ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσεγγιστικοί Αλγόριθμοι για το Πρόβλημα του Δάσους Steiner

Νίκος Σ. Γιαχούδης

ΕΠΙΒΛΕΠΩΝ: **Βασίλειος Ζησιμόπουλος**, Καθηγητής, ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2017

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσεγγιστικοί Αλγόριθμοι για το Πρόβλημα του Δάσους Steiner

Νίκος Σ. Γιαχούδης
Α.Μ.: Μ1356

ΕΠΙΒΛΕΠΩΝ: **Βασίλειος Ζησιμόπουλος**, Καθηγητής, ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Βασίλειος Ζησιμόπουλος**, Καθηγητής, ΕΚΠΑ
Ορέστης Τελέλης, Επίκουρος Καθηγητής, ΠΑΠΕΙ

ΙΟΥΝΙΟΣ 2017

ΠΕΡΙΛΗΨΗ

Το πρόβλημα Δάσους Steiner είναι μία γενίκευση του προβλήματος Δέντρου Steiner και κατ' επέκταση του Ελάχιστου Επικαλύπτοντος Δέντρου. Αυτά τα προβλήματα είναι ευρέως γνωστά προβλήματα συνδυαστικής βελτιστοποίησης. Επίσης, βρίσκουν πολλές εφαρμογές όπως, η σχεδίαση VLSI κυκλωμάτων, η σχεδίαση τηλεφωνικών δικτύων και η ανακατασκευή φυλογενετικών δέντρων. Επομένως, η μελέτη αυτών των προβλημάτων και η σχεδίαση αλγορίθμων, που τα επιλύουν, είναι σημαντική. Επιλύοντας τα γενικά προβλήματα επιλύουμε και όλα τα σχετικά πρακτικά προβλήματα.

Έχει αποδειχθεί από τον Karp ότι το αντίστοιχο πρόβλημα απόφασης του Δέντρου Steiner, και κατ' επέκταση του Δάσους Steiner, είναι *NP-Πλήρες*. Επομένως, το να βρούμε σε πωλυωνυμικό χρόνο τη βέλτιστη λύση είναι *NP-Δύσκολο*, το οποίο σημαίνει ότι θα πρέπει να σχεδιάσουμε προσεγγιστικούς αλγόριθμους. Για αρκετό χρονικό διάστημα οι καλύτεροι αλγόριθμοι συνδυαστικής φύσεως δεν είχαν κάποια σταθερή προσέγγιση, η μόνη 2-προσεγγιστική μέθοδος που είχαμε ήταν μια μέθοδος αρχικού - δυϊκού από την περιοχή του Γραμμικού Προγραμματισμού.

Πρόσφατα, όμως, οι Gupta και Kumar μας έδωσαν έναν 96-προσεγγιστικό αλγόριθμο συνδυαστικής φύσεως. Καλούμαστε, λοιπόν, να βελτιώσουμε την ανάλυση αυτού του αλγορίθμου καθώς και να τον συγκρίνουμε με τον Άπληστο Αλγόριθμο, ο οποίος δεν έχει σταθερή προσέγγιση. Στην παρούσα διπλωματική εργασία δείξαμε ότι ο "Αδηφάγος" Αλγόριθμος είναι 2-προσεγγιστικός στην κατηγορία των Δέντρων. Επίσης, πραγματοποιήθηκε μία υλοποίηση των δύο αλγορίθμων με σκοπό την πειραματική τους σύγκριση. Είδαμε ότι, οι δύο αλγόριθμοι βρίσκουν λύσεις πολύ κοντά στη βέλτιστη όσον αφορά τα Δέντρα, στην κατηγορία των Πλήρη γραφημάτων είδαμε ότι ο Άπληστος Αλγόριθμος έχει καλύτερη επίδοση και στην κατηγορία των Ψευδοτυχαίων γραφημάτων η επίδοση των αλγορίθμων είναι παρόμοια.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αλγοριθμική Επιχειρησιακή Έρευνα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Προσεγγιστικοί Αλγόριθμοι, Δάσος Steiner, Δέντρο Steiner, Θεωρία Γραφημάτων, Συνδυαστική Βελτιστοποίηση

ABSTRACT

The Steiner Forest Problem is a generalization of Steiner Tree and moreover the Minimum Spanning Tree problem. Those problems are widely known as combinatorial optimization problems. Furthermore, they found many applications such as the design of VLSI circuits, the design of telecommunication networks and the reconstruction of phylogenetic trees. Therefore, finding solutions to those problems means that we solve all practical problems connected to them.

Karp had shown that the decision problem of Steiner Tree is *NP-Complete*. Thus, finding an optimal solution in polynomial time for that problem, consequently for the Steiner Forest Problem, is *NP-Hard*, so we should design approximation algorithms. For a long time the best known purely combinatorial algorithms did not have constant approximation ratios, and the only constant approximation ratio method was a primal - dual method.

Recently, Gupta and Kumar have given us a 96-approximation algorithm, which is purely combinatorial. In the current dissertation, we are called to improve the analysis of this algorithm (called Gluttonous), as well as compare it to another non-constant approximation Greedy algorithm. We show that Gluttonous is a 2-approximation algorithm on Trees and implement the two algorithms so we can compare them on specific instances. In practise, the Greedy and Gluttonous algorithms are very close to an optimal solution for Tree instances, for Pseudo-random Graphs they have similar behaviour and for the case of Complete Graphs the Greedy algorithm outruns the Gluttonous.

SUBJECT AREA: Algorithmic Operations Research

KEYWORDS: Approximation Algorithms, Steiner Forest, Steiner Tree, Graph Theory, Combinatorial Optimization

ΠΕΡΙΕΧΟΜΕΝΑ

1 Το Πρόβλημα Δέντρου και Δάσους Steiner **15**

1.1 Βιβλιογραφική Επισκόπηση 16

1.2 Το Πρόβλημα Δέντρου Steiner 17

1.2.1 Πρόβλημα Ελάχιστου Επικαλύπτοντος Δέντρου 19

1.2.2 Επίλυση του προβλήματος Δέντρου Steiner 19

1.3 Πρόβλημα Δάσους Steiner 20

1.4 Κλασικός Άπληστος Αλγόριθμος (Paired Greedy Algorithm) 21

1.4.1 Παράδειγμα σε κυβικό γράφημα 22

2 Αλγόριθμοι Επίλυσης για το Πρόβλημα Δάσους Steiner **25**

2.1 Μετρικές Γραφημάτων 25

2.1.1 Παράδειγμα 26

2.1.2 Μετρικές Δέντρων 26

2.2 Ο ”Αδηφάγος” Αλγόριθμος Gluttonous 27

2.3 Ανάλυση του “Αδηφάγου” Αλγορίθμου 29

2.4 Ανάλυση του “Αδηφάγου” Αλγορίθμου σε Δέντρα 31

2.4.1 Η Περίπτωση Πιστού Δάσους 31

2.4.2 Η Περίπτωση μη Πιστού Δάσους 31

2.5 Ανάλυση του Άπληστου Αλγορίθμου σε Δέντρα 33

3 Υλοποίηση και Πειράματα **35**

3.1 Υλοποίηση 35

3.1.1 Γράφημα 35

3.1.2 Ο Άπληστος Αλγόριθμος 36

3.1.3 Ο “Αδηφάγος” Αλγόριθμος 37

3.1.4 Βέλτιστη λύση 37

3.2 Πειράματα 38

3.2.1 Σχεδόν τυχαίο γράφημα 38

3.2.2 Πλήρες γράφημα 39

3.2.3 Δέντρα 39

3.2.4 Αποτελέσματα πειραμάτων 40

4 Συμπεράσματα **43**

4.1 Ενδιαφέροντα Προβλήματα 43

ΠΑΡΑΡΤΗΜΑΤΑ **44**

I Πίνακες Αποτελεσμάτων **45**

II Συνοπτικά αποτελέσματα για μεγάλα στιγμιότυπα **49**

III Υλοποίηση σε Python **51**

ΑΝΑΦΟΡΕΣ **73**

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

1.1 Ο κύκλος Euler φαίνεται με τις διακεκομμένες ακμές. 19

1.2 Κύκλος Hamilton, οι γεμάτοι κύκλοι είναι οι κόμβοι που θέλουμε να καλύψουμε. 20

1.3 Γράφημα $G = (V,\ E)$, όπου $|V| = 8,\ |E| = 12$, girth$(G) = 4$. 22

2.1 Το γράφημα G από το οποίο θα υπολογίσουμε μια μετρική κλειστότητας που φαίνεται στο Σχήμα 2.2. 25

2.2 Η μετρική κλειστότητας M του γραφήματος G του Σχήματος 2.1. 26

2.3 Στην πρώτη σειρά φαίνεται η εκτέλεση του “Αδηφάγου” Αλγόριθμου και στη δεύτερη σειρά η εκτέλεση του Άπληστου Αλγόριθμου. 28

2.4 Ένα Gadget. 32

2.5 Δύο Gadgets συνδεδεμένα. 33

3.1 Συνοπτικά τα αποτελέσματα για τυχαία γραφήματα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$. 40

3.2 Συνοπτικά τα αποτελέσματα για πλήρη γραφήματα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$. 41

3.3 Συνοπτικά τα αποτελέσματα για δέντρα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$. 42

II.i Συνοπτικά τα αποτελέσματα του συνολικού κόστους σε πλήρη γραφήματα. 49

II.ii Συνοπτικά τα αποτελέσματα της τυπικής απόκλισης σε πλήρη γραφήματα. 49

II.iii Συνοπτικά τα αποτελέσματα του συνολικού κόστους σε τυχαία γραφήματα. 50

II.iv Συνοπτικά τα αποτελέσματα της τυπικής απόκλισης σε τυχαία γραφήματα. . 50

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

I.i Πειράματα σε τυχαία γραφήματα με $n = \{5, 10, 15\}$. 45
I.ii Πειράματα σε πλήρη γραφήματα με $n = \{5, 10, 15\}$. 46
I.iii Πειράματα σε δέντρα με $n = \{10, 20, 40\}$. 47

1. ΤΟ ΠΡΟΒΛΗΜΑ ΔΕΝΤΡΟΥ ΚΑΙ ΔΑΣΟΥΣ STEINER

Σε αυτό το κεφάλαιο θα αναφερθούμε στο πρόβλημα του Δέντρου Steiner (Steiner Tree) και ποια είναι η σχέση του με το πρόβλημα του Ελάχιστου Επικαλύπτοντος Δέντρου (Minimum (Cost) Spanning Tree), κυρίως όμως θα ασχοληθούμε με το πρόβλημα του Δάσους Steiner (Steiner Forest). Τα παραπάνω προβλήματα ανήκουν στην οικογένεια των προβλημάτων συνδυαστικής βελτιστοποίησης, πολλά από τα οποία ανήκουν στην κλάση *NP* προβλημάτων και πιο συγκεκριμένα είναι *NP-Δύσκολα*. Επομένως η εύρεση της βέλτιστης λύσης σε πολυωνυμικό χρόνο είναι αδύνατη, εκτός και αν $P = NP$. Θα πρέπει, λοιπόν, να αναπτύξουμε προσεγγιστικούς αλγόριθμους πολυωνυμικού χρόνου για να μπορέσουμε, τουλάχιστον, να προσεγγίσουμε τη βέλτιστη λύση σε πωλυωνυμικό χρόνο, έτσι ώστε οι αλγόριθμοι να έχουν πρακτικές εφαρμογές. Παρακάτω θα αναφέρουμε κάποια θεωρήματα, καθώς και ορισμούς προβλημάτων, έτσι όπως αναφέρονται στα [9, 24], για να έχουμε ένα κοινό πλαίσιο, σύμφωνα με το οποίο να μπορούμε να συζητήσουμε και να αναλύσουμε τα συγκεκριμένα προβλήματα, αλλά και τους τρόπους επίλυσής τους.

Δύο πολύ κλασικά προβλήματα στον χώρο της πληροφορικής είναι, το πρόβλημα του Συντομότερου Μονοπατιού και το πρόβλημα του Ελάχιστου Επικαλύπτοντος Δέντρου. Τα προβλήματα αυτά είναι πολυωνυμικού χρόνου σε αντιδιαστολή με τα προβλήματα που εξετάζει η παρούσα εργασία. Κάποιοι από τους πιο γνωστούς αλγορίθμους που επιλύουν αυτά τα προβλήματα είναι οι εξής. Για το πρόβλημα του Συντομότερου Μονοπατιού υπάρχει ο αλγόριθμος του Dijkstra [10], ο οποίος είναι και ο μοναδικός, πρακτικά, γνωστός βέλτιστος αλγόριθμος για το συγκεκριμένο πρόβλημα. Για το Ελάχιστο Επικαλύπτον Δέντρο υπάρχουν οι εξής δύο αλγόριθμοι, του Kruskal [18] και του Prim [22], οι οποίοι είναι και οι πιο γνωστοί. Αξίζει να αναφερθούμε στα παραπάνω προβλήματα γιατί είναι, όπως θα δούμε στη συνέχεια, ειδικές περιπτώσεις του προβλήματος Δέντρου Steiner.

Τα δύο προβλήματα που καλούμαστε να μελετήσουμε στην παρούσα εργασία είναι το πρόβλημα του Δέντρου Steiner και το πρόβλημα του Δάσους Steiner. Το πρώτο ορίζεται άτυπα ως εξής: έστω ένα βεβαρημένο, μη κατευθυνόμενο γράφημα και ένα υποσύνολο $\mathscr{D}, |\mathscr{D}| = k$ των κόμβων του γραφήματος. Πρέπει να υπολογίσουμε ένα υπογράφημα το οποίο θα περιέχει τουλάχιστον τους κόμβους του $\mathscr{D}$, υπό τον περιορισμό του ότι, το άθροισμα των βαρών αυτού του γραφήματος να είναι το ελάχιστο δυνατό. Το πρόβλημα του Δάσους Steiner ορίζεται άτυπα ως εξής: έστω ένα βεβαρημένο, μη κατευθυνόμενο γράφημα και ένα σύνολο ζευγαριών κόμβων του γραφήματος. Πρέπει να υπολογίσουμε ένα υπογράφημα με τους εξής περιορισμούς

- Κάθε ζευγάρι κόμβων θα πρέπει να ανήκει σε ένα συνεκτικό τμήμα αυτού του γραφήματος.
- Το άθροισμα των βαρών αυτού του γραφήματος να είναι το ελάχιστο δυνατό.

Τα δύο προβλήματα ορίζονται τυπικά μετά την βιβλιογραφική επισκόπηση.

1.1 Βιβλιογραφική Επισκόπηση

Το πρόβλημα Δέντρου Steiner έχει ερευνηθεί ευρέως, καθώς υπάρχουν πολλές παραλλαγές [14]. Το πρόβλημα, όπως προαναφέρθηκε, είναι *NP-Δύσκολο*, το οποίο καθιστά την επίλυσή του σε πολυωνυμικό χρόνο αδύνατη. Επομένως, οι αλγόριθμοι πολυωνυμικού χρόνου που υπάρχουν είναι προσεγγιστικοί. Ο καλύτερος λόγος προσέγγισης σύμφωνα με το [14] είναι $\ln(4+\epsilon) < 1.39$ από Byrka και λοιποί [5], όπου η μέθοδος τους είναι βασισμένη σε Γραμμικό Προγραμματισμό. Επίσης έχει αποδειχθεί ότι το πρόβλημα δεν προσεγγίζεται με λόγο προσέγγισης καλύτερο του $96/95$ [8]. Αντίθετα, αν το γράφημα εισόδου είναι επίπεδο, τότε υπάρχει ένα Προσεγγιστικό Σχήμα Πολυωνυμικού Χρόνου ή αλλίως Polynomial Time Approximation Scheme(PTAS) [4]. PTAS είναι ένας αλγόριθμος, ο οποίος πετυχαίνει ένα λόγο προσέγγισης $1 + \epsilon$, δηλαδή μπορεί να μας δώσει μια λύση που είναι ϵ κοντά στη βέλτιστη επιλέγοντας όποιο ϵ επιθυμούμε. Επίσης σύμφωνα με το [14] το πρόβλημα σε επίπεδα γραφήματα είναι επιλύσιμο σε χρόνο της τάξης του $O(3^k n + 2^k(n \log n + m))$, όπου $n = |V|$ είναι το πλήθος τον κόμβων $k = |S|$ είναι το πλήθος των steiner κόμβων και $m = |E|$ το πλήθος τον ακμών [11, 15].

Ένα πιο γενικό πρόβλημα είναι το πρόβλημα Δάσους Steiner. Επίσης έχει γίνει αρκετή έρευνα σε αυτό το πρόβλημα με πολλές παραλλαγές και πολλά αποτελέσματα [21]. Ένα από τα πρώτα, αν όχι το πρώτο, αποτέλεσμα σταθερού λόγου προσέγγισης ήταν αυτό των Agrawal, Klein και Ravi [1], στο οποίο επιλύουν το πρόβλημα με έναν $2-1/k$ προσεγγιστικό αλγόριθμο [1]. Η μέθοδος των Agrawal, Klein και Ravi απλοποιήθηκε από τους Goemans και Williamson γενικεύοντας τη μέθοδο, ουσιαστικά μια μέθοδος αρχικού-δυϊκού από την περιοχή του Γραμμικού Προγραμματισμού, με λόγο προσέγγισης 2, για το Δάσος Steiner αλλά και για άλλα συνδυαστικά προβλήματα [12].

Ένας άπληστος συνδυαστικός αλγόριθμος, που δεν βασίζεται στην μέθοδο αρχικού-δυϊκου, είναι ο *Paired Greedy Algorithm*, στον οποίο θα αναφερόμαστε ώς *Κλασικό Άπληστο Αλγόριθμο* από εδώ και στο εξής. Αυτή η άπληστη μέθοδος είναι ίσως η πιο απλή μέθοδος που μπορεί να εφαρμοστεί, όπως αναφέρεται και από τους Chen, Roughgarden και Valiant [6]. Όμως, δεν έχει σταθερή προσέγγιση, η οποία είναι της τάξης του $\Theta(\log n)$.

Ένα από τα πιο πρόσφατα αποτελέσματα, για το οποίο γίνεται μια ανάλυση στην παρούσα εργασία, είναι των Gupta και Kumar [13], όπου αποδεικνύουν σταθερή προσέγγιση για τον αλγόριθμο *Gluttonous*, στον οποίο θα αναφερόμαστε ως *Αδηφάγο* από εδώ και στο εξής. Για αρκετό καιρό παρέμενε ανοιχτό ερώτημα, αν κάποιος συνδυαστικός αλγόριθμος, που δεν βασίζεται στη μέθοδο αρχικού-δυϊκού, είχε σταθερή προσέγγιση. Με το αποτέλεσμα των Gupta και Kumar, ότι ο Αδηφάγος είναι ένας 96 προσεγγιστικός αλγόριθμος, [13], κλείνουν το παραπάνω ερώτημα.

Όπως μπορούμε να παρατηρήσουμε τα αποτελέσματα χρονολογικά και συνοπτικά είναι τα εξής:

- Agrawal και λοιποί παρουσιάζουν έναν $(2 - 1/k)$-προσεγγιστικό αλγόριθμο, όπου $k = |\mathscr{D}|$ [1]
- Goemans και Williamson γενικεύουν την παραπάνω μέθοδο και παρουσιάζουν έναν

2-προσεγγιστικό αλγόριθμο [12]

- Gupta και Kumar παρουσιάζουν έναν 96-προσεγγιστικό αλγόριθμο [13]

Το αξιοσημείωτο σε αυτή τη χρονολογική σειρά είναι ότι τα πιο πρόσφατα αποτελέσματα μας παρουσιάζουν αλγορίθμους με χειρότερο λόγο προσέγγισης από ότι τα παλαιότερα. Αυτό συμβαίνει γιατί στην προσπάθεια να σχεδιαστούν απλούστεροι και γενικότεροι αλγόριθμοι η ανάλυση δυσκολεύει, άρα είναι δυσκολότερο να αποδείξουμε μικρό λόγο προσέγγισης. Θα μπορούσαμε να πούμε λοιπόν ότι υπάρχει μία συσχέτιση μεταξύ της απλοϊκότητας ενός αλγορίθμου, η οποία είναι αντιστρόφως ανάλογη της ανάλυσης του.

