



ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πρόγραμμα Μεταπτυχιακών Σπουδών «Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Πτυχιακής Εργασίας	Μια Υλοποίηση του Αλγόριθμου του Arora για το πρόβλημα k-TSP An Implementation of Arora's Algorithm for the k-TSP problem
Ονοματεπώνυμο Φοιτητή	Μηνάς Αλβέρτης
Πατρώνυμο	Ιωάννης
Αριθμός Μητρώου	ΜΠΠΛ2204
Επιβλέπων	Χαράλαμπος Κωνσταντόπουλος, Καθηγητής

Ημερομηνία Παράδοσης

Οκτώβριος 2026

Τριμελής Εξεταστική Επιτροπή

Χαράλαμπος Κωνσταντόπουλος
Επιβλέπων Καθηγητής

Ιωάννης Βενέτης
Επικ. Καθηγητής

Ιωάννης Τασούλας
Επικ. Καθηγητής

Copyright ©

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν αποκλειστικά τον συγγραφέα και δεν αντιπροσωπεύουν τις επίσημες θέσεις του Πανεπιστημίου Πειραιώς.

Ως συγγραφέας της παρούσας εργασίας δηλώνω πως η παρούσα εργασία δεν αποτελεί προϊόν λογοκλοπής και δεν περιέχει υλικό από μη αναφερόμενες πηγές.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον Επιβλέποντα της εργασίας Καθηγητή Χαράλαμπο Κωνσταντόπουλο για την πολύτιμη υποστήριξη, τις επικοινωνιακές συμβουλές και την άριστη συνεργασία.

Περίληψη

Το Πρόβλημα του Πλανώδιου Πωλητή αποτελεί ένα από τα πρώτα γνωστά προβλήματα στην θεωρία των γραφημάτων, του οποίου η δημοφιλία εξηγείται από τις πρακτικές εφαρμογές στις οποίες μία αποδοτική υλοποίηση θα έδινε λύση. Ανά τα χρόνια έχει χρησιμοποιηθεί ως πρόβλημα σταθμός για την αξιολόγηση νέων αλγορίθμων και οι τεχνικές που έχουν εφαρμοστεί προσέφεραν στην Επιστήμη των Υπολογιστών με τις καινοτομίες τους στα θέματα των ευριστικών και προσεγγιστικών τεχνικών.

Στην παρούσα έρευνα παρουσιάζεται, επεξηγείται και υλοποιείται ο προσεγγιστικός αλγόριθμος του Sanjeev Arora, ο οποίος μετέπειτα επεκτείνεται ώστε να μπορεί να αντιμετωπίσει και μία παραλλαγή του Προβλήματος του Πλανώδιου Πωλητή, στην οποία ο ζητούμενος αριθμός των κόμβων μίας λύσης διαφοροποιείται από τον ορισμό του αρχικού προβλήματος. Παραθέτονται με επεξηγήσεις κομμάτια ψευδοκώδικα καθώς και αποτελέσματα της εκτέλεσης των αλγορίθμων σε κάθε περίπτωση.

Η παρούσα εργασία έχει ως στόχο να αποδείξει ότι αλγόριθμοι με αυστηρή μαθηματική θεμελίωση και μαθηματικά αποδεδειγμένη ακρίβεια είναι δυνατόν να έχουν την πρακτική αξία η οποία έχει πιο συχνά συνδεθεί με τους ευριστικούς αλγορίθμους στη σύγχρονη βιβλιογραφία. Σκοπεύει επίσης να δώσει κατευθυντήριες γραμμές για το πώς θα μπορούσαν να προσαρμοστούν τα τμήματα στα οποία η μεταφορά από τη θεωρία στην πράξη θα οδηγούσαν σε εξαντλητικούς υπολογισμούς.

Λέξεις Κλειδιά: Επιστήμη Υπολογιστών, Πρόβλημα Πλανώδιου Πωλητή, Προσεγγιστικοί Αλγόριθμοι, Πρόβλημα Βελτιστοποίησης

Abstract

The Travelling Salesperson Problem is one of the first well-known problems of graph theory, which owes its popularity to the number of practical applications that its implementation can solve. Over the years it has been used as a benchmark for the evaluation of newer algorithms and its proposed solution methods have offered a great number of innovations concerning matters related to heuristics and approximation techniques in Computer Science.

In this thesis, the approximation algorithm of Sanjeev Arora is presented, explained and implemented, and later expanded so that it can offer solution to a variation of the problem, where the needed number of nodes of a solution is differentiated. Code segments are presented in pseudocode, as well as the results of the algorithms' executions in each case.

The present work aims to prove that algorithms with strict mathematical foundation and mathematically proven accuracy also have the practicality that has lately been associated mainly with heuristic algorithms. It also aims to suggest adjustments to parts of the algorithms that appear to involve too many calculations when the theory is adapted for implementation.

Keywords: Computer Science, Travelling Salesperson Problem, Approximation Algorithms, Optimization Problem

Πίνακας Περιεχομένων

Copyright©	i
Ευχαριστίες	ii
Περίληψη	iii
Abstract	iii
KeyWords	iii
Πίνακας Περιεχομένων	iv
Κατάλογος Εικόνων	v
Κατάλογος Πινάκων	vii
1. Εισαγωγή	1
2. Σχετική Έρευνα	4
2.1 Αλγόριθμος του Arora	4
2.2 Αλγόριθμος του Gottlieb	8
3. TSP Αλγόριθμος του Arora	11
3.1 Φάση Προεπεξεργασίας	11
3.1.1	11
3.1.2	15
3.2 Φάση Τετραδικού Δέντρου	15
3.2.1	15
3.2.2	18
3.3 Φάση Εύρεσης Μονοπατιών	20
3.3.1	20
3.3.2	23
3.4 Αποτελέσματα	26
4. k-TSP Αλγόριθμος του Arora	40
4.1 k-TSP χωρίς άκρα	40
4.2 k-TSP με άκρα	42
4.3 Αποτελέσματα	45
5. Βελτιώσεις βασισμένες στον αλγόριθμο Gottlieb	57
5.1 Φάση Προεπεξεργασίας	57
5.2 Δυναμικός Προγραμματισμός	58
5.3 Αποτελέσματα	66
Συμπεράσματα	69
Πίνακας Ορολογίας	70
Πίνακας Συντμήσεων-Αρκτικόλεξων-Ακρονυμίων	70
Βιβλιογραφία	71

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

Εικόνα 1. Απλό παράδειγμα TSP	2
Εικόνα 2. Τα τελικά τετράγωνα σκιαγραφημένα	4
Εικόνα 3. Διαδικασία δημιουργίας μίας διαδρομής Arora	5
Εικόνα 4. Σταδιακή διάσπαση κύκλων	6
Εικόνα 5. Επιλογή διαδρομών χωρίς προορισμό για επίτευξη k-μονοπατιού, $k=2$	7
Εικόνα 6. k-διαδρομή με άκρα S-T, $k=3$.	8
Εικόνα 7. Ταξινομημένοι προορισμοί και παράθυρο των προορισμών 2 και 5 ($W_{2,5}$)	9
Εικόνα 8. Πιθανή διαδρομή Gottlieb ($i=12$, $d=11$, $k=6$).	10
Εικόνα 9. Ένας πιθανός κλοιός	12
Εικόνα 10. Πιθανοί κλοιοί με επιλογή κάτω αριστερά και πάνω αριστερά γωνίας	12
Εικόνα 11. Ο μετασχηματισμένος κλοιός	13
Εικόνα 12. Τοποθέτηση πλέγματος σε κλοιό ($e=1$) για $n=5$ πόλεις	14
Εικόνα 13. Οι αρχικές πόλεις και οι προβολές τους στο πλέγμα	14
Εικόνα 14. Προβολή πόλεων στην ίδια τομή	15
Εικόνα 15. Αντικατάσταση πόλης-αντιπροσώπου από συστάδα	16
Εικόνα 16. Οι τελικοί κόμβοι, σε κατάλληλο βάθος επιπέδου	17
Εικόνα 17. (a,b)-μετατόπιση του τετραδικού δέντρου	18
Εικόνα 18. Ορισμός θυρών για $k=4$	19
Εικόνα 19. Διαίρεση του τετραδικού δέντρου και προσθήκη θυρών	20
Εικόνα 20. Παραδείγματα υπολογισμού θυρών	20
Εικόνα 21. Τμήματα κόμβων	21
Εικόνα 22. Δημιουργία ενός τμήματος του κόμβου Q, αγνοώντας τις πόλεις	22
Εικόνα 23. Μερικά τμήματα για κάθε έναν από 4 αδερφικούς κόμβους	23
Εικόνα 24. Συνένωση των τμημάτων	24
Εικόνα 25. Διαδοχικές συνενώσεις τμημάτων	25
Εικόνα 26. Στιγμιότυπο LIN105	28
Εικόνα 27. Στιγμιότυπο BERLIN52	29
Εικόνα 28. Στιγμιότυπο EIL101	30
Εικόνα 29. Στιγμιότυπο A280	31
Εικόνα 30. Τοποθέτηση 1, 2 και 3 θυρών ανά πλευρά κόμβου	32
Εικόνα 31. Στιγμιότυπο CH130	33
Εικόνα 32. Στιγμιότυπο CH150	34
Εικόνα 33. Στιγμιότυπο KROA100	35
Εικόνα 34. Στιγμιότυπο KROC100	36
Εικόνα 35. Στιγμιότυπο PR76	37
Εικόνα 36. Στιγμιότυπο KROD100	38
Εικόνα 37. Στιγμιότυπο ST70	39
Εικόνα 38. Πιθανά μονοπάτια για $k = 2$ (μπλε) και $k = 3$ (πράσινο)	41
Εικόνα 39. Πιθανά k-TSP μονοπάτια από το S στο T για $k=3$ (γαλάζιο), $k=5$ (μπλε)	43
Εικόνα 40. Προβληματικές περιπτώσεις άκρων διαδρομής	44
Εικόνα 41. Όταν η εκκίνηση και ο τερματισμός βρίσκονται σε κόμβους χωρίς κοινές θύρες	44
Εικόνα 42. Οπτικοποίηση των ειδικών περιπτώσεων	45
Εικόνα 43. Τμήμα cache αρχείου για το πρόβλημα EIL101	47

Εικόνα 44. Μεγέθυνση σε επίφοβο κομμάτι της διαδρομής LIN105, $k=25$	47
Εικόνα 45. Στιγμιότυπο LIN105	49
Εικόνα 46. Στιγμιότυπο BERLIN52	50
Εικόνα 47. Στιγμιότυπο EIL101	51
Εικόνα 48. Στιγμιότυπο ST70	52
Εικόνα 49. Στιγμιότυπο PR76	53
Εικόνα 50. Στιγμιότυπο KROA100	54
Εικόνα 51. Στιγμιότυπο KROC100	55
Εικόνα 52. Στιγμιότυπο KROD100	56
Εικόνα 53. Περιστροφή του επιπέδου	57
Εικόνα 54. Υπολογισμός γωνίας περιστροφής	58
Εικόνα 55. Οι διαδρομές που αντιστοιχούν στα $G(S,1,k)$	60
Εικόνα 56. Διαδρομές Gottlieb 9 πόλεων στο παράθυρο της πόλης i	65
Εικόνα 57. Πιθανή σύνθεση διαδρομών για το Gottlieb($T,T,7$)	66
Εικόνα 58. Βελτίωση διαδρομής του test με τις μετατροπές της υπορουτίνας Gottlieb	67
Εικόνα 59. Βελτίωση διαδρομής του test2 με τις μετατροπές της υπορουτίνας Gottlieb	68
Εικόνα 60. Στο στιγμιότυπο Tsiligrirides1 οι διαδρομές των δύο υλοποιήσεων είναι ίδιες	68

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Πίνακας 1. Σύγκριση προσεγγίσεων με βέλτιστες λύσεις για το TSP	26
Πίνακας 2. Προσεγγίσεις για το k-TSP	47

1. Εισαγωγή

Το Πρόβλημα του Πλανώδιου Πωλητή (Travelling Salesperson Problem, TSP) αποτέλεσε από τον 19ο αιώνα αλγοριθμική πρόκληση. Συνοπτικά, δοθέντων ενός σημείου εκκίνησης, μίας λίστας με προορισμούς και των αποστάσεων μεταξύ τους, επιδιώκεται να βρεθεί ένας βέλτιστος τρόπος έτσι ώστε ένας πωλητής να περάσει από όλους τους προορισμούς και να επιστρέψει στο σημείο εκκίνησης. Δηλαδή η λύση του προβλήματος θα είναι ο κύκλος που θα διέρχεται από κάθε προορισμό και θα έχει το ελάχιστο μήκος.

Η δυσκολία επίλυσής του έγκειται στο γεγονός ότι η διατύπωσή του ως προβλήματος απόφασης, δηλαδή το ερώτημα *‘δοθέντος ενός μήκους L , υπάρχει κύκλος που να διέρχεται από όλους τους προορισμούς και να έχει μήκος μικρότερο του L ’*, ανήκει στην τάξη των NP-πλήρων προβλημάτων, δηλαδή ο χειρότερος χρόνος εκτέλεσης για έναν αλγόριθμο που το επιλύει αυξάνεται εκθετικά με τον αριθμό των πόλεων[1] και είναι NP-δύσκολο ακόμα και στην Ευκλείδεια περίπτωση[2, 3, 4]. Παρόλα αυτά, είναι γνωστές αρκετές υλοποιήσεις που προσεγγίζουν σε εύλογο χρονικό διάστημα τη βέλτιστη λύση ακόμα και σε εκατομμύρια πόλεις (*world tour*), με απόκλιση της τάξης μόλις του 1%[5].

Έστω $G(V,E)$ ένας γράφος, με V και E το σύνολο των κορυφών και ακμών του γραφήματος αντίστοιχα. Όταν αναφερόμαστε σε ένα στιγμιότυπο TSP θα εννοούμε το σύνολο

$$\{G(V,E), S \in V\}$$

όπου:

- V το σύνολο των προορισμών
- E το σύνολο των ακμών (αποστάσεις)
- S το σημείο εκκίνησης

Ένα τέτοιο στιγμιότυπο παρουσιάζεται στο Σχήμα 1.1.

Ο αριθμός πιθανών λύσεων που θα έπρεπε να δοκιμαστούν είναι εκθετικά αυξανόμενος καθώς οι προορισμοί αυξάνονται, το οποίο καθιστά αδύνατη την εξαντλητική αναζήτηση. Ήδη σε ένα παράδειγμα TSP 10 πόλεων, οι πιθανές διαδρομές που θα έπρεπε να συγκριθούν ώστε να επιλεγθεί η βέλτιστη είναι $9!$, δηλαδή περίπου 360 χιλιάδες.

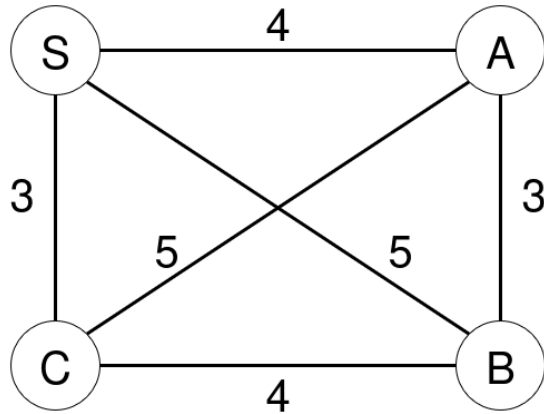
Η ιδιαίτερη φύση του προβλήματος αυτού οδήγησε σε εναλλακτικές αντιμετώπισεις του, δηλαδή μέσω ευρετικών και μέσω προσεγγιστικών αλγορίθμων. Οι πρώτοι αρχίζοντας από μία τυχαία λύση χρησιμοποιούν τακτικές ώστε να την προσαρμόσουν μέχρι να μπορέσει να γίνει αποδεκτή για την εκάστοτε περίπτωση[6, 7, 8]. Οι προσεγγιστικοί, με τους οποίους ασχολείται και η παρούσα εργασία, αποτελούν αλγορίθμους που επιχειρούν την εύρεση μίας αρκετά-καλής λύσης, της οποίας η ποιότητα μπορεί να επιβεβαιωθεί με μαθηματική θεμελίωση. Λέγοντας αρκετά-καλή θα εννοούμε μία λύση της οποίας η απόδοση, ανάλογα με το πρόβλημα, είναι το πολύ α φορές χειρότερη από την θεωρητική βέλτιστη, ή αλλιώς, για ένα πρόβλημα ελαχιστοποίησης όπως το TSP:

$$F_{appr} / F_{opt} \leq \alpha, \alpha = 1 + \varepsilon, \varepsilon > 0 \quad (1.1)$$

όπου F_{appr} το κόστος της λύσης που προκύπτει από τον προσεγγιστικό αλγόριθμο και F_{opt} το κόστος της βέλτιστης λύσης, της οποίας την τιμή δε γνωρίζουμε εκ των προτέρων[9].

Σχετικά με τους προσεγγιστικούς αλγορίθμους, ο πρώτος σημαντικός σταθμός στην ιστορία του TSP ήταν ο αλγόριθμος των Χριστοφίδη και Serdyukon[10, 11], ο οποίος επιστρέφει στη χειρότερη περίπτωση μία λύση με κόστος 1,5 φορές μεγαλύτερο από της βέλτιστης, δηλαδή έχει λόγο προσέγγισης $\alpha = 1,5$. Για τα τελευταία πενήντα χρόνια αυτό αποτελούσε και το καλύτερο πάνω όριο αυτού του προβλήματος[12, 13], μέχρι το 2020 να βρεθεί ένα ελάχιστα καλύτερο[14]. Όμως έχουν βρεθεί καλύτερες προσεγγίσεις σε πιο συγκεκριμένες εκδοχές του προβλήματος, η καθεμία κατάλληλα προσαρμοσμένη στον τύπο του γραφήματος στον οποίο ειδικεύεται (2, 3 ή n διαστάσεις, συμμετρικές αποστάσεις μεταξύ κόμβων, κόμβοι με χρονικά παράθυρα, κόμβοι με βαθμολογίες κ.ά.). Ενδεικτικά αναφέρουμε το Ευκλείδειο

TSP, που η παρούσα εργασία πραγματεύεται[15, 16] και το TSP σε επίπεδα γραφήματα[17]. Αυτές οι δύο εκδοχές στις οποίες αποδειχτεί η ύπαρξη PTAS, δηλαδή προσεγγιστικού αλγόριθμου για τον οποίο ισχύει η εξίσωση 1.1.



Εκκίνηση: S

Προορισμοί: [A,B,C]

Αποστάσεις:

$[c_{SA} = 4, c_{SB} = 5, c_{SC} = 3,$
 $c_{AB} = 3, c_{AC} = 5, c_{BC} = 4]$

ΣΧΗΜΑ 1.1: Απλό παράδειγμα TSP

Το TSP αποτέλεσε εφαλτήριο για την εμφάνιση και περισσότερων προβλημάτων, τα οποία προσθέτουν περιορισμούς και συνθήκες στην εύκολα κατανοητή αρχική μορφή του. Μία γνωστή παραλλαγή αυτού του τύπου αποτελεί το Πρόβλημα του Πλανώδιου Πωλητή με k Στάσεις (k-TSP), στο οποίο ζητάμε λύσεις ελάχιστου μήκους που να διέρχονται τουλάχιστον από k προορισμούς ($k \leq |P|$). Επομένως όταν αναφερόμαστε σε ένα στιγμιότυπο k-TSP θα εννοούμε το σύνολο

$$\{G(V, E), S \in V, k \leq |V|\}$$

όπου:

- V το σύνολο προορισμών
- E το σύνολο ακμών (αποστάσεις)
- S το σημείο εκκίνησης
- k ο επιθυμητός αριθμός προορισμών

Στην παρούσα εργασία θα ασχοληθούμε με υλοποιήσεις για τα παραπάνω δύο προβλήματα, και συγκεκριμένα θα υλοποιήσουμε

1. τον προσεγγιστικό αλγόριθμο του Arora για το TSP[15]
2. μία επέκταση του αλγορίθμου του Arora για το k-TSP
3. μία παραλλαγή της υλοποίησης του Arora για το k-TSP[21]

Η παραλλαγή της υλοποίησης του αλγορίθμου k-TSP θα έχει την επιλογή για προσθήκη σημείου λήξης, επιστρέφοντας ως αποτέλεσμα ένα μονοπάτι ανάμεσα σε δύο άκρα και όχι κύκλο, και κάποιες βελτιώσεις βασισμένες στη μελέτη πάνω στο πρόβλημα του Προσανατολισμού (Orienteering) από τους Gottlieb et al[18]. Αναφέρουμε πληροφοριακά ότι στο πρόβλημα του Προσανατολισμού, ο περιπλανώμενος πωλητής έχει περιορισμούς όσον αφορά τα καύσιμα ή τον χρόνο και σε κάθε προορισμό

του αντιστοιχεί μία βαθμολογία, την οποία λαμβάνουμε υπ' όψιν για την επιλογή της βέλτιστης διαδρομής. Για περισσότερες πληροφορίες σε αυτή την κατηγορία προβλημάτων παραπέμπουμε τον αναγνώστη στην αρχική δουλειά του Balas[19] και Golden et al.[20], καθώς και στην έρευνα των Toth και Vigo[21] σχετικά με το παρεμφερές πρόβλημα της Δρομολόγησης Οχημάτων (Vehicle Routing).

2. Σχετική Έρευνα

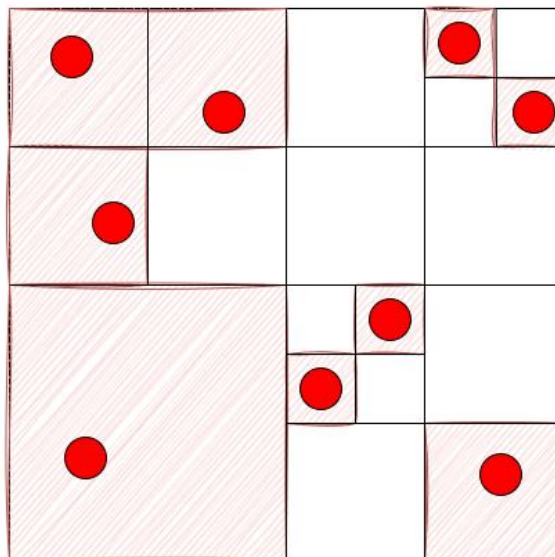
Όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο, οι υλοποιήσεις των αλγορίθμων που θα παρουσιαστούν βασίζονται στα αποτελέσματα της έρευνας του Arora[15].

2.1. Αλγόριθμος του Arora

Ο αλγόριθμος του Sanjeev Arora παρουσιάστηκε πρώτη φορά στο άρθρο με τίτλο “Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems”. Είναι σχεδιασμένος για προβλήματα στα οποία οι αποστάσεις των προορισμών ισούνται με την ευκλείδεια απόστασή τους και είναι πάντα συμμετρικές, δηλαδή

$$c_{AB} = c_{BA} \forall A, B \in V \quad (2.1)$$

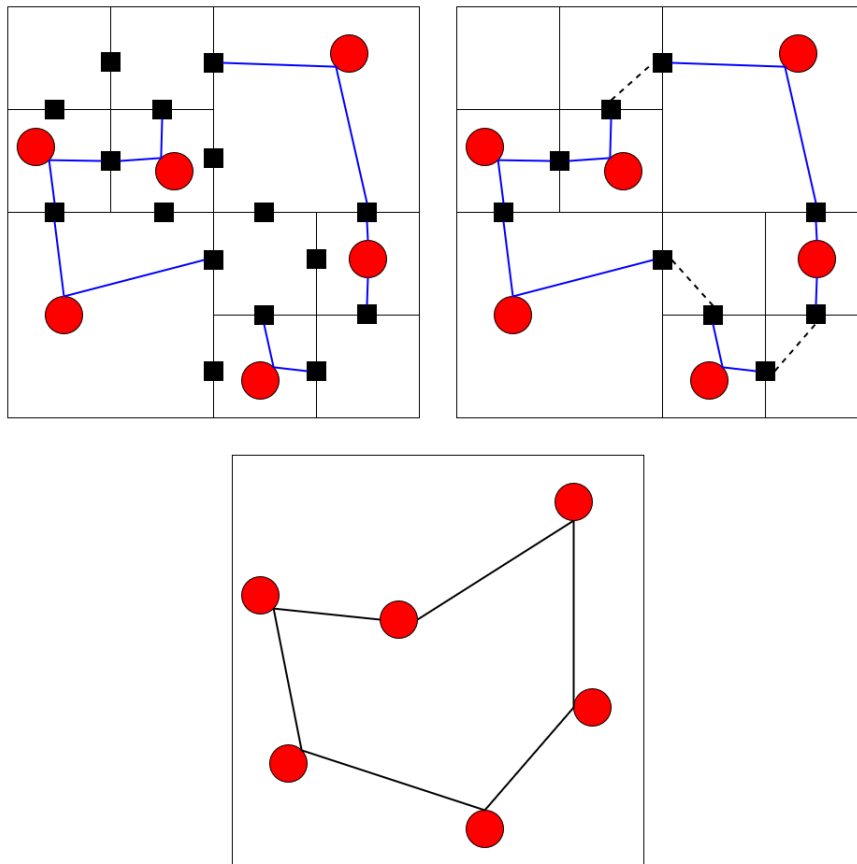
Η βασική ιδέα έγκειται στην αντιμετώπιση του TSP μέσω μίας γεωμετρικής προσέγγισης, χωρίζοντας το αρχικό πρόβλημα σε μικρότερα, επιλύοντάς τα και έπειτα συνθέτοντας τις λύσεις μέσω δυναμικού προγραμματισμού. Αξίζει να αναφερθεί ότι μία τέτοια αντιμετώπιση ήταν ήδη διαδεδομένη στα προβλήματα του TSP[22].



ΕΙΚΟΝΑ 2.1.1: Τα τελικά τετράγωνα σκιαγραφημένα

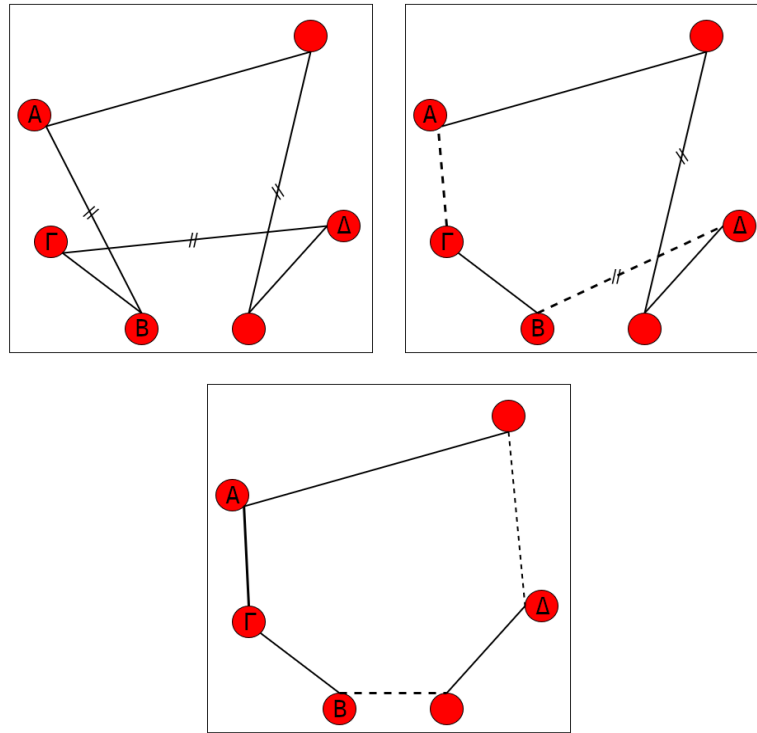
Αρχικά οι προορισμοί οριοθετούνται μέσα σε ένα ελάχιστου εμβαδού πιθανό τετράγωνο, το οποίο διαμερίζουμε σε τέσσερα ισεμβαδικά μικρότερα τετράγωνα. Κάθε ένα από αυτά διαμερίζεται με τη σειρά του σε άλλα τέσσερα τετράγωνα και η διαδικασία συνεχίζει αναδρομικά μέχρι κάθε προορισμός να ανήκει σε ένα ακριβώς τετράγωνο και κάθε τετράγωνο να περιέχει το πολύ έναν προορισμό. Τα τελικά αυτά τετράγωνα θα τα αναφέρουμε ως *οριστικοποιημένα* (Εικόνα 2.1.1).

Σε κάθε πλευρά των οριστικοποιημένων τετραγώνων ορίζουμε έναν αριθμό από θύρες (*portals*), οι οποίες θα αποτελούν τα μοναδικά σημεία μέσω των οποίων θα μπορεί ένα μονοπάτι να εισέρχεται σε αυτά. Όπως βλέπουμε και στην Εικόνα 2.1.2, συνδέοντας τον προορισμό με κάποιο ζευγάρι θυρών ή συνδέοντας απευθείας κάποιο ζευγάρι θυρών στην περίπτωση που το τετράγωνο δεν περιέχει προορισμό, δημιουργούμε τμήματα διαδρομών. Ενώνοντας τμήματα που έχουν ένα κοινό άκρο, δηλαδή τμήματα που διέρχονται από την ίδια θύρα, μπορούμε να καταλήξουμε σε μία κυκλική διαδρομή. Σε περίπτωση που η λύση έχει κύκλους, δηλαδή μέρη της διαδρομής που τέμνονται μεταξύ τους, τους διασπάμε για να καταλήξουμε σε μία ζητούμενη λύση.



ΕΙΚΟΝΑ 2.1.2: Διαδικασία δημιουργίας μίας διαδρομής Arora

Όπως φαίνεται και στην *Εικόνα 2.1.3*, η διάσπαση αυτή επιτυγχάνεται βρίσκοντας το ζεύγος τεμνόμενων ακμών και αλλάζοντας κατάλληλα τα άκρα τους. Συγκεκριμένα, αν AB και $\Gamma\Delta$ είναι οι δύο ακμές που τέμνονται, επιτυγχάνουμε τη διάσπαση του κύκλου αν αντικαταστήσουμε αυτές τις δύο ακμές από δύο άλλες, εκ των οποίων η πρώτη θα έχει ως άκρα τις αρχές των αρχικών ακμών ($A\Gamma$) και η δεύτερη τα τέλη τους ($B\Delta$). Καθώς αυτό μπορεί να δημιουργήσει νέο ζευγάρι τεμνόμενων ακμών σε επόμενο κομμάτι του μονοπατιού, επαναλαμβάνουμε αυτή τη διαδικασία μέχρι να μην υπάρχει κανένας κύκλος.



ΕΙΚΟΝΑ 2.1.3: Σταδιακή διάσπαση κύκλων

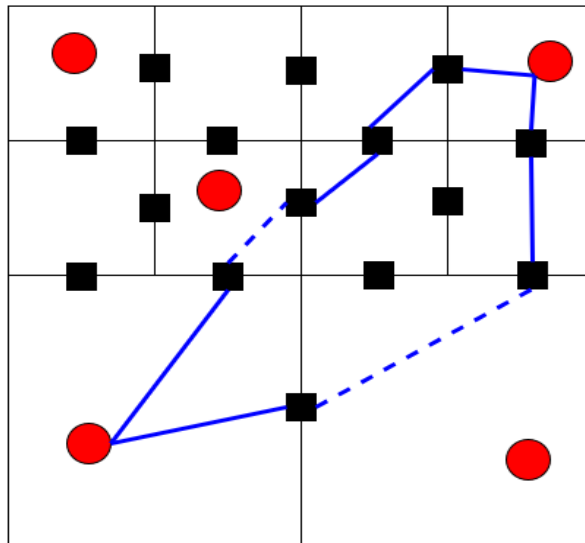
Ονομάζοντας f_{arora} το κόστος της λύση που επιστρέφει ο αλγόριθμος και t τον χρόνο υπολογισμού που χρειάζεται, αποδεικνύεται ότι

$$\frac{f_{arora}}{f_{opt}} = 1 + \frac{1}{c}, \quad c > 0, \quad t = O(n \log(n)^{O(c)}) \quad (2.2)$$

Για την επίλυση του k -TSP εφαρμόζεται και πάλι η ιδέα του σταδιακού χωρισμού ενός αρχικού τετραγώνου σε τετράδες. Η βασική διαφορά έγκειται στον τρόπο που επιλέγουμε και υπολογίζουμε τα τμήματα διαδρομών σε αυτά τα τετράγωνα. Καθώς οι λύσεις δεν είναι αναγκαίο να περνάνε από κάθε προορισμό, εφόσον το k είναι μικρότερο από το συνολικό πλήθος των προορισμών, στα οριστικοποιημένα τετράγωνα κρατάμε δύο πιθανά τμήματα για κάθε ζευγάρι θυρών:

- ένα που διέρχεται από τον προορισμό, εφόσον το τετράγωνο περιέχει έναν
- ένα που δεν διέρχεται από προορισμό και απλά ενώνει τις δύο θύρες

Με την εισαγωγή των δεύτερων τμημάτων μπορούμε με κατάλληλες επιλογές να επιτύχουμε κάθε πιθανό αριθμό προορισμών στην τελική λύση. Αν ο ζητούμενος αριθμός προορισμών είναι k , αρκεί να επιλέξουμε τυχαία k τμήματα τα οποία να περιέχουν έναν προορισμό και έπειτα να επιλέξουμε μόνο τα τμήματα που ενώνουν απευθείας τις θύρες (Εικόνα 2.1.4).

