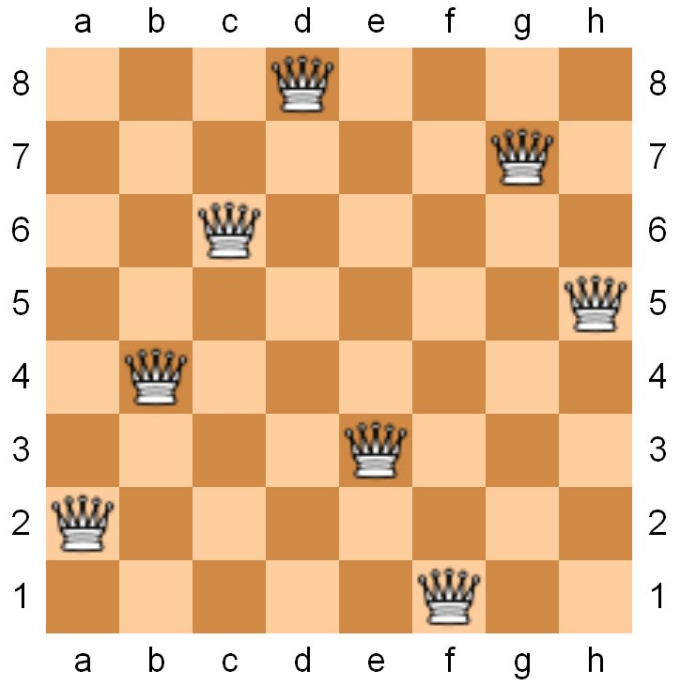
Τεχνητη Νοημοσυνη

Πρόβλημα n-Queens Αναφορά

Ομάδα:	**ΑΜ:**
Σωτήρης Δημητράς	2013030123
Νεκτάριος Σφυρής	2013030058
Τάσος Καραγεωργιάδης	2013030135



1 Μέρος Α

Σε αυτό το μέρος έπρεπε αρχικά να βρούμε το μαθηματικό μοντέλο που περιγράφει το πρόβλημα, έπειτα με βάση αυτό, να αναπτύξουμε κώδικα ο οποίος θα μας επιστρέφει τον αριθμό των βασιλισσών που απειλούνται σε οποιαδήποτε κατάσταση του προβλήματος. Τέλος ορίσαμε τα "μέτρα απόδοσης" προκειμένου να αξιολογήσουμε τις διάφορες υλοποιήσεις του προβήματος.

1.1 Μαθηματικό Μοντέλο:

$\boldsymbol{B}$: Συνόλο, σκακιέρα $\boldsymbol{n*n}$
όπου **n** είναι η μεταβλητή για των αριθμό των στηλών και των γραμμών.

$$y(i,j) = \begin{cases} 1 & ,αν\ υπάρχει\ βασίλισσα\ στην\ θέση\ (i,j)\ στην\ σκακιέρα\ B \\ 0 & ,αλλιώς \end{cases}$$

- $\sum_{i=1}^{n}$ y(i,j) =1 ,$\forall$ $i \in$ *{1,..,n}* ,μόνο μια βασίλισσα σε κάθε γραμμή.
- $\sum_{j=1}^{n}$ y(i,j) =1 ,$\forall$ $j \in$ *{1,..,n}* ,μόνο μια βασίλισσα σε κάθε στήλη.
- $\sum_{j+i=k}^{n}$ y(i,j) $\leq$1 ,$\forall$ $k \in$ *{-n+1, ..n-1}* , πάνω διαγώνιος .
- $\sum_{j-i=k}^{n}$ y(i,j) $\leq$1 ,$\forall$ $k \in$ *{-n+1, ..n-1}* , κάτω διαγώνιος.

