ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Διπλωματική Εργασία

Αλγόριθμοι Εύρεσης Φυσικών Αποδείξεων

Βουδούρης Αλέξανδρος Ανδρέας

Α.Μ. 4417

voudouris@ceid.upatras.gr

Επιβλέπων Καθηγητής
Σταύρος Κοσμαδάκης

Περιεχόμενα

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κύριο Σταύρο Κοσμαδάκη, για τις πολύτιμες συμβουλές και την καθοδήγηση που μου προσέφερε κατά την διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας.

Εισαγωγή

Η παρούσα διπλωματική εργασία ασχολείται με την προτασιακή λογική και τους αλγορίθμους αναζήτησης φυσικών αποδείξεων λογικών συνεπαγωγών μέσω του αποδεικτικού συστήματος Natural Deduction.

Η εργασία χωρίζεται σε τέσσερα κεφάλαια και ένα παράρτημα.

Στο πρώτο κεφάλαιο γίνεται μια μικρή εισαγωγή στην προτασιακή λογική και στις έννοιες που την αφορούν και οι οποίες θα μας φανούν χρήσιμες στα επόμενα κεφάλαια. Επίσης, γίνεται μια απλή παρουσίαση του αποδεικτικού συστήματος Natural Deduction πάνω στο οποίο θα βασιστούν οι αλγόριθμοι αναζήτησης που θα παρουσιάσουμε.

Στο δεύτερο κεφάλαιο παρουσιάζεται ο βασικός αλγόριθμος αναζήτησης φυσικών αποδείξεων ο οποίος στηρίζεται στο αποδεικτικό σύστημα Natural Deduction. Παρουσιάζουμε τα βασικά τμήματα από τα οποία αποτελείται ο αλγόριθμος, τον συνολικό αλγόριθμο, μια απόδειξη της ορθότητας και της πληρότητας του και τέλος, μια μελέτη της πολυπλοκότητας του.

Στο τρίτο κεφάλαιο προσπαθούμε να βελτιώσουμε τον αλγόριθμο του δεύτερου κεφαλαίου. Σε πρώτο βήμα αναγνωρίζουμε τα προβλήματα του αλγορίθμου και έπειτα προσπαθούμε να τα διορθώσουμε αλλάζοντας το υποκείμενο αποδεικτικό σύστημα χωρίς ωστόσο να επηρεάσουμε την ορθότητα και την πληρότητα του.

Στο τέταρτο κεφάλαιο σχολιάζουμε λίγο την υλοποίηση του βασικού αλγορίθμου. Γίνεται μια μικρή παρουσίαση του προγράμματος καθώς και των βασικών του στοιχείων. Ο κώδικας της υλοποίηση παρουσιάζεται ξεχωριστά στο παράρτημα στο τέλος της εργασίας.

Κεφάλαιο 1
Προτασιακή Λογική

1.1 Σύνταξη Προτασιακής Λογικής

Όπως κάθε γλώσσα, έτσι και κάθε λογική έχει το δικό της λεξιλόγιο και σύνταξη. Θα δούμε τα σύμβολα με τα οποία κατασκευάζουμε προτάσεις καθώς και ποιές από αυτές ανήκουν στην γλώσσα που ορίζει η προτασιακή λογική.

Το λεξιλόγιο της λογικής αποτελείται από προτασιακά γράμματα, σταθερές, λογικούς τελεστές και παρενθέσεις. Πιο συκρεκριμένα:

Υποθέτουμε ότι διαθέτουμε ένα άπειρο σύνολο από προτασιακά γράμματα $p_1,p_2,\ldots$ τα οποία είναι διακριτά μεταξύ τους. Για χάρη ευκολίας θα χρησιμοποιούμε τα γράμματα p,q,t,...

Υπάρχουν δύο σταθερές από την στιγμή που η λογική μας είναι δίτιμη και κάθε τύπος μπορεί να πάρει είτε αληθή είτε ψευδή τιμή. Συμβολίζουμε με $\top$ την τιμή «αληθής» και με $\bot$ την τιμή «ψευδής».

Οι λογικοί τελεστές που θα χρησιμοποιήσουμε είναι οι εξής:

Άρνηση ($\neg$)

Σύζευξη ($\wedge$)

Διάζευξη ($\vee$)

Συνεπαγωγή ($\rightarrow$)

Ο τελεστής $\neg$ είναι ο μοναδικός τελεστής μίας θέσης (δηλαδή δέχεται ένα όρισμα) ενώ όλοι οι άλλοι είναι τελεστές δύο θέσεων και τους λέμε για συντομία δυαδικούς τελεστές.

Πλέον διαθέτουμε όλα τα σύμβολα που θα χρησιμοποούμε για να ορίζουμε λέξεις της λογικής.

Ας δούμε τώρα ποιες λέξεις με σύμβολα από το παραπάνω λεξιλόγιο ανήκουν στην γλώσσα με βάση τους επόμενους ορισμούς.

Ορισμός 1. Ένας προτασιακός ατομικός τύπος είναι είτε κάποιο προτασιακό γράμμα είτε μία από τις σταθερές $\top$ και $\bot$. □

Ορισμός 2. Το σύνολο των προτασιακών τύπων που ορίζουν την γλώσσα της προτασιακής λογικής είναι το μικρότερο σύνολο **P** τέτοιο ώστε να ισχύει ότι:

(1) Αν p είναι ένας ατομικός τύπος τότε $p \in P$.

(2) Αν $\varphi \in P$ τότε $\neg\varphi \in P$.

(3) Αν $\odot$ είναι ένας δυαδικός λογικός τελεστής και $\varphi,\psi \in P$ τότε $(\varphi \odot \psi) \in P$. □

Το σύνολο **P** αποτελεί το σύνολο των έγκυρων τύπων για την προτασιακή λογική.

Για παράδειγμα, ο τύπος $\neg((p_1 \wedge p_2) \vee \neg(p_3 \wedge \neg\top))$ είναι ένας έγκυρος προτασιακός τύπος.

Συνήθως, για χάρη ευκολίας, δεν βάζουμε πάντα παρενθέσεις παρά μόνο όταν ο προτασιακός τύπος είναι εξαιρετικά σύνθετος ώστε να μπορούμε να καταλάβουμε τους υπο-τύπους του.

Για παράδειγμα, με βάση τον τυπικό ορισμό πρέπει να γράψουμε $((p \vee q) \wedge t)$ αλλά μπορούμε πιο απλά να γράψουμε $(p \vee q) \wedge t$ αφαιρώντας τις εξωτερικές παρενθέσεις.

1.2 Το θεώρημα Μοναδικής Ανάγνωσης

Το συντακτικό μιας γλώσσας είναι αντικείμενο μελέτης της θεωρίας αυτομάτων. Με βάση τον ορισμό 2, το σύνολο των προτασιακών τύπων είναι μια γλώσσα χωρίς συμφραζόμενα, δηλαδή κάθε τύπος έχει ένα και μόνο νόημα. Αυτό συνοψίζεται στο επόμενο θεώρημα.

Θεώρημα Μοναδικής Ανάγνωσης. Ένας προτασιακός τύπος ανήκει σε ακριβώς μία από τις επόμενες κατηγορίες:

(1) Ατομικός.

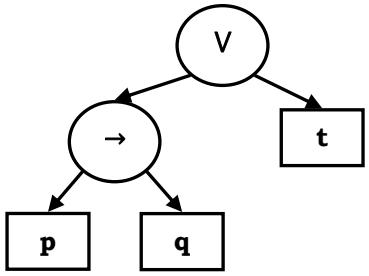
(2) ¬φ, με φ προτασιακό τύπο.

(3) (φ⊙ψ), με ⊙ δυαδικό λογικό τελεστή και φ,ψ προτασιακούς τύπους. □

Το παραπάνω θεώρημα μας δίνει την γραμματική ενός προτασιακού τύπου ως εξής:

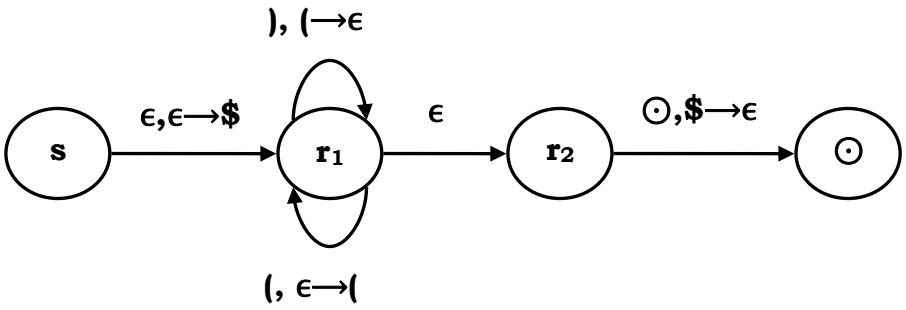
φ : Ατομικός | ¬φ | (φΛφ) | (φVφ) | (φ→φ)
Ατομικός : p,q,r,... | ⊤ | ⊥

Με βάση αυτή την γραμματική μπορούμε να κάνουμε δενδρική ανάλυση ενός προτασιακού τύπου. Για παράδειγμα, ο τύπος (p→q)Vt μπορεί να αναλυθεί με βάση το δένδρο συντακτικής ανάλυσης:



1.2.1 Εύρεση top-level λογικού Συνδετικού

Για την εύρεση του top-level λογικού συνδετικού ενός τύπου δεν απαιτείται πλήρης συντακτική ανάλυση και δημιουργία του αντίστοιχου δένδρου συντακτικής ανάλυσης. Αντίθετα, είναι πιο εύκολο να χρησιμοποιήσουμε ένα απλό αυτόματο στοίβας το οποίο διατρέχει τον τύπο ως εξής:



Η λειτουργία του αυτομάτου με λίγα λόγια είναι η εξής:

(1) Εισάγουμε στην στοίβα το σύμβολο $ για να γνωρίζουμε πότε είναι κενή.

(2) Αν κατά το parsing της φόρμουλας βρούμε μια αριστερή παρένθεση τότε εισάγουμε στην στοίβα το σύμβολο (.

(3) Αντίθετα, αν κατά το parsing της φόρμουλας βρούμε μια δεξιά παρένθεση τότε αφαιρούμε από την στοίβα το σύμβολο (.

(4) Αν βρούμε κάποιο συνδετικό $\odot$ τότε ελέγχουμε αν η στοίβα είναι κενή και αν είναι τότε αυτό είναι το top-level συνδετικό.

Η υλοποίηση του αυτόματου μπορεί να γίνει εύκολα με ένα μετρητή όπου μια εισαγωγή στην στοίβα αντιστοιχεί σε αύξηση του μετρητή και μια αφαίρεση από αυτήν αντιστοιχεί σε μείωση του.

1.3 Σημασιολογία Προτασιακής Λογικής

Η κλασική λογική όπως έχουμε ήδη αναφέρει είναι δίτιμη. Έστω ότι το σύνολο λογικών τιμών είναι το $TR=\{0,1\}$. Τα 0 και 1 είναι δύο διακριτά αντικείμενα στα οποία εμείς θα δώσουμε το νόημα «ψέμα» και «αλήθεια» αντίστοιχα.

Πλέον, κάθε λογικός τελεστής μπορεί να χαρακτηριστεί ως μια απεικόνιση από το σύνολο TR και πάλι στο σύνολο TR ανάλογα με το είδος και την λειτουργία του.

Η σημασιολογία ενός λογικού τελεστή περιγράφεται από τον πίνακα αληθείας του στον οποίο μπορούμε να δούμε το αποτέλεσμα που έχει η επίδραση του με βάση την σημασιολογία που έχει δοθεί στα ορίσματα του.

Για τους τελεστές που μας ενδιαφέρουν έχουμε:

p	$\neg$p
0	1
1	0

p	q	p $\wedge$ q	p $\vee$ q	p $\rightarrow$ q
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

Μπορούμε να φτιάξουμε έναν πίνακα αληθείας για κάθε προτασιακό τύπο και να δούμε πως η τιμή αληθείας του συσχετίζεται με τις τιμές αληθείας των ατομικών τύπων από τους οποίους αποτελείται με βάση πάντα τους λογικούς τελεστές που τους συνδέουν.

Πιο συγκεκριμένα, με βάση τις τιμές αληθείας των προτασιακών γραμμάτων και την σημασιολογία των λογικών τελεστών μπορούμε να υπολογίσουμε την τιμή αληθείας κάθε προτασιακού τύπου.

Ορισμός 3. Μια απόδοση τιμών αληθείας είναι μια απεικόνιση ν από το σύνολο των προτασιακών τύπων στο σύνολο λογικών τιμών TR ως εξής:

(1) $\nu(\bot)=0$ και $\nu(\top)=1$

(2) $\nu(\neg\varphi)=\neg\nu(\varphi)$

(3) $\nu(\varphi\odot\psi)=\nu(\varphi)\odot\nu(\psi)$ □

Για παράδειγμα, ας υποθέσουμε ότι έχουμε τρία προτασιακά γράμμα p,q,r και μια απόδοση τιμών αληθείας ν τέτοια ώστε ν(p)=1, ν(q)=0 και ν(r)=0. Τότε, μπορούμε να υπολογίσουμε την τιμή αληθείας του τύπου φ=¬((p→q)∨r) ως εξής:

ν(φ)= ν(¬((p→q)∨r))
= ¬ν((p→q)∨r)
= ¬[ν(p→q)∨ν(r)]
= ¬[(ν(p)→ν(q))∨ν(r)]
= ¬[(1→0)∨0]
= ¬(0∨0)
= ¬0
= 1

1.4 Ταυτολογίες Και Ικανοποιησιμότητα

Στην προτασιακή λογική, μας ενδιαφέρουν ιδιαίτερα οι προτασιακοί τύποι που είναι αληθείς υπό οποιαδήποτε συνθήκη και ονομάζονται ταυτολογίες.

Ορισμός 4. Ένας προτασιακός τύπος φ είναι ταυτολογία αν ισχύει ότι ν(φ)=1 για κάθε πιθανή απόδοση τιμών αληθείας ν στα προτασιακά γράμματα από τα οποία αποτελείται. Αντίθετα, αν δεν υπάρχει καμία απόδοση τιμών αληθείας ν τέτοια ώστε ν(φ)=1 τότε ο προτασιακός τύπος φ καλείται αντίφαση. □

Για να ελέγξουμε αν ένας προτασιακός τύπος είναι ταυτολογία τότε θα πρέπει να μελετήσουμε την συμπεριφορά του για κάθε πιθανή απόδοση τιμών αληθείας στα προτασιακά γράμματα.

Στην ουσία, αν ο τύπος περιέχει n προτασιακά γράμματα τότε πρέπει να ελέγξουμε συνολικά 2^n διαφορετικές περιπτώσεις. Ένας εύκολος τρόπος είναι με χρήση του πίνακα αληθείας αλλά δεν είναι και τόσο αποδοτικός αφού θα πρέπει να ελέγξουμε όλες τις πιθανές περιπτώσεις και επομένως, η πολυπλοκότητα είναι εκθετική.

Συνήθως μας ενδιαφέρει η συμπεριφορά των τύπων όταν είναι χωρισμένοι σε σύνολα και άρα θέλουμε να δούμε πότε ένα τέτοιο σύνολο είναι ικανοποιήσιμο.

Ορισμός 5. Ένα σύνολο προτασιακών τύπων Σ είναι ικανοποιήσιμο αν υπάρχει μια απόδοση τιμών αληθείας ν έτσι ώστε ν(φ)=1 για κάθε τύπο φ∈Σ. □

Ένας εύκολος τρόπος ελέγχου ικανοποιησιμότητας συνόλου είναι να φτιάξουμε έναν πίνακα αληθείας για κάθε μέλος του και να δούμε αν υπάρχει έστω και μία συγκεκριμένη απόδοση τιμών στα προτασιακά γράμματα για την οποία όλα τα μέλη είναι αληθή. Αυτή η μέθοδος και πάλι δεν είναι αποδοτική καθώς στην χειρότερη περίπτωση θα πρέπει να ελέγξουμε όλες τις γραμμές όλων αυτών των επιμέρων πινάκων αληθείας οι οποίες είναι εκθετικές σε πλήθος.

Υπάρχει μια απλή σύνδεση μεταξύ της έννοιας της ταυτολογίας τύπου και της ικανοποιησιμότητας συνόλου και είναι η εξής:

Ο τύπος φ είναι ταυτολογία αν το σύνολο {¬φ} είναι μη-ικανοποιήσιμο

Με βάση αυτήν την σχέση μπορούμε να χρησιμοποιήσουμε αποδεικτικά συστήματα ώστε να δούμε αν ένας τύπος είναι ταυτολογία ή όχι. Για τον σκοπό αυτό, υποθέτουμε την άρνηση του

τύπου και ανάλογα με τους κανόνες που μας παρέχει το αποδεικτικό σύστημα προσπαθούμε να αποδείξουμε μέσω λογικών συνεπαγωγών τον ατομικό τύπο $\perp$.

1.5 Λογική Συνεπαγωγή

Έστω $\Sigma=\{\varphi_1,\varphi_2,\ldots,\varphi_N\}$ ένα σύνολο από προτασιακούς τύπους και ψ ένας προτασιακός τύπος. Το γεγονός ότι το σύνολο Σ συνεπάγεται λογικά τον τύπο ψ το συμβολίζουμε ως εξής:

$$\boldsymbol{\Sigma \models \psi}$$

Αυτό σημαίνει ότι για κάθε πιθανή απόδοση τιμών αληθείας ν στα προτασιακά γράμματα από τα οποία αποτελούνται οι προτασιακοί τύποι που συνθέτουν το σύνολο Σ πρέπει να ισχύει:

$$\textbf{Αν για κάθε } \boldsymbol{\varphi_i \in \Sigma} \textbf{ ισχύει ότι } \boldsymbol{\nu(\varphi_i)=1} \textbf{ τότε } \boldsymbol{\nu(\psi)=1}$$

Δηλαδή, το σύνολο Σ συνεπάγεται λογικά τον τύπο ψ αν και μόνο αν η λογική διατηρείται.

Για παράδειγμα ισχύει ότι:

$$\mathbf{p \wedge q \models p \vee q}$$

Για να ισχύει η λογική σύζευξη $p \wedge q$ πρέπει να υπάρχει μια απόδοση τιμών αληθείας ν τέτοια ώστε $\nu(p)=1$ και $\nu(q)=1$. Τότε ισχύει $\nu(p \vee q)=\nu(p)$ or $\nu(q)=1$ και άρα η λογική συνεπαγωγή όντως είναι αληθής.

Ένας άλλος τρόπος να σκεφτούμε την λογική συνεπαγωγή είναι η εξής: Αν το σύνολο Σ είναι ικανοποιήσιμο τότε το ίδιο πρέπει να ισχύει και για το σύνολο $\Sigma \cup \{\psi\}$.

Αυτό μπορούμε αλλιώς να το εκφράσουμε και με αντιθετοαντιστροφή ως εξής:

$$\boldsymbol{\Sigma \models \psi \;\Leftrightarrow\; \Sigma \cup \{\neg\psi\} \models \textbf{false}}$$

Αν το σύνολο Σ είναι κενό τότε έχουμε ότι $\boldsymbol{\models \psi}$ τότε ο τύπος ψ αποτελεί μια ταυτολογία καθώς ισχύει χωρίς καμία υπόθεση. Μάλιστα, μπορούμε να δούμε ότι για να είναι ο ψ ταυτολογία, πρέπει $\boldsymbol{\{\neg\psi\} \models \textbf{false}}$ το οποίο μας δείχνει την σχέση ταυτολογίας και ικανοποιησιμότητας που είχαμε παρατηρήσει.

Αντίθετα, η λογική συνεπαγωγή δεν ισχύει αν υπάρχει μια απόδοση τιμών αληθείας ν τέτοια ώστε να ισχύει:

$$\textbf{Αν για κάθε } \boldsymbol{\varphi_i \in \Sigma} \textbf{ ισχύει ότι } \boldsymbol{\nu(\varphi_i)=1} \textbf{ τότε } \boldsymbol{\nu(\psi)=0}$$

Αυτό το γεγονός το συμβολίζουμε ως εξής:

$$\boldsymbol{\Sigma \not\models \psi}$$

Στην ουσία, για να αποδείξουμε ότι ένα σύνολο Σ δεν συνεπάγεται λογικά έναν τύπο ψ, αρκεί να βρούμε ένα αντιπαράδειγμα.

Για παράδειγμα ισχύει ότι:

$$\mathbf{p \vee q \not\models p \wedge q}$$

Για να ισχύει η λογική διάζευξη $p \vee q$ πρέπει να υπάρχει μια απόδοση τιμών αληθείας ν τέτοια ώστε $\nu(p)=1$ ή $\nu(q)=1$. Μια τέτοια απόδοση τιμών αληθείας ν είναι και εκείνη όπου $\nu(p)=1$ και $\nu(q)=0$ και άρα ισχύει $\nu(p \wedge q)=\nu(p)$ and $\nu(q)=0$.

1.6 Συμβολικές Αποδείξεις

Έστω $\Sigma = \{\varphi_1,\varphi_2,\ldots,\varphi_N\}$ ένα σύνολο προτασιακών τύπων το οποίο θεωρούμε ικανοποιήσιμο και αποκαλούμε σύνολο αρχικών υποθέσεων (ή βάση γνώσης στην τεχνητή νοημοσύνη).

Το γεγονός ότι ένας προτασιακός τύπος ψ μπορεί να αποδειχτεί με βάση αυτό το σύνολο Σ το συμβολίζουμε ως εξής:

$$\boldsymbol{\Sigma \vdash \psi}$$

Η απόδειξη ενός τύπου ψ με βάση ένα σύνολο αρχικών υποθέσεων Σ είναι μια διαδικασία με την οποία μέσω του Σ και ενός συνόλου μηχανισμών συλλογισμού μπορούμε να καταλήξουμε στο συμπέρασμα ότι όντως ισχύει ο τύπος ψ.
Το σύνολο μηχανισμών συλλογισμού συνήθως είναι ένα σύνολο επιτρεπτών κανόνων οι οποίοι ορίζουν ένα αποδεικτικό σύστημα. Κάθε αποδεικτικό σύστημα θα πρέπει να έχει την ιδιότητα της ορθότητας και της πληρότητας ώστε οι αποδείξεις του να είναι έγκυρες.

Ιδιότητα Ορθότητας. Αν $\boldsymbol{\Sigma \vdash \psi}$ τότε $\boldsymbol{\Sigma \models \psi}$.

Δηλαδή, αν ένας τύπος ψ μπορεί να αποδειχτεί από ένα σύνολο υποθέσεων Σ τότε θα πρέπει να συνεπάγεται και λογικά από αυτό.

Ιδιότητα Πληρότητας. Αν $\boldsymbol{\Sigma \models \psi}$ τότε $\boldsymbol{\Sigma \vdash \psi}$.

Δηλαδή, κάθε φόρμουλα ψ που συνεπάγεται λογικά από ένα σύνολο υποθέσεων Σ πρέπει να μπορεί και να αποδειχτεί από αυτό.

Αν ένα αποδεικτικό σύστημα είναι ορθό και πλήρες τότε μπορούμε να το χρησιμοποιήσουμε ώστε να αποδείξουμε λογικές συνεπαγωγές.

1.7 Το Αποδεικτικό Σύστημα Natural Deduction

1.7.1 Μορφή Των Natural Deduction Αποδείξεων

Μια απόδειξη στο Natural Deduction (ND) είναι μια ακολουθία τύπων οποιουδήποτε μήκους με τους εξής περιορισμούς: Οι πρώτοι n τύποι συνιστούν το σύνολο αρχικών υποθέσεων και ο τελευταίος τύπος είναι το τελικό συμπέρασμα. Οι ενδιάμεσοι τύποι πρέπει να προκύπτουν από τους προηγούμενους με βάση τους κανόνες του συστήματος.

Δηλαδή, μια απόδειξη του ND έχει την εξής γενική μορφή:

$$
\begin{array}{l}
\mathbf{1.}\ \varphi_1 \\
\mathbf{2.}\ \varphi_2 \\
\quad\vdots \\
\mathbf{n.}\ \varphi_n \\
\quad\vdots \\
\mathbf{j.}\ \varphi_j \\
\quad\vdots \\
\mathbf{m.}\ \psi
\end{array}
$$

Επομένως, μια απόδειξη είναι μια ακολουθία γραμμών κάθε μία από τις οποίες περιέχει έναν αριθμό που την διακρίνει από τις υπόλοιπες και έναν τύπο ο οποίος πρέπει να είναι έγκυρος με βάση τις αρχικές υποθέσεις.

1.7.2 Σχολιασμός Αποδείξεων

Σε κάθε γραμμή της απόδειξης μπορούμε να συμπεριλάβουμε και ένα σχόλιο εξήγησης του πως προέκυψε ο τύπος που περιέχει η συγκεκριμένη γραμμή. Το σχόλιο μπορεί να είναι είτε ότι ο τύπος είναι υπόθεση είτε ότι ο τύπος προέκυψε από την εφαρμογή ενός κανόνα σε ένα συγκεκριμένο πλήθος προηγούμενων γραμμών.

1.7.3 Υπο-Αποδείξεις

Μια υπο-απόδειξη είναι μια απόδειξη η οποία γίνεται στα πλαίσια μιας άλλης απόδειξης και παίζει βοηθητικό ρόλο. Ξεκινά με υποθέσεις οι οποίες δεν είναι γνωστό το αν είναι αληθείς με σκοπό να δει απλά που καταλήγουν και αν μπορεί να βγει κάποιο χρήσιμο συμπέρασμα από αυτήν την διαδικασία το οποίο θα χρησιμοποιηθεί στην κύρια απόδειξη.

Μια υπο-αποδείξη εμφανίζεται μέσα στην κύρια απόδειξη ως ένα block το οποίο περιέχει ένα σύνολο νέων υποθέσεων και ένα σύνολο συμπερασμάτων που εξάγονται με βάση αυτές τις νέες υποθέσεις.

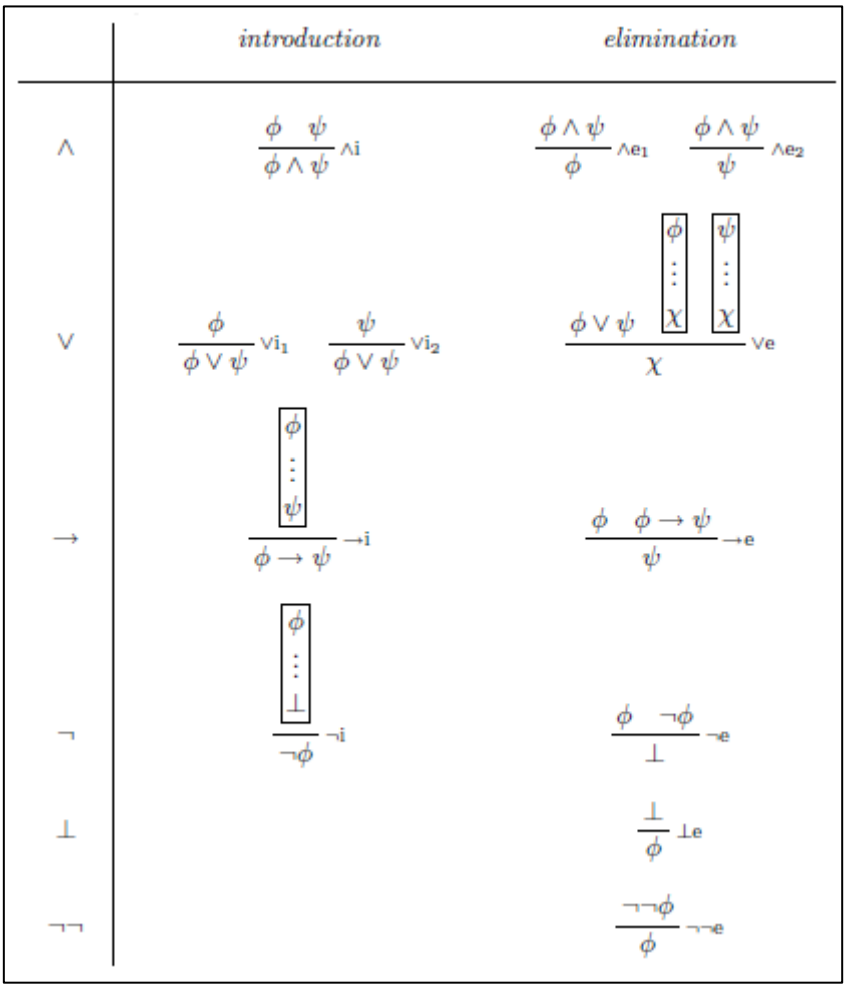
Μια γραμμή μέσα στην υπο-απόδειξη μπορεί να χρησιμοποιήσει οποιαδήποτε γραμμή μέσα από την υπο-απόδειξη αλλά και από ιεραρχικά υψηλότερα επίπεδα. Αντίθετα, μια γραμμή η οποία ανήκει σε μεγαλύτερο επίπεδο (δηλαδή είναι έξω από μια υπο-απόδειξη) δεν μπορεί να χρησιμοποιεί γραμμές της υπο-απόδειξης αφού αυτές περιέχουν τύπους οι οποίοι μπορεί και να μην ισχύουν.