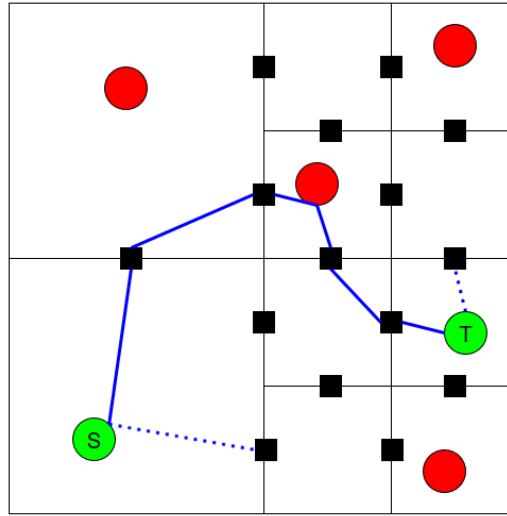


ΕΙΚΟΝΑ 2.1.4: Επιλογή διαδρομών χωρίς προορισμό για επίτευξη k-μονοπατιού, k=2

Η πολυπλοκότητα είναι ανάλογη του απλού TSP, με τον χρόνο υπολογισμού να αυξάνεται πολλαπλασιαστικά κατά k, δηλαδή

$$t = O(nk \log(n)^{O(c)}) \quad (2.3)$$

Με κατάλληλες προσαρμογές, τις οποίες θα δούμε αναλυτικά στην υλοποίηση με ψευδοκώδικα, μπορούμε να χειριστούμε και την έννοια των άκρων στις k-TSP διαδρομές, δηλαδή αντί για k-TSP μονοπάτια να υπολογίσουμε μη κυκλικά μονοπάτια με συγκεκριμένους προορισμούς για αρχή και τέλος τους. Εφαρμόζοντας τον επιπλέον περιορισμό ότι πρέπει πάντα να επιλέγεται ένα τμήμα που να διέρχεται από τον προορισμό που θεωρούμε αρχή και ένα τμήμα που να διέρχεται από τον προορισμό που θεωρούμε τέλος, η αντιμετώπιση θα είναι ίδια με αυτήν που περιγράφηκε παραπάνω για το k-TSP (Εικόνα 2.1.5).



ΕΙΚΟΝΑ 2.1.5: k-διαδρομή με άκρα S-T, k=3. Με διακεκομμένες φαίνονται τα κομμάτια που δε θα προσμετρηθούν στην τελική λύση

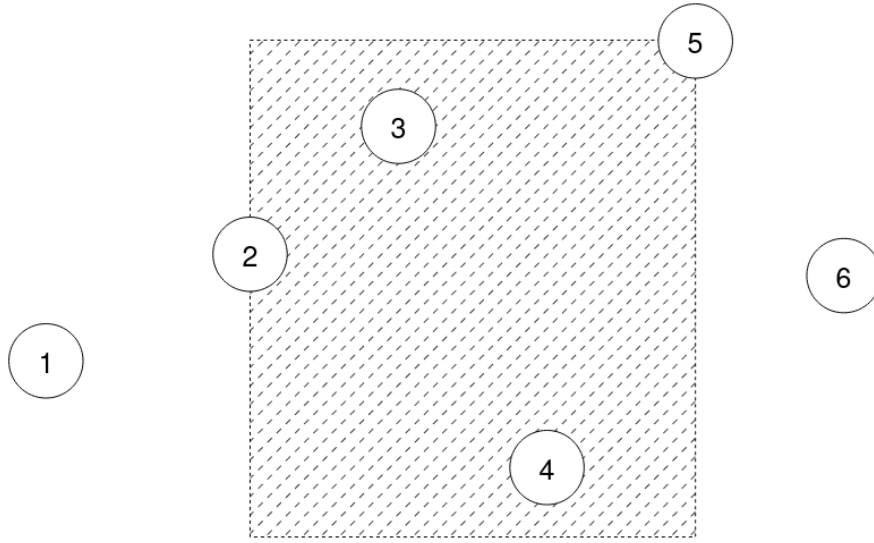
2.2 Βελτίωση k-TSP

Ο αλγόριθμος Gottlieb παρουσιάστηκε πρώτη φορά στο άρθρο “Faster algorithms for orienteering and k-TSP”[18] και επιλύει το πρόβλημα του Προσανατολισμού χρησιμοποιώντας τον αλγόριθμο του Arora για το k-TSP με άκρα ως υπορουτίνα. Παρατηρήθηκε ότι ο τρόπος που υλοποιείται η υπορουτίνα προσφέρει καλύτερα αποτελέσματα σε κάποια από τα στιγμιότυπα, στα οποία προηγουμένως εφαρμόσαμε την υλοποίηση που ήταν βασισμένη μόνο στο άρθρο του Arora. Την παραθέτουμε συνοπτικά.

Αρχικά τοποθετούμε τους προορισμούς σε σειρά αύξουσας τετμημένης και ορίζουμε ως παράθυρο (*window*) δύο προορισμών i και j το σύνολο $W_{i,j}$ όλων των προορισμών που περιέχονται μεταξύ τους (Εικόνα 2.2.1). Αναθέτουμε σε κάθε προορισμό τον δείκτη που δείχνει την αύξουσα θέση του στο σύνολο των προορισμών και χρησιμοποιούμε αυτόν τον δείκτη σημειωμένο με περισπωμένη για να αναπαραστήσουμε τον προορισμό. Δηλαδή ο i -οστός προορισμός θα γράφεται ως $\tilde{i}$.

Έστω S και T τα σημεία εκκίνησης και λήξης αντίστοιχα που μας έχουν δοθεί. Ορίζουμε $f_{gottlieb}(\tilde{i}, \tilde{j}, k)$, $\tilde{j} \in W_{S, \tilde{i}}$ το μονοπάτι ελάχιστου μήκους στο παράθυρο $W_{S, \tilde{i}}$ που διέρχεται από k προορισμούς, έχοντας ως σημείο εκκίνησης το S και σημείο κατάληξης το $\tilde{j}$. Θα είναι η λύση Gottlieb για τις τρεις αυτές παραμέτρους (*παράθυρο, σημείο κατάληξης, αριθμός προορισμών*).

Ορίζουμε $f_{arora}(\tilde{i}, \tilde{j}, \tilde{x}, \tilde{y}, k)$ την λύση που επιστρέφει ο k-TSP αλγόριθμος Arora με άκρα $\tilde{x}, \tilde{y}$ που διέρχεται από k προορισμούς στο παράθυρο $W_{\tilde{i}, \tilde{j}}$.



ΕΙΚΟΝΑ 2.2.1: Ταξινομημένοι προορισμοί και παράθυρο των προορισμών 2 και 5 ($W_{2,5}$)

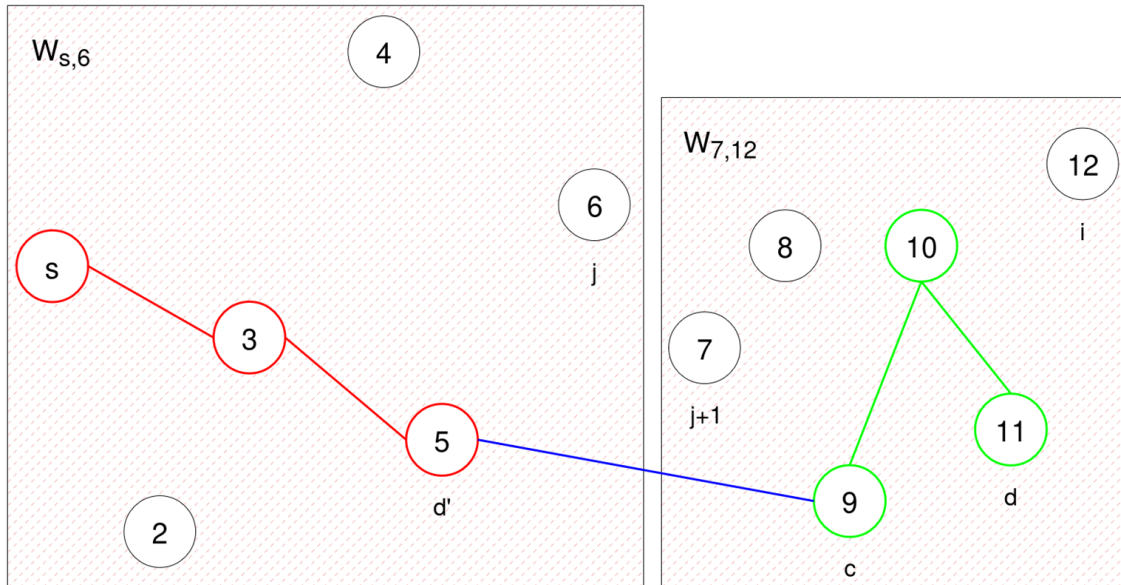
Με βάση τα παραπάνω, ως βέλτιστο μονοπάτι k προορισμών με σημείο εκκίνησης το S και κατάληξης ένα οποιοδήποτε $\tilde{d}$ στο παράθυρο $W_{S,\tilde{d}}$, ο αλγόριθμος επιλέγει την

$$f_{gottlieb}(\tilde{d}, k) = \min_{j \leq \tilde{d}, \tilde{d}' \leq j, j < \tilde{c} \leq \tilde{d}, k' < k} \{f_{gottlieb}(j, \tilde{d}', k') + |\tilde{d}' - \tilde{c}| + f_{arora}(\widetilde{j+1}, \tilde{c}, \tilde{d}, k - k')\} \quad (2.4)$$

Δηλαδή κάθε φορά σταθεροποιεί μία ήδη υπολογισμένη λύση η οποία διέρχεται από λιγότερους προορισμούς από τους επιθυμητούς και καταλήγει σε έναν προορισμό μικρότερο από τον επιθυμητό, και την επεκτείνει με μία k -TSP λύση στους προορισμούς που περισσεύουν, κρατώντας στο τέλος τον καλύτερο συνδυασμό (Εικόνα 2.2.2). Μετά το τέλος της αλγοριθμικής διαδικασίας, οι λύσεις του k -TSP για τις διάφορες τιμές της παραμέτρου k θα είναι οι

$$f_{gottlieb}(\tilde{n}, \tilde{t}, k), \quad k \leq |P| \quad (2.5)$$

Στα κεφάλαια που ακολουθούν θα δούμε σε μεγαλύτερο βάθος όλους τους παραπάνω αλγορίθμους, θα παρουσιαστούν κομμάτια ψευδοκώδικα που αντιστοιχούν στην υλοποίηση σε Java που αποτελεί το κύριο μέλημα της εργασίας, θα συζητηθούν οι περιορισμοί, τα εμπόδια και οι παρρεκλίσεις που κρίθηκαν απαραίτητες έτσι ώστε να μεταφερθούν τα θεωρητικά ευρήματα σε πρακτικές λύσεις καθώς και θα παρουσιάζονται στιγμιότυπα από τις εκτελέσεις του κώδικα στα πραγματικά αυτά παραδείγματα.



ΕΙΚΟΝΑ 2.2.2: Πιθανή διαδρομή Gottlieb ($i=12$, $d=11$, $k=6$). Ο προορισμός $j=6$ ορίζει το μέρος της διαδρομής που θα επιλεγεί από τις ήδη υπολογισμένες και το μέρος στο οποίο θα εφαρμοστεί ο αλγόριθμος Arora

Είναι σημαντικό να αναφερθεί ότι οι χρόνοι εκτέλεσης δυσχεραίνουν όσο γενικεύουμε το πρόβλημα, έχοντας πολύ καλές αντιμετωπίσεις στα προβλήματα TSP και οριακά διαχειρίσιμες στην τελευταία προσαρμογή του k -TSP. Αυτό δεν είναι κάτι που μας αποθάρρυνε, καθώς αυτή ήταν και μία από τις παρατηρήσεις του Arora στο άρθρο του, δηλαδή ότι η μεταφορά των θεωρητικών μοντέλων στην πράξη αναμένεται να έχει αποτρεπτική βραδύτητα σε σύγκριση με άλλους υπάρχοντες αλγορίθμους (π.χ. Lin-Kernighan[6]), γεγονός που όμως είναι λογικό κρίνοντας το πλήθος των πράξεων των θεωρητικών διατυπώσεων. Παρόλα αυτά θα αποτελέσει μία βάση και οδηγό για να προκύψουν ιδέες για τα σημεία στα οποία ίσως θα μπορούσε να προσαρμοστεί η θεωρία και με άλλες τακτικές, ώστε να γίνει μία ανταλλαγή ποιότητας λύσης με χρόνο εκτέλεσης (π.χ. προσαρμογή και ευριστικών τεχνικών). Μάλιστα ο ίδιος ο Arora παρατηρεί κάποιες τέτοιες βελτιώσεις άλλων ερευνητών που προσεγγίζουν από παρόμοια οπτική το πρόβλημα π.χ. στους χρόνους εκτέλεσης σε επίπεδα μεγαλύτερων διαστάσεων[23] και στην αποτυχαιοποίηση του αλγορίθμου[24]. Η προαναφερθείσα τυχαιοποίηση (Randomization) θα παρουσιαστεί κατά τη μετέπειτα υλοποίηση.

3. Ο Αλγόριθμος Arora για το Πρόβλημα TSP

Για όλη την υπόλοιπη εργασία θα αναφερόμαστε στους προορισμούς ως πόλεις, για συνοχή με την αντίστοιχη ονομασία στον κώδικα (*towns*). Επιπλέον με n θα αναφερόμαστε στο πλήθος των πόλεων.

3.1 Φάση Προεπεξεργασίας

Στην πρώτη φάση του αλγορίθμου θα κανονικοποιήσουμε τις συντεταγμένες των πόλεων. Χρησιμοποιώντας καρτεσιανούς μετασχηματισμούς θα μεταφέρουμε τις πόλεις κατάλληλα ώστε οι συντεταγμένες να είναι μη-αρνητικές και έπειτα θα τις προβάλλουμε σε ένα πλέγμα, ώστε να επιβάλλουμε μία κοινή ελάχιστη απόσταση μεταξύ τους. Αυτό θα βοηθήσει στην επόμενη φάση, όπου θα πρέπει να οριοθετήσουμε την κάθε πόλη μέσα σε ένα οριστικοποιημένο τετράγωνο.

3.1.1 Κλοιός

Ο κλοιός είναι ένα από τα τετράγωνα ελάχιστου μήκους πλευράς μέσα στο οποίο περιέχονται όλες οι πόλεις (Εικόνα 3.1.1.1). Για τον ορισμό ενός κλοιού μας αρκεί το μήκος της πλευράς του και οι συντεταγμένες της κάτω αριστερά του γωνίας.

Έστω $T(x_i, y_i) \forall i \in \{0, \dots, n-1\}$ οι συντεταγμένες των πόλεων. Έστω

$$\begin{cases} d_x = \max(x_i - x_j) \\ d_y = \max(y_i - y_j) \end{cases} \forall i, j \in \{0, \dots, n-1\}, i \neq j \quad (3.1)$$

οι μέγιστες αποστάσεις στον οριζόντιο και κάθετο άξονα και

$$\begin{cases} T_L(x_L, y_L) \mid x_L \leq x_i \\ T_D(x_D, y_D) \mid y_D \leq y_i \end{cases} \forall i \in \{0, \dots, n-1\} \quad (3.2)$$

οι συντεταγμένες της πιο αριστερής και της πιο χαμηλής πόλης αντίστοιχα, δηλαδή οι συντεταγμένες των πόλεων με την μικρότερη τετμημένη και την μικρότερη τεταγμένη.

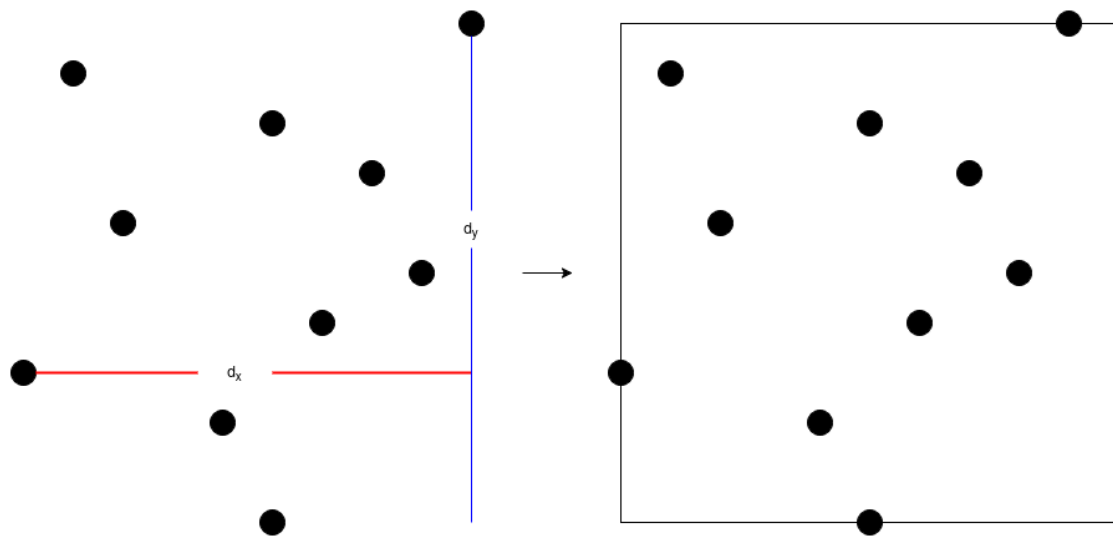
Τότε το μήκος της πλευράς ενός πιθανού κλοιού και οι συντεταγμένες της κάτω αριστερής γωνίας του δίνονται από τους τύπους

$$side = \max(d_x, d_y), \text{ corner} = (x_L, y_D) \quad (3.3)$$

```
void createBoundingBox(Towns):
```

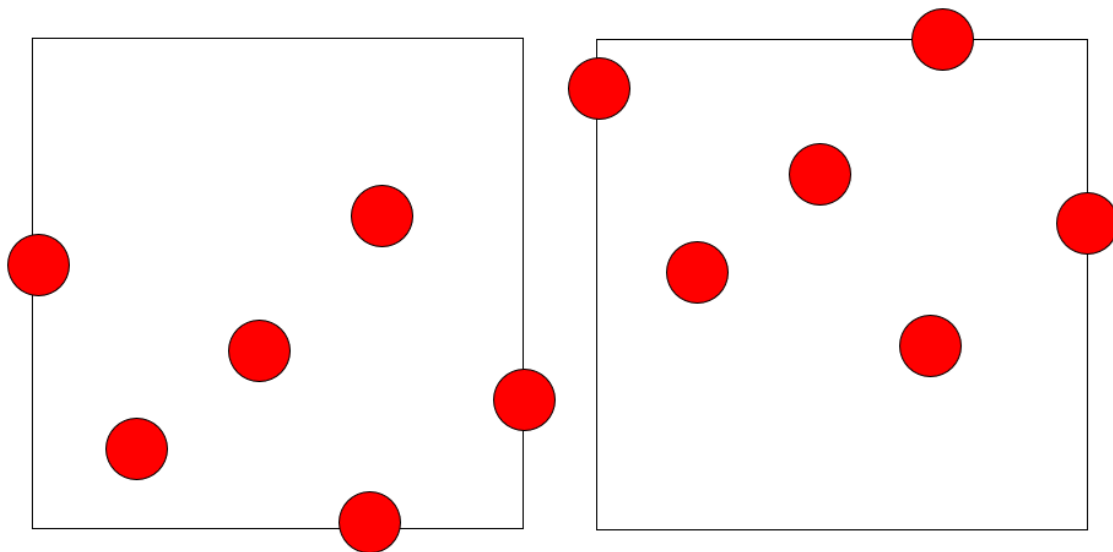
```
  for town in Towns:
    find x_min, x_max, y_min, y_max
    d_x := x_max - x_min
    d_y := y_max - y_min
    side := max(d_x, d_y)
    corner := (x_min, y_min)
```

Όπως βλέπουμε παραπάνω, στην πράξη αρκεί να διατρέξουμε όλες τις πόλεις και να κρατήσουμε τις ελάχιστες και τις μέγιστες τετμημένες και τεταγμένες. Η κάτω γωνία του κλοιού θα είναι το ζεύγος της ελάχιστης τετμημένης και τεταγμένης και η πλευρά του θα είναι η μεγαλύτερη από τις διαφορές των τετμημένων και των τεταγμένων (Εικόνα 3.1.1.1).

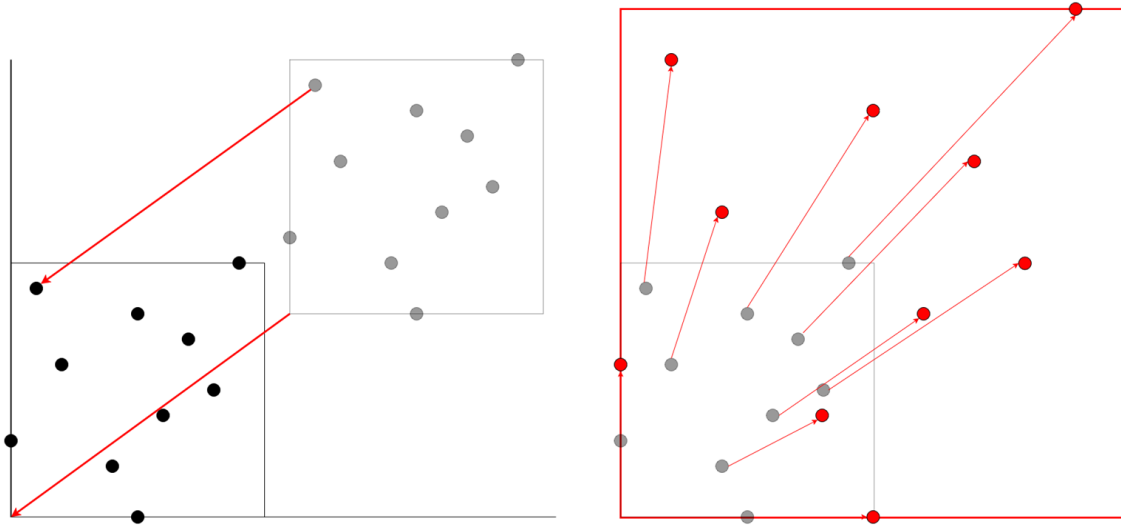


ΕΙΚΟΝΑ 3.1.1.1: Ένας πιθανός κλοιός

Προφανώς υπάρχουν παραπάνω από ένας κλοιοί, οι οποίοι αποτελούν οριζόντιες ή κάθετες μετατοπίσεις αυτού που επιλέξαμε παραπάνω (Εικόνα 3.1.1.2).



ΕΙΚΟΝΑ 3.1.1.2: Πιθανοί κλοιοί με επιλογή κάτω αριστερά και πάνω αριστερά γωνίας



ΕΙΚΟΝΑ 3.1.1.3: Ο μετασχηματισμένος κλοιός

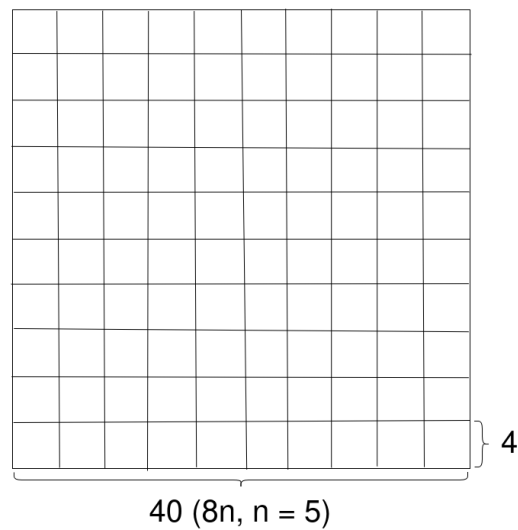
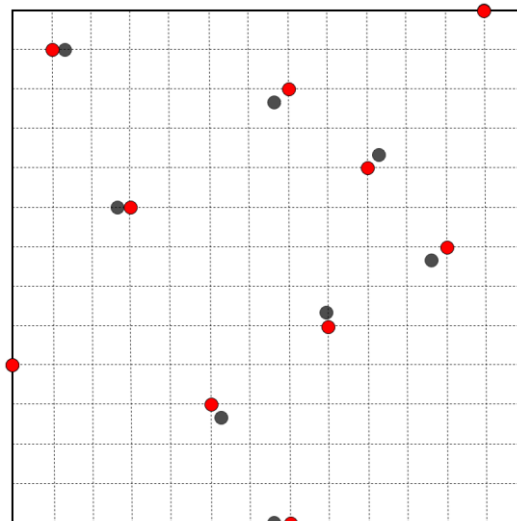
Μετά τους παραπάνω υπολογισμούς

- μετακινούμε τον κλοιό στην αρχή των αξόνων
- μετασχηματίζουμε τις συντεταγμένες του κλοιού και των πόλεων (Εικόνα 3.1.1.3)

Για τον μετασχηματισμό θα χρησιμοποιήσουμε το λεγόμενο πλέγμα. Με τον όρο πλέγμα θα εννοούμε ένα σύνολο από οριζόντιες και κάθετες παράλληλες γραμμές, με σταθερή απόσταση μεταξύ τους (Εικόνα 3.1.1.4). Το πόσο πυκνές θα είναι αυτές οι γραμμές θα καθορίζεται από μία παράμετρο $e \in (0,1]$, την οποία ονομάζουμε συντελεστή ακριβείας του πλέγματος.

Σκοπός μας είναι να ορίσουμε ένα κατάλληλο πλέγμα, το οποίο θα μπορεί να μετασχηματιστεί περαιτέρω για να εξασφαλίσει μία αρκετά μεγάλη απόσταση μεταξύ δύο διακριτών πόλεων, κάτι το οποίο θα είναι χρήσιμο στην επόμενη φάση του αλγορίθμου.

Αρχικά ορίζουμε ένα πλέγμα με απόσταση γραμμών $e \frac{side}{2n}$ στον αρχικό κλοιό και προβάλλουμε κάθε πόλη στην πιο κοντινή τομή του πλέγματος. Η συγκεκριμένη επιλογή διασφαλίζει τουλάχιστον $4n^2$ τομές για τις n πόλεις, πλήθος αρκετά μεγάλο για να μην είναι συχνό φαινόμενο δύο πόλεις να αντιστοιχούν στην ίδια τομή. Προβάλλουμε κάθε πόλη στην κοντινότερη τομή του πλέγματος και έπειτα μεγενθύνοντας με λόγο μεγέθυνσης $\frac{8n}{e \cdot side}$, οι συντεταγμένες των προβολών γίνονται πάντα ακέραιοι αριθμοί και μάλιστα η ελάχιστη απόσταση μεταξύ δύο τυχαίων θα είναι πάντα τουλάχιστον 4 μονάδες μήκους (Εικόνα 3.1.1.5).

ΕΙΚΟΝΑ 3.1.1.4: Τοποθέτηση πλέγματος σε κλοιό ($e=1$) για $n=5$ πόλεις

ΕΙΚΟΝΑ 3.1.1.5: Οι αρχικές πόλεις και οι προβολές τους στο πλέγμα

Αυτή η διαδικασία περιγράφεται στο παρακάτω κομμάτι ψευδοκώδικα. Πλέον όταν μιλάμε για πόλεις θα αναφερόμαστε σε αυτές ακριβώς τις μετασχηματισμένες προβολές. Στην επόμενη φάση του αλγορίθμου σχηματίζουμε το τετραδικό δέντρο.

```
void moveBox(Towns, boundingBox):
    for town in Towns:
        town.x := town.x - boundingBox.corner.x
        town.y := town.y - boundingBox.corner.y

void snapToGridAndScale(Towns, boundingBox):
    var gridUnit = (e * boundingBox.side) / (2 * n)
    var scalingFactor = (8 * n) / (e * boundingBox.side)
```

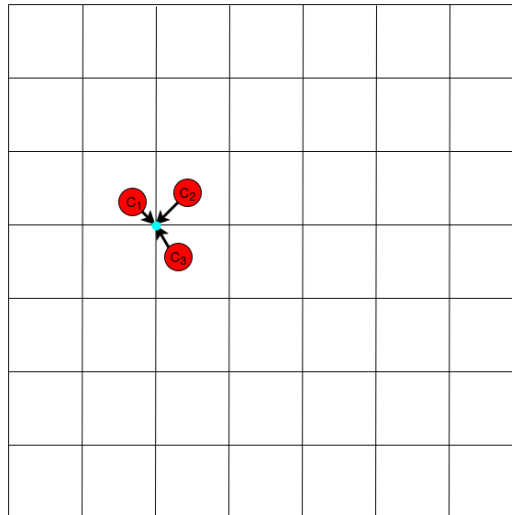
```

for town in Towns:
    town.x := x of nearest gridLine
    town.y := y of nearest gridLine
    town.x := town.x * scalingFactor
    town.y := town.y * scalingFactor

```

3.1.2 Συστάδες πόλεων

Κατά το καρφίτσωμα των πόλεων στο πλέγμα είναι πιθανό δύο ή παραπάνω πόλεις να αντιστοιχούν στην ίδια τομή (Εικόνα 3.1.2.1). Αυτές οι πόλεις θα λέμε ότι συνιστούν μία συστάδα. Αν και τα μικρότερα ε, δηλαδή τα πιο πυκνά πλέγματα, ελαχιστοποιούν αυτή την πιθανότητα, δε γίνεται με κάποιον τρόπο να αποφευχθεί καθολικά. Επομένως σε αυτή την περίπτωση θα κρατάμε την προβολή μόνο μίας πόλης ως αντιπρόσωπο όλων των υπόλοιπων πόλεων της ίδιας τομής και μόνο αυτή θα λαμβάνουμε υπ' όψιν στους επόμενους υπολογισμούς. Έστω ότι αυτή είναι η πόλη C_1 . Όταν βρεθεί η τελική TSP διαδρομή, η θέση της C_1 θα αντικατασταθεί από μία αλληλουχία όλων των πόλεων της συστάδας της σε μία βέλτιστη σειρά (Εικόνα 3.1.2.2). Αυτή η βέλτιστη σειρά μπορεί να υπολογισθεί εξαντλητικά, καθώς το μέγεθος των συστάδων θα είναι συνήθως μικρό.



ΕΙΚΟΝΑ 3.1.2.1: Προβολή πόλεων στην ίδια τομή

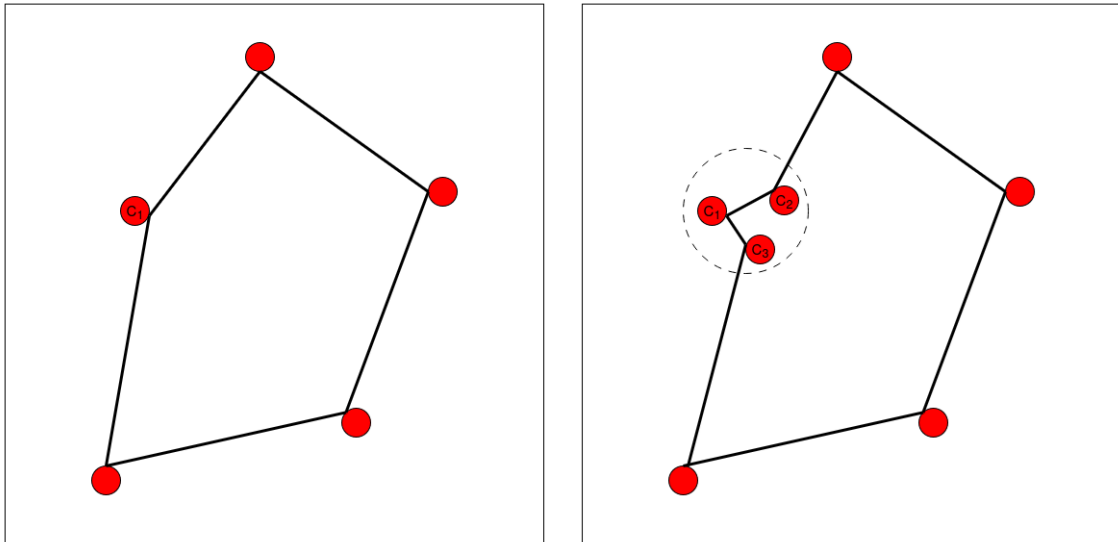
3.2 Φάση Τετραδικού Δέντρου

3.2.1 Τετραδικοί Κόμβοι

Μετά την προβολή των πόλεων στο πλέγμα ορίζουμε διακριτές περιοχές του μετασχηματισμένου κλοιού οι οποίες περιέχουν το πολύ μία πόλη. Για να το επιτύχουμε διαμερίζουμε τον κλοιό, στον οποίο πλέον θα αναφερόμαστε ως τετραδικό κόμβο επιπέδου-0, σε 4 ισεμβαδικά τετράγωνα. Αυτά τα τετράγωνα θα είναι οι τετραδικοί κόμβοι επιπέδου-1. Κάθε κόμβος διαιρείται περαιτέρω και αναδρομικά, μέχρι να υπάρξει μία πιθανή επιλογή τετραδικών κόμβων οποιουδήποτε επιπέδου έτσι ώστε

$$\forall i \in 0, \dots, n-1 \exists quadNode: \begin{cases} T_i \in quadNode \\ T_j \notin quadNode \end{cases} \forall j \neq i \quad (3.4)$$

όπου T_i η i -οστή πόλη. Δηλαδή επιζητούμε για κάθε πόλη να υπάρχει ακριβώς ένας τετραδικός κόμβος που να περιέχει μόνο αυτή την πόλη.



ΕΙΚΟΝΑ 3.1.2.2: Αντικατάσταση πόλης-αντιπροσώπου από συστάδα

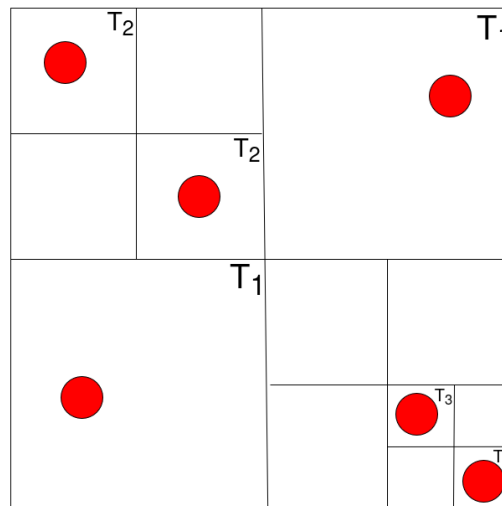
Εφεξής, θα χρησιμοποιούμε τον όρο κόμβος για τους τετραδικούς κόμβους. Θα χρησιμοποιήσουμε 3 υπομεθόδους.

1. Η *assignPointsToNode*, δοθείσας μίας πόλης και ενός κόμβου, αναθέτει την πόλη σε ένα από τα 4 παιδιά του κόμβου βάσει της θέσης της.
2. Η *finaliseNodePointsOrDivide* δοθέντος ενός κόμβου είτε οριστικοποιεί τον κόμβο είτε τον διαμερίζει περαιτέρω στα 4 παιδιά του. Η οριστικοποίηση θεωρείται αποδεκτή όταν στον κόμβο δεν έχει ανατεθεί πάνω από 1 πόλη.
3. Η *updateCornersAndPortals* αναλαμβάνει την οριστικοποίηση. Συγκεκριμένα αποθηκεύει τον κόμβο αφότου υπολογίσει τις θύρες του βάσει της παραμέτρου *portalsPerNodeSide*.

