι σε ένα μονοπάτι επιτρέπονται βρόχοι. Με έναν επιπρόσθετο έλεγχο αυτό μπορεί να αποτραπεί και έτσι ο αλγόριθμος να βρίσκει μόνο απλά μονοπάτια.

6.3 Ψευδοκώδικας

Ορισμοί:

G(V, E): κατευθυνόμενος γράφος με βάρη, περιέχει ένα σύνολο κόμβων V και ένα σύνολο ακμών E.
w(u, v): Βάρος της ακμής uv, μη αρνητικό (προφανώς είναι πιθανόν να είναι διαφορετικό του **w(v,u)**).

s: ο κόμβος πηγή
t: ο κόμβος προορισμού
K: ο ζητούμενος αριθμός συντομότερων διαδρομών
$\mathbf{P_u}$: ένα μονοπάτι από τον **s** στον **u**
S: ένα σύνολο που θα περιέχει τα μονοπάτια που έχουν βρεθεί μέχρι ένα συγκεκριμένο σημείο του αλγορίθμου
SP: το σύνολο που θα περιέχει τα συντομότερα μονοπάτια από τον **s** στον **t** που έχουν βρεθεί
$\mathbf{count_u}$: ο αριθμός των συντομότερων μονοπατιών που έχουν βρεθεί με τελικό κόμβο τον **u**

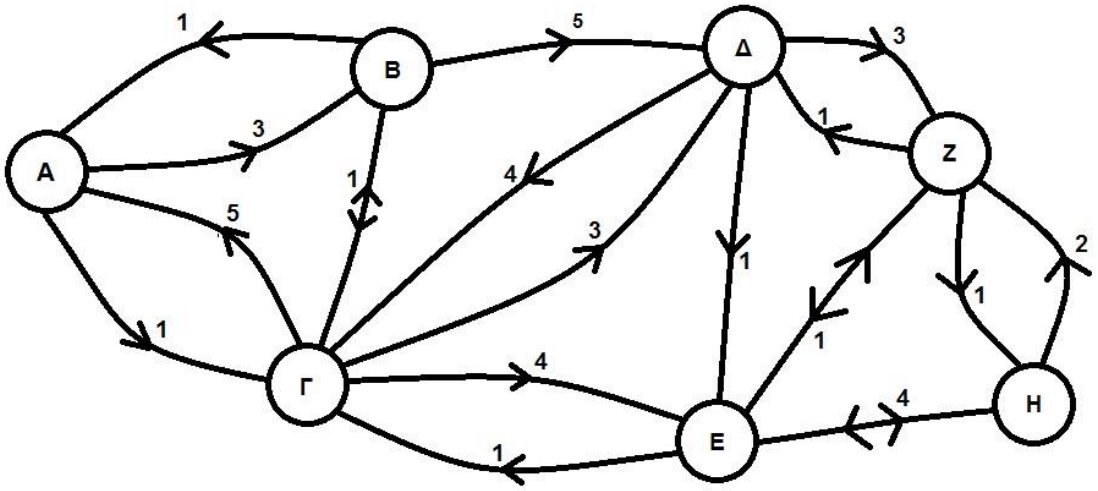
Αλγόριθμος:

```
SP = empty,
count_u = 0, for all u in V
insert path P_s = {s} into S with cost 0
while S is not empty and count_t < K:
     let P_u be the shortest cost path in S with cost C
     S = S - {P_u }, count_u = count_u + 1
     if u = t then SP = SP U P_u
     if count_u ≤ K then
     for each vertex v adjacent to u:
          let P_v be a new path with cost C + w(u, v) formed by
     concatenating edge (u, v) to path P_u
     insert P_v into S
```

6.4 Παράδειγμα Εκτέλεσης Αλγορίθμου

Ο αλγόριθμος που παρουσιάστηκε στην προηγούμενη παράγραφο επιτρέπει βρόχους στα μονοπάτια που σημαίνει πώς σε ένα μονοπάτι μπορεί να εμφανίζεται ένας κόμβος πάνω από μία φορά. Σ' αυτήν την υποενότητα θα χρησιμοποιηθεί ένας επιπλέον έλεγχος ώστε να αποφευχθούν οι βρόχοι στα μονοπάτια. Ο έλεγχος αυτός θα κοιτάει εάν ένας κόμβος που προστίθεται σε ένα μονοπάτι υπάρχει ήδη στο μονοπάτι. Αν υπάρχει τότε δεν προεκτείνει το μονοπάτι αλλιώς προχωράει κανονικά.

Έστω ο παρακάτω γράφος:



Ζητείται η εύρεση των Κ συντομότερων διαδρομών από τον κόμβο Α στον κόμβο Δ για Κ = 4.

Λόγω του ότι όλα τα μονοπάτια στο συγκεκριμένο κόμβο αρχίζουν από τον κόμβο πηγή (εδώ ο Α), ένα μονοπάτι `P = {A,α3,α6,α2,α8}` θα συμβολίζεται $P_{\alpha 8}$`(α3,α6,α2)` υποδηλώνοντας τους ενδιάμεσους κόμβους (αν υπάρχουν) και τον τελικό κόμβο.

Έστω S το σύνολο που θα αποθηκεύουμε τα μονοπάτια. Αρχικά ξεκινώντας από τον κόμβο Α έχουμε το μοναδικό μονοπάτι P_A `= {A}` με κόστος 0.
Θα συμβολίζουμε το σύνολο S ως εξής:

`S = {Pa : ca, Pb : cb, Pg : cg}`

Ο παραπάνω συμβολισμός σημαίνει πως το σύνολο S περιέχει τα μονοπάτια `Pa, Pb, Pg` με κόστη `ca, cb, cg` αντίστοιχα.

Επομένως το σύνολο S αρχικά θα είναι:

$S = \{P_A : 0\}$

Στο σημείο αυτό ο αλγόριθμος επιλέγει το μονοπάτι μέσα από το σύνολο S με το ελάχιστο κόστος. Εδώ είναι το P_A. Αφαιρούμε το P_A από το σύνολο S και σημειώνουμε πώς για τον κόμβο Α έχουμε βρει ένα ελάχιστο μονοπάτι.

Εφόσον ο κόμβος Α δεν είναι ο τελικός κόμβος, δηλαδή ο Δ, δεν προσθέτουμε το μονοπάτι στο σύνολο SP.

Στην συνέχεια εξετάζουμε αν ο μετρητής του κόμβου Α είναι μικρότερος του Κ, δηλαδή 4.
Εάν είναι τότε προχωράμε για να επεκτείνουμε το μονοπάτι αλλιώς ψάχνουμε το επόμενο μονοπάτι του συνόλου S με το ελάχιστο κόστος. Είναι προφανές πως εφόσον έχουμε αφαιρέσει το μονοπάτι αυτό από το σύνολο S δεν υπάρχει περίπτωση να το ξαναεντοπίσουμε.
Εδώ ο μετρητής του Α είναι 1 < 4 άρα συνεχίζουμε για να δημιουργήσουμε τις προεκτάσεις το μονοπατιού.

Εφόσον ο Α έχει 2 γειτονικούς κόμβους (τον Β και τον Γ) θα δημιουργήσουμε 2 νέα μονοπάτια.
Τα μονοπάτια αυτά θα είναι τα Α – Β και Α – Γ με κόστη 0 + 3 = 3 και 0 + 1 = 1 αντίστοιχα.
Τα παραπάνω μονοπάτια προστίθενται στο σύνολο S. Άρα το σύνολο S θα είναι:

$S = \{P_B : 3,\ P_\Gamma : 1\}$

Στο σημείο αυτό αρχίζουμε την διαδικασία από την αρχή και αναζητούμε το ελάχιστο μονοπάτι μέσα στο σύνολο S.

Το επόμενο ελάχιστο μονοπάτι θα είναι το P_Γ. Αφαιρούμε το P_Γ από το S και αυξάνουμε τον μετρητή του Γ κατά 1.

Εφόσον ο κόμβος Γ δεν είναι ο τελικός κόμβος δεν προσθέτουμε το μονοπάτι στο σύνολο SP.

Συνεχίζουμε την προέκταση του P_Γ αφού ο μετρητής του Γ είναι μικρότερος του 4.

Ο κόμβος Γ έχει γειτονικούς κόμβους τους Α, Β, Δ, Ε όμως επειδή ο Α υπάρχει ήδη στο μονοπάτι P_Γ αγνοείται. Επομένως τα μονοπάτια που δημιουργούνται και προστίθενται στο σύνολο S είναι τα $P_B(\Gamma)$ με κόστος 1 + 1 = 2 , $P_\Delta(\Gamma)$ με κόστος 1 + 3 = 4 και $P_E(\Gamma)$ με κόστος 1 + 4 = 5.

Το σύνολο S θα είναι: $S = \{P_B : 3,\ P_B(\Gamma) : 2,\ P_\Delta(\Gamma) : 4,\ P_E(\Gamma) : 5\}$

Το ελάχιστο μονοπάτι θα είναι τώρα το $P_B(\Gamma)$, έτσι αφαιρείται από το S και ο μετρητής του Β αυξάνεται κατά 1.

Εφόσον ο κόμβος Β δεν είναι ο τελικός κόμβος δεν προσθέτουμε το μονοπάτι στο σύνολο SP.

Συνεχίζουμε την προέκταση του $P_B(\Gamma)$ αφού ο μετρητής του Β είναι μικρότερος του 4.

Ο κόμβος Β έχει γειτονικούς κόμβους τους Α, Γ, Δ, όμως επειδή ο Α και ο Γ υπάρχουν ήδη στο μονοπάτι $P_B(\Gamma)$ αγνοούνται. Επομένως τα μονοπάτια που δημιουργούνται και προστίθενται στο σύνολο S είναι το $P_\Delta(\Gamma-B)$ με κόστος 2 + 5 = 7.

Το σύνολο S θα είναι: $S = \{P_B : 3,\ P_\Delta(\Gamma) : 4,\ P_E(\Gamma) : 5,\ P_\Delta(\Gamma-B) : 7\}$

Το ελάχιστο μονοπάτι είναι το P_B, έτσι αφαιρείται από το S και ο μετρητής του Β αυξάνεται κατά 1.

Ο κόμβος Β δεν είναι ο τελικός κόμβος άρα δεν προσθέτουμε το μονοπάτι στο σύνολο SP και συνεχίζουμε με την προέκταση του αφού ο μετρητής του Β είναι μικρότερος του 4.

Ο κόμβος Β έχει γειτονικούς κόμβους τους Α, Γ, Δ, όμως επειδή ο Α υπάρχει ήδη στο μονοπάτι P_B αγνοείται. Επομένως τα μονοπάτια που δημιουργούνται και προστίθενται στο σύνολο S είναι τα $P_\Gamma(B)$ με κόστος 3 + 1 = 4 και $P_\Delta(B)$ με κόστος 3 + 5 = 8.

Το σύνολο S θα είναι:
$S = \{ P_\Delta(\Gamma) : 4,\ P_E(\Gamma) : 5,\ P_\Delta(\Gamma-B) : 7,\ P_\Gamma(B) :4,\ P_\Delta(B) : 8\}$

Το ελάχιστο μονοπάτι θα είναι τώρα το $P_\Delta(\Gamma)$, έτσι αφαιρείται από το S και ο μετρητής του Δ αυξάνεται κατά 1.

Ο κόμβος Δ είναι ο τελικός κόμβος άρα αποθηκεύουμε το μονοπάτι $P_\Delta(\Gamma\text{-}B)$ στο σύνολο SP.
$SP = \{ P_\Delta(\Gamma) : 4\}$

Εφόσον ο Δ είναι ο τελικός κόμβος δεν θέλουμε να συνεχίσουμε την προέκταση του μονοπατιού. Αν συνεχιστεί το μονοπάτι τότε για καταλήξουμε πάλι στον κόμβο Δ κάποια στιγμή στο μέλλον θα έχουμε βρόχο.

Το σύνολο S θα είναι: $S = \{ P_\Delta(\Gamma-B) : 7,\ P_E(\Gamma) : 5,\ P_\Gamma(B) :4,\ P_\Delta(B) : 8\}$

Το ελάχιστο μονοπάτι θα είναι τώρα το $P_\Gamma(B)$, έτσι αφαιρείται από το S και ο μετρητής του Γ αυξάνεται κατά 1.

Ο κόμβος Γ δεν είναι ο τελικός κόμβος άρα το μονοπάτι $P_\Gamma(B)$ δεν προστίθεται στο σύνολο SP.

Συνεχίζουμε την προέκταση του $P_\Gamma(B)$ αφού ο μετρητής του Γ είναι μικρότερος του 4.

Ο κόμβος Γ έχει γειτονικούς κόμβους τους Α, Β, Δ, Ε αλλά ο Α και ο Β αγνοούνται καθώς εμπεριέχονται στο μονοπάτι $P_\Gamma(B)$. Τα νέα μονοπάτια θα είναι τα $P_\Delta(B-\Gamma)$ με κόστος 4 + 3 = 7 και $P_E(B-\Gamma)$ με κόστος 4 + 4 = 8.

Το σύνολο S θα είναι:
$S = \{ P_\Delta(\Gamma-B) : 7,\ P_E(\Gamma) : 5,\ P_\Delta(B) : 8,\ P_\Delta(B-\Gamma) : 7,\ P_E(B-\Gamma) : 8\}$

Το ελάχιστο μονοπάτι θα είναι τώρα το $P_E(\Gamma)$, έτσι αφαιρείται από το S και ο μετρητής του Ε

αυξάνεται κατά 1.

Ο κόμβος Ε δεν είναι ο τελικός κόμβος άρα δεν αποθηκεύουμε το μονοπάτι $P_E(\Gamma)$ στο σύνολο SP.

Συνεχίζουμε την προέκταση του $P_E(\Gamma)$ αφού ο μετρητής του Ε είναι μικρότερος του 4.

Ο κόμβος Ε έχει γειτονικούς κόμβους τους Γ, Ζ, Η όμως επειδή ο Γ υπάρχει ήδη στο μονοπάτι $P_E(\Gamma)$ αγνοείται. Επομένως τα μονοπάτια που δημιουργούνται και προστίθενται στο σύνολο S είναι τα $P_Z(\Gamma-E)$ με κόστος 5 + 1 = 6 και $P_H(\Gamma-E)$ με κόστος 5 + 4 = 9.

Το σύνολο S θα είναι:
$S = \{P_\Delta(\Gamma-B):7, P_\Delta(B):8, P_\Delta(B-\Gamma):7, P_E(B-\Gamma):8, P_Z(\Gamma-E):6, P_H(\Gamma-E):9\}$

Το ελάχιστο μονοπάτι θα είναι τώρα το $P_Z(\Gamma-E)$. Αφαιρείται από το σύνολο S και αυξάνεται ο μετρητής του Ζ κατά 1.

Ο Ζ δεν είναι ο τελικός κόμβος. Το $P_Z(\Gamma-E)$ δεν αποθηκεύεται στο σύνολο SP και αφού ο μετρητής του Ζ είναι μικρότερος του 3 συνεχίζουμε την προέκταση του μονοπατιού.

