



Σχολή Θετικών Επιστημών και

Τεχνολογίας

Μεταπτυχιακή Εξειδίκευση στα

Πληροφοριακά Συστήματα (ΠΛΣ)

Διπλωματική Εργασία

Υπολογισμός Γράφου Ορατότητας για Μετακίνηση Διαμέσου

Εμποδίων με χρήση Εξελικτικών Αλγορίθμων

Αθανάσιος Πανάρετος

Επιβλέπων καθηγητής: Βασίλειος Καρυώτης

Πάτρα, Σεπτέμβριος 2021

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή («συγγραφέας/δημιουργός») που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης ο συγγραφέας/δημιουργός εκχωρεί στο ΕΑΠ, μη αποκλειστική άδεια χρήσης του δικαιώματος αναπαραγωγής, προσαρμογής, δημόσιου δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσής τους διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος και για όλο το χρόνο διάρκειας των δικαιωμάτων πνευματικής ιδιοκτησίας. Η ανοικτή πρόσβαση στο πλήρες κείμενο για μελέτη και ανάγνωση δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, αποθήκευση, πώληση, εμπορική χρήση, μετάδοση, διανομή, έκδοση, εκτέλεση, «μεταφόρτωση» (downloading), «ανάρτηση» (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού. Ο συγγραφέας/δημιουργός διατηρεί το σύνολο των ηθικών και περιουσιακών του δικαιωμάτων.

Υπολογισμός Γράφου Ορατότητας για Μετακίνηση
Διαμέσου Εμποδίων με χρήση Εξελικτικών Αλγορίθμων

Αθανάσιος Πανάρετος

Επιτροπή Επίβλεψης Διπλωματικής Εργασίας

Επιβλέπων Καθηγητής:
Βασίλειος Καρυώτης
Αναπληρωτής Καθηγητής
Τμήμα Πληροφορικής
Ιόνιο Πανεπιστήμιο

Συν-Επιβλέπων Καθηγητής:
Αλέξανδρος Φραγκιαδάκης
Συνεργαζόμενος Ερευνητής, Ινστιτούτο
Πληροφορικής, Ίδρυμα Τεχνολογίας και
Έρευνας

Πάτρα, Σεπτέμβριος 2021

*Ευχαριστώ τον επιβλέποντα καθηγητή μου κ. Βασίλειο Καρυώτη
για τις συμβουλές και την καθοδήγηση κατά την εκπόνηση της διπλωματικής εργασίας.*

Περίληψη

Στην παρούσα εργασία παρουσιάζουμε δύο γενετικούς αλγορίθμους σχεδιασμού κίνησης για ένα ρομπότ το οποίο μετακινείται διαμέσου πολυγωνικών εμποδίων, τον **Γενετικό Αλγόριθμο #1** και τον **Γενετικό Αλγόριθμο #2**.

Στον Αλγόριθμο #1 χρησιμοποιούμε αρχικά μια απλοϊκή μέθοδο για να κατασκευάσουμε το γράφο ορατότητας, δηλαδή το γράφο που έχει ως κόμβους τις κορυφές των εμποδίων και ακμές τα ευθύγραμμα τμήματα που ενώνουν τις ορατές μεταξύ τους κορυφές. Ακολούθως χρησιμοποιούμε γενετικές διαδικασίες για να υπολογίσουμε το συντομότερο μονοπάτι από έναν τυχαίο κόμβο-εκκίνηση προς έναν τυχαίο κόμβο-προορισμού. Κωδικοποιήσαμε τα χρωμοσώματα του πληθυσμού ως λίστες, τα στοιχεία των οποίων αποτελούν τους κόμβους από τους οποίους διέρχεται το συντομότερο μονοπάτι.

Στον Αλγόριθμο #2 υπολογίζουμε το γράφο ορατότητας με εξελικτικές διαδικασίες και εντοπίζουμε το συντομότερο μονοπάτι με χρήση του αλγορίθμου Dijkstra. Σε αυτό τον αλγόριθμο κωδικοποιήσαμε τα μέλη του πληθυσμού ως δισδιάστατους πίνακες οι οποίοι αντιστοιχούν στους πίνακες γειτνίασης του κάθε γράφου.

Αρχικά περιγράφουμε την έννοια του χώρου διαμόρφωσης μέσα στον οποίο κινείται το ρομπότ. Στη συνέχεια εξετάζουμε τις πιο βασικές μεθόδους απεικόνισης του χώρου αυτού, την απεικόνιση με χρήση πλέγματος και την απεικόνιση με χρήση οδικών χαρτών. Έπειτα παρουσιάζουμε τον αλγόριθμο **Compute Path** (Berg, 2008) ο οποίος δέχεται ως είσοδο έναν οδικό χάρτη και επιστέφει ένα μονοπάτι μεταξύ του σημείου εκκίνησης και τερματισμού το οποίο αποφεύγει συγκρούσεις με τυχόν εμπόδια κατά μήκος της διαδρομής. Ακολούθως, αναφέρουμε εν συντομία διάφορες μεθόδους βελτιστοποίησης. Αφού παραθέσουμε τα χαρακτηριστικά των γενετικών αλγορίθμων, περιγράφουμε τον τρόπο υλοποίησης του **Γενετικού Αλγορίθμου #1** σε γλώσσα προγραμματισμού Python. Ο αλγόριθμος δέχεται ως είσοδο τον επιθυμητό από το χρήστη αριθμό πολυγωνικών εμποδίων καθώς και δύο σημεία στο επίπεδο, στη συνέχεια χρησιμοποιεί τη συνάρτηση `generatePolygon()` η οποία παράγει τυχαία πολύγωνα στο επίπεδο και κατασκευάζει τον γράφο ορατότητας που δημιουργείται από τις κορυφές των εμποδίων και τα σημεία εκκίνησης-τερματισμού. Έπειτα δημιουργεί αρχικό πληθυσμό ατόμων, δηλαδή πιθανά μονοπάτια τα οποία όμως έχουν ως πρώτο και τελευταίο κόμβο τα σημεία εκκίνησης και τερματισμού. Από τον αρχικό πληθυσμό δημιουργούνται με κάθε επανάληψη της γενετικής διαδικασίας νέες γενιές ατόμων. Κάθε γενιά όμως περιέχει όλο και πιο σύντομα μονοπάτια, ενώ στην τελική επανάληψη επιλέγεται η καλύτερη λύση. Στο Κεφάλαιο 5

παρουσιάζουμε τα αποτελέσματα των εκτελέσεων του Γενετικού Αλγορίθμου #1 για διάφορα σενάρια. Παρατηρούμε ότι το μονοπάτι που επιστρέφει κάθε φορά ο Αλγόριθμος #1 προσεγγίζει σε μεγάλο βαθμό το μονοπάτι που επιστρέφει ο αλγόριθμος **Dijkstra**, ενώ σε ορισμένες περιπτώσεις τα δύο μονοπάτια ταυτίζονται.

Στο Κεφάλαιο 6 περιγράφουμε τον τρόπο υλοποίησης του **Γενετικού Αλγορίθμου #2** και στο Κεφάλαιο 7 παρουσιάζουμε σε διαγράμματα τα αποτελέσματα από την εκτέλεση του για διαφορετικό πλήθος κόμβων και εμποδίων. Σε κάθε σενάριο αποτυπώνουμε τις ποσοστιαίες διαφορές¹ στα μήκη των μονοπατιών που προκύπτουν αν σε κάθε μέλος-γράφο του αρχικού και τελικού πληθυσμού εφαρμοσθεί ο αλγόριθμος Dijkstra. Με αυτό τον τρόπο γίνεται πιο κατανοητή στον αναγνώστη η μεταβολή στο μέσο όρο των τιμών της αντικειμενικής συνάρτησης που προκαλούν οι γενετικές διαδικασίες στα μέλη κάθε γενιάς. Τέλος, σχεδιάζουμε το διάγραμμα με την επίδοση των παρακάτω αλγορίθμων:

- Γενετικός Αλγόριθμος #2
- αλγόριθμος brute-force
- αλγόριθμος Lee

Ο Γενετικός Αλγόριθμος #1 αν και είναι αρκετά πιο αργός (χρόνος κατασκευής πλήρους γράφου ορατότητας και χρόνος γενετικών διαδικασιών) σε σχέση με τις παραδοσιακές μεθόδους εύρεσης συντομότερου μονοπατιού (π.χ., αλγόριθμος Dijkstra) καταλήγει ορισμένες φορές στη βέλτιστη λύση², ενώ τις περισσότερες φορές οι λύσεις που βρίσκει την προσεγγίζουν. Το πλεονέκτημα του Γενετικού Αλγορίθμου #1 είναι ότι επιστρέφει πολλαπλές λύσεις (όσες και τα διαφορετικά μέλη/μονοπάτια του πληθυσμού της τελευταίας γενιάς). Επίσης με κατάλληλη τροποποίηση στον υπολογισμό του μήκους των μονοπατιών μπορεί να λειτουργήσει και με αρνητικά βάρη στις ακμές.

Ο Γενετικός Αλγόριθμος #2 είναι πιο γρήγορος από τη μέθοδο brute-force λόγω του ότι χρειάζεται να υπολογίσει ένα μικρό μέρος μόνο από το σύνολο των ακμών του πλήρους γράφου ορατότητας και επιπλέον τα μονοπάτια τα οποία βρίσκει έχουν μήκος που διαφέρει ελάχιστα από τη βέλτιστη λύση. Ο χρόνος εκτέλεσης του Γενετικού Αλγορίθμου #2 μπορεί να βελτιωθεί ακόμη περισσότερο αν σε κάθε γενιά αποθηκεύσουμε π.χ., σε ένα λεξικό τα μέλη (γράφους) του πληθυσμού με το αντίστοιχο συντομότερο μονοπάτι έτσι ώστε κατά τον υπολογισμό του μέσου όρου της αντικειμενικής συνάρτησης να μην επαναυπολογίζουμε τις τιμές των συναρτήσεων για τα μέλη που προϋπήρχαν.

¹ Σε σχέση με τη βέλτιστη λύση.

² Βλέπε σενάρια 1.10 και 1.12

Λέξεις – Κλειδιά

Υπολογιστική γεωμετρία, γράφος ορατότητας, βέλτιστη διαδρομή, εξελικτικοί αλγόριθμοι, πολυγωνικά εμπόδια, σχεδιασμός διαδρομής

Abstract

In this work we present two different genetic motion planning algorithms, **Genetic Algorithm #1** and **Genetic Algorithm #2**.

In the first one, we use a simple (naïve) method to construct the visibility graph that is the graph created from the vertices of polygonal obstacles and the ‘visible’ edges. Next we use genetic operators to calculate the shortest path between the start and end point. For that purpose we have coded the population chromosomes as lists whose elements are the nodes that belong to the path connecting source and destination.

In the second algorithm we first use genetic operators to construct the visibility graph and then calculate the shortest path using the Dijkstra algorithm. In this approach, we have coded the population chromosomes as two dimensional arrays corresponding to the adjacency matrices of each graph.

At first, we present a description of the configuration space. Afterwards, we examine two approaches in environmental modeling, grid-based approach and roadmaps. Then we present the **Compute Path** algorithm that uses a road map and reports a collision free path between start and end point, and next we briefly mention some improvement methods. After listing the characteristics of Genetic Algorithms, we present **Genetic Algorithm #1**. The algorithm is implemented in Python programming language and uses the desired number of polygonal obstacles and two points in the plane as input and then it calls `GeneratePolygons()` function. This function generates random polygons in the plane. Then the Genetic Algorithm #1 constructs the visibility graph created by the barrier peaks and start-end points. It then creates an initial population of possible solutions representing paths, which consist of start and end point as their first and last node respectively. With each repetition of genetic process, new generations of individuals are created from the initial population. However, each generation consists from shorter paths compared with the previous one, while in the final generation the best solution is chosen. In Chapter 5 we present the results produced by Genetic Algorithm #1 for various scenarios. We observed that the path/solution returned each time by the Genetic Algorithm #1 approaches the path returned by **Dijkstra’s algorithm**, while in some cases the paths are identical.

In Chapter 6, we describe the implementation method of **Genetic Algorithm #2**, while in Chapter 7 we present the results produced for different number of nodes and obstacles. In

each scenario we capture the percentage differences³ in the lengths of the paths that result if the Dijkstra algorithm is applied to each-graph member of the initial and final population. In this way, the change in the average of the values of the objective function caused by the genetic processes in the members of each generation, becomes more comprehensible. Finally, we design the diagram with the performance of the following algorithms:

- Genetic Algorithm #2
- Brute-force algorithm
- Lee's algorithm

Regarding Genetic Algorithm #1, although much slower (complete graph construction time plus genetic process time) compared to traditional search methods (e.g., Dijkstra algorithm), while most of the time it approaches the optimal solution⁴, sometimes the solution coincides with it. The advantage of Genetic Algorithm #1 is that it returns multiple solutions (as many as the different members/paths of last generation). Also with a suitable modification in the calculation of the length of the paths, it can work with negative weights at the edges.

Genetic Algorithm #2 is faster than the brute-force method because it needs to calculate only a small part of the total edges of the complete visibility graph and in addition the resulting path lengths differ slightly than the optimal path length. The execution time of Genetic Algorithm #2 can be further improved if in each generation we store, for example in a dictionary, the members (graphs) of the population with the corresponding shortest path so that when calculating the average of the objective function, we avoid recalculating the values of the objective function of the pre-existing members.

Keywords

Computational Geometry, visibility graph, optimal path, evolutionary algorithms, polygon obstacles, path planning

³ In relation to the global optimum.

⁴ See scenarios 1.10 and 1.12

Πίνακας περιεχομένων

Περίληψη	v
Abstract	viii
Πίνακας Περιεχομένων	x
Κατάλογος Εικόνων	xii
Κατάλογος Πινάκων/Διαγραμμάτων	xix
1 Εισαγωγή	1
2 Αλγόριθμοι σχεδιασμού κίνησης.....	5
2.1 Χώρος διαμόρφωσης	5
2.2 Αναπαραστάσεις χώρου διαμόρφωσης	7
2.3 Απεικόνιση χώρου διαμόρφωσης με χρήση πλέγματος (Grid-based method).....	8
2.4 Γράφος δειγματοληψίας πλέγματος και γενετικοί αλγόριθμοι (Grid-based Graph with Genetic Algorithms)	10
2.5 Οδικοί χάρτες (road-maps)	12
2.5.1 Διαμέριση του Χώρου Διαμόρφωσης σε Κελιά (Cell Decomposition)	12
2.5.2 Πιθανολογική Χάραξη Πορείας (Probabilistic Road-map-PRM).....	18
2.5.3 Διαγράμματα Voronoi	19
2.5.4 Γράφος Ορατότητας (Visibility Graph)	21
2.6 Μέθοδοι Βελτιστοποίησης (Improvement Methods).....	26
2.6.1 Γειτονικές Λύσεις.....	26
2.6.2 Μέθοδος Gradient	26
2.6.3 Μέθοδος tabu	27
2.6.4 Μέθοδος Ανόπτησης (Simulated Annealing)	27
2.6.5 Γενετικοί Αλγόριθμοι (Genetic Algorithms)	28
3 Γενετικοί Αλγόριθμοι.....	29
3.1 Γενετικός Αλγόριθμος – μια πρώτη ματιά	29
3.2 Βασικά Συστατικά Γενετικού Αλγορίθμου	31
3.2.1 Αναπαράσταση.....	31
3.2.2 Αντικειμενική Συνάρτηση (Fitness Function)	33
3.2.3 Πληθυσμός	33
3.2.4 Μηχανισμός επιλογής γονιών (Parent Selection).....	33
3.2.5 Γενετικοί Τελεστές.....	34
3.2.6 Αντικατάσταση (Survivor Selection).....	37
4 Σχεδίαση Γενετικού Αλγορίθμου #1	38
4.1 Σχεδίαση Αλγορίθμου #1	39
4.2 Μέρος Πρώτο – Κατασκευή Πολυγωνικών Εμποδίων	40
4.3 Μέρος Δεύτερο – Κατασκευή Γράφου Ορατότητας	63
4.4 Μέρος Τρίτο – Αρχικοποίηση Πληθυσμού	72
4.4.1 Κατασκευή Πίνακα starting_pool[]	73
4.4.2 Αρχικοποίηση Πίνακα population[]	78
4.5 Μέρος Τέταρτο – Αντικειμενική Συνάρτηση (Fitness Function).....	84
4.6 Μέρος Πέμπτο – Κύριος Βρόχος Επανάληψης.....	85
4.6.1 Κατασκευή Δομής Δεδομένων data4	85
4.6.2 Επιλογή γονιών	89
4.6.2.1 Tournament	90
4.6.2.2 Κατασκευή Πίνακα pool[]	91
4.6.3 Διασταύρωση (crossover)	96

4.6.4 Υπολογισμός Τιμής Αντικειμενικής Συνάρτησης Παιδιών	100
4.6.5 Μετάλλαξη (Mutation).....	102
4.6.5.1 Συνάρτηση g_shuffle()	108
4.6.5.2 Συνάρτηση swap().....	112
4.6.5.3 Συνάρτηση deletion()	116
4.6.5.4 Συνάρτηση cluster_swap2()	119
4.6.6 Ενημέρωση Πίνακα population[]	134
4.6.7 Υπολογισμός Μέσης Τιμής Αντικειμενικής Συνάρτησης.....	142
4.7 Μέρος Έκτο – Έξοδος	144
5 Παρουσίαση Αποτελεσμάτων Γενετικού Αλγορίθμου #1	151
5.1 Παραμετροποίηση	151
5.2 Μελέτη Αποτελεσμάτων Γενετικού Αλγορίθμου #1	152
6 Σχεδίαση Γενετικού Αλγορίθμου #2	180
6.1 Σχεδίαση Αλγορίθμου #2	181
6.2 Μέρος Πρώτο – Δημιουργία Ακμών Εμποδίων	182
6.3 Μέρος Δεύτερο – Πρόσθεση Ακμών Μεταξύ Εμποδίων	184
6.4 Μέρος Τρίτο – Αρχικοποίηση Πληθυσμού	188
6.5 Μέρος Τέταρτο – Κύριος Βρόχος Επανάληψης	192
6.5.1 Επιλογή Γονιών (Parent Selection)	192
6.5.2 Tournament	192
6.5.3 Διασταύρωση (crossover)	192
6.5.4 Μετάλλαξη (Mutation).....	193
6.5.5 Υπολογισμός Μέσου όρου Αντικειμενικής Συνάρτησης και Μέσου όρου Πλήθους Ακμών	196
6.6 Μέρος Πέμπτο – Έξοδος	198
7 Παρουσίαση Αποτελεσμάτων Γενετικού Αλγορίθμου #2	199
7.1 Παραμετροποίηση	199
7.2 Μελέτη Αποτελεσμάτων Γενετικού Αλγορίθμου #2	199
8 Επίλογος	241
8.1 Σύνοψη και συμπεράσματα για τον Γενετικό Αλγόριθμο #1	241
8.2 Σύνοψη και συμπεράσματα για τον Γενετικό Αλγόριθμο #2	247
8.3 Μελλοντικές κατευθύνσεις-βελτιώσεις Γενετικού Αλγορίθμου #1	249
8.4 Μελλοντικές κατευθύνσεις-βελτιώσεις Γενετικού Αλγορίθμου #2	257
Παράρτημα Α Τραπεζοειδής χάρτης T(S)	260
Παράρτημα Β Αλγόριθμος Trapezoidal Map (S)	264
Παράρτημα Γ Γράφος Ορατότητας και Δίκτυα Αισθητήρων	268
Βιβλιογραφία	272

Κατάλογος Εικόνων

Εικόνα 1 Συνεχής μέθοδος Dijkstra (Continuous Dijkstra method). (πηγή: https://weber.itn.liu.se/~valpo40/pages/kspSlides.pdf).....	2
Εικόνα 2 Rapidly-Exploring Random Tree (πηγή: Wikipedia)	3
Εικόνα 3 Μετατόπιση R(6,4) (πηγή: Computational Geometry, 2008).....	5
Εικόνα 4 Ένα μονοπάτι στο χώρο εργασίας και η αντίστοιχη καμπύλη στο χώρο διαμόρφωσης (πηγή: Computational Geometry, 2008)	7
Εικόνα 5 Κατηγοριοποίηση του προβλήματος εύρεσης διαδρομής (πηγή: Zhang, 2018) ..	8
Εικόνα 6 Απεικόνιση του χώρου διαμόρφωσης με χρήση πλέγματος (grid based approach) (πηγή: Andayesh, 2014)	9
Εικόνα 7 Επιτρεπτές κινήσεις ενός ρομπότ όταν ο χώρος διαμόρφωσης διακριτοποιηθεί με χρήση πλέγματος (πηγή: Givigi, 2017).....	9
Εικόνα 8 Τρία συντομότερα μονοπάτια (με κόκκινο, ροζ και μπλε χρώμα) στα οποία καταλήγει ο Γενετικός Αλγόριθμος (πηγή: AL-Taharwa, 2008).....	10
Εικόνα 9 Καμπύλη Bezier με τέσσερα σημεία ελέγχου (πηγή: Wikipedia)	11
Εικόνα 10 Μονοπάτι που επιστρέφει ο γενετικός αλγόριθμος (χωρίς εξομάλυνση) (πηγή: Ma, 2020)	11
Εικόνα 11 Μονοπάτι (με κόκκινο χρώμα) που επιστρέφει ο γενετικός αλγόριθμος (με εξομάλυνση) (πηγή: Ma, 2020).....	12
Εικόνα 12 Free configuration space C_{free} (πηγή: Computational Geometry, 2008).....	13
Εικόνα 13 Τραπεζοειδής χάρτης του ελεύθερου χώρου C_{free} (πηγή: Computational Geometry, 2008)	14
Εικόνα 14 Οδικός χάρτης (πηγή: Computational Geometry, 2008)	15
Εικόνα 15 Διαδρομή από το σημείο p_{start} στο σημείο p_{goal} (πηγή: Computational Geometry, 2008)	16
Εικόνα 16 Αλγόριθμος Compute Path (πηγή: Computational Geometry, 2008).....	16
Εικόνα 17 Παράδειγμα μεθόδου PRM. a) Τυχαία επιλεγμένοι κόμβοι b) Κατασκευή οδικού χάρτη d) Χάραξη διαδρομής (πηγή: Salzman, 2019).....	18
Εικόνα 18 Αποτύπωση του χώρου διαμόρφωσης (πηγή: Garrido, 2006).....	19
Εικόνα 19 Διάγραμμα Voronoi (πηγή: Garrido, 2006).....	20
Εικόνα 20 Διαδρομή αποφυγής εμποδίων για μετακίνηση από αρχικό σε τελικό σημείο (πηγή: Garrido, 2006).....	20
Εικόνα 21 Σχεδίαση διαδρομής με χρήση διαγράμματος Voronoi και της μεθόδου Fast Marching (πηγή: Garrido, 2006)	21
Εικόνα 22 Σχεδίαση γράφου ορατότητας για την εύρεση της συντομότερης διαδρομής (πηγή: Andayesh, 2014)	21
Εικόνα 23 Υποστηρικτικά τμήματα t_1, t_2, t_3, t_4 μεταξύ των πολυγώνων P_i και P_j (πηγή: Rohnert, 1986)	23
Εικόνα 24 Ο αρχικός (πλήρης) γράφος ορατότητας με 7 εμποδία και 29 κορυφές (πηγή: Liu, 1989).....	23
Εικόνα 25 Ο υπογράφος ορατότητας με 5 εμποδία και 20 κορυφές (πηγή: Liu, 1989)	24
Εικόνα 26 Βελτιωμένος υπογράφος με μόνο 12 κορυφές. Ο υπογράφος αυτός προκύπτει με διαφορετική επιλογή περιττών εμποδίων κατά την εφαρμογή του αλγορίθμου (πηγή: Liu, 1989).....	24
Εικόνα 27 Το συντομότερο μονοπάτι μεταξύ των σημείων S και G το οποίο προκύπτει μετά από οποιονδήποτε αλγόριθμο αναζήτησης π.χ. Dijkstra, A* (πηγή: Liu, 1989).....	24
Εικόνα 28 Το διάγραμμα $VV^{(c)}$ για τέσσερα κυρτά πολυγωνικά εμποδία. Το σύνορο των εμποδίων απεικονίζεται με συμπαγή (μπλε) γραμμή, το τμήμα του διαγράμματος Voronoi	

απεικονίζεται με στικτή (κόκκινη) γραμμή και οι ακμές του γράφου ορατότητας	
απεικονίζονται με διακεκομμένη (μαύρη) γραμμή. (πηγή: Wein, 2005).....	25
Εικόνα 29 Gradient method (πηγή: Dhaenens, 2002).....	27
Εικόνα 30 Το γενικό σχήμα ενός γενετικού αλγορίθμου σε ψευδοκώδικα (πηγή: Introduction to Evolutionary Computing, 2015)	30
Εικόνα 31 Το γενικό σχήμα ενός γενετικού αλγορίθμου σε διάγραμμα ροής (πηγή: Introduction to Evolutionary Computing, 2015)	31
Εικόνα 32 Κώδικας Gray(πηγή: Wikipedia).....	32
Εικόνα 33 Από πάνω προς τα κάτω: swap, insertion, scramble ,inversion mutations (πηγή: Introduction to Evolutionary Computing, 2015)	35
Εικόνα 34 Ανασυνδυασμός γονιδίων (πηγή: Wikipedia)	35
Εικόνα 35 Πάνω: one-point crossover, κάτω: n-point crossover με n=2 (πηγή: Introduction to Evolutionary Computing, 2015).....	36
Εικόνα 36 Uniform crossover (πηγή: Introduction to Evolutionary Computing, 2015).....	37
Εικόνα 37 Διάγραμμα ροής Γενετικού Αλγορίθμου #1	39
Εικόνα 38 Μη έγκυρα πολύγωνα	45
Εικόνα 39 Μη έγκυρα πολύγωνα	46
Εικόνα 40 Εκφυλισμός πολυγώνου σε ευθύγραμμο τμήμα	46
Εικόνα 41 Κατανομή πολυγώνων πριν τη χρήση της check_polygons()	47
Εικόνα 42 Κατανομή πολυγώνων πριν τη χρήση της check_polygons()	48
Εικόνα 43 Έγκυρα και μη έγκυρα πολύγωνα (πηγή: https://shapely.readthedocs.io/en/latest/manual.html?highlight=Polygon#shapely.geometry.Polygon.orient	51
Εικόνα 44 Μη έγκυρες ακμές αφού διέρχονται διαμέσου των εμποδίων	58
Εικόνα 45 Έγκυρες ακμές (δεν διέρχονται πλέον διαμέσου των εμποδίων)	62
Εικόνα 46 Γράφος με 18 κόμβους	65
Εικόνα 47 Εύρεση πιο σύντομου μονοπατιού για γράφο με 20 κόμβους χρησιμοποιώντας τη συνάρτηση nx.dijkstra_path() της βιβλιοθήκης networkx	66
Εικόνα 48 Μονοπάτια με παραμέτρους cut=1 και cut=2.....	79
Εικόνα 49 Συσταδοποίηση 87 κόμβων με χρήση του εργαλείου KMeans της βιβλιοθήκης sklearn	123
Εικόνα 50 Σχεδίαση ευθείας Γραμμικής Παλινδρόμησης με χρήση του εργαλείου LinearRegression.....	127
Εικόνα 51 Διάγραμμα ροής στο στάδιο της μετάλλαξης.....	141
Εικόνα 52 Συντομότερη διαδρομή από κόμβο '25' προς κόμβο '26' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 27 κόμβους.....	154
Εικόνα 53 Συντομότερη διαδρομή από κόμβο '25' προς κόμβο '26' με χρήση Αλγορίθμου Dijkstra για Γράφο με 27 κόμβους	154
Εικόνα 54 Συντομότερη διαδρομή από κόμβο '35' προς κόμβο '36' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 37 κόμβους.....	156
Εικόνα 55 Συντομότερη διαδρομή από κόμβο '35' προς κόμβο '36' με χρήση Αλγορίθμου Dijkstra για Γράφο με 37 κόμβους	156
Εικόνα 56 Συντομότερη διαδρομή από κόμβο '40' προς κόμβο '41' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 42 κόμβους.....	158
Εικόνα 57 Συντομότερη διαδρομή από κόμβο '40' προς κόμβο '41' με χρήση Αλγορίθμου Dijkstra για Γράφο με 42 κόμβους	158
Εικόνα 58 Συντομότερη διαδρομή από κόμβο '45' προς κόμβο '46' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 47 κόμβους.....	161
Εικόνα 59 Συντομότερη διαδρομή από κόμβο '45' προς κόμβο '46' με χρήση Αλγορίθμου Dijkstra για Γράφο με 47 κόμβους	161

Εικόνα 60	Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 57 κόμβους.....	163
Εικόνα 61	Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Αλγορίθμου Dijkstra για Γράφο με 57 κόμβους	163
Εικόνα 62	Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 57 κόμβους.....	165
Εικόνα 63	Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Αλγορίθμου Dijkstra για Γράφο με 57 κόμβους	165
Εικόνα 64	Συντομότερη διαδρομή από κόμβο '60' προς κόμβο '61' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 62 κόμβους.....	167
Εικόνα 65	Συντομότερη διαδρομή από κόμβο '60' προς κόμβο '61' με χρήση Αλγορίθμου Dijkstra για Γράφο με 62 κόμβους	167
Εικόνα 66	Συντομότερη διαδρομή από κόμβο '80' προς κόμβο '81' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 82 κόμβους.....	169
Εικόνα 67	Συντομότερη διαδρομή από κόμβο '80' προς κόμβο '81' με χρήση Αλγορίθμου Dijkstra για Γράφο με 82 κόμβους	169
Εικόνα 68	Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 97 κόμβους.....	171
Εικόνα 69	Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Αλγορίθμου Dijkstra για Γράφο με 97 κόμβους	171
Εικόνα 70	Συντομότερη διαδρομή από κόμβο '100' προς κόμβο '101' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 102 κόμβους.....	173
Εικόνα 71	Συντομότερη διαδρομή από κόμβο '100' προς κόμβο '101' με χρήση Αλγορίθμου Dijkstra για Γράφο με 102 κόμβους	174
Εικόνα 72	Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 97 κόμβους.....	175
Εικόνα 73	Συντομότερη διαδρομή από κόμβο '100' προς κόμβο '101' με χρήση Αλγορίθμου Dijkstra για Γράφο με 97 κόμβους	175
Εικόνα 74	Συσταδοποίηση 97 κόμβων και σχεδίαση ευθείας Γραμμικής Παλινδρόμησης	176
Εικόνα 75	Συντομότερη διαδρομή από κόμβο '90' προς κόμβο '91' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 92 κόμβους.....	177
Εικόνα 76	Συντομότερη διαδρομή από κόμβο '90' προς κόμβο '91' με χρήση Αλγορίθμου Dijkstra για Γράφο με 92 κόμβους	178
Εικόνα 77	Συσταδοποίηση 92 κόμβων και σχεδίαση ευθείας Γραμμικής Παλινδρόμησης	178
Εικόνα 78	Διάγραμμα ροής Γενετικού Αλγορίθμου #2.....	179
Εικόνα 79	Ασύνδετα πολυγωνικά εμπόδια	181
Εικόνα 80	Σύνδεση κόμβων διαφορετικών εμποδίων	183
Εικόνα 81	Συνδεδεμένος γράφος	184
Εικόνα 82	Διάγραμμα Number of edges/Iterations.....	185
Εικόνα 83	Διάγραμμα μετασχηματισμού ανεξάρτητου τμήματος σε συνδεδεμένο γράφο με 122 κόμβους.	186
Εικόνα 84	Αρχικός (συνδεδεμένος) γράφος και συντομότερη διαδρομή με χρήση αλγορίθμου	187
Εικόνα 85	Ποσοστιαίες διαφορές τιμών αντικειμενικής συνάρτησης για τα μέλη του αρχικού πληθυσμού σε σχέση με τη βέλτιστη λύση	188
Εικόνα 86	Ποσοστιαίες διαφορές τιμών αντικειμενικής συνάρτησης για τα μέλη του τελικού πληθυσμού σε σχέση με τη βέλτιστη λύση	189

Εικόνα 87	Λύση στην οποία καταλήγει ο Γενετικός Αλγόριθμος #2	191
Εικόνα 88	Βέλτιστη λύση (global optimum). Στον τίτλο αναφέρεται και ο χρόνος εκτέλεσης της απλοϊκής μεθόδου εύρεσης του πλήρους γράφου	191
Εικόνα 89	Εντολή double-edge swap() (πηγή: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.swas.double_edge_swap.html	193
Εικόνα 90	Συσταδοποίηση 97 κόμβων και σχεδίαση ευθείας γραμμικής παλινδρόμησης	194
Εικόνα 91	Πλήθος μεταλλάξεων που συνέβησαν σε κάθε γενιά. Στον τίτλο καταχωρούμε επίσης πόσες από αυτές ήταν έγκυρες καθώς και την γενιά στην οποία συνέβησαν.	194
Εικόνα 92	Ιστόγραμμα που απεικονίζει: στον άξονα x τις έγκυρες ακμές οι οποίες προστέθηκαν κατά τη διάρκεια των μεταλλάξεων και στον άξονα y το πόσες φορές προστέθηκε η καθεμία.	195
Εικόνα 93	Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	196
Εικόνα 94	Διάγραμμα μεταβολής μέσου όρου ακμών	197
Εικόνα 95	Σενάριο 2.1-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	200
Εικόνα 96	Σενάριο 2.1-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	201
Εικόνα 97	Σενάριο 2.1-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	201
Εικόνα 98	Σενάριο 2.1-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	202
Εικόνα 99	Σενάριο 2.1-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	202
Εικόνα 100	Σενάριο 2.1-Πλήθος μεταλλάξεων ανά γενιά	203
Εικόνα 101	Σενάριο 2.1-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης	203
Εικόνα 102	Σενάριο 2.1-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	204
Εικόνα 103	Σενάριο 2.1-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	204
Εικόνα 104	Σενάριο 2.1-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	205
Εικόνα 105	Σενάριο 2.1-Αλγόριθμος Lee	205
Εικόνα 106	Σενάριο 2.2-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	206
Εικόνα 107	Σενάριο 2.2-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	207
Εικόνα 108	Σενάριο 2.2-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	207
Εικόνα 109	Σενάριο 2.2-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	208
Εικόνα 110	Σενάριο 2.2-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	208
Εικόνα 111	Σενάριο 2.2-Πλήθος μεταλλάξεων ανά γενιά	209
Εικόνα 112	Σενάριο 2.2-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης	209
Εικόνα 113	Σενάριο 2.2-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	210
Εικόνα 114	Σενάριο 2.2-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	210
Εικόνα 115	Σενάριο 2.2-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	211

Εικόνα 116 Σενάριο 2.2-Αλγόριθμος Lee	211
Εικόνα 117 Σενάριο 2.3-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	212
Εικόνα 118 Σενάριο 2.3-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	213
Εικόνα 119 Σενάριο 2.3-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	213
Εικόνα 120 Σενάριο 2.3-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	214
Εικόνα 121 Σενάριο 2.3-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	214
Εικόνα 122 Σενάριο 2.3-Πλήθος μεταλλάξεων ανά γενιά	215
Εικόνα 123 Σενάριο 2.3-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	215
Εικόνα 124 Σενάριο 2.3-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	216
Εικόνα 125 Σενάριο 2.3-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	216
Εικόνα 126 Σενάριο 2.3-Αλγόριθμος Lee	216
Εικόνα 127 Σενάριο 2.4-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	217
Εικόνα 128 Σενάριο 2.4-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	218
Εικόνα 129 Σενάριο 2.4-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	218
Εικόνα 130 Σενάριο 2.4-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	219
Εικόνα 131 Σενάριο 2.4-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	220
Εικόνα 132 Σενάριο 2.4-Πλήθος μεταλλάξεων ανά γενιά	220
Εικόνα 133 Σενάριο 2.4-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	220
Εικόνα 134 Σενάριο 2.4-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	221
Εικόνα 135 Σενάριο 2.4-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	221
Εικόνα 136 Σενάριο 2.4-Αλγόριθμος Lee	221
Εικόνα 137 Σενάριο 2.5-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	222
Εικόνα 138 Σενάριο 2.5-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	223
Εικόνα 139 Σενάριο 2.5-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	223
Εικόνα 140 Σενάριο 2.5-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	224
Εικόνα 141 Σενάριο 2.5-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	224
Εικόνα 142 Σενάριο 2.5-Πλήθος μεταλλάξεων ανά γενιά	225
Εικόνα 143 Σενάριο 2.5-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης	225
Εικόνα 144 Σενάριο 2.5-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	226

Εικόνα 145 Σενάριο 2.5-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	226
Εικόνα 146 Σενάριο 2.5-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	227
Εικόνα 147 Σενάριο 2.5-Αλγόριθμος Lee	227
Εικόνα 148 Σενάριο 2.6-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	228
Εικόνα 149 Σενάριο 2.6-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	229
Εικόνα 150 Σενάριο 2.6-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	229
Εικόνα 151 Σενάριο 2.6-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	230
Εικόνα 152 Σενάριο 2.6-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	231
Εικόνα 153 Σενάριο 2.6-Πλήθος μεταλλάξεων ανά γενιά	231
Εικόνα 154 Σενάριο 2.6-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης	231
Εικόνα 155 Σενάριο 2.6-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	232
Εικόνα 156 Σενάριο 2.6-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	232
Εικόνα 157 Σενάριο 2.6-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	233
Εικόνα 158 Σενάριο 2.6-Αλγόριθμος Lee	233
Εικόνα 159 Σενάριο 2.7-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.	234
Εικόνα 160 Σενάριο 2.7-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά	235
Εικόνα 161 Σενάριο 2.7-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης	235
Εικόνα 162 Σενάριο 2.7-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.	236
Εικόνα 163 Σενάριο 2.7-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.	236
Εικόνα 164 Σενάριο 2.7-Πλήθος μεταλλάξεων ανά γενιά	237
Εικόνα 165 Σενάριο 2.7-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	237
Εικόνα 166 Σενάριο 2.7-Τελικός γράφος Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra	238
Εικόνα 167 Σενάριο 2.7-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση	238
Εικόνα 168 Σενάριο 2.7-Αλγόριθμος Lee	238
Εικόνα 169 Διαφορές αποστάσεων μονοπατιών.....	239
Εικόνα 170 Πλήθος ακμών που υπολογίζει κάθε αλγόριθμος.....	239
Εικόνα 171 Επίδοση Γενετικού Αλγορίθμου #2 (Genetic algorithm), απλοϊκής μεθόδου (brute-force algorithm) και αλγορίθμου Lee (Lee's algorithm).....	240
Εικόνα 172 Ο προσιτός χώρος εργασίας για ένα ρομπότ προσδεμένο μέσω ενός καλωδίου μήκους L. Η μπλε περιοχή είναι ο χώρος εργασίας χωρίς την παρουσία εμποδίων (όλος ο χώρος W) ενώ η γραμμοσκιασμένη περιοχή είναι ο χώρος εργασίας με εμπόδια. (πηγή: researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot)	245
Εικόνα 173 Το ρομπότ δεν μπορεί να διασχίσει τη συντομότερη διαδρομή (γ) μεταξύ των σημείων q_s και q_g με αρχική διαμόρφωση καλωδίου C_i λόγω του περιορισμένου μήκους	

του καλωδίου. (πηγή: researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot)	246
Εικόνα 174 Η καμπύλη γ_1 είναι ομοτοπική με την καμπύλη γ_2 αφού υπάρχει μια συνεχής παραμόρφωση από τη μια στην άλλη αλλά όχι με την καμπύλη γ_3 . (πηγή: researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot)	247
Εικόνα 175 Άθροισμα Minkowski (πηγή: Computational Geometry, 2008)	253
Εικόνα 176 Κύκλος Hamilton για γράφο με έξι κόμβους (πηγή: Wikipedia)	254
Εικόνα 177 Κύκλος Hamilton για κατευθυνόμενο γράφο με επτά κόμβους. Η μόνη διαδρομή η οποία επισκέπτεται όλους τους κόμβους ακριβώς μια φορά είναι η $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$. Παρατηρούμε επίσης τη συμπληρωματικότητα μεταξύ των βάσεων που αναπαριστούν τον κόμβο 4 και εκείνων που αναπαριστούν τις ακμές 3-4 και 4-5 (πηγή: Molecular computing: Does DNA compute?, 1996)	255
Εικόνα 178 Τεχνική PCR (πηγή: Wikipedia)	256
Εικόνα 179 Το μονόκλωνο RNA αριστερά και το δίκλωνο DNA στα δεξιά (πηγή: Wikipedia)	256
Εικόνα 180 Διαίρεση του S σε τομείς (πηγή: Computational Geometry, 2008)	260
Εικόνα 181 Τραπεζοειδής χάρτης (πηγή: Computational Geometry, 2008)	260
Εικόνα 182 Πλευρές μιας όψης ενός Τραπεζοειδούς χάρτη (πηγή: Computational Geometry, 2008)	262
Εικόνα 183 Ονομασία ευθυγράμμων τμημάτων που περικλείουν ένα τραπέζιο Δ (πηγή: Computational Geometry, 2008)	262
Εικόνα 184 Τέσσερις από τις πέντε περιπτώσεις για την αριστερή πλευρά του τραπεζίου Δ (πηγή: Computational Geometry, 2008)	263
Εικόνα 185 Τα γειτονικά τραπέζια με το Δ είναι σκιασμένα με γκρι (πηγή: Computational Geometry, 2008)	263
Εικόνα 186 Ο τραπεζοειδής χάρτης δύο τμημάτων και η δομή δεδομένων του (πηγή: Computational Geometry, 2008)	265
Εικόνα 187 Αλγόριθμος Trapezoidal Map (πηγή: Computational Geometry, 2008)	266
Εικόνα 188 Αλγόριθμος Follow Segment (πηγή: Computational Geometry, 2008)	267
Εικόνα 189 Το νέο τμήμα s_i βρίσκεται εξ ολοκλήρου μέσα στο τραπέζιο Δ (πηγή: Computational Geometry, 2008)	268
Εικόνα 190 Overlay network VON (πηγή: www.researchgate.net/publication/224500413_Visibility-Graph-Based_Shortest-Path_Geographic_Routing_in_Sensor_Networks)	269
Εικόνα 191 (a) Γράφος Ορατότητας (b) Μειωμένος Γράφος Ορατότητας. Με γκρι χρώμα απεικονίζονται τα εμπόδια και η κόκκινη γραμμή απεικονίζει τη συντομότερη διαδρομή από το s στο t. (πηγή: www.researchgate.net/publication/224500413_Visibility-Graph-Based_Shortest-Path_Geographic_Routing_in_Sensor_Networks)	270

Κατάλογος Πινάκων/Διαγραμμάτων

Πίνακας 1	58
Πίνακας 2	59
Πίνακας 3 Σενάριο 1.1	153
Πίνακας 4 Σενάριο 1.2	155
Πίνακας 5 Σενάριο 1.3	157
Πίνακας 6 Σενάριο 1.4	160
Πίνακας 7 Σενάριο 1.5	162
Πίνακας 8 Σενάριο 1.6	164
Πίνακας 9 Σενάριο 1.7	166
Πίνακας 10 Σενάριο 1.8	168
Πίνακας 11 Σενάριο 1.9	170
Πίνακας 12 Σενάριο 1.10	172
Πίνακας 13 Σενάριο 1.11	174
Πίνακας 14 Σενάριο 1.12	177
Πίνακας 15 Σενάριο 2.1	200
Πίνακας 16 Σενάριο 2.2	206
Πίνακας 17 Σενάριο 2.3	212
Πίνακας 18 Σενάριο 2.4	217
Πίνακας 19 Σενάριο 2.5	222
Πίνακας 20 Σενάριο 2.6	228
Πίνακας 21 Σενάριο 2.7	234

Διάγραμμα 1

Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.1	153
---	-----

Διάγραμμα 2

Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.2	155
---	-----

Διάγραμμα 3

Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.3	157
---	-----

Διάγραμμα 4

Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.4	160
---	-----

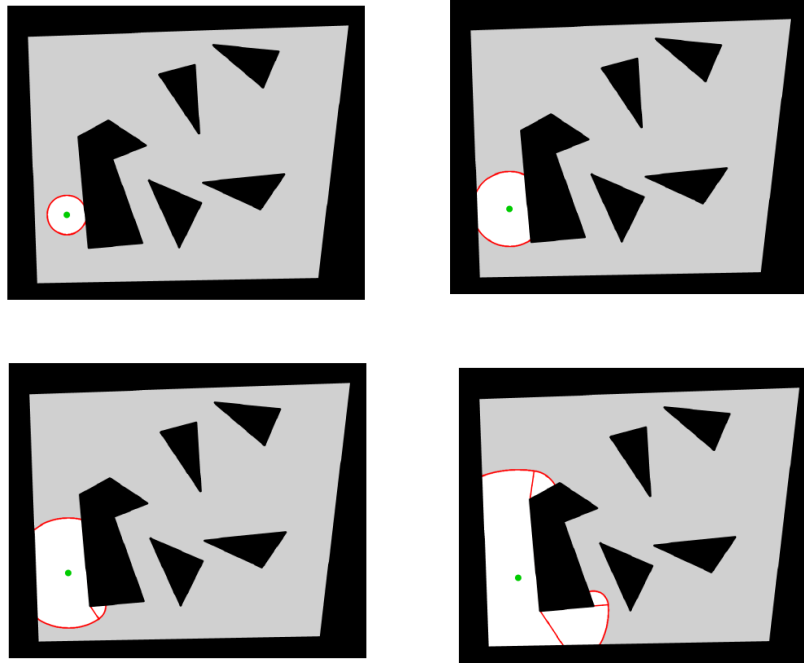
Διάγραμμα 5	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.5.....	162
Διάγραμμα 6	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.6.....	164
Διάγραμμα 7	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.7.....	166
Διάγραμμα 8	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.8.....	168
Διάγραμμα 9	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.9.....	170
Διάγραμμα 10	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.10.....	172
Διάγραμμα 11	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.11.....	174
Διάγραμμα 12	
Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.12.....	177

1. Εισαγωγή

Ο σχεδιασμός πλοήγησης είναι ένα από τα πιο σημαντικά προβλήματα στο χώρο της ρομποτικής. Στόχος ενός αλγορίθμου σχεδιασμού κίνησης είναι η εύρεση της βέλτιστης διαδρομής από ένα αρχικό σε ένα τελικό σημείο. Μια διαδρομή λέμε ότι είναι βέλτιστη όταν: i) το ρομπότ καταφέρνει να αποφεύγει τις συγκρούσεις με τα εμπόδια, και ii) η διαδρομή ικανοποιεί ορισμένα κριτήρια. Με άλλα λόγια ο σχεδιασμός πλοήγησης διαμέσου μιας περιοχής με εμπόδια είναι ένα πρόβλημα βελτιστοποίησης μιας συνάρτησης (π.χ., η απόσταση να είναι η μικρότερη δυνατή) με κάποιους περιορισμούς (π.χ., η διαδρομή να αποφεύγει τα εμπόδια) (Song, 2016).

Η πολυπλοκότητα του προβλήματος σχεδιασμού μιας ασφαλούς διαδρομής διαμέσου εμποδίων εξαρτάται από δύο παράγοντες: α) από τη διάσταση του χώρου διαμόρφωσης, και β) από την απόσταση ασφαλείας από τα εμπόδια μέσα στην οποία επιτρέπεται να κινηθεί το ρομπότ (Salzman, 2019). Ο χώρος διαμόρφωσης (Configuration space) καθορίζεται από τις παραμέτρους που χρειαζόμαστε για να περιγράψουμε τη θέση και τον προσανατολισμό του ρομπότ, ενώ η διάσταση (d) του χώρου διαμόρφωσης είναι ο αριθμός των παραμέτρων που χρειάζονται για να αναπαραστήσουμε μια τυχαία διαμόρφωση. Η διάσταση του χώρου διαμόρφωσης ισοδυναμεί με τους βαθμούς ελευθερίας του ρομπότ. Στην παρούσα εργασία υποθέτουμε ότι το ρομπότ μπορεί μόνο να μετατοπίζεται στο επίπεδο.

Στις δύο διαστάσεις (two dimensional Euclidean shortest path) το πρόβλημα μπορεί να λυθεί σε πολυωνυμικό χρόνο. Οι αλγόριθμοι αυτού του είδους βασίζονται σε δύο προσεγγίσεις (Inkulu, 2010) οι οποίες διαφέρουν στον τρόπο με τον οποίο διακριτοποιούν το πρόβλημα. Σύμφωνα με την πρώτη, αρχικά κατασκευάζεται ο γράφος ορατότητας, μια ειδική δομή δεδομένων που συναντάται στην υπολογιστική γεωμετρία. Ο γράφος ορατότητας έχει ως κόμβους τις κορυφές των εμποδίων. Στη συνέχεια εφαρμόζεται ένας αλγόριθμος εύρεσης συντομότερης διαδρομής (π.χ., Dijkstra). Στη δεύτερη προσέγγιση (continuous Dijkstra method) ένα κύμα εξαπλώνεται από ένα αρχικό σημείο μέχρι να συναντήσει όλα τα υπόλοιπα, όπως φαίνεται στις παρακάτω εικόνες:



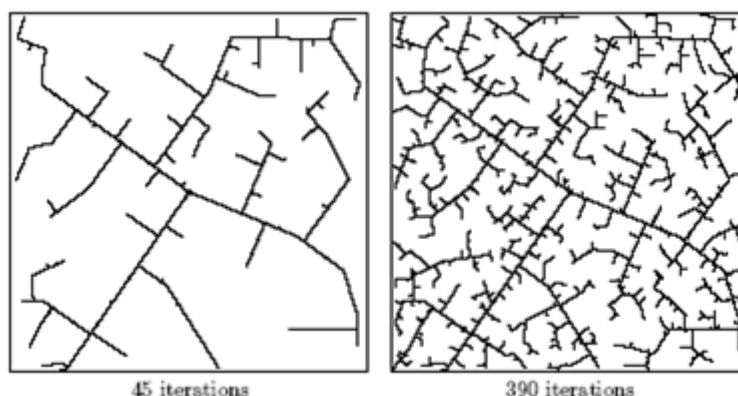
Εικόνα 1 Συνεχής μέθοδος Dijkstra (Continuous Dijkstra method). (πηγή: <https://weber.itn.liu.se/~valpo40/pages/kspSlides.pdf>)

Το πρόβλημα της εύρεσης συντομότερης διαδρομής σε δύο διαστάσεις μελετήθηκε αρχικά από τους Lozano-Perez et. al. (Lozano-Perez, 1979). Ο αλγόριθμος που κατασκεύασαν είχε χρονική πολυπλοκότητα $O(n^3)$. Αργότερα οι Scharir et. al. (Scharir, 1984) πρότειναν μια μέθοδο με χρονική πολυπλοκότητα $O(n^2 \log n)$ κατά την οποία πρώτα κατασκεύασαν το γράφο ορατότητας με κόμβους τις κορυφές των εμποδίων. Έπειτα οι Reif et. al. (Reif, 1985) απέδειξαν ότι μπορούν να κατασκευάσουν έναν αλγόριθμο με χρονική πολυπλοκότητα ίση με $O(n(k + \log n))$, όπου k είναι ο αριθμός των συνδεδεμένων στοιχείων (islands) μέσα στο χώρο των εμποδίων.

Επίσης οι Kapoor et. al. (2000) εφηύραν έναν αλγόριθμο ο οποίος κατασκευάζει το γράφο ορατότητας σε χρόνο $O(n \log n + E)$ όπου E ο αριθμός των ακμών του γράφου. Τέλος οι Inkulu et. al. (2010) χρησιμοποίησαν τη δεύτερη προσέγγιση (continuous Dijkstra method) αφού πρώτα εφάρμοσαν Τριγωνοποίηση του χώρου διαμόρφωσης και εντόπισαν τις διασταυρώσεις ανάμεσα στα τρίγωνα. Ο αλγόριθμος που κατασκεύασαν τρέχει σε χρόνο $O(n + m(\log m)(\log n))$ όπου m είναι ο αριθμός των εμποδίων και n ο αριθμός των κορυφών. Πρέπει να σημειώσουμε ότι ο γράφος ορατότητας διαθέτει στη χειρότερη περίπτωση n^2 ακμές, άρα οποιοσδήποτε αλγόριθμος εύρεσης συντομότερης διαδρομής ο

οποίος αρχικά χρειάζεται να κατασκευάσει το γράφο, θα έχει χρονική πολυπλοκότητα ίση με $O(n^2)$.

Ένας λόγος για τον οποίο το πρόβλημα του σχεδιασμού πλοήγησης διαμέσου εμποδίων είναι δύσκολο, οφείλεται στην πολυπλοκότητα υπολογισμού των εμποδίων όσο αυξάνεται το πλήθος τους. Επίσης η κλασική προσέγγιση στο πρόβλημα σχεδιασμού πλοήγησης έχει πολλά μειονεκτήματα, όπως η αυξημένη χρονική πολυπλοκότητα σε προβλήματα με περισσότερες από δύο διαστάσεις και ο εγκλωβισμός σε τοπικά ελάχιστα (Masehian, 2007). Προκειμένου να βελτιωθεί η αποδοτικότητα των κλασικών μεθόδων, αναπτύχθηκαν διάφορες πιθανολογικές μέθοδοι, όπως π.χ., η Πιθανολογική Χάραξη Πορείας (RPM), καθώς και η μέθοδος με χρήση δέντρων (Rapidly-Exploring Random Tree-RTT).



Εικόνα 2 Rapidly-Exploring Random Tree (πηγή: Wikipedia)

Ακόμη, οι ευρετικοί αλγόριθμοι (heuristic algorithms), όπως π.χ., η μέθοδος ανόπτησης (Simulated Annealing-SA), έχουν τη δυνατότητα να ξεφεύγουν από τα τοπικά ελάχιστα του χώρου αναζήτησης. Άλλες μέθοδοι που χρησιμοποιούνται είναι η μέθοδος με χρήση νευρωνικών δικτύων (Artificial Neural Network-ANN), οι Γενετικοί Αλγόριθμοι (Genetic Algorithms-GA), η Ασαφής Λογική (Fuzzy Logic-FL), η μέθοδος Ταμπού (Tabu Search-TS), η μέθοδος με σμήνος σωματιδίων (Particle Swarm Optimization-PSO) και η μέθοδος με βάση τη λειτουργία των αποικιών των μυρμηγκιών (Ant Colony Optimization-ACO).

Στην παρούσα διπλωματική θα χρησιμοποιήσουμε τη μέθοδο των Γενετικών Αλγορίθμων. Οι αλγόριθμοι αυτού του είδους χρησιμοποιούν μηχανισμούς της Εξελικτικής Βιολογίας όπως είναι η μετάλλαξη (mutation), η διασταύρωση (genetic recombination) και η επιλογή

(selection), ψάχνοντας σε όλο το χώρο των δυνατών λύσεων για να βρουν τη βέλτιστη λύση.

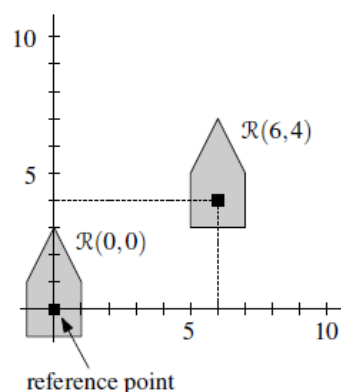
2 Αλγόριθμοι Σχεδιασμού Κίνησης

Ο σχεδιασμός κίνησης περιλαμβάνει πέντε επιμέρους στοιχεία: το χώρο διαμόρφωσης, την αναπαράσταση των εμποδίων, τις διάφορες προσεγγίσεις με βάση τις οποίες σχεδιάζεται η λύση, τις μεθόδους αναζήτησης του καλύτερου μονοπατιού και την τελική βελτίωση της ποιότητας της διαδρομής (Hwang, 1992). Στις επόμενες υποενότητες του κεφαλαίου αφού δούμε τι είναι ο χώρος διαμόρφωσης περιγράφουμε διάφορες μεθόδους για την αναπαράσταση του χώρου αυτού.

2.1 Χώρος Διαμόρφωσης

Ένα από τα βασικά προβλήματα κατά το σχεδιασμό της κίνησης είναι ο υπολογισμός ενός συνεχόμενου μονοπατιού το οποίο συνδέει μια αρχική διαμόρφωση S (configuration S) με μια τελική διαμόρφωση G (configuration G), χωρίς να υπάρχει σύγκρουση με εμπόδια (πηγή: Wikipedia). Η γεωμετρική απεικόνιση του ρομπότ και των εμποδίων μπορεί να γίνει με ένα δισδιάστατο ή τρισδιάστατο χώρο εργασίας (workspace), ενώ η κίνηση του ρομπότ μπορεί να αναπαρασταθεί σαν ένα μονοπάτι μέσα σε ένα χώρο διαμόρφωσης (configuration space). Τι είναι όμως ο χώρος διαμόρφωσης;

Ας υποθέσουμε ότι ένα ρομπότ κινείται στο επίπεδο μέσα σε ένα χώρο εργασίας ο οποίος αποτελείται από ένα σύνολο εμποδίων $S = \{P_1, P_2, \dots, P_t\}$. Επίσης ας υποθέσουμε ότι αυτό παριστάνεται ως ένα απλό πολύγωνο. Η μετατόπιση ή αλλιώς η διαμόρφωση του ρομπότ μπορεί να περιγραφεί με ένα διάνυσμα το οποίο συμβολίζουμε ως $R(x, y)$. Για παράδειγμα αν το ρομπότ είναι το πολύγωνο με κορυφές τα σημεία $(1, -1)$, $(1,1)$, $(0,3)$, $(-1, 1)$, $(-1,-1)$ τότε οι κορυφές του $R(6,4)$ είναι τα σημεία $(7,3)$, $(7,5)$, $(6,7)$, $(5,5)$ και $(5,3)$ όπως φαίνεται στο παρακάτω σχήμα:



Εικόνα 3 Μετατόπιση $R(6,4)$ (πηγή: Computational Geometry, 2008)

Επίσης αν ορίσουμε ως σημείο αναφοράς ένα σημείο το οποίο βρίσκεται στο εσωτερικό του ρομπότ τότε μπορούμε να περιγράψουμε τη μετατόπιση του απλά αναφέροντας τις συντεταγμένες του σημείου αναφοράς. Για παράδειγμα, για τη μετατόπιση $R(0,0)$ μπορούμε να ορίσουμε ως σημείο αναφοράς την αρχή των αξόνων $(0,0)$. Ως εκ τούτου η μετατόπιση $R(x,y)$ σημαίνει ότι το ρομπότ έχει το σημείο αναφοράς του στη θέση (x,y) .

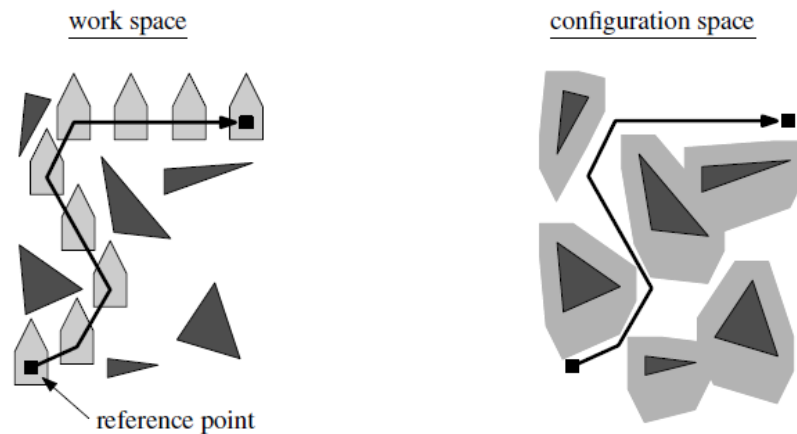
Αν θέλουμε το ρομπότ να μπορεί να αλλάξει τον προσανατολισμό του με περιστροφή γύρω π.χ., από το σημείο αναφοράς, τότε χρειαζόμαστε ακόμη μια παράμετρο, έστω φ , για να μπορούμε να ορίσουμε την κατεύθυνση του. Σε αυτή την περίπτωση η μετατόπιση του συμβολίζεται ως $R(x,y,\varphi)$.

Η διαμόρφωση απεικονίζει τη θέση του ρομπότ σε μια τυχαία χρονική στιγμή ενώ ο χώρος διαμόρφωσης C είναι το σύνολο όλων των δυνατών διαμορφώσεων. Εάν το ρομπότ απεικονίζεται ως ένα απλό σημείο το οποίο κινείται στο επίπεδο, τότε ο χώρος διαμόρφωσης C είναι κι αυτός επίπεδος και μια συγκεκριμένη διαμόρφωση μπορεί να αναπαρασταθεί χρησιμοποιώντας τις συντεταγμένες (x,y) .

Γενικά μια τυχαία τοποθέτηση ενός ρομπότ περιγράφεται από μια σειρά παραμέτρων οι οποίες αντιστοιχούν στους βαθμούς ελευθερίας του ρομπότ. Για ένα ρομπότ το οποίο μπορεί μόνο να μετατοπιστεί, οι βαθμοί ελευθερίας είναι δύο, ενώ όταν μπορεί επιπλέον να περιστραφεί οι βαθμοί ελευθερίας είναι τρεις. Ο χώρος παραμετροποίησης καλείται διαφορετικά χώρος διαμόρφωσης και συμβολίζεται με $C(R)$. Ένα σημείο p μέσα στο χώρο διαμόρφωσης αντιστοιχεί σε μια συγκεκριμένη μετατόπιση $R(p)$ του ρομπότ μέσα στο χώρο εργασίας για αυτό μπορούμε να πούμε ότι ο χώρος διαμόρφωσης ενός ρομπότ που κινείται στο επίπεδο ταυτίζεται με το χώρο εργασίας. Ένα ρομπότ σε σχήμα πολυγώνου παριστάνεται ως ένα σημείο μέσα στο χώρο διαμόρφωσης και αντίστροφα, οποιοδήποτε σημείο μέσα στο χώρο διαμόρφωσης αντιστοιχεί σε μια πραγματική μετατόπιση του ρομπότ μέσα στο χώρο εργασίας.

Όμως δεν είναι όλα τα σημεία του χώρου διαμόρφωσης επιτρεπτά σημεία, δηλαδή σημεία στα οποία μπορεί αυτό να έχει πρόσβαση, όπως είναι για παράδειγμα τα σημεία τομής του ρομπότ με τυχόν εμπόδια που βρίσκονται μέσα στο χώρο εργασίας. Ο χώρος στον οποίο το ρομπότ δεν έχει πρόσβαση καλείται απαγορευμένος (forbidden space) και συμβολίζεται με $C_{forb}(R,S)$. Ο υπόλοιπος χώρος καλείται ελεύθερος χώρος (free space) και συμβολίζεται με $C_{free}(R,S)$. Ένα μονοπάτι το οποίο μπορεί να ακολουθήσει ένα

ρομπότ χωρίς όμως να συγκρούεται με τα εμπόδια αντιστοιχεί σε μια καμπύλη μέσα στον ελεύθερο χώρο C_{free} . Στην παρακάτω εικόνα μπορούμε να δούμε στο αριστερό μέρος την κίνηση του ρομπότ από μια αρχική σε μια τελική θέση μέσα στο χώρο εργασίας και στο δεξιό μέρος το χώρο διαμόρφωσης όπου ο απαγορευμένος χώρος είναι η σκιασμένη γκρι περιοχή:

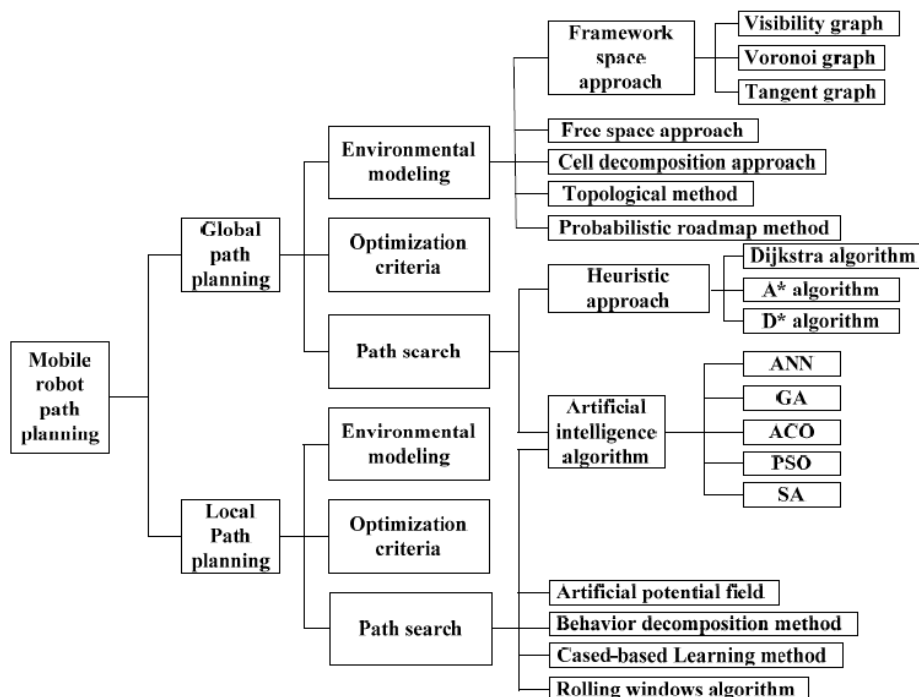


Εικόνα 4 Ένα μονοπάτι στο χώρο εργασίας και η αντίστοιχη καμπύλη στο χώρο διαμόρφωσης (πηγή: Computational Geometry, 2008)

Αφού μπορούμε να αναπαραστήσουμε τις μετατοπίσεις του ρομπότ ως σημεία και τα μονοπάτια που αυτό μπορεί να ακολουθήσει ως καμπύλες που ανήκουν στο χώρο διαμόρφωσης, μπορούμε να αναπαραστήσουμε και ένα εμπόδιο P ως το σύνολο των σημείων p του χώρου διαμόρφωσης για τα οποία η μετατόπιση $R(p)$ τέμνει το P . Το σύνολο αυτό ονομάζεται χώρος διαμόρφωσης εμποδίου (configuration –space obstacle) ή αλλιώς $C_{obstacle}$ του εμποδίου P . Στη συνέχεια της εργασίας θα θεωρούμε ότι το ρομπότ δεν συγκρούεται με τα εμπόδια όταν έρχεται σε επαφή μαζί τους, δηλαδή όταν κινείται κατά μήκος μιας ακμής ενός πολυγωνικού εμποδίου.

2.2 Αναπαραστάσεις Χώρου Διαμόρφωσης

Το πρόβλημα του σχεδιασμού κίνησης μπορεί να χωριστεί σε δύο μεγάλες κατηγορίες. Στην πρώτη κύρια κατηγορία η απαραίτητη πληροφορία για τη σχεδίαση της διαδρομής (θέσεις εμποδίων, κλπ.) είναι εκ των προτέρων γνωστή (global path planning), ενώ στη δεύτερη το ρομπότ έχει στη διάθεσή του μόνο την τρέχουσα πληροφορία που λαμβάνει από τους αισθητήρες του (local path planning). Στην επόμενη εικόνα φαίνεται η κατηγοριοποίηση του προβλήματος εύρεσης διαδρομής:



Εικόνα 5 Κατηγοριοποίηση του προβλήματος εύρεσης διαδρομής (πηγή: Zhang, 2018)

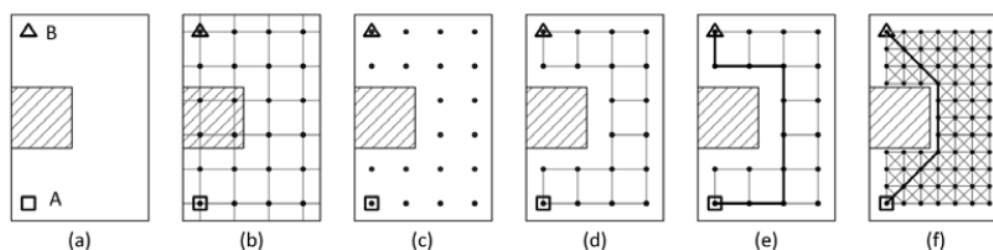
Η παρούσα εργασία ασχολείται με την πρώτη κύρια κατηγορία προβλημάτων (global path planning). Πιο συγκεκριμένα, στο Γενετικό Αλγόριθμο #1, αφού κατασκευάσουμε το γράφο ορατότητας με κόμβους τις κορυφές των πολυγωνικών εμποδίων και ακμές τα ευθύγραμμα τμήματα τα οποία ενώνουν τις ορατές κορυφές, θα χρησιμοποιήσουμε γενετικές διαδικασίες για να βρούμε το πιο σύντομο μονοπάτι από ένα τυχαίο κόμβο-εκκίνησης προς ένα τυχαίο κόμβο προορισμού. Στο Γενετικό Αλγόριθμο #2 υπολογίζουμε το γράφο ορατότητας με γενετικές διαδικασίες και στη συνέχεια χρησιμοποιούμε τον αλγόριθμο Dijkstra για να βρούμε το συντομότερο μονοπάτι.

Στη συνέχεια του Κεφαλαίου παρουσιάζουμε τις βασικές μεθόδους αναπαράστασης του χώρου διαμόρφωσης για το πρόβλημα του σχεδιασμού πλοήγησης διαμέσου εμποδίων. Σημειώνουμε ότι στη βιβλιογραφία συναντούμε αρκετές παραλλαγές των μεθόδων αυτών.

2.3 Απεικόνιση Χώρου Διαμόρφωσης με Χρήση Πλέγματος (Grid-based method)

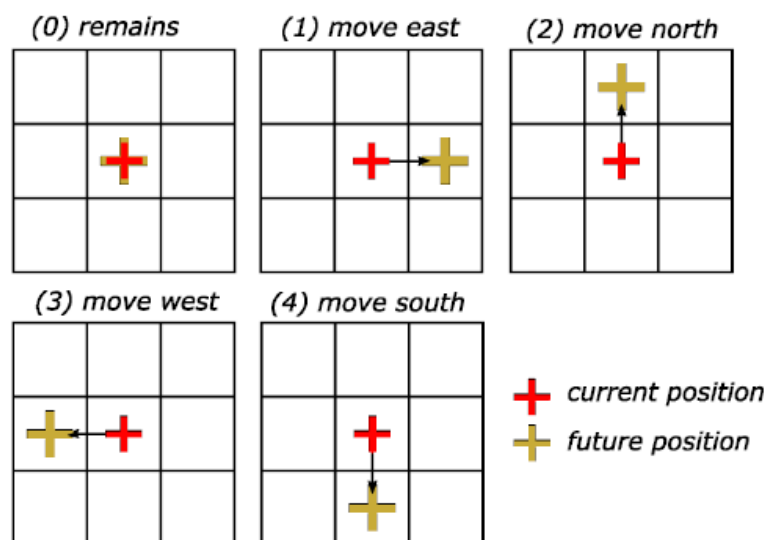
Η εύρεση της βέλτιστης διαδρομής απαιτεί αρχικά να κατασκευάσουμε μια κατάλληλη απεικόνιση (environmental modeling) των εμποδίων μέσα στο χώρο διαμόρφωσης. Μια από τις μεθόδους διακριτοποίησης του χώρου διαμόρφωσης είναι με χρήση τετραγωνικού πλέγματος (grid-based approach ή αλλιώς search-based approach ή Approximate cell decomposition). Σύμφωνα με αυτή ο χώρος καταστάσεων διαιρείται σε τετράγωνα

προκαθορισμένου μεγέθους τα οποία σχηματίζουν ένα πλέγμα όπως φαίνεται στην παρακάτω εικόνα:



Εικόνα 6 Απεικόνιση του χώρου διαμόρφωσης με χρήση πλέγματος (grid based approach) (πηγή: Andayesh, 2014)

Η χρήση του τετραγωνικού πλέγματος ως τρόπος διακριτοποίησης του χώρου διαμόρφωσης περιορίζει αυτόματα τον τρόπο κίνησης του ρομπότ. Στην παρακάτω εικόνα παρουσιάζονται οι επιτρεπτές κινήσεις ενός ρομπότ που μετακινείται μέσα σε ένα πλέγμα:



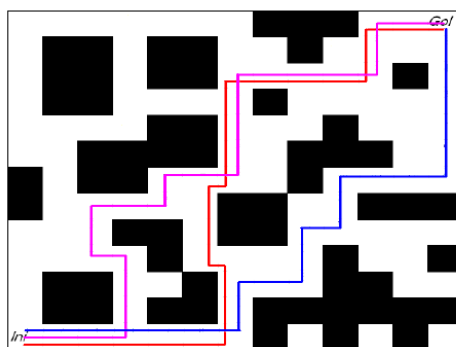
Εικόνα 7 Επιτρεπτές κινήσεις ενός ρομπότ όταν ο χώρος διαμόρφωσης διακριτοποιηθεί με χρήση πλέγματος (πηγή: Givigi, 2017)

Ο στόχος είναι να βρούμε τη συντομότερη διαδρομή από το σημείο A στο σημείο B αποφεύγοντας το εμπόδιο. Σχεδιάζουμε λοιπόν ένα τετραγωνικό πλέγμα όπου κάθε τετράγωνο έχει εμβαδόν $r \times r$. Όσο μικρότερη είναι η πλευρά r , τόσο μεγαλύτερη θα είναι η πυκνότητα του γράφου, άρα το ζητούμενο μονοπάτι θα έχει μεγαλύτερη ακρίβεια. Στη συνέχεια εντοπίζουμε όλους τους δυνατούς κόμβους. Κάθε κόμβος βρίσκεται στο σημείο τομής δυο ευθυγράμμων τμημάτων. Στη συνέχεια απαλείφουμε όσους κόμβους

βρίσκονται μέσα στο εμπόδιο. Έπειτα ενώνουμε τους εναπομείναντες κόμβους ώστε να δημιουργήσουμε τις ακμές του γράφου. Στην εικόνα 7b παρατηρούμε ότι κάθε κόμβος έχει το πολύ τέσσερις γείτονες. Τέλος στην εικόνα 7e εντοπίζουμε το πιο σύντομο μονοπάτι από τον κόμβο A στον κόμβο B χρησιμοποιώντας κάποιον γνωστό αλγόριθμο (π.χ., Dijkstra, Bellman-Ford, A*). Παρατηρούμε ότι στην εικόνα 7f, όπου τώρα κάθε κόμβος έχει 8 γείτονες, το μονοπάτι έχει μικρότερο μήκος σε σχέση με αυτό της εικόνας 7e. Η ποιότητα του μονοπατιού στο οποίο καταλήγει αυτή η μέθοδος εξαρτάται αποκλειστικά από την ανάλυση του πλέγματος. Όπως αναφέρει στην εργασία της η Ηλιακοπούλου (2012): «*Η αυξημένη πυκνότητα σημαίνει ότι περισσότερες περιοχές μπορούν να γίνουν προσβάσιμες μέσω του γράφου, ειδικά εκείνες που χαρακτηρίζονται από στενά περάσματα. Ταυτόχρονα όμως, η μεγάλη πυκνότητα υποδηλώνει αυξημένο πλήθος κόμβων στο γράφο, οι οποίοι επιβραδύνουν την εκτέλεση των αλγορίθμων διάσχισής του.*» Άρα λοιπόν, όταν τα κελιά του πλέγματος είναι μεγάλα σε μέγεθος, η μέθοδος αυτή αδυνατεί να βρει διαδρομές μέσα από στενά περάσματα του χώρου εργασίας.

2.4 Γράφος Δειγματοληψίας Πλέγματος και Γενετικοί Αλγόριθμοι (Grid-based Graph with Genetic Algorithms)

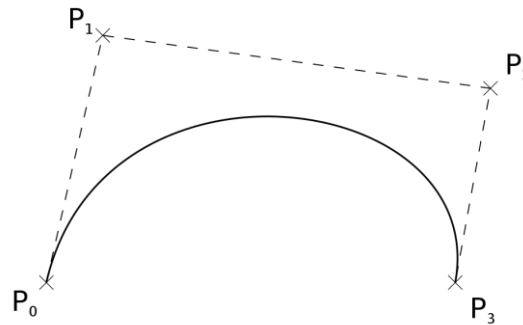
Στην εργασία τους οι AL-Taharwa et. al. (2008), αφού αναπαρέστησαν το χώρο διαμόρφωσης με χρήση πλέγματος (grid-based approach), στη συνέχεια χρησιμοποίησαν Γενετικούς Αλγορίθμους για να βρουν το συντομότερο μονοπάτι διαμέσου εμποδίων. Στην παρακάτω εικόνα φαίνονται τρία διαφορετικά μονοπάτια στα οποία καταλήγει η μέθοδός τους:



Εικόνα 8 Τρία συντομότερα μονοπάτια (με κόκκινο, ροζ και μπλε χρώμα) στα οποία καταλήγει ο Γενετικός Αλγόριθμος (πηγή: AL-Taharwa, 2008)

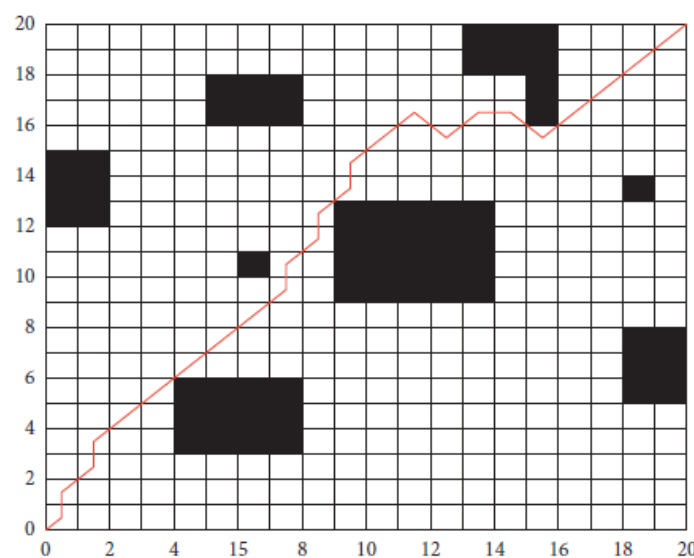
Σημειώνουμε ότι η κίνηση του ρομπότ στην Εικόνα 9 είναι είτε οριζόντια, είτε κάθετη. Μια λύση για τη βελτίωση της ποιότητας του μονοπατιού (path smoothing) που προκύπτει

όταν χρησιμοποιούμε αυτή τη μέθοδο παρουσιάζουν στο άρθρο τους οι Ma et. al. (2020). Η λύση που προτείνουν βασίζεται στη χρήση γενετικών αλγορίθμων καθώς και στη χρήση των καμπύλων Bezier. Μια καμπύλη Bezier ορίζεται από ένα σύνολο από σημεία ελέγχου (control points) από το P_0 έως το P_n , όπου n είναι η τάξη της καμπύλης (Wikipedia, 2021). Στην παρακάτω εικόνα φαίνεται μια καμπύλη Bezier με τέσσερα σημεία ελέγχου:



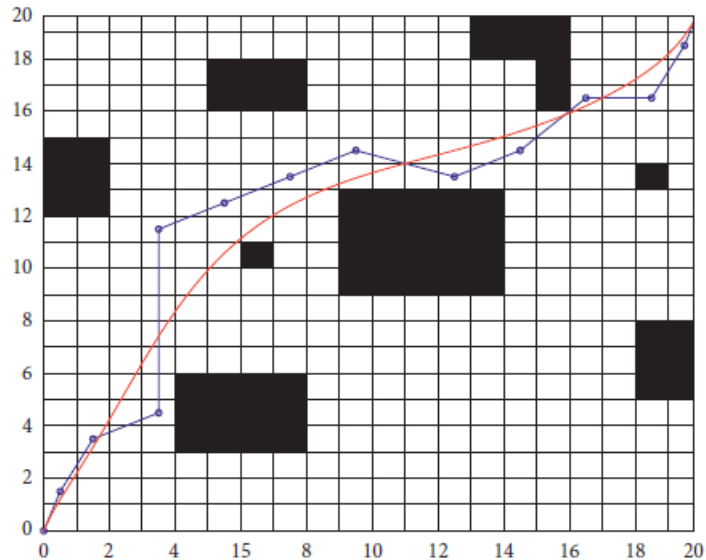
Εικόνα 9 Καμπύλη Bezier με τέσσερα σημεία ελέγχου (πηγή: Wikipedia)

Αρχικά χρησιμοποιούνται γενετικές διαδικασίες για να καθοριστούν τα σημεία ελέγχου. Έπειτα το μήκος της καμπύλης επιλέγεται ως η αντικειμενική συνάρτηση που θα πρέπει να βελτιστοποιηθεί έτσι ώστε ο γενετικός αλγόριθμος να συγκλίνει στο συντομότερο μονοπάτι. Στην παρακάτω εικόνα παρουσιάζεται το μονοπάτι που επιστρέφει ο γενετικός αλγόριθμος πριν την εξομάλυνση:



Εικόνα 10 Μονοπάτι που επιστρέφει ο γενετικός αλγόριθμος (χωρίς εξομάλυνση) (πηγή: Ma, 2020)

Στην παρακάτω εικόνα φαίνεται το μονοπάτι που προκύπτει μετά την εξομάλυνση με χρήση καμπύλων Bezier:



Εικόνα 11 Μονοπάτι (με κόκκινο χρώμα) που επιστρέφει ο γενετικός αλγόριθμος (με εξομάλυνση) (πηγή: Ma, 2020)

Η προσέγγιση με χρήση πλέγματος λειτουργεί αποδοτικά όσο το πλήθος των εμποδίων παραμένει μικρό. Όσο όμως το πλήθος των εμποδίων αυξάνεται τότε η προσέγγιση καταλήγει σε διαδρομές που δεν ικανοποιούν τα αρχικά κριτήρια ή στη χειρότερη περίπτωση η μέθοδος αυτή δεν επιστρέφει καμία λύση.

2.5 Οδικοί Χάρτες (Road maps)

Μια ακόμη μέθοδος για το σχεδιασμό πλοήγησης διαμέσου εμποδίων βασίζεται στην κατασκευή μιας ειδικής δομής που ονομάζεται οδικός χάρτης (road map). Ένας οδικός χάρτης είναι ένας γράφος ενσωματωμένος στον ελεύθερο χώρο C_{free} τον οποίο θα συμβολίζουμε με G_{road} . Στη συνέχεια παρουσιάζουμε διάφορες μεθόδους κατασκευής οδικών χαρτών.

2.5.1 Διαμέριση του Χώρου Διαμόρφωσης σε Κελιά (Cell Decomposition)

Οι αλγόριθμοι αυτού του είδους χωρίζουν τον ελεύθερο χώρο C_{free} σε περιοχές που ονομάζονται κελιά (cells). Η διαμέριση σε κελιά θα πρέπει να ικανοποιεί τρία κριτήρια (LaValle, 2006):

1. Ο υπολογισμός ενός μονοπατιού ανάμεσα σε δύο σημεία μέσα σε ένα κελί θα πρέπει να γίνεται εύκολα.
2. Η πληροφορία για το ποιοι είναι οι γείτονες κάθε κελιού θα πρέπει να είναι εύκολα διαθέσιμη.

3. Για δύο οποιαδήποτε σημεία εκκίνησης (q_I)-τερματισμού (q_G) θα πρέπει γρήγορα να εντοπίζονται τα κελιά που τα περιέχουν.

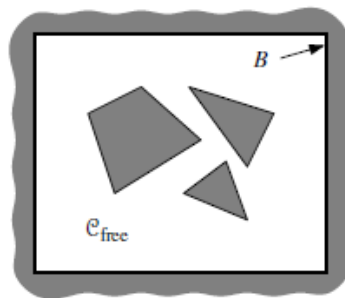
Όταν η διαμέριση του ελεύθερου χώρου σε κελιά ικανοποιεί τις τρεις παραπάνω προϋποθέσεις, τότε ο σχεδιασμός πλοήγησης μπορεί να μετατραπεί σε πρόβλημα αναζήτησης σε γράφο.

Υποθέτουμε ότι μέσα στο χώρο διαμόρφωσης υπάρχουν πολυγωνικά εμπόδια τα οποία δεν τέμνονται μεταξύ τους, καθώς και ότι ο συνολικός αριθμός των κορυφών είναι n . Συμβολίζουμε τα εμπόδια ως P_1, P_2, \dots, P_t .

Αντί να βρούμε ένα μονοπάτι από το σημείο εκκίνησης στο σημείο τερματισμού κατασκευάζουμε μια δομή δεδομένων στην οποία αποθηκεύουμε την αναπαράσταση του ελεύθερου χώρου C_{free} . Για να απλοποιήσουμε το πρόβλημα, περιορίζουμε την κίνηση του ρομπότ μέσα σε ένα ορθογώνιο παραλληλόγραμμο B το οποίο περιέχει τα εμπόδια. Ο ελεύθερος χώρος C_{free} αποτελείται τώρα από το μέρος του B το οποίο δεν καλύπτεται από εμπόδια δηλαδή:

$$C_{free} = B \setminus \bigcup_{i=1}^t P_i$$

Στην Εικόνα 12 βλέπουμε μια αναπαράσταση του ελεύθερου χώρου C_{free} :



Εικόνα 12 Free configuration space C_{free} (πηγή: Computational Geometry, 2008)

Για να αναπαραστήσουμε τον ελεύθερο χώρο μπορούμε να χρησιμοποιήσουμε τον τραπεζοειδή χάρτη (Παράρτημα Α). Ο αλγόριθμος Compute Free Space που παραθέτουμε παρακάτω χρησιμοποιεί τον αλγόριθμο Trapezoidal Map (Παράρτημα Β) ως υπορουτίνα.

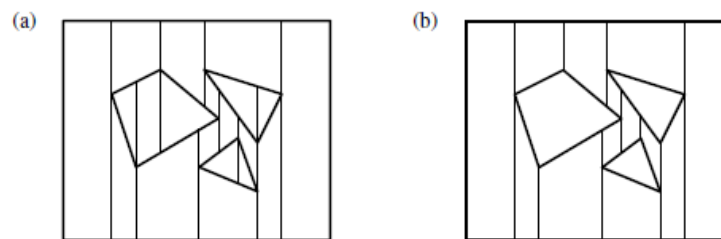
Algorithm COMPUTEFREE SPACE(S)

Input. A set S of disjoint polygons.

Output. A trapezoidal map of $C_{\text{free}}(\mathcal{R}, S)$ for a point robot \mathcal{R} .

1. Let E be the set of edges of the polygons in S .
2. Compute the trapezoidal map $\mathcal{T}(E)$ with algorithm **TRAPEZOIDALMAP** described in Chapter 6.
3. Remove the trapezoids that lie inside one of the polygons from $\mathcal{T}(E)$ and return the resulting subdivision.

Ο αλγόριθμος δέχεται σαν είσοδο ένα σύνολο S από πολυγωνικά εμπόδια τα οποία δεν τέμνονται μεταξύ τους και επιστρέφει τον τραπεζοειδή χάρτη του συνόλου αυτού. Στην Εικόνα 13 φαίνεται στο αποτέλεσμα εκτέλεσης του αλγορίθμου:



Εικόνα 13 Τραπεζοειδής χάρτης του ελεύθερου χώρου C_{free} (πηγή: Computational Geometry, 2008)

Παρατηρούμε στην Εικόνα 13(b) ότι έχουν αφαιρεθεί τα τραπέζια που βρίσκονται στο εσωτερικό των εμποδίων. Αυτό γίνεται στη γραμμή 3 του αλγορίθμου Compute Free Space. Αυτό μπορούμε να το καταφέρουμε επειδή γνωρίζουμε για κάθε τραπέζιο Δ την ακμή $\text{top}(\Delta)$ και επίσης γνωρίζουμε σε ποιο εμπόδιο ανήκει αυτή η ακμή.

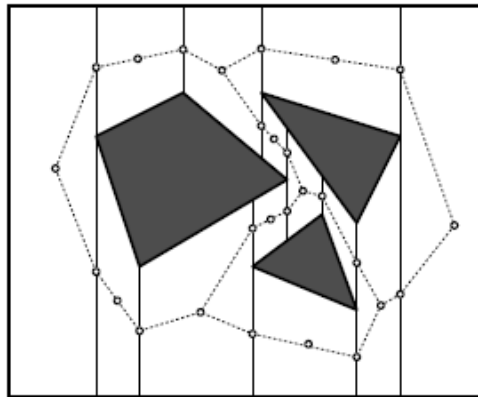
Ο χρόνος κατασκευής του παραπάνω τραπεζοειδούς χάρτη ισούται με $O(n \log n)$ (Παράρτημα Β). Θα συμβολίζουμε τον χάρτη του ελεύθερου χώρου ως εξής: $T(C_{\text{free}})$.

Πώς μπορούμε να χρησιμοποιήσουμε το χάρτη αυτόν για να βρούμε ένα μονοπάτι από ένα σημείο εκκίνησης p_{start} προς ένα σημείο τερματισμού p_{goal} ;⁵

Αν τα δύο σημεία βρίσκονται μέσα στο ίδιο τραπέζιο τότε το ζητούμενο μονοπάτι είναι απλώς η ευθεία γραμμή που ενώνει τα δύο σημεία. Αν βρίσκονται σε διαφορετικά τραπέζια τότε θα πρέπει να κατασκευάσουμε έναν οδικό χάρτη (road map), δηλαδή έναν γράφο ενσωματωμένο στον ελεύθερο χώρο C_{free} τον οποίο θα συμβολίζουμε με G_{road} . Καταρχήν θα πρέπει να παρατηρήσουμε ότι δύο γειτονικά τραπέζια μοιράζονται μία κοινή κάθετη ακμή. Τοποθετούμε λοιπόν έναν κόμβο στο κέντρο κάθε τραπέζιου και έναν

⁵ Παρατηρούμε ότι όλα τα τραπέζια στα οποία έχει διαμεριστεί ο ελεύθερος χώρος είναι κυρτά πολύγωνα, δηλαδή για οποιαδήποτε δύο σημεία μέσα σε ένα τραπέζιο υπάρχει μια ακμή που τα ενώνει.

