ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΟΜΕΑΣ ΠΡΟΗΓΜΕΝΩΝ ΕΦΑΡΜΟΓΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΟΠΙΚΗ ΑΝΑΖΗΤΗΣΗ ΔΥΟ ΦΑΣΕΩΝ ΓΙΑ ΔΥΝΑΜΙΚΑ ΠΡΟΒΛΗΜΑΤΑ ΙΚΑΝΟΠΟΙΗΣΗΣ ΠΕΡΙΟΡΙΣΜΩΝ

ΔΡΑΓΩΓΙΑΣ ΑΘΑΝΑΣΙΟΣ

ΕΠΙΒΛΕΠΩΝ:

ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ ΣΤΕΡΓΙΟΥ ΚΩΝΣΤΑΝΤΙΝΟΣ

ΚΟΖΑΝΗ (ΙΟΥΛΙΟΣ, 2012)

Περίληψη

Στην παρούσα εργασία γίνεται προσπάθεια επίλυσης δυναμικών προβλημάτων ικανοποίησης περιορισμών (Dynamic Constraint Satisfaction Problem - DCSPs) με πεπερασμένο πεδίο τιμών. Το DCSP θεωρείται ως μια ακολουθία από προβλήματα ικανοποίησης περιορισμών (Constraint Satisfaction Problems - CSPs). Όπου η δυναμικότητά του για λόγους απλότητας ορίζεται ως η προσθήκη σταθερού πλήθους περιορισμών σε κάθε CSP της ακολουθίας. Κύριος στόχος της εργασίας είναι η διερεύνηση μεθόδου που επικεντρώνεται στη σταθερότητα (stability) λύσεων που εντοπίζονται κατά την επίλυση ενός DCSP. Η μεθοδολογία βασίζεται σε τοπική αναζήτηση δύο φάσεων με δυνατότητα χρήσης πληροφορίας που λαμβάνεται από προγενέστερη λύση. Αυτή η μέθοδος προσπαθεί να διατηρήσει την αρχική ανάθεση, ενώ εξακολουθεί να διεξάγει μια αναζήτησης προς το καλύτερο. Τα αποτελέσματα της μεθόδου αυτής παρότι παρουσιάζουν αδυναμία ως προς τη βελτίωση της απόστασης των λύσεων των αρχικών CSPs του προς επίλυση DCSP, παρουσιάζουν αξιοσημείωτη βελτίωση των ενδιάμεσων λύσεων της ακολουθίας των CSPs. Γεγονός που το καθιστά ιδιαίτερα ενδιαφέρον για περαιτέρω διερεύνηση.

Ευχαριστίες

Με την ολοκλήρωση της παρούσας Διπλωματικής εργασίας θα ήθελα να ευχαριστήσω θερμά τον Επίκουρο καθηγητή και επιβλέποντα κ. Στεργίου Κωνσταντίνο για την υπομονή και εμπιστοσύνη του καθώς και την καθοδήγηση και την πολύτιμη βοήθεια που μου παρείχε.

Επίσης θα ήθελα να ευχαριστήσω την οικογένεια μου για την αμέριστη συμπαράσταση και υποστήριξη τους.

Τέλος θα ήθελα να ευχαριστήσω όλους όσους ήταν δίπλα μου όλα αυτά τα χρόνια και με βοήθησαν να πραγματοποιήσω τους στόχους μου.

Περιεχόμενα

Περίληψη **1**
Ευχαριστίες **2**
Εισαγωγή **4**
Κεφάλαιο 1: Επίλυση Προβλημάτων **5**
Κεφάλαιο 2: Προβλήματα ικανοποίησης περιορισμών **7**
2.1 Γενικά 7
2.2 Ορισμός CSP 7
2.2.1 Πεδίο τιμών 8
2.2.2 Περιορισμοί 9
2.3 Τεχνικές επίλυσης CSPs 9
2.3.1 Συστηματική αναζήτηση 9
2.3.2 Τεχνικές συνέπειας 11
2.3.3 Συστηματική αναζήτηση και συνέπεια 13
2.3.4 Στοχαστικοί και ευριστικοί αλγόριθμοι 15
2.3.5 Εφαρμογές 17
Κεφάλαιο 3: Δυναμικό CSP (DynamicCSP - DCSP) **19**
3.1 Περιγραφή 19
3.2 Μέθοδοι επίλυσης 20
3.3 Παραδείγματα CSP - DCSP 22
Κεφάλαιο 4: Αλγόριθμοι επίλυσης DCSP **24**
4.1 Γενικά 24
4.2 Τοπική Αναζήτηση - CSPs 25
4.3 Τοπική Αναζήτηση Δύο φάσεων - CSPs 26
4.3.2 Αλγόριθμος 28
Κεφάλαιο 5: Πειραματικά Αποτελέσματα **33**
Συμπεράσματα – Μελλοντική Εργασία **45**
Βιβλιογραφία **46**
Παράρτημα **47**

Εισαγωγή

Τα CSPs είναι προβλήματα που αφορούν την ανάθεση τιμών σε μεταβλητές προκειμένου να ικανοποιηθούν περιορισμοί μεταξύ των υποσυνόλων των μεταβλητών. Τα DCSPs είναι CSPs που αλλάζουν κατά διαστήματα στην πάροδο του χρόνου, με απώλεια ή αύξηση των τιμών, των μεταβλητών, ή των περιορισμών. Αυτές οι αλλαγές μπορεί να οδηγήσουν σε ακύρωση της τρέχουσας λύσης, όπως και όταν υπάρξει αλλαγή τιμής, ή όταν μειωθεί το πλήθος των αποδεκτών πλειάδων σε έναν περιορισμό, ή όταν υπάρξει προσθήκη νέου περιορισμού. Μια δυνητικά σημαντική διάκριση είναι μεταξύ των προσωρινών και μόνιμων αλλαγών στο πρόβλημα. Προηγούμενες έρευνες έχουν προσανατολιστεί προς την τελευταία περίπτωση.

Στη συγκεκριμένη εργασία διερευνάτε η επίλυση DCSP όπου η “δυναμικότητα” του χαρακτηρίζεται από την αλλαγή της ακολουθίας των CSPs, που συμπεριλαμβάνει, με την προσθήκη περιορισμών. Αρχικά, θα γίνει μια γενική περιγραφή σχετικά με την επίλυση προβλημάτων (κεφάλαιο 1). Στη συνέχεια, θα εξεταστούν τα CSPs, γενική περιγραφή και τεχνικές επίλυσής τους (κεφάλαιο 2). Κατόπιν, δίδεται περιγραφή του DCSP με τις τεχνικές επίλυσης του (Κεφάλαιο 3), ενώ στη συνέχεια αναπτύσσονται οι μέθοδοι που χρησιμοποιήθηκαν για την επίλυση αυτού (Κεφάλαιο 4). Σε επόμενο στάδιο ακολουθεί η παρουσίαση της πειραματικής μελέτης που πραγματοποιήθηκε για τη λήψη αποτελεσμάτων, στην οποία θα συμπεριλαμβάνεται περιγραφή και επεξήγηση των αποτελεσμάτων, καθώς και σύγκριση των αποτελεσμάτων μεταξύ των μεθόδων(Κεφάλαιο 5). Τέλος, θα παρατεθούν συμπεράσματα που προκύπτουν και σύντομη αναφορά για μελλοντική εργασία.

Κεφάλαιο 1: Επίλυση Προβλημάτων

Στον πραγματικό κόσμο είναι πολύ συνηθισμένα τα προβλήματα που προκύπτουν να περιέχουν δύσκολους και εύκολους περιορισμούς.

Η επίλυση προβλημάτων είναι κλάδος της τεχνητής νοημοσύνης (ΤΝ) ο οποίος αφορά τον σχεδιασμό κατάλληλων ενεργειών με στόχο την άφιξη ενός ελέγξιμου συστήματος σε μία αποδεκτή τελική κατάσταση, εκκινώντας από κάποια προκαθορισμένη αρχική κατάσταση. Συνήθως αυτό γίνεται μέσω ενός αλγορίθμου ο οποίος λαμβάνει ως είσοδο το δοθέν πρόβλημα και επιστρέφει ως έξοδο μία λύση σε αυτό, αφού αξιολογήσει πρώτα μία ομάδα υποψηφίων λύσεων. Πρόκειται για ένα θεμελιώδες γνωστικό πεδίο της τεχνητής νοημοσύνης το οποίο γνώρισε μεγάλη ανάπτυξη ήδη από τη δεκαετία του 1950. Αποτέλεσε μία προσπάθεια αλγοριθμικής εξομοίωσης της διαδικασίας της σκέψης, ενώ με τον καιρό ενσωμάτωσε μεθοδολογίες από τη θεωρία βελτιστοποίησης και τη θεωρία γράφων.

Ένα πρόβλημα είναι ένα σύνολο αντικειμένων, ιδιοτήτων και σχέσεων το οποίο ορίζεται από μία αρχική κατάσταση, μία επιθυμητή τελική κατάσταση και τις επιτρεπτές ενέργειες στα αντικείμενα του προβλήματος. Στόχος είναι, ξεκινώντας από την αρχική κατάσταση, να γίνει μία κατάλληλη ακολουθία ενεργειών η οποία να καταλήγει στην τελική κατάσταση. Αυτή η διαδικασία ονομάζεται επίλυση του προβλήματος (π.χ. η διεξαγωγή μίας παρτίδας σκάκι) και αποτέλεσε στόχο της ΤΝ από τη δεκαετία του '50. Η επίλυση προβλημάτων έχει προφανώς σπουδαίες εφαρμογές στην απόδειξη θεωρημάτων, στο χρονοπρογραμματισμό ενεργειών, στη διεξαγωγή παιγνίων κλπ, ενώ θεωρείται κεντρικό χαρακτηριστικό της ευφυΐας. Βασικό στοιχείο στην επίλυση προβλημάτων είναι η αναπαράσταση τους και γι' αυτό το σκοπό υπάρχουν δύο μεθοδολογίες: η αναπαράσταση με χώρο καταστάσεων και η αναπαράσταση με αναγωγή. Στη μέθοδο της αναγωγής δομική μονάδα περιγραφής του προβλήματος είναι η ίδια η περιγραφή αναλυόμενη σε πολλαπλές, απλούστερες εκδοχές. Αυτή η ανάλυση συμβαίνει διαδοχικά ώσπου να καταλήξει σε αρχέγονα προβλήματα επιλυόμενα με προφανή τρόπο. Προγραμματιστικά η αναγωγή υλοποιείται με αναδρομή και κεντρική έννοια σε αυτήν αποτελούν οι τελεστές αναγωγής, διαδικασίες οι οποίες ανάγουν ένα πρόβλημα σε υποπροβλήματα.

Ο πιο κατάλληλος τρόπος γραφικής αναπαράστασης του προβλήματος είναι ένας δενδρικός γράφος με ρίζα την αρχική κατάσταση, φύλλα τις τελικές καταστάσεις και τα αδιέξοδα, κόμβους τις ενδιάμεσες καταστάσεις και κλαδιά τους τελεστές μετάβασης. Επέκταση ενός κόμβου ονομάζεται η εύρεση όλων των παιδιών του στο δένδρο μέσω της εφαρμογής σε αυτόν όλων των πιθανών τελεστών. Οι λύσεις του προβλήματος

είναι μονοπάτια από τη ρίζα σε κάποιο φύλλο του δένδρου που αντιστοιχεί σε τελική κατάσταση. Σε πραγματικά προβλήματα το μέγεθος αυτού του δένδρου γίνεται εξαιρετικά μεγάλο μετά την επέκταση λίγων μόλις κόμβων και επομένως η αναζήτηση λύσεων σε ένα τέτοιο δένδρο καθίσταται εξαιρετικά χρονοβόρα. Αυτό το ζήτημα στη βιβλιογραφία της ΤΝ αναφέρεται ως *συνδυαστική έκρηξη*.

Στη μέθοδο χώρου καταστάσεων βασική δομική μονάδα είναι η κατάσταση, το σύνολο δηλαδή των αντικειμένων που εμπλέκονται στο πρόβλημα συν τις ιδιότητες τους και τις μεταξύ τους σχέσεις. Η κατάσταση ορίζεται σε ένα απλουστευμένο, αφαιρετικό μοντέλο του κόσμου και το σύνολο των καταστάσεων (στιγμιότυπων) στις οποίες μπορεί να βρεθεί αυτός ο κόσμος του προβλήματος ονομάζεται χώρος καταστάσεων. Το ίδιο το πρόβλημα ορίζεται με βάση την αρχική κατάσταση από την οποία ξεκινάμε, την επιθυμητή τελική κατάσταση στην οποία πρέπει να καταλήξουμε (ή πολλές δυνατές τελικές) και το σύνολο των τελεστών μετάβασης, επιτρεπτών πράξεων δηλαδή που μπορούν να εκτελεστούν στα αντικείμενα μίας κατάστασης οδηγώντας σε μια άλλη (π.χ. στην αναπαράσταση μίας παρτίδας σκάκι τελεστής είναι η έγκυρη μετακίνηση ενός πιονιού). Λύση του προβλήματος είναι μία ακολουθία διαδοχικών τελεστών μετάβασης και καταστάσεων που ξεκινά από μία αρχική κατάσταση και καταλήγει σε μία τελική.

Κεφάλαιο 2: Προβλήματα ικανοποίησης περιορισμών

2.1 Γενικά

Η δυνατότητα επίλυσης πολύ δύσκολων προβλημάτων του πραγματικού κόσμου, καθιστά τον *προγραμματισμό με περιορισμούς* (Constraint Programming, CP)μια ιδιαίτερα ενδιαφέρον τεχνική προς ερευνά, κερδίζοντας την προσοχή των ειδικών.

Ο *CP* αποτελεί ένα πλαίσιο εργασιών για την επίλυση συνδυαστικών προβλημάτων βελτιστοποίησης. Η βασική ιδέα στην οποία στηρίζεται είναι η μοντελοποίηση προβλήματος σαν ένα σύνολο μεταβλητών, σε κάθε μια από τις μεταβλητές αντιστοιχεί ένα πεδίο τιμών, και ένα σύνολο περιορισμών για τους πιθανούς συνδυασμούς τιμών μεταξύ των μεταβλητών. Καθώς τα πεδία τιμών είναι συνήθως πεπερασμένα, τα προβλήματα αυτά αναφέρονται γενικά σαν Προβλήματα Ικανοποίησης Περιορισμών (*Constraint Satisfaction Problems ή CSPs*). Τα CSPs έχουν μια πλούσια ιστορία στην Τεχνητή Νοημοσύνη (Davis & Rosenfeld, 1981, Dechter, Meiri, &Pearl, 1991, Dechter & Pearl, 1988, Freuder, 1989,1990,Mackworth, 1977, Mackworth&Freuder, 1985, Villain&Kautz, 1986,Waltz, 1975). Σκοπός των προβλημάτων αυτών είναι η εύρεση των τιμών των μεταβλητών, τέτοιων ώστε να ικανοποιούνται όλοι οι περιορισμοί του προβλήματος.

2.2 Ορισμός CSP

Σε γενικές γραμμές, ένα CSP είναι ένα πρόβλημα που ορίζεται ως

- Ένα σύνολο μεταβλητών $X=\{x_1, \ldots, x_n\}$
- Κάθε μεταβλητή x_i, έχει ένα μη κενό πεπερασμένο σύνολο D_i των δυνατών της τιμών (πεδίο ορισμού της).
- Ένα σύνολο περιορισμών C.
- Κάθε περιορισμός, αναφέρεται σε κάποιο υποσύνολο των μεταβλητών, καθορίζει τους επιτρεπτούς συνδυασμούς τιμών μεταξύ των μεταβλητών όπου αναφέρεται.

Μια κατάσταση του προβλήματος ορίζεται με ανάθεση τιμών σε μερικές ή όλες τις μεταβλητές (πλήρης ανάθεση). Μια ανάθεση τιμής που δεν παραβιάζει κανέναν περιορισμό ονομάζεται συνεπής ή νόμιμη.

Ως λύση ενός CSP ορίζεται η ανάθεση τιμών σε όλες τις μεταβλητές, από το πεδίο ορισμού τους, με τέτοιο τρόπο, ώστε να ικανοποιούνται όλοι οι περιορισμοί ταυτόχρονα. Μερικά προβλήματα ικανοποίησης περιορισμών απαιτούν επίσης μια λύση που μεγιστοποιεί μια αντικειμενική συνάρτηση.

Ανάλογα με το πρόβλημα, μπορεί να χρειαζόμαστε:

- Μόνο μία λύση, χωρίς ιδιαίτερη προτίμηση για το ποια,
- Όλες τις λύσεις,
- Μια βέλτιστη, ή τουλάχιστον πολύ καλή λύση, σύμφωνα με μια δοθείσα αντικειμενική συνάρτηση που ορίστηκε με βάση κάποιες ή όλες τις μεταβλητές.

Οι μέθοδοι εύρεσης λύσεων σε ένα CSP χωρίζονται σε δύο κύριες κατηγορίες, μέθοδοι που στοχεύουν:

1. σε μερικές λύσεις (ή μερικές αναθέσεις τιμών)
2. σε πλήρεις λύσεις (ή αναθέσεις τιμών σε όλες τις μεταβλητές).

2.2.1 Πεδίο τιμών

Τα πεδία τιμών των μεταβλητών χαρακτηρίζονται ως πεπερασμένα, άπειρα ή συνεχή.

Στα CSPs με πεπερασμένα πεδία ορισμού περιλαμβάνονται τα προβλήματα ικανοποίησης περιορισμών Boole (BooleanCSP), όπου οι μεταβλητές τους μπορεί να έχουν τιμή είτε αληθές«1» είτε ψευδές «0».

Οι διακριτές μεταβλητές που έχουν άπειρα πεδία (π.χ. το σύνολο των ακεραίων ή το σύνολο συμβολοσειρών), για παράδειγμα ο χρονοπρογραμματισμός οικοδομικών εργασιών σε ένα ημερολόγιο, δεν είναι δυνατόν να περιγράφουν περιορισμούς με απαρίθμηση όλων των επιτρεπτών συνδυασμών τιμών. Στην περίπτωση αυτή χρησιμοποιείται μια γλώσσα περιορισμών. Για παράδειγμα, αν η εργασία1, που χρειάζεται πέντε ημέρες για την υλοποίησή της, πρέπει να προηγείται από την εργασία2, τότε χρειαζόμαστε μια γλώσσα περιορισμών με αλγεβρικές ανισότητες. Επίσης δεν είναι πλέον δυνατόν να επιλύονται αυτοί οι περιορισμοί με απαρίθμηση

όλων των δυνατών αναθέσεων(άπειρες αναθέσεις). Υπάρχουν ειδικοί αλγόριθμοι επίλυσης για γραμμικούς περιορισμούς σε ακέραιες μεταβλητές.

Τα CSPs με συνεχή πεδίο τιμών είναι πολύ συνηθισμένα στον πραγματικό κόσμο και μελετώνται εκτεταμένα στο πεδίο της επιχειρησιακής έρευνας. Η γνωστότερη κατηγορία αυτού του είδους CSPs είναι τα προβλήματα γραμμικού προγραμματισμού, όπου οι περιορισμοί πρέπει να είναι γραμμικές ανισότητες που σχηματίζουν μια κυρτή περιοχή.

2.2.2 Περιορισμοί

Οι τύποι των περιορισμών εξαρτώνται από το πλήθος των μεταβλητών που περιλαμβάνουν (π.χ. μοναδιαίος περιορισμός: περιορίζει την τιμή μίας μόνο μεταβλητής, δυαδικός περιορισμός: συσχετίζει δύο μεταβλητές…). Επίσης οι περιορισμοί χαρακτηρίζονται ως απόλυτοι (απόλυτη περιορισμοί: η παραβίασή τους προκαλεί την απόρριψη μιας ενδεχόμενης λύσης) και προτίμησης (περιορισμοί προτίμησης: υποδεικνύουν ποιες λύσεις είναι προτιμότερες).

Τα CSPs όπου οι καταστάσεις τους και ο έλεγχος στόχου τους συμμορφώνεται με μια καθιερωμένη, δομημένη και απλή αναπαράσταση, παρέχουν την δυνατότητα να οριστούν αλγόριθμοι αναζήτησης που εκμεταλλεύονται τη δομή των καταστάσεων και χρησιμοποιούν ευριστικούς μηχανισμούς γενικής χρήσης και όχι ειδικούς για συγκεκριμένο πρόβλημα, ώστε να γίνει δυνατή η επίλυση μεγάλων προβλημάτων.

2.3 Τεχνικές επίλυσης CSPs

2.3.1 Συστηματική αναζήτηση

Από θεωρητικής σκοπιάς, η λύση προβλημάτων ικανοποίησης περιορισμών με χρήση συστηματικής εξερεύνησης του χώρου των λύσεων είναι τετριμμένη. Αντίθετα, πρακτικά, αν και οι μέθοδοι συστηματικής έρευνας (χωρίς επιπλέον βελτιώσεις) φαίνονται πολύ απλές και μη αποδοτικές είναι πολύ σημαντικές γιατί αποτελούν τη βάση για πιο εξελιγμένους και αποδοτικούς αλγορίθμους.

Ο βασικό αλγόριθμος για CSPs, ο οποίος ψάχνει στο χώρο των πλήρων αναθέσεων, είναι ο generate-and-test (GT). Η ιδέα του GT είναι απλή. Αρχικά, γίνεται μια πλήρης ανάθεση τιμών σε όλες τις μεταβλητές (τυχαία) και στη συνέχεια, αν η ανάθεση αυτή ικανοποιεί όλους τους περιορισμούς, τότε η λύση έχει βρεθεί. Διαφορετικά γίνεται μια νέα ανάθεση τιμών στις μεταβλητές.

Ο αλγόριθμος GT είναι ένας αδύναμος αλγόριθμος που χρησιμοποιείται εάν αποτύχουν όλοι οι υπόλοιποι. Η αποδοτικότητά του είναι φτωχή εξαιτίας μη ενημερωμένης γεννήτριας και αργής διαπίστωσης ασυνεπειών. Επομένως, δύο τρόποι αύξησης της αποδοτικότητας του είναι οι:

Η γεννήτρια αποτιμήσεων είναι «έξυπνη». Με τον όρο αυτό εννοούμε πως παράγει μια πλήρης αποτίμηση με τέτοιο τρόπο ώστε οι συγκρούσεις που βρίσκονται στη φάση ελέγχου να ελαχιστοποιούνται. Αυτή είναι η ιδέα των στοχαστικών αλγορίθμων που βασίζονται στη τοπική αναζήτηση.

