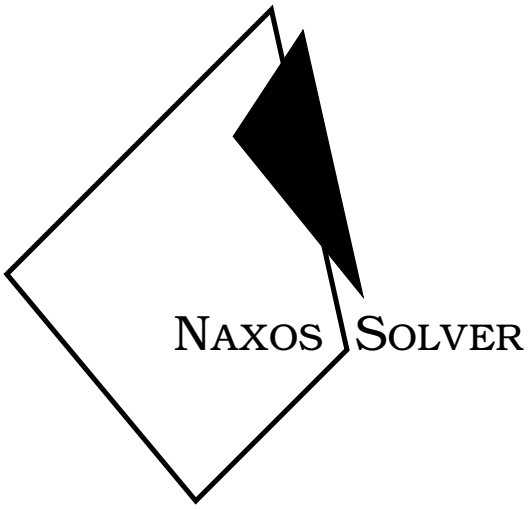


Εγχειρίδιο Χρήσης

Νίκος Ποθητός

1 Ιουνίου 2009

Περιεχόμενα

1 Εισαγωγή

Ο Naxos Solver είναι ένας *επιλυτής* (solver) προβλημάτων ικανοποίησης περιορισμών. Πήρε το όνομα «Naxos» από το νησί[1] στο οποίο άρχισε να χτίζεται –και στο οποίο έφθασε σε ένα τελικό, ώριμο στάδιο. Υλοποιήθηκε στα πλαίσια πτυχιακής εργασίας του συγγραφέα με επιβλέποντα καθηγητή τον Παναγιώτη Σταματόπουλο στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Αθηνών. Ο σκοπός αυτού του εγχειριδίου είναι να δώσει τις πληροφορίες που χρειάζεται ένας προγραμματιστής εφαρμογών (application developer) της βιβλιοθήκης.

Ένα *πρόβλημα ικανοποίησης περιορισμών* (constraint satisfaction problem – CSP) περιέχει ένα σύνολο από *περιορισμένες μεταβλητές* (constrained variables) –μπορούμε να τις αναφέρουμε απλά και σαν *μεταβλητές*· κάθε μεταβλητή αντιστοιχεί σε ένα *πεδίο τιμών* (domain). Οι περιορισμένες μεταβλητές συνδέονται μεταξύ τους μέσα από ένα σύνολο *περιορισμών* (constraints). Γενικά, ένας περιορισμός που αφορά συγκεκριμένες περιορισμένες μεταβλητές είναι ένα σύνολο με όλους τους έγκυρους συνδυασμούς τιμών που μπορούν να ανατεθούν. Για παράδειγμα, για τις μεταβλητές x_1 και x_2 με πεδία τιμών $\{0, 1, 2, 3\}$, ο περιορισμός της ισότητας μπορεί να δηλωθεί σαν $\mathcal{C}(\{x_1, x_2\}, \{(0,0), (1,1), (2,2), (3,3)\})$. Παρότι αυτός ο συμβολισμός είναι όσο πιο γενικός γίνεται, στην πράξη (δηλαδή στον Προγραμματισμό με Περιορισμούς) χρησιμοποιούμε απλές σχέσεις για να περιγράψουμε τα δίκτυα περιορισμών. Στο παραπάνω παράδειγμα, ο περιορισμός μπορεί να γραφεί απλά σαν $x_1 = x_2$. Μία *λύση* σε ένα πρόβλημα ικανοποίησης περιορισμών είναι μία έγκυρη ανάθεση μίας τιμής σε κάθε μεταβλητή, η οποία ικανοποιεί όλους τους περιορισμούς. Τέλος, αξίζει να σημειωθεί ότι το πλεονέκτημα του Προγραμματισμού με Περιορισμούς (Constraint Programming) είναι ότι επιτρέπει τον διαχωρισμό της διαδικασίας διατύπωσης ενός προβλήματος από τον μηχανισμό ο οποίος το επιλύει.

Ο Naxos Solver είναι μια βιβλιοθήκη που επιτρέπει τη λύση προβλημάτων μέσω προγραμματισμού με περιορισμούς, η οποία σχεδιάστηκε για το αντικειμενοστρεφές περιβάλλον της γλώσσας C++. Ο επιλυτής είναι ασφαλής για χρήση σε πολυνηματικό περιβάλλον (threadsafe). Δεν θα πρέπει να χρησιμοποιούνται «εσωτερικές» κλάσεις, μέθοδοι, συναρτήσεις του κ.λπ. οι οποίες δεν περιγράφονται στο εγχειρίδιο χρήσης, καθώς μπορεί να αλλάξουν στο μέλλον. Ακόμα, για αποφυγή παρεξηγήσεων, καλό είναι να μην ονομά-

[1] Πιο συγκεκριμένα, αναφερόμαστε στο χωριό Κορωνίδα της Νάξου.

ζουμε τις δικές μας μεταβλητές, κλάσεις κ.λπ. με ονόματα που ξεκινούν από «Ns», καθώς αυτό είναι το πρόθεμα για τις ενσωματωμένες κλάσεις και σταθερές του επιλυτή. Τέλος, σημειώνεται ότι ο επιλυτής δεν κάνει κανένα έλεγχο για τυχόν υπερχειλίσεις (π.χ. κατά την πρόσθεση δύο μεγάλων ακεραίων), για λόγους απόδοσης.

Μέρος του σχεδιασμού και της «ονοματολογίας» του επιλυτή είναι επηρεασμένο από τη μοντελοποίηση της Standard Template Library (STL) για τη C++.[2] Π.χ. χρησιμοποιήθηκαν και υλοποιήθηκαν αρκετοί επαναλήπτες (iterators).

Δεν έχουμε διάκριση των κλάσεων του Naxos Solver σε κλάσεις-χειριστήρια (handle-classes) και κλάσεις-υλοποίησης (implementation-classes), όπως γίνεται στον Ilog Solver. Ο λόγος αυτής της διαφοροποίησης στον Ilog Solver, έχει να κάνει με την προσπάθεια να διαχειριστεί αυτόματα τη μνήμη αλά Java. Σε κάθε κλάση-χειριστήριο, υπάρχει μία αναφορά σε μία κλάση-υλοποίησης. Είναι δυνατόν πολλές κλάσεις-χειριστήρια να δείχνουν στην ίδια κλάση-υλοποίησης. Η κλάση-υλοποίησης διαγράφεται από τη μνήμη, μόνο όταν πάψει να υφίσταται και η τελευταία κλάση-χειριστήριο που δείχνει σε αυτήν. (Κάτι αντίστοιχο γίνεται και στην Java με τις αναφορές.) Επομένως, στον Ilog Solver είναι δυνατόν π.χ. σε μία συνάρτηση να δημιουργούμε αυτόματες μεταβλητές για να περιγράψουμε έναν περιορισμό και αυτός, καθώς και οι μεταβλητές που αφορά, να υπάρχουν και μετά το τέλος της συνάρτησης· σε αντίστοιχη περίπτωση στον Naxos Solver θα έχουμε σφάλμα κατάτμησης (segmentation fault).

2 Διαχείριση Σφαλμάτων

Ξεκινάμε από τη διαχείριση σφαλμάτων, για την οποία είναι σημαντικό να φροντίζουμε, όποιο πρόγραμμα και να γράφουμε. Στον Naxos Solver πρέπει να συλλαμβάνονται οι εξαιρέσεις τύπου `NsException`. Η κλάση αυτή είναι υποκλάση της `logic_error`, η οποία με τη σειρά της παράγεται από την `exception`. Αρκεί λοιπόν να συλλαμβάνουμε τον τύπο `exception`· με τη μέθοδο `what()` αυτής της βασικής κλάσης, θα πάρουμε σε μία συμβολοσειρά ένα μήνυμα για το σφάλμα που συνέβη.

```
#include <naxos.h>
```

[2] B. Eckel and C. Allison. *Thinking In C++ Vol. 2: Practical Programming*. Prentice Hall, 2nd edition, 2003.

```
using namespace naxos;
using namespace std;

int  main (void)
{
    try  {

        // ... CODE OF THE PROGRAM ... //

    } catch (exception& exc)  {
        cerr << exc.what() << "\n";

    } catch (...)  {
        cerr << "Unknown exception" << "\n";
    }
}
```

Δεν είναι σωστή προγραμματιστική τακτική να εντάσσουμε τις εξαιρέσεις στο σώμα των αλγορίθμων μας. Συνήθως, οι εξαιρέσεις αποτελούν το «περιτύλιγμα» των προγραμμάτων μας.