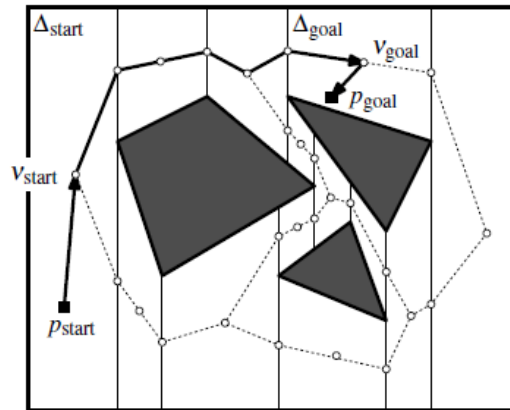
κόμβο στο μέσο καθεμιάς από τις κάθετες ακμές. Μια ακμή μεταξύ δύο κόμβων υπάρχει μόνο αν ο πρώτος βρίσκεται στο κέντρο του τραπεζίου και ο δεύτερος στη μια πλευρά του ίδιου τραπεζίου. Αν λοιπόν ακολουθήσουμε την ακμή ανάμεσα σε δύο κόμβους του οδικού χάρτη G_{road} , αυτό ισοδυναμεί με κίνηση του ρομπότ σε ευθεία γραμμή. Τα παραπάνω φαίνονται πιο απτά στην Εικόνα 14:



Εικόνα 14 Οδικός χάρτης (πηγή: Computational Geometry, 2008)

Μπορούμε να κατασκευάσουμε τον οδικό χάρτη σε χρόνο $O(n)$ χρησιμοποιώντας τη δομή δεδομένων D του τραπεζοειδούς χάρτη (βλέπε Εικόνα 186). Έπειτα, χρησιμοποιώντας τα τόξα του οδικού χάρτη μπορούμε να μετακινηθούμε από ένα κόμβο στο κέντρο ενός τραπεζίου προς εάν άλλον κόμβο στο κέντρο ενός διαφορετικού τραπεζίου διάμεσου των κόμβων των κοινών ακμών.

Για να σχεδιάσουμε ένα οποιοδήποτε μονοπάτι από ένα αρχικό σημείο p_{start} προς ένα τελικό σημείο p_{goal} πρώτα εντοπίζουμε τα τραπέζια Δ_{start} και Δ_{goal} που περιέχουν τα αντίστοιχα σημεία. Αν είναι τα ίδια τότε δεν μπορούμε παρά να μετακινηθούμε από το p_{start} στο p_{goal} σε ευθεία γραμμή. Αλλιώς συμβολίζουμε με v_{start} και v_{goal} τους κόμβους στα κέντρα των τραπεζίων Δ_{start} και Δ_{goal} . Χωρίζουμε το μονοπάτι από το p_{start} στο p_{goal} σε τρία μέρη. Το πρώτο μέρος είναι η ευθεία που ενώνει τα σημεία p_{start} και v_{start} , το δεύτερο μέρος είναι το μονοπάτι από το v_{start} στο v_{goal} και το τρίτο μέρος είναι η ευθεία από το v_{goal} στο p_{goal} . Το ζητούμενο μονοπάτι απεικονίζεται στην παρακάτω εικόνα:



Εικόνα 15 Διαδρομή από το σημείο p_{start} στο σημείο p_{goal} (πηγή: Computational Geometry, 2008)

Ο αλγόριθμος με τον οποίο υπολογίζουμε το ζητούμενο μονοπάτι παρουσιάζεται στην Εικόνα 16:

Algorithm COMPUTEPATH($\mathcal{T}(\mathcal{C}_{free}), \mathcal{G}_{road}, p_{start}, p_{goal}$)

Input. The trapezoidal map $\mathcal{T}(\mathcal{C}_{free})$ of the free space, the road map \mathcal{G}_{road} , a start position p_{start} , and goal position p_{goal} .

Output. A path from p_{start} to p_{goal} if it exists. If a path does not exist, this fact is reported.

1. Find the trapezoid Δ_{start} containing p_{start} and the trapezoid Δ_{goal} containing p_{goal} .
2. **if** Δ_{start} or Δ_{goal} does not exist
3. **then** Report that the start or goal position is in the forbidden space.
4. **else** Let v_{start} be the node of \mathcal{G}_{road} in the center of Δ_{start} .
5. Let v_{goal} be the node of \mathcal{G}_{road} in the center of Δ_{goal} .
6. Compute a path in \mathcal{G}_{road} from v_{start} to v_{goal} using breadth-first search.
7. **if** there is no such path
8. **then** Report that there is no path from p_{start} to p_{goal} .
9. **else** Report the path consisting of a straight-line motion from p_{start} to v_{start} , the path found in \mathcal{G}_{road} , and a straight-line motion from v_{goal} to p_{goal} .

Εικόνα 16 Αλγόριθμος Compute Path (πηγή: Computational Geometry, 2008)

Ο αλγόριθμος **Compute Path** παίρνει σαν είσοδο τον τραπεζοειδή χάρτη του ελεύθερου χώρου $T(C_{free})$, τον οδικό χάρτη G_{road} , το σημείο εκκίνησης p_{start} και το σημείο τερματισμού p_{goal} . Το πρώτο ερώτημα που τίθεται είναι αν το μονοπάτι που επιστρέφει ο παραπάνω αλγόριθμος προβλέπει έτσι ώστε να αποφεύγονται συγκρούσεις με τυχόν εμπόδια, και το δεύτερο αν πάντα μπορούμε να βρούμε ένα τέτοιο μονοπάτι. Η απάντηση στο πρώτο ερώτημα είναι θετική αφού το μονοπάτι αποτελείται από τμήματα τα οποία διέρχονται μέσα από τα τραπέζια και κάθε τραπέζιο ανήκει στον ελεύθερο χώρο C_{free} . Όσον αφορά το δεύτερο ερώτημα ας υποθέσουμε ότι ένα τέτοιο μονοπάτι όντως υπάρχει. Τότε τα σημεία p_{start} και p_{goal} θα πρέπει να βρίσκονται σε τραπέζια που ανήκουν στον ελεύθερο χώρο οπότε θα πρέπει να αποδείξουμε ότι υπάρχει ένα μονοπάτι από τον κόμβο v_{start} προς τον κόμβο v_{goal} . Το μονοπάτι από το σημείο εκκίνησης προς το σημείο τερματισμού πρέπει να διασχίζει μια σειρά από τραπέζια $\Delta_1, \Delta_2, \dots, \Delta_k$. Εξ ορισμού $\Delta_1 = \Delta_{start}$ και $\Delta_k = \Delta_{goal}$. Ας υποθέσουμε ότι v_i είναι ο κόμβος του τραpezίου Δ_i . Αν το μονοπάτι προχωράει από το Δ_i στο Δ_{i+1} τότε τα δύο τραπέζια πρέπει να είναι γείτονες άρα μοιράζονται μια κοινή κάθετη ακμή. Όμως ο γράφος G_{road} έχει κατασκευαστεί έτσι ώστε δύο κόμβοι γειτονικών τραpezίων να ενώνονται μέσω ενός κόμβου στην κοινή τους ακμή. Οπότε υπάρχει ένα μονοπάτι, το οποίο αποτελείται από δύο ακμές, από τον κόμβο v_i προς τον v_{i+1} . Ως εκ τούτου υπάρχει επίσης ένα μονοπάτι από τον κόμβο v_1 προς τον κόμβο v_k . Ο αλγόριθμος Αναζήτηση κατά Πλάτος (BFS) εξετάζει πρώτα τους γείτονες του τρέχοντος κόμβου πριν προχωρήσει στους κόμβους του επόμενου επιπέδου και βρίσκει ένα μονοπάτι από τον κόμβο v_{start} στον κόμβο v_{goal} .

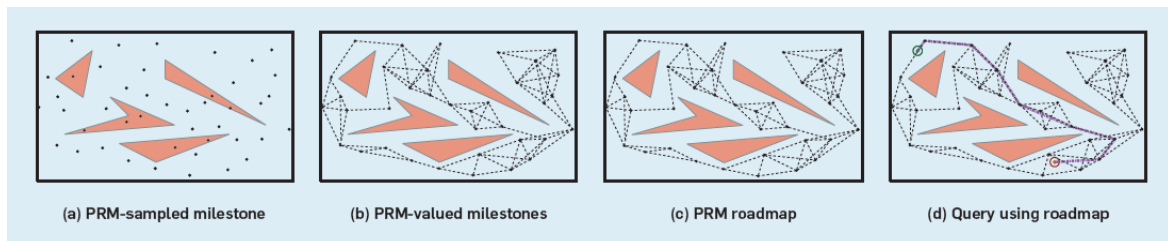
Για να υπολογίσουμε τα τραπέζια Δ_{start} και Δ_{goal} μέσα στα οποία βρίσκονται τα σημεία p_{start} και p_{goal} χρειαζόμαστε χρόνο $O(\log n)$ (Παράρτημα Β). Επίσης ο χρόνος που χρειάζεται ο αλγόριθμος Αναζήτηση κατά Πλάτος για να υπολογίσει ένα μονοπάτι είναι $O(n)$. Μετά από αυτές τις παρατηρήσεις μπορούμε να παραθέσουμε το παρακάτω Θεώρημα:

Έστω ότι R είναι ένα σημειακό ρομπότ το οποίο κινείται ανάμεσα σε ένα σύνολο S από πολυγωνικά εμπόδια, τα οποία αποτελούνται συνολικά από n ακμές. Μπορούμε να επεξεργαστούμε το σύνολο S σε χρόνο $O(n \log n)$ έτσι ώστε να υπολογίσουμε ένα μονοπάτι ανάμεσα σε δύο οποιαδήποτε σημεία p_{start} και p_{goal} σε χρόνο $O(n)$.

Αν και ο αλγόριθμος Compute Path μας εγγυάται ότι το μονοπάτι που υπολογίζεται δεν ενέχει κίνδυνο σύγκρουσης με κάποιο εμπόδιο, εντούτοις δεν εγγυάται ότι το μονοπάτι είναι και το συντομότερο.

2.5.2 Πιθανολογική Χάραξη Πορείας (Probabilistic Road-Map Method-PRM)

Η μέθοδος αυτή προτάθηκε από τους Kavraki et. al. (1996). Ο οδικός χάρτης σύμφωνα με τη μέθοδο αυτή κατασκευάζεται προοδευτικά. Κατά την πρώτη φάση της μεθόδου επιλέγονται τυχαία σημεία μέσα στον ελεύθερο χώρο C_{free} . Τα σημεία αυτά αποτελούν τους κόμβους N ενός μη κατευθυνόμενου γράφου $R = (N, E)$. Για κάθε κόμβο c μέσα στο σύνολο N επιλέγουμε ένα υποσύνολο κόμβων του N και προσπαθούμε να συνδέσουμε τον τρέχοντα κόμβο με καθένα από τους υπολοίπους. Αν υπάρχει ένα μονοπάτι (local path) που να συνδέει τον κόμβο c με τον επιλεγμένο κόμβο n και το οποίο δεν τέμνει κάποιο εμπόδιο, τότε η ακμή (c, n) τοποθετείται στο σύνολο E . Ο στόχος είναι να δημιουργηθεί ένας όσο το δυνατόν καλύτερα συνεκτικός γράφος του οποίου οι κόμβοι κατανέμονται ομοιόμορφα μέσα στον ελεύθερο χώρο. Στη δεύτερη φάση ο οδικός χάρτης που κατασκευάστηκε χρησιμοποιείται για τη χάραξη της διαδρομής από μια αρχική διαμόρφωση s σε μια τελική διαμόρφωση g . Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα εκτέλεσης του αλγορίθμου PRM:



Εικόνα 17 Παράδειγμα μεθόδου PRM. a) Τυχαία επιλεγμένοι κόμβοι b) Τυχαία σημεία μέσα στον ελεύθερο χώρο c) Κατασκευή οδικού χάρτη d) Χάραξη διαδρομής (πηγή: Salzman, 2019)

Στο σημείο αυτό θα πρέπει να τονίσουμε ότι η βασική ιδέα της μεθόδου PRM είναι να χρησιμοποιεί τα τοπικά μονοπάτια (local paths) για να συνδέσει δύο κοντινά σημεία. Όσο πιο μεγάλη είναι η απόσταση μεταξύ δύο σημείων q και q' τόσο μεγαλύτερη είναι η πιθανότητα ότι το τοπικό μονοπάτι $\pi(q, q')$ θα διέρχεται διαμέσου κάποιου εμποδίου (Salzman, 2019). Επίσης ένα άλλο χαρακτηριστικό αυτής της μεθόδου είναι ότι όσο πιο μεγάλο είναι το τυχαίο δείγμα από τα σημεία που βρίσκονται μέσα στο χώρο διαμόρφωσης τόσο η πιθανότητα σύγκλισης του αλγορίθμου σε λύση τείνει στη μονάδα. Τέλος αν ο χρόνος κατασκευής του χάρτη είναι σύντομος τότε αυτός ενδέχεται να μην αναπαριστά πλήρως τη συνεκτικότητα του ελεύθερου χώρου C_{free} .

2.5.3 Διαγράμματα Voronoi

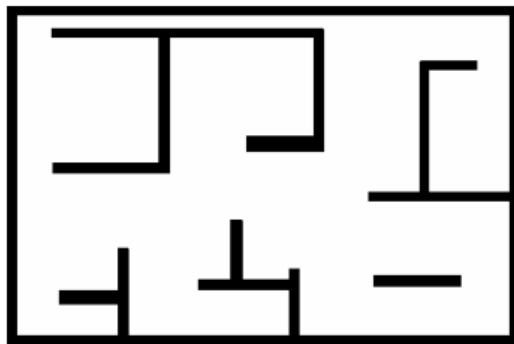
Μια άλλη μέθοδος κατασκευής οδικών χαρτών είναι με χρήση διαγραμμάτων Voronoi. Η μέθοδος αυτή έχει το πλεονέκτημα ότι ο οδικός χάρτης απέχει όσο το δυνατόν περισσότερο από τα εμπόδια. Έστω F ένα σύνολο κόμβων στο επίπεδο. Για κάθε κόμβο p αντιστοιχούμε μια περιοχή τέτοια ώστε για κάθε σημείο q που βρίσκεται μέσα σε αυτή να ισχύει:

$$V_F(p) = \{q: d(q, p) \leq d(q, F/\{p\})\}$$

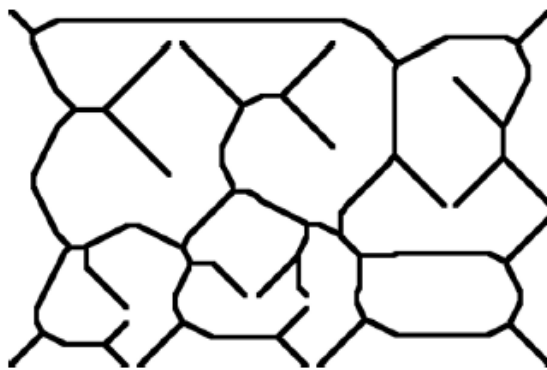
Το διάγραμμα Voronoi του συνόλου F είναι η ένωση των συνόρων όλων των περιοχών Voronoi (Garrido, 2006):

$$VD(F) = \bigcup_{p \in F} \partial V_F(p)$$

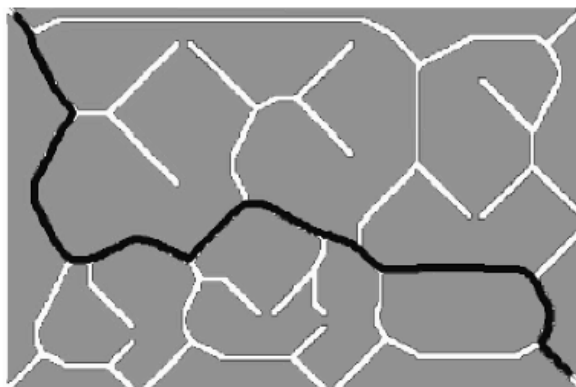
Τα ευθύγραμμα τμήματα που χωρίζουν τις περιοχές Voronoi λέγονται ακμές Voronoi. Στην εικόνα 18 απεικονίζονται τα εμπόδια μέσα στο χώρο διαμόρφωσης, στην Εικόνα 19 το διάγραμμα Voronoi και στην Εικόνα 20 η διαδρομή από μια αρχική σε μια τελική θέση:



Εικόνα 18 Αποτύπωση του χώρου διαμόρφωσης (πηγή: Garrido, 2006)

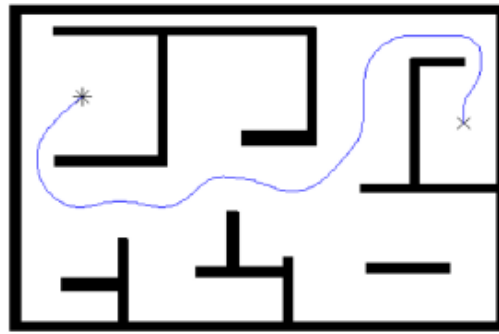


Εικόνα 19 Διάγραμμα Voronoi (πηγή: Garrido, 2006)



Εικόνα 20 Διαδρομή αποφυγής εμποδίων για μετακίνηση από αρχικό σε τελικό σημείο (πηγή: Garrido, 2006)

Μια μέθοδος που χρησιμοποιεί το διάγραμμα Voronoi μαζί με τη μέθοδο Fast Marching προτάθηκε από τους Garrido et. al. (2006). Η μέθοδος αυτή αφαιρεί από το διάγραμμα Voronoi εμπόδια τα οποία δημιουργούν στενά περάσματα. Έπειτα χρησιμοποιώντας τη μέθοδο Fast Marching βρίσκει ένα ασφαλές μονοπάτι το οποίο είναι απαλλαγμένο από συχνές εναλλαγές κατεύθυνσης. Ένα παράδειγμα εφαρμογής της μεθόδου αυτής φαίνεται στην παρακάτω εικόνα:

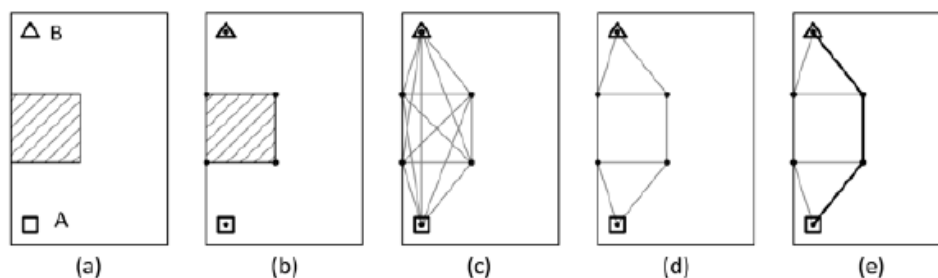


Εικόνα 21 Σχεδίαση διαδρομής με χρήση διαγράμματος Voronoi και της μεθόδου Fast Marching (πηγή: Garrido, 2006)

Είναι σημαντικό να τονίσουμε ότι η μέθοδος εύρεσης της συντομότερης διαδρομής με χρήση του διαγράμματος Voronoi επιχειρεί να κατασκευάσει έναν οδικό χάρτη ο οποίος αναπαριστά πλήρως τον ελεύθερο χώρο C_{free} , με μια μόνο προσπάθεια (Kavraki, 1996) σε αντίθεση με τη μέθοδο PRM η οποία επιχειρεί να κατασκευάσει το χάρτη σταδιακά.

2.5.4 Γράφος Ορατότητας (Visibility Graph)

Η απεικόνιση του χώρου διαμόρφωσης με χρήση γράφου ορατότητας φαίνεται στην παρακάτω εικόνα:



Εικόνα 22 Σχεδίαση γράφου ορατότητας για την εύρεση της συντομότερης διαδρομής (πηγή: Andayesh, 2014)

Οι κόμβοι του γράφου ορατότητας ταυτίζονται με τις κορυφές των εμποδίων και τους κόμβους εκκίνησης – τερματισμού. Κάθε ακμή παριστάνει μια σύνδεση δύο κόμβων όταν οι δύο κόμβοι είναι αμοιβαία ορατοί. Τα βήματα για τη δημιουργία του γράφου ορατότητας είναι τα ακόλουθα:

1. Εντοπισμός των κόμβων του γράφου: κορυφές των εμποδίων και οι δύο κόμβοι εκκίνησης-τερματισμού.
2. Σύνδεση όλων των κόμβων ανά δύο έτσι ώστε να σχηματίσουμε όλες τις δυνατές ακμές του γράφου.
3. Διαγραφή των ακμών που διέρχονται διαμέσου των εμποδίων.
4. Αναζήτηση της συντομότερης διαδρομής από το σημείο εκκίνησης στο σημείο τερματισμού.

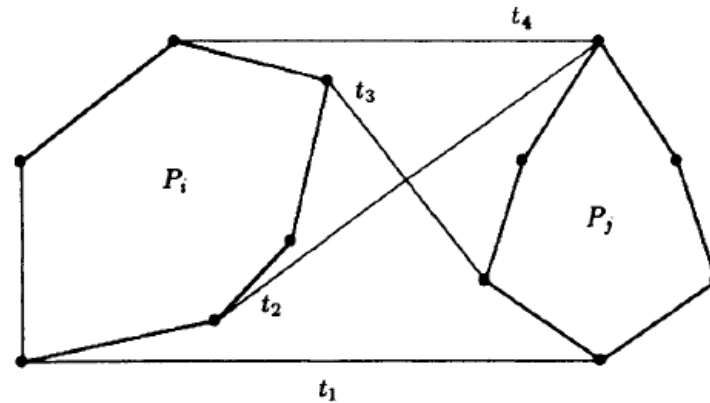
Πιο συγκεκριμένα στην εικόνα 22b εντοπίζουμε όλους τους κόμβους του γράφου. Στην εικόνα 22c ενώνουμε όλους τους κόμβους ανά δύο ώστε να δημιουργήσουμε μια δεξαμενή ακμών. Στη συνέχεια κάθε ακμή θα συγκριθεί με όλα τα εμπόδια που βρίσκονται στο χώρο εργασίας. Οι ακμές που διέρχονται διαμέσου των εμποδίων διαγράφονται οπότε καταλήγουμε στην εικόνα 22d. Τέλος αναζητούμε τη συντομότερη διαδρομή μεταξύ των σημείων A και B χρησιμοποιώντας κάποιον γνωστό αλγόριθμο αναζήτησης (π.χ., Dijkstra).

Η προσέγγιση με χρήση γράφου ορατότητας οδηγεί σε διαδρομές με μεγαλύτερη ακρίβεια που ικανοποιούν τα αρχικά κριτήρια (π.χ., η διαδρομή να είναι η συντομότερη δυνατή). Ο χρόνος εκτέλεσης του αλγορίθμου σε αυτή την περίπτωση θα είναι το άθροισμα του χρόνου που χρειάζεται για να κατασκευάσουμε το γράφο συν το χρόνο που χρειάζεται για να αναζητήσουμε το συντομότερο μονοπάτι. Μόλις κατασκευάσουμε το γράφο ορατότητας τότε χρησιμοποιώντας κάποιο γνωστό αλγόριθμο (Dijkstra, A*, Bellman-Ford, κλπ.) μπορούμε να εντοπίσουμε το συντομότερο μονοπάτι.

Η πολυπλοκότητα του αλγορίθμου κατασκευής του γράφου ορατότητας είναι $O(n^2)$. Ωστόσο για ένα χώρο διαμόρφωσης *C-space* με περισσότερες από δύο διαστάσεις το πρόβλημα γίνεται NP-hard (Goerzen, 2010).

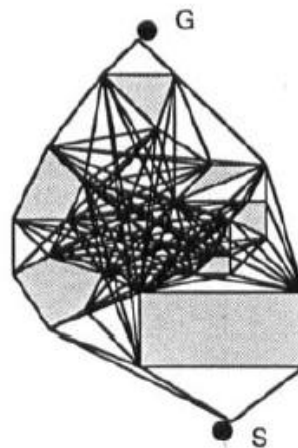
Ο Rohnert (1986) ασχολήθηκε με τη χάραξη μονοπατιού διαμέσου κυρτών πολυγώνων πλήθους f με χρήση του γράφου ορατότητας και απέδειξε ότι μπορεί να υπολογίσει το γράφο σε χρόνο $O(n + f^2 \log n)$. Η παρατήρηση που τον βοήθησε στον υπολογισμό του γράφου είναι ότι μεταξύ δύο κυρτών πολυγώνων υπάρχουν το πολύ τέσσερα υποστηρικτικά τμήματα ⁶(supporting segments) όπως φαίνεται στην παρακάτω εικόνα:

⁶ **Υποστηρικτικό** λέγεται το ευθύγραμμο τμήμα όπου ολόκληρο το πολύγωνο βρίσκεται μέσα στο ημι-επίπεδο που σχηματίζει η ευθεία μέρος της οποίας είναι το συγκεκριμένο ευθύγραμμο τμήμα (Sridharan, 2004)

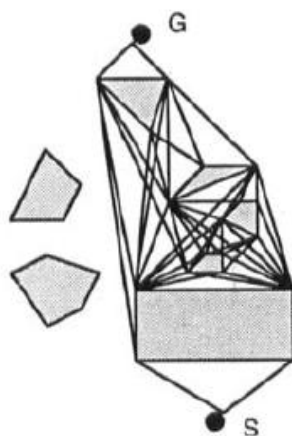


Εικόνα 23 Υποστηρικτικά τμήματα t_1, t_2, t_3, t_4 μεταξύ των πολυγώνων P_i και P_j (πηγή: Rohnert, 1986)

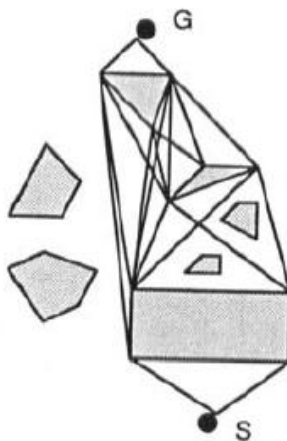
Στην εργασία τους οι Liu et. al. (Liu, 1989) επινόησαν μια μέθοδο με την οποία μειώνουν το μέγεθος του γράφου ορατότητας αφαιρώντας τα περιττά εμπόδια δηλαδή τα εμπόδια των οποίων οι κορυφές δεν αποτελούν ενδιάμεσους κόμβους του συντομότερου μονοπατιού. Με αυτό τον τρόπο δημιουργείται ένας υπογράφος με συνολικά $k < n$ εμπόδια. Ο συγκεκριμένος αλγόριθμος έχει χρονική πολυπλοκότητα $O(k^2n)$. Το κλειδί στη μέθοδο αυτή είναι να επιλεγούν σωστά τα εμπόδια που θα αφαιρεθούν. Στις επόμενες εικόνες φαίνεται πρώτα ο αρχικός (πλήρης) γράφος ορατότητας και στη συνέχεια δύο περιπτώσεις εφαρμογής της μεθόδου αυτής:



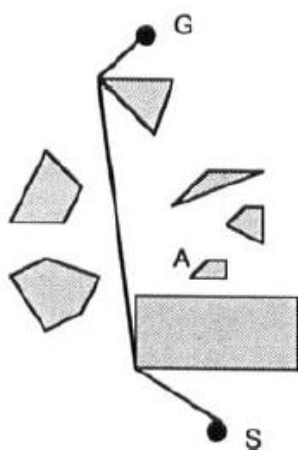
Εικόνα 24 Ο αρχικός (πλήρης) γράφος ορατότητας με 7 εμπόδια και 29 κορυφές (πηγή: Liu, 1989)



Εικόνα 25 Ο υπογράφος ορατότητας με 5 εμποδία και 20 κορυφές (πηγή: Liu, 1989)



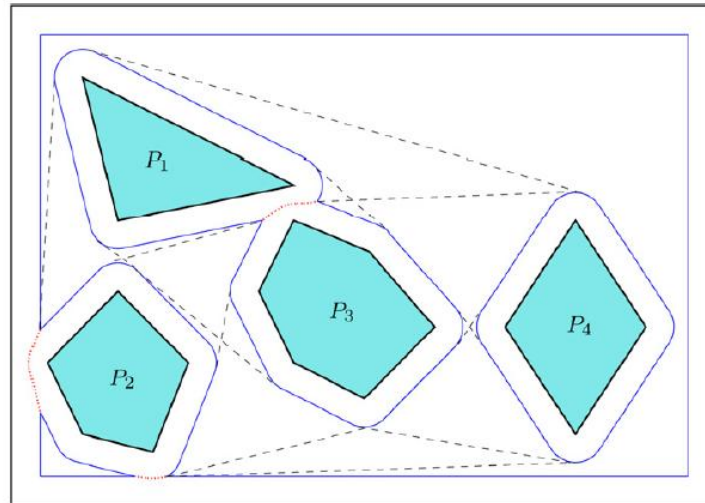
Εικόνα 26 Βελτιωμένος υπογράφος με μόνο 12 κορυφές. Ο υπογράφος αυτός προκύπτει με διαφορετική επιλογή περιττών εμποδίων κατά την εφαρμογή του αλγορίθμου (πηγή: Liu, 1989)



Εικόνα 27 Το συντομότερο μονοπάτι μεταξύ των σημείων S και G το οποίο προκύπτει μετά από οποιονδήποτε αλγόριθμο αναζήτησης π.χ. Dijkstra, A* (πηγή: Liu, 1989)

Μια ακόμη μέθοδος η οποία προτάθηκε από τους Wein et. al. (2005) συνδυάζει το γράφο ορατότητας με το διάγραμμα Voronoi ώστε να εξομαλύνει τις απότομες αλλαγές

κατεύθυνσης της διαδρομής που παρατηρούνται στις κορυφές των πολυγωνικών εμποδίων. Η μέθοδος τους εξασφαλίζει επίσης ότι το μονοπάτι θα είναι το πιο σύντομο και ότι θα απέχει από τα εμπόδια απόσταση ίση με μια προκαθορισμένη παράμετρο c . Όσο αυτή η παράμετρος αυξάνεται από το μηδέν στο άπειρο, ο γράφος ορατότητας μετατρέπεται στο αντίστοιχο διάγραμμα Voronoi. Το διάγραμμα $VV^{(c)}$ που προκύπτει με την μέθοδο αυτή απεικονίζεται παρακάτω:



Εικόνα 28 Το διάγραμμα $VV^{(c)}$ για τέσσερα κυρτά πολυγωνικά εμπόδια. Το σύνορο των εμποδίων απεικονίζεται με συμπαγή (μπλε) γραμμή, το τμήμα του διαγράμματος Voronoi απεικονίζεται με στικτή (κόκκινη) γραμμή και οι ακμές του γράφου ορατότητας απεικονίζονται με διακεκομμένη (μαύρη) γραμμή. (πηγή: Wein, 2005)

2.6 Μέθοδοι Βελτιστοποίησης (Improvement Methods)

Ο σκοπός της ενότητας αυτής είναι να παρουσιάσει κάποιες μεθόδους (metaheuristics) που χρησιμοποιούνται σε προβλήματα βελτιστοποίησης. Οι μέθοδοι αυτές αρχικοποιούνται με μια ή περισσότερες τυχαίες (ή μπορεί και όχι) λύσεις και στη συνέχεια μέσω συγκεκριμένων διαδικασιών προσπαθούν να εντοπίσουν τη βέλτιστη λύση (Dhaenens, 2002).

2.6.1 Γειτονικές Λύσεις

Για κάθε εφικτή λύση S μπορούμε να δημιουργήσουμε ένα σύνολο από διαφορετικές λύσεις οι οποίες όμως διαφέρουν ελάχιστα από την αρχική. Αυτό γίνεται μετασχηματίζοντας την αρχική λύση σε τοπικό επίπεδο διατηρώντας έτσι τις βασικές της ιδιότητες. Για παράδειγμα, στο πρόβλημα της εύρεσης του συντομότερου μονοπατιού από το σημείο A στο σημείο G έστω ότι η αρχική λύση είναι η $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$. Αν αλλάξουμε τις θέσεις των κόμβων B και C διατηρώντας όμως ως αρχικό και τελικό κόμβο τον κόμβο A και G αντίστοιχα (βασική ιδιότητα), τότε θα πάρουμε τη λύση $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow G$.

2.6.2 Μέθοδος Gradient

Σύμφωνα με τη μέθοδο αυτή στην περιοχή της τρέχουσας λύσης αναζητούμε μια άλλη λύση η οποία έχει μικρότερο κόστος (τιμή). Αν βρεθεί, θέτουμε αυτή ίση με την τρέχουσα λύση και συνεχίζουμε την αναζήτηση έως ότου φτάσουμε σε κάποιο τοπικό μέγιστο. Η εξερεύνηση της γειτονικής περιοχής μπορεί να γίνει είτε με τυχαίο τρόπο είτε εφαρμόζοντας κάποια κριτήρια ελέγχου, ώστε να οδηγούμαστε σε καλύτερες λύσεις ή ακόμη μπορούμε να εξερευνήσουμε όλες τις γειτονικές λύσεις ξεχωριστά και να επιλέξουμε την καλύτερη. Ειδική περίπτωση αυτής της μεθόδου είναι αυτή κατά την οποία επιλέγουμε έναν αριθμό $r < n$, έστω $r=3$ (όπου n είναι το πλήθος των κόμβων από το οποίο θέλουμε να αποτελείται η λύση) και έπειτα ελέγχοντας όλες τις πιθανές μεταθέσεις των τριών στοιχείων προσπαθούμε να βρούμε αν οδηγούμαστε σε καλύτερες λύσεις. Άρα λοιπόν όσο το $r \rightarrow n$, τόσο πιο κοντά οδηγούμαστε στην καλύτερη (τοπικά) λύση. Ο αλγόριθμος της μεθόδου αυτής παρουσιάζεται παρακάτω:

```

Method of descent
/* Initializations */
Select the initial solution  $S$ 
 $S_{better} \leftarrow S$ 
 $Cost(better) \leftarrow Cost(S)$ 
 $NbIteration \leftarrow 0$ 
/* Loop */
While  $NbIteration < NbMaximum$  do
     $NbIteration \leftarrow NbIteration + 1$ 
    Select  $S'$  in the neighborhood of  $S$ 
    If  $Cost(S') < Cost(S)$ 
        then  $Cost(better) \leftarrow Cost(S')$ 
             $S_{better} \leftarrow S'$ 
             $S \leftarrow S'$ 
    End If
End While

```

Εικόνα 29 Gradient method (πηγή: Dhaenens, 2002)

Η μέθοδος αυτή αν και είναι γρήγορη έχει το μειονέκτημα ότι μόλις όταν εντοπίσει ένα τοπικό ελάχιστο, εγκλωβίζεται σε αυτό.

2.6.3 Μέθοδος tabu

Σε κάθε επανάληψη του αλγορίθμου εξετάζουμε τη γειτονιά της τρέχουσας λύσης και επιλέγουμε εκείνη τη λύση η οποία μας οδηγεί στη μεγαλύτερη μείωση ή στη μικρότερη αύξηση του κόστους. Με αυτό τον τρόπο αν και δεν οδηγούμαστε πάντα σε καλύτερες λύσεις ωστόσο μας δίνεται η δυνατότητα να εξερευνήσουμε διαφορετικές γειτονιές. Για να αποφύγει ο αλγόριθμος να κλειδωθεί ανάμεσα σε δύο λύσεις, διατηρεί μια λίστα με τις λύσεις που εξερευνά κάθε φορά. Έτσι μια καινούρια λύση αντικαθιστά την παλαιότερη μέσα στη λίστα. Η μέθοδος αυτή απαιτεί μια συνθήκη τερματισμού, όπως π.χ., η μέθοδος να τερματίζει όταν η επανάληψη ξεπεράσει έναν προκαθορισμένο αριθμό.

2.6.4 Μέθοδος Ανόπτησης (Simulated Annealing)

Και σε αυτή τη μέθοδο θεωρούμε αρχικά μια τρέχουσα λύση και εξετάζουμε τις γειτονικές λύσεις για να βρούμε ποια από αυτές μπορεί να γίνει η τρέχουσα. Αρχικοποιούμε μια παράμετρο (T), και σε κάθε επανάληψη του αλγορίθμου μειώνουμε την τιμή της παραμέτρου σύμφωνα με κάποιον κανόνα. Όσο πιο αργά μειώνεται η παράμετρος T , τόσο πιθανότερο είναι να εντοπίσουμε τη βέλτιστη (global optimum) λύση με το αντίστοιχο όμως υπολογιστικό κόστος.

2.6.5 Γενετικοί Αλγόριθμοι (Genetic Algorithms)⁷

Αρχικά οι γενετικοί αλγόριθμοι θεωρούν ένα σύνολο από πιθανές λύσεις οι οποίες αποτελούν τον πληθυσμό (population). Στα άτομα του πληθυσμού εφαρμόζονται οι γενετικές διαδικασίες της διασταύρωσης, της μετάλλαξης και της επιλογής με στόχο να εκμεταλλευτούμε τα καλύτερα χαρακτηριστικά κάθε γενιάς. Έτσι κάθε γενιά αποτελείται από βελτιωμένες λύσεις, ενώ ταυτόχρονα αποφεύγεται η περίπτωση όπου ο αλγόριθμος εγκλωβίζεται σε κάποιο τοπικό ελάχιστο. Η δυσκολία στην εφαρμογή γενετικών αλγορίθμων βρίσκεται στην κατάλληλη κωδικοποίηση των λύσεων αλλά και στην εφαρμογή των γενετικών τελεστών. Σε προβλήματα μετάθεσης⁸ όπως είναι το πρόβλημα σχεδίασης του συντομότερου μονοπατιού, ο στόχος είναι η εύρεση της κατάλληλης θέσης κάθε χαρακτήρα, δηλαδή κάθε κόμβου στο ζητούμενο πρόβλημα (κάθε κόμβος εμφανίζεται μόνο μια φορά). Κατά το στάδιο της επιλογής επιλέγονται ως γονείς τα άτομα με τη μεγαλύτερη τιμή αντικειμενικής συνάρτησης. Αν η τιμή κάθε ατόμου του πληθυσμού είναι f_i , ενώ η συνολική τιμή της αντικειμενικής συνάρτησης του πληθυσμού είναι F , τότε η πιθανότητα να επιλεγεί κάποιο άτομο ισούται με f_i/F . Κατά το στάδιο της

διασταύρωσης από δύο γονείς προκύπτουν ένα ή δυο παιδιά τα οποία κληρονομούν τα χαρακτηριστικά των γονιών τους. Στο στάδιο αυτό μπορούν να προκύψουν μη εφικτές λύσεις, δηλαδή λύσεις στις οποίες κάποιοι κόμβοι να επαναλαμβάνονται. Το στάδιο της μετάλλαξης προσφέρει γενετική ποικιλομορφία, οπότε καινούριες λύσεις μπορούν να εισέλθουν στον πληθυσμό. Αν και οι αλγόριθμοι αυτού του είδους βρίσκουν πολλές λύσεις, παρουσιάζουν το μειονέκτημα ότι πρέπει να γίνει σωστή επιλογή διάφορων παραμέτρων όπως το μέγεθος του πληθυσμού, η συχνότητα μετάλλαξης, κλπ.

⁷ Οι Γενετικοί Αλγόριθμοι αναλύονται περισσότερο στο Κεφάλαιο 3

⁸ Προβλήματα μετάθεσης ονομάζονται τα προβλήματα των οποίων οι πιθανές λύσεις αναπαρίστανται ως μεταθέσεις κάποιων αντικειμένων. Για παράδειγμα στο πρόβλημα του πλανόδιου πωλητή αν οι π.χ., πέντε πόλεις που πρέπει να επισκεφθεί είναι οι Α,Β,Γ,Δ,Ε τότε μία λύση (ξεκινώντας από την Α) θα μπορούσε να είναι η Α-Ε-Γ-Β-Δ-Α.

3. Γενετικοί Αλγόριθμοι

Προβλήματα βελτιστοποίησης καλούνται τα προβλήματα στα οποία ψάχνουμε να βρούμε τη βέλτιστη λύση, όπως είναι η εύρεση του συντομότερου μονοπατιού από ένα σημείο Α σε ένα σημείο Β. Όπως αναφέρει ο Δημητρίου (2020, σελ. 51): «*Σε προβλήματα σαν αυτά, εκείνο που κάνουμε είναι να εξερευνήσουμε το σύνολο (χώρο) των πιθανών λύσεων του προβλήματος, για να βρούμε κάποια ή κάποιες που μας ικανοποιεί. Με άλλα λόγια αναζητούμε μία ή περισσότερες λύσεις μέσα από τον χώρο των πιθανών λύσεων, και γι' αυτό τον λόγο, τα προβλήματα αυτά ονομάζονται προβλήματα αναζήτησης (search problems).*»

3.1 Γενετικός Αλγόριθμος – μια πρώτη ματιά

Μια διαφορετική προσέγγιση για την επίλυση προβλημάτων βελτιστοποίησης επιτυγχάνεται με τη χρήση γενετικών αλγορίθμων. Η βασική ιδέα είναι η εξής: δεδομένου ενός πληθυσμού ατόμων μέσα σε ένα περιβάλλον με περιορισμένους πόρους, ο ανταγωνισμός για τους πόρους αυτούς οδηγεί στη φυσική επιλογή (Eiben & Smith, 2015). Στην αρχή δημιουργούμε ένα τυχαίο σύνολο από υποψήφιες λύσεις. Στη συνέχεια, χρησιμοποιούμε ένα μέτρο σύγκρισης των λύσεων, το οποίο ονομάζουμε *αντικειμενική συνάρτηση* (fitness function). Εφαρμόζουμε την αντικειμενική συνάρτηση πάνω σε αυτές και όποια λύση έχει τη μεγαλύτερη τιμή επιλέγεται για να παίξει το ρόλο του γονέα. Στη λύση που θα επιλεγεί εφαρμόζουμε διάφορους τελεστές. Ο ένας από αυτούς είναι η *διασταύρωση* (recombination) η οποία μπορεί να εφαρμοστεί σε δύο ή περισσότερους γονείς και παράγει μία ή περισσότερες υποψήφιες λύσεις-παιδιά. Ο δεύτερος τελεστής είναι η *μετάλλαξη* (mutation) η οποία εφαρμόζεται σε μια υποψήφια λύση και παράγει μια νέα υποψήφια λύση. Εφαρμόζοντας αυτούς τους τελεστές στους γονείς, κατασκευάζουμε ένα σύνολο από νέες υποψήφιες λύσεις. Τότε αυτές συγκρίνονται μεταξύ τους, με βάση την τιμή τους, αλλά και με τις παλιές οπότε και επιλέγονται οι καλύτερες για να δημιουργήσουν τη νέα γενιά. Αυτή η διαδικασία επαναλαμβάνεται μέχρι να βρεθεί η καλύτερη λύση, διαφορετικά επαναλαμβάνεται για ένα προκαθορισμένο αριθμό επαναλήψεων. Συνοπτικά οι γενετικοί τελεστές που εφαρμόζονται στους γενετικούς αλγορίθμους είναι οι ακόλουθοι:

- *διασταύρωση και μετάλλαξη*, οι οποίες δημιουργούν την απαραίτητη διαφοροποίηση στον πληθυσμό,

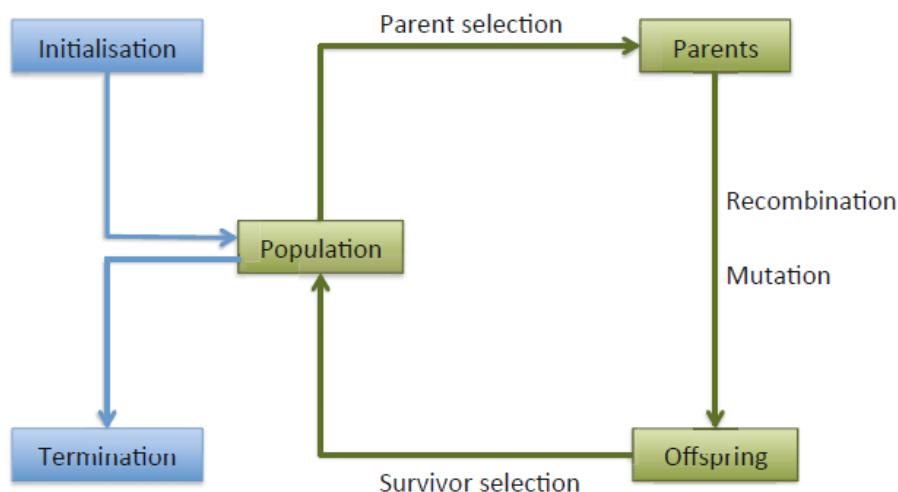
- *επιλογή*, με την οποία επιτυγχάνεται η αύξηση του μέσου όρου της ποιότητας των λύσεων.

Ο συνδυασμός αυτών των δύο τελεστών προκαλεί την ολοένα και αυξανόμενη τιμή της αντικειμενικής συνάρτησης των ατόμων του πληθυσμού με συνέπεια να προσεγγίζουμε την καλύτερη λύση με το πέρασμα των γενιών. Επίσης, μια άλλη οπτική γωνία μας δίνεται αν δούμε την αντικειμενική συνάρτηση όχι σαν μια συνάρτηση που πρέπει να βελτιστοποιηθεί αλλά σαν μια έκφραση των απαιτήσεων του περιβάλλοντος. Οπότε αυτά τα άτομα του πληθυσμού που καλύπτουν σε μεγαλύτερο βαθμό αυτές τις απαιτήσεις έχουν μεγαλύτερη πιθανότητα να δημιουργήσουν απογόνους.

Εδώ θα πρέπει να σημειώσουμε ότι πολλά από τα εργαλεία των γενετικών αλγορίθμων εμπεριέχουν την τυχειότητα ως παράμετρο. Για παράδειγμα, κατά την επιλογή, τα πιο κατάλληλα άτομα επιλέγονται τυχαία, άρα υπάρχει η πιθανότητα και στα άτομα με χαμηλή τιμή της αντικειμενικής συνάρτησης να επιλεγούν ως γονείς. Επίσης κατά τη διασταύρωση, το ποια από τα μέρη του χρωμοσώματος (δηλαδή των λύσεων) από κάθε γονέα θα ανασυνδυαστούν ώστε να προκύψει ο απόγονος, γίνεται με τυχαίο τρόπο. Ακόμη, κατά το στάδιο της μετάλλαξης, ο τρόπος με τον οποίο επιλέγεται το γονίδιο (περιοχή του χρωμοσώματος) το οποίο θα υποστεί τη μετάλλαξη γίνεται με τυχαίο τρόπο. Το γενικό σχήμα του γενετικού αλγορίθμου δίνεται σε ψευδοκώδικα στην Εικόνα 30 και σε διάγραμμα ροής στην Εικόνα 31:

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Εικόνα 30 Το γενικό σχήμα ενός γενετικού αλγορίθμου σε ψευδοκώδικα (πηγή: Introduction to Evolutionary Computing, 2015)



Εικόνα 31 Το γενικό σχήμα ενός γενετικού αλγορίθμου σε διάγραμμα ροής (πηγή: Introduction to Evolutionary Computing, 2015)

3.2 Βασικά Συστατικά Γενετικού Αλγορίθμου

Τα εργαλεία-βασικά συστατικά ενός γενετικού αλγορίθμου είναι:

- αναπαράσταση (ορισμός του ατόμου),
- αντικειμενική συνάρτηση (fitness function),
- πληθυσμός (population),
- μηχανισμός επιλογής γονιών (parent selection),
- γενετικοί τελεστές (διασταύρωση, μετάλλαξη),
- αντικατάσταση (survivor selection).

Παρακάτω εξετάζουμε το κάθε συστατικό ξεχωριστά.

3.2.1 Αναπαράσταση

Το πρώτο βήμα κατά την επίλυση ενός προβλήματος με χρήση γενετικού αλγορίθμου είναι να βρεθεί ο κατάλληλος τρόπος αναπαράστασης των πιθανών λύσεων ώστε να είναι δυνατή η αποθήκευση και επεξεργασία τους από τον υπολογιστή. Η αναπαράσταση ή αλλιώς κωδικοποίηση των δυνατών λύσεων ονομάζεται αλλιώς και γονότυπος. Επίσης οι δυνατές λύσεις ονομάζονται και χρωμοσώματα. Μια πιθανή αναπαράσταση για παράδειγμα είναι η δυαδική αναπαράσταση, σύμφωνα με την οποία ο γονότυπος

αποτελείται από μια σειρά από δυαδικά ψηφία. Αν χρησιμοποιήσουμε μήκος bit-string ίσο με 8 μπορούμε να αναπαραστήσουμε:

$$2^8 = 256 \text{ διαφορετικές λύσεις}$$

Ένα πρόβλημα που παρουσιάζεται με τη δυαδική αναπαράσταση είναι ότι κάθε bit έχει και διαφορετική σημασία με το πιο σημαντικό ψηφίο συνήθως να είναι το πρώτο από τα αριστερά. Αν λοιπόν θέλουμε να αναπαραστήσουμε ακέραιους ως δυαδικούς αριθμούς, είναι πιθανό μια μετάλλαξη σε ένα από τα δυαδικά ψηφία να έχει διαφορετική επίπτωση. Μία πιθανή λύση θα ήταν να χρησιμοποιήσουμε κώδικα Gray σύμφωνα με τον οποίο κάθε ακέραιος ο οποίος αναπαριστάται με αυτό τον κώδικα έχει απόσταση Hamming ίση με ένα. Ο κώδικας Gray για 4 δυαδικά ψηφία φαίνεται στην παρακάτω εικόνα:

	Gray code			
	4	3	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	1	0	0
9	1	1	0	1
10	1	1	1	1
11	1	1	1	0
12	1	0	1	0
13	1	0	1	1
14	1	0	0	1
15	1	0	0	0

Εικόνα 32 Κώδικας Gray (πηγή: Wikipedia)

Άλλες μορφές αναπαραστάσεων είναι με μορφή ακεραίων ή δεκαδικών αριθμών όταν για παράδειγμα θέλουμε να αναπαραστήσουμε φυσικές ποσότητες όπως μήκος, πλάτος, ύψος ή βάρος. Επίσης ένας άλλος τύπος αναπαράστασης είναι οι μεταθέσεις. Μετάθεση είναι η τοποθέτηση ενός συνόλου n στοιχείων σε μια σειρά. Το πλήθος των μεταθέσεων ενός συνόλου n στοιχείων ισούται με $n!$.

Εδώ θα πρέπει να σημειώσουμε ότι ο χώρος στον οποίο συμβαίνει η αναζήτηση των πιθανών λύσεων καλείται χώρος γονοτύπου και οι πιθανές λύσεις αναπαρίστανται ως

σημεία μέσα στο χώρο αυτό. Επίσης κάθε μια θέση στο γονότυπο καταλαμβάνεται από μια μεταβλητή ή αλλιώς ένα γονίδιο.

3.2.2 Αντικειμενική Συνάρτηση (Fitness Function)

Η αντικειμενική συνάρτησης είναι μια συνάρτηση η οποία αναθέτει σε κάθε άτομο του πληθυσμού μια τιμή. Με βάση αυτή την τιμή γίνεται η σύγκριση μεταξύ των μελών του πληθυσμού, είτε κατά το στάδιο της επιλογής γονιών, είτε κατά το στάδιο της αντικατάστασης. Για παράδειγμα, σε ένα πρόβλημα εύρεσης του συντομότερου μονοπατιού μεταξύ δύο σημείων, η αντικειμενική συνάρτηση υπολογίζει τις αποστάσεις μεταξύ των ενδιάμεσων σημείων της διαδρομής και αθροίζοντάς τες προκύπτει το συνολικό μήκος της διαδρομής.

3.2.3 Πληθυσμός

Ο πληθυσμός αποτελεί το σύνολο των δυνατών λύσεων και ενδέχεται μέσα σε αυτό να υπάρχουν πολλά αντίγραφα ενός ατόμου. Συνήθως στις περισσότερες εφαρμογές γενετικών αλγορίθμων ο πληθυσμός διατηρείται σταθερός. Ενώ οι γενετικοί τελεστές (διασταύρωση, μετάλλαξη) δρουν σε επίπεδο ατόμου, οι μηχανισμοί επιλογής γονιών και αντικατάστασης δρουν σε επίπεδο πληθυσμού. Άρα αυτό που αλλάζει από γενιά σε γενιά είναι ο πληθυσμός. Όταν μιλάμε για το βαθμό διαφοροποίησης ενός πληθυσμού, εννοούμε τον αριθμό των διαφορετικών λύσεων οι οποίες είναι παρούσες σε κάθε γενιά.

3.2.4 Μηχανισμός Επιλογής Γονιών (Parent Selection)

Ο ρόλος αυτού του μηχανισμού είναι να ξεχωρίσει τα άτομα που διαθέτουν ποιοτικά χαρακτηριστικά με βάση την τιμή που προκύπτει από την αντικειμενική συνάρτηση επιτρέποντας τους να πάρουν μέρος στη δημιουργία της νέας γενιάς. Μαζί με το μηχανισμό αντικατάστασης, ο μηχανισμός επιλογής γονιών είναι υπεύθυνος ώστε να βελτιώνει τα ποιοτικά χαρακτηριστικά του πληθυσμού σε κάθε γενιά. Αν και τα άτομα με υψηλότερη τιμή αντικειμενικής συνάρτησης έχουν μεγαλύτερη πιθανότητα να επιλεγούν ως γονείς, παρόλα αυτά και στα άτομα με χαμηλή τιμή αντικειμενικής συνάρτησης δίνεται

μια μικρή πιθανότητα να πάρουν μέρος στη δημιουργία της νέας γενιάς ώστε να αποφεύγεται το φαινόμενο ο αλγόριθμος να εγκλωβίζεται σε τοπικά ελάχιστα.

3.2.5 Γενετικοί Τελεστές

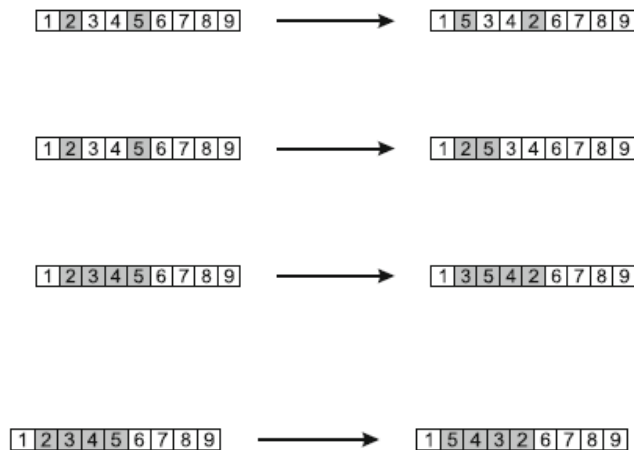
Ο ρόλος των γενετικών τελεστών είναι να δημιουργήσουν νέα άτομα δηλαδή νέες υποψήφιες λύσεις. Διακρίνονται στις μεταλλάξεις (mutation) και στις διασταυρώσεις (recombination-crossover).

Μετάλλαξη (mutation)

Μετά το στάδιο της διασταύρωσης εφαρμόζεται η μετάλλαξη η οποία προκαλεί μια μικρή, τυχαία αλλαγή στο γονότυπο, οπότε προκύπτει ένα ελαφρώς διαφορετικό παιδί. Η μετάλλαξη αντιστοιχεί σε ένα μικρό βήμα μέσα στο χώρο αναζήτησης των πιθανών λύσεων. Τα είδη των μεταλλάξεων που υπάρχουν εξαρτώνται από τον τύπο της αναπαράστασης που θα επιλέξουμε. Για παράδειγμα, αν επιλέξουμε ως αναπαράσταση των λύσεων τη μετάθεση τότε πιθανές μεταλλάξεις είναι:

- η *ανταλλαγή* (swap mutation): δύο θέσεις (γονίδια) στο χρωμόσωμα επιλέγονται τυχαία και ανταλλάσσουν θέσεις,
- η *εισαγωγή* (insertion mutation): δύο γονίδια επιλέγονται τυχαία και το δεύτερο μεταφέρεται δίπλα στο πρώτο,
- το *ανακάτεμα* (scramble mutation): ένα υποσύνολο του χρωμοσώματος τοποθετείται εκ νέου σε διαφορετικές θέσεις,
- η *αντιστροφή* (inversion mutation): επιλέγονται δύο σημεία του χρωμοσώματος και αντιστρέφεται η σειρά με την οποία εμφανίζονται οι τιμές ανάμεσα στα δύο σημεία.

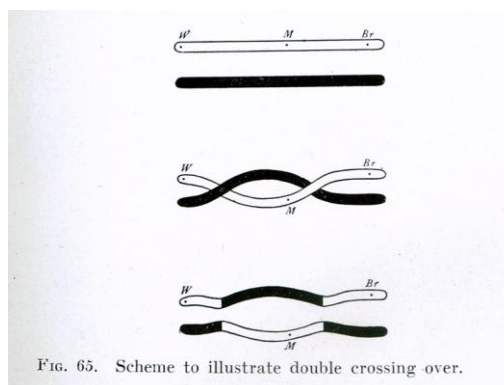
Όλες οι παραπάνω μεταλλάξεις παρουσιάζονται στην Εικόνα 33:



Εικόνα 33 Από πάνω προς τα κάτω: swap, insertion, scramble ,inversion mutations (πηγή: Introduction to Evolutionary Computing, 2015)

Διασταύρωση (recombination –crossover)

Σε βιολογικό επίπεδο ανασυνδυασμός είναι η ανταλλαγή γενετικού υλικού (γονιδίων) μεταξύ ομόλογων χρωμοσωμάτων. Κατά τον ανασυνδυασμό, τμήμα ενός πατρικού χρωμοσώματος αλλάζει θέση με το αντίστοιχο τμήμα του μητρικού χρωμοσώματος (Ανασυνδυασμός γονιδίων, 2020). Ο ανασυνδυασμός γονιδίων φαίνεται στην παρακάτω εικόνα:



Εικόνα 34 Ανασυνδυασμός γονιδίων (πηγή: Wikipedia)

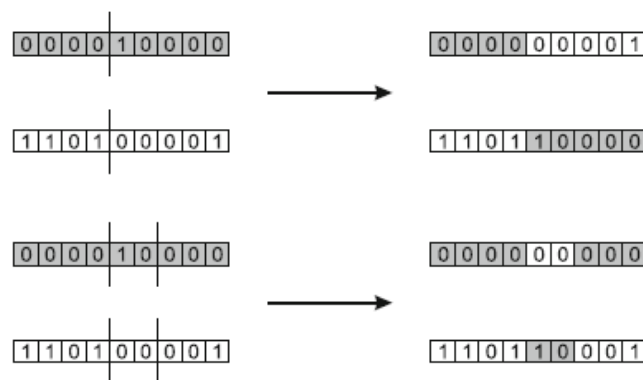
Αυτός ο γενετικός τελεστής επιτρέπει τη συνένωση μέρους της πληροφορίας από δύο ή και περισσότερους γονείς σε ένα ή δύο παιδιά. Όπως και στη μετάλλαξη, η επιλογή του ποια τμήματα από τον κάθε γονέα θα μεταφερθούν στο παιδί γίνεται με τυχαίο τρόπο. Η

βασική αρχή πίσω από τον ανασυνδυασμό γονιδίων είναι η εξής: μέσω του ζευγαρώματος δύο ατόμων του πληθυσμού, τα οποία έχουν διαφορετικά αλλά επιθυμητά χαρακτηριστικά, μπορούμε να δημιουργήσουμε έναν απόγονο που να συνδυάζει και τα δύο.

Τα βασικά είδη διασταύρωσης που χρησιμοποιούμε όταν έχουμε δυαδική αναπαράσταση είναι τα εξής:

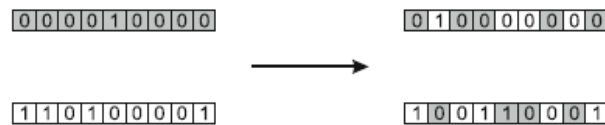
- διασταύρωση ενός σημείου (one-point crossover): επιλέγεται ένας τυχαίος αριθμός στο διάστημα $[1, \ell-1]$ όπου ℓ είναι το μήκος της δυαδικής αναπαράστασης, χωρίζουμε τους δύο γονείς σε αυτό το σημείο και δημιουργούμε δύο παιδιά ανταλλάσσοντας χιαστί τα τμήματα των γονιών.
- διασταύρωση πολλών σημείων (n-point crossover): το χρωμόσωμα διαιρείται σε περισσότερα από δύο τμήματα και τα παιδιά κληρονομούν με εναλλαγή ένα τμήμα από κάθε γονέα.
- ομοιόμορφη διασταύρωση (uniform crossover): παράγουμε ℓ τυχαίους αριθμούς στο διάστημα $[0,1]$, έναν για κάθε γονίδιο. Αν για κάθε θέση ο αριθμός αυτός είναι μικρότερος από μια παράμετρο p (συνήθως $p=0.5$) τότε το γονίδιο κληρονομείται από τον πρώτο γονέα αλλιώς από το δεύτερο.

Οι δύο πρώτες από τις παραπάνω περιπτώσεις παρουσιάζονται στην Εικόνα 35:



Εικόνα 35 Πάνω: one-point crossover, κάτω: n-point crossover με $n=2$ (πηγή: Introduction to Evolutionary Computing, 2015)

Στην Εικόνα 36 παρουσιάζεται η ομοιόμορφη διασταύρωση για έναν πίνακα τυχαίων αριθμών ίσο με $[0.3, 0.6, 0.1, 0.4, 0.8, 0.7, 0.3, 0.5, 0.3]$ και παράμετρο $p=0.5$:



Εικόνα 36 Uniform crossover (πηγή: Introduction to Evolutionary Computing, 2015)

3.2.6 Αντικατάσταση (Survivor Selection)

Όπως και με το μηχανισμό επιλογής γονιών έτσι και ο ρόλος της αντικατάστασης είναι να ξεχωρίσει τα άτομα του πληθυσμού βάσει της τιμής της αντικειμενικής συνάρτησης. Το στάδιο της αντικατάστασης όμως λαμβάνει χώρα μετά τη δημιουργία απογόνων. Αφού το μέγεθος του πληθυσμού διατηρείται σταθερό, χρειαζόμαστε έναν μηχανισμό που θα καθορίζει το ποια άτομα θα απαρτίζουν την επόμενη γενιά. Το κριτήριο για το ποια άτομα θα επιλεγούν είναι συνήθως η τιμή της αντικειμενικής συνάρτησης αλλά και η ηλικία. Μία μέθοδος επιλογής είναι να βαθμολογήσουμε το σύνολο γονέων και παιδιών και να επιλέξουμε αυτούς με την υψηλότερη βαθμολογία. Μια άλλη μέθοδος είναι η επιλογή να γίνει μόνο από τους απογόνους.

4. Σχεδίαση Γενετικού Αλγορίθμου #1

Στα πλαίσια της παρούσας διπλωματικής αναπτύξαμε δύο Γενετικούς Αλγορίθμους. Στον Αλγόριθμο #1, αφού υπολογίσουμε το γράφο ορατότητας ενός συνόλου σημείων τα οποία αποτελούν τις κορυφές πολυγωνικών εμποδίων στη συνέχεια χρησιμοποιούμε γενετικές διαδικασίες για να βρούμε το συντομότερο μονοπάτι. Η κατασκευή του γράφου γίνεται με την απλοϊκή μέθοδο (naïve method) σύμφωνα με την οποία για κάθε κόμβο εξετάζουμε αν αυτός είναι ορατός από όλους τους άλλους.

Στον Αλγόριθμο #2, αφού δημιουργήσουμε έναν συνδεδεμένο γράφο, κατασκευάζουμε έναν πληθυσμό από γράφους ορατότητας και με τη βοήθεια γενετικών διαδικασιών δημιουργούμε καινούριους γράφους προσθέτοντας έγκυρες ακμές δηλαδή ακμές που δεν τέμνουν τα εμπόδια⁹. Ο στόχος είναι να δημιουργήσουμε ένα γράφο με τον ελάχιστο αριθμό ακμών που θα μας επιτρέψει να βρούμε μια διαδρομή διάμεσου των εμποδίων η οποία θα προσεγγίζει τη βέλτιστη.

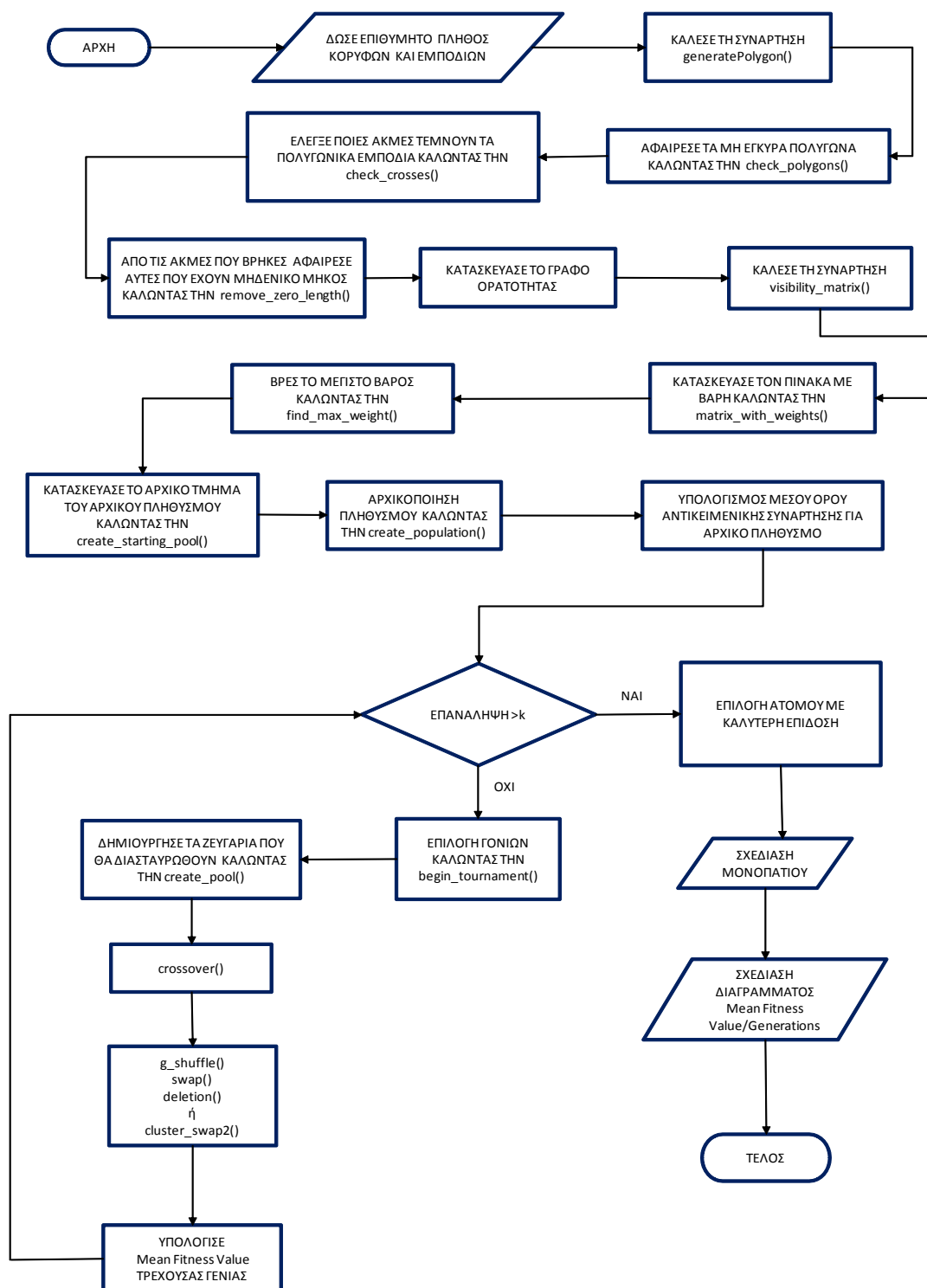
Στο κεφάλαιο αυτό εξετάζουμε τον Αλγόριθμο #1. Αρχικά υπολογίζουμε το γράφο ορατότητας και στη συνέχεια κατασκευάζουμε τον πίνακα με βάρη¹⁰ και τον χρησιμοποιούμε για να βρούμε το μέγιστο βάρος, δηλαδή το άθροισμα όλων των ακμών του γράφου. Τα άτομα του πληθυσμού του αλγορίθμου, δηλαδή τα μονοπάτια που ξεκινούν από τον κόμβο-πηγή προς τον κόμβο-προορισμό έχουν κωδικοποιηθεί ως λίστες με στοιχεία τις κορυφές των πολυγώνων. Οι λίστες μπορούν εύκολα να τροποποιηθούν, για παράδειγμα μπορούμε να διαγράψουμε ή να αφαιρέσουμε στοιχεία από αυτές. Η κάθε γενιά αποτελείται από τα άτομα του πληθυσμού, δηλαδή τις λίστες πάνω στις οποίες εφαρμόζονται οι γενετικοί τελεστές της επιλογής, της διασταύρωσης και της μετάλλαξης. Μετά από κάποιο αριθμό γενεών ο αλγόριθμος συγκλίνει σε λύση η οποία αποτελεί το συντομότερο μονοπάτι από τον κόμβο-πηγή προς τον κόμβο-προορισμό. Στις επόμενες ενότητες αυτού του κεφαλαίου παρουσιάζεται με λεπτομέρεια η σχεδίαση του Γενετικού Αλγορίθμου #1.

⁹ Υπενθυμίζουμε ότι υπάρχει μια ακμή μεταξύ των κορυφών v και w αν το v βλέπει το w ή αλλιώς αν το ευθύγραμμο τμήμα \overline{vw} δεν τέμνει το εσωτερικό κανενός εμποδίου που ανήκει στο σύνολο S .

¹⁰ Σε κάθε ακμή έχει ανατεθεί ένας αριθμός (βάρος) ο οποίος αντιπροσωπεύει το μήκος της.

4.1 Σχεδίαση Αλγορίθμου #1

Το διάγραμμα ροής του αλγορίθμου φαίνεται παρακάτω:



Εικόνα 37 Διάγραμμα ροής Γενετικού Αλγορίθμου #1

4.2 Μέρος Πρώτο – Κατασκευή Πολυγωνικών Εμποδίων

Ο αλγόριθμος δέχεται ως είσοδο τον αριθμό (v) των κορυφών κάθε εμποδίου (όλα τα εμπόδια έχουν τον ίδιο αριθμό κορυφών), το συνολικό αριθμό των εμποδίων (`num_of_obstacles`), το πλήθος των επαναλήψεων (`num_of_gen`) που θα κάνει ο αλγόριθμος (και κατά συνέπεια το πλήθος των γενεών που θα δημιουργήσει), τη συχνότητα μετάλλαξης (`mut_f`), και το πλήθος των τμημάτων από τα οποία θέλουμε να αποτελούνται οι λύσεις. Και οι πέντε παράμετροι ρυθμίζονται από το χρήστη. Ακολουθεί ο κώδικας παραμετροποίησης:

```
1    v = int(input("Enter number of vertices of each polygon ,  
between 3 and 5 please: "))  
  
2    num_of_obstacles = int(input("Enter number of obstacles ,  
between 2 and 21 please: "))  
  
3    num_of_gen = int(input("Enter number of Generations , above  
1 please: "))  
  
4    mut_f = float(input("Enter mutation frequency, less than 0.5  
please: "))  
  
5    num_of_seg = int(input("Enter number of path's segments,  
between 2 and 3 please: "))  
  
6    finish2 = round(time.perf_counter(),1)
```

Στη γραμμή 6, η μεταβλητή `finish2` χρησιμεύει για τη μέτρηση του χρόνου που χρειάζεται ο χρήστης για να εισάγει τα δεδομένα ούτως ώστε να μπορέσουμε να μετρήσουμε τον καθαρό χρόνο εκτέλεσης του Αλγορίθμου #1.

Στη συνέχεια καλείται η συνάρτηση `generatePolygon()` την οποία εντοπίσαμε στον διαδικτυακό τόπο:

<https://stackoverflow.com/questions/8997099/algorithm-to-generate-random-2d-polygon>

Ο κώδικας της `generatePolygon()` δίνεται στην επόμενη σελίδα:

```
1     def generatePolygon(ctrX, ctrY, aveRadius, irregularity,
2     spikeyness, numVerts):
3         def clip(x, min, max):
4             if (min > max):
5                 return x
6             elif (x < min):
7                 return min
8             elif (x > max):
9                 return max
10            else:
11                return x
12    irregularity = clip(irregularity, 0, 1) * 2 * math.pi /
numVerts
13    spikeyness = clip(spikeyness, 0, 1) * aveRadius
14
15    # generate n angle steps
16    angleSteps = []
17    lower = (2 * math.pi / numVerts) - irregularity
18    upper = (2 * math.pi / numVerts) + irregularity
19    sum = 0
20    for i in range(numVerts):
21        tmp = random.uniform(lower, upper)
22        angleSteps.append(tmp)
23        sum = sum + tmp
24
25    # normalize the steps so that point 0 and point n+1 are the
same
26    k = sum / (2 * math.pi)
27    for i in range(numVerts):
28        angleSteps[i] = angleSteps[i] / k
29
30    # now generate the points
31    points = []
32    angle = random.uniform(0, 2 * math.pi)
33    for i in range(numVerts):
34        r_i = clip(random.gauss(aveRadius, spikeyness), 0, 2 *
aveRadius)
35        x = ctrX + r_i * math.cos(angle)
36        y = ctrY + r_i * math.sin(angle)
37        points.append((int(x), int(y)))
38
39        angle = angle + angleSteps[i]
40
41    return points
```

Αυτό που κάνει η συνάρτηση `generatePolygon()` είναι να παίρνει ως όρισμα τις συντεταγμένες ενός σημείου το οποίο χρησιμοποιεί ως κέντρο ενός κύκλου. Επίσης παίρνει ως ορίσματα τις παραμέτρους `aveRadius`, `irregularity` και `spikeyness` με τις δύο τελευταίες να παίρνουν τιμές στο διάστημα $[0,1]$. Η πρώτη παράμετρος καθορίζει πόσο μεγάλο θα είναι το πολύγωνο, η δεύτερη καθορίζει την απόσταση μεταξύ κάθε κορυφής στην περιφέρεια του κύκλου και η τρίτη την απόσταση κάθε σημείου από το κέντρο του κύκλου. Στη συνέχεια επιλέγει τυχαία σημεία, το πλήθος των οποίων ισούται με τον επιθυμητό αριθμό κορυφών (n) κάθε πολυγώνου. Τα σημεία αυτά βρίσκονται στην περιφέρεια του κύκλου με ακτίνα `aveRadius` και τα οποία συνδέονται με ακμές. Στις γραμμές 2-10 υλοποιείται η συνάρτηση `clip(x,min,max)` ως εσωτερική συνάρτηση της `generatePolygon()`. Τελικά η συνάρτηση `generatePolygon()` επιστρέφει τις συντεταγμένες των κορυφών του πολυγώνου. Η `generatePolygon()` καλείται τόσες φορές όσες η παράμετρος `num_of_obstacles` που έχει εισάγει στην αρχή ο χρήστης. Σημειώνουμε ότι η συνάρτηση επιστρέφει κοίλα αλλά και κυρτά πολύγωνα.

Στο πρόγραμμά μας θέτουμε ως default τιμές στις παραμέτρους `aveRadius`, `irregularity` και `spikeyness` τις παρακάτω τιμές:

- `aveRadius = 8`,
- `irregularity = 0`,
- `spikeyness = 0.5`.

Μετά από δοκιμές παρατηρήσαμε ότι δίνοντας ως default τιμές για την τετμημένη και τεταγμένη του κέντρου κάθε κύκλου τις παρακάτω τιμές:

- `c_x = 30*(i % 3)`,
- `c_y = i*i + 40`,

όπου i ο αριθμός της μεταβλητής `num_of_obstacles` με $1 < i < \text{num_of_obstacles}$, τα πολυγωνικά εμπόδια κατανέμονται ομοιόμορφα στο επίπεδο. Σημειώνουμε ότι στον τύπο που δίνει την τετμημένη του κέντρου ο πολλαπλασιαστής με τιμή 30 υποδηλώνει την απόσταση ανάμεσα σε κάθε εμπόδιο και η διαίρεση με `mod3` υποδηλώνει ότι τα εμπόδια τοποθετούνται το ένα πάνω στο άλλο σε κάθε βήμα i με $i = 1, 2, 3$. Ακολουθεί ο βρόχος όπου καλείται η συνάρτηση `generatePolygon()`:

[illegible]

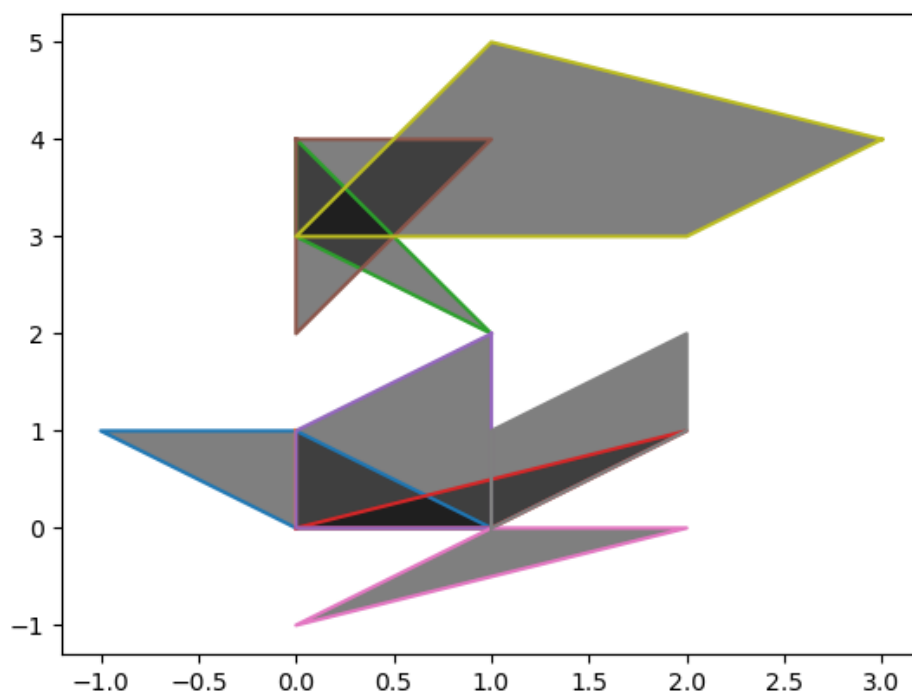
Στη συνέχεια τοποθετούμε τις συντεταγμένες των κορυφών κάθε πολυγώνου στον πίνακα `ob_3[]`. Έπειτα, με βάση τα σημεία αυτά κατασκευάζουμε αντικείμενα τύπου `Polygon` χρησιμοποιώντας τη βιβλιοθήκη `Shapely` της `Python` και τα τοποθετούμε σε έναν πίνακα (`polys[]`). Κατασκευάζουμε ακόμη ένα λεξικό (`pos10{}`) που έχει ως κλειδιά τον αύξοντα αριθμό του κάθε πολυγώνου και ως τιμές τις συντεταγμένες των κορυφών. Ακολουθεί ο αντίστοιχος κώδικας:

```
1     print(f"\n-----DICTIONARY POS10 : KEYS = NUMBERS *****  
VALUES = COORDS -----")  
  
2     pos10 = {}  
  
3     for i in range(len(ob_3)):  
4         pos10[i] = ob_3[i]  
5  
6     print(f" POS10: {pos10}")  
7  
8     polys = []  
9     polys2 = []  
10  
11    for i in range(len(ob_3)):  
12        s = Polygon(ob_3[i])  
13        print(f" S: {s}")  
14        polys.append(s)
```

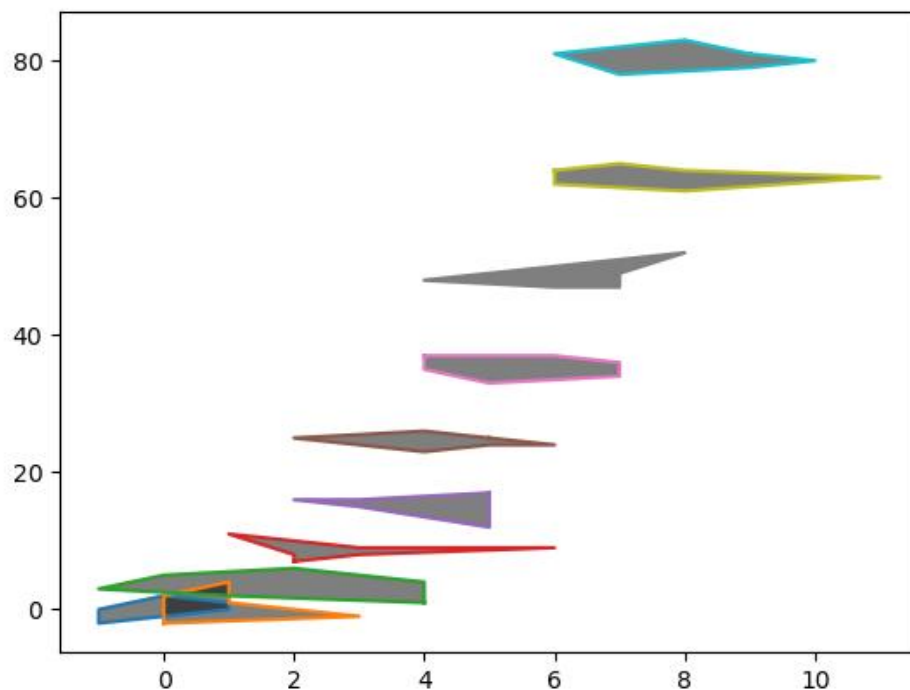
Τα πολύγωνα που επιστρέφει η `generatePolygon()` μπορεί να βρίσκονται σε τρεις πιθανές καταστάσεις:

- να μην τέμνουν το ένα το άλλο,
- να τέμνουν το ένα το άλλο,
- να έχουν εκφυλιστεί σε ευθύγραμμο τμήμα λόγω του ότι είτε όλες οι κορυφές έχουν ίδια τετμημένα (οριζόντιο ευθύγραμμο τμήμα) είτε ίδια τεταγμένα (κάθετο ευθύγραμμο τμήμα).

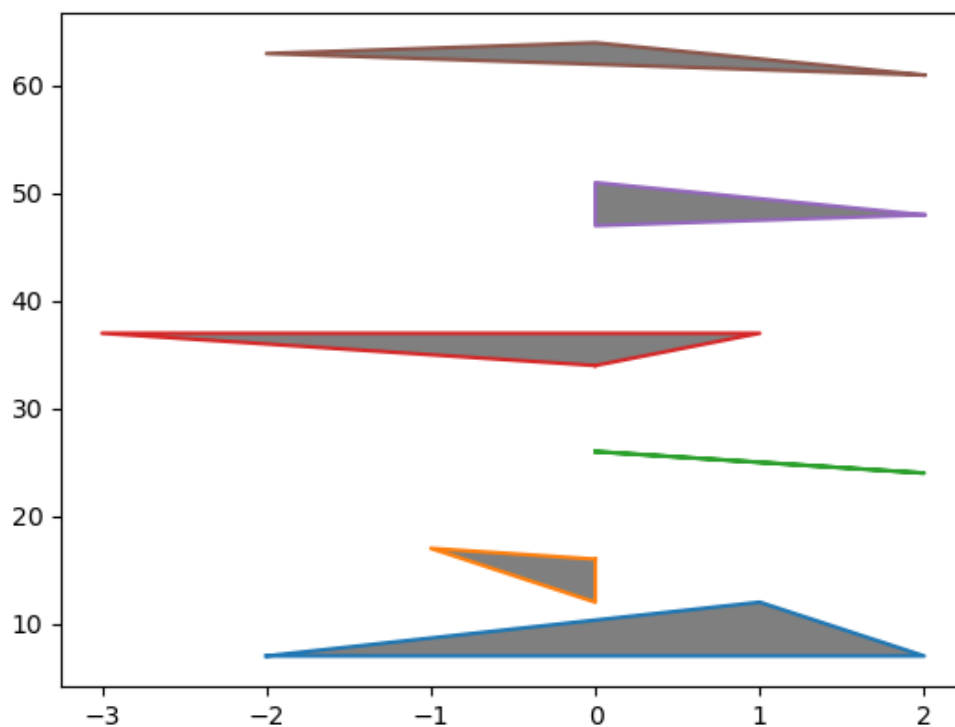
Στις παρακάτω εικόνες φαίνονται διάφορες πιθανές κατανομές των πολυγώνων στο επίπεδο:



Εικόνα 38 Μη έγκυρα πολύγωνα

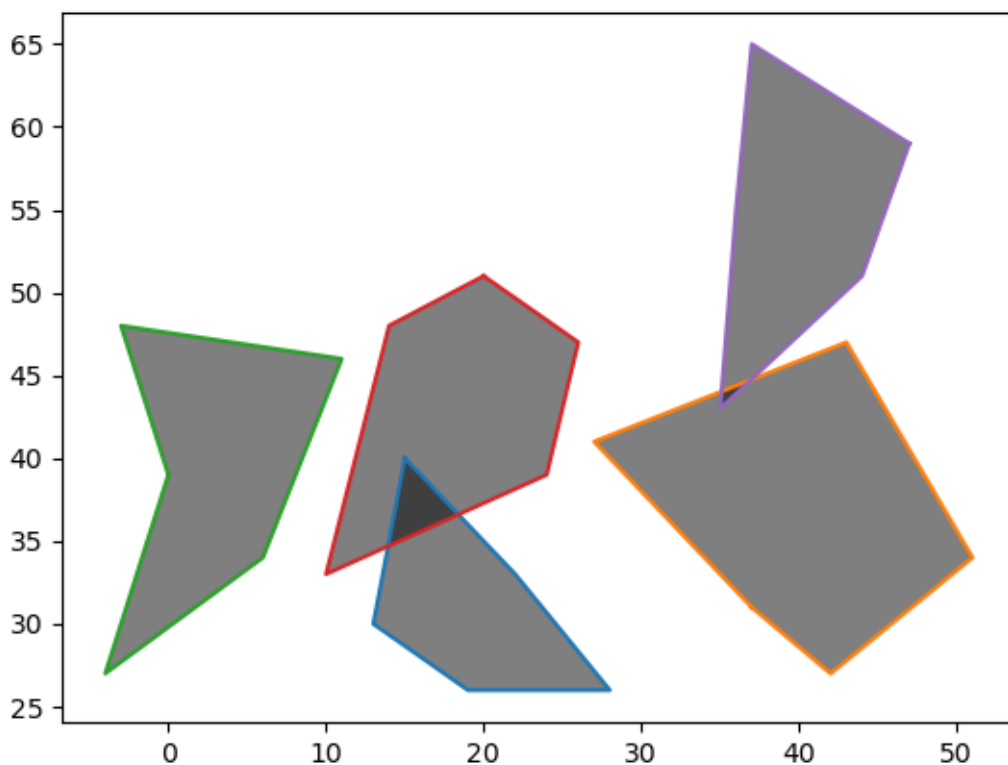


Εικόνα 39 Μη έγκυρα πολύγωνα



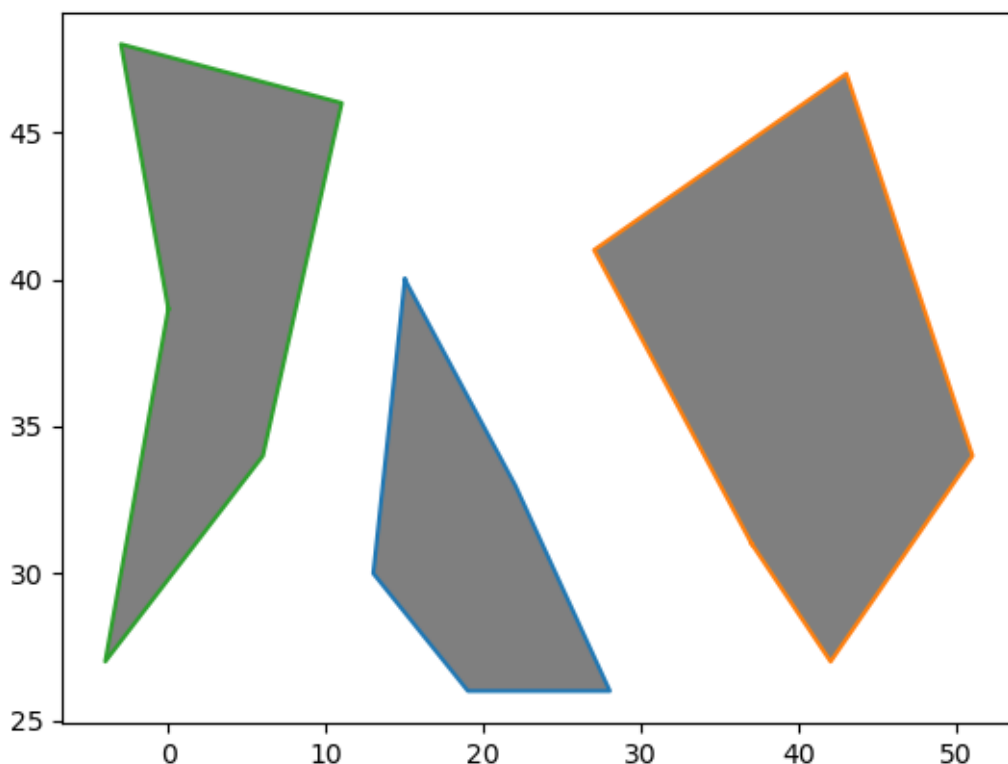
Εικόνα 40 Εκφυλισμός πολυγώνου σε ευθύγραμμο τμήμα

Για αυτό το λόγο κατασκευάσαμε τη συνάρτηση `check_polygons()` η οποία όταν εφαρμοστεί για μια κατανομή πολυγώνων όπως στην Εικόνα 41:



Εικόνα 41 Κατανομή πολυγώνων πριν τη χρήση της `check_polygons()`

επιστρέφει τα παρακάτω έγκυρα πολυγωνικά εμπόδια, όπως αυτά φαίνονται στην Εικόνα 42:



Εικόνα 42 Κατανομή πολυγώνων μετά τη χρήση της `check_polygons()`

Η συνάρτηση `check_polygons()` δέχεται ως ορίσματα τον πίνακα με τις συντεταγμένες των κορυφών των πολυγώνων (πίνακας `ob_3[]`) και τον πίνακα με τα αντικείμενα τύπου `Polygon` (πίνακας `polys[]`). Ο κώδικας της συνάρτησης ακολουθεί παρακάτω:

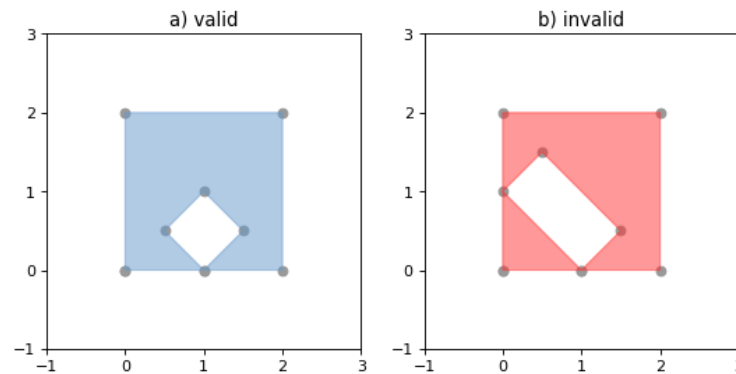
```

1 def check_polygons(array1, array2):
2
3     function_indexes = []
4     indexes2 = []
5
6     for i in range(len(array1)):
7         indexes2.append(i)
8     print(f" INITIAL INDEXES: {indexes2}")
9
10    polys3 = []
11    function_c_polygons2 = 0
12
13    for i in range(0, len(array2)):
14
15        if array2[i].is_valid: #
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>CHECK IF POLYGON IS DEGENERATED
INTO LINE !!!!!!!
16            print(f"\n-----POLYGON {i}
IS VALID!!!")
17
18            polys3.append(array2[i]) #
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> NOW POLYS3 HAS ONLY VALID
POLYGONS
19                function_c_polygons2 += 1
20            else:
21                continue
22
23    print(f"\nTHERE ARE {function_c_polygons2} VALID
POLYGONS.")
24    print("\nNOW LETS CHECK FOR INTERSECTIONS...")
25
26    my_m = []
27    c_polygons3 = 0
28
29    polys4 = []
30    for i in range(0, len(polys3) - 1):
31        for j in range(i + 1, len(polys3)):
32            if polys3[i].intersects(polys3[j]):
33                my_m.append(1)
34                c_polygons3 += 1
35            else:
36                my_m.append(0)
37                print(f"{i, j}")
38                polys4.append((i, j)) #
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> POLYS4 HAS THE TUPLES WE NEED !!
39
40    print(f"\n MY_M:{my_m} \nTHERE ARE :{c_polygons3}
INTERSECTIONS BETWEEN VALID POLYGONS !!!")
41
42    polys5 = []
43    for i in range(0, len(polys3) - 1):
44        for j in range(i + 1, len(polys3)):
45            print(f"{i, j}")

```

Διπλωματική Εργασία

Στη γραμμή 15 χρησιμοποιώντας την ενσωματωμένη συνάρτηση `is_valid()` της βιβλιοθήκης `Shapely` ξεχωρίζουμε τα έγκυρα από τα μη έγκυρα πολύγωνα. Ένα παράδειγμα έγκυρου και μη έγκυρου πολυγώνου φαίνεται στη παρακάτω εικόνα:



Εικόνα 43 Έγκυρα και μη έγκυρα πολύγωνα (πηγή:

<https://shapely.readthedocs.io/en/latest/manual.html?highlight=Polygon#shapely.geometry.polygon.Polygon>)

Αποθηκεύουμε όσα πολύγωνα ικανοποιούν τον έλεγχο εγκυρότητας στον πίνακα `polys3[]`, και στη συνέχεια ελέγχουμε ποια από αυτά τέμνουν ανά δύο το ένα το άλλο χρησιμοποιώντας ένα διπλό βρόχο και την ενσωματωμένη συνάρτηση `intersects()` της βιβλιοθήκης `Shapely`, όπως βλέπουμε στη γραμμή 32. Αποθηκεύουμε σε άλλο πίνακα όσα από τα πολύγωνα περάσουν και τον δεύτερο έλεγχο και εκτυπώνουμε πόσα από τα αρχικά πολύγωνα ήταν έγκυρα και επιπλέον δεν τέμνονται με άλλα. Τελικά η `check_polygons()` επιστρέφει έναν πίνακα με τα έγκυρα πολύγωνα (`polys2[]`), τους δείκτες (`indexes`) που αντιστοιχούν στον αύξοντα αριθμό καθενός από τα έγκυρα πολύγωνα αλλά και το πλήθος των έγκυρων πολυγώνων (`c_polygons`).