Η γεννήτρια είναι συγχωνευμένη με τον δοκιμαστή, δηλαδή, η εγκυρότητα του κάθε περιορισμού ελέγχεται αμέσως μόλις λάβουν τιμές οι μεταβλητές που εμπλέκονται σε αυτόν. Αυτή η μέθοδος χρησιμοποιείται στην προσέγγιση με υπαναχώρηση (backtracking).

Η υπαναχώρηση (BT) είναι η μέθοδος λύσης CSP που επεκτείνει αυξητικά μια μερική λύση (ορισμένες μεταβλητές έχουν λάβει τιμές συνεπείς προς τους περιορισμούς) του προβλήματος προς την ολική. Ο τρόπος λειτουργίας της είναι, επαναλαμβανόμενα, να επιλέγει μια τιμή για μια νέα μεταβλητή η οποία θα είναι συνεπής με τις τιμές της τρέχουσας μερικής λύσης.

Όπως αναφέρθηκε και παραπάνω, μπορούμε να δούμε την BT σαν μια συγχώνευση των φάσεων παραγωγής και ελέγχου του αλγορίθμου GT. Αναθέτονται τιμές στις μεταβλητές σειριακά και μόλις όλες οι μεταβλητές κάποιου περιορισμού έχουν πάρει τιμή, γίνεται έλεγχος για την εγκυρότητά του. Εάν η μερική λύση παραβιάζει κάποιον από τους περιορισμούς, πραγματοποιείται υπαναχώρηση στη πιο πρόσφατα αρχικοποιημένη μεταβλητή που έχει ακόμα εναλλακτικές αρχικοποιήσεις. Γίνεται αντιληπτό ότι, όποτε μια μερική αρχικοποίηση παραβιάζει κάποιον περιορισμό, η υπαναχώρηση μπορεί να διαγράψει ένα υποχώρο του καρτεσιανού προϊόντος όλων των πεδίων ορισμού των μεταβλητών. Κατά συνέπεια, η υπαναχώρηση είναι πολύ καλύτερη από την μέθοδο GT, αν και η πολυπλοκότητά της για τα περισσότερα προβλήματα είναι και πάλι εκθετικού βαθμού.

Υπάρχουν τρία μειονεκτήματα της καθιερωμένης (χρονολογικής) υπαναχώρησης:

- Το thrashing: η συνεχόμενη αποτυχία λόγω του ίδιου λάθους.
- Η περιττή εργασία, αφού οι συγκρουόμενες μεταβλητές τιμών δεν συγκρατούνται στη μνήμη
- Η αργή εύρεση της σύγκρουσης, αφού η σύγκρουση δεν μπορεί να γίνει αντιληπτή πριν συμβεί.

Έχουν προταθεί μέθοδοι για την αντιμετώπιση των δύο πρώτων μειονεκτημάτων όπως οι μέθοδοι Backjumping και Backmarking, αλλά περισσότερη προσοχή δόθηκε στο να ανιχνεύονται οι ασυνέπειες νωρίτερα με χρήση τεχνικών συνέπειας.

2.3.2 Τεχνικές συνέπειας

Μια άλλη προσέγγιση για την λύση CSP είναι βασισμένη στην αφαίρεση ασυνεπών τιμών από τα πεδία ορισμού των μεταβλητών μέχρι την εύρεση της λύσης. Αυτές οι μέθοδοι ονομάζονται τεχνικές συνέπειας και παρουσιάστηκαν για πρώτη φορά στο πρόβλημα μαρκαρίσματος σκηνής (scenelabeling). Να σημειώσουμε ότι οι τεχνικές συνέπειας είναι μη-ντετερμινιστικές σε αντίθεση με την ντετερμινιστική αναζήτηση.

Υπάρχουν αρκετές μέθοδοι συνέπειας αλλά οι περισσότερες από αυτές δεν είναι πλήρεις. Επομένως, οι τεχνικές συνέπειας σπάνια χρησιμοποιούνται μόνες τους για τη λύση ενός CSP.

Τα ονόματα των βασικών τεχνικών συνέπειας προέρχονται από έννοιες γράφων. Το CSP αντιπροσωπεύεται συνήθως ως γράφος (δίκτυο) περιορισμών, όπου οι κόμβοι αντιστοιχούν στις μεταβλητές και οι ακμές αντιστοιχούν στους περιορισμούς. Η συγκεκριμένη διαδικασία απαιτεί το CSP να έχει μια συγκεκριμένη μορφή, γνωστή ως δυαδικό CSP (binary CSP), δηλαδή να περιέχει μοναδιαίους και δυαδικούς περιορισμούς μόνο. Έχει δειχτεί ότι αυθαίρετα CSPs μπορούν να μετατραπούν σε δυαδικά, αλλά η διαδικασία της δυαδικοποίησης πιθανότατα δεν θα αξίζει τον κόπο αφού οι αλγόριθμοι μπορούν εύκολα να επεκταθούν για να μπορούν να χειριστούν μη δυαδικά CSPs.

Η πιο απλή τεχνική συνέπειας είναι η συνέπεια κόμβου (node consistency, NC). Απλά αφαιρεί τιμές από τα πεδία ορισμού μεταβλητών οι οποίες είναι ασυνεπείς με μοναδιαίους περιορισμούς των αντίστοιχων μεταβλητών.

Η πιο διαδεδομένη τεχνική συνέπειας είναι η συνέπεια τόξου (arc consistency, AC). Αυτή η τεχνική αφαιρεί τιμές από τα πεδία τιμών μεταβλητών οι οποίες δεν είναι συνεπείς με δυαδικούς περιορισμούς. Συγκεκριμένα, το τόξο (V_i, V_j) είναι συνεπές αν και μόνο αν για κάθε τιμή x του τρέχοντος πεδίου ορισμού του Vi που ικανοποιεί τους περιορισμούς του V_i, υπάρχει κάποια τιμή y στο πεδίο ορισμού του V_j ώστε αν V_i=x και V_j=y, να μην παραβιάζεται ο δυαδικός περιορισμός ανάμεσα στο V_i και στο V_j.

Υπάρχουν αρκετοί αλγόριθμοι συνέπειας τόξου, που ξεκινούν από τον AC-1 και καταλήγουν κάπου στον AC-7. Οι αλγόριθμοι αυτοί είναι βασισμένοι στις επαναληπτικές αναθεωρήσεις τόξων έως ότου να επιτευχθεί μια συνεπής κατάσταση να γίνει το πεδίο τιμής κάποιας μεταβλητής άδειο. Οι πιο δημοφιλείς ανάμεσά του είναι οι AC-3 και AC-4. Ο AC-3 πραγματοποιεί αναθεωρήσεις μόνο στα τόξα τα οποία είναι πιθανό να επηρεάστηκαν από μια προηγούμενη αναθεώρηση. Δεν απαιτεί κάποια ειδική δομή δεδομένων, σε αντίθεση με τον AC-4, ο οποίος λειτουργεί με μεμονωμένα ζευγάρια τιμών για να αφαιρέσει πιθανή ανεπάρκεια που προκύπτει από τον επαναλαμβανόμενο έλεγχο ζευγαριών. Χρειάζεται ειδική δομή δεδομένων για να λειτουργήσει, για να μπορεί να «θυμάται» τα ζευγάρια των ασυνεπών τιμών των ασυνεπών μεταβλητών και είναι, συνεπώς, λιγότερο αποδοτικός σε θέματα μνήμης από τον AC-3.

Ακόμα περισσότερες ασυνεπείς τιμές μπορούν να αφαιρεθούν με τη χρήση τεχνικών συνέπειας μονοπατιού (Path Consistency, PC). Η συνέπεια μονοπατιού απαιτεί, για κάθε ζευγάρι τιμών δύο μεταβλητών X και Y, οι οποίες ικανοποιούν τον αντίστοιχο δυαδικό περιορισμό, να υπάρχει τιμή για κάθε μεταβλητή σε κάποιο μονοπάτι ανάμεσα στη X και στη Y, τέτοια ώστε όλοι οι δυαδικοί περιορισμοί σε αυτό το μονοπάτι να ικανοποιούνται. Έχει δειχτεί από τον Montanary ότι ένα CSP έχει συνέπεια μονοπατιού αν και μόνο αν κάθε μονοπάτι του με μήκος δύο έχει συνέπεια μονοπατιού. Επομένως, οι αλγόριθμοι συνέπειας μονοπατιού μπορούν να εργαστούν με τρειπλέτες μεταβλητών (μονοπάτια μεγέθους δύο). Υπάρχουν αρκετοί τέτοιοι αλγόριθμοι όπως ο PC-1 και ο PC-2, αλλά χρειάζονται εκτεταμένη απεικόνιση των περιορισμών, πράγμα που κοστίζει πολύ σε μνήμη.

Όλες οι παραπάνω τεχνικές συνέπειας καλύπτονται από μια γενική έννοια της K-συνέπειας και της ισχυρής k-συνέπειας. Ένας γράφος περιορισμών είναι k-συνεπής αν, για κάθε συνδυασμό τιμών για K-1 μεταβλητές που ικανοποιούν όλους τους περιορισμούς των μεταβλητών αυτών, υπάρχει μια τιμή για την αυθαίρετη K-οστή μεταβλητή ώστε όλοι οι περιορισμοί ανάμεσα στις K μεταβλητές να ικανοποιούνται. Ένας γράφος περιορισμών είναι ισχυρά K-συνεπής αν είναι J-συνεπής για κάθε J≤K.

Συμπληρώνοντας τα προηγούμενα, μπορούμε να πούμε ότι:

- Η συνέπεια κόμβου είναι ισοδύναμη με ισχυρή 1-συνέπεια.
- Η συνέπεια τόξου είναι ισοδύναμη με ισχυρή 2-συνέπεια.
- Η συνέπεια μονοπατιού είναι ισοδύναμη με ισχυρή 3-συνέπεια.

Έχουν δημιουργηθεί αλγόριθμοι για να κάνουν ένα γράφο περιορισμών ισχυρά K-συνεπή για K>2, αλλά πρακτικά χρησιμοποιούνται σπάνια λόγω θεμάτων απόδοσης. Αν και αυτοί οι αλγόριθμοι αφαιρούν περισσότερες ασυνέπειες από κάθε αλγόριθμο συνέπειας τόξου, δεν εξαλείφουν την ανάγκη για αναζήτηση. Μπορούμε να δούμε, ότι αν ένας γράφος περιορισμών με N κόμβους είναι ισχυρά N-συνεπής, τότε η λύση στο CSP μπορεί να βρεθεί χωρίς αναζήτηση. Αλλά, η χειρότερη δυνατή πολυπλοκότητα σε ένα τέτοιο γράφο είναι εκθετική. Δυστυχώς, εάν ένας γράφος περιορισμών είναι ισχυρά K-συνεπής για κάποιο K<N, τότε, γενικά, η αναζήτηση με υπαναχώρηση δεν μπορεί να αποφευχθεί, ακόμα υπάρχουν, δηλαδή, ασυνεπείς τιμές.

Επειδή οι περισσότερες τεχνικές συνέπειας (NC, AC, PC) δεν είναι πλήρεις, με την έννοια ότι αφήνουν κάποιες ασυνεπείς τιμές, οι περιορισμένες μορφές αυτών των τεχνικών αφαιρούν ίδιες ποσότητες ασυνεπειών, αλλά είναι πιο αποδοτικές. Για παράδειγμα, η κατευθυνόμενη συνέπεια τόξου (Directional Arc Consistency, DAC) επισκέπτεται κάθε τόξο μόνο μια φορά και έτσι απαιτεί λιγότερο υπολογισμό από την AC-3 και λιγότερη μνήμη από την AC-4. Παρόλα αυτά, η DAC μπορεί να επιτύχει πλήρη συνέπεια τόξου για πολλά προβλήματα.

2.3.3 Συστηματική αναζήτηση και συνέπεια

Οι συστηματικές τεχνικές αναζήτησης και (μερικές) τεχνικές συνέπειας μπορούν να χρησιμοποιηθούν μόνες τους για να λύσουν CSP, αλλά αυτό γίνεται σπάνια. Ο συνδυασμός των δύο προσεγγίσεων είναι ένας πιο κοινός τρόπος λύσης CSP.

Οι τεχνικές LookBack χρησιμοποιούν ελέγχους συνέπειας σε ήδη αρχικοποιημένες μεταβλητές. Η αναζήτηση με υπαναχώρηση (BT) είναι ένα απλό παράδειγμα αυτής της κατηγορίας τεχνικών. Για να αποφευχθούν κάποια προβλήματα της BT, όπως το thrashing και η περιττές εργασίες, έχουν προταθεί άλλες παρόμοιες τεχνικές.

Η υπαναχώρηση με άλμα (Backjumping, BJ) είναι μια μέθοδος που χρησιμοποιούμε για να αποφύγουμε το thrashing της BT, τη συνεχόμενη αποτυχία λόγω του ίδιου

σφάλματος. Η λειτουργία της BJ είναι ίδια με της BT, εκτός από το σημείο που πραγματοποιείται η υπαναχώρηση. Και οι δύο αλγόριθμοι επιλέγουν μια μεταβλητή κάθε φορά και ψάχνουν για μια τιμή για την μεταβλητή αυτή, σιγουρεύοντας ότι η νέα ανάθεση είναι συμβατή με τις αναθέσεις που έχουν γίνει μέχρι εκείνο το σημείο. Ωστόσο, αν η BJ βρει μια ασυνέπεια, αναλύει την κατάσταση για να εντοπίσει την πηγή της ασυνέπειας αυτής. Χρησιμοποιεί τους περιορισμούς που παραβιάζονται για να εντοπίσει τη μεταβλητή που ευθύνεται για τις συγκρούσεις. Αν όλες οι τιμές του πεδίου ορισμού έχουν ελεγχθεί, τότε επιστρέφει στη πιο πρόσφατα συγκρουόμενη μεταβλητή. Αυτή είναι η βασική διαφορά με τη BT, οποία επιστρέφει πάντα στη αμέσως προηγούμενη μεταβλητή.

Άλλοι αλγόριθμοι της κατηγορίας LookBack είναι οι Backchecking (BC) και Backmarking (BM). Στόχος τους είναι να εξαλείψουν τη περιττή εργασία που γίνεται στη BT. Τόσο ο Backchecking, όσο και ο απόγονός του Backmarking είναι χρήσιμοι αλγόριθμοι για τη μείωση των ελέγχων συμβατότητας. Για παράδειγμα, αν ο αλγόριθμος εντοπίσει ότι η ανάθεση Y/b (ανάθεση της τιμής b στη μεταβλητή Y) δεν είναι συμβατή με καμία πρόσφατη ανάθεση X/a (ανάθεση της τιμής a στη μεταβλητή X), τότε κρατάει στη μνήμη αυτή την ασυμβατότητα. Έτσι, εφόσον η X/a είναι δεσμευμένη, η Y/b δεν θα εξεταστεί πάλι.

Ο Backmarking είναι μια βελτίωση του Backchecking που αποφεύγει τόσο περιττούς ελέγχους περιορισμών, όσο και περιττές ανακαλύψεις ασυνεπειών. Μειώνει τον αριθμό ελέγχων συμβατότητας, κρατώντας στη μνήμη, για κάθε τρέχουσα ανάθεση, τις μη συμβατές πρόσφατες αναθέσεις. Επιπλέον, αποφεύγει τους επαναλαμβανόμενους ελέγχους συμβατότητας, οι οποίοι έχουν ήδη γίνει κ ήταν επιτυχείς.

Όλοι οι αλγόριθμοι της κατηγορίας LookBack έχουν κοινό μειονέκτημα τον αργό εντοπισμό της σύγκρουσης. Λύνουν την ασυνέπεια αφού έχει συμβεί αλλά δεν τη προλαβαίνουν πριν συμβεί. Επομένως, προτάθηκαν αλγόριθμοι που εντάχθηκαν στη κατηγορία LookAhead, οι οποίοι προλαβαίνουν τις ασυνέπειες πριν δημιουργηθούν.

Ο πρώιμος έλεγχος (Forward Checking, FC) είναι το πιο απλό παράδειγμα αλγορίθμων αυτής της κατηγορίας. Πραγματοποιεί ελέγχους συνέπειας τόξου σε ζευγάρια μεταβλητών που αποτελούνται από μια μεταβλητή δεν έχει αρχικοποιηθεί ακόμα και μία που έχει. Πρακτικά, όταν ανατίθεται μια τιμή στη τρέχουσα μεταβλητή, οποιαδήποτε τιμή στο πεδίου ορισμού κάποιας μελλοντικής μεταβλητής η οποία συγκρούεται με αυτή την ανάθεση, αφαιρείται (προσωρινά) από το πεδίο ορισμού. Επομένως, ο FC διατηρεί τη σταθερότητα ότι για κάθε μεταβλητή που δεν έχει

αρχικοποιηθεί ακόμα, υπάρχει τουλάχιστον μία τιμή στο πεδίο ορισμού της, η οποία να είναι συμβατή με τις αναθέσεις τιμών στις μεταβλητές που έχουν αρχικοποιηθεί μέχρι εκείνη τη στιγμή. Ο FC κάνει περισσότερη δουλειά από την BT όταν κάθε ανάθεση προστίθεται στη τρέχουσα μερική λύση και, γενικά, είναι πάντα καλύτερη επιλογή από τη χρονολογική υπαναχώρηση.

Ακόμα περισσότερες μελλοντικές ασυνέπειες αφαιρούνται με τη μέθοδο μερικής προεξέτασης (PartialLookAhead, PLA). Σε αντίθεση με τον FC, που πραγματοποιεί ελέγχους στους περιορισμούς της τρέχουσας μεταβλητής και των μελλοντικών μεταβλητών, ο PLA επεκτείνει αυτό τον έλεγχο συνέπειας στις μεταβλητές που δεν έχουν απευθείας σύνδεση με τις αρχικοποιημένες μεταβλητές, χρησιμοποιώντας κατευθυνόμενη συνέπεια τόξου.

Η προσέγγιση που χρησιμοποιεί πλήρης συνέπεια τόξου μετά από κάθε βήμα ανάθεσης ονομάζεται πλήρης προεξέταση ((Full) LookAhead, LA) ή αλλιώς διατήρηση συνέπειας τόξου (MaintainingArcConsistency, MAC). Μπορεί να χρησιμοποιήσει αλγόριθμο συνέπειας τόξου για να επιτύχει συνέπεια, ωστόσο, πρέπει να σημειωθεί ότι ο LA κάνει περισσότερη δουλειά από τον FC και η PLC, όταν κάθε ανάθεση προστίθεται στη τρέχουσα μερική λύση. Στη πραγματικότητα, ο LA, σε μερικές περιπτώσεις, μπορεί να είναι πιο “ακριβός” υπολογιστικά από την BT και γι’ αυτό ο FC και η BT χρησιμοποιούνται ακόμα σε εφαρμογές.

2.3.4 Στοχαστικοί και ευριστικοί αλγόριθμοι

Εκτός των αλγορίθμων ικανοποίησης περιορισμών που επεκτείνουν μια μερική συνεπής λύση σε μια πλήρης ανάθεση, η οποία ικανοποιεί όλους τους περιορισμούς, άπληστες στρατηγικές τοπικής αναζήτησης έχουν αρχίσει να κερδίζουν το ενδιαφέρον.

Οι αλγόριθμοι τοπικής αναζήτησης, όπως «Αναζήτηση με αναρρίχηση λόφων – Hillclimbing» και «Αναζήτηση ελάχιστων συγκρούσεων – MinConflicts», αποδεικνύονται πολύ αποτελεσματικοί για την επίλυση πολλών CSPs. Η τοπική αναζήτηση ξεκινά με την πλήρης ανάθεση τιμών σε μεταβλητές. Όταν υπάρχει παραβίαση περιορισμών γίνεται ανάθεση νέας τιμής σε μεταβλητή που επιλέγεται. Η επιλογή μεταβλητής είναι τυχαία ή με ευριστικό μηχανισμό από το σύνολο των μεταβλητών που συμμετέχουν σε παραβίαση περιορισμών.

Ο πιο προφανής ευριστικός μηχανισμός είναι να επιλέγεται η τιμή που προκύπτει με τον ελάχιστο αριθμό συγκρούσεων με άλλες μεταβλητές. Ευριστικό μηχανισμό ελάχιστων συγκρούσεων (min-conflicts). Η τοπική αναζήτηση μπορεί εύκολα να επεκταθεί στα CSPs με αντικειμενικές συναρτήσεις.

Ο αλγόριθμος Min-Conflicts είναι ιδιαίτερα αποτελεσματικός για την επίλυση πολλών CSPs, στην περίπτωση που του δοθεί μια λογική αρχική κατάσταση. Παρουσιάζει εντυπωσιακή συμπεριφορά σε προβλήματα, όπου ο χρόνος εκτέλεσής του είναι σχεδόν ανεξάρτητος του μεγέθους του προβλήματος. Επίσης είναι κατάλληλος για την επίλυση δύσκολων προβλημάτων. Για παράδειγμα έχει χρησιμοποιηθεί για το χρονοπρογραμματισμό των παρατηρήσεων στο διαστημικό τηλεσκόπιο Hubble, μειώνοντας τον χρόνο χρονοπρογραμματισμού, μιας εβδομάδας παρατηρήσεων, από τρεις εβδομάδες σε 10 λεπτά.

Ο αλγόριθμος Min-Conflicts υλοποιείται ως εξής:

1. Ξεκινάμε την ανάθεση των μεταβλητών με τυχαίες τιμές από τα πεδία τιμών τους.
2. Αν οι τιμές των μεταβλητών δεν παραβιάζουν τους περιορισμούς του προβλήματος τότε έχει βρεθεί λύση στο πρόβλημα. Επιστρέφει τις τιμές που ανατέθηκαν ως λύση.
3. Επιλέγει τυχαία, μια μεταβλητή από το σύνολο των μεταβλητών που παραβιάζουν τους περιορισμούς. Εξετάζει για την τυχαία μεταβλητή όλες τις δυνατές τιμές που μπορεί να πάρει.
    - Αν κάποια από τις τιμές της μεταβλητής, που εξετάστηκαν, μειώνει το πλήθος των περιορισμών που παραβιάζονται τότε ανέθεσε την τιμή στην μεταβλητή.
    - Αν δεν υπάρχει η παραπάνω τιμή τότε επιλέγει μια τιμή που να διατηρεί το ίδιο πλήθος περιορισμών που παραβιάζονται.
    - Αν δεν υπάρχουν οι παραπάνω τιμές, τότε διατηρεί την αρχική τιμή της εξεταζόμενης μεταβλητής.
4. Επιστρέφει στο βήμα 2.

