

**Πιθανοτικοί Αλγόριθμοι για Ανίχνευση Κύκλων και Διατήρηση Τοπολογικής Διάταξης σε Δυναμικά Γραφήματα**

**Μοναστηρλή Χρυσάνθη**

**Πτυχιακή Εργασία**

Επιβλέπων: Λ. Γεωργιάδης

Ιωάννινα, Οκτώβριος 2020

**Τμήμα Μηχ. Η/Υ & Πληροφορικής**

**Πανεπιστήμιο Ιωαννίνων**

**Department of Computer Science & Engineering**

**University of Ioannina**

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον Αναπληρωτή Καθηγητή κο Λουκά Γεωργιάδη που επέβλεψε την παρούσα εργασία για την καθοδήγηση, τη συνεργασία και την υποστήριξη καθ’ όλη τη διάρκεια της εκπόνησης καθώς και τον Καθηγητή κο Λεωνίδα Παληό και τον Αναπληρωτή Καθηγητή κο Χάρη Παπαδόπουλο για τις σημαντικές παρατηρήσεις τους.

Οκτώβριος 2020

Μοναστηρλή Χρυσάνθη

Περίληψη

Το αντικείμενο αυτής της εργασίας είναι η μελέτη και η υλοποίηση δύο αποδοτικών online αλγόριθμων για την ανίχνευση κύκλου (cycle detection) και την τήρηση τοπολογικής διάταξης (topological ordering) σε δυναμικά κατευθυνόμενα γραφήματα στα οποία προστίθενται ακμές. Οι δύο σχετιζόμενοι αλγόριθμοι στηρίζονται σε σύνολα που ενημερώνονται σε σταθερό αντισταθμιστικό χρόνο κάθε φορά που προστίθεται μία ακμή, όπως τα σύνολα προγόνων και απογόνων ενός κόμβου, καθώς και στη δειγματοληψία των κόμβων με μία πιθανότητα O( για να επιτύχουν απόδοση ανώτερη από προηγούμενες προσεγγίσεις. Η υλοποίηση έγινε με χρήση της γλώσσας προγραμματισμού Python.

**Λέξεις Κλειδιά:** δυναμικά γραφήματα, ανίχνευση κύκλου, τοπολογική διάταξη, online αλγόριθμοι

Abstract

The topic of this thesis is the study and implementation of two high performing online algorithms for cycle detection and topological ordering in dynamic graphs, i.e., graphs where edges are constantly added. The two algorithms are related and they both rely on node sets that are updated in constant amortized time each time an edge is added. These sets include node ancestors and descendants as well as a special set obtained by sampling all nodes with an O( probability. Compared to previous approaches these algorithms achieve superior performance for a wide range of graph sparsity. For this thesis they were both implemented in the Python programming language.

**Keywords:** dynamic graphs, cycle detection, topological ordering, online algorithms

Πίνακας περιεχομένων

Κεφάλαιο 1. Εισαγωγή 1

1.1 Αντικείμενο της πτυχιακής 1

1.2 Οργάνωση του τόμου 2

Κεφάλαιο 2. Περιγραφή Θέματος 3

Κεφάλαιο 3. Αλγόριθμοι επίλυσης 5

3.1 Ορολογία και μεταβλητές που χρησιμοποιούνται 5

3.2 Σύνολα προγόνων και απογόνων κόμβου 6

3.3 Επισκόπηση του αλγόριθμου ανίχνευσης κύκλων 7

Κεφάλαιο 4. Διατήρηση τοπολογικής διάταξης 11

4.1 Επισκόπηση του αλγόριθμου τοπολογικής διάταξης 12

Κεφάλαιο 5. Υλοποίηση των αλγόριθμων 17

5.1 Η γλώσσα προγραμματισμού Python 17

5.2 Η βιβλιοθήκη NetworkX 19

5.3 Τα κύρια τμήματα κώδικα/ Οι μεταβλητές 19

5.4 Παραδείγματα εκτέλεσης 20

5.5 Ανάλυση της υλοποίησης 24

5.5.1 Υλοποίηση του αλγόριθμου ανίχνευσης κύκλου 24

5.5.2 Υλοποίηση του αλγόριθμου τήρησης τοπολογικής διάταξης 28

Κεφάλαιο 6. Συμπεράσματα 35

# Εισαγωγή

## Αντικείμενο της πτυχιακής

Το αντικείμενο αυτής της πτυχιακής εργασίας είναι οι αλγόριθμοι που επιλύουν δύο βασικά προβλήματα σε δυναμικά κατευθυνόμενα γραφήματα. Τα δύο προβλήματα που παρουσιάζονται είναι η ανίχνευση κύκλου και η τήρηση έγκυρης τοπολογικής διάταξης τα οποία έχουν βέλτιστες λύσεις μόνο για μη κατευθυνόμενα γραφήματα.

Ο στόχος της εργασίας είναι η μελέτη των δύο αλγορίθμων και η υλοποίηση τους σε γλώσσα προγραμματισμούς Python. Οι αλγόριθμοι στηρίζονται σε δειγματοληψία των κόμβων με δεδομένη πιθανότητα και τήρηση συνόλων προγόνων/ απογόνων και άλλων ώστε να μειωθεί η συνολική πολυπλοκότητα όπως αυτή εκφράζεται από το συνολικό χρόνο των ενημερώσεων.

Τα γραφήματα που μελετώνται είναι κατευθυνόμενα (directed) και δυναμικά (dynamic), και κατασκευάζονται με τη σταδιακή προσθήκη ακμών. Τέτοιου τύπου γραφήματα έχουν αντιστοίχιση με πολλά πραγματικά προβλήματα, όπως π.χ. προβλήματα χρονοδρομολόγησης και η απόδοση των online αλγόριθμων που περιγράφονται είναι ιδιαίτερα ικανοποιητική σε σχέση με το μέγεθος του γράφου.

Όπως φαίνεται από τις δομές δεδομένων που τηρούνται στα πλαίσια των αλγόριθμων, το μέγεθος του προβλήματος μειώνεται με τη δειγματοληψία χωρίς να έχει αρνητικό αντίκτυπο στην αποτελεσματικότητά τους για την ανίχνευση κύκλου και την τήρηση τοπολογικής διάταξης. Αυτό το βλέπουμε από τα παραδείγματα εκτέλεσης, όπου από παρακολούθηση μέρους των κόμβων εξάγουμε συμπεράσματα για το σύνολο του γραφήματος.

## Οργάνωση του τόμου

Η δομή της εργασίας έχει ως εξής: στο κεφάλαιο 2 περιγράφεται πιο αναλυτικά το πρόβλημα που επιλύεται προγραμματιστικά.

Στο κεφάλαιο 3 ορίζονται τα σύνολα κόμβων στα οποία στηρίζονται οι δύο αλγόριθμοι που υλοποιούνται καθώς και ο αλγόριθμος ανίχνευσης κύκλου.

Στο κεφάλαιο 4 παρουσιάζεται ο αλγόριθμος διατήρησης τοπολογικής διάταξης.

Στο κεφάλαιο 5 αναλύεται η υλοποίηση των δύο αλγόριθμων και περιγράφονται τα σημαντικότερα βήματά τους.

Στο κεφάλαιο 6 παρατίθενται τα συμπεράσματα της εργασίας.

# Περιγραφή Θέματος

Στους αλγόριθμους που χειρίζονται δυναμικά γραφήματα ο στόχος είναι να διατηρείται μία σημαντική λειτουργικότητα ενόσω κάποιος «αντίπαλος» αλλάζει συνεχώς το γράφημα. Με άλλα λόγια, ο αλγόριθμος καλείται να μπορεί να χειριστεί απευθείας (online) μία σειρά από ενημερώσεις όπου ως ενημέρωση μπορεί να λογίζεται είτε η προσθήκη είτε η κατάργηση μιας ακμής. Εάν οι ενημερώσεις αφορούν μόνο προσθήκες, ο αλγόριθμος καλείται αυξητικός (incremental) ενώ αν γίνονται μόνο διαγραφές μειωτικός (decremental). Εάν ο αλγόριθμος μπορεί να χειριστεί και τις δύο περιπτώσεις καλείται πλήρως δυναμικός. Είναι επιθυμητό, σε έναν online αλγόριθμο, ο χρόνος που απαιτείται για να χειριστεί τις αλλαγές στο γράφημα (εισαγωγές ή διαγραφές) να είναι ο ελάχιστος δυνατός.

Οι αποδοτικοί αλγόριθμοί σε μη κατευθυνόμενα γραφήματα έχουν μελετηθεί σε βάθος και οι λύσεις πολύ βασικών προβλημάτων έχουν σχεδόν βελτιστοποιηθεί, αλλά αυτό δεν ισχύει στα κατευθυνόμενα γραφήματα όπου η πολυπλοκότητα είναι σαφώς μεγαλύτερη.

