Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Σχολή Θετικών Επιστημών

Πτυχιακή εργασία

Εύρεση μεγίστου μονοπατιού σε co-comparability γραφήματα

Μαρία Παππά

Οκτώβριος 2016

Ευχαριστίες

Θέλω να ευχαριστήσω την οικογένεια μου που με στηρίζει αμέριστα στις αποφάσεις μου όλα αυτά τα χρόνια. Επίσης θέλω να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου κ. Λεωνίδα Παληό, που με την πολύτιμη βοήθεια, καθοδήγηση και υπομονή του ολοκληρώθηκε αυτή η πτυχιακή εργασία.

Περίληψη

Σε ένα γράφημα το πρόβλημα το μεγίστου μονοπατιού απαιτεί να υπολογίσουμε ένα απλό μονοπάτι του γραφήματος που περιλαμβάνει το μεγαλύτερο δυνατό αριθμό κόμβων που ανήκουν σε αυτό. Το πρόβλημα αυτό αποτελεί μια γενίκευση του γνωστού προβλήματος της εύρεσης του *Hamiltonian* μονοπατιού. Στο πρόβλημα της εύρεσης του Hamiltonian μονοπατιού καθορίζεται αν ένας γράφος είναι Hamiltonian. Ένας Hamiltonian γράφος περιέχει ένα απλό μονοπάτι στο οποίο κάθε κόμβος του γραφήματος εμφανίζεται αποκλειστικά μία φορά. Το πρόβλημα εύρεσης του μεγίστου μονοπατιού ή ισοδύναμα το πρόβλημα εύρεσης ενός μέγιστου Hamiltonian υπογράφου σε ένα γράφημα G, είναι ένα *NP-complete* πρόβλημα. Υπάρχουν ορισμένες οικογένειες γραφημάτων, όπως τα δέντρα, για τις οποίες έχουν ήδη βρεθεί πολυωνυμικοί αλγόριθμοι που δίνουν λύση στο Hamiltonian πρόβλημα. Πρόσφατα αποδείχθηκε ότι το πρόβλημα εύρεσης του μεγίστου μονοπατιού στα *cocomparability* γραφήματα μπορεί να επιλυθεί σε πολυωνυμικό χρόνο, κάνοντας χρήση δυναμικού προγραμματισμού στη διάταξη των κόμβων ενός γραφήματος. Σε αυτή την πτυχιακή εργασία μελετήσαμε τον αλγόριθμο των G.B. Merzios, D.G. Corneil και παραθέτουμε μία πολυωνυμική λύση για την εύρεση του μεγίστου μονοπατιού σε μία σημαντική και πολύ γνωστή κλάση γραφημάτων, τα cocomparability γραφήματα. Η λύση που μελετήσαμε πραγματοποιείται με χρήση δυναμικού προγραμματισμού. Ο αλγόριθμος που υλοποιήσαμε είναι ένας αλγόριθμος δυναμικού προγραμματισμού που χρησιμοποιεί τον αλγόριθμο της λεξικογραφικής εις βάθος διερεύνησης *lexicographic depth first search* (LDFS). Τα αποτελέσματα αποδεικνύουν πως αυτή η προσέγγιση δυναμικού προγραμματισμού μπορεί να φανεί χρήσιμη και αποτελεσματική στη γενίκευσή της και στην μετέπειτα εύρεση μεγίστων μονοπατιών σε ακόμα μεγαλύτερες κλάσεις γραφημάτων.

Κεφάλαιο 1

Εισαγωγή

Σε αυτό το κεφάλαιο θα αναφερθούμε σε κάποιους βασικούς ορισμούς. Θα ορίσουμε το αντικείμενο μελέτης και τη δομή της πτυχιακής μας εργασίας.

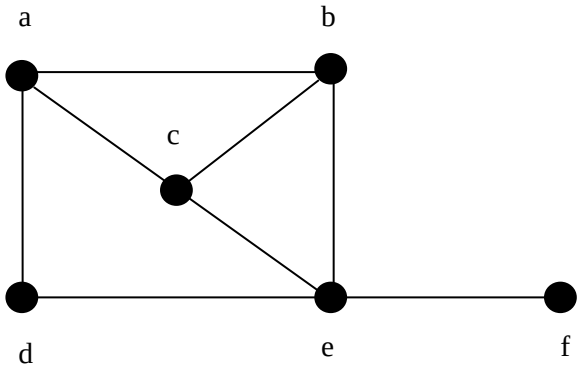
1.1 Βασικοί Ορισμοί

Πριν αρχίσουμε να εξετάζουμε λεπτομερώς τα επιμέρους τμήματα της πτυχιακής εργασίας, αξίζει να αφιερώσουμε λίγο χρόνο για να δούμε τους βασικούς ορισμούς που συνδέονται άμεσα με το αντικείμενό της.

ΟΡΙΣΜΟΣ 1.1.1. *Γράφος ή γράφημα (graph) G είναι μια δομή δεδομένων που αποτελείται από ένα ζεύγος συνόλων G = (V,E). Το σύνολο V αποτελεί το σύνολο των κορυφών του γραφήματος. Ο αριθμός των κορυφών n = |V| αποτελεί τον βαθμό (order) του G. Το σύνολο E περιέχει το σύνολο των ακμών του γραφήματος.*

Υπάρχουν διαφορετικοί τύποι γραφημάτων ανάλογα με το είδος και το πλήθος των ακμών που συνδέουν το ζεύγος των κορυφών.

ΟΡΙΣΜΟΣ 1.1.2. *Ένα μη κατευθυνόμενο γράφημα (undirected graph) ή απλώς γράφημα, είναι ένα διατεταγμένο ζεύγος $G = (V, E)$, όπου $V = \{v_1, v_2, \dots, v_n\}$ ένα μη κενό και πεπερασμένο σύνολο με n διακεκριμένα στοιχεία, που αποτελούν το σύνολο των κορυφών του γραφήματος και όπου $E = \{e_1, e_2, \dots, e_m\}$, με $m \geq 0$, ένα σύνολο υποσυνόλων του V, το καθένα εκ των οποίων έχει δύο στοιχεία του V, που συντελούν τις ακμές του γραφήματος.*



Σχήμα 1.1.1. Ένα μη-κατευθυνόμενο γράφημα G=(V, E) με V = {a, b, c, d, e, f} και E = { (a, b), (a, d), (a, c), (c, e), (d, e), (e, f), (b, c), (b, e) }

ΟΡΙΣΜΟΣ 1.1.3 *Μονοπάτι (path) P_k ενός γραφήματος G = (V, E) είναι μία ακολουθία ακμών μεταξύ δύο κορυφών u και v, τέτοια ώστε το u να είναι το πρώτο στοιχείο της ακολουθίας και το v το τελευταίο. Οι κορυφές οι οποίες περιλαμβάνονται στην ακολουθία ακμών πρέπει να είναι διαφορετικές μεταξύ τους.*

Οι κορυφές u και v της ακολουθίας ονομάζονται άκρα του μονοπατιού. Όταν το μονοπάτι έχει την ίδια αρχική και τελική κορυφή ονομάζεται κύκλος.

ΟΡΙΣΜΟΣ 1.1.4 *Ένα απλό μονοπάτι είναι μια διαδοχική ακολουθία ακμών στην οποία η κάθε ακμή περιέχεται μία και μοναδική φορά.*

Ένα παράδειγμα ενός απλού μονοπατιού σύμφωνα με το Σχήμα 1.1.1. είναι το μονοπάτι που σχηματίζεται από την ακολουθία των κόμβων : (a, b, c, e). Ένα μονοπάτι όπως το μονοπάτι που σχηματίζεται από τους κόμβους ; (a, b, e, c, b) δεν αποτελεί απλό μονοπάτι, καθώς σχηματίζεται κύκλος. Στην πτυχιακή εργασία μελετήσαμε απλά μονοπάτια.

ΟΡΙΣΜΟΣ 1.1.5 *Έστω G = (V, E) ένα γράφημα και έστω S υποσύνολο του V. Καλούμε γειτονιά του S στο G το σύνολο των κόμβων του G που είναι συνδεδεμένες με έναν κόμβο u και δεν ανήκουν στο S. Αν $u \in V$ τότε ορίζουμε $N_G(u)$ σ = $N_G(\{u\})$. Αν για έναν κόμβο $x \in V$ ισχύει ότι $N_G(x) = \varnothing$ τότε λέμε ότι ο x είναι απομονωμένος κόμβος.*

ΟΡΙΣΜΟΣ 1.1.6 *Με Adj(u_i) συμβολίζεται το σύνολο γειτνίασης του κόμβου u_i. Συνεπώς αν (u, v) $\in$ E τότε το v ανήκει στο σύνολο γειτνίασης του κόμβου u.*

ΟΡΙΣΜΟΣ 1.1.7 *Συμπληρωματικό ενός γραφήματος G = (V, E) είναι το γράφημα $\bar{G}$ = (V, $\bar{E}$) που έχει το ίδιο σύνολο κορυφών (V) με το G και περιλαμβάνει μια ακμή {a, b} στο σύνολο $\bar{E}$ αν και μόνο αν η ακμή {a, b} δεν ανήκει στο E.*

1.2. Cocomparability Γραφήματα

Σε αυτή την ενότητα θα αναλύσουμε την οικογένεια γραφημάτων που μελετήσαμε, τα cocomparability γραφήματα.

Τα cocomparability γραφήματα εκπροσωπούν μια πολύ γνωστή και σημαντική κλάση που υπάγεται στα τέλεια γραφήματα, η οποία είναι μια υπέρ-κλάση των *interval* και *permutation* γραφημάτων. Τα interval και permutation γραφήματα έχουν την ίδια γραμμική δομή με τα cocomparability γραφήματα.

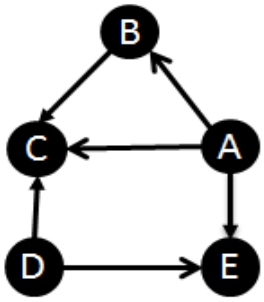
Πριν όμως δούμε τι είναι ένα cocomparability γράφημα ας δούμε τι είναι ένα *comparability* γράφημα.

ΟΡΙΣΜΟΣ 1.2.1. *Ένα comparability γράφημα είναι ένα μη-κατευθυνόμενο γράφημα τέτοιο ώστε:*

1. Υπάρχει ανάθεση φορών στις ακμές του ώστε αν $a \rightarrow b$ και $b \rightarrow c$ τότε υπάρχει η ακμή ac και έχει φορά $a \rightarrow c$.

2. Δεν σχηματίζεται κατευθυνόμενος κύκλος.

Με απλά λόγια comparability είναι ένα γράφημα αν μπορούμε να αναθέσουμε κατεύθυνση στις ακμές του.



Σχήμα 1.2.1. Ένα cocomparability γράφημα

ΟΡΙΣΜΟΣ 1.2.2. *Ένα cocomparability γράφημα είναι ένα γράφημα G του οποίου το συμπληρωματικό γράφημα $\bar{G}$ είναι ένα comparability γράφημα.*

Συμπερασματικά αν υπάρχει μια ανάθεση των ακμών του $\bar{E}$ τέτοια ώστε να υπάρχει μια ακμή με φορά από το x στο y και μία ανάθεση από το y στο z, τότε υπάρχει ακμή xz με φορά από το x στο z.

Με άλλα λόγια, ένα γράφημα G = (V, E), οι ακμές του οποίου είναι ακριβώς οι συγκρίσιμες δυάδες μιας συγκεκριμένης διάταξης π του γραφήματος, ονομάζεται comparability γράφημα. Το συμπληρωματικό γράφημα G του οποίου οι ακμές είναι συμπληρωματικά συγκρίσιμα ζεύγη του π, ονομάζεται cocomparability γράφημα του π.

1.3 Αντικείμενο της Πτυχιακής Εργασίας

Στην πτυχιακή εργασία μελετήσαμε και παρουσιάζουμε τον πολυωνυμικό αλγόριθμο για την εύρεση του μεγίστου μονοπατιού σε μία γνωστή κλάση γραφημάτων τα cocomparability γραφήματα.

Το πρόβλημα εύρεσης της Hamiltonian διαδρομής σε cocomparability γραφήματα έχει αποδειχθεί ότι είναι πρόβλημα πολυωνυμικού χρόνου δεκαετίες πριν, όμως η πολυπλοκότητα του προβλήματος στην κλάση των cocomparability γραφημάτων παρέμενε ανοιχτή. Πιο συγκεκριμένα η πολυπλοκότητα του προβλήματος εύρεσης του μεγίστου μονοπατιού παραμένει ανοιχτή σε περισσότερες από μια κλάσεις γραφημάτων που υπάγονται στα *permutation* γραφήματα.

Σε αυτή την εργασία παραθέτουμε την πολυωνυμική λύση του προβλήματος του μεγίστου μονοπατιού στα cocomparability γραφήματα. Δοθέντος ενός cocomparability γραφήματος G, θέλουμε να υπολογίσουμε το μέγιστο απλό μονοπάτι. Το μέγιστο απλό μονοπάτι είναι μια απλή διαδρομή στο G η οποία έχει μέγιστο μήκος. Ο αλγόριθμος που υλοποιήσαμε εκτελεί έναν απλό υπολογισμό κάνοντας χρήση δυναμικού προγραμματισμού. Η εφαρμογή του δυναμικού προγραμματισμού γίνεται σε μία LDFS διάταξη των κόμβων του γραφήματος. Η υλοποίηση των αλγορίθμων που μελετήθηκαν έγινε σε γλώσσα C.