3 Περιορισμένες Μεταβλητές

Ο επιλυτής υποστηρίζει *ακέραιες περιορισμένες μεταβλητές πεπερασμένων πεδίων* (finite domain integer constrained variables). Η κλάση με τις οποίες αναπαρίστανται είναι η `NsIntVar`, η οποία περιέχει τις παρακάτω μεθόδους.

`NsIntVar(NsProblemManager& pm, NsInt min, NsInt max)` Είναι μία μέθοδος-κατασκευής (constructor). Το `pm` είναι ο διαχειριστής προβλήματος στον οποίον ανήκει η μεταβλητή (βλ. § 5) και τα `min` και `max`, το ελάχιστο και το μέγιστο του πεδίου τιμών της, που συμβολίζεται και με `[min..max]`.

Ο τύπος `NsInt` μπορεί να θεωρηθεί ότι είναι σε θέση να αναπαραστήσει τουλάχιστον τους ακέραιους αριθμούς εκείνους που είναι δυνατόν να αναπαρασταθούν με έναν `long`. Η ελάχιστη τιμή του τύπου `NsInt`, ισούται με τη σταθερά `NsMINUS_INF` και η μέγιστη με `NsPLUS_INF`. (Για τον μη προσημασμένο τύπο `NsUInt`, η μέγιστη τιμή είναι `NsUPLUS_INF`.)

Η ελάχιστη τιμή ενός πεδίου πρέπει να είναι γνήσια μεγαλύτερη από `NsMINUS_INF`, όπως και η μέγιστη τιμή πρέπει να είναι γνήσια μικρότερη από `NsPLUS_INF`, καθώς αυτές οι σταθερές αποτελούν ειδικές τιμές που αντιπροσωπεύουν το άπειρο, όπως θα δούμε και παρακάτω.

`NsIntVar()` Στο στιγμιότυπο της κλάσης που δημιουργείται με αυτήν τη μέθοδο-κατασκευής, είναι δυνατόν στη συνέχεια να ανατεθεί κάποια έκφραση (μέσω και του υπερφορτωμένου τελεστή «=»), όπως στην τρίτη γραμμή του παρακάτω παραδείγματος.

```
NsIntVar  X(pm,0,3), Y(pm,-1,15), Z;
NsIntVar  W = X + 3*Y;
Z = W * W;
```

Στη δεύτερη γραμμή του παραδείγματος, χρησιμοποιήθηκε μία άλλη μέθοδος-κατασκευής, που παίρνει σαν όρισμα μία *Expression* (εδώ, η έκφραση ήταν η «`X + 3*Y`»).

`void remove(NsInt val)` Αφαιρεί την τιμή `val` από το πεδίο της μεταβλητής.

`void remove(NsInt min, NsInt max)` Αφαιρεί από το πεδίο της μεταβλητής τις τιμές που βρίσκονται στο διάστημα $[\texttt{min}, \texttt{max}]$. Αντί να τις αφαιρούμε μία-μία με τη `remove(val)`, είναι πιο αποδοτικό να καλούμε αυτή τη μέθοδο.

`void removeAll()` «Αδειάζει» το πεδίο τιμών της μεταβλητής. Πρακτικά, αυτό που γίνεται είναι η ενημέρωση του διαχειριστή προβλήματος, ότι υπήρξε ασυνέπεια –λόγω του κενού πεδίου τιμών. Αυτή η μέθοδος είναι χρήσιμη όταν θέλουμε να προκαλέσουμε αποτυχία στην αναζήτηση. Π.χ. όταν θέλουμε να αποτύχει ένας στόχος, κατά την εκτέλεσή του, καλούμε αυτή τη μέθοδο για μία οποιαδήποτε μεταβλητή.[3]

`void set(NsInt val)` Αναθέτει την τιμή `val` στη μεταβλητή· συνεπώς, η μεταβλητή γίνεται δεσμευμένη.

Ακολουθούν οι μέθοδοι που δεν προκαλούν καμία αλλαγή στη μεταβλητή για την οποία καλούνται.

[3] Ενώ για να δείξουμε ότι ένας στόχος πετυχαίνει, κάνουμε την `GOAL()` να επιστρέψει `0`, για να δείξουμε ότι απέτυχε, υπάρχει αυτός ο λιγότερο κομψός τρόπος. (Περισσότερα για τον μηχανισμό στόχων υπάρχουν στην § 8.)

`bool contains(NsInt val)` Επιστρέφει `true` αν η τιμή `val` ανήκει στο πεδίο της μεταβλητής.

`NsInt min()` Η ελάχιστη τιμή του πεδίου τιμών.

`NsInt max()` Η μέγιστη τιμή του πεδίου τιμών.

`NsUInt size()` Αριθμός των τιμών που περιέχονται στο πεδίο τιμών.

`bool isBound()` Επιστρέφει `true` αν η μεταβλητή είναι δεσμευμένη.

`NsInt value()` Χρησιμοποιείται όταν η μεταβλητή είναι δεσμευμένη και επιστρέφει τη (μοναδική) τιμή της. Αν κληθεί για μία μη δεσμευμένη μεταβλητή, προκαλείται σφάλμα.

Αντί αυτής, θα μπορούσε να χρησιμοποιηθεί μία μέθοδος, από τις `min()` και `max()`. Ο λόγος ύπαρξης αυτής της μεθόδου, αφορά την αναγνωσιμότητα του κώδικα.

`NsInt next(NsInt val)` Επιστρέφει τη μικρότερη τιμή στο πεδίο τιμών, που είναι γνήσια μεγαλύτερη από την `val`. Αν δεν υπάρχει κάποια τέτοια τιμή, επιστρέφεται `NsPLUS_INF`. Η `val` μπορεί να πάρει και τις ειδικές τιμές `NsMINUS_INF` και `NsPLUS_INF`.

`NsInt previous(NsInt val)` Ορίζεται ανάλογα με τη `next()`. Επιστρέφει τη μεγαλύτερη τιμή στο πεδίο τιμών, που είναι γνήσια μικρότερη από την `val`. Αν δεν υπάρχει κάποια τέτοια τιμή, επιστρέφεται `NsMINUS_INF`. Η `val` μπορεί να πάρει και τις ειδικές τιμές `NsMINUS_INF` και `NsPLUS_INF`.

Στη συνέχεια παρουσιάζονται δύο επαναλήπτες για την κλάση και παραδείγματα χρήσης τους.

`NsIntVar::const_iterator` Διατρέχει τις τιμές του πεδίου της μεταβλητής. Π.χ. με τον παρακάτω κώδικα, τυπώνονται οι τιμές της μεταβλητής, σε αύξουσα σειρά.

```
for (NsIntVar::const_iterator v = Var.begin();
                              v != Var.end();   ++v)
        cout << *v << "\n";
```

NsIntVar::const_reverse_iterator Διατρέχει τις τιμές του πεδίου της μεταβλητής με αντίστροφη σειρά. Π.χ. με τον παρακάτω κώδικα, τυπώνονται οι τιμές της μεταβλητής, σε φθίνουσα σειρά.

```
for (NsIntVar::const_reverse_iterator v=Var.rbegin();
                          v != Var.rend();   ++v)
        cout << *v << "\n";
```

NsIntVar::const_gap_iterator Διατρέχει τα κενά που υπάρχουν εντός του πεδίου της μεταβλητής. Αν χρησιμοποιούσαμε τη γλώσσα των μαθηματικών, θα λέγαμε ότι παίρνουμε όλες τις τιμές του συμπληρώματος του πεδίου τιμών της μεταβλητής, ως προς το $[\texttt{min}, \texttt{max}]$ (όπου `min` το ελάχιστο και `max` το μέγιστο του πεδίου τιμών της μεταβλητής). Π.χ.

```
for (NsIntVar::const_gap_iterator g = Var.gap_begin();
                             g != Var.gap_end();  ++g)
        cout << *g << "\n";
```

Τέλος, σημειώνεται ότι ο τελεστής «<<» έχει υπερφορτωθεί, για να είναι δυνατόν να τυπώνεται μία μεταβλητή σε ένα ρεύμα εξόδου, γράφοντας π.χ. «`cout << Var;`».