Τα δυναμικά, κατευθυνόμενα γραφήματα που αποτελούν το αντικείμενο αυτής της εργασίας είναι αρχικά κενά, και σε κάθε βήμα γίνεται προσθήκη μίας κατευθυνόμενης ακμής που ενώνει δύο κορυφές (incremental online). Οι αλγόριθμοι που υλοποιήθηκαν και παρουσιάζονται επιλύουν δύο θεμελιώδη προβλήματα, την online ανίχνευση κύκλου (cycle detection) και την online διατήρηση τοπολογικής διάταξης (topological sort). Στην ανίχνευση κύκλου που δημιουργείται από προσθήκη ακμής (incremental cycle detection) δίδεται ένα κατευθυνόμενο γράφημα που δεν περιέχει κύκλο και ο αλγόριθμος παρακολουθεί τις προσθήκες ακμών και αναφέρει την πρώτη προσθήκη που σχηματίζει κύκλο. Το πρόβλημα της διατήρησης τοπολογικής διάταξης συνδέεται στενά με την ανίχνευση κύκλων και αφορά στη διατήρηση μιας έγκυρης τοπολογικής διάταξης των κόμβων του άκυκλου γραφήματος την οποία ενημερώνει κάθε φορά που προστίθεται μία ακμή.

Τα δύο αυτά προβλήματα απαντώνται συχνά σε εφαρμογές χρονοδρομολόγησης (scheduling) όπου κάποιες εργασίες πρέπει να ολοκληρωθούν πριν από άλλες. Σε τέτοιες εφαρμογές, οι εργασίες και οι περιορισμοί που τις αφορούν αναπαριστώνται από κατευθυνόμενα γραφήματα με κάθε κόμβο να αντιστοιχεί σε μία εργασία και κάθε ακμή μεταξύ δύο εργασιών να αντιστοιχεί σε προαπαιτούμενο για την εκκίνησή της.

Ο στόχος και των δύο αλγορίθμων είναι η ελαχιστοποίηση της χρονικής πολυπλοκότητας (όπως αυτός εκφράζεται από το συνολικό χρόνο για όλες τις ενημερώσεις). Σε ένα γράφημα με n κόμβους και m ακμές, η τελική πολυπλοκότητα (το σύνολο του χρόνου για όλες τις ενημερώσεις) για τους δύο αλγόριθμους είναι για την ανίχνευση κύκλου και για την τήρηση τοπολογικής διάταξης, κάτι που αποτελεί βελτίωση σε σχέση με παλιότερες προσεγγίσεις.

# Αλγόριθμοι επίλυσης

Σε αυτό το κεφάλαιο περιγράφονται οι αλγόριθμοι που μελετήθηκαν και υλοποιήθηκαν για την επίλυση των δύο προβλημάτων που παρουσιάστηκαν παραπάνω σε κατευθυνόμενα άκυκλα γραφήματα. Και οι δύο αλγόριθμοι στηρίζονται σε κάποιες βασικές παρατηρήσεις/ θεωρήματα και στην τήρηση κάποιων συνόλων για όλους τους κόμβους ή μέρος αυτών τα οποία ενημερώνονται κάθε φορά που προστίθεται μία ακμή.

Ο συνολικός αλγόριθμος είναι ένας πιθανοτικός αλγόριθμος τύπου Las Vegas (Las Vegas randomized algorithm) ο οποίος παράγει πάντα σωστό αποτέλεσμα και έχει πολυπλοκότητα εκτέλεσης που εξαρτάται από τα χαρακτηριστικά της εισόδου (πλήθος κόμβων/ ακμών). Η απόδοση του αλγόριθμου είναι βελτιωμένη σε σχέση με άλλες προσεγγίσεις για διάφορες τιμές πυκνότητας των ακμών (graph density).

## Ορολογία και μεταβλητές που χρησιμοποιούνται

Η ορολογία και οι μεταβλητές κατάστασης είναι κοινές για τους δύο αλγόριθμους. Το κατευθυνόμενο γράφημα G είναι αρχικά κενό και σε κανένα σημείο της διαδικασίας δεν περιέχει κύκλο, δηλαδή οι προσθήκες ακμών σταματούν όταν προκύψει κύκλος.

Συμβολίζουμε την τελική μορφή του γραφήματος, πριν την εισαγωγή της ακμής που δημιουργεί κύκλο, ως όπου V το σύνολο κόμβων (το οποίο παραμένει σταθερό σε όλη τη διάρκεια εκτέλεσης), και το σύνολο ακμών πλην αυτής που δημιουργεί τον πρώτο κύκλο. Ως προς το πλήθος κόμβων και ακμών ισχύει:

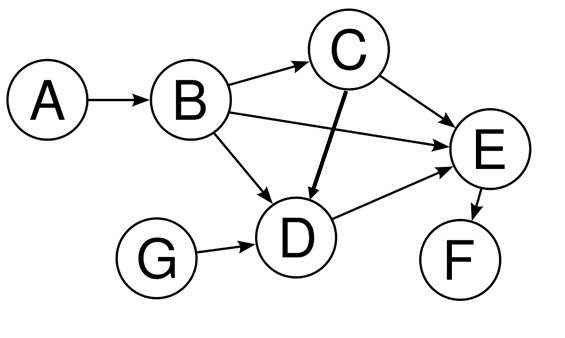
* και

## Σύνολα προγόνων και απογόνων κόμβου

Η βασική λογική του αλγόριθμου περιστρέφεται γύρω από την καταγραφή συνόλων προγόνων και απογόνων όλων ή μέρους των κόμβων του γραφήματος. **Πρόγονος** (ancestor) ενός κόμβου *v* ∈ *V* είναι ένας κόμβος *u* ∈ *V* εάν υπάρχει μονοπάτι από τον *u* στον *v* στο τρέχον γράφημα (*G*). Το σύνολο των προγόνων του κόμβου *v* αναφέρεται ως *A(v)*.

Αντίστοιχα, **απόγονος** (descendant) ενός κόμβου *v* ∈ *V* είναι ένας κόμβος *u* ∈ *V* εάν υπάρχει μονοπάτι από τον *v* στον *u* στο τρέχον γράφημα (*G*). Το σύνολο των απογόνων του κόμβου *v* αναφέρεται ως *D(v)*.

Κάθε κόμβος είναι πρόγονος και απόγονος του εαυτού του ενώ δύο κόμβοι έχουν σχέση συγγένειας (related) εάν ο ένας είναι πρόγονος του άλλου. Οι παραπάνω έννοιες γίνονται πιο σαφείς μέσω ενός παραδείγματος. Στην Εικόνα 1 φαίνεται ένα ενδεικτικό κατευθυνόμενο, άκυκλο γράφημα με 7 κόμβους και 9 ακμές.



Εικόνα 1: Κατευθυνόμενο γράφημα με 7 κόμβους και 9 ακμές

Σε αυτό το γράφημα, οι απόγονοι του κόμβου *G* είναι:

D(G) = {D, E, F, G}

ενώ οι πρόγονοι του κόμβου E είναι:

A(E) = {A,B,C,D,E,G}

Ο κόμβος G δεν έχει άλλους προγόνους εκτός από τον εαυτό του, άρα:

A(G)={G}

## Επισκόπηση του αλγόριθμου ανίχνευσης κύκλων

Το ζητούμενο σε αυτόν τον αλγόριθμο είναι να διαπιστωθεί εάν η εισαγωγή μιας ακμής (u, v) δημιουργεί κύκλο. Η πιο απλή λύση είναι να διατρέξουμε τις ακμές του γραφήματος από τον v και να δούμε αν φτάνουμε στον u αλλά αυτό απαιτεί χρόνο O(m), δηλαδή συνολικά για m εισαγωγές, το οποίο δεν είναι αποδοτικό. Για να επιτύχουμε καλύτερο χρόνο εκτέλεσης θα πρέπει να μικρύνουμε το σύνολο των κόμβων στους οποίους θα κάνουμε αναζήτηση όταν ξεκινάμε από τον v.

