$\mu \Pi \lambda \forall$
Διαπανεπιστημιακό Πρόγραμμα Μεταπτυχιακών Σπουδών
στη Λογική και Θεωρία Αλγορίθμων και Υπολογισμού

Αλγόριθμοι δρομολόγησης και ανάθεσης συχνοτήτων σε πλήρως οπτικά δίκτυα: υλοποιήσεις και πειραματικές συγκρίσεις

Γεώργιος Γεωργίου

Διπλωματική εργασία

Επιβλέπων: Άρης Παγουρτζής

ΑΘΗΝΑ 2005

Περίληψη

Η πολύ γρήγορη και αξιόπιστη ανταλλαγή δεδομένων σε παγκόσμιας κλίμακας δίκτυα τα τελευταία χρόνια είναι εφικτή μέσω των εξελίξεων στην τεχνολογία και στην αρχιτεκτονική των δικτύων. Μια τέτοια σημαντική τεχνολογία είναι η χρήση των οπτικών ινών για τη μεταφορά τεράστιας ποσότητας πληροφορίας. Η αλματώδης ανάπτυξη της τεχνολογίας των οπτικών ινών οδηγεί σταδιακά σε δίκτυα τα οποία αποτελούνται μόνο από οπτικές συνδέσεις. Τα δίκτυα αυτά ονομάζονται πλήρως οπτικά δίκτυα.

Σε ένα πλήρως οπτικό δίκτυο με πολυπλεξία διαίρεσης μήκους κύματος πολλές συνδέσεις μπορούν να χρησιμοποιούν την ίδια οπτική ίνα ταυτόχρονα αν το σήμα μεταφέρεται πάνω σε διαφορετικά μήκη κύματος. Μια σύνδεση πρέπει να χρησιμοποιεί το ίδιο μήκος κύματος από την αρχή μέχρι το τέλος της σύνδεσης. Όμως τα διαθέσιμα μήκη κύματος είναι περιορισμένα σε αριθμό. Ο παραπάνω περιορισμός συνδέει τα πλήρως οπτικά δίκτυα με αλγοριθμικά προβλήματα χρωματισμού μονοπατιών.

Στο πρόβλημα χρωματισμού μονοπατιών θέλουμε να αναθέσουμε χρώματα (μήκη κύματος) και μονοπάτια σε ένα σύνολο αιτήσεων σύνδεσης έτσι ώστε μονοπάτια που χρησιμοποιούν την ίδια οπτική ίνα να έχουν διαφορετικά χρώματα.

Στην εργασία αυτή παρουσιάζονται και υλοποιούνται στο προγραμματιστικό περιβάλλον της PASCAL αλγόριθμοι για χρωματισμό μονοπατιών σε δακτυλίους. Τέτοιου είδους αλγόριθμοι βρίσκουν μεγάλη εφαρμογή σε οπτικά δίκτυα, αφού ένα τέτοιο δίκτυο μπορεί να παρασταθεί από έναν δακτύλιο όπου οι κορυφές είναι οι κόμβοι που θέλουν να επικοινωνήσουν, τα μονοπάτια είναι οι αιτήσεις επικοινωνίας και τα χρώματα τα διαθέσιμα μήκη κύματος.

Τα δύο προβλήματα τα οποία μελετήθηκαν από τη βιβλιογραφία και για τα οποία υλοποιούνται οι αντίστοιχοι αλγόριθμοι είναι τα MaxPC και MaxRPC σε δακτυλίους. Το πρόβλημα της *εύρεσης μέγιστου χρωματισμού μονοπατιών* (**MaxPC**) είναι το εξής: Δοθέντος ενός γράφου G(V,E), ενός συνόλου μονοπατιών P και ενός αριθμού χρωμάτων w, ζητάμε μια λύση που θα χρωματίζει τον μέγιστο δυνατό αριθμό μονοπατιών ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα.

Maximum Path Coloring Problem (MaxPC)

Είσοδος: Ένας γράφος G(V,E), ένα σύνολο από μονοπάτια P και ένας αριθμός από διαθέσιμα χρώματα w.
Αποδεκτή λύση*:* Χρωματισμός ενός υποσυνόλου P'$\subseteq$ P έτσι ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα.
Στόχος: Μια λύση που χρωματίζει τον μέγιστο αριθμό μονοπατιών.

Το πρόβλημα *της δρομολόγησης και εύρεσης μέγιστου χρωματισμού μονοπατιών* (**MaxRPC**) είναι το εξής: Δοθέντος ενός γράφου G(V,E), ενός συνόλου ζευγαριών κόμβων (αιτήσεων σύνδεσης) και ενός αριθμού χρωμάτων w, ζητάμε μια αντιστοίχηση μονοπατιών σε ένα υποσύνολο αιτήσεων (δρομολόγηση) και ένα χρωματισμό αυτών των μονοπατιών, ώστε

επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα και επιπλέον να ικανοποιείται ο μέγιστος δυνατός αριθμός αιτήσεων.

Maximum Routing and Path Coloring Problem (MaxRPC)

Είσοδος: Ένας γράφος G(V,E), ένα σύνολο από ζευγάρια μονοπατιών (αιτήσεις σύνδεσης) και ένας αριθμός από διαθέσιμα χρώματα w.
Αποδεκτή λύση*:* Μια αντιστοίχηση μονοπατιών σε ένα υποσύνολο αιτήσεων (δρομολόγηση) και ένα χρωματισμό αυτών των μονοπατιών, ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα
Στόχος: Μια λύση που χρωματίζει τον μέγιστο αριθμό μονοπατιών δηλαδή ικανοποιεί τον μέγιστο αριθμό αιτήσεων.

Οι προσεγγιστικοί αλγόριθμοι που υλοποιήθηκαν πετυχαίνουν τον καλύτερο παράγοντα προσέγγισης από τους μέχρι τώρα γνωστούς αλγόριθμους για τέτοιου είδους προβλήματα. Μαζί με τους αλγόριθμους παρουσιάζονται και βελτιώσεις τους που όπως έδειξαν τα αντίστοιχα πειράματα που έγιναν πετυχαίνουν στην πράξη ακόμα καλύτερα αποτελέσματα. Κάποιες από αυτές τις βελτιώσεις είναι ιδέες που παρουσιάζονται για πρώτη φορά στα πλαίσια αυτής της εργασίας.

Το πρώτο κεφάλαιο της εργασίας είναι εισαγωγικό. Σ΄ αυτό παρουσιάζονται βασικές έννοιες για τα οπτικά δίκτυα και τη σχέση τους με γράφους και προβλήματα χρωματισμού σε γράφους. Στο δεύτερο κεφάλαιο παρουσιάζονται τα δύο προβλήματα χρωματισμού MaxPC και MaxRPC για τα οποία αναπτύχθηκαν οι αλγόριθμοι της εργασίας και γίνεται επισκόπηση των μέχρι τώρα γνωστών αποτελεσμάτων για τα προβλήματα αυτά. Στο τρίτο κεφάλαιο παρουσιάζονται οι αλγόριθμοι μαζί με τις βελτιώσεις τους. Στο τέταρτο κεφάλαιο γίνεται αναφορά στο προγραμματιστικό περιβάλλον της υλοποίησης και παρουσιάζονται οι τεχνικές προγραμματισμού, δομές δεδομένων και κώδικας από βιβλιοθήκες που χρησιμοποιήθηκαν. Στο πέμπτο κεφάλαιο παρουσιάζονται μια σειρά από πειράματα που έγιναν με σκοπό την σύγκριση των αλγορίθμων και των βελτιώσεών τους. Στο πέμπτο κεφάλαιο που μπορεί να χρησιμοποιηθεί και ως reference manual (εγχειρίδιο αναφοράς) γίνεται παρουσίαση των διαδικασιών που υλοποιήθηκαν. Στο τελευταίο κεφάλαιο παρουσιάζονται διάφορα συμπεράσματα που προέκυψαν κατά την ανάπτυξη της εργασίας και δίνονται ιδέες για βελτιώσεις. Στο παράρτημα Α και Β υπάρχει ο κώδικας της υλοποίησης σε PASCAL.

Κεφάλαιο 1

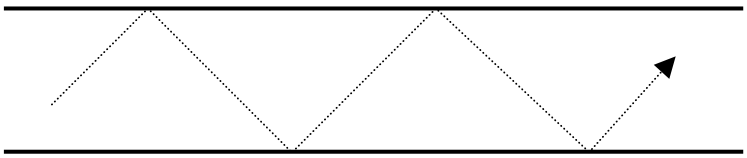
Εισαγωγή

1.1 Οπτικά δίκτυα

Η χρησιμοποίηση των οπτικών ινών στα σύγχρονα τηλεπικοινωνιακά δίκτυα αποτέλεσε επανάσταση στην τεχνολογία επικοινωνιών και είχε σαν αποτέλεσμα μεγαλύτερο ρυθμό μετάδοσης και λιγότερο αριθμό σφαλμάτων.

Ένα οπτικό σύστημα μετάδοσης αποτελείται από την πηγή φωτός, ένα μέσο μετάδοσης και έναν ανιχνευτή. Ένας παλμός φωτός αντιστοιχεί στο bit 1 ενώ η απουσία του στο bit 0. Το μέσο μετάδοσης είναι μια λεπτή ίνα γυαλιού. Ο ανιχνευτής δημιουργεί έναν ηλεκτρικό παλμό όταν πέφτει πάνω του φως. Μια οπτική ίνα όπου στο ένα άκρο της έχει μια πηγή φωτός και στο άλλο έναν ανιχνευτή αποτελεί ένα σύστημα μετάδοσης όπου το ηλεκτρικό σήμα μετατρέπεται σε παλμούς φωτός, μεταδίδεται και στη λήψη μετατρέπεται πάλι σε ηλεκτρικό σήμα.

Σε ένα τέτοιο οπτικό σύστημα η μετάδοση γίνεται χωρίς απώλειες λόγω του φαινομένου της *ολικής ανάκλασης* σύμφωνα με το οποίο το φως ανακλάται διαρκώς στα τοιχώματα της οπτικής ίνας με αποτέλεσμα να ταξιδεύει μέσα σε αυτή.



Σχήμα 1.1: Φαινόμενο ολικής ανάκλασης σε οπτική ίνα.

Επίσης λόγω της φύσης του φωτός που δεν επηρεάζεται από άλλα πεδία όπως συμβαίνει με τα ηλεκτρόνια η μετάδοση γίνεται με ελάχιστες παρεμβολές.

Αποτέλεσμα είναι ο πολύ υψηλός ρυθμός μετάδοσης που φτάνει τα 50.000 Gbps που όμως περιορίζεται στα 1Gbps λόγω των μετατροπέων ηλεκτρικών σημάτων σε οπτικά και αντίστροφα..

Εκτός από τον υψηλότερο ρυθμό μετάδοσης σε σχέση με τα συμβατικά δίκτυα το οπτικό σήμα μεταδίδεται χωρίς την ανάγκη ενίσχυσης σε αποστάσεις μέχρι και 30Km, χωρίς να επηρεάζεται από διακυμάνσεις της τάσης τροφοδοσίας και από ηλεκτρομαγνητικές παρεμβολές. Επιπλέον το μικρό τους βάρος και το ότι είναι πολύ λεπτές είναι ένα σημαντικό πλεονέκτημα σε σχέση με τα συμβατικά χάλκινα καλώδια.

Το κύριο μειονέκτημα της οπτικής τεχνολογίας είναι το αυξημένο κόστος σε σχέση με τα συμβατικά δίκτυα που οφείλεται κυρίως στους οπτικούς δρομολογητές που εκμεταλλεύονται την WDM τεχνική. Στο μέλλον όμως το κόστος προβλέπεται ότι θα μειωθεί σημαντικά.

1.2 Πλήρως οπτικά δίκτυα

Τα δίκτυα στα οποία όλες οι συνδέσεις μεταξύ των τερματικών κόμβων γίνονται με οπτικές ίνες ονομάζονται *πλήρως οπτικά δίκτυα* Σε ένα τέτοιο δίκτυο η πολυπλεξία μεταξύ δύο κόμβων γίνεται με την *πολυπλεξία διαίρεσης μήκους κύματος* (WDM) που είναι μια παραλλαγή της διαδικασίας διαίρεσης συχνότητας. Τα διαφορετικά σήματα μεταδίδονται μέσα στην ίνα ως φωτεινά σήματα με διαφορετικό μήκος κύματος. Μπορούμε να φανταστούμε τα διαφορετικά μήκη κύματος σαν διαφορετικά χρώματα και γι αυτό και οι οπτικές ίνες ονομάζονται και πολυχρωματικές ίνες. Για να έχουμε αμφίδρομη σύνδεση χρησιμοποιούνται δύο ίνες μία ανά κατεύθυνση ή για κάθε κατεύθυνση χρησιμοποιούνται τα μισά χρώματα της ίνας.

Το πρόβλημα της WDM είναι η δρομολόγηση. Αν χρησιμοποιήσουμε μεταγωγή πακέτου τότε ο ρυθμός μετάδοσης θα μειωνόταν σημαντικά λόγω των οπτικών μετατροπέων. Γι αυτό το λόγο χρησιμοποιούνται οι *παθητικοί οπτικοί μεταγωγείς.* Αυτοί βασίζονται στην αρχή ότι διαφορετικές ακτίνες με διαφορετικό μήκος κύματος έχουν διαφορετική γωνία διάθλασης όταν διέρχονται από μέσα με διαφορετικό δείκτη διάθλασης.

Σχήμα 1.2: Αρχή οπτικού μεταγωγέα.

Με τους οπτικούς μετατροπείς μπορούμε να διαχωρίσουμε τα οπτικά σήματα που πολυπλέκονται σε μια οπτική ίνα και να δρομολογήσουμε καθένα από τα σήματα σε διαφορετικές κατευθύνσεις. Έτσι πετυχαίνουμε ταχύτερη δρομολόγηση αλλά και πιο αποδοτική αφού γίνεται με παθητικό τρόπο και κατά συνέπεια με μειωμένη πιθανότητα σφάλματος. Η μετάδοση λοιπόν γίνεται μέσω μιας ακτίνας η οποία διατηρεί το ίδιο μήκος κύματος από τον αποστολέα μέχρι τον παραλήπτη.

Αυτός ο τρόπος μετάδοσης έχει έναν περιορισμό. Αν έχουμε μια οπτική ίνα ανάμεσα σε δύο κόμβους τότε η επικοινωνία τους δεσμεύει ένα χρώμα της ίνας. Όμως ο αριθμός των χρωμάτων είναι περιορισμένος. Στη συνέχεια θα δούμε αλγόριθμους που αντιμετωπίζουν το παραπάνω πρόβλημα.

1.3 Γράφοι και δίκτυα

Ένας γράφος αποτελείται από κόμβους και ακμές. Μπορούμε λοιπόν να μοντελοποιήσουμε ένα τηλεπικοινωνιακό δίκτυο με ένα γράφο αντιστοιχίζοντας κάθε κόμβο του δικτύου σε έναν κόμβο του γραφήματος και κάθε σύνδεση μεταξύ δύο κόμβων του δικτύου σε μία ακμή. Απαραίτητη

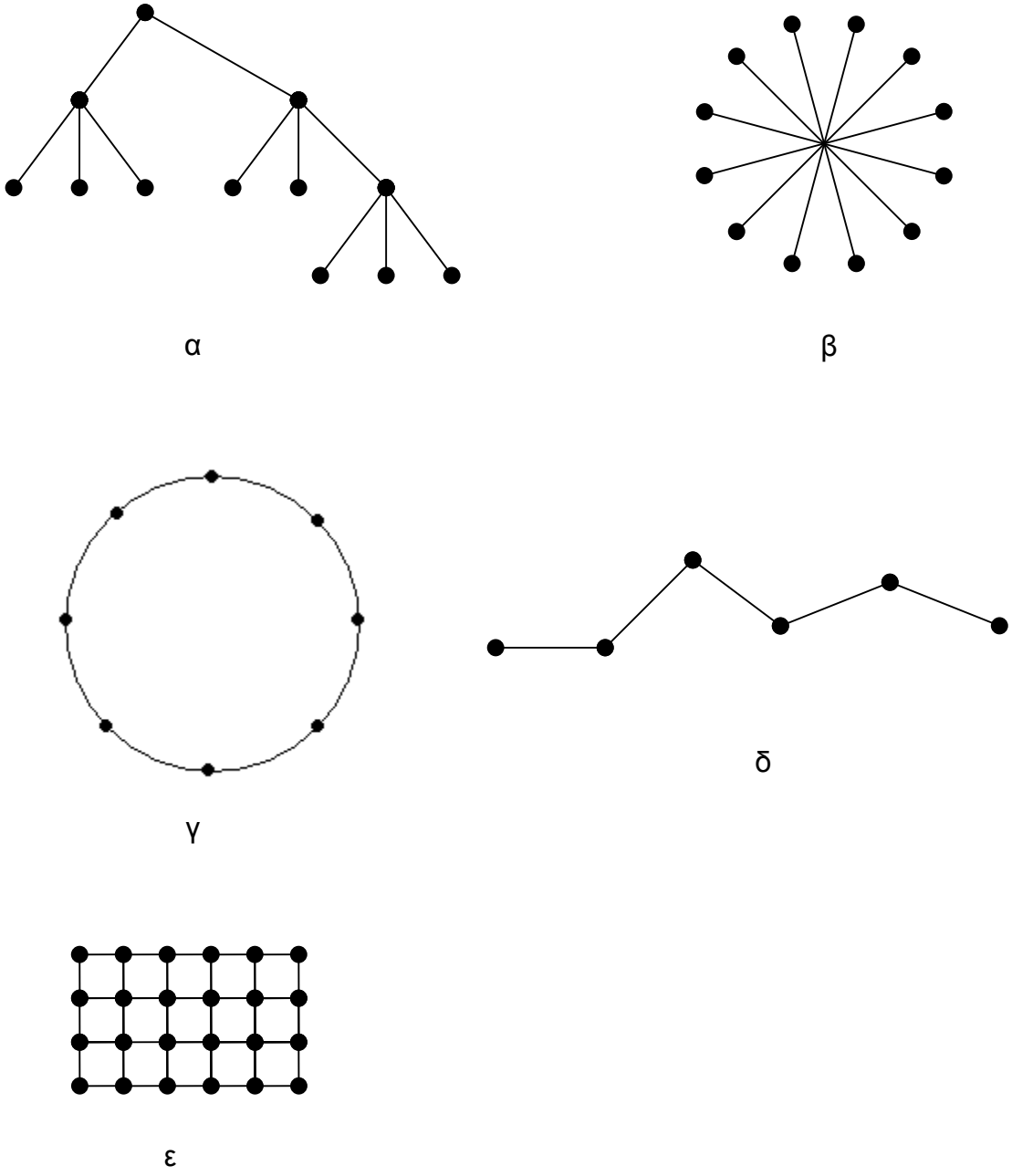
προϋπόθεση για να έχει νόημα η επικοινωνία από ένα κόμβο σε έναν άλλο είναι ο γράφος να είναι συνεκτικός.

Όταν δύο κόμβοι θέλουν να επικοινωνήσουν μπορεί να μας δίνεται από την αρχή το μονοπάτι που πρέπει να ακολουθήσουμε για να γίνει η επικοινωνία ή μπορεί να μας δίνονται μόνο ο αρχικός και ο τελικός κόμβος οπότε πρέπει να βρεθεί το μονοπάτι που θα ακολουθήσει το σήμα, πρέπει δηλαδή να γίνει και δρομολόγηση. Αν έχουμε ακυκλικό γράφο π.χ. δένδρο ή αλυσίδα τότε υπάρχει μόνο ένα μονοπάτι που συνδέει δύο κόμβους οπότε μια αίτηση σύνδεσης μεταξύ δύο κόμβων του δικτύου περιγράφεται από ένα μονοπάτι που παράγεται σε πολυωνυμικό χρόνο. Αν ο γράφος είναι κυκλικός τότε μπορεί να υπάρχουν περισσότερα από ένα μονοπάτια που συνδέουν δύο κόμβους οπότε πρέπει να δρομολογήσουμε τις αιτήσεις σύνδεσης επιλέγοντας ένα από τα μονοπάτια. Όταν ένα μονοπάτι περιγράφει την επικοινωνία μεταξύ δύο κόμβων τότε αυτό παίρνει ένα χρώμα. Όμως κανένα άλλο μήνυμα δεν μπορεί να περάσει από μια οπτική ίνα χρησιμοποιώντας το ίδιο χρώμα, οπότε αν δύο μονοπάτια χρησιμοποιούν μια ίδια ακμή τότε πρέπει να χρωματιστούν με διαφορετικά χρώματα. Στο [14] παρουσιάζονται τρόποι δρομολόγησης σε πλήρως οπτικά δίκτυα.

Αν έχουμε μονόδρομη επικοινωνία μεταξύ δύο κόμβων τότε μπορούμε να χρησιμοποιήσουμε διπλά κατευθυνόμενο γράφο με κατευθυνόμενα μονοπάτια και η επικάλυψή τους είναι διαφορετική από αυτή σε αμφίδρομη επικοινωνία

Στο σχήμα1.3 παρουσιάζονται διάφορες τοπολογίες γράφων που μπορούν να μοντελοποιήσουν τηλεπικοινωνιακά δίκτυα. Από αυτά η αλυσίδα και ο δακτύλιος θα είναι οι κύριες τοπολογίες για τις οποίες θα περιγράψουμε αλγόριθμους χρωματισμού μονοπατιών.

Πρόσφατα ερευνητές έχουν αρχίσει να ερευνούν οπτικά δίκτυα με πολλαπλές παράλληλες ίνες που συνδέουν γειτονικούς κόμβους [19]. Σε αυτά τα δίκτυα, δύο μονοπάτια που συνδέουν δύο κόμβους μπορούν να χρησιμοποιήσουν το ίδιο μήκος κύματος αν μεταφέρονται από διαφορετικές ίνες. Σε κάθε κόμβο, το σήμα που φτάνει σε οποιαδήποτε ίνα μιας εισερχόμενης σύνδεσης μπορεί να προωθηθεί σε μια τυχαία ίνα μιας εξερχόμενης σύνδεσης, χωρίς όμως να αλλάζει το μήκος κύματος. Ένα πλήρως οπτικό δίκτυο με πολλαπλές ίνες μπορεί να μοντελοποιηθεί σαν ένας γράφος $G=(V,E)$ με μια συνάρτηση $\mu:E\rightarrow N$, όπου $\mu(e)$ είναι ο αριθμός των ινών στη σύνδεση $e\in E$.



Σχήμα 1.3 Τοπολογίες γράφων: α) δένδρο, β) αστέρας γ) δακτύλιος δ) αλυσίδα ε) πλέγμα

1.4 Κλασσικά προβλήματα χρωματισμού σε γράφους

Το πρόβλημα *χρωματισμού κόμβων (graph coloring GC)* ορίζεται ως εξής: έχουμε ένα γράφο G(V,E) και θέλουμε να βρούμε τον ελάχιστο αριθμό χρωμάτων ώστε να χρωματιστούν όλοι οι κόμβοι του γράφου χωρίς δύο γειτονικοί κόμβοι να έχουν ίδια χρώματα. Ο αριθμός αυτός ονομάζεται *χρωματικός αριθμός* χ(G) του G και ένα κάτω όριό του είναι η *μέγιστη κλίκα* του γράφου. Στο αντίστοιχο πρόβλημα απόφασης ρωτάμε αν υπάρχει κ-χρωματισμός των κόμβων ώστε γειτονικοί κόμβοι να έχουν διαφορετικά χρώματα τέτοιος ώστε κ<=c.Το πρόβλημα απόφασης είναι NP-πλήρες [1], ενώ το πρόβλημα βελτιστοποίησης είναι μη προσεγγίσιμο [2].

Graph Coloring (GC)

Είσοδος: Ένας γράφος G(V,E).
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του V ώστε γειτονικοί κόμβοι να έχουν διαφορετικά χρώματα.
Στόχος: Εύρεση χ(G) = min k

Decision Graph Coloring (DecisionGC)

Είσοδος: Ένας γράφος G(V,E), αριθμός χρωμάτων c>0.
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του V ώστε γειτονικοί κόμβοι να έχουν διαφορετικά χρώματα.
Ερώτηση: Υπάρχει κ-χρωματισμός τέτοιος ώστε κ≤c;

Το *πρόβλημα χρωματισμού ακμών (edge coloring EC)* ορίζεται ως εξής: έχουμε ένα γράφο G(V,E) και θέλουμε να βρούμε τον ελάχιστο αριθμό χρωμάτων ώστε να χρωματιστούν όλες οι ακμές του γράφου χωρίς να υπάρχουν ακμές με το ίδιο χρώμα που να προσπίπτουν στον ίδιο κόμβο. Ο αριθμός αυτός ονομάζεται *χρωματικός δείκτης* χ'(G) του G και ένα κάτω όριό του είναι ο *μέγιστος βαθμός* του γράφου Δ(G). Αποδεικνύεται ότι ισχύει το θεώρημα:

Θεώρημα Vizing. Για κάθε απλό γράφο G ισχύει Δ(G) ≤ χ'(G)≤ Δ(G)+1.
Δηλαδή αρκούν Δ(G)+1 χρώματα για να χρωματιστούν όλες οι ακμές του γράφου.
Επίσης αποδεικνύεται ότι σε πολυγράφο (multigraph) ισχύει Δ(G) ≤ χ'(G)≤ Δ(G)+μ (μ η μέγιστη πολλαπλότητα ακμής)

Στο αντίστοιχο πρόβλημα απόφασης ρωτάμε αν υπάρχει κ-χρωματισμός των ακμών ώστε ακμές που προσπίπτουν στον ίδιο κόμβο να έχουν διαφορετικά χρώματα έτσι ώστε χ'(G)=Δ(G) ή χ'(G)=Δ(G)+1. Το πρόβλημα απόφασης είναι NP-πλήρες [3], ενώ από το θεώρημα Vizing

προκύπτει ότι για το πρόβλημα βελτιστοποίησης υπάρχει προσεγγιστικός αλγόριθμος με λόγω προσέγγισης 1+1/Δ(G), δηλαδή ασυμπτωτικά βέλτιστος.

Edge Coloring (EC)

Είσοδος: Ένας γράφος G(V,E).
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του E ώστε ακμές που προσπίπτουν στον ίδιο κόμβο να έχουν διαφορετικά χρώματα.
Στόχος: Εύρεση χ'(G) = min k

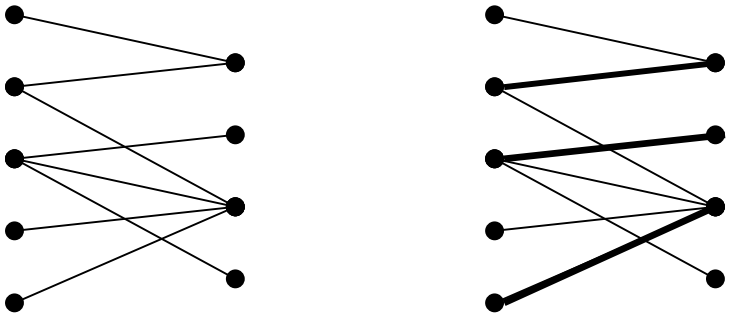
Decision Edge Coloring (DecisionEC)

Είσοδος: Ένας γράφος G(V,E).
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του E ώστε ακμές που προσπίπτουν στον ίδιο κόμβο να έχουν διαφορετικά χρώματα.
Ερώτηση: χ'(G)=Δ(G) ή χ'(G)=Δ(G)+1.

1.5 Ταιριάσματα

Δοθέντος ενός μη κατευθυνόμενου γράφου G(*V,E*), ένα *ταίριασμα (matching)* είναι ένα υποσύνολο ακμών M $\subseteq$ E τέτοιο ώστε για κάθε κόμβο υ$\in$V, το πολύ μια ακμή του M προσπίπτει στο υ. Ο κόμβος υ$\in$V είναι ταιριασμένος από το ταίριασμα M αν κάποια ακμή του M προσπίπτει στον υ, αλλιώς δεν είναι ταιριασμένος. Ένα *μέγιστο ταίριασμα (maximum matching)* είναι ένα ταίριασμα με μέγιστη πληθικότητα. Ένα ταίριασμα σε διμερές γράφο ονομάζεται *διμερές ταίριασμα (bipartite matching).* Στους αλγόριθμους που θα παρουσιαστούν στη συνέχεια της εργασίας το *μέγιστο ταίριασμα* και το *μέγιστο διμερές ταίριασμα* θα χρησιμοποιηθούν σαν μέρος της υλοποίησης των αλγορίθμων..

Διμερή ταιριάσματα μπορούν να υπολογιστούν σε χρόνο O(VE) [20], ενώ οι Micali-Vazirani παρουσιάζουν στο [17] έναν $O(n^{2.5})$ αλγόριθμο για τον υπολογισμό μέγιστου ταιριάσματος σε γενικό γράφο. Στο σχήμα 1.4 φαίνεται ένας διμερής γράφος και το μέγιστο διμερές ταίριασμά του.



Σχήμα 1.4 : α) ένας διμερής γράφος G(V,E), β) το μέγιστο διμερές ταίριασμά του με πληθικότητα 3.

Κεφάλαιο 2

Χρωματισμός μονοπατιών

Το πρόβλημα *χρωματισμού μονοπατιών (path coloring PC)* ορίζεται ως εξής: Δοθέντος ενός γράφου G(V,E) και ενός συνόλου P από μονοπάτια στον G ζητάμε το ελάχιστο κ για έναν κ-χρωματισμό του P ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικά χρώματα. Δηλαδή μονοπάτια που έχουν μια κοινή ακμή, να μην έχουν το ίδιο χρώμα. Το PC μπορεί να αναχθεί στο GC.
Το πρόβλημα απόφασης (DecisionPC) ορίζεται ως εξής: Δοθέντος ενός γράφου G(V,E), ενός συνόλου P από μονοπάτια στον G και ενός αριθμού χρωμάτων $c>0$ ρωτάμε αν υπάρχει κ-χρωματισμός του P τέτοιος ώστε $\kappa \leq c$ και επικαλυπτόμενα μονοπάτια να έχουν διαφορετικά χρώματα.

Το DecisionPC ανήκει στο NP αφού μπορεί κάποιος να επιβεβαιώσει μια λύση που έχει πληθικότητα μικρότερη από c σε πολυωνυμικό χρόνο. Το GC αποδεικνύεται ότι μπορεί να αναχθεί στο PC οπότε το DecisionPC είναι NP-πλήρες [4]. Η αναγωγή αυτή διατηρεί την μη προσεγγισιμότητα και κατά συνέπεια το PC είναι μη προσεγγίσιμο με παράγοντα n^{δ} για κάποιο $\delta>0$.

Ένα κάτω όριο στη λύση του PC είναι το φορτίο του γράφου, δηλαδή το μέγιστο φορτίο ακμής στο γράφο, αφού για να χρωματιστούν όλα τα μονοπάτια που περνούν από την ακμή με το μέγιστο φορτίο, με διαφορετικά χρώματα, απαιτούνται τουλάχιστον τόσα χρώματα όσα το μέγιστο φορτίο.