4 Πίνακες Περιορισμένων Μεταβλητών

Ο προκαθορισμένος τύπος (ευέλικτου) πίνακα του επιλυτή, με στοιχεία περιορισμένες μεταβλητές, είναι ο `NsIntVarArray`. Μπορούμε να αναφερόμαστε στο i-οστό στοιχείο ενός τέτοιου πίνακα `VarArr`, μέσω της γνωστής παράστασης `VarArr[`i`]`. Ο προκαθορισμένος τύπος για το i, είναι ο `NsIndex`.

Μετά τη δημιουργία ενός `NsIntVarArray`, αυτός δεν περιέχει κανένα στοιχείο. Οι μεταβλητές που τον συνθέτουν, εισάγονται είτε στην αρχή, είτε στο τέλος του, όπως μπορεί να γίνει σε μία συνδεδεμένη λίστα. Η μέθοδος-κατασκευής του δεν παίρνει ορίσματα. Ακολουθούν οι υπόλοιπες μέθοδοι αυτής της κλάσης.

NsIndex size() Επιστρέφει το μέγεθος του πίνακα.

bool empty() Επιστρέφει `true` αν ο πίνακας είναι άδειος.

NsIntVar& back() Η τελευταία περιορισμένη μεταβλητή του πίνακα.

`NsIntVar& front()` Η πρώτη περιορισμένη μεταβλητή του πίνακα.

`void push_back(`*Expression***`)`** Εισαγωγή στο τέλος του πίνακα, της μεταβλητής που θα ισούται με την έκφραση *Expression*. Όπως θα αναλυθεί σε επόμενη παράγραφο, μία *Expression* μπορεί απλά να είναι μία περιορισμένη μεταβλητή, ή κάποιος συνδυασμός μεταβλητών. Π.χ.

```
VarArr.push_back( NsIntVar(pm, 10, 30) );
VarArr.push_back( 3*VarX + VarY );
VarArr.push_back( VarX > 11 );
```

Στην πρώτη εντολή παραπάνω, ουσιαστικά εισάγαμε στο τέλος του πίνακα `VarArr` μία καινούργια μεταβλητή με πεδίο `[10..30]`.

Με την τελευταία εντολή, εισάγουμε μία περιορισμένη μεταβλητή στο τέλος του πίνακα, που θα είναι `1` αν ισχύει `VarX > 11`, αλλιώς `0`. (Μπορεί φυσικά το πεδίο της να είναι και το `[0..1]`, σε περίπτωση που κάποιες τιμές της `VarX` είναι μικρότερες και κάποιες μεγαλύτερες του `11`.) Έχουμε δηλαδή έναν μετα-περιορισμό.

`void push_front(`*Expression***`)`** Όπως η `push_back()`, αλλά η εισαγωγή γίνεται στην αρχή του πίνακα.

Ακολουθούν οι επαναλήπτες για τους πίνακες. Χρησιμοποιώντας τους, μπορούμε να διατρέχουμε εύκολα όλες τις μεταβλητές του πίνακα.

`NsIntVarArray::iterator` Επαναλήπτης για να παίρνουμε κατά σειρά τις μεταβλητές του πίνακα. Π.χ. παρακάτω αφαιρούμε την τιμή 2 από όλες τις μεταβλητές του `VarArr`:

```
for (NsIntVarArray::iterator  V = VarArr.begin();
                           V != VarArr.end();   ++V)
        V->remove(2);
```

`NsIntVarArray::const_iterator` Αυτός ο επαναλήπτης υπάρχει για την περίπτωση που δεν θέλουμε να τροποποιήσουμε τις μεταβλητές. Μπορούμε π.χ. χρησιμοποιώντας τον, να τυπώσουμε τις μεταβλητές ενός πίνακα.

```
for (NsIntVarArray::const_iterator  V=VarArr.begin();
                                 V != VarArr.end(); ++V)
        cout << *V << "\n";
```

Τέλος, σημειώνεται ότι ο τελεστής «<<» έχει υπερφορτωθεί και για τους πίνακες, οι οποίοι μπορούν να τυπωθούν, γράφοντας π.χ. «`cout<<VarArr;`».

4.1 Πίνακες Γενικής Χρήσης

Στην περίπτωση που θέλουμε να δημιουργήσουμε έναν πίνακα με τις δυνατότητες/μεθόδους του `NsIntVarArray`, μπορούμε να χρησιμοποιήσουμε την template κλάση `NsDeque<`*τύπος_στοιχείων*`>`. Π.χ. με το

```
NsDeque<int>  arr;
```

δηλώνουμε ότι ο `arr` είναι ένας ευέλικτος πίνακας ακεραίων, κενός αρχικά, ενώ με το

```
NsDeque<int>  arr(5);
```

δηλώνουμε ότι αρχικά θα περιέχει `5` στοιχεία. Ο *τύπος_στοιχείων* δεν έχει νόημα να είναι `NsIntVar`, αφού για αυτήν την περίπτωση αντί για `NsDeque` μπορούμε να χρησιμοποιήσουμε απευθείας τον `NsIntVarArray`.[4]

5 Διαχειριστής Προβλήματος

Πριν ξεκινήσουμε να διατυπώνουμε ένα πρόβλημα, πρέπει να ορίσουμε ένα *διαχειριστή προβλήματος* (κλάση `NsProblemManager`). Ο διαχειριστής αυτός συγκρατεί όσες πληροφορίες χρειάζονται για τις μεταβλητές, το δίκτυο περιορισμών που κατασκευάζεται και τους στόχους που θα εκτελεστούν. Η μέθοδος-κατασκευής του δεν παίρνει ορίσματα. Ακολουθούν οι υπόλοιπες μέθοδοι.

`void add(`*ExprConstr***`)`** Προσθέτει τον περιορισμό που περιγράφει η έκφραση περιορισμού *ExprConstr* (βλ. § 6.1). Σε μία έκφραση περιορισμού χρησιμοποιούνται οι τελεστές των συνθηκών (`<`, `==`, `!=` κ.λπ.), ή ενσωματωμένες εκφράσεις όπως η `NsAllDiff()` που επιβάλλει όλα τα στοιχεία ενός πίνακα να είναι διαφορετικά. Π.χ.

[4] Ο `NsDeque` πρόκειται ουσιαστικά για μια επέκταση του τύπου `std::deque` της πρότυπης βιβλιοθήκης της C++. Η ειδοποιός διαφορά τους είναι ότι στον `NsDeque` γίνεται πάντα ο έλεγχος για το αν κινούμαστε εντός των ορίων του πίνακα· σε περίπτωση που υπερβούμε τα όρια, προκαλείται η αντίστοιχη εξαίρεση.

```
pm.add( 3*VarX != VarY/2 );
pm.add( VarW == -2 || VarW >= 5 );
pm.add( NsAllDiff(VarArr) );
```

`void addGoal(NsGoal* goal)` Προσθήκη του `goal` στη λίστα με τους προς ικανοποίηση στόχους (βλ. § 8).

`bool nextSolution()` Εύρεση της επόμενης λύσης. Ικανοποίηση των στόχων που έχουν τεθεί. Αν δεν βρεθεί λύση, επιστρέφεται `false`.

`void minimize(`*Expression***`)`** Δίνει οδηγία στον επιλυτή να προσπαθήσει να ελαχιστοποιήσει την τιμή της *Expression*. Κάθε φορά που ο επιλυτής βρίσκει μία λύση, θα πρέπει η τιμή της *Expression* να είναι μικρότερη από αυτήν της προηγούμενης λύσης (αλγόριθμος branch-and-bound). Ουσιαστικά, μετά από κάθε λύση που δίνει η `nextSolution()`, αν η μέγιστη τιμή της *Expression* είναι a, επιβάλλεται ο περιορισμός *Expression* $< a$, για την επόμενη φορά που θα κληθεί η `nextSolution()`. Δηλαδή σε κάθε λύση, η *Expression* βγαίνει μειωμένη. Αν δεν μπορεί να μειωθεί περαιτέρω, η `nextSolution()` θα επιστρέψει `false` και θα πρέπει να έχουμε αποθηκευμένη κάπου την τελευταία (βέλτιστη) λύση. Π.χ.

```
pm.minimize( VarX + VarY );
while (pm.nextSolution() != false)
    {  /* STORE SOLUTION */  }
```

Αν επιθυμούμε να μεγιστοποιήσουμε μία *Expression*, απλά βάζουμε ένα «`-`» μπροστά της και καλούμε τη `minimize()` για αυτήν.