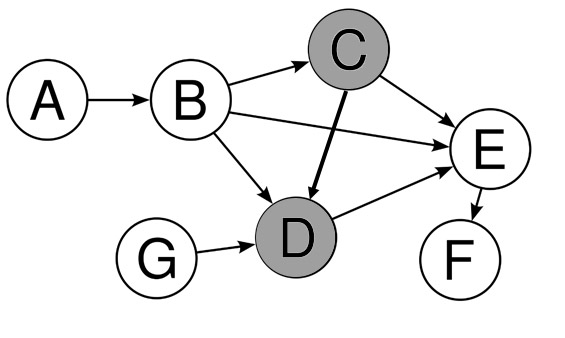
Στον αλγόριθμο που μελετάμε, αυτό γίνεται εισάγοντας την έννοια της ισοδυναμίας κόμβων. Δύο κόμβοι x και y είναι ισοδύναμοι (equivalent) εάν συνδέονται με σχέση συγγένειας και ισχύει ότι

A(x) = A(y) και D(x) = D(y)

Αποδεικνύεται ότι εάν δύο κόμβοι είναι μέρος ενός κύκλου θα είναι και ισοδύναμοι. Συνεπώς, κατά την εισαγωγή της ακμής (*u*, *v*) αρκεί στην αναζήτηση μας να ασχοληθούμε μόνο με κόμβους ισοδύναμους του *v*. Αυτό, ωστόσο, δε μειώνει τη συνολική πολυπλοκότητα άρα η έννοια της ισοδυναμίας πρέπει να χαλαρώσει περαιτέρω ώστε να απλοποιηθεί η εύρεση των συνόλων ισοδύναμων κόμβων. Σε αυτό συμβάλει η δειγματοληψία των κόμβων και η δημιουργία του υποσυνόλου S.

Με δεδομένο ένα υποσύνολο των κόμβων *S*, δύο κόμβοι είναι *S*-ισοδύναμοι εάν έχουν συγγένεια και ισχύει

και



Εικόνα 2: Παράδειγμα S-ισοδύναμων κόμβων

Για την αποσαφήνιση της έννοιας της S-ισοδυναμίας κόμβων, ας θεωρήσουμε το γράφημα στην Εικόνα 2, όπου οι κόμβοι C και D επισημαίνονται ως S-ισοδύναμοι. Για τον κόμβο C, τα σύνολα προγόνων και απογόνων είναι:

A(C) = {A, B, C} και D(C) = {C, D, E, F}

ενώ για τον κόμβο D ισχύει:

A(D) = {A, B, C, D, G} και D(G) = {D, E, F}

Εάν το σύνολο S που προέκυψε από τυχαία δειγματοληψία των κόμβων είναι:

S = {A, E, F}

τότε, οι τομές των συνόλων προγόνων και απογόνων με το S θα είναι:

κόμβος C: {A} και {E, F}

κόμβος D: {A} και {E, F}

Άρα οι δύο κόμβοι είναι S-ισοδύναμοι.

Οι κόμβοι του συνόλου *S* προκύπτουν με δειγματοληψία κάθε κόμβου του *V* με πιθανότητα . Αποδεικνύεται ότι με υψηλή πιθανότητα κάθε κόμβος *x* σε ένα κατευθυνόμενο άκυκλο γράφημα θα είναι *S*-ισοδύναμος με άλλους κόμβους. Αντίστροφα, εάν ο κόμβος *x* βρεθεί *S*-ισοδύναμος με περισσότερους από κόμβους τότε με υψηλή πιθανότητα το γράφημα περιέχει κύκλο ο οποίος διέρχεται από τον κόμβο *x*.

Επίσης, με συνολικό χρόνο ενημέρωσης είναι εφικτό σε χρόνο να ελέγξουμε αν 2 κόμβοι *x* και *y* είναι *S*-ισοδύναμοι. Όπως είναι εμφανές η απόδοση βελτιώνεται σημαντικά όταν η αναζήτηση περιορίζεται σε *S*-ισοδύναμους κόμβους του τελικού άκρου (end) της ακμής που εισάγεται. Παρόλο που υπάρχει δυνατότητα για περαιτέρω βελτίωση εισάγοντας την έννοια των ενίοτε *S*-ισοδύναμων κόμβων, δηλαδή των κόμβων που είναι *S*-ισοδύναμοι σε κάποιο σημείο της διαδικασίας εισαγωγής ακμών στο γράφημα, αυτό δεν είναι πρακτικά υλοποιήσιμο.

Το ερώτημα είναι ποια σύνολα μπορούν να τηρούνται για κάθε κόμβο σε πρακτικά αξιοποιήσιμο χρόνο. Με δεδομένο σύνολο κόμβων S ⊆ V, είναι εφικτή η ενημέρωση των παρακάτω συνόλων σε χρόνο καθ’όλη τη διαδικασία εισαγωγής ακμών:

1. Τα σύνολα *A(s)* και *D(s)* για κάθε *s* ∈ *S.*
2. Τα σύνολα και για κάθε *v* ∈ *V*, όπου και .
3. Τα σύνολα A(v) ⊆ A(v) για κάθε *v* ∈ *V*. Ένας κόμβος *u* που είναι πρόγονος του *v* θα προστεθεί στο σύνολο A(v) εάν οι *u* και *v* είναι *S*-ισοδύναμοι.
4. Μίας δομής δεδομένων η οποία απαντά σε χρόνο εάν δύο κόμβοι είναι *S*-ισοδύναμοι.

Με βάση τις παραπάνω μεταβλητές κατάστασης και τις ενημερώσεις, τα βήματα για την ανίχνευση κύκλου κατά την εισαγωγή της ακμής (u, v) στο γράφημα G έχουν ως εξής:

i) Αρχικοποίηση του συνόλου των κόμβων που θα πρέπει να διερευνηθούν

TO\_EXPLORE = {v}

ii) Ενόσω το σύνολο *TO\_EXPLORE* δεν είναι κενό αφαιρούμε από αυτό τυχαίο στοιχείο *w* (λειτουργία pop)

1. Εάν *w* = *u* υπάρχει κύκλος // έχουμε βρει ένα μονοπάτι από τον v στον u
2. Εάν *w* ∈ A(u) υπάρχει κύκλος /\* αφού ο w είναι πρόγονος του u, εντοπίσαμε τον κύκλο u…w…u \*/
3. Εάν *u* ∈ A(w) δεν απαιτείται κάποια ενέργεια /\* ένα μονοπάτι από τον u στον w υπήρχε την εισαγωγή \*/
4. Εάν *w* και *u* δεν είναι *S*-ισοδύναμοι δεν απαιτείται κάποια ενέργεια
5. Αλλιώς, το *u* προστίθεται στο σύνολο A(w) και

Για κάθε ακμή (*w*, *z*) ∈ *Ε*

Προσθήκη *z* στο σύνολο *TO\_EXPLORE*

Οι Bernstein και Chechik απέδειξαν ότι ο παραπάνω αλγόριθμος έχει χρόνο εκτέλεσης με μεγάλη πιθανότητα.

# Διατήρηση τοπολογικής διάταξης

Σε μία έγκυρη τοπολογική διάταξη των κόμβων κάθε κόμβος έπεται των προγόνων του και προηγείται των απογόνων του. Όπως αναφέρθηκε και παραπάνω, η τήρηση τοπολογικής διάταξης σε γράφημα έχει πολλές πρακτικές εφαρμογές όπως π.χ. ο χρονοπρογραμματισμός ενός έργου που χωρίζεται σε επιμέρους παραδοτέα που είναι προαπαιτούμενα για τα επόμενα.

Ο online αλγόριθμος για την τήρηση τοπολογικής διάταξης αξιοποιεί τις ίδιες δομές δεδομένων με τον αλγόριθμο ανίχνευσης κύκλου, δηλαδή τα σύνολα απογόνων, προγόνων κλπ αλλά είναι πιο τεχνικός και έχει μεγαλύτερη πολυπλοκότητα στη λειτουργία του.

Η γενική ιδέα του αλγόριθμου επικεντρώνεται στην προσεγγιστική εκτίμηση της θέσης ενός κόμβου v στη διάταξη με βάση τον αριθμό των προγόνων και των απογόνων του και βασίζεται στην παρατήρηση ότι οι κόμβοι με πολλούς προγόνους θα απαντώνται αργότερα στη διάταξη ενώ αυτοί με πολλούς απογόνους νωρίτερα. Ο αλγόριθμος ομαδοποιεί τους κόμβους που έχουν κοντινές θέσεις στη λίστα με χρήση κάδων (buckets). Οι κόμβοι μετακινούνται μεταξύ κάδων καθώς εισάγονται ακμές ενώ κόμβοι που έχουν «συγγενικές» σχέσεις (απογόνων ή προγόνων) βρίσκονται σε διαφορετικά buckets ή διατάσσονται κατάλληλα εντός ενός bucket. Η λογική της τοποθέτησης και οι σχέσεις που ισχύουν για τα buckets προγόνων/απογόνων εξηγούνται αναλυτικά παρακάτω. Η πολυπλοκότητα του αλγόριθμου έγκειται στην τήρηση σωστής διάταξης ΕΝΤΟΣ κάθε bucket.

