ΤΕΙ ΗΠΕΙΡΟΥ
ΤΜΗΜΑ ΤΗΛΕΠΛΗΡΟΦΟΡΙΚΗΣ & ΔΙΟΙΚΗΣΗΣ

ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΔΡΟΜΟΛΟΓΗΣΗ ΚΑΙ ΠΟΛΥ-ΧΡΩΜΑΤΙΣΜΟ ΜΟΝΟΠΑΤΙΩΝ ΣΕ ΓΡΑΦΗΜΑΤΑ

ΕΙΣΗΓΗΤΗΣ ΚΑΘΗΓΗΤΗΣ
ΧΑΡΙΛΟΓΗΣ ΒΑΣΙΛΕΙΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΤΟΥ
ΚΑΡΑΓΕΩΡΓΟΥ ΤΙΜΟΘΕΟΥ
Α.Μ. 1026

ΑΡΤΑ 2006

Περιεχόμενα

Πρόλογος.

Στην εργασία αυτή υλοποιούνται αλγόριθμοι για δρομολόγηση και πολυ-χρωματισμό μονοπατιών σε γραφήματα. Ασχολούμαστε με συγκεκριμένες κατηγορίες γραφημάτων, όπως είναι οι αλυσίδες, οι δακτύλιοι και οι αστέρες.

Η μέθοδος στην οποία στηρίζονται οι αλγόριθμοι τους οποίους υλοποιήσαμε είναι η μετατροπή της εισόδου σε διμερές γράφημα και στη συνέχεια ο χρωματισμός του διμερούς γραφήματος που προκύπτει. Επομένως, πρέπει να υλοποιηθεί και ένας αλγόριθμος χρωματισμού μονοπατιών σε διμερή γραφήματα.

Η είσοδος του προγράμματος είναι ο αριθμός τον διαθέσιμων χρωμάτων που έχουμε για τον χρωματισμό, ο αριθμός των κόμβων που έχει το γράφημά μας, ο αριθμός των μονοπατιών και τα μονοπάτια που θέλουμε να χρωματίσουμε.

Στη συγκεκριμένη εργασία έχουν υλοποιηθεί οι αλγόριθμοι για δρομολόγηση και πολυ-χρωματισμό σε γραφήματα, που έχουν προταθεί από τους C. Nomikos, A. Pagourgis και S. Zachos [17], για αλυσίδες, δακτυλίους και αστέρες. Για τον χρωματισμό ακμών σε διμερή γραφήματα έχει υλοποιηθεί ο αλγόριθμος που προτάθηκε από τους R. Cole και J. Hopcroft [3].

Ο αλγόριθμος δρομολόγησης και πολυ-χρωματισμού σε αλυσίδες ταυτίζεται με τον αλγόριθμο πολυ-χρωματισμού σε αλυσίδες, καθώς στις αλυσίδες δεν έχουμε δρομολόγηση. Ο αλγόριθμος αποτελείται από έξη βήματα. Από αυτά τα τέσσερα πρώτα αποτελούν τους μετασχηματισμούς για καταλήξουμε σε διμερές γράφημα, το πέμπτο βήμα αποτελεί τον χρωματισμό του διμερούς γραφήματος και τέλος το έκτο βήμα είναι η ανάθεση των χρωμάτων στα αρχικά μονοπάτια.

Για τους δακτυλίους τα πράγματα είναι λίγο διαφορετικά, εφόσον μπορούμε να έχουμε δρομολόγηση. Έχουμε, δηλαδή, τον αλγόριθμο για πολυ-χρωματισμό και τον αλγόριθμο για δρομολόγηση και πολυ-χρωματισμό. Στον αλγόριθμο για δρομολόγηση και πολυ-χρωματισμό απλώς κάνουμε μια δρομολόγηση των ελαχίστων μονοπατιών.

Βρίσκουμε, δηλαδή, πιο μονοπάτι είναι πιο μικρό ανάμεσα σε δύο εναλλακτικά μονοπάτια. Για παράδειγμα, έχουμε το μονοπάτι {0,4}, το οποίο το συγκρίνουμε με το {4,0}, και κρατάμε αυτό που είναι πιο μικρό. Στη συνέχεια χρησιμοποιούμε τον αλγόριθμο του πολυ-χρωματισμού για δακτυλίους. Στον αλγόριθμο πολυ-χρωματισμού μετατρέπουμε τον δακτύλιο σε αλυσίδα και λύνουμε το πρόβλημα του πολυ-χρωματισμού σε αλυσίδες και έπειτα χρωματίζουμε τα αντίστοιχα μονοπάτια του δακτυλίου.

Τέλος, για τους αστέρες τα πράγματα είναι λίγο πολύ ίδια με τις αλυσίδες. Έτσι, πρέπει με κάποιους μετασχηματισμούς –όχι τόσο πολύπλοκους όσο στις αλυσίδες- να καταλήξουμε σε ένα διμερές γράφημα, το οποίο χρωματίζεται, και στη συνέχεια αναθέτουμε τα αντίστοιχα χρώματα στα μονοπάτια του αστέρα.

Καταλήγοντας θα πρέπει να αναφέρουμε ότι, ενώ το πρόβλημα για τις αλυσίδες λύνεται βέλτιστα, το πρόβλημα για τους δακτυλίους και τους αστέρες δεν λύνεται βέλτιστα, αλλά με ένα προσεγγιστικό παράγοντα δύο. Αυτό συμβαίνει, γιατί τα δύο τελευταία προβλήματα είναι NP-δύσκολα.

1. Εισαγωγή.

1.1 Εισαγωγή στα αμιγώς οπτικά δίκτυα.

Στόχος της εργασίας αυτής είναι να υλοποιηθούν αλγόριθμοι για την επίλυση προβλημάτων στα αμιγώς οπτικά δίκτυα. Ωστόσο θα ήταν χρήσιμο, πριν ασχοληθούμε αναλυτικά με το θέμα μας, να αναφερθούμε με συντομία στα αμιγώς οπτικά δίκτυα (all-optical networks).

Το πεδίο της οπτικής μετάδοσης είναι νέο και δυναμικό (ακόμη αναπτύσσεται). Έχουν περάσει μόνο 30 χρόνια από τότε που ο Charls Kao αντιλήφθηκε ότι μπορούσε κανείς να στέλνει υψίρρυθμα φωτεινά μηνύματα από ένα λεπτό γυάλινο νήμα. Σχεδόν την ίδια χρονική περίοδο εφευρέθηκε και το laser διόδου ημιαγωγού. Η εξασθένηση του φωτός στα πρώτα πειράματα του Kao ήταν ισχυρότατη, περίπου 1 dB ανά 30cm, και η φωτεινή πηγή ήταν μια πρωτόγονη δίοδος φωτο-εκπομπής (LED) στο ορατό φως. Σήμερα οι οπτικές ίνες που έχουν εγκατασταθεί παρουσιάζουν εξασθένηση 1dB ανά πέντε χιλιόμετρα μήκους και υπάρχουν εμπορικά διαθέσιμες δίοδοι laser στο εγγύς υπέρυθρο που μπορούν να διαμορφωθούν σε ρυθμούς που υπερβαίνουν αρκετά Gbits ανά sec.

Πρέπει να πούμε ότι, μέχρι στιγμής, υπάρχουν τρεις γενεές δικτύων ανάλογα με τη χρήση οπτικών ινών. Τα αμιγώς οπτικά δίκτυα αποτελούν την τρίτη γενεά δικτύων. Στην πρώτη γενεά δικτύων δεν χρησιμοποιήθηκαν καθόλου οπτικές ίνες. Στην δεύτερη γενεά χρησιμοποιήθηκε σαν υποκατάστατο του χαλκού. Έτσι τα μεγάλων αποστάσεων καλώδια αντικαταστάθηκαν από οπτικές ίνες, όχι όμως και οι καλωδιώσεις στις οποίες καταλήγουν τα σήματα εξαιτίας του τεράστιου κόστους κατασκευής συσκευών, οι οποίες έχουν ενσωματωμένες οπτικές ίνες. Έτσι τα σήματα, που μεταδίδονταν, έφευγαν από μία

συσκευή πομπό ενός δικτύου δεύτερης γενεάς, μετατρέπονταν σε οπτικό σήμα, το οποίο μεταδιδόταν ταχύτατα χωρίς την ανάγκη ενισχυτή η αναμεταδότη εξαιτίας του μικρού σφάλματος (είναι χαρακτηριστικό ότι ενώ με τα καλώδια χαλκού χρειάζεται ενισχυτής σήματος ανά 3-7km, το σήμα που διέρχεται από οπτική ίνα χρειάζεται ενίσχυση στα 60-80km) και στη συνέχεια μετατρέπονταν πάλι σε ηλεκτρικό για να καταλήξει στην συσκευή δέκτη. Η διαφορά των δικτύων τρίτης γενεάς από τα δίκτυα δεύτερης είναι ότι υπάρχει ένας ελάχιστος αριθμός ηλεκτρο-οπτικών και οπτικο-ηλεκτρικών μετατροπών. Επίσης μία άλλη πολύ σημαντική διαφορά είναι ότι στα δίκτυα δεύτερης γενεάς ο κόμβος από τον οποίο περνούσε ένα σήμα διακινεί πληροφορία για κάποιον άλλο κόμβο (δηλαδή το σήμα στέλνεται σε έναν κόμβο κι από εκεί στέλνεται σε έναν άλλο). Ενώ στα αμιγώς οπτικά δίκτυα (τρίτη γενιά) κανείς κόμβος δεν είναι υποχρεωμένος να διακινεί πληροφορία για λογαριασμό άλλων κόμβων.

Η από σημείο σε σημείο απόσταση, που καλύπτουν τα ήδη εγκατεστημένα οπτικά συστήματα, κυμαίνεται από πολύ μικρές αποστάσεις μέσα στον υπολογιστή ή στο δωμάτιο υπολογιστών μέχρι διηπειρωτικές συνδέσεις με υποβρύχια καλώδια. Από τα μέσα της δεκαετίας του 60 η ιστορία των οπτικών επικοινωνιών σημείωσε μια διαρκή πρόοδο σε πολλά μέτωπα. Η μία εφεύρεση μετά την άλλη προώθησαν τις οπτικές ίνες στη σημερινή τους θέση ως επίλεκτης τεχνολογίας για όλα τα συστήματα επικοινωνίας που χρησιμοποιούν σταθερές διαδρομές μήκους πάνω από μερικά μέτρα. Μπορεί κανείς, σύμφωνα με τον P.E. Green [8], να διακρίνει την ανάδειξη δύο κατευθύνσεων στο μελλοντικό κόσμο των επικοινωνιών: μία πρώτη όπου ουσιαστικά όλες οι επικοινωνίες σταθερών σημείων θα γίνονται μάλλον με οπτικές ίνες παρά με χάλκινα σύρματα, ενώ, για να συνεργαστεί με τη βασική αυτή υποδομή, θα υπάρχει ένας δεύτερος κόσμος "αιθέρων", δηλαδή της κινητής ραδιομετάδοσης και των υπέρυθρων επικοινωνιών, που θα εμπλέκει τηλεφωνία και τερματικά τοποθετημένα στο αυτοκίνητο, το αεροπλάνο, το πλοίο, ή τα κινητά τηλέφωνα.

1.2 Στοιχεία θεωρίας γραφημάτων – Ορισμοί.

Μετά από αυτή την πολύ μικρή εισαγωγή στις οπτικές ίνες και τα οπτικά δίκτυα και πριν περάσουμε στην περαιτέρω ανάλυση, θα πρέπει να δώσουμε ορισμένα βασικά στοιχεία της θεωρίας γραφημάτων.

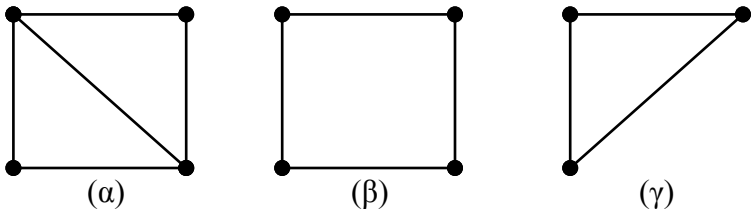
Γράφημα ή γράφος ονομάζεται ένα ζεύγος (V, E), όπου V είναι ένα μη κενό σύνολο κορυφών του γραφήματος και E είναι ένα σύνολο διμελών υποσυνόλων του V, το οποίο μπορεί να είναι και κενό.

Η ακμή e = (υ, u) λέμε ότι ενώνει τις κορυφές υ και u. Οι κορυφές υ και u ονομάζονται άκρα της ακμής e. Η ακμή e λέμε ότι προσπίπτει στις υ, u. Ο αριθμός των ακμών που προσπίπτουν σε μια κορυφή υ λέγεται βαθμός της υ. Ο μέγιστος βαθμός από όλες τις κορυφές ενός γραφήματος ονομάζεται βαθμός του γραφήματος.

Σε ένα γράφημα G = (V, E), ονομάζουμε μονοπάτι μεταξύ δύο κορυφών u και υ μία ακολουθία ακμών $\{u, \upsilon_1\}, \{\upsilon_1, \upsilon_2\}, \ldots, \{\upsilon_{n-1}, \upsilon\}$ που είναι διαφορετικές μεταξύ τους. Λέμε ότι το μονοπάτι συνδέει ή ενώνει τις κορυφές u και υ. Επίσης λέμε ότι το μονοπάτι περιέχει, χρησιμοποιεί ή περνάει από τις κορυφές $\upsilon_0, \upsilon_1, \ldots, \upsilon_n$. Το πλήθος των ακμών που περιέχονται σε ένα μονοπάτι ονομάζεται μήκος μονοπατιού. Ένα μονοπάτι περιγράφεται πλήρως από την ακολουθία κορυφών $\upsilon_0, \upsilon_1, \ldots, \upsilon_n$.

Σε ένα γράφημα G = (V, E) ονομάζουμε κύκλο μία ακολουθία ακμών $\{\upsilon_0, \upsilon_1\}, \{\upsilon_1, \upsilon_2\}, \ldots, \{\upsilon_{n-1}, \upsilon_n\}, \{\upsilon_n, \upsilon_0\}$ τέτοια ώστε οι κορυφές $\upsilon_0, \upsilon_1, \upsilon_2, \ldots, \upsilon_n$, να είναι διαφορετικές μεταξύ τους, δηλαδή κύκλος είναι ένα κλειστό μονοπάτι. Το μήκος ενός κύκλου είναι ίσο με το πλήθος των ακμών που περιέχει. Είναι φανερό ότι για να έχουμε κύκλο πρέπει να έχουμε τουλάχιστο τρεις ακμές στο γράφημα, δηλαδή το ελάχιστο μήκος ενός κύκλου είναι τρία (3).

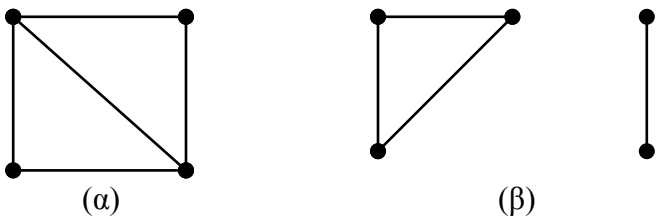
Υπογράφημα ενός γραφήματος G = (V, E) ονομάζεται ένα γράφημα G’ = (V’, E’), αν και μόνο αν το V’ είναι υποσύνολο του V και το E’ υποσύνολο του E. Αν V’ = V, τότε G’ = (V’, E’) ονομάζεται παράγον υπογράφημα του G = (V, E) με σύνολο κορυφών το V’ και με το μέγιστο δυνατό σύνολο ακμών, δηλαδή αυτό που περιέχει όλες τις ακμές του E με άκρα στο V’.



Σχήμα 1.1. (α) Το γράφημα G. (β). Ένα υπογράφημα του G με 4 κορυφές.
(γ). Ένα υπογράφημα του G με τρεις κορυφές.

Αν σε ένα μονοπάτι μεταξύ u και υ αντιστρέψουμε την φορά των ακμών παίρνουμε ένα μονοπάτι μεταξύ υ και u. Επιπλέον, από δύο μονοπάτια, ένα μεταξύ u και w και ένα μεταξύ w και υ, μπορούμε να κατασκευάσουμε ένα μονοπάτι μεταξύ του u και του υ παραθέτοντας τις δύο ακολουθίες ακμών και απαλείφοντας τον κύκλο που πιθανά δημιουργείται. Τέλος μπορούμε να θεωρήσουμε ότι κάθε κορυφή υ ενώνεται με τον εαυτό της με ένα μονοπάτι μήκους 0. Άρα η σχέση "συνδέεται" είναι ανακλαστική, συμμετρική, μεταβατική και χωρίζει τις κορυφές του γραφήματος σε κλάσεις ισοδυναμίας. Κορυφές που ανήκουν στην ίδια κλάση συνδέονται με κάποιο μονοπάτι, ενώ κορυφές σε διαφορετικές κλάσεις δεν συνδέονται. Συνεπώς τα δύο άκρα κάθε ακμής βρίσκονται στην ίδια κλάση ισοδυναμίας. Ονομάζουμε συνεκτικές συνιστώσες ενός γραφήματος G τα υπογραφήματά του που παράγονται από τις κλάσεις ισοδυναμίας της σχέσης "συνδέεται".

Ένα γράφημα ονομάζεται συνεκτικό, αν αποτελείται από μία συνεκτική συνιστώσα. Σε ένα συνεκτικό γράφημα υπάρχει μονοπάτι ανάμεσα σε οποιοδήποτε ζεύγος κορυφών.

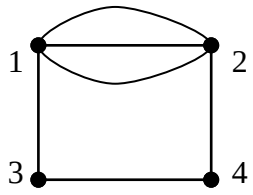


Σχήμα 1.1. (α). Συνεκτικό γράφημα με τέσσερις κορυφές. (β). Μη συνεκτικό
γράφημα με πέντε κορυφές.

Ένα γράφημα ονομάζεται διμερές αν οι κορυφές του μπορούν να χωριστούν σε δύο σύνολα V_1 και V_2 έτσι ώστε κάθε ακμή να ενώνει μία κορυφή του V_1 με μία του V_2.

Σε ένα γράφημα G = (V, E) ονομάζουμε ταίριασμα ένα υπογράφημά του στο οποίο κάθε κορυφή έχει βαθμό 1 και ονομάζουμε ανεξάρτητο σύνολο ένα υποσύνολο του V που δεν περιέχει γειτονικές κορυφές.

Σε ένα πολυγράφημα επιτρέπεται να έχουμε περισσότερες από μία ακμές ανάμεσα σε δύο κορυφές υ και u. Το πλήθος των παράλληλων ακμών {u, υ} ονομάζεται πολυπλοκότητα της ακμής {u, υ}. Για την περιγραφή ενός μονοπατιού σε ένα



Σχήμα 1.2 Πολυγράφημα: ανάμεσα στις κορυφές 1 και 2 έχουμε τρεις ακμές.

πολυγράφημα δεν αρκεί η απαρίθμηση των κορυφών του. Περισσότερες πληροφορίες για τη Θεωρία γραφημάτων μπορούν να αναζητηθούν στον Berge [2].

1.3 Η έννοια του χρωματισμού

Χρωματισμός ενός συνόλου αντικειμένων S είναι μια συνάρτηση από το S σε ένα σύνολο χρωμάτων C (δηλαδή η ανάθεση ενός χρώματος σε κάθε αντικείμενο), η οποία πληροί κάποιους περιορισμούς.

Σε χρωματισμό με απαγορευμένα υποσύνολα οι περιορισμοί έχουν την παρακάτω μορφή: δίνεται ένα υποσύνολο R του δυναμοσυνόλου του S και θέλουμε τα στοιχεία κάθε συνόλου στο R να μην είναι χρωματισμένα με το ίδιο χρώμα.

Σε αυτή την περίπτωση αν εναλλάξουμε τη χρήση δύο χρωμάτων, αντικαταστήσουμε ένα χρώμα με ένα νέο ή γενικότερα μεταθέσουμε τυχαία τα χρώματα, ο νέος χρωματισμός ικανοποιεί επίσης τους ίδιους περιορισμούς. Δηλαδή η ανάθεση ενός χρώματος σε ένα αντικείμενο δεν έχει καμία άλλη σημασία πέρα από το ότι σχετίζει το

αντικείμενο αυτό με τα άλλα αντικείμενα του ίδιου χρώματος. Άρα ο χρωματισμός ορίζει μία διαμέριση S_1, S_2, του S_k του συνόλου S τέτοια ώστε κανένα στοιχείο του R να μην είναι υποσύνολο κάποιου S_i. Αντίστροφα κάθε τέτοια διαμέριση μπορεί να χρησιμοποιηθεί για την κατασκευή χρωματισμών (χρησιμοποιούμε k διακριτά χρώματα, για κάθε S_i).

Στη συνέχεια θα ασχοληθούμε με χρωματισμούς στους οποίους τα απαγορευμένα υποσύνολα είναι ζεύγη στοιχείων του S. Το πρόβλημα χρωματισμού των στοιχείων του ενός συνόλου με απαγορευμένα ζευγάρια μπορεί να διατυπωθεί γραφοθεωρητικά: δίνεται ένα γράφημα (S, R) και ζητείται να χρωματιστούν οι κορυφές του έτσι ώστε γειτονικές κορυφές να έχουν διαφορετικά χρώματα.

Άρα ο χρωματισμός ενός συνόλου S με απαγορευμένα ζευγάρια ανάγεται στο χρωματισμό των κορυφών ενός γραφήματος. Τα υποσύνολα στα οποία διαμερίζεται το σύνολο των κορυφών του γραφήματος από τον χρωματισμό είναι ανεξάρτητα σύνολα (αφού δεν υπάρχει καμία ακμή ανάμεσα σε κορυφές με το ίδιο χρώμα).

Ένα άλλο πρόβλημα χρωματισμού σε γράφημα είναι ο χρωματισμός του συνόλου των ακμών του, έτσι ώστε ακμές που προσπίπτουν στην ίδια κορυφή να έχουν διαφορετικό χρώμα. Τα υποσύνολα στα οποία διαμερίζεται το σύνολο των ακμών του γραφήματος από τον χρωματισμό είναι ταιριάσματα (αφού δεν υπάρχει καμία κοινή κορυφή ανάμεσα σε ακμές με το ίδιο χρώμα).

Ο χρωματισμός των ακμών ενός γραφήματος είναι χρωματισμός με απαγορευμένα ζευγάρια και άρα μπορεί να αναχθεί στο χρωματισμό των κορυφών ενός γραφήματος. Η αντίθετη αναγωγή δεν είναι δυνατή, αφού π.χ. ο χρωματισμός των ακμών του {{1,2,3,4}, {{1, 2}, {1, 3}, {1, 4}}} δεν μπορεί να αναχθεί σε χρωματισμό ακμών κάποιου γραφήματος. Η αντίθετη αναγωγή δεν είναι δυνατή ακόμη κι αν χρησιμοποιήσουμε πολυγράφημα. Άρα ο χρωματισμός ακμών δεν μπορεί να περιγράψει ένα οποιοδήποτε πρόβλημα χρωματισμού με απαγορευμένα ζευγάρια.

1.4 Βέλτιστοι χρωματισμοί

Ένας προφανής τρόπος να χρωματίσουμε τις κορυφές ενός γραφήματος (ώστε γειτονικές κορυφές να έχουν διαφορετικά χρώματα) είναι να χρησιμοποιήσουμε ένα χρώμα για κάθε κορυφή. Ενδιαφερόμαστε όμως για χρωματισμούς που χρησιμοποιούν

λιγότερα χρώματα. Δηλαδή, θέλουμε να διαμερίσουμε το σύνολο των κορυφών σε ανεξάρτητα σύνολα που έχουν περισσότερα από ένα στοιχεία.

Ονομάζουμε χρωματικό αριθμό ενός γραφήματος τον ελάχιστο αριθμό χρωμάτων με τον οποίο μπορούν να χρωματιστούν οι κορυφές του έτσι ώστε γειτονικές κορυφές να έχουν διαφορετικά χρώματα. Ένας ισοδύναμος με τον παραπάνω ορισμός είναι ο εξής: χρωματικός αριθμός ενός γραφήματος είναι ο ελάχιστος αριθμός ανεξάρτητων συνόλων που μπορούμε να διαμερίσουμε τις κορυφές του γραφήματος.

Το πρόβλημα του προσδιορισμού του χρωματικού αριθμού (GC) είναι υπολογιστικά δύσκολο. Συγκεκριμένα το αντίστοιχο πρόβλημα απόφασης είναι NP-πλήρες [12], ενώ το πρόβλημα βελτιστοποίησης είναι μη προσεγγίσιμο [14].

Ονομάζουμε χρωματικό δείκτη ενός γραφήματος τον ελάχιστο αριθμό χρωμάτων με τον οποίο μπορούν να χρωματιστούν οι ακμές του έτσι ώστε οι ακμές που προσπίπτουν στην ίδια κορυφή να έχουν διαφορετικά χρώματα. Ένας ισοδύναμος ορισμός είναι ο εξής: χρωματικός δείκτης ενός γραφήματος είναι ο ελάχιστος αριθμός ταιριασμάτων που μπορούμε να διαμερίσουμε τις ακμές του γραφήματος.

Το πρόβλημα του χρωματικού δείκτη ενός γραφήματος (EC) μπορεί να λυθεί προσεγγιστικά με παράγοντα προσέγγισης 4/3 [9]. Το αντίστοιχο πρόβλημα απόφασης είναι NP-πλήρες [10].

1.5 Εισαγωγή στο πρόβλημα πολύ-χρωματισμού.

Στα οπτικά δίκτυα είναι απαραίτητο να βελτιστοποιήσουμε την χρήση των διαθέσιμων συχνοτήτων. Με δοσμένες κάποιες απαιτήσεις ο στόχος είναι να τις ικανοποιήσουμε με χρήση ενός ελάχιστου αριθμού συχνοτήτων. Έστω ότι θέλουμε να ικανοποιήσουμε ένα σύνολο συνδιαλέξεων ανάμεσα σε κόμβους οπτικού δικτύου. Η σύνδεση δύο κόμβων που θέλουν να επικοινωνήσουν γίνεται μέσω μιας ακολουθίας φυσικών συνδέσμων (οπτικών ινών). Μέσα από την ίδια οπτική ίνα μπορούν να περάσουν περισσότερες από μία συνδιαλέξεις, πρέπει όμως κάθε μια από αυτές να χρησιμοποιεί διαφορετική οπτική συχνότητα. Η κάθε συνδιάλεξη πρέπει να περάσει από όλες τις οπτικές ίνες μέσω της ίδιας συχνότητας. Αυτό οφείλεται στο γεγονός ότι αλλαγή συχνότητας δεν μπορεί να γίνει μόνο με οπτικά μέσα. Ο αριθμός των διαθέσιμων συχνοτήτων περιορίζεται από την τεχνολογία, παρότι η οπτική ίνα σαν μέσο έχει

τεράστια χωρητικότητα. Από τα παραπάνω γίνεται φανερό γιατί πρέπει να γίνεται βέλτιστη χρήση των συχνοτήτων. Το πρόβλημα αυτό μπορεί να αναπαρασταθεί ως πρόβλημα χρωματισμού μονοπατιών. Η τοπολογία του δικτύου περιγράφεται από ένα γράφημα. Κάθε ακολουθία συνδέσμων για την πραγματοποίηση μιας συνδιάλεξης αντιστοιχεί σε μονοπάτια. Τα μονοπάτια περιγράφουν τις ανάγκες για οπτικές συχνότητες σε κάθε φυσικό σύνδεσμο. Οι οπτικές συχνότητες αντιστοιχούν σε χρώματα. Η διατήρηση της ίδιας συχνότητας σε όλους τους συνδέσμους εξασφαλίζεται από το ότι κάθε μονοπάτι χρωματίζεται από μόνο ένα χρώμα. Επίσης δύο συνδιαλέξεις που περνούν από την ίδια οπτική ίνα αντιστοιχούν σε μονοπάτια που περνούν από την ίδια ακμή και άρα χρωματίζονται με διαφορετικό χρώμα.

Στη συνέχεια θα παρουσιάσουμε μία παραλλαγή του παραπάνω προβλήματος, με το να επιτρέπουμε τη χρήση πολλαπλών παράλληλων συνδέσεων με σκοπό να μπορέσουμε να ικανοποιήσουμε όλο το σύνολο των απαιτήσεων, ακόμη και αν το διαθέσιμο εύρος ζώνης είναι ανεπαρκές. Σε αυτή τη νέα προσέγγιση σκοπός είναι να ελαχιστοποιήσουμε τον αριθμό των ενεργών συνδέσεων και επομένως να μειώσουμε το κόστος του δικτύου.

Το παραπάνω πρόβλημα μπορούμε να το ανάγουμε σε αντίστοιχα προβλήματα με γραφήματα. Δοσμένων μιας λίστας από ζεύγη κόμβων και ενός αριθμού από διαθέσιμα χρώματα σκοπός είναι να δρομολογήσουμε μονοπάτια με τα δοσμένα ζεύγη κόμβων σαν σημεία τερματισμού και να αναθέσουμε χρώματα στα μονοπάτια ελαχιστοποιώντας τις συγκρούσεις χρωμάτων (μονοπάτια που χρησιμοποιούν την ίδια ακμή να έχουν, αν γίνεται, διαφορετικό χρώμα) σε όλες τις δυνατές δρομολογήσεις και χρωματισμούς. Παρακάτω θα δούμε αποτελεσματικούς αλγορίθμους για επίλυση του προβλήματος για απλές τοπολογίες δικτύων, όπως είναι οι αλυσίδες, οι αστέρες και οι δακτύλιοι. Για τις αλυσίδες οι λύσεις μας είναι βέλτιστες, ενώ για τους αστέρες και τους δακτυλίους, όπου το παραπάνω πρόβλημα είναι NP-δύσκολο, οι λύσεις μας προσεγγίζουν με έναν παράγοντα δύο τη βέλτιστη λύση. Το κλειδί της λύσης του προβλήματος είναι η μετατροπή του σε πρόβλημα χρωματισμού ακμών σε διμερή γραφήματα.