Έπειτα γνωρίζοντας τη μεταβλητή (`c_polygons`) αλλά και το πλήθος των κορυφών (v) υπολογίζουμε τον αύξοντα αριθμό του κόμβου-πηγή (`my_source2`) και του κόμβου-προορισμού (`my_target2`) ως εξής:

- `my_source2 = c_polygons*v,`
- `my_target2 = my_source2 + 1.`

Αυτός θα είναι και ο αναγνωριστικό αριθμός των δύο κόμβων στο γράφο που θα κατασκευάσουμε παρακάτω.

Στη συνέχεια χρησιμοποιώντας το λεξικό `pos10{}` κατασκευάζουμε τον πίνακα `polys2-coords2[]`, ο οποίος περιέχει τις συντεταγμένες των κορυφών των τελικών πολυγώνων. Ο αντίστοιχος κώδικας φαίνεται παρακάτω:

```
1 polys2_coords = []
2 for i in range(len(indexes)):
3     for key, value in pos10.items():
4         if key == indexes[i]:
5             pos10.get(key)
6             print(f"KEY:      {key}      -      VALUE:
{pos10.get(key)}")
7             polys2_coords.append(pos10.get(key))
8
9 for i in range(len(polys2_coords)):
10     print(f" {i}: COORDS: {polys2_coords[i]}")
```

Έπειτα δίνουμε τυχαίες τιμές στις συντεταγμένες του κόμβου-πηγή και του κόμβου-προορισμού χρησιμοποιώντας τη συνάρτηση `random.randint()` για οποιοδήποτε επιθυμητό διάστημα. Μετά από δοκιμές, επιλέξαμε για τον κόμβο-πηγή τα διαστήματα $[-20,-10]$ και $[0,40]$ για την τετμημένη και τεταγμένη αντίστοιχα και για τον κόμβο-προορισμό τα διαστήματα $[60,80]$ και $[170,190]$. Ακολουθεί ο αντίστοιχος κώδικας:

```
1     while True:
2         coords10 = []
3         s_coords1 = random.randint(-20,-10)
4         coords10.append(s_coords1)
5
6         s_coords2 = random.randint(0,40)
7         coords10.append(s_coords2)
8
9         t_coords1 = random.randint(60,80)
10        coords10.append(t_coords1)
11
12        t_coords2 = random.randint(170,190)
13        coords10.append(t_coords2)
14
15        coords11 = iter(coords10)
16        print(f"\nCOORDS11: {coords11}")
17        coords12 = list(zip(coords11, coords11))
18        print(f"\nCOORDS12: {coords12}")
19        coords13, coords14 = [], []
20        coords13.append(coords12[0])
21        coords14.append(coords12[1])
22        print(f"\nCOORDS13: {coords13} - COORDS14: {coords14}")
23
24        s8 = Point(coords12[0]) # SOURCE
25        s9 = Point(coords12[1]) # TARGET
26
27        for j in range(len(polys2)):
28
29            if not (polys2[j].contains(s8) and
30                    polys2[j].contains(s9)):
31                print(f"\nSOURCE'S COORDS: {coords13}")
32                print(f"\nTARGET'S COORDS: {coords14}")
33                polys2_coords.append(coords13)
34                polys2_coords.append(coords14)
35                break
```

Στις γραμμές 24 και 25, με τις τυχαίες τιμές που μας επέστρεψε η συνάρτηση `random.randint()` κατασκευάζουμε δύο αντικείμενα τύπου `Point`. Στη γραμμή 29 χρησιμοποιώντας τη συνάρτηση `contains()` της βιβλιοθήκης `Shapely` ελέγχουμε για κάθε πολύγωνο το οποίο βρίσκεται μέσα στον πίνακα `polys2[]` με τα έγκυρα πολύγωνα, αν κάποιο από αυτό περιέχει τα αντικείμενα τύπου `Point` που αντιστοιχούν στον κόμβο-πηγή

και κόμβο-προορισμό. Αν περάσουν τον έλεγχο, αποθηκεύουμε τις συντεταγμένες των δύο κόμβων στον πίνακα `polys2_coords[]` και έτσι έχουμε συγκεντρώσει τις συντεταγμένες όλων των κόμβων του γράφου μέσα σε ένα πίνακα. Αν δεν περάσουν τον έλεγχο επαναλαμβάνουμε τη διαδικασία χρησιμοποιώντας έναν ατέρμονα βρόχο.

Στη συνέχεια μετατρέπουμε τον πίνακα με τις συντεταγμένες των κόμβων σε μια λίστα από πλειάδες (list of tuples), έτσι ώστε να χειριζόμαστε πιο αποδοτικά τα δεδομένα μας. Ονομάζουμε τη λίστα αυτή `ob_1`. Η κατασκευή της λίστας φαίνεται παρακάτω:

```
1  ob_1 = []
2  for i in range((len(polys2_coords))):
3      for j in range(len(polys2_coords[i])):
4          ob_1.append(polys2_coords[i][j])
5
6  print(f"\nFINAL POINTS: {ob_1}")
```

Όπως παρατηρούμε στο διάγραμμα ροής, στο επόμενο βήμα του αλγορίθμου καλούμε τη συνάρτηση `check_crosses()` η οποία δέχεται ως όρισμα τη λίστα `ob_1` και ένα αντικείμενο τύπου `Polygon`. Με αυτή τη συνάρτηση επιδιώκουμε να ελέγξουμε ποιες ακμές οι οποίες σχηματίζονται από κάθε ζεύγος σημείων που βρίσκεται μέσα στη λίστα `ob_1` δεν τέμνουν καθένα από τα πολύγωνα που βρίσκονται μέσα στον `polys2[]`. Με άλλα λόγια, αναζητούμε τα ευθύγραμμα τμήματα \overline{vw} μεταξύ των κορυφών v και w τα οποία δεν τέμνουν το εσωτερικό κανενός από τα εμπόδια. Για αυτό το λόγο, καλούμε τη συνάρτηση `check_crosses()` τόσες φορές όσες είναι το πλήθος των πολυγωνικών εμποδίων. Σημειώνουμε ότι το πλήθος των ακμών ενός συνόλου n σημείων ισούται με τους συνδυασμούς των n ανά k :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \xrightarrow{k=2} \frac{n!}{2!(n-2)!}$$

Ακολουθεί ο κώδικας της συνάρτησης `check_crosses()`:

```
1 def check_crosses(points, polygon):
2     lines = []
3     e_list = list()
4     solutions = []
5     length_sol = []
6     c5000 = 0
7     ct = 0
8     ct_ar = []
9     cf_ar = []
10    cf = 0
11    valid = False
12    valid_array = []
13
14    for i in points:
15        for j in points:
16            line = LineString([i, j])
17            if line.crosses(polygon) or polygon.contains(line):
18                valid = True
19                print(f"\nLINE: {i, j} ....VALID:{valid}")
20                valid_array.append(valid)
21                ct += 1
22                ct_ar.append(ct)
23                print(f"ct:{ct}")
24            else:
25                valid = False
26                print(f"\nLINE:{i, j} ...VALID:{valid}")
27                valid_array.append(valid)
28                cf += 1
29                print(f"cf:{cf}")
30                cf_ar.append(cf)
31
32            c5000 += 1
33            lines.append(i)
34            lines.append(j)
35            e_list.append(line)
36            length_sol.append(line.length)
37            solutions.append(list(line.coords))
38
39    return valid_array, ct_ar, cf_ar, lines, e_list, solutions,
length_sol, c5000
```


Στη γραμμή 16 η συνάρτηση χρησιμοποιεί ένα αντικείμενο τύπου `LineString` για να κατασκευάσει μια ακμή, το οποίο αρχικοποιείται κάθε φορά με δυο πλειάδες από τη λίστα `ob_1`. Στη γραμμή 17 χρησιμοποιεί τις ενσωματωμένες συναρτήσεις `crosses()` και `contains()` της βιβλιοθήκης `Shapely` και ελέγχει αν η ακμή τέμνει κάποιο από τα πολυγωνικά εμπόδια ή αν κάποιο εμπόδιο περιέχει ολόκληρη ακμή ή κάποιο τμήμα αυτής. Τελικά η συνάρτηση επιστρέφει έναν πίνακα (`valid_array[]`) με boolean τιμές (`True` αν τέμνει, `False` αν δεν τέμνει), έναν πίνακα `e_list[]` που περιέχει αντικείμενα τύπου `LineString` που αναπαριστούν όλες τις έγκυρες ακμές, έναν πίνακα `solutions[]` που περιέχει λίστες με πλειάδες, όπου οι πλειάδες περιέχουν τις συντεταγμένες των κορυφών των ακμών, έναν πίνακα `length_sol[]` με τα μήκη των παραπάνω ακμών και έναν μετρητή που αντιστοιχεί στο πλήθος των έγκυρων ακμών.

Στη συνέχεια τοποθετούμε τον πίνακα `valid_array[]` από κάθε πολύγωνο μέσα σε νέο πίνακα με όνομα `valid_total[]`. Μετατρέπουμε τον πίνακα `valid_total[]` σε `numpy array` για καλύτερο χειρισμό των δεδομένων, χρησιμοποιώντας τη βιβλιοθήκη `numpy`. Μετά κατασκευάζουμε τον πίνακα `non_zero[]` χρησιμοποιώντας την ενσωματωμένη συνάρτηση `np.sum()` της βιβλιοθήκης `numpy`. Εξετάζουμε όλα τα στοιχεία του πίνακα `non_zero[]` και αν κάποια από αυτά είναι ίσο με μηδέν τοποθετούμε σε νέο πίνακα `valid_edges[]` τον αντίστοιχο δείκτη *i*. Επίσης με τον μετρητή `c6000` μετράμε το πλήθος των τελικών έγκυρων ακμών. Παρακάτω δίνουμε τον κώδικα καθώς και ένα παράδειγμα εκτέλεσης για δύο τριγωνικά εμπόδια:

```
1   for i in range(len(valid_total)):
2       print(f"{i}: {valid_total[i]}")
3
4   n_valid_total = np.array(valid_total)
5
6   c6000 = 0
7   valid_edges = []
8   non_zero = np.sum(n_valid_total, axis=0)
9   print(f"NON: {non_zero}")
10  for i in range(len(non_zero)):
11      if non_zero[i] == 0:
12          valid_edges.append(i)
13          c6000 += 1
14      else:
15          continue
```

0: [True, False, False, False, False, False, False, False, False, False, False, False, False, False, False, True, True, True, False, False, True, False, False, False, False, False, True, False, False, False]

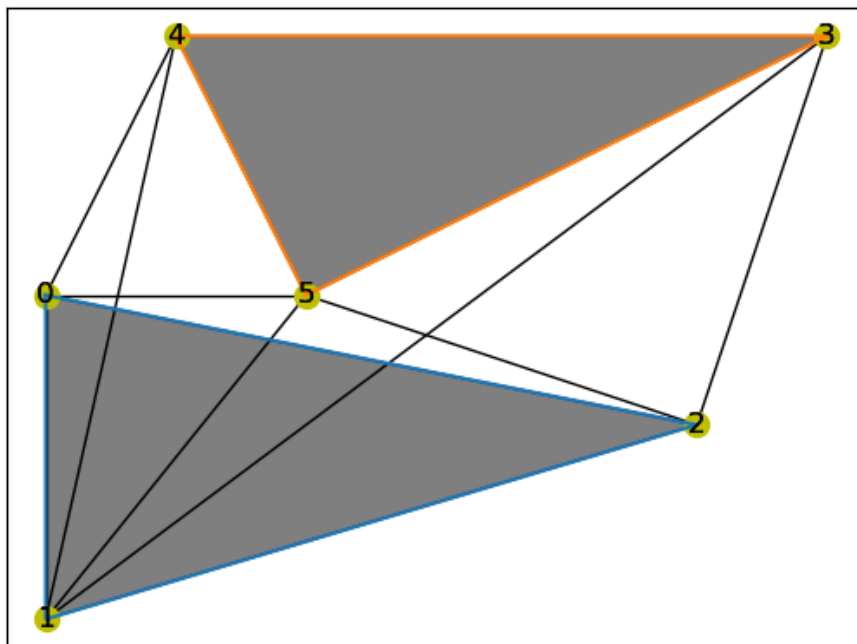
1: [False, False, False, False, True, False, False, False, False, False, False, False, False, False, False, False, False, False, True, False, False, True, False, False, False, False, False, False, False, False]

NON: [1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 1 0 1 0 0 0
0 0 1 0 0 0]

THERE ARE: **26** FINAL VALID EDGES OUT OF: **36**

FINAL VALID EDGES: [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19, 22, 23, 25, 27, 28, 29, 30, 31, 33, 34, 35]

Πιο συγκεκριμένα παρατηρήσαμε ότι ο έλεγχος που διενεργείται από τη συνάρτηση `check_crosses()` δεν είναι αρκετός για να μας επιστρέψει τις ζητούμενες ακμές (ευθύγραμμα τμήματα \overline{vw}). Για παράδειγμα, όταν το αποτέλεσμα της κλήσης της συνάρτησης `check_crosses()` για δύο τριγωνικά εμπόδια χρησιμοποιηθεί για να σχεδιάσουμε τις ζητούμενες ακμές, παίρνουμε το παρακάτω αποτέλεσμα:



Εικόνα 44 Μη έγκυρες ακμές (διέρχονται διαμέσου των εμποδίων)

Η εξήγηση δίνεται παρακάτω:

Έστω ότι έχουμε τα σημεία $points=[0,1,2,3,4,5]$ και τα πολύγωνα $polygons=[0,1]$.

Κατασκευάζουμε τους δύο πίνακες:

point	Polygon	line	query	result	append(line)?
point[0]=0	p[0]				
		0-1	line crosses(p[0])?	NO	YES
		0-2	line crosses(p[0])?	NO	YES
	p[1]	0-3	line crosses(p[1])?	YES	NO
		0-4	line crosses(p[1])?	NO	YES
		0-5	line crosses(p[1])?	NO	YES

Πίνακας 1

point	Polygon	line	query	result	append (line)?
point[1]=1	p[0]				
		1-0	line crosses(p[0])?	NO	YES
		1-2	line crosses(p[0])?	NO	YES
	p[1]	1-3	line crosses(p[1])?	NO	YES
		1-4	line crosses(p[1])?	NO	YES
		1-5	line crosses(p[1])?	NO	YES

Πίνακας 2

Όπως βλέπουμε και στην Εικόνα 44 ενώ πράγματι στην ερώτηση αν π.χ., η ακμή 1-3 τέμνει το πολύγωνο 1 με κορυφές [3,4,5] η απάντηση είναι ‘NO’, η συνάρτηση λανθασμένα δεν εξετάζει αν η τρέχουσα ακμή τέμνει και το πολύγωνο από το οποίο προέρχονται οι ακμές αυτές. Αντίθετα, στην ερώτηση αν η ακμή 0-3 τέμνει το πολύγωνο 1 η απάντηση είναι ‘YES’, οπότε σωστά δεν σχεδιάζει την ακμή αυτή.

Με τη χρήση λοιπόν του πίνακα non_zero[] καταφέρνουμε και εντοπίζουμε τις ζητούμενες ακμές, και αποθηκεύουμε τις συντεταγμένες των κορυφών των ακμών στο νέο πίνακα f_solutions[]. Όμως δεν έχουμε τελειώσει ακόμα. Ο πίνακας f_solutions[] περιέχει και ακμές με μηδενικά μήκη λόγω του ότι μερικές από τις ακμές (τόσες όσες είναι οι συνολικοί κόμβοι) έχουν σαν κορυφές το ίδιο σημείο. Λαμβάνουμε λοιπόν μέτρα ώστε να τις αφαιρέσουμε.

Για αυτό το λόγο κατασκευάζουμε ένα πολύ χρήσιμο λεξικό με όνομα pos1{} το οποίο περιέχει ως κλειδιά (keys) τους αύξοντες αριθμούς (ξεκινώντας από το μηδέν) όλων των σημείων που βρίσκονται μέσα στη λίστα ob_1 και ως τιμές (values) τις συντεταγμένες κάθε σημείου. Η κατασκευή του λεξικού pos1{} δίνεται παρακάτω:

```

1    print("-----NODE  DICTIONARY-----")
2
3    pos1 = {}
4
5    for i in range(len(ob_1)):

```

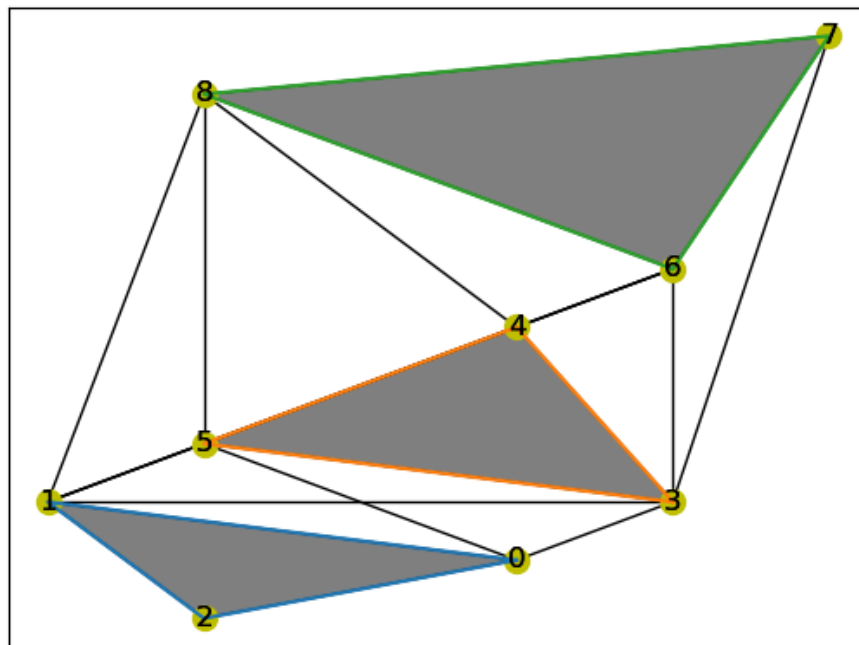
```
4         pos1[i] = ob_1[i]
5     print(pos1)
```

Παρακάτω υλοποιούμε τη συνάρτηση `remove_zero_length()` η οποία δέχεται ως όρισμα τον πίνακα με τις συντεταγμένες των κορυφών των ακμών (`f_solutions[]`) και το λεξικό `pos1{}`. Αυτό που κάνει η συνάρτηση `remove_zero_length()` είναι να κατασκευάζει με τη βοήθεια του λεξικού `pos1{}` έναν πίνακα (βλέπε γραμμή 18, πίνακας `lines2[]`), ο οποίος περιέχει τα κλειδιά-αύξοντες αριθμούς των δύο κορυφών από κάθε ακμή π.χ., `[0 1 0 2 0 3 0 4 1 0 1 1 1 2 1 3...]`. Στη συνέχεια μετατρέπει τα κλειδιά σε πλειάδες και όπου σε κάθε πλειάδα δεν υπάρχει το ίδιο κλειδί, αποθηκεύει την πλειάδα σε καινούριο πίνακα (βλέπε γραμμή 30, πίνακας `lines300[]`). Το αποτέλεσμα για τον παραπάνω πίνακα θα είναι `[0 1 0 2 0 3 0 4 1 2 1 3...]`, όπου παρατηρούμε ότι η άκυρη ακμή 1-1 έχει αφαιρεθεί. Τελικά η συνάρτηση επιστρέφει τον πίνακα `lines_500[]` με τα ζητούμενα κλειδιά καθώς και έναν μετρητή `c_200` που δείχνει το τελικό πλήθος των έγκυρων ακμών. Ακολουθεί ο κώδικας της `remove_zero_length()`:

```
1     def remove_zero_length(array1, pos):
2
3         lines2000 = []
4         for i in range((len(array1))):
5             for j in range(len(array1[i])):
6                 lines2000.append(array1[i][j]) # BE CAREFUL
7         WHERE TO APPEND f_solutions !!!!
8
9         print(f" LINES2000: {lines2000}")
10        n_lines2000 = np.array(lines2000)
11        print(f" LINES_SHAPE: {n_lines2000.shape}")
12
13        print("\n-----LINES_2   (KEYS=NODES) -----
14        -----")
15        lines2 = []
16
17        for i in lines2000:
18            for key, val in pos.items():
19                if i == val:
20                    lines2.append(key)
21
22        print(f"LINES2: {lines2}")
23
24        n_lines2 = np.array(lines2)
25        print(f"LINES_2 SHAPE: {n_lines2.shape} ")
26
27        print("\n-----REMOVE ZERO LENGTH EDGES -----
28        -----")
```

```
27         print("\n-----FIRST MAKE A LIST OF TUPLES... -----  
-----")  
28  
29         it = iter(lines2)  
30         lines300 = list(zip(it, it))  
31         print(f" LINES300: {lines300} ")  
32  
33         print("\n----- ...THEN REMOVE ZERO LENGTH EDGES ---  
-----")  
34  
35         lines400 = []  
36         for i in range(len(lines300)):  
37             if lines300[i][0] != lines300[i][1]:  
38                 lines400.append(lines300[i])  
39  
40         print(" ")  
41         print(f" * " * 20 + "SUCCESS " + " * " * 20)  
42         print(" ")  
43         print(f"OLD LINES300: {lines300} ")  
44         print(f"NEW LINES400: {lines400} ")  
45  
46         n_lines400 = np.array(lines400)  
47         print(n_lines400.shape)  
48         f_c_200 = n_lines400.shape[0]  
49         print(" ")  
50  
51         print("----- CONVERT LINES400 SO TO CONTAINS ONLY  
KEYS=NODES (EQUAL TO Lines2 ARRAY) ----")  
52  
53         f_lines500 = []  
54  
55         for t in lines400:  
56             for x in t:  
57                 f_lines500.append(x)  
58  
59         print(f" LINES500: {f_lines500}")  
60  
61         return f_lines500, f_c_200
```

Αν τώρα με το αποτέλεσμα που έχει επιστρέψει η συνάρτηση σχεδιάσουμε τις ακμές για τρία τριγωνικά εμπόδια θα πάρουμε το παρακάτω σχήμα:



Εικόνα 45 Έγκυρες ακμές (δεν διέρχονται πλέον διαμέσου των εμποδίων)

4.3 Μέρος Δεύτερο – Κατασκευή Γράφου Ορατότητας

Στη συνέχεια, έχοντας στη διάθεσή μας τον πίνακα με τα κλειδιά-αύξοντες αριθμούς ακμών (πίνακας `arr_500[]`) καθώς και το πλήθος των έγκυρων ακμών (μετρητής `c_200`), είναι εύκολο να κατασκευάσουμε το γράφο ορατότητας. Πρώτα δημιουργούμε έναν άδειο γράφο με την εντολή:

➤ `G = nx.Graph()`

και στη συνέχεια χρησιμοποιούμε την ενσωματωμένη συνάρτηση `Graph.add_edge()` της βιβλιοθήκης `networkx` για να προσθέσουμε ακμές.

Χρησιμοποιούμε επίσης τις ενσωματωμένες συναρτήσεις:

- `nx.generate_edgelist()` η οποία μας επιστρέφει όλες τις ακμές του γράφου,
- `nx.is_connected()` η οποία επιστρέφει `True` αν ο γράφος είναι συνδεδεμένος,
- `G.neighbors()` η οποία επιστρέφει τους γείτονες κάθε κόμβου του γράφου,
- `nx.connected_components()` η οποία επιστρέφει τους κόμβους του γράφου.

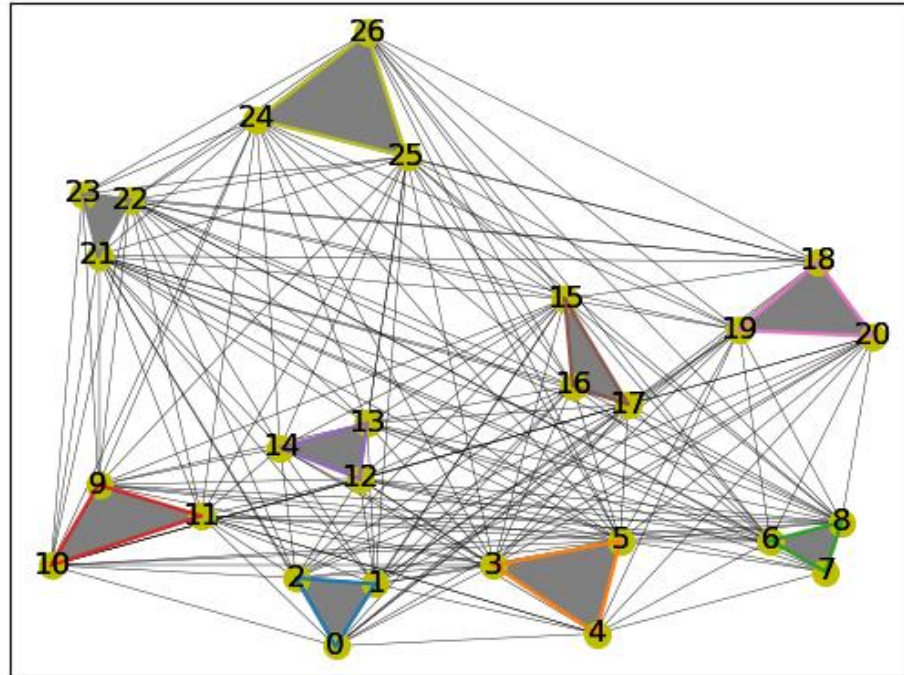
Αν καλέσουμε τις τρεις τελευταίες συναρτήσεις για το γράφο της Εικόνας 45 θα πάρουμε το παρακάτω αποτέλεσμα:

```
-----IS CONNECTED -----
True

----- NEIGHBOURS 1-----
0 has neighbours: [1, 2, 3, 5]
1 has neighbours: [0, 2, 3, 4, 5, 6, 8]
2 has neighbours: [0, 1]
3 has neighbours: [0, 1, 4, 5, 6, 7]
4 has neighbours: [1, 3, 5, 6, 8]
5 has neighbours: [0, 1, 3, 4, 6, 8]
6 has neighbours: [1, 3, 4, 5, 7, 8]
7 has neighbours: [3, 6, 8]

-----COMPONENTS-----
{0, 1, 2, 3, 4, 5, 6, 7}
```


Στην παρακάτω εικόνα κατασκευάζουμε έναν ακόμη γράφο με 27 κόμβους και καλούμε τις συναρτήσεις `is_connected()` και `G.neighbors()`:



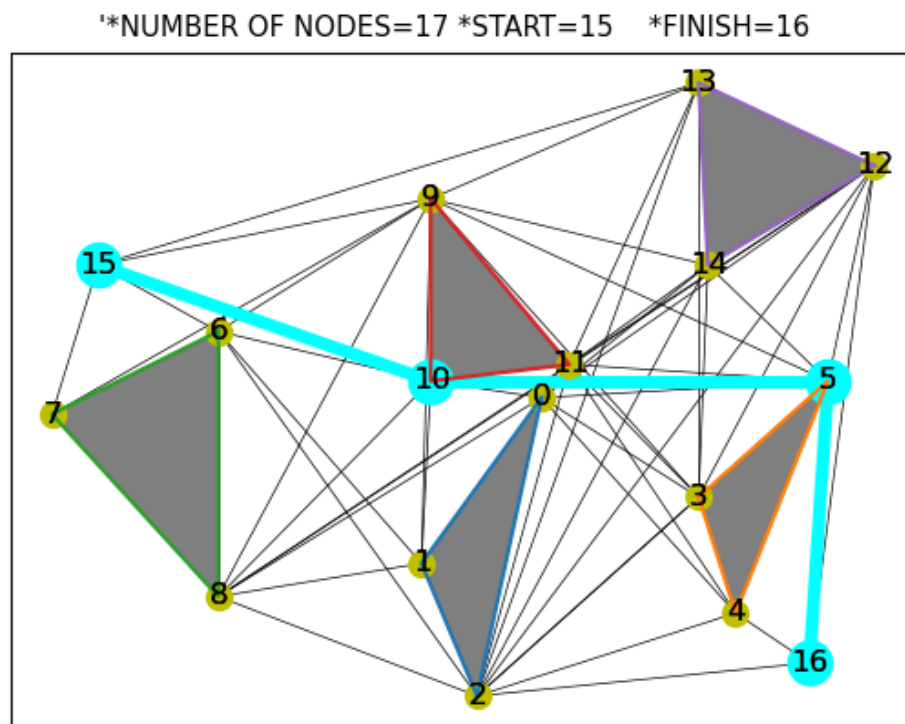
Εικόνα 46 Γράφος Ορατότητας με 27 κόμβους

```
-----IS CONNECTED -----
True
```

```
----- NEIGHBOURS 1-----
0 has neighbours: [1, 2, 3, 4, 10, 11, 15, 16, 17, 18, 19, 20, 21]
1 has neighbours: [0, 2, 3, 4, 5, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 25]
2 has neighbours: [0, 1, 3, 5, 10, 11, 12, 14, 15, 16, 17, 20, 21, 22]
3 has neighbours: [0, 1, 2, 4, 5, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 24, 25]
4 has neighbours: [0, 1, 3, 5, 6, 7, 9, 11, 12, 14, 19, 20]
5 has neighbours: [1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 16, 17, 19, 20, 21, 22, 24, 25, 26]
6 has neighbours: [4, 5, 7, 8, 9, 11, 12, 13, 15, 16, 17, 19, 20, 21, 22, 26]
7 has neighbours: [4, 5, 6, 8, 12, 13, 21, 22]
8 has neighbours: [5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 17, 19, 20, 21, 22, 25, 26]
```

```
9 has neighbours: [3, 4, 5, 6, 8, 10, 11, 12, 13, 14, 15, 21, 22, 23, 24, 25]
10 has neighbours: [0, 2, 3, 5, 8, 9, 11, 12, 17, 20, 21, 22, 23, 24]
11 has neighbours: [0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 21, 22, 24, 25]
12 has neighbours: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 25]
13 has neighbours: [1, 3, 5, 6, 7, 8, 9, 12, 14, 15, 16, 17, 21, 22, 24, 25]
14 has neighbours: [1, 2, 3, 4, 9, 11, 12, 13, 15, 21, 22, 24, 25]
15 has neighbours: [0, 1, 2, 3, 6, 8, 9, 12, 13, 14, 16, 17, 18, 19, 21, 22, 24, 25, 26]
16 has neighbours: [0, 1, 2, 3, 5, 6, 8, 12, 13, 15, 17, 21, 22, 24, 25, 26]
17 has neighbours: [0, 1, 2, 3, 5, 6, 8, 10, 12, 13, 15, 16, 18, 19, 20, 21]
18 has neighbours: [0, 15, 17, 19, 20, 21, 22, 23, 24, 25, 26]
19 has neighbours: [0, 1, 3, 4, 5, 6, 8, 15, 17, 18, 20, 22, 24, 25, 26]
20 has neighbours: [0, 1, 2, 3, 4, 5, 6, 8, 10, 17, 18, 19]
21 has neighbours: [0, 1, 2, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 22, 23, 24, 25]
22 has neighbours: [2, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 18, 19, 21, 23, 24, 25, 26]
23 has neighbours: [9, 10, 18, 21, 22, 24, 25, 26]
24 has neighbours: [3, 5, 9, 10, 11, 13, 14, 15, 16, 18, 19, 21, 22, 23, 25, 26]
25 has neighbours: [1, 3, 5, 8, 9, 11, 12, 13, 14, 15, 16, 18, 19, 21, 22, 23, 24, 26]
26 has neighbours: [5, 6, 8, 15, 16, 18, 19, 22, 23, 24, 25]
```

Αν τώρα βάλουμε και τους κόμβους αφετηρίας (start-15) και προορισμού (finish-16) και καλέσουμε τη συνάρτηση `nx.dijkstra_path()` της βιβλιοθήκης `networkx` και σχεδιάσουμε το αποτέλεσμα χρησιμοποιώντας τη συνάρτηση `plt.plot()` της βιβλιοθήκης `Matplotlib` θα πάρουμε την παρακάτω εικόνα:



Εικόνα 47 Εύρεση πιο σύντομου μονοπατιού για γράφο με 17 κόμβους χρησιμοποιώντας τη συνάρτηση `nx.dijkstra_path()` της βιβλιοθήκης `Networkx`

Αν και οι ενσωματωμένες συναρτήσεις της βιβλιοθήκης Networkx είναι χρήσιμες για να ελέγξουμε ότι ο γράφος που έχουμε κατασκευάσει είναι συνδεδεμένος, ποιοι είναι οι γείτονες κάθε κόμβου, αν υπάρχει πράγματι συντομότερο μονοπάτι από τον κόμβο-πηγή προς τον κόμβο-προορισμού, κλπ., στόχος του Γενετικού Αλγορίθμου #1 είναι η εύρεση του πιο σύντομου μονοπατιού χρησιμοποιώντας γενετικές διαδικασίες.

Συνεχίζοντας σύμφωνα με το διάγραμμα ροής του Γενετικού Αλγορίθμου #1 βλέπουμε ότι μετά την κατασκευή του γράφου ορατότητας (G), καλούμε τη συνάρτηση `visibility_matrix()` που δέχεται ως όρισμα το γράφο G . Η συνάρτηση κατασκευάζει τον πίνακα γειννίας του γράφου ορατότητας, ο οποίος είναι συμμετρικός ως προς τη κύρια διαγώνιο, ισχύει δηλαδή ότι:

$$a_{ij} = a_{ji}$$

και επιπλέον:

$$a_{ij} = 0 \text{ για } i = j$$

Παρακάτω δίνουμε τον κώδικα της συνάρτησης:

```
1 def visibility_matrix(G):
2     vis_array = []
3     for i in range(G.order()):
4         vis_array.append([])
5         for j in range(G.order()):
6             if j in list(G.neighbors(i)):
7                 vis_array[i].append(1)
8
9             else:
10                vis_array[i].append(0)
11
12    print(vis_array)
13    f_vis2_array = np.array(vis_array)
14
15    print("\n-----VISIBILITY MATRIX-----
16    print(f_vis2_array.reshape(G.order(), G.order()))
17
18    print("\n----- MY MATRIX -----
19
20    my_matrix = nx.to_numpy_matrix(G,
21    nodelist=sorted(G.nodes()))
22    print(f"MY_MATRIX: \n {my_matrix}")
23    return f_vis2_array
```

Όπως παρατηρούμε μπορούμε να κατασκευάσουμε τον πίνακα γειτνίασης με τρεις τρόπους. Με τον 1^ο τρόπο χρησιμοποιούμε την ενσωματωμένη συνάρτηση `G.order()` της βιβλιοθήκης `Networkx` η οποία επιστρέφει το πλήθος των κόμβων του γράφου και με ένα διπλό βρόχο κατασκευάζουμε το ζητούμενο πίνακα `vis_array[]`. Παρακάτω ακολουθεί το αποτέλεσμα που εκτυπώνει η συνάρτηση για έναν γράφο με 9 κόμβους:

```
-----VIS_ARRAY-----  
[[0, 1, 1, 1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 1, 0, 0, 1], [1, 1,  
0, 1, 1, 1, 0, 1, 1], [1, 1, 1, 0, 1, 1, 0, 1, 0], [1, 1, 1, 1, 0,  
0, 1, 1, 1], [0, 1, 1, 1, 0, 0, 1, 0, 1], [0, 0, 0, 0, 1, 1, 0, 1,  
1], [1, 0, 1, 1, 1, 0, 1, 0, 1], [0, 1, 1, 0, 1, 1, 1, 1, 0]]
```

Για καλύτερο οπτικό αποτέλεσμα μετατρέπουμε τον `vis_array[]` σε `numpy array` και στη γραμμή 16 χρησιμοποιώντας την ενσωματωμένη συνάρτηση `reshape()` της βιβλιοθήκης `Numpy` αλλάζουμε σχήμα στον πίνακα όπως φαίνεται παρακάτω:

```
-----VISIBILITY MATRIX-----  
[[0 1 1 1 1 0 0 1 0]  
 [1 0 1 1 1 1 0 0 1]  
 [1 1 0 1 1 1 0 1 1]  
 [1 1 1 0 1 1 0 1 0]  
 [1 1 1 1 0 0 1 1 1]  
 [0 1 1 1 0 0 1 0 1]  
 [0 0 0 0 1 1 0 1 1]  
 [1 0 1 1 1 0 1 0 1]  
 [0 1 1 0 1 1 1 1 0]]
```

Τώρα είναι εμφανές ότι τα στοιχεία της κύριας διαγωνίου είναι όλα μηδέν όπως περιμέναμε. Ο δεύτερος τρόπος επιστρέφει τον πίνακα `vis2_array[]` τον οποίο θα χρειαστούμε αργότερα.

Ο τρίτος τρόπος κατασκευής του πίνακα γειτνίασης χρησιμοποιεί την ενσωματωμένη συνάρτηση `nx.to_numpy_matrix(G, nodelist=sorted(G.nodes()))` και η οποία ταξινομεί τα στοιχεία του πίνακα κατά αύξουσα σειρά (από τον κόμβο '0' στον κόμβο '1', στον κόμβο '2', κλπ.) και επιστρέφει τον πίνακα `my_matrix` όπως φαίνεται παρακάτω:

MYMATRIX:

```
[[0. 1. 1. 1. 1. 0. 0. 1. 0.]
 [1. 0. 1. 1. 1. 1. 0. 0. 1.]
 [1. 1. 0. 1. 1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1. 1. 0. 1. 0.]
 [1. 1. 1. 1. 0. 0. 1. 1. 1.]
 [0. 1. 1. 1. 0. 0. 1. 0. 1.]
 [0. 0. 0. 0. 1. 1. 0. 1. 1.]
 [1. 0. 1. 1. 1. 0. 1. 0. 1.]
 [0. 1. 1. 0. 1. 1. 1. 1. 0.]]
```

Έχοντας στη διάθεσή μας τον πίνακα γειτνίασης `vis2_array[]` καλούμε τη νέα συνάρτηση `matrix_with_weights()`. Στόχος μας είναι να κατασκευάσουμε τον πίνακα με βάρη δηλαδή τον πίνακα που περιέχει τα μήκη όλων των ακμών. Ακολουθεί ο κώδικας της `matrix_with_weights()`:

```
1 def matrix_with_weights(array1, array2):
2     A99 = []
3     f_my_matrix2 = [] # ***** ADJACENCY
MATRIX WITH WEIGHTS *****
4
5     for i in range((len(array1))):
6         f_my_matrix2.append([])
7         for j in range((len(array1))):
8             if array1[i][j] != 0:
9                 A99.append(array2[i])
10                A99.append(array2[j])
11
12                line5 = LineString(A99)
13                A99.clear()
14
15                f_my_matrix2[i].append(round(line5.length, 3))
16            else:
17                f_my_matrix2[i].append(0)
18
19        for i in range((len(f_my_matrix2))):
20            print(f_my_matrix2[i])
21
22        return f_my_matrix2
```

Η συνάρτηση `matrix_with_weights()` δέχεται ως όρισμα τον πίνακα γειτνίασης `vis2_array[]` και τον πίνακα `ob_1[]` που περιέχει τις συντεταγμένες όλων των σημείων. Όπως παρατηρούμε στη γραμμή 8, ελέγχει ποια στοιχεία του πίνακα γειτνίασης είναι διάφορα του μηδενός. Όταν βρει τέτοιο στοιχείο τοποθετεί τις συντεταγμένες των κόμβων i, j στον πίνακα `A99[]` και στη συνέχεια κατασκευάζει ένα αντικείμενο τύπου `LineString`, υπολογίζει το μήκος του αντικειμένου και το τοποθετεί στον πίνακα `f_my_matrix2[]`. Προσέχουμε σε κάθε επανάληψη να αδειάζουμε τον `A99[]` (γραμμή 13, εντολή `A99.clear()`) ώστε η συνάρτηση να υπολογίζει σωστά το μήκος κάθε ακμής που αποτελείται από δύο μόνο κόμβους και όχι το μήκος αντικειμένων `LineString` που αποτελούνται από πολλές ακμές. Τελικά η συνάρτηση επιστρέφει τον πίνακα `my_matrix2[]`. Παρακάτω βλέπουμε το αποτέλεσμα που εκτυπώνεται μετά την κλήση της συνάρτησης για έναν γράφο με εννιά κόμβους:

```
MY_MATRIX2: [[0, 1.0, 1.414, 2.828, 3.606, 0, 0, 4.528, 0],
[1.0, 0, 1.0, 2.236, 3.162, 2.828, 0, 0, 11.885],
[1.414, 1.0, 0, 1.414, 2.236, 2.236, 0, 3.808, 11.543],
[2.828, 2.236, 1.414, 0, 1.0, 1.0, 0, 3.536, 0],
[3.606, 3.162, 2.236, 1.0, 0, 0, 1.0, 2.915, 10.112],
[0, 2.828, 2.236, 1.0, 0, 0, 1.0, 0, 9.341],
[0, 0, 0, 0, 1.0, 1.0, 0, 3.808, 9.124],
[4.528, 0, 3.808, 3.536, 2.915, 0, 3.808, 0, 12.5],
[0, 11.885, 11.543, 0, 10.112, 9.341, 9.124, 12.5, 0]]
```

Παρατηρούμε ότι ο πίνακας με βάρη είναι κι αυτός συμμετρικός ως προς την κύρια διαγώνιο.

Στη συνέχεια του διαγράμματος ροής καλείται η συνάρτηση `find_max_weight()`. Στόχος είναι να βρούμε το μέγιστο βάρος το οποίο ισούται με το άθροισμα των βαρών όλων των ακμών του γράφου. Ο κώδικας δίνεται παρακάτω:

```
1 def find_max_weight(array1):  
2     max_weight_row = np.add.reduce(array1)  
3     print(f" MAX_WEIGHT PER ROW: {max_weight_row}")  
4     f_max_weight = np.add.reduce(max_weight_row)  
5  
6     return f_max_weight
```

Η συνάρτηση παίρνει ως όρισμα τον πίνακα με βάρη `my_matrix2[]` και επιστρέφει το μέγιστο βάρος `max_weight`. Για να το βρούμε καλούμε στη γραμμή 2 την ενσωματωμένη συνάρτηση `np.add_reduce()` της βιβλιοθήκης Numpy και η οποία προσθέτει τα στοιχεία κάθε σειράς. Στη γραμμή 4 ξανακαλούμε την ίδια συνάρτηση για τον πίνακα-γραμμή που έχει προκύψει, οπότε παίρνουμε το συνολικό άθροισμα.

Ο λόγος για τον οποίο βρίσκουμε το `max_weight` είναι ο εξής: γνωρίζουμε ότι στον αλγόριθμο Dijkstra κατά το πρώτο βήμα θέτουμε την απόσταση προς όλους τους κόμβους ίση με άπειρο και στη συνέχεια σε κάθε γύρο αντικαθιστούμε με την ελάχιστη έως τώρα απόσταση. Για να υλοποιήσουμε προγραμματιστικά την έννοια του απείρου πρέπει να βρούμε μια κατάλληλη τιμή και η οποία στην περίπτωσή μας είναι η μεταβλητή `max_weight`.

4.4 Μέρος Τρίτο – Αρχικοποίηση Πληθυσμού

Όπως αναφέραμε και στο Κεφάλαιο 3, ο πληθυσμός αποτελεί το σύνολο των δυνατών λύσεων (μονοπατιών) και ενδέχεται μέσα σε αυτό να υπάρχουν πολλά αντίγραφα ενός ατόμου. Στον αλγόριθμο που παρουσιάζουμε, ο πληθυσμός παραμένει σταθερός από γενιά σε γενιά. Ο τρόπος που επιλέξαμε να γίνει η αρχικοποίηση του πληθυσμού αποτελείται από δύο μέρη. Στο πρώτο μέρος κατασκευάζουμε το αρχικό τμήμα του πληθυσμού (πίνακας `starting_pool[]`). Το σκεπτικό ήταν να χρησιμοποιήσουμε τον πίνακα γειτνίασης που έχουμε κατασκευάσει (πίνακας `vis2_array[]`). Από τον πίνακα αυτόν βρίσκουμε τους γείτονες του κόμβου-πηγή, οπότε αρχικοποιούμε τον `starting_pool[]` με πίνακες όπου ο καθένας αποτελείται από δύο μόνο κόμβους, τον κόμβο-πηγή και έναν από τους γείτονες του. Με αυτό τον τρόπο διευκολύνουμε τον αλγόριθμο ώστε να έχει καλύτερη επίδοση.

Η συνάρτηση `generatePolygon()` επιστρέφει τα πολύγωνα σε θέσεις οι οποίες δεν είναι εκ των προτέρων γνωστές. Επίσης, οι συντεταγμένες των κόμβων πηγής/προορισμού δεν είναι από πριν γνωστές αφού για τη δημιουργία τους χρησιμοποιούμε τη συνάρτηση `random.randint()`¹¹. Επομένως, μετά από πειράματα, παρατηρήσαμε ότι ο κόμβος-πηγή ενδέχεται να έχει μικρό πλήθος γειτόνων.

Εδώ θα πρέπει να σημειώσουμε ότι όταν το πλήθος των γειτόνων του κόμβου-πηγή είναι σχετικά μικρό, αντίστοιχα μικρός σε μέγεθος θα είναι και ο αρχικός πληθυσμός οπότε η επίδοση του γενετικού αλγορίθμου θα επηρεαστεί προς το χειρότερο. Για αυτό το λόγο λαμβάνουμε κατάλληλα μέτρα, όπως θα δούμε στη συνέχεια.

¹¹ Θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση `random.uniform()` η οποία επιστρέφει δεκαδικούς αριθμούς ανάμεσα σε κάποιο επιθυμητό διάστημα. Χρησιμοποιήσαμε όμως τη συνάρτηση `random.randint()` η οποία επιστρέφει ακέραιους αριθμούς ώστε να διευκολυνθούμε κατά την επεξεργασία των αποτελεσμάτων.

4.4.1 Κατασκευή Πίνακα `starting_pool[]`

Όπως παρατηρούμε από το διάγραμμα ροής καλούμε κατευθείαν τη συνάρτηση `create_starting_pool[]`. Ακολουθεί ο κώδικας:

```
1     def create_starting_pool(array1, pos, array2, my_s2, my_t2):
2
3         f_index22 = False
4         my_source = array1[len(array1) - 2]
5         print(f"\nSOURCE NODE NUMBER IS: {my_source}")
6         #     ADJUST     ACCORDING     TO     NUMBER     OF     NODES
*****
7         source_neighhors = array1[my_s2]
8         print(f"\n SOURCE_NEIGHBORS: {source_neighhors}")
9
10        #     *****     SOURCE_NEIGHBORS2     ARRAY
TO STORE SOURCE NEIGHBORS
11
12        f_source_neighhors2 = []
13        for i in range(len(source_neighhors)):
14            if source_neighhors[i] == 1:
15                f_source_neighhors2.append(i)
16
17        print(f"\n SOURCE_NEIGHBORS_2: {f_source_neighhors2}")
18
19        print(f"\n-----CREATE SLICED MATRICES -----
-----")
20        f_starting_pool = []
21        if len(f_source_neighhors2) < 5:
22            print("\n SOURCE NEIGHBORS < 5")
```

```
23         for i in range(len(f_source_neighbors2)):
24             for j in range(0, i + 1):
25                 for k in range(0, j + 1):
26                     m = [my_s2] #
***** INITIALIZE WITH SOURCE NODE
NUMBER
27                     m.append(f_source_neighbors2[j])
28                     print(f' M: {m}')
29                     f_starting_pool.append(m)
30     else:
31         print("\n SOURCE NEIGHBORS > 5")
32         for i in range(len(f_source_neighbors2)):
33             for j in range(0, i + 1):
34                 m = [my_s2] #
***** INITIALIZE WITH SOURCE NODE
NUMBER
35                 m.append(f_source_neighbors2[j])
36                 print(f' M: {m}')
37                 f_starting_pool.append(m)
38
39     print(f"\nSTARTING POOL: {f_starting_pool}")
40     n_starting_pool = np.array(f_starting_pool)
41     print(f"\nSTARTING          POOL          SHAPE:
{n_starting_pool.shape}")
42
43     all_neighbors = set(f_source_neighbors2)
44     print(f"\nSET OF NEIGHBORS: {all_neighbors}")
45
46     # ***** NODE_ARR2  ARRAY  TO  STORE
SOURCE'S NODES COORDINATES
```

```
47
48         f_node_arr2 = []
49
50         for j in f_source_neighhors2:
51             f_node_arr2.append(pos.get(j))
52
53         #         ADJUST         ACCORDING         TO         NODE         NUMBER
54         *****
55         if array2[my_t2] in f_node_arr2:
56             f_index = f_source_neighhors2.index(j) # INDEX TO
57             STORE TARGET NODE
58
59             print(f" \nTARGET2 FOUND AT POSITION: {f_index+1}
60             ")
61
62             f_index22 = True
63
64         else:
65
66             print(" - - -TARGET2 NOT IN SOURCE NEIGHBORS - -
67             -")
68
69
70
71         print(f"\n NODE_ARRAY2: {f_node_arr2}")
72
73         return         f_starting_pool,         f_node_arr2,
74         f_source_neighhors2, f_index22
```

Η συνάρτηση δέχεται ως ορίσματα τον πίνακα γειτνίασης (`vis2_array[]`), το λεξικό με τις συντεταγμένες των σημείων (`pos1{}`), τον πίνακα με όλες τις συντεταγμένες (`ob_1[]`) και τέλος τους αύξοντες αριθμούς των κόμβων πηγής και προορισμού που έχουμε ήδη υπολογίσει (`my_source2` και `my_target2`).

Στη γραμμή 3 αρχικοποιούμε τη `boolean` μεταβλητή `f_index22` σε `False`. Η μεταβλητή αυτή μας ενημερώνει για το αν ο κόμβος-προορισμού είναι ήδη μέσα στους γείτονες του κόμβου-πηγή.

Στη γραμμή 4 εντοπίζουμε το τμήμα του πίνακα γειτνίασης που περιέχει τους γείτονες του κόμβου-πηγή ('1' αν είναι γείτονας, '0' αν δεν είναι). Τον ονομάζουμε πίνακα `my_source[]`.

Στη γραμμή 7 βρίσκουμε τον ίδιο πίνακα χρησιμοποιώντας τώρα το όρισμα `my_source2`. Ονομάζουμε τον πίνακα αυτόν `source_neighhors[]`.

Στις γραμμές 12-15 βρίσκουμε το δείκτη i που αντιστοιχεί σε κάθε γείτονα και τους αποθηκεύουμε στον πίνακα `f_source_neighhors2[]`. Ένα παράδειγμα εκτέλεσης του κώδικα δίνεται παρακάτω:

```
----- SOURCE'S NEIGHBORS-----  
SOURCE_NEIGHBORS: [1 0 1 1 1 0 1 0 1]  
SOURCE_NEIGHBORS_2: [0, 2, 3, 4, 6, 8]
```

Στη γραμμή 21 λαμβάνουμε τα κατάλληλα μέτρα για το αν ο γείτονας έχει κάτω ή πάνω από πέντε γείτονες. Αν έχει κάτω από 5 γείτονες χρησιμοποιούμε ένα τριπλό βρόχο και στη γραμμή 26 αρχικοποιούμε τον πίνακα `m[]` μόνο με τον κόμβο-πηγή. Στη γραμμή 27 επαυξάνουμε τον `m[]` με κάποιον από τους γείτονές του και στη γραμμή 29 τοποθετούμε τον `m[]` στον πίνακα `f-starting_pool[]`. Ακολουθεί ένα παράδειγμα εκτέλεσης του παραπάνω κώδικα όταν ο κόμβος-πηγή έχει το αναγνωριστικό νούμερο '7':

```
-----CREATE SLICED MATRICES -----  
M: [7, 0]  
M: [7, 2]  
M: [7, 3]  
M: [7, 4]  
M: [7, 6]  
M: [7, 8]  
STARTING POOL: [[7, 0], [7, 2], [7, 3], [7, 4], [7, 6], [7, 8]]
```

Στις γραμμές 32-37, όταν ο κόμβος-πηγή έχει περισσότερους από πέντε γείτονες, χρησιμοποιούμε ένα διπλό αντί για τριπλό βρόχο και κάνουμε ακριβώς τα ίδια με πριν.

Στις γραμμές 40-41 μετατρέπουμε τον `f_starting_pool[]` σε `numpy array` και υπολογίζουμε το μέγεθος του.

Στις γραμμές 48-51 βρίσκουμε τις συντεταγμένες κάθε γείτονα από το λεξικό `pos1{ }` και στη γραμμή 54-55 ελέγχουμε αν ο κόμβος-προορισμού βρίσκεται μέσα στους γείτονες. Αν βρίσκεται τότε στη γραμμή 57 αλλάζουμε τον δείκτη `f_index22` από `False` σε `True`.

Τελικά η συνάρτηση επιστρέφει τον πίνακα `starting_pool[]`, τον πίνακα με τις συντεταγμένες των γειτόνων `node_arr2[]`, τον πίνακα με τους αύξοντες αριθμούς των κόμβων των γειτόνων `source_neighhors2[]` και τη μεταβλητή `index22`.

4.4.2 Αρχικοποίηση Πίνακα population[]

Στο επόμενο βήμα του γενετικού αλγορίθμου καλούμε τη συνάρτηση `create_population()`.

Ο αντίστοιχος κώδικας δίνεται παρακάτω:

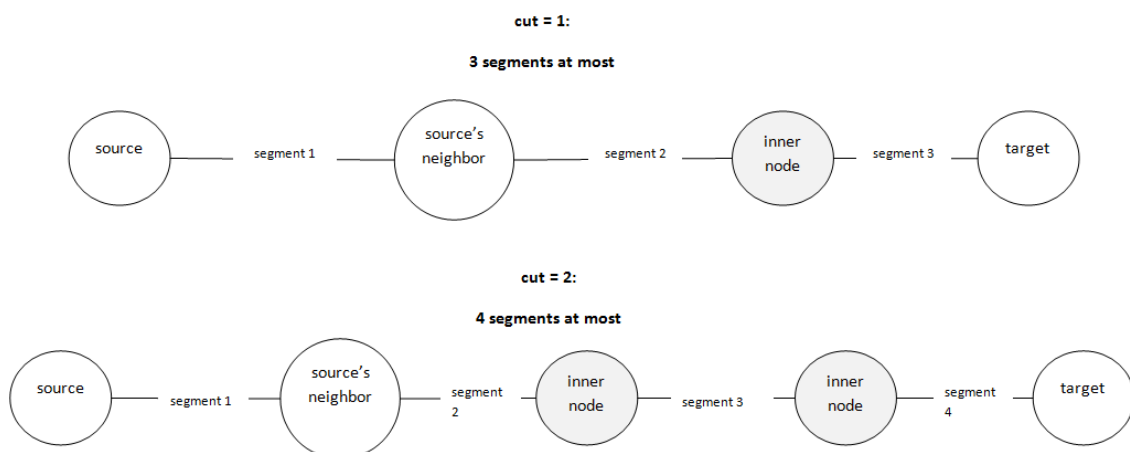
```

1 def create_population(f_cut, array1, array2, my_s2):
2
3     f_population = []
4     array_to_conc = []
5
6     for i in range(len(array1)): # ITERATION AS LONG AS
STARTING_POOL'S LENGTH !!
7         # ADJUST RANGE ACCORDING TO NUMBER OF NODES IN
THE GRAPH !!
8         # ADJUST NUMBER OF SOURCE NODE !!!
9         m2 = [x for x in range(len(array2)) if x !=
my_s2]
10        print(f" M2: {m2}")
11        random_m2 = random.sample(m2[:my_s2], k=f_cut) #
>>>>>>>>>>>>>>> SET CUTOFF OF SOLUTIONS
12        print(f"RANDOM M2: {random_m2}")
13        new_m2 = random_m2[:]
14        print(f"NEW M2: {new_m2}")
15        new_m2.extend([m2[len(m2) - 1]])
16        print(f"FINAL M2: {new_m2}")
17        array_to_conc.append(new_m2)
18        print("\n-----")
19
20    print(f" ARRAY_TO_CONCATENATE: {array_to_conc}")
21
22    print(f"\n-----
POPULATION CREATION -----")
23    # CREATE POPULATION
24    for i in range(len(array1)):
25        p1 = array1[i].copy()
26        print(f" P1: {p1}")
27        p2 = array_to_conc[i].copy()
28        print(f"P2: {p2}")
29
30        # CREATION OF MEMBERS
31        member = np.concatenate([p1, p2])
32        print(member.dtype)
33        f_population.append(member)
34        print(f" MEMBER: {member} ")
35
36    return f_population

```

Η συνάρτηση δέχεται ως ορίσματα την παράμετρο `f_cut`, τον πίνακα `starting_pool[]` που δημιουργήσαμε στο προηγούμενο βήμα, τον πίνακα `ob_1[]` με τις συντεταγμένες όλων των σημείων και τον αύξοντα αριθμό του κόμβου-πηγή `my_source2`.

Η παράμετρος `f_cut` δηλώνει από πόσα τμήματα αποτελούνται τα μονοπάτια δηλαδή τα άτομα του πληθυσμού. Το παρακάτω σχήμα εξηγεί καλύτερα το ρόλο της παραμέτρου `f_cut`:



Εικόνα 48 Μονοπάτια με παραμέτρους `cut=1` και `cut=2`

Η παράμετρος `f_cut` καθορίζεται από το χρήστη και αποθηκεύεται στη μεταβλητή `num_of_seg`.

Στις γραμμές 3-4 αρχικοποιούμε δύο πίνακες, τον `f_population[]` ο οποίος είναι και ο πίνακας που επιστρέφει τελικά η συνάρτηση και τον βοηθητικό πίνακα `array_to_conc[]`. Στη γραμμή 9 κατασκευάζουμε ένα βοηθητικό πίνακα τον `m2[]` και με τη χρήση της συνάρτησης `range()` τον αρχικοποιούμε με τους αύξοντες αριθμούς όλων των κόμβων του γράφου εκτός του κόμβου-πηγή.

Στη γραμμή 11 χρησιμοποιούμε την ενσωματωμένη συνάρτηση `random.sample()` και επιλέγουμε μέσα από τον πίνακα `m2[]` τόσους τυχαίους αριθμούς όσους καθορίζει η παράμετρος `f_cut` τους οποίους αποθηκεύουμε στον πίνακα `random_m2[]`. Χρησιμοποιούμε την `random.sample()` διότι επιλέγει τυχαίους αριθμούς χωρίς επανατοποθέτηση, έτσι ώστε να μην έχουμε ίδιους κόμβους μέσα στους πίνακες που κατασκευάζουμε. Επίσης καθορίζουμε το όριο (`m2[:my_source2]`) έτσι ώστε η δειγματοληψία να γίνεται χωρίς να επιλέγεται ο τελευταίος κόμβος που αναπαριστά τον κόμβο-προορισμό. Θέτουμε τον πίνακα `new_m2[]` ίσο με τον `random_m2[]` και στη

γραμμή 15 προσθέτουμε στον new_m2[] και τον κόμβο-προορισμό. Ακολουθεί ένα παράδειγμα με κόμβο-πηγή τον κόμβο '6' και κόμβο-προορισμό τον κόμβο '7':

```
M2: [0, 1, 2, 3, 4, 5, 7]
RANDOM M2: [0, 3, 5]
NEW M2: [0, 3, 5]
FINAL M2: [0, 3, 5, 7]
```

```
-----
M2: [0, 1, 2, 3, 4, 5, 7]
RANDOM M2: [4, 2, 1]
NEW M2: [4, 2, 1]
FINAL M2: [4, 2, 1, 7]
```

```
-----
M2: [0, 1, 2, 3, 4, 5, 7]
RANDOM M2: [5, 2, 1]
NEW M2: [5, 2, 1]
FINAL M2: [5, 2, 1, 7]
```

```
-----
M2: [0, 1, 2, 3, 4, 5, 7]
RANDOM M2: [0, 1, 2]
NEW M2: [0, 1, 2]
FINAL M2: [0, 1, 2, 7]
```

Στη γραμμή 17 τοποθετούμε όλους τους πίνακες `new_m2[]` μέσα στον `array_to_conc[]`.

Στη γραμμή 24 ξεκινάμε ένα `loop` τόσες φορές όσο είναι το μέγεθος του `starting_pool[]`.

Στη γραμμή 25 δημιουργούμε με τη συνάρτηση `copy()` τον πίνακα `p1[]` ο οποίος είναι αντίγραφο των πινάκων `m[]` οι οποίοι βρίσκονται μέσα στον `starting_pool[]`. Δημιουργούμε αυτά τα αντίγραφα διότι θέλουμε οι όποιες τροποποιήσεις να γίνονται μόνο στους πίνακες `p1[]` και όχι στους αρχικούς πίνακες `m[]`.

Στη γραμμή 27 κάνουμε το ίδιο με πριν και κατασκευάζουμε τα αντίγραφα `p2[]`, αλλά τώρα για τους πίνακες `m2[]` που βρίσκονται μέσα στον `array_to_conc[]`.

Στη γραμμή 31 δημιουργούμε τα πρώτα άτομα του πληθυσμού χρησιμοποιώντας τη συνάρτηση `np_concatenate()` της βιβλιοθήκης `Numpy` συνενώνοντας τους πίνακες `p1[]` και `p2[]`.

Στη γραμμή 33 τοποθετούμε όλα τα άτομα του πληθυσμού στον πίνακα `f_population[]`.

Ακολουθεί ένα παράδειγμα εκτέλεσης:

```
-----POPULATION CREATION -----  
-----  
P1: [6, 1]  
P2: [1, 0, 4, 7]  
int32  
MEMBER: [6 1 1 0 4 7]  
P1: [6, 1]  
P2: [5, 2, 4, 7]  
int32  
MEMBER: [6 1 5 2 4 7]  
P1: [6, 2]  
P2: [3, 1, 4, 7]  
int32  
MEMBER: [6 2 3 1 4 7]  
P1: [6, 1]  
P2: [3, 5, 0, 7]  
int32  
P1: [6, 3]  
P2: [3, 5, 0, 7]  
int32  
MEMBER: [6 3 3 5 0 7]  
P1: [6, 4]  
P2: [2, 4, 3, 7]  
int32  
MEMBER: [6 4 2 4 3 7]
```

Αν στη συνέχεια εκτυπώσουμε τα περιεχόμενα του `f_population[]` θα πάρουμε το παρακάτω αποτέλεσμα:

```
-----POPULATION-----
0: [6 1 1 0 4 7]
1: [6 1 5 2 4 7]
2: [6 2 3 1 4 7]
3: [6 1 3 5 0 7]
4: [6 2 1 2 0 7]
5: [6 3 5 1 3 7]
6: [6 1 5 4 0 7]
7: [6 2 4 5 0 7]
8: [6 3 0 4 1 7]
9: [6 4 3 1 5 7]
10: [6 1 4 3 0 7]
11: [6 2 5 4 1 7]
12: [6 3 3 5 0 7]
13: [6 4 2 4 3 7]
14: [6 5 1 3 4 7]
15: [6 1 4 3 5 7]
16: [6 2 5 3 0 7]
17: [6 3 5 3 0 7]
18: [6 4 0 5 3 7]
19: [6 5 3 4 2 7]
20: [6 7 1 0 3 7]
```

Έπειτα από πειράματα, παρατηρήσαμε ότι ο τυχαίος τρόπος με τον οποίο δημιουργούμε τον αρχικό πληθυσμό αποτελείται στη μεγάλη πλειονότητα από άτομα τα οποία δεν αποτελούν υπαρκτά μονοπάτια-λύσεις. Επίσης, αρκετά μονοπάτια περιέχουν ίδιους κόμβους, όπως συμβαίνει παραπάνω για το μέλος 12 (ο κόμβος 3 επαναλαμβάνεται). Όλες όμως οι τυχαίες λύσεις που δημιουργήσαμε αποτελούνται από τον κόμβο-πηγή στην 1^η θέση και τον κόμβο-προορισμό στην τελευταία θέση.

Σημειώνουμε ότι το μέγεθος του πληθυσμού δεν είναι κάθε φορά γνωστό, αφού εξαρτάται από το πλήθος των γειτόνων του κόμβου-πηγή. Αποθηκεύουμε το μέγεθος του στη μεταβλητή *c1* σύμφωνα με τον παρακάτω κώδικα:

```
1    n_population = np.array (population)
2    c1 = n_population.shape
```

Στόχος μας είναι υποβάλλοντας τον πληθυσμό στις γενετικές διαδικασίες της διασταύρωσης, μετάλλαξης και επιλογής, μετά από κάποιο προκαθορισμένο αριθμό γενιών, όλα ή τα περισσότερα άτομα του πληθυσμού να έχουν αντικατασταθεί από την καλύτερη λύση, δηλαδή το πιο σύντομο μονοπάτι.

4.5 Μέρος Τέταρτο – Αντικειμενική Συνάρτηση (Fitness Function)

Αφού δημιουργήσαμε τα πρώτα μέλη του πληθυσμού θα πρέπει να τους αποδώσουμε κάποια τιμή με βάση την οποία θα συγκρίνονται. Στο πρόβλημα εύρεσης του συντομότερο μονοπατιού μεταξύ δύο σημείων η αντικειμενική συνάρτηση που τελικά επιλέξαμε υπολογίζει τις αποστάσεις μεταξύ των ενδιάμεσων σημείων της διαδρομής και αθροίζοντάς τες προκύπτει το συνολικό μήκος της διαδρομής.

Έπειτα αθροίζουμε όλες τις τιμές των μελών του πληθυσμού και διαιρώντας με το σταθερό σε κάθε γενιά μέγεθος του πίνακα `population[]` υπολογίζουμε το μέσο όρο κάθε γενιάς (Mean Fitness Value). Στόχος μας είναι να καταγράψουμε το πώς μεταβάλλεται ο μέσος όρος από γενιά σε γενιά.

Εδώ πρέπει να σημειώσουμε ότι αφού αναζητούμε το συντομότερο μονοπάτι επιδιώκουμε να αντικαθιστούμε σε κάθε γενιά τα άτομα του πληθυσμού με άτομα που έχουν όσο το δυνατόν χαμηλότερη τιμή αντικειμενικής συνάρτησης. Ο κώδικας που υπολογίζει τις τιμές της αντικειμενικής συνάρτησης των ατόμων του αρχικού πληθυσμού και επιπρόσθετα βρίσκει τη μέση τιμή τους ακολουθεί παρακάτω:

[illegible]

```
22         else:
23             print(f"Path not valid...")
24             A199.append(max_weight)
25
26     print("\n-----MEAN FITNESS VALUE -----\\n")
27     arr700 = []
28     print(f" A199: {A199}")
29     A200 = np.array(A199)
30     athroisma1 = np.sum(A200)
31     print(f"ATHROISMA: {athroisma1}")
32
33     mean_distance = athroisma1 / c1[0]
34     print(f" MEAN DISTANCE: {mean_distance}")
35     arr700.append(mean_distance)
```

Στη γραμμή 2 αρχικοποιούμε τον πίνακα A199[] στον οποίο αποθηκεύουμε κάθε φορά την τιμή της αντικειμενικής συνάρτησης κάθε μέλους του πληθυσμού.

Στη γραμμή 4 ξεκινάμε ένα βρόχο με αριθμό επαναλήψεων ίσο με το μέγεθος του πίνακα population[] και στη γραμμή 5 χρησιμοποιούμε την ενσωματωμένη συνάρτηση is_path() της βιβλιοθήκης networkx για να ελέγξουμε αν το άτομο αντιστοιχεί σε επαρκές μονοπάτι (λύση).

Αν αποτελεί λύση τότε στις γραμμές 11-14 με τη βοήθεια του λεξικού pos1{} τοποθετούμε τις συντεταγμένες των κόμβων του μονοπατιού στον πίνακα coords_of_node[].

Στη γραμμή 18 δημιουργούμε ένα αντικείμενο τύπου LineString, υπολογίζουμε το μήκος του και το τοποθετούμε στον πίνακα A199[].

Στις γραμμές 22-24, αν το άτομο δεν αποτελεί επαρκή λύση τότε του δίνουμε τιμή αντικειμενικής συνάρτησης ίση με max_weight. Ένα παράδειγμα εκτέλεσης του κώδικα δίνεται παρακάτω:

```
-----
Path not valid...
Path not valid...
Path not valid...
3-st path [6, 1, 3, 5, 0, 7] is valid !
COORDS: [(-12, 27), (21, 46), (60, 45), (66, 36), (38, 45), (79,
179)]
Path has length: 257.451300654278
4-st path [6, 2, 1, 2, 0, 7] is valid !
COORDS: [(-12, 27), (29, 39), (21, 46), (29, 39), (38, 45), (79,
179)]
Path has length: 214.9290447310476
Path not valid...
Path not valid...
```

```
7-st path [6, 2, 4, 5, 0, 7] is valid !  
COORDS: [(-12, 27), (29, 39), (59, 43), (66, 36), (38, 45), (79,  
179)]  
Path has length: 252.42796845634658
```

Στη γραμμή 27 αρχικοποιούμε τον πίνακα `arr700[]` στον οποίο αποθηκεύουμε σε κάθε γενιά το μέσο όρο των τιμών της αντικειμενικής συνάρτησης κάθε μέλους του πληθυσμού. Εδώ πρέπει να προσέξουμε ότι η αρχικοποίηση του πίνακα `arr700[]` γίνεται εκτός του κύριου βρόχου επανάληψης.

Στη γραμμή 33 αποθηκεύουμε στη μεταβλητή `mean-distance` το μέσο όρο μόνο για τον αρχικό πληθυσμό που κατασκευάσαμε στο τρίτο μέρος και την αποθηκεύουμε στον πίνακα `arr700[]`.

4.6 Μέρος Πέμπτο – Κύριος Βρόχος Επανάληψης

Αφού έχουμε στη διάθεσή μας τον πίνακα `A199[]` στον οποίο κρατάμε το `score` κάθε μέλους του πληθυσμού και τον πίνακα `arr700[]` στον οποίο κρατάμε το μέσο όρο κάθε γενιάς, προχωράμε στο επόμενο βήμα του αλγορίθμου και εισερχόμαστε στον κυρίως βρόχο.

4.6.1 Κατασκευή Δομής Δεδομένων `data4`

Δίνουμε κατευθείαν τον κώδικα παρακάτω:

```
1     num_of_mut = 0  
2  
3     num_of_swap = 0  
4     num_of_shuffle = 0  
5     num_of_del = 0  
6     k = 0  
7  
8     # ADJUST K LIMIT = (GENERATIONS +1) !!!!  
9  
10    while k < num_of_gen: #  
***** FIRST CHECKPOINT  
*****  
11        print("\n-----DATA STRUCTURE_4-----  
-----\n")  
12        data4 = np.zeros(c1[0], dtype={'names': ('number',  
'path', 'total_fitness'),
```

```

13                                     'formats':
14     ('i4', 'object', 'f8'))
15     data4['number'] = [i for i in range(c1[0])]
16     data4['path'] = population
17     data4['total_fitness'] = A199
18     print(data4)
19     if k == 0:
20         print(' * ' * 20 + f"\n  INITIAL MEAN FITNESS
VALUE IS: {mean_distance}")
21
22     if k == 101: # *****
SECOND CHECKPOINT *****
23         print(" \n- - - - -GENERATIONS ABOVE 100 - - - -
- - -")
24         break
25
26     print(f"\n* * * * * START NEXT
GENERATION NUMBER {k + 1} * * * * *\n")
27
28     cut4 = len(population)
29     print(f"CUT4: {cut4}")
30
31     if cut4 % 2 == 0:
32         cut44 = cut4
33         print(f"CUT44: {cut44}")
34     else:
35         cut44 = cut4 - 1
36         print(f"CUT44: {cut44}")
37
38     print(f"-----{cut44} RANDOM PARENTS-----
-----")
39     parents1 = random.choices(population, k=cut44)
40     print(f" PARENTS1: {parents1}")
41
42     arr2200 = []
43     arr2300 = []
44     arr2600 = []
45     for i in range(len(parents1)):
46         print(data4[i])
47         arr2200.append(data4[i])
48         arr2300.append(data4[i]['total_fitness'])
49         arr2600.append(data4[i]['path'])
50
51     print("-----ARRAY 2200 - PARENTS1
DATA-----")
52     print(f" ARRAY 2200 : {arr2200}")
53     for i in range(len(arr2200)):
54         print(arr2200[i])
55
56     print("-----ARRAY 2300 - PARENTS1 FITNESS---
-----")
57     print(f" ARRAY 2200 : {arr2300}")
58     for i in range(len(arr2300)):
59         print(f" {i}: {arr2300[i]}")

```

Στις γραμμές 1-5 οι οποίες βρίσκονται εκτός βρόχου, αρχικοποιούμε τέσσερις μεταβλητές την `num-of_mut`, η οποία μετράει το πλήθος όλων των μεταλλάξεων, την `num_of_swap` που μετράει το πλήθος των μεταλλάξεων τύπου `swap`, την `num_of_shuffle` που μετράει το πλήθος των μεταλλάξεων τύπου `shuffle` και την `num_of_del` που μετράει το πλήθος των μεταλλάξεων τύπου `deletion`. Αυτές οι μεταβλητές θα χρειαστούν αργότερα κατά το στάδιο της μετάλλαξης.

Στη γραμμή 6 αρχικοποιούμε τη μεταβλητή k η οποία μετράει τον αριθμό των γενιών. Η μεταβλητή `num_of_gen` στη γραμμή 10 συμβολίζει το πλήθος των επαναλήψεων και επιλέγεται από το χρήστη.

Μέσα στο βρόχο, στις γραμμές 12-17 κατασκευάζουμε μια πολύ χρήσιμη δομή, την `data4[()]`. Τη δυνατότητα να κατασκευάζουμε τέτοιες δομές, οι οποίες ονομάζονται `structured arrays`, μας τη δίνει η βιβλιοθήκη `Numpy`. Η δομή `data4` μας δίνει τη δυνατότητα να αποθηκεύουμε διαφορετικές κατηγορίες δεδομένων σε μια μόνο δομή.

Όπως βλέπουμε στη γραμμή 12 αρχικοποιούμε τρία πεδία με διαφορετικά `formats`:

- πεδίο `'number'` όπου αποθηκεύονται ακέραιοι,
- πεδίο `'path'` όπου αποθηκεύονται αντικείμενα `array`,
- πεδίο `'total_fitness'` όπου αποθηκεύονται πραγματικοί.

Χρησιμοποιούμε για την αρχικοποίηση τη συνάρτηση `np.zeros()` με όρισμα τη μεταβλητή `c1[0]`, η οποία είναι ίση με το μέγεθος του πληθυσμού (διάσταση του πίνακα `population[[]]`).

Στη γραμμή 15 αριθμούμε κάθε άτομο του πληθυσμού, στη γραμμή 16 αποδίδουμε στο πεδίο `'path'` τα άτομα που βρίσκονται κάθε φορά μέσα στον `population[[]]` και στη γραμμή 17 αποδίδουμε σε κάθε μέλος το αντίστοιχο `score` που έχουμε αποθηκεύσει στον `A199[[]]`.

Εδώ πρέπει να σημειώσουμε ότι η θέση της δομής `data4[()]` είναι καίρια διότι μέσα στον κυρίως βρόχο τα πεδία της θα ανανεώνονται σε κάθε γενιά.

Ακολουθεί ένα παράδειγμα για έναν αρχικό πληθυσμό με 16 μέλη, όπου φαίνεται η ευκολία στην επεξεργασία των δεδομένων που μας προσφέρει η δομή `data4[()]`:

-----DATA STRUCTURE_4-----

```
[ ( 0, array([6, 1, 1, 0, 4, 7]), 2914.002      )
  ( 1, array([6, 1, 5, 2, 4, 7]), 2914.002      )
  ( 2, array([6, 2, 3, 1, 4, 7]), 2914.002      )
  ( 3, array([6, 1, 3, 5, 0, 7]), 257.45130065)
  ( 4, array([6, 2, 1, 2, 0, 7]), 214.92904473)
  ( 5, array([6, 3, 5, 1, 3, 7]), 2914.002      )
  ( 6, array([6, 1, 5, 4, 0, 7]), 2914.002      )
  ( 7, array([6, 2, 4, 5, 0, 7]), 252.42796846)
  ( 8, array([6, 3, 0, 4, 1, 7]), 2914.002      )
  ( 9, array([6, 4, 3, 1, 5, 7]), 2914.002      )
  (10, array([6, 1, 4, 3, 0, 7]), 2914.002      )
  (11, array([6, 2, 5, 4, 1, 7]), 2914.002      )
  (12, array([6, 3, 3, 5, 0, 7]), 2914.002      )
  (13, array([6, 4, 2, 4, 3, 7]), 270.88785548)
  (14, array([6, 5, 1, 3, 4, 7]), 2914.002      )
  (15, array([6, 1, 4, 3, 5, 7]), 2914.002      )
```

]

4.6.2 Επιλογή Γονιών (Parent Selection)

Μέσα στον κυρίως βρόχο επιλέγουμε από τον πληθυσμό τα άτομα εκείνα που θα παίξουν το ρόλο των γονιών. Υλοποιήσαμε τον γενετικό αλγόριθμο με τέτοιο τρόπο ώστε το μοναδικό στάδιο στο οποίο εφαρμόζεται η διαδικασία της επιλογής να είναι το στάδιο επιλογής γονιών. Σε πρώτη φάση, αφού επιλέξουμε τους γονείς διεξάγουμε έναν διαγωνισμό (tournament) μεταξύ τους ώστε τα άτομα με το μεγαλύτερο score (winners) να αποτελέσουν τελικά τους γονείς θα διασταυρωθούν. Στη συνέχεια αποθηκεύουμε σε ζευγάρια τους γονείς αυτούς μέσα στον πίνακα `pool[]`. Όσα ζευγάρια είναι μέσα στον πίνακα `pool[]` τόσα παιδιά θα γεννηθούν, αφού υλοποιήσαμε τον αλγόριθμο με τέτοιο τρόπο ώστε από κάθε ζευγάρι να προέρχεται μόνο ένα παιδί.

Προκειμένου όμως κάθε ζευγάρι να αποτελείται από δύο μόνο άτομα πρέπει να λάβουμε μέτρα. Για αυτό στις γραμμές 28-35 του κύριου βρόχου κάνουμε το εξής: αρχικοποιούμε τη μεταβλητή `cut4` ίση με το μέγεθος του πίνακα `population[]`. Αν το μέγεθος είναι άρτιο αριθμός τότε η μεταβλητή `cut44` ισούται με την `cut4` αλλιώς ισούται με ένα άτομο λιγότερο.

Στη γραμμή 39 του κύριου βρόχου επιλεγούμε τους γονείς με τη συνάρτηση `random.choises()`, η οποία γίνεται με επανατοποθέτηση, επομένως δυο παιδιά μπορεί να έχουν τους ίδιους γονείς. Αποθηκεύουμε τους γονείς που επιλέξαμε στον πίνακα `parents1[]`.

Στις γραμμές 45-49 χρησιμοποιούμε τη δομή `data4[()]` και για όσους γονείς βρίσκονται μέσα στον `parents1[]` αποθηκεύουμε:

- στον `arr2200[]` όλα τα δεδομένα της δομής για αυτό το γονιό,
- στον `arr2300[]` την τιμή της αντικειμενικής συνάρτησης για αυτό το γονιό,
- στον `arr2600[]` το αντίστοιχο μονοπάτι.

Αφού έχουμε αποθηκεύσει στον `parents1[]` άρτιο αριθμό γονιών καλούμε τη συνάρτηση `begin_tournament()` και τους συγκρίνουμε ανά δύο μεταξύ τους. Χωρίζουμε αυτή την ενότητα σε δύο υποενότητες για καλύτερη κατανόηση.

4.6.2.1 Tournament

Ακολουθεί ο κώδικας της συνάρτησης `begin_tournament()`:

```
1  def begin_tournament(array1, array2, array3):
2      parents2 = []
3      arr2500 = []
4      f_arr2700 = []
5
6      for j in range(0, len(array1) - 1, 2):
7          if array1[j] <= array1[j + 1]:
8              parents2.append(array1[j])
9              arr2500.append(array2[j])
10             f_arr2700.append(array3[j])
11         else:
12             parents2.append(array1[j + 1])
13             arr2500.append(array2[j + 1])
14             f_arr2700.append(array3[j + 1])
15     print("----- PARENTS2 FITNESS ----
-----")
16     print(f" ARRAY PARENTS2 : {parents2}")
17
18     for i in range(len(parents2)):
19         print(f" {i}: {parents2[i]}")
20
21     print("-----ARRAY 2500 - PARENTS2
DATA -----")
22     print(f" ARRAY 2500 : {arr2500}")
23     for i in range(len(arr2500)):
24         print(arr2500[i])
25
26     print("-----ARRAY 2700 - PARENTS2
PATHS-----")
27     print(f" ARRAY 2700 : {f_arr2700}")
28     for i in range(len(f_arr2700)):
29         print(f_arr2700[i])
30
31     return f_arr2700
```

Η συνάρτηση παίρνει ως όρισμα τον πίνακα `arr2200[]`, όπου όπως είπαμε έχουμε αποθηκεύσει τον αύξοντα αριθμό, το μονοπάτι που παριστάνει και την τιμή της αντικειμενικής συνάρτησης για κάθε γονιό. Σαν δεύτερο όρισμα παίρνει τον πίνακα `arr2300[]`, όπου έχουμε αποθηκεύσει μόνο την τιμή της αντικειμενικής συνάρτησης και τέλος σαν τρίτο όρισμα παίρνει τον πίνακα `arr2600[]`, όπου έχουμε αποθηκεύσει το αντίστοιχο μονοπάτι.

Στη γραμμή 2 αρχικοποιούμε έναν καινούριο πίνακα τον `parents2[]` μέσα στον οποίο θα αποθηκεύσουμε την τιμή της αντικειμενικής συνάρτησης των νικητών (winners) του διαγωνισμού. Στη γραμμή 7 συγκρίνουμε την τιμή της αντικειμενικής συνάρτησης των γονιών ανά δύο μεταξύ τους και τοποθετούμε για όποιο γονέα έχει τη μικρότερη τιμή:

- στον πίνακα `parents2[]` την τιμή αυτή,
- στον πίνακα `arr2500[]` όλα τα δεδομένα του νικητή,
- στον πίνακα `f_arr2700[]` το μονοπάτι που αναπαριστά ο νικητής.

Τελικά η συνάρτηση επιστρέφει τον πίνακα `arr2700[]`. Σημειώνουμε ότι επιλέγοντας σε κάθε επανάληψη ως γονείς τα άτομα με την καλύτερη (μικρότερη) τιμή αντικειμενικής συνάρτησης αναγκάζουμε το γενετικό αλγόριθμο να συγκλίνει στην επιθυμητή λύση δηλαδή σε διαδρομές με όλο και μικρότερο μήκος. Αυτό συμβαίνει διότι η κατάλληλη πληροφορία μεταδίδεται μέσω της διαδικασίας της διασταύρωσης (crossover) από τους γονείς στα παιδιά. Αν δεν εφαρμόζαμε καμία συνθήκη ελέγχου ως προς την επιλογή των γονιών, ο αλγόριθμος απλώς θα διεξήγαγε μια τυχαία αναζήτηση (blind search) μέσα στο χώρο των δυνατών λύσεων.

4.6.2.2. Κατασκευή Πίνακα `pool[]`

Αφού έχουμε στη διάθεσή μας τον πίνακα `arr2700[]` με τις καλύτερες λύσεις (μονοπάτια) καλούμε την επόμενη συνάρτηση `create_pool()` με την οποία τοποθετούμε ζευγάρια νικητών μέσα στον πίνακα `pool[]`. Όσα ζευγάρια έχει μέσα ο πίνακας `pool[]` τόσα παιδιά θα γεννηθούν. Παρακάτω δίνουμε τον αντίστοιχο κώδικα:

```
1 def create_pool(array1):
2
3     function_pool = []
4     if len(array1) != 2: # IF THERE ARE ONLY TWO WINNERS
5         BYPASS
```

```
6         cut3 = len(array1) % 2
7         print(f"CUT3: {cut3}")
8         if cut3 != 0:
9             cut33 = int(len(array1) / 2)
10            if cut33 % 2 != 0:
11                cut333 = cut33 + 1
12                print(f"CUT333:{cut333}")
13            else:
14                cut333 = cut33
15                print(f"CUT333:{cut333}")
16        else:
17            cut33 = int(len(array1) / 2)
18            print(f"CUT33:{cut33}")
19            if cut33 % 2 != 0:
20                cut333 = cut33 - 1
21                print(f"CUT333:{cut333}")
22            else:
23                cut333 = cut33
24                print(f"cut333: {cut333}")
25
26        for i in range(cut333): # ADJUST PAIRS ACCORDING
TO NUMBER OF WINNERS
27            function_pool.append(
28                random.sample(array1, k=2)) # SAMPLE
WITHOUT REPLACEMENT SO THAT NO PAIRS HAVE SAME PARENTS
29        else:
30            for _ in range(10):
31
32                function_pool.append(array1[0]) # CREATE
POOL WITH THE SAME TEN PARENTS
33
34            for pair in function_pool:
35                print(pair)
36
37            print(f" POOL IS : {[pairs1 for pairs1 in
function_pool]} \t ")
38            return function_pool
```

Η συνάρτηση δέχεται ως όρισμα τον πίνακα `arr2700[]` του προηγούμενου βήματος. Επειδή στον πίνακα `pool[]` αποθηκεύουμε ζευγάρια νικητών πρέπει να λάβουμε μέτρα ώστε το πλήθος των στοιχείων του πίνακα να είναι άρτιος αριθμός. Αρχικοποιούμε λοιπόν στη γραμμή 6 μια νέα μεταβλητή την `cut3` και την θέτουμε ίση με το υπόλοιπο της διαίρεσης του μήκους του πίνακα `arr2700[]` με το 2. Στη γραμμή 8 ελέγχουμε αν το υπόλοιπο είναι διάφορο του μηδενός και αν είναι τότε ορίζουμε μια νέα μεταβλητή την `cut33` και την θέτουμε ίση με το πηλίκο της διαίρεσης του μήκους του `arr2700[]` με το 2. Αν το αποτέλεσμα είναι διάφορο του μηδενός ορίζουμε μια νέα μεταβλητή την `cut333` ίση με την `cut33` συν 1 αλλιώς θέτουμε την `cut333` ίση με την `cut33`. Αν το υπόλοιπο της διαίρεσης του `cut3` με το 2 είναι ίσο με το μηδέν επαναλαμβάνουμε το προηγούμενο

κομμάτι αλλά τώρα στην περίπτωση που το πηλίκο της διαίρεσης της `cut33` με το 2 είναι διάφορο του μηδενός θέτουμε την `cut333` ίση με την `cut33` μείον 1. Αν υπάρχουν μόνο δύο νικητές (έλεγχος στη γραμμή 4), τότε προσπερνάμε όλα τα παραπάνω και στις γραμμές 30-32 τοποθετούμε στον `pool[]` δέκα ίδιους γονείς. Τελικά σε κάθε περίπτωση με το παραπάνω έλεγχο εξασφαλίζουμε ότι ο πίνακας `pool[]` θα αποθηκεύει άρτιο αριθμό ζευγαριών.

Στη γραμμή 28 με τη βοήθεια της ενσωματωμένης συνάρτησης `random.sample()` επιλέγουμε από τον `pool[]` δύο γονείς κάθε φορά χωρίς επανατοποθέτηση. Τελικά η συνάρτηση `create_pool[]` επιστρέφει τον πίνακα `pool[]`.