`void objectiveUpperLimit(NsInt max)` Κατά τη διάρκεια της αναζήτησης η μέθοδος αυτή θέτει ένα άνω φράγμα ίσο με `max` για το κόστος της λύσης ενός προβλήματος (που εκφράζεται από τη μεταβλητή που παίρνει ως όρισμα η `NsProblemManager::minimize()`). Με άλλα λόγια θέτει τον μοναδιαίο περιορισμό «*μεταβλητή_κόστους* $\leq$ `max`».

Εξάλλου, όταν έχει ήδη ξεκινήσει η αναζήτηση –με μία κλήση της `nextSolution()`–, ο επιλυτής δεν υποστηρίζει πλέον την επιβολή νέων περιορισμών μέσω της `NsProblemManager::add`, όπως π.χ. στο «`pm.add(X <= 5)`». Η `objectiveUpperLimit` έρχεται να καλύψει αυτό το κενό, όσον αφορά τη μεταβλητή κόστους.

`void timeLimit(unsigned long secs)` Η αναζήτηση θα διαρκέσει το πολύ `secs` δευτερόλεπτα. Μετά το πέρας αυτού του χρονικού διαστήματος, η `nextSolution()` επιστρέφει `false`.

`void realTimeLimit(unsigned long secs)` Κάνει ό,τι και η προηγούμενη μέθοδος, με τη διαφορά ότι εδώ τα `secs` δευτερόλεπτα είναι πραγματικός χρόνος, ενώ στην προηγούμενη μέθοδο ήταν ο καθαρός χρόνος που έχει δοθεί από το σύστημα, αποκλειστικά στον επιλυτή (CPU time).

`void backtrackLimit(unsigned long x)` Για μη μηδενικό `x`, κάνει τη `nextSolution()` να επιστρέφει `false` μετά από `x` οπισθοδρομήσεις από τη στιγμή που κλήθηκε αυτή η μέθοδος.

`unsigned long numFailures()` Επιστρέφει τον αριθμό των αποτυχιών που έχουν συμβεί κατά την αναζήτηση.

`unsigned long numBacktracks()` Επιστρέφει τον αριθμό των οπισθοδρομήσεων που έχουν γίνει κατά στην αναζήτηση.

`unsigned long numGoals()` Επιστρέφει τον αριθμό των στόχων που έχουν εκτελεστεί.

`unsigned long numVars()` Επιστρέφει τον αριθμό των περιορισμένων μεταβλητών που έχουν δημιουργηθεί. Σημειώνεται ότι στον αριθμό συμπεριλαμβάνονται και τυχόν ενδιάμεσες/βοηθητικές μεταβλητές που έχουν δημιουργηθεί αυτόματα από τον επιλυτή.

`unsigned long numConstraints()` Επιστρέφει τον αριθμό των περιορισμών που υπάρχουν. Σημειώνεται ότι στον αριθμό συμπεριλαμβάνονται και τυχόν ενδιάμεσοι περιορισμοί που έχουν δημιουργηθεί αυτόματα από τον επιλυτή.

`unsigned long numSearchTreeNodes()` Επιστρέφει τον αριθμό των κόμβων του δένδρου αναζήτησης λύσης που έχει επισκεφθεί –έως τώρα– ο επιλυτής.

`void searchToGraphFile(char *fileName)` Αποθηκεύει σε ένα αρχείο με όνομα `fileName` μία αναπαράσταση του δένδρου αναζήτησης λύσης. Η μορφή του αρχείου ονομάζεται `dot` και η εφαρμογή Graphviz υποστηρίζει τη γραφική απεικόνισή του.

`void restart()` Επαναφέρει τις περιορισμένες μεταβλητές (δηλαδή τα πεδία τιμών τους) στην κατάσταση ακριβώς που ήταν λίγο μετά την *πρώτη* κλήση της `nextSolution()`. Συγκεκριμένα, επαναφέρει την κατάσταση που υπήρχε ακριβώς πριν την κλήση της `nextSolution()`, συν τις όποιες αλλαγές έγιναν από την πρώτη επιβολή συνέπειας-ακμών (βλ. § 8 για τον ορισμό της τελευταίας έννοιας). Με άλλα λόγια, αυτή η μέθοδος φέρνει το δίκτυο περιορισμών/μεταβλητών στην πρώτη κατάσταση συνέπειας-ακμών που έχει βρεθεί ποτέ (πριν εκτελεστεί ο οποιοσδήποτε στόχος).

Η μέθοδος ακυρώνει επίσης όλους τους στόχους που επρόκειτο να εκτελεστούν. Δηλαδή, αν επιθυμούμε να ξαναξεκινήσει η αναζήτηση (κλήση `nextSolution()`) μετά από μία `restart()`, πρέπει να δηλώσουμε κάποιον στόχο να εκτελεστεί (μέσω της `addGoal()`), ειδάλλως δεν υπάρχει στόχος!

Η `restart()` *δεν* πειράζει την μεταβλητή που έχει περαστεί σαν όρισμα της `minimize()` –λέγεται και *αντικειμενική μεταβλητή* ή *μεταβλητή κόστους*. Ήτοι *δεν* την επαναφέρει στην αρχική της κατάσταση. Δηλαδή αν αρχικά η μεταβλητή κόστους είχε τιμές `[0..100]` και πριν κληθεί η `restart()` έχει τιμές `[0..10]`, τότε μετά την κλήση της `restart()` θα έχει τιμές πάλι `[0..10]`.

Η μέθοδος δεν θα πρέπει να καλείται μέσα από στόχους, αλλά έξω από αυτούς. Π.χ. μπορεί να καλείται εκεί που καλείται και η `nextSolution()`.

6 Εκφράσεις

Για να συνδέσουμε τις μεταβλητές, εκμεταλλευόμαστε τους υπερφορτωμένους τελεστές και δημιουργούμε εκφράσεις και συνδυασμούς από εκφράσεις. Μία απλή έκφραση μπορεί να είναι και μία μεταβλητή ή ένας ακέραιος. Μία έκφραση συμβολίζεται με *Expression*.

6.1 Εκφράσεις Περιορισμών

Συμβολίζονται με *ExprConstr* και είναι υποκατηγορία των εκφράσεων *Expression*. Χρησιμοποιούνται κυρίως σαν ορίσματα της `NsProblemManager::`

`add()` και για τη δημιουργία μετα-περιορισμών. Οι παρακάτω παραστάσεις είναι *ExprConstr*:

- *Expression*$_1$ op *Expression*$_2$, $\quad$ op $\in \{$`<`, `<=`, `>`, `>=`, `==`, `!=`$\}$
- `!(` *ExprConstr* `)`
- *ExprConstr*$_1$ op *ExprConstr*$_2$, $\quad$ op $\in \{$`&&`, `||`$\}$
- `NsIfThen(` *ExprConstr*$_1$ `,` *ExprConstr*$_2$ `)`
- `NsEquiv(` *ExprConstr*$_1$ `,` *ExprConstr*$_2$ `)`
- `NsAllDiff(` $VarArr$ `)`

Ο ορισμός είναι λοιπόν αναδρομικός. Η τελευταία έκφραση συμβολίζει ότι οι περιορισμένες μεταβλητές του $VarArr$ (που είναι πίνακας του τύπου `NsIntVarArray`) είναι διαφορετικές μεταξύ τους.[5]

Ένας περιορισμός `NsIfThen(`p`,`q`)` έχει την έννοια της λογικής πρότασης $p \Rightarrow q$, ενώ ο περιορισμός `NsEquiv(`p`,`q`)` έχει την έννοια του $p \Leftrightarrow q$.[6]

Παρακάτω παρουσιάζονται μερικές εκφράσεις περιορισμών:

```
VarX < VarY
!( X==Y  ||  X+Y==-3 )
(X==Y) != 1
```

6.2 Γενικές Εκφράσεις

Εκτός από τις *ExprConstr*, στην κατηγορία των γενικών εκφράσεων *Expression*, ανήκουν και οι εξής παραστάσεις:

- *Expression*$_1$ op *Expression*$_2$, $\quad$ op $\in \{$`+`, `-`, `*`, `/`, `%`$\}$

[5] Εφόσον χρησιμοποιηθεί η έκφραση `NsAllDiff(` $VarArr$`,` $Capacity$ `)`, όπου $Capacity$ θετικός ακέραιος, τότε θα γίνει μεγαλύτερη διάδοση περιορισμών. Π.χ. αν $VarArr = \{$`[1..2]`, `[1..2]`, `[1..5]`$\}$, τότε από την έκφραση `NsAllDiff(` $VarArr$`,1 )` θα προκύψει ότι $VarArr = \{$`[1..2]`,`[1..2]`,`[3..5]`$\}$ (ενώ από την απλή έκφραση `NsAllDiff(` $VarArr$ `)` θα είχαμε κάποια αφαίρεση τιμής μόνο κατά την ανάθεση τιμής σε μεταβλητή του $VarArr$). Ωστόσο, αυτή η ισχυρότερη μορφή συνέπειας είναι πιθανόν να «κοστίσει» σε χρόνο. Τέλος, με τον ακέραιο $Capacity$ δηλώνεται ο αριθμός των εμφανίσεων που μπορεί να έχει το πολύ, μία τιμή εντός του πίνακα $VarArr$.