Path Coloring (PC)

Είσοδος: Ένας γράφος G(V,E), σύνολο P από μονοπάτια στον G.
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του P ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικά χρώματα.
Στόχος: Εύρεση OPT-χρωματισμού τέτοιος ώστε OPT= min κ

Decision Path Coloring (DecisionPC)

Είσοδος: Ένας γράφος G(V,E), σύνολο P από μονοπάτια στον G, αριθμός χρωμάτων $c>0$.
Αποδεκτή λύση: Ένας κ-χρωματισμός των στοιχείων του P ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικά χρώματα.
Ερώτηση: Υπάρχει κ-χρωματισμός τέτοιος ώστε $\kappa \leq c$;

2.1 MaxPC και MaxRPC

Τα προβλήματα για τα οποία υλοποιούνται αλγόριθμοι στη συνέχεια της εργασίας είναι τα MaxPC και MaxRPC.

Το πρόβλημα της *εύρεσης μέγιστου χρωματισμού μονοπατιών (MaxPC)* είναι το εξής: Δοθέντος ενός γράφου G(V,E), ενός συνόλου

μονοπατιών P και ενός αριθμού χρωμάτων w, ζητάμε μια λύση που θα χρωματίζει τον μέγιστο δυνατό αριθμό μονοπατιών ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα.

Maximum Path Coloring Problem (MaxPC)

Είσοδος: Ένας γράφος G(V,E), ένα σύνολο από μονοπάτια P και ένας αριθμός από διαθέσιμα χρώματα w.
Αποδεκτή λύση*:* Χρωματισμός ενός υποσυνόλου P'$\subseteq$ P έτσι ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα.
Στόχος: Μια λύση που χρωματίζει τον μέγιστο αριθμό μονοπατιών.

Στο παραπάνω πρόβλημα τα μονοπάτια είναι δεδομένα και κατά συνέπεια δε χρειάζεται να γίνει δρομολόγηση. Αν η δρομολόγηση δεν δίνεται τότε έχουμε το πρόβλημα *της δρομολόγησης και εύρεσης μέγιστου χρωματισμού μονοπατιών (MaxRPC)* που είναι το εξής: Δοθέντος ενός γράφου G(V,E), ενός συνόλου ζευγαριών κόμβων (αιτήσεων σύνδεσης) και ενός αριθμού χρωμάτων w, ζητάμε μια αντιστοίχηση μονοπατιών σε ένα υποσύνολο αιτήσεων (δρομολόγηση) και ένα χρωματισμό αυτών των μονοπατιών, ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα και επιπλέον να ικανοποιείται ο μέγιστος δυνατός αριθμός αιτήσεων.

Maximum Routing and Path Coloring Problem (MaxRPC)

Είσοδος: Ένας γράφος G(V,E), ένα σύνολο από ζευγάρια μονοπατιών (αιτήσεις σύνδεσης) και ένας αριθμός από διαθέσιμα χρώματα w.
Αποδεκτή λύση*:* Μια αντιστοίχηση μονοπατιών σε ένα υποσύνολο αιτήσεων (δρομολόγηση) και ένα χρωματισμό αυτών των μονοπατιών, ώστε επικαλυπτόμενα μονοπάτια να έχουν διαφορετικό χρώμα
Στόχος: Μια λύση που χρωματίζει τον μέγιστο αριθμό μονοπατιών δηλαδή ικανοποιεί τον μέγιστο αριθμό αιτήσεων..

2.2 Χρήση του MaxPC και του MaxRPC σε διάφορες εφαρμογές

Τα προβλήματα MaxPC και MaxRPC μπορούν να χρησιμοποιηθούν σαν ρουτίνες σε διάφορες εφαρμογές, όπως σε οπτικά δίκτυα ή για τον προγραμματισμό εργασιών (scheduling).

Όπως έχει αναφερθεί στο κεφάλαιο 1 ένα οπτικό δίκτυο μπορεί να παρασταθεί με ένα γράφο, όπου οι κόμβοι του δικτύου αντιστοιχούν στους κόμβους του γράφου, η διαδρομή που ακολουθεί το σήμα στα μονοπάτια του γράφου και τα διαθέσιμα μήκη κύματος στα διαθέσιμα χρώματα. Το πρόβλημα που προκύπτει είναι πώς να ικανοποιήσουμε όσο το δυνατόν περισσότερες αιτήσεις επικοινωνίας στο δίκτυο. Μια προσέγγιση στο πρόβλημα μπορεί να είναι η εξής: Τρέχω μια φορά το MaxPC και παίρνω μια λύση για το πρόβλημα. Αφού ικανοποιηθούν οι παραπάνω αιτήσεις

ξανατρέχω το MaxPC στις υπόλοιπες αιτήσεις με σκοπό να βελτιώσω την αρχική λύση. Μια άλλη προσέγγιση μπορεί να είναι η ακόλουθη: Τρέχω το MaxPC για τις μακρινές αιτήσεις και στη συνέχεια ξανατρέχω για τις κοντινές αιτήσεις. Η τελική λύση προκύπτει από τον συνδυασμό των δύο λύσεων.

Μια άλλη εφαρμογή όπου μπορούν να χρησιμοποιηθούν τα MaxPC και MaxRPC είναι ο προγραμματισμός εργασιών (scheduling). Ένα γραμμικό scheduling μπορεί να παρασταθεί από μια αλυσίδα όπου τα μονοπάτια της αλυσίδας αντιστοιχούν στις εργασίες που πρέπει να γίνουν και το μήκος του κάθε μονοπατιού αντιστοιχεί στη χρονική διάρκεια της κάθε εργασίας. Τα διαθέσιμα χρώματα αντιστοιχούν στους διαθέσιμους επεξεργαστές. Για το συγκεκριμένο πρόβλημα έχω από την προηγούμενη παράγραφο πολυωνυμικό αλγόριθμο που λύνει βέλτιστα το πρόβλημα. Ένα περιοδικό scheduling μπορεί να παρασταθεί από ένα δακτύλιο. Τα μονοπάτια και πάλι αντιστοιχούν στις εργασίες και το μήκος τους στη χρονική διάρκεια της κάθε εργασίας. Τα χρώματα αντιστοιχούν στους διαθέσιμους επεξεργαστές. Χρησιμοποιώντας τους αλγόριθμους για το MaxPC και MaxRPC που θα δούμε στο επόμενο κεφάλαιο μπορούμε να έχουμε προσεγγιστική λύση στο πρόβλημα με παράγοντα προσέγγισης 2/3.

2.3 Επισκόπηση γνωστών αποτελεσμάτων για το PC-MaxPC και το RPC-MaxRPC

Το MaxPC και το MaxRPC σε αλυσίδες λύνονται βέλτιστα σε πολυωνυμικό χρόνο. Στο [5] παρουσιάζεται ένας γραμμικός αλγόριθμος από τους Carlisle και Lloyd. Στο [6] δίνεται ένας πιο απλός άπληστος αλγόριθμος που δίνει επίσης βέλτιστη λύση σε πολυωνυμικό χρόνο. Ο αλγόριθμος αυτός χρησιμοποιείται σαν υπορουτίνα στην υλοποίηση των αλγορίθμων στη συνέχεια της εργασίας.

Συνοπτικά ο αλγόριθμος είναι ο εξής: Η αλυσίδα διατρέχεται ακμή προς ακμή ξεκινώντας από αριστερά. Αν σε κάποια ακμή επικαλύπτονται περισσότερα από w μονοπάτια τότε διαγράφονται τα μονοπάτια που εκτείνονται περισσότερο προς τα δεξιά. Αυτός ο αλγόριθμος είναι για μη κατευθυνόμενο γράφο, με μικρές αλλαγές όμως μπορεί να χρησιμοποιηθεί και για κατευθυνόμενο.

Συμπέρασμα: Το MaxPC (και κατά συνέπεια και το MaxRPC) σε κατευθυνόμενη και σε μη κατευθυνόμενη μορφή λύνονται βέλτιστα σε πολυωνυμικό χρόνο αν ο γράφος είναι αλυσίδα.

Το PC σε δακτύλιο αποδεικνύεται NP-hard από τους Garey, Jonhnson και Miller στο [9]. Ο Karapetian στο [10] παρουσιάζει έναν 3/2-προσεγγιστικό αλγόριθμο για το PC σε δακτύλιο.

Το MaxPC σε δακτυλίους αποδεικνύεται NP-hard από τους Nomikos, Pagourtzis, Zachos στο [22]. Στην ίδια εργασία παρουσιάζεται 2/3-προσεγγιστικός αλγόριθμος για κατευθυνόμενο δακτύλιο. Με βάση αποτέλεσμα των Hsu και Tsai [23] οι Wan και Liu [7] παρουσιάζουν έναν (1-1/e)-προσεγγιστικό αλγόριθμο για το MaxPC σε δακτύλιο. Ο Kumar στο [11]

παρουσίασε έναν πιθανοτικό αλγόριθμο που λύνει το MaxPC σε δακτύλιο, με μεγάλη πιθανότητα, με παράγοντα προσέγγισης $\approx 1/1.36$.

Το RPC μπορεί να λυθεί σε πολυωνυμικό χρόνο σε αλυσίδα και σε δένδρα φραγμένου βαθμού ενώ σε δακτύλιο είναι NP-hard [4]. Στο [14] παρουσιάζεται ένας 2-προσεγγιστικός αλγόριθμος για το RPC σε μη κατευθυνόμενο γράφο και στο [15] για κατευθυνόμενο.

Το MaxRPC σε δακτύλιο είναι NP-hard [12,13]. Για το MaxRPC οι Wan και Liu στο [7] παρουσιάζουν έναν (1-1/e)-προσεγγιστικό αλγόριθμο (e η βάση του φυσικού λογάριθμου) για δακτυλίους και δένδρα και έναν προσεγγιστικό αλγόριθμο σταθερού συντελεστή (constant ratio) για meshes. Οι Erlebach και Jansen στο [8] παρουσιάζουν έναν (1-1/e)-προσεγγιστικό αλγόριθμο για το κατευθυνόμενο MaxRPC σε δένδρα φραγμένου βαθμού (bounded degree trees) και έναν 0.451-προσεγγιστικό αλγόριθμο για δένδρα. Στο [22] παρουσιάζεται ένας 7/11-προσεγγιστικός αλγόριθμος για κατευθυνόμενο δακτύλιο.

2.4 Προσέγγιση της εργασίας στα MaxPC και MaxRPC

Στην εργασία αυτή υλοποιήθηκαν 2/3 προσεγγιστικοί αλγόριθμοι για το MaxPC και για το MaxRPC, δηλαδή αλγόριθμοι που πετυχαίνουν πολύ καλύτερη προσέγγιση από τους μέχρι τώρα γνωστούς αλγόριθμους όπως είδαμε στην προηγούμενη ενότητα.

Για το κάθε πρόβλημα υλοποιήθηκαν τρεις αλγόριθμοι. Ο πρώτος αλγόριθμος για το MaxPC πάρθηκε από το [21]. Στη συνέχεια υλοποιήθηκε ένας άλλος αλγόριθμος που αποτελεί σημαντική βελτίωση του προηγούμενου αλγόριθμου και βρίσκεται στο [22]. Τέλος υλοποιήθηκε ένας τρίτος αλγόριθμος παρόμοιος με τον προηγούμενο που όμως είναι πιο απλός προγραμματιστικά αφού χρησιμοποιεί πιο απλές δομές δεδομένων και πιο απλό κώδικα. Πρακτικά ο συγκεκριμένος αλγόριθμος πετυχαίνει και καλύτερα αποτελέσματα σε αρκετά στιγμιότυπα, οπότε αποτελεί μια ακόμα βελτίωση.

Για το MaxRPC επίσης υλοποιήθηκαν τρεις αλγόριθμοι. Ο πρώτος βρίσκεται και αυτός στο [21]. Ο δεύτερος αλγόριθμος υλοποιήθηκε στα πλαίσια αυτής της διπλωματικής εργασίας και χρησιμοποιεί ιδέες από τον αλγόριθμο για το MaxPC στο [22]. Ο τρίτος αλγόριθμος υλοποιήθηκε επίσης στα πλαίσια αυτής της εργασίας και αποτελεί βελτίωση του δεύτερου αλγόριθμου.

Κεφάλαιο 3

Προσεγγιστικοί αλγόριθμοι για MaxPC και MaxRPC

Οι προσεγγιστικοί αλγόριθμοι που ακολουθούν παρουσιάζονται στις εργασίες [21], [22] του Χρήστου Νομικού, Άρη Παγουρτζή και Στάθη Ζάχου.

3.1 Ένας 2/3-προσεγγιστικός αλγόριθμος για το MaxPC

ALG για το MaxPC:

Είσοδος: ένας δακτύλιος G, ένα σύνολο μονοπατιών P και ένας αριθμός χρωμάτων w.
Έξοδος: ένας χρωματισμός ενός υποσυνόλου του P.

ALG:
Διάλεξε μια τυχαία ακμή e του G.
Διαμέρισε το P σε ένα σύνολο Q που περιέχει μονοπάτια που περνούν από την e και σε ένα σύνολο C που περιέχει τα υπόλοιπα μονοπάτια (δύο μονοπάτια από το Q δεν μπορούν να χρωματιστούν με το ίδιο χρώμα, τα μονοπάτια στο C είναι σε αλυσίδα).
Εκτέλεσε τους αλγόριθμους ALG_1 και ALG_2 ανεξάρτητα.
Βγάλε σαν έξοδο τη λύση με τη μεγαλύτερη πληθικότητα.

ALG_1: Χρωμάτισε τον μέγιστο αριθμό μονοπατιών στο C χρησιμοποιώντας έναν αλγόριθμο για αλυσίδες (βλ. συμπέρασμα στην παράγραφο 2.2).
Αν όλα τα μονοπάτια στο C χρωματίστηκαν χρησιμοποίησε τα ελεύθερα χρώματα (ένα για κάθε μονοπάτι) για το χρωματισμό των μονοπατιών του Q.
ALG_2: Βρες ένα μέγιστο ταίριασμα (πληθικότητας μ) στο διμερή γράφο $H=(Q \cup C,E)$ όπου $(q,c) \in E$ αν και μόνο αν $q \in Q$ και $c \in C$ δεν επικαλύπτονται. Χρωμάτισε κάθε ζευγάρι του ταιριάσματος χρησιμοποιώντας ένα χρώμα για κάθε ζευγάρι, μέχρις ότου να μην υπάρχουν άλλα ζευγάρια ή να μην υπάρχουν άλλα χρώματα.

Πρόταση: Ο παραπάνω αλγόριθμος χρωματίζει τουλάχιστον 2/3 του μέγιστου αριθμού των μονοπατιών που είναι δυνατό να χρωματιστούν.

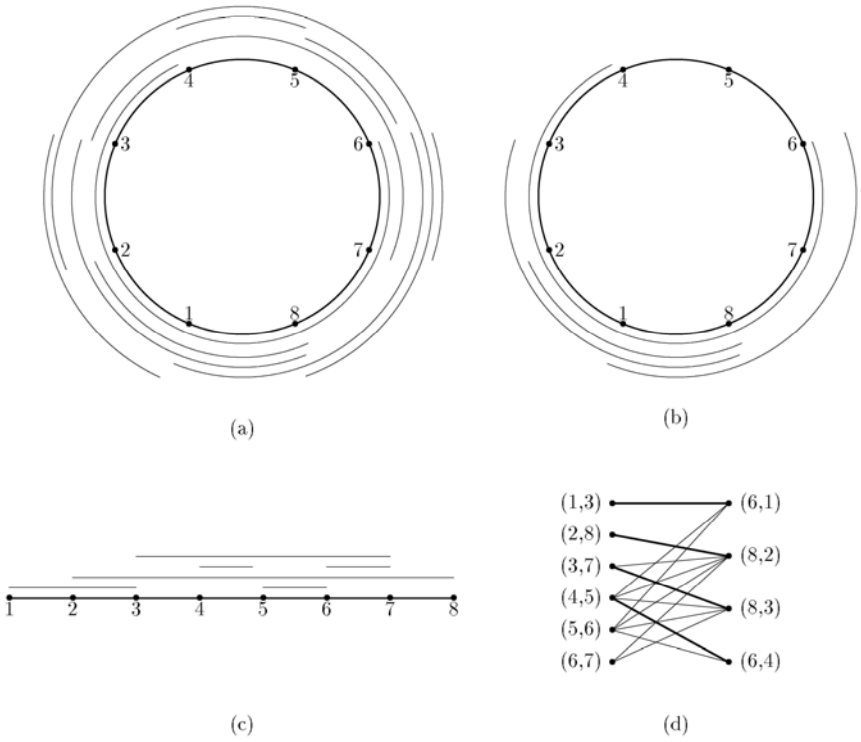
Απόδειξη: Έστω SOL_0 ο αριθμός των μονοπατιών στο C που χρωματίζονται από τον αλγόριθμο της αλυσίδας, $SOL_1(SOL_2)$ ο αριθμός των μονοπατιών που χρωματίζονται από τον $ALG_1(ALG_2)$ και SOL ο αριθμός των μονοπατιών που χρωματίζονται από τον ALG. Έστω επίσης OPT ο μέγιστος αριθμός μονοπατιών που είναι δυνατό να χρωματιστούν από έναν οποιοδήποτε αλγόριθμο που έχει διαθέσιμα w χρώματα.

Αν όλα τα μονοπάτια χρωματίστηκαν τότε προφανώς SOL=OPT οπότε χωρίς βλάβη της γενικότητας υποθέτουμε ότι ούτε ο ALG_1 ούτε ο ALG_2 χρωματίζουν όλα τα μονοπάτια. Τότε ισχύουν τα παρακάτω:

1. $SOL_0 \leq SOL_1$
2. $SOL = \max(SOL_1, SOL_2)$ οπότε $SOL \geq SOL_i$, $0 \leq i \leq 2$.
3. $SOL_1 \geq w$: αν ένα χρώμα παραμείνει αχρησιμοποίητο τότε όλα τα μονοπάτια στο C χρωματίστηκαν από τον αλγόριθμο για αλυσίδες και όλα τα μονοπάτια στο Q επίσης χρωματίστηκαν αλλά από την υπόθεση έχουμε ότι δεν χρωματίστηκαν όλα τα χρώματα.
4. $OPT \leq SOL_0 + w$: κάθε αλγόριθμος μπορεί να χρωματίσει το πολύ SOL_0 μονοπάτια στο C και το πολύ w μονοπάτια στο Q.
5. $OPT \leq SOL_1 + w$ (από 1, 4)
6. $SOL_1 \geq 1/2 OPT$ (από 3, 5)
7. $OPT \leq SOL_1 + \mu$: σε σχέση με τη SOL_1 μια βέλτιστη λύση μπορεί μόνο να περιέχει επιπλέον χρωματισμένα μονοπάτια από το Q ώστε κάθε μονοπάτι να έχει ίδιο χρώμα με κάποιο από το C. Τέτοια μονοπάτια αντιστοιχούν σε γειτονικές κορυφές στο διμερή γράφο H που κατασκευάστηκε από τον ALG_2. Οπότε ο αριθμός αυτών των μονοπατιών είναι το πολύ μ.
8. $OPT \leq SOL_1 + \min(\mu, w)$: (από 5, 7)
9. $SOL_2 \geq 2\min(\mu, w)$: ο ALG_2 χρωματίζει ζευγάρια του ταιριάσματος έως ότου δεν υπάρχουν άλλα ζευγάρια ή άλλα χρώματα.
10. $SOL \geq 2/3$ OPT (από 2, 8, 9)

Παρατήρηση: από το (6) προκύπτει ότι ο ALG_1 χρωματίζει τουλάχιστον 1/2 OPT μονοπάτια.

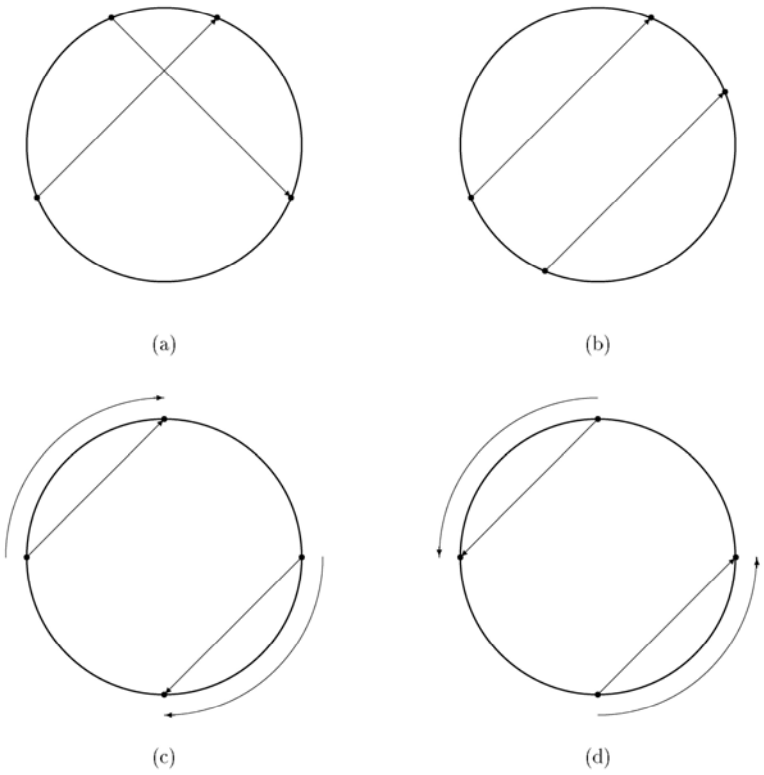
Πολυπλοκότητα: Η διαμέριση του P απαιτεί O(m) χρόνο (m ο αριθμός των μονοπατιών). Ο χρωματισμός των μονοπατιών στο C μπορεί να εκτελεστεί σε χρόνο O(m) χρησιμοποιώντας τον αλγόριθμο των Carlisle και Lloyd [5] δεδομένου ότι τα μονοπάτια είναι ταξινομημένα κατά το σημείο τερματισμού τους. Αυτή η ταξινόμηση μπορεί να γίνει σε O(m+n) (n ο αριθμός των κορυφών) χρόνο με bucket sort. Ο χρωματισμός των μονοπατιών στο Q μπορεί να εκτελεστεί σε O(m) χρόνο. Το πιο αργό βήμα του ALG είναι ο υπολογισμός του ταιριάσματος στον ALG_2. Αποδεικνύεται ότι το H είναι chordal GGG διμερής γράφος οπότε το ταίριασμα M μπορεί να υπολογιστεί σε χρόνο $O(m^2)$ χρησιμοποιώντας τον αλγόριθμο στο [16]. Επιπλέον η ειδική μορφή του γράφου H επιτρέπει τον υπολογισμό του M σε O((n+m)*log min (l,n)) (l το μέγιστο φορτίο), χρησιμοποιώντας έναν άπληστο αλγόριθμο. Το μέρος χρωματισμού του ALG_2 και η επιλογή της λύσης με τη μέγιστη πληθικότητα μπορεί να υπολογιστεί σε χρόνο O(m). Συνεπώς η πολυπλοκότητα του ALG για το MaxPC είναι O((n+m)*log min(m,n)).



Σχήμα 3.1 a) Ένα στιγμιότυπο του MaxPC b) Το σύνολο μονοπατιών Q όπου όλα τα μονοπάτια περνούν από την ακμή {8,1} c) Το σύνολο C του οποίου τα μονοπάτια είναι σε αλυσίδα d) Ένα μέγιστο ταίριασμα μεταξύ του C και Q.

3.2 Ένας 2/3-προσεγγιστικός αλγόριθμος για το MaxRPC

Ο αλγόριθμος που ακολουθεί χρησιμοποιεί παρόμοιες ιδέες με αυτόν για το MaxPC με τις ανάλογες αλλαγές. Κάθε αίτηση σύνδεσης αντιστοιχεί σε μια χορδή του δακτυλίου. Δύο αιτήσεις των οποίων ο χορδές τέμνονται δεν μπορούν να δρομολογηθούν χωρίς τα μονοπάτια τους να επικαλύπτονται. Δύο αιτήσεις θα ονομάζονται *συμβατές* όταν τα μονοπάτια τους δεν επικαλύπτονται (σχήμα 3.2).



Σχήμα 3.2 a,b) μη συμβατές αιτήσεις c) δεξιόστροφα συμβατές αιτήσεις
d) αριστερόστροφα συμβατές αιτήσεις.

ALG για το MaxRPC:

Είσοδος: ένας δακτύλιος G, ένα σύνολο αιτήσεων R και ένας αριθμός χρωμάτων w.
Έξοδος: μονοπάτια και χρωματισμός ενός υποσυνόλου του R.

ALG:
Εκτέλεσε τους αλγόριθμους ALG_1 και ALG_2 ανεξάρτητα.
Βγάλε σαν έξοδο τη λύση με τη μεγαλύτερη πληθικότητα.

ALG_1: Διάλεξε μια τυχαία ακμή e του G. Δρομολόγησε τις αιτήσεις έτσι ώστε τα μονοπάτια να αποφεύγουν την e. Έστω C το σύνολο μονοπατιών που προκύπτει. Χρωμάτισε τον μέγιστο αριθμό μονοπατιών στο C χρησιμοποιώντας τον αλγόριθμο για αλυσίδες.
ALG_2: Βρες ένα μέγιστο ταίριασμα (πληθικότητας μ) στο γράφο H=(R,E) όπου R το σύνολο των αιτήσεων και το E περιέχει όλα τα ζευγάρια των συμβατών αιτήσεων. Δρομολόγησε τα ταιριασμένα ζευγάρια ώστε να μην επικαλύπτονται και χρωμάτισε τα δύο μονοπάτια με το ίδιο χρώμα, μέχρις ότου δεν υπάρχουν άλλα ζευγάρια ή άλλα χρώματα.

Θεώρημα: Ο παραπάνω αλγόριθμος ικανοποιεί τουλάχιστον 2/3 από τον μέγιστο αριθμό των αιτήσεων που είναι δυνατό να ικανοποιηθούν ταυτόχρονα.

Απόδειξη: Έστω SOL_1(SOL_2) ο αριθμός των αιτήσεων που ικανοποιούνται από τον ALG_1(ALG_2) και SOL ο αριθμός των αιτήσεων που ικανοποιούνται από τον ALG. Έστω επίσης OPT ο μέγιστος αριθμός αιτήσεων στο R που είναι δυνατό να ικανοποιηθούν από έναν οποιοδήποτε αλγόριθμο που έχει διαθέσιμα w χρώματα.

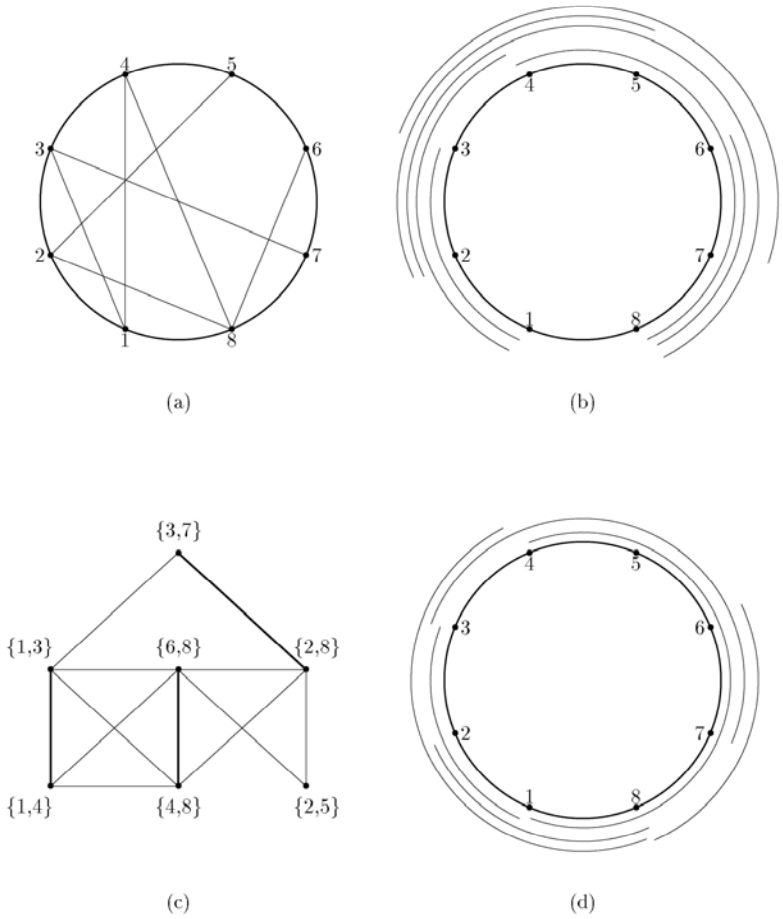
Χωρίς βλάβη της γενικότητας υποθέτουμε ότι ούτε ο ALG_1 ούτε ο ALG_2 ικανοποιούν όλες τις αιτήσεις διαφορετικά SOL=OPT. Τότε ισχύουν τα παρακάτω:

1. $SOL=\max(SOL_1,SOL_2)$.
2. $SOL_1 \geq w$: αν υπάρχει ένα αχρησιμοποίητο χρώμα τότε όλα τα μονοπάτια στο C έχουν χρωματιστεί από τον αλγόριθμο της αλυσίδας, αλλά από την υπόθεση έχουμε ότι δεν ικανοποιούνται όλες οι αιτήσεις.
3. $OPT \leq SOL_1+w$:
   Έστω A μια βέλτιστη λύση που ικανοποιεί OPT αιτήσεις. Στην A ο αριθμός των μονοπατιών που δρομολογούνται (και χρωματίζονται) αποφεύγοντας την e είναι το πολύ SOL_1, αφού ο αλγόριθμος της αλυσίδας είναι βέλτιστος. Επιπλέον, η A μπορεί να έχει το πολύ w μονοπάτια που δρομολογούνται μέσω της e, αφού αυτά τα μονοπάτια επικαλύπτονται.
4. $SOL_1 \geq 1/2OPT$: (από 2, 3).
5. $OPT \leq SOL_1+\mu$:
   Για την βέλτιστη λύση A με τις αιτήσεις που δρομολογούνται μέσω της e με ένα μοναδικό χρώμα υπάρχει μια άλλη βέλτιστη λύση B με αυτές τις αιτήσεις να δρομολογούνται αποφεύγοντας την e (με τον ίδιο χρωματισμό). Οπότε σε σύγκριση με την SOL_1, η βέλτιστη λύση μπορεί μόνο να περιέχει επιπλέον αιτήσεις που δρομολογούνται μέσω της e και χρωματίζονται έτσι ώστε κάθε αίτηση να έχει το ίδιο χρώμα με μια αίτηση που δρομολογείται αποφεύγοντας την e. Τέτοια ζευγάρια αιτήσεων αντιστοιχούν σε γειτονικές κορυφές στο γράφο H που δημιουργείται από τον ALG_2. Οπότε ο αριθμός αυτών των αιτήσεων είναι το πολύ μ.
6. $OPT \leq SOL_1+\min(\mu,w)$: (από 3, 5)
7. $SOL_2 \geq 2*\min(\mu,w)$:
   ο ALG_2 δρομολογεί και χρωματίζει τα ταιριασμένα ζευγάρια αιτήσεων έως ότου δεν υπάρχουν άλλα ζευγάρια ή άλλα χρώματα

8. SOL≥2/3OPT: (από 1, 6, 7)

Παρατήρηση: Όπως και για το MaxPC από το (4) συμπεραίνουμε ότι ο ALG_1 πετυχαίνει παράγοντα προσέγγισης 1/2 για το MaxRPC σε μη κατευθυνόμενο δακτύλιο.

