ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

# [HRY 419]-Ανάπτυξη Εργαλείων CAD για Σχεδίαση Ολοκληρωμένων Κυκλωμάτων

Αναφορά 4ου εργαστηρίου

*Ιωάννης Περίδης*
*A.M. 2018030069*

*10 Μαΐου* 2022

## 1 Εισαγωγή:

Στην εργαστηριακή άσκηση αυτή , μας ζητήθηκε να γίνει placement των standard shells, με σκοπό την βελτιστοποίηση των ελάχιστων γραμμών της τελικής διάταξης, χρησιμοποιώντας μια μέθοδο Monte Carlo/Simulated Annealing. Μετά την διαδικασία αυτή, θα έχουμε μια ολοκληρωμένη λύση Place & Route για ένα κύκλωμα.

## 2 Αλγόριθμος Simulated Annealing & Υλοποίηση:

Η μέθοδος Simulated Annealing , είναι μια επαναληπτική μέθοδος που μας βοηθά να φτάσουμε στο βέλτιστο Placement ή σε κάτι που βρίσκεται πολύ κοντά σε αυτό. Λειτουργεί ως εξής.

Έχουμε μια αρχική κατάσταση, δηλαδή το πρώτο placement των std shells, το οποίο παίρνεται σαν είσοδος έτοιμο από το netlist.Πάνω σε αυτό το placement καλείται η συνάρτηση Route and Compact η οποία αφού κάνει του routing και το compaction, επιστρέφει σαν αποτέλεσμα τον τελικό αριθμό σειρών που θα έχει η λύση. Μέσα σε έναν βρόγχο επαναλήψεων , επιχειρούμε πάνω στο αρχικό placement να κάνουμε 1 τυχαία (θα εξηγηθεί παρακάτω αναλυτικά το πως) αλλαγή θέσεων μεταξύ 2 std shells. Υστέρα , μετά την αλλαγή ξανακαλούμε την συνάρτηση Route and Compact , για να βρούμε την νέα λύση, δηλαδή τον νέο αριθμών σειρών που χρειάζεται το κύκλωμα. Αν η λύση αυτή είναι καλύτερη, αυτό σημαίνει ότι θα δίνει λιγότερες σειρές, τότε κρατάμε την αλλαγή αύτη θέσεων στα shells πάντοτε και την αποθηκεύουμε στον πίνακα με τα placements. Αντιθέτως, αν η λύση είναι χειρότερη ή ίδια με προηγουμένως, αρά θα δίνει ίσες ή παραπάνω σειρές , τότε συνήθως δεν την κρατάμε. Συγκεκριμένα, για να αποφευχθεί το να κολλήσει ο αλγόριθμος σε κάποιο τοπικό μέγιστο και να μην μπορεί να βρει την βέλτιστη λύση ( το ολικό μέγιστο), πρέπει μετά από κάποιες επαναλήψεις να αποδεχόμαστε και μια χειρότερη λύση από αυτή που έχουμε τώρα και να την αποθηκεύουμε και στο Placement ( θα εξηγηθεί παρακάτω αναλυτικά το πως). Η διαδικασία αυτή επαναλαμβάνεται μέχρι να φτάσουμε έναν συγκεκριμένο αριθμό επαναλήψεων που έχουμε προ επιλεγμένο. Τέλος, αφού έχει δημιουργηθεί το ολικό placement μας, που θα είναι κάτι κοντά στο βέλτιστο, τότε τρέχουμε μια τελευταία φορά την Route and Compact ,και δίνουμε σαν είσοδο αυτό και έτσι βρίσκουμε την τελική λύση με τις ελάχιστες σειρές.

Η υλοποίηση των παραπάνω σε κώδικα ήταν αρκετά απλή. Ο αλγόριθμος σχεδιάστηκε, όπως περιγράφεται μέσα σε έναν βρόγχο while, ο οποίος τερματίζει όταν η συνθήκη ελέγχου if για το αν ο αριθμός της επανάληψης ξεπέρασε το τερματικό όριο , γίνει αληθής. Μέσα στον

βρόγχο ,γίνεται έλεγχος με if που συγκρίνει τις σειρές και βλέπει αν η λύση είναι καλύτερη ή χειρότερη. Αν είναι χειρότερη , ξαναγίνεται έλεγχος με If για το αν ο αριθμός επαναλήψεων, είναι μεγαλύτερος ή όχι από το όριο αποδοχής χειρότερης λύσης. Σημαντικό είναι να καταλάβουμε πως δημιουργείται ο τελικός πίνακας Placement[] , ο οποίος περιέχει την βέλτιστή μας διάταξη. Δημιουργούμε στην αρχή τον πίνακα newPlacement[] ο οποίος είναι ίδιος με τον Placement[] αρχικά. Σε κάθε επανάληψη, γίνεται η αλλαγή κελιών στον πίνακα αυτό και μπαίνει σαν όρισμα στην RouteAndCompact(), αν η αλλαγή δώσει καλύτερο αποτέλεσμα , τότε αντιστοιχίζουμε τα στοιχεία του placement[] με αυτά του newPlacement[], αν όχι , τότε ο Placement[] δεν αλλάζει παραμένει ίδιος. Τελικά η συνάρτηση RouteAndCompact(), καλείται 1 φορά έξω από τον βρόγχο, με όρισμα τώρα τον placement[], ο οποίος έχει διαμορφωθεί μόνο με τις καλύτερες λύσεις και παίρνουμε το τελικό αποτέλεσμα.

