Υλοποίηση Αλγορίθμων για AT-free Γραφήματα

Δημήτριος Σίντος

Διπλωματική Εργασία

Επιβλέπων: Λεωνίδας Παληός

Ιωάννινα, Φεβρουάριος, 2024

ΤΜΉΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΉΣ
ΠΑΝΕΠΙΣΤΉΜΙΟ ΙΩΑΝΝΊΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

ΕΥΧΑΡΙΣΤΙΕΣ

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή μου κ. Λεωνίδα Παληό, για την άριστη συνεργασία μας και τις εύστοχες παρατηρήσεις του σε κάθε βήμα και δυσκολία της εργασίας αυτής. Επίσης, ιδιαίτερα ευχαριστώ τους γονείς και συγγενείς μου για την συνεχή τους υποστήριξη και καθοδήγηση, καθώς και τους συμφοιτητές και φίλους μου για το ενδιαφέρον και την συμπαράστασή τους.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ

Δημήτριος Σίντος, Δίπλωμα, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2024.
Υλοποίηση Αλγορίθμων για AT-free Γραφήματα.
Επιβλέπων: Λεωνίδας Παληός, Καθηγητής.

Αστεροειδές τριάδα (AT σύντομα) είναι ένα σύνολο τριών κορυφών ενός γραφήματος τέτοιο ώστε να υπάρχει μονοπάτι μεταξύ οποιωνδήποτε δύο από αυτές αποφεύγοντας τη γειτονιά της τρίτης. Τα γραφήματα που δεν περιέχουν αστεροειδή τριάδα ονομάζονται AT-free. Η κατηγορία των AT-free γραφημάτων είναι ένας τύπος γραφήματος για τον οποίο πολλά προβλήματα που είναι NP-πλήρη σε γενικότερα γραφήματα μπορούν να λυθούν σε πολυωνυμικό χρόνο.

Σε αυτή τη διπλωματική εργασία έχουμε υλοποιήσει τρεις αλγορίθμους πολυωνυμικού χρόνου για τα AT-free γραφήματα.

Συγκεκριμένα, τον αλγόριθμο υπολογισμού μέγιστου ανεξάρτητου συνόλου των Hajo Broersma , Ton Kloks , Dieter Kratsch , και Haiko Müller[2]. Τον αλγόριθμο υπολογισμού Ελάχιστου Κυρίαρχου Συνόλου του Dieter Kratsch[18]. Και τον αλγόριθμο για το πρόβλημα του 3-Χρωματισμού του Juraj Stacho[16].

ABSTRACT

Dimitrios Sintos, Diploma, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, February 2024.
Implementation of Algorithms for AT-free Graphs.
Advisor: Leonidas Palios, Professor.

Asteroidal triple (AT for short) is a set of three vertices of a graph such that there is a path between any two of them avoiding the neighborhood of the third. Graphs that do not contain an asteroidal triple are called AT-free. The class of AT-free graphs is a type of graph for which many problems that are NP-complete in general graphs can be solved in polynomial time.

In this thesis, we have implemented three polynomial-time algorithms for AT-free graphs.

Specifically, the algorithm for computing the maximum independent set of Hajo Broersma, Ton Kloks, Dieter Kratsch, and Haiko Müller[2]. The algorithm for computing the Minimum Dominating Set of Dieter Kratsch[2]. And the algorithm for the 3-Coloring problem of Juraj Stacho[16].

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

Η αστεροειδή τριάδα (asteroidal triple) εισήχθη το 1962 για να χαρακτηρίσουν τα interval γραφήματα ως εκείνα τα χορδωτά γραφήματα που δεν περιέχουν μια αστεροειδή τριάδα[19]. Αποτελούν μια μεγάλη κατηγορία γραφημάτων που περιλαμβάνει interval, permutation, trapezoid, και cocomparability γραφήματα. Στη παρούσα εργασία παρουσιάζουμε την υλοποίηση τριών αλγορίθμων για τα AT-free γραφήματα που τρέχουν σε πολυωνυμικό χρόνο. Τον αλγόριθμο των Broersma, H., Kloks, T., Kratsch, D. και Müller, H.[2] για τον υπολογισμό του μέγιστου ανεξάρτητου συνόλου, με χρονική πολυπλοκότητα $O(n^4)$. Τον αλγόριθμο του Dieter Kratsch[18] για τον υπολογισμό του ελάχιστου κυρίαρχου συνόλου, με χρονική πολυπλοκότητα $O(n^6)$. Και τον αλγόριθμο του Juraj Stacho[16] για την επίλυση του προβλήματος του 3-χρωματισμού, με χρονική πολυπλοκότητα $O(n^2m)$

1.1 Βασικοί Ορισμοί

Ακολουθούν βασικοί ορισμοί που απαιτούνται για την παρούσα εργασία.

Ορισμός 1.1. *Γράφημα (Graph)* είναι μια δομή που αποτελείται από ένα σύνολο κορυφών που συνδέονται μεταξύ τους με ένα σύνολο ακμών και το συμβολίζουμε με $G = (V, E)$, όπου V και E είναι τα σύνολα των κορυφών και των ακμών αντίστοιχα.

Ορισμός 1.2. *Μονοπάτι (Path)* μεταξύ δύο κορυφών σε ένα γράφημα ονομάζεται μια ακολουθία διαφορετικών κορυφών, όπου κάθε κορυφή της ακολουθίας συνδέεται με την επόμενή της μέσω ακμής.

Ορισμός 1.3. Έστω γράφημα G. Λέμε ότι ένα σύνολο κορυφών $S \subseteq V(G)$ είναι *ανεξάρτητο σύνολο* του G αν κανένα ζεύγος κορυφών από το S δεν είναι ακμή του G.

Ορισμός 1.4. Ο *αριθμός ανεξαρτησίας* $\alpha(G)$ ενός γραφήματος G, είναι το μέγιστο πλήθος ενός ανεξάρτητου συνόλου του G.

Ορισμός 1.5. Ένα *κυρίαρχο σύνολο* ενός γραφήματος G είναι ένα υποσύνολο κορυφών D τέτοιο ώστε κάθε κορυφή του G είτε ανήκει στο D είτε είναι γειτονική με μία κορυφή που ανήκει στο D.

Ορισμός 1.6. Ο *αριθμός κυριαρχίας* $\gamma(G)$ είναι ο αριθμός των κορυφών στο μικρότερο σύνολο κυριαρχίας του G.

Ορισμός 1.7. Ο *χρωματικός αριθμός* ενός γραφήματος G είναι το ελάχιστο πλήθος χρωμάτων που απαιτούνται για να χρωματιστούν οι κορυφές του G έτσι ώστε να μην υπάρχουν δύο γειτονικές κορυφές με το ίδιο χρώμα.

Ορισμός 1.8. Το *πρόβλημα του 3-χρωματισμού* αφορά εάν ένα γραφήμα G μπορεί να χρωματιστεί χρησιμοποιώντας μόνο τρία χρώματα, έτσι ώστε να μην υπάρχουν δύο γειτονικές κορυφές με το ίδιο χρώμα.

Ορισμός 1.9. Μια *αστεροειδής τριάδα* σε ένα γράφημα αποτελείται από τρεις μη γειτονικές κορυφές, έτσι ώστε να υπάρχει μονοπάτι μεταξύ κάθε ζευγαριού από αυτές που αποφεύγει την κλειστή γειτονιά της τρίτης.

Ορισμός 1.10. Ένα γράφημα ονομάζεται *AT-free* εάν δεν περιέχει καμία αστεροειδή τριάδα.

1.2 Αντικείμενο της Διπλωματικής Εργασίας

Στην παρούσα διπλωματική εργασία, μελετούμε και υλοποιούμε τρεις αλγορίθμους για την επίλυση προβλημάτων σε AT-free γραφήματα.