Άλλο πλεονέκτημα της τοπικής αναζήτησης είναι ότι μπορεί να χρησιμοποιείται σε δυναμικό περιβάλλον όπου το πρόβλημα αλλάζει. Αυτό είναι ιδιαίτερα σημαντικό σε προβλήματα χρονοπρογραμματισμού. Ένας εβδομαδιαίος χρονοπρογραμματισμός αεροπορικών πτήσεων μπορεί να περιλαμβάνει χιλιάδες πτήσεις και δεκάδες χιλιάδες τοποθετήσεις προσωπικού, αλλά η κακοκαιρία μπορεί να κάνει τον

χρονοπρογραμματισμό ανέφικτο. Θα θέλαμε να ενημερώσουμε τον χρονοπρογραμματισμό με έναν ελάχιστο αριθμό αλλαγών. Αυτό επιτυγχάνεται με έναν αλγόριθμο τοπικής αναζήτησης που ξεκινά από τον τρέχοντα χρονοπρογραμματισμό. Μια αναζήτηση με υπαναχώρηση με το νέο σύνολο περιορισμών απαιτεί συνήθως πολύ περισσότερο χρόνο και θα μπορούσε να βρει μια λύση με πολλές αλλαγές σε σχέση με τον τρέχοντα χρονοπρογραμματισμό.

2.3.5 Εφαρμογές

Η προσέγγιση προβλημάτων ως CSPs έχει εφαρμοστεί επιτυχώς σε πολλές διαφορετικές περιοχές τόσο διαφορετικές μεταξύ τους όσο η ανάλυση της δομής του DNA, ο χρονοπρογραμματισμός των νοσοκομείων ή το σχεδιασμό των βιομηχανιών.

Τα προβλήματα ανάθεσης ήταν ίσως ο πρώτος τύπος βιομηχανικής εφαρμογής που επιλύθηκε με εργαλεία περιορισμών. Ένα χαρακτηριστικό παράδειγμα είναι η κατανομή θέσεων στάθμευσης για τους αερολιμένες, όπου τα αεροσκάφη πρέπει να σταθμεύσουν στη διαθέσιμη στάση κατά τη διάρκεια της παραμονής στον αερολιμένα (αερολιμένας Roissy στο Παρίσι) ή η κατανομή γκισέ για τις αίθουσες αναχώρησης (διεθνής αερολιμένας Χονγκ Κονγκ). Ένα άλλο παράδειγμα είναι η κατανομή αγκυροβολίων για σκάφη στο λιμάνι (διεθνή τερματικά Χονγκ Κονγκ.

Ένας άλλος χαρακτηριστικός τομέας εφαρμογής περιορισμών είναι η ανάθεση προσωπικού όπου οι κανόνες και οι κανονισμοί εργασίας επιβάλλουν δύσκολους περιορισμούς. Η σημαντική πτυχή σε αυτά τα προβλήματα είναι η απαίτηση να ισορροπηθεί η εργασία μεταξύ διάφορων προσώπων.

Πιθανώς ο επιτυχέστερος τομέας εφαρμογής για περιορισμούς με πεπερασμένο πεδίο είναι τα προβλήματα προγραμματισμού, όπουοι περιορισμοί εκφράζουν πραγματικά όρια. Η χρήση περιορισμών σε συστήματα προχωρημένου σχεδιασμού και προγραμματισμού αυξάνεται λόγω της τρέχουσας τάσης για παραγωγή κατόπιν παραγγελίας.

Μια άλλη μεγάλη περιοχή της εφαρμογής περιορισμών είναι η διαχείριση και διαμόρφωση δικτύων. Αυτά τα προβλήματα περιλαμβάνουν τον προγραμματισμό της τοποθέτησης καλωδίων των δικτύων τηλεπικοινωνιών σε κάποιο κτήριο (FranceTelecom) ή τον επανασχεδιασμό δικτύων ηλεκτροδότησης για τον σχεδιασμό συντήρησης χωρίς να διακόπτονται οι υπηρεσίες των πελατών (Ether). Ένα άλλο

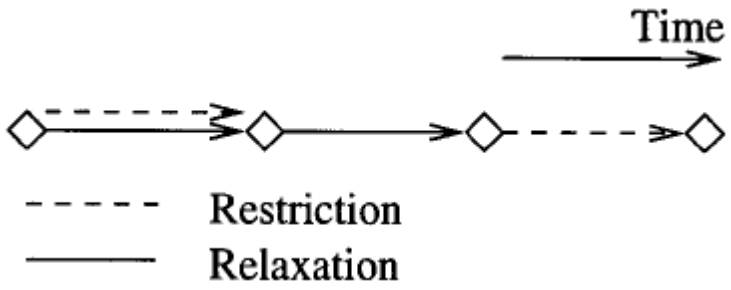
παράδειγμα είναι η βέλτιστη τοποθέτηση σταθμών βάσης στα ασύρματα εσωτερικά δίκτυα τηλεπικοινωνιών.

Παραδείγματα και εφαρμογές CSPs μπορεί να εντοπιστούν σε πολλούς τομείς, όπως η κατανομή πόρων στον προγραμματισμό, χρονική λογική, η επεξεργασία φυσικής γλώσσας, βελτιστοποίηση ερωτημάτων στη βάση δεδομένων, τεχνητή όραση, θεωρία γράφων, σχεδιασμό κυκλωμάτων κλπ.

Κεφάλαιο 3: Δυναμικό CSP (DynamicCSP - DCSP)

3.1 Περιγραφή

Το δυναμικό πρόβλημα ικανοποίησης περιορισμών (Dynamic CSP - DCSP) χαρακτηρίστηκε για πρώτη φορά από τους Dechter & Dechter ως ένα δυναμικό περιβάλλον εκφρασμένο σε μια ακολουθία από CSPs, που συνδέονται με τους περιορισμούς και “χαλαρώσεις”, που προκύπτουν με την προσθήκη ή αφαίρεση περιορισμών από το πρόβλημα αντίστοιχα (Σχήμα 1).



Σχήμα1. Μοντελοποίηση εξελισσόμενου προβλήματος μέσω του DCSP

Σημειώνετε ότι η προσθήκη ή αφαίρεση στοιχείων του πεδίου μεταβλητής είναι ουσιαστικά η προσθήκη ή αφαίρεση μοναδιαίου περιορισμού στο πρόβλημα. Αρχικά, κάθε πρόβλημα της ακολουθίας μπορεί να λυθεί από την αρχή χρησιμοποιώντας στατικές τεχνικές επίλυσης CSP,αλλά αυτή η μέθοδος απορρίπτει όλες τις ενέργειες που πραγματοποιήθηκαν για την επίλυση του προηγούμενου (πιθανώς παρόμοιου) προβλήματος. Αυτή η άσκοπη προσπάθεια είναι συνήθως μη αποδεκτή σε εφαρμογές πραγματικού χρόνου που μπορούν να δεχτούν απότομες αλλαγές στο πρόβλημα. Το “κλειδί” για την αποτελεσματική επίλυση των DCSPs είναι η επαναχρησιμοποίηση όσο το δυνατόν περισσότερης πληροφορίας που απαιτήθηκε για την κατά σειρά επίλυση των προηγούμενων προβλημάτων για την επίλυση του τρέχον CSP.

3.2 Μέθοδοι επίλυσης

Οι υπάρχουσες προσεγγίσεις για DCSP μπορούν να χαρακτηριστούν γενικά σε τρεις κατηγορίες. Η πρώτη μέθοδος (van Hentenryck & Provost, 1991)προσπαθεί να λύσει κάθε πρόβλημα στην ακολουθία από το μηδέν (δηλαδή, ένα κενό σύνολο από αναθέσεις), χρησιμοποιώντας ένα στατικό αλγόριθμο επίλυσης CSP, αλλά χρησιμοποιεί τις αναθέσεις από τη λύση του προηγούμενου προβλήματος για την καθοδήγηση της τρέχουσας προσπάθειας. Λαμβάνοντας υπόψη τις αναθέσεις για κάθε μεταβλητή, επιχειρεί την ανάθεση χρησιμοποιώντας πρώτα την προηγουμένη ανάθεση.

Μια δεύτερη μέθοδος, γνωστή ως τοπική διόρθωση (Mintonetal, 1992, Verfaillie&Schiex, 1994), διατηρεί όλες τις αναθέσεις από τη λύση του προηγούμενου προβλήματος της ακολουθίας για να χρησιμοποιηθεί ως σημείο έναρξης. Αυτή η αρχική ανάθεση μεταβλητής τροποποιείται σταδιακά από μια σειρά τροποποιήσεων σε επιμέρους αναθέσεις εωσότου προκύψει μία αποδεκτή λύση στο τρέχον πρόβλημα. Οι τροποποιήσεις λαμβάνουν τη μορφή επαναλαμβανόμενης ανάθεσης στοιχείου για την επίλυση περισσότερων συγκρούσεων (δηλαδή, παραβιάσεις των περιορισμών) που αφορούν την εν λόγω μεταβλητή. Ο ευριστικός αλγόριθμος επιδιόρθωσης (Mintonetal., 1992) στοχεύει σε τοπική επιδιόρθωση όπου αναζητά, μέσω των πιθανών επιδιορθώσεων, την ελαχιστοποίηση όσο το δυνατόν λιγότερων περιορισμών που παραβιάζονται σε κάθε στάδιο για την καθοδήγηση της αναζήτησης. Οι τοπικές αλλαγές του αλγόριθμου (Verfaillie&Schiex, 1994) διατηρούν μια λίστα των τρέχων μεταβλητών που δεν έχουν προσδιοριστεί. Όταν οι αλλαγές σε ένα DCSP εμπεριέχουν προσθήκη νέων μεταβλητών στο πρόβλημα ή μεταβλητές των οποίων η τρέχουσα ανάθεση έχει χαθεί. Αυτός ο αλγόριθμος επιλέγει επαναλαμβανόμενα την επιδιόρθωση μιας μεταβλητής από το σύνολο που δεν έχουν ανατεθεί και αναθέτει μια τιμή σε αυτή. Αν η ενέργεια αυτή προκαλεί παραβιάσεις περιορισμού, τότε τοπικές αλλαγές προσπαθούν να επιδιορθώσουν το τρέχον σύνολο των αναθέσεων για την επίλυση των ασυνεπειών. Οι αλλαγές του τοπικού αλγόριθμου καθιστούν βελτίωση της ευριστικής επιδιόρθωσης με την αποφυγή της επιδιόρθωσης των μεταβλητών που δεν έχουν καμία σχέση με τις υπάρχουσες ασυνέπειες, και ως εκ τούτου σχετίζονται με στατικούς αλγορίθμους επίλυσης CSP, όπως Backjump.

Τέλος, οι μέθοδοι καταγραφής περιορισμού (Schiex & Verfaillie 1993, VanHentenryck & Provost, 1991) συνάγουν νέους περιορισμούς από το υφιστάμενο ορισμό του προβλήματος που δεν επιτρέπει την ασυνεπή ανάθεση συνδυασμών που προκύπτει

από τους αρχικούς περιορισμούς. Οι αιτιολογήσεις των εισαχθέντων περιορισμών καταγράφονται ώστε να μπορούν να χρησιμοποιηθούν σε μελλοντικά προβλήματα που οι κατέχουν ίδιες αιτιολογήσεις για να συγκλίνουν σε μια λύση πιο γρήγορα. Οι αλλαγές τοπικού αλγόριθμου μπορούν να βελτιωθούν μέσω της χρήσης ενός τέτοιου συστήματος (Verfaillie & Schiex, 1994).

Μια δεύτερη διατύπωση του DCSP που αναπτύχθηκε πιο πρόσφατα είναι γνωστή ως περιοδικό DCSP (Freuder & Wallace, 1998). Η τεχνική αυτή βασίζεται στο γεγονός πως πολλές τροποποιήσεις σε ένα πρόβλημα μπορεί να είναι προσωρινές και ότι οι αλλαγές αυτές μπορεί διαδοχικά να αντιστραφούν. Το περιοδικό DCSP επικεντρώνεται σε αλλαγές του συνόλου των περιορισμών που θα μπορούσαν να καταστήσουν μη έγκυρη την τρέχουσα λύση. Αυτές οι αλλαγές περιλαμβάνουν την προσθήκη μοναδιαίων περιορισμών, η προσθήκη των n-αδικών περιορισμών, και την απώλεια μερικών αποδεκτών πλειάδων σε υπάρχοντα προβλήματα. Ο στόχος του περιοδικού DCSP είναι η εύρεση λύσεων που δεν “αποτυγχάνουν” από τις αλλαγές αυτές. Μια καλή λύση για το περιοδικό DCSP είναι συνεπώς η σταθερή, σε μια ελαφρώς διαφορετική έννοια παραμένει αποδεκτή υπό το πρίσμα των αλλαγών στο πρόβλημα.

Ένα σημαντικό σημείο που πρέπει να σημειωθεί είναι ότι όλες οι αλλαγές υποστηρίζονται από το αποτέλεσμα του περιοδικού DCSP στην προσωρινή απώλεια ενός ή περισσότερων στοιχείων τομέα ως πιθανά μέλη των αποδεκτών λύσεων. Υπό το πρίσμα αυτού, η πιθανότητα της απώλειας μπορεί να σχετίζεται με κάθε στοιχείο του τομέα. Μια σταθερή λύση μπορεί τότε να βρει χρήση σε ένα σύνολο αναθέσεων με όσο το δυνατόν χαμηλότερη πιθανότητα απώλειας. Δεδομένου ότι είναι απίθανο, οι επιμέρους απώλεια πιθανοτήτων να είναι γνωστή εκ των προτέρων, η πληροφορία αυτή εξάγεται δυναμικά με την καταγραφή της συχνότητας της απώλειας της κάθε τιμής καθώς εξελίσσεται το πρόβλημα.

Οι περιοδικές τεχνικές λύσης DCSP έχουν αναπτυχθεί χρησιμοποιώντας μια τοπική τεχνική επιδιόρθωσης, η οποία χρησιμοποιεί την αναρρίχηση λόφων(hill-climbing), λαμβάνοντας υπόψη τις πιθανότητες απώλειας για τη μεγιστοποίηση της σταθερότητας. Αν και ο hill-climbing δεν εγγυάται ότι η βέλτιστη λύση θα βρεθεί, αυτό είναι ένα αμφισβητήσιμο σημείο, δεδομένου ότι οι πιθανότητες απώλειας είναι μόνο εκτιμήσεις. Το επιχείρημα υπέρ της αρχή του hill-climbing είναι ότι δεν υπάρχει κανένας περιορισμός σχετικά με το χρονοδιάγραμμα των αλλαγών στο πρόβλημα. Ως εκ τούτου, εάν το πρόβλημα αλλάζει, ενώ μια πλήρης μέθοδος είναι στη διαδικασία

για την επίλυσή του, ένα από τα στοιχεία του τομέα που έχουν ανατεθεί μπορεί να χαθεί, αναγκάζοντας την ανάκληση όλων των αναθέσεων μετά από αυτό το σημείο. Επιπλέον, οι υπολογισμοί ευστάθειας μπορεί να καθιστούν αβάσιμοι, αφαιρώντας την εγγύηση της εύρεσης μιας βέλτιστης σταθερής λύσης. Οι μέθοδοι hill-climbing δεν επηρεάζονται από αυτά τα προβλήματα. Με γεγονός την αλλαγή, ο hill-climbing μπορεί να προχωρήσει κανονικά, σταματώντας μόνο για την ανάθεση εκ νέου των μεταβλητών που τα ανατιθέμενα στοιχεία τους έχουν αφαιρεθεί. Αυτή η τεχνική λύση έχει δοκιμαστεί σε graph-colouring και σε τυχαία CSPs(Wallace & Freuder, 1998). Διαπιστώθηκε ότι η επιδιόρθωση μερικής λύσης όχι μόνο αυξάνει την απόδοση για την εξεύρεση λύσης στο τρέχον πρόβλημα, αλλά είναι επίσης αποτελεσματική για την εξεύρεση σταθερών λύσεων(δηλαδή, εκείνες που είναι ικανές να παραμείνουν σε ισχύ για την αντιμετώπιση της αλλαγής). Η τεχνική της δυναμικής απόκτησης πιθανοτήτων απώλειας για τα επιμέρους στοιχεία τομέα ήταν ιδιαίτερα επιτυχής οδηγώντας σε σταθερές λύσεις, ιδιαιτέρα όταν υπάρχουν πολύ λίγες σταθερές λύσεις για να βρεθούν.

Αν και έχουν προταθεί αρκετές στρατηγικές για χειρισμό DCSPs, υπάρχουν ακόμη σημαντικά περιθώρια βελτίωσης όταν τα προβλήματα βρίσκονται σε κρίσιμη περιοχή πολυπλοκότητας.

3.3 Παραδείγματα CSP - DCSP

Στον πραγματικό κόσμο πολλά από τα προβλήματα που προκύπτουν μπορούν να μοντελοποιηθούν ως προβλήματα CSPs. Για παράδειγμα ο χρονοπρογραμματισμός για τις βάρδιες υπαλλήλων ενός εργοστασίου μπορεί να εκφραστεί ως CSP. Το πρόβλημα μπορεί να μοντελοποιηθεί ως εξής: μεταβλητές = βάρδιες, πεδίο τιμών των μεταβλητών = υπάλληλοι και περιορισμοί = περιορισμοί που εξάγονται από το πρόβλημα (όπως το πλήθος των υπαλλήλων που εργάζονται σε κάθε βάρδια). Το CSP παρουσιάζει στατικό χαρακτήρα δεν μπορεί να προσφέρει λύση σε δυναμικά περιβάλλοντα. Στο παραπάνω παράδειγμα μπορεί να προκύψει μια αναγκαία άμεση αλλαγή στο χρονοπρογραμματισμό (νέο CSP) λόγω ασθένειας υπαλλήλου ή υπαλλήλων , βλάβη σε τομέα του εργοστασίου. Το γεγονός αυτό της δυναμικής συμπεριφοράς αντιμετωπίζεται ως πρόβλημα DCSP (ακολουθία από CSPs). Για παράδειγμα σε περίπτωση ασθένειας υπαλλήλου το “αρχικό” CSP (αρχική μοντελοποίηση CSP) αλλάζει με την προσθήκη περιορισμού (νέο CSP) ώστε να μην

ληφθεί υπόψη ο ασθενής-υπάλληλος κατά τον χρονοπρογραμματισμό. Σε περίπτωση που ο ασθενείς υπάλληλος έχει αναρρώσει και είναι σε θέση να εργαστεί τότε το πρόβλημα αλλάζει εκ νέου. Το γεγονός αυτό των δυναμικών καταστάσεων εκφράζεται με την ακολουθία των CSPs.

Για να προκύψει μια πλήρης λύση κατά την επίλυση ενός CSP θα πρέπει να ικανοποιούνται όλοι οι περιορισμοί ταυτόχρονα ή αλλιώς να μην παραβιάζονται οι περιορισμοί. Πέρα της πλήρης λύσης υπάρχουν περιπτώσεις επίλυσης CSPs όπου οδηγούμαστε σε μερική λύση με ένα ορισμένο πλήθος παραβιάσεων – συγκρούσεων των περιορισμών. Σύμφωνα με αυτό για την αξιολόγηση της μερικής – πλήρης λύσης εισάγουμε την έννοια του πλήθους των συγκρούσεων "*conflict*".

Για την επίλυση κάθε CSP της ακολουθίας των CSPs ενός DCSP χρησιμοποιούνται αναθέσεις από τη λύση του προηγούμενου CSP για την καθοδήγηση της τρέχουσας προσπάθειας. Το γεγονός επαναχρησιμοποίησης πληροφορίας του προηγούμενου CSP για την επίλυση του τρέχον CSP χαρακτηρίζει την "σύνδεση" συσχέτισης των συνεχόμενων λύσεων. Σύμφωνα με αυτό για αξιολόγηση της συσχέτισης μεταξύ δύο συνεχόμενων λύσεων που στηρίζεται στο "πόσες από τις αναθέσεις του προηγούμενου CSP έχουν αλλάξει" εκφράζεται με την έννοια της απόστασης "*distance*".

Υποθέτουμε ένα πρόβλημα CSP με μεταβλητές(variables): x1, x2, x3, x4 και x5, όπου κάθε μεταβλητή έχει πεδίο τιμών (domain): {0,1,2,3,4} και ένα πλήθος περιορισμών (constraints) που είναι: $x1 < x2$, $x3 > x2$, $x4 > x5$, $x5 > x1$, $x1 \leq x4$ και $x3 \geq x4$. Μια από τις λύσεις του παραπάνω CSP είναι: x1=0, x2=1, x3=4, x4=3 και x5=2 (πλήρης λύση με μηδενικές συγκρούσεις – conflict = 0).

Έστω λόγω αλλαγών στο περιβάλλον (δυναμικότητα του περιβάλλοντος) το πρόβλημα μεταβάλλεται (DCSP). Η μεταβολή που εμφανίζει μεταφράζεται με την προσθήκη επιπλέον περιορισμού στο CSP με αποτέλεσμα τη δημιουργία ενός νέου CSP. Με την προσθήκη του περιορισμού η προηγούμενη λύση παύει να ισχύει. Ας υποθέσουμε πως ο περιορισμός που προστίθεται είναι ο $x5 < x2$. Χρησιμοποιώντας τις αναθέσεις της προηγούμενης λύσης καταλήγουμε σε μια νέα λύση από τις λύσεις του νέου CSP η οποία είναι: x1=0, x2=2, x3=4, x4=3 και x5=1. Η λύση αυτή παρουσιάζει μηδενικές συγκρούσεις – conflict = 0 και η απόστασή της από την προηγούμενη λύση είναι distance=2 (έγιναν δύο νέες αναθέσεις τιμών, στις μεταβλητές x2 και x5). Λόγω της δυναμικότητας, υπάρχει δυνατότητα περαιτέρω αλλαγών στο σύνολο των περιορισμών.