# 3 Recipe:

Το recipe ή αλλιώς η λεγόμενη συνταγή του αλγορίθμους μας, αποτελούν ορισμένα χαρακτηριστικά που ορίζουν σε μεγάλο βαθμό την αποδοτικότητα και την ταχύτητα της τελικής μας λύσης. Τα χαρακτηριστικά αυτά είναι, το όριο αποδοχής χειρότερης λύσης και το όριο της τερματικής συνθήκης του αλγορίθμου.

Όσον αφορά το όριο αποδοχής χειρότερης λύσης, πρέπει να γίνει κατανοητό πως στην αρχή που τρέχουμε τον αλγόριθμο, επιθυμούμε να δεχόμαστε μια κακή λύση πιο συχνά με μικρότερο όριο. Αντίθετα, όσο ο αλγόριθμος τρέχει για περισσότερη ώρα και φτάνει σε μεγαλύτερο βάθος, δεν θέλουμε να δεχόμαστε τόσο συχνά χειρότερες λύσεις. Επομένως, αρχίζουμε από έναν μικρό αριθμό ορίου και αφού δεχτούμε κάθε φορά μια χειρότερη λύση, αυξάνουμε το όριο. Στην άσκησή μας, χρησιμοποιήθηκε αρχικό όριο acceptBadSolution=8 και κάθε φορά που ο αριθμός επαναλήψεων το ξεπέρναγε , τότε διπλασιάζαμε το όριο acceptBadSolution =acceptBadSolution*2 . Εννοείται πως κάθε φορά που διαλέγουμε μια λύση ο αριθμός επαναλήψεων , αρικοποιείται πάλι στο 0.

Συνεχίζοντας, όσον αφορά το όριο της τερματικής συνθήκης, ορίστηκε σαν την εκφώνηση στο 100. Αν το ξεπεράσει ο αριθμός επαναλήψεων , τότε γίνεται έξοδος από τον βρόγχο και τελειώνει ο αλγόριθμος του Placement.

Πρέπει να γίνει κατανοητό πως κανένα recipe δεν είναι ιδανικό για όλα τα κυκλώματα. Κάθε διαφορετικό κύκλωμα αποκρίνεται στην βέλτιστη του κατάσταση σε διαφορετικό recipe και απαιτείται μεγάλη τεχνογνωσία και πολλές δοκιμές για να βρεθεί το τέλειο recipe για κάθε ένα. Στόχος μας, είναι η εύρεση ενός recipe που να δίνει αρκετά καλή απόδοση (δηλαδή λίγες γραμμές στην λύση ) και γρήγορη ταχύτητα (δηλαδή να συγκλίνει στην λύση με μικρό αριθμό επαναλήψεων) σε όλα τα κυκλώματα.

# 4 Random Number Generator & Switch Std Shells:

Για την αλλαγή δύο τυχαίων standard shells, πρέπει να χρησιμοποιήσουμε μια γεννήτρια παραγωγής τυχαίων αριθμών. Η RNG αυτή, είναι σημαντικό να είναι καλή με ομοιόμορφη κατανομή και να έχει την ικανότητα να επαναλάβει τα πειράματά της . Αυτό διότι πολλές ακολουθίες τυχαίων αριθμών, διαφέρουν σε αρκετά μεγάλο βαθμό στην αποδοτικότητα της λύσης , δηλαδή στον ελάχιστο αριθμό σειρών που δίνουν. Επομένως είναι σημαντικό, να γνωρίζουμε ποιες ακολουθίες μας έδωσαν καλά και χειρότερα αποτελέσματα για κάθε κύκλωμα, με σκοπό να μπορούμε να αναπαράξουμε τις βέλτιστες λύσεις. Αυτό , γίνεται επιλέγοντας διαφορετικό σπόρο σαν όρισμα της RNG μας και σημειώνοντας τους σπόρους που ήταν οι καλύτεροι ,οι οποίοι φυσικά διαφέρουν για κάθε ένα κύκλωμα.

Στην περίπτωσή μας, χρησιμοποιήσαμε σαν RNG την συνάρτηση της γλώσσας C , void rand() και για να δώσουμε σαν όρισμα τον σπόρο που θέλαμε κάθε φορά, χρησιμοποιήθηκε η srand(seed).

Η υλοποίηση σε κώδικα της αλλαγής των 2 κελιών ήταν αρκετά απλή. Βρέθηκαν 2 τυχαίοι αριθμοί από το 0 μέχρι το Number Of Shells-1 (δηλαδή για όλα τα στοιχεία του πίνακα Placement). Αυτό έγινε μέσω της **rand()%(Number Of Shells-1).** Στην συνέχεια, αντικαταστάθηκαν **οι τιμές των 2 τυχαίων στοιχείων στον πίνακα newPlacement[]**. Με αυτόν τον τρόπο, η γεννήτρια θα βρίσκει 2 αριθμούς οι οποίοι θα χρησιμοποιούνται μετά για να αλλάξουν τα δύο std shells, χωρίς να έχει σημασία ποιος είναι ο αριθμός της πύλης (οπότε δεν επηρεάζει που οι πύλες έχουν τελείως τυχαίους αριθμούς).