1.4. Σχετικά Ερευνητικά Αποτελέσματα

Σε αυτή την παράγραφο σημειώνονται σχετικά ερευνητικά αποτελέσματα στα οποία βασιστήκαμε για την διεξαγωγή της πτυχιακής μας εργασίας. Η εργασία των Corneil, Dalton, Habib An *unified view of graph searching* που δημοσιεύτηκε το 2008, εισήγαγε τη έννοια της προεπεξεργασίας των κόμβων ενός γραφήματος λεξικογραφικά, κάνοντας χρήση του αλγορίθμου λεξικογραφικής αναζήτησης εις βάθος για πρώτη φορά (LexDFS). Στη συνέχεια η μελέτη των Corneil, Dalton, Habib στην εργασία *Certifying algorithm for minimum path cover problem on cocomparability graphs using LDFS*, έκανε για πρώτη φορά τη χρήση της LDFS διάταξη κόμβων με σκοπό την εύρεση του ελαχίστου μονοπατιού στα cocomparability γραφήματα. Πιο πρόσφατα η έρευνα των Ioannidou, Mertzios και Nikolopoulos με θέμα *The longest path problem has a polynomial solution on interval graphs*, η οποία δημοσιεύθηκε το 2011, πραγματεύεται μια πολυωνυμική λύση πάνω στο πρόβλημα εύρεσης του μεγίστου μονοπατιού στην κλάση των interval γραφημάτων.

1.5. Δομή Πτυχιακή Εργασίας

Στο πρώτο κεφάλαιο κάναμε μια εισαγωγή στους βασικούς ορισμούς που σχετίζονται με το αντικείμενο της πτυχιακής μας εργασία και έναν καθορισμό του προβλήματος μελέτης. Στο δεύτερο κεφάλαιο παραθέτουμε το απαραίτητο θεωρητικό υπόβαθρο, συμπεριλαμβάνοντας τα χαρακτηριστικά της ακολουθίας των κόμβων των cocomparability γραφημάτων και τις LDFS διατάξεις τους. Στο κεφάλαιο 3 μελετούμε την υλοποίηση του αλγορίθμου εύρεσης του μεγίστου μονοπατιού, τις εισόδους και εξόδους που δέχεται και αποδίδει, τις δομές δεδομένων που χρησιμοποιήθηκαν, επιλεγμένες συναρτήσεις που υλοποιήθηκαν καθώς επίσης και ενδεικτικές εκτελέσεις του αλγορίθμου. Τέλος, στο

Κεφάλαιο 4 αναφερόμαστε στα συμπεράσματα της μελέτης μας και ακόμα στις επεκτάσεις του αντικειμένου μελέτης σε ερευνητικό επίπεδο.

Κεφάλαιο 2

Ο αλγόριθμος LDFS$^+$

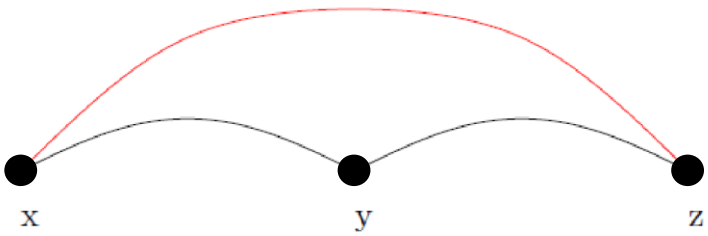
Σε αυτό το κεφάλαιο αναφερόμαστε στο θεωρητικό υπόβαθρο της μελέτης για την υλοποίηση του αλγορίθμου της λεξικογραφικής διερεύνησης εις βάθος, καθώς επίσης και στη κωδικοποίηση του αλγορίθμου.

2.1. Θεωρητικό Υπόβαθρο

Σε αυτό την ενότητα του κεφαλαίου επεκτεινόμαστε στα χαρακτηριστικά της διάταξης των κόμβων των cocomparability γραφημάτων που μελετούμε.

ΘΕΩΡΗΜΑ 2.1.1. *Το G = (V, E) είναι cocomparability γράφημα αν και μόνο αν υπάρχει μια διάταξη των κόμβων του V τέτοια ώστε για κάθε x < y < z, xz ∈ E ή yz ∈ E ή και τα δύο.*

Το παραπάνω θεώρημα αποτυπώνεται για την τριάδα κόμβων x, y, z στο παρακάτω σχήμα.



Σχήμα 2.1.1. Μεταβατική διάταξη των κόμβων στα cocomparability γραφήματα.

ΟΡΙΣΜΟΣ 2.1.1. *Έστω ένα γράφημα G = (V, E). Μία διάταξη των κόμβων του V είναι umbrella-free διάταξη αν για κάθε x < y < z, xz ∈ E συνεπάγεται ότι xy ∈ E ή yz ∈ E (ή και τα δύο).*

Αυτό το γεγονός είναι λογικό καθώς ταιριάζει με την μεταβατική φύση των cocomparability γραφημάτων.

ΛΗΜΜΑ 2.1.1. G = (V, E) είναι ένα *cocomparability* γράφημα αν και μόνο αν υπάρχει μια *umbrella-free* διάταξη των κόμβων του.

Επομένως η χαρακτηριστική ιδιαιτερότητα στη σύνδεση των κόμβων ενός cocomparability γραφήματος μπορούμε να πούμε ότι εξασφαλίζεται από την ύπαρξη μιας umbrella-free ακολουθίας των κόμβων του.

ΛΗΜΜΑ 2.1.2. Το G = (V, E) είναι ένα συνεκτικό γράφημα αν και μόνο αν υπάρχει μία ακολουθία των στοιχείων (που ονομάζεται *i*-διάταξη) του V έτσι ώστε για όλα τα x < y < z, αν ισχύει xz ∈ E τότε επίσης ισχύει xy ∈ E.

Λαμβάνοντας υπόψη τα παραπάνω ερχόμαστε στο συμπέρασμα ότι μία *i*-διάταξη ενός συνεκτικού γραφήματος G είναι επίσης *umbrella-free* διάταξη.

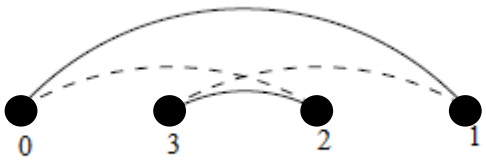
ΟΡΙΣΜΟΣ 2.1.2. *Έστω γράφημα G = (V, E) και σ(i) = (u_o , u_1, ..., u_n) μια διάταξη των κόμβων του. Έστω (a, b ,c) μία επιλογή τριών κόμβων του έτσι ώστε a <$_\sigma$ b <$_\sigma$ c, με ac ∈ E και ab ∉ E. Αν ∃ d ∈ V τέτοιος ώστε:*

1. a <$_\sigma$ d <$_\sigma$ c, στη διάταξη των κόμβων (σ) του γραφήματος G = (V, E).

2. db ∈ E και dc ∉ E.

Τότε οι κόμβοι a, b, c λέμε ότι είναι μια σωστή τριάδας κόμβων στο γράφημα. Σε διαφορετική περίπτωση λέμε ότι είναι λάθος τριάδα κόμβων.

Ένα καλό παράδειγμα μια καλής τριάδας κόμβων (0, 1, 2) και ένας κόμβος 3 αποτυπώνεται στην παρακάτω εικόνα.



Σχήμα 2.1.2: Μία σωστή τριάδα (0, 1, 2) και ένας κόμβος 3 που ανήκει στην διάταξη σ = (0, 1, 2, 3).

ΟΡΙΣΜΟΣ 2.1.3. *Έστω γράφημα G = (V, E) είναι ένα γράφημα με μια διάταξη των κόμβων του σ(i) = {ui, ..., uk}. Αν η διάταξη σ(i) = {ui, ..., uk} αποτελείται μόνο από σωστές τριάδες επιλογής τότε η διάταξη των κόμβων ονομάζεται LDFS διάταξη.*

Συνεπώς αυτό μας δίνει το συμπέρασμα ότι μια LDFS διάταξη κόμβων είναι και *umbrella-free* διάταξη.

2.2.Ο Αλγόριθμος LDFS$^+$

Σε αυτή την ενότητα παραθέτουμε τον πρώτο αλγόριθμο που υλοποιήσαμε ο οποίος ονομάζεται αλγόριθμος λεξικογραφικής διερεύνησης εις βάθος (*lexicographic depth first search)* LDFS$^+$. Ο αλγόριθμος μοιάζει εξαιρετικά με τον ιδιαίτερα γνωστό αλγόριθμο λεξικογραφικής διερεύνησης εις πλάτος (*lexicographic breath first search*). Ως είσοδο ο αλγόριθμος δέχεται ένα συνεκτικό γράφημα G = (V, E) με n κόμβους και μια διάταξη π των στοιχείων του V. Ο αλγόριθμος επιστρέφει μια ακολουθία σ = (σ(1),σ(2),...,σ(n)) των κόμβων του γραφήματος. Δοθέντος ενός γραφήματος G = (V, E), αρχικά ο LDFS$^+$ εκχωρεί μια ταμπέλα με την κενή τιμή (ε) σε όλους του κόμβους του γραφήματος. Η επιλογή του κόμβου γίνεται με τον εξής τρόπο: η επιλογή του κόμβου εκκίνησης γίνεται πάντα επιλέγοντας τον δεξιότερο μη αριθμημένο κόμβου της διάταξης εισόδου. Ο LDFS$^+$ αλγόριθμος επιλέγει πάντα το δεξιότερο και μη αριθμημένο κόμβο της διάταξης εισόδου π και εκτελεί την λεξικογραφική διερεύνηση εις βάθος. Αν δύο υποψήφιοι κόμβοι έχουν τον ίδιο βαθμό επιλέγει πάλι τον δεξιότερο, διαφορετικά επιλέγει τον κόμβο με τον μεγαλύτερο βαθμό.

ΘΕΩΡΗΜΑ 2.2.1. *Δοθέντος ενός γραφήματος G = (V, E), μια διάταξη σ των στοιχείων του V μπορεί να επιστραφεί από την εφαρμογή του* LDFS$^+$ *στο G αν και μόνο αν είναι μία LDFS ακολουθία.*

Η παραπάνω θεώρηση πρακτικά σκιαγραφείται στην παρακάτω κωδικοποίηση του αλγορίθμου LDFS$^+$.

Αλγόριθμος 1. LDFS$^+$ (G, π)

Είσοδος: Συνεκτικό γράφημα G = (V, E) και μία ακολουθία π (του V)

Έξοδος: Μία LDFS ακολουθία σ των κόμβων του G

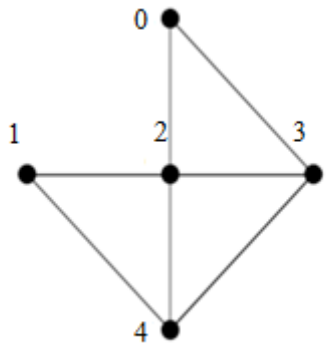
1. Ανάθεσε την ετικέτα ε σε όλους τους κόμβους
2. για i = 1 μέχρι n κάνε τα παρακάτω:
3. Επίλεξε τον δεξιότερο κόμβο u στην π από τους μη-αριθμημένους κόμβους με τη μεγαλύτερη λεξικογραφική ετικέτα
4. $\sigma_u(i) \leftarrow u$ { ανάθεσε το u τον αριθμό i }
5. για κάθε μη-αριθμημένο κόμβο $w \in N(u)$ κάνε:
6. ετικέτα(w) ←(i,ετικέτα(w))

7. Επίστρεψε την ακολουθία $\sigma_u = (\sigma_u(1), \sigma_u(2), \ldots, \sigma_u(n))$.

Ο LDFS$^+$ μπορεί να υλοποιηθεί σε **O(min{n2, n+mlogn})** χρόνο, παρόλο που η μέχρι τώρα γρηγορότερη υλοποίηση του τρέχει σε χρόνο **O(min{n2, n+mloglogn}).**

Θέτοντας την σ = LDFS$^+$(G, u) επιστρέφεται μια LDFS διάταξη που είναι cocomparability διάταξη αν και η u είναι cocomparability διάταξη. Χοντρικά, ο αλγόριθμος LDFS$^+$ είναι μία DFS αναζήτηση όπου οι δεσμοί είναι χωρισμένοι έτσι ώστε να ευνοούν τις κορυφές με γείτονες τις κορυφές που έχουν πιο πρόσφατα επισκεφθεί.

2.2.1. Ενδεικτικές Εφαρμογές του Αλγορίθμου



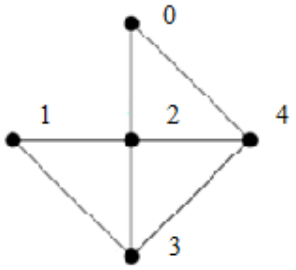
Σχήμα 2.2.1.1. Γράφημα G

Έστω ένα cocomparability γράφημα G = (V, E). Εφαρμόζουμε τον αλγόριθμο LDFS$^+$ στη διάταξη π, προκειμένου να δημιουργήσουμε μια LDFS *umbrella free* διάταξη σ των στοιχείων του V. Ας υποθέσουμε ότι έχουμε αυτό το γράφημα G και μία cocomparability διάταξη π = {0, 1, 2, 3, 4}.

Έχουμε την διάταξη π = {4, 3, 2, 1, 0} η οποία χρησιμοποιείται στην εφαρμογή του Αλγόριθμου 1. Αρχικά επιλέγεται ο κόμβος 4 που είναι ο δεξιότερος κόμβος της διάταξης. Τώρα από τον κόμβο 4 υπάρχει ένα σύνδεσμος για κάθε κόμβο μεταξύ των κόμβων 3, 2 και 1. Επιλέγεται ο κόμβος 3, εφόσον είναι ο δεξιότερος κόμβος στην διάταξη π, τον οποίο ακολουθεί ο κόμβος 2. Στη συνέχεια μεταξύ των κόμβων 0 και 2 επιλέγεται ο δεξιότερος πάλι κόμβος 2. Έπειτα μεταξύ των κόμβων 0 και 1 επιλέγεται ο κόμβος 0, επειδή επισκέφθηκε πιο πρόσφατα και άρα έχει μεγαλύτερη προτεραιότητα από τον 1. Τέλος ο κόμβος 0. Επομένως η διάταξη εξόδου είναι η σ = {4, 3, 2, 0, 1}

Η επιλογή των κόμβων από την εκτέλεση σε στάδια φαίνεται στο παρακάτω σχήμα.

Βήμα 1.	[4] 3 2 1 0
Βήμα 2.	4 [3] 2 1 0
Βήμα 3.	4 3 [2] 0 1
Βήμα 4.	4 3 2 [0] 1
Βήμα 5.	4 3 2 0 [1]



Σχήμα 2.2.1.2. G(V, E) ένα cocomparability γράφημα, και βήμα προς βήμα ο υπολογισμός της LDFS διάταξης ξεκινώντας από τον κόμβο 4.