Ειδικότερα, μελετάμε τον αλγόριθμο των Hajo Broersma , Ton Kloks , Dieter Kratsch , και Haiko Müller[2] για τον υπολογισμό του ανεξάρτητου αριθμού. Τον αλγόριθμο του Dieter Kratsch[18] για τον εντοπισμό ελάχιστου κυρίαρχου συνόλου και τον αλγόριθμο του Juraj Stacho[16] για την αντιμετώπιση του προβλήματος του 3-χρωματισμού σε AT-free γραφήματα.

Ο σκοπός αυτής της εργασίας είναι η βαθύτερη κατανόηση της δομής και των αλγοριθμικών ιδιοτήτων των AT-free γραφημάτων, μέσω της εφαρμογής και επικύρωσης αυτών των αλγορίθμων. Όλοι αυτοί οι αλγόριθμοι επιτυγχάνουν το αποτέλεσμα τους σε πολυωνυμικό χρόνο.

1.3 Σχετικά Ερευνητικά Αποτελέσματα

Η εύρεση της αλγοριθμικής πολυπλοκότητας των ανεξάρτητων συνόλων σε AT-free γραφήματα αποτελεί μια σημαντική πρόκληση. Αν και τα ανεξάρτητα σύνολα είναι ένα κλασσικό NP-πλήρες πρόβλημα, τα AT-free γραφήματα παρουσιάζουν μοναδικές προκλήσεις. Σε αντίθεση με άλλες υποκατηγορίες, όπως τα cocomparability γραφήματα, τα γραφήματα AT-free δεν είναι τέλεια. Ως εκ τούτου, οι πολυωνυμικοί αλγόριθμοι που αναπτύχθηκαν για τέλεια γραφήματα, όπως αυτοί των Grötschel, Lovász και Schrijver[22], δεν εφαρμόζονται σε αυτή την περίπτωση.

Για τον υπολογισμό ενός ελάχιστου κυρίαρχου συνόλου έχουν σχεδιαστεί αλγόριθμοι πολυωνυμικού χρόνου για πολλές κατηγορίες γράφων (βλ.[26, 6]). Για παράδειγμα, υπάρχουν αποδοτικοί αλγόριθμοι για τον υπολογισμό ενός ελάχιστου κυρίαρχου συνόλου για τις ακόλουθες κατηγορίες γραφημάτων: interval graphs [21], strongly chordal graphs[11, 20] , cographs[3], permutation graphs[4, 8, 25, 25], k-polygon graphs [7], cocomparability graphs [14, 17], circular-arc graphs [23] and dually chordal graphs[1]. Ιδιαίτερα ενδιαφέρον είναι ο αλγόριθμος των Breu και Kirkpatrick σχετικά με μια υποκατηγορία των AT-free γραφημάτων. Έχουν δώσει αλγορίθμους $O(nm^2)$ για τον υπολογισμό ελάχιστου κυρίαρχου συνόλου και ενός ολικού ελάχιστου κυρίαρχου συνόλου στα cocomparability γραφήματα[14]

Το πρόβλημα του χρωματισμού είναι ένα από τα πρώτα προβλήματα που γνωρίζουμε ότι είναι NP-hard[10]. Αυτό ισχύει και για ιδικές κλάσεις γραφημάτων όπως τα planar graphs[9], line graphs[15], regular graphs[5] ή ακόμα και για σταθερό αριθμό k χρωμάτων (γνωστό και ώς το πρόβλημα του 3-χρωματισμού)[9]. Αντίθετα, για σενάρια με δύο ή λιγότερα χρώματα, το πρόβλημα επιλύεται σε πολυωνυμικό χρόνο. Αυτό ισχύει επίσης για ορισμένες κατηγορίες γραφημάτων με μοναδικές ιδιότητες, όπως τα interval graphs [12], chordal graphs [12], comparability graphs [12], και γενικότερα για τέλεια γραφήματα[13].

1.4 Δομή της Διπλωματικής Εργασίας

Η διπλωματική εργασία αναπτύσσεται σε τρία κεφάλαια. Στο κεφάλαιο 2 θα μελετήσουμε του τρεις αλγορίθμους. Στο επόμενο κεφάλαιο 3, παρουσιάζεται η λεπτομερής αναπαράσταση των δεδομένων, οι χρήσιμες δομές δεδομένων και η υλοποίηση κάθε αλγορίθμου. Στο τελευταίο κεφάλαιο ?? γίνεται σύνοψη των ευρημάτων, αναφέρονται δυνατές βελτιώσεις και προτείνεται πώς η υπάρχουσα υλοποίηση θα μπορούσε να τροποποιηθεί για την εκπόνηση πιστοποιητικών που επιβεβαιώνουν την ακρίβεια των αποτελεσμάτων.

ΚΕΦΑΛΑΙΟ 2

Οι Αλγόριθμοι

Σε αυτό το κεφάλαιο θα μελετήσουμε με λεπτομέρεια όλους τους αλγορίθμους που υλοποιήσαμε για τα AT-free γραφήματα. Για κάθε αλγόριθμο θα δώσουμε τον συμβολισμό και τα λήμματα που χρειάζονται για την περιγραφή του. Αμέσως μετά, θα εξηγήσουμε την πολυπλοκότητά του και συγκεκριμένα βήματά του, που θεωρήσαμε πιο ιδιαίτερα. Τέλος θα δώσουμε και ένα απλό παράδειγμα επίλυσης του κάθε αλγορίθμου.

Διευκρινίζουμε ότι δεν αποδεικνύουμε την ορθότητα του κάθε αλγορίθμου που χρησιμοποιούμε. Οι αποδείξεις αυτές βρίσκονται στις αντίστοιχες αναφορές([2], [18], [16])

2.1 Υπολογισμός Μέγιστου Ανεξάρτητου Συνόλου

Συμβολίζουμε τον αριθμό των κορυφών ενός γραφήματος $G = (V, E)$ με n και τον αριθμό των ακμών με m. Υπενθυμίζουμε ότι ένα ανεξάρτητο σύνολο σε ένα γράφημα G είναι ένα σύνολο από ζεύγη με μη γειτονικές κορυφές. Ο αριθμός ανεξαρτησίας ενός γραφήματος G, συμβολίζεται ως $\alpha(G)$ είναι το μέγεθος του μεγαλύτερου ανεξάρτητου συνόλου στο γράφημα. Οι βασικές δομικές ιδιότητες που πρέπει να αναλύσουμε πριν την περιγραφή του αλγορίθμου είναι τα $Components$ και τα $Intervals$.

Σε ένα AT-free γράφημα G όπου το x και το y είναι δύο ξεχωριστές μη γειτονικές κορυφές του G. Συμβολίζουμε με $C^x(y)$ το $Component$ του $G - N[x]$ όπου εμπεριέχεται το y, και με $r(x)$ τον αριθμό των $Components$ του $G - N[x]$.

Ορισμός 2.1. Μια κορυφή $z \in V \setminus \{x, y\}$ είναι μεταξύ των κορυφών x και y εάν οι

x και z βρίσκονται στον ίδιο $Component$ του $G - N[x]$. Εναλλακτικά, η κορυφή z είναι μεταξύ των x και y στο γράφημα G αν υπάρχει μονοπάτι από τον x στον z που αποφεύγει το $N[y]$ και μονοπάτι από τον y στον z που αποφεύγει το $N[x]$.

Ορισμός 2.2. Το διάστημα $I = I(x, y)$ του G είναι το σύνολο όλων των κορυφών του G που βρίσκονται μεταξύ x και y. Συνεπώς, $I(x, y) = C_x(y) \cap C_y(x)$.

Ο αλγόριθμος των Broersma, H., Kloks, T., Kratsch, D. και Müller, H. του καθορίζει τον αριθμό ανεξαρτησίας κάθε $Component$ και κάθε $Interval$ χρησιμοποιώντας τις σχέσεις που δίνονται στα Λήμματα 2.1, 2.2 και 2.3.

Λήμμα 2.1. *Έστω ότι $G = (V, E)$ είναι οποιαδήποτε γράφημα. Τότε*

$$\alpha(G) = 1 + \max_{x \in V} \left(\sum_{i=1}^{r(x)} \alpha(C_i^x) \right)$$

όπου $C_1^x, C_2^x, ..., C_{r(x)}^x$ τα $Compontes$ του $G - N[x]$.