Κεφάλαιο 4: Αλγόριθμοι επίλυσης DCSP

4.1 Γενικά

Στην παρούσα εργασία γίνεται προσπάθεια επίλυσης DCSPs με τη χρήση αλγόριθμου τοπικής αναζήτησης. Η επιλογή του αλγόριθμου τοπικής αναζήτησης βασίζεται στις δύο κύριες χαρακτηριστικές ιδιότητές τους: αποτελεσματικότητα (efficiency) και ικανότητα επίλυσης ακολουθίας δύσκολων CSPs. Τα DCSP που χρησιμοποιούνται για την διερεύνηση των δυνατοτήτων των προτεινόμενων μεθόδων επίλυσης προκύπτουν με την προσθήκη σταθερού πλήθους περιορισμών.

Εξετάζεται η χρήση ενός κοινού αλγόριθμου τοπικής αναζήτησης και συγκεκριμένα του αλγόριθμου Min-conflicts προς επίλυση DCSP. Αυτή η μέθοδος προσπαθεί να διατηρήσει την αρχική ανάθεση, ενώ εξακολουθεί να διεξάγει αναζήτηση προς το καλύτερο.

Επιπλέον προτείνεται μια δεύτερη μέθοδος η οποία βασίζεται σε τοπική αναζήτηση δύο φάσεων. Η οποία στοχεύει στη σταθερότητα (stability) λύσεων αναζητώντας γειτονική λύση “κοντινή” σε σχέση με την προηγούμενη της ακολουθίας των CSPs.

Για τον έλεγχο της αποτελεσματικότητας των δύο μεθόδων κατασκευάζονται, σύμφωνα με την παραμετροποίηση που εισάγει ο χρήστης, τυχαία προβλήματα DCSP. Οι τιμές των παραμέτρων επηρεάζουν σημαντικά στην απόκριση των μεθόδων, όπως είναι αναμενόμενο και στον τρόπο σύγκρισής τους. Οι παράμετροι αυτοί είναι:

- το πλήθος των μεταβλητών,
- το αρχικό πλήθος των περιορισμών,
- πεπερασμένου πεδίου τιμών των μεταβλητών,
- το ποσοστό των μεταβλητών που ικανοποιεί τους περιορισμούς,
- το μέγιστο πλήθος των επαναλήψεων του αλγόριθμου Min-conflicts για την εύρεση λύσης,
- το πλήθος των CSP(μήκος της ακολουθίας από CSPs),
- το πλήθος των περιορισμών που προστίθεται σε κάθε νέο CSP

Συγκεκριμένα η κατασκευή ενός DCSP, ξεκινά με τη δημιουργία ενός αρχικού τυχαίου CSP, με τυχαία επιλογή ζευγών μεταβλητών για τον προσδιορισμό ύπαρξης και ικανοποίησης περιορισμών. Στη συνέχεια δημιουργούνται αλυσιδωτά τόσα νέα CSP,

με προσθήκη περιορισμών στο προηγούμενο CSP, όσο το μήκος της ακολουθίας των CSPs εκτός του αρχικού CSP.

4.2 Τοπική Αναζήτηση - CSPs

Η μέθοδος τοπικής αναζήτησης προς επίλυση DCSP που προτείνετε βασίζεται στην δυνατότητα του αλγόριθμου min-conflicts να επιλύει δύσκολα CSPs. Η διαδικασία επίλυσης που ακολουθείται κάνει επαναλαμβανόμενη χρήση του αλγόριθμου min-conflicts με στόχο την εύρεση της καλύτερης λύσης. Συγκεκριμένα η μέθοδος υλοποιείται ως εξής:

1. Εισαγωγή αρχικού CSP προς επίλυση. Τυχαία ανάθεση τιμών, αναζήτηση καλύτερης λύσης με επαναλαμβανόμενη χρήση του αλγόριθμου min-conflicts. Καταχώρηση της λύσης που προέκυψε.
2. Προσθήκη νέων περιορισμών στο προηγούμενο CSP με αποτέλεσμα την δημιουργία νέου CSP και αναζήτηση νέας λύσης.
3. Αναζήτηση καλύτερης λύσης του νέου CSP, με τη χρήση της λύσης του προηγούμενου CSP για την αρχική ανάθεση τιμών στις μεταβλητές και στη συνέχεια επαναλαμβανόμενη χρήση του αλγόριθμου min-conflicts. Καταχώρηση της λύσης που προέκυψε.
4. Πήγαινε στο βήμα 2 (τόσες φορές όσο το μήκος της ακολουθίας των CSPs).
5. Εμφάνιση των λύσεων του κάθε CSP τη ακολουθίας

Η μέθοδος προσπαθεί μέσω της επαναλαμβανόμενης χρήσης του αλγορίθμου min-conflicts με τη χρήση της προηγούμενης λύσης, σε συνδυασμό με ευριστικές συναρτήσεις (για την αποφυγή τοπικών ελάχιστων), να βρει λύση στο επόμενο CSP της ακολουθίας. Στοχεύει σε αναζήτηση προς τη καλύτερη λύση χωρίς να λαμβάνει υπόψη την απόσταση των γειτονικών λύσεων. Η παραμετροποίηση για την δημιουργία του DCSP καθορίζει σημαντικά το βαθμό της δυσκολίας του εκάστοτε προβλήματος.

4.3 Τοπική Αναζήτηση Δύο φάσεων - CSPs

4.3.1 Πολυκριτηριακό πρόβλημα βελτιστοποίησης (multi-criteria optimization problem)

Πολλά προβλήματα που αφορούν τον σχεδιασμό σύνθετων συστημάτων έχουν διατυπωθεί ως προβλήματα βελτιστοποίησης, όπου οι επιλογές σχεδίασης είναι κωδικοποιημένες ως τιμές μεταβλητών απόφασης και οι σχετικές ικανότητες κάθε επιλογής εκφράζονται μέσω κόστους/χρησιμότητας της συνάρτησης σύμφωνα με τις μεταβλητές απόφασης. Στις περισσότερες καταστάσεις βελτιστοποίησης του πραγματικού κόσμου ωστόσο το κόστος της συνάρτησης είναι πολυδιάστατο.

Για παράδειγμα, ένα κινητό τηλέφωνο προς αναπτύξει ή αγορά μπορεί να αξιολογηθεί σύμφωνα με το κόστος, το μέγεθος, την αυτονομία ενέργειας και επιδόσεων. Η μορφοποίηση των ιδιοτήτων του μπορεί να είναι καλύτερή από μία άλλη συσκευή, σύμφωνα με ένα κριτήριο, αλλά μπορεί να είναι χειρότερη, σύμφωνα με κάποιο άλλο κριτήριο. Κατά συνέπεια, δεν υπάρχει μοναδική βέλτιστη λύση, αλλά μάλλον ένα σύνολο από αποτελεσματικές λύσεις, γνωστές ως Pareto λύσεις, χαρακτηρίζονται από το γεγονός ότι το κόστος τους δεν μπορεί να βελτιωθεί σε μια διάσταση, χωρίς να επιδεινωθεί σε ένα άλλη. Το σύνολο όλων των λύσεων Pareto, αντιπροσωπεύει το πρόβλημα συμβιβασμού. Η δειγματοληψία του συνόλου αυτού με αντιπροσωπευτικό τρόπο είναι ένα πολύ χρήσιμο βοήθημα στη διαδικασία λήψης αποφάσεων.

Πολυκριτηριακά ή πολλαπλών στόχων προβλήματα βελτιστοποίησης έχουν μελετηθεί από το ξεκίνημα της σύγχρονης βελτιστοποίησης χρησιμοποιώντας διάφορες τεχνικές, ανάλογα με τη φύση των υποκείμενων προβλημάτων βελτιστοποίησης (γραμμική, μη γραμμική, συνδυαστική).

Μια προσέγγιση αποτελείται εξ ορισμό από τη συγκέντρωση μονοδιάστατου κόστους/χρησιμότητας της συνάρτησης με την εφαρμογή σταθμισμένου αθροίσματος από διαφορά κόστη. Κάθε επιλογή από ένα σύνολο συντελεστών για το άθροισμα αυτό θα οδηγεί σε μια βέλτιστη λύση για το μονοδιάστατο πρόβλημα, το οποίο είναι επίσης μια λύση κατά Pareto για το αρχικό πρόβλημα.

Μια άλλη δημοφιλής κατηγορία τεχνικών βασίζεται στην ευριστική αναζήτηση, κυρίως εξελικτικών αλγορίθμων, που χρησιμοποιούνται για την επίλυση προβλημάτων που σχετίζονται με την εξερεύνηση του σχεδιασμού των ενσωματωμένων συστημάτων.

Ένα σημαντικό ζήτημα σε αυτές τις τεχνικές είναι η εύρεση ουσιωδών μετρήσεων της ποιότητα για σύνολα λύσεων που αυτές παρέχουν.

Οι πιο αποτελεσματικές στοχαστικές προσεγγίσεις σε πολυκριτηριακά συνδυαστικά προβλήματα βελτιστοποίησης εφαρμόζουν την διαδικασία της τοπικής αναζήτησης, ενσωματωμένη σε κάποιου ψηλότερου επίπεδου στρατηγικής αναζήτησης ώστε να αποκομίσουν κατά προσέγγιση ένα καλό σύνολο από αποτελεσματικές λύσεις. Προκειμένου να εξασφαλιστεί ότι επιτυγχάνεται η διασπορά του συνόλου των αποτελεσματικών λύσεων έχουν προταθεί πολλές τεχνικές. Αυτές μπορούν να ταξινομηθούν σε μία από τις δύο κύριες ομάδες:

A. Μέθοδοι που συγκεντρώνουν αντικειμενικές συναρτήσεις και δυναμικά τροποποιούν την κατεύθυνση της αναζήτησης κατά τη διαδικασία της αναζήτησης ή σε περίπτωση ορισμένων εκτελέσεων,
B. Μέθοδοι που βασίζονται στη σχέση υπεροχής (Pareto) ως αποδεκτό κριτήριο στην αναζήτηση της διάκρισης μεταξύ των υποψήφιων λύσεων.

Ο δεύτερος τύπος μεθόδων αποφεύγει τη συγκέντρωση κριτηρίων. Πρόσφατα, σύμφωνα με συλλογιστικές εμπειρικές ενδείξεις προτάθηκε ότι οι μέθοδοι που ανήκουν στην πρώτη ομάδα είναι ιδιαίτερα αποτελεσματικοί, ο κύριος λόγος είναι ότι οι αλγόριθμοι τοπικής αναζήτησης μπορούν να οδηγήσουν σε αποτελέσματα πιο εύκολα με συγκέντρωση αντικειμενικών συναρτήσεων. Ωστόσο, οι περισσότεροι "αποτελεσματικοί" αλγόριθμοι είναι περισσότερο σύνθετες στρατηγικές αναζήτησης.

Η ομαδοποίηση πολυκριτηριακών προβλημάτων σε διάφορα μονοκριτηριακά προβλήματα με τη χρήση συναρτήσεων έχει αποδειχθεί ότι είναι μια αποτελεσματική προσέγγιση για πολυκριτηριακά συνδυαστικά προβλήματα βελτιστοποίησης. Πολλές από αυτές τις ομαδοποιήσεις προτάθηκαν, συμπεριλαμβανομένης της χρήσης σταθμισμένων συναρτήσεων Tchebycheff καθώς και σταθμισμένων γραμμικών συναρτήσεων.

Η διαδικασία επίλυσης προβλημάτων συνδυαστικής βελτιστοποίησης (στη συγκεκριμένη περίπτωση DCSPs) που προτείνεται, σε σύγκριση με την μέθοδο επίλυσης της προηγούμενης ενότητας, είναι η τοπική αναζήτηση δύο φάσεων.

4.3.2 Αλγόριθμος

Η μέθοδος τοπικής αναζήτησης δύο φάσεων στοχεύει στην αποτελεσματικότητα (efficiency) εύρεσης καλύτερης λύσης, και τη δυνατότητα εύρεσης λύσης κοντινής απόστασης με την γειτονική της. Το γεγονός αυτό επιτυγχάνεται με τη χρήση δύο ευριστικών μηχανισμών. Ο πρώτος στοχεύει στην αποτελεσματικότητα (efficiency) με τη χρήση του min-conflicts με επαναλαμβανόμενο τρόπο και ο δεύτερος στη σταθερότητα (stability) μεταξύ των λύσεων που προκύπτουν από την επίλυση των προβλημάτων της ακολουθίας των CSPs.

Σε γενικές γραμμές η διαδικασία που ακολουθείται είναι η εξής: αρχικά γίνεται η δημιουργία του πρώτου CSP, της ακολουθίας των CSPs, με τυχαία ανάθεση τιμών και έπειτα η επίλυσή του με επαναλαμβανόμενη χρήση του αλγόριθμου Min-Conflicts. Στη συνέχεια γίνεται προσθήκη περιορισμών στο πρώτο CSP με αποτέλεσμα τη δημιουργία ενός *νέου* “δεύτερου” CSP. Σύμφωνα με την προηγούμενη λύση, του πρώτου CSP, ο Min-Conflicts με επαναλαμβανόμενο τρόπο και η χρήση σταθμισμένης αντικειμενικής συνάρτησης πραγματοποιείται αναζήτηση λύσης μηδενικής ή μικρής απόστασης από τη προηγούμενη. Η διαδικασία αυτή επίλυσης του δεύτερου CSP επαναλαμβάνεται μέχρι και την επίλυση του τελευταίου CSP της ακολουθίας των CSPs, στόχος η εύρεση συνεχόμενων λύσεων κοντινών μεταξύ τους.

Ο τροποποιημένος αλγόριθμος Min-Conflicts που χρησιμοποιείται υλοποιείται ως εξής:

5. Εισαγωγή παραμέτρων προβλήματος CSP προς επίλυση.
    - Αν το CSP είναι το πρώτο της ακολουθίας τότε γίνεται τυχαία ανάθεση των μεταβλητών
    - αλλιώς, ανάθεση τιμών των μεταβλητών με τη χρήση της λύσης του προηγούμενου CSP
6. Αν οι τιμές των μεταβλητών δεν παραβιάζουν τους περιορισμούς του προβλήματος τότε έχει βρεθεί λύση στο πρόβλημα. Επιστρέφει τις τιμές που ανατέθηκαν ως λύση, με το πλήθος των παραβιάσεων (conflicts) και την απόστασή (distance) της τρέχουσας λύσης με αυτής του προηγούμενου CSP(όπως είναι λογικό δεν μετριέται η απόσταση της λύσης του πρώτου CSP).

7. Επιλογή τυχαία, μιας μεταβλητής από το σύνολο των μεταβλητών που παραβιάζουν τους περιορισμούς. Εξετάζονται για τη μεταβλητή αυτή όλες οι δυνατές τιμές που μπορεί να πάρει.
    - Αν κάποια από τις τιμές της μεταβλητής, που εξετάστηκαν, μειώνει το πλήθος των περιορισμών που παραβιάζονται τότε ανέθεσε την τιμή στην μεταβλητή (νέα λύση).
    - Αν δεν υπάρχει η παραπάνω τιμή τότε επιλέγει μια τιμή που να διατηρεί το ίδιο πλήθος περιορισμών που παραβιάζονται (νέα λύση).
    - Αν δεν υπάρχουν οι παραπάνω τιμές, τότε διατηρεί την αρχική τιμή της εξεταζόμενης μεταβλητής.
8. Πήγαινε στο βήμα 3 (τόσες φορές όσες πλήθος των επαναλήψεων).
9. Αν υπάρχει προσωρινή βέλτιστη λύση σύγκρινε τη με τη λύση που προκύπτει στο βήμα 3, σύμφωνα με τις τιμές της σταθμισμένης αντικειμενικής συνάρτησης που επιτυγχάνουν η κάθε μια ξεχωριστά. Θέσε τη λύση με τη μικρότερη τιμή της αντικειμενικής συνάρτησης ως προσωρινά βέλτιστη.
10. Αλλιώς θέσε τη λύση ως προσωρινά βέλτιστη.
11. Πήγαινε στο βήμα 2 (τόσες φορές όσες πλήθος των επαναλήψεων).
12. Βέλτιστη λύση = προσωρινά βέλτιστη λύση
13. Επιστροφή βέλτιστης λύσης

Ο ψευδοκώδικας που περιγράφει των αλγόριθμο δύο φάσεων είναι ο εξής:

Function TwoPhase(*solv, *initSolv, *prevSolv, **constraints, ****constraint_matrices, d, n, repeat_times, repeat_Min_Conf, firstTime, firstTimeMinConf, Z, total_constraints, countMinConf, countRandVar, *previous, *SDistances, *SConflicts, problem, W1, W2)

Είσοδοι :

*solv, τρέχουσα λύση
*initSolv, αρχική λύση
*prevSolv, προηγούμενη λύση -> για την εύρεση της βέλτιστης λύσης
**constraints, πίνακας περιορισμών
****constraint_matrices, πίνακας περιορισμών που ικανοποιούνται
n, πλήθος των μεταβλητών
d, πεδίο τιμών των μεταβλητών
repeat_times, το πλήθος των τυχαίων μεταβλητών που ελέγχονται κάθε φορά που εκτελείται ο αλγόριθμος Min-Conflicts
repeat_Min_Conf, το πλήθος των επαναλήψεων του αλγόριθμου Min-Conflicts
firstTime, πρώτη φορά που εκτελείται ο αλγόριθμος δύο φάσεων
firstTimeMinConf, πρώτη φορά που εκτελείται ο αλγόριθμος Min-Conflicts
Z, μεταβλητή που εκφράζει την τιμή της ευριστικής συνάρτησης προς ελαχιστοποίηση ->Z=min(w1*normConflicts + w2*normDistance)
total_constraints, πλήθος περιορισμών

countMinConf, μετρητής επαναλήψεων του αλγόριθμου Min-Conflicts
countRandVar, μετρητής τυχαίων μεταβλητών που ελέγχονται
*previous, πίνακας τιμών - αποτελεσμάτων(παραβιάσεις,απόσταση....) προηγούμενης λύσης
*SDistances, πίνακας απόστασης κάθε CSP της ακολουθίας
*SConflicts, πίνακας παραβιάσεων κάθε CSP της ακολουθίας
problem, πλήθος προβλημάτων CSPs ακολουθίας
W1, συντελεστής βαρύτητας παραβιάσεων
W2, συντελεστής βαρύτητας απόστασης

Αρχικοποίηση μετρητών - παραμέτρων

// χρήση προηγούμενης λύσης προς αρχικοποίηση του Min_Conflicts....Ανάθεση τιμών -> Προηγούμενη Λύση

Αν firstTime = false
Τότε for i = 0 to n -> initSolv[i] = solv[i]

While -> repeat_Min_Conf ≠ 0
countMinConf++
rounds = repeat_times
Αν firstTime = true // Αρχική Τυχαία ανάθεση τιμών στις μεταβλητές
Τότε for i = 0 to n solv[i] = (int) (((float)d)*rand()/(RAND_MAX+1.0))
for(int i = 0; i < n; i++) initSolv[i] = solv[i] // Αρχική λύση

//Έλεγχος αν παραβιάζονται οι περιορισμοί
check_values(solv, constraints, constraint_matrices, n, d)
//Υπολογισμός ευριστικής συνάρτησης
Z = CalcZ(conflicts, total_constraints, distance, n, W1, W2)
Αν Z=0 Τότε SDistances[problem] = distance
SConflicts[problem] = conflicts
Επιστροφή Z

While -> rounds ≠ 0
countRandVar++
for i = 0 to n temp[i] = solv[i]
Αν conflicts ≠ 0
//Επιλογή τυχαίας μεταβλητής
rvar = selectRandVar(solv, initSolv, previousvar, constraints, constraint_matrices, n, d);

// τυχαία μεταβλητή που ελέγχθηκε
previousvar = rvar;

//Εξέτασε για την τυχαία μεταβλητή όλες τις δυνατές τιμές
For i = 0 to d
temp[rvar] = i;
tempconflicts = check_values(temp, constraints, constraint_matrices, n, d);
Αν tempconflicts < conflicts ή
tempconflicts = conflicts

// Επιλογή Λύσης με λιγότερες Παραβιάσεις
conflicts = tempconflicts;
solv[rvar] = i;

```
                //Έλεγχος απόστασης από προηγούμενη λύση
                distance = CountDistance(initSolv, solv, n);

                Z = CalcZ( conflicts, total_constraints, distance, n, W1, W2);

                Αν conflicts=0 και distance=0 => Z=0 βρέθηκε ΛΥΣΗ
                        //Καταχώρηση λύσης
                        SDistances[problem] = distance;
                        SConflicts[problem] = conflicts;
                        Επιστροφή Z

                rounds = rounds - 1;


                //Βέλτιστη Λύση μεταξύ των επαναλήψεων του "Min_Conflicts"
                bestSolution(prevSolv, solv, firstTimeMinConf, Z, prevZ, conflicts,
                                total_constraints, distance, n, W1, W2, countMinConf,
                                countRandVar, previous);

                repeat_Min_Conf = repeat_Min_Conf - 1;

                firstTimeMinConf = false;
                firstTime = false;


// Αναζήτηση της καλύτερης λύσης
Αν  Z >  prevZ[0]
                For  i = 0 to n  solv[i] = prevSolv[i]

                conflicts = previous[0];            // prevconflicts=previous[0]
                distance = previous[1];             // prevdistance=previous[1],
                countMinConf = previous[2];     // prevcountMinConf=previous[2]
                countRandVar = previous[3];     // prevcountRandVar=previous[3]
                Z = prevZ[0];


SConflicts[problem] = conflicts;
SDistances[problem] = distance;

// Χρήση της λύσης που βρέθηκε σε επόμενο πρόβλημα
For  i = 0  to n   initSolv[i] = solv[i]

Επιστροφή Z;
```

Η διαδικασία επίλυσης DCSP της μεθόδου ακολουθεί τα παρακάτω βήματα:

Πρώτη φάση: Εύρεση αρχικής “καλής” λύσης που λαμβάνει υπόψη μόνο ένα στόχο (αρχικό CSP της ακολουθίας του DCSP), βελτιστοποίηση ενός μόνο κριτηρίου, με την

επαναλαμβανόμενη χρήση του min-conflicts (τροποποιημένου ώστε να αποφεύγονται οι τοπικές λύσεις).