1.2 Το Πρόβλημα Δέντρου Steiner

Το πρόβλημα του Δέντρου Steiner, όπως αναφέρεται στο [24] (Κεφάλαιο 3), πρωτοεμφανίστηκε σε ένα γράμμα που έστειλε ο Gauss στον Schumacher. Το συγκεκριμένο πρόβλημα είναι ευρέως γνωστό σήμερα και είναι ένα από τα κεντρικά προβλήματα στο πεδίο των προσεγγιστικών αλγορίθμων. Φυσικά το πρόβλημα βρίσκει εφαρμογές σε πολλά προβλήματα του κόσμου μας όπως στα τηλεφωνικά δίκτυα, η σχεδίαση VLSI αλλά ακόμα και η ανακατασκευή Φυλογενετικών Δέντρων στην περιοχή της Υπολογιστικής Βιολογίας.

Αναλυτικότερα, στη σχεδίαση VLSI κυκλομάτων, έτσι όπως αναφέρεται στο [7], μπορούμε να θεωρήσουμε ως κόμβους ενός γραφήματος, τους ακροδέκτες κάποιου ολοκληρωμένου κυκλώματος. Με αυτόν τον τρόπο όταν έχουμε ένα σύνολο από ακροδέκτες που είναι απαραίτητο να λάβουν το ίδιο ηλεκτρικό σήμα, τότε προκύπτει η εξής παραλλαγή του προβλήματος Δέντρου Steiner. Δεδομένου γραφήματος και κάποιων συνόλων από κόμβους, να υπολογίσουμε ένα ελάχιστο συνεκτικό υπογράφημα τέτοιο ώστε τουλάχιστον ένας κόμβος από κάθε σύνολο να ανήκει σε αυτό [16].

Όσον αφορά τα τηλεφωνικά δίκτυα, σύμφωνα με το [7], φανταστείτε ότι έχουμε ένα τηλεφωνικό δίκτυο όπου είναι συνδεδεμένοι αρκετοί χρήστες και το δίκτυο είναι βέλτιστο ως προς τις αποστάσεις. Όταν λοιπόν ένας καινούριος χρήστης θέλει να συνδεθεί και αυτός στο δίκτυο θα πρέπει να τον εισάγουμε διατηρώντας τη βέλτιστη κατασκευή του. Αυτή η διαφοροποίηση του Δέντρου Steiner ανήκει στην οικογένεια προβλημάτων που είναι γνωστά ως προβλήματα πραγματικού χρόνου (on-line problems) [20, 23].

Μια άλλη εκδοχή του προβλήματος προέρχεται από τον κόσμο της Υπολογιστικής Βιολογίας και πιο συγκεκριμένα από την ανακατασκευή Φυλογενετικών Δέντρων. Όπως αναφέρεται στο [19], για τους Βιολόγους τα φύλα του δέντρου αντιπροσωπεύουν τα είδη που υπάρχουν, ενώ οι εσωτερικοί κόμβοι είναι οι πρόγονοι που έχουν εξαφανιστεί πλέον και το μήκος των ακμών αντιπροσωπεύει τον χρόνο που χρειάστηκε η διαδικασία της εξέλιξης από το ένα είδος στο άλλο. Έτσι λοιπόν αν έχουμε γνώση των χρόνων εξέλιξης μεταξύ των ειδών, σκοπός είναι να κατασκευάσουμε ένα δέντρο ελάχιστων αποστάσεων το οποίο, θα έχει ως φύλα τα υπάρχοντα είδη. Οι ελάχιστες αποστάσεις προκύπτουν σύμφωνα με την αρχή που λέει ότι η φύση θα βρει πάντα το συντομότερο δρόμο ως προς την εξέλιξη.

Άρα το πρόβλημα που προκύπτει είναι ότι οι κόμβοι που θέλουμε τελικά να ανήκουν στη λύση του Δέντρου Steiner, πρέπει να είναι φύλα του δέντρου, το οποίο ονομάζεται Πλήρες

Δέντρο Steiner (Full Steiner Tree). Παραπέμπουμε τον αναγνώστη στο [19] για περισσότερες πληροφορίες σχετικές με αυτό το πρόβλημα.

Επομένως, όπως είδαμε υπάρχουν διάφορες εφαρμογές για το συγκεκριμένο πρόβλημα το οποίο υποδεικνύει την ανάγκη για μελέτη αυτού του προβλήματος και σχετικών προβλημάτων. Όπως θα δούμε αργότερα, το πρόβλημα του Δάσους Steiner είναι μια γενίκευση του προβλήματος Δέντρου Steiner.

Παραθέτουμε το παρακάτω πρόβλημα όπως αναφέρεται στο [24]:

Πρόβλημα 1. *[24] (Κεφάλαιο 3, Πρόβλημα 3.1)*

Δεδομένου γραφήματος $G = (V, E, \text{cost})$ *όπου* $\text{cost} : E \to \mathbb{R}^+$ *και δύο σύνολα* ***απαραίτητων*** *κορυφών* $\mathscr{D}$ *και* ***Steiner*** *κορυφών* $\mathscr{S}$*, όπου* $\mathscr{D} \cup \mathscr{S} = V$ *και* $\mathscr{D} \cap \mathscr{S} = \emptyset$ *βρείτε ένα* δέντρο ελάχιστου κόστους *στο* G *το οποίο περιέχει όλες τις κορυφές στο* $\mathscr{D}$ *και οποιοδήποτε υποσύνολο του* $\mathscr{S}$.

Η τριγωνική ανισότητα είναι σημαντική για το πρόβλημα, καθώς μας διευκολύνει στην ανάλυσή του. Έστω ένα πλήρες γράφημα χωρίς κατευθύνσεις G, όπου για οποιεσδήποτε τρεις κορυφές u, v, w ισχύει ότι $\text{cost}(u, v) \leq \text{cost}(u, w) + \text{cost}(w, v)$. Το πρόβλημα υπό τον παραπάνω περιορισμό ονομάζεται πρόβλημα *μετρικού Δέντρου Steiner*.

Θεώρημα 1. *[24] (Κεφάλαιο 3, Θεώρημα 3.2)*

Υπάρχει μια αναγωγή από το πρόβλημα Δέντρου Steiner στο πρόβλημα μετρικού Δέντρου Steiner, η οποία διατηρεί τον παράγοντα προσέγγισης.

Το παραπάνω θεώρημα αποδεικνύεται στο Κεφάλαιο 3 του [24], εδώ παραθέτουμε μια γενική περιγραφή της απόδειξης για λόγους πληρότητας. Η βασική ιδέα είναι η μετατροπή ενός στιγμιότυπου Δέντρου Steiner στο *μετρικό* Δέντρο Steiner.

Έστω, λοιπόν, ένα στιγμιότυπο $\mathcal{I}$ του προβλήματος Δέντρου Steiner, τα συστατικά του μέρη είναι, ένα γράφημα $G = (V, E, \text{cost})$ και τα δύο σύνολα $\mathscr{S}$ και $\mathscr{D}$, όπου τα $\mathscr{S}$ και $\mathscr{D}$ είναι το σύνολο κόμβων Steiner και κορυφών που πρέπει να ανήκουν στη λύση, αντίστοιχα.

Η μετατροπή του $\mathcal{I}$ σε $\mathcal{I}'$ γίνεται ως εξής: κατασκευάζουμε ένα πλήρες μη κατευθυνόμενο βεβαρημένο γράφημα G' στο V και θέτουμε σαν βάρος κάθε ακμής (u, v) το κόστος του συντομότερου μονοπατιού $\text{sp}(u, v)$ στο G, όπου με $\text{sp}(u, v)$ συμβολίζουμε το σύνολο ακμών του συντομότερου μονοπατιού μεταξύ των κόμβων u, v. Οι υπόλοιπες παράμετροι παραμένουν ίδιες. Σύμφωνα με αυτήν την κατασκευή, παρατηρούμε ότι το κόστος της βέλτιστης λύσης στο $\mathcal{I}'$ είναι το πολύ όσο και το κόστος της βέλτιστης λύσης στο $\mathcal{I}$.

Τέλος για τη μετατροπή της λύσης T' από το $\mathcal{I}'$ στο $\mathcal{I}$, έχουμε τα εξής: αντικαθιστούμε κάθε ακμή (u, v) στο T' με το αντίστοιχο μονοπάτι στο G. Εν γένη μπορεί τελικά να έχουμε κύκλους στο αποτέλεσμα, άρα αφαιρούμε τους κύκλους και καταλήγουμε στο T που θα είναι λύση για το $\mathcal{I}$.

Με αυτόν τον τρόπο οποιαδήποτε προσέγγιση στο πρόβλημα του μετρικού Steiner δέντρου μπορεί να μεταφερθεί σε μια προσέγγιση στο γενικό πρόβλημα Δέντρου Steiner. Επίσης στην Ενότητα 1.2.2 θα δούμε έναν τρόπο προσέγγισης του προβλήματος που βασίζεται στην εύρεση ενός ελάχιστου δέντρου επικάλυψης. Η ιδέα είναι ότι αν υπολογίσουμε ένα δέντρο επικάλυψης πάνω στο σύνολο $\mathscr{D}$ θα έχουμε μια εφικτή λύση.

1.2.1 Πρόβλημα Ελάχιστου Επικαλύπτοντος Δέντρου

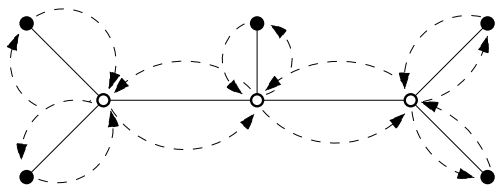
Το πρόβλημα του Ελάχιστου Επικαλύπτοντος Δέντρου είναι ειδική περίπτωση του προβλήματος Δέντρου Steiner (όπου όλοι οι κόμβοι του γραφήματος είναι τερματικοί κόμβοι που πρέπει να συνδεθούν). Η ιδέα αυτού του προβλήματος, όπως αναφέρεται και στο [9](Κεφάλαιο 23), έρχεται επίσης από το πεδίο των ηλεκτρονικών κυκλωμάτων όταν προσπαθούμε να ενώσουμε πολλά άκρα των δομικών στοιχείων ενός κυκλώματος και θέλουμε να χρησιμοποιήσουμε όσο το δυνατόν λιγότερα αλλά και κοντύτερα καλώδια. Το ότι το αποτέλεσμα είναι δέντρο βοηθάει επίσης στο ότι καλώδια δεν θα διασταυρώνονται. Ένα επίπεδο γράφημα θα ήταν αρκετό αλλά το δέντρο έχει λιγότερες ακμές από ένα γενικό επίπεδο γράφημα και άρα έχει χαμηλότερο κόστος.

Πρόβλημα 2. *Έστω ένα μη κατευθυνόμενο γράφημα $G = (V, E, c)$ όπου $c : E \to \mathbb{R}^+$ είναι μια συνάρτηση κόστους των ακμών. Θέλουμε να βρούμε ένα υποσύνολο $T \subseteq E$ έτσι ώστε το $G' = (V, T, c)$ να μην περιέχει κύκλους, να είναι συνεκτικό και το συνολικό κόστος* $\text{cost}(T) = \sum_{e \in T} c(e)$ *να είναι το ελάχιστο δυνατό.*

Παρατηρούμε ότι το γράφημα G' επειδή δεν έχει κύκλους πρέπει να είναι δέντρο. Επίσης αν στο πρόβλημα του Steiner δέντρου επιλέξουμε σαν σύνολο $\mathscr{D} = V$ τότε θα έχουμε ακριβώς το πρόβλημα του ελαχίστου δέντρου επικάλυψης (MST).

1.2.2 Επίλυση του προβλήματος Δέντρου Steiner

Ο αλγόριθμος είναι αρκετά απλός, αρκεί να ακολουθήσουμε κάποιους από τους αλγορίθμους για την επίλυση του ελαχίστου δέντρου επικάλυψης, όπως τον αλγόριθμο Kruskal [18] ή τον αλγόριθμο Prim [22], στο σύνολο $\mathscr{D}$ και έτσι θα έχουμε μια εφικτή λύση για το πρόβλημα του Steiner Δέντρου, έτσι αρκεί να βρούμε τι λόγο προσέγγισης έχουμε για αυτή τη λύση. Παραθέτουμε παρακάτω το αντίστοιχο θεώρημα έτσι όπως παρουσιάζεται στο [24] Κεφάλαιο 3, Θεώρημα 3.3.



Σχήμα 1.1: Ο κύκλος Euler φαίνεται με τις διακεκομμένες ακμές.

Θεώρημα 2. *[24] (Κεφάλαιο 3, Θεώρημα 3.3)*

Το κόστος ενός ελαχίστου δέντρου επικάλυψης στο $\mathscr{D}$ είναι το πολύ δύο φορές το κόστος της βέλτιστης λύσης.

Η απόδειξη παρουσιάζεται στο [24]. Ουσιαστικά υπολογίζουμε έναν κύκλο Euler, για παράδειγμα όπως αυτός που φαίνεται στο Σχήμα 1.1, και έτσι θα έχουμε $2 \cdot OPT$. Με έναν

κύκλο Hamilton, για παράδειγμα όπως φαίνεται στο Σχήμα 1.2, μπορούμε να αφαιρέσουμε κάποιες ακμές από τον κύκλο Euler και τέλος αφαιρούμε και μια ακόμα ακμή για να πάρουμε την τελική λύση χωρίς κύκλους, η οποία μπορούμε να δούμε ότι θα είναι το πολύ $2 \cdot OPT$.

1.3 Πρόβλημα Δάσους Steiner

Το πρόβλημα Δάσους Steiner είναι μια γενίκευση του Προβλήματος 1. Δηλαδή αντί η λύση να αποτελείται από ένα μοναδικό δέντρο, μπορεί στη λύση να υπάρχουν περισσότερα του ενός δέντρα. Πιο επίσημα σύμφωνα με τον ορισμό του [24].

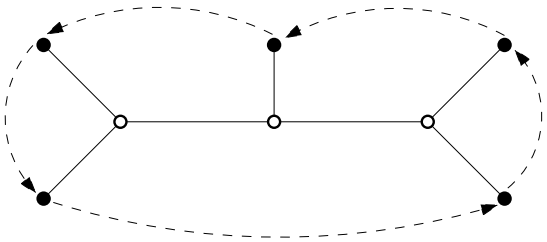
Πρόβλημα 3. *[24](Κεφάλαιο 22, Πρόβλημα 22.1)*

Δεδομένου γραφήματος $G = (V, E, w)$, όπου $w : E \to \mathbb{R}^+$ είναι μια συνάρτηση βάρους και μια συλλογή από ανεξάρτητα υποσύνολα $S_1, S_2, \cdots, S_k$ του V, βρείτε ένα υπογράφημα ελάχιστου κόστους F του G έτσι ώστε κάθε ζευγάρι κόμβων σε ένα σύνολο S_i να ανήκει σε ένα συνεκτικό τμήμα του F, για κάθε i.

Ένας άλλος ορισμός του προβλήματος σύμφωνα με το [13] είναι ο εξής:

Πρόβλημα 4. *Δεδομένου γραφήματος $G = (V, E, w)$, όπου $w : E \to \mathbb{R}^+$ είναι μια συνάρτηση βάρους και ένα σύνολο cD ζευγαριών κορυφών (s, t), όπου $s, t \in V$, βρείτε ένα υπογράφημα ελάχιστου κόστους F του G έτσι ώστε κάθε ζευγάρι κόμβων $(s, t) \in \mathscr{D}$ να ανήκει σε ένα συνεκτικό τμήμα του F.*

Βέβαια, στην παρούσα εργασία το κύριο πρόβλημα συμπεριλαμβάνει και τον περιορισμό της μετρικής όπως είδαμε στο πρόβλημα μετρικού Δέντρου Steiner. Όπως και στο πρόβλημα μετρικού Δέντρου Steiner αρκεί να ισχύει ότι το γράφημα G είναι ένα πλήρες μη κατευθυνόμενο γράφημα όπου για οποιεσδήποτε τρεις κορυφές u, v, r ισχύει η τριγωνική ανισότητα, δηλαδή ότι $w(u, v) \leq w(u, r) + w(r, v)$.



Σχήμα 1.2: Κύκλος Hamilton, οι γεμάτοι κύκλοι είναι οι κόμβοι που θέλουμε να καλύψουμε.

Η απόδειξη του παρακάτω θεωρήματος δίνεται σαν άσκηση στο [24].

Θεώρημα 3. *[24] (Κεφάλαιο 22, Άσκηση 22.2)*

Υπάρχει μια αναγωγή από το πρόβλημα Δάσους Steiner στο πρόβλημα μετρικού Δάσους Steiner, η οποία διατηρεί τον παράγοντα προσέγγισης.

Η απόδειξη είναι όμοια με την απόδειξη του Θεωρήματος 1. Θα ξεκινήσουμε μετατρέποντας ένα στιγμιότυπο I του προβλήματος Δάσους Steiner, που περιέχει το $G = (V, E, w)$ και το σύνολο $\mathscr{D}$, σε ένα στιγμιότυπο I' του προβλήματος μετρικού Δάσους Steiner. Όπως στην απόδειξη του Θεωρήματος 1 κατασκευάζουμε με τον ίδιο τρόπο το πλήρες γράφημα $G' = (V, E', w')$ πάνω στο σύνολο V και ορίζουμε μία καινούρια συνάρτηση βάρους $w' : E' \to \mathbb{R}^+$ ως εξής:

$$w'(u, v) = \sum_{e \in sp(u,v)} w(e)$$

όπου $sp(u, v)$ συμβολίζουμε το συντομότερο μονοπάτι μεταξύ των u και v στο G. Το σύνολο $\mathscr{D}$ παραμένει ίδιο. Παρατηρούμε ότι για κάθε ακμή (u, v) που υπάρχει και στα δύο γραφήματα G και G' ισχύει ότι το βάρος της στο G' θα είναι το πολύ όσο και το βάρος της στο G. Άρα το συνολικό κόστος της βέλτιστης λύσης στο I' θα είναι το πολύ όσο το συνολικό κόστος της βέλτιστης λύσης στο I.

Για να μετατρέψουμε μια λύση F' του I' σε λύση F του I, αρκεί να αντικαταστήσουμε για κάθε ακμή (u, v) του F' το αντίστοιχο μονοπάτι μεταξύ των κόμβων στο G. Παρατηρούμε ότι το αποτέλεσμα θα είναι μια εφικτή λύση του I που μπορεί να περιέχει κύκλους, άρα αφαιρούμε απλά κάποιες ακμές για να πάρουμε την τελική λύση F για το I.

Η πιο κλασική μέθοδος επίλυσης του προβλήματος Δάσους Steiner είναι με ένα σχήμα αρχικού-δυϊκού της θεωρίας Γραμμικού Προγραμματισμού. Ουσιαστικά έχουμε μια μοντελοποίηση του προβλήματος σε γραμμικό πρόγραμμα και με τη βοήθεια του δυϊκού γραμμικού προγράμματος μπορούμε και βρίσκουμε μια λύση. Αποτελέσματα για αυτήν την τεχνική μπορούν να βρεθούν στο [1].

Στο επόμενο κεφάλαιο θα δούμε μερικούς συνδυαστικούς τρόπους επίλυσης του προβλήματος Δάσους Steiner με άπληστες τεχνικές. Μία πολύ καλή και συμπιεσμένη παρουσίαση όλων των προβλημάτων που έχουν να κάνουν με το Δέντρο/Δάσος Steiner αλλά και διάφορα άλλα προβλήματα με γραφήματα βρίσκεται στο [14].

1.4 Κλασικός Άπληστος Αλγόριθμος (Paired Greedy Algorithm)

Ο αλγόριθμος είναι απλός στην περιγραφή του. Έστω ένα σύνολο ζευγαριών $(s_i, t_i) \in \mathscr{D},\ s_i,\ t_i \in V$ και ένα γράφημα $G = (V, E)$. Για κάθε ζευγάρι $(s, t) \in \mathscr{D}$ υπολογίζουμε το συντομότερο μονοπάτι μεταξύ των s και t και εισάγουμε τις ακμές του μονοπατιού στη λύση $\mathscr{F}$, μηδενίζοντας το βάρος των ακμών του ελάχιστου μονοπατιού.