[6] Οι δύο περιορισμοί μπορούν να εκφραστούν και με έναν άλλον –ισοδύναμο– τρόπο στον επιλυτή σαν `(` `!`p `||` q `)` και `(` p `==` q `)` αντίστοιχα.

- `NsAbs(` $Expression$ `)`
- `NsMin(` $VarArr$ `)`
- `NsMax(` $VarArr$ `)`
- `NsSum(` $VarArr$`,` $start$`,` $length$ `)`
- $IntArr$`[` $Expression$ `]`

Μία ***Expression*** μπορεί –εκτός από το να συμμετέχει σε μία έκφραση περιορισμού– να ανατεθεί σε μία μεταβλητή. Π.χ. μπορούμε να γράψουμε

```
NsIntVar  X = Y + 3/Z;
NsIntVar  W = NsSum(VarArrA) - (NsMin(VarArrB) == 10);
```

Σημειώνεται μάλιστα, ότι από το να γράφουμε `VarArr[0] + VarArr[1] + VarArr[2]`, είναι πιο αποδοτικό να χρησιμοποιούμε την ισοδύναμη έκφραση `NsSum(VarArr,0,3)`. Το δεύτερο και το τρίτο όρισμα του `NsSum` είναι προαιρετικά και συμβολίζουν αντίστοιχα τη θέση του πρώτου στοιχείου του `VarArr` το οποίο θα μπει στο άθροισμα και τον αριθμό των μετέπειτα στοιχείων (μαζί με το πρώτο) που θα συνυπολογιστούν στο άθροισμα. Αν δεν υπάρχουν αυτά τα δύο ορίσματα, τότε όλος ο πίνακας μπαίνει στο άθροισμα.

Ο λόγος που το `NsSum` είναι η αποδοτικότερη από τις δύο εκφράσεις, έχει να κάνει με το γεγονός ότι για την πρώτη παράσταση *θα δημιουργηθεί μία ενδιάμεση μεταβλητή* που θα ισούται με `VarArr[0] + VarArr[1]` και σε αυτήν θα προστεθεί η `VarArr[2]`. Αυτό δεν θα γίνει για το `NsSum`.

Το `NsAbs` δίνει την απόλυτη τιμή. Τα `NsMin` και `NsMax`, δίνουν το ελάχιστο και το μέγιστο αντίστοιχα του πίνακα τον οποίο δέχονται σαν όρισμα.

6.2.1 Ο Περιορισμός Element

Αφιερώνουμε μία ξεχωριστή παράγραφο για την τελευταία έκφραση «$IntArr$`[` $Expression$ `]`», καθώς αφορά έναν ιδιαίτερο περιορισμό τον οποίο ονομάζουμε ***element***.[7] Για λόγους απλότητας θεωρούμε ότι η έκφραση $Expression$ είναι απλά μία περιορισμένη μεταβλητή $VarIndex$, η οποία χρησιμοποιείται

[7] Η ονομασία «element» προέρχεται από τον χώρο του λογικού προγραμματισμού.

σαν «δείκτης» στον πίνακα ακεραίων $IntArr$. Τονίζεται ότι ο $IntArr$ είναι πίνακας που περιέχει *ακέραιες* τιμές, αφού είναι τύπου `NsDeque<NsInt>`· *δεν* περιέχει περιορισμένες μεταβλητές γιατί τότε θα ήταν τύπου `NsIntVarArray`.

Για να κατανοήσουμε τη χρησιμότητα του περιορισμού, θα δούμε ένα παράδειγμα. Έστω ότι έχουμε έναν πίνακα `NsDeque<NsInt> grades` με τους βαθμούς οκτώ φοιτητών. Και έστω ότι το πεδίο τιμών της $VarIndex$ περιέχει όλες τις θέσεις του πίνακα, δηλαδή τις `[0..7]`. Αν επιθυμούμε το πεδίο τιμών μίας μεταβλητής $VarValue$ να περιέχει όλες τις βαθμολογίες, θέτουμε τον περιορισμό «$VarValue$ `== grades[`$VarIndex$`]`».

(Στην περίπτωση που θέσουμε έναν ακόμα περιορισμό, π.χ. «$VarValue$ `>= 9`», τότε το πεδίο τιμών της $VarIndex$ θα περιοριστεί έτσι ώστε να περιέχει μόνο τους αύξοντες αριθμούς των φοιτητών που βαθμολογήθηκαν με άριστα, δηλαδή με 9 ή 10.)[8]

6.3 Εκφράσεις Πινάκων

Τέλος, υπάρχει μια ειδική, ανεξάρτητη κατηγορία εκφράσεων, που αφορά ανάθεση σε πίνακα μεταβλητών τύπου `NsIntVarArray`. Περιλαμβάνει τις παρακάτω παραστάσεις, για τον περιορισμό Inverse (βλ. § 6.3.1).

- `NsInverse(` $VarArr$ `)`
- `NsInverse(` $VarArr$`,` $maxdom$ `)`

Με $maxdom$ συμβολίζουμε τον αριθμό των στοιχείων του αντίστροφου πίνακα που θέλουμε να φτιάξουμε. Αν δεν υπάρχει τιμή για αυτό το όρισμα, θεωρούμε ότι $maxdom = \max_{V \in VarArr}\{V.\max\}$. Σε κάθε περίπτωση, το $maxdom$ πρέπει να είναι μεγαλύτερο ή ίσο από αυτήν την τιμή. Π.χ.

```
NsIntVarArray  VarArrB = NsInverse(VarArrA);
NsIntVarArray  VarArrC;
VarArrC = NsInverse(VarArrA, 100);
```

6.3.1 Ο Περιορισμός Inverse

Ο περιορισμός *Inverse* (Αντιστροφή), εφαρμόζεται ανάμεσα σε δύο πίνακες από περιορισμένες μεταβλητές. Έστω ότι έχουμε έναν πίνακα Arr, του οποίου

[8] Ο περιορισμός element εκτός από τη μορφή «$VarValue$ `==` $IntArr$`[`$VarIndex$`]`» που είδαμε ότι μπορεί να πάρει, είναι δυνατόν να διατυπωθεί όπως και στον λογικό προγραμματισμό σαν «`NsElement(`$VarIndex$`,`$IntArr$`,`$VarValue$`)`».

οι τιμές των μεταβλητών είναι θετικές και επιθυμούμε ο «αντίστροφός» του να είναι ο $ArrInv$ και ακόμα, έστω ότι το D_x συμβολίζει το πεδίο τιμών μιας μεταβλητής x. Τότε θα ισχύει:

$$\forall v \in D_{ArrInv[i]}, \quad D_{Arr[v]} \ni i.$$

Αν δεν υπάρχει v, τέτοιο ώστε $i \in D_{Arr[v]}$, τότε το πεδίο της $ArrInv[i]$ θα περιέχει την ειδική τιμή -1 και μόνο.

Απλοποιώντας τους συμβολισμούς, μπορούμε να γράψουμε ότι πρέπει να ισχύει:

$$Arr[ArrInv[i]] = i \quad \text{και} \quad ArrInv[Arr[i]] = i.$$

Εξ ου και η ονομασία «Inverse». Βέβαια, αυτές οι σχέσεις θα είχαν νόημα, αν όλες μεταβλητές των δύο πινάκων ήταν δεσμευμένες και αν η μοναδική τιμή που θα περιείχε το πεδίο τιμών κάθε μεταβλητής, συμβολιζόταν με το ίδιο το όνομα της μεταβλητής. Ακόμα θα έπρεπε $\forall i$, $ArrInv[i] \neq -1$.