### *Παρακάτω, φαίνεται η αλλαγή των κελιών σε κώδικα:*

```
shelNo1=rand()%(numOfShells-1);
shelNo2=rand()%(numOfShells-1);

//we swap the 2 random elements VALUES of the arrays
temp=newPlacement[shelNo1];
newPlacement[shelNo1]=newPlacement[shelNo2];
newPlacement[shelNo2]=temp;
```

# 5 Αποτελέσματα , Στατιστικά Στοιχεία, Γραφικές & Σχολιασμός:

### *Παρακάτω, φαίνονται τα αποτελέσματα της τελικής λύσης σε γραμμές (final rows),των συνολικών επαναλήψεων (total iterations), του χρόνου που χρειάστηκε σε δευτερόλεπτα (sec) για πολλούς διαφορετικούς σπόρους (seeds) και για μια συγκεκριμένη συνταγή (recipe) για κάθε ένα από τα αρχεία του dataset που δόθηκαν:*

Το recipe που χρησιμοποιήθηκε είναι αυτό που αναφέρεται παραπάνω:
**Accept Bad Solution=8,**
**Termination=100,**
**Accept Bad Solution*2** μετά από αποδοχή κακής λύσης.
Υπογραμμίστηκαν με κόκκινο οι σειρές που δίνουν την καλύτερη λύση με το αντίστοιχο seed τους.

| *FILE INPUT* | *ROWS IN SOLUTION* | *TOTAL ITERATIONS* | *SECONDS* | *SEED* |
|---|---|---|---|---|
| ***Askhsh_2_FA_1bit_v1*** | **2** | **244** | **0,047** | **1** |
| | 3 | 253 | 0,063 | 2 |
| | 3 | 229 | 0,063 | 3 |
| | 3 | 225 | 0,063 | 6 |
| | 3 | 227 | 0,063 | 10 |
| ***Askhsh_2_FA_1bit_v2*** | **2** | **227** | **0,047** | **1** |
| | 3 | 221 | 0,047 | 2 |
| | **2** | **221** | **0,047** | **3** |
| | **2** | **221** | **0,047** | **6** |

| | | | | |
|---|---|---|---|---|
| | 3 | 221 | 0,047 | 10 |
| ***Askhsh_2_FA_4bit_v1*** | 11 | 221 | 0,094 | 1 |
| | **4** | **297** | **0,121** | **2** |
| | 10 | 227 | 0,094 | 3 |
| | 10 | 265 | 0,133 | 5 |
| | **4** | **340** | **0,125** | **6** |
| | **4** | **290** | **0,172** | **9** |
| | 11 | 223 | 0,078 | 10 |
| ***Askhsh_2_FA_4bit_v2*** | 9 | 221 | 0,121 | 1 |
| | 12 | 221 | 0,140 | 2 |
| | 14 | 221 | 0,130 | 3 |
| | **4** | **371** | **0,190** | **5** |
| | 6 | 221 | 0,106 | 6 |
| | 12 | 221 | 0,131 | 9 |
| | 12 | 252 | 0,137 | 10 |
| ***Askhsh_2_FA_4bit_v3*** | 8 | 228 | 0,117 | 1 |
| | 5 | 333 | 0,195 | 2 |
| | 10 | 241 | 0,131 | 3 |
| | 10 | 283 | 0,133 | 5 |
| | **4** | **262** | **0,201** | **6** |
| | 6 | 248 | 0,118 | 9 |
| | 8 | 280 | 0,181 | 10 |
| ***Askhsh_2_FA_8bit_v1*** | 11 | 257 | 0,240 | 1 |
| | 21 | 224 | 0,187 | 2 |
| | 13 | 338 | 0,311 | 3 |
| | 22 | 222 | 0,178 | 5 |
| | **10** | **285** | **0,278** | **6** |
| | 25 | 226 | 0,209 | 10 |
| ***Askhsh_2_FA_8bit_v2*** | 18 | 230 | 0,196 | 1 |
| | 15 | 253 | 0,234 | 2 |
| | 17 | 295 | 0,300 | 3 |
| | 21 | 244 | 0,264 | 5 |
| | **11** | **336** | **0,344** | **6** |
| | 12 | 253 | 0,237 | 10 |

Παρατηρούμε λοιπόν πως ο σπόρος που χρησιμοποιείται στο κάθε κύκλωμα παίζει σημαντικό ρόλο στην ποιότητα του αποτελέσματος και είναι διαφορετικός για όλα τα κυκλώματα. Ένας αρκετά καλός σπόρος γενικά για κάθε κύκλωμα αποδείχθηκε να είναι το 6. Ακόμη, είναι φανερό πως πολλές φορές μπορεί να υπάρχουν παραπάνω από 1ας σπόρος που δίνει την καλύτερη λύση (λιγότερες γραμμές), αλλά σίγουρα θα διαφέρουν στο πλήθος επαναλήψεων και στον πραγματικό χρόνο που τρέχει το πρόγραμμα που χρειάστηκαν για να φτάσουν εκεί. Επομένως , επιλέγουμε αυτόν που θα φτάσει στο βέλτιστο αποτέλεσμα, με τις ελάχιστες επαναλήψεις.

***Παρακάτω, φαίνεται ο αντίστοιχος πίνακας με πριν, μόνο που αυτή τη φορά, έγινε αλλαγή του recipe. Συγκεκριμένα, αφαιρέθηκε το όριο αποδοχής κακής λύσης και ο αλγόριθμος έγινε άπληστος :***

Αναλυτικότερα, έγινε δοκιμή του αλγόριθμου, σε μια greedy (άπληστη ) μορφή του , όπου δεν δέχεται καμία χειρότερη λύση , αλλά μόνο καλύτερες. Αυτό έχει σαν αποτέλεσμα να υπάρχει όπως εξηγήθηκε παραπάνω η περίπτωση του να κολλήσει σε ένα τοπικό μέγιστο και να μην βρει την καλύτερη λύση .