Ο αλγόριθμος που διαμερίζει το σύνολο των κόμβων είναι ο ακόλουθος:

1. Θέσε ως τρέχοντα κόμβο το επίπεδο-0 (Root).
2. Διαμέρισε τον τρέχοντα κόμβο στα 4 παιδιά του.
3. Ανάθεσε πόλεις σε κάθε παιδί.
 - 3.1. Αν ένα παιδί έχει 0 ή 1 πόλεις, υπολόγισε τις θύρες του.
 - 3.2. Αν ένα παιδί έχει 2 ή παραπάνω πόλεις, θέσε ως τρέχοντα κόμβο το παιδί και πήγαινε στο βήμα 2.

Μετά την ολοκλήρωση του αλγορίθμου περιμένουμε έναν διαμερισμό του Κλοιού, όπως αυτόν που φαίνεται στην Εικόνα 3.2.1.1.



ΕΙΚΟΝΑ 3.2.1.1: Οι τελικοί κόμβοι, σε κατάλληλο βάθος επιπέδου ώστε να περιέχουν το πολύ 1 πόλη

Στο παρακάτω κομμάτι ψευδοκώδικα υλοποιούνται ακριβώς τα βήματα 1 έως 3. Συγκεκριμένα, η αναδρομική διαδικασία αρχίζει δίνοντας στην `divideQuadLeaf` τον κόμβο επιπέδου-0 (`root`), δηλαδή τον κόμβο που περιέχει όλες τις πόλεις. Για κάθε πόλη του `root` ελέγχουμε σε ποιο από τα τέσσερα παιδιά του ανήκει και καταφέρνουμε έτσι έναν πρώτο διαχωρισμό των πόλεων σε κόμβους επιπέδου-1.

Όταν ολοκληρωθεί αυτή η διαδικασία ελέγχουμε αν κάποιος κόμβος είναι έτοιμος να οριστικοποιηθεί, δηλαδή αν έχει το πολύ μία πόλη. Σε αυτή την περίπτωση μπορούμε να αποθηκεύσουμε τον κόμβο και να υπολογίσουμε τις θύρες του. Αντίθετα σε κάθε μη-οριστικοποιημένο κόμβο θα επαναλάβουμε την `divideQuadLeaf` χωρίζοντας περαιτέρω τις τουλάχιστον δύο πόλεις στα τέσσερα παιδιά του και θα αντιμετωπίσουμε καθένα από αυτά με αντίστοιχο τρόπο.

Στο τέλος θα έχουμε μία συλλογή από κόμβους που θα περιέχουν το πολύ 1 πόλη και τις αντίστοιχες θύρες του, των οποίων η ανάθεση θα περιγραφεί παρακάτω.

Είναι σημαντικό να αναφερθεί ότι στη θεωρητική προσέγγιση του `Arora` η δημιουργία του τετραδικού δέντρου εμπεριέχει και ένα επιπλέον βήμα τυχαιοποίησης, το οποίο παραλείφθηκε από την υλοποίησή μας για λόγους απλούστευσης των υπολογισμών.

```
finalisedNodes = {}
root = 0-level quadNode

void assignTownsAndPortals():
    divideQuadLeaf(root)

void divideQuadLeaf(quadNode):
    for town in quadNode
        assignPointsToNode(quadNode, town)
    finalizeNodePointsOrDivide(quadNode)

void assignPointsToNode(quadNode, town):
    for node in quadNode.children
        if town inside node
            node.addTown(town)

void finalizeNodePointsOrDivide(quadNode):
```

```

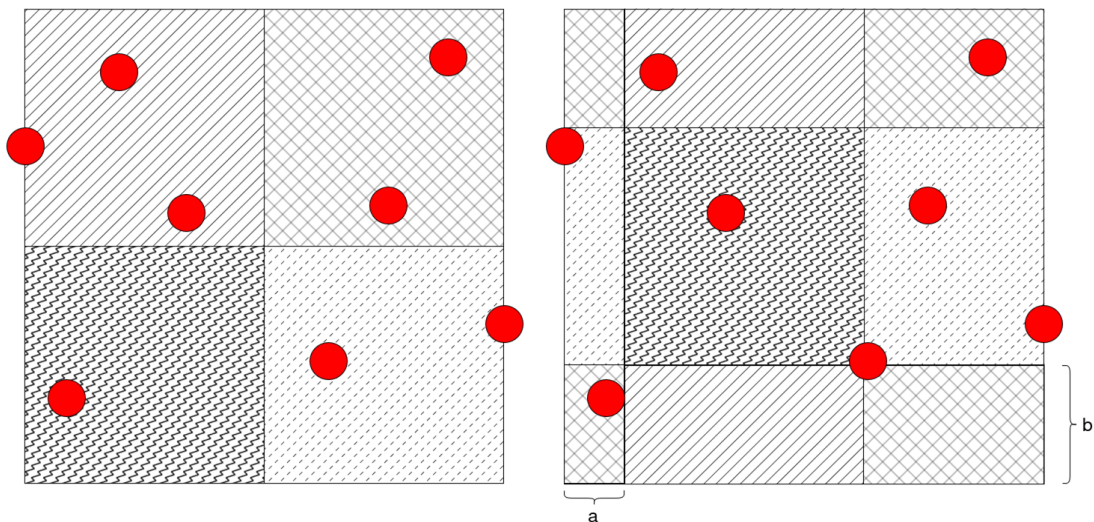
for child in quadNode
    if child.towns > 1
        divideQuadLeaf(child)
    else
        updateCornersAndPortals(child)

void updateCornersAndPortals(quadNode):
    finalisedNodes.add(quadNode)
    calculate quadNode portals

```

Συγκεκριμένα, μετά τον καθορισμό του αρχικού τετραδικού δέντρου επιλέγουμε τυχαία δύο ακεραίους $a, b \in \mathbb{N} \cap [0, L)$, όπου L το μήκος της πλευράς του κλοιού. Έπειτα μετασχηματίζουμε τις (x, y) συντεταγμένες κάθε οριζόντιας και κάθετης γραμμής του τετραδικού δέντρου στις $a + x \bmod L, b + y \bmod L$ αντίστοιχα, δημιουργώντας το τυχαιοποιημένο αναδιπλωμένο τετραδικό δέντρο με (a, b) μετατόπιση (Εικόνα 3.2.1.2). Η θέση των πόλεων στον κλοιό δεν υφίσταται αυτή τη μετατόπιση.

Η εφαρμογή του αλγορίθμου στην τυχαία αυτή επιλογή τετραδικού δέντρου αποδεικνύεται ότι επιτυγχάνει με πιθανότητα τουλάχιστον 0.5 την ζητούμενη α -προσέγγιση. Μάλιστα αυτή η μετατόπιση του δέντρου φαίνεται να είναι κατάλληλη για να αντιμετωπίσει και ένα πρόβλημα το οποίο παρατηρήθηκε κατά τις πρακτικές εφαρμογές της υλοποίησης, αυτό της *αδιαφορίας για την ευρύτερη γεωμετρική περιοχή*. Πόλεις που, κοιτώντας τη βέλτιστη λύση, φαινόταν ότι έπρεπε να τοποθετηθούν κοντά στη διαδρομή, ήταν πολύ πιθανό να ανήκουν σε παιδιά διαφορετικών κόμβων και έτσι να ομαδοποιούνται ξεχωριστά. Αυτή η μετατόπιση, για διάφορες δοκιμές του a και b , θα μπορούσε να οδηγήσει σε τουλάχιστον έναν ορισμό κόμβων που θα ήταν κατάλληλος για να διατηρηθεί η ενότητα της περιοχής, αφού αλλάζοντας τα όρια κάθε κόμβου στο δέντρο αλλάζουν και οι πόλεις που θα ανήκουν σε κάθε κόμβο και άρα η προτεραιότητα προσπέλασής τους. Γι' αυτό το λόγο κρίνεται ότι μάλλον αυτή θα έπρεπε να είναι η πρώτη προσθήκη για βελτίωση της παρούσας υλοποίησης.



ΕΙΚΟΝΑ 3.2.1.2: (a, b) -μετατόπιση του τετραδικού δέντρου, εμφάνιση μόνο των κόμβων επιπέδου-1

3.2.2 Θύρες

Πριν συνεχίσουμε στη φάση της εύρεσης των μονοπατιών είναι χρήσιμο να σημειωθούν κάποιες παρατηρήσεις σχετικά με την ανάθεση των θυρών.

Κατά την διάρκεια της υλοποίησης έγιναν κάποιες παρρεκλίσεις από την θεωρητική θεμελίωση του Αγορα, με γνώμονα τον μικρότερο υπολογιστικό φόρτο. Η πιο ουσιαστική έχει σχέση με τον υπολογισμό των θυρών στους κόμβους και βασίζεται στην αντιμετώπιση που περιγράφεται στο άρθρο “An Empirical Analysis of Approximation Algorithms for Euclidean TSP”[25]. Εκεί οι ερευνητές αποφάσισαν να μη τοποθετούνται θύρες στις γωνίες των κόμβων και μείωσαν τον αριθμό των θυρών στην πλευρά των κόμβων από m , $m \in \mathbb{N}$, σε μία. Η λογική τους ήταν ότι, καθώς οι θύρες δε θα αποτελούν σημεία της τελικής λύσης, ο περιορισμός τους αυτός δε θα επηρέαζε αισθητά το τελικό αποτέλεσμα. Τα συμπεράσματα τους σε γνωστά στιγμιότυπα του προβλήματος TSP φαίνεται να τους επιβεβαίωσαν.

Στην ίδια λογική αποφασίστηκε να μη συμπεριληφθούν οι γωνίες των κόμβων ως θύρες και στη δική μας υλοποίηση. Όσον αφορά τον αριθμό των θυρών ανά πλευρά κόμβου προτιμήθηκε, αντί της σταθερής χρήσης μόνο μίας θύρας, ο ορισμός μίας παραμέτρου κατά την εκτέλεση του αλγορίθμου, ώστε ανάλογα με τον διαθέσιμο χρόνο να μπορούν να δοκιμαστούν και πιθανές λύσεις με περισσότερες θύρες.

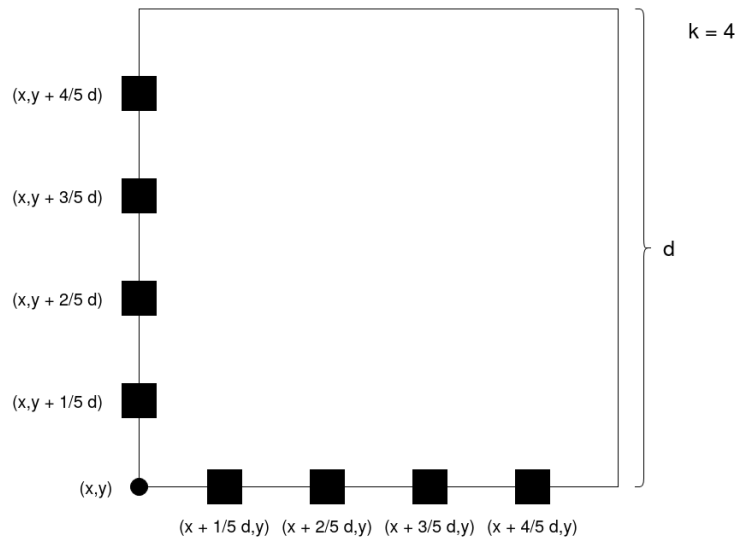
Βάσει των παραπάνω, δοθέντος ενός κόμβου οι θύρες της κάτω πλευράς του θα είναι

$$\left(quadNode.x + \frac{k \cdot quadNode.side}{portalsPerSide + 1}, quadNode.y \right) \mid k \in \{1, \dots, portalsPerSide\} \quad (3.5)$$

Αν η παράμετρος $portalsPerSide$ ισούται με 1, τότε η μοναδική Θύρα θα είναι η

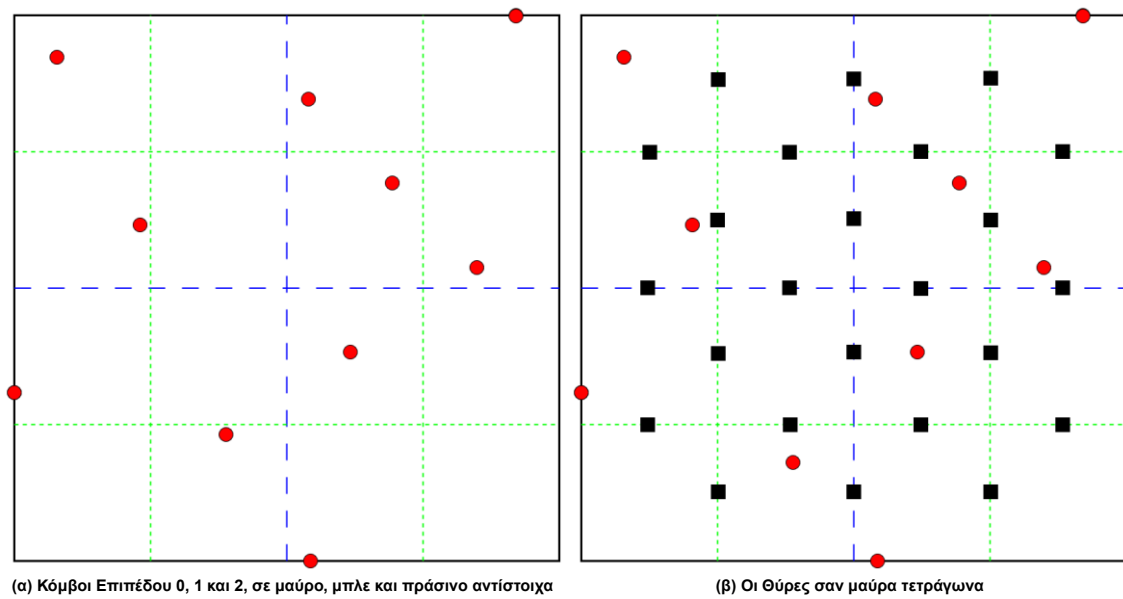
$$\left(quadNode.x + \frac{quadNode.side}{2}, quadNode.y \right) \quad (3.6)$$

δηλαδή το μέσο της πλευράς (Εικόνα 3.2.2.1).

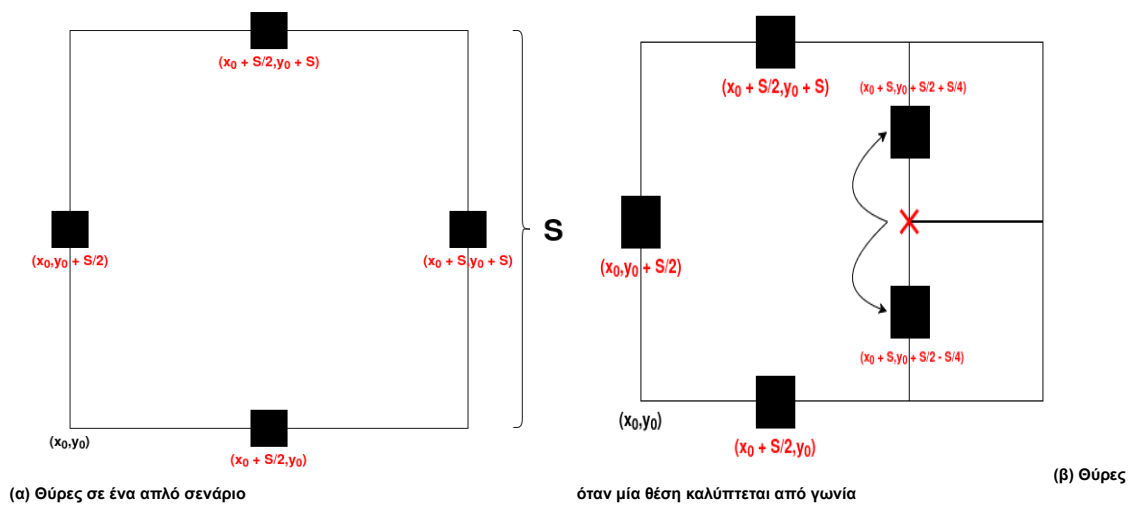


ΕΙΚΟΝΑ 3.2.2.1: Ορισμός θυρών για $k=4$

Αντίστοιχα υπολογίζονται και οι θύρες στις υπόλοιπες πλευρές (Εικόνα 3.2.2.2). Μία προβληματική περίπτωση εμφανίζεται όταν οι συντεταγμένες μίας θύρας ταυτίζονται με αυτές κάποιας γωνίας ενός άλλου κόμβου. Αφού αυτό δεν είναι αποδεκτό στην υλοποίησή μας, σε αυτή την περίπτωση η θύρα αυτή δεν τοποθετείται, αλλά αντικαθιστάται από 2 ή περισσότερες, οι οποίες αναθέτονται στους κόμβους οι οποίοι μοιράζονται τη γωνία (Εικόνα 3.2.2.3).



ΕΙΚΟΝΑ 3.2.2.2: Διαίρεση του τετραδικού δέντρου και προσθήκη θυρών



ΕΙΚΟΝΑ 3.2.2.3: Παραδείγματα υπολογισμού θυρών

3.3 Φάση Εύρεσης Μονοπατιών

3.3.1 Μονοπάτια Κόμβων

Μετά τον σχηματισμό του τετραδικού δέντρου και της ανάθεσης θυρών στα παιδιά των επιπέδων του μπορεί να αρχίσει ο υπολογισμός των μονοπατιών.

Αρχικά υπολογίζονται όλα τα πιθανά μονοπάτια μεταξύ θυρών σε κάθε οριστικοποιημένο κόμβο. Έστω Θ_Q το σύνολο των θυρών σε έναν οριστικοποιημένο κόμβο Q . Τα μονοπάτια του Q , τα οποία θα αναφέρονται ως Τμήματα, είναι τα στοιχεία του συνόλου

$$SEG_Q = \{AB \mid A, B \in \theta_Q, A \neq B\} \quad (3.7)$$

Το μήκος ενός Τμήματος θα είναι

- η Ευκλείδεια απόσταση των άκρων του, εφόσον ο Κόμβος δεν περιέχει Πόλη
- το άθροισμα των Ευκλείδειων αποστάσεων κάθε άκρου από την Πόλη, εφόσον ο Κόμβος περιέχει Πόλη

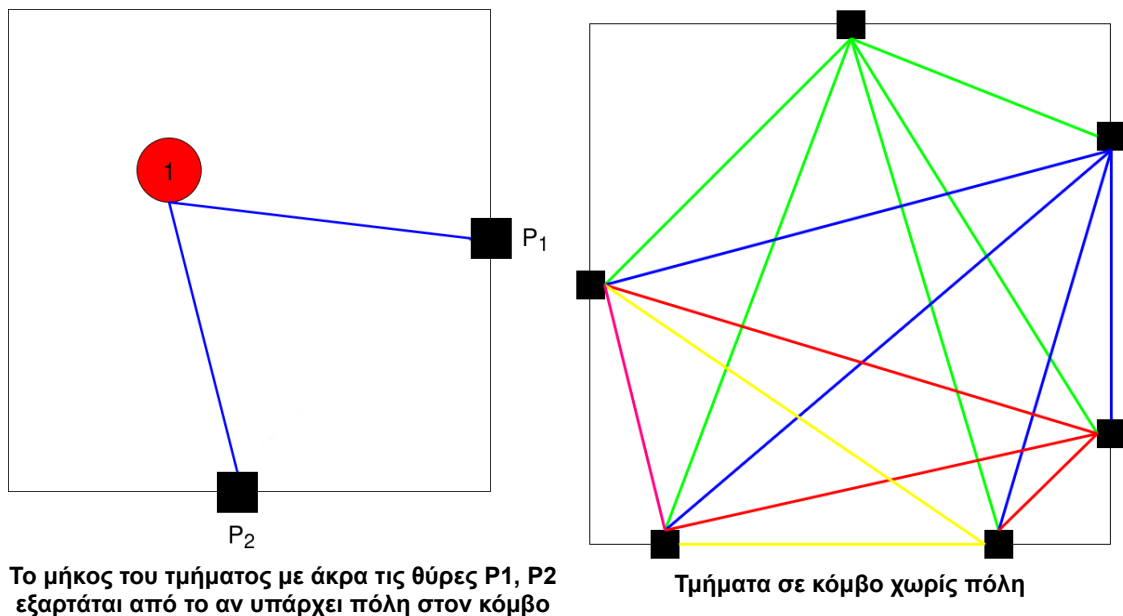
Επομένως, όπως βλέπουμε στην Εικόνα 3.3.1.1,

- σε έναν κόμβο χωρίς πόλη τα τμήματα θα αποτελούνται από όλα τα πιθανά ευθύγραμμα τμήματα με άκρα κάποιες θύρες του κόμβου
- σε έναν κόμβο με πόλη το κάθε τμήμα θα αποτελείται από δύο διαδοχικά ευθύγραμμα τμήματα, όπου το καθένα θα έχει για άκρα του αυτή την πόλη και μία από τις θύρες του κόμβου.

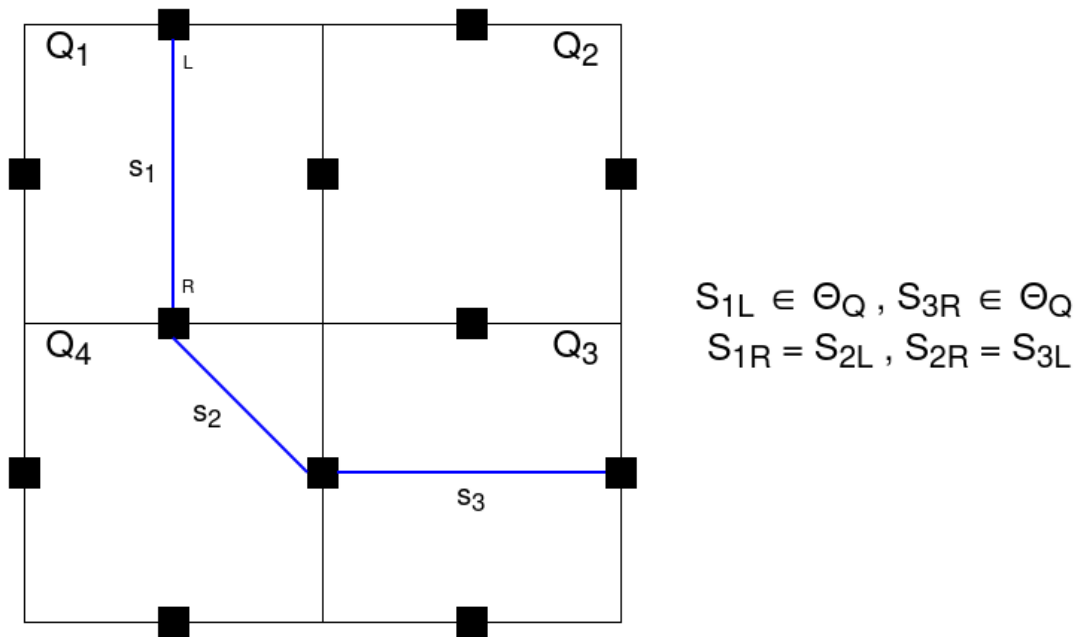
Τα τμήματα ενός κόμβου επιπέδου- i υπολογίζονται συνδυάζοντας τμήματα των επιπέδου- $(i+1)$ παιδιών του, τα οποία έχουν κάποιο κοινό άκρο. Δηλαδή το σύνολο SEG_Q με όλα τα τμήματα του κόμβου θα αποτελείται από συλλογές τμημάτων με το πολύ τέσσερα στοιχεία, έτσι ώστε:

- κάθε τμήμα να ανήκει σε μόνο ένα παιδί
- η αρχή του πρώτου τμήματος και το τέλος του τελευταίου να είναι θύρες του γονέα
- τα διαδοχικά τμήματα να έχουν κοινό άκρο

Στην Εικόνα 3.3.1.2 φαίνεται μια σχηματική αναπαράσταση ενός στοιχείου της παραπάνω συλλογής.



ΕΙΚΟΝΑ 3.3.1.1: Τμήματα κόμβων



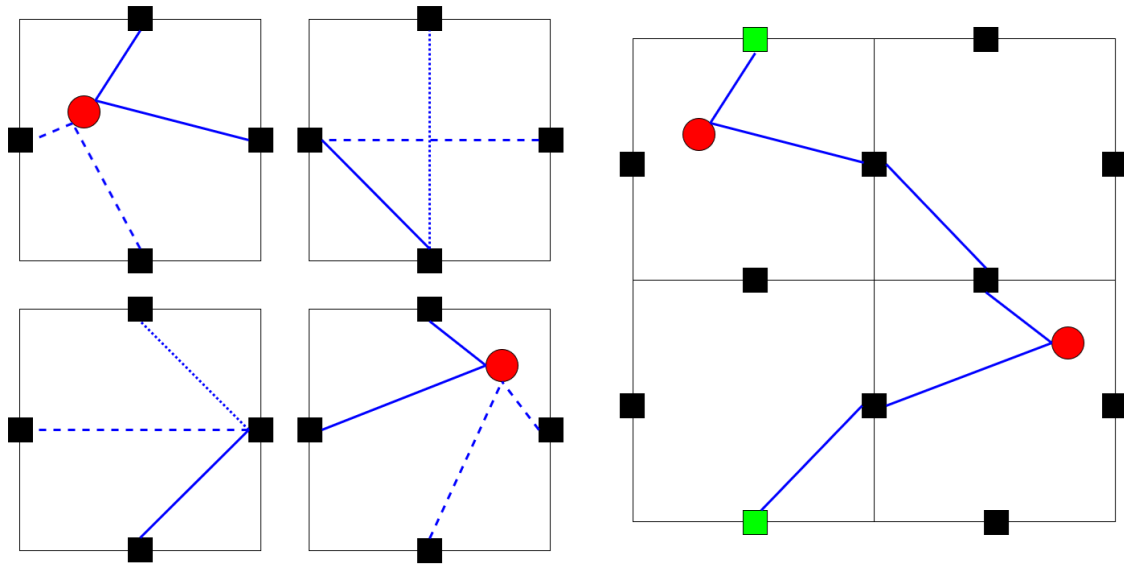
ΕΙΚΟΝΑ 3.3.1.2: Δημιουργία ενός τμήματος του κόμβου Q, αγνοώντας τις πόλεις

```

void iterate(Nodes)
    Parents := {}
    for q in Nodes:
        findPaths(q)
        if (q.parent not in Parents) add q.parent to Parents
    iterate(Parents)

void findPaths(q)
    for x != y in q.portals:
        if q is finalized:
            add xy to q.paths
        else:
            find all sequences of children segments from x to y
            add them to q.paths

```



ΕΙΚΟΝΑ 3.3.1.3: (αριστερά) Μερικά τμήματα για κάθε έναν από 4 αδερφικούς κόμβους (δεξιά) Μία επιλογή τμημάτων των αδερφικών κόμβων που δημιουργεί τμήμα του γονεϊκού κόμβου

Όπως φαίνεται και στον ψευδοκώδικα επιλέγουμε τους τελικούς κόμβους, όπως αυτοί βρέθηκαν στην ενότητα 3.2.1 και για κάθε ζευγάρι θυρών τους υπολογίζουμε όλα τα τμήματά τους. Μετά από αυτό τον υπολογισμό αποθηκεύουμε σε μία συλλογή τους γονείς τους, ώστε σε επόμενο στάδιο να υπολογίσουμε τα δικά τους τμήματα. Αυτό, σε αντίθεση με τους τελικούς κόμβους, δεν γίνεται απλά ενώνοντας κατάλληλα τις θύρες τους, αλλά συνδυάζοντας κατάλληλα τα ήδη υπολογισμένα τμήματα των παιδιών τους (Εικόνα 3.3.1.3). Αυτή τη διαδικασία μπορεί να συνεχιστεί αναδρομικά μέχρι να φτάσουμε στο Επίπεδο-1, δηλαδή στα 4 αρχικά Παιδιά. Σε αυτό το στάδιο η αντιμετώπιση μας θα αλλάξει, καθώς από το επίπεδο-1 στο επίπεδο-0 πρέπει να συνθέσουμε διαδρομές και όχι απλά τμήματα που να συνδέουν κάποιες θύρες.

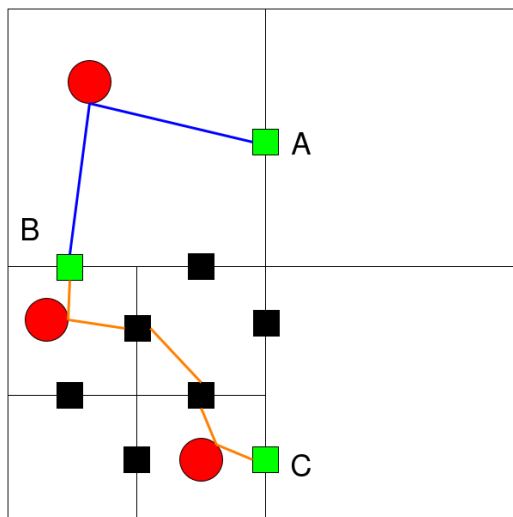
3.3.2 TSP Μονοπάτια

Έχοντας υπολογίσει όλα τα πιθανά τμήματα των 4 κόμβων επιπέδου-1, μπορούμε να βρούμε το βέλτιστο μονοπάτι που διέρχεται από κάθε πόλη. Έστω Q1 ένας από τους κόμβους επιπέδου-1.

Επιλέγουμε ένα τυχαίο τμήμα AB του Q1, με A,B θύρες του Q1. Αρχικά βρίσκουμε έναν γειτονικό κόμβο με τον οποίο ο Q1 να μοιράζεται την θύρα B, δηλαδή έναν από τους δύο κόμβους που έχει κοινή πλευρά με τον Q1, έστω Q2. Επιλέγουμε όλα τα τμήματά του Q2 τα οποία αρχίζουν από τη θύρα B. Αφού το ένα έχει για δεξί άκρο και το άλλο για αριστερό άκρο το B, τα τμήματα αυτά είναι διαδοχικά. Έστω ότι ένα τέτοιο τμήμα είναι το BC (Εικόνα 3.3.2.1). Επαναλαμβάνουμε παρομοίως βρίσκοντας τον γειτονικό κόμβο του Q2 που μοιράζεται μαζί του την θύρα C και επιλέγουμε τα τμήματά του που έχουν για άκρο το C. Συνεχίζουμε την επανάληψη μέχρι:

- να εξαντλήσουμε και τους 4 κόμβους επιπέδου-1 και το άκρο του τελευταίου τμήματος να ταυτίζεται με την αρχή του πρώτου (A). Τότε το μονοπάτι είναι μία πιθανή λύση του TSP.
- να εξαντλήσουμε και τους 4 κόμβους επιπέδου-1 και το άκρο του τελευταίου τμήματος να μην ταυτίζεται με την αρχή του πρώτου (A). Τότε το μονοπάτι απορρίπτεται και δοκιμάζουμε άλλο υποψήφιο τμήμα.
- να μην υπάρχει Κόμβος που να περιέχει μία θύρα και να μην τον έχουμε ήδη επισκεφθεί.

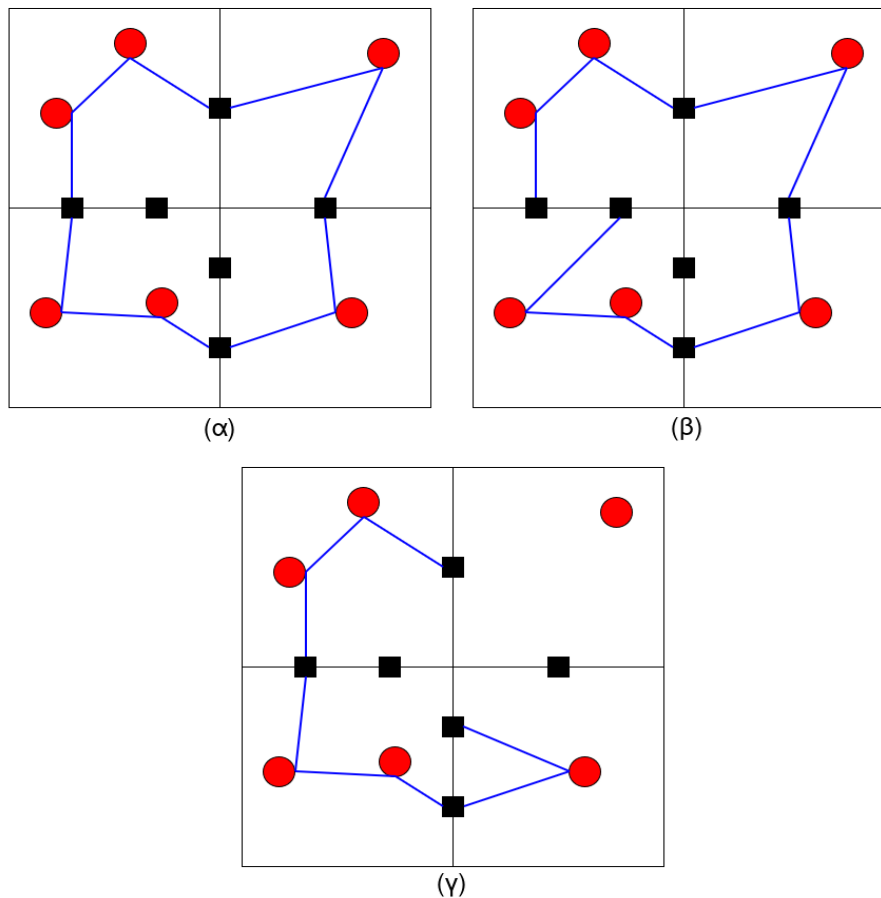
Στην Εικόνα 3.3.2.2 φαίνονται αυτά τα σενάρια.



ΕΙΚΟΝΑ 3.3.2.1: Συνένωση των τμημάτων με άκρα AB του Q1 και BC ενός γειτονικού κόμβου Q2

Στον ψευδοκώδικα, στη μέθοδο `stitch` θεωρούμε ένα τμήμα `segment` του Κόμβου Q1, με άκρα τις θύρες `entrance` και `stitch`, και προσθέτουμε αυτό το τμήμα στο τρέχον μονοπάτι. Βρίσκουμε ποιος γειτονικός κόμβος μοιράζεται τη θύρα `stitch` και καλούμε τη μέθοδο `branch` γι' αυτόν τον κόμβο.

Στη μέθοδο `branch` θεωρούμε όλα τα τμήματα του κόμβου που έχουν για άκρο το `stitch`, δηλαδή το τέλος του προηγούμενου τμήματος του μονοπατιού. Αν κάποιο από αυτά τα τμήματα έχει για άλλο άκρο το `entrance` τότε έχουμε κλείσει επιτυχώς έναν κύκλο και επομένως έχουμε μία TSP διαδρομή. Αν όχι συνεχίζουμε τη διαδικασία, χρησιμοποιώντας ως νέο `stitch` το τέλος του τμήματος και ψάχνοντας να το προεκτείνουμε με κάποιο τμήμα ενός επόμενου γειτονικού κόμβου.