Παρακάτω βλέπουμε ένα παράδειγμα εκτέλεσης της συνάρτησης `begin_tournament()` για μέγεθος πίνακα `parents1[]` ίσο με 36 άτομα και στη συνέχεια τη δημιουργία ζευγαριών από τη συνάρτηση `create_pool[]`:

```

-----ARRAY 2200 - PARENTS1 DATA-----
---
(0, array([12, 1, 2, 4, 3, 13]), 299.27754114)
(1, array([12, 1, 7, 1, 6, 13]), 5132.918)
(2, array([12, 3, 4, 10, 6, 13]), 5132.918)
(3, array([12, 1, 9, 0, 7, 13]), 5132.918)
(4, array([12, 4, 5, 0, 10, 13]), 5132.918)
(5, array([12, 2, 9, 6, 11, 13]), 5132.918)
(6, array([12, 3, 9, 0, 5, 13]), 5132.918)
(7, array([12, 4, 0, 5, 10, 13]), 5132.918)
(8, array([12, 2, 8, 1, 10, 13]), 5132.918)
(9, array([12, 3, 1, 2, 11, 13]), 5132.918)
(10, array([12, 11, 1, 11, 8, 13]), 5132.918)
(11, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(12, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(13, array([12, 7, 4, 2, 9, 13]), 316.41732835)
(14, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(15, list([12, 3, 0, 13]), 219.51275676)
(16, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(17, array([12, 3, 2, 4, 3, 13]), 306.48785539)
(18, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(19, array([12, 9, 1, 10, 6, 13]), 268.91708441)
(20, array([12, 1, 2, 10, 1, 13]), 270.39148692)
(21, array([12, 1, 2, 10, 6, 13]), 283.99804938)
(22, array([12, 3, 0, 13]), 219.51275676)
(23, array([12, 3, 2, 4, 3, 13]), 306.48785539)
(24, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(25, list([12, 2, 3, 13]), 218.55570737)
(26, list([12, 1, 2, 4, 3, 13]), 299.27754114)
(27, list([12, 3, 0, 13]), 219.51275676)
(28, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(29, array([12, 1, 2, 4, 3, 13]), 299.27754114)
(30, array([12, 9, 1, 10, 6, 13]), 268.91708441)
(31, array([12, 9, 1, 10, 6, 13]), 268.91708441)
(32, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(33, array([12, 9, 1, 10, 6, 13]), 268.91708441)
(34, array([12, 2, 3, 0, 10, 13]), 255.45356018)
(35, array([12, 9, 1, 10, 6, 13]), 268.91708441)

```

```

-----ARRAY 2500 - PARENTS2 DATA -----
----
(0, array([12, 1, 2, 4, 3, 13]), 299.27754114)
(2, array([12, 3, 4, 10, 6, 13]), 5132.918)
(4, array([12, 4, 5, 0, 10, 13]), 5132.918)
(6, array([12, 3, 9, 0, 5, 13]), 5132.918)
(8, array([12, 2, 8, 1, 10, 13]), 5132.918)
(11, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(12, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(15, list([12, 3, 0, 13]), 219.51275676)
(16, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(18, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(20, array([12, 1, 2, 10, 1, 13]), 270.39148692)
(22, array([12, 3, 0, 13]), 219.51275676)
(25, list([12, 2, 3, 13]), 218.55570737)
(27, list([12, 3, 0, 13]), 219.51275676)
(28, array([12, 9, 1, 10, 1, 13]), 255.31052195)
(30, array([12, 9, 1, 10, 6, 13]), 268.91708441)
(32, array([12, 2, 3, 0, 6, 13]), 234.1546859)
(34, array([12, 2, 3, 0, 10, 13]), 255.45356018)

----- PAIRS -----
[array([12, 2, 8, 1, 10, 13]), array([12, 3, 4, 10, 6, 13])]
[[12, 2, 3, 13], array([12, 9, 1, 10, 1, 13])]
[array([12, 2, 3, 0, 10, 13]), array([12, 9, 1, 10, 6, 13])]
[array([12, 2, 3, 0, 6, 13]), array([12, 2, 3, 0, 6, 13])]
[array([12, 2, 3, 0, 6, 13]), array([12, 9, 1, 10, 6, 13])]
[[12, 2, 3, 13], array([12, 9, 1, 10, 1, 13])]
[array([12, 3, 0, 13]), array([12, 9, 1, 10, 1, 13])]
[array([12, 9, 1, 10, 1, 13]), [12, 3, 0, 13]]

```


4.6.3 Διασταύρωση (crossover)

Αφού έχουμε στη διάθεσή μας τον πίνακα `pool[]` με τα ζευγάρια, προχωράμε στο επόμενο βήμα του γενετικού αλγορίθμου και καλούμε τη συνάρτηση `crossover()`. Στο στάδιο αυτό επιλέγουμε τυχαία ένα σημείο-κόμβο από έναν από τους δύο γονείς, οπότε το μονοπάτι που αναπαριστά κάθε γονέας χωρίζεται σε δύο τμήματα. Στη συνέχεια, ο καθένας ανταλλάσσει ένα από τα δύο κομμάτια με τον άλλο, οπότε προκύπτουν νέες λύσεις-παιδιά. Ο κώδικας παρουσιάζεται παρακάτω:

```
1 def crossover(G, array1):
2     new_childs = []
3
4     for pairs in array1:
5         # BE CAREFUL WHEN PAIRS[0] HAS ONLY 2 NODES !!!!!
6         if len(pairs[0]) > 2 and len(pairs[1]) > 2:
7
8             point = random.randrange(1, len(pairs[0]) -
9             1, 1)
10            if point < len(pairs[1]): # MEASURES SO
11            THAT NO CHILDREN WITH IRRELEVANT TARGET NODE
12
13            p3 = pairs[0][0:point].copy()
14            p4 = pairs[1][point:].copy()
15            child = np.concatenate([p3, p4])
16
17            new_childs.append(child)
18
19            print(pairs)
20            print(child)
21        else:
22            print(" CROSSOVER POINT OUT OF BOUNDS
23            !")
24            child = pairs[1]
25
26        else:
27            print("PARENTS CAN'T EXCHANGE INFORMATION")
28
29            if len(pairs[0]) <= len(pairs[1]):
30                n_pairs1 = np.array(pairs[0])
31                new_childs.append(n_pairs1)
32            else:
33                n_pairs2 = np.array(pairs[1])
34                new_childs.append(n_pairs2)
35
36            print(f"NEW CHILDREN: {new_childs}")
37
38            print("-----ARRAY 500 (VALID CHILDREN) -
39            -----")
40            f_arr500 = []
```

```
37
38     for child in new_childs:
39         print(nx.is_path(G, child))
40         if nx.is_path(G, child):
41             f_arr500.append(child)
42             print(f"CHILD {child} IS VALID")
43         else:
44             print(f"CHILD {child} IS NOT VALID")
45
46     print(f" ARRAY_500: {f_arr500} \n")
47
48     f_arr501 = []
49     for child in f_arr500:
50         f_arr501.append(child.tolist())
51
52     print(f" ARRAY_501: {f_arr501} \n")
53
54     f_c_11 = len(f_arr500)
55
56     return f_c_11, f_arr500, f_arr501
```

Η συνάρτηση δέχεται ως όρισμα τον γράφο ορατότητας (G) και τον πίνακα `pool[]`. Στη γραμμή 4 ξεκινάμε ένα βρόχο για κάθε ζευγάρι μέσα στον `pool[]` και στη γραμμή 6 λαμβάνουμε μέτρα για την περίπτωση που κάποιος γονιός αναπαριστά μονοπάτι μόνο με δύο κόμβους, τον κόμβο-πηγή και τον κόμβο-προορισμό, οπότε δεν μπορεί να πραγματοποιηθεί ανταλλαγή πληροφορίας. Στη γραμμή 8 χρησιμοποιούμε την ενσωματωμένη συνάρτηση `random.randrange()` για να διαλέξουμε ένα τυχαίο σημείο-κόμβο από τον πρώτο γονέα. Προσέχουμε ώστε τα όρια να είναι τέτοια ώστε να μην επιλεγεί ούτε ο κόμβος-πηγή ούτε ο κόμβος-προορισμού. Στη γραμμή 9 λαμβάνουμε επιπλέον μέτρα ώστε το σημείο που έχει επιλεγεί να είναι μικρότερο από το μήκος του δεύτερου γονέα. Με αυτό τον τρόπο αποφεύγεται το πρόβλημα να γεννιούνται παιδιά όπου ο κόμβος-προορισμού είναι διαφορετικός του επιθυμητού. Στη γραμμή 11 δημιουργούμε ένα αντίγραφο του πρώτου τμήματος του πρώτου γονέα και στη γραμμή 12 δημιουργούμε ένα αντίγραφο του δεύτερου τμήματος του δεύτερου γονέα. Στη γραμμή 13 χρησιμοποιούμε τη συνάρτηση `np.concatenate()` και συνενώνουμε τα δυο κομμάτια, οπότε δημιουργούμε τον πίνακα `child[]` τον οποίο και αποθηκεύουμε στον βοηθητικό πίνακα με τα νέα παιδιά `new_childs[]`. Στη γραμμή 21 αν το σημείο που έχει επιλεγεί είναι μεγαλύτερο από το μονοπάτι που αναπαριστά ο δεύτερος γονιός, τότε τοποθετούμε αυτόν στον πίνακα `child[]`. Στις γραμμές 26-31, λαμβάνουμε μέτρα ώστε αν το μήκος του πρώτου γονέα είναι μικρότερο από το μήκος του δεύτερου, τότε μετατρέπουμε τον πρώτο γονέα σε numpy array και τοποθετούμε αυτόν μέσα στον πίνακα με τα παιδιά

`new_childs[]`, αλλιώς τοποθετούμε τον δεύτερο γονέα. Επειδή δεν γνωρίζουμε αν όλα τα παιδιά που θα γεννηθούν αναπαριστούν έγκυρες λύσεις, στις γραμμές 38-41 πραγματοποιούμε αυτό τον έλεγχο χρησιμοποιώντας την ενσωματωμένη συνάρτηση `is_path()`. Όσα από τα παιδιά αποτελούν έγκυρες λύσεις τα τοποθετούμε στον πίνακα `f_arr500[]`. Στις γραμμές 48-51, μετατρέπουμε όσα παιδιά είναι μέσα στον `f_arr500[]` σε λίστα και στη γραμμή 54 αποδίδουμε στη μεταβλητή `c_11` τιμή ίση με το πλήθος των έγκυρων παιδιών. Τελικά η συνάρτηση `crossover()` επιστρέφει τη μεταβλητή `c_11`, τον `arr_500[]` και τον `arr501[]`. Παρακάτω ακολουθεί ένα παράδειγμα εκτέλεσης του παραπάνω κώδικα:

```
----- CROSSOVER -----

[array([12,  1,  4, 11,  5, 13]), array([12,  8,  7,  1,  8, 13])]
[12  1  7  1  8 13]

[array([12,  4,  3, 11,  1, 13]), array([12,  2,  9, 10,  4, 13])]
[12  4  3 10  4 13]

[array([12,  4,  2,  3,  4, 13]), array([12,  1,  9, 11,  0, 13])]
[12  4  2  3  0 13]

[array([12,  8,  7,  1,  8, 13]), array([12,  4,  3, 11,  1, 13])]
[12  8  7  1  1 13]

[array([12,  1, 11,  7,  9, 13]), array([12,  2,  9, 10,  4, 13])]
[12  2  9 10  4 13]

[array([12,  4,  2,  3,  4, 13]), array([12,  1,  4, 11,  5, 13])]
[12  1  4 11  5 13]

NEW CHILDREN: [array([12,  1,  7,  1,  8, 13]), array([12,  4,  3,
10,  4, 13]), array([12,  4,  2,  3,  0, 13]), array([12,  8,  7,
1,  1, 13]), array([12,  2,  9, 10,  4, 13]), array([12,  1,  4,
11,  5, 13])]

-----ARRAY 500 (VALID CHILDREN) -----
--
False
CHILD [12  1  7  1  8 13] IS NOT VALID
False
CHILD [12  4  3 10  4 13] IS NOT VALID

True
CHILD [12  4  2  3  0 13] IS VALID
False
CHILD [12  8  7  1  1 13] IS NOT VALID
False
CHILD [12  2  9 10  4 13] IS NOT VALID
```

```
False
CHILD [12  1  4 11  5 13] IS NOT VALID

ARRAY_500: [array([12,  4,  2,  3,  0, 13])]

ARRAY_501: [[12, 4, 2, 3, 0, 13]]

THERE ARE 1 NEW CHILDREN !!!
```

4.6.4 Υπολογισμός Τιμής Αντικειμενικής Συνάρτησης Παιδιών

Στη συνέχεια αφού έχουμε αποθηκεύσει στον πίνακα `arr500[]` τις καινούριες λύσεις-άτομα του πληθυσμού, μπορούμε να υπολογίσουμε την τιμή της αντικειμενικής συνάρτησης δηλαδή το μήκος τους. Ο κώδικας φαίνεται παρακάτω:

```
1      print("-----CALCULATE CHILDREN'S
FITNESS (BEFORE MUTATION) -----")
2          arr600 = []
3
4          for i in range(len(arr501)):
5              member_to_list = list(arr501[i])
6              print(f" {i}-st path {member_to_list} is valid
!")
7
8              coords_of_node2 = [] # ARRAY TO STORE
COORDINATES OF PATH NODES
9              for j in member_to_list:
10                 for key, value in pos1.items():
11                     if j == key:
12
13                     coords_of_node2.append(pos1.get(key))
14                     print(f" COORDS: {coords_of_node2}")
15
16                     # PLACE ALL COORDINATES IN LINESTRING
17                     line7 = LineString(coords_of_node2)
18                     print(f" Path has length: {line7.length}")
19                     # WHERE TO APPEND VALID CHILDREN'S LENGTH
20                     arr600.append(line7.length)
21
22             print("-----ARRAY 600 - CHILDREN'S
DATA (BEFORE MUTATION) -----")
23             print(f" ARRAY 600 : {arr600}")
24             for i in range(len(arr600)):
25                 print(arr600[i])
26                 A199.append(arr600[i])
27
28             print(f"NEW_A199: {A199}")
```

Στη γραμμή 2 αρχικοποιούμε το νέο πίνακα `arr600[]` μέσα τον οποίο θα αποθηκεύουμε τα μήκη των νέων λύσεων. Στις γραμμές 4-19 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσες είναι τα έγκυρα παιδιά. Καταρχήν εκτυπώνουμε τα έγκυρα να παιδιά, και στη συνέχεια με τη βοήθεια του λεξικού `pos1{}` που περιέχει όλες τις συντεταγμένες των κόμβων του γράφου τοποθετούμε στον βοηθητικό πίνακα `coords_of_node2[]` τις συντεταγμένες των κόμβων που αποτελούν το νέο μονοπάτι. Στη γραμμή 16 δημιουργούμε ένα αντικείμενο `LineString` με ορίσματα τις συντεταγμένες που μόλις βρήκαμε. Υπολογίζουμε το μήκος του αντικειμένου και το τοποθετούμε στον πίνακα `arr600[]`.

Στις γραμμές 23-25 ακολουθεί ένα σημαντικό βήμα αφού ενημερώνουμε τον πίνακα `A199[]` με τα νέα μήκη μονοπατιών που υπάρχουν μέσα στον `arr600[]`. Ακολουθεί ένα παράδειγμα εκτέλεσης του παραπάνω κώδικα για το μοναδικό παιδί που προέκυψε στην προηγούμενη ενότητα:

```
-----CALCULATE CHILDREN'S FITNESS (BEFORE
MUTATION)-----
0-st path [12, 4, 2, 3, 0, 13] is valid !
COORDS: [(-18, 24), (52, 35), (23, 36), (32, 37), (38, 46), (70,
181)]
Path has length: 258.48905563231284
-----ARRAY 600 - CHILDREN'S DATA (BEFORE
MUTATION) -----
ARRAY 600 : [258.48905563231284]
258.48905563231284

NEW_A199: [4361.588, 4361.588, 4361.588, 4361.588, 4361.588,
4361.588, 4361.588, 4361.588, 4361.588, 4361.588, 4361.588,
4361.588, 4361.588, 4361.588, 4361.588, 276.1367919771668,
4361.588, 257.749754932307, 4361.588, 4361.588, 4361.588,
4361.588, 4361.588, 4361.588, 271.2841826734718,
264.05231061335246, 276.1367919771668, 276.1367919771668,
258.48905563231284]
```

ο πίνακας **A199[]**
έχει ενημερωθεί
με το νέο μήκος
μονοπατιού

4.6.5 Μετάλλαξη (Mutation)

Αφού έχουμε υπολογίσει την τιμή της αντικειμενικής συνάρτησης των παιδιών, ξεκινά η γενετική διαδικασία της μετάλλαξης. Κατά το στάδιο αυτό δίνεται η δυνατότητα στον αλγόριθμο να εξερευνήσει το χώρο των υποψηφίων λύσεων. Λέμε τότε ότι ο πληθυσμός εγκαταλείπει περιοχές με χαμηλή τιμή αντικειμενικής συνάρτησης και αρχίζει να σκαρφαλώνει σε περιοχές με άλλα τοπικά μέγιστα. Η μετάλλαξη καθορίζεται αν θα γίνει από τη μεταβλητή `mut_f`. Η συχνότητα μετάλλαξης `mut_f` καθορίζεται από το χρήστη στην αρχή του προγράμματος. Αν ο τυχαίος αριθμός `ran_num` που παρήγαγε η συνάρτηση `random.random()` είναι μικρότερος της μεταβλητής `mut_f` τότε ξεκινά η διαδικασία της μετάλλαξης, η οποία στον αλγόριθμό μας υλοποιείται από την κλήση τριών συναρτήσεων (`g_shuffle()`, `swap()`, `deletion()`) υποβάλλοντας έτσι το νέο άτομο σε τρεις διαφορετικές μεταλλάξεις. Αν όμως ο τυχαίος αριθμός είναι μεγαλύτερος από τη μεταβλητή `mut_f` τότε θα πρέπει να διώξουμε από τον πληθυσμό τόσα τυχαία παλιά μέλη όσος είναι ο αριθμός των έγκυρων παιδιών και τελικά να τοποθετήσουμε τα καινούρια άτομα.

Παρακάτω δίνουμε τον κώδικα που μόλις περιγράψαμε έως και την κλήση των τριών συναρτήσεων, και στη συνέχεια θα παρουσιάσουμε τον κώδικα για κάθε μια από τις μεταλλάξεις σε τρεις διαφορετικές υποενότητες:

[illegible]

Στη γραμμή 3 καλούμε όπως είπαμε τη συνάρτηση `random.random()` η οποία επιστρέφει έναν τυχαίο αριθμό και τον αποδίδουμε στην τιμή της μεταβλητής `ran_num`. Στη γραμμή 6 εκτυπώνουμε τη συχνότητα μετάλλαξης, στη γραμμή 7 διενεργούμε έλεγχο μεταξύ των δύο μεταβλητών και στη γραμμή 10 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσες τα έγκυρα παιδιά (μήκος του πίνακα `arr500[]`). Στη γραμμή 11 επιλέγουμε έναν τυχαίο αριθμό μέσα στο διάστημα που αντιστοιχεί στο μήκος του πίνακα `population[]` και στη γραμμή 12 αφαιρούμε από τον πληθυσμό το παλιό μέλος με αναγνωριστικό αριθμό ίσο με τον αριθμό που επιλέχθηκε. Τοποθετούμε το παλιό μέλος μέσα στον πίνακα `discard[]` και ταυτόχρονα ενημερώνουμε τον πίνακα `A199[]` και τοποθετούμε το μήκος του παλιού μονοπατιού μέσα στον πίνακα `dis_length[]` για να έχουμε καλύτερη εποπτεία της λειτουργίας του κώδικα. Τελικά στη γραμμή 19 προσθέτουμε στον πληθυσμό το νέο μέλος. Αν κατά τον έλεγχο η `ran_num` είναι μικρότερη της `mut_f` τότε στις γραμμές 24, 25, 26 καλούμε τις τρεις συναρτήσεις `g_shuffle()`, `swap()` και `deletion()` και αναθέτουμε τα επιστρεφόμενα αποτελέσματα στους πίνακες (`arr60001`, `arr300001`), (`arr601`, `arr3001`), (`arr6001`, `arr30001`) αντίστοιχα. Ακολουθεί ένα παράδειγμα εκτέλεσης του παραπάνω κώδικα:

```
-----NEW POPULATION   NUMBER 3 -----
-----
0: [12  1 11  7  9 13]
1: [12  1  5  1  9 13]
2: [12  2  8  3  7 13]
3: [12  1 11  7  9 13]
4: [12  2  6 10  4 13]
5: [12  3 10  2 11 13]
6: [12  1  2  5  7 13]
7: [12  2  9 10  4 13]
8: [12  3  3  8 11 13]
9: [12  4  3 11  1 13]
10: [12  1  4  2  1 13]
11: [12  2  2 11  4 13]
12: [12  3  9  2  7 13]
13: [12  4  7  6  1 13]
14: [12  8  6  8  9 13]
15: [12  1  9 11  0 13]
16: [12  2  0 11  4 13]
17: [12  3 11  8  7 13]
18: [12  4  2  3  4 13]
19: [12  8  7  8  4 13]
20: [12  9  1  6  0 13]
21: [12  1  4 11  5 13]
22: [12  2  7  8  9 13]
23: [12  3 10 11  0 13]
24: [12  4 10  3  4 13]
```

παλιός
πληθυσμός 3^{ης}
γενιάς

```
25: [12  8  7  1  8 13]
26: [12  9  7 11  4 13]
27: [12 11  2  8  3 13]
```

* * * * * START NEXT GENERATION NUMBER 4 * * * *

```
ARRAY_500: [array([12,  3,  7,  6,  1, 13])]
```

```
ARRAY_501: [[12, 3, 7, 6, 1, 13]]
```

νέο παιδί
στη 4^η γενιά

THERE ARE 1 NEW CHILDREN !!!

-----CALCULATE CHILDREN'S FITNESS (BEFORE
MUTATION)-----

0-st path [12, 3, 7, 6, 1, 13] is valid !

COORDS: [(-18, 24), (32, 37), (55, 47), (66, 51), (24, 48), (70,
181)]

Path has length: 271.2841826734718

-----ARRAY 600 - CHILDREN'S DATA (BEFORE
MUTATION)-----

```
ARRAY 600 : [271.2841826734718]
```

```
DISCARDED : [array([12,  4,  7,  6,  1, 13])] -  
[277.77027668222536]
```

Αφαιρεί από τον
πληθυσμό το παλιό μέλος
με fitness 277.77

-----NEW POPULATION NUMBER 4 -----

0:	[12	1	11	7	9	13]
1:	[12	1	5	1	9	13]
2:	[12	2	8	3	7	13]
3:	[12	1	11	7	9	13]
4:	[12	2	6	10	4	13]
5:	[12	3	10	2	11	13]
6:	[12	1	2	5	7	13]
7:	[12	2	9	10	4	13]
8:	[12	3	3	8	11	13]
9:	[12	4	3	11	1	13]
10:	[12	1	4	2	1	13]
11:	[12	2	2	11	4	13]
12:	[12	3	9	2	7	13]
13:	[12	8	6	8	9	13]
14:	[12	1	9	11	0	13]
15:	[12	2	0	11	4	13]
16:	[12	3	11	8	7	13]
17:	[12	4	2	3	4	13]
18:	[12	8	7	8	4	13]
19:	[12	9	1	6	0	13]
20:	[12	1	4	11	5	13]
21:	[12	2	7	8	9	13]
22:	[12	3	10	11	0	13]
23:	[12	4	10	3	4	13]
24:	[12	8	7	1	8	13]
25:	[12	9	7	11	4	13]
26:	[12	11	2	8	3	13]
27:	[12	3	7	6	1	13]

4^η γενιά με το
νέο μέλος

-----DATA STRUCTURE_4-----

```
[ ( 0, array([12, 1, 11, 7, 9, 13]), 4361.588 )
  ( 1, array([12, 1, 5, 1, 9, 13]), 4361.588 )
  ( 2, array([12, 2, 8, 3, 7, 13]), 4361.588 )
  ( 3, array([12, 1, 11, 7, 9, 13]), 4361.588 )
  ( 4, array([12, 2, 6, 10, 4, 13]), 4361.588 )
  ( 5, array([12, 3, 10, 2, 11, 13]), 4361.588 )
  ( 6, array([12, 1, 2, 5, 7, 13]), 4361.588 )
  ( 7, array([12, 2, 9, 10, 4, 13]), 4361.588 )
  ( 8, array([12, 3, 3, 8, 11, 13]), 4361.588 )
  ( 9, array([12, 4, 3, 11, 1, 13]), 4361.588 )
  (10, array([12, 1, 4, 2, 1, 13]), 4361.588 )
  (11, array([12, 2, 2, 11, 4, 13]), 4361.588 )
  (12, array([12, 3, 9, 2, 7, 13]), 4361.588 )
  (13, array([12, 8, 6, 8, 9, 13]), 4361.588 )
  (14, array([12, 1, 9, 11, 0, 13]), 4361.588 )
  (15, array([12, 2, 0, 11, 4, 13]), 4361.588 )
  (16, array([12, 3, 11, 8, 7, 13]), 4361.588 )
  (17, array([12, 4, 2, 3, 4, 13]), 276.13679198)
  (18, array([12, 8, 7, 8, 4, 13]), 4361.588 )
  (19, array([12, 9, 1, 6, 0, 13]), 257.74975493)
  (20, array([12, 1, 4, 11, 5, 13]), 4361.588 )
  (21, array([12, 2, 7, 8, 9, 13]), 4361.588 )
  (22, array([12, 3, 10, 11, 0, 13]), 4361.588 )
  (23, array([12, 4, 10, 3, 4, 13]), 4361.588 )
  (24, array([12, 8, 7, 1, 8, 13]), 4361.588 )
  (25, array([12, 9, 7, 11, 4, 13]), 4361.588 )
  (26, array([12, 11, 2, 8, 3, 13]), 4361.588 )
  (27, array([12, 3, 7, 6, 1, 13]), 271.28418267) ]
```

Η δομή **data4** έχει
ενημερωθεί σωστά

4.6.5.1 Συνάρτηση g_shuffle()

Στην περίπτωση όπου η μεταβλητή `ran_num < mut_f`, τότε καλούνται με τη σειρά οι συναρτήσεις `g_shuffle()`, `swap()` και `deletion()`. Παρακάτω δίνουμε τον κώδικα για τη `g_shuffle()`:

[illegible]

```
35         print("-----SHUFFLE_3001 - MUTANT  
CHILDREN -----")  
36         print(f" SHUFFLE_3001:{shu_3001}")  
37         for i in range(len(shu_3001)):  
38             print(shu_3001[i])  
39         print("----- MUTANT CHILDREN'S  
FITNESS AFTER SHUFFLE-----")  
40         # MOVE 1 TAB LEFT OR NOT ??????????????????????  
41  
42         for i in range(len(shu_3001)):  
43             member_to_list = list(shu_3001[i])  
44             print(f" {i}-st path {member_to_list} is  
valid !")  
45  
46             coords_of_node3 = [] # ARRAY TO STORE  
COORDINATES OF PATH NODES  
47             for j in member_to_list:  
48                 for key, value in pos.items():  
49                     if j == key:  
50  
coords_of_node3.append(pos1.get(key))  
51             print(f" COORDS: {coords_of_node3}")  
52  
53             # PLACE ALL COORDINATES IN LINESTRING  
54             line8 = LineString(coords_of_node3)  
55             print(f" Path has length: {line8.length}")  
56             # WHERE TO APPEND VALID CHILDREN'S LENGTH  
?????????????  
57             shu_601.append(line8.length)  
58             n_arr601 = np.array(shu_601)  
59             print(f"SHUFFLE_3001 LENGTH: {len(shu_3001)}")  
60             print("===== FINISH SHUFFLE =====")  
61  
62             return shu_601, shu_3001
```

Η συνάρτηση δέχεται ως όρισμα το γράφο ορατότητας G , τον πίνακα με τα έγκυρα παιδιά `arr501[]` και το λεξικό `pos1{}` με τις συντεταγμένες. Η μετάλλαξη αναδιατάσσει τα εσωτερικά στοιχεία-κόμβους του πίνακα-παιδιού.

Στις γραμμές 2-3 αρχικοποιούμε δυο πίνακες τον `shu_601[]` στον οποίο θα αποθηκεύουμε το μήκος των μεταλλαγμένων παιδιών, και τον `shu_3001[]` όπου θα αποθηκεύουμε τα μονοπάτια που αντιστοιχούν στα μεταλλαγμένα παιδιά. Στη γραμμή 6 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσα είναι τα έγκυρα παιδιά μέσα στον `arr501[]`. Στη γραμμή 8 λαμβάνουμε μέτρα ώστε η μετάλλαξη να συμβαίνει μόνο σε όσα άτομα του πληθυσμού αποτελούνται από πέντε κόμβους και πάνω. Στη γραμμή 10 δημιουργούμε ένα αντίγραφο του παιδιού με όνομα `chi[]`, ώστε από δω και πέρα η όποια επεξεργασία να γίνεται μόνο στο αντίγραφο. Στη γραμμή 13 δημιουργούμε έναν υποπίνακα `sub_chi[]` και αφήνουμε απέξω τον αρχικό κόμβο-πηγή και τον τελικό κόμβο-προορισμού. Στη γραμμή 14 χρησιμοποιούμε την ενσωματωμένη συνάρτηση `random.shuffle()` της βιβλιοθήκης Numpy και αναδιατάσσουμε τα στοιχεία του υποπίνακα `sub_chi[]`. Στη γραμμή 15 αντιγράφουμε τον `sub_chi[]` σε νέο πίνακα `new_chi[]` και στη γραμμή 16 δημιουργούμε τον πίνακα `new_chi_1[]` ο οποίος περιέχει μόνο τον αρχικό κόμβο. Στη γραμμή 19 προσθέτουμε στον πίνακα `new_chi_1[]` και τον γείτονα του και στη γραμμή 21 προσθέτουμε στον ίδιο πίνακα και τα αναδιατεταγμένα στοιχεία του πίνακα `new_chi[]`. Τέλος στη γραμμή 23 προσθέτουμε στον `new_chi_1` και τον κόμβο-προορισμού. Στις γραμμές 32-33 ελέγχουμε αν το μεταλλαγμένο παιδί αναπαριστά έγκυρη λύση και αν ναι τότε το αποθηκεύουμε στον `shu_3001[]`. Στις γραμμές 42-57 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσα είναι τα μεταλλαγμένα παιδιά μέσα στον πίνακα `shu_3001[]` με σκοπό να βρούμε το μήκος τους. Κατά τα γνωστά χρησιμοποιούμε το λεξικό `pos1{}` και δημιουργώντας με τις συντεταγμένες των κόμβων αντικείμενα τύπου `LineString` αποθηκεύουμε τα μήκη τους στον πίνακα `shu_601[]`. Τελικά η συνάρτηση επιστρέφει τους πίνακες `shu_601[]` και `shu_3001[]`. Ακολουθεί ένα παράδειγμα του πως λειτουργεί η συνάρτηση:

```
-----MUTATE-----
True
===== START SHUFFLE =====

CHILD TO BE SHUFFLED: [55, 1, 9, 18, 56]
NEW_CHI_1 : [55]
NEW_CHI_1 WITH NEIGHBOR: [55, 1]
NEW_CHI_2 : [55, 1, 9, 18]
NEW_CHI_3 : [55, 1, 9, 18, 56]
```

```
SHUFFLED ELEMENTS : [9, 18]
NEW_CHI : [9, 18]
X-MEN : [55, 1, 9, 18, 56]
PREVIOUS CHILD : [55, 1, 9, 18, 56]
-----SHUFFLE_3001 - MUTANT CHILDREN -----
SHUFFLE_3001:[[55, 1, 9, 18, 56]]
[55, 1, 9, 18, 56]
----- MUTANT CHILDREN'S FITNESS AFTER SHUFFLE-----
-----
0-st path [55, 1, 9, 18, 56] is valid !
COORDS: [(-17, 11), (42, 39), (47, 53), (33, 54), (75, 180)]
Path has length: 227.0243667910314

CHILD TO BE SHUFFLED: [55, 1, 9, 18, 32, 56]
NEW_CHI_1 : [55]
NEW_CHI_1 WITH NEIGHBOR: [55, 1]
NEW_CHI_2 : [55, 1, 18, 9, 32]
NEW_CHI_3 : [55, 1, 18, 9, 32, 56]
SHUFFLED ELEMENTS : [18, 9, 32]
NEW_CHI : [18, 9, 32]
X-MEN : [55, 1, 18, 9, 32, 56]
PREVIOUS CHILD : [55, 1, 9, 18, 32, 56]
-----SHUFFLE_3001 - MUTANT CHILDREN -----
SHUFFLE_3001:[[55, 1, 9, 18, 56], [55, 1, 18, 9, 32, 56]]
[55, 1, 9, 18, 56]
[55, 1, 18, 9, 32, 56]
----- MUTANT CHILDREN'S FITNESS AFTER SHUFFLE-----
-----
0-st path [55, 1, 9, 18, 56] is valid !
COORDS: [(-17, 11), (42, 39), (47, 53), (33, 54), (75, 180)]
Path has length: 227.0243667910314
SHUFFLE_601 LENGTH: 2
1-st path [55, 1, 18, 9, 32, 56] is valid !
COORDS: [(-17, 11), (42, 39), (33, 54), (47, 53), (62, 96), (75, 180)]
Path has length: 227.37668214811967

SHUFFLE_3001 LENGTH: 2
```


4.6.5.2 Συνάρτηση swap()

Η συνάρτηση swap() δέχεται ως όρισμα το γράφο ορατότητας G , τον αναγνωριστικό αριθμό του κόμβου-πηγή (my_source2), τον πίνακα με τα έγκυρα παιδιά arr501[], το λεξικό pos1{} και τον πίνακα source_neighhors2[] τον οποίο επιστρέφει η συνάρτηση create_starting_pool[] και περιέχει τους αναγνωριστικούς αριθμούς των γειτόνων του κόμβου-πηγή, π.χ.:

```
SOURCE_NEIGHBORS_2: [0, 2, 3, 4, 6, 8]
```

Παρακάτω δίνουμε τον κώδικα:

```
1      def swap(G, s_n2, my_s2, array, pos):
2
3          swap_601 = []
4          discarded2 = []
5          discarded3 = []
6          swap_3001 = []
7          print("===== START SWAP =====")
8
9          for child in array: # FOR EVERY VALID KID
10             print(f"\nCHILD TO BE SWAPPED: {child}")
11             if len(child) > 2: # BE CAREFUL WHEN CHILD ARRAY
HAS ONLY 2 NODES !!!
12
13                 point2 = random.randrange(1, len(child) - 1,
14 1)
15                 dis3 = child.pop(point2)
16                 discarded2.append(dis3)
17                 print(f" DISCARDED NODE: {discarded2}")
18
19                 new_nde = random.randrange(0, my_s2, 1) #
PICK A NUMBER BETWEEN 0 AND SOURCE-1
20                 discarded3.append(new_nde)
21                 print(f" NEW NODE: {discarded3}")
22
23                 if new_nde == dis3 or new_nde in s_n2:
24                     continue
25                 else:
26                     child.insert(point2, new_nde)
27                     print(f" X-MEN : {child}")
28
29                     print(f"X-MEN: {child}")
30                 else:
31                     continue
32
33                 if nx.is_path(G, child):
34                     swap_3001.append(child)
35
36                     print("----- SWAP_3001 - MUTANT
CHILDREN -----")
37                     print(f" SWAP_3001:{swap_3001}")
```

```
37         for i in range(len(swap_3001)):
38             print(swap_3001[i])
39
40         print("----- MUTANT CHILDREN'S
FITNESS AFTER SWAP -----")
41         # MOVE 1 TAB LEFT OR NOT ??????????????????????
42
43         for i in range(len(swap_3001)):
44             member_to_list = list(swap_3001[i])
45             print(f" {i}-st path {member_to_list} is
valid !")
46
47             coords_of_node3 = [] # ARRAY TO STORE
COORDINATES OF PATH NODES
48             for j in member_to_list:
49                 for key, value in pos.items():
50                     # for key, value in pos1.items():
51
52                     if j == key:
53
54                 coords_of_node3.append(pos.get(key))
55                 print(f" COORDS: {coords_of_node3}")
56
57                 # PLACE ALL COORDINATES IN LINESTRING
58                 line8 = LineString(coords_of_node3)
59                 print(f" Path has length: {line8.length}")
60                 # WHERE TO APPEND VALID CHILDREN'S LENGTH
61                 swap_601.append(line8.length)
62                 n_arr601 = np.array(swap_601)
63                 print(f"SWAP_3001 LENGTH: {len(swap_3001)}")
64                 print("===== FINISH SWAP =====")
65                 return swap_601, swap_3001
```

Η μετάλλαξη επιλέγει τυχαία έναν κόμβο και τον αντικαθιστά με έναν άλλο μέσα από το σύνολο των κόμβων που ανήκουν στο γράφο G .

Στις γραμμές 3 και 6 αρχικοποιούμε δυο νέους πίνακες, τον `swap_601[]` στον οποίο θα αποθηκεύουμε το μήκος των μεταλλαγμένων παιδιών και τον `swap_3001[]` όπου θα αποθηκεύουμε τα μονοπάτια που αντιστοιχούν στα μεταλλαγμένα παιδιά. Στη γραμμή 9 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσα είναι τα έγκυρα παιδιά μέσα στον `arr501[]`. Στη γραμμή 11 λαμβάνουμε μέτρα ώστε η μετάλλαξη να πραγματοποιείται για μονοπάτια με πάνω από τρεις κόμβους. Στη γραμμή 13 επιλέγουμε ένα τυχαίο σημείο εκτός του αρχικού και τελικού κόμβου, στη γραμμή 14 αφαιρούμε τον επιλεγμένο κόμβο και στη γραμμή 18 επιλέγουμε ένα νέο κόμβο από το σύνολο των κόμβων του γράφου G εκτός του αρχικού και τελικού κόμβου. Τοποθετούμε το νέο κόμβο στον πίνακα `discarded3[]` για καλύτερη εποπτεία και στη γραμμή 23 πραγματοποιούμε έλεγχο για να δούμε αν ο νέος κόμβος είναι ίδιος με αυτόν που είχε αφαιρεθεί προηγουμένως ή αν ο νέος κόμβος ανήκει στους γείτονες του κόμβου-πηγή. Αν δε συμβαίνει τίποτα από τα δύο τότε τοποθετούμε τον νέο κόμβο στην κατάλληλη θέση. Στις γραμμές 32-33 πραγματοποιούμε έλεγχο αν το μεταλλαγμένο παιδί αντιστοιχεί σε έγκυρο μονοπάτι και αν ναι το τοποθετούμε στον πίνακα `swap_3001[]`. Στις γραμμές 47-58 αφού τα βρούμε τις συντεταγμένες των κόμβων του νέου μονοπατιού, υπολογίζουμε το μήκος του μονοπατιού και το τοποθετούμε στον πίνακα `swap_601[]`. Τελικά η συνάρτηση επιστρέφει τους πίνακες `swap_601[]` και `swap_3001[]`. Ακολουθεί ένα παράδειγμα που δείχνει τα αποτελέσματα της μετάλλαξης `swap`:

```
===== START SWAP =====

CHILD TO BE SWAPPED: [50, 13, 37, 51]
DISCARDED NODE: [13]
NEW NODE: [4]
X-MEN : [50, 4, 37, 51]
X-MEN: [50, 4, 37, 51]
----- SWAP_3001 - MUTANT CHILDREN -----
SWAP_3001: []
----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----
-----

CHILD TO BE SWAPPED: [50, 13, 37, 51]
DISCARDED NODE: [13, 13]
NEW NODE: [4, 41]
X-MEN : [50, 41, 37, 51]
X-MEN: [50, 41, 37, 51]
----- SWAP_3001 - MUTANT CHILDREN -----
```

```
SWAP_3001:[]
----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----
----

CHILD TO BE SWAPPED: [50, 13, 37, 51]
DISCARDED NODE: [13, 13, 37]
NEW NODE: [4, 41, 26]
X-MEN : [50, 13, 26, 51]
X-MEN: [50, 13, 26, 51]
----- SWAP_3001 - MUTANT CHILDREN -----
SWAP_3001:[[50, 13, 26, 51]]
[50, 13, 26, 51]
----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----
----

0-st path [50, 13, 26, 51] is valid !
COORDS: [(-14, 39), (-10, 55), (6, 73), (74, 183)]
Path has length: 169.89691694267293

SWAP_3001 LENGTH: 1
```

4.6.5.3 Συνάρτηση deletion()

Η συνάρτηση deletion() δέχεται ως όρισμα το γράφο G , τον πίνακα με τα έγκυρα παιδιά arr501[] και το λεξικό με τις συντεταγμένες pos1{}. Η μετάλλαξη αφαιρεί από το αρχικό παιδί έναν τυχαίο κόμβο. Παρουσιάζουμε τον κώδικα της συνάρτησης παρακάτω:

```
1 def deletion(G, array, pos):
2     del_601 = []
3     del_3001 = []
4     discarded2 = []
5     print(" ===== START DELETION =====")
6
7     for child in array: # FOR EVERY VALID KID
8         print(f"\nCHILD TO BE DELETED: {child}")
9         if len(child) > 2: # BE CAREFUL WHEN CHILD ARRAY
HAS ONLY 2 NODES !!!
10
11             point2 = random.randrange(1, len(child) - 1,
12 )
13             dis3 = child.pop(point2)
14             discarded2.append(dis3)
15             print(f" DISCARDED NODE: {discarded2}")
16             print(f"X-MEN: {child}")
17         else:
18             continue
19
20         if nx.is_path(G, child):
21             del_3001.append(child)
22
23         print("-----DELETION_3001 - MUTANT
CHILDREN -----")
24
25         print(f" DELETION_3001:{del_3001}")
26         for i in range(len(del_3001)):
27             print(del_3001[i])
28
29         print("-----ARRAY 3000 - MUTANT
CHILDREN'S FITNESS AFTER DELETION -----")
30
31         # MOVE 1 TAB LEFT OR NOT ??????????????????????
32
33         for i in range(len(del_3001)):
34             member_to_list = list(del_3001[i])
35             print(f" {i}-st path {member_to_list} is
valid !")
36
37             coords_of_node3 = [] # ARRAY TO STORE
COORDINATES OF PATH NODES
38             for j in member_to_list:
39                 for key, value in pos.items():
40                     if j == key:
```

```

39         coords_of_node3.append(pos.get(key))
40         print(f" COORDS: {coords_of_node3}")
41
42         # PLACE ALL COORDINATES IN LINESTRING
43         line8 = LineString(coords_of_node3)
44         print(f" Path has length: {line8.length}")
45         # WHERE TO APPEND VALID CHILDREN'S LENGTH
46         del_601.append(line8.length)
47         n_arr601 = np.array(del_601)
48         print(f"DELETION_601 LENGTH:
{len(del_601)}")
49         print(" ===== FINISH DELETION =====")
50         return del_601, del_3001

```

Στις γραμμές 2 και 3 αρχικοποιούμε δυο νέους πίνακες, τους `del_601[]` και `del_3001[]`. Στη γραμμή 7 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσες ο αριθμός των έγκυρων παιδιών μέσα στον πίνακα `arr_501[]`. Στη γραμμή 9 λαμβάνουμε μέτρα ώστε η μετάλλαξη να μη συμβαίνει σε μονοπάτια με δυο μόνο κόμβους δηλαδή τον αρχικό και τελικό κόμβο. Στη γραμμή 11 επιλέγουμε ένα τυχαίο σημείο και στη γραμμή 12 το αφαιρούμε από το παιδί. Στις γραμμές 19-20 τοποθετούμε το μεταλλαγμένο παιδί μέσα στον πίνακα `arr_3001[]` αν περάσει τον έλεγχο εγκυρότητας. Στις γραμμές 35-46 υπολογίζουμε το μήκος του νέου μονοπατιού και το τοποθετούμε στον πίνακα `del_601[]`. Τελικά η συνάρτηση επιστρέφει τους πίνακες `del_601[]` και `del_3001[]`. Ακολουθεί ένα παράδειγμα όπου φαίνεται το αποτέλεσμα της μετάλλαξης deletion:

```

===== START DELETION =====

CHILD TO BE DELETED: [45, 0, 12, 0, 40, 46]
DISCARDED NODE: [40]
X-MEN: [45, 0, 12, 0, 46]
-----DELETION_3001 - MUTANT CHILDREN -----
DELETION_3001:[]
-----ARRAY 3000 - MUTANT CHILDREN'S FITNESS AFTER
DELETION -----

CHILD TO BE DELETED: [45, 0, 37, 31, 46]
DISCARDED NODE: [40, 37]
X-MEN: [45, 0, 31, 46]
-----DELETION_3001 - MUTANT CHILDREN -----
DELETION_3001:[[45, 0, 31, 46]]
[45, 0, 31, 46]
-----ARRAY 3000 - MUTANT CHILDREN'S FITNESS AFTER
DELETION -----
0-st path [45, 0, 31, 46] is valid !
COORDS: [(-10, 14), (31, 55), (55, 104), (62, 181)]

```

Path has length: 189.86217464042267

DELETION_3001 LENGTH: 1

4.6.5.4 Συνάρτηση `cluster_swap2()`

Σε αυτή την υποενότητα παρουσιάζουμε μια παραλλαγή της συνάρτησης `swap()`. Εκμεταλλευόμαστε το γεγονός ότι η συνάρτηση `generatePolygon()` κατανέμει ομοιόμορφα τα εμπόδια στο επίπεδο, οπότε χρησιμοποιούμε την τεχνική συσταδοποίησης των K -μέσων (K -means) για να ομαδοποιήσουμε τους κόμβους σε συστάδες.

Κατά την τεχνική συσταδοποίησης επιλέγονται K αρχικά κέντρα βάρους όπου K μια παράμετρος ορισμένη από το χρήστη, συγκεκριμένα το πλήθος των επιθυμητών συστάδων (Tan, 2020). Κάθε σημείο που αναπαριστά έναν κόμβο του γράφου ορατότητας αποδίδεται στο πιο κοντινό κέντρο βάρους και κάθε σύνολο σημείων συνιστά μια συστάδα. Σημειώνουμε ότι το κέντρο βάρους σχεδόν ποτέ δεν αντιστοιχεί σε ένα πραγματικό σημείο δεδομένων δηλαδή σε κάποιο κόμβο του γράφου.

Είδαμε ότι κατά τη μετάλλαξη `swap` εντοπίζουμε τυχαία έναν κόμβο μέσα σε κάθε παιδί και αφού τον αφαιρέσουμε τον αντικαθιστούμε με κάποιον άλλο εκτός του κόμβου-πηγή και του κόμβου-προορισμού. Στόχος μας είναι κατά τη μετάλλαξη `cluster_swap2()` ο νέος κόμβος να επιλέγεται μέσα από ένα σύνολο «καλύτερων» κόμβων δηλαδή των κόμβων εκείνων που βρίσκονται κοντά στη νοητή γραμμή που συνδέει τον κόμβο-πηγή με τον κόμβο-προορισμού.

Για να ξεκινήσουμε τη διαδικασία συσταδοποίησης χρειαζόμαστε τις συντεταγμένες x και y των κόμβων καθώς και μια δομή για να μας πληροφορεί για το ποιοι κόμβοι βρίσκονται σε κάθε συστάδα από αυτές τις οποίες θα επιστρέψει ο αλγόριθμος K -means. Εισάγουμε λοιπόν στο πρόγραμμα τη βιβλιοθήκη `Pandas` της `Python` και δημιουργούμε ένα αντικείμενο `DataFrame` που περιέχει των αύξοντα αριθμό και τις συντεταγμένες κάθε κόμβου. Δίνουμε στη συνέχεια ένα παράδειγμα που απεικονίζει τη δομή `my_points2` για γράφο με 93 κόμβους:


```
MY_POINTS_2:
      point      x      y
0         0     17     31
1         1     32     34
2         2     42     41
3         3     34     56
4         4     20     47
..      ...    ..    ...
87        87     40    397
88        88     35    409
89        89     24    407
90        90    -12     16
91        91     83    327

[92 rows x 3 columns]
```

Στη συνέχεια εισάγουμε από τη βιβλιοθήκη Sklearn της Python το εργαλείο KMeans και προχωράμε στη συσταδοποίηση των κόμβων του γράφου. Ο αντίστοιχος κώδικας φαίνεται παρακάτω:

[illegible]

```

28 #pos1 = {}
29 for i in range(len(centers)):
30     cent_dict[i] = centers[i]
31 print("\n CENTROIDS DICTIONARY: ")
32 print(f"{cent_dict} ")
33 print("\n-----")
34 centers2 = pd.Series(cent_dict)
35 print("\nCENTROIDS SERIES: ")
36 print(f"{centers2}")
37 print("\n-----")
38 centers3 = pd.DataFrame()
39 centers3['Coordinates'] = centers2.values
40 print("\nCENTROIDS DATAFRAME: ")
41 print(f"{centers3}")

```

Στη γραμμή 1 καλούμε τη συνάρτηση KMeans() με παράμετρο ίση με τον αριθμό των συστάδων που επιθυμούμε. Επιλέξαμε για τιμή της παραμέτρου τη μεταβλητή `c_polygons` στην οποία έχει αποθηκευτεί το πλήθος των έγκυρων πολυγωνικών εμποδίων. Στη γραμμή 3 τοποθετούμε στο μοντέλο τις συντεταγμένες των κόμβων και αποθηκεύουμε το αποτέλεσμα της συσταδοποίησης στον πίνακα `y_predicted[]`. Στη γραμμή 6 δημιουργούμε μια νέα στήλη στη δομή `my_points2` η οποία περιέχει τις συστάδες που επέστρεψε ο αλγόριθμος K-means. Αν εκτυπώσουμε το αποτέλεσμα θα πάρουμε την παρακάτω εικόνα:

```

MY_POINTS_2:
   point  x    y  cluster
0      0  17   31        16
1      1  32   34        16
2      2  42   41        16
3      3  34   56        16
4      4  20   47        16
..     ... ..   ...     ...
87     87  40  397         11
88     88  35  409         11
89     89  24  407         11
90     90 -12   16          9
91     91  83  327          4

[92 rows x 4 columns]

```

Τώρα γνωρίζουμε σε ποια συστάδα βρίσκεται κάθε κόμβος. Στη γραμμή 9 βρίσκουμε τις συντεταγμένες των κεντροειδών και τις αποθηκεύουμε στον πίνακα `centers[]`. Στη γραμμή 19 κατασκευάζουμε το λεξικό `pos111{}` που περιέχει ως κλειδιά τους κόμβους και ως τιμές τον αριθμό συστάδας που ανήκει κάθε κόμβος. Αν το εκτυπώσουμε θα έχει τη παρακάτω μορφή:

NODES & CLUSTERS DICTIONARY:

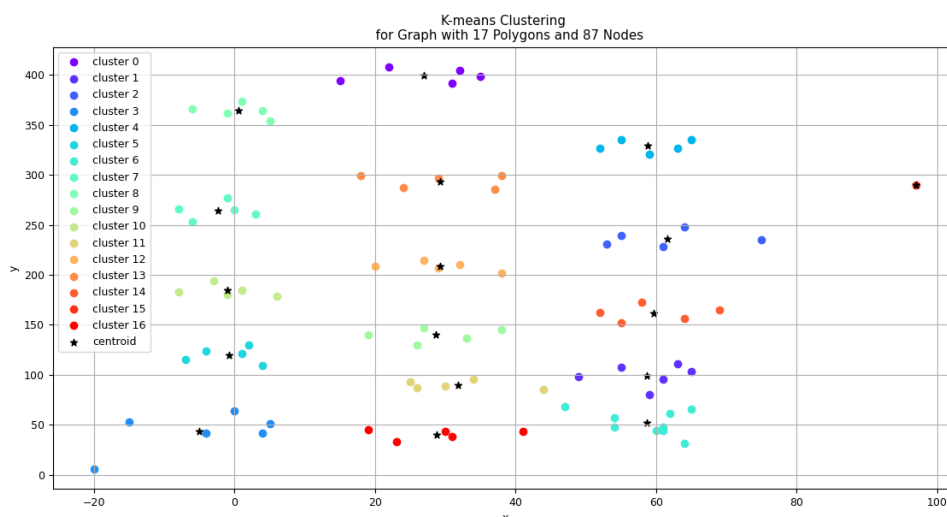
```
{0: 16, 1: 16, 2: 16, 3: 16, 4: 16, 5: 6, 6: 6, 7: 6, 8: 6, 9: 14,
10: 9, 11: 9, 12: 9, 13: 9, 14: 9, 15: 14, 16: 14, 17: 14, 18: 14,
19: 14, 20: 17, 21: 17, 22: 17, 23: 17, 24: 17, 25: 1, ...}
```

Στη γραμμή 27 κατασκευάζουμε το λεξικό `cent_dict{}` που περιέχει ως κλειδιά τα κεντροειδή και ως τιμές τις συντεταγμένες τους. Στη γραμμή 38 κατασκευάζουμε μια ακόμα δομή `DataFrame` που περιέχει τα κεντροειδή και τις συντεταγμένες τους. Η δομή για 18 κεντροειδή (άρα 18 συστάδες) απεικονίζεται παρακάτω:

CENTROIDS DATAFRAME:

	Coordinates
0	[0.19999999999999574, 265.0]
1	[30.8, 86.8]
2	[0.9999999999999964, 366.20000000000005]
3	[1.7999999999999972, 185.4]
4	[64.5, 329.16666666666663]
5	[34.166666666666664, 141.83333333333334]
6	[61.75, 40.0]
7	[60.0, 234.2]
8	[29.0, 295.6]
9	[-1.5, 43.0]
10	[57.6, 102.2]
11	[30.0, 401.2]
12	[59.0, 166.0]
13	[-0.20000000000000284, 119.8]
14	[57.0, 61.5]
15	[33.6, 209.2]
16	[29.0, 41.79999999999998]
17	[0.19999999999999574, 74.0]

Αν σχεδιάσουμε τώρα το γράφημα που προκύπτει από τη συσταδοποίηση K-means για γράφο με 17 πολυγωνικά εμπόδια και 87 κόμβους θα πάρουμε την παρακάτω εικόνα:



Εικόνα 49 Συσταδοποίηση 87 κόμβων με χρήση του εργαλείου KMeans της βιβλιοθήκης sklearn

Στην παραπάνω εικόνα φαίνονται οι 17 συστάδες με διαφορετικό χρώμα η καθεμία. Τα κεντροειδή απεικονίζονται με μαύρο χρώμα. Παρατηρούμε ότι το κεντροειδές της συστάδας 16 το οποίο βρίσκεται πάνω δεξιά συμπίπτει με τον κόμβο-προορισμό. Επίσης ο κόμβος-πηγή έχει ενσωματωθεί στη συστάδα 3. Σημειώνουμε ότι δεν χρησιμοποιήσαμε το εργαλείο MinMaxScaler της βιβλιοθήκης sklearn για να τροποποιήσουμε τα διαστήματα των τιμών των αξόνων αφού οι τιμές των συντεταγμένων x, y είναι της ίδιας τάξης μεγέθους.

Θα μπορούσαμε να ολοκληρώσουμε εδώ την παραλλαγή της μετάλλαξης `swap()` εισάγοντας στη συνάρτηση έναν ακόμη έλεγχο. Δηλαδή αν ο καινούριος κόμβος δεν βρίσκεται στη ίδια συστάδα με τον κόμβο που αφαιρέθηκε τότε τοποθέτησε τον στη θέση του παλιού κόμβου. Με αυτό τον τρόπο αναγκάζουμε τον Γενετικό Αλγόριθμο να ψάξει και σε άλλες περιοχές. Ή επίσης θα μπορούσαμε να θέσουμε ως κριτήριο για το αν θα εισαχθεί ένας κόμβος στο μονοπάτι το γεγονός ότι το κεντροειδές της συστάδας στην οποία ανήκει είναι πιο κοντά στον κόμβο-προορισμού μειώνοντας έτσι το μήκος του μονοπατιού. Ο έλεγχος αυτός όμως θα μπορούσε να γίνει υπολογίζοντας απευθείας την απόσταση νέου κόμβου-κόμβου_προορισμού οπότε ο υπολογισμός της συσταδοποίησης θα ήταν περιττός. Στόχος μας είναι χρησιμοποιώντας την ευθεία που συνδέει τον κόμβο-

πηγή με τον κόμβο-προορισμού να υπολογίσουμε τις αποστάσεις των κεντροειδών από την ευθεία αυτή και να επιλέξουμε τα κεντροειδή εκείνα και συνεπώς τις συστάδες που βρίσκονται πιο κοντά στην ευθεία.

Στη συνέχεια χωρίζουμε τις συντεταγμένες των κεντροειδών με τον παρακάτω κώδικα και κατασκευάζουμε τη δομή `centers4`:

```

1 # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> CENTROIDS
COORDINATES DATAFRAME <<<<<<<<<<<<<<<<,
2     x2_coord = []
3     y2_coord = []
4     for t in centers:
5         x2_coord.append(t[0])
6         y2_coord.append(t[1])
7
8     print(f"CENTROIDS x: {x2_coord}")
9     print(f"CENTROIDS y: {y2_coord}")
10
11     x22_coord = np.array(x2_coord).reshape(-1,1)
12     y22_coord = np.array(y2_coord).reshape(-1,1)
13
14     centers4 = pd.DataFrame()
15     centers4['x'] = x2_coord
16     centers4['y'] = y2_coord
17     print("\nCENERS4:")
18     print(f"{centers4}")

```

Η δομή centers4 είναι όπως παρακάτω:

CENTERS4:

	x	y
0	0.200000	265.000000
1	30.800000	86.800000
2	1.000000	366.200000
3	1.800000	185.400000
4	64.500000	329.166667
5	34.166667	141.833333
6	61.750000	40.000000
7	60.000000	234.200000
8	29.000000	295.600000
9	-1.500000	43.000000
10	57.600000	102.200000
11	30.000000	401.200000
12	59.000000	166.000000
13	-0.200000	119.800000
14	57.000000	61.500000
15	33.600000	209.200000
16	29.000000	41.800000
17	0.200000	74.000000

Εισάγουμε από τη βιβλιοθήκη `sklearn` το εργαλείο `LinearRegression` και προχωρούμε σύμφωνα με τον παρακάτω κώδικα:

[illegible]

```

46             best_nodes.append(key)
47
48     print("\nBEST NODES:")
49     print(f"{best_nodes}")
50

```

Στη γραμμή 2-3 υπολογίζουμε τις συντεταγμένες του κόμβου πηγή-προορισμού και στη γραμμή 10 δημιουργούμε ένα αντικείμενο `LinearRegression`. Στην γραμμή 11 αρχικοποιούμε το μοντέλο με τις συντεταγμένες που υπολογίσαμε και στη γραμμή 15 αποθηκεύουμε στον πίνακα `y_pred_2[]` τις y τιμές οι οποίες υπολογίζονται από το μοντέλο αν του δώσουμε ως είσοδο τις x τιμές των κεντροειδών. Στη γραμμή 20-22 αποθηκεύουμε στον πίνακα `difference[]` τις αποστάσεις των κεντροειδών από την ευθεία που συνδέει κόμβο-πηγή με κόμβο-προορισμό. Εισάγουμε στη δομή `centers4` μια νέα στήλη με τις αποστάσεις αυτές και παίρνουμε το παρακάτω αποτέλεσμα:

CENTERS4:

	x	y	Difference
0	0.200000	265.000000	[209.06105263157895]
1	30.800000	86.800000	[69.31368421052632]
2	1.000000	366.200000	[307.64210526315793]
3	1.800000	185.400000	[124.22315789473686]
4	64.500000	329.166667	[62.72982456140346]
5	34.166667	141.833333	[25.301754385964898]
6	61.750000	40.000000	[217.43421052631578]
7	60.000000	234.200000	[17.505263157894774]
8	29.000000	295.600000	[145.37894736842108]
9	-1.500000	43.000000	[7.373684210526314]
10	57.600000	102.200000	[141.6484210526316]
11	30.000000	401.200000	[247.7052631578947]
12	59.000000	166.000000	[82.4315789473684]
13	-0.200000	119.800000	[65.17052631578949]
14	57.000000	61.500000	[180.38421052631577]
15	33.600000	209.200000	[43.91999999999999]
16	29.000000	41.800000	[108.42105263157896]
17	0.200000	74.000000	[18.06105263157896]

Στη γραμμή 37 ταξινομούμε και αποθηκεύουμε στον πίνακα `dif2[]` τις συστάδες εκείνες των οποίων τα κεντροειδή απέχουν τη μικρότερη απόσταση από την ευθεία. Τέλος, στις γραμμές 42-46 βρίσκουμε και αποθηκεύουμε στον πίνακα `best_nodes[]` τους κόμβους οι οποίοι ανήκουν στις συστάδες αυτές. Το τελικό αποτέλεσμα για ένα γράφο με 92 κόμβους φαίνεται παρακάτω:

```
4 CENTROIDS WITH MIN DISTANCE:
```

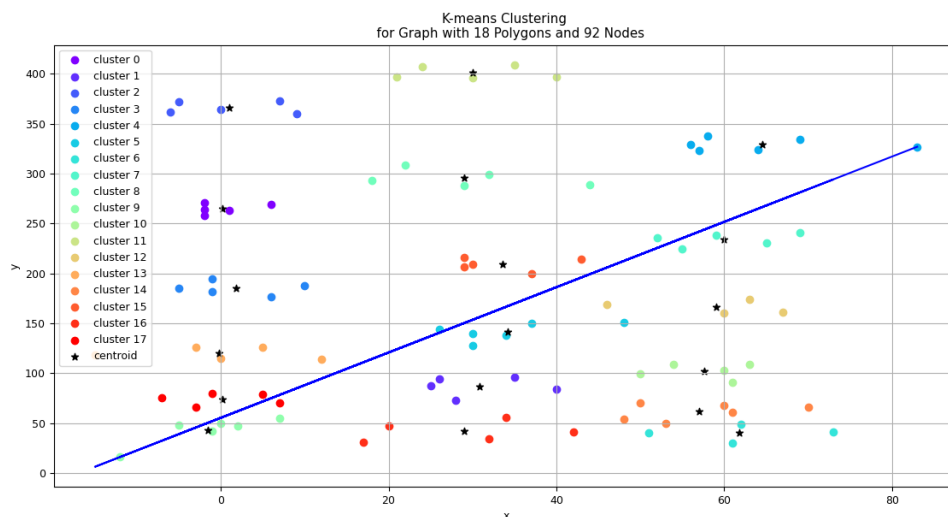
```
[[ 9]  
 [ 7]  
[17]  
 [ 5]]
```

```
===== BEST NODES =====
```

```
BEST NODES:
```

```
[10, 11, 12, 13, 14, 90, 60, 61, 62, 63, 64, 20, 21, 22, 23, 24,  
40, 41, 42, 43, 44, 48]
```

Πλέον η νέα μετάλλαξη `cluster_swap2` θα επιλέγει μόνο από αυτό τον πίνακα τυχαίους προς εισαγωγή κόμβους αποκλείοντας όλους τους απομακρυσμένους κόμβους με αποτέλεσμα να μειώνεται το μήκος του μονοπατιού. Παρακάτω φαίνεται η ευθεία που ενώνει τους κόμβους πηγή και προορισμού για τον παραπάνω γράφο με 92 κόμβους, ενώ αποτυπώνονται και οι 18 συστάδες:



Εικόνα 50 Σχεδίαση ευθείας Γραμμικής Παλινδρόμησης με χρήση του εργαλείου `LinearRegression`.

Παρατηρούμε ότι πράγματι οι συστάδες 9, 7, 17, 5 βρίσκονται πιο κοντά στην ευθεία απ' ό,τι οι υπόλοιπες, ενώ μια συστάδα συμπίπτει με τον κόμβο-προορισμού.

Μετά από αυτή τη διαδικασία και έχοντας στη διάθεση μας τον πίνακα `best_nodes[]`, προχωράμε στην εξέταση του κώδικα της συνάρτησης `cluster_swap2()`:

```
1 def cluster_swap2(G, s_n2, array, pos, pos2, array2):
2
3     swap_601 = []
4     discarded2 = []
5     discarded3 = []
6     swap_3001 = []
7     print("===== START CLUSTER-SWAP2
=====")
8
9     for child in array: # FOR EVERY VALID KID
10         print("-----")
11         print(f"\nCHILD TO BE CLUSTER_SWAPPED: {child}")
12         if len(child) > 2: # BE CAREFUL WHEN CHILD ARRAY
HAS ONLY 2 NODES !!!
13
14             point2 = random.randrange(1, len(child) - 1,
1)
15             dis3 = child.pop(point2)
16             discarded2.append(dis3)
17             print(f"\n DISCARDED NODE: {discarded2}")
18             for key , value in pos2.items():
19                 if dis3 == key:
20                     print(f"DISCARDED NODE BELONGS TO
CLUSTER: {pos2.get(key)}")
21
22             new_nde = random.choice(array2) # PICK A
NUMBER FROM BEST NODES ARRAY
23             discarded3.append(new_nde)
24             print(f"\n NEW NODE: {discarded3}")
25             for key , value in pos2.items():
26                 if new_nde == key:
27                     print(f"NEW NODE BELONGS TO
CLUSTER: {pos2.get(key)}")
28
29             if new_nde == dis3 or new_nde in s_n2 \
30                 or pos2.get(new_nde) ==
pos2.get(dis3) \
31                 or new_nde == my_source2 \
32                 or new_nde == my_target2:
33                 continue
34             else:
35                 child.insert(point2, new_nde)
36                 print(f" X-MEN : {child}")
37
38         else:
39             continue
40
41         if nx.is_path(G, child):
42             swap_3001.append(child)
43
44         print("----- SWAP_3001 - MUTANT
CHILDREN -----")
45         print(f" SWAP_3001:{swap_3001}")
46         for i in range(len(swap_3001)):
47             print(swap_3001[i])
```

```
48
49         print("----- MUTANT CHILDREN'S
FITNESS AFTER SWAP -----")
50         # MOVE 1 TAB LEFT OR NOT ??????????????????????
51
52         for i in range(len(swap_3001)):
53             member_to_list = list(swap_3001[i])
54             print(f" {i}-st path {member_to_list} is
valid !")
55
56             coords_of_node3 = [] # ARRAY TO STORE
COORDINATES OF PATH NODES
57             for j in member_to_list:
58                 for key, value in pos.items():
59                     # for key, value in pos1.items():
60
61                     if j == key:
62
63                         coords_of_node3.append(pos.get(key))
64                         print(f" COORDS: {coords_of_node3}")
65
66                         # PLACE ALL COORDINATES IN LINESTRING
67                         line8 = LineString(coords_of_node3)
68                         print(f" Path has length: {line8.length}")
69                         # WHERE TO APPEND VALID CHILDREN'S LENGTH
70                         ????????????
71                         swap_601.append(line8.length)
72                         n_arr601 = np.array(swap_601)
73                         print(f"SWAP_3001 LENGTH: {len(swap_3001)}")
74                         print("===== FINISH SWAP =====")
75                         return swap_601, swap_3001
```

Πριν ξεκινήσουμε την εξέταση του κώδικα σημειώνουμε ότι οι συγκριτικοί έλεγχοι που διενεργούμε στο Κεφάλαιο 5 χωρίζονται σε δύο κατηγορίες. Στην πρώτη, κατά το στάδιο της μετάλλαξης καλούνται σειριακά και οι τρεις συναρτήσεις `g_shuffle()`, `swap()` και `deletion()`, και στη δεύτερη κατηγορία κατά το στάδιο της μετάλλαξης καλείται μόνο η `cluster_swap2()`.

Η συνάρτηση `cluster_swap2()` δέχεται ως όρισμα το γράφο G , τον πίνακα με τους γείτονες του κόμβου-πηγή `source_neighhors2`, τον πίνακα `arr501[]` που περιέχει τα παιδιά, το λεξικό με τις συντεταγμένες των κόμβων `pos1{}`, το λεξικό `pos111{}` που περιέχει ως κλειδιά τους κόμβους και ως τιμές τη συστάδα στην οποία ανήκει κάθε κόμβος και τέλος τον πίνακα `best_nodes[]`.

Όπως και στη συνάρτηση `swap()` στη γραμμή 12 παίρνουμε μέτρα ώστε το μήκος του παιδιού να είναι μεγαλύτερο από δύο. Στις γραμμές 18-19 εξετάζουμε σε ποια συστάδα ανήκει ο κόμβος που αφαιρέθηκε. Στη γραμμή 22 επιλέγουμε έναν τυχαίο κόμβο μέσα από τον πίνακα `best_nodes[]`. Στις γραμμές 29-32 λαμβάνουμε μέτρα ώστε αν ο νέος κόμβος ανήκει στην ίδια συστάδα ή αν είναι ο κόμβος-πηγή ή προορισμού τότε ο έλεγχος προχωρεί στο επόμενο παιδί αλλιώς να εισάγει το νέο κόμβο. Λαμβάνουμε τα παραπάνω μέτρα διότι τώρα παρατηρήσαμε ότι μέσα στον πίνακα `best_nodes[]` ενδέχεται να βρίσκεται και ο κόμβος πηγής ή προορισμού αφού υπάρχει η πιθανότητα να ανήκει σε κάποια συστάδα από αυτές που έχουν επιλεγεί ως καλύτερες. Το υπόλοιπο κομμάτι κώδικα είναι ίδιο με αυτό της συνάρτησης `swap()`.

Παρακάτω δίνεται ένα παράδειγμα κλήσης της συνάρτησης `cluster_swap2()`:

```

MUTATION FREQUENCY: 0.9
-----MUTATE-----
True
===== START CLUSTER-SWAP =====
-----

CHILD TO BE CLUSTER_SWAPPED: [70, 11, 28, 50, 71]

DISCARDED NODE: [50]
DISCARDED NODE BELONGS TO CLUSTER: 9

NEW NODE: [11]
NEW NODE BELONGS TO CLUSTER: 12
-----

CHILD TO BE CLUSTER_SWAPPED: [70, 13, 28, 50, 71]

DISCARDED NODE: [50, 50]
DISCARDED NODE BELONGS TO CLUSTER: 9

NEW NODE: [11, 60]
NEW NODE BELONGS TO CLUSTER: 5
X-MEN : [70, 13, 28, 60, 71]
----- SWAP_3001 - MUTANT CHILDREN -----
SWAP_3001:[]
----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----
-----

CHILD TO BE CLUSTER_SWAPPED: [70, 11, 28, 50, 71]

DISCARDED NODE: [50, 50, 11]
DISCARDED NODE BELONGS TO CLUSTER: 12

NEW NODE: [11, 60, 13]
NEW NODE BELONGS TO CLUSTER: 12
-----

CHILD TO BE CLUSTER_SWAPPED: [70, 11, 28, 50, 71]

DISCARDED NODE: [50, 50, 11, 11]
DISCARDED NODE BELONGS TO CLUSTER: 12

NEW NODE: [11, 60, 13, 60]
NEW NODE BELONGS TO CLUSTER: 5
X-MEN : [70, 60, 28, 50, 71]
----- SWAP_3001 - MUTANT CHILDREN -----
SWAP_3001:[]
----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----
-----

CHILD TO BE CLUSTER_SWAPPED: [70, 11, 28, 50, 71]

DISCARDED NODE: [50, 50, 11, 11, 28]

```

Πίνακας new_node[]
στον οποίο
αποθηκεύονται όλοι
οι τυχαίοι νέοι κόμβοι

DISCARDED NODE BELONGS TO CLUSTER: 2

NEW NODE: [11, 60, 13, 60, 22]

NEW NODE BELONGS TO CLUSTER: 6

X-MEN : [70, 11, 22, 50, 71]

----- SWAP_3001 - MUTANT CHILDREN -----

SWAP_3001: [[70, 11, 22, 50, 71]]

[70, 11, 22, 50, 71]

----- MUTANT CHILDREN'S FITNESS AFTER SWAP -----

0-st path [70, 11, 22, 50, 71] is valid !

COORDS: [(-20, 15), (-6, 52), (7, 77), (7, 180), (77, 314)]

Path has length: 321.9200960086298

===== BEST NODES =====

BEST NODES:

[71, 20, 21, 22, 23, 24, 10, 11, 12, 13, 14, 70, 60, 61, 62, 63, 64]

Μέχρι στιγμής όλοι οι νέοι κόμβοι προέρχονται από τον πίνακα best_nodes[.].

Έγκυρο παιδί. Αποθηκεύεται στον πίνακα swap_3001[.].

4.6.6 Ενημέρωση Πίνακα population[]

Όταν τελειώσει η εκτέλεση των τριών συναρτήσεων τότε ελέγχουμε αν τουλάχιστον ένας από τους τρεις πίνακες shu_601[] ή swap_601[] ή del_601[] είναι γεμάτος. Αν είναι τότε αφαιρούμε τα παλιά μέλη και αποθηκεύουμε στους πίνακες:

- discarded4[] τόσα άτομα όσα βρίσκονται στον swap_3001[],
- discarded5[] τόσα άτομα όσα βρίσκονται στον del_3001[],
- discarded6[] τόσα άτομα όσα βρίσκονται στον shu_3001[].

Στη συνέχεια προσθέτουμε στον πληθυσμό τα καινούρια μεταλλαγμένα παιδιά. Στο παράδειγμα που ακολουθεί στον πίνακα shu_3001[] έχει προστεθεί ένα παιδί, στον πίνακα del_3001[] έχουν προστεθεί δυο παιδιά ενώ ο πίνακας swap_3001[] είναι άδειος. Τελικά προστίθενται στον πληθυσμό συνολικά τρία νέα άτομα:

```
-----MUTATE-----
True
===== START SHUFFLE =====

CHILD TO BE SHUFFLED: [12, 2, 1, 2, 5, 13]
NEW_CHI_1 : [12]
NEW_CHI_1 WITH NEIGHBOR: [12, 2]
NEW_CHI_2 : [12, 2, 2, 1, 5]
NEW_CHI_3 : [12, 2, 2, 1, 5, 13]
SHUFFLED ELEMENTS : [2, 1, 5]
NEW_CHI : [2, 1, 5]
X-MEN : [12, 2, 2, 1, 5, 13]
PREVIOUS CHILD : [12, 2, 1, 2, 5, 13]
-----SHUFFLE_3001 - MUTANT CHILDREN -----
---
SHUFFLE_3001:[]
----- MUTANT CHILDREN'S FITNESS AFTER SHUFFLE---
-----

CHILD TO BE SHUFFLED: [12, 2, 1, 2, 8, 13]
NEW_CHI_1 : [12]
NEW_CHI_1 WITH NEIGHBOR: [12, 2]
NEW_CHI_2 : [12, 2, 1, 2, 8]
NEW_CHI_3 : [12, 2, 1, 2, 8, 13]
SHUFFLED ELEMENTS : [1, 2, 8]
NEW_CHI : [1, 2, 8]
X-MEN : [12, 2, 1, 2, 8, 13]
PREVIOUS CHILD : [12, 2, 1, 2, 8, 13]
-----SHUFFLE_3001 - MUTANT CHILDREN -----
---
SHUFFLE_3001:[[12, 2, 1, 2, 8, 13]]
[12, 2, 1, 2, 8, 13]
```

```

----- MUTANT CHILDREN'S FITNESS AFTER SHUFFLE---
-----
0-st path [12, 2, 1, 2, 8, 13] is valid !
COORDS: [(-16, 15), (29, 35), (24, 41), (29, 35), (7, 45),
(68, 177)]
Path has length: 234.44408466027005

CHILD TO BE SHUFFLED: [12, 2, 1, 9, 10, 13]
NEW_CHI_1 : [12]
NEW_CHI_1 WITH NEIGHBOR: [12, 2]
NEW_CHI_2 : [12, 2, 9, 1, 10]
NEW_CHI_3 : [12, 2, 9, 1, 10, 13]
SHUFFLED ELEMENTS : [9, 1, 10]
NEW_CHI : [9, 1, 10]
X-MEN : [12, 2, 9, 1, 10, 13]
PREVIOUS CHILD : [12, 2, 1, 9, 10, 13]
-----SHUFFLE_3001 - MUTANT CHILDREN -----
----
SHUFFLE_3001:[[12, 2, 1, 2, 8, 13]]
[12, 2, 1, 2, 8, 13]
----- MUTANT CHILDREN'S FITNESS AFTER SHUFFLE---
-----
0-st path [12, 2, 1, 2, 8, 13] is valid !
COORDS: [(-16, 15), (29, 35), (24, 41), (29, 35), (7, 45),
(68, 177)]
Path has length: 234.44408466027005
SHUFFLE_3001 LENGTH: 1
===== FINISH SHUFFLE =====
===== START SWAP =====

CHILD TO BE SWAPPED: [12, 2, 1, 2, 5, 13]
DISCARDED NODE: [2]
NEW NODE: [3]

CHILD TO BE SWAPPED: [12, 2, 1, 2, 8, 13]
DISCARDED NODE: [2, 1]
NEW NODE: [3, 1]

CHILD TO BE SWAPPED: [12, 2, 1, 9, 10, 13]
DISCARDED NODE: [2, 1, 2]
NEW NODE: [3, 1, 3]
SWAP_3001 LENGTH: 0
===== FINISH SWAP =====
===== START DELETION =====

CHILD TO BE DELETED: [12, 2, 1, 5, 13]
DISCARDED NODE: [5]
X-MEN: [12, 2, 1, 13]
-----DELETION_3001 - MUTANT CHILDREN -----
----
DELETION_3001:[[12, 2, 1, 13]]
[12, 2, 1, 13]
-----ARRAY 3000 - MUTANT CHILDREN'S FITNESS
AFTER DELETION -----
0-st path [12, 2, 1, 13] is valid !

```



```

COORDS: [(-16, 15), (29, 35), (24, 41), (68, 177)]
Path has length: 199.99508576653227

CHILD TO BE DELETED: [12, 2, 2, 8, 13]
DISCARDED NODE: [5, 2]
X-MEN: [12, 2, 8, 13]
-----DELETION_3001 - MUTANT CHILDREN -----
----
    DELETION_3001:[[12, 2, 1, 13], [12, 2, 8, 13]]
    [12, 2, 1, 13]
    [12, 2, 8, 13]
    -----ARRAY 3000 - MUTANT CHILDREN'S FITNESS
AFTER DELETION -----
    0-st path [12, 2, 1, 13] is valid !
    COORDS: [(-16, 15), (29, 35), (24, 41), (68, 177)]
    Path has length: 199.99508576653227
    1-st path [12, 2, 8, 13] is valid !
    COORDS: [(-16, 15), (29, 35), (7, 45), (68, 177)]
    Path has length: 218.82358530845673

CHILD TO BE DELETED: [12, 1, 9, 10, 13]
DISCARDED NODE: [5, 2, 1]
X-MEN: [12, 9, 10, 13]
-----DELETION_3001 - MUTANT CHILDREN -----
----
    DELETION_3001:[[12, 2, 1, 13], [12, 2, 8, 13]]
    [12, 2, 1, 13]
    [12, 2, 8, 13]
    -----ARRAY 3000 - MUTANT CHILDREN'S FITNESS
AFTER DELETION -----
    0-st path [12, 2, 1, 13] is valid !
    COORDS: [(-16, 15), (29, 35), (24, 41), (68, 177)]
    Path has length: 199.99508576653227
    1-st path [12, 2, 8, 13] is valid !
    COORDS: [(-16, 15), (29, 35), (7, 45), (68, 177)]
    Path has length: 218.82358530845673
    DELETION_3001 LENGTH: 2
    ===== FINISH DELETION =====
    ----- END OF MUTATION -----
--

3001 SWAP_ARRAY NOT EMPTY: False

30.001 DELETION_ARRAY NOT EMPTY: True

300.001 SHUFFLE_ARRAY NOT EMPTY: True
30.000 NOT EMPTY: True
DISCARDED5 : [array([12, 2, 1, 9, 10, 13]), array([12,
1, 9, 2, 1, 13])]

300.000 NOT EMPTY: True
DISCARDED6 : [array([12, 1, 5, 8, 10, 13])]

```

-----NEW POPULATION NUMBER 13 -----

```

0: [12 0 10 2 5 13]
1: [12 0 1 7 6 13]
2: [12 0 7 0 5 13]
3: [12 2 11 4 9 13]
4: [12 0 5 1 8 13]
5: [12 2 11 6 5 13]
6: [12 3 9 8 1 13]
7: [12 0 7 0 6 13]
8: [12 3 0 6 2 13]
9: [12 5 8 10 2 13]
10: [12 3 4 2 8 13]
11: [12 0 9 6 5 13]
12: [12 3 10 6 3 13]
13: [12 5 5 4 11 13]
14: [12 6 5 9 11 13]
15: [12 8 6 8 10 13]
16: [12 0 6 5 7 13]
17: [12 5 7 6 9 13]
18: [12 6 5 10 7 13]
19: [12 3 4 0 8 13]
20: [12 6 10 0 8 13]
21: [12 1 9 4 7 13]
22: [12 1 9 4 7 13]
23: [12 1 0 11 1 13]
24: [12 2 1 9 10 13]
25: [12 2 1 9 8 13]
26: [12 5 0 11 1 13]
27: [12 2 1 9 8 13]
28: [12 2 1 9 8 13]
29: [12 2 1 9 8 13]
30: [12 1 9 4 7 13]
31: [12 1 9 2 8 13]
32: [12 1 9 4 7 13]
33: [12 2 1 9 8 13]
34: [12 1 9 2 1 13]
35: [12 2 1 2 5 13]
36: [12 2 1 9 8 13]
37: [12 2 1 9 8 13]
38: [12 2 1 9 8 13]
39: [12 2 1 9 10 13]
40: [12 2 1 9 8 13]
41: [12 2 9 4 7 13]
42: [12, 2, 1, 13]
43: [12, 2, 8, 13]
44: [12, 2, 1, 2, 8, 13]

```

Προστίθενται στον
πληθυσμό τρία νέα
άτομα .

Εάν παρατηρήσουμε προσεκτικά το παραπάνω παράδειγμα θα δούμε ότι όταν καλείται η συνάρτηση `swar()` για το παιδί `[12, 2, 1, 2, 13]`, αφαιρείται ο κόμβος `[2]`, αλλά όταν προστίθεται ο κόμβος `[3]` δεν οδηγεί σε έγκυρο παιδί. Το πρόγραμμα όμως κρατάει στη

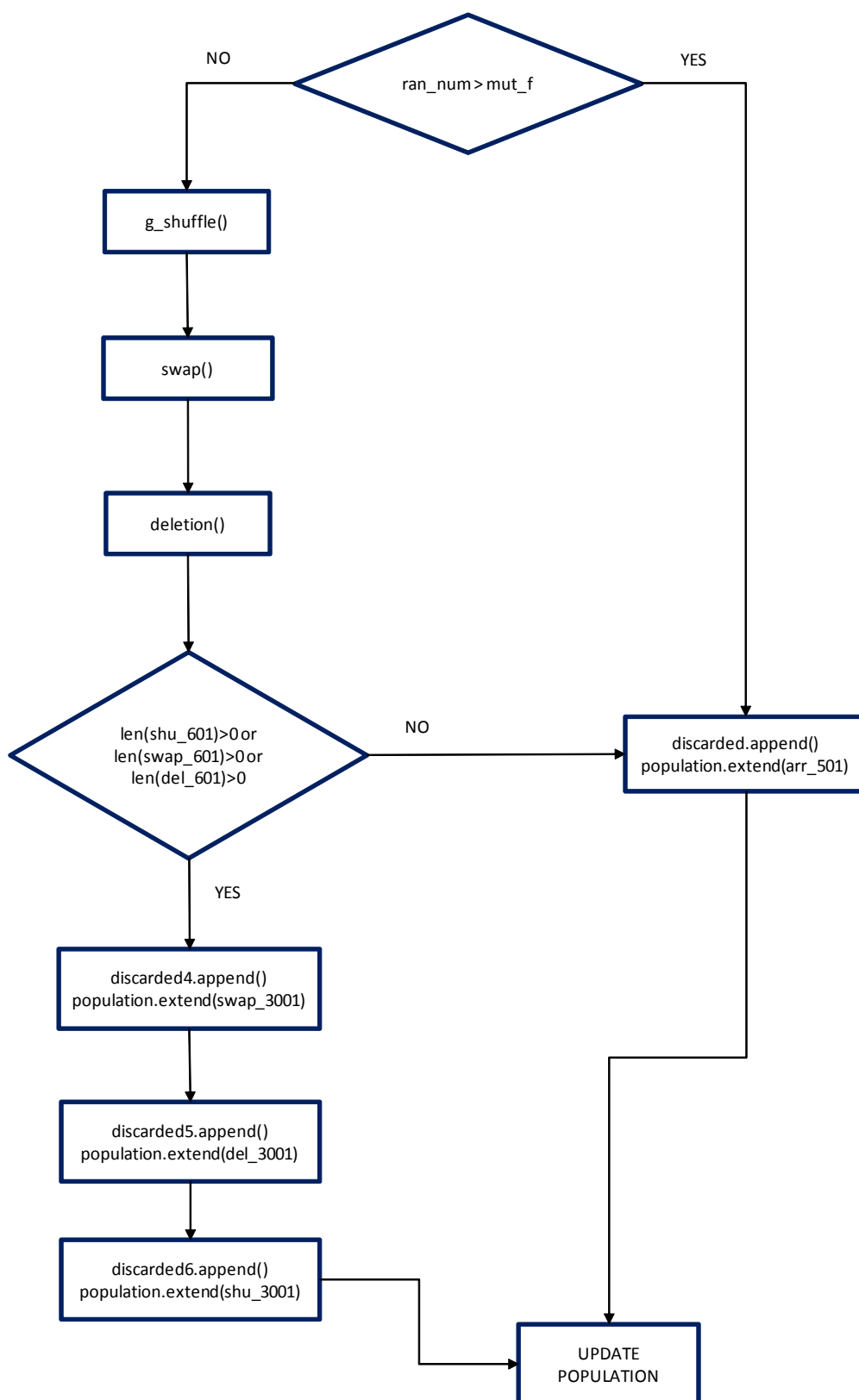
μνήμη το μεταλλαγμένο παιδί που έχει απομείνει δηλαδή το [12, 2, 1, 5, 13] και η συνάρτηση `deletion()` που ξεκινά αμέσως μετά εφαρμόζεται τελικά πάνω σε αυτό. Υπάρχει επομένως το ενδεχόμενο ενώ κάποιος από τους τρεις πίνακες `shu_601[]`, `swap_601[]`, `del_601[]` είναι γεμάτος, τελικά οι τρεις συνεχόμενες μεταλλάξεις να μην οδηγήσουν σε έγκυρο μονοπάτι. Σε αυτή την περίπτωση, ο αλγόριθμος προσθέτει στον πληθυσμό το μη έγκυρο μονοπάτι με τιμή `fitness function` ίση με `max-weight`.

Μια ακόμη περίπτωση είναι αυτή όπου μετά τις μεταλλάξεις και οι τρεις πίνακες είναι άδειοι. Τότε ο αλγόριθμος προσθέτει στον πληθυσμό το αρχικό έγκυρο παιδί που βρίσκεται μέσα στον πίνακα `arr_501[]`.

Τέλος αν το πρόγραμμα δεν μπει καθόλου μέσα στο βρόχο των μεταλλάξεων τότε τοποθετεί στον πληθυσμό πάλι το αρχικό έγκυρο παιδί του πίνακα `arr_501[]`. Ακολουθεί το αντίστοιχο κομμάτι κώδικα και το αντίστοιχο διάγραμμα ροής:

```
1         if len(arr601) != 0 or len(arr6001) != 0 or
len(arr60001) != 0:
2
3             print(f"\n3001 SWAP_ARRAY NOT EMPTY:
{len(arr3001) != 0}")
4             print(f"\n30.001 DELETION_ARRAY NOT EMPTY:
{len(arr30001) != 0}")
5             print(f"\n300.001 SHUFFLE_ARRAY NOT EMPTY:
{len(arr300001) != 0}")
6
7             if len(arr601) != 0:
8                 num_of_swap += 1
9
10                discarded4 = []
11                for i in range(len(arr3001)): # BGALE
TOSA PAIDIA OSA EINAI MESA STON ARRAY 3001 !!!
12                    # IF NO MUTATION HAS TAKEN PLACE
THEN ARRAY-3001 IS EMPTY
13                    # SO POPULATION IS UNCHANGED
14                    point = random.randrange(1,
len(population) - 1, 1)
15                    dis = population.pop(point)
16                    discarded4.append(dis)
17                    print(f" DISCARDED4 : {discarded4}")
18                    print(f" POPULATION : {population}")
# POPULATION MINUS DISCARDED MEMBERS
19
20                population.extend(arr3001) # EXTEND
WITH NEW ARRAY 3001 OF MUTANT CHILDREN !!!
21
22                if len(arr6001) != 0:
23                    num_of_del += 1
24                    print(f" 30.000 NOT EMPTY:
{len(arr30001) != 0}")
25
26                discarded5 = []
27                for i in range(len(arr30001)): #
BGALE TOSA PAIDIA OSA EINAI MESA STON ARRAY 3001 !!!
28                    # IF NO MUTATION HAS TAKEN PLACE
THEN ARRAY-3001 IS EMPTY
29                    # SO POPULATION IS UNCHANGED
30                    point = random.randrange(1,
len(population) - 1, 1)
31                    dis = population.pop(point)
32                    discarded5.append(dis)
33                    print(f" DISCARDED5 : {discarded5}")
34                    print(f" POPULATION : {population}")
# POPULATION MINUS DISCARDED MEMBERS
35                population.extend(arr30001) # EXTEND
WITH NEW ARRAY 30.001 OF MUTANT CHILDREN !!!
36
37                if len(arr60001) != 0:
38                    num_of_shuffle += 1
39                    print(f" 300.000 NOT EMPTY:
{len(arr300001) != 0}")
```

```
40
41         discarded6 = []
42         for i in range(len(arr300001)): #
BGALE TOSA PAIDIA OSA EINAI MESA STON ARRAY 3001 !!!
43             # IF NO MUTATION HAS TAKEN PLACE
THEN ARRAY-3001 IS EMPTY
44             # SO POPULATION IS UNCHANGED
45             point = random.randrange(1,
len(population) - 1, 1)
46             dis = population.pop(point)
47             discarded6.append(dis)
48             print(f" DISCARDED6 : {discarded6}")
49             print(f" POPULATION : {population}")
# POPULATION MINUS DISCARDED MEMBERS
50             population.extend(arr300001) # EXTEND
WITH NEW ARRAY 300.001 OF MUTANT CHILDREN !!!
51         else:
52             discarded = []
53             dis_length = []
54             for i in range(len(arr500)):
55                 point =
random.randrange(len(population) - 1)
56                 dis = population.pop(point)
57                 discarded.append(dis)
58                 dis2 = A199.pop(point)
59                 dis_length.append(dis2)
60
61             print(f" DISCARDED : {discarded} -
{dis_length}")
62             print(f" POPULATION : {population}")
63             population.extend(arr500)
64
```



Εικόνα 51 Διάγραμμα ροής στο στάδιο της μετάλλαξης

4.6.7 Υπολογισμός Μέσης Τιμής Αντικειμενικής Συνάρτησης

Αφού έχει ενημερωθεί ο πίνακας `population[]`, υπολογίζουμε τη μέση τιμή της αντικειμενικής συνάρτησης για τα μέλη της τρέχουσας γενιάς. Θα χρησιμοποιήσουμε τον πίνακα `A199[]`, όπου αποθηκεύουμε τα μήκη των μονοπατιών καθώς και τον πίνακα `ap700[]`, τον οποίο συναντήσαμε και στην Ενότητα 4.5, και στον οποίο αποθηκεύουμε κάθε φορά τη μέση τιμή κάθε γενιάς.