Πολυπλοκότητα: Παρόμοια με τον αλγόριθμο για το MaxPC το μέρος του αλγόριθμου που απαιτεί τον περισσότερο χρόνο για το MaxRPC είναι ο υπολογισμός του ταιριάσματος στον ALG_2, που μπορεί να υπολογιστεί σε χρόνο $O(m^{2.5})$ χρησιμοποιώντας τον αλγόριθμο στο [17]. Τα υπόλοιπα βήματα απαιτούν χρόνο O(n+m) οπότε η συνολική πολυπλοκότητα είναι $O(n+m^{2.5})$.



Σχήμα 3.3 a) Ένα στιγμιότυπο του MaxRPC, οι αιτήσεις παρουσιάζονται σαν χορδές b) Το σύνολο μονοπατιών P_e που δρομολογούνται αποφεύγοντας την ακμή {8,1} c) Ένα μέγιστο ταίριασμα στο γράφο συμβατότητας αιτήσεων d) Μια δρομολόγηση και χρωματισμός από τον ALG_2 σύμφωνα με το μέγιστο ταίριασμα.

3.3 Ένας βελτιωμένος 2/3-προσεγγιστικός αλγόριθμος για το MaxPC

Παρακάτω ακολουθεί ένας απλός αλγόριθμος για το MaxPC (ALG_1) που πετυχαίνει παράγοντα προσέγγισης 1/2. Ο ALG_1 θα χρησιμοποιηθεί παρακάτω σαν υπορουτίνα για την δημιουργία του αλγόριθμου ALG_2 που είναι μια βελτίωση του αλγόριθμου που παρουσιάστηκε στην προηγούμενη ενότητα. Ο αλγόριθμος ALG_2 παρουσιάζεται στο [22].

ALG_1 για το MaxPC.

Είσοδος: Ένας δακτύλιος R_n, ένα σύνολο P από μονοπάτια και ένας αριθμός διαθέσιμων χρωμάτων w.
Έξοδος: Ένας χρωματισμός ενός υποσυνόλου του P με w χρώματα.

ALG_1:

1. Διαμέρισε το P σε ένα σύνολο P_e που περιέχει μονοπάτια που περνούν από την ακμή e={n,1} και σε ένα σύνολο P_c που περιέχει τα υπόλοιπα μονοπάτια (P_e={(i,j)$\in$P| i>j} and P_c=P-P_e).
2. Υπολόγισε μια βέλτιστη λύση του στιγμιότυπου (C_n, P_c, w) χρησιμοποιώντας έναν αλγόριθμο για αλυσίδα.
3. Αν μερικά χρώματα παραμένουν αχρησιμοποίητα χρησιμοποίησε τα για να χρωματίσεις τα μονοπάτια στο P_e (καθένα με ένα διαφορετικό χρώμα).

Σχόλια: Προφανώς ο ALG_1 δίνει διαφορετικά χρώματα σε επικαλυπτόμενα μονοπάτια. Τα μονοπάτια στο P_c είναι σε μια αλυσίδα C_n αφού αποφεύγουν την ακμή {n,1}. Ένα στιγμιότυπο (C_n, P_c, w) δημιουργείται με φορτίο L_c. Ο αλγόριθμος των Carlisle-Lloyd χρησιμοποιείται για να χρωματίσει έναν μέγιστο αριθμό μονοπατιών στο P_c. Αν L_c<w τότε όλα τα μονοπάτια στο P_c χρωματίζονται χρησιμοποιώντας L_c χρώματα αφήνοντας w-L_c χρώματα αχρησιμοποίητα. Από την παράγραφο 3.1 προκύπτει ότι ο ALG_1 πετυχαίνει παράγοντα προσέγγισης 1/2 για το MaxPC σε δακτύλιο σε χρόνο O(n+m).

Το μειονέκτημα του ALG_1 είναι ότι χρησιμοποιεί διαφορετικά χρώματα για τα μονοπάτια στο P_c και στο P_e. Παρακάτω ακολουθεί ο βελτιωμένος αλγόριθμος ALG_2 που πετυχαίνει παράγοντα προσέγγισης 2/3 ξαναχρησιμοποιώντας χρώματα για τα μονοπάτια στο P_e που ήδη χρησιμοποιούνται στο P_c.

Ο ALG_2 ξεκινά καλώντας τον ALG_1 και δημιουργώντας έτσι έναν αρχικό χρωματισμό. Όταν ένα χρώμα χρησιμοποιείται μόνο μια φορά από ένα μονοπάτι τότε το μονοπάτι ονομάζεται μοναχικό. Τα μοναχικά μονοπάτια αποχρωματίζονται και τα χρώματά τους χρησιμοποιούνται για ζευγάρια μονοπατιών p$\in$$P_c$ και q$\in$$P_e$. Αυτά τα μονοπάτια που δεν μπορούν να μοιραστούν ένα χρώμα προκύπτουν από το μέγιστο ταίριασμα στον αντίστοιχο διμερή γράφο H. Αυτός ο αναχρωματισμός επαναλαμβάνεται μέχρις ότου δεν υπάρχουν άλλα μοναχικά μονοπάτια ή να μην υπάρχουν άλλα υποψήφια ζευγάρια.

ALG_2 για το MaxPC.

Είσοδος: Ένας δακτύλιος R_n, ένα σύνολο P από μονοπάτια και ένας αριθμός διαθέσιμων χρωμάτων w.
Έξοδος: Ένας χρωματισμός ενός υποσυνόλου του P με w χρώματα.

ALG_2:

1. Κάλεσε τον αλγόριθμο ALG_1.
2. Βάλε κάθε αχρησιμοποίητο χρώμα σε μια λίστα ελεύθερων χρωμάτων FC. Για κάθε χρώμα c που χρησιμοποιείται μια φορά αποχρωμάτισε το αντίστοιχο μονοπάτι και βάλε το c στην FC.
3. Βρες το μέγιστο ταίριασμα M στο διμερή γράφο $H=(P_cP_e, E)$, στον οποίο το E περιέχει την ακμή (p,q) με $p\in P_c$ και $q\in P_e$ αν το p και q δεν επικαλύπτονται.
4. Όσο M και FC δεν είναι κενά επανέλαβε
   διέγραψε ένα ζευγάρι (p,q) από το M
   αν color(p)=a $\neq$ nil τότε
   αποχρωμάτισε το p
   αν num(a)=1 τότε
   αποχρωμάτισε το μοναχικό μονοπάτι a
   βάλε το a στο FC
   διέγραψε ένα χρώμα c από το FC
   χρωμάτισε τα p και q με το c
5. αν υπάρχουν ακόμα ελεύθερα χρώματα χρησιμοποίησέ τα για αχρωμάτιστα μονοπάτια στο P με τυχαίο τρόπο.

Παρατήρηση: Στον ALG_2 το βήμα (3) του ALG_1 μπορεί να παραληφθεί αφού τα μονοπάτια που χρωματίζονται σε αυτό το βήμα είναι μοναχικά μονοπάτια και θα αποχρωματιστούν στο βήμα (2) του ALG_2. Είναι όμως χρήσιμο στην ανάλυση της πολυπλοκότητας του ALG_2.

Ανάλυση του παράγοντα προσσέγγισης: Έστω OPT, OPT_c ο μέγιστος αριθμός μονοπατιών που μπορούν να χρωματιστούν με w χρώματα, στο P, P_c, SOL_1 ο αριθμός των μονοπατιών που χρωματίζονται από τον ALG_1. Έστω $\mu=|M|$ και SOL_2 ο αριθμός των μονοπατιών που χρωματίζονται από τον ALG_2. Αν $w\geq L_c+|P_e|$ τότε $SOL_2=OPT=m$. Για τη μη τετριμμένη περίπτωση όπου $w<L_c+|P_e|$ θα δείξουμε ότι $SOL_2\geq 2/3OPT$.

Λήμμα 1: $SOL_1\leq SOL_2$

Απόδειξη: Μετά το βήμα (1) ο ALG_2 έχει SOL_1 χρωματισμένα μονοπάτια. Στα βήματα (2) και (4) το πολύ ένα μονοπάτι είναι αχρωμάτιστο κάθε φορά που ένα χρώμα εισάγεται στο FC. Όμως όταν ένα χρώμα διαγράφεται από το FC τα χρωματισμένα μονοπάτια αυξάνονται κατά ένα ή δύο, οπότε αν το FC είναι άδειο όταν ο ALG_2 τερματίζει τότε $SOL_2\geq SOL_1$. Αν κάποια χρώματα παραμένουν στο FC μετά τον τερματισμό του ALG_2 τότε όλα τα μονοπάτια έχουν χρωματιστεί, δηλαδή $SOL_2=m\geq SOL_1$.

Λήμμα 2: $OPT \leq SOL_2+\mu$

Απόδειξη: Έστω ένας βέλτιστος χρωματισμός με OPT μονοπάτια χρωματισμένα και τ ο αριθμός των μονοπατιών στο P_e που χρησιμοποιούν ένα χρώμα με κάποιο μονοπάτι από το P_c. Προφανώς $\tau \leq \mu$.
Θα μετατρέψουμε την βέλτιστη λύση σε μια λύση που μπορεί να συγκριθεί με την SOL_1 και κατά συνέπεια με την SOL_2. Κατ αρχήν αποχρωμάτισε τα μονοπάτια στο P_e που έχουν το ίδιο χρώμα με κάποιο μονοπάτι στο P_c. Τα χρώματα στο P_e είναι διαφορετικά από αυτά στο P_c. Μετατόπισε όσα περισσότερα χρώματα από το P_e στο P_c. Ένα χρώμα που ελευθερώνεται από τον αποχρωματισμό ενός μονοπατιού στο P_e χρησιμοποιείται για να χρωματίσει ένα μονοπάτι στο P_c. Σ' αυτό το σημείο OPT-τ μονοπάτια έχουν χρωματιστεί.
Περίπτωση 1: Όλα τα μονοπάτια στο P_c έχουν χρωματιστεί. Σ' αυτήν την περίπτωση ο ALG_1 χρωματίζει όλα τα μονοπάτια στο P_c χρησιμοποιώντας το πολύ το ίδιο αριθμό χρωμάτων, οπότε μπορεί να χρωματίσει τον ίδιο αριθμό μονοπατιών στο P_e όπως και ο παραπάνω χρωματισμός. Οπότε $SOL_1 \geq OPT-\tau$.
Περίπτωση 2: Δεν χρωματίστηκαν όλα τα μονοπάτια στο P_c και κανένα χρώμα δεν υπάρχει στο P_e . Όπότε $SOL_1 \geq OPT_c \geq OPT-\tau$.

Και στις δύο περιπτώσεις από το λήμμα 1 προκύπτει $OPT \leq SOL_1+\tau \leq SOL_2+\mu$.

Λήμμα 3: $OPT \leq SOL_2+\min(\mu,w)$.

Απόδειξη: Από την προηγούμενη παράγραφο προκύπτει ότι $OPT \leq SOL_1+w$. Από το λήμμα 1 έχουμε $OPT \leq SOL_2+w$. Σε συνδυασμό με το λήμμα 2 έχουμε την απόδειξη.

Λήμμα 4: $SOL_2 \geq 2*\min(\mu,w)$.

Απόδειξη: Ομοίως μπορούμε να δείξουμε ότι $SOL_2 \geq \min(2\mu,2w)$. Αν το βήμα (4) τερματίζει επειδή δεν υπάρχουν ελεύθερα χρώματα τότε όλα τα w χρώματα έχουν δοθεί σε τουλάχιστον δύο μονοπάτια. Έτσι $SOL_2 \geq 2w$. Διαφορετικά το βήμα (4) τερματίζει επειδή όλα τα ζευγάρια στο M έχουν χρωματιστεί που σημαίνει ότι $SOL_2 \geq 2\mu$. Σε οποιαδήποτε περίπτωση η SOL_2 είναι μεγαλύτερη ή ίση με 2μ ή με $2w$ οπότε $SOL_2 \geq \min(2\mu,2w)$.

Θεώρημα: Ο ALG_2 πετυχαίνει παράγοντα προσέγγισης 2/3 για το MaxPC σε δακτυλίους.

Απόδειξη: Από λήμμα 3,4 ;έχουμε:
$OPT \leq SOL_2+\min(\mu,w) \leq SOL_2+1/2SOL_2=3/2SOL_2$ από το οποίο προκύπτει: $SOL_2 \geq 2/3OPT$.

Το πιο πολύπλοκο βήμα του ALG_2 είναι το βήμα (3) όπου γίνεται ο υπολογισμός του μέγιστου ταιριάσματος. Κατά την υλοποίηση του αλγόριθμου όπως περιγράφεται στα επόμενα κεφάλαια χρησιμοποιείται μια η ρουτίνα για την εύρεση ταιριάσματος σε διμερή γράφο που χρησιμοποιείται και στον

αλγόριθμο της προηγούμενης ενότητας. Όμως ο γράφος H είναι ένας chordal γράφος και μπορούμε να αποφύγουμε την κατασκευή του και τον περίπλοκο υπολογισμό του ταιριάσματος χρησιμοποιώντας το παρακάτω λήμμα:

Λήμμα 5: Έστω p=(a,b) ένα μονοπάτι στο P_c τέτοιο ώστε το a είναι minimum. Έστω q=(c,d) ένα μονοπάτι στο P_e το οποίο δεν επικαλύπτεται με το p και τέτοιο ώστε το c είναι minimum. Τότε υπάρχει ένα μέγιστο ταίριασμα στο H που περιέχει το ζευγάρι (p,q).

Απόδειξη: Ας υποθέσουμε ότι το M είναι ένα μέγιστο ταίριασμα στο H και ότι το (p,q) δεν ανήκει στο M. Τουλάχιστον ένα από τα p,q πρέπει να είναι ταιριασμένο στο M, διαφορετικά το $M\cup\{(p,q)\}$ είναι ένα ταίριασμα με περισσότερα στοιχεία από του M. Έχουμε δύο περιπτώσεις:
Περίπτωση 1: Μόνο ένα από τα p,q είναι ταιριασμένο στο M. Τότε προσθέτοντας το (p,q) στο M και αφαιρώντας το ζευγάρι που περιέχει το p ή το q έχουμε ένα μέγιστο ταίριασμα M' που περιέχει το (p,q).
Περίπτωση 2: Το M περιέχει δυο ζευγάρια (p,q') και (p',q), όπου p'=(a',b'), q'=(c',d'). Αφού τα μονοπάτια σε ένα ζευγάρι δεν επικαλύπτονται έχουμε $b'\leq c$ και $d'\leq a$. Επιπλέον λόγω της επιλογής των p,q έχουμε $c\leq c'$ και $a\leq a'$. Από τις παραπάνω ανισότητες έχουμε $b'\leq c'$ και $d'\leq a'$, δηλαδή τα p',q' δεν επικαλύπτονται. Οπότε μπορούμε να έχουμε ένα μέγιστο ταίριασμα στο H αντικαθιστώντας τα (p,q') και (p',q) στο M με τα (p,q) και (p',q').

Από το παραπάνω λήμμα μπορεί να προκύψει ο παρακάτω αλγόριθμος που υπολογίζει το μέγιστο ταίριασμα:

Είσοδος: ένας δακτύλιος R_n και δύο σύνολα μονοπατιών P_c, P_e στον R_n.
Έξοδος: ένα σύνολο μέγιστης πληθικότητας από μη επικαλυπτόμενα μονοπάτια (p,q) με $p\in P_c$ και $q\in P_e$.

3.3.1 Επιπλέον βελτίωση για το MaxPC.

Παρακάτω ακολουθεί ένας αλγόριθμος που αντικαθιστά το μέγιστο ταίριασμα στους παραπάνω αλγόριθμους και είναι μια απλή υλοποίηση του αλγορίθμου που παρουσιάζεται στο [22]. Ο αλγόριθμος έχει το πλεονέκτημα ότι είναι πολύ εύκολος στην υλοποίηση, ενώ πετυχαίνει και καλύτερα αποτελέσματα στην πράξη.

Αλγόριθμος fast matching
Ταξινόμησε τα μονοπάτια στο P_c $(a_1,b_1),\ldots,(a_s,b_s)$ έτσι ώστε $a_i\leq a_{i+1}$.
Πάρε ένα μονοπάτι από το P_c και ταίριαξέ τα με κάποιο από το P_e αν υπάρχει κάποιο μονοπάτι με το οποίο δεν επικαλύπτεται.
Συνέχισε με όλα τα μονοπάτια στο P_c.

Ο παραπάνω αλγόριθμος εύκολα αποδεικνύεται ότι μπορεί να υλοποιηθεί σε χρόνο $O(n^2)$ χρησιμοποιώντας ταξινόμηση φυσαλίδας, ενώ αν χρησιμοποιηθούν πιο περίπλοκες δομές δεδομένων για την περιγραφή των

μονοπατιών και καλύτερη μέθοδος ταξινόμησης ο αλγόριθμος απαιτεί χρόνο O(n+ (m logL)).

3.4 Ένας βελτιωμένος 2/3-προσεγγιστικός αλγόριθμος για το MaxRPC

Σ' αυτήν την ενότητα παρουσιάζεται ένας βελτιωμένος αλγόριθμος για το MaxRPC. Ο αλγόριθμος αναπτύχθηκε στα πλαίσια αυτής της διπλωματικής εργασίας και χρησιμοποιεί ορισμένες ιδέες από τον βελτιωμένο αλγόριθμο για το MaxPC της ενότητας 3.3. Ο αλγόριθμος ALG_1 και πάλι χρησιμοποιείται σαν υπορουτίνα για τον ALG_2.

ALG:
Εκτέλεσε τους αλγόριθμους ALG_2 και ALG_3 ανεξάρτητα.
Βγάλε σαν έξοδο τη λύση με τη μεγαλύτερη πληθικότητα.

ALG_1 για το MaxRPC.

Είσοδος: Ένας δακτύλιος R_n, ένα σύνολο P από μονοπάτια και ένας αριθμός διαθέσιμων χρωμάτων w.
Έξοδος: Ένας χρωματισμός ενός υποσυνόλου του P με w χρώματα.

ALG_1:
1. Διαμέρισε το P σε ένα σύνολο P_e που περιέχει μονοπάτια που περνούν από την ακμή e={n,1} και σε ένα σύνολο P_c που περιέχει τα υπόλοιπα μονοπάτια (P_e={(i,j)$\in$P| i>j} and P_c=P-P_e).
2. Υπολόγισε μια βέλτιστη λύση του στιγμιότυπου (C_n, P_c, w) χρησιμοποιώντας έναν αλγόριθμο για αλυσίδα.
3. Αν μερικά χρώματα παραμένουν αχρησιμοποίητα χρησιμοποίησε τα για να χρωματίσεις τα μονοπάτια στο P_e (καθένα με ένα διαφορετικό χρώμα).

ALG_2:

Βρες ένα μέγιστο ταίριασμα (πληθικότητας μ) στο γράφο H=(R,E) όπου R το σύνολο των αιτήσεων και το E περιέχει όλα τα ζευγάρια των συμβατών αιτήσεων. Δρομολόγησε τα ταιριασμένα ζευγάρια ώστε να μην επικαλύπτονται και χρωμάτισε τα δύο μονοπάτια με το ίδιο χρώμα, μέχρις ότου δεν υπάρχουν άλλα ζευγάρια ή άλλα χρώματα.

ALG_3:
1. Κάλεσε τον αλγόριθμο ALG_1.
2. Βρες το μέγιστο ταίριασμα M στο διμερή γράφο H=(P_cP_e, E), στον οποίο το E περιέχει την ακμή (p,q) με p$\in P_c$ και q$\in P_e$ αν το p και q δεν επικαλύπτονται.

3. Όσο στο M υπάρχουν ζευγάρια με ένα τουλάχιστον μονοπάτι αχρωμάτιστο επανέλαβε
    αποχρωμάτισε όλα τα μοναχικά μονοπάτια
    βάλε τα ελεύθερα χρώματα στο FC
    αν το FC δεν είναι κενό τότε
        βρες ένα ζευγάρι (p,q) από το M με ένα τουλάχιστον από τα δυο μονοπάτια αχρωμάτιστο
        διέγραψε το ζευγάρι (p,q) από το M
        αποχρωμάτισε το p, q
        χρωμάτισε τα p και q με ένα χρώμα από το FC
4. αν υπάρχουν ακόμα ελεύθερα χρώματα χρησιμοποίησέ τα για αχρωμάτιστα μονοπάτια στο P με τυχαίο τρόπο.

Παρατήρηση: Ο αλγόριθμος ALG είναι ο αλγόριθμος της ενότητας 3.2 με προσθήκη του ALG_3. Στον ALG_3 και πάλι χρησιμοποιείται η ιδέα του βελτιωμένου αλγόριθμου για το MaxPC, δηλαδή αποχρωματίζω μονοπάτια που χρησιμοποιούν ένα χρώμα μια μόνο φορά και με αυτό χρωματίζω δύο μονοπάτια. Μπορεί να αποδειχθεί ότι ο αλγόριθμος είναι 2/3-προσεγγιστικός αλλά στην πράξη πετυχαίνει καλύτερα αποτελέσματα από τον αρχικό αλγόριθμο σε αρκετά στιγμιότυπα.

Η ανάλυση του παράγοντα προσέγγισης είναι παρόμοια με αυτή που παρουσιάστηκε στις ενότητες 3.2 και 3.3.

3.4.1 Επιπλέον βελτίωση για το MaxRPC.

Για το MaxRPC υλοποιήθηκε μια ακόμα βελτίωση με την προσθήκη μιας ακόμα διαδικασίας στον προηγούμενο αλγόριθμο της ενότητας 3.4. Η διαδικασία εκτελείται μετά τον ALG_3 οπότε έχουμε και την τελική έξοδο του αλγόριθμου. Η διαδικασία ονομάζεται color_lonely και λειτουργεί ως εξής:

Color_lonely:

Βρες ένα αχρωμάτιστο μονοπάτι p.
Βρες ένα μοναχικό μονοπάτι που να είναι συμβατό με το p.
Χρωμάτισε το p με το χρώμα του μοναχικού μονοπατιού.

Αυτό που πετυχαίνει η παραπάνω διαδικασία είναι το εξής: σε κάποια στιγμιότυπα υπάρχουν χρώματα που έχουν χρησιμοποιηθεί μόνο μια φορά (σε ένα μοναχικό μονοπάτι). Για κάποιο όμως από αυτά τα μοναχικά μονοπάτια μπορεί να υπάρχει κάποιο αχρωμάτιστο συμβατό μονοπάτι, το οποίο και χρωματίζεται με το ίδιο χρώμα. Οπότε σε τέτοια στιγμιότυπα ο αλγόριθμος πετυχαίνει καλύτερη πληθικότητα.

Κεφάλαιο 4

Υλοποίηση αλγορίθμων για MaxPC και MaxRPC

4.1 Προγραμματιστικό περιβάλλον, τεχνικές προγραμματισμού, δομές δεδομένων.

Η υλοποίηση των αλγορίθμων έγινε σε γλώσσα προγραμματισμού Pascal. Χρησιμοποιήθηκε Borland Pascal αλλά με ελάχιστες αλλαγές οι αλγόριθμοι τρέχουν σε οποιαδήποτε έκδοση της γλώσσας. Η Pascal επιλέχθηκε γιατί υποστηρίζει όλες τις απαραίτητες δομές δεδομένων και τύπους δεδομένων. Ακόμα στη βιβλιογραφία καθώς και στο διαδίκτυο μπορεί κάποιος να βρει βιβλιοθήκες με διαδικασίες που χρησιμεύουν στην ανάπτυξη των συγκεκριμένων αλγορίθμων. Ο έτοιμος κώδικας που χρησιμοποιήθηκε παρουσιάζεται στην επόμενη ενότητα (4.2).

Για την παράσταση των γράφων, μονοπατιών, αιτήσεων σύνδεσης και των υπολοίπων εννοιών χρησιμοποιήθηκαν οι γνωστές από την βιβλιογραφία δομές δεδομένων. Ένας γράφος παριστάνεται με λίστα γειτνίασης (adjacency list):

```
PtrToNode = ^node;
node = record
         id: vertex;
         next: PtrToNode
       end;
graph = record
          size: integer; (* number of vertices *)
          AdjLists: array [vertex] of PtrToNode
        end;
```

Στους αλγόριθμους για το MaxRPC χρησιμοποιήθηκε και πίνακας γειτνίασης (adjacency matrix).

Για τα μονοπάτια και τις αιτήσεις σύνδεσης χρησιμοποιήθηκε πίνακας από record type όπου καταγράφονται η αρχή, το τέλος, το χρώμα, και η κατάσταση του μονοπατιού ή της αίτησης.:

```
const MaxVertex = 100;  (* max number of nodes to handle *)

type path = record
             pstart: integer;
             pend: integer;
             pcolor: integer;
             pstatus: integer;
           end;
     request = record
                 rstart: integer;
                 rend: integer;
                 rcolor: integer;
               end;
```

```
vertex = 1 .. MaxVertex;
path_array = array[vertex] of path;
request_array = array[vertex] of request;
```

Πολύ μεγάλη σημασία δόθηκε στις διάφορες μεταβλητές που αποτελούν παραμέτρους των αλγορίθμων. Δεν χρησιμοποιήθηκαν καθόλου global μεταβλητές αλλά όλες οι παράμετροι περνάνε στην κάθε διαδικασία μέσω μεταβλητών. Έτσι κάθε διαδικασία είναι ανεξάρτητη από το υπόλοιπο πρόγραμμα και μπορεί να χρησιμοποιηθεί με μικρές ή καθόλου αλλαγές σε άλλα προγράμματα. Η βασικές παράμετροι των αλγορίθμων είναι ο αριθμός των κόμβων του δακτυλίου, ο αριθμός των μονοπατιών για το MaxPC ή των αιτήσεων για το MaxRPC και ο αριθμός των χρωμάτων. Οι τιμές των παραμέτρων αυτών μπορούν να παραχθούν τυχαία ή να δίνονται σαν σταθερές, οπότε ο χρήστης να μπορεί να πειραματιστεί με τους αλγόριθμους κάνοντας μόνο αλλαγές σε κάποιες σταθερές.

4.2 Κώδικας από βιβλιοθήκες.

Για την υλοποίηση των αλγορίθμων του MaxPC και MaxRPC χρησιμοποιήθηκαν δύο έτοιμες διαδικασίες. Για το MaxPC χρησιμοποιήθηκε η διαδικασία Match (από το [18]) η οποία βρίσκει το μέγιστο ταίριασμα σε ένα διμερή γράφο. Για το MaxRPC χρησιμοποιήθηκε η διαδικασία general_match η οποία βρίσκει το μέγιστο ταίριασμα σε ένα γενικό γράφο. Ο κώδικας για τη διαδικασία αυτή γράφτηκε από τον Steven Crocker (Department of Computer Science, University of Chicago) και αποτελεί μια υλοποίηση του αλγόριθμου για μέγιστο ταίριασμα των Micali-Vazirani. Οι παραπάνω διαδικασίες και οι αλλαγές που έγιναν στον κώδικά τους παρουσιάζονται στο επόμενο κεφάλαιο.

Κεφάλαιο 5

Πειράματα – αποτελέσματα.

Μετά την ολοκλήρωση της κωδικοποίησης των αλγορίθμων έγιναν μια σειρά από πειράματα με σκοπό την σύγκριση των αλγορίθμων και των βελτιώσεών τους αλλά και την καλύτερη κατανόησή τους. Για να γίνει άμεση σύγκριση των αλγορίθμων με τις βελτιώσεις τους ο κώδικάς των τριών διαφορετικών εκδοχών του κάθε προβλήματος (MaxPC, MaxRPC) συγχωνεύτηκε σε ένα πρόγραμμα. Το πρόγραμμα εκτελούνταν συγκεκριμένο αριθμό φορών αλλάζοντας διάφορες παραμέτρους, όπως θα αναφερθεί παρακάτω για το κάθε πείραμα ξεχωριστά και καταγράφονταν το ποσοστό επιτυχίας του κάθε αλγόριθμου. Για την πιο εύκολη ανάγνωση των στατιστικών δημιουργήθηκαν τα αντίστοιχα ραβδογράμματα.

5.1 Πειράματα MaxPC

Πείραμα Α

Το πείραμα Α είχε σαν σκοπό την σύγκριση των τριών αλγορίθμων του MaxPC σε τυχαία στιγμιότυπα. Ο αριθμός των κόμβων κρατήθηκε σταθερός (n=100), ο αριθμός των μονοπατιών ήταν τυχαίος στο διάστημα 50 έως 300 και ο αριθμός των διαθέσιμων χρωμάτων επίσης τυχαίος από 5 έως 30. Το πρόγραμμα εκτελέστηκε 1000 φορές. Παρακάτω φαίνονται τα ποσοστά των μονοπατιών που χρωματίστηκαν από τον κάθε αλγόριθμο:

ALG1	ALG2	ALG3
0.381	0.391	0.392

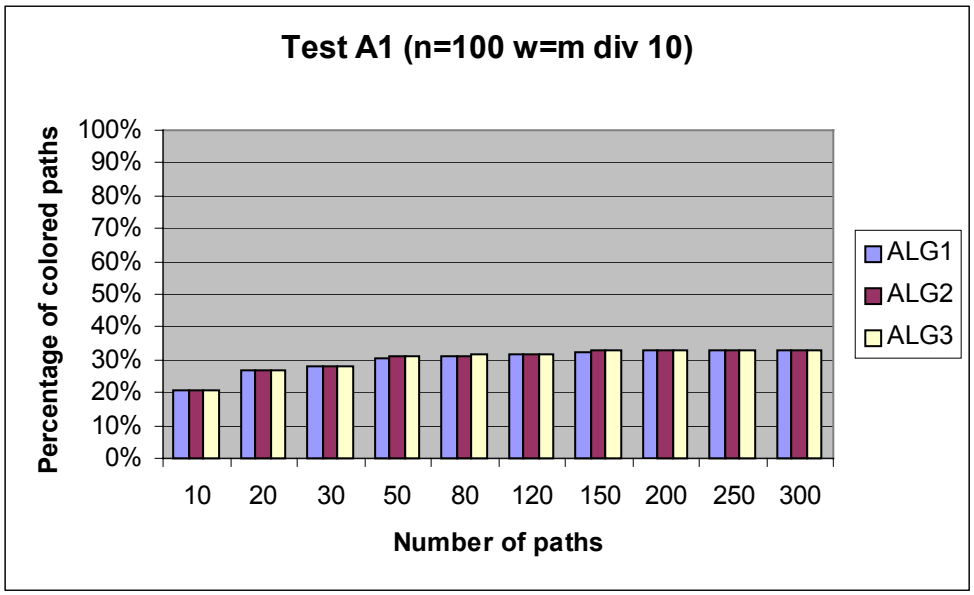
Παρατηρήσεις:

Από την παρατήρηση των ποσοστών προκύπτει ότι η βελτίωση (ALG2) του αρχικού αλγόριθμου πετυχαίνει αρκετά καλύτερα αποτελέσματα, ενώ η τρίτη εκδοχή με το QuickMatching πετυχαίνει ακόμα καλύτερα αποτελέσματα με μικρή όμως διαφορά από τον δεύτερο αλγόριθμο. Η μικρή μόνο διαφορά είναι φυσιολογική αφού οι αλγόριθμοι είναι παρόμοιοι και αλλάζει μόνο ο τρόπος που υλοποιείται το μέγιστο ταίριασμα.