ΕΙΚΟΝΑ 3.3.2.2: Διαδοχικές συνενώσεις τμημάτων οδηγούν σε μία έγκυρη (α) και δύο άκυρες (β,γ) διαδρομές TSP

Συνεχίζοντας με αυτό τον τρόπο είτε θα οδηγηθούμε σε μία TSP διαδρομή είτε σε κάποιο αδιέξοδο. Στην πρώτη περίπτωση ελέγχουμε το μήκος της διαδρομής και την αποθηκεύουμε αν αυτό είναι μικρότερο απ' το μήκος της τελευταίας αποθηκευμένης λύσης. Στη δεύτερη περίπτωση, αφαιρούμε από το τρέχον μονοπάτι το τελευταίο τμήμα, δηλαδή αυτό που μας οδήγησε στο αδιέξοδο, και δοκιμάζουμε την επόμενη πιθανή επιλογή.

Μετά το τέλος της αναδρομής, η μεταβλητή solution θα έχει το συντομότερο μονοπάτι το οποίο διέρχεται από τις n πόλεις. Κρατώντας τη σειρά των πόλεων όπως εμφανίστηκε στο μονοπάτι, αγνοώντας πια τις θύρες και σπάζοντας τους κύκλους όπως εμφανίζονται, υπολογίζουμε το πραγματικό μήκος της λύσης.

```
void stitch(Nodes)
    node := Q1
    for segment in node.segments:
        entrance := segment.start
        path := {segment}
        stitch := segment.end
        neighbour := unvisited node that contains stitch
        branch(entrance, segment, neighbour, path)
        remove segment from path

void branch(entrance, segment, node, path)
```

```

stitch := segment.end
for nextSegment in node.segments such that stitch in nextSegment:
    add nextSegment to path
    if entrance is endpoint of nextSegment:
        if solution != {}:
            solution = path
        else:
            compare path with solution
            if shorter set path as solution
    else
        newStitch := nextSegment endpoint not equal to stitch
        neighbour := unvisited node that contains newStitch
        branch(entrance, nextSegment, neighbour, path)
remove nextSegment from path

```

3.4 Αποτελέσματα

Ακολουθούν οι λύσεις που προκύπτουν από την υλοποίησή μας στα γνωστά στιγμιότυπα του προβλήματος TSP με καρτεσιανές συντεταγμένες και γνωστές βέλτιστες λύσεις, όπως αναγράφονται στο απωθητήριο TSPLIB[26, 27], καθώς και η σύγκριση των προσεγγίσεών μας με αυτές τις λύσεις. Θα αναφέρουμε το μήκος των βέλτιστων μονοπατιών που επισκέπτονται όλες τις πόλεις, το μήκος των καλύτερων προσεγγίσεών μας καθώς και την απόκλιση τους από την βέλτιστη. Στα σχετικά σχήματα θα φαίνεται και ακριβώς η σειρά επίσκεψης των πόλεων για καθεμία από τις λύσεις.

Θα παρατηρήσουμε ότι υπάρχουν παραδείγματα που διευκολύνουν τον αλγόριθμο, καθώς η βέλτιστη λύση φαίνεται να μην έχει μια συγκεκριμένη γεωμετρία και έτσι η προσέγγιση με την τετραπλή διαίρεση του χώρου χωρίς την τυχαιοποίηση της (a,b)-μετατόπισης δε λειτουργεί αρνητικά. Σε όσα η απόδοση είναι λίγο χειρότερη, θα δούμε ότι υπάρχουν συνευθειακές πόλεις και “καλά ορισμένες” ομάδες πόλεων που αποτελούν περιπτώσεις τις οποίες δεν μπορούμε να λάβουμε υπ’ όψιν, καθώς η σειρά με την οποία συνδέονται τα επιμέρους κομμάτια διαδρομών είναι συγκεκριμένη στον αλγόριθμό μας. Δηλαδή είναι σύνηθες η αρχική διαίρεση του χώρου να χωρίζει αυτές τις καλά ορισμένες ομάδες, των οποίων τα χωρισμένα κομμάτια θα ληφθούν υπ’ όψιν ξεχωριστά, μαζί με τις υπόλοιπες πόλεις του κόμβου τους.

Για όλα τα παραδείγματα δοκιμάστηκαν διαφορετικοί συνδυασμοί θυρών ανά πλευρά και παραμέτρον μετασχηματισμού ε ώστε να επιλεγεί η βέλτιστη λύση. Συγκεκριμένα δοκιμάστηκαν όλοι οι πιθανοί συνδυασμοί για

- 1, 2, ..., 10 θύρες ανά πλευρά
- ε με τιμές 1, 0,5, 0,2, 0,1, 0,01

Σε αντίθεση με τις υλοποιήσεις που θα δούμε έπειτα για το πρόβλημα του k-TSP δε χρησιμοποιήθηκαν πολυνηματικές τεχνικές (multithreading) καθώς η επίδοση του αλγορίθμου ήταν ικανοποιητική ακόμα και για μεγαλύτερο αριθμό θυρών. Παρόλα αυτά σαν δικλείδα ασφαλείας προστέθηκε ένα ανώτατο όριο στον χρόνο εκτέλεσης του αλγορίθμου για κάθε συνδυασμό αριθμού θυρών και ε.

ΠΙΝΑΚΑΣ 1: Σύγκριση προσεγγίσεων με βέλτιστες λύσεις για το TSP

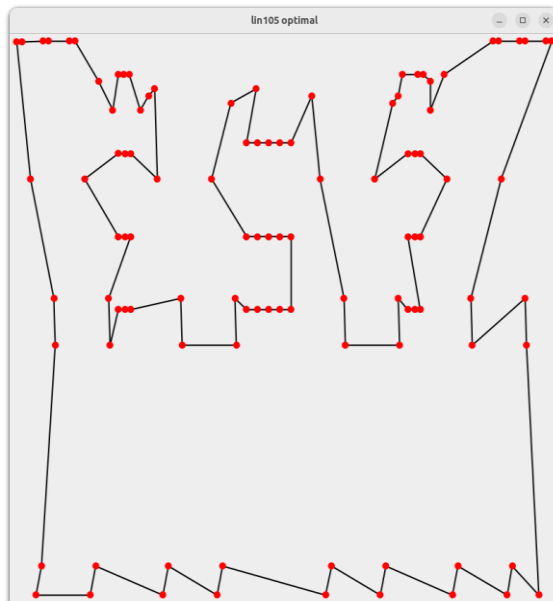
	Βέλτιστη	Καλύτερη Προσέγγιση	Απόκλιση
LIN105	14379 μ.	15418 μ.	-7.22%
BERLIN52	7542 μ.	8139 μ.	-7.92%
EIL101	629 μ.	699 μ.	-11.26%

KROA100	21282 μ.	23968 μ.	-12.63%
ST70	675 μ.	760 μ.	-12.69%
KROC100	20749 μ.	23740 μ.	-14.42%
CH130	6110 μ.	7023 μ.	-14.95%
PR76	108159 μ.	124342 μ.	-14.96%
CH150	6528 μ.	7642 μ.	-17.07%
A280	2579 μ.	3037 μ.	-17.77%
KROD100	21294 μ.	25607 μ.	-20.26%

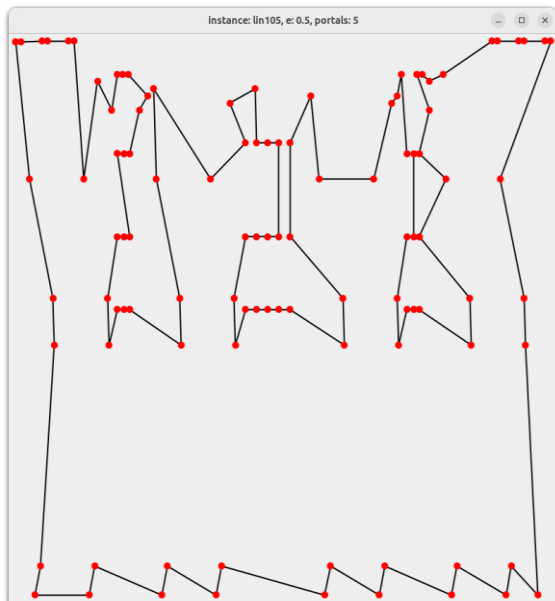
Ο παραπάνω πίνακας εμφανίζει συνοπτικά τις αποκλίσεις των καλύτερων προσεγγίσεών μας από τις βέλτιστες λύσεις σε ένα υποσύνολο των παραδειγμάτων του TSPLIB. Είναι σημαντικό να αναφερθεί ότι αυτό το υποσύνολο περιέχει μερικές και από τις χειρότερες προσεγγίσεις ($> 15\%$), καθώς σε πολλά από τα παραδείγματα στα οποία η απόκλιση ήταν του επιπέδου 5-10% το TSPLIB δεν προσέφερε τη βέλτιστη διαδρομή καθαντή παρά μόνο το μήκος της. Αυτά τα παραδείγματα δεν προτιμήθηκαν στο πλαίσιο της παρουσίασης, καθώς δε θα μπορούσαν να αποδοθούν γραφικά και να δώσουν μία πιο ουσιαστική σύγκριση των λύσεων.

Ακόμα και αν λάβουμε υπ' όψιν την παραπάνω σημείωση, βλέπουμε ότι ακόμα και χωρίς την (a,b)-μετατόπιση στην χειρότερη περίπτωση έχουμε μία λύση κατά το 1/5 χειρότερη από την βέλτιστη, με τις περισσότερες να κυμαίνονται γύρω από το -15%. Συγκεκριμένα, αν βγάzaμε έναν μέσο όρο η απόδοση του αλγορίθμου μας σε αυτά τα 11 παραδείγματα είναι -13.74%.

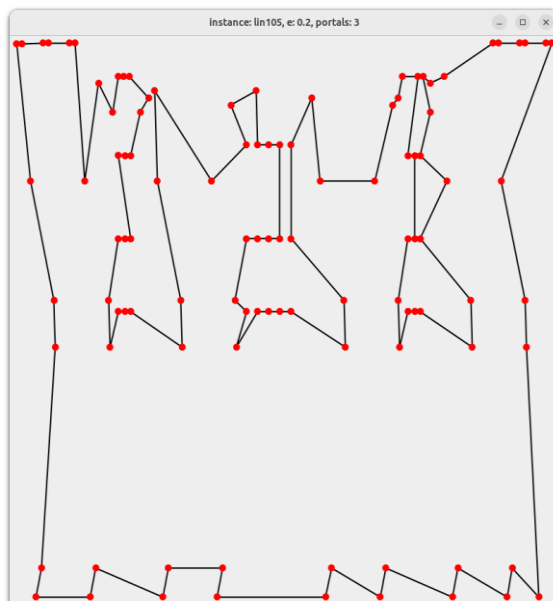
Στις επόμενες σελίδες θα παρουσιαστούν γραφικά οι βέλτιστες λύσεις και οι βέλτιστες προσεγγίσεις, καθώς και κάποιες λιγότερο αποδοτικές προσεγγίσεις συνοδευόμενες από τις παραμέτρους και το μήκος τους. Αυτό που θα παρατηρηθεί από το πρώτο στιγμιότυπο και θα επαληθευτεί και στα επόμενα είναι ότι η ελαχιστοποίηση του e , δηλαδή η αύξηση πυκνότητας του κλοιού, και η μεγιστοποίηση του αριθμού θυρών, δηλαδή η αύξηση των πιθανών επιλογών μονοπατιών μέσα σε κάθε κόμβο, δεν οδηγούν απαραίτητα σε καλύτερη λύση. Μάλιστα, στις περισσότερες περιπτώσεις ήδη από τις 5 θύρες μπορούσε να παρατηρηθεί μία στασιμότητα ως προς τη βέλτιστη λύση και, συγκεκριμένα στα παραδείγματα που δεν υπήρχαν πολύ κοντινές πόλεις, το e μπορούσε να μείνει στις ψηλές του τιμές χωρίς επιπτώσεις, δηλαδή στις τιμές 1 και 0.5.



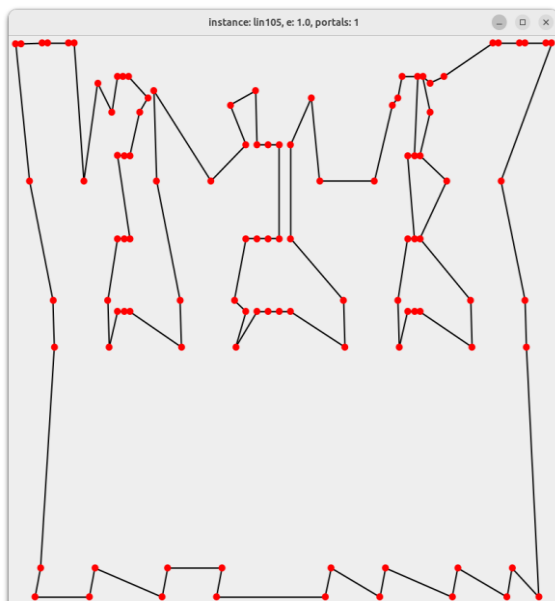
Βέλτιστη Λύση (14379 μ.)



$e = 0.5$, 5 θύρες (βέλτιστη προσέγγιση, 15418 μ.)

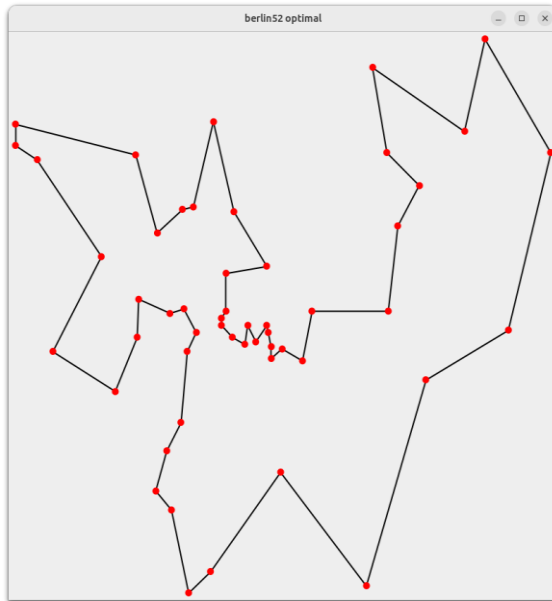


$e = 0.2$, 3 θύρες (15557 μ.)



$e = 1$, 1 θύρα (15553 μ.)

ΕΙΚΟΝΑ 3.4.1: Στιγμιότυπο LIN105



Βέλτιστη Λύση (7542 μ.)



e=1, 1 θύρα (βέλτιστη προσέγγιση, 8139 μ.)

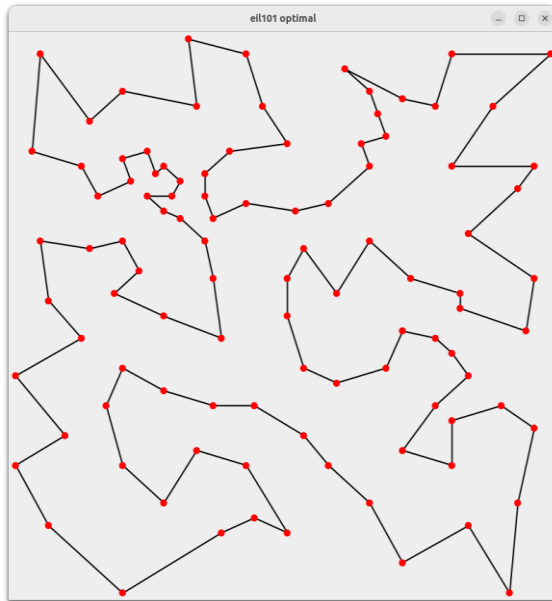


e=0.5, 4 θύρες (8319 μ.)

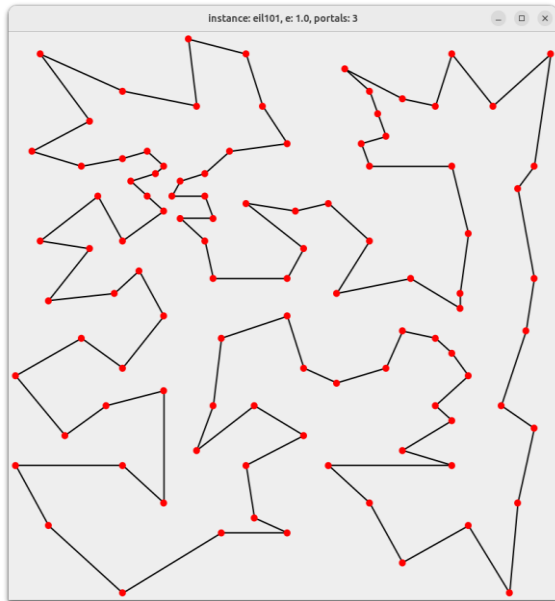


e=1, 5 θύρες (8139 μ.)

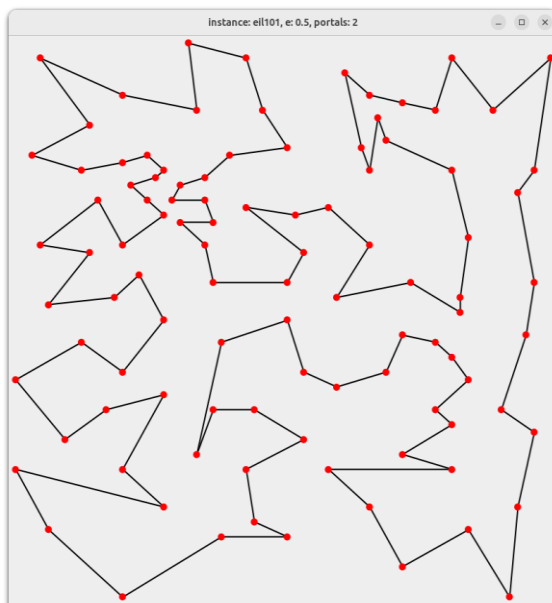
ΕΙΚΟΝΑ 3.4.2: Στιγμιότυπο BERLIN52



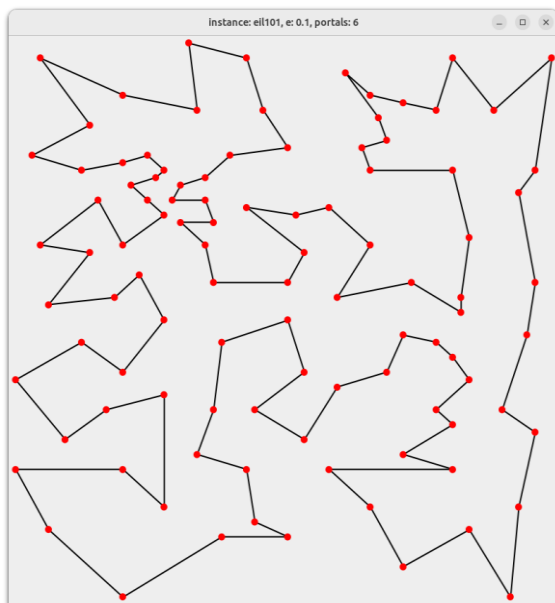
Βέλτιστη Λύση (629 μ.)



e=1, 3 θύρες (βέλτιστη προσέγγιση, 699 μ.)

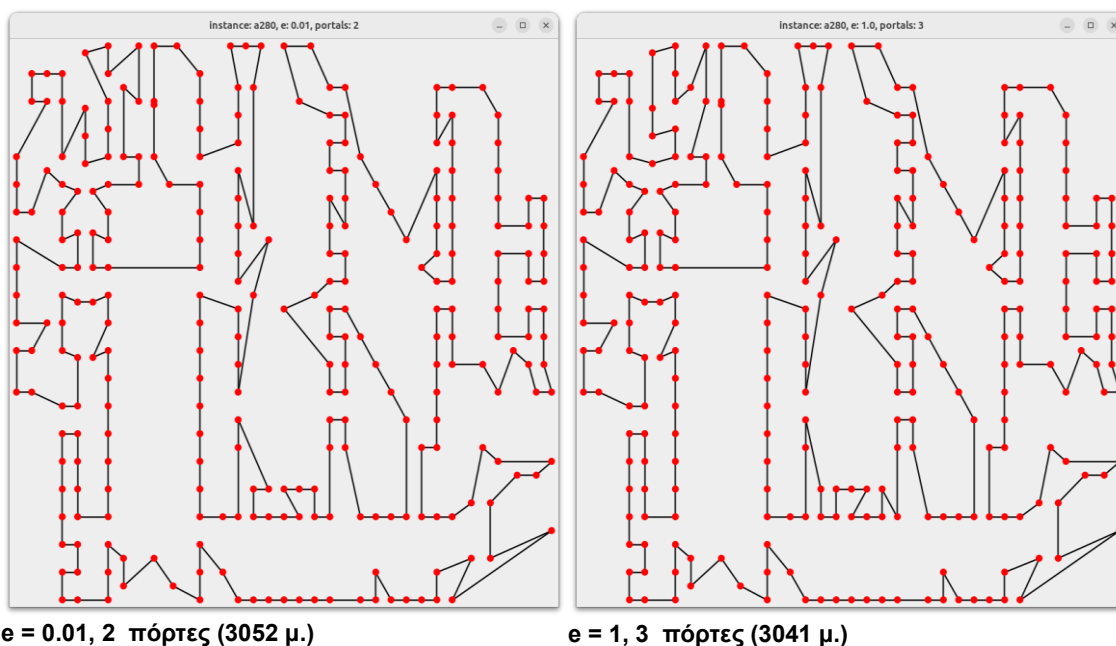
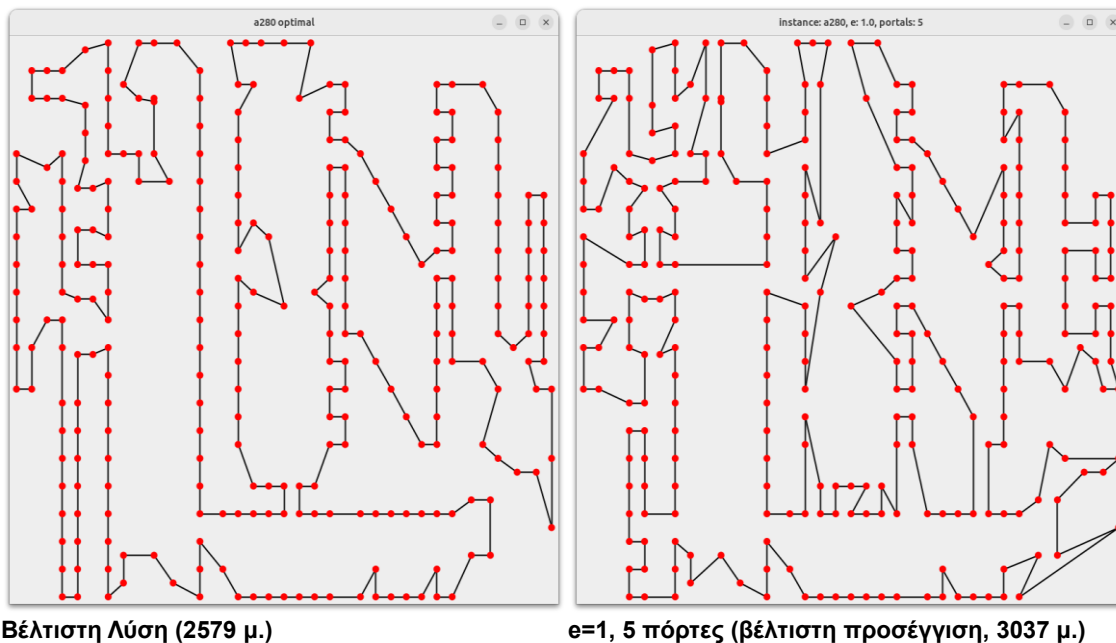


e = 0.5, 2 θύρες (709 μ.)



e = 0.1, 6 θύρες (700 μ.)

ΕΙΚΟΝΑ 3.4.3: Στιγμιότυπο EIL101



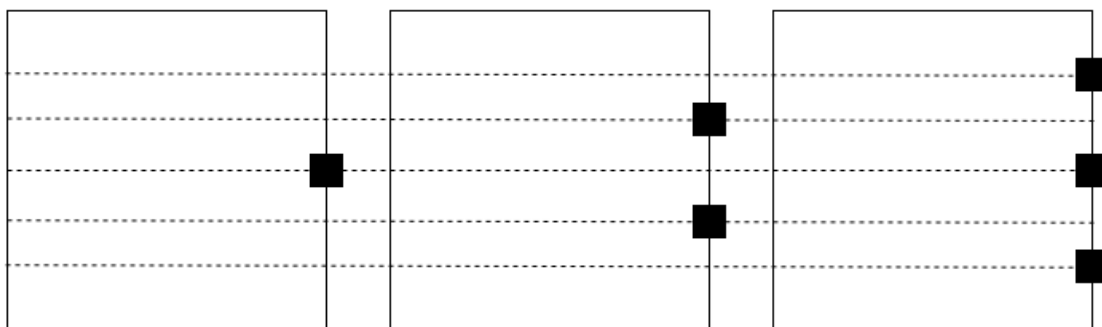
ΕΙΚΟΝΑ 3.4.4: Στιγμιότυπο A280

Στο συγκεκριμένο στιγμιότυπο (Εικόνα 3.4.4) παρατηρούμε και μια ιδιαιτερότητα του αλγορίθμου. Αν και θα περιμέναμε ο μεγαλύτερος αριθμός θυρών ανά πλευρά πάντα να βελτιώνει ή στην χειρότερη να αφήνει ακριβώς ίδιο το μήκος της προσέγγισης, αυτό δεν συμβαίνει απαραίτητα σε κάθε περίπτωση. Για παράδειγμα, για $e = 1$ η προσέγγιση με 1 θύρα ανά πλευρά δίνει λύση με κόστος περίπου 3042 ενώ με 2 θύρες ανά πλευρά περίπου 3051, δηλαδή 10 μονάδες μεγαλύτερη. Αυτό εξηγείται από τον τρόπο που οι

θύρες τοποθετούνται στις πλευρές και που δεν εξασφαλίζει ότι όλες οι θύρες που εμφανίζονται σε μία εκτέλεση του αλγορίθμου θα εμφανίζονται και στα επόμενα.

Συγκεκριμένα, έστω d το μήκος της πλευράς. Όταν θέτουμε 1 θύρα ανά πλευρά, αυτή θα τοποθετηθεί σε απόσταση $d/2$ από το ένα άκρο, ενώ όταν θέσουμε 2 αυτές θα τοποθετηθούν σε αποστάσεις $d/3$ και $2d/3$. Από τη στιγμή που οι θύρες μίας εκτέλεσης δεν αποτελούν υπερσύνολο του άλλου, είναι πιθανό η προσέγγιση που προκύπτει από το τρέξιμο με λιγότερες θύρες να τυχαίνει να είναι καλύτερη (Εικόνα 3.4.5).

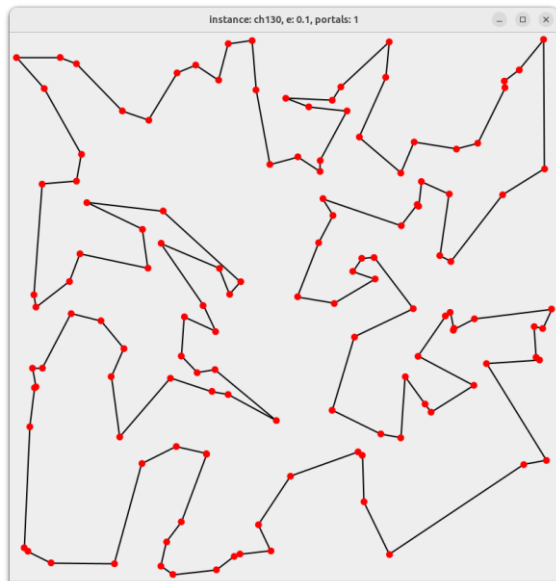
Αυτή η θεωρία φαίνεται να επιβεβαιώνεται όταν θέτουμε 3 θύρες ανά πλευρά. Αυτές τοποθετούνται σε αποστάσεις $d/4$, $2d/4 = d/2$ και $3d/4$ και επομένως θα αποτελούν υπερσύνολο των θυρών του τρεξίματος με 1 θύρα. Όπως είναι αναμενόμενο η προσέγγιση είναι καλύτερη (3040 μονάδες), ενώ στην χειρότερη θα ήταν ακριβώς η ίδια.



ΕΙΚΟΝΑ 3.4.5: Τοποθέτηση 1, 2 και 3 θυρών ανά πλευρά κόμβου



Βέλτιστη Λύση (6110 μ.)



$e=0.1$, 1 θύρα (βέλτιστη προσέγγιση, 7023 μ.)

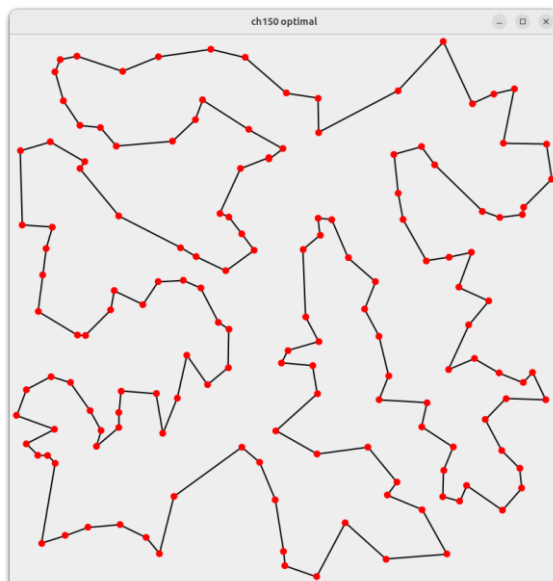


$e=0.5$, 3 θύρες (7080 μ.)



$e=0.2$, 4 θύρες (7061 μ.)

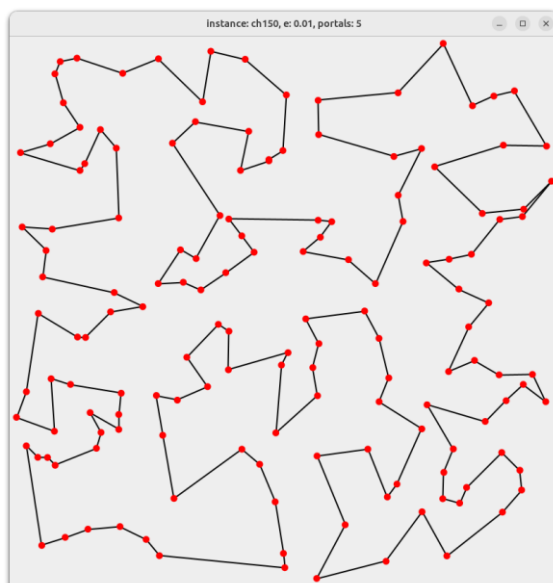
ΕΙΚΟΝΑ 3.4.6: Στιγμιότυπο CH130



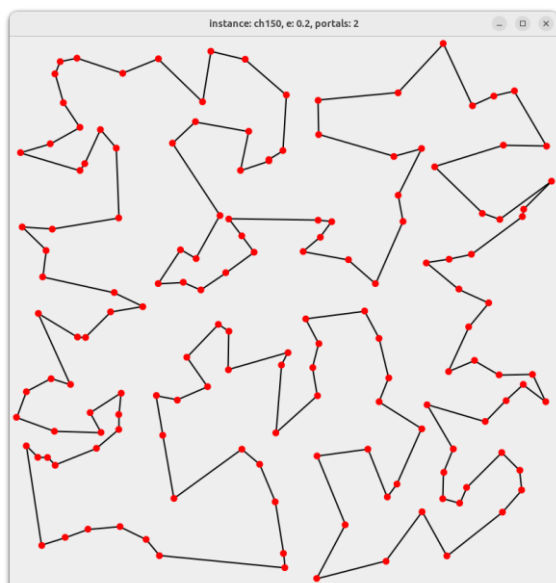
Βέλτιστη Λύση (6528 μ.)



$e = 0.5$, 2 θύρες (βέλτιστη προσέγγιση, 7642 μ.)



$e = 0.01$, 5 θύρες (7665 μ.)

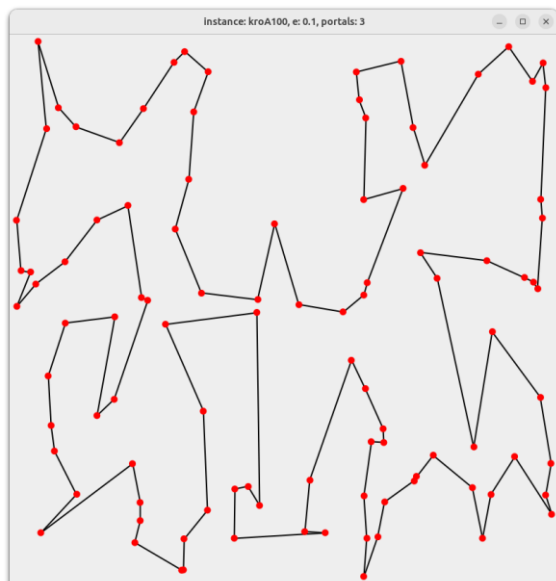


$e = 0.2$, 2 θύρες (7643 μ.)