Δίνουμε παρακάτω το αντίστοιχο κομμάτι κώδικα:

```
1     print(f"-----NEW POPULATION  NUMBER {k +
1} -----")
2         for i in range(len(population)):
3             print(f" {i}: {population[i]}")
4
5         print("-----NEW MEMBERS DATA -----
-----")
6         A199.clear()
7
8
9
10        for i in range(len(population)):
11            if nx.is_path(G, population[i]):
12
13                member_to_list3 = list(population[i])
14                print(f" {i + 1}-st path {member_to_list3}
is valid !")
15
16                coords_of_node5 = [] # ARRAY TO STORE
COORDINATES OF PATH NODES
17                for j in member_to_list3:
18                    for key, value in pos1.items():
19                        if j == key:
20
21                    coords_of_node5.append(pos1.get(key))
22                    print(f" COORDS: {coords_of_node5}")
23
24                    # PLACE ALL COORDINATES IN LINESTRING
25                    line9 = LineString(coords_of_node5)
26                    print(f" Path has length: {line9.length}")
27                    A199.append(line9.length)
28
29                else:
30                    print(f"Path not valid...")
31                    A199.append(max_weight)
32
33        print(f"----- A199 = NUMBER {k + 1}
GENERATION'S FITNESS-----")
34
35        for i in range(len(A199)):
36            print(f" {i}: {A199[i]}")
```

```
36
37     athroisma = 0
38     for i in A199:
39         athroisma = athroisma + i
40     print(f"\n-----MESOS OROS FITNESS OF
NUMBER {k + 1} GENERATION-----")
41     print(f" ATHROISMA VALUES: {athroisma}")
42     mesos_oros = round((athroisma / len(population)), 2) #
BE CAREFUL TO DIVIDE WITH THE CORRECT CONSTANT !!
43     arr700.append(mesos_oros)
44     print(f" MESOS OROS : {mesos_oros} ")
45     k += 1
46     if k == 40: # ***** THIRD
CHECKPOINT *****
47         print(" !!!!!!! FINISH !!!!!!!")
48         break
```

Στη γραμμή 6 αδειάζουμε τον πίνακα A199[] για να δεχτεί τις νέες τιμές της αντικειμενικής συνάρτησης και στη γραμμή 10 ξεκινάμε ένα βρόχο με τόσες επαναλήψεις όσες τα άτομα που περιέχει ο πίνακας population[]. Στις γραμμές 11-20 χρησιμοποιούμε τη συνάρτηση is_path() και το λεξικό pos1{} για να βρούμε τις συντεταγμένες κάθε κόμβου μέσα σε κάθε μονοπάτι και στις γραμμές 24-30 ενημερώνουμε τον A199[] με τα νέα μήκη μονοπατιών. Στη γραμμή 37 αρχικοποιούμε τη μεταβλητή athroisma η οποία κρατάει κάθε φορά το άθροισμα όλων των μηκών μέσα στον A199[] και στη γραμμή 42 υπολογίζουμε το μέσο όρο διαιρώντας με το μήκος του πίνακα population[], το οποίο όπως είπαμε διατηρείται σταθερό σε κάθε γενιά ανεξάρτητα από πόσα παιδιά έχουν γεννηθεί ή από πόσα έχουν επιζήσει των μεταλλάξεων. Στη γραμμή 43 αποθηκεύουμε το μέσο όρο για την τρέχουσα γενιά στον πίνακα arr700[] και στη γραμμή 45 προχωράμε στην επόμενη γενιά. Στη γραμμή 46 έχουμε τον τελικό έλεγχο αν η μεταβλητή k έχει φτάσει στον επιθυμητό αριθμό γενιών. Αν ναι, τότε βγαίνουμε από τον κύριο βρόχο.

4.7 Μέρος Έκτο - Έξοδος

Αφού έχουμε αποθηκεύσει όλες τις μέσες τιμές της αντικειμενικής συνάρτησης μέσα στον πίνακα `avg700[]` μπορούμε να σχεδιάσουμε το διάγραμμα Mean Fitness Value/Generations. Στο διάγραμμα απεικονίζονται επίσης τα μήκη των παρακάτω πινάκων αλλά και οι μεταβλητές:

- population[],
- parents1[] = πλήθος αρχικών γονιών,
- winners = parents2[],
- pool[] = πλήθος ζευγαριών,
- mut_f = συχνότητα μετάλλαξης,
- Generations = k,
- mean_distance = αρχική μέση τιμή αντικειμενικής συνάρτησης,
- num_of_mut = πλήθος όλων των μεταλλάξεων,
- num_of_del/num_of_swap/num_of_shuffle = πλήθος μεταλλάξεων τύπου deletion/swap/shuffle.

Σχεδιάζουμε επίσης το γράφο G που δημιουργείται με τις κορυφές των πολυγωνικών εμποδίων, τη συντομότερη διαδρομή, καθώς και το μήκος της συντομότερης διαδρομής που βρίσκει ο γενετικός αλγόριθμος.

Τέλος κατασκευάζουμε ένα αντίγραφο του γράφου G , τον ονομάζουμε γράφο D και σχεδιάζουμε τη συντομότερη διαδρομή που βρίσκει ο αλγόριθμός Dijkstra για τον γράφο με βάρη D . Στον τίτλο του διαγράμματος εμφανίζουμε και το αντίστοιχο μήκος για σύγκριση.

Ακολουθεί ο αντίστοιχος κώδικας:

```

1    num_of_mut += num_of_shuffle + num_of_del + num_of_swap
2    print("\n ----- VALUES IN EACH GENERATION -----
\n")
3    print(f"ARRAY 700: {arr700}")
4    for i in range(len(arr700)):
5        print(f"{i}-th generation: {arr700[i]}")
6
7    print("\n")
8    print(f"\n * * * * * NUMBER OF MUTATIONS:
{num_of_mut} " + ' * ' * 30)
9    print("===== FITTEST PATH
=====")
10
11   # ARRAY shortest STORES SHORTEST ROUTE
12   shortest_array = []
13   A201 = np.array(A199)
14   print(f"A201: {A201}")

```

```

15
16     mean_value = np.mean(A201)
17
18     print(f" MEAN_1: {mean_value}")
19
20     # CHECK IF FITNESS HASN'T CHANGE AT ALL...
21     if max_weight-10 < mean_value < max_weight + 10:
22         print("FITNESS DIDN'T CHANGE !")
23         shortest_array = nx.dijkstra_path(G, my_source2,
my_target2) # IF PATH IS NOT VALID IT WILL PLOT AFTER ALL !!
24         print(f" SHORTEST ROUTE  IS:{shortest_array}")
25
26     # IF IT HAS CHANGE THEN FIND MINIMUM VALUE...
27     else:
28         shortest = round(np.min(A201), 3)
29         print(f" SHORTEST LENGTH:  {shortest}")
30         shortest_array = data4[data4['total_fitness'] ==
min(data4['total_fitness'])]['path']
31         print(f" SHORTEST_ARRAY: {shortest_array}")
32
33         print(f" GA's ROUTE  IS:{shortest_array[0]}")
34
35     # ***** PLOT MEAN FITNESS VALUE
*****
36
37     # xs = [x for x in range(k + 1)] # BE CAREFUL OF RANGE WHEN
USE "FOR" COMMAND!!
38     xs = [x for x in range(k + 1)] # COMMAND WHEN YOU USE
"WHILE" LOOP
39     ys = arr700
40
41     fig1 = plt.figure(1)
42     ax = plt.axes()
43
44     plt.rcParams.update({'font.size': 9})
45     plt.title(f'*POPULATION={len(population)}
*PARENTS={len(parents1)} *WINNERS={len(arr2700)} *POOL={len(pool)}
\n'
46             f'*MUTATION FREQUENCY={mut_f} *GENERATIONS={k}
*INITIAL FITNESS={round(mean_distance, 2)} \n '
47             f'*MUTATIONS={num_of_mut}
*DELETION={num_of_del} *SWAP={num_of_swap}
*SHUFFLE={num_of_shuffle}')
48
49
50     plt.xlabel("Generations")
51     plt.ylabel("Mean Fitness Value")
52     plt.plot(xs, ys)
53     # plt.ylim = (0, 400)
54     # plt.xlim = (0, 100)
55     plt.grid(True)
56     finish = round(time.perf_counter(),1)
57     finish3 = round((finish-finish2),1)
58     # ***** PLOT ROUTE
*****

```

[illegible]

```

100         for j in range(len(my_matrix2)):
101             if my_matrix2[i][j] != 0:
102                 D.add_edge(i, j, weight=my_matrix2[i][j])
103
104
105     shortest_array2 = nx.dijkstra_path(D, my_source2,
my_target2)
106     length_d = round(nx.dijkstra_path_length(D, my_source2,
my_target2), 3)
107     # IF PATH IS NOT VALID IT WILL PLOT AFTER ALL !!
108     print(f" DIJKSTRA'S ROUTE: {shortest_array2}")
109
110     fig3 = plt.figure(3)
111
112     fixedPos = pos1
113     fixed_nodes = fixedPos.keys()
114     pos = nx.spring_layout(D, pos=fixedPos, fixed=fixed_nodes)
115     nx.draw_networkx(D, pos, node_color='y', node_size=100,
width=0.3)
116
117     nx.draw_networkx_labels(D, pos, )
118
119     path_11 = shortest_array2
120     path_edges11 = zip(path_11, path_11[1:])
121     path_edges11 = set(path_edges11)
122     nx.draw_networkx_nodes(D, pos, nodelist=path_11,
node_color='red')
123     nx.draw_networkx_edges(D, pos, edgelist=path_edges11,
edge_color='red', width=5)
124     plt.title(f"*START={my_source2}    *FINISH={my_target2}
*CUTOFF={num_of_seg + 2} \n"
125             f" *NUMBER OF NODES={len(my_matrix2)}
*DIJKSTRA'S ROUTE WITH WEIGHTS: {shortest_array2} \n"
126             f" *DIJKSTRA'S DISTANCE: {length_d}")
127
128
129     for i in range(len(polys2)):
130         plt.plot(*polys2[i].exterior.xy)
131         xpoly1, ypoly1 = polys2[i].exterior.xy
132         plt.fill(xpoly1, ypoly1, alpha=0.5, fc='#0000')
133
134     print("\n Finish running after:" ,finish2)
135     print("\n Finish running after:" ,finish)
136     print("\n Finish running after:" ,finish3)
137     print(f"\n MUTATION FREQUENCY: {mut_f}")
138     print(f"\n CUTOFF: {num_of_seg}")
139     print(f"\n DIJKSTRA'S DISTANCE: {length_d}")
140     print(f"\n GA's DISTANCE: {shortest}")
141     plt.show()

```

Στη γραμμή 1 θέτουμε τη μεταβλητή `num_of_mut` ίση με το άθροισμα των τιμών των μεταβλητών που αντιστοιχούν στο πλήθος καθενός από τα τρία είδη μετάλλαξης και στη γραμμή 4-5 εκτυπώνουμε το περιεχόμενο του πίνακα `arr700[]`. Στη γραμμή 13

μετατρέπουμε τον πίνακα A199[], ο οποίος περιέχει τώρα τις τιμές των αντικειμενικών συναρτήσεων των ατόμων της τελευταίας γενιάς σε numpy array. Μέσα στον A199[] βρίσκεται πλέον η καλύτερη λύση που έχει βρει ο αλγόριθμος. Στη γραμμή 16 βρίσκουμε το μέσο όρο των τιμών του A199[] και τον αποδίδουμε στη μεταβλητή mean_value. Στις γραμμές 21-23 λαμβάνουμε μέτρα έτσι ώστε όταν η μεταβολή της αρχικής μέσης τιμής της αντικειμενικής συνάρτησης είναι πολύ μικρή, να σχεδιάζεται το μονοπάτι που επιστρέφει η ενσωματωμένη συνάρτηση dijkstra_path(), να καταχωρείται στον πίνακα sortest_array[] και να εκτυπώνεται τελικά το αντίστοιχο μήνυμα.

Σε διαφορετική περίπτωση, στη γραμμή 28 βρίσκουμε το ελάχιστο μήκος μέσα στον A199[] με τη βοήθεια της συνάρτησης np.min() και στη γραμμή 30 βρίσκουμε το μονοπάτι με τη βοήθεια της δομής δεδομένων data4 []. Στη γραμμή 31 εκτυπώνουμε όλο τον πίνακα A199[] με τις μέσες τιμές και στη γραμμή 33 επιλέγουμε την 1^η από τις τιμές αυτές.

Στις γραμμές 38-39 δίνουμε τις κατάλληλες τιμές στον άξονα x: 'Generations' και y: 'Mean Fitness Value'. Πιο συγκεκριμένα, στον άξονα x δίνεται το πλήθος των επαναλήψεων του κύριου βρόχου και στον άξονα y οι μέσες τιμές της αντικειμενικής συνάρτησης που βρίσκονται μέσα στον arr700[]. Στις γραμμές 41-55 χρησιμοποιούμε τη συλλογή συναρτήσεων matplotlib.pyplot για να σχεδιάσουμε το διάγραμμα 'Mean Fitness Value/Generations'. Δημιουργούμε ένα αντικείμενο με όνομα 'fig1', ενώ στις γραμμές 56-57 χρησιμοποιούμε δυο μεταβλητές τις finish και finish3 για να χρονομετρούμε την επίδοση του αλγορίθμου. Η μεταβλητή finish3 ισούται με την τιμή (finish-finish2), όπου η finish2 μετράει το χρόνο που χρειάζεται ο χρήστης για να πληκτρολογήσει στην αρχή του προγράμματος το επιθυμητό πλήθος των κορυφών (v) κάθε πολυγώνου και το επιθυμητό πλήθος των εμποδίων (num_of_obstacles), ενώ η μεταβλητή finish μετράει το συνολικό χρόνο εκτέλεσης του προγράμματος.

Στη γραμμή 59 δημιουργούμε ένα νέο αντικείμενο με όνομα 'fig2', αφού τελικά θα σχεδιάσουμε δυο διαγράμματα.

Στη γραμμή 63 καλούμε τη συνάρτηση nx.spring_layout() της βιβλιοθήκης networkx, στην οποία δίνουμε ως όρισμα το γράφο G, το λεξικό pos1{} με τις συντεταγμένες των κορυφών και τα κλειδιά του λεξικού δηλαδή τους αύξοντες αριθμούς των κόμβων ώστε να αναγνωρίζουμε κάθε κόμβο στο γράφο. Στη γραμμή 68 λαμβάνουμε πάλι τα ίδια μέτρα στη περίπτωση που η μέση τιμή της αντικειμενικής συνάρτησης έχει μεταβληθεί

ελάχιστα. Χρησιμοποιούμε τις συναρτήσεις `draw_networ_nodes()` και `draw_network_edges()` για να σχεδιάσουμε το γράφο και εκτυπώνουμε στον τίτλο την εξής πληροφορία:

- ‘START’ = κόμβος-πηγή
- ‘FINISH’ = κόμβος-προορισμού
- ‘CUTOFF’ = πλήθος τμημάτων από τα οποία αποτελείται το μονοπάτι
- ‘NUMBER OF NODES’ = πλήθος κόμβων
- ‘GA’s ROUTE’ = πιο σύντομη διαδρομή γενετικού αλγορίθμου
- ‘TARGET IN NEIGHBORS’ = False/True
- ‘GENETIC ALGORITHM’S DISTANCE’ = μήκος διαδρομής
- ‘FINISH RUNNING AFTER’ = ένδειξη χρονομέτρου

Στην περίπτωση που ο αλγόριθμος δεν βρει καμία λύση προστίθεται στον τίτλο το πεδίο:

- ‘SHORTEST ROUTE – DIJKSTRA’ = αποτέλεσμα κλήσης συνάρτησης `dijkstra_path()`

Στις γραμμές 98 -102 κατασκευάζουμε ένα αντίγραφο του γράφου G , το γράφο D , και στις γραμμές 122-123 χρησιμοποιούμε τις ίδιες συναρτήσεις με πριν για να σχεδιάσουμε το μονοπάτι που βρίσκει ο αλγόριθμος Dijkstra.

Ακολουθεί ένα παράδειγμα εκτέλεσης του παραπάνω κώδικα. Τα αντίστοιχα διαγράμματα παρουσιάζονται στο επόμενο Κεφάλαιο όπου θα μελετήσουμε την επίδοση του αλγορίθμου:

```
* * * * * NUMBER OF MUTATIONS: 9 * * * * *  
* * * * * * * * * * * * * * * * * * * * *  
*  
===== FITTEST PATH =====  
A201: [180.75950874 180.75950874 180.75950874 180.75950874  
180.75950874  
180.75950874 180.75950874 180.75950874 180.75950874 180.75950874  
180.75950874 180.75950874 180.75950874 180.75950874 180.75950874  
180.75950874 180.75950874 180.75950874 180.75950874 180.75950874  
180.75950874 180.75950874 180.75950874 180.75950874 180.75950874  
180.75950874 180.75950874 180.75950874]  
MEAN_1: 180.75950874020432  
SHORTEST LENGTH: 180.76  
SHORTEST_ARRAY: [list([9, 10]) array([ 9, 10]) list([9, 10])  
array([ 9, 10])  
array([ 9, 10]) array([ 9, 10]) array([ 9, 10]) array([ 9, 10])  
array([ 9, 10]) array([ 9, 10]) array([ 9, 10]) array([ 9, 10])  
array([ 9, 10]) array([ 9, 10]) array([ 9, 10]) array([ 9, 10])  
array([ 9, 10]) array([ 9, 10]) array([ 9, 10])]
```

GA's ROUTE IS:[9, 10]

DIJKSTRA'S ROUTE: [9, 10]

Finish running after: 24.9

Finish running after: 26.9

Finish running after: 2.0

MUTATION FREQUENCY: 0.2

CUTOFF: 3

DIJKSTRA'S DISTANCE: 180.76

GA's DISTANCE: 180.76

Τελικός χρόνος
εκτέλεσης αλγορίθμου
(ισούται με τη διαφορά
των δύο προηγούμενων
χρόνων)

5. Παρουσίαση Αποτελεσμάτων Γενετικού Αλγορίθμου #1

Σε αυτό το κεφάλαιο αφού αναφερθούμε στην παραμετροποίηση του Αλγορίθμου #1 θα παρουσιάσουμε τα αποτελέσματα από την εκτέλεση του για διαφορετικό πλήθος κόμβων.

5.1 Παραμετροποίηση

Κατά την εκκίνηση του αλγορίθμου οι βασικές παράμετροι που πρέπει να ρυθμιστούν είναι οι εξής:

- πλήθος των κορυφών του κάθε πολυγώνου,
- πλήθος των εμποδίων,
- πλήθος των επαναλήψεων-γενεών,
- την πιθανότητα να γίνει μετάλλαξη,
- πλήθος τμημάτων από τα οποία θέλουμε να αποτελείται η διαδρομή.

Υπογραμμίζουμε ότι λόγω του ότι η συνάρτηση `generatePolygon()` επιστρέφει κάθε φορά ένα πολύγωνο, τα πολυγωνικά εμπόδια αποτελούνται όλα από τον ίδιο αριθμό κορυφών. Επίσης, όπως αναφέραμε στην αρχή του προηγούμενου κεφαλαίου, έχουμε θέσει ως default τιμές για τις συντεταγμένες του κέντρου του κύκλου που δέχεται ως όρισμα η παραπάνω συνάρτηση τις τιμές:

- $c_x = 30 * (i \% 3)$,
- $c_y = i * i + 40$.

όπου i είναι ο αριθμός του εμποδίου. Με αυτές τις τιμές παρατηρήσαμε ότι τα εμπόδια κατανέμονται πιο ομοιόμορφα στο επίπεδο οπότε:

- αυξάνει η πιθανότητα να υπάρχει πράγματι μονοπάτι από τον κόμβο-πηγή στον κόμβο-προορισμού,
- δημιουργούνται πολλές εναλλακτικές διαδρομές τις οποίες μπορεί να εξετάσει ο αλγόριθμος.

Κατά την επιλογή του πλήθους των τμημάτων από τα οποία επιθυμεί να αποτελείται η διαδρομή, το πρόγραμμα προτρέπει το χρήστη να εισάγει είτε τον αριθμό '2', είτε τον αριθμό '3'. Αν η επιλογή είναι ο αριθμός '1' τότε η διαδρομή θα αποτελείται από το πολύ τρία τμήματα άρα από τέσσερις κόμβους οπότε δεν μπορεί να εφαρμοστεί η μετάλλαξη `g_shuffle`, επομένως ο γενετικός αλγόριθμος ενδέχεται να μην καταλήξει σε λύση.

Σημειώνουμε ότι κατά την επιλογή της συχνότητας μετάλλαξης όσο μεγαλύτερη τιμή δώσει ο χρήστης, τόσο ο αλγόριθμος καθυστερεί να καταλήξει σε λύση. Το ίδιο συμβαίνει και με την επιλογή μεγάλου αριθμού επαναλήψεων.

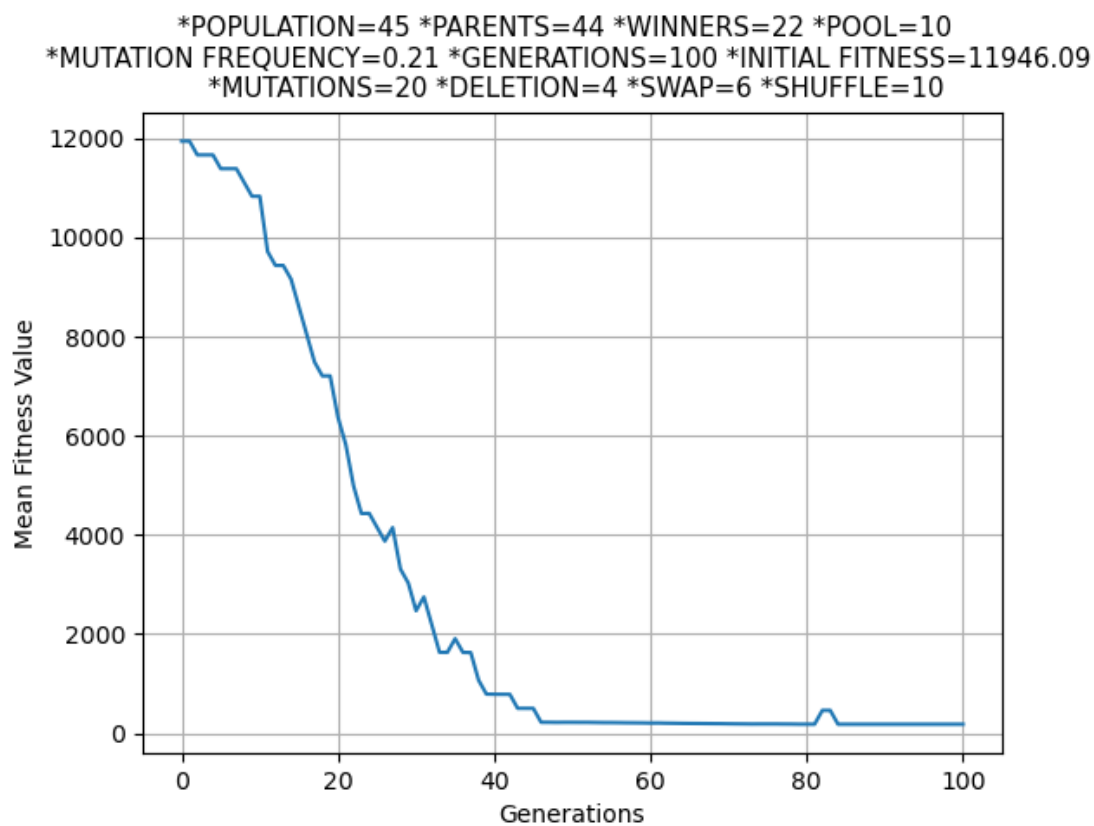
Πρέπει να προσθέσουμε ότι αν και ο χρήστης επιλέγει στην εκκίνηση του προγράμματος μια τιμή π.χ., την τιμή '3' για τον αριθμό των τμημάτων από τα οποία επιθυμεί να αποτελείται η διαδρομή, ο αλγόριθμος προσθέτει στην τιμή αυτή συν δύο ακόμη τμήματα, την ακμή του κόμβου-πηγή με το γείτονά του και την ακμή που καταλήγει στον κόμβο-προορισμό. Άρα τελικά ψάχνει για λύσεις με πέντε (3+2) τμήματα. Το αντίστοιχο πεδίο στον τίτλο του γραφήματος με ένδειξη 'CUTOFF' μας πληροφορεί για την προηγούμενη παρατήρηση. Επίσης, αν και ο αλγόριθμος στην περίπτωση αυτή ψάχνει για μονοπάτια με πέντε τμήματα, ενδέχεται τελικά να επιστρέψει μονοπάτι με λιγότερα τμήματα ή ακόμη και ένα αφού η μετάλλαξη deletion έχει κατασκευαστεί έτσι ώστε να αφαιρεί ακμές.

5.2 Μελέτη Αποτελεσμάτων Γενετικού Αλγορίθμου #1

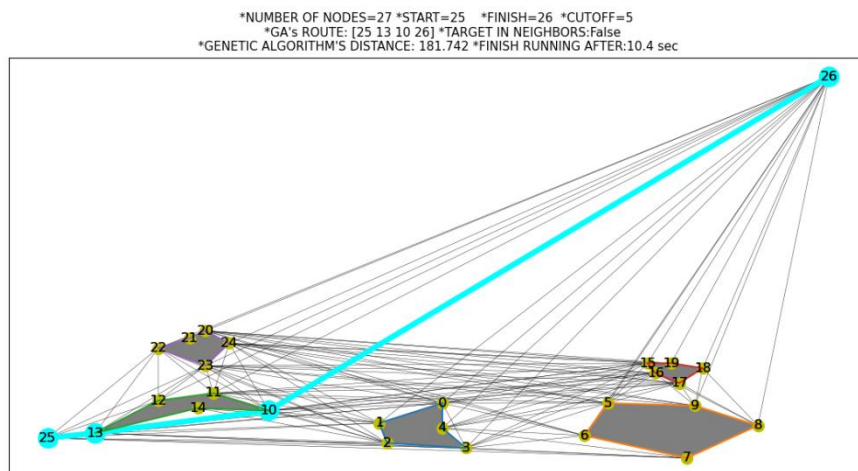
Παρακάτω διεξάγουμε συγκριτικούς ελέγχους για την εύρεση του συντομότερου μονοπατιού με την εφαρμογή του γενετικού αλγορίθμου που κατασκευάσαμε και του αλγορίθμου Dijkstra για τον ίδιο γράφο ορατότητας. Οι δύο αλγόριθμοι αξιολογούνται ως προς το μήκος της διαδρομής που επιστρέφει ο καθένας. Κάθε συγκριτικός έλεγχος μας δείχνει τις παραμέτρους με τις οποίες έτρεξε ο γενετικός αλγόριθμος, όπως τον αριθμό των επαναλήψεων, τη συχνότητα μετάλλαξης, το πλήθος των μεταλλάξεων από κάθε τύπο μετάλλαξης, τον αριθμό των ατόμων του πληθυσμού, τον αριθμό των ζευγαριών ο οποίος ισούται με τον αριθμό των παιδιών σε κάθε γενιά και το χρόνο εκτέλεσης του γενετικού αλγορίθμου. Σε κάθε σενάριο παρουσιάζουμε εικόνες με τη διαδρομή ανάμεσα στα εμπόδια την οποία επιστρέφει κάθε αλγόριθμος. Οι θέσεις των κόμβων πηγής και προορισμού έχουν επιλεγεί με τέτοιο τρόπο ώστε να δημιουργούνται πολλές εναλλακτικές διαδρομές, οπότε αναγκάζουμε κατά κάποιο τρόπο τον γενετικό αλγόριθμο να ξεκινήσει τις γενετικές διαδικασίες. Για αυτό το λόγο έχουμε προσαρμόσει κατάλληλα τα διαστήματα από τα οποία η συνάρτηση `random.randint()` επιλέγει τις συντεταγμένες του κάθε κόμβου έτσι ώστε οι δύο κόμβοι να βρίσκονται σε σχετικά μεγάλη απόσταση ο ένας από τον άλλο. Υπενθυμίζουμε ότι ο τύπος μετάλλαξης `cluster_swap2()` εξετάζεται χωριστά από τις τρεις υπόλοιπες μεταλλάξεις. Τα σενάρια 1.11 και 1.12 αφορούν αυτή τη μετάλλαξη.

Σενάριο 1.1- Γράφος Ορατότητας με 27 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	45
Αριθμός ζευγαριών/παιδιών	10
Μεταλλάξεις	20
Deletion	4
Swap	6
Shuffle	10
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	181.742
Μήκος μονοπατιού Dijkstra	175.574
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	10.4 sec

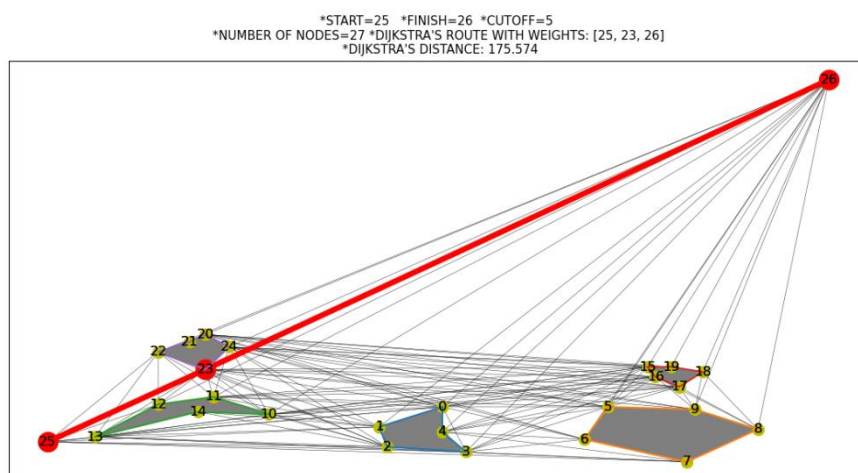
Πίνακας 3 Σενάριο 1.1



Διάγραμμα 1 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.1



Εικόνα 52 Συντομότερη διαδρομή από κόμβο '25' προς κόμβο '26' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 27 κόμβους

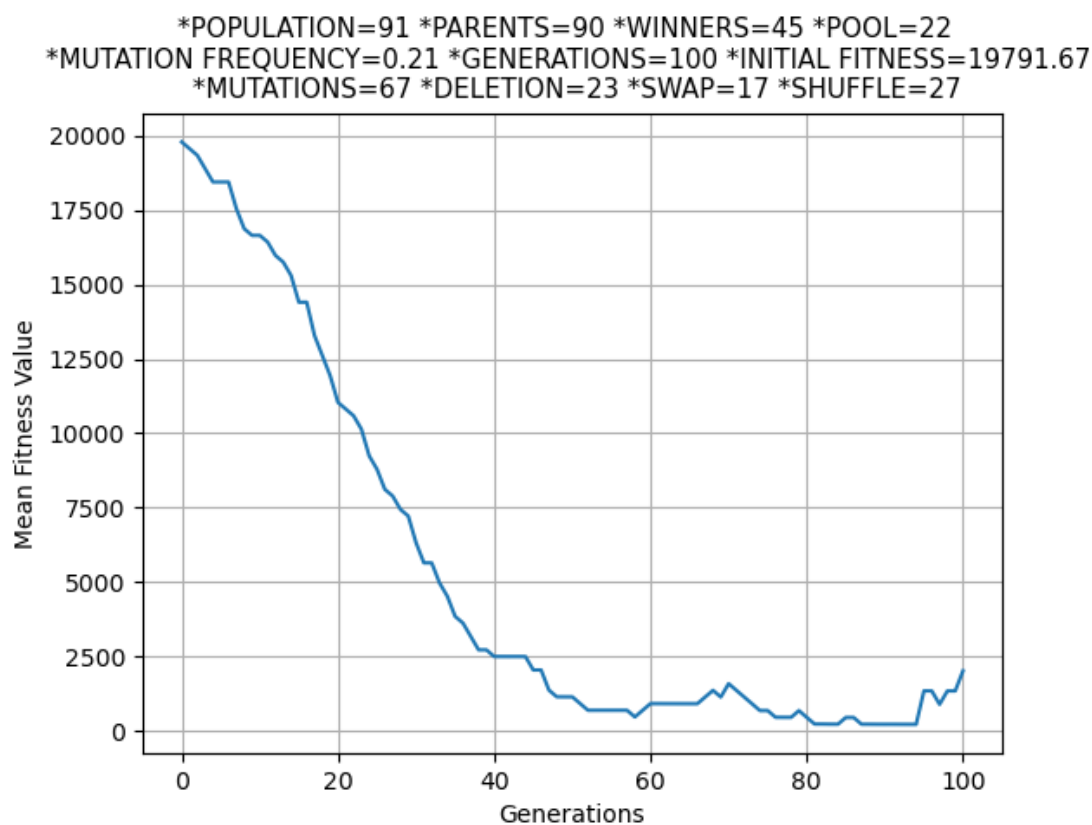


Εικόνα 53 Συντομότερη διαδρομή από κόμβο '25' προς κόμβο '26' με χρήση Αλγορίθμου Dijkstra για Γράφο με 27 κόμβους

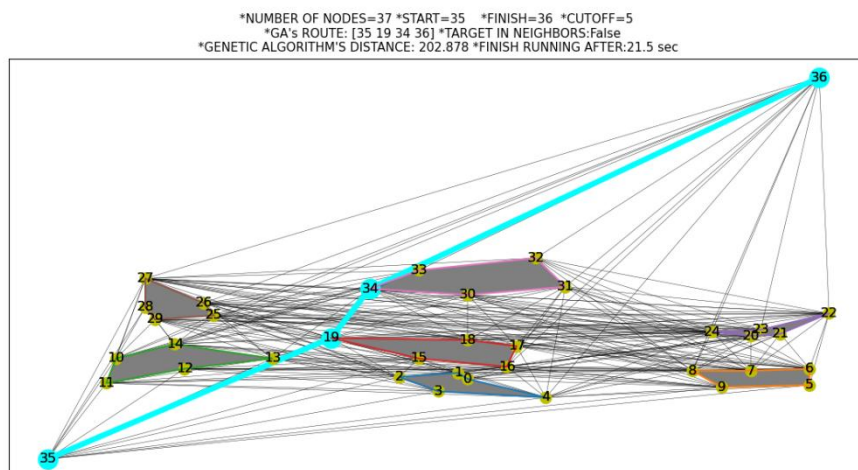
Στο σενάριο 1.1 παρατηρούμε ότι ο Γενετικό Αλγόριθμος #1 καταλήγει τελικά σε μονοπάτι το οποίο έχει σχεδόν το ίδιο μήκος με το μήκος της διαδρομής που επιστρέφει ο Αλγόριθμος Dijkstra. Από το διάγραμμα Mean Fitness Value/Generations παρατηρούμε ότι ο γενετικός αλγόριθμος έχει καταλήξει σε λύση ήδη από τη 45^η γενιά. Επίσης παρατηρούμε ότι ενώ η μέση τιμή της αντικειμενικής συνάρτησης ξεκινά από τιμή περίπου ίση με 12000, μετά από 45 γενιές η τιμή αυτή σχεδόν μηδενίζεται επομένως συμπεραίνουμε ότι ο πληθυσμός έχει αντικατασταθεί από αντίγραφα της μέχρι τότε πιο σύντομης διαδρομής [25, 13, 10, 26].

Σενάριο 1.2 - Γράφος Ορατότητας με 37 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	91
Αριθμός ζευγαριών/παιδιών	22
Μεταλλάξεις	67
Deletion	23
Swap	17
Shuffle	27
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	202.878
Μήκος μονοπατιού Dijkstra	202.374
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	21.5 sec

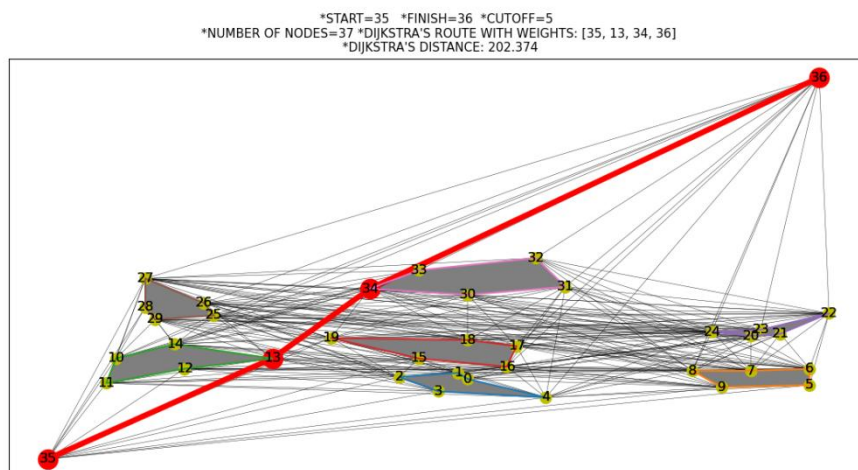
Πίνακας 4 Σενάριο 1.2



Διάγραμμα 2 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.2



Εικόνα 54 Συντομότερη διαδρομή από κόμβο '35' προς κόμβο '36' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 37 κόμβους

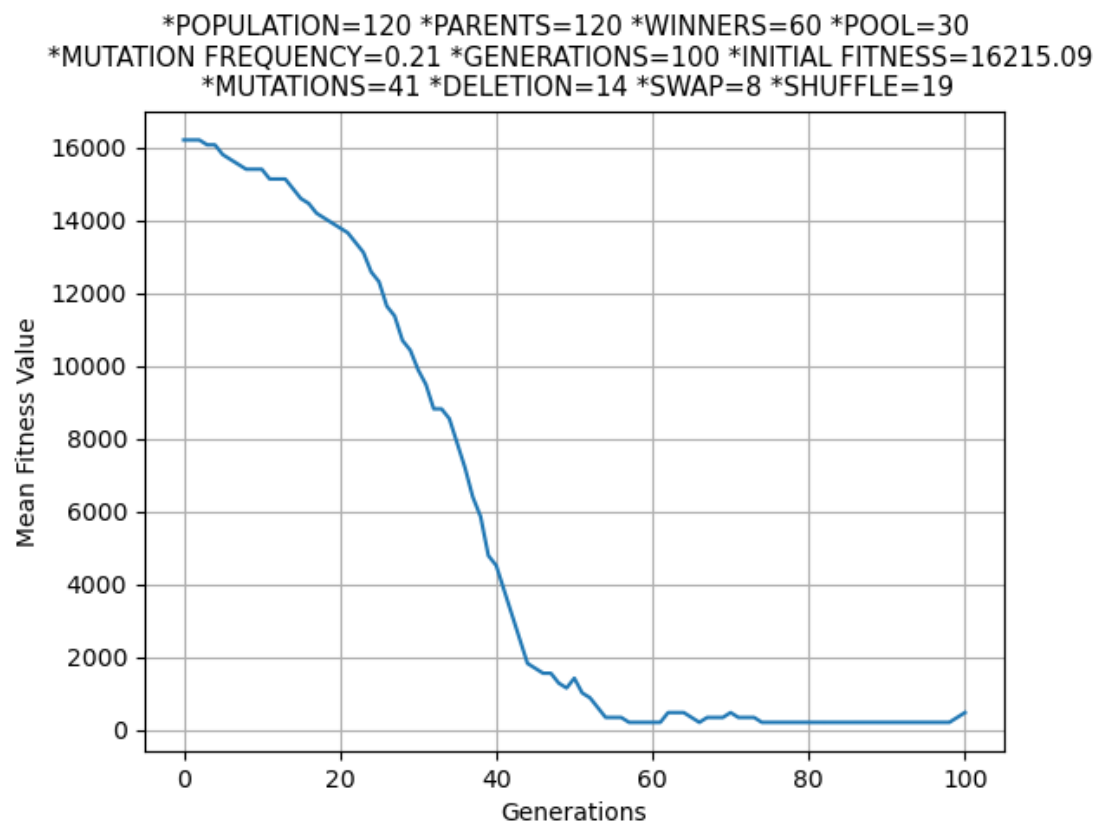


Εικόνα 55 Συντομότερη διαδρομή από κόμβο '35' προς κόμβο '36' με χρήση Αλγορίθμου Dijkstra για Γράφο με 37 κόμβους

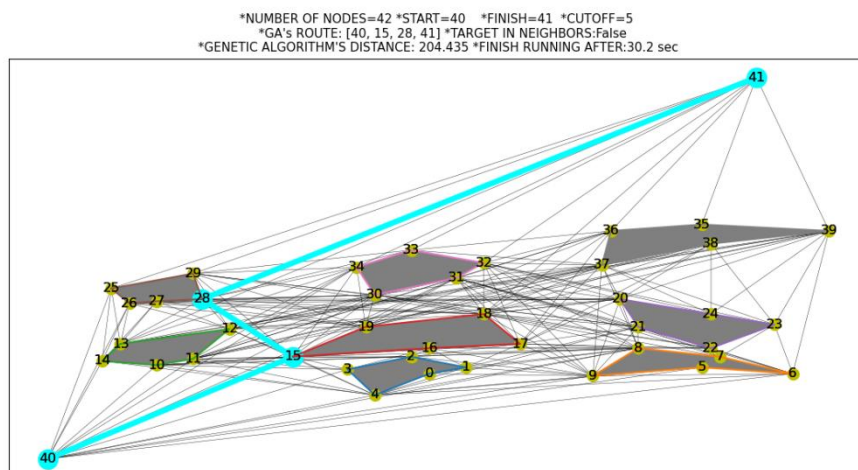
Στο σενάριο 1.2 όπου ο γράφος ορατότητας αποτελείται από 37 κόμβους παρατηρούμε και πάλι ότι το μήκος του μονοπατιού που υπολογίζει ο Γενετικός Αλγόριθμος #1 είναι σχεδόν ίδιο με αυτό στο οποίο καταλήγει ο Αλγόριθμος Dijkstra. Λόγω του ότι τώρα γεννιούνται περισσότερα παιδιά σε κάθε γενιά, ο αριθμός των μεταλλάξεων είναι αυξημένος σε σχέση με το Σενάριο 1 για αυτό και ο χρόνος εκτέλεσης του Γενετικού Αλγορίθμου #1 έχει αυξηθεί, αν και ο αριθμός γενεών παραμένει ο ίδιος.

Σενάριο 1.3 - Γράφος Ορατότητας με 42 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	120
Αριθμός ζευγαριών/παιδιών	30
Μεταλλάξεις	41
Deletion	14
Swap	8
Shuffle	19
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	204.435
Μήκος μονοπατιού Dijkstra	195.49
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	30.2 sec

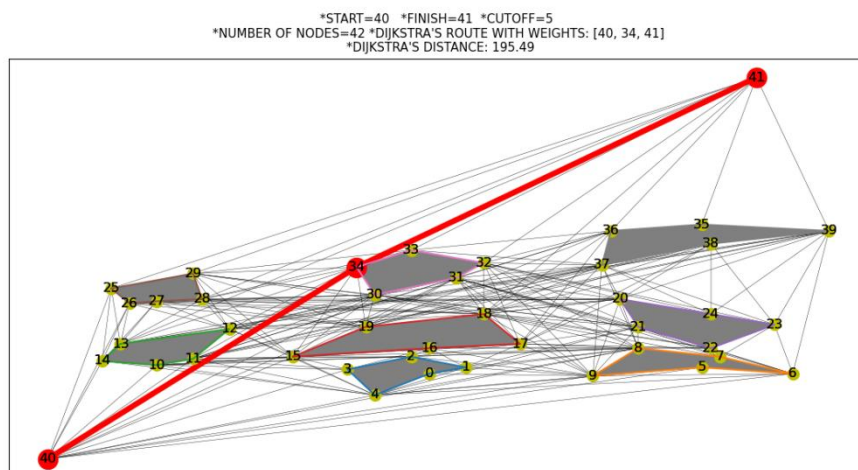
Πίνακας 5 Σενάριο 1.3



Διάγραμμα 3 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.3



Εικόνα 56 Συντομότερη διαδρομή από κόμβο '40' προς κόμβο '41' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 42 κόμβους



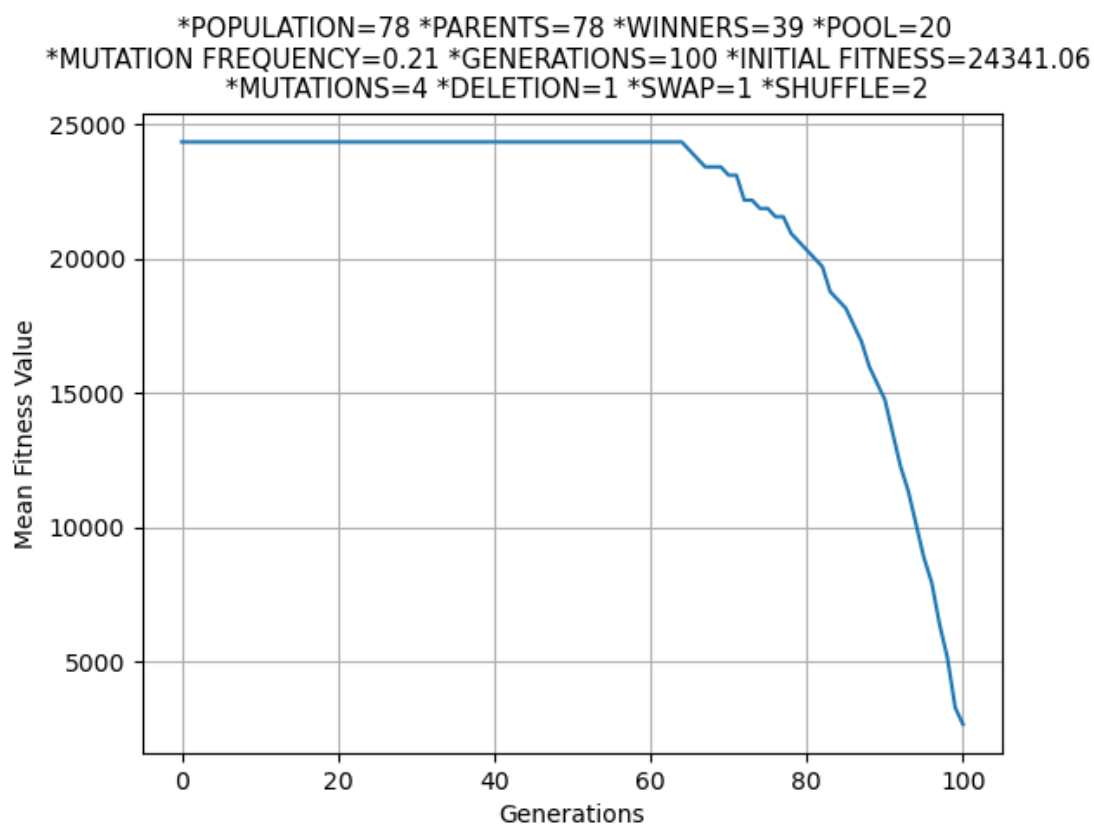
Εικόνα 57 Συντομότερη διαδρομή από κόμβο '40' προς κόμβο '41' με χρήση Αλγορίθμου Dijkstra για Γράφο με 42 κόμβους

Στο σενάριο 1.3 παρατηρούμε ότι υπάρχει διαφορά στις συντομότερες διαδρομές που προτείνει κάθε αλγόριθμος με αυτή του αλγορίθμου Dijkstra να είναι η συντομότερη και να αποτελείται από 2 τμήματα ενώ η διαδρομή που προτείνει ο Γενετικός Αλγόριθμος αποτελείται από τρία τμήματα. Λόγω του ότι αυξάνονται οι κόμβοι του γράφου αυξάνεται και ο πληθυσμός, επειδή αυξάνεται η πιθανότητα ο κόμβος-πηγή να έχει περισσότερους γείτονες. Αντίστοιχα αυξάνεται και το πλήθος των ζευγαριών/παιδιών. Επίσης λόγω των περισσότερων διασταυρώσεων μεταξύ των ατόμων του πληθυσμού αυξάνεται ο χρόνος εκτέλεσης του αλγορίθμου. Και στο Σενάριο 1.3 όμως παρατηρούμε ότι ο Γενετικός Αλγόριθμος #1 έχει καταλήξει σε λύση ήδη από την 60^η γενιά. Σημειώνουμε ότι έχουμε

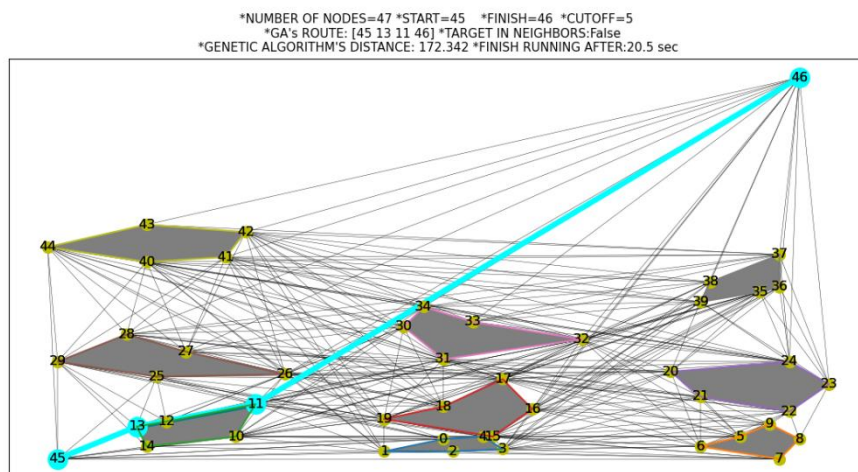
σχεδιάσει τον αλγόριθμο έτσι ώστε από κάθε διασταύρωση να προκύπτει ένα μόνο παιδί. Επίσης, έχουμε επιλέξει κατά τη διασταύρωση το χρωμόσωμα του κάθε γονέα να χωρίζεται σε δύο τμήματα (one-point crossover).

Σενάριο 1.4 - Γράφος Ορατότητας με 47 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	78
Αριθμός ζευγαριών/παιδιών	20
Μεταλλάξεις	4
Deletion	1
Swap	1
Shuffle	2
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	172.342
Μήκος μονοπατιού Dijkstra	170.936
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	20.5 sec

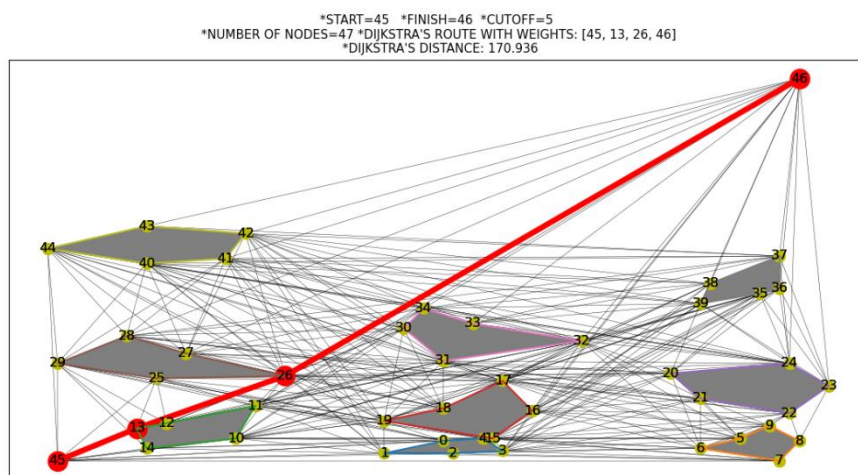
Πίνακας 6 Σενάριο 1.4



Διάγραμμα 4 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.4



Εικόνα 58 Συντομότερη διαδρομή από κόμβο '45' προς κόμβο '46' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 47 κόμβους

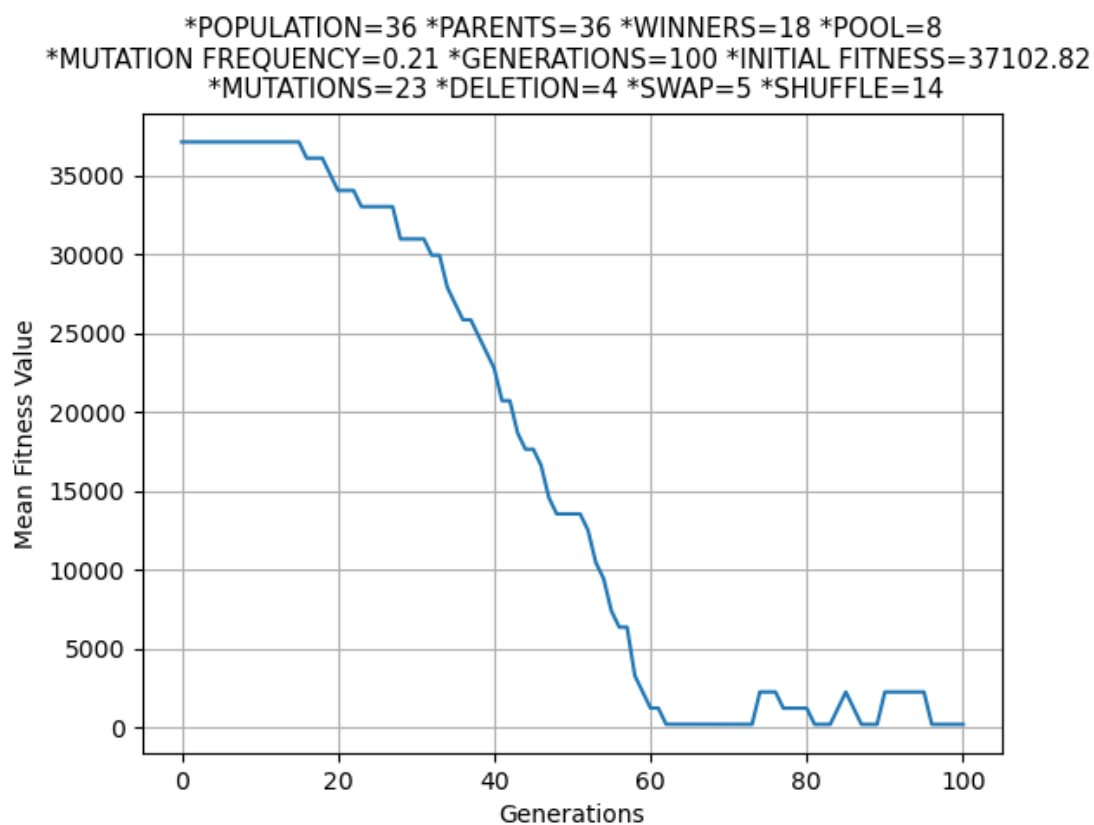


Εικόνα 59 Συντομότερη διαδρομή από κόμβο '45' προς κόμβο '46' με χρήση Αλγορίθμου Dijkstra για Γράφο με 47 κόμβους

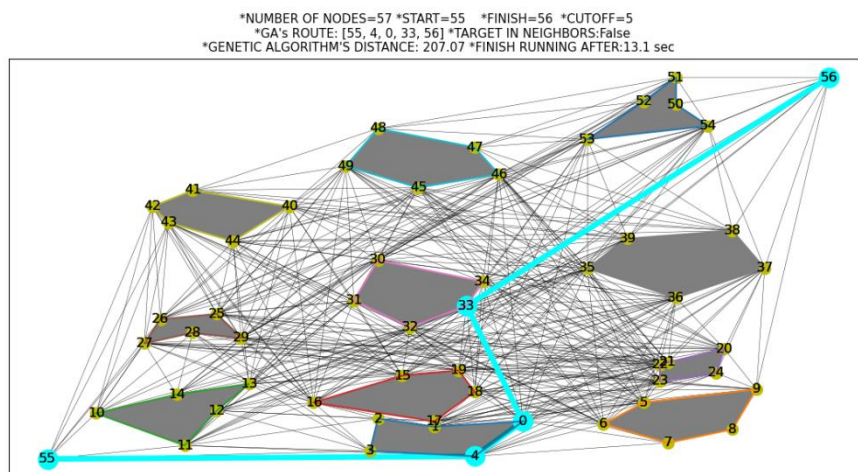
Στο Διάγραμμα 4 του σεναρίου 1.4 παρατηρούμε ότι μέχρι την 60^η γενιά η μεταβολή στη μέση τιμή της αντικειμενικής συνάρτησης είναι μηδενική. Αυτό σημαίνει ότι δεν εισέρχονται στον πληθυσμό καινούρια άτομα με χαμηλότερη τιμή αντικειμενικής συνάρτησης. Παρόλα αυτά με μόνο τέσσερις μεταλλάξεις εμφανίζεται μέχρι την 100^η γενιά στον πληθυσμό το μονοπάτι με μήκος 172.342 οπότε και επιλέγεται από τον Γενετικό Αλγόριθμο#1 σαν βέλτιστη λύση.

Σενάριο 1.5 - Γράφος Ορατότητας με 57 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	36
Αριθμός ζευγαριών/παιδιών	8
Μεταλλάξεις	23
Deletion	4
Swap	5
Shuffle	14
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	207.07
Μήκος μονοπατιού Dijkstra	173.021
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	13.1 sec

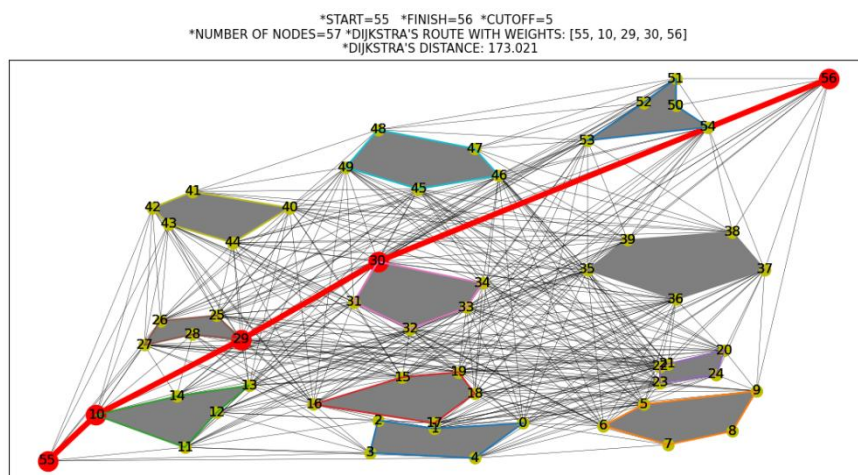
Πίνακας 7 Σενάριο 1.5



Διάγραμμα 5 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.5



Εικόνα 60 Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 57 κόμβους

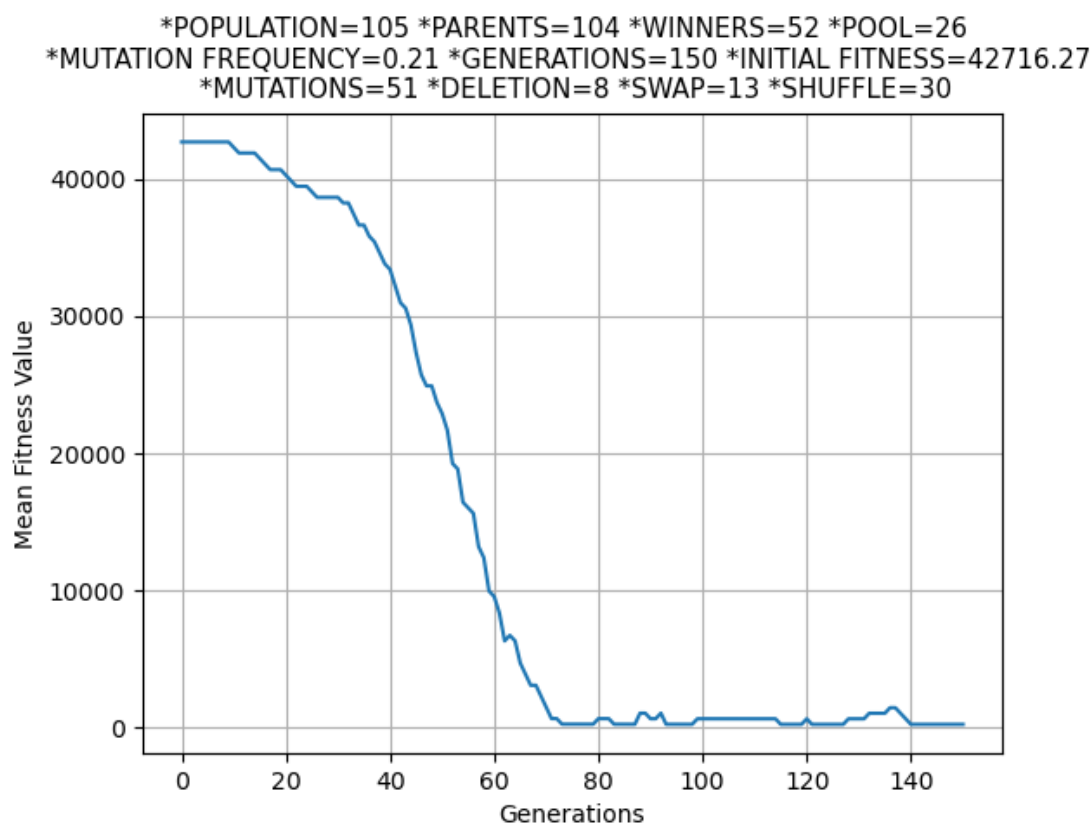


Εικόνα 61 Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Αλγορίθμου Dijkstra για Γράφο με 57 κόμβους

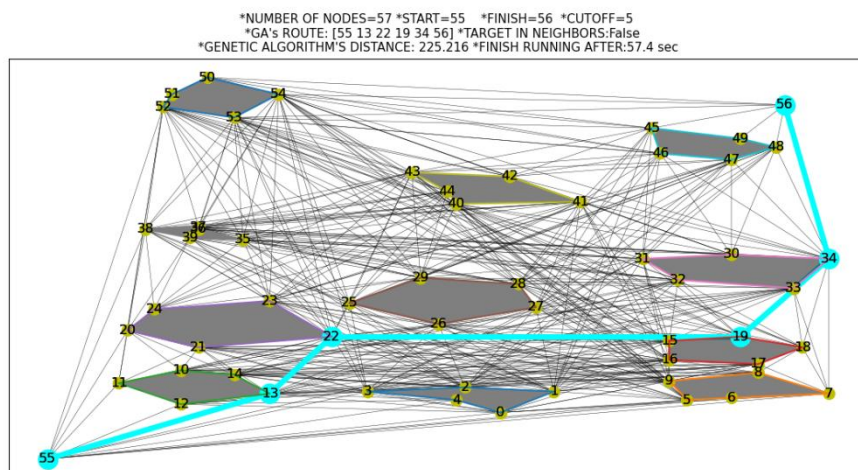
Στο σενάριο 1.5 παρατηρούμε ότι η απόκλιση των δύο διαδρομών είναι σχετικά μεγάλη με τη συντομότερη διαδρομή να βρίσκει και πάλι ο Αλγόριθμος Dijkstra. Αυτό συμβαίνει διότι όπως παρατηρούμε και στον Πίνακα 7, ο πληθυσμός αποτελείται μόνο από 36 άτομα επομένως δημιουργούνται μόνο 8 ζευγάρια, άρα και 8 παιδιά σε κάθε γενιά. Αποδεικνύεται επομένως ότι για να ανακαλύψει εναλλακτικές λύσεις ο Γενετικός Αλγόριθμος #1 απαιτεί μεγάλο μέγεθος πληθυσμού εις βάρος όμως του χρόνου εκτέλεσης.

Σενάριο 1.6 - Γράφος Ορατότητας με 57 κόμβους	
Αριθμός γενεών	150
Πληθυσμός	105
Αριθμός ζευγαριών/παιδιών	26
Μεταλλάξεις	51
Deletion	8
Swap	13
Shuffle	30
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	225.216
Μήκος μονοπατιού Dijkstra	191.308
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	57.4 sec

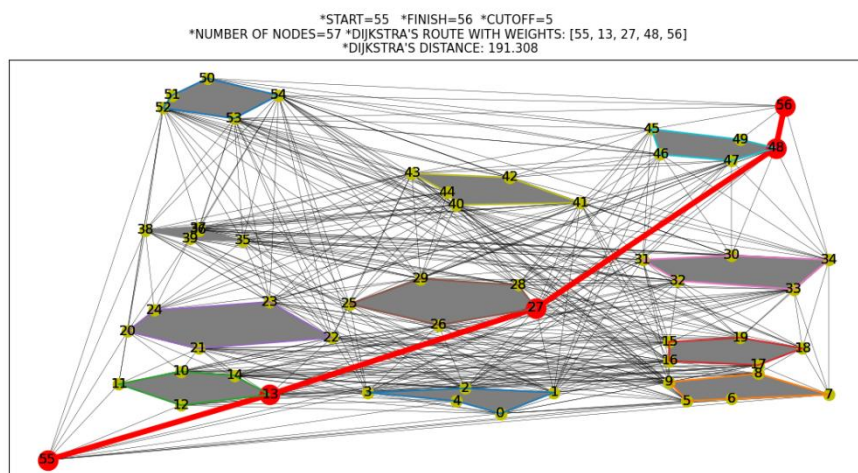
Πίνακας 8 Σενάριο 1.6



Διάγραμμα 6 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.6



Εικόνα 62 Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 57 κόμβους

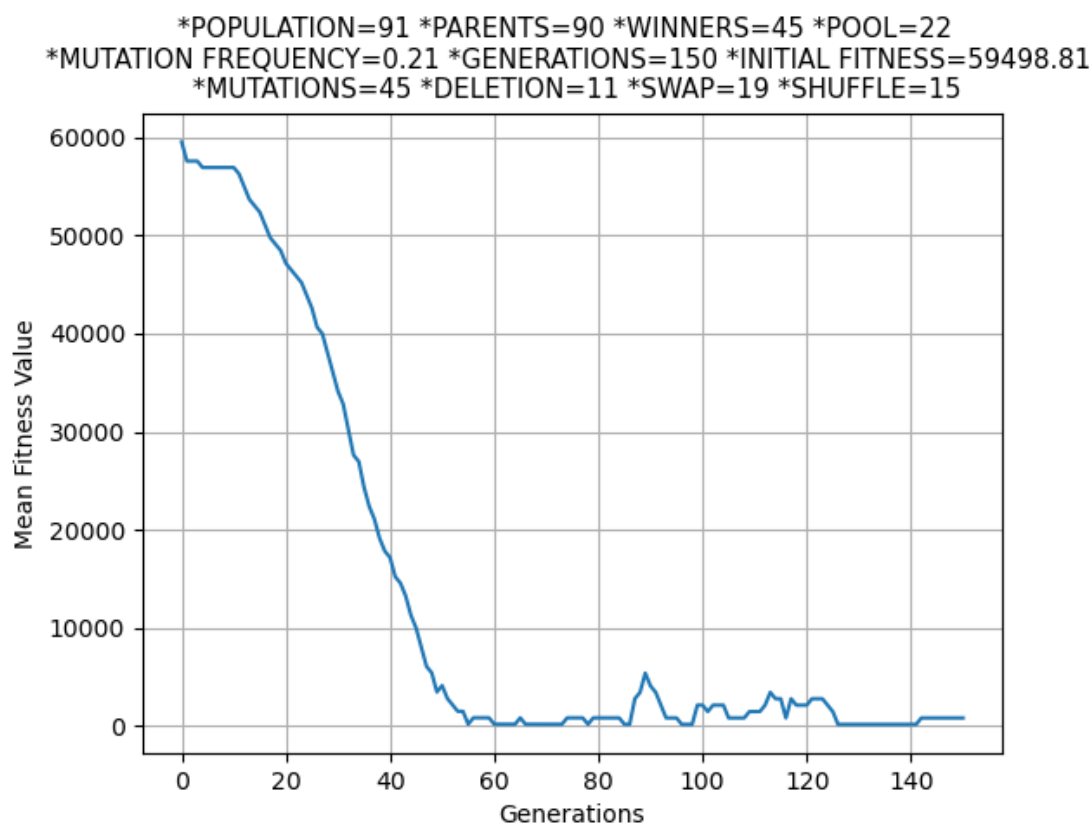


Εικόνα 63 Συντομότερη διαδρομή από κόμβο '55' προς κόμβο '56' με χρήση Αλγορίθμου Dijkstra για Γράφο με 57 κόμβους

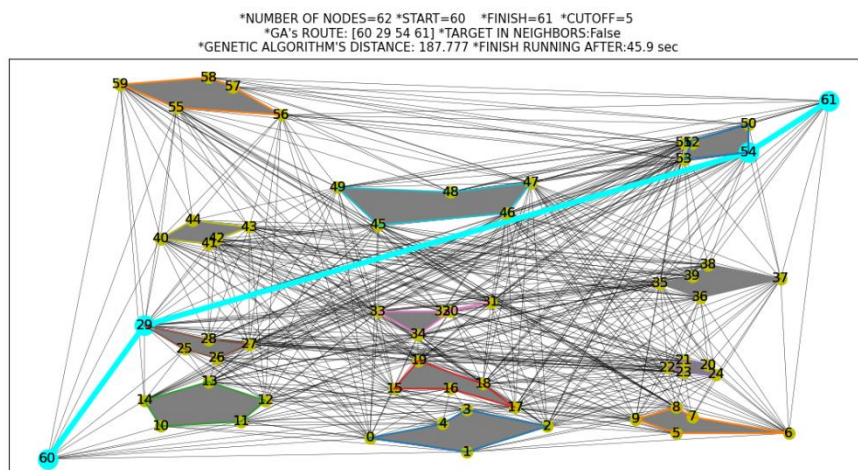
Στο σενάριο 1.6 αυξήσαμε τον αριθμό των γενεών από 100 σε 150 και παρατηρούμε ότι υπήρξε αύξηση και στον αριθμό των μεταλλάξεων με 51 συνολικά μεταλλάξεις. Ήδη όμως από την 70^η περίπου γενιά ο αλγόριθμος έχει εντοπίσει την καλύτερη λύση που μπορεί να βρει. Διαπιστώνουμε ότι η αύξηση του αριθμού των επαναλήψεων επιφέρει και χρονική επιβάρυνση αφού ο χρόνος εκτέλεσης έχει αυξηθεί στα 57.4 δευτερόλεπτα. Επίσης διαπιστώνουμε τη σημασία που έχει το κριτήριο τερματισμού στην επίδοση του αλγορίθμου.

Σενάριο 1.7 - Γράφος Ορατότητας με 62 κόμβους	
Αριθμός γενεών	150
Πληθυσμός	91
Αριθμός ζευγαριών/παιδιών	22
Μεταλλάξεις	45
Deletion	11
Swap	19
Shuffle	15
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	187.777
Μήκος μονοπατιού Dijkstra	181.747
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	45.9 sec

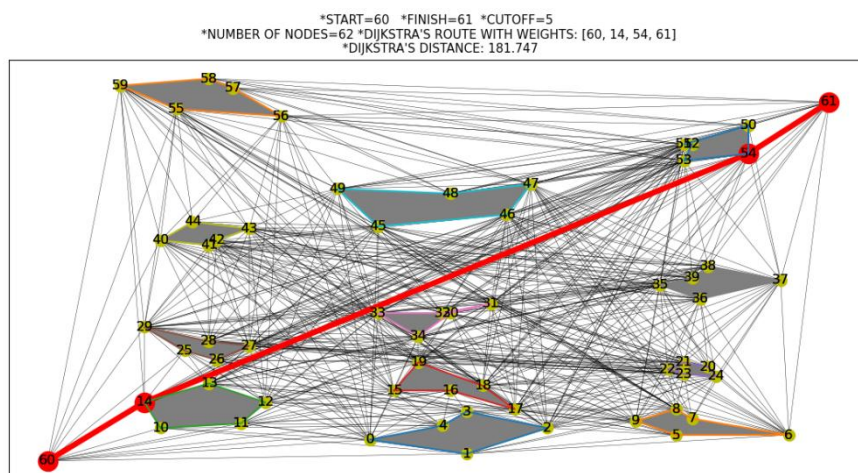
Πίνακας 9 Σενάριο 1.7



Διάγραμμα 7 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.7



Εικόνα 64 Συντομότερη διαδρομή από κόμβο '60' προς κόμβο '61' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 62 κόμβους



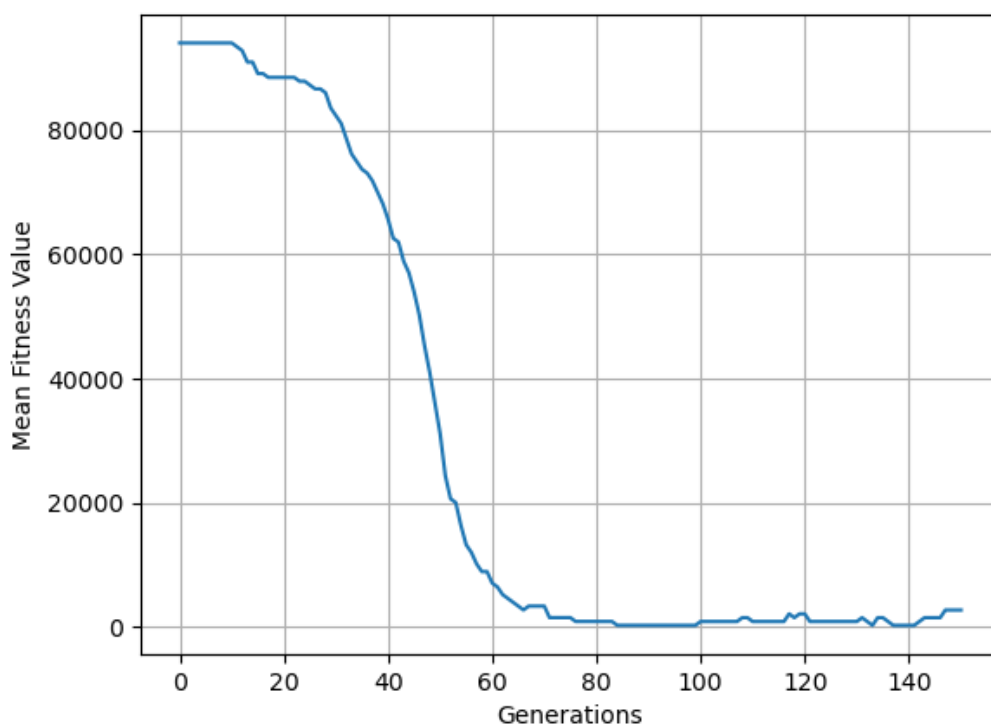
Εικόνα 65 Συντομότερη διαδρομή από κόμβο '60' προς κόμβο '61' με χρήση Γενετικού Αλγορίθμου για Γράφο με 62 κόμβους

Στο σενάριο 1.7 παρατηρούμε ότι οι δύο αλγόριθμοι επιστρέφουν διαδρομές με ίδια περίπου απόσταση για γράφο με 62 κόμβους. Όπως παρατηρούμε και από το Διάγραμμα 7 στο σενάριο αυτό άτομα με χαμηλή τιμή αντικειμενικής συνάρτησης έχουν ήδη εισέλθει στον πληθυσμό στην 60^η γενιά.

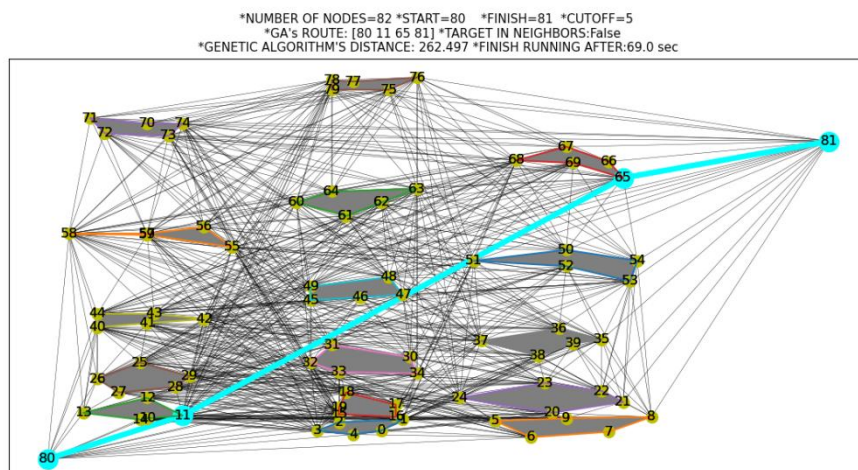
Σενάριο 1.8 - Γράφος Ορατότητας με 82 κόμβους	
Αριθμός γενεών	150
Πληθυσμός	153
Αριθμός ζευγαριών/παιδιών	38
Μεταλλάξεις	58
Deletion	23
Swap	13
Shuffle	22
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	262.497
Μήκος μονοπατιού Dijkstra	259.578
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	69.0 sec

Πίνακας 10 Σενάριο 1.8

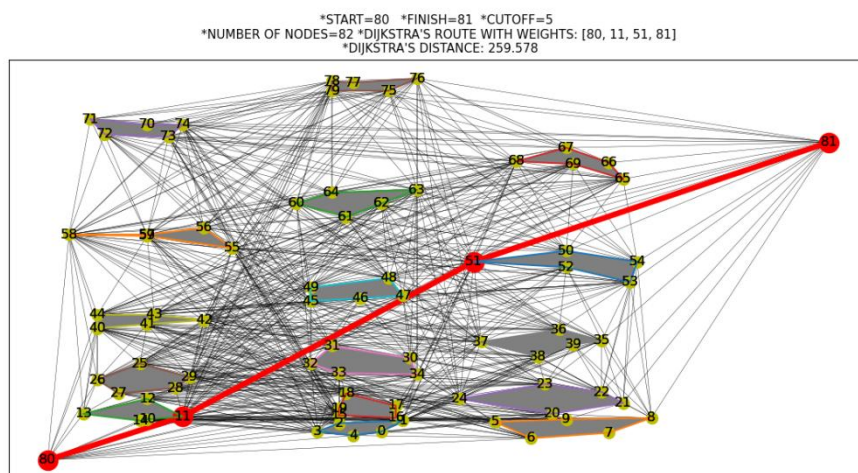
*POPULATION=153 *PARENTS=152 *WINNERS=76 *POOL=38
 *MUTATION FREQUENCY=0.21 *GENERATIONS=150 *INITIAL FITNESS=94006.4
 *MUTATIONS=58 *DELETION=23 *SWAP=13 *SHUFFLE=22



Διάγραμμα 8 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.8



Εικόνα 66 Συντομότερη διαδρομή από κόμβο '80' προς κόμβο '81' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 82 κόμβους

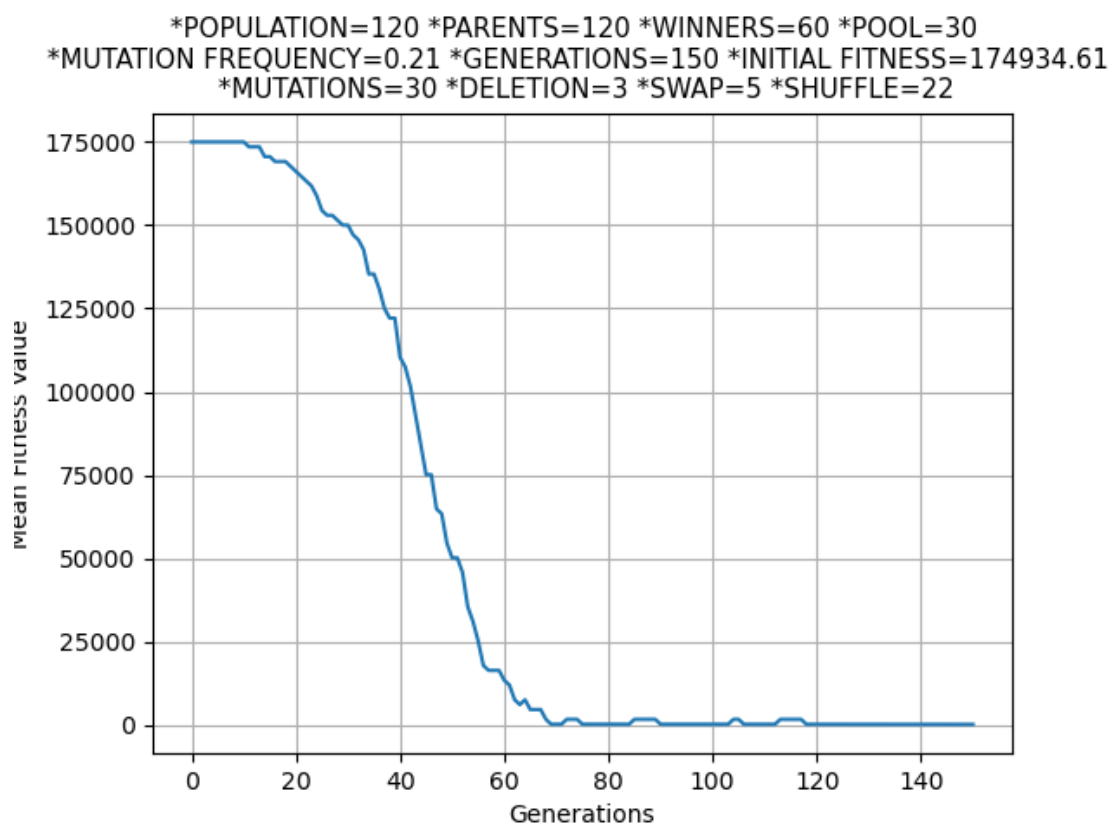


Εικόνα 67 Συντομότερη διαδρομή από κόμβο '80' προς κόμβο '81' με χρήση Αλγορίθμου Dijkstra για Γράφο με 82 κόμβους

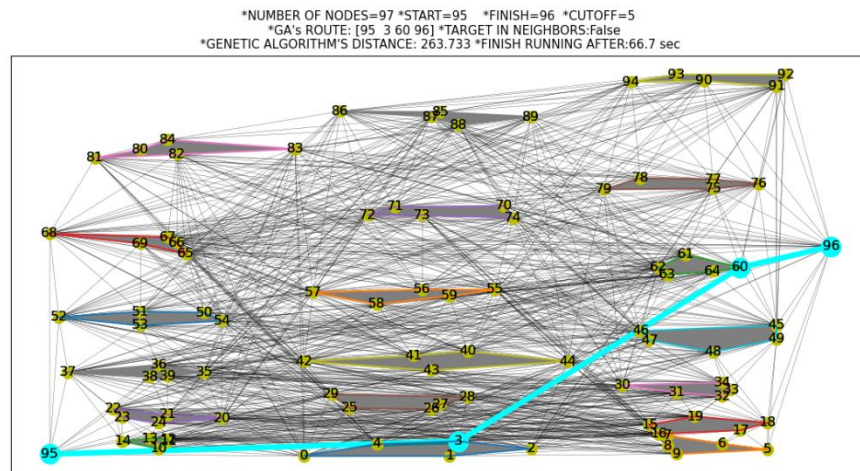
Στο σενάριο 1.8 ο χρόνος εκτέλεσης του αλγορίθμου ξεπερνάει τα 60 δευτερόλεπτα. Παρόλα αυτά το μήκος των δύο διαδρομών είναι περίπου ίδιο. Θα πρέπει να σημειώσουμε ότι ο χρόνος εκτέλεσης του Γενετικού Αλγορίθμου #1 επιβαρύνεται και από το γεγονός ότι πρέπει να διεξάγει ελέγχους ώστε η συνάρτηση generatePolygon() να επιστρέφει έγκυρα πολύγωνα. Επιπλέον έλεγχοι ώστε τα τυχαία σημεία πηγής-προορισμού να μην βρίσκονται μέσα στα τυχαία εμποδία προσθέτουν ακόμη μεγαλύτερη χρονική επιβάρυνση.

Σενάριο 1.9 - Γράφος Ορατότητας με 97 κόμβους	
Αριθμός γενεών	150
Πληθυσμός	120
Αριθμός ζευγαριών/παιδιών	30
Μεταλλάξεις	30
Deletion	3
Swap	5
Shuffle	22
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	263.733
Μήκος μονοπατιού Dijkstra	243.387
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	66.7 sec

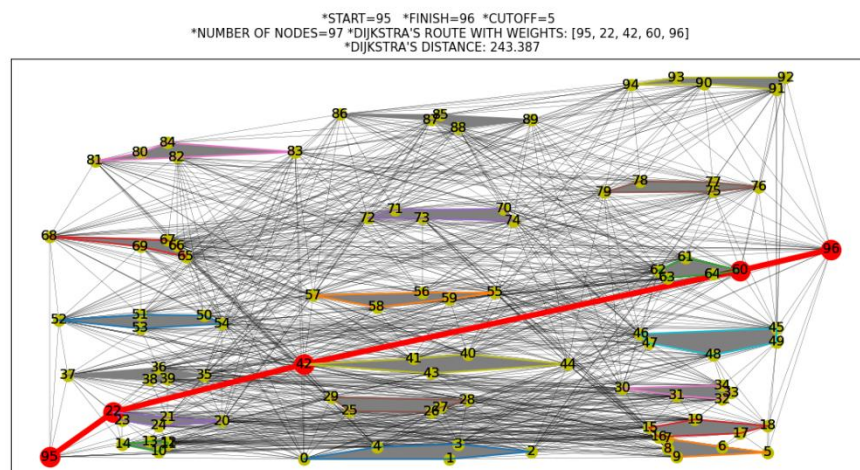
Πίνακας 11 Σενάριο 1.9



Διάγραμμα 9 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.9



Εικόνα 68 Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 97 κόμβους



Εικόνα 69 Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Αλγορίθμου Dijkstra για Γράφο με 97 κόμβους

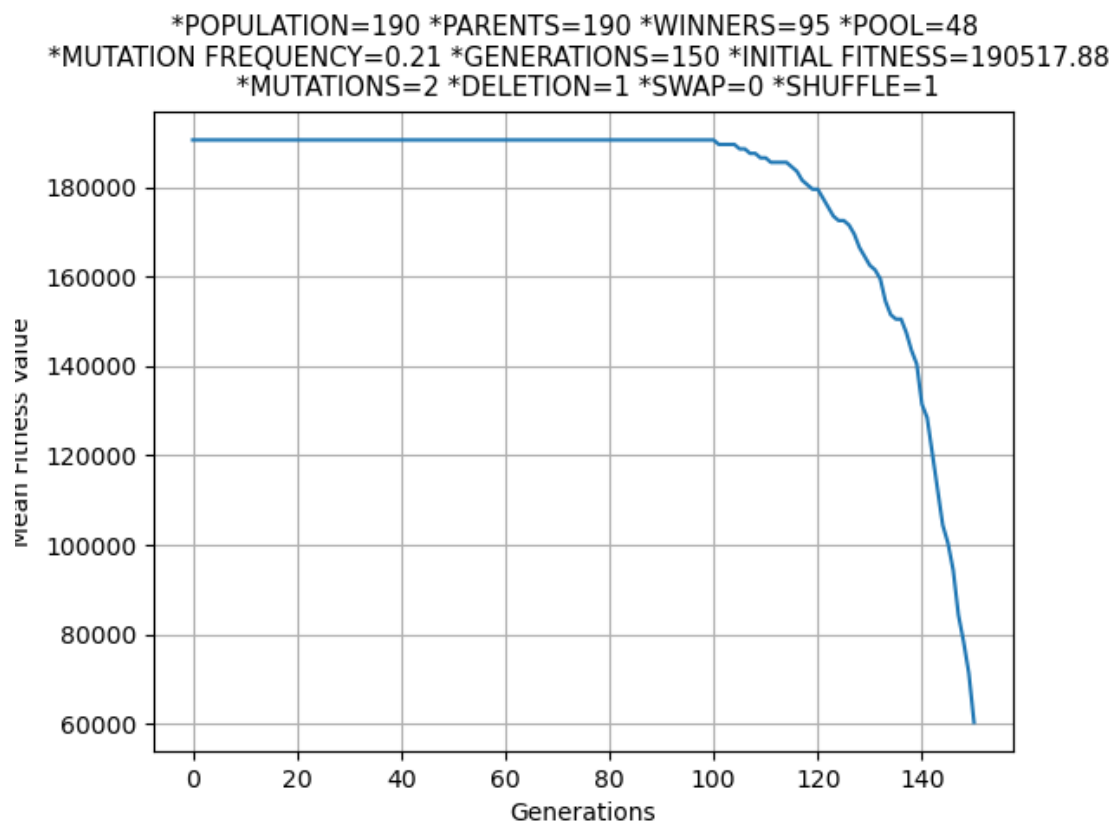
Στο σενάριο 1.9 τα 19 πολύγωνα μαζί με τον αρχικό και τελικό κόμβο σχηματίζουν γράφο με 97 κόμβους. Μειώσαμε το πλάτος των ακμών του γράφου σε:

$$\text{width}=0.3,$$

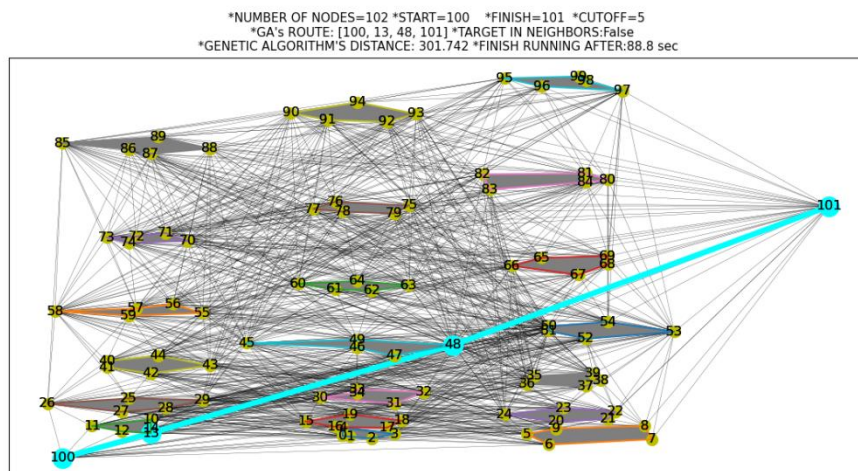
ώστε τα εμπόδια να είναι πιο ευδιάκριτα.

