ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΠΕΙΡΑΜΑΤΙΚΗ ΜΕΛΕΤΗ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ BREAK OUT

ΜΠΑΤΣΟΥ ΜΑΡΙΑ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ:

ΔΡ. ΚΩΣΤΑΝΤΙΝΟΣ ΣΤΕΡΓΙΟΥ

ΚΟΖΑΝΗ ΙΟΥΝΙΟΣ 2016

Περίληψη

Στην εργασία αυτή, υλοποιούνται οι αλγόριθμοι min conflicts και break out στην γλώσσα προγραμματισμού Scala και συγκρίνονται πάνω σε προβλήματα ικανοποίησης περιορισμών και συγκεκριμένα σε graph coloring. Ο min conflicts είναι ο βασικός hill climbing αλγόριθμος και ο break out είναι μια παραλλαγή του. Ενώ το μεγαλύτερο πρόβλημα του min conflicts είναι ότι μπορεί να «κολλήσει» σε τοπικό ελάχιστο, ο break out έχει ένα τρόπο για να ξεφεύγει από τέτοιες καταστάσεις. Επιχειρείται λοιπόν η θεωρητική θεμελίωση του λόγου για τον οποίον ο Break Out έχει καλύτερη απόδοση από τον Min Conflicts καθώς και η υποστήριξη και απόδειξη αυτού με πειραματική μελέτη. Εξετάζεται το πόσο κοντά στην λύση μπορεί να φτάσει ο κάθε αλγόριθμος και σε πόσο χρόνο. Ακόμη, παρουσιάζονται μερικές έννοιες οι οποίες θα βοηθήσουν στην καλύτερη κατανόηση της πραγματευόμενης πειραματικής μελέτης καθώς και γιατί τα αποτελέσματά της, προέκυψαν κατά αυτόν τον τρόπο.

Abstract

In this paper, the Min Conflicts and Break Out algorithms are implemented and compared on constraints satisfaction problems and more specifically on graph coloring. Min Conflicts is the basic Hill Climbing algorithm and Break Out is another version of it. While the biggest problem on Min Conflicts is that it can get stuck in a local minimum situation, the Break Out algorithm has a way of escaping it. This is an attempt to find and explain the theoretical base on the reason why Break Out has a better performance than Min Conflicts and then back up these theories with experimental trials. The matter of these experiments is to examine how close to a solution each of the algorithms can get and in how much time. Furthermore, a few basic concepts on problem solving are presented in order to make the understanding of the said experimental trials easier.

Ευχαριστίες

Ευχαριστώ τον Δρ. Κωνσταντίνο Στεργίου για την πολύτιμη βοήθεια και καθοδήγησή του. Ακόμη ευχαριστώ τον συνάδελφο Δημοσθένη Μιχαηλίδη για τις καθοριστικές υποδείξεις του σχετικά με την γλώσσα προγραμματισμού Scala. Τέλος δεν θα μπορούσα να παραλείψω τους γονείς μου, τους οποίους ευχαριστώ για την στήριξή, την κατανόηση και την υπομονή τους.

ΠΕΡΙΕΧΟΜΕΝΑ

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κατάλογος Γραφημάτων

Κεφάλαιο 1

Εισαγωγή

Σε αυτό το κεφάλαιο παρουσιάζονται θέματα όπως το τι είναι η τεχνητή νοημοσύνη και ποιά είναι τα είδη των προβλημάτων που την ενδιαφέρουν. Εξετάζεται ο ορισμός και η δομή του προβλήματος. Δίνεται έμφαση στα προβλήματα ικανοποίησης περιορισμών και ειδικά στα graph coloring, τα οποία θα χρησιμοποιηθούν αργότερα για την εξέταση και σύγκριση των αλγορίθμων Break Out και Min Conflicts.

1.1 Τι είναι η τεχνητή νοημοσύνη

Η επιστήμη της τεχνητής νοημοσύνης ασχολείται με τους τρόπους με τους οποίους μπορεί να σχεδιαστεί ένα υπολογιστικό σύστημα ώστε να μιμηθεί όσο καλύτερα γίνεται την ανθρώπινη ευφυΐα. Τον τρόπο δηλαδή με τον οποίο ένας άνθρωπος σκέφτεται, μαθαίνει, επεξεργάζεται ερεθίσματα του περιβάλλοντός του, λύνει προβλήματα κλπ [1]. Δανείζεται πολλές αρχές των μαθηματικών, της ψυχολογίας, της φιλοσοφίας, της γλωσσολογίας, της τεχνολογίας υπολογιστών και των νευροεπιστημών. Σύμφωνα με τους Barr και Feigenbaum , είναι ο τομέας της επιστήμης των υπολογιστών, που ασχολείται με τη σχεδίαση ευφυών (νοημόνων) υπολογιστικών συστημάτων, δηλαδή συστημάτων που επιδεικνύουν χαρακτηριστικά που σχετίζουμε με την νοημοσύνη στην ανθρώπινη συμπεριφορά. Ακόμη, οι Russel και Norving, στο βιβλίο τους «Τεχνητή νοημοσύνη, μια σύγχρονη προσέγγιση», το οποίο αποτέλεσε βασικό βοήθημα αυτής της εργασίας, χαρακτηρίζουν την τεχνητή νοημοσύνη ως την επιστήμη η οποία όχι μόνο επιχείρησε να κατανοήσει αλλά και να κατασκευάσει νοήμονες οντότητες [2].

Η ιστορία της ξεκινά το 1943, όταν ο McCulloch, νευροεπιστήμονας και ο Pitts, επιστήμονας της λογικής [3], δημοσιεύουν την εργασία τους και καταφέρνουν να αποδείξουν ότι ένα δίκτυο νευρώνων συμπεριφέρεται όπως οι δικτυακές δομές. Υποστηρίζουν ακόμη ότι κάποια δίκτυα μπορούν να αποκτήσουν την δυνατότητα της

μάθησης. Αυτήν, χαρακτηρίζεται ως η πρώτη επίσημη εργασία σχετική με τεχνητή νοημοσύνη [2].

Επτά χρόνια αργότερα, το 1950, ο Alan Turing, πρωτοπόρος της εποχής στα μαθηματικά και στην κρυπτογραφία, συνθέτει ένα τεστ, το οποίο αποφασίζει αν αυτός που εξετάζεται είναι άνθρωπος ή μηχανή [2]. Είναι ουσιαστικά ένα παιχνίδι ερωτήσεων και απαντήσεων με όνομα "Can Machines Think?" (Μπορούν οι μηχανές να σκεφτούν;). Ένας άνθρωπος κάνει μια γραπτή ερώτηση και από την απάντηση που θα λάβει, προσπαθεί να καταλάβει αν ο συνομιλητής του είναι άνθρωπος ή μηχανή. Αν η τεχνητή νοημοσύνη καταφέρει να πείσει τον εξεταστή ότι μιλάει με άλλον άνθρωπο για πάνω από το 30% της διάρκειας της συνομιλίας, τότε έχει περάσει το τεστ. Η τεχνική αυτή χρησιμοποιείται μέχρι σήμερα και επιτυχημένη τεχνητή νοημοσύνη θεωρείται αυτήν που θα καταφέρει να περάσει το τεστ. Μέχρι την συγγραφή της παρούσας εργασίας (Αύγουστος 2016) το τεστ Turing θεωρείται από πολλούς ότι έχει περαστεί επιτυχώς μόνο από το πρόγραμμα υπολογιστή Eugene Goostman (Chatbot), τον Ιούνιο του 2014. Κατάφερε να πείσει 33% των εξεταστών, οι οποίοι για πέντε λεπτά συνομιλούσαν με τον Eugene αλλά ταυτόχρονα και με ανθρώπους, σε 300 συνολικά συνομιλίες. Πήραν μέρος 30 εξεταστές (10 συνομιλίες ο καθένας) οι οποίοι μπορούσαν να ρωτήσουν ότι ήθελαν. Ο Eugene αναπτύχθηκε από τους Vladimir Veselov και Eugene Demchenko, σε μια προσπάθεια που ξεκίνησε το 2001 θέλοντας να πείσουν ότι ο Eugene είναι ένα δεκατριάχρονο αγόρι που τα ξέρει όλα [4]. Κάποιοι, αμφισβητούν την επιτυχία του Eugene λέγοντας ότι δεν μπορεί να βγει αξιόλογο συμπέρασμα μετά από τόσο μικρή χρονική διάρκεια διαλόγων.

Η επιστήμη αυτή εκτός από πολύ ενδιαφέρουσα φαίνεται και πολλά υποσχόμενη. Όμως καθώς εξελίσσεται η τεχνολογία και μαζί της και η τεχνητή νοημοσύνη, η συζήτηση γύρω από τα θέματα ηθικής που προκύπτουν, φουντώνει. Ο Stephen Hawking φαίνεται να πιστεύει πως η τεχνητή νοημοσύνη του μέλλοντος θα είναι τόσο καλή στο να επιτυγχάνει τους στόχους της, που αν αυτοί δεν συμβαδίζουν πλέον με των ανθρώπων, τότε θα υπάρχει σοβαρό πρόβλημα [5]. Προκύπτει ακόμα και το ερώτημα του για ποιον θα είναι διαθέσιμη. Όσο πιο πλούσιος κανείς, τόσο πιο πιθανό θα είναι να διαχειριστεί ευρήματα της τεχνητής νοημοσύνης? Τι σημαίνει αυτό για τους υπόλοιπους? Δυστυχώς η περαιτέρω ανάλυση αυτού του θέματος είναι έξω από τα πλαίσια της παρούσας εργασίας.

1.2 Βασικά στοιχεία προβλημάτων και απλοί αλγόριθμοι αναζήτησης

Ένα πρόβλημα μπορεί να οριστεί ως μια κατάσταση η οποία χρήζει αντιμετώπισης, απαιτεί λύση και η δε λύση της δεν είναι προφανής ή γνωστή. Ένα καλά διατυπωμένο και δομημένο πρόβλημα, κάνει τον δρόμο προς την λύση πολύ πιο εύκολη υπόθεση και βελτιώνει τις πιθανότητες εύρεσης απάντησης. Ο όρος δομή, αναφέρεται στα στοιχεία ενός προβλήματος και τους τρόπους που αυτά μπορεί να συνδέονται μεταξύ τους. Μερικές κατηγορίες περιγράφονται συνοπτικά παρακάτω.

Επιλύσιμα: Αυτά για τα οποία είναι αποδεδειγμένο ότι υπάρχει λύση ή προβλήματα με παρόμοια διατύπωση με αυτή κάποιου αποδεδειγμένα επιλύσιμου.

Ανοικτά: Η λύση τους δεν έχει βρεθεί αλλά δεν έχει αποδειχθεί ότι δεν έχουν λύση.

Άλυτα: Είναι τα προβλήματα εκείνα για τα οποία έχει αποδειχθεί ότι δεν υπάρχει λύση.

Δομημένα: Η επίλυση τους προέρχεται από συγκεκριμένη διαδικασία. Πχ δευτεροβάθμια εξίσωση.

Ημιδομημένα: Αυτά στα οποία η λύση αναζητείται μέσα σε ένα εύρος πιθανών λύσεων. Πχ Η επιλογή διαδρομής από το σπίτι στο σχολείο.

Αδόμητα: Οι λύσεις τους δεν μπορούν να δομηθούν. Σε αυτές τις περιπτώσεις ο ανθρώπινος παράγοντας παίζει πολύ σημαντικό ρόλο. Πχ πως θα οργανωθεί ένα πάρτι.

Απόφασης: Όπου η λύση έχει την μορφή «ναι» ή «όχι» και προσπαθεί να διαπιστωθεί αν υπάρχει απάντηση που να ικανοποιεί τα δεδομένα του προβλήματος.

Υπολογιστικά: Όπου χρειάζονται υπολογιστικές διαδικασίες για να βρεθεί απάντηση.

Ο λόγος για την ανάθεση επίλυσης προβλημάτων σε υπολογιστές είναι συνήθως η πολυπλοκότητα υπολογισμών, η επαναληπτικότητα διαδικασιών και το μεγάλο πλήθος δεδομένων. Ακόμη, ένας υπολογιστής μπορεί να εκτελεί εντολές με πολύ μεγάλη ταχύτητα [6].

Η διαδικασία με την οποία επιλύονται προβλήματα ονομάζεται αλγόριθμος. Πιο αυστηρά, ως αλγόριθμος ορίζεται μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος [6]. Για την ανάλυση ενός αλγορίθμου και τον καθορισμό της

αποδοτικότητάς του, χρησιμοποιείται ο όρος πολυπλοκότητα. Η πολυπλοκότητα μπορεί να μετράται σε σχέση με τον χρόνο- άρα χρονική, ή σε σχέση με την μνήμη- χωρική. Ο χρόνος που χρειάζεται για να λυθεί ένα πρόβλημα μπορεί να είναι γραμμικός, εκθετικός, λογαριθμικός ή πολυωνυμικός [7]. Όταν είναι πολυωνυμικός ($O(n^k)$), το πρόβλημα ανήκει στην κλάση πολυωνυμικών προβλημάτων P και θεωρείται εύκολο, αν δεν λύνεται σε πολυωνυμικό χρόνο τότε είναι κλάσης NP και θεωρείται δύσκολο. Υπάρχει ακόμη και η υποκατηγορία NP-hard, που αναφέρεται στα πάρα πολύ δύσκολα προβλήματα [2].

Σημαντικός όρος γι' αυτή την ενότητα, είναι το θεώρημα της μη πληρότητας του Godel. Με απλά λόγια και σε σχέση με τα υπολογιστικά προβλήματα το θεώρημα αυτό μπορεί να ερμηνευθεί ως η απόδειξη ότι ακόμη και να υπάρχει λύση σε ένα πρόβλημα, δεν είναι σίγουρο ότι θα βρεθεί. Οι Russell και Norving, απλουστεύοντας το θεώρημα γράφουν ότι «η βελόνα μπορεί να βρίσκεται μέσα στα άχυρα, αλλά καμία διαδικασία δεν εγγυάται ότι θα βρεθεί» [2]. Το θεώρημα της μη πληρότητας, έχει χρησιμοποιηθεί για να αποδείξει ότι ένας υπολογιστής, δεν θα καταφέρει ποτέ να φτάσει την ανθρώπινη ευφυΐα [8].

Για την επίλυση προβλημάτων τεχνητής νοημοσύνης χρησιμοποιούνται πολύ συχνά οι λεγόμενοι πράκτορες επίλυσης προβλημάτων. Οι πράκτορες διαθέτουν μηχανισμούς αναγνώρισης του περιβάλλοντος στο οποίο βρίσκονται καθώς και μηχανισμούς δράσης. Αξιολογούν την κατάσταση και αποφασίζουν πιο θα είναι το επόμενο βήμα στην αναζήτηση της λύσης [9]. Ένας πράκτορας είναι υπεύθυνος για την διατύπωση του στόχου, ώστε να μπορέσει να οργανώσει τις ενέργειές του. Ακόμη, πρέπει να διατυπώσει το πρόβλημα έτσι ώστε να ξέρει ποιες καταστάσεις πρέπει να αξιολογήσει. Τώρα είναι έτοιμος να ξεκινήσει την αναζήτηση της λύσης. Θα δημιουργήσει λοιπόν μια σειρά ενεργειών που θα τον οδηγούν στο επιθυμητό αποτέλεσμα. Το τελευταίο που έχει να κάνει, είναι να εκτελέσει αυτή τη σειρά ενεργειών ώστε να φτάσει στον στόχο [2]. Το πόσο έξυπνος είναι ένας πράκτορας καθορίζεται από τον δημιουργό του και από το πόσες πληροφορίες θα του δώσει. Μερικοί πράκτορες έχουν όμως και την δυνατότητα μάθησης αφού κατά την αλληλεπίδρασή τους με τον χώρο καταστάσεων μπορούν να αποκτήσουν πρόσθετες πληροφορίες και να βελτιώσουν την συμπεριφορά τους μέσω της επανάληψης. Αυτοί ονομάζονται πράκτορες μάθησης [10]. Τα προγράμματα πράκτορα αντιστοιχίζουν την γνώση από τις πληροφορίες αυτές, με

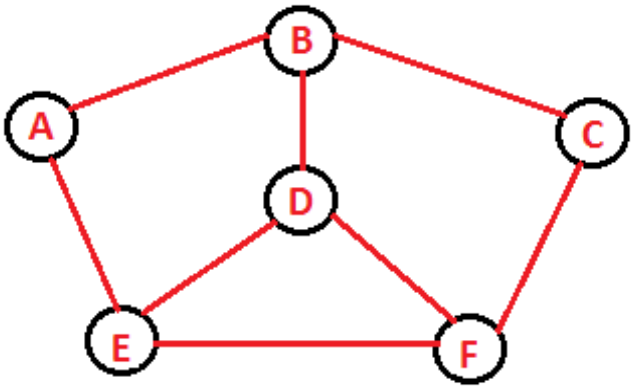
ενέργειες [2]. Παρακάτω βλέπουμε τον ρόλο που παίζουν οι πράκτορες στην δόμηση ενός προβλήματος.

Σ’ αυτή την εργασία υιοθετήθηκε το μοντέλο δόμησης προβλημάτων που παρουσιάζουν οι Norving και Russell. Πρώτα καθορίζεται η αρχική κατάσταση, από πού δηλαδή θα ξεκινήσει η διερεύνηση. Μετά αποφασίζεται μια συνάρτηση διαδόχων, η οποία δείχνει τις επιτρεπτές ενέργειες για κάθε κατάσταση. Επίσης ορίζεται ο χώρος καταστάσεων, δηλαδή οι θέσεις που μπορεί να βρεθεί ο πράκτορας ξεκινώντας από την αρχική κατάσταση και κινούμενος σύμφωνα με την συνάρτηση διαδόχων. Ακολουθεί ο έλεγχος στόχου, όπου αξιολογείται αν μια κατάσταση είναι η κατάσταση στόχου, δηλαδή η λύση. Τέλος είναι απαραίτητη μια συνάρτηση που να αναθέτει ένα κόστος σε κάθε διαδρομή. Αυτήν ονομάζεται συνάρτηση κόστους διαδρομής. Μια τέτοια θα μπορούσε να είναι το άθροισμα των κόμβων που επισκέφτηκε ο πράκτορας μέχρι τον στόχο [2].

Τέτοιου είδους δομή έχουν και τα προβλήματα αναζήτησης που ενδιαφέρουν άμεσα την εργασία αυτή. Μια πολύ γνωστή κατηγορία αλγορίθμων αναζήτησης είναι αυτοί της τυφλής αναζήτησης οι οποίοι χρησιμοποιούνται όταν δεν υπάρχουν πληροφορίες που επιτρέπουν την αξιολόγηση των καταστάσεων. Σ’ αυτού του είδους αλγορίθμους συγκαταλέγονται η αναζήτηση πρώτα σε βάθος, η αναζήτηση πρώτα σε πλάτος, η επαναληπτική εκβάθυνση, η αναζήτηση διπλής κατεύθυνσης, η επέκταση και οριοθέτηση, και η ακτινωτή αναζήτηση [10].