2.2.3. Ο Αλγόριθμος Εύρεσης Μεγίστου Μονοπατιού

Σε αυτή την ενότητα κάνουμε μια εισαγωγή στο θεωρητικό μέρος της υλοποίησης του κύριου αλγόριθμου που μελετήσαμε για την κλάση των cocomparability γραφημάτων. Δίνουμε μια σύντομη επισκόπηση του Αλγορίθμου 2. Παίρνει ως είσοδο ένα cocomparability γράφημα G = (V, E) και μία umbrella-free διάταξη π του συνόλου V. Ως στάδιο προεπεξεργασίας ο αλγόριθμος εφαρμόζει τον αλγόριθμο LDFS$^+$ που αναπτύξαμε παραπάνω, προκειμένου να υπολογιστεί μια umbrella-free διάταξη των κόμβων του G. Σε ότι ακολουθεί στο υπόλοιπο μέρος του αλγόριθμου, υλοποιώντας το με δυναμικό προγραμματισμό, χτίζουμε έναν πίνακα όπου για κάθε ζεύγος i, j ∈ {1, 2, …, n} για κάθε κορυφή u_k ∈ V(G(i,j)) η εγγραφή P(u_k; i, j) αποθηκεύει τις διατεταγμένες κορυφές ενός μεγίστου μονοπατιού του G(i, j), με το u_k ως τον τελευταίο κόμβο. Το μήκος του μονοπατιού P αποθηκεύεται στη δομή l(u_k; i, j). Είναι σημαντικό σε αυτό το σημείο να αναφέρουμε ότι στα επαναληπτικά for που φαίνονται στις γραμμές 3 και 4 του αλγορίθμου είναι εμφανής η επαγωγική υπόθεση. Αυτό σημαίνει ότι μπορεί να υποτεθεί κατά τη διάρκεια της {i, j}-στης αρχικοποίησης του κορμού του δυναμικού προγραμματισμού, οι τιμές των εγγραφών P(u_k; i', j') και l(u_k; i', j') έχουν υπολογιστεί σωστά για κάθε i' > i σε προηγούμενες αρχικοποιήσεις του αλγορίθμου.

Κατά την αρχικοποίηση μια συγκεκριμένης δυάδας {i, j} επιθυμούμε να αρχικοποιήσουμε τα μονοπάτια που δεν χρησιμοποιούν τον κόμβο ui ως ενδιάμεσο κόμβο. Αυτό εκφράζεται στις γραμμές 5-8 του Αλγορίθμου 2. Για ένα μονοπάτι που έχει το uy ∈ V(G(i+1, j)) ως τον τελευταίο του κόμβο, θα αποθηκευθεί στην εγγραφή P(u_k; i+1, j). Για ένα μονοπάτι με το ui ως τον τελευταίο του κόμβο, ενδιαφερόμαστε μόνο για την περίπτωση που το ui ∈ V(G(i, j)), και έτσι το αρχικοποιούμε ως εξής: P(u_k; i, j) = (u_i).

Έπειτα, χρησιμοποιώντας το επαγωγικό βήμα του αλγορίθμου (βλ. γραμμές 9-17) καθορίζουμε τον τρόπο με τον οποίο οι εγγραφές του πίνακα μπορούν να επεκταθούν με την προσθήκη του κόμβου ui, στην περίπτωση που ui ∈ V(G(i,j)). Πρώτα, παρατηρούμε ότι αν ένα μονοπάτι P των G(i,j) που εμπεριέχει τον κόμβο ui έχει τουλάχιστον δύο κόμβους, τότε το P πρέπει να εμπεριέχει έναν κόμβο u_x ∈ V(G(i+1, j)) με $u_x u_i$ ∈ E. Κάθε κόμβος u_x, μπορεί

να εξυπηρετήσει με δύο διαφορετικές ιδιότητες, με σκοπό να βρεθεί το πιθανό μέγιστο μονοπάτι και στη συνέχεια να αποθηκευθεί στον πίνακα μου κρατά τα μονοπάτια. Στην πρώτη περίπτωση, προσθέτουμε μια ακμή u_xu_i σε ένα μέγιστο μονοπάτι του G(i+1,j) με u_x τον τελευταίο κόμβο. Αυτό μπορεί να δημιουργήσει ένα μονοπάτι με τον κόμβο ui ως τον τελευταίο του κόμβο, το οποίο είναι μεγαλύτερο από το τρέχον αποθηκευμένο στην εγγραφή $P(u_k; i, j)$. Αυτή την περίπτωση περιγράφουμε στις γραμμές 11-13. Η δεύτερη ιδιότητα που μπορεί να υιοθετήσει ο κόμβος ux είναι να αποτελέσει τον σύνδεσμο ανάμεσα στις δύο ακμές u_xu_i και $u_iu_{y'}$. Αυτή η κατάσταση περιγράφεται στις γραμμές 14-17.

ΠΑΡΑΤΗΡΗΣΗ 2.3.1. *Έστω G=(V,E) ένα cocomparability γράφημα και* $\sigma = (u_1, u_2,\ldots, u_n, u_{n+1})$ *μία LDFS umbrella-free διάταξη των στοιχείων του V και του στοιχείου* u_{n+1}. *Για κάθε ζευγάρι από τους δείκτες* $i, j \in \{1, 2,\ldots, n\}$ *και για κάθε κόμβο* $u_k \in V(G(i, j))$ *ορίζουμε με το* $P(u_k; i, j)$ *ένα μέγιστο μονοπάτι του* $G(i, j)$ *με* u_k *ως τον τελευταίο κόμβο και με* $l(u_k; i, j)$ *το μήκος* $|P(u_k; i, j)|$ *του* $P(u_k; i, j)$ *δηλαδή ο αριθμός των κόμβων του* $P(u_k; i, j)$.

Ο υπολογισμός του μέγιστου μονοπατιού σε ένα cocomparability γράφημα σκιάγραφείται στο παρακάτω σχήμα.

Αλγόριθμος 2. Υπολογισμός του μέγιστου μονοπατιού σε ένα cocomparability γράφημα

Είσοδος: Ένα cocomparability γράφημα $G = (V,E)$ με $|V| = n$ και μία umbrella-free διάταξη π του V
Output: Ένα μέγιστο μονοπάτι του G

1. Τρέξε το LDFS+ βήμα προ-επεξεργασίας στην π που επιστρέφει την LDFS umbrella-free διάταξη σ
2. Πρόσθεσε ένα απομονωμένο κόμβο u_{n+1} στο σ; ώστε $\sigma = (u_1, u_2, \ldots, u_n, u_{n+1})$
3. **για** $i = n$ μέχρι 1 **κάνε**
4. **για** $j = i$ μέχρι n **do**
5. **για** κάθε $u_y \in V(G(i + 1, j))$ **κάνε**
6. $P(u_y; i, j) \leftarrow P(u_y; i + 1, j)$;
   $l(u_y; i, j) \leftarrow l(u_y; i + 1, j)$ {αρχικοποίηση}
7. **αν** $u_i \in V(G(i, j))$ **τότε**
8. $P(u_i; i, j) \leftarrow (u_i)$;
   $l(u_i; i, j) \leftarrow 1$ {αρχικοποίηση}
9. **για** κάθε $u_x \in V(G(i + 1, j))$ **κάνε**
10. **αν** $u_i \in V(G(i, j))$ και $u_x \in N(u_i)$ **τότε**
11. **αν** $l(u_i; i, j) < l(u_x; i + 1, j) + 1$ **τότε**
12. $P(u_i; i, j) \leftarrow (P(u_x; i + 1, j), u_i)$
13. $l(u_i; i, j) \leftarrow l(ux; i + 1, j) + 1$
14. **για** κάθε $u_y \in V(G(i + 1, x - 1))$ **κάνε**
15. **αν** $l(u_y; i, j) < l(u_x; i+ 1, j) + l(u_y; i + 1, x - 1) + 1$ **τότε**
16. $P(u_y; i, j) \leftarrow (P(u_x; i + 1, j), u_i, P(u_y; i + 1, x - 1))$
17. $l(u_y; i, j) \leftarrow l(u_x; i + 1, j) + l(u_y; i + 1, x - 1) + 1$
18. **επίστρεψε** ένα μονοπάτι $P(u_k; 1, n)$ με $l(u_k; 1, n) = \max\{l(u_y; 1, n) \mid u_y \in V\}$

Κεφάλαιο 3

Η Υλοποίηση του Αλγορίθμου

Σε αυτό το κεφάλαιο παρουσιάζουμε τον πολυωνυμικό αλγόριθμο που υπολογίζει το μέγιστο μονοπάτι ενός cocomparability γραφήματος. Η υλοποίηση του αλγορίθμου πραγματοποιήθηκε σε γλώσσα C.

3.1. Είσοδος - Έξοδος

Σε αυτό το σημείο περιγράφουμε περιληπτικά πως λειτουργεί ο αλγόριθμος εύρεσης του μεγίστου μονοπατιού. Ο αλγόριθμος δέχεται ως είσοδο ένα cocomparability γράφημα *G = (V, E)* και μία *umbrella free* διάταξη *π* που περιλαμβάνει τα στοιχεία του V. Αρχικά ως βήμα προ-επεξεργασίας, εφαρμόζουμε τον αλγόριθμο $LDFS^+$ στη διάταξη *π*, προκειμένου να δημιουργήσουμε μια LDFS *umbrella free* διάταξη *σ* των στοιχείων του V.

Το κύριο θεώρημα αυτής της ενότητας αποδεικνύει ότι ο Αλγόριθμος 2 υπολογίζει σε $O(n^4)$ χρόνο το μέγιστο μονοπάτι σε ένα cocomparability γράφημα με n κόμβους.

ΘΕΩΡΗΜΑ 1. *Δοθέντος ενός cocomparability γραφήματος G=(V,E) με n κόμβους, ο Αλγόριθμος 2 υπολογίζει το μέγιστο μονοπάτι P σε χρόνο $O(n^4)$.*

Απόδειξη. Στην πρώτη γραμμή ο Αλγόριθμος 2 εφαρμόζει μία $LDFS^+$ προ-επεξεργασία της δοθείσας *umbrella-free* διάταξης *π* των κόμβων που περιέχονται στο V. Η διάταξη εξόδου LDFS αποτελεί επίσης μια διάταξη *umbrella-free*. Στην δεύτερη γραμμή, ο αλγόριθμος προσθέτει ένα απομονωμένο κόμβο u_{n+1} στη διάταξη *σ* δεξιά όλων των κόμβων του V. Δηλαδή θεωρούμε χωρίς βλάβη της γενικότητας ότι $\sigma = (u_1, u_2, \ldots, u_n, u_{n+1})$. Παρατηρούμε ότι η *σ* παραμένει LDFS *umbrella-free* διάταξη, ακόμα και μετά την εισαγωγή του κόμβου u_{n+1} σε αυτήν. Επίσης, παρατηρούμε ότι κάθε μέγιστο μονοπάτι του G, μετά από αυτή την αλλαγή, παραμένει μέγιστο. Έτσι, για να υπολογίσουμε το μέγιστο μονοπάτι του G, αρκεί από το Θεώρημα 2 να υπολογίσουμε το μέγιστο μονοπάτι του G (λαμβάνοντας υπ'όψιν τη μορφή της διάταξης *σ*), δηλαδή το μέγιστο μονοπάτι μεταξύ όλων των μεγίστων μονοπατιών.

Στις γραμμές 3-17, ο Αλγόριθμος 2 επαναλαμβάνεται για κάθε ζευγάρι από δείκτες *i, j* ∈ {1, 2,…, n} και υπολογίζει ένα μονοπάτι $P(u_k; i, j)$ και μία τιμή $l(u_k; i, j)$ για κάθε κόμβο $u_k \in V(G(i, j))$. Θα αποδείξουμε με επαγωγή στο i ότι το $P(u_k; i, j)$ είναι πράγματι το μέγιστο μονοπάτι του $G(i, j)$ με u_k να αποτελεί τον τελευταίο κόμβο και με $l(u_k; i, j) = | P(u_k; i, j) |$.

Για την επαγωγική βάση, έστω ότι $i = n$; Σε αυτή την περίπτωση ισχύει ότι $j = n$ (βλ. γραμμή 4). Ακόμα, $u_i \notin N(u_{i+1})$ για i = n, εφόσον u_{n+1} είναι ένας απομονωμένος κόμβος, και αυτό εκτελείται στη γραμμή 8. Σε αυτή τη γραμμή, ο αλγόριθμος υπολογίζει το μονοπάτι $P(u_n; n,n) = (u_n)$, που είναι ξεκάθαρα το μόνο (και επίσης το μέγιστο) απλό μονοπάτι του G(n, n) με u_n ως τον τελευταίο του κόμβο. Έπειτα, εφόσον G(n+1, n) = ∅, οι γραμμές και 10-17 δεν εκτελούνται καθόλου. Αυτό αποδεικνύει την επαγωγική βάση.

3.2. Δομές Δεδομένων

Σε αυτή την ενότητα περιγράφουμε τις δομές δεδομένων που χρησιμοποιήθηκαν κατά την κωδικοποίηση του αλγορίθμου. Χρησιμοποιούμε ένα δισδιάστατο πίνακα a για να αποθηκεύσουμε για κάθε κόμβο *i* τους *j* γείτονες κόμβους του. Ο πίνακας αυτός καλείται πίνακας γειτνίασης και είναι ένας απλός τρόπος με τον οποίο αναπαριστούμε το γράφημά μας. Η διάταξη των κόμβων εισαγωγής που δέχεται ο αλγόριθμος $LDFS^+$ είναι ένας μονοδιάστατος πίνακας s, ο οποίος αποθηκεύει την διάταξη εισαγωγής που χρησιμοποιείται ως βήμα προ-επεξεργασίας στον τελικό αλγόριθμο. Για την υλοποίηση του δυναμικού προγραμματισμού για τον υπολογισμό των μονοπατιών του γράφου, δημιουργούμε έναν ακέραιο πίνακα τεσσάρων διαστάσεων τον ονομάζουμε *path*. Στις εγγραφές του πίνακα είναι αποδεκτές οι τιμές για κάθε ζεύγος δεικτών *i, j* ∈ {1, 2,…, n}. Για κάθε κόμβο $u_y \in V(G(i, j))$ η εγγραφή $path(u_y; i, j, k)$ αποθηκεύει τη διάταξη των κόμβων που ανήκουν στο μονοπάτι του $G(i, j)$ με τον κόμβο u_y να αποτελεί τον τρέχοντα κόμβο του μονοπατιού. Η τιμή k προσδιορίζει τη σειρά του κόμβου στο μονοπάτι που δημιουργείται από τον συνδυασμό των δεικτών i, j. Το μήκος του μονοπατιού $path(u_y; i, j, k)$ αποθηκεύεται στην εγγραφή $length(u_y; i, j)$.