1.7.4 Κανόνες Του Συστήματος

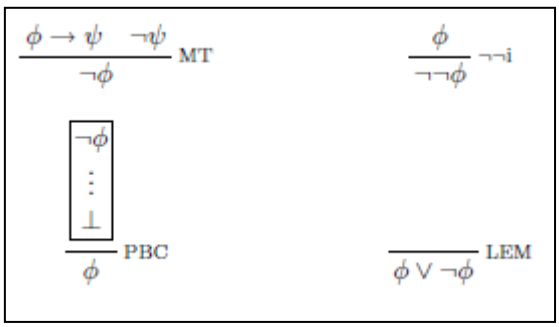
Οι κανόνες του συστήματος ND χωρίζονται σε κανόνες introduction (σύνθεση τύπων για την απόδειξη υπερ-τύπου) και elimination (διάσπαση τύπου για την απόδειξη υπο-τύπου).

Ακόμη, μπορούμε να κατηγοριοποιήσουμε τους κανόνες σε βασικούς και παραγόμενους. Οι βασικοί κανόνες είναι μέρος του συστήματος ενώ οι παραγόμενοι δεν είναι αλλά μπορούν να αποδειχτούν άμεσα από τους βασικούς και συνεπώς, μπορούμε να τους χρησιμοποιούμε στις αποδείξεις μας.

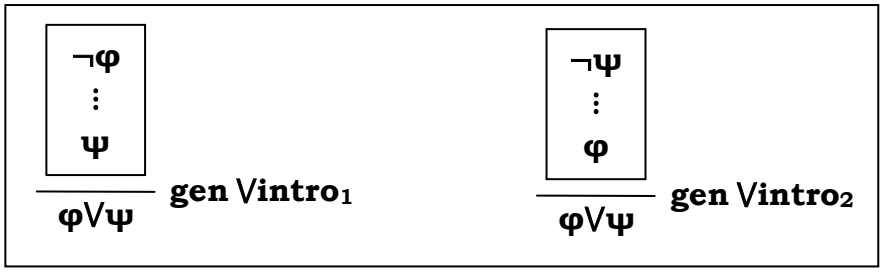
Οι βασικοί κανόνες του συστήματος είναι οι εξής:



Οι παραγόμενοι κανόνες του συστήματος είναι οι εξής:



Ένας τελευταίος και πολύ χρήσιμος κανόνας του συστήματος είναι ο γενικός Vintro ο οποίος ορίζεται ως εξής:



1.7.5 Ένα Παράδειγμα

Το ND μας παρέχει μόνο με τους αναγκαίους, ως προς την πληρότητα, κανόνες. Πράγματι, το σύστημα δεν μας δίνει βασικό κανόνα ούτε καν για την ιδιότητα της αντιμεταθετικότητας την οποία μπορούμε να εκφράσουμε σε μορφή απόδειξης ως εξής: (p∧q)∧r ⊢ p∧(q∧r).

Ωστόσο, χρησιμοποιώντας τους βασικούς κανόνες, μπορούμε να βρούμε μια απόδειξη:

1.	**(p∧q)∧r**	
2.	**p∧q**	**∧elim 1**
3.	**r**	**∧elim 1**
4.	**p**	**∧elim 2**
5.	**q**	**∧elim 2**
6.	**(q∧r)**	**∧intro 3,5**
7.	**p∧(q∧r)**	**∧intro 4,6**

Απόδειξη (p∧q)∧r ⊢ p∧(q∧r)

1.7.6 Ορθότητα Και Πληρότητα Του Natural Deduction

Θεώρημα Ορθότητας ND. Αν $\Sigma \vdash_{ND} \psi$ τότε $\Sigma \models \psi$.

Θεώρημα Πληρότητας ND. Αν $\Sigma \models \psi$ τότε $\Sigma \vdash_{ND} \psi$.

Με τα δύο αυτά θεωρήματα μπορούμε να δούμε ότι μια απόδειξη του ND είναι έγκυρη καθώς ισχύει η αντίστοιχη λογική συνεπαγωγή και επίσης, μέσω του ND μπορούμε να βρούμε μια απόδειξη για κάθε έγκυρη λογική συνεπαγωγή.

Αποδείξεις για τα θεωρήματα μπορούν να βρεθούν στο βιβλίο Logic In Computer Science των Michael Huth και Mark Ryan όπου γίνεται παρουσίαση του συστήματος ND και των κανόνων του.

1.8 Το Πρόβλημα Ικανοποιησιμότητας - SAT

1.8.1 Κανονική Συζευκτική Μορφή

Ορισμός 6. Ένας τύπος φ είναι σε κανονική συζευκτική μορφή (Conjunctive Normal Form-CNF) αν είναι η σύζευξη ενός συνόλου προτάσεων (clauses) οι οποίες είναι διαζεύξεις όρων (literals) οι οποίοι είναι προτασιακά γράμματα σε κανονική ή συμπληρωματική μορφή. □

Για παράδειγμα, ο τύπος φ=(¬pVq) Λ (tV¬q) είναι σε κανονική συζευκτική μορφή καθώς είναι η σύζευξη των προτάσεων ¬pΛq και tΛ¬q οι οποίες με την σειρά τους αποτελούνται από τους όρους ¬p, q, t και ¬q.

1.8.2 Το SAT

Το πρόβλημα ικανοποιησιμότητας λογικού τύπου είναι ένα εξαιρετικής σημασίας πρόβλημα στην επιστήμη των υπολογιστών και το πρώτο που αποδείχτηκε ότι είναι **NP**-πλήρες.

Το πρόβλημα αφορά στην εύρεση μιας ανάθεσης ν λογικών τιμών στα προτασιακά γράμματα ενός λογικού τύπου σε CNF έτσι ώστε ο τύπος να ικανοποιείται.

Το k-SAT είναι μια απο τις βασικές παραλλαγές του SAT όπου ο τύπος πρέπει να είναι σε k-CNF, δηλαδή, οι προτάσεις από τις οποίες αποτελείται θα πρέπει να έχουν ακριβώς k όρους. Το 2-SAT είναι το μοναδικό στιγμιότυπο αυτής της κλάσης προβλημάτων το οποίο μπορεί να επιλυθεί σε πολυωνυμικό χρόνο και επομένως ανήκει στην κλάση **P**.

Μια άλλη παραλλαγή του SAT είναι το MAX-SAT στο οποίο στόχος είναι η εύρεση ανάθεσης τέτοιας ώστε να μεγιστοποιηθεί το πλήθος των προτάσεων που ικανοποιούνται. Προφανώς, το MAX-SAT είναι το SAT αν ικανοποιούνται όλες οι προτάσεις και είναι ένα πρόβλημα το οποίο επιδέχεται σχετικά καλή προσέγγιση μέσω πιθανοτικών αλγορίθμων.

1.8.3 Σχέση SAT Και Natural Deduction

Το αποδεικτικό σύστημα ND μπορεί να συσχετιστεί με το SAT με δύο τρόπους.

Ο πρώτος είναι ότι μπορεί να ελέγξει αν μια συγκεκριμένη ανάθεση λογικών τιμών ικανοποιεί όντως έναν λογικό τύπο βρίσκοντας την αντίστοιχη απόδειξη. Η ανάθεση μπορεί να εκφραστεί ως η σύζευξη των προτασιακών γραμμάτων έτσι ώστε αν στο προτασιακό γράμμα p έχει δοθεί η τιμή ν(p)=1 τότε αυτό εμφανίζεται σε κανονική μορφή ενώ αν του έχει δοθεί η τιμή ν(p)=0 τότε εμφανίζεται σε συμπληρωματική μορφή.

Επομένως, η ανάθεση είναι το σύνολο αρχικών υποθέσεων της απόδειξης και ο στόχος είναι ο λογικός τύπος τον οποίο ελέγχουμε αν ικανοποιεί η συγκεκριμένη ανάθεση.

Για παράδειγμα, έστω ότι θέλουμε να ελέγξουμε αν η ανάθεση ν(p)=1,ν(q)=0 ικανοποιεί τον λογικό τύπο φ=(pVq) Λ (pV¬q). Τότε, η ζητούμενη απόδειξη είναι η p,¬q ⊢ (pVq) Λ (pV¬q).

Για αυτήν την ζητούμενη απόδειξη μπορούμε να εφαρμόσουμε τους κανόνες του ND και αν όντως μπορούμε να βρούμε μια απόδειξη τότε έχουμε αποδείξει ότι η συγκεκριμένη ανάθεση ικανοποιεί τον λογικό τύπο.

Το ND συσχετίζεται περισσότερο με το co-SAT παρά με το ίδιο το SAT. Πράγματι, μπορούμε να φτιάξουμε μια ζητούμενη απόδειξη η οποία έχει ως αρχική υπόθεση έναν λογικό τύπο και στόχο το ⊥.

Η εφαρμογή των κανόνων του ND σε μια τέτοια απόδειξη θα έχει ως αποτέλεσμα την τυφλή αναζήτηση του ⊥. Δηλαδή, γίνεται προσπάθεια απόδειξης ότι ο δεδομένος λογικός τύπος δεν μπορεί να ικανοποιηθεί ποτέ.

Αν βρεθεί μια κατάλληλη απόδειξη τότε έχουμε ένα πιστοποιητικό ότι ο λογικός τύπος αυτός δεν μπορεί να ικανοποιηθεί και επομένως, μπορούμε να απαντήσουμε αρνητικά στο SAT για τον συγκεκριμένο τύπο.

Επειδή ο λογικός τύπος είναι σε CNF, ο βασικός κανόνας του ND που μπορεί να εφαρμοστεί είναι ο Velim μέσω του οποίου θα ξεκινήσει ένα πλήθος εμφωλευμένων υπο-αποδείξεων που στην ουσία είναι μια προσπάθεια ελέγχου κάθε πιθανής (από τις 2^n) ανάθεσεις λογικών τιμών στα προτασιακά γράμματα με στόχο πάντα την απόδειξη μη-ικανοποιησιμότητας.

1.9 Σύνολα Hintikka

Ορισμός 7. Ένα σύνολο προτασιακών τύπων Η είναι σύνολο Hintikka αν έχει τις παρακάτω ιδιότητες.

(1) Για κάθε προτασιακό γράμμα p: $p \in H \Rightarrow \neg p \notin H$, $\neg p \in H \Rightarrow p \notin H$.

(2) $\bot \notin H$ και $\neg \top \notin H$.

(3) $\neg\neg\varphi \in H \Rightarrow \varphi \in H$.

(4) $\varphi \wedge \psi \in H \Rightarrow \varphi \in H$ και $\psi \in H$.

(5) $\neg(\varphi \vee \psi) \in H \Rightarrow \neg\varphi \in H$ και $\neg\psi \in H$.

(6) $\varphi \vee \psi \in H \Rightarrow \varphi \in H$ ή $\psi \in H$.

(7) $\neg(\varphi \wedge \psi) \in H \Rightarrow \neg\varphi \in H$ ή $\neg\psi \in H$.

(8) $\varphi \rightarrow \psi \in H \Rightarrow \neg\varphi \in H$ ή $\psi \in H$.

(9) $\neg(\varphi \rightarrow \psi) \in H \Rightarrow \varphi \in H$ και $\neg\psi \in H$. □

Τα σύνολα Hintikka είναι εξαιρετικά σημαντικά καθώς ισχύει το εξής λήμμα:

Λήμμα Hintikka. Κάθε προτασιακό σύνολο Hintikka είναι ικανοποιήσιμο. □

Αν ένα σύνολο προτασιακών τύπων Σ μπορεί να επεκταθεί με κατάλληλο τρόπο σε ένα σύνολο Hintikka τότε το Σ είναι, με βάση το λήμμα Hintikka, ικανοποιήσιμο. Αυτό είναι χρήσιμο για την απόδειξη της πληρότητας ενός αποδεικτικού συστήματος.

Πράγματι, αν η εφαρμογή των κανόνων ενός αποδεικτικού συστήματος σε ένα σύνολο Σ που αποτελείται από αρχικές υποθέσεις έχει ως αποτέλεσμα την επέκταση του Σ σε ένα σύνολο Hintikka τότε το σύστημα μπορεί να αποδείξει κάθε τύπο ο οποίος συνεπάγεται λογικά από το Σ.

Περισσότερες λεπτομέρειες σχετικά με τα σύνολα Hintikka καθώς και μια απόδειξη του λήμματος Hintikka μπορούν να βρεθούν στο βιβλίο First-Order Logic And Automated Theorem Proving του Melvin Fitting.

Κεφάλαιο 2
Ένας Αλγόριθμος Εύρεσης Φυσικών Αποδείξεων

2.1 Βασική Ιδέα Του Αλγορίθμου

Έστω ένα σύνολο αρχικών υποθέσεων Σ και ένας στόχος G. Σκοπός του αλγορίθμου είναι η εύρεση μιας απόδειξης Σ⊢G βασιζόμενος στους κανόνες του αποδεικτικού συστήματος ND.

Ο αλγόριθμος αποτελείται από δύο φάσεις. Στην πρώτη φάση κάνει μια καθοδηγούμενη από τον στόχο αναζήτηση όπου αναλύει τον στόχο και θέτει υπο-στόχους ώστε να απλοποιήσει την απόδειξη. Στην δεύτερη φάση κάνει μια αναζήτηση ωμής βίας όπου προσπαθεί να συνδυάσει τις γραμμές της μέχρι-τώρα απόδειξης με σκοπό την εφαρμογή των κανόνων του ND για την παραγωγή νέων τύπων μέχρι τελικά να παραχθεί ο απλοποιημένος στόχος που τέθηκε κατά την πρώτη φάση.

Πιο συγκεκριμένα έχουμε τα εξής:

Στην πρώτη φάση που την ονομάζουμε φάση Backward Reasoning (BR-φάση) ο αλγόριθμος αναλύει τον στόχο και με βάση το top-level λογικό συνδετικό του προσπαθεί να θέσει κάποιον πιο απλό στόχο ο οποίος θα είναι στην ουσία υπο-τύπος του αρχικού στόχου κάνοντας μαζί και υποθέσεις αν κρίνεται απαραίτητο.

Αυτή η φάση μπορεί να γίνει είτε με αναδρομή είτε με την χρήση μιας στοίβας όπου αποθηκεύονται οι στόχοι του τίθονται κατά την διάρκεια της ανάλυσης. Η φάση ολοκληρώνεται όταν φτάσουμε σε στόχο που είναι ατομικός, δηλαδή είτε προτασιακό γράμμα είτε κάποια από τις σταθερές ⊤ και ⊥.

Προφανώς, στην περίπτωση που κάποιος στόχος είναι το ⊤ τότε εύκολα καταλαβαίνουμε ότι η απόδειξη ολοκληρώθηκε καθώς το true είναι πάντα και χωρίς καμία υπόθεση αληθές.

Στην δεύτερη φάση που την ονομάζουμε φάση Forward Reasoning (FR-φάση) ο αλγόριθμος προσπαθεί να συνδυάσει τους τύπους του συνόλου Σ και τις υποθέσεις που ίσως και να έγιναν κατά την BR-φάση ώστε να παράξει νέους τύπους με σκοπό φυσικά την απόδειξη του τελικού απλού στόχου που είναι στην κορυφή της στοίβας.

Κατά την FR-φάση οι γραμμές της μέχρι-τώρα απόδειξης που περιέχουν τους τύπους που είναι έγκυροι (με βάση τις υποθέσεις) συνδυάζονται με συγκεκριμένη σειρά και ο αλγόριθμος προσπαθεί να εφαρμόσει τους κανόνες του ND με συγκεκριμένο τρόπο ώστε να αποφύγει τις όποιες περιττές παραγωγές.

Όταν και αν επιτευχθεί ο απλός στόχος κατά την FR-φάση τότε άμεσα έχουν επιτευχθεί και οι υπόλοιποι υπερ-στόχοι και ένας ένας προστίθενται στην τελική απόδειξη.

Για να είναι πλήρης μιας απόδειξη θα πρέπει να υπάρχουν σχόλια σε κάθε γραμμή της ώστε να είναι γνωστό το ποιος κανόνας χρησιμοποιήθηκε και σε ποιες γραμμές για την παραγωγή της. Αυτά τα σχόλια γράφονται κατά την παραγωγή αλλά και κατά την επιστροφή της αναδρομής όσον αφορά τους υπερ-στόχους.

Τα σχόλια είναι αρκετά σημαντικά καθώς μπορούν να χρησιμοποιηθούν για την απομάκρυνση των περιττών γραμμών. Πράγματι, η FR-φάση είναι μια τυφλή αναζήτηση και έτσι μπορεί να έχουν παραχθεί τύποι που στην πραγματικότητα να μην χρειάζονται για την απόδειξη του στόχου και άρα καλό θα ήταν να απομακρυνθούν.

2.2 BR-Φάση

Κατά την BR-φάση ο αλγόριθμος κάνει συντακτική ανάλυση του προτασιακού τύπου ο οποίος είναι ο τρέχων στόχος με βάση την γραμματική της προτασιακής λογικής ώστε να βρει το top-level συνδετικό του και ανάλογα να εφαρμόσει κάποιον από τους BR-κανόνες του ND που θα παρουσιάσουμε στην συνέχεια ως στοιχειώδεις αλγοριθμικές διαδικασίες.

Οι BR-κανόνες του ND είναι οι κανόνες introduction οι οποίοι στην ουσία δημιουργούν όλο και πιο μεγάλους τύπους και έτσι, με κάποιον τρόπο μπορούν και μας λένε το τι χρειάζονται ώστε να παραχθούν. Αυτές τις απαιτήσεις μπορούμε να τις εξάγουμε κατά την BR-φάση και να τις ενσωματώσουμε ως υποθέσεις και υπο-στόχους στην απόδειξη μειώνοντας κατά αυτόν τον τρόπο τον φόρτο εργασίας.

Έτσι, κάθε νέος στόχος αναλύεται και με βάση αυτόν δημιουργείται ένας νέος στόχος μέχρι ο τελευταίος στόχος να είναι ένας ατομικός τύπος οπότε και η BR-φάση τερματίζει. Η επιλογή μας να τερματίσουμε την BR-φάση όταν ένας τύπος γίνει ατομικός τύπος και όχι η σταθερά ⊥ γίνεται για το ενδεχόμενο ο ατομικός αυτός τύπος να μπορεί να παραχθεί σχετικά γρήγορα.

Η BR-φάση μπορεί να υλοποιηθεί είτε με χρήση μιας στοίβας όπου αποθηκεύουμε τους υπο-στόχους όταν δημιουργούνται είτε με αναδρομή καθώς όταν θέτουμε έναν στόχο, αναγάγουμε την αρχική μας απόδειξη σε μια άλλη. Μπορούμε να συνδυάσουμε τους δύο αυτούς τρόπους καθώς κάθε ένας μας εξυπηρετεί σε διαφορετικές περιπτώσεις όπως θα δούμε παρακάτω.

2.2.1 Κανόνας IMPLIES-Introduction

Αν το top-level συνδετικό του στόχου είναι το implies (→) τότε η απόδειξη έχει την μορφή:

$$\mathbf{\Sigma \vdash \varphi \rightarrow \psi}$$

Με βάση τον κανόνα →intro για να έχουμε ως αποτέλεσμα έναν τύπο φ→ψ θα πρέπει να υποθέσουμε τον υπο-τύπο φ και να θέσουμε ως νέο στόχο τον υπο-τύπο ψ. Άρα:

Η απόδειξη $\mathbf{\Sigma \vdash \varphi \rightarrow \psi}$ ανάγεται στην απόδειξη $\mathbf{\Sigma, \varphi \vdash \psi}$

Ο τύπος φ θα γραφεί στην απόδειξη ως νέα υπόθεση ενώ ο τύπος ψ θα εισαχθεί στην στοίβα στόχων και άρα θα γίνει ο τρέχων στόχος ο οποίος πρέπει σε επόμενο βήμα να αναλυθεί κατά τον ίδιο τρόπο εκτός και αν είναι ατομικός τύπος οπότε και η BR-φάση τερματίζει.

2.2.2 Κανόνας NOT-Introduction

Αν το top-level συνδετικό του στόχου είναι το not (¬) τότε η απόδειξη έχει την μορφή:

$$\mathbf{\Sigma \vdash \neg \varphi}$$

Με βάση τον κανόνα ¬intro για να έχουμε ως αποτέλεσμα έναν τύπο ¬φ θα πρέπει να υποθέσουμε τον υπο-τύπο φ και να θέσουμε ως νέο στόχο το bottom ⊥. Άρα:

Η απόδειξη $\mathbf{\Sigma \vdash \neg \varphi}$ ανάγεται στην απόδειξη $\mathbf{\Sigma, \varphi \vdash \bot}$

Ο τύπος φ θα γραφεί στην απόδειξη ως νέα υπόθεση ενώ το bottom ⊥ θα εισαχθεί στην στοίβα στόχων, θα γίνει ο τρέχων στόχος και η BR-φάση θα τερματίσει καθώς έχουμε φτάσει σε έναν ατομικό τύπο.

2.2.3 Κανόνας OR-Introduction

Αν το top-level συνδετικό του στόχου είναι το or (∨) τότε η απόδειξη έχει την μορφή:

$$\Sigma \vdash \varphi \vee \psi$$

Με βάση τον γενικό κανόνα ∨intro για να έχουμε ως αποτέλεσμα τον τύπο φ∨ψ μπορούμε να κάνουμε ένα από τα εξής δύο πράγματα:

(1) Υποθέτουμε τον υπο-τύπο ¬φ και θέτουμε ως νέο στόχο τον υπο-τύπο ψ.

(2) Υποθέτουμε τον υπο-τύπο ¬ψ και θέτουμε ως νέο στόχο τον υπο-τύπο φ.

Με βάση αυτές τις δύο πιθανές εκδοχές μπορούμε να πούμε ότι

Η απόδειξη $\Sigma \vdash \varphi \vee \psi$ ανάγεται
είτε στην απόδειξη $\Sigma, \neg\varphi \vdash \psi$ είτε στην απόδειξη $\Sigma, \neg\psi \vdash \varphi$

Στην πρώτη περίπτωση, ο τύπος ¬φ θα γραφεί στην απόδειξη ως νέα υπόθεση και ο τύπος ψ θα εισαχθεί στην στοίβα στόχων και θα γίνει ο νέος στόχος. Στην δεύτερη περίπτωση, ο τύπος ¬ψ θα γραφεί στην απόδειξη ως νέα υπόθεση και ο τύπος φ θα εισαχθεί στην στοίβα στόχων και θα γίνει ο νέος στόχος.

Η επιλογή μας για το ποιά από τις δύο υπο-αποδείξεις θα κάνουμε είναι απλή. Οποιαδήποτε από τις αποδείξεις αυτές πρέπει να μας οδηγήσει σε σωστό αποτέλεσμα αφού αν αποτύχει η μία τότε σίγουρα θα αποτύχει και η άλλη. Άρα, απλά επιλέγουμε την πρώτη.

2.2.4 Κανόνας AND-Introduction

Αν το top-level συνδετικό του στόχου είναι το and (∧) τότε η απόδειξη έχει την μορφή:

$$\Sigma \vdash \varphi \wedge \psi$$

Με βάση τον κανόνα ∧intro για να έχουμε ως αποτέλεσμα έναν τύπο φ∧ψ θα πρέπει να απόδείξουμε τόσο τον υπο-τύπο φ όσο και τον υπο-τύπο ψ με βάση τις ίδιες υποθέσεις.

Η απόδειξη $\Sigma \vdash \varphi \wedge \psi$ ανάγεται στις αποδείξεις $\Sigma \vdash \varphi$ και $\Sigma \vdash \psi$

Οι τύποι φ και ψ εισάγονται μαζί στην στοίβα στόχων και πρέπει να επιτευχθούν και οι δύο για να μπορέσει να τερματίσει η απόδειξη.

Επειδή οι δύο νέοι στόχοι φ και ψ μπορεί να χρειαστούν περαιτέρω ανάλυση, η απόδειξη θα πρέπει να χωριστεί αναδρομικά σε δύο υπο-αποδείξεις οι οποίες πρεπει να γίνουν και οι δύο. Αν έστω και μία από αυτές αποτύχει τότε αποτυγχάνει και η αρχική απόδειξη.

Δηλαδή, στο σημείο αυτό θα γίνει αναδρομή και δεν θα χρησιμοποιηθεί η στοίβα στόχων που χρησιμοποιείται μέχρι αυτή την στιγμή. Στην ουσία, μέσω της αναδρομής, θα δημιουργηθούν δύο νέες στοίβες στόχων στις κορυφές των οποίων θα μπουν οι δύο νέοι διακριτοί στόχοι που πρέπει να επιτευχθούν και οι δύο ώστε να μπορέσει να επιτευχθεί και η αρχική απόδειξη.

Με αυτόν τον τρόπο αποφεύγουμε την υλοποίηση μιας πολύπλοκης δομής LIFO η οποία δεν θα έχει σταθερό αριθμό αντικειμένων σε κάθε θέση της καθώς οι στόχοι αυξάνουν. Άλλωστε, ενδέχεται να αναλυθεί περαιτέρω μόνο ο ένας από τους δύο νέους στόχους με αποτέλεσμα να γίνει πολύ δύσκολη η διαχείριση μιας τέτοιας δομής.

2.2.5 Εξαίρεση: Κανόνας PBC

Ο κανόνας PBC χρησιμοποιείται μεν με BR-τρόπο αλλά δεν εφαρμόζεται κατά την BR-φάση. Χρησιμοποιείται κατά την διάρκεια της FR-φάσης όταν η απόδειξη έχει φτάσει σε ένα σημείο όπου δεν μπορεί να γίνει άλλος συνδυασμός γραμμών.

Ο κανόνας PBC προσφέρει μια διέξοδο στην απόδειξη στην περίπτωση που ο στόχος δεν είναι το bottom ⊥ με την εισαγωγή μιας νέας υπόθεσης και την αλλαγή του στόχου.

Αν ο στόχος είναι ένα προτασιακό γράμμα p τότε με βάση τον κανόνα PBC αρκεί να κάνουμε ως νέα υπόθεση την άρνηση του (¬p) και να προσπαθήσουμε να αποδείξουμε το bottom ⊥. Δηλαδή:

Η απόδειξη $\mathbf{\Sigma \vdash p}$ ανάγεται στην απόδειξη $\mathbf{\Sigma, \neg p \vdash \bot}$

Ο τύπος ¬p γράφεται στην απόδειξη ως νέα υπόθεση και το ⊥ μπαίνει στην στοίβα στόχων.

Αν ο κανόνας PBC εφαρμοζόταν κατά την BR-φάση τότε ενδέχεται να κάναμε πιο πολύπλοκη την απόδειξη από ότι πραγματικά χρειαζόταν αφού το προτασιακό γράμμα που είναι ο στόχος ίσως να είναι στο σύνολο αρχικών υποθέσεων ή να μπορεί να παραχθεί εύκολα από αυτό.

2.2.6 Διαδικασία Σχολιασμού Στόχων

Όταν δημιουργούμε μια απόδειξη πρέπει σε κάθε γραμμή να γράφουμε και από ένα σχόλιο που να μας λέει ποιος κανόνας χρησιμοποιήθηκε και σε ποιες γραμμές ώστε να παραχθεί ο τύπος της γραμμής που είμαστε.

Για να έχουμε καλό σχολιασμό των στόχων που τίθονται κατά την διάρκεια της BR-φάσης θα πρέπει να ακολουθήσουμε μια διαδικασία όπου όταν ένας νέος στόχος εισάγεται στην στοίβα πρέπει να σχολιάζει πως ο υπερ-στόχος θα μπορέσει να παραχθεί από αυτόν και την υπόθεση που ίσως έγινε.