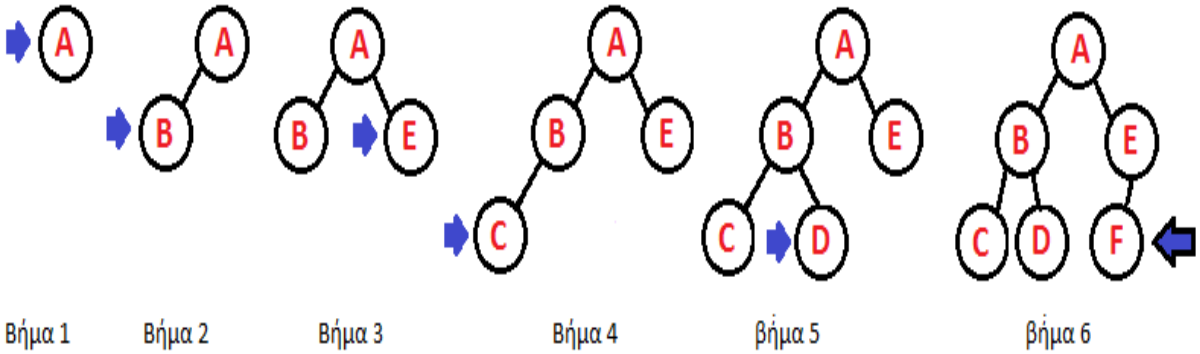
Ως παράδειγμα θα εξεταστεί ο αλγόριθμος της αναζήτησης πρώτα σε πλάτος ο οποίος βρίσκει πάντα την καλύτερη λύση, όμως έτσι προκαλεί το μειονέκτημα ότι ο χώρος αναζήτησης μπορεί να γίνει πάρα πολύ μεγάλος [11]. Για να περιγραφεί καλύτερα η λειτουργία του, χρησιμοποιούνται δένδρα. Έτσι, επεκτείνεται πρώτα ο κόμβος ρίζα και μετά επεκτείνονται όλοι οι διάδοχοί του ξεκινώντας από αριστερά προς τα δεξιά. Κάθε φορά ελέγχεται αν ο κόμβος που επεκτάθηκε είναι ο στόχος. Πρώτα δηλαδή επεκτείνονται όλοι οι κόμβοι του πρώτου επιπέδου του δένδρου, μετά του δεύτερου επιπέδου κοκ [2].

Στο παρακάτω γράφημα αναζητείται μια διαδρομή από τον κόμβο A στον κόμβο F.



Εικόνα 1 Γράφημα αναζήτησης διαδρομής

Κάθε νέο δένδρο δείχνει το επόμενο βήμα της αναζήτησης πρώτα σε πλάτος:



Εικόνα 2 Επεκτάσεις κόμβων στην αναζήτηση πρώτα σε πλάτος

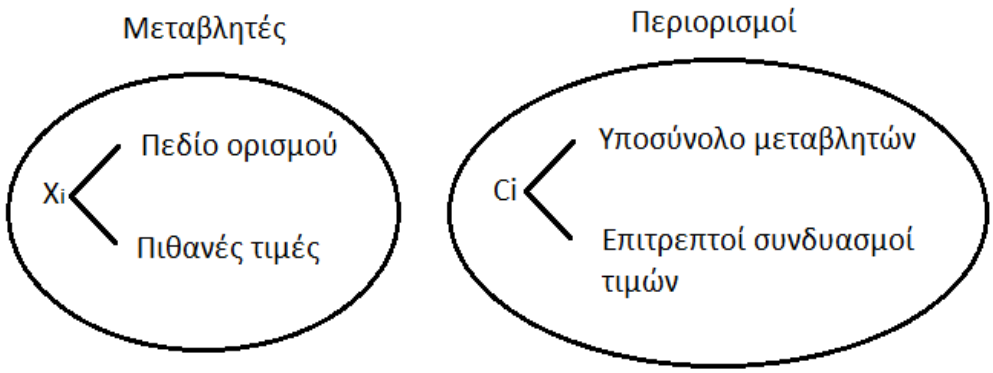
Μετά από 6 βήματα, βρίσκει την κατάσταση στόχου άρα η λύση είναι η διαδρομή A- E- F. Όπως ήταν φανερό και από το γράφημα, είναι η πιο σύντομη διαδρομή, οπότε είναι η βέλτιστη λύση. Όπως αναφέρθηκε παραπάνω σε τέτοιου είδους προβλήματα, δεν υπάρχει τρόπος να αξιολογηθεί μια κατάσταση. Για παράδειγμα δεν υπάρχει κάποια πληροφορία η οποία να επιβεβαιώνει ότι ο κόμβος B είναι καλύτερος από τον E για ανάπτυξη ή το αντίθετο. Γι αυτό και εξετάζονται όλοι οι κόμβοι με την σειρά μέχρι να βρεθεί ο στόχος.

Το πρόβλημα αυτό είναι εκθετικής πολυπλοκότητας που σημαίνει ότι αν εξετάζονταν ένα μεγαλύτερο παράδειγμα όπου ο στόχος ήταν σε μεγάλο βάθος, οι απαιτήσεις χώρου και χρόνου θα γίνονταν πάρα πολύ μεγάλες.

1.3 Προβλήματα ικανοποίησης περιορισμών

Σε ένα πρόβλημα ικανοποίησης περιορισμών υπάρχει μια ομάδα μεταβλητών, όπου στην κάθε μεταβλητή, ανατίθεται μια τιμή μέσα από ένα πεδίο ορισμού [12]. Υπάρχει ένα σύνολο περιορισμών, οι οποίοι επιβάλλουν σχέσεις μεταξύ των μεταβλητών. Για να επιτευχθεί λύση, πρέπει να ικανοποιηθούν όλοι οι περιορισμοί.

Για να περιγραφούν τέτοιου είδους προβλήματα, χρησιμοποιούνται οι όροι κατάσταση και χώρος καταστάσεων. Ως κατάσταση, ορίζεται μια ανάθεση τιμών από το πεδίο ορισμού, σε όλες τις μεταβλητές, άσχετα από το αν ικανοποιεί ή όχι τους περιορισμούς [2]. Ο χώρος καταστάσεων είναι όλες οι καταστάσεις μαζί, εκεί δηλαδή που τελικά λαμβάνει χώρα η αναζήτηση της λύσης. Επομένως, η κατάσταση λύσης, πρέπει να ικανοποιεί το σύνολο των περιορισμών [13]. Στα πλαίσια των προβλημάτων αναζήτησης που εξετάζει η εργασία, ο χώρος είναι πεπερασμένος και δεν ενδιαφέρει το μονοπάτι προς την λύση. Στα προβλήματα ικανοποίησης περιορισμών ο χώρος καταστάσεων αυξάνεται γραμμικά ανάλογα με το μέγεθος του προβλήματος [14].



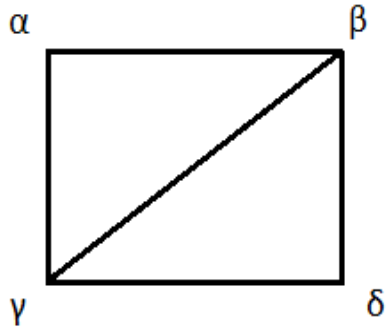
Εικόνα 3 Αναπαράσταση δομής προβλημάτων ικανοποίησης περιορισμών

Σε εργασία του Lemaitre και Verfaillie, ο όρος πρόβλημα ικανοποίησης περιορισμών, περιγράφεται ως ένα σύνολο (X,C,D), όπου το $X=\{x_1,\ldots,x_n\}$ είναι ένα σετ μεταβλητών, το $D=\{d_1,\ldots,d_n\}$ είναι ένα σετ πεπερασμένων πεδίων ορισμού για τις μεταβλητές και C είναι ένα σετ περιορισμών. Ένας περιορισμός $c=(X_c, R_c)$ ορίζεται ως ένα υποσύνολο X_c των μεταβλητών του X και ένα υποσύνολο R_c επιτρεπόμενων ζευγών τιμών. Μια λύση του προβλήματος είναι η ανάθεση τιμών σε όλες τις μεταβλητές, έτσι ώστε να ικανοποιούνται όλοι οι περιορισμοί. [15]

Αν ένας περιορισμός αποτελείται από μία μεταβλητή, χαρακτηρίζεται ως μοναδιαίος. Αν αποτελείται από 2, ως δυαδικός και αν αποτελείται από παραπάνω, ως ανώτερης τάξης. Τα προβλήματα που εξετάζονται σε αυτή την εργασία περιλαμβάνουν μόνο δυαδικούς περιορισμούς.

Οποιοδήποτε πρόβλημα ικανοποίησης περιορισμών μπορεί να αναπαρασταθεί με ένα γράφημα, όπου κάθε γραμμή, αναπαριστά έναν περιορισμό μεταξύ δύο μεταβλητών ή αλλιώς, κόμβων. [2]

Παράδειγμα: Αν δίνονταν πρόβλημα με το παρακάτω σύνολο περιορισμών: $X=\{\alpha\neq\beta, \alpha\neq\gamma, \beta\neq\delta, \beta\neq\gamma, \gamma\neq\delta\}$, τότε το γράφημα του θα ήταν το παρακάτω. Οι περιορισμοί συμβολίζονται με τα ευθύγραμμα τμήματα που ενώνουν τους κόμβους που αντιστοιχούν στον κάθε ένα.



Εικόνα 4 Το μη κατευθυνόμενο γράφημα περιορισμών

Το πόσο πολύπλοκη μπορεί να είναι η διαδικασία επίλυσης ενός τέτοιου προβλήματος, εξαρτάται άμεσα από την δομή του γραφήματος περιορισμών του. Αυτά με δομή δένδρου, λύνονται σε γραμμικό χρόνο [2].

Γνωστά προβλήματα ικανοποίησης περιορισμών είναι το n-queens, οι πύργοι Ανόι κλπ. Παραδείγματα προβλημάτων στην καθημερινή ζωή μπορεί να χαρακτηριστούν αυτά σχετικά με κατανομή εργασιών, κατανομή προσωπικού, χρονοπρογραμματισμό ή διαχείρισης ενός ωρολογίου προγράμματος [2].

Παράδειγμα: Έστω ότι θέλουμε να βρούμε την σειρά με την οποία θα καθίσουν 4 μαθητές A, B, C, D. Ο μαθητής A δεν θέλει να καθίσει δίπλα στον B και τον C, και ο μαθητής D δίπλα στον B. Έχουμε στην διάθεση μας 4 καρέκλες στην σειρά.

- Μεταβλητές: A, B, C, D (οι μαθητές)
- Πεδίο τιμών: 1,2,3,4 (οι καρέκλες)
- Περιορισμοί: [A-B]>1, [A-C]>1,[D-B]>1

Οι περιορισμοί παίρνουν αυτή την μορφή αφού οι καρέκλες στις οποίες θα καθίσουν οι μαθητές που συμμετέχουν στον εκάστοτε περιορισμό, δεν πρέπει να είναι συνεχόμενες. Η διαφορά δηλαδή των θέσεων πρέπει να είναι μεγαλύτερη από ένα.

Σε αυτό το πρόβλημα υπάρχουν 2 λύσεις. Η πρώτη είναι να καθίσουν οι μαθητές με αυτή την σειρά, A-D-C-B. Αφού λύση είναι μια πλήρης ανάθεση τιμών στις μεταβλητές η απάντηση διαμορφώνεται ως εξής: νστην μεταβλητή A να δοθεί η τιμή 1 από το πεδίο ορισμού, στην μεταβλητή B η τιμή 4, στην C η τιμή 3 και στην D η τιμή 2. Η δεύτερη λύση είναι να καθίσουν με αυτή τη σειρά, B-C-D-A. Δηλαδή η μεταβλητή A παίρνει την τιμή 4, η B την τιμή 1, η C την τιμή 2 και η D την 3. Σε οποιαδήποτε άλλη περίπτωση ανάθεσης τιμών, κάποιος από τους τρείς περιορισμούς δεν θα ικανοποιείται.

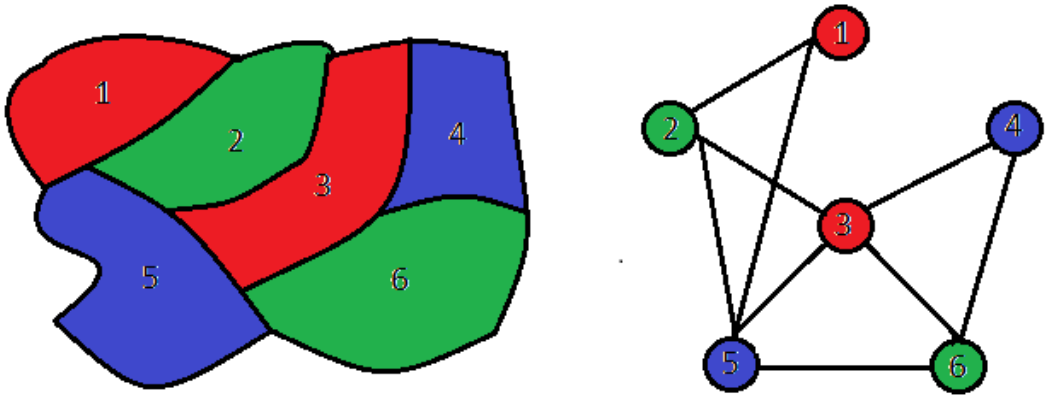
Για να φτάσει κανείς στις παραπάνω λύσεις μπορεί να κάνει δοκιμές, να χρησιμοποιήσει αλγόριθμους αναζήτησης, αλγόριθμους ελέγχου συνέπειας ή τοπική αναζήτηση. Ο αριθμός παραβιάσεων περιορισμών μπορεί να χρησιμοποιηθεί ως δείκτης του πόσο κοντά βρίσκεται η λύση [14]. Όσο πιο λίγες οι παραβιάσεις, τόσο πιο κοντά η λύση.

1.4 Προβλήματα Graph Coloring

Η θεωρία χρωματισμού γραφημάτων είναι πολύ σημαντική στον κλάδο των διακριτών μαθηματικών, αφού ασχολείται με το πώς μπορεί να χωριστεί ένα σετ αντικειμένων σε ομάδες- κλάσεις, σύμφωνα με κάποιους κανόνες. Οι κανόνες καθορίζουν το αν τα αντικείμενα ενός ζεύγους, μπορούν να ανήκουν στην ίδια κλάση ή όχι. Για να ξεχωρίζουν οι κλάσεις μεταξύ τους, χρησιμοποιούνται διαφορετικά χρώματα [19].

Προβλήματα χρωματισμού γραφήματος σύμφωνα με τον Bohlin [21], είναι αυτά που περιγράφονται ως εξής: σε ένα μη κατευθυνόμενο γράφημα (N, E), όπου N είναι ένα σύνολο κόμβων και E ένα σύνολο ακμών, γίνεται ανάθεση χρωμάτων στους κόμβους από ένα σύνολο k χρωμάτων, έτσι ώστε κόμβοι που συνδέονται, να μην έχουν το ίδιο χρώμα. Δηλαδή, γίνεται ανάθεση τιμών στα στοιχεία του γραφήματος, σύμφωνα με κάποιους κανόνες. Τα χρώματα ή αλλιώς τιμές, δίνονται μέσα από ένα πεπερασμένο πεδίο ορισμού και ο στόχος είναι να δοθούν έτσι ώστε να μην παραβιάζεται κανένας περιορισμός.

Αν για παράδειγμα έπρεπε να χρωματιστεί ο παρακάτω χάρτης ώστε καμία χώρα να μην έχει ίδιο χρώμα με την διπλανή της, και μπορούσαν να χρησιμοποιηθούν μόνο 3 χρώματα, θα προέκυπτε το παρακάτω σχήμα και το αντίστοιχο συνδεδεμένο γράφημα ως λύση.



Εικόνα 5 Πρόβλημα χρωματισμού γραφήματος

Σε αυτή την περίπτωση οι κόμβοι 1 έως 6 είναι οι μεταβλητές του προβλήματος και συμβολίζουν ο καθένας μια χώρα του χάρτη. Το πεδίο ορισμού αποτελείται από τρεις τιμές, τα τρία δηλαδή πιθανά χρώματα και οι κόμβοι μπορούν να πάρουν από εκεί τιμές, που υπαγορεύονται από τους περιορισμούς. Οι περιορισμοί που σχηματίστηκαν από την διατύπωση του προβλήματος και το σχήμα του χάρτη είναι οι: 1≠2, 1≠5, 2≠3, 5≠3, 3≠4, 3≠6, 5≠6, 4≠6, 2≠5 και ο κάθε ένας συμβολίζεται με μια ακμή στο γράφημα. Αυτό σημαίνει ότι οι κόμβοι που ενώνονται με ακμή δεν μπορεί να έχουν το ίδιο χρώμα.

Κεφάλαιο 2

Τοπική αναζήτηση

Οι αλγόριθμοι break out και min conflicts κατατάσσονται στους αλγορίθμους τοπικής αναζήτησης. Έτσι είναι πολύ σημαντικό να εξεταστεί η έννοια αυτή και να παρουσιαστούν παραδείγματα ώστε να μπορέσουν να γίνουν κατανοητά τα επόμενα μέρη της εργασίας.

2.1 Τοπική αναζήτηση και προβλήματα ικανοποίησης περιορισμών

Αυτό που κάνει την τοπική αναζήτηση να ξεχωρίζει από τις άλλες μορφές αναζήτησης είναι το γεγονός ότι το μονοπάτι προς την λύση δεν έχει καμία σημασία και ότι μπορεί να βρίσκει λύση ακόμη και σε προβλήματα με άπειρο χώρο καταστάσεων [2]. Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο για τον αλγόριθμο αναζήτησης πρώτα σε πλάτος, μόλις έβρισκε τον στόχο γνώριζε πως η διαδρομή μέχρι εκεί ήταν η λύση. Όμως πολλές φορές δεν ενδιαφέρει τόσο η διαδρομή όσο η τελική διάταξη. Για παράδειγμα μια τέτοια περίπτωση είναι το πρόβλημα των 8 βασιλισσών, όπου πρέπει να τοποθετηθούν 8 βασίλισσες στην σκακιέρα ώστε να μην απειλούνται μεταξύ τους. Δεν έχει σημασία δηλαδή με ποια σειρά θα τοποθετηθούν στην σκακιέρα αλλά ποια είναι η διάταξή τους.

Η τοπική αναζήτηση ξεκινά με μια πλήρη ανάθεση τιμών σε κάθε μεταβλητή και επαναληπτικά προσπαθεί να βελτιώσει τις αναθέσεις αυτές σε κάθε βήμα. Στα προβλήματα ικανοποίησης περιορισμών η επαναληπτική διαδικασία που ακολουθείται σε μορφή αλγορίθμου είναι η παρακάτω.

ΑΛΓΟΡΙΘΜΟΣ Τοπική Αναζήτηση

Inputs

V, ένα σύνολο μεταβλητών

dom, το πεδίο τιμών

C, ένα σύνολο περιορισμών προς ικανοποίηση

Outputs

πλήρης ανάθεση τιμών ώστε να ικανοποιούνται όλοι οι περιορισμοί

for each μεταβλητή X **do**

A[X] ← τυχαία μεταβλητή από το πεδίο ορισμού

while (το κριτήριο για να σταματήσουν οι επαναλήψεις δεν έχει επαληθευτεί & ο πίνακας A δεν είναι μια ικανοποιητική ανάθεση)

Select μια μεταβλητή Y και μια τιμή V από το πεδίο ορισμού

Set A[Y] ←V

if (ο πίνακας A είναι μια ικανοποιητική ανάθεση τιμών) **then**

return A

Ο παραπάνω αλγόριθμος ξεκινάει αναθέτοντας τυχαία τιμές από το πεδίο ορισμού, σε όλες τις μεταβλητές του προβλήματος, τις οποίες αποθηκεύει στον πίνακα A. Στον βρόγχο while γίνεται μια τοπική αναζήτηση μέσα στον χώρο των αναθέσεων ώστε να επιλεχθεί τυχαία μια μεταβλητή στην οποία αντιστοιχίζεται και μια νέα τιμή από το πεδίο ορισμού. Εάν μετά την νέα αυτή ανάθεση τιμής, ο πίνακας που προκύπτει ικανοποιεί όλους τους περιορισμούς, τότε αυτή είναι η λύση και έτσι επιστρέφεται ο πίνακας A. Αλλιώς συνεχίζεται η ίδια διαδικασία είτε μέχρι να επαληθευτεί το κριτήριο τερματισμού είτε αν πριν από αυτό έχει βρεθεί λύση. Το κριτήριο τερματισμού χρησιμοποιείται ώστε να αποφασιστεί το πότε θα σταματήσει η αναζήτηση και μπορεί να είναι απλά ο αριθμός βημάτων της επαναληπτικής διαδικασίας. Αν δεν οριστεί ένα τέτοιο κριτήριο και δεν υπάρχει λύση στο δοσμένο πρόβλημα, η εκτέλεση του αλγορίθμου δεν θα σταματήσει ποτέ [20].