Χρησιμότητα του Περιορισμού. Ο περιορισμός μπορεί να χρησιμοποιηθεί σε δυϊκές (dual) μοντελοποιήσεις ένος προβλήματος. Π.χ. έστω ότι έχουμε ένα πλήθος από εργασίες για να ανατεθούν σε κάποια άτομα. Μία πιθανή μοντελοποίηση είναι να έχουμε μία μεταβλητή για κάθε εργασία με πεδίο το σύνολο των ατόμων. Μία άλλη είναι να έχουμε μία μεταβλητή για κάθε άτομο με πεδίο το σύνολο των εργασιών. Προφανώς το πρόβλημα έχει νόημα όταν υπάρχουν και κάποιοι περιορισμοί για το πώς πρέπει να γίνουν οι αναθέσεις. Πιθανώς, στη διατύπωση κάποιων περιορισμών να βολεύει η πρώτη μοντελοποίηση, ενώ για κάποιους άλλους να βολεύει η δεύτερη.

Σε αυτή την περίπτωση δίνεται η δυνατότητα από τον επιλυτή να χρησιμοποιούμε και τις δύο μοντελοποιήσεις. Οι μεταβλητές όμως των δύο μοντελοποιήσεων δεν είναι άσχετες. Κάπως θα πρέπει να πούμε κάτι σαν

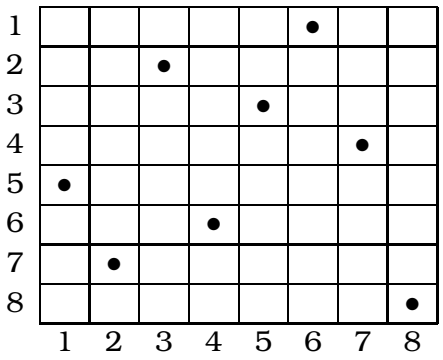
$$X[i] = j \iff Y[j] = i.$$

Αυτό γίνεται με έναν περιορισμό Inverse.

7 Παραδείγματα

7.1 Το Πρόβλημα των N Βασιλισσών

Σαν παράδειγμα θα διατυπώσουμε ένα πραγματικό πρόβλημα.



Σχήμα 1: 8 βασίλισσες που δεν απειλούνται

7.1.1 Ορισμός

Το πρόβλημα των N βασιλισσών συνίσταται στην τοποθέτηση N βασιλισσών σε μία $N \times N$ σκακιέρα, με τέτοιο τρόπο ώστε καμία να μην απειλείται. Με άλλα λόγια θέλουμε να τοποθετήσουμε N στοιχεία σε ένα πλέγμα $N \times N$, έτσι ώστε κανένα από αυτά να μην έχει κοινή γραμμή, στήλη, ή διαγώνιο με κάποιο άλλο. Στο Σχήμα 1 φαίνεται ένα παράδειγμα για $N = 8$.

Συνεπώς σε κάθε στήλη $0, 1, \ldots, N-1$ θα αντιστοιχεί και μία βασίλισσα. Μένει να δούμε σε ποια γραμμή θα τοποθετήσουμε την καθεμία. Οπότε οι άγνωστοι του προβλήματος είναι οι μεταβλητές X_i με $0 \leq X_i \leq N-1$, όπου X_i είναι η γραμμή στην οποία βρίσκεται η βασίλισσα της στήλης i.

Όσον αφορά τους περιορισμούς που ισχύουν, καταρχήν οι βασίλισσες πρέπει να μην απειλούνται στις γραμμές, δηλαδή

$$X_i \neq X_j, \quad \forall i \neq j \tag{1}$$

Επίσης δεν πρέπει να απειλούνται στις δύο διαγώνιους, οπότε

$$X_i + i \neq X_j + j \quad \text{και} \quad X_i - i \neq X_j - j, \quad \forall i \neq j \tag{2}$$

Το $X_i + i$ αντιστοιχεί στην πρώτη και το $X_i - i$ στη δεύτερη διαγώνιο της βασίλισσας της στήλης i.

7.1.2 Κώδικας

Στον κώδικα για τον επιλυτή, οι μεταβλητές X_i αναπαρίστανται με έναν πίνακα `Var` για τον οποίο θα απαιτήσουμε, σύμφωνα με την (1), να έχει διαφορετικά μεταξύ τους στοιχεία. Όσον αφορά την (2), φτιάχνουμε δύο ακόμα

πίνακες, τον `VarPlus` και τον `VarMinus`, με στοιχεία τα $X_i + i$ και $X_i - i$ αντίστοιχα. Και για αυτούς τους πίνακες, θα δηλώσουμε ότι ισχύει ο περιορισμός ότι τα στοιχεία που περιέχουν διαφέρουν μεταξύ τους.

```
int  N = 8;
NsProblemManager  pm;

NsIntVarArray  Var, VarPlus, VarMinus;
for (int i=0;  i<N;  ++i)   {
    Var.push_back( NsIntVar(pm, 0, N-1) );
    VarPlus.push_back(  Var[i] + i );
    VarMinus.push_back( Var[i] - i );
}
pm.add( NsAllDiff(Var) );
pm.add( NsAllDiff(VarPlus) );
pm.add( NsAllDiff(VarMinus) );

pm.addGoal( new NsgLabeling(Var) );
while (pm.nextSolution() != false)
    cout << "Solution: " << Var << "\n";
```

7.2 $SEND + MORE = MONEY$

Ένα άλλο παράδειγμα, αφορά ένα γνωστό πρόβλημα *κρυπτάριθμου* (cryptarithm). Σε αυτά τα προβλήματα, έχουμε κάποιες αριθμητικές σχέσεις μεταξύ λέξεων, όπως η $SEND + MORE = MONEY$. Κάθε γράμμα των λέξεων αναπαριστά ένα συγκεκριμένο ψηφίο (0 ως 9) και έτσι κάθε λέξη αντιπροσωπεύει έναν δεκαδικό αριθμό. Δύο διαφορετικά γράμματα δεν πρέπει να παριστάνουν το ίδιο ψηφίο. Π.χ. στη σχέση $SEND + MORE = MONEY$, όπου υπάρχει το E, θα βάλουμε ένα ψηφίο. Το ίδιο και για τα υπόλοιπα γράμματα, στα οποία όμως θα ανατεθεί διαφορετικό ψηφίο, από αυτό του E. Το ζητούμενο είναι μετά από τις όποιες αναθέσεις, να ισχύει η σχέση. Το πρόβλημα διατυπώνεται στον επιλυτή ως εξής:

```
NsProblemManager  pm;

NsIntVar  S(pm,1,9), E(pm,0,9), N(pm,0,9), D(pm,0,9),
          M(pm,1,9), O(pm,0,9), R(pm,0,9), Y(pm,0,9);

NsIntVar  send  =           1000*S + 100*E + 10*N + D;
```

```
NsIntVar  more  =           1000*M + 100*O + 10*R + E;
NsIntVar  money = 10000*M + 1000*O + 100*N + 10*E + Y;

pm.add( send + more == money );

NsIntVarArray  letters;
letters.push_back( S );
letters.push_back( E );
letters.push_back( N );
letters.push_back( D );
letters.push_back( M );
letters.push_back( O );
letters.push_back( R );
letters.push_back( Y );
pm.add( NsAllDiff(letters) );

pm.addGoal( new NsgLabeling(letters) );
if (pm.nextSolution() != false)   {
    cout << "    " << send.value() << "\n"
         << " +  " << more.value() << "\n"
         << " = " << money.value() << "\n";
}
```

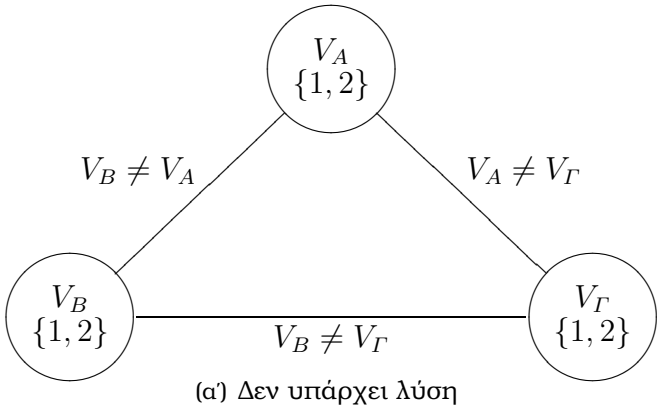
Το αποτέλεσμα που προκύπτει αν τρέξουμε το εκτελέσιμο για το παραπάνω πρόγραμμα, είναι το εξής:

```
   9567
 +  1085
 = 10652
```