Ο Ζ έχει γειτονικούς κόμβους τους Δ, Ε, Η αλλά ο Ε αγνοείται γιατί υπάρχει στο μονοπάτι ήδη. Τα νέα μονοπάτια θα είναι τα $P_\Delta(\Delta-E-Z)$ με κόστος 6 + 1 = 7 και $P_H(\Gamma-E-Z)$ με κόστος 6 + 1 = 7.

Το σύνολο S θα είναι:
$S = \{P_\Delta(\Gamma-B):7, P_\Delta(B):8, P_\Delta(B-\Gamma):7, P_E(B-\Gamma):8, P_H(\Gamma-E-Z):7, P_\Delta(\Gamma-E-Z):7, P_H(\Gamma-E):9\}$

Σε αυτό το σημείο ο αλγόριθμος θα συνεχίσει κανονικά μέχρι να βρει τα 4 πρώτα μονοπάτια που να καταλήγουν στο Δ. Παρατηρούμε όμως πως υπάρχουν 3 μονοπάτια στο σύνολο S με ελάχιστο κόστος σε σχέση με τα υπόλοιπα και καταλήγουν στον κόμβο Δ. Εφόσον δεν επεξεργαζόμαστε γράφο με αρνητικά βάρη μπορούμε κατευθείαν να προσθέσουμε αυτά τα 3 μονοπάτια στο σύνολο SP.

Το παραπάνω σχόλιο μπορεί να υλοποιηθεί και να προστεθεί στον βασικό αλγόριθμο ώστε να αποφευχθούν περιττές επαναλήψεις.

Επομένως θα έχουμε συνολικά τα 4 ζητούμενα μονοπάτια τα οποία θα είναι τα εξής:

$SP = \{P_\Delta(\Gamma) : 4, P_\Delta(\Gamma-B) : 7, P_\Delta(B) : 7, P_\Delta(\Gamma-E-Z) : 7\}$

6.5 Εφαρμογές

Ο αλγόριθμος δρομολόγησης K συντομότερων μονοπατιών χρησιμοποιείται στις παρακάτω εφαρμογές:

- σχεδιασμός γεωγραφικών μονοπατιών
- δρομολόγηση δικτύων
- hypothesis generation in computational linguistics
- sequence alignment and metabolic pathway finding in bioinformatics
- multiple object tracking
- οδικών δικτύων - δίκτυα μεταφορών : οι διασταυρώσεις θεωρούνται κόμβοι και οι δρόμοι θεωρούνται ακμές ενός γράφου που ενώνουν δύο διασταυρώσεις

7

Υλοποίηση Αλγορίθμου Dijkstra σε Python

Στην παρούσα διπλωματική εργασία χρησιμοποιήθηκε η έκδοση 3.4 της προγραμματικής γλώσσας Python. Στην συνέχεια θα παρουσιαστούν τα αρχεία κώδικα που κατασκευάστηκαν και χρησιμοποιήθηκαν για την εκτέλεση του αλγορίθμου Dijkstra. Στην παρούσα ενότητα θα παρουσιαστούν παραδείγματα εκτέλεσης τμημάτων του κώδικα πάνω σε μια μικρή δομή γράφου ώστε να γίνει πιο κατανοητά.

7.1 Δημιουργία Αρχείου Λίστας Γειτνίασης (Adjacency List)

Ο παρακάτω κώδικας δέχεται ως όρισμα ένα αρχείο μορφής .xls που περιέχει μια Λίστα Συνδέσεων (Connectivity Table) και παράγει ένα αρχείο μορφής .csv που περιέχει την αντίστοιχη Λίστα Γειτνίασης. Το αρχείο που περιέχει τον παρακάτω κώδικα ονομάζεται: `Connectivity2AdjacencyListFile.py`

```
   """Read Connectivity Table Structure from Excel File and Create
a Csv File with the Adjacency List Structure"""

   #Import Library to Read Excel Files
1  import xlrd,sys,csv

   #Open the Excel File
2  workbook = xlrd.open_workbook(sys.argv[1])

   #Choose the first sheet
3  worksheets = workbook.sheet_names()
4  worksheet = workbook.sheet_by_name(worksheets[0])
```

```
   #col_values(A,B): A determines the column, B determines the
cell from which you start to scan
   #e.g col_values(4,7) means select the fifth(5th) column and
scan from 8th cell until the last cell in this column

   #Save Link IDs to determine the number of links
5  linkIDs = []
6  for i in worksheet.col_values(1,1):
7     linkIDs.append(i)

   #Create the Adjacency List structure as dictionary
{node:[node1,cost1,node2,cost2,...,noden,costn]}
   #Every key is a node from Graph and its value is a list
[neighbour1,cost1,neighbour2,cost2,...,neighbourn,costn]
8  from collections import defaultdict
9  Adjacency_List = defaultdict(list)

   #For every row in Connectivity Table
10 for i in range(1,len(linkIDs)+1):
      #The 2nd column has the nodes which a link starts
11    from_node = int(worksheet.row(i)[2].value)
      #The 3rd column has the nodes which a link ends
12    to_node = int(worksheet.row(i)[3].value)
      #The 9th column has the costs - weights of links
13    cost = float(worksheet.row(i)[9].value)
      #Add end node as neighbour of start node
14    Adjacency_List[from_node].append(to_node)
15    Adjacency_List[to_node].append(from_node)#For undirected
Graph!!!!

      #Add cost of start to end node
16    Adjacency_List[from_node].append(cost)
17    Adjacency_List[to_node].append(cost)#For undirected
Graph!!!!

   #Create a Csv File to write the Adjacency List
18 csvfile = open(sys.argv[2], 'w', newline='\n')
19 writer = csv.writer(csvfile, delimiter=';',
quoting=csv.QUOTE_MINIMAL)
20 for node in Adjacency_List:
21    writer.writerow([node] + [i for i in Adjacency_List[node]])
   # The last two lines are for delete the new line (\n) character
at the end of the file
```

Παρακάτω παρουσιάζεται ένα τμήμα ενός αρχείου GE_Rail_Links.xls όπου περιέχει την λίστα συνδέσεων ενός σιδηροδρομικού δικτύου της Γερμανίας.

```
 1 2502923 252423 252424 Germany  107150011  0  1  1  24,547
 2 2502925 252424 252425 Germany  107150011  0  0  1  14,403
 3 2001089 252074 201165 Germany  107060000  0  0  1   5,185
 4 2001090 201118 201117 Germany  107110110  0  0  1  12,714
 5 2503260 252648 252649 Germany  107110110  0  0  1   6,755
 6 2001091 252649 201115 Germany  107070203  0  1  1  14,902
 7 2503262 251384 252650 Germany  107110101  0  0  1   5,724
 8 2503263 252650 252647 Germany  107110101  0  0  1   1,314
 9 2503265 252651 252652 Germany  107100203  0  0  1   3,726
10 2001092 200712 201131 Germany  107100203  0  1  1  12,549
11 2503267 252654 252193 Germany  107100106  0  0  1   3,445
12 2503268 252193 252195 Germany  107100106  0  0  1   1,162
13 2503269 252647 252655 Germany  107110101  0  1  1  26,677
14 2001093 201070 251382 Germany  107110109  0  1  1  20,393
15 2503271 252656 252650 Germany  107110101  0  0  1    0,98
16 2503039 252480 250874 Germany  107090409  0  1  1   9,295
17 2503297 252667 252184 Germany  107100110  0  1  1   2,325
```

Η πρώτη στήλη παρουσιάζει το ID της κάθε σύνδεσης. Η τρίτη και η τέταρτη στήλη περιέχει τους κόμβους από και προς αντίστοιχα. Η ένατη στήλη περιέχει το μήκος της σιδηροδρομικής γραμμής δηλαδή το κόστος της αντίστοιχης ακμής του γράφου. Όπως φαίνεται και στις γραμμές 11, 12 και 13 του κώδικα οι στήλες που μας ενδιαφέρουν είναι η τρίτη, η τέταρτη και η δέκατη. Για παράδειγμα με την εντολή:
`from_node = int(worksheet.row(i)[2].value)` αποθηκεύουμε το ID του κόμβου που βρίσκεται στην γραμμή i και στην στήλη 3 (η αρίθμηση στις λίστες της Python ξεκινάει από το 0 – μηδέν).
Μετά την εκτέλεση του παραπάνω κώδικα το αρχείο που θα προκύψει θα έχει την παρακάτω μορφή:

```
200704;250876;5.231;200703;7.274
200705;251711;25.742;251712;7.134
200706;251708;8.988;251707;15.727
200707;251819;6.4559999999999995;251772;16.425
200708;201154;10.767;201155;11.717
200709;250581;11.813;250270;30.983
200710;252685;32.759;252686;4.986
200711;201133;7.569;201132;4.521
200712;201131;12.549;251385;13.325
200713;251778;9.835;201140;2.019
200714;251386;13.012;201064;5.024
200715;251791;1.818;252666;9.756
```

Το παραπάνω τμήμα του αρχείου Adjacency_ListGE.csv παρουσιάζει την λίστα γειτνίασης του σιδηροδρομικού δικτύου της Γερμανίας.

Η πληροφορία που δίνεται στην πρώτη γραμμή είναι πώς ο κόμβος `200704` συνδέεται με τον κόμβο `250876` με κόστος `5.231` και με τον κόμβο `200703` με κόστος `7.274`.

Η παραπάνω μορφή αρχείου θα χρησιμοποιείται ως είσοδος για τα επόμενα προγράμματα υλοποίησης της δομής ενός γράφου πάνω στον οποίο θα εκτελείται ο αλγόριθμος Dijkstra.

7.2 Δημιουργία Αρχείου Δομής Γράφου

Το παρακάτω αρχείο ονομάζεται `Graph_Structure.py` και υλοποιεί την δομή ενός γράφου που χρειάζεται για την εκτέλεση των αλγορίθμων Dijkstra και K-Best Paths στην συνέχεια.

```
   """ Creation of Graph Structure """

1  from collections import defaultdict
2  import random

   ##Creating Graph Structure
3  class Graph:

4    def __init__(self):
       #A Graph has a set of nodes
5      self.nodes = set()
      #The set of Edges is a dictionary with {key:value} with key
as a node and value as a set of its neighbours
6      self.edges = defaultdict(set)
       #Every cost-weight of an edge is saved in a dictionary
{key:value} with key as a pair-tuple (nodei,nodej) and as value
the cost of the edge nodei-nodej
7      self.distances = {}

8    def __str__(self):
9      return '\nThe Nodes of the Graph are: ' + str(self.nodes) +
'\n' + \
10   '\nThe Edges of the Graph are: ' + str(self.edges.items()) +
'\n' +    \
11   '\nThe Distances of the Graph are: ' + str(self.distances)

12   def addnode(self, value):
       #A new node added to the set of nodes
13     self.nodes.add(value)

14   def addedge(self, from_node, to_node, distance):
       #To add an edge we add a neighbour to list of neighbours of
the node:from_node to dictionary Graph.edges
15     self.edges[from_node].add(to_node)
```

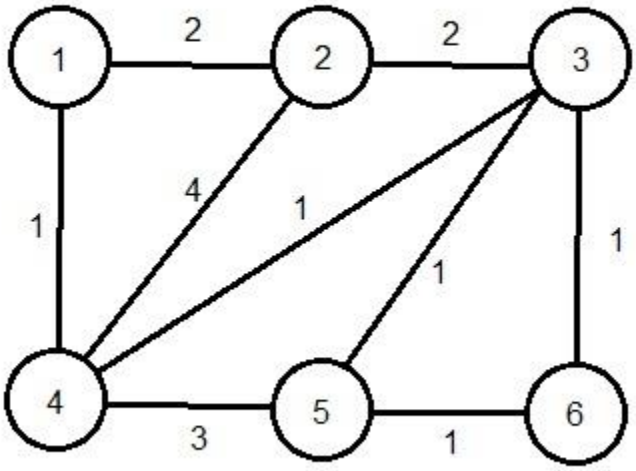
```
16      self.edges[to_node].add(from_node) #For non-directed Graph

        #To add or change a cost of an edge we add or change the
value of the edge:(from_node,to_node) in the dictionary
Graph.distances
17      self.distances[(from_node, to_node)] = distance
18      self.distances[(to_node, from_node)] = distance #For non-
directed Graph

19    def Path(self,start,end): # Detects a path between two nodes

20      if start == end: return 1 # If start is end...

21      remaining_nodes = self.nodes.copy() # A set that we remove
every node we have visited. Contains every node that we have not
visited yet

22      visited = set([start]) # Contains every node we have
visited

23      current_node = start # Starting from the start node

24      while 1 :
            # Access every neighbour of the current node
25          for neighbour in self.edges[current_node]:
26            if neighbour in remaining_nodes: # If we have not
visited it before
27              visited.add(neighbour) # Add it to visited set

            # Remove the current node so we won't access it again
in the future
28          remaining_nodes.remove(current_node)
            # if we access all nodes that we could access
29          if not remaining_nodes&visited: break
            # Access randomly a node from the set of nodes we have
not access yet
            # and they are neighbours of previous nodes we accessed
before
30          current_node =
random.sample(visited&remaining_nodes,1)[0]

31          if current_node == end: # If we access the end we stop
32            return 1 # Return true

33      return 0 # If we have visited all nodes that we could and
we did not access the end return false
```

```
34    def Connected(self): # Check if the Graph is a Connected
Graph
35      strongconnection = 0
36      random_node = random.sample(self.nodes,1)[0]
37      for node in self.nodes:
38          if self.Path(random_node,node) and
self.Path(node,random_node):
39              strongconnection = strongconnection + 1
40          if not(self.Path(random_node,node) or
self.Path(node,random_node)):
41              return 'The graph is not connected. Two nodes that
are not connected are:'+str(random_node)+' '+str(node)
        # If the connections between all nodes are |V| means that
every node is connected to all other nodes
42      if strongconnection == len(self.nodes):
43        return 'The Graph is strongly connected'
44      else: return 'The Graph is weakly connected'
```

Όπως βλέπουμε από την γραμμή κώδικα 3 ο γράφος είναι μια κλάση που περιέχει ένα σύνολο από κόμβους (γραμμή 5), ένα λεξικό με ακμές (γραμμή 6) και ένα λεξικό με κόστη – αποστάσεις των ακμών (γραμμή 7). Στην Python ένα λεξικό (dictionary) είναι μια δομή της μορφής:
{κλειδί 1 : τιμή 1 , κλειδί 2 : τιμή 2, … , κλειδί ν : τιμή ν}. Με αυτόν τον τρόπο μπορούμε να έχουμε πρόσβαση σε τιμές όταν γνωρίζουμε το κλειδί και το αντίστροφο. Αξιοσημείωτο είναι πως στην Python το κλειδί (key) αλλά και η τιμή (value) σε ένα λεξικό μπορεί να είναι μια οποιαδήποτε δομή όπως μια λίστα, μια συμβολοσειρά, μια κλάση ακόμα και ένα λεξικό.
Το σύνολο των κόμβων στην κλάση `Graph` είναι ένα set της Python. Έστω οι κόμβοι 1, 2, 3 ,4 ,5 ,6 του γράφου που θέλουμε να επεξεργαστούμε. Το σύνολο των κόμβων του γράφου θα είναι:
`Graph.nodes = {1, 2, 3, 4, 5, 6}`

Το σύνολο των ακμών στην κλάση `Graph` θα είναι ένα λεξικό που θα έχει ως κλειδιά τους κόμβους και ως τιμές ένα σύνολο των αντίστοιχων γειτόνων.