Πείραμα A1

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 10. Το πρόγραμμα όπως και για όλα τα υπόλοιπα πειράματα του MaxPC εκτελέστηκε 500 φορές. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,206	0,206	0,206
20	0,268	0,268	0,268
30	0,279	0,281	0,28
50	0,307	0,311	0,311
80	0,312	0,314	0,315
120	0,316	0,318	0,317
150	0,325	0,327	0,327
200	0,327	0,328	0,329
250	0,329	0,33	0,33
300	0,331	0,332	0,332



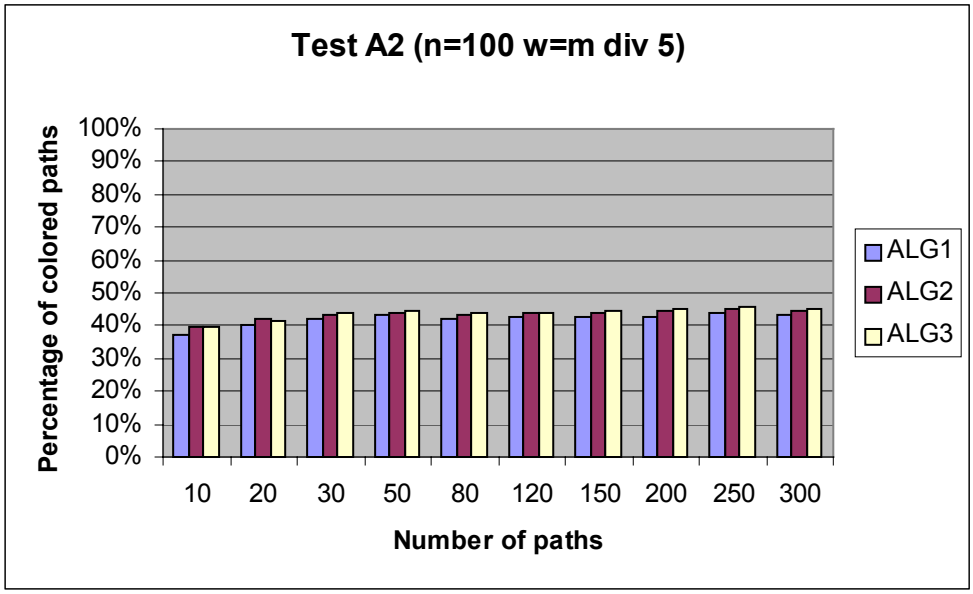
Παρατηρήσεις:

Εδώ φαίνεται ότι για τις περισσότερες τιμές του m ο δεύτερος και τρίτος αλγόριθμος είναι καλύτεροι από τον πρώτο και ο τρίτος γενικά καλύτερος από τον δεύτερο, όμως καθώς ο αριθμός των μονοπατιών αυξάνει οι αλγόριθμοι ALG2 και ALG3 συμπεριφέρονται παρόμοια.

Πείραμα Α2

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 5. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,375	0,394	0,394
20	0,405	0,419	0,414
30	0,42	0,431	0,437
50	0,432	0,442	0,448
80	0,42	0,434	0,437
120	0,425	0,441	0,442
150	0,427	0,442	0,448
200	0,429	0,445	0,45
250	0,437	0,451	0,456
300	0,433	0,448	0,454



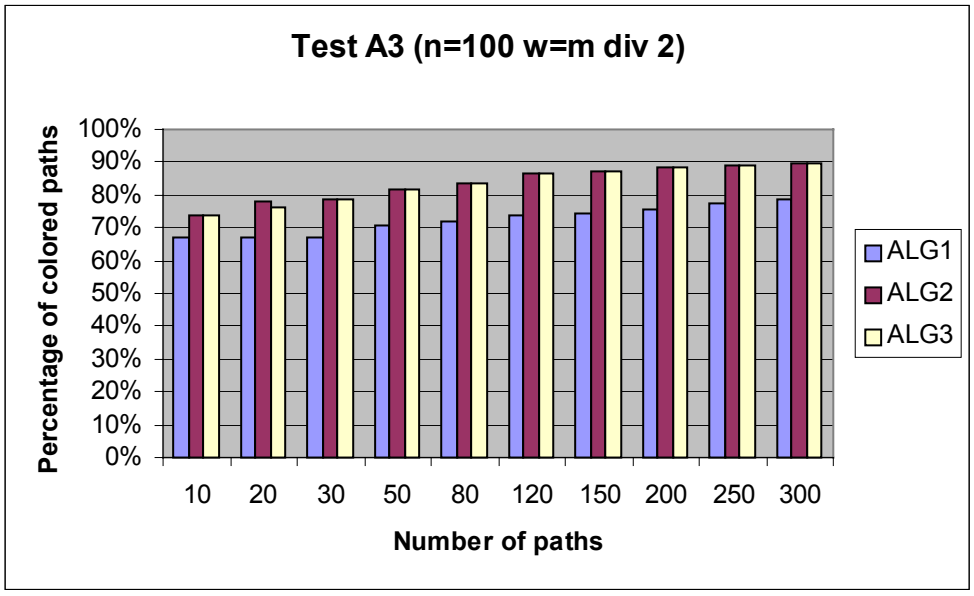
Παρατηρήσεις:

Στο πείραμα αυτό ο αριθμός των χρωμάτων είναι μεγαλύτερος σε σχέση με τον αριθμό των μονοπατιών από ότι στο προηγούμενο πείραμα. Αμέσως φαίνεται η βελτίωση που προσφέρουν οι ALG2 και ALG3. Επίσης φαίνεται ότι ο ALG3 πετυχαίνει στα περισσότερα στιγμιότυπα καλύτερα αποτελέσματα.

Πείραμα A3

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 2. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,673	0,739	0,739
20	0,668	0,778	0,761
30	0,671	0,789	0,789
50	0,705	0,819	0,816
80	0,718	0,837	0,835
120	0,736	0,864	0,865
150	0,746	0,874	0,872
200	0,759	0,883	0,882
250	0,773	0,891	0,892
300	0,786	0,899	0,899



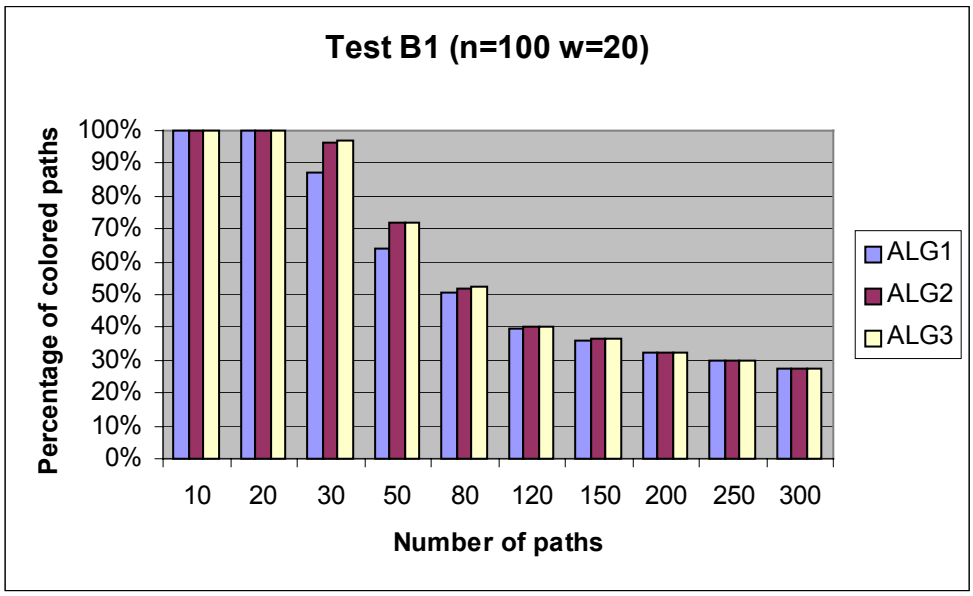
Παρατηρήσεις:

Εδώ όπου τα χρώματα είναι ακόμα περισσότερα σε σχέση με τον αριθμό των μονοπατιών παρατηρούμε ότι οι ALG2 και ALG3 πετυχαίνουν πολύ καλύτερα αποτελέσματα από τον αρχικό αλγόριθμο, ενώ μεταξύ τους δεν υπάρχει μεγάλη διαφορά.

Πείραμα B1

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=20. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	1	1	1
20	1	1	1
30	0,871	0,965	0,968
50	0,64	0,721	0,718
80	0,504	0,518	0,526
120	0,395	0,402	0,404
150	0,36	0,363	0,364
200	0,325	0,326	0,326
250	0,297	0,297	0,297
300	0,277	0,277	0,277



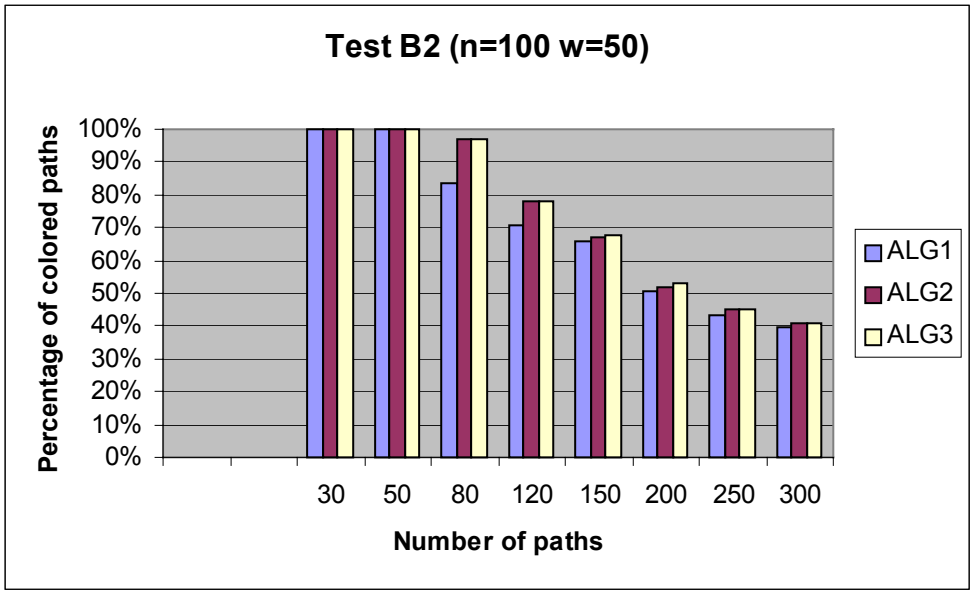
Παρατηρήσεις:

Προφανώς για τις δύο πρώτες τιμές του m όλοι οι αλγόριθμοι χρωματίζουν όλα τα μονοπάτια αφού ο αριθμός των χρωμάτων είναι μεγαλύτερος ή ίσος με τον αριθμό των μονοπατιών. Στη συνέχεια φαίνεται ότι οι ALG2 και ALG3 πετυχαίνουν σημαντική βελτίωση όσο ο αριθμός των μονοπατιών είναι λίγο μεγαλύτερος από τον αριθμό των χρωμάτων. Καθώς ο αριθμός των μονοπατιών γίνεται όλο και μεγαλύτερος σε σχέση με τα διαθέσιμα χρώματα οι ALG2 και ALG3 πετυχαίνουν ελάχιστη ή καθόλου βελτίωση.

Πείραμα B2

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=50. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

M	ALG1	ALG2	ALG3
30	1	1	1
50	1	1	1
80	0,838	0,968	0,969
120	0,706	0,783	0,781
150	0,661	0,669	0,675
200	0,504	0,521	0,531
250	0,434	0,449	0,454
300	0,399	0,406	0,409



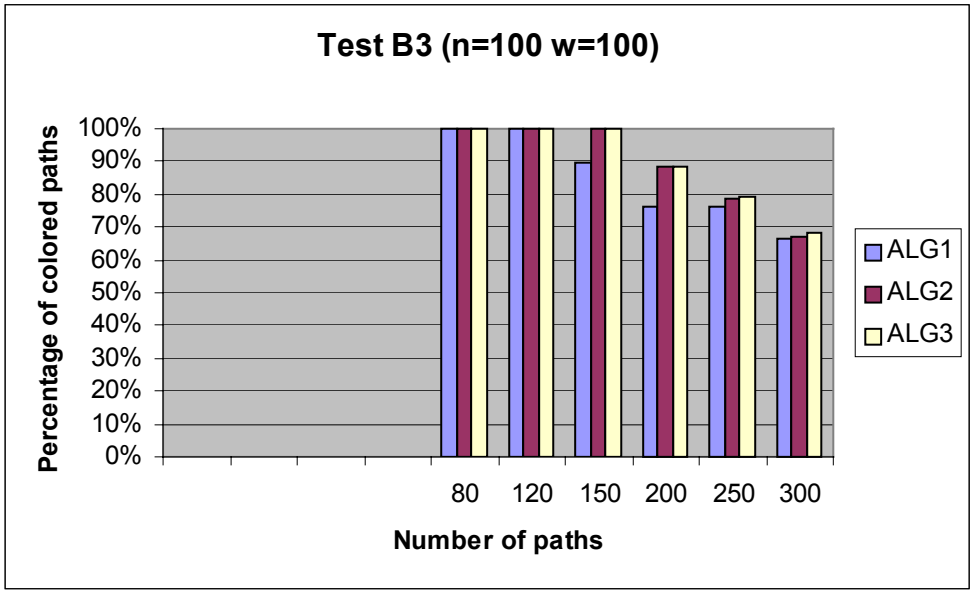
Παρατηρήσεις:

Οι παρατηρήσεις του προηγούμενου πειράματος ισχύουν και εδώ, όμως τώρα που τα χρώματα είναι περισσότερα από ότι στο προηγούμενο πείραμα ο ALG3 γενικά πετυχαίνει αρκετά καλύτερα αποτελέσματα από τον ALG2, ενώ τώρα που τα διαθέσιμα χρώματα είναι πιο πολλά, ακόμα και για μεγάλο αριθμό μονοπατιών οι ALG2 και ALG3 πετυχαίνουν σημαντική βελτίωση σε σχέση με τον ALG_1

Πείραμα B3

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=100. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
80	1	1	1
120	1	1	1
150	0,896	0,999	0,999
200	0,763	0,887	0,886
250	0,765	0,789	0,79
300	0,667	0,67	0,681



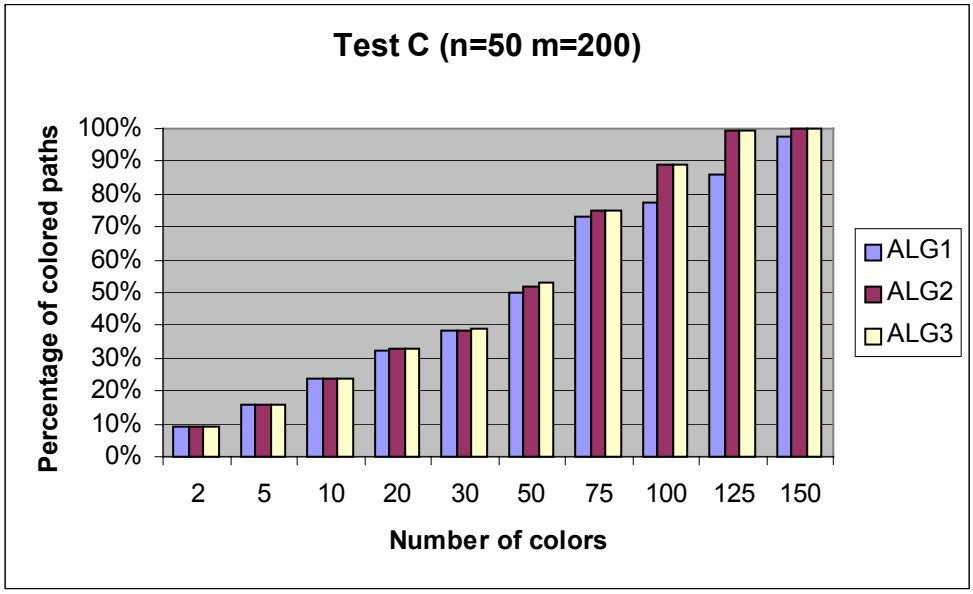
Παρατηρήσεις:

Τα αποτελέσματα του πειράματος είναι παρόμοια με των δύο προηγούμενων. Οι ALG2 και ALG3 πετυχαίνουν αρκετά καλύτερα αποτελέσματα σε σχέση με τον ALG1 τώρα που τα διαθέσιμα χρώματα είναι ακόμα περισσότερα, ενώ ο ALG3 είναι γενικά καλύτερος από τον ALG1.

Πείραμα Γ

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=50), ο αριθμός των μονοπατιών επίσης σταθερός m=200 και ο αριθμός των χρωμάτων w πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

w	ALG1	ALG2	ALG3
2	0,089	0,089	0,089
5	0,16	0,16	0,16
10	0,235	0,235	0,235
20	0,326	0,327	0,328
30	0,383	0,387	0,389
50	0,503	0,517	0,528
75	0,731	0,747	0,748
100	0,772	0,891	0,889
125	0,858	0,996	0,995
150	0,973	1	1



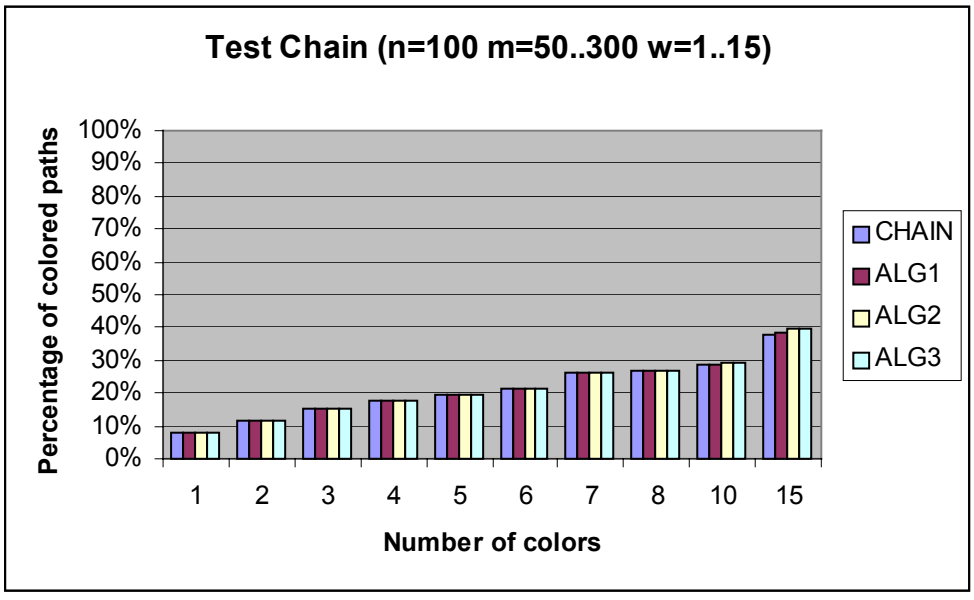
Παρατηρήσεις:

Για τις πρώτες τιμές του w όπου ο αριθμός των χρωμάτων είναι πολύ μικρός σε σχέση με τον αριθμό των μονοπατιών οι ALG2 και ALG3 πετυχαίνουν μικρή ή καθόλου βελτίωση. Καθώς ο αριθμός των χρωμάτων αυξάνει η βελτίωση είναι σημαντική. Για w=150 οι ALG2 και ALG3 πετυχαίνουν απόδοση 100%.

Πείραμα Δ

Το πείραμα Δ είχε σαν σκοπό την σύγκριση του απλού αλγόριθμου της αλυσίδας που πετυχαίνει παράγοντα προσέγγισης ½ με τους τρεις άλλους αλγόριθμους του MaxPC. Τα στιγμιότυπα ήταν τυχαία σε σχέση με τον αριθμό των μονοπατιών ενώ αυξάνονταν ο αριθμός των χρωμάτων. Ο αριθμός των κόμβων κρατήθηκε σταθερός (n=100), ο αριθμός των μονοπατιών ήταν τυχαίος στο διάστημα 50 έως 300 και ο αριθμός των διαθέσιμων χρωμάτων ήταν στο διάστημα από 1 έως 15. Παρακάτω φαίνονται τα ποσοστά των μονοπατιών που χρωματίστηκαν από τον κάθε αλγόριθμο και το αντίστοιχο ραβδόγραμμα:

w	CHAIN	ALG1	ALG2	ALG3
1	0,08	0,08	0,08	0,08
2	0,113	0,113	0,113	0,113
3	0,15	0,15	0,15	0,15
4	0,177	0,177	0,177	0,177
5	0,197	0,197	0,197	0,197
6	0,216	0,216	0,216	0,216
7	0,261	0,261	0,263	0,265
8	0,266	0,266	0,269	0,269
10	0,288	0,288	0,29	0,29
15	0,377	0,386	0,395	0,395



Παρατηρήσεις:

Για τις πρώτες τιμές του w όπου ο αριθμός των χρωμάτων είναι πολύ μικρός σε σχέση με τον αριθμό των μονοπατιών όλοι οι αλγόριθμοι ακόμα και ο πολύ απλός της αλυσίδας πετυχαίνουν τα ίδια αποτελέσματα. Μόνο όταν τα χρώματα αυξηθούν οι ALG1, ALG2, ALG3 πετυχαίνουν βελτίωση. Αυτό σημαίνει ότι για λίγα χρώματα αρκεί να χρησιμοποιήσω τον απλό αλγόριθμο

της αλυσίδας. Για λίγο μεγαλύτερες τιμές του w (π.χ. w=10) η βελτίωση των άλλων αλγορίθμων δεν είναι πολύ σημαντική οπότε και πάλι μπορώ να χρησιμοποιήσω τον αλγόριθμο της αλυσίδας αν αυτό που με ενδιαφέρει περισσότερο είναι η ευκολία υλοποίησης και όχι τόσο η καλύτερη λύση.

5.2 Πειράματα MaxRPC

Πείραμα Α

Όπως και στο αντίστοιχο πείραμα για το MaxPC το πείραμα Α είχε σαν σκοπό την σύγκριση των τριών αλγορίθμων του MaxRPC σε τυχαία στιγμιότυπα. Ο αριθμός των κόμβων κρατήθηκε σταθερός (n=40), ο αριθμός των μονοπατιών ήταν τυχαίος στο διάστημα 20 έως 120 και ο αριθμός των διαθέσιμων χρωμάτων επίσης τυχαίος από 2 έως 12. Το πρόγραμμα εκτελέστηκε 1000 φορές. Παρακάτω φαίνονται τα ποσοστά των αιτήσεων που ικανοποιήθηκαν από τον κάθε αλγόριθμο:

ALG1	ALG2	ALG3
0,508	0,511	0,517

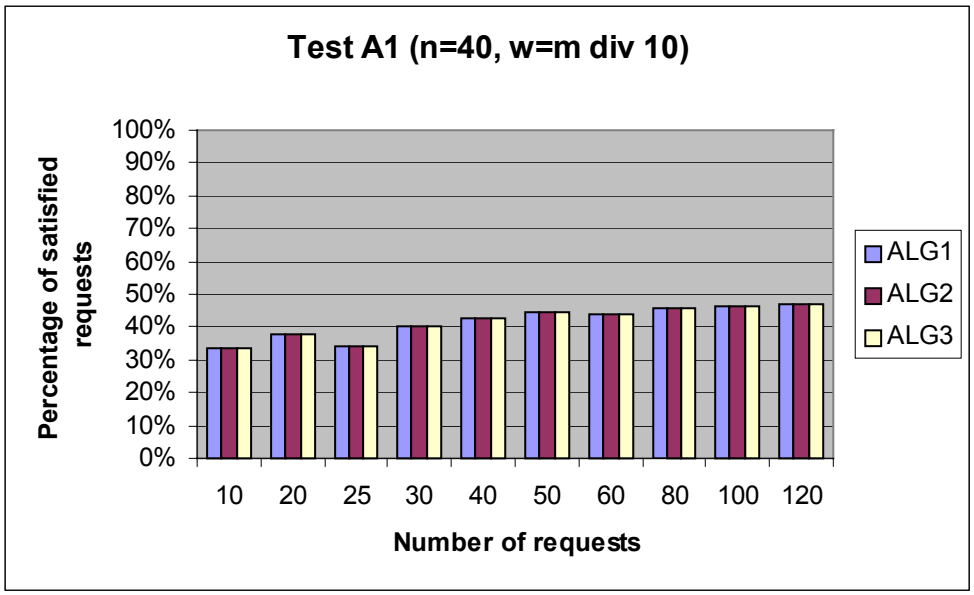
Παρατηρήσεις:

Από τον πίνακα προκύπτει ότι ο ALG2 αποτελεί καλή βελτίωση του ALG1, ενώ ο ALG3 πετυχαίνει ακόμα καλύτερα αποτελέσματα. Ο ALG3 είναι μια βελτίωση του ALG2 αφού προκύπτει από την προσθήκη στον ALG2 της διαδικασίας color_lonely όπως έχει αναφερθεί σε προηγούμενο κεφάλαιο.

Πείραμα A1

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 10. Το πρόγραμμα όπως και για όλα τα υπόλοιπα πειράματα του MaxRPC εκτελέστηκε 500 φορές. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,335	0,335	0,335
20	0,379	0,379	0,379
25	0,344	0,344	0,344
30	0,405	0,405	0,405
40	0,427	0,427	0,427
50	0,447	0,447	0,447
60	0,441	0,441	0,441
80	0,457	0,457	0,457
100	0,463	0,463	0,463
120	0,47	0,47	0,47



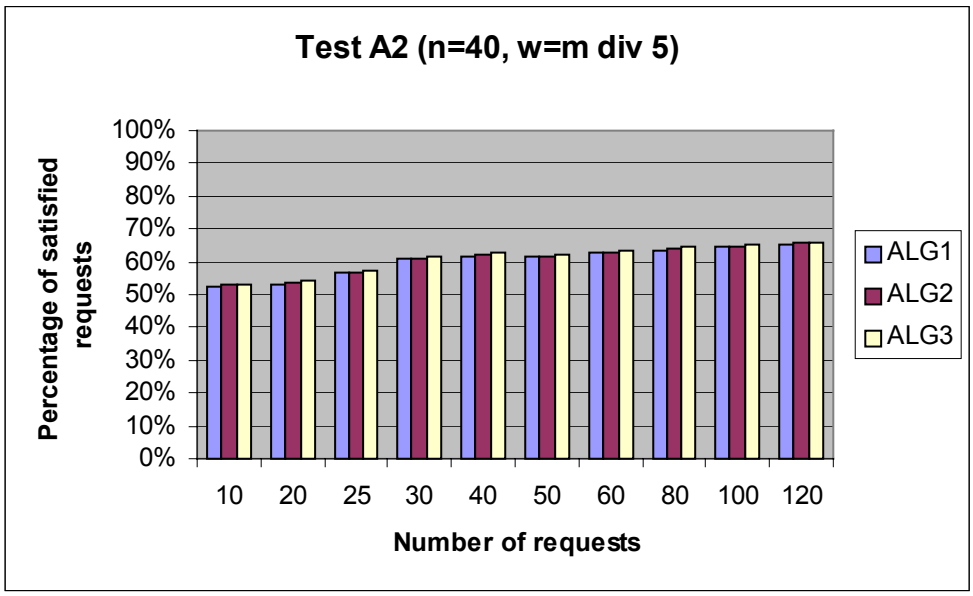
Παρατηρήσεις:

Από το πείραμα αυτό προκύπτει ότι για λίγα χρώματα σε σχέση με τον αριθμό των αιτήσεων (w=m div 10), οι τρεις αλγόριθμοι πετυχαίνουν τα ίδια ποσοστά επιτυχίας. Το βύθισμα για m=25 οφείλεται στο ότι το 25 δεν είναι πολλαπλάσιο του 10 όποτε ο αριθμός των χρωμάτων σε σχέση με τον αριθμό των αιτήσεων είναι μικρότερος σε σχέση με τις διπλανές τιμές.

Πείραμα Α2

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 5. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,527	0,533	0,533
20	0,528	0,538	0,54
25	0,566	0,569	0,571
30	0,61	0,611	0,613
40	0,617	0,621	0,626
50	0,614	0,617	0,622
60	0,628	0,63	0,635
80	0,637	0,638	0,644
100	0,648	0,649	0,654
120	0,655	0,656	0,661



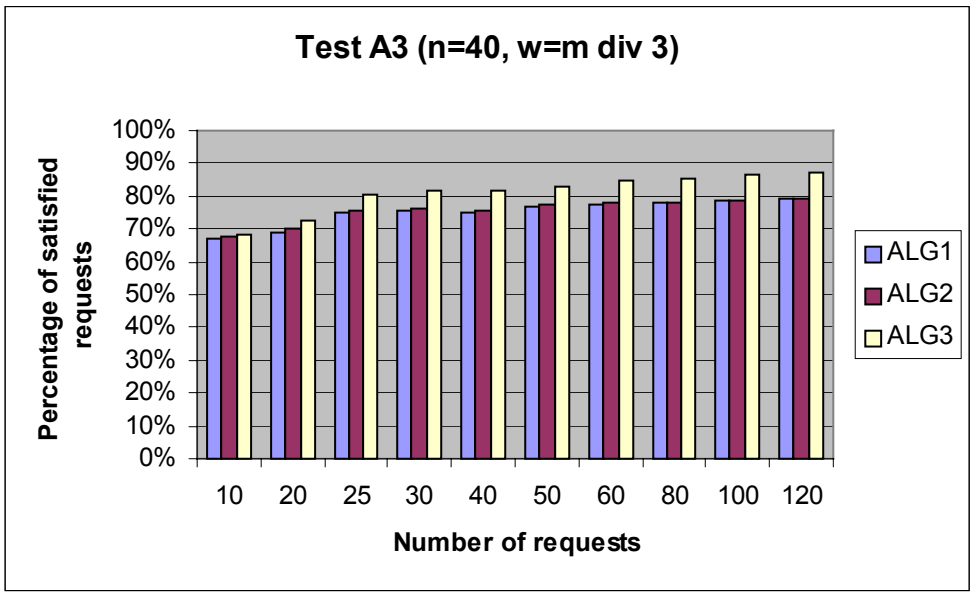
Παρατηρήσεις:

Σ΄ αυτό το πείραμα φαίνεται ότι καθώς τα διαθέσιμα χρώματα είναι περισσότερα (από ότι στο προηγούμενο πείραμα) σε σχέση με τον αριθμό των αιτήσεων οι ALG2 και ALG3 πετυχαίνουν σε όλες τις τιμές του m καλύτερα αποτελέσματα.