1.2 Καθορισμός Μέτρων απόδοσης:

Τα μέτρα απόδοσης που θα χρησιμοποιήσουμε για να αξιολογήσουμε την εκάστοτε υλοποίηση είναι ***α)*** το πόσο κοντά φτάνουμε στη βέλτιστη λύση h=0, όπου h ο αριθμός των "ζευγαριών"-βασιλισσών που απειλούνται, ***β)*** ο χρόνος που χρειάζεται για να πλησιάσουμε στην βέλτιστη λύση ο οποίος είναι ανάλογος της χωρικής πολυπλοκότητας (διάσταση-μέγεθος της σκακιέρας και αριθμός βασιλισσών) και την πολυπλοκότητα της υλοποίησης. Αξίζει να σημειώθει ότι ανάλογα με τον αλγοριθμό δεν είναι δεδομένο ότι θα φτάσουμε στην βέλτιστη λύση h=0 π.χ. για hill-climb μπορεί να φτάσουμε σε μία υποβέλτιστη λύση.

2 Μέρος Β

2.1 Simulated Annealing:

Ο πρώτος αλγόριθμος που επιλέξαμε για την επίλυση του προβλήματος των N-Βασιλισσών είναι ο αλγόριθμος ***Simulated Annealing***. Ο λόγος που υλοποιήσαμε τον αλγόριθμο αυτό οφείλεται κυρίως σε ένα paper το οποίο μελετήσαμε. Το συγκεκριμένο paper *(Comparison of Heuristic Algorithms for the N-Queen Problem, Ivica Martinjak & Marin Golub)* κάνει αναφορά και συγκρίνει ορισμένους ευριστικούς αλγόριθμους, εκ των οποίων ένας από αυτούς είναι και το ***Simulated Annealing(SA)***. Πιο συγκεκριμένα, καταλήγει στο συμπέρασμα ότι η πολυπλοκότητα του αλγορίθμου είναι $\boldsymbol{O(n^2)}$ και ότι ο χρόνος εκτέλεσής του είναι πολυωνυμικός.

Ο αλγόριθμος ***Simulated Annealing*** (προσομοίωση ανόπτησης) είναι μια ευριστική τεχνική η οποία χρησιμοποιείται για να ξεφύγουμε από μία τοπική βέλτιστη λύση. Η τεχνική αυτή βασίζεται στην μέθοδο ψύξης ενός μετάλλου κατά την επεξεργασία του (γνωστή ως "ανόπτηση"). Σε κάθε επανάληψη, η νέα υποψήφια λύση Υ (την οποία επιστρέφει η ευριστική συνάρτηση), γίνεται η νέα καλύτερη λύση του προβλήματος X (καθώς και η νέα τρέχουσα κατάσταση), αν είναι καλύτερη από την τρέχουσα καλύτερη λύση. Ωστόσο, μερικές φορές επιτρέπεται η αντικατάσταση της τρέχουσας λύσης με μία νέα υποψήφια λύση ακόμα και αν η νέα αυτή κατάσταση δεν είναι καλύτερη από την τρέχουσα. Ο ρόλος της "θερμοκρασίας" T, είναι αυτό που καθορίζει πότε επιτρέπονται αυτές οι αποδοχές. Η λογική αυτή αποτελεί ένα μηχανισμό διαφυγής από την τοπικά βέλτιστη λύση. Όσο υψηλότερη είναι η τιμή της θερμοκρασίας, τόσο μεγαλύτερη είναι η πιθανότητα να αντικατάστασης της τρέχουσας λύσης με μία χειρότερη λύση. Η τιμή της μεταβλητής T μειώνεται κατά τη διάρκεια του χρόνου σύμφωνα με το συντελεστή ψύξης α.

```
simulated-annealing(initial solution)                                  1
  let solution be initial                                              2
  let t be an initial temperature                                      3
  until t is almost zero                                               4
    let neighbor be a random neighbor of solution                      5
    if the cost of neighbor is less than the cost of solution          6
        let solution be neighbor                                       7
        stop if the cost is now 0                                      8
    otherwise                                                          9
        let c be the cost increase                                     10
        compute p = e^(-c/t)                                           11
        with probability p, let solution be neighbor                   12
    multiply t by a decay rate                                         13
  return solution                                                      14
```