| ***FILE INPUT*** | ***ROWS IN SOLUTION*** | ***TOTAL ITERATIONS*** | ***SECONDS*** | ***SEED*** |
|---|---|---|---|---|
| ***Askhsh_2_FA_1bit_v1*** | 2 | 148 | 0,039 | 1 |
| | **2** | **147** | **0,041** | **2** |
| ***Askhsh_2_FA_1bit_v2*** | 2 | 107 | 0,046 | 1 |
| | **2** | **101** | **0,081** | **2** |
| ***Askhsh_2_FA_4bit_v1*** | 6 | 102 | 0,077 | 6 |
| | **4** | **127** | **0,110** | **10** |
| ***Askhsh_2_FA_4bit_v2*** | 6 | 101 | 0,079 | 5 |
| | **4** | **188** | **0,133** | **10** |
| ***Askhsh_2_FA_4bit_v3*** | 4 | 262 | 0,187 | 6 |
| | **4** | **262** | **0,129** | **6** |
| ***Askhsh_2_FA_8bit_v1*** | 12 | 104 | 0,197 | 6 |
| | **10** | **104** | **0,142** | **2** |
| ***Askhsh_2_FA_8bit_v2*** | 16 | 102 | 0,193 | 6 |
| | **12** | **189** | **0,238** | **2** |

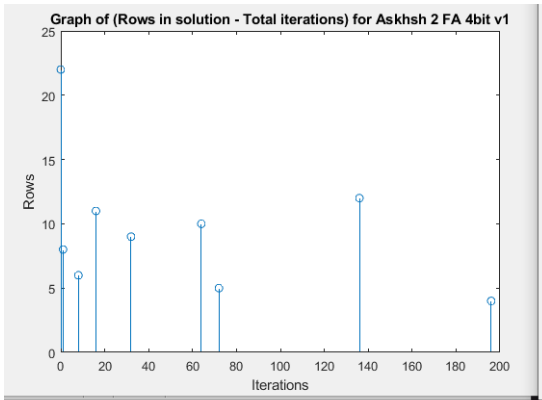
Παρατηρήθηκε πως όπως ήταν αναμενόμενο, όλες οι συνολικές επαναλήψεις , είναι σε πολύ μεγάλο βαθμό μειωμένες. Αυτό διότι δεν κάνει κύκλους σε χειρότερες λύσεις, μέχρι να βρει την καλύτερη.
Επίσης, στον πίνακα απεικονίζονται 2 σπόροι. Ο πρώτος σπόρος που δοκιμάστηκε ήταν ίδιος με τον σπόρο που έδινε την καλύτερη λύση προηγουμένως στο κάθε ένα κύκλωμα (με το παλιό recipe). Οι συγκεκριμένοι σπόροι αυτοί , έδωσαν χειρότερα αποτελέσματα από πριν (δηλαδή περισσότερες γραμμές στην λύση), αλλά φυσικά με λιγότερες επαναλήψεις.
Παρόλα αυτά, μετά από δοκιμές κι άλλον σπόρων, απεικονίστηκαν στον πίνακα και οι σπόροι που έδιναν τα καλύτερα αποτελέσματα για τον greedy τώρα αλγόριθμο. Τα αποτελέσματα αυτά ήταν εξίσου καλά σε ποιότητα λύσης με τα προηγούμενα , αλλά ήταν καλύτερα λόγω των λιγότερων επαναλήψεών τους.
Συμπερασματικά, παρατηρήθηκε πως ο αλγόριθμος με τους ίδιους σπόρους με πριν , πράγματι κόλλησε σε ένα τοπικό ελάχιστο σειρών και όχι στο ολικό, αλλά με κάποιους διαφορετικούς σπόρους κατάφερε και το βρήκε. Το γεγονός αυτό βέβαια δεν σημαίνει τίποτα , στην γενική μορφή του αλγορίθμου, απλά τυχαίνει στα κυκλώματα που εξετάστηκαν στο dataset μας, να βρίσκει και ο greedy αλγόριθμος το ολικό ελάχιστο σειρών, σε αλλά κυκλώματα μπορεί να μην το έβρισκε με κανέναν σπόρο.

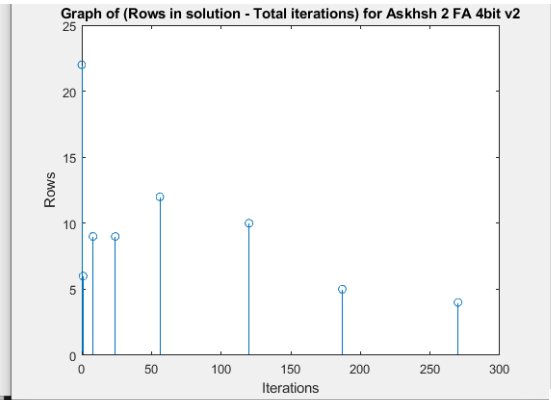
Στο σημείο αυτό πρέπει να αναφερθεί πως δοκιμάστηκε να τρέξει ο greedy αλγόριθμος για ένα αρκετά μεγαλύτερο τερματικό όριο , πλέον ίσο με 2700, αντί 100. Αποτέλεσμα αυτού ήταν πράγματι στον αθροιστή των 4Bits, να βρει μια ακόμα καλύτερη λύση από τις 4 γραμμές, δηλαδή να βρει και την optimal λύση και στην πραγματικότητα ίση με 3 γραμμές. Αντίστοιχα για τον αθροιστή των 8bit, κατάφερε να πλησιάσει ακόμα πιο κοντά στην βέλτιστη λύση , συγκεκριμένα στις 8 σειρές από τις 10 που έβρισκε πριν. Αυτό φυσικά ,

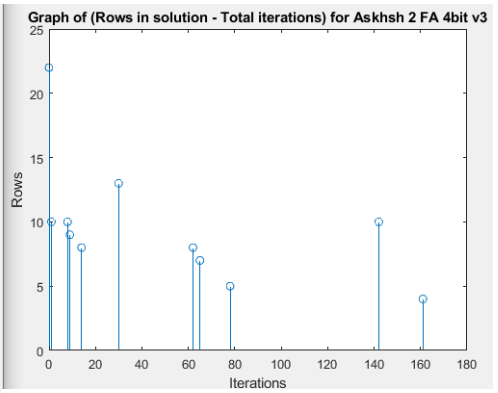
χρειάστηκε τεράστιο αριθμών επαναλήψεων της τάξης των 5000.