Λήμμα 2.2. *Έστω ότι $G = (V, E)$ είναι ένα AT-free γράφημα. Έστω $x \in V$ και έστω C^x ένα $Component$ του $G - N[x]$. Τότε*

$$\alpha(C^x) = 1 + \max_{y \in C^x} \left(\alpha(I(x, y)) + \sum_i \alpha(D_i^y) \right)$$

όπου τα D_i^y είναι $Components$ του $G - N[y]$ που εμπεριέχονται στο C^x.

Λήμμα 2.3. *Έστω ότι $G = (V, E)$ είναι ένα AT-free γράφημα. Έστω $I = I(x, y)$ είναι ένα $Interval$ του G. Αν $I = \emptyset$ τότε $\alpha(I) = 0$. Αλλιώς*

$$\alpha(I) = 1 + \max_{s \in I} \left(\alpha(I(x, s)) + \alpha(I(s, y)) + \sum_i \alpha(C_i^s) \right)$$

Σε αυτό το σημείο μπορούμε να δώσουμε τον αλγόριθμο για τον υπολογισμό του αριθμού ανεξαρτησίας $\alpha(G)$ για AT-free γραφήματα, ο οποίος είναι βασισμένος στον δυναμικό προγραμματισμό.

Αλγόριθμος 2.1 Αλγόριθμος υπολογισμού αριθμού ανεξαρτησίας σε AT-free γραφήματα

Είσοδος: Ένα AT-free γράφημα G.

Έξοδος: Αριθμός ανεξαρτησίας $\alpha(G)$

1: Για κάθε $x \in V$, υπολόγισε όλα τα $Components$ $C_1^x, C_2^x, \ldots, C_{r(x)}^x$
2: Για κάθε ζευγάρι μη γειτονικών κορυφών x και y, υπολόγισε το $Interval$ $I(x, y)$.
3: Ταξινόμησε όλα τα $Components$ και τα $Intervals$ με βάση το μη-αύξοντα αριθμό κορυφών.
4: Υπολόγισε τα $\alpha(C)$ και $\alpha(I)$ για κάθε $Components$ C και κάθε $Interval$ I με τη σειρά του Βήματος 3.
5: Υπολόγισε το $\alpha(G)$.

Ορισμός 2.3. Υπάρχει αλγόριθμος χρόνου $O(n^4)$ για τον υπολογισμό του ανεξάρτητου αριθμού ενός AT-free γραφήματος.

Για την πολυπλοκότητα του αλγορίθμου μελετάμε κάθε βήμα ξεχωριστά. Το πρώτο βήμα μπορεί να υλοποιηθεί σε χρόνο $O(n(n + m))$ χρησιμοποιώντας έναν γραμμικό αλγόριθμο για τον υπολογισμό των $Components$.

Το βήμα 2 υπολογίζει $Intervals$ για τις μη γειτονικές κορυφές x και y, χρησιμοποιώντας την τομή των συνιστωσών $C_x(y)$ και $C_y(x)$. Η διαδικασία, που εκτελείται σε χρόνο O(n) για κάθε $Interval$, οδηγεί σε συνολική χρονική πολυπλοκότητα $O(n^3)$. Η υλοποίηση αξιοποιεί ένα λεξικό με αντικείμενα της κλάσης $Interval$, παρέχοντας έναν αποτελεσματικό τρόπο διαχείρισης και υπολογισμού εντός το πολύ n^2 $Component$ και $Intervals$.

Με την χρήση του Bucket sort το βήμα 3 υλοποιείται σε $O(n^2)$.

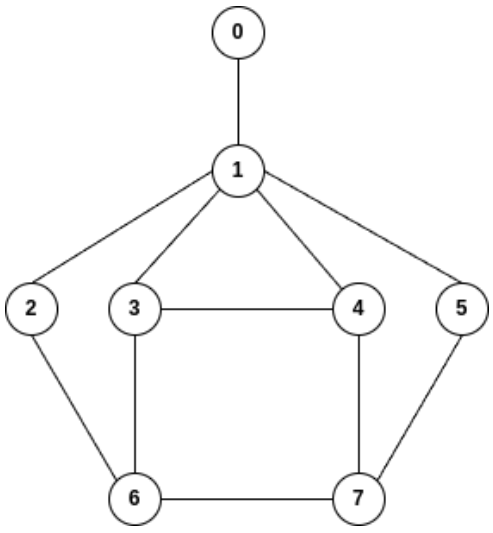
Το σημείο φραγμού για τη χρονική πολυπλοκότητα του αλγορίθμου μας είναι το βήμα 4 αφού εκτελείται σε $O(n^4)$. Για κάθε $Component$ C_x του $G - N[x]$ και μια κορυφή $y \in C_x$, ο αλγόριθμος πρέπει να υπολογίσει τα $Components$ του $G - N[y]$ που περιέχονται στην C_x. Αυτό γίνεται σε χρόνο $O(|C_x|)$ για σταθερές κορυφές x και y στο C_x, με αποτέλεσμα ο συνολικός χρόνος να είναι $O(n^3)$ για όλα τα $Compoents$. Θεωρώντας ένα $Interval$ $I(x, y)$ και μια κορυφή $s \in I$, ο αλγόριθμος πρέπει να αθροίσει τους αριθμούς ανεξαρτησίας των $Components$ C_s του $G - N[s]$ που περιέχονται στο I. Η λειτουργία αυτή απαιτεί χρόνο $O(|I(x, y)|)$ για ένα σταθερό $I(x, y)$ και μια κορυφή $s \in I$. Ο συνολικός χρόνος για τον υπολογισμό του $\alpha(I)$ για

όλα τα $Intervals$ είναι $O(n^3)$.

Είναι σαφές ότι το βήμα 5 μπορεί να γίνει σε χρόνο $O(n^2)$. Έτσι ο χρόνος εκτέλεσης του αλγορίθμου μας είναι $O(n^4)$.

Ο αλγόριθμος έτσι όπως παρουσιάζεται από τους Broersma, H., Kloks, T., Kratsch, D. και Müller, H υπολογίζει τον αριθμό ανεξαρτησίας ενός AT-free γραφήματος. Έχουμε επεκτείνει την υλοποίηση του αλγορίθμου έτσι ώστε στο βήμα 5 ο αλγόριθμος να επιστρέφει και ένα μέγιστο ανεξάρτητο σύνολο. Ο τρόπος με τον οποίο το πετύχαμε αυτό είναι προσθέτοντας μια επιπλέον δομή αποθήκευσης, στην περίπτωση μας ένα λεξικό. Κάθε φόρα που υπολογίζαμε το μέγιστο alpha είτε για τα $Intervals$ 2.3, είτε για τα $Components$ 2.2 αποθηκεύαμε στο λεξικό όλες τις ακμές από τα $Intervals$ και $Components$ που χρησιμοποιήθηκαν για να υπολογιστεί το μέγιστο alpha. Με αυτόν τον τρόπο στο τελευταίο βήμα είμαστε σε θέσει να επιστρέψουμε όλους τους κόμβους που χρειάστηκαν για τον υπολογισμό των alpha κάθε $Component$ του αθροίσματος 2.1 καθώς επίσης και το x που αντιστοιχεί στο 1 του τύπου.

Παρακάτω παραθέτουμε ένα παράδειγμα εκτέλεσης του αλγορίθμου για το AT-free γράφημα 2.1.



Σχήμα 2.1: Ένα AT-free γράφημα με 8 κόμβους

Μετά την εκτέλεση του πρώτου βήματος τα $Components$ που υπολογίσαμε φαίνονται στον πίνακα 2.1

Πίνακας 2.1: Components του γραφήματος μετά την εκτέλεση του πρώτου βήματος

Component	**Σύνολο κορυφών**	**Alpha**
C_1^1	{6, 7}	-
C_1^4	{2, 6}	-
C_2^4	{0}	-
C_3^4	{5}	-
C_1^3	{2}	-
C_2^3	{7, 5}	-
C_3^3	{0}	-
C_1^2	{3, 4, 5, 7}	-
C_2^2	{0}	-
C_1^0	{4, 3, 2, 6, 7, 5}	-
C_1^6	{1, 4, 0, 5}	-
C_1^7	{1, 3, 2, 0}	-
C_1^5	{3, 4, 6, 2}	-
C_2^5	{0}	-

Για το δεύτερο βήμα του αλγορίθμου, παίρνοντας όλους τους συνδυασμούς μη γειτονικών κορυφών για το παρών γράφημα. Τα $Intervals$ που προκύπτουν φαίνονται στον παρακάτω πίνακα2.2.