Έστω ο παρακάτω γράφος:



Το σύνολο των ακμών θα είναι:

```
Graph.edges={1:{2,4},2:{1,3,4},3:{2,4,5,6},4:{1,2,3,5},5:{3,4,6},6
:{3,5}}
```

Οι αποστάσεις των ακμών στην κλάση `Graph` θα είναι ένα λεξικό που θα έχει ως κλειδιά τις ακμές και ως τιμές το αντίστοιχο κόστος - απόσταση. Μια ακμή είναι μια τούπλα (α , β) δηλώνοντας ότι ο κόμβος α συνδέεται με τον κόμβο β. Για τον παραπάνω γράφο οι αποστάσεις θα είναι:

```
Graph.distances =
{(1,2):2,(3,2):2,(3,6):1,(4,5):3,(4,1):1,(5,4):3,(2,1):2,(6,3):1,(
5,6):1,(2,3):2,(1,4):1,(4,3):1,(4,2):4,(5,3):1,(3,4):1,(2,4):4,(6,
5):1,(3,5):1}
```

Λόγω του ότι ο παραπάνω γράφος δεν είναι προσανατολισμένος βλέπουμε πως εμφανίζονται δύο φορές τα κόστη για την ίδια ακμή αφού για την Python το ζεύγος (α,β) είναι διαφορετικό από το ζεύγος (β,α). Τα παραπάνω έχουν υλοποιηθεί θεωρώντας τον γράφο κατευθυνόμενο στην γενική περίπτωση. Εάν στο πρόβλημα που χρησιμοποιείται η κλάση `Graph` γνωρίζουμε εξ' αρχής πως ο γράφος είναι μη προσανατολισμένος τότε οι γραμμές 16 και 18 προστίθενται στον κώδικα.

Η κλάση περιέχει τις μεθόδους `addnode` και `addedge` ώστε να προστίθενται νέοι κόμβοι και νέες ακμές στον ήδη υπάρχον γράφο.

Τέλος οι μέθοδοι Path και Connected υπάρχουν ώστε εάν είναι απαραίτητο να ελέγχεται εάν υπάρχει μονοπάτι ανάμεσα σε δύο κόμβους του γράφου και εάν ο γράφος είναι συνδεδεμένος.

7.3 Αρχείο Δημιουργίας ενός Γράφου από ένα αρχείο Λίστας Γειτνίασης

Ο παρακάτω κώδικας χρησιμοποιείται για δημιουργεί μια δομή γράφου όπως περιγράφηκε στην προηγούμενη υποενότητα μέσω ενός αρχείου .csv που περιέχει μια λίστα γειτνίασης.

```
"""Read from a file with an Adjacency List Data to Create the Graph
Structure"""

import Graph_Structure, csv

" This is when we have Adjacency List in Csv File "
## Reading from Csv File
class Graph_from_Adj_List_Csv_File:


    def __init__(self):
        # Create an empty Adjacency List
        self.Adjacency_List = []
        # Create an empty Graph Structure
        self.Graph = Graph_Structure.Graph()

    def read_file(self,file):
        # Open the file
        with open(file,'r') as csvfile:
            reader = csv.reader(csvfile, delimiter=';', quotechar='|')
            # Create a major list of lists: every small list is a row
from file
            for row in reader:
                self.Adjacency_List.append(row)

    def create_graph(self):
        # Create the Graph Structure
        for row in self.Adjacency_List:
            # Add every node from the first column to the set of nodes of
Graph
            self.Graph.addnode(int(row[0]))
            # For every row take every element starting from the second
element and put it in the Graph Structure as: node1,cost1,node2,cost2
etc...
            for i in range(1,len(row)-1,2):
                self.Graph.addnode(int(row[i]))

self.Graph.addedge(int(row[0]),int(row[i]),float(row[i+1]))
```

Το παραπάνω αρχείο ονομάζεται `CreateGraphFromFile.py` και καλείται από το αρχείο εκτέλεσης του αλγορίθμου Dijkstra και K-Best Paths.

7.4 Αρχείο Εκτέλεσης Αλγορίθμου Dijkstra

7.4.1 Κώδικας

Το παρακάτω αρχείο είναι μια συνάρτηση εκτέλεσης του αλγορίθμου Dijkstra, η οποία λαμβάνει ως είσοδο μια δομή γράφου που περιγράφηκε στις προηγούμενες υποενότητες, έναν κόμβο αρχή και έναν κόμβο τέλους. Η συνάρτηση επιστρέφει το συντομότερο μονοπάτι ανάμεσα στον κόμβο αρχή και τέλους και το κόστος – απόσταση που χρειάζεται για να το διανύσουμε.

```
""" Dijkstra Algorithm: Finds the shortest path in a Graph
between two nodes"""

1  def dijkstra(graph, source, destination):

2      visited_nodes = {source: 0} #A dictionary with nodes as
keys and corresponding costs from source as values
3      Paths = {} #A dictionary with nodes as keys and previous
nodes as values e.g. {'A':'B'} means that going to node A we must
first go through B

4      nodes = set(graph.nodes) #nodes is a set of all nodes from
graph

5      active_nodes = set() # This is the active set of nodes:
From here we search the minimum node every time to run the
Dijkstra Algorithm
6      active_nodes.add(source) # The first time we have only the
source node inside

       # Definition! Minimum node is the node that the path from
source to it has the minimum cost so far
7      min_node = source # The first time the minimum node is the
source node

8      while min_node != destination: # until we find the
destination node

           ## Find the minimum cost node in visited dictionary
9          for node in active_nodes: # For every existed node so
far, the first time we have only the source node inside
11                 if min_node is None:
12                     min_node = node
13                 elif visited_nodes[node] <
visited_nodes[min_node]:
```

```
14                      min_node = node

            # If we find the destination node as the minimum node
we stop the loop
15          if min_node == destination:
16              continue

17          active_nodes.remove(min_node) # Remove minimum node
from the set of active nodes so we won't access it again in the
future
18          current_weight = visited_nodes[min_node] # The cost of
min_node

            ## Update costs for neighbour nodes
19          for neighbour in graph.edges[min_node]: # Access the
neighbour nodes of min_node

20              weight = current_weight +
graph.distances[(min_node, neighbour)] # Calculate cost for every
neighbour as (cost from source to min_node) + (cost from min_node
to neighbour)

                # Update dictionaries if we did not visit a
neighbour before OR the current cost is less than an already
existed cost
21              if neighbour not in visited_nodes:
22                  visited_nodes[neighbour] = weight # Add the
node to dictionary with its cost
23                  Paths[neighbour] = min_node # Add the node to
dictionary with its previous node
24                  active_nodes.add(neighbour) # Add the current
node to active set of nodes to search the min_node when the main
loop starts again
25              elif weight < visited_nodes[neighbour]:
26                  visited_nodes[neighbour] = weight # Update the
dictionary
27                  Paths[neighbour] = min_node # Update the list

28          min_node = None



        ## Create the Shortest Path
29      current = destination # Start from Destination
30      path = [] # Empty path
        # Backtracking the Paths List
```

```
31      while current != source:
32          path.append(current)
33          current = Paths[current]
34      path.append(current)

35      return visited_nodes, path[::-1]
```

Για την εκτέλεση του παραπάνω κώδικα θα χρησιμοποιηθούν οι δομές που σχηματίζονται στις γραμμές 2, 3 και 5.

Στην γραμμή 2 η εντολή `visited_nodes = {source: 0}`θα δημιουργήσει ένα λεξικό το οποίο θα χρησιμοποιηθεί για να αποθηκεύει και να ενημερώνει τις αποστάσεις από την πηγή του κάθε κόμβου που ο αλγόριθμος επεξεργάζεται. Αρχικά ο πρώτος κόμβος στο λεξικό θα είναι ο κόμβος πηγή (source) και θα έχει προφανώς απόσταση μηδέν από τον εαυτό του.

Στην γραμμή 3 η εντολή `Paths = {}` δημιουργεί ένα κενό αρχικά λεξικό. Το λεξικό Paths θα έχει κλειδιά τους κόμβους του γράφου. Για κάθε κόμβο – κλειδί, η τιμή του θα είναι ο προηγούμενος κόμβος στο μονοπάτι που θα δημιουργείται κατά την διάρκεια εκτέλεσης του αλγορίθμου. Για παράδειγμα 'A':'B' σημαίνει πώς για να πάμε στον κόμβο 'A' θα περάσουμε από τον 'B' προηγουμένως. Το λεξικό Paths θα χρησιμοποιηθεί κατά το τέλος του αλγορίθμου για να εξάγουμε το ζητούμενο μονοπάτι.

Στην γραμμή 5 η εντολή `active_nodes = set()` δημιουργεί ένα κενό αρχικά σύνολο. Στο σύνολο αυτό ο αλγόριθμος θα αποθηκεύει τους κόμβους που έχει επισκεφθεί αλλά δεν έχει επεξεργαστεί ακόμα. Όπως θα δείξουμε παρακάτω ο αλγόριθμος επεξεργάζεται τον κόμβο που βρίσκει ότι έχει την ελάχιστη απόσταση από την πηγή σε σχέση με τους κόμβους που έχει μέχρι ήδη τώρα επισκεφθεί. Αυτή είναι μια βασική διαφορά που έχει η παραπάνω υλοποίηση σε σχέση με τον κλασσικό αλγόριθμο Dijkstra και συμβαίνει για λόγους ταχύτητας εκτέλεσης. Υπενθυμίζουμε πώς ο κλασικός αλγόριθμος Dijkstra θέτει αρχικά όλους τους κόμβους του γράφου να έχουν απόσταση από την πηγή στο άπειρο. Κατά συνέπεια όταν αναζητείται η μικρότερη απόσταση ο αλγόριθμος υποχρεούται να ψάχνει σε ένα τεράστιο σύνολο κόμβων. Αρχικά το σύνολο αυτό είναι μεγέθους |V| δηλαδή όσοι είναι οι κόμβοι του γράφου. Παρόλο που στην πορεία της εκτέλεσης το σύνολο αυτό θα μειώνεται, σε περιπτώσεις που το μονοπάτι το οποίο ζητείται είναι ανάμεσα σε δυο κόμβους που βρίσκονται κοντά μεταξύ τους, ο αλγόριθμος αναζητά άσκοπα μέσα σε ένα τεράστιο σύνολο από κόμβους. Με την εισαγωγή λοιπόν του συνόλου active_nodes αποτρέπεται αυτή η περιττή δαπάνη και ο αλγόριθμος ψάχνει πάντα μέσα από ένα σύνολο που μεγαλώνει αλλά στην χειρότερη περίπτωση δεν φτάνει την τάξη μεγέθους |V|. Προφανώς για ξεκινήσει ο αλγόριθμος το σύνολο αυτό θα περιέχει μόνο τον κόμβο πηγή όπως προστίθεται στην γραμμή 6.

Στις γραμμές κώδικα 9 έως 14 ο αλγόριθμος αναζητά τον κόμβο με την ελάχιστη απόσταση από την πηγή μέσα από το σύνολο active_nodes όπως περιγράψαμε και παραπάνω. Στην πρώτη επανάληψη προφανώς ο ελάχιστος κόμβος θα είναι ίδιος ο κόμβος πηγή. Όταν βρεθεί ο ελάχιστος κόμβος θα αφαιρείται από το ενεργό σύνολο αναζήτησης όπως φαίνεται στην γραμμή 17.

Ο αλγόριθμος θα συνεχίζει να εκτελείται μέχρι να βρεθεί πως ο ελάχιστος κόμβος είναι ο τελικός κόμβος. Αυτό φαίνεται στις γραμμές 8 και 15.

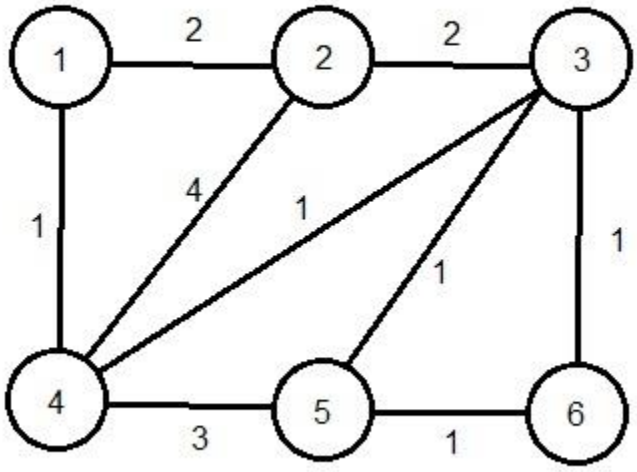
Στις γραμμές 18 έως 28 ο αλγόριθμος ενημερώνει τα λεξικά visited_node , Paths και το σύνολο active_nodes με βάση τους γειτονικούς κόμβους του ελάχιστου κόμβου που έχει εντοπίσει. Αρχικά αποθηκεύει το κόστος – απόσταση του ελάχιστου κόμβου (γραμμή 18). Στην συνέχεια για κάθε γείτονα του ελάχιστου κόμβου (γραμμή 19) ελέγχει εάν υπάρχει ήδη στο λεξικό visited_nodes (γραμμή 21) και αν δεν υπάρχει τον προσθέτει μαζί με το κόστος του (γραμμή 22). Το κόστος του κάθε γείτονα προκύπτει ως το άθροισμα του κόστους του ελάχιστου κόμβου και του κόστους της ακμής που συνδέει το ελάχιστο κόμβο με τον αντίστοιχο γειτονικό (γραμμή 20). Επίσης εάν δεν υπάρχει στο σύνολο visited_nodes τον προσθέτει στο σύνολο active_nodes (γραμμή 24). Εάν ο γειτονικός κόμβος που εξετάζεται είχε εισαχθεί σε παρελθοντική επανάληψη στο σύνολο visited_nodes τότε συγκρίνεται το νέο κόστος με το ήδη υπάρχον. Εάν το ήδη υπάρχον κόστος είναι μεγαλύτερο από το νέο, αντικαθίσταται (γραμμές 25 και 26).
Στις γραμμές 23 και 27 ενημερώνουμε το λεξικό Paths για τον κάθε γείτονα ώστε να γνωρίζουμε πώς ο προηγούμενος του είναι ο ελάχιστος κόμβος εάν ισχύουν οι συνθήκες των γραμμών 21 ή 25. Στην γραμμή 28 κάνουμε reset την μεταβλητή που αποθηκεύει τον ελάχιστο κόμβο ώστε να βρούμε τον νέο ελάχιστο στην επόμενη επανάληψη.