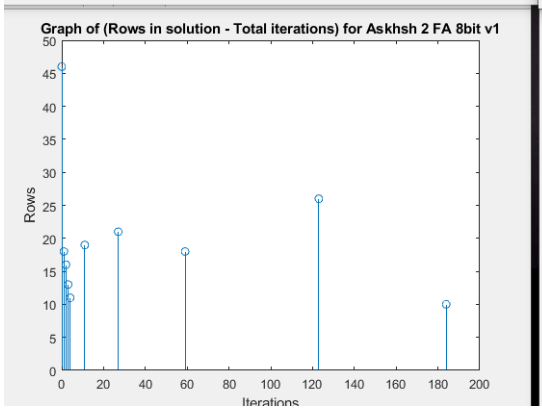
## *Παρακάτω, φαίνονται οι γραφικές παραστάσεις (Σειρών της Λύσης-Συνολικό Αριθμό Επαναλήψεων):*

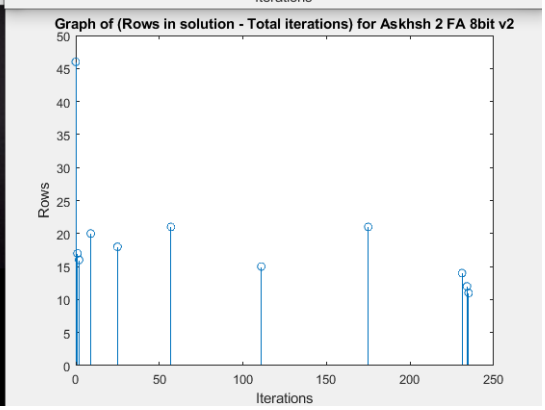
Σχεδιάστηκαν στο εργαλείο MATLAB οι γραφικές παραστάσεις (Rows in Solution-Total Iterations). Συγκεκριμένα, τοποθετήθηκε ένα σημείο κάθε φορά που αποδεχόμαστε μια λύση επειδή είναι καλύτερη ή κάθε φορά που αποδεχόμαστε μια χειρότερη λύση, επειδή φτάσαμε το όριο αποδοχής κακής λύσης. Παρακάτω φαίνονται οι γραφικές αυτές για τα κυκλώματα των αθροιστών 4 και 8 Bit.