Έστω ότι ο τελευταίος στόχος είναι της μορφής φ⊙ψ και ότι πρέπει να γίνει η υπόθεση x και να τεθεί ο στόχος y με βάση κάποιον κανόνα K. Αν έχουμε N υποθέσεις στην απόδειξη τότε η νέα υπόθεση x θα πρέπει να γραφεί στην (N+1)-οστή γραμμή και έτσι άμεσα γνωρίζουμε ότι ο στόχος φ⊙ψ θα παραχθεί με την εφαρμογή του κανόνα K στην γραμμή N+1.

Ακόμη, για να είναι πλήρης ο σχολιασμός απαιτείται και η γραμμή όπου γίνεται επιτυχής και ο στόχος y αλλά αυτό δεν είναι ακόμα γνωστό και θα πρέπει να συμπληρωθεί όταν προστεθεί στην απόδειξη κατά την FR-φάση.

Πιο συγκεκριμένα, όταν ο τελευταίος στόχος γραφεί στην απόδειξη μαθαίνουμε την γραμμή στην οποία υπάρχει και έτσι, παίρνουμε τον επόμενο στόχο από την στοίβα, ενημερώνουμε το σχόλιο του ως προς το ποια επιπλέον γραμμή χρησιμοποιείται για την απόδειξη του και στην συνέχεια κάνουμε επαναληπτικά το ίδιο και για τους επόμενους στόχους που υπάρχουν στην στοίβα μέχρι αυτή τελικά να μείνει κενή και άρα, να έχουν γραφεί όλοι στην απόδειξη.

Σημαντική είναι η παρατήρηση του γεγονότος ότι λόγω της ύπαρξης του κανόνα Λelim, μπορεί να υπάρχουν πολλές στοίβες που διασυνδέονται δημιουργώντας έτσι μια δομή δένδρου με ρίζα την αρχική στοίβα στόχων. Ο τελευταίος στόχος μια στοίβας που έχει παιδιά σε αυτό το δένδρο θα πρέπει να περιμένει να γραφούν όλοι οι στόχοι από τις δύο αυτές στοίβες ώστε να σχολιαστεί με βάση τους πρώτους τους στόχους για να γραφεί και αυτός στην απόδειξη.

2.3 FR-Φάση

Κατά την FR-φάση ο αλγόριθμος συνδυάζει επαναληπτικά τις γραμμές της απόδειξης με έναν κατάλληλο τρόπο ώστε να εφαρμοστούν οι FR-κανόνες και να αποδειχτούν νέοι τύποι που με την σειρά τους θα πρέπει να συνδυαστούν με τους υπάρχοντες τύπους μέχρι τελικά να γίνει η απόδειξη του τελικού απλουστευμένου στόχου που έχει τεθεί κατά την BR-φάση.

Οι FR-κανόνες είναι οι κανόνες elimination οι οποίοι παράγουν στην ουσία υπο-τύπους από τους τύπους που δέχονται ως είσοδο και αυτό είναι το ζητούμενο, αφού ο στόχος είναι κάποιο προτασιακό γράμμα ή το $\perp$.

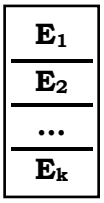
Επίσης, χρησιμοποιείται και ένα σύνολο από ειδικούς μετασχηματισμούς οι οποίοι αλλάζουν έναν τύπο σε κάποια ισοδύναμη μορφη. Παραδείγματα μετασχηματισμών είναι οι κανόνες De Morgan.

Ύστερα από την απόδειξη ενός νέου τύπου πρέπει να γίνεται έλεγχος για το αν είναι ο στόχος και αν είναι τότε η απόδειξη τερματίζει με επιτυχία. Αντίθετα, η απόδειξη κρίνεται αποτυχημένη όταν δεν μπορεί να γίνει κανένας συνδυασμός γραμμών ούτε και να εφαρμοστεί κάποιος από τους διαθέσιμους μετασχηματισμούς.

2.3.1 Επίπεδα Υποθέσεων

Ο αλγόριθμος στην ουσία παίρνει δύο γραμμές της απόδειξης, κάνει συντακτική ανάλυση των τύπων που περιέχουν και ελέγχει αν μπορεί να εφαρμοστεί κάποιος από τους FR-κανόνες και αν ναι, τότε αποδεικνύει ένα νέο τύπο ο οποίος θα πρέπει να ελεγθεί για το αν είναι ο στόχος. Ακόμη, υπάρχει η δυνατότητα να πάρει δύο φορές την ίδια γραμμή για να εφαρμόσει κανόνες οι οποίοι έχουν ως είσοδο μόνο ένα τύπο τον οποίο αναλύουν στους υπο-τύπους του.

Οι γραμμές της απόδειξης χωρίζονται σε επίπεδα ώστε να γίνει καλύτερη διαχείριση τους και να μην επαναλαμβάνονται οι εφαρμογές κανόνων σε γραμμές που ήδη έχουν εφαρμοστεί. Η απόδειξη δηλαδή είναι χωρισμένη ως εξής:



Το πρώτο επίπεδο υποθέσεων E_1 αποτελείται από το σύνολο των αρχικών υποθέσεων Σ ενώ θα μπορούσε αρχικά να υπάρχει και ένα δεύτερο επίπεδο E_2 το οποίο αποτελείται από τις υποθέσεις που ίσως έγιναν κατά την διάρκεια της BR-φάσης.

Όταν ο αλγόριθμος συνδυάζει δύο επίπεδα E_k - E_l τότε κάθε γραμμή του επιπέδου E_k πρέπει να συνδυαστεί με κάθε γραμμή του επιπέδου E_l ώστε να εφαρμοστεί κάποιος κανόνας.

Οι νέες γραμμές που παράγονται από έναν συνδυασμό επιπέδων αποτελούν μέρος ενός νέου επιπέδου. Ειδικότερα, για την συνολική παραγωγή ενός νέου επιπέδου E_k πρέπει να γίνουν οι συνδυασμοί των επιπέδων $\mathbf{E_1 - E_{k-1}, E_2 - E_{k-1}, ..., E_{k-1} - E_{k-1}}$.

Όταν γίνεται συνδυασμός ενός επιπέδου με τον εαυτό του τότε κάθε γραμμή του συνδυάζεται με κάθε άλλη γραμμή του και με τον εαυτό της και είναι ο μοναδικός τρόπος για να εφαρμοστεί ένας κανόνας που δέχεται ως είσοδο μόνο ένα τύπο όπως ο $\wedge$elim.

2.3.2 Κανόνας IMPLIES-Elimination

Αν ο τύπος της μίας γραμμής έχει την μορφή $\varphi_1 \rightarrow \varphi_2$ και ο τύπος της δεύτερης γραμμής είναι ο φ_1 τότε μπορεί να εφαρμοστεί ο κανόνας →elim με αποτέλεσμα την απόδειξη του φ_2.

Επομένως, όταν εφαρμόζεται ο κανόνας →elim, δημιουργείται μια νέα γραμμή στην απόδειξη η οποία περιέχει τον τύπο φ_2 και έπειτα γίνεται έλεγχος για το αν ο φ_2 είναι ο στόχος.

2.3.3 Κανόνας NOT-Elimination

Αν οι δύο συνδυαζόμενες γραμμές περιέχουν αντικρουόμενους τύπους φ και ¬φ τότε μπορεί να εφαρμοστεί ο κανόνας ¬elim με αποτέλεσμα την απόδειξη του ⊥.

Επομένως, όταν εφαρμόζεται ο κανόνας ¬elim, δημιουργείται μια νέα γραμμή στην απόδειξη η οποία περιέχει την σταθερά ⊥ και έπειτα γίνεται έλεγχος για το αν το ⊥ είναι ο στόχος. Στην περίπτωση που το ⊥ δεν είναι ο στόχος, μπορεί να εφαρμοστεί άμεσα ο κανόνας ⊥elim για να αποδειχτεί ο στόχος.

2.3.4 Κανόνας BOTTOM-Elimination

Αν ο στόχος είναι κάποιο προτασιακό γράμμα p και μια γραμμή της απόδειξης έχει ως τύπο το ⊥ τότε μπορεί να εφαρμοστεί ο κανόνας ⊥elim με αποτέλεσμα την απόδειξη του p.

Δηλαδή, όταν εφαρμόζεται ο κανόνας ⊥elim, δημιουργείται μια νέα γραμμή στην απόδειξη η οποία περιέχει τον στόχο p και άρα η απόδειξη κρίνεται επιτυχημένη.

Ο αλγόριθμος θα πρέπει να ελέγχει για την εφαρμογή του κανόνα ⊥elim σε κάθε νέα γραμμή που παράγεται κατά την εκτέλεση της FR-φάσης για την απόδειξη του στόχου αφού κάθε νέος τύπος ενδέχεται να είναι το ⊥ (πέρα από την περίπτωση του ¬elim όπου όντως είναι).

2.3.5 Κανόνας AND-Elimination

Αν κάποια γραμμή συνδυαστεί με τον εαυτό της και περιέχει τύπο μορφής $\varphi_1 \wedge \varphi_2$ τότε μπορεί να εφαρμοστεί ο κανόνας ∧elim με αποτέλεσμα την απόδειξη των φ_1 και φ_2.

Επομένως, όταν εφαρμόζεται ο κανόνας ∧elim, δημιουργούνται δύο νέες γραμμές οι οποίες περιέχουν τους τύπους φ_1 και φ_2 αντίστοιχα. Έπειτα, γίνεται έλεγχος για το αν κάποιος από τους δύο αυτούς νέους τύπους είναι ο στόχος.

Μια βελτίωση είναι η εξής: Πριν από την εγγραφή του δεύτερου τύπου μπορεί να γίνεται ένας έλεγχος για το αν ο πρώτος τύπος είναι ο στόχος ώστε σε περίπτωση που είναι, να αποφεύγεται η εγγραφή ενός περιττού, ως προς την απόδειξη του στόχου, τύπου.

Ο κανόνας ∧elim είναι ο μοναδικός που παράγει δύο νέες γραμμές κατά την εφαρμογή του.

2.3.6 Κανόνας OR-Elimination

Αν κάποια γραμμή συνδυαστεί με τον εαυτό της και περιέχει τύπο μορφής $\varphi_1 \vee \varphi_2$ τότε μπορεί να γίνει αναδρομική αναγωγή της απόδειξης σε δύο υπο-αποδείξεις με ίδιο στόχο αλλά άλλο σύνολο υποθέσεων με σκοπό να εφαρμοστεί ο κανόνας ∨elim.

Άρα:

Η απόδειξη $\boldsymbol{\Sigma, \varphi_1 \vee \varphi_2 \vdash G}$ ανάγεται στις αποδείξεις $\boldsymbol{\Sigma, \varphi_1 \vdash G}$ και $\boldsymbol{\Sigma, \varphi_2 \vdash G}$

Για την εφαρμογή του κανόνα Velim κάνουμε αναδρομή όπως ακριβώς κάνουμε και κατά την BR-φάση όταν εφαρμόζουμε τον κανόνα Λintro καθώς θα πρέπει με διαφορετικές υποθέσεις να καταλήξουμε στον ίδιο στόχο.

Η αναγωγή γίνεται κατά την FR-φάση όπου έχουν ήδη γίνει συνδυασμοί επιπέδων και έτσι, στις νέες υπο-αποδείξεις οι συνδυασμοί επιπέδων θα πρέπει να συνεχίσουν από το σημείο στο οποίο σταμάτησαν πριν ο έλεγχος περάσει στις υπο-αποδείξεις.

Αν και οι δύο υπο-αποδείξεις είναι επιτυχείς τότε ο αλγόριθμος δημιουργεί μια νέα γραμμή η οποία περιέχει τον στόχο G και επομένως η κύρια απόδειξη κρίνεται επιτυχημένη.

2.3.7 Κανόνας MT

Αν ο τύπος της μιας γραμμής έχει την μορφή $\varphi_1 \rightarrow \varphi_2$ και ο τύπος της δεύτερης γραμμής είναι ο $\neg\varphi_2$ τότε μπορεί να εφαρμοστεί ο κανόνας MT με αποτέλεσμα την παραγωγή του $\neg\varphi_1$.

Επομένως, όταν εφαρμόζεται ο κανόνας MT, δημιουργείται μια νέα γραμμή η οποία περιέχει τον τύπο $\neg\varphi_1$ και έπειτα γίνεται έλεγχος αν ο $\neg\varphi_1$ είναι ο στόχος.

2.3.8 Κανόνας NOT-NOT-Elimination

Αν κάποια γραμμή συνδυαστεί με τον εαυτό της και περιέχει τύπο μορφής $\neg\neg\varphi$ τότε μπορεί να εφαρμοστεί ο κανόνας ¬¬elim με αποτέλεσμα την απόδειξη του φ.

Επομένως, όταν εφαρμόζεται ο κανόνας ¬¬elim, δημιουργείται μια νέα γραμμή η οποία έχει τον τύπο φ και έπειτα ακολουθεί έλεγχος για το αν ο τύπος φ είναι ο στόχος.

2.4 Μετασχηματισμοί

Οι μετασχηματισμοί χρησιμοποιούν γνωστές λογικές ισοδυναμίες για την μετατροπή τύπων σε ισοδύναμες μορφές οι οποίες μπορεί να είναι πιο χρήσιμες κατά την προσπάθεια απόδειξης νέων τύπων κατά την διάρκεια της FR-φάσης.

Ωστόσο, οι μετασχηματισμοί εφαρμόζονται μόνο όταν δεν μπορούν να γίνουν άλλοι συνδυασμοί γραμμών για την εφαρμογή των βασικών κανόνων. Ο λόγος για τον οποίο ακολουθούμε αυτή την στρατηγική είναι το γεγονός ότι οι μετασχηματισμοί ως κανόνες δεν αποτελούν μέρος του συστήματος και ως μοναδικό σκοπό έχουν την αλλαγή της μορφής κάποιου τύπου χωρίς να μας ενδιαφέρει η απόδειξη του στόχου (αν και η μετασχηματισμένη αυτή μορφή μπορεί και να είναι ο στόχος).

2.4.1 Μετασχηματισμός ∨to∧ DM

Αν κάποια γραμμή της απόδειξης έχει τύπο της μορφής $\neg(\varphi_1 \vee \varphi_2)$ τότε με βάση τον κανόνα De Morgan $\neg(\varphi_1 \vee \varphi_2) \equiv (\neg\varphi_1) \wedge (\neg\varphi_2)$ μπορούμε να δημιουργήσουμε μια νέα γραμμή η οποία να περιέχει τον ισοδύναμο τύπο $(\neg\varphi_1) \wedge (\neg\varphi_2)$.

2.4.2 Μετασχηματισμός ∧to∨ DM

Αν κάποια γραμμή της απόδειξης έχει τύπο της μορφής $\neg(\varphi_1 \wedge \varphi_2)$ τότε με βάση τον κανόνα De Morgan $\neg(\varphi_1 \wedge \varphi_2) \equiv (\neg\varphi_1) \vee (\neg\varphi_2)$ μπορούμε να δημιουργήσουμε μια νέα γραμμή η οποία να περιέχει τον ισοδύναμο τύπο $(\neg\varphi_1) \vee (\neg\varphi_2)$.

2.4.3 Μετασχηματισμός →to∧ Transformation

Αν κάποια γραμμή της απόδειξης έχει τύπο της μορφής $\neg(\varphi_1 \rightarrow \varphi_2)$ τότε μπορούμε να αξιοποιήσουμε την λογική ισοδυναμία $\neg(\varphi_1 \rightarrow \varphi_2) \equiv \varphi_1 \wedge (\neg\varphi_2)$ ώστε να δημιουργήσουμε μια νέα γραμμή με τον ισοδύναμο τύπο $\varphi_1 \wedge (\neg\varphi_2)$.

2.4.4 Μετασχηματισμός →to∨ Transformation

Αν κάποια γραμμή της απόδειξης έχει τύπο της μορφής $\varphi_1 \rightarrow \varphi_2$ τότε μπορούμε να αξιοποιήσουμε την λογική ισοδυναμία $\varphi_1 \rightarrow \varphi_2 \equiv (\neg\varphi_2) \vee \varphi_1$ ώστε να δημιουργήσουμε μια νέα γραμμή με τον ισοδύναμο τύπο $(\neg\varphi_2) \vee \varphi_1$.

2.4.5 Συνδυασμός Μετασχηματισμών Με Βασικούς Κανόνες

Οι μετασχηματισμοί παράγουν τύπους οι οποίοι είναι είτε συζεύξεις είτε διαζεύξεις. Συνεπώς, μπορούμε να συνδυάσουμε τους μετασχηματισμούς με τους κανόνες ∧elim και ∨elim για να κάνουμε ακόμη πιο γρήγορο τον αλγόριθμο.

Πιο συγκεκριμένα:

Οι μετασχηματισμοί ∨to∧ DM και →to∧ μπορούν να συνδυαστούν με τον κανόνα ∧elim ώστε να παραχθούν άμεσα σε δύο νέες γραμμές οι δύο υπο-τύποι του ισοδύναμου νέου τύπου που παράγουν ο καθένας.

Οι μετασχηματισμοί ∧to∨ DM και →to∨ μπορούν να συνδυαστούν με τον κανόνα ∨elim ώστε να γίνουν σε κάθε περίπτωση να γίνουν δύο νέες υποθέσεις και να ξεκινήσουν αναδρομικά οι δύο νέες υπο-αποδείξεις όπως ορίζει ο κανόνας ∨elim.

2.4.6 Ορθότητα Μετασχηματισμών

Οι μετασχηματισμοί δεν είναι μέρος των βασικών κανόνων του συστήματος ND και επομένως θα πρέπει να δείξουμε την ορθότητα τους γράφοντας αποδείξεις όπου με βασική υπόθεση την είσοδο ενός μετασχηματισμού και με χρήση των βασικών (ή παραγόμενων) κανόνων του ND καταλήγουμε στον ισοδύναμο τύπο που παράγει ο μετασχηματισμός αυτός.

1. $\neg(\varphi_1 \rightarrow \varphi_2)$

2. $\neg\varphi_1$ **Υπόθεση**

3. φ_1 **Υπόθεση**
4. $\perp$ **¬elim 2,3**
5. φ_2 **⊥elim 47**

6. $\varphi_1 \rightarrow \varphi_2$ **→intro {3-5}**
7. $\perp$ **¬elim 1,6**

8. φ_1 **PBC {2-7}**

9. φ_2 **Υπόθεση**

10. φ_1 **Υπόθεση**
11. φ_2 **copy 9**

12. $\varphi_1 \rightarrow \varphi_2$ **→intro {3-5}**
13. $\perp$ **¬elim 1,6**

14. $\neg\varphi_2$ **¬intro {9-13}**
15. $\varphi_1 \wedge (\neg\varphi_2)$ **∧intro 8,14**

Απόδειξη μετασχηματισμού
→to∧ Transformation

1. $\varphi_1 \rightarrow \varphi_2$

2. φ_1 **Υπόθεση**
3. φ_2 **→elim 1,2**

4. $(\neg\varphi_1) \vee \varphi_2$ **gen ∨intro {2-3}**

Απόδειξη μετασχηματισμού
→to∨ Transformation

1. $\neg(\varphi_1 \vee \varphi_2)$

2. φ_1 **Υπόθεση**
3. $\varphi_1 \vee \varphi_2$ **∨intro 2**
4. $\perp$ **¬elim 1,3**

5. $\neg\varphi_1$ **¬intro {2-4}**

6. φ_2 **Υπόθεση**
7. $\varphi_1 \vee \varphi_2$ **∨intro 6**
8. $\perp$ **¬elim 1,7**

9. $\neg\varphi_2$ **¬intro {6-8}**
10. $(\neg\varphi_1) \wedge (\neg\varphi_2)$ **∧intro 5,9**

Απόδειξη μετασχηματισμού
∨to∧ DM

1. $\neg(\varphi_1 \wedge \varphi_2)$

2. φ_1 **Υπόθεση**

3. φ_2 **Υπόθεση**
4. $\varphi_1 \wedge \varphi_2$ **∧intro 2,3**
5. $\perp$ **¬elim 1,4**

6. $\neg\varphi_2$ **¬intro {3-5}**

7. $(\neg\varphi_1) \vee (\neg\varphi_2)$ **gen ∨intro {2-6}**

Απόδειξη μετασχηματισμού
∧to∨ DM

Η ορθότητα των μετασχηματισμών είναι βασικό στοιχείο της ορθότητας του αλγορίθμου αφού για να είναι ορθός ο αλγόριθμος, δηλαδή να παράγει ορθά αποτελέσματα με βάση το σύνολο των αρχικών υποθέσεων, θα πρέπει να στηρίζεται σε ορθούς κανόνες.

Από την στιγμή που γνωρίζουμε ότι οι βασικοί κανόνες του συστήματος ND είναι ορθοί τότε πρέπει να είναι και οι μετασχηματισμοί ορθοί αφού χρησιμοποιούμε τους βασικούς κανόνες για να τους αποδείξουμε.

2.5 Απομάκρυνση Περιττών Γραμμών

Όταν αποδεικνύεται ένας νέος τύπος τότε η γραμμή που τον περιέχει πρέπει να περιέχει και ένα σχόλιο στο οποίο περιγράφεται το πώς έγινε η απόδειξη του. Αυτά τα σχόλια μπορούν να χρησιμοποιηθούν ώστε να βελτιστοποιήσουμε την απόδειξη αφού έχει ολοκληρωθεί. Δηλαδή, μπορούμε να απομακρύνουμε γραμμές με τύπους οι οποίοι ναι μεν αποδείχθηκαν αλλά στην πραγματικότητα δεν χρησίμευσαν στην απόδειξη του στόχου.

Για τον σκοπό αυτό κάνουμε μια αναζήτηση καθοδηγούμενη από τον στόχο ώστε να βρούμε όλες τις γραμμές της απόδειξης που πραγματικά βοήθησαν στην απόδειξη του ως εξής:

Διατρέχουμε την απόδειξη από το τέλος προς την αρχή και μαρκάρουμε γραμμές. Στο τέλος της απόδειξης βρίσκεται ο στόχος και άρα από τα σχόλια αυτής της γραμμής βλέπουμε ποιες γραμμές πρέπει να μαρκάρουμε αρχικά. Έπειτα, κατά την διάτρεξη, αν η γραμμή στην οποία είμαστε είναι μαρκαρισμένη, τότε μαρκάρουμε και τις γραμμές που βοήθησαν για την δική της παραγωγή με βάση τα σχόλια που έχει. Τέλος, μαρκάρονται σίγουρα οι γραμμές με τις αρχικές υποθέσεις ανεξάρτητα αν αυτές βοήθησαν ή όχι στην απόδειξη του στόχου.

Στο τέλος αυτής της διαδικασίας, όλες οι μαρκαρισμένες γραμμές είναι αυτές που αποτελούν την πραγματική απόδειξη. Οι υπόλοιπες μπορούν απλά να απομακρυνθούν.

2.6 Ο Συνολικός Αλγόριθμος

Έστω Σ ένα σύνολο αρχικών υποθέσεων και G ένας προτασιακός τύπος-στόχος. Ο αλγόριθμος θα εκτελέσει τα επόμενα βήματα για να εξακριβώσει αν ο τύπος G μπορεί να αποδειχτεί από το αρχικό σύνολο Σ με βάση τους κανόνες και τους μετασχηματισμούς που περιγράψαμε στις προηγούμενες παραγράφους.

Η βασική λειτουργία του αλγορίθμου χωρίζεται σε μια φάση Backward ανάλυσης και σε μια φάση Forward ανάλυσης. Για αυτόν τον λόγο θα τον ονομάσουμε Backward-Forward Proof-Searcher (BFPS).

Βήμα 1: Αρχικός Έλεγχος.

Αν ο στόχος G ανήκει στο σύνολο Σ ή είναι η σταθερά ⊤ τότε:
Η απόδειξη ολοκληρώνεται με επιτυχία.
Ο αλγόριθμος τερματίζει χωρίς να προχωρήσει στα επόμενα βήματα.

Βήμα 2: Backward Reasoning Φάση.

Όσο (ο τρέχων στόχος δεν είναι ατομικός τύπος) επανάλαβε:

Με συντακτική ανάλυση βρες το top-level λογικό συνδετικό του τρέχων στόχου.

Ανάλογα με το top-level συνδετικό εκτέλεσε τον ανάλογο BR-κανόνα θέτοντας έτσι έναν νέο υποστόχο G_{New} και κάνοντας μια νέα υπόθεση φ αν απαιτείται η οποία προστίθενται στο σύνολο υποθέσεων Σ=ΣU{φ}.

Συμπλήρωσε το σχόλιο του προηγούμενου στόχου με τον κανόνα που χρησιμοποιήθηκε και την γραμμή όπου αποθηκεύτηκε η νέα υπόθεση, αν φυσικά έγινε κάποια υπόθεση.

Αν (ο νέος στόχος G_{New} είναι η σταθερά ⊤) τότε:
Η απόδειξη ολοκληρώνεται με επιτυχία μέσω του κανόνα ⊤intro.

Αν (ο νέος στόχος G_{New} ανήκει στο σύνολο υποθέσεων Σ) τότε:
Η απόδειξη ολοκληρώνεται με επιτυχία μέσω του κανόνα copy.

Βήμα 3: Forward Reasoning Φάση.

Μέχρι να ολοκληρωθεί η απόδειξη επανάλαβε:

Συνδύασε κατάλληλα τα επίπεδα γραμμών που υπάρχουν για εφαρμογή των FR-κανόνων →elim, ¬elim, Λelim, Velim, MT και ¬¬elim.

Αν (έγινε κάποια απόδειξη) τότε:
Αν (ο νέος τύπος είναι ο στόχος) τότε: Η απόδειξη ολοκληρώνεται με επιτυχία.

Αν (αποδείχτηκε η σταθερά ⊥) τότε:
Η απόδειξη ολοκληρώνεται με επιτυχία μέσω του κανόνα ⊥elim.

Αν (αποδείχτηκε τύπος με μορφή ¬¬φ) τότε:
Εφάρμοσε τον κανόνα ¬¬elim για την απόδειξη του τύπου φ.
Αν (ο τύπος φ είναι ο στόχος) τότε: Η απόδειξη ολοκληρώνεται με επιτυχία.

Αν (αποδείχτηκε τύπος με μορφή φΛψ) τότε:
Εφάρμοσε τον κανόνα Λelim για την απόδειξη των τύπων φ και ψ.
Αν (ο φ ή ο ψ είναι ο στόχος) τότε: Η απόδειξη ολοκληρώνεται με επιτυχία.

Αλλιώς (δεν έγινε κάποια απόδειξη):
Αν (ο στόχος είναι κάποιο προτασιακό γράμμα) τότε: Εφάρμοσε τον κανόνα PBC.

Αλλιώς (ο στόχος είναι η σταθερά ⊥):
Σάρωσε την μέχρι-τώρα απόδειξη για εφαρμογή κάποιου μετασχηματισμού από τους ∨to∧ DM, ∧to∨ DM και →to∧ transformation.

Αν (έγινε κάποια απόδειξη) τότε:
Αν (ο νέος τύπος είναι ο στόχος) τότε: Η απόδειξη ολοκληρώνεται με επιτυχία.

Αν (δεν πετύχει κάποιος μετασχηματισμός) τότε:
Σάρωσε την μέχρι-τώρα απόδειξη για εφαρμογή του →to∨ transformation.

Αν (έγινε κάποια απόδειξη) τότε:
Αν (ο νέος τύπος είναι ο στόχος) τότε: Η απόδειξη ολοκληρώνεται με επιτυχία.

Αν (δεν πετύχει ο μετασχηματισμός) τότε:
Η απόδειξη ολοκληρώνεται με αποτυχία και ο αλγόριθμος τερματίζει.

Βήμα 4: Εγγραφή Στόχων Στην Απόδειξη.

Όσο (η στοίβα στόχων δεν είναι κενή) επανάλαβε:
Γράψε σε μια νέα γραμμή τον στόχο που είναι στην κορυφή της στοίβας.
Ενημέρωσε το σχόλιο του στόχου ότι χρησιμοποιήθηκε επίσης η προτελευταία γραμμή της απόδειξης.

Βήμα 5: Απομάκρυνση Περιττών Γραμμών.

Μάρκαρε τις πρώτες γραμμές της απόδειξης όπου είναι αποθηκευμένοι οι τύποι του αρχικού συνόλου υποθέσεων Σ καθώς και την τελευταία γραμμή όπου είναι αποθηκευμένος ο στόχος.

Σάρωσε την απόδειξη από το τέλος προς την αρχή:

Αν (η τρέχουσα γραμμή είναι μαρκαρισμένη) τότε:
Μάρκαρε τις γραμμές που βρίσκονται στο σχόλιο της τρέχουσας γραμμής.