Υπάρχουν άπληστοι αλγόριθμοι τόσο για το πρόβλημα Δέντρου Steiner όσο και για το πρόβλημα Δάσους Steiner. Οι Imase και Waxman [17] παρουσιάζουν έναν αλγόριθμο για το Δέντρο Steiner, ο οποίος αποδεικνύεται ως ένας $O(\log k)$-προσεγγιστικός αλγόριθμος. Για το πρόβλημα Δάσους Steiner οι Awerbuch και λοιποί [2] παρουσιάζουν έναν $O(\log^2 k)$-προσεγγιστικό αλγόριθμο, για τον οποίο είναι ανοιχτό ερώτημα εάν τελικά μπορούμε να βελτιώσουμε το λόγο προσέγγισης του σε $O(\log k)$.

Ο άπληστος αλγόριθμος για το Δάσος Steiner που θα μελετήσουμε παρακάτω αναφέρεται και από τους Chen και λοιπούς [6]. Ο λόγος προσέγγισης του συγκεκριμένου αλγόριθμου

έχει κάτω όριο $\Omega(\log n)$. Μια περιγραφή της απόδειξης για το κάτω όριο είναι εξής: Έστω ένα γράφημα $G = (V, E)$ το οποίο έχει τις εξής ιδιότητες

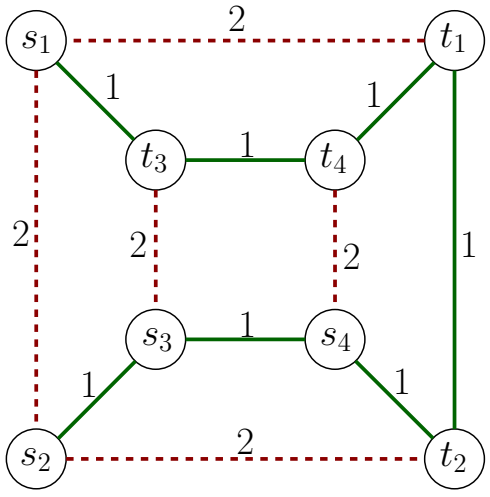
- είναι κυβικό γράφημα, δηλαδή ο βαθμός κάθε κόμβου είναι ακριβώς 3
- έχει περιορισμένη περιφέρεια, δηλαδή $\text{girth}(G) \geq c \log n$, όπου c είναι μια σταθερά και $n = |V|$.

ως περιφέρεια (girth) ορίζεται το μήκος του μικρότερου κύκλου που περιέχεται στο γράφημα. Για την κατασκευή τέτοιων γραφημάτων μπορεί κάποιος να αναφερθεί στον Biggs [3]. Υπολογίζουμε ένα επικαλύπτον δέντρο και το κρατάμε σταθερό, έχουμε $E' = E \setminus E(T)$. Από το γράφημα $G' = (V, E')$ υπολογίζουμε ένα μεγιστικό ταίριασμα (maximal matching) το οποίο θα έχει μέγεθος $\Omega(|E'|) = \Omega(n)$. Τα βάρη στις ακμές $E(T)$ του δέντρου θα είναι μονάδα και το βάρος των ακμών E' θα είναι $\frac{c}{2} \log n$. Το ταίριασμα καθορίζει το σύνολο $\mathscr{D}$ που περιέχει τις απαιτήσεις.

Ο αλγόριθμος θα επιλέξει κάθε ακμή που ενώνει το αντίστοιχο ζεύγος και άρα ανάλογα με το μέγεθος του ταιριάσματος πετυχαίνεται κόστος $\Omega(n \frac{c}{2} \log n) = \Omega(n \log n)$. Προφανώς, σαν βέλτιστη λύση θα επιλέγαμε τις ακμές του δέντρου που θα μας δώσει κόστος $OPT = n - 1$. Άρα τελικά ο λόγος προσέγγισης θα είναι

$$\Omega(\frac{n \log n}{n-1}) = \Omega(\log n).$$

1.4.1 Παράδειγμα σε κυβικό γράφημα



Σχήμα 1.3: Γράφημα $G = (V,\ E)$, **όπου** $|V| = 8,\ |E| = 12, \text{girth}(G) = 4$.

Θα παρουσιάσουμε ένα συγκεκριμένο παράδειγμα σε κυβικό γράφημα με περιφέρεια 4, για να δούμε και πως δουλεύει ο συγκεκριμένος αλγόριθμος πάνω σε μια τέτοια κατασκευή, αλλά και για να έχουμε μια γενική αίσθηση για τη λειτουργία του.

Έστω το γράφημα που φαίνεται στο Σχήμα 1.3, όπως μπορούμε να παρατηρήσουμε κάθε κόμβος του γραφήματος έχει βαθμό 3, δηλαδή έχει τρεις προσπίπτουσες ακμές. Η περιφέρεια του γραφήματος είναι $\text{girth}(G) = 4$, γιατί, ενώ δεν υπάρχει κύκλος μήκους 3, υπάρχει κύκλος μήκους 4. Επομένως, ο μικρότερος κύκλος που υπάρχει στο G είναι μήκους 4 το οποίο είναι ο ορισμός της περιφέρειας. Από τον περιορισμό $\text{girth}(G) \geq c \log n$ για την περιφέρεια μπορούμε να υπολογίσουμε ότι:

$$\frac{\text{girth}(G)}{\log n} \geq c$$

επειδή το c θα πάρει μέρος στο αποτέλεσμα του άπληστου αλγόριθμου θέλουμε να έχει όσο το δυνατόν μεγαλύτερη τιμή, άρα

$$c = \frac{4}{\log 8} = \frac{4}{3}$$

Με πράσινο χρώμα συμβολίζουμε τις ακμές του επικαλύπτοντος δέντρου T και με κόκκινο διακεκομμένο συμβολίζουμε τις ακμές $E' = E \setminus E(T)$. Οι ακμές που ανήκουν στο T έχουν βάρος 1 και οι ακμές που ανήκουν στο E' έχουν βάρος $\frac{c}{2} \log n = \frac{4}{6} \log 8 = 2$. Από το γράφημα $G' = (V,\ E')$ βρίσκουμε το τέλειο ταίριασμα (perfect matching) το οποίο θα έχει μέγεθος $|M| = 4$. Ο άπληστος αλγόριθμος θα επιλέξει τις ακμές που ενώνουν κάθε ζευγάρι αντίστοιχα γιατί αυτή η ακμή θα είναι και το ελάχιστο μονοπάτι μεταξύ του κάθε ζευγαριού, ενώ η βέλτιστη λύση είναι να επιλέξουμε όλες τις ακμές του T. Επομένως θα έχουμε $SOL_{GRD} = |M| \times \frac{c}{2} \log n = 8$ και $SOL_{OPT} = n - 1 = 7$.

2. ΑΛΓΟΡΙΘΜΟΙ ΕΠΙΛΥΣΗΣ ΓΙΑ ΤΟ ΠΡΟΒΛΗΜΑ ΔΑΣΟΥΣ STEINER

Σε αυτό το κεφάλαιο θα παρουσιάσουμε μια γενική ανάλυση των δύο βασικών συνδυαστικών αλγορίθμων, για την επίλυση του Δάσους Steiner, που ασχολείται η παρούσα εργασία. Στο πρώτο τμήμα αναφέρουμε μερικά πράγματα για μετρικές γραφημάτων, αλλά και για τον αδηφάγο αλγόριθμο ως προς τη λειτουργία του. Στο δεύτερο μέρος παρουσιάζουμε την ανάλυση, έτσι όπως αυτή διατυπώνεται στο [13]. Η ανάλυση του [13] βρίσκεται στην Ενότητα 2.3. Στη συνέχεια παρουσιάζουμε τη δική μας ανάλυση του αδηφάγου, αλλά και του άπληστου αλγορίθμου, στα δέντρα, Ενότητα 2.4.

2.1 Μετρικές Γραφημάτων

Η μετρική κλειστότητας συντομότερων μονοπατιών (shortest path closure) είναι σημαντική για το πρόβλημα, γιατί μπορούμε να μετατρέψουμε ένα αυθαίρετο γράφημα σε ένα μετρικό γράφημα, στο οποίο θα ισχύει η τριγωνική ανισότητα. Επομένως, όπως είδαμε και προηγουμένως, διευκολύνεται με αυτόν τον τρόπο η ανάλυση. Θα αναφέρουμε κάποιες γενικές πληροφορίες αλλά κυρίως θα επικεντρωθούμε σε μετρικές κλειστότητας συντομότερων μονοπατιών που προκύπτουν από δέντρα.

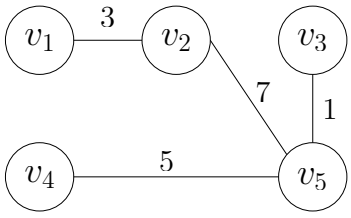
Έστω ένα γράφημα $G = (V, E, w)$ και $w : E \to \mathbb{R}$ μια συνάρτηση βάρους για κάθε ακμή του G.

Ορισμός 1. *Μετρική κλειστότητας ενός γραφήματος $G = (V, E, w)$ ονομάζουμε το γράφημα $M = (V, E', d)$ όπου $d : E' \to \mathbb{R}$ είναι μια συνάρτηση βάρους, που συνιστά μετρική και ο M είναι κλίκα.*

Η κατασκευή της μετρικής κλειστότητας πετυχαίνεται απλά αν κατασκευάσουμε ένα πλήρες γράφημα στο V όπου το βάρος κάθε ακμής u, v θα είναι το κόστος του ελάχιστου μονοπατιού μεταξύ των u και v στο G. Δηλαδή:

$$d(u, v) = \sum_{e \in \mathsf{sp}(u,v)} w(e)$$

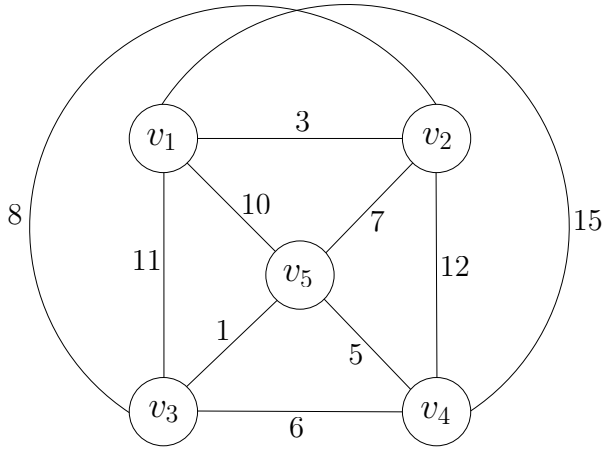
όπου $\mathsf{sp}(u, v)$ είναι το συντομότερο μονοπάτι μεταξύ των δύο κόμβων.



Σχήμα 2.1: Το γράφημα G από το οποίο θα υπολογίσουμε μια μετρική κλειστότητας που φαίνεται στο Σχήμα 2.2.

2.1.1 Παράδειγμα

Θεωρούμε το γράφημα του Σχήματος 2.1 και τη μετρική κλειστότητας αυτού του γραφήματος στο Σχήμα 2.2. Όπως μπορούμε να δούμε η μετρική κλειστότητας M είναι ένα γράφημα K_5, επίσης οι ακμές που υπήρχαν ήδη στο G υπάρχουν και στην M. Επομένως, σε ένα στιγμιότυπο του προβλήματος Δάσους Steiner η βέλτιστη λύση θα είναι ίδια και στα δύο γραφήματα. Επίσης, παρατηρούμε ότι οι ακμές του M που υπάρχουν και στο G έχουν λιγότερο βάρος από τις ακμές του M που αντιστοιχούν σε μονοπάτια του G.



Σχήμα 2.2: Η μετρική κλειστότητας M του γραφήματος G του Σχήματος 2.1.

2.1.2 Μετρικές Δέντρων

Πρέπει να αναφέρουμε κάποιες παρατηρήσεις που ισχύουν σε μετρικές που έχουν παραχθεί από γραφήματα που ανήκουν στην οικογένεια των δέντρων.

Λήμμα 1. *Κάθε ακμή (u, v) στη μετρική (κλίκα) είναι ένα μοναδικό μονοπάτι στο δέντρο.*

Απόδειξη. Κάθε ακμή (u, v) τις μετρικής υπολογίζεται βρίσκοντας το ελάχιστο μονοπάτι μεταξύ των κόμβων u και v. Το ελάχιστο μονοπάτι μεταξύ των u και v στο δέντρο είναι μοναδικό. Αν υπήρχαν παραπάνω από δύο μονοπάτια για να πάμε από τον u στον v τότε το γράφημα δεν θα ήταν δέντρο. Επομένως κάθε ακμή τις μετρικής θα αντιστοιχεί σε ένα μοναδικό μονοπάτι στο δέντρο. □

Λήμμα 2. *Η ακμές του δέντρου που συνδέουν τερματικά, αντιστοιχούν σε ακμές της μετρικής και το βάρος τους θα είναι μικρότερο από τις ακμές που αντιστοιχούν σε μονοπάτια του δέντρου.*

Απόδειξη. Σύμφωνα με τον ορισμό μια ακμή της μετρικής που συνδέει δύο τερματικά αντιστοιχεί στο συντομότερο μονοπάτι των τερματικών στο δέντρο. Αν δύο τερματικά ενώνονται με ακμή στο δέντρο τότε η ακμή αυτή είναι το συντομότερο μονοπάτι και θα είναι η αντίστοιχη ακμή που συνδέει τα δύο τερματικά στη μετρική.

Έστω u και v δύο τερματικά και έστω $e = (u, v)$ και $e' = (u, v)$ οι ακμές στο δέντρο T και στη μετρική M αντίστοιχα. Αν υποθέσουμε ότι $d(e') < w(e)$ τότε υπάρχει συντομότερο μονοπάτι μεταξύ των u, v στο δέντρο από την ακμή που τους ενώνει, άτοπο. Επίσης αν υποθέσουμε ότι $d(e') > w(e)$ τότε δεν έχουμε κατασκευάσει σωστά τη μετρική καθότι το μονοπάτι που αντιστοιχεί η ακμή e' δεν είναι το συντομότερο. Επομένως πρέπει $d(e') = w(e)$. □

2.2 Ο "Αδηφάγος" Αλγόριθμος Gluttonous

Όπως είδαμε και στο προηγούμενο κεφάλαιο μας δίνουν ένα γράφημα $G = (V, E, w)$, όπου $w : E \to \mathbb{R}^+$ είναι η συνάρτηση βάρους. Να θυμίσουμε ότι θέλουμε να ισχύει η τριγωνική ανισότητα και για να το κάνουμε αυτό υπολογίζουμε ένα καινούριο γράφημα $M = (V, E', d)$ όπως είδαμε προηγουμένως, δηλαδή την κλειστή μετρική του G. Επίσης έχουμε ένα σύνολο απαιτήσεων $\mathscr{D} \subset \binom{V}{2}$, δηλαδή ένα σύνολο ζευγαριών s_i, t_i. Άρα με είσοδο $(M, \mathscr{D})$ ο αδηφάγος αλγόριθμος μας επιστρέφει ένα δάσος $\mathscr{F} = \{T_1, \cdots, T_k\}$. Επίσης το κόστος ενός δέντρου $T = (V_T, E_T)$ είναι cost$(T) = \sum_{e \in E_T} d(e)$ και το κόστος όλου του δάσους είναι cost$(\mathscr{F}) = \sum_{T \in \mathscr{F}}$ cost(T).

Αλγόριθμος 1 "Αδηφάγος" (Gluttonous).

1: $\mathscr{C} \leftarrow$ τετριμμένη ομαδοποίηση
2: $E_{sol} \leftarrow \emptyset$
3: **ΟΣΟ** υπάρχουν ενεργές ομάδες στο $\mathscr{C}$ **ΚΑΝΕ**
4: βρίσκουμε τις δύο πλησιέστερες ομάδες C_1, C_2
5: $E_{sol} = E_{sol} \cup \{e | e \in \text{sp}(C_1, C_2)\}$
6: ανανεώνουμε την ομαδοποίηση: $\mathscr{C} = (\mathscr{C} \setminus \{C_1, C_2\}) \cup (C_1 \cap C_2)$
7: **ΤΕΛΟΣ ΟΣΟ**
8: **ΕΠΕΣΤΡΕΨΕ** ένα μεγιστικό υπό-γράφημα $\mathscr{F}$ χωρίς κύκλους από το E_{sol}

Επομένως, ορίζουμε σαν στιγμιότυπο του Steiner δάσους το ζευγάρι $\mathcal{I} = (M, \mathscr{D})$, επιπλέον ορίζουμε ως *ομαδοποίηση* ένα σύνολο ομάδων $\mathscr{C} = \{C_1, C_2, \cdots, C_l\}$ όπου κάθε ομάδα αποτελείται από τερματικούς κόμβους. Μια ομάδα C_i θα λέμε ότι είναι ενεργή αν και μόνο αν υπάρχει ζευγάρι $(s_j, t_j) \in \mathscr{D}$ τέτοιο ώστε:

- Αν $s_j \in C_i$ τότε $t_j \notin C_i$
- Αν $s_j \notin C_i$ τότε $t_j \in C_i$

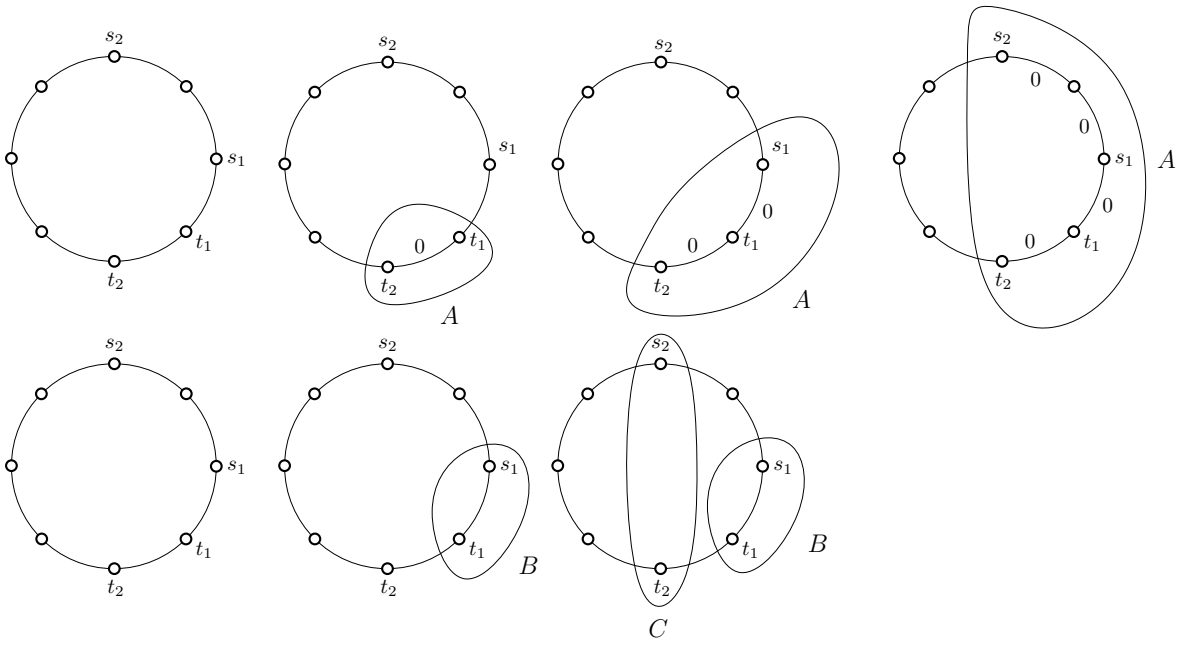
Η αρχική ομαδοποίηση είναι η *τετριμμένη* ομαδοποίηση, δηλαδή για το κάθε τερματικό κόμβο δημιουργούμε μια μοναδιαία ομάδα. Επομένως στην αρχή όλες οι ομάδες θα είναι ενεργές. Στη γραμμή 4 η απόσταση των ομάδων εξαρτάται από την τρέχουσα ομαδοποίηση. Δηλαδή, ξεκινώντας με την τετριμμένη ομαδοποίηση η απόσταση υπολογίζεται σύμφωνα με τη συνάρτηση $d : E' \to \mathbb{R}^+$, πιο συγκεκριμένα η απόσταση μεταξύ δύο ομάδων C_1 και C_2 ορίζεται ως εξής:

$$d_{M_{\mathscr{C}}}(C_1, C_2) = \min_{s \in C_1, t \in C_2} d(s, t)$$

Για την αρχική τετριμμένη ομαδοποίηση $\mathscr{C}_{init}$ η συνάρτηση απόστασης $d_{M_{\mathscr{C}_{init}}}(\cdot,\cdot)$ ταυτίζεται με την συνάρτηση d της μετρικής κλειστότητας $M = (V, E', d)$. Άρα στη γραμμή 6 εκτός από την ανανέωση της ομαδοποίησης πρέπει να ανανεώσουμε και την κλειστή μετρική στην αντίστοιχη $M_{\mathscr{C}}$.