Η αποθήκευση της τρέχουσας διάταξης (για τις ακμές που έχουν εισαχθεί) γίνεται σε μία δομή δεδομένων διατήρησης της διάταξης μιας λίστας (list order maintenance data structure) η οποία υποστηρίζει τις ακόλουθες λειτουργίες:

1. Insert(X, Y): δεδομένου ενός δείκτη στο στοιχείο X της λίστας, να εισαχθεί το στοιχείο Y αμέσως μετά το X στη διάταξη.
2. Insert-Before(X, Y): δεδομένου ενός δείκτη στο στοιχείο X, να εισαχθεί το στοιχείο Y αμέσως πριν το X στη διάταξη
3. Delete(X): δεδομένου ενός δείκτη στο στοιχείο X, να διαγραφεί το X από τη διάταξη.
4. Order(X, Y): επιστρέφει εάν το X ή το Y προηγείται στη διάταξη.

Στη βιβλιογραφία έχουν προταθεί δομές δεδομένων οι οποίες επιτυγχάνουν Ο(1) χρόνο ανά λειτουργία (χειρότερης περίπτωσης ή αντισταθμιστικό), αλλά η υλοποίηση τους είναι αρκετά περίπλοκη. Για τους σκοπούς της πτυχιακής, αρκεστήκαμε σε μια απλούστερη υλοποίηση με μια διατεταγμένη λίστα L. Ανά πάσα στιγμή, η λίστα L θα περιέχει τους κόμβους του γραφήματος καθώς και O(nlog2(n)) πλασματικά (dummy) στοιχεία, ο ρόλος των οποίων εξηγείται παρακάτω. Συμβολίζουμε με το ότι ο κόμβος *x* προηγείται του *y* στην τοπολογική διάταξη. Αυτό σημαίνει ότι αν , τότε .

## Επισκόπηση του αλγόριθμου τοπολογικής διάταξης

Σε σύγκριση με τον αλγόριθμο ανίχνευσης κύκλου, ο αλγόριθμος τήρησης τοπολογικής διάταξης είναι πολύ πιο τεχνικός. Η βασική ιδέα, ωστόσο, είναι ίδια. Αρχικά, γίνεται δειγματοληψία ενός συνόλου S κόμβων με πιθανότητα

Στη συνέχεια, ο αλγόριθμος διατηρεί μία προσεγγιστική τοπολογική διάταξη ομαδοποιώντας κάθε κόμβο v σε κάδους ανάλογα με τις τιμές

Το ζεύγος τιμών που λαμβάνεται υπόψη για την εύρεση της προσεγγιστικής θέσης ενός κόμβου στην τοπολογική διάταξη L είναι . Όσο μεγαλύτερη είναι η ποσότητα τόσο πιο πολλούς προγόνους έχει ο v, συνεπώς θα πρέπει να μη βρίσκεται στην αρχή της τοπολογικής διάταξης. Ως δευτερεύον κριτήριο κατάταξης, όσο πιο μεγάλη είναι η ποσότητα τόσο πιο πολλούς απογόνους έχει ο κόμβος v, γεγονός που υποδεικνύει ότι θα απαντάται νωρίτερα στην τοπολογική διάταξη.

Ο αλγόριθμος δημιουργεί ένα κάδο (bucket) για κάθε πιθανό συνδυασμό |A(v)∩S|,-|D(v)∩S| και διατηρεί την αναλλοίωτη συνθήκη(invariant) ότι κάθε κόμβος που βρίσκεται σε μικρότερο bucket (με λεξικογραφική σειρά) θα προηγείται στην τοπολογική διάταξη από έναν που βρίσκεται σε μεγαλύτερο. Αποδεικνύεται ότι κάθε κόμβος μπορεί να αλλάξει bucket το πολύ φορές ενώ οι μετακινήσεις εντός buckets είναι συνολικά τάξης .

Από τη λειτουργία του αλγόριθμου σε σχέση με την τοποθέτηση των κόμβων σε κάδους, προκύπτει ότι αν για δύο κόμβους ισχύει τότε αφού κάθε κόμβος προηγείται εξ ορισμού των απογόνων του στην τοπολογική διάταξη ή αλλιώς και . Επίσης, εάν u και v έχουν σχέση συγγένειας και τότε οι δύο κόμβοι είναι S-ισοδύναμοι.

Η δεικτοδότηση (indexing) των κάδων και των λοιπών συνόλων/ τιμών γίνεται με χρήση δύο ακέραιων τιμών i και j, με και . Το bucket Bi,j ορίζεται ως εξής:

Ισχύει ότι αν i>i' ή i=i' και j>j’. Στο γράφημα υπάρχουν συνολικά κάδοι.

Όταν ξεκινά, ο αλγόριθμος τοποθετεί στην τοπολογική διάταξη ένα στοιχείο κράτησης θέσης (placeholder) Pi,j για κάθε κάδο Bi,j με αύξουσα σειρά (των κάδων/buckets). Στη συνέχεια, τοποθετεί όλους τους κόμβους του γραφήματος αμέσως μετά το bucket με τυχαία σειρά. Η τοπολογική διάταξη θα διατηρεί την ακόλουθη αναλλοίωτη συνθήκη (bucket invariant): εάν τότε το στοιχείο θα προηγείται πάντα του στην τοπολογική διάταξη. Επίσης, εάν τότε το v έπεται πάντα του στοιχείου κράτησης θέσης στη λίστα L της τοπολογικής διάταξης αλλά προηγείται πάντα του , όπου είναι το επόμενο στοιχείο κράτησης θέσης μετά το .

Κατά την εισαγωγή μίας ακμής ένας κόμβος ενδέχεται να μετακινηθεί σε μεγαλύτερο ή μικρότερο bucket. Τα σύνολα UPi,j και DOWNi,j ορίζονται ως εξής:

Όπως είναι λογικό, οι κόμβοι που ανήκουν σε αυτά τα δύο σύνολα θα χρειαστεί να τοποθετηθούν σε νέες θέσεις στην τοπολογική διάταξη.

Τα αναλυτικά βήματα του αλγόριθμου κατά την προσθήκη της ακμής (u, v) στο γράφημα G έχουν ως εξής:

Στο πρώτο μέρος του αλγόριθμου μετακινούνται τα στοιχεία που άλλαξαν κάδους στις νέες τους θέσεις στην τοπολογική διάταξη.

1. Εύρεση συνόλων UP, DOWN, και όλων των μη κενών συνόλων UPi,j , DOWNi,j
2. Για κάθε μη κενό σύνολο UPi,j
3. Ταξινόμηση των στοιχείων του UPi,j με αύξουσα σειρά με βάση τη σειρά των κόμβων στην τοπολογική διάταξη L.
4. Για κάθε w ∈ UPi,j με φθίνουσα σειρά:
   1. Διαγραφή του w από την τρέχουσα θέση του στην τοπολογική διάταξη.
   2. Εισαγωγή του w αμέσως μετά το στοιχείο κράτησης θέσης Pi,j στην L
5. Για κάθε μη κενό σύνολο DOWNi,j
6. Ταξινόμηση των στοιχείων του DOWNi,j με αύξουσα σειρά με βάση τη σειρά των κόμβων στην τοπολογική διάταξη L.
7. Για κάθε w ∈ DOWNi,j με αύξουσα σειρά:
   1. Διαγραφή του w από την τρέχουσα θέση του στην τοπολογική διάταξη.
   2. Έστω ότι Bi’,j’ είναι το επόμενο bucket μετά το Bi,j
   3. Εισαγωγή του w αμέσως πριν το στοιχείο κράτησης θέσης Pi’,j’ στην L

Στο δεύτερο μέρος διορθώνεται η σειρά των στοιχείων εντός του bucket που περιέχει τα u και v. Εφόσον δε βρίσκονται στο ίδιο bucket δεν απαιτείται κάποια ενέργεια γιατί έχουν ήδη διαταχθεί σωστά με βάση τα buckets.

1. Εάν Bnew(u) = Bnew(v)

Αρχικοποίηση συνόλου to-explore = {v}

Αρχικοποίηση συνόλου to-change= ∅ . Αυτό το σύνολο θα περιέχει όλους τους κόμβους που πρόκειται να μετακινηθούν.