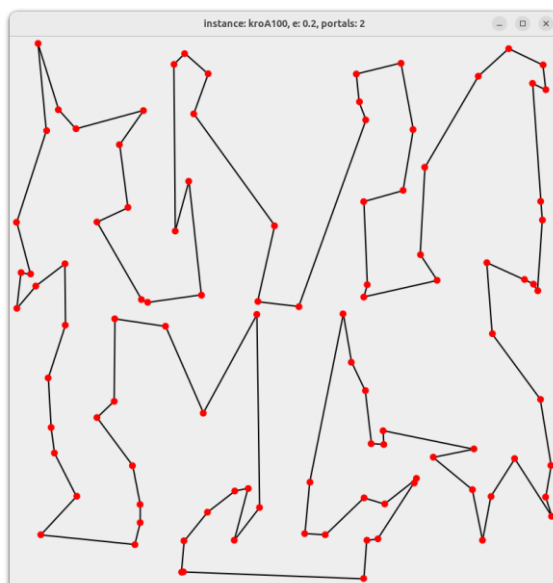
ΕΙΚΟΝΑ 3.4.7: Στιγμιότυπο CH150



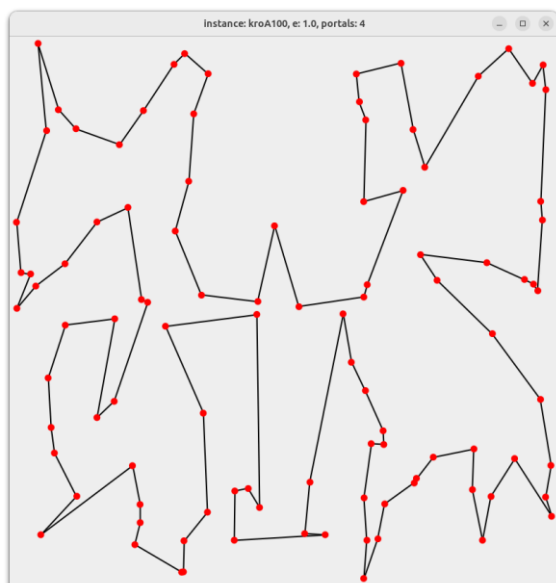
Βέλτιστη Λύση (21282 μ.)



$e = 0.1$, 3 θύρες (βέλτιστη προσέγγιση, 23968 μ.)

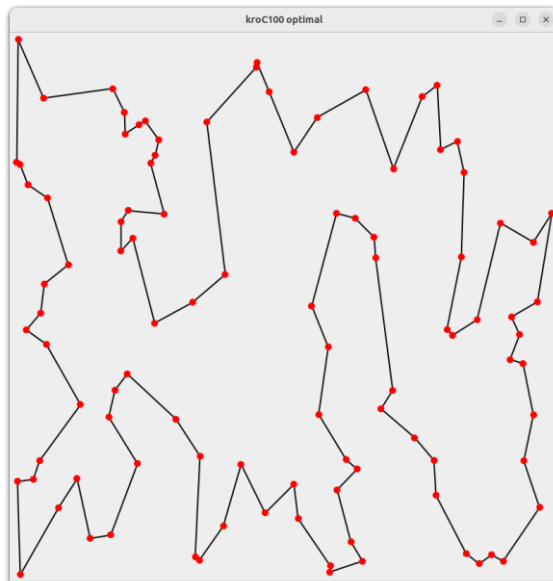


$e = 0.2$, 2 θύρες (26362 μ.)

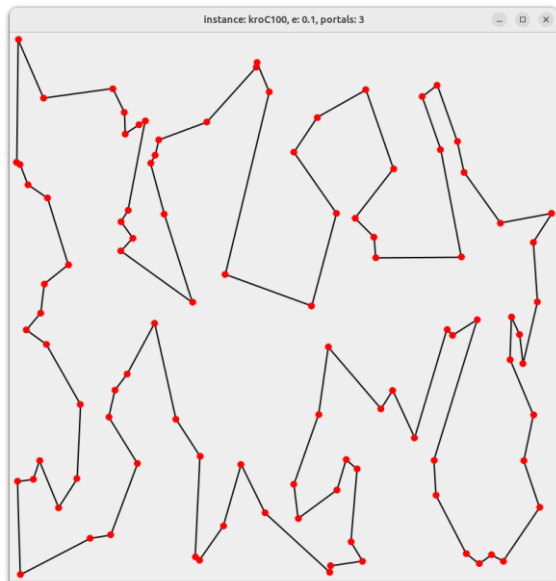


$e = 1$, 4 θύρες (24093 μ.)

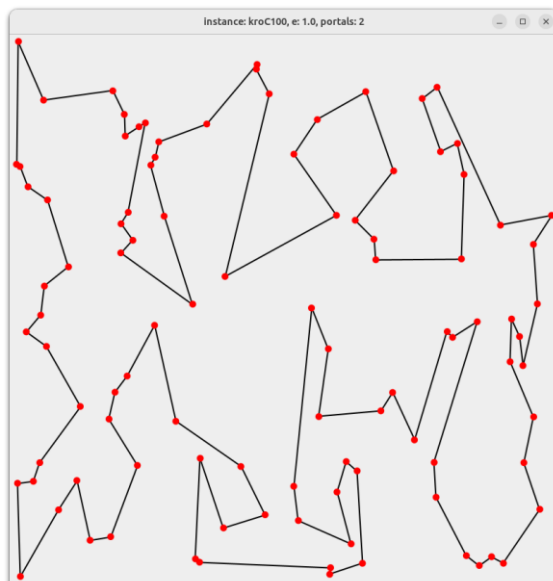
ΕΙΚΟΝΑ 3.4.8: Στιγμιότυπο KROA100



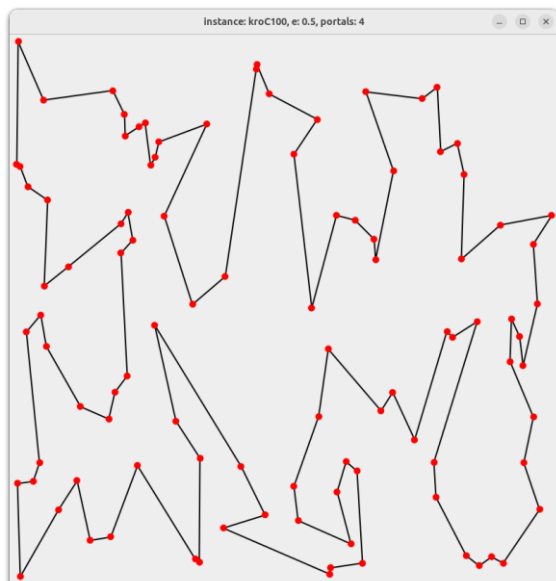
Βέλτιστη Λύση (20749 μ.)



$\epsilon = 0.1$, 3 θύρες (βέλτιστη προσέγγιση, 23740 μ.)

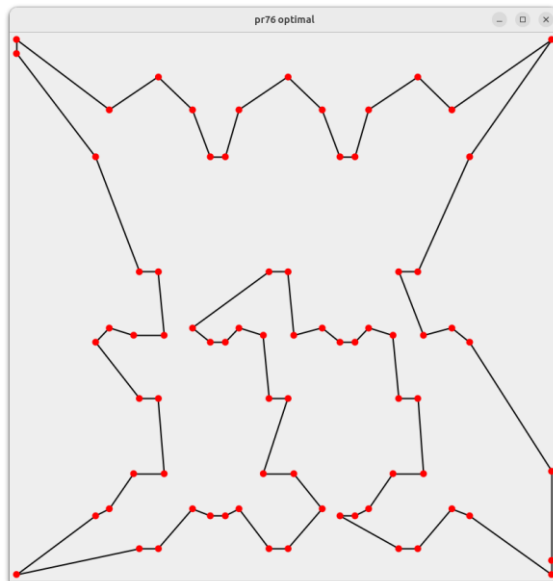


$\epsilon = 1$, 2 θύρες (25330 μ.)

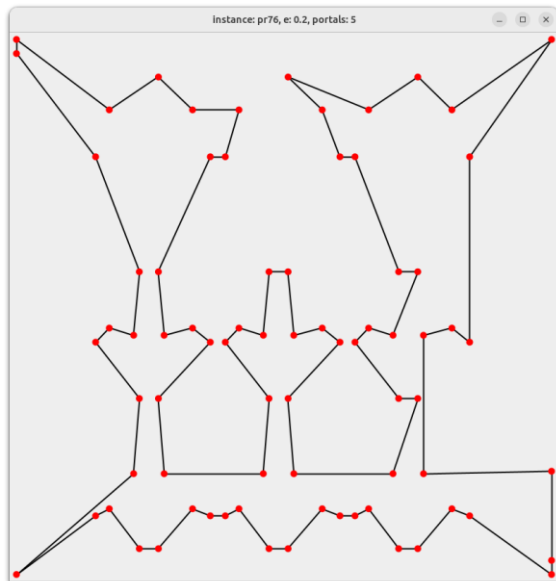


$\epsilon = 0.5$, 4 θύρες (24622 μ.)

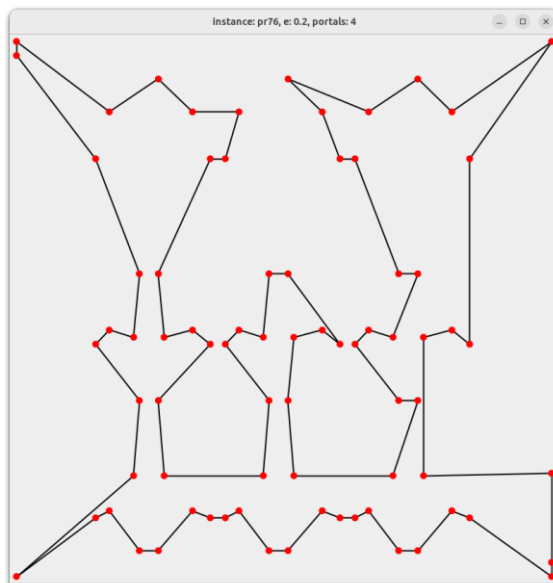
ΕΙΚΟΝΑ 3.4.9: Στιγμιότυπο KROC100



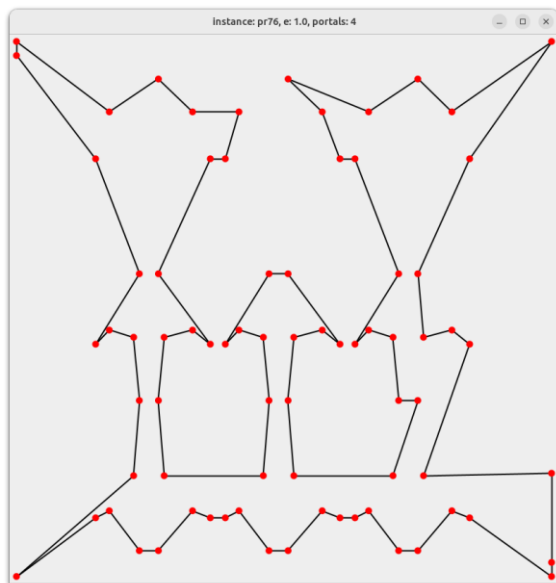
Βέλτιστη Λύση (108159 μ.)



$e = 0.2, 5$ θύρες (βέλτιστη προσέγγιση, 124342 μ.)

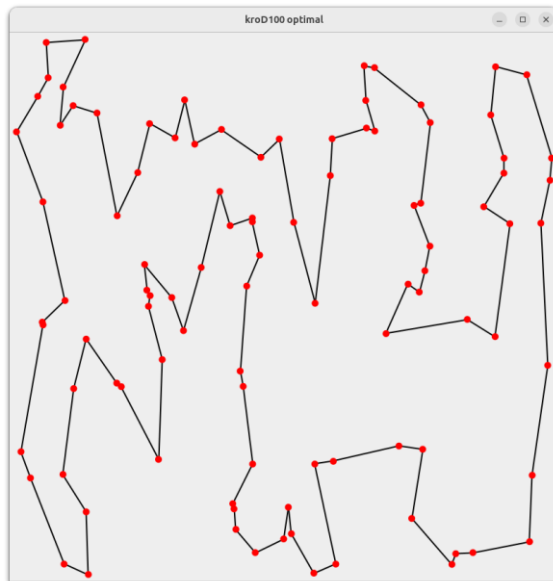
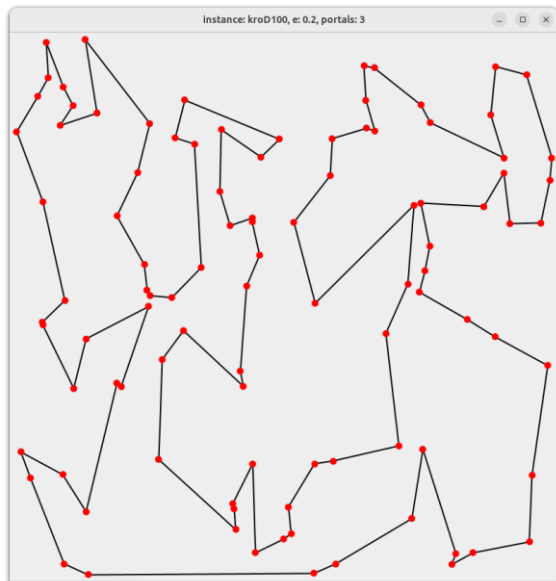
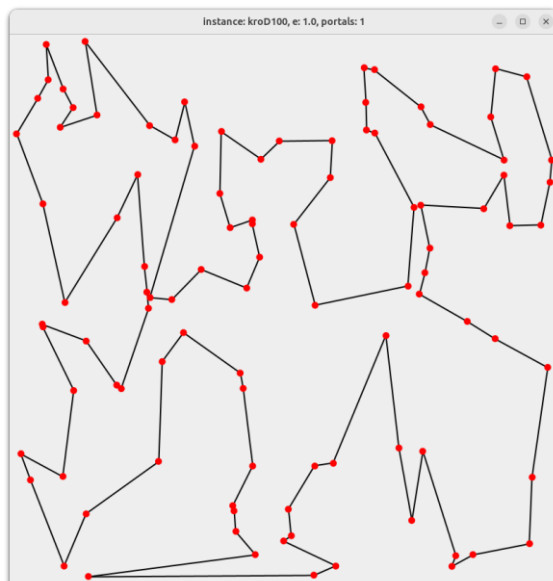


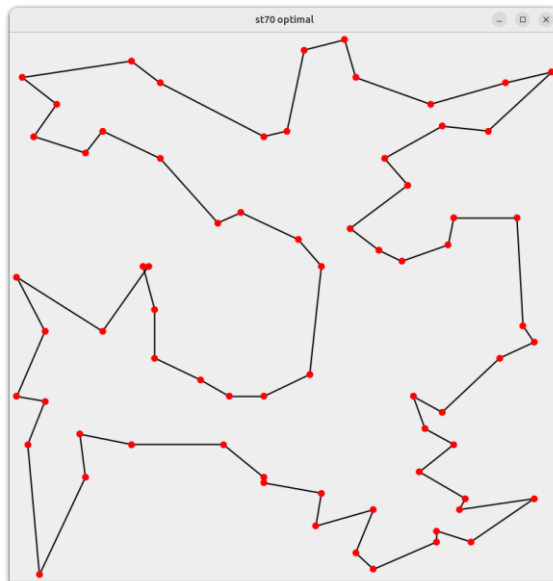
$e = 0.2, 4$ θύρες (124509 μ.)



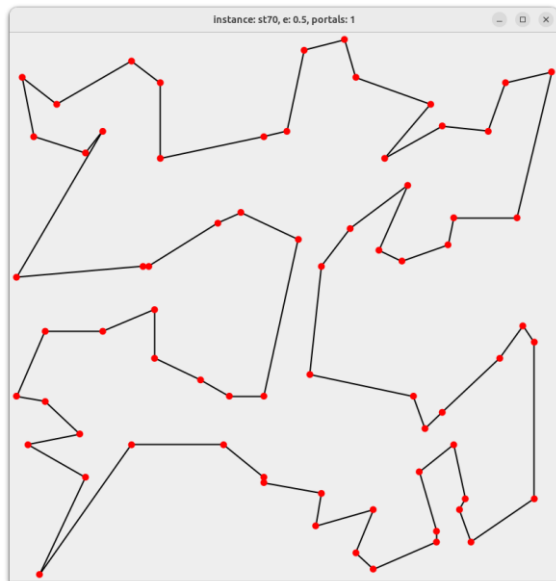
$e = 1, 4$ θύρες (125126 μ.)

ΕΙΚΟΝΑ 3.4.10: Στιγμιότυπο PR76

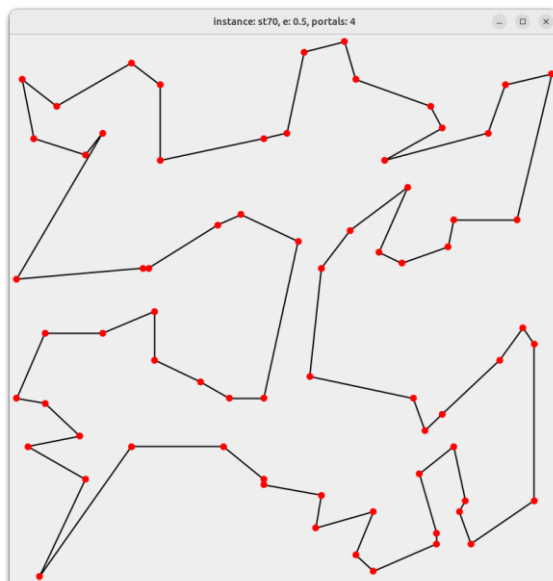
**Βέλτιστη Λύση (21294 μ.)** **$e = 0.2$, 3 θύρες (βέλτιστη προσέγγιση, 25607 μ.)** **$e = 1$, 1 θύρα (26550 μ.)** **$e = 0.5$, 2 θύρες (26563 μ.)****ΕΙΚΟΝΑ 3.4.11: Στιγμιότυπο KROD100**



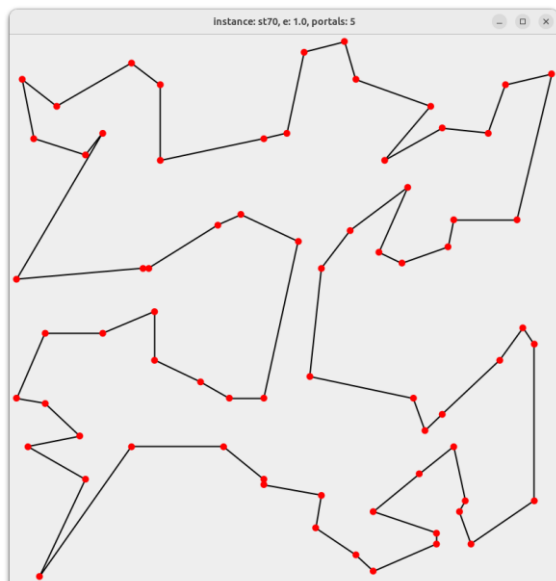
Βέλτιστη Λύση (675 μ.)



$e = 0.5$, 1 θύρα (βέλτιστη προσέγγιση, 760 μ.)



$e = 0.5$, 4 θύρες (762 μ.)



$e = 1$, 5 θύρες (761 μ.)

ΕΙΚΟΝΑ 3.4.12: Στιγμιότυπο ST70

4. k-TSP Αλγόριθμος Arora

Αρχικά θα ασχοληθούμε με τον σχεδιασμό ενός αλγορίθμου βασισμένου στις ιδέες του Arora, που να επιστρέφει κλειστά μονοπάτια (η αρχή ταυτίζεται με το τέλος) τα οποία διέρχονται όμως από k ακριβώς πόλεις, με το k να είναι ένας αριθμός από το 1 μέχρι το n . Θα ονομάσουμε αυτή την εκδοχή “k-TSP χωρίς άκρα”.

4.1 k-TSP χωρίς άκρα

Για την υλοποίησή μας θα χρειαστούν κάποιες προσαρμογές και προσθήκες στην υλοποίηση του κεφαλαίου 3. Καθώς η φάση της προεπεξεργασίας και του τετραδικού δέντρου δεν χρειάζονται κάποια αλλαγή, στο παρόν κεφάλαιο θα αναφερθούμε μόνο στην φάση εύρεσης μονοπατιών.

Έστω Q_T το σύνολο των πόλεων ενός κόμβου Q . Τότε έχοντας έναν οριστικοποιημένο κόμβο Q , τα τμήματά του θα είναι το σύνολο

$$SEG_Q = \begin{cases} \{AB \mid A, B \in \theta_Q, A \neq B\}, & |Q_T| = 0 \\ \{U\{AB, ATB\} \mid A, B \in \theta_Q, A \neq B\}, & |Q_T| = 1 \end{cases} \quad (4.1)$$

δηλαδή

- αν ο οριστικοποιημένος κόμβος δεν έχει πόλη, τα τμήματά του θα είναι αυτά ακριβώς που είχε και στο απλό TSP, δηλαδή όλα τα ευθύγραμμα τμήματα με άκρα δύο διαφορετικές θύρες του.
- αν ο οριστικοποιημένος κόμβος έχει πόλη, τα τμήματά του θα είναι παραπάνω από αυτά του απλού TSP. Συγκεκριμένα στα τμήματα του απλού TSP θα προσθέσουμε και όλα τα ευθύγραμμα τμήματα με άκρα τις θύρες του, δηλαδή θα θεωρούμε αποδεκτό και ένα τμήμα που αγνοεί την μοναδική πόλη του κόμβου.

Η προσθήκη αυτή μας επιτρέπει να διερχόμαστε από κάποιους κόμβους χωρίς να περνάμε από την πόλη τους, το οποίο σημαίνει ότι όπως σχηματίζουμε το μονοπάτι κατά τα γνωστά, ο τελικός αριθμός πόλεων του μονοπατιού μπορεί και να είναι μικρότερος από n .

Η διαδικασία ένωσης των τμημάτων είναι ίδια με το TSP. Αν θέλουμε τις λύσεις για όλα τα πιθανά k με μία εκτέλεση του αλγορίθμου, προσαρμόζουμε κατάλληλα την επιλογή λύσεων ώστε να κρατάμε για κάθε k το μονοπάτι με το ελάχιστο μήκος που διέρχεται από τόσες πόλεις.

```
void stitch(Nodes)
    node := Q1
    for segment in node.segments:
        entrance := segment.start
        Path := {segment}
        stitch := segment.end
        neighbour := unvisited node that contains stitch
        branch(entrance, stitch, neighbour, Path)
        remove segment from Path

void branch(entrance, stitch, node, path)
    for nextSegment in node.segments such that stitch in nextSegment:
        add nextSegment to Path
        if entrance is endpoint of nextSegment:
            k := |Path.towns|
            if solutions[k] !exist:
```

```

        solutions[k] = Path
    else if Path shorter than solutions[k]:
        solutions[k] = Path

    else
        newStitch := nextSegment endpoint not equal to stitch
        neighbour := unvisited node that contains newStitch
        branch(entrance, newStitch, neighbour, Path)
        remove nextSegment from Path

```

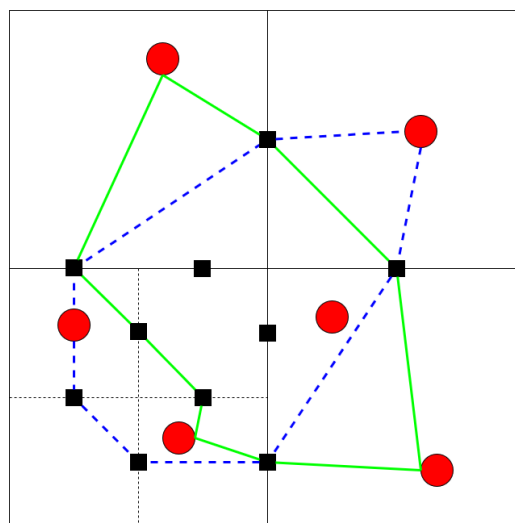
Στο παραπάνω κομμάτι ψευδοκώδικα βλέπουμε αυτή τη διαδικασία χωρισμένη στις δύο μεθόδους.

Αρχικά, στη μέθοδο stitch επιλέγουμε ένα τμήμα του αρχικού κόμβου και ορίζουμε το ένα του άκρο ως αρχή του μονοπατιού ενώ το άλλο ως συνδετικό κρίκο (stitch), σημείο επέκτασης με συνένωση κάποιου άλλου τμήματος. Ορίζουμε επίσης μία συλλογή Path η οποία θα κρατάει όλα τα τμήματα του μονοπατιού. Για να προχωρήσουμε σε αυτή την επέκταση βρίσκουμε τον άλλο κόμβο που περιέχει τον συνδετικό κρίκο, τον επονομαζόμενο γείτονα (neighbour). Έπειτα καλούμε την μέθοδο branch δίνοντας ως παραμέτρους την αρχή του μονοπατιού, τον συνδετικό κρίκο, τον γειτονικό κόμβο και το μονοπάτι.

Η μέθοδος branch επιλέγει τμήματα του γειτονικού κόμβου που αρχίζουν από τον συνδετικό κρίκο. Αν κάποιο από αυτά τα τμήματα έχει για άκρο την αρχή του μονοπατιού σημαίνει ότι έχουμε γυρίσει στο σημείο που αρχίσαμε και επομένως μπορούμε να προσθέσουμε το μονοπάτι στη συλλογή των λύσεων, αφού το συγκρίνουμε με κάποια ήδη αποθηκευμένη λύση που διέρχεται από ίδιο αριθμό πόλεων. Στην περίπτωση που το τρέχον τμήμα δεν έχει για άκρο την αρχή του μονοπατιού, θα θεωρήσουμε για νέο συνδετικό κρίκο το άκρο του τμήματος που δεν είναι ο προηγούμενος συνδετικός κρίκος και θα επαναλάβουμε την διαδικασία, αρχίζοντας την αναδρομή. Θα βρούμε τον νέο γειτονικό κόμβο, δηλαδή τον κόμβο που θα περιέχει τον νέο συνδετικό κρίκο, και θα καλέσουμε ξανά την μέθοδο branch με τις κατάλληλες παραμέτρους.

Η αναδρομή θα τερματίζεται πάντα το πολύ σε βάθος 4 κλήσεων, καθώς οι κόμβοι εμφανίζονται σε γειτονίες των τεσσάρων. Όταν ολοκληρώνεται η κάθε κλήση, δηλαδή συνεχίζει την εκτέλεσή της η μέθοδος branch μετά την γραμμή που καλεί τον εαυτό της, αφαιρούμε το τελευταίο τμήμα του τρέχοντος μονοπατιού για να δοκιμάσουμε μία επόμενη επιλογή.

Για την συγκεκριμένη εκδοχή του αλγορίθμου επιλέχθηκε να δοθεί μόνο το υποθετικό παράδειγμα της Εικόνας 4.1.1, αφού η εκδοχή με τα δοθέντα άκρα παρουσιάζει μεγαλύτερο ενδιαφέρον και αισθητή διαφοροποίηση από τα παραδείγματα του προηγούμενου κεφαλαίου.



ΕΙΚΟΝΑ 4.1.1: Πιθανά μονοπάτια για $k = 2$ (μπλε) και $k = 3$ (πράσινο)

4.2 k-TSP με Άκρα

Έχοντας υπολογίσει όλα τα πιθανά τμήματα των κόμβων Επιπέδου-1 με τον παραπάνω τρόπο, μπορούμε να προσαρμόσουμε την βέλτιστη k -διαδρομή δοθέντων μίας πόλης εκκίνησης $\tilde{S}$ και μίας πόλης τερματισμού $\tilde{E}$. Έστω QS ο Κόμβος Επιπέδου-1 που περιέχει την εκκίνηση και QE αυτός που περιέχει τον τερματισμό.

Συλλέγουμε κάθε Τμήμα $A\tilde{S}B$ του QS , με A, B θύρες του QS και το σπάμε σε δύο Τμήματα $\tilde{S}A$ και $\tilde{S}B$. Αρχίζοντας με το $\tilde{S}A$ βρίσκουμε τον γειτονικό Κόμβο που περιέχει τη θύρα A και έπειτα όλα τα τμήματά του που αρχίζουν απ' αυτή τη θύρα. Επαναλαμβάνουμε τη διαδικασία για κάθε τμήμα που βρήκαμε έτσι, ενώνοντας τα διαδοχικά τμήματα για να δημιουργήσουμε ένα μονοπάτι μέχρις ότου βρούμε ένα τμήμα που να περιέχει το $\tilde{E}$. Ανάλογη διαδικασία θα εκτελέσουμε έπειτα και για το $\tilde{S}B$ και θα κρατήσουμε τη λύση με το μικρότερο κόστος.

```
void stitch(Nodes)
    node := QS
    for segment in QS.segments, s~ in Segment:
        Path := {(s~, segment.start)}
        stitch := segment.start
        neighbour := unvisited node that contains stitch
        branch(segment, neighbour, Path)
        remove segment from Path
        Path := {(s~, segment.end)}
        stitch := segment.end
        neighbour := unvisited node that contains stitch
        branch(segment, neighbour, Path)

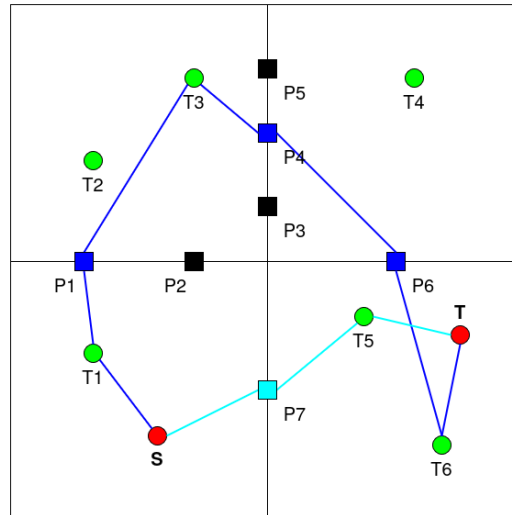
    remove segment from Path

void branch(segment, node, Path)
    stitch := segment.end
    for nextSegment in node.segments such that stitch in nextSegment:
        add nextSegment to Path
        if e~ in nextSegment:
            remove part after e~ from Path
            Solutions[|Path.towns|] = Path
        else
            newStitch := nextSegment endpoint not equal to stitch
            neighbour := unvisited node that contains newStitch
            branch(nextSegment, neighbour, Path)
    remove nextSegment from Path
```

Το παραπάνω τμήμα ψευδοκώδικα θυμίζει στη γενική ιδέα αυτό που είδαμε στην εκδοχή χωρίς άκρα, με κάποιες αναγκαίες αλλαγές ακριβώς για να λαμβάνει υπ' όψιν το γεγονός ότι πλέον δεν υπολογίζουμε κύκλους αλλά μονοπάτια ανάμεσα σε δύο ξεχωριστές πόλεις.

Θεωρούμε ως αρχικό κόμβο αυτόν που περιέχει την πόλη εκκίνησης και επιλέγουμε ένα τμήμα που όντως διέρχεται από αυτή την πόλη. Χωρίζουμε το τμήμα σε δύο κομμάτια, δηλαδή το αριστερά της πόλης και το δεξιά της πόλης, και αντιμετωπίζουμε το καθένα από αυτά με την λογική του αλγορίθμου της ενότητας 4.1. Οι μόνες διαφορές στην αναδρομή εμφανίζονται στη συνθήκη τερματισμού και στην

επεξεργασία του τελευταίου προς προσθήκη τμήματος. Αφού πλέον δεν ελέγχουμε κατά πόσο βρήκαμε ξανά το σημείο εκκίνησης αλλά το αν φτάσαμε επιτυχώς στην πόλη τερματισμού, η συνθήκη είναι ακριβώς το το τμήμα να περιέχει την πόλη τερματισμού. Στην περίπτωση που αυτό συμβαίνει, δε χρειάζεται να κρατήσουμε στην λύση ολόκληρο το τμήμα αλλά μόνο το κομμάτι του μέχρι την πόλη τερματισμού, μειώνοντας λίγο παραπάνω το συνολικό μήκος του μονοπατιού (Εικόνα 4.2.1).

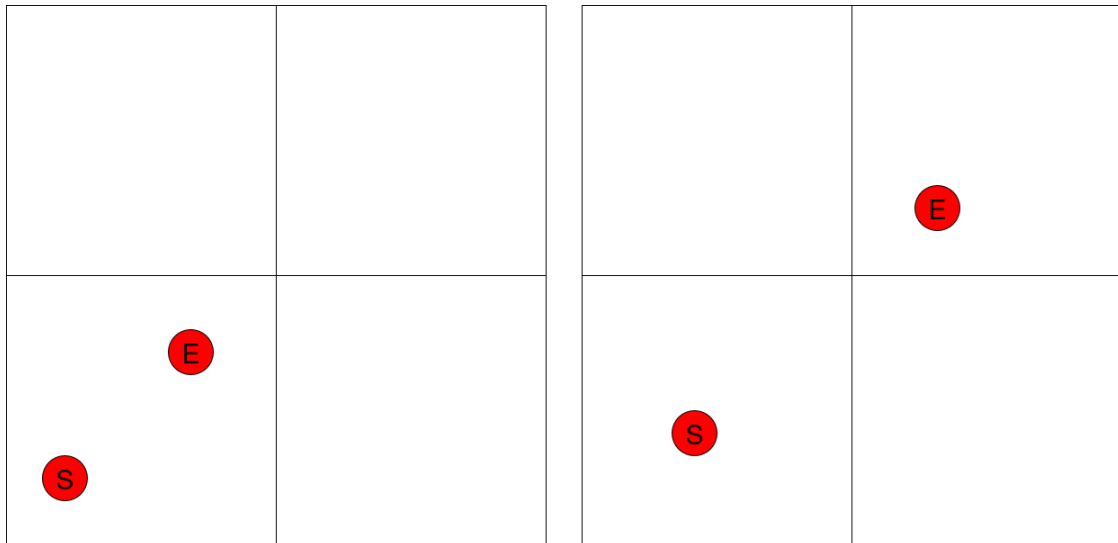


ΕΙΚΟΝΑ 4.2.1: Πιθανά k -TSP μονοπάτια από το S στο T για $k=3$ (γαλάζιο), $k=5$ (μπλε)

Η φύση του αλγορίθμου απαιτεί ειδική αντιμετώπιση στις δύο περιπτώσεις, οι οποίες φαίνονται στην Εικόνα 4.2.2:

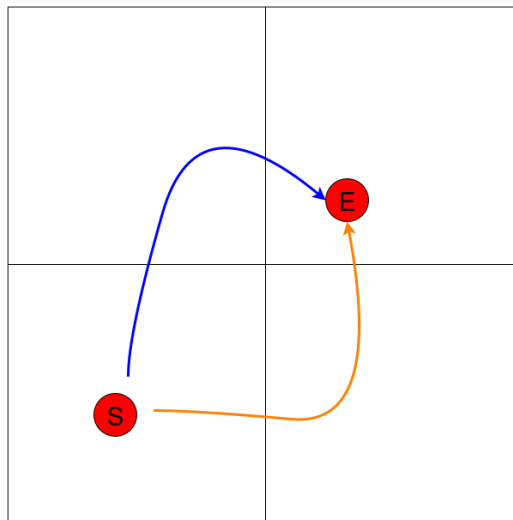
- οι $\tilde{S}, \tilde{E}$ ανήκουν στον ίδιο Κόμβο Επιπέδου-1
- οι $\tilde{S}, \tilde{E}$ ανήκουν σε Κόμβους Επιπέδου-1 που δεν έχουν καθόλου κοινές θύρες (απέναντι Κόμβοι)

Στην πρώτη περίπτωση, αν και τα k -TSP Μονοπάτια για $k \in \{2, \dots, k'\}$, $k' = |QS.towns|$ υπολογίζονται χωρίς κάποια περαιτέρω προσαρμογή και βρίσκονται εξ ολοκλήρου μέσα στον Κόμβο QS , κάθε μονοπάτι που θα περιέχει τμήματα άλλων κόμβων θα καταλήγει αναγκαστικά σε αδιέξοδο, καθώς η τελική συνθήκη της μεθόδου *branch* θα είναι πάντα ψευδής. Αν φύγουμε από τον αρχικό κόμβο, ποτέ δε θα βρούμε Τμήμα που να περιέχει την $\tilde{E}$. Για να επεκταθούμε και σε λύσεις που διέρχονται από περισσότερες από k' Πόλεις, προσαρτούμε την Πόλη $\tilde{E}$ στο τέλος κάθε τέτοιου αδιεξόδου (Εικόνα 4.2.4α).