Σύμφωνα με τα παραπάνω ένα μέγιστο μονοπάτι του $G = G(1,n)$ θα αποθηκευτεί στην εγγραφή $path(u_y; 1, j, k)$ για ένα κόμβο u_y που μεγιστοποιεί το $length(u_y; 1, j)$ ανάμεσα σε όλους τους κόμβους $u_y \in V$. Χρειαζόμαστε επίσης, έναν 3-διάστατο πίνακα ακεραίων που να φυλάσσει το incomparability γράφημα που τον καλούμε incomp. Ο incomp πίνακας αποτελεί έναν βοηθητικό πίνακα, που χρησιμοποιούμε για να αποθηκεύσουμε μια transitive ανάθεση στις κατευθύνσεις των ακμών του συμπληρωματικού του γραφήματος εισόδου. Τα στοιχεία του πίνακα χωρίζονται σε δύο σύνολα, που εκφράζουν τις δύο διαφορετικές εκδοχές των κατευθύνσεων των ακμών του γραφήματος. Κάθε εγγραφή του πίνακα είναι ένας συνδυασμός ακμών. Δηλαδή για παράδειγμα η κατευθυνόμενη ακμή ab είναι γειτονική στην bc και αν δεν υπάρχει ακμή ac τότε για τις ακμές ab και bc, έχουμε incomp[a][b][c] = 1. Ακόμα για την dfs αναζήτηση, που πραγματοποιούμε στον incomparability γράφο, χρησιμοποιούμε έναν ακέραιο 2-διάστατο πίνακα visited. Ο μονοδιάστατος πίνακας ακεραίων outdegree, που χρησιμοποιείται κατά την τοπολογική ταξινόμηση, αποθηκεύει για κάθε κόμβο τις ακμές που ξεκινάνε από αυτόν και έχουν κατεύθυνση προς κάποιον άλλον κόμβο.

3.3. Επιλεγμένες Συναρτήσεις

Σε αυτή την ενότητα γίνεται αναφορά και λεπτομερής περιγραφή των συναρτήσεων που δημιουργήσαμε και κάνουμε χρήση για τη επίλυση του προβλήματος της μελέτης μας.

3.3.1. Η Συνάρτηση allocate_space

Η συνάρτηση allocate_space είναι τύπου void. Μέσα σε αυτή γίνεται η δυναμική δέσμευση του απαραίτητου χώρου για τους πίνακες που χρησιμοποιούμε στο πρόγραμμα μας. Δέχεται ως ορίσματα τον αριθμό των κόμβων του γραφήματος (int v_num), ένα πίνακα από διευθύνσεις του πίνακα γειτνίασης του γραφήματος (int ***adj_addr), έναν ακέραιο πίνακα που φυλάσσει τις διευθύνσεις του πίνακα διάταξης των κόμβων (int ** sigma_addr), έναν πίνακα ακεραίων που αποθηκεύει τις διευθύνσεις των μονοπατιών γραφήματος (int *****path_addr) και έναν πίνακα ακεραίων που αποθηκεύει τις διευθύνσεις από τα μήκη των μονοπατιών του γραφήματος (int ****length).

```
void allocate_space(int v_num, int ***adj_addr, int **sigma_addr, int *****path_addr, int ****length_addr){

	int i, j, k;

	int **temp_adj, *temp_sigma, ****temp_path, ***temp_length;

	/* allocate space for adjacency matrix */

	temp_adj = (int **)malloc(v_num * sizeof(int *));

	if (temp_adj == NULL){

		printf("\nNo space for adjacency matrix. Aborting...\n");

		exit(-1);

	}

	for (i=v_num; --i >= 0;  ){

		temp_adj[i] = (int *)malloc(v_num * sizeof(int));

		if (temp_adj[i] == NULL){

			printf("\nNo space for adjacency row. Aborting...\n");

			exit(-1);

		}

	}

	*adj_addr = temp_adj;

	/* allocate space for ordering */

	temp_sigma = (int *)malloc(v_num * sizeof(int));

	if (temp_sigma == NULL){

		printf("\nNo space for ordering. Aborting...\n");

		exit(-1);
```

```
}
*sigma_addr = temp_sigma;
/* allocate space for path matrix */
temp_path = (int ****)malloc(v_num * sizeof(int ***));
if (temp_path == NULL){
        printf("\nNo space for path matrix. Aborting...\n");
        exit(-1);
}
for (i=v_num; --i >= 0;  ){
        temp_path[i] = (int ***)malloc(v_num * sizeof(int **));
        if (temp_path[i] == NULL){
                printf("\nNo space for paths. Aborting...\n");
                exit(-1);
        }
        for (j=v_num; --j >= 0;  ){
                temp_path[i][j] = (int **)malloc(v_num * sizeof(int *));
                if (temp_path[i][j] == NULL){
                        printf("\nNo space for paths. Aborting...\n");
                        exit(-1);
                }
                for (k=v_num; --k >= 0;  )
                {
                        temp_path[i][j][k] = (int *)malloc(v_num * sizeof(int));
                        if (temp_path[i][j][k] == NULL){
                                printf("\nNo space for paths. Aborting...\n");
                                exit(-1);
                        }
                }
        }
}
*path_addr = temp_path;
/* allocate space for length matrix */
temp_length = (int ***)malloc(v_num * sizeof(int **));
if (temp_length == NULL){
```

```
                printf("\nNo space for length matrix. Aborting...\n");
                exit(-1);
        }
        for (i=v_num; --i >= 0;  ){
                temp_length[i] = (int **)malloc(v_num * sizeof(int *));
                if (temp_length[i] == NULL){
                        printf("\nNo space for lengths. Aborting...\n");
                        exit(-1);
                }
                for (j=v_num; --j >= 0;  ){
                        temp_length[i][j] = (int *)malloc(v_num * sizeof(int));
                        if (temp_length[i][j] == NULL){
                                printf("\nNo space for lengths. Aborting...\n");
                                exit(-1);
                        }
                }
        }
        *length_addr = temp_length;
}
```

3.3.2. Η Συνάρτηση deallocate_space

Η συνάρτηση deallocate_space είναι τύπου void. Παίρνει τα ίδια ορίσματα με την allocate_space. Η λειτουργία της είναι να αποδεσμεύει τον χώρο που δεσμεύθηκε στην allocate_space.

```
void deallocate_space(int v_num, int **adj, int *sigma, int ****path, int ***length){
        int i, j, k;
        /* deallocate space for adjacency matrix */
        for (i=v_num; --i >= 0;  )
                free(adj[i]);
        free(adj);
        /* deallocate space for ordering */
        free(sigma);
        /* deallocate space for path matrix */
        for (i=v_num; --i >= 0;  ){
```

```
                for (j=v_num; --j >= 0;  ){
                        for (k=v_num; --k >= 0;  )
                                free(path[i][j][k]);
                        free(path[i][j]);
                }
                free(path[i]);
        }
        free(path);
        /* deallocate space for length matrix */
        for (i=v_num; --i >= 0;  ){
                for (j=v_num; --j >= 0;  )
                        free(length[i][j]);
                free(length[i]);
        }
        free(length);
}
```

3.3.3. Η Συνάρτηση read_edges

Η συνάρτηση read_edges αποτελεί μια συνάρτηση τύπου void, η οποία δέχεται ως είσοδο από τον χρήστη τις ακμές του γραφήματος εισόδου. Κάνει χρήση ενός δισδιάστατου πίνακα ακεραίων, που αποθηκεύει τη γειτνίαση των κόμβων του γραφήματος εισόδου από τον χρήστη (int **a), καθώς επίσης τον ακέραιο (int size) που αποτελεί το πλήθος των κόμβων του γραφήματος. Στόχος της συνάρτησης είναι να διαβάσει από τον χρήστη τον αριθμό των κόμβων του γραφήματος, τις επιτρεπτές ακμές που μπορούν να αποτελέσουν το γράφημα και τέλος να αρχικοποιήσει τον πίνακα γειτνίασης με τιμές ένα ή μηδέν. Η τιμή 1 υποδηλώνει ότι ο κόμβος i είναι γείτονας με τον κόμβο j, ενώ η τιμή 0 ότι δεν είναι γείτονας.

```
void  read_edges(int **a, int size){
        int e_num, i, j, v1, v2;
        /* initialize adjacency matrix */
        for (i=size; --i >= 0;  )
                for (j=size; --j >= 0;  )
                     a[i][j] = 0;
                /* read edges and update adjacency matrix */
                printf("Give the number of edges: ");
                scanf("%d", &e_num);
```

```
        for (i=0; i < e_num; ++i){
                printf("Give the endpoint ids of edge %d: ", i+1);
                scanf("%d %d", &v1, &v2);
                if (v1 < 0 || v1 >= size || v2 < 0 || v2 >= size || v1 == v2){
                        printf("Illegal edge. Ignored...\n");
                        continue;
                }
                if (a[v1][v2] != 0){
                        printf("Duplicate edge. Ignored...\n");
                        continue;
                }
                a[v1][v2] = a[v2][v1] = 1;
        }
}
```

3.3.4. Η Συνάρτηση print_adj_matrix

Η συνάρτηση print_adj_matrix αποτελεί μια συνάρτηση τύπου void που δέχεται ως είσοδο τον δισδιάστατο πίνακα γειτνίασης (int **a) και το ακέραιο μέγεθος του (int size). Η λειτουργία της συνάρτησης είναι να τυπώνει τον πίνακα γειτνίασης του γραφήματος που δέχεται.

```
void print_adj_matrix(int **a, int size)
{
        int i, j;
        printf("         ");
        for (i=0; i < size; ++i)
                printf(" %2d", i);
        printf("\n");
           for (i=0;  i < size; ++i){
                printf("vertex %2d: ", i);
                for (j=0;  j < size; ++j)
                        printf(" %2d", a[i][j]);
                printf("\n");
        }
}
```

3.3.5. Η Συνάρτηση init_ordering

Η συνάρτηση init_ordering είναι μια συνάρτηση τύπου void. Δέχεται ως ορίσματα τον πίνακα που περιλαμβάνει την διάταξη των κόμβων (int *s) και την ακέραια μεταβλητή με το μέγεθος του πίνακα (int size). Αρχικοποιεί τον πίνακα με την διάταξη s. Ο πίνακας αρχικοποιείται ως εξής: αν έχουμε ένα γράφημα για παράδειγμα 5 κόμβων (0, 1, 2, 3, 4) τότε s[4]=4, s[3]=3, κλπ. Η διάταξη των κόμβων χρησιμοποιείται ακολούθως στην ldfs_plus αναζήτηση.

```
void init_ordering(int *s, int size){

        int i;

        for (i=size; --i >= 0; )

                s[i] = i;

}
```

3.3.6. Η Συνάρτηση print_int_array

Η συνάρτηση print_int_array είναι τύπου void και παίρνει ορίσματα ένα μονοδιάστατο πίνακα ακεραίων (int *a) και το μήκος του πίνακα (int size). Η λειτουργία της συνάρτησης είναι να τυπώνει τον πίνακα εισόδου.

```
void print_int_array(int *a, int size){

        int i;

        for (i=0; i < size; ++i)

                printf(" %2d", a[i]);

        printf("\n");

}
```