Ενόσω to-explore ≠ ∅

w = to-explore.pop-arbitrary-element /\*εξαγωγή αυθαίρετου στοιχείου από το σύνολο to-explore \*/

**4a**. Εάν δεν απαιτείται κάποια ενέργεια /\*οι κόμβοι u και w είναι ήδη σωστά διατεταγμένοι \*/

**4b**. Αλλιώς εάν Bnew(w) ≠ Bnew(u) δεν απαιτείται κάποια ενέργεια /\* οι κόμβοι u και w είναι ήδη σωστά διατεταγμένοι μέσω των κάδων \*/

**4c**. Αλλιώς

Προσθήκη w στο σύνολο to-change

Για κάθε ακμή (w, z)

Προσθήκη του z στο σύνολο to-explore

**4d**. Ταξινόμηση των στοιχείων του συνόλου to-change με αύξουσα σειρά με βάση τη θέση στην τοπολογική διάταξη L.

**4e**. Για κάθε w ∈ to-change με φθίνουσα σειρά, κάνε εισαγωγή αμέσως μετά το u στην L. Η επεξεργασία του συνόλου to-change γίνεται από το τέλος προς την αρχή, εξ ου και η φθίνουσα σειρά.

1. Αλλιώς εάν Bnew(u) ≠ Bnew(v) δεν απαιτείται κάποια ενέργεια /\* οι κόμβοι u και v είναι ήδη σωστά διατεταγμένοι μέσω των κάδων \*/

Οι Bernstein και Chechik απέδειξαν ότι ο παραπάνω αλγόριθμος έχει χρόνο εκτέλεσης με μεγάλη πιθανότητα.

# Υλοποίηση των αλγόριθμων

Η υλοποίηση των δύο αλγόριθμων έγινε στη γλώσσα προγραμματισμού Python. Η συγκεκριμένη γλώσσα επελέγη γιατί είναι σύγχρονη, ευέλικτη και διαθέτει μεγάλο πλήθος χρήσιμων βιβλιοθηκών και μεγάλη κοινότητα υποστήριξης. Η υλοποίηση έγινε στην 64bit έκδοση της Python 3.7 σε λειτουργικό σύστημα Microsoft Windows με χρήση του ενσωματωμένου περιβάλλοντος ανάπτυξης IDLE (Python’s Integrated Development and Learning Environment).

Σε αυτό το κεφάλαιο παρουσιάζονται αρχικά κάποια στοιχεία σχετικά με τη γλώσσα Python και το βασικό πακέτο για γραφήματα που χρησιμοποιήθηκε και στη συνέχεια περιγράφονται τα κύρια τμήματα κώδικα, δίνονται παραδείγματα από οθόνες εκτέλεσης και γίνεται ανάλυση της υλοποίησης.

## Η γλώσσα προγραμματισμού Python

Η Python είναι μία δημοφιλής και ευέλικτη γλώσσα προγραμματισμού που βασίζεται σε διερμηνέα (interpreter) αντί για μεταγλωττιστή (compiler). Υποστηρίζει διάφορα είδη προγραμματισμού όπως διαδικαστικό και αντικειμενοστραφή. Δημιουργήθηκε στα τέλη της δεκαετίας του 80 στο Εθνικό Ερευνητικό Κέντρο Μαθηματικών και Πληροφορικής της Ολλανδίας (National Research Institute for Mathematics and Computer Science (Centrum Wiskunde & Informatica - CWI) από τον Guido van Rossum.

Η υλοποίησή της ξεκίνησε τον Δεκέμβριο του 1989 και η πρώτη της έκδοση (0.9.0) ξεκίνησε να διατίθεται τον Φεβρουάριο του 1991. Το όνομα της γλώσσας δεν προέρχεται από το φίδι αλλά από την τηλεοπτική σειρά του BBC “Monty Python's Flying Circus”. Μέχρι το 2020, έχουν εκδοθεί 4 σειρές εκδόσεων της Python: 0.9.x (1991 – 1993), 1.x (1994 – 2000), 2.x (2000 - 2010) και η σειρά εκδόσεων 3.x (2008 - ) η οποία είναι η τρέχουσα, με τελευταία έκδοση την Python 3.8. Τα κύρια χαρακτηριστικά της γλώσσας είναι:

* Χρήση εσοχών για την οριοθέτηση μπλοκ κώδικα (π.χ. βρόχων επανάληψης ή δομών ελέγχου). Αναφέρεται και ως meaningful white space. Στην Python δεν υπάρχουν άγκιστρα {} ή άλλες λέξεις κλειδιά για να δηλώσουν την αρχή και το τέλος των μπλοκς ομαδοποιημένων εντολών. Επίσης, το τέλος μιας εντολής δηλώνεται από την αλλαγή γραμμής και όχι από κάποιον ειδικό χαρακτήρα.
* Dynamic typing. Οι μεταβλητές στην Python δεν έχουν τύπο και δε δηλώνονται. Επίσης, οι μεταβλητές επιτρέπεται να αλλάζουν τύπο κατά την εκτέλεση, πχ σε έναν ακέραιο μπορεί να αποδοθεί μία συμβολοσειρά. Οι μεταβλητές στην Python είναι στην ουσία αναφορές (references) σε αντικείμενα τα οποία περιέχουν τα δεδομένα και μεταβλητές με την ίδια τιμή δείχνουν στο ίδιο αντικείμενο. Όταν όλες οι μεταβλητές που δείχνουν σε ένα αντικείμενο αλλάξουν τιμή, αυτό καταργείται και γίνεται περισυλλογή απορριμμάτων.
* Ενσωματωμένοι τύποι δεδομένων: στην Python όλα είναι αντικείμενα (Objects), ακόμα και οι ίδιες οι κλάσεις. Εκτός από ακέραιους και αριθμούς κινητής υποδιαστολής, η Python υποστηρίζει μιγαδικούς και strings τα οποία είναι αμετάβλητα (immutable). Επίσης υποστηρίζονται διάφοροι τύποι συλλογών (Collections) όπως λίστες, ζεύγη, σύνολα και χάρτες/ λεξικά.
* Υποστήριξη πολλών ειδών προγραμματισμού (programming paradigms): όπως αναφέρθηκε και παραπάνω, η Python υποστηρίζει πλήρως τόσο τον διαδικαστικό (imperative) και τον αντικειμενοστραφή (object oriented) προγραμματισμό.

Η Python διαθέτει επίσης πολλές βιβλιοθήκες (πακέτα) γενικής και ειδικής χρήσης όπως το NumPy για την υποστήριξη μεγάλων πινάκων και το NetworkX για γραφήματα το οποίο χρησιμοποιείται εκτεταμένα στην υλοποίηση των ζητούμενων αλγόριθμων. Στην παράγραφο που ακολουθεί επισημαίνονται κάποια βασικά στοιχεία σχετικά με το NetworkX.

## Η βιβλιοθήκη NetworkX

Η βιβλιοθήκη Network είναι δωρεάν λογισμικό ανοικτού κώδικα (με άδεια χρήσης BSD License) που εγκαθίσταται μέσω του διαχειριστή πακέτων (pip) της Python, με εκτέλεση της εντολής:

pip install networkx

Το αντικείμενο της βιβλιοθήκης είναι η ανάλυση πολύπλοκων δικτύων. Σε αυτή την κατεύθυνση, το πακέτο επιτρέπει τη δημιουργία απλών γραφημάτων, κατευθυνόμενων γραφημάτων και γραφημάτων με πολλαπλές παράλληλες ακμές μεταξύ δύο κόμβων (multigraph) μέσω κλάσεων που διαθέτει. Επίσης, έχει υλοποιημένους πολλούς δημοφιλείς αλγόριθμους γραφημάτων και εργαλεία ανάλυσης δικτύων. Οι κόμβοι σε ένα δίκτυο μπορούν να είναι οτιδήποτε, π.χ. κείμενο, εικόνες, εγγραφές XML) ενώ οι ακμές μπορούν επίσης να φέρουν οτιδήποτε ως δεδομένο, π.χ. να έχουν βάρη ή να αντιστοιχούν σε χρονοσειρές.

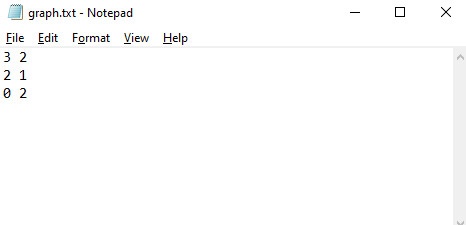
## Τα κύρια τμήματα κώδικα/ Οι μεταβλητές

Οι δύο αλγόριθμοι υλοποιήθηκαν σε κοινό αρχείο καθώς βασίζονται στις ίδιες δομές δεδομένων και επιπλέον, ο αλγόριθμος για την τήρηση τοπολογικής διάταξης προϋποθέτει άκυκλο γράφημα, συνεπώς οι εισαγωγές ακμών πρέπει να ελέγχονται για τη δημιουργία κύκλων. Οι βασικές ενότητας του κώδικα είναι

1. Συμπερίληψη απαιτούμενων πακέτων και Αρχικοποίηση μεταβλητών/ συνόλων/ λιστών
   1. Αρχικοποίηση συνόλων και κλειδιών
2. Δημιουργία του γραφήματος (από αρχείο ή με τυχαία επιλογή)
   1. Προσθήκη κόμβων και δημιουργία πίνακα ακμών (χωρίς αυτές να περιλαμβάνονται στο γράφημα)
3. Προσθήκη ακμών (μία τη φορά)
   1. Ενημέρωση μεταβλητών (π.χ. σύνολα προγόνων/ απογόνων κλπ)
   2. Έλεγχος για τη δημιουργία κύκλου
   3. Ενημέρωση τοπολογικής διάταξης που τηρείται online ώστε να αποτελεί έγκυρη τοπολογική διάταξη για το τρέχον γράφημα.
4. Εμφάνιση εκτεταμένων μηνυμάτων εάν έχει επιλεγεί να γίνεται verbose έκδοση.