Simulated Annealing Pseudocode

Για την επίλυση του προβλήματος, αρχικά χρειάστηκε να δημιουργήσουμε μία κλάση η οποία αφορά τη σκακιέρα στην οποία θα τοποθετήσουμε τις βασίλισσες. Το αρχείο με ονομασία *Board.py* την κλάση με όνομα **chessBoard**. Στη κλάση αυτή βρίσκονται όλες οι απαρατήτες συναρτήσεις που χρειάστηκαν για να καταφέρουμε να βρούμε μία λύση για το πρόβλημά μας. Συγκεκριμένα περιέχει τις εξείς μεθόδους:

- createBoard() - δημιουργεί ένα N*N πίνακα και τοποθετεί τις βασίλισσες
- updateBoard() - ενημερώνει τον πίνακα (εναλλαγή γειτώνων στο ΣΑ αλγόρυθμο)

- printBoard() - εκτυπώνει τον πίνακα
- createAndEnalBoard() - Βοηθητική συνάρτηση
- checkForConflicts() - Ελέγχει για ονφλιστς μεταξύ των βασιλισσών
- prepareForVisualize() - Μετατρέπει τον N*N πίνακα σε διάνυσμα γραμμής μεγέθος N. Σε κάθε θέση (στήλη) του διανύσματος αυτού, βρίσκεται η θέση γραμμής της βασίλισσας στον πίνακα. Το διάνυσμα αυτό, το τροφοδοτούμε στο μία νέα κλάση που έχουμε δημιουργήσει ώστε να παράξει και να αποθηκεύσει μια εικόνα. (αρχείο *visualize.py* με κλάση GameBoard()).

Στη συνέχεια προχωρήσαμε στην υλοποίηση του αλγορίθμου ***Simulated Annealing***, έχοντας ως πρότυπο τον παραπάνω ψευδοκώδικα αλλά και από το paper που αναφέραμε στην αρχή. Η διαδικασία που ακολουθήσαμε είναι όπως αυτή περιγράφηκε νωρίτερα. Στο πέρας της εκτέλεσης λαμβάνουμε ως έξοδο τρεις τιμές. Η πρώτη αφορά τον αριθμό των iterations που χρειάστηκαν ώστε ο αλγόριθμος να βρει κάποια λύση. Στην συνέχεια επιστρέφει ένα αντικείμενου τύπου chessBoard το οποίο περιέχει τη λύση του προβλήματος (αν τα κατάφερε να βρει κάποια λύση). Από το στοιχείο αυτό μπορούμε να εξετάσουμε αν υπάρχουν conflicts και συνεπώς αν βρήκαμε λύση ή όχι στο πρόβλημα .Τέλος μας επιστρέφει και τον αντίστοιχο χρόνο εκτέλεσης του προγράμματος. Έχοντας αυτές τις πληροφορίες για κάθε αλγόριθμο, είμαστε σε θέση να τρέξουμε πολλαπλές φορές ένα test για ένα εύρος τιμών και να αξιολογήσουμε και να συγκρίνουμε του δύο αλγορίθμους που υλοποιήσαμε.

Αξίζει να σημειωθεί πως στο τέλος εκτέλεσης ενός αλγορίθμου αναπαριστάμε την λύση γραφικά, δημιουργώντας και αποθηκεύοντας μια εικόνα με μία σκακιέρα μεγέθους N*N και N βασίλησσες τοποθετημένες στις θέσεις λύσης τους (αν έχει βρεθεί, διαφορετικά στην τελευταία κατάσταση στην οποία βρέθηκε ο αλγόριθμος). Το αρχείο που φέρει τη λειτουργία αυτή είναι το *visualize.py* και περιέχει μια κλάση η οποία καλείται Gameboard. Για την δημιουργία του αρχείου αυτού χρησιμοποιήθηκε η βιβλιοθήκη Tkinter.Τέλος, τις εικόνες αυτές τις αποθηκεύουμε τόσο σε ξεχωριστό φάκελο για κάθε αλγόριθμο, αλλά και τις κατηγοριοποιούμε σύμφωνα με τον μέγεθος του προβλήματος. Στο όνομα της εικόνας αυτή, προσθέτουμε επίσης τον αριθμό των συγκρούσεων, αν υπάρχουν.

2.2 Min-Conflicts:

Για το ζητούμενο αυτό, προσεγγίσαμε το πρόβλημα των n – Queens με έναν ***Min – Conflicts*** αλγόριθμο. Γενικά ο ***Min – Conflicts algorithm*** είναι ένας search algorithm για την επίλυση ***C***onstraint ***S***atisfaction ***P***roblems.

Πιο συγκεκριμένα αρχίζουμε την επίλυση δίνοντας στο πρόγραμμά μας τον αριθμό N της σκακιέρας (N*N). Στη συνέχεια αρχικοποιούμε τις θέσεις των βασιλισσών (στο (i,i), με τυχαίο τρόπο για i από 0 μέχρι N-1) στη σκακιέρα και υπολογίζουμε τα hits κάθε κελιού, δεδομένων των αρχικών τους θέσεων. Γνωρίζοντας ότι μόνο μία βασίλισσα μπορεί να βρίσκεται στην ίδια γραμμή και στην ίδια στήλη, ελέγχουμε τις πιθανές κινήσεις της κάθε μιας μόνο στην στήλη στην οποία τοποθετήθηκε αρχικά. Αυτό μειώνει τις συνολικές συγκρίσεις που χρειάζονται για τον υπολογισμό των hits και τις πιθανές θέσεις μετακίνησης μίας βασίλισσας καθώς την μετακινούμε μόνο στη στήλη της, μεταξύ N γραμμών. Δηλαδή ελέγχουμε μόνο ανά γραμμή και διαγώνιο, κάτι που κάνει το πρόγραμμα συνολικά πιο γρήγορο.

Στη συνέχεια, όπως σε ένα ***CSP***, για δεδομένο αριθμό από βήματα-επαναλήψεις (πρωτού τα παρατήσουμε, στην δική μας περίπτωση για *1000,500,250..* iterations), επιλέγουμε τυχαία μία στήλη που το κελί στο οποίο βρίσκεται η βασίλισσα δεν έχει 0 hits, δηλαδή επιδέχεται βελτίωση, και υπολογίζουμε τα hits που θα είχαμε σε κάθε σειρά αν μετακινούσαμε εκεί τη βασίλισσα. Αφού βρούμε τις τιμές των ηιτς για κάθε γραμμή, επιλέγουμε greedily να μετακινήσουμε τη βασίλισσα στη γραμμή όπου τα hits είναι τα λιγότερα δυνατά. Η τυχαιότητα αυτή στην επιλογή βασίλισσας βοηθάει στην αποφυγή του αλγορίθμου από *local minima* (δηλαδή να γίνουν επιλογές μετακινήσεων που

δεν οδηγούν σε περεταίρω βελτίωση του προβλήματος, άρα όχι στη λύση του). Επίσης αν η μετακίνηση μιας βασίλισσας μπορούσε να οδηγήσει μόνο σε χειρότερες, σε θέμα hits καταστάσεις από ότι είναι ήδη, τότε η βασίλισσα δεν μετακινείται και παραμένει στη θέση της.

Στην περίπτωση όπου για όλες τις βασίλισσες τα hits της θέσης στην οποία βρίσκονται είναι 0, τότε έχουμε επιλύσει το πρόβλημα και προχωράμε στην αναπαράστασή του. Αν όμως μέσα στον αριθμό iterations που επιλέξαμε δεν βρούμε λύση, τότε τερματίζουμε το πρόγραμμα επιστρέφοντας exception με τον αριθμό των ζευγαριών - βασιλισσών που έμειναν να απειλούνται.

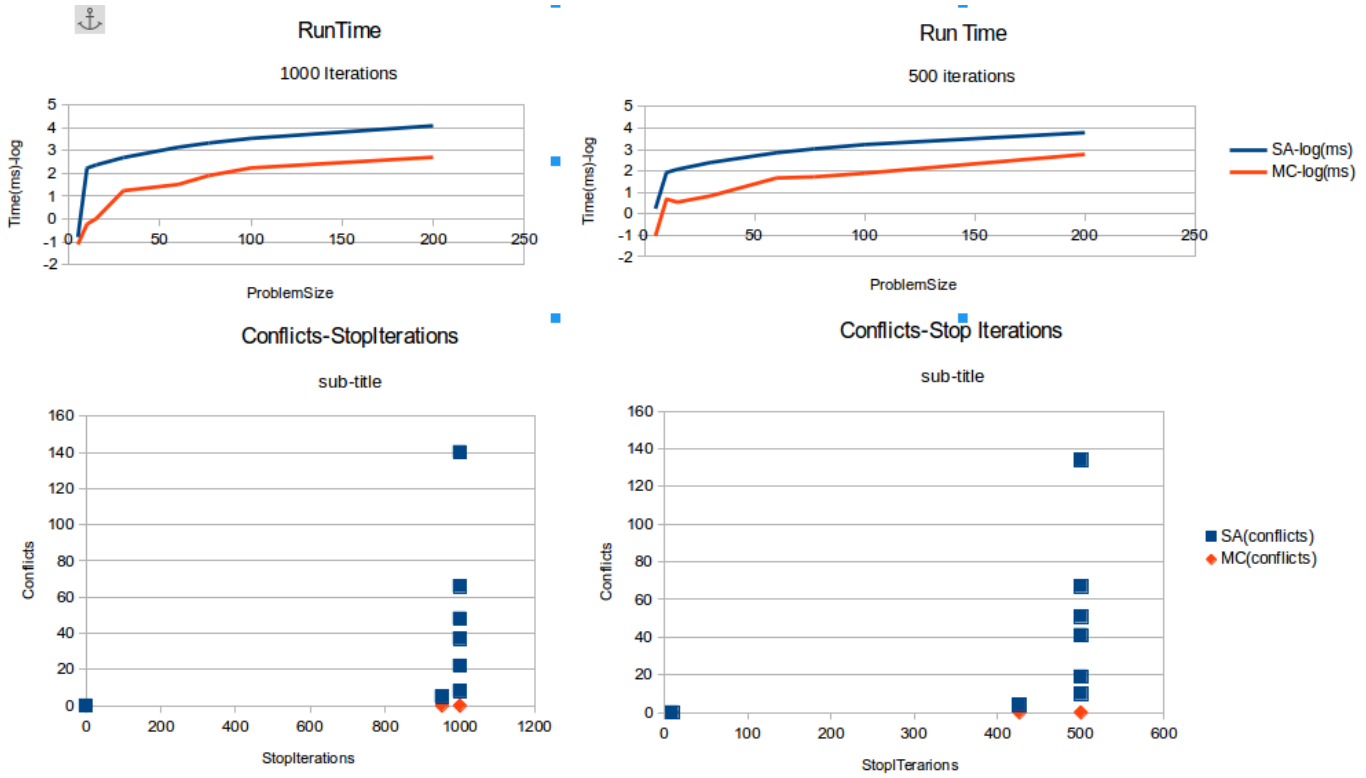
Ο χρόνος επίλυσης αυτού του αλγορίθμου είναι ανεξάρτητος του μεγέθους του προβλήματος. Όσο μεγάλο και αν είναι το πρόβλημα θα το λύσει σε σχεδόν κοινό μικρό αριθμό βημάτων.

2.3 Successor Function:

Όσον αφορά την *successor function* έχουμε ορίσει ουσιαστικά μία συνάρτηση που δημιουργεί τυχαία ένα πίνακα με τις πιθανές θέσεις των βασιλισσών στην σκακιέρα. Αυτόν τον πίνακα τον θεωρούμε ως μία αρχική κατάσταση του προβλήματος μας, την οποία θέτουμε ως είσοδο, ώστε ο κάθε αλγόριθμος να ξεκινήσει για την εύρεση λύσης. Οι μεταβάσεις από την μία κατάσταση στην επόμενη ορίζονται αυστήρα από τα βήματα του κάθε αλγορίθμου.

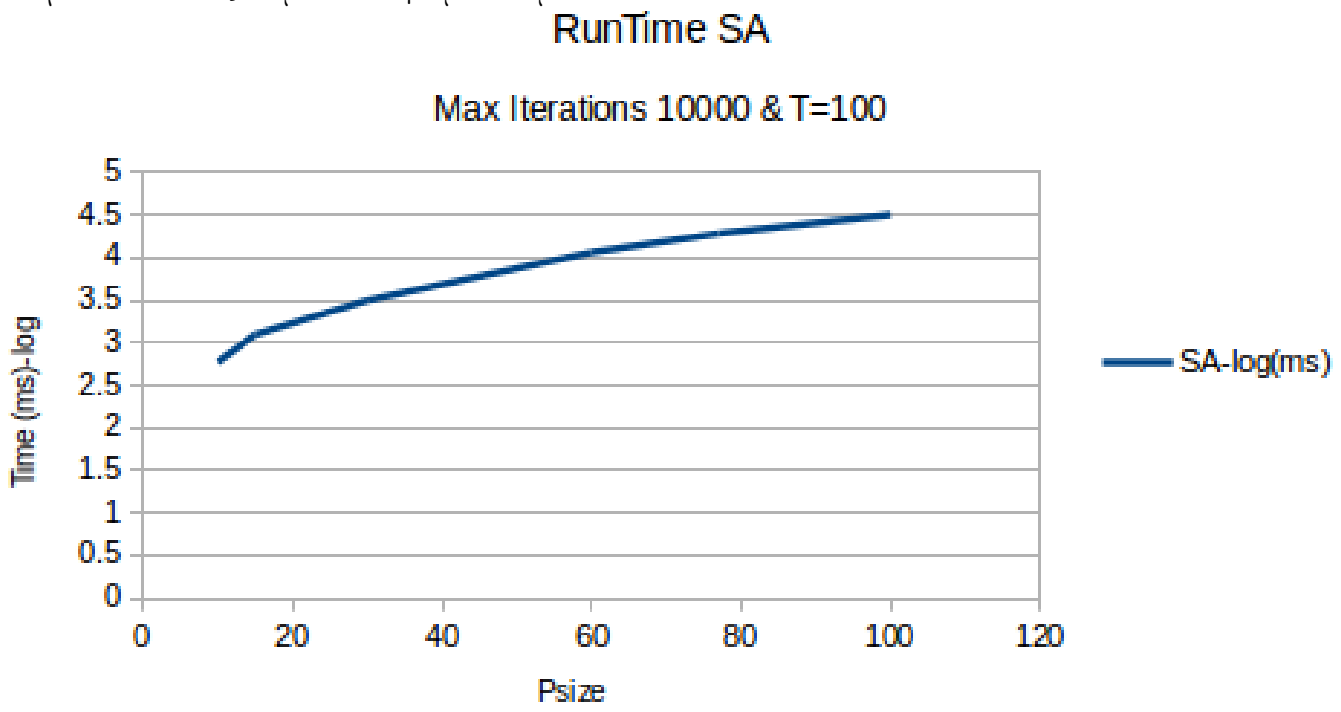
3 Πειράματα-Μετρήσεις

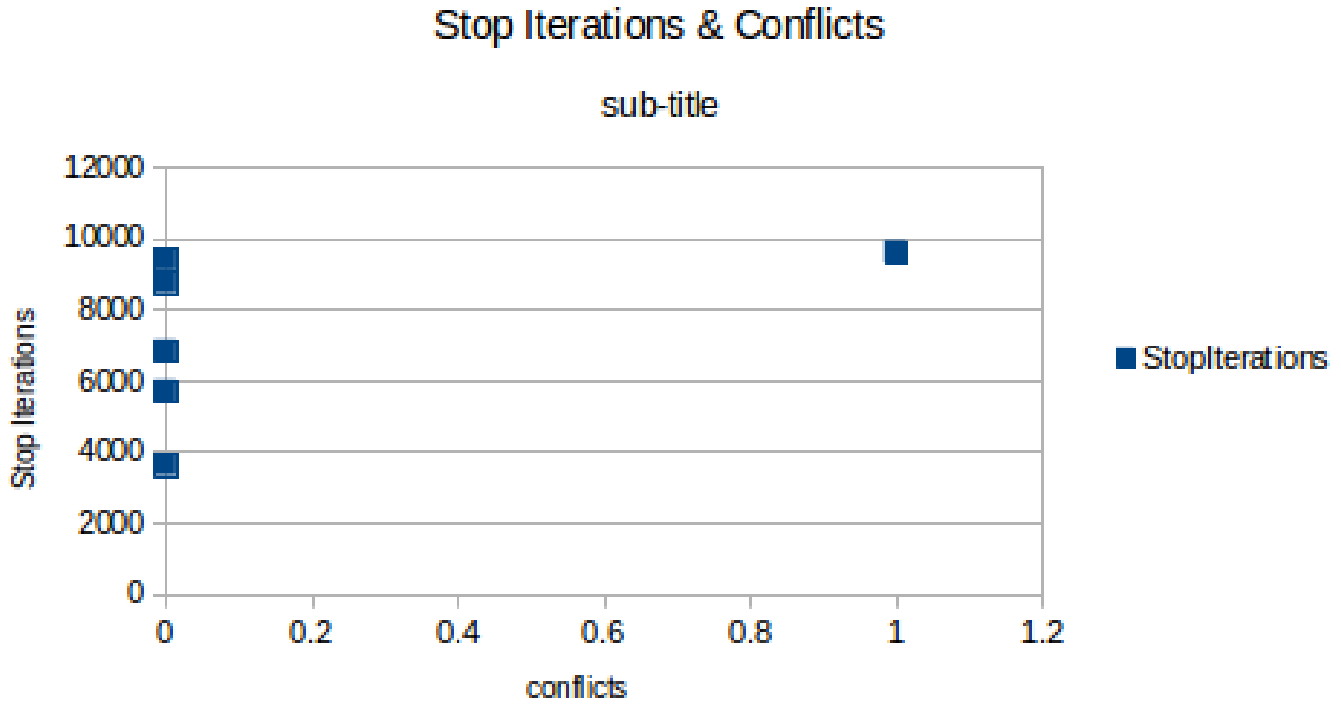
Σε αυτό το κομμάτι της εργασίας ασχοληθήκαμε με την αξιολόγηση των δύο αλγορίθμων που επιλέξαμε. Πιο συγκεκριμένα ορίζαμε ένα μέγιστο αριθμό επαναλήψεων που θα τρέχει ο κάθε αλγόριθμος•για κάθε είσοδο με διαφορετικό N εκτελούσαμε τον κώδικα μας 10 φορές καταγράφοντας τις μέσες τιμές του χρόνου εκτέλεσης,του αριθμού των συγκρούσεων και του αριθμό των επαναλήψεων για την εύρεση λύσης. Τα αποτελέσματα αυτά έχουν καταγραφεί αναλυτικά στο αρχείο *Out.txt* στο φάκελο του κώδικα. Ωστόσο παραθέτουμε παρακάτω τα γραφήματα για ένα αντιπροσωπευτικό δείγμα πειραμάτων μαζί με τους αντίστοιχους πίνακες.



Συγκρίση των δύο αλγορίθμων για μέγιστο αριθμό επαναλήψεων 500 & 1000

Στην συνέχεια αυξήσαμε τον μέγιστο αριθμό των επαναλήψεων(μ.α.ε) για τον αλγοριθμό Simulated Annealing σε 10000 και επιπλέον αλλάξαμε την μεταβλητή θερμοκρασίας στο 100, αντίθετα με πριν που το αυξάναμε ανάλογα με τον μ.α.ε .





Τροποποιήση παραμέτρων για SA

Τέλος παρακάτω φαίνονται και όλοι οι σχετικοί συγκριτικοί πίνακες από τους οποίους προέκυψαν τα παραπάνω γραφήματα.Να σημειωθεί ότι οι χρόνοι στα γραφήματα είναι σε λογαριθμική

κλίμακα ενώ στους πίνακες οι πραγματικές τους τιμές σε *ms*.

Max Number of Iterations=500

Psize	SA(ms)	MC(ms)	iterations(stop)	SA(conflicts)	MC(conflicts)
5	1.736	0.094	9	0	0
10	85.083	4.86	426	4	0
15	119.96	3.539	500	10	0
30	248.576	6.887	500	19	0
60	709.465	47.076	500	41	0
77	1070.148	53.164	500	51	0
100	1704.661	78.538	500	67	0
200	6114.029	596.979	500	134	0

Max Number of Iterations=1000

Psize	SA(ms)	MC(ms)	iterations(stop)	SA(conflicts)	MC(conflicts)
5	0.1618	0.0771	0	0	0
10	169.5	0.597	952	5	0
15	233.456	1.0368	1000	8	0
30	491.678	17.422	1000	22	0
60	1388.307	32.604	1000	37	0
77	2137.435	80.596	1000	48	0
100	3386.998	172.864	1000	66	0
200	12075.165	506.878	1000	140	0

Max Number of Iterations=10000 & T=100

Psize	SA-log(ms)	Conflicts	StopIterations
10	588.328	0	3640
15	1244.452	0	5700
30	3132.622	0	6842
60	11497.388	0	8773
77	19164.972	0	9411
100	31623.677	1	9631

Συγκριτικοί Πίνακες

4 Συμπεράσματα

Από όλα τα παραπάνω δεδομένα προκύπτει ότι η αναζήτηση με ***Min-Conflicts*** απαιτεί πολύ μικρότερο αριθμό επαναλήψεων και συνεπώς λιγότερο χρόνο για να φτάσει σε μια λύση, ενώ ο ***Simulated Annealing*** απαιτεί 5-10 φορές περισσότερες επαναλήψεις. Αυτό συνεπάγεται παραπάνω χρόνο για να λύσει το πρόβλημα, κάθως επίσης δεν βρίσκει λύση πάντα καθώς το μέγεθος του προβήματος αυξάνει. Επίσης αξιοσημείωτο είναι και ο ρόλος που παίζει η παράμετρος T του αλγορίθμου (SA), κάθως για μέγαλες τιμές του T μένει πιο εύκολα σε υποβέλτιστες λύσεις αντίθετα για T=100 καταλήγει σε βέλτιστη λύση σχεδόν πάντα.

5 Παράρτημα

Κώδικες-Παραδείγματα

- http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4DC9292839FE7B1AFABA1EDB8183
  doi=10.1.1.57.4685&rep=rep1&type=pdf

- https://pdfs.semanticscholar.org/ba4b/31ca6ce1f85056afe213270b9d90c446fbd2.pdf

- https://en.wikipedia.org/wiki/Min-conflicts_algorithm

- https://gist.github.com/vedantk/747203

- http://modelai.gettysburg.edu/2016/pyconsole/ex3/index.html

For how to execute the programm look at README.txt file in source code.