3.3.7. Η Συνάρτηση get_ordering

Η συνάρτηση get_ordering είναι τύπου void. Παίρνει ως ορίσματα τον πίνακα γειτνίασης του γραφήματος εισόδου (int **a), το πλήθος των κορυφών του γραφήματος και τον πίνακα που φυλάσσει την ενημερωμένη διάταξη των κόμβων (int *ordering). Σε αυτό το σημείο ελέγχουμε αν ένα γράφημα είναι comparability μέσω του incomparability γραφήματος. Για την αναπαράσταση του incomparability γραφήματος, όπου για παράδειγμα η κατευθυνόμενη ακμή ab είναι γειτονική με την κατευθυνόμενη bc, χρησιμοποιούμε έναν 3-διάστατο πίνακα γειτνίασης incomp[N][N][N], όπου N το πλήθος των κόμβων του αρχικού δοθέντος γραφήματος. Τότε για τις ακμές ab και bc, έχουμε incomp[a][b][c] = 1. Η λειτουργία της συνάρτησης είναι να δημιουργεί μια incomparability διάταξη του γραφήματος, να ελέγχει αν αυτό είναι διμερές και να κάνει τοπολογική ταξινόμηση στο συμπληρωματικό γράφημα του γραφήματος εισόδου. Αρχικά δεσμεύουμε δυναμικά χώρο για το

incomparability γράφημα. Μέσα στη συνάρτηση καλούμε τη μέθοδο init_incomp_graph, η οποία αρχικοποιεί το incomparability γράφημα. Στη συνέχεια ελέγχουμε αν αυτό είναι διμερές μέσω της κλήσης της μεθόδου check_bipartite. Έπειτα τυπώνουμε τον πίνακα γειτνίασης του incomparability γραφήματος. Ακολούθως κάνουμε τοπολογική ταξινόμηση με τη βοήθεια της μεθόδου topological_sort, χρησιμοποιώντας τον ενημερωμένο πίνακα γειτνίασης που περιέχει πληροφορίες για το γράφημα εισόδου αλλά και για την ανάθεση των ακμών του συμπληρωματικού του. Τέλος, αποδεσμεύουμε τον πίνακα του incomparability γραφήματος με την χρήση της συνάρτησης free().

```
void get_ordering(int **a, int size, int *ordering){

        int i, j, ***incomp;


        /* allocate space for incomparability adjacency matrix */

        incomp = (int ***)malloc(size * sizeof(int **));

        if (incomp == NULL){

                printf("\nNo space for incomparability graph. Aborting...\n");

                exit(-1);

        }

        for (i=size; --i >= 0;  ){

                incomp[i] = (int **)malloc(size * sizeof(int *));

                if (incomp[i] == NULL){

                        printf("\nNo space for incomparability graph. Aborting...\n");

                        exit(-1);

                }

                for (j=size; --j >= 0;  ){

                        incomp[i][j] = (int *)malloc(size * sizeof(int));

                        if (incomp[i][j] == NULL){

                                printf("\nNo space for incomparability graph. Aborting...\n");

                                exit(-1);

                        }

                }

        }


        init_incomp_graph (incomp, size, a);

        printf("\nInitialization of incomparability graph complete.\n");

        printf("\nThe adjacency matrix of the incomp-graph (v_num = %d) is:\n", size);

        print_incomp_adj_matrix(incomp, size);
```

```
        check_bipartite (incomp, size, a);
        printf("\nThe incomparability graph is bipartite.\n");
        printf("\nThe updated adjacency matrix of the given graph is:\n");
        print_adj_matrix(a, size);

        topological_sorting(a, size, ordering);
        printf("\nTopological sorting of complement graph complete.\n");
         /* change 2s to 0s in adjacency matrix of given graph */
        for (i=size; --i >= 0;  ){
                for (j=size; --j >= 0;  )
                        if (a[i][j] == 2)
                          a[i][j] = 0;
        }
        printf("\nThe cleaned adjacency matrix of the given graph is:\n");
        print_adj_matrix(a, size);

        /* deallocate space for incomparability adjacency matrix */
        for (i=size; --i >= 0;  ){
                for (j=size; --j >= 0;  )
                        free(incomp[i][j]);
                free(incomp[i]);
        }
        free(incomp);
}
```

3.3.8. Η Συνάρτηση init_incomp_graph

Η συνάρτηση init_incomp_graph είναι μια συνάρτηση τύπου void. Λαμβάνει ως ορίσματα ένα τρισδιάστατο πίνακα ακέραιων τιμών που αποθηκεύει το βοηθητικό incomparability γράφημα (int ***incomp), μια ακέραια τιμή με το πλήθος των κόμβων του γραφήματος (int v_num) και τον πίνακα γειτνίασης του γραφήματος εισόδου (int **a). Δημιουργεί δοθέντος του συμπληρωματικό γραφήματος της εισόδου, ένα γράφημα που κάθε κορυφή του είναι ο συνδυασμός δύο κορυφών του αρχικού γράφου. Αν ένας συνδυασμός κόμβων του συμπληρωματικού γραφήματος υπάρχει και συνορεύει με κάποιον άλλο συνδυασμό κόμβων, τότε υπάρχει ακμή στο incomparability γράφημα και γίνεται ενημέρωση του πίνακα incomp.

```
void init_incomp_graph(int ***incomp, int v_num, int **a){
        int i, j, k;

        for (i = v_num; --i >= 0;  )
                for (j = v_num; --j >= 0;  )
                        for (k = v_num; --k >= 0;  )
                                /* add edge (i,j)-(j,k) in auxiliary graph */
                                /* if {i,j}, {j,k} not edges in given graph*/
                                /* and                          */
                                /* either i=k or {i,k} edge in given graph */
                                if (a[i][j] == 0 && i != j
                                 && a[j][k] == 0 && j != k
                                 && (i == k || a[i][k] == 1) )
                                        incomp[i][j][k] = 1;
                                else
                                        incomp[i][j][k] = 0;
}
```

3.3.9. Η Συνάρτηση print_incomp_adj_matrix

Η μέθοδος print_incomp_adj_matrix είναι μία void συνάρτηση. Παίρνει ως ορίσματα έναν τρισδιάστατο πίνακα ακεραίων που φυλάσσει το incomparability γράφημα (int ***m) και το ακέραιο πλήθος των κόμβων του γραφήματος εισόδου (int size). Η λειτουργία της είναι η εκτύπωση στην οθόνη του βοηθητικού πίνακα που περιλαμβάνει το incomparability γράφημα.

```
void print_incomp_adj_matrix(int ***m, int size){
        int i, j, k;
        printf("          ");
        for (i=0; i < size; ++i)
           printf(" %2d", i);
        printf("\n");
         for (i=0;  i < size; ++i)
                for (j=0;  j < size; ++j){
                        printf("edge %2d-%2d: ", i, j);
                         for (k=0;  k < size; ++k)
```

```
                printf(" %2d", m[i][j][k]);
            printf("\n");
        }
}
```

3.3.10. Η Συνάρτηση check_bipartite

Η συνάρτηση check_bipartite είναι τύπου void. Στην συνάρτηση εισάγονται ως ορίσματα το incomparability γράφημα (int ***incomp), το πλήθος των κόμβων (int v_num), και ο πίνακας γειτνίασης του γραφήματος εισόδου (int **a). Αρχικά δεσμεύουμε δυναμικά τον απαιτούμενο χώρο για τον πίνακα visited. Στην συνέχεια αρχικοποιούμε τον πίνακα visited ως εξής : για κάθε στοιχείο που βρίσκεται στη διαγώνιο του δισδιάστατου πίνακα (δηλαδή για τα στοιχεία (1,1) (2,2), …) δίνουμε στον πίνακα visited την τιμή -2 (αφού δεν είναι επιθυμητό να υπάρχει κόμβος που να ενώνεται με μια ακμή με τον εαυτό του). Για κάθε συνδυασμό των (i, j) αν δεν σχηματίζεται ακμή στον πίνακα γειτνίασης, τότε αρχικοποιούμε το visited για τους δείκτες (i, j) καθώς και για τους συμμετρικούς του (j, i) με την τιμή μηδέν. Σε αντίθετη περίπτωση αρχικοποιούμε τον πίνακα visited για τους δείκτες (i, j) καθώς και για τους συμμετρικούς του (j, i) με την τιμή μείον δύο. Στην συνέχεια κάνουμε dfs αναζήτηση στο βοηθητικό incomparability γράφημα. Έπειτα συλλέγουμε τις κατευθύνσεις των ακμών του δοθέντος γραφήματος και τις αποθηκεύω σε έναν πίνακα a. Τέλος αποδεσμεύουμε τον πίνακα visited.

```
void check_bipartite(int ***incomp, int v_num, int **a)
{
        int i, j, **visited;
        /* allocate space for array visited */
        visited = (int **)malloc(v_num * sizeof(int *));
        if (visited == NULL){
                printf("\nNo space for incomparability graph. Aborting...\n");
                exit(-1);
        }
        for (i=v_num; --i >= 0;){
                visited[i] = (int *)malloc(v_num * sizeof(int));
                if (visited[i] == NULL){
                        printf("\nNo space for incomparability row. Aborting...\n");
                        exit(-1);
                }
        }
```

```
        /* initialize array visited for dfs on incomparability graph */
        for (i = v_num; --i >= 0;  ){
                visited[i][i] = -2;   /* no such node in aux. graph */
                for (j = i; --j >= 0;  )
                        if (a[i][j] == 0)   /* no edge in given graph */{
                                visited[i][j] = visited[j][i] = 0;
                        }
                        else    /* no such node in auxiliary graph */
                                visited[i][j] = visited[j][i] = -2;
        }

        /* run DFS on the (auxiliary) incomparability graph */
        for (i = v_num; --i >= 0;  )
                /* check for j < i; (i,j) and (j,i) are adjacent */
                for (j = i; --j >= 0;  )
                        if (visited[i][j] == 0)
                                dfs(incomp, v_num, i, j, visited, 1);
        /* collect edge orientations of given graph; store in a */
        /* note that the corresponding entries of array a      */
        /* correspond to non-edges of the given graph and thus  */
        /* these entries initially have values equal to 0      */
        for (i = v_num; --i >= 0;  )
                for (j = i; --j >= 0;  )
                        switch (visited[i][j]){
                        case  1: a[i][j] = 2;
                                  break;
                        case -1: a[j][i] = 2;
                                  break;
                        }
        /* de-allocate auxiliary space for array viisted */
        for (i = v_num; --i >= 0;  )
                free(visited[i]);
        free(visited);
}
```

3.3.11. Η Συνάρτηση dfs

Η συνάρτηση dfs είναι μια συνάρτηση τύπου void. Δέχεται ως ορίσματα τον πίνακα που περιλαμβάνει το incomparability γράφημα (int ***incomp), το πλήθος των κόμβων του γραφήματος εισόδου, δυο ακέραιες μεταβλητές που περιλαμβάνουν τους κόμβους επιλογής, έναν δισδιάστατο βοηθητικό πίνακα ακεραίων που εκφράζει αν επισκεφθήκαμε κάποιον κόμβο ή όχι (int **visited) και μία βοηθητική μεταβλητή (int k). Η συνάρτηση κάνει αναζήτησης εις βάθος στο incomparability γράφημα. Αυτό πραγματοποιείται γιατί επιθυμούμε να ελέγξουμε το incomparability γράφημα είναι διμερές.

```
void dfs(int ***incomp, int v_num, int v1, int v2, int **visited, int k){

        int i, t;


        visited[v1][v2] = k;   /* visited node; bipart.-set <-- k */


        for (i = v_num; --i >= 0;  )

                if (incomp[v1][v2][i] == 1){

                        t = visited[v2][i];

                        if (t == 0)   /* non-visited node */

                                dfs(incomp, v_num, v2, i, visited, -k);

                        else

                                if (t == k){

                                        printf("Graph is not bipartite ");

                                        printf("(v1 = %d, v2 = %d, i = %d)\n", v1, v2, i);

                                        exit(-2);

                                }

                }

}
```

3.3.12. Η Συνάρτηση topological_sorting

Η συνάρτηση topological_sorting είναι μία συνάρτηση τύπου void. Δέχεται ως ορίσματα ένα δισδιάστατο πίνακα γειτνίασης (int **a), το πλήθος των κόμβων του γραφήματος (int v_num), και τον πίνακα με την διάταξη των κόμβων του γραφήματος (int *ordering). Αρχικά δεσμεύουμε δυναμικά χώρο για τον πίνακα outdegree, που αποθηκεύει τον αριθμό των ακμών που ξεκινάνε από κάθε κόμβο. Έπειτα υπολογίζουμε το out-degree του κάθε κόμβου και το αποθηκεύω στον πίνακα outdegree. Αποθηκεύουμε στο ordering πρώτα τους κόμβους με μηδενικό out-degree. Στη συνέχεια σε κάθε βήμα τυπώνουμε τον

ενημερωμένους πίνακες outdegree και ordering. Η επεξεργασία των κόμβων γίνεται ως εξής : αρχικά τοποθετούμε αριστερά στη διάταξη ordering τους κόμβους με μηδενικό out-degree. Όσο υπάρχουν κόμβοι που δεν έχουν εισαχθεί στην διάταξη ordering (δηλαδή δεν έχουν ταξινομηθεί ακόμα) για κάθε κόμβο που συνορεύει με κάποιον άλλον και η ακμή που δημιουργείται από το συνδυασμό τους είναι κατευθυνόμενη (π.χ. για τους κόμβους i, j να έχουμε a[i][j] = 2, η τιμή δύο εκφράζει την κατευθυνόμενη ακμή στον ενημερωμένο πίνακα γειτνίασης), αν το out-degree > 0 τότε είτε το out-degree = 0 είτε το out-degree > 1. Σε περίπτωση που το out-degree το κόμβου ισοδυναμεί με την τιμή 1 τότε ενημερώνεται η διάταξη με την τιμή του κόμβου κ προχωράμε παρακάτω. Σε περίπτωση που είναι μεγαλύτερη του 1 η τιμή του τότε συνεχίσουμε και ψάχνουμε τους υπόλοιπους κόμβους που γειτονεύουν με αυτόν. Τέλος σε περίπτωση που δεν εισαχθούν όλοι οι κόμβοι στην διάταξη μας, τότε ο γράφος δεν είναι κατευθυνόμενος και δεν εμφανίζει κύκλο.

```
void topological_sorting(int **a, int v_num, int *ordering){

        int i, j, t, counter, *outdegree;


        /* allocate space for auxiliary outdegree array */

        outdegree = (int *)malloc(v_num * sizeof(int));

        if (outdegree == NULL){

                printf("\nNo space for outdegree matrix. Aborting...\n");

                exit(-1);

        }


        /* compute out-degrees of nodes and store in outdegree array */

        counter = 0;

        for (i = v_num; --i >= 0;  ){

                t = 0;

                for (j = v_num; --j >= 0;  )

                        if (a[i][j] == 2)   /* oriented edge (i,j) */

                                ++t;

                outdegree[i] = t;


                /* collect nodes with outdegree 0 */

                if (t == 0)

                        ordering[counter++] = i;

        }

        printf("\nCurrent out-degrees: ");

        print_int_array(outdegree, v_num);
```

```
        printf("Current ordering: ");
        print_int_array(ordering, counter);
        if (counter == v_num)   /* all nodes in ordering */
                return;

        /* process nodes */
        for (i = 0; i < counter; ++i){
                t = ordering[i];   /* process node t */
                for (j = v_num; --j >= 0;  )
                        if (a[j][t] == 2   /* oriented edge (j,t) */
                         && outdegree[j] > 0){
                                if (--(outdegree[j]) == 0)
                                        ordering[counter++] = j;
                                printf("Current out-degrees: ");
                                print_int_array(outdegree, v_num);
                                printf("Current ordering: ");
                                print_int_array(ordering, counter);
                                if (counter == v_num)
                                        return;  /* all nodes in ordering */
                        }
        }
        /* IMPORTANT: if we get here, then NOT ALL nodes in ordering */
        printf("\nThe oriented graph is not a DAG! Aborting...\n");
        exit(-3);
}
```