Σε αυτό το σημείο πρέπει να εισαχθεί και ο ορός των ευριστικών μηχανισμών (heuristics). Είναι η μέθοδος σύμφωνα με την οποία αποφασίζεται το επόμενο βήμα της αναζήτησης [16]. Η επιλογή heuristic βασίζεται σε πληροφορίες για το εκάστοτε πρόβλημα και όταν είναι σωστή μπορεί να κάνει την διαδικασία της αναζήτησης ευκολότερη και γρηγορότερη. Στην περίπτωση των προβλημάτων ικανοποίησης περιορισμών, κάθε κατάσταση αξιολογείται από το επιλεγμένο heuristic για να διαπιστωθεί αν είναι η κατάσταση στόχου, δηλαδή η λύση. Το heuristic δεν μπορεί να εγγυηθεί ότι η λύση που βρέθηκε είναι η καλύτερη, αλλά είναι μια λύση από τις πολλές που μπορεί να υπάρχουν. Ακόμη η αναζήτηση μπορεί να μην επισκεφτεί ολόκληρο των

χώρο καταστάσεων ή να επισκεφτεί την ίδια κατάσταση πολλές φορές. Αυτό σημαίνει ότι μπορεί να ψάχνει για λύση απείρως και γι' αυτό όταν χρησιμοποιείται σε αλγορίθμους, ορίζεται ένας πεπερασμένος αριθμός βημάτων ή συγκεκριμένος χρόνος εκτέλεσης [17]. Το min conflict heuristic έχει αναγνωριστεί ως ένα από τα πιο ισχυρά heuristic για εύρεση μιας λύσης στα προβλήματα ικανοποίησης περιορισμών. Αυτό το heuristic, κατά την διαδικασία επιλογής τιμής μια μεταβλητής, επιλέγει αυτήν που ελαχιστοποιεί τον αριθμό παραβιάσεων των περιορισμών, σε σχέση με τις άλλες τιμές[18].

Στην τοπική αναζήτηση δεν υπάρχει λόγος να αποθηκεύονται διαδρομές. Αυτό που εξετάζεται κάθε φορά είναι η τρέχουσα κατάσταση. Έτσι η απαίτηση μνήμης είναι μικρή και σταθερή. Γενικά τέτοιου είδους αλγόριθμοι είναι ιδανικοί για προβλήματα βελτιστοποίησης όπου πρέπει να βρεθεί η καλύτερη λύση σύμφωνα με κάποια αντικειμενική συνάρτηση [2]. Πολύ συχνά χρησιμοποιείται το heuristic του min conflicts. Ακόμη, οι αλγόριθμοι τοπικής αναζήτησης δεν είναι πλήρεις. Αυτό όπως εξηγήθηκε και παραπάνω, σημαίνει ότι αν δεν υπάρχει λύση, θα ψάχνουν για πάντα (εκτός αν υπάρχει κάποια συνθήκη τερματισμού) ενώ αν υπάρχει λύση δεν είναι σίγουρο ότι θα την βρουν. Ο βρόγχος της επανάληψης εγκαταλείπεται είτε όταν η λύση είναι αρκετά καλή (σύμφωνα με το heuristic, πχ ανάλογα με το πόσοι περιορισμοί ικανοποιήθηκαν), είτε μετά από έναν προκαθορισμένο αριθμό επαναλήψεων. Η διαδικασία αυτή ονομάζεται επαναληπτική διόρθωση [21].

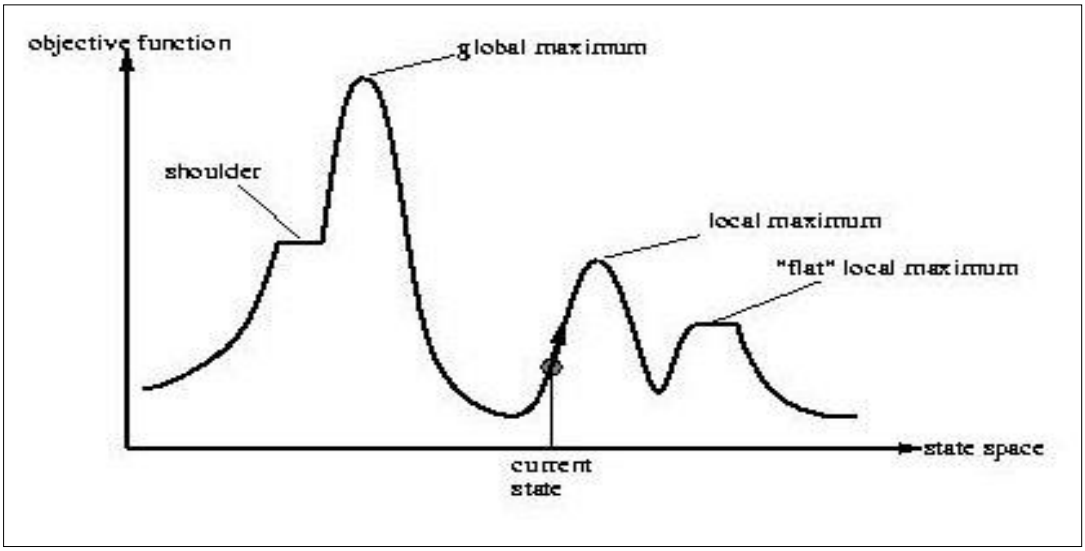
Η τοπική αναζήτηση περνάει πολύ χρόνο ψάχνοντας σε περιοχές του πεδίου καταστάσεων, που μπορεί να μην υπάρχει περίπτωση για βελτίωση. Αφού όμως βασίζεται στην τοπική βελτίωση, μπορεί να κολλήσει σε τοπικό ελάχιστο. Σε ένα πρόβλημα ικανοποίησης περιορισμών, αυτό σημαίνει ότι έχει βρεθεί σε μια κατάσταση όπου όλες οι αναθέσεις τιμών από το πεδίο ορισμού, δίνουν υψηλότερο κόστος από ότι η τρέχουσα τιμή της μεταβλητής [21]. Αυτό είναι και το κύριο πρόβλημα των αλγορίθμων τοπικής αναζήτησης. Υπάρχουν όμως και κάποιοι τρόποι για να «ξεφύγει» ένας αλγόριθμος από το τοπικό ελάχιστο. Μια κλασική τακτική είναι η τυχαία επανεκκίνηση σε άλλο σημείο του χώρου καταστάσεων, μετά από συγκεκριμένο αριθμό επαναλήψεων. Υπάρχει και ο τυχαίος περίπατος, όπου γίνεται μια τυχαία κίνηση στον χώρο αναζήτησης, με πιθανότητα σε κάθε επανάληψη δοσμένη από τον χρήστη [21].

2.2 Αλγόριθμοι τοπικής αναζήτησης

Σ' αυτή την υποενότητα περιγράφονται τέσσερις κλασικοί αλγόριθμοι τοπικής αναζήτησης. Παρουσιάζεται η βασική τους μορφή, υπάρχουν όμως και πολλές άλλες εκδοχές τους. Δίνεται περισσότερη έμφαση στην αναρρίχηση λόφων καθώς οι αλγόριθμοι break out και min conflicts, είναι παραλλαγές της.

Αναζήτηση με αναρρίχηση λόφων (Hill Climbing)

Εδώ ο αλγόριθμος Hill climbing εξετάζεται όπως χρησιμοποιείται σε προβλήματα ικανοποίησης περιορισμών. Όπως ισχύει για τους αλγόριθμους τοπικής αναζήτησης, ο hill climbing δεν διατηρεί ιστορικό των κινήσεών του. Σε κάθε επανάληψη αποθηκεύεται η τρέχουσα κατάσταση, δηλαδή μια μεταβλητή του προβλήματος καθώς και η τιμή που της αντιστοιχεί εκείνη την στιγμή. Η αναρρίχηση λόφων εξετάζει μόνο τι συμβαίνει με τους κοντινούς γείτονές της, δηλαδή εξετάζει μόνο τον χώρο καταστάσεων που προκύπτει αν η τρέχουσα μεταβλητή προς εξέταση, πάρει όλες τις πιθανές τιμές που αυτή μπορεί να πάρει. Για παράδειγμα, στο πρόβλημα των 8 βασιλισσών, η τρέχουσα μεταβλητή θα ήταν μία από τις 8 βασίλισσες και η τιμή της θα ήταν η θέση της πάνω στην σκακιέρα. Ο Hill Climbing εξετάζει τι θα γίνονταν αν η τρέχουσα βασίλισσα μετακινούνταν σε όλες τις γειτονικές τις θέσεις σε απόσταση ενός τετραγώνου γύρω από αυτήν. Μπορεί έτσι πολύ εύκολα έτσι να παγιδευτεί, αφού μπορεί καμία νέα θέση της βασίλισσας ,να μην οδηγεί σε καλύτερη κατάσταση και άρα δεν μπορεί να προκύψει καλύτερη επόμενη κίνηση. Αυτό συμβαίνει είτε γιατί πέφτει σε τοπικό μέγιστο (Εικόνα 6), είναι δηλαδή στην ψηλότερη κορυφή της γειτονιάς αλλά όχι και στην υψηλότερη όλου του χώρου καταστάσεων, είτε επειδή συναντά μια κορυφογραμμή, δηλαδή μια ακολουθία τοπικών μεγίστων, είτε συναντά ένα οροπέδιο. Οροπέδιο λέγεται η κατάσταση στην οποία ένα μέρος του χώρου καταστάσεων είναι επίπεδο, που σημαίνει ότι η συνάρτηση αξιολόγησης θα είναι επίπεδη και όλοι οι γείτονες θα δίνουν την ίδια τιμή [2].



Εικόνα 6 Τοπικό μέγιστο

Παρακάτω δίνεται ο hill climbing σε μορφή ψευδοκώδικα.

Function HILL-CLIMBING (πρόβλημα) **returns** μια κατάσταση τοπικού μεγίστου
Inputs: πρόβλημα, ένα πρόβλημα
Local variables: τρέχων, ένας κόμβος
γειτονικός, ένας κόμβος

τρέχων ← Make-Node (Initial-State [πρόβλημα])
loop do
γειτονικός ← διάδοχος με μεγαλύτερη τιμή από τον κόμβο τρέχων
if VALUE[γειτονικός] ≤ VALUE[τρέχων] **then return** STATE[τρέχων]
τρέχων ← γειτονικός

Εικόνα 7 Ο αλγόριθμος αναρρίχησης λόφων

Υπάρχουν και πολλές άλλες παραλλαγές της αναρρίχησης λόφων. Για παράδειγμα υπάρχει η στοχαστική[1] αναρρίχηση λόφων, στην οποία επιλέγεται τυχαία μια κίνηση προς τα επάνω από τις διαθέσιμες. Ακόμη υπάρχει η αναρρίχηση λόφων με την πρώτη επιλογή, όπου βασικά είναι σαν την στοχαστική αλλά παράγει τυχαία διαδοχικές καταστάσεις μέχρι να βρεθεί κάποια καλύτερη από την τρέχουσα [2].

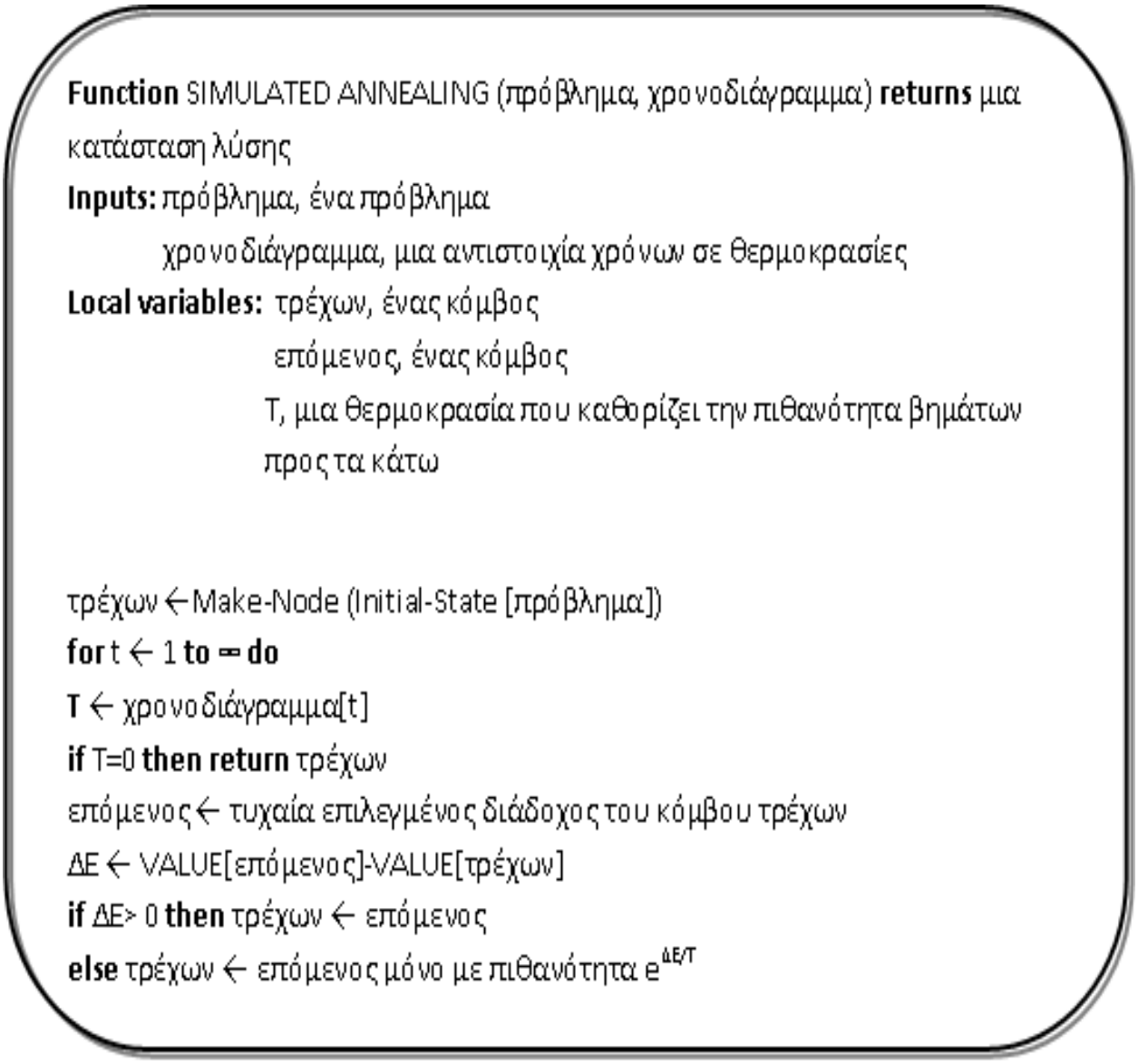
Αναζήτηση με προσομοιωμένη ανόπτηση (Simulated annealing)

Η προσομοιωμένη ανόπτηση είναι ένας συνδυασμός της αναρρίχησης λόφων και του τυχαίου περιπάτου. Παρακάτω εξετάζεται η γενική μορφή του αλγορίθμου. Ο τυχαίος περίπατος όπως αναφέρθηκε σε προηγούμενο κεφάλαιο είναι ένας τρόπος απόδρασης από τα τοπικά μέγιστα και στην προκειμένη περίπτωση, τοπικά ελάχιστα, αφού τώρα δίνεται βάρος στην ελαχιστοποίηση του κόστους. Οι Russell και Norving, φαντάζονται την αναζήτηση αυτή ως μια κατάσταση όπου ένα μπαλάκι του πινγκ πονγκ πρέπει να μπει στην βαθύτερη ρωγμή μιας ανώμαλης επιφάνειας. Κάποιος μπορεί να ταράξει την επιφάνεια ώστε το μπαλάκι να βγει από μια ρωγμή, δηλαδή από ένα τοπικό ελάχιστο, αλλά δεν πρέπει να την τραντάξει τόσο πολύ ώστε να ξεφύγει και από το ολικό ελάχιστο, δηλαδή την λύση [2].

Σαν αλγόριθμος μοιάζει με τον hill climbing αλλά αντί να διαλέγει την καλύτερη κατάσταση, διαλέγει τυχαία μια κίνηση. Αν δεν βελτιώνει την κατάσταση, η κίνηση αυτή, γίνεται δεκτή με πιθανότητα μικρότερη του 1, ανάλογα με το πόσο κακή ήταν. Χρησιμοποιείται ο συμβολισμός «θερμοκρασία[2]» και ο όρος χρονοδιάγραμμα. Αν το χρονοδιάγραμμα χαμηλώνει την θερμοκρασία αργά, κάποια στιγμή θα βρεθεί ένα καθολικό βέλτιστο, με πιθανότητα που φτάνει το ένα. Η προσομοιωμένη ανόπτηση, δεν «κατεβαίνει» ποτέ σε χειρότερες καταστάσεις [2].

[1] Ο όρος στοχαστική αναφέρεται στο γεγονός ότι χρησιμοποιεί μεθόδους βασισμένες στην τυχαιότητα διαφόρων στοιχείων πχ τυχαία βήματα, τυχαία αρχική κατάσταση.

[2] Χρησιμοποιείται ο όρος θερμοκρασία, αφού ανόπτηση είναι η διαδικασία με την οποία ένα μέταλλο ή γυαλί ζεσταίνει πάρα πολύ και ύστερα ψύχεται σιγά σιγά ώστε να σκληρύνει.



Εικόνα 8 Ο αλγόριθμος προσομοιωμένη ανόπτηση

Τοπική ακτινωτή αναζήτηση (Local beam search)

Η τοπική ακτινωτή αναζήτηση είναι αρκετά διαφορετική από τις δύο προηγούμενες αφού κρατάει παραπάνω από μια καταστάσεις στην μνήμη και παρέχει έτσι την δυνατότητα οπισθοδρόμησης. Στα προβλήματα ικανοποίησης περιορισμών ξεκινά παράγοντας τυχαία k καταστάσεις και τους διαδόχους τους και υπολογίζει κάθε φορά το πλήθος των μη ικανοποιημένων περιορισμών που προκύπτουν. Αν κάποια από αυτές τις καταστάσεις είναι ο στόχος σταματά εκεί. Αλλιώς επιλέγει τους k καλύτερους διαδόχους και παράγει τους δικούς τους διαδόχους κοκ. Οι γείτονες κόμβοι σε κάθε επίπεδο και πριν την παραγωγή νέων διαδόχων ανταλλάσσουν πληροφορίες μεταξύ

τους, ώστε να μεταφέρονται οι πόροι της αναζήτησης εκεί που γίνεται μεγαλύτερη πρόοδος. Αυτό όμως σημαίνει ότι η αναζήτηση μπορεί να συγκεντρωθεί σε μια μικρή μόνο περιοχή του χώρου καταστάσεων, στην οποία μπορεί τελικά να μην βρίσκεται ο στόχος [2].