Σενάριο 1.10 - Γράφος Ορατότητας με 102 κόμβους	
Αριθμός γενεών	150
Πληθυσμός	190
Αριθμός ζευγαριών/παιδιών	48
Μεταλλάξεις	2
Deletion	1
Swap	0
Shuffle	1
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	301.742
Μήκος μονοπατιού Dijkstra	301.742
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	88.8 sec

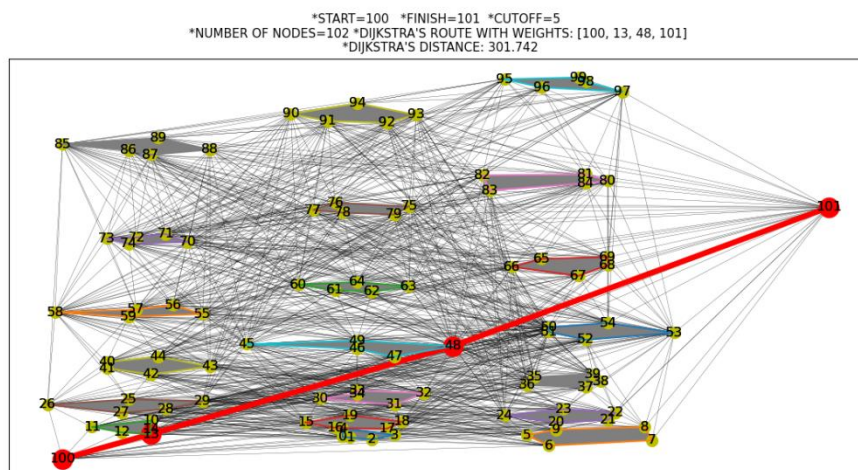
Πίνακας 12 Σενάριο 1.10



Διάγραμμα 10 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.10



Εικόνα 70 Συντομότερη διαδρομή από κόμβο '100' προς κόμβο '101' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 102 κόμβους

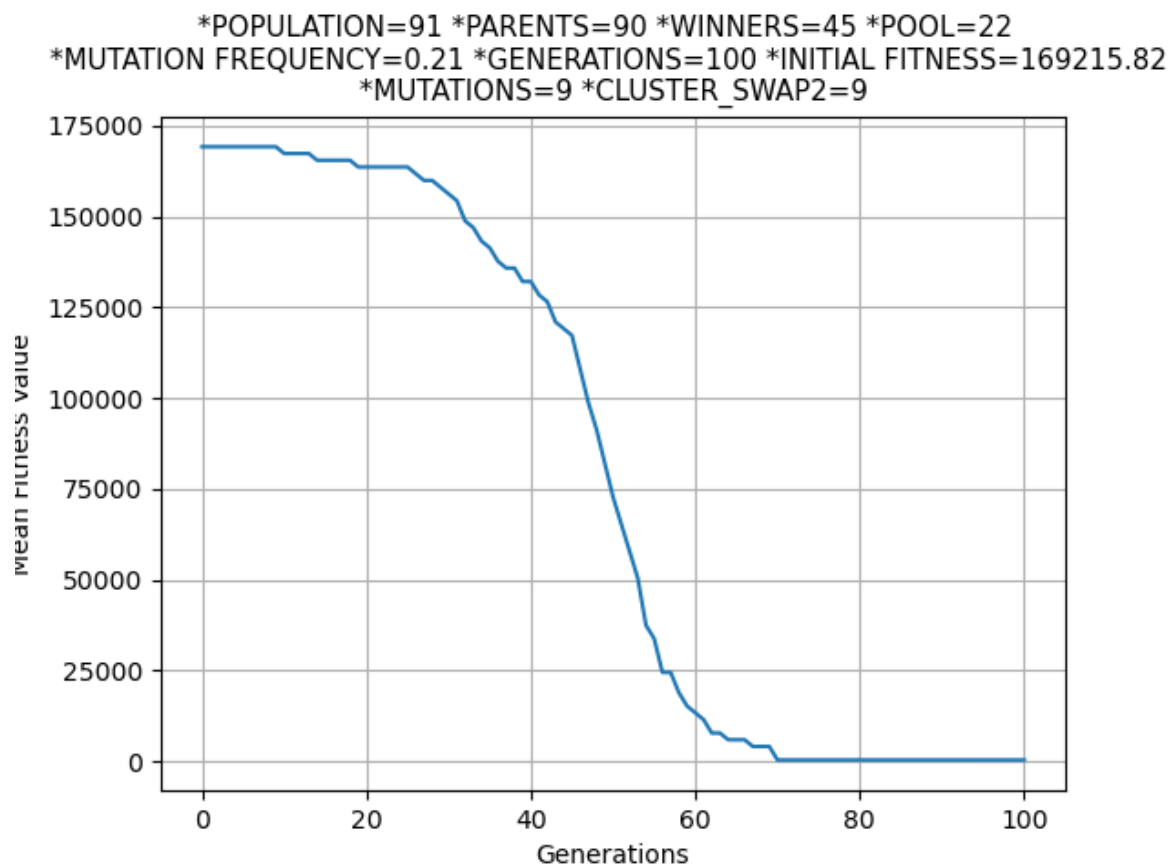


Εικόνα 71 Συντομότερη διαδρομή από κόμβο '100' προς κόμβο '101' με χρήση Αλγορίθμου Dijkstra για Γράφο με 102 κόμβους

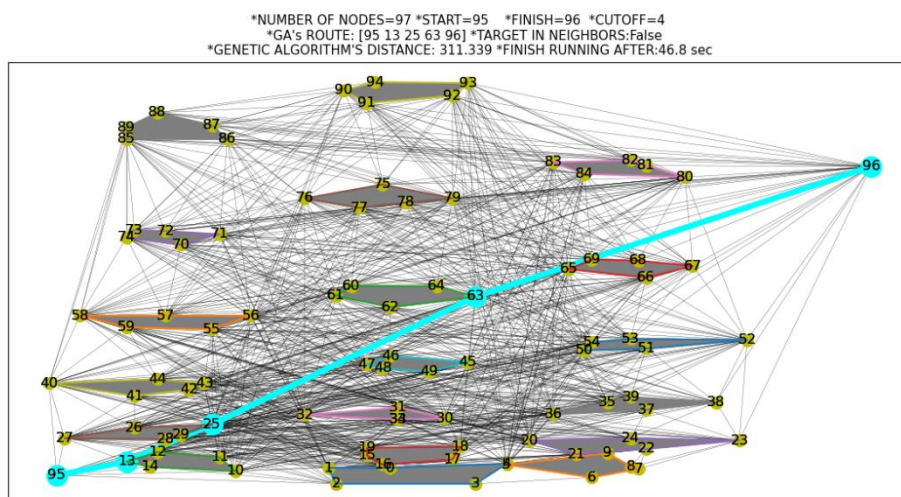
Στο σενάριο 1.10 όπως παρατηρούμε και στο Διάγραμμα 10 ο Γενετικός Αλγόριθμος #1 μέχρι την 100^η γενιά δεν έχει καταφέρει να βρει λύση. Παρόλα αυτά από την 101^η μέχρι την 150^η γενιά καταφέρνει με μόνο 2 μεταλλάξεις να βρει την ίδια λύση με τον Αλγόριθμο Dijkstra!

Σενάριο 1.11 - Γράφος Ορατότητας με 97 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	91
Αριθμός ζευγαριών/παιδιών	22
Μεταλλάξεις	9
Cluster_swap2	9
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	311.339
Μήκος μονοπατιού Dijkstra	310.596
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	46.8 sec

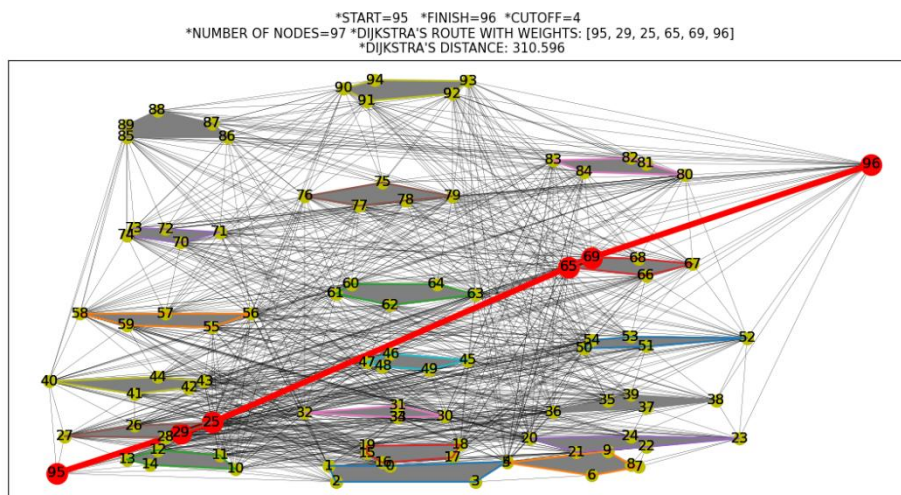
Πίνακας 13 Σενάριο 1.11



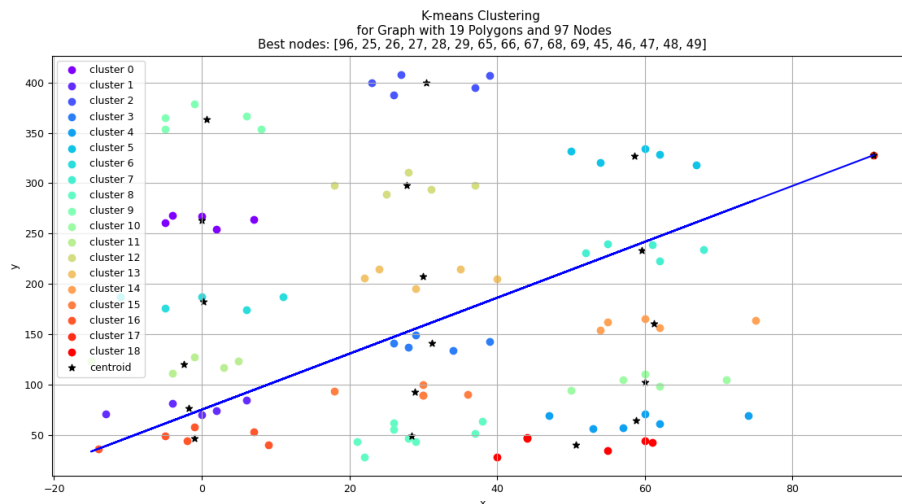
Διάγραμμα 11 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.11



Εικόνα 72 Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 97 κόμβους



Εικόνα 73 Συντομότερη διαδρομή από κόμβο '95' προς κόμβο '96' με χρήση Αλγορίθμου Dijkstra για Γράφο με 97 κόμβους

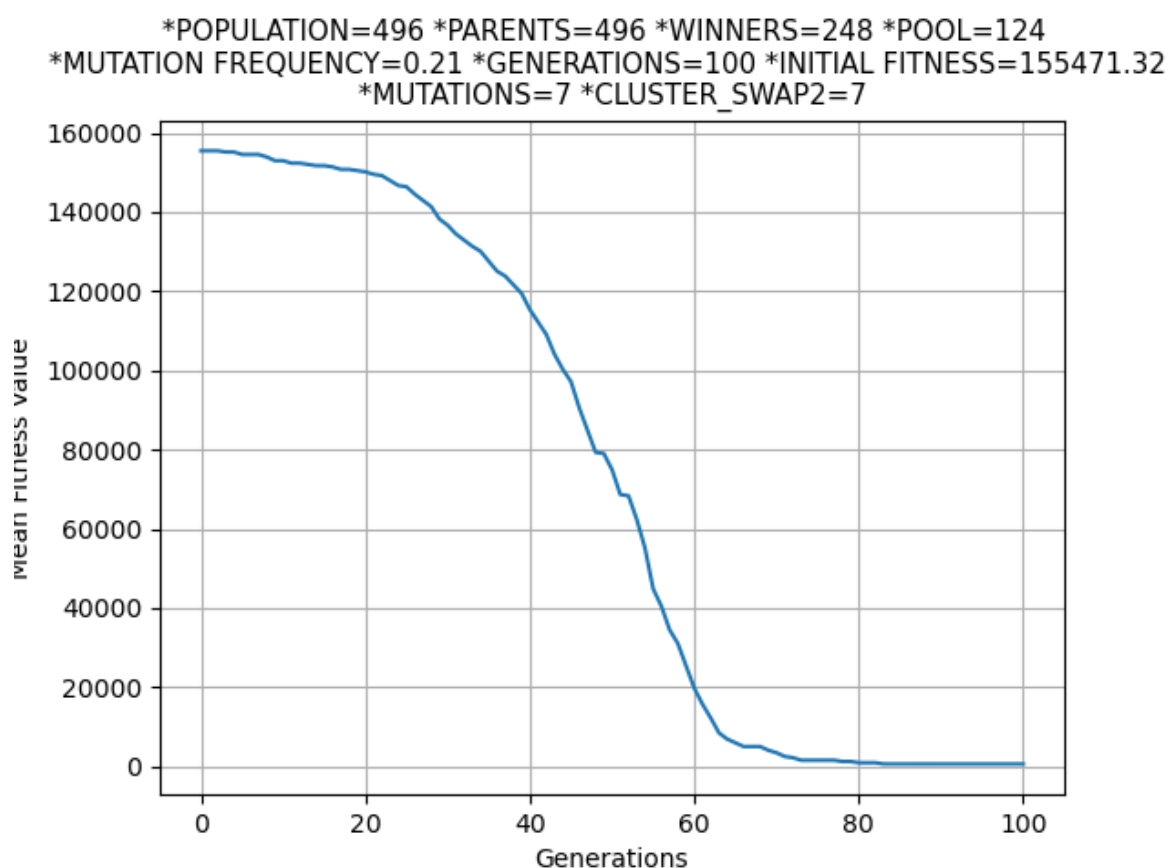


Εικόνα 74 Συσταδοποίηση 97 κόμβων και σχεδίαση ευθείας Γραμμικής Παλινδρόμησης

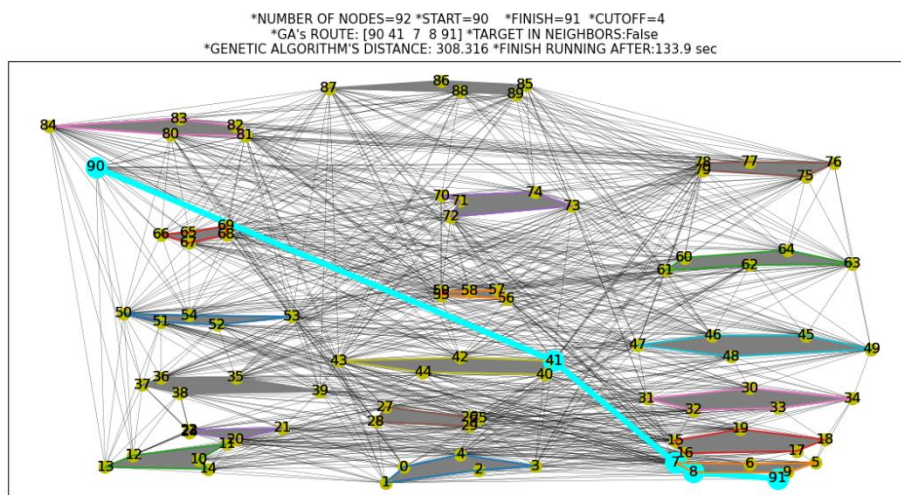
Στο σενάριο 1.11 χρησιμοποιούμε μόνο τον τύπο μετάλλαξης `cluster_swap2`. Παραθέτουμε εκτός από τα μονοπάτια που επιστρέφει κάθε αλγόριθμος και το γράφημα με την κατανομή των συστάδων που επιστρέφει ο αλγόριθμος K-means στον τίτλο του οποίου εμφανίζουμε και τον πίνακα με τους καλύτερους για επιλογή κόμβους. Υπενθυμίζουμε ότι το πλήθος των συστάδων που επιστρέφει ο αλγόριθμος επιλέξαμε να είναι ίσο με το πλήθος των πολυγώνων. Μια άλλη επιλογή θα ήταν η παράμετρος αυτή να επιλέγεται από το χρήστη. Σημειώνουμε ότι όσο μικρότερο πλήθος συστάδων επιλέξουμε τόσο περισσότεροι κόμβοι θα ανήκουν σε κάθε συστάδα, άρα τόσο μεγαλύτερη πιθανότητα δύο κόμβοι να ανήκουν στην ίδια συστάδα, επομένως η συνάρτηση `cluster_swap2()` θα προχωρά συνεχώς στο επόμενο παιδί, οπότε τελικά το πλήθος των μεταλλάξεων θα μειωθεί. Παρατηρούμε ότι ο Γενετικός Αλγόριθμος #1 συγκλίνει σε λύση μετά την 70^η γενιά περίπου. Παρόλα αυτά τα μήκη των μονοπατιών είναι περίπου ίσα ($311.339 \cong 310.596$) αν και αποτελούνται από διαφορετικούς κόμβους. Είναι αξιοσημείωτο ότι ο αλγόριθμος Dijkstra έχει επιστρέψει μονοπάτι του οποίου οι 4 ενδιάμεσοι κόμβοι πράγματι περιέχονται στον πίνακα `best_nodes[]`. Αντίθετα ο κόμβος '63' ο οποίος βρίσκεται στο μονοπάτι που έχει επιστρέψει ο Γενετικός Αλγόριθμος δεν βρίσκεται μέσα στον `best_nodes[]`. Αυτό συμβαίνει γιατί έχουμε επιλέξει μέσα στον `best_nodes[]` να βρίσκονται οι κόμβοι που ανήκουν σε τέσσερις μόνο συστάδες, άρα αν η μια από αυτές ανήκει εξ ολοκλήρου στον κόμβο-προορισμό (κόμβος '96'), όπως παρατηρούμε στο γράφημα, τότε απομένουν μόνο τρεις συστάδες, άρα και λιγότεροι κόμβοι.

Σενάριο 1.12 - Γράφος Ορατότητας με 92 κόμβους	
Αριθμός γενεών	100
Πληθυσμός	496
Αριθμός ζευγαριών/παιδιών	124
Μεταλλάξεις	7
Cluster_swap2	7
Συχνότητα μετάλλαξης	0.21
Μήκος μονοπατιού Γενετικού Αλγορίθμου	308.316
Μήκος μονοπατιού Dijkstra	308.317
Χρόνος εκτέλεσης Γενετικού Αλγορίθμου	133.9 sec

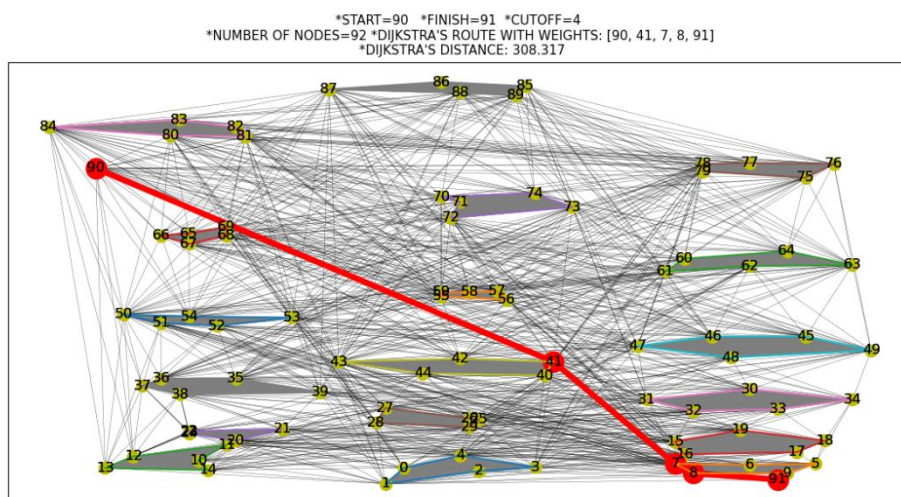
Πίνακας 14 Σενάριο 1.12



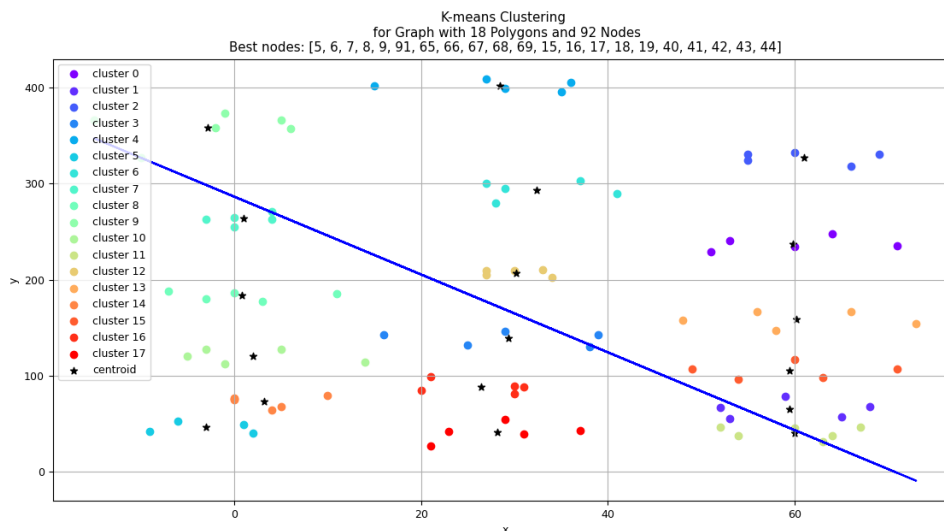
Διάγραμμα 12 Μεταβολή μέσης τιμής αντικειμενικής συνάρτησης για το Σενάριο 1.12



Εικόνα 75 Συντομότερη διαδρομή από κόμβο '90' προς κόμβο '91' με χρήση Γενετικού Αλγορίθμου #1 για Γράφο με 92 κόμβους



Εικόνα 76 Συντομότερη διαδρομή από κόμβο '90' προς κόμβο '91' με χρήση Αλγορίθμου Dijkstra για Γράφο με 92 κόμβους



Εικόνα 77 Συσταδοποίηση 92 κόμβων και σχεδίαση ευθείας Γραμμικής Παλινδρόμησης

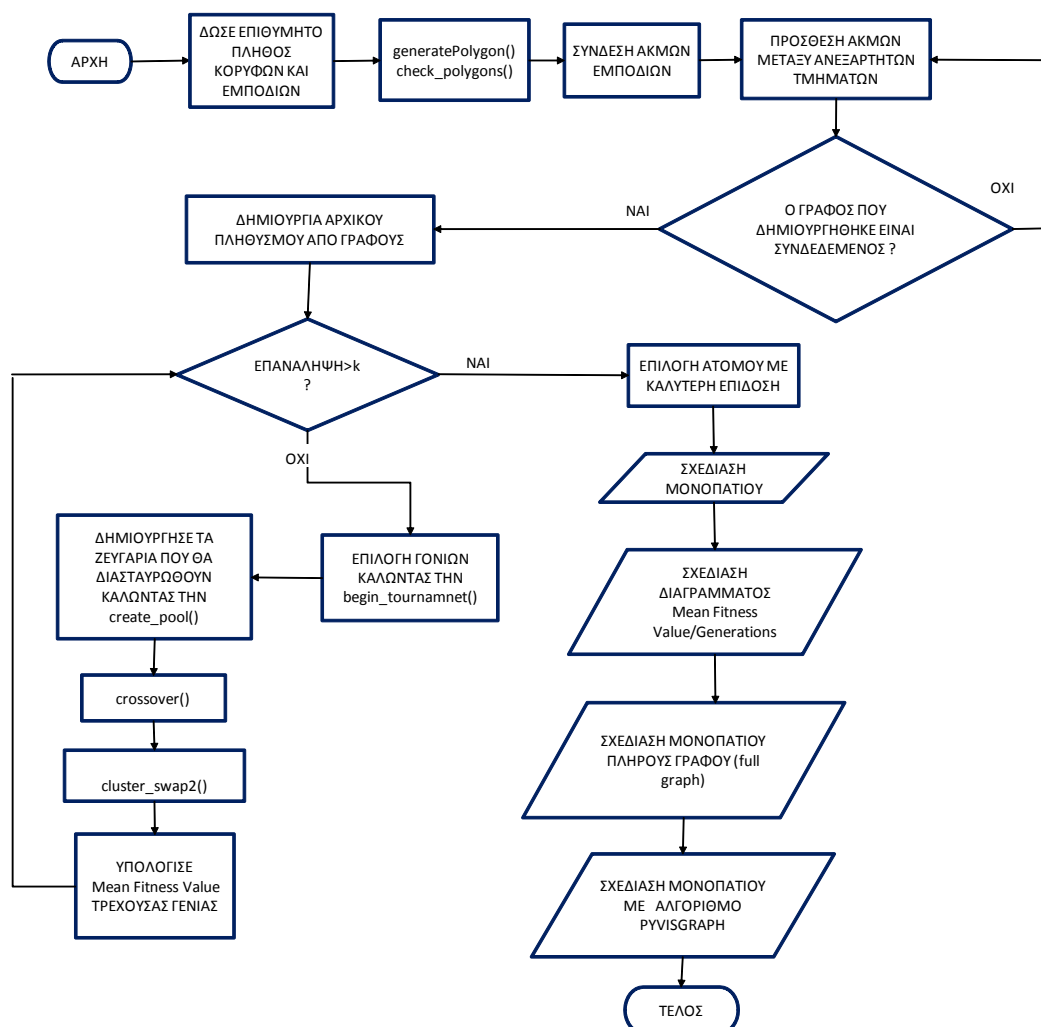
Στο σενάριο 1.12 οι δύο αλγόριθμοι καταλήγουν στο ίδιο βέλτιστο (global optimum) μονοπάτι [90, 41, 7, 8, 91]. Ο χρόνος εκτέλεσης του Γενετικού Αλγορίθμου #1 όμως είναι αυξημένος (133.9 sec) διότι ο αρχικός πληθυσμός αποτελείται από 496 άτομα, οπότε αντίστοιχα μεγάλο είναι και το πλήθος των ζευγαριών (124 ζευγάρια). Ο αρχικός πληθυσμός αυξήθηκε διότι ο κόμβος εκκίνησης (κόμβος '90') έχει τώρα περισσότερους γείτονες. Και οι τρεις ενδιάμεσοι κόμβοι του μονοπατιού ('41', '7', '8') βρίσκονται όπως ήταν αναμενόμενο μέσα στον πίνακα best_nodes[].

6. Σχεδίαση Γενετικού Αλγορίθμου #2

Σε αυτό το κεφάλαιο παρουσιάζουμε τον Γενετικό Αλγόριθμο #2. Υπενθυμίζουμε ότι στον αλγόριθμο αυτό, αφού δημιουργήσουμε έναν συνδεδεμένο γράφο, κατασκευάζουμε έναν πληθυσμό από γράφους ορατότητας και με τη βοήθεια γενετικών διαδικασιών δημιουργούμε καινούριους γράφους προσθέτοντας έγκυρες ακμές, δηλαδή ακμές που δεν τέμνουν τα εμπόδια. Επιδιώκουμε να δημιουργήσουμε ένα γράφο με τον ελάχιστο αριθμό ακμών που θα μας επιτρέψει να βρούμε μια διαδρομή η οποία θα προσεγγίζει τη βέλτιστη. Σε κάθε μέλος του πληθυσμού αποδίδουμε ως τιμή της αντικειμενικής συνάρτησης το μήκος του μονοπατιού που επιστρέφει ο αλγόριθμος Dijkstra. Επομένως η σύγκριση δύο μελών του πληθυσμού γίνεται ως προς την τιμή αυτή. Όσο μικρότερη η τιμή τόσο καλύτερη η λύση. Για την κατασκευή των πολυγωνικών εμποδίων χρησιμοποιούμε την ίδια μέθοδο όπως στον Αλγόριθμο #1. Όπως και πριν, οι συντεταγμένες των κορυφών των πολυγωνικών εμποδίων, αλλά και οι συντεταγμένες των κόμβων εκκίνησης-τερματισμού δεν είναι εκ των προτέρων γνωστές με αποτέλεσμα να αυξάνει ο βαθμός δυσκολίας υλοποίησης του Αλγορίθμου #2.

6.1 Σχεδίαση Αλγορίθμου #2

Ακολουθεί το διάγραμμα ροής για τον Αλγόριθμο #2:



Εικόνα 78 Διάγραμμα ροής Γενετικού Αλγορίθμου #2

6.2 Μέρος Πρώτο-Δημιουργία Ακμών Εμποδίων

Αφού κατασκευάσουμε τα πολυγωνικά εμπόδια όπως στον Γενετικό Αλγόριθμο #2 και έχοντας στη διάθεσή μας τις συντεταγμένες των κορυφών των εμποδίων, καθώς και του κόμβου εκκίνησης-τερματισμού, συνδέουμε τις κορυφές κάθε εμποδίου χρησιμοποιώντας τον παρακάτω κώδικα:

```

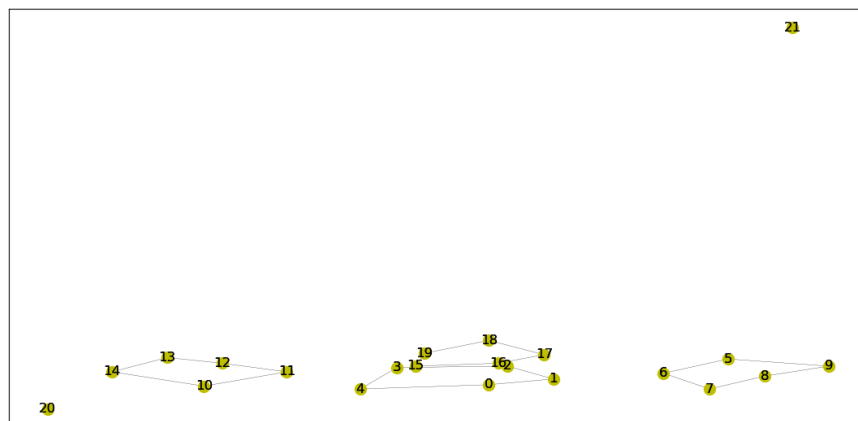
1      print("----- EXECUTE - CREATE GRAPH -----")
2
3      G = nx.Graph()
4
5      j = 0
6
7      v2 = 0
8
9      c1 = 0
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1
```

```

21         if my_target2 == key:
22             G.add_node(key)
23
24
25     print(" ")
26     print("-----IS CONNECTED -----")
27     print(nx.is_connected(G))
28     print(" ")

```

Αν σχεδιάσουμε το αποτέλεσμα που επιστρέφει ο παραπάνω κώδικας στην περίπτωση που έχουμε τέσσερα εμπόδια θα πάρουμε την παρακάτω εικόνα:



Εικόνα 79 Ασύνδετα πολυγωνικά εμπόδια

Όπως παρατηρούμε ο γράφος δεν είναι συνδεδεμένος. Αν εκτυπώσουμε τα στοιχεία από τα οποία αποτελείται έχουμε το παρακάτω αποτέλεσμα:

```

-----IS CONNECTED -----
False

```


-----COMPONENTS-----

{0, 1, 2, 3, 4}

{5, 6, 7, 8, 9}

{10, 11, 12, 13, 14}

{15, 16, 17, 18, 19}

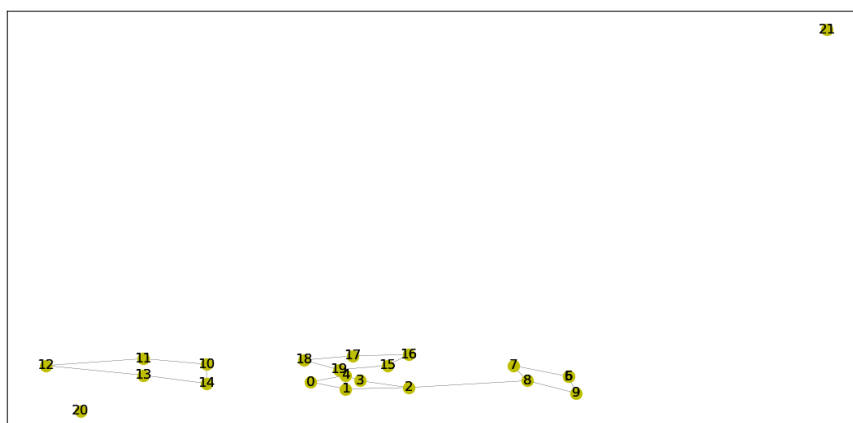
{20}

{21}

Ο γράφος αποτελείται από έξι ασύνδετα μέρη: τέσσερα πολυγωνικά εμπόδια και 2 μοναχικούς κόμβους.

6.3 Μέρος Δεύτερο-Πρόσθεση Ακμών Μεταξύ Εμποδίων

Αν στον παραπάνω ασύνδετο γράφο προσθέσουμε μια ακμή με έλεγχο ώστε να μην τέμνει κάποιο εμπόδιο θα πάρουμε το παρακάτω αποτέλεσμα:

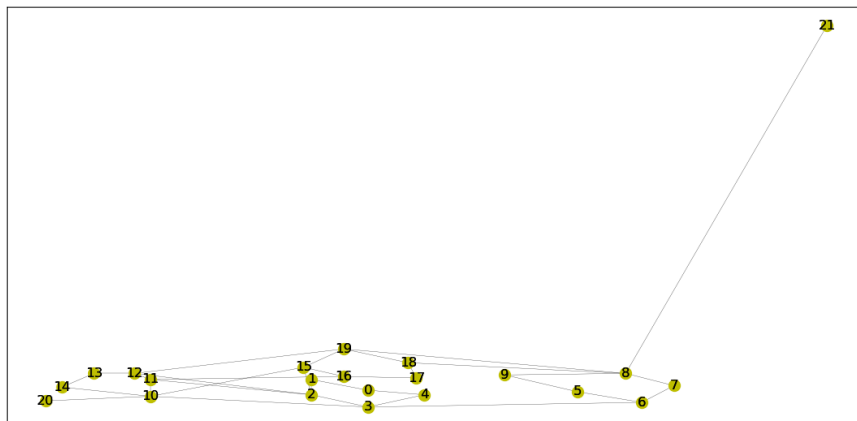


Εικόνα 80 Σύνδεση κόμβων διαφορετικών εμποδίων

Όπως παρατηρούμε στην παραπάνω εικόνα οι κόμβοι '2' και '8' οι οποίοι ανήκουν σε διαφορετικά εμπόδια είναι πλέον συνδεδεμένοι (δημιουργία ακμής 2-8). Το μεγαλύτερο ανεξάρτητο τμήμα αποτελείται πλέον από το σύνολο των κόμβων των δύο εμποδίων που ενώθηκαν με την ακμή 2-8 (κόμβοι '0', '1', '2', '3', '6', '7', '8', '9').

Στη συνέχεια φτιάχνουμε ένα βρόχο και προσθέτουμε ακμές έως ότου ο γράφος συνδεθεί¹² για πρώτη φορά, όπως φαίνεται στην παρακάτω εικόνα:

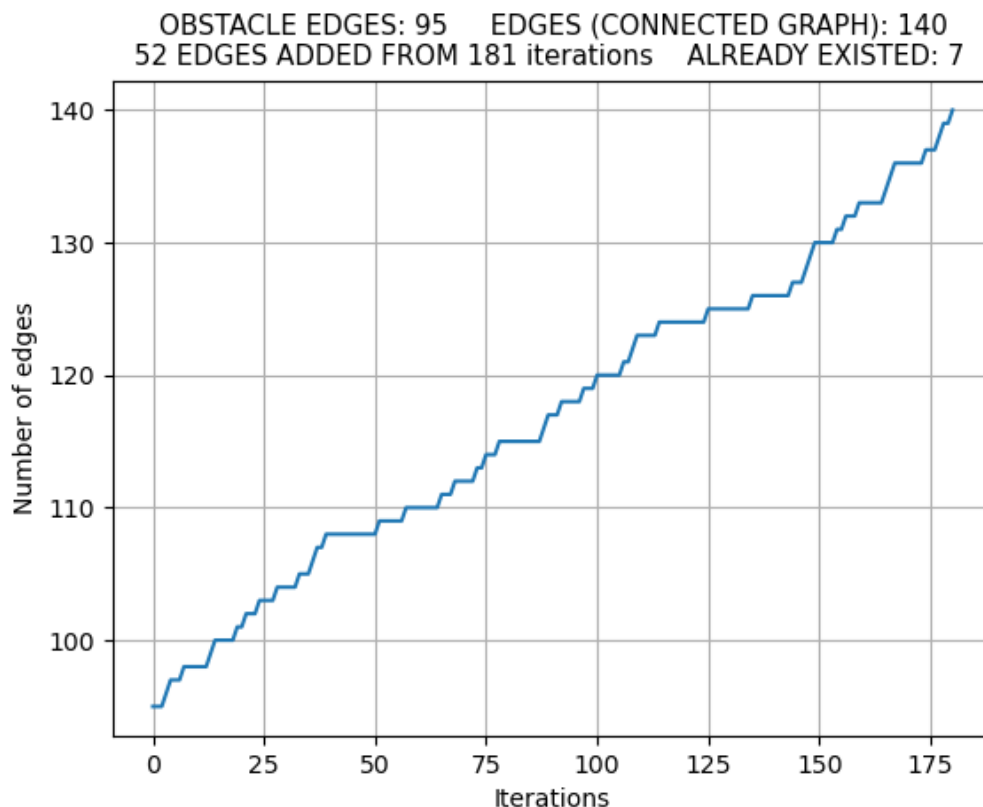
¹² Συνδεδεμένος λέγεται ο γράφος για τον οποίο για οποιοδήποτε ζεύγος κόμβων υπάρχει μονοπάτι που τους συνδέει.



Εικόνα 81 Συνδεδεμένος γράφος

Ας υποθέσουμε ότι έχουμε έναν γράφο με 19 πεντάγωνα. Με επιπλέον τους κόμβους εκκίνησης-τερματισμού, έχουμε συνολικά 97 κόμβους. Αν σχεδιάσουμε το διάγραμμα που μας δείχνει τον αριθμό των ακμών που προσθέτονται σε κάθε επανάληψη έως ότου ο γράφος συνδεθεί για πρώτη φορά¹³ θα πάρουμε το παρακάτω αποτέλεσμα:

¹³ Υπάρχουν πολλοί τρόποι σύνδεσης ενός γράφου.



Εικόνα 82 Διάγραμμα Number of edges/Iterations

Αρχικά έχουμε $5 \times 19 = 95$ ακμές¹⁴. Όπως παρατηρούμε χρειάστηκε να προστεθούν 52 ακμές ώστε ο γράφος να συνδεθεί για πρώτη φορά. Τελικά ο αρχικός γράφος που δημιουργήθηκε και ο οποίος είναι ο πρόγονος των μελών του αρχικού πληθυσμού, αποτελείται από 140 ακμές. Ο αριθμός αυτός είναι το:

$$\frac{140}{1081} \times 100 = 13\%$$

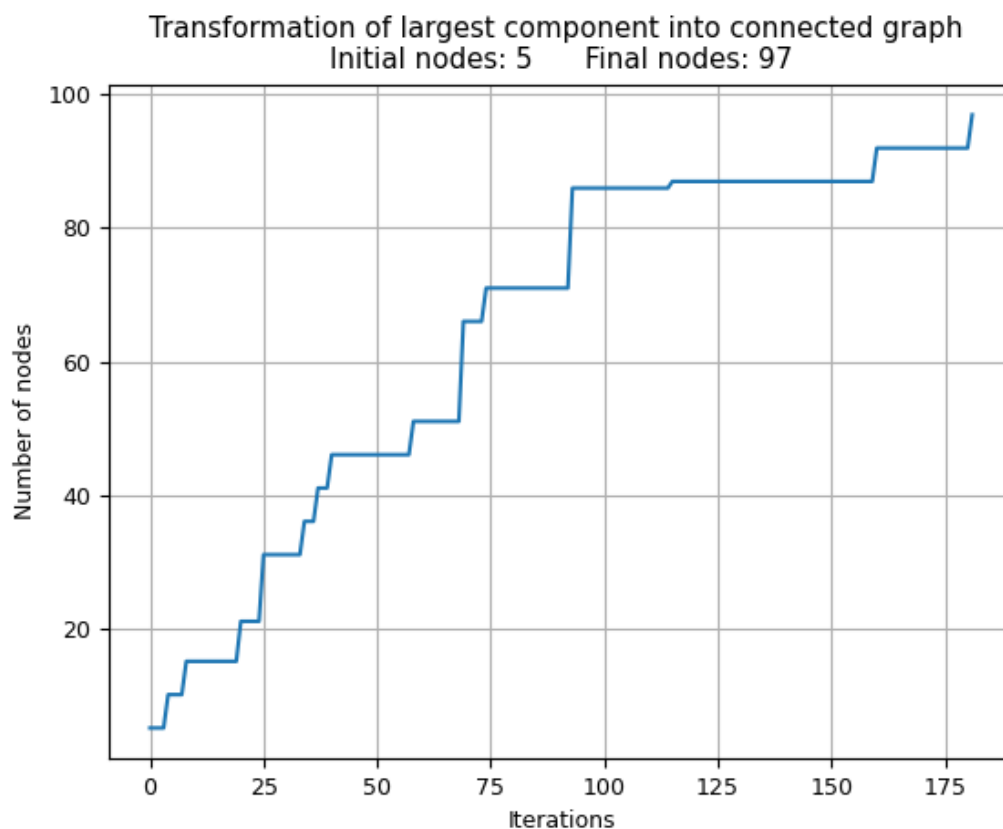
του συνόλου των ακμών του πλήρους γράφου ορατότητας¹⁵ όπου ο αριθμός 1081 αντιστοιχεί στις ακμές του πλήρους γράφου. Επίσης παρατηρούμε ότι χρειάστηκαν 181 επαναλήψεις για να συνδεθεί ο γράφος για πρώτη φορά.

Είδαμε ότι αρχικά έχουμε τόσα ασύνδετα μέρη όσα και ο αριθμός των εμποδίων (συν τους δύο κόμβους εκκίνησης-τερματισμού). Όσο προστίθενται ακμές, το μεγαλύτερο συσσωμάτωμα κόμβων (Largest Component) μετασχηματίζεται σε έναν ολοένα και μεγαλύτερο γράφο. Η διαδικασία τερματίζει όταν το συσσωμάτωμα αυτό γίνει

¹⁴ Κάθε πολύγωνο έχει τόσες ακμές όσες είναι και οι κορυφές.

¹⁵ Τη μέθοδο κατασκευής του πλήρους γράφου ορατότητας την περιγράψαμε αναλυτικά στον Γενετικό Αλγόριθμο #1.

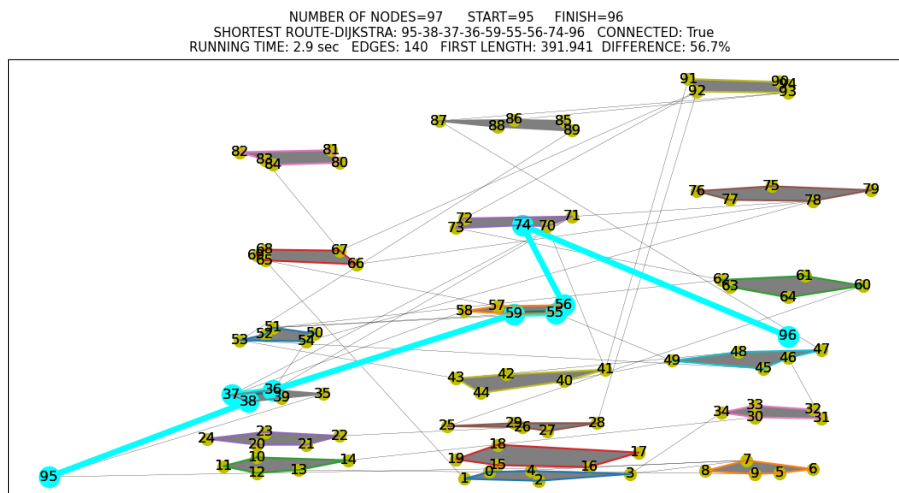
συνδεδεμένο. Στη συνέχεια σχεδιάζουμε το διάγραμμα που μας δείχνει αυτό το μετασχηματισμό:



Εικόνα 83 Διάγραμμα μετασχηματισμού ανεξάρτητου τμήματος σε συνδεδεμένο γράφο με 97 κόμβους.

Όπως παρατηρούμε το μεγαλύτερο μέρος των κόμβων (περίπου 85 κόμβοι) έχει συνδεθεί πριν την 100^η επανάληψη, ενώ χρειάστηκαν όπως είδαμε και προηγουμένως 181 επαναλήψεις ώστε ο γράφος να συνδεθεί για πρώτη φορά.

Αν σχεδιάσουμε το γράφο που προκύπτει καθώς και τη συντομότερη διαδρομή που επιστρέφει ο αλγόριθμος Dijkstra παίρνουμε το παρακάτω αποτέλεσμα:



Εικόνα 84 Αρχικός (συνδεδεμένος) γράφος και συντομότερη διαδρομή με χρήση αλγορίθμου Dijkstra

Παρατηρούμε ότι ο αρχικός γράφος κατασκευάστηκε σε χρόνο 2.9 sec, αποτελείται όπως είδαμε και παραπάνω από 140 ακμές, ενώ η ποσοστιαία διαφορά της τιμής της αντικειμενικής συνάρτησης του σε σχέση με τη βέλτιστη λύση είναι ίση με 56.7%.

6.4 Μέρος Τρίτο – Αρχικοποίηση Πληθυσμού

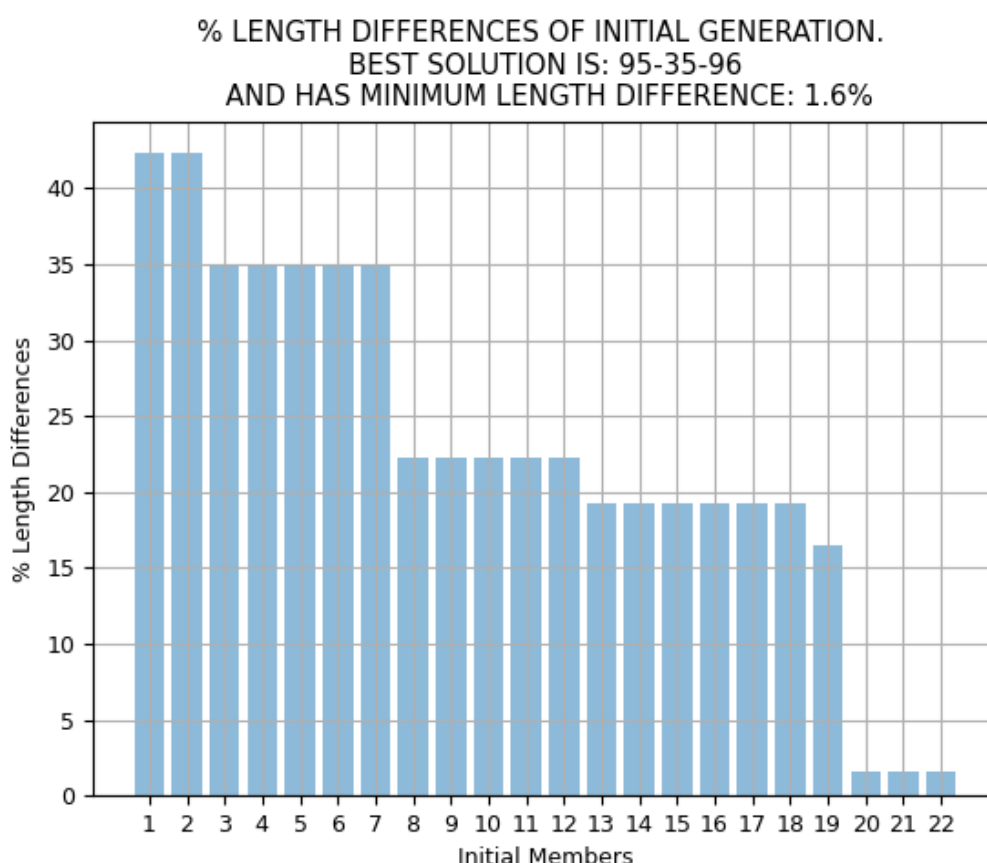
Στη συνέχεια φτιάχνουμε έναν πληθυσμό από γράφους. Ξεκινώντας από τον αρχικό συνδεδεμένο γράφο που κατασκευάσαμε στο δεύτερο μέρος, προσθέτουμε μια ακμή αν αυτή δεν τέμνει κάποιο εμπόδιο. Παρατηρούμε ότι:

1. Αφού ο αρχικός γράφος είναι συνδεδεμένος τότε θα είναι και οι υπόλοιποι.
2. Τα μέλη του πληθυσμού που προκύπτουν διαφέρουν ως προς μια ακμή, αλλιώς είναι ίδια.
3. Όλα τα μέλη είναι έγκυροι γράφοι ορατότητας.
4. Έστω και μία ακμή όμως μπορεί να δημιουργεί καλύτερο μονοπάτι.

Κατά την αρχικοποίηση του πληθυσμού επιδιώκουμε οι κόμβοι εκκίνησης-τερματισμού να έχουν συνδεθεί με όσο το δυνατόν περισσότερους γείτονές τους. Για αυτό το λόγο κατασκευάζουμε έναν πίνακα που να περιέχει τους δύο αυτούς κόμβους και στη συνέχεια με χρήση της συνάρτησης `random.sample()` επιλέγουμε έναν κόμβο από τους δύο (πιθανότητα 50%). Στη συνέχεια επιλέγουμε έναν οποιοδήποτε από τους υπολοίπους κόμβους του γράφου και κατασκευάζουμε ένα αντικείμενο τύπου `LineString`. Έπειτα ελέγχουμε αν η ακμή αυτή τέμνει κάποιο από τα εμπόδια. Αν όχι, την προσθέτουμε στον

αρχικό γράφο και υπολογίζουμε τη συντομότερη διαδρομή που επιστρέφει ο αλγόριθμος Dijkstra για το νέο γράφο.

Σημειώνουμε ότι όσο αυξάνει ο αριθμός των εμποδίων ο πληθυσμός γίνεται μικρότερος αφού οι μη έγκυρες ακμές είναι τώρα περισσότερες. Αν σχεδιάσουμε ένα ιστόγραμμα όπου στον άξονα x τοποθετούμε τα αριθμημένα μέλη του πληθυσμού και στον άξονα y την ποσοστιαία διαφορά μεταξύ της τιμής της αντικειμενικής συνάρτησης για κάθε μέλος και την τιμή που έχει η βέλτιστη λύση, παίρνουμε το παρακάτω αποτέλεσμα:



Εικόνα 85 Ποσοστιαίες διαφορές τιμών αντικειμενικής συνάρτησης για τα μέλη του αρχικού πληθυσμού σε σχέση με τη βέλτιστη λύση

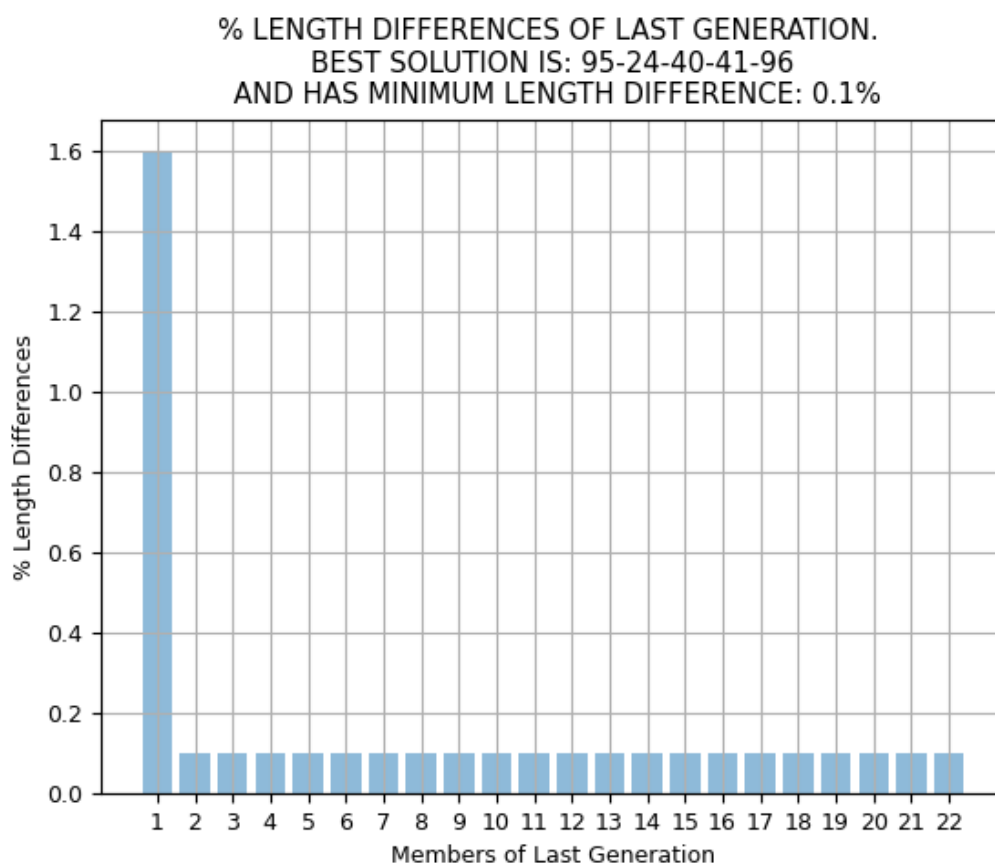
Παρατηρούμε ότι τα δύο πρώτα μέλη έχουν διαφορά σε σχέση με τη βέλτιστη λύση πάνω από 40%, τα επόμενα πέντε μέλη έχουν διαφορά ίση με 35%, τα επόμενα πέντε έχουν διαφορά ίση με περίπου 22%, τα επόμενα έξι μέλη έχουν διαφορά κάτω από 20%, ένα μέλος έχει διαφορά ίση με 16%, ενώ τρία μέλη έχουν διαφορά ίση με 1.6%. Μπορούμε λοιπόν να ισχυριστούμε ότι η λύση αυτή προσεγγίζει ικανοποιητικά τη βέλτιστη λύση.

Στόχος του Γενετικού Αλγορίθμου #2 είναι μέσω των γενετικών διαδικασιών, δηλαδή της επιλογής, της διασταύρωσης και της μετάλλαξης, να προσθέσουμε τον ελάχιστο αριθμό

ακμών ούτως ώστε ο γράφος που θα σχηματιστεί να οδηγεί σε διαδρομή η οποία θα προσεγγίζει τη βέλτιστη λύση την οποία εγγυάται ο αλγόριθμος Dijkstra, εφόσον όμως εφαρμοσθεί στον πλήρη γράφο ορατότητας (full graph).

Από το παραπάνω ιστόγραμμα είναι εμφανές ότι ο αλγόριθμος ενδέχεται να εγκλωβιστεί σε κάποιο τοπικό ελάχιστο, δηλαδή στις λύσεις που αντιπροσωπεύουν τα δύο πρώτα μέλη του αρχικού πληθυσμού (διαφορά 40%) ή στις λύσεις που αντιπροσωπεύουν τα επόμενα πέντε μέλη του αρχικού πληθυσμού (διαφορά 35%) ή στα λύσεις που αντιπροσωπεύουν τα επόμενα πέντε μέλη (διαφορά 22%), κ.ο.κ. Με την επιλογή κατάλληλης συχνότητας μετάλλαξης, επιδιώκουμε να αποφύγουμε αυτά τα εμπόδια.

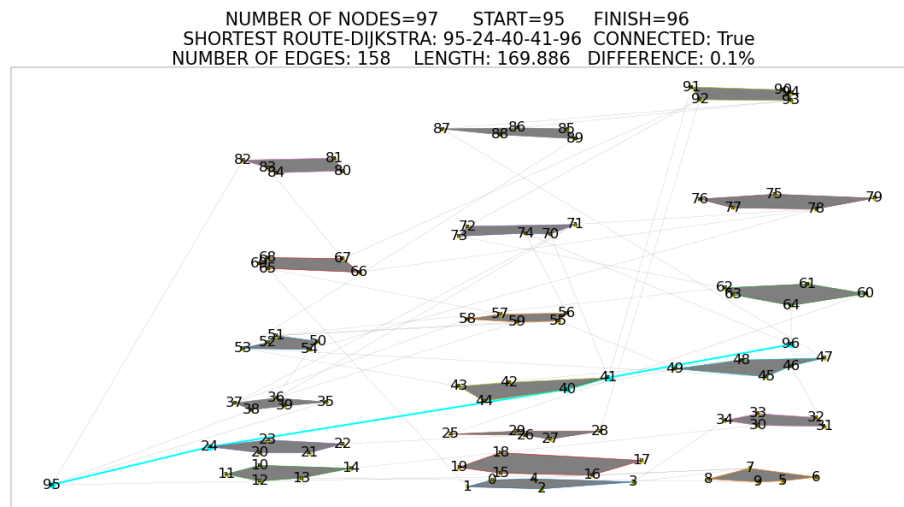
Μετά από ένα προκαθορισμένο αριθμό γενεών επιδιώκουμε η τελευταία γενιά να έχει τα χαρακτηριστικά που απεικονίζονται στο παρακάτω ιστόγραμμα:



Εικόνα 86 Ποσοστιαίες διαφορές τιμών αντικειμενικής συνάρτησης για τα μέλη του τελικού πληθυσμού σε σχέση με τη βέλτιστη λύση

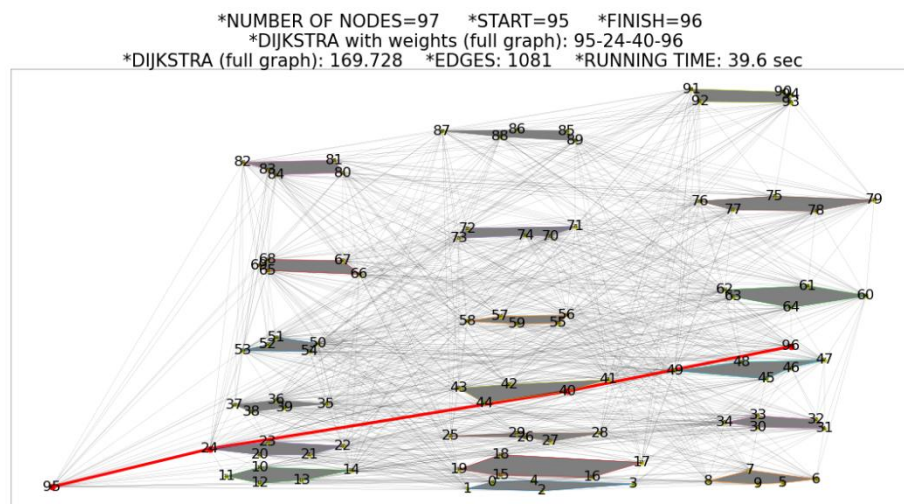
Παρατηρούμε ότι 21 από τα 22 μέλη του πληθυσμού έχουν τιμή αντικειμενικής συνάρτησης ίση με το 0.1% της βέλτιστης λύσης. Αυτή είναι λοιπόν και η λύση στην οποία καταλήγει ο Γενετικός Αλγόριθμος #2 και η οποία αντιστοιχεί στο μονοπάτι 95-24-

40-41-96. Αν σχεδιάσουμε τη διαδρομή με τη βοήθεια του πακέτου Matplotlib παίρνουμε το παρακάτω αποτέλεσμα:



Εικόνα 87 Λύση στην οποία καταλήγει ο Γενετικός Αλγόριθμος #2

Παρακάτω σχεδιάζουμε και τη βέλτιστη λύση που επιστρέφει ο αλγόριθμος Dijkstra εάν εφαρμοσθεί στον πλήρη γράφο ορατότητας:



Εικόνα 88 Βέλτιστη λύση (global optimum). Στον τίτλο αναφέρεται και ο χρόνος εκτέλεσης της απλοϊκής μεθόδου εύρεσης του πλήρους γράφου

Παρατηρούμε ότι ο Γενετικός Αλγόριθμος #2 χρειάστηκε να υπολογίσει μόνο 158 από τις 1081 ακμές του πλήρους γράφου για να βρει μια λύση που διαφέρει μόνο 0.1% σε σχέση με τη βέλτιστη. Επίσης παρατηρούμε ότι ο αλγόριθμος βελτιώνει το μήκος μονοπατιού του αρχικού γράφου κατά:

$$\frac{391.941 - 169.886}{391.941} \times 100 = 56.6\%$$

6.5 Μέρος Τέταρτο – Κύριος Βρόχος Επανάληψης

Όπως αναφέραμε και προηγουμένως, στόχος του Γενετικού Αλγορίθμου #2 είναι μέσω των γενετικών διαδικασιών να προσθέτει σταδιακά σε κάθε γενιά όχι μόνο τις πιο κατάλληλες ακμές, αλλά και τον ελάχιστο αριθμό αυτών έτσι ώστε η λύση στην οποία θα καταλήξει να προσεγγίζει τη βέλτιστη.

6.5.1 Επιλογή Γονιών (Parent Selection)

Αρχικά αποθηκεύουμε τα μέλη του πληθυσμού ως δισδιάστατους πίνακες (πίνακες γειτνίασης). Στη συνέχεια επιλέγουμε τόσους γονείς όσοι είναι το μέγεθος του πληθυσμού λαμβάνοντας μέτρα ώστε ο αριθμός αυτός να είναι άρτιος.

6.5.2 Tournament

Όπως και στον Γενετικό Αλγόριθμο #1, οι γονείς από τους οποίους θα προέλθουν οι νέες λύσεις επιλέγονται καλώντας τη συνάρτηση `begin_tournament()`. Η συνάρτηση αυτή συγκρίνει ανά δύο τις τιμές των αντικειμενικών συναρτήσεων των μελών και επιλέγονται εκείνοι οι γονείς-γράφοι των οποίων τα μονοπάτια είναι τα συντομότερα.

6.5.3 Διασταύρωση (crossover)

Συνεχίζοντας στην επόμενη γενετική διαδικασία (διασταύρωση) παρατηρούμε το εξής: για να διατηρείται η ιδιότητα ο γράφος να παραμένει συνδεδεμένος, αλλά και οι ακμές να είναι έγκυρες θα πρέπει να χρησιμοποιήσουμε το λογικό OR (εντολή `np.logical_or()`):

A	B	A OR B
1	0	1
0	1	1
1	1	1
0	0	0

Με αυτό τον τρόπο κατά τη διασταύρωση δύο γονιών :

1. εκεί όπου υπήρχε ακμή μόνο από τον ένα γονέα αυτή διατηρείται,
2. εκεί όπου είχαν ακμή και οι δύο αυτή διατηρείται,
3. εκεί όπου κανένας γονέας δεν είχε ακμή **δεν δημιουργούμε καινούρια** διότι δεν γνωρίζουμε αν ο προκύπτων γράφος ορατότητας θα είναι έγκυρος (ακμές μόνο μεταξύ visible vertices).

6.5.4 Μετάλλαξη (Mutation)

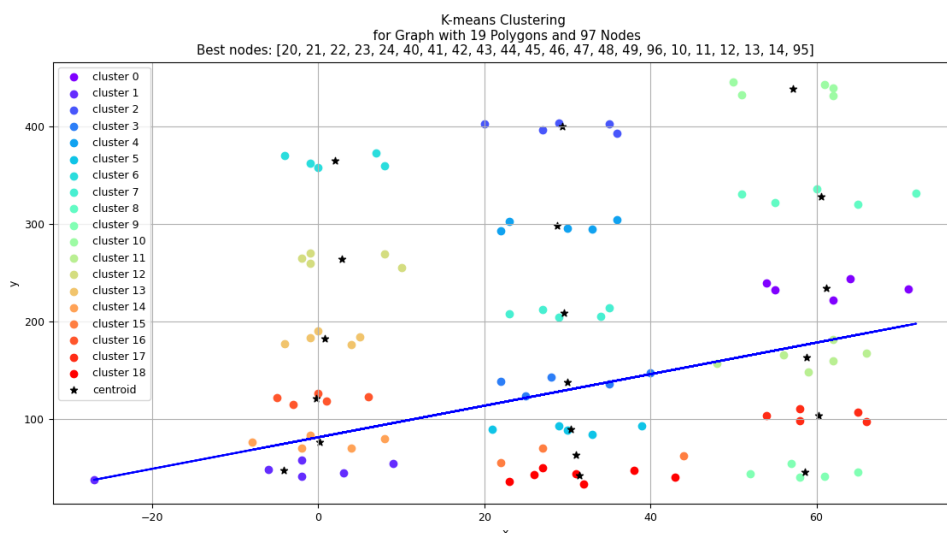
Στο στάδιο αυτό διαφοροποιούμαστε σε σχέση με την προσέγγιση που ακολουθήσαμε στον Γενετικό Αλγόριθμο #1. Ο λόγος είναι ότι αν χρησιμοποιήσουμε την ενσωματωμένη συνάρτηση `nx.double-edge swap()` της βιβλιοθήκης Networkx της Python ενδέχεται να προκύψουν μη έγκυρες ακμές αφού:

$u-v$	becomes	u	v
$x-y$		x	y

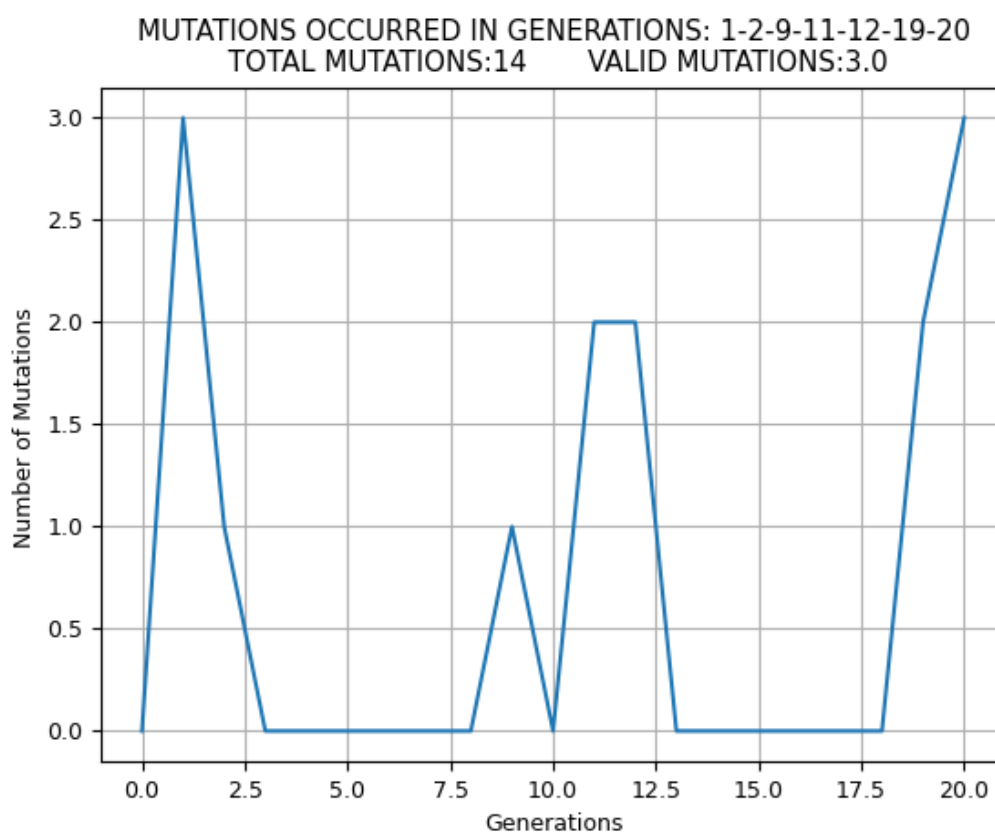
Εικόνα 89 Εντολή `double-edge swap()` (πηγή: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.swas.double_edge_swap.html)

Δηλαδή οι νέες ακμές $u-x$ και $v-y$ ενδέχεται να τέμνουν κάποιο από τα εμπόδια. Χρησιμοποιούμε λοιπόν μόνο τη συνάρτηση `cluster_swap2()`, την οποία χρησιμοποιήσαμε και στον Γενετικό Αλγόριθμο #1. Η συνάρτηση αυτή εφαρμόζει συσταδοποίηση K-Means στους κόμβους του γράφου, ενώ στη συνέχεια υπολογίζει την ευθεία γραμμικής παλινδρόμησης που συνδέει αρχικό και τελικό κόμβο. Αποθηκεύει στη συνέχεια στον πίνακα `best_nodes[]` τους κόμβους εκείνους που ανήκουν στις συστάδες που βρίσκονται πιο κοντά στην ευθεία γραμμικής παλινδρόμησης. Στη συνέχεια, επιλέγει δύο τυχαίους κόμβους οι οποίοι όμως ανήκουν σε διαφορετικές συστάδες. Μετά τη μετάλλαξη ελέγχουμε αν η νέα ακμή τέμνει κάποιο εμπόδιο. Κατά το στάδιο αυτό αποθηκεύουμε τον αριθμό των μεταλλάξεων σε κάθε γενιά, τον αριθμό αυτών οι οποίες ήταν τελικά έγκυρες καθώς και τις νέες ακμές (σε ιστόγραμμα) οι οποίες προστέθηκαν στα μέλη. Αν

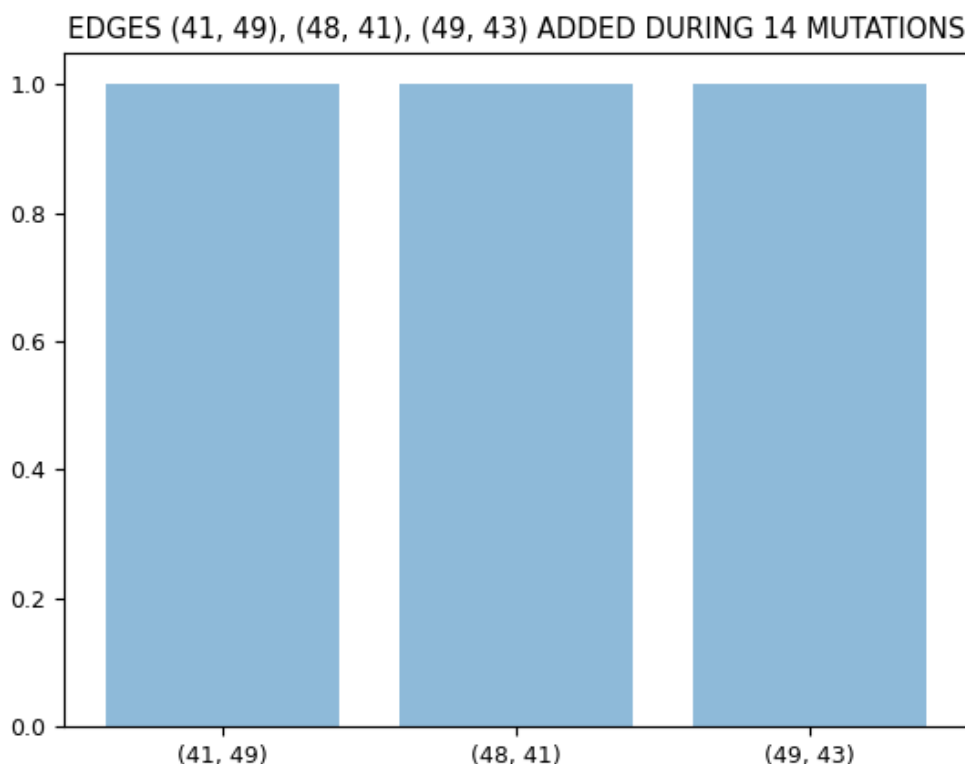
σχεδιάσουμε τα παραπάνω μαζί με το διάγραμμα συσταδοποίησης K-Means παίρνουμε τα ακόλουθα διαγράμματα για το γράφο με τους 97 κόμβους που εξετάζουμε στο παράδειγμα:



Εικόνα 90 Συσταδοποίηση 97 κόμβων και σχεδίαση ευθείας γραμμικής παλινδρόμησης



Εικόνα 91 Πλήθος μεταλλάξεων που συνέβησαν σε κάθε γενιά. Στον τίτλο καταχωρούμε επίσης πόσες από αυτές ήταν έγκυρες καθώς και την γενιά στην οποία συνέβησαν.



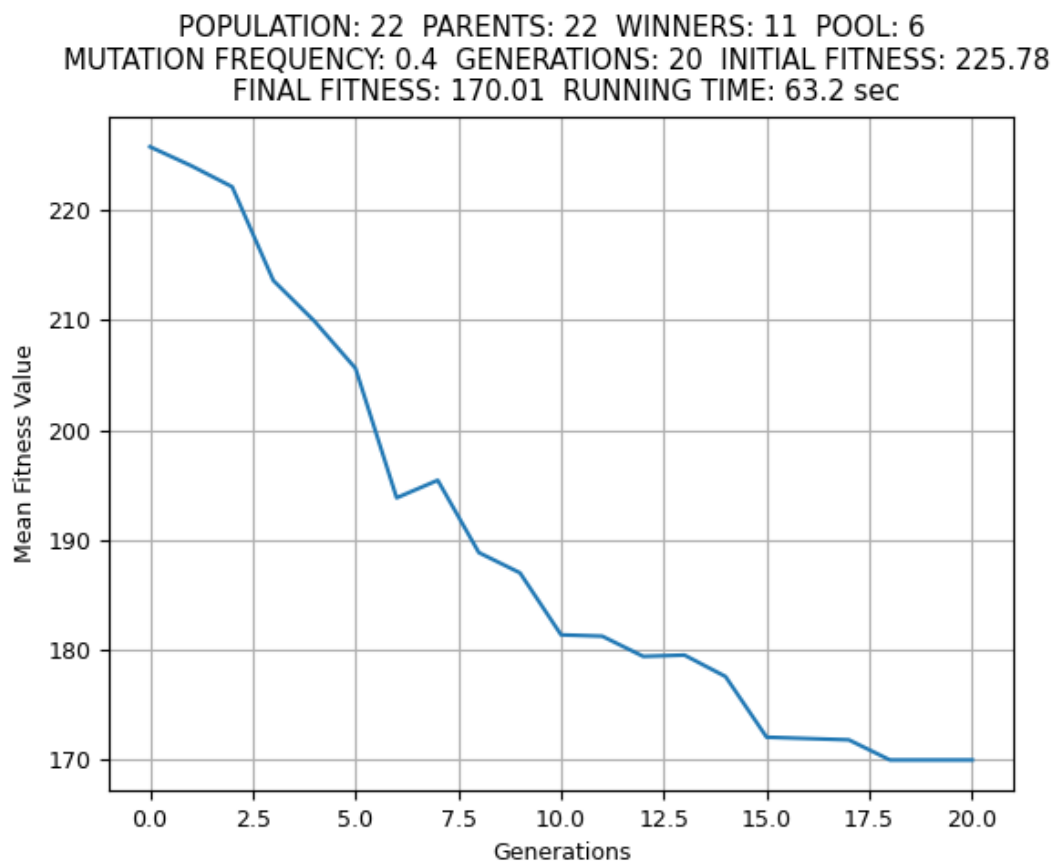
Εικόνα 92 Ιστόγραμμα που απεικονίζει: στον άξονα x τις έγκυρες ακμές οι οποίες προστέθηκαν κατά τη διάρκεια των μεταλλάξεων και στον άξονα y το πόσες φορές προστέθηκε η καθεμία.

Αν παρατηρήσουμε είτε την Εικόνα 90 είτε την Εικόνα 91 θα διαπιστώσουμε ότι πράγματι οι κόμβοι των ακμών 41-49, 48-41 και 49-43 ανήκουν σε διαφορετικές συστάδες. Παρόλα αυτά καμία από τις παραπάνω ακμές δεν αποτελεί ακμή της τελικής διαδρομής που υπολογίζει ο Γενετικός Αλγόριθμος #2. Αυτό σημαίνει ότι ενώ σε κάποια γενιά υπήρχαν μέλη με αυτά τα χαρακτηριστικά (εμφάνιση δηλαδή των συγκεκριμένων ακμών στους γράφους), λόγω του ότι δεν προσέφεραν στα μέλη που τα κατείχαν κάποιο πλεονέκτημα έναντι των ανταγωνιστών τους¹⁶, τελικά τα περισσότερα από αυτά δεν κατάφεραν να επιβιώσουν κατά το στάδιο της επιλογής. Τονίζουμε το γεγονός ότι αν παρατηρήσουμε την Εικόνα 86, τα 21 από τα 22 μέλη του πληθυσμού της τελευταίας γενιάς είναι όλα ίδια άρα κανένα από αυτά τα μέλη δεν διαθέτει τα συγκεκριμένα χαρακτηριστικά.

¹⁶ Τα συγκεκριμένα χαρακτηριστικά δεν οδηγούν σε συντομότερα μονοπάτια, άρα δεν μειώνουν την τιμή της αντικειμενικής συνάρτησης των μελών που τα κατέχουν.

6.5.5 Υπολογισμός Μέσου όρου Αντικειμενικής Συνάρτησης και Μέσου όρου Πλήθους Ακμών

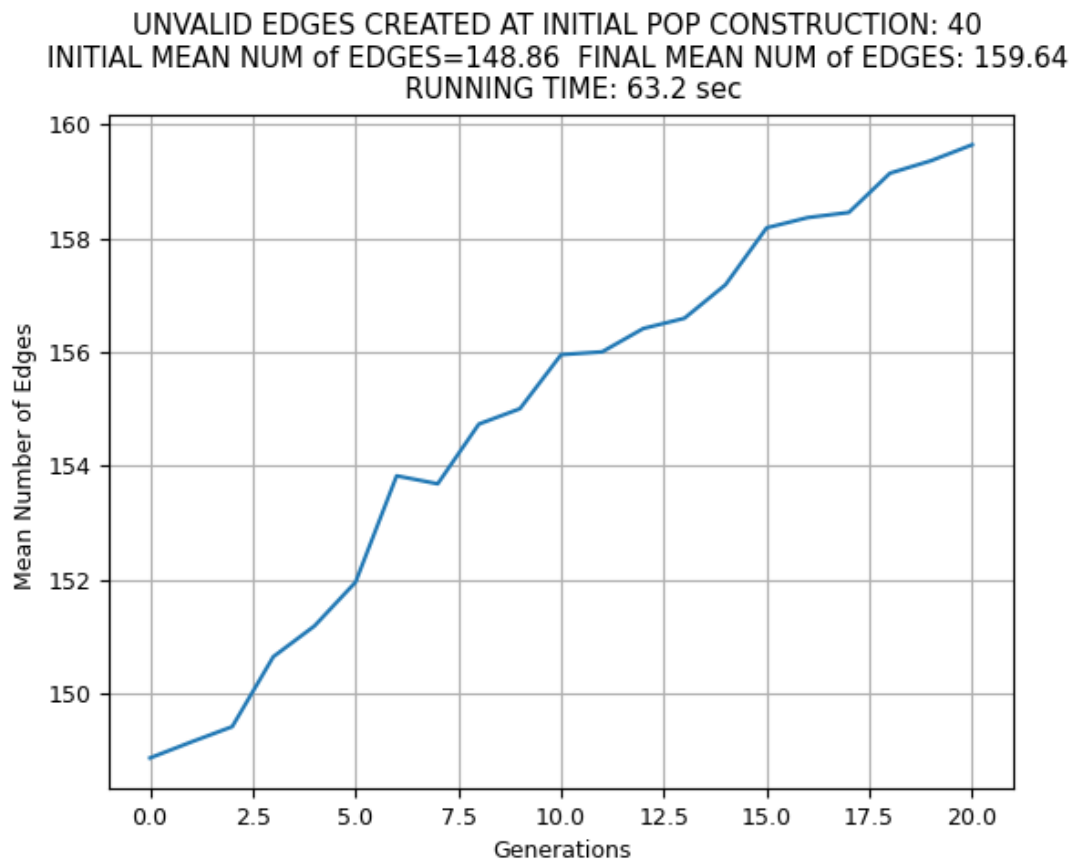
Σε κάθε γενιά υπολογίζουμε το μέσο όρο των μηκών των συντομότερων μονοπατιών καθώς και το μέσο όρο των ακμών από τις οποίες αποτελούνται οι γράφοι κάθε γενιάς έτσι ώστε να διαπιστώσουμε πώς μεταβάλλονται αυτά τα ποσά. Αν σχεδιάσουμε τα αντίστοιχα διαγράμματα για το παράδειγμα του γράφου με 97 κόμβους με το οποίο ασχολούμαστε παίρνουμε τα παρακάτω αποτελέσματα:



Εικόνα 93 Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης

Παρατηρούμε ότι πράγματι ο μέσος όρος (170.01) της τιμής της αντικειμενικής συνάρτησης της τελευταίας γενιάς προσεγγίζει το μήκος του μονοπατιού (169.886) της λύσης την οποία επιστρέφει ο Γενετικός Αλγόριθμος #2. Επίσης παρατηρούμε ότι η ποσοστιαία μεταβολή του μέσου όρου είναι:

$$\frac{225.78 - 170.01}{225.78} \times 100 = 24.7\%$$



Εικόνα 94 Διάγραμμα μεταβολής μέσου όρου ακμών

Παρατηρούμε ότι ο αρχικός μέσος όρος ακμών των μελών ήταν 149 ακμές, ενώ ο μέσος όρος των ακμών των μελών της τελευταία γενιάς είναι 160 ακμές. Συμπερασματικά σε διάστημα 20 γενεών προστέθηκαν στα μέλη του πληθυσμού 11 ακμές. Ο αριθμός αυτός ήταν αρκετός ώστε η λύση στην οποία κατέληξε ο Γενετικός Αλγόριθμος #2 να προσεγγίζει τη βέλτιστη (διαφορά 0.1%).

Στον τίτλο προσθέσαμε και το χρόνο εκτέλεσης του Γενετικού Αλγορίθμου #2. Παρατηρούμε ότι αν και ο αλγόριθμος υπολόγισε λιγότερες ακμές (160) από αυτές του πλήρους γράφου ορατότητας (1081), παρόλα αυτά ο χρόνος εκτέλεσης (63.2 sec) είναι μεγαλύτερος από το χρόνο εκτέλεσης (39.6 sec) της απλοϊκής μεθόδου (naïve method). Επίσης στον τίτλο αναφέρεται και το πλήθος των μη έγκυρων ακμών που εντοπίστηκαν κατά την αρχικοποίηση του πληθυσμού. Όσο περισσότερες μη έγκυρες ακμές εντοπίζονται, τόσο περισσότερο καθυστερεί ο αλγόριθμος.

6.6 Μέρος Πέμπτο – Έξοδος

Κατά την έξοδο του προγράμματος εκτός από τα παραπάνω διαγράμματα παρουσιάζουμε και το μονοπάτι που υπολογίζεται μέσω του Pyvisgraph-Python Visibility Graph (<https://github.com/TaipanRex/pyvisgraph/blob/master/README.md>, 2021). Το Pyvisgraph είναι ένα πακέτο Python με άδεια MIT για τη δημιουργία γράφων ορατότητας όταν δίνεται ως είσοδος μια λίστα από πολυγωνικά εμπόδια. Ο αλγόριθμος κατασκευής του γράφου ορατότητας τρέχει σε χρόνο $O(n^2 \log n)$ (Lee, 1978). Η συντομότερη διαδρομή βρίσκεται χρησιμοποιώντας τον αλγόριθμο Dijkstra.

7. Παρουσίαση Αποτελεσμάτων Γενετικού Αλγορίθμου #2

Σε αυτό το κεφάλαιο, αφού αναφερθούμε στην παραμετροποίηση του Γενετικού Αλγορίθμου #2, θα παρουσιάσουμε τα αποτελέσματα από την εκτέλεσή του για διαφορετικό πλήθος κόμβων.

7.1 Παραμετροποίηση

Κατά την εκκίνηση του αλγορίθμου οι βασικές παράμετροι που πρέπει να ρυθμιστούν είναι οι εξής:

- πλήθος των κορυφών του κάθε πολυγώνου,
- πλήθος των εμποδίων,
- την πιθανότητα να γίνει μετάλλαξη.

7.2 Μελέτη Αποτελεσμάτων Γενετικού Αλγορίθμου #2

Παρακάτω παρουσιάζουμε την επίδοση του Γενετικού Αλγορίθμου #2 για διάφορα σενάρια. Σε κάθε σενάριο παρουσιάζουμε το διάγραμμα μεταβολής του μέσου όρου τιμών της αντικειμενικής συνάρτησης, το διάγραμμα μεταβολής του μέσου όρου ακμών, το πλήθος των μεταλλάξεων, τον αρχικό γράφο, τις ποσοστιαίες διαφορές των τιμών αντικειμενικής συνάρτησης των μελών του αρχικού και τελικού πληθυσμού σε σχέση με τη βέλτιστη λύση. Τέλος, σε κάθε σενάριο συγκρίνουμε τις διαδρομές και το χρόνο εκτέλεσης των τριών ακόλουθων αλγορίθμων:

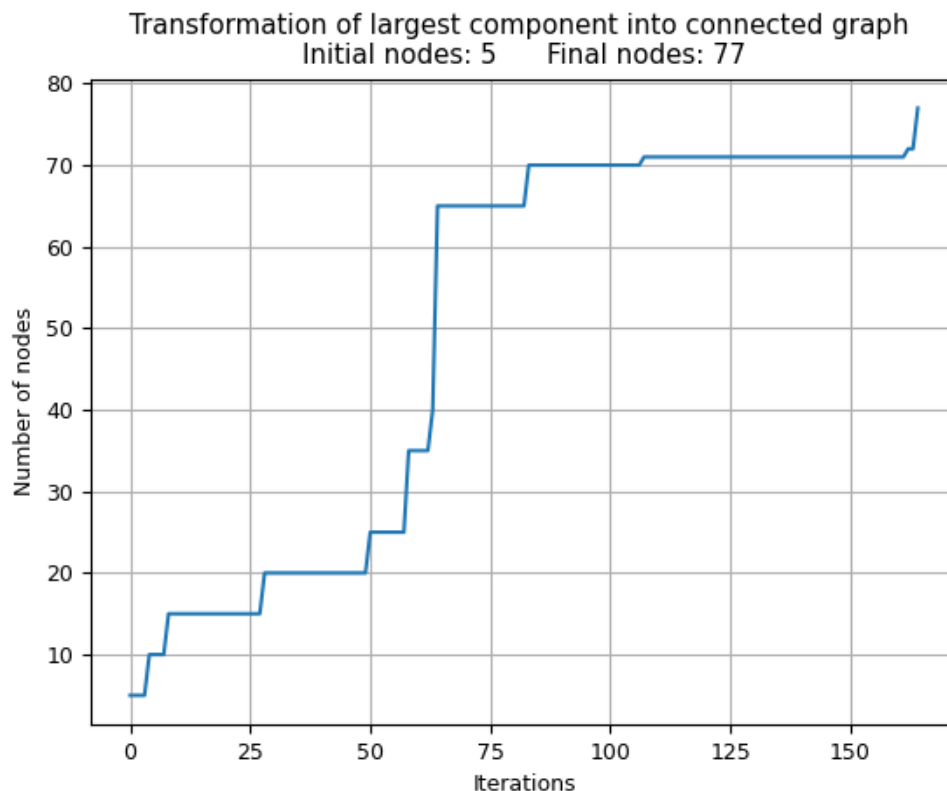
- Γενετικός Αλγόριθμος #2,
- απλοϊκή μέθοδος (naïve method)¹⁷,
- αλγόριθμος Lee¹⁸.

¹⁷ Πολυπλοκότητα $O(n^3)$

¹⁸ (“An $O(n^2 \log n)$ Algorithm for Computing Visibility Graphs”, 2020)

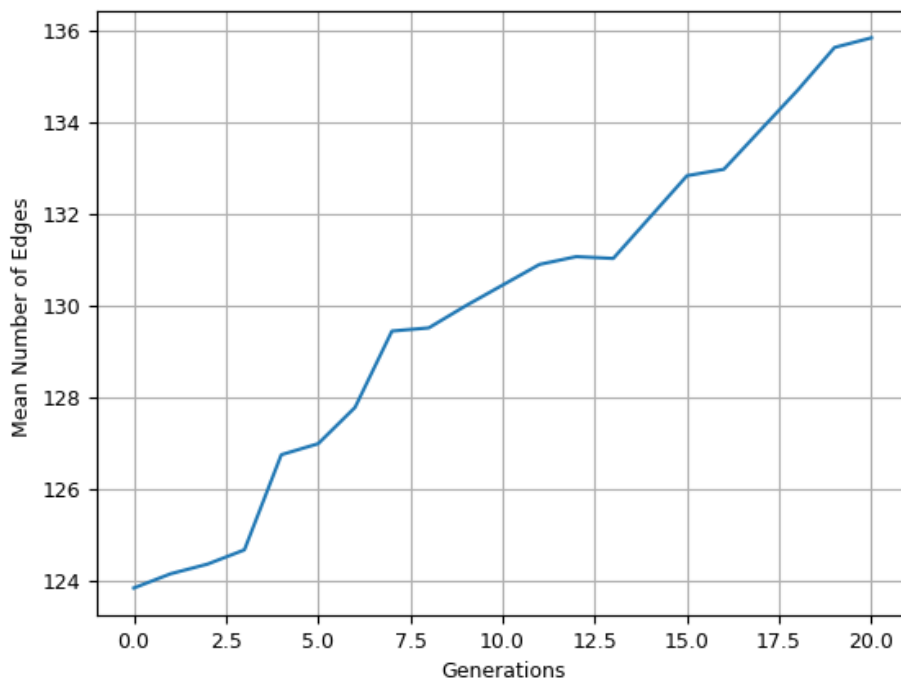
Σενάριο 2.1- Γράφος Ορατότητας με 77 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	29
Συνολικές μεταλλάξεις/Εγκυρες	11/1
Συχνότητα μετάλλαξης	0.3
Διαφορά αποστάσεων αρχικού γράφου	32%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	4.5%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	0.4%
Συνολικές ακμές (Γενετικός Αλγ. #2)	135
Συνολικές ακμές (Full Graph)	763
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	189.491
Μήκος μονοπατιού (Full Graph)	188.682
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	63.9 sec
Χρόνος εκτέλεσης (Full Graph)	19.9 sec
Χρόνος εκτέλεσης (Lee)	0.5 sec

Πίνακας 15 Σενάριο 2.1



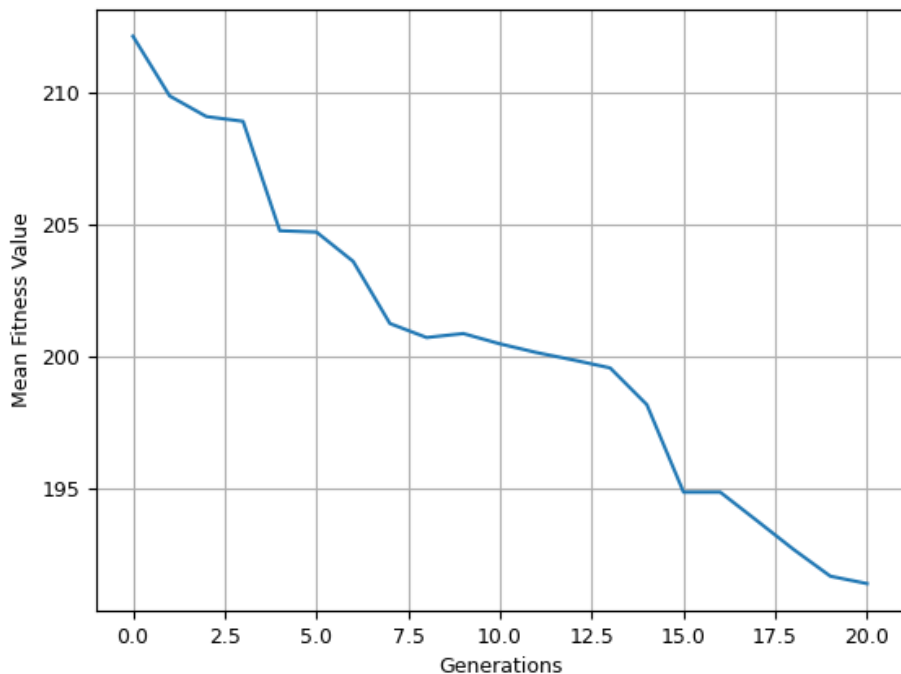
Εικόνα 95 Σενάριο 2.1-Πλήθος επαναλήψεων έως όπου ο γράφος συνδεθεί για πρώτη φορά.