Πίνακας 2.2: Intervals του γραφήματος μετά την εκτέλεση του δεύτερου βήματος

Intervals	**Σύνολο κορυφών**	**Alpha**
I(2, 5)	{3, 4}	-
I(2, 7)	{3}	-
I(5, 2)	{3, 4}	-
I(5, 6)	{4}	-
I(0, 7)	{2, 3}	-
I(0, 6)	{4, 5}	-
I(7, 2)	{3}	-
I(7, 0)	{2, 3}	-
I(6, 5)	{4}	-
I(6, 0)	{4, 5}	-

Όπως φαίνεται και στους πίνακες 2.3 και 2.4,μετά το τρίτο βήμα τα *Components* και *Intervals* ταξινομούνται με βάση το πλήθος των κορυφών τους, από το μικρότερο στο μεγαλύτερο.

Πίνακας 2.3: Ταξινόμηση Components με βάση το σύνολο κορυφών

Components	**Σύνολο Κορυφών**	**Alpha**
C_2^4	{0}	-
C_3^4	{5}	-
C_1^3	{2}	-
C_3^3	{0}	-
C_2^2	{0}	-
C_2^5	{0}	-
C_1^1	{6, 7}	-
C_1^4	{2, 6}	-
C_2^3	{7, 5}	-
C_1^2	{3, 4, 5, 7}	-
C_1^6	{1, 4, 0, 5}	-
C_1^7	{1, 3, 2, 0}	-
C_1^5	{3, 4, 6, 2}	-
C_1^0	{4, 3, 2, 6, 7, 5}	-

Πίνακας 2.4: Ταξινόμηση Intervals με βάση το σύνολο κορυφών

Intervals	**Σύνολο Κορυφών**	**Alpha**
I(2, 7)	{3}	-
I(5, 6)	{4}	-
I(7, 2)	{3}	-
I(6, 5)	{4}	-
I(2, 5)	{3, 4}	-
I(5, 2)	{3, 4}	-
I(0, 7)	{2, 3}	-
I(0, 6)	{4, 5}	-
I(7, 0)	{2, 3}	-
I(6, 0)	{4, 5}	-

Χρησιμοποιώντας τα λήμματα 2.1, 2.2 και εφαρμόζοντας τεχνικές δυναμικού προγραμματισμού, μετά την εκτέλεση του βήματος τέσσερα έχουν υπολογιστεί τα alpha όλων των $Components$ και $Intervals$. Τα αποτελέσματα φαίνονται στους πίνακες 2.5 και 2.6

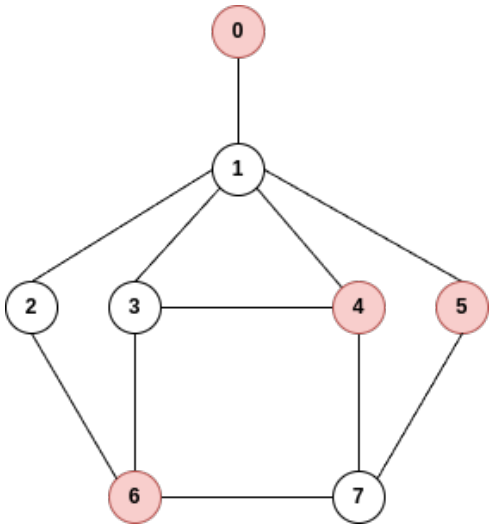
Πίνακας 2.5: Πίνακας Components μετά την εκτέλεση του βήματος 4 του αλγορίθμου

Components	**Σύνολο Κορυφών**	**Alpha**
C_2^4	$\{0\}$	1
C_3^4	$\{5\}$	1
C_1^3	$\{2\}$	1
C_3^3	$\{0\}$	1
C_2^2	$\{0\}$	1
C_2^5	$\{0\}$	1
C_1^1	$\{6, 7\}$	1
C_1^4	$\{2, 6\}$	1
C_2^3	$\{7, 5\}$	1
C_1^2	$\{3, 4, 5, 7\}$	2
C_1^6	$\{1, 4, 0, 5\}$	3
C_1^7	$\{1, 3, 2, 0\}$	3
C_1^5	$\{3, 4, 6, 2\}$	2
C_1^0	$\{4, 3, 2, 6, 7, 5\}$	3

Πίνακας 2.6: Πίνακας Intervals μετά την εκτέλεση του βήματος 4 του αλγορίθμου

Intervals	**Σύνολο Κορυφών**	**Alpha**
I(2, 7)	{3}	1
I(5, 6)	{4}	1
I(7, 2)	{3}	1
I(6, 5)	{4}	1
I(2, 5)	{3, 4}	1
I(5, 2)	{3, 4}	1
I(0, 7)	{2, 3}	2
I(0, 6)	{4, 5}	2
I(7, 0)	{2, 3}	2
I(6, 0)	{4, 5}	2

Στο τελευταίο βήμα με την εφαρμογή του λήμματος 2.1, είναι προφανές ότι ο αριθμός ανεξαρτησίας για το παρόν γράφημα είναι τέσσερα. Αυτό μπορεί, στο συγκεκριμένο παράδειγμα, να προκύψει από διάφορες κορυφές. Στο παρακάτω σχήμα φαίνεται ένα μέγιστο ανεξάρτητο σύνολο που έχει προκύψει από τον κόμβο 4.



Σχήμα 2.2: Μέγιστο Ανεξάρτητο Σύνολο σε AT-free γράφημα

2.2 Υπολογισμός Ελάχιστου Κυρίαρχου Συνόλου

2.3 Το Πρόβλημα του 3-Χρωματισμού

ΚΕΦΑΛΑΙΟ 3

Η Υλοποίηση

Υλοποιήσαμε όλους τους αλγορίθμους που αναφέραμε στο Κεφάλαιο 2 και σχεδιάσαμε μια γραφική διαπροσωπεία για την οπτικοποίηση της εισόδου και των αποτελεσμάτων. Η υλοποίηση έγινε στη γλώσσα προγραμματισμού **Python 3** και η οπτικοποίηση με την βιβλιοθήκη **PyVis**[24]

3.1 Οργάνωση Κώδικα

Η γενική αρχιτεκτονική για την επίλυση του κάθε αλγορίθμου είναι η ίδια σε κάθε περίπτωση. Πιο συγκεκριμένα η υλοποίηση οργανώνεται στα αρχεία:

- `main.py` : η αρχή του εκτελέσιμου προγράμματος.
- `graph.py` : η διευθύνουσα κλάση του γραφήματος.
- `polynomial_time_algorithm.py` : η διευθύνουσα κλάση για την υλοποίηση του αλγορίθμου.

Επιπλέον αρχεία υλοποιούν κλάσεις οι οποίες είναι απαραίτητες για την υλοποίηση των επιμέρους αλγορίθμων. Αναλυτικά θα τις αναλύσουμε στις αντίστοιχες ενότητες.

3.2 Είσοδος - Έξοδος

Κάθε αλγόριθμος που έχουμε υλοποιήσει παίρνει σαν είσοδο ένα AT-free γράφημα. Η αναπαράσταση του γραφήματος έχει γίνει με την κλάση Graph. Έχουμε επιλέξει ο χρήστης μπορεί να ορίσει ένα γράφημα σε αρχείο JSON καθώς η διαχείριση του είναι αρκετά εύκολη από γλώσσες προγραμματισμού όπως η **Python**.