Γενετικοί αλγόριθμοι (Genetic algorithms)

Οι γενετικοί αλγόριθμοι μοιάζουν με την ακτινική αναζήτηση στο γεγονός ότι παρακολουθούν παραπάνω από μια καταστάσεις. Οι καταστάσεις αυτές προκύπτουν από τον συνδυασμό δύο γονικών καταστάσεων. Στην αρχή παράγεται ο λεγόμενος πληθυσμός, k δηλαδή τυχαίες καταστάσεις. Κάθε κατάσταση ονομάζεται άτομο και αναπαρίσταται με μια συμβολοσειρά από ένα αλφάβητο. Υπάρχει μια συνάρτηση καταλληλότητας η οποία αξιολογεί όλες τις καταστάσεις και επιστρέφει τις υψηλότερες τιμές για τις καλύτερες καταστάσεις. Μετά γίνεται η αναπαραγωγή της επόμενης γενιάς καταστάσεων, μετά από τυχαία επιλογή γονέων. Από δύο γονείς μπορεί να προκύψει ένας απόγονος ή και παραπάνω, ανάλογα με τις απαιτήσεις του προβλήματος. Επιλέγεται τυχαία ένα σημείο διασταύρωσης μέσα στις θέσεις της συμβολοσειράς. Από πολύ διαφορετικούς γονείς, προκύπτουν πολύ διαφορετικοί απόγονοι. Τέλος, γίνεται μετάλλαξη τυχαίων στοιχείων των συμβολοσειρών σύμφωνα με μια μικρή ανεξάρτητη πιθανότητα. Το πλεονέκτημα αυτών των αλγορίθμων βρίσκεται στην διαδικασία της διασταύρωσης, αφού σε κάθε νέα γενιά κρατιούνται τα καλύτερα στοιχεία των προηγούμενων [2].

```
Function GENETIC ALGORITHM (πληθυσμός, FITNESS FN) returns ένα άτομο
Inputs: πληθυσμός, ένα σύνολο ατόμων
        FITNESS FN, συνάρτηση που μετρά την καταλληλότητα ενός ατόμου
Local variables: τρέχων, ένας κόμβος
                 επόμενος, ένας κόμβος
                 T, μια θερμοκρασία που καθορίζει την πιθανότητα βημάτων
                 προς τα κάτω

repeat
  νέος_πληθυσμός ← κενό σύνολο
  loop for i from 1 to SIZE[πληθυσμός] do
    x ← RANDOM-SELECTION (πληθυσμός, FITNESS FN)
    y ← RANDOM-SELECTION (πληθυσμός, FITNESS FN)
    παιδί ← REPRODUCE(x, y)
    if (μικρή τυχαία πιθανότητα) then παιδί ← MUTATE(παιδί)
    πρόσθεσε παιδί σε νέος_πληθυσμός
  πληθυσμός ← νέος_πληθυσμός
until κάποιο άτομο είναι αρκετά κατάλληλο ή έχει περάσει αρκετός χρόνος
return το καλύτερο άτομο από τον πληθυσμό, σύμφωνα με την FITNESS-FN


function REPRODUCE(x,y) returns ένα άτομο
  inputs: x, y, γονικά άτομα

  n ← LENGTH(x)
  c ← τυχαίος αριθμός από 1 μέχρι n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))
```

Εικόνα 9 Ο γενετικός αλγόριθμος

Κεφάλαιο 3

Οι εξεταζόμενοι αλγόριθμοι

Αφού έχουν εξηγηθεί πλέον όλες οι απαραίτητες έννοιες σχετικά με το πώς χειρίζεται η τεχνητή νοημοσύνη τα προβλήματα και πως μπορούν αυτά να αντιμετωπιστούν με αλγορίθμους, θα εξηγηθούν οι δύο αλγόριθμοι που θα συγκριθούν μεταξύ τους σε προβλήματα ικανοποίησης περιορισμών. Και οι δύο αναπτύχθηκαν σε γλώσσα προγραμματισμού Scala[3] η οποία επιλέχθηκε λόγο της απλότητας χειρισμού δομών δεδομένων. Στα δύο υποκεφάλαια που ακολουθούν δίνεται ψευδοκώδικας των αλγορίθμων που χρησιμοποιήθηκαν για τα πειράματα και αναλύονται οι βασικές συναρτήσεις και δομές δεδομένων. Ακόμη δίνονται μερικές πληροφορίες για την δομή των εξεταζόμενων προβλημάτων ώστε να γίνουν καλύτερα κατανοητοί οι αλγόριθμοι.

Τα προβλήματα που εξετάζονται αποτελούνται από τρία αρχεία. Στο αρχείο με κατάληξη _con, σε κάθε σειρά βρίσκεται και ένας περιορισμός με την μορφή X Y <>, όπου X είναι η πρώτη μεταβλητή και Y η δεύτερη μεταβλητή που εμπλέκονται στον περιορισμό. Το σύμβολο <> δείχνει τον τύπο του περιορισμού. Σε όλα τα προβλήματα χρησιμοποιείται ο ίδιος τύπος και συμβολίζει ότι η πρώτη και δεύτερη μεταβλητή δεν πρέπει να έχουν την ίδια τιμή. Επίσης όλοι οι περιορισμοί είναι δυαδικοί, αποτελούνται δηλαδή από δύο στοιχεία (μεταβλητές). Στο δεύτερο αρχείο, με κατάληξη _dom, βρίσκονται οι πληροφορίες σχετικά με το πεδίο ορισμού, από εκεί δηλαδή που παίρνουν τιμές οι μεταβλητές. Σε όλα τα προβλήματα υπάρχει ένα μόνο πεδίο ορισμού της μορφής 0, 1, 2, 3, 4. Δηλαδή αποτελούνται από ακέραιες τιμές που ξεκινάν από το 0 και αυξάνονται με βήμα 1. Τέλος στο τρίτο αρχείο με κατάληξη _var, είναι καταγεγραμμένες όλες οι μεταβλητές και το πεδίο ορισμού από το οποίο παίρνουν τιμές.

Στην παρούσα εργασία εξετάζεται το πόσο κοντά στην λύση μπορεί να φτάσει ο κάθε αλγόριθμος, οπότε το τελικό αποτέλεσμα που επιστρέφουν οι αλγόριθμοι είναι οι συγκρούσεις που απομένουν και όχι οι τιμές των μεταβλητών που προκαλούν την λύση, αν αυτή βρεθεί. Σύγκρουση λέγεται η περίπτωση στην οποία δεν ικανοποιείται ένας

[3] Η scala αναπτύχθηκε από τον Martin Odesky και η πρώτη έκδοσή της κυκλοφόρησε το 2003. Συνδυάζει χαρακτηριστικά του αντικειμενοστραφούς αλλά και του συναρτησιακού προγραμματισμού [22].

περιορισμός. Αν για παράδειγμα ένας περιορισμός που αποτελείται από δύο μεταβλητές, επέβαλε στην πρώτη μεταβλητή που τον απαρτίζει να είναι ίση με την δεύτερη και αυτή δεν ήταν, θα υπήρχε σύγκρουση. Στα συγκεκριμένα προβλήματα υπάρχει σύγκρουση όταν η πρώτη και η δεύτερη μεταβλητή που αποτελούν τον περιορισμό, έχουν την ίδια τιμή, ενώ ο περιορισμός της μορφής <> επιβάλει να έχουν διαφορετικές.

Σε κάθε επαναληπτικό βήμα των αλγορίθμων εξετάζεται εάν η τρέχουσα κατάσταση, το σύνολο δηλαδή των τιμών που αντιστοιχούν σε κάθε μεταβλητή, είναι η κατάσταση στόχου. Η αποτίμηση των καταστάσεων γίνεται σύμφωνα με το πλήθος των συγκρούσεων. Αν δηλαδή στην εξεταζόμενη κατάσταση συμβαίνει έστω και μια σύγκρουση τότε το πρόβλημα δεν έχει λυθεί, ενώ αν ο αριθμός των συγκρούσεων είναι μηδενικός, τότε αυτή είναι η κατάσταση στόχου. Αυτό σημαίνει ότι το πρόβλημα λύθηκε.

3.1 Ο αλγόριθμος Min Conflicts

Ο Min Conflicts όπως αναφέρθηκε και παραπάνω είναι ένας Hill climbing αλγόριθμος για προβλήματα ικανοποίησης περιορισμών. Αρχικά αναθέτει τιμές σε όλες τις μεταβλητές, διαλέγοντας μία από το πεδίο τιμών που τους αντιστοιχεί και μετά επιλέγει τυχαία μια μεταβλητή, από το σύνολο των μεταβλητών που συμμετέχουν σε κάποια σύγκρουση για να την εξετάσει. Στην συνέχεια δοκιμάζει όλες τις τιμές από το πεδίο ορισμού εκτός από την τρέχουσα και κρατάει τελικά αυτήν που ελαχιστοποιεί τον αριθμό των συγκρούσεων. Αν υπάρχουν περισσότερες από μια τιμές που το κάνουν αυτό, επιλέγεται μια τυχαία. Η διαδικασία συνεχίζει καθ’ αυτόν τον τρόπο έως ότου βρεθεί λύση ή έχει εκτελεστεί ένας προκαθορισμένος αριθμός επαναλήψεων.

Το μεγαλύτερο πρόβλημα του Min Conflicts είναι το ότι μπορεί εύκολα να παγιδευτεί σε τοπικό ελάχιστο, ειδικά αν οι λύσεις είναι αραιά κατανεμημένες στον χώρο αναζήτησης. Δηλαδή η κάθε νέα τιμή που θα δοκιμάζει στην εκάστοτε μεταβλητή προς εξέταση, θα δίνει είτε μεγαλύτερο είτε ίσο αριθμό συγκρούσεων συγκριτικά με την αρχική ανάθεση, οπότε δεν θα μπορεί να γίνει επιλογή νέας τιμής που επιφέρει μικρότερο αριθμό συγκρούσεων. Σε αυτή την περίπτωση δεν αλλάζει η τιμή που αρχικά είχε δοθεί στην μεταβλητή. Για να ξεπεραστεί αυτό το πρόβλημα ο Min Conflicts σε

κάθε επανάληψη εξετάζει μια άλλη τυχαία μεταβλητή ώστε να ξεφεύγει από το αδιέξοδο. Μέχρι να κολλήσει σε τοπικό ελάχιστο, σε κάθε επανάληψη μειώνει τον αριθμό των συγκρούσεων και δεν «πέφτει» ποτέ σε χειρότερη κατάσταση. Συνήθως προκαθορίζονται και συγκεκριμένες επαναλήψεις ή χρόνος εκτέλεσης ώστε αν δεν υπάρχει λύση στο εξεταζόμενο πρόγραμμα να μην εκτελείτε επ'άπειρον ο αλγόριθμος [23].

Algorithm MIN CONFLICTS
input: πρόβλημα ικανοποίησης περιορισμών
περιορισμοί, *πίνακας με τα ζεύγη των περιορισμών*
βήματα, *ο αριθμός των βημάτων που επιτρέπονται*
output: συγκρούσεις, *ο αριθμός των συγκρούσεων που έμειναν*

τρέχουσα_κατάσταση, *αρχική τυχαία ανάθεση τιμών στις μεταβλητές*
for i=1 to βήματα **do**
τρέχουσα_μεταβλητή ←επιλέγεται τυχαία από τις μεταβλητές που συμμετέχουν σε σύγκρουση
καλύτερη_τιμή ← δοκιμάζονται όλες οι τιμές από το πεδίο ορισμού (εκτός από την τρέχουσα) και κρατάει αυτήν που ελαχιστοποιεί τις συγκρούσεις, αλλάζει αν υπάρχει καλύτερη από την αρχική
συγκρούσεις ← συγκρούσεις που προκύπτουν με την απόδοση της καλύτερης τιμής στην τρέχουσα μεταβλητή
set τρέχουσα_μεταβλητή (τιμή) ←καλύτερη τιμή
return συγκρούσεις

Παρακάτω παρουσιάζονται οι βασικές συναρτήσεις και δομές δεδομένων του αλγορίθμου Min Conflicts που υλοποιήθηκε. Η σειρά που παραθέτονται είναι και αυτή με την οποία καλούνται από τον κώδικα.

Η συνάρτηση getTxtString

```
def getTxtString(path: String) = {
    val fileSource = io.Source.fromFile(path)
    val txtString = fileSource.mkString
    fileSource.close()
    txtString
  }
```

Αυτή είναι η πρώτη συνάρτηση που καλείται από τον βασικό κώδικα και διαβάζει το αρχείο με τους περιορισμούς, που βρίσκεται στον υπολογιστή εκεί που δείχνει η παράμετρος path. Μετατρέπει τα στοιχεία του αρχείου σε string για να μπορέσει να τα διαβάσει και επεξεργαστεί.

Η συνάρτηση getConstraints

```
  def getConstraints(txt: String) = {
    for (line <- txt.split("\n"); nodes = line.split(" ");
if nodes.length > 1)
      yield Array(nodes(0).toInt,nodes(1).toInt)
  }
```

Εδώ χωρίζεται το αρχείο των περιορισμών τύπου string που δημιούργησε η getTxtString. Δημιουργεί δύο στήλες την nodes(0) και την nodes(1), όπου στην πρώτη στήλη καταχωρείται η πρώτη μεταβλητή που εμπλέκεται στον περιορισμό και στην δεύτερη ο δεύτερος. Όπου υπάρχει νέα γραμμή στο αρχείο, ξεκινάει ένα νέο ζευγάρι περιορισμού και όπου υπάρχει κενό, χωρίζονται οι μεταβλητές οι οποίες τον αποτελούν. Αυτή η συνάρτηση ανατίθεται στην μεταβλητή constraints, οπότε όταν συναντάται παρακάτω, αναφέρεται ως πίνακας constraints.

Η συνάρτηση getMaxValue.

```
  def getMaxValue(conPath: String) = {
    val domPath = conPath.substring(0,conPath.length - 7) +
"dom.txt"
```

```
    val fileSource = io.Source.fromFile(domPath)
    val txtString = fileSource.mkString
    fileSource.close()
    txtString.split("\\s+").last.toInt
  }
```

Η συνάρτηση αυτή ψάχνει στο αρχείο _dom όπου βρίσκονται οι τιμές του πεδίου ορισμού και βρίσκει τη μεγαλύτερη τιμή που μπορεί αυτό να πάρει. Αυτή η τιμή χρησιμοποιείται ώστε να καθοριστεί το μέγεθος του πεδίου ορισμού και οι τιμές του. Αν για παράδειγμα η τιμή που επιστρέφει αυτή η συνάρτηση είναι το 6, τότε το πεδίο ορισμού είναι το 0, 1, 2, 3, 4, 5, 6.

Η συνάρτηση getRandomValues

```
  def getRandomValues(maxValue:Int, length: Int) = {
    (for (i <- 0 until length) yield
Random.nextInt(maxValue + 1)).toArray
  }
```

Η getRandomValues χρησιμοποιείται για να δοθούν τυχαίες τιμές από το πεδίο ορισμού (δηλαδή από 0 ως maxValue με βήμα 1), σε όλες τις μεταβλητές. Παίρνει ως παράμετρο την maxValue, η οποία είναι η τιμή που επέστρεψε η συνάρτηση getMaxValue και το length που ορίζει το μήκος του πίνακα στον οποίο θα δοθούν αυτές οι τυχαίες τιμές. Όταν καλείται αυτή η συνάρτηση το length που της δίνεται είναι το lastNode+1. Αυτό σημαίνει ότι τελικά δημιουργείται πίνακας μεγέθους όσο ο μεγαλύτερη μεταβλητή +1. Αν δηλαδή η μεγαλύτερη μεταβλητή είναι το 59 και η maxValue το 3, δημιουργείται πίνακας 60 θέσεων και η κάθε θέση έχει μια τιμή από το 0 ως το 3. (Το +1 χρησιμοποιείται επειδή το μέτρημα των τιμών ξεκινά από την τιμή 0)

Η συνάρτηση getConflicts

```
def getConflicts(constraints: Array[Array[Int]], values:
Array[Int])

=constraints.filter(c => values(c(0)) == values(c(1)))
```

Αυτή η συνάρτηση δημιουργεί έναν υποπίνακα του πίνακα constraints, ο οποίος περιέχει τους περιορισμούς. Ο νέος πίνακας που επιστρέφει η συνάρτηση, περιέχει τις γραμμές του constraints στις οποίες υπάρχει σύγκρουση, δηλαδή ελέγχει, με την εντολή filter, στον πίνακα values, με οδηγό τον πίνακα constraints, αν το στοιχείο constraint(0) είναι ίσο με το constraint(1). Με πιο απλά λόγια εξετάζει τον πίνακα constraints με τους περιορισμούς και κοιτάει και στον πίνακα values τι τιμή έχει η κάθε μεταβλητή. Αν οι δύο μεταβλητές του περιορισμού έχουν ίδια τιμή, τότε αντιγράφει αυτή τη γραμμή του constraints στον νέο πίνακα που επιστρέφει.

Η συνάρτηση getConflictsForNewValue

```
  def getConflictsForNewValue(constraints:
Array[Array[Int]], values: Array[Int], index: Int,
newValue: Int) = {
    val newValues = values.clone
    newValues(index) = newValue
    getConflicts(constraints, newValues)
  }
```

Η βασική λειτουργία αυτής της συνάρτησης είναι η εύρεση του αριθμού των συγκρούσεων για κάθε νέα τιμή από το πεδίο ορισμού για την τρέχουσα μεταβλητή. Καλείται εξωτερικά τόσες φορές όσες οι τιμές του πεδίου ορισμού και κάθε φορά η νέα τιμή της επανάληψης περνιέται ως τέταρτη παράμετρος. Αν δηλαδή το πεδίο ορισμού είναι το 0, 1, 2, 3 τότε θα κληθεί 3 φορές, θα εξετάσει το πόσες συγκρούσεις προκαλεί κάθε μια από τις τιμές του πεδίου ορισμού εκτός από την τιμή της αρχικής ανάθεσης, με την οποία και θα συγκρίνει τις υπόλοιπες. Αφού θα εξαιρέσει την τρέχουσα τιμή της μεταβλητής, υπάρχει η εντολή `val newValues = values.clone` ώστε να μην αλλάζει η τρέχουσα τιμή του currentNode. Εξωτερικά πάλι αυτής της συνάρτησης αποθηκεύεται η τιμή που προκαλεί τις λιγότερες συγκρούσεις και πόσες ήταν αυτές.

Η ανάθεση currentNode

```
val currentNode =

nodesInConflict(Random.nextInt(nodesInConflict.length))
```

Πριν ξεκινήσει η διαδικασία των επαναληπτικών βημάτων του αλγορίθμου πρέπει να επιλεχθεί τυχαία μια μεταβλητή προς εξέταση μέσα από τις συγκρούσεις. Σε μια προηγούμενη εκδοχή του προγράμματος, η επιλογή αυτή υλοποιούνταν με την χρήση μια συνάρτησης όμως αργότερα αντικαταστάθηκε για λόγους απλότητας από μια ανάθεση. Έτσι το αντικείμενο currentNode είναι πλέον υπεύθυνο για τον έλεγχο του πίνακα που περιέχει τις συγκρούσεις (δηλαδή του nodesInConflict) και την επιλογή μια τυχαίας μεταβλητής του. Ο πίνακας nodesInConflicts έχει προηγουμένως καθαριστεί από τα διπλά στοιχεία ώστε όλες οι μεταβλητές να έχουν την ίδια πιθανότητα τυχαίας επιλογής. Έτσι επιλέγεται τυχαία μια μεταβλητή προς εξέταση από αυτούς που συμμετέχουν σε παραβίαση περιορισμού, δηλαδή σε σύγκρουση.