UNVALID EDGES CREATED AT INITIAL POP CONSTRUCTION: 40
INITIAL MEAN NUM of EDGES=123.86 FINAL MEAN NUM of EDGES: 135.83
RUNNING TIME: 63.9 sec

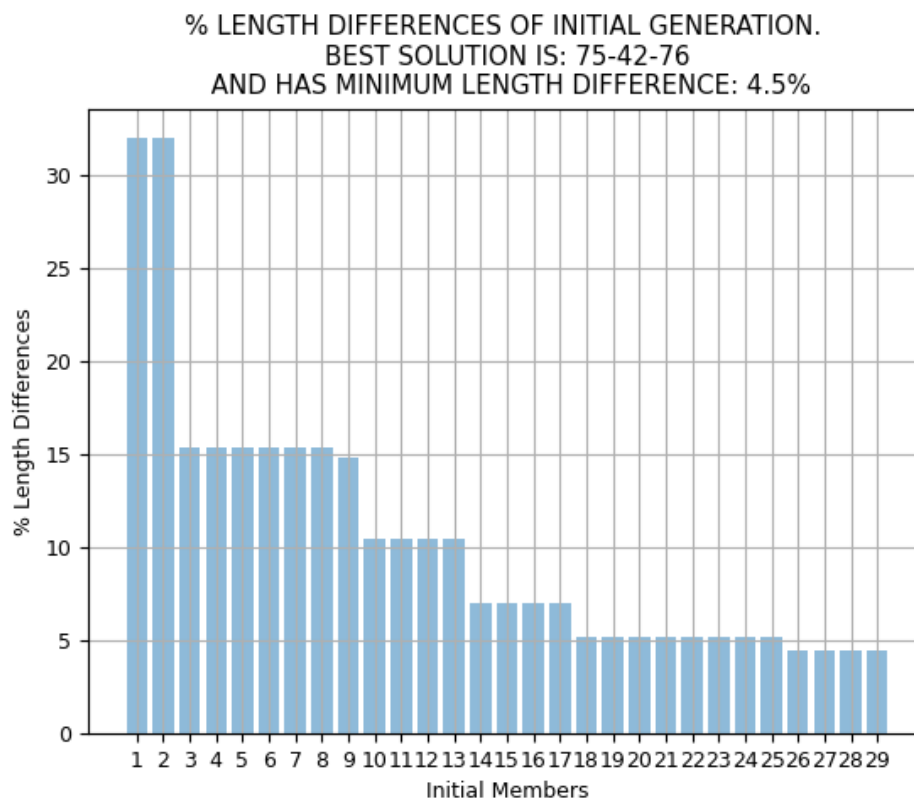


Εικόνα 96 Σενάριο 2.1-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά

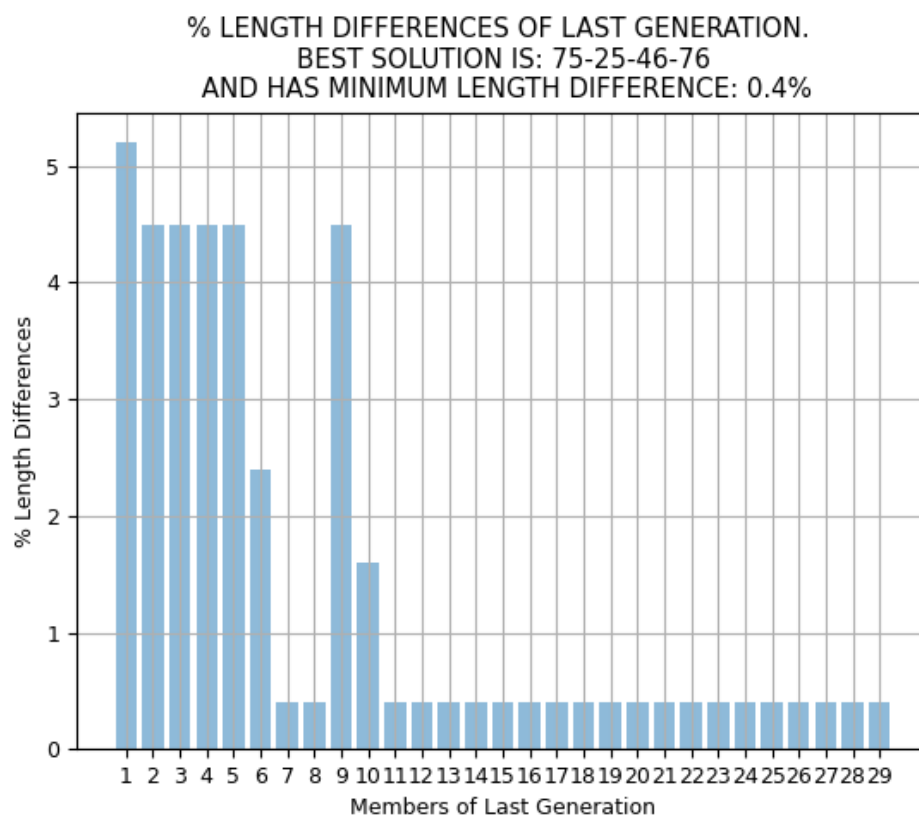
POPULATION: 29 PARENTS: 28 WINNERS: 14 POOL: 6
MUTATION FREQUENCY: 0.3 GENERATIONS: 20 INITIAL FITNESS: 212.13
FINAL FITNESS: 191.42 RUNNING TIME: 63.9 sec



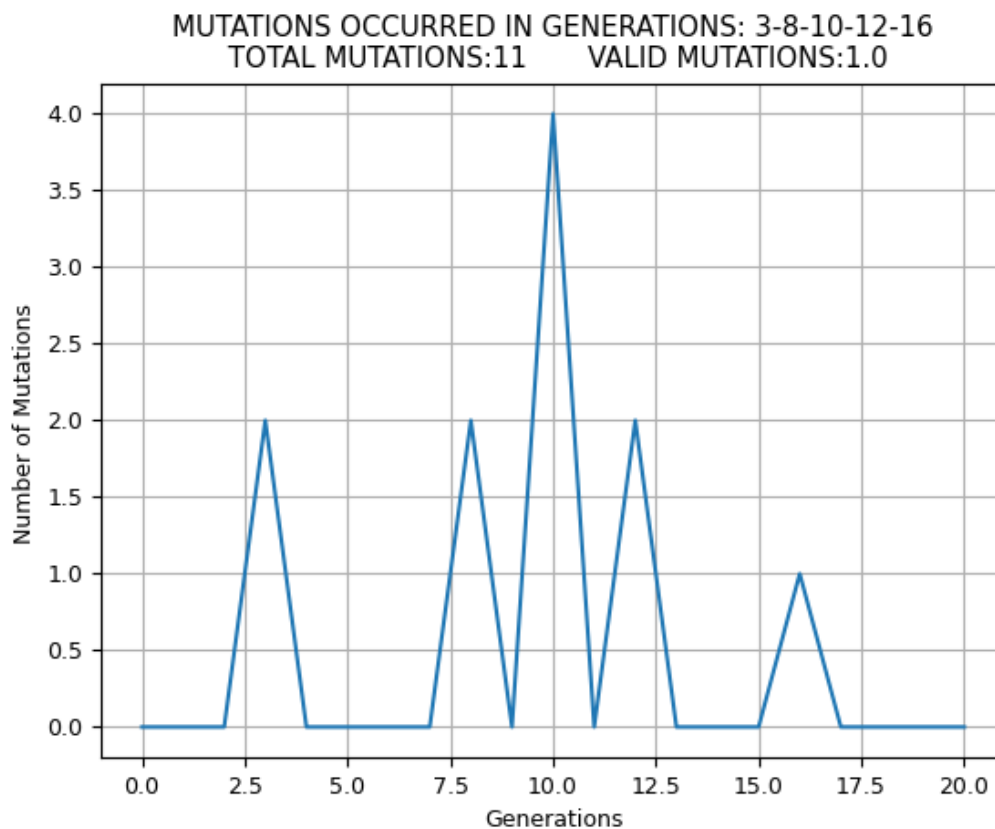
Εικόνα 97 Σενάριο 2.1-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



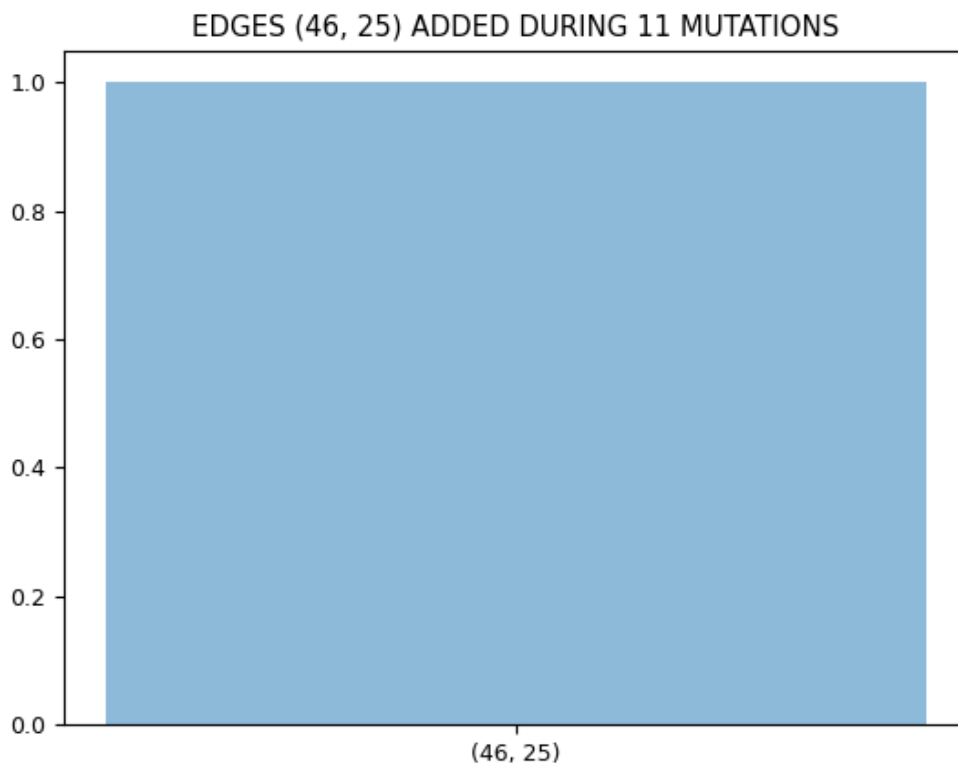
Εικόνα 98 Σενάριο 2.1-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.



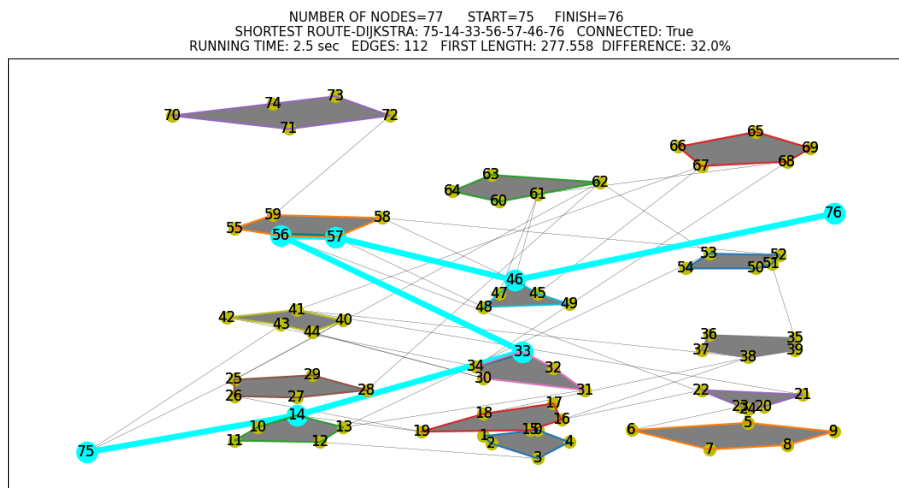
Εικόνα 99 Σενάριο 2.1-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.



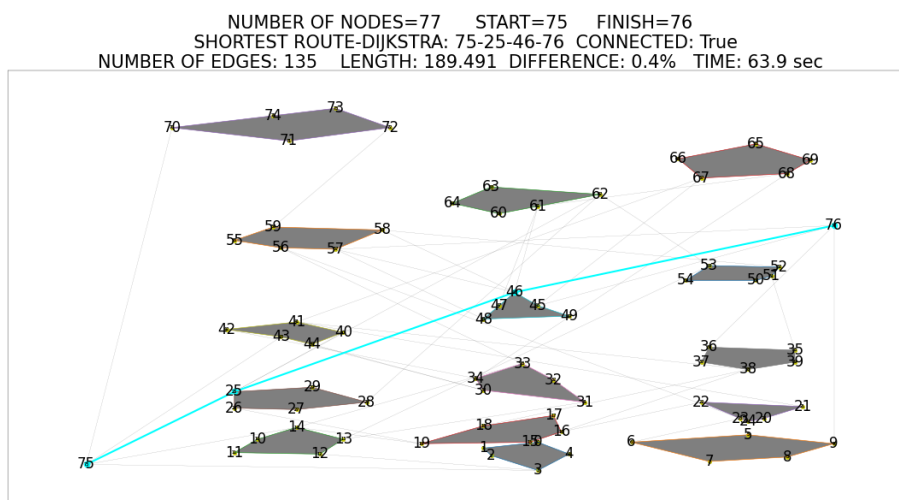
Εικόνα 100 Σενάριο 2.1-Πλήθος μεταλλάξεων ανά γενιά



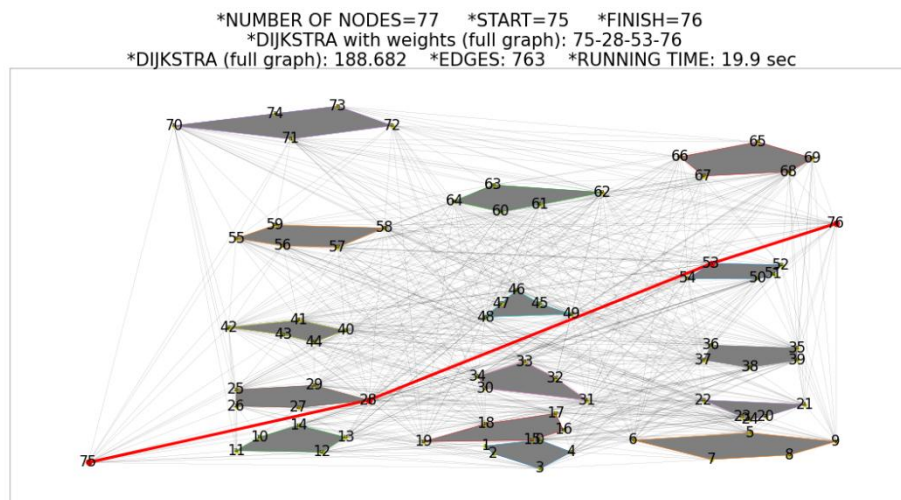
Εικόνα 101 Σενάριο 2.1-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης.



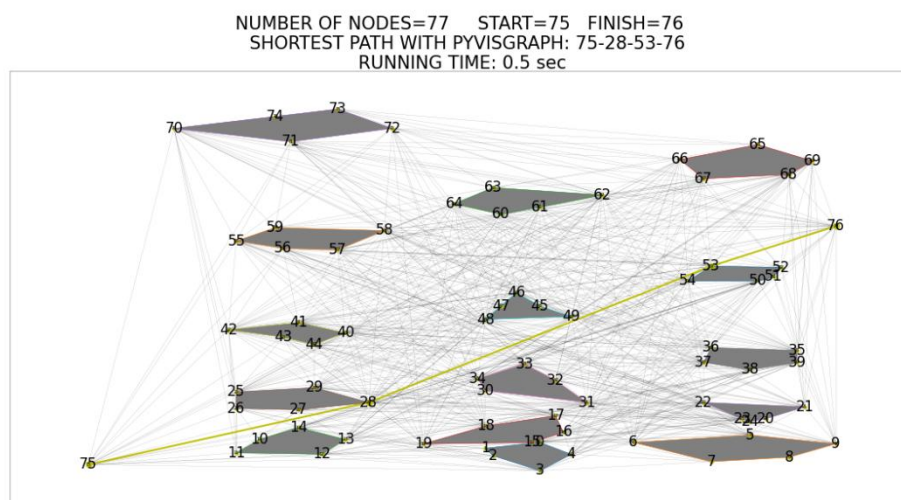
Εικόνα 102 Σενάριο 2.1-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 103 Σενάριο 2.1-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



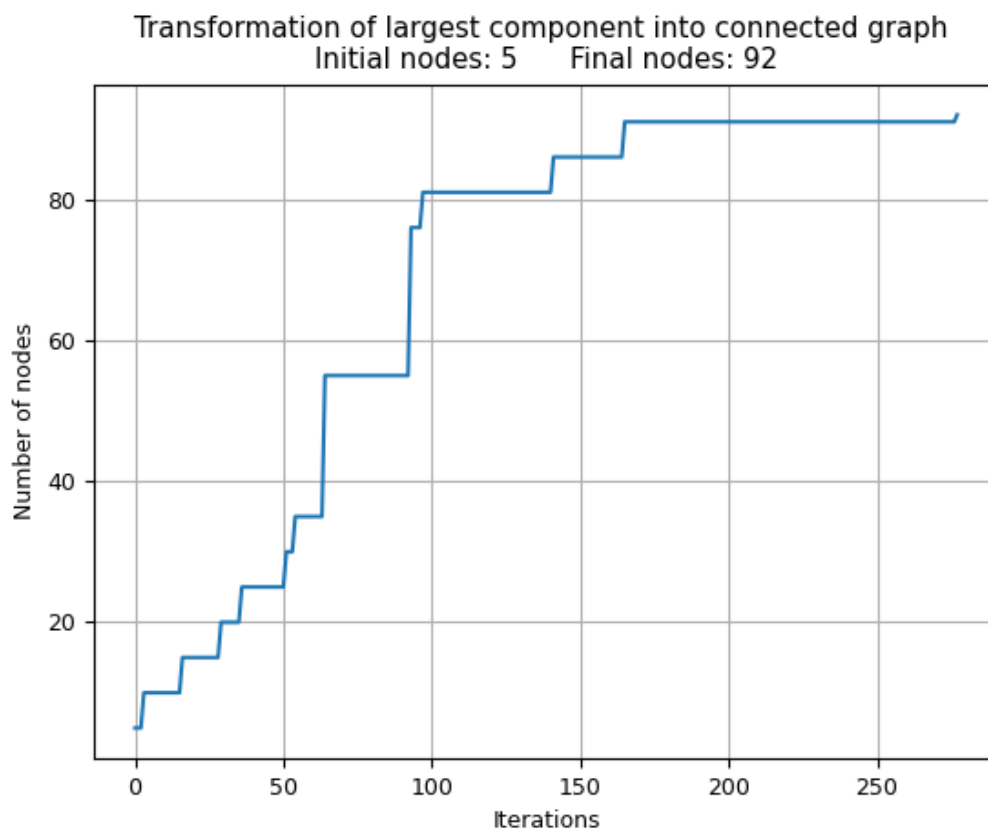
Εικόνα 104 Σενάριο 2.1-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



Εικόνα 105 Σενάριο 2.1-Αλγόριθμος Lee

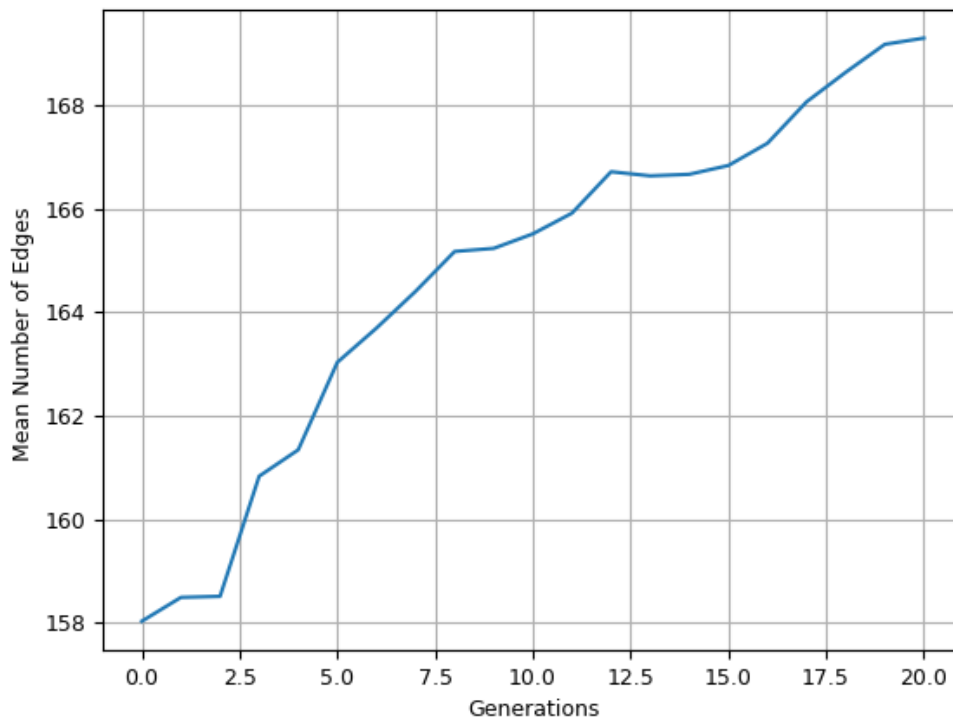
Σενάριο 2.2- Γράφος Ορατότητας με 92 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	35
Συνολικές μεταλλάξεις/Εγκυρες	16/4
Συχνότητα μετάλλαξης	0.3
Διαφορά αποστάσεων αρχικού γράφου	14.2%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	1.7%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	0.6%
Συνολικές ακμές (Γενετικός Αλγ. #2)	172
Συνολικές ακμές (Full Graph)	942
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	223.817
Μήκος μονοπατιού (Full Graph)	222.468
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	134.4 sec
Χρόνος εκτέλεσης (Full Graph)	33.9 sec
Χρόνος εκτέλεσης (Lee)	0.7 sec

Πίνακας 16 Σενάριο 2.2



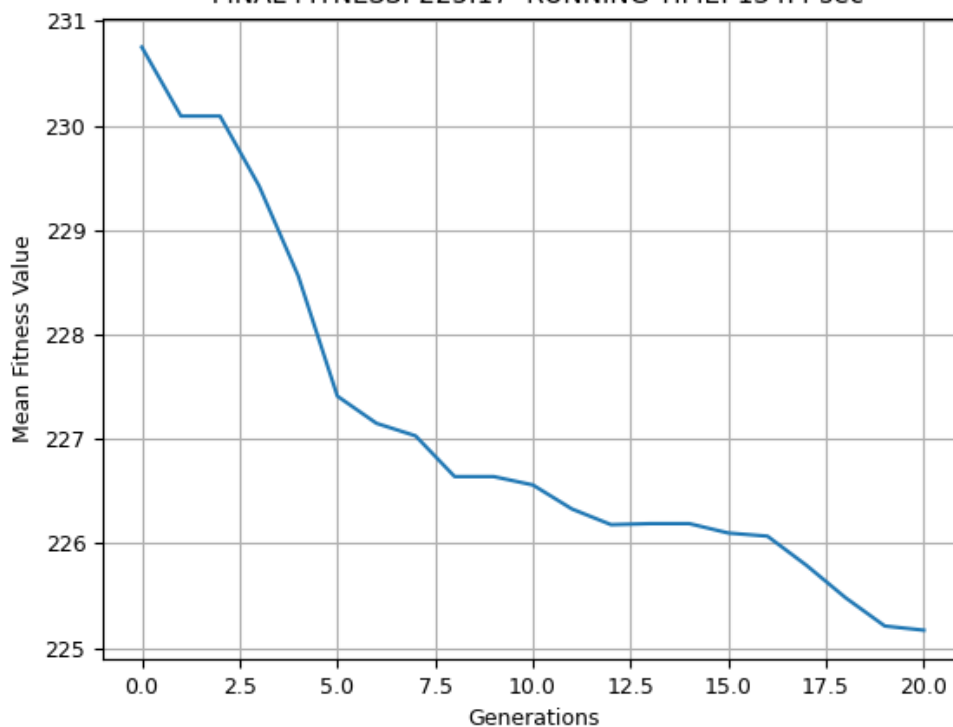
Εικόνα 106 Σενάριο 2.2-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.

UNVALID EDGES CREATED AT INITIAL POP CONSTRUCTION: 40
INITIAL MEAN NUM of EDGES=158.03 FINAL MEAN NUM of EDGES: 169.29
RUNNING TIME: 134.4 sec

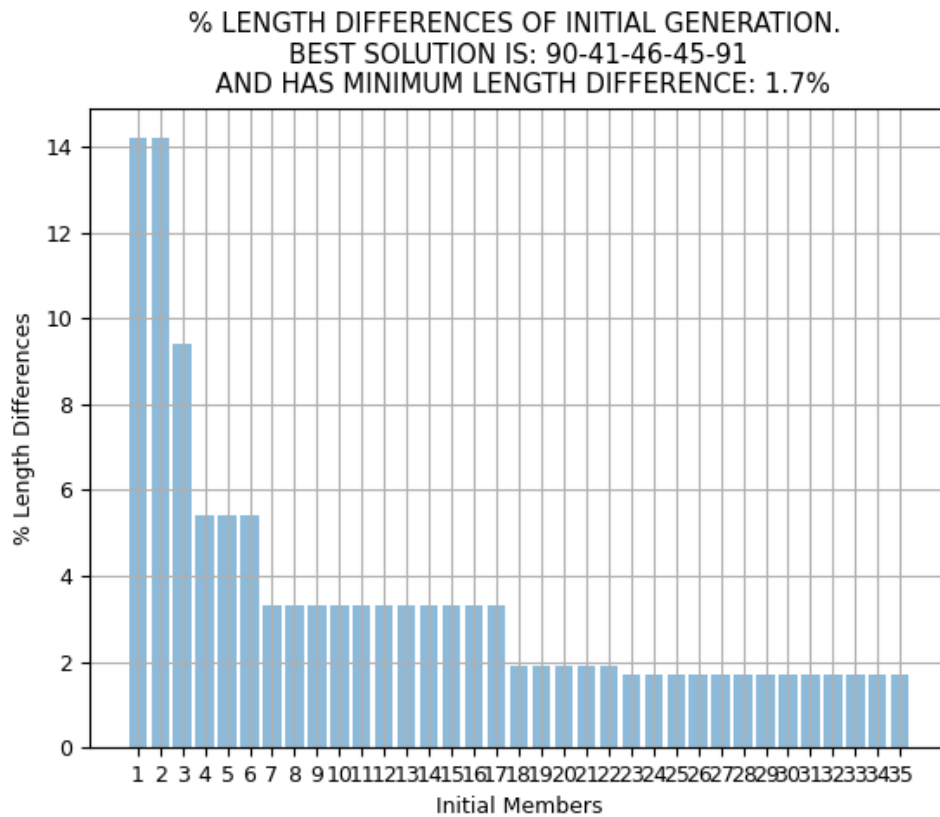


Εικόνα 107 Σενάριο 2.2-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά

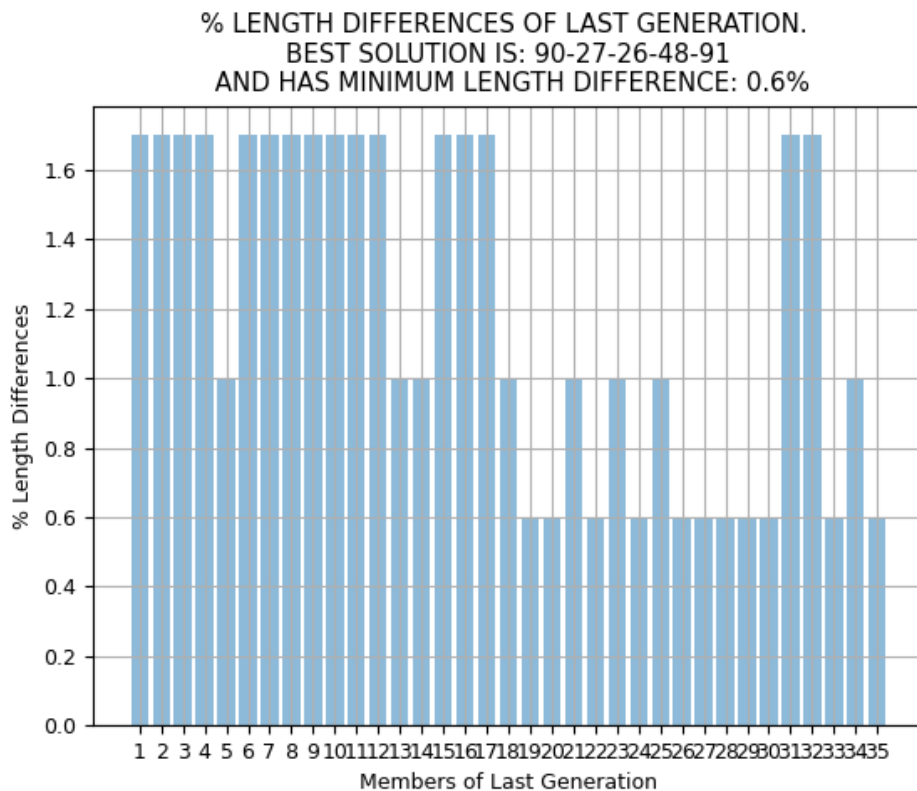
POPULATION: 35 PARENTS: 34 WINNERS: 17 POOL: 8
MUTATION FREQUENCY: 0.3 GENERATIONS: 20 INITIAL FITNESS: 230.75
FINAL FITNESS: 225.17 RUNNING TIME: 134.4 sec



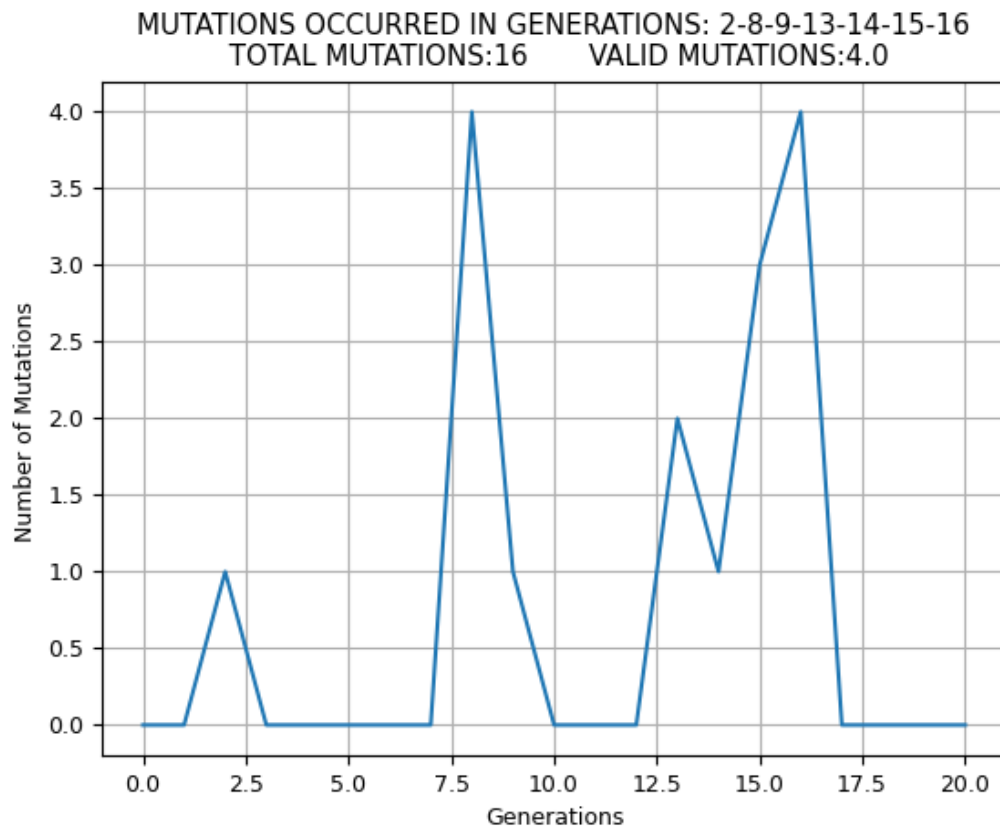
Εικόνα 108 Σενάριο 2.2-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



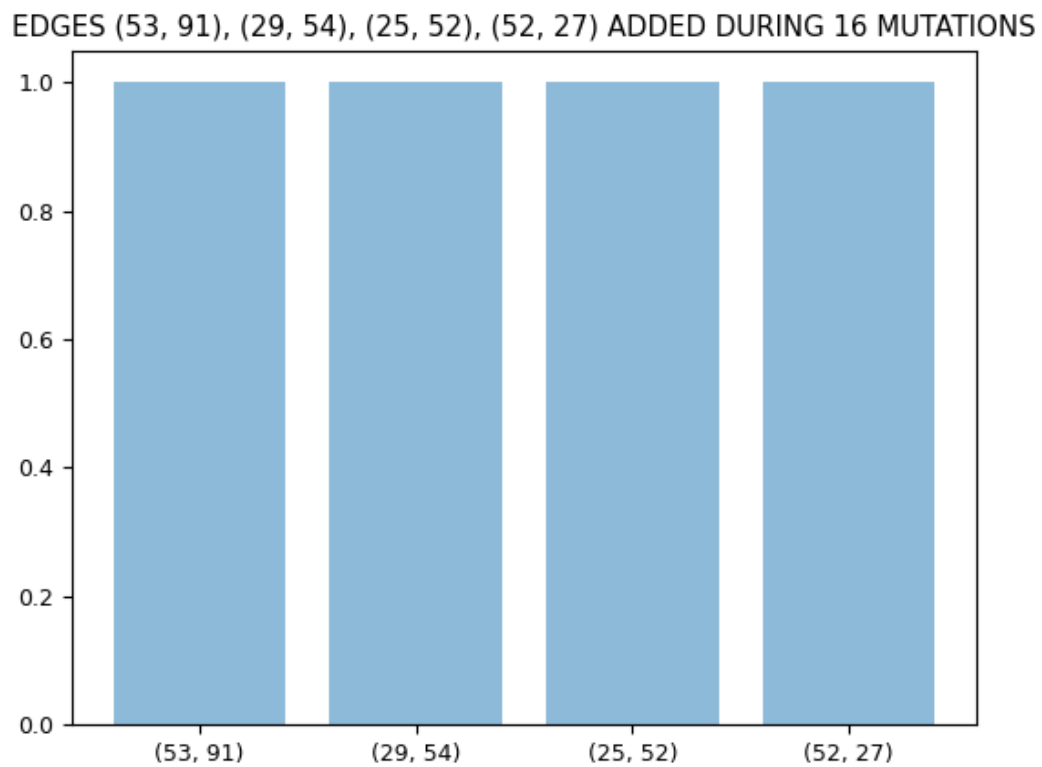
Εικόνα 109 Σενάριο 2.2-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.



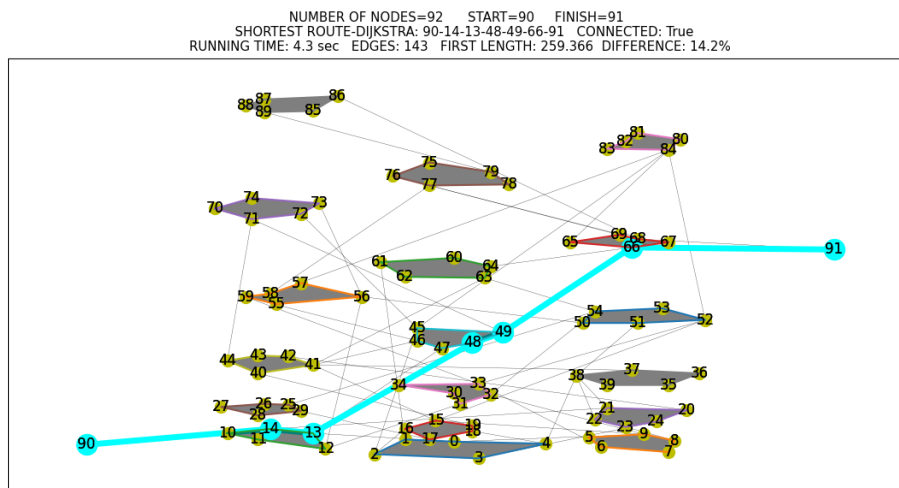
Εικόνα 110 Σενάριο 2.2-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.



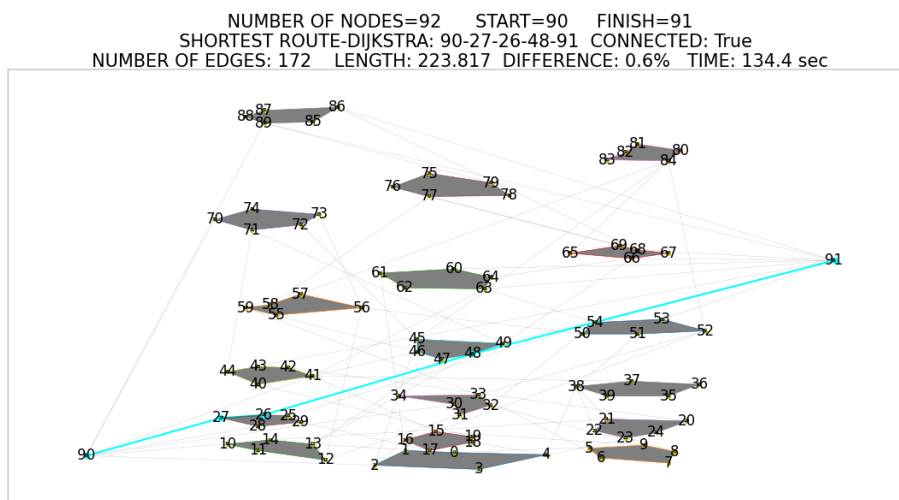
Εικόνα 111 Σενάριο 2.2-Πλήθος μεταλλάξεων ανά γενιά



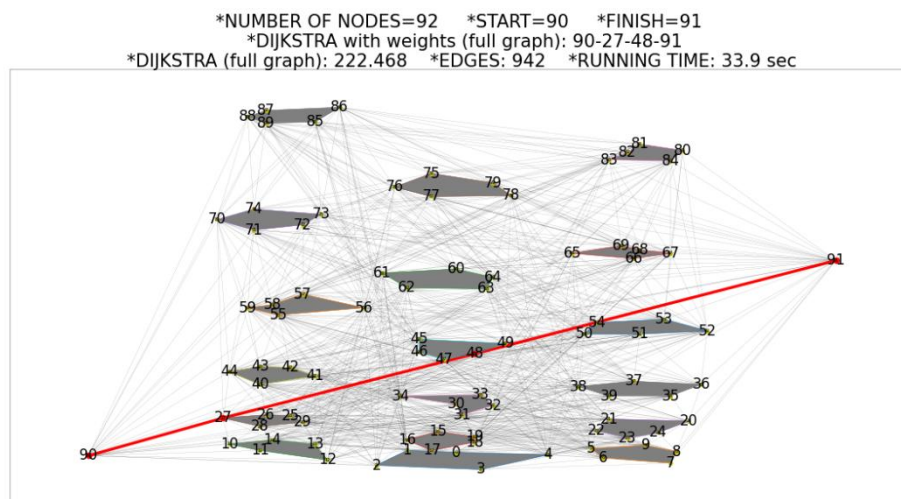
Εικόνα 112 Σενάριο 2.2-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης



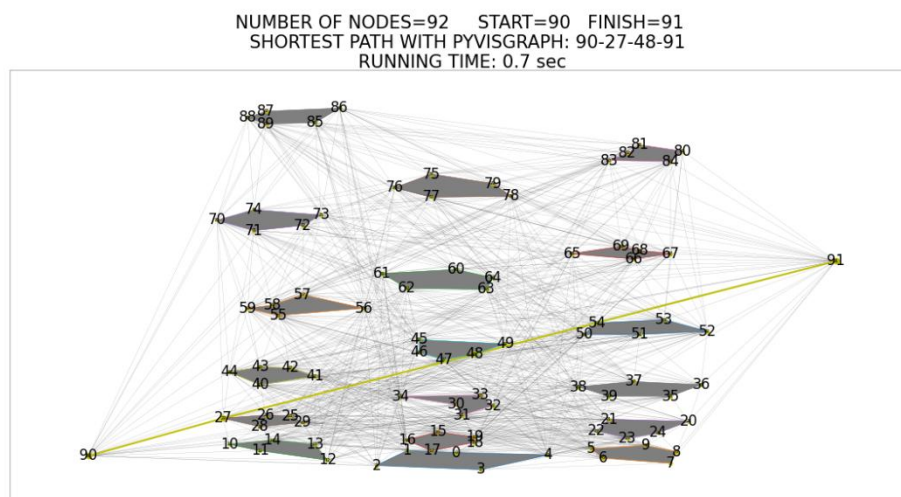
Εικόνα 113 Σενάριο 2.2-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 114 Σενάριο 2.2-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



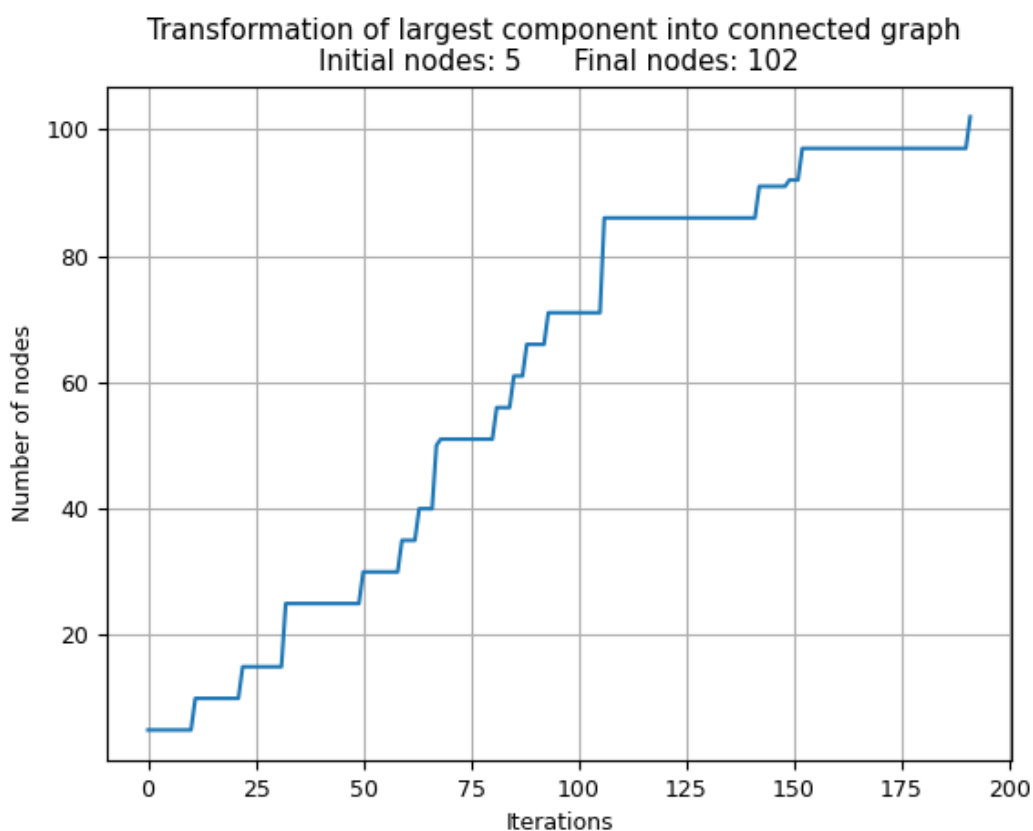
Εικόνα 115 Σενάριο 2.2-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



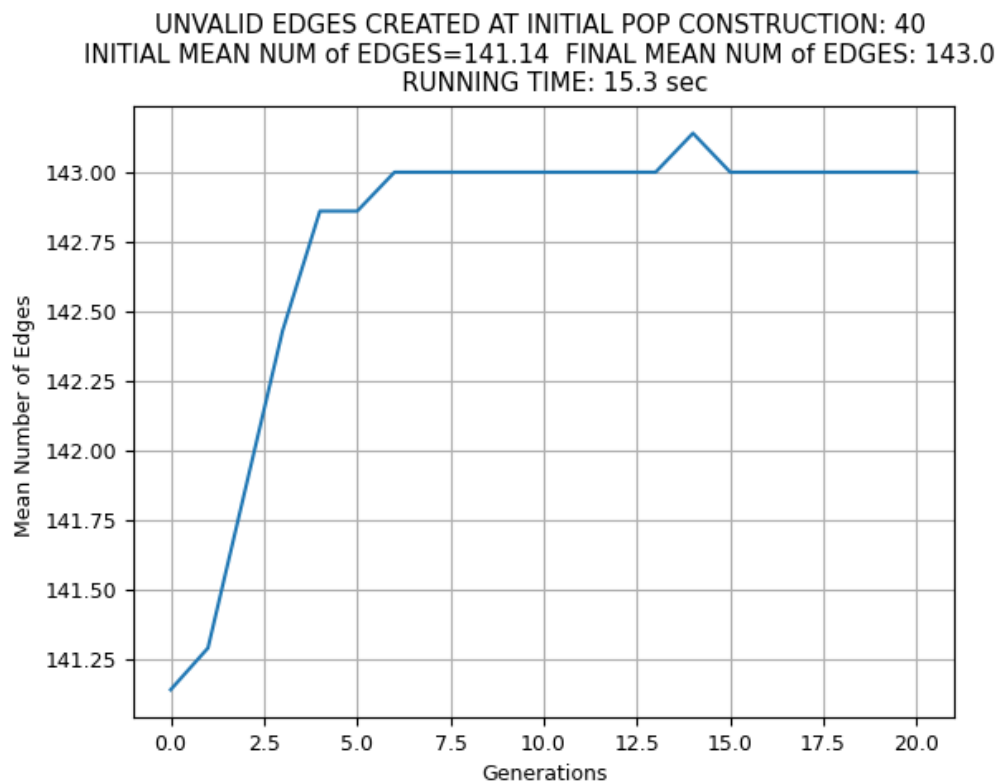
Εικόνα 116 Σενάριο 2.2-Αλγόριθμος Lee

Σενάριο 2.3- Γράφος Ορατότητας με 102 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	7
Συνολικές μεταλλάξεις/Εγκυρες	3/0
Συχνότητα μετάλλαξης	0.3
Διαφορά αποστάσεων αρχικού γράφου	11.5%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	1%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	1%
Συνολικές ακμές (Γενετικός Αλγ. #2)	143
Συνολικές ακμές (Full Graph)	1114
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	502.937
Μήκος μονοπατιού (Full Graph)	498.127
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	15.3 sec
Χρόνος εκτέλεσης (Full Graph)	29.1 sec
Χρόνος εκτέλεσης (Lee)	0.7 sec

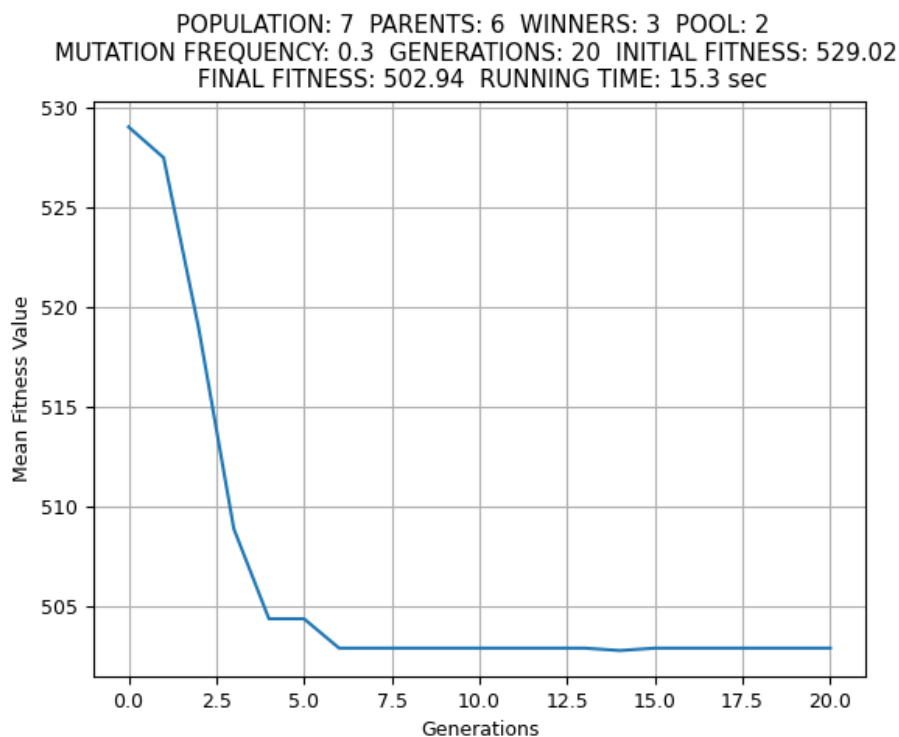
Πίνακας 17 Σενάριο 2.3



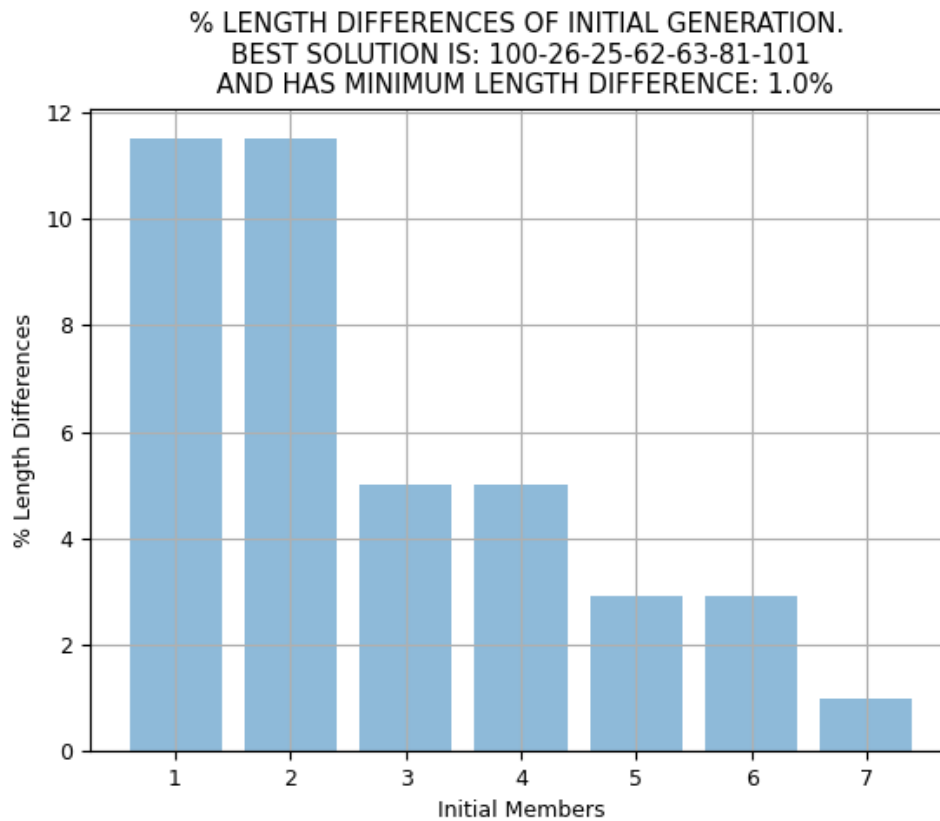
Εικόνα 117 Σενάριο 2.3-Πλήθος επαναλήψεων έως όπου ο γράφος συνδεθεί για πρώτη φορά.



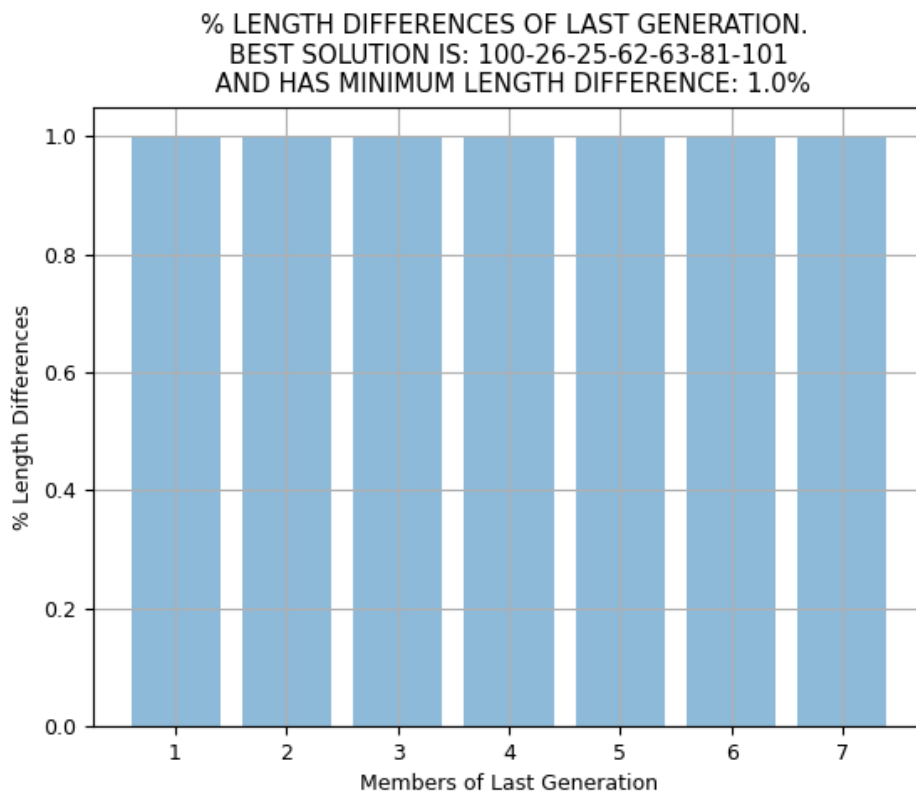
Εικόνα 118 Σενάριο 2.3-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά



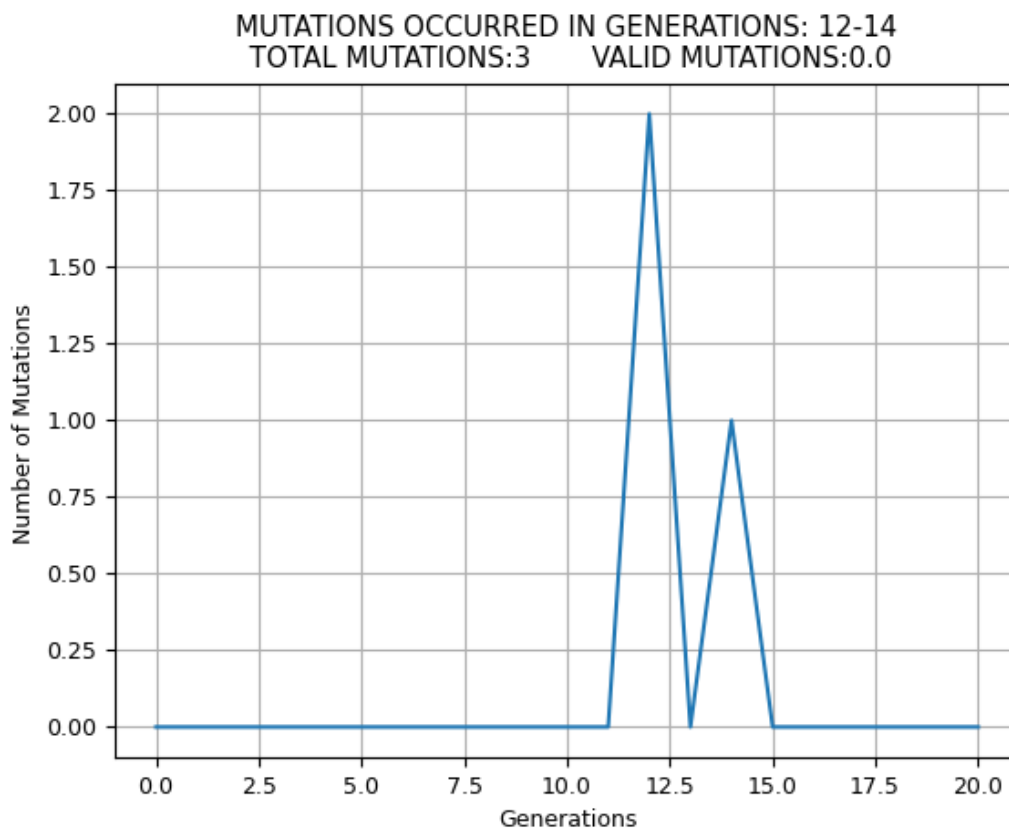
Εικόνα 119 Σενάριο 2.3-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



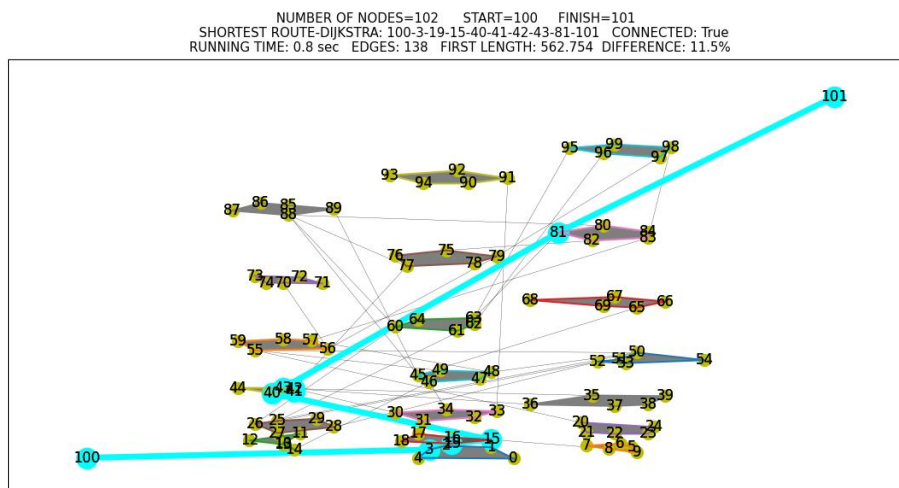
Εικόνα 120 Σενάριο 2.3-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.



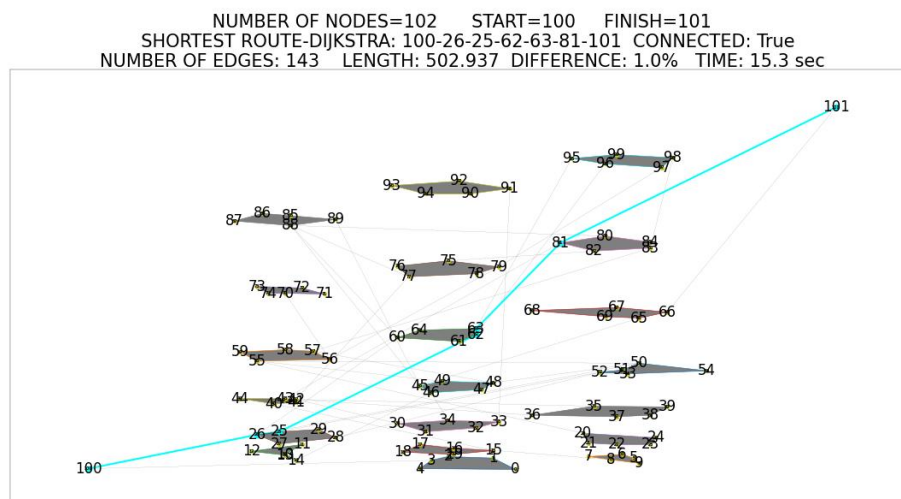
Εικόνα 121 Σενάριο 2.3-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.



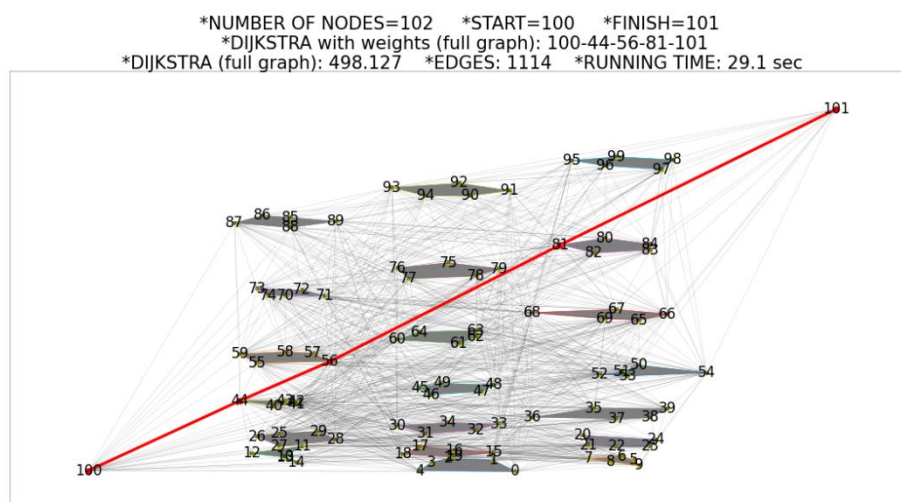
Εικόνα 122 Σενάριο 2.3-Πλήθος μεταλλάξεων ανά γενιά



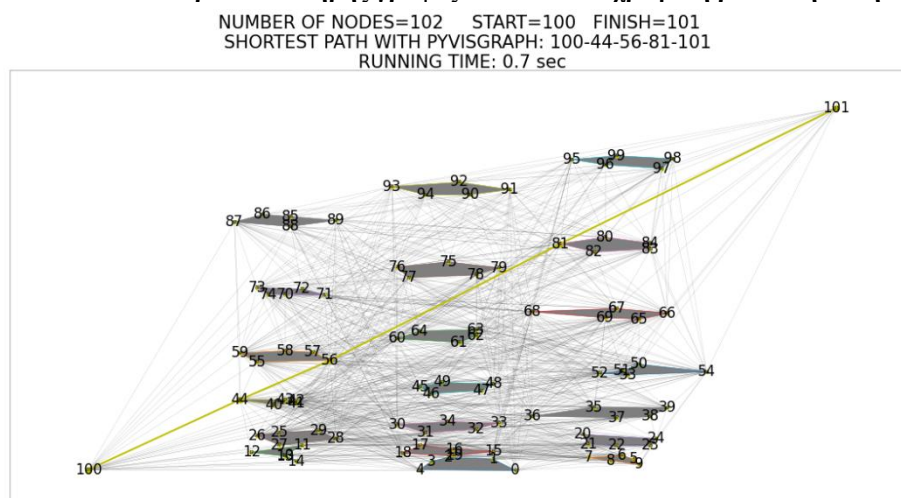
Εικόνα 123 Σενάριο 2.3-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 124 Σενάριο 2.3-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



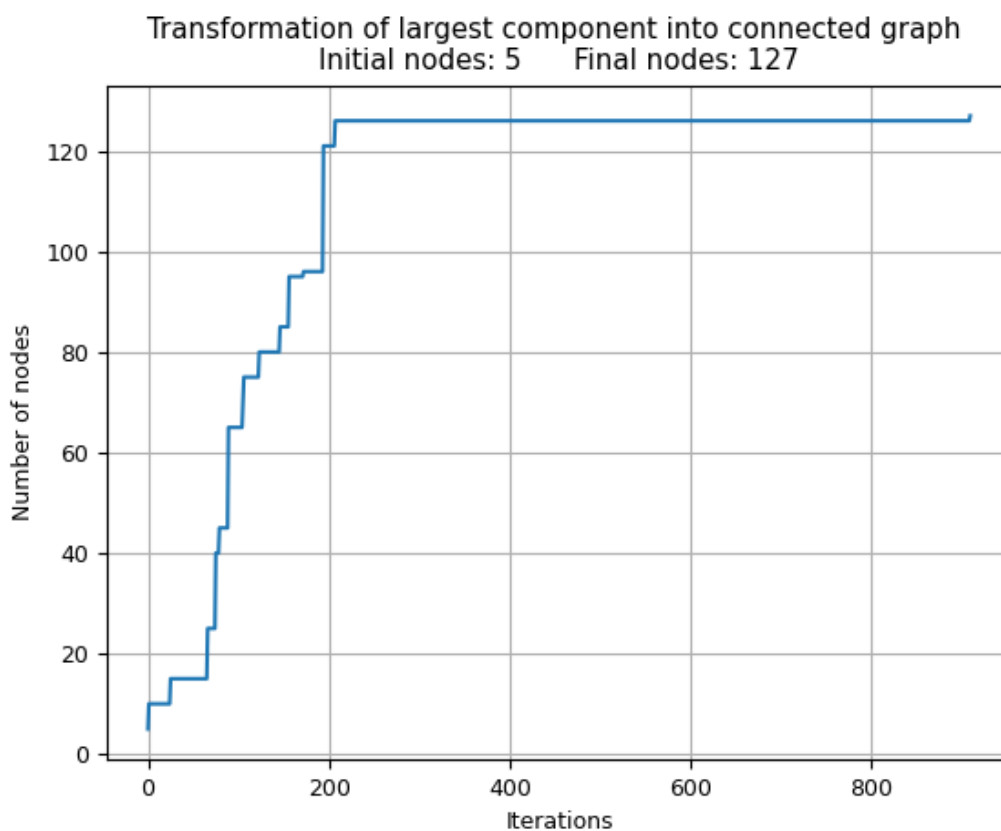
Εικόνα 125 Σενάριο 2.3-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



Εικόνα 126 Σενάριο 2.3-Αλγόριθμος Lee

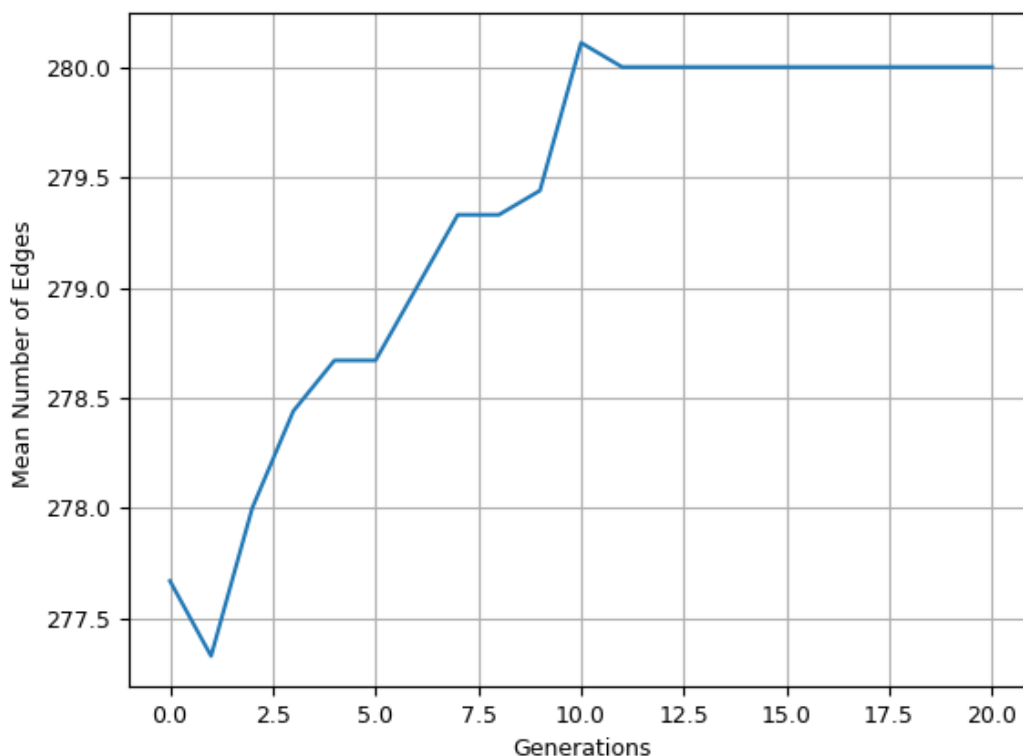
Σενάριο 2.4- Γράφος Ορατότητας με 127 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	9
Συνολικές μεταλλάξεις/Εγκυρες	1/0
Συχνότητα μετάλλαξης	0.3
Διαφορά αποστάσεων αρχικού γράφου	12.2%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	0.2%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	0.2%
Συνολικές ακμές (Γενετικός Αλγ. #2)	280
Συνολικές ακμές (Full Graph)	1513
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	530.829
Μήκος μονοπατιού (Full Graph)	529.986
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	70.7 sec
Χρόνος εκτέλεσης (Full Graph)	89.5 sec
Χρόνος εκτέλεσης (Lee)	1.2 sec

Πίνακας 18 Σενάριο 2.4



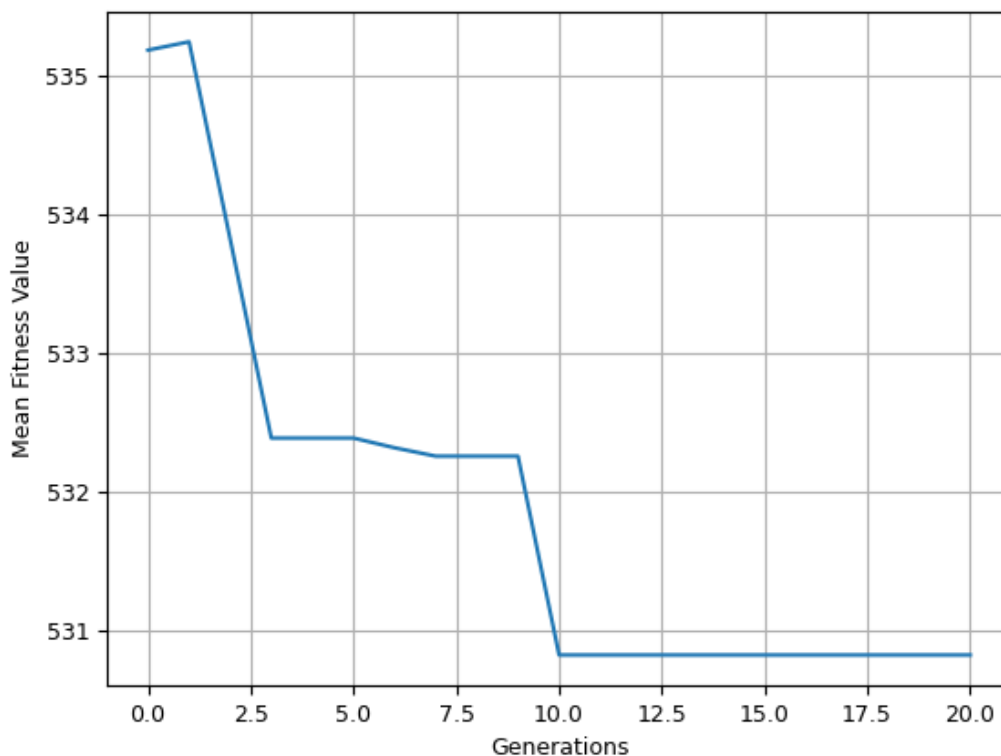
Εικόνα 127 Σενάριο 2.4-Πλήθος επαναλήψεων έως όπου ο γράφος συνδεθεί για πρώτη φορά.

UNVALID EDGES CREATED AT INITIAL POP CONSTRUCTION: 20
INITIAL MEAN NUM of EDGES=277.67 FINAL MEAN NUM of EDGES: 280.0
RUNNING TIME: 70.7 sec

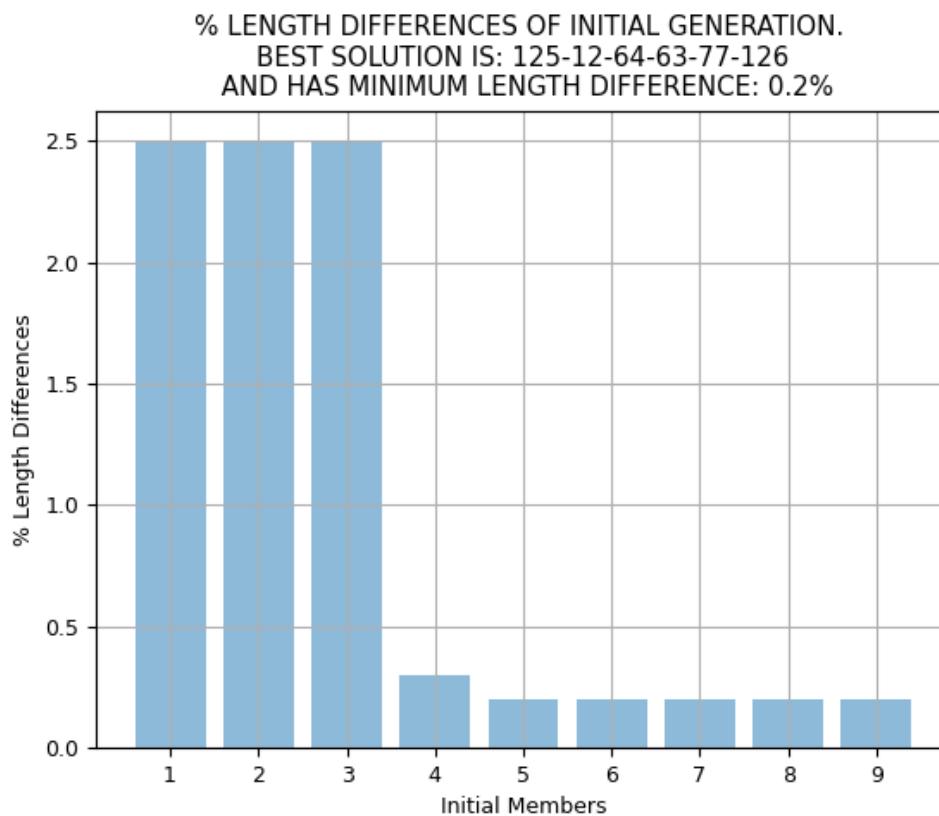


Εικόνα 128 Σενάριο 2.4-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά

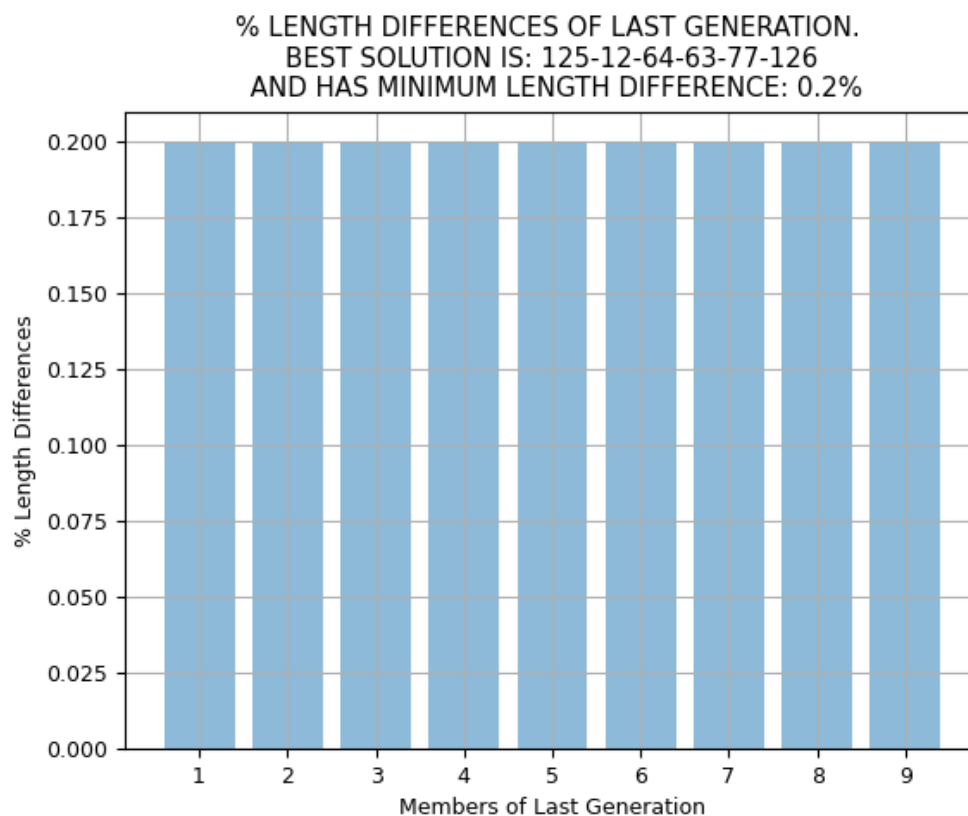
POPULATION: 9 PARENTS: 8 WINNERS: 4 POOL: 2
MUTATION FREQUENCY: 0.2 GENERATIONS: 20 INITIAL FITNESS: 535.18
FINAL FITNESS: 530.83 RUNNING TIME: 70.7 sec



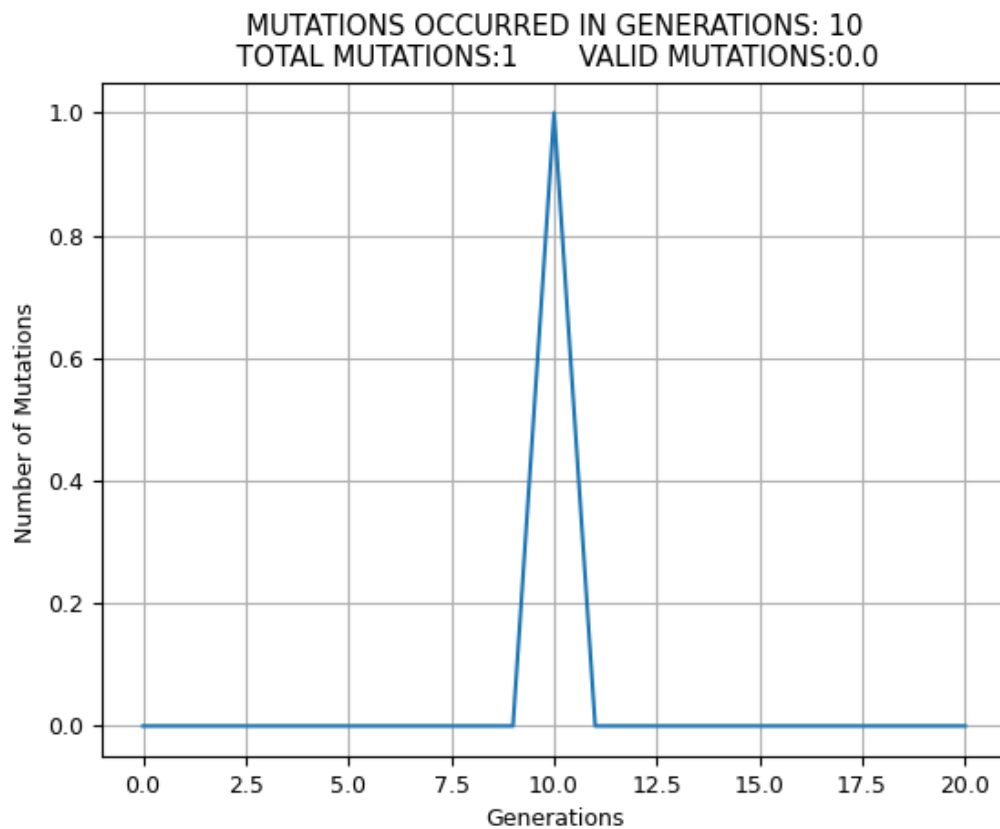
Εικόνα 129 Σενάριο 2.4-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



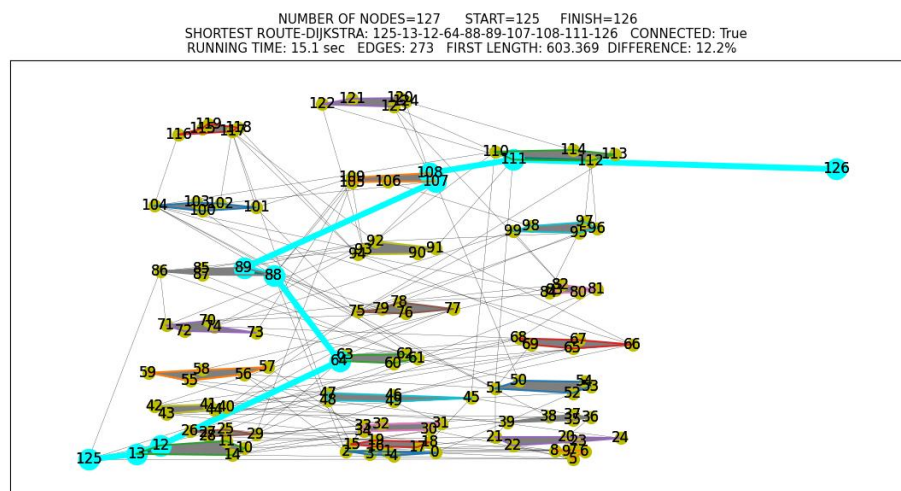
Εικόνα 130 Σενάριο 2.4-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση.



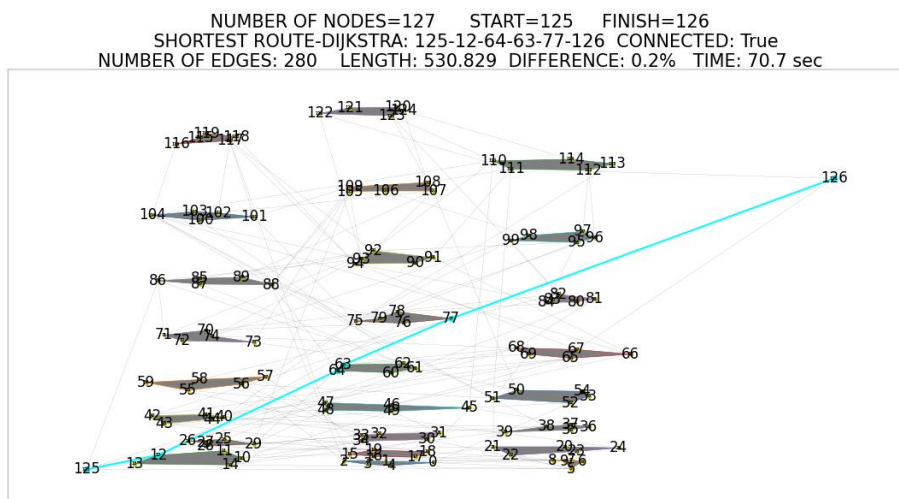
Εικόνα 131 Σενάριο 2.4-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση.



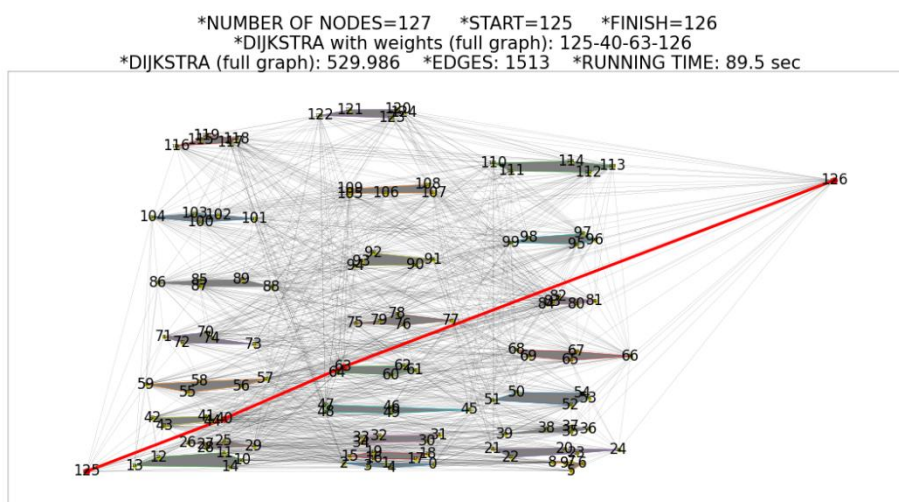
Εικόνα 132 Σενάριο 2.4-Πλήθος μεταλλάξεων ανά γενιά



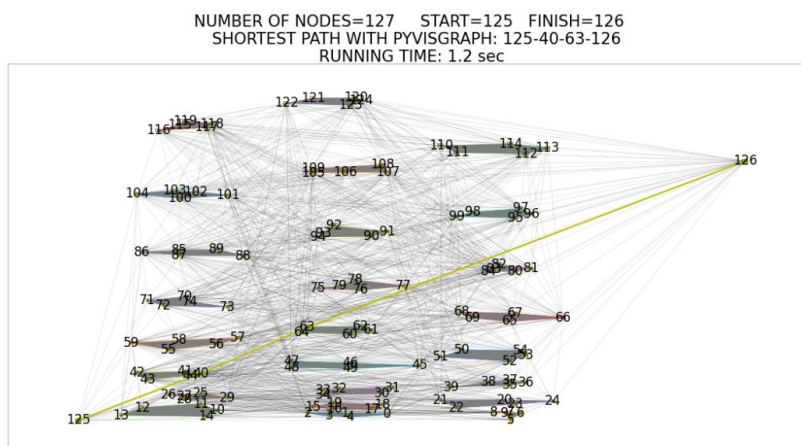
Εικόνα 133 Σενάριο 2.4-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 134 Σενάριο 2.4-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



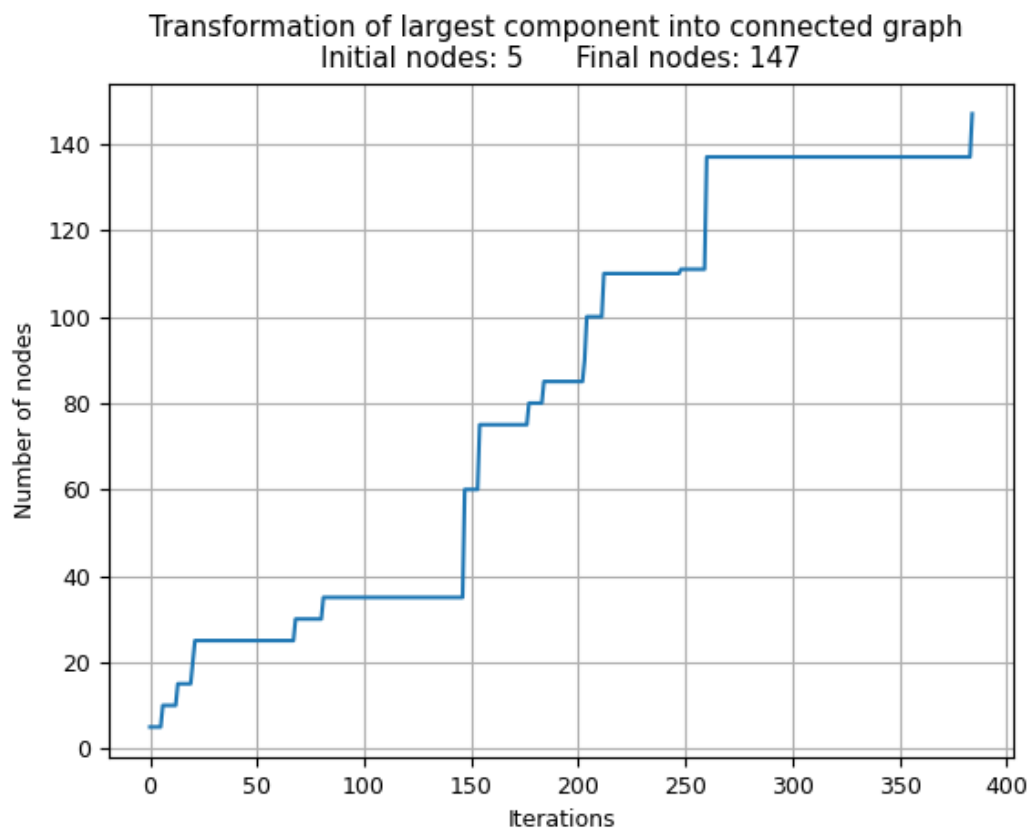
Εικόνα 135 Σενάριο 2.4-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



Εικόνα 136 Σενάριο 2.4-Αλγόριθμος Lee

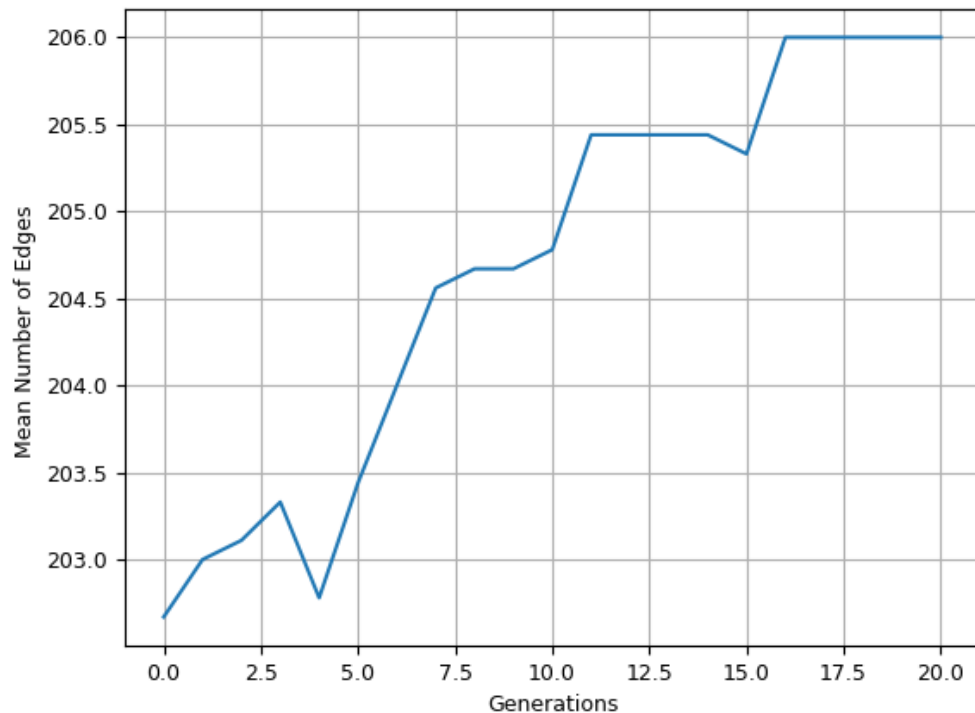
Σενάριο 2.5- Γράφος Ορατότητας με 147 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	9
Συνολικές μεταλλάξεις/Εγκυρες	1/1
Συχνότητα μετάλλαξης	0.2
Διαφορά αποστάσεων αρχικού γράφου	8.7%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	1.7%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	1.7%
Συνολικές ακμές (Γενετικός Αλγ. #2)	206
Συνολικές ακμές (Full Graph)	1778
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	520.119
Μήκος μονοπατιού (Full Graph)	511.307
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	68.7 sec
Χρόνος εκτέλεσης (Full Graph)	139.1 sec
Χρόνος εκτέλεσης (Lee)	1.6 sec

Πίνακας 19 Σενάριο 2.5



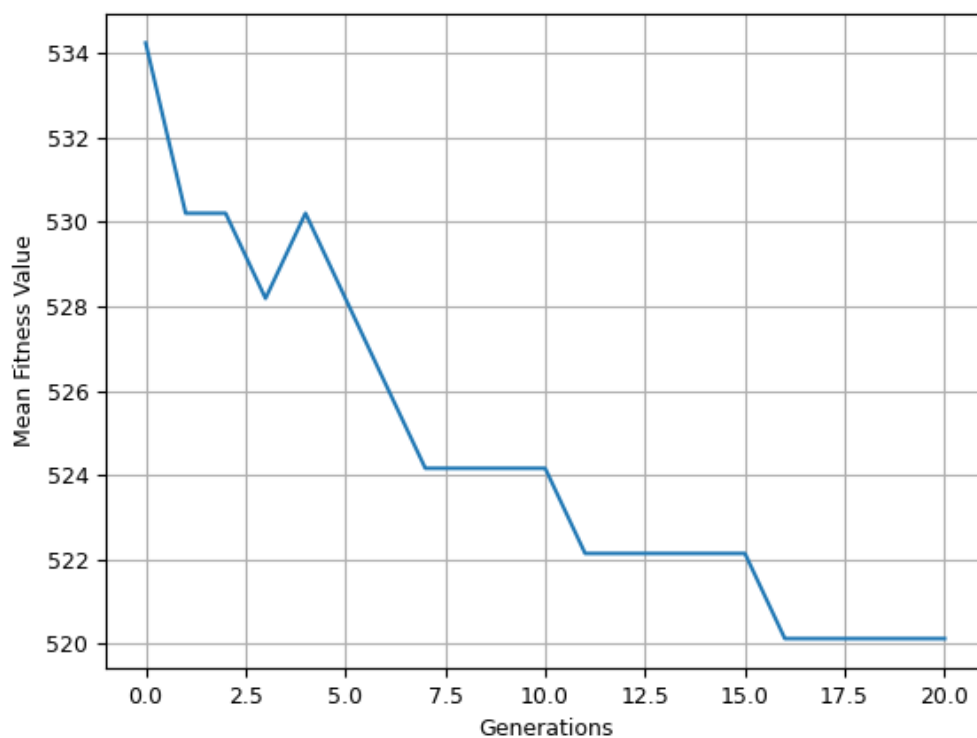
Εικόνα 137 Σενάριο 2.5-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.