3.3.13. Η Συνάρτηση print_temp_ordering

Η συνάρτηση print_temp_ordering είναι μια συνάρτηση τύπου void. Δέχεται ως είσοδο έναν πίνακα ακεραίων s (int *s), που περιλαμβάνει τη διάταξη των κόμβων, τον ακέραιο size και ένα ακέραιο i. Η συνάρτηση τυπώνει τα βήματα που εκτελούνται κατά το βήμα της $LDFS^+$ επεξεργασίας της διάταξης. Ο ακέραιος i εκφράζει τον κόμβο στον οποίο εφαρμόζεται ο $LDFS^+$ αλγόριθμος κάθε φορά.

```
void print_temp_ordering(int *s, int size, int i){
        int t;
        printf("\nstep %2d: ", i);
        for (t=0; t < i; ++t)
           printf(" %2d", s[t]);
        printf(" [%2d]", s[i]);
        for (t=i; ++t < size; )
            printf(" %2d", s[t]);
        printf("\n");
}
```

3.3.14. Η συνάρτηση ldfs_plus

Η συνάρτηση ldfs_plus, είναι μια συνάρτηση τύπου void. Δέχεται ως είσοδο τον δισδιάστατο πίνακα γειτνίασης a, μια μεταβλητή τύπου ακεραίου που περιλαμβάνει το πλήθος των κόμβων, καθώς επίσης ένα πίνακα ακεραίων s που περιλαμβάνει την διάταξη εισόδου. Η συνάρτηση αρχικά αντιστρέφει την διάταξη εισόδου s, έτσι ώστε να χρησιμοποιήσει στη συνέχεια πρώτα τον δεξιότερο κόμβο που αυτή περιλαμβάνει και πάντα τους δεξιότερους επόμενους κόμβους. Έπειτα για κάθε κόμβο που περιλαμβάνεται στην διάταξη s, βρίσκει τους γείτονες του και τους αποθηκεύει σε μία προσωρινή διάταξη. Τέλος μετατοπίζει τους μη γείτονες του κόμβου προς τα δεξιά της διάταξης και τους γείτονες στην αριστερά της διάταξης. Σε κάθε βήμα τυπώνει την τρέχουσα διάταξη.

```
void ldfs_plus(int **a, int v_num, int *s){
        int i, j, v, t, *temp_n, count_n;

        /* allocate space for temp_n */
        temp_n = (int *)malloc(v_num*sizeof(int));
        if (temp_n == NULL){
                printf("\nNo space for ldfs. Aborting...\n");
                exit(-1);
        }

        /* reverse the order in s */
        for (i=0, j=v_num; --j > i;  ++i){
                t = s[i];
                s[i] = s[j];
                s[j] = t;
        }
```

```
        /* reorder s[i+1,...,v_num-1] by prepending neighbors of s[i] */
        /* maintain relative ordering of neighbors and non-neighbors  */
        for (i=0; i < v_num; ++i){
                v = s[i];

                /* collect neighbors of v in temp_n */
                for (count_n=0, t=i; ++t < v_num;  )
                if (a[v][s[t]] == 1)
                        temp_n[count_n++] = s[t];

                /* shift non-neighbors of v to the end of s */
                /* first, traverse non-neighbors of v at the end of s */
                for (t=v_num; --t > i;  )
                        if (a[v][s[t]] == 1)
                                break;
                /* shift remaining non-neighbors at the end of s */
                for (j=t; --t > i;  )
                        if (a[v][s[t]] == 0)
                                s[j--] = s[t];
                /* finally, copy neighbors at the beginning of s */
                if (i + count_n != j)
                        printf("*** ERROR in constructing new ordering! ***\n");
                for (; --count_n >= 0;  )
                        s[j--] = temp_n[count_n];

                /* print current oredring */
                print_temp_ordering(s, v_num, i);
        }

        free(temp_n);
}
```

3.3.15. Η Συνάρτηση compute_paths

Η συνάρτηση compute_paths είναι μια συνάρτηση τύπου void. Παίρνει ως ορίσματα έναν δισδιάστατο πίνακα ακεραίων που περιέχει τις πληροφορίες γειτνίασης για όλους τους κόμβους, ένα ακέραιο που περιέχει το πλήθος των κόμβων του γραφήματος που εξετάζουμε, ένα πίνακα με τη προ-επεξεργασμένη διάταξη των κόμβων, ένα 4-διάστατο πίνακα που φυλάσσονται τα μονοπάτια και ένα τρισδιάστατο πίνακα length που περιέχει το μήκος των μονοπατιών. Είναι η μέθοδος στην οποία υπολογίζονται τα μήκη των μονοπατιών ολόκληρου του γραφήματος. Αρχικά εκτελούμε τον αλγόριθμο ldfs_plus για να πάρουμε την επεξεργασμένη διάταξη των κόμβων. Έπειτα προσθέτουμε ένα απομονωμένο dummy κόμβο στην προ-επεξεργασμένη διάταξη των κόμβων. Στη συνέχεια για τον δείκτη i από n (όπου n το πλήθος των κόμβων) μέχρι 1, για κάθε δείκτη j που ισούται με τον δείκτη i και για όσο ο δείκτης j είναι μικρότερος από το πλήθος των κόμβων, στην περίπτωση όπου ui ∈ V(G(i, j)), ο αλγόριθμος αρχικοποιεί στην γραμμή 8 οι τιμές path(ui, i, j) = (ui) και length(ui, i, j) = 1. Αλλιώς, στην περίπτωση όπου το ui ∈ V (G((i, j)) , ο αλγόριθμος δεν εκτελεί τη γραμμή 8, δεδομένου ότι οι τιμές path(ui, i, j, k) και length(ui, i, j) δεν μπορούν να προσδιοριστούν. Στη συνέχεια σε έναν πίνακα τεσσάρων διαστάσεων, όπου για κάθε ζεύγος δεικτών i, j ∈ {1, 2,…, n} και για κάθε κόμβο u_y ∈ V(G(i, j)) η εγγραφή path(u_y; i, j, k) αποθηκεύει τη διάταξη των κόμβων που ανήκουν στο μέγιστο μονοπάτι του *G(i, j)* με τον u_y να αποτελεί τον τελευταίο κόμβο του μονοπατιού και το k την θέση του κόμβου στο μονοπάτι. Το μήκος του μονοπατιού path(u_y; i, j, k) αποθηκεύεται στην εγγραφή length(u_y; i, j). Κατά την αρχικοποίηση ενός συγκεκριμένου ζεύγους δεικτών {i, j} (βλ. γραμμές 5-8, Αλγόριθμος 2), επιθυμούμε να αρχικοποιήσουμε τα μονοπάτια που δεν χρησιμοποιούν τον κόμβο u_i ως ενδιάμεσο κόμβο. Ένα μονοπάτι που έχει το u_y ∈ V(G(i+1, j)) ως τον τελευταίο του κόμβο, θα αποθηκευθεί στην εγγραφή path(u_y; i+1, j, k). Για ένα μονοπάτι με το u_i ως τον τελευταίο του κόμβο, δίνουμε σημασία μόνο στην περίπτωση που το u_i ∈ V(G(i, j)), και έτσι το αρχικοποιούμε ως εξής: path(u_k; i, j, 0) = (u_i). Στην συνέχεια, (βλ. γραμμές 9-17) επεκτείνουμε τις εγγραφές του πίνακα με την προσθήκη του κόμβου u_i για την περίπτωση που u_i ∈ V(G(i,j)). Πρώτα, ελέγχουμε αν ένα μονοπάτι P των G(i,j) που εμπεριέχει τον κόμβο u_i έχει τουλάχιστον δύο κόμβους. Αν ναι, τότε το P πρέπει να εμπεριέχει έναν κόμβο u_x ∈ V(G(i+1, j)) με u_xu_i ∈ E. Για κάθε κόμβο u_x, υπάρχουν δύο διαφορετικοί ρόλοι που μπορεί να παίξει για να υπολογίσει το πιθανό μέγιστο μονοπάτι για να αποθηκευθεί στον πίνακα. Πρώτα, προσθέτοντας μια ακμή u_xu_i σε ένα μέγιστο μονοπάτι του G(i+1,j) με u_x τον τελευταίο κόμβο μπορεί να δημιουργήσει ένα μονοπάτι με τον κόμβο u_i ως τον τελευταίο του κόμβο, το οποίο είναι μεγαλύτερο από το τρέχον αποθηκευμένο στην εγγραφή path(u_i; i, j, k). Αυτή την περίπτωση περιγράφεται στις γραμμές 11-13. Ο δεύτερος ρόλος που μπορεί να αποτελέσει ο κόμβος u_x είναι να εξυπηρετήσει ως τον συνδετικό κρίκο ανάμεσα στις δύο ακμές u_xu_i και $u_iu_{y'}$. Αυτή η κατάσταση περιγράφεται στις γραμμές 14-17.

```
void compute_paths(int **a, int v_num, int *s, int ****path, int ***length){

        int i, j, y, k, x, t, value;

        char *ptr;


        ldfs_plus(a, v_num, s);
```

```
/* 2: add isolated dummy vertex un+1 to ó=(u1,u2,...,un,un+1) */
/* there is no need to add the extra vertex if we note that: */
/* uy in V(G(i+1,j))  where  i+1 <= y <= j                */
/*              <===>  j == v_num-1 || a[s[y]][s[j+1]] == 0 */
/* ui in V(G(i,j))  <===>  j==v_num-1 || a[s[i]][s[j+1]] == 0 */
/* 3: for i = n downto 1 do */
for (i = v_num; i >= 0; --i){
        printf("i = %d \n", i);
        /* 4: for j = i to n do */
        for (j = i; j < v_num; ++j){
                printf(" j = %d \n", j);
                /* NOTE: I execute steps 7-8 before steps 5-6 */
                /* 7: if ui in V(G(i, j)); note that i <= j */
                if (j == v_num-1 || a[s[i]][s[j+1]] == 0){
                        printf("  ui in V(G(i,j)) \n");
                        path[i][i][j][0] = s[i];
                        length[i][i][j] = 1;
                }
                if (i < j)  /* i < j  <==> i+1 <= j */{
                /* G(i + 1, j) not empty */
                  printf("  G(i+1,j) is not empty \n");
                        /* 5: for every uy in V(G(i + 1, j)) do */
                        for (y = i+1; y <= j ; ++y){
                                printf("   y = %d \n", y);

                                /* check if uy in V(G(i+1,j)) */
                                if (j == v_num-1 || a[s[y]][s[j+1]] == 0){
                                        printf("    uy in V(G(i+1,j)) \n");
                                        /* 6: P(uy;i,j) <- P(uy;i+1,j); */
                                        /* l(uy;i,j) <- l(uy;i+1,j); */
                                        length[y][i][j] = k = length[y][i+1][j];
                                        for ( ; --k >= 0;  )
                                                path[y][i][j][k] = path[y][i+1][j][k];
                                }
```

```
}


/* 9: for every ux in V(G(i+1, j)) do */
for(x = i+1; x <= j; ++x){
                printf("   x = %d \n", x);
                /* 10: if ui in V(G(i, j)) ... */
                if ( (j == v_num-1 || a[s[i]][s[j+1]] == 0)/* 10: ... and ux in N(ui) */
                  && a[s[x]][s[i]] == 1 ){
                                printf("    ui in V(G(i,j) and ux in N(ui) \n");
                                /* 11: if l(ui;i,j) < l(ux;i+1,j)+1 */
                                if (length[i][i][j] <  length[x][i+1][j] + 1){
                                                printf("     update path and length \n");
                                                /* 12: P(ui;i,j) <- (P(ux;i+1,j),ui); */
                                                /* 13: l(ui;i,j) <- l(ux;i+1,j) + 1; */
                                                length[i][i][j] = k = length[x][i+1][j] + 1;
                                                path[i][i][j][--k] = s[i];
                                                for ( ; --k >= 0;  )
                                                                path[i][i][j][k] = path[x][i+1][j][k];
                                }


                 /* 14: for every uy in V(G(i+1, x-1)) do */
                for(y = i+1; y < x; ++y){
                                printf("    y = %d \n", y);
                                /* 15: if l(uy;i,j) < l(ux;i+1,j)+l(uy;i+1,x-1)+1 */
                            if (length[y][i][j] < length[x][i+1][j]+ length[y][i+1][x-1] + 1)
                                                {
                                                 printf("     update path and length \n");
                                                /*16: P(uy;i,j) <-(P(ux;i+1,j),ui,P(uy;i+1,x-1));*/
                                                /* 17: l(uy;i,j) <- l(ux;i+1,j)+l(uy;i+1,x-1) + 1; */
                                                k = length[y][i+1][x-1];
                                                length[y][i][j] = t = k + length[x][i+1][j] + 1;
                                                for ( ; --k >= 0;  )
                                                    path[y][i][j][--t] = path[y][i+1][x-1][k];
                                                k = length[x][i+1][j];
```

```
                                                path[y][i][j][k] = s[i];
                                                for ( ; --k >= 0;  )
                                                        path[y][i][j][k] = path[x][i+1][j][k];
                                                }
                                        }
                                }
                        }
                }
        }
}
```

3.3.16. Η Συνάρτηση compute_max_path_legth

Η συνάρτηση compute_max_path_length είναι μια συνάρτηση τύπου int και παίρνει ως ορίσματα τον τρισδιάστατο πίνακα length που περιλαμβάνει τα μήκη των μονοπατιών που σχηματίζονται με συνδυασμούς των κόμβων, καθώς επίσης και το ακέραιο πλήθος των κόμβων του γραφήματος. Επιστρέφει μονοπάτι με το μεγαλύτερο μήκος, δηλαδή το μονοπάτι του γραφήματος που περιλαμβάνει το μεγαλύτερο πλήθος των κόμβων του γραφήματος. Για κάθε συνδυασμό των κόμβων του γραφήματος αν το τρέχον μήκος είναι μεγαλύτερο από το μήκος της προηγούμενης εγγραφής τότε ανανεώνεται και φυλάσσεται σε μια μεταβλητή ακέραιου τύπου (int cur_max_legth).

```
int compute_max_path_length(int ***length, int v_num)
{
        int i, j, m=v_num-1, cur_max_length=0;
        for (i=v_num; --i >= 0;  )
        {
                for (j=v_num; --j >= 0;  )
                {
                        if (length[i][j][m] > cur_max_length)
                                cur_max_length = length[i][j][m];
                }
        }
        return(cur_max_length);
}
```