Αλλιώς (η τρέχουσα γραμμή δεν είναι μαρκαρισμένη):
Διέγραψε την τρέχουσα γραμμή από την απόδειξη.

2.7 Ορθότητα Και Πληρότητα Αλγορίθμου

Θα αποδείξουμε ότι ο αλγόριθμος BFPS είναι ορθός και πλήρης. Με άλλα λόγια, θα δείξουμε ότι βρίσκει μόνο έγκυρες αποδείξεις καθώς και ότι μπορεί να βρει όλες τις έγκυρες αποδείξεις με βάση ένα σύνολο υποθέσεων.

2.7.1 Ορθότητα Αλγορίθμου

Θεώρημα. Αν ο αλγόριθμος BFPS μπορεί να βρει μια απόδειξη $\Sigma \vdash G$ τότε ισχύει $\Sigma \models G$.

Απόδειξη. Το θέωρημα ορθότητας του αλγορίθμου συνεπάγεται άμεσα από την ιδιότητα της ορθότητας του αποδεικτικού συστήματος ND καθώς ο αλγόριθμος χρησιμοποιεί κανόνες από το σύστημα, ενώ και οι μετασχηματισμοί μπορούν να αποδειχτούν όπως έχουμε ήδη δείξει με βάση αυτούς τους κανόνες. □

Άρα, αν ένας τύπος μπορεί να αποδειχτεί από τον αλγόριθμο με βάση ένα σύνολο υποθέσεων Σ τότε συνεπάγεται και λογικά από αυτό. Αυτό σημαίνει ότι ο αλγόριθμος βρίσκει αποδείξεις οι οποίες είναι έγκυρες.

2.7.2 Πληρότητα Αλγορίθμου

Λήμμα 1. Αν κατά την FR-φάση του αλγορίθμου δεν αποδεικνύεται το ⊥ τότε η απόδειξη μπορεί να επεκταθεί σε ένα σύνολο Hintikka.

Απόδειξη. Έστω Σ το σύνολο αρχικών υποθέσεων για το οποίο εφαρμόζεται η FR-φάση κατά την διάρκεια της οποίας δεν αποδεικνύεται το ⊥. Έστω ότι με $\Sigma \vdash_{FR} \varphi$ συμβολίζουμε το γεγονός ότι κατά την FR-φάση του αλγορίθμου αποδεικνύεται ο τύπος φ με βάση ένα σύνολο αρχικών υποθέσεων Σ. Θα αποδείξουμε ότι η FR-φάση του αλγορίθμου έχει τις εξής ιδιότητες:

(1) $\Sigma \vdash_{FR} \varphi \Rightarrow \Sigma \nvdash_{FR} \neg\varphi$

(2) $\Sigma \vdash_{FR} \neg\neg\varphi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$

(3) $\Sigma \vdash_{FR} \varphi\wedge\psi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\psi \nvdash_{FR} \bot$

(4) $\Sigma \vdash_{FR} \neg(\varphi\vee\psi) \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\neg\psi \nvdash_{FR} \bot$

(5) $\Sigma \vdash_{FR} \varphi\vee\psi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\psi \nvdash_{FR} \bot$

(6) $\Sigma \vdash_{FR} \neg(\varphi\wedge\psi) \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\neg\psi \nvdash_{FR} \bot$

(7) $\Sigma \vdash_{FR} \varphi\rightarrow\psi \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\psi \nvdash_{FR} \bot$

(8) $\Sigma \vdash_{FR} \neg(\varphi\rightarrow\psi) \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\neg\psi \nvdash_{FR} \bot$

Με βάση τις ιδιότητες αποδεικνύουμε ότι αν ο αλγόριθμος μπορεί να κάνει μια απόδειξη τότε το σύνολο των παραγόμενων τύπων μαζί με το σύνολο υποθέσεων είναι συνεπές και μπορεί να επεκταθεί σε ένα σύνολο Hintikka.

(1) Έστω ότι ο αλγόριθμος μπορεί να κάνει τις αποδείξεις $\Sigma \vdash_{FR} \varphi$ και $\Sigma \vdash_{FR} \neg\varphi$. Τότε, κατά την FR-φάση θα μπορέσει να συνδυάσει τους τύπους φ και ¬φ και να εφαρμόσει το κανόνα ¬elim με αποτέλεσμα να αποδείξει το ⊥ το οποίο έχουμε υποθέσει ότι δεν γίνεται. Συνεπώς, αν ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi$ τότε $\Sigma \nvdash_{FR} \neg\varphi$.

(2) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \neg\neg\varphi$ ενώ ισχύει $\Sigma,\varphi \vdash_{FR} \bot$. Τότε, με εφαρμογή του κανόνα $\neg\neg$elim ο αλγόριθμος είναι σε θέση να αποδείξει κατά την FR-φάση τον τύπο φ και άρα το ⊥ το οποίο έχουμε υποθέσει ότι δεν γίνεται. Συνεπώς, αν ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \neg\neg\varphi$ τότε $\Sigma,\varphi \nvdash_{FR} \bot$.

(3) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi\wedge\psi$ ενώ ισχύει ότι $\Sigma,\varphi \vdash_{FR} \bot$ ή $\Sigma,\psi \vdash_{FR} \bot$. Τότε, με εφαρμογή του κανόνα $\wedge$elim ο αλγόριθμος είναι σε θέση να αποδείξει κατά την FR-φάση τους τύπους φ και ψ και άρα και το ⊥ το οποίο έχουμε υποθέσει ότι δεν γίνεται. Άρα, αν ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi\wedge\psi$ τότε $\Sigma,\varphi \nvdash_{FR} \bot$ και $\Sigma,\psi \nvdash_{FR} \bot$.

(4) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \neg(\varphi\vee\psi)$. Τότε, με εφαρμογή του μετασχηματισμού $\vee$to$\wedge$ DM μπορεί να κάνει και την απόδειξη $\Sigma \vdash_{FR} (\neg\varphi)\wedge(\neg\psi)$. Συνεπώς, από την ιδιότητα (3) πρέπει $\Sigma,\neg\varphi \nvdash_{FR} \bot$ και $\Sigma,\neg\psi \nvdash_{FR} \bot$.

(5) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi\vee\psi$ ενώ ισχύει ότι $\Sigma,\varphi \vdash_{FR} \bot$ και $\Sigma,\psi \vdash_{FR} \bot$. Τότε, με εφαρμογή του κανόνα $\vee$elim ο αλγόριθμος είναι σε θέση να αποδείξει κατά την FR-φάση το ⊥ το οποίο είναι έχουμε υποθέσει ότι δεν γίνεται. Επομένως, αν ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi\vee\psi$ τότε $\Sigma,\varphi \nvdash_{FR} \bot$ ή $\Sigma,\psi \nvdash_{FR} \bot$.

(6) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \neg(\varphi\wedge\psi)$. Τότε, με εφαρμογή του μετασχηματισμού $\wedge$to$\vee$ DM μπορεί να κάνει και την απόδειξη $\Sigma \vdash_{FR} (\neg\varphi)\vee(\neg\psi)$. Συνεπώς, από την ιδιότητα (5) πρέπει $\Sigma,\neg\varphi\nvdash\bot$ ή $\Sigma,\neg\psi\nvdash\bot$.

(7) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \varphi\rightarrow\psi$. Τότε, με την εφαρμογή του μετασχηματισμού $\rightarrow$to$\vee$ Transformation μπορεί να κάνει και την απόδειξη $\Sigma \vdash_{FR} (\neg\varphi)\vee\psi$. Άρα, από την ιδιότητα (5) πρέπει $\Sigma,\neg\varphi \nvdash_{FR} \bot$ ή $\Sigma,\psi \nvdash_{FR} \bot$.

(8) Έστω ότι ο αλγόριθμος μπορεί να κάνει την απόδειξη $\Sigma \vdash_{FR} \neg(\varphi\rightarrow\psi)$. Τότε, με την εφαρμογή του μετασχηματισμού $\rightarrow$to$\wedge$ Transformation μπορεί να κάνει και την απόδειξη $\Sigma \vdash_{FR} \varphi\wedge(\neg\psi)$. Άρα, από την ιδιότητα (3) πρέπει $\Sigma,\varphi \nvdash_{FR} \bot$ και $\Sigma,\neg\psi \nvdash_{FR} \bot$.

Οπότε, η απόδειξη μπορεί να επεκταθεί σε ένα σύνολο Hintikka το οποίο με βάση το λήμμα Hintikka θα είναι ικανοποιήσιμο και αφού είναι υπερσύνολο του Σ, συνεπάγεται ότι και το Σ είναι ικανοποιήσιμο. □

Λήμμα 2. Αν ισχύει ότι $\Sigma\vDash G$ τότε η FR-φάση του αλγορίθμου αποδεικνύει είτε τον στόχο G ο οποίος είναι είτε ατομικός τύπος είτε το ⊥ μετά από την εκτέλεση του κανόνα PBC.

Απόδειξη. Θα αποδείξουμε το Λήμμα 2 με εις άτοπο απαγωγή. Άρα, έστω ότι το Λήμμα 2 δεν ισχύει. Οπότε, κατά την FR-φάση ο αλγόριθμος δεν αποδεικνύει το G ούτε και το ⊥ μετά από την εκτέλεση του κανόνα PBC.

Περίπτωση G≡⊥:

Αν δεν ισχύει το Λήμμα 2 τότε ο αλγόριθμος δεν μπορεί να αποδείξει το ⊥ και επομένως από το Λήμμα 1 το Σ πρέπει να είναι ικανοποιήσιμο αφού η απόδειξη μπορεί να επεκταθεί σε ένα σύνολο Hintikka. Αυτό όμως είναι άτοπο καθώς $\Sigma\vDash G \Rightarrow \Sigma\vDash$false.

Περίπτωση G≡p:

Αν δεν ισχύει το Λήμμα 2 τότε ο αλγόριθμος δεν μπορεί να αποδείξει το προτασιακό γράμμα p και έτσι, με βάση την λειτουργία του θα εκτελέσει τον κανόνα PBC με αποτέλεσμα το σύνολο Σ να εμπλουτιστεί με την υπόθεση ¬p ενώ νέος στόχος να τεθεί το ⊥.

Ωστόσο, ο αλγόριθμος δεν μπορεί να αποδείξει το ⊥ μετά την εκτέλεση του κανόνα PBC. Έτσι, με βάση το Λήμμα 1, το σύνολο Σ∪{¬p} είναι ικανοποιήσιμο το οποίο όμως είναι άτοπο αφού Σ⊨p ⇒ Σ,¬p⊨false.

Επομένως, σε κάθε περίπτωση υποθέτοντας ότι το Λήμμα 2 δεν ισχύει καταλήγουμε σε άτοπο και άρα, το Λήμμα 2 ισχύει. □

Θεώρημα. Αν ισχύει Σ⊨G τότε ο αλγόριθμος BFPS μπορεί να βρει μια απόδειξη Σ⊢G.

Απόδειξη. Ο αλγόριθμος κατά την αναζήτηση της απόδειξης Σ⊢G, θα εκτελέσει την BR-φάση η οποία θα έχει ως αποτέλεσμα την αναγωγή της απόδειξης σε μια απλούστερη απόδειξη Σ′⊢G′ όπου το Σ′ είναι ένα υπερσύνολο του Σ και ο στόχος G′ είναι ένας ατομικός τύπος, δηλαδή είτε ένα προτασιακό γράμμα είτε το ⊥.

Λόγω της ορθότητας της BR-φάσης ισχύει ότι αν Σ⊨G τότε και Σ′⊨G′ και με βάση το λήμμα 2, ο αλγόριθμος είναι σε θέση να κάνει την απόδειξη Σ′⊢G′. □

Επομένως, ο αλγόριθμος είναι πλήρης και μπορεί να βρει αποδείξεις για όλους τους τύπους οι οποίοι συνεπάγονται λογικά από το αρχικό σύνολο υποθέσεων Σ. Αυτό βέβαια, με βάση το γεγονός ότι το υποκείμενο πρόβλημα είναι το SAT, συνεπάγεται ότι στην χειρότερη περίπτωση ο αλγόριθμος είναι εκθετικός.

Μπορούμε να παρατηρήσουμε στο σημείο αυτό ότι, η πληρότητα του αλγορίθμου (και γενικά του συστήματος που περιγράψαμε) είναι ανεξάρτητη από την στρατηγική σειρά εκτέλεσης των κανόνων και των μετασχηματισμών της FR-φάσης. Για να είναι πλήρες το σύστημα αρκεί να γίνει μια δίκαιη εφαρμογή όλων των κανόνων και των μετασχηματισμών με την έννοια ότι όλοι οι κανόνες και οι μετασχηματισμοί πρέπει κάποια στιγμή να εφαρμοστούν.

2.8 Πολυπλοκότητα Αλγορίθμου

Θα αναλύσουμε την πολυπλοκότητα του αλγορίθμου σε μια απλή περίπτωση όπου υπόθεση είναι ένας λογικός τύπος σε 2-CNF με N προτάσεις και στόχος είναι το ⊥.

2.8.1 Μέγεθος Απόδειξης

Αφού στόχος είναι το ⊥ το οποίο είναι ατομικός τύπος, δεν θα εκτελεστεί η BR-φάση και άρα, δεν θα γίνουν επιπλέον υποθέσεις.

Από την στιγμή που οι N υποθέσεις είναι 2-προτάσεις, στην FR-φάση μπορεί να εφαρμοστεί μόνο ο κανόνας Velim ο οποίος θα απαιτήσει να γίνουν συνολικά 2^N υπο-αποδείξεις.

Σε κάθε μία από αυτές τις υπο-αποδείξεις θα γίνουν N υποθέσεις, μία για κάθε 2-πρόταση, οι οποίες είναι όλες όροι, δηλαδή, είναι προτασιακά γράμματα ή οι αρνήσεις τους. Επομένως, ο μοναδικός κανόνας που μπορεί να εφαρμοστεί είναι ο ¬elim για την απόδειξη του ⊥ μόνο σε περίπτωση όπου υπάρχουν αντικρουόμενοι όροι.

Επομένως, στο τέλος της απόδειξης, κάθε μία από τις 2^N υπο-αποδείξεις θα έχει το πολύ N+1 γραμμές το οποίο σημαίνει ότι το μέγεθος της απόδειξης είναι $O(2^N N)$.

Ο κανόνας Velim είναι υπεύθυνος για την εκθετική έκρηξη που συμβαίνει κατά την εύρεση μιας απόδειξης καθώς μπορεί να χρειαστεί να γίνουν πολλές εμφωλευμένες υπο-αποδείξεις η κάθε μία με άλλο σύνολο υποθέσεων αλλά ίδιο στόχο.

2.8.2 Απαιτήσεις Προσωρινού Χώρου

Είδαμε ότι κατά την εκτέλεση της FR-φάσης, απαιτούνται συνολικά 2^N υπο-αποδείξεις λόγω της εφαρμογής του κανόνα Velim. Ωστόσο, αυτές οι υπο-αποδείξεις δεν γίνονται παράλληλα αλλά η μία μετά την άλλη με αποτέλεσμα στην ουσία να γίνει μία αναζήτηση βάθους. Οπότε, οι απαιτήσεις χώρου είναι μικρότερες.

Στο συγκεκριμένο παράδειγμα, μπορούμε να φανταστούμε την λειτουργία του αλγορίθμου ως ένα δυαδικό δένδρο στο οποίο η ρίζα έχει όλες τις αρχικές υποθέσεις οι οποίες προφανώς θα πρέπει να είναι αποθηκευμένες στην μνήμη. Σε κάθε στιγμή, ο έλεγχος του προγράμματος είναι σε ένα κόμβο και όταν γίνεται εφαρμογή του Velim, ο κόμβος αυτός αποκτά δύο παιδιά κάθε ένα από τα οποία έχει ως ετικέτα την νέα υπόθεση που γίνεται και η οποία θα πρέπει να είναι επίσης αποθηκευμένη στην μνήμη.

Επειδή οι υπο-αποδείξεις γίνονται η μία μετά την άλλη, στην μνήμη υπάρχει πάντα μόνο ένα μονοπάτι αυτού του δένδρου το οποίο θα έχει μήκος N. Επομένως, στην μνήμη θα πρέπει να υπάρχουν οι N αρχικές υποθέσεις, οι N υποθέσεις της τρέχουσας υπο-απόδειξης και τέλος, οι N υποθέσεις των εναλλακτικών υπο-αποδείξεων που θα γίνουν στην συνέχεια. Άρα, συνολικά απαιτείται 3N προσωρινή μνήμη.

Για να το καταλάβουμε ακόμη περισσότερο, ας φανταστούμε ότι περπατάμε πάνω σε ένα απο αυτά τα μονοπάτια καθώς δημιουργείται. Όταν είμαστε σε έναν κόμβο και γίνεται εφαρμογή του Velim εμείς θα προχωρήσουμε σε ένα από τα δύο παιδιά του αλλά επίσης, θα πρέπει να θυμόμαστε και την υπόθεση που είχαμε κάνει στο άλλο του παιδί ώστε να επιστρέψουμε και να πάμε και από τον εναλλακτικό δρόμο.

Με άλλα λόγια, δημιουργούμε ένα δένδρο αναζήτησης βάθους.

Κεφάλαιο 3
Μερικές Βελτιώσεις

3.1 Το Πρόβλημα Του Κανόνα Velim

Ο αλγόριθμος BFPS που παρουσιάσαμε στο δεύτερο κεφάλαιο είναι ορθός και πλήρης αλλά δυστυχώς είναι εξαιρετικά αργός καθώς ακόμη και για ένα απλό παράδειγμα όπως αυτό όπου η υπόθεση είναι σε 2-CNF δημιουργεί μια απόδειξη εκθετικού μεγέθους και φυσικά απαιτεί εκθετικό χρόνο για την δημιουργία της.

Επομένως, ο αλγόριθμος μας δεν είναι αποδοτικός. Αυτό οφείλεται στην εφαρμογή του κανόνα Velim ο οποίος απαιτεί να γίνουν δύο υπο-αποδείξεις. Δηλαδή, σε κάθε εφαρμογή του κανόνα αυτού το μέγεθος της απόδειξης διπλασιάζεται.

Αφού ο κανόνας Velim είναι υπεύθυνος για την εκθετική έκρηξη του αλγορίθμου, είναι φυσικό το να σκεφτούμε ότι η απομάκρυνση του θα βελτιώσει την απόδοση του αλγορίθμου τόσο απο την άποψη του απαιτούμενου χρόνου όσο και από την άποψη του μεγέθους της απόδειξης.

Ωστόσο, η απομάκρυνση του Velim επηρεάζει την πληρότητα του αλγορίθμου και φυσικά του υποκείμενου αποδεικτικού συστήματος. Επομένως, δεν θα απομακρύνουμε απλά τον κανόνα Velim αλλά θα τον αντικαταστήσουμε με ένα σύνολο νέων παραγόμενων κανόνων οι οποίοι θα έχουν ως αποτέλεσμα πιο γρήγορη απόκριση του αλγορίθμου χωρίς να επηρεάζουν αρνητικά την πληρότητα και την ορθότητα των αποδείξεων που μπορεί να κατασκευάσει.

3.2 Η Βασική Ιδέα Του Βελτιωμένου Αλγορίθμου

Ο νέος αλγόριθμος αποτελεί παραλλαγή-βελτίωση του BFPS και επομένως, η βασική λογική είναι η ίδια. Ο αλγόριθμος αποτελείται από δύο βασικές φάσεις στις οποίες εφαρμόζονται ένα σύνολο από κανόνες και μετασχηματισμούς με σκοπό την απόδειξη ενός στόχου, δεδομένου ενός συνόλου υποθέσεων.

Η πρώτη φάση, η οποία και εδώ καλείται BR-φάση, αποσκοπεί στην εύρεση μιας ισοδύναμης μορφής της ζητούμενης απόδειξης έτσι ώστε ο στόχος να γίνει όσο πιο απλός γίνεται. Για τον σκοπό αυτό, χρησιμοποιούνται οι BR-κανόνες με τον ίδιο τρόπο που χρησιμοποιούνται και στον αλγόριθμο BFPS με μια μικρή παραλλαγή που θα δούμε στην συνέχεια.

Η δεύτερη φάση, η οποία και πάλι καλείται FR-φάση, αποσκοπεί στην εύρεση μια απόδειξης του απλοποιημένου στόχου. Για τον σκοπό αυτό, χρησιμοποιούνται οι FR-κανόνες που γνωρίσαμε στον αλγόριθμο BFPS εκτός από τον Velim ο οποίος όπως είπαμε είναι υπεύθυνος για την εκθετική έκρηξη και άρα, θα αντικατασταθεί από ένα σύνολο παραγόμενων κανόνων.

Όπως βλέπουμε, ο βελτιωμένος αλγόριθμος στη γενική του μορφή είναι ίδιος με τον BFPS με πολύ μικρές διαφορές οι οποίες ωστόσο, θα κάνουν την διαφορά.

3.3 BR-Φάση

Έστω ότι μας ζητείται μια απόδειξη $\Sigma \vdash G$, όπου Σ είναι ένα σύνολο αρχικών υποθέσεων και G είναι ένας τύπος-στόχος (απλός ή σύνθετος).

Κατά την BR-φάση, εκτελούνται οι γνωστοί μας BR-κανόνες με αποτέλεσμα να γίνουν μερικές νέες υποθέσεις όπως ορίζουν οι χρησιμοποιούμενοι κανόνες (βλέπε παράγραφο 2.2).

Η κύρια διαφορά από την BR-φάση του αλγορίθμου BFPS είναι ότι η BR-φάση εδώ τερματίζει όταν ο απλοποιημένος στόχος είναι το ⊥. Αυτή η αλλαγή κάνει πιο ορθογώνια την λειτουργία του αλγορίθμου καθώς, ο κανόνας PBC μπορεί πλέον να χρησιμοποιηθεί κατά την BR-φάση όπου είναι και φυσικό να χρησιμοποιηθεί.

Οι BR-κανόνες είναι οι εξής: **→intro**, **¬intro**, **∨intro**, **∧intro** και **PBC**.

Επομένως, στο τέλος της BR-φάσης, η απόδειξη $\Sigma \vdash G$ θα έχει μετασχηματιστεί σε μια φυσικά συνεπαγώμενη ζητούμενη απόδειξη $\Sigma_N \vdash \perp$ όπου στην ουσία είναι $\Sigma_N = \Sigma \cup \{\neg G\}$.

Η λειτουργία της BR-φάσης είναι το να θέσει μια ερώτηση αντιθετοαντιστροφής. Αν θέλουμε να αποδείξουμε ότι κάτι ισχύει υπό κάποιες συνθήκες τότε θα πρέπει να μην ισχύει η άρνηση του υπό τις ίδιες συνθήκες.

3.4 FR-Φάση

Κατά την FR-φάση του αλγορίθμου μπορούν να εφαρμοστούν οι βασικοί FR-κανόνες και οι γνωστοί μετασχηματισμοί που παρουσιάσαμε στο δεύτερο κεφάλαιο (βλέπε παραγράφους 2.3 και 2.4) εκτός φυσικά από τον κανόνα Velim για τους γνωστούς πλέον λόγους.

Οι βασικοί FR-κανόνες είναι οι εξής: **→elim**, **¬elim**, **∧elim**, **MT** και **¬¬elim**.

Προσοχή θέλει το γεγονός ότι από την στιγμή που πάντα στόχος είναι το ⊥(λόγω της εκτέλεσης της BR-φάσης), δεν υπάρχει η ανάγκη για χρήση του κανόνα ⊥elim.

Οι βασικοί μετασχηματισμοί είναι οι εξής: **∨to∧ DM**, **∧to∨ DM**, **→to∧ Transformation** και **→to∨ Transformation**.

Η απομάκρυνση του Velim, έχει ως αποτέλεσμα ένας τέτοιος αλγόριθμος να μην είναι πλήρης και επομένως, θα πρέπει να προσθέσουμε ένα νέο σύνολο παραγόμενων κανόνων με σκοπό να καλύψουμε αυτό το κενό.

Οι παραγόμενοι αυτοί κανόνες είναι οι Res και ¬toRes τους οποίους θα παρουσιάσουμε στην συνέχεια με λεπτομέρειες.

3.4.1 Ο Παραγόμενος Κανόνας Res

Αν σε δύο προς συνδυασμό γραμμές της μέχρι τώρα απόδειξης υπάρχουν οι τύποι $\boldsymbol{\varphi_1 \vee \varphi_2}$ και $\boldsymbol{\neg\varphi_1}$ τότε μπορεί να εφαρμοστεί ο παραγόμενος κανόνας Res με αποτέλεσμα την παραγωγή μιας νέας γραμμής στην απόδειξη η οποία θα περιέχει τον τύπο $\boldsymbol{\varphi_2}$.

Ο παραγόμενος κανόνας Res στην ουσία εκτελεί ένα βήμα επίλυσης (resolution) και γι' αυτό έχει και αυτό το όνομα.

Το νόημα του είναι ότι, σε περίπτωση όπου έχουμε έναν τύπο μορφής $\varphi_1 \vee \varphi_2$ τότε ξέρουμε ότι ισχύει είτε ο τύπος φ_1 είτε ο τύπος φ_2 είτε και οι δύο. Έτσι, σε περίπτωση όπου γνωρίζουμε ότι ισχύει $\neg\varphi_1$ τότε υποχρεωτικά πρέπει να ισχύει ο φ_2 και επομένως, μπορούμε να τον εξάγουμε ως συμπέρασμα.

Αυτή είναι η βασική λογική της επίλυσης η οποία σε πιο γενική μορφή αποτελεί από μόνη της ένα αποδεικτικό σύστημα. Εμείς εδώ, χρησιμοποιούμε βοηθητικά μια απλή μορφή της.

Ο Res είναι ορθός αφού μπορούμε εύκολα να βρούμε μια ND-απόδειξη για αυτόν στην οποία χρησιμοποιούμε τον κανόνα Velim ο οποίος γνωρίζουμε ότι είναι ορθός.

1. $\varphi_1 \vee \varphi_2$
2. $\neg\varphi_1$

3. φ_1	**Υπόθεση**
4. $\perp$	**¬elim 2,3**
5. φ_2	**⊥elim 47**

6. φ_2	**Υπόθεση**

7. φ_2 ∨elim 2

Απόδειξη Παραγόμενου
Κανόνα Res

Το γεγονός ότι χρησιμοποιούμε τον κανόνα ∨elim ώστε να αποδείξουμε τον κανόνα Res είναι μια καλή ένδειξη της πληρότητας του νέου μας αλγορίθμου.

3.4.2 Ο Παραγόμενος Κανόνας ¬toRes

Ο παραγόμενος κανόνας ¬toRes αποτελεί μια βοηθητική παραλλαγή του μετασχηματισμού Res και μπορεί να χρησιμοποιηθεί στην περίπτωση όπου η απόδειξη δεν μπορεί να συνεχίσει αλλά υπάρχει τουλάχιστον μια γραμμή με τύπο σε διαζευκτική μορφή $\boldsymbol{\varphi_1 \vee \varphi_2}$.

Τότε, με βάση αυτόν τον παραγόμενο κανόνα, μπορεί να ξεκινήσει μια νέα υπο-απόδειξη με βασική υπόθεση έναν από τους δύο τύπους (έστω τον φ_1) και στόχο το $\perp$. Αν η υπο-απόδειξη γίνει με επιτυχία τότε μπορούμε να συμπεράνουμε τον άλλο τύπο (έστω τον φ_2).

Η υπο-απόδειξη που γίνεται (αν φυσικά μπορεί να γίνει) στην ουσία αποδεικνύει ότι ισχύει ο τύπος $\neg\varphi_1$ μέσω του κανόνα ¬intro και άρα στην συνέχεια είναι μια απλή εφαρμογή του Res.

Η λειτουργία του παραγόμενου κανόνα συνοπτικά είναι η εξής:

Η απόδειξη $\boldsymbol{\Sigma, \varphi_1 \vee \varphi_2 \vdash \perp}$ με δεδομένο ότι $\boldsymbol{\Sigma, \varphi_1 \vdash \perp}$
ανάγεται στην απόδειξη $\boldsymbol{\Sigma, \varphi_1 \vee \varphi_2, \varphi_2 \vdash \perp}$.

Ο νέος αυτός κανόνας είναι ορθός καθώς μπορεί να αποδειχτεί εύκολα με χρήση του κανόνα ¬intro και του παραγόμενου κανόνα Res όπως σκιαγραφήσαμε λίγο πιο πάνω.