2. Χρωματισμός και Πολύ-χρωματισμός.

2.1 Χρωματισμός και πολύ-χρωματισμός μονοπατιών γραφημάτων.

Στα αμιγώς οπτικά (all-optical) δίκτυα ένα πολύ σημαντικό ζήτημα, όπως αναφέρθηκε, είναι η βελτιστοποίηση της χρήσης του διαθέσιμου εύρους ζώνης (bandwidth). Σ' αυτά τα δίκτυα πολλαπλές συνδέσεις μπορούν να υλοποιηθούν μέσω της ίδιας οπτικής ίνας: πολλά σήματα μπορούν να μεταδοθούν μέσω της ίδιας ίνας με χρήση διαφορετικών συχνοτήτων. Εκτός αυτού, κάθε σήμα πρέπει να μεταδίδεται στο ίδιο μήκος κύματος σε κάθε ίνα από τον πομπό στον δέκτη. Παρόμοια προβλήματα βελτιστοποίησης μπορούν να αναχθούν σε προβλήματα χρωματισμού μονοπατιών σε γραφήματα.

Η πρότυπη εξήγηση για τον όρο "*βέλτιστη χρήση εύρους ζώνης*" (optimal use of bandwidth) είναι: δοθείσης μιας σειράς από επικοινωνιακές απαιτήσεις, να ελαχιστοποιήσουμε τον ολικό αριθμό των μηκών κύματος που απαιτούνται για να ικανοποιήσουμε όλες τις απαιτήσεις του προβλήματος. Το αντίστοιχο θεωρητικό πρόβλημα με χρήση γραφημάτων είναι γνωστό ως *Routing and Path Coloring Problem(RPC):* δεδομένου ενός γραφήματος G(V,E) και μιας λίστας ζευγαριών κόμβων από το V πρέπει να καθορίσουμε και να χρωματίσουμε τα μονοπάτια που συνδέουν τους κόμβους κάθε ζευγαριού, έτσι ώστε μονοπάτια που χρησιμοποιούν την ίδια ακμή να έχουν διαφορετικά χρώματα και έτσι να έχουμε τον ελάχιστο αριθμό από χρώματα.

Η τεχνολογία των οπτικών ινών δεν υποστηρίζει την διπλή κατεύθυνση στη χρήση της ίνας, έτσι, για παράδειγμα, δεν είναι δυνατό δύο αντίθετα κατευθυνόμενα

σήματα να χρησιμοποιούν την ίδια ίνα. Γι' αυτό το λόγο ενώσεις μεταξύ δύο κόμβων, συνήθως, αποτελούνται από δύο αντίθετα κατευθυνόμενες οπτικές ίνες. Παρόλα αυτά ένα γενικό πρόβλημα χρησιμοποιεί μη κατευθυνόμενα γραφήματα και μονοπάτια [20]. Το μοντέλο αυτό ανταποκρίνεται στην περίπτωση που η επικοινωνία για κάθε απαίτηση είναι διπλής κατεύθυνσης και σήματα και από τις δύο κατευθύνσεις πρέπει να χρησιμοποιούν το ίδιο σύνολο συνδέσμων και το ίδιο μήκος κύματος. Ένα άλλο μοντέλο που έχει προταθεί [4] κάνει χρήση συμμετρικών κατευθυνόμενων γραφημάτων και κατευθυνόμενων μονοπατιών (paths). Το μοντέλο αυτό μας δίνει περισσότερη ευελιξία καθώς επιτρέπει ανεξάρτητη χρήση των δύο αντίθετων ινών της σύνδεσης. Για παράδειγμα, έχει σαν αποτέλεσμα καλύτερη δέσμευση εύρους ζώνης στην περίπτωση της μονόπλευρης επικοινωνίας. Εμείς θα ασχοληθούμε αποκλειστικά με το μη-κατευθυνόμενο μοντέλο.

Στην πράξη το διαθέσιμο εύρος ζώνης μιας οπτικής ίνας είναι περιορισμένο. Έτσι εισάγουμε ένα καινούριο πρόβλημα που στηρίζεται πάνω στην εξής ιδέα: όλες οι απαιτήσεις μπορούν να ικανοποιηθούν με το διαθέσιμο εύρος ζώνης και με τη χρήση παράλληλων συνδέσεων. Ο στόχος είναι να ελαχιστοποιήσουμε τον συνολικό αριθμό των ενεργών συνδέσμων στο δίκτυο κι έτσι, μ' αυτόν τον τρόπο, να μειώσουμε το κόστος του δικτύου. Ο αριθμός των παράλληλων συνδέσεων που χρειάζονται μεταξύ δύο γειτονικών κόμβων στο δίκτυο είναι ο μέγιστος αριθμών των συνδέσεων που χρησιμοποιούν το ίδιο μήκος κύματος και περνούν από την ίδια σύνδεση ανάμεσα σε δύο κόμβους.

Το παραπάνω πρόβλημα είναι γνωστό ως Δρομολόγηση και Πολυ-χρωματισμός Διαδρομών (*Routing and Path Multi-Coloring Problem - RPMC)*: δοσμένου ενός μη-κατευθυνόμενου γραφήματος G(E, V), μιας λίστας ζευγαριών κόμβων από το V και ενός αριθμού διαθέσιμων χρωμάτων w καθορίστε μονοπάτια που συνδέουν τους κόμβους των δοσμένων ζευγαριών και χρωμάτων (ανάμεσα στο 1 και στο w), έτσι ώστε το κόστος

$$\sum_{e \varepsilon E} \max_{1 \le c \le w} \mu(e,c)$$

να είναι ελάχιστο, όπου μ(e,c) είναι η πολλαπλότητα του χρώματος c στην ακμή e (δηλαδή ο αριθμός των μονοπατιών που χρησιμοποιούν την ακμή e και χρωματίστηκαν με το χρώμα c).

Ένα άλλο μοντέλο που χρησιμοποιεί παράλληλες συνδέσεις προτάθηκε πρόσφατα [4]: ο αριθμός των παράλληλων συνδέσεων δίνεται και είναι ο ίδιος για όλες τις ακμές και ο σκοπός είναι να ελαχιστοποιήσουμε τον αριθμό των χρωμάτων. Η προσέγγιση αυτή είναι διαφορετική από την δικιά μας, καθώς δεν υποθέτει ούτε περιορισμένο εύρος ζώνης ούτε ελαχιστοποίηση του αριθμού των ενεργών συνδέσεων στο δίκτυο.

Σε μια χρήσιμη παραλλαγή των προβλημάτων χρωματισμού μονοπατιών (path coloring problems) τα μονοπάτια δίνονται εκ των προτέρων και ζητείται μόνο ο χρωματισμός που βελτιστοποιεί το αποτέλεσμα. Συμβολίζουμε τις παραλλαγές των RPC και RPMC με PC και PMC αντίστοιχα. Το PC είναι ισοδύναμο με χρωματισμό κόμβων σε μια ευρεία οικογένεια γράφων διασταύρωσης. Εκτός από το ενδιαφέρον που παρουσιάζουν σαν προβλήματα, αλγόριθμοι προβλημάτων χρωματισμού καθορισμένων μονοπατιών (prescibed-paths) είναι χρήσιμα για την επίλυση προβλημάτων μη-καθορισμένων μονοπατιών. Σ' αυτό το σημείο θα πρέπει να αναφέρουμε ότι για τοπολογίες δέντρων οι δύο αυτές εκδοχές συμπίπτουν.

Εδώ παρουσιάζουμε αλγόριθμους που λύνουν το RPMC για αλυσίδες βέλτιστα και το RPMC για αστέρες και δακτυλίους με παράγοντα 2. Οι άπληστοι αλγόριθμοι, στα συγκεκριμένα προβλήματα δεν φαίνονται ικανοποιητικοί. Οι τεχνικές που παρουσιάζουμε εδώ ξεπερνούν το πρόβλημα με ένα αρκετά κομψό τρόπο: με τη χρήση μετασχηματισμών σε χρωματισμό ακμών διμερών γραφημάτων. Συμπεραίνουμε ότι ο προσεγγιστικός παράγοντας (2) μπορεί να υποστεί περαιτέρω βελτίωση, παρ' ότι για τους δακτυλίους δεν είναι χειρότερος από τον, μέχρι τώρα, καλύτερο γνωστό προσεγγιστικό παράγοντα για RPC.

Τα RPC και PC έχουν μελετηθεί για πολλές τοπολογίες, όπως για παράδειγμα αλυσίδες, δακτύλιοι, δέντρα, δέντρα δακτυλίων και άλλα.

Για τοπολογίες δέντρων τα προβλήματα συμπίπτουν και μπορούν να λυθούν βέλτιστα σε πολυωνυμικό χρόνο για αλυσίδες [19], καθώς επίσης και για καθορισμένου βαθμού δέντρα [18]. Από την άλλη πλευρά, είναι γνωστό [23] ότι το πρόβλημα για τοπολογίες αστέρων είναι ισοδύναμο με το πρόβλημα χρωματισμού ακμών που είναι NP-δύσκολο [10], γι' αυτό για μη καθορισμένου βαθμού δέντρων το πρόβλημα είναι NP-δύσκολο. Χρησιμοποιώντας την παραπάνω ισοδυναμία μπορούμε να επιτύχουμε έναν

προσεγγιστικό αλγόριθμο για RPC (και PC) για αστέρες που έχει προσεγγιστικά παράγοντα τον ίδιο με τον καλύτερο αλγόριθμο για χρωματισμό ακμών (Edge Coloring). Ο αλγόριθμος των Hochbaum, Nishizeki και Shmoys [9] πετυχαίνει παράγοντα 4/3 που είναι ο καλύτερος δυνατός, εκτός αν P=NP. Ο ίδιος προσεγγιστικός παράγοντας μπορεί να επιτευχθεί για μη-καθορισμένου βαθμού δέντρα χρησιμοποιώντας μια τεχνική μετονομασμού χρώματος [23].

Το πρόβλημα PC για δακτυλίους, που είναι επίσης γνωστό και ως: *circular-arc coloring problem*, έχει αποδειχθεί ότι είναι NP-δύσκολο [7], ενώ υπάρχει και ένας προσεγγιστικός αλγόριθμος που δίνει παράγοντα 3/2 [11]. Το RPC για δακτυλίους είναι επίσης NP-δύσκολο [18] και υπάρχει προσεγγιστικός αλγόριθμος με παράγοντα 2 [16, 20]. Αντίστοιχα αποτελέσματα υπάρχουν και για τοπολογίες δέντρων δακτυλίων.

Οι Li και Simha [13] καθώς και οι Margara και Simon [15] μελετούν μια γενίκευση του PC προβλήματος, στο οποίο υπάρχει ένας συγκεκριμένος αριθμός παράλληλων συνδέσεων που χρησιμοποιείται σ’ όλο το δίκτυο αντί σε μία μόνο ίνα. Μάλιστα έχει δειχθεί ότι για τοπολογίες αστέρων στις οποίες ο αριθμός των παράλληλων οπτικών ινών k είναι ζυγός (παραπομπή 1) μπορεί να επιτευχθεί χρωματισμός που χρησιμοποιεί βέλτιστο αριθμό από w_{lb} (=load/k) μήκη κύματος ανά οπτική ίνα (που είναι φυσικό κάτω άκρο), όταν έχουμε k περιττός, δεν δίνεται κάποιο άνω φράγμα, παρόλα αυτά όμως μπορούμε να χρησιμοποιήσουμε k-1 οπτικές ίνες οπότε κάνουμε αναγωγή στο προηγούμενο πρόβλημα, που θα μας δώσει ένα πάνω φράγμα περίπου $k/(k-1)*w_{lb}$ συχνότητες ανά ίνα. Για k ίνες απαιτήσεις δίνουν ένα άνω φράγμα της τάξης $(k+1)/k*w_{lb}$ συχνότητες ανά ίνα. Παρόμοια αποτελέσματα αποδείχθηκαν και από τους Margara και Simon [15].

2.2 Τοπολογίες γραφημάτων

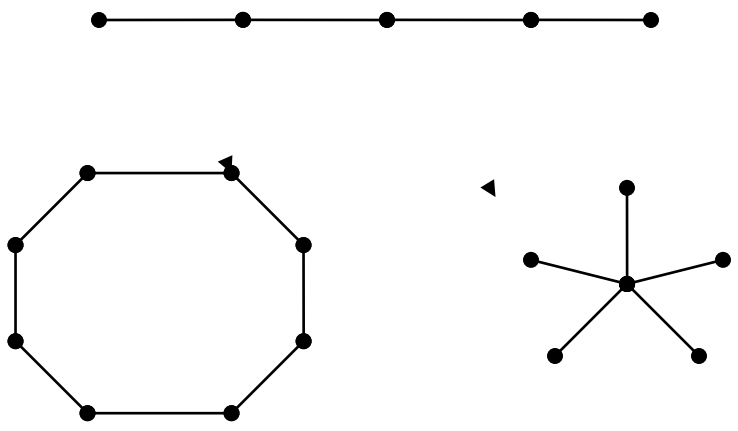
Στη συνέχεια θα αναφερθούμε στις τοπολογίες για την επίλυση προβλημάτων στα οποία αναφερθήκαμε στην πιο πάνω ενότητα, και θα δώσουμε κάποιους ορισμούς, οι οποίοι θα μας βοηθήσουν στην περαιτέρω κατανόηση της ανάλυσης που θα ακολουθήσει..

Θα θεωρήσουμε τα γραφήματα μη κατευθυνόμενα και ότι όλα τα μονοπάτια είναι απλά, δεν επιτρέπουμε δηλαδή την επανάληψη κόμβων. Δοθέντος ενός γραφήματος, το n δηλώνει τον αριθμό των κόμβων οι οποίοι αριθμούνται από 0,1,…,n

Αλυσίδα είναι ένα γράφημα για το οποίο V = {1,2,…,n) και E = {{i,i+1}|1<i<n-1}. Ανάμεσα σε δύο οποιεσδήποτε κορυφές μιας αλυσίδας υπάρχει ένα μοναδικό μονοπάτι το οποίο περιγράφεται πλήρως από τις κορυφές αυτές.

Δακτύλιος είναι ένα γράφημα για το οποίο V = {1,2,…,n} και E = {{i,i+1}|1<i<n-1}U{{n,1}}. Σε ένα δακτύλιο υπάρχουν δύο μονοπάτια ανάμεσα σε δύο κορυφές i και j. Αν i<j συμβολίζουμε με {i,j} το μονοπάτι που δεν περιέχει την ακμή {1,n} και {j,I} αυτό που την περιέχει. Μπορούμε από μία αλυσίδα με n κορυφές να κατασκευάσουμε ένα δακτύλιο με n κορυφές προσθέτοντας μία ακμή, την {n,1} ή ένα δακτύλιο με n-1 κορυφές ταυτοποιώντας τις κορυφές 1 και n της αλυσίδας. Αντίστροφα από ένα δακτύλιο με n κορυφές μπορούμε να πάρουμε μία αλυσίδα με n κορυφές αφαιρώντας την ακμή {n,1} ή μια αλυσίδα με n+1 κορυφές με διαχωρισμό της κορυφής 1 σε δύο άλλες (1 και n+1).

Αστέρας είναι ένα γράφημα το οποίο είναι συνεκτικό και δεν έχει κύκλους, είναι δηλαδή μια ειδική κατηγορία δέντρου. Όλα τα μονοπάτια σε ένα αστέρι έχουν μήκος ένα ή δύο και περιγράφονται από τα δύο άκρα τους. Στο σχήμα 1 παρουσιάζονται όλες οι παραπάνω τοπολογίες γραφημάτων.



Σχήμα 1 Τοπολογίες γραφημάτων. Πάνω: αλυσίδα, αριστερά:δακτύλιος, δεξιά: αστέρας

Ένα στιγμιότυπο του RPMC είναι μία τριάδα (G,R,w), όπου G είναι ένα γράφημα, R είναι η λίστα των ζευγαριών των κόμβων και w ένας θετικός ακέραιος. Ομοίως, στιγμιότυπο του PMC είναι μία τριάδα (G,P,w), όπου P είναι η λίστα των μονοπατιών. Δοθέντος ενός στιγμιότυπου I του RPMC ή του PMC συμβολίζουμε με OPT(I) το κόστος μιας βέλτιστης λύσης του αντίστοιχου προβλήματος για το στιγμιότυπο I. Λέμε ότι ένας αλγόριθμος πετυχαίνει παράγοντα προσέγγισης α >1, αν για κάθε πιθανό στιγμιότυπο I το κόστος της λύσης που επιστρέφει ο αλγόριθμος δεν είναι μεγαλύτερο από α*OPT(I). Για ένα στιγμιότυπο του PMC συμβολίζουμε με L(e,P) το βάρος της ακμής e, δηλαδή τον αριθμό των μονοπατιών στο P που χρησιμοποιούν την e. Θα πρέπει να παρατηρήσουμε ότι σε οποιοδήποτε πολύ-χρωματισμό μονοπατιών στο P με w χρώματα για κάθε ακμή e υπάρχει τουλάχιστο ένα χρώμα c τέτοιο ώστε μ(e,c)≥[L(e,P)/w]. Επομένως το κόστος για οποιαδήποτε λύση είναι τουλάχιστο $LB(I)=\sum_{e\varepsilon E}$ [L(e,P)/w]. Γι' αυτό το λόγο το LB(I) είναι ένα κάτω φράγμα για το OPT(I).

3. Πολύ-χρωματισμός σε αλυσίδες

3.1 Αλγόριθμος πολύ-χρωματισμού για αλυσίδες.

Στην παράγραφο αυτή θα παρουσιάσουμε έναν πολυωνυμικό αλγόριθμο που λύνει βέλτιστα το PMC για αλυσίδες και είναι ισοδύναμος και για τη λύση του RPMC. Έστω (G,P,w) ένα στιγμιότυπο του PMC, στο οποίο το G είναι αλυσίδα με n κόμβους. Λέμε ότι ένα μονοπάτι που συνδέει τους κόμβους i και j αρχίζει στο i και τελειώνει στο j, αν i<j.

Η τεχνική που θα ακολουθήσουμε για τον πολύ-χρωματισμό βασίζεται σε μετατροπή από το PMC σε χρωματισμό ακμών σε διμερή γράφημα. Αυτή η τεχνική εφαρμόζεται απευθείας σε λίστες μονοπατιών στις οποίες το βάρος κάθε ακμής είναι πολλαπλάσιο του w και κανένας κόμβος δεν είναι ταυτόχρονα αρχικό σημείο για ένα μονοπάτι και τελικό ενός άλλου μονοπατιού. Γενικά χρειάζεται ένας μετασχηματισμός του P, πριν γίνει η μετατροπή, με σκοπό να πάρουμε μια νέα λίστα μονοπατιών P΄ με τις επιθυμητές και απαιτούμενες ιδιότητες. Στη συνέχεια ακολουθεί ο αλγόριθμος:

Βήμα 1^{ο}: Για να κάνουμε όλα τα βάρη των ακμών ακέραια πολλαπλάσια του w προσθέτουμε μερικά μονοπάτια μήκους ένα (όπου χρειάζεται). Ο αριθμός των παραπάνω μονοπατιών για μια ακμή e είναι w - (L(e,P) mod w).

Βήμα 2^{ο}: Όσο υπάρχει μονοπάτι p_1 που τελειώνει και μονοπάτι p_2 που αρχίζει στον ίδιο κόμβο, τα ενώνουμε για να κατασκευάσουμε ένα μεγαλύτερο μονοπάτι. Συμβολίζουμε με P΄ τη λίστα με τα μονοπάτια που προκύπτουν.

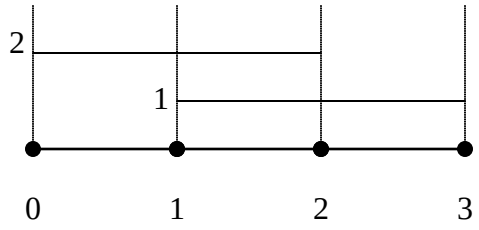
Βήμα 3^{ο}: Για κάθε κόμβο υ που είναι τελικό σημείο μονοπατιού, το P΄ περιέχει είτε μονοπάτια που αρχίζουν στον υ είτε μονοπάτια που τελειώνουν στον υ. Και στις δύο περιπτώσεις ο αριθμός αυτών των μονοπατιών είναι ακέραια πολλαπλάσια του w. Τα χωρίζουμε σε συλλογές από w στοιχεία με αυθαίρετο τρόπο. Συμβολίζουμε το σύνολο με όλες τις συλλογές μονοπατιών που αρχίζουν (τελειώνουν) με B (E αντίστοιχα).

Βήμα 4^{ο}: Κατασκευάζουμε ένα διμερές γράφημα H = (B,E,A). Κάθε μονοπάτι στο P ανήκει σε ακριβώς δύο συλλογές, σε μια που "αρχίζει" και σε μια που "τελειώνει". Για κάθε μονοπάτι στο P υπάρχει μία ακμή στο A που ενώνει την συλλογή που αρχίζει και την συλλογή που τελειώνει. Άρα, το H είναι ένα w-κανονικό διμερές γράφημα.

Βήμα 5^{ο}: Χρωματίζουμε τις ακμές του H με w χρώματα χρησιμοποιώντας έναν κατάλληλο αλγόριθμο. Σε κάθε μονοπάτι του P΄ αναθέτουμε το χρώμα της αντίστοιχης ακμής του H [21].

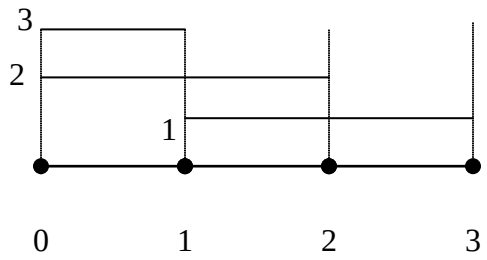
Βήμα 6^{ο}: Σε κάθε μονοπάτι του P ανατίθεται το χρώμα του αντίστοιχου (μεγάλου) μονοπατιού του P΄.

Θα πρέπει, σ' αυτό το σημείο, να αναφερθούμε λίγο στην αναγκαιότητα της πραγμάτωσης του βήματος ένα και του βήματος δύο. Έστω ότι δεν προσθέτουμε τα μονοπάτια μήκους ένα. Τότε υπάρχει κίνδυνος μονοπάτια που κανονικά θα έπρεπε να πάρουν διαφορετικά χρώματα να πάρουν το ίδιο. Αυτό φαίνεται στο παρακάτω πολύ απλό παράδειγμα. Έστω ότι έχουμε μία αλυσίδα με τρεις κόμβους και έστω ότι έχουμε τα μονοπάτια {0,2} και {1,3} (σχήμα 3.1) τα οποία θέλουμε να χρωματίσουμε με δύο χρώματα.



Σχήμα 3.1 Παράδειγμα μη χρήσης του βήματος ένα.

Είναι φανερό ότι ένα από τα δύο θα πάρει το χρώμα ένα και το άλλο θα πάρει το χρώμα 2. Ο αλγόριθμος θα δώσει και στα δύο το ίδιο χρώμα αφού θα θεωρήσει ότι ανήκουν σε διαφορετικές συλλογές πράγμα που σημαίνει ότι μπορούν να πάρουν το ίδιο χρώμα. Έτσι καταλήγουμε σε μη βέλτιστο χρωματισμό. Το ίδιο συμβαίνει και αν δεν ενώσουμε τα μονοπάτια σε ένα μεγαλύτερο. Σ' αυτή την περίπτωση έχουμε ένα μονοπάτι το οποίο έχει προστεθεί από το βήμα ένα του αλγορίθμου, όπως φαίνεται στο παρακάτω σχήμα,



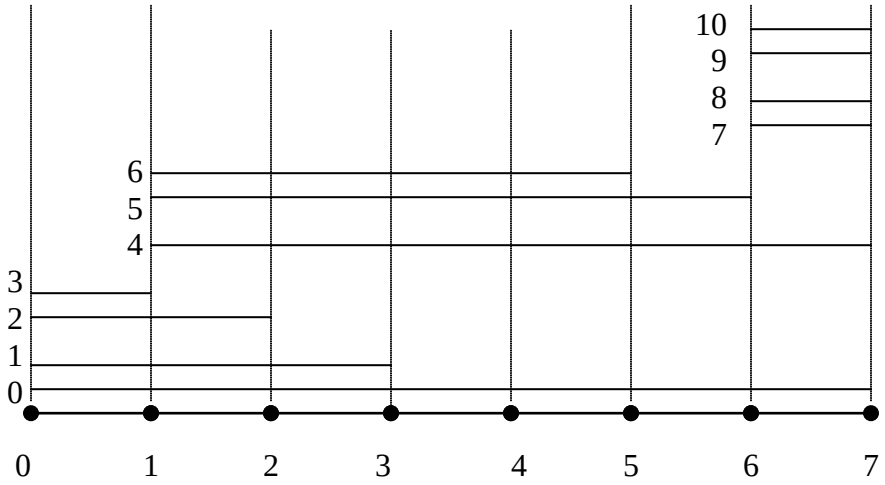
Σχήμα 3.2 Παράδειγμα μη χρήσης του βήματος δύο.

αλλά δεν ενώνουμε το νέο μονοπάτι με το {1,3} για να φτιάξουμε το μεγάλο μονοπάτι {0,2} και στη συνέχεια να χρωματίσουμε τα δύο μονοπάτια που μένουν. Έτσι πάλι ο αλγόριθμος βάζει σε διαφορετικές συλλογές τα μονοπάτια {0,2} και {1,2} και τα χρωματίζει μη βέλτιστα με το ίδιο χρώμα. Ενώ έχει σωστά χρωματίσει με διαφορετικό χρώμα τα μονοπάτια {0,2} και {0,1} που ανήκουν στην ίδια συλλογή.

3.2 Παράδειγμα χρήσης του Αλγορίθμου.

Σε αυτή την παράγραφο θα δώσουμε ένα παράδειγμα επίλυσης του PMC για αλυσίδες. Έστω ότι έχουμε μια αλυσίδα με 8 κόμβους αριθμημένους από το 0 έως το 7. για την αλυσίδα δίνονται έντεκα μονοπάτια: {0,7}, {0,3}, {0,2}, {0,1}, {1,7}, {1,6}, {1,5}, {6,7}, {6,7}, {6,7} και {6,7}. Επιθυμούμε να γίνει χρωματισμός με w = 4 χρώματα. Στο σχήμα 3.3 φαίνεται η αλυσίδα με τα μονοπάτια.

Στο πρώτο βήμα ο αλγόριθμος βάζει μονοπάτια μήκους ένα ώστε σε κάθε ακμή τα επικαλυπτόμενα μονοπάτια να είναι ακέραια πολλαπλάσια του w (που στην περίπτωσή μας είναι τέσσερα).



Σχήμα 3.3 Η αλυσίδα με τα μονοπάτια.

Τα μονοπάτια μαζί με τα μονοπάτια μήκους ένα που προσθέτει ο αλγόριθμος τα κρατάμε στην εξής δομή:

```
struct path{
        int begin;
        int end;
        int color;
}P[MAX];
```

στην οποία το begin είναι η αρχή του μονοπατιού, το end είναι το τέλος του μονοπατιού και color είναι το χρώμα με το οποίο θα χρωματιστεί το μονοπάτι.

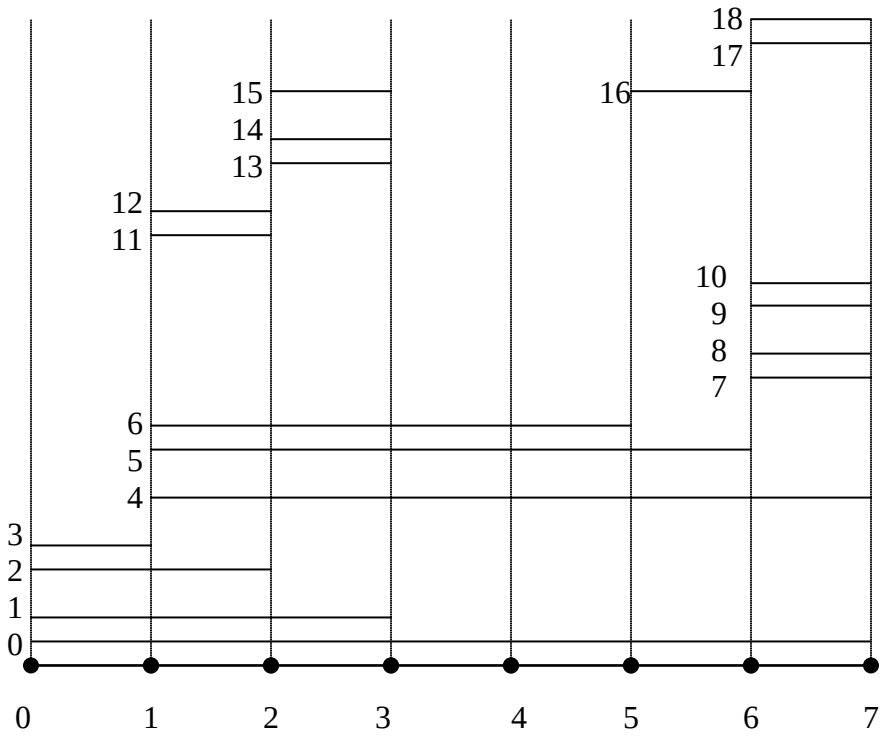
Για να βρούμε το βάρος της κάθε ακμής εκτός της πρώτης χρησιμοποιούμε τον τύπο:

Load[i]=Load[i-1]+NumberOfBeginningNodes[i]-NumberOfEndingNodes[i]

δηλαδή το βάρος της ακμής είναι ίσο με το βάρος της προηγούμενης συν τον αριθμό των μονοπατιών που αρχίζουν μείον τον αριθμό μονοπατιών που τελειώνουν. Ενώ το βάρος της πρώτης ακμής είναι ίσο με τον αριθμό των μονοπατιών που ξεκινάν από τον κόμβο 0.