Πείραμα Α3

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=100), ο αριθμός των μονοπατιών m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν w=m div 2. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	0,668	0,679	0,683
20	0,692	0,7	0,728
25	0,751	0,758	0,804
30	0,756	0,762	0,815
40	0,749	0,759	0,815
50	0,771	0,775	0,829
60	0,775	0,779	0,848
80	0,779	0,781	0,852
100	0,787	0,789	0,864
120	0,792	0,794	0,873



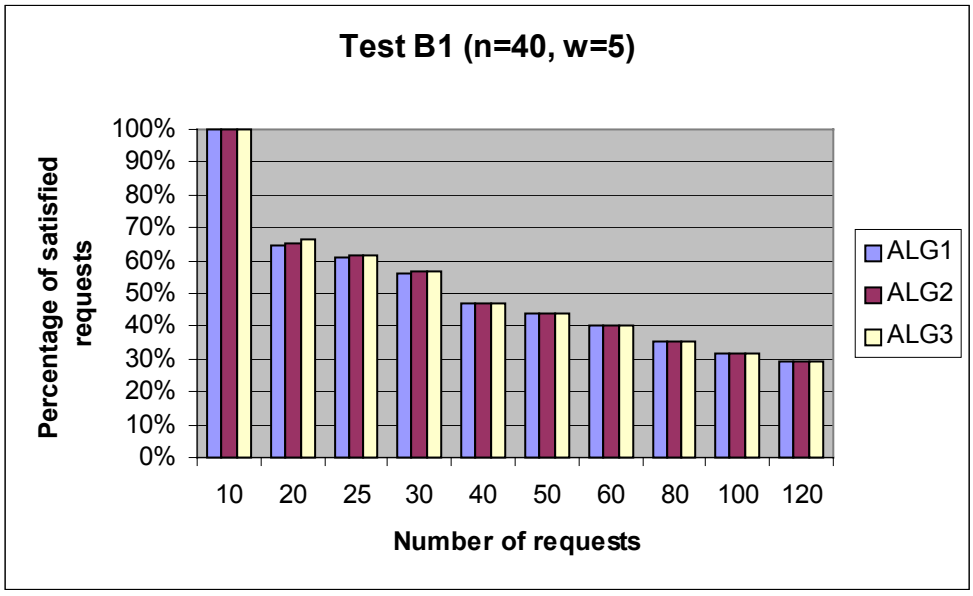
Παρατηρήσεις:

Σ΄ αυτό το πείραμα ο αριθμός των χρωμάτων ήταν ακόμα μεγαλύτερος σε σχέση με τον αριθμό των αιτήσεων (από ότι στα προηγούμενα δύο πειράματα) και οι ALG2 και ALG3 πετυχαίνουν καλύτερα αποτελέσματα. Ο ALG3 μάλιστα αποτελεί πολύ σημαντική βελτίωση σε σχέση με τον ALG2.

Πείραμα B1

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=5. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
10	1	1	1
20	0,648	0,652	0,665
25	0,611	0,613	0,616
30	0,564	0,566	0,566
40	0,469	0,469	0,469
50	0,438	0,438	0,438
60	0,403	0,403	0,403
80	0,352	0,352	0,352
100	0,32	0,32	0,32
120	0,29	0,29	0,29



Παρατηρήσεις:

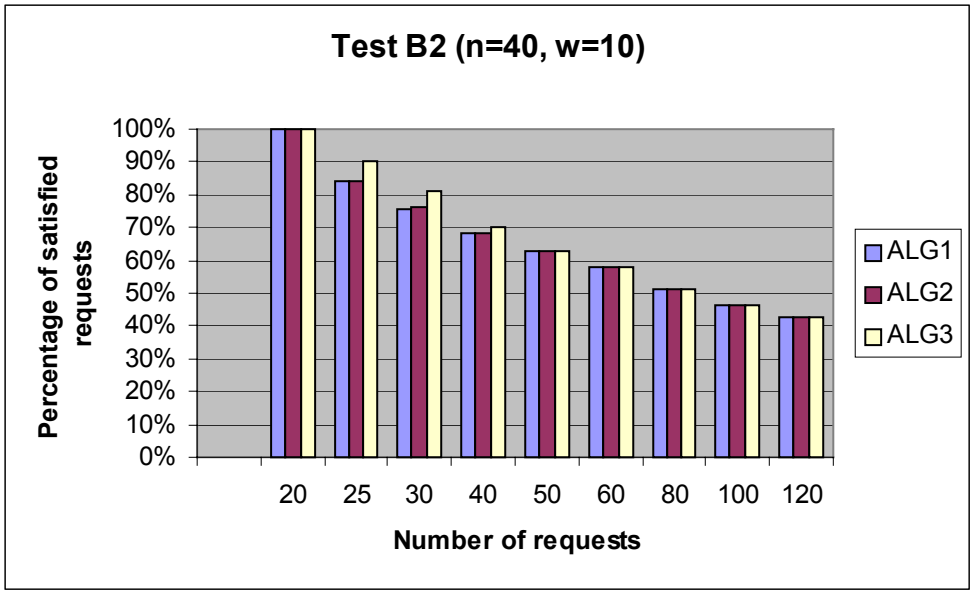
Το πρώτο που παρατηρούμε εδώ είναι ότι για m=10 και οι τρεις αλγόριθμοι πετυχαίνουν απόδοση 100%. Αυτό εξηγείται ως εξής: Για w=m div 2 όλοι οι αλγόριθμοι για το MaxRPC πετυχαίνουν απόδοση 100% για την πλειοψηφία των στιγμιότυπων. Αυτό συμβαίνει γιατί στα περισσότερα στιγμιότυπα ο γράφος των συμβατών αιτήσεων έχει τέλειο ταίριασμα και άρα τα χρώματα που απαιτούνται για να χρωματίσουν όλα τα ζευγάρια των αιτήσεων αρκεί να είναι τα μισά σε σχέση με τον συνολικό αριθμό των αιτήσεων. Στη συνέχεια και καθώς τα χρώματα γίνονται όλο και πιο λίγα σε

σχέση με τον αριθμό των αιτήσεων, οι ALG2 και ALG3 πετυχαίνουν μικρή ή καθόλου βελτίωση.

Πείραμα B2

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=10. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
20	1	1	1
25	0,839	0,843	0,901
30	0,756	0,765	0,814
40	0,68	0,685	0,702
50	0,626	0,627	0,629
60	0,579	0,58	0,582
80	0,512	0,512	0,512
100	0,464	0,464	0,464
120	0,429	0,429	0,429



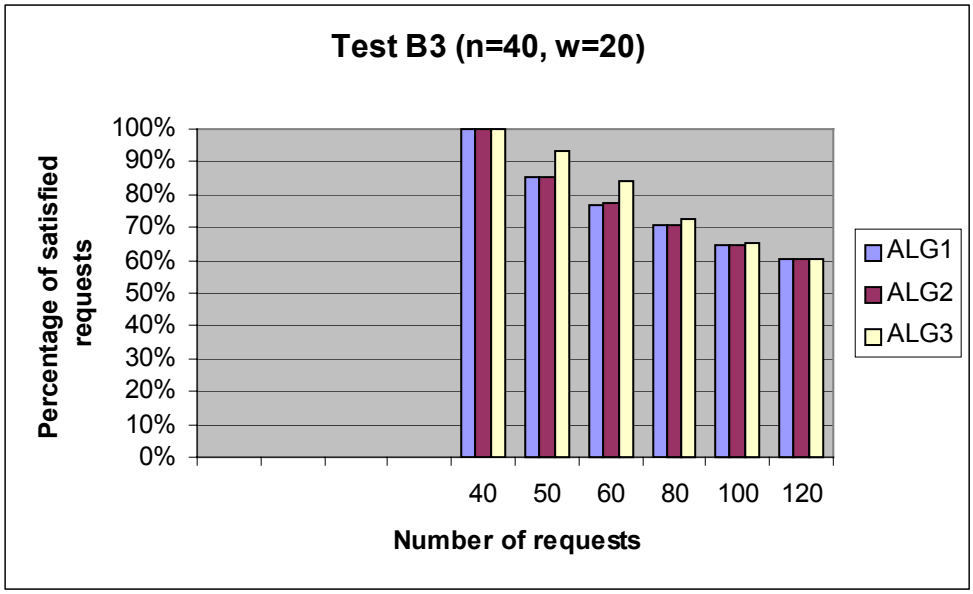
Παρατηρήσεις:

Σ΄ αυτό το πείραμα ισχύουν και πάλι οι παρατηρήσεις του προηγούμενου πειράματος. Επιπλέον φαίνεται η πολύ καλύτερη απόδοση του ALG3 όσο ο αριθμός των χρωμάτων είναι μεγάλος σε σχέση με τον αριθμό των αιτήσεων.

Πείραμα B3

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων m πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα και ο αριθμός των χρωμάτων ήταν σταθερός w=20. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

m	ALG1	ALG2	ALG3
40	1	1	1
50	0,854	0,854	0,93
60	0,771	0,774	0,842
80	0,705	0,708	0,728
100	0,648	0,649	0,654
120	0,603	0,603	0,604



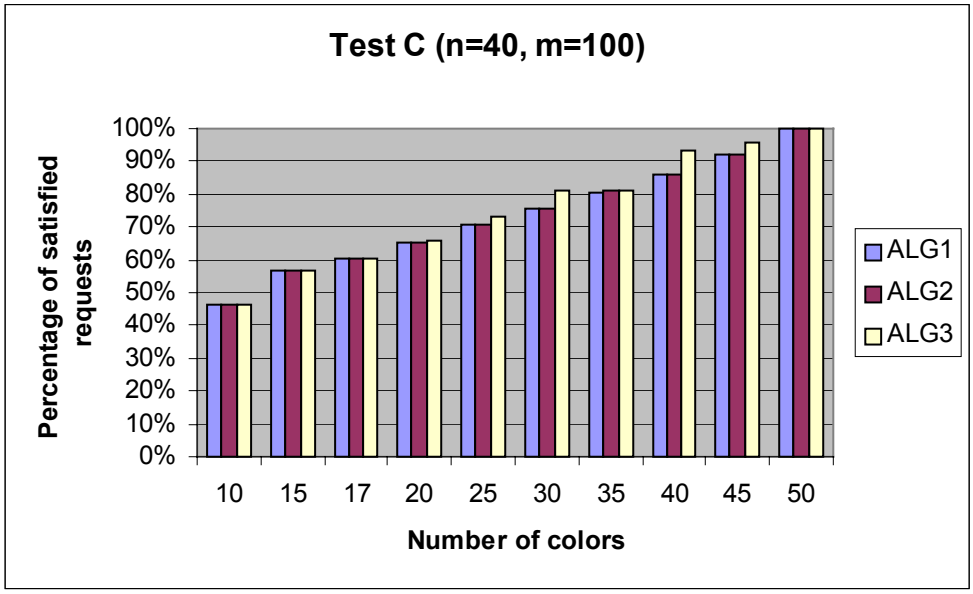
Παρατηρήσεις:

Στο πείραμα αυτό επαληθεύονται οι παρατηρήσεις των δύο προηγούμενων πειραμάτων.

Πείραμα Γ

Σ΄ αυτό το πείραμα ο αριθμός των κόμβων ήταν σταθερός (n=40), ο αριθμός των αιτήσεων επίσης σταθερός m=100 και ο αριθμός των χρωμάτων w πήρε τις τιμές που φαίνονται στον παρακάτω πίνακα. Ακολουθεί πίνακας τιμών και το αντίστοιχο ραβδόγραμμα.

w	ALG1	ALG2	ALG3
10	0,464	0,464	0,464
15	0,567	0,567	0,567
17	0,604	0,605	0,606
20	0,651	0,652	0,658
25	0,707	0,709	0,733
30	0,757	0,759	0,814
35	0,806	0,808	0,809
40	0,86	0,861	0,93
45	0,921	0,921	0,958
50	1	1	1



Παρατηρήσεις:

Στο πείραμα αυτό φαίνεται και πάλι ότι για λίγα χρώματα σε σχέση με τον αριθμό των αιτήσεων οι αλγόριθμοι πετυχαίνουν τα ίδια αποτελέσματα. Καθώς ο αριθμός των χρωμάτων γίνεται όλο και μεγαλύτερος σε σχέση με τον αριθμό των αιτήσεων οι ALG2 και ALG3 πετυχαίνουν καλύτερα αποτελέσματα, με τον ALG3 να είναι πολύ καλύτερος. Για w=50 δηλαδή όταν τα χρώματα είναι ίσα με τα μισά μονοπάτια όπως έχουμε εξηγήσει και παραπάνω και οι τρεις αλγόριθμοι έχουν 100% απόδοση.

Κεφάλαιο 6

Περιγραφή διαδικασιών που υλοποιήθηκαν.

6.1 Διαδικασίες MaxPC

Read_ring procedure

Λειτουργία: Εισάγει τα δεδομένα του προβλήματος.

Δήλωση:

```
procedure read_ring(var p_matrix: path_array; var ring_size: integer;
                    var n_paths: integer;     var n_colors: integer);
```

Σχόλια: Στη διαδικασία αυτή εισάγονται τα δεδομένα του προβλήματος. Τα δεδομένα μπορούν να δοθούν ως σταθερές από το χρήστη ή να παραχθούν τυχαία στιγμιότυπα με τη βοήθεια της συνάρτησης Random της Pascal. Στην μεταβλητή p_matrix αποθηκεύονται τα μονοπάτια ενώ στις ring_size, n_paths, n_colors ο αριθμός των κόμβων, των μονοπατιών και των χρωμάτων αντίστοιχα. Ο μόνος περιορισμός που υπάρχει κατά τη δημιουργία των μονοπατιών είναι τα μονοπάτια να μην έχουν την ίδια αρχή και τέλος. Τα μονοπάτια αρχικοποιούνται ως αχρωμάτιστα, μη ελεγμένα. Δηλαδή όσα από αυτά θα μπούνε στην αλυσίδα θα ελεγχθούν για το αν μπορούν να χρωματιστούν με κάποιο χρώμα. Επίσης αρχικά όλα τα μονοπάτια αρχικοποιούνται ως στοιχεία του συνόλου Q, δηλαδή κατά τη δημιουργία του δακτυλίου όλα τα μονοπάτια ανήκουν στο σύνολο Q μέχρι να δημιουργηθεί η αλυσίδα από τη διαδικασία create_chain.

Create_chain procedure

Λειτουργία: Κατασκευάζει την αλυσίδα.

Δήλωση:

```
procedure create_chain(var p_matrix: path_array;
                       var c_matrix:path_array;
                       ring_size: integer;   n_paths: integer;
                       var chain_paths: integer;
                       var random_e: integer);
```

Σχόλια: Η διαδικασία αυτή κατασκευάζει την αλυσίδα, από το σύνολο των μονοπατιών (p_matrix) και αποθηκεύει τα μονοπάτια της στη μεταβλητή c_matrix. Η ακμή που επιλέγεται για τη δημιουργία της αλυσίδας αποθηκεύεται στη μεταβλητή random_e και είναι για λόγους απλότητας η

τελευταία ακμή του δακτυλίου. Έχει γίνει όμως πρόβλεψη ώστε η ακμή να είναι τυχαία ή να επιλέγεται με κριτήρια. Οι μεταβλητές ring_size, n_paths, chain_paths ;έχουν αντίστοιχα τον αριθμό των κόμβων, μονοπατιών, μονοπατιών της αλυσίδας. Η διαδικασία ελέγχει αν ένα μονοπάτι περνά από την τελευταία ακμή του δακτυλίου και αν όχι το χαρακτηρίζει ως μονοπάτι της αλυσίδας (p_matrix.in_set:=1). Όλα τα χαρακτηρισμένα μονοπάτια εισάγονται στην αλυσίδα (c_matrix).

Chain_coloring procedure

Λειτουργία: Χρωματίζει την αλυσίδα.

Δήλωση:

```
procedure chain_coloring(var c_matrix: path_array;
                         var clr_matrix: color_array;
                         var clr_usage_matrix: color_array;
                         c_length,n_paths,n_colors: integer);
```

Σχόλια: Η διαδικασία αυτή δίνει χρώμα σε κάθε μονοπάτι της αλυσίδας. Στις μεταβλητές clr_matrix και clr_usage_matrix καταγράφονται αντίστοιχα τα ελεύθερα χρώματα και πόσες φορές χρησιμοποιείται το κάθε χρώμα. Οι μεταβλητές c_length, n_paths, n_colors έχουν αντίστοιχα το μήκος της αλυσίδας, τον αριθμό των μονοπατιών της αλυσίδας και τον αριθμό των χρωμάτων. Αρχικά όλα τα χρώματα είναι αχρησιμοποίητα. Η αλυσίδα ελέγχεται κόμβο κόμβο από αριστερά προς τα δεξιά. Σε κάθε κόμβο ελέγχονται (pstatus:=1) τα μονοπάτια που ξεκινούν και τελειώνουν στον συγκεκριμένο κόμβο και καταγράφεται ο αριθμός των μονοπατιών που επικαλύπτονται. Αν ο αριθμός των επικαλύψεων είναι μεγαλύτερος του αριθμού των χρωμάτων τότε το μονοπάτι που τελειώνει πιο δεξιά χαρακτηρίζεται ως ανενεργό (pstatus:=0). Τα μονοπάτια που μπορούν να χρωματιστούν χαρακτηρίζονται ως ενεργά (pstatus:=2). Τα ενεργά μονοπάτια χρωματίζονται και καταγράφεται η χρήση του κάθε χρώματος.

Create_setQ procedure

Λειτουργία: Δημιουργεί το σύνολο μονοπατιών Q.

Δήλωση:

```
procedure create_setQ(var p_matrix: path_array; (* all paths *)
                      var q_matrix: path_array; (* paths in set Q *)
                      n_paths: integer;
                      var q_n_paths:integer;
                      share_e: integer);
```

Σχόλια: Για όλα τα μονοπάτια (p_matrix) γίνεται έλεγχος αν ανήκουν στην αλυσίδα (in_set=1). Όσα δεν ανήκουν εισάγονται στο σύνολο Q (q_matrix). Οι

μεταβλητές n_paths, q_n_paths έχουν αντίστοιχα τον αριθμό όλων των μονοπατιών και τον αριθμό των μονοπατιών του συνόλου Q. Η share_e είναι η κοινή ακμή των μονοπατιών του Q.

Use_free_colors procedure

Λειτουργία: Χρησιμοποιεί τα ελεύθερα χρώματα.

Δήλωση:

```
procedure use_free_colors(var clr_usage_matrix: color_array;
                          var path_matrix: path_array;
                          n_paths,n_colors: integer);
```

Σχόλια: Μετά τον χρωματισμό των μονοπατιών της αλυσίδας μπορεί να υπάρχουν αχρησιμοποίητα χρώματα. Αυτά χρησιμοποιούνται για να χρωματίσουν μονοπάτια του συνόλου Q. Η χρήση των χρωμάτων καταγράφεται στη μεταβλητή clr_usage_matrix, ενώ τα μονοπάτια στην path_matrix. Οι μεταβλητές n_paths, n_colors έχουν αντίστοιχα τον συνολικό αριθμό των μονοπατιών και των χρωμάτων.

Create_bipartiteH procedure

Λειτουργία: Δημιουργεί το διμερή γράφο Η από τα σύνολα C και Q.

Δήλωση:

```
procedure create_bipartiteH(var g: graph;
                                c_matrix,q_matrix: path_array;
                                c_n_paths,q_n_paths: integer);
```

Σχόλια: Η διαδικασία αυτή δημιουργεί έναν διμερή γράφο g με τη μορφή λίστας γειτνίασης. Η αριστερή μεριά του γράφου αποτελείται από τα μονοπάτια της αλυσίδας και η δεξιά από τα μονοπάτια του συνόλου Q. Για κάθε μονοπάτι της αλυσίδας γίνεται έλεγχος αν δεν επικαλύπτεται με κάθε ένα από τα μονοπάτια του συνόλου Q. Κάθε φορά που βρίσκεται ένα συμβατό ζεύγος (όπου τα μονοπάτια δεν επικαλύπτονται) προστίθεται η αντίστοιχη ακμή στο διμερή γράφο. Οι c_matrix, q_matrix έχουν τα μονοπάτια της αλυσίδας και του συνόλου Q αντίστοιχα, ενώ οι c_n_paths και q_n_paths τα αντίστοιχα πλήθη των μονοπατιών.

Match procedure

Λειτουργία: Βρίσκει το μέγιστο ταίριασμα σε ένα διμερή γράφο.

Δήλωση:

```
procedure Match(var G: graph;
                lside,rside: integer;
                var mate: partners;
                var mqueue: queue_array; var mqhead,mqtail: integer);
```

Σχόλια: Η διαδικασία αυτή βρίσκει ένα μέγιστο ταίριασμα σε ένα διμερή γράφο σε χρόνο O(sqrt(V)*E). Τα αποτελέσματα καταγράφονται στον πίνακα mate. Η μεταβλητή G είναι ο διμερής γράφος ενώ οι lside και rside είναι ο αριθμός των αριστερών κόμβων και των δεξιών κόμβων αντίστοιχα. Η διαδικασία προϋποθέτει την ύπαρξη μιας σειράς διαδικασιών που χειρίζονται ουρές. Η mqueue είναι η ουρά, ενώ οι mqhead, mqtail είναι η αρχή και το τέλος της ουράς. Όπως έχει αναφερθεί και στο προηγούμενο κεφάλαιο η διαδικασία έχει παρθεί από το [18].

Print_graph procedure

Λειτουργία: Εκτυπώνει μια λίστα γειτνίασης που αντιπροσωπεύει έναν γράφο.

Δήλωση:

```
procedure print_graph(g: graph);
```

Σχόλια: Η διαδικασία εκτυπώνει τις κορυφές του γράφου g και για κάθε κορυφή τις αντίστοιχες γειτονικές κορυφές. Χρησιμοποιεί τη διαδικασία print_list.

Print_list procedure

Λειτουργία: Εκτυπώνει τους κόμβους μιας λίστας γειτνίασης.

Δήλωση:

```
procedure print_list(p: PtrToNode);
```

Σχόλια: Η διαδικασία εκτυπώνει τους κόμβους της λίστας γειτνίασης p.

Create_queue procedure

Λειτουργία: Αρχικοποίηση της ουράς.

Δήλωση:

```
procedure CreateQueue(var head,tail: integer);
```

Enqueue procedure

Λειτουργία: Εισάγει στην ουρά.

Δήλωση:

```
procedure Enqueue(var q: queue_array;
                  var head,tail: integer; x: vertex);
```

Σχόλια: Εισάγει την κορυφή x στην ουρά q. Head, tail η αρχή και το τέλος της ουράς.

Dequeue procedure

Λειτουργία: Διαγράφει από την ουρά.

Δήλωση:

```
procedure Dequeue(var q: queue_array;
                  var head,tail: integer; var x: vertex);
```

Σχόλια: Διαγράφει από την ουρά q τον κόμβο x. Head, tail η αρχή και το τέλος της ουράς.

IsEmptyQueue function

Λειτουργία: Αληθής αν η ουρά είναι άδεια.

Δήλωση:

```
function IsEmptyQueue(head,tail: integer): boolean;
```

6.2 Διαδικασίες βελτίωσης του MaxPC.

Uncolor_lonely_paths procedure

Λειτουργία: Αποχρωματίζει ένα μοναχικό μονοπάτι.

Δήλωση:

```
procedure uncolor_lonely_paths(var c_matrix: path_array;
                               var clr_usage_matrix: color_array;
                               n_paths,n_colors: integer);
```

Σχόλια: Η διαδικασία αυτή βρίσκει τα μοναχικά μονοπάτια στην αλυσίδα c_matrix, τα αποχρωματίζει και ελευθερώνει το χρώμα τους, έτσι ώστε αυτό να μπορεί να χρησιμοποιηθεί πιθανόν από περισσότερα από ένα μονοπάτια. Η μεταβλητή clr_usage_matrix καταγράφει τη χρήση των χρωμάτων. Οι μεταβλητές n_paths, n_colors είναι αντίστοιχα ο αριθμός των μονοπατιών της αλυσίδας και ο αριθμός των διαθέσιμων χρωμάτων.

Color_mates procedure

Λειτουργία: Υλοποιεί το βήμα 4 του ALG_2 της ενότητας 3.3.
Δήλωση:

```
procedure color_mates(m_match: partners;
                      var c_matrix: path_array;
                      var q_matrix: path_array;
                      var clr_usage_matrix: color_array;
                      c_n_paths,q_n_paths,n_colors: integer);
```

Σχόλια: Η διαδικασία χρωματίζει τα μονοπάτια της αλυσίδας c_matrix και του συνόλου q_matrix χρησιμοποιώντας το μέγιστο ταίριασμα m_match και τον πίνακα καταγραφής της χρήσης των χρωμάτων clr_usage_matrix. Οι μεταβλητές c_n_paths, q_n_paths, n_colors είναι αντίστοιχα οι αριθμοί των μονοπατιών της αλυσίδας, των μονοπατιών του συνόλου Q και του αριθμού των διαθέσιμων χρωμάτων.

Bubble_sort procedure

Λειτουργία: Ταξινόμηση φυσαλίδας.

Δήλωση:

```
procedure bubble_sort(var matrix: path_array; n: integer);
```

Σχόλια: Ταξινομεί τα n μονοπάτια του πίνακα matrix κατά αύξουσα σειρά σύμφωνα με την αρχή του κάθε μονοπατιού.

Custom_match procedure

Λειτουργία: Υλοποιεί τον αλγόριθμο **Fast Matching** της παραγράφου 3.3.1.

Δήλωση:

```
procedure custom_match (var mate: partners;
                        c_matrix,q_matrix: path_array;
                        c_n_paths,q_n_paths: integer);
```

Σχόλια: Η διαδικασία αυτή βρίσκει ένα μέγιστο ταίριασμα μεταξύ των μονοπατιών της αλυσίδας c_matrix και του συνόλου μονοπατιών q_matrix. Το αποτέλεσμα καταγράφεται στον πίνακα mate. Οι μεταβλητές c_n_paths, q_n_paths είναι αντίστοιχα ο αριθμός των μονοπατιών της αλυσίδας και του συνόλου Q. Τα μονοπάτια των δύο παραπάνω συνόλων έχουν προηγουμένως ταξινομηθεί κατά αύξουσα σειρά σύμφωνα με την αρχή των μονοπατιών.

6.3 Διαδικασίες MaxRPC

Οι τέσσερις παρακάτω διαδικασίες είναι παρόμοιες με τις αντίστοιχες που χρησιμοποιήθηκαν για τους αλγόριθμους του MaxPC και για την κατανόησή τους αρκεί κάποιος να ανατρέξει στην ενότητα 5.1:

Read_ring procedure
Create_chain procedure
Chain_coloring procedure
Use_free_colors procedure

Παρακάτω ακολουθούν οι υπόλοιπες διαδικασίες για το MaxRPC.

Create_compatibilityG procedure

Λειτουργία: Δημιουργεί τον πίνακα συμβατότητας αιτήσεων.

Δήλωση:

```
procedure create_compatibilityG(r_matrix: request_array;
                                var cmp_matrix: compatibility_array;
                                n_requests: integer;
                                var n_edges: integer);
```

Σχόλια: Η διαδικασία αυτή δημιουργεί τον πίνακα συμβατότητας cmp_matrix των αιτήσεων. Ο πίνακας είναι δισδιάστατος με ίσο αριθμό γραμμών και στηλών που είναι ίσος με τον αριθμό των αιτήσεων. Ο πίνακας αρχικοποιείται με όλα τα κελιά του ίσα με 0. Όταν μια αίτηση είναι συμβατή με μια άλλη τότε το αντίστοιχο κελί του πίνακα γίνεται 1. Οι αιτήσεις είναι αποθηκευμένες στον πίνακα r_matrix. Η μεταβλητή n_requests έχει τον αριθμό τον αιτήσεων, ενώ η n_edges έχει στο τέλος της διαδικασίας τον αριθμό των συμβατών αιτήσεων.

General_match procedure

Λειτουργία: Βρίσκει το μέγιστο ταίριασμα σε ένα γενικό γράφο.

Δήλωση:

```
procedure general_match(cmp_matrix: compatibility_array;
                        var matesg: partners;
                        n_nodes: integer;
                        n_edges: integer;
                        var fmcount: integer);
```

Σχόλια: Η διαδικασία αυτή είναι μια κωδικοποίηση του αλγόριθμου των Micali-Vazirani για την εύρεση του μέγιστου ταιριάσματος σε έναν γενικό γράφο [17]. Ο κώδικας έχει γραφτεί από τον Steven Crocker του Πανεπιστημίου του

Chicago όπως αναφέρθηκε στην ενότητα 4.2. Η διαδικασία χρησιμοποιεί τον πίνακα συμβατότητας αιτήσεων cmp_matrix που δημιουργείται στη διαδικασία create_compatibilityG. Ο πίνακας αυτός είναι ο πίνακας γειτνίασης του γράφου του οποίου θέλω να υπολογίσω το μέγιστο ταίριασμα. Η διαδικασία δημιουργεί τον πίνακα matesg με τους ταιριασμένους κόμβους και τον αριθμό των ζευγαριών fmcount. Οι μεταβλητές n_nodes, n_edges έχουν αντίστοιχα τον αριθμό των κόμβων και των ακμών του γραφήματος.

Η μόνη αλλαγή που έγινε στην διαδικασία ήταν στον τρόπο που διαβάζονται τα δεδομένα. Η διαδικασία περιμένει τα δεδομένα (αναπαράσταση του γράφου) να είναι στην μορφή DIMACS (πλήρης περιγραφή υπάρχει στα σχόλια της διαδικασίας από τον Steven Crocker). Ο γράφος όμως που δίνεται από τις προηγούμενες διαδικασίες σαν είσοδος είναι ένας πίνακας γειτνίασης. Τα δεδομένα διαβάζονται στην υποδιαδικασία getgraph, όπου και έγιναν οι αλλαγές, δηλαδή αντί τα δεδομένα να διαβάζονται από το πληκτρολόγιο ή από αρχείο εισάγονται στις αντίστοιχες μεταβλητές από τον πίνακα γειτνίασης cmp_matrix.

6.4 Διαδικασίες βελτίωσης του MaxRPC

Rpc_color_mates procedure

Λειτουργία: Υλοποιεί το βήμα 3 του ALG_3 της ενότητας 3.4.