Για να γίνουν καλύτερα κατανοητά όλα τα παραπάνω καθώς και η σειρά με την οποία καλούνται τελικά οι συναρτήσεις δίνεται το παρακάτω κομμάτι του προγράμματος. Τα κόκκινα γράμματα δίπλα στις γραμμές του κώδικα περιγράφουν την λειτουργία κάθε εντολής.

```
object MinConflicts {

  def algorithm(repetitions: Int, filePath: String) = {
    val startingTime = System.currentTimeMillis κλήση της συνάρτησης που μετράει τον χρόνο εκτέλεσης του αλγορίθμου
    val txt = getTxtString(filePath)  κλήση της getTxtString που χωρίζει το αρχείο με τους περιορισμούς
    val constraints = getConstraints(txt) κλήση της getConstraints που δημιουργεί τον δυσδιάστατο πίνακα με τα ζευγάρια των περιορισμών
    val lastNode = constraints.flatten.max εύρεση πλήθους μεταβλητών
    val maxValue = getMaxValue(filePath) εύρεση μεγαλύτερης τιμής του πεδίου ορισμού
    val values = getRandomValues(maxValue, lastNode + 1) απόδοση τυχαίων τιμών (από 0 εώς maxValue με βήμα 1) σε όλες τις μεταβλητές
    var minNumOfConflicts = constraints.length αρχική ανάθεση ως μικρότερο αριθμό συγκρούσεων, το μήκος του πίνακα constraints με τους περιορισμούς

    for (step <- 0 to repetitions) {από εδώ ξεκινά ο κόμβος των επαναλήψεων, τα βήματα (repetitions, καθορίζονται στην συνάρτηση main από τον χρήστη
      val conflicts = getConflicts(constraints, values)κλήση της getConflicts που επιστρέφει πίνακα συγκρούσεων, σύμφωνα με τις τρέχουσες τυχαίες τιμές
      val nodesInConflict = conflicts.flatten.distinct ο πίνακας με τις συγκρούσεις καθαρίζεται από τα διπλά στοιχεία
      val currentNode = nodesInConflict(Random.nextInt(nodesInConflict.length)) τυχαία επιλογή από τον πίνακα με τις μεταβλητές σε σύγκρουση
      val valueWithConflictsMap = (for(i <- 0 to maxValue) δοκιμάζονται όλες οι τιμές του πεδίου ορισμού εκτός από την τρέχουσα και αποθηκεύεται αυτήν
        yield getConflictsForNewValue(constraints, values, currentNode, i).length -> i).toMap  που προκαλεί τις λιγότερες συγκρούσεις
      val minConflictsValue = valueWithConflictsMap.min._2
      values(currentNode) = minConflictsValue η τρέχουσα μεταβλητή που εξετάζεται παίρνει την τιμή που προκαλεσε τις λιγότερες συγκρούσεις
      minNumOfConflicts = conflicts.length ως μικρότερος αριθμός συγκρούσεων τίθεται η τελευταία τιμή του πίνακα conflicts με τις συγκρούσεις

    }
```

Εικόνα 10 Κωδικας σε Scala για τον MinConflicts

Όπως φαίνεται στην παραπάνω εικόνα, οι περισσότερες συναρτήσεις του προγράμματος επιστρέφουν κάποιον πίνακα ο οποίος ανατίθεται σε κάποια μεταβλητή στο κύριο κομμάτι του κώδικα, γι αυτό και δεν παρουσιάζονται ξεχωριστά οι πίνακες και οι άλλες δομές που χρησιμοποιήθηκαν. Για παράδειγμα με την εντολή val values = getRandomValues(maxValue, lastNode + 1), ανατίθεται στην values ο πίνακας που παράγει η getRandomNodes. Έτσι όταν χρησιμοποιείται η values είναι σαν να καλείται η getRandomNodes. Ακόμη η values μπορεί να περαστεί και ως παράμετρος σε κάποια άλλη συνάρτηση πχ val conflicts = getConflicts(constraints, values), οπότε είναι σαν η getConflicts να παίρνει ως παράμετρο τον πίνακα με τους περιορισμούς και τον πίνακα με τις αντίστοιχες τιμές των μεταβλητών των περιορισμών.

Υπάρχει όμως και μια δομή δεδομένων στην Scala, η λεγόμενη Map, η οποία αποτελείται βασικά από ζεύγη κλειδιών και των αντίστοιχων τιμών τους. Για παράδειγμα η συλλογή Map{(“x”, 5), (“y”, 6), (“k”, 7) } δηλώνει ότι το κλειδί x αντιστοιχίζεται με την τιμή 5, το y με την τιμή 6 κοκ. Παρακάτω δίνεται το κομμάτι κώδικα που χρησιμοποιείται στην εργασία.

```
val valueWithConflictsMap = (for(i <- 0 to maxValue)
  yield getConflictsForNewValue(constraints, values,
currentNode,i).length -> i).toMap
val minConflictsValue = valueWithConflictsMap.min._2
```

Εδώ καλείται η getConflictsForNewValue στην αρχή του προγράμματος, τόσες φορές, όσες οι τιμές στο πεδίο ορισμού. Το maxValue περνίεται και σαν παράμετρος στην συνάρτηση ώστε σε κάθε επανάληψη να δοκιμάζεται μια νέα τιμή εκτός της αρχικής και υπολογίζονται οι συγκρούσεις που προκαλούνται. Αν για παράδειγμα το πεδίο ορισμού είναι 0, 1, 2, θα γίνουν δύο επαναλήψεις και για την κάθε νέα τιμή θα υπολογιστούν και θα αποθηκευτούν οι τιμές στον Map. Δηλαδή αν η αρχική τιμή ήταν η 0 και είχαν προκληθεί 30 συγκρούσεις και για τις υπόλοιπες τιμές του πεδίου ορισμού 1 και 2 προκληθηκαν 54 και 43 αντίστοιχα τότε ο Map διαμορφώνεται ως εξής: (30, 0), (54, 1) και (43, 2) όπου το κλειδί είναι οι συγκρούσεις και οι τιμές των κλειδιών είναι οι τιμές που τις προκαλούν. Στην τελευταία εντολή δίνεται στην μεταβλητή minConflictsValue η τιμή που αντιστοιχεί στο μικρότερο κλειδί του map, αναθέτοντας της έτσι την τιμή που προκάλεσε τις λιγότερες συγκρούσεις.

3.2 Ο αλγόριθμος Break Out

Ο break out είναι όπως και ο min conflicts ένας αλγόριθμος hill climbing, αυτό όμως που τον διαφοροποιεί είναι η ικανότητά του να ξεφεύγει από τα τοπικά ελάχιστα χρησιμοποιώντας έναν μηχανισμό επιπρόσθετων βαρών. Θεωρητικά ο break out δεν είναι πλήρης, όμως στην πράξη βρίσκει σχεδόν πάντα λύση στα επιλύσιμα προβλήματα. Σε εργασία του Paul Morris, έχει αποδειχθεί ότι υπάρχει και μια εκδοχή του που είναι πλήρης αλλά δεν είναι αποδοτική. Έτσι λοιπόν ο break out χρησιμοποιεί Min Conflicts heuristic, δηλαδή η αποτίμηση των καταστάσεων γίνεται σύμφωνα με τον αριθμό των συγκρούσεων και χρησιμοποιεί και την τεχνική της δυναμικής απόδοσης βαρών ως τρόπο απόδρασης από τοπικό ελάχιστο. Με την ανάθεση και αύξηση των βαρών στους περιορισμούς, η τοπογραφία της επιφάνειας κόστους μεταβάλλεται και έτσι αν η τρέχουσα κατάσταση ήταν τοπικό ελάχιστο, αυτό μπορεί να αλλάξει. Ο break out έχει δηλαδή την δυνατότητα να αλλάζει την συνάρτηση κόστους έτσι ώστε η αξιολογούμενη τιμή της εξεταζόμενης κατάστασης να γίνει μεγαλύτερη από αυτή των γειτονικών της καταστάσεων. Αν δεν βρεθεί σε τοπικό ελάχιστο, κάνει αλλαγές στις μεταβλητές των τιμών ώστε να μειωθεί το συνολικό κόστος και αναθέτει στην τρέχουσα μεταβλητή την τιμή που προκαλεί το μικρότερο άθροισμα βαρών [14]. Πλέον δηλαδή δεν εξετάζεται ο αριθμός των συγκρούσεων όπως στον Min Conflicts αλλά το άθροισμα των βαρών το οποίο προκύπτει από αυτές.

Παρακάτω περιγράφονται οι καταστάσεις στις οποίες μπορεί να βρεθεί ο αλγόριθμος και ο τρόπος αντιμετώπισης τοπικών ελαχίστων με την απόδοση βαρών στους περιορισμούς:

- Ξεκινά παράγοντας μια ‘‘ελαττωματική’’ λύση, δίνοντας τυχαίες τιμές από το πεδίο ορισμού σε όλες τις μεταβλητές. Σε κάθε βήμα της επανάληψης, γίνονται αλλαγές στις τιμές των μεταβλητών από το πεδίο ορισμού με σκοπό την μείωση του κόστους (λιγότερες συγκρούσεις), ώσπου να βρεθεί τελικά μια λύση.
- Μπορεί όμως η διαδικασία να παγιδευτεί σε τοπικό ελάχιστο, όπου καμία αλλαγή στην τιμή της μεταβλητής δεν θα μπορεί να επιφέρει μικρότερο αριθμό συγκρούσεων, δηλαδή να μειώσει το κόστος, ενώ υπάρχει έστω ακόμη μια σύγκρουση.

- Ένα "βάρος" συνδέεται με κάθε περιορισμό. Για μια κατάσταση (δηλαδή μια πλήρη ανάθεση τιμών στις μεταβλητές), υπολογίζεται ένα κόστος το οποίο παράγεται από το άθροισμα των βαρών των περιορισμών που δεν ικανοποιούνται. Στον αλγόριθμο υπάρχει ένας δυσδιάστατος πίνακας ο οποίος στην πρώτη στήλη κρατάει την πρώτη μεταβλητή η οποία απαρτίζει τον περιορισμό και στην δεύτερη στήλη, την δεύτερη μεταβλητή, όπως περιγράφηκε και στον Min Conflicts. Τώρα στον Break Out, εισάγεται στον ίδιο πίνακα και μια τρίτη στήλη όπου αποθηκεύονται τα αντίστοιχα βάρη των περιορισμών. Στο πρώτο βήμα της επαναληπτικής διαδικασίας, κάθε γραμμή αυτής της στήλης παίρνει τον αριθμό 1.
- Η διαδικασία της τοπικής αναζήτησης συνεχίζεται μέχρι να βρεθεί σε τοπικό ελάχιστο, αλλά πλέον δεν ενδιαφέρει ο αριθμός των συγκρούσεων αλλά το βάρος του κάθε περιορισμού.
- Στο τοπικό ελάχιστο, τα βάρη των περιορισμών που δεν ικανοποιούνται στην τρέχουσα κατάσταση, αυξάνονται κατά ένα, δηλαδή ο αριθμός που υπάρχει στην τρίτη στήλη και στην αντίστοιχη γραμμή, αυξάνεται κατά ένα, ώστε το συνολικό κόστος της τρέχουσας κατάστασης να αυξηθεί. Έτσι την επόμενη φορά που συναντάται αυτός ο περιορισμός κατά την διαδικασία της αναζήτησης, ο αλγόριθμος θα γνωρίζει ότι είναι προβληματικός, αφού έχει μεγάλο βάρος και άρα είναι πολύ δύσκολο να ικανοποιηθεί.

Παρατίθεται ο ψευδοκώδικας του αλγόριθμου break out, που χρησιμοποιήθηκε για την πειραματική μελέτη των προβλημάτων στην παρούσα εργασία.

Algorithm BREAK OUT

input: πρόβλημα ικανοποίησης περιορισμών
περιορισμοί, *πίνακας με τα ζεύγη των περιορισμών*
βήματα, *ο αριθμός των βημάτων που επιτρέπονται*
τρέχουσα_κατάσταση, *αρχική τυχαία ανάθεση τιμών στις μεταβλητές*
τρέχουσα_μεταβλητή, η μεταβλητή προς εξέταση

output: συγκρούσεις, *ο αριθμός των συγκρούσεων που έμειναν*

for i=1 το βήματα **do**

άθροισμα_βαρών ←*άθροισμα των βαρών που προκαλεί η τρέχουσα τιμή στην τρέχουσα_μεταβλητή (την 1η φορά όλα 1)*

καλύτερο_άθροισμα_βαρών ← το μικρότερο άθροισμα βαρών που προκύπτει από την δοκιμή όλων των τιμών του πεδίου

καλύτερη_τιμή ← η τιμή που ελαχιστοποιεί το άθροισμα βαρών

set *τρέχουσα_μεταβλητή* (τιμή) = καλύτερη_τιμή

If καλύτερο_άθροισμα_βαρών **>** άθροισμα_βαρών (*έλεγχος για τοπικό ελάχιστο)* **then**

αύξησε τα βάρη στους περιορισμούς που δεν ικανοποιούνται και περιέχουν την τρέχουσα μεταβλητή, κατά 1

else set *τρέχουσα_μεταβλητή* (τιμή) ←καλύτερη τιμή

return συγκρούσεις

Παρακάτω παρουσιάζονται οι βασικές συναρτήσεις και δομές δεδομένων του αλγορίθμου Break Out που υλοποιήθηκε. Η σειρά που παραθέτονται είναι και αυτή με την οποία καλούνται στον τον κώδικα επίσης οι συναρτήσεις οι οποίες είναι ίδιες με του Min Conflicts, δεν ξαναπαρουσιάζονται.

Η συνάρτηση getTxtString

Η συνάρτηση getTxtString είναι ίδια με αυτήν του Min Conflicts και οπότε καλείται από εκεί μία φορά στην αρχή για να διαβαστεί το αρχείο του προβλήματος και να το μετατρέψει σε string.

Η συνάρτηση getConstraints

```
def getConstraints(txt: String) = {
    for (line <- txt.split("\n"); nodes = line.split(" ");
if nodes.length > 1)
      yield Array(nodes(0).toInt,nodes(1).toInt, 1)
  }
```

Η getConstraints είναι παρόμοια με αυτή που χρησιμοποιήθηκε στον Min Conflicts, δημιουργεί δηλαδή τις δύο στήλες από τα ζεύγη μεταβλητών για τους περιορισμούς χωρίζοντας το αρχείο προβλήματος. Προσθέτει όμως και μια τρίτη στήλη στον πίνακα constraints με τα βάρη των περιορισμών, που την γεμίζει με 1, αφού στο πρώτο βήμα της επανάληψης όλα τα βάρη είναι ίσα με ένα. Όπως και στον Min Conflicts η συνάρτηση αυτήν ανατίθεται στην μεταβλητή constraints, οπότε όπου χρησιμοποιείται παρακάτω, πρέπει να θυμάται κανείς ότι αναφέρεται στις δύο στήλες με τις μεταβλητές που αποτελούν τον περιορισμό και την μία στήλη με τα αντίστοιχα βάρη.

Η συνάρτηση getLastNode

```
def getLastNode(constraints: Array[Array[Int]]) = {
    val constraintsWithoutWeights = for (constraint <-
constraints) yield Array(constraint(0),constraint(1))
    constraintsWithoutWeights.flatten.max
  }
```

Εδώ εντοπίζεται η μεγαλύτερη μεταβλητή ψάχνοντας στον πίνακα constraints με τους περιορισμούς. Για να γίνει αυτό δημιουργεί έναν πίνακα παίρνοντας μόνο τις 2 πρώτες στήλες του πίνακα constraints (η τρίτη στήλη περιέχει τα βάρη, οπότε προς το παρόν αγνοείται) με όνομα constraintsWithoutWeights και βρίσκει το μεγαλύτερο στοιχείο του. Η μεγαλύτερη μεταβλητή είναι χρήσιμος αφού στην συνέχεια του προγράμματος πρέπει να δοθούν τυχαία τιμές σε όλες τις μεταβλητές και χρησιμοποιείται η επιστρεφόμενη τιμή της προκειμένης συνάρτησης ως το μήκος του νέου αυτού πίνακα.

Η συνάρτηση getMaxValue

Χρησιμοποιείται η ίδια συνάρτηση από τον Min Conflicts. Όπως και πριν η getMaxValue επιστρέφει την μεγαλύτερη τιμή του πεδίου ορισμού.

Η συνάρτηση getRandomValue

Για ακόμη μία φορά χρησιμοποιείται συνάρτηση από τον Min Conflicts ώστε να δοθούν τυχαίες τιμές από 0 έως maxValue σε όλες τις μεταβλητές του προβλήματος.

Η ανάθεση CurrentNode

```
currentNode =
nodesInConflict(Random.nextInt(nodesInConflict.length))
```

Όπως και στον Min Conflicts πριν ξεκινήσει η διαδικασία των επαναληπτικών βημάτων του αλγορίθμου πρέπει να επιλεχθεί τυχαία μια μεταβλητή προς εξέταση μέσα από τις συγκρούσεις. Το ίδιο συμβαίνει και στο τέλος του κόμβου των επαναλήψεων ώστε να επιλεχθεί μια νέα μεταβλητή προς εξέταση. Σε μια προηγούμενη εκδοχή του προγράμματος, η επιλογή αυτή υλοποιούνταν με την χρήση μια συνάρτησης που καλούνταν από τον MinConflicts, όμως αργότερα αντικαταστάθηκε για λόγους απλότητας από μια ανάθεση. Έτσι λοιπόν το αντικείμενο currentNode είναι πλέον υπεύθυνο για τον έλεγχο του πίνακα που περιέχει τις συγκρούσεις (δηλαδή του nodesInConflict) και την επιλογή μια τυχαίας μεταβλητής του. Ο πίνακας nodesInConflicts έχει προηγουμένως καθαριστεί από τα διπλά στοιχεία ώστε όλες οι μεταβλητές να έχουν την ίδια πιθανότητα τυχαίας επιλογής.

Η συνάρτηση getSumOfWeightsInConflicts

```
  def getSumOfWeightsInConflict(constraints:
Array[Array[Int]], values: Array[Int], index: Int,
newValue: Int) = {
    val newValues = values.clone
    newValues(index) = newValue
    val weights = for (constraint <- constraints; if
newValues(constraint(0)) == newValues(constraint(1))) yield
constraint(2)
    weights.sum
  }
```

Αυτό που κάνει αυτή η συνάρτηση είναι να υπολογίζει το άθροισμα των βαρών που προκύπτει για κάθε νέα τιμή που της δίνεται ως παράμετρος. Την πρώτη φορά που καλείται μέσα στον βρόγχο των επαναλήψεων επιστρέφει τα βάρη των περιορισμών, όπου εκείνη την στιγμή είναι όλα 1. Λίγο μετά, πάλι μέσα στον ίδιο βρόγχο ξανακαλείται ώστε να υπολογίσει τα βάρη, αν δοθούν και οι υπόλοιπες τιμές από το πεδίο ορισμού. Για κάθε νέα τιμή που της δίνεται, κοιτάει στον πίνακα με τους περιορισμούς πού έχει προκληθεί σύγκρουση, δηλαδή που η τιμή της μεταβλητής στην

πρώτη στήλη είναι ίση με την τιμή της μεταβλητής στην δεύτερη στήλη, στην ίδια γραμμή. Όπου συμβαίνει αυτό, παράγει μία τρίτη στήλη μόνο με τις γραμμές που έχουν σύγκρουση και την αποθηκεύει σε έναν νέο πίνακα με όνομα weights του οποίου βρίσκει το άθροισμα όλων των στοιχείων. Το άθροισμα αυτό, είναι αυτό των βαρών. Το επόμενο βήμα στον κώδικα μετά την κλίση αυτής της συνάρτησης είναι να κρατηθεί σε μία μεταβλητή το μικρότερο άθροισμα των βαρών και η τιμή η οποία το προκάλεσε. Ακριβώς μετά γίνεται ο έλεγχος για τοπικό ελάχιστο, ελέγχεται δηλαδή αν το άθροισμα των βαρών που προκύπτουν από τις υπόλοιπες τιμές (εκτός από την τρέχουσα) είναι μεγαλύτερο από αυτό που προέκυψε με την τρέχουσα τιμή.