Δεύτερη φάση: Η πρώτη φάση παρέχει τη λύση "έναρξης" για τη δεύτερη φάση. Ξεκινά από αυτή τη λύση για την αναζήτηση μη- επικρατέστερων λύσεων, μια σειρά από διαφορετικές μορφές του προβλήματος με βάση των ομαδοποιήσεων των κριτηρίων. Στην οποία ο τοπικός αλγόριθμος αναζήτησης Min-Conflicts εφαρμόζεται σε μια ακολουθία διαφορετικών συνόλων από στόχους (CSPs), όπου κάθε σύνολο μετατρέπει το πρόβλημα σε ένα ενιαίο κριτήριο. Σημαντικό για τη δεύτερη φάση είναι ότι τα διαδοχικά σύνολα και τοπικές αναζητήσεις αντιμετωπίζονται ως μια αλυσίδα:

Ένα σύνολο a_i+1 τροποποιεί ελαφρώς την έμφαση που δόθηκε στους διαφορετικούς στόχους όταν συγκριθεί με το σύνολο a_i . Η τοπική αναζήτηση για το σύνολο a_i+1 ξεκίνησε από την τοπική βέλτιστη λύση s_i* που επιστράφηκε από το σύνολο a_i. Το βασικό κίνητρο για μια τέτοια προσέγγιση είναι η αξιοποίηση της αποτελεσματικότητας των αλγόριθμων τοπικής αναζήτησης για τα μοναδικά προβλήματα στόχου.

Στην παρούσα εργασία υποθέτουμε ότι η ομαδοποίηση πολυκριτηριακών προβλημάτων σε διάφορα μονοκριτηριακά προβλήματα βασίζεται στη χρήση μιας σταθμισμένης αντικειμενικής γραμμικής ευριστικής συνάρτησης με κανονικοποιημένα τα διανύσματα βάρους της. Στην περίπτωση αυτή η χρησιμότητα μιας λύσης αξιολογείται από το πλήθος των παραβιάσεων και την απόσταση των μεταξύ τους λύσεων σε κανονικοποιημένη μορφή (1)(2) σύμφωνα με τις τιμές των βαρών της ευριστικής συνάρτησης (3).

$$conflicts = \frac{\text{Πλήθος από ζεύγη περιορισμών που παραβιάζονται}}{\text{Ζεύγη περιορισμών}} \quad (1)$$

$$distance = \frac{\text{πλήθος μεταβλητών με διαφορετική ανάθεση τιμή ςαπό προηγούμενη λύση}}{\text{πλήθος μεταβλητών}} \quad (2)$$

$$\mathrm{f}(\mathrm{w1}, \mathrm{w2}) \;=\; \mathrm{w1} * \mathrm{conflicts} \;+\; \mathrm{w2} * \mathrm{distance} \quad (3)$$

Η μέθοδος με την αντικειμενική συνάρτηση τροποποιεί δυναμικά την κατεύθυνση της αναζήτησης κατά τη διαδικασία της αναζήτησης. Ώστε να αποκομισθεί κατά προσέγγιση ένα καλό σύνολο από αποτελεσματικές λύσεις.

Κεφάλαιο 5: Πειραματικά Αποτελέσματα

Για την πειραματική διερεύνηση των αλγορίθμων χρησιμοποιείται τυχαία δημιουργία δυαδικών CSPs.

Ένα DCSP δημιουργείται με τον καθορισμό – αρχικοποίηση των παρακάτω παραμέτρων:

var : πλήθος μεταβλητών προβλήματος.

val: πεδίο τιμών κάθε μεταβλητής. Για λόγου απλοποίησης του προβλήματος, θεωρούμε ότι όλες οι μεταβλητές έχουν το ίδιο πεδίο τιμών.

start_con: πλήθος περιορισμών αρχικού CSP.

sats_con: ποσοστό περιορισμών που ικανοποιούνται.

times_MC: πλήθος επαναλήψεων του αλγόριθμουmin-conflicts.

add_con: πλήθος περιορισμών που προστίθενται σε κάθε CSP(πέραν του αρχικού).

Όλα τα DCSP, προς διερεύνηση, δημιουργούνται με τη χρήση των ίδιων παραμέτρων, γεγονός που τα χαρακτηρίζει ως προβλήματα της ίδιας κλάσης.

Το ενδιαφέρον των πειραμάτων προς σύγκριση των δύο μεθόδων με κύριο κριτήριο τη σταθερότητα (stability) στην εύρεση λύσης. Επικεντρώνεται στο πλήθος των CSPs για τα οποία έχει βρεθεί συνεχόμενη λύση, στο κρίσιμο σημείο για το ποιο ο αλγόριθμος αδυνατεί να βρει λύση (όπου σταματά η συνέχεια των λύσεων), στη μείωση της απόστασης μεταξύ των συνεχόμενων λύσεων, στην εύρεση λύσης συνεχόμενη των προηγούμενων λύσεων.

Μετά από αναζήτηση, μέσω μεγάλου πλήθους δοκιμών σε τιμές των παραμέτρων, οδηγηθήκαμε για την δημιουργία δύσκολων DCSP στις παραμέτρους που ακολουθούν. Δύσκολα DCSPs χαρακτηρίζονται τα προβλήματα για τα οποία με την προσθήκη μικρού πλήθους περιορισμών αυξάνεται ραγδαία η δυσκολία επίλυσης τους.

var = 50
val = 9
start_con = 71
sats_con= 59%
times_MC= 200
add_con= 5

Σύμφωνα με τις παραπάνω παραμέτρους εξάγονται τα αποτελέσματα του πίνακα που ακολουθεί.

Για την ανάλυση, σύγκριση και εξαγωγή συμπερασμάτων σχετικά με την επίλυση των DCSPs με τη χρήση των δύο μεθόδων χρησιμοποιήθηκαν οι εξής μετρήσεις:

Total solved Problems: πλήθος συνεχόμενων λύσεων (conflicts=0) από την επίλυση του αρχικού CSP και μετά.

Κρίσιμο σημείο conflicts: πλήθος παραβιάσεων περιορισμών στο κρίσιμο σημείο. Κρίσιμο σημείο θεωρείται το σημείο (CSP) της ακολουθίας των CSPs όπου η μέθοδος επίλυσης αδυνατεί να βρει λύση.

Κρίσιμο σημείο distance: απόσταση του κρίσιμου σημείου από την προηγούμενη λύση.

Πριν το κρίσιμο σημείο avg distance: ο μέσος όρος τη απόστασης των συνεχόμενων λύσεων.

40 CSPs: πλήθος από CSP προς επίλυση (μήκος της ακολουθίας των CSPs του DCSP).

- Avg confl. -> μέσος όρος παραβιάσεων που προκύπτουν από την επίλυση των 40 CSPs.
- Avg dist. -> μέσος όρος απόστασης μεταξύ των λύσεων των 40 CSPs.

Συντελεστές: σταθμισμένης ευριστικής συνάρτησης της μεθόδου δύο φάσεων:

- W1 -> συντελεστής βαρύτητας παραβιάσεων
- W2 -> συντελεστής βαρύτητας απόστασης

DCSP	Min-conflicts						Two phase Min-conflicts						
	Total solved Problems	Πριν το κρίσιμο σημείο avg distance	κρίσιμο σημείο conflict	κρίσιμο σημείο distance	40 CSPs		Total solved Problems	συντελεστές	Πριν το κρίσιμο σημείο avg distance	κρίσιμο σημείο conflicts	κρίσιμο σημείο distance	40 CSPs	
					Avg confl.	Avg dist.						avg confl	avg dist.
1	2	2	1	4	8.179	1.923	2	W1=0.9 W2=0.1	2	1	4	7.846	1.897

2	6	3.2	1	2	11.18	2.385	6	W1=0.95 W2=0.05	3.2	1	2	7.231	2.615
3	4	2	1	2	11.21	1.821	4	W1=0.9 W2=0.1	2	1	2	10.8	1.615
4	1	-	1	5	7.462	2.154	1	W1=0.95 W2=0.05	-	1	5	6.718	2
5	3	2	1	4	8.282	2.59	3	W1=0.95 W2=0.05	2	1	4	7.744	2.513
6	8	1.571	1	1	8.538	2	8	W1=0.95 W2=0.05	1.571	1	1	8.538	1.897
7	5	2.5	1	2	9	2.308	5	W1=0.95 W2=0.05	2.5	1	2	8.026	1.872
8	4	3	1	2	9.462	1.769	4	W1=0.9 W2=0.1	3	1	1	7.897	1.538
9	1	-	1	2	7.923	1.795	1	W1=0.85 W2=0.15	-	1	2	9.692	1.513
10	6	1.2	1	2	7.308	2.179	6	W1=0.95 W2=0.05	1.2	1	1	7	2.231
11	4	2.333	2	7	9.179	1.897	4	W1=0.95 W2=0.05	2.333	2	3	9.128	1.821
12	5	2	1	2	9.103	2.641	5	W1=0.95 W2=0.05	2	1	2	8.641	2.256
13	1	-	1	0	9.564	2.103	1	W1=0.9 W2=0.1	-	1	0	9.974	1.256
14	8	2.857	1	0	8.821	2.103	8	W1=0.9 W2=0.1	2.875	1	0	10.51	1.538
15	3	3	1	4	8.077	2.205	3	W1=0.95 W2=0.05	3	1	4	8	2.026
16	6	3	1	2	9.282	2	6	W1=0.85 W2=0.15	3	1	2	7.923	1.667
17	12	2.545	1	0	9.41	2.103	12	W1=0.9 W2=0.1	2.545	1	0	8.949	1.949
18	4	2	1	2	7.897	2.513	4	W1=0.9 W2=0.1	2	1	2	7.564	2.026

19	4	0.667	1	3	11.95	2.231	4	W1=0.95 W2=0.05	0.667	1	3	8.41	1.923
20	1	-	1	2	8.103	2.051	1	W1=0.85 W2=0.15	-	1	1	8.821	1.718
21	11	2.3	1	0	10.36	2.128	11	W1=0.95 W2=0.05	2.3	1	0	10.36	2.077
22	6	1.8	1	5	6.231	2.256	6	W1=0.95 W2=0.05	1.8	1	5	6.205	2
23	3	4.5	1	3	9.487	1.949	3	W1=0.9 W2=0.1	4.5	1	2	9	1.744
24	4	5.333	2	1	8.718	2.795	4	W1=0.85 W2=0.15	5.333	2	0	9.256	1.974
25	9	3.25	1	5	7.385	2.282	9	W1=0.9 W2=0.1	3.25	1	5	7.872	2.231
26	5	2	1	2	7.846	2.154	5	W1=0.95 W2=0.05	2	1	2	8.59	1.769
27	8	3.571	1	3	7.154	2.59	8	W1=0.95 W2=0.05	3.571	1	3	7.179	2.513
28	2	2	1	4	9.128	2.205	2	W1=0.95 W2=0.05	2	1	3	9.462	1.795
29	2	3	1	3	10.26	2.282	2	W1=0.95 W2=0.05	3	1	3	9.538	2.077
30	3	2.5	1	2	8.026	2.154	3	W1=0.95 W2=0.05	2.5	1	2	8.308	2.077
31	4	1	1	0	7.872	2.026	4	W1=0.9 W2=0.1	1	1	0	7.077	1.538
32	3	5	1	0	8.436	2.333	3	W1=0.95 W2=0.05	5	1	0	9.667	2.282
33	6	1.8	1	2	9.564	1.846	6	W1=0.95 W2=0.05	1.8	1	1	8.513	1.744
34	4	2.333	1	2	6.487	2.026	4	W1=0.95 W2=0.05	2.333	1	1	6.128	2.179
35	4	2.333	1	0	7.769	1.744	4	W1=0.9 W2=0.1	2.333	1	0	6.949	2.077

36	1	-	1	1	6.538	2.128	1	W1=0.95 W2=0.05	-	1	1	6.308	2.026
37	3	2.5	1	2	12.49	1.974	3	W1=0.95 W2=0.05	2.5	1	1	11.08	1.769
38	2	3	1	4	8.769	2.179	2	W1=0.9 W2=0.1	3	1	4	6.641	2.051
39	6	2.4	1	7	8.821	2.744	6	W1=0.9 W2=0.1	2.4	1	6	8.513	2.154
40	3	2	2	2	9.103	2.026	3	W1=0.85 W2=0.15	2	2	2	9.308	1.769
41	5	1.75	1	1	9.051	2.179	5	W1=0.95 W2=0.05	1.75	1	1	9.051	2.179
42	5	1.75	1	3	10.56	2.128	5	W1=0.9 W2=0.1	1.75	1	3	10.03	2.077
43	1	-	1	2	9.744	2	1	W1=0.85 W2=0.15	-	1	2	8.949	1.692
44	1	-	1	3	10.31	1.769	1	W1=0.95 W2=0.05	-	1	3	10.21	1.718
45	4	4.667	1	3	9.59	1.795	4	W1=0.95 W2=0.05	4.667	1	2	9.564	2.128
46	3	1.5	1	3	9.256	1.923	3	W1=0.9 W2=0.1	1.5	1	3	9.179	1.769
47	5	1.75	1	1	8.538	2.103	5	W1=0.9 W2=0.1	1.75	1	1	8.615	2
48	3	3	1	2	11.03	2.179	3	W1=0.9 W2=0.1	3	1	2	9.077	1.795
49	5	2.25	2	3	7.205	2.231	5	W1=0.95 W2=0.05	2.25	2	2	7.615	1.744
50	2	0	1	1	8.026	2.333	2	W1=0.9 W2=0.1	0	1	1	7.718	2.128

Επιπλέον τιμές παραμέτρων που οδηγούν σε δημιουργία δύσκολων DCSPs είναι:

var = 70
val = 9

start_con = 72

sats_con= 54%

times_MC= 200

add_con= 5

Σύμφωνα με τις παραπάνω παραμέτρους εξάγονται τα ακόλουθα αποτελέσματα :

DCSP	Min-conflicts						Two phase Min-conflicts						
	Total solved Problems	Πριν το κρίσιμο σημείο avg distance	κρίσιμο σημείο conflicts	κρίσιμο σημείο distance	40 CSPs		Total solved Problems	συντελεστές	Πριν το κρίσιμο σημείο avg distance	κρίσιμο σημείο conflicts	κρίσιμο σημείο distance	40 CSPs	
					Avg confl.	Avg dist.						Avg confl	Avg dist.
1	3	2.5	1	6	7.974	3.282	3	W1=0.9 W2=0.1	2.5	1	5	6.897	2.769
2	5	2.75	1	1	10.54	2.282	5	W1=0.9 W2=0.1	2.75	1	1	9.949	2.205
3	7	2.667	2	3	6.821	2.205	7	W1=0.85 W2=0.15	2.667	2	3	7.333	2.154
4	1	-	1	4	7.41	2.641	1	W1=0.9 W2=0.1	-	1	2	8.897	1.974
5	2	3	1	3	9.462	2.513	2	W1=0.9 W2=0.1	3	1	2	9.41	2.462
6	2	1	2	2	7.538	1.974	2	W1=0.95 W2=0.05	1	2	1	7.538	1.974
7	7	1.333	1	3	8.692	1.974	7	W1=0.9 W2=0.1	1.333	1	3	8.333	1.974
8	2	0	1	1	8.179	2.513	2	W1=0.9 W2=0.1	0	1	1	7.462	2.205
9	4	3	1	3	7.692	3	4	W1=0.85	3	1	2	7.774	2.615

								W2=0.15					
10	4	3	1	3	8.154	2.41	4	W1=0.95 W2=0.05	3	1	3	7.385	2.179
11	6	2.6	1	1	4.359	2.026	6	W1=0.95 W2=0.05	2.6	1	1	3.897	2.026
12	7	2.5	1	5	6.692	2.359	7	W1=0.95 W2=0.05	2.5	1	4	6.846	2.128
13	7	2.167	1	3	8	2.128	7	W1=0.85 W2=0.15	2.167	1	3	8.744	1.872
14	7	2	1	2	8.282	2.154	7	W1=0.95 W2=0.05	2	1	2	8.282	2.103
15	10	2.889	1	2	9.744	2.692	10	W1=0.95 W2=0.05	2.889	1	2	7.667	2.538
16	6	3	1	3	9.333	2.846	6	W1=0.9 W2=0.1	3	1	2	7.231	2.795
17	7	2.167	1	2	6.974	2.282	7	W1=0.95 W2=0.05	2.167	1	2	6.462	2.667
18	2	1	1	4	7.897	2.487	2	W1=0.95 W2=0.05	1	1	2	7.821	2.41
19	5	3.25	1	0	9.205	2.795	5	W1=0.9 W2=0.1	3.25	1	0	9.051	2.718
20	7	2.5	1	2	6.718	2.795	7	W1=0.95 W2=0.05	2.5	1	2	5.462	2.564
21	6	2	1	6	8.513	2.436	6	W1=0.95 W2=0.05	2	1	6	8.308	2.41
22	6	1.8	1	3	8.436	2.795	6	W1=0.95 W2=0.05	1.8	1	3	7.821	2.487
23	1	-	1	5	9.744	2.897	1	W1=0.9 W2=0.1	-	1	5	8.974	2.846
24	9	2.375	1	2	6.718	2.077	9	W1=0.95 W2=0.05	2.375	1	2	6.641	2.051
25	2	2	1	3	9.462	2.846	2	W1=0.85 W2=0.15	2	1	3	8.769	2.462
26	2	1	1	2	9.872	2.077	2	W1=0.8	1	1	2	8.897	1.615

								W2=0.2					
27	2	2	1	4	13	2.923	2	W1=0.8 W2=0.2	2	1	4	8.744	1.692
28	2	1	1	5	10.72	2.769	2	W1=0.85 W2=0.15	1	1	5	9.974	3
29	4	2.667	1	2	8.41	2.564	4	W1=0.95 W2=0.05	2.667	1	2	8.154	2.359
30	6	2.4	1	1	9.59	2.385	6	W1=0.9 W2=0.1	2.4	1	1	9.256	2.256
31	3	3	1	5	8	2.256	3	W1=0.95 W2=0.05	3	1	5	6.949	2.641
32	2	3	1	2	10.13	3.103	2	W1=0.85 W2=0.15	3	1	2	8.897	2.769
33	5	2.75	1	5	7.769	2.487	5	W1=0.85 W2=0.15	2.75	1	5	8.385	2.231
34	10	2.667	1	1	7.846	2.641	10	W1=0.8 W2=0.2	2.667	1	1	8.077	1.897
35	3	1.5	1	4	9.949	2.359	3	W1=0.9 W2=0.1	1.5	1	4	8.179	2.564
36	3	2.5	1	2	8.026	2.718	3	W1=0.9 W2=0.1	2.5	1	1	8.513	2.308
37	2	2	1	0	10.54	1.846	2	W1=0.85 W2=0.15	2	1	0	9.564	1.744
38	9	2.125	2	3	6.436	2.692	9	W1=0.95 W2=0.05	2.125	2	1	6.667	2.436
39	7	2.167	1	6	9.41	2.846	7	W1=0.95 W2=0.05	2.167	1	6	8.667	2.744
40	8	3	1	4	7.718	2.974	8	W1=0.9 W2=0.1	3	1	4	8.051	2.462
41	6	1.6	1	1	10.15	1.949	6	W1=0.9 W2=0.1	1.6	1	1	7.615	2.205
42	5	2.25	1	4	8.949	2.795	5	W1=0.85 W2=0.15	2.25	1	4	7.897	2.308
43	4	2	1	3	8.231	2.59	4	W1=0.9	2	1	2	10.15	2.026

								W2=0.1					
44	6	2	1	2	9.564	2.256	6	W1=0.9 W2=0.1	2	1	2	9.564	2.256
45	6	1.2	1	3	6.128	2.615	6	W1=0.9 W2=0.1	1.2	1	3	6.333	2.487
46	8	2.174	1	2	4.718	2.41	8	W1=0.95 W2=0.05	2.714	1	2	4.641	2.487
47	8	1.857	1	3	8.487	2.641	8	W1=0.85 W2=0.15	1.857	1	3	8.949	2.487
48	1	-	1	2	8.256	2.897	1	W1=0.8 W2=0.2	-	1	1	8.769	1.615
49	5	1.5	1	2	6.538	1.718	5	W1=0.75 W2=0.25	1.5	1	2	10.41	1.308
50	1	-	2	2	7.333	2.359	1	W1=0.95 W2=0.05	-	2	2	6.744	1.872

Στις παραπάνω μετρήσεις καθορίζεται ως βήμα αλλαγής των συντελεστών βαρών τιμή ίση με 0.05.

Από τα παραπάνω αποτελέσματα διαπιστώνεται ότι οι μέθοδοι βρίσκουν λύση σε τουλάχιστον ένα CSP της ακολουθίας των δύσκολων DCSPs. Παρατηρείται ότι η μέθοδος τοπικής αναζήτησης δύο φάσεων αδυνατεί να βρει νέα συνεχόμενη λύση, να μειώσει την απόσταση μεταξύ των συνεχόμενων λύσεων σε σχέση με την απλή μέθοδο τοπικής αναζήτησης. Παρόλα αυτά, η μέθοδος δύο φάσεων οδηγεί σε μερικές περιπτώσεις σε μικρή “βελτίωση” του κρίσιμου σημείου, μικρή μείωση της απόστασης και των περιορισμών από την προηγούμενη λύση. Καθώς επίσης αξιοσημείωτη είναι η βελτίωση (μείωση της απόστασης και των περιορισμών) που επιτυγχάνει σε προβλήματα μετά από το κρίσιμο σημείο όπου σε αρκετές περιπτώσεις η μέθοδος οδηγεί σε εύρεση νέων λύσεων.

Παρατηρώντας τους συντελεστές των βαρών, διακρίνεται ότι η λύση που προτείνει ο αλγόριθμος δύο φάσεων εντοπίζεται σε τιμές των συντελεστών που κυμαίνονται αντίστοιχα για w1 = 1 έως 0.8 και w2 = 0 έως 2. Στα παρακάτω αποτελέσματα διερευνάται, στο τι επιπτώσεις έχει, αν το βήμα αλλαγής συντελεστών από 0.05 μειωθεί σε 0.005.