Δήλωση:

```
procedure rpc_color_mates(var m_match: partners;
                          var c_matrix: path_array;
                          var clr_usage_matrix: color_array;
                          c_n_paths,n_colors: integer);
```

Σχόλια: Η διαδικασία αναχρωματίζει τα μονοπάτια της αλυσίδας c_matrix, χρησιμοποιώντας το μέγιστο ταίριασμα m_match και τον πίνακα καταγραφής της χρήσης των χρωμάτων clr_usage_matrix. Οι μεταβλητές c_n_paths, n_colors είναι αντίστοιχα οι αριθμοί των μονοπατιών της αλυσίδας, και του αριθμού των διαθέσιμων χρωμάτων.

Color_lonely procedure

Λειτουργία: Υλοποιεί την βελτίωση της παραγράφου 3.4.1.

Δήλωση:

```
procedure color_lonely(var c_matrix: path_array;
                       cmp_matrix: compatibility_array;
                       n_paths,n_colors: integer;
                       var clr_usg_matrix: color_array);
```

Σχόλια: Η διαδικασία αναχρωματίζει τα μονοπάτια της αλυσίδας c_matrix, χρησιμοποιώντας τον πίνακα συμβατότητας αιτήσεων cmp_matrix και τον πίνακα καταγραφής της χρήσης των χρωμάτων clr_usage_matrix. Οι μεταβλητές n_paths, n_colors είναι αντίστοιχα οι αριθμοί των μονοπατιών της αλυσίδας, και του αριθμού των διαθέσιμων χρωμάτων.

Κεφάλαιο 7

Συμπεράσματα – ανοιχτά ερωτήματα

Στην εργασία αυτή παρουσιάστηκαν αλγόριθμοι για τα προβλήματα χρωματισμού μονοπατιών MaxPC και MaxRPC. Για καθένα από τα προβλήματα παρουσιάστηκαν και αντίστοιχες βελτιώσεις. Στη συνέχεια οι αλγόριθμοι και οι βελτιώσεις τους υλοποιήθηκαν στο προγραμματιστικό περιβάλλον της PASCAL. Τέλος παρουσιάστηκαν τα αποτελέσματα μιας σειράς πειραμάτων που προέκυψαν από την εκτέλεση των διαφόρων υλοποιήσεων και είχαν σαν σκοπό την σύγκριση των αλγορίθμων και των βελτιώσεών τους καθώς και την καλύτερη κατανόησή τους.

Κατά την υλοποίηση των αλγορίθμων προέκυψαν δυσκολίες που οφείλονται στο προγραμματιστικό περιβάλλον της PASCAL και έχουν σχέση με τη διαχείριση της διαθέσιμης μνήμης. Το πρόβλημα είχε σχέση με τον αριθμό των μονοπατιών που ουσιαστικά είναι το μέγεθος της εισόδου. Όταν οι αλγόριθμοι για το κάθε πρόβλημα εκτελούνταν ξεχωριστά τότε δεν υπήρχε πρόβλημα με τον αριθμό των μονοπατιών. Όταν όμως εκτελούνταν ταυτόχρονα και οι τρεις μορφές του αλγόριθμου κατά τη δημιουργία των πειραμάτων ο αριθμός των μονοπατιών έπρεπε να περιοριστεί διαφορετικά εμφανίζονταν μηνύματα λάθους από το περιβάλλόν της PASCAL που είχαν σχέση με τη διαθέσιμη μνήμη. Το πρόβλημα ήταν πιο έντονο κατά την εκτέλεση των πειραμάτων για το MaxRPC γιατί οι δομές δεδομένων που χρησιμοποιούνται από τους αλγόριθμους είναι πιο περίπλοκες από αυτές του MaxPC.

Μετά τη μελέτη των αλγορίθμων, την υλοποίησή τους και την παρουσίαση των αποτελεσμάτων των πειραμάτων που έγιναν μπορούν να γίνουν κάποιες προτάσεις για μελλοντική εργασία. Μια βελτίωση στην υλοποίηση των αλγορίθμων θα ήταν η εξής: Σε κάθε αλγόριθμο να γίνεται μέτρηση του πραγματικού χρόνου εκτέλεσής του. Με αυτό τον τρόπο μπορεί να γίνει μια άλλου είδους σύγκριση των αλγορίθμων. Έτσι ένας αλγόριθμος που εκτελείται πιο γρήγορα μπορεί να προτιμηθεί από έναν άλλον που εκτελείται πιο αργά ακόμα και αν δεν πετυχαίνει καλύτερη λύση αν βέβαια το ζητούμενο είναι η πιο άμεση απόκριση.

Επίσης αν το ζητούμενο είναι η πιο απλή υλοποίηση και όχι η καλύτερη λύση τότε όπως προκύπτει και από το πείραμα Δ για το MaxPC αρκεί να χρησιμοποιηθεί ο απλός αλγόριθμος για την αλυσίδα που πετυχαίνει παράγοντα προσέγγισης ½ . Ο συγκεκριμένος αλγόριθμος μάλιστα για μικρό αριθμό χρωμάτων σε σχέση με τον αριθμό των μονοπατιών πετυχαίνει σχεδόν τα ίδια αποτελέσματα με τους άλλους αλγόριθμους.

Μια άλλη βελτίωση θα ήταν η επιλογή της ακμής που χωρίζει τον δακτύλιο στην αλυσίδα και στα υπόλοιπα μονοπάτια να γίνεται με συγκεκριμένο τρόπο. Στους αλγόριθμους που υλοποιήθηκαν η ακμή αυτή είναι για λόγους ευκολίας η τελευταία ακμή του δακτυλίου, ενώ εύκολα η επιλογή της μπορεί να γίνει τυχαία. Για να πάρουμε καλύτερα αποτελέσματα θα μπορούσαμε να επιλέγουμε διαδοχικά όλες τις ακμές, να εκτελούσαμε τους

αλγόριθμους και τελικά να επιλέγονταν η καλύτερη λύση που προκύπτει από την επιλογή συγκεκριμένης ακμής.

Ένα ανοιχτό ερώτημα που προκύπτει μετά την παρουσίαση των αλγορίθμων και των βελτιώσεών τους είναι το αν θα μπορούσε να βελτιωθεί ο παράγοντας προσέγγισης. Οι αλγόριθμοι όπως έχει αναφερθεί είναι 2/3 προσεγγιστικοί, ίσως όμως με βελτιώσεις στην υλοποίηση να πετυχαίνουν πρακτικά καλύτερα αποτελέσματα. Μια τέτοια όμως απόδειξη ΄θα ήταν δύσκολο να γίνει στα πλαίσια της συγκεκριμένης εργασίας.

Ένα ακόμα πιο ενδιαφέρον ερώτημα είναι το αν οι βελτιώσεις μπορούν να πετύχουν καλύτερη συμπεριφορά στη μέση περίπτωση, ή ακόμα περισσότερο αν μπορεί να γίνει θεωρητική εκτίμηση του παράγοντα προσέγγισης στη μέση περίπτωση. Πολύ μεγάλο ενδιαφέρον επίσης θα είχε η ανάλυση του αλγόριθμου της αλυσίδας (που είναι πολύ εύκολα υλοποιήσιμος) στη μέση περίπτωση. Το πρόβλημα που αμέσως προκύπτει είναι το πώς μπορεί να οριστεί η μέση περίπτωση. Μια ιδέα θα ήταν οι δακτύλιοι με αριθμό κόμβων μέσα σε ένα διάστημα να έχουν την ίδια πιθανότητα εμφάνισης. Ο αριθμός των μονοπατιών να βρίσκεται και αυτός μέσα σε κάποιο διάστημα που να έχει σχέση με τον αριθμό των κόμβων του δακτυλίου ενώ ο αριθμός των χρωμάτων να είναι μικρός σε σχέση με τον αριθμό των μονοπατιών πράγμα το οποίο συμβαίνει και στην πράξη. Η μελέτη του παραπάνω προβλήματος και η αντίστοιχη ανάλυση και απόδειξη αφήνεται σαν ιδέα για μελλοντική εργασία.

Παράρτημα Α

Κώδικας MaxPC και επιπλέον διαδικασίες βελτίωσης

```
program max_pc;
(*
   2/3 aproximation algorithm for MaxPC
*)

uses crt;

const MaxVertex = 150;  (* max number of nodes to handle *)
      unmatched = 0;
      max_path = MaxVertex;   (* max number of paths in the ring *)

type path = record
              pstart: integer;
              pend: integer;
              pcolor: integer;
              pstatus: integer; (*1=currently checking 2=checked
active
                                  0=checked overlap   -3=initialize*)
              in_set: integer;  (*1=set C  2=set Q*)
            end;

     vertex = 1 .. MaxVertex;

     path_array = array[vertex] of path;
     color_array= array[vertex] of integer;

     queue_array = array[vertex] of integer;

     PartnerName = unmatched .. MaxVertex;
     partners = array [vertex] of PartnerName;

     PtrToNode = ^node;
     node = record
              id: vertex;
              next: PtrToNode
            end;
     graph = record
               size: integer; (* number of vertices *)
               AdjLists: array [vertex] of PtrToNode
                 (* array for random access to list heads *)
             end;

var ring_path_matrix: path_array;
    chain_path_matrix: path_array;
    q_path_matrix: path_array;
    color_usage_matrix: color_array;
    size_of_ring,
    num_of_colors,num_of_paths,
    num_chain_paths,num_q_paths,
    share_edge: integer;
    color_matrix: color_array;
    cardinality1,cardinality2: integer; (* cardinality of ALG1,Alg2
*)
```

```
    max_cardinality: integer;

    bipartiteH: graph;   (* bipartite graph H from sets C, Q *)
    queue: queue_array ; (* queue for BFS*)
    qhead,qtail: integer; (* head,tail of queue *)
    i,test: integer;
    max_match: partners;   (* output of max match *)



procedure CreateQueue(var head,tail: integer);
begin
   head:=1;
   tail:=1;
end; (* createQueue *)


procedure Enqueue(var q: queue_array;
                  var head,tail: integer; x: vertex);
begin
   q[tail]:=x;
   if tail=MaxVertex then
      begin
         tail:=1;
         (*writeln('queue overflow');*)
      end
   else
      tail:=tail+1;
end; (* Enqueue *)


function IsEmptyQueue(head,tail: integer): boolean;
begin
   if tail=head then
      IsEmptyQueue:=true
   else IsEmptyQueue:=false;
end; (* IsEmptyQueue *)


procedure Dequeue(var q: queue_array;
                  var head,tail: integer; var x: vertex);
begin
   x:= q[head];
   if head=MaxVertex then
      begin
         (*writeln('queue underflow');*)
         head:=1
      end
   else
      head:=head+1;
end; (* Dequeue *)


procedure Match(var G: graph;
                lside,rside: integer;
                (*var error: boolean; (* Set if graph not bipartite.
*)
                var mate: partners;
                var mqueue: queue_array; var mqhead,mqtail: integer);
  (* This procedure finds a maximum matching in bipartite graphs
     in O(sqrt(V)*E) time, recording the result in mate[ ]. *)
```

```
  (* Program assumes the existence of a queue package with operations
       procedure CreateQueue, function IsEmptyQueue: boolean,
       procedure Enqueue(v: vertex), and procedure Dequeue(var v:
vertex). *)

  label 1, 99;
  (* The breadth-first search assigns level numbers to left-side
vertices;
       unmatched left-side vertices are at level zero.
     Matched right-side vertices are not assigned level numbers,
       because we simply pass through them.
     Unmatched right-side vertices are assigned the level number
       that their mates would be assigned if they were matched. *)
  const Left = -1; (* values that are not valid levels *)
        Right = -2;
  type LevelsAndSides = array [vertex] of integer;
  var level: LevelsAndSides;
      dfs_array: array [vertex] of PtrToNode;
        (* For right-side vertices, dfs records how far along we are
           in the adjacency list during the depth-first search. *)
      i,CurrentLevel: integer;
      RightSideEnqueued: boolean;
      v, w: vertex;
      p: PtrToNode;


  procedure DFS(v: vertex; l: integer);
    (* Uses O(E) time and O(V) storage (actually the length of
       the shortest augmenting path) for the implicit stack. *)
    var dummy: boolean;
    procedure RecDFS(v: vertex; l: integer; var success: boolean);
      label 99;
      begin
        (* v is a right-side vertex. *)
        while dfs_array[v] <> nil do
          begin
            if level[dfs_array[v]^.id] = l - 1
              then begin
                     if l = 1
                        then success := true
                        else RecDFS(mate[dfs_array[v]^.id],l-
1,success);
                     if success
                        then begin
                               (* Augment.  Switching the status of
the
                                  unmatched edges also switches the
status
                                  of the matched edges. *)
                               mate[dfs_array[v]^.id] := v; mate[v]
:= dfs_array[v]^.id;
                               (* Prevent reuse of this left-side
vertex. *)
                               level[dfs_array[v]^.id] := Left;
                               goto 99
                             end
                   end;
            dfs_array[v] := dfs_array[v]^.next
          end;
  99: end; (* RecDFS *)
```

```
    begin (* DFS *)
      dummy := false;
      RecDFS(v,l,dummy);
      level[v] := Right
    end; (* DFS *)

  begin (* Match *)
    CreateQueue(mqhead,mqtail);
    (* Define the bipartite structure of the graph (see Exercise
4.8),
       exiting with an error if the graph is not bipartite. *)
    (*ggggggggggg... *)
    (* Form the empty or other initial matching. *)
    for v := 1 to G.size do mate[v] := unmatched;

    (* flag left and right nodes of graph *)
    for i:=1 to lside do
       level[i]:= Left;
    for i:=lside+1 to lside+rside do
       level[i]:= Right;


    while true do
      begin
        (* Find a maximal set of augmenting paths for the next
shortest length.
           Exit loop if no augmenting path exists. *)
        (* O(V) initialization *)
        for v := 1 to G.size do
          case level[v] of
            Left:  if mate[v] = unmatched then
                      begin
                         level[v] := 0;
                         Enqueue(mqueue,mqhead,mqtail,v)
                      end;
            Right: dfs_array[v] := G.AdjLists[v]
          end;
        CurrentLevel := 1; (* the level about to be enqueued *)
        RightSideEnqueued := false;

        (* Expand the breadth-first search to the next level.  As
soon as
           an unmatched right-side vertex has been discovered, we
stop
           enqueueing left-side vertices.  We continue the
exploration of
           the current level, but only look for additional unmatched
right-side
           vertices.  When we advance to the next level, we might
have to
           discard some left-side vertices that were enqueued before
the first
           unmatched right-side vertex was discovered. *)

        while not IsEmptyQueue(mqhead,mqtail) do
          begin
            Dequeue(mqueue,mqhead,mqtail,v);
            if level[v] = CurrentLevel
              then if not RightSideEnqueued
                     then CurrentLevel := CurrentLevel + 1
```

```
                    else begin (* Discard useless left-side
vertices. *)
                           while mate[v] <> unmatched do
Dequeue(mqueue,mqhead,mqtail,v);
                           goto 1
                         end;
            p := G.AdjLists[v];
            while p <> nil do
              begin
                if mate[p^.id] = unmatched
                  then begin
                         if level[p^.id] = Right (* not already in
the queue *)
                           then begin
                                  level[p^.id] := CurrentLevel;
                                  RightSideEnqueued := true;
                                  Enqueue(mqueue,mqhead,mqtail,p^.id)
                                end
                       end
                  else begin
                         w := mate[p^.id];
                         if (level[w] = Left (* unseen *)) and
                            not RightSideEnqueued (* and still
relevant *)
                           then begin
                                  level[w] := CurrentLevel;
                                  Enqueue(mqueue,mqhead,mqtail,w)
                                end
                       end;
                p := p^.next
              end
          end;
        (* Maximum matching has been found. *)
        goto 99;

     1: (* Start the depth-first search from the right-side vertices,
           searching for a maximal set of shortest augmenting paths.
*)
        (* One unmatched right-side vertex has already been dequeued.
*)
        DFS(v,CurrentLevel);
        while not IsEmptyQueue(mqhead,mqtail) do
          begin
            Dequeue(mqueue,mqhead,mqtail,v);
            DFS(v,CurrentLevel)
          end;

        (* Reset level[ ] for the next iteration. *)
        for v := 1 to G.size do
          if level[v] >= 0
            then level[v] := Left
      end;
99:
  end; (* Match *)


procedure read_ring(var p_matrix: path_array; var ring_size: integer;
                    var n_paths: integer;     var n_colors: integer);
(* reads the ring, the paths, the number of colors on the ring *)
var i,j: integer;
    double_path: boolean;
```

```
begin
   randomize;
   repeat
      ring_size:= random(MaxVertex div 3)+1;
   until ring_size>2;
   n_paths:= random(ring_size*3)+1;
   n_colors:= random(ring_size)+1;

   for i:=1 to n_paths do
   begin
      p_matrix[i].pstart:=0;
      p_matrix[i].pend:=0;
   end;

   writeln('ring_size=',ring_size,' n_paths=',n_paths,'
n_colors=',n_colors);

   for i:=1 to n_paths do
   begin
      repeat
         double_path:= false;
         p_matrix[i].pstart:= random(ring_size)+1;
         repeat
            p_matrix[i].pend:= random(ring_size)+1;
         until p_matrix[i].pstart<>p_matrix[i].pend;

         for j:=1 to n_paths do
         begin
            if (p_matrix[j].pstart=p_matrix[i].pstart) and
               (p_matrix[j].pend=p_matrix[i].pend) and (i<>j) then
            begin
               double_path:=true;
            end;
         end;
      until double_path=false;
   end;


   for i:=1 to n_paths do
   begin
      p_matrix[i].pstatus:=-3;
      p_matrix[i].pcolor:=-3;
      p_matrix[i].in_set:=2;   (*initialize path in set Q *)
   end;

end; (*procedure read_ring*)


procedure create_chain(var p_matrix: path_array; var
c_matrix:path_array;
                       ring_size: integer;   n_paths: integer;
                       var chain_paths: integer;
                       var random_e: integer);
var i: integer;
    e_start,e_end: integer;

begin (* procedure create_chain *)
   randomize;
   random_e:=random(ring_size-1)+1;
```

```
   random_e:=ring_size;
   writeln('random_edge=',random_e);
   e_start:=random_e;
   if random_e<ring_size then
      e_end:=random_e+1
   else
      e_end:=1;

   for i:=1 to n_paths do
   begin
      if p_matrix[i].pstart<p_matrix[i].pend then
      begin
         if (p_matrix[i].pend<=e_start) or
(p_matrix[i].pstart>=e_end) then
         begin
            p_matrix[i].in_set:=1;
         end;
      end;

      if p_matrix[i].pstart>p_matrix[i].pend then
      begin
         if random_e<>ring_size then
         begin
            if (p_matrix[i].pend<=e_start) and
(p_matrix[i].pstart>=e_end) then
            begin
               p_matrix[i].in_set:=1;
            end;
         end;
      end;
   end; (* for *)

   chain_paths:=0;
   for i:= 1 to n_paths do
   begin
      if p_matrix[i].in_set=1 then
      begin
         chain_paths:=chain_paths+1;
         c_matrix[chain_paths].pstart:= p_matrix[i].pstart;
         c_matrix[chain_paths].pend:=   p_matrix[i].pend;
      end;
   end;

   for i:=1 to chain_paths do
   begin
      c_matrix[i].pstatus:=-3;
      c_matrix[i].pcolor:=-3
   end;
end; (* procedure create_chain *)


procedure create_setQ(var p_matrix: path_array; (* all paths *)
                      var q_matrix: path_array; (* paths in set Q *)
                      ring_size:integer; n_paths: integer;
                      var q_n_paths:integer;
                      var clr_matrix: color_array;
                      n_colors: integer;
                      share_e: integer);
var i,k: integer;

begin (* procedure create_setq *)
```

```
   q_n_paths:=0;

   for i:= 1 to n_paths do
   begin
      if p_matrix[i].in_set<> 1 then
      begin
         q_n_paths:= q_n_paths+1;
         q_matrix[q_n_paths].pstart:= p_matrix[i].pstart;
         q_matrix[q_n_paths].pend:=   p_matrix[i].pend;
         q_matrix[q_n_paths].pstatus:= p_matrix[i].pstatus;
         q_matrix[q_n_paths].pcolor:=   p_matrix[i].pcolor;

      end;
   end;

   writeln('qset   status,color');
   for i:=1 to q_n_paths do
   begin
      writeln(q_matrix[i].pstart:2,' ',q_matrix[i].pend:2,'   ',
              q_matrix[i].pstatus,' ',q_matrix[i].pcolor);
   end;
end; (* procedure create_setq *)


procedure chain_coloring(var c_matrix: path_array;
                         var clr_matrix: color_array;
                         var clr_usage_matrix: color_array;
                         c_length,n_paths,n_colors: integer);

var i,j,k,
    rmax,rpath,
    overlaps,
    color: integer;

begin (* Procedure chain_coloring *)
   overlaps:=0;
   for i:=1 to c_length do   (* c_length=size_of_ring *)
   begin
      for j:=1 to n_paths do  (* number of chain paths *)
      begin
         if (c_matrix[j].pend=i) and (c_matrix[j].pstatus=1) then
         begin
            overlaps:=overlaps-1;
            c_matrix[j].pstatus:=2;
         end;
         if c_matrix[j].pstart=i then
         begin
            overlaps:=overlaps+1;
            c_matrix[j].pstatus:=1;
         end
      end; (* for j *)

      if overlaps>n_colors then
      begin
      (*writeln('overlap');*)
         for j:=1 to overlaps-n_colors do
         begin
            rmax:=-1;
            for k:=1 to n_paths do
            begin
               if (c_matrix[k].pend>rmax) and
```

```
                  (c_matrix[k].pstatus=1) then
               begin
                  rmax:= c_matrix[k].pend;
                  rpath:= k;
               end;
            end;
            c_matrix[rpath].pstatus:=0;
         end;
         overlaps:=n_colors;
      end; (* if overlaps>num_of_colors*)
   end; (*for i*)


   (*initialize all colors availiable*)
   for i:=1 to n_colors do
   begin
      clr_matrix[i]:=1;
      clr_usage_matrix[i]:=0;
   end;

   (* color active paths *)
   for i:=1 to c_length do  (* c_length= ring_size *)
   (* free colors *)
   begin
      for j:=1 to n_paths do   (* n_paths= num of chain paths*)
      begin
         if (c_matrix[j].pend=i) and (c_matrix[j].pstatus=2) then
         begin
            (* free color *)
            clr_matrix[c_matrix[j].pcolor]:=1;
         end;
      end;

      (* color next path *)
      for j:=1 to n_paths do
      begin
         if (c_matrix[j].pstart=i) and (c_matrix[j].pstatus=2) then
         begin
            k:=1;
            while clr_matrix[k]=0 do
            begin
               k:=k+1;
            end;
            c_matrix[j].pcolor:=k;
            clr_matrix[k]:=0; (* flag color as used *)
            clr_usage_matrix[k]:=clr_usage_matrix[k]+1;
         end;
      end;
   end;

   writeln('chain  status,color');
   for i:=1 to n_paths do
   begin
      writeln(c_matrix[i].pstart:2,' ',c_matrix[i].pend:2,'   ',
              c_matrix[i].pstatus,' ',c_matrix[i].pcolor);
   end;
   readln;
end; (* procedure chain_coloring *)


procedure use_free_colors(var clr_usage_matrix: color_array;
```

```
                          var path_matrix: path_array;
                          n_paths,n_colors: integer);
(* use free colors to color paths in arbitrary way *)
var i,j,uncolored: integer;

begin
   for i:=1 to n_colors do
   begin
      if clr_usage_matrix[i]=0 then
      begin
         for j:=1 to n_paths do
         begin
            if path_matrix[j].pcolor<=0 then
            begin
               uncolored:=j;
            end;
         end;
         path_matrix[uncolored].pcolor:=i;
         clr_usage_matrix[i]:=1;
      end;
   end;
end;   (* procedure use_free_colors *)


procedure create_bipartiteH(var g: graph;
                                c_matrix,q_matrix: path_array;
                                c_n_paths,q_n_paths: integer);
(* creates graph H from set C and Q *)

var p,q: PtrToNode;
    i,j,graph_size: integer;

begin
   graph_size:= c_n_paths+q_n_paths;
   g.size:= graph_size;

   for i:=1 to graph_size do
   begin
      g.AdjLists[i]:=nil;
   end;

   for i:=1 to c_n_paths do
   begin
      for j:=1 to q_n_paths do
      begin
         if (c_matrix[i].pstart<c_matrix[i].pend) and
            (q_matrix[j].pstart<q_matrix[j].pend) then
         begin
            if (c_matrix[i].pend<=q_matrix[j].pstart) or
               (q_matrix[j].pend<=c_matrix[i].pstart) then
            begin
               (*add edge*)
               new(p);
               p^.id:=c_n_paths+j;
               p^.next:=g.AdjLists[i];
               g.AdjLists[i]:=p;

               new(q);
               q^.id:=i;
               q^.next:=g.AdjLists[c_n_paths+j];
               g.AdjLists[c_n_paths+j]:=q;
```

```
            end;
         end;

         if (c_matrix[i].pstart<c_matrix[i].pend) and
            (q_matrix[j].pend<q_matrix[j].pstart) then
         begin
            if (c_matrix[i].pstart>=q_matrix[j].pend) and
               (c_matrix[i].pend<=q_matrix[j].pstart) then
            begin
               (* add edge *)
               new(p);
               p^.id:=c_n_paths+j;
               p^.next:=g.AdjLists[i];
               g.AdjLists[i]:=p;

               new(q);
               q^.id:=i;
               q^.next:=g.AdjLists[c_n_paths+j];
               g.AdjLists[c_n_paths+j]:=q;
            end;
         end;

         if (q_matrix[j].pstart<q_matrix[j].pend) and
            (c_matrix[i].pend<c_matrix[i].pstart) then
         begin
            if (q_matrix[j].pstart>=c_matrix[i].pend) and
               (q_matrix[j].pend<=c_matrix[i].pstart) then
            begin
               (* add edge*)
               new(p);
               p^.id:=c_n_paths+j;
               p^.next:=g.AdjLists[i];
               g.AdjLists[i]:=p;

               new(q);
               q^.id:=i;
               q^.next:=g.AdjLists[c_n_paths+j];
               g.AdjLists[c_n_paths+j]:=q;
            end;
         end;

      end; (* for j  *)
   end; (* for i *)
end; (*procedure create_bipartiteH *)


procedure print_list(p: PtrToNode);
(* prints the nodes of the list p *)
begin
   if p=nil then write('0');
   while p<>nil do
   begin
      write(p^.id,' ');
      p:= p^.next;
   end;
end; (* procedure print_list *)


procedure print_graph(g: graph);
var i: integer;
begin
```

```
   readln;
   writeln('bipartiteH:');
   for i:=1 to g.size do
   begin
      print_list(g.AdjLists[i]);
      writeln;
   end;
end; (* procedure print graph *)

begin (***************** main ********************)
clrscr;
begin

read_ring(ring_path_matrix,size_of_ring,num_of_paths,num_of_colors);

   create_chain(ring_path_matrix,chain_path_matrix,

size_of_ring,num_of_paths,num_chain_paths,share_edge);

   chain_coloring(chain_path_matrix,color_matrix,color_usage_matrix,
                  size_of_ring,
                  num_chain_paths,num_of_colors);

   create_setQ(ring_path_matrix,q_path_matrix,
               size_of_ring,num_of_paths,num_q_paths,
               color_matrix,num_of_colors,share_edge);

   for i:=1 to num_chain_paths do
      ring_path_matrix[i]:= chain_path_matrix[i];
   for i:=1 to num_q_paths do
      ring_path_matrix[i+num_chain_paths]:= q_path_matrix[i];

   use_free_colors(color_usage_matrix,ring_path_matrix,
                   num_of_paths,num_of_colors);

   writeln;
   writeln ('final coloring');
   for i:=1 to num_of_paths do

writeln(ring_path_matrix[i].pstart:3,ring_path_matrix[i].pend:3,
              ring_path_matrix[i].pcolor:3);

   create_bipartiteH(bipartiteH,chain_path_matrix,q_path_matrix,
                     num_chain_paths,num_q_paths);

   match(bipartiteH,num_chain_paths,num_q_paths,
         max_match,queue,qhead,qtail);

   (* print Alg1 cardinality *)
   cardinality1:=0;
   for i:= 1 to num_of_paths do
   begin
      if ring_path_matrix[i].pcolor>0 then
         cardinality1:=cardinality1+1;
   end;
   writeln('cardinality Alg1: ',cardinality1);

   print_graph(bipartiteH);

   writeln ('Max matching: ');
   for i:=1 to bipartiteH.size do
```

```
   begin
      write(max_match[i],' ');
   end;

   (* print Alg2 cardinality *)
   cardinality2:=0;
   for i:=1 to num_chain_paths do
   begin
      if max_match[i]<>0 then
         cardinality2:=cardinality2+1;
   end;
   if num_of_colors<cardinality2 then
      cardinality2:=num_of_colors;

   writeln;
   cardinality2:=2*cardinality2;
   writeln('cardinality Alg2= ',cardinality2);

   writeln;
   if cardinality1>=cardinality2 then
      max_cardinality:= cardinality1
   else
      max_cardinality:= cardinality2;
   writeln('max cardinality= ',max_cardinality);

   readln;
end.  (* main *)



procedure uncolor_lonely_paths(var c_matrix: path_array;
                               var clr_usage_matrix: color_array;
                               n_paths,n_colors: integer);
var i,j: integer;

begin
   for i:= 1 to n_colors do
   begin
      if clr_usage_matrix[i]=1 then
      begin
         clr_usage_matrix[i]:=0;  (* free color i *)
         for j:=1 to n_paths do
         begin
            if c_matrix[j].pcolor=i then
            begin
               (* uncolor lonely path *)
               c_matrix[j].pcolor:=-3;
               (*writeln('uncolor ');*)
            end
         end (* for j*)
      end
   end; (* for i*)

   writeln('chain  status,color');
   for i:=1 to n_paths do
   begin
      writeln(c_matrix[i].pstart:2,' ',c_matrix[i].pend:2,'   ',
              c_matrix[i].pstatus,' ',c_matrix[i].pcolor);
   end;

   writeln('color usage');
```

```
   for i:=1 to n_colors do
      write(clr_usage_matrix[i],' ');
   readln;
end; (* procedure uncolor_lonely_paths *)


procedure color_mates(m_match: partners;
                      var c_matrix: path_array;
                      var q_matrix: path_array;
                      var clr_usage_matrix: color_array;
                      c_n_paths,q_n_paths,n_colors: integer);
(* step4 of alg2 computer networks *)
var i,j,p,q,curent_color: integer;
    mates,free_colors: integer;
    found_mate: boolean;

begin
   mates:=0;
   for i:=1 to c_n_paths do
   begin
      if m_match[i]>0 then
         mates:=mates+1;
   end;

   free_colors:=0;
   for i:=1 to n_colors do
   begin
      if clr_usage_matrix[i]=0 then
         free_colors:=free_colors+1;
   end;

   while (mates>0) and (free_colors>0) do
   begin
      i:=1; found_mate:=false;
      while (i<=c_n_paths) and (found_mate=false) do
      begin
         if m_match[i]>0 then
         begin
            p:=i;
            q:=m_match[i];
            m_match[q]:=0;
            m_match[p]:=0;
            mates:=mates-1;
            found_mate:=true;
         end
         else
           i:=i+1;
      end;

      if c_matrix[p].pcolor>0 then
      begin
         curent_color:=c_matrix[p].pcolor;
         c_matrix[p].pcolor:=-3;

         clr_usage_matrix[curent_color]:=
clr_usage_matrix[curent_color]-1;
         if clr_usage_matrix[curent_color]=1 then
         begin
            clr_usage_matrix[curent_color]:=0;
            free_colors:=free_colors+1;
            (* uncolor lonely path *)
```

```
            for j:=1 to c_n_paths do
            begin
               if c_matrix[j].pcolor=curent_color then
                  c_matrix[j].pcolor:=-3;
            end;
            for j:=1 to q_n_paths do
            begin
               if q_matrix[j].pcolor=curent_color then
                  q_matrix[j].pcolor:=-3;
            end;
         end;
      end;

      i:=1;
      while (clr_usage_matrix[i]>0) and (i<=n_colors) do
      begin
         i:=i+1;
      end;
      if i<=n_colors then
      begin
         clr_usage_matrix[i]:=2;
         c_matrix[p].pcolor:=i;
         q_matrix[q-c_n_paths].pcolor:=i;
         free_colors:=free_colors-1;
      end;
   end;   (* while *)
end;   (* procedure color_mates *)


procedure use_free_colors(var clr_usage_matrix: color_array;
                          var path_matrix: path_array;
                          n_paths,n_colors: integer);
(* use free colors to color paths in arbitrary way *)
var i,j,uncolored: integer;

begin
   for i:=1 to n_colors do
   begin
      if clr_usage_matrix[i]=0 then
      begin
         for j:=1 to n_paths do
         begin
            if path_matrix[j].pcolor<=0 then
            begin
               uncolored:=j;
            end;
         end;
         path_matrix[uncolored].pcolor:=i;
         clr_usage_matrix[i]:=1;
      end;
   end;
end;   (* procedure use_free_colors *)



procedure custom_match (var mate: partners;
                        c_matrix,q_matrix: path_array;
                        c_n_paths,q_n_paths: integer);
var i,j: integer;
    match_ok: boolean; (* true if c,q paths do not overlap *)
    not_found_mate: boolean; (* true if no mate found *)
```

```
    free_q: array[vertex] of integer; (* =0 not free to be used in
match *)
begin
   match_ok:=false;
   for i:=1 to MaxVertex do mate[i]:=0;
   for i:=1 to q_n_paths do free_q[i]:=1;

   for i:=1 to c_n_paths do
   begin
      not_found_mate:=true;
      j:=1;
      while (not_found_mate=true) and (j<=q_n_paths) do
      begin
         match_ok:=false;
         if (c_matrix[i].pstart<c_matrix[i].pend) and
            (q_matrix[j].pstart<q_matrix[j].pend) then
         begin
            if (c_matrix[i].pend<=q_matrix[j].pstart) or
               (q_matrix[j].pend<=c_matrix[i].pstart) then
            begin
               match_ok:=true;
            end;
         end;

         if (c_matrix[i].pstart<c_matrix[i].pend) and
            (q_matrix[j].pend<q_matrix[j].pstart) then
         begin
            if (c_matrix[i].pstart>=q_matrix[j].pend) and
               (c_matrix[i].pend<=q_matrix[j].pstart) then
            begin
               match_ok:=true;
            end;
         end;

         if (q_matrix[j].pstart<q_matrix[j].pend) and
            (c_matrix[i].pend<c_matrix[i].pstart) then
         begin
            if (q_matrix[j].pstart>=c_matrix[i].pend) and
               (q_matrix[j].pend<=c_matrix[i].pstart) then
            begin
               match_ok:=true;
            end;
         end;

         if (match_ok=true) and (free_q[j]=1) then
         begin
            mate[i]:=c_n_paths+j;
            mate[c_n_paths+j]:=i;
            free_q[j]:=0;
            not_found_mate:=false;
         end;
         j:=j+1;
      end; (* while j  *)
   end; (* for i *)
end; (* procedure custom_match *)



procedure bubble_sort(var matrix: path_array; n: integer);
var i,j: integer;
    temp: path;
```

```
begin
   for i:=2 to n do
   begin
      for j:= n downto i do
      begin
         if matrix[j-1].pstart>matrix[j].pstart then
         begin
            temp:= matrix[j-1];
            matrix[j-1]:=matrix[j];
            matrix[j]:= temp;
         end
      end
   end
end; (* procedure bubble_sort *)
```