Από το βάρος της κάθε ακμής βρίσκουμε και πόσα είναι τα μονοπάτια μήκους ένα που πρέπει να προσθέσουμε σε κάθε ακμή, έτσι ώστε τα βάρη των ακμών να είναι ακέραια πολλαπλάσια του w (που στην περίπτωσή μας είναι τέσσερα). Έτσι παρατηρούμε ότι το βάρος της ακμής {0,1} είναι τέσσερα, άρα δεν χρειάζεται να

προσθέσουμε κανένα μονοπάτι. Το βάρος όμως της ακμής {1,2} είναι έξη, άρα θα πρέπει να προσθέσουμε ακόμη δύο μονοπάτια μήκους ένα ώστε η ακμή να αποκτήσει βάρος οκτώ (που είναι ακέραιο πολλαπλάσιο του τέσσερα). Με την ίδια λογική προστίθενται, όπου χρειάζεται και άλλα μονοπάτια μήκους ένα. Στο επόμενο σχήμα φαίνονται τα μονοπάτια που είχαμε αρχικά μαζί με τα νέα μονοπάτια μήκους ένα.



Σχήμα 3.4 Η έξοδος του βήματος ένα.

Στη συνέχεια αυτό που κάνει ο αλγόριθμος, που υλοποιήσαμε, είναι να βρίσκει μονοπάτια που τελειώνουν σε ένα κόμβο και να τα ενώνει σε ένα μεγαλύτερο μονοπάτι. Για τα μεγάλα μονοπάτια χρησιμοποιούμε τη δομή:

```
struct path1{
        int begin;
        int end;
        int color;
}NewP[MAX];
```

η οποία έχει τα ίδια στοιχεία με τη δομή που εξετάσαμε παραπάνω.

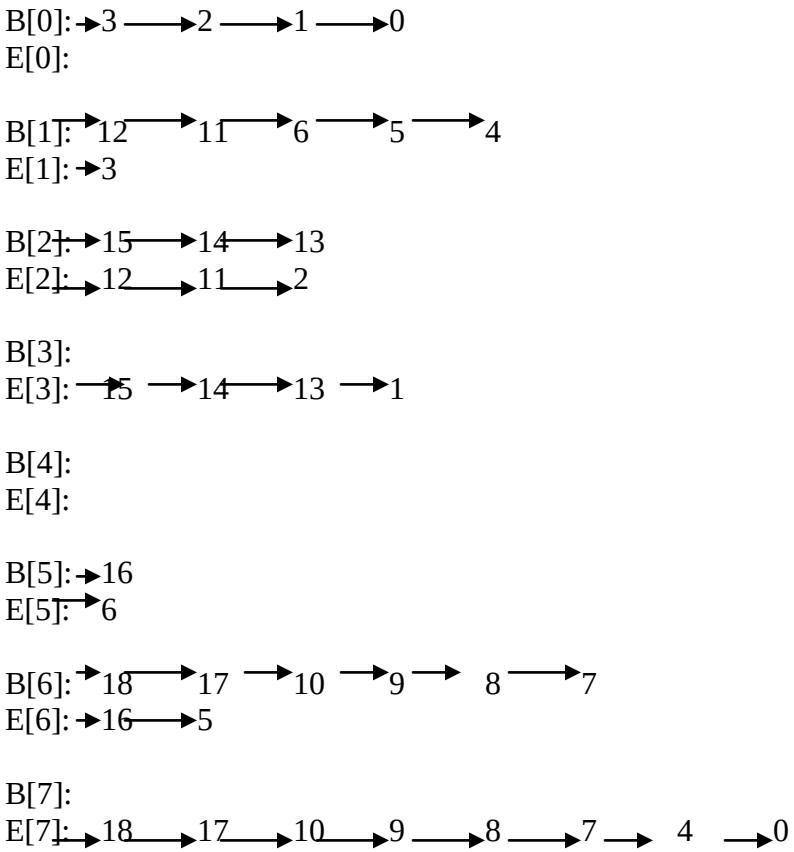
Για να τοπετύχουμε αυτό χρησιμοποιούμε και δύο βοηθητικούς πίνακες λιστών τους B[i] και E[i]. Στη δικιά μας περίπτωση το i παίρνει τιμές από το 0 μέχρι το 7.Στο

B[i] θα βάλουμε τα μονοπάτια που ξεκινάν από τον κόμβο i, ενώ στο E[i] βάζουμε τα μονοπάτια που τελειώνουν στον κόμβο i. Τα B[i] και E[i] ορίζονται ως εξής:

```
typedef struct liststr{
        int pathid;
        struct liststr *next;
}*list;
list B[MAX], E[MAX];
```

το pathid είναι το νούμερο του μονοπατιού που ξεκινάει από κάποιο κόμβο.

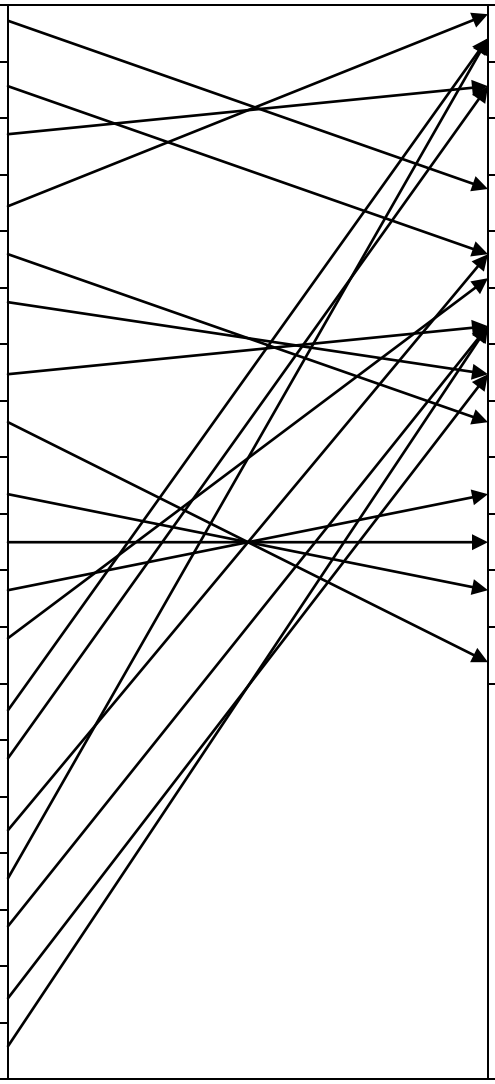
Επίσης θα χρειαστούμε και έναν βοηθητικό πίνακα, τον οποίο τον ονομάζουμε LongPath[j] με j να είναι ο αριθμός των μονοπατιών. Στο παράδειγμά μας μετά το βήμα ένα του αλγορίθμου έχουμε συνολικά 18 μονοπάτια, άρα το j κυμαίνεται από 0 έως 17.



Σχήμα 3.5 Οι τιμές των βοηθητικών B[i], E[i]. Στον κόμβο 0 ξεκινάν τα μονοπάτια 3, 2, 1 και 0 κει δεν τελειώνει κανένα. Ενώ για παράδειγμα στον κόμβο 2 ξεκινάν τα μονοπάτια 15, 14 και 13 και τελειώνουν τα μονοπάτια 12, 11 και 2.

Στη συνέχεια αυτό που πρέπει να κάνουμε είναι να αντιστοιχίσουμε κάθε μονοπάτι του P με ένα μεγάλο μονοπάτι που θα μπει στην λίστα NewP, που αναφέραμε προηγουμένως. Για να το πετύχουμε αυτό χρησιμοποιούμε τον πίνακα LongPath. Πρέπει να γεμίσουμε τον πίνακα LongPath, ο οποίος θα αντιστοιχίσει κάθε μονοπάτι που υπάρχει μετά το βήμα ένα με ένα καινούριο μεγάλο μονοπάτι. Διατρέχουμε τα B[i] και E[i], έστω ότι είμαστε στον κόμβο 0. Από τον κόμβο 0, όπως φαίνεται από το παραπάνω σχήμα ξεκινάν τα μονοπάτια 3, 2, 1 και 0 (αυτοί οι αριθμοί αντιστοιχούν στα μονοπάτια

#	Αρχή	Τέλος	LongPath
0	0	7	3
1	0	3	2
2	0	2	1
3	0	1	0
4	1	7	7
5	1	6	6
6	1	5	5
7	6	7	11
8	6	7	10
9	6	7	9
10	6	7	8
11	1	2	4
12	1	2	0
13	2	3	1
14	2	3	4
15	2	3	0
16	5	6	5
17	6	7	6
18	6	7	5



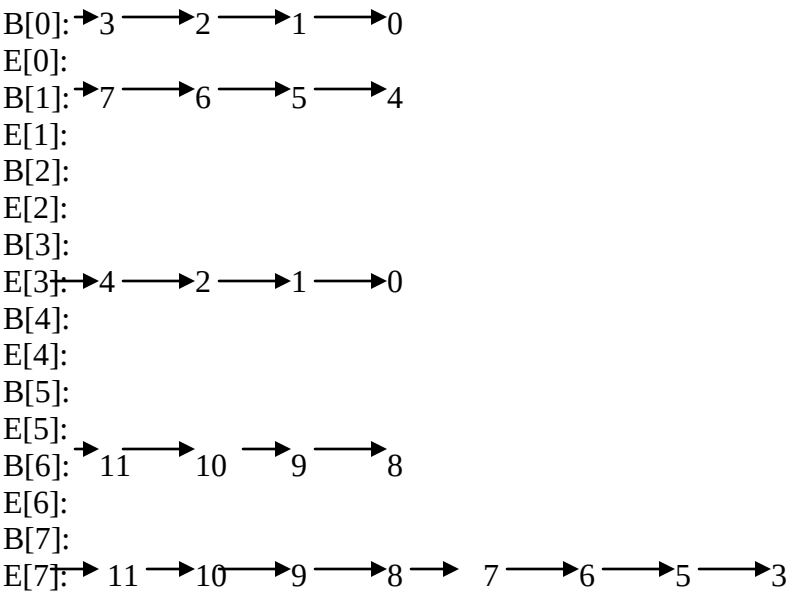
P’

0	0	3
1	0	3
2	0	3
3	0	7
4	1	3
5	1	7
6	1	7
7	1	7
8	6	7
9	6	7
10	6	7
11	6	7

Σχήμα 3.6. Τα μονοπάτια του P και η αντιστοίχισή τους με τα μεγάλα μονοπάτια του P’

{0,1}, {0,2}, {0,3} και {0,7}) άρα στο LongPath[3] θα βάλω την τιμή 0, στο LongPath[2] θα βάλω την τιμή 1, στο LongPath[1] βάζω την τιμή 2 και τέλος στο LongPath[0] βάζω την τιμή 3. οι παραπάνω αναθέσεις τιμών σημαίνουν ότι το 3^{o} μονοπάτι θα ανήκει στο μεγάλο μονοπάτι 0, το 2^{o} μονοπάτι ανήκει στο μεγάλο μονοπάτι 1 κοκ. Στον επόμενο κόμβο παρατηρούμε ότι ξεκινάν τα μονοπάτια 12, 11, 6 και 4 και τελειώνει το μονοπάτι 3. αυτό σημαίνει ότι το μονοπάτι 3 θα ενωθεί με το μονοπάτι 12 σε ένα μεγάλο μονοπάτι. Επομένως το LongPath[12] θα είναι ίσο με το LongPath[3] δηλαδή ίσο με 0. αυτό σημαίνει ότι το μονοπάτι 12 ανήκει στο μεγάλο μονοπάτι 0. με την ίδια λογική αντιστοιχίζονται όλα τα μικρά μονοπάτια από ένα μεγάλο. Στο σχήμα 3.6 φαίνονται τα μικρά μονοπάτια και η αντιστοίχηση με τα μεγάλα.

Στο επόμενο βήμα ακολουθεί η ο χωρισμός σε συλλογές το πολύ w (4 στο παράδειγμά μας) στοιχείων. Για να κάνουμε αυτό τον χωρισμό θα χρειαστούμε πάλι τους πίνακες λιστών B[i] και E[i]. Αυτή τη φορά στο B[i] θα βάλουμε τα μεγάλα μονοπάτια που ξεκινάν από τον κόμβο i, ενώ στο E[i] βάζουμε τα μεγάλα μονοπάτια που τελειώνουν στον κόμβο i. Οι B[i] και E[i] έχουν τις τιμές του παρακάτω σχήματος.



Σχήμα 3.7. Οι τιμές των B[i] και E[i] για τα νέα μεγάλα μονοπάτια.

Τα μεγάλα μονοπάτια 3, 2, 1 και 0 που ξεκινάν στον κόμβο 0 θα αποτελέσουν την συλλογή 0. Τα μονοπάτια 7, 6, 5, και 4 που ξεκινάν από τον κόμβο 1 θα αποτελέσουν τη συλλογή 1 και τέλος τα μονοπάτια 11, 10, 9, 8 που ξεκινάν από τον κόμβο 6 θα

αποτελέσουν τη συλλογή 2. οπότε έχουμε τρεις συλλογές για τα μονοπατιών που αρχίζουν. Πρέπει τώρα να βρούμε και τις συλλογές των μονοπατιών που τελειώνουν. Ελέγχουμε τον πίνακα E[i] και παρατηρούμε ότι τα μονοπάτια 4, 2, 1, 0 τελειώνουν στον κόμβο 3, αυτά τα μονοπάτια αποτελούν την συλλογή 0. Στον κόμβο 7 τελειώνουν τα μονοπάτια 11, 10, 9, 8, 7, 6, 5, 3. Αυτά τα μονοπάτια είναι παραπάνω από w = 4 επομένως θα πρέπει να χωριστούν σε δύο συλλογές w στοιχείων. Τα πρώτα 4 αποτελούν μία συλλογή (την συλλογή 1) ενώ τα 4 τελευταία αποτελούν την άλλη συλλογή (συλλογή 2).

Αφού τελειώσουμε με το βήμα του χωρισμού σε συλλογές, ακολουθεί η κατασκευή του διμερούς πολυγραφήματος. Για το διμερές πολυγράφημα έχουμε την εξής δομή:

```
typedef struct setstr{
        int JoinNode;
        int EdgeLoad;
        int PositionNumber;
        int B_Color[MAX];
        struct setstr *VtoV;
        struct setstr *VtoH1;
        struct setstr *VtoH2;
        struct setstr *next;
         struct setstr *prev;
}*set;

set V1[MAX], V2[MAX];
```

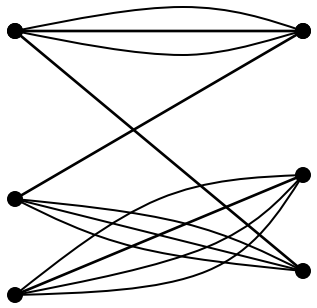
το V1 είναι το ένα σύνολο (αριστερό) του διμερούς γραφήματος για το οποίο δεν υπάρχει ακμή που συνδέει τους κόμβους του, ενώ V2 είναι το δεύτερο σύνολο (δεξί) για το οποίο ισχύει το ίδιο. Έχουμε για παράδειγμα ότι V1[0] είναι ο κόμβος 0 του συνόλου V1.

Όσον αφορά την παραπάνω δομή JoinNode είναι ο η ακμή που συνδέει δύο κόμβους. EdgeLoad είναι η πολλαπλότητα της συγκεκριμένης ακμής. PositionNumber είναι ένας ακέραιος που μας βοηθάει στο να γνωρίζουμε η ακμή που είμαστε σε πιο μεγάλο μονοπάτι αντιστοιχεί. B_Color[] είναι ο πίνακας που θα χρωματιστεί. Είναι πίνακας και όχι μόνο ακέραιος, επειδή έχουμε ακμές με μεγαλύτερη του ενός πολλαπλότητα. Οι δείκτες next και prev δείχνουν στον επόμενο και στον προηγούμενο κόμβο της λίστας. Ο δείκτης VtoV, δείχνει στο αντίστοιχο στοιχείο του V2. Έτσι για παράδειγμα αν ο κόμβος 0 του V1 συνδέεται με τον κόμβο 1 του V2 τότε έχουμε έναν

δείκτη που δείχνει σ' αυτόν τον κόμβο και αντιθέτως. Για τους δύο δείκτες που μένουν, τον VtoH1 και VtoH2 δες το επόμενο κεφάλαιο.

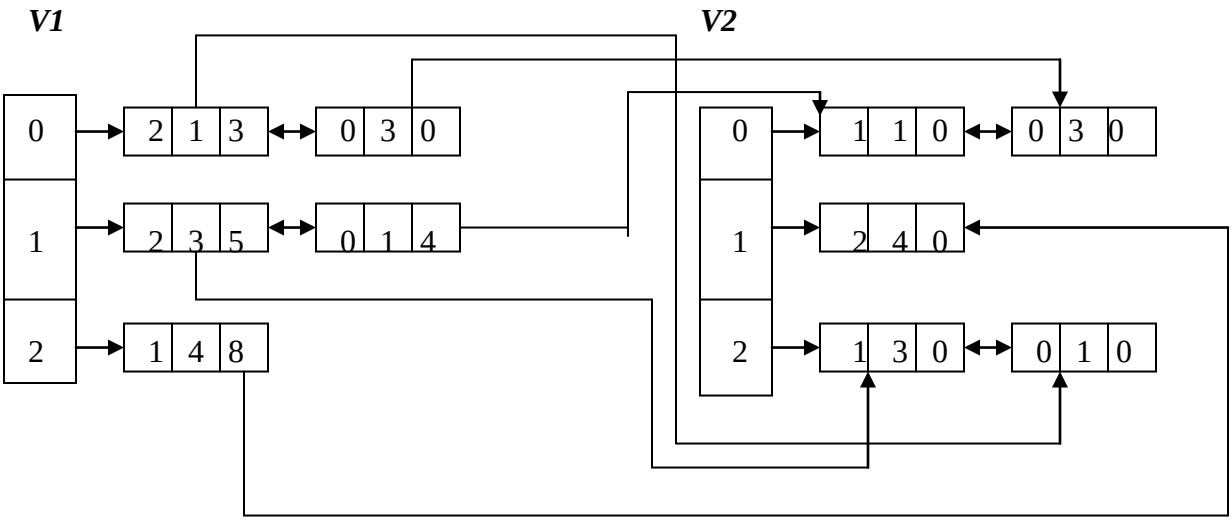
Όπως γίνεται κατανοητό γεμίζοντας το V1 έχουμε όλη την πληροφορία για το γράφημα. Το V2 δηλαδή, αποτελεί επανάληψη πληροφορίας. Παρόλα αυτά θέλουμε και το V2 και κάθε φορά που χρειάζεται, παρακάτω, να φτιάξουμε ένα νέο γράφημα δημιουργούμε και τα δύο σύνολα..

Για να βρούμε το ζητούμενο διμερές γράφημα δουλεύουμε ως εξής: στη συλλογή 0 που αρχίζει έχουμε το μονοπάτι 3. αυτό το μονοπάτι καταλήγει στη συλλογή 2 που τελειώνει άρα θα ενώσουμε τον κόμβο 0 του διμερούς γραφήματος με τον κόμβο 2. τα μονοπάτια 2, 1, 0 της συλλογής 0 που αρχίζει καταλήγουν στη συλλογή 0 που τελειώνει. Άρα θα βάλουμε μία ακμή που συνδέει τον κόμβο 0 του συνόλου V1 με τον κόμβο 0 του συνόλου V2 με πολλαπλότητα 3 γιατί έχουμε τρία μονοπάτια που ξεκινάν από την συλλογή 0 που αρχίζει και καταλήγουν στη συλλογή 0 που τελειώνει. Με αυτό τον τρόπο κατασκευάζουμε το γράφημα του σχήματος 3.8.



Σχήμα 3.8. Το διμερές γράφημα που προκύπτει μετά το τέλος του 4^{ου} βήματος.

Στο επόμενο σχήμα φαίνεται πώς είναι η αναπαράσταση που κρατάει ο υπολογιστής στη μνήμη. Φαίνεται καθαρά ότι έχουμε επανάληψη πληροφορίας. Από το παρακάτω σχήμα παρατηρούμε ότι ο κόμβος 0 του V1 συνδέεται με τον κόμβο 2 του V2 με πολλαπλότητα 1 και με τον κόμβο 0 του V2 με πολλαπλότητα 3.



Σχήμα 3.9. Η αναπαράσταση του γραφήματος που κρατάμε στον υπολογιστή.

Στο πέμπτο βήμα του αλγορίθμου πετυχαίνεται ο χρωματισμός με w = 4 χρώματα των ακμών του διμερούς γραφήματος. Η μέθοδος που χρησιμοποιείται για τον χρωματισμό αναλύεται στο επόμενο κεφάλαιο.

Τέλος αφού γίνει και ο χρωματισμός του διμερούς γραφήματος κάθε μεγάλο μονοπάτι θα πρέπει να πάρει ένα χρώμα. Για να γίνει αυτό έχουμε χρησιμοποιήσει το πεδίο PositionNumber στη δομή που έχουμε το γράφημα.

Τέλος σε κάθε μονοπάτι του αρχικού συνόλου ανατίθεται το αντίστοιχο χρώμα που έχει πάρει στο μεγάλο μονοπάτι.

3.3 Ανάλυση του Αλγορίθμου.

Για να αποδείξουμε την ορθότητα του αλγορίθμου αρκεί να δείξουμε πως για κάθε στιγμιότυπο I ο αλγόριθμος επιστρέφει μια λύση με κόστος LB(I), αφού αυτό είναι το κάτω φράγμα για το OPT(I) .

Έστω ότι με $b_υ$, $e_υ$ συμβολίζουμε τον αριθμό των μονοπατιών που ξεκινάνε (αντίστοιχα τελειώνουν) από τον (στον) κόμβο υ του P΄. Μετά τα βήματα 1 και 2 του αλγορίθμου, για κάθε κόμβο υ:

- $|b_υ-e_υ|$ είναι η διαφορά των βαρών δύο συνεχόμενων ακμών, γι' αυτό το λόγο είναι ακέραιο πολλαπλάσιο του w.
- $b_υ = 0$ ή $e_υ = 0$.

Γι' αυτό το λόγο για κάθε υ, και το $b_υ$ και $e_υ$ είναι πολλαπλάσια του w. Άρα μετά το βήμα 3 κάθε συλλογή στο B (αντίστοιχα στο E) περιέχει ακριβώς w μονοπάτια.

Στο βήμα 5, καθένα από τα w διαθέσιμα χρώματα χρησιμοποιείται ακριβώς μια φορά για κάθε συλλογή από μονοπάτια. Αυτό μας εγγυάται μια ομοιόμορφη κατανομή των χρωμάτων ανάμεσα στα μονοπάτια που χρησιμοποιούν μια ακμή. Γι' αυτό το λόγο για κάθε ακμή e κάθε χρώμα χρησιμοποιείται ακριβώς L(e,P)/w φορές (το L(e,P) με το w διαιρούνται ακριβώς).

Ο αριθμός των νέων μονοπατιών στο βήμα 1 επιλέχθηκε με τέτοιο τρόπο ώστε L(e,P΄)/w= [L(e,P)/w]. Άρα η το κόστος της λύσης είναι $\sum_{eεE}$ L(e,P΄)/w = $\sum_{eεE}$ [L(e,P)/w] = LB(I), που είναι βέλτιστο.

Έστω m = |P| και m' = |P'|. Τα βήματα 1,2 και 6 χρειάζονται O((m+n*w)). Παρατηρούμε ότι οι λίστες από αρχή και τέλος μονοπατιών για κάθε κόμβο μπορούν να κατασκευαστούν σε χρόνο γραμμικό ως προς τον αριθμό των μονοπατιών (συν O(n) για την αρχικοποίηση, αν υπάρχουν λιγότερα του n μονοπατια). Τα βήματα 3 και 4 απαιτούν O(m'+n). Η πολυπλοκότητα του βήματος 5 εξαρτάται από στον αλγόριθμο που θα χρησιμοποιηθεί για τον χρωματισμό ακμών σε διμερές γράφημα. Αν το w παίρνει μικρές τιμές, για παράδειγμα w = O(logm'), τότε ο πιο αποτελεσματικός αλγόριθμος είναι αυτός του Schrijver [21]. Με χρήση αυτού του αλγορίθμου η πολυπλοκότητα του βήματος 5 είναι O(m'*w). Αφού m' = m + O(n*w), η πολυπλοκότητα του αλγορίθμου για PMC (και για RPMC) είναι O((m+n*w)*w).

Απ' την άλλη πλευρά αν το w είναι μεγαλύτερο, μπορούμε να χρησιμοποιήσουμε έναν άλλο αλγόριθμο, των Cole και Hopcroft [3], που επιτυγχάνει πολυπλοκότητα χρόνου ίση με O(m'*logm'). Σ' αυτή την περίπτωση η συνολική πολυπλοκότητα είναι O((m + n*w)*log(m+n*w)).

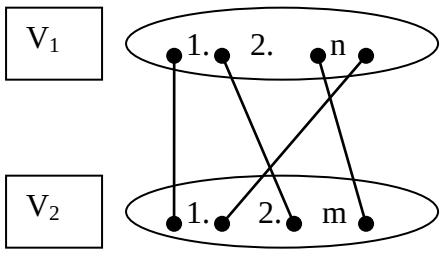
Θα πρέπει να παρατηρήσουμε ότι στη χειρότερη περίπτωση m’ = m + Θ(n*w). Ας πάρουμε για παράδειγμα το επόμενο στιγμιότυπο: η αλυσίδα περιέχει ζυγό αριθμό από κόμβους 1,2,…,n και υπάρχουν m = w + n/2 μονοπάτια με w+1 μονοπάτια να αρχίζουν από τον κόμβο 1 και ένα μονοπάτι αρχίζει από κάθε έναν από τους περισσευούμενους μονούς κόμβους, ενώ ένα μονοπάτι τελειώνει σε ζυγό κόμβο, εκτός από τον κόμβο n στον οποίο τελειώνουν w+1 μονοπάτια. Είναι εύκολο να ελέγξουμε ότι τα βάρη των ακμών κυμαίνονται από w σε w+1, που έχει σαν αποτέλεσμα την πρόσθεση n(w-1)/2 νέα μονοπάτια. Επίσης καμία σύνδεση δεν εκτελείται στο δεύτερο βήμα. Γι’ αυτό το λόγο ο αριθμός των μονοπατιών μπορεί να αυξηθεί περισσότερο από έναν σταθερό παράγοντα εξαιτίας των δύο πρώτων βημάτων.

4. Χρωματισμός ακμών σε διμερή γραφήματα.

4.1 Ορισμοί.

Πριν παρουσιάσουμε τους αλγόριθμους για χρωματισμό ακμών σε διμερή γραφήματα, στους οποίους συνεχώς αναφερόμασταν στους παραπάνω αλγορίθμους, θα πρέπει να αναφερθούμε σε ορισμένους ορισμούς οι οποίοι θα μας βοηθήσουν στην πλήρη κατανόηση των αλγορίθμων αυτών.

Έχουμε ήδη αναφέρει ότι ένα γράφημα συμβολίζεται ως G = (V, E) όπου V είναι το σύνολο των κόμβων και E το σύνολο των ακμών. Ένα διμερές γράφημα συμβολίζεται ως G = (V_1, V_2, E) όπου τα V_1, V_2, είναι σύνολα κόμβων τα οποία δεν συνδέονται μεταξύ τους, και E είναι το σύνολο των ακμών με $E \leq V_1 \times V_2$,όπως φαίνεται στο σχήμα 5.1. Με D συμβολίζουμε το μέγιστο βαθμό από όλους τους κόμβους στο $V = V_1 \cup V_2$ και με M συμβολίζουμε το σύνολο των κόμβων που έχουν βαθμό D.



Σχήμα 6.1. Διμερές γράφημα

Ένα γράφημα λέγεται κανονικό (regular), αν όλοι οι κόμβοι του έχουν τον ίδιο βαθμό. Ένα διμερές γράφημα λέμε ότι είναι ημικανονικό, αν όλοι οι κόμβοι του V_1 έχουν τον ίδιο βαθμό D, που είναι ο μέγιστος βαθμός από όλους τους κόμβους στο G.