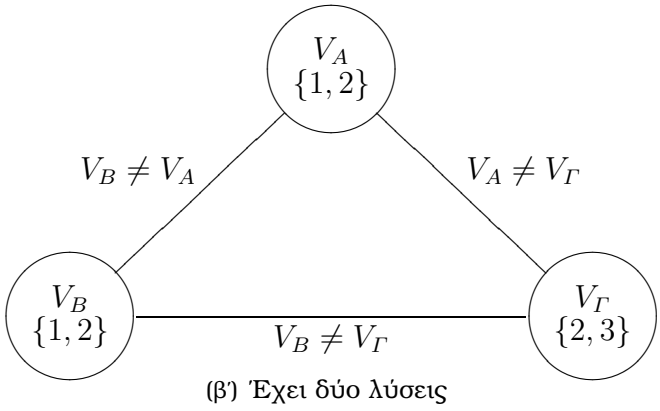
8 Αναζήτηση Λύσης μέσω Στόχων

8.1 Εισαγωγή

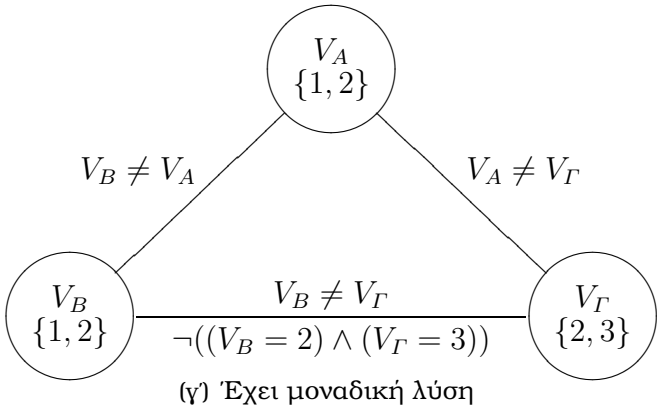
Ένα ζεύγος μεταβλητών (x, x') είναι συνεπές, αν για κάθε τιμή v του πεδίου τιμών της x, υπάρχει μία τιμή v' στο πεδίο τιμών της x' τέτοια ώστε να ικανοποιούνται όλοι οι περιορισμοί που συνδέουν τις δύο μεταβλητές. Όταν κάθε τέτοιο ζευγάρι μεταβλητών είναι συνεπές, λέμε ότι το δίκτυο περιορισμών χαρακτηρίζεται από *συνέπεια-ακμών* (arc-consistency). Η συνέπεια-ακμών δεν οδηγεί απαραίτητα σε μία λύση –αλλά αν δεν υπάρχει συνέπεια-ακμών,



(α') Δεν υπάρχει λύση



(β') Έχει δύο λύσεις



(γ') Έχει μοναδική λύση

Σχήμα 2: Τρεις γράφοι με συνέπεια ως προς τις ακμές

είμαστε σίγουροι ότι δεν υπάρχει λύση–, εκτός εάν τη συνδυάσουμε με μία μέθοδο αναζήτησης. Η χρησιμότητά της έχει να κάνει με τη μείωση του χώρου αναζήτησης τον οποίο μία συνηθισμένη μέθοδος αναζήτησης με την οποία συνδυάζεται –όπως η πρώτα-κατά-βάθος αναζήτηση (Depth First Search – DFS), η αναζήτηση περιορισμένων ασυμφωνιών (Limited Discrepancy Search – LDS) κ.ά.– πρέπει να εξερευνήσει.

Όπως είναι γνωστό λοιπόν, στα περισσότερα προβλήματα δεν αρκεί μόνο η επιβολή συνέπειας-ακμών για να βρεθεί μία λύση (βλ. και Σχήμα 2). Από ένα σημείο και μετά, πρέπει να αρχίσουμε την αναζήτηση, επαναλαμβάνοντας την ανάθεση τιμών σε μεταβλητές και ελέγχοντας κάθε φορά –π.χ. μετά από κάθε ανάθεση– αν το δίκτυο περιορισμών χαρακτηρίζεται από συνέπεια-ακμών, σύμφωνα και με την πρακτική της *διατήρησης συνέπειας ακμών* (maintaining arc consistency – MAC).[9] Αν μία ανάθεση τιμής προκαλέσει ασυνέπεια, τότε θα πρέπει να ακυρωθεί και να επιλεγεί μία άλλη τιμή.

Για να διευκολυνθεί, ή, καλύτερα, για να καθοδηγηθεί η αναζήτηση, έχει υλοποιηθεί στον επιλυτή ένας *μηχανισμός στόχων*. Ο προγραμματιστής που χρησιμοποιεί τον επιλυτή μπορεί να ορίσει τους δικούς του στόχους, ή να εκμεταλλευτεί τους ενσωματωμένους. Συνήθως, ένας στόχος προκαλεί την ανάθεση τιμής σε μία περιορισμένη μεταβλητή, ή την αφαίρεση τιμής από το πεδίο της. Αν στην πορεία της αναζήτησης καταλήξουμε σε αδιέξοδο, οι στόχοι που οδήγησαν σε αυτό ακυρώνονται αυτόματα από τον επιλυτή και το δίκτυο περιορισμών με τις μεταβλητές που το συνθέτουν επανέρχεται στην κατάσταση που βρισκόταν πριν εκτελεστούν οι στόχοι.

Γενικά, ένας στόχος μπορεί να αναθέτει ή να αφαιρεί τιμές σε μία ή περισσότερες μεταβλητές, ή να χρησιμοποιείται για την επιλογή μεταβλητής για να της ανατεθεί τιμή και, κατά αυτόν τον τρόπο, να καθορίζει τη μέθοδο αναζήτησης. Ένας στόχος, κατά τον τερματισμό της εκτέλεσής του, μπορεί προαιρετικά να «γεννήσει» έναν άλλον στόχο· αυτή η δυνατότητα μπορεί να προσδώσει χαρακτηριστικά αναδρομής στη διατύπωση των στόχων. Τέλος, δεν θα πρέπει να παραλείψουμε τους *μετα-στόχους* AND και OR. Τους χαρακτηρίσαμε σαν «μετα-στόχους», γιατί είναι στόχοι που απλά υπάρχουν για να εξυπηρετούν, ο καθένας τους, δύο άλλους στόχους, τους οποίους λέμε *υπο-στόχους*. Για να πετύχει ο στόχος-AND, θα πρέπει να ικανοποιηθούν και οι δύο υπο-στόχοι του, ενώ για να πετύχει ο OR, αρκεί να ικανοποιηθεί ένας

[9] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. 2nd Int. Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 125–129, 1994.

από τους υπο-στόχους του.

Αξίζει να σημειωθεί ότι οι στόχοι-OR, είναι γνωστοί και σαν *σημεία επιλογής* (choice points). Πράγματι, είναι σημεία στα οποία έχουμε δύο εναλλακτικές επιλογές, δηλαδή σημεία στα οποία διακλαδώνεται το δένδρο αναζήτησης. Ο στόχος-OR υπαγορεύει να επιλέξουμε ένα υπο-στόχο του και αν τελικά δεν πετύχει αυτός, να αναιρεθούν οι όποιες (αλυσιδωτές) αλλαγές που αυτός επέφερε στα πεδία των μεταβλητών και να δοκιμάσουμε τον άλλον υπο-στόχο. Αν φυσικά και αυτός αποτύχει, τότε και ο στόχος-OR αποτυγχάνει.

8.2 Αντικειμενοστρεφής Μοντελοποίηση

Η αφηρημένη βασική κλάση για έναν στόχο στον NAXOS SOLVER, ορίζεται ως εξής:

```
class  NsGoal  {
    public:
        virtual bool  isGoalAND (void)  const;
        virtual bool  isGoalOR  (void)  const;
        virtual NsGoal*  getFirstSubGoal  (void)  const;
        virtual NsGoal*  getSecondSubGoal (void)  const;

        virtual NsGoal*  GOAL (void) = 0;
};
```

Από την κλάση αυτή, παράγονται οι κλάσεις των μετα-στόχων `NsgAND` και `NsgOR`, των οποίων η μέθοδος-κατασκευής παίρνει δύο ορίσματα (τύπου «`NsGoal*`»), που δεν είναι τίποτα άλλο παρά οι δύο υπο-στόχοι τους. Όλες οι μέθοδοι της `NsGoal`, εκτός από την `GOAL()`, αφορούν τους μετα-στόχους. Ο προγραμματιστής που θα ορίσει δικούς του στόχους, χρειάζεται να ασχοληθεί μόνο με την `GOAL()`.