Παράρτημα Β

Κώδικας MaxRPC και επιπλέον διαδικασίες βελτίωσης

```
program max_rpc;
(* 2/3 aproximation algorithm for MaxRPC *)

uses crt;

const MaxVertex = 90;  (* max number of nodes to handle *)

type path = record
              pstart: integer;
              pend: integer;
              pcolor: integer;
              pstatus: integer; (*1=currently checking 2=checked
active
                                   0=checked overlap   -3=initialize*)
            end;

     request = record
                 rstart: integer;
                 rend: integer;

                 rcolor: integer;
               end;

     vertex = 1 .. MaxVertex;

     path_array = array[vertex] of path;
     request_array = array[vertex] of request;
     color_array= array[vertex] of integer;
     compatibility_array= array[vertex,vertex] of integer;
(*adjacensy matrix*)


var ring_request_matrix: request_array;
    chain_path_matrix: path_array;
    compatibility_matrix: compatibility_array;
    size_of_ring,
    num_of_colors,
    num_chain_paths,
    share_edge: integer;
    num_of_requests: integer;
    num_of_edges: integer; (* num of edges in compatibility graph *)
    num_of_matched: integer; (* mun of matched nodes *)
    color_matrix: color_array;
    color_usage_matrix: color_array;
    ggmax_color_used: integer; (* max num of color used in chain
coloring*)
    cardinality1,cardinality2: integer; (* cardinality of ALG1,Alg2
*)
    max_cardinality: integer;
    i,test: integer;
```

```
procedure read_ring(var r_matrix: request_array; var ring_size:
integer;
                    var n_requests: integer;     var n_colors:
integer);
(* reads the ring, the requests, the number of colors on the ring *)
var i,j: integer;
    double_request: boolean;

begin
   randomize;
   repeat
      ring_size:= random(MaxVertex div 3)+1;
   until ring_size>2;
   n_requests:= random(ring_size*3)+1;
   n_colors:= random(ring_size)+1;

   for i:=1 to n_requests do
   begin
      r_matrix[i].rstart:=0;
      r_matrix[i].rend:=0;
   end;

   writeln('ring_size=',ring_size,' n_requests=',n_requests,'
n_colors=',n_colors);

   for i:=1 to n_requests do
   begin
      repeat
         double_request:= false;
         r_matrix[i].rstart:= random(ring_size-1)+1;
         repeat
            r_matrix[i].rend:= random(ring_size)+1;
         until r_matrix[i].rstart<r_matrix[i].rend;

         for j:=1 to n_requests do
         begin
            if (r_matrix[j].rstart=r_matrix[i].rstart) and
               (r_matrix[j].rend=r_matrix[i].rend) and (i<>j) then
            begin
               double_request:=false;(*true;*)
            end;
         end;
      until double_request=false;
   end;

   for i:=1 to n_requests do
   begin
      r_matrix[i].rcolor:=-3;
   end;
end; (*procedure read_ring*)


procedure create_chain(var r_matrix: request_array; var
c_matrix:path_array;
                       ring_size: integer;   n_requests: integer;
                       var chain_paths: integer;
                       var random_e: integer);
var i: integer;
    e_start,e_end: integer;

begin (* procedure create_chain *)
```

```
   randomize;
   random_e:=random(ring_size-1)+1;

   random_e:=ring_size;

   e_start:=random_e;
   if random_e<ring_size then
      e_end:=random_e+1
   else
      e_end:=1;

   for i:=1 to n_requests do
   begin
      c_matrix[i].pstart:= r_matrix[i].rstart;
      c_matrix[i].pend:= r_matrix[i].rend;
   end; (* for *)

   chain_paths:=n_requests;

   for i:=1 to chain_paths do
   begin
      c_matrix[i].pstatus:=-3;
      c_matrix[i].pcolor:=-3
   end;
end; (* procedure create_chain *)


procedure create_compatibilityG(r_matrix: request_array;
                                var cmp_matrix: compatibility_array;
                                n_requests: integer;
                                var n_edges: integer);
var i,j: integer;

begin (* procedure create_compatibilityG *)
   for i:= 1 to n_requests do
   begin
      for j:=1 to n_requests do
      begin
         cmp_matrix[i,j]:=0;
      end;
   end;

   for i:= 1 to n_requests do
   begin
      for j:=1 to n_requests do
      begin
          if ((r_matrix[i].rstart=r_matrix[j].rstart) or
             (r_matrix[i].rstart=r_matrix[j].rend)   or
             (r_matrix[i].rend= r_matrix[j].rstart)  or
             (r_matrix[i].rend= r_matrix[j].rend)) and (i<>j) then
             begin
                cmp_matrix[i,j]:=1;
             end;

          if (r_matrix[i].rend<r_matrix[j].rstart) or
             (r_matrix[i].rstart>r_matrix[j].rend) then
          begin
             cmp_matrix[i,j]:=1;
          end;

          if (r_matrix[i].rstart<r_matrix[j].rstart) and
```

```
            (r_matrix[i].rend>r_matrix[j].rend) then
         begin
            cmp_matrix[i,j]:=1;
         end;

         if (r_matrix[i].rstart>r_matrix[j].rstart) and
            (r_matrix[i].rend<r_matrix[j].rend) then
         begin
            cmp_matrix[i,j]:=1;
         end;
      end; (* for j*)
   end; (*for i*)

   n_edges:=0;
   for i:=1 to n_requests do
   begin
      for j:=1 to n_requests do
      begin
         if cmp_matrix[i,j]=1 then
            n_edges:=n_edges+1;
      end;
   end;
   n_edges:=n_edges div 2;
   writeln('n_edges= ',n_edges);

   writeln('compatibility matrix');
   for i:=1 to n_requests do
   begin
      for j:=1 to n_requests do
      begin
         write(cmp_matrix[i,j]);
      end;
      writeln;
   end;
end; (* procedure create_compatibilityG *)


procedure chain_coloring(var c_matrix: path_array;
                         var clr_matrix: color_array;
                         var clr_usage_matrix: color_array;
                         c_length,n_paths,n_colors: integer);

var i,j,k,
    rmax,rpath,
    overlaps,
    color: integer;

begin (* Procedure chain_coloring *)
   overlaps:=0;
   for i:=1 to c_length do   (* c_length=size_of_ring *)
   begin
      for j:=1 to n_paths do  (* number of chain paths *)
      begin
         if (c_matrix[j].pend=i) and (c_matrix[j].pstatus=1) then
         begin
            overlaps:=overlaps-1;
            c_matrix[j].pstatus:=2;
         end;
         if c_matrix[j].pstart=i then
         begin
            overlaps:=overlaps+1;
```

```
         c_matrix[j].pstatus:=1;
      end
   end; (* for j *)

   if overlaps>n_colors then
   begin
   (*writeln('overlap');*)
      for j:=1 to overlaps-n_colors do
      begin
         rmax:=-1;
         for k:=1 to n_paths do
         begin
            if (c_matrix[k].pend>rmax) and
               (c_matrix[k].pstatus=1) then
            begin
               rmax:= c_matrix[k].pend;
               rpath:= k;
            end;
         end;
         c_matrix[rpath].pstatus:=0;
      end;
      overlaps:=n_colors;
   end; (* if overlaps>num_of_colors*)
end; (*for i*)


(*initialize all colors availiable
  usage of each color = 0 *)
for i:=1 to n_colors do
begin
   clr_matrix[i]:=1;

   clr_usage_matrix[i]:=0;
end;

(* color active paths *)
for i:=1 to c_length do  (* c_length= ring_size *)
(* free color when path ends *)
begin
   for j:=1 to n_paths do   (* n_paths= num of chain paths*)
   begin
      if (c_matrix[j].pend=i) and (c_matrix[j].pstatus=2) then
      begin
         (* free color *)
         clr_matrix[c_matrix[j].pcolor]:=1;
      end;
   end;

   (* color next path *)
   for j:=1 to n_paths do
   begin

      if (c_matrix[j].pstart=i) and (c_matrix[j].pstatus=2) then
      begin
         k:=1;
         while clr_matrix[k]=0 do
         begin
            k:=k+1;
         end;
         c_matrix[j].pcolor:=k;
         clr_matrix[k]:=0; (* flag color as used *)
```

```
            clr_usage_matrix[k]:=clr_usage_matrix[k]+1;
          end;
        end;
     end;
end; (* procedure chain_coloring *)


procedure general_match(cmp_matrix: compatibility_array;
                        n_nodes: integer;
                        n_edges: integer;
                        var fmcount: integer);

(*  This file contains an implementation of                 *)
(*  The Micali-Vazirani Maximum Cardinality Matching Algorithm    *)
(*                                                  *)
(*  coded by Steven Crocker                             *)
(*  present address: Department of Computer Science,        *)
(*               University of Chicago                  *)
(*                                                  *)
(*    email address: crocker@cs.uchicago.edu                *)
(*                                                  *)
(*                                                  *)
(*  The program expects its input file to be in the DIMACS  *)
(*  Implementation Challenge format for graphs represented by   *)
(*  edge adjacencies.  E.g.  A triangle could be give as below: *)
(*                                                  *)
(*  c  Comments begin with a c in column 1,                 *)
(*  c  The problem statement, begins with a p in column 1 and     *)
(*  c  has the word "edge" just after the p to specify edge       *)
(*  c  format.    Following that, the next two columns contain    *)
(*  c  the number of verices and the number of edges.       *)
(*  c  Edges are given 1 per line, each line begining with an e   *)
(*  c  in column 1 and containing vertex 1 and vertex 2 in the  *)
(*  c  next two columns.  Any addtional information on an edge  *)
(*  c  line is ignored.                                 *)
(*  c  Here is the triangle:                            *)
(*  p  edge 3 3                                         *)
(*  e  1 2                                          *)
(*  e  1 3                                          *)
(*  e  2 3                                          *)
(*                                                  *)
(*                                                  *)
(*  Input is passed to the program through standard input.  *)
(*  The following two lines would run the matching program on     *)
(*  a file named triangle containing the graph given above. *)
(*                                                  *)
(*  pc cardmatch.p -o cardmatch                             *)
(*  cardmatch < triangle                                *)
(*                                                  *)
(*                                                  *)
(*  Details of how the algorithm works can be found in          *)
(*  S. Micali and V.V. Vazirani, "An O(sqrt(|V|)*|E|) algorithm   *)
(*  for finding maximum matching in general graphs" in Proc.      *)
(*  21st Annual IEEE Symposium on Foundations of Computer   *)
(*  Science, 1980, pp. 17-27.                           *)
(*                                                  *)


const showmatch = true;             (* True => print out the mate of
each vertex in the matching. *)
```

```
      showtime  = false;            (* True => print out the
cardinality of the matching and
                                        the number of milliseconds
taken to find it. *)


var
    numvert, numedges : integer;        (* numvert, numedges are the
number of vertices and edges *)
    (*fmcount  : integer;*)   (* counts of the number of matched
vertices from the two algorithms *)
    starttime,endtime,fasttime : integer;              (* timing
quantities *)

    grfilename : packed array [1..100] of char;


procedure fastmatch;

type  vertexptr = ^vertex;
      edgeptr = ^edge;
      blossptr = ^bloss;
      levelptr = ^level;

      side = (left,right,none);

      vertex = record
            num   :     integer;          (* vertex number (for
debugging purposes) *)
            elist       :     edgeptr;          (* header of edge
list *)
            mate  :     vertexptr;        (* mate *)
            medge :     edgeptr;          (* matched edge *)
            prednx,                       (* header of forward list
of pred *)
            predpr      :     edgeptr;          (* header of reverse
list of pred *)
            predcnt :   integer;          (* predecessor count *)
            blossom     :     blossptr;         (* blossom *)
            markside :  side;             (* side of blossom vertex
is on in ddfs *)
            markcall :  integer;          (* call # of blossaug this
vtx visited on *)
            llside,
            rlside      :     vertexptr;        (* dbl link list for
vertices with same side marking *)
            bstar :     vertexptr;        (* base-star structure *)
            visblos :   blossptr;         (* blossom searching when
visited *)
            visside :   side;             (* side of blossom
searching when visited *)
            present :   boolean;
            evlev :     integer;          (* even level *)
            nxeven      :     vertexptr;        (* next vertex with
this even level *)
            odlev :     integer;          (* odd level *)
            nxodd :     vertexptr;        (* next vertex with this
odd level *)
            level :     integer;
            anom  :     edgeptr;          (* header of anomalies list
for this vertex *)
```

```
            lastused :  edgeptr;
            lastvisited :     edgeptr;
            parvert :   vertexptr;        (* father vertex pointer
from the ddfs *)
            paredge :   edgeptr;          (* father edge pointer from
the ddfs *)
            pathvert :  vertexptr;        (* a vertex on the path *)
            pathedge :  edgeptr;          (* an edge on the path *)
            estack      :     vertexptr;        (* stack pointer for
topological erase *)
            next  :     vertexptr;        (* next vertex in the graph
*)
            end;

      edge = record
            num   :     integer;          (* number of this edge *)
            v1, v2      :     vertexptr;        (* two vertices
defining the edge *)
            nx1, nx2 :  edgeptr;          (* next edges incident to
v1 and v2, respectively *)
            prednx      :     edgeptr;          (* prednedge *)
            predpr :    edgeptr;          (* predledge *)
            predvtx :   vertexptr;        (* predvertex *)
            bridnx      :     edgeptr;          (* brid *) (* next
bridge on the same level *)
            visblos :   blossptr;         (* vis[].blosm *)
            visside :   side;             (* vis[].side *)
            used  :     boolean;
            isabridge : boolean;          (* true if this edge is a
bridge *)
            nxanom      :     edgeptr;
            next  :     edgeptr;          (* next edge in the graph
*)
            end;

      bloss = record
            base  :     vertexptr;        (* defblos[ ,base] *) (*
base vertex of blossom *)
            peake :     edgeptr;          (* defblos[ ,peake] *) (*
peak edge of blossom *)
            end;

      level = record
            levnum      :     integer;          (* the level number
*)
            vertices :  vertexptr;        (* head of vertex list of
vertices on this level *)
            firstbridge :     edgeptr;          (* first of the
bridges on this level *)
            lastbridge :      edgeptr;          (* last of the
bridges on this level *)
            next  :     levelptr;         (* next level *)
            end;


var vmin : vertexptr;
    emin : edgeptr;
    inf  : integer;
    goodgraph : boolean;
```

```
procedure getgraph (var vmin : vertexptr;
                var emin : edgeptr;
                var goodgraph: boolean);
                  (*  get the next graph specification *)
                        (*  and initialize the adjacency lists of
edges *)



                             (* on entry -- the file 'graph' contains
the vertex pairs
                                  defining the edges of the graph
                                on exit  -- the vertex and edge nodes
have been built and
                                  initialized to the adjacency lists
of the vertices,
                                  where each edge has one record and
is on the adjacency
                                  list of two vertices.
                             *)

label 1;

type  blockptr = ^block;
      block = record
            verts : array [0..511] of vertexptr;
            next : blockptr;
            end;

var vt1,vt2  :  vertexptr;
    nuedge  : edgeptr;
    nuv1, nuv2 : integer;
    cnt,i,j : integer;
    c,d1,d2,d3,d4 : char;
    firstblk, lastblk, vblk : blockptr;


function vrec (v : integer) : vertexptr;

var i, vblknum, vloc : integer;
    blk : blockptr;

begin
   vblknum := v div 512;
   vloc := v mod 512;
   blk := firstblk;
   for i := 1 to vblknum do blk := blk^.next;
   vrec := blk^.verts[vloc];
end;


procedure setupverts ( numverts : integer; var vmin : vertexptr);

var         max,                    (* the number of vertices in the
last block *)
      i,                      (* the number of the current block *)
      j,                      (* the location of the current vertex
in the current block *)
      final,                        (* the number of the last block
needed *)
```

```
      loc0num : integer;            (* the vertex number of the
vertex in location 0 of a block *)
      lastvert : vertexptr;

begin
   final := numverts div 512;
   loc0num := 0;
   firstblk := nil;
   lastblk := nil;
   lastvert := nil;
   for i := 0 to final do begin
      new(vblk);
      if firstblk = nil then begin
         firstblk := vblk;
         lastblk := vblk;
       lastblk^.next := nil;
         end
      else begin
       lastblk^.next := vblk;
       lastblk := vblk;
       end;
      if i = final then max := numverts mod 512
      else max := 511;
      with lastblk^ do begin
       next := nil;
       for j := 0 to max do begin
          new(verts[j]);
            if lastvert = nil then lastvert := verts[j];
          lastvert^.next := verts[j];
          lastvert := verts[j];
          with verts[j]^ do begin
             elist := nil;
             num := loc0num + j;
             mate := nil;
             medge := nil;
             next := nil;
             end;
           end;
       loc0num := loc0num + 512;
       end;
      end;
   vmin := firstblk^.verts[1];
end;


procedure skipwhite(var c : char);

begin
   while not eoln and (c in [' ','  ']) do
      read(c);
end;


function skipjunk(var c : char; d : char): boolean;
            (* Skip ahead to a line where a character the same as d
is the first in the line. *)
            (* Assumes c is the first character of the first line to
be checked. *)

begin
   skipwhite(c);
```

```
   while (c <> d) and not eof do begin
      readln;
      read(c);
      skipwhite(c);
      end;
   skipjunk := not eof;
end;


begin   (* procedure getgraph *)
   goodgraph := false;
   (*read(c);*) c:='p';
   if skipjunk(c,'p') then begin
      (*read(d1);*) d1:='e';
      skipwhite(d1);
      if d1 <> 'e' then begin
          writeln(' Error in input format: only edge adjacency graph
specifications accepted.');
          goto 1;
          end;
      (*read(d2,d3,d4);*) d2:='d';d3:='g';d4:='e';
      (*read(numvert,numedges);*)
      numvert:= n_nodes;
      numedges:= n_edges;
      setupverts(numvert,vmin);
      emin := nil;
      (*readln;*)   (* no more to read from the problem line *)
      cnt := 0;

(*
      while (cnt < numedges) do begin
       while eoln do readln;
         if eof then begin
            writeln(' Error in graph spec, not enough edges.');
            goto 1;
            end;
         read(c); c:='e';
         if not skipjunk(c,'e') then begin
            writeln(' Error in graph specification.');
            goto 1;
            end
         else begin
            cnt := cnt+1;
            new(nuedge);
*)
      for i:=1 to n_nodes do
      begin
         for j:=1 to n_nodes do
         begin
            if (j>i) and (cmp_matrix[i,j]=1) then
            begin
               (*readln(nuv1,nuv2);*)
               cnt:=cnt+1;
               new(nuedge);
               nuv1:=i; nuv2:=j;
               vt1 := vrec(nuv1);
               vt2 := vrec(nuv2);
               with nuedge^ do begin
                  num := cnt;
                  v1 := vt1;
                  v2 := vt2;
```

```
                  nx1:= vt1^.elist;             (* insert ce into
lvx's and rvx's edge lists *)
                  nx2:= vt2^.elist;
                  vt1^.elist := nuedge;
                  vt2^.elist := nuedge;
                  end;
               nuedge^.next := emin;
               emin := nuedge;
            end;
         end;     (* for j *)
      end; (* for i *)

      goodgraph := true;
      end;
1:
end;   (* procedure getgraph *)


procedure printmatch;
                                    (* print the current matching *)

const tab = '     ';
var i : vertexptr;

begin
   fmcount := 0;
   i := vmin;
   while i <> nil do begin
      if (i^.mate <> nil) then
         fmcount := fmcount + 1;
      i := i^.next;
      end;
   write ('s  ',(fmcount div 2):1);

   if showtime then
         write(tab,' time=',fasttime:1,' msec., |V|=',numvert:1,',
|E|=',numedges:1);

   writeln;

   if showmatch then begin
      i := vmin;
      while i <> nil do begin
         if (i^.mate <> nil) then
         if i^.mate^.num > i^.num then writeln ('m  ',i^.num:3,'
',i^.mate^.num:3);
         i := i^.next;
         end;
      end;
end;


procedure match (vmin : vertexptr; emin : edgeptr);

(* find a maximal matching for a graph in O(E*sqrt(V)) time *)

type
        updn = -1..+1;
        evodd = (even, odd);
var
        serlev,temp : integer;
```

```
        strtside : array [side] of vertexptr;
      augmented,augoccur : boolean;
      nx, ce : edgeptr;
      vinit : vertexptr;
        thiscallba : integer;                    (* unique integer
identifying this ba call  *)
        cv,cu : vertexptr;
        predve : edgeptr;
      maxlev : levelptr;                 (* maxlev is highest nonempty
level *)
      dummybloss : blossptr;
      activelevel, nextlevel : levelptr;         (* current and next
level of the bfs *)


function findlevel(clevnum : integer):levelptr;
            (* find the record of level number clevnum *)

var i : integer;
    maxlevnum : integer;
    plevptr : levelptr;
    nlev : levelptr;

begin
   maxlevnum := maxlev^.levnum;
   if clevnum > maxlevnum then begin
      plevptr := maxlev;
      for i := maxlevnum + 1 to clevnum do begin
       new(nlev);
       plevptr^.next := nlev;
       plevptr := nlev;
         nlev^.levnum := i;
       nlev^.vertices := nil;
       nlev^.firstbridge := nil;
       nlev^.lastbridge := nil;
         nlev^.next := nil;
       end;
      maxlev := nlev;      (* set new maximum level to search to. *)
      findlevel := nlev;
      end
   else begin
      nlev := activelevel;
      while nlev^.levnum < clevnum do nlev := nlev^.next;
      findlevel := nlev;
      end;
end;


procedure setlev (eo:evodd; vert:vertexptr; lev:levelptr);
                           (* insert vert on the vertex list for
level lev *)
                           (*  on entry -- the evlev/odlev of vert
is infinite
                               on exit  -- the evlev/odlev of vert =
lev^.levnum,
                                           vert is in the
evlev/odlev list for level lev,
                                           vert^.level is the
minimum of the even/odd levels of vert
                           *)
```

```
begin
   if lev = nil then
      lev := findlevel(activelevel^.levnum + 1);
   with vert^ do
      with lev^ do begin
         case eo of
            even : begin
                      evlev := levnum;          (* vert^.evlev :=
lev^.levnum *)
                      nxeven := vertices;      (* vert^.nxeven :=
lev^.vertices *)
                      vertices := vert;         (* lev^.vertices :=
vert *)
                   end;
            odd :  begin
                      odlev := levnum;          (* vert^.odlev :=
lev^.levnum *)
                      nxodd := vertices;  (* vert^.nxodd :=
clev^.vertices *)
                      vertices := vert;         (* lev^.vertices :=
vert; *)
                   end;
            end;
         if levnum < level then level := levnum;     (* change level
of vertex *)
         end;
end;


procedure addbridge (lev : levelptr; edg:edgeptr);
                           (* add edg to bridges(lev) *)

                           (* on entry -- brid(edg) is undefined
                              on exit  -- brid(edg) = lev,
                                          edg is in the list for
bridges on level lev,
                                          maxlev is the highest
nonempty level
                           *)


begin
   if lev^.firstbridge = nil then
      lev^.firstbridge := edg
   else
      lev^.lastbridge^.bridnx := edg;
   lev^.lastbridge := edg;
   edg^.bridnx := nil;
   edg^.isabridge := true;
end;


procedure addpred (l:vertexptr; e: edgeptr);
                                (* add e to the predecessor list of l
*)
                                (* on exit -- e is on the predecessor
list of l,
                                              the predecessor count
of l has been incremented by 1 *)
```

```
var  fst : edgeptr;

begin
   fst := l^.prednx;
   e^.prednx := fst;
   if fst <> nil then
      fst^.predpr :=e;
   e^.predvtx := l;
   l^.prednx := e;
   e^.predpr := nil;
   l^.predcnt:= l^.predcnt + 1;
end;


procedure addanom (l:vertexptr; e:edgeptr);
                            (* add e to the anomalies list of l *)
                            (* on exit -- e is on the anomalies list
of l *)

begin
   e^.nxanom := l^.anom;
   l^.anom := e;
end;


function otherv (v : vertexptr; e : edgeptr) : vertexptr ;
                                      (*  return the vertex linked to
v by e *)
begin
   if e^.v1 = v then otherv := e^.v2
      else otherv := e^.v1;
end;


procedure blossaug (w1,w2 : vertexptr; var augmented : boolean ;
peakedg : edgeptr);
                                   (* do a double depth first search
to find and augment a path or define a blossom *)
                                   (* on entry -- there exists two
paths from at least one free vertex
                                                  to the vertices w1
and w2, neither path includes the
 other vertex.
                                      on exit  -- an augmenting path
has been found through w1-peakedg-w2
                                                  and has been
augmented and erased
                                                        or
                                                  no augmenting path
of length <= 2*serlev + 1 exists
                                                  through w1-
peakedge-w2 and a new blossom has been
                                                  formed with w1 and
w2 as the peaks
                                                        or
                                                  the paths from free
vertices to w1 and w2 are not
                                                  vertex disjoint
with an augmenting path of length 2*serlev+1
                                                  previously
augmented and erased
```

```
                                        *)

var
    u,ur,ul,vl,vr,w,barrier : vertexptr;
    dcv, bstardcv : vertexptr;
    anomedg,ranedg,lanedg : edgeptr;
    s : side;
    temp : integer;
    bridlevel : levelptr;
    blossomfound : boolean;
    nubloss : blossptr;


procedure connectpath (v1,v2 : vertexptr; eg : edgeptr; dir : updn);

                           (* connect the path list between vertices
v1 and v2 *)
                           (* on entry -- v1 and v2 are vertices, eg
is an edge incident to one of the vertices
                              on exit  -- if direction is positive
then the list in path goes from v2 to v1
                                                otherwise it goes from v1 to
v2
                           *)

begin
   if dir = -1 then begin
        v1^.pathvert := v2;
        v1^.pathedge := eg;
        end
      else begin
        v2^.pathvert := v1;
        v2^.pathedge := eg;
        end;
end;


procedure findpath (high,low : vertexptr; b : blossptr;direction :
updn; sidesearch:side); forward;


procedure openbl (ent,base : vertexptr; direct : updn);
                          (* find an alternating path from ent to base
filling in all vertices in the blossom of ent *)
                          (* on entry -- ent is the only vertex in the
path which is in
                                             the blossom of ent
                             on exit  -- path has been expanded to
contain an alternating path in
                                              the graph from ent to the
base of ent^.blossom
                          *)

var b : blossptr;
    pe : edgeptr;
    pl,pr : vertexptr;

begin
   b := ent^.blossom;
   if ent^.level mod 2 = 0 then
      findpath (ent,base,b,direct,ent^.markside)
```

```
     else begin
      pe := b^.peake;
      pl := pe^.v1;
      pr := pe^.v2;

      if ent^.markside = left then begin
         findpath (pl,ent,b,(direct*-1),left);
         findpath (pr,base,b,direct,right);
         connectpath (pl,pr,pe,-1*direct);
         end
        else begin
         findpath (pr,ent,b,(direct*-1),right);
         findpath (pl,base,b,direct,left);
         connectpath (pr,pl,pe,-1*direct);
         end;
      end;
end;


function bastar (v : vertexptr):vertexptr;
                        (* find the vertex which is the base* of v *)
                        (* on entry -- v is a vertex in the graph
                           on exit  -- bastar is the base* of v,
                                       the path traversed has been
compressed
                        *)

var n, vnxt : vertexptr;

begin
   n := v;
   while n^.bstar <> nil do n := n^.bstar;
   while v <> n do begin
      vnxt := v^.bstar;
      v^.bstar := n;
      v := vnxt;
      end;
   bastar := n;
end;


procedure findpath;

(* procedure findpath (high,low : vertexptr; b : blossptr;direction :
updn; sidesearch:side); *)


                         (* Build a path in the graph from (high to
low/low to high)
                            when directon is (+1/-1) using the father
tree developed in blossaug *)
                         (* on entry -- high is an ancestor of low in
the father tree
                            on exit  -- path contains an ordered list
of the vertices in
                                        the path from high to low, if
direction = +1 on entry then the list is
                                        from high to low, otherwise