DCSP	Two phase Min-conflicts – βήμα 0.05							Two phase Min-conflicts– βήμα 0.005						
	Total solved Problems	συντελεστές	Πριν το κρίσιμο σημείο avgdistance	κρίσιμο σημείο conflicts	κρίσιμο σημείο distance	40 CSPs		Total solved Problems	συντελεστές	Πριν το κρίσιμο σημείο avgdistance	κρίσιμο σημείο conflicts	κρίσιμο σημείο distance	40 CSPs	
						Avgconfl	Avg dist.						Avgconfl	Avg dist.
1	3	W1=0.9 W2=0.1	2.5	1	5	6.897	2.769	3	W1=0.995 W2=0.005	2.5	1	5	6.897	2.769
2	5	W1=0.9 W2=0.1	2.75	1	1	9.949	2.205	5	W1=0.91 W2=0.09	2.75	1	1	**9,641**	2.205
3	7	W1=0.85 W2=0.15	2.667	2	3	7.333	2.154	7	W1=0.86 W2=0.14	2,667	2	3	**7,308**	2,154
4	1	W1=0.9 W2=0.1	-	1	2	8.897	1.974	1	W1=0.915 W2=0.085	-	1	2	8,897	1,974
5	2	W1=0.9 W2=0.1	3	1	2	9.41	2.462	2	W1=0.995 W2=0.005	3	1	2	9,41	2,462
6	2	W1=0.95 W2=0.05	1	2	1	7.538	1.974	2	W1=0.995 W2=0.005	2	2	1	**7,513**	2
7	7	W1=0.9 W2=0.1	1.333	1	3	8.333	1.974	7	W1=0.995 W2=0.005	1.333	1	3	**8.308**	2.231
8	2	W1=0.9 W2=0.1	0	1	1	7.462	2.205	2	W1=0.995 W2=0.005	0	1	1	7.462	2.205
9	4	W1=0.85 W2=0.15	3	1	2	7.774	2.615	4	W1=0.85 W2=0.015	3	1	2	7.744	2.615
10	4	W1=0.95 W2=0.05	3	1	3	7.385	2.179	4	W1=0.835 W2=0.165	3	1	3	**6.897**	2.308
11	6	W1=0.80 W2=0.2	2.6	1	1	3.769	1.641	6	W1=0.80 W2=0.2	2.6	1	1	3.769	1.641
1	7	W1=0.95	2.5	1	4	6.846	2.128	7	W1=0.995	2.5	1	4	6.846	2.128

2		W2=0.05							W2=0.005					
13	7	W1=0.85 W2=0.15	2.167	1	3	8.744	1.872	7	W1=0.855 W2=0.145	2.167	1	3	**8.59**	1.846
14	7	W1=0.95 W2=0.05	2	1	2	8.282	2.103	7	W1=0.865 W2=0.135	2	1	2	**8.154**	2.103
15	10	W1=0.95 W2=0.05	2.889	1	2	7.667	2.538	10	W1=0.995 W2=0.005	2.889	1	2	7.667	2.538
16	6	W1=0.9 W2=0.1	3	1	2	7.231	2.795	6	W1=0.9 W2=0.1	3	1	2	7.231	2.795
17	7	W1=0.95 W2=0.05	2.167	1	2	6.462	2.667	7	W1=0.995 W2=0.005	2.167	1	2	6.462	2.667
18	2	W1=0.95 W2=0.05	1	1	2	7.821	2.41	2	W1=0.89 W2=0.11	1	1	2	**7.513**	2.436
19	5	W1=0.9 W2=0.1	3.25	1	0	9.051	2.718	5	W1=0.9 W2=0.1	3.25	1	0	9.051	2.718
20	7	W1=0.95 W2=0.05	2.5	1	2	5.462	2.564	7	W1=0.995 W2=0.005	2.5	1	2	5.462	2.564
21	6	W1=0.95 W2=0.05	2	1	6	8.308	2.41	6	W1=0.995 W2=0.005	2	1	6	8.308	2.41
22	6	W1=0.95 W2=0.05	1.8	1	3	7.821	2.487	6	W1=0.895 W2=0.105	1.8	1	3	**7.538**	2.308
23	1	W1=0.9 W2=0.1	-	1	5	8.974	2.846	1	W1=0.915 W2=0.085	-	1	5	8.974	2.846
24	9	W1=0.95 W2=0.05	2.375	1	2	6.641	2.051	9	W1=0.995 W2=0.005	2.375	1	2	6.641	2.051
25	2	W1=0.85 W2=0.15	2	1	3	8.769	2.462	2	W1=0.825 W2=0.175	2	1	3	**7.538**	2.128

26	2	W1=0.8 W2=0.2	1	1	2	8.897	1.615	2	W1=0.83 W2=0.17	1	1	2	**8.615**	1.744
27	2	W1=0.8 W2=0.2	2	1	4	8.744	1.692	2	W1=0.815 W2=0.185	2	1	4	**8.077**	1.846
28	2	W1=0.85 W2=0.15	1	1	5	9.974	3	2	W1=0.87 W2=0.13	1	1	5	9.974	3
29	4	W1=0.95 W2=0.05	2.667	1	2	8.154	2.359	4	W1=0.995 W2=0.005	2.667	1	2	8.154	2.359
30	6	W1=0.9 W2=0.1	2.4	1	1	9.256	2.256	6	W1=0.915 W2=0.085	2.4	1	1	**9.205**	2.231

Με την μείωση του βήματος αλλαγής των συντελεστών βαρών w1 και w2 παρατηρείται περαιτέρω βελτίωση των λύσεων σε προβλήματα μετά του κρίσιμου σημείου.

Συμπεράσματα – Μελλοντική Εργασία

Στην εργασία αυτή ασχοληθήκαμε με την επίλυση δύσκολων DCSPs χρησιμοποιώντας δύο μεθόδους τοπικής αναζήτησης. Οι μέθοδοι βασίζονται στις δυνατότητες που παρέχει ο αλγόριθμος min-conflicts. Η πρώτη μέθοδος που εξετάστηκε αποδίδει τις περισσότερες φορές λύση σε τουλάχιστον ένα CSP της ακολουθίας CSPs του εκάστοτε DCSP, χωρίς να λαμβάνει υπόψη τη απόσταση μεταξύ των λύσεων που βρίσκει. Το γεγονός αυτό χαρακτηρίζει την σταθερότητα (stability) της μεθόδου στην εύρεση λύσεων. Στη σταθερότητα αυτή στοχεύει η δεύτερη μέθοδος. Η δεύτερη μέθοδος προσπαθεί να αξιοποιήσει την πληροφορία που προσφέρεται από τη λύση του “προηγούμενου προβλήματος” με αλυσιδωτή διαδικασία μέσω της ευριστικής αντικειμενικής γραμμικής συνάρτησής της. Η μέθοδος αυτή παρουσιάζει αδυναμία στη βελτίωση της απόστασης των συνεχόμενων λύσεων των αρχικών CSPs της ακολουθίας. Παρά το γεγονός αδυναμίας παρουσιάζονται αξιοσημείωτες βελτιώσεις σε λύσεις μετά το κρίσιμο σημείο που οδηγούν σε αρκετές περιπτώσεις σε εύρεση νέων λύσεων. Επιπλέον καθοριστικής σημασίας σε αρκετές περιπτώσεις εμφανίζεται το βήμα αλλαγής των συντελεστών της αντικειμενικής γραμμικής συνάρτησής της.

Σύμφωνα με τη παραπάνω διερεύνηση και τη πολυκριτηριακή φύση των DCSP εμφανίζονται περαιτέρω περιοχές προς έρευνα για βελτίωση της μεθόδου τοπικής αναζήτησης δύο φάσεων. Όπως διερεύνηση για πιο αποδοτική ευριστική συνάρτηση με αποτελεσματικότερη χρήση της πληροφορίας “προηγούμενων λύσεων” και περαιτέρω δοκιμές στις τιμές των παραμέτρων. Στην παρούσα εργασία ως DCSP θεωρήθηκε η προσθήκη σταθερού πλήθους περιορισμών σε μια ακολουθία CSPs, σαν μελλοντική εργασία κρίνεται η συμπεριφορά της μεθόδου σε περιπτώσεις αφαίρεσης ή συνδυασμού (πρόσθεσης - αφαίρεσης) σταθερού ή μεταβαλλόμενου πλήθους περιορισμών

Επιπλέον με την παραπάνω εργασία προκύπτουν τα εξής ερωτήματα:

Υπάρχει η δυνατότητα διάσπασης ενός CSP σε επιμέρους προβλήματα με την κατηγοριοποίηση των περιορισμών και αν αυτή είναι αποτελεσματική;

Ποια είναι η «πληροφορία» που προσφέρεται και πώς μπορεί να χρησιμοποιηθεί;

Βιβλιογραφία

1. “Problem-Structure vs. Solution-Based Methods for Solving Dynamic Constraint Satisfaction Problems” Richard j. Wallace and Diarmuid Grimes.
2. “Solving Dynamic Constraint Satisfaction Problems: Relations between Problem Alteration and Search Performance” Richard j. Wallace, Diarmuid Grimes and Eugene C. Freuder.
3. “A Two-Phase Local Search for the Biobjective Traveling Salesman Problem” Luis Paquete and Thomas Stützle.
4. “Solution Reuse in Dynamic Constraint Satisfaction Problems” Gerard Verfaillie and Thomas Schiex.
5. http://el.wikipedia.org/
6. http://journals.cambridge.org/action/displayFulltext?type=1&fid=36068&jid=KER&volumeId=14&issueId=03&aid=36067
7. Kumar, V., “Algorithms for Constraint Satisfaction Problems: A Survey, AI Magazine” 13(1): 32-44, 1992.
8. Montanary, U.: Networks of constraints fundamental properties and applications to picture processing, in: Information Sciences 7: 95-132, 1974.
9. Haralick, R.M., Elliot, G.L.: Increasing tree search efficiency for constraint satisfaction problems, in: Artificial Intelligence 14:263-314, 1980.
10. Nilsson, N.J.: Principles of Artificial Intelligence, Tioga, Palo Alto, 1980.
11. Tsang, E.: Foundations of Constraint Satisfaction, Academic Press, London, 1995.
12. Lawler, E.W., Wood, D.E.: Branch-and-bound methods: a survey, in: Operations Research 14:699-719, 1966.
13. “Artificial Intelligence – A Modern Approach” Stuart Russell & Peter Norvig, Second Edition, 2003.
14. Freuder, E.C., Wallace, R.J.: Partial Constraint Satisfaction, in: Artificial Intelligence, 1-3(58): 21-70, 1992.
15. “Hard, flexible and dynamic constraint satisfaction” IAN MIGUEL and QIANG SHEN
16. “Approximating the Pareto Front of Multi-Criteria Optimization Problems” Julien Legriel, Colas Le Guernic, Scott Cotton, and Oded Maler

Παράρτημα

Στο σημείο αυτό παραθέτονται οι κώδικες υλοποίησης των μεθόδων σε γλώσσα προγραμματισμού C++.