Στο επόμενο κομμάτι του κώδικα (γραμμές 29 έως 34) η επαναληπτική διαδικασία έχει τερματίσει που σημαίνει ότι ο τελικός κόμβος έχει βρεθεί. Αποτρέπουμε την περίπτωση ο τελικός κόμβος να μην υπάρχει στον γράφο ή να υπάρχει αλλά να μην συνδέεται με τον αρχικό στην κώδικα κλήσης της συνάρτησης Dijkstra όπως θα δούμε στην επόμενη υποενότητα. Στις γραμμές 29 έως 34 λοιπόν χρησιμοποιούμε το λεξικό Paths και με οπισθοδρόμηση (γραμμή 33) δημιουργούμε μια λίστα path που περιέχει το συντομότερο μονοπάτι από τον κόμβο αρχή έως τον τελικό κόμβο αλλά με ανάστροφη σειρά.

Τέλος στην γραμμή 34 η συνάρτηση επιστρέφει τα κόστη των ενδιάμεσων και του τελικού κόμβου, καθώς επίσης και το ανάστροφο μονοπάτι path.

7.4.2 Παράδειγμα εκτέλεσης του κώδικα

Έστω ο παρακάτω γράφος:



Όπως είδαμε και στην ενότητα 7.2 ο παραπάνω γράφος έχει την εξής δομή:

```
Graph.nodes = {1, 2, 3, 4, 5, 6}  (δομή κόμβων)

Graph.edges={1:{2,4},2:{1,3,4},3:{2,4,5,6},4:{1,2,3,5},5:{3,4,6},6
:{3,5}} (δομή ακμών)

Graph.distances =
{(1,2):2,(3,2):2,(3,6):1,(4,5):3,(4,1):1,(5,4):3,(2,1):2,(6,3):1,(
5,6):1,(2,3):2,(1,4):1,(4,3):1,(4,2):4,(5,3):1,(3,4):1,(2,4):4,(6,
5):1,(3,5):1} (δομή αποστάσεων)
```

Θεωρώντας λοιπόν ως είσοδο τον παραπάνω γράφο, κόμβο αρχή τον κόμβο `1` και τελικό κόμβο τον κόμβο `6` θα εκτελέσουμε τον κώδικα βήμα προς βήμα ώστε να γίνει η παράσταση των δομών και ο τρόπος με τον οποίο αλλάζουν κατά την εκτέλεση του κώδικα.

Αρχικά θα έχουμε:

`visited_nodes = {1:0}` καθώς ο κόμβος πηγή είναι ο `1`.
`Paths = {}`
`active_nodes = {1}`, ξεκινάμε τον έλεγχο από τον κόμβο πηγή.
`min_node = 1`

Μπαίνοντας στην επανάληψη ελέγχουμε αν ο `min_node` είναι ίσος με `6` που είναι ο τελικός κόμβος.

Στην πρώτη επανάληψη η εύρεση του ελαχίστου δεν έχει νόημα αφού γνωρίζουμε μόνο έναν κόμβο, τον αρχικό. Αφαιρούμε τον κόμβο 1 από το σύνολο active_nodes και αποθηκεύουμε την τιμή 0 στο current_weight αφού αυτή είναι η απόσταση του min_node από τον κόμβο 1 (γραμμή 18).
Το graph_edges[min_node] είναι ίσο με {2,4} από την δομή ακμών, άρα οι γείτονες του 1 είναι ο 2 και ο 4 και για τον κάθε έναν θα αποθηκεύσουμε τις αποστάσεις τους αφού κανείς από τους δύο δεν υπάρχει στο visited_nodes (γραμμές 21 έως 24).
Το κόστος για τον κάθε κόμβο είναι το άθροισμα 0 (κόστος min_node) συν το κόστος της ακμής.
Για τον 2 είναι: current_weight + graph.distances[(min_node,2)] = 0 + graph.distances[(1,2)] = 2
Για τον 4 είναι: current_weight + graph.distances[(min_node,4)] = 0 + graph.distances[(1,4)] = 1
Ενημερώνουμε το Paths λεξικό και εισάγουμε τους κόμβους 2 και 4 στο σύνολο active_nodes.
«Μηδενίζουμε» το min_node (γραμμή 28) και συνεχίζουμε πάλι από την αρχή της επανάληψης.

Τώρα θα έχουμε:
visited_nodes = {1:0, 2:2, 4:1}
Paths = {2:1, 4:1}
active_nodes = {2, 4}

Ψάχνουμε στο σύνολο active_nodes τον κόμβο εκείνο που έχει το μικρότερο κόστος στο λεξικό visited_nodes. Εδώ είναι ο 4 αφού visited_nodes[2] > visited_nodes[4] οπότε min_node = 4 και current_weight = visited_nodes[4] = 1. Αφαιρούμε τον 4 από το σύνολο active_nodes.

Το graph_edges[min_node] είναι ίσο με {1,2,3,5} από την δομή ακμών.
Οι κόμβοι 3 και 5 δεν υπάρχουν στο λεξικό visited_nodes άρα προστίθενται μαζί με το βάρος τους το οποίο θα είναι:
Για τον 3 είναι: current_weight + graph.distances[(min_node,3)] = 1 + graph.distances[(4,3)] = 2
Για τον 5 είναι: current_weight + graph.distances[(min_node,5)] = 1 + graph.distances[(4,5)] = 4
Για τον κόμβο 2 επειδή υπάρχει στο visited_nodes συγκρίνουμε το προηγούμενο κόστος με το νέο και αν είναι μικρότερο το αντικαθιστούμε.
Νέο κόστος για τον 2:
current_weight + graph.distances[(min_node,2)] = 1 + graph.distances[(4,2)] = 5
είναι μεγαλύτερο του visited_nodes[2] = 2 άρα δεν το αντικαθιστούμε.
Ο κόμβος 1 δεν υπάρχει πια στο σύνολο active_nodes που σημαίνει ότι είναι επεξεργασμένος και δεν ασχολούμαστε πλέον με αυτόν.

Ενημερώνουμε το Paths για τους κόμβους 3 και 5 και τους εισάγουμε στο σύνολο active_nodes.

Οι δομές θα είναι τώρα:
`visited_nodes = {1:0, 2:2, 3:2, 4:1, 5:4}`
`Paths = {2:1, 3:4, 4:1, 5:4}`
`active_nodes = {2, 3, 5}`

Ψάχνουμε στο σύνολο active_nodes τον κόμβο εκείνο που έχει το μικρότερο κόστος στο λεξικό `visited_nodes`. Εδώ είναι ο `2` αφού `visited_nodes[5] > visited_nodes[3] ≥ visited_nodes[2]` οπότε `min_node = 2` και `current_weight = visited_nodes[2] = 2`. Αφαιρούμε τον 2 από το σύνολο `active_nodes.`

Το `graph_edges[min_node]` είναι ίσο με `{1,3,4}` από την δομή ακμών.
Όλοι οι κόμβοι αυτοί υπάρχουν στο λεξικό visited_nodes άρα υπολογίζεται το νέο κόστος αυτών που υπάρχουν στο σύνολο active_nodes (εδώ μόνο ο 3):
Για τον 3 είναι: `current_weight + graph.distances[(min_node,3)] = 2 + graph.distances[(2,3)] = 4`
Επειδή `visited_nodes[3] = 2 < 4` δεν αντικαθίσταται.

Σε αυτήν την επανάληψη δεν συνέβη καμία αλλαγή εκτός του ότι αφαιρέθηκε από το active_nodes ο κόμβος `2.`

Οι δομές θα είναι τώρα:
`visited_nodes = {1:0, 2:2, 3:2, 4:1, 5:4}`
`Paths = {2:1, 3:4, 4:1, 5:4}`
`active_nodes = {3, 5}`

Ανάμεσα τους 3 και 5 το κόστος του 3 είναι το μικρότερο (`visited_nodes[5] > visited_nodes[3]`) άρα `min_node = 3` και `current_weight = 2`. Ο κόμβος 3 αφαιρείται από το `active_nodes`.

Το `graph_edges[min_node]` είναι ίσο με `{2, 4, 5, 6}` από την δομή ακμών.
Οι κόμβοι 2 και 4 δεν υπάρχουν στο σύνολο `active_nodes` άρα δεν αλλάζει τίποτα για αυτούς.
Το νέο κόστος του κόμβου 5 θα είναι:
`current_weight + graph.distances[(min_node,5)] = 2 + graph.distances[(3,5)] = 3`
Αφού `3 < visited_nodes[5] = 4` αντικαθιστούμε το παλιό με το νέο κόστος.
Επίσης αντικαθιστούμε τον κόμβο `Paths[5]` με τον `min_node`.
Για τον κόμβο `6: current_weight + graph.distances[(min_node,6)] = 2 + graph.distances[(3,6)] = 3`
Προσθέτουμε τον `6` στο `active_nodes` και στο `visited_nodes` μαζί με το κόστος του.

Τώρα οι δομές θα είναι:
`visited_nodes = {1:0, 2:2, 3:2, 4:1, 5:3, 6:3}`
`Paths = {2:1, 3:4, 4:1, 5:3, 6:3}`
`active_nodes = {5, 6}`

Τώρα επειδή τα κόστη των 5 και 6 είναι ίσα ο αλγόριθμος θα επιλέξει με βάση το πώς είναι διατεταγμένοι οι κόμβοι στο σύνολο active_nodes. Ας θεωρήσουμε ταξινομημένο σύνολο και για λόγους πληρότητας ο επόμενος κόμβος θα είναι ο 5.
Ο κόμβος 5 αφαιρείται από το σύνολο active_nodes.

Το graph_edges[5] είναι ίσο με {4, 3, 6} από την δομή ακμών. Αλλά μόνο ο 6 είναι μέσα στο active_nodes. Συγκρίνοντας το προηγούμενο με το νέο κόστος δεν αλλάζει τίποτα.
Τελικώς αφού ο ελάχιστος κόμβος και ταυτόχρονα ο τελευταίος (κάτι που δεν είναι απαραίτητο) είναι ο 6, δηλαδή ο τελικός κόμβος ο βρόχος τερματίζει.

Τελικώς οι δομές θα είναι:
visited_nodes = {1:0, 2:2, 3:2, 4:1, 5:3, 6:3}
Paths = {2:1, 3:4, 4:1, 5:3, 6:3}
active_nodes = {6}

Στις τελευταίες γραμμές 29 με 34 συμβαίνουν τα εξής:

current = 6 (τελικός κόμβος)
path = []
βρόχος με τερματισμό όταν current ίσο με 1 (κόμβος πηγή)
path = [6]
current = Paths[6] = 3
path = [6,3]
current = Paths[3] = 4
path = [6,3,4]
current = Paths[4] = 1
τέλος βρόχου
path = [6,3,4,1]

Η συνάρτηση θα επιστρέψει το λεξικό visited_nodes = {1:0, 2:2, 3:2, 4:1, 5:3, 6:3} και την λίστα path[::-1] = [1,4,3,6] που δηλώνει το συντομότερο μονοπάτι από τον κόμβο 1 προς τον κόμβο 6.

Ο λόγος που επιστρέφεται όλο το λεξικό visited_nodes και όχι μόνο το visited[6] , δηλαδή το κόστος του μονοπατιού, είναι ότι μπορεί η εφαρμογή που θα χρησιμοποιηθεί να θέλει και τα ενδιάμεσα κόστη των κόμβων που αποτελούν το συντομότερο αυτό μονοπάτι.

7.5 Αρχείο Κλήσης και Εκτύπωσης του Αλγορίθμου Dijkstra για έναν Γράφο δοσμένο σε αρχείο μορφής λίστας γειτνίασης

Το παρακάτω αρχείο κώδικα ονομάζεται `Run_Dijkstra.py` και χρησιμοποίει όλα τα παραπάνω αρχεία που περιγράφτηκαν σ' αυτήν την ενότητα με σκοπό την εξαγωγή του συντομότερου μονοπατιού ανάμεσα σε δύο κόμβους που εισάγονται από τον χρήστη, για έναν γράφο που εισάγεται από ένα αρχείο μορφής λίστας γειτνίασης.

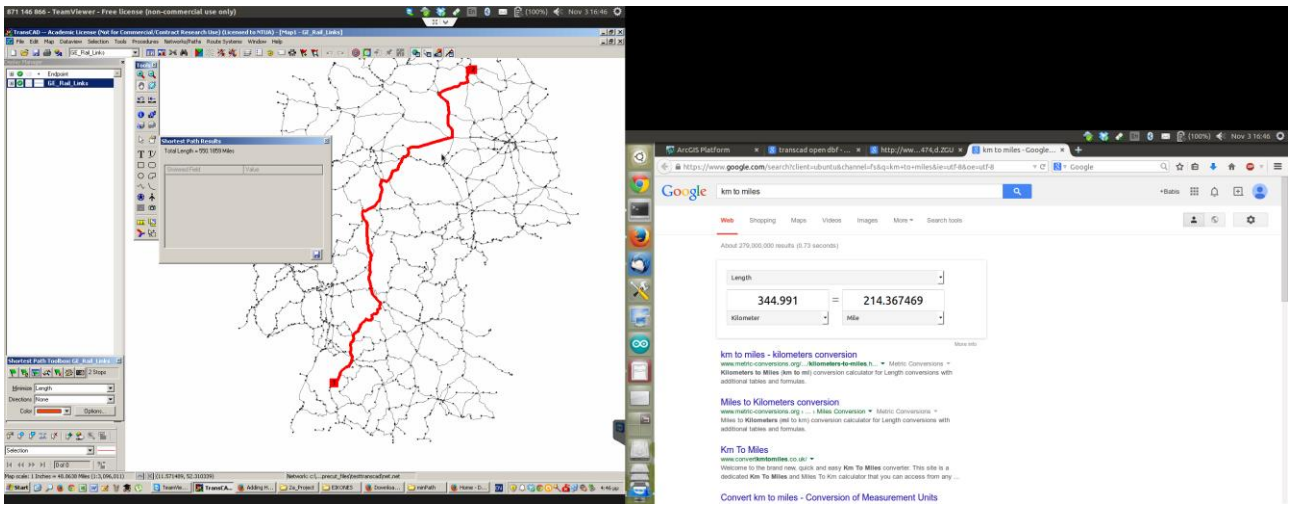
```
""" Run Dijkstra Algorithm on a Graph Structure given by a File
"""
""" Argument 1 must be the file for reading and argument 2 must be
a file for writing results """

1  import sys, Dijkstra, CreateGraphFromFile, Graph_Structure, csv

   # Create a Graph from File
2  MasterGraph =
CreateGraphFromFile.Graph_from_Adj_List_Csv_File()
   # Read File
3  MasterGraph.read_file(sys.argv[1])
   # Create Graph
4  MasterGraph.create_graph()
5  Graph = MasterGraph.Graph

   ## Check if Graph is Connected
6  print(Graph.Connected())

   ## Printing the Graph
7  print('\nThe Number of Nodes is:')
8  print(len(Graph.nodes))
9

   ## Insert start and end node from keyboard
10 print('\nEnter the Source Node')
11 start = int(input()) # Enter source node from keyboard
12 while start not in Graph.nodes:
13      print('\nThe Node does not exist in Graph')
14      print('\nPlease Enter Again the Source Node')
15      start = int(input()) # Enter source node from keyboard

16 print('\nEnter Destination Node')
17 end = int(input()) # Enter destination node from keyboard

18 while end not in Graph.nodes:
19     print('\nThe Node does not exist in Graph')
```

```
20      print('\nPlease Enter Again the Destination Node')
21      end = int(input()) # Enter destination node from keyboard


22 if Graph.Path(start,end):
23      print('There is a Path')
24      ## Run Dijkstra Algorithm
25      Costs, Path = Dijkstra.dijkstra(Graph,start,end)
        # Print Result Shortest Path
26      print('\nThe Shortest Path is',Path)
27      print('\nWith Cost: ',Costs[end])

        ## Write the Path in A Csv File
28      csvfile = open(sys.argv[2], 'w', newline='')
29
writer=csv.writer(csvfile,delimiter=';',quoting=csv.QUOTE_MINIMAL)
30      for node in Path:
31        writer.writerow([node] + [Costs[node]])
```

Όπως θα παρατηρήθηκε οι εντολές στις γραμμές 6, 12, 18 και 22 υπάρχουν για λόγους πληρότητας της εφαρμογής. Για παράδειγμα δεν έχει νόημα να αναζητηθεί το συντομότερο μονοπάτι ανάμεσα σε δύο κόμβους εάν οι δύο κόμβοι δεν συνδέονται ή αν ένας από τους δύο δεν υπάρχει στον γράφο.