Το αρχείο JSON έχει τα κλειδιά "num_of_vertices" και "num_of_edges" τα οποία δηλώνουν τον αριθμό των κόμβος του γραφήματος και ακμών αντίστοιχα. Οι τιμές που παίρνουν είναι θετική ακέραιοι $\mathbb{N}$. Επιπλέον υπάρχει το κλειδί "edges" το οποίο έχει όλες τις ακμές του γραφήματος. Η δήλωση των ακμών γίνεται με έναν δισδιάστατο πίνακα που περιέχει ζεύγη ακεραίων που υποδηλώνουν τους συνδεδεμένους κόμβους.

Το αρχείο υφίσταται μια ανάλυση προκειμένου να επιβεβαιωθεί η απουσία AT (asteroidal triple). Αυτή η ανάλυση δεν είναι η βέλτιστη αλλά μας διασφαλίζει ότι το γράφημα είναι AT-free. Η κλάση για την ανάλυση του αρχείου:

Listing 3.1: graph_parser.py

```
import networkx as nx
import json
from graph import Graph
from itertools import combinations


class GraphParser:
    @staticmethod
    def parse_graph_from_file(file_path):
        with open(file_path, 'r') as file:
            data = json.load(file)

        num_of_vertices = data['num_of_vertices']
        num_of_edges = data['num_of_edges']
        edges = [tuple(edge) for edge in data['edges']] # Convert edge lists to tuples

        # Verify if we got the correct number of edges
        if len(edges) != num_of_edges:
            raise ValueError("The number of edges specified does not match the number of edges
                parsed.")

        if GraphParser.has_asteroidal_triple(num_of_vertices, edges):
```

```
            raise ValueError("The graph is no AT-free.")
        else:
            return Graph(num_of_vertices, num_of_edges, edges)

    @staticmethod
    def has_asteroidal_triple(num_of_vertices, edges):
        # Create a graph
        G = nx.Graph()
        G.add_nodes_from(range(num_of_vertices))
        G.add_edges_from(edges)

        # Generate all possible combinations of 3 vertices
        vertex_combinations = combinations(range(num_of_vertices), 3)

        # Check for each combination
        for v1, v2, v3 in vertex_combinations:
            if GraphParser.has_path_avoiding_neighbors(G, v1, v2, v3) \
                    and GraphParser.has_path_avoiding_neighbors(G, v2,v3,v1) \
                        and GraphParser.has_path_avoiding_neighbors(G, v3, v1, v2):
                print("Found an asteroidal triple: ", v1, v2, v3)
                return True
        return False

    @staticmethod
    def has_path_avoiding_neighbors(G, source, target, avoid):
        """
        Check if there is a path from source to target in graph G
        avoiding neighbors of avoid.
        """
        neighbors_to_avoid = set(G.neighbors(avoid))
        visited = set()
        queue = [source]

        while queue:
            node = queue.pop(0)
            visited.add(node)
            if node == target:
                return True
            for neighbor in G.neighbors(node):
                if neighbor != avoid and neighbor not in visited and neighbor not in queue and
                    neighbor not in neighbors_to_avoid:
```

```
                queue.append(neighbor)

    return False
```

Αν η ανάλυση του γραφήματος είναι επιτυχημένη, τότε δημιουργείται ένα αντικείμενο τύπου Graph. Παρακάτω παρουσιάζεται ο ορισμός του Graph, καθώς και τα πεδία που δημιουργούνται. Τα συγκεκριμένα πεδία είναι κοινά και για τους τρεις αλγορίθμους που έχουμε υλοποιήσει.

Listing 3.2: Ορισμός κλάσης Graph

```
class Graph:
        def __init__(self, num_of_vertices,  num_of_edges, edges, vertices=None):
                self.num_of_vertices = num_of_vertices
                self.num_of_edges = num_of_edges
                self.vertices = set(str(i) for i in range(num_of_vertices)) if vertices is
                     None else vertices
                self.edges = set((str(u), str(v)) for u, v in edges)
                self.adjacency_list = {str(vertex): set() for vertex in self.vertices}
                for edge in self.edges:
                        u, v = edge
                        self.adjacency_list[u].add(v)
                        self.adjacency_list[v].add(u)
```

Η έξοδος στον αλγόριθμο υπολογισμού μέγιστου ανεξάρτητου συνόλου είναι ο αριθμός ανεξαρτησίας του γραφήματος καθώς και ένα σετ από κόμβους που είναι ένα μέγιστο ανεξάρτητο σύνολο.

Ο αλγόριθμος υπολογισμού του ελάχιστου κυρίαρχου συνόλου επιστρέφει ένα σετ κόμβων που αποτελεί ένα ελάχιστο κυρίαρχο σύνολο.

Για το πρόβλημα του 3-χρωματισμού, η έξοδος του αλγορίθμου παρουσιάζεται σε μορφή λεξικού, όπου κάθε κόμβος του γραφήματος αντιστοιχεί σε ένα κλειδί, και η σχετική τιμή είναι το χρώμα που έχει ανατεθεί σε αυτόν τον κόμβο (π.χ., ”red”, ”green”, ”blue”).

Σε όλους τους αλγορίθμους εκτελούμε την συνάρτηση `show()` της κλάσης Graph. Η συνάρτηση επιστρέφει ένα αρχείο HTML το οποίο περιέχει την οπτικοποίηση του γραφήματος.

3.3 Δομές Δεδομένων

Οι δομές που χρησιμοποιήσαμε για την επίλυση των αλγορίθμων είναι δομές που μας παρέχει η γλώσσα προγραμματισμού **Python**.Η επιλογή των δομών έγινε με σκοπό τη διατήρηση της χωρικής και χρονικής πολυπλοκότητας για κάθε αλγόριθμο. Παρόλο που σε ορισμένες περιπτώσεις θα μπορούσαμε να επιτύχουμε μείωση της χωρικής πολυπλοκότητας εάν είχαμε επιλέξει τη δομή της λίστας αντί του λεξικού, προτιμήσαμε να διατηρήσουμε τη σαφήνεια και την ευανάγνωστη μορφή του κώδικά μας. Κατανοούμε ότι η κατανόηση του κώδικα είναι εξίσου σημαντική με τυχόν οικονομία σε χώρο που θα μπορούσαμε να επιτύχουμε.

Αρχικά η κλάση Graph 3.2 διαχειρίζεται τις παρακάτω δομές δεδομένων που είναι κοινές και για τους τρεις αλγορίθμους που έχουμε υλοποιήσει:

1. πλειάδα `tuple`: Οι πλειάδες (tuples) στην **Python** αποτελούν μια αμετάβλητη δομή δεδομένων που επιτρέπει την ομαδοποίηση στοιχείων με σειρά. Οι ακμές του γραφήματος αναπαρίστανται ως πλειάδες, δηλαδή ζεύγη ακεραίων αριθμών.

2. δομή συνόλων `set`: Τα σύνολα στην **Python** είναι δομές δεδομένων που αποθηκεύουν μοναδικά στοιχεία, χωρίς να διατηρούν τη σειρά εισαγωγής. Η μοναδικότητα επιτυγχάνεται μέσω της χρήσης hash tables, που επιτρέπουν γρήγορες λειτουργίες εισαγωγής, διαγραφής και αναζήτησης με χρόνο εκτέλεσης $O(1)$ σε μέσο όρο. Κατά τη διάρκεια της εισαγωγής, το στοιχείο υπολογίζει ένα hash value, το οποίο καθορίζει τη θέση του στο hash table. Κατά την αναζήτηση, το hash table χρησιμοποιείται για να εντοπίσει γρήγορα τη θέση του στοιχείου. Παρόλα αυτά, σε περιπτώσεις συγκρούσεων (hash collisions), όπου δύο στοιχεία έχουν το ίδιο hash value, η απόδοση μπορεί να μειωθεί, αυξάνοντας τον χρόνο αναζήτησης στο χειρότερο σενάριο σε $O(n)$. Χρησιμοποιούμε τα σύνολα για να ορίσουμε τους κόμβους και τις ακμές του γραφήματος.