1. $\varphi_1 \vee \varphi_2$

B. | **φ_1** |
| ⋮ |
| $\perp$ |

2. $\neg\varphi_1$ ¬intro B
3. φ_2 Res 1,2

Απόδειξη Παραγόμενου
Κανόνα ¬to Res

Ο ¬toRes είναι ένας εξαιρετικά ενδιαφέρον κανόνας καθώς διαφέρει από όλους όσους έχουμε δει μέχρι στιγμής.

Πρόκειται για τον μοναδικό κανόνα ο οποίος κάνει μια υπο-απόδειξη η επιτυχία της οποίας δεν συνεπάγεται άμεσα και την επιτυχία της αρχικής απόδειξης. Ο μόνος λόγος για τον οποίο κάνει την υπο-απόδειξη αυτή είναι για να συνεχίσει με κάποιο τρόπο την απόδειξη. Μπορούμε να πούμε ότι ο ¬toRes είναι ένας κανόνας ο οποίος θέτει έναν ενδιάμεσο στόχο η απόδειξη του οποίου είναι απαραίτητη για την συνέχεια.

Στην χειρότερη περίπτωση, ο κανόνας ¬toRes θα λειτουργήσει ακριβώς όπως ο Velim. Στην πραγματικότητα ο ¬toRes πρόκειται για μια πρώτα κατά βάθος εκτέλεση του κανόνα Velim.

Ας αναλογιστούμε λιγάκι την λειτουργία του Velim και του ¬toRes. Ο Velim για την απόδειξη $\Sigma,\varphi_1 \vee \varphi_2 \vdash \bot$ απαιτεί να γίνουν με επιτυχία οι υπο-αποδείξεις $\Sigma,\varphi_1 \vdash \bot$ και $\Sigma,\varphi_2 \vdash \bot$. Ο κανόνας ¬toRes προσπαθεί να κάνει την υπο-απόδειξη $\Sigma,\varphi_1 \vdash \bot$ με σκοπό να βγάλει ως συμπέρασμα τον τύπο φ_2 και έπειτα να αποδείξει το $\bot$. Στην ουσία και οι δύο κανόνες απαιτούν ακριβώς το ίδιο πράγμα και επομένως είναι λογικό το νέο σύστημα να είναι επίσης πλήρες.

Η διαφορά των δύο αυτών κανόνων και το πλεονέκτημα του ¬toRes είναι η ύπαρξη του Res ο οποίος συνδυαστικά μπορεί να επιταχύνει την απόδειξη αν του δώσουμε προτεραιότητα κατά την εφαρμογή. Αυτό θα γίνει κατανοητό όταν θα εξετάσουμε την περίπτωση του 2-SAT όπου ο αλγόριθμος με την χρήση αυτών των δύο κανόνων μπορεί να αποδείξει πολύ πιο γρήγορα το ζητούμενο.

Χαρακτηριστικό των κανόνων ¬toRes και Res είναι το γεγονός ότι ο Res μπορεί να αποδείξει τον ¬toRes σε επίπεδο Natural Deduction και να παραχθεί από αυτόν κατά την εκτέλεση του αλγορίθμου.

3.5 Ο Συνολικός Βελτιωμένος Αλγόριθμος

Ο αλγόριθμος αποτελεί μια βελτίωση του BFPS που παρουσιάσαμε στο δεύτερο κεφάλαιο και για αυτόν τον λόγο θα τον ονομάσουμε BFPS+.

Στην συνέχεια θα παρουσιάσουμε συνοπτικά τον αλγόριθμο με τις διαφορές που έχει από τον BFPS που παρουσιάσαμε στο κεφάλαιο 2.

Βήμα 1: Αρχικός Έλεγχος.

Βήμα 2: Backward Reasoning Φάση.

Εδώ, η μόνη διαφορά από τον BFPS είναι η μεταφορά του PBC από την FR-φάση με σκοπό ο απλουστευμένος στόχος να είναι πάντα το $\bot$.

Βήμα 3: Forward Reasoning Φάση.

Εδώ, η κύρια διαφορά είναι η απομάκρυνση του κανόνα Velim και η εισαγωγή των κανόνων Res και ¬toRes. Οι δύο αυτοί κανόνες είναι οι τελευταίοι που θα προσπαθήσει ο αλγόριθμος να εφαρμόσει μετά και από τους μετασχηματισμούς δίνοντας προτεραιότητα στον Res.

Βήμα 4: Εγγραφή Στόχων Στην Απόδειξη.

Βήμα 5: Απομάκρυνση Περιττών Γραμμών.

3.6 Ορθότητα Και Πληρότητα Αλγορίθμου

3.6.1 Ορθότητα Αλγορίθμου

Θεώρημα. Αν ο αλγόριθμος BFPS+ μπορεί να βρει μια απόδειξη $\Sigma \vdash G$ τότε ισχύει $\Sigma \models G$.

Απόδειξη. Το θέωρημα ορθότητας του αλγορίθμου συνεπάγεται άμεσα από την ορθότητα των βασικών κανόνων αλλά και των δύο παραγόμενων κανόνων που προσθέσαμε αντί του Velim την οποία αποδείξαμε όταν τους παρουσιάσαμε. □

3.6.2 Πληρότητα Αλγορίθμου

Λήμμα 3. Αν κατά την FR-φάση του αλγορίθμου δεν αποδεικνύεται το ⊥ τότε η απόδειξη μπορεί να επεκταθεί σε ένα σύνολο Hintikka.

Απόδειξη. Θα αποδείξουμε το λήμμα 3 όπως αποδείξαμε και το λήμμα 1 όταν μελετήσαμε την πληρότητα του BFPS. Θα δείξουμε ότι αν ο αλγόριθμος μπορεί να κατασκευάσει μια απόδειξη έτσι ώστε κατά την FR-φάση να μην αποδεικνύεται το ⊥ τότε το σύνολο υποθέσεων μαζί με το σύνολο των παραγόμενων κανόνων που αποτελούν την απόδειξη μπορεί να επεκταθεί σε ένα ικανοποιήσιμο σύνολο Hintikka.

Οι ιδιότητες που πρέπει να ικανοποιεί ο αλγόριθμος είναι οι εξής:

(1) $\Sigma \vdash_{FR} \varphi \Rightarrow \Sigma \nvdash_{FR} \neg\varphi$

(2) $\Sigma \vdash_{FR} \neg\neg\varphi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$

(3) $\Sigma \vdash_{FR} \varphi\wedge\psi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\psi \nvdash_{FR} \bot$

(4) $\Sigma \vdash_{FR} \neg(\varphi\vee\psi) \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\neg\psi \nvdash_{FR} \bot$

(5) $\Sigma \vdash_{FR} \varphi\vee\psi \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\psi \nvdash_{FR} \bot$

(6) $\Sigma \vdash_{FR} \neg(\varphi\wedge\psi) \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\neg\psi \nvdash_{FR} \bot$

(7) $\Sigma \vdash_{FR} \varphi\rightarrow\psi \Rightarrow \Sigma,\neg\varphi \nvdash_{FR} \bot$ **ή** $\Sigma,\psi \nvdash_{FR} \bot$

(8) $\Sigma \vdash_{FR} \neg(\varphi\rightarrow\psi) \Rightarrow \Sigma,\varphi \nvdash_{FR} \bot$ **και** $\Sigma,\neg\psi \nvdash_{FR} \bot$

Οι αποδείξεις των ιδιοτήτων (1), (2), (3), (4) και (8) είναι ίδιες με τις αντίστοιχες του λήμματος 1 στο κεφάλαιο 2 (βλέπε παράγραφο 2.7.2).

Εμείς εδώ, θα αποδείξουμε βασικά την ιδιότητα (5) καθώς οι (6) και (7) στηρίζονται στην (5).

(5) Έστω $\Sigma \vdash_{FR} \varphi\vee\psi$ ενώ ισχύει $\Sigma,\varphi \vdash_{FR} \bot$ και $\Sigma,\psi \vdash_{FR} \bot$. Τότε, με την εφαρμογή του παραγόμενου κανόνα ¬toRes ο αλγόριθμος είναι σε θέση να αποδείξει κατά την FR-φάση το ⊥ καθώς από την γνώση των $\varphi\vee\psi$ και $\Sigma,\varphi \vdash_{FR} \bot$ μπορεί να αποδείξει το ψ και από την γνώση της απόδειξης $\Sigma,\psi \vdash_{FR} \bot$ μπορεί να αποδείξει και το ⊥. Επομένως, φτάσαμε σε άτοπο καθώς αρχίσαμε με την υπόθεση ότι ο αλγόριθμος κατά την FR-φάση δεν αποδεικνύει το ⊥. □

Λήμμα 4. Αν $\Sigma \models$ false τότε η FR-φάση του αλγορίθμου αποδεικνύει το ⊥.

Απόδειξη. Έστω ότι δεν ισχύει το λήμμα 4 και άρα, ο αλγόριθμος κατά την FR-φάση δεν θα αποδείξει το ⊥. Συνεπώς, από το λήμμα 3, έχουμε ότι το σύνολο αρχικών υποθέσεων Σ είναι ικανοποιήσιμο καθώς η απόδειξη που βρίσκει ο αλγόριθμος μπορεί να επεκταθεί σε σύνολο Hintikka. Αυτό όμως είναι άτοπο καθώς ισχύει $\Sigma \models$ false. □

Θεώρημα. Αν ισχύει Σ⊨G τότε ο αλγόριθμος BFPS+ μπορεί να βρει μια απόδειξη Σ⊢G.

Απόδειξη. Ο αλγόριθμος κατά την αναζήτηση της απόδειξης Σ⊢G, θα εκτελέσει την BR-φάση η οποία θα έχει ως αποτέλεσμα την αναγωγή της απόδειξης σε μια απλούστερη απόδειξη Σ'⊢⊥ όπου, το Σ' είναι ένα υπερσύνολο του Σ και πιο συγκεκριμένα είναι Σ'=Σ∩{¬G}.

Λόγω της ορθότητας της BR-φάσης ισχύει ότι αν Σ⊨G τότε Σ'⊨false και με βάση το λήμμα 4 ο αλγόριθμος είναι σε θέση να κάνει την απόδειξη Σ'⊢⊥. □

3.7 Ανάλυση Πολυπλοκότητας Στην Περίπτωση Του 2-SAT

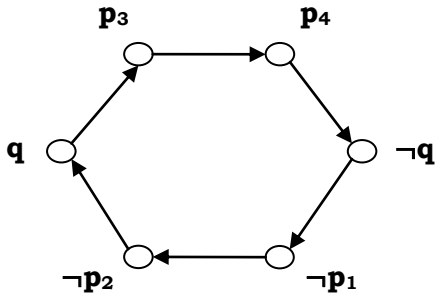
Ο αλγόριθμος BFPS είναι εκθετικός ακόμη και στην περίπτωση του 2-SAT (βλέπε παράγραφο 2.8). Ας δούμε πως συμπεριφέρεται ο βελτιωμένος μας αλγόριθμος.

Έστω το εξής παράδειγμα:

$$\neg q \vee p_1,\ \neg p_1 \vee p_2,\ \neg p_2 \vee \neg q,\ q \vee \neg p_3,\ p_3 \vee \neg p_4,\ p_4 \vee q \vdash \bot$$

Η απόδειξη μπορεί να γίνει καθώς, μπορούμε να φτιάξουμε ένα γράφημα εξαρτήσεων μεταξύ των λογικών τιμών που μπορούμε να δώσουμε στα προτασιακά γράμματα. Κάνουμε την εξής υπόθεση: Αν υπάρχει ένας τύπος **x∨y** και δώσουμε την τιμή false στο προτασιακό γράμμα x τότε για να ικανοποιηθεί ο τύπος αυτός θα πρέπει να δώσουμε την τιμή true στο προτασιακό γράμμα y. Επομένως, μπορούμε να κατασκευάσουμε ένα γράφημα εξαρτήσεων όπου αν στο x δώσουμε το false τότε πρέπει να δώσουμε false και στο ¬y.

Για το παράδειγμα μας, έχουμε το εξής γράφημα εξαρτήσεων:



Μπορούμε να παρατηρήσουμε ότι στον κύκλο που δημιουργήθηκε υπάρχει τόσο το q όσο και το ¬q με αποτέλεσμα, ο αντίστοιχος 2-CNF τύπος να μην μπορεί να ικανοποιηθεί και άρα θα πρέπει να υπάρχει μια συμβολική απόδειξη όπου οι προτάσεις αυτές αποδεικνύουν το ⊥.

Με την δημιουργία αυτού του γραφήματος εξαρτήσεων μπορούμε να δούμε αν ένας τύπος σε 2-CNF μπορεί να ικανοποιηθεί ή όχι σε γραμμικό χρόνο. Το μόνο που έχουμε να κάνουμε είναι να βρούμε τις ισχυρά συνεκτικές συνιστώσες. Αν σε κάθε συνεκτική συνιστώσα υπάρχει ένα ζεύγος αντίθετων προτασιακών γραμμάτων τότε ο τύπος είναι μη-ικανοποιήσιμος.

Να επισημάνουμε ότι για τύπους σε k-CNF με k>2 το γράφημα εξαρτήσεων δεν είναι εύκολο να δημιουργηθεί και επομένως δεν είναι εύκολη και η επίλυση του προβλήματος.

Ο αλγόριθμος BFPS θα αναγκαστεί να εφαρμόσει συνεχώς τον κανόνα Velim με αποτέλεσμα φυσικά την δημιουργία μιας εκθετικά μεγάλης απόδειξης. Ο βελτιωμένος αλγόριθμος BFPS+, μπορεί κατά κάποιον τρόπο να δημιουργήσει το γράφημα εξαρτήσεων και επομένως, να βρει μια μικρότερη συμβολική απόδειξη μέσω του κανόνα ¬toRes σε συνδυασμό με τον Res.

Η προτεραιότητα που δίνουμε στον κανόνα Res έχει αποτέλεσμα την αποφυγή της χειρότερης περίπτωσης όπου ο ¬toRes συμπεριφέρεται σαν τον Velim.

Στον παράδειγμα μας, ο αλγόριθμος θα καταλήξει στο συμπέρασμα ότι πρέπει να εφαρμόσει τον κανόνα ¬toRes για να μπορέσει να συνεχίσει. Επομένως, με βάση τον πρώτο διαζευκτικό τύπο θα προσπαθήσει να κάνει μια υπο-απόδειξη όπου υποθέτει το ¬q και θέτει ως στόχο το ⊥. Η συνέχεια είναι η εφαρμογή του Res ο οποίος επιτρέπει πολύ πιο γρήγορα την απόδειξη του ⊥ χωρίς να χρειαστεί να γίνει κάποια άλλη υπο-απόδειξη.

Πιο συγκεκριμένα, η απόδειξη που θα βρει ο BFPS+ είναι η εξής:

1. $\neg q \vee p_1$
2. $\neg p_1 \vee p_2$
3. $\neg p_2 \vee \neg q$
4. $q \vee \neg p_3$
5. $p_3 \vee \neg p_4$
6. $p_4 \vee q$

7. ¬q	**Υπόθεση**
8. $\neg p_3$	**Res 4,7**
9. $\neg p_4$	**Res 5,8**
10. q	**Res 6,9**
11. ⊥	**¬elim**

12. p_1 ¬toRes 1,{7,11}
13. p_2 Res 2,12
14. ¬q Res 3,13
15. $\neg p_3$ Res 4,14
16. $\neg p_4$ Res 5,14
17. q Res 6,16
18. ⊥ ¬elim 14,17

Η απόδειξη που βρίσκει ο αλγόριθμος BFPS+

Στο συγκεκριμένο παράδειγμα, η υπο-απόδειξη που γίνεται αντιστοιχεί στην διάσχιση μισού κύκλου επί του γραφήματος εξαρτήσεων για τον εντοπισμό δύο αντικρουόμενων προτασιακών γραμμάτων. Ο μισός κύκλος γίνεται επειδή τυχαίνει η πρώτη πρόταση η οποία οδηγεί στην εφαρμογή του ¬toRes να περιέχει ένα από τα δύο αντικρουόμενα προτασιακά γράμματα. Σε άλλη περίπτωση χρειάζεται να γίνει το πολύ ένας ολόκληρος κύκλος.

Αφού γίνει η υπο-απόδειξη τότε απαιτείται η διάσχιση το πολύ ολόκληρου του κύκλου με την συνεχόμενη εφαρμογή του Res ώστε να εντοπιστούν και πάλι τα αντικρουόμενα προτασιακά γράμματα και να αποδειχτεί το ⊥.

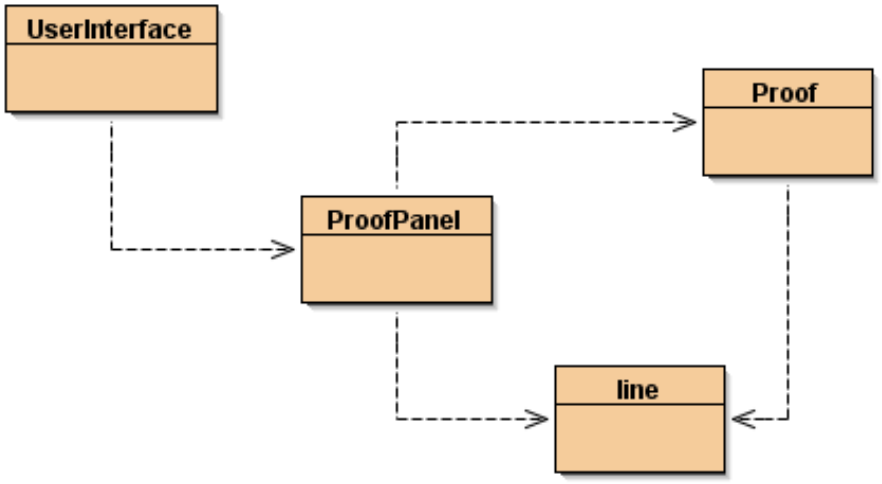
Συνεπώς, ο αλγόριθμος BFPS+ είναι το πολύ 2 φορές χειρότερος από τον καλύτερο γραμμικό αλγόριθμο το οποίο σημαίνει ότι και αυτός να είναι γραμμικός.

Κεφάλαιο 4
Σχολιασμός Υλοποίησης

4.1 Εισαγωγικά Στοιχεία

Η υλοποίηση του αλγορίθμου BFPS τον οποίο παρουσιάσαμε στο κεφάλαιο 2 πραγματοποιήθηκε με την γλώσσα προγραμματισμού JAVA η οποία διευκόλυνε κατά πολύ την δημιουργία μιας γραφικής διεπαφής μέσω της οποίας ο χρήστης μπορεί να εισάγει είσοδο και επίσης να δει την απόδειξη που παράγει ο αλγόριθμος.

Η υλοποίηση βασίστηκε σε μια αντικειμενοστρεφή προσέγγιση και οι βασικές κλάσεις και το πως αυτές αλληλεπιδρούν μεταξύ τους φαίνονται στο παρακάτω διάγραμμα.



Η κλάση UserInterface δημιουργεί το βασικό παράθυρο το οποίο περιέχει έναν υποδοχέα ο οποίος είναι αντικείμενο της κλάσης ProofPanel και περιέχει όλα τα γραφικά συστατικά της διεπαφής ενώ είναι και υπεύθυνος για τον έλεγχο της εισόδου.

Μια απόδειξη είναι μια ακολουθία από γραμμές οι οποίες περιέχουν αριθμούς, τύπους καθώς και ειδικά σχόλια. Κάθε τέτοια γραμμή είναι ένα αντικείμενο της κλάσης line.

Τέλος, η κλάση Proof αποτελείται από ένα αρκετά μεγάλο σύνολο μεθόδων οι οποίες μπορούν να συνδυάστούν ώστε να υλοποιήσουν τον αλγόριθμο BFPS όπως τον έχουμε γνωρίσει.

Η ροή του προγράμματος είναι η εξής:

Ο χρήστης τρέχει το πρόγραμμα δημιουργώντας ένα αντικείμενο της κλάσης UserInterface η οποία με την σειρά της δημιουργεί ένα αντικείμενο της κλάσης ProofPanel ώστε να δημιουργηθεί η γραφική διεπαφή.

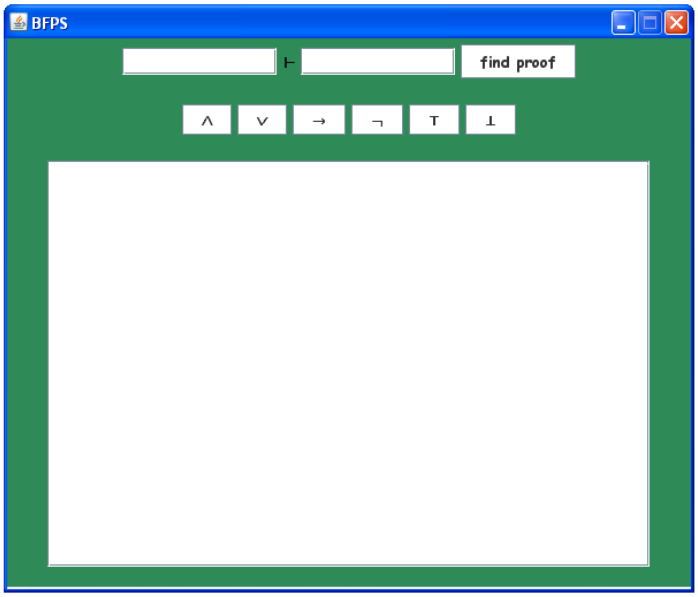
Μέσω της γραφικής διεπαφής, ο χρήστης μπορεί να δώσει ως είσοδο ένα σύνολο προτασιακών τύπων οι οποίοι αποτελούν τις υποθέσεις της ζητούμενης απόδειξης καθώς και έναν στόχο.

Έπειτα, γίνεται έλεγχος για το αν οι τύποι που έχουν δωθεί είναι σωστοί με βάση την γραμματική της προτασιακής λογικής και αν είναι τότε δημιουργείται ένα αντικείμενο Proof και άρα, τρέχει ο αλγόριθμος BFPS για την εύρεση της απόδειξης.

Αφού τρέξει ο αλγόριθμος, η απόδειξη είναι αποθηκευμένη σε ένα διάνυσμα αντικειμένων της κλάσης line και έπειτα εμφανίζεται στον χρήστη μέσω της γραφικής διεπαφής.

4.2 Η Γραφική Διεπαφή

Η γραφική διεπαφή του προγράμματος φαίνεται στην επόμενη εικόνα.

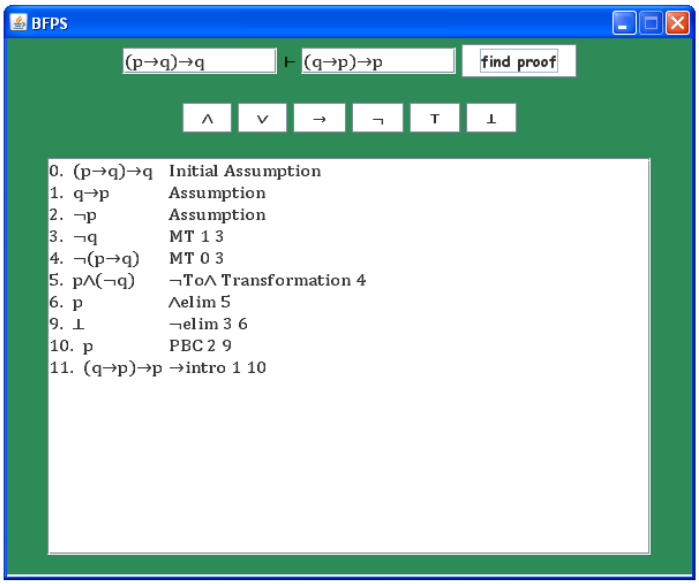


Έχουμε δύο πεδία κειμένου τα οποία χωρίζονται από το ειδικό σύμβολο απόδειξης ⊢. Έτσι, ο χρήστης στο πρώτο πεδίο μπορεί να εισάγει το σύνολο υποθέσεων και στο δεύτερο μπορεί να εισάγει τον στόχο.

Για την δημιουργία των λογικών τελεστών, ο χρήστης μπορεί να χρησιμοποιήσει τα αντίστοιχα κουμπιά που υπάρχουν κάτω από τα πεδία κειμένου υπόθεσης και στόχου.

Αφού ο χρήστης εισάγει τις υποθέσεις και τον στόχο, μπορεί να πατήσει το κουμπί εύρεσης απόδειξης ώστε να ξεκινήσει η λειτουργία του αλγορίθμου και να παραχθεί η ζητούμενη απόδειξη η οποία θα εμφανιστεί στην μεγάλη περιοχή κειμένου που υπάρχει ακριβώς κάτω από τα κουμπιά λογικών τελεστών.

Για παράδειγμα, για την απόδειξη (p→q)→q⊢(q→p)→p έχουμε το εξής αποτέλεσμα:



4.3 Αναπαράσταση Απόδειξης

Μια απόδειξη είναι μια ακολουθία γραμμών και επομένως μπορεί να αναπαρασταθεί με ένα διάνυσμα αντικειμένων τύπου line.

Κάθε αντικείμενο της κλάσης line διαθέτει τα εξής χαρακτηριστικά:

- Έναν αριθμό με βάση τον οποίο διακρίνονται οι διάφορες γραμμές της απόδειξης.
- Ένα προτασιακό τύπο.
- Ένα σχόλιο που εξηγεί το πώς αποδείχτηκε ο τύπος που περιέχει η γραμμή.
- Το πολύ 5 ακόμη αριθμούς που δείχνουν ποιες προηγούμενες γραμμές της απόδειξης χρησιμοποιήθηκαν ώστε να αποδειχτεί ο τύπος.
- Μια λογική μεταβλητή η οποία δείχνει αν η συγκεκριμένη γραμμή είναι χρήσιμη στην απόδειξη του στόχου και άρα, το αν πρέπει να συμπεριληφθεί στην απόδειξη κατά την απομάκρυνση περιττών γραμμών.
- Μια λογική μεταβλητή η οποία δείχνει αν κατά τις αναδρομικές κλήσεις έχει εκτελεστεί ο κανόνας Velim σε περίπτωση φυσικά που ο τύπος είναι σε διαζευκτική μορφή.

Επίσης, κάθε αντικείμενο της κλάσης line περιέχει και ένα σύνολο μεθόδων μέσω των οποίων γίνεται ανάθεση και ανάκτηση τιμής των χαρακτηριστικών.

4.4 Υλοποίηση Του Αλγορίθμου

Η υλοποίηση του αλγορίθμου είναι εμφωλευμένη στην υλοποίηση της κλάσης Proof.

Η κλάσης Proof αποτελείται από ένα σύνολο μεθόδων οι οποίες μεταξύ άλλων υλοποιούν τους κανόνες του συστήματος, την BR-φάση, την FR-φάση και την βελτιστοποίηση της απόδειξης.

Κάθε απόδειξη είναι αντικείμενο της κλάσης Proof και έχει τα εξής χαρακτηριστικά:

- Ένα διάνυσμα αντικειμένων line όπου αποθηκεύεται η απόδειξη.
- Μια στοίβα γραμμών η οποία αναπαριστά την στοίβα στόχων.
- Μια δομή διανυσμάτων αντικειμένων line η οποία αναπαριστά τα επίπεδα υποθέσεων.
- Μια μεταβλητή η οποία ορίζει ποιο είναι το επόμενο επίπεδο υποθέσεων.
- Μια μεταβλητή η οποία δείχνει το σημείο της απόδειξης από όπου θα πρέπει να γίνει έλεγχος για την εφαρμογή κάποιου μετασχηματισμού.

Ο δημιουργός της κλάσης Proof είναι αυτός ο οποίος καλεί με κατάλληλη σειρά τις μεθόδους με τις οποίες υλοποιούνται οι BR και FR φάσεις. Με την σειρά τους αυτές οι μέθοδοι καλούν τις κατάλληλες μεθόδους οι οποίες υλοποιούν τους κανόνες και τους μετασχηματισμούς του συστήματος όπως παρουσιάσαμε στο κεφάλαιο 2.

Όταν επιστρέψει ο δημιουργός της κλάσης στην ουσία έχει τερματίσει η λειτουργία του BFPS και η απόδειξη είναι αποθηκευμένη στο κατάλληλο διάνυσμα αντικειμένων αν φυσικά είναι επιτυχής. Αν η απόδειξη έχει αποτύχει τότε το διάνυσμα αυτό είναι κενό.

