ΕΘΝΙΚΟ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

Καθηγητής : ***Κουμπαράκης Μανόλης*** **Ημ/νία παράδοσης:** ***11/01/2011***

Ονομ/μο φοιτητή : ***Μπεγέτης Νικόλαος***
Α.Μ.: ***1115200700281***

Δεύτερο πακέτο ασκήσεων

Σχόλια και παρουσίαση αποτελεσμάτων για Πρόβλημα 1:

Όπως ζητείται κατά την εκκίνηση του προγράμματος ορίζουμε ένα αντικείμενο *puzzle* το οποίο κατά την αρχικοποίηση του λαμβάνει το όνομα του puzzle *(puzzle0-puzzle6)* στο οποίο θέλουμε να βρεθεί λύση ορίζοντάς το πρόβλημα αναζήτησης λύσης ως πρόβλημα ικανοποίησης περιορισμών, δηλαδή για να συμπληρωθούν τα κενά κουτάκια του *kakuro-puzzle* θα πρέπει να γίνουν κάποιοι έλεγχοι συνέπειας που υπακούουν σε κάποιους περιορισμούς που δίνονται από τη συνάρτηση *has_conflict*. Πριν όμως φτάσουμε εκεί, στο κυρίως πρόγραμμά μας η είσοδος του puzzle γίνεται κάνοντας *import* το αρχείο *input_puzzles.py* το οποίο μέσα περιέχει *7 kakuro puzzles (4 που δίνονταν από την εκφώνηση και άλλα 3 τα οποία επιλέξαμε εμείς).* Αυτά είναι ορισμένα ως λίστες οι οποίες περιέχουν υπολίστες *(γραμμές του puzzle)* που μπορούν να έχουν σαν στοιχεία τους τρία είδη τετραγώνων *(τετράγωνα ανά στήλη του puzzle)* (κενά μαύρα τετράγωνα’*’, κενά λευκά τετράγωνα’_’ και μαύρα τετράγωνα που περιέχουν αθροίσματα’[‘’, 4], όπου ‘’=τίποτα και τέσσερα το άθροισμα που πρέπει να έχουν όλα τα συνεχόμενα προς τα δεξιά λευκά τετράγωνα’).

Κατά την αρχικοποίηση του αντικειμένου *puzzle* καλείται η συνάρτηση *finput* η οποία ψάχνει να βρει τους οριζόντιους και κάθετους γείτονες κάθε λευκού-κενού τετραγώνου γιατί χρειάζονται για τους περιορισμούς του kakuro. Αυτό το κάνει επιστρέφοντας ένα *dictionary* όπου έχει σαν κλειδιά *(keys)* τις συντεταγμένες σε μορφή *tuples* των λευκών τετραγώνων και τιμές μία *list* ανά κλειδί που περιέχει τις συντεταγμένες όλων των γειτόνων του κλειδιού. Επίσης, η συνάρτηση επιστρέφει μία *list* με αθροίσματα, τα οποία έχουν μορφή *λίστας* με *2* στοιχεία, όπου το πρώτο στοιχείο τους είναι *το ένα από τα δύο αθροίσματα* που μπορεί να περιέχουν τα μαύρα τετράγωνα και το άλλο στοιχείο του είναι μία ακόμα *λίστα* που περιλαμβάνει τις συντεταγμένες όλων των τετραγώνων που το περιεχόμενό τους πρέπει να αθροίζει στο άθροισμα αυτό. Τέλος η συνάρτηση εκτυπώνει το άδειο puzzle.

Έπειτα αφού γίνει η αρχικοποίηση του βασικού αντικειμένου *puzzle* ως πρόβλημα *CSP* ο χρήστης καλείται να επιλέξει ανάμεσα στους 4 αλγορίθμους προβλημάτων ικανοποίησης περιορισμών που δίνονται από την εκφώνηση *(BT, BT+MRV, FC, FC+MRV)* για να βρει λύση στο puzzle. Αφού επιλέξει, ξεκινάει *ο αλγόριθμος οπισθοδρόμησης* να χρονομετρείται και σταματάει μόλις θα έχει βρεθεί λύση. Έπειτα εκτυπώνονται τα αποτελέσματα:

- *Εκτύπωση λύσης του puzzle με στοιχισμένο τρόπο αντικαθιστώντας τα κενά'_' με τα αντίστοιχα νούμερα (1-9),*
- *Συνολικές συγκρούσεις περιορισμών για έλεγχο συνέπειας*
- *Συνολικές αναθέσεις τιμών στις μεταβλητές του CSP*
- *συνολικός χρόνος εκτέλεσης αλγορίθμου.*

Στην κλάση *Kakuro_puzzle* γίνεται *override* η συνάρτηση *has_conflict* του αρχείου *csp.py* την οποία έχουμε ξανα-ορίζει εμείς στο αρχείο *problem2_1.py* .

Η συνάρτηση *has_conflict* ουσιαστικά είναι η συνάρτηση που υλοποιεί τους ελέγχους συνέπειας με βάση του περιορισμούς, που στην περίπτωσή μας είναι *α)* όλα τα γειτονικά λευκά τετράγωνα ανά στήλη ή ανά γραμμή να αθροίζουν στον αριθμό που βρίσκεται στο επάνω ή αντίστοιχα αριστερό από αυτά μαύρο τετράγωνο και *β)* κάθε λευκό τετράγωνο να ελέγχει να μην έχει την ίδια τιμή με τους οριζόντιους ή κάθετους γείτονές του. Σε περίπτωση που ικανοποιούνται και οι *2* συνθήκες δεν γίνεται σύγκρουση και επιστρέφεται *false*, διαφορετικά γίνεται σύγκρουση και επιστρέφεται *true*.

Συνεπώς έπειτα από όλα τα παραπάνω μπορούμε να συμπεράνουμε ότι το puzzle μας όντως είναι μοντελοποιημένο ως *πρόβλημα ικανοποίησης περιορισμών* αφού μπορούμε να το διατυπώσουμε με τα εξής τρία στοιχεία:

Μεταβλητές: Ως μεταβλητές ορίζουμε όλα τα λευκά-κενά τετράγωνα του puzzle

Πεδίο τιμών: Το πεδίο τιμών κάθε μεταβλητής είναι τα νούμερα *1-9*

Περιορισμοί: Όπως αναφέραμε προηγουμένως οι περιορισμοί μεταξύ των λευκών γειτονικών τετραγώνων είναι: *α)* όλα τα γειτονικά λευκά τετράγωνα ανά στήλη ή ανά γραμμή να αθροίζουν στον αριθμό που βρίσκεται στο επάνω ή αντίστοιχα αριστερό από αυτά μαύρο τετράγωνο και *β)* κάθε λευκό τετράγωνο να έχει μοναδική τιμή ανάμεσα σε όλους τους οριζόντιους και κάθετους γείτονές του.

Όπως ξαναγράψαμε προηγουμένως χρησιμοποιήθηκαν τέσσερις αλγόριθμοι προβλημάτων ικανοποίησης περιορισμών (BT, BT+MRV, FC, FC+MRV) για να βρεθεί λύση στο puzzle. Λίγα λόγια για αυτούς πριν περάσουμε στις εκτυπώσεις και το σχολιασμό των αποτελεσμάτων:

- ***BT :*** Είναι αλγόριθμος χωρίς πληροφόρηση και γι' αυτό δεν αναμένουμε να είναι αποτελεσματικός για μεγάλα προβλήματα. Βασίζεται στην αναδρομική αναζήτηση πρώτα σε βάθος επιλέγοντας τιμές μόνο για μία μεταβλητή τη φορά και υπαναχωρεί όταν μία μεταβλητή δεν έχει άλλες νόμιμες τιμές που μπορούν να της ανατεθούν.
- ***BT+MRV :*** Είναι ο ίδιος αλγόριθμος *BT* που όμως αυτή τη φορά δεν επιλέγει απλώς την επόμενη μεταβλητή στην οποία δεν έχει ανατεθεί τιμή με τη σειρά που προκύπτει από τη λίστα των μεταβλητών, αλλά χρησιμοποιεί τον ευρεστικό μηχανισμό των ελάχιστων απομενουσών τιμών(*MRV*) μειώνοντας έτσι τον παράγοντα διακλάδωσης των μελλοντικών επιλογών με την εκλογή

της μεταβλητής που ενέχεται στο μεγαλύτερο αριθμό περιορισμών ως προς τις άλλες μεταβλητές στις οποίες δεν έχει ανατεθεί τιμή.

- ***FC :*** Ο αλγόριθμος οπισθοδρόμησης χρησιμοποιεί την *FC* ως έναν τρόπο να αξιοποιεί καλύτερα τους περιορισμούς κατά την αναζήτηση. Αυτό γίνεται με τον εξής τρόπο: Όποτε ανατίθεται τιμή σε μία μεταβλητή *X*, η διαδικασία του πρώιμου ελέγχου εξετάζει κάθε μεταβλητή *Y* που δεν της έχει ανατεθεί τιμή η οποία συνδέεται με την *X* με έναν περιορισμό, και διαγράφει από το πεδία της *Y* οποιαδήποτε τιμή που είναι συνεπής με την τιμή που επιλέχθηκε για την *X.*
- ***FC+MRV :*** Ο αλγόριθμος FC+MRV συνδυάζει τον πρώιμο έλεγχο που περιγράψαμε ακριβώς από πάνω μαζί με την ευρετική συνάρτηση ελάχιστων απομενουσών τιμών με πολύ αποδοτικό τρόπο και γι' αυτό το λόγο θεωρείται ως ο αποδοτικότερος μεταξύ αυτών των τεσσάρων.

Ακολουθούν 4 εκτυπώσεις για το πλέγμα του puzzle0 που μας δόθηκε στην εκφώνηση με καθέναν αλγόριθμο και μία για το πλέγμα του puzzle6 Έπειτα ακολουθούν συγκρίσεις των αλγορίθμων στα 7 puzzles με βάση τις συγκρούσεις, τις αναθέσεις τιμών και τον χρόνο εκτέλεσης των αλγορίθμων:

Python Shell

File Edit Shell Debug Options Windows Help

```
32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Type the kakuro puzzle name from the input_puzzles.py that you want to find a so
lution(eg. type: puzzle0,...,puzzle6) : puzzle0
==========================================
||  *    *        *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  _        _       ||
||['', 10] _     _       _        _       ||
||['', 3] _      _       *        *       ||
==========================================
Switch the algorithm you want to use for finding a solution :
1. BT (simple backtracking search)
2. BT+MRV (backtracking search with the heurestic mechanism MRV)
3. FC (forward checking search)
4. FC+MRV (forward checking search with the heurestic mechanism MRV)

Type : 1
Simple backtracking search(BT) is selected


Puzzle solution:
==========================================
||  *    *        *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  2        1       ||
||['', 10] 3     1       4        2       ||
||['', 3] 1      2       *        *       ||
==========================================

Total puzzle conflicts:  104
Total assignments:  18
Execution time : 0.005 seconds
>>> |
```

Ln: 33 Col: 4

Python Shell

File Edit Shell Debug Options Windows Help

```
Total puzzle conflicts:  104
Total assignments:  18
Execution time : 0.005 seconds
>>> ================================ RESTART ================================
>>>
Type the kakuro puzzle name from the input_puzzles.py that you want to find a so
lution(eg. type: puzzle0,...,puzzle6) : puzzle0
==========================================
||  *    *        *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  _        _       ||
||['', 10] _     _       _        _       ||
||['', 3] _      _       *        *       ||
==========================================
Switch the algorithm you want to use for finding a solution :
1. BT (simple backtracking search)
2. BT+MRV (backtracking search with the heurestic mechanism MRV)
3. FC (forward checking search)
4. FC+MRV (forward checking search with the heurestic mechanism MRV)

Type : 2
Backtracking search with the heurestic mechanism MRV(BT+MRV) is selected


Puzzle solution:
==========================================
||  *    *        *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  2        1       ||
||['', 10] 3     1       4        2       ||
||['', 3] 1      2       *        *       ||
==========================================

Total puzzle conflicts:  104
Total assignments:  18
Execution time : 0.003 seconds
>>> |
```

Ln: 63 Col: 4

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help

Total puzzle conflicts:  104
Total assignments:  18
Execution time : 0.003 seconds
>>> ================================ RESTART ================================
>>>
Type the kakuro puzzle name from the input_puzzles.py that you want to find a so
lution(eg. type: puzzle0,...,puzzle6) : puzzle0
==========================================
||  *   *       *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  _       _       ||
||['', 10] _    _       _       _       ||
||['', 3] _     _       *       *       ||
==========================================
Switch the algorithm you want to use for finding a solution :
1. BT (simple backtracking search)
2. BT+MRV (backtracking search with the heurestic mechanism MRV)
3. FC (forward checking search)
4. FC+MRV (forward checking search with the heurestic mechanism MRV)

Type : 3
Forward checking search(FC) is selected


Puzzle solution:
==========================================
||  *   *       *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  2       1       ||
||['', 10] 3    1       4       2       ||
||['', 3] 1     2       *       *       ||
==========================================

Total puzzle conflicts:  177
Total assignments:  15
Execution time : 0.004 seconds
>>> |
                                                                   Ln: 93 Col: 4
```

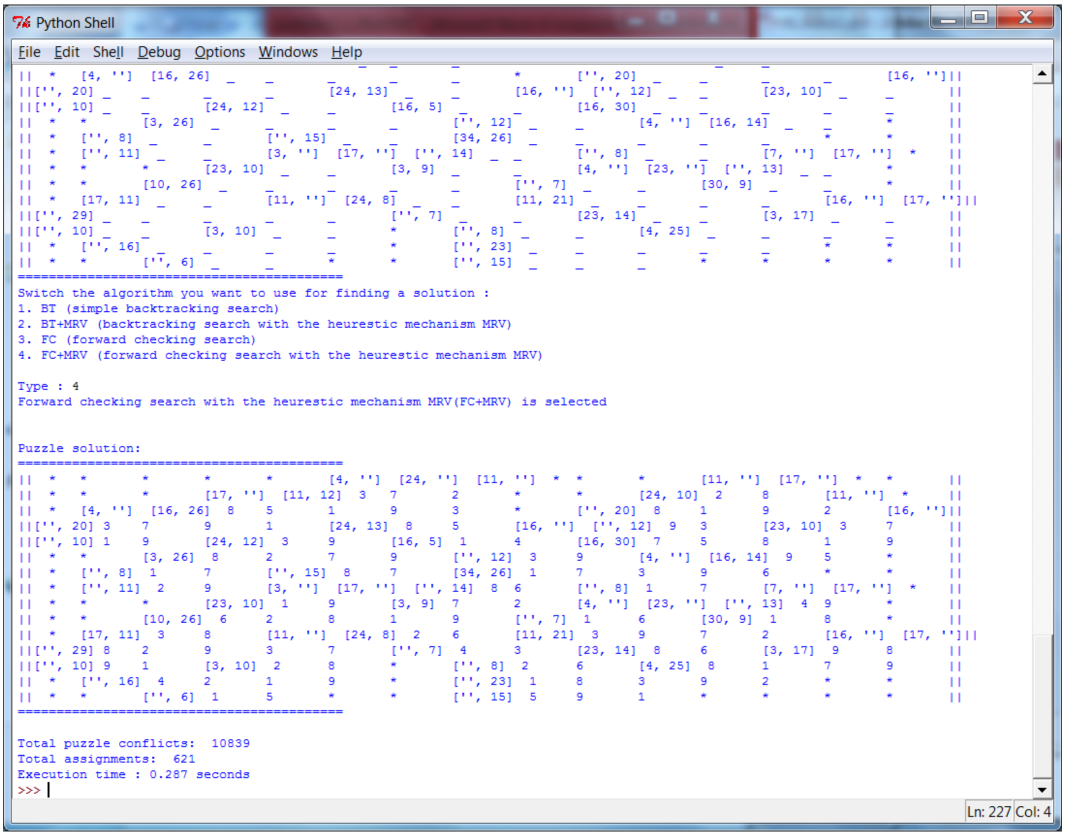
```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help

Total puzzle conflicts:  177
Total assignments:  15
Execution time : 0.004 seconds
>>> ================================ RESTART ================================
>>>
Type the kakuro puzzle name from the input_puzzles.py that you want to find a so
lution(eg. type: puzzle0,...,puzzle6) : puzzle0
==========================================
||  *   *       *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  _       _       ||
||['', 10] _    _       _       _       ||
||['', 3] _     _       *       *       ||
==========================================
Switch the algorithm you want to use for finding a solution :
1. BT (simple backtracking search)
2. BT+MRV (backtracking search with the heurestic mechanism MRV)
3. FC (forward checking search)
4. FC+MRV (forward checking search with the heurestic mechanism MRV)

Type : 4
Forward checking search with the heurestic mechanism MRV(FC+MRV) is selected


Puzzle solution:
==========================================
||  *   *       *       [6, '']  [3, '']||
||  *   [4, '']  [3, 3]  2       1       ||
||['', 10] 3    1       4       2       ||
||['', 3] 1     2       *       *       ||
==========================================

Total puzzle conflicts:  133
Total assignments:  11
Execution time : 0.003 seconds
>>> |
                                                                   Ln: 123 Col: 4
```



Όπως φαίνεται από τις παραπάνω εκτυπώσεις για το πρώτο πλέγμα *(puzzle0)* και οι τέσσερις αλγόριθμοι που χρησιμοποιήσαμε κατάφεραν και βρήκαν λύση που να ικανοποιεί τους περιορισμούς όπως ορίζονται στους κανόνες.

Εδώ αυτό που αξίζει να προσέξουμε είναι ότι για μικρά και εύκολα puzzles όλοι οι αλγόριθμοι κυμαίνονται περίπου στους ίδιους χρόνους εύρεσης λύσης. Επίσης μπορούμε να παρατηρήσουμε ότι με βάση τις συγκρίσεις που έγιναν για την εύρεση λύσης οι *BT* και *BT+MVR* εκτός του ότι εκτελούν τον ίδιο αριθμό συγκρίσεων έχουν και τον ελάχιστο αριθμό συγκρίσεων σε σχέση με τους άλλους δύο αλγορίθμους, με πλεονασμό όμως αναθέσεων μιας και οι δύο έκαναν τουλάχιστον 3 αναθέσεις περισσότερες σε μεταβλητές από ότι ο *FC* και ο *FC+MVR.*

Παρακάτω παρατίθενται πίνακες και σχολιασμός με μετρικές: *1)* τις συγκρούσεις, *2)* τις αναθέσεις τιμών και *3)* τους χρόνους εκτελέσεων των αλγορίθμων:

Πίνακας συνολικών συγκρούσεων για έλεγχο συνέπειας

Kakuro_Puzzles	*BT*	*BT+MRV*	*FC*	*FC+MRV*
Puzzle0	*104*	*104*	*177*	*133*
Puzzle1	*371*	*371*	*524*	*381*
Puzzle2	*9.561.523*	*9.561.523*	*38.943*	*946*
Puzzle3	-	-	-	*1.541.414*
Puzzle4	-	-	-	*289*
Puzzle5	-	-	-	*14549*
Puzzle6	-	-	-	*10839*

Πίνακας συνολικών αναθέσεων τιμών

Kakuro_Puzzles	*BT*	*BT+MRV*	*FC*	*FC+MRV*
Puzzle0	*18*	*18*	*15*	*11*
Puzzle1	*46*	*46*	*38*	*28*
Puzzle2	*1.062.402*	*1.062.402*	*544.237*	*13.155*
Puzzle3	*-*	*-*	*-*	*79782*
Puzzle4	*-*	*-*	*-*	*4451*
Puzzle5	*-*	*-*	*-*	*457*
Puzzle6	*-*	*-*	*-*	*621*

Πίνακας συνολικών χρόνων εκτέλεσης αλγορίθμων

Kakuro_Puzzles	*BT*	*BT+MRV*	*FC*	*FC+MRV*
Puzzle0	*0.005 sec*	*0.003 sec*	*0.004 sec*	*0.003 sec*
Puzzle1	*0.004 sec*	*0.010 sec*	*0.011 sec*	*0.009 sec*
Puzzle2	*90.094 sec*	*86.671 sec*	*5.480 sec*	*0.133 sec*
Puzzle3	*>5 min*	*>5 min*	*>5 min*	*20.599 sec*
Puzzle4	*>5 min*	*>5 min*	*>5 min*	*0.077 sec*
Puzzle5	*>5 min*	*>5 min*	*>5 min*	*0.243 sec*
Puzzle6	*>5 min*	*>5 min*	*>5 min*	*0.398 sec*

Οι παραπάνω μετρικές αποτελούν αξιόπιστα μέτρα σύγκρισης των αλγορίθμων γιατί το καθένα από αυτά στοχεύει σε κομμάτια των αλγορίθμων που σε άλλους έχουν υλοποιηθεί ώστε να βρίσκεται λύση με πιο μεγάλη σιγουριά μεν αλλά κάνοντας περισσότερους ελέγχους π.χ στον αλγόριθμο *FC* παρατηρούμε ότι στα πρώτα puzzles οι έλεγχοι ήταν σχεδόν οι διπλάσιοι από τους αντίστοιχους σε *BT* και *BT+MVR* αλλά εξαιτίας αυτών των ελέγχων οι αναθέσεις ήταν λιγότερες και όσο ανέβαινει η δυσκολία του *puzzle* το χάσμα των αναθέσεων διευρύνεται εκθετικά γιατί κατά πάσα πιθανότητα εκτελεί συνεχώς τις ίδιες οπισθοχωρήσεις. Ο λόγος αυτός φυσικά έχει αντίκτυπο και στον χρόνο εκτέλεσης. Συνεπώς τα μέτρα σύγκρισης είναι ικανοποιητικά.

Με βάση λοιπόν τα παραπάνω πινακάκια και τους ορισμούς που δώσαμε αξίζει να παρατηρήσουμε ότι οι ορισμοί αντιπροσωπεύουν τα αποτελέσματα που προέκυψαν. Συνεπώς όπως περιμέναμε τα μικρής δυσκολίας puzzle όλοι οι αλγόριθμοι κατάφεραν να τα λύσουν σε θεμιτό χρόνο και μάλιστα για τα πολύ εύκολα οι αλγόριθμοι *BT*, *BT+MRV* σε κάποιες συγκρίσεις τα πήγαν καλύτερα από τους άλλους δύο αλγορίθμους.

Είναι όμως ξεκάθαρο ότι ο αλγόριθμος *FC+MRV* εξαιτίας του πρώιμου ελέγχου που κάνει και γλυτώνει πολλές πιθανές οπισθοχωρήσεις και εξαιτίας της έξυπνης τακτικής με την οποία επιλέγει την επόμενη μεταβλητή προς ανάθεση τιμής είναι ο πιο αποδοτικός από όλους ακόμα και με μεγάλη διαφορά από τον *FC*. Αυτό φαίνεται στα αποτελέσματα αφού ο *FC+MVR* είναι ο μοναδικός αλγόριθμος που κατάφερε να βρει λύση και σύντομα μάλιστα σε όλα τα *puzzle*. Οι υπόλοιποι κατάφεραν να φτάσουν μέχρι τα puzzles μεσαίας δυσκολίας μόνο και σε χρόνο καθόλου αποδοτικό.