3.3.17. Η Συνάρτηση print_max_paths

Η συνάρτηση print_max_paths είναι μια συνάρτηση τύπου void, που δέεται ως ορίσματα έναν τετραδιάστατο πίνακα που περιέχει τα μονοπάτια (int ****path), έναν τρισδιάστατο πίνακα που περιλαμβάνει τα μήκη των μονοπατιών (int ***length), το ακέραιο πλήθος των κόμβων (int v_num) και το μέγιστο μήκος μονοπατιού (int max_length). Για κάθε συνδυασμό των κόμβων αν το μήκος του μονοπατιού που σχηματίζεται από τους δείκτες i, j, m είναι ίσο με το μέγιστο μήκος τότε τυπώνει το μονοπάτι.

```
void print_max_paths(int ****path, int ***length, int v_num, int max_length)

{

        int i, j, k, t, m=v_num-1;

        for (i=0; i < v_num; ++i){

                for (j=0; j < v_num; ++j){

                        if (length[i][j][m] == max_length){

                                printf("\nPath[%2d;%2d,%2d] = ", i, j, m);

                                for (k=0; k < max_length; ++k)

                                        printf("%2d ", path[i][j][m][k]);

                                printf("\n");

                        }

                }

        }

}
```

3.3.18. Η Συνάρτηση main

Η συνάρτηση main είναι τύπου int. Η πρώτη εντολή που εκτελείται στο εσωτερικό της, εμφανίζει ένα μήνυμα που ζητά από τον χρήστη να εισάγει τον αριθμό των κόμβων που επιθυμεί και έπειτα διαβάζει από το πληκτρολόγιο την επιλογή του χρήστη. Στη συνέχεια δεσμεύει χώρο (ανάλογα με τον αριθμό των κόμβων που εισάγεται), για τον πίνακα γειτνίασης του γραφήματος, για τον πίνακα που περιλαμβάνει τη διάταξη των κόμβων, για τον πίνακα με τα μονοπάτια και για τον πίνακα με τα μήκη των μονοπατιών. Αυτό συμβαίνει με την κλήση της συνάρτησης allocate_space. Έπειτα διαβάζει από τον χρήστη τους κόμβους, που αποτελούν τις κορυφές των ακμών του γραφήματος, με την βοήθεια της συνάρτησης read_edge. Στη συνέχεια καλείται η print_adj_matrix και τυπώνει στην οθόνη τον πίνακα γειτνίασης του εισαγόμενου γραφήματος. Μετά η συνάρτηση init_ordering αρχικοποιεί σε αύξουσα σειρά την διάταξη των κόμβων τους πίνακα σίγμα (που χρησιμοποιείται στην compute_paths) και η print_int_array την τυπώνει. Η κλήση της συνάρτησης get_ordering μας δίνει μια incomparability διάταξη των κόμβων του υπάρχοντος γραφήματος. Δηλαδή μιας συμπληρωματικής διάταξης ενός comparability γραφήματος (cocomparability). Αποδεικνύεται ότι η συγκεκριμένη διάταξη συμπίπτει με τη διάταξη που προκύπτει, από μια transitive ανάθεση κατευθύνσεων στις ακμές του συμπληρωματικού γραφήματος (το οποίο

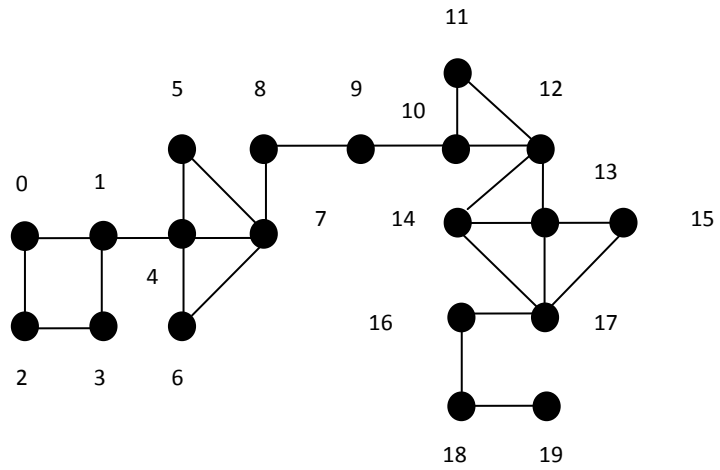
είναι comparability). Μια ανάθεση κατευθύνσεων στις ακμές ενός γραφήματος είναι transitive εάν στο προκύπτον κατευθυνόμενο γράφημα: εάν πρώτον έχουμε ακμές ab, bc και δεν υπάρχει η ακμή ac, τότε οι ακμές δεν μπορεί να έχουν "ομόρροπες" φορές, δηλαδή, δεν μπορεί και οι δύο είτε να κατευθύνονται προς το b είτε να απομακρύνονται από το b και δεύτερον δεν επιτρέπεται να υπάρχει κατευθυνόμενος κύκλος. Ακολούθως, η print_int_array τυπώνει την ενημερωμένη διάταξη sigma, που περιλαμβάνει την incomparability διάταξη των κόμβων. Έπειτα η compute_paths υπολογίζει τα μονοπάτια και τα μήκη των μονοπατιών σύμφωνα με το πώς ορίζεται στον αλγόριθμο που υπολογίζει τα μονοπάτια και καλείται η compute_max_length για να υπολογιστεί το μέγιστος μήκος μονοπατιού σε ολόκληρο το γράφημα. Στη συνέχεια τυπώνουμε το μήκος καθώς επίσης και καλώντας της print_max_paths το μονοπάτι ή τα μονοπάτια που αντιστοιχούν σε αυτό. Τέλος αποδεσμεύουμε το χώρο που το πρόγραμμά μας έκανε χρήση, καλώντας την deallocate_space.

```
int main()

{

        int v_num, e_num, i, max_length;

        int *sigma, **adj, ****path, ***length;

        printf("Give the number of vertices: ");

        scanf("%d", &v_num);


        /* allocate space for auxiliary matrices */

        allocate_space(v_num, &adj, &sigma, &path, &length);


        read_edges(adj, v_num);

        printf("\nThe adjacency matrix of the graph (v_num = %d) is:\n", v_num);

        print_adj_matrix(adj, v_num);


        init_ordering(sigma, v_num);

        printf("\nThe initial ordering is:  ");

        print_int_array(sigma, v_num);


        get_ordering(adj, v_num, sigma);

        printf("\nThe new ordering is:  ");

        print_int_array(sigma, v_num);
```

```
        compute_paths(adj, v_num, sigma, path, length);


        max_length = compute_max_path_length(length, v_num);
        printf("\nMaximum path length = %d\n", max_length - 1);


        printf("\nLongest Paths\n");
        print_max_paths(path, length, v_num, max_length);



        /* deallocate space for auxiliary matrices */
        deallocate_space(v_num, adj, sigma, path, length);
}
```

3.4. Ενδεικτικές Εκτελέσεις του Αλγορίθμου

Σε αυτό το σημείο παραθέτουμε ένα παράδειγμα γραφήματος που μελετήσαμε. Έστω το παρακάτω γράφημα.



Σχήμα 3.4.1. Γράφημα εισόδου G = (20, 27)

Αρχικά εισάγουμε στο πρόγραμμα τα δεδομένα του γραφήματος μας. Δεσμεύουμε τον απαιτούμενο χώρο για τα δεδομένα (πίνακα γειτνίασης, βοηθητική συμβολοσειρά κτλ).

Ο πίνακας γειτνίασης του δοθέντος γραφήματος είναι ο παρακάτω:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
vertex 0:	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vertex 1:	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vertex 2:	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vertex 3:	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vertex 4:	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
vertex 5:	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
vertex 6:	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
vertex 7:	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
vertex 8:	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
vertex 9:	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
vertex 10:	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	0	0
vertex 11:	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
vertex 12:	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0
vertex 13:	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0
vertex 14:	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
vertex 15:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
vertex 16:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
vertex 17:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
vertex 18:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
vertex 19:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

Δημιουργούμε το incomparability γράφημα που προκύπτει από το γράφημα εισόδου και ελέγχουμε αν αυτό είναι διμερές. Ο έλεγχος μας δίνει ως αποτέλεσμα ότι το γράφημα είναι διμερές. Κάνουμε τοπολογική ταξινόμηση του incomparability γραφήματος. Σύμφωνα με την ακολουθία των κόμβων που προκύπτουν από την τοπολογική ταξινόμηση ενημερώνουμε την ακολουθία sigma, η οποία χρησιμοποιείται για την ταξινόμηση εις βάθος του cocomparability γραφήματος.

Η αρχική συμβολοσειρά (sigma) που χρησιμοποιείται για την ταξινόμηση εις βάθος από το πρόγραμμα μας είναι η εξής: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19.

Η καινούργια ενημερωμένη συμβολοσειρά που προκύπτει έπειτα από τη δημιουργία του incomparability γραφήματος, τον έλεγχο αν αυτό είναι διμερές και την τοπολογική ταξινόμηση των κόμβων που το συντελούν είναι: 1 0 2 3 7 5 4 6 9 8 12 11 10 14 13 17 15 18 16 19.

Ο αλγόριθμος LDFS$^+$ επιστρέφει την ενημερωμένη ακολουθία σ = (σ(1), σ(2), ... , σ(19)) των κόμβων του γραφήματος. Δοθέντος ενός γραφήματος G = (20, 27), αρχικά ο αλγόριθμος LDFS$^+$ εκχωρεί μια ταμπέλα με την κενή τιμή (*ε*) σε όλους του κόμβους του γραφήματος. Η επιλογή του κόμβου εκκίνησης γίνεται πάντα επιλέγοντας τον δεξιότερο μη αριθμημένο κόμβου της διάταξης εισόδου (για αυτόν τον λόγο και αντιστρέφουμε την σειρά της ακολουθίας εισόδου). Ο LDFS$^+$ αλγόριθμος επιλέγει πάντα το δεξιότερο και μη αριθμημένο κόμβο της διάταξης εισόδου και εκτελεί την λεξικογραφική διερεύνηση εις βάθος. Αν δύο υποψήφιοι κόμβοι έχουν τον ίδιο βαθμό επιλέγει πάλι τον δεξιότερο, διαφορετικά επιλέγει τον κόμβο με τον μεγαλύτερο βαθμό.

Αντιστρέφω την συμβολοσειρά sigma, αφού την εισάγω στον αλγόριθμο LDFS$^+$. Ξεκινώντας από το πρώτο στοιχείο (κόμβος 19), της ανεστραμμένης πλέον, συμβολοσειράς sigma, συλλέγουμε τους κόμβους που είναι γειτονικοί του κόμβου 19 (κόμβοι: 18 και 17). Οι κόμβοι που είναι γειτονικοί του μετακινούνται στην αρχή της συμβολοσειράς. Οι κόμβοι που δεν είναι γειτονικοί του μετακινούνται στο τέλος της συμβολοσειράς. Δηλαδή, όπως φαίνεται και στο βήμα 0, από την αρχική συμβολοσειρά 19 16 18 15 17 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1, μεταθέτουμε τους γειτονικούς κόμβους (18 και 17) που συναντάμε στην αρχή της συμβολοσειράς και τους μη γειτονικούς στο τέλος της.

Με τον ίδιο τρόπο ενημερώνουμε την LDFS+ συμβολοσειρά και στα υπόλοιπα βήματα. Αυτό στην πράξη φαίνεται παρακάτω. Εκτελώντας τον αλγόριθμο LDFS$^+$ παίρνω τα παρακάτω αποτελέσματα:

βήμα 0: [19] 18 17 16 15 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1

βήμα 1: 19 [18] 16 17 15 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1

βήμα 2: 19 18 [16] 17 15 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1

βήμα 3: 19 18 16 [17] 15 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1

βήμα 4: 19 18 16 17 [15] 13 14 10 11 12 8 9 6 4 5 7 3 2 0 1

βήμα 5: 19 18 16 17 15 [13] 14 12 10 11 8 9 6 4 5 7 3 2 0 1

βήμα 6: 19 18 16 17 15 13 [14] 12 10 11 8 9 6 4 5 7 3 2 0 1

βήμα 7: 19 18 16 17 15 13 14 [12] 10 11 8 9 6 4 5 7 3 2 0 1

βήμα 8: 19 18 16 17 15 13 14 12 [10] 11 9 8 6 4 5 7 3 2 0 1

βήμα 9: 19 18 16 17 15 13 14 12 10 [11] 9 8 6 4 5 7 3 2 0 1

βήμα 10: 19 18 16 17 15 13 14 12 10 11 [9] 8 6 4 5 7 3 2 0 1

βήμα 11: 19 18 16 17 15 13 14 12 10 11 9 [8] 7 6 4 5 3 2 0 1

βήμα 12: 19 18 16 17 15 13 14 12 10 11 9 8 [7] 6 4 5 3 2 0 1

βήμα 13: 19 18 16 17 15 13 14 12 10 11 9 8 7 [6] 4 5 3 2 0 1

βήμα 14: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 [4] 5 2 3 0 1

βήμα 15: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 [5] 2 3 0 1

βήμα 16: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 5 [2] 3 0 1

βήμα 17: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 5 2 [3] 1 0

βήμα 18: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 5 2 3 [1] 0

βήμα 19: 19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 5 2 3 1 [0]

Η τελική ακολουθία επιστροφής (sigma) της μεθόδου είναι η παρακάτω:

19 18 16 17 15 13 14 12 10 11 9 8 7 6 4 5 2 3 1 0

Έπειτα, με την συμβολοσειρά που μας επιστρέφει η εκτέλεση της μεθόδου LDFS^{+}, δεχόμαστε μία umbrella-free διάταξη που αποθηκεύουμε στης sigma.