Ο δημιουργός τροφοδοτείται με αρχικές τιμές για όλα τα χαρακτηριστικά που αναφέρθηκαν παραπάνω για την πιο εύκολη εύρεση υπο-αποδείξεων. Μια υπο-απόδειξη είναι αντικείμενο της ίδιας κλάσης και αυτό προκαλεί αναδρομική δημιουργία αντικειμένου μέσα στην κλάση.

Σχολιασμένη Βιβλιογραφία

Huth, Ryan. Logic In Computer Science.

Ένα βιβλίο σχετικό με το αποδεικτικό σύστημα Natural Deduction. Παρουσιάζει τους κανόνες του συστήματος τόσο για την προτασιακή λογική όσο και για την λογική πρώτης τάξης ενώ περιέχει και ένα μεγάλο σύνολο παραδειγμάτων και ασκήσεων. Επίσης, περιέχει αποδείξεις της ορθότητας και της πληρότητας του συστήματος καθώς και αλγορίθμους επίλυσης του SAT για ειδικές περιπτώσεις.

Fitting. First-Order Logic And Automated Theorem Proving.

Ένα γενικό βιβλίο για την λογική και τον αυτόματο συλλογισμό με αποδεικτικά συστήματα. Ορίζει ρητά τόσο την προτασιακή λογική όσο και την λογική πρώτης τάξης ενώ περιγράφει τα αποδεικτικά συστήματα Tableau, Resolution και Natural Deduction. Περιέχει και μερικές γενικές μεθοδολογίες για την απόδειξη της πληρότητας ενός αποδεικτικού συστήματος όπως τα σύνολα Hintikka και οι ιδιότητες συνέπειας.

Bolotov, Bocharov, Gorchakov, Shangin. Automated First Order Natural Deduction.

Μια εργασία η οποία παρουσιάζει έναν αλγόριθμο εύρεσης αποδείξεων για την λογική πρώτης τάξης βασισμένο στο αποδεικτικό σύστημα Natural Deduction. Περιγράφει αλγοριθμικά τις ιδέες της προς-τα-πίσω και προς-τα-εμπρός ανάλυσης οι οποίες παρουσιάστηκαν και σε αυτή την διπλωματική εργασία με λίγο διαφορετικό τρόπο. Η εργασία περιλαμβάνει αποδείξεις για την ορθότητα και την πληρότητα του αλγορίθμου με χρήση των συνόλων Hintikka.

Russel, Norvig. Τεχνητή Νοημοσύνη, Μια Σύγχρονη Προσέγγιση.

Ένα βιβλίο σχετικό με την τεχνητή νοημοσύνη. Περιέχει αλγορίθμους αναζήτησης, γενετικούς αλγορίθμους, νευρωνικά δίκτυα, θέματα ρομποτικής καθώς και το πως η μαθηματική λογική και ο αυτόματος συλλογισμός μπορούν να χρησιμοποιηθούν στην πράξη μέσω της έννοιας του λογικού πράκτορα.

Sipser. Εισαγωγή Στην Θεωρία Υπολογισμού.

Ένα βιβλίο σχετικό με την θεωρία υπολογισμού. Περιγράφει τα βασικά μοντέλα υπολογισμού τα οποία είναι τα πεπερασμένα αυτόματα, τα αυτόματα στοίβας και οι μηχανές turing καθώς και τις κλάσεις πολυπλοκότητας P και NP. Περιέχει αποδείξεις NP-πληρότητας για σημαντικά προβλήματα μεταξύ των οποίων περιέχεται και το SAT το οποίο αποτελεί το πρώτο πρόβλημα που αποδείχτηκε NP-πλήρες.

Dasgupta, Papadimitriou, Vazirani. Αλγόριθμοι.

Ένα βιβλίο σχετικό με σχεδιασμό και ανάλυση αλγορίθμων. Παρουσιάζει γενικές και ειδικές μεθοδολογίες σχεδιασμού αλγορίθμων σχετικά με την συνδυαστική και την θεωρία γραφημάτων. Επεξηγεί με απλό τρόπο τον αλγόριθμο αναζήτησης βάθους και τον αλγόριθμο εύρεσης των συνεκτικών συνιστωσών ενός κατευθυνόμενου γραφήματος. Τέλος, περιέχει μια εισαγωγή στην NP-πληρότητα και στο πως μπορούμε να την αντιμετωπίσουμε με ευρετικές μεθόδους και προσεγγιστικούς αλγορίθμους.

Παράρτημα
Κώδικας Υλοποίησης

Π.1 Η Κλάση UserInterface

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class UserInterface extends JFrame{
   ProofPanel proofPanel;
   public UserInterface(){ ... }
   public static void main(String){ ... }
}

public UserInterface(){
   super("BFPS");
   try{
      proofPanel=new ProofPanel();
      add(proofPanel);
      setSize(600,500);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
      setResizable(false);
   }
   catch (Exception e){}
}

public static void main(String []args){
   UserInterface UI=new UserInterface();
}
```

Π.2 Η Κλάση line

```
public class line{
   int number;
   String formula, comment;
   int pline0, pline1, pline2, pline3, pline4;
   boolean isUsed, orUsed;
   public line(){ ... }
   public line(String){ ... }
   public line(int,String,Strin,int,int){ ... }
   public void setNumber(int) { ... }
   public int getNumber(){ ... }
   public void setFormula(String) { ... }
   public String getFormula(){ ... }
   public void setComment(String) { ... }
   public String getComment(){ ... }
   public void setPLine(int) { ... }
   public void setPLine0(int) { ... }
   public int getPLine0(){ ... }
   public void setPLine1(int){ ... }
   public int getPLine1(){ ... }
   public void setPLine2(int) { ... }
   public int getPLine2(){ ... }
   public void setPLine3(int) { ... }
```

```
   public int getPLine3(){ ... }
   public void setPLine4(int){ ... }
   public int getPLine4(){ ... }
   public void setUsed(){ ... }
   public boolean getUsed(){ ... }
   public void setOr(){ ... }
   public boolean getOr(){ ... }
}
public line(){
   number=-1;
   formula=new String();
   comment=new String();
   pline0=pline1=pline2=pline3=pline4=-1;
   isUsed=false;
   orUsed=false;
}
public line(String formula){
   number=-1;
   this.formula=formula;
   comment=new String();
   pline0=pline1=pline2=pline3=pline4=-1;
   isUsed=false;
   orUsed=false;
}
public line(int number,String formula,String comment){
   this.number=number;
   this.formula=formula;
   this.comment=comment;
   pline0=pline1=pline2=pline3=pline4=-1;
   isUsed=false;
   orUsed=false;
}
public line(int number,String formula,String comment,int a,int b){
   this.number=number;
   this.formula=formula;
   this.comment=comment;
   pline0=a; pline1=b;
   pline2=pline3=pline4=-1;
   isUsed=false;
   orUsed=false;
}
public void setNumber(int number){
   this.number=number;
}
public int getNumber(){
   return number;
}
public void setFormula(String formula){
   this.formula=formula;
}
```

```
public String getFormula(){
   return formula;
}
public void setComment(String comment){
   this.comment=comment;
}
public String getComment(){
   return comment;
}
public void setPLine(int pline){
   if (pline0==-1) pline0=pline;
   else if (pline1==-1) pline1=pline;
   else if (pline2==-1) pline2=pline;
   else if (pline3==-1) pline3=pline;
   else if (pline4==-1) pline4=pline;
}
public void setPLine0(int pline){
   pline0=pline;
}
public int getPLine0(){
   return pline0;
}
public void setPLine1(int pline){
   pline1=pline;
}
public int getPLine1(){
   return pline1;
}
public void setPLine2(int pline){
   pline2=pline;
}
public int getPLine2(){
   return pline2;
}
public void setPLine3(int pline){
   pline3=pline;
}
public int getPLine3(){
   return pline3;
}
public void setPLine4(int pline){
   pline4=pline;
}
public int getPLine4(){
   return pline4;
}
public void setUsed(){
   isUsed=true;
}
```

```
public boolean getUsed(){
   return isUsed;
}
public void setOr(){
   orUsed=true;
}
public boolean getOr(){
   return orUsed;
}
```

Π.3 Η Κλάση ProofPanel

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class ProofPanel extends JPanel{
   // Το Background χρώμα
   Color color=new Color(0x2E8B57);
   // Οι χρησιμοποιούμενες γραμματοσειρές
   Font font1=new Font("Cambria Math",Font.BOLD,15);
   Font font2=new Font("Cambria Math",Font.BOLD,13);
   // Λογικές μεταβλητές για να ξέρουμε ποιο συστατικό έχασε εστιάση
   boolean hypothesis_focus=false,theorem_focus=false;
   // Συστατικά για το πρώτο υπο-πάνελ στο οποίο ορίζεται η απόδειξη
   JTextField hypothesis=new JTextField(10);
   JLabel proof_symbol=new JLabel("\u22a2");
   JTextField theorem=new JTextField(10);
   JButton find_proof=new JButton("find proof");
   JPanel subPanel1=new JPanel();
   /* Συστατικά για το δεύτερο υπο-πάνελ στο οποίο έχουμε κάποια κουμπιά για
      την εύκολη εισαγωγή των λογικών τελεστών */
   JButton and=new JButton("\u2227");          // AND
   JButton or=new JButton("\u2228");           // OR
   JButton implies=new JButton("\u2192");      // IMPLIES
   JButton not=new JButton("\u00ac");          // NOT
   JButton true_value=new JButton("\u22a4");   // TRUE
   JButton false_value=new JButton("\u22a5");  // BOTTOM
   JPanel subPanel2=new JPanel();
   // Συστατικά για το τρίτο υπο-πάνελ στο οποίο θα εμφανίζεται η απόδειξη
   JTextArea proof_area=new JTextArea(18,40);
   JScrollPane proof=new JScrollPane(proof_area);
   JPanel subPanel3=new JPanel();
   public ProofPanel() throws Exception{ ... }
   class FocusOperator extends FocusAdapter{
       public void focusLost(FocusEvent){ ... }
   }
   class MouseOperator extends MouseAdapter{
       public void mouseClicked(MouseEvent){ ... }
       void write_textfield(String){ ... }
       Vector<line> AnalyzeHypothesis(String){
```

```
        boolean checkFormula(String){ ... }
        void create_proof(){ ... }
    }
}
public ProofPanel() throws Exception{
    super();
    // Κατασκευή του πρώτου υπο-πάνελ όπου ορίζεται η απόδειξη
    proof_symbol.setForeground(new Color(0x0c0c0c));
    proof_symbol.setFont(font1);
    hypothesis.setFont(font1);
    theorem.setFont(font1);
    find_proof.setFont(new Font("Comic Sans MS",Font.BOLD,13));
    find_proof.setBackground(Color.white);
    hypothesis.addFocusListener(new FocusOperator());
    theorem.addFocusListener(new FocusOperator());
    find_proof.addMouseListener(new MouseOperator());
    subPanel1.add(hypothesis);
    subPanel1.add(proof_symbol);
    subPanel1.add(theorem);
    subPanel1.add(find_proof);
    // Κατασκευή του δεύτερου υπο-πάνελ όπου γίνεται επιλογή λογικών τελεστών
    and.setFont(font2);
    or.setFont(font2);
    implies.setFont(font2);
    not.setFont(font2);
    true_value.setFont(font2);
    false_value.setFont(font2);
    and.setBackground(Color.white);
    or.setBackground(Color.white);
    implies.setBackground(Color.white);
    not.setBackground(Color.white);
    true_value.setBackground(Color.white);
    false_value.setBackground(Color.white);
    and.addMouseListener(new MouseOperator());
    or.addMouseListener(new MouseOperator());
    implies.addMouseListener(new MouseOperator());
    not.addMouseListener(new MouseOperator());
    true_value.addMouseListener(new MouseOperator());
    false_value.addMouseListener(new MouseOperator());
    subPanel2.add(and);
    subPanel2.add(or);
    subPanel2.add(implies);
    subPanel2.add(not);
    subPanel2.add(true_value);
    subPanel2.add(false_value);
    // Κατασκευή του τρίτου υπο-πάνελ όπου εμφανίζεται η απόδειξη
    proof_area.setFont(font1);
    subPanel3.add(proof);
    // Τα υπο-πάνελς θα είναι το ένα πάνω από το άλλο
    setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));
```

```
   // Τοποθέτηση των υπο-πάνελ στο κεντρικό πάνελ
   add(subPanel1);
   add(subPanel2);
   add(subPanel3);
   // Background χρωματισμός των υπο-πάνελ
   subPanel1.setBackground(color);
   subPanel2.setBackground(color);
   subPanel3.setBackground(color);
}
public void focusLost(FocusEvent event){
   if (event.getSource()==hypothesis){
      hypothesis_focus=true;
      theorem_focus=false;
   }
   else if (event.getSource()==theorem){
      hypothesis_focus=false;
      theorem_focus=true;
   }
}
public void mouseClicked(MouseEvent event){
   if (event.getSource()==and) write_textfield("\u2227");
   else if (event.getSource()==or) write_textfield("\u2228");
   else if (event.getSource()==implies) write_textfield("\u2192");
   else if (event.getSource()==not) write_textfield("\u00ac");
   else if (event.getSource()==true_value) write_textfield("\u22a4");
   else if (event.getSource()==false_value) write_textfield("\u22a5");
   else if (event.getSource()==find_proof){
      proof_area.setText("");
      create_proof();
   }
}
void write_textfield(String text){
   String s;
   if (hypothesis_focus){
      s=hypothesis.getText()+text;
      hypothesis.setText(s);
      hypothesis.requestFocus();  // Ξανα-πάρε την εστίαση
   }
   else if (theorem_focus){
      s=theorem.getText()+text;
      theorem.setText(s);
      theorem.requestFocus();    // Ξανα-πάρε την εστίαση
   }
}
Vector<line> AnalyzeHypothesis(String H){
   // Δημιούργησε ένα διάνυσμα υποθέσεων
   Vector<line> HV=new Vector<line>();
   // Το String των αρχικών υποθέσεων αναλύεται σε tokens με βάση το κόμμα
   StringTokenizer tokens=new StringTokenizer(H,",",false);
   // Κάθε token-υπόθεση είναι και μια γραμμή στην απόδειξη
```

```
   while (tokens.hasMoreTokens()){
      // Πάρε το επόμενο token
      String s=tokens.nextToken();
      // Πρόσθεσε στο διάνυσμα υποθέσεων την νέα υπόθεση
      HV.add(new line(HV.size(),s,"Initial Assumption"));
   }
   // Επιστροφή του διανύσματος υποθέσεων
   return HV;
}
boolean checkFormula(String formula){
   int counter=0;   // Μετρητής παρενθέσεων
   boolean OK=true; // Μεταβλητή ελέγχου
   // Αν ο τύπος είναι μέσα σε παρενθέσεις τότε
   if (formula.startsWith("(") && formula.endsWith(")")){
     // Αν ενδιάμεσα δεν υπάρχουν παρενθέσεις τότε λάθος
     if (formula.length()>0 && formula.indexOf("(",1)==-1) OK=false;
   }
   if (OK){
     // Διάτρεξη του τύπου
     for (int i=0; i<formula.length(); i++){
        // Αν βρήκες αριστερή παρένθεση τότε αύξησε τον μετρητή
        if (formula.charAt(i)=='(') ++counter;
        // Αλλιώς αν βρήκες δεξιά παρένθεση τότε
        else if (formula.charAt(i)==')'){
           // Αν ο μετρητής είναι μηδέν τότε έχουμε ) πριν από ( άρα λάθος
           if (counter==0) OK=false;
           // Αν προηγείται ( τότε έχουμε συνδυασμό () που δεν έχει νόημα
           if (i-1>=0 && formula.charAt(i-1)=='(') OK=false;
           // Αλλιώς μείωσε κατά ένα τον μετρητή
           else --counter;
        }
        // Αλλιώς αν βρήκες κάποιο συνδετικό τότε
        else if (formula.charAt(i)=='\u2227' || formula.charAt(i)=='\u2228' ||
                 formula.charAt(i)=='\u2192' || formula.charAt(i)=='\u00ac')
        {
           // Αν είναι στην αρχή της φόρμουλας και διαφορετικό από not τότε λάθος
           if (i==0 && formula.charAt(i)!='\u00ac') OK=false;
           // Αν είναι η φόρμουλα είναι ένα σκέτο not τότε λάθος
           if  (i==0  &&  formula.charAt(i)=='\u00ac'  &&  i==formula.length()-1){
               OK=false;
           }
           // Αν είναι στο τέλος της φόρμουλας τότε λάθος
           if (i==formula.length()-1) OK=false;
           // Αν ακολουθεί και πάλι κάποιο συνδετικό τότε λάθος
           if (i+1<formula.length() &&
              (formula.charAt(i+1)=='\u2227' || formula.charAt(i+1)=='\u2228' ||
               formula.charAt(i+1)=='\u2192' || formula.charAt(i+1)=='\u00ac'))
           { OK=false; }
           // Αν ακολουθεί δεξιά παρένθεση τότε λάθος
```

```
            if (i+1<formula.length() && formula.charAt(i+1)==')') OK=false;
            // Αν προηγείται αριστερή παρένθεση και δεν είναι το not τότε λάθος
            if (i>0 && formula.charAt(i-1)=='(' && formula.charAt(i)!='\u00ac')
            { OK=false; }
         }
         // Διαφορετικά προτασιακό γράμμα
         else{
            // Αν δεν ακολουθεί συνδετικό ή δεξιά παρένθεση τότε λάθος
            if (i+1<formula.length() && formula.charAt(i+1)!='\u2227' &&
                formula.charAt(i+1)!='\u2228' && formula.charAt(i+1)!='\u2192'
                && formula.charAt(i+1)!='\u00ac' && formula.charAt(i+1)!=')')
            { OK=false; }
            /* Αν προηγείται αριστερή παρένθεση και ακολουθεί δεξιά παρένθεση τότε
               δεν έχει νόημα - λάθος (φ) */
            if (i-1>=0 && i+1<formula.length() && formula.charAt(i-1)=='('
                && formula.charAt(i+1)==')')
            { OK=false; }
         }
         // Αν βρέθηκε λάθος τότε τερματισμός
         if (!OK) return false;
      }
   }
   // Αν έχουμε μετρητή διάφορο του μηδενός η το OK είναι false τότε σφάλμα
   if (counter!=0 || !OK) return false;
   // Μόνο αν βγει από το for loop είναι όλα εντάξει με τον τύπο
   return true;
}
void create_proof(){
   // Ανάκτηση των αρχικών υποθέσεων
   String H=hypothesis.getText();
   // Δημιουργία ενός διανύσματος υποθέσεων
   Vector<line> HV=AnalyzeHypothesis(H);
   // Διάτρεξη του διανύσματος υποθέσεων
   for (int i=0; i<HV.size(); i++){
      String formula=(String)HV.get(i).getFormula();
      // Αν υπάρχει συντακτικό λάθος σε κάποια υπόθεση τότε
      if (!checkFormula(formula)){
         // Εμφάνιση κατάλληλου μηνύματος λάθους
         JOptionPane.showMessageDialog(null,"Syntax error in formula "+formula,
                                       "Error!",JOptionPane.WARNING_MESSAGE);
         return;  // Επιστροφή - Δεν προχωράμε στην απόδειξη
      }
   }
   // Ανάκτηση του στόχου
   String G=theorem.getText();
   // Αν ο στόχος δεν υπάρχει τότε λάθος
   if (G.trim().compareTo("")==0){
      // Εμφάνιση κατάλληλου μηνύματος λάθους
      JOptionPane.showMessageDialog(null,"Proof must have a goal!",
                                    "Error!",JOptionPane.WARNING_MESSAGE);
```

```
      return;  // Επιστροφή - Δεν προχωράμε στην απόδειξη
   }
   // Αν υπάρχει συντακτικό λάθος στον στόχο τότε
   if (!checkFormula(G)){
      // εμφάνιση κατάλληλου μηνύματος λάθους
      JOptionPane.showMessageDialog(null,"Syntax error in formula "+G,
                                    "Error!",JOptionPane.WARNING_MESSAGE);
      return;  // Επιστροφή - Δεν προχωράμε στην απόδειξη
   }
   // Δημιουργία ενός αντικειμένου απόδειξης
   Proof proof=new Proof(HV,G,new Vector<Vector<line>>(),0,0);
   // Αν η απόδειξη είναι κενή τότε απτέτυχε
   if (proof.P.size()==0){
      // εμφάνιση κατάλληλου μηνύματος λάθους
      JOptionPane.showMessageDialog(null,"Proof failure!",
                                    "Failure!",JOptionPane.WARNING_MESSAGE);
      return;  // Επιστροφή - Δεν προχωράμε στην εμφάνιση της απόδειξης
   }
   // Η απόδειξη είναι αποθηκευμένη στο διάνυσμα P του αντικειμένου proof
   for (int i=0; i<proof.P.size(); i++){
      line L=(line)proof.P.get(i);
      // Αν η γραμμή χρησιμοποιείται στην απόδειξη τότε εμφάνιση της
      if (L.getUsed()){
         String s;
         if (L.getPLine0()==-1){
            s=L.getNumber()+".  "+L.getFormula()+"\t"+L.getComment()+"\n";
         }
         else if (L.getPLine1()==-1){
            s=L.getNumber()+".  "+L.getFormula()+"\t"+L.getComment()+
              " "+L.getPLine0()+"\n";
         }
         else if (L.getPLine2()==-1){
            s=L.getNumber()+".  "+L.getFormula()+"\t"+L.getComment()+
              " "+L.getPLine0()+" "+L.getPLine1()+"\n";
         }
         else{
            s=L.getNumber()+".  "+L.getFormula()+"\t"+L.getComment()+" "+
              L.getPLine0()+" "+L.getPLine1()+" "+L.getPLine2()+
              " "+L.getPLine3()+" "+L.getPLine4()+"\n";
         }
         proof_area.append(s);
      }
   }
}
```