Ταίριασμα (matching), $N \leq E$, είναι ένα υποσύνολο των ακμών που έχει την ιδιότητα ότι δεν υπάρχουν δύο ακμές που να έχουν το κοινό τέλος. Ένα ταίριασμα λέμε ότι *καλύπτει* (cover) ένα σύνολο κόμβων, U, αν κάθε κόμβος στο U είναι κόμβος τέλους ακμής στο N.

Euler partition (Διαμέριση Euler) είναι ένα partition (διαμέριση) των ακμών σε ανοικτά και κλειστά μονοπάτια, έτσι ώστε ο κάθε κόμβος με περιττό βαθμό είναι στο τέλος ακριβώς ενός ανοιχτού μονοπατιού και κάθε κόμβος ζυγού βαθμού δεν είναι στο τέλος κανενός ανοικτού μονοπατιού. Πρέπει να αναφέρουμε ότι Euler partitions (διαμερίσεις Euler) υπάρχούν σε όλα τα γραφήματα και όχι μόνο σε διμερή.

Euler split (Διαχωρισμός Euler) ενός διμερούς γραφήματος $G = (V_1, V_2, E)$ είναι ένα ζευγάρι διμερών γραφημάτων $G_1 = (V_1, V_2, E_1)$, $G_2 = (V_1, V_2, E_2)$, όπου τα E_1 και E_2 σχηματίζονται από ένα Euler partition (διαμέριση Euler) του E τοποθετώντας εναλλακτικά μονοπάτια στα E_1 και E_2. Κάθε κόμβος με ζυγό βαθμό στο G θα έχει τον ίδιο βαθμό στα G_1 και G_2, ενώ κάθε κόμβος με περιττό βαθμό στο G θα έχει βαθμούς στο G_1 και G_2 που θα διαφέρουν κατά ένα. Αυτό σημαίνει ότι, αν το D είναι ζυγός, τότε όλοι οι κόμβοι στο M έχουν βαθμό D/2 στα G_1 και G_2 και αυτός είναι και ο μέγιστος βαθμός στα G_1 και G_2. Υπάρχει αλγόριθμος ο οποίος βρίσκει ένα Euler split (διαχωρισμός Euler) σε χρόνο O(E) [5]. Euler splits (Διαχωρισμοί Euler), όπως ορίσθηκαν παραπάνω, υπάρχουν μόνο σε διμερή γραφήματα.

Ένα partition (διαμέριση) ενός διμερούς γραφήματος $G = (V_1, V_2, E)$ είναι ένα ζευγάρι διμερών γραφημάτων $G_1 = (V_1, V_2, E_1)$, $G_2 = (V_1, V_2, E_2)$, όπου τα E_1 και E_2 είναι μη συνδεδεμένα και η ένωση τους μας δίνει το E. Το partition (διαμέριση) περιέχει το M (M-containing), αν οι κόμβοι, που έχουν μέγιστο βαθμό στο G_1, περιέχουν το M και αν οι κόμβοι που έχουν μέγιστο βαθμό στο G_2, περιέχουν το M, όπου M, όπως έχουμε αναφέρει, είναι το σύνολο των κόμβων που έχουν το μέγιστο βαθμό στο G.

Ένας χρωματισμός ακμών ενός γραφήματος συνδέει ένα χρώμα με κάθε ακμή στο γράφημα, έτσι ώστε να μην υπάρχουν δύο ακμές με το ίδιο χρώμα που να έχουν το

ίδιο τέλος. Έχει δειχθεί [1] ότι κάθε διμερές γράφημα έχει ελάχιστο χρωματισμό ακμών με D χρώματα, όπου D είναι ο μέγιστος βαθμός όλων των κόμβων του γραφήματος.

4.2 Αλγόριθμος για την εύρεση ενός Διαχωρισμού Euler (Euler split).

Πριν προχωρήσουμε στους δύο αλγορίθμους για την εύρεση ταιριάσματος σε διμερή γραφήματα, θα πρέπει να αναφερθούμε σε έναν τρόπο εύρεσης ενός διαχωρισμού Euler (Euler split), καθώς και οι δύο οι αλγόριθμοι που θα εξετάσουμε στη συνέχεια αλλά και ο αλγόριθμος χρωματισμού διμερών γραφημάτων κάνει χρήση του διαχωρισμού Euler (Euler split). Σ' αυτό το σημείο θα πρέπει να αναφέρουμε ότι ο παρακάτω αλγόριθμος έχει αναπτυχθεί για πολυγραφήματα.

Έστω ότι έχουμε το διμερές γράφημα $G(V_1, V_2, E)$ του οποίου οι ακμές μπορούν να έχουν πολλαπλότητα μεγαλύτερη του 1 (δηλαδή, για παράδειγμα, από τον κόμβο i στον κόμβο j να μην υπάρχει μόνο μία ακμή αλλά n). Θέλουμε να πάρουμε από το G δύο γραφήματα $G_1(V_{11}, V_{12}, E_1)$, $G_2(V_{21}, V_{22}, E_2)$

Αρχίζουμε από έναν κόμβο με περιττό βαθμό:

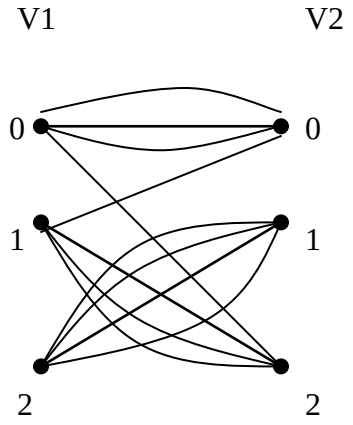
- Αν η ακμή που θα διασχίσουμε έχει πολλαπλότητα 2κ+1, έστω από τη μεριά του V_1 βάζω στο V_{11} πλευρά πολλαπλότητας κ+1 και στο V_{21} πλευρά πολλαπλότητας κ και αλλάζω μεριά, δηλαδή από το V_1 πάω στο V_2.
- Αν η ακμή που θα διασχίσουμε έχει πολλαπλότητα 2κ, τότε ενεργούμε όπως στην παραπάνω περίπτωση, αλλά δεν αλλάζουμε μεριά και συνεχίζουμε από το V_1.

Και στις δύο περιπτώσεις διαγράφουμε την ακμή και την συμμετρική της. Στη συνέχεια ψάχνουμε για άλλον κόμβο με περιττό βαθμό και ακολουθούμε τα παραπάνω βήματα. Αν δεν υπάρχει κόμβος με περιττό βαθμό, τότε επιλέγουμε έναν κόμβο με άρτιο βαθμό. Επίσης σ' αυτό το σημείο θα πρέπει να αναφέρουμε ότι, αφού τελειώσουμε με όλους τους κόμβους, στη συνέχεια θα πρέπει να φτιάξουμε και τα V_{12} και V_{22} από τα V_{11} και V_{21} .

Καταλήγοντας θα πρέπει να πούμε ότι φυσικά κάθε φορά που θέλουμε να πάμε από έναν κόμβο του V_1 στον αντίστοιχο του V_2 αυτό θα πρέπει να γίνεται σε χρόνο O(1) και όχι κάθε φορά να αναζητούμε τον αντίστοιχο κόμβο, γιατί αλλιώς ξεφεύγουμε από τον χρόνο που απαιτούμε να έχει ο συγκεκριμένος αλγόριθμος, που είναι O(E). Ένας τρόπος για να γίνει αυτό είναι από κάθε κόμβο του V_1 να υπάρχει ένας δείκτης στους αντίστοιχους κόμβους του V_2 [5].

4.3 Παράδειγμα Διαχωρισμού Euler (Euler Split).

Έστω ότι έχουμε το γράφημα του σχήματος 4.1, το οποίο προκύπτει από τους μετασχηματισμούς που κάναμε στο προηγούμενο κεφάλαιο. Θέλουμε να βρούμε ένα Euler Split του γραφήματος.

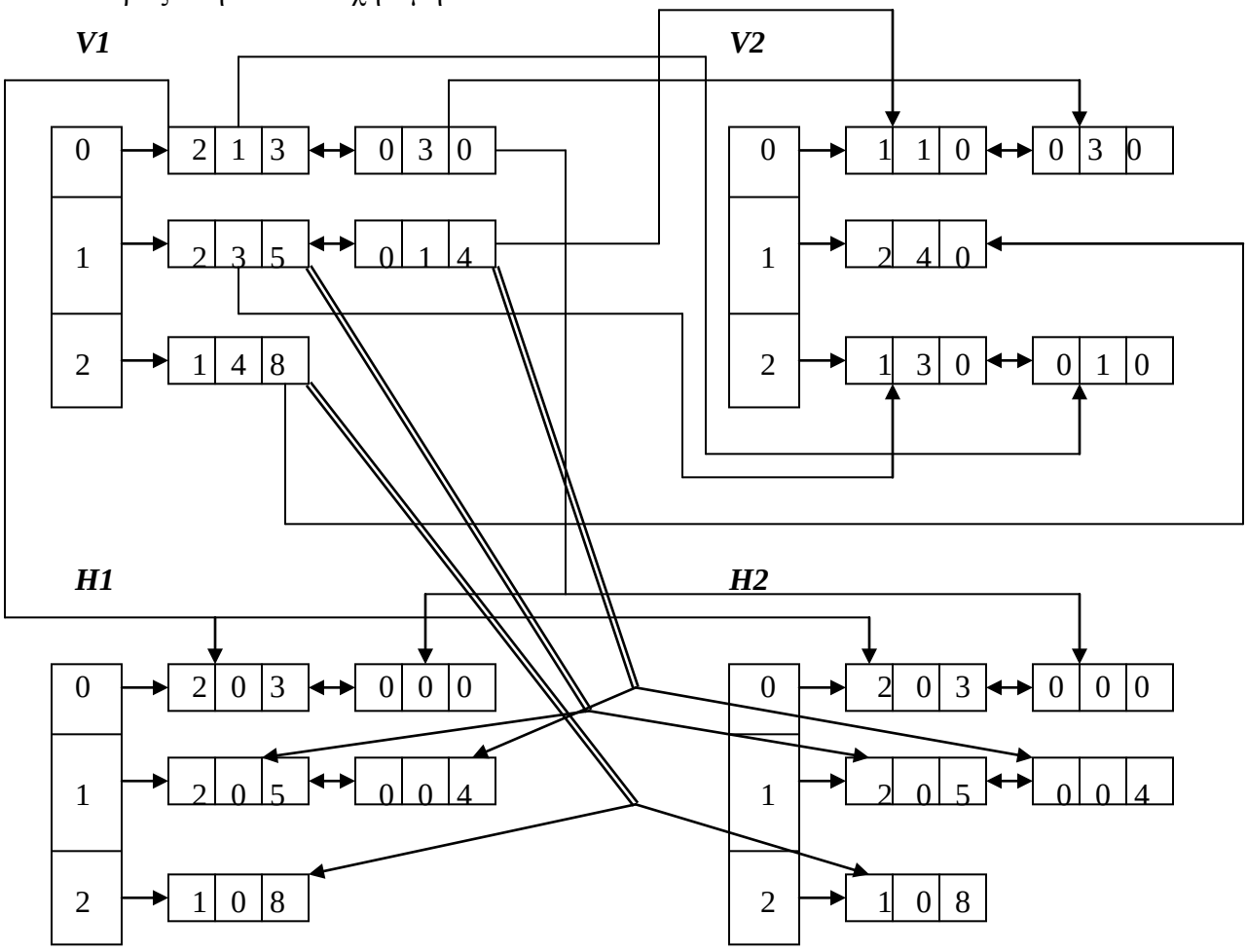


Σχήμα 4.1. Το γράφημα το οποίο προκύπτει από τους μετασχηματισμούς του κεφαλαίου 3.

Κάνουμε αντιγραφή του V1 σε δύο πίνακες λιστών τους H1, H2. Βάζοντας πολλαπλότητα 0 σε κάθε κόμβο. Τα H1 και H2 είναι τα δύο γραφήματα που μας δίνει το Euler Split. Στην δομή:

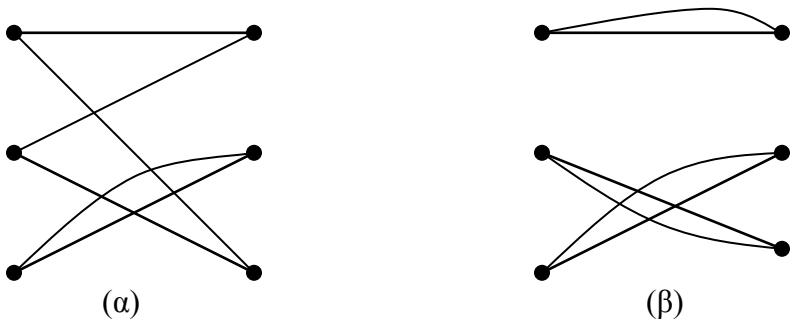
```
typedef struct setstr{
   int JoinNode;
   int EdgeLoad;
   int PositionNumber;
   int B_Color[MAX];
   struct setstr *VtoV;
   struct setstr *VtoH1;
   struct setstr *VtoH2;
   struct setstr *next;
   struct setstr *prev;
}*set;
```

υπάρχουν ακόμη δύο δείκτες, οι VtoH1 και VtoH2, τους οποίους τους χρησιμοποιούμε για να συνδέσουμε κάθε κόμβο του V1 με τον αντίστοιχο κόμβο στα H1 και H2, όπως φαίνεται στο παρακάτω σχήμα. Με αυτό τον τρόπο, όταν βρω την κατάλληλη πολλαπλότητα που πρέπει να μπει σε κάθε κόμβο, πηγαίνω κατευθείαν στα H1 και H2 και βάζω την αντίστοιχη τιμή.



Σχήμα 4.2. οι πίνακες λιστών που δημιουργεί το πρόγραμμα.

Θα ακολουθήσουμε τα βήματα του αλγορίθμου που αναπτύξαμε παραπάνω. Παρατηρούμε ότι δεν υπάρχει κόμβος με περιττό βαθμό, άρα θα αρχίσουμε από κάποιον με άρτιο βαθμό. Έστω ότι επιλέγουμε τον κόμβο 0 του V1. Ο κόμβος 0, όπως φαίνεται από το παραπάνω σχήμα, συνδέεται με τον κόμβο 2 του V2 με πολλαπλότητα 1. Άρα πηγαίνουμε στο H1 (μέσω του δείκτη VtoH1) και βάζουμε πολλαπλότητα 1 στον αντίστοιχο κόμβο και στον αντίστοιχο του H2 αφήνουμε την πολλαπλότητα 0. Εφόσον η πολλαπλότητα είναι περιττή, μεταβαίνουμε στο V2 και πηγαίνουμε σε ένα κόμβο με τον οποίο συνδέεται ο κόμβος 2 του V2 όπου βρισκόμαστε. Στη δικιά μας περίπτωση είναι ο μηδέν. Το πρόγραμμα ελέγχει αν ο επόμενος κόμβος είναι NULL. Αν είναι, τότε παίρνουμε τον προηγούμενο. Γι' αυτό το λόγο θέλουμε διπλά συνδεδεμένες λίστες και όχι απλά συνδεδεμένες. Φυσικά αν δεν έχουμε επόμενο ή προηγούμενο κόμβο, τότε έχουμε τελειώσει με το μονοπάτι οπότε πρέπει να αρχίσουμε από άλλο κόμβο του V1. Κάθε φορά σβήνουμε από τα V1 και V2 τους κόμβους που έχουμε επισκεφτεί ώστε την επόμενη φορά που θα ελέγξουμε να μην υπάρχουν κόμβοι (στις λίστες) που έχουμε ήδη επισκεφτεί. Συνεχίζοντας, είμαστε στον κόμβο 2 του V2 ο οποίος, όπως είπαμε, συνδέεται με τον κόμβο 1 του V1 με πολλαπλότητα 3, άρα μέσω των δεικτών VtoV (για να πάμε στον αντίστοιχο κόμβο του V1) και VtoH2 (για να μεταβούμε στον αντίστοιχο κόμβο του H2) βάζουμε στο H2 πολλαπλότητα 2 και στο H1 πολλαπλότητα 1. Πάλι η πολλαπλότητα είναι περιττή οπότε συνεχίζουμε από το V1. Αυτό επαναλαμβάνεται μέχρι τα V1 και V2 να γίνουν κενά. Στο παρακάτω σχήμα φαίνονται τα γραφήματα που προκύπτουν από το Euler Split.



Σχήμα 4.3. (α) Το γράφημα H1, (β) το γράφημα H2

Κλείνοντας, πρέπει να αναφέρουμε ότι μετά το τέλος του Euler Split τα δύο γραφήματα θα περιέχουν μερικές ακμές με πολλαπλότητα 0. Πρέπει να σβήσουμε αυτές τις ακμές από τα H1 και H2.

4.4 Πρώτος αλγόριθμος για την εύρεση ταιριάσματος σε διμερή γραφήματα.

Ο αλγόριθμος ο οποίος παρουσιάζεται σε αυτή την παράγραφο τρέχει σε χρόνο O(ElogV). Έστω G το γράφημα στο οποίο θέλουμε να βρούμε ένα ταίριασμα (matching). Εστω M το σύνολο κόμβων που έχουν μέγιστο βαθμό. Αν ο μέγιστος βαθμός όλων των κόμβων του G είναι ένα, τότε το E, που είναι το σύνολο των ακμών του G, είναι το ζητούμενο ταίριασμα. Διαφορετικά (max degree $\neq$ 1) γίνεται ένα partition (διαμέριση) του G σε δύο γραφήματα $G_1 = (V_1, V_2, E_1)$ και $G_2 = (V_1, V_2, E_2)$ όπου τα E_1 και E_2 δεν είναι κενά, $E_1, E_2 \neq Ø$. Ο αλγόριθμος στη συνέχεια εφαρμόζεται περιοδικά στο γράφημα με το μικρότερο σύνολο ακμών από τα G_1 και G_2.

Για να κάνουμε partition (διαμέριση) το G πραγματοποιείται ένας Euler split (διαχωρισμός Euler), ο οποίος μας δίνει τα γραφήματα H_1 και H2. Αν το D, ο μέγιστος βαθμός όλων των κόμβων, είναι ζυγός τότε οι κόμβοι στο M έχουν βαθμούς D/2 και στο H_1 και στο H_2, άρα τα H_1, H_2 μας δίνουν ένα partition του G που περιέχει το M (M-containing). Αν όμως το D είναι περιττός, τότε μετακινούμε ακμές ανάμεσα στα H_1 και H_2, έτσι ώστε οι κόμβοι στο H_1 να έχουν μέγιστο βαθμό το D_1 και οι κόμβοι στο H_2 να έχουν μέγιστο βαθμό το D_2 και όλοι οι κόμβοι στο M να έχουν βαθμό D_1 στο H_1 και D_2 στο H_2 για κάποιο ζεύγος (D_1, D_2) με $D_1 + D_2 = D$. Η μέθοδος για μετακίνηση ακμών περιγράφεται παρακάτω.

Αν το D είναι περιττός και $D = 4r + e$ με $e = \pm 1$, τότε κάθε κόμβος στο M έχει βαθμό 2r σε ένα από τα H_1 ή H_2 και βαθμό $2r + e$ στο άλλο. Έτσι σε ένα από τα H_1 ή H_2 τουλάχιστο οι μισοί κόμβοι στο M έχουν βαθμό $2r + e$. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι στο H_1 οι μισοί τουλάχιστο κόμβοι στο M έχουν βαθμό $2r + e$. Έστω M_1 το σύνολο των κόμβων από το M που έχουν βαθμό $2r + e$ στο H_1 (και άρα βαθμό 2r στο H2) και έστω M_2 οι κόμβοι που περισσεύουν από το M.

Γενικά θα έχουμε την περίπτωση όπου οι κόμβοι στο M_1 έχουν βαθμό 2k στο H_2 και βαθμό D – 2k στο H_1, ενώ οι κόμβοι στο M_2 έχουν βαθμό D – 2k στο H_1 και βαθμό 2k + d στο H_2 με d = ± 1.

Κάνουμε ένα Euler split (διαχωρισμό Euler) του H_2 το οποίο μας δίνει τα γραφήματα H_{21} και H_{22}. Οι κόμβοι από το M_1 έχουν βαθμό k και στο H_{21} και στο H_{22}. Μερικοί κόμβοι από το M_2 έχουν βαθμό k στο H_{21} και βαθμό k + d στο H_{22}, ενώ άλλοι έχουν βαθμό k + d στο H_{21} και βαθμό k στο H_{22}. Χωρίς βλάβη της γενικότητας ας υποθέσουμε ότι στο H_{21} τουλάχιστο οι μισοί από τους κόμβους του M_2 έχουν βαθμό k + d (αλλιώς αλλάζουμε τις ετικέτες των H_{21} και H_{22}). Έστω M_{21} οι κόμβοι του M_2 που έχουν βαθμό k + d στο H_{21} (και επομένως βαθμό k στο H_{22}), και έστω M_{22} οι εναπομείναντες κόμβοι του M_2. Οι κόμβοι του M_2 έχουν βαθμό k στο H_{21} και βαθμό k + d στο H_{22}.

Στο γράφημα που προκύπτει από την ένωση των H_1 και H_2 (H_1 U H_2) οι κόμβοι στο M_1 U M_{21} έχουν βαθμό D – k και οι κόμβοι στο M_{22} έχουν βαθμό D – k -d. Στο γράφημα H_{22} οι κόμβοι στο M_1 U M_{21} έχουν βαθμό k και οι κόμβοι στο M_{22} έχουν βαθμό k + d. Το M_1 γίνεται ίσο με το M_1 U M_{21} και το M_2 γίνεται ίσο με το M_{22}, που μειώνει το μέγεθος του M_2 κατά ένα παράγοντα της τάξης τουλάχιστο δύο, και τα H_1 U H_{21} και H_{22} γίνονται πλέον τα H_1 και H_2.

Η διαδικασία επαναλαμβάνεται μέχρι M_1 = M, όταν οι κόμβοι του M έχουν όλοι το ίδιο περιττό μέγιστο βαθμό στο H_1 και τον ίδιο ζυγό μέγιστο βαθμό στο H_2. Τα H_1 και H_2 μας δίνουν το επιθυμητό partition (διαμέριση) του G σε G_1 και G_2. Στη συνέχεια ακολουθεί ο αλγόριθμος της παραπάνω διαδικασίας στην περίπτωση που το D είναι περιττός αριθμός.

```
procedure PARTITION
begin
    comment. This procedure partitions the bipartite graph G = (V1, V2, E)
      into bipartite graphs G1 = (V1, V2, E1) and G2 = (V1, V2, E2), in the
      case that D, the maximum degree of any vertex in V1 U V2, is one of
      the form 4r + e, e = ± 1.
    M = set off maximum degree vertices in G;
    Let H1, H2 be an Euler split of G;
    Comment. At least half the vertices in M have degree 2r + e in one of H1
      or H2. Let it be in H1. Let k = r, d =e.
    Let M1 = {υ | υεM , and υ has degree 2r + e in H1};
    Let M2 = M – M1;
    while (|M2| ≠ 0) do
    begin
        comment. The vertices of M1 have degree D – 2k in H1 and
          degree 2k in H2, while the vertices in M2 have degree D – 2k – d
          in H1, and degree2k + d in H2, for some d = ± 1.
        Let H21, H22 be an Euler split of H2;
        comment. At least half the vertices in M2 have degree k + d in
          either H21 or H22. Let it be in H21.
        Let M21 = {υεM2 and υ has degree k + d in H21}
        Let M22 = M2 – M21;
        if k is even
            then H1 = H1 U H21, H2 = H22;
            else H1 = H22, H2 = H1 U H21;
        M1 = M1 U M21, M2 = M22;
        comment. If k is even then the vertices in M1 have degree D – k
          in H1 and degree k in H2, while the vertices in M2 have degree
          D – k –d in H1 and degree k + d in H2. Set  k =k/2.
          Otherwise k is odd and the vertices in M1 have degree k in H1
          and degree D – k in H2, while the vertices in M2 have degree
          k+ d in H1 and degree D – k –d in H2.
        Then set k = (D - k)/2 and d = - d.
    end
    G1 = H1, G2 = H2;
end
```

Σχήμα 5.1. Αλγόριθμος διαμέρισης (partition) για το γράφημα G.

Για τον παραπάνω αλγόριθμο υπάρχει απόδειξη της ορθότητας του από τους R. Cole και J. Hopcroft [3].

4.5 Δεύτερος αλγόριθμος για την εύρεση ταιριάσματος σε διμερή γραφήματα.

Ο δεύτερος αυτός αλγόριθμος τρέχει σε χρόνο $O(E+VlogV(logD)^2)$. Έστω G = (V_1, V_2, E) το διμερές γράφημα για το οποίο θέλουμε να κάνουμε ταίριασμα (matching) που επικαλύπτει το M. Με το να σβήσουμε μερικές ακμές από το E και αυξάνοντας την πολλαπλότητα άλλων ακμών κατασκευάζεται ένα πολύ-γράφημα (multigraph) H, όπου ο βαθμός κάθε κόμβου στο H είναι ο ίδιος με τον αντίστοιχο βαθμό στο G. Επιπλέον το H θα έχει μόνο VlogD πολύ-ακμές (multiedges). Είναι φανερό ότι το M είναι το σύνολο των κόμβων με τον μέγιστο βαθμό στο H. Επομένως υπάρχει ένα ταίριασμα στο H, που είναι επίσης και ταίριασμα στο G, που επικαλύπτει το M. Χρησιμοποιείται ο πρώτος αλγόριθμος για την εύρεση αυτού του ταιριάσματος. Πρέπει να αναφέρουμε ότι εξαιτίας των πολλαπλών ακμών στο H ο χρόνος περάτωσης είναι πιο γρήγορος από τον χρόνο που χρειάζεται ο πρώτος αλγόριθμος, που βρίσκει ταίριασμα κατευθείαν στο G.

Για να απλοποιήσουμε την χρονική ανάλυση κόμβοι με μικρό βαθμό ($\leq$ D/2) "συγχωνεύονται", έτσι ώστε, με τη εξαίρεση το πολύ δύο κόμβων, όλοι οι κόμβοι στο V1 U V2 έχουν βαθμούς μεταξύ [D/2] και D στο G. Τώρα E = O(V * D). Δύο κόμβοι u, v συγχωνεύονται και αντικαθίστανται με έναν κόμβο w, και όλες οι ακμές που είχαν σαν τελικό σημείο τον u ή τον v τώρα έχουν τον w. Για να κρατήσουμε την ιδιότητα του G, δηλαδή το ότι είναι διμερές, κόμβοι από V_1 δεν συγχωνεύονται με κόμβους από το V_2.

Το H το παίρνουμε από το G βρίσκοντας κύκλους ανάμεσα σε ακμές με διπλή πολλαπλότητα και αφαιρώντας τις άλλες ακμές. Διαδοχικά όλοι οι πιθανοί κύκλοι βρίσκονται. Μεταξύ των ακμών με πολλαπλότητα ένα, με πολλαπλότητα δύο, πολλαπλότητα τέσσερα και ούτω καθεξής μέχρι το πολύ με πολλαπλότητα $2^{[logD]}$. Έστω r = [logD]. Το γράφημα που άγεται από τις ακμές με πολλαπλότητα 2^r είναι άκυκλο, αφού κάθε κόμβος μπορεί να έχει το πολύ μία πολύ-ακμή (multiedge) και επομένως δεν υπάρχουν κύκλοι ανάμεσα σε αυτές τις πλευρές.

Για να βρούμε τους κύκλους ανάμεσα σε πλευρές με δοσμένη πολλαπλότητα κάνουμε μία DFS (Depth First Finding) διέλευση. Όταν βρίσκεται ένας κύκλος, οι πλευρές του αφαιρούνται από το DFS δέντρο και στις εναλλακτικές δίνεται διπλή πολλαπλότητα. Η DFS διέλευση συνεχίζεται από τον κόμβο που ήταν στη ρίζα του κύκλου. Όταν η διέλευση επιστρέφει από ένα φύλλο, επειδή η μόνη ακμή είναι σκελετική

ακμή (spanning edge) στον κόμβο, η σκελετική ακμή και ο κόμβος σβήνονται, αφού αποτελούν μέρος από ένα άκυκλο γράφημα. Επομένως σε κάθε στιγμή το DFS δέντρο αποτελείται από ένα απλό μονοπάτι από τη ρίζα του δέντρου μέχρι τον κόμβο που εξετάζεται αυτή τη στιγμή.

Όταν μία σκελετική ακμή διαγράφεται από το δέντρο της DFS διέλευσης, αυτή προστίθεται στο H με την κατάλληλη πολλαπλότητα. Επίσης το άκυλο γράφημα με πολλαπλότητα ακμών 2^r προστίθεται στο H, έτσι ώστε το H είναι μία ένωση το πολύ logD άκυκλων πολύ-γράφων με πλευρές πολλαπλότητας 1, 2, 4, … 2^r, αντιστοίχως. Το H έχει O(VlogD) πολύ-ακμές (multiedges).