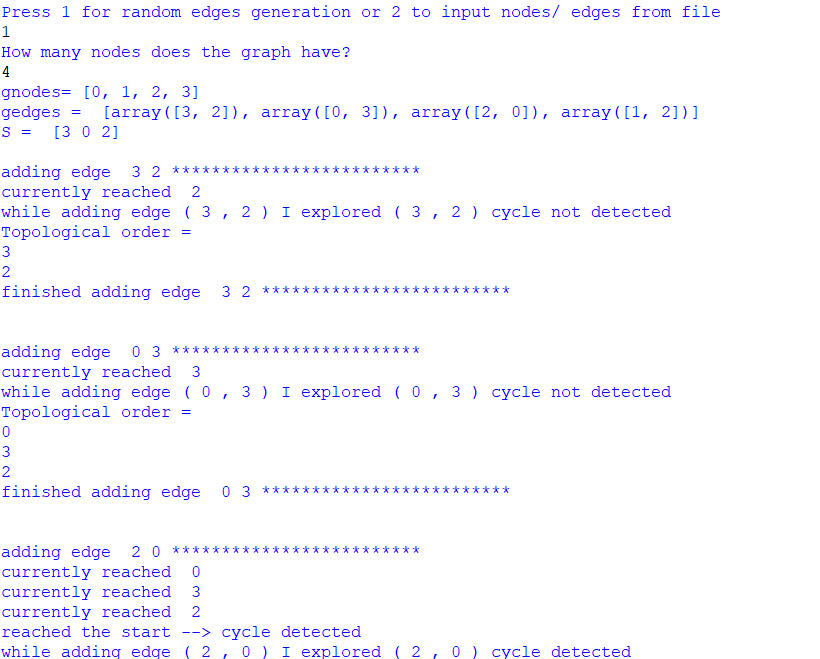
## Παραδείγματα εκτέλεσης

Η εκτέλεση γίνεται μέσω του περιβάλλοντος IDLE. Το πρόγραμμα μπορεί να παράγει με τυχαία επιλογή τις ακμές του γραφήματος ή να διαβάσει το γράφημα από ένα αρχείο κειμένου που έχει δημιουργήσει ο χρήστης στον ίδιο κατάλογο με όνομα graph.txt. Η δομή του αρχείου κειμένου φαίνεται στην Εικόνα 1. Οι δύο ακέραιοι στην πρώτη γραμμή του αρχείου αντιστοιχούν στα πλήθη κόμβων και ακμών ενώ οι γραμμές που ακολουθούν προσδιορίζουν τις ακμές ως ζεύγη από ids κορυφών.

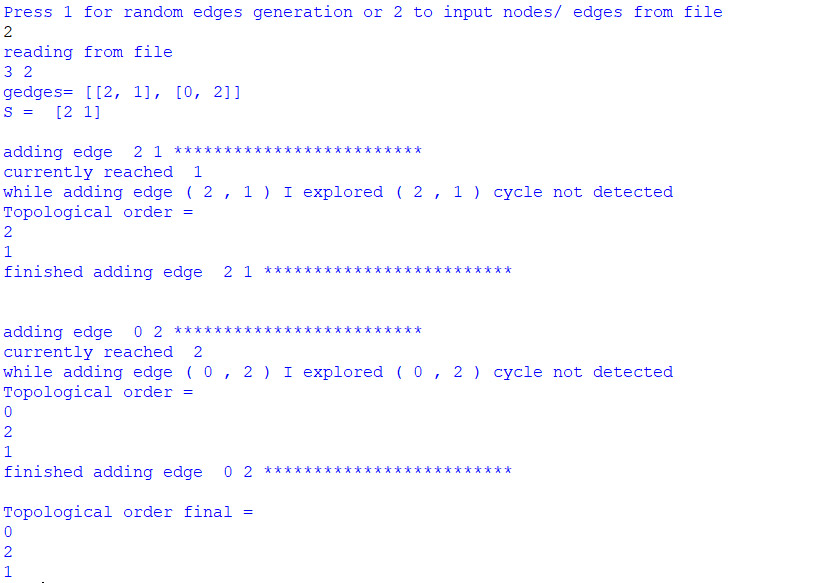


Εικόνα 3: Περιεχόμενα αρχείου κειμένου που αντιστοιχεί σε γράφημα με 3 κόμβους και 2 ακμές

Κατά την έναρξη της εκτέλεσης, ο χρήστης ερωτάται εάν επιθυμεί την παραγωγή του γραφήματος με τυχαίο τρόπο (επιλογή 1) ή την ανάγνωσή του από αρχείο (επιλογή 2). Σε περίπτωση που αυτός διαλέξει το πρώτο, θα πρέπει να εισάγει στη συνέχεια το πλήθος των κόμβων n του τελικού γραφήματος (Εικόνα 2).

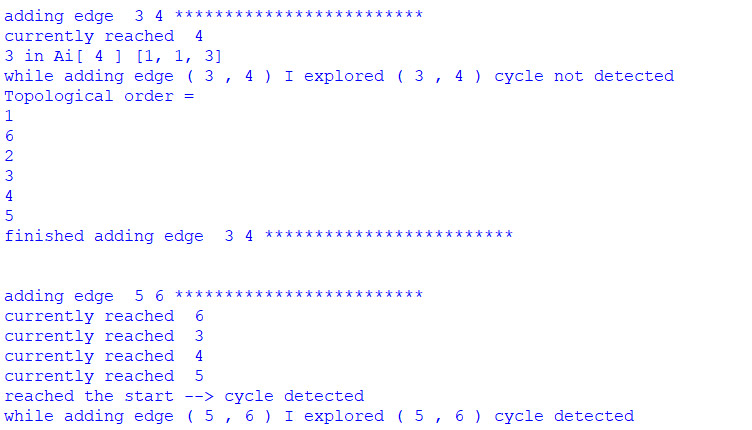


Εικόνα 4: Παράδειγμα εκτέλεσης με τυχαία παραγωγή γραφήματος

………………….

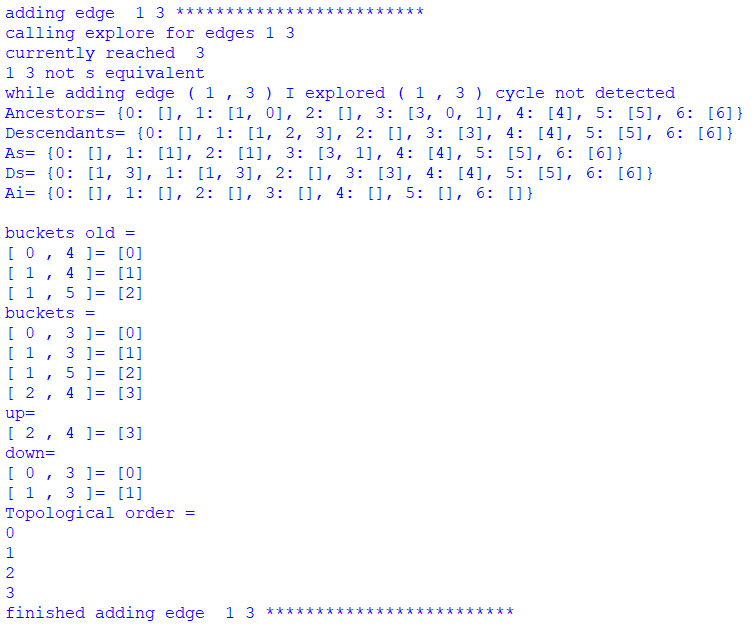
Εικόνα 5: Παράδειγμα εκτέλεσης με ανάγνωση γραφήματος από αρχείο

Από τη στιγμή που δημιουργείται το γράφημα ο αλγόριθμος αρχικοποιεί τις δομές δεδομένων που απαιτούνται και ξεκινά τον κύριο βρόχο προσθήκης των ακμών ελέγχοντας εάν δημιουργείται κύκλος. Στην περίπτωση που το γράφημα παραμένει άκυκλο υπολογίζει και την τρέχουσα τοπολογική διάταξη. Εάν η τρέχουσα ακμή δημιουργεί κύκλο, η εκτέλεση διακόπτεται και ο χρήστης ενημερώνεται με κατάλληλο μήνυμα (Εικόνα 4).