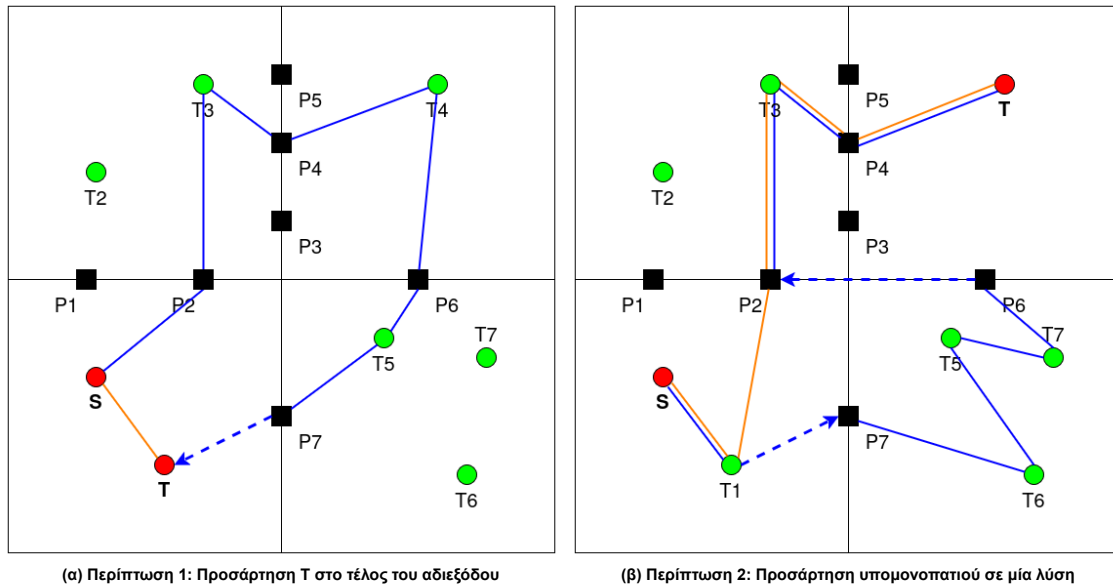


ΕΙΚΟΝΑ 4.2.2: Προβληματικές περιπτώσεις άκρων διαδρομής

Στη δεύτερη περίπτωση, έστω Q_0 , Q_2 οι απέναντι Κόμβοι (π.χ. κάτω-αριστερά και πάνω-δεξιά). Καθώς ο αλγόριθμος ενώνει Τμήματα που ανήκουν σε γειτονικούς κόμβους, ένα μονοπάτι θα φτάσει πάντα το $\tilde{E}$ χωρίς να προσπελάσει έναν από τους οριστικοποιημένους κόμβους και επομένως θα υπάρχει πάντα κάποιο k για το οποίο δε θα έχει βρεθεί k -TSP Μονοπάτι (Εικόνα 4.2.3). Έστω Q_3 ο κόμβος που δεν έχει προσπελαστεί (π.χ. κάτω-δεξιά).



ΕΙΚΟΝΑ 4.2.3: Όταν η εκκίνηση και ο τερματισμός βρίσκονται σε κόμβους χωρίς κοινές Θύρες, το μονοπάτι ολοκληρώνεται πριν την προσπέλαση και των 4 κόμβων



ΕΙΚΟΝΑ 4.2.4: Οπτικοποίηση των ειδικών περιπτώσεων

Για να υπολογίσουμε λύσεις που επισκέπτονται και τον Q_3 , σε κάθε υπάρχουσα λύση προσαρτούμε πριν από το Τμήμα του Κόμβου Q_2 τις βέλτιστες διαδρομές για να επισκεφθούμε τις $\{1, \dots, k'\}$, $k' = |Q_3.towns|$ Πόλεις του Q_3 . Έπειτα επαναλαμβάνουμε την προσάρτηση αυτών των βέλτιστων διαδρομών μετά το Τμήμα του Κόμβου Q_2 και, συγκρίνοντας τις δύο τελικές διαδρομές που προκύπτουν, κρατάμε την πιο σύντομη (Εικόνα 4.2.4β).

Με την παραπάνω επεξεργασία μπορούμε να βρούμε τα k-TSP Μονοπάτια για κάθε k σε κάθε περίπτωση. Είναι σημαντικό να αναφερθεί ότι η αντιμετώπιση αυτών των ειδικών περιπτώσεων δεν είναι κάτι που αναφέρεται στο άρθρο του Arora, το οποίο δίνει μόνο γενικές κατευθυντήριες γραμμές για τη μεταφορά του αρχικού αλγορίθμου και σε άλλα προβλήματα, συμπεριλαμβανομένου και του k-TSP. Επιπλέον, από τη στιγμή που οι διατυπώσεις του άρθρου ασχολούνται σε κάθε περίπτωση με κύκλους, είναι λογικό προβλήματα της παραπάνω φύσης να μην εμφανίζονται. Σε μελλοντική επέκταση της παρούσας εργασίας θα μπορούσαν να διερευνηθούν και πιο αποδοτικοί ή ευφάνταστοι τρόποι για να ληφθούν υπ' όψιν οι ειδικές περιπτώσεις κατά τη διαδικασία σύνθεσης του μονοπατιού στο k-TSP με άκρα.

4.3 Αποτελέσματα

Ακολουθούν τα ευρήματα της υλοποίησής μας σε μερικά από τα στιγμιότυπα του προβλήματος TSP του προηγούμενου κεφαλαίου, με την επιπλέον προσθήκη της παραμέτρου k. Για κάθε παράδειγμα επιλέχθηκαν αυθαίρετα για σημεία εκκίνησης και λήξης οι πόλεις με τη μικρότερη και τη μεγαλύτερη τετμημένη αντίστοιχα. Σε κάθε Παράδειγμα θα παρουσιάζονται οι λύσεις για 4 διαφορετικές επιλογές του k, συγκεκριμένα για $k \in \left\{ \left\lfloor \frac{n}{4} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \left\lceil 3 \frac{n}{4} \right\rceil, n \right\}$.

Επιπλέον, καθώς ο χρόνος εκτέλεσης γίνεται αισθητά μεγαλύτερος στο πρόβλημα του k-TSP, αρχίζουμε να κάνουμε χρήση των λεγόμενων cache αρχείων στις εκτελέσεις. Δηλαδή αποθηκεύουμε σε απλά αρχεία (txt) τις σημαντικές πληροφορίες μετά από ένα επιτυχές τρέξιμο (αριθμός πόλεων, αλληλουχία πόλεων, μήκη διαδρομών) έτσι ώστε να μπορούμε να ξαναδούμε τα αποτελέσματα χωρίς να χρειάζεται να περιμένουμε να ολοκληρωθεί ξανά η εκτέλεση. Ενδεικτικά παραθέτουμε σχετικά κομμάτια κώδικα.

```
Points := {points of the instance}
```

```
validNumbers := [(Points.size() / 4), Points.size() / 2, 3 * Points.size() / 4, Points.size()]
```

```

static void readCacheOrSolve() {
    try (BufferedReader br := new BufferedReader(new FileReader("example.cache"))) {
        return readKTspCacheFile(br, Points)
    } catch (IOException ex) {
        // No cache file exists
        Run the algorithm
    }
}

static int readKTspCacheFile(BufferedReader br, Points) {
    String line := br.readLine();
    while (line != null) {
        // Solution parts in cache separated by spaces
        String[] segments := line.split(" ")
        int pointNumber := Integer.parseInt(segments[0])
        if (pointNumber in validNumbers) {
            // Extract the coordinate numbers by manipulating the segment with the points
            String[] pointsToDraw := segments[1].split("\\\\,");
            ArrayList<Point> solutionPoints := new ArrayList<>();
            for (String point : pointsToDraw) {
                String[] coords := point.split("\\(")[1].split("\\)")[0].split(",");
                nextPoint := new Point(coords[0], coords[1])
                solutionPoints.add(nextPoint);
            }
            Solution kTspSolution := new Solution(segments[2], solutionPoints);
            plotKTSPORienteeringGraphs(kTspSolution);
        }
        line := br.readLine();
    }
    return 0;
}

```

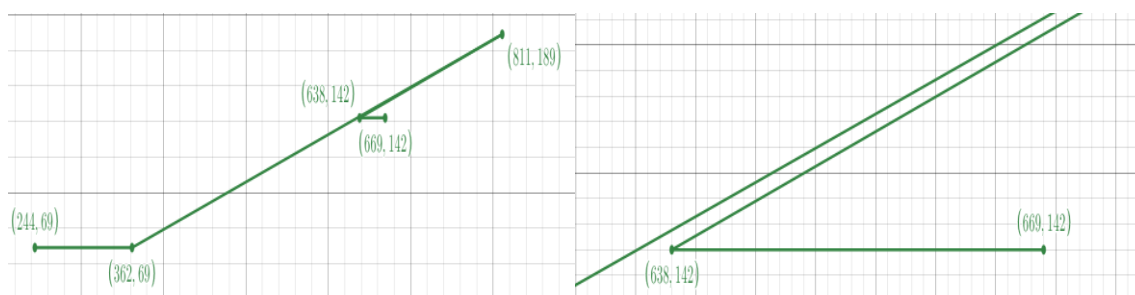
Όπως βλέπουμε παραπάνω, πριν την εκτέλεση του αλγορίθμου ελέγχουμε αν υπάρχει αποθηκευμένο ήδη κάποιο αρχείο cache. Αν δε βρεθεί κάποιο, ο αλγόριθμος εκτελείται κανονικά, βάσει όσων περιγράψαμε στο Κεφάλαιο. Σε αντίθετη περίπτωση, διαβάζουμε κάθε γραμμή του αρχείου και ελέγχουμε αν το πρώτο τμήμα της, στο οποίο αναγράφεται ο αριθμός των πόλεων από τις οποίες διέρχεται η εκάστοτε διαδρομή, αντιστοιχεί σε κάποιον από τους αριθμούς πόλεων των οποίων θέλουμε να αναπαραστήσουμε τη διαδρομή τους γραφικά (validNumbers).

2	(2.0,48.0), (57.0,48.0)	55.0
3	(2.0,48.0), (56.0,39.0), (57.0,48.0)	63.8002479108214
4	(2.0,48.0), (26.0,35.0), (56.0,39.0), (57.0,48.0)	66.61556516689289
5	(2.0,48.0), (10.0,43.0), (26.0,35.0), (56.0,39.0), (57.0,48.0)	66.643
6	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (56.0,39.0), (57.0,	
7	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (35.0,35.0), (56.0,	
8	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0)	Μήκος διαδρομής
9	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (35.0,35.0), (56.0,	
10	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (35.0,35.0), (41.0	
11	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (35.0,35.0), (41.0	
12	(2.0,48.0), (10.0,43.0), (17.0,34.0), (26.0,35.0), (35.0,35.0), (41.0	
13	(2.0,48.0), (10.0,43.0), (14.0,37.0), (17.0,34.0), (35.0,35.0), (25.0	

ΕΙΚΟΝΑ 4.3.1: Τμήμα cache αρχείου για το πρόβλημα EIL101

Κρατώντας μόνο τις επιθυμητές λύσεις, εξάγουμε με κατάλληλη επεξεργασία (*τμηματικό χώρισμα στις παρενθέσεις και στα κόμματα*) τις συντεταγμένες από το δεύτερο τμήμα τους. Στην Εικόνα 4.3.1 φαίνεται μέρος ενός cache αρχείου με σημειωμένα τα ξεχωριστά τμήματα.

Τέλος, σε κάποια από τα γραφήματα ενδέχεται λόγω κλίμακας μέρη της διαδρομής να φαίνεται ότι τέμνονται. Σε αυτές τις περιπτώσεις έγινε έλεγχος της αλληλουχίας των σημείων όπως αυτά είχαν αποθηκευτεί στα cache αρχεία και διαπιστώθηκε ότι δε συμβαίνει κάτι τέτοιο και απλά οι συντεταγμένες είναι αρκετά κοντινές, ώστε το πάχος γραμμής ή το μέγεθος των κύκλων που αντιπροσωπεύουν τις πόλεις να δημιουργεί λανθασμένες εντυπώσεις. Παραθέτουμε ενδεικτικά ένα μέρος της διαδρομής LIN105 για 25 πόλεις. Στην λύση μας το συγκεκριμένο τμήμα εμφανίζεται ανάποδα (*συμμετρικό ως προς χ'χ'*) αφού η αρχή των αξόνων στο παραθυρικό περιβάλλον της Java είναι πάνω αριστερά.



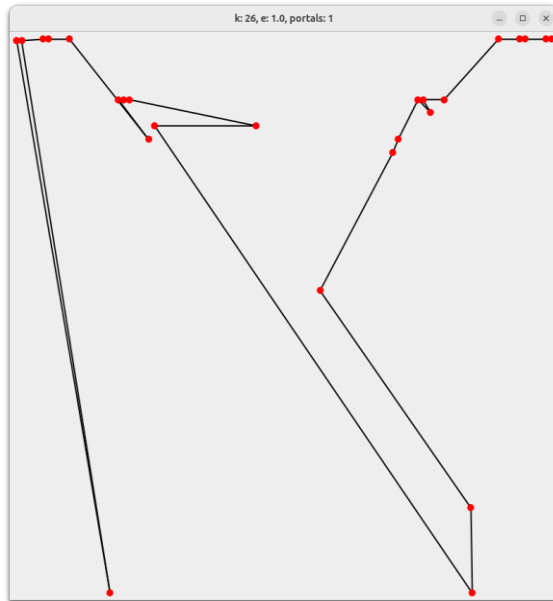
ΕΙΚΟΝΑ 4.3.2: Μεγέθυνση σε επίφοβο κομμάτι της διαδρομής LIN105, k=25.

Ακολουθεί πίνακας με τα μήκη διαδρομών ανά αριθμό πόλεων για τα παραδείγματα TSP στα οποία εφαρμόστηκε και το k-TSP, καθώς και τα σχετικά γραφήματα.

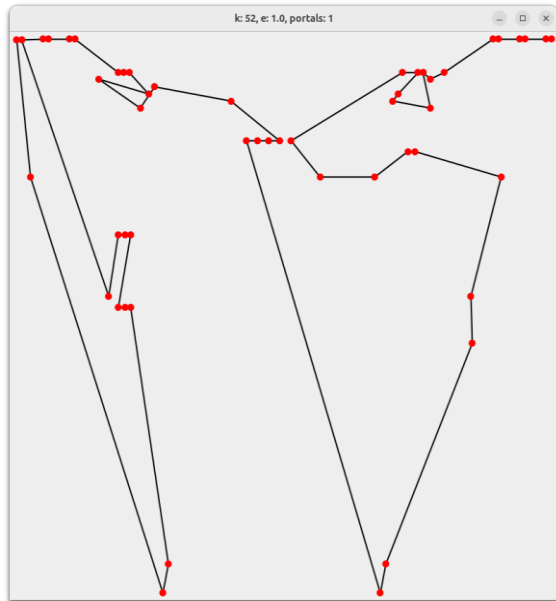
ΠΙΝΑΚΑΣ 2: Προσεγγίσεις για το k-TSP

	N/4 πόλεις	N/2 πόλεις	3N/4 πόλεις	N πόλεις
LIN105	8255 μ.	11184 μ.	13443 μ.	18412 μ.

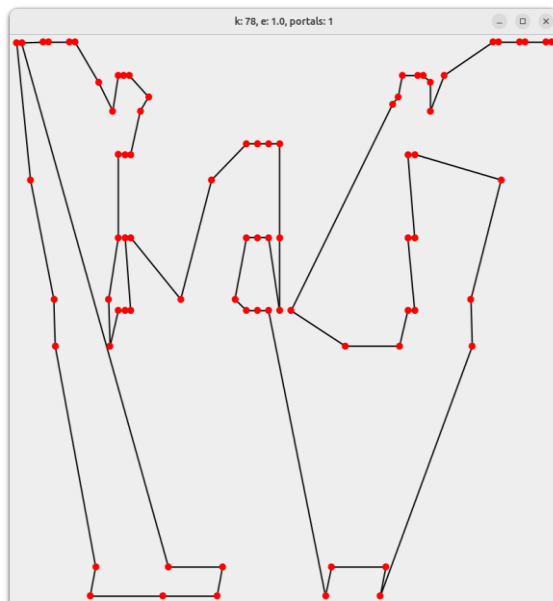
BERLIN52	4276 μ.	5075 μ.	6179 μ.	8573 μ.
EIL101	167 μ.	360 μ.	554 μ.	794 μ.
ST70	207 μ.	487 μ.	677 μ.	852 μ.
PR76	40544 μ.	85229 μ.	108297 μ.	135871 μ.
KROA100	7598 μ.	13322 μ.	21312 μ.	27763 μ.
KROC100	11915 μ.	15716 μ.	20516 μ.	25842 μ.
KROD100	12958 μ.	15314 μ.	20445 μ.	29558 μ.



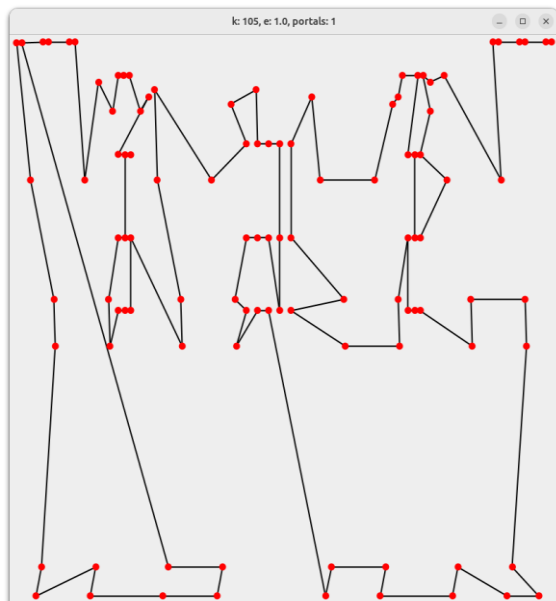
k=26



k=52

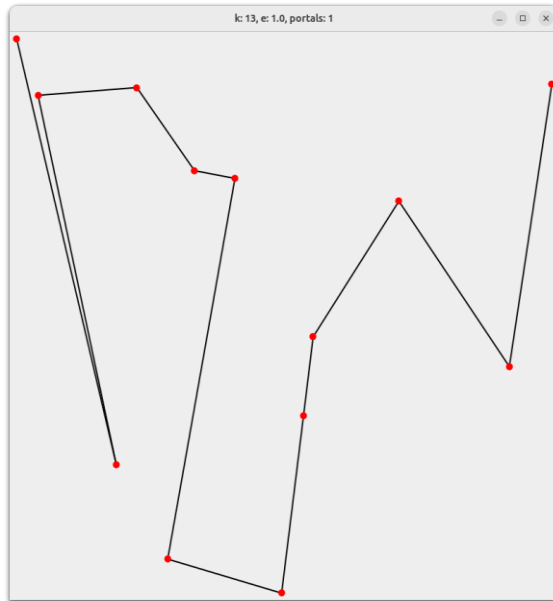


k=78

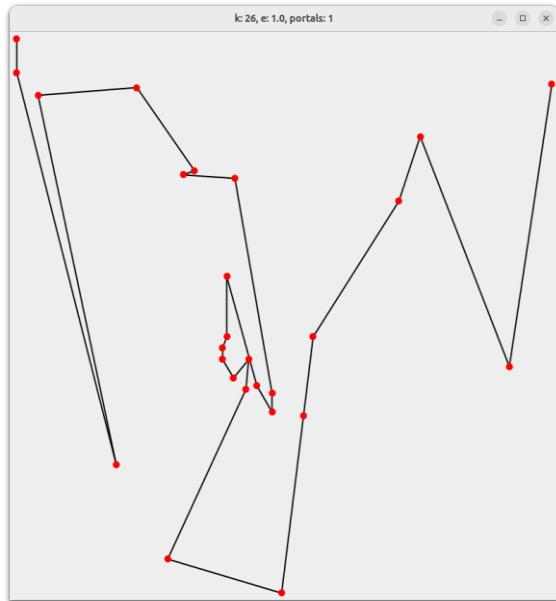


k=105

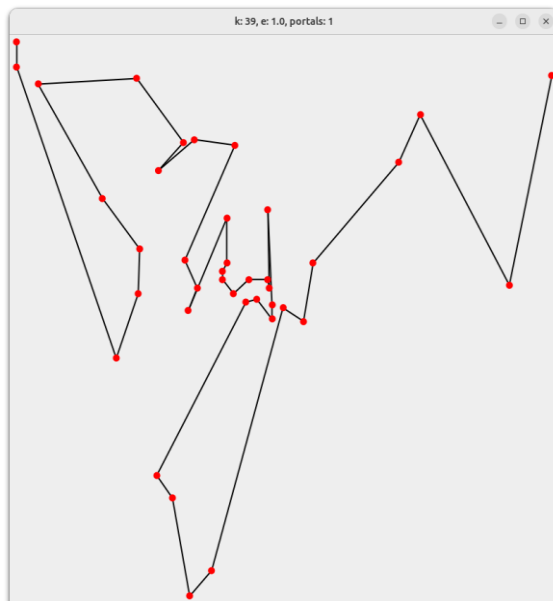
ΕΙΚΟΝΑ 4.3.1: Στιγμιότυπο LIN105



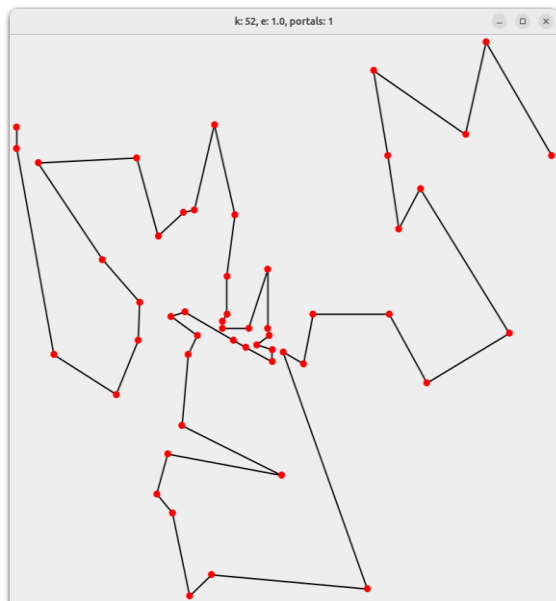
k=13



k=26

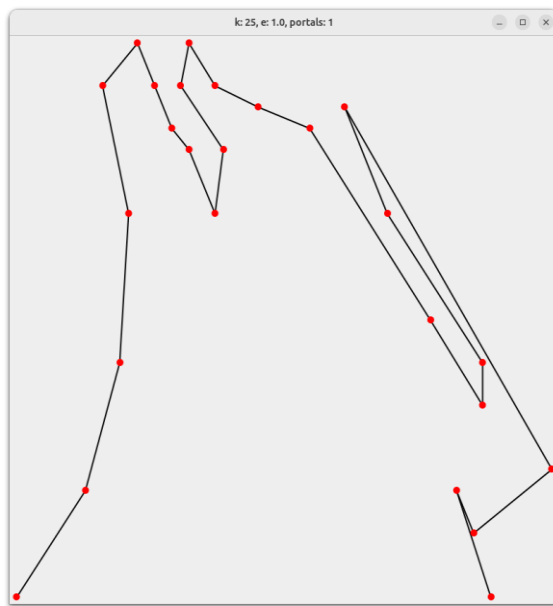


k=39



k=52

ΕΙΚΟΝΑ 4.3.2: Στιγμιότυπο BERLIN52



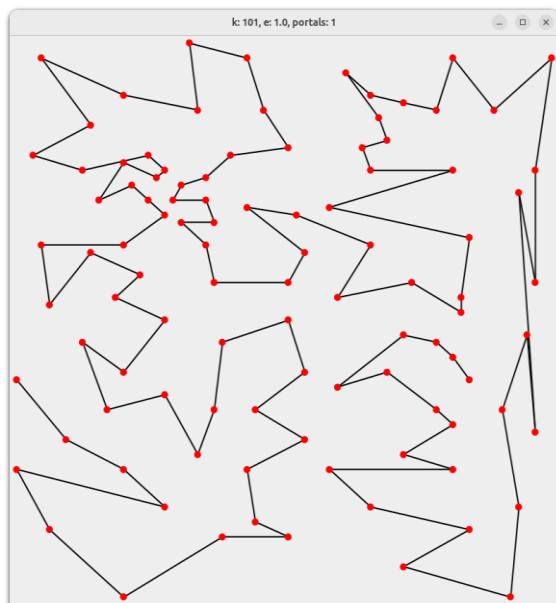
k=25



k=50



k=75

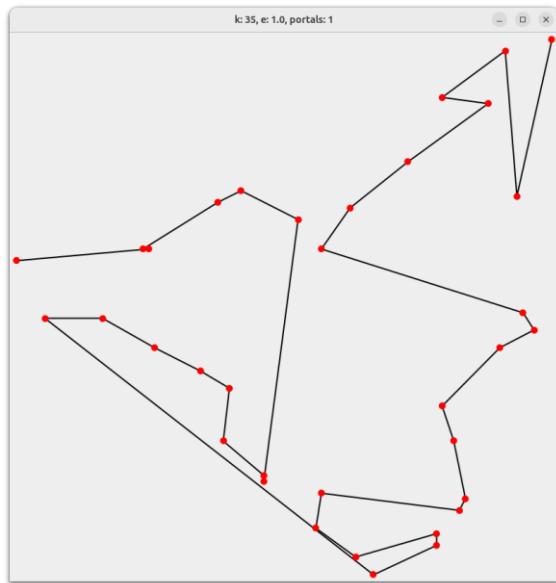


k=101

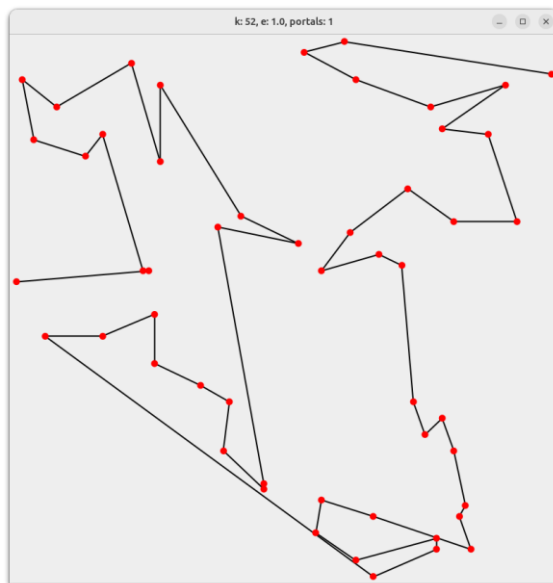
ΕΙΚΟΝΑ 4.3.3: Στιγμιότυπο EIL101



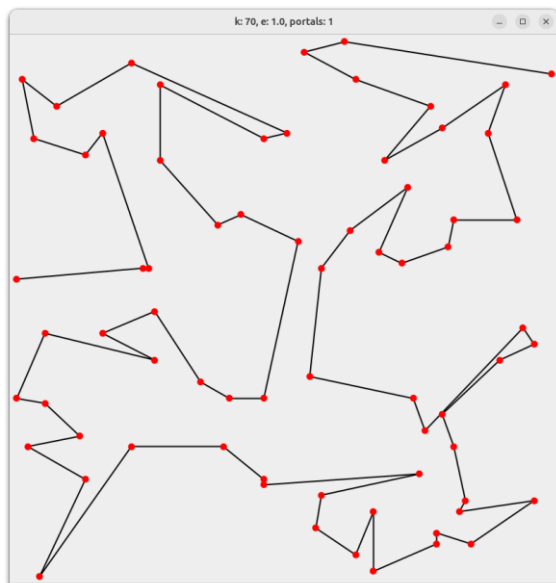
k=17



k=35

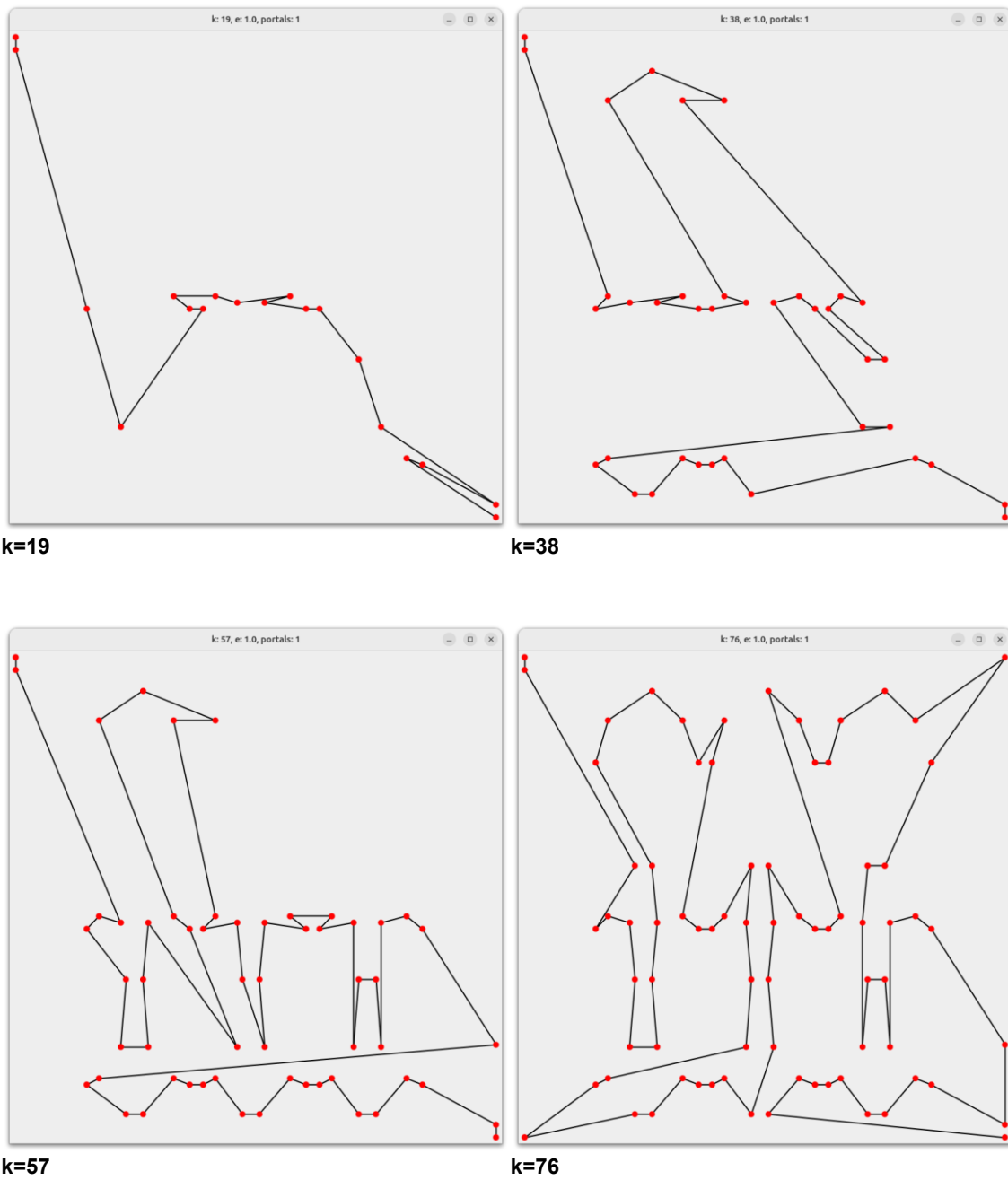


k=52

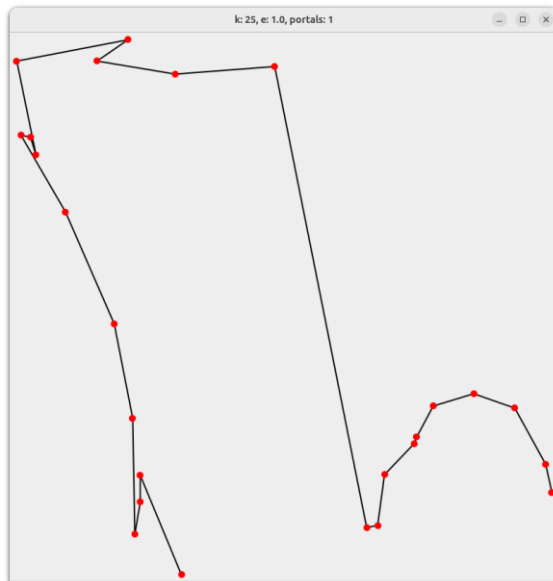
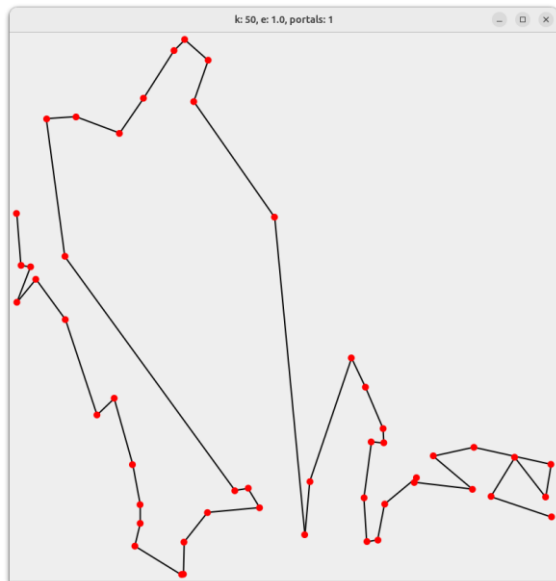
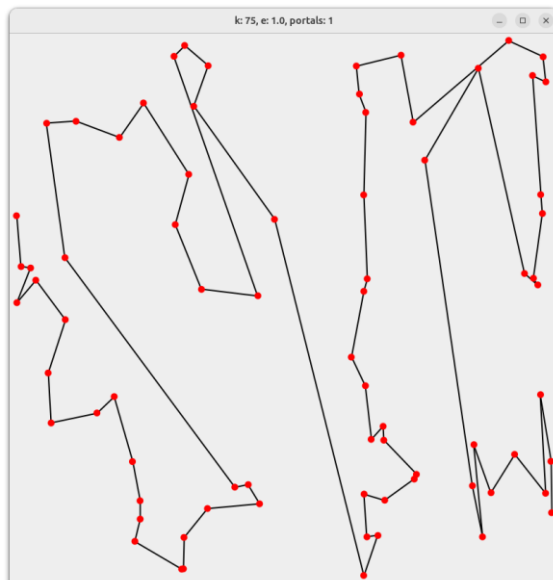
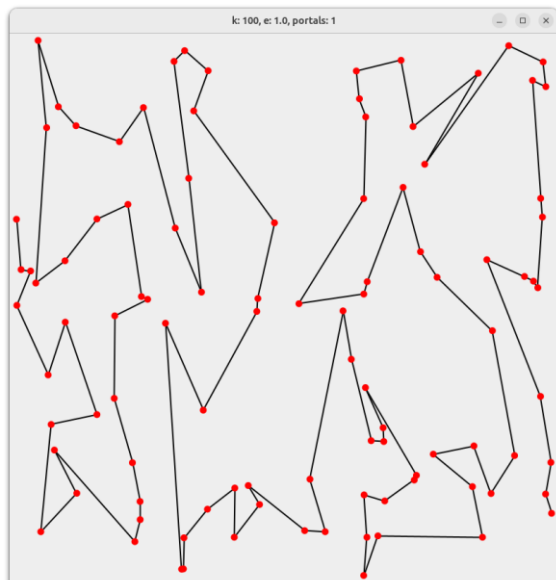