7.6 Εφαρμογή του Υλοποιημένου σε Python Αλγορίθμου Dijkstra σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών

Χρησιμοποιήθηκε το πρόγραμμα TransCAD ώστε να υπολογιστεί η συντομότερη διαδρομή ανάμεσα σε δύο κόμβους του σιδηροδρομικού δικτύου της Γερμανίας.



Ο κόμβος που επιλέχθηκε ως αρχή είναι ο 252103 και ως τελικός ο 252078.
Το μήκος που το πρόγραμμα TransCAD έδωσε είναι 214,3683 μίλια δηλαδή 344,991 χιλιόμετρα.
Τρέχοντας το `Run_Dijkstra.py` με ορίσματα το αρχείο `Adjacency_ListGE.csv` και το `Dijkstra_Path.csv` έχουμε ακριβώς το ίδιο αποτέλεσμα.

```
Enter the Source Node
252103

Enter Destination Node
252078
There is a Path

The Shortest Path is [252103, 250553, 251159, 251160, 251161, 251158, 255285, 25
0885, 200685, 250883, 250882, 200720, 250761, 252471, 251178, 250755, 252077, 25
2078]

With Cost:  344.991
```

Και το αρχείο `Dijkstra_Path.csv` είναι:

```
252103;0
250553;30.042
251159;57.775000000000006
251160;65.56400000000001
251161;73.76400000000001
251158;80.59500000000001
255285;82.91600000000001
250885;84.674
200685;89.834
250883;106.02600000000001
250882;111.39000000000001
200720;129.681
250761;191.27200000000002
252471;200.776
251178;241.28900000000002
250755;284.659
252077;318.977
252078;344.991
```

8

Υλοποίηση Αλγορίθμου Dijkstra σε GNU C

Εισαγωγή

Στην ενότητα αυτή παρουσιάζεται ο κώδικας του αρχείου `Dijkstra.c` όπου περιέχει την συνάρτηση υλοποίησης του αλγορίθμου Dijkstra και την κύρια συνάρτηση στην οποία εισάγονται τα δεδομένα του γράφου από ένα αρχείο μορφής λίστας γειτνίασης, ο αρχικός και ο τελικός κόμβος, η κλήσης της συνάρτησης Dijkstra καθώς και η εκτύπωση των αποτελεσμάτων.

Για την υλοποίηση των παρακάτω χρησιμοποιήθηκαν στατικές δομές της προγραμματιστικής γλώσσας C. Οι στατικές δομές προτιμήθηκαν έναντι των δυναμικών για λόγους ταχύτητας και μεγαλύτερης αξιοπιστίας για την εφαρμογή τους σε δεδομένα μεγάλου μεγέθους.

8.1 Κώδικας

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define Max_Nodes 10000
4  #define MaxCost 1000
5  #define Max_Neighbours 7

6  FILE *f;

// DIJKSTRA FUNCTION

7  void Dijkstra(long numofnodes, long source, long destination,
long Nodes[Max_Nodes][Max_Neighbours],double
Costs[Max_Nodes][Max_Neighbours],  double dist[],long prev[]){
8
9   int visited[numofnodes];
10  int
i,j,k,neighbourNode,neighbourIndex,source_index,min_index,count;
11  float neighbourCost,Cost2MinNode,mincost;


    //Find the position of Source Node
12  for(i=0;i<numofnodes;i++)  if (source == Nodes[i][0])
source_index = i;

    //Initialize starting costs from source node to infinity
    //(maximum possible cost)
13  for(i=0;i<numofnodes;i++) dist[i] = MaxCost;

14  dist[source_index] = 0; // The cost of source from itself is
zero (0)

    // Initialize the cost of source neighbours
15  for(j=1;j<Max_Neighbours;j++)
16   if (Costs[source_index][j]!=-1) // Find the neighbours
17    for (i=0;i<numofnodes;i++) // Find the index of every
neighbour
18     if (Nodes[i][0] == Nodes[source_index][j])
19      dist[i] = Costs[source_index][j];



    //Initialize Other Matrices
20  for(i=0;i<numofnodes;i++) visited[i]=0,prev[i] =
Nodes[source_index][0];
```

```
   //dist Matrix is a Matrix that has in first line all Nodes by
name(ID)
   //and in second line all costs from Source Node

21  visited[source_index]=1; // Flag the source node as a visited
node

22  for(count=1;count<numofnodes;count++){

  //Find Minimum Cost Node (The nearest to source not visited
Node)
23  mincost = MaxCost;
24  for(i=0;i<numofnodes;i++)
25   if(dist[i]<mincost && !visited[i])
26    mincost = dist[i], min_index=i;

27  visited[min_index]=1;


28  if (Nodes[min_index][0]==destination) break;

    //Update costs of neighbour nodes of minimum node
29  Cost2MinNode = dist[min_index];
30  for(j=1;j<Max_Neighbours;j++){
31   if (Costs[min_index][j] != -1){
      // Save neighbour node and its cost
32    neighbourCost = Costs[min_index][j],neighbourNode =
Nodes[min_index][j];

    // Find the neighbour's index
33  for(i=0;i<numofnodes;i++) if ( Nodes[i][0] == neighbourNode)
neighbourIndex = i;

    // Compare previous cost with new and replace it if the
previous was greater than the new
34  if((Cost2MinNode+neighbourCost<dist[neighbourIndex]) &&
!visited[neighbourIndex])
35
dist[neighbourIndex]=Cost2MinNode+neighbourCost,prev[neighbourInde
x]= Nodes[min_index][0];}}}}
```

```
// MAIN FUNCTION

int main()
{

 long
Adjacency_List_Nodes[Max_Nodes][Max_Neighbours],Path_list[Max_Node
s],Shortest_Path[Max_Nodes];
 double
Adjacency_List_Costs[Max_Nodes][Max_Neighbours],Distances[Max_Node
s];

 int i,j,k,count_lines=0,total_lines=0, NumofNodes=0;
 long destination_node, source_node, node;
 char ch='c';


 //Initialize Adjacency List
 for(i=0;i<Max_Nodes;i++)
  for(j=0;j<Max_Neighbours;j++)
   Adjacency_List_Nodes[i][j]=-1;
 for(i=0;i<Max_Nodes;i++)
  for(j=0;j<Max_Neighbours;j++)
   Adjacency_List_Costs[i][j]=-1;




 //Scanning from Adjacency List File
 f = fopen ("Adjacency_ListGE.csv", "rt");     //Open file
 // Count the number of lines
 while ((ch =getc(f))!=EOF)
  if( ch== '\n') NumofNodes++;
 printf("The Nodes are:%d\n",NumofNodes);

 rewind(f); // Reset the file pointer

 //Scan the first node
 fscanf(f,"%d",&Adjacency_List_Nodes[0][0]);
 ch = getc(f);
 while (ch !=EOF){

  // When we detect the ; character we know that follows
  // a triad (int,;,float) for neighbour node and its cost
  j=1;
```

```
57   while(ch==';'){
58    fscanf(f,"%d%c%lf",&Adjacency_List_Nodes[count_lines][j],
&ch,&Adjacency_List_Costs[count_lines][j]);
59    ch = getc(f);
60    j++;}

     // When we detect the \n character we know that follows
     // the next node in adjacency list
61   if (ch=='\n'){
62    count_lines++;
63    if (count_lines < NumofNodes)
64     fscanf(f,"%d",&Adjacency_List_Nodes[count_lines][0]);
65    ch = getc(f);}}

66  printf("\n Enter the Source Node:\n");
67  scanf("%d",&source_node);

68  printf("\n Enter the Destination Node:\n");
69  scanf("%d",&destination_node);

    //Dijkstra for Adjacency Lists
70  Dijkstra (NumofNodes, source_node, destination_node,
Adjacency_List_Nodes, Adjacency_List_Costs, Distances, Path_list);

    //Printing Shortest Path
71  printf("\n The Shortest Path is:\n");
72  node = destination_node, k=0;
73  Shortest_Path[k] = node;
    // Backtracking using the Path_list matrix
74  while (node!=source_node){
    //Find Node Index
75  for (i=0;i<NumofNodes;i++)
76   if (node == Adjacency_List_Nodes[i][0])
77    node = Path_list[i], break;
78   k++;
79   Shortest_Path[k] = node;}
80  printf("\n");
81  printf("%d ",Shortest_Path[k]);
82  for (i=k-1; i>=0; i--)
83   printf(", %d ",Shortest_Path[i]);

84  for(i=0;i<NumofNodes;i++)
85   if (Adjacency_List_Nodes[i][0] == destination_node)
86    printf("\nWith cost:%f",Distances[i]);

87 exit(0);}
```

8.2 Επεξήγηση Κώδικα

Στις γραμμές 1 και 2 εισάγονται οι βασικές βιβλιοθήκες της προγραμματιστικής γλώσσας C.

Στις γραμμές 3 έως 5 ορίζονται σταθερές για τα βασικά χαρακτηριστικά των δομών που θα χρησιμοποιηθούν όπως ο μέγιστος αριθμός κόμβων το μέγιστο κόστος και μέγιστος αριθμός γειτόνων που μπορεί να έχει ένας κόμβος στο δίκτυο.
Οι αριθμοί αυτοί επιλέχθηκαν για την υλοποίηση του κώδικα πάνω στο σιδηροδρομικό δίκτυο της Ευρώπης, για το οποίο μετρήθηκε ο συνολικός αριθμός των κόμβων και ο μέγιστος αριθμός γειτόνων.

Στην γραμμή 6 ορίζεται ο δείκτης f που θα χρησιμοποιηθεί για την ανάγνωση και εγγραφή αρχείων.

Στις γραμμές 7 έως 35 υπάρχει η συνάρτηση Dijkstra η οποία υλοποιεί τον αλγόριθμο Dijkstra.
Την συνάρτηση Dijkstra θα την επεξηγήσουμε στην επόμενη υποενότητα στην οποία θα εκτελέσουμε τον κώδικα βήμα προς βήμα με ένα παράδειγμα γράφου.

Στην γραμμή 36 ορίζεται η συνάρτηση main η οποία θα είναι η κύρια συνάρτηση εκτέλεσης του παραπάνω κώδικα.

Στις γραμμές 37 έως 41 ορίζονται οι δομές που θα χρησιμοποιηθούν στην κύρια συνάρτηση.
Οι βασικές δομές είναι οι εξής:

Ένας πίνακας ακεραίων Adjacency_List_Nodes ο οποίος θα περιέχει την λίστα γειτνίασης του γράφου χωρίς όμως τα κόστη, δηλαδή σε κάθε γραμμή το πρώτο στοιχείο θα είναι ο κόμβος και τα υπόλοιπα οι γείτονες του.
Τα κόστη θα υπάρχουν στον πίνακα Adjacency_List_Costs ο οποίος θα έχει ακριβώς το ίδιο μέγεθος με τον Adjacency_List_Nodes αλλά στην θέση του κάθε γείτονα θα περιέχει το αντίστοιχο κόστος.

Ο πίνακας Path_list (γραμμή 37) θα έχει μήκος τον αριθμό των κόμβων που υπάρχουν στον γράφο. Η διάταξη των κόμβων θα καθορίζεται από τον πίνακα Adjacency_List_Nodes στην πρώτη στήλη του. Η κάθε θέση του κόμβου στην πρώτη στήλη του πίνακα Adjacency_List_Nodes θα καθορίζει την θέση του κόμβου στον πίνακα Path_list αλλά το στοιχείο που θα έχει αυτή η θέση θα είναι ο προηγούμενος κόμβος στο μονοπάτι που θα δημιουργείται.
Για παράδειγμα έστω ότι η πρώτη στήλη του πίνακα Adjacency_List_Nodes περιέχει τους κόμβους [Α, Β, Γ] αυτό σημαίνει πώς ο πίνακας Path_list θα έχει μήκος 3 με την πρώτη θέση να αντιστοιχεί στον κόμβο Α την δεύτερη θέση στο κόμβο Β και την τρίτη θέση στον κόμβο Γ. Ο Path_list μπορεί να είναι [Β, Γ, Β]. Αυτό αυτομάτως σημαίνει ότι ο προηγούμενος κόμβος του Α είναι ο Β, ο προηγούμενος κόμβος του Β θα είναι ο Γ και ο προηγούμενος κόμβος του Γ θα είναι ο Β.

Ακριβώς με την ίδια λογική κατασκευάζεται και ο πίνακας Distances (γραμμή 38) αλλά αντί να περιέχει τους προηγούμενους κόμβους θα περιέχει την απόσταση – κόστος από την πηγή. Στο ίδιο παράδειγμα αν η πηγή είναι ο Β κόμβος ο πίνακας Distances μπορεί να είναι [2.3, 0, 3.1] που σημαίνει πώς ο Α απέχει απόσταση 2.3 από τον Β, ο Β απέχει προφανώς 0 από τον εαυτό του και ο Γ απέχει από τον Β απόσταση 3.1.

Ο πίνακας Shortest_Path θα περιέχει το συντομότερο μονοπάτι από τον κόμβο πηγή προς τον τελικό κόμβο και θα κατασκευάζεται με την βοήθεια του Path_list μέσω οπισθοδρόμησης.

Στις γραμμές 42 έως 47 αρχικοποιούνται οι πίνακες Adjacency_List_Nodes και Adjacency_List_Costs θέτοντας όλες τις τιμές στο -1 ώστε να μπορεί να αναγνωρίζεται στην συνέχεια αν ο πίνακας θα θεωρείται «κενός» στην θέση αυτή. Γνωρίζουμε φυσικά πως ο κώδικας δεν θα χρησιμοποιηθεί σε γράφο με αρνητικά βάρη στις ακμές του και πώς κανένα κόμβος δεν θα έχει ID με τιμή -1.

Στις γραμμές 48 έως 65 διαβάζεται το αρχείο της λίστας γειτνίασης το οποία θα έχει την παρακάτω μορφή:

```
200704;250876;5.231;200703;7.274
200705;251711;25.742;251712;7.134
200706;251708;8.988;251707;15.727
200707;251819;6.4559999999999995;251772;16.425
200708;201154;10.767;201155;11.717
200709;250581;11.813;250270;30.983
200710;252685;32.759;252686;4.986
```

Το πρώτο στοιχείο κάθε γραμμής θα είναι ένας κόμβος και τα επόμενα θα είναι ένας γειτονικός κόμβος ακολουθούμενος με την απόσταση του από τον πρώτο κόμβο.

Στις γραμμές 49 έως 52 ο στόχος είναι να βρεθεί το σύνολο τον κόμβων που έχουν γείτονες και κατ' επέκταση ο αριθμός των γραμμών του αρχείου. Με αυτήν την πληροφορία δεν θα δαπανώνται άσκοπες προσπελάσεις στους πίνακες σε περιπτώσεις που ο γράφος είναι μικρός διότι θα γνωρίζουμε το ακριβές μέγεθος που θα πρέπει να έχει ο κάθε πίνακας.
Στις γραμμές 55 έως 65 εισάγονται οι τιμές του αρχείου στους αντίστοιχους πίνακες.

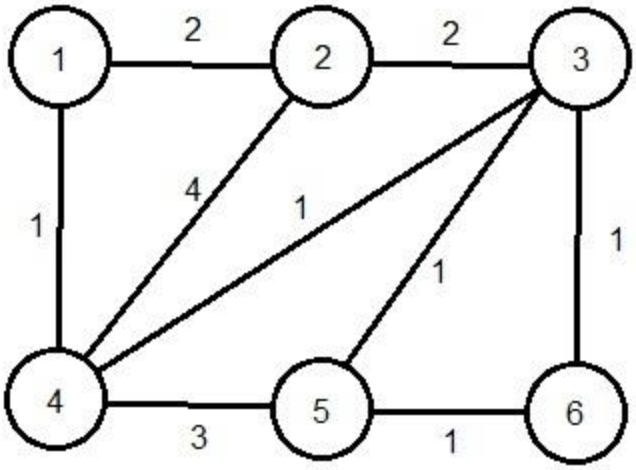
Στις γραμμές 66 έως 69 εισάγονται από τον χρήστη ο αρχικός και ο τελικός κόμβος.

Η συνάρτηση Dijkstra καλείται (γραμμή 70) με παραμέτρους τον αριθμό των κόμβων του γράφου, τον αρχικό και τελικό κόμβο, και τους πίνακες Adjacency_List_Nodes, Adjacency_List_Costs, Distances και Path_list.

Στις γραμμές 71 έως 86 δημιουργείται ο πίνακας Shortest_Path μέσω οπισθοδρόμησης χρησιμοποιώντας τον πίνακα Path_list και εκτυπώνεται το αποτέλεσμα.

8.3 Παράδειγμα εκτέλεσης της συνάρτησης Dijkstra

Έστω ο γράφος παρακάτω γράφος:



Το αρχείο που θα περιέχει την λίστα γειτνίασης του παραπάνω γράφου θα έχει την παρακάτω μορφή:

```
1;2;2.0;4;1.0
2;1;2.0;3;2.0;4;4.0
3;2;2.0;4;1.0;5;1.0;6;1.0
4;1;1.0;2;4.0;3;1.0;5;3.0
5;3;1.0;4;3.0;6;1.0
6;3;1.0;5;1.0
```

Μετά την εκτέλεση των γραμμών 48 έως 65 οι πίνακες `Adjacency_List_Nodes` και `Adjacency_List_Costs` θα έχουν πάρει την παρακάτω μορφή:

`Adjacency_List_Nodes`

1	2	4	-1	-1	-1	-1
2	1	3	4	-1	-1	-1
3	2	4	5	6	-1	-1
4	1	2	3	5	-1	-1
5	3	4	6	-1	-1	-1
6	5	3	-1	-1	-1	-1

`Adjacency_List_Costs`

-1	2.0	1.0	-1	-1	-1	-1
-1	2.0	2.0	4.0	-1	-1	-1
-1	2.0	1.0	1.0	1.0	-1	-1
-1	1.0	4.0	3.0	1.0	-1	-1
-1	1.0	3.0	1.0	-1	-1	-1
-1	1.0	1.0	-1	-1	-1	-1

Έστω ότι ζητείται να βρεθεί το συντομότερο μονοπάτι από τον κόμβο 1 στον κόμβο 6.
Στην συνάρτηση `Dijkstra` λοιπόν θα είναι ως παράμετροι ο αριθμός των κόμβων `numofnodes` (εδώ 6), η πηγή (κόμβος 1) , ο τελικός κόμβος 6, οι παραπάνω πίνακες που μέσα στην συνάρτηση θα ονομάζονται `Nodes` και `Costs` και οι πίνακες `dist` και `prev` που θα διαμορφωθούν μέσω της εκτέλεσης της συνάρτησης.

Αρχικά (γραμμή 12) θα βρεθεί η θέση του κόμβου πηγή στον πίνακα Nodes. Στο παράδειγμα αυτό θα έχει την θέση 0 (η αρίθμηση των στοιχείων στους πίνακες της C ξεκινάει από το 0).

Στην συνέχεια θέτουμε την απόσταση όλων και κόμβων από τον κόμβο πηγή στο άπειρο δηλαδή στο MaxCost (γραμμή 13).
Θέτουμε στην γραμμή 14 την απόσταση του κόμβου πηγή από τον εαυτό του στο μηδέν προφανώς.

Στις γραμμές 15 έως 19 στόχος είναι να βρούμε τα κόστη των γειτόνων του κόμβου πηγή και να τα προσθέσουμε στον πίνακα dist.
Ο πίνακας `dist` θα είναι: `dist = [0 2.0 1000 1.0 1000 1000]`

Στην γραμμή 20 θεωρούμε (ως αρχικοποίηση0 πως όλοι οι κόμβοι έχουν προηγούμενο κόμβο τον κόμβο πηγή, άρα: `prev = [1 1 1 1 1 1]`

Θα χρησιμοποιηθεί ο πίνακας visited όπου θα χρησιμοποιεί την διάταξη των `dist` και `prev` όσων αφορά τους κόμβους αλλά θα έχει την πληροφορία εάν ο κόμβο έχει επεξεργασθεί ή όχι.
Στην γραμμή 21 θέτουμε τον κόμβο πηγή ως επεξεργασμένο αφού έχουμε ενημερώσει την απόσταση των γειτόνων του στον πίνακα `dist`.
`visited = [1 0 0 0 0 0]`

Η βασική επαναληπτική διαδικασία ξεκινάει από την γραμμή 22 και τελειώνει στην γραμμή 35 όπου τελειώνει και η ίδια η συνάρτηση.

Η βασική επανάληψη θα εκτελεστεί n φορές, όπου n ο αριθμός των κόμβων, στην χειρότερη δυνατή περίπτωση, διαφορετικά θα διακοπεί μόλις εντοπίσει τον τελικό κόμβο ως τον κόμβο με την ελάχιστη απόσταση που δεν είναι επεξεργασμένος (γραμμή 28).

Σε κάθε βρόχο της επανάληψης θα γίνονται τα εξής:

Αρχικά θα εντοπίζεται ο κόμβος που δεν είναι επεξεργασμένος και έχει την μικρότερη απόσταση στον πίνακα `dist` (γραμμές 23 έως 26).
Αφού εντοπιστεί μαρκάρεται ως επεξεργασμένος (γραμμή 27) και στην συνέχεια, αν δεν είναι ο τελικός κόμβος, θα προσπελαστούν οι γείτονες του (γραμμές 29 έως 33).
Αν για κάθε γείτονα το νέο υπολογισμένο κόστος είναι μικρότερο από το κόστος που είναι σημειωμένο στον πίνακα `dist` τότε το αντικαθιστά και επίσης σημειώνεται στον πίνακα `prev` ότι ο προηγούμενος κόμβος θα είναι ο κόμβος που εξετάζονται οι γείτονες του.

Στην πρώτη επανάληψη επομένως θα έχουμε ως κόμβο με το ελάχιστο κόστος που δεν είναι επεξεργασμένος τον κόμβο 4.

Στον πίνακα `Nodes` εντοπίζεται ο κόμβος 1 ως γείτονας αλλά επειδή είναι επεξεργασμένος δεν επηρεάζεται. Ο επόμενος γείτονας του 4 είναι ο 2 και επειδή το νέο κόστος του θα είναι:
1 + 4 = 5 > 2 δεν αντικαθιστά το κόστος στον πίνακα `dist`. Ο επόμενος γείτονας του 4 είναι ο 3 και επειδή το νέο κόστος του 1 + 1 = 2 είναι μικρότερο του 1000 αντικαθίσταται στον πίνακα `dist`. Τελευταίος γείτονας του 4 είναι ο 5 και ομοίως με τον 3 το κόστος του 1 + 3 = 4 < 1000 αντικαθίσταται στον πίνακα `dist`.

Οι πίνακες `visited, dist` και `prev` θα είναι τώρα:

```
visited = [1 0 0 1 0 0]
dist = [0 2 2 1 4 1000]
prev =[1 1 4 1 4 1]
```

Στην δεύτερη επανάληψη ο επόμενος κόμβος με το ελάχιστο κόστος που δεν είναι επεξεργασμένος είναι ο 2.
Οι γείτονες του 2 είναι οι 1, 3, 4 σύμφωνα με τον πίνακα `Nodes`. Ο 1 και ο 4 είναι επεξεργασμένοι κόμβοι άρα δεν αλλάζει τίποτα για αυτούς. Ο κόμβος 3 έχει νέο κόστος 2 + 2 = 4 αλλά στον πίνακα `dist` έχει 2 άρα δεν αλλάζει τίποτα για αυτόν επίσης.

Οι πίνακες `visited, dist` και `prev` θα είναι τώρα:

```
visited = [1 1 0 1 0 0]
dist = [0 2 2 1 4 1000]
prev =[1 1 4 1 4 1]
```

Στην τρίτη επανάληψη ο επόμενος κόμβος με το ελάχιστο κόστος που δεν είναι επεξεργασμένος είναι ο 3.
Οι γείτονες του 3 είναι οι 2, 4, 5, 6 σύμφωνα με τον πίνακα `Nodes`. Ο κόμβος 2 και ο 4 είναι επεξεργασμένοι κόμβοι άρα δεν αλλάζει τίποτα για αυτούς. Ο κόμβος 5 έχει νέο κόστος 2 + 1 = 3 και στον πίνακα `dist` έχει 4 άρα αλλάζει και στον πίνακα `prev` παίρνει ως προηγούμενο του τον κόμβο 3. Ο κόμβος 6 έχει νέο κόστος 2 + 1 = 3 και στον πίνακα `dist` έχει 1000 άρα αλλάζει και στον πίνακα `prev` παίρνει ως προηγούμενο του τον κόμβο 3.

Οι πίνακες `visited, dist` και `prev` θα είναι τώρα:

```
visited = [1 1 1 1 0 0]
dist = [0 2 2 1 3 3]
prev =[1 1 4 1 3 3]
```

Στην τέταρτη επανάληψη ο επόμενος κόμβος με το ελάχιστο κόστος που δεν είναι επεξεργασμένος είναι ο 5.
Οι γείτονες του 5 είναι οι 3, 4, 6 σύμφωνα με τον πίνακα `Nodes`. Ο κόμβος 3 και ο 4 είναι επεξεργασμένοι κόμβοι άρα δεν αλλάζει τίποτα για αυτούς. Ο κόμβος 6 έχει νέο κόστος 3 + 1 = 4 αλλά στον πίνακα `dist` έχει 3 άρα δεν αλλάζει κάτι για αυτόν επίσης.

Οι πίνακες `visited, dist` και `prev` θα είναι τώρα:

```
visited = [1 1 1 1 1 0]
dist = [0 2 2 1 3 3]
prev =[1 1 4 1 3 3]
```

Τέλος στην πέμπτη επανάληψη ο κόμβος που επιλέγεται είναι ο 6 και επειδή είναι και ο τελικός κόμβος ο βρόχος τερματίζει.

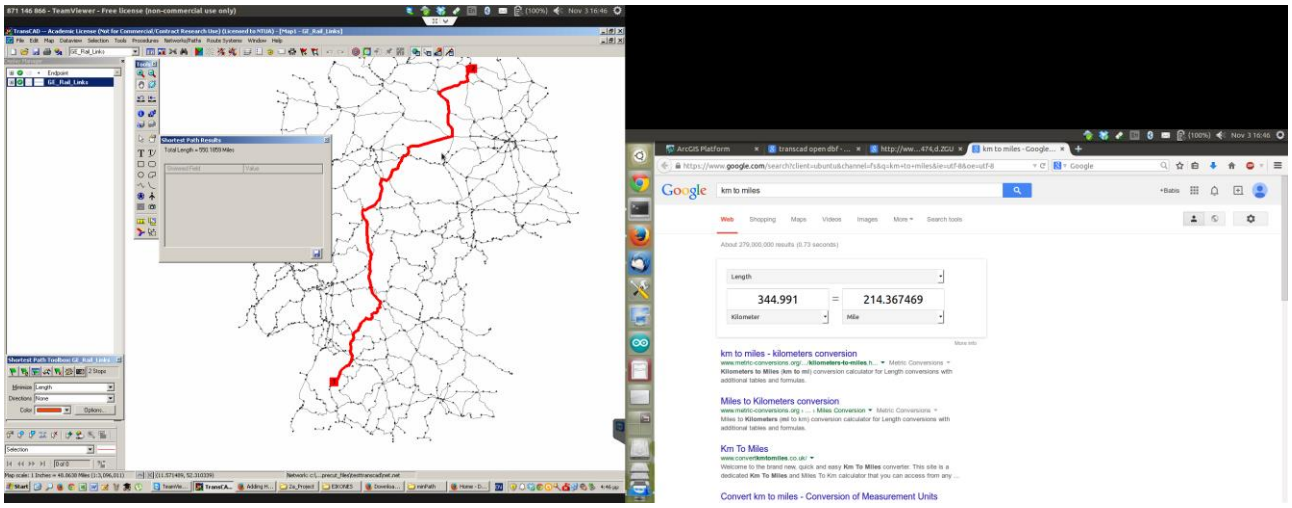
Μέσω της συνάρτησης `Dijkstra` οι πίνακες `dist` και `prev` περνάνε στην κύρια συνάρτηση ως οι πίνακες `Distances` και `Path_list`, επομένως μπορεί να σημειωθεί το συντομότερο μονοπάτι στον πίνακα `Shortest_Path`.

Αρχίζοντας από τον τελικό κόμβο (γραμμή 72) θα χρησιμοποιείται ο πίνακας `Path_list` για να σημειώνεται ο προηγούμενος κάθε φορά κόμβο και στην συνέχεια ο προηγούμενος μέχρι να εντοπιστεί ο κόμβος πηγή (γραμμή 74 έως 79)

Ο πίνακας `Shortest_Path` θα είναι: `[6 3 4 1]` και θα εκτυπώνεται αντίστροφα.
Το κόστος του μονοπατιού θα βρίσκεται από τον πίνακα `Distances` ως το κόστος του τελικού κόμβου από την πηγή.

8.4 Εφαρμογή του Υλοποιημένου σε GNU C Αλγορίθμου Dijkstra σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών με έμφαση στην ταχύτητα λειτουργίας

Χρησιμοποιήθηκε το πρόγραμμα TransCAD ώστε να υπολογιστεί η συντομότερη διαδρομή ανάμεσα σε δύο κόμβους του σιδηροδρομικού δικτύου της Γερμανίας.



Ο κόμβος που επιλέχθηκε ως αρχή είναι ο 252103 και ως τελικός ο 252078.
Το μήκος που το πρόγραμμα TransCAD έδωσε είναι 214,3683 μίλια δηλαδή 344,991 χιλιόμετρα.
Εκτελώντας τον κώδικα Dijkstra.c και δίνοντας τους ίδιους κόμβους ως είσοδο έχουμε το παρακάτω αποτέλεσμα:

```
The Nodes are:1447

 Enter the Source Node:
252103

 Enter the Destination Node:
252078

 The Shortest Path is:

252103 , 250553 , 251159 , 251160 , 251161 , 251158 , 255285 , 250885 , 200685 ,
 250883 , 250882 , 200720 , 250761 , 252471 , 251178 , 250755 , 252077 , 252078

With cost:344.990990
--------------------------------
```

Βλέπουμε πως είναι το αποτέλεσμα είναι ίδιο και μάλιστα η προγραμματιστική γλώσσα C μας δίνει ακριβέστερη λύση καθώς έχουμε χρησιμοποιήσει double δομή για την αποθήκευση του κόστους της κάθε ακμής.

9

Υλοποίηση Αλγορίθμου Κ - Συντομότερων Μονοπατιών σε Python

9.1 Αρχείο εκτέλεσης του Αλγορίθμου Κ - Συντομότερων Μονοπατιών

Το παρακάτω αρχείο υλοποιεί το αλγόριθμο Κ συντομότερων μονοπατιών ως συνάρτηση που δέχεται ως όρισμα έναν γράφο, τους κόμβους αρχή και τέλος και το Κ. Η δομή του γράφου θα είναι η δομή που χρησιμοποιήθηκε και για την εκτέλεση του αλγορίθμου Dijkstra σε Python στις προηγούμενες ενότητες. Το αρχείο ονομάζεται `KShortestPaths.py`

```
""" K best Shortest Paths Algorithm: Finds the K Shortest Paths
from a source to destination in a graph """

def KShortestPaths(Graph,start,end,K):

    #Create a Class for Paths
      class Algorithm_Path:

          def __init__(self,path,cost):

              self.path = path # path must be a list
              self.cost = cost # cost is a float number

          def __str__(self):

              return 'The path is: ' + str(self.path) + ' \n' +
'With Cost: ' + str(self.cost)
```

```
        def expand(self,node,cost):

            self.path.append(node)
            self.cost = self.cost + cost

    # count is a dictionary that keeps tracking how many times the algorithm has visited a curtain node
    count = {x : 0 for x in Graph.nodes} # Count = 0 for all nodes in Graph at the start

    # All_Paths is a list of tuples: Every tuple is a pair of a list and an integer
    # The list of a tuple represents a path and the integer represents the cost so far on the path
    All_Paths = [] # Initialize the All_Paths List

    # ShortestPaths is a list that saves every minimum path from source to destination
    ShortestPaths = [] # Initialize the ShortestPaths list
    # Path are tuples (list, int) that are the elements of the All_Paths list
    Path = Algorithm_Path([start],0)
    All_Paths.append(Path)
    # e.g. All_Paths = [([1,2],3),([1],5),([2,3],2),([4,6,7,8,1],1)]
    ## The algorithm ends when All_Paths is empty or we have visited the destination K times
    while All_Paths and count[end] < K:

        ## Search for the minimum existed Path in All_Paths
        Minimum_Path = All_Paths[0]
        for Path in All_Paths:
            if Path.cost < Minimum_Path.cost:
                Minimum_Path = Path

        # Save temporarily the cost of minimum path
        tempcost = Minimum_Path.cost
        # Remove the minimum path from the main list
        All_Paths.remove(Minimum_Path)
        # Save the last node of the minimum path
        currentnode = Minimum_Path.path[-1]
        # Plus one visit of the last node of minimum path
        count[currentnode] = count[currentnode] + 1

        ## If the last node was the destination then we save the Path as a Shortest Path
```

```
25          if currentnode == end:
26              ShortestPaths.append(Minimum_Path)

27          else: ## Create the new paths
28               currentpath = Minimum_Path.path # Save the minimum
path
29               for nearnode in Graph.edges[currentnode]: # Search
every neighbour of the last node of minimum path
30                   if nearnode not in currentpath: #There is a
possibility that we cross an edge twice in the same path
                     # Create the new path as an extension of the
minimum path
31                      New_Path = currentpath[:]
32                      New_Path.append(nearnode)
                     # Add the new path to All_Paths
33                      Path = Algorithm_Path(New_Path, tempcost +
Graph.distances[(currentnode, nearnode)])
34                      All_Paths.append(Path)
                     #break

35      return ShortestPaths
```

Στις γραμμές 2 έως 10 ορίζεται μια κλάση ως μονοπάτι που περιέχει μια λίστα και έναν αριθμό. Η λίστα παριστά το μονοπάτι, δηλαδή την ακολουθία κόμβων και ο αριθμός το συνολικό κόστος του μονοπατιού αυτού.

Στην γραμμή 11 ορίζεται το λεξικό count το οποίο θα χρησιμοποιηθεί για να ενημερώνει τους μετρητές των κόμβων, οι οποίοι θα διατηρούν το πόσες φορές έχει ο αλγόριθμος αποθηκεύσει ελάχιστο μονοπάτι με τελικό τον κόμβο αυτόν.

Η λίστα `All_Paths` (γραμμή 12) θα είναι το σύνολο μέσα στο οποίο θα αποθηκεύονται όλα τα μονοπάτια και θα αναζητείται αυτό με το ελάχιστο κόστος.

Η λίστα `ShortestPaths` (γραμμή 13) θα είναι το σύνολο των ελάχιστων μονοπατιών που έχουν ως τελικό κόμβο, τον κόμβο προορισμού. Προφανώς όταν ο παραπάνω αλγόριθμος τερματίσει η λίστα `ShortestPaths` θα έχει μέγεθος K αφού αναζητούνται τα πρώτα K καλύτερα μονοπάτια.

Στις γραμμές 17 έως 20 αναζητείται το μονοπάτι μέσα στο `All_Paths` με το ελάχιστο κόστος.

Όταν βρεθεί το μονοπάτι με το ελάχιστο κόστος αφαιρείται από το `All_Paths` (γραμμή 22) και ο μετρητής του τελικού του κόμβου αυξάνεται κατά 1 (γραμμή 24)

Αν ο τελικός κόμβος του μονοπατιού είναι ο κόμβος προορισμού τότε το μονοπάτι αποθηκεύεται στο `ShortestPaths` (γραμμές 25 και 26) διαφορετικά επεκτείνεται με βάση τους γείτονες του τελικού κόμβου. Τα νέα μονοπάτια αποθηκεύονται στο `All_Paths` (γραμμές 28 έως 34).

Η συνθήκη στην γραμμή 30 έχει προστεθεί για να αποτρέψει τους βρόχους μέσα σε ένα μονοπάτι, δηλαδή να έχει το μονοπάτι έναν κόμβο παραπάνω από μία φορά.

Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι να βρεθούν και τα K μονοπάτια ή η λίστα `All_Paths` γίνει κενή που σημαίνει πως δεν υπάρχουν άλλα μονοπάτια ανάμεσα στους δύο κόμβους που δίνονται (γραμμή 16).

9.2 Αρχείο Κλήσης και Εκτύπωσης του Αλγορίθμου K Συντομότερων Μονοπατιών για έναν Γράφο δοσμένο σε αρχείο μορφής Λίστας Γειτνίασης

Το παρακάτω αρχείο ονομάζεται `Run_K-Best_Algorithm.py` και δέχεται ως όρισμα ένα αρχείο μορφής λίστας γειτνίασης ενός γράφου.

```
""" Run K-Best Algorithm on a Graph Structure given by a File """
""" Argument 1 must be the file for reading """

1  import sys, KShortestPaths, CreateGraphFromFile,
Graph_Structure, csv

   # Create a Graph from File
2  MasterGraph =
CreateGraphFromFile.Graph_from_Adj_List_Csv_File()
   # Read File
3  MasterGraph.read_file(sys.argv[1])
   # Create Graph
4  MasterGraph.create_graph()
5  Graph = MasterGraph.Graph

   ## Check if Graph is Connected
6  print(Graph.Connected())

   ## Printing the Graph
7  print('\nThe Number of Nodes is:')
8  print(len(Graph.nodes))
9

   ## Insert start and end node from keyboard
10 print('\nEnter the Source Node')
11 start = int(input()) # Enter source node from keyboard
12 while start not in Graph.nodes:
13     print('\nThe Node does not exist in Graph')
14     print('\nPlease Enter Again the Source Node')
15     start = int(input()) # Enter source node from keyboard
```

```
16 print('\nEnter Destination Node')
17 end = int(input()) # Enter destination node from keyboard

18 while end not in Graph.nodes:
19     print('\nThe Node does not exist in Graph')
20     print('\nPlease Enter Again the Destination Node')
21     end = int(input()) # Enter destination node from keyboard


22 if Graph.Path(start,end):
23     print('There is a Path')

24     print('How many Paths do you want to find? Please enter the
K number')
25     K = input()

       ## Run K-Shortest Paths Algorithm
26     print('\nNow Printing the K Paths')
27     ShortestPaths=
KShortestPaths.KShortestPaths(Graph,start,end,K)
28     print('\n\n The Shortest Paths are:\n')
29     for i in range(0,K):
30         print('Path ' + str(i+1) + ' :' ,
ShortestPaths[i].path)
31         print('With Cost: ', ShortestPaths[i].cost)

32 else: print ('There is not Path between the ' + str(start) + '
node and the ' + str(end) + ' node.')
```

Στις γραμμές 2 έως 5 ο κώδικας δημιουργεί τον γράφο χρησιμοποιώντας τα αρχεία που περιγράφηκαν στην ενότητα υλοποίησης του αλγορίθμου Dijkstra.

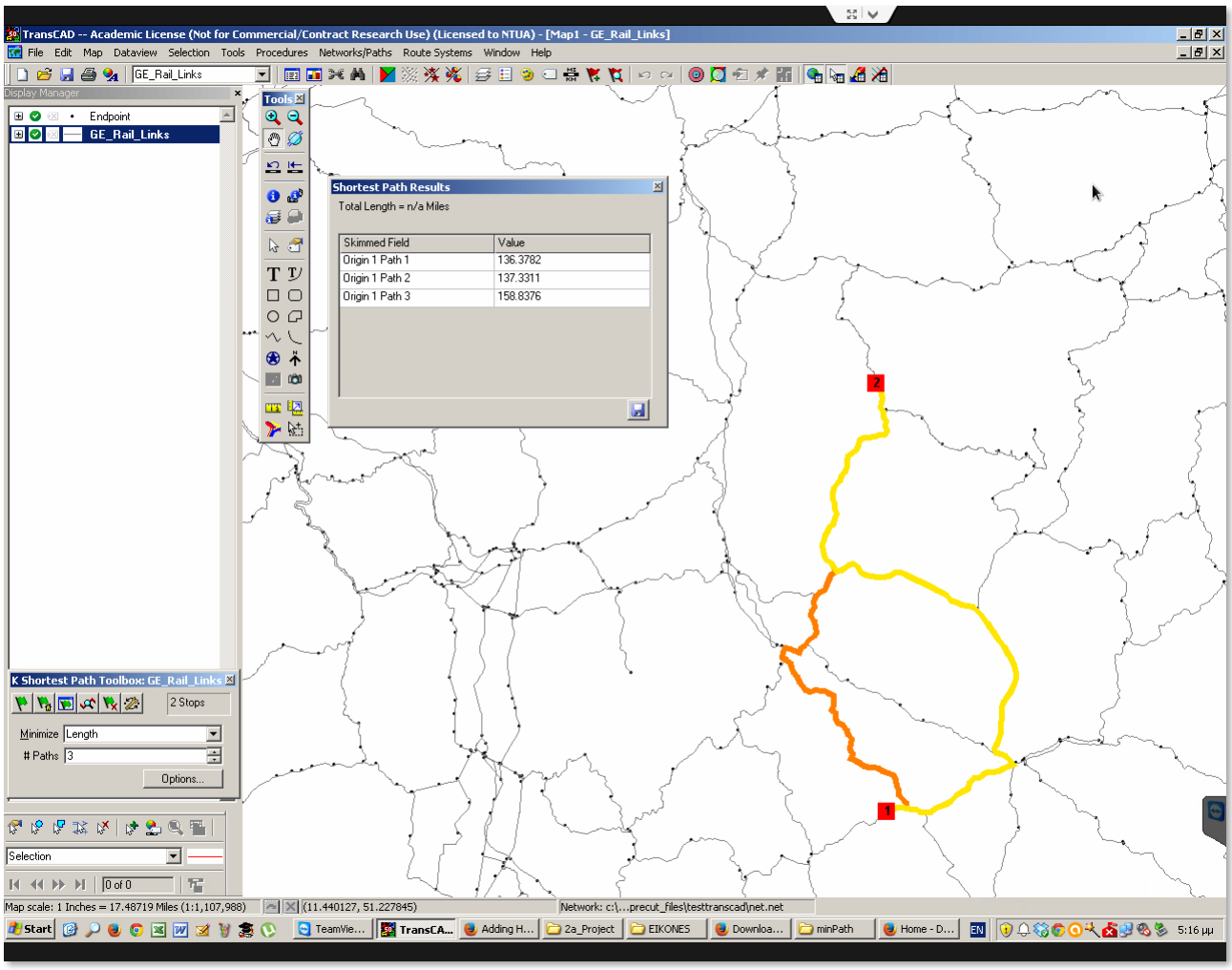
Οι γραμμές 6 έως 22 αποτελούν εντολές ελέγχου για το αν ο γράφος είναι συνδεδεμένος, αν υπάρχουν οι κόμβοι που δίνονται στον γράφο και εάν υπάρχει τουλάχιστον ένα μονοπάτι ανάμεσα στους δύο δοσμένους κόμβους, και εντολές εισόδου μέσω των οποίων ο χρήστης θα δώσει τους κόμβους αρχή και προορισμός.

Στις γραμμές 25 και 27 εισάγεται επίσης από τον χρήστη ο επιθυμητός αριθμός μονοπατιών και καλείται η συνάρτηση `KShortestPaths` με ορίσματα τον γράφο που δημιουργήθηκε από το αρχείο λίστας γειτνίασης, τους κόμβους αρχή και προορισμού και τον αριθμό K.

Τέλος (γραμμές 28 έως 31) εκτυπώνονται στην οθόνη του χρήστη τα K συντομότερα μονοπάτια.

9.3 Εφαρμογή του Υλοποιημένου σε Python Αλγορίθμου K-Best σε πραγματικά Ευρωπαϊκά Δίκτυα Μεταφορών

Χρησιμοποιήθηκε το πρόγραμμα TransCAD ώστε να υπολογιστούν οι 3 συντομότερες διαδρομές ανάμεσα σε δύο κόμβους του σιδηροδρομικού δικτύου της Γερμανίας.



Ο κόμβος που επιλέχθηκε ως αρχή είναι ο 252498 και ως τελικός ο 252110.
Το μήκος που το πρόγραμμα TransCAD έδωσε είναι 136,3782 μίλια δηλαδή 219,4794 χιλιόμετρα για το πρώτο καλύτερο μονοπάτι, 137,3311 μίλια (221,01298 χιλιόμετρα) για το δεύτερο καλύτερο μονοπάτι και 158,8386 μίλια (255,6259 χιλιόμετρα) για το τρίτο καλύτερο μονοπάτι.
Εκτελώντας τον κώδικα `Run_K-Best_Algorithm.py` και δίνοντας τους ίδιους κόμβους ως είσοδο έχουμε το παρακάτω αποτέλεσμα:

```
How many Paths do you want to find? Please enter the K number
3

Now Calculating the K Paths
K-Best for 3 Paths run in: 10.806618928909302


 The Shortest Paths are:

Path 1 : [252498, 252497, 202160, 250921, 250922, 250920, 250288, 250287, 251733
, 252608, 255267, 252607, 252601, 252111, 252110]
With Cost:  219.47900000000004
Path 2 : [252498, 252497, 202160, 250921, 250922, 250920, 250288, 250287, 251733
, 252608, 255267, 252607, 252601, 252594, 252111, 252110]
With Cost:  221.01300000000003
Path 3 : [252498, 252497, 255253, 252617, 252618, 252619, 252703, 251748, 251747
, 200725, 251746, 251745, 200726, 251744, 202159, 200724, 251743, 251742, 251741
, 252112, 252113, 252114, 252115, 251733, 252608, 255267, 252607, 252601, 252111
, 252110]
With Cost:  255.62600000000003
```

Βλέπουμε πως είναι το αποτέλεσμα είναι ίδιο για όλα τα μονοπάτια με απόκλιση τάξης του εκατοστού του μέτρου.

10

Συμπεράσματα - Προτάσεις για συνέχιση της εργασίας

10.1 Συμπεράσματα

Τόσο από την μελέτη που έγινε στην εργασία αυτή του προσκηνίου της έρευνας και της τεχνολογικής ανάπτυξης στον τομέα των αλγορίθμων εύρεσης βέλτιστων διαδρομών σε γράφους (δίκτυα μεταφορών), όσο και από τα αποτελέσματα των δοκιμών της παρούσας εργασίας πρόεκυψαν τα ακόλουθα συμπεράσματα:

1. Για να πετύχει κανείς το μεγαλύτερο εύρος εφαρμοσιμότητας των αλγορίθμων δρομολόγησης, είναι σκόπιμο να χρησιμοποιήσει τον αλγόριθμο Dijkstra ο οποίος συνδυάζει με τον καλύτερο τρόπο γενικότητα χρήσης και υψηλή επίδοση. Είναι δυνατόν να χρησιμοποιηθούν άλλοι αλγόριθμοι για την επίλυσης του ίδιου προβλήματος με ακόμα καλύτερη επίδοση (όπως ο αλγόριθμος A*) αλλά οι αλγόριθμοι αυτοί υστερούν σε γενικότητα. Για τον λόγο αυτό υλοποιήθηκε ο αλγόριθμος Dijkstra τόσο στην προγραμματιστική γλώσσα Python όσο και στην C με τρόπο που θα εξηγηθεί στα επόμενα.

2. Ενώ ο αλγόριθμος Dijkstra δίνει πολύ καλά αποτελέσματα στην πλειοψηφία των γενικού τύπου προβλημάτων εύρεσης βέλτιστων διαδρομών (και των περισσότερων προβλημάτων που παρουσιάζονται στην περίπτωση των δικτύων μεταφορών) οι αλγόριθμοι εύρεσης των K καλύτερων διαδρομών δίνουν ελάχιστα διαφέρουσες εναλλακτικές διαδρομές και επομένως δεν βοηθούν στο πρόβλημα της εύρεσης καθοριστικά διαφορετικών λύσεων, κάτι που οι ειδικοί στις μεταφορές επιζητούν. Για τον λόγο αυτό ο αλγόριθμος K-BEST υλοποιήθηκε μόνο σε Python ώστε να γίνουν οι σχετικές δοκιμές. Είναι σκόπιμο να αναζητηθεί ειδική μορφή του αλγορίθμου η οποία έχει την δυνατότητα να βρίσκει καθοριστικά διαφορετικές λύσεις (οι ειδικοί στις μεταφορές δεν έχουν ακόμα ποσοτικοποιήσεις την έννοια των "καθοριστικά διαφορετικών" διαδρομών).

3. Για την υλοποίηση των αλγορίθμων χρησιμοποιήθηκε καταρχήν η προγραμματιστική γλωσσά Python, επειδή μπορεί να θεωρηθεί ως ένα "σύστημα ταχείας ανάπτυξης πρωτοτύπων" (rapid prototyping system) και είναι ευρέως διαδεδομένη. Η προγραμματιστική γλώσσα Python, με τις υψηλού επιπέδου δομές δεδομένων που διαθέτει, διευκολύνει τον προγραμματιστή στην υλοποίηση των συγκεκριμένων αλγορίθμων και την απλούστευση της μορφής των σχετικών προγραμμάτων.

4. Δεδομένου ότι από την διερεύνηση που έγινε κρίθηκε ότι ο αλγόριθμος Dijkstra είναι ο προτιμητέος, ως γενικού τύπου και ταυτόχρονα ιδιαίτερα αποδοτικός αλγόριθμος για τις εφαρμογές των μεταφορικών δικτύων, επιζητήθηκε πιο αποδοτικός τρόπος υλοποίησης του. Για αυτόν τον λόγο έγινε και ειδική υλοποίηση του αλγορίθμου στην προγραμματιστική γλώσσα C, με κριτήριο την αποδοτική σε χρόνο εκτέλεση των σχετικών προγραμμάτων. Αντιθέτως, επειδή κρίθηκε ότι η χρήση του K-Best αλγορίθμου δεν έδωσε τα επιζητούμενα, από τους ειδικούς στις μεταφορές, αποτελέσματα δεν έγινε η υλοποίηση του στην γλώσσα C. Η κατεύθυνση της έρευνας και ανάπτυξης πρέπει να στραφεί προς τον ακριβή καθορισμό της έννοιας "καθοριστικά διαφορετικών" διαδρομών και, ενδεχομένως, προς την ανάπτυξη νέων μορφών του ιδίου ή άλλου αλγορίθμου που θα καλύψει σε μεγαλύτερο βαθμό τις ανάγκες των ειδικών στις μεταφορές.

5. Από την εξαντλητική μελέτη των διαφορετικών μορφών υλοποίησης του αλγορίθμου Dijkstra που αναφέρονται στην σύγχρονη βιβλιογραφία, έχει φανεί ότι για να εκτελεστεί γρήγορα ο αλγόριθμος χρειάζεται:

α) ο αλγόριθμος έχει ανάγκη από εργαλεία λογισμικού, τα οποία στηρίζονται στον γρήγορο χειρισμό δεδομένων της μορφής "κλειδί - τιμή" κα όχι στους κλασσικούς πίνακες των σχεσιακών βάσεων δεδομένων

β) να βρεθεί τρόπος να εκτελεστεί αποδοτικά η πλέον χρονοβόρα ενέργεια του αλγορίθμου, η οποία είναι η εύρεση, μέσα από ένα σύνολο, ενός ζεύγους "κλειδί - τιμή" με το κλειδί να έχει την ελάχιστη τιμή. Οι συνιστώμενοι στην βιβλιογραφία τρόποι στηρίζονται στην χρήση δομών τύπου σωρού (heap) με πλέον αποδοτική την χρήση του σωρού Fibonacci (Fibonacci Heap), όμως σε σύγχρονες εμπορικές εφαρμογές έχουν αρχίσει να χρησιμοποιούνται για τον σκοπό αυτό δομές ισορροπημένων ν - αδικών δέντρων (B Trees ,B+ trees).Στην παρούσα εργασία όμως δεν έγινε σχετική υλοποίηση, λόγω της πολυπλοκότητας των δομών αυτών.

6. Η δοκιμή των αλγορίθμων έγινε πάνω σε διάφορα υποδίκτυα των μεταφορικών δικτύων που χρησιμοποιεί η Γενική Διεύθυνση για μεταφορές και ενέργεια (DG TREN) της ευρωπαϊκής κοινότητας. Για την επαλήθευση της ορθότητας των αποτελεσμάτων χρησιμοποιήθηκε ένα αντίστοιχο εργαλείο εύρεσης βέλτιστων διαδρομών της κοινότητας το οποίο έδωσε ακριβώς τα ίδια αποτελέσματα (στο σημείο αυτό αξίζει να σημειωθεί ότι ένας από τους κύριους σκοπούς της εργασίας αυτής είναι η παροχή της δυνατότητας εφαρμογής παραλλαγών του αλγορίθμου Dijkstra για την επίλυση ειδικών κατηγοριών προβλημάτων με υψηλή αποδοτικότητα, κάτι που απαιτεί παρέμβαση στην εσωτερική δομή του αλγορίθμου)

10.2 Μελλοντικές επεκτάσεις

Φυσική συνέχεια της παρούσας εργασίας και, επομένως, προτάσεις για την συνέχιση της εργασίας, αποτελούν τα ακόλουθα:

1. Εξέταση της χρήσης δομών B trees και B+ trees για την αποτελεσματική εκτέλεση της εύρεσης του στοιχείου κλειδιού με την ελάχιστη τιμή μέσα σε ένα σύνολο.

2. Επέκταση του υλοποιηθέντος αλγορίθμου ώστε αυτός να μπορεί να χρησιμοποιεί γενικευμένο κόστος, το οποίο μπορεί να προκύπτει ως συνάρτηση περισσοτέρων μεταβλητών. Οι νέες μορφές του αλγορίθμου που θα προκύψουν πρέπει να υλοποιηθούν τόσο σε Python όσο και σε C.

3. Όπως ήδη αναφέρθηκε, για την εύρεση των K καλύτερων διαδρομών πρέπει να αναζητηθούν μορφές του σχετικού αλγορίθμου, οι οποίες θα μπορούν να χρησιμοποιηθούν για την εύρεση καθοριστικά διαφορετικών λύσεων. Για τον σκοπό αυτό θα χρειαστεί η συνεργασία μηχανικών υπολογιστών και μηχανικών μεταφορών.

Βιβλιογραφία

[MRK04] A.R. Mashaghi, A. Ramezanpour, V. Karimipour (2004). "Investigation of a protein complex network". European Physical Journal B41(1): 113–121.

[RK11] Rosen, Kenneth H. Discrete mathematics and its applications(7th ed.). New York: McGraw-Hill.

[ADG+10] Abraham, Ittai; Delling, Daniel; Goldberg, Andrew V.; Werneck, Renato F.research.microsoft.com/pubs/142356/HL-TR.pdf "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks". Symposium on Experimental Algorithms, pages 230-241, 2011.

[SP09] Sanders, Peter (March 23, 2009). "Fast route planning". Google Tech Talk.

[CD96] Chen, Danny Z. (December 1996). "Developing algorithms and software for geometric path planning problems".

[CLR09] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest.Introduction to Algorithms

[CHE70] Cheney, C.J. (November 1970). "A Nonrecursive List Compacting Algorithm".Communications of the ACM 13 (11): 677–678.

[ES11] Even, Shimon (2011), Graph Algorithms (2nd ed.), Cambridge University Press, pp. 46–48,ISBN 978-0-521-73653-4.

[DT10] Dijkstra, Edsger; Thomas J. Misa, Editor (August 2010). "An Interview with Edsger W. Dijkstra".Communications of the ACM 53 (8): 41–47.

[BG00] Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm". Digraphs: Theory, Algorithms and Applications (First ed.). ISBN 978-1-84800-997-4.

[SED02] Sedgewick, Robert (2002). "Section 21.7: Negative Edge Weights". Algorithms in Java (3rd ed.). ISBN 0-201-36121-3.

[KT06] Kleinberg, Jon; Tardos, Éva (2006). Algorithm Design. New York: Pearson Education, Inc.

[DSS+09] Delling, D. and Sanders, P. and Schultes, D. and Wagner, D. (2009). "Engineering route planning algorithms". Algorithmics of large and complex networks. Springer. pp. 117–139.

[ZC09] Zeng, W.; Church, R. L. (2009). "Finding shortest paths on real road networks: the case for A*".International Journal of Geographical Information Science 23.

[HNR68] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107.

[RN95] Russell, Stuart; Norvig, Peter (2003) [1995]. Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall. pp. 97–104.][Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.

[KM03] Klein, Dan; Manning, Christopher D. (2003). "A* parsing: fast exact Viterbi parse selection". Proc. NAACL-HLT.

[GY03] Gross, Jonathan L.; Yellen, Jay (2003), Handbook of Graph Theory, Discrete Mathematics and Its Applications, CRC Press, p. 65

[CLR+01] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001),Introduction to Algorithms, MIT Press and McGraw-Hill, ISBN 978-0-262-03293-3. Section 25.3, "Johnson's algorithm for sparse graphs", pp. 636–640.]

[JON77] Johnson, Donald B. (1977), Efficient algorithms for shortest paths in sparse networks, Journal of the ACM 24 (1): 1–13

[SUU74] Suurballe, J. W. (1974), Disjoint paths in a network, Networks 14 (2): 125–145

[XAV10] Xavier Anguera, "Speaker Diarization: A Review of Recent Research", retrieved 19. August 2010, IEEE TASLP

[EPP98] Eppstein, D. (1998). "Finding the K shortest paths". SIAM J. Comput. 28 (2): 652–673.

[YEN71] Yen J. Y: “Finding the K-Shortest Loopless Paths in a Network”. Management Science 1971;