Εικόνα 6: Μήνυμα ανίχνευσης κύκλου και διακοπή εκτέλεσης

Για την περίπτωση που ο χρήστης επιθυμεί να παρακολουθήσει πιο αναλυτικά τη λειτουργία των αλγόριθμων, παρέχονται δύο λογικές μεταβλητές από τις οποίες η πρώτη εμφανίζει πολύ πιο αναλυτικές εκτυπώσεις μεταβλητών σε κάθε προσθήκη ακμής και η δεύτερη διακόπτει την εκτέλεση μετά από κάθε προσθήκη και περιμένει από το χρήστη είσοδο (να πατήσει το πλήκτρο Enter) για να συνεχίσει. Στην Εικόνα 5 φαίνεται η αναλυτική έξοδος κατά την προσθήκη μιας ακμής. Μεταξύ των μεταβλητών που εκτυπώνονται είναι οι κάδοι πριν την προσθήκη της ακμής, οι τρέχοντες κάδοι, τα σύνολα up και down (όσα δεν είναι κενά) και η τοπολογική διάταξη που έχει υπολογιστεί με βάση τις ακμές που έχουν προστεθεί.



Εικόνα 7: Εκτέλεση με αναλυτικές εκτυπώσεις (verbose output)

## Ανάλυση της υλοποίησης

Στην ενότητα αυτή παρουσιάζονται τα σημαντικά τμήματα του κώδικα που σχετίζονται με τις βασικές δομές δεδομένων και τις κύριες λειτουργίες, πρώτα για την ανίχνευση κύκλου και μετά για την τοπολογική διάταξη.

### Υλοποίηση του αλγόριθμου ανίχνευσης κύκλου

Το γράφημα δημιουργείται με χρήση κατασκευαστή του πακέτου NetworkX για αντικείμενα που αντιστοιχούν σε κατευθυνόμενους γράφους.

#create a directed graph object

*G = nx.DiGraph()*

Οι κόμβοι και οι ακμές στο γράφημα επίσης προστίθενται με μεθόδους του ίδιου πακέτου.

gnodes = []

for node in range(num\_nodes):

gnodes.append(node)

G.add\_nodes\_from(gnodes)

G.add\_edge(start, end)

Κατά την τυχαία παραγωγή γραφήματος, οι ακμές παράγονται με τη μέθοδο random του πακέτου NumPy η οποία κάνει δειγματοληψία 2 διαφορετικών κόμβων που θα αποτελέσουν τα άκρα της ακμής από το σύνολο των κόμβων του γραφήματος.

edge\_nodes = np.random.choice(list(gnodes), 2, replace=False)

Η νέα ακμή ενδεχομένως να είναι ήδη στη λίστα ακμών συνεπώς πρέπει να γίνει έλεγχος εάν υπάρχει ήδη και αν ναι, να παραχθούν τα άκρα της ξανά.

while found == True:

edge\_nodes = []

edge\_nodes = np.random.choice(list(gnodes), 2, replace=False)

found = False

for i in range(len(gedges)):

if gedges[i][0] == edge\_nodes[0] and gedges[i][1] == edge\_nodes[1]:

found = True

Για τα σύνολα προγόνων, απογόνων κλπ χρησιμοποιούνται αντικείμενα dict της Python, δηλαδή λεξικά. Δε χρησιμοποιούνται σύνολα γιατί αυτά δεν μπορούν να δεικτοδοτηθούν με δείκτες όπως τα λεξικά.

ancestors = dict()

#As = ancestors intersect S for all nodes in S

As = dict()

#set Ai is a subset of ancestors based on S equivalency

Ai = dict()

descendants = dict()

#Ds = descendants intersect S for all nodes in S

Ds = dict()

Τα κλειδιά για αυτά τα λεξικά είναι τα ids των κόμβων και αρχικοποιούνται ως εξής (ο πίνακας current\_nodes περιέχει όλους τους κόμβους του γραφήματος):

for node in current\_nodes:

ancestors.setdefault(node, [])

As.setdefault(node, [])

Ai.setdefault(node, [])

descendants.setdefault(node, [])

Ds.setdefault(node, [])

Όπως φαίνεται παραπάνω, αρχικά όλα τα λεξικά είναι κενά. Για να αναφερθώ, για παράδειγμα, στους απογόνους ενός κόμβου node1, μπορώ να χρησιμοποιήσω τη μεταβλητή descendants[node1] που θα μου δώσει την αντίστοιχη λίστα.

Η έννοια της S-ισοδυναμίας είναι πολύ σημαντική στην ανίχνευση κύκλων και για το λόγο αυτό υπάρχει συνάρτηση που παίρνει ως ορίσματα 2 κόμβους και επιστρέφει αν είναι S-ισοδύναμοι.

#returns 1 if nodes are S equivalent

def nodes\_S\_equivalent(node1, node2):

if len(As[node1]) == len(As[node2]) and len(Ds[node1])==len(Ds[node2]):

return 1

else:

return 0

Η κύρια διαδικασία κατά την ανίχνευση κύκλου είναι η εξερεύνηση πιθανών διαδρομών που ξεκινούν από το τελικό άκρο end της ακμής που προστέθηκε και καταλήγουν στο αρχικό άκρο start (εάν υπάρχει κύκλος). Αυτή η διαδικασία καλείται για όλους τους κόμβους που είναι τελικά άκρα ακμών που ξεκινούν από το end και βασίζεται στην S-ισοδυναμία και στην παρουσία στο σύνολο Ai (το σύνολο των προγόνων που είναι και S-ισοδύναμοι).

#returns 1 if cycle detected when inserting edge u, v

def explore(u, v):

to\_explore = set()

to\_explore.add(v)

while len(to\_explore)>0:

w = to\_explore.pop()

if w==u:

return 1

if w in Ai[u]:

return 1

else:

if u in Ai[w]:

print(u, "in Ai[",w,"]")

else:

if nodes\_S\_equivalent(u, w)==0:

Ai[w].append(u)

for e in gedges:

if e[0]==w:

to\_explore.add(e[1])

return 0

Ο κύριος βρόχος της προσθήκης ακμών διατρέχει τον πίνακά τους και επεξεργάζεται σύνολα προτού καλέσει την παραπάνω διαδικασία. Παρατίθενται ενδεικτικές ενημερώσεις συνόλων.

for e in gedges:

start = e[0]

end = e[1]

if current\_node in S:

descendants[current\_node].append(end)

if end in S:

Ds[current\_node].append(end)

ancestors[end].append(current\_node)

if current\_node in S:

As[end].append(current\_node)

if nodes\_S\_equivalent(current\_node, end):

Ai[end].append(current\_node)

Όπως φαίνεται από τις παραπάνω εντολές τα σύνολα ancestors και descendants τηρούνται για τους κόμβους που ανήκουν στο σύνολο S ενώ τα σύνολα As και Ds για όλους τους κόμβους. Κατά την προσθήκη μια ακμής (start, end) επηρεάζονται τα σύνολα των δύο άκρων της αλλά και τα σύνολα για όλους τους προγόνους και απογόνους αυτών. Συνεπώς, πρέπει να γίνει διερεύνηση των κόμβων που είναι προσβάσιμοι από το end με διάσχιση του γραφήματος (μέθοδος dfs\_preorder\_nodes του NetworkX) και να γίνουν αντίστοιχες ενημερώσεις προτού κληθεί η μέθοδος explore που παρουσιάστηκε παραπάνω.

dfs\_reachable = list(nx.dfs\_preorder\_nodes(G,end))

for node in dfs\_reachable:

……….

### Υλοποίηση του αλγόριθμου τήρησης τοπολογικής διάταξης

Ο αλγόριθμος τήρησης τοπολογικής διάταξης χρησιμοποιεί επιπλέον λεξικά τα οποία δεικτοδοτούνται με δύο δείκτες i και j, συνεπώς απαιτούν τα κλειδιά τους να είναι ζεύγη. Από τους δύο δείκτες ο πρώτος έχει αύξουσα σειρά (ξεκινά από το 0 και καταλήγει σε κάποιο άνω όριο) ενώ ο δεύτερος έχει φθίνουσα σειρά και ξεκινάει από το δικό του άνω όριο. Τα ζεύγη των κλειδιών δημιουργούνται με βάση τα όρια τους και ανατίθενται στα λεξικά.