Η μέθοδος compute_paths υπολογίζει όλους τους συνδυασμούς των πιθανών μονοπατιών του γραφήματος εισόδου. Για κάθε λοιπόν συνδυασμό των δύο δεικτών i και j (που παίρνουν τιμές από μηδέν μέχρι το πλήθος των κόμβων του γραφήματος) υπολογίζονται δυναμικά όλα τα μονοπάτια του γραφήματος.

Σε αυτό το σημείο είναι σημαντικό να σημειώσουμε πως πριν εκτελέσουμε με τη σειρά τα βήματα του αλγόριθμου εύρεσης του μεγίστου μονοπατιού, που παραθέσαμε προγενέστερα, κάνουμε πάντα τον εξής έλεγχο: για κάθε κόμβο u_i ($i \epsilon [0, 19]$) αν ο δείκτης i είναι μικρότερος ή ίσος του δείκτη j, ενημερώνουμε τον πίνακα που φυλάσσει τα μονοπάτια του γραφήματος με τον κόμβο που έχει την θέση i στη συμβολοσειρά sigma και επιπροσθέτως αρχικοποιούμε το τρέχον μήκος με την τιμή ένα. Αυτός ο κόμβος εισαγωγής αποτελεί τον πρώτο κόμβο που εισάγουμε στο εκάστοτε μονοπάτι και για αυτό τον λόγο έχει διαφορετική μεταχείριση.

Αυτό πρακτικά στην δική μας εκτέλεση γίνεται με τον εξής τρόπο: ξεκινώντας από τον δείκτη i με την τιμή 19 (το πλήθος των κόμβων είναι 20) κάνοντας τον παραπάνω έλεγχο παρατηρούμε ότι είναι αληθής, δηλαδή οι δείκτες i και j έχουν την ίδια τιμή (19). Ενημερώνουμε το μονοπάτι που σχηματίζεται από τους δείκτες i και j και στην αρχή του μονοπατιού (στην θέση 0), εισάγουμε τον κόμβο που βρίσκεται στην $19^{η}$ της συμβολοσειράς sigma. Αρχικοποιούμε το μήκος του παραπάνω μονοπατιού με την τιμή ένα, αφού το μονοπάτι περιέχει αυτή την στιγμή αυτό τον μοναδικό κόμβο (τον κόμβο 0). Στη συνέχεια συναντάμε μία δεύτερη περίπτωση για την οποία για τους δείκτες 18 και 19 κάνουμε τον έλεγχο και βλέπουμε ότι ο δείκτης i είναι μικρότερος από τον j. Ενημερώνουμε και πάλι τον

πίνακα που περιέχει το μονοπάτι που σχηματίζεται από τους δείκτες 18 και 19 και εισάγουμε στην αρχή του (θέση 0), τον κόμβο που βρίσκεται στην $18^{η}$ θέση της συμβολοσειράς sigma (δηλαδή τον κόμβο 2) και αρχικοποιούμε τον πίνακα με το μήκος του με την τιμή ένα. Με τον ίδιο τρόπο αρχικοποιούνται και τα υπόλοιπα μονοπάτια για τα οποία ισχύει ο παραπάνω έλεγχος.

Σε διαφορετική περίπτωση (δηλαδή όταν ο δείκτης i είναι μικρότερος του j), αυτό πρακτικά μεταφράζεται ότι ο δείκτης i+1 είναι μικρότερος του δείκτη j. Αυτό σημαίνει ότι το τρέχον μονοπάτι που συντελείται από του παραπάνω δείκτες δεν είναι άδειο. Δηλαδή στο εσωτερικό του φυλάσσει έναν ή περισσότερους κόμβους και σε αυτή την περίπτωση ενημερώνουμε τους πίνακες μονοπατιού και του μήκους του. Για κάθε κόμβο που περιέχεται στο υπο-γράφημα που σχηματίζεται από τους δείκτες i+1, j ελέγχουμε αν ο κόμβος u_y $\epsilon\, V(G(i+1,j))$ και ενημερώνουμε τον πίνακα για το μονοπάτι που έχει το u_y $\epsilon\, V(G(i+1,\ j)$ ως τον τελευταίο του κόμβο. Αυτό θα αποθηκευθεί στην εγγραφή $P(u_y;\ i+1,\ j)$. Ενημερώνουμε το μήκος του μονοπατιού.

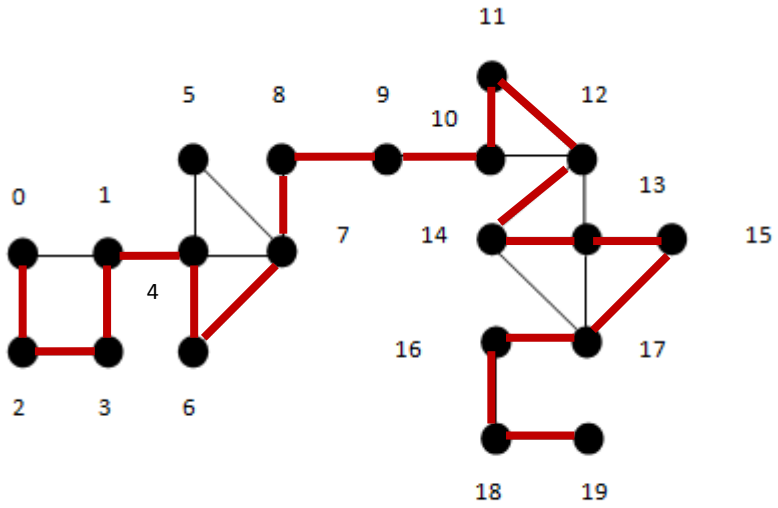
Για παράδειγμα στην εκτέλεση του προγράμματος η αρχικοποίηση γίνεται πρώτα για τον δείκτη i = 19. Ο κόμβος u_{19} που ανήκει στο υπο-γράφημα V(G(19,19)) είναι ο κόμβος 0 και τον εισάγουμε στην πρώτη θέση της εγγραφής του μονοπατιού. Η θέση όμως που εισάγεται έχει στο εσωτερικό της τους κόμβους 15 και 2. Διατηρούμε ένα μετρητή που μας ενημερώνει για το πλήθος των κόμβων που περιέχονται στην εγγραφή του πίνακα. Έτσι διατρέχουμε τον πίνακα για αυτές τις θέσεις και εισάγουμε στην θέση μηδέν τον κόμβο u_{19}, δηλαδή τον κόμβο που στην ακολουθία sigma έχει την θέση 19. Το μονοπάτι που σχηματίζεται από τους δείκτες i, j (18 και 19 αντίστοιχα), στην πρώτη του θέση ενημερώνεται με τον κόμβο 0. Με την ίδια λογική ενημερώνονται και τα υπόλοιπα μονοπάτια που δημιουργούνται από αντίστοιχους συνδυασμού δεικτών.

Έπειτα, χρησιμοποιώντας το επαγωγικό βήμα του αλγορίθμου (βλ. γραμμές 9-17) καθορίζουμε τον τρόπο με τον οποίο οι εγγραφές του πίνακα μπορούν να επεκταθούν με την προσθήκη του κόμβου u_i, στην περίπτωση που $u_i \in V(G(i, j))$. Πρώτα, παρατηρούμε ότι αν ένα μονοπάτι P των G(i, j) που εμπεριέχει τον κόμβο u_i έχει τουλάχιστον δύο κόμβους, τότε το P πρέπει να εμπεριέχει έναν κόμβο $u_x \in V(G(i+1, j))$ με $u_x u_i \in E$. Κάθε κόμβος u_x, μπορεί να εξυπηρετήσει με δύο διαφορετικές ιδιότητες, με σκοπό να βρεθεί το πιθανό μέγιστο μονοπάτι και στη συνέχεια να αποθηκευθεί στον πίνακα μου κρατά τα μονοπάτια. Στην πρώτη περίπτωση, προσθέτουμε μια ακμή $u_x u_i$ σε ένα μέγιστο μονοπάτι του G(i+1,j) με u_x τον τελευταίο κόμβο. Εκτελώντας το πρόγραμμα μας καθορίζουμε τον τρόπο με τον οποίο οι εγγραφές του πίνακα μπορούν να επεκταθούν με την προσθήκη του κόμβου u_i, στην περίπτωση που $u_i \in V(G(i, j))$, κάνοντας στον κώδικά μας τον έλεγχο του αν ο u_x βρίσκεται στην γειτονιά του u_i. Το πρώτο αποτέλεσμα που παίρνουμε εκτελώντας το πρόγραμμά μας, αφορά τον κόμβο u_{18} που ανήκει στο υπο-γράφημα που σχηματίζεται από τους δείκτες 18 και 19 αντίστοιχα. Ο κόμβος 18 συνορεύει με τον κόμβο 19 και αφού η ακμή $u_{18}u_{19}$ αν η προσθήκη του κόμβου αυξάνει το μήκος του τρέχοντος μονοπατιού τότε ενημερώνεται ο πίνακας που φυλλάσει τις εγγραφές των μονοπατιών και το μήκος για την εγγραφή που αφορά τον κόμβο u_i στο μονοπάτι που σχηματίζεται από τους δείκτες 18 και 19 αυξάνεται κατά ένα. Το μονοπάτι που σχηματίζεται από τους δείκτες 18 και 19 και έχει ως τελευταίο κόμβο τον κόμβο u_{18} ενημερώνεται με τον εξής τρόπο: στην θέση k = 0 του πίνακα εγγραφής path[i][i][j][k] συνενώνεται ο κόμβος u_{19} με τον κόμβο u_i, δηλαδή προστίθεται η ακμή $u_{18}u_{19}$ και με ότι είναι ενωμένος ο κόμβος u_{19}.

Η δεύτερη ιδιότητα που μπορεί να υιοθετήσει ο κόμβος u_x είναι να αποτελέσει τον σύνδεσμο ανάμεσα στις δύο ακμές u_xu_i και $u_iu_{y'}$. Αυτή η κατάσταση περιγράφεται στις γραμμές 14-17 του αλγόριθμου εύρεσης του μεγίστου μονοπατιού. Ενδεικτικά παραθέτουμε το ακόλουθο τμήμα εκτέλεσης του προγράμματος μας για αυτή την περίπτωση.

Για κάθε κόμβο u_y που ανήκει στο V(G(i+1, x-1)), αν το μήκος του μονοπατιού που περιέχει τον κόμβο u_i είναι μικρότερο από το μήκος που σχηματίζεται από την συνένωση του μονοπατιού που περιέχει τον κόμβο u_x και σχηματίζεται από τους δείκτες i+1 και j και του μονοπατιού που περιέχει τον κόμβο u_y και σχηματίζεται από τους δείκτες i+1 και x-1. Αν ισχύει ο παραπάνω έλεγχος τότε ενημερώνονται οι εγγραφές του πίνακα με τα μονοπάτια.

Τέλος, με τη βοήθεια της συνάρτησης compute_max_length, επιλέγουμε την εγγραφή του πίνακα με τα μονοπάτια η οποία έχει το μεγαλύτερο μήκος εκτελώντας έναν απλό έλεγχο για τα μήκη των μονοπατιών που εκχωρήθηκαν δυναμικά στην δομή που φυλάσσει τα μονοπάτια. Το μέγιστο μονοπάτι δοθέντος του παραπάνω γραφήματος είναι το 0 2 3 1 4 6 7 8 9 10 11 12 14 13 15 17 16 18 19 και έχει μήκος 18. Το αποτέλεσμα απεικονίζεται στο παρακάτω σχήμα.



Σχήμα 4.3.2. Γραφική απεικόνιση μέγιστου μονοπατιού εξόδου.

Κεφάλαιο 4

Συμπεράσματα - Επεκτάσεις

Παρουσιάσαμε έναν πολυωνυμικό αλγόριθμο βασισμένο στον Αλγόριθμο των G.B. Merzios, D.G. Corneil για την εύρεση του μεγίστου μονοπατιού σε cocomparability γραφήματα ο οποίος τρέχει σε χρόνο $O(n^4)$. Ο αλγόριθμος έκανε χρήση μιας LDFS διάταξης. Η υλοποίηση μας εφαρμόστηκε στην δομή των cocomparability γραφημάτων. Είναι ακόμα ανοιχτό το ερώτημα αν υπάρχει πολυωνυμικός αλγόριθμος για την επίλυση του προβλήματος σε μεγαλύτερες και πιο γενικευμένες κλάσεις γραφημάτων. Τα αποτελέσματα μας ελπίζουμε να χρησιμοποιηθούν και να γενικευθούν για την λύση του προβλήματος εύρεσης σε μεγαλύτερες τάξεις γραφημάτων. Κοιτάζοντας την επίδραση του αλγορίθμου $LDFS^+$ στα cocomparability γραφήματα, συμπεραίνουμε πόσο σημαντικό βήμα είναι στην απλούστερη δόμηση του αλγορίθμου επίλυσης του προβλήματος εύρεσης μεγίστων μονοπατιών. Για αυτό το λόγο ίσως είναι απαραίτητο να εξερευνηθούν οι πιθανές επεκτάσεις που ο $LDFS^+$ αλγόριθμος μπορεί να προσφέρει στην εξερεύνηση των γραφημάτων.

Βιβλιογραφία

[1] G.B. Merzios, D.G. Corneil, *A simple polynomial algorithm for the longest path problem on cocomparability graphs*, *SIAM J. Discrete Math*, 26(3), 940–963.
[2] K. Ioannidou, G.B. Mertzios, and S. D. Nikolopoulos, *The longest path problem has a polynomial solution on interval graphs*, Algorithmica, 61 (2011), 320–341.
[3] M. Habib, C. Paul, *A simple linear time algorithm for cograph recognition.* Discrete Applied Mathematics 145 (2005) 183 – 197.
[4] E. A. Hansen and I. Abdoulahi, *General Error Bounds in Heuristic Search Algorithms for Stochastic Shortest Path Problems,* Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16).
[5] D.G. Corneil, R.M. Krueger, The *longest path problem has a polynomial solution on interval graphs*, SIAM J. DISCRETE MATH Vol. 26, No. 3, 940–963.

[6] D.G. Corneil, B. Dalton, M. Habib, *An unified view of graph searching*, SIAM Journal on Computing 42:3, 792-807.