Αυτό γίνεται αλλάζοντας τη συνάρτηση απόστασης d, για την ακρίβεια η απόσταση μεταξύ δύο τερματικών της ίδιας ομάδας μηδενίζεται και οι αποστάσεις μεταξύ τερματικού και Steiner κόμβου όπως και η απόσταση μεταξύ δύο τερματικών κόμβων που ανήκουν όμως σε διαφορετικές ομάδες παραμένουν ίδιες. Η απόσταση λοιπόν είναι το ελάχιστο μονοπάτι σε αυτό το καινούριο γράφημα όπου έχουν γίνει αλλαγές στη συνάρτηση απόστασης d.

Πριν προχωρήσουμε στην ανάλυση των Gupta και Kumar[13] ας δούμε ένα αναλυτικό παράδειγμα με την εκτέλεση του αλγορίθμου και πως αυτή διαφέρει από την εκτέλεση του Άπληστου.



Σχήμα 2.3: Στην πρώτη σειρά φαίνεται η εκτέλεση του "Αδηφάγου" Αλγόριθμου και στη δεύτερη σειρά η εκτέλεση του Άπληστου Αλγόριθμου.

Όπως μπορούμε να δούμε στο Σχήμα 2.3 ο "Αδηφάγος" θα συγχωνεύσει πρώτα τους κόμβους t_1, t_2 δημιουργώντας μια ομάδα A. Στη συνέχεια θα προσθέσει τον κόμβο s_1 στην ομάδα A και τέλος θα προσθέσει και τον κόμβο s_1. Αντίθετα ο Άπληστος Αλγόριθμος θα δημιουργήσει μια ομάδα με το πρώτο ζευγάρι (s_1, t_1) και στο επόμενο βήμα θα δημιουργήσει μια ομάδα με το δεύτερο ζευγάρι (s_2, t_2). Όπως μπορούμε να παρατηρήσουμε και οι δύο αλγόριθμοι κάθε φορά που δημιουργούν μια ομάδα μηδενίζουν το βάρος των ακμών του αντίστοιχου συντομότερου μονοπατιού.

2.3 Ανάλυση του "Αδηφάγου" Αλγορίθμου

Θα εξετάσουμε πρώτα την ανάλυση του αλγόριθμου σύμφωνα με τους Gupta, Kumar [13]. Ξεκινάμε με έναν βασικό ορισμό τον οποίο θα χρησιμοποιήσουμε αργότερα και στη δική μας ανάλυση.

Ορισμός 2. *[13](Ορισμός 3.2) Ένα δάσος $\mathscr{F}$ είναι πιστό σε μια ομαδοποίηση $\mathscr{C}$ αν κάθε ομάδα $C \in \mathscr{C}$ περιέχεται εξολοκλήρου σε ένα δέντρο $T \in \mathscr{F}$ ή αλλιώς $\forall C \in \mathscr{C},\ \exists T \in \mathscr{F},\ s.t.\ C \subseteq T$.*

Παρατηρούμε ότι όλα τα δάση (εφικτές λύσεις) είναι πιστά στην τετριμμένη ομαδοποίηση. Το παρακάτω θεώρημα είναι το ένα από τα δύο συστατικά για την απόδειξη του σταθερού λόγου προσέγγισης, όπως διατυπώνεται στο [13].

Θεώρημα 4. *[13] (Θεώρημα 3.3)*

Έστω $\mathscr{F}_{OPT}$ η βέλτιστη λύση για κάποιο στιγμιότυπο $\mathcal{I} = (M, \mathscr{D})$ του προβλήματος. Υπάρχει κάποια άλλη λύση $\mathscr{F}'$ για το ίδιο στιγμιότυπο $\mathcal{I}$ τέτοια ώστε:

- $\text{cost}(\mathscr{F}') \leq 2 \cdot \text{cost}(\mathscr{F}_{OPT})$
- *το $\mathscr{F}'$ είναι πιστό στην τελική ομαδοποίηση του αδηφάγου $\mathscr{C}_f$.*

Θα αναφέρουμε την ιδέα τις απόδειξης. Για την αναλυτική απόδειξη παραπέμπουμε στο [13]. Η ιδέα είναι καθώς εκτελείται ο "Αδηφάγος" Αλγόριθμος να γίνονται κάποιες αλλαγές στη βέλτιστη λύση και έτσι να υπολογίσουμε τον λόγο προσέγγισης. Στην αρχή ξεκινάμε με $\mathscr{F}_{OPT} = \mathscr{F}'$ και στη συνέχεια παρακολουθούμε την εκτέλεση του "Άδηφάγου" Αλγόριθμου. Κάθε φορά που ο αλγόριθμος συγχωνεύει δύο ομάδες C_1, C_2 δύο είναι οι πιθανές περιπτώσεις. Είτε το $\mathscr{F}'$ παραμένει πιστό στην ομαδοποίηση, είτε όχι, που σημαίνει ότι δύο δέντρα T_1, T_2 συγχωνεύονται. Στην πρώτη περίπτωση συνεχίζουμε κανονικά την εκτέλεση, στη δεύτερη περίπτωση προσθέτουμε τις ακμές για να συγχωνεύσουμε τα T_1, T_2. Τελικά, με ένα πάνω όριο στο κόστος του μονοπατιού που προστίθεται για τη συγχώνευση προκύπτει το θεώρημα.

Έτσι, στην περίπτωση που το βέλτιστο δάσος $\mathscr{F}_{OPT}$ δεν είναι πιστό στην τελική ομαδοποίηση $\mathscr{C}_f$, το μετατρέπουμε σε πιστό. Άρα, το τελικό κόστος θα είναι το πολύ δύο φορές το αρχικό κόστος. Με το επόμενο θεώρημα, έτσι όπως διατυπώνεται στο [13], καταλήγουμε στο σταθερό λόγο προσέγγισης του "Αδηφάγου" Αλγόριθμου.

Θεώρημα 5. *[13](Θεώρημα 3.4)*

Αν το $\mathscr{F}_{OPT}$ είναι πιστό στην τελική ομαδοποίηση $\mathscr{C}_f$ του αδηφάγου τότε ο αλγόριθμος μας δίνει τελικό κόστος $48 \cdot \text{cost}(\mathscr{F}_{OPT})$.

Θα αναφέρουμε σε γενικές γραμμές την απόδειξη αυτού του θεωρήματος. Όπως μπορούμε να δούμε στο [13], κατά την εκτέλεση του αλγορίθμου, διατηρείται ένα δάσος. Το δάσος αυτό θα είναι μια υποψήφια λύση, που θα καταλήξει ο αλγόριθμος και θα είναι σχετικό με τη βέλτιστη λύση. Άρα, αν θεωρήσουμε ότι το βέλτιστο δάσος $\mathscr{F}_{OPT}$ είναι πιστό

(αν δεν είναι απλά το μετατρέπουμε ανεβάζοντας το κόστος κατά, το πολύ, δύο φορές του αρχικού από Θεώρημα 4), τότε σε κάθε επανάληψη του αδηφάγου το κόστος του υποψήφιου δάσους $\mathscr{F}_t$ θα μειώνεται κατά μια ποσότητα. Η ποσότητα αυτή θα είναι ένα σταθερό ποσοστό του κόστους που επιβάλει ο αδηφάγος, άρα, τελικά θα έχουμε σταθερή προσέγγιση.

Πιο συγκεκριμένα, δύο αλλαγές συμβαίνουν κατά τη διάρκεια της εκτέλεσης του "Αδηφάγου" Αλγόριθμου. Αλλαγή στο στιγμιότυπο $\mathcal{I}_t$ και αλλαγή στο δάσος $\mathscr{F}_t$. Η αλλαγή στο $\mathcal{I}_t$ γίνεται ανάλογα με την ομαδοποίηση $\mathscr{C}_t$. Στην αρχή της επανάληψης t θα έχουμε και την ομαδοποίηση $\mathscr{C}_t$, από αυτήν κατασκευάζουμε το στιγμιότυπο $\mathcal{I}_t$ που θα αποτελείται από κόμβους που θα αντιστοιχούν στις ενεργές ομάδες της ομαδοποίησης. Αυτό δεν θα επηρεάσει καθόλου την τελική ομαδοποίηση $\mathscr{C}_f$ γιατί οι ομάδες, που δεν είναι ενεργές, δεν συγχωνεύονται περαιτέρω.

Η δεύτερη αλλαγή είναι στο υποψήφιο δάσος, το οποίο θα είναι εφικτή λύση για το $\mathcal{I}_t$ που έχει μείνει. Ουσιαστικά καθώς αλλάζει το στιγμιότυπο, ισχύουν οι δύο παρακάτω ιδιότητες. Όταν μια από αυτές παραβιάζεται, τότε θα πρέπει να γίνουν οι αντίστοιχες αλλαγές στο δάσος $\mathscr{F}_t$ για διορθωθεί η παραβίαση των ιδιοτήτων.

1. Πρέπει το δάσος $\mathscr{F}_t$ να είναι εφικτή λύση για το υπολειπόμενο στιγμιότυπο $\mathcal{I}_t$.

2. Πρέπει το δάσος $\mathscr{F}_t$ να διατηρεί την δομή του αρχικού δάσους (π.χ. $\mathscr{F}_{OPT}$). Δηλαδή αν δύο ενεργά τερματικά που περιέχονται σε ένα δέντρο στο $\mathscr{F}_{OPT}$ τότε οι ομάδες που ανήκουν θα πρέπει να περιέχονται σε ένα δέντρο στο $\mathscr{F}_t$.

Η πρώτη ιδιότητα είναι προφανής, γιατί αν το δάσος δεν είναι εφικτή λύση τότε δεν έχει νόημα να το αναλύσουμε. Όσο για τη δεύτερη ιδιότητα, θα πρέπει να ισχύει για να διατηρηθεί ουσιαστικά η πιστότητα. Στην αρχή θα ισχύει από την υπόθεση, μετά οι ομάδες που θα συγχωνεύονται θα ανήκουν στο ίδιο δέντρο γιατί ξέρουμε ότι το δάσος $\mathscr{F}_{OPT}$ θα είναι πιστό στην τελική ομαδοποίηση $\mathscr{C}_f$.

Άρα, όταν δύο ομάδες συγχωνεύονται και ο αδηφάγος θα έχει επιλέξει άλλο μονοπάτι (εσωτερικό του δέντρου), θα δημιουργηθεί κύκλος. Από αυτόν τον κύκλο αφαιρούμε την ακμή με το μεγαλύτερο κόστος. Επίσης, αφαιρούμε τους κόμβους Steiner (ουσιαστικά ομάδες που δεν είναι ενεργές) με βαθμό 2 και παρακολουθούμε το συνολικό κόστος, μέσω του δυναμικού $\psi(e)$ κάθε ακμής e.

Στη συνέχεια μετράμε το κόστος που επιβάλει ο αδηφάγος Δ_t στην αντίστοιχη επανάληψη με το δυναμικό των ακμών που ανήκουν στο σύνολο $\mathsf{del}(\infty)$, το οποίο είναι το σύνολο όλων των ακμών που αφαιρέθηκαν από κύκλους, για ένα συγκεκριμένο δέντρο του $\mathscr{F}_{OPT}$ έστω $T^* \in \mathscr{F}_{OPT}$. Έτσι, μετά από το Θεώρημα 6, έτσι όπως διατυπώνεται στο [13], καταλήγουμε στο:

$$\sum_{t \in R_{iter}} \Delta_t \leq 48 \sum_{e \in \mathsf{del}(\infty)} \psi(e) = 48 \cdot \mathsf{cost}(T^*)$$

δηλαδή το άθροισμα του κόστους συγχώνευσης του αδηφάγου σε όλες τις σχετικές επαναλήψεις με το δέντρο T^* είναι το πολύ 48 φορές το άθροισμα όλων των δυναμικών του

αντίστοιχου δέντρου που ουσιαστικά είναι το κόστος του δέντρου. Τέλος αν αθροίσουμε για όλα τα δέντρα θα πάρουμε ότι το συνολικό κόστος θα είναι $48 \cdot \text{cost}(\mathscr{F}_{OPT})$.

Θεώρημα 6. *[13](Θεώρημα 3.7)*

Αν t_0 είναι σχετική επανάληψη τότε υπάρχουν τουλάχιστον $N_{t_0}/8$ ακμές στο del(∞) *με δυναμικό τουλάχιστον $\Delta_{t_0}/6$.*

2.4 Ανάλυση του "Αδηφάγου" Αλγορίθμου σε Δέντρα

Ο αδηφάγος αλγόριθμος έχει δύο περιπτώσεις στην ανάλυσή του, όπως είδαμε και προηγουμένως μπορεί το δάσος μιας λύσης να είναι είτε πιστό είτε όχι. Επομένως θα ήταν καλό να μελετήσουμε τις δύο περιπτώσεις ξεχωριστά και στο τέλος να κάνουμε μια συνολική σύγκριση.

2.4.1 Η Περίπτωση Πιστού Δάσους

Στην περίπτωση που το δάσος είναι πιστό στην τελική ομαδοποίηση $\mathscr{C}_f$ του αλγορίθμου, οι τερματικοί κόμβοι κάθε ομάδας S θα ανήκουν σε ένα μοναδικό δέντρο T^* στη λύση F^*.

Λήμμα 3. *Έστω T_1^* και T_2^*, κάποια δέντρα που ανήκουν στο F^*. Κατά την εκτέλεση του αλγορίθμου δεν θα υπάρξει συγχώνευση των T_1^*, T_2^*.*

Απόδειξη. Έστω ότι ο αλγόριθμος έχει ενώσει δύο διαφορετικά δέντρα του F^*, αυτό μπορεί να γίνει αν κάποια στιγμή κατά την εκτέλεση συγχώνευσε δύο ομάδες, έστω S_1 και S_2, που ανήκαν σε διαφορετικά δέντρα. Αυτό όμως δεν μπορεί να έχει συμβεί γιατί τότε η ομάδα $S = S_1 \cup S_2$ θα είχε τερματικά σε δύο διαφορετικά δέντρα και το F^* δεν θα ήταν πιστό, το οποίο είναι η αρχική υπόθεσή μας. □

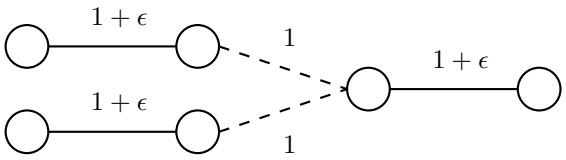
Λήμμα 4. *Αν το δάσος της βέλτιστης λύσης είναι πιστό στην τελική ομαδοποίηση του αδηφάγου, τότε ο αδηφάγος βρίσκει τη βέλτιστη λύση.*

Απόδειξη. Από το Λήμμα 3 βλέπουμε ότι αν το δάσος κάποιας λύσης είναι πιστό στην τελική ομαδοποίηση του αδηφάγου τότε ο αλγόριθμος δεν θα συγχωνεύσει κάποια δέντρα της λύσης. Ακολουθώντας την ίδια λογική λοιπόν με την απόδειξη για τον κλασικό άπληστο, ο αλγόριθμος θα είναι βέλτιστος. □

2.4.2 Η Περίπτωση μη Πιστού Δάσους

Σε αυτή τη περίπτωση ο αδηφάγος αλγόριθμος συνδέει κάποια δέντρα της βέλτιστης λύσης και αυτό θα δώσει μια λύση με κόστος μεγαλύτερο από αυτό της βέλτιστης. Όμως σύμφωνα με τους Gupta και Kumar [13], για να είναι το δάσος της βέλτιστης λύσης πιστό στην τελική ομαδοποίηση του αδηφάγου, οι ακμές που θα προστεθούν δεν θα αυξήσουν το κόστος της λύσης πάνω από $2 \cdot OPT$.

Λήμμα 5. *Ο λόγος προσέγγισης του αλγόριθμου Gluttonous σε μια οικογένεια από γραφήματα, που αποτελείται από gadgets (Σχήμα 2.4) όπως φαίνεται στο Σχήμα 2.5, είναι* 2.



Σχήμα 2.4: Ένα Gadget.

Απόδειξη. Πρώτα θα διευκρινίσουμε την απεικόνιση των ακμών. Οι διακεκομμένες ακμές είναι κανονικές ακμές, οι υπόλοιπες συμβολίζουν τα ζευγάρια κόμβων που θα ανήκουν στις απαιτήσεις. Ας ονομάσουμε το σύνολο των διακεκομμένων ακμών (”κακές” ακμές) E_{bad} και το σύνολο των γεμάτων ακμών (”καλές” ακμές) E_{good}.

Αν παρατηρήσουμε το Σχήμα 2.5 και εκτελέσουμε τον Gluttonous θα δούμε ότι στο τέλος της εκτέλεσης ισχύουν τα εξής: η τελική ομαδοποίηση είναι μία μοναδική ομάδα που περιέχει όλα τα τερματικά και ο Gluttonous έχει επιλέξει στη λύση όλες τις ακμές του γραφήματος.

Για την βέλτιστη λύση παρατηρούμε ότι αρκεί να επιλέξουμε την απευθείας ακμή που συνδέει κάθε ζευγάρι ξεχωριστά.

Αν υποθέσουμε ότι έχουμε ένα γράφημα με λ gadgets, τότε θα έχουμε ότι $|E_{bad}| = 3\lambda - 1$ και $|E_{good}| = 3\lambda$.

Επομένως:

$$SOL_{OPT} = (1+\epsilon)\,|E_{good}|$$

και:

$$SOL_{GLUT} = (1+\epsilon)\,|E_{good}| + |E_{bad}|$$

Άρα, ο λόγος προσέγγισης θα είναι:

$$\rho = \frac{SOL_{GLUT}}{SOL_{OPT}} = \frac{(1+\epsilon)\,|E_{good}| + |E_{bad}|}{(1+\epsilon)\,|E_{good}|} = 1 + \frac{|E_{bad}|}{(1+\epsilon)\,|E_{good}|}$$

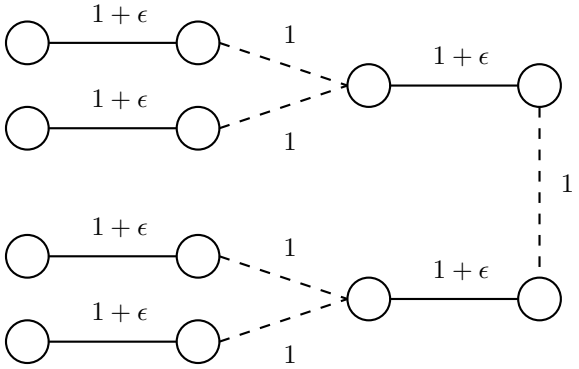
Επειδή $|E_{good}| = 3\lambda$ και $|E_{bad}| = 3\lambda - 1$, καταλήγουμε στο εξής:

$$\rho = 1 + \frac{1}{1+\epsilon} - \frac{1}{(1+\epsilon)3\lambda}$$

Άρα, αν επιλέξουμε ένα $\epsilon > 0$ αλλά πολύ κοντά στο 0 όταν το $\lambda \to \infty$ το $\rho \to 2$. □

Στα δέντρα μετά την μετατροπή του δάσους της βέλτιστης λύσης σε πιστό, προκύπτει ότι στο δεύτερο τμήμα της ανάλυσης του αδηφάγου αλγόριθμου δεν θα αυξηθεί περαιτέρω το κόστος. Επομένως, μαζί με το σφιχτό παράδειγμα του Λήμματος 5 και το Λήμμα 4 προκύπτει το παρακάτω θεώρημα.