#keys for dictionaries (tuples)

dicts\_keys = []

for i in range(i\_limit):

for j in range(j\_limit):

dicts\_keys.append((i,j))

#the tuple is the key

for tupleT in dicts\_keys:

buckets.setdefault(tupleT, [])

placeholders.setdefault(tupleT, -1)

up.setdefault(tupleT, [])

down.setdefault(tupleT, [])

Τα στοιχεία κράτησης θέσης για κάθε κάδο παίρνουν αρνητικές τιμές για να μπορούν να διαχωριστούν από τους κόμβους και τοποθετούνται στη λίστα L που αντιστοιχεί στην τοπολογική διάταξη με τη σειρά των κλειδιών.

#placeholder elements in list L

placeInList = 0

#for every bucket

for i in range(i\_limit):

for j in range(j\_limit):

placeholders[(i,j)] = [-1 - placeInList]

L.append(placeholders[(i,j)][0])

placeInList = placeInList + 1

Οι 4 βασικές λειτουργίες της λίστας της τοπολογικής διάταξης (εισαγωγή, εισαγωγή-πριν, διαγραφή, έλεγχος σειράς) υλοποιούνται ως συναρτήσεις που παίρνουν και την ίδια τη λίστα ως όρισμα.

def listInsertBefore(X, Y, L):

i = L.index(X)

L.insert(i, Y)

#insert method inserts BEFORE given position

return 0

def listInsert(X, Y, L):

i = L.index(X)

L.insert(i+1, Y)

return 0

def delete(X, L):

L.remove(X);

return 0

def order(X, Y, L):

#returns true if X before Y, false otherwise

return (L.index(X)<L.index(Y))

Η ταξινόμηση συνόλων όπου απαιτείται γίνεται με χρήση του αλγόριθμους QuickSort.

Εντός του βρόχου που προσθέτει ακμές στο γράφημα, και μετά την προσθήκη της τρέχουσας ακμής, γίνεται έλεγχος και αν δεν έχει δημιουργηθεί κύκλος υπολογίζεται μια έγκυρη, τρέχουσα τοπολογική διάταξη. Αρχικά, τα buckets του προηγούμενου γύρου κρατιούνται σε μία μεταβλητή διότι πρόκειται να τροποποιηθούν. Στη συνέχεια, για κάθε κόμβο που έχει ήδη προστεθεί στο γράφημα (μέσω προηγούμενων ακμών), βρίσκεται το προηγούμενο bucket και υπολογίζεται το καινούριο ώστε να διαπιστωθεί εάν πρέπει να μπει σε σύνολο up ή down.

new\_bucket\_i = int(len(As[node1])\*\*(1.0/len(Ds[node1])))

new\_bucket\_j = 12\*Slen - len(As[node1])

if new\_bucket\_i > old\_bucket\_i:

if up[(new\_bucket\_i, new\_bucket\_j)].count(node1)==0:

up[(new\_bucket\_i, new\_bucket\_j)].append(node1)

if new\_bucket\_i < old\_bucket\_i:

if down[(new\_bucket\_i, new\_bucket\_j)].count(node1)==0:

down[(new\_bucket\_i, new\_bucket\_j)].append(node1)

if new\_bucket\_i == old\_bucket\_i and new\_bucket\_j > old\_bucket\_j:

if up[(new\_bucket\_i, new\_bucket\_j)].count(node1)==0:

up[(new\_bucket\_i, new\_bucket\_j)].append(node1)

if new\_bucket\_i == old\_bucket\_i and new\_bucket\_j < old\_bucket\_j:

if down[(new\_bucket\_i, new\_bucket\_j)].count(node1)==0:

down[(new\_bucket\_i, new\_bucket\_j)].append(node1)

#add node to new bucket

buckets[(new\_bucket\_i, new\_bucket\_j)].append(node1)

Στη συνέχεια γίνεται επεξεργασία των μη κενών συνόλων up και down, με ταξινόμηση των στοιχείων τους με βάση τη θέση στην τοπολογική διάταξη και τοποθέτησή τους με αντίστροφη ή ορθή σειρά στην αρχή ή στο τέλος του νέου τους bucket, όπως καθορίζει ο αλγόριθμος. Παρατίθεται ο κώδικας για τα σύνολο up.

#For each non-empty set UPi,j

#Sort elements in UPi,j in increasing order according to L.

for i in range(i\_limit):

for j in range(j\_limit):

if len(up[(i,j)])>0:

#sort set according to order in list if more than one elements

if len(up[(i,j)])>1:

quickSort(up[(i,j)], L)

startI = len(up[(i,j)]) - 1

while startI>=0:

el = up[(i,j)][startI]

delete(el, L)

pl = placeholders[(i, j)][0]

listInsert(pl, el, L)

startI = startI - 1

Στη συνέχεια, πρέπει να καθοριστεί εάν τα δύο άκρα της ακμής που μόλις προστέθηκε βρίσκονται στο ίδιο bucket διότι τότε θα πρέπει να γίνει περαιτέρω διερεύνηση της σειράς των κόμβων που αυτός περιέχει. Όπως και στην ανίχνευση κύκλου διερευνώνται κόμβοι που είναι τερματικά άκρα ακμών που ξεκινούν από το end ή τους απογόνους του (σύνολο to\_explore) και βρίσκονται πριν το start στην τρέχουσα τοπολογική διάταξη (σύνολο to\_change).

if new\_bucket\_i\_start==new\_bucket\_i\_end and new\_bucket\_j\_start==new\_bucket\_j\_end:

to\_explore\_ts = set()

to\_explore\_ts.add(end)

to\_change = []

while len(to\_explore\_ts)>0:

w = to\_explore\_ts.pop()

if order(start, w, L) == False:

new\_bucket\_i\_w = int(len(As[w])\*\*(1.0/len(Ds[w])))

new\_bucket\_j\_w = 12\*Slen - len(As[w])

if new\_bucket\_i\_start == new\_bucket\_i\_w and new\_bucket\_j\_start == new\_bucket\_j\_w:

to\_change.append(w)

#4d for every edge (w, z)

#add z to explore

#check edges already added

for ge in gedges:

if ge[0] == start and ge[1] == end:

break

else:

if ge[0] == w:

to\_explore\_ts.add(ge[1])

#4d

if len(to\_change)>1:

quickSort(to\_change, L)

#4e For each w in to\_change in decreasing order insert w right after u in L

startI = len(to\_change) - 1

while startI>=0:

el = to\_change[startI]

if (start!=el):

delete(el, L)

listInsert(start, el, L)

startI = startI - 1

# Συμπεράσματα

Στο πλαίσιο αυτής της εργασίας μελετήθηκαν και υλοποιήθηκαν με χρήση της γλώσσας προγραμματισμού Python δύο αλγόριθμοι για την αποδοτική ανίχνευση κύκλου και την τήρηση τοπολογικής διάταξης σε δυναμικά κατευθυνόμενα γραφήματα στα οποία προστίθενται ακμές. Οι δύο σχετιζόμενοι αλγόριθμοι στηρίζονται σε σύνολα που ενημερώνονται με σταθερή αντισταθμιστική πολυπλοκότητα κάθε φορά που προστίθεται μία ακμή όπως τα σύνολα προγόνων και απογόνων ενός κόμβου καθώς και στη δειγματοληψία των κόμβων με μία πιθανότητα O(.

Ο αλγόριθμος της τήρησης τοπολογικής διάταξης ομαδοποιεί τους κόμβους σε κάδους χρησιμοποιώντας δύο δείκτες που βασίζονται στο πλήθος προγόνων και απογόνων και στη συνέχεια ελέγχει τη σειρά των κόμβων εντός κάθε bucket και αν χρειάζεται κάνει εσωτερικές μετακινήσεις. Η σειρά των buckets αντιστοιχεί και στη σειρά των κόμβων στην τοπολογική διάταξη. Η τελική πολυπλοκότητα (το σύνολο όλων των ενημερώσεων) για τους δύο αλγόριθμους είναι για την ανίχνευση κύκλου και για την τήρηση τοπολογικής διάταξης, κάτι που αποτελεί βελτίωση σε σχέση με παλιότερες προσεγγίσεις.

Βιβλιογραφία

1. Bernstein, A. and Chechik, S., 2018. Incremental topological sort and cycle detection in expected total time. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 21-34). Society for Industrial and Applied Mathematics.
2. NetworkX documentation, <http://networkx.github.io/>
3. Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2014. Data structures and algorithms in Java. John Wiley & Sons.
4. Kleinberg, J. and Tardos, E., 2006. Algorithm design. Pearson Education India.