Εμείς έχουμε γραφήματα με πολλαπλότητα μεγαλύτερη του ένα. Οπότε αναλύουμε την πολλαπλότητα σε άθροισμα δυνάμεων του δύο (κάθε δύναμη του δύο είναι ένα επίπεδο). Πρέπει να κάνουμε DFS για κάθε δύναμη του δύο που είναι μικρότερη ή ίση του μεγίστου βαθμού του γραφήματος. Στο παρακάτω κομμάτι κώδικα φαίνεται πώς γίνεται η κλίση της DFS διέλευσης - που σαν ορίσματα παίρνει το γράφημα και τον μετρητή που μας δείχνει σε ποιο επίπεδο είμαστε- από το πρόγραμμα, ενώ αμέσως μετά φαίνεται ο έλεγχος που κάνουμε για να δούμε αν η ακμή, την οποία ελέγχουμε, ανήκει στο επίπεδο που βρισκόμαστε κάθε φορά. Όπου tmp->EdgeLoad είναι η πολλαπλότητα της ακμής:

```
D = MaxDegree;
for(i = 1; i <= D; i *= 2){
        DFS(V_1, V_2, i);
}

if(((tmp->EdgeLoad)/i) % 2 != 0){
        /*Η ακμή ανήκει στο επίπεδο που είμαι*/
}
```

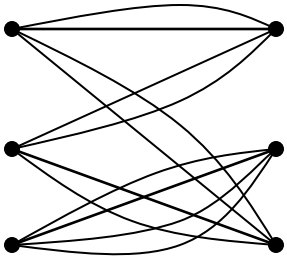
Από το H ένα ταίριασμα βρίσκεται με χρήση του πρώτου αλγορίθμου. Τέσσερα αντίγραφα κάθε ακμής υπάρχουν, καθένα σε ένα από τα H_1, H_2, H_{21}, H_{22}. Για να χρησιμοποιήσουμε αποτελεσματικά την πολύ-ακμή (multiedge), όταν κάνουμε ένα Euler split (διαχωρισμό Euler), κάθε ακμή κατασκευάζεται να βρίσκεται τόσο συχνά όσο είναι δυνατό στο σημείο μέσα στο μονοπάτι. Οι πολλαπλότητες των αντιγράφων των ακμών μεταβάλλονται ανάλογα. Αλλιώς χρησιμοποιείται ο πρώτος αλγόριθμος χωρίς αλλαγές.

Ο σκοπός που γίνεται η παραπάνω DFS διέλευση είναι ώστε το γράφημα που θα προκύψει να έχει λίγες ακμές με μεγάλη πολλαπλότητα. Έτσι ώστε όταν καλέσουμε μετά τον πρώτο αλγόριθμο για το ταίριασμα αυτό να γίνει πιο γρήγορα μιας και το Euler split που χρησιμοποιείται θα κατατάξει πιο γρήγορα τις ακμές.

4.6 Παράδειγμα εύρεσης ταιριάσματος.

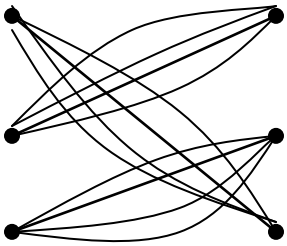
Στο σχήμα 4.1 φαίνεται το γράφημα για το οποίο θέλουμε να βρούμε ένα ταίριασμα των μεγιστοβάθμιων κόμβων του. Στο γράφημα αυτό, πριν κάνουμε χρήση του πρώτου αλγορίθμου για την εύρεση του ζητούμενου ταιριάσματος, θα πρέπει να κάνουμε DFS διέλευση και να σβήσουμε τους κύκλους που σχηματίζονται σε κάθε επίπεδο. Στην προκειμένη περίπτωση ο μέγιστος βαθμός είναι 4, άρα το D = 4. Θα καλέσουμε την DFS για i = 1, 2, 4.

Ξεκινάμε από τον πρώτο κόμβο του V1. Ο κόμβος 0 του V1 συνδέεται με τον κόμβο 2 του V2 με πολλαπλότητα 1. Άρα αυτή την ακμή την ελέγχουμε στο επίπεδο που βρισκόμαστε, εφόσον ο έλεγχος (1/1) % 2 != 0 είναι αληθής. Στη συνέχεια ο κόμβος 2 του V2 συνδέεται με τον κόμβο 1 του V1 με πολλαπλότητα 3- και αυτή η ακμή ανήκει στο επίπεδο που βρισκόμαστε. Συνεχίζοντας, ο κόμβος 1 του V1 συνδέεται με τον κόμβο 0 του V2 με πολλαπλότητα 1- και αυτή η ακμή τηρεί τις προϋποθέσεις, άρα ανήκει στον επίπεδο που είμαστε. Στο σημείο που βρισκόμαστε, παρατηρούμε ότι ο κόμβος 0 του V2 συνδέεται με τον κόμβο 0 του V1 με ακμή πολλαπλότητας 3, δηλαδή με ακμή που ανήκει στο επίπεδο 1. Άρα έχουμε βρει έναν κύκλο που θα πρέπει να σβήσουμε. Μειώνοντας και αυξάνοντας εναλλάξ τις πολλαπλότητες κατά i πηγαίνουμε πίσω στην αρχή του κύκλου, που βρίσκεται στον κόμβο 0 του V1. Έτσι η ακμή που κλείνει τον κύκλο έχει τώρα πολλαπλότητα 3-1 = 2. Η ακμή (1,0) έχει πολλαπλότητα 1+1 = 2 κοκ. Από την αρχή του κύκλου συνεχίζει η DFS, χωρίς όμως να μπορεί να επισκεφτεί κάποιον κόμβο, εφόσον δεν υπάρχει πλέον ακμή στο επίπεδο που είμαστε, που να συνδέει δύο κόμβους. Στο παρακάτω σχήμα φαίνεται πως έχει μεταβληθεί το γράφημά μας μετά από την DFS διέλευση στο επίπεδο 1.



Σχήμα 4.4. Το γράφημα που προκύπτει από την DFS διέλευση στο επίπεδο 1.

Αφού τελειώσουμε με το πρώτο επίπεδο, αυξάνουμε το i . Το i έχει τώρα την τιμή 1*2 = 2. Δουλεύοντας με τον ίδιο ακριβώς τρόπο στο νέο γράφημα ελέγχω ακμές που ανήκουν στο επίπεδο 2. Και πάλι βρίσκω κύκλο, καθώς από τον κόμβο 0 του V1 πάω στον κόμβο 2 του V2. Από εκεί πάω στον κόμβο 1 του V1 και καταλήγω στον κόμβο 0 του V2, όπου πάλι κλείνει κύκλος. Αλλάζω εναλλάξ τις πολλαπλότητες μειώνοντας και αυξάνοντας κατά 2 αυτή τη φορά, αφού είμαι στο δεύτερο επίπεδο, οπότε προκύπτει το γράφημα του σχήματος 4.5.

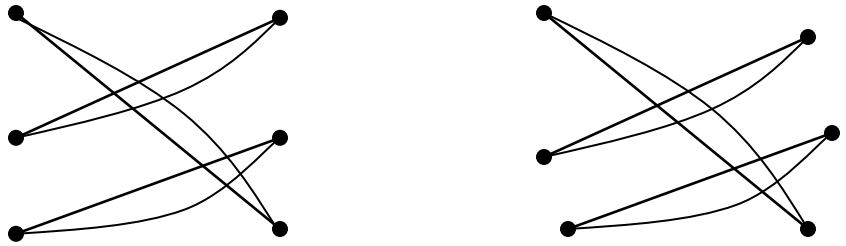


Σχήμα 4.4. Το γράφημα που προκύπτει από την DFS διέλευση στο επίπεδο 2.

Τέλος, πάλι αυξάνουμε το i , το οποίο γίνεται τώρα 2*2 = 4. Στο επίπεδο αυτό δεν σχηματίζονται κύκλοι, οπότε δεν αλλάζει τίποτα στο γράφημα που έχουμε.

Έχοντας πλέον βρει το γράφημα που δίνει η DFS πρέπει να κάνουμε χρήση του πρώτου αλγορίθμου για την εύρεση ενός ταιριάσματος των μεγιστοβάθμιων κόμβων. Ελέγχουμε πρώτα αν ο μέγιστος βαθμός είναι 1. Αν είναι, τότε έχουμε βρει το ζητούμενο ταίριασμα. Στη περίπτωσή μας ο μέγιστος βαθμός είναι 4, επομένως πρέπει να

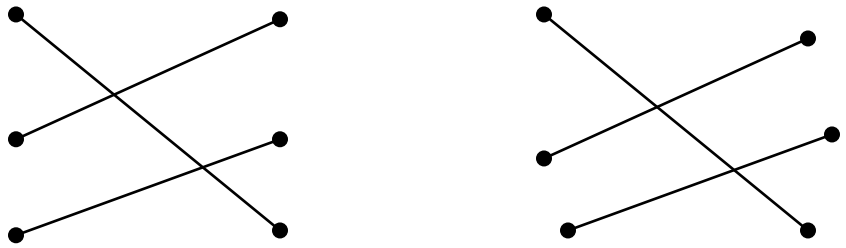
καλέσουμε την συνάρτηση partition για να βρούμε μία διαμέριση του αρχικού γραφήματος. Επειδή έχουμε άρτιο βαθμό, η partition απλώς κάνει ένα Euler Split και τα δύο γραφήματα που παίρνουμε είναι η ζητούμενη διαμέριση. Στο επόμενο σχήμα φαίνεται αυτή η διαμέριση.



(α) (β)

Σχήμα 4.4. Τα γραφήματα που προκύπτουν από το partition.

Ελέγχουμε πάλι αν κάποιο από τα γραφήματα έχει μέγιστο βαθμό ένα. Αν συμβαίνει αυτό, τότε έχουμε βρει και το ζητούμενο ταίριασμα. Αν δεν έχουμε μέγιστο βαθμό ένα σε κανένα γράφημα, παίρνουμε το γράφημα με τις λιγότερες πλευρές (στη περίπτωσή μας είναι και τα δύο ίδια) και κάνουμε πάλι partition. Το επόμενο partition, μας δίνει τα εξής γραφήματα:



(α) (β)

Σχήμα 4.5. Τα γραφήματα που προκύπτουν από το δεύτερο partition.

Τώρα έχουμε κάποιο γράφημα με μέγιστο βαθμό ένα (στην περίπτωσή μας έχουμε και τα δύο με μέγιστο βαθμό ένα και διαλέγουμε ένα από τα δύο). Επομένως έχουμε βρει το ζητούμενο ταίριασμα.

4.6 Χρωματισμός ακμών σε διμερή γραφήματα.

Στην παράγραφο αυτή παρουσιάζεται ένας αλγόριθμος εύρεσης ενός ελάχιστου χρωματισμού ακμών σε διμερή γραφήματα ο οποίος απαιτεί χρόνο O(ElogV). Έστω G το γράφημα τις ακμές του οποίου θέλουμε να χρωματίσουμε και έστω D ο μέγιστος βαθμός των κόμβων του G, τότε D χρώματα χρησιμοποιούνται για τον χρωματισμό. Όπως φαίνεται στον παρακάτω αλγόριθμο, με S συμβολίζεται η συλλογή των συνόλων των ακμών που χρωματίζονται όμοια.

```
procedure Color
begin
        if D is odd
                then using the second matching algorithm, find a matching N
                   covering vertices in M. Color the edges in N with one color and
                   delete N from G.
                S := SU{N}, D:=D-1;
        Make an Euler split of G to give two bipartite graphs G1 and G2, each
           having D/2 as a maximum degree of its vertices;
        Wlog assume G1 has a smaller edge set than G2
          (otherwise swap the labels of G1 and G2)
        Color(G1);
        Let 2k<D/2=2k+1-r. Add r sets of colored edges to G2, and delete them
           from S;
        Color(G2);
end
```

Σχήμα 5.2: Αλγόριθμος χρωματισμού ακμών σε διμερή γραφήματα.

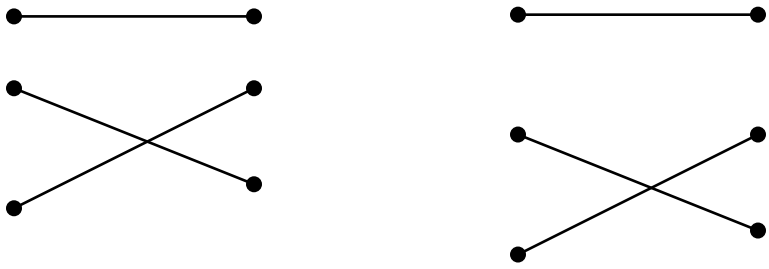
Στον παραπάνω αλγόριθμο η συνθήκη τερματισμού είναι για την πρώτη αναδρομή αν το G1 είναι NULL, και αντίστοιχα για το G2, αν το G2 είναι NULL. Δηλαδή το πρόγραμμα τρέχει όσο τα γραφήματα που δημιουργούνται δεν είναι κενά.

Μία παρόμοια μέθοδος χρωματισμού χρησιμοποιείται από τους H. Gabow και O. Kariv [6]. Η απόδειξη της χρησιμοποίησης ακριβώς D χρωμάτων για τον παραπάνω χρωματισμό μπορεί να γίνει με επαγωγή.

4.7 Παράδειγμα χρωματισμού ακμών σε διμερή γραφήματα.

Θέλουμε να χρωματίσουμε με D = 4 χρώματα το γράφημα του σχήματος 4.1. Το πρόγραμμα ελέγχει αν ο μέγιστος βαθμός είναι περιττός. Στην περίπτωση που εξετάζουμε ο μέγιστος βαθμός είναι άρτιος. Οπότε κάνουμε ένα Euler Split παίρνοντας τα γραφήματα του σχήματος 4.3. Από τα δύο αυτά γραφήματα το H2 είναι μικρότερο, οπότε αντιγράφουμε το H2 στο G1 και το H1 στο G2. Στη συνέχεια καλούμε αναδρομικά την ίδια συνάρτηση πρώτα για το G1 και στην συνέχεια για το G2.

Ας δούμε τι γίνεται με την αναδρομική κλήση της συνάρτησης Color για το G1. Πρέπει να βρούμε ποιος είναι ο μέγιστος βαθμός στο G1. Βρίσκουμε ότι ο μέγιστος βαθμός είναι 2,δηλαδή άρτιος. Άρα κάνουμε πάλι Euler Split και παίρνουμε τα εξής γραφήματα:



(α) (β)

Σχήμα 4.6. (α) Το γράφημα H1, (β) το γράφημα H2

Βάζουμε στο G1 το H1 και καλούμε πάλι αναδρομικά τη συνάρτηση για το νέο G1 και στη συνέχεια για το νέο G2. Ο μέγιστος βαθμός είναι περιττός, άρα θα βρούμε ένα ταίριασμα και θα χρωματίσουμε αυτό το ταίριασμα με ένα χρώμα. Στη συνέχεια σβήνουμε από το γράφημά μας τις ακμές του ταιριάσματος, μειώνουμε τον αριθμό των χρωμάτων κατά ένα και συνεχίζουμε μέχρι να χρωματιστούν όλες οι ακμές.

5. Πολυ-χρωματισμός σε Δακτυλίους και Αστέρες.

5.1 Ο αλγόριθμος PMC για δακτυλίους.

Πρώτα παρουσιάζουμε τον αλγόριθμο για PMC ο οποίος θα συνδυαστεί με ένα routing ελαχίστων μονοπατιών για να μας δώσει τον αλγόριθμο για το RPMC. Και οι δύο αλγόριθμοι πετυχαίνουν προσεγγιστικό παράγοντα 2.

Έστω I=(G,P,w) είναι ένα στιγμιότυπο του PMC στο οποίο το G είναι ένας δακτύλιος με n κόμβους αριθμημένους με τη φορά του ρολογιού από το 0 έως το n-1. Συμβολίζουμε το μονοπάτι από τον κόμβο u στον υ σύμφωνα με την κίνηση των δεικτών του ρολογιού με <u,υ> και λέμε ότι αρχίζει στον u και τελειώνει στον υ. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι όλες οι ακμές στο G χρησιμοποιούνται από κάποιο μονοπάτι (αλλιώς αφαιρούμε μια αχρησιμοποίητη ακμή και παίρνουμε ένα στιγμιότυπο αλυσίδας.). Για τον λόγο αυτό OPT(I)≥n. Δοθείσης μιας λίστας μονοπατιών P δακτυλίου συμβολίζουμε με $P_υ$ την λίστα των μονοπατιών στο P που περιέχουν τον υ σαν εσωτερικό κόμβο. Ορίζουμε το clockwise span (μέτρημα με την φορά του ρολογιού) d(υ) του κόμβου υ να είναι η μέγιστη απόσταση (με κατεύθυνση σύμφωνα με τη φορά του ρολογιού) από τον υ προς τελευταίο κόμβο κάθε μονοπατιού στο $P_υ$. Αν το $P_υ$ είναι κενό τότε d(υ) = 0.

Πρώτα ο αλγόριθμος υπολογίζει το clockwise span για όλους τους κόμβους και μετονομάζει τους κόμβους έτσι ώστε ο νέος κόμβος 0 να έχει το ελάχιστο clockwise span. Αντί για d(0) γράφουμε d για συντομία. Μετά ο αλγόριθμος μετατρέπει το δοσμένο στιγμιότυπο δακτυλίου σε στιγμιότυπο αλυσίδας (G',P',w). Το γράφημα αλυσίδα G' περιέχει n+d+1 κόμβους αριθμημένους από το 0 έως το n+d. Για κάθε μονοπάτι <i, j>εP, αν i<j, τότε και το P' περιέχει το <i, j>, αλλιώς το P' περιέχει το <i,j+n>. Βέλτιστος

πολυχρωματισμός πετυχαίνεται με χρήση του αλγορίθμου PMC για αλυσίδες. Τελικά κάθε μονοπάτι στο P παίρνει το χρώμα του αντίστοιχου μονοπατιού στο P'.

Πρόταση 2: ο αλγόριθμος για PMC σε δακτυλίους υπολογίζει έναν πολυχρωματισμό με κόστος OPT + d.

Απόδειξη: έστω Li = L({i, (i+1) mod n}, P) και Li' = L({i,i+1},P'). Είναι εύκολο να δούμε ότι, αν ένα μονοπάτι στο P χρησιμοποιεί την ακμή {i, i+1} και $0 \le i \le$ d-1, Li = Li' + Li+n'. Παρατηρούμε επίσης ότι Li = Li' για $d \le i \le n-1$. Το κόστος της λύσης που υπολογίζει ο αλγόριθμος είναι:

$$\sum_{i=0}^{n+d-1}[L_i'/w]=\sum_{i=0}^{d-1}([L_i'/w]+[L_{i+n}'/w])+\sum_{i=d}^{n-1}[L_i'/w]\le\sum([(L_i'+L_{i+n}')/w]+1)+\sum_{i=d}^{n-1}[L_i/w]$$

$$=\sum_{i=0}^{n-1}[L_i/w] + d \le OPT(I) + d$$

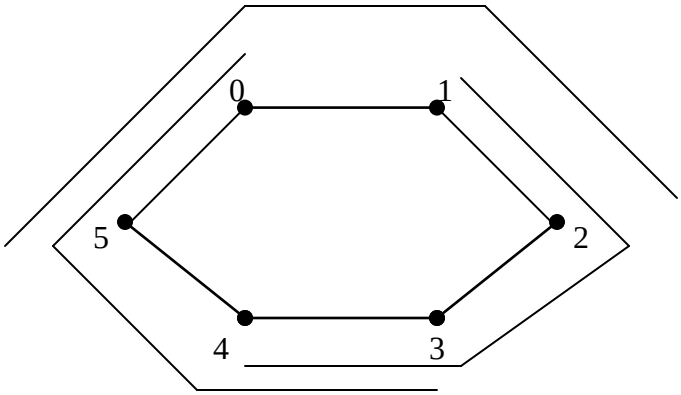
Στη χειρότερη περίπτωση d = n – 2 και το κόστος της λύσης είναι το πολύ 2*OPT(I), αφού $OPT(I) \ge n$.

Η πολυπλοκότητα του αλγορίθμου είναι ίδια με αυτή του αλγορίθμου για αλυσίδες, μιας και ο υπολογισμός του clockwise span και οι μετασχηματισμοί μπορούν να επιτευχθούν σε χρόνο O(m+n).

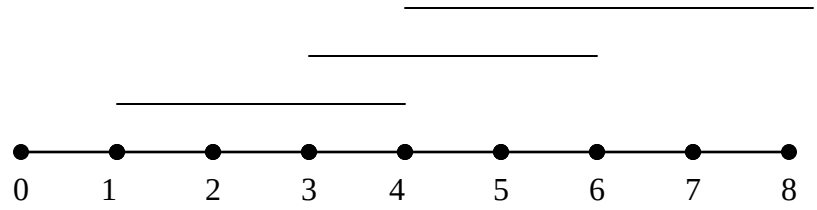
5.2 Παράδειγμα βέλτιστης λύσης.

Έστω ότι έχουμε τον δακτύλιο του σχήματος 5.1, και θέλουμε να χρωματίσουμε τα μονοπάτια {1,4}, {3,0} και {5,2} με w = 2 χρώματα. Αρχικά πρέπει να υπολογίσουμε το clockwise span για τον κόμβο 0. Στο παράδειγμά μας έχουμε d(0) = 2. άρα στην αλυσίδα που θα κατασκευάσουμε έχουμε 6+2+1 = 9 κόμβους αριθμημένους από το 0 έως το 8. το μονοπάτι {1,4}, αφού το i<j θα μπει ως έχει στα μονοπάτια της αλυσίδας. Το ίδιο δε συμβαίνει για το μονοπάτι {3,0} και {5,2}. Στην αλυσίδα θα μπουν τα μονοπάτια {3,6} και {5,8}. Επομένως προκύπτει η αλυσίδα του σχήματος 5.2.

Με χρήση του αλγορίθμων που παρουσιάστηκαν στα κεφάλαια 3 και 4, γίνεται χρωματισμός της αλυσίδας. Έτσι τα μονοπάτια {1,4} και {5,8} θα χρωματιστούν με το χρώμα, 1 ενώ το μονοπάτι {5,8} θα χρωματιστεί με το χρώμα 2.



Σχήμα 5.1. Ο δακτύλιος με τα μονοπάτια.



Σχήμα 5.2. Η αλυσίδα που προκύπτει.

Αφού τελειώσουμε με τον χρωματισμό της αλυσίδας πρέπει να αναθέσουμε τα αντίστοιχα χρώματα στα αρχικά μονοπάτια του δακτυλίου. Έτσι το μονοπάτι {1,4} του δακτυλίου θα πάρει ο ίδιο χρώμα με το μονοπάτι {1,4} της αλυσίδας, δηλαδή το χρώμα 1. Ομοίως το μονοπάτι {3,0} θα πάρει το χρώμα 2 και τέλος το μονοπάτι {5,2} θα πάρει το χρώμα 1.

Παρατηρούμε ότι το κόστος του χρώματος 1 στην ακμή {1,2} είναι 2. δηλαδή θέλουμε δύο συνδέσμους. Παρόλα αυτά ο παραπάνω χρωματισμός είναι και βέλτιστος μιας και τρία επικαλυπτόμενα μονοπάτια δεν μπορούν να χρωματιστούν με μόνο δύο χρώματα.

5.3 Παράδειγμα μη βέλτιστης λύσης.

5.4 Ο αλγόριθμος RPMC για δακτυλίους.

Ο αλγόριθμος για RPMC κάνει ένα routing ελαχίστων μονοπατιών, δηλαδή επιλέγει το μικρότερο από δύο εναλλακτικά μονοπάτια. Αν και τα δύο είναι ίσα, τότε επιλέγουμε ένα τυχαία. Στη συνέχεια χρησιμοποιεί τον αλγόριθμο για PMC για να χρωματίσει το αποτέλεσμα των μικρότερων μονοπατιών.

Η επιλογή των μικρότερων μονοπατιών ελαχιστοποιεί το άθροισμα των βαρών σε όλες τις ακμές και μειώνει το πάνω όριο για το d ($d<n/2$) . Το όριο για το κόστος της λύσης που μας δίνει ο παραπάνω αλγόριθμος μας το δίνει η ακόλουθη πρόταση:

Πρόταση 3: Ο αλγόριθμος για RPMC σε δακτυλίους υπολογίζει έναν πολυχρωματισμό με κόστος το πολύ OPT+n/2+d.

Απόδειξη: Έστω P η λίστα των μικρότερων μονοπατιών που κατασκευάζεται από τον αλγόριθμο και P* η λίστα των μονοπατιών σε μια βέλτιστη λύση. Συμβολίζουμε με L_i =L({i, (i+1) mod n},P*). Ορίζουμε f(i) = (i+[n/2]) mod n αντιποδικό κόμβο i στον δακτύλιο. Είναι εύκολο να δει κανείς ότι το f είναι μια μετάθεση του συνόλου των κόμβων.

Αφού τα μονοπάτια στο P έχουν μήκος το πολύ [n/2], καθένα από αυτά μπορούν να χρησιμοποιούν το πολύ μια από τις ακμές {i, (i+1) mod n} και {f(i), (f(i)+1) mod n}. Επομένως αντικαθιστώντας ένα μικρότερο μονοπάτι με το συμπληρωματικό δεν μειώνεται το άθροισμα των βαρών των δύο αυτών πλευρών. Αυτό υποδηλώνει ότι $L_i+L_{f(i)}\leq L^*_i+L^*_{f(i)}$. Από την πρόταση 2 το κόστος της προσεγγιστικής λύσης είναι το πολύ:

$$\sum_{i=0}^{n-1}[L_i/w] + d \leq \tfrac{1}{2}\sum_{n-1}^{n-1}([L_i/w]+[L_{f(i)}/w]) + d \leq \tfrac{1}{2}\sum_{i=0}^{n-1}([L^*_i/w]+[L^*_{f(i)}/w])+d$$

$$=\sum_{i=0}^{n-1} [L_i/w] + n/2+d \leq OPT(I) + n/2 + d$$

Πρόταση 4[η]: ο αλγόριθμος για RPMC για δακτυλίους πετυχαίνει προσεγγιστικό παράγοντα 2.

Απόδειξη: αν OPT(I) ≥ n, τότε το αποτέλεσμα βγαίνει κατευθείαν από την πρόταση 3 με χρήση του γεγονότος ότι d<n/2.

Υποθέτουμε ότι OPT(I) = n – s για κάποιον θετικό ακέραιο s. Αν η βέλτιστη λύση περιέχει μόνο μονοπάτια μήκους μικρότερου του n/2, τότε τα μονοπάτια αυτά είναι ακριβώς τα μονοπάτια που επέλεξε ο αλγόριθμος. Πέραν αυτού, αφού υπάρχουν ακμές που δεν χρησιμοποιούνται, ο αλγόριθμος για PMC θα κατασκευάσει ένα στιγμιότυπο αλυσίδας και θα πετύχει μια βέλτιστη λύση.

Ας υποθέσουμε τώρα ότι η βέλτιστη λύση περιέχει ένα μονοπάτι μήκους το πολύ n/2. Χωρίς βλάβη της γενικότητας μπορούμε να υποθέσουμε ότι αυτό το μονοπάτι είναι το <u,0> για κάποιο κόμβο u≤n/2. Έστω {α,α+1} και {b,b-1} είναι η πρώτη και η τελευταία αχρησιμοποίητη πλευρά που συναντάμε, αν διασχίσουμε τον δακτύλιο σύμφωνα με τη φορά του ρολογιού αρχίζοντας από το μηδέν. Προφανώς όλες οι αχρησιμοποίητες πλευρές βρίσκονται μεταξύ του α και του b, που σημαίνει ότι s≤b-a. Ας υποθέσουμε τα σύνολα των κόμβων V_1 = {υ | α<υ<b} και V_2 = {υ | υ<α ή υ≥b}. Παρατηρούμε ότι τα V_1 και V_2 δεν σχετίζονται, αφού σε διαφορετική περίπτωση κάποια από τις ακμές {α,α+1} και {b,b-1} θα χρησιμοποιούνταν. Επιπλέον, κάθε μονοπάτι που συνδέει δύο κόμβους στο V_1 μέσω του κόμβου b δεν μπορεί να είναι μικρότερο, αφού περιέχει το <υ,0>. Συνεπώς ελάχιστα μονοπάτια που περιέχουν τον b σαν εσωτερικό κόμβο συνδέουν κόμβους στο V_2 και περιέχουν όλες τις ακμές μεταξύ α και b. Άρα το clockwise span του b για το routing που πραγματοποιεί ο αλγόριθμος είναι το πολύ n/2-(b-α)≤n/2-s. Αφού d≤d(b), από την πρόταση 3 συμπεραίνουμε ότι το κόστος της λύσης είναι:

$$OPT(I) + n/2 + d(b) \leq OPT(I) + n/2 + n/2 – s \leq OPT(I) + n – s = 2OPT(I)$$

Η επιλογή των μικρότερων μονοπατιών απαιτεί O(m) χρόνο. Άρα, όπως και στις προηγούμενες περιπτώσεις, ο συνολικός χρόνος του αλγορίθμου εξαρτάται από τον

χρόνο του αλγορίθμου για χρωματισμό ακμών σε διμερές γραφήματα, ο οποίος, όπως αναφέραμε, τρέχει σε O((m+n*w)*w) (Schrijver [23]) ή σε O((m+n*w)log(m+n*w) (Cole Hopcroft [3]).

5.5 Αλγόριθμος πολύ-χρωματισμού σε αστέρες.

Σ' αυτό το κεφάλαιο θα ασχοληθούμε με το PMC για μια ειδική κατηγορία δέντρων, τους αστέρες. Θα πρέπει να αναφέρουμε ότι πάλι, όπως και στην περίπτωση των αλυσίδων, το RPMC ταυτίζεται με το PMC.