Θεώρημα 7. *Ο “Αδηφάγος” Αλγόριθμος σε μετρικές κλειστότητας συντομότερου μονοπατιού, που προκύπτουν από δέντρα, είναι 2-προσεγγιστικός.*



Σχήμα 2.5: Δύο Gadgets συνδεδεμένα.

2.5 Ανάλυση του Άπληστου Αλγορίθμου σε Δέντρα

Όπως είδαμε στην Ενότητα 2.2 οι δύο συνδυαστικοί αλγόριθμοι είναι ο *Κλασικός Άπληστος Αλγόριθμος (Paired Greedy Algorithm)* και ο *"Αδηφάγος" Αλγόριθμος (Gluttonous Algorithm)*. Από εδώ και στο εξής θα αναφερόμαστε στον Κλασικό Άπληστο Αλγόριθμο ως απλά Άπληστο Αλγόριθμο. Η ανάλυση για τον "Αδηφάγο" Αλγόριθμο περιλαμβάνει την έννοια της πιστότητας (faithfulness) η οποία αντιστοιχεί στον Άπληστο Αλγόριθμο ως εξής:

Λήμμα 6. *Το δάσος οποιασδήποτε εφικτής λύσης σε γράφημα δέντρου είναι πιστό στην τελική ομαδοποίηση του Άπληστου Αλγόριθμου.*

Απόδειξη. Σύμφωνα με τον ορισμό της πιστότητας, οι ομάδες της τελικής ομαδοποίησης θα πρέπει να περιέχονται σε ένα μοναδικό δέντρο της λύσης. Οι ομάδες που δημιουργεί ο Άπληστος Αλγόριθμος είναι τα ίδια τα ζευγάρια τερματικών που θέλουμε να υπάρχει μονοπάτι μεταξύ τους. Για παράδειγμα, αν ένα ζευγάρι είναι το (u, v) τότε ο αλγόριθμος φτιάχνει μια ομάδα με αυτά τα τερματικά $\{u, v\}$ και δεν τη συγχωνεύει με καμία άλλη. Για να είναι η λύση εφικτή πρέπει κάθε ζευγάρι τερματικών να ανήκει σε κάποιο δέντρο, επομένως το δάσος οποιασδήποτε εφικτής λύσης είναι πιστό (faithful) στην τελική ομαδοποίηση του Άπληστου Αλγορίθμου. □

Θεώρημα 8. *Ο Άπληστος Αλγόριθμος είναι βέλτιστος στα δέντρα.*

Απόδειξη. Για να καταλάβουμε γιατί συμβαίνει αυτό, πρώτα πρέπει να εξετάσουμε πως θα ήταν μια λύση σε ένα δέντρο. Αν αφαιρέσουμε μια ακμή από ένα δέντρο, τότε το γράφημα δεν θα είναι πλέον συνεκτικό. Για να είναι ένα δάσος εφικτή λύση, θα πρέπει οι κόμβοι του κάθε ζευγαριού τερματικών να ανήκουν και οι δύο το ίδιο δέντρο της λύσης. Έστω (s, t) ένα ζευγάρι τερματικών και έστω $P(s, t)$ το σύνολο ακμών του μονοπατιού ανάμεσα στους δύο τερματικούς κόμβους. Αν αφαιρέσουμε από τη λύση κάποια ακμή που να ανήκει στο $P(s, t)$ τότε οι δύο κόμβοι θα ανήκουν σε διαφορετικά συνεκτικά τμήματα. Άρα, θα πρέπει όλες οι ακμές στα μονοπάτια των ζευγαριών τερματικών να ανήκουν στη λύση.

Βέβαια κάποια μονοπάτια μπορεί να επικαλύπτονται. Σε αυτήν την περίπτωση αν αφαιρέσουμε κάποια ακμή από την τομή των μονοπατιών τότε διαχωρίζουμε όσα ζευγάρια έχουν

κοινές ακμές στην τομή. Δηλαδή, έστω (s_1, t_1) και (s_2, t_2) και έστω ότι $P(s_1, t_1) \cap P(s_2, t_2) \neq \emptyset$. Τότε αν αφαιρέσουμε μια ακμή από την τομή διαχωρίζουμε αυτά τα δύο ζευγάρια, αντίστοιχα μπορούμε να γενικεύσουμε εύκολα για περισσότερα από δύο ζευγάρια/μονοπάτια.

Πως θα ήταν η δομή της βέλτιστης λύσης; Είδαμε παραπάνω ότι δεν μπορούμε να αφαιρέσουμε κάποια ακμή από κάποιο μονοπάτι σε οποιαδήποτε λύση (άρα και στη βέλτιστη). Επίσης αν προσθέσουμε κάποια ακμή που δεν ανήκει σε κανένα μονοπάτι που ενώνει κάποιο ζευγάρι τερματικών θα είναι πλεονασμός, γιατί αν δεν την είχαμε επιλέξει στη λύση, δεν θα χωρίζαμε κάποιο ζευγάρι τερματικών. Επομένως, η βέλτιστη λύση θα πρέπει να περιέχει ακριβώς τις ακμές των μονοπατιών που ενώνουν τα ζευγάρια τερματικών. Το οποίο είναι ακριβώς αυτό που κάνει ο άπληστος αλγόριθμος. □

3. ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΑ

3.1 Υλοποίηση

Η υλοποίηση πραγματοποιήθηκε χρησιμοποιώντας τις βιβλιοθήκες που παρέχει το σύστημα διαχείρισης πακέτων *ANACONDA* (για περισσότερες πληροφορίες μπορείτε να ακολουθήσετε τον εξής σύνδεσμο `https://docs.continuum.io/`). Για την υλοποίηση των δύο αλγορίθμων χρησιμοποιήσαμε τη γλώσσα προγραμματισμού *Python 2.7* και το περιβάλλον προγραμματισμού *JetBrains PyCharm Community Edition 2016.3*, επομένως παρακάτω θα μιλάμε με κάποιους όρους της συγκεκριμένης γλώσσας που όμως είναι κατανοητοί από το όνομα, ακόμα και αν δεν είναι μπορεί κάποιος να βρει εύκολα τη λειτουργία τους γιατί είναι ευρέως γνωστοί. Οτιδήποτε άλλο έχει να κάνει με βαθύτερες έννοιες θα εξηγείται.

Η υλοποίηση που πραγματοποιήθηκε ουσιαστικά είναι μια κλάση για την αναπαράσταση των γραφημάτων, μια κλάση για τον ”Αδηφάγο” Αλγόριθμο και μια κλάση για τον Απλοϊκό Άπληστο Αλγόριθμο στον οποίο θα αναφερόμαστε ως Άπληστο Αλγόριθμο.

3.1.1 Γράφημα

Πρώτα έπρεπε να υπάρχει ένας τρόπος να αναπαραστήσουμε ένα γράφημα προγραμματιστικά. Η αναπαράσταση γίνεται μέσω της κλάσης (class) `Graph`. Τα χαρακτηριστικά (attributes) αυτής της κλάσης είναι ένα λεξικό (dictionary) για την αναπαράσταση των κόμβων και τον ακμών και ένα λεξικό για την αναπαράσταση του βάρους κάθε ακμής. Η κλάση περιλαμβάνει διάφορες μεθόδους (methods) που χρησιμεύουν στην επεξεργασία ενός αντικειμένου(object) της κλάσης αυτής.

Η κατασκευή (instantiation) ενός αντικειμένου της κλάσης γίνεται μέσω του κατασκευαστή(constructor), το μόνο όρισμα (parameter) που μπορεί να δώσει ο χρήστης είναι μια λίστα (list) από ζευγάρια της μορφής `(u, v)` που αντιπροσωπεύουν της ακμές του γραφήματος που θέλουμε να κατασκευάσουμε, αν δεν δοθεί μια τέτοια λίστα τότε το αντικείμενο που παράγεται είναι ένα κενό γράφημα (κανένας κόμβος και καμία ακμή). Από τη λίστα των ακμών εξάγουμε τους κόμβους του γραφήματος και έπειτα για κάθε κόμβο δημιουργούμε μια λίστα (list) από τους γείτονες του. Άν η λίστα (list) των ακμών περιέχει ακμές της μορφής `(u, v, w)` το `w` προστίθεται σαν βάρος της συγκεκριμένης ακμής στο λεξικό (dictionary) των βαρών.

Έχουν υλοποιηθεί διάφορες μέθοδοι (methods) για την επεξεργασία των δεδομένων του γραφήματος όπως αλλαγή του κόστους κάποιας ακμής, ανάκτηση της λίστας γειτόνων κάποιου κόμβου, υπολογισμός του συνολικού κόστους μιας λίστας από ακμές και άλλα. Οι βασικές μέθοδοι όμως είναι οι εξής:

- `metric_closure()`
- `make_random_weighted_tree(num_of_nodes, min_w, max_w)`

- `make_random_weighted_graph(num_of_nodes, min_w, max_w)`
- `make_complete_graph(num_of_nodes, min_w, max_w)`

Η μέθοδος `metric_closure()` χρησιμοποιείται για την κατασκευή της κλειστής μετρικής του γραφήματος για την οποία την καλούμε. Ο αλγόριθμος για την κατασκευή μιας τέτοιας μετρικής έχει προαναφερθεί στο Κεφάλαιο 2.1, η μέθοδος επιστρέφει το νέο γράφημα που αντιπροσωπεύει τη μετρική αλλά και ένα λεξικό που αντιστοιχεί κάθε ακμή της μετρικής με το αντίστοιχο μονοπάτι του αρχικού γραφήματος.

Η άλλες τρεις μέθοδοι κατασκευάζουν τρία είδη γραφημάτων, όπου για ορίσματα δίνουμε τον αριθμό των κόμβων(`num_of_nodes`), το ελάχιστο βάρος(`min_w`) που μπορεί να έχει κάθε ακμή καθώς και το μέγιστο βάρος(`max_w`) που μπορεί να έχει κάθε ακμή. Με τη μέθοδο

```
make_random_weighted_tree(num_of_nodes, min_w, max_w)
```

κατασκευάζουμε ένα τυχαίο δέντρο, όπου το κόστος κάθε ακμής είναι ένας τυχαίος ακέραιος αριθμός στο διάστημα $[min\_w, max\_w]$. Η μέθοδος

```
make_random_weighted_graph(num_of_nodes, min_w, max_w)
```

κατασκευάζει ένα σχεδόν τυχαίο γράφημα με τυχαία βάρη ανάμεσα στα όρια $[min\_w, max\_w]$. Η τρίτη μέθοδος

```
make_complete_graph(num_of_nodes, min_w, max_w)
```

κατασκευάζει το πλήρες γράφημα όπου το κόστος κάθε ακμής είναι ένας τυχαίος ακέραιος αριθμός στο διάστημα $[min\_w, max\_w]$. Η αναλυτικότερη περιγραφή των τριών μεθόδων για τη δημιουργία γραφημάτων βρίσκεται σε παρακάτω ενότητες αντίστοιχα για κάθε μέθοδο. Για κάθε τυχαία διαδικασία χρησιμοποιήθηκε το πακέτο `numpy.random`.

3.1.2 Ο Άπληστος Αλγόριθμος

Ο Άπληστος Αλγόριθμος, όπως και ο "Αδηφάγος" Αλγόριθμος παρακάτω, έχουν υλοποιηθεί σε ξεχωριστές κλάσεις. Για να χρησιμοποιήσει κάποιος τους αλγορίθμους αρκεί να έχει κατασκευάσει ένα αντικείμενο `graph` τις κλάσης `Graph` και στη συνέχεια να κατασκευάσει ένα αντικείμενο του αντίστοιχου αλγορίθμου και μετά να καλέσει την μέθοδο `run()`. Για παράδειγμα:

```
graph = Graph()
graph.make_complete_graph(10, 1, 2)

greedy = PairedGreedy(graph, [(1, 2), (3, 5)])
greedy.run()
print(greedy.Results)
```

εκτός από το γράφημα στην κατασκευή του αντικειμένου του αλγορίθμου πρέπει να δώσουμε σαν όρισμα και τα ζευγάρια που θέλουμε να ανήκουν σε συνεκτικό μέρος της λύσης. Όπως φαίνεται τα αποτελέσματα μπορούμε να τα δούμε στο χαρακτηριστικό του αντικειμένου `Results` το οποίο περιέχει το συνολικό κόστος της λύσης καθώς και τις ακμές που αποτελούν τη λύση.

Ο αλγόριθμος πρώτα υπολογίζει την κλειστή μετρική του γραφήματος που δίνουμε σαν όρισμα και μετά απλά επιλέγει από τη μετρική της ακμές που αντιστοιχούν στα ζευγάρια που δώσαμε σαν είσοδο. Για κάθε ακμή της μετρικής προσθέτουμε στη λύση το αντίστοιχο μονοπάτι στο αρχικό γράφημα. Καθαρίζουμε τη λύση από διπλές ακμές και από κύκλους, εφαρμόζοντας μια προσπέλαση πρώτα κατά βάθος και κρατάμε το συνολικό κόστος των ακμών της λύσης καθώς και τη λίστα των ακμών της λύσης.

3.1.3 Ο “Αδηφάγος” Αλγόριθμος

Η υλοποίηση του ”Αδηφάγου” Αλγόριθμου έχει γίνει στο ίδιο πλαίσιο με του Άπληστου Αλγόριθμου, δηλαδή υπάρχει ένα χαρακτηριστικό `results` στο οποίο τελικά αποθηκεύονται τα αποτελέσματα (’TotalCost’ και ’Forest’). Επίσης όταν κατασκευάζουμε ένα αντικείμενο της κλάσης αυτής το συνδέουμε με κάποιο συγκεκριμένο γράφημα. Στον κατασκευαστή της κλάσης αυτό που γίνεται είναι να κατασκευάσουμε τη κλειστή μετρική του γραφήματος που δώσαμε ως όρισμα για την κατασκευή του αντικειμένου. Στη συνέχεια μπορούμε να καλέσουμε τη μέθοδο `run()` και μετά να διαβάσουμε το αποτέλεσμα από το χαρακτηριστικό `results`. Παρακάτω φαίνεται ένα παράδειγμα εκτέλεσης:

```
graph = Graph()
graph.make_complete_graph(10, 1, 2)

glut = Gluttonous(graph, [(1, 2), (3, 5)])
glut.run()
print(glut.results['Forest'])
print(glut.results['TotalCost'])
```

Ο αλγόριθμος έχει αναφερθεί σε προηγούμενο κεφάλαιο αλλά τον αναφέρουμε και εδώ συνοπτικά για λόγους πληρότητας.

Στην αρχή κατασκευάζουμε την τετριμμένη ομαδοποίηση και στη συνέχεια ξεκινάμε τη βασική επανάληψη. Στην επανάληψη συγχωνεύουμε διαδοχικά τις κοντινότερες ομάδες τερματικών μέχρι να μην υπάρχουν πλέον ενεργές ομάδες. Τα βασικά σημεία είναι η ανανέωση της ομαδοποίησης και η ανανέωση του δάσους. Επίσης για την αναπαράσταση των ομάδων υλοποιήθηκε η κλάση `SuperNode` με μεθόδους όπως `merge()` και `find_two_closest_active_`

3.1.4 Βέλτιστη λύση

Για να μπορέσουμε να βρούμε πειραματικά τους λόγους προσέγγισης πρέπει κάπως να υπολογίσουμε τη βέλτιστη λύση για κάποιο στιγμιότυπο του προβλήματος και μετά να

συγκρίνουμε με τη λύση που δίνουν οι αλγόριθμοι μας. Για την εύρεση της βέλτιστης λύσης χρησιμοποιήθηκε ένα μοντέλο ροών το οποίο περιγράφετε στο αρχείο `steinerf.mod` σύμφωνα με τη γλώσσα *GMPL* (GNU MathProg Language). Επομένως κατασκευάστηκαν αρχεία δεδομένων `.dat` (π.χ. `lp-clique-5-2-1-2-inst0.dat`) και με τη βοήθεια του λογισμικού ανοιχτού κώδικα *GLPK* (GNU Linear Programming Kit) εκτελούμε σε περιβάλλον Linux (στον ίδιο φάκελο όπου βρίσκονται τα αρχεία μοντέλου και δεδομένων) την εντολή

```
glpsol --math -m steinerf.mod -d lp-clique-5-2-1-2-inst0.dat
```

ως αποτέλεσμα λαμβάνουμε την βέλτιστη λύση καθώς και πληροφορίες για τους λόγους προσέγγισης και των λύσεων των δύο αλγορίθμων.

3.2 Πειράματα

Επόμενο της υλοποίησης ήταν η εκτέλεση κάποιων πειραμάτων. Η αρχική ιδέα ήταν να μπορέσουμε να βρούμε κάποιο «κακό» στιγμιότυπο του προβλήματος με σκοπό την χείριστη επίδοση των αλγορίθμων που αναφέραμε. Φυσικά η υλοποίηση βοήθησε και στην κατανόηση του προβλήματος αλλά και της δομής των κακών στιγμιότυπων.

Το υλικό που χρησιμοποιήσαμε είναι ένας φορητός υπολογιστής με επεξεργαστή τεσσάρων πυρήνων και μνήμη RAM μεγέθους 8 GByte. Αναφέρονται σε παρακάτω ενότητα οι μέσοι χρόνοι εκτέλεσης στα διάφορα στιγμιότυπα του προβλήματος. Επίσης σε γραφήματα μεγάλου μεγέθους η εύρεση βέλτιστης τιμής χρονικά είναι αδύνατη και για αυτόν τον λόγο συγκρίνονται οι λύσεις των δύο προσεγγιστικών αλγορίθμων.

3.2.1 Σχεδόν τυχαίο γράφημα

Στην υλοποίηση της κλάσης `Graph` υπάρχει η μέθοδος:

```
def make_random_weighted_graph(self, num_of_nodes=5, min_w=1, max_w=2)
```

Σε αυτή τη μέθοδο υλοποιήσαμε ουσιαστικά την κατασκευή ενός τυχαίου γραφήματος, όπως βλέπουμε η μέθοδος δέχεται τρία ορίσματα(χωρίς να λαμβάνουμε υπ' όψιν το όρισμα `self`). Ο αλγόριθμος κατασκευάζει ένα γράφημα με `num_of_nodes` κόμβους όπου οι ακμές έχουν αντίστοιχα από το ελάχιστο έως και το μέγιστο βάρος που δίνεται. Τα βάρη στις ακμές είναι ακέραιοι και ο αλγόριθμος είναι ο εξής:

1. Έστω `nodes` ένα σύνολο n κόμβων.
2. Για κάθε συνδυασμό κόμβων στο `nodes`

```
for u, v in combinations(nodes, 2):
```

ρίχνουμε ένα νόμισμα

```
coin = random.random_integers(0, 1)
```

και αν το αποτέλεσμα είναι 1 τότε, προσθέτουμε την αντίστοιχη ακμή στο σύνολο ακμών

```
\texttt{edges.append((u, v))}.
```

3. Στη συνέχεια για τα ζευγάρια κόμβων σε αύξουσα σειρά (δηλ. $(1,2),(2,3),(3,4),\cdots$) προσθέτουμε τις ακμές που δεν υπάρχουν. Με αυτόν τον τρόπο θα είναι σίγουρο ότι το τελικό γράφημα θα είναι συνεκτικό.
4. Τέλος προσθέτουμε τυχαία βάρη για κάθε ακμή.