Π.4 Η Κλάση Proof

```
import java.util.*;
public class Proof{
   public Vector<line> P=new Vector<line>();
   Stack<line> S=new Stack<line>();
   Vector<Vector<line>> L=new Vector<Vector<line>>();
```

```
   int next_level;
   int start_of_subroutine;
   int start_of_impliesToOr;
   public Proof(Vector<line>,String,Vector<Vector<line>>,int,int){ ... }
   public void optimization(){ ... }
   Vector<String> AnalyzeFormula(String){ ... }
   boolean isGoal(line,boolean) { ... }
   int checkAssumptionsForGoal(boolean){ ... }
   void copy(line){ ... }
   void finishProof(){ ... }
   void BR(){ ... }
   line impliesIntro(String,String){ ... }
   line notIntro(String){ ... }
   line orIntro(String,String){ ... }
   void andIntro(String,String){ ... }
   void updateHyperGoal(String){ ... }
   void updateHyperGoalAndIntro(int,int){ ... }
   void FR(){ ... }
   void initializeLevels(){ ... }
   boolean firstCombination(){ ... }
   boolean nextCombination(){ ... }
   Vector<line> combineLevels(Vector<line>,Vector<line>,boolean){ ... }
   boolean impliesElim(line,line){ ... }
   boolean notElim(line,line){ ... }
   boolean andElim(line){ ... }
   boolean bottomElim(line){ ... }
   boolean notnotElim(line){ ... }
   boolean modusTolens(line,line){ ... }
   boolean PBC(){ ... }
   boolean orElim(String,String,int){ ... }
   void updateGoalOrElim(int,int,int,int,int){ ... }
   boolean checkForSubroutine(){ ... }
   boolean checkForImpliesToOR(){ ... }
   String addBrackets(String,String,String,String,String){ ... }
}
public Proof(Vector<line> H,String G,Vector<Vector<line>> VL,
             int start_of_subroutine,int start_of_impliesToOr){
   // Αρχικοποίηση μεταβλητών της απόδειξης
   next_level=0;
   this.start_of_subroutine=start_of_subroutine;
   this.start_of_impliesToOr=start_of_impliesToOr;
   //Οι αρχικές υποθέσεις τοποθετούνται στην απόδειξη
   for (int i=0; i<H.size(); i++){
       line L=(line)H.get(i);
       P.add(L);
   }
   // Τα επίπεδα προστίθενται στο L αν υπάρχουν
   if (VL.size()>0){
      for (int i=0; i<VL.size(); i++){
          Vector<line> V=new Vector<line>();
          for (int j=0; j<VL.get(i).size(); j++) V.add((line)VL.get(i).get(j));
          L.add(V);
```

```
      }
   }
   // Αν δόθηκε κενός στόχος τότε δεν προχωράει η απόδειξη
   if (G.compareTo("")==0) return;
   // Αν στόχος είναι το true τότε επιτυχία
   if (G.compareTo("\u22a4")==0){
      P.add(new line(P.size(),"\u22a4","\u22a4intro"));
      optimization();
      return;
   }
   // Τοποθέτηση του στόχου G στην στοίβα στόχων
   S.push(new line(G));
   // Έλεγχος αν κάποια από τις αρχικές υποθέσεις είναι ο στόχος
   if ( checkAssumptionsForGoal(true)!=-1 ){
      optimization();
      return;
   }
   BR();  // Κάνε Backward Reasoning
   /* Αν η απόδειξη είναι κενή τότε δεν μπορεί να προχωρήσει το FR και η απόδειξη
      κρίνεται αποτυχημένη. Αντίστοιχα, αν η στοίβα είναι κενή η απόδειξη δεν
      μπορεί να συνεχίσει (κάνε βελτιστοποίηση). */
   if (P.size()==0 || S.size()==0){
      optimization();
      return;
   }
   FR();  // Κάνε Forward Reasoning
   /* Αν υπάρχει ακόμα κάποιος στόχος τότε η απόδειξη δεν
      ολοκληρώθηκε επιτυχώς. Άρα, καθαρισμός της απόδειξης */
   if (S.size()>0) P.clear();
   // Διαφορετικά κάνε βελτιστοποίηση της απόδειξης
   else optimization();
}
public void optimization(){
   // Αν δεν υπάρχει απόδειξει τότε δεν υπάρχει λόγος να τρέξει
   if (P.size()==0) return;
   // Η τελευταία γραμμή της απόδειξης θα χρησιμοποιηθεί
   P.get(P.size()-1).setUsed();
   // Διάτρεξη της απόδειξης από το τέλος προς την αρχή
   for (int i=P.size()-1; i>-1; i--){
       // Οι αρχικές υποθέσεις χρησιμοποιούνται στην απόδειξη
       if (P.get(i).getComment().compareTo("Initial Assumption")==0){
          P.get(i).setUsed();
       }
       /* Αν η γραμμή χρησιμοποιείται τότε οι γραμμές από τις έχει βγει
          πρέπει να  χρησιμοποιηθούν */
       else if ( (boolean)P.get(i).getUsed() ){
          // Αν έχει χρησιμοποιήσει μια γραμμή τότε
          if ((int)P.get(i).getPLine0()!=-1){
             // Θέσε ως ενεργή την γραμμή αυτή
             P.get(P.get(i).getPLine0()).setUsed();
```

```
            }
            // Αν έχει χρησιμοποιήσει και δεύτερη γραμμή τότε
            if ((int)P.get(i).getPLine1()!=-1){
               // Θέσε ως ενεργή την γραμμή αυτή
               P.get(P.get(i).getPLine1()).setUsed();
            }
            // Αν έχει χρησιμοποιήσει και τρίτη γραμμή τότε
            if ((int)P.get(i).getPLine2()!=-1){
               // Θέσε ως ενεργή την γραμμή αυτή
               P.get(P.get(i).getPLine2()).setUsed();
            }
            // Αν έχει χρησιμοποιήσει και τέταρτη γραμμή τότε
            if ((int)P.get(i).getPLine3()!=-1){
               // Θέσε ως ενεργή την γραμμή αυτή
               P.get(P.get(i).getPLine3()).setUsed();
            }
            // Αν έχει χρησιμοποιήσει και πέμπτη γραμμή τότε
            if ((int)P.get(i).getPLine4()!=-1){
               // Θέσε ως ενεργή την γραμμή αυτή
               P.get(P.get(i).getPLine4()).setUsed();
            }
         }
      }
   }
Vector<String> AnalyzeFormula(String formula){
   // Διάνυσμα όπου θα αποθηκευτούν οι υποτύποι και το top-level συνδετικό
   Vector<String> A=new Vector<String>(3);
   // Η φόρμουλα θα αναλυθεί σε tokens με βάση τα συνδετικά και τις παρενθέσεις
   StringTokenizer t=new StringTokenizer(formula,"\u2227\u2228\u2192\u00ac()",true);
   // Το top-level συνδετικό
   String c=new String();
   // Ο παρακάτω μετρητής λειτουργεί σαν την στοίβα ενός αυτομάτου (PDA)
   int counter=0;
   /* Πρέπει να βρούμε την θέση που έχει το συνδετικό στην φόρμουλα για να
       μπορέσουμε να την διαχωρίσουμε στα βασικά της μέρη */
   int position=-1;
   // Επανάληψη όσο υπάρχουν tokens
   while (t.hasMoreTokens()){
      // Πάρε το επόμενο token
      String s=t.nextToken();
      // Η θέση στην θέση το token στην φόρμουλα
      position=position+s.length();
      // Αν το token έχει μέσα αριστερή παρένθεση τότε αύξησε τον μετρητή
      if (s.compareTo("(")==0) ++counter;
      // Αλλιώς αν το token έχει μέσα δεξιά παρένθεση τότε μείωσε τον μετρητή
      else if (s.compareTo(")")==0) --counter;
      // Αλλιώς αν το token είναι συνδετικό και ο μετρητής είναι μηδέν βρέθηκε
      else if ( (s.compareTo("\u2227")==0 || s.compareTo("\u2228")==0
                || s.compareTo("\u2192")==0 || s.compareTo("\u00ac")==0)
                && (counter==0) )
      {
```

```
            c=s;    // Αποθήκευση του συνδετικού
            break;  // Δεν χρειάζεται να συνεχίσουμε την ανάλυση
         }
      }
      // Αν ο τύπος δεν μπόρεσε να αναλύθεί καθόλου τότε είναι προτασιακό γράμμα
      if (position==-1){
         A.add(formula);  // Το προτασιακό γράμμα είναι στην πρώτη θέση
         A.add("");       // Δεν υπάρχει συνδετικό
         A.add("");       // Δεν υπάρχει δεύτερο μέρος
      }
      // Αν ο τύπος είναι η άρνηση ενός υποτύπου τότε έχουμε μοναδιαίο συνδετικό
      else if (position==0){
         A.add("");    // Δεν υπάρχει πρώτο μέρος
         A.add(c);     // Αποθήκευση του συνδετικού
         // Ο υποτύπος είναι είναι υποσυμβολοσειρά της φόρμουλας από την position+1
         String subtype=formula.substring(position+1);
         // Αν στην πρώτη θέση του υπάρχει δεξιά παρένθεση τότε πρέπει να φύγει
         String subformula;
         if (subtype.startsWith("(")){
            subformula=subtype.substring(1,subtype.length()-1);
         }
         else subformula=subtype;
         A.add(subformula);  // Αποθήκευση της υποφόρμουλας
      }
      // Διαφορετικά έχουμε δυαδικό συνδετικό και περιπτώσεις or,and και implies
      else{
         // Αφαίρεση παρανθέσεων από την πρώτη υποφόρμουλα αν χρειάζεται
         String subtype1=formula.substring(0,position);
         String subformula1;
         if (subtype1.startsWith("(")){
            subformula1=subtype1.substring(1,subtype1.length()-1);
         }
         else subformula1=subtype1;
         A.add(subformula1);  // Αποθήκευση της πρώτης υποφόρμουλας
         A.add(c);  // Αποθήκευση του συνδετικού
         // Αφαίρεση παρενθέσεων από την δεύτερη υποφόρμουλα αν χρειάζεται
         String subtype2=formula.substring(position+1);
         String subformula2;
         if (subtype2.startsWith("(")){
            subformula2=subtype2.substring(1,subtype2.length()-1);
         }
         else subformula2=subtype2;
         A.add(subformula2);  // Αποθήκευση της δεύτερης υποφόρμουλας
      }
      return A; // Επιστροφή της ανάλυσης μέσω του διανύσματος A
   }
   boolean isGoal(line L,boolean doCopy){
      // Πάρε τον τελευταίο στόχο από την στοίβα.
      line G=(line)S.pop();
```

```
   // Αν η φόρμουλα της γραμμής L είναι η φόρμουλα του στόχου G τότε
   if (L.getFormula().compareTo(G.getFormula())==0){
      // Αν στην στοίβα υπάρχει και άλλος στόχος τότε
      if (S.size()>0){
         // Πάρε τον επόμενο στόχο από την στοίβα
         line hyperGoal=(line)S.pop();
         // Θέσε κάποια χρησιμοποιούμενη γραμμή για τον υπερστόχο
         // Αν ο στόχος αντιγραφεί τότε χρήση της επόμενη γραμμής
         if (doCopy) hyperGoal.setPLine(P.size());
         // Διαφορετικά χρήση της τρέχουσας γραμμής
         else hyperGoal.setPLine(L.getNumber());
         // Εισήγαγε και πάλι τον υπερστόχο στην στοίβα
         S.push(hyperGoal);
      }
      return true;
   }
   S.push(G);
   return false;
}
int checkAssumptionsForGoal(boolean doCopy){
   // Διάτρεξη του διανύσματος απόδειξης
   for (int i=0; i<P.size(); i++){
       // Έλεγξε αν η τρέχουσα γραμμή της απόδειξης είναι στόχος τότε
       if ( isGoal((line)P.get(i),doCopy) ){
          // Αντιγραφή της γραμμής στο τέλος της απόδειξης
          if (doCopy) copy((line)P.get(i));
          // Η απόδειξη πρέπει να τελειώσει. Επιστροφή της γραμμής επιτυχίας.
          return i;
       }
   }
   // Η απόδειξη δεν πρέπει να τελειώσει
   return -1;
}
void copy(line L){
   P.add(new line(P.size(),L.getFormula(),"Copy",L.getNumber(),-1));
}
void finishProof(){
   // Αν η στοίβα είναι κενή τότε δεν υπάρχει στόχος
   if (S.size()==0) return;
   // Κάθε στόχος πρέπει να ενημερωθεί ως προς τα σχόλια του
   While ( S.size()>0 ){
      // Πάρε τον στόχο στην κορυφή της στοίβας
      line goal=(line)S.pop();
      // Η τελευταία γραμμή είναι η γραμμή όπου θα γραφτεί ο στόχος goal
      goal.setNumber(P.size());
      // Ο στόχος γράφεται στην απόδειξη
      P.add(goal);
      // Αν στην στοίβα υπάρχουν και άλλοι στόχοι τότε πρέπει να ενημερωθούν
      if (S.size()>0){
         // Πάρε τον επόμενο κορυφαίο στόχο από την στοίβα
         line hyperGoal=(line)S.pop();
```

```
            // Κάποια χρησιμοποιούμενη γραμμή είναι η γραμμή του goal
            hyperGoal.setPLine(goal.getNumber());
            // Ξανα-τοποθέτησε τον στόχο hyperGoal πίσω στην στοίβα
            S.push(hyperGoal);
         }
      }
   }
void BR(){
   // Συνεχόμενη επανάληψη μέχρι ο τελευταίος στόχος να μην μπορεί να αναλυθεί
   while (true){
      /* Πάρε τον στόχο που είναι στην κορυφή της στοίβας ως γραμμή απόδειξης
          χωρίς να τον αφαιρέσεις από την στοίβα */
      line lastGoal=(line)S.peek();
      // Ανάλυση του στόχου για εύρεση του βασικού του συνδετικού
      Vector<String> A=AnalyzeFormula(lastGoal.getFormula());
      // Πάρε το συνδετικό από το διάνυσμα A
      String connector=(String)A.get(1);
      // Αν είναι κενό τότε ο στόχος δεν μπορεί να αναλυθεί περαιτέρω
      if (connector.compareTo("")==0){
         // Αν ο υποστόχος είναι το true τότε επιτύχια
         if (A.get(0).compareTo("\u22a4")==0){
            P.add(new line(P.size(),"\u22a4","\u22a4intro"));
            finishProof();
            return;
         }
         // Διαφορετικά το BR σταματά
         else return;
      }
      /*
         Ανάλογα με το βασικό συνδετικό του βασικού στόχου μπορούν να εφαρμοστούν
         οι κανόνες (implies introduction), (not introduction), (or introduction)
         και (and introduction).
         Ανάλογα με το ποιος κανόνας θα εφαρμοστεί θα τοποθετηθεί στην στοίβα
         ένας νέος στόχος εκτός από την περίπτωση του and introduction όπου η
         Απόδειξη χωρίζεται σε δύο υπο-αποδείξεις.
      */
      // Αν το συνδετικό είναι το implies τότε τρέχει η impliesIntro()
      if (connector.compareTo("\u2192")==0){
         S.push( impliesIntro((String)A.get(0),(String)A.get(2)) );
      }
      // Αν το συνδετικό είναι το not τότε τρέχει η notIntro()
      else if (connector.compareTo("\u00ac")==0){
         S.push( notIntro((String)A.get(2)) );
      }
      // Αν το συνδετικό είναι το or τότε τρέχει η orIntro()
      else if (connector.compareTo("\u2228")==0){
         S.push( orIntro((String)A.get(0),(String)A.get(2)) );
      }
      // Αν το συνδετικό είναι το and τότε τρέχει η andIntro() και τερματίζει
      else if (connector.compareTo("\u2227")==0){
         andIntro((String)A.get(0),(String)A.get(2));
```

```
            break;
         }
         /* Εφόσον έχει τεθεί κάποιος νέος στόχος πρέπει να γίνει έλεγχος αν μπορεί
            να επιτευχθεί άμεσα με βάση τις υποθέσεις που έχουμε μέχρι τώρα. */
         if (checkAssumptionsForGoal(true)!=-1 ){
            finishProof();
            return;
         }
      }
   }
   line impliesIntro(String H,String G){
      // Υποθέτουμε το H με μια νέα γραμμή στην απόδειξη
      P.add(new line(P.size(),H,"Assumption"));
      /* Ενημέρωση του υπερστόχου ότι χρησιμοποιεί την νέα υπόθεση και ότι θα βγει
         με τον κανόνα implies introduction */
      updateHyperGoal("\u2192intro");
      // Επιστροφή του νέου στόχου ως γραμμή απόδειξης
      return new line(G);
   }
   line notIntro(String G){
      // Υποθέτουμε το G με μια νέα γραμμή στην απόδειξη
      P.add(new line(P.size(),G,"Assumption"));
      /* Ενημέρωση του υπερστόχου ότι χρησιμοποιεί την νέα υπόθεση και ότι θα βγει
         με τον κανόνα not introduction */
      updateHyperGoal("\u00acintro");
      // Επιστροφή του bottom ως γραμμή απόδειξης
      return new line("\u22a5");
   }
   line orIntro(String H,String G){
      // Υποθέτουμε το not H με μια νέα γραμμή στην απόδειξη
      P.add(new line(P.size(),"\u00ac"+H,"Assumption"));
      /* Ενημέρωση του υπερστόχου ότι χρησιμοποιεί την νέα υπόθεση και ότι θα βγει
         με τον κανόνα or introduction */
      updateHyperGoal("\u2228intro");
      // Επιστροφή του νέου στόχου ως γραμμή απόδειξης
      return new line(G);
   }
   void andIntro(String G1,String G2){
      int first_goal_number,second_goal_number;
      /* Τοποθέτηση του G1 στην στοίβα στόχων και έλεγχος αν αποδεικνύεται από
         τις υποθέσεις */
      S.push(new line(G1));
      // Αν μπορεί να αποδειχτεί από τις υποθέσεις τότε
      if ( (first_goal_number=checkAssumptionsForGoal(false))!=-1 ){
         G1="";  // εκμηδένισε τον στόχο
      }
      // Διαφορετικά
      else S.pop();  // Αφαίρεσε τον G1 από την στοίβα
      /* Τοποθέτηση του G2 στην στοίβα και έλεγχος αν αποδεικνύεται από
         τις υποθέσεις */
```

```
S.push(new line(G2));
// Αν μπορεί να αποδειχτεί από τις υποθέσεις τότε
if ( (second_goal_number=checkAssumptionsForGoal(false))!=-1 ){
   G2="";  // Εκμηδένισε τον στόχο
}
// Διαφορετικά
else S.pop();  // Αφαίρεσε τον G2 από την στοίβα
// Η απόδειξη χωρίζεται σε δύο υπο-αποδείξεις.
// Η πρώτη απόδειξη έχει όλες τις μέχρι τώρα υποθέσεις και στόχο τον G1
Proof subProof1=new Proof(P,G1,L,start_of_subroutine,start_of_impliesToOr);
// Αν αποτύχει η πρώτη απόδειξη τότε αποτυγχάνει ολόκληρη η απόδειξη
if (subProof1.P.size()==0){
   P.clear();
   return;
}
// Η δεύτερη απόδειξη έχει όλες τις μέχρι τώρα υποθέσεις και στόχο τον G2
Proof subProof2=new Proof(P,G2,L,start_of_subroutine,start_of_impliesToOr);
// Αν αποτύχει η δεύτερη απόδειξη τότε αποτυγχάνει ολόκληρη η απόδειξη
if (subProof2.P.size()==0){
   P.clear();
   return;
}
// Κράτα το μέγεθος της απόδειξης μέχρι αυτό το σημείο
int initial_size=P.size();
// Αν το μέγεθος της πρώτης απόδειξης είναι μεγαλύτερο από το αρχικό τότε
if (subProof1.P.size()>initial_size){
   // Προσθήκη των έξτρα γραμμών στην απόδειξη
   for (int i=initial_size; i<subProof1.P.size(); i++){
       P.add((line)subProof1.P.get(i));
   }
   // Κράτα την γραμμή όπου γράφηκε ο πρώτος υποστόχος
   first_goal_number=P.size()-1;
}
// Αν το μέγεθος της δεύτερης απόδειξης είναι μεγαλύτερο από το αρχικό τότε
if (subProof2.P.size()>initial_size){
   // Μετρητής για ενημέρωση αριθμών γραμμών της δεύτερη απόδειξης
   int counter=0;
   // Προσθήκη των έξτρα γραμμών στην απόδειξη
   for (int i=initial_size; i<subProof2.P.size(); i++){
       /* Αριθμός γραμμής είναι το μέγεθος της πρώτης απόδειξης
          συν τον μετρητή */
       subProof2.P.get(i).setNumber(subProof1.P.size()+counter) ;
       ++counter;
       P.add((line)subProof2.P.get(i));
   }
   // κράτα την γραμμή όπου γράφηκε ο δεύτερος υποστόχος
   second_goal_number=P.size()-1;
}
// Ενημέρωση του υπερστόχου
updateHyperGoalAndIntro(first_goal_number,second_goal_number);
```

```
   // Η απόδειξη τερματίζει
   finishProof();
}
void updateHyperGoal(String comment){
   // Πάρε τον κορυφαίο στόχο από την στοίβα με αφαίρεση από την στοίβα
   line Goal=(line)S.pop();
   // Ορισμός του σχολίου
   Goal.setComment(comment);
   // Ορισμός κάποια χρησιμοποιούμενη γραμμής (η τελευταία υπόθεση)
   Goal.setPLine(P.size()-1);
   // Εισαγωγή του στόχου και πάλι πίσω στην στοίβα
   S.push(Goal);
}
void updateHyperGoalAndIntro(int a,int b){
    // Πάρε τον κορυφαίο στόχο από την στοίβα
    line G=(line)S.pop();
    // Ο στόχος βγαίνει με and introduction
    G.setComment("\u2227intro");
    // Η πρώτη γραμμή είναι η γραμμή όπου επιτεύχθει ο πρώτος υποστόχος
    G.setPLine0(a);
    // Η δεύτερη γραμμή είναι η γραμμή όπου επιτεύχθει ο δεύτερο υποστόχος
    G.setPLine1(b);
    // Ξανα-τοποθέτησε τον στόχο στην στοίβα
    S.push(G);
}
void FR(){
   /*
      Μεταβλητή στην οποία κρατάμε τον αριθμοδείκτη του τελευταίου επιπέδου και
      με βάση την οποία μπορέσουμε να ελέγξουμε αν παράχθηκε ή όχι κάποιο νέο
      επίπεδο κατά την φάση των συνδυασμών
   */
   int last_level;
   // Δημιουργία επιπέδων με βάση την μέχρι τώρα απόδειξη αν δεν υπάρχουν ήδη
   if (L.size()==0){
      initializeLevels();     // Αρχικοποιήση επιπέσων υποθέσεων
      last_level=L.size()-1;  // Κράτα το τελευταίο επίπεδο
      // Αρχικός συνδυασμός επιπέδων για κανόνες (implies,not,and,or)-elim.
      // Αν η μέθοδος firstCombination επιστρέψει true τότε ο στόχος επιτεύχθει
      if ( firstCombination() ) return;
      // Διαφορετικά έλεγξε μήπως δεν παράχθηκε κάποιο επίπεδο
      else if (L.size()-1==last_level){
         // Δες αν μπορείς να κάνεις PBC
         if ( PBC() ){
            // Έλεγξε αν το νέος στόχος ικανοποιείται από τις υποθέσεις
            if (checkAssumptionsForGoal(true)!=-1 ){
               // Τερματισμός της απόδειξης
               finishProof();
               return;
            }
         }
         // Διαφορετικά προσπάθησε να εκτελέσεις κάποια υπορουτίνα
```

```
      else if ( checkForSubroutine() ){
         // Έλεγξε αν η γραμμή που βγήκε είναι στόχος
         if ( isGoal(P.get(P.size()-1),false) ){
            // Η απόδειξη πρέπει να τερματιστεί
            finishProof();
            return;
         }
      }
      // Διαφορετικά προσπάθησε να κάνεις κάποιο implies σε or
      else if ( checkForImpliesToOR() ){
         // Έλεγξε αν η γραμμή που βγήκε είναι στόχος
         if ( isGoal(P.get(P.size()-1),false) ){
            // Η απόδειξη πρέπει να τερματιστεί
            finishProof();
            return;
         }
      }
     // Διαφορετικά, η απόδειξη απέτυχε
     else{
         // Καθαρισμός της απόδειξης
         P.clear();
         return;
      }
   }
}
/*
   Πλέον έχουν δημιουργηθεί τα επίπεδα και πρέπει να γίνει επαναληπτικός
   συνδυασμός τους για την παραγωγή νέων φορμουλών μέχρι να επιτευχθεί ο
   στόχος ή να αποτύχει η απόδειξη
*/
last_level=L.size()-1;  // Κράτα το τελευταίο επίπεδο
next_level=L.size();    // Κράτα το επόμενο επίπεδο
while (true){
    // Αν η μέθοδος nextCombination επιστρέφει true τότε ο στόχος επιτεύχθει
    if ( nextCombination() ) return;
    // Διαφορετικά έλεγξε μήπως δεν παράχθηκε κάποιο επίπεδο
    else if (L.size()-1==last_level){
       // Δες αν μπορείς να κάνεις PBC
       if ( PBC() ){
          // Έλεγξε αν το νέος στόχος ικανοποιείται από τις υποθέσεις
          if (checkAssumptionsForGoal(true)!=-1 ){
             // Τερματισμός της απόδειξης
             finishProof();
             return;
          }
       }
       // Διαφορετικά προσπάθησε να εκτελέσεις κάποια υπορουτίνα
       else if ( checkForSubroutine() ){
           // Έλεγξε αν η γραμμή που βγήκε είναι στόχος
           if ( isGoal(P.get(P.size()-1),false) ){
              // Η απόδειξη πρέπει να τερματιστεί
```

```
                finishProof();
                return;
            }
        }
        // Διαφορετικά προσπάθησε να κάνεις κάποιο implies σε or
        else if ( checkForImpliesToOR() ){
            // Έλεγξε αν η γραμμή που βγήκε είναι στόχος
            if ( isGoal(P.get(P.size()-1),false) ){
                // Η απόδειξη πρέπει να τερματιστεί
                finishProof();
                return;
            }
        }
        // Διαφορετικά, η απόδειξη απέτυχε
        else{
            P.clear();
            return;
        }
    }
    // Διαφορετικά παράχθηκε νέο επίπεδο
    else{
        last_level=L.size()-1;  // Ενημέρωση του τελευταίου επιπέδου
        next_level=L.size();    // Ενημέρωση του επόμενου επιπέδου
    }
  }
}
void initializeLevels(){
  // Δημιουργία δύο διανυσμάτων διότι αρχικά θα υπάρχουν το πολύ 2 επίπεδα
  Vector<line> level1=new Vector<line>();
  Vector<line> level2=new Vector<line>();
  // Διάτρεξη της μέχρι τώρα απόδειξης
  for (int i=0; i<P.size(); i++){
      // Αν η γραμμή είναι αρχική υπόθεση τότε προστίθεται στον πρώτο επίπεδο
      if (P.get(i).getComment().compareTo("Initial Assumption")==0){
        level1.add((line)P.get(i));
      }
      // Διαφορετικά είναι υπόθεση που έγινε κατά την BR-ανάλυση
      else{
        level2.add((line)P.get(i));
      }
  }
  // Εισαγωγή στο διάνυσμα επιπέδων μόνο των επιπέδων που έχουν στοιχεία
  if (level1.size()>0) L.add(level1);
  if (level2.size()>0) L.add(level2);
  // Επόμενο επίπεδο είναι το 1ο ή το 2ο ανάλογα με το πόσα επίπεδα υπάρχουν ήδη
  next_level=L.size();
}
boolean firstCombination(){
  // Δήλωση του νέου επιπέδου
  Vector<line> newLevel=new Vector<line>();
```

```
    // Αν υπάρχει μόνο ένα επίπεδο στην απόδειξη σε αυτό το σημείο τότε
    if (L.size()==1){
      // Συνδυασμός του μηδενικού επιπέδου μόνο με τον εαυτό του
      newLevel=combineLevels((Vector<line>)L.get(0),
                             (Vector<line>)L.get(0),true);
    }
    // Διαφορετικά αν υπάρχουν δύο επίπεδα στην απόδειξη σε αυτό το σημείο τότε
    else{
      // Συνδυασμός του μηδενικού επιπέδου με τον εαυτό του
      Vector<line> newLevel1;
      newLevel1=combineLevels((Vector<line>)L.get(0),
                              (Vector<line>)L.get(0),true);
      // Συνδυασμός του μηδενικού επιπέδου με το πρώτο επίπεδο
      Vector<line> newLevel2;
      newLevel2=combineLevels((Vector<line>)L.get(0),
                              (Vector<line>)L.get(1),false);
      // Συνδυασμός του πρώτου επιπέδου με τον εαυτό του
      Vector<line> newLevel3;
      newLevel3=combineLevels((Vector<line>)L.get(1),
                              (Vector<line>)L.get(1),true);
      // Πρόσθεση των διανυσμάτων newLevel1,newLevel2,newLevel2 στο newLevel
      for (int i=0; i<newLevel1.size(); i++){
          newLevel.add( (line)newLevel1.get(i) );
      }
      for (int i=0; i<newLevel2.size(); i++){
          newLevel.add( (line)newLevel2.get(i) );
      }
      for (int i=0; i<newLevel3.size(); i++){
          newLevel.add( (line)newLevel3.get(i) );
      }
    }
    // Αν ο στόχος επιτεύχθει τότε επιστροφή επιτυχίας
    if (S.size()==0 || P.size()==0) return true;
    // Αν γράφηκε κάποια νέα γραμμή στην απόδειξη
    if (newLevel.size()>0){
       // Εισαγωγή του νέου επιπέδου στο διάνυσμα επιπέδων
       L.add(newLevel);
       // Αύξηση του επόμενου επιπέδου
       next_level=L.size();
    }
    return false;  // Δεν επιτεύχθει ο στόχος
}
boolean nextCombination(){
    // Δήλωση ενός νέου επιπέδου
    Vector<line> newLevel=new Vector<line>();
    // Κράτα την θέση του τελευταίου επίπεδου στο διάνυσμα επιπέδων
    int last=next_level-1;
    // Συνδυασμός τελευταίου επιπέδου με όλα τα προηγούμενα
    for (int i=0; i<next_level-1; i++){
        Vector<line> level;
```

```
        level=combineLevels((Vector<line>)L.get(i),
                            (Vector<line>)L.get(last),false);
        // Αν το νέο επίπεδο είναι κενό και ο στόχος επιτεύχθει τότε επιτυχία
        if (S.size()==0 || P.size()==0) return true;
        // Διαφορετικά πρόσθεσε τις νέες γραμμές (αν υπάρχουν) στο νέο επίπεδο
        for (int j=0; j<level.size(); j++){
            newLevel.add( (line)level.get(j) );
        }
    }
    // Συνδυασμός τελευταίου επιπέδου με τον εαυτό του
    Vector<line> level;
    level=combineLevels((Vector<line>)L.get(last),
                        (Vector<line>)L.get(last),true);
    // Αν επιτεύχθει ο στοχος τότε επιστροφή true
    if (S.size()==0 || P.size()==0) return true;
    // Διαφορετικά, προσθήκη των νέων γραμμών (αν υπάρχουν) στο νέο επίπεδο
    for (int i=0; i<level.size(); i++){
        newLevel.add( (line)level.get(i) );
    }
    // Αν το νέο επίπεδο δεν είναι κενό τότε
    if (newLevel.size()>0){
       // Εισαγωγή του νέου επιπέδου στο διάνυσμα επιπέδων
       L.add(newLevel);
       // Αύξηση του επόμενου επιπέδου
       next_level=L.size();
    }
    return false;  // Δεν επιτεύχθει ο στόχος
}
Vector<line> combineLevels(Vector<line> level1,Vector<line> level2,
                           boolean isEqual)
{
    // Δημιουργία ενός νέου διανύσματος όπου θα αποθηκευτούν τα αποτελέσματα
    Vector<line> newLevelPart=new Vector<line>();
    // Αν η στοίβα είναι κενή επιστρέφουμε κενό διάνυσμα
    if (S.size()==0) return newLevelPart;
    // Από που θα ξεκινήσει ο αριθμοδείκτης του δεύτερου for-loop
    int jStart=0;
    /*
       Κάθε γραμμή του 1ου επιπέδου θα συνδυαστεί με κάθε γράμμη του 2ου επιπέδου.
       Οπότε έχουμε ένα διπλό for-loop όπου το i αντιστοιχεί σε μια γραμμή του 1ου
       επιπέδου και το j σε μια γραμμή του 2ου επιπέδου
    */
    for (int i=0; i<level1.size(); i++){