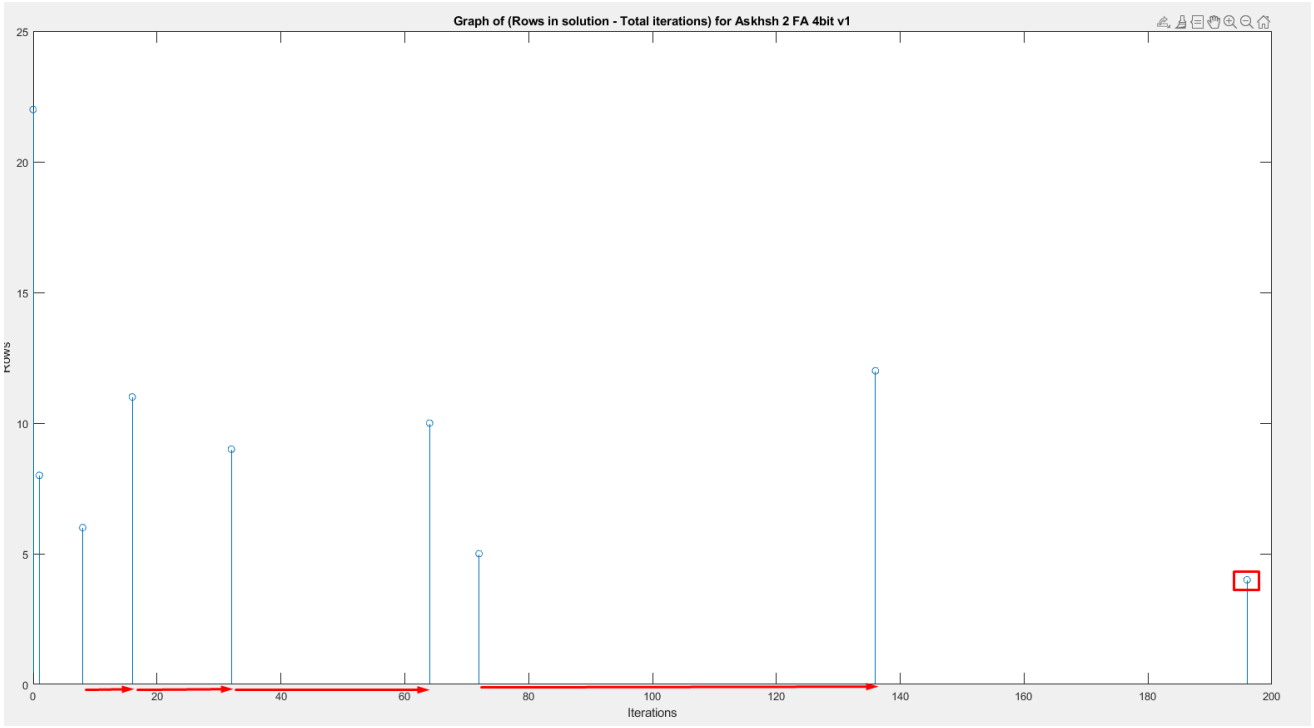



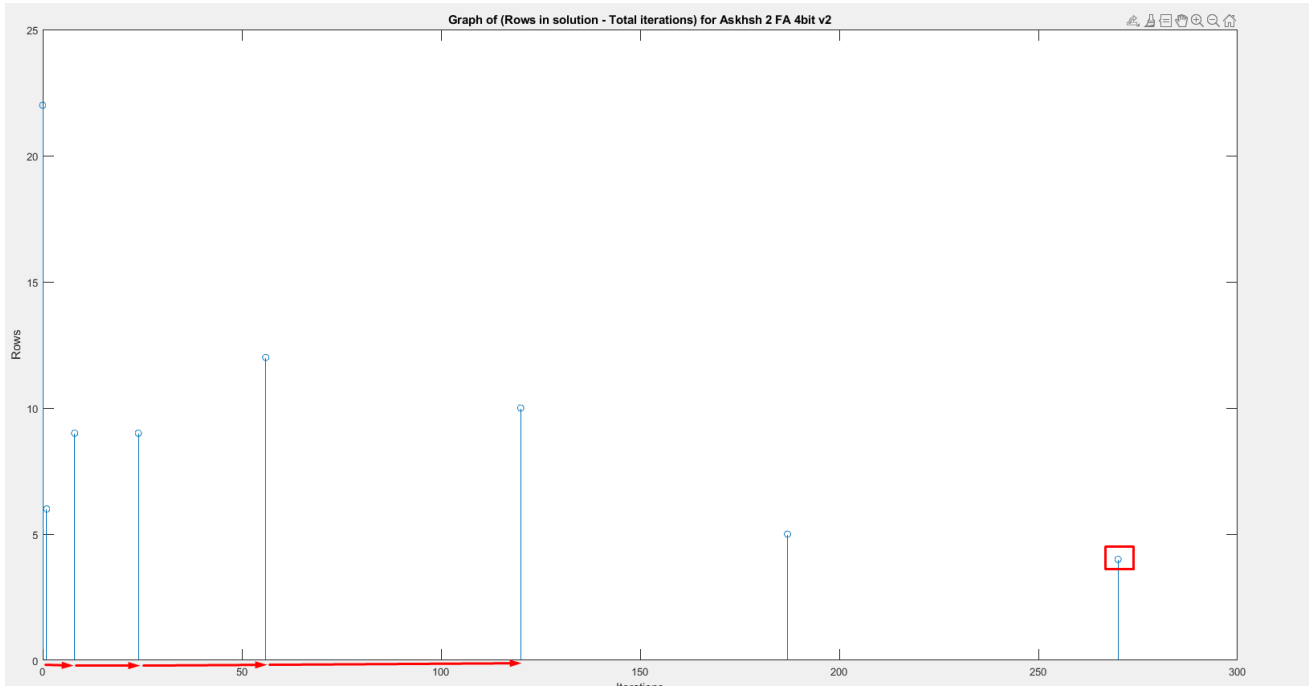



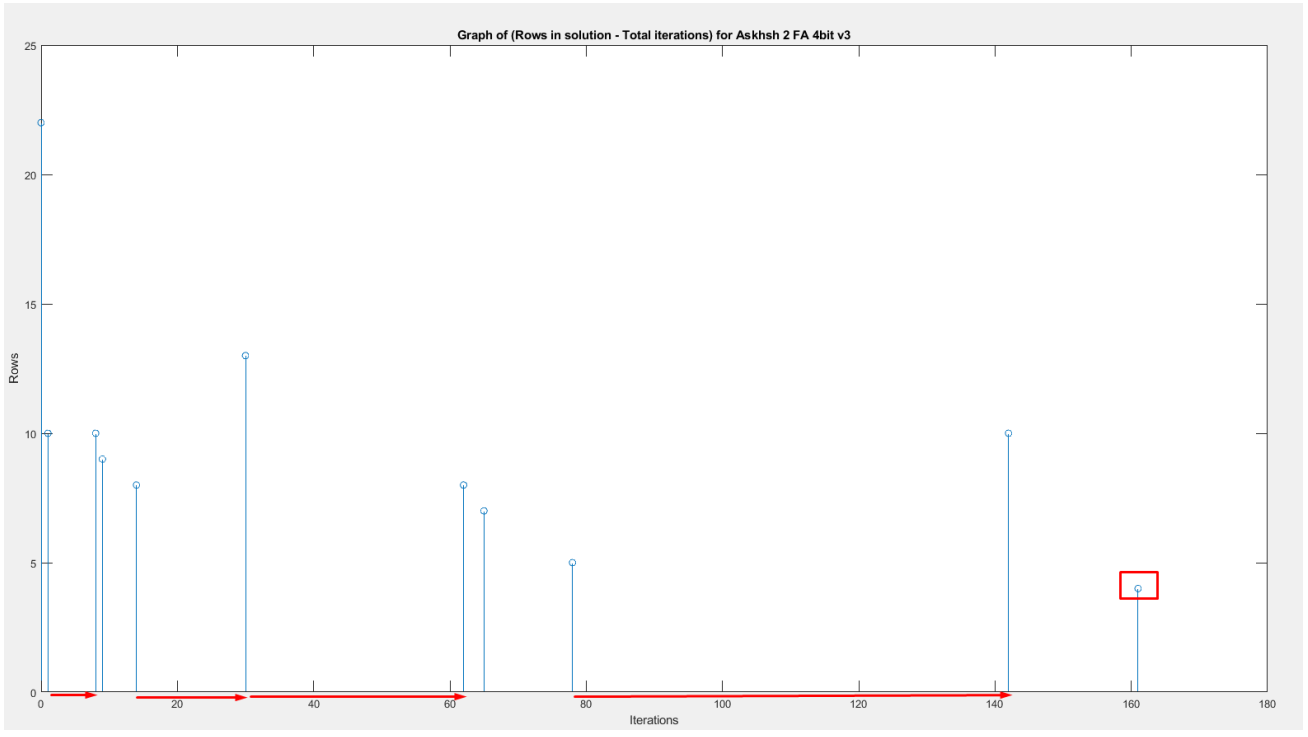

Στην συνέχει παραδίδονται οι ίδες γραφικές μεγενθημένες έτσι ώστε να φανούν αναλυτικά οι μεταβάσεις του αλγορίθμου του κάθε κυκλώματος. Έχουν σημειωθεί με κόκκινο βέλος οι μεταβάσεις που μας οδηγούν σε αποδοχή χειρότερης λύσης. Η βέλτιστη λύση έχει σημειωθεί με ένα κόκκινο τετραγωνάκι.

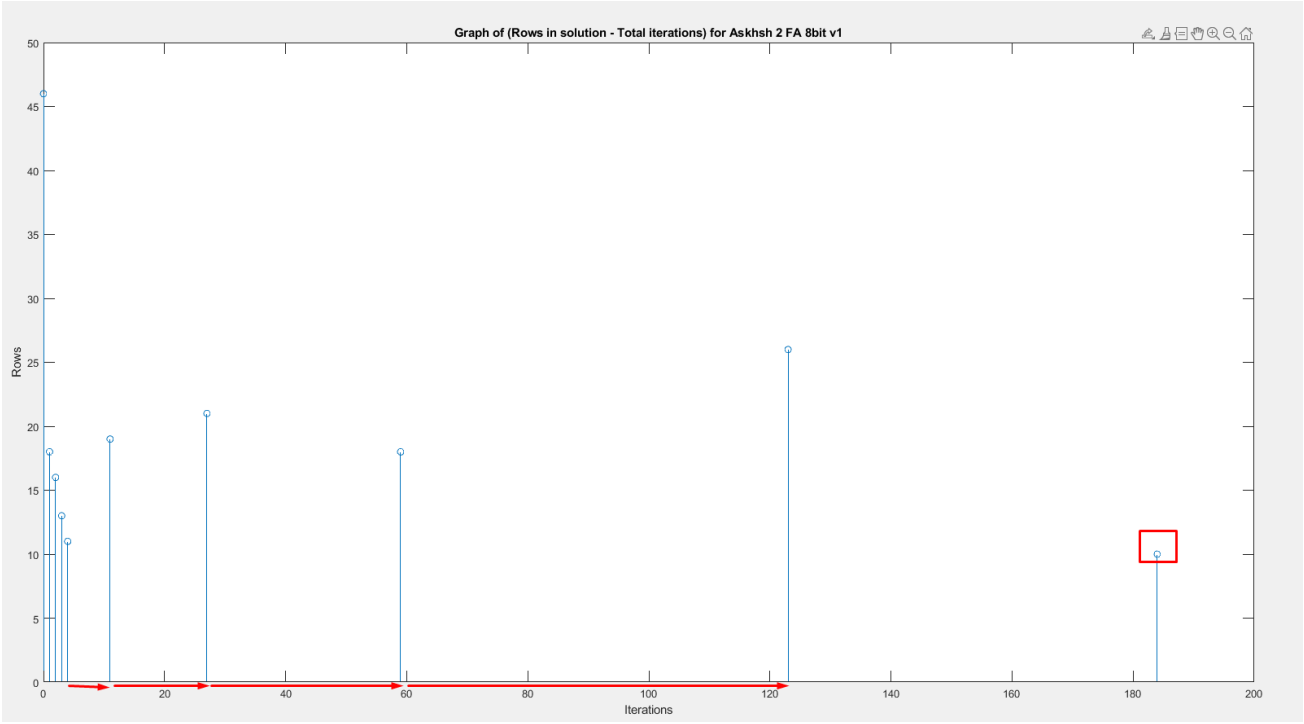
Θα παρατηρήσετε πως πράγαματι ο αλγόριθμος λειτουργέι σωστά. Αυτό διότι, όντως αποδεχόμαστε μια κακή λύση όσο προχωράει ο αλγόριθμσο τόσο και πιο σπάνια , φάινεται στο μήκος από τα κόκκινα βελάκια πως κάθε ένα είναι ίσο με 2 φορές από το προιγούμενο του. Επίσης καταλαβαίνουμε πως ο αλγόριθμος προχωράει βήμα- βήμα και μειώνει τις σειρές μέχιρ να βρεί την βέλτιστη λύση και όντως χρησιμοποιεί την αποδοχή μιας χειρότερης λύσης, για να ξεφύγει από το τπικό ακρότατο και ύστερα βρίσκει το ολικό.