Στη συνέχεια παρουσιάζεται ένας αποτελεσματικός προσεγγιστικός αλγόριθμος που λύνει το πρόβλημα του PMC σε αστέρες. Για κάθε στιγμιότυπο I ο αλγόριθμος επιστρέφει μια λύση με κόστος το πολύ OPT(I)+*l*, όπου *l* είναι ο αριθμός των ακμών με θετικό βάρος. Ο προσεγγιστικός παράγοντας είναι 2, αφού OPT(I) > *l*.Χωρίς σφάλμα της γενικότητας μπορούμε να υποθέσουμε ότι το *l* ισούται με το βαθμό του αστέρα, αφού ακμές με βαθμό 0 μπορούν να αμεληθούν. Επίσης μπορούμε να υποθέσουμε ότι δεν υπάρχουν μονοπάτια με μήκος 1 –αν υπάρχουν μπορούμε να τα χρωματίσουμε στο τέλος χωρίς να αυξήσουμε τον προσεγγιστικό παράγοντα. (Μονοπάτια με μήκος 1 μπορούν να χρωματιστούν ανεξάρτητα για κάθε ακμή με χρήση μερικώς χρησιμοποιημένα χρώματα ή προσθέτοντας παράλληλες ακμές που θα χρειάζονταν έτσι κι αλλιώς).

Ο αλγόριθμος χρησιμοποιεί μια διμερή τεχνική χρωματισμού ακμών (bipartite edge coloring), που είναι όμοια με αυτή που χρησιμοποιήθηκε στο προηγούμενο κεφάλαιο. Στην περίπτωση των αστέρων, όμως, δεν χρειάζονται πρόσθετα μονοπάτια.

Βήμα 1^{ο}: Αναθέτουμε μια αυθαίρετη διεύθυνση σε κάθε μονοπάτι στο P.

Βήμα 2^{ο}: Για κάθε κόμβο υ:

- τα εξερχόμενα μονοπάτια $out_υ$ (σύμφωνα με την διεύθυνση που δόθηκε στο βήμα 1), διαιρούνται σε $[out_υ/w]$ συλλογές με το πολύ w στοιχεία με αυθαίρετο τρόπο.
- ομοίως, τα εισερχόμενα μονοπάτια $in_υ$, χωρίζονται σε $[in_υ/w]$ συλλογές.

Συμβολίζουμε τις συλλογές των εξερχόμενων μονοπατιών με V_{out} (αντίστοιχα των εισερχόμενων με V_{in}).

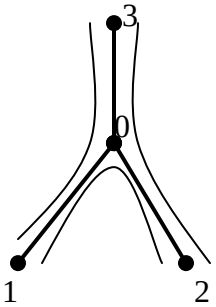
Βήμα 3^{ο}: ένα διμερές γράφημα $H=(V_{out}, V_{in}, A)$ κατασκευάζεται. Για κάθε μονοπάτι στο P υπάρχει μία ακμή στο A που συνδέει την συλλογή που αρχίζει και την συλλογή που τελειώνει. Άρα το H είναι ένα διμερές γράφημα βαθμού το πολύ w.

Βήμα 4^{ο}:Πετυχαίνεται χρωματισμός ακμών στο H. Σε κάθε μονοπάτι ανατίθεται το αντίστοιχο χρώμα της ακμής στο H.

Σ' αυτό το σημείο θα πρέπει να αναφέρουμε για το βήμα 1, ότι δεν υπάρχει κάποιο κριτήριο διεύθυνσης που μπορούμε να δώσουμε στα μονοπάτια του P που να μας εξασφαλίζει καλύτερο αποτέλεσμα. Στην έλλειψη ενός τέτοιου κριτηρίου οφείλεται το γεγονός ότι ο αλγόριθμος έχει προσεγγιστικό παράγοντα 2. παρόλα αυτά, ο προσεγγιστικός παράγοντας 2 πιθανό να μπορεί να μειωθεί σε 4/3. Ενώ, όπως έχει ήδη αναφερθεί, ο αλγόριθμος των , Hochbaum, Nishizeki και Shmoys [9] πετυχαίνει παράγοντα 4/3, που είναι ο καλύτερος δυνατός, εκτός αν P = NP.

5.6 Παράδειγμα βέλτιστης λύσης.

Στην παράγραφο αυτή θα δώσουμε ένα παράδειγμα για το οποίο ο παραπάνω αλγόριθμος για αστέρες δίνει βέλτιστη λύση. Έστω ότι έχουμε έναν αστέρα που αποτελείται από 4 κόμβους, με κεντρικό κόμβο τον κόμβο 0. Ο αριθμός χρωμάτων w, με τον οποίο θέλουμε να χρωματίσουμε τα μονοπάτια, είναι ίσος με δύο και έστω ότι έχουμε τα εξής μονοπάτια {1, 3}, {1, 2} και {2, 3} όπως φαίνεται και στο σχήμα 5.1.

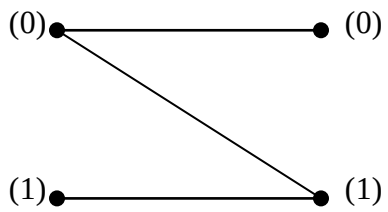


Σχήμα 5.1. Ο αστέρας με τα μονοπάτια.

Στο βήμα ένα του αλγορίθμου αναθέτουμε μια αυθαίρετη διεύθυνση σε κάθε μονοπάτι. Εμείς αποφασίζουμε να δώσουμε εξερχόμενη διεύθυνση σε αυτά τα μονοπάτια {i, j} για τα οποία ισχύει ότι το i>j. ενώ δίνουμε εισερχόμενη διεύθυνση στα μονοπάτια {i, j} για τα οποία ισχύει ότι το i<j. στην περίπτωση μας και τα τρία μονοπάτια που έχουμε, έχουνε την ίδια εξερχόμενη διεύθυνση.

Στο δεύτερο βήμα θα πρέπει να χωρίσουμε τα παραπάνω μονοπάτια σε συλλογές. Έχουμε δύο συλλογές εξερχόμενων μονοπατιών. Τα μονοπάτια {1, 3} και {1, 2} θα ανήκουν στη πρώτη συλλογή των εξερχόμενων μονοπατιών, ενώ το μονοπάτι {2, 3} ανήκει στη δεύτερη συλλογή εξερχόμενων μονοπατιών. Επίσης έχουμε ότι στον δεύτερο κόμβο καταλήγει ένα μονοπάτι, το {1, 2} ενώ στον τρίτο κόμβο καταλήγουν δύο μονοπάτια, τα {1, 3} και {2, 3}.

Στο τρίτο βήμα έχουμε την κατασκευή του διμερούς γραφήματος το οποίο θέλουμε στη συνέχεια να χρωματίσουμε με w = 2 χρώματα. Στο παρακάτω σχήμα φαίνεται το διμερές γράφημα.



Σχήμα 5.2. Το διμερές γράφημα που κατασκευάζεται στο τρίτο βήμα.

Στη συνέχεια πετυχαίνεται χρωματισμός των ακμών του παραπάνω γραφήματος με τον τρόπο που περιγράφεται στο κεφάλαιο τέσσερα. Από τον χρωματισμό προκύπτει ότι η ακμή που συνδέει τον κόμβο 0 του συνόλου V1 με τον κόμβο 0 του V2 παίρνει το χρώμα 1 ενώ η ακμή που συνδέει τον 0 του V1 με τον 1 του V2 παίρνει το χρώμα 2. Τέλος η ακμή που συνδέει τον κόμβο 1 του V1 με τον κόμβο 1 του V2 παίρνει το χρώμα 1.

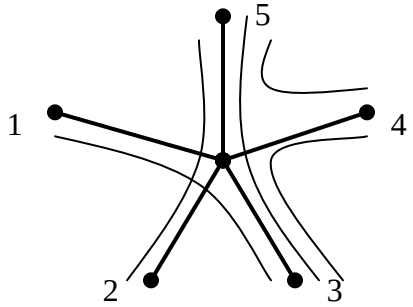
Αν αντιστοιχίσουμε τις χρωματισμένες, πλέον, ακμές του διμερούς γραφήματος με τα μονοπάτια του αστέρα θα πάρουμε ότι στα μονοπάτια {1, 2} και {2, 3} αντιστοιχεί το χρώμα 1 ενώ στο μονοπάτι {1, 3} αντιστοιχεί το χρώμα 2.

Όπως παρατηρούμε από τον παραπάνω χρωματισμό χρειαζόμαστε ένα σύνδεσμο παραπάνω για το μονοπάτι {1, 2}, καθώς δεν μπορούμε να χρωματίσουμε τα μονοπάτια με μόνο δύο χρώματα. Παρόλα αυτά, η λύση που παίρνουμε από τον αλγόριθμο είναι βέλτιστη.

5.7 Παράδειγμα μη βέλτιστης λύσης.

Το παράδειγμα που ακολουθεί μας δείχνει ότι ο αλγόριθμος δεν δίνει πάντα βέλτιστες λύσεις. Έτσι θα δούμε, στο παράδειγμα που ακολουθεί, ότι ενώ θα μπορούσαμε να έχουμε στον ίδιο σύνδεσμο κάποια μονοπάτια ο αλγόριθμος χρησιμοποιεί παραπάνω συνδέσμους.

Έστω ότι έχουμε έναν αστέρα με έξη κόμβους όπως φαίνεται στο σχήμα 5.4. έστω ότι τα χρώματα με τα οποία θέλουμε να χρωματίσουμε τα μονοπάτια είναι w = 3 και έστω ότι έχουμε τα εξής μονοπάτια: {1, 3}, {2, 5}, {3, 5}, {3, 4} και {4, 5}.



Σχήμα 5.4. Ο αστέρας με τα μονοπάτια.

Στο βήμα ένα του αλγορίθμου όπως και στο προηγούμενο παράδειγμα αποφασίζουμε να δώσουμε εξερχόμενη διεύθυνση σε αυτά τα μονοπάτια {i, j} για τα οποία ισχύει ότι το i>j. ενώ δίνουμε εισερχόμενη διεύθυνση στα μονοπάτια {i, j} για τα

οποία ισχύει ότι το i<j. Στην περίπτωση μας και τα έξη μονοπάτια που έχουμε, έχουνε την ίδια εξερχόμενη διεύθυνση.

Στο δεύτερο βήμα θα πρέπει να χωρίσουμε τα παραπάνω μονοπάτια σε συλλογές. Έχουμε τέσσερις συλλογές εξερχόμενων μονοπατιών και καμία εισερχόμενων. Το μονοπάτι (1, 3) πρώτη συλλογή των εξερχόμενων μονοπατιών, ενώ το μονοπάτι {2, 5} ανήκει στη δεύτερη συλλογή εξερχόμενων μονοπατιών. Τα μονοπάτια {3, 5} και {3, 4} ανήκουν στην τρίτη συλλογή και τέλος το μονοπάτι {4, 5} ανήκει στην τέταρτη συλλογή.. Επίσης έχουμε ότι στον τρίτο κόμβο καταλήγει ένα μονοπάτι, το {1, 3} στον τέταρτο καταλήγει το {3, 4} και στον πέμπτο καταλήγουν τα μονοπάτια {3,5} και {4, 5}. Σ' αυτό το βήμα, εξαιτίας της αυθαίρετης διεύθυνσης που δώσαμε στο πρώτο βήμα, οδηγούμαστε σε μη βέλτιστη λύση. Παρατηρούμε ότι το μονοπάτι {1, 3} έχει μπει σε διαφορετική συλλογή από ότι τα μονοπάτια {3, 5} και {3, 4} ενώ θα μπορούσε να μπει στην ίδια. Αυτό σημαίνει, για τη λύση μας, ότι αυτό το μονοπάτι θα είναι σε διαφορετικό σύνδεσμο από τα άλλα δύο μονοπάτια ενώ θα μπορούσε να ήταν στον ίδιο. Άρα η λύση που θα πάρουμε δεν είναι βέλτιστη.

5.8 Ανάλυση του Αλγορίθμου.

Ο αλγόριθμος πετυχαίνει προσεγγιστικό παράγοντα 2.Για να το αποδείξουμε αυτό πρέπει να αποδείξουμε πρώτα την εξής πρόταση:

Πρόταση 1: *Ο αλγόριθμος για PMC σε αστέρες υπολογίζει ένα πολύ-χρωματισμό τέτοιο ώστε για κάθε ακμή και κάθε χρώμα c,* $\mu(e,c) \leq [L(e,P)/w]+1$.

Απόδειξη: ο αλγόριθμος για διμερή χρωματισμό ακμών μας εγγυάται ότι κανένα χρώμα δεν χρησιμοποιείται δεύτερη φορά για μονοπάτια στην ίδια συλλογή. Άρα για κάθε ακμή e, προσπίπτουσα σε ένα φύλλο υ, κάθε χρώμα επαναλαμβάνεται το πολύ

$$[out_\upsilon/w] + [in_\upsilon/w] \leq [(out_\upsilon + in_\upsilon)/w] + 1 \leq [L(e,R)/w] + 1$$

φορές.

Σύμφωνα με την παραπάνω πρόταση το κόστος της λύσης που πετυχαίνει ο αλγόριθμος είναι το πολύ:

$$\sum_{e\varepsilon E}[L(e,P)/w] + l \leq OPT(I) + l \leq 2OPT(I).$$

όμοια με τον αλγόριθμο για αλυσίδες, η πολυπλοκότητα εξαρτάται από τον αλγόριθμο που θα επιλεγεί για τον χρωματισμό ακμών στο διμερές γράφημα. Στο βήμα 4 με χρήση του αλγορίθμου του Schrijver πετυχαίνουμε χρονική πολυπλοκότητα O(m*w). Όλα τα βήματα εκτός του 4ου απαιτούν O(m) χρόνο. Για w = Ω(logm), μπορούμε να κάνουμε χρήση του αλγορίθμου των Cole και Hopcroft με αποτέλεσμα να έχουμε πολυπλοκότητα O(m*logm).

Παράρτημα

Οδηγίες λειτουργίας του προγράμματος.

Όταν τρέχουμε το πρόγραμμα ο υπολογιστής μας ζητάει να βάλουμε τον αριθμό των χρωμάτων με τα οποία επιθυμούμε να χρωματίσουμε ην αλυσίδα. Στη συνέχεια μας ζητείται ο αριθμός των κόμβων που αποτελούν την αλυσίδα. Κατόπιν ο αριθμός των μονοπατιών που έχουμε στην αλυσίδα. Τέλος ο χρήστης πρέπει να βάλει και τα μονοπάτια. Κάθε φορά που ο χρήστης βάζει ένα μονοπάτι γίνεται έλεγχος αν το μονοπάτι αυτό ανήκει σε αλυσίδα. Αν ανήκει τότε προχωράμε στο επόμενο μονοπάτι. Αν όχι τότε τυπώνεται ένα μήνυμα λάθους και ο χρήστης καλείται να προσδιορίσει πάλι το μονοπάτι. Έτσι για παράδειγμα το μονοπάτι {5,1} δεν γίνεται δεκτό, όπως δεν γίνεται δεκτό και το μονοπάτι {6,9} να στην αλυσίδα έχουμε 6 μόνο κόμβους.

Αφού τελειώσουμε με την είσοδο, το πρόγραμμα βγάζει τα αποτελέσματα για κάθε ένα βήμα από τα έξη του αλγορίθμου. Έτσι, τυπώνουμε τον πίνακα με τα νέα μονοπάτια μήκους ένα που βάλαμε στο πρώτο βήμα. Τυπώνουμε τον πίνακα που έχει τα μεγάλα μονοπάτια του βήματος δύο. Συνεχίζοντας τυπώνονται οι συλλογές που ανήκει καθένα από τα μεγάλα μονοπάτια. Τυπώνεται το διμερές γράφημα που παράγεται από τις παραπάνω συλλογές. Στη συνέχεια γίνεται ο χρωματισμός του διμερούς γραφήματος και τέλος, τυπώνονται τα μεγάλα μονοπάτια με το χρώμα που πήραν από τον χρωματισμό του διμερούς γραφήματος καθώς επίσης και το χρώμα που παίρνουν τα αντίστοιχα μικρά μονοπάτια.

Στη συνέχεια ακολουθεί η υλοποίηση του αλγορίθμου των C. Nomikos, A. Pagourtzis και S Zachos για δρομολόγηση και πολυ-χρωματισμό μονοπατιών σε αλυσίδες, καθώς ο αλγόριθμος των R. Cole και HopCroft για χρωματισμό ακμών σε διμερή γραφήματα.