Η συνάρτηση getUdatedConstraints

```
  def getUpdatedConstraints(constraints: Array[Array[Int]],
values: Array[Int], index: Int) = {
    for (constraint <- constraints; newWeight = if
(areNodesNeighborsWithSameValue(constraint,values,index))
constraint(2) + 1
    else constraint(2))
      yield Array(constraint(0),constraint(1),newWeight)
  }

}
```

Η πρώτη παράμετρος είναι ο πίνακας constraints, η δεύτερη οι τιμές του πεδίου ορισμού και η τρίτη η τρέχουσα μεταβλητή που εξετάζεται. Αυτή η συνάρτηση καλείται μόνο στην περίπτωση που επαληθευτεί το if που ελέγχει για τοπικό ελάχιστο. Πρέπει δηλαδή τώρα να αυξηθούν τα βάρη στους περιορισμούς που περιέχουν την τρέχουσα μεταβλητή και δεν έχουν ικανοποιηθεί, κατά ένα. Για να το διαπιστώσει αυτό, καλεί την Boolean συνάρτηση areNodesNeighborsWithSameValue. Αν αυτή επιστρέψει θετική απάντηση τότε αυξάνονται κατά ένα τα βάρη στους αντίστοιχους περιορισμούς. Αυτό που τελικά επιστρέφει είναι ένας πίνακας όπου στην πρώτη στήλη υπάρχει ο πρώτη μεταβλητή του περιορισμού, στην δεύτερη ο δεύτερη και στην τρίτη τα ανανεωμένα πλέον βάρη.

Η συνάρτηση areNodesNeighborsWithSameValue

```
  private def
areNodesNeighborsWithSameValue(constraint:Array[Int],
values: Array[Int], index: Int) = {
    (constraint(0) == index || constraint(1) == index) &&
values(constraint(0)) == values(constraint(1))
  }
```

Αυτή είναι μια Boolean συνάρτηση που σημαίνει ότι επιστρέφει «ναι» ή «όχι». Ελέγχει μέσα στον πίνακα με τους περιορισμούς αν η πρώτη μεταβλητή του ή η δεύτερη, είναι ο τρέχουσα μεταβλητή, καθώς και αν έχει ίδια τιμή με αυτήν που είναι ζευγάρι. Αν ισχύει αυτό επιστρέφει «ναι» και σημαίνει ότι ο περιορισμός που περιέχει την τρέχουσα μεταβλητή δεν έχει ικανοποιηθεί και άρα πρέπει να αυξηθεί το βάρος του. Αλλιώς επιστρέφει αρνητική απάντηση και η getUpdatedConstraints δεν αλλάζει τίποτα για τον συγκεκριμένο περιορισμό.

Χρησιμοποιείται και στον Break Out η δομή δεδομένων Map, αυτή τη φορά για να βρεθούν τα αθροίσματα των βαρών και οι τιμές οι οποίες τα προκαλούν.

```
val valueWithSumOfWeightsInConflictsMap = (for(i <- 0 to
maxValue)
  yield getSumOfWeightsInConflict(constraints, values,
currentNode, i) -> i).toMap
val valueWithSumOfWeightsInConflictsMinElement =
valueWithSumOfWeightsInConflictsMap.min
val minSumOfWeightsInConflictsValue =
valueWithSumOfWeightsInConflictsMinElement._2
val minSumOfWeightsInConflicts =
valueWithSumOfWeightsInConflictsMinElement._1
```

Η συνάρτηση getSumOfWeightsInConflict καλείται τόσες φορές, όσες οι τιμές στο πεδίο ορισμού. Το maxValue περνίεται και σαν παράμετρος στην συνάρτηση ώστε σε κάθε επανάληψη να δοκιμάζεται μια νέα τιμή. Αφού δημιουργηθεί η δομή map με κλειδί το άθροισμα των βαρών και με αντίστοιχη τιμή κλειδιού την τιμή που το

προκαλεί, αποθηκεύονται στην valueWithSumOfWeightsInConflictsMinElement. Μετά, ανατίθεται στην minSumOfWeightsInConflictsValue η τιμή που προκαλεί το μικρότερο άθροισμα βαρών και στην minSumOfWeightsInConflicts το μικρότερο άθροισμα.

Όπως έγινε και με τον αλγόριθμο Min Conflicts παρακάτω παρατίθεται το κομμάτι του προγράμματος στο οποίο φαίνεται πιο καθαρά η σειρά με την οποία καλούνται οι συναρτήσεις και πως χρησιμοποιούνται οι διάφορες δομές δεδομένων.

```
object BreakOut {

  def algorithm(repetitions: Int, filePath: String) = {
    val startingTime = System.currentTimeMillis κλήση της συνάρτησης συστήματος που μετράει τον χρόνο εκτέλεσης του αλγορίθμου
    val txt = MinConflicts.getTxtString(filePath) κλήση της getTxtString από τον MinConflicts για επεξεργασία του αρχείου με τους περιορισμούς
    var constraints = getConstraints(txt) κλήση της getConstraints που δημουργεί πίνακα με 3 στήλες. Στην 1η στήλη μπαίνει η 1η μεταβλητή του περιορισμού, στην 2η η 2η και στην 3η τα βάρη τους
    val lastNode = getLastNode(constraints) εύρεση πλήθους μεταβλητών
    val maxValue = MinConflicts.getMaxValue(filePath) κλήση getMaxValue από τον MC, για εύρεση της μεγαλύτερης τιμής του πεδίου ορισμού
    val values = MinConflicts.getRandomValues(maxValue, lastNode + 1) κλήση της getRandomalues από τον MC, για απόδοση τυχαίων τιμών από το πεδίο ορισμού στις μεταβλητές
   val conflicts = MinConflicts.getConflicts(constraints, values) κλήση της getConflicts, που επιστρέφει πίνακα με συγκρούσεις όπως διαμορφώθηκαν από τις τρέχουσες τιμές των μεταβλητών
    val nodesInConflict = conflicts.flatten.distinct ο πίνακας συγκρούσεων καθαρίζεται από διπλά στοιχεία ώστε όλες οι μεταβλητές να έχουν την ίδια πιθανότητα να επιλεχθούν για εξέταση
    var currentNode = nodesInConflict(Random.nextInt(nodesInConflict.length)) τυχαία επιλογή μεταβλητής προς εξέταση από τον πίνακα με τις συγκρούσεις
    var minNumOfConflicts = constraints.length αρχικοποίηση του ελάχιστου αριθμού συγκρούσεων ως το μήκος του πίνακα με τους περιορισμούς

    for (step <- 0 to repetitions){ από εδώ ξεκινά ο κόμβος των επαναλήψεων, οι οποίες καθορίζονται κάθε φορά από τον χρήστη
      val sumOfWeightsInConflicts = getSumOfWeightsInConflict(constraints, values, currentNode, values(currentNode)) επιστρέφει το άθροισμα των βαρών που προκύπτει
      val valueWithSumOfWeightsInConflictsMap = (for(i <- 0 to maxValue) δοκιμή των υπόλοιπων τιμών του πεδίου ορισμού
        yield getSumOfWeightsInConflict(constraints, values, currentNode, i) -> i).toMap βρίσκεται το άθροισμα των βαρών που προκαλεί η κάθε τιμή του πεδίου ορισμού
      val valueWithSumOfWeightsInConflictsMinElement = valueWithSumOfWeightsInConflictsMap.min στην δομή map μένει μόνο το μικρότερο άθροισμα βαρών και η τιμή που το προκάλεσε
      val minSumOfWeightsInConflictsValue = valueWithSumOfWeightsInConflictsMinElement._2 αποθηκεύεται το μικρότερο άθροισμα βαρών
      val minSumOfWeightsInConflicts = valueWithSumOfWeightsInConflictsMinElement._1 αποθηκεύεται ξεχωριστά η τιμή που προκάλεσε το μικρότερο άθροισμα
      if (sumOfWeightsInConflicts == minSumOfWeightsInConflicts || minSumOfWeightsInConflicts > sumOfWeightsInConflicts ){έλεγχος για τοπικό ελάχιστο
        constraints = getUpdatedConstraints(constraints, values, currentNode) ενημέρωση των βαρών στους περιορισμούς που είναι συγκρούσεις (αύξηση βάρους κατά 1 στην 3η στήλη του
      } else {  αν δεν βρίσκεται σε τοπικό ελάχιστο, υπάρχει τιμή που ελαχιστοποιεί τις συγκρούσεις                                                                    πίνακα των περιορισμών)
        values(currentNode) = minSumOfWeightsInConflictsValue η τιμή αυτή, ανατίθεται στο κόμβο που εξετάζονταν
      }
      currentNode = nodesInConflict(Random.nextInt(nodesInConflict.length)) επιλέγεται νέος κόμβος προς εξέταση μέσα από αυτούς που συμμετέχουν σε σύγκρουση
```

Εικόνα 11 Κομμάτι του κώδικα σε Scala για τον Break Out

Κεφάλαιο 4

Πειραματική μελέτη και αποτελέσματα

Οι δύο αλγόριθμοι που περιγράφηκαν στο προηγούμενο κεφάλαιο εξετάστηκαν ως προς το πόσο κοντά στην λύση ενός προβλήματος ικανοποίησης περιορισμών μπορούν να φτάσουν. Συγκρίνεται επίσης το χρονικό διάστημα που δαπάνησε ο καθένας μέχρι το τέλος των προκαθορισμένων επαναλήψεων. Αυτό που επιστρέφει ο κάθε αλγόριθμος δηλαδή και αυτό που τελικά μελετάται, είναι ο αριθμός των ελάχιστων συγκρούσεων που σημειώθηκε μέσα στις επαναλήψεις και σε πόσα milliseconds τις εκτέλεσε.

Από ότι έχει παρουσιαστεί μέχρι στιγμής θα πρέπει κανείς να περιμένει ότι ο Break Out θα είναι καλύτερος από τον Min Conflicts αφού κατά βάθος είναι ίδιοι αλλά ο Break Out μέσω του επιπρόσθετου μηχανισμού βαρών, καταφέρνει να αποδρά από τα τοπικά ελάχιστα και άρα μπορεί να συνεχίσει την αναζήτηση καλύτερης κατάστασης. Από την άλλη, ο Min Conflicts καλυτερεύει συνεχώς την κατάστασή του αφού αν οι νέες τιμές που δοκιμάζει δεν δίνουν καλύτερα αποτελέσματα κρατάει την τρέχουσα. Άρα όταν κολλήσει σε τοπικό ελάχιστο δεν θα προχωρήσει ποτέ σε καλύτερη από την τρέχουσα λύση. Ακόμη αναμένεται ότι ο Break Out ενώ μπορεί να φτάσει κοντά στην λύση, σε επόμενες επαναλήψεις μπορεί η κατάστασή του να χειροτερέψει. Αυτό συμβαίνει επειδή μετά την απόδραση από τοπικό ελάχιστο μπορεί να βρεθεί σε χειρότερη κατάσταση από ότι πριν από αυτό. Από την θεωρία που παρουσιάστηκε σχετικά με τα προβλήματα χρωματισμού γραφήματος και τα προβλήματα ικανοποίησης περιορισμών, αναμένεται ακόμη ότι όσο πιο μεγάλο είναι το πεδίο ορισμού τόσο πιο αποδοτικοί θα είναι οι αλγόριθμοι. Τέλος και σχετικά με τον χρόνο εκτέλεσης, θα υπέθετε κανείς ότι ο Break Out θα πρέπει να είναι πιο αργός αφού εκτελεί περισσότερες διαδικασίες σε σχέση με τον απλό Min Conflicts. Ελέγχει για τοπικό ελάχιστο και αναθέτει βάρη.

Παρακάτω περιγράφεται η πειραματική διαδικασία και τα αποτελέσματά της. Επιβεβαιώνονται άραγε τα παραπάνω θεωρητικά συμπεράσματα από τα πειράματα;

4.1 Η Διαδικασία της πειραματικής μελέτης

Αφού δημιουργήθηκαν οι αλγόριθμοι Min Conflicts και Break Out, ξεκίνησε η πειραματική διαδικασία. Εξετάστηκαν 100 προβλήματα ικανοποίησης περιορισμών,

διαφόρων δυσκολιών και μεγεθών. Το μεγαλύτερο πρόβλημα που εξετάστηκε αποτελούνταν από 1.916 μεταβλητές, οι οποίες συμμετείχαν σε 12.506 περιορισμούς και έπαιρναν τιμές από ένα πεδίο ορισμού τριών τιμών. Το μικρότερο πρόβλημα που εξετάστηκε αποτελούνταν από 11 μεταβλητές, 20 περιορισμούς και πεδίο ορισμού με 3 τιμές. Κάθε πρόβλημα εξετάστηκε 10 φορές και ο αριθμός των βημάτων ορίστηκε στις 50.000. Ο αριθμός των επαναλήψεων αυτών, επιλέχθηκε έτσι ώστε ακόμη και στα πολύ μεγάλα προβλήματα (με πάνω από 5.000 περιορισμούς), να προλαβαίνουν οι αλγόριθμοι να δείξουν τι μπορούν να κάνουν. Για παράδειγμα ο break out με λίγες επαναλήψεις μπορεί να μην συναντούσε τοπικό ελάχιστο και έτσι η απόδοσή του δεν θα ήταν πολύ καλύτερη από αυτή του min conflicts. Αρχικά ο αριθμός αυτός είχε οριστεί στα 1.000 αλλά μετά από σύντομη περίοδο δοκιμών διαφόρων τιμών, κρίθηκε καταλληλότερος ο αριθμός των 50.000. Επίσης, οι περισσότερες επαναλήψεις δίνουν ένα καλύτερο και πιο ολοκληρωμένο στατιστικό δείγμα προς μελέτη.

Για την καταγραφή του χρόνου εκτέλεσης των προγραμμάτων χρησιμοποιήθηκε η συνάρτηση *currentTimeMillis* η οποία δίνει την τρέχουσα ώρα σε milliseconds. Τοποθετήθηκε έτσι δύο φορές μέσα σε κάθε αλγόριθμο, μια στην αρχή και μια στο τέλος και έγινε η αφαίρεση για να βρεθεί ο χρόνος που χρειάστηκε μέχρι τον τερματισμό.

Σε αυτό το σημείο θεωρήθηκε σημαντικό να δοθεί η σύνταξη της main συνάρτησης του προγράμματος ώστε να αποκτήσει ο αναγνώστης μια εικόνα του πως εξελίχθηκε η πειραματική διαδικασία.

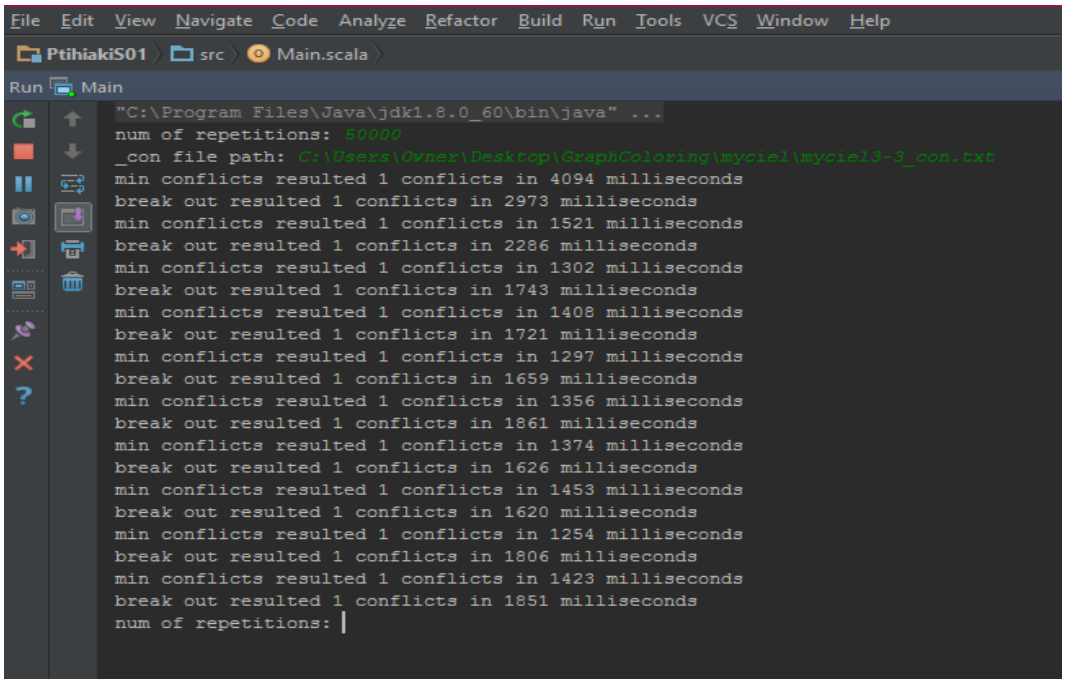
```
object Main {


  def main(args: Array[String]) = {
    while (true){
      print("num of repetitions: ")
      val repetitions = scala.io.StdIn.readLine().toInt
      print("_con file path: ")
      val filePath = scala.io.StdIn.readLine()
      for (i <- 1 to 10){
        MinConflicts.algorithm(repetitions, filePath)
```

BreakOut.*algorithm*(repetitions, filePath)

} } } }

Αρχικά εισάγεται από τον χρήστη ο αριθμός των βημάτων αναζήτησης. Ακριβώς μετά εισάγεται το μονοπάτι του txt αρχείου προβλήματος με κατάληξη _con που περιέχει τους περιορισμούς. Οι αλγόριθμοι καλούνται μέσα στον βρόγχο επανάληψης for, δέκα φορές. Στο τέλος αυτής της διαδικασίας συλλέγονταν τα αποτελέσματα.



Εικόνα 12 Στιγμιότυπο εκτέλεσης προγράμματος

Συλλέχθηκαν λοιπόν οι συγκρούσεις που απέμεναν σε κάθε εκτέλεση του προγράμματος και οι αντίστοιχοι χρόνοι. Στο τέλος υπολογίσθηκαν οι μέσοι όροι για τις 10 εκτελέσεις.

Να σημειωθεί ότι τα προγράμματα εκτελέστηκαν σε υπολογιστή Acer με επεξεργαστή Intel Core Duo 2.13GHz και 4GB RAM.

4.2 Αποτελέσματα

Παρακάτω, δίνονται αναλυτικά τα αποτελέσματα της πειραματικής διαδικασίας και για τα 100 προβλήματα και παρουσιάζονται κάποια στατιστικά στοιχεία. Στον πίνακα που ακολουθεί, στην $2^η$, την $3^η$, την $4^η$ και την $5^η$ στήλη, υπάρχουν τα στοιχεία του κάθε προβλήματος. Το όνομα, οι μεταβλητές, οι περιορισμοί στους οποίους συμμετείχαν και

οι τιμές του πεδίου ορισμού, αντίστοιχα. Στην 6η στήλη με όνομα MIN CONF είναι καταγεγραμμένοι οι μέσοι όροι των συγκρούσεων που δεν μπόρεσε να επιλύσει ο Min Conflicts κατά την διάρκεια των 50.000 επαναλήψεων και στην επόμενη ακριβώς στήλη, οι αντίστοιχοι χρόνοι μετρημένοι σε milliseconds[4].Στις δύο στήλες που ακολουθούν υπάρχουν τα ίδια στοιχεία για τον αλγόριθμο Break Out. Πρώτα οι συγκρούσεις και μετά οι χρόνοι εκτέλεσης.