UNVALID EDGES CREATED AT INITIAL POP CONSTRUCTION: 20
INITIAL MEAN NUM of EDGES=202.67 FINAL MEAN NUM of EDGES: 206.0
RUNNING TIME: 68.7 sec

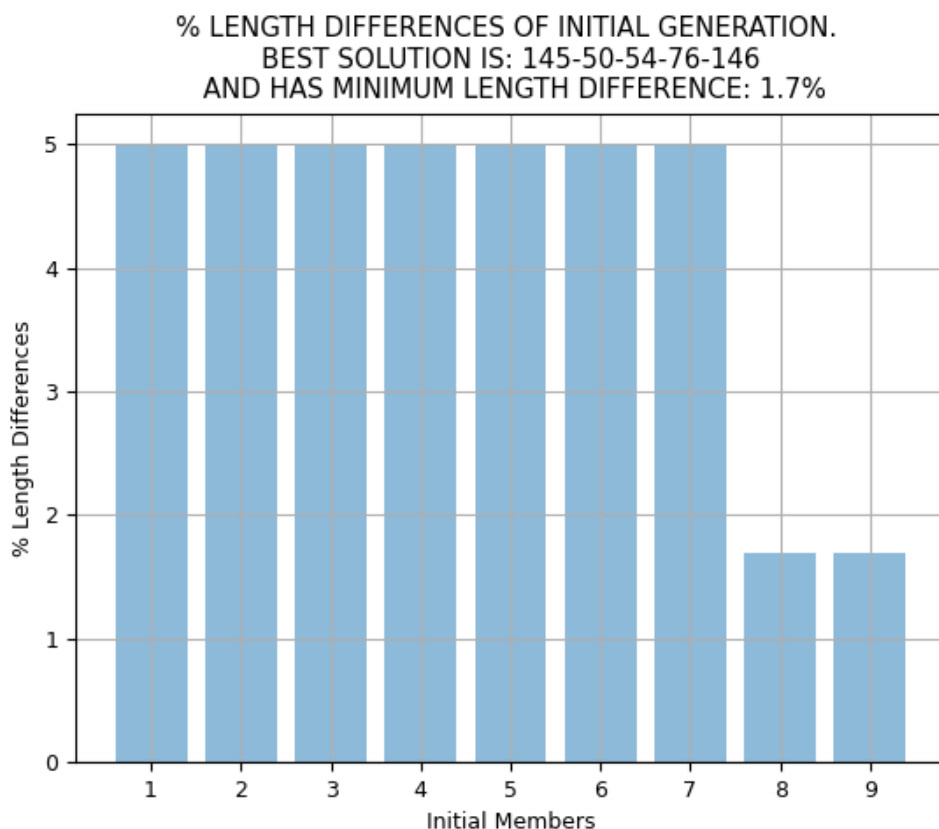


Εικόνα 138 Σενάριο 2.5-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά

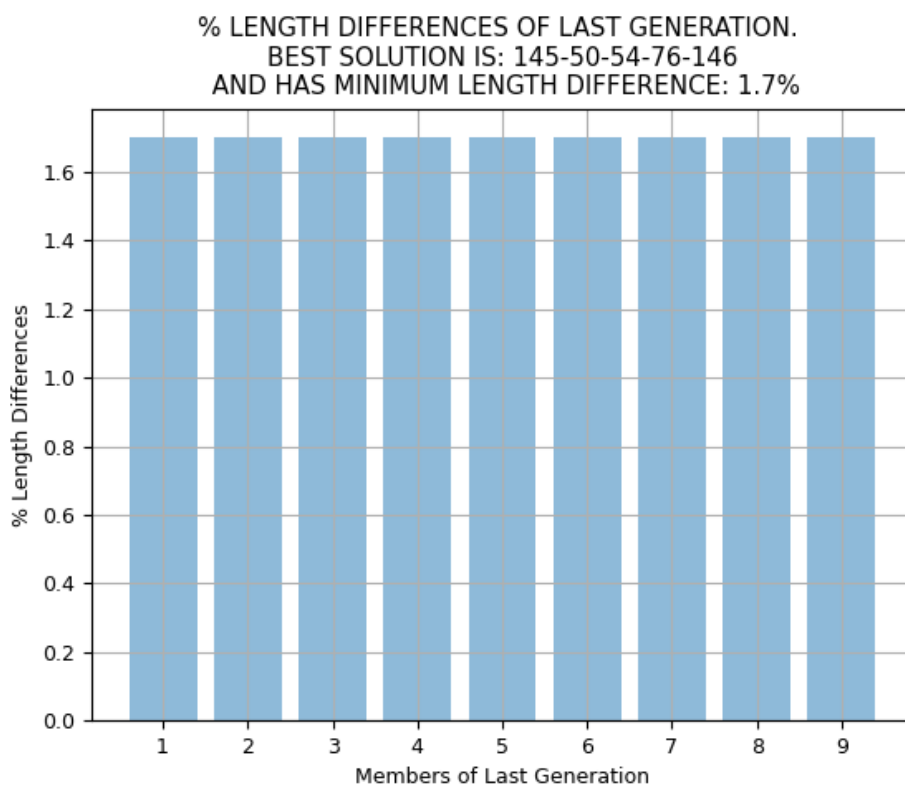
POPULATION: 9 PARENTS: 8 WINNERS: 4 POOL: 2
MUTATION FREQUENCY: 0.2 GENERATIONS: 20 INITIAL FITNESS: 534.25
FINAL FITNESS: 520.12 RUNNING TIME: 68.7 sec



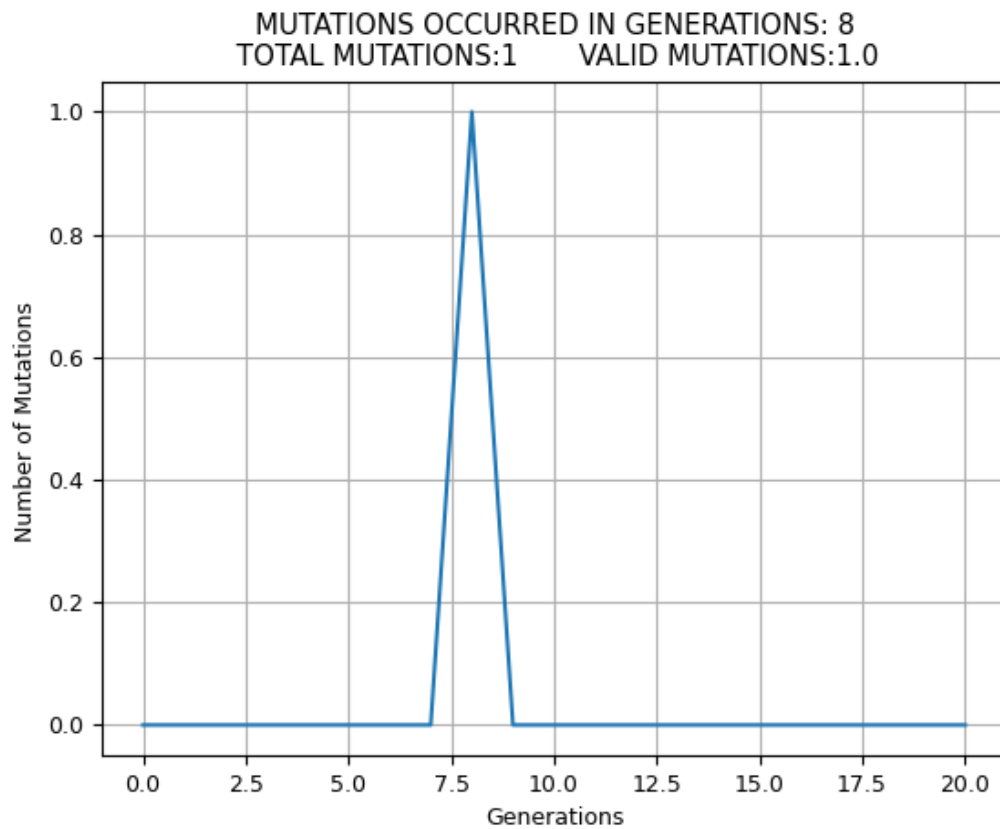
Εικόνα 139 Σενάριο 2.5-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



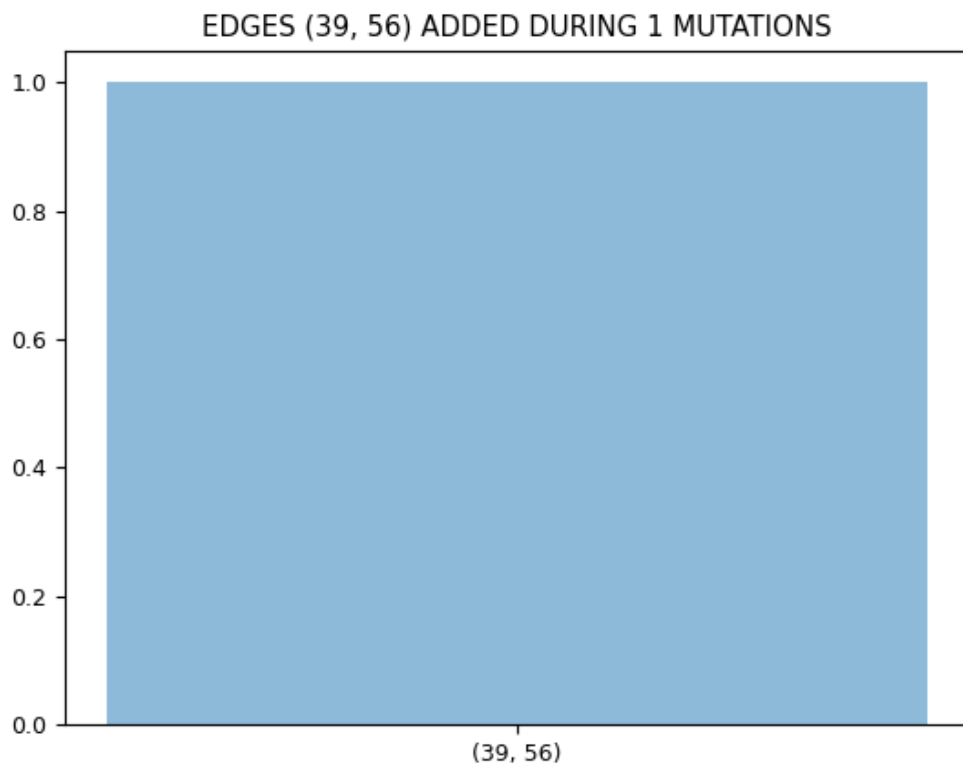
Εικόνα 140 Σενάριο 2.5-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση



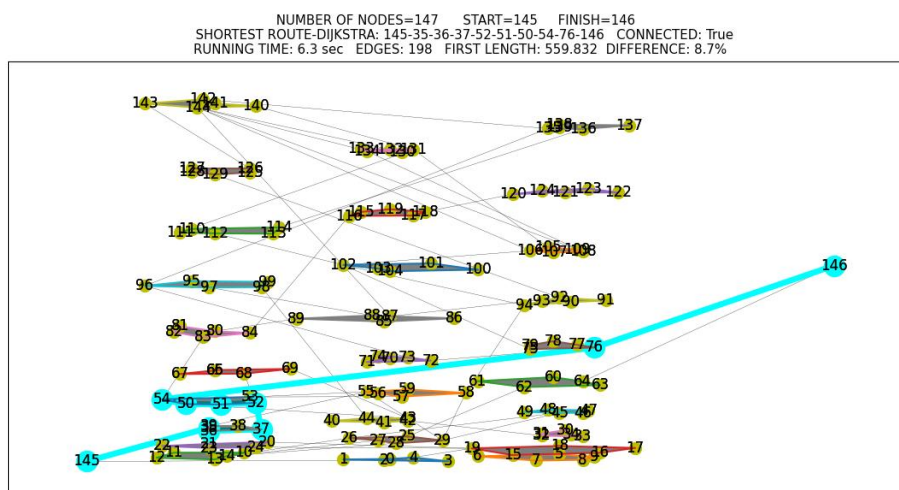
Εικόνα 141 Σενάριο 2.5-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση



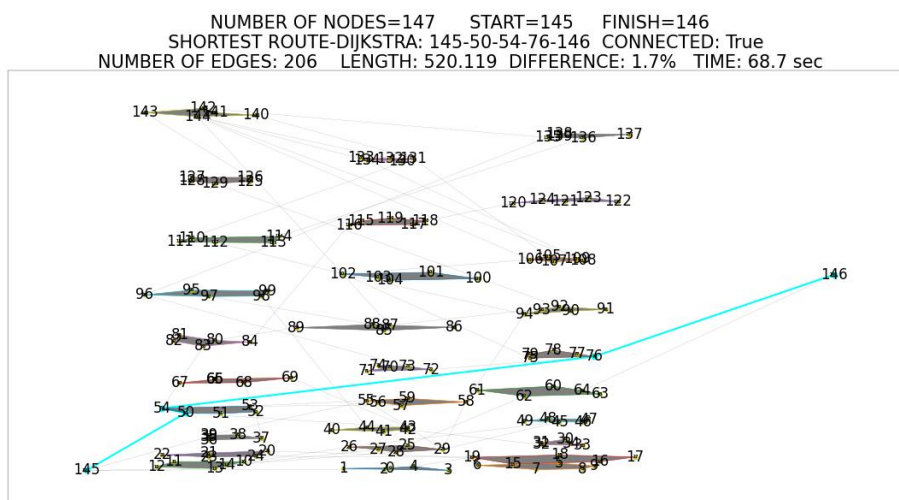
Εικόνα 142 Σενάριο 2.5-Πλήθος μεταλλάξεων ανά γενιά



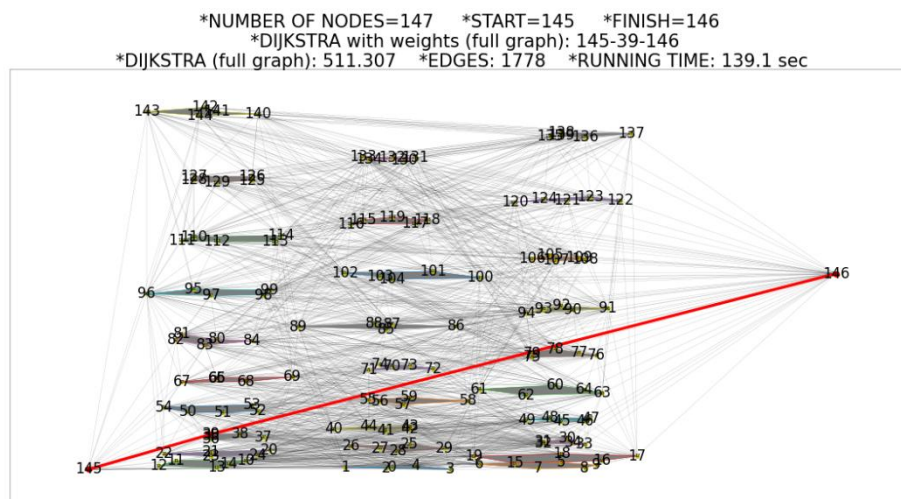
Εικόνα 143 Σενάριο 2.5-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης



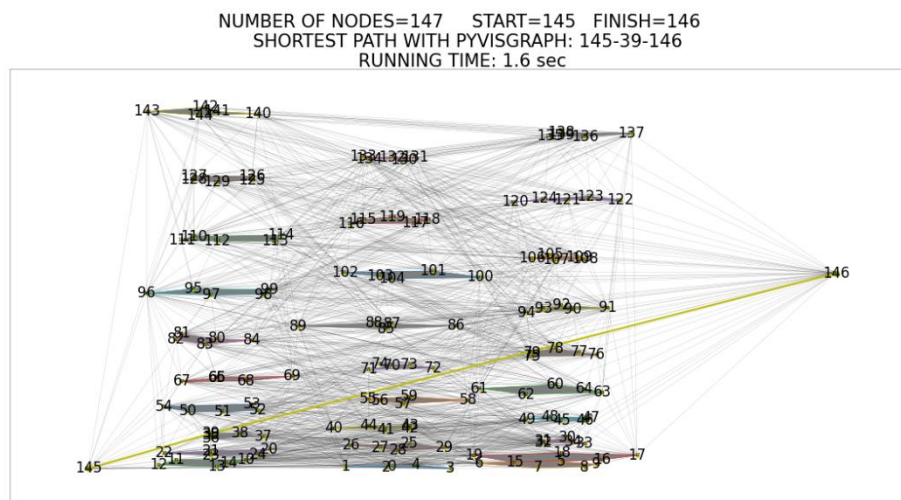
Εικόνα 144 Σενάριο 2.5-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 145 Σενάριο 2.5-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



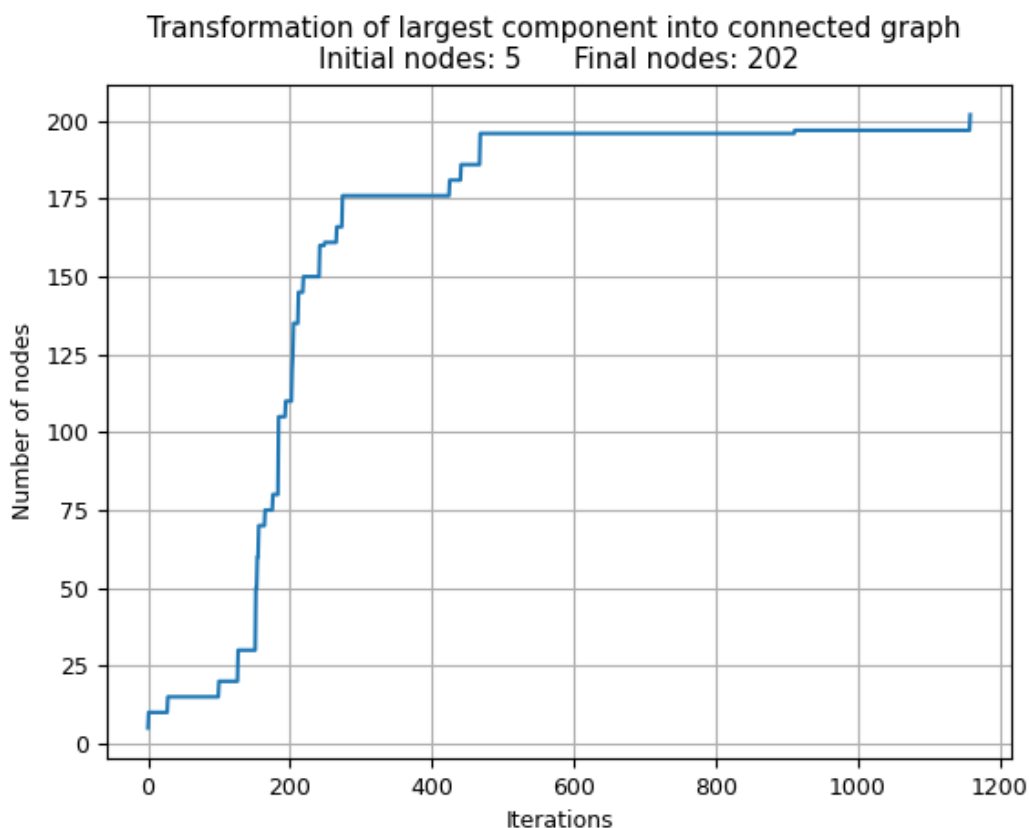
Εικόνα 146 Σενάριο 2.5-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



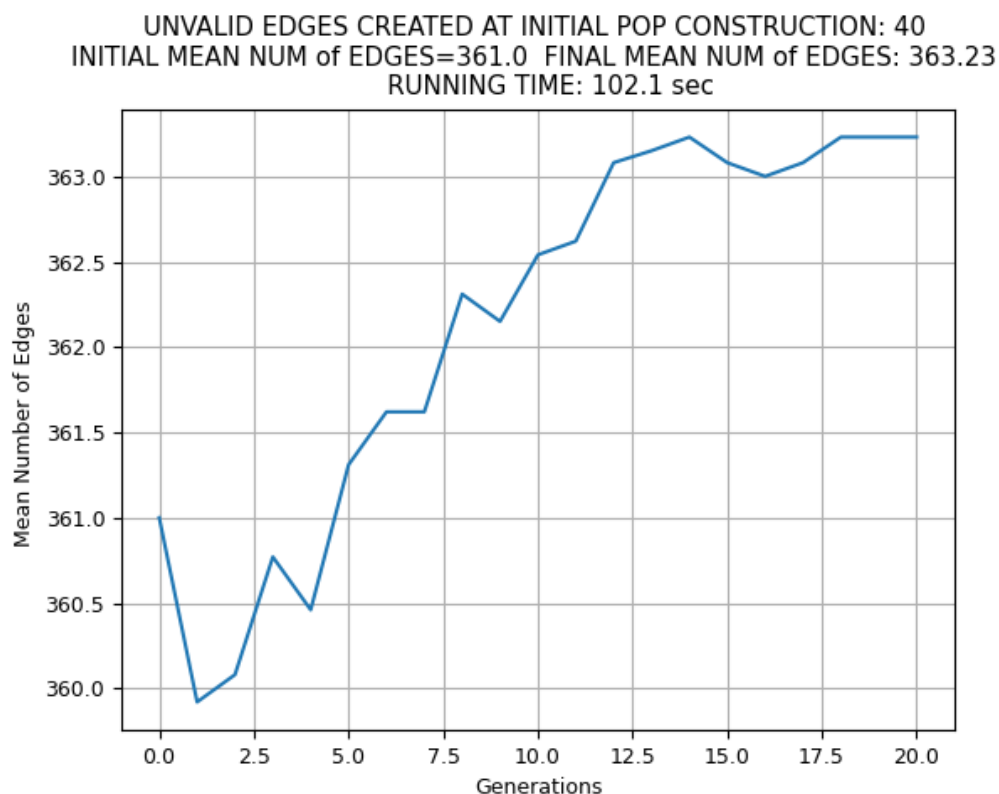
Εικόνα 147 Σενάριο 2.5-Αλγόριθμος Lee

Σενάριο 2.6- Γράφος Ορατότητας με 202 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	13
Συνολικές μεταλλάξεις/Εγκυρες	5/1
Συχνότητα μετάλλαξης	0.4
Διαφορά αποστάσεων αρχικού γράφου	3.8%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	0.1%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	0.1%
Συνολικές ακμές (Γενετικός Αλγ. #2)	360
Συνολικές ακμές (Full Graph)	2882
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	995.371
Μήκος μονοπατιού (Full Graph)	994.608
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	102.1 sec
Χρόνος εκτέλεσης (Full Graph)	251.8 sec
Χρόνος εκτέλεσης (Lee)	3.2 sec

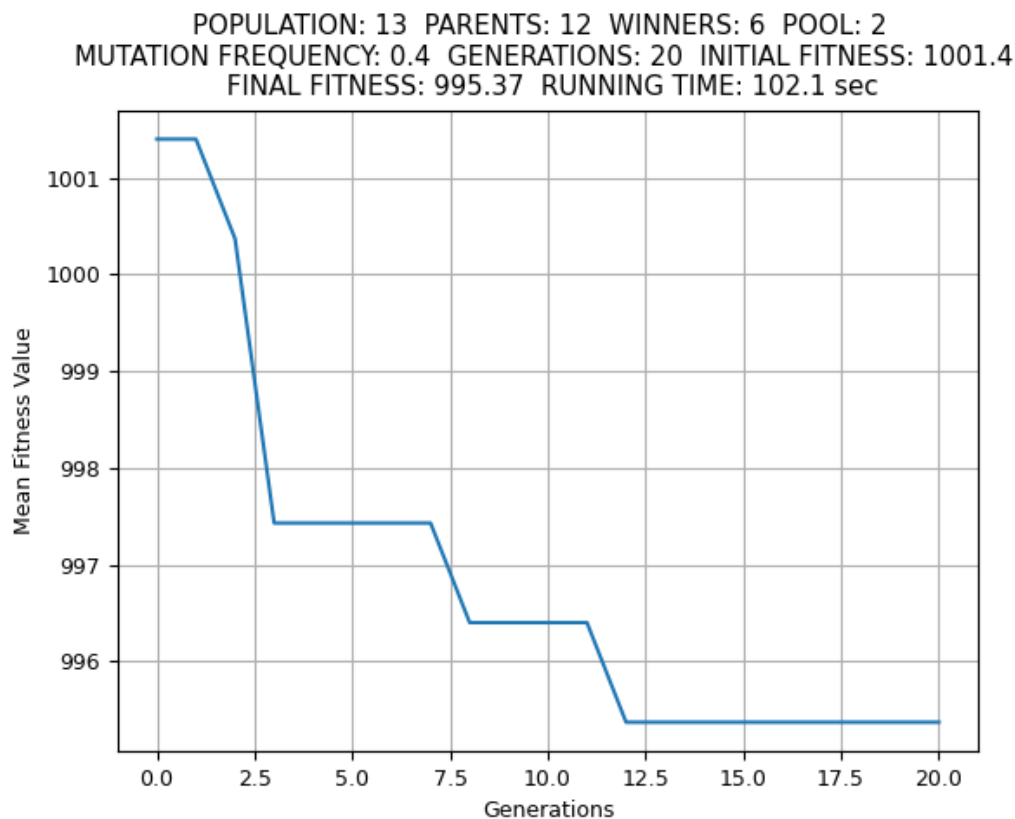
Πίνακας 20 Σενάριο 2.6



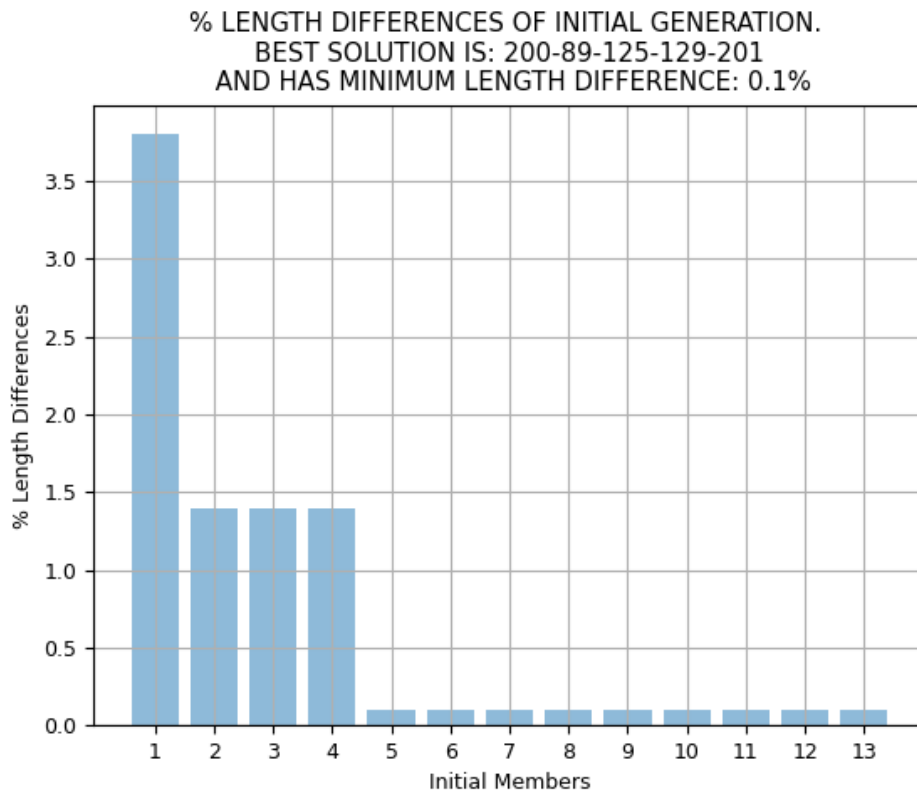
Εικόνα 148 Σενάριο 2.6-Πλήθος επαναλήψεων έως όπου ο γράφος συνδεθεί για πρώτη φορά.



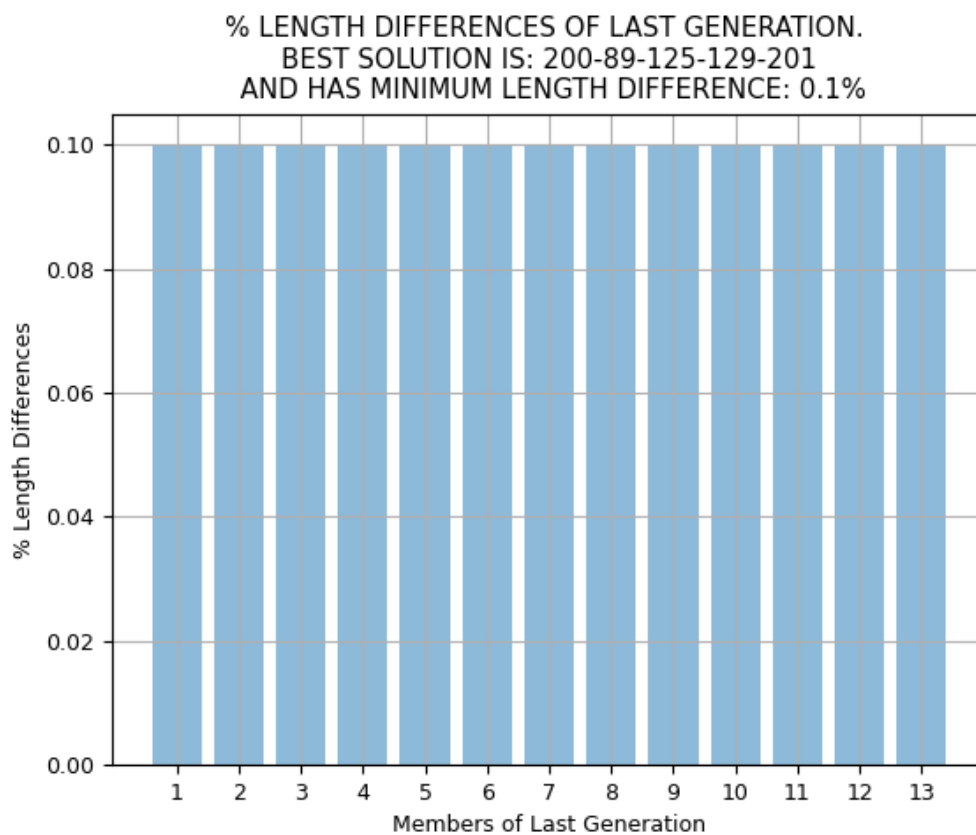
Εικόνα 149 Σενάριο 2.6-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά



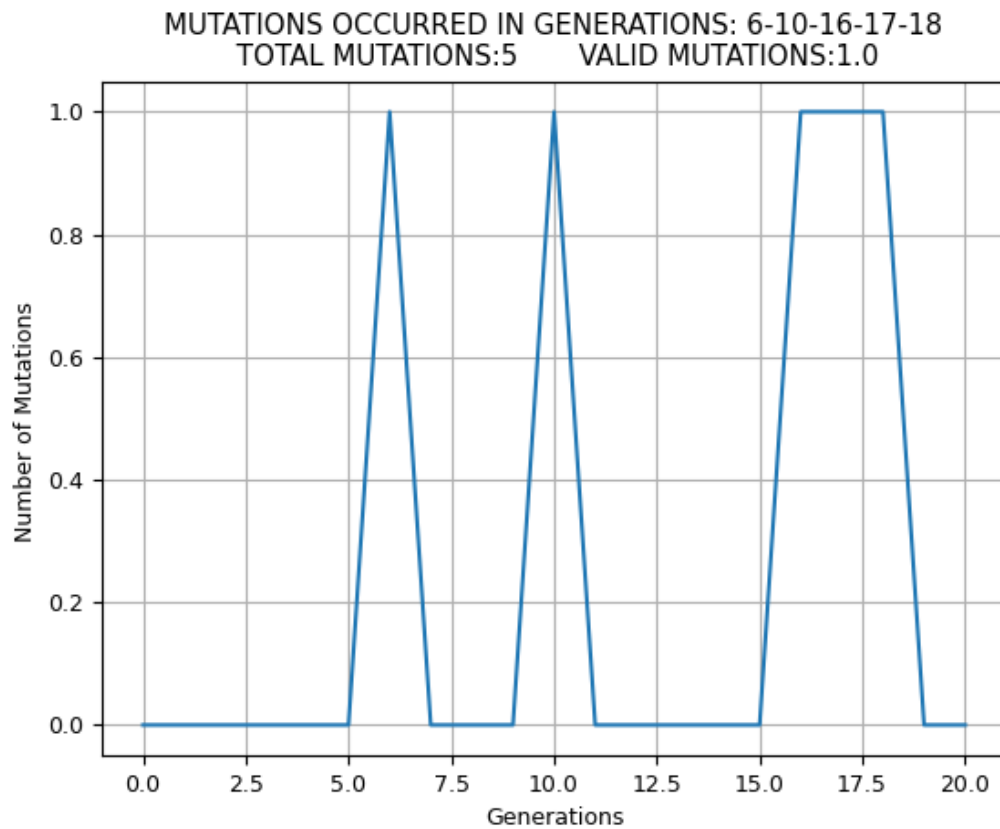
Εικόνα 150 Σενάριο 2.6-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



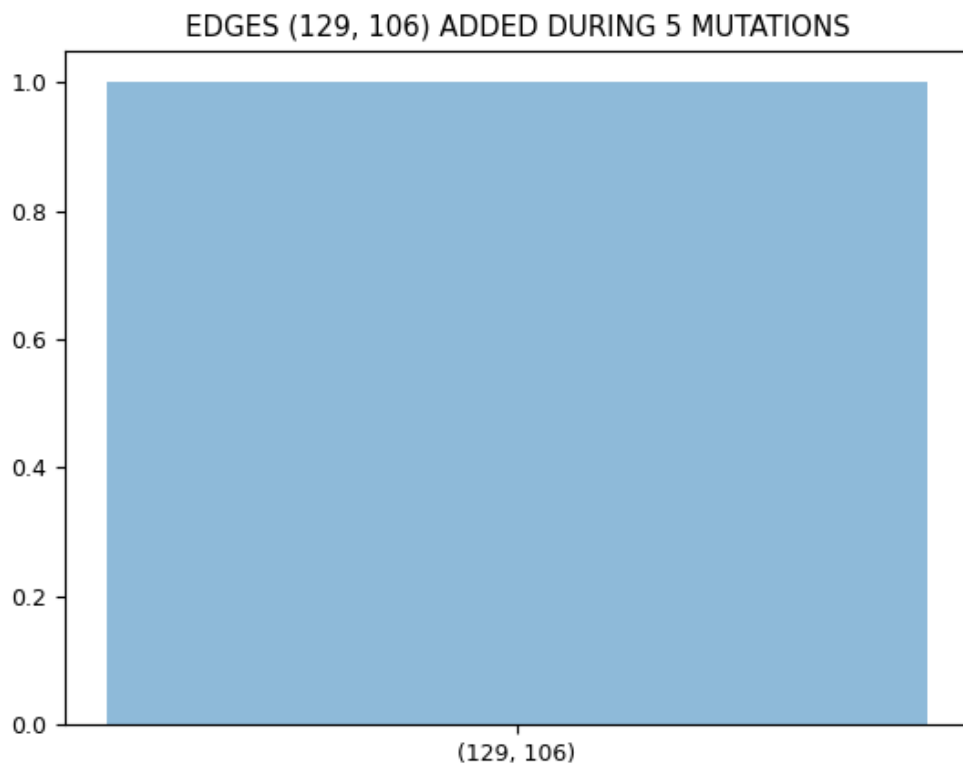
Εικόνα 151 Σενάριο 2.6-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση



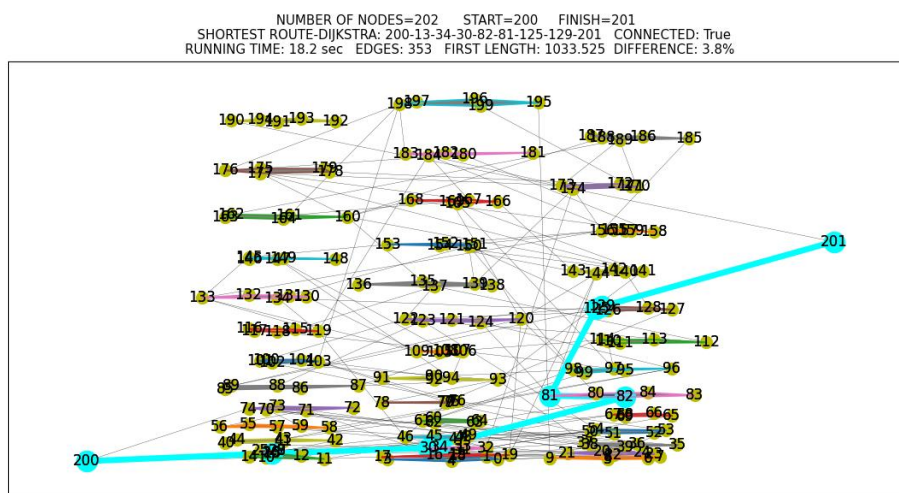
Εικόνα 152 Σενάριο 2.6-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση



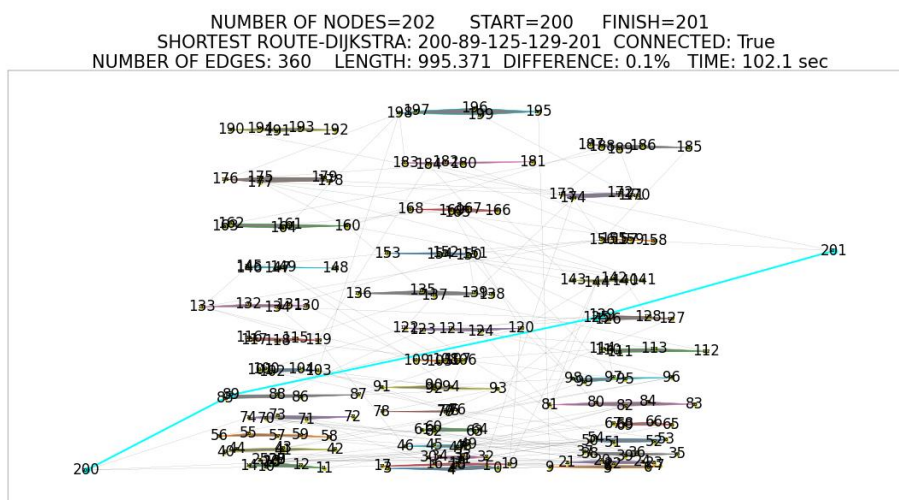
Εικόνα 153 Σενάριο 2.6-Πλήθος μεταλλάξεων ανά γενιά



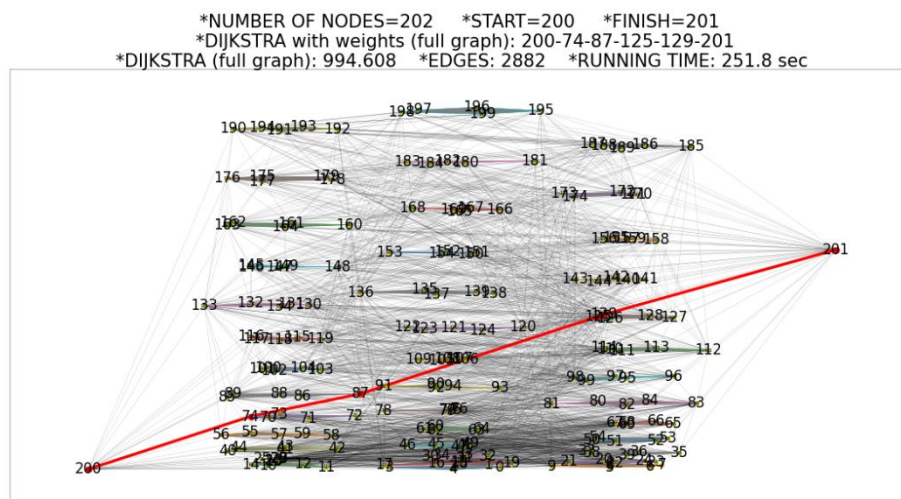
Εικόνα 154 Σενάριο 2.6-Ακμές οι οποίες προστέθηκαν στη φάση της μετάλλαξης



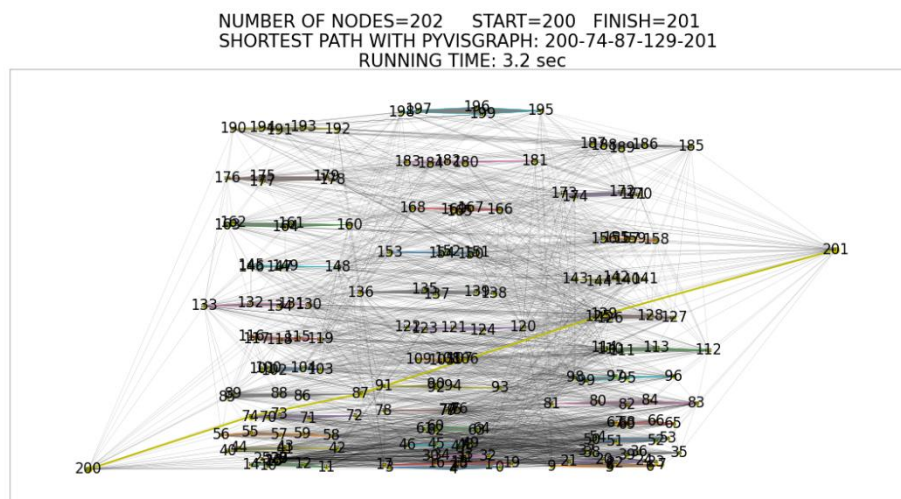
Εικόνα 155 Σενάριο 2.6-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 156 Σενάριο 2.6-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



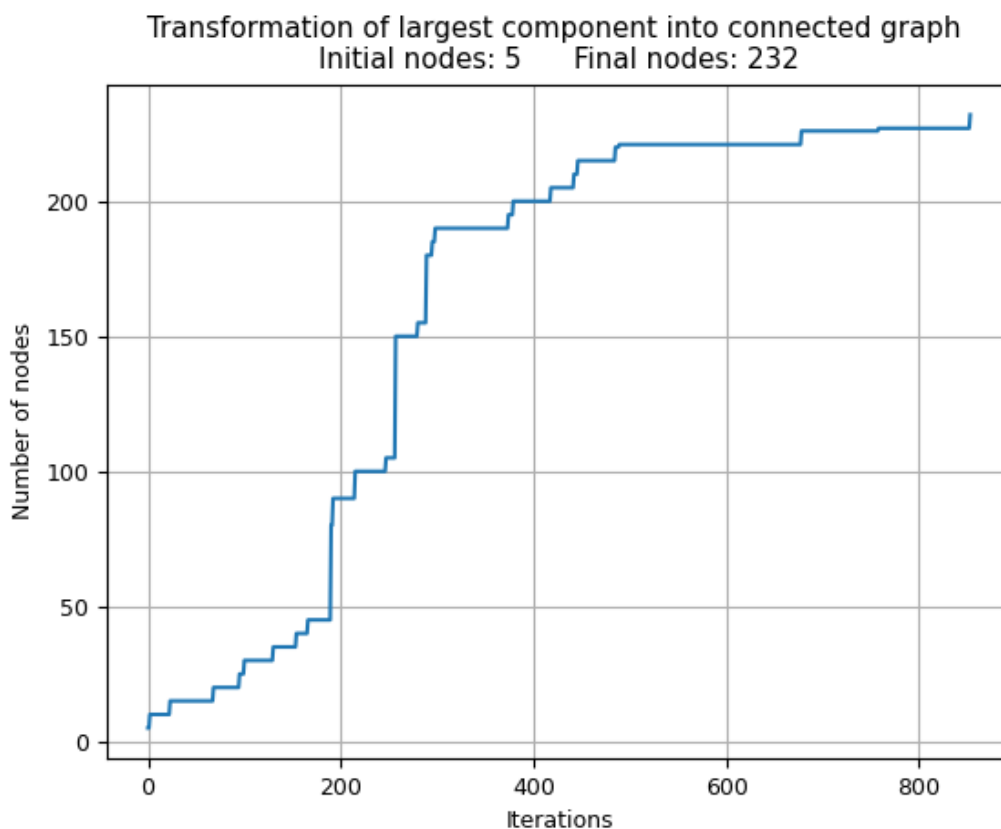
Εικόνα 157 Σενάριο 2.6-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



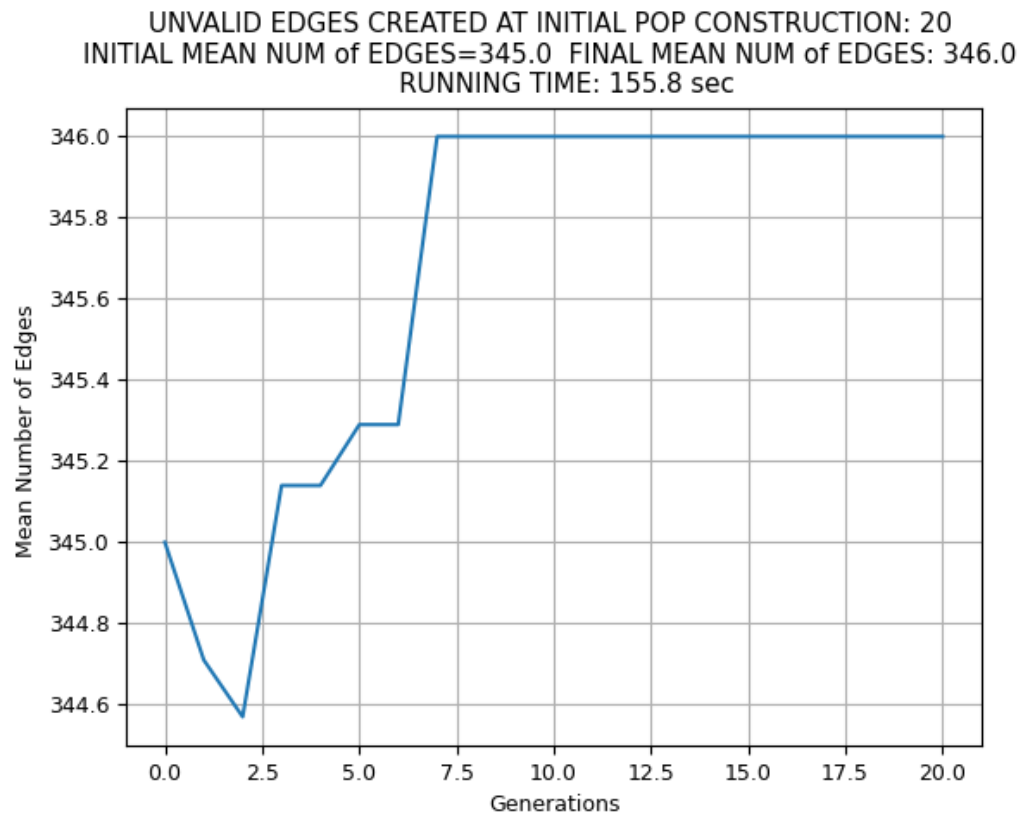
Εικόνα 158 Σενάριο 2.6-Αλγόριθμος Lee

Σενάριο 2.7- Γράφος Ορατότητας με 232 κόμβους	
Αριθμός γενεών	20
Πληθυσμός	7
Συνολικές μεταλλάξεις/Εγκυρες	0/0
Συχνότητα μετάλλαξης	0.2
Διαφορά αποστάσεων αρχικού γράφου	36.3%
Μικρότερη διαφορά αποστάσεων αρχικού πληθυσμού	0.4%
Μικρότερη διαφορά αποστάσεων τελικού πληθυσμού	1.8%
Συνολικές ακμές (Γενετικός Αλγ. #2)	346
Συνολικές ακμές (Full Graph)	3983
Μήκος μονοπατιού (Γενετικός Αλγ. #2)	1234.574
Μήκος μονοπατιού (Full Graph)	1212.717
Χρόνος εκτέλεσης (Γενετικός Αλγ. #2)	155.8 sec
Χρόνος εκτέλεσης (Full Graph)	554.5 sec
Χρόνος εκτέλεσης (Lee)	4.2 sec

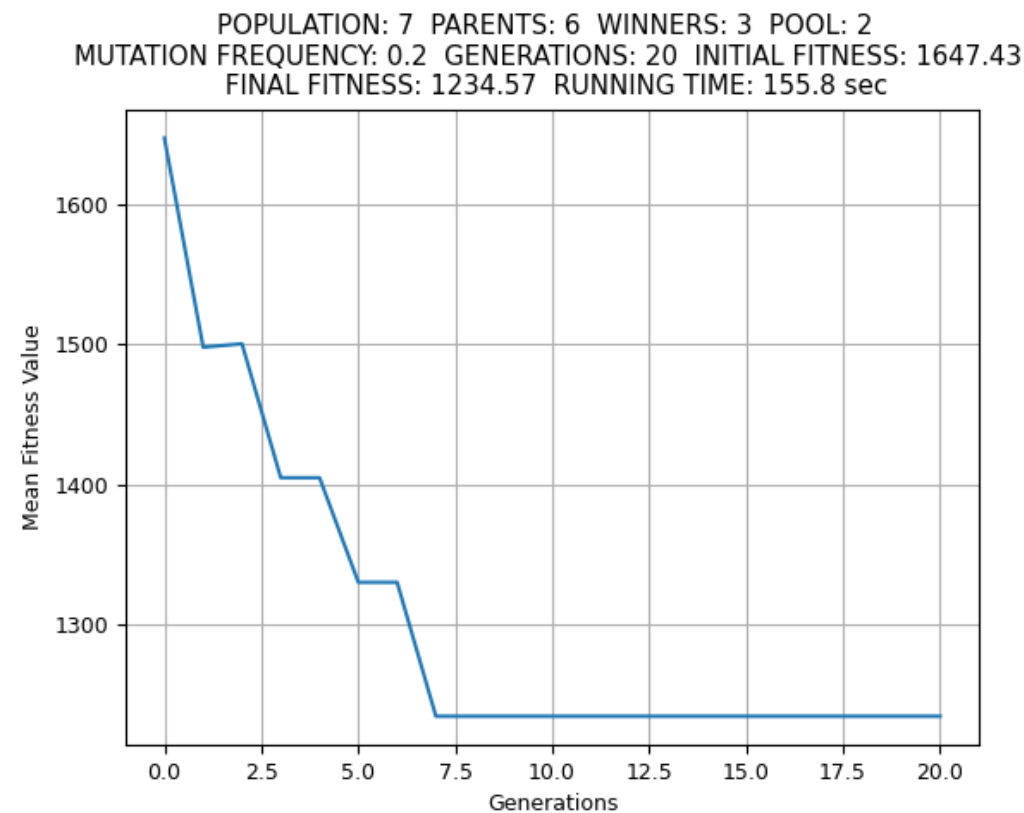
Πίνακας 21 Σενάριο 2.7



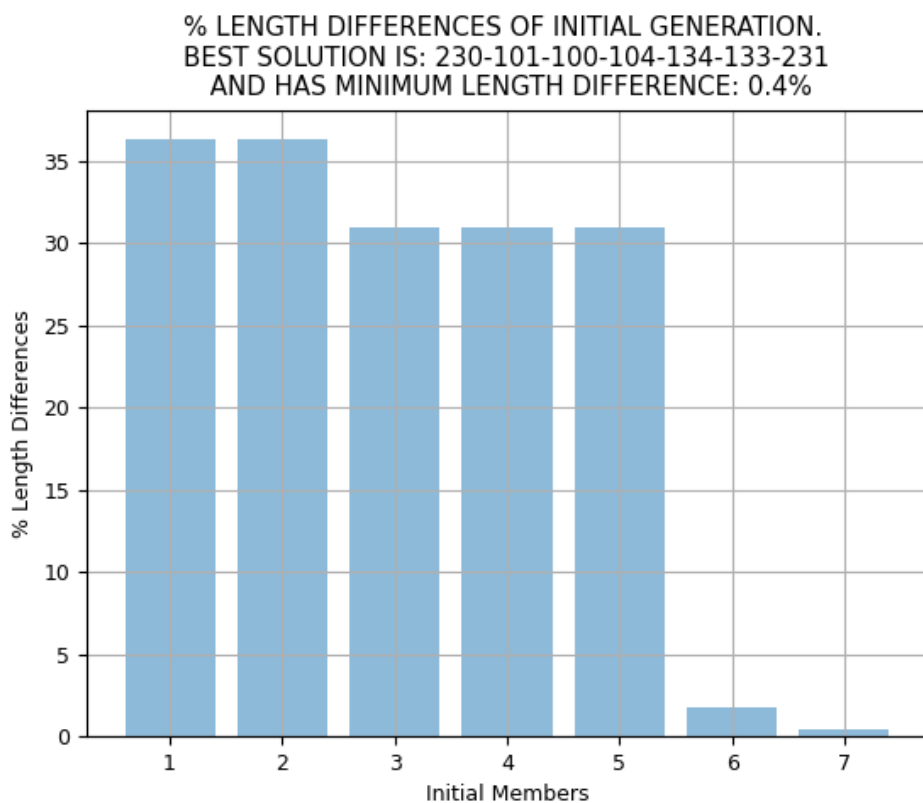
Εικόνα 159 Σενάριο 2.7-Πλήθος επαναλήψεων έως ότου ο γράφος συνδεθεί για πρώτη φορά.



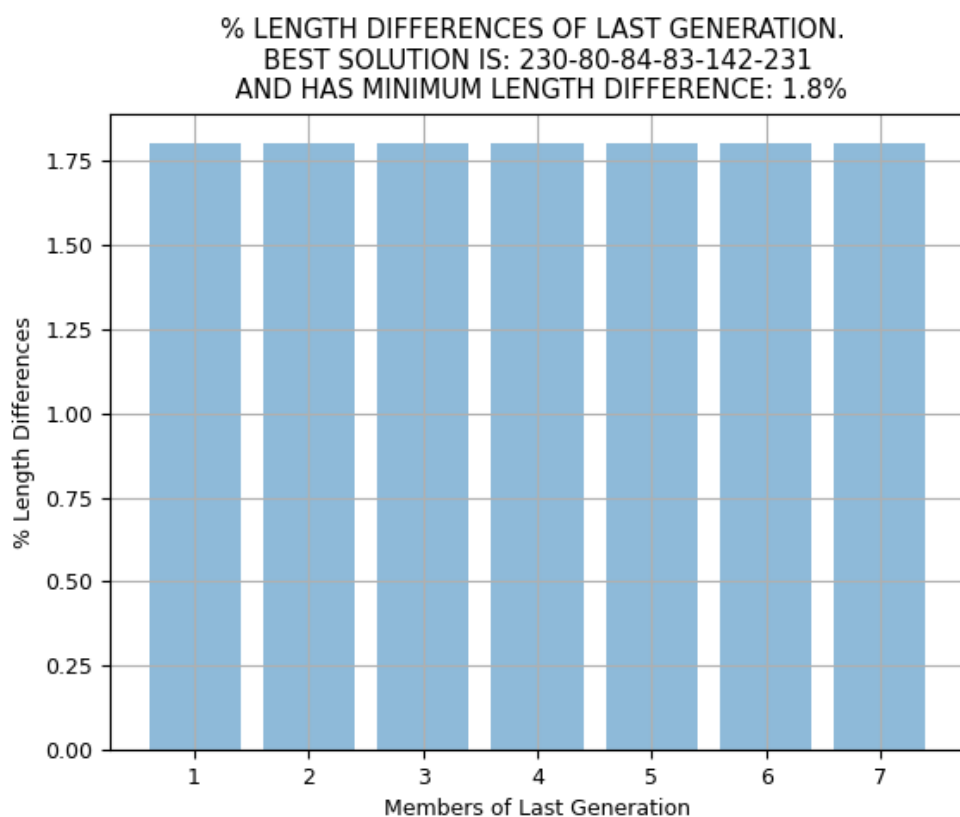
Εικόνα 160 Σενάριο 2.7-Διάγραμμα μεταβολής μέσου όρου ακμών που προστίθενται ανά γενιά



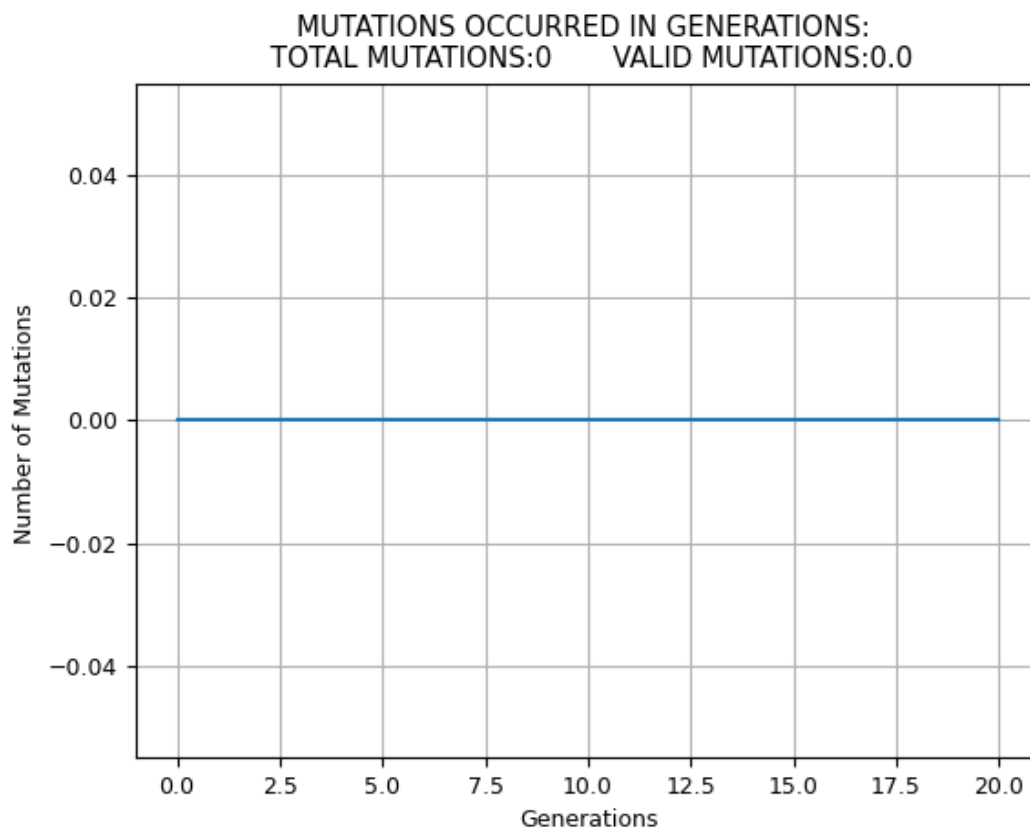
Εικόνα 161 Σενάριο 2.7-Διάγραμμα μεταβολής μέσου όρου τιμών αντικειμενικής συνάρτησης



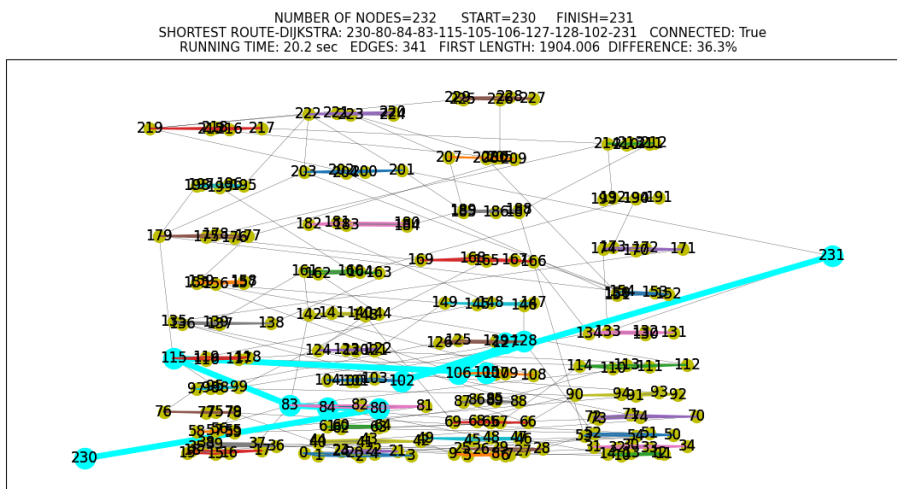
Εικόνα 162 Σενάριο 2.7-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του αρχικού πληθυσμού και στη βέλτιστη λύση



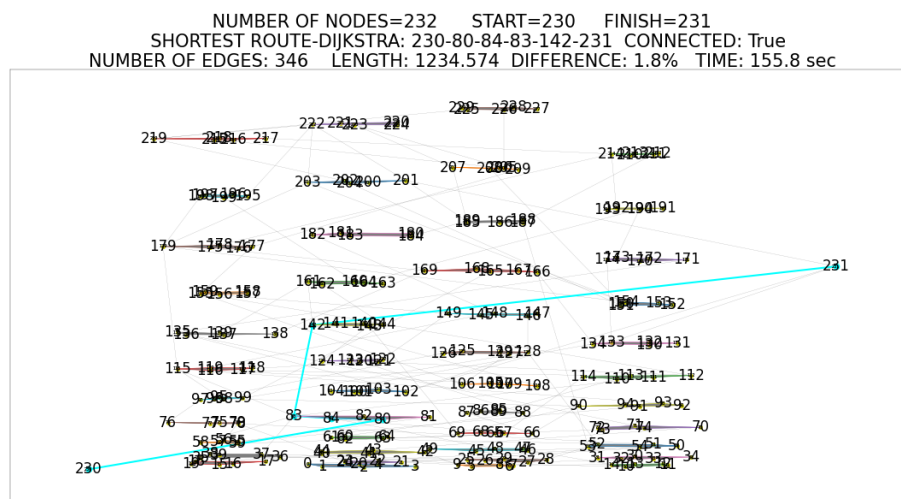
Εικόνα 163 Σενάριο 2.7-Ποσοστιαίες διαφορές ανάμεσα στις τιμές αντικειμενικής συνάρτησης των μελών του τελικού πληθυσμού και στη βέλτιστη λύση



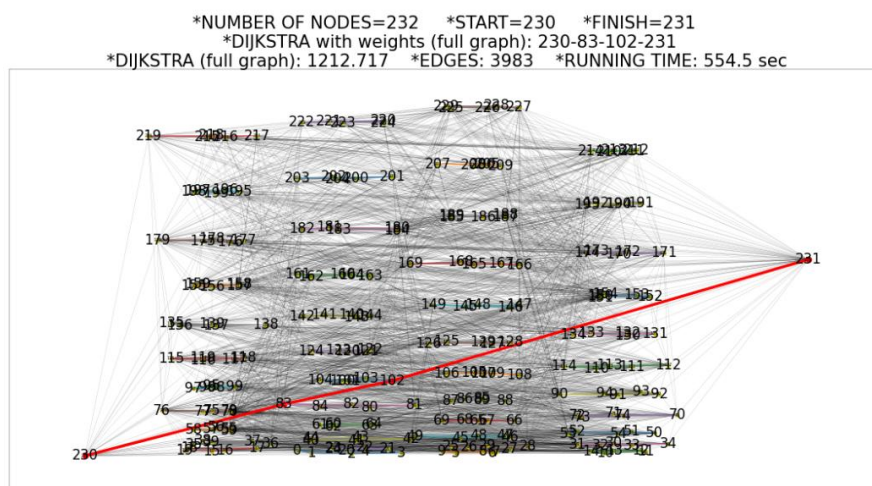
Εικόνα 164 Σενάριο 2.7-Πλήθος μεταλλάξεων ανά γενιά



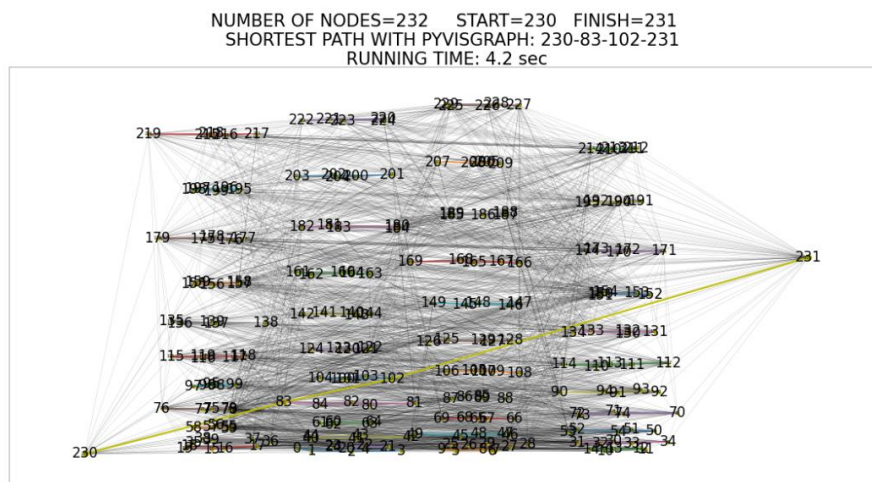
Εικόνα 165 Σενάριο 2.7-Αρχικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra



Εικόνα 166 Σενάριο 2.7-Τελικός γράφος. Απεικονίζεται και το μονοπάτι που επιστρέφει ο αλγόριθμος Dijkstra

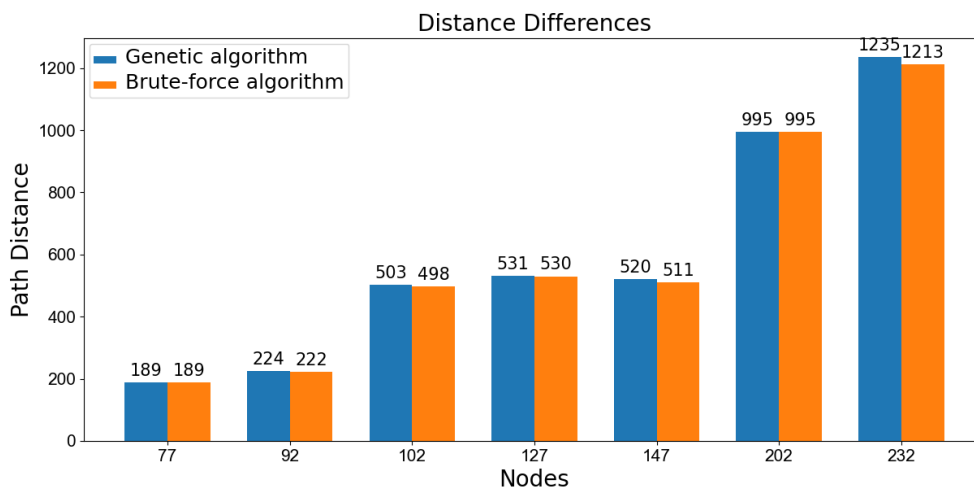


Εικόνα 167 Σενάριο 2.7-Πλήρης γράφος. Με κόκκινο χρώμα η βέλτιστη λύση



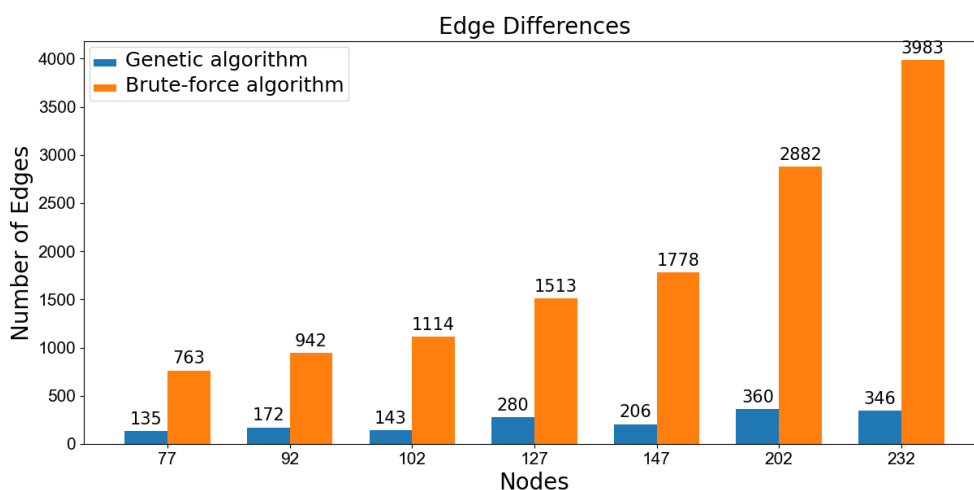
Εικόνα 168 Σενάριο 2.7-Αλγόριθμος Lee

Στην Εικόνα 169 απεικονίζονται οι διαφορές ανάμεσα στα μήκη των διαδρομών που υπολογίζει ο Γενετικός Αλγόριθμος #2 και σε αυτά που υπολογίζει η απλοϊκή μέθοδος (βέλτιστες λύσεις):



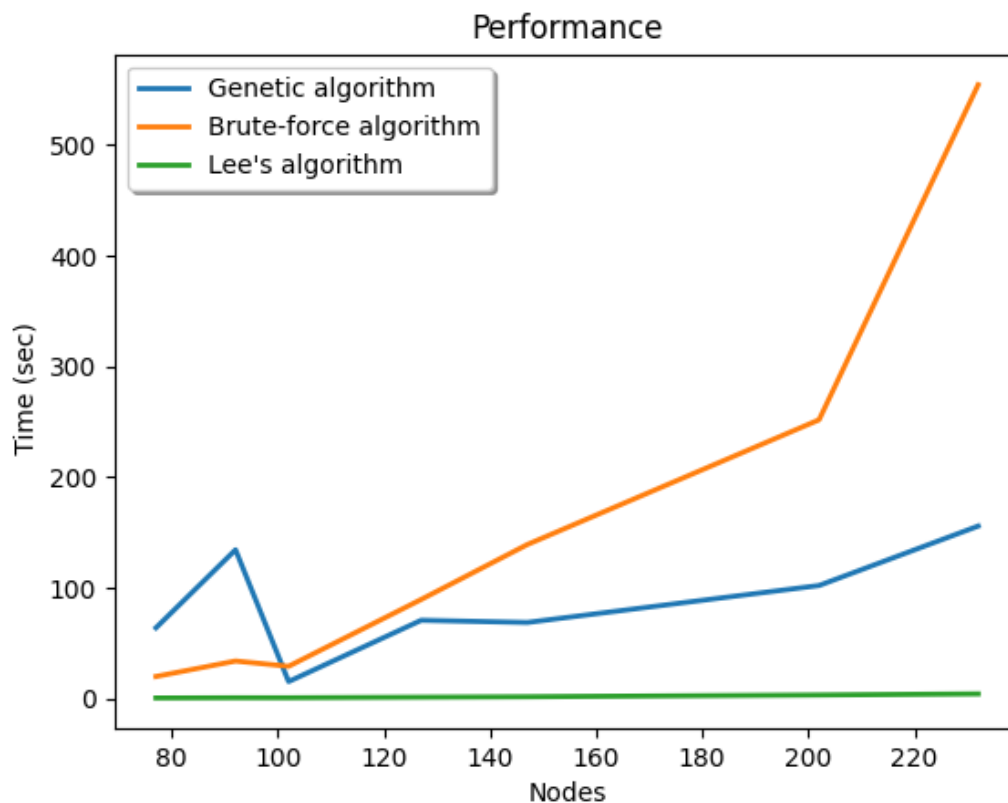
Εικόνα 169 Διαφορές αποστάσεων μονοπατιών

Στην Εικόνα 170 απεικονίζονται οι διαφορές ανάμεσα στο πλήθος των ακμών που χρειάζεται να υπολογίσει ο Γενετικός Αλγόριθμος #2 και στο πλήθος που χρειάζεται να υπολογίσει η απλοϊκή μέθοδος:



Εικόνα 170 Πλήθος ακμών που υπολογίζει κάθε αλγόριθμος

Στο παρακάτω διάγραμμα φαίνεται η επίδοση του Γενετικού Αλγορίθμου #2 (Genetic algorithm), της απλοϊκής μεθόδου (brute-force algorithm) και του αλγορίθμου Lee¹⁹ (Lee's algorithm) σε συνάρτηση με το πλήθος των κόμβων:



Εικόνα 171 Επίδοση Γενετικού Αλγορίθμου #2 (Genetic algorithm), απλοϊκής μεθόδου (brute-force algorithm) και αλγορίθμου Lee (Lee's algorithm)

¹⁹ Η κατασκευή του γράφου ορατότητας σύμφωνα με τον αλγόριθμο Lee και η εύρεση του συντομότερου μονοπατιού έγινε με χρήση του πακέτου Pyvisgraph.

8 Επίλογος

Στα πλαίσια της παρούσας διπλωματικής σχεδιάστηκαν και υλοποιήθηκαν δύο γενετικοί αλγόριθμοι που επιλύουν το πρόβλημα σχεδιασμού πλοήγησης διαμέσου πολυγωνικών εμποδίων. Ο Γενετικός Αλγόριθμος #1, αφού κατασκευάσει το γράφο ορατότητας με την απλοϊκή μέθοδο (naïve method) στη συνέχεια χρησιμοποιεί γενετικές διαδικασίες για να υπολογίσει το συντομότερο μονοπάτι διαμέσου πολυγωνικών εμποδίων. Ο τρόπος υλοποίησης βασίστηκε στην κωδικοποίηση ατόμων του πληθυσμού με πίνακες, τα στοιχεία των οποίων αποτελούν κόμβους του γράφου ορατότητας που δημιουργείται από τις κορυφές των πολυγωνικών εμποδίων καθώς και από τα σημεία εκκίνησης-τερματισμού.

Ο Γενετικός Αλγόριθμος #2 αρχικά χρησιμοποιεί γενετικές διαδικασίες για να κατασκευάσει το μειωμένο γράφο ορατότητας και στη συνέχεια υπολογίζει τη συντομότερη διαδρομή καλώντας τον αλγόριθμο Dijkstra. Σε αυτή την περίπτωση τα μέλη του πληθυσμού αντιπροσωπεύουν γράφους οι οποίοι κωδικοποιούνται με χρήση των πινάκων γειτνίασης που αντιστοιχούν σε καθένα από αυτούς.

8.1 Σύνοψη και συμπεράσματα για τον Γενετικό Αλγόριθμο #1

Αρχικά ο Αλγόριθμος #1 δέχεται ως είσοδο τον επιθυμητό από το χρήστη αριθμό εμποδίων καθώς και το επιθυμητό πλήθος κορυφών (≥ 3) ανά εμπόδιο. Στη συνέχεια καλεί επαναληπτικά τη συνάρτηση `generatePolygon()`, η οποία επιστρέφει ένα τυχαίο πολύγωνο στο επίπεδο, τόσες φορές όσες έχει δηλώσει ο χρήστης. Η πρωτοτυπία του αλγορίθμου που κατασκευάσαμε έγκειται στο γεγονός ότι οι συντεταγμένες των κορυφών των πολυγώνων δεν είναι γνωστές εξ αρχής, οπότε ο ίδιος ο αλγόριθμος θα πρέπει να εξετάζει αν κάποια από τα πολυγωνικά εμπόδια τέμνονται μεταξύ τους, άρα και να τα απορρίπτει καλώντας τη συνάρτηση `check_polygons()`. Ακολούθως αναπτύξαμε μια τεχνική με την οποία ανακαλύπτουμε ποιες από τις ακμές που ενώνουν ανά δύο τις κορυφές διέρχονται μέσα από τα πολυγωνικά εμπόδια καλώντας τη συνάρτηση `check_crosses()`. Για να μειώσουμε όσο το δυνατόν περισσότερο το πλήθος των ακμών καλούμε στη συνέχεια τη συνάρτηση `remove_zero_length()`, η οποία αφαιρεί τις ακμές με μηδενικό μήκος και στη συνέχεια κατασκευάζουμε το γράφο ορατότητας που προκύπτει από τις κορυφές των πολυγώνων καθώς και τα επίσης τυχαία σημεία εκκίνησης-τερματισμού. Μάλιστα έχουμε λάβει μέτρα ώστε τα τελευταία να μην βρίσκονται εντός των πολυγωνικών εμποδίων. Έπειτα καλούμε τη συνάρτηση `visibility_matrix()`. Η

συνάρτηση κατασκευάζει τον πίνακα γειτνίασης του γράφου ορατότητας ο οποίος είναι συμμετρικός ως προς τη κύρια διαγώνιο. Γνωρίζοντας τον πίνακα γειτνίασης μπορούμε να εντοπίσουμε τους γείτονες του κόμβου εκκίνησης οι οποίοι χρησιμοποιούνται αργότερα κατά την αρχικοποίηση του πληθυσμού. Καλώντας στη συνέχεια τη συνάρτηση `matrix_with_weights()` υπολογίζουμε τον πίνακα, κάθε στοιχείο του οποίου αποτελεί το μήκος όλων των ζευγών ακμών του γράφου. Στόχος είναι καλώντας τη συνάρτηση `find_max_weight()` να υπολογίσουμε το άθροισμα όλων των μηκών ώστε να αποδώσουμε την τιμή αυτή στην αντικειμενική συνάρτηση που χαρακτηρίζει όποια από τα μέλη του αρχικού πληθυσμού δεν αποτελούν έγκυρα μονοπάτια. Δημιουργούμε τμήμα του αρχικού πληθυσμού καλώντας τη συνάρτηση `create_starting_pool()`. Φροντίζουμε ώστε το αρχικό τμήμα των μελών του αρχικού πληθυσμού να αποτελείται από τον κόμβο εκκίνησης και οποιονδήποτε από τους γειτονικούς του κόμβους. Με αυτό τον τρόπο βοηθούμε τον αλγόριθμο ώστε μετά την ολοκλήρωση των γενετικών διαδικασιών να καταλήξει σε λύση. Ο αρχικός πληθυσμός δημιουργείται με τυχαίο τρόπο μέσω της κλήσης της συνάρτησης `create_population()` και τα μέλη του ενδέχεται είτε να αποτελούν είτε να μην αποτελούν έγκυρα μονοπάτια. Στην πρώτη περίπτωση υπολογίζουμε το μήκος του μονοπατιού και αποδίδουμε την τιμή αυτή στην αντικειμενική συνάρτηση, ενώ στη δεύτερη θέτουμε την τιμή της αντικειμενικής συνάρτησης ίσης με την τιμή που επιστρέφει η συνάρτηση `find_max_weight()`. Στη συνέχεια υπολογίζουμε το μέσο όρο των τιμών της αντικειμενικής συνάρτησης για κάθε μέλος του πληθυσμού. Εδώ θα πρέπει να σημειώσουμε ότι το πλήθος των μελών του αρχικού πληθυσμού δεν είναι γνωστό εξ αρχής αλλά εξαρτάται κάθε φορά από το πλήθος των γειτόνων του κόμβου εκκίνησης. Μετά από πειράματα παρατηρήσαμε ότι όταν το πλήθος των γειτόνων ήταν πολύ μικρό τότε ο αρχικός πληθυσμός αποτελούνταν από λίγα άτομα οπότε ο γενετικός αλγόριθμος αδυνατούσε να μεταβάλει μέσω των γενετικών διαδικασιών το μέσο όρο των τιμών της αντικειμενικής συνάρτησης των μελών οπότε η λύση που επέστρεφε ήταν ένα μη έγκυρο μονοπάτι με τιμή αντικειμενικής συνάρτησης ίσο με `max_weight`. Για αυτό το λόγο ο αλγόριθμος λαμβάνει μέτρα ώστε όταν εντοπίσει ότι το πλήθος των γειτόνων του αρχικού κόμβου είναι μικρότερο από πέντε, τότε μέσω ενός τριπλού βρόγχου σχεδόν τριπλασιάζει τα μέλη του αρχικού πληθυσμού.

Αφού αρχικοποιηθεί ο πληθυσμός στη συνέχεια ξεκινάει ο κυρίως βρόχος του γενετικού αλγορίθμου. Επιλέγονται τυχαία τόσοι γονείς όσο το πλήθος των μελών του πληθυσμού. Σημειώνουμε ότι το πλήθος των ατόμων του πληθυσμού διατηρείται σταθερό σε κάθε

γενιά. Ο αλγόριθμος λαμβάνει μέτρα προκειμένου κάθε φορά να επιλέγει άρτιο αριθμό γονέων ώστε έπειτα καλώντας τη συνάρτηση `begin_tournament()` να συγκρίνει σε σχέση με την τιμή της αντικειμενικής συνάρτησης ανά δύο τα άτομα του πληθυσμού και να επιλέγει αυτό με τη μικρότερη τιμή, αφού στόχος είναι να βρεθεί η συντομότερη διαδρομή. Καλείται έπειτα η συνάρτηση `create_pool()` η οποία επιλέγει τυχαία ζευγάρια πάνω στα οποία θα εφαρμοστεί η διαδικασία της διασταύρωσης. Ο αλγόριθμος φροντίζει και πάλι ώστε το πλήθος των ζευγαριών που περιέχει ο πίνακας `pool[]` να είναι άρτιος αριθμός. Έχουμε επιλέξει ώστε από κάθε ζευγάρι να προκύπτει μόνο ένα παιδί. Μετά την ολοκλήρωση της διαδικασίας της διασταύρωσης ο αλγόριθμος ελέγχει αν ο απόγονος αποτελεί έγκυρο μονοπάτι. Αν ναι, τότε εφόσον ικανοποιείται η συνθήκη όπου ένας τυχαία παραγόμενος αριθμός είναι μεγαλύτερος από τη συχνότητα μετάλλαξης, ξεκίνα η διαδικασία της μετάλλαξης. Στόχος των μεταλλάξεων είναι ο γενετικός αλγόριθμος να εξετάσει και άλλες πιθανές λύσεις με μικρότερη τιμή αντικειμενικής συνάρτησης. Η κωδικοποίηση των ατόμων ως πίνακες οι οποίοι στη γλώσσα προγραμματισμού Python έχουν την ιδιότητα ότι μπορούν εύκολα να τροποποιηθούν, μας επιτρέπει να κατασκευάσουμε αρκετά είδη μετάλλαξης. Εμείς παρουσιάσαμε τέσσερα είδη μετάλλαξης τα οποία και εφαρμόζονται στους απογόνους καλώντας τις συναρτήσεις `swap()`, `shuffle()`, `delete()` και `cluster_swap2()`. Τα πρώτα τρία είδη μεταλλάξεων εφαρμόζονται σειριακά πάνω στους απογόνους ενώ το τέταρτο είδος μετάλλαξης `cluster_swap2()` επιλέξαμε να το εξετάσουμε ξεχωριστά. Στη σχεδίαση της μετάλλαξης `cluster_swap2()` μας κατηύθυνε το γεγονός ότι καλούμε τη συνάρτηση `generatePolygon()` χρησιμοποιώντας κατάλληλες παραμέτρους ώστε τα πολυγωνικά εμπόδια να τοποθετούνται στο επίπεδο σύμφωνα με κάποια δομή και όχι τυχαία. Έπειτα καλώντας την κατάλληλη συνάρτηση προχωρούμε σε συσταδοποίηση των εμποδίων, ενώ στη συνέχεια υπολογίζοντας τις συντεταγμένες των κεντροειδών, ανακαλύπτουμε ποιοι κόμβοι βρίσκονται πιο κοντά στην ευθεία που ενώνει τους κόμβους εκκίνησης-τερματισμού. Το σύνολο αυτών των κόμβων αποτελούν τους κόμβους οι οποίοι είναι περισσότερο πιθανό να βρίσκονται στο τελικό συντομότερο μονοπάτι, οπότε βοηθούμε τον αλγόριθμο να ψάχνει σε μικρότερο εύρος κόμβων.

Όπως αναφέραμε παραπάνω, οι συντεταγμένες των κορυφών των πολυγωνικών εμποδίων δεν είναι γνωστές εξ αρχής. Αυτή η τυχαιότητα στον τρόπο λειτουργίας του αλγορίθμου απαιτεί μεγάλο πλήθος ελέγχων (έλεγχος για έγκυρα πολύγωνα, έλεγχος ώστε τα σημεία εκκίνησης-τερματισμού να βρίσκονται εκτός των εμποδίων, έλεγχος ώστε οι ακμές του γράφου να μην τέμνουν τα εμπόδια, έλεγχος ώστε τα πολύγωνα να μην εκφυλίζονται σε

γραμμές, κλπ.) για αυτό και αν και στις περισσότερες των περιπτώσεων ο αλγόριθμος καταφέρνει να δώσει αξιόπιστα αποτελέσματα, ο χρόνος εκτέλεσης του αλγορίθμου είναι αυξημένος.

Επιπρόσθετα ο χρόνος εκτέλεσης επιβαρύνεται από το γεγονός ότι κατά το στάδιο της διασταύρωσης ο αλγόριθμος εξετάζει αν το μήκος των πινάκων που αντιστοιχούν στους γονείς είναι μεγαλύτερο του δύο. Με αυτό τον τρόπο προφυλάσσουμε τον πληθυσμό από το να εισαχθούν σε αυτόν άτομα με λανθασμένο κόμβο προορισμού. Επιπλέον έλεγχοι κατά το στάδιο της διασταύρωσης ώστε ο κόμβος που επιλέγεται για να διαχωριστεί το χρωμόσωμα σε δύο μέρη να είναι διαφορετικός από τους κόμβους εκκίνησης-τερματισμού αυξάνουν ακόμη περισσότερο το χρόνο εκτέλεσης.

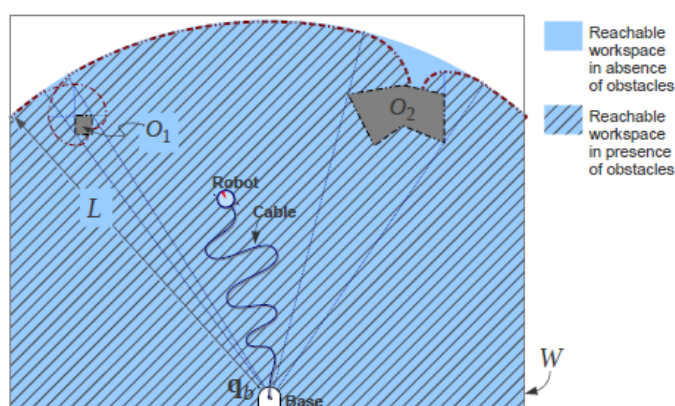
Ένα σημείο στο οποίο ο Γενετικός