Ο κώδικας τουPMC για αλυσίδες.

```
#include<stdio.h>
#include<stdlib.h>


#define MAX 100

#define WHITE 1
#define GRAY 2
#define BLACK 3


struct path{  /*Struct whith the given paths*/
   int begin;
   int end;
   int color;
}P[MAX];

struct path1{/*Struct whith long paths paths*/
   int begin;
   int end;
   int color;

}NewP[MAX];

typedef struct liststr{ /*Struct we use to find the long paths and the groups*/
   int pathid;
   struct liststr *next;
}*list;

list B[MAX], E[MAX];

typedef struct setstr{/*Struct for the bipartite graph.*/
   int JoinNode;
   int EdgeLoad;
   int PositionNumber;
   int B_Color[MAX];
   struct setstr *VtoV;
   struct setstr *VtoH1;
   struct setstr *VtoH2;
   struct setstr *next;
   struct setstr *prev;
}*set;

set V1[MAX], V2[MAX], ReplicaV1[MAX], ReplicaV2[MAX];
/* we must keep V1 and V2 unchanged so we'll work on replicans*/

set H1V1[MAX], H1V2[MAX], H2V1[MAX], H2V2[MAX]; /*The two bipartite graphs we
                        are going to get after Euler split*/

set RepH1V1[MAX], RepH1V2[MAX], RepH2V1[MAX], RepH2V2[MAX];
set Rep1H1V1[MAX], Rep1H1V2[MAX], Rep1H2V1[MAX], Rep1H2V2[MAX];
set Match1[MAX], Match2[MAX], RepMatch1[MAX], RepMatch2[MAX];


set ColoredV1[MAX], ColoredV2[MAX];

void InsertList(list *lst, int pthid);

void CountingSort(int n, int k, int A[MAX], int D[MAX], int F[MAX]);
```

```
/*
   n: length of arrays A,D and F,
   k: range of integers in arrays A,D and F,
   A: the array that we want to sort,
   D and F two arrays that we need to change the potition of their elements in
   the same way as A. These two arrays are not sorted after the end of
   Counting sort.
*/
void Color(set V_1[], set V_2[]);
void Matching(set V_1[], set V_2[]);
void DFS(set V_1[], set V_2[], int i);
void DFS_VISIT(set V_1[], set V_2[], int , int);
void Partition(set V_1[], set V_2[]);
void EulerSplit(set V_1[], set V_2[]);
void ErasePath(set lst1[], set lst2[], int num);
void Union(set V_1[], set V_2[]);
void InsertSet(set *lst, int nds, int lds, int pst);
void DeleteSet(set *lst, set ptr);
int FindOdd(int array[], int length);
int FindEven(int array[], int length);
int FindNorm(int array[], int length);


int NodeDegree[MAX];
int NodeDegree2[MAX];
int groupBctr = 0, groupEctr = 0;

int DFS_time = 0, DFS_P1[MAX], DFS_P2[MAX];/*for the DFS.*/
int DFS_ColorV1[MAX], DFS_ColorV2[MAX];/*for the DFS.*/
set DFS_EdgeP1[MAX], DFS_EdgeP2[MAX];
int d1[MAX], d2[MAX];
int f1[MAX], f2[MAX];
int DFS_InV1 = 1;
int NumberOfColors;

int main()
{

   int NumberOfNodes, NumberOfPaths;
   int BeginNode, EndNode, NumberOfExtraEdges;

   int NumberOfBeginningNodes[MAX]; /*array that holds number of Beginning
                        paths for all edges*/
   int NumberOfEndingNodes[MAX]; /*array that holds number of ending paths
                        for all edges*/
   int Load[MAX], LongPath[MAX]; /* array that holds the load of every edge*/
   int i, j, k, l; /*simple counters*/
   int  m = 0, newm = 0; /* m(newm) : number of paths at P(NewP)*/
   int GroupB[MAX], GroupE[MAX];
   list temp, temp1;
   set temp2;
   int NumberArray[100];/*Array to help as find the potion of one element
                     after we sort in step 4*/
   int multiplicity = 0;


   puts("Please enter the maximum number of\n"
      "colors you want to use for Multi-Coloring:");
   scanf("%d",&NumberOfColors);

   puts("Please enter the number of nodes:");
   scanf("%d",&NumberOfNodes); puts("");
```

```
   puts("Please enter the number of paths:");
   scanf("%d",&NumberOfPaths);



   for ( i = 0; i < MAX; i++ ){ /*Give starting number 0*/
      NumberOfBeginningNodes[i] = 0;
      NumberOfEndingNodes[i] = 0;
      Load[i] = 0;
      LongPath[i] = 0;
      GroupB[i] = 0;
      GroupE[i] = 0;
      NodeDegree[i] = 0;
      NodeDegree2[i] = 0;
   }

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

      printf("Enter the beging node of path %d: ",i);
      scanf("%d",&BeginNode);
      printf("Enter the ending node of path %d: ",i);
      scanf("%d",&EndNode);

      if (BeginNode > EndNode){     /*Check if given number make a chain*/
        puts("No going back allowed in Chains\n Give other path");
        i--;
      }
      else if(BeginNode < 0 || BeginNode > NumberOfNodes-1 || EndNode < 0
      || EndNode > NumberOfNodes-1){
        puts("\bError! Path not acceptable. \nGive other path");
        i--;
      }
      else{  /* if we have an acceptable path*/
         NumberOfBeginningNodes[BeginNode]++;
         NumberOfEndingNodes[EndNode]++;

         P[i].begin = BeginNode;    //InsertList into list of paths values.
         P[i].end = EndNode;       //let the color be 0 at the beginig.
         P[i].color = 0;
         m++;

      }
      puts("_____________________________________________________");
   }

/*------------------------- STEP 1 ----------------------------*/

/*add extra paths of length one in order to make all edge loads
integral multiples ow NumberOfColors*/

   Load[0] = NumberOfBeginningNodes[0]; /*finds the load of every edge*/
   for ( i = 1; i < NumberOfNodes; i++){
      Load[i] = Load[i-1] + NumberOfBeginningNodes[i] -NumberOfEndingNodes[i];
   }
   for ( i = 0; i < NumberOfNodes; i++){
      NumberOfExtraEdges = Load[i] % NumberOfColors;
      if(NumberOfExtraEdges != 0)
         for(j = 0; j < NumberOfColors - NumberOfExtraEdges; j++){
            NumberOfBeginningNodes[i]++;
            NumberOfEndingNodes[i+1]++;

              P[m].begin = i;
```

```
                P[m].end = i+1;
                P[m].color = 0;
                m++;  /*m is the total number of
                     paths in P*/
            }
    }
    puts("Exit of STEP 1:");
    puts("_________________\n");
    for ( i = 0; i < m; i++ ){
        printf("(%d,%d)-->", P[i].begin,P[i].end);
    }
        puts("");
/*---------------------- END OF STEP 1 -------------------------*/


/*----------------------     STEP 2    -------------------------*/
/*While there is a path P1 ending and a path P2 beginning at the
the same vertex we connect them to construct a larger path.*/

    for( i = 0; i < m; i++){
         InsertList(&B[P[i].begin],i);
         InsertList(&E[P[i].end],i);
    }

/*  for( i = 0; i < NumberOfNodes; i++){
        printf("B[%d]: ",i);
        temp = B[i];
        while(temp != NULL){
            printf("%d-->",temp->pathid);
            temp = temp->next;
        }puts("");
    }
    for( i = 0; i < NumberOfNodes; i++){
        printf("E[%d]: ",i);
        temp = E[i];
        while(temp != NULL){
            printf("%d-->",temp->pathid);
            temp = temp->next;
        }puts("");
    }
*/
    j = 0;
    for( i = 0; i < NumberOfNodes; i++){/*giving values to LongPath*/
        temp = B[i];
        temp1 = E[i];
        if(temp1 == NULL){

            while(temp != NULL){
                LongPath[temp->pathid] = j;
                j++;
                temp = temp->next;
            }
        }
        else if(temp1 != NULL && temp != NULL){
            while(temp1 != NULL){
                LongPath[temp->pathid] = LongPath[temp1->pathid];

                temp1 = temp1->next;
                temp = temp->next;

                if(temp == NULL)
                    break;
```

```
            }
            while(temp != NULL){
               LongPath[temp->pathid] = j;
               j++;
               temp = temp->next;
            }
         }
      }
      newm = j; /*j is the number of paths in NewP*/

/*    for(i = 0; i < m; i++)printf("%d, ",LongPath[i]);puts("");*/

      for(i = 0; i < m; i++){
         NewP[LongPath[i]].begin = P[i].begin;
         NewP[LongPath[i]].end = P[i].end;
      }
      for(i = 0; i < m; i++){
         if(P[i].begin <= NewP[LongPath[i]].begin)
            NewP[LongPath[i]].begin = P[i].begin;
         if(P[i].end >= NewP[LongPath[i]].end)
            NewP[LongPath[i]].end = P[i].end;
      }
      puts("\nExit of STEP 2:");
      puts("_________________\n");
      for ( i = 0; i < newm; i++ ){
         printf("(%d,%d)-->", NewP[i].begin,NewP[i].end);
      }
      puts("");

/*---------------------- END OF STEP 2 -------------------------*/

/*------------------------- STEP 3 ----------------------------*/
/*Partition the paths into collections of w elements.*/

      for( i = 0; i < NumberOfNodes; i++){
         B[i] = NULL;
         E[i] = NULL;
      }
      for( i = 0; i < newm; i++){
          InsertList(&B[NewP[i].begin],i);
          InsertList(&E[NewP[i].end],i);
      }
/*    for( i = 0; i < NumberOfNodes; i++){
         printf("B[%d]: ",i);
         temp = B[i];
         while(temp != NULL){
            printf("%d-->",temp->pathid);
            temp = temp->next;
         }puts("");
      }
      for( i = 0; i < NumberOfNodes; i++){
         printf("E[%d]: ",i);
         temp = E[i];
         while(temp != NULL){
            printf("%d-->",temp->pathid);
            temp = temp->next;
         }puts("");
      }
*/
      for(i = 0; i <NumberOfNodes; i++){
```

```
      temp = B[i];
      temp1 = E[i];
      if(temp1 == NULL && temp != NULL){/*We are interested in conditions
                  that one of the pointers is not null. We can't
                  have both pointers not equal to null at the
                  same time*/
         while(temp != NULL){

            for(j = 0; j < NumberOfColors; j++){/*We give values to group
                        if at B[i] there are more than nodes
                        than the number of colors then
                        temp->next != Null so the program
                        remains inside the while.*/
               GroupB[temp->pathid] = groupBctr;
               temp = temp->next;
            }
            groupBctr++;
         }
      }
      else if(temp1 != NULL && temp == NULL){
         while(temp1 != NULL){

            for(j = 0; j < NumberOfColors; j++){
               GroupE[temp1->pathid] = groupEctr;
               temp1 = temp1->next;
            }
            groupEctr++;
         }

      }
   }
   /*printf("\n%d, %d\n",groupBctr,groupEctr);*/
   puts("\nExit of STEP 3:");
   puts("_____");
   puts("GoupB: ");
   for(i = 0; i < newm; i++)
      printf("%d ",GroupB[i]);
   puts("");
   puts("GoupE: ");
   for(i = 0; i < newm; i++)
      printf("%d ",GroupE[i]);

/*---------------------- END OF STEP 3 -------------------------*/


/*-------------------------- STEP 4 ----------------------------*/
/*The construction of the Bipartite graph*/

   for(i = 0; i < newm; i++)/*Give starting numbers to NumberArray.*/
      NumberArray[i] = i;


/*The construction of V1 (The set of nodes in set 1 of the bipartite graph).
Nodes in set one don't have edges or paths that unites them.*/

   CountingSort(newm,groupEctr,GroupE,GroupB,NumberArray);/*Sorting GroupE*/
   CountingSort(newm,groupBctr,GroupB,GroupE,NumberArray);/*Sorting GroupB*/
   /*We first need to sort the array by GroupE and then by GroupB
   cause we want to have the same numbers of GroupE adjacently.*/
   k = 0;

   for(i = 0; i < groupBctr; i++)
```

```
      for(j = i*NumberOfColors; j < NumberOfColors*(1+i); j++){

         k = j;
         while(GroupE[j] == GroupE[k] && k < NumberOfColors*(1+i)){
            multiplicity++;
            k++;
         }

         InsertSet(&V1[i], GroupE[j], multiplicity, j);
         InsertSet(&ReplicaV1[i], GroupE[j], multiplicity, j);

         InsertSet(&V2[GroupE[j]], GroupB[j], multiplicity, 0);/* we don't
                                    need the number of position
                                    cause we have it from V1*/
         InsertSet(&ReplicaV2[GroupE[j]], GroupB[j], multiplicity, 0);
         /*ReplicaV1 and ReplicaV2 is the graph we are going to use.
         We want V1 and V2 later on*/

         V1[i]->VtoV = V2[GroupE[j]];
         ReplicaV1[i]->VtoV = ReplicaV2[GroupE[j]];

         V2[GroupE[j]]->VtoV = V1[i];
         ReplicaV2[GroupE[j]]->VtoV = ReplicaV1[i];

         multiplicity = 0;
         j = k-1;
      }
   /*printf("\n Testing conection: %d %d\n",V2[0]->VtoV->JoinNode,
   V2[0]->VtoV->EdgeLoad);*/
   puts("\nExit of STEP 4:");
   puts("_________________");
   puts("");

   for( i = 0; i < groupBctr; i++){
      printf("V1[%d]: ",i);
      temp2 = V1[i];
      while(temp2 != NULL){
         printf("(%d,%d,%d)-->",temp2->JoinNode, temp2->EdgeLoad, temp2->PositionNumber);
         temp2 = temp2->next;
      }puts("");
   }

   puts("");
   for( i = 0; i < groupEctr; i++){
      printf("V2[%d]: ",i);
      temp2 = V2[i];
      while(temp2 != NULL){
         printf("(%d,%d,%d)-->",temp2->JoinNode, temp2->EdgeLoad, temp2->PositionNumber);
         temp2 = temp2->next;
      }puts("");
   }

/*---------------------- END OF STEP 4 -------------------------*/

/*----------------------   STEP 5   -----------------------------*/

   for(i = 0; i< groupBctr; i++){
      temp2 = V1[i];
      while(temp2 != NULL){
         InsertSet(&ColoredV1[i],temp2->JoinNode, temp2->EdgeLoad, temp2->PositionNumber);
         for(j = 0; j < NumberOfColors; j++)
            ColoredV1[i]->B_Color[j] = 0;
```

```
            temp2 = temp2->next;
        }

    }
    Color(V1, V2);
//  Matching(V1,V2);
/*    puts("\nExit of STEP 5:");
    puts("_________________");
    puts("");

     puts("\n Testing Graph Match1.");
     for( i = 0; i < groupBctr; i++){
        printf("Match1[%d]: ",i);
        temp2 = Match1[i];
        while(temp2 != NULL){
            printf("(%d,%d,%d)-->",temp2->JoinNode, temp2->EdgeLoad, temp2->PositionNumber);
            temp2 = temp2->next;
        }puts("");
    }
    puts("\n Testing Graph Match2.");
    for( i = 0; i < groupBctr; i++){
        printf("Match2[%d]: ",i);
        temp2 = Match2[i];
        while(temp2 != NULL){
            printf("(%d,%d,%d)-->",temp2->JoinNode, temp2->EdgeLoad, temp2->PositionNumber);
            temp2 = temp2->next;
        }puts("");
    }
//printf("asda %d, %d\n",H2V1[0]->VtoV->JoinNode,H2V1[0]->VtoV->EdgeLoad);

/*---------------------- END OF STEP 5 -------------------------*/

/*------------------------- STEP 6 ----------------------------*/
    for(j = 0; j < groupBctr; j++){
        temp2 = ColoredV1[j];
        while(temp2 != NULL){
            for(i = 0; i < temp2->EdgeLoad; i++)
                NewP[temp2->PositionNumber+i].color = temp2->B_Color[i];
            temp2 = temp2->next;
        }
    }
    puts("\nExit of STEP 6:");
    puts("_________________");
    puts("");
    puts("Long Paths with Colors");
    for ( i = 0; i < newm; i++ ){
        printf("(%d,%d,%d)-->", NewP[i].begin,NewP[i].end,NewP[i].color);
    }
    puts("");

    for ( i = 0; i < m; i++ ){
        P[i].color = NewP[NumberArray[LongPath[i]]].color;
    }
    puts("\nPaths with Colors.");
    for ( i = 0; i < m; i++ ){
        printf("(%d,%d,%d)-->", P[i].begin,P[i].end,P[i].color);
    }
    puts("");
/*---------------------- END OF STEP 6 -------------------------*/
    printf("\nPress a key to exit.\n");
    scanf("%d",&exit);
    return 0;
```

```
}

void InsertList(list *lst, int pthid)
{
   list newptr;

   if((newptr = (list )malloc(sizeof(struct liststr)))==NULL){
      fprintf(stderr,"Cannot allocate memory. EXITING.");
      exit(1);
   }
   newptr->pathid = pthid;
   newptr->next = *lst;
   *lst = newptr;
}
void CountingSort(int n, int k, int A[MAX], int D[MAX], int F[MAX])
{
   int C[MAX],B[MAX],G[MAX],H[MAX],i,j;

   for(i = 0; i < k; i++)
      C[i] = 0;
   for(j = 0; j < n; j++)
      C[A[j]]++;
   for(i = 1; i < k; i++)
      C[i] = C[i] + C[i-1];

   for(j = n-1; j >= 0; j--){
      B[C[A[j]]-1] = A[j];
      G[C[A[j]]-1] = D[j];
      H[C[A[j]]-1] = F[j];
      C[A[j]]--;
   }
   for(i = 0; i < n; i++){
      A[i] = B[i];
      D[i] = G[i];
      F[i] = H[i];
   }
}
void Color(set V_1[], set V_2[]){
            int ctr, EdgeNum1 = 0, EdgeNum2 = 0, j = 0, i,MaxDegree;
            set G1V1[MAX], G1V2[MAX], G2V1[MAX], G2V2[MAX];
            set tmp,tmp1,temp2, temp[groupBctr];
   static int test=0;
   static int D = NumberOfColors;

            for(ctr = 0; ctr < groupBctr; ctr++){/*Find degree*/
                        NodeDegree[ctr] = 0;
                        tmp = V_1[ctr];
                        while(tmp != NULL){
                                    NodeDegree[ctr] += tmp->EdgeLoad;
                                    tmp=tmp->next;
                        }
            }
   MaxDegree = NodeDegree[0];
            for(ctr = 0; ctr < groupBctr; ctr++){
      G1V1[ctr] = NULL;
      G1V2[ctr] = NULL;
      G2V1[ctr] = NULL;
      G2V2[ctr] = NULL;
      temp[ctr] = NULL;
   }
   if(MaxDegree % 2 != 0){
      Matching(V_1, V_2);
```

```
for(ctr = 0; ctr < groupBctr; ctr++){
        tmp1 = Match1[ctr];
        tmp = ColoredV1[ctr];
        while(tmp != NULL){
           if(tmp->JoinNode == tmp1->JoinNode){
              j = 0;
              while(tmp->B_Color[j] != 0)
                 j++;
              tmp->B_Color[j] = D;
              /*printf("(%d %d %d) %d\n",tmp->JoinNode,tmp->EdgeLoad,tmp->B_Color[j],j);*/
           }
           tmp = tmp->next;
        }
     }
D = D - 1;
for(ctr = 0; ctr < groupBctr; ctr++){
        tmp1 = Match1[ctr];
        tmp = V_1[ctr];
        while(tmp != NULL){
           if(tmp->JoinNode == tmp1->JoinNode){
              tmp->EdgeLoad--;
              tmp->VtoV->EdgeLoad--;
           }
           tmp = tmp->next;
        }
     }
     for(ctr = 0; ctr < groupBctr; ctr++){/*In our graphs we have some edges
                        with multiplisity of zero. We must delete them*/
        tmp = V_1[ctr];
        while(tmp != NULL){
           if(tmp->EdgeLoad == 0)
              DeleteSet(&V_1[ctr], tmp);
           tmp = tmp->next;
        }
     }
     for(ctr = 0; ctr < groupBctr; ctr++){
        tmp = V_2[ctr];
        while(tmp != NULL){
           if(tmp->EdgeLoad == 0)
              DeleteSet(&V_2[ctr], tmp);
           tmp = tmp->next;
        }
     }/*End of deleting edges with zero multiplicity*/


                        for(ctr = 0; ctr < groupBctr; ctr++){
                                Match1[ctr] = NULL;
                                Match2[ctr] = NULL;
                        }
           }

           EulerSplit(V_1, V_2);


           for(ctr = 0; ctr < groupBctr; ctr++){/*Copying H1V1 to new graph cause
                                                        we must crear H1V1 for the next Euler Split.*/
                        tmp = H1V1[ctr];
                        while(tmp != NULL){
                         InsertSet(&G1V1[ctr],tmp->JoinNode, tmp->EdgeLoad, tmp->PositionNumber);
                        InsertSet(&G1V2[tmp->JoinNode], ctr, tmp->EdgeLoad, 0);
                         G1V1[ctr]->VtoV = G1V2[tmp->JoinNode];
                        G1V2[tmp->JoinNode]->VtoV = G1V1[ctr];
```

```
                        EdgeNum1 += tmp->EdgeLoad;
                        tmp = tmp->next;
                }
        }


        for(ctr = 0; ctr < groupBctr; ctr++){/*Copying H2V1 to new graph cause
                                                        we must crear H1V1 for the next Euler Split.*/
                tmp = H2V1[ctr];
                while(tmp != NULL){
                InsertSet(&G2V1[ctr],tmp->JoinNode, tmp->EdgeLoad, tmp->PositionNumber);
                InsertSet(&G2V2[tmp->JoinNode], ctr, tmp->EdgeLoad, 0);
                G2V1[ctr]->VtoV = G2V2[tmp->JoinNode];
                G2V2[tmp->JoinNode]->VtoV = G2V1[ctr];
                        EdgeNum2 += tmp->EdgeLoad;
                        tmp = tmp->next;
                }
        }

        for(ctr = 0; ctr < groupBctr; ctr++){/*Initiate graphs produced by Euler Split.*/
                H1V1[ctr] = NULL; H1V2[ctr] = NULL;
                H2V1[ctr] = NULL; H2V2[ctr] = NULL;
        }
        if(EdgeNum1 > EdgeNum2){/*if 1st graph larger swap graphs.*/
                for(ctr = 0; ctr < groupBctr; ctr++){
                        temp[ctr] = G1V1[ctr];
                        G1V1[ctr] = G2V1[ctr];
                        G2V1[ctr] = temp[ctr];
                }
                for(ctr = 0; ctr < groupBctr; ctr++){
                        temp[ctr] = G1V2[ctr];
                        G1V2[ctr] = G2V2[ctr];
                        G2V2[ctr] = temp[ctr];
                }
        }
//test++;                   system("PAUSE");
//if(test == 2){system("PAUSE");exit(0);}
if(G1V1[0] !=NULL)
   Color(G1V1, G1V2);
if(G2V1[0] !=NULL)
   Color(G2V1, G2V2);

//          free(tmp);
}
void Matching(set V_1[], set V_2[]){
        int i, D, ctr;
   int EdgeNum1 = 0, EdgeNum2 = 0;
   set tmp,temp2,temp;
        set DFS_G1[MAX], DFS_G2[MAX];

   for(ctr = 0; ctr < groupBctr; ctr++){
      DFS_G1[ctr] = NULL;
      DFS_G2[ctr] = NULL;
   }
   for(ctr = 0; ctr < groupBctr; ctr++){
      tmp = V_1[ctr];
      while(tmp != NULL){
         InsertSet(&DFS_G1[ctr], tmp->JoinNode, tmp->EdgeLoad, tmp->PositionNumber);
            tmp = tmp->next;
      }
   }
```

```
for(ctr = 0; ctr < groupBctr; ctr++){
   tmp = V_2[ctr];
   while(tmp != NULL){
      InsertSet(&DFS_G2[ctr], tmp->JoinNode, tmp->EdgeLoad, 0);
           tmp = tmp->next;
   }
}

        for(ctr = 0; ctr < groupBctr; ctr++){/*Find degree*/
                    NodeDegree[ctr] = 0;
                    tmp = V_1[ctr];
                    while(tmp != NULL){
                                NodeDegree[ctr] += tmp->EdgeLoad;
                                tmp=tmp->next;
                    }
        }


        D = NodeDegree[0];
        for(i = 1; i <= D; i *= 2){
                    DFS(V_1, V_2, i);
        }

for(ctr = 0; ctr < groupBctr; ctr++){/*In our graphs we have some edges
                    with multiplisity of zero. We must delete them*/
   temp = V_1[ctr];
   while(temp != NULL){
      if(temp->EdgeLoad == 0)
         DeleteSet(&V_1[ctr], temp);
      temp = temp->next;
   }
}

for(ctr = 0; ctr < groupBctr; ctr++){
   temp = V_2[ctr];
   while(temp != NULL){
      if(temp->EdgeLoad == 0)
         DeleteSet(&V_2[ctr], temp);
      temp = temp->next;
   }
}/*End of deleting edges with zero multiplicity*/
        for(ctr = 0; ctr < groupBctr; ctr++){/*Copying G to Match*/
           temp = V_1[ctr];
           while(temp != NULL){
               InsertSet(&Match1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
               InsertSet(&Match2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
              Match1[ctr]->VtoV = Match2[temp->JoinNode];
              Match2[temp->JoinNode]->VtoV = Match1[ctr];
              temp = temp->next;
           }
        }


        for(ctr = 0; ctr < groupBctr; ctr++){/*Finding degree*/
                    NodeDegree[ctr] = 0;
                    tmp = Match1[ctr];
                    while(tmp != NULL){
                                NodeDegree[ctr] += tmp->EdgeLoad;
                                tmp=tmp->next;
                    }
        }
```

```
if(NodeDegree[0] != 1){
  Partition(Match1, Match2);/*After Partition Match1,Match2 = NULL*/
  while(1){
       for(ctr = 0; ctr < groupBctr; ctr++){/*Finding degree of H1V1*/
       NodeDegree[ctr] = 0;
       tmp = H1V1[ctr];
       while(tmp != NULL){
          NodeDegree[ctr] += tmp->EdgeLoad;
          EdgeNum1 += NodeDegree[ctr];/*Must find number of edges*/
          tmp=tmp->next;
       }
    }
       for(ctr = 0; ctr < groupBctr; ctr++){/*Finding degree of H2V1*/
       NodeDegree2[ctr] = 0;
       tmp = H2V1[ctr];
       while(tmp != NULL){
          NodeDegree2[ctr] += tmp->EdgeLoad;
          EdgeNum2 += NodeDegree2[ctr];/*Must find number of edges*/
          tmp=tmp->next;
       }
    }
    if(NodeDegree[0] == 1){/*if H1V1 is the matching*/
               for(ctr = 0; ctr < groupBctr; ctr++){/*Copying matching H1V1
                                   to Match1*/
          temp = H1V1[ctr];
                    while(temp != NULL){
                    InsertSet(&Match1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
             temp = temp->next;
             }
       }
       for(ctr = 0; ctr < groupBctr; ctr++){
          temp = Match1[ctr];
          while(temp != NULL){
                       InsertSet(&Match2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                       Match1[ctr]->VtoV = Match2[temp->JoinNode];
                              Match2[temp->JoinNode]->VtoV = Match1[ctr];
                       temp = temp->next;
          }
       }/*End of Copying matching*/
       for(ctr = 0; ctr < groupBctr; ctr++){/*Must set all pointers
                                   we're going to use later
                                   to NULL*/
          H1V1[ctr] = NULL; H1V2[ctr] = NULL;
          H2V1[ctr] = NULL; H2V2[ctr] = NULL;

          RepMatch1[ctr] = NULL; RepMatch2[ctr] = NULL;
       }
       break;
    }
    else if(NodeDegree2[0] == 1){/*H2V1 is the matching*/
               for(ctr = 0; ctr < groupBctr; ctr++){
          temp = H2V1[ctr];
                    while(temp != NULL){
                    InsertSet(&Match1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
             temp = temp->next;
             }
       }
       for(ctr = 0; ctr < groupBctr; ctr++){
          temp = Match1[ctr];
          while(temp != NULL){
                       InsertSet(&Match2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                       Match1[ctr]->VtoV = Match2[temp->JoinNode];
```

```
                        Match2[temp->JoinNode]->VtoV = Match1[ctr];
                    temp = temp->next;
                }
            }
            for(ctr = 0; ctr < groupBctr; ctr++){

                H1V1[ctr] = NULL; H1V2[ctr] = NULL;
                H2V1[ctr] = NULL; H2V2[ctr] = NULL;

                RepMatch1[ctr] = NULL; RepMatch2[ctr] = NULL;
            }
            break;
        }
        else if(NodeDegree[0] != 1 && NodeDegree2[0] != 1){/*if none of
                        H1V1, H2V1 is the matching*/
            if(EdgeNum1 <= EdgeNum2){/*must partition again graph with
                        smallest number of edges*/
                for(ctr = 0; ctr < groupBctr; ctr++){
                    temp = H1V1[ctr];
                  while(temp != NULL){
                    InsertSet(&RepMatch1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
                temp = temp->next;
                }
            }
            for(ctr = 0; ctr < groupBctr; ctr++){
                temp = RepMatch1[ctr];
                while(temp != NULL){
                        InsertSet(&RepMatch2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                        RepMatch1[ctr]->VtoV = RepMatch2[temp->JoinNode];
                                RepMatch2[temp->JoinNode]->VtoV = RepMatch1[ctr];
                            temp = temp->next;
                        }
            }
            }
            else{
                for(ctr = 0; ctr < groupBctr; ctr++){
                    temp = H2V1[ctr];
                  while(temp != NULL){
                    InsertSet(&RepMatch1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
                temp = temp->next;
                }
            }
            for(ctr = 0; ctr < groupBctr; ctr++){
                temp = RepMatch1[ctr];
                while(temp != NULL){
                        InsertSet(&RepMatch2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                        RepMatch1[ctr]->VtoV = RepMatch2[temp->JoinNode];
                                RepMatch2[temp->JoinNode]->VtoV = RepMatch1[ctr];
                            temp = temp->next;
                        }
            }
            }
            for(ctr = 0; ctr < groupBctr; ctr++){
                H1V1[ctr] = NULL; H1V2[ctr] = NULL;
                H2V1[ctr] = NULL; H2V2[ctr] = NULL;

            }

            Partition(RepMatch1, RepMatch2);
        }
    }
}
```

```
//		free(tmp);
}
void DFS(set V_1[], set V_2[], int i)
{
		int ctr;
		set tmp;
		int Find = 0, InV1 = 0;

		DFS_time = 0;
		for(ctr = 0; ctr < groupBctr; ctr++){
		DFS_ColorV1[ctr] = WHITE;
			DFS_ColorV2[ctr] = WHITE;

	d1[ctr] = 0;
	d2[ctr] = 0;
	f1[ctr] = 0;
	f2[ctr] = 0;
			DFS_EdgeP1[ctr] = NULL;
	DFS_P1[ctr] = 0;
			DFS_P2[ctr] = 0;
			DFS_EdgeP1[ctr] = NULL;
		}

		for(ctr = 0; ctr < groupBctr; ctr++){
			if(DFS_ColorV1[ctr] == WHITE){/*If we find a white node then

		we must check the connection with

		other nodes in level we are.*/
				tmp = V_1[ctr];
				while(tmp != NULL){
					if(((tmp->EdgeLoad)/i) % 2 != 0){
						Find = 1;
						break;
					}
					else tmp = tmp->next;
				}
				if(Find == 1){
		DFS_InV1 = 1;
					DFS_VISIT(V_1, V_2, ctr, i);
					Find = 0;
				}
			}
		}
/*		for(ctr = 0; ctr < groupBctr; ctr++)
			printf("d1[%d]:%d f1[%d]:%d\n",ctr, d1[ctr],ctr,f1[ctr]);
		for(ctr = 0; ctr < groupBctr; ctr++)
			printf("d2[%d]:%d f2[%d]:%d\n",ctr, d2[ctr],ctr,f2[ctr]);
   puts("");
  */

}
void DFS_VISIT(set V_1[], set V_2[], int ctr, int i)
{
		set tmp;
   set tmp1, tmp2;
   int CircleBeginning_d;

		int Find = 0,test = 0;

		if(DFS_InV1 == 1){
```

```
                DFS_ColorV1[ctr] = GRAY;
                d1[ctr] = DFS_time+1;
                tmp = V_1[ctr];
        }
        else{
                DFS_ColorV2[ctr] = GRAY;
                d2[ctr] = DFS_time+1;
                tmp = V_2[ctr];
        }
tmp1 = tmp;
        DFS_time++;

        while(tmp1 != NULL){/*for each node in Adj[u]*/

                while(tmp != NULL){
                        if(((tmp->EdgeLoad)/i) % 2 != 0){
                           if(DFS_InV1 == 1){/*We are at V1*/
                              if(DFS_ColorV2[tmp->JoinNode] == WHITE){
                                 DFS_P2[ctr] = tmp->JoinNode;
                                 DFS_EdgeP2[ctr] = tmp;
                                 DFS_InV1 = 0;
                                 DFS_VISIT(V_1, V_2, tmp->JoinNode, i);
                                 break;
                        }
                        else{
                           if(DFS_ColorV2[tmp->JoinNode] == GRAY){
                              if(d1[ctr] - d2[tmp->JoinNode] > 1){
                                 int num = 1;
                                 printf("Found circle at level %d\n",i);
                                 CircleBeginning_d = d2[tmp->JoinNode];

                                 while(1){
                                    if(num % 2 != 0){
                                     tmp->EdgeLoad -= i;
                                      tmp->VtoV->EdgeLoad -= i;
                                 }
                              else{
                                 tmp->EdgeLoad += i;
                                  tmp->VtoV->EdgeLoad += i;
                              }
                             num++;

                        if(DFS_InV1 == 0){
                           DFS_ColorV2[ctr] = WHITE;
                           tmp2 = V_2[ctr];
                           while(tmp2!=NULL){
                              if(d2[ctr] - d1[tmp2->JoinNode] == 1){
                                 tmp = tmp2;
                                 break;
                              }
                              else tmp2 = tmp2->next;
                           }
                           //tmp = DFS_EdgeP2[ctr];
                           ctr = tmp->JoinNode;
                           DFS_InV1 = 1;
                        }
                        else{
                           DFS_ColorV1[ctr] = WHITE;
                           tmp2 = V_1[ctr];
                           while(tmp2!=NULL){
                              if(d1[ctr] - d2[tmp2->JoinNode] == 1){
                                 tmp = tmp2;
```

```
                        break;
                    }
                    else tmp2 = tmp2->next;
                }
                //tmp = DFS_EdgeP1[ctr];
                ctr = tmp->JoinNode;
                DFS_InV1 = 0;
            }
             if(d2[tmp->JoinNode] == CircleBeginning_d){
                DFS_time = d2[ctr];
                tmp->EdgeLoad += i;
                tmp->VtoV->EdgeLoad += i;
                tmp1 = tmp;
                break;
             }
          }
       }
       else tmp = tmp->next;
    }
    else tmp = tmp->next;
  }
}
else{/*We are at V2*/
  if(DFS_ColorV1[tmp->JoinNode] == WHITE){
    DFS_P1[ctr] = tmp->JoinNode;
    DFS_EdgeP1[ctr] = tmp;
    DFS_InV1 = 1;
                                        DFS_VISIT(V_1, V_2, tmp->JoinNode, i);
    break;
  }
  else{
    if(DFS_ColorV1[tmp->JoinNode] == GRAY){
      if(d2[ctr] - d1[tmp->JoinNode] > 1){
        int num = 1;
        printf("Found circle 2 at level %d\n",i);
        CircleBeginning_d = d1[tmp->JoinNode];

        while(1){
          if(num % 2 != 0){
            tmp->EdgeLoad -= i;
            tmp->VtoV->EdgeLoad -= i;
          }
          else{
            tmp->EdgeLoad += i;
            tmp->VtoV->EdgeLoad += i;
          }
          num++;

          if(DFS_InV1 == 0){
            DFS_ColorV2[ctr] = WHITE;
            tmp2 = V_2[ctr];
            while(tmp2!=NULL){
              if(d2[ctr] - d1[tmp2->JoinNode] == 1){
                tmp = tmp2;
                break;
              }
              else tmp2 = tmp2->next;
            }

            //tmp = DFS_EdgeP1[ctr];
            ctr = tmp->JoinNode;
            DFS_InV1 = 1;
```

```
                        }
                        else{
                           DFS_ColorV1[ctr] = WHITE;
                           tmp2 = V_1[ctr];
                           while(tmp2!=NULL){
                              if(d1[ctr] - d2[tmp2->JoinNode] == 1){
                                 tmp = tmp2;
                                 break;
                              }
                              else tmp2 = tmp2->next;
                           }
                           //tmp = DFS_EdgeP2[ctr];

                           ctr = tmp->JoinNode;
                           DFS_InV1 = 0;
                        }
                        if(d1[tmp->JoinNode] == CircleBeginning_d){
                           DFS_time = d1[ctr];
                           tmp->EdgeLoad += i;
                           tmp->VtoV->EdgeLoad += i;
                           tmp1 = tmp;
                           break;
                         }
                     }
                  }
                  else tmp = tmp->next;
               }
               else tmp = tmp->next;
            }
         }
                              }
                              else tmp = tmp->next;
                     }
      tmp1 = tmp1->next;
         }

         if(DFS_InV1 == 1){
                  DFS_ColorV1[ctr] = BLACK;
                  f1[ctr] = DFS_time+1;
      DFS_InV1 = 0;
         }
         else{
                  DFS_ColorV2[ctr] = BLACK;
                  f2[ctr] = DFS_time+1;
      DFS_InV1 = 1;
         }
         DFS_time++;
}

void Partition(set V_1[], set V_2[])
{

   int ctr;
   set temp;
         set tmp[groupBctr];/*for swap*/
   int M[groupBctr], M1[groupBctr], M2[groupBctr];
   int M21[groupBctr], M22[groupBctr];
   int k = 0, D = 0, d, NoNds = 0;/*NoNds: Number of Nodes*/
         int MaxNodes = 0, mxn = 0;/*The number of tnodes that have max degree*/
         int value;

   for(ctr = 0; ctr < groupBctr; ctr++){
```

```
   NodeDegree[ctr] = 0;
   temp = V_1[ctr];
   while(temp != NULL){
      NodeDegree[ctr] += temp->EdgeLoad;
      temp = temp->next;
   }
}
for(ctr = 0; ctr < groupBctr; ctr++){
   NodeDegree2[ctr] = 0;
   temp = V_2[ctr];
   while(temp != NULL){
      NodeDegree2[ctr] += temp->EdgeLoad;
      temp = temp->next;
   }
}

/*The way the graph is constracted all vertices have the
        maximum degree witch is equal to w (NumberOfColors) so we
                only need to check only one to see if max degree is odd or even.*/

if(NodeDegree[0] % 2 == 0)
   EulerSplit(V_1, V_2);/*This is the wanted Partition*/
else{
                D = NodeDegree[0];
                for(ctr = 0; ctr < groupBctr; ctr++){
                        M[ctr] = 1; /*All nodes n graph have max degree*/
                        MaxNodes++;
                        M1[ctr] = 0;
                        M2[ctr] = 0;
                        M21[ctr] = 0;
                        M22[ctr] = 0;
                }

                if((D-1) % 4 == 0){/*Finding of k and from formula
                                                4k+d = D, d = -1 or 1*/
                        k = (D-1)/4;
                        d = 1;
                }
                else{
                        k = (D+1)/4;
                        d = -1;
                }
                /*printf("\nD: %d, k: %d, d: %d\n",D,k,d);*/

   EulerSplit(V_1, V_2);

                /*We must find whitch one of H1V1 and H2V1 have degree 2k+d.
                        If its H2V1 the we must swap H1v1 with H2v2 and H1V2 with H2v2.*/

                for(ctr = 0; ctr < groupBctr; ctr++){/*We must find the degrees of one

                        of H1V1, H2V1.*/
                        NodeDegree[ctr] = 0;
        temp = H1V1[ctr];
              while(temp != NULL){
              NodeDegree[ctr] += temp->EdgeLoad;
           temp = temp->next;
              }
        }

                NoNds = 0;/*Looking how many nodes in H1V1 have degree 2k+d*/
                for(ctr = 0; ctr < groupBctr; ctr++){
```

```
                        if(M[ctr] == 1)
                                if(NodeDegree[ctr] == 2*k+d)
                                        NoNds++;
                }

                if(MaxNodes % 2 == 0)
                        mxn = MaxNodes/2;
                else
                        mxn = (MaxNodes+1)/2;

                if(NoNds < mxn){/*We must swap H1V1 and H2V1 and then H1V2 and H2V2*/
                        for(ctr = 0; ctr < groupBctr; ctr++){
                                tmp[ctr] = H1V1[ctr];
                                H1V1[ctr] = H2V1[ctr];
                                H2V1[ctr] = tmp[ctr];
                        }
                        for(ctr = 0; ctr < groupBctr; ctr++){
                                tmp[ctr] = H1V2[ctr];
                                H1V2[ctr] = H2V2[ctr];
                                H2V2[ctr] = tmp[ctr];
                        }
                }
                for(ctr = 0; ctr < groupBctr; ctr++){/*We must find the degrees of one

                        of H1V1.*/
                        NodeDegree[ctr] = 0;
        temp = H1V1[ctr];
             while(temp != NULL){
             NodeDegree[ctr] += temp->EdgeLoad;
           temp = temp->next;
             }
        }
                for(ctr = 0; ctr < groupBctr; ctr++){
                        if(M[ctr] == 1){
                                if(NodeDegree[ctr] == 2*k+d)
                                        M1[ctr] = 1;
                        }
                }
                for(ctr = 0; ctr < groupBctr; ctr++)
                        M2[ctr] = M[ctr] - M1[ctr];

                value = FindNorm(M2, groupBctr);


                if(value != 0){/*We need replica only if we'll go inside the while*/
                        for(ctr = 0; ctr < groupBctr; ctr++){/*Must have a copy of the euler split resut*/
        temp = H1V1[ctr];
                   while(temp != NULL){
                   InsertSet(&RepH1V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
                                                InsertSet(&Rep1H1V1[ctr],temp->JoinNode, temp->EdgeLoad,
temp->PositionNumber);
                temp = temp->next;
        }
                }
                   for(ctr = 0; ctr < groupBctr; ctr++){
                temp = RepH1V1[ctr];
                   while(temp != NULL){

                        InsertSet(&RepH1V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                        RepH1V1[ctr]->VtoV = RepH1V2[temp->JoinNode];
                        RepH1V2[temp->JoinNode]->VtoV = RepH1V1[ctr];
                        temp = temp->next;
```

```
            }
               }
   for(ctr = 0; ctr < groupBctr; ctr++){/*Must have a copy of the euler split resut*/
   temp = H2V1[ctr];
while(temp != NULL){
            InsertSet(&RepH2V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
            InsertSet(&Rep1H2V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
            temp = temp->next;
      }
}
               for(ctr = 0; ctr < groupBctr; ctr++){
            temp = RepH2V1[ctr];
               while(temp != NULL){

                              InsertSet(&RepH2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                              RepH2V1[ctr]->VtoV = RepH2V2[temp->JoinNode];
                              RepH2V2[temp->JoinNode]->VtoV = RepH2V1[ctr];
                              temp = temp->next;
                  }
               }

            }

            while(value != 0){puts("In While!!!!!!!!!!");
                        for(ctr = 0; ctr < groupBctr; ctr++){/*We must clearH1V1 and H2V1*/
                                    H1V1[ctr] = NULL; H1V2[ctr] = NULL;
                                    H2V1[ctr] = NULL; H2V2[ctr] = NULL;
                        }
                        EulerSplit(RepH2V1, RepH2V2);

                        for(ctr = 0; ctr < groupBctr; ctr++){/*We must find the degrees of one

                        of H1V1, H2V1.(Its H21 and H22 actually,

                        as written at the algorithm)*/
                                    NodeDegree[ctr] = 0;
            temp = H1V1[ctr];
            while(temp != NULL){
            NodeDegree[ctr] += temp->EdgeLoad;
               temp = temp->next;
                  }
            }
                        MaxNodes = 0;/*Must find number of nodes in M2*/
                        for(ctr = 0; ctr < groupBctr; ctr++)
                                    if(M2[ctr] == 1)
                                                MaxNodes++;

                        NoNds = 0;/*Looking how many nodes in H1V1 have degree k+d*/
                        for(ctr = 0; ctr < groupBctr; ctr++){
                                    if(M2[ctr] == 1)
                                                if(NodeDegree[ctr] == k+d)
                                                            NoNds++;
                        }
                        if(MaxNodes % 2 == 0)
                                    mxn = MaxNodes/2;
                        else
                                    mxn = (MaxNodes+1)/2;

                        if(NoNds < mxn){/*We must swap H1V1 and H2V1 and then H1V2 and H2V2*/
                                    for(ctr = 0; ctr < groupBctr; ctr++){
                                                tmp[ctr] = H1V1[ctr];
                                                H1V1[ctr] = H2V1[ctr];
```

```
                                        H2V1[ctr] = tmp[ctr];
                                }
                                for(ctr = 0; ctr < groupBctr; ctr++){
                                        tmp[ctr] = H1V2[ctr];
                                        H1V2[ctr] = H2V2[ctr];
                                        H2V2[ctr] = tmp[ctr];
                                }
                        }

                        /*We must find again the degrees of the H1V1*/
                        for(ctr = 0; ctr < groupBctr; ctr++)
                                NodeDegree[ctr] = 0;
                                for(ctr = 0; ctr < groupBctr; ctr++){/*We must find the degrees of one

                                of H1V1.*/

                        temp = H1V1[ctr];
                              while(temp != NULL){
                              NodeDegree[ctr] += temp->EdgeLoad;
                           temp = temp->next;
                           }
                }

                        for(ctr = 0; ctr < groupBctr; ctr++){/*Finding of M21*/
                                if(M2[ctr] == 1){
                                        if(NodeDegree[ctr] == k+d)
                                                M21[ctr] = 1;
                                }
                        }

                        for(ctr = 0; ctr < groupBctr; ctr++)/*Finding of M22*/
                                M22[ctr] = M2[ctr] - M21[ctr];

                        if(k % 2 == 0){/*Finding of New RepH1V1 and RepH2V1*/
                                for(ctr = 0; ctr < groupBctr; ctr++)
                                        Rep1H1V2[ctr] = NULL; /*Clear V2 set of Rep1H1V2 cause

we'll change Rep1H1V1. We'll fix this

later on*/

                        Union(Rep1H1V1, H1V1); /*(H1 = H1 U H21). After union we must fix Rep1H1V2*/

                        for(ctr = 0; ctr < groupBctr; ctr++){
                        temp = Rep1H1V1[ctr];
                           while(temp != NULL){
                                InsertSet(&Rep1H1V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                                Rep1H1V1[ctr]->VtoV = Rep1H1V2[temp->JoinNode];
                                Rep1H1V2[temp->JoinNode]->VtoV = Rep1H1V1[ctr];
                                temp = temp->next;
                        }
                                }/*Finished with Rep1H1V1 and Rep1H1V2*/

                                for(ctr = 0; ctr < groupBctr; ctr++){/*Must clear all graph Rep1H2*/
                                        Rep1H2V1[ctr] = NULL;
                                        Rep1H2V2[ctr] = NULL;
                                }
        for(ctr = 0; ctr < groupBctr; ctr++){/*RepH2V1 = H2V1 (H2 = H22)*/
        temp = H2V1[ctr];
        while(temp != NULL){
        InsertSet(&Rep1H2V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
              temp = temp->next;
```

```
}
}
            for(ctr = 0; ctr < groupBctr; ctr++){
        temp = Rep1H2V1[ctr];
            while(temp != NULL){

                    InsertSet(&Rep1H2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                    Rep1H2V1[ctr]->VtoV = RepH2V2[temp->JoinNode];
                  Rep1H2V2[temp->JoinNode]->VtoV = Rep1H2V1[ctr];
                temp = temp->next;
            }
            }/*Finished with Rep1H2V1 and Rep1H2V2, Now we must fix RepH2V1 and RepH2V2
                                    witch we're going to use to make an Euler Split again.*/

for(ctr = 0; ctr < groupBctr; ctr++){/*creating RepH2V1 (needed to

                                make the Euler Split))*/
temp = Rep1H2V1[ctr];
while(temp != NULL){
InsertSet(&RepH2V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
    temp = temp->next;
}
}
            for(ctr = 0; ctr < groupBctr; ctr++){
        temp = RepH2V1[ctr];
            while(temp != NULL){

                    InsertSet(&RepH2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                    RepH2V1[ctr]->VtoV = RepH2V2[temp->JoinNode];
                  RepH2V2[temp->JoinNode]->VtoV = RepH2V1[ctr];
                temp = temp->next;
            }
            }/*End of making graph for the Euler Split.*/

            }
            else{

                for(ctr = 0; ctr < groupBctr; ctr++){/*Must clear all graph Rep1H1*/
                        Rep1H1V1[ctr] = NULL;
                        Rep1H1V2[ctr] = NULL;
                }
for(ctr = 0; ctr < groupBctr; ctr++){/*RepH1V1 = H2V1 (H1 = H22)*/
temp = H2V1[ctr];
while(temp != NULL){
InsertSet(&Rep1H1V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
    temp = temp->next;
}
}
            for(ctr = 0; ctr < groupBctr; ctr++){
        temp = Rep1H1V1[ctr];
            while(temp != NULL){

                    InsertSet(&Rep1H1V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                    Rep1H1V1[ctr]->VtoV = RepH1V2[temp->JoinNode];
                  Rep1H1V2[temp->JoinNode]->VtoV = Rep1H1V1[ctr];
                temp = temp->next;
            }
            }/*End of finding graph Rep1H1V1.*/
                for(ctr = 0; ctr < groupBctr; ctr++)
                        Rep1H2V2[ctr] = NULL; /*Clear V2 set of Rep1H2V2 cause

we'll change Rep1H2V1. We'll fix this later on*/
```

```
                    Union(Rep1H2V1, H1V1); /*(H2 = H2 U H21). After union we must fix Rep1H2V2*/

                    for(ctr = 0; ctr < groupBctr; ctr++){
                    temp = Rep1H2V1[ctr];
                       while(temp != NULL){
                              InsertSet(&Rep1H2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                              Rep1H2V1[ctr]->VtoV = Rep1H2V2[temp->JoinNode];
                              Rep1H2V2[temp->JoinNode]->VtoV = Rep1H2V1[ctr];
                              temp = temp->next;
                    }
                              }/*Finished with Rep1H2V1 and Rep1H2V2. After that we must create
RepH2V1 and RepH2V2*/

          for(ctr = 0; ctr < groupBctr; ctr++){/*creating RepH2V1 (needed to

                                        make the Euler Split))*/
          temp = Rep1H2V1[ctr];
          while(temp != NULL){
          InsertSet(&RepH2V1[ctr],temp->JoinNode, temp->EdgeLoad, temp->PositionNumber);
               temp = temp->next;
          }
          }
                    for(ctr = 0; ctr < groupBctr; ctr++){
                    temp = RepH2V1[ctr];
                       while(temp != NULL){

                                  InsertSet(&RepH2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                                  RepH2V1[ctr]->VtoV = RepH2V2[temp->JoinNode];
                                RepH2V2[temp->JoinNode]->VtoV = RepH2V1[ctr];
                               temp = temp->next;
                    }
                    }/*End of making graph for the Euler Split.*/

                    }

                    for(ctr = 0; ctr < groupBctr; ctr++){/*Finding of M1 and M2*/
                              if(M1[ctr] != M21[ctr])
                                        M1[ctr] == 1;
                              M2[ctr] = M22[ctr];
                    }

                    value = FindNorm(M2, groupBctr);/*Must find Norm of M2 again to

          check if while loop will

          continue.*/

                    if(value == 0){

                              for(ctr = 0; ctr < groupBctr; ctr++){/*We must clear H1V1 and H2V1*/
                                        H1V1[ctr] = NULL; H1V2[ctr] = NULL;
                                        H2V1[ctr] = NULL; H2V2[ctr] = NULL;
                              }/*After that we are going to copy Rep1H1V1 to H1V1 and
                                                  Rep1H2V1 to H2V1 cause these are the two graphs of the
                                                  partition function.*/

                              for(ctr = 0; ctr < groupBctr; ctr++){
          temp = Rep1H1V1[ctr];
                    while(temp != NULL){
                                                  InsertSet(&H1V1[ctr],temp->JoinNode, temp->EdgeLoad,
temp->PositionNumber);
```

```
                    temp = temp->next;
                    }
                        }

                            for(ctr = 0; ctr < groupBctr; ctr++){
                        temp = H1V1[ctr];
                            while(temp != NULL){

                                    InsertSet(&H1V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                                    H1V1[ctr]->VtoV = H1V2[temp->JoinNode];
                                    H1V2[temp->JoinNode]->VtoV = H1V1[ctr];
                                    temp = temp->next;
                        }
                        }
                                    for(ctr = 0; ctr < groupBctr; ctr++){
                    temp = Rep1H2V1[ctr];
                            while(temp != NULL){
                                                        InsertSet(&H2V1[ctr],temp->JoinNode, temp->EdgeLoad,
temp->PositionNumber);
                    temp = temp->next;
                    }
                        }
                            for(ctr = 0; ctr < groupBctr; ctr++){
                        temp = H2V1[ctr];
                            while(temp != NULL){

                                    InsertSet(&H2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
                                    H2V1[ctr]->VtoV = H2V2[temp->JoinNode];
                                    H2V2[temp->JoinNode]->VtoV = H2V1[ctr];
                                    temp = temp->next;
                        }
                        }
                        }

                            k = (D-k)/2;/*New values of k and d*/
                            d = -d;

                    }

   }/*End of else*/
}
void EulerSplit(set V_1[], set V_2[])
{
   int i = 0, j = -1, ctr, end = 0, k = 0;
   set temp;

   for(ctr = 0; ctr < groupBctr; ctr++){
      NodeDegree[ctr] = 0;
      NodeDegree2[ctr] = 0;
   }

   for(ctr = 0; ctr < groupBctr; ctr++){
      temp = V_1[ctr];
      while(temp != NULL){
         NodeDegree[ctr] += temp->EdgeLoad;
         temp = temp->next;
      }
   }
   for(ctr = 0; ctr < groupBctr; ctr++){
      temp = V_2[ctr];
      while(temp != NULL){
         NodeDegree2[ctr] += temp->EdgeLoad;
```

```
      temp = temp->next;
   }
}

for(ctr = 0; ctr < groupBctr; ctr++){/*Coping data from V1 to H1V1 and H2V2
                        assigning multiplisity zero to every node*/
   temp = V_1[ctr];
   while(temp != NULL){
      InsertSet(&H1V1[ctr],temp->JoinNode, 0, temp->PositionNumber);
      InsertSet(&H2V1[ctr],temp->JoinNode, 0, temp->PositionNumber);
      temp->VtoH1 = H1V1[ctr];
      temp->VtoH2 = H2V1[ctr];
      temp = temp->next;
   }
}

while(1){/*While there are paths*/

   i = FindOdd(NodeDegree, groupBctr);/*Finding node with
                        odd degree in V1*/
   if(i == -1)
      j = FindOdd(NodeDegree2, groupBctr);/*Finding node with
                        odd degree in V2 if there are no in V1*/

   if(i == -2)/*There are no more paths so end of while loop*/
      break;

   if(j == -1){
      if(i != -1)
         ErasePath(V_1, V_2, i);
      else{
         k = FindEven(NodeDegree, groupBctr);
         if(k != -1)
            ErasePath(V_1, V_2, k);
      }
   }
   else if(j != -1)
      ErasePath(V_2, V_1, j);

   i = 0;/*Must have the counters initiative again cause we're going
         to use them again after ending with one path*/
   j = -1;

}
for(ctr = 0; ctr < groupBctr; ctr++){/*In our graphs we have some edges
                     with multiplisity of zero. We must delete them*/
temp = H1V1[ctr];
   while(temp != NULL){
      if(temp->EdgeLoad == 0)
         DeleteSet(&H1V1[ctr], temp);
      temp = temp->next;
   }
}
for(ctr = 0; ctr < groupBctr; ctr++){
   temp = H2V1[ctr];
   while(temp != NULL){
      if(temp->EdgeLoad == 0)
         DeleteSet(&H2V1[ctr], temp);
      temp = temp->next;
   }
}
/*We must create and H1V2 and H2V2 from H1V1 and H2V1*/
```

```
   for(ctr = 0; ctr < groupBctr; ctr++){
      temp = H1V1[ctr];
      while(temp != NULL){

       InsertSet(&H1V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
       H1V1[ctr]->VtoV = H1V2[temp->JoinNode];
       H1V2[temp->JoinNode]->VtoV = H1V1[ctr];
       temp = temp->next;
      }
   }

   for(ctr = 0; ctr < groupBctr; ctr++){
      temp = H2V1[ctr];
      while(temp != NULL){

       InsertSet(&H2V2[temp->JoinNode], ctr, temp->EdgeLoad, 0);
       H2V1[ctr]->VtoV = H2V2[temp->JoinNode];
       H2V2[temp->JoinNode]->VtoV = H2V1[ctr];
       temp = temp->next;
      }
   }
//    free(temp);
}
void ErasePath(set lst1[], set lst2[], int num)
{
   set ptr, tmp,temp2;
   int Odd = 0,i;
            int InV1 = 0;
            int JoinNum;

   ptr = lst1[num];
   tmp = ptr;

   while(1){
      if(ptr->VtoH1 != NULL){ /*if we are at V1*/
         if(ptr->EdgeLoad % 2 == 0){
            ptr->VtoH1->EdgeLoad = (ptr->EdgeLoad)/2;
            ptr->VtoH2->EdgeLoad = (ptr->EdgeLoad)/2;
            Odd = 0;
         }
         else{
            ptr->VtoH1->EdgeLoad = (int)(ptr->EdgeLoad)/2 + 1;
            ptr->VtoH2->EdgeLoad = (int)(ptr->EdgeLoad)/2;
            Odd = 1;
         }
                                    InV1 = 1;/*We are at V1*/
         /*We must discrease the number of degree of the nodes we
         are going to erase*/
         NodeDegree[ptr->VtoV->JoinNode] = NodeDegree[ptr->VtoV->JoinNode] - ptr->EdgeLoad;
         NodeDegree2[ptr->JoinNode] = NodeDegree2[ptr->JoinNode] - ptr->EdgeLoad;
      }
      else if(ptr->VtoH1 == NULL){ /*if we are at V2*/
         if(ptr->EdgeLoad % 2 == 0){
            ptr->VtoV->VtoH1->EdgeLoad = (ptr->EdgeLoad)/2;
            ptr->VtoV->VtoH2->EdgeLoad = (ptr->EdgeLoad)/2;
            Odd = 0;
         }
         else{

            ptr->VtoV->VtoH1->EdgeLoad = (int)(ptr->EdgeLoad)/2;
            ptr->VtoV->VtoH2->EdgeLoad = (int)(ptr->EdgeLoad)/2 + 1;
            Odd = 1;
```

```
   }

   NodeDegree2[ptr->VtoV->JoinNode] = NodeDegree2[ptr->VtoV->JoinNode] - ptr->EdgeLoad;
   NodeDegree[ptr->JoinNode] = NodeDegree[ptr->JoinNode] - ptr->EdgeLoad;
}


if(Odd == 1){/*If multiplisity is odd we must change set*/
   JoinNum = ptr->JoinNode;
   ptr = ptr->VtoV;

                        if(InV1 == 1)/*If we are at V1*/
      DeleteSet(&lst1[ptr->JoinNode], tmp);/*delete node*/
                        else
                                DeleteSet(&lst2[ptr->JoinNode], tmp);
   /*InV1 = 0;  /*We change set From V1 to V2 so must
         update information*/

   if(InV1 == 1)
      InV1 = 0;
   else
      InV1 = 1;


}
else /*if(Odd == 0)*/{
                        JoinNum = ptr->VtoV->JoinNode;
                        if(InV1 == 1)/*If we are at V1**/
      DeleteSet(&lst2[ptr->JoinNode], tmp->VtoV);/*delete node*/
                        else
                                DeleteSet(&lst1[ptr->JoinNode], tmp->VtoV);

               }


tmp = ptr; /*tmp has the same value with ptr*/

if(ptr->next != NULL){
   ptr = ptr->next;
                        if(InV1 == 1)/*If we are at V1*/
      DeleteSet(&lst1[JoinNum], tmp);/*delete node*/
                        else
                                DeleteSet(&lst2[JoinNum], tmp);
   tmp = ptr;
}
else{
      if(ptr->prev != NULL){
               ptr = ptr->prev;
               if(InV1 == 1)/*If we are at V1*/
                        DeleteSet(&lst1[JoinNum], tmp);/*delete node*/
               else
                        DeleteSet(&lst2[JoinNum], tmp);
               tmp = ptr;
      }
      else /*if(ptr->nex t == NULL && ptr->prev == NULL)*/{
               if(InV1 == 1)/*If we are at V1*/
                        DeleteSet(&lst1[JoinNum], ptr);/*delete last node*/
               else
                        DeleteSet(&lst2[JoinNum], ptr);

      break;
      }
```

```
        }
        tmp = ptr;
        Odd = 0;/*We must initiate the number for the next*/
                        InV1 = 0;/*step of the while loop.*/
    }
//          free(tmp);
//          free(ptr);
}
void Union(set V_1[], set V_2[])
{
    int ctr = 0, different = 0;
    set tmp1, tmp2, tmp3, tmp4, tmp5;
            int LtoG = 0;

    tmp1 = V_1[0];
    tmp2 = V_2[0];
    if(tmp1->next != NULL && tmp2->next != NULL){/*We must check if the order
                            is different in graph 1 and graph 2*/
        if(tmp1->JoinNode < tmp1->next->JoinNode &&
                            tmp2->JoinNode > tmp2->next->JoinNode)
            different = 1;
        if(tmp1->JoinNode > tmp1->next->JoinNode &&
                            tmp2->JoinNode < tmp2->next->JoinNode){
            different = 1;
                        }
    }
            if(tmp1->next != NULL)
                        if(tmp1->JoinNode > tmp1->next->JoinNode)
                                    LtoG = 1;

    for(ctr = 0; ctr < groupBctr; ctr++){
        tmp1 = V_1[ctr];
        tmp2 = V_2[ctr];

        if(different == 1){ /*If the order is different we must move
                        tmp2 to the end*/
            while(tmp2->next != NULL)
                tmp2 = tmp2->next;
        }
        while(tmp1 != NULL && tmp2 != NULL){
                                    if(LtoG == 0){/*If we have first < second*/
            if(tmp1 != NULL && tmp2 != NULL){
                        if(tmp2->JoinNode < tmp1->JoinNode && tmp1->prev == NULL){
                                    InsertSet(&V_1[ctr], tmp2->JoinNode, tmp2->EdgeLoad, tmp2->PositionNumber);
                                    tmp1 = tmp1->prev;
                                    if(different == 1)
                                                tmp2 = tmp2->prev;
                                    else
                                                tmp2 = tmp2->next;
                        }
                        else if(tmp2->JoinNode > tmp1->JoinNode && tmp1->next != NULL){
                                    if(tmp1->next->JoinNode > tmp2->JoinNode){

                                                if(different == 1)
                                                            tmp4 = tmp2->prev;
                                                else
                                                            tmp4 = tmp2->next;

                                                tmp3 = tmp1->next;
                                                tmp1->next = tmp2;
                                                tmp3->prev = tmp2;
```

```
                                        tmp2->next = tmp3;
                                        tmp2->prev = tmp1;
                                        tmp1 = tmp3;
                                        tmp5 = tmp1;
                                        tmp2 = tmp4;

                        }
                        else if(tmp1->next->JoinNode <= tmp2->JoinNode){
                                tmp5 = tmp1;
                                tmp1 = tmp1->next;
        }
}
                        else if(tmp2->JoinNode == tmp1->JoinNode){
                                tmp1->EdgeLoad += tmp2->EdgeLoad;
                                tmp5 = tmp1;
                                tmp1 = tmp1->next;
                                if(different == 1)
                                        tmp2 = tmp2->prev;
                                else
                                        tmp2 = tmp2->next;
                        }
        }
                        if(tmp1 == NULL && tmp2 != NULL){
                                tmp5->next = tmp2;
                                tmp2->prev = tmp5;
                                tmp2 = NULL;
                        }
}
                        else/*if(LtoG == 1)*/{/*If first > second*/

                                if(tmp1 != NULL && tmp2 != NULL){
                                        if(tmp2->JoinNode > tmp1->JoinNode && tmp1->prev == NULL){
                                                InsertSet(&V_1[ctr], tmp2->JoinNode, tmp2->EdgeLoad, tmp2-
>PositionNumber);
                                                tmp1 = tmp1->prev;
                                                if(different == 1)
                                                        tmp2 = tmp2->prev;
                                                else
                                                        tmp2 = tmp2->next;
                                        }
                                        else if(tmp2->JoinNode < tmp1->JoinNode && tmp1->next != NULL){
                                                if(tmp1->next->JoinNode > tmp2->JoinNode){

                                                        if(different == 1)
                                                                tmp4 = tmp2->prev;
                                                        else
                                                                tmp4 = tmp2->next;

                                                        tmp3 = tmp1->next;
                                                        tmp1->next = tmp2;
                                                        tmp3->prev = tmp2;

                                                        tmp2->next = tmp3;
                                                        tmp2->prev = tmp1;

                                                        tmp1 = tmp3;
                                                        tmp5 = tmp1;
                                                        tmp2 = tmp4;

                                                }
                                                else if(tmp1->next->JoinNode >= tmp2->JoinNode){
                                                        tmp5 = tmp1;
```

```
                                                tmp1 = tmp1->next;
                                        }
                                }
                                else if(tmp2->JoinNode == tmp1->JoinNode){
                                        tmp1->EdgeLoad += tmp2->EdgeLoad;
                                        tmp5 = tmp1;
                                        tmp1 = tmp1->next;
                                        if(different == 1)
                                                tmp2 = tmp2->prev;
                                        else
                                                tmp2 = tmp2->next;
                                }
                        }
                if(tmp1 == NULL && tmp2 != NULL){
                                        tmp5->next = tmp2;
                                        tmp2->prev = tmp5;
                                        tmp2 = NULL;
                }
                                }
        }
    }
//    free(tmp1);
//    free(tmp2);
//    free(tmp3);
//    free(tmp4);
//            free(tmp5);
}
void InsertSet(set *lst, int nds, int lds, int pst)
{
   set newptr;

   if((newptr = (set )malloc(sizeof(struct setstr)))==NULL){
      fprintf(stderr,"Cannot allocate memory. EXITING.");
      exit(1);
   }

   newptr->JoinNode = nds;
   newptr->EdgeLoad = lds;
   newptr->PositionNumber = pst;

   newptr->next = *lst;
   if(*lst != NULL)
      newptr->next->prev = newptr;
   *lst = newptr;
   newptr->prev = NULL;
}
void DeleteSet(set *lst, set ptr)
{
   if(ptr->prev != NULL)
      ptr->prev->next = ptr->next;
      else{
                                *lst = ptr->next;
                        }
   if(ptr->next != NULL)
      ptr->next->prev = ptr->prev;


}
int FindOdd(int array[MAX], int length)
{
   int i = 0, j = -1, end = 0;
```

```
   for(i = 0; i < length; i++){
      if(array[i] % 2 != 0){
         j = i;
         break;
      }
      if(array[i] == 0)
         end++;
   }
   if(end == groupBctr)
      return (-2);

   return (j);
}
int FindEven(int array[], int length){
   int i = 0, j = -1;

   for(i = 0; i < length; i++)
      if(array[i] % 2 == 0 && array[i] != 0){
         j = i;
         break;
      }
   return (j);
}
int FindNorm(int array[], int length)
{
            int i, num;
            num = array[0];
            for(i = 0; i < length-1; i++)
                        num += array[i+1];
            return (num);
}
```