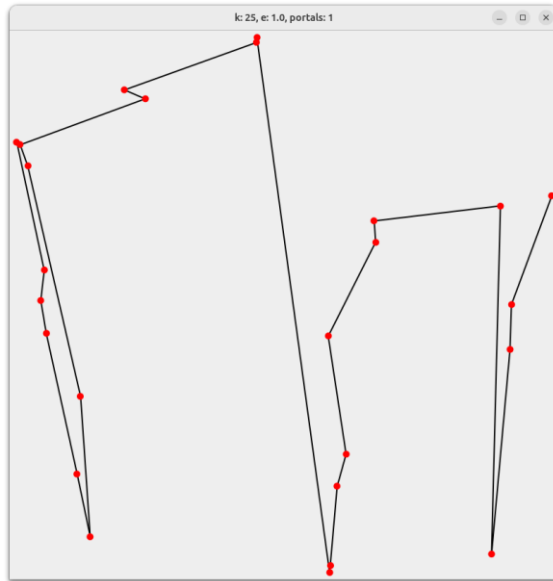
k=70

ΕΙΚΟΝΑ 4.3.4: Στιγμιότυπο ST70

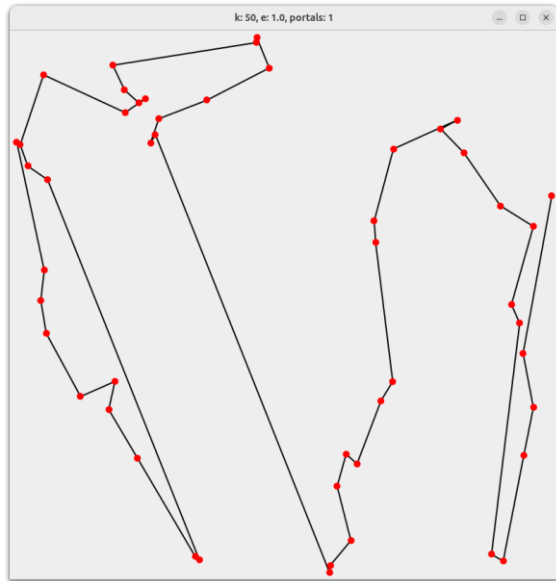


ΕΙΚΟΝΑ 4.3.5: Στιγμιότυπο PR76

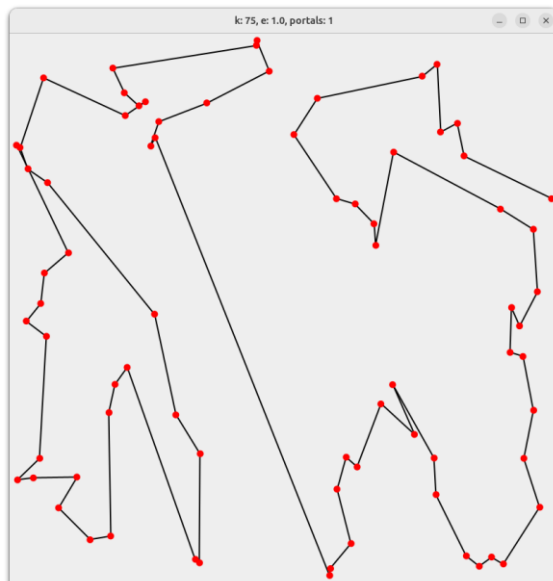
**k=25****k=50****k=75****k=100****ΕΙΚΟΝΑ 4.3.6: Στιγμιότυπο ΚΡΟΑ100**



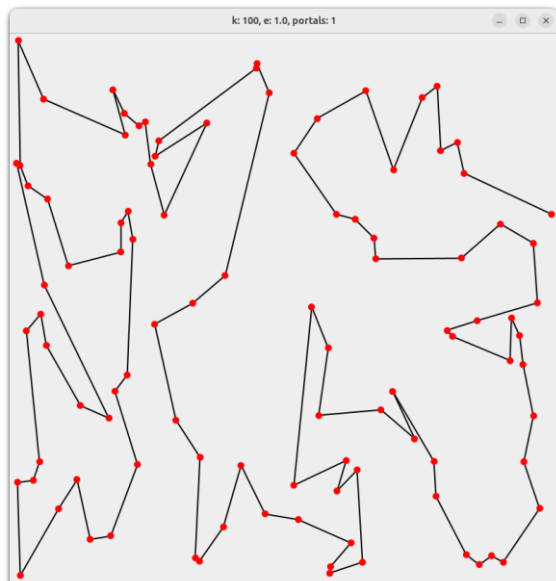
k=25



k=50

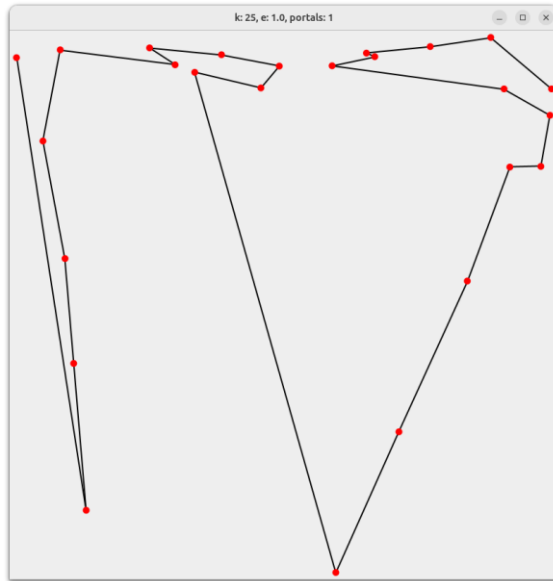


k=75



k=100

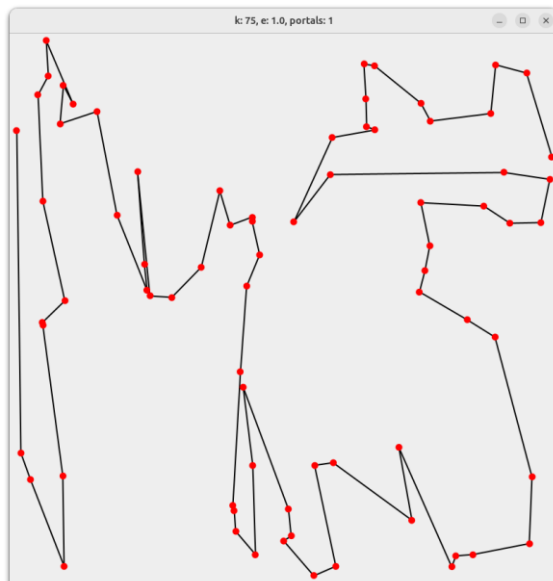
ΕΙΚΟΝΑ 4.3.7: Στιγμιότυπο KROC100



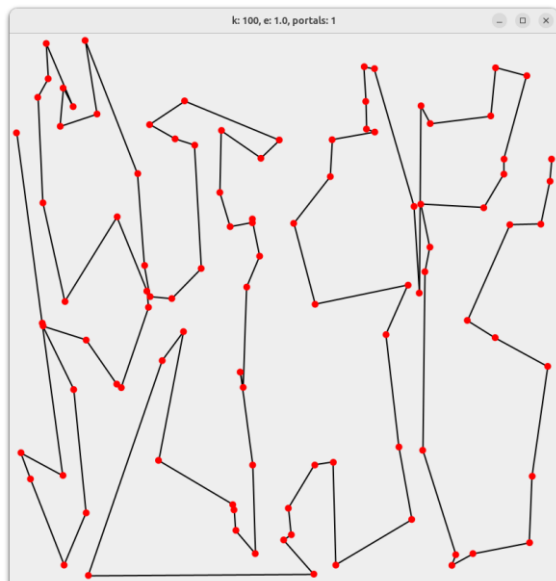
k=25



k=50



k=75



k=100

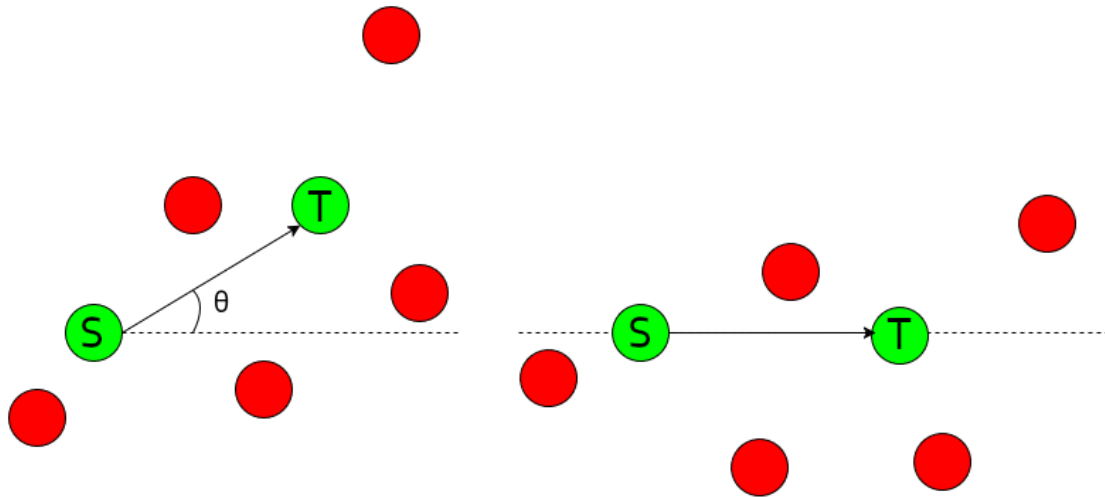
ΕΙΚΟΝΑ 4.3.8: Στιγμιότυπο KROD100

5. Βελτιώσεις βασισμένες στον αλγόριθμο Gottlieb

Βασιζόμενοι στην υλοποίηση του αλγορίθμου Arora για το πρόβλημα k-TSP με γνωστά άκρα, μπορούμε να προχωρήσουμε σε κάποιες βελτιώσεις βασισμένες στην έρευνα των Gottlieb et al[18], στη βάση της υπόθεσης ότι ο πιο αναλυτικός τρόπος εύρεσης μονοπατιών που περιγράφεται εκεί θα οδηγήσει σε καλύτερες προσεγγίσεις. Εκεί, χρησιμοποιώντας ως υπορουτίνα τον αλγόριθμο του Arora, επιστρέφεται μία λύση για το πρόβλημα του Προσανατολισμού, δηλαδή η λύση με τη μεγαλύτερη συνολική βαθμολογία η οποία τηρεί ένα δοθέν χρονικό όριο. Στο πλαίσιο της εργασίας θα επικεντρωθούμε σε αυτή την υπορουτίνα με απώτερο σκοπό να επιτύχουμε καλύτερα αποτελέσματα στην υλοποίηση του k-TSP και επομένως η επίλυση του Προσανατολισμού θεωρείται εκτός των ορίων της έρευνάς μας.

5.1 Φάση Προεπεξεργασίας

Σε αντιστοιχία με τον αλγόριθμο του Arora, και στον αλγόριθμο του Gottlieb χρειάζεται μία αρχική προεπεξεργασία πριν υπολογίσουμε τις πιθανές λύσεις στο σύνολο των μετασχηματισμένων παραδειγμάτων που προέκυψαν από τον αλγόριθμο του Korula. Στη συγκεκριμένη περίπτωση αυτό που μας ενδιαφέρει είναι η ευθεία που ενώνει το σημείο εκκίνησης και λήξης να είναι οριζόντια, επομένως πρέπει να περιστρέψουμε κατάλληλα το επίπεδο (Εικόνα 5.1.1).



ΕΙΚΟΝΑ 5.1.1: Περιστροφή του επιπέδου

Για να το επιτύχουμε θεωρούμε το σημείο εκκίνησης S ως αρχή των αξόνων και υπολογίζουμε τη γωνία, έστω θ , που σχηματίζει με τον άξονα των τετμημένων το σημείο λήξης T. Ορίζουμε ως νέες συντεταγμένες μίας πόλης (x, y) τις

$$(x', y'): \begin{cases} x' = \cos(\theta) \cdot x - \sin(\theta) \cdot y \\ y' = \cos(\theta) \cdot y + \sin(\theta) \cdot x \end{cases} \quad (5.1)$$

```
void rotateOnEndpointAxis(Towns):
    // Ορίζουμε την εκκίνηση ως αρχή των αξόνων
    for town in Towns:
        town.x := town.x - start.x
        town.y := town.y - start.y
    // Υπολογίζουμε τη γωνία
    cosSign := if end.x >= 0 then 1 else -1
```

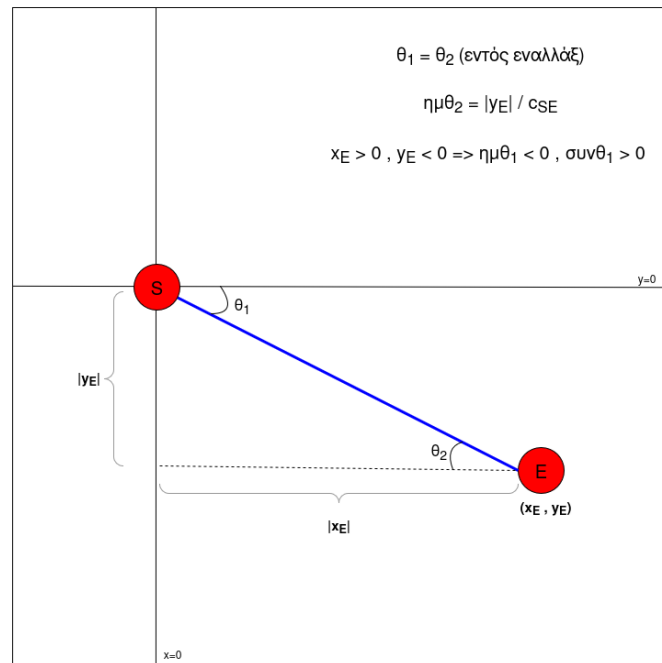
```

sinSign := if end.y <= 0 then 1 else -1
sin = sinSign * (absoluteValue(end.y) / squareRoot(end.x ^ 2 + end.y ^ 2));
cos = cosSign * squareRoot(1 - sin ^ 2);
// Περιστρέφουμε τα σημεία
for town in Towns:
    town.x := town.x * cos - town.y * sin
    town.y := town.y * cos + town.x * sin

```

Το μόνο μέρος του ψευδοκώδικα που ίσως να χρειάζεται μια ειδική αναφορά είναι ο υπολογισμός της γωνίας περιστροφής. Έχοντας μεταφέρει το σημείο εκκίνησης στην αρχή των αξόνων μπορούμε βάσει των συντεταγμένων του σημείου λήξης να γνωρίζουμε το άνοιγμα της γωνίας (1ο, 2ο, 3ο ή 4ο τεταρτημόριο) και άρα το πρόσημο του ημιτόνου και του συνημιτόνου. Η τιμή του μπορεί να υπολογιστεί εύκολα με Πυθαγόρειο θεώρημα, κάνοντας τις κατάλληλες προβολές (Εικόνα 5.1.2). Ο λόγος που προτιμήθηκε αυτή η αντιμετώπιση από μία αναλυτικής γεωμετρίας είναι γιατί παρατηρήθηκε ότι ήταν λιγότερο ευάλωτη σε σφάλματα στρογγυλοποίησης που προέκυπταν από τους αναλυτικούς τύπους.

Αφότου ταξινομήσουμε τα σημεία σε αύξουσα σειρά τετμημένων, μπορούμε να εφαρμόσουμε το βασικό μέρος του αλγορίθμου.



ΕΙΚΟΝΑ 5.1.2: Υπολογισμός γωνίας περιστροφής

5.2 Δυναμικός προγραμματισμός

Αρχικά θα λέμε ότι μία πόλη είναι μικρότερη από κάποια άλλη αν η τετμημένη της είναι μικρότερη από την δεύτερη. Δηλαδή

$$\tilde{i} = (x_i, y_i), \tilde{j} = (x_j, y_j) : \tilde{i} \leq \tilde{j} \Leftrightarrow x_i \leq x_j \quad (5.2)$$

Επιπλέον με τον συμβολισμό $\tilde{i}$, $1 \leq i \leq n$ θα εννοούμε την i -οστή πόλη βάσει της ταξινόμησής τους κατά την προεπεξεργασία που παρουσιάστηκε στην προηγούμενη ενότητα. Για παράδειγμα με $\tilde{1}$ θα αναφερόμαστε στην πρώτη πόλη, δηλαδή την πόλη με την μικρότερη τετμημένη. Επίσης υπενθυμίζουμε

ότι έχουμε ορίσει ως παράθυρο $W_{\tilde{l}, \tilde{j}}$ δύο προορισμών $\tilde{l}, \tilde{j}$ το σύνολο όλων των προορισμών που περιέχονται μεταξύ τους. Τέλος με $Arora(\tilde{l}, \tilde{j}, \tilde{x}, \tilde{y}, k)$ θα αναπαριστούμε την λύση του k-TSP αλγορίθμου Arora με άκρα $\tilde{x}, \tilde{y}$ στο παράθυρο $W_{\tilde{l}, \tilde{j}}$ (προφανώς $\tilde{l} \leq \tilde{x}, \tilde{y} \leq \tilde{j}$) και με τον δείκτη Gottlieb $G(\tilde{l}, \tilde{j}, k)$ θα περιγράψουμε μία διαδρομή η οποία

- αρχίζει από την πόλη εκκίνησης s
- καταλήγει στην Πόλη $\tilde{j}$
- διέρχεται από k πόλεις
- όλες οι πόλεις από τις οποίες διέρχεται είναι μικρότερες της $\tilde{l}$

Θεωρούμε τη συλλογή V στην οποία αποθηκεύουμε ζευγάρια δεικτών Gottlieb με διαδρομές. Για να βάλουμε κάποιες αρχικές τιμές στην συλλογή αυτή, αποθηκεύουμε τα ζεύγη

$$\{ G(s, \tilde{l}, k), Arora(\tilde{l}, s, s, \tilde{l}, k) \} \forall \tilde{l}: \tilde{l} < s \forall k \in \{2, \dots, |W_{1, \tilde{l}}|\} \quad (5.3)$$

Στην παραπάνω σχέση:

- ο συμβολισμός $G(s, \tilde{l}, k)$ είναι ο δείκτης Gottlieb και αναπαριστά μία διαδρομή η οποία αρχίζει από την πόλη εκκίνησης s , καταλήγει στην πόλη $\tilde{l}$ και διέρχεται από k πόλεις, οι οποίες είναι μικρότερες της s . Δηλαδή σηματοδοτεί μία διαδρομή που διέρχεται από πόλεις που είναι πριν από το δοθέν σημείο εκκίνησης. Στον κώδικα αναγράφεται ως GottliebKey.
- ο συμβολισμός $Arora(\tilde{l}, s, s, \tilde{l}, k)$ είναι ο δείκτης Arora και αναπαριστά τη λύση του k-TSP με βάση τον αλγόριθμο του Arora, με άκρα τα s και $\tilde{l}$, η οποία διέρχεται από k πόλεις που βρίσκονται ανάμεσα στο σημείο εκκίνησης και την s . Στον κώδικα αναγράφεται ως AroraKey.

Επομένως η σχέση αυτή αντιστοιχεί τις διαδρομές που διατρέχουν k πόλεις πριν την εκκίνηση με τις λύσεις του k-TSP των ίδιων άκρων. Για παράδειγμα η διαδρομή που θα αντιστοιχεί στον δείκτη $G(s, \tilde{l}, 3)$ στην *Εικόνα 5.2.1* θα είναι η διαδρομή που αρχίζει από το σημείο εκκίνησης, καταλήγει στην πρώτη πόλη 1, η οποία έχει την μικρότερη τετμημένη, και διέρχεται από 3 πόλεις. Υπολογίζεται εφαρμόζοντας τον αλγόριθμο k-TSP του Arora για $k=3$ και άκρα s και $\tilde{l}$, στο παράθυρο $W_{\tilde{l}, s}$. Για αυτόν τον λόγο, όπως φαίνεται και στην παραπάνω σχέση, το k σε αυτή την περίπτωση μπορεί να έχει τιμή το πολύ $|W_{\tilde{l}, s}|$, δηλαδή όσο είναι το πλήθος των πόλεων πριν και την εκκίνηση.

Με τους παραπάνω υπολογισμούς εξασφαλίζουμε ότι ακόμα και τα μονοπάτια που θα περιέχουν πόλεις μικρότερες της πόλης εκκίνησης θα αρχίζουν από τη σωστή εκκίνηση.

```
// To solutions είναι HashMap που έχει τις τελικές λύσεις αντιστοιχισμένες
```

```
// με τον κατάλληλο δείκτη Gottlieb.
```

```
HashMap solutions{[key: GottliebKey] : [value: solution]}
```

```
void iteratePointsBeforeStart(start, Points):
```

```
    startIndex := Points.indexOf(start)
```

```
    // Αρχικά μας απασχολούν μόνο οι πόλεις πριν την εκκίνηση.
```

```
    InitialWindow := points[0, ..., startIndex]
```

```
    for point in InitialWindow:
```

```
        // Στο S κρατάμε κάθε φορά τις λύσεις του k-TSP με άκρα τα έναρξη και point,
```

```
        // οι οποίες βρίσκονται στο παράθυρο από την πρώτη πόλη μέχρι την έναρξη.
```

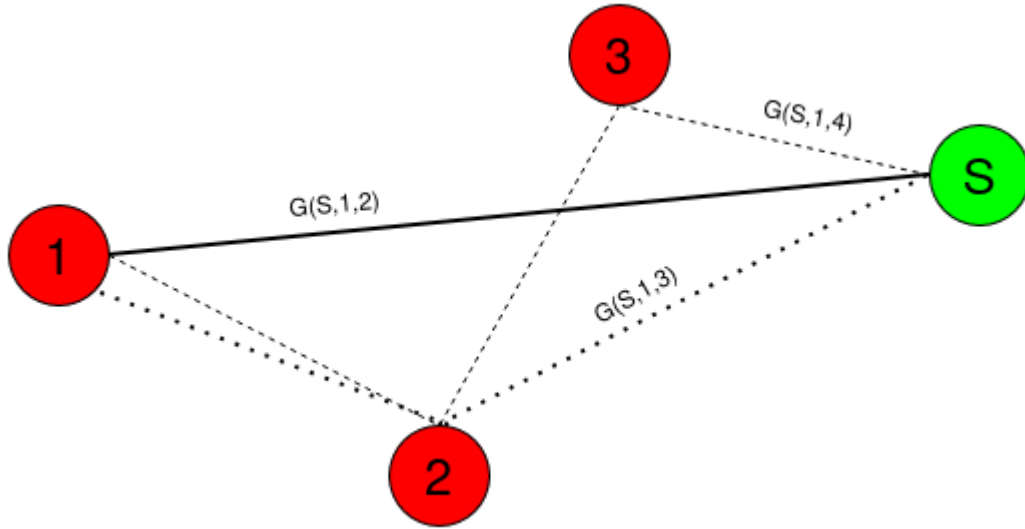
```
        // Το key είναι ο αριθμός των πόλεων, το value η αντίστοιχη διαδρομή.
```

```
        HashMap S{[key: k] : [value: solution]} =
```

```

// Η υπορουτίνα arora επιστρέφει τις λύσεις για κάθε k
// μέχρι και το startIndex+1.
aroraSubroutine(0, startIndex, start, point, startIndex+1)
for entry in S:
    gottliebKey = new GottliebKey(start, point, entry.key)
    solutions.put(gottliebKey, entry.value)

```



ΕΙΚΟΝΑ 5.2.1: Οι διαδρομές που αντιστοιχούν στα $G(S,1,k)$. Ανάλογες διαδρομές θα υπολογιστούν και για τα $G(S,2,k)$, $G(S,3,k)$

Στο παραπάνω κομμάτι ψευδοκώδικα περιγράφεται ακριβώς αυτή η διαδικασία. Το σημείο start αποτελεί τη δοθείσα εκκίνηση, επομένως αρχικά ορίζουμε το παράθυρο από την πρώτη πόλη μέχρι και την εκκίνηση, δηλαδή ασχολούμαστε μόνο με τις πόλεις που είναι μικρότερες της πόλης εκκίνησης. Για κάθε μία από αυτές εφαρμόζουμε τον αλγόριθμο του Arora για το k-TSP με άκρα την πόλη εκκίνησης και την τρέχουσα πόλη, στο παράθυρο που ορίσαμε και για όλες τις πιθανές τιμές του k (2 έως και μέγεθος παραθύρου). Όπως είδαμε στο προηγούμενο Κεφάλαιο, αυτός επιστρέφει έναν χάρτη κατακερματισμού (HashMap) που έχει για κλειδιά (keys) τον αριθμό των πόλεων από τις οποίες διέρχεται μία διαδρομή και για τιμές (values) τις διαδρομές αυτές. Κάθε διαδρομή που υπολογίζεται κατά αυτό τον τρόπο την αποθηκεύουμε μαζί με τον κατάλληλο δείκτη Gottlieb.

Έχοντας τις πρώτες διαδρομές μπορούμε να επεκταθούμε μέχρι να βρούμε τις λύσεις που να καταλήγουν στο δοθέν σημείο λήξης. Έστω $\tilde{l}$ μία πόλη μεγαλύτερη από την εκκίνηση s και $\tilde{d}$ μία πόλη ανάμεσα στην εκκίνηση s και στην $\tilde{l}$, δηλαδή $\tilde{d} < \tilde{l}$. Η διαδρομή με δείκτη $G(\tilde{l}, \tilde{d}, k)$ θα είναι η βέλτιστη του συνόλου

$$\{G(\tilde{j}, \tilde{d}', k') + |\tilde{c} - \tilde{d}'| + \text{Arora}(\widetilde{j+1}, \tilde{l}, \tilde{c}, \tilde{d}, k - k'), \\ \tilde{j} \leq \tilde{l}, \tilde{d}' \leq \tilde{j}, \tilde{j} \leq \tilde{c} \leq \tilde{l}, k' < k\} \quad (5.3)$$

Δηλαδή ψάχνουμε τη βέλτιστη διαδρομή που αρχίζει από την εκκίνηση, καταλήγει στην πόλη $\tilde{d}$ και διέρχεται από k πόλεις μικρότερες της πόλης $\tilde{l}$. Σε αυτή τη σχέση:

- ο δείκτης $G(\tilde{j}, \tilde{d}', k')$ αποτελεί την ήδη υπολογισμένη, μέσω του δυναμικού προγραμματισμού, βέλτιστη διαδρομή με άκρα την εκκίνηση s και την πόλη $\tilde{d}'$, που διέρχεται από k' πόλεις στο παράθυρο $W_{\tilde{j}, \tilde{j}}$, όπου $\tilde{j} \leq \tilde{i}$. Επομένως είναι ο βέλτιστος τρόπος να φτάσουμε από την εκκίνηση στην πόλη $\tilde{d}'$, περνώντας από k' πόλεις που είναι όλες μικρότερες της $\tilde{j}$.
- ο δείκτης $Arora(\tilde{j} + 1, \tilde{i}, \tilde{c}, \tilde{d}, k - k')$ είναι η λύση του k-TSP με άκρα τις πόλεις $\tilde{c}$ και $\tilde{d}$, στο παράθυρο $W_{\tilde{j}+1, \tilde{i}}$ και για πλήθος πόλεων $k - k'$. Η πόλη $\tilde{c}$ αποτελεί μια οποιαδήποτε πόλη μεγαλύτερη της $\tilde{j}$ και μικρότερη της $\tilde{i}$ και έτσι οι λύσεις που βρίσκουμε αρχίζουν από την πόλη $\tilde{c}$, καταλήγουν στη ζητούμενη $\tilde{d}$, περνώντας από όσες πόλεις μεγαλύτερες της $\tilde{j}$ χρειάζεται για να φτάσουμε τις ζητούμενες $k - k'$.
- η απόσταση $|\tilde{c} - \tilde{d}'|$ είναι ακριβώς το τμήμα που χρειάζεται για να ενώσουμε τις δύο διαδρομές

Όπως φαίνεται και στη σχέση, τα μόνα κοινά σημεία ανάμεσα σε όλους τους πιθανούς τρόπους να υπολογίσουμε τα ζευγάρια διαδρομών είναι η εκκίνηση τους, που είναι σταθερά η πόλη s , η κατάληξή τους, που είναι σταθερά η πόλη $\tilde{d}$, το πλήθος των πόλεων από τις οποίες συνολικά διέρχονται, που είναι σταθερά k , και τα όρια των πόλεων, καθώς όλες είναι πριν και την πόλη $\tilde{i}$. Με αυτά τα σημεία σταθερά, δοκιμάζουμε όλους τους πιθανούς τρόπους να επιλέξουμε σε ποια πόλη θα καταλήγει η διαδρομή Gottlieb, από ποια μεγαλύτερη πόλη θα αρχίζει η διαδρομή Arora, καθώς και από πόσες πόλεις θα διέρχεται η καθεμία από τις δύο διαδρομές. Όταν ολοκληρωθεί η διαδικασία θα έχουμε τον πιο σύντρομο τρόπο να φτάσουμε στην πόλη $\tilde{d}$ μέσα στο παράθυρο πόλεων που ορίζει η $\tilde{i}$, συνδέοντας πάντα μία διαδρομή Gottlieb με μία διαδρομή Arora, των οποίων τα όρια θα είναι μη επικαλυπτόμενα.

Η παραπάνω διαδικασία παρουσιάζεται συνοπτικά και στην *Εικόνα 5.2.2*, μετά την παρουσίαση του σχετικού ψευδοκώδικα.

```
// Το fastAroraSolutions είναι ένα HashMap στο οποίο θα αποθηκεύσουμε εκ των προτέρων
// τις λύσεις της υπορουτίνας Arora που θα χρειαστούμε μετέπειτα. Οι διαδρομές θα είναι
// αντιστοιχισμένες με τους κατάλληλους δείκτες Arora.
HashMap fastAroraSolutions[[key: AroraKey] : [value: solution]]
```

```
// Οι παρακάτω υπολογισμοί γίνονται για τις διαδρομές μετά το σημείο εκκίνησης.
```

```
void multiThreadedTSP(Points, startIndex):
```

```
    for i in points.indices > startIndex:
```

```
        Tasks = [ ]
```

```
        for j in points.indices > i:
```

```
            Tasks += threadTask(i, j, startIndex)
```

```
        execute all tasks
```

```
        Tasks = [ ]
```

```
void threadTask(int i, int j, int index):
```

```
    for x in points.indices > index:
```

```
        for y in points.indices > x :
```

```
            if (!(x <= i <= y) or !(x <= j <= y)) then continue;
```

```
            // Για κάθε διαφορετική επιλογή πόλεων στο παράθυρο των x και y
```

```
            // εισάγουμε σε ένα ξεχωριστό task το k-TSP που θα εκτελεστεί.
```

```
            Task task = {
```

```
                // Στο S αντιστοιχίζουμε κάθε λύση του k-TSP
```

```
// με τον αριθμό των πόλεων από τις οποίες διέρχεται.
HashMap S{[key: k] : [value: solution]} =
    aroraSubroutine(x, y, points[i], points[j], |i-j|)
for entry in S:
    key = new AroraKey(x, y, points[i], points[j], entry.key)
    fastAroraSolutions.put(key, entry.value)
}
taskList.add(task);
```

Καθώς η διαδικασία που περιγράφεται στον αλγόριθμο είναι ιδιαίτερα απαιτητική από άποψη υπολογιστικής δύναμης έπρεπε να βρεθούν τρόποι ώστε να επιταχυνθεί και να διευκολυνθεί η εφαρμογή της σε ρεαλιστικά δεδομένα. Το παραπάνω κομμάτι ψευδοκώδικα δείχνει ακριβώς μία τέτοια πρωτοβουλία, στην οποία αξιοποιούνται οι δυνατότητες παράλληλου προγραμματισμού της Java. Η αρχική ιδέα, όπως παρουσιάζεται και στην περιγραφή του θεωρητικού αλγορίθμου, ήταν να γίνει ένας εκ των προτέρων υπολογισμός όλων των k-TSP λύσεων με Arora που θα χρησιμοποιηθούν στη συνέχεια της υλοποίησης, ώστε ένας σοβαρός όγκος υπολογισμών να είναι έτοιμος προς χρήση όταν θα χρειάζεται να συνθέτουμε κομμάτια διαδρομών στο τελευταίο μέρος του αλγορίθμου.