3.2.2 Πλήρες γράφημα

Η υλοποίηση για την κατασκευή πλήρες γραφήματος πραγματοποιήθηκε στην παρακάτω μέθοδο της κλάσης `Graph`.

```
def make_complete_graph(self, num_of_nodes=5, min_w=1, max_w=2)
```

Η μέθοδος δέχεται ως ορίσματα το πλήθος των κόμβων και το ελάχιστο και μέγιστο κόστος που μπορούν να έχουν οι ακμές του γραφήματος. Ο αλγόριθμος κατασκευής είναι πολύ απλός. Για κάθε ζευγάρι κόμβων ορίζουμε μια ακμή και της εκχωρούμε ένα τυχαίο κόστος στο διάστημα $[min\_w, max\_w]$.

3.2.3 Δέντρα

Η υλοποίηση της κατασκευής δεντρικών γραφημάτων βρίσκεται στη μέθοδο

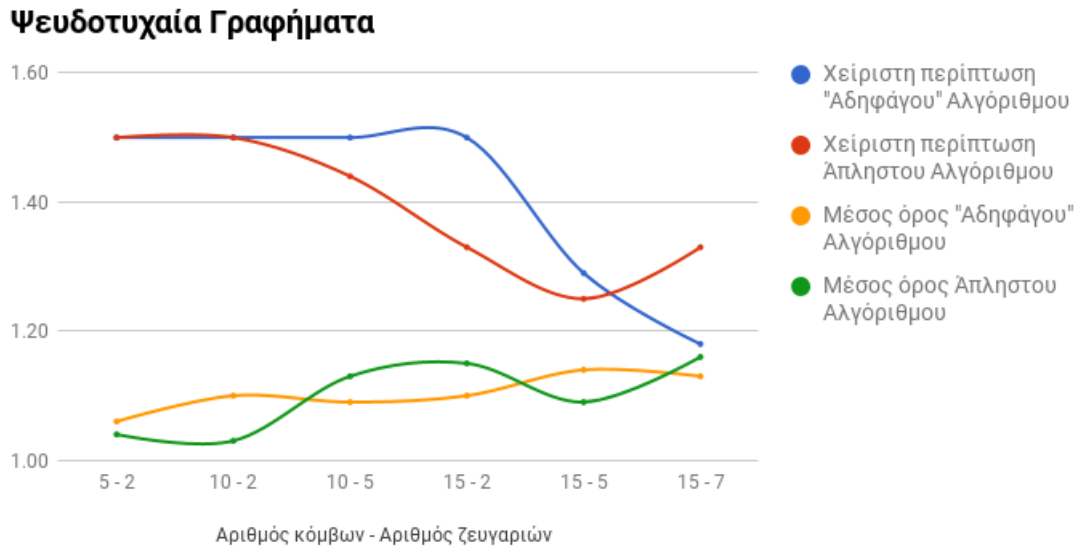
```
def make_random_weighted_tree(self, num_of_nodes=5, min_w=1, max_w=2)
```

της κλάσης `Graph`. Για να κατασκευάσουμε ένα δέντρο θα πρέπει να είμαστε σίγουροι ότι δεν δημιουργούνται κύκλοι. Στην τεχνική που ακολουθήσαμε επιλέγουμε ουσιαστικά κάθε φορά έναν κόμβο που είναι φύλλο (γονιός χωρίς απογόνους) και προσθέτουμε ένα ή περισσότερα παιδιά. Έτσι είναι σαν να κατασκευάζουμε το δέντρο από τη ρίζα προς τα φύλα.

Επιλέγουμε έναν τυχαίο κόμβο από τους συνολικούς κόμβους που θα είναι η ρίζα και τον αφαιρούμε από τους συνολικούς κόμβους. Θεωρούμε κάθε παιδί που προστίθεται

στο γράφημα να είναι γονέας χωρίς παιδιά και για κάθε γονέα χωρίς παιδιά, όσο υπάρχουν ακόμα κόμβοι να προσθέσουμε, αναθέτουμε ένα παιδί. Στη συνέχεια με πιθανότητα $1/2$ μπορεί να ανατεθεί παραπάνω από ένα παιδί στον κάθε γονέα. Όταν προσθέσουμε όλους τους κόμβους από το σύνολο των συνολικών κόμβων η διαδικασία σταματάει. Για να χρησιμοποιήσουμε αυτή τη μέθοδο αρκεί να κατασκευάσουμε ένα κενό γράφημα και στη συνέχεια να καλέσουμε την συγκεκριμένη μέθοδο:

```
tree = Graph()
tree.make_random_weighted_tree(10, 1, 2)
```



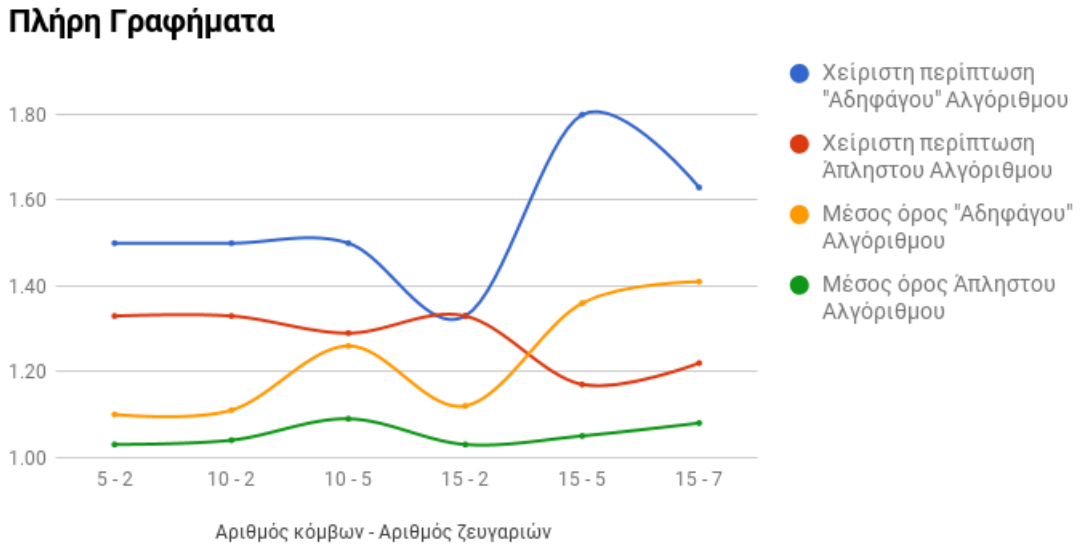
Σχήμα 3.1: Συνοπτικά τα αποτελέσματα για τυχαία γραφήματα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$.

3.2.4 Αποτελέσματα πειραμάτων

Σε αυτήν την ενότητα θα συζητήσουμε τα πειραματικά αποτελέσματα καθώς και τους χρόνους εκτέλεσης των δύο αλγορίθμων.

Τα αποτελέσματα βρίσκονται στο Παράρτημα Ι. Θα αναφέρουμε κάποιους συμβολισμούς για την καλύτερη κατανόηση της σημασίας των αποτελεσμάτων. Σε κάθε πίνακα χρησιμοποιούμε σαν μέτρο σύγκρισης, το λόγο προσέγγισης των δύο αλγορίθμων σε τρεις περιπτώσεις. Η πρώτη περίπτωση, όπως φαίνεται και από τους πίνακες, είναι η περίπτωση όπου ο ”Αδηφάγος” Αλγόριθμος είχε την χειρότερη επίδοση (σε σχέση πάντα με την βέλτιστη λύση). Η δεύτερη περίπτωση είναι αυτή στην οποία ο Άπληστος Αλγόριθμος είχε την χειρότερη επίδοση και τέλος ο μέσος όρος των επιδόσεων των δύο αλγορίθμων. Η μόνη εξαίρεση είναι για τα δέντρα, όπου ο Άπληστος Αλγόριθμος είναι πάντα βέλτιστος, όποτε δεν έχει νόημα να ελέγξουμε την χειρότερή του επίδοση.

Σε κάθε πίνακα έχουμε πακέτα εκτελέσεων, δηλαδή οι στήλες όπου το όνομα ξεκινάει με rdm- ή clq- ή tr-, αντιπροσωπεύουν πολλαπλά πειράματα(εκτέλεση των αλγορίθμων σε πολλαπλά στιγμιότυπα) των δύο αλγορίθμων. Κάθε τέτοιο πακέτο εκτελέσεων συμπεριλαμβάνει 100 στιγμιότυπα όταν το μέγεθος είναι μικρό και 10 στιγμιότυπα στα μεγαλύτερα γραφήματα. Αυτοί οι αριθμοί έχουν επιλεγεί σύμφωνα με τους χρόνους εκτέλεσης, γιατί σε κάθε εκτέλεση θα έπρεπε να υπολογίζουμε και τη βέλτιστη λύση το οποίο είναι υπερβολικά χρονοβόρο. Για παράδειγμα η εύρεση της βέλτιστης λύσης στα μεγαλύτερα στιγμιότυπα μπορούσε να διαρκέσει μερικές μέρες.

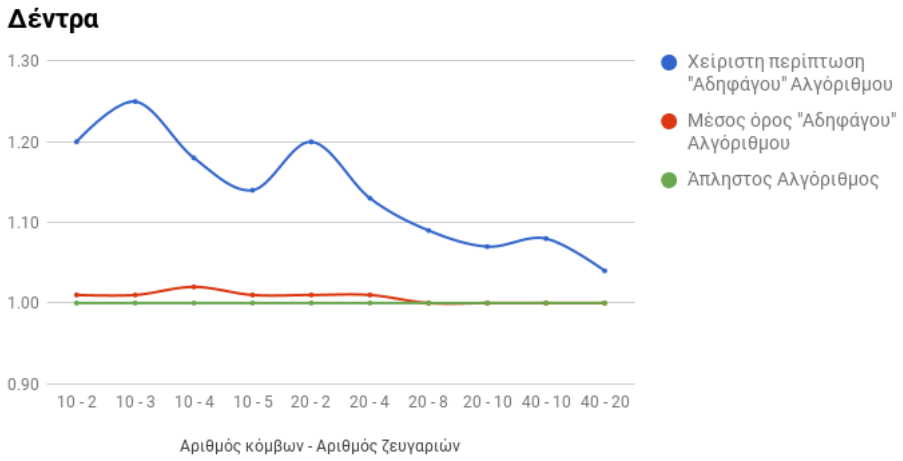


Σχήμα 3.2: Συνοπτικά τα αποτελέσματα για πλήρη γραφήματα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$.

Σε κάθε πακέτο εκτελέσεων από τον τίτλο, ο οποίος είναι και ο τίτλος των αρχείων που περιέχουν τα στιγμιότυπα, μπορούμε να βρούμε πληροφορίες για της παραμέτρους του κάθε στιγμιότυπου. Ο πρώτος αριθμός δείχνει το n, δηλαδή το μέγεθος του γραφήματος, ο δεύτερος αριθμός δείχνει το πλήθος των ζευγαριών τερματικών κόμβων, η αλλιώς σύμφωνα με την ανάλυση των αλγορίθμων το $|\mathscr{D}|$. Ο τρίτος και τέταρτος αριθμός συμβολίζουν το ελάχιστο και μέγιστο βάρος που μπορεί να έχει κάποια ακμή αντίστοιχα. Επίσης τα γράμματα στην αρχή του ονόματος συμβολίζουν τον τύπο του γραφήματος, το οποίο όμως είναι εμφανές και από τη λεζάντα του κάθε πίνακα.

Επομένως μπορούμε να δούμε ότι πραγματοποιήθηκαν πειράματα σε τρία είδη συνεκτικών γραφημάτων (τυχαία γραφήματα, πλήρη γραφήματα, δέντρα). Για τα τυχαία γραφήματα πραγματοποιήθηκαν πειράματα για $n = \{5, 10, 15\}$ και βάρη στα διαστήματα ακεραίων $[1, 2]$ και $[1, 1000]$ για κάθε μέγεθος γραφήματος. Αντίστοιχα τα ίδια πειράματα έγιναν και για τα πλήρη γραφήματα και για τα δέντρα επειδή η εύρεση της βέλτιστης λύσης

ήταν πιο γρήγορη πραγματοποιήθηκαν πειράματα για $n = \{10, 20, 40\}$ στα οποία αν και υπήρχαν κάποια δύσκολα στιγμιότυπα για τον "Αδηφάγο" Αλγόριθμο, κατά μέσο όρο είναι και αυτός πολύ κοντά στη βέλτιστη λύση. Παρατηρούμε από τα αποτελέσματα ότι, όταν ο ένας αλγόριθμος δεν έχει καλή επίδοση (δηλ. χειρότερη επίδοση "Αδηφάγου" Αλγορίθμου και χειρότερη επίδοση Άπληστου Αλγορίθμου), ο άλλος συνήθως είναι βέλτιστος. Αυτό μπορούμε να το παρατηρήσουμε πολύ έντονα για παράδειγμα στον πίνακα I.ii, όπου στα στιγμιότυπα clq-5-2-1-2 όταν ο "Αδηφάγος" Αλγόριθμος έχει χειρότερη επίδοση με λόγο προσέγγισης 1.5 ο Άπληστος Αλγόριθμος είναι βέλτιστος ενώ όταν ο Άπληστος Αλγόριθμος κάνει κακή επίδοση με λόγο προσέγγισης 1.33 ο "Αδηφάγος" Αλγόριθμος είναι βέλτιστος. Βέβαια ως προς τον μέσο όρο οι αλγόριθμοι φαίνεται να είναι αρκετά κοντά. Ίσως θα ήταν ενδιαφέρον να δούμε τη συμβαίνει σε κάθε στιγμιότυπο όπου είναι τουλάχιστον ένας από τους δύο αλγορίθμους βέλτιστος.



Σχήμα 3.3: Συνοπτικά τα αποτελέσματα για δέντρα όπου τα βάρη στις ακμές μπορούν να πάρουν τιμές στο διάστημα $[1, 2]$. Στον κάθετο άξονα έχουμε τον λόγο προσέγγισης και στον οριζόντιο το μέγεθος του γραφήματος καθώς και το $|\mathscr{D}|$.

Τα αποτελέσματα του Παραρτήματος I φαίνονται συνοπτικά στα Σχήματα 3.1, 3.2 και 3.3. Επίσης στο Παράρτημα II φαίνονται συνοπτικά κάποια αποτελέσματα για μεγαλύτερου μεγέθους στιγμιότυπα όπου η εύρεση της βέλτιστης λύσης είναι αδύνατη (χρονικά). Σε αυτήν την περίπτωση, συγκρίνουμε τους αλγορίθμους καθαρά από το συνολικό κόστος που προκύπτει, επίσης κάνουμε μία σύγκριση της τυπικής απόκλισης στα αποτελέσματα του κάθε αλγορίθμου.

4. ΣΥΜΠΕΡΑΣΜΑΤΑ

Συμπερασματικά, σκοπός αυτής της εργασίας ήταν να μελετήσουμε τη συμπεριφορά και να αναλύσουμε δύο άπληστους αλγόριθμους, που επιλύουν προσεγγιστικά και σε πολυωνυμικό χρόνο το πρόβλημα Δάσους Steiner. Πιο συγκεκριμένα, μελετήσαμε τον Άπληστο Αλγόριθμο και τον ”Αδηφάγο” Αλγόριθμο, για τους οποίους πραγματοποιήθηκε θεωρητική αλλά και πρακτική μελέτη.

Η θεωρητική μελέτη πραγματοποιήθηκε σε συγκεκριμένη ομάδα γραφημάτων, τα δέντρα. Παρατηρήσαμε ότι, ο Άπληστος Αλγόριθμος είναι βέλτιστος στη συγκεκριμένη περίπτωση, σε αντίθεση με τον ”Αδηφάγο” Αλγόριθμο, ο οποίος είναι ένας 2-προσεγγιστικός αλγόριθμος για το πρόβλημα Δάσους Steiner. Συγκριτικά, ο ”Αδηφάγος” Αλγόριθμος έχει χειρότερη επίδοση στη συγκεκριμένη ομάδα γραφημάτων γιατί, συγχωνεύει ζευγάρια τα οποία μπορεί να μήν είναι απαραίτητο να ανήκουν στο ίδιο δέντρο της λύσης, με αποτέλεσμα να δίνει υψηλότερο κόστος ως λύση. Ενώ, ο Άπληστος Αλγόριθμος υπολογίζει μονοπάτια μόνο μεταξύ των τερματικών κόμβων που ανήκουν στο ίδιο ζευγάρι και έτσι πετυχαίνει βέλτιστη συμπεριφορά.

Όσον αφορά την συγκριτική μελέτη, υλοποιήθηκαν και αξιολογήθηκαν οι δύο αλγόριθμοι. Η δημιουργία δεδομένων εισόδου, δηλαδή τα γραφήματα που δίνονται σαν είσοδο στους αλγόριθμους, ήταν τεχνητή. Οι αλγόριθμοι αξιολογήθηκαν σε τρία διαφορετικά είδη γραφημάτων και τελικά συγκεντρώθηκαν αποτελέσματα ως προς τον λόγο προσέγγισης των δύο αλγορίθμων. Παρατηρήσαμε ότι στα δέντρα τα αποτελέσματα ήταν αναμενόμενα, καθώς ο Άπληστος Αλγόριθμος ήταν βέλτιστος. Παρόλα αυτά, ο ”Αδηφάγος” Αλγόριθμος δεν απείχε πολύ από την βέλτιστη λύση. Στα πλήρη γραφήματα είδαμε ότι κατά μέσο όρο ο Άπληστος Αλγόριθμος έχει καλύτερη επίδοση, ενώ στα ψευδο-τυχαία γραφήματα οι δύο αλγόριθμοι φαίνεται να έχουν παρόμοια επίδοση.

4.1 Ενδιαφέροντα Προβλήματα

Φυσικά παραμένουν κάποια προβλήματα ανοιχτά τα οποία είναι:

- ”Σφιχτή” ανάλυση στον προσεγγιστικό λόγο του ”Αδηφάγου” Αλγόριθμου.
- Μελέτη των αλγορίθμων σε άλλες ομάδες γραφημάτων.
- Μελέτη των αλγορίθμων με είσοδο μόνο την μετρική κλειστότητας συντομότερων μονοπατιών.
- Εκτενέστερη πειραματική μελέτη.

Επίσης θα ήταν ενδιαφέρουσα η υλοποίηση των αλγορίθμων χρησιμοποιώντας το προγραμματιστικό πακέτο NetworkX (https://networkx.github.io/), το οποίο περιλαμβάνει πολλές μεθόδους για τη αναπαράσταση και μελέτη γραφημάτων. Ανώτερος σκοπός αυτής της υλοποίησης θα ήταν, ίσως, η ενσωμάτωση των αλγορίθμων για την επίλυση του προβλήματος Δάσους Steiner στο συγκεκριμένο προγραμματιστικό πακέτο.

ΠΑΡΑΡΤΗΜΑ Ι. ΠΙΝΑΚΕΣ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

Πίνακας Ι.i: Πειράματα σε τυχαία γραφήματα με $n = \{5, 10, 15\}$.