3. λεξικό `dictionary`: Τα λεξικά στην **Python** είναι δομές δεδομένων που επιτρέπουν την αποθήκευση και την ανάκτηση δεδομένων με βάση τα κλειδιά. Κάθε κλειδί στο λεξικό είναι μοναδικό και συσχετίζεται με μια τιμή. Η κύρια ιδιότητα των λεξικών είναι η αποδοτική αναζήτηση, καθώς η πρόσβαση σε μια τιμή γίνεται σε σταθερό χρόνο $O(1)$ χάρη στη χρήση hash table. Τα λεξικά μπορούν να περιέχουν διάφορους τύπους δεδομένων ως τιμές, όπως

αριθμούς, συμβολοσειρές, λίστες, και ακόμη και άλλα λεξικά. Η ενημέρωση, εισαγωγή, και διαγραφή στοιχείων σε ένα λεξικό γίνεται με διαφορετικές μεθόδους, προσφέροντας μεγάλη ευελιξία στη διαχείριση τους. Χρησιμοποιούμε τα λεξικά για να ορίσουμε τις λίστες γειτνίασης. Κάθε κόμβος (κλειδί) είναι συσχετισμένος με ένα σύνολο από γειτονικούς κόμβους. Η επανάληψη μέσω των ακμών και η ενημέρωση των συνόλων γειτονικών κόμβων είναι γραμμική ως προς τον αριθμό των ακμών. Συνεπώς, ο συνολικός χρόνος εκτέλεσης για τη δημιουργία των λιστών γειτνίασης είναι $O(m)$, όπου m είναι ο αριθμός των ακμών που παρέχονται στη κλάση Graph.

Πιο συγκεκριμένα θα δούμε τις δομές δεδομένων που χρησιμοποιήσαμε για κάθε αλγόριθμο στις παρακάτω παραγράφους.

Ο Αλγόριθμος υπολογισμός μέγιστου ανεξάρτητου συνόλου 2.1 χρειάζεται και διαχειρίζεται επιπλέον τις παρακάτω δομές δεδομένων:

1. λεξικά: Η κλάση Graph αυτού το αλγορίθμου έχει επιπλέον μεταβλητές για την επίλυση του. Πιο συγκεκριμένα ένα λεξικό για τα $Components$ χρησιμοποιεί ως κλειδί μια πλειάδα, όπου το πρώτο στοιχείο αναφέρεται στον κόμβο προέλευσης, ενώ το δεύτερο στοιχείο (αριθμός i) καθορίζει τη μοναδικότητα του $Component$. Αυτό σημαίνει ότι για κάθε κόμβο, μπορεί να υπάρχουν πολλαπλά $Components$. Η χρήση μιας πλειάδας ως κλειδί επιτρέπει τον εύκολο και γρήγορο προσδιορισμό και ανάκτηση των $Components$ που αντιστοιχούν σε έναν συγκεκριμένο κόμβο. Επιπλέον χρησιμοποιούμε ένα λεξικό το οποίο έχει κλειδιά τους κόμβους x του γραφήματος και σαν τιμές τον αριθμό των $Componentes$ που έχουν στο $G - N[x]$. Αντίστοιχα για τα $Intervals$ ένα λεξικό με κλειδιά πλειάδες με μη γειτονικές ακμές. Ένα λεξικό που για όλες τις μη γειτονικές κορυφές x και y έχει ως τιμή ένας δείκτης $P(x, y)$ στο $Component$ του $C^x(y)$. Για την κλάση PolynomialTimeAlgorithm, απαιτούνται δύο επιπλέον λεξικά, τα οποία έχουν κλειδιά παρόμοια με τα λεξικά που διατηρούν τις πληροφορίες για τα $Components$ και τα $Intervals$. Τα νέα λεξικά, λειτουργούν ως αντίγραφα για την αποθήκευση των κόμβων που χρησιμοποιήθηκαν για τον υπολογισμό των μέγιστων τιμών από τους τύπους 2.2 και 2.3. Η χρήση αυτών των δομών είναι απαραίτητη προκειμένου ο αλγόριθμος να είναι σε θέση να επιστρέψει, εκτός από τον αριθμό ανεξαρτησίας, και το ανεξάρτητο σύνολο που προκύπτει από την εκτέλεση των εν λόγω τύπων.

Για τον Αλγόριθμο υπολογισμού ελάχιστου κυρίαρχου συνόλου 2.2 χρειάζεται η παρακάτω δομή δεδομένων:

1. λεξικά: Στην κλάση PolynomialTimeAlgorithm όπου υλοποιούμε τον αλγόριθμο έχουμε αναπαραστήσει την δομή H, η οποία διατηρεί τα BFS-levels του γραφήματος για κάποιον κόμβο $x \in V$, με ένα λεξικό. Το παίρνει ως κλειδί έναν θετικό ακέραιο που δηλώνει το επίπεδο του BFS και σαν τιμή ένα σύνολο `set` με τους κόμβους που υπάρχουν στο εκάστοτε επίπεδο.

   Επιπλέον ένα λεξικό για να ορίσουμε την δομή A όπου σαν κλειδί έχει τον αριθμό i σε κάθε βρόχο της εκτέλεση του αλγορίθμου και σαν τιμή έχει μια ουρά που περιέχει μια ταξινομημένη πλειάδα $(S, S, val(S))$.

Τέλος για την επίλυση του Πρόβληματος 3-Χρωματισμού χρησιμοποιήσαμε επιπλέον τις δομές:

1. λεξικό: Στην κλάση Graph χρειαστήκαμε μια δομή λεξικού για να αποθηκεύει τα $Blocks$ του γραφήματος.

2. σύνολο: Για τον χρωματισμό του γραφήματος χρειάστηκε να αποθηκεύσουμε σε ένα σύνολο τα $cutpoints$ του.

3.4 Υπολογισμός Μέγιστου Ανεξάρτητου Συνόλου

Στο παρόν κεφάλαιο, παρουσιάζουμε ενδεικτικές κλάσεις που αποτελούν το αλγοριθμικό τμήμα για τον υπολογισμό του μέγιστου ανεξάρτητου συνόλου. Προχωρούμε στην παρουσίαση αυτών των κλάσεων με τη σειρά που καλούνται στο πλαίσιο του συνολικού αλγοριθμικού σχεδιασμού. Σημειώνουμε πως παραλείπουμε τις κλάσεις που υλοποιούν αλγορίθμους από άλλες εργασίες, όπως ο αλγόριθμος του Bucket sort.

3.4.1 Η Κλάση PolynomialTimeAlgorithm

Η κλάση `PolynomialTimeAlgorithm` αποτελεί την κύρια κλάση για τον υπολογισμό του μέγιστου ανεξάρτητου συνόλου.