Το πρόβλημα που εμφανίστηκε ήταν ότι, αν αυτοί οι υπολογισμοί γίνονταν σειριακά, το πρόγραμμα δε θα μπορούσε να τελειώσει σε επιθυμητό χρόνο. Πληροφοριακά, καθώς χρειαζόμαστε όλες τις διαδρομές για κάθε πιθανή επιλογή εκκίνησης και λήξης, αλλά και για κάθε πιθανή επιλογή παραθύρου που τις περιέχει, οι εκτελέσεις του αλγορίθμου του k-TSP σε αριθμό προσεγγίζουν την τάξη του n^4 , δηλαδή 4 εμφωλευμένους βρόχους επανάληψης σε όλους τους δείκτες. Με την χρήση των πολυνηματικών τεχνικών υπολογίζουμε ταυτόχρονα τα k-TSP για τους δύο εσωτερικούς βρόχους, το οποίο τυπικά είναι σαν να είχαμε n^2 εκτελέσεις. Δυστυχώς, λόγω περιορισμών του υπολογιστικού μας συστήματος, δεν μπορέσαμε να χρησιμοποιήσουμε ακόμα παραπάνω νήματα για περισσότερους παράλληλους υπολογισμούς, αλλά τουλάχιστον σε ένα παράδειγμα 30 πόλεων φάνηκε έστω και με μεγάλες καθυστερήσεις τελικά να λαμβάνουμε αποτέλεσμα. Σαφώς για να είναι ο αλγόριθμος πρακτικός οφείλουν να γίνουν ακόμα περισσότερες βελτιστοποιήσεις και εξοικονομήσεις, όπως άλλωστε και ο ίδιος ο Arora είχε προειδοποιήσει ακόμα και για την απλούστερη εφαρμογή του απλού TSP, αλλά η παρούσα αντιμετώπιση αποτελεί μια ελπιδοφόρα πρώτη βάση.

```
// Το solutions είναι το ίδιο HashMap που χρησιμοποιούμε
// και στην iteratePointsBeforeStart.
HashMap solutions{[key: GottliebKey] : [value: Solution]}

void iteratePointsAfterStart(start, Points):
    int index = Points.indexOf(start);
    for i in Points.indices after index:
        base = new GottliebKey(Points[i], start, 1)
        // Η διαδρομή με προορισμό την εκκίνηση που διέρχεται από 1 πόλη
        // προφανώς αποτελείται μόνο από την εκκίνηση.
        solutions.put(base, [start])
        // Έπειτα υπολογίζουμε για κάθε άλλο προορισμό στο παράθυρο.
        calculateForAllDestinationsInWindow(start, Points, index, i)

void calculateForAllDestinationsInWindow(start, Points, index, i):
    for d in Points.indices from index to i:
        GottliebKey goal = new GottliebKey(Points[i], Points[d], 2);
        // Η διαδρομή με κατάληξη μία πόλη d που διέρχεται από δύο πόλεις
        // προφανώς αποτελείται μόνο από την εκκίνηση και την πόλη d.
```

```
solutions.put(goal, new Solution([start, Points[d]])
calculateForAllNumberOfTowns(Points, index, i, d);
```

Έχοντας υπολογίσει ήδη τις λύσεις του Arora για κάθε πιθανή επιλογή πόλεων σε κάθε πιθανό παράθυρο πόλεων, μπορούμε τελικά να επεκταθούμε και στις διαδρομές με τις πόλεις που έπονται της εκκίνησης. Αρχικά αποθηκεύουμε τις προφανείς διαδρομές, οι οποίες δε χρειάζονται κάποια διαδικασία υπολογισμού καθώς διέρχονται μόνο από μία πόλη. Σαφώς η πόλη αυτή είναι η πόλη εκκίνησης και αυτές ακριβώς οι διαδρομές αντιστοιχούν στους δείκτες Gottlieb (Points[i], start, 1) και τις αποθηκεύουμε κι αυτές στα solutions.

Έπειτα για κάθε πόλη μικρότερη της τρέχουσας θέλουμε να βρούμε όλες τις διαδρομές k-πόλεων που καταλήγουν σε αυτές μέσα στο παράθυρο της τρέχουσας πόλης, δηλαδή στην περιοχή που ορίζουν η πρώτη, και πιο μικρή, πόλη και η τρέχουσα. Η προφανής είναι η διαδρομή 2 πόλεων, η οποία αποτελείται μόνο από την πόλη εκκίνησης και την κάθε πόλη που είναι μικρότερη της τρέχουσας. Για όλες τις διαδρομές περισσότερων από 2 πόλεων, η λογική περιγράφεται στην calcForAllNumberOfTowns. Έχουν επιλεγεί ονόματα μεταβλητών αντίστοιχα με αυτά που επιλέχθηκαν και στη σχέση 5.6 για λόγους ευκρίνειας.

```
// Το solutions είναι το ίδιο HashMap που χρησιμοποιούμε
// και στην iteratePointsBeforeStart.
HashMap solutions{[key: GottliebKey] : [value: Solution]}
// Το fastAroraSolutions είναι το ίδιο HashMap που χρησιμοποιούμε
// στην multiThreadedTSP.
HashMap fastAroraSolutions{[key: AroraKey] : [value: solution]}
// Στον aroraSolutions κρατάμε τις λύσεις που προκύπτουν από την υπορουτίνα του Arora
// αντιστοιχισμένες με τον αριθμό των πόλεων από τις οποίες διέρχονται.
HashMap aroraSolutions{[key: k] : [value: solution]}

void calcForAllNumberOfTowns(Points, index, i, d):
    for k from 3 to i:
        best = [ ]
        goal = new GottliebKey(Points[i], Points[d], k)
        // Για όλα τα υπόλοιπα k (k>2) υλοποιούμε τον τύπο 5.6
        // εκμεταλλευόμενοι τα περιεχόμενα του fastAroraSolutions που υπολογίσαμε ήδη.
        for j from index to i:
            for d' from 0 to j:
                for c from j+1 to i:
                    if (fastAroraSolutions.containsKey({Points[c], Points[d], j+1, i, k})):
                        for t from 2 to k:
                            key = {Points[c], Points[d], j+1, i, t}
                            aroraSolutions.put(t, fastAroraSolutions.get(key));
                    else:
                        aroraSolutions = aroraSubroutine(Points[c], Points[d], j+1, i, k)
                        // Επικαιροποιούμε τον fastArora με τη νέα λύση
                        // για πιθανή μελλοντική χρήση.
                        for t from 2 to k:
                            key = {Points[c], Points[d], j+1, i, t}
                            fastAroraSolutions.put(key, aroraSolutions.get(t));
                        for k' from 1 to min(k - 1, j + 1):
                            tempKey = {Points[j], Points[d'], k'}
```

```

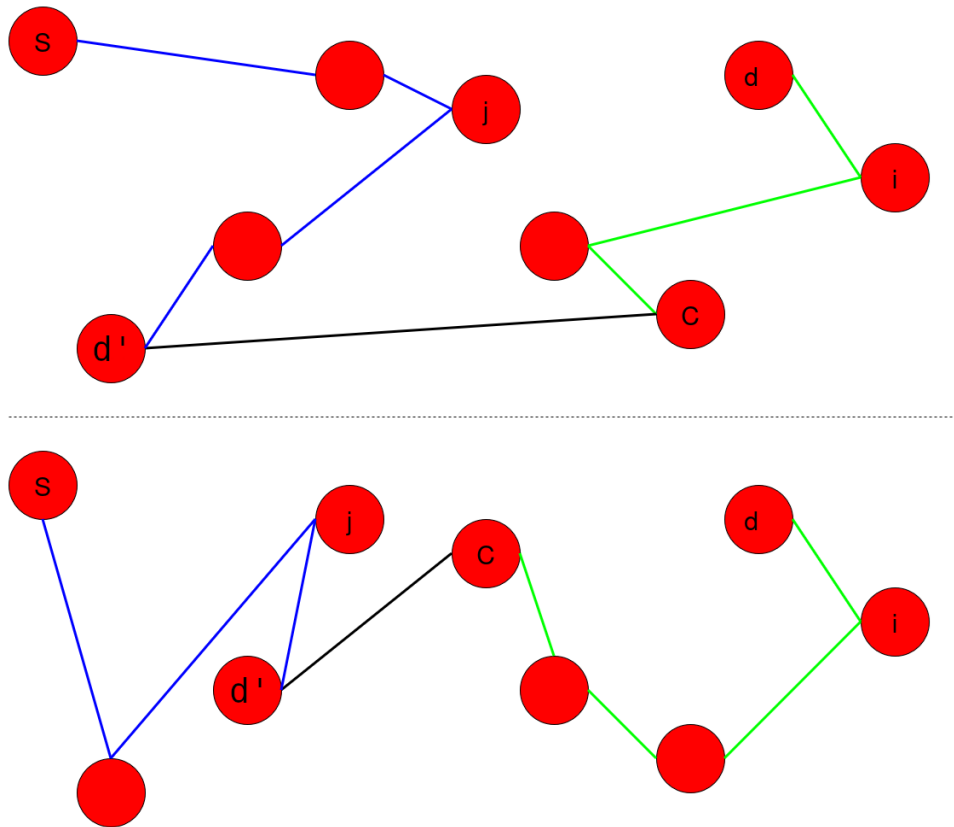
// Βεβαιώνουμε ότι έχει ήδη αποθηκευτεί η λύση
// που αντιστοιχεί στον συγκεκριμένο δείκτη Gottlieb.
if solutions.keys.contains(tempKey):
    tempSolution = solutions.get(tempKey);
    aroraSolution = aroraSolutions.get(k - k');
    tempSequence = tempSolution.sequence;
    // Συνενώνουμε με τη λύση του Arora στις k-k' πόλεις.
    tempSequence.addAll(aroraSolution.sequence);
    if tempSequence.score > best.score:
        best = tempSequence

if best != []:
    solutions.put(goal, new Solution(best))

```

Αυτό που περιγράφεται στο παραπάνω κομμάτι ψευδοκώδικα είναι η βασική λογική του αλγορίθμου του Gottlieb όπως αναφέρεται και στο άρθρο. Δηλαδή, για να το περιγράψουμε με πιο απλά λόγια, στο παράθυρο που ορίζει η πόλη που εξετάζουμε ξεχωρίζουμε δύο περιοχές χωρίς τομή (διαμερισμός δύο τμημάτων), το όριο των οποίων είναι η πόλη σε έναν δείκτη j . Μέσα στις επαναλήψεις αφήνουμε αυτόν τον δείκτη να πάρει κάθε τιμή μέχρι τον δείκτη i και έτσι δοκιμάζουμε όλους τους πιθανούς διαμερισμούς. Στην αριστερή περιοχή δοκιμάζουμε κάθε επιλογή διαδρομής που έχουμε ήδη βρει στην εκτέλεση του αλγορίθμου, επιλέγοντας με τη σειρά διαφορετικούς αριθμούς πόλεων και διαφορετικά σημεία λήξης, και στην δεξιά περιοχή εφαρμόζουμε τον αλγόριθμο του Arora για το k-TSP με παράθυρο όλα τη δεξιά περιοχή, σημείο λήξης το σημείο λήξης που επιζητούσαμε εξ αρχής και σημείο έναρξης κάθε πιθανή επιλογή σε αυτή την περιοχή. Κρατώντας και ενώνοντας τις δύο αυτές διαδρομές, εφόσον ο συνολικός αριθμός πόλεων από τις οποίες διέρχονται είναι ο επιθυμητός, βρίσκουμε μία πιθανή λύση, την οποία και αποθηκεύουμε εφόσον έχει καλύτερη βαθμολογία από τις παλιότερες.

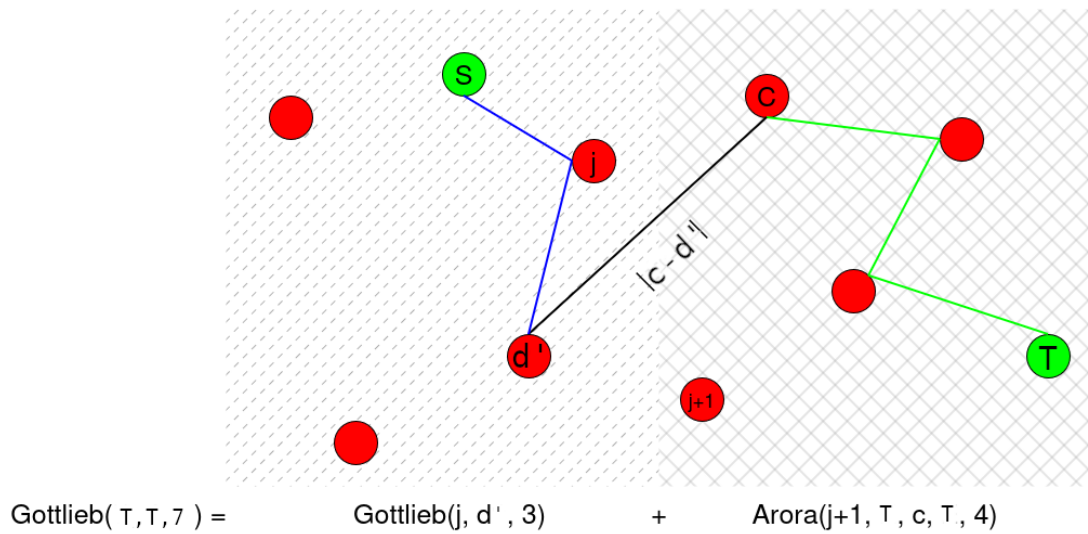
Όπως αναφέρθηκε και νωρίτερα, στην Εικόνα 5.2.2 βλέπουμε τις δύο αυτές περιοχές που ορίζονται από δύο επιλογές του δείκτη j στο παράθυρο $W_{\tilde{1}, \tilde{i}}$ που ορίζει η πόλη $\tilde{i}$. Η μπλε διαδρομή είναι η διαδρομή Gottlieb που αντιστοιχεί κάθε φορά στον δείκτη $(Points[j], Points[d'], |j|)$, δηλαδή βρίσκεται εξ ολοκλήρου στο παράθυρο του δείκτη j , καταλήγει στην πόλη με δείκτη d' και διέρχεται από όλες τις πόλεις της περιοχής. Η πράσινη διαδρομή είναι η διαδρομή Arora που αντιστοιχεί στον δείκτη $(Points[j + 1], Points[i], Points[c], Points[d], |i - j - 1|)$, δηλαδή βρίσκεται στο μέρος του παραθύρου του δείκτη i που δεν ανήκει στο παράθυρο του δείκτη j , αρχίζει από την πόλη με δείκτη c , καταλήγει στην πόλη με δείκτη d και διέρχεται από όλες τις πόλεις της περιοχής. Η μαύρη ακμή δείχνει την σύνδεση των δύο διαδρομών, η οποία μας δίνει σε κάθε περίπτωση μία διαφορετική επιλογή για την διαδρομή Gottlieb με δείκτη $(Points[i], Points[d], 9)$, δηλαδή τη διαδρομή που βρίσκεται στο παράθυρο του δείκτη i , καταλήγει στην πόλη με δείκτη d και διέρχεται και από τις 9 πόλεις του παραδείγματος. Προφανώς η διαδρομή που θα κρατήσουμε είναι αυτή με το μικρότερο συνολικό μήκος.



ΕΙΚΟΝΑ 5.2.2: Διαδρομές Gottlieb 9 πόλεων στο παράθυρο της πόλης i, με διαφορετικούς ενδιαμέσους δείκτες

Στην Εικόνα 5.2.3 βλέπουμε πιο αναλυτικά τη διαδικασία σύνθεσης διαδρομών. Έστω T η πόλη λήξης και θέλουμε να βρούμε μία πιθανή λύση που να αντιστοιχεί στον δείκτη $Gottlieb(T, T, 7)$, δηλαδή μία διαδρομή στο παράθυρο όλων των πόλεων, που να καταλήγει στην πόλη λήξης και να διέρχεται από 7 εκ των 10 πόλεων. Ένας πιθανός συνδυασμός, χωρίζοντας τον χώρο των πόλεων με σύνορο την πόλη j , είναι

- της ήδη υπολογισμένης κατά τη διάρκεια της εκτέλεσης διαδρομής $Gottlieb(\tilde{j}, \tilde{d}', 3)$ στα αριστερά, δηλαδή η διαδρομή που αρχίζει από την πόλη εκκίνησης, καταλήγει στην πόλη $\tilde{d}' < \tilde{j}$ και διέρχεται από 3 πόλεις
- με την $Arora(\tilde{j} + 1, T, \tilde{c}, T, 4)$ στα δεξιά, δηλαδή την εφαρμογή του αλγορίθμου του Arora για το k-TSP με άκρα την πόλη $\tilde{c}$ και την πόλη λήξης T , στο παράθυρο $W_{\tilde{j}+1, T}$, δηλαδή όλων των πόλεων μετά την πόλη $\tilde{j}$, και για $k = 4$. Σημειώνουμε ότι αυτή η διαδρομή Arora είναι υπολογισμένη εξ αρχής, στο σημείο του αλγορίθμου που υπολογίζουμε πολυνηματικά όλες τις πιθανές διαδρομές Arora που θα μπορούσαν να χρησιμοποιηθούν.



ΕΙΚΟΝΑ 5.2.3: Πιθανή σύνθεση διαδρομών για το $Gottlieb(T, T, 7)$

Είναι σαφές ότι η αλλαγή οποιασδήποτε παραμέτρου σε αυτό το παράδειγμα θα οδηγούσε σε άλλη πιθανή λύση 7 πόλεων, των οποίων τα μήκη θα μπορούσαμε μετά να συγκρίνουμε για να επιλέξουμε την καλύτερη. Όταν αναφερόμαστε σε παραμέτρους εννοούμε τον δείκτη j που ορίζει το σύνορο του διαμερισμού, τον δείκτη d' που ορίζει το σημείο λήξης της αριστερής διαδρομής, τον δείκτη c που ορίζει το σημείο εκκίνησης της δεξιάς διαδρομής, καθώς και τον αριθμό πόλεων από τις οποίες θέλουμε να διέρχεται καθένα από τα τμήματα της σύνθεσης (στο παράδειγμα $k' = 3$, $k - k' = 4$).

5.3 Αποτελέσματα

Παρακάτω θα παραθέσουμε τα αποτελέσματα των μετατροπών αυτού του Κεφαλαίου σε σύγκριση με αυτά της υλοποίησης του Κεφαλαίου 4. Θα χρησιμοποιήσουμε το σύνολο Tsiligirides1[28], όπως είναι δημοσιευμένο στην σελίδα του τμήματος Μηχανολόγων Μηχανικών του πανεπιστημίου του Leuven[29], το οποίο αν και χρησιμοποιείται σε προβλήματα Προσανατολισμού, μας βοηθάει στη συγκεκριμένη περίπτωση λόγω του μικρού αριθμού κόμβων του. Επίσης θα χρησιμοποιηθούν δύο δοκιμαστικά σύνολα δικής μας κατασκευής, τα οποία έχουν εξαχθεί από μεγαλύτερα στιγμιότυπα TSP του προηγούμενου κεφαλαίου. Ο λόγος της συμπερίληψης των δύο αυτών μικρότερων συνόλων είναι πρακτικός, καθώς το πλήθος πράξεων απέκλειε την επιλογή στιγμιότυπων με πλήθος πόλεων όσο αυτών που είχαμε στα προηγούμενα κεφάλαια, όπως θα φανεί και από τους χρόνους εκτέλεσης που επίσης παραθέτονται.

Στο δοκιμαστικό σύνολο test ορίσαμε τις εξής (x, y) πόλεις:

$(0,0)$, $(16,20)$, $(7,18)$, $(8,17)$, $(5,9)$, $(2,8)$, $(10,7)$, $(8,23)$, $(12,13)$, $(14,3)$

Στο δοκιμαστικό σύνολο test2 ορίσαμε τις εξής (x, y) πόλεις:

$(0,0)$, $(7,18)$, $(16,20)$, $(8,17)$, $(5,9)$, $(2,8)$, $(10,7)$, $(8,23)$, $(12,13)$, $(14,3)$, $(5,22)$, $(19,7)$, $(12,27)$, $(10,10)$, $(8,21)$, $(3,11)$, $(20,18)$, $(9,14)$, $(0,12)$, $(17,0)$

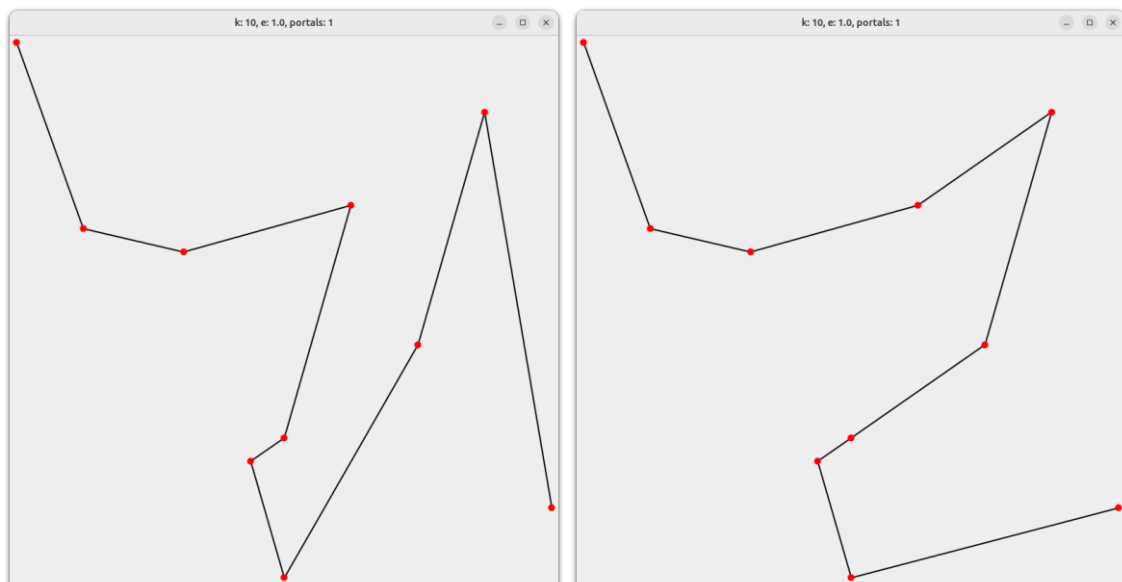
Εκτελώντας στο test πρώτα τον αλγόριθμο του προηγούμενου κεφαλαίου, δηλαδή αντιμετωπίζοντας το σύνολο χρησιμοποιώντας μόνο την k -TSP έκδοση του αλγορίθμου του Arora, η τελική διαδρομή είχε μήκος 71.6 μονάδων. Εκτελώντας τον αλγόριθμο αυτού του κεφαλαίου, ο οποίος επεκτείνει τον αλγόριθμο του Arora, είδαμε η προτεινόμενη λύση να βελτιώνεται κατά 25% περίπου, δηλαδή είχε μήκος 53.4. Παρόλα αυτά ο χρόνος εκτέλεσης καταλήγει να είναι 58 δευτερόλεπτα, έναντι των 300 milliseconds της αρχικής υλοποίησης.

Στα επόμενα δύο παραδείγματα (test2, tsiligirides1) τα αποτελέσματα δεν είναι τόσο αισιόδοξα, καθώς στο ένα η βελτίωση του τελικού μονοπατιού είναι τόσο μικρή που μπορεί να αμεληθεί, ενώ στο δεύτερο καταλήγει ακριβώς στην ίδια λύση. Μάλιστα και στις δύο περιπτώσεις, τα αποτελέσματα αυτά

συννοδεύονται από μία καθόλου πρακτική αύξηση του χρόνου εκτέλεσης. Τα στιγμιότυπα έχουν προφανώς εκτελεστεί σε έναν χαμηλών δυνατοτήτων προσωπικό φορητό υπολογιστή, το οποίο περιορίζει την ταχύτητα εκτέλεσης των πράξεων και κυρίως τον αριθμό νημάτων που μπορούν ταυτόχρονα να χρησιμοποιηθούν, αλλά μία μελλοντική βελτίωση θα έπρεπε να αναγνωρίσει τα μέρη της θεωρίας στα οποία θα μπορούσε να γίνει επέμβαση, ώστε οι χρόνοι να περιοριστούν στα όρια του εφικτού και έτσι να έχει νόημα η προσπάθεια εύρεσης μίας πιθανά καλύτερης λύσης. Μία τέτοια επέμβαση θα ήταν ανάλογη με τον περιορισμό του αριθμού των θυρών στον αρχικό αλγόριθμο του Arora. Παρόλα αυτά, το τελικό συμπέρασμα φαίνεται να επιβεβαιώνει την αρχική υπόθεση ότι η μετατροπή του αλγορίθμου βασισμένη στις παρατηρήσεις των Gottlieb et al. δεν επιστρέφει χειρότερη λύση από αυτή της προηγούμενης υλοποίησης.

Συνοπτικά παραθέτουμε τα συμπεράσματα από τα άλλα δύο στιγμιότυπα:

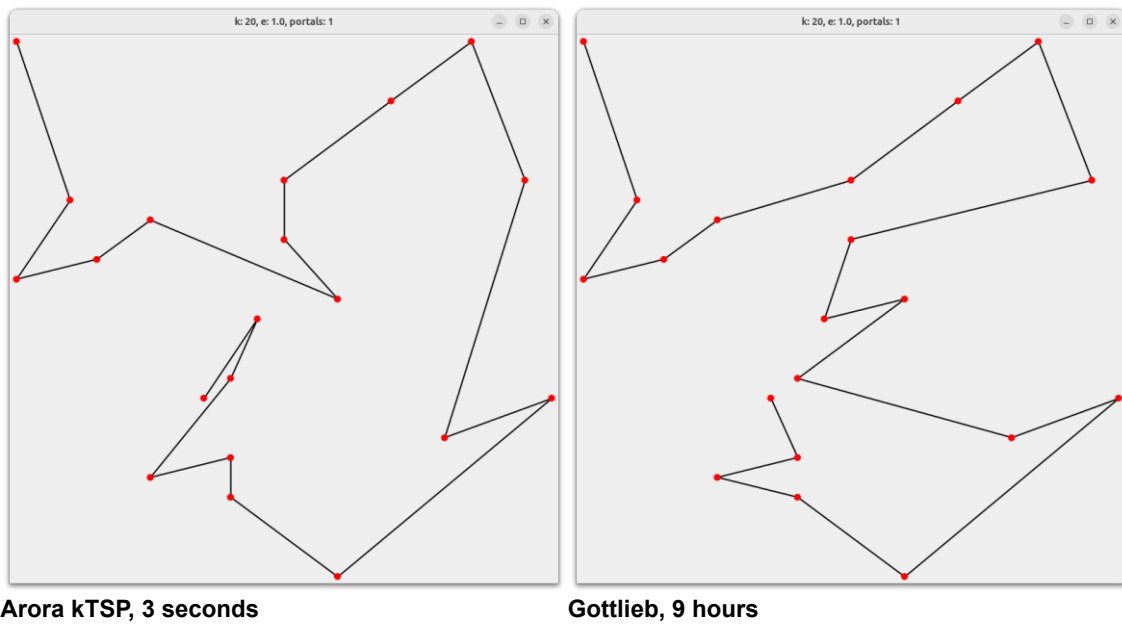
- test2: το αρχικό μήκος διαδρομής των 105 μονάδων βελτιώνεται στις 104, δηλαδή παρατηρείται βελτίωση μικρότερη του 1%. Ο χρόνος εκτέλεσης είναι απαγορευτικός, καθώς χρειάστηκαν 9 ώρες για να προκύψει αποτέλεσμα, έναντι των 3 δευτερολέπτων της πρώτης υλοποίησης.
- tsiligirides1: οι δύο αλγόριθμοι επιστρέφουν ακριβώς την ίδια βέλτιστη λύση. Ο χρόνος εκτέλεσης είναι επίσης απαγορευτικός, καθώς χρειάστηκαν 4.5 ώρες για να προκύψει αποτέλεσμα έναντι των 15 δευτερολέπτων της πρώτης υλοποίησης.



Arora kTSP, 0.3 seconds

Gottlieb, 58 seconds

ΕΙΚΟΝΑ 5.3.1: Βελτίωση διαδρομής του test με τις μετατροπές της υπορουτίνας Gottlieb



ΕΙΚΟΝΑ 5.3.2: Βελτίωση διαδρομής του test2 με τις μετατροπές της υπορουτίνας Gottlieb



ΕΙΚΟΝΑ 5.3.3: Στο στιγμιότυπο Tsiligirides1 οι διαδρομές των δύο υλοποιήσεων είναι ίδιες

Συμπεράσματα

Στην παραπάνω εργασία έγιναν υλοποιήσεις θεωρητικών προσεγγιστικών αλγορίθμων για τα προβλήματα του Περιπλανώμενου Πωλητή και του Περιπλανώμενου Πωλητή με k στάσεις. Προσαρμόζοντας κατάλληλα τη θεωρία και αξιοποιώντας δοκιμασμένες τεχνικές για πιο αποδοτικούς υπολογισμούς (π.χ. πολυνηματικές τεχνικές, caching) καταφέραμε να εμφανίσουμε επιτυχείς προσεγγίσεις για διάφορα στιγμιότυπα των παραπάνω προβλημάτων βασιζόμενοι στα ευρήματα του Aroga, και ελέγξαμε και αξιολογήσαμε την πιθανή χρησιμότητα των ευρημάτων των Gottlieb et al. στην περαιτέρω βελτίωσή τους.

Η ευρεία εφαρμογή τέτοιων υλοποιήσεων, μετά την αντιμετώπιση και των προαναφερθέντων προβλημάτων απόδοσης, έχει δύο βασικά πλεονεκτήματα. Αρχικά η μαθηματική θεμελίωση προσφέρει μία εκ των προτέρων βεβαίωση σχετικά με την ποιότητα της προσέγγισης συγκριτικά με την άγνωστη βέλτιστη και, επιπλέον, οι πρακτικοί περιορισμοί οδηγούν αναγκαστικά σε καινοτομίες όσον αφορά τη βελτίωση της απόδοσης χρόνου και πόρων. Συγκεκριμένα στα προβλήματα που αντιμετωπίσαμε στην παρούσα εργασία, από μόνη της η γεωμετρική προσέγγιση φανερώνει μία εγγενή δομή σε αυτά και, γνωρίζοντάς την, μπορούμε στοχευμένα να εφαρμόσουμε τους παρόντες αλγορίθμους σε στιγμιότυπα στα οποία οι κόμβοι φαίνεται να ακολουθούν αυτή τη δομή (π.χ. κόμβοι σε διακριτές συστάδες, κόμβοι με συμμετρικότητα). Αυτή η νέα γνώση μπορεί να αποδειχτεί ιδιαίτερα χρήσιμη, καθώς ανάλογα τη φύση του στιγμιότυπου θα μπορούμε να έχουμε μία πιο αποδοτική επιλογή αλγορίθμου για την επίλυσή του.

Γι' αυτούς τους λόγους μία ακόμα βαθύτερη ανάλυση παρόμοιων θεωρητικών αλγορίθμων, με έμφαση στην πρακτική υλοποίησή τους, κρίνεται σημαντική.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος Όρος	Ελληνικός Όρος
Travelling Salesperson Problem	Πρόβλημα του Πλανώδιου Πωλητή
Προσανατολισμός	Orienteering
Θύρα	Portal
Παράθυρο	Window
Πολυνηματικός	Multithreading
Επικάλυψη	Override
Μεταβλητή αληθείας	Boolean
Χάρτης κατακερματισμού	HashMap
Κλειδί	Key
Τιμή	Value
Τυχαιοποίηση	Randomization

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

NP	Non-deterministic Polynomial
TSP	Travelling Salesperson Problem
k-TSP	k-Travelling Salesperson Problem

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Garey, M., Johnson, D.S (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness
2. Garey, M., Graham, R.L., Johnson, D.S. (1976) Some NP-complete geometric problems
3. Papadimitriou, C.H. (1977) The Euclidean traveling salesman problem is NP-complete
4. Trevisan, L. (2000) When Hamming meets Euclid: the approximability of geometric TSP and Steiner tree
5. Labbé, M., Laporte, G., Rodríguez Martín, I., Salazar González, J.J. (2004) The Ring Star Problem: Polyhedral analysis and exact algorithm
6. Lin, S., Kernighan, B.W. (1973) An effective heuristic algorithm for the traveling salesman problem
7. Kirkpatrick, S., Gelatt, C. D. Jr., Vecchi, M. P. (1983) Optimization by simulated annealing
8. Helsgaun, K. (2000) An effective implementation of the Lin–Kernighan traveling salesman heuristic
9. Vazirani, V. V. (2001) Approximation algorithms
10. Christofides, N. (1976) Worst-case analysis of a new heuristic for the travelling salesman problem
11. Serdyukov, A. (1978) On some extremal walks in graphs
12. Shmoys, D.B., Willmason, D.P. (1990) Analyzing the Held-Karp TSP bound: A monotonicity property with application
13. Wolsey, L.A (1980) Heuristic analysis, linear programming and branch and bound
14. Karlin, A.R. et al. (2020) A (Slightly) Improved Approximation Algorithm for Metric TSP
15. Arora, S. (1998) Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems
16. Mitchell, J.S.B (1999) Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems
17. Klein, P.N. (2008) A Linear-Time Approximation Scheme for TSP in Undirected Planar Graphs with Edge-Weights
18. Gottlieb, L. A., Krauthgamer, R., & Rika, H. (2020) Faster algorithms for orienteering and k-TSP
19. Balas, E. (1995) The prize collecting traveling salesman problem
20. Golden, B.L., Levy, L., Vohra, R. (1987) The orienteering problem
21. Toth, P., Vigo, D. (2002) The Vehicle Routing Problem, SIAM Monographs on Discrete Mathematics and Applications
22. Karp, R.M. (1977) Probabilistic analysis of partitioning algorithms for the TSP in the plane
23. Rao, S., Smith, W. (1998) Approximating geometric graphs via “spanners” and “banyans”
24. Czumaj, A., Lingas, A. (1998) A polynomial time approximation scheme for Euclidean min-imum cost k-connectivity
25. Rodeker, B., Cifuentes, M. V., & Favre, L. (2009) An Empirical Analysis of Approximation Algorithms for Euclidean TSP
26. Reinelt, G. (1991) TSPLIB: A Traveling Salesman Problem Library
27. TSPLIB95, Heidelberg University
28. Tsiligirides, T. (1984) Heuristic Methods Applied to Orienteering
29. The Orienteering Problem: Test Instances, Leuven University