// Μέθοδος τοπικής αναζήτησης επίλυσης DCSP

```
#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <stdlib.h>
#include <math.h>

using namespace std;

int intcomp(void const *ip, void const *jp)
{
                        int i = *((int const *)(ip));
                        int j = *((int const *)(jp));
                        return i-j;
}


void zeroita(int **array, int nrows, int ncolumns){
        int i, j;
        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncolumns; j++)      array[i][j] = 0;
        }
}
void zeroitb(int ****array, int nrows, int ncolumns, int var_a, int var_b){
        int i, j=0;
        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncolumns; j++){
                        //printf("var_a=%d  var_b=%d  i=%d  j=%d \n", var_a,var_b,i,j);
                        array[var_a][var_b][i][j] = 0;
                        //printf("array[%d][%d][%d][%d] = %d \n", var_a,var_b,i,j,array[var_a][var_b][i][j]);
                }
        }
}

void generate_constraints(int **constraints, int n, int k, int total_constraints)
{
  int i, j, l, *temp;
  char good;

  temp = (int *)malloc(k*sizeof(int));

  //loop that generates constraints
  for (i=0; i<total_constraints; i++) {

     good=0;

     //loop that generates a set of variables until a set that has not been generated before
     while (!good) {

        for (j=0; j<k; j++) temp[j]=-1;

     //loop that generates arity number of variables for each constraint
        for (j=0; j<k; j++) {
          temp[j] = (int) (((float)n)*rand()/(RAND_MAX+1.0));
     //check if this variable is already taken
          for(l=0; l<j; l++)  if (temp[l]==temp[j]) break;
          if (l<j) //broken: var. already taken
```

```
          j--; //retry for current position
        }

        //for (l=0; l<k; l++) printf("temp[%d] %d ",l, temp[l]);
        //printf("\n");

   //check if the generated set of variables has been generated before
   //sort variables in the constraint
   //this will ensure that no duplicate sets of variables exist
       qsort(temp, k, sizeof(int), intcomp);

        //for (l=0; l<k; l++) printf("temp[%d] %d ",l, temp[l]);
        //printf("\n");

        if (constraints[temp[0]][temp[1]] == 0) {
          constraints[temp[0]][temp[1]] = 1;
          good = 1;
        }
   //else start the loop again (generate a new constraint)

     } //EndWhile
  }

  //// PRINT THE CONSTRAINTS
  //printf("----------Constraints---------\n");
  //for (i=0; i<n; i++)
  //    for (j=0; j<n; j++)
                        //if (constraints[i][j] == 1)
                        //              printf("%s : %d-%d = %d\n","constraint", i, j, constraints[i][j]);

  free(temp);
}
void generate_tuples(int ****constraint_matrices, int total_tuples, int k, int d, int var_a, int var_b)
{
  char good;
  int j, i, *temp;

  zeroitb(constraint_matrices, d, d,var_a, var_b);

  temp = (int *)malloc(k*sizeof(int));

  // generate the allowed tuples for the constraints
  for (j=0; j<total_tuples; j++) {
     good = 0;

     // generate a tuple that has not been generated before
     // for this constraint
     while (!good) {
       for (i=0; i<k; i++)
          temp[i] = (int) (((float)d)*rand()/(RAND_MAX+1.0));

      if (constraint_matrices[var_a][var_b][temp[0]][temp[1]] == 0) {
          constraint_matrices[var_a][var_b][temp[0]][temp[1]] = 1;
          good = 1;
                        }
     }
  }

  //// PRINT THE CONSTRAINT MATRIX
  //printf("\n%s : %d-%d\n\n","------Tuples of Satisfied constraint", var_a, var_b);
  //if (constraint_matrices[var_a][var_b] == NULL) printf("NULL\n");
  //else {
  //   for (i=0; i<d; i++) {
  //      for (j=0; j<d; j++) printf("%d ",constraint_matrices[var_a][var_b][i][j]);
  //          printf("\n");
  //   }
  //}
  //printf("\n");

  free(temp);
}


void print_tuples(int ****constraint_matrices, int total_tuples, int k, int d, int var_a, int var_b)
{
  // PRINT THE CONSTRAINT MATRIX
```

```
  printf("\n%s : %d-%d\n\n","------Tuples of Satisfied constraint", var_a, var_b);
  if (constraint_matrices[var_a][var_b] == NULL) printf("NULL\n");
  else {
    for (int i=0; i<d; i++) {
      for (int j=0; j<d; j++) printf("%d ",constraint_matrices[var_a][var_b][i][j]);
        printf("\n");
    }
  }
  printf("\n");
}



void Initialize_constraints(int **constraints, int n){

        //-------------  Initialize **constaints ---------------------


        if(constraints == NULL){
                fprintf(stderr, "out of memory\n");
                system("PAUSE");
                //return 0;
        }

        for(int i = 0; i < n; i++){
                constraints[i] = (int *)malloc(n * sizeof(int));
                if(constraints[i] == NULL){
                        fprintf(stderr, "out of memory\n");
                        system("PAUSE");
                        //return 0;
                }
        }

        zeroita(constraints, n, n);
//-------------------------------------------------------------------------
}
void Initialize_constraints(int ****constraint_matrices, int d, int n){

//-------------  Initialize ****constraint_matrices -------------------------

        for(int m = 0; m < n; m++){
                constraint_matrices[m] = (int ***) malloc( n * sizeof(int **) );
                if(constraint_matrices[m] == NULL){
                        fprintf(stderr, "out of memory1\n");
                        system("PAUSE");
                        //return 0;
                }



        for(int i = 0; i < n; i++){
                constraint_matrices[m][i] = (int **) malloc( d * sizeof(int *) );
                if(constraint_matrices[m][i] == NULL){
                        fprintf(stderr, "out of memory2\n");
                        system("PAUSE");
                        //return 0;
                }

                for(int r = 0; r < d; r++){
                        constraint_matrices[m][i][r] = (int *) malloc( d * sizeof(int ) );
                        if(constraint_matrices[m][i][r] == NULL){
                                fprintf(stderr, "out of memory3\n");
                                system("PAUSE");
                                //return 0;
                        }

                        for(int p = 0; p <d; p++){
                                constraint_matrices[m][i][r][p] = (int ) malloc(1 * sizeof( int ) );
                                if(constraint_matrices[m][i][r][p] == NULL){
                                        fprintf(stderr, "out of memory4\n");
                                        system("PAUSE");
                                        //return 0;
                                }
                        }
                }
        }
```

```
 }

//---------------------------------------------------------------------------

}


int check_values(int *solv, int **constraints, int ****constraint_matrices, int n, int d)
{
	int count = 0;
	for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
		if (constraints[i][j] == 1){
			if(constraint_matrices[i][j][solv[i]][solv[j]]==0) count = count + 1;
			//printf("constraints[%d][%d]= %d \n",i,j, constraints[i][j]);
			//printf("constraint_matrices[%d][%d][%d][%d] = %d \n", i,j,solv[i],solv[j],
constraint_matrices[i][j][solv[i]][solv[j]]);
		}

	return count;
}


int selectRandVar(int *solv, int previousvar,int **constraints, int ****constraint_matrices, int n, int d){

	int *violate,  vvar = 0, randvar=-1, select;
	violate = (int *)malloc(d * d * d * d * sizeof(int ));

	for (int i=0; i<n; i++)
    for (int j=0; j<n; j++){
		if (constraints[i][j] == 1 && constraint_matrices[i][j][solv[i]][solv[j]]==0){
			violate[vvar] = i;
			vvar = vvar + 1;		// Αυξάνεται η πιθανότητα στις μεταβλητές όπου
παραουσιάζεται μεγαλύτερο πλήθος "παραβιάσεων".
			violate[vvar] = j;
			vvar = vvar + 1;
			//printf("constraints[%d][%d]= %d \n",i,j, constraints[i][j]);
			//printf("constraint_matrices[%d][%d][%d][%d] = %d \n", i,j,solv[i],solv[j],
constraint_matrices[i][j][solv[i]][solv[j]]);
		}
			//printf("constraints[%d][%d]= %d \n",i,j, constraints[i][j]);
			//printf("constraint_matrices[%d][%d][%d][%d] = %d \n", i,j,solv[i],solv[j],
constraint_matrices[i][j][solv[i]][solv[j]]);
}

		//for (int i=0; i<vvar; i++) printf("violate[%d]= %d \n",i, violate[i]);	// Εκτύπωση μεταβλητών που
παραβιάζονται



/*----- Ισοπίθανη επιλογή τυχαίας μεταβλητής----
		int *temp;
		int	count = 0;
		temp = (int *)malloc(d * d * sizeof(int ));
		for (int i=0; i<vvar; i++){
			for (int j=i+1; j<vvar; j++)	if(violate[i] == violate[j] && violate[i]!=-1) violate[j]=-1;
			if(violate[i]!=-1){
				temp[count] = violate[i];
				count = count + 1;
			}
		}
	for (int i=0; i<count; i++) printf("viol-temp[%d]= %d \n",i, temp[i]);
	for (int i=0; i<vvar; i++) printf("violate[%d]= %d \n",i, violate[i]);
	free(violate);

	select = (int) (((float)count)*rand()/(RAND_MAX+1.0));

	while(previousvar == temp[select]) select = (int) (((float)count)*rand()/(RAND_MAX+1.0));

	randvar = temp[select];

	free(temp);

	return randvar;
-------------------------------------------------*/
```

```
	select = (int) (((float)vvar)*rand()/(RAND_MAX+1.0));

	while(previousvar == violate[select]) select = (int) (((float)vvar)*rand()/(RAND_MAX+1.0));
	randvar = violate[select];
	//printf("vvar = %d \n", vvar);
	//printf("randvar = %d \n", randvar);
	//printf("select = %d \n", select);

	free(violate);

	return randvar;


}


void bestSolution(int *prevSolv, int * solv, bool firstTimeMinConf, int conflicts, int n, int countMinConf, int countRandVar, int *previous){

	// prevcountMinConf=previous[0], prevconflicts=previous[1], prevcountRandVar=previous[2]

	if(firstTimeMinConf == true){
		for(int i = 0; i < n; i++) prevSolv[i] = solv[i];

		previous[1] = conflicts;
		previous[0] = countMinConf;
		previous[2] = countRandVar;
	}

	else{

		if(conflicts <= previous[1]) {
			for(int i = 0; i < n; i++)  prevSolv[i] = solv[i];
			previous[1] = conflicts;
			previous[0] = countMinConf;
			previous[2] = countRandVar;
		}

	}

}


int CountDistance(int *initSolv, int *solv, int n){

	int countdist = 0;
	for(int i = 0; i < n; i++)  if(initSolv[i] == solv[i]) countdist = countdist + 1;

	countdist = n - countdist;

	return countdist;
}


int Min_Conflicts(int *solv, int *initSolv, int *prevSolv, int **constraints, int ****constraint_matrices, int d, int n, int repeat_times, int repeat_Min_Conf,bool firstTime, bool firstTimeMinConf, int conflicts, int countMinConf, int countRandVar, int *previous, int *SDistances, int *SConflicts, int problem)
{
	int previousvar, tempconflicts=0, rounds, rvar, *temp;
	int distance; // απόσταση της "τρέχουσας" Λύσης από την προηγούμενη Λύση

	countMinConf=0;
	countRandVar=0;

	temp = (int *)malloc(n * sizeof(int ));

	previousvar=-1; // Για να αποφευχθεί η επιλογή της ίδιας μεταβλητής με πρίν
	conflicts=0;


	// χρήση προηγούμενης λύσης προς αρχικοποίηση του Min_Conflicts.....Ανάθεση τιμών -> Προηγούμενη Λύση
	if(firstTime == false){
		for(int i = 0; i < n; i++) initSolv[i] = solv[i];
```

```
                //PRINT INIT VALUES
                //printf("\n-----INIT VALUES----firstTime== false\n");
                //for (int l=0; l<n; l++) printf("initSolv[%d]= %d \n",l, initSolv[l]);
        }


// Repeat "N" times (user selection) of Min_Conflicts .....Επανάληψη Min_Conflicts (Όχι επανακίνηση)
        while(repeat_Min_Conf!=0){

                countMinConf++;
                //printf("\nTimes of MinConf= %d\n", countMinConf);

                rounds = repeat_times;

                // Αρχική Τυχαία ανάθεση τιμών στις μεταβλητες
                if(firstTime == true){
                        for(int i = 0; i < n; i++) solv[i] = -1;
                        for(int i = 0; i < n; i++) solv[i] = (int) (((float)d)*rand()/(RAND_MAX+1.0));

                        for(int i = 0; i < n; i++) initSolv[i] = solv[i]; // αρχική ανάθεση τιμών

                        //PRINT INIT VALUES
                        //printf("\n-----INIT VALUES----firstTime== true\n");
                        //for (int l=0; l<n; l++) printf("initSolv[%d]= %d \n",l, initSolv[l]);
                }


                //Έλεγχος αν παραβιάζονται οι περιορισμοί
                conflicts = check_values(solv, constraints, constraint_matrices, n, d);



                if(conflicts==0){
                        free(temp);
                        //printf("\n-----SOLUTION = INIT VALUES = Best Solution ----\n");
                        //printf("\nTimes of MinConf= %d\n", countMinConf);
                        //printf("\nTimes of RandVar= 0 \n");
                        distance = CountDistance(initSolv, solv, n);
                        //printf("\nDistance of Solution= %d\n", distance);
                        SDistances[problem] = distance;
                        SConflicts[problem] = conflicts;
                        return 0;
                }

                while(rounds!=0){

                        countRandVar = countRandVar + 1;

                        for(int i = 0; i < n; i++) temp[i] = solv[i];

                        //Επιλογή τυχαίας μεταβλητής (σε περιορισμό που παραβιάζεται)
                        rvar = selectRandVar(solv, previousvar, constraints, constraint_matrices, n, d);
                        //printf("rvar= %d \n", rvar);

                        previousvar = rvar;       //προηγούμενη τυχαία μεταβλητή που ελέγχθηκε

                        //Εξέτασε για την τυχαία μεταβλητη όλες τις δυνατές τιμές
                        for(int i = 0; i < d; i++){
                                temp[rvar] = i;
                                tempconflicts = check_values(temp, constraints, constraint_matrices, n, d);

                                //printf("rvar_conflicts= %d \n", tempconflicts);


                                if(tempconflicts < conflicts || tempconflicts == conflicts){                // Επιλογή
Λύσης με Λιγότερες ΠΑΡΑΒΙΑΣΕΙΣ
                                        conflicts = tempconflicts;
                                        solv[rvar] = i;
                                        //printf("rvar_value= %d \n", i);
                                 }
                        }


                        //Αν δεν παραβιάζονται οι περιορισμοί (conflicts=0) βρέθηκε ΛΥΣΗ!
                        if(conflicts==0){
```

```
                                        //printf("\n-----SOLUTION : conflicts=0 ----\n");
                                        //for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);

                                        free(temp);
                                        //printf("\nTimes of MinConf= %d\n", countMinConf);
                                        //printf("\nTimes of RandVar= %d\n", countRandVar);

                                        distance = CountDistance(initSolv, solv, n);
                                        //printf("\nDistance of Solution= %d\n", distance);

                                        SDistances[problem] = distance;
                                        SConflicts[problem] = conflicts;
                                        return 0;
                                }

                                //printf("\n-----SOLUTION----\n");
                                        //for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);
                                        //printf("\nrvar= %d\n", rvar);

                                rounds = rounds - 1;
                                //printf("\nconflicts= %d\n", conflicts);
                        }

                        //Βέλτιστη Λύση μεταξύ των επαναλήψεων του "Min_Conflicts"
                        bestSolution(prevSolv, solv, firstTimeMinConf, conflicts, n, countMinConf, countRandVar,
previous);

                        // prevcountMinConf=previous[0], prevconflicts=previous[1], prevcountRandVar=previous[2]

                        //printf("\nShow the prevconflicts= %d\n", previous[1]);
                        //printf("\nShow the conflicts= %d\n", conflicts);

                        repeat_Min_Conf = repeat_Min_Conf - 1;

                        firstTimeMinConf = false;
                        firstTime = false;
                        //system("PAUSE");


                }
////////////Repeat Min_Conf


                if(conflicts > previous[1]) {
                        for(int i = 0; i < n; i++)  solv[i] = prevSolv[i] ;   // BEST solution

                        conflicts = previous[1];
                        countMinConf = previous[0];
                        countRandVar = previous[2];
                }



                //printf("\n-----recent Better SOLUTION----\n");
                //for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);

                //printf("\nTimes of MinConf= %d\n", previous[0]);
                //printf("\nTimes of RandVar= %d\n", previous[2]);

                distance = CountDistance(initSolv, solv, n);
                //printf("\nDistance of Solution= %d\n", distance);
                //printf("\n----------------------------\n");

                SDistances[problem] = distance;
                SConflicts[problem] = conflicts;
                //system("PAUSE");

                for(int i = 0; i < n; i++)  initSolv[i] = solv[i]; // Χρήση της λύσης σε επόμενο πρόβλημα

                free(temp);
                return conflicts;
}
```

```
void gen_Constraints(int **constraints, int n, int k, int total_constraints, int addConstrains, bool firstTime){

	if(firstTime == true){
		generate_constraints(constraints, n, k, total_constraints);
	}

	if(firstTime == false){
		generate_constraints(constraints, n, k, addConstrains);
	}

}


void gen_Tuples(int **constraints, int **prevconstraints, bool firstTime, int n, int ****constraint_matrices, int total_tuples, int k, int d, int var_a, int var_b){

	if(firstTime == true){
	  for (int i=0; i<n; i++)
		for (int j=0; j<n; j++)
			if (constraints[i][j] == 1){
				var_a = i;
				var_b = j;
				// Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη μεταβλητών όπου υπάρχει περιορισμός
				generate_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);
				prevconstraints[i][j] = 1;
			}
	}

	if(firstTime == false){
	  for (int i=0; i<n; i++)
		for (int j=0; j<n; j++)
			if (constraints[i][j] == 1 && prevconstraints[i][j] != 1){
				var_a = i;
				var_b = j;
				// Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη μεταβλητών όπου υπάρχει περιορισμός και δεν έχει δημιουργηθεί ξανά
				generate_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);
				prevconstraints[i][j] = 1;
			}
	}


	  for (int i=0; i<n; i++)
		for (int j=0; j<n; j++)
			if (constraints[i][j] == 1){
				var_a = i;
				var_b = j;
				// print_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);	// Εκτύπωση Πίνακα "ικανοποίησης περιορισμών"
			}


}


int _tmain(int argc, _TCHAR* argv[])
{
	int ****constraint_matrices,**constraints, **prevconstraints, total_tuples, total_constraints, k, d, n, var_a=-1, var_b=-1, problems, addConstrains;
	int *solve, *prevSolv, repeat_times, solution=-1, repeat_Min_Conf;
	int conflicts=0, countMinConf=0, countRandVar=0;

	int *previous, *initSolv;
	int *SDistances, *SConflicts;

	char seed;

	float tpers;
	bool firstTime = true, firstTimeMinConf;


  cout << endl<< "		Min-Conflicts		" << endl;
  cout <<	"_____" << endl<< endl;
```

```
cout << "Enter the number of Variables: ";
cin >> n;
cout << endl;

float maxn = n*n/2 - n;
printf("\n(Max constrains= %.0f) ", maxn);
cout << "Enter the number of first CSP constrains: ";
cin >> total_constraints;
cout << endl;

cout << "Enter the number of values: ";
cin >> d;
cout << endl;

cout << "Enter the percentage (%) of satisfy constrains: ";
cin >> tpers;
cout << endl;

total_tuples = ceil(((float)d*d)*tpers/100);
k=2; // temp[0] and temp[1]

cout << "How many times select (violated) random variable to evaluate: ";
cin >> repeat_times;
cout << endl;

cout << "How many times run Min_Conflicts: ";
cin >> repeat_Min_Conf;
cout << endl;

cout << "How many Problems to Solve: ";
cin >> problems;
cout << endl;

cout << "How many Constrains be added in every new Problem: ";
cin >> addConstrains;
cout << endl;

cout << "New SEED for Srand(y/n): ";
cin >> seed;
cout << endl;


//_____________________________ Random SEED to Srand() for each problem _____________________________
        FILE *fpw, *fpr;
        int pot = 0;       // point of text
        //int *RandSeed;      // random numbers unique for each problem
        int RandSeed;
        int num, out;

        //RandSeed = (int *)malloc(problems * sizeof(int ));

        char outputFilename[] = "../../seed.txt"; // path: ..\Visual Studio 2010\Projects

        if(seed == 'y'){
                fpw = fopen(outputFilename, "w");

                if (fpw == NULL) {
                        fprintf(stderr, "Can't open output file %s!\n", outputFilename);
                        exit(1);
                        }

                srand( (unsigned int)time(NULL) );    // initialize random seed

                //for (int i=0; i<problems; i++){
                        out = (int) (((float)(unsigned int)time(NULL))*rand()/(RAND_MAX+1.0));
                        fprintf(fpw, "%d\n", out);
                        //}

                fclose(fpw);
        }



        fpr = fopen(outputFilename, "r");
```

```
	if (fpr == NULL){
		fprintf(stderr, "Can't open output file %s!\n", outputFilename);
		exit(1);
	}

	//int probl = problems;
  //while( fscanf(fpr, "%d", &num) != EOF ) RandSeed[pot++] = num;
	while( fscanf(fpr, "%d", &num) != EOF ) RandSeed = num;
	//for (int i=0; i<problems; i++)
		printf("RandSeed= %d \n", RandSeed);

	fclose(fpr);

//_____________________________________________________________________


	solve = (int *)malloc(n * sizeof(int ));
	prevSolv = (int *)malloc(n * sizeof(int ));	// Προηγούμενη Λύση
	initSolv = (int *)malloc(n * sizeof(int ));	// προηγούμενη λύση προς αρχικοποίηση Min_Conflicts
	constraints = (int **)malloc(n * sizeof(int *));
	prevconstraints= (int **)malloc(n * sizeof(int *));
	constraint_matrices = (int ****) malloc( n * sizeof(int ***) );


	previous = (int *)malloc(3 * sizeof(int ));
	// prevcountMinConf=previous[0], prevconflicts=previous[1], prevcountRandVar=previous[2]


	SDistances  = (int *)malloc(problems * sizeof(int ));
	// SDistances : πίνακας για τον υπολογισμό του ΜΟ των αποστάσεων


	SConflicts  = (int *)malloc(problems * sizeof(int ));
	//SConflicts : πίνακας για τον υπολογισμό του ΜΟ των παραβιάσεων


   Initialize_constraints(constraints, n);
	Initialize_constraints(prevconstraints, n);
	Initialize_constraints(constraint_matrices, d,n);

	int problem=0;

  while (problem != problems){

	  srand( RandSeed );    // initialize random seed

	  // Δημιουργία Πίνακα ΠΕΡΙΟΡΙΣΜΩΝ
	  gen_Constraints(constraints, n, k, total_constraints, addConstrains, firstTime);

	  // Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη μεταβλητών όπου υπάρχει περιορισμός
	  gen_Tuples(constraints, prevconstraints, firstTime, n, constraint_matrices, total_tuples, k, d, var_a, var_b);

	  firstTimeMinConf = true;


		solution = Min_Conflicts(solve, initSolv, prevSolv, constraints, constraint_matrices, d,n,
repeat_times, repeat_Min_Conf, firstTime, firstTimeMinConf, conflicts, countMinConf, countRandVar,  previous,
SDistances, SConflicts, problem);


		//if(solution==0)	printf("\nGOAL! -> Conflicts= %d \n", solution);

		//else{
		//	printf("\nConflicts= %d \n\n", solution);
		//}

		problem = problem + 1;

		firstTime = false;

		//printf("\ntotal_constraints = %d \n", total_constraints);
		total_constraints = total_constraints + addConstrains;


		if(total_constraints > (n*n-n)/2){
```

```
                                printf("\n A lot of Constrains!!!!\n");
                                printf("\nConstrains = %d \n", total_constraints);
                                break;
                        }
                        //system("PAUSE");
                }



                //Υπολογισμός Μ.Ο. αποστάσεων και Μ.Ο. παραβιάσεων
        int MODistances=0, MOConflicts=0;

        int total_solv_problems = 0;      // Total solved Problems
        int cri_point_conf = 0;                         // κρίσιμο σημείο conflicts
        int cri_point_dist = 0;                         // κρίσιμο σημείο distance
        float before_cri_avg_dist = 0;  // Πριν το κρίσιμο σημείο avg distance
        bool cri_point = false;
        int cri_sum_dist=0;


        printf("\n\n");

        for (int p=0; p < problems; p++){
                        MOConflicts = MOConflicts + SConflicts[p];
                        printf("SConflicts[%d]= %d \n",p, SConflicts[p]);
                        if(SConflicts[p]==0 && cri_point==false) total_solv_problems++;
                        if(SConflicts[p]>0 && cri_point==false) { cri_point_conf=SConflicts[p];  cri_point=true; }
                }


        printf("\n\n");

        for (int p=1; p < problems; p++){
                        MODistances = MODistances + SDistances[p];
                        printf("SDistances[%d]= %d \n",p, SDistances[p]);
                        if(SConflicts[p]>0 && cri_point==true) { cri_point_dist=SDistances[p];  cri_point=false; }
                }


        for (int t=1; t < total_solv_problems; t++){
                        cri_sum_dist = cri_sum_dist + SDistances[t];
                        before_cri_avg_dist = (float)cri_sum_dist/(total_solv_problems-1);
                }

         printf("\n----- Problems = %d ------\n", problems);

                printf("Total solved Problems = %d \n", total_solv_problems);
                printf("Before critical point avg distance = %.3f \n", before_cri_avg_dist);

                printf("critical point conflicts = %d \n", cri_point_conf);
                printf("critical point distance = %d \n", cri_point_dist);

                printf("CSPs.Avg.Conflicts = %.3f \n", (float)MOConflicts/(problems-1));
                printf("CSPs.Avg.Distances = %.3f \n", (float)MODistances/(problems-1));


                system("PAUSE");

                return 0;
}
```

// Μέθοδος τοπικής αναζήτησης δύο φάσεων επίλυσης DCSP

```
#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <stdlib.h>
```

```
#include <math.h>

using namespace std;

int intcomp(void const *ip, void const *jp)
{
                        int i = *((int const *)(ip));
                        int j = *((int const *)(jp));
                        return i-j;
}


void zeroita(int **array, int nrows, int ncolumns){
        int i, j;
        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncolumns; j++)      array[i][j] = 0;
        }
}
void zeroitb(int ****array, int nrows, int ncolumns, int var_a, int var_b){
        int i, j=0;
        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncolumns; j++){
                        //printf("var_a=%d  var_b=%d  i=%d  j=%d \n", var_a,var_b,i,j);
                        array[var_a][var_b][i][j] = 0;
                        //printf("array[%d][%d][%d][%d] = %d \n", var_a,var_b,i,j,array[var_a][var_b][i][j]);
                }
        }
}


void generate_constraints(int **constraints, int n, int k, int total_constraints)
{
  int i, j, l, *temp;
  char good;

  temp = (int *)malloc(k*sizeof(int));

  //loop that generates constraints
  for (i=0; i<total_constraints; i++) {

     good=0;

     //loop that generates a set of variables until a set that has not been generated before
     while (!good) {

        for (j=0; j<k; j++) temp[j]=-1;

     //loop that generates arity number of variables for each constraint
        for (j=0; j<k; j++) {
          temp[j] = (int) (((float)n)*rand()/(RAND_MAX+1.0));
     //check if this variable is already taken
          for(l=0; l<j; l++)  if (temp[l]==temp[j]) break;
          if (l<j) //broken: var. already taken
          j--; //retry for current position
        }

        //for (l=0; l<k; l++) printf("temp[%d] %d ",l, temp[l]);
        //printf("\n");

    //check if the generated set of variables has been generated before
    //sort variables in the constraint
    //this will ensure that no duplicate sets of variables exist
        qsort(temp, k, sizeof(int), intcomp);

        //for (l=0; l<k; l++) printf("temp[%d] %d ",l, temp[l]);
        //printf("\n");

        if (constraints[temp[0]][temp[1]] == 0) {
          constraints[temp[0]][temp[1]] = 1;
          good = 1;
        }
    //else start the loop again (generate a new constraint)

     } //EndWhile
  }
```

```
 // PRINT THE CONSTRAINTS
/* printf("-----------Constraints---------\n");
 for (i=0; i<n; i++)
    for (j=0; j<n; j++)
                    if (constraints[i][j] == 1)
                            printf("%s : %d-%d = %d\n","constraint", i, j, constraints[i][j]);*/

 free(temp);
}
void generate_tuples(int ****constraint_matrices, int total_tuples, int k, int d, int var_a, int var_b)
{
 char good;
 int j, i, *temp;

 zeroitb(constraint_matrices, d, d,var_a, var_b);

 temp = (int *)malloc(k*sizeof(int));

 // generate the allowed tuples for the constraints
 for (j=0; j<total_tuples; j++) {
    good = 0;

    // generate a tuple that has not been generated before
    // for this constraint
    while (!good) {
      for (i=0; i<k; i++)
        temp[i] = (int) (((float)d)*rand()/(RAND_MAX+1.0));

     if (constraint_matrices[var_a][var_b][temp[0]][temp[1]] == 0) {
        constraint_matrices[var_a][var_b][temp[0]][temp[1]] = 1;
        good = 1;
                    }
    }
 }

 //// PRINT THE CONSTRAINT MATRIX
 //printf("\n%s : %d-%d\n\n","------Tuples of Satisfied constraint", var_a, var_b);
 //if (constraint_matrices[var_a][var_b] == NULL) printf("NULL\n");
 //else {
 //   for (i=0; i<d; i++) {
 //      for (j=0; j<d; j++) printf("%d ",constraint_matrices[var_a][var_b][i][j]);
 //         printf("\n");
 //   }
 //}
 //printf("\n");

 free(temp);
}

void print_tuples(int ****constraint_matrices, int total_tuples, int k, int d, int var_a, int var_b)
{
 // PRINT THE CONSTRAINT MATRIX
 printf("\n%s : %d-%d\n\n","------Tuples of Satisfied constraint", var_a, var_b);
 if (constraint_matrices[var_a][var_b] == NULL) printf("NULL\n");
 else {
    for (int i=0; i<d; i++) {
      for (int j=0; j<d; j++) printf("%d ",constraint_matrices[var_a][var_b][i][j]);
        printf("\n");
    }
 }
 printf("\n");
}

void Initialize_constraints(int **constraints, int n){

        //-------------  Initialize **constaints ----------------------


        if(constraints == NULL){
                fprintf(stderr, "out of memory\n");
                system("PAUSE");
                //return 0;
        }

        for(int i = 0; i < n; i++){
                constraints[i] = (int *)malloc(n * sizeof(int));
```

```
                        if(constraints[i] == NULL){
                                fprintf(stderr, "out of memory\n");
                                system("PAUSE");
                                //return 0;
                        }
                }

                zeroita(constraints, n, n);
//---------------------------------------------------------------------
}
void Initialize_constraints(int ****constraint_matrices, int d, int n){

//------------  Initialize ****constraint_matrices -------------------------

                for(int m = 0; m < n; m++){
                        constraint_matrices[m] = (int ***) malloc( n * sizeof(int **) );
                        if(constraint_matrices[m] == NULL){
                                fprintf(stderr, "out of memory1\n");
                                system("PAUSE");
                                //return 0;
                        }



                for(int i = 0; i < n; i++){
                        constraint_matrices[m][i] = (int **) malloc( d * sizeof(int *) );
                        if(constraint_matrices[m][i] == NULL){
                                fprintf(stderr, "out of memory2\n");
                                system("PAUSE");
                                //return 0;
                        }

                        for(int r = 0; r < d; r++){
                                constraint_matrices[m][i][r] = (int *) malloc( d * sizeof(int ) );
                                if(constraint_matrices[m][i][r] == NULL){
                                        fprintf(stderr, "out of memory3\n");
                                        system("PAUSE");
                                        //return 0;
                                }

                                for(int p = 0; p <d; p++){
                                        constraint_matrices[m][i][r][p] = (int ) malloc(1 * sizeof( int ) );
                                        if(constraint_matrices[m][i][r][p] == NULL){
                                                fprintf(stderr, "out of memory4\n");
                                                system("PAUSE");
                                                //return 0;
                                        }
                                }
                        }
                }
 }

//---------------------------------------------------------------------

}


int check_values(int *solv, int **constraints, int ****constraint_matrices, int n, int d)
{
                int count = 0;
                for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)
                        if (constraints[i][j] == 1){
                                if(constraint_matrices[i][j][solv[i]][solv[j]]==0) count = count + 1;
                                //printf("constraints[%d][%d]= %d \n",i,j, constraints[i][j]);
                                //printf("constraint_matrices[%d][%d][%d][%d]       =       %d       \n",       i,j,solv[i],solv[j],
constraint_matrices[i][j][solv[i]][solv[j]]);
                        }

                return count;
}

int CountDistance(int *initSolv, int *solv, int n){

                int countdist = 0;
                for(int i = 0; i < n; i++)  if(initSolv[i] == solv[i]) countdist = countdist + 1;
```

```
	countdist = n - countdist;

	return countdist;
}


int selectRandVar(int *solv, int *initSolv, int previousvar,int **constraints, int ****constraint_matrices, int n, int d){

	int *violate,  vvar = 0, randvar=-1, select;
	violate = (int *)malloc(d * d * d * d * sizeof(int ));

	for (int i=0; i<n; i++){
		for (int j=0; j<n; j++){
			if (constraints[i][j] == 1 && constraint_matrices[i][j][solv[i]][solv[j]]==0)  //όπου υπάρχει περιορισμός και παραβίαση
			{
				violate[vvar] = i;
				vvar = vvar + 1;		// Αυξάνεται η πιθανότητα στις μεταβλητές όπου παραουσιάζεται μεγαλύτερο πλήθος "παραβιάσεων".
				violate[vvar] = j;
				vvar = vvar + 1;
				//printf("constraints[%d][%d]= %d \n",i,j, constraints[i][j]);
				//printf("constraint_matrices[%d][%d][%d][%d] = %d \n", i,j,solv[i],solv[j], constraint_matrices[i][j][solv[i]][solv[j]]);
			}

		//Επιλογή τυχαίας μεταβλητής (σε περιορισμό που παραβιάζεται και απόσταση=0 από αρχική ΛΥΣΗ ??Συντελεστής Βαρύτητας?????)
			//if (initSolv[j]!=solv[j]) //όπου υπάρχει διαφορετική τιμή από την αρχική λύση
			//{
			//	violate[vvar] = j;
			//	vvar = vvar + 1;
			//	//printf("initSolv[%d]=%d != solv[%d]= %d \n",j, initSolv[j], j, solv[j]);
			//}

		}

			//if (initSolv[i]!=solv[i]) //όπου υπάρχει διαφορετική τιμή από την αρχική λύση
			//{
			//	violate[vvar] = i;
			//	vvar = vvar + 1;
			//	//printf("initSolv[%d]=%d != solv[%d]= %d \n",i, initSolv[i], i, solv[i]);
			//}

	}


		//for (int i=0; i<vvar; i++) printf("violate[%d]= %d \n",i, violate[i]); // Εκτύπωση μεταβλητών που παραβιάζονται



/*----- Ισοπίθανη επιλογή τυχαίας μεταβλητής----
		int *temp;
		int	count = 0;
		temp = (int *)malloc(d * d * sizeof(int ));
		for (int i=0; i<vvar; i++){
			for (int j=i+1; j<vvar; j++)	if(violate[i] == violate[j] && violate[i]!=-1) violate[j]=-1;
			if(violate[i]!=-1){
				temp[count] = violate[i];
				count = count + 1;
			}
		}
	for (int i=0; i<count; i++) printf("viol-temp[%d]= %d \n",i, temp[i]);
	for (int i=0; i<vvar; i++) printf("violate[%d]= %d \n",i, violate[i]);
	free(violate);

	select = (int) (((float)count)*rand()/(RAND_MAX+1.0));

	while(previousvar == temp[select]) select = (int) (((float)count)*rand()/(RAND_MAX+1.0));

	randvar = temp[select];

	free(temp);
```

```
	return randvar;
-----------------------------------------------*/


	select = (int) (((float)vvar)*rand()/(RAND_MAX+1.0));

	while(previousvar == violate[select]) select = (int) (((float)vvar)*rand()/(RAND_MAX+1.0));
	randvar = violate[select];
	//printf("vvar = %d \n", vvar);
	//printf("randvar = %d \n", randvar);
	//printf("select = %d \n", select);

	free(violate);

	return randvar;

}


float CalcZ(int conflicts,int total_constraints, int distance, int n, float w1, float w2)
{
	float normConflicts, normDistance;
	float Z=-1;
	normConflicts= (float)conflicts/total_constraints;
	normDistance= (float)distance/n;

	Z = w1*normConflicts + w2*normDistance;

	return Z;
}


void bestSolution(int *prevSolv, int * solv, bool firstTimeMinConf, float Z, float *prevZ, int conflicts, int total_constraints, int distance, int n, float W1, float W2, int countMinConf, int countRandVar, int *previous){

	// prevconflicts=previous[0], prevdistance=previous[1], prevcountMinConf=previous[2], prevcountRandVar=previous[3]

	if(firstTimeMinConf == true){
		for(int i = 0; i < n; i++) prevSolv[i] = solv[i];

		previous[0] = conflicts;
		previous[1] = distance;
		previous[2] = countMinConf;
		previous[3] = countRandVar;
		prevZ[0] = CalcZ( conflicts, total_constraints, distance, n, W1, W2);
		//printf("\nprevZ[0]= %.3f\n", prevZ[0]);
		//printf("\nZ[0]= %.3f\n", Z);
	}

	else{

		if(Z < prevZ[0]) {
			for(int i = 0; i < n; i++)  prevSolv[i] = solv[i];
			previous[0] = conflicts;
			previous[1] = distance;
			previous[2] = countMinConf;
			previous[3] = countRandVar;
			prevZ[0] = Z;
		}
	}
}


float TwoPhase(int *solv, int *initSolv, int *prevSolv, int **constraints, int ****constraint_matrices, int d, int n, int repeat_times, int repeat_Min_Conf,bool firstTime, bool firstTimeMinConf, float Z, int conflicts, int total_constraints, int countMinConf, int countRandVar, int *previous, int *SDistances, int *SConflicts, int problem,float W1, float W2)
{
	int previousvar, tempconflicts=0, rounds, rvar, *temp;
	int distance=0; // απόσταση της "τρέχουσας" Λύσης από την προηγούμενη Λύση - θεωρώ αρχική απόσταση = 0

	float normConflicts=0, normDistance=0;
	// Z=min(w1*normConflicts + w2*normDistance) : Στόχος αλγόριθμου δύο φάσεων η ελαχιστοποίηση της τιμής του Z->0 βέλτιστη λύση
	// W1:συντελεστής βαρύτητας παραβιάσεων
```

```
	// W2:συντελεστής βαρύτητας απόστασης
	// normConflicts: κανονικοποιημένη [0,1] τιμή παραβιάσεων (πλήθος παραβιάσεων /πλήθος περιορισμών)
	// normDistance: κανονικοποιημένη [0,1] τιμή απόστασης από προηγούμενη λύση (απόσταση από προηγούμενη λύση/πλήθος μεταβλητών)

	float *prevZ;
	prevZ = (float *)malloc(1 * sizeof(int ));


	countMinConf=0;
	countRandVar=0;

	temp = (int *)malloc(n * sizeof(int ));

	previousvar=-1; // Για να αποφευχθεί η επιλογή της ίδιας μεταβλητής με πρίν
	conflicts=0;


	// χρήση προηγούμενης λύσης προς αρχικοποίηση του Min_Conflicts.....Ανάθεση τιμών -> Προηγούμενη Λύση
	if(firstTime == false){

		for(int i = 0; i < n; i++) initSolv[i] = solv[i] ;

		//PRINT INIT VALUES
		//printf("\n-----INIT VALUES----firstTime== false\n");
		//for (int l=0; l<n; l++) printf("initSolv[%d]= %d \n",l, initSolv[l]);
	}


  // Repeat "N" times (user selection) of Min_Conflicts
	while(repeat_Min_Conf!=0){


		//printf("\nTimes of MinConf= %d\n", countMinConf);
		countMinConf++;

		rounds = repeat_times;

		// Αρχική Τυχαία ανάθεση τιμών στις μεταβλητες
		if(firstTime == true){
			for(int i = 0; i < n; i++) solv[i] = -1;
			for(int i = 0; i < n; i++) solv[i] = (int) (((float)d)*rand()/(RAND_MAX+1.0));

			for(int i = 0; i < n; i++) initSolv[i] = solv[i]; // αρχική ανάθεση τιμών

			//PRINT INIT VALUES
			//printf("\n-----INIT VALUES----firstTime== true\n");
			//for (int l=0; l<n; l++) printf("initSolv[%d]= %d \n",l, initSolv[l]);
		}


		//Έλεγχος αν παραβιάζονται οι περιορισμοί
		conflicts = check_values(solv, constraints, constraint_matrices, n, d);

		//printf("\nConflicts= %d\n", conflicts);
		//printf("\nDistance= %d\n", distance);


		Z = CalcZ( conflicts, total_constraints, distance, n, W1, W2);
		//printf("\nZ= %.3f\n", Z);

		if(Z==0){
			free(temp);
			/*printf("\n-----SOLUTION = INIT VALUES = Best Solution ----\n");
			printf("\nTimes of MinConf= %d\n", countMinConf);
			printf("\nTimes of RandVar= %d\n", countRandVar);
			printf("\nConflicts= %d\n", conflicts);
			printf("\nDistance= %d\n", distance);
			printf("\nZ= %.3f\n", Z);*/
			SDistances[problem] = distance;
			SConflicts[problem] = conflicts;
			return 0;
		}
```

```
		while(rounds!=0){

			countRandVar++;

			for(int i = 0; i < n; i++) temp[i] = solv[i];

			if(conflicts!=0){
				rvar = selectRandVar(solv, initSolv, previousvar, constraints, constraint_matrices, n, d);
				//printf("\nrvar= %d\n", rvar);

				previousvar = rvar;	//προηγούμενη τυχαία μεταβλητή που ελέγχθηκε

				//Εξέτασε για την τυχαία μεταβλητη όλες τις δυνατές τιμές
				for(int i = 0; i < d; i++){
					temp[rvar] = i;
					tempconflicts = check_values(temp, constraints, constraint_matrices, n, d);
					if(tempconflicts < conflicts || tempconflicts == conflicts){
						// Επιλογή Λύσης με Λιγότερες ΠΑΡΑΒΊΑΣΕΙΣ
						conflicts = tempconflicts;
						solv[rvar] = i;
					 }
				}
			}


			//Έλεγχος απόστασης προηγούμενης λύσης
			distance = CountDistance(initSolv, solv, n);


			Z = CalcZ( conflicts, total_constraints, distance, n, W1, W2);

			//printf("\nZ= %.3f\n", Z);


			//Αν conflicts=0 και distance=0 βρέθηκε ΛΥΣΗ!
			if(Z==0){
				//printf("\n-----SOLUTION : conflicts=0 and distance=0 ----\n");
				//for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);

				free(temp);
				//printf("\nTimes of MinConf= %d\n", countMinConf);
				//printf("\nTimes of RandVar= %d\n", countRandVar);

				SDistances[problem] = distance;
				SConflicts[problem] = conflicts;
				return 0;
			}

			//printf("\n-----SOLUTION----\n");
				//for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);
				//printf("\nrvar= %d\n", rvar);

			rounds = rounds - 1;

		}

		//Βέλτιστη Λύση μεταξύ των επαναλήψεων του "Min_Conflicts"
		bestSolution(prevSolv, solv, firstTimeMinConf, Z, prevZ, conflicts, total_constraints, distance, n, W1, W2, countMinConf, countRandVar, previous);

		//			prevconflicts=previous[0], prevdistance=previous[1], prevcountMinConf=previous[2],prevcountRandVar=previous[3]

		//printf("\nShow the prevconflicts= %d\n", previous[0]);
		//printf("\nShow the prevdistance= %d\n", previous[1]);
		//printf("\nShow the conflicts= %d\n", conflicts);
		//printf("\nShow the distance= %d\n", distance);
		//printf("\nShow the countMinConf= %d\n", countMinConf);

		repeat_Min_Conf = repeat_Min_Conf - 1;

		firstTimeMinConf = false;
		firstTime = false;
```

```
                //printf("\nZ= %.3f\n", Z);
                //printf("\nprevZ[0]= %.3f\n", prevZ[0]);

                //system("PAUSE");

        }
////////////Repeat Min_Conf
        //printf("\nZ= %.3f\n", Z);
        //printf("\nprevZ[0]= %.3f\n", prevZ[0]);

        if(Z >  prevZ[0]) {
                for(int i = 0; i < n; i++)  solv[i] = prevSolv[i] ;   // BEST solution


                conflicts = previous[0];
                distance = previous[1];
                countMinConf = previous[2];
                countRandVar = previous[3];
                Z = prevZ[0];
        }



        //printf("\n----- recent Better SOLUTION----\n");
        //for (int l=0; l<n; l++) printf("solv[%d]= %d \n",l, solv[l]);

        //printf("\nTimes of MinConf= %d\n", countMinConf);
        //printf("\nTimes of RandVar= %d\n", countRandVar);
        //printf("\nConflicts= %d\n", conflicts);
        //printf("\nDistance= %d\n", distance);
        //printf("\nZ= %.3f\n", Z);
        //printf("\n----------------------------\n");

        SDistances[problem] = distance;
        SConflicts[problem] = conflicts;

        for(int i = 0; i < n; i++)  initSolv[i] = solv[i]; // Χρήση της λύσης σε επόμενο πρόβλημα

        free(temp);
        free(prevZ);
        return Z;
}



void gen_Constraints(int **constraints, int n, int k, int total_constraints, int addConstrains, bool firstTime){

        if(firstTime == true){
                generate_constraints(constraints, n, k, total_constraints);
        }

        if(firstTime == false){
                generate_constraints(constraints, n, k, addConstrains);
        }
}



void gen_Tuples(int **constraints, int **prevconstraints, bool firstTime, int n, int ****constraint_matrices, int
total_tuples, int k, int d, int var_a, int var_b){

        if(firstTime == true){
          for (int i=0; i<n; i++)
                 for (int j=0; j<n; j++)
                        if (constraints[i][j] == 1){
                                var_a = i;
                                var_b = j;
                                // Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη
μεταβλητών όπου υπάρχει περιορισμός
                                generate_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);
                                prevconstraints[i][j] = 1;
                        }
        }
```

```
		if(firstTime == false){
		  for (int i=0; i<n; i++)
			for (int j=0; j<n; j++)
				if (constraints[i][j] == 1 && prevconstraints[i][j] != 1){
					var_a = i;
					var_b = j;
					// Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη μεταβλητών όπου υπάρχει περιορισμός και δεν έχει δημιουργηθεί ξανά
					generate_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);
					prevconstraints[i][j] = 1;
				}
		}


		  for (int i=0; i<n; i++)
			for (int j=0; j<n; j++)
				if (constraints[i][j] == 1){
					var_a = i;
					var_b = j;
					//print_tuples(constraint_matrices, total_tuples, k, d, var_a, var_b);
				}


}



int _tmain(int argc, _TCHAR* argv[])
{
		int ****constraint_matrices, **constraints, **prevconstraints, total_tuples, total_constraints, k, d, n, var_a=-1, var_b=-1, problems, addConstrains;
		int *solve, *prevSolv, repeat_times, repeat_Min_Conf;
		int conflicts=0, countMinConf=0, countRandVar=0;

		int *previous, *initSolv;
		int *SDistances, *SConflicts;

		float Z=-1; // Z=min(w1*normConflicts + w2*normDistance) : Στόχος αλγόριθμου δύο φάσεων η ελαχιστοποίηση της τιμής του Z->0 βέλτιστη λύση
		float solutionZ=-1;

		float tpers;



  cout << endl<< "		Min-Conflicts  - TwoPhase		" << endl;
  cout <<	"_____________________________________________" << endl<< endl;

  cout << "Enter the number of Variables: ";
  cin >> n;
  cout << endl;

  float maxn = n*n/2 - n;
  printf("\n(Max constrains= %.0f) ", maxn);
  cout << "Enter the number of constrains -> first CSP: ";
  cin >> total_constraints;
  cout << endl;

  cout << "Enter the number of values: ";
  cin >> d;
  cout << endl;

  cout << "Enter the percentage (%) of satisfy constrains: ";
  cin >> tpers;
  cout << endl;

  total_tuples = ceil(((float)d*d)*tpers/100);
  k=2; // temp[0] and temp[1]

  cout << "How many times select (violated) random variable to evaluate: ";
  cin >> repeat_times;
  cout << endl;

  cout << "How many times run Min_Conflicts: ";
  cin >> repeat_Min_Conf;
  cout << endl;
```

```
  cout << "How many Problems to Solve: ";
  cin >> problems;
  cout << endl;

  cout << "How many Constrains be added in every new Problem: ";
  cin >> addConstrains;
  cout << endl;


//_____________ Random SEED to Srand() for each problem - Το ίδιο Seed με τον MinConflict _____________
        FILE *fpr;
        int pot = 0;       // point of text
        //int *RandSeed;      // random numbers unique for each problem
        int RandSeed;
        int num;

        //RandSeed = (int *)malloc(problems * sizeof(int ));

        char outputFilename[] = "../../seed.txt"; // path: ..\Visual Studio 2010\Projects

        fpr = fopen(outputFilename, "r");

        if (fpr == NULL){
                fprintf(stderr, "Can't open output file %s!\n", outputFilename);
                exit(1);
        }

   //while( fscanf(fpr, "%d", &num) != EOF )           RandSeed[pot++] = num;

        while( fscanf(fpr, "%d", &num) != EOF )  RandSeed = num;

        //for (int i=0; i<problems; i++)
        printf("numbers= %d \n", RandSeed);

        fclose(fpr);
//_____________________________________________________________________________


        float w1=1.0, w2=0;
        int tconstraints = total_constraints;

        while(w1 > 0.8){                  //w1 -> conflicts w2-> distance

                printf("------------------------------  \n");
                printf("\n w1 = %f , w2 = %f\n", w1, w2);
                printf("------------------------------  \n");

                bool firstTime = true, firstTimeMinConf;
                total_constraints = tconstraints;

        solve = (int *)malloc(n * sizeof(int ));
        prevSolv = (int *)malloc(n * sizeof(int ));       // Προηγούμενη Λύση
        initSolv = (int *)malloc(n * sizeof(int ));    // προηγούμενη λύση προς αρχικοποίηση Min_Conflicts
        constraints = (int **)malloc(n * sizeof(int *));
        prevconstraints= (int **)malloc(n * sizeof(int *));
        constraint_matrices = (int ****) malloc( n * sizeof(int ***) );


        previous = (int *)malloc(4 * sizeof(int ));
        // prevcountMinConf=previous[0], prevconflicts=previous[1], prevcountRandVar=previous[2]


        SDistances  = (int *)malloc(problems * sizeof(int ));
        // SDistances: πίνακας για τον υπολογισμό του MO των αποστάσεων


        SConflicts  = (int *)malloc(problems * sizeof(int ));
        //SConflicts: πίνακας για τον υπολογισμό του MO των παραβιάσεων


   Initialize_constraints(constraints, n);
        Initialize_constraints(prevconstraints, n);
        Initialize_constraints(constraint_matrices, d,n);

        int problem=0;
```

```
		while (problem != problems){

			srand( RandSeed );    // initialize random seed

			// Δημιουργία Πίνακα ΠΕΡΙΟΡΙΣΜΩΝ
			gen_Constraints(constraints, n, k, total_constraints, addConstrains, firstTime);


			// Δημιουργία Πίνακα "ικανοποίησης περιορισμών" μόνο σε ζεύγη μεταβλητών όπου υπάρχει περιορισμός
			gen_Tuples(constraints, prevconstraints, firstTime, n, constraint_matrices, total_tuples, k, d, var_a, var_b);


			firstTimeMinConf = true;


			solutionZ = TwoPhase(solve, initSolv, prevSolv, constraints, constraint_matrices, d,n, repeat_times, repeat_Min_Conf, firstTime, firstTimeMinConf, Z, conflicts, total_constraints, countMinConf, countRandVar,  previous, SDistances, SConflicts, problem,w1, w2);

			/*if(solutionZ==0) printf("\nGOAL!!! -> Z=0 Conflicts=0 and Distance=0 \n");

			else{
				printf("\nConflicts= %d \n", SConflicts[problem]);
				printf("\nDistance= %d \n", SDistances[problem]);
			}*/

			problem = problem + 1;
			firstTime = false;

			//printf("\ntotal_constraints = %d \n", total_constraints);
			total_constraints = total_constraints + addConstrains;


			if(total_constraints > (n*n-n)/2){
				printf("\n A lot of Constrains!!!!\n");
				printf("\nConstrains = %d \n", total_constraints);
				break;
			}

			//system("PAUSE");
		}


//		//Υπολογισμός Μ.Ο. αποστάσεων και Μ.Ο. παραβιάσεων
//  int MODistances=0, MOConflicts=0;
//
//  printf("\n\n");
//  for (int p=0; p < problems; p++){				///----------**********************
		//   MOConflicts = MOConflicts + SConflicts[p];
		//   printf("SConflicts[%d]= %d \n",p, SConflicts[p]);
		//}

//   printf("\n\n");
//  for (int p=1; p < problems; p++){				///----------**********************
		//		MODistances = MODistances + SDistances[p];
		//		printf("SDistances[%d]= %d \n",p, SDistances[p]);
		//}


//
//   printf("\n\n----- Problems = %d ------\n", problems);
		//printf("Avg.Distances  =  %.3f  \n",  (float)MODistances/(problems-1));		///----------**********************
		//printf("Avg.Conflicts = %.3f \n", (float)MOConflicts/(problems-1));


//-------------------------------------------------------------------------------------------------------------------------------------------
		//Υπολογισμός Μ.Ο. αποστάσεων και Μ.Ο. παραβιάσεων
  int MODistances=0, MOConflicts=0;

  int total_solv_problems = 0;	// Total solved Problems
  int cri_point_conf = 0;			// κρίσιμο σημείο conflicts
  int cri_point_dist = 0;			// κρίσιμο σημείο distance
```

```
	float before_cri_avg_dist = 0;  // Πριν το κρίσιμο σημείο avg distance
	bool cri_point = false;
	bool cont_solv = false;
	int cri_sum_dist=0;


	printf("\n\n");

	for (int p=0; p < problems; p++){
		MOConflicts = MOConflicts + SConflicts[p];
		printf("SConflicts[%d]= %d \n",p, SConflicts[p]);
		if(SConflicts[p]==0 && cri_point==false) total_solv_problems++;
		if(SConflicts[p]>0 && cri_point==false) { cri_point_conf=SConflicts[p];  cri_point=true; }
	}


	printf("\n\n");

	for (int p=1; p < problems; p++){
			MODistances = MODistances + SDistances[p];
			printf("SDistances[%d]= %d \n",p, SDistances[p]);
			if(SConflicts[p]>0 && cri_point==true) { cri_point_dist=SDistances[p];  cri_point=false; }
	}


	for (int t=1; t < total_solv_problems; t++){
			cri_sum_dist = cri_sum_dist + SDistances[t];
			before_cri_avg_dist = (float)cri_sum_dist/(total_solv_problems-1);
	}

	printf("cri_sum_dist = %d \n", cri_sum_dist);

	 printf("\n----- Problems = %d ------\n", problems);

		printf("Total solved Problems = %d \n", total_solv_problems);
		printf("Before critical point avg distance = %.3f \n", before_cri_avg_dist);

		printf("critical point conflicts = %d \n", cri_point_conf);
		printf("critical point distance = %d \n", cri_point_dist);

		printf("CSPs.Avg.Conflicts = %.3f \n", (float)MOConflicts/(problems-1));
		printf("CSPs.Avg.Distances = %.3f \n", (float)MODistances/(problems-1));
//-------------------------------------------------------------------------------------------------------------------------------------------------------------
			free(solve);
			free(prevSolv);
			free(initSolv);
			free(previous);
			free(SDistances);
			free(SConflicts);

			for(int i = 0; i < n; i++)free(constraints[i]);
			free(constraints);
			for(int i = 0; i < n; i++)free(prevconstraints[i]);
			free(prevconstraints);
			for(int i = 0; i < n; i++){
				for(int j = 0; j < n; j++){
					for(int k = 0; k < d; k++){
						free(constraint_matrices[i][j][k]);
					}
				}
			}
			free(constraint_matrices);

			printf("------------------------------  \n");
			printf("\n w1 = %f , w2 = %f\n", w1,w2);
			printf("------------------------------  \n");

			w1 = w1 - 0.005;
			w2 = w2 + 0.005;
			//system("PAUSE");
		}

		system("PAUSE");

		return 0;
}
```