polynomial_time_algorithm.py

```
from graph import Graph
from bucket_sort import BucketSort

from itertools import combinations



class PolynomialTimeAlgorithm:
    def __init__(self,graph : Graph):
        self.graph = graph

        self.max_interval_vertices_set = {}
        self.max_component_vertices_set = {}




    def computing_independent_set(self):
        # Step 1. For every x □V compute all components C1x , C2x , . . . , Cr(x)
        # Step 2. For every pair of nonadjacent vertices x and y compute the interval I(x, y).
        # Step 3. Sort all the components and intervals according to nondecreasing number of
        # vertices.
        # Step 4. Compute α(C) and α(I) for each component C and each interval I in the
        # order of Step 3.
        # Step 5. Compute α(G).

        # Step 1
        self.graph.compute_all_components()
        print("\nComponents:")
        for component in self.graph.components.values():
            print(component)

        print("self.graph.num_of_components:", self.graph.num_of_components)



        # Step 2
        self.graph.compute_all_intervals()
        print("\nIntervals:")
        for interval in self.graph.intervals.values():
            print(interval)
```

```
    # Step 3

    # Sort components
    bucket_sort_components = BucketSort(list(self.graph.components.keys()), key_function=
        lambda x: len(self.graph.components[x]))
    sorted_components = bucket_sort_components.sort()

    # Sort intervals
    bucket_sort_intervals = BucketSort(list(self.graph.intervals.keys()), key_function=
        lambda x: len(self.graph.intervals[x]))
    sorted_intervals = bucket_sort_intervals.sort()

    print("\nSorted components:", sorted_components, len(sorted_components), type(
        sorted_components))
    print("\nSorted intervals:", sorted_intervals, len(sorted_intervals), type(
        sorted_intervals))
    # Step 4
    for key in sorted_components:
        self.alpha_C(key)

    # Step 5
    print('\n\n\n')
    print("Graph intependent set number:", self.alpha_G())


def alpha_C(self,component_key):
    component = self.graph.components[component_key]
    print(component)
    if component.alpha is not None:
        return component.alpha

    x = component.x

    component_vertices = component.vertices

    max_alpha = 0
    self.max_component_vertices_set[component_key] = set()
    max_y = None
    max_D_y_components_vertices = set()
    for y in component_vertices: # y E Cx
```

```
            alpha_I_xy = self.alpha_I((x, y))

            D_y_components = self.compute_D_iy(y, component_vertices)

            alpha_D_sum = sum(self.alpha_C(D_iy) for D_iy in D_y_components)
            if max_alpha <= alpha_I_xy + alpha_D_sum:
                max_alpha = alpha_I_xy + alpha_D_sum
                max_y = y
                max_D_y_components_vertices = set()
                for D_iy in D_y_components:
                    max_D_y_components_vertices = max_D_y_components_vertices.union(self.
                        max_component_vertices_set[D_iy])


        alpha_value = 1 + max_alpha

        self.max_component_vertices_set[component_key] = max_D_y_components_vertices.union({
            max_y})
        if (x,max_y) in self.graph.intervals:
            self.max_component_vertices_set[component_key] = self.max_component_vertices_set[
                component_key].union(self.max_interval_vertices_set[(x,max_y)])
        print(f"self.max_component_vertices_set[{component_key}]:{self.
            max_component_vertices_set[component_key]}")

        component.alpha = alpha_value


        print(f'Computed Component,{component_key} Alpha: {self.graph.components[component_key]}
            ')

        return alpha_value

    def alpha_I(self, I):

        try:
            interval = self.graph.intervals[I]
            print(interval)
            if interval.alpha is not None:
                print(f"Interval ({I}) is already computed with alpha: {interval.alpha}")
                return interval.alpha
```

```
x = interval.x
y = interval.y
I_vertices = interval.vertices

max_alpha = 0
self.max_interval_vertices_set[I] = set()
max_s = None
max_C_s_components_vertices = set()
for s in I_vertices:
    alpha_I_xs = self.alpha_I((x, s))
    alpha_I_sy = self.alpha_I((s, y))
    C_s_components = self.compute_C_is(s, I_vertices)
    alpha_C_sum = sum(self.alpha_C(C_is) for C_is in C_s_components)
    # max_alpha = max(max_alpha, alpha_I_xs + alpha_I_sy + alpha_C_sum)
    if max_alpha <= alpha_I_xs + alpha_I_sy + alpha_C_sum:
        max_alpha = alpha_I_xs + alpha_I_sy + alpha_C_sum
        max_s = s
        max_C_s_components_vertices = set()
        for C_is in C_s_components:
            max_C_s_components_vertices = max_C_s_components_vertices.union(self.
                max_component_vertices_set[C_is])


alpha_value = 1 + max_alpha
print("interval alpha:", alpha_value)

print("max_C_s_components_vertices", max_C_s_components_vertices)
interval_x_s_vertices = self.graph.intervals[(x, max_s)].vertices if (x, max_s) in
    self.graph.intervals else set()
print("interval_x_s_vertices:", interval_x_s_vertices)
interval_s_y_vertices = self.graph.intervals[(max_s, y)].vertices if (max_s, y) in
    self.graph.intervals else set()
print("interval_s_y_vertices:", interval_s_y_vertices)
self.max_interval_vertices_set[I] = interval_x_s_vertices.union(interval_s_y_vertices
    ).union(max_C_s_components_vertices).union({max_s})
print(f"self.max_interval_vertices_set[{I}]: {self.max_interval_vertices_set[I]}")

interval.alpha = alpha_value


print(f'Computed Interval({I}) alpha:{self.graph.intervals[I]}')
```

```
            return alpha_value
        except:
            print("Interval", I, "not found")
            return 0

    def compute_components_subset(self, vertex, target_vertices, num_components):
        """
        Computes the components of G - N[vertex] contained in the target set.

        :param vertex: The vertex whose neighborhood defines the components.
        :param target_vertices: The set of vertices that the component should be a subset of.
        :param num_components: The number of components to consider.
        :return: A list of component keys whose vertices are a subset of the target set.
        """
        computed_components = []

        for i in range(num_components):
            component_key = (vertex, i)
            component = self.graph.components[component_key]
            if component.vertices.issubset(target_vertices):
                computed_components.append(component_key)

        return computed_components

    def compute_D_iy(self, y, Cx_vertices):
        """
        D_iy are the components of G - N[y] contained in Cx.
        """
        return self.compute_components_subset(y, Cx_vertices, self.graph.num_of_components[y])

    def compute_C_is(self, s, I_vertices):
        """
        C_is are the components of G - N[s] contained in I.
        """
        return self.compute_components_subset(s, I_vertices, self.graph.num_of_components[s])

    def alpha_G(self):
        """
        G is the graph
        """
```

```
        max_alpha = 0
        max_alpha_component = None
        for x in self.graph.vertices:
            alpha_C_sum = sum(self.alpha_C((x,i)) for i in range(self.graph.num_of_components[x])
                )
            if max_alpha < alpha_C_sum:
                max_alpha = alpha_C_sum
                max_alpha_component = x



        print("max_alpha_component:", max_alpha_component)
        independent_set = {max_alpha_component}
        for i in range(self.graph.num_of_components[max_alpha_component]):
            component_key = (max_alpha_component, i)
            independent_set = independent_set.union(self.max_component_vertices_set[component_key
                ])
        print("max independent set:", independent_set)
        self.graph.independent_set = independent_set
        # self.graph.show()
        return max_alpha + 1

    def run(self):
        return self.computing_independent_set()
```

Εξηγούμε αναλυτικά τις συναρτήσεις την κλάσεις:

1. Κατασκευαστής (__init__): Δέχεται ένα αντικείμενο graph και αρχικοποιεί τα χαρακτηριστικά της κλάσης, όπως τα σύνολα ”max_interval_vertices_set” και ”max_component_vertices_set”.

2. Η συνάρτηση `computing_independent_set`: Είναι η κύρια συνάρτηση που εκτελεί τα βήματα του αλγορίθμου για τον υπολογισμό του μέγιστου ανεξάρτητου συνόλου.

3. Η συνάρτηση `alpha_C`: Υλοποιεί το λήμμα 2.3.

4. Η συνάρτηση `alpha_I`: Υλοποιεί το λήμμα 2.2.

5. Η συνάρτηση `alpha_G`: Υλοποιεί το λήμμα 2.1.

6. Η συνάρτηση `compute_components_subset` λαμβάνει ως είσοδο έναν κόμβο `vertex`, ένα σύνολο κορυφών `target_vertices`, και τον αριθμό των *Components* που πρέπει να εξεταστούν `num_components`. Επιστρέφει μια λίστα με τα κλειδιά των *Components* που ανήκουν στο σύνολο `target_vertices`. Αυτή η λειτουργία είναι κρίσιμη για τον υπολογισμό των ανεξάρτητων συνόλων στη βάση των Λημμάτων 2.2 και 2.3.

7. Η συνάρτηση `compute_D_iy` χρησιμοποιεί την `compute_components_subset` για να υπολογίσει τις συνιστώσες `D_iy`, δηλαδή εκείνες που περιέχονται στο `Cx_vertices` και ανήκουν στη γειτονιά του `y`. Η είσοδος αυτής της συνάρτησης περιλαμβάνει τον κόμβο `y` και το υποσύνολο κορυφών `Cx_vertices`, ενώ η έξοδός της είναι η λίστα των *Components* `D_iy`.