Μεγέθη σύγκρισης	rdm-5-2-1-2		rdm-5-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.5	1.0	1.5
Χείριστη περίπτωση άπληστου	1.5	1.0	1.6	1.0
Μέσος όρος	1.04	1.06	1.03	1.03
Μεγέθη σύγκρισης	rdm-10-2-1-2		rdm-10-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.5	1.00	1.46
Χείριστη περίπτωση άπληστου	1.5	1.0	1.58	1.00
Μέσος όρος	1.03	1.10	1.04	1.03
Μεγέθη σύγκρισης	rdm-10-5-1-2		rdm-10-5-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.50	1.00	1.21
Χείριστη περίπτωση άπληστου	1.44	1.00	1.62	1.00
Μέσος όρος	1.13	1.09	1.15	1.02
Μεγέθη σύγκρισης	rdm-15-2-1-2		rdm-15-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.50	1.00	1.17
Χείριστη περίπτωση άπληστου	1.33	1.00	1.28	1.00
Μέσος όρος	1.15	1.10	1.08	1.04
Μεγέθη σύγκρισης	rdm-15-5-1-2		rdm-15-5-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.14	1.29	1.10	1.08
Χείριστη περίπτωση άπληστου	1.25	1.00	1.34	1.00
Μέσος όρος	1.09	1.14	1.10	1.02
Μεγέθη σύγκρισης	rdm-15-7-1-2		rdm-15-7-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.09	1.18	1.17	1.04
Χείριστη περίπτωση άπληστου	1.33	1.08	1.37	1.00
Μέσος όρος	1.16	1.13	1.16	1.01

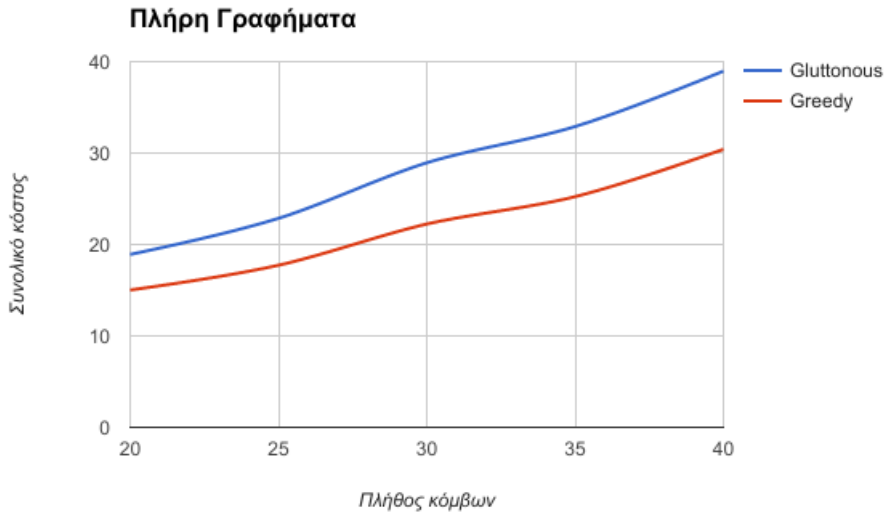
Πίνακας I.ii: Πειράματα σε πλήρη γραφήματα με $n = \{5, 10, 15\}$.

Μεγέθη σύγκρισης	clq-5-2-1-2		clq-5-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.50	1.00	1.42
Χείριστη περίπτωση άπληστου	1.33	1.00	1.83	1.00
Μέσος όρος	1.03	1.10	1.04	1.03
Μεγέθη σύγκρισης	clq-10-2-1-2		clq-10-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.50	1.00	1.38
Χείριστη περίπτωση άπληστου	1.33	1.00	1.48	1.01
Μέσος όρος	1.04	1.11	1.04	1.04
Μεγέθη σύγκρισης	clq-10-5-1-2		clq-10-5-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.50	1.00	1.39
Χείριστη περίπτωση άπληστου	1.29	1.29	1.56	1.00
Μέσος όρος	1.09	1.26	1.14	1.03
Μεγέθη σύγκρισης	clq-15-2-1-2		clq-15-2-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.33	1.00	1.02
Χείριστη περίπτωση άπληστου	1.33	1.00	1.33	1.00
Μέσος όρος	1.03	1.12	1.07	1.002
Μεγέθη σύγκρισης	clq-15-5-1-2		clq-15-5-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.00	1.80	1.07	1.13
Χείριστη περίπτωση άπληστου	1.17	1.50	1.21	1.01
Μέσος όρος	1.05	1.36	1.08	1.03
Μεγέθη σύγκρισης	clq-15-7-1-2		clq-15-7-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.13	1.63	1.06	1.11
Χείριστη περίπτωση άπληστου	1.22	1.33	1.38	1.00
Μέσος όρος	1.08	1.41	1.14	1.02

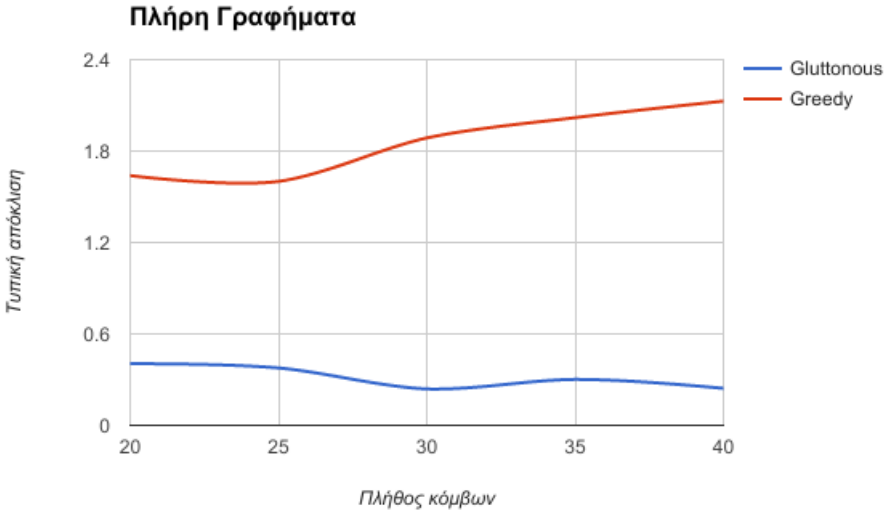
Πίνακας I.iii: Πειράματα σε δέντρα με $n = \{10, 20, 40\}$.

Μεγέθη σύγκρισης	tr-10-2-1-2		tr-10-3-1-2	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.2	1.0	1.25
Μέσος όρος	1.0	1.008	1.0	1.014
Μεγέθη σύγκρισης	tr-10-4-1-2		tr-10-5-1-2	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.182	1.0	1.143
Μέσος όρος	1.0	1.018	1.0	1.005
Μεγέθη σύγκρισης	tr-10-2-1-1000		tr-10-3-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.324	1.0	1.155
Μέσος όρος	1.0	1.009	1.0	1.009
Μεγέθη σύγκρισης	tr-10-4-1-1000		tr-10-5-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.184	1.0	1.2
Μέσος όρος	1.0	1.006	1.0	1.005
Μεγέθη σύγκρισης	tr-20-2-1-2		tr-20-4-1-2	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.2	1.0	1.125
Μέσος όρος	1.0	1.005	1.0	1.007
Μεγέθη σύγκρισης	tr-20-8-1-2		tr-20-10-1-2	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.087	1.0	1.074
Μέσος όρος	1.0	1.003	1.0	1.003
Μεγέθη σύγκρισης	tr-20-2-1-1000		tr-20-4-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.318	1.0	1.173
Μέσος όρος	1.0	1.005	1.0	1.009
Μεγέθη σύγκρισης	tr-20-8-1-1000		tr-20-10-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.087	1.0	1.081
Μέσος όρος	1.0	1.005	1.0	1.002
Μεγέθη σύγκρισης	tr-40-10-1-2		tr-40-20-1-2	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.083	1.0	1.038
Μέσος όρος	1.0	1.002	1.0	1.001
Μεγέθη σύγκρισης	tr-40-10-1-1000		tr-40-20-1-1000	
	Άπληστος	Αδηφάγος	Άπληστος	Αδηφάγος
Χείριστη περίπτωση αδηφάγου	1.0	1.088	1.0	1.086
Μέσος όρος	1.0	1.003	1.0	1.001

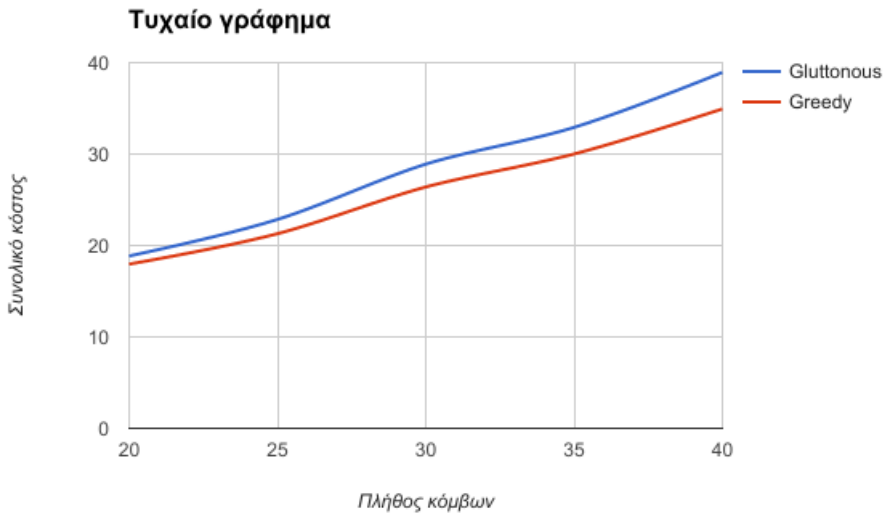
ΠΑΡΑΡΤΗΜΑ ΙΙ. ΣΥΝΟΠΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ ΓΙΑ ΜΕΓΑΛΑ ΣΤΙΓΜΙΟΤΥΠΑ



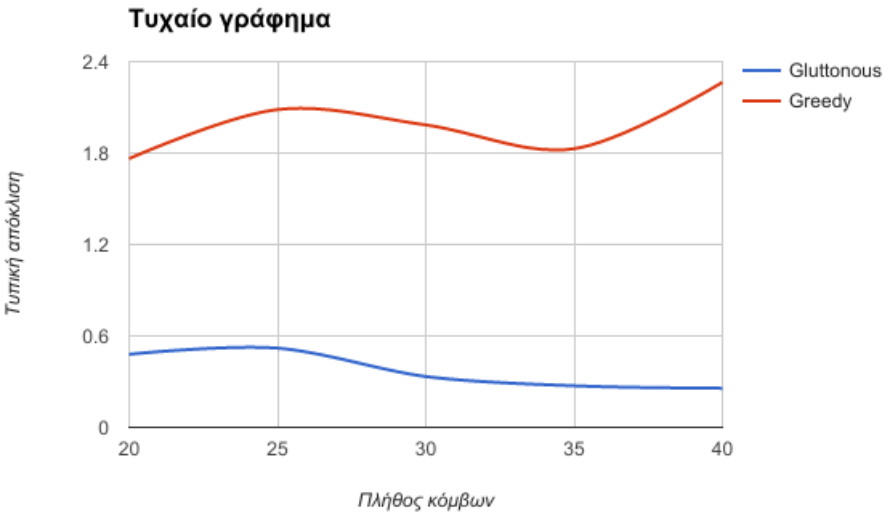
Σχήμα ΙΙ.i: Συνοπτικά τα αποτελέσματα του συνολικού κόστους σε πλήρη γραφήματα.



Σχήμα ΙΙ.ii: Συνοπτικά τα αποτελέσματα της τυπικής απόκλισης σε πλήρη γραφήματα.



Σχήμα II.iii: Συνοπτικά τα αποτελέσματα του συνολικού κόστους σε τυχαία γραφήματα.



Σχήμα II.iv: Συνοπτικά τα αποτελέσματα της τυπικής απόκλισης σε τυχαία γραφήματα.

ΠΑΡΑΡΤΗΜΑ III. ΥΛΟΠΟΙΗΣΗ ΣΕ PYTHON

Listing III.i: graph.py

```
""" Nikos Giachoudis """
# from heapq import *
from priodict import priorityDictionary
from numpy import random
from itertools import combinations


class Graph:
  """
    This is a class which represent a network graph.
    In this class are implemented methods for network graph manipulation.
    Such as getting the neighbours of a node, transform a path (list of
    nodes) into a list of edges on the path, computing a metric closure of a
    graph and others. Check constructor for more info on creating a graph
    object and check other methods to get insight on how to use them.
  """

  @property
  def graph_as_dictionary(self):
    """
    :return: A dictionary representing this graph
    """
    return self._graph_dictionary

  @property
  def nodes(self):
    """
    :return: A list of every node in the graph
    """
    return sorted(self._graph_dictionary.keys())

  @property
  def edges(self):
    """
    A property of returning a list of edges for visual purposes.
    :return: a list of tuples (u, v) representing an edge of the graph.
    """

    if self._graph_dictionary:
      edges = list()
```

```
    for node in self._graph_dictionary:
      for neighbour in self._graph_dictionary[node]:
        if (node, neighbour) not in edges and \
                    (neighbour, node) not in edges:
          edges.append((node, neighbour))

    return edges
  else:
    print("There are no edges.")
    return []

@property
def weights(self):
  """
  A property of returning a dictionary of edge weights for visual purposes.
  :return: dictionary with keys a tuple (u, v) representing an edge and
  value an integer representing the weight of (u, v) edge
  """
  if self._weights_dictionary:
    return self._weights_dictionary
  else:
    print("There are no weights")
    return {}

def __init__(self, edges=None):
  """
  Constructor.
  :param edges: A list of tuples that represent edges of a graph either
  (u, v) or (u, v, w) for weighted graphs.
  :return: a Graph object.
  """
  graph_dict = dict() # dictionary for the adjacency lists
  weights_dict = dict() # dictionary associating each edge with a weight

  # if nothing is given return the empty graph
  if edges:
    nodes = [edge[0] for edge in edges] + [edge[1] for edge in edges]
    nodes = list(set(nodes)) # get rid of duplicates
    for node in nodes:
      for edge in edges:
        # Make an adjacency list representation
        if node == edge[0]:
          graph_dict.setdefault(edge[0], []).append(edge[1])
```

```
            elif node == edge[1]:
              graph_dict.setdefault(edge[1], []).append(edge[0])
            # add weights if there are
            if len(edge) > 2:
              if edge[0] < edge[1]:
                weights_dict[(edge[0], edge[1])] = edge[2]
              else:
                weights_dict[(edge[1], edge[0])] = edge[2]

        for neighbours in graph_dict.values():
          neighbours.sort()

        # copy is for that we don't want a by reference assignment
        self._graph_dictionary = graph_dict.copy()
        self._weights_dictionary = weights_dict.copy()
      else:
        self._graph_dictionary = dict()
        self._weights_dictionary = dict()

    def copy(self, graph):
      """
      A method for making a graph duplicate
      :param graph: copy this graph to the object calling copy
      """
      self._graph_dictionary = graph.graph_as_dictionary.copy()
      self._weights_dictionary = graph.weights.copy()

    def cost_of_edges(self, edges):
      """
      This method computes the sum of the weights on some edges of this graph.
      :param edges: a list of edges.
      :return: the sum of weights of edges.
      """
      total_cost = 0
      for edge in edges:
        if edge not in self.weights.keys():
          u, v = edge
          total_cost += self.weights[(v, u)]
        else:
          total_cost += self.weights[edge]

      return total_cost

    def neighbours(self, node):
```

```
    """
    :param node: A node of the graph
    :return: A list of nodes adjacent to given node
    """
    if node in self._graph_dictionary.keys():
      return self._graph_dictionary[node]
    else:
      print("There is no such node! ", node)
      return None

  def set_weight(self, edge, weight):
    """
    Setting a weight for an edge in the graph. (This does not create new
    edges!)
    :param edge:
    :param weight:
    :return:
    """
    u, v = edge
    if edge in self.weights.keys():
      self.weights[edge] = weight
    else:
      if (v, u) in self.weights.keys():
        self.weights[(v, u)] = weight
      else:
        print("No such edge")

  @classmethod
  def path_to_edges(cls, path):
    """
    A method for getting the edges of a particular path.
    The path is in the form of a list of nodes.
    :param path: A list of nodes, representing a path in the graph.
    :return: A list of (u, v) tuples, representing edges.
    """
    edges = list()
    for i in range(len(path) - 1):
      u = path[i]
      v = path[i + 1]
      if u < v:
        edges.append((u, v))
      else:
        edges.append((v, u))
```

```
    return edges

  def dfs_traversal(self):
    """
    This method returns a new Graph object after traversing the Graph object
    it is called in a depth first fashion.

    For example:
    g = Graph()
    g.make_complete_graph()
    dfs_g = g.dfs_traversal()

    The dfs_g object will be a tree.
    :return: a Graph object representing a tree graph.
    """
    explored = set()
    edges = []

    # checking each node to not miss disconnected components
    for node in self.nodes:

      # if node isn't explored yet we should explore them
      if node not in explored:
        # the actual exploration happening
        explored_nodes, explored_edges = self._explore(node)

        # in each exploration we technically explore a whole continent
        for _ in explored_nodes:
          explored.add(_)

        # update edge accumulator to construct a graph from them
        edges += explored_edges

    # return the actual graph which should not contain any cycles
    return Graph(edges)

  def _explore(self, start):
    """
    Helper method for dfs_traversal() method. Basically an exploration of a
    whole connected component.
    :param start:
    :return:
    """
    visited = list()
```

```
    explored = list()
    edges = list()

    nodes = [start]
    visited.append(start)
    current_node = start
    while visited:
      current_neighbours =\
      [_ for _ in self._graph_dictionary[current_node]]

      next_node = current_neighbours.pop()
      while current_neighbours and (next_node in visited or \
                    next_node in explored):
        next_node = current_neighbours.pop()

      if next_node not in visited and next_node not in explored:
        edges.append((current_node, next_node))
        nodes.append(next_node)
        visited.append(next_node)
        current_node = next_node
      else:
        explored.append(current_node)
        visited.pop()
        if visited:
          current_node = visited[-1]
    return nodes, edges

  def metric_closure(self):
    """
    This method computes a metric closure M of a graph G. A metric closure
    is a complete graph over the set of nodes of G, the weight of each edge
    is determined by the length of the shortest path on G.
    :return: the new metric graph and a dictionary that has edges as keys
    and a list of edges as values representing the shortest path each edge
    corresponds to.
    """
    path_connections = {}
    edges = []
    nodes = self.nodes

    pairs = combinations(nodes, 2)
    for u, v in pairs:
      path_edges = Graph.path_to_edges(self.shortest_path(u, v))
      weight = self.cost_of_edges(path_edges)
```

```
      path_connections.setdefault((u, v), sorted(path_edges))
      edges.append((u, v, weight))

    return Graph(edges), path_connections

  def make_random_weighted_graph(self, num_of_nodes=5, min_w=1, max_w=2):
    """

    :param num_of_nodes:
    :param min_w:
    :param max_w:
    :return:
    """
    if num_of_nodes > 0:
      nodes = range(1, num_of_nodes + 1)
      edges = []
      for u, v in combinations(nodes, 2):
        coin = random.random_integers(0, 1)
        if coin:
          # print (u, v)
          edges.append((u, v))

      for i in range(len(nodes)-1):
        u = nodes[i]
        v = nodes[i+1]
        if (u, v) not in edges and (v, u) not in edges:
          edges.append((u, v))

      weighted_edges = []
      for u, v in edges:
        w = random.random_integers(min_w, max_w)
        weighted_edges.append((u, v, w))

      self.__init__(weighted_edges)

  def make_complete_graph(self, num_of_nodes=5, min_w=1, max_w=2):
    """
    For each combination of node pairs there is an edge. You can call this
    method after instantiating an object.
    For example:
    g = Graph()
    g.make_complete_graph(10, 5, 1000)

    :param num_of_nodes: number of nodes (default 5)
```

```
    :param min_w: minimum edge weight (default 1)
    :param max_w: maximum edge weight (default 2)
    """
    edges = list()
    for i in range(num_of_nodes - 1):
      for j in range(i + 1, num_of_nodes):
        edges.append((i + 1, j + 1,
                random.random_integers(min_w, max_w)))

    self.__init__(edges)
    # return Graph(edges)

  def make_random_weighted_tree(self, num_of_nodes=5, min_w=1, max_w=2):
    """
    This method constructs a tree graph with random weights in [min_w, max_w].
    Algorithm
    1. make a list of all the nodes
    2. pick our first parent from that list
    3. while there are more nodes to put into the graph add them

    Each parent node could have more than one child, the number of which
    depends on coin flips.
    :param num_of_nodes:
    :param min_w:
    :param max_w:
    :return:
    """
    edges = [] # final edges of the graph

    # a list of nodes to populate the graph
    total_nodes = list(range(1, num_of_nodes + 1))

    parent = random.choice(total_nodes) # pick our first parent
    total_nodes.remove(parent) # remove that node from the total

    # and put him into the list of parents without children, since it is our
    # first parent
    parents_without_children = [parent]

    while total_nodes:
      # each parent must have at least one child
      if parent in parents_without_children:
        # pick a random node to be a child
        child = random.choice(total_nodes)
```

```
        total_nodes.remove(child) # remove it from the total

        # this parent now have at least one child
        parents_without_children.remove(parent)
        # this child could be a potential parent
        parents_without_children.append(child)

        # append the new edge between the parent and child node with a
        # random weight
        edges.append((parent, child,
               random.random_integers(min_w, max_w)))

      # check for possibly more children
      coin = random.randint(2)
      if coin and total_nodes: # if heads (1) we add another child
        child = random.choice(total_nodes)
        total_nodes.remove(child)

        # note that we have already removed the parent from the list of
        # parents without children
        parents_without_children.append(child)

        edges.append((parent, child,
               random.random_integers(min_w, max_w)))
      else: # if tails (0) we switch to another parent without children
        parent = random.choice(parents_without_children)

    self.__init__(edges)

  def __str__(self):
    s = ''
    if self.weights != {}:
      for edge in self.edges:
        s += str(edge) + ' ' + str(self.weights[edge]) + '\n'
    else:
      for edge in self.edges:
        s += str(edge) + '\n'

    s += 'Number of edges (m): ' + str(len(self.edges)) + '\n'
    s += 'Number of nodes (n): ' + str(len(self.nodes))

    return s

  # dijkstra's algorithm for shortest paths
```

```
# David Eppstein, UC Irvine, 4 April 2002

# http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/117228

def dijkstra(self, start, end=None):
  """
  Find shortest paths from the start vertex to all vertices nearer than
  or equal to the end.

  The input graph G is assumed to have the following representation:
  A vertex can be any object that can be used as an index into a
  dictionary. G is a dictionary, indexed by vertices. For any vertex v,
  G[v] is itself a dictionary, indexed by the neighbors of v. For any
  edge v->w, G[v][w] is the length of the edge. This is related to the
  representation in <http://www.python.org/doc/essays/graphs.html> where
  Guido van Rossum suggests representing graphs as dictionaries mapping
  vertices to lists of outgoing edges, however dictionaries of edges have
  many advantages over lists: they can store extra information (here, the
  lengths), they support fast existence tests, and they allow easy
  modification of the graph structure by edge insertion and removal.
  Such modifications are not needed here but are important in many
  other graph algorithms. Since dictionaries obey iterator protocol, a
  graph represented as described here could be handed without modification
  to an algorithm expecting Guido's graph representation.

  Of course, G and G[v] need not be actual Python dict objects, they can
  be any other type of object that obeys dict protocol, for instance one
  could use a wrapper in which vertices are URLs of web pages and a call
  to G[v] loads the web page and finds its outgoing links.

  The output is a pair (D,P) where D[v] is the distance from start to v
  and P[v] is the predecessor of v along the shortest path from s to v.

  dijkstra's algorithm is only guaranteed to work correctly when all edge
  lengths are positive. This code does not verify this property for all
  edges (only the edges examined until the end vertex is reached), but
  will correctly compute shortest paths even for some graphs with negative
  edges, and will raise an exception if it discovers that a negative edge
  has caused it to make a mistake.

  Note: There are some changes to the algorithm to fit the representation
  of the graph in this particular class
  """
```

```
    D = {} # dictionary of final distances
    P = {} # dictionary of predecessors
    Q = priorityDictionary() # estimated distances of non-final vertices
    Q[start] = 0

    for v in Q:
      D[v] = Q[v]
      if v == end: break

      for w in self.neighbours(v):
        if (v, w) in self.weights:
          vwLength = D[v] + self.weights[(v, w)]
        elif (w, v) in self.weights:
          vwLength = D[v] + self.weights[(w, v)]
        if w in D:
          if vwLength < D[w]:
            raise (ValueError,
                "dijkstra: found better path to already-final\
                vertex")
        elif w not in Q or vwLength < Q[w]:
          Q[w] = vwLength
          P[w] = v

    return (D, P)

  def shortest_path(self, start, end):
    """
    Find a single shortest path from the given start vertex to the given end
    vertex.
    The input has the same conventions as dijkstra().
    The output is a list of the vertices in order along the shortest path.
    """

    d, p = self.dijkstra(start, end)
    path = []
    while 1:
      path.append(end)
      if end == start: break
      end = p[end]
    path.reverse()
    return path

if __name__ == '__main__':
  """
```

```
  This is the main way for testing quickly if everything works. Someone
  can run the file with python graph.py and any code in this main will be
  executed.
  """
#  g = Graph([(1, 2, 1), (1, 3, 1), (3, 2, 1)])
#  g = Graph([(1, 2), (2, 3), (3, 1), (4, 5), (5, 6), (6, 4)])
#  g = Graph([(1, 2, 1), (2, 3, 1), (3, 1, 1), (4, 1, 1), (5, 1, 1),
#         (6, 2, 1), (7, 2, 1), (8, 3, 1), (9, 3, 1)])
#  print(g)
#  print(g.nodes)
#  print(g.edges)
#  print(g.shortest_path(1, 3))
#  print(g.path_to_edges(g.shortest_path(1, 3)))
#  g = Graph()
#  g.make_random_weighted_tree(8)
#  print(g)
#  m, pConn = g.metric_closure()
#  print(m)
#  print(pConn)
#
#  g.make_random_weighted_graph(5)
#  g.make_complete_graph(10)
#
#  print g
#
#  print g.dfs_traversal()
```