it is from low to high
                         *)
```

```
var e : edgeptr;
    u,v : vertexptr;
    falsebloss : boolean;

begin
   if high <> low then begin
     falsebloss := false;
     v := high;
     u := v;
     predve := high^.prednx;
     while (u <> low) do begin
        if predve = nil then predve := v^.prednx;
                                            (* Find next unvisited
predecessor edge. *)
        while (predve^.visblos = b ) and (predve^.prednx <> nil) do
           predve := predve^.prednx;
        v^.lastvisited := predve;    (* Update start of unvisited
predecessors. *)
                                             (* Act on the kind of
predecessor edge found. *)
        if predve^.visblos = b then begin         (* No more
unvisited predecessor edges. *)
           v := v^.parvert;                       (* So backup. *)
           predve:=v^.lastvisited;
           end
        else if ((v^.blossom <> b) and falsebloss) then begin
                                                      (* Wrong path
down blossom, reverse and erase. *)
           v := v^.parvert;
           predve:=v^.lastvisited;
           end
        else begin
           falsebloss := false;
           if v^.blossom = b then begin         (* Choose an
unvisited predecessor edge. *)
              predve^.visblos := b;                  (* Mark this
edge "visited" by the b call to findpath. *)
              predve^.visside := sidesearch;
              u := otherv (v,predve);
              end
           else u := v^.blossom^.base;             (* Pass over the
blossom, for now. *)
           if u <> low then
              if ((u^.visside = sidesearch) and (u^.visblos = b))
(* u is 'visited' *)
                 or (u^.level <= low^.level )
(* or missed low mark => wrong path *)
                 or ((u^.blossom = b) and (u^.markside <>
sidesearch))  (* or u is on the other half of path

through the blossom *)
                 then begin        (* This edge fails, perhaps
because of a wrong blossom, so try again. *)
                 falsebloss := true;
                 end
              else begin           (* Found a good prospect.  Step
down it and continue the dfs of findpath. *)
                 u^.visblos := b;
                 u^.visside := sidesearch;
                 u^.parvert := v;
                 u^.paredge := predve;
```

```
                v := u;
                predve:=v^.lastvisited;
                end;  (* else *)
          end (* else *)
       end; (* while*)

                           (* path found *)
   u^.parvert := v;
   u^.paredge := predve;
   e := u^.paredge;
   while v <> high do begin               (* u = low *)
      connectpath (u,v,e,direction);
      u := v;
      v := v^.parvert;
      e := u^.paredge;
      end;
   connectpath (u,v,e,direction);
   if direction = +1 then begin           (* initialize for traversal
of path which opens blossoms  *)
      v := high;
      u := high^.pathvert;
      end
   else begin
      u := low;
      v := low^.pathvert;
      end;
   while not (((direction=+1)and(v=low))or((direction=-
1)and(u=high))) do begin
                                           (* traverse the path in
correct direction. *)
      if v^.blossom <> b then openbl (v,u,direction);
      if direction = +1 then begin        (* step down the path *)
         v := u;
         u := u^.pathvert;
         end
      else begin
         u := v;
         v := v^.pathvert;
         end;
      end; (* while *)
   end; (* if high <> low *)
end;  (* findpath *)


procedure augment (lv,rv : vertexptr);
                           (*  Augment the path between lv and rv.
*)
                           (* on entry -- path contains an augmenting
path from lv to rv
                              on exit  -- the matching along this
path has been augmented
                           *)

var mv : vertexptr;
    e : edgeptr;

begin
   repeat                             (*  Traverse the augmenting path
and *)
      mv := lv^.pathvert;          (*  change the matching.  *)
      e := lv^.pathedge;
```

```
      lv^.mate :=mv;
      lv^.medge :=e;
      mv^.mate :=lv;
      mv^.medge :=e;
      lv := mv^.pathvert;
   until mv = rv;
end;


procedure toperase (vl,vr : vertexptr);
                        (* do a topological erase on the path
between vl and vr *)
                        (* on entry -- path contains a path from vl
to vr,  all vertices and edges
                                    on this path are present
                           on exit  -- all vertices on the path are
not present, all edges incident
                                    to a vertex which is no
longer present are not present,
                                    all vertices which have no
present edges incident to them are not present
                        *)

var erasetop : vertexptr;
    es : boolean;
    vert : vertexptr;


procedure pushvert ( v : vertexptr);
                         (* push v onto the stack of vertices to be
erased *)

begin
   v^.estack := erasetop;
   erasetop := v;
end;


procedure popvert (var v : vertexptr; var empty : boolean);
                        (* pop v from the stack of vertices to be
erased *)

begin
   if erasetop = nil then empty := true
   else begin
      v := erasetop;
      empty := false;
      erasetop := erasetop^.estack;
      v^.estack := nil;
      end;
end;


procedure erase ( v : vertexptr );
                      (* erase vertex v and edges incident to it *)
                      (* on entry -- v is a vertex in the graph
                         on exit  -- v and all its incident edges
have been erased, neighboring vertices
                                    which are now unreachable from a
free vertex through breadth
```

```
                                    first search paths are stacked
for erasing, edges incident to v which
                                    are in the bfs graph are erased
from the predecessor lists of
                                    the neighboring vertices
                     *)

var e : edgeptr;
    u : vertexptr;

begin
   v^.present := false;                                        (* erase
v *)
   e := v^.elist;
   while e <> nil do                                (* for each present
edge incident to v *)
    with e^ do begin
      if (predvtx <> nil) and (predvtx <> v) then begin  (* if bfs
goes from v thru e to u *)
      if predvtx^.present then begin
            u := predvtx;
            u^.predcnt := u^.predcnt -1;          (* decrement
topological count of u *)
            if (u^.predcnt = 0) then pushvert(u);          (* if u
unreachable by any present

bfs path, stack u for erasing *)
                        (* remove e from pred graph *)
          if prednx <> nil then prednx^.predpr := predpr;
            if predpr <> nil then predpr^.prednx := prednx
          else u^.prednx := prednx;
            end;
       end;
      if v = v1 then e := nx1 else e := nx2;          (* get next
edge incident to v *)
      end;
end;

(* procedure toperase (vl,vr : vertexptr)  *)

begin
   erasetop := nil;
   while vl <> vr do begin
      erase (vl);
      vl := vl^.pathvert;
      end;
   erase (vl);
   popvert (vert,es);
   while not es do begin
      erase (vert);
      popvert (vert,es);
      end;
end;


procedure mrkside (vrtx: vertexptr; sd : side);
                            (* mark the vrtx as belonging to the left
or right dfs tree *)
                            (* on entry -- vrtx does not belong to
either dfs tree
```

```
                                on exit  -- vrtx is marked sd and is
inserted in the list of vertices on that tree
                              *)

var i: vertexptr;

begin
   i := strtside [sd];
   vrtx^.rlside := i;
   vrtx^.markside := sd;
   strtside [sd] := vrtx;
   vrtx^.llside := nil;
   vrtx^.markcall := thiscallba;
   if i <> nil then i^.llside := vrtx;
end;


procedure rmvmark (vrtx : vertexptr );
                               (* remove vrtx from the bfs tree it is
in *)
                               (* on entry -- vrtx belongs to one of
the bfs trees
                                  on exit  -- vrtx in is neither dfs
tree, the call to blossaug it was marked
                                                 is undefined
                               *)

var  l,r : vertexptr;

begin
 if vrtx <> nil then begin
   l := vrtx^.llside;
   r := vrtx^.rlside;
   if l = nil then strtside [vrtx^.markside] := r
      else l^.rlside := r;
   if r <> nil then r^.llside := l;
   vrtx^.markcall := -1;
   end;
end;


procedure pushleft(var vl,vr : vertexptr; var lanedg : edgeptr);

begin
      if lanedg = nil then lanedg := vl^.prednx;
      while (lanedg^.used) and (lanedg^.prednx <> nil) do
         lanedg := lanedg^.prednx;                (* Find next unused
ancestor edge of vl. *)
      vl^.lastused := lanedg;
      if lanedg^.used then          (* vl has no more unused ancestor
edges *)
         if vl^.parvert = nil then blossomfound := true  (* create a
blossom *)
         else begin
               vl := vl^.parvert;
               lanedg:=vl^.lastused;
               end
      else begin          (* vl has unused ancestor edges *)
            lanedg^.used := true;
            ul := otherv (vl,lanedg);
```

```
            ul := bastar (ul);

            if ul^.markcall <> thiscallba then begin
               mrkside (ul,left);
               ul^.parvert:= vl;
               ul^.paredge := lanedg;
               vl := ul;
               lanedg:=vl^.lastused;
               end
            else if (ul = barrier) or (ul <> vr) then begin
               if (ul = barrier) and (ul=vr) and (dcv=nil) then dcv
:= ul;
               end
            else begin
               rmvmark (ul);
               mrkside (ul,left);
               vr := vr^.parvert;
               ranedg:=vr^.lastused;
               ul^.parvert := vl;
               ul^.paredge := lanedg;
               vl := ul;
               lanedg:=vl^.lastused;
               dcv := ul;
               end
            end
end;


procedure pushright(var vl,vr : vertexptr; var ranedg : edgeptr);
begin
      if ranedg = nil then ranedg := vr^.prednx;
      while (ranedg^.used) and (ranedg^.prednx <> nil) do
         ranedg := ranedg^.prednx;                    (* Find next
unused ancestor edge of vr. *)
      vr^.lastused := ranedg;
      if ranedg^.used then           (* vr has no more unused
ancestor edges *)
         if vr = barrier then begin
            if vl^.parvert = nil then blossomfound := true
            else begin
               vr := dcv;
               ranedg:=vr^.lastused;
               barrier := dcv;
               rmvmark (vr);            (* remove vr's old mark *)
               mrkside (vr,right);
               vl := vl^.parvert;
               lanedg:=vl^.lastused;
               end;
            end
         else begin
            vr := vr^.parvert;
            ranedg:=vr^.lastused;
            end
      else begin                      (* vr has unused ancestor edges
*)
         ranedg^.used := true;
         ur := otherv (vr,ranedg);
         ur := bastar (ur);
         if ur^.markcall <> thiscallba then begin           (* ur is
unmarked *)
            mrkside (ur,right);
```

```
            ur^.parvert := vr;
            ur^.paredge := ranedg;
            vr := ur;
            ranedg:=vr^.lastused;
            end
           else begin                              (* u is marked *)
            if ur = vl then dcv := ur;
            end
         end
end;


(* procedure blossaug (w1,w2 : vertexptr; var augmented : boolean ;
peakedge : edgeptr);   *)
                              (* Either define a new blossom or
discover that w1 and w2 are in the *)
                         (* same blossom or find an augmenting path.
*)
                              (* On return-- augmented = true if an
augmenting path was found   *)

begin
   blossomfound := false;
   augmented := false;
   if w1^.present and w2^.present  (* Neither w1 nor w2 were on an
augmenting path previously found *)
      and
      (bastar(w1) <> bastar(w2)) then begin       (* w1 and w2 are
not already in the same blossom *)

                           (* initialize for double depth first search
*)
      vl := bastar(w1);
      vr := bastar(w2);
      lanedg := vl^.lastused;
      ranedg := vr^.lastused;
      thiscallba := thiscallba + 1;
      strtside [left]:=nil;
      strtside [right] := nil;
      mrkside (vl,left);
      mrkside (vr,right);
      vl^.parvert := nil;
      dcv := nil;
      barrier := vr;

      while not blossomfound and not augmented do              (*
ddfs search  *)
         if (vl^.mate = nil) and (vr^.mate = nil) then begin   (*  vl
and vr are free *)
            findpath (w1,vl,nil,-1,left);
            findpath (w2,vr,nil,+1,right);
            connectpath (w1,w2,peakedg,-1);
            augment (vl,vr);
            toperase (vl,vr);
            augmented := true;
            end
         else if vl^.level >= vr^.level then    (* vl,vr not both
free and level(vl) <= level(vr) *)
            pushleft(vl,vr,lanedg)
         else            (*  vl,vr not both free and level (vr) >
level(vl) *)
```

```
            pushright(vl,vr,ranedg);

      if blossomfound then begin
         rmvmark(dcv);                              (* dcv is not in
the blossom *)
       new(nubloss);                             (* create a new blossom
*)
         nubloss^.base := dcv;           (* assign it a base vertex
*)
         nubloss^.peake := peakedg;      (*       and a peak edge *)
         bstardcv := bastar(dcv);

         for s := left to right do begin         (* put each vertex
marked left or right during *)
                                                 (*      this call in
the new blossom *)
            u := strtside [s];
            while u<>nil do begin
               u^.blossom := nubloss;
               u^.bstar := bstardcv;           (* add vertices of
blossom to base* tree *)
               case u^.level mod 2 of
                  0: setlev (odd,u,findlevel(2*serlev + 1 -
u^.evlev));
                  1:  begin
                      setlev (even,u,findlevel(2*serlev + 1 -
u^.odlev));
                      anomedg := u^.anom;
                      while anomedg <> nil do begin
                         w := otherv (u,anomedg);
                         temp := (u^.evlev + w^.evlev) div 2;
                    bridlevel := findlevel(temp);
                         addbridge (bridlevel,anomedg);
                         anomedg^.used := true;
                         anomedg := anomedg^.nxanom;
                         end;              (*while*)
                      end;           (* 1: *)
               end;         (* cases *)
               u := u^.rlside;
               end;            (* while u <> nil *)
            end;               (* for s := ... *)
         end;                  (* if blossomfound ... *)
      end;                     (* if w1^.present... *)
end;        (* procedure blossaug *)


procedure bfsinitvertices;

var cv : vertexptr;
    zerolev : levelptr;
begin
   zerolev :=findlevel(0);
   cv := vmin;
   while cv <> nil do begin
      with cv^ do begin
         bstar := nil;
         level := inf;
         lastused := nil;
         lastvisited := nil;
         present := true;
         odlev := inf;
```

```
         nxodd := nil;
         nxeven := nil;
         blossom := nil;
         predpr := nil;
         prednx := nil;
         anom := nil;
         visside := none;
         visblos := dummybloss;
         if mate = nil then begin
            setlev (even,cv,zerolev);
            predcnt := 1;
            end
         else begin
            evlev := inf;
            predcnt := 0;
            end;
       end;
      cv := cv^.next;
      end;
end;


procedure bfsinitedges;

var ce : edgeptr;
begin
   ce := emin;
   while ce <> nil do begin
      with ce^ do begin
         visblos := dummybloss;
         visside := none;
         used := false;
         isabridge := false;
         bridnx := nil;
         predvtx := nil;
         end;
      ce := ce^.next;
      end;
end;


procedure bfsinitsearch;

begin
   serlev := -1;
   new(maxlev);
   maxlev^.levnum := -1;
   maxlev^.firstbridge := nil;     (* bridges[maxlev] set to empty
set *)
   maxlev^.lastbridge := nil;
   maxlev^.vertices := nil;
   activelevel := maxlev;
   augmented := false;
end;


procedure matchinit;

begin
   inf := numvert + 1;
   new(dummybloss);
```

```
   thiscallba := 0;
   vinit := vmin;
   while vinit <> nil do begin
      vinit^.markcall := 0;
      vinit := vinit^.next;
      end;
end;


begin    (* match *)
   matchinit;
   augmented := true;
   while augmented do begin
      bfsinitsearch;
      bfsinitvertices;
      bfsinitedges;
      while not augmented and (serlev < maxlev^.levnum) do begin
         serlev := serlev + 1;               (* increment to next
breadth first search level *)
       activelevel := findlevel(serlev);
       if serlev < maxlev^.levnum then
            nextlevel := findlevel(serlev+1)
       else
          nextlevel := nil;
         cv := activelevel^.vertices;             (* set current
vertex *)
        if  serlev mod 2 = 0  then begin      (* search level is
even *)
            while  cv <> nil  do begin    (* for each vertex with
evenlevel(v) = search levl *)
               ce := cv^.elist;
               while ce <> nil do begin     (* for each unmatched
unused neighbor *)
                  cu := otherv(cv,ce);
                  if ce^.v1=cv then         (* get next edge incident
to v.  *)
                     nx := ce^.nx1
                  else
                     nx := ce^.nx2;
                  if (cv^.mate <> cu) and not ce^.used then    (* an
unmatched, unused neighbor *)
                     if (cu^.evlev < inf) and (not ce^.isabridge)
then begin   (* evenlevel u is finite *)
                        temp := (cu^.evlev+cv^.evlev) div 2;  (* add
ce to brigdes of level temp *)
                        addbridge (findlevel(temp),ce);
                        end
                     else if not ce^.isabridge then begin
                        if cu^.odlev = inf then setlev
(odd,cu,nextlevel);
                        if cu^.odlev = serlev + 1 then
                           addpred (cu,ce)  (* add cv to predecessors
of cu *)
                        else if cu^.odlev < serlev then
                           addanom (cu,ce);      (* add cv to
anomalies of cu *)
                        end;
                  ce := nx;
                  end;
               cv := cv^.nxeven;           (*  process next vertex
on this search level *)
```

```
            end;        (* while ... *)
               end       (* even search level and all vertices on
this level *)
         else                    (* odd search level *)
            while cv <> nil do begin         (* for all vertices at
this level *)
               ce := cv^.medge;
               cu := cv^.mate;

               if (cu^.odlev = serlev) and (not ce^.isabridge) then
begin    (* handle bridges *)
                  temp := (cu^.odlev+cv^.odlev) div 2;
                  addbridge (findlevel(temp),ce);         (* add
matching edge to bridges(temp) *)
                  end
               else if cu^.odlev = inf then begin        (* handle
predecessors  *)
                  setlev (even,cu,nextlevel);
                  addpred (cu,ce);          (* add the matching
vertex of v to predecessors u *)
                  end;
               cv := cv^.nxodd;             (* process next vertex on
this odd search level *)
               end;             (* while cv <> nil *)
         ce := activelevel^.firstbridge;
         while ce <> nil do begin               (* for each edge in
bridges of this level call blossaug *)
            blossaug (ce^.v1,ce^.v2,augoccur,ce);
            if augoccur then begin
            augmented := true;
            end;
            ce := ce^.bridnx;                 (* next edge is bridges
(searchlevel) *)
            end;
         end;                 (* while not augmented and ... *)
      end;                    (* while augmented *)
   end;   (* match *)


begin (* fastmatch *)
    getgraph(vmin,emin,goodgraph);
    if goodgraph then begin
        starttime := 1(*clock*);
      match(vmin,emin);
        endtime := 2(*clock*);
        fasttime := endtime-starttime;
        printmatch;
      end;
end;    (* fastmatch *)

(*  end O(E*sqrt(V)) matching algorithm *)
begin
   (*while not eof do*)
      fastmatch;
      (*readln;*)
end; (* procedure general_match*)



begin (***************** main ********************)
clrscr;
```

```
begin
   read_ring(ring_request_matrix,size_of_ring,
             num_of_requests,num_of_colors);

   create_chain(ring_request_matrix,chain_path_matrix,

size_of_ring,num_of_requests,num_chain_paths,share_edge);

   chain_coloring(chain_path_matrix,color_matrix,color_usage_matrix,
                  size_of_ring,num_chain_paths,num_of_colors);

   writeln;
   writeln ('chain coloring');
   for i:=1 to num_chain_paths do

writeln(chain_path_matrix[i].pstart:3,chain_path_matrix[i].pend:3,
              chain_path_matrix[i].pcolor:3);


   create_compatibilityG(ring_request_matrix,compatibility_matrix,
                         num_of_requests,num_of_edges);

   general_match(compatibility_matrix,num_of_requests,num_of_edges,
                 num_of_matched);

   readln;

   (* print Alg1,2 cardinality *)
   cardinality1:=0;
   for i:= 1 to num_chain_paths do
   begin
      if chain_path_matrix[i].pcolor>0 then
         cardinality1:=cardinality1+1;
   end;
   writeln('cardinality Alg1: ',cardinality1);

   if num_of_colors>=(num_of_matched div 2) then
      cardinality2:=num_of_matched
   else
      cardinality2:=num_of_colors*2;
   writeln('cardinality Alg2= ',cardinality2);

   writeln;
   if cardinality1>=cardinality2 then
      max_cardinality:= cardinality1
   else
      max_cardinality:= cardinality2;
   writeln('max cardinality= ',max_cardinality);

   readln;
end.  (* main *)




procedure uncolor_lonely_paths(var c_matrix: path_array;
                               var clr_usage_matrix: color_array;
                               n_paths,n_colors: integer);
var i,j: integer;
```

```
begin
   for i:= 1 to n_colors do
   begin
      if clr_usage_matrix[i]=1 then
      begin
         clr_usage_matrix[i]:=0;  (* free color i *)
         for j:=1 to n_paths do
         begin
            if c_matrix[j].pcolor=i then
            begin
               (* uncolor lonely path *)
               c_matrix[j].pcolor:=-3;
               (*writeln('uncolor ');*)
            end
         end (* for j*)
      end
   end; (* for i*)
(*
   writeln('chain  status,color');
   for i:=1 to n_paths do
   begin
      writeln(c_matrix[i].pstart:2,' ',c_matrix[i].pend:2,'   ',
              c_matrix[i].pstatus,' ',c_matrix[i].pcolor);
   end;

   writeln('color usage');
   for i:=1 to n_colors do
      write(clr_usage_matrix[i],' ');
   readln;
*)
end; (* procedure uncolor_lonely_paths *)


procedure rpc_color_mates(var m_match: partners;
                          var c_matrix: path_array;
                          var clr_usage_matrix: color_array;
                          c_n_paths,n_colors: integer);
(* custom step4 of alg2 computer networks for MaxRpc*)
var i,p,q,curent_color: integer;
    mates,free_colors: integer;
    found_mate: boolean;

begin
   mates:=0;
   for i:=1 to c_n_paths do
   begin
      if m_match[i]>0 then
         mates:=mates+1;
   end;

   free_colors:=0;
   for i:=1 to n_colors do
   begin
      if clr_usage_matrix[i]=0 then
         free_colors:=free_colors+1;
   end;

   while (mates>0) (*and (free_colors>0)*) do
   begin
```

```
uncolor_lonely_paths(c_matrix,clr_usage_matrix,c_n_paths,n_colors);

      free_colors:=0;
      for i:=1 to n_colors do
      begin
         if clr_usage_matrix[i]=0 then
            free_colors:=free_colors+1;
      end;

      if free_colors>0 then
      begin
         i:=1; found_mate:=false;
         while (i<=c_n_paths) and (found_mate=false) do
         begin
            if m_match[i]>0 then
            begin
               p:=i;
               q:=m_match[i];
               m_match[q]:=0;
               m_match[p]:=0;
               mates:=mates-1;
               found_mate:=true;
            end
            else
              i:=i+1;
         end;

         if (found_mate=true) and
            ((c_matrix[p].pcolor=-3) or (c_matrix[q].pcolor=-3)) then
         begin
            curent_color:=c_matrix[p].pcolor;
            c_matrix[p].pcolor:=-3;
            clr_usage_matrix[curent_color]:=
clr_usage_matrix[curent_color]-1;
            curent_color:=c_matrix[q].pcolor;
            c_matrix[q].pcolor:=-3;
            clr_usage_matrix[curent_color]:=
clr_usage_matrix[curent_color]-1;

            i:=1;
            while (clr_usage_matrix[i]>0) and (i<=n_colors) do
            begin
               i:=i+1;
            end;
            if i<=n_colors then
            begin
               clr_usage_matrix[i]:=2;
               c_matrix[p].pcolor:=i;
               c_matrix[q].pcolor:=i;
               free_colors:=free_colors-1;
            end;
         end (* if found_mate *)
         else
            mates:=0;
      end (* if free_colors*)
      else mates:=0;
   end;   (* while *)
end;   (* procedure rpc_color_mates *)
```

```
procedure use_free_colors(var clr_usage_matrix: color_array;
                          var path_matrix: path_array;
                          n_paths,n_colors: integer);
(* use free colors to color paths in arbitrary way *)
var i,j,uncolored: integer;

begin
   for i:=1 to n_colors do
   begin
      if clr_usage_matrix[i]=0 then
      begin
         for j:=1 to n_paths do
         begin
            if path_matrix[j].pcolor<=0 then
            begin
               uncolored:=j;
            end;
         end;
         path_matrix[uncolored].pcolor:=i;

         clr_usage_matrix[i]:=1;
      end;
   end;
end;   (* procedure use_free_colors *)



procedure color_lonely(var c_matrix: path_array;
                       cmp_matrix: compatibility_array;
                       n_paths,n_colors: integer;
                       var clr_usg_matrix: color_array);

(* color an uncolored path same as a lonely compatible path *)
var i,j,lonely_color,lonely_path: integer;

begin (* color_lonely *)
   for i:=1 to n_paths do
   begin
      if c_matrix[i].pcolor=-3 then
      begin
         lonely_color:=0;
         lonely_path:=0;
         for j:=1 to n_colors do
            if clr_usg_matrix[j]=1 then
               lonely_color:=j;

         for j:=1 to n_paths do
            if c_matrix[j].pcolor=lonely_color then
               lonely_path:=j;

         if (cmp_matrix[i,lonely_path]=1) and (lonely_color>0)
             and (lonely_path>0)then
         begin
            c_matrix[i].pcolor:=lonely_color;
            clr_usg_matrix[lonely_color]:=2;
         end;
      end;
   end; (* for *)
end;  (* color_lonely *)
```

Βιβλιογραφικές αναφορές

1. Richard M. Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, 1972.

2. Casten Lund and Mihalis Yannakakis. On the hardness of aproximating minimization problems. In *ACM Symposium on the Theory of Computing*, 1993.

3. Ian Holyer. The NP-completeness of edge coloring. *SIAM Journal on Computing,* 10(4):718-720, 1981.

4. Χρήστος Νομικός. *Χρωματισμός μονοπατιών σε γραφήματα.* PhD thesis, Εθνικό Μετσόβιο Πολυτεχνείο, Αθήνα, 1997.

5. M.C. Carlisle, E.L. Lloyd, On the k-coloring of intervals, *Discrete App. Math. 59(1995) 225-235.*

6. C. Nomikos, S. Zachos, Coloring a Maximum number of paths in a graph, in *ICALP Satellite Workshop on Algorithm Aspects of Communication, Bologna, July 11-12-1997.*

7. P.J. Wan, L. Liu, Maximal throughput in wavelength-routed optical networks, *DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 46(1998) 15-26.*

8. T. Erlebach, K. Jansen, Maximizing the number of connections in optical tree networks, in: *Proc. ISAAC'98, LNCS, vol. 1533, Springer, Berlin, 1998, pp. 179-188.*

9. M. Garey, D.Johnson, G. Miller, C. Papadimitriou, The complexity of coloring circular arcs and chords, *SIAM J. Alg. Disc. Math. 1 (2) (1980) 216-227*

10. I. A. Karapetian, On the coloring of circular arc graphs, *Dock. Rep. Acad. Sci. Arm. Sov. Soc. Rep. 70 (5).(1980) 306-311 (in Russian). English translation by D. Gamarnik, D. Williamson, N. Edwards in http:// www.orie.cornell.edu/nedwards/wdm-routing/.*

11. Vijay Kumar, Approximating circular arc coloring and bandwidth allocation in all-optical ring networks, in: *Proceedings of Approximation Algorithms for Combinatorial Optimization Problems (AOORIX ' 98), Aalborg, Denmark, 1998.*

12. T. Erlebach, Scheduling connections in fast networks, *Ph.D. Thesis, Technishe Universitat Munchen, May 1999.*

13. C Nomikos, Routing and Path Coloring in Rings: NP-completeness, *Technical Report 15-2000, University of Ioannina, 2000.*

14. P. Raghavan, E. Upfal, Efficient routing in all-optical networks, in: *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing STOC 1994, pp. 134-143.*

15. M. Mihail, C. Kaklamanis, S. Rao, Efficient access to optical bandwidth, in: *Proceedings of the 36th Symposium on Foundations of computer Science FOCS 1995, pp. 548-557.*

16. T.H. Ma, J.P. Spinrad, Avoiding Matrix Multiplication, in: *16th workshop on Graph Theory, LNCS 484, 61-71, 1990.*

17. S. Micali, V.V Vazirani, An $O(n^{2.5})$ Algorithm for Maximum Matching in General Graphs, in: *Proc. 21st Annual Symposium on the Foundations of Computer Science, pp. 17-27, 1980*

18. Stefen s. Skiena, The Algorithm Design Manual, *Springer-Verlag New York, 1998.*

19. Thomas Erlebach, Aris Pagourtzis, Katerina Potika, Stamatis Stefanakos, Resource Allocation Problems in Multifiber WDM Tree Networks in:, *Proc.of WG 2003. LNCS 2880, pp. 218-229, Springer-Verlag.*

20. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to Algorithms, *The MIT Press, 2000.*

21. Christos Nomikos, Aris Pagourtzis, Stathis Zachos, Minimizing Request Blocking in All-Optical Rings. Proc. *INFOCOM 2003*.

22. Christos Nomikos, Aris Pagourtzis, Stathis Zachos, Satisfying a Maximum Number of pre-routed requests in all-optical rings, *Computer Networks 42 (2003) 55-63.*

23. W.L. Hsu, K.H. Tsai, Linear time algorithms for circular arc graphs and circle graphs, *Inform. Process. Lett. 40 (1991) 123-129.*