8. Η συνάρτηση `compute_C_is` λειτουργεί παρόμοια με την `compute_D_iy`, αλλά χρησιμοποιείται για τον υπολογισμό των *Components* `C_is`, που περιέχονται στο `I_vertices` και ανήκουν στη γειτονιά του `s`. Η είσοδος περιλαμβάνει τον κόμβο `s` και το υποσύνολο κορυφών `I_vertices`, ενώ η έξοδός της είναι η λίστα των *Components* `C_is`.

graph.py

```
from itertools import combinations
from pyvis.network import Network
from component import Component
from interval import Interval

import os


class Graph:
    show_count = 0 # Class-level variable to keep track of show calls

    def __init__(self, num_of_vertices, num_of_edges, edges, vertices=None):
        self.num_of_vertices = num_of_vertices
        self.num_of_edges = num_of_edges
        self.vertices = set(str(i) for i in range(num_of_vertices)) if vertices is None else
            vertices
        self.edges = set((str(u), str(v)) for u, v in edges)
        self.adjacency_list = {str(vertex): set() for vertex in self.vertices}
```

```
    for edge in self.edges:
        u, v = edge
        self.adjacency_list[u].add(v)
        self.adjacency_list[v].add(u)

    self.components = {}
    self.num_of_components = {}
    self.intervals = {}

    self.independent_set = set()
    self.non_adjacent_vertices_component_pointer = {}


def compute_all_components(self):
    for vertex in self.vertices:
        vertex_components = self.compute_components_of_vertex(vertex)
        self.num_of_components[vertex] = len(vertex_components)
        for i in range(len(vertex_components)):
            self.components[(vertex, i)] = Component(vertex, i, vertex_components[i])




def compute_components_of_vertex(self,vertex):
    components_vertices = self.vertices - self.closed_neighborhood(vertex)
    components = []
    visited = set()

    def dfs(node,component):
        visited.add(node)
        component.add(node)
        self.non_adjacent_vertices_component_pointer[(vertex, node)] = (vertex, len(
            components))
        for neighbor in self.adjacency_list[node]:
            if neighbor not in visited and neighbor in components_vertices:
                dfs(neighbor, component)

    for v in components_vertices:
        if v not in visited:
            component = set()
```

```
            dfs(v, component)
            components.append(component)

    return components



def closed_neighborhood(self, vertex):
    return {vertex}.union(self.adjacency_list[vertex])

def closed_neighborhood_of_set(self, vertex_set):
    closed_neighborhood = set()
    for vertex in vertex_set:
        closed_neighborhood = closed_neighborhood.union(self.closed_neighborhood(vertex))
    return closed_neighborhood



def compute_all_intervals(self):
    for x in self.vertices:
        for y in self.vertices:
            if x != y and y not in self.adjacency_list[x]:
                interval = self.compute_interval(x, y)
                if interval:
                    self.intervals[(x, y)] = Interval(x, y, interval)
    return self.intervals


def compute_interval(self, x, y):
    if y in self.adjacency_list[x]:
        raise ValueError("Vertices x and y are adjacent. Interval can only be computed for
            nonadjacent vertices.")

    Cx_y = None
    Cy_x = None

    Cx_y_pointer = self.non_adjacent_vertices_component_pointer[(x, y)]
    Cy_x_pointer = self.non_adjacent_vertices_component_pointer[(y, x)]

    Cx_y = self.components[Cx_y_pointer].vertices
    Cy_x = self.components[Cy_x_pointer].vertices
```

```
    interval = Cx_y.intersection(Cy_x) if Cx_y is not None and Cy_x is not None else None
    return interval


def copy(self):
    return Graph(self.num_of_vertices, self.num_of_edges, self.edges, self.vertices)

def show(self, graph_name='graph'):
    Graph.show_count += 1
    print("Showing graph:", Graph.show_count)
    net = Network(height="500px", width="100%", bgcolor="#222222", font_color="white")

    for vertex in self.vertices:
        if vertex in self.independent_set:
            net.add_node(vertex, color="red")
        else:
            net.add_node(vertex)



    for edge in self.edges:
        u, v = edge
        net.add_edge(u, v, color="white")

    script_dir = os.path.dirname(os.path.abspath(__file__))
    output_dir = os.path.join(script_dir, '..', 'output-graphs')
    file_name = os.path.join(output_dir, f"{graph_name}-{Graph.show_count}.html")
    net.show(file_name)
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] A. Brandstädt, V.D. Chepoi, and F.F. Dragan. The algorithmic use of hypertree structure and maximum neighbourhood orderings. *Discrete Applied Mathematics*, 82(1-3):43–77, 1998.

[2] Kloks T. Kratsch D. Broersma, H. and H. Müller. Independent sets in asteroidal triple-free graphs. *SIAM Journal on Discrete Mathematics*, 12(2):276–287, 1999.

[3] D.G. Corneil and Y. Perl. Clustering and domination in perfect graphs. *Discrete Appl. Math.*, 9:27–39, 1984.

[4] D.G. Corneil and L. Stewart. Dominating sets in perfect graphs. *Discrete Math.*, 86:145–164, 1990.

[5] D.P. Dailey. Uniqueness of colorability and colorability of planar 4-regular graphs are np-complete. *Discrete Math.*, 30:289–293, 1980.

[6] Johnson D.S. The np-completeness column: an ongoing guide. *J. Algorithms*, 5:147–160, 1984.

[7] E.S. Elmallah and L.K. Stewart. Independence and domination in polygon graphs. *Discrete Appl. Math.*, 44:65–77, 1993.

[8] M. Farber and M. Keil. Domination in permutation graphs. *J. Algorithms*, 6:309–321, 1985.

[9] Johnson D. Stockmeyer L. Garey, M.R. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1:237–267, 1976.

[10] Johnson D.S. Garey, M.R. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[11] Chang G.J. Labelling algorithms for domination problems in sunfree chordal graphs. *Discrete Appl. Math.*, 22:21–34, 1988.

[12] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. North-Holland, Amsterdam, 2nd edn edition, 2004.

[13] Lovász L. Schrijver A. Grötschel, M. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[14] D.G. Kirkpatrick H. Breu. Algorithms for dominating and steiner set problems in cocomparability graphs. manuscript, 1993.

[15] I. Holyer. The np-completeness of edge-coloring. *SIAM J. Comput.*, 10:718–720, 1981.

[16] 3-Colouring AT-Free Graphs in Polynomial Time. Juraj stacho. *Algorithmica*, 64(3):384–399, 2012.

[17] D. Kratsch and L. Stewart. Domination on cocomparability graphs. *SIAM J. Discrete Math.*, 6:400–417, 1993.

[18] Dieter Kratsch. Domination and total domination on asteroidal triple-free graphs. *Discrete Applied Mathematics*, 99(1-3):111–123, 2000.

[19] C. G. Lekkerkerker and J. Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math*, 51(1):45–64, 1964.

[20] Farber M. Domination, independent domination, and duality in strongly chordal graphs. *Discrete Appl. Math.*, 7:115–130, 1984.

[21] Chang M.-S. Efficient algorithms for the domination problems on interval and circular-arc graphs. *SIAM Journal on Computing*, 27(6):1671–1694, 1998.

[22] Sau-I. Mertzios, G.B. and S. Zaks. The recognition of tolerance and bounded tolerance graphs. *SIAM Journal on Computing*, 40(5):1234–1257, 2011.

[23] Chang M.S. Efficient algorithms for the domination problems on interval and circular-arc graphs. *SIAM J. Comput.*, 27:1671–1694, 1998.

[24] pyvis Contributors. Interactive network visualizations — pyvis 0.1.3.1 documentation, n.d. Version 0.1.3.1.

[25] Liang-Y.D. Dhall S.K. Rhee, C. and S. Lakshmivarahan. An o(n + m)-time algorithm for finding a minimum-weight dominating set in a permutation graph. *SIAM J. Comput.*, 25:404–419, 1996.

[26] Arnborg S. Efficient algorithms for combinatorial problems on graphs with bounded decomposability — a survey. *Bit Numerical Mathematics*, 25(1):1–23, 1985.