Listing III.ii: paired_greedy.py

```
""" Nikos Giachoudis """
from src.graph import Graph


class PairedGreedy:
  """
  This is a class for the paired greedy method of solving Steiner Forest
  Problem.
  To use this class you need firstly to instantiate an object of this class
  bounded to a particular instance of the problem. Then just run the run()
  method and the results can be found to the results property.
  """

  @property
  def results(self):
    """
```

```
    A property to read the results of the algorithm.
    :return: a dictionary with keys "TotalCost" and "Forest".
    """

    return self.__results

  def __init__(self, graph, demands):
    """
    Constructor.

    :param graph: a Graph object.
    :param demands: a list of pairs. The pairs should consist of integers
            representing nodes that exist in graph.
    """

    # M, edge to shortest paths map
    self.__metric, self.__edge2path = graph.metric_closure()
    self.__graph = graph # G
    self.__demands = demands # D
    self.__results = None

  def run(self):
    """
    The main method, basically the implementation of the paired greedy
    algorithm for the Steiner Forest Problem.
    :return:
    """

    forest = []
    # for each demand in D
    for demand in self.__demands:
      # if there is in M
      if demand in self.__metric.weights:
        # and get the corresponding edges in G
        forest += self.__edge2path[demand]
      # there should be the inverse of demand in M
      else:
        # and get the corresponding edges in G
        forest += self.__edge2path[demand[::-1]]

    forest = list(set(forest)) # removing duplicates

    # instantiate a graph object from the edges of the solution
    g_forest = Graph(forest)
```

```
    # compute a maximal acyclic graph
    g_forest = g_forest.dfs_traversal()

    # compute the total cost of the edges
    total_cost = self.__graph.cost_of_edges(g_forest.edges)

    self.__results = {"TotalCost": total_cost,
             "Forest": g_forest}


if __name__ == '__main__':
  # This is the main way for testing quickly if everything works. Someone
  # can run the file with python paired_greedy.py and any code in this main
  # will be executed.

#  g = Graph()
#  g = Graph([(1, 2, 1), (2, 3, 2), (2, 5, 1), (3, 4, 1), (4, 5, 2)])
#  g = Graph([(1, 2, 1), (2, 3, 1), (3, 1, 1), (4, 1, 1), (5, 1, 1), (6, 2, 1)
    ↪ ,
#         (7, 2, 1), (8, 3, 1), (9, 3, 1)])
#  g.make_random_weighted_graph(10)
#  g.makeWeightedBinaryTree()
#  g.make_random_weighted_tree(8)
#  g.make_complete_graph(10)
#  m, _d = g.metric_closure()
#
#  print g
#
#  grd = PairedGreedy(g, [(4, 9), (6, 5), (7, 8)])
#  grd.run()
#  print grd.results
#  print grd.results['Forest']
#
#  grd = PairedGreedy(g, [(2, 4), (1, 3)])
#  grd.run()
#  print grd.results
#  print grd.results['Forest']
#
#  grd = PairedGreedy(g, [(3, 4), (3, 5)])
#  grd.run()
#  print (grd.results)
```

Listing III.iii: gluttonous.py

```
""" Nikos Giachoudis """
from graph import Graph
from super_node import SuperNode

class Gluttonous:
  """
  This is a class for the gluttonous method of solving Steiner Forest Problem.
  To use this class you need firstly to instantiate an object of this class
  bounded to a particular instance of the problem and give it some demands.
  For example:
  graph = Graph()
  graph.make_random_weighted_graph(10, 1, 2)

  glut = Gluttonous(graph, [(1, 2), (2, 4)])

  # Then just run the run() method and the results can be found to the results
  # property.
  glut.run()

  print(glut.results['Forest'])
  print(glut.results['TotalCost'])
  """

  @property
  def results(self):
    """
    This is a property for accessing the results of the algorithm. If we didn'
    ↪ t
    call run() method yet, results will be None.
    :return: a dictionary with keys ['Forest', 'TotalCost']
    """
    return self.__results

  def __init__(self, graph, demands):
    """
    This is the constructor for instantiate an object for the gluttonous
    ↪ method
    bound on a specific instance of Steiner Forest Problem.
    :param graph: Metric closure graph object of the Graph class.
    :param demands: list with pairs of nodes that need to be connected.
    """
    self.__metric, self.__edge2path = graph.metric_closure()
    self.__graph = graph
    self.__demands = demands
```

```
    self.__results = None

  def run(self):
    """
    This method implements the algorithm. It uses another class called
    ↪ SuperNode
    wich is essentially a group of graph nodes. With merging SuperNode objects
    ↪  it
    will find a solution to a Steiner Forest Problem and write the results to
    ↪ the
    results property.
    """
    clustering = []
    active_supernodes = []
    forest = []
    temp_g = Graph()
    temp_g.copy(self.__metric)

    # begin with trivial clustering
    nodes = set([d[0] for d in self.__demands] + [d[1] for d in self.__demands
    ↪ ])
    for node in nodes:
      new_supernode = SuperNode(node)
      clustering.append(new_supernode)
      active_supernodes.append(new_supernode)

    # main algorithm
    while active_supernodes:

      (S1, S2), path = \
      SuperNode.find_two_closest_active_supernodes(temp_g, active_supernodes)

      # change the metric
      edges = Graph.path_to_edges(path)
      for edge in edges:
        temp_g.set_weight(edge, 0)

      ### UPDATE CLUSTERING ###

      # first remove the old SuperNodes
      clustering.remove(S1)
      clustering.remove(S2)
      active_supernodes.remove(S1)
      active_supernodes.remove(S2)
```

```
      # make a new SuperNode by merging the old ones
      new_S = SuperNode.merge(S1, S2)
      clustering.append(new_S) # add it to the clustering
      if new_S.is_active(self.__demands):
        active_supernodes.append(new_S) # if the new SuperNode is active add
                        # it to the active ones

      ### UPDATE FOREST ###
      for edge in Graph.path_to_edges(path):
        if edge not in self.__edge2path:
          print "!!!SOMETHING WENT WRONG!!!"
        forest += self.__edge2path[edge]

      # forest += self.__graph.path_to_edges(path)

    # clean forest
    clean_forest = list(set(forest))
    # g_forest = clean_forest
    # total_cost = self.__graph.cost_of_edges(g_forest)
    g_forest = Graph(clean_forest)
    total_cost = self.__graph.cost_of_edges(g_forest.edges)

    self.__results = {'TotalCost': total_cost,
             'Forest': g_forest}

if __name__ == '__main__':
  """
  This is the main way for testing quickly if everything works. Someone
  can run the file with python gluttonous.py and any code in this main will be
  executed.
  """
  g = Graph()
  # g = Graph([(1, 2, 1), (2, 3, 2), (2, 5, 1), (3, 4, 1), (4, 5, 2)])
  # g = Graph([(1, 2, 1), (2, 3, 1), (3, 1, 1), (4, 1, 1), (5, 1, 1), (6, 2,
    ↪ 1),
  #      (7, 2, 1), (8, 3, 1), (9, 3, 1)])
  g.make_random_weighted_graph(100, 1, 1000)
  # g.makeWeightedBinaryTree()
  # g = Graph([(1, 2, 1), (1, 3, 1), (2, 4, 1), (2, 5, 1)])

  # print g

  glt = Gluttonous(g, [(4, 9), (6, 5), (7, 8)])
```

```
  glt.run()
  print glt.results
  print glt.results['Forest']

  # glut = Gluttonous(g, [(2, 4), (1, 3)])
  # glut.run()
  # print(glut.results)
  #
  # glut = Gluttonous(g, [(3, 4), (3, 5)])
  # glut.run()
  # print(glut.results)
```

Listing III.iv: super_node.py

```
""" Nikos Giachoudis """
from graph import Graph


class SuperNode(object):
  """
  A class for making supernodes. A SuperNode is a cluster of nodes from graph
   ↪ G.
  """

  @property
  def cluster(self):
    return self.__cluster

  def __init__(self, node=None):
    """
    Constructor. Creates a super node with at most one node.

    :param node: Default value is None but someone can give a node (number)
   ↪ that
          will be included into the super node.
    """
    if node is not None:
      self.__cluster = [node]
    else:
      self.__cluster = []

  def __len__(self):
    """
    :return: An integer representing how many nodes there are in this super
   ↪ node.
```

```
    """
    return len(self.cluster)

  def __str__(self):
    return str(self.__cluster)

  def add_nodes(self, nodes):
    """
    A method for adding nodes to the current super node.

    :param nodes: a list of nodes.
    """
    for node in nodes:
      if node not in self.__cluster:
        self.__cluster.append(node)

  def is_active(self, demands):
    """
    A method testing if this super node is active or not. The definition of an
    active super node is as follows:
    For each terminal node u in this super node if the mate v does not belong
    ↪ to
    this super node then the super node is active, otherwise it is not.

    :param demands: a list of tuples showing the couples.
    :return: True or False
    """
    for u, v in demands:
      if u in self.__cluster and v not in self.__cluster or\
        u not in self.__cluster and v in self.__cluster:
        return True
    return False

  @classmethod
  def merge(cls, S1, S2):
    """
    A static method for merging super nodes.
    :param S1: a super node
    :param S2: another super node
    :return: merged super node
    """
    s = SuperNode()
    s.add_nodes(S1.cluster)
    s.add_nodes(S2.cluster)
```

```
    return s

  @classmethod
  def find_two_closest_active_supernodes(cls, graph, active_supernodes):
    min_distance = 999999 # TODO better initialization of min distance
    min_path = []
    closest_supernodes = ()
    for i in range(len(active_supernodes) - 1):
      for j in range(i+1, len(active_supernodes)):
        for u in active_supernodes[i].cluster:
          for v in active_supernodes[j].cluster:
            # using Dijkstra
            path = graph.shortest_path(u, v)
            dist = graph.cost_of_edges(Graph.path_to_edges(path))

            # using breadth first search
            # dist, path = graph.bfs_shortest(u, v)

            # breaking ties consistently
            if dist < min_distance:
              min_distance = dist
              min_path = path
              closest_supernodes = (active_supernodes[i],
                          active_supernodes[j])
            elif dist == min_distance\
               and (active_supernodes[i].cluster,
                 active_supernodes[j].cluster) <\
                   (closest_supernodes[0].cluster,
                    closest_supernodes[1].cluster):

              min_distance = dist
              min_path = path
              closest_supernodes = (active_supernodes[i],
                          active_supernodes[j])

    return closest_supernodes, min_path
```

Listing III.v: steinerf.mod

```
param file, symbolic := "out.txt";
param gl;
/* gluttonous solution */

param g;
```

```
/* greedy solution */

param n, integer, >= 2;
/* number of nodes */

param k, integer, >= 1;
/* number of demands */

set V := {1..n};
/* set of nodes */

set E, within V cross V;
/* set of edges */

check{(i,j) in E}: i < j;
/* no loops, every edge is indexed only once */

set EE := setof{(i,j) in E} (i,j) union setof{(i,j) in E} (j,i);
/* set of directed flow edges */

set D := {1..k};
param orig{d in D}, integer, >=1, <=n;
param dest{d in D}, integer, >=1, <=n;

param c{(i,j) in E};
/* edge costs */

var x{(i,j) in E}, binary;
/* edge variables */

var f{(i,j) in EE, d in D}, binary;
/* flow variables */

minimize obj: sum{(i,j) in E} c[i,j]*x[i,j];
s.t. flow{i in V, d in D}:
 sum{(j,i) in EE} f[j,i,d]
 + (if i=orig[d] then 1)
 =
 sum{(i,j) in EE} f[i,j,d]
 + (if i=dest[d] then 1);

s.t. bound: sum{(i,j) in E} c[i,j]*x[i,j] <= g;

s.t. edge1{(i,j) in E, d in D}:
```

```
 x[i,j] >= f[i,j,d];

s.t. edge2{(i,j) in E, d in D}:
 x[i,j] >= f[j,i,d];

solve;

printf "Number of nodes = %d, Number of demand pairs = %d\n", n, k;
printf "Optimum cost = %f, Greedy Cost = %f, Gluttonous Cost = %f\n",obj.val,
    ↪ g, gl;
printf "Aproximation ratios: Greedy Cost = %f, Gluttonous Cost = %f\n\n",g/
    ↪ obj.val,gl/obj.val;

printf "%f,%f\n",g/obj.val,gl/obj.val >> file;

end;
```

ΑΝΑΦΟΡΕΣ

[1] Ajit Agrawal, Philip Klein, and R Ravi. When trees collide: An approximation algorithm for the generalized steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, 1995.

[2] Baruch Awerbuch, Yossi Azar, and Yair Bartal. On-line generalized steiner problem. *Theoretical Computer Science*, 324(2-3):313–324, 2004.

[3] Norman Biggs. Constructions for cubic graphs with large girth. *The Electronic Journal of Combinatorics*, 5(1):A1, 1998.

[4] Glencora Borradaile, Philip Klein, and Claire Mathieu. An o (n log n) approximation scheme for steiner tree in planar graphs. *ACM Transactions on Algorithms (TALG)*, 5(3):31, 2009.

[5] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An improved lp-based approximation for steiner tree. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 583–592. ACM, 2010.

[6] Ho-Lin Chen, Tim Roughgarden, and Gregory Valiant. Designing network protocols for good equilibria. *SIAM Journal on Computing*, 39(5):1799–1832, 2010.

[7] Xiuzhen Cheng and Ding-Zhu Du. *Steiner trees in industry*, volume 11. Springer Science & Business Media, 2013.

[8] Miroslav Chlebík and Janka Chlebíková. The steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207–214, 2008.

[9] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] Ranel E Erickson, Clyde L Monma, and Arthur F Veinott Jr. Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research*, 12(4):634–664, 1987.

[12] Michel X Goemans and David P Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[13] Anupam Gupta and Amit Kumar. Greedy algorithms for steiner forest. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 871–878. ACM, 2015.

[14] Mathias Hauptmann and Marek Karpiński. *A compendium on steiner tree problems*. Inst. für Informatik, 2013.

[15] Stefan Hougardy, Jannik Silvanus, and Jens Vygen. Dijkstra meets steiner: a fast exact goal-oriented steiner tree algorithm. *Mathematical Programming Computation*, pages 1–68, 2014.

[16] Edmund Ihler, Gabriele Reich, and Peter Widmayer. Class steiner trees and vlsi-design. *Discrete Applied Mathematics*, 90(1-3):173–194, 1999.

[17] Makoto Imase and Bernard M Waxman. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[18] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[19] Chin Lung Lu, Chuan Yi Tang, and Richard Chia-Tung Lee. The full steiner tree problem. *Theoretical Computer Science*, 306(1-3):55–67, 2003.

[20] Mark Manasse, Lyle McGeoch, and Daniel Sleator. Competitive algorithms for on-line problems. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 322–333. ACM, 1988.

[21] Vangelis Th Paschos, Orestis A Telelis, and Vassilis Zissimopoulos. Steiner forests on stochastic metric graphs. In *International Conference on Combinatorial Optimization and Applications*, pages 112–123. Springer, 2007.

[22] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6):1389–1401, 1957.

[23] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[24] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.