        /* Αν έχουμε το ίδιο επίπεδο τότε κάθε γραμμή i συνδυάζεται με τον
           εαυτό της και όλες τις επόμενες */
        if (isEqual){
           jStart=i; // Η αρχή είναι η ίδια η γραμμή
        }
        for (int j=jStart; j<level2.size(); j++){
            /* Αν i==j και έχουμε μόνο ένα επίπεδο τότε μπορεί να εκτελεστεί μόνο
```

```
   κάποιος κανόνας που χρησιμοποιεί μόνο μια γραμμή */
if (i==j && isEqual){
   // Έλεγχος για τον κανόνα not not elimination
   if ( notnotElim((line)level1.get(i)) ){
      // Αν η τελευταία γραμμή είναι στόχος τότε
      if ( isGoal(P.get(P.size()-1),false) ){
         // Η απόδειξη πρέπει να τερματιστεί
         finishProof();
         // Επιστροφή κενού διανύσματος
         return new Vector<line>();
      }
      // Αλλιώς έλεγχος για bottom elimination στην νέα γραμμή
      else if ( bottomElim((line)P.get(P.size()-1)) ){
         // Η τελευταία γραμμή είναι σίγουρα στόχος
         isGoal(P.get(P.size()-1),false);
         // Η απόδειξη πρέπει να τερματιστεί
         finishProof();
         // Επιστροφή κενού διανύσματος
         return new Vector<line>();
      }
      // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
      else newLevelPart.add(P.get(P.size()-1));
   }
   // Έλεγχος για τον κανόνα and elimination
   else if ( andElim((line)level1.get(i)) ){
      // Πρέπει να γίνουν δύο έλεγχοι για στόχο (δύο νέες γραμμές)
      // Αν η προτελευταία γραμμή είναι στόχος τότε
      if ( isGoal(P.get(P.size()-2),false) ){
         // Διαγραφή της επόμενης γραμμής από την απόδειξη
         P.remove(P.size()-1);
         // Η απόδειξη πρέπει να τερματιστεί
         finishProof();
         // Επιστροφή κενού διανύσματος
         return new Vector<line>();
      }
      // Αλλιώς έλεγχος για bottom elimination στην νέα γραμμή
      else if ( bottomElim((line)P.get(P.size()-2)) ){
         // Αφαίρεση της δεύτερη γραμμής που έβγαλε το and elim
         P.remove(P.size()-2);
         // Η τελευταία γραμμή είναι σίγουρα στόχος
         isGoal(P.get(P.size()-1),false);
         // Η απόδειξη πρέπει να τερματιστεί
         finishProof();
         // Επιστροφή κενού διανύσματος
         return new Vector<line>();
      }
      // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
      else newLevelPart.add(P.get(P.size()-2));
      // Αν η τελευταία γραμμή είναι στόχος τότε
      if ( isGoal(P.get(P.size()-1),false) ){
         // Η απόδειξη πρέπει να τερματιστεί
         finishProof();
```

```
            // Επιστροφή κενού διανύσματος
            return new Vector<line>();
         }
         // Αλλιώς έλεγχος για bottom elimination στην νέα γραμμή
         else if ( bottomElim((line)P.get(P.size()-1)) ){
            // Η τελευταία γραμμή είναι σίγουρα στόχος
            isGoal(P.get(P.size()-1),false);
            // Η απόδειξη πρέπει να τερματιστεί
            finishProof();
            // Επιστροφή κενού διανύσματος
            return new Vector<line>();
         }
         // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
         else newLevelPart.add(P.get(P.size()-1));
      }
      // Τέλος, έλεγχος αν μπορεί να εκτελεστεί ο κανόνας or elimination
      else{
         // Ανάλυση της γραμμής
         String formula=(String)level1.get(i).getFormula();
         Vector<String> A=AnalyzeFormula(formula);
         // Αν είναι or και δεν έχει ξαναγίνει μέσα στις αναδρομές τότε
         if (A.get(1).compareTo("\u2228")==0 && !level1.get(i).getOr()){
            // Το orElim δεν πρέπει να ξαναγίνει μέσα στις αναδρομές
            level1.get(i).setOr();
            // Κράτα τον αριθμός της γραμμής
            int number=level1.get(i).getNumber();
            // Αν εκτελεστεί με επιτυχία το or elim τότε
            if ( orElim((String)A.get(0),(String)A.get(2),number) ){
               // Η απόδειξη πρέπει να τερματίσει
               finishProof();
               // Επιστροφή κενού διανύσματος
               return new Vector<line>();
            }
         }
         // Διαφορετικά
         else{
            // Αν η απόδειξη είναι κενή τότε η απόδειξη απέτυχε
            if (P.size()==0) return new Vector<line>();
         }
      }
   }
   /* Αλλιώς, μπορούν να εκτελεστούν κανόνες οι οποίοι χρησιμοποιούν
      δύο γραμμές. */
   else{
      // Έλεγχος για τον κανόνα implies Elimination
      if ( impliesElim((line)level1.get(i),(line)level2.get(j)) ){
         // Αν η τελευταία γραμμή είναι στόχος τότε
         if ( isGoal(P.get(P.size()-1),false) ){
            // Η απόδειξη πρέπει να τερματιστεί
            finishProof();
            // Επιστροφή κενού διανύσματος
            return new Vector<line>();
```

```
                }
                // Αλλιώς έλεγχος για bottom elimination στην νέα γραμμή
                else if ( bottomElim((line)P.get(P.size()-1)) ){
                    // Η τελευταία γραμμή είναι σίγουρα στόχος
                    isGoal(P.get(P.size()-1),false);
                    // Η απόδειξη πρέπει να τερματιστεί
                    finishProof();
                    // Επιστροφή κενού διανύσματος
                    return new Vector<line>();
                }
                // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
                else newLevelPart.add(P.get(P.size()-1));
            }
            // Έλεγχος για τον κανόνα not Elimination
            else if ( notElim((line)level1.get(i),(line)level2.get(j)) ){
                // Αν η τελευταία γραμμή είναι στόχος τότε
                if ( isGoal(P.get(P.size()-1),false) ){
                    // Η απόδειξη πρέπει να τερματιστεί
                    finishProof();
                    // Επιστροφή κενού διανύσματος
                    return new Vector<line>();
                }
                // Αλλιώς έλεγχος για bottom elimination στην νέα γραμμή
                else if ( bottomElim((line)P.get(P.size()-1)) ){
                    // Η τελευταία γραμμή είναι σίγουρα στόχος
                    isGoal(P.get(P.size()-1),false);
                    // Η απόδειξη πρέπει να τερματιστεί
                    finishProof();
                    // Επιστροφή κενού διανύσματος
                    return new Vector<line>();
                }
                // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
                else newLevelPart.add(P.get(P.size()-1));
            }
            // Έλεγχος για τον κανόνα Modus Tolens
            else if ( modusTolens((line)level1.get(i),(line)level2.get(j)) ){
                // Αν η τελευταία γραμμή είναι στόχος τότε
                if ( isGoal(P.get(P.size()-1),false) ){
                    // Η απόδειξη πρέπει να τερματιστεί
                    finishProof();
                    // Επιστροφή κενού διανύσματος
                    return new Vector<line>();
                }
                // Διαφορετικά βάλτην στο επόμενο επίπεδο και συνέχισε
                else newLevelPart.add(P.get(P.size()-1));
            }
        }
    }
}
// Επιστροφή των νέων γραμμών
return newLevelPart;
}
```

```
boolean impliesElim(line L1,line L2){
   // Ανάλυση της φόρμουλας που υπάρχει στην πρώτη γραμμή
   Vector<String> A1=AnalyzeFormula(L1.getFormula());
   /* Αν το βασικό συνδετικό είναι το implies με πρώτο μέρος την φόρμουλα της L2
      τότε εφαρμόζεται ο κανόνας και στην απόδειξη γράφεται το δεύτερο μέρος */
   if (A1.get(1).equals("\u2192") && A1.get(0).equals(L2.getFormula())){
      String newF1=(String)A1.get(2);
      P.add(new line(P.size(),newF1,"\u2192elim",L1.getNumber(),L2.getNumber()));
      return true;  // Επιτυχία
   }
   // Ανάλυση της φόρμουλας που υπάρχει στην δεύτερη γραμμή
   Vector<String> A2=AnalyzeFormula(L2.getFormula());
   /* Αν το βασικό συνδετικό είναι το implies με πρώτο μέρος την φόρμουλα της L1
      τότε εφαρμόζεται ο κανόνας και στην απόδειξη γράφεται το δεύτερος μέρος */
   if (A2.get(1).equals("\u2192") && A2.get(0).equals(L1.getFormula())){
      String newF2=(String)A2.get(2);
      P.add(new line(P.size(),newF2,"\u2192elim",L1.getNumber(),L2.getNumber()));
      return true;  // Επιτυχία
   }
   return false;  // Αποτυχία
}
boolean notElim(line L1,line L2){
   // Ανάλυση της φόρμουλας που υπάρχει στην πρώτη γραμμή
   Vector<String> A1=AnalyzeFormula(L1.getFormula());
   /*
      Αν το βασικό συνδετικό είναι το not με δεύτερο μέρος την φόρμουλα της L2
      τότε μπορεί να εφαρμοστεί ο κανόνας και στην απόδειξη γράφεται το bottom
   */
   if (A1.get(1).equals("\u00ac") && A1.get(2).equals(L2.getFormula())){
       P.add(new line(P.size(),"\u22a5","\u00acelim",L1.getNumber(),L2.getNumber()));
      return true;  // Επιτυχία
   }
   // Ανάλυση της φόρμουλας που υπάρχει στην δεύτερη γραμμή
   Vector<String> A2=AnalyzeFormula(L2.getFormula());
   /*
      Αν το βασικό συνδετικό είναι το not με δεύτερο μέρος την φόρμουλα της L1
      τότε μπορεί να εφαρμοστεί ο κανόνας και στην απόδειξη γράφεται το bottom
   */
   if (A2.get(1).equals("\u00ac") && A2.get(2).equals(L1.getFormula())){
       P.add(new line(P.size(),"\u22a5","\u00acelim",L1.getNumber(),L2.getNumber()));
      return true;  // Επιτυχία
   }
   return false;  // Αποτυχία
}
boolean andElim(line L){
   // Ανάλυση της φόρμουλας
   Vector<String> A=AnalyzeFormula(L.getFormula());
   // Αν βασικό συνδετικό είναι το and τότε γράφονται και οι δύο φόρμουλες
   if (A.get(1).equals("\u2227")){
      P.add(new line(P.size(),(String)A.get(0),"\u2227elim",L.getNumber(),-1));
```

```
      P.add(new line(P.size(),(String)A.get(2),"\u2227elim",L.getNumber(),-1));
      return true;  // Επιτυχία
   }
   return false;  // Αποτυχία
}
boolean bottomElim(line L){
   // Αν η γραμμή έχει ως φόρμουλα το bottom τότε στην απόδειξη γράφεται ο στόχος
   if (L.getFormula().equals("\u22a5")){
      String goal=(String)S.peek().getFormula()
      P.add(new line(P.size(),goal,"\u22a5elim",L.getNumber(),-1));
      return true;  // Επιτυχία
   }
   return false;  // Αποτυχία
}
boolean notnotElim(line L){
   // Ανάλυση της φόρμουλας
   Vector<String> A=AnalyzeFormula(L.getFormula());
   // Αν βασικό συνδετικό της φόρμουλα είναι το not τότε
   if (A.get(1).compareTo("\u00ac")==0){
      // Ανάλυση της φόρμουλα στα δεξιά του συνδετικού
      Vector<String> B=AnalyzeFormula((String)A.get(2));
      // Αν βασικό συνδετικό είναι το not τότε γράφεται η υποφόρμουλα δεξιά
      if (B.get(1).compareTo("\u00ac")==0){
         String newF=(String)B.get(2);
         P.add(new line(P.size(),newF,"\u00ac\u00acelim",L.getNumber(),-1));
         return true;  // Επιτυχία
      }
   }
   return false;  // Αποτυχία
}
boolean modusTolens(line L1,line L2){
   // Ανάλυση των δύο γραμμών
   Vector<String> A=AnalyzeFormula(L1.getFormula());
   Vector<String> B=AnalyzeFormula(L2.getFormula());
   // Έλεγχος αν η πρώτη γραμμή είναι της μορφής φ implies ψ
   if (A.get(1).compareTo("\u2192")==0){
      // Έλεγχος αν η δεύτερη γραμμή είναι της μορφής not ψ
      if (B.get(1).equals("\u00ac") && B.get(2).equals((String)A.get(2))){
         // Μπορεί να εκτελεστεί ο κανόνας και έτσι, παράγεται το not φ
         // Η νέα φόρμουλα που θα προστεθεί στην απόδειξη
         String formula=new String();
         // Ανάλυση του φ
         Vector<String> F=AnalyzeFormula((String)A.get(0));
         // Αν δεν είναι ατομική φόρμουλα τότε θα έχει παρενθέσεις γύρω της
         if (!F.get(1).equals("")) formula="\u00ac"+"("+(String)A.get(0)+")";
         // Διαφορετικά αν η φόρμουλα είναι το bottom τότε έχουμε το true
         else if (A.get(0).compareTo("\u22a5")==0) formula="\u22a4";
         // Διαφορετικά η φόρμουλα δεν πρέπει να έχει παρενθέσεις γύρω της
         else formula="\u00ac"+(String)A.get(0);
```

```
            // Εισαγωγή της νέας φόρμουλας στην απόδειξη
            P.add(new line(P.size(),formula,"MT",L1.getNumber(),L2.getNumber()));
            return true;  // Επιτυχία
         }
      }
      // Έλεγχος αν η δεύτερη γραμμή είναι της μορφής φ implies ψ
      if (B.get(1).compareTo("\u2192")==0){
         // Έλεγχος αν η πρώτη γραμμή είναι της μορφής not ψ
         if (A.get(1).equals("\u00ac") && A.get(2).equals((String)B.get(2))){
            // Μπορεί να εκτελεστεί ο κανόνας και έτσι, παράγεται το not φ
            // Η νέα φόρμουλα που θα προστεθεί στην απόδειξη
            String formula=new String();
            // Ανάλυση του φ
            Vector<String> F=AnalyzeFormula((String)B.get(0));
            // Αν δεν είναι ατομική φόρμουλα τότε θα έχει παρενθέσεις γύρω της
            if (F.get(1).compareTo("")!=0){
               formula="\u00ac"+"("+(String)B.get(0)+")";
            }
            // Διαφορετικά αν η φόρμουλα είναι το bottom τότε έχουμε το true
            else if (B.get(0).equals("\u22a5")) formula="\u22a4";
            // Διαφορετικά η φόρμουλα δεν πρέπει να έχει παρενθέσεις γύρω της
            else formula="\u00ac"+(String)B.get(0);
            P.add(new line(P.size(),formula,"MT",L1.getNumber(),L2.getNumber()));
            return true;  // Επιτυχία
         }
      }
      return false;  // Αποτυχία
   }
   boolean PBC(){
      // Δες ποια φόρμουλα είναι στην κορυφή της στοίβας
      line l=(line)S.peek();
      // Αν στόχος είναι το bottom τότε δεν μπορεί να γίνει PBC
      if (l.getFormula().compareTo("\u22a5")==0) return false;
      // Διαφορετικά μπορεί να εκτελεστεί το PBC
      else{
         // Στην απόδειξη προστίθεται το not του στόχου ως υπόθεση
         P.add(new line(P.size(),"\u00ac"+l.getFormula(),"Assumption"));
         // Ενημέρωση του υπερστόχου
         updateHyperGoal("PBC");
         // Στην στοίβα προστίθεται το bottom ως ο νέος στόχος
         S.push(new line("\u22a5"));
         // Δημιουργία νέου επιπέδου
         Vector<line> newLevel=new Vector<line>();
         newLevel.add(new line(P.size(),"\u00ac"+l.getFormula(),"Assumption"));
         L.add(newLevel);
         return true;  // Επιτυχία
      }
   }
```

```
boolean orElim(String a,String b,int number){
   // Θα κρατήσουμε τα μέχρι τώρα επίπεδα στο διάνυσμα επιπέδων VL
   Vector<Vector<line>> VL=new Vector<Vector<line>>();
   // Αν υπάρχουν ήδη επίπεδα τότε πρόσθεση των επιπέδων του L στο VL
   if (L.size()>0){
      // Αντιγραφή κάθε επιπέδου του L
      for (int i=0; i<L.size(); i++){
          Vector<line> V=new Vector<line>();
          // Αντιγραφή κάθε στοιχείου του τρέχοντος επιπέδου στο βοήθημα V
          for (int j=0; j<L.get(i).size(); j++){
              V.add((line)L.get(i).get(j));
          }
          VL.add(V);
      }
   }
   // Προσωρινό επίπεδο για βοήθημα
   Vector<line> V=new Vector<line>();
   // Θα προσπαθήσουμε να αποδείξουμε με υπόθεση την πρώτη φόρμουλα τον στόχο
   // Πρόσθεσε στην απόδειξη ως υπόθεση την πρώτη φόρμουλα
   P.add(new line(P.size(),a,"Assumption"));
   // Η υπόθεση αυτή αποτελεί από μόνη της ένα νέο επίπεδο υποθέσεων
   V.add(P.get(P.size()-1));
   VL.add(V);
   // Κάνε απόδειξη του στόχου με υποθέσεις όλη την μέχρι τώρα απόδειξη
   Proof subProof1=new Proof(P,
                             (String)S.peek().getFormula(),
                             VL,
                             start_of_subroutine,
                             start_of_impliesToOr);
   // Αν η απόδειξη που πήραμε πίσω είναι κενή τότε η απόδειξη απέτυχε
   if (subProof1.P.size()==0){
      P.clear();      // Καθαρισμός της ολικής απόδειξης ως ένδειξη αποτυχίας
      return false;  // Αποτυχία – Μην συνεχίσεις την απόδειξη
   }
   // Αφαίρεση της τελευταίας γραμμής που ήταν υπόθεση για την πρώτη απόδειξη
   P.remove(P.size()-1);
   // Θα προσπαθήσουμε να αποδείξουμε με υπόθεση την δεύτερη φόρμουλα τον στόχο
   // Πρόσθεσε στην απόδειξη ως υπόθεση την δεύτερη φόρμουλα
   P.add(new line(P.size(),b,"Assumption"));
   // Το τελευταίο επίπεδο του VL πλέον πρέπει να περιέχει την δεύτερη φόρμουλα
   VL.get(VL.size()-1).remove(0);
   VL.get(VL.size()-1).add(P.get(P.size()-1));
   // Κάνε μια απόδειξη του στόχου με υποθέσεις όλη την μέχρι τώρα απόδειξη
   Proof subProof2=new Proof(P,
                             (String)S.peek().getFormula(),
                             VL,
                             start_of_subroutine,
                             start_of_impliesToOr);
   // Αν η απόδειξη που πήραμε πίσω είναι κενή τότε η απόδειξη απέτυχε
```

```
   if (subProof2.P.size()==0){
      P.clear();      // Καθαρισμός της ολικής απόδειξης ως ένδειξη αποτυχίας
      return false;  // Αποτυχία – Μην συνεχίσεις την απόδειξη
   }
   // Αφαίρεση της τελευταίας γραμμής που ήταν υπόθεση για την δεύτερη απόδειξη
   P.remove(P.size()-1);
   // Κράτα το αρχικό μέγεθος της απόδειξης
   int iSize=P.size();
   // Αντιγραφή της πρώτης απόδειξης στην βασική απόδειξη
   for (int i=iSize; i<subProof1.P.size(); i++) P.add((line)subProof1.P.get(i));
   // Αντιγραφή της δεύτερη απόδειξης στην βασική απόδειξη
   for (int i= iSize; i<subProof2.P.size(); i++){
       // Κάθε γραμμή αλλάζει νούμερο για να μπει στο τέλος της απόδειξης
       subProof2.P.get(i).setNumber(P.size());
       /*
          Αν για να βγει αυτή η γραμμή χρησιμοποίησε γραμμή πέρα από τις γραμμές
          της βασικής απόδειξης τότε πρέπει να γίνει shift κατά το μέγεθος της
          πρώτης απόδειξης
       */
       if (subProof2.P.get(i).getPLine0()>=iSize){
          subProof2.P.get(i).setPLine0
          (subProof2.P.get(i).getPLine0()+subProof1.P.size()-iSize);
       }
       if (subProof2.P.get(i).getPLine1()>=iSize){
          subProof2.P.get(i).setPLine1
          (subProof2.P.get(i).getPLine1()+subProof1.P.size()-iSize);
       }
       if (subProof2.P.get(i).getPLine2()>=iSize){
          subProof2.P.get(i).setPLine2
          (subProof2.P.get(i).getPLine2()+subProof1.P.size()-iSize);
       }
       if (subProof2.P.get(i).getPLine3()>=iSize){
          subProof2.P.get(i).setPLine3
          (subProof2.P.get(i).getPLine3()+subProof1.P.size()-iSize);
       }
       if (subProof2.P.get(i).getPLine4()>=iSize){
          subProof2.P.get(i).setPLine4
          (subProof2.P.get(i).getPLine4()+subProof1.P.size()-iSize);
       }
       // Πρόσθεση της γραμμής στην απόδειξη
       P.add((line)subProof2.P.get(i));
   }
   // Ενημέρωση του στόχου για να τερματίσει η απόδειξη
   updateGoalOrElim(number,iSize,subProof1.P.size()-1,subProof1.P.size(),P.size()-1);
   return true;  // Επιτυχία
}
```

```
void updateGoalOrElim(int pline0,int pline1,int pline2,int pline3,int pline4){
   // Πάρε τον κορυφαίο στόχο από την στοίβα
   line G=(line)S.pop();
   // Ο στόχος βγαίνει με or elimination
   G.setComment("\u2228elim");
   // Θέση όλων των χρησιμοποιούμενων γραμμών
   G.setPLine0(pline0);
   G.setPLine1(pline1);
   G.setPLine2(pline2);
   G.setPLine3(pline3);
   G.setPLine4(pline4);
   // Επανατοποθέτηση του στόχου στην στοίβα
   S.push(G);
}
boolean checkForSubroutine(){
   // Αποτυχία αν η θέση εκκίνησης του ελέγχου ξεπερνά το μέγεθος της απόδειξης
   if (start_of_subroutine>=P.size()) return false;
   // Σε περίπτωση εκτέλεσης κάποιου μετασχηματισμού θα δημιουργηθεί νέο επίπεδο
   Vector<line> newLevel=new Vector<line>();
   // Διάτρεξη της απόδειξης από την θέση εκκίνησης ελέγχου
   for (int i=start_of_subroutine; i<P.size(); i++){
       // Ανάλυση της τρέχουσας φόρμουλας
       Vector<String> A=AnalyzeFormula(P.get(i).getFormula());
       // Αν το βασικό συνδετικό είναι το not
       if (A.get(1).compareTo("\u00ac")==0){
          // Ανάλυση της υποφόρμουλας
          Vector<String> B=AnalyzeFormula((String)A.get(2));
          // Αν έχει ως βασικό συνδετικό το implies τότε είναι not(φ implies ψ)
          if (B.get(1).compareTo("\u2192")==0){
             // Πρέπει να γίνει εισαγωγή του φ and (not ψ) στην απόδειξη
             line newLine=new line(P.size(),
                                   addBrackets("",
                                               (String)B.get(0),
                                               "\u2227",
                                               "\u00ac",
                                               (String)B.get(2)),
                                    "\u00acTo\u2227 Transformation",i,-1);
             P.add(newLine);
             // Η νέα γραμμή μπαίνει και ως ένα ξεχωριστό επίπεδο
             newLevel.add(newLine);
             L.add(newLevel);
             start_of_subroutine=i+1;  // Ενημέρωση της θέσης εκκίνησης ελέγχου
             return true;  // Επιτυχία
          }
          // Αν η έχει ως βασικό συνδετικό το or τότε είναι not(φ or ψ)
          else if (B.get(1).compareTo("\u2228")==0){
             // Πρέπει να γίνει εισαγωγή του (not φ) and (not ψ) στην απόδειξη
             line newLine=new line(P.size(),
                                   addBrackets("\u00ac",
                                               (String)B.get(0),
```

```
                                                "\u2227",
                                                "\u00ac",
                                                (String)B.get(2)),
                                    "\u2228To\u2227 DM",i,-1);
            P.add(newLine);
            // Η νέα γραμμή μπαίνει και ως ένα ξεχωριστό επίπεδο
            newLevel.add(newLine);
            L.add(newLevel);
            start_of_subroutine=i+1;  // Ενημέρωση της θέσης εκκίνησης ελέγχου
            return true;  // Επιτυχία
         }
         // Αν έχει ως βασικό συνδετικό το and τότε not (φ and ψ)
         else if (B.get(1).compareTo("\u2227")==0){
            // Πρέπει να γίνει η εισαγωγή του (not φ) or (not ψ) στην απόδειξη
            line newLine=new line(P.size(),
                                  addBrackets("\u00ac",
                                              (String)B.get(0),
                                              "\u2228",
                                              "\u00ac",
                                              (String)B.get(2)),
                                  "\u2227To\u2228 DM",i,-1);
            P.add(newLine);
            // Η νέα γραμμή μπαίνει και ως ένα ξεχωριστό επίπεδο
            newLevel.add(newLine);
            L.add(newLevel);
            start_of_subroutine=i+1;  // Ενημέρωση της θέσης εκκίνησης ελέγχου
            return true;  // Επιτυχία
         }
      }
   }
   // Αποτυχία
   start_of_subroutine=P.size();
   return false;
}
boolean checkForImpliesToOR(){
   // Αποτυχία αν η θέση εκκίνησης του ελέγχου ξεπερνά το μέγεθος της απόδειξης
   if (start_of_impliesToOr>=P.size()) return false;
   // Σε περίπτωση εκτέλεσης κάποιου μετασχηματισμού θα δημιουργηθεί νέο επίπεδο
   Vector<line> newLevel=new Vector<line>();
   // Διάτρεξη της απόδειξης από την θέση εκκίνησης του ελέγχου
   for (int i=start_of_impliesToOr; i<P.size(); i++){
      // Ανάλυση της τρέχουσας φόρμουλας
      Vector<String> A=AnalyzeFormula(P.get(i).getFormula());
      // Αν το βασικό συνδετικό είναι το implies τότε είναι φ implies ψ
      if (A.get(1).compareTo("\u2192")==0){
         // Πρέπει να γίνει εισαγωγή του (not) φ or ψ στην απόδειξη
         line newLine=new line(P.size(),
                               addBrackets("\u00ac",
                                           (String)A.get(0),
                                           "\u2228",
```

```
                                                    "",
                                                    (String)A.get(2)),
                                        "\u2192To\u2228 Tranformation",i,-1);
            P.add(newLine);
            // Η νέα γραμμή μπαίνει και ως ένα ξεχωριστό επίπεδο
            newLevel.add(newLine);
            L.add(newLevel);
            start_of_impliesToOr=i+1;  // Ενημέρωση της θέσης εκκίνησης ελέγχου
            return true;  // Επιτυχία
         }
      }
      // Αποτυχία
      start_of_impliesToOr=P.size();
      return false;
   }
   String addBrackets(String c1,String p,String c2,String c3,String q){
      // Δήλωση ενός νέου string το οποίο θα επιστραφεί
      String s=new String();
      // Ανάλυση των υποφορμουλών p και του q
      Vector<String> A=AnalyzeFormula(p);
      Vector<String> B=AnalyzeFormula(q);
      // Αν είναι και τα δύο προτασιακά γράμματα
      if ( A.get(1).equals("") && B.get(2).equals("") ){
         // Αν p και q δεν έχουν συνδετικά μπροστά τους
         if (c1.equals("") && c3.equals("")) s=p+c2+q;
         // Διαφορετικά αν μόνο το p έχει συνδετικό μπροστά του
         else if (!c1.equals("") && c3.equals) s="("+c1+p+")"+c2+q;
         // Διαφορετικά αν μόνο το q έχει συνδετικό μπροστά του
         else if (c1.equals("") && !c3.equals("")) s=p+c2+"("+c3+q+")";
         // Διαφορετικά και τα δύο έχουν συνδετικά
         else s="("+c1+p+")"+c2+"("+c3+q+")";
      }
      // Αν μόνο το p είναι προτασιακό γράμμα
      else if ( A.get(1).equals("") && !B.get(2).equals("")){
         // Αν p και q δεν έχουν συνδετικά μπροστά τους
         if (c1.equals("") && c3.equals("")) s=p+c2+"("+q+")";
         // Διαφορετικά αν μόνο το p δεν έχει συνδετικό μπροστά του τότε
         if (c1.equals("") && !c3.equals("")) s=p+c2+"("+c3+"("+q+")"+")";
         // Διαφορετικά αν μόνο το q δεν έχει συνδετικό μπροστά του τότε
         else if (!c1.equals("") && c3.equals("")) s="("+c1+p+")"+c2+"("+q+")";
         // Διαφορετικά
         else s="("+c1+p+")"+c2+"("+c3+"("+q+")"+")";
      }
      // Αν μόνο το q είναι προτασιακό γράμμα
      else if ( !A.get(1).equals("") && B.get(2).equals("")){
         // Αν p και q δεν έχουν συνδετικά μπροστά τους
         if (c1.equals("") && c3.equals("")) s="("+p+")"+c2+q;
         // Διαφορετικά αν μόνο το p δεν έχει συνδετικό μπροστά του τότε
         if (c1.equals("") && !c3.equals("")) s="("+p+")"+c2+"("+c3+q+")";
```

```
      // Διαφορετικά αν μόνο το q δεν έχει συνδετικό μπροστά του τότε
      else if (!c1.equals("") && c3.equals("")) s="("+c1+"("+p+")"+")"+c2+q;
      // Διαφορετικά
      else s="("+c1+"("+p+")"+")"+c2+"("+c3+q+")";
   }
   // Διαφορετικά κανένα δεν είναι προτασιακό γράμμα
   else{
      // Αν p και q δεν έχουν συνδετικά μπροστά τους
      if (c1.equals("") && c3.equals("")) s="("+p+")"+c2+"("+q+")";
      // Διαφορετικά αν μόνο το p δεν έχει συνδετικό μπροστά του τότε
      if (c1.equals("") && !c3.equals("")) s="("+p+")"+c2+"("+c3+"("+q+")"+")";
      // Διαφορετικά αν μόνο το q δεν έχει συνδετικό μπροστά του τότε
      else if (!c1.equals("") && c3.equals(""))
         s="("+c1+"("+p+")"+")"+c2+"("+q+")";
      // Διαφορετικά
      else s="("+c1+"("+p+")"+")"+c2+"("+c3+"("+q+")"+")";
   }
   return s;  // Επιστροφή του string με παρενθέσεις αν χρειάστηκε
}
```