Α/Α	ΟΝΟΜΑ ΑΡΧΕΙΟΥ	ΚΟΜΒΟΙ	ΠΕΡΙΟΡΙΣΜΟΙ	ΤΙΜΕΣ	MIN CONF	(MC)TIME IN mSecs	BREAK OUT	(BO)TIME IN mSecs
1	homer-5_con	561	1628	5	**84,2**	**12129,2**	**87,4**	**25170,2**
2	homer-8_con	561	1628	8	**25**	**86921,8**	**21,4**	**103684,7**
3	homer-10_con	561	1628	10	**10,3**	**35589,7**	**6**	**47521,5**
4	ash331GPIA-3_con	662	4185	3	**290,5**	**63257,8**	**301,1**	**218079,6**
5	ash608GPIA-3_con	1216	7844	3	**503,4**	**126236,9**	**594,8**	**440704,1**
6	ash958GPIA-3_con	1916	12506	3	**819,4**	**204755,1**	**1085,1**	**726559,8**
7	will199GPIA-5_con	701	7065	5	**142,1**	**44569,5**	**81,7**	**185274,8**
8	1-fullins-3-3_con	30	100	3	**2,8**	**1391,5**	**2**	**6030,7**
9	1-fullins-4-4_con	93	593	4	**6,6**	**1608,6**	**2**	**12159,2**
10	1-fullins-5-4_con	282	3247	4	**27,4**	**14614,8**	**16**	**64433,9**
11	2-fullins-3-4_con	52	201	4	**2,4**	**1734,3**	**1**	**10786,7**
12	2-fullins-4-4_con	212	1621	4	**10,9**	**16623,5**	**5**	**90191,6**
13	3-fullins-3-5_con	80	346	5	**2,2**	**6591,8**	**1**	**18964**
14	4-fullins-3-6_var	114	541	6	**1**	**1226,9**	**1**	**2817**
15	3-fullins-4-5_con	405	3524	5	**11**	**21441,8**	**3,1**	**71536,2**
16	4-fullins-4-6_con	690	6650	6	**10,9**	**62502,8**	**3,5**	**270798,2**
17	5-fullins-3-6_con	154	792	6	**4,7**	**5710,6**	**3**	**16537,1**
18	1-insertions-4-3_con	67	232	3	**9,3**	**1196,6**	**6**	**4419,6**
19	1-insertions-5-4_con	202	1227	4	**15,9**	**6542,1**	**7,7**	**23516,8**
20	2-insertions-3-3_con	37	72	3	**1,3**	**479,1**	**1**	**1543,4**
21	2-insertions-4-3_con	149	541	3	**15,7**	**2582,7**	**9,2**	**11077,9**
22	3-insertions-3-3_con	56	110	3	**4**	**668,7**	**1**	**2448,9**
23	3-insertions-4-3_con	281	1046	3	**24,9**	**4730,3**	**12,1**	**21281,1**
24	4-insertions-3-3_con	79	156	3	**4,2**	**543,5**	**1**	**3824,3**
25	4-insertions-4-3_con	475	1795	3	**43,4**	**5400,1**	**17,4**	**41640,6**
26	3-insertions-5-3_con	1406	9695	3	**195,6**	**73508,4**	**167,2**	**560232,1**
27	1-insertions-6-4_con	607	6337	4	**81,4**	**48296,6**	**57,4**	**354468,4**
28	2-insertions-5-3_con	597	3936	3	**119,4**	**30336,6**	**102,2**	**224122,9**
29	lei450-05a-04_con	450	5714	4	**583,2**	**88284,3**	**508,3**	**369374,3**
30	lei450-05a-05_con	450	5714	5	**276,5**	**68711,7**	**13,1**	**342228,4**

[4] 1 second = 1000 milliseconds

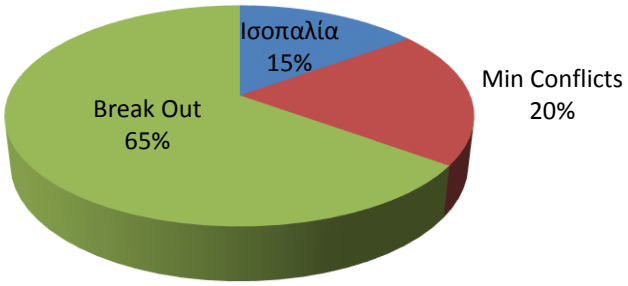
31	mug88-1-3_con	88	146	3	**8,4**	**930,1**	**1**	**3590,3**
32	mug88-1-4_con	88	146	4	**0**	**7957**	**0**	**9886,2**
33	mug88-25-3_con	88	146	3	**8,3**	**7626**	**1**	**11086,1**
34	mug88-25-4_con	88	146	4	**0**	**3370,3**	**0**	**4600,3**
35	mug100-1-3_con	100	166	3	**8,4**	**3007,5**	**1**	**3602,6**
36	mug100-1-4_con	100	166	4	**0**	**3067,5**	**0**	**3593,2**
37	mug100-25-3_con	100	166	3	**9,1**	**3044,1**	**1**	**3668,8**
38	mug100-25-4_con	100	166	4	**0**	**2842,1**	**0**	**3583,8**
39	myciel3-3_con	11	20	3	**1**	**451,6**	**1**	**596,9**
40	myciel3-4_con	11	20	4	**0**	**441,2**	**0**	**528**
41	myciel4-4_con	23	71	4	**1**	**1420,2**	**1**	**1684,5**
42	myciel4-5_con	23	71	5	**0**	**1239,3**	**0**	**1627,5**
43	myciel5-4_con	47	236	4	**6,2**	**3618,1**	**4**	**4745,9**
44	myciel5-5_con	47	236	5	**1**	**10066,5**	**1**	**13346,5**
45	myciel5-6_con	47	236	6	**0**	**12128,8**	**0**	**14289,5**
46	myciel6-4_con	95	755	4	**24**	**33139,4**	**16,8**	**44478,2**
47	myciel6-5_con	95	755	5	**6,9**	**33560,8**	**4**	**11190,56**
48	myciel6-6_con	95	755	6	**1**	**36308**	**1**	**48255**
49	myciel6-7_con	95	755	7	**0**	**35451,3**	**0**	**47479,6**
50	anna-5_con	138	493	5	**20,9**	**24154,8**	**17,2**	**31746,5**
51	anna-8_con	138	493	7	**3,5**	**9464,9**	**3**	**12460,7**
52	david-5_con	87	406	5	**25,5**	**19087,4**	**23,1**	**25081**
53	david-8_con	87	406	8	**6**	**7504**	**4**	**9834,5**
54	huck-5_con	74	301	5	**22,8**	**5168,4**	**21,4**	**6911,5**
55	huck-8_con	74	301	8	**6,2**	**5779,8**	**6**	**7531,1**
56	jean-5_con	80	254	5	**16,1**	**4539**	**14,3**	**5850,6**
57	jean-7_con	80	254	7	**5,5**	**4722,8**	**4**	**15176,8**
58	games120-5_con	120	638	5	**33,5**	**10733,8**	**26,6**	**14724,5**
59	games120-7_con	120	638	7	**10,2**	**11726,5**	**8**	**15642,1**
60	games120-8_con	120	638	8	**2,3**	**11743,4**	**2**	**15440,8**
61	games120-9_con	120	638	9	**0**	**22305,5**	**0**	**27190,9**
62	miles250-6_con	128	387	6	**6,7**	**7036,9**	**4**	**9154,8**
63	miles250-7_con	128	387	7	**3,7**	**7216,7**	**1**	**9422,9**
64	miles250-8_con	128	387	8	**1,3**	**14056,8**	**0**	**16968,8**
65	miles500-5_con	128	1170	5	**120,3**	**30672,9**	**126,6**	**39240,9**
66	miles500-10_con	128	1170	10	**28,3**	**24332,5**	**25,7**	**32887**
67	miles500-15_con	128	1170	15	**8,2**	**40810,2**	**5,4**	**55722,3**
68	miles500-18_con	128	1170	18	**4,4**	**38365,2**	**2**	**43839,8**
69	queen5-5-4_con	25	160	4	**15,4**	**2570,7**	**12**	**3540,5**
70	queen6-6-6_con	36	290	6	**8,1**	**4824,1**	**4**	**6470,4**
71	queen7-7-6_con	49	476	6	**23,3**	**7736,6**	**16,1**	**10473,1**
72	queen8-8-8_con	64	728	8	**16,2**	**19212,8**	**5,7**	**23597,7**
73	queen8-12-8_con	96	1368	8	**52,6**	**42835,8**	**47,4**	**46069**
74	queen9-9-8_con	81	1056	8	**56,1**	**26292,7**	**51,4**	**33894,7**
75	queen10-10-8_con	100	1470	8	**58,4**	**27135,9**	**54,2**	**35605,1**

76	queen10-10-12_con	100	1470	12	**7,6**	**34438,9**	**0**	**43352,9**
77	queen11-11-8_con	121	1980	8	**93,6**	**89242,3**	**91,3**	**211557,9**
78	queen11-11-12_con	121	1980	12	**16,3**	**44972,3**	**7,9**	**54994,5**
79	queen12-12-13_con	144	2596	13	**19,2**	**102803,9**	**11,3**	**135282,9**
80	queen13-13-10_con	169	3328	10	**109,1**	**122529,2**	**112,4**	**162037,2**
81	queen13-13-14_con	169	3328	14	**23,9**	**165759,8**	**14,6**	**217196,2**
82	queen14-14-12_con	196	4186	12	**89,4**	**152451,1**	**89,9**	**165352,7**
83	mulsol-i-1-05_con	197	3925	5	**392,4**	**134838,7**	**435,1**	**174408,9**
84	mulsol-i-1-10_con	197	3925	10	**159,5**	**125268,3**	**173**	**170034,7**
85	mulsol-i-1-15_con	197	3925	15	**89,2**	**165203,3**	**92,7**	**216931,2**
86	mulsol-i-1-20_con	197	3925	20	**54,4**	**237483,3**	**55,1**	**241917**
87	mulsol-i-1-25_con	197	3925	25	**35,2**	**195727,3**	**34,7**	**293600,9**
88	mulsol-i-2-05_con	188	3885	5	**228,1**	**93562,1**	**265,8**	**132726,2**
89	mulsol-i-2-10_con	188	3885	10	**80,3**	**168464,3**	**87,8**	**248063,2**
90	mulsol-i-2-15_con	188	3885	15	**40,1**	**215372,9**	**41,1**	**235546,9**
91	mulsol-i-2-20_con	188	3885	20	**22**	**266954,6**	**22**	**341587,7**
92	mulsol-i-2-25_con	188	3885	25	**12**	**426494,9**	**12**	**467516,7**
93	zeroin-i-2-05_con	211	3541	5	**188,7**	**55488,4**	**223,2**	**71487,8**
94	zeroin-i-2-10_con	311	3541	10	**66,3**	**79748**	**75**	**93832,9**
95	zeroin-i-2-15_con	311	3541	15	**30,7**	**283967,5**	**32,6**	**301388**
96	zeroin-i-2-20_con	311	3541	20	**14,6**	**343887,7**	**14,2**	**371146,7**
97	zeroin-i-3-05_con	206	3540	5	**187,6**	**192888,7**	**218,7**	**232813**
98	zeroin-i-3-10_con	206	3540	10	**67,2**	**248303,1**	**74,3**	**291045,4**
99	zeroin-i-3-15_con	206	3540	15	**31,1**	**90701,3**	**32,1**	**98755**
100	zeroin-i-3-20_con	206	3540	20	**14,8**	**94827,3**	**14**	**98313,7**

Πίνακας 1 Πειραματικά Αποτελέσματα

Ποιος αλγόριθμος έλυσε τα περισσότερα προβλήματα

Αν παρατηρήσει κανείς τα αποτελέσματα σε σχέση με το ποιος αλγόριθμος έφτασε πιο κοντά στην λύση, η υπεροχή του Break Out είναι φανερή. Στο 65% των προβλημάτων ο Break Out κατάφερε να αφήσει λιγότερες συγκρούσεις σε σχέση με τον Min Conflicts, ο οποίος ήταν καλύτερος μόνο στο 20% των περιπτώσεων. Στο υπόλοιπο 15% ήρθαν ισοπαλία.

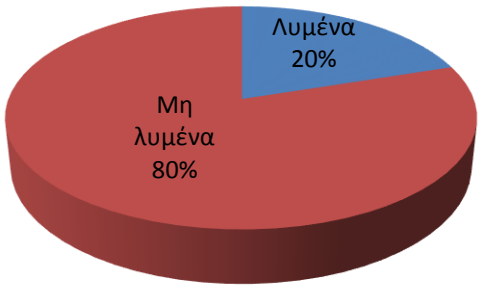


Γράφημα 1 Σύγκριση συγκρούσεων

Ενδιαφέρον παρουσιάζει το γεγονός ότι στις 15 υποθέσεις ισοπαλίας, οι 8 αναφέρονται σε περιπτώσεις που το πρόβλημα λύθηκε (οι αλγόριθμοι επέστρεψαν 0 εναπομένουσες συγκρούσεις), οι 5 άφησαν μία μόνο σύγκρουση και οι υπόλοιπες δύο παραπάνω.

Σε πόσα προβλήματα βρέθηκε λύση

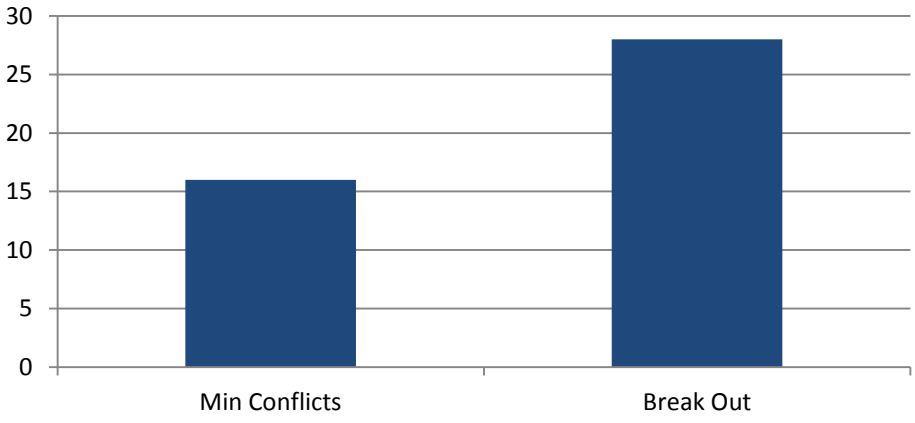
Στο σύνολο των 100 προβλημάτων βρέθηκε λύση σε 11 από αυτά. Ο Break Out ήταν και πάλι καλύτερος, με μικρή όμως διαφορά. Ο Min Conflicts κατάφερε να λύσει 9 προβλήματα, ενώ ο Break Out έλυσε αυτά τα 9, συν άλλα 2. Αυτό σημαίνει ότι ο Break Out έχει 18% περισσότερες πιθανότητες να φτάσει σε λύση. Να αναφερθεί ότι όλα αυτά τα προβλήματα θα μπορούσαν να χαρακτηριστούν μικρά αφού δεν περιείχαν πάνω από 1500 περιορισμούς. Τα εφτά από αυτά είχαν από 20 μέχρι 236 περιορισμούς να επιλύσουν και τα υπόλοιπα από 755 ως 1470 αλλά τα πεδία ορισμού που τους αντιστοιχούν έχουν τουλάχιστον 7 τιμές. Η σχέση μεταξύ τιμών πεδίου ορισμού και λύσης αναλύεται παρακάτω αλλά για την ώρα να σημειωθεί ότι το μεγάλο πεδίο ορισμού σημαίνει και μεγαλύτερες πιθανότητες για λύση. Ακόμη είναι σημαντικό να αναφερθεί ότι δεν ήταν γνωστό αν προβλήματα που εξετάστηκαν ήταν επιλύσιμα ή όχι.



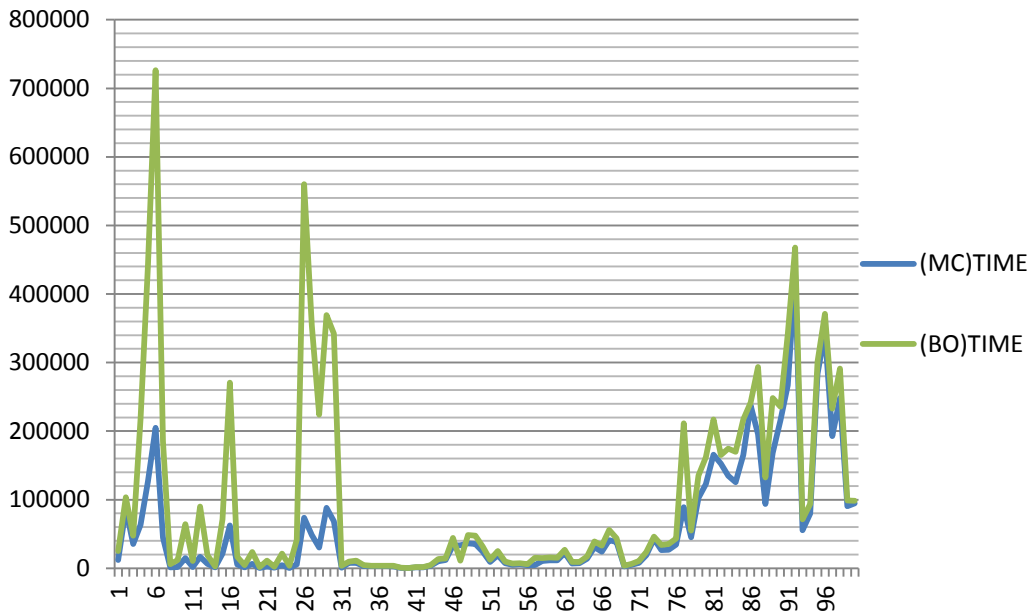
Γράφημα 2 Λυμένα και μη λυμένα

Οι χρόνοι εκτέλεσης για τον κάθε αλγόριθμο

Για να εκτελεστούν και τα 100 προβλήματα από 10 φορές το καθένα με 50.000 βήματα χρειάστηκαν περίπου 44 ώρες. Οι 16 από αυτές δαπανήθηκαν στον Min Conflicts και οι υπόλοιπες 28 στον Break Out. Αυτό καθιστά τον Break Out 42% πιο αργό από τον Min Conflicts, με μέσο χρόνο εκτέλεσης ενός προβλήματος τα 10,1 δευτερόλεπτα έναντι του Min Conflicts με 5,8 δευτερόλεπτα.



Γράφημα 3 Σύγκριση χρόνων εκτέλεσης αλγορίθμων



Γράφημα 4 Ο χρόνος εκτέλεσης του Break Out σε σχέση με τον Min Conflicts

Το Γράφημα 5 δείχνει πως ο Break Out (πράσινη γραμμή) χρειάζονταν περισσότερο χρόνο από τον Min Conflicts (μπλε γραμμή) για την εκτέλεση του ίδιου προβλήματος. Ο Break Out δεν έκανε καλύτερο χρόνο σε ούτε μια από τις 100 περιπτώσεις.