Κάθε στόχος που ορίζεται από έναν προγραμματιστή που χρησιμοποιεί τον επιλυτή, θα είναι μία κλάση η οποία θα παράγεται, έστω και έμμεσα, από τη βασική κλάση `NsGoal`. Συνεπώς θα πρέπει να ορίζεται στον κάθε στόχο, η μέθοδος `GOAL()`. Η κλάση του στόχου μπορεί ασφαλώς να περιέχει και οποιεσδήποτε άλλες μεθόδους –πλην αυτών της βασική κλάσης, προς αποφυγή συγχύσεων με τους μετα-στόχους.

Η `GOAL()` είναι μία κρίσιμη μέθοδος, καθώς εκτελείται κάθε φορά που ο επιλυτής προσπαθεί να ικανοποιήσει έναν στόχο. Η μέθοδος αυτή επιστρέφει

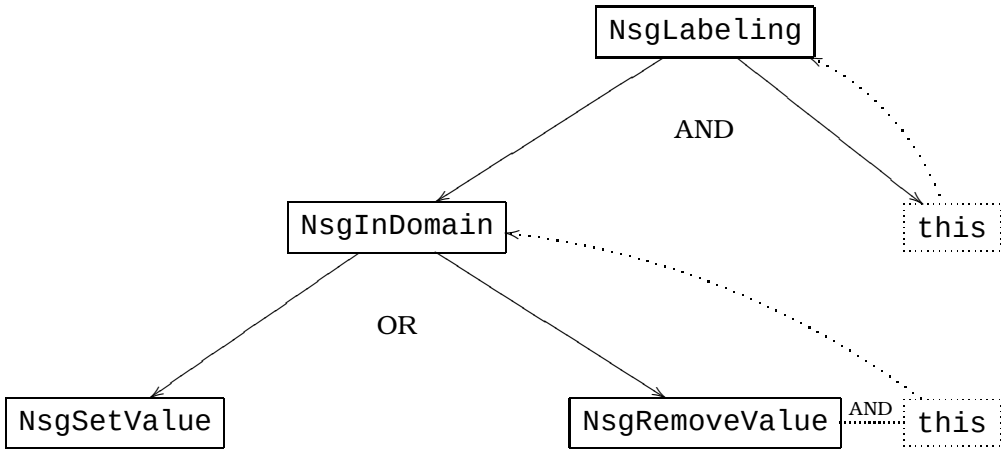
έναν δείκτη σε `NsGoal`, δηλαδή επιστρέφει τον επόμενο στόχο προς ικανοποίηση. Αν ο δείκτης είναι το `0`, αυτό σημαίνει ότι ο τρέχων στόχος πέτυχε –και έτσι δεν απαιτείται η δημιουργία κάποιου άλλου στόχου.

Ας δούμε ένα παράδειγμα στόχων, οι οποίοι είναι μάλιστα ενσωματωμένοι στον επιλυτή, καθώς χρησιμοποιούνται ευρέως. Πρόκειται για τους στόχους της πρώτα-κατά-βάθος αναζήτησης (depth-first-search – DFS).

```
class NsgInDomain : public NsGoal  {
  private:
    NsIntVar&  Var;

  public:
    NsgInDomain (NsIntVar& Var_init)
      : Var(Var_init)    {    }

    NsGoal*  GOAL (void)
    {
       if (Var.isBound())
           return  0;
       NsInt  value = Var.min();
       return ( new NsgOR( new NsgSetValue(Var,value),
                      new NsgAND( new NsgRemoveValue(Var,value),
                                  new NsgInDomain(*this) ) ) );
    }
};


class NsgLabeling : public NsGoal  {
  private:
    NsIntVarArray&  VarArr;

  public:
    NsgLabeling (NsIntVarArray& VarArr_init)
      : VarArr(VarArr_init)    {    }

    NsGoal*  GOAL (void)
    {
        int  index = -1;
        NsUInt  minDom = NsUPLUS_INF;
        for (NsIndex i = 0;   i < VarArr.size();   ++i)   {
            if ( !VarArr[i].isBound()
                     &&  VarArr[i].size() < minDom )
            {
                minDom = VarArr[i].size();
                index = i;
```

```
            }
        }
        if (index == -1)
            return  0;
        return  ( new NsgAND( new NsgInDomain(VarArr[index]),
                              new NsgLabeling(*this) ) );
    }
};
```

Παρατηρούμε ότι στην επιστρεφόμενη τιμή της `GOAL()` (όταν αυτή δεν είναι `0`) και στις μεθόδους κατασκευής των μετα-στόχων `NsgAND` και `NsgOR`, χρησιμοποιούμε τον τελεστή `new`. Αυτό είναι απαραίτητο να γίνεται κάθε φορά, για να φτιάξουμε έναν δείκτη σε στόχο. Ο επιλυτής αναλαμβάνει στην πρώτη ευκαιρία να σβήσει έναν άχρηστο στόχο με τον τελεστή `delete`· για αυτό, *όλοι οι στόχοι που φτιάχνουμε, θα πρέπει να έχουν δημιουργηθεί με* `new` *και να* μην *γίνονται* `delete` *από εμάς*.



Σχήμα 3: Συνδυασμός στόχων που αποτελούν τον `NsgLabeling`

Όσον αφορά την πρακτική σημασία του παραδείγματος, όταν ζητήσουμε από τον επιλυτή να ικανοποιηθεί ο στόχος `NsgLabeling(VarArr)`, τότε αναμένουμε να ανατεθούν τιμές σε όλες τις μεταβλητές του πίνακα `VarArr`. Έτσι, ο στόχος `NsgLabeling`, στη μέθοδο `GOAL()` πάντα, διαλέγει μία μεταβλητή (συγκεκριμένα εκείνη με το μικρότερο πεδίο, σύμφωνα με το ευριστικό first-fail). Έπειτα προστάζει να δοθεί τιμή στη μεταβλητή (μέσω του στόχου

`NsgInDomain` ο οποίος αναθέτει σε μία μεταβλητή την ελάχιστη τιμή του πεδίου της) *και* να ικανοποιηθεί ο στόχος `this`. Αυτός ο στόχος –που παραπέμπει και σε ένα είδος «αναδρομής»– δημιουργεί ένα άλλο στιγμιότυπο της `NsgLabeling`, πανομοιότυπο με το τρέχον στιγμιότυπο. Με το `this`, κατ' ουσίαν προτρέπουμε τον επιλυτή να αναθέσει τιμή και στις υπόλοιπες μεταβλητές του πίνακα `VarArr`. Όταν η `GOAL()` επιστρέψει `0`, θα έχουμε τελειώσει επιτυχώς (Σχήμα 3).

Ενώ ο `NsgLabeling` επιλέγει μία μεταβλητή για να της ανατεθεί τιμή, ο `NsgInDomain` επιλέγει την τιμή που θα ανατεθεί. Πιο συγκεκριμένα, επιλέγει πάντα την ελάχιστη τιμή του πεδίου της μεταβλητής. Έπειτα καλεί τον ενσωματωμένο στόχο `NsgSetValue` που απλά αναθέτει την τιμή στη μεταβλητή. Αν στη συνέχεια βρεθεί ότι η τιμή αυτή δεν ανήκει σε κάποια λύση, τότε αφαιρείται από το πεδίο της μεταβλητής με τον στόχο `NsgRemoveValue` και στη συνέχεια θα ανατεθεί μία άλλη τιμή (στόχος «`NsgInDomain(*this)`»).

Συνήθως, για την αντιμετώπιση δύσκολων και μεγάλων προβλημάτων, επιβάλλεται να ορίσουμε τους δικούς μας στόχους, σαν τους `NsgLabeling` και `NsgInDomain`. Ο σκοπός είναι να γίνει η αναζήτηση πιο αποδοτική, μέσω πιο εύστοχων επιλογών, που προκύπτουν από ευριστικές συναρτήσεις.

Ευρετήριο