Βιβλιογραφία

1. C. Berge.
Graphs and Hypergraphs.
North Holland, Amsterdam, 1973.

2. C. Berge.
The Theory of Graphs and its Applications.
John Wiley, 1962.

3. R. Cole, J.E. Hopcroft.
On Edge Coloring Bipartite Graphs.
SIAM Journal of Computing, 11(3), 1982.

4. T. Erlebach, K. Jansen, C. Kaklamanis, M. Mihail and P. Persiano.
Optimal Wavelength Routing on Directed Fiber Trees.
Theoretical Computer Science, 333(1-2), 1999.

5. H. Gabow.
Using Euler partitions to edge color bipartite multigraphs.
International Journal of Computing Information Science, 5(1976).

6. H. Gabow and O. Kariv.
Algorithms for Edge Coloring Dipartite Graphs and Multigraphs.
SIAM Journal of Computing. 5, 1976.

7. M. Garey, D. Jonshon, G. Miller and C. Papadimitriou.
The Complexity of Coloring Circular Arcs and Cords.
SIAM J. Alg. Disc. Math., Vol 1(2), 1980.

8. P.E. Green.
Δίκτυα Οπτικών Ινών.
Μεταφραση Κ. Καρούμπαλος,
Εκδ. Παπασωτηρίου. Αθήνα 1994.

9. D.S. Hockbaum, T. Nishizeki and D.B. Shmoys.
Better than Best Possible Algorithm to Edge Color Multigraphs.
Journal of Algorithms, 7(1), 1986.

10. I. Holyer.
The NP-completeness of Edge Coloring.
SIAM Journal of Computing, 10, 1981.

11. I.A. Karapetian.
On the Coloring of Circular Arc Graphs.
Docladi (Reports) of the Academy of Science of the Armenian Soviet Socialist Republic 70(5), 1980, (in Russian). English translation by D. Gamarnik, D. Williamson and N. Edwards in http://www.orie.cornell.edu/~nedwards/wdm-routing/.

12. R. M. Karp.
Redusibility Among Combinational Problems.
Comlexity of Computer Computations, edited by J.W. Thatcher and R.E. Miller, Plenum Press, New York, 1972.

13. G. Li and R. Simba.
On the Wave length Assignment Problem in Multifiber WDM Star and Ring Networks.
Proceedings of INFOCOM 2000.

14. C. Lund and M. Yannakakis.
On the Hardness of Approximating Minimization Problems.
Proc. ACM Symposium On the Theory of Computing STOC 1993. pp. 286-293.

15. L. Margara and J. Simon.
Wave length Assignment problem in All-Optical Networks with k Fibers per Link.
Proceedings ICALP 2000.

16. M. Mihail, C. Kaklamanis and S. Rao.
Efficient Access to Optical Bandwidth.
Proceedings of the 36th Annual Symposium on Foundations of Computer Science FOCS 1995.

17. C. Nomikos, A. Pagourtzis and S. Zachos.
Routing and Path Multi-Coloring.
Information Processing Letters 80(2000) pp. 249-256

18. C. Nomikos.
Path Colorings in Graphs.
Ph. D. Thesis, Dept. of Electrical and Computer Engineering, NTUA , Athens, 1997.

19. S. Olariu.
An Optimal Greedy Heuristic to Color Interval Graphs.
Information Processin Letters, 37, 1991.

20. P. Raghavan and E. Upfal.
Efficient Routing in All-Optical Networks.
Proceedings of the 26th Annual ACM Symposium on the Theory of Computing STOC 1994.

21. A. Schrijver.
Bipartite Edge Coloring in O(Δm) Time.
SIAM Journal of Computing, 28(3), 1999.

22. R. Tarjan.
Decomposition by Clique separators.
Discrete Mathematics, 55, 1985.