Σε ποια προβλήματα είναι καλύτερος ο Break Out

Φαίνεται να έχει ενδιαφέρον και ο τρόπος κατανομής των περιπτώσεων στις οποίες ο Break Out είναι καλύτερος από τον Min Conflicts. Ο χωρισμός σε κατηγορίες έγινε αυθαίρετα, ανάλογα με τον αριθμό των περιορισμών. Θα μπορούσε να επιλεγεί και κάποιο άλλο στοιχείο ως μέτρο χωρισμού, αλλά προτιμήθηκαν οι περιορισμοί αφού αντικατοπτρίζουν καλύτερα το μέγεθος ενός προβλήματος. Όπως αναφέρθηκε και προηγουμένως οι περισσότεροι περιορισμοί που συναντήθηκαν σε ένα πρόβλημα ήταν οι 12.506 ενώ οι λιγότεροι ήταν 20.

Διαμορφώθηκαν οι παρακάτω τέσσερις κατηγορίες μικρό, μεσαίο, μεγάλο και πολύ μεγάλο, όπου στην πρώτη υπάρχουν 49 προβλήματα, στην δεύτερη 18, στην τρίτη 25 και στην τέταρτη 8.

Κατηγορία	Περιορισμοί	Προβλήματα	BO	MC	Ισοπαλία
Μικρό	0-1000	49	36	0	13
Μεσαίο	1000-3000	18	17	1	0
Μεγάλο	3000-5000	25	6	17	2
Πολύ μεγάλο	5000-…..	8	6	2	0
		100	65	20	15

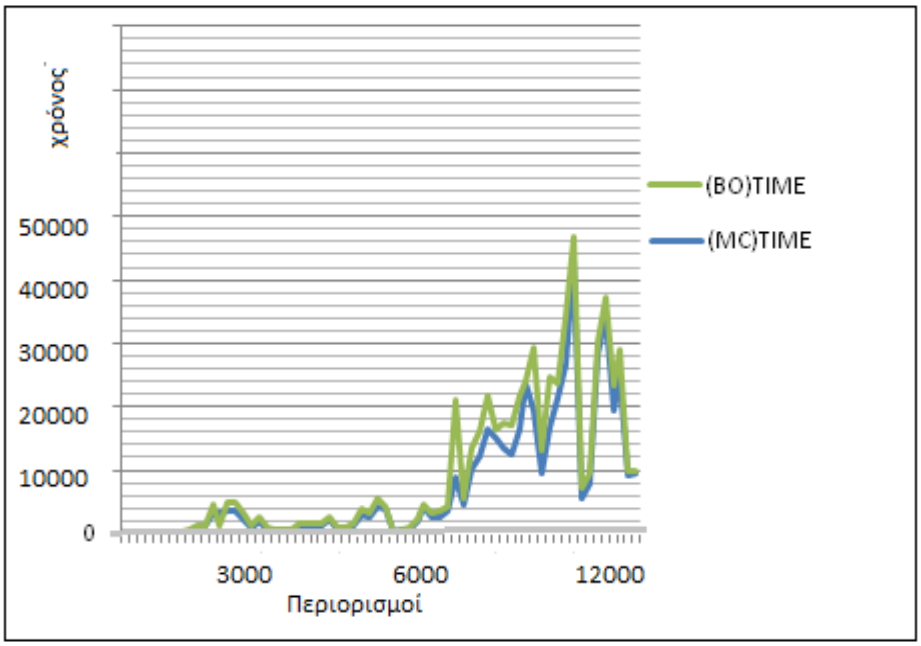
Πίνακας 2 Κατηγορίες προβλημάτων

Στην τέταρτη στήλη με όνομα BO σημειώνεται ο αριθμός των προβλημάτων ανά κατηγορία, στα οποία ο Break Out έφερε καλύτερα αποτελέσματα από ότι ο Min Conflicts. Στην επόμενη στήλη με όνομα MC σημειώνονται τα προβλήματα στα οποία ήταν καλύτερος ο Min Conflicts και στην τελευταία στήλη σε πόσα ήταν ισοπαλία. Είναι εύκολο να παρατηρήσει κανείς ότι στα μικρά προβλήματα ο Break Out ήταν πολύ καλύτερος από τον αντίπαλό του, εκτός από 13 περιπτώσεις ισοπαλίας. Πολύ καλά τα πήγε και στα μεσαία, όπου ο Min Conflicts ήταν καλύτερος μόνο μια φορά έναντι των 17 του Break Out. Στα μεγάλα όμως προβλήματα η κατάσταση αντιστρέφεται και ο Min Conflicts είναι φανερά καλύτερος αφού «κερδίζει» σε 17 περιπτώσεις, ενώ ο Break Out σε 6 και υπάρχουν και 2 ισοπαλίες. Αυτό ίσως οφείλεται και στο γεγονός ότι τα 18 από τα 25 προβλήματα της κατηγορίας είναι της ίδιας κλάσης προβλημάτων (musol και zeroin) και στα 14 από αυτά, ο αριθμός των μεταβλητών και των περιορισμών αλλάζει ελάχιστα ή καθόλου κάθε φορά (αλλάζει το πεδίο ορισμού). Αν τα προβλήματα είχαν κατηγοριοποιηθεί ώστε οι δύο μεσαίες κατηγορίες να ήταν μια και έτσι να υπήρχαν 49 προβλήματα στην πρώτη και 43 στην δεύτερη τότε ο Break Out θα ήταν καλύτερος κατά 5 περιπτώσεις.

Σχέση αριθμού περιορισμών και χρόνου εκτέλεσης

Στο παρακάτω γράφημα γίνεται σύγκριση των μεγεθών χρόνου εκτέλεσης και περιορισμών. Παρατηρείται δηλαδή, το πώς η αύξηση των περιορισμών, επηρεάζει τον χρόνο εκτέλεσης του κάθε αλγορίθμου. Σε γενικές γραμμές καθώς μεγαλώνει ο χώρος καταστάσεων, φαίνεται να αυξάνεται και ο χρόνος εκτέλεσης. Πρέπει όμως να ληφθεί υπόψη ότι ο χρόνος δεν επηρεάζεται μόνο από τους περιορισμούς αλλά και από άλλους παράγοντες, όπως το πόσο μεγάλο ήταν το πεδίο ορισμού και άρα πόσες νέες τιμές

έπρεπε να δοκιμαστούν κάθε φορά αλλά και από εξωτερικούς παράγοντες όπως το hardware και η κατάσταση στην οποία βρίσκονταν ο υπολογιστής στον οποίο εκτελούνταν τα προγράμματα κλπ.



Γράφημα 5 Σχέση χρόνου εκτέλεσης –περιορισμών προβλήματος

Στον οριζόντιο άξονα αυξάνονται οι τιμές των περιορισμών ενώ στον κατακόρυφο οι τιμές του χρόνο εκτέλεσης. Το πράσινο χρώμα είναι για τον Break Out και το μπλε για τον Min Conflicts. Φαίνεται ότι και οι δύο αλγόριθμοι ακολουθούν σχεδόν ίδιο μοτίβο σε σχέση με την αύξηση χρόνου, με τον Break Out να βρίσκεται σε μεγαλύτερα επίπεδα κάτι που είναι λογικό αφού όπως αναφέρθηκε προηγουμένως είναι πιο αργός από τον Min Conflicts.

Σχέση μεγέθους πεδίου ορισμού και πιθανότητας καλύτερων αποτελεσμάτων

Παρατηρήθηκε ακόμη πως καθώς αυξάνονται τα στοιχεία του πεδίου ορισμού σε ένα πρόβλημα ενώ οι μεταβλητές και οι περιορισμοί παραμένουν ίδιοι, αυξάνονται και οι πιθανότητες αυτό να λυθεί. Σε όλες τις περιπτώσεις, οι συγκρούσεις που απέμειναν μετά τις 50.000 επαναλήψεις και αφού κάθε φορά το πεδίο ορισμού μεγάλωνε, είχαν μειωθεί. Αυτό φυσικά είναι λογικό, αφού αν σκεφτεί κανείς το πρόβλημα χρωματισμού χάρτη που εξηγήθηκε στο πρώτο κεφάλαιο της εργασίας, θα διαπιστώσει εύκολα ότι αν

υπήρχε ακόμη ένα χρώμα, ο χρωματισμός θα ήταν πιο εύκολος, ενώ αν υπήρχε ένα λιγότερο, ο χρωματισμός θα ήταν βασικά αδύνατος. Παρακάτω δίνονται 5 γραμμές του πίνακα 1 με τα πειραματικά αποτελέσματα όλων των προβλημάτων, ώστε να υποστηριχθεί το παραπάνω συμπέρασμα. Ο Min Conflicts ξεκινά με 392,4 εναπομένουσες συγκρούσεις και πεδίο ορισμού με 5 τιμές και καταλήγει σε 35,2 συγκρούσεις όταν το πεδίο ορισμού είναι 25. Οι αντίστοιχες τιμές για τον Break Out είναι 435,1 και 34,7 συγκρούσεις.

		Μεταβλητές	Περιορισμοί	Πεδίο Ορισμού	Min Conflicts	Break Out
83	mulsol-i-1-05_con	197	3925	5	**392,4**	**435,1**
84	mulsol-i-1-10_con	197	3925	10	**159,5**	**173**
85	mulsol-i-1-15_con	197	3925	15	**89,2**	**92,7**
86	mulsol-i-1-20_con	197	3925	20	**54,4**	**55,1**
87	mulsol-i-1-25_con	197	3925	25	**35,2**	**34,7**

Πίνακας 3 Σχέση πεδίου ορισμού και συγκρούσεων

Στην συγκεκριμένη περίπτωση μπορεί κανείς να παρατηρήσει ότι καθώς αυξάνεται το πεδίο ορισμού, ο Break Out φαίνεται να κλείνει την ψαλίδα της διαφοράς του με τον Min Conflicts και στο τέλος καταφέρνει να δώσει καλύτερα αποτελέσματα. Στις 23 παρόμοιες περιπτώσεις στις οποίες τα προβλήματα ήταν ίδια εκτός από το πεδίο ορισμού που αυξάνονταν, στις 8 παρατηρήθηκε φαινόμενο όπως το παραπάνω όπου ο Break Out καλυτερεύει την απόδοσή του σε σχέση με τον Min Conflicts. Στις 12 ήταν ήδη καλύτερος ενώ στις υπόλοιπες ήταν είτε ισοπαλία είτε ο Min Conflicts ήταν καλύτερος.

Η απόδοση του χρόνου εκτέλεσης δεν μελετάται εκτενώς αφού όπως αναφέρθηκε και στις πληροφορίες για το hardware, ο υπολογιστής στον οποίο εκτελέστηκαν τα προβλήματα δεν έδινε αξιόπιστα αποτελέσματα, λόγω της παλαιότητας και πολυχρηστίας του. Για παράδειγμα στο πρόβλημα homer-5_con, για τον Min Conflicts, ο μέσος όρος χρόνου εκτέλεσης για 50.000 επαναλήψεις και για 10 εκτελέσεις ήταν 12129,2 milliseconds ενώ σε άλλη περίπτωση εκτέλεσης του σε διαφορετική χρονική στιγμή, ήταν 18956,2 milliseconds. Γεγονός όμως παραμένει ότι ο Break out είναι πιο αργός από τον Min Conflicts.

Κεφάλαιο 5

Συμπεράσματα

Στην εισαγωγή του προηγούμενου κεφαλαίου είχαν γίνει κάποιες «προβλέψεις», σχετικά με τα αποτελέσματα της πειραματικής διαδικασίας που θα έπρεπε να αναμένει κανείς, βασιζόμενος στην θεωρία που παρατέθηκε στα πρώτα κεφάλαια. Θα εξεταστεί παρακάτω αν επιβεβαιώθηκαν αυτά και θα σχολιαστούν και άλλα συμπεράσματα που προέκυψαν μετά την διεξαγωγή των στατιστικών στοιχείων.

Η πρώτη υπόθεση ήταν ότι ο Break Out θα έχει καλύτερη απόδοση από τον Min Conflicts. Αυτό όντως συνέβη αφού όπως φαίνεται και στο Γράφημα 1 έφερε καθαρά καλύτερα αποτελέσματα σε 65 από τις 100 περιπτώσεις που μελετήθηκαν. Ακόμη σε δοκιμές που δεν έχουν καταγραφεί παραπάνω και παρατηρούνταν η έξοδος σε κάθε ένα από τα επαναληπτικά βήματα, ήταν φανερό ότι ο Min Conflicts μόνο βελτίωνε την έξοδό του ενώ ο Break Out έκανε «κύκλους» στα αποτελέσματά του αλλά όπως αποδείχθηκε, στο 65% των περιπτώσεων μέσα στις 50.000 επαναλήψεις υπήρχε έστω μια φορά που ήταν καλύτερος από τον Min Conflicts. Κατάφερε επίσης να επιβεβαιωθεί και το γεγονός ότι όσο μεγαλώνει το πεδίο ορισμού ενός προβλήματος τόσο πιο κοντά στην λύση μπορεί αυτό να φτάσει (Πίνακας 3). Τέλος ο χρόνος εκτέλεσης του Break Out ήταν όντως μεγαλύτερος από αυτόν του Min Conflicts (Γράφημα 4).

Τα αποτελέσματα στα γραφήματα σχετικά με τον χρόνο, υπάρχουν κυρίως για να δώσουν μια γενική εικόνα της συμπεριφοράς της διάρκειας εκτέλεσης των αλγορίθμων. Τα γραφήματα που έχουν σχεδιαστεί παραπάνω εξετάζουν τον χρόνο σε σχέση με τους περιορισμούς. Παρόλα αυτά επειδή αυτό το μέτρο επηρεάζεται από το πεδίο ορισμού, τις μεταβλητές και την κατανομή των λύσεων δεν είναι σωστό να εμπιστευθεί κανείς μόνο τα παραπάνω γραφήματα για να βγάλει συμπεράσματα.

Αν το ζητούμενο αυτής της εργασίας ήταν να αποδειχθεί ότι ο Break Out είναι ένας πολύ αποτελεσματικός αλγόριθμος σε προβλήματα ικανοποίησης περιορισμών τότε αυτό επιτεύχθηκε. Φάνηκε ακόμη ότι η μέθοδος ανάθεσης βαρών στους περιορισμούς ήταν πολύ αποδοτική και αν και δεν βρέθηκε λύση σε παραπάνω από 11 προβλήματα στα 100, αυτό δεν πρέπει να φαίνεται λίγο ή απογοητευτικό αφού δεν ήταν γνωστό το

αν τα προβλήματα που εξετάστηκαν είχαν όντως λύση. Αν ο χρόνος εκτέλεσης είναι αδιάφορος, τότε ο Break Out είναι μια πολύ καλή επιλογή για επίλυση προβλημάτων τοπικής αναζήτησης, ειδικά σε σχέση με τον βασικό Hill Climbing αλγόριθμο, Min Conflicts. Στα μικρά προβλήματα που ο χρόνος εκτέλεσης είναι αμελητέος (πχ 400 millisecond για τον Min Conflicts και 500 για τον Break Out), ο Break Out είναι μια καθαρά καλύτερη επιλογή.

Κεφάλαιο 6

Μελλοντική Εργασία

Για μελλοντική ενασχόληση με τον Break Out θα προτείνονταν η χρήση προβλημάτων με γνωστό το αν έχουν λύση ή όχι ώστε να σχηματισθεί μια άποψη για το ποσοστό στο οποίο ο Break Out μπορεί να την βρει εφόσον αυτή υπάρχει, αφού όπως αναφέρει ο Paul Morris ο αλγόριθμος δεν είναι πλήρης αλλά στην πράξη βρίσκει σχεδόν πάντα την λύση. Ακόμη θα είχε ενδιαφέρον μια έρευνα ώστε να καθοριστεί η συμπεριφορά του Break Out σε μεγαλύτερα προβλήματα με περισσότερες επαναλήψεις αφού σε αυτή την εργασία καλύφθηκαν μόνο 8 περιπτώσεις στις οποίες το πρόβλημα είχε παραπάνω από 5000 περιορισμούς.

Ακόμη όπως αναφέρθηκε η τεχνολογία του υπολογιστή στον οποίο εκτελέστηκαν οι αλγόριθμοι ήταν αρκετά ξεπερασμένη και δεν μπόρεσε να γίνει έτσι εκτενέστερη έρευνα σχετικά με τον χρόνο εκτέλεσης. Θα μπορούσε λοιπόν κάποιος να τρέξει τα ίδια ή και παρόμοια προβλήματα σε υπολογιστή νεότερης τεχνολογίας ώστε να διεξαχθούν πιο έγκυρα και ακριβή αποτελέσματα καθώς και πως αυτός ο χρόνος διαμορφώνεται σε σχέση με άλλα στοιχεία των προβλημάτων.

Αν θέλει κανείς να ασχοληθεί με την τεχνητή νοημοσύνη, οι αλγόριθμοι αναζήτησης και τα προβλήματα ικανοποίησης περιορισμών είναι μόνο η αρχή. Σαν επιστήμη συγκεντρώνει τεράστιο ενδιαφέρον και απλώνεται σε πολλά άλλα επιστημονικά πεδία. Ρομποτική, νευροεπιστήμες, μαθηματικά και ηλεκτρονικά παιχνίδια, είναι μόνο λίγα από τα πεδία στα οποία συναντάται.

Βιβλιογραφία

[1] https://el.wikipedia.org/wiki/Τεχνητή_νοημοσύνη

[2] Stuart Russell & Peter Norvig, Τεχνητή Νοημοσύνη, μια σύγχρονη προσέγγιση, Κλειδάριθμος, 2η έκδοση, 2005

[3] www.mind.ilstu.edu/curriculum/modOverview.php?modGUI=212

[4] http://www.reading.ac.uk/news-and-events/releases/PR583836.aspx

[5]http://anonhq.com/stephen-hawking-warns-robots-nuclear-war-aliens-will-wipe-humanity-less-100-years/

[6] http://ebooks.edu.gr/modules/ebook/show.php/DSGL-C101/36/198,1059/

[7] https://www.cs.ucy.ac.cy/~mavronic/Classes/cs232/Notes/notes1.pdf

[8] http://dimitris.webgalaxy.gr/science-thewrima-mi-plirotitas.php

[9] Kostas Stergiou, Lecture 2 for the course of Artificial Intelligence at UOWM

[10] Kostas Stergiou, Lecture 7 for the course of Artificial Intelligence at UOWM

[11] http://aibook.csd.auth.gr/include/slides/Chap03.pdf

[12]http://ai.uom.gr/Courses/AdvancedArtificialIntelligence/Slides/Constraints_and_CLP .pdf

[13] http://aibook.csd.auth.gr/include/slides/Chap02.pdf

[14] Paul Morris, The Breakout method from escaping local minima, Paper for IntelliCorp, 1993

[15] Michele Lemaitre & Gerard Verfaillie, An incomplete method for solving distributed valued constraint satisfaction problems, Paper, 2002

[16] Kostas Stergiou, Lecture 3 for the course of Artificial Intelligence at UOWM

[17] https://en.wikipedia.org/wiki/Heuristic_(computer_science)

[18] Makoto Yokoo, Weak-commitment search for solving constraint satisfaction problems, Paper for NTT Communications science laboratories, 1994

[19] Tommy R.Jensen and Bjarne Toft, Graph Coloring problems, 1995

[20] http://artint.info/html/ArtInt_83.html

[21] Marcus Bohlin, Constraint satisfaction by local search, Paper for the Swedish Institute of Computer Science, 2002

[22] https://el.wikipedia.org/wiki/Scala

[23] https://en.wikipedia.org/wiki/Min-conflicts_algorithm