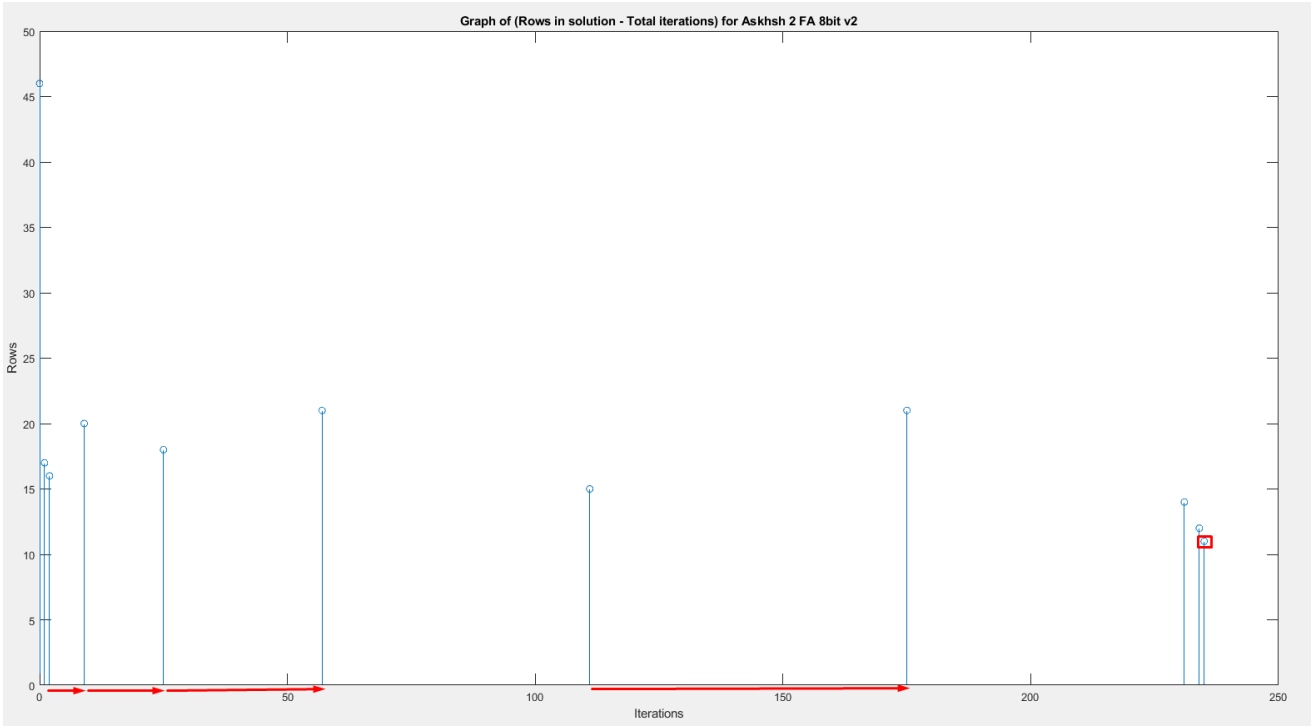
Να σημειωθεί εδώ πως το τελευταίο σημείο κάθε γραφικής είναι εκείνο το σημείο που γίνετια τελευταία φορά αποδοχή λύσης με οποιονδίποτε τρόπο. Ύστερα από αυτό το σημείο έχουμε κι άλλες επαναλήψεις (που δεν φαίνονται στο διάγραμμα), αλλά αυτές δεν προλαάβινουν να αποδεχτούν οποιαδήποτε άλλη λύση γαιτί ξεπερνάνε το όριο τερματισμού ( για αυτό δεν έχουν μπει στο διάγραμμα).


Graph of (Rows in solution - Total iterations) for Askhsh 2 FA 4bit v1
Rows
Iterations
0
5
10
15
20
25
0
20
40
60
80
100
120
140
160
180
200


Graph of (Rows in solution - Total iterations) for Askhsh 2 FA 4bit v2
Rows
0
5
10
15
20
25
0
50
100
150
200
250
300


Graph of (Rows in solution - Total iterations) for Askhsh 2 FA 4bit v3
Rows
Iterations
0
5
10
15
20
25
0
20
40
60
80
100
120
140
160
180




***Τέλος, φαίνεται παρακάτω ένα παράδειγμα αποτελέσματος που εκτυπώνει το πρόγραμμα μας, εκτυπώνει και την διαδικασία του αλγορίθμου και το τελικό αποτέλεσμα σε γραμμές και συνολικές επαναλήψεις:***

```
***These are the steps of the Place & Route algorithm:***
============================================================
We found a better solution: Iteration Number:  1 | Total Itterations:  0 | Tows in Solution:10
We accept a worse solution: Iteration Number:  9 | Total Itterations:  8 | Tows in Solution:10
We found a better solution: Iteration Number:  1 | Total Itterations:  9 | Tows in Solution: 9
We found a better solution: Iteration Number:  5 | Total Itterations: 14 | Tows in Solution: 8
We accept a worse solution: Iteration Number: 17 | Total Itterations: 30 | Tows in Solution:13
We accept a worse solution: Iteration Number: 33 | Total Itterations: 62 | Tows in Solution: 8
We found a better solution: Iteration Number:  3 | Total Itterations: 65 | Tows in Solution: 7
We found a better solution: Iteration Number: 13 | Total Itterations: 78 | Tows in Solution: 5
We accept a worse solution: Iteration Number: 65 | Total Itterations:142 | Tows in Solution:10
We found a better solution: Iteration Number: 19 | Total Itterations:161 | Tows in Solution: 4

*****ROWS IN FINAL SOLUTION*****
========================================
Number of rows after the Place & Route algorithm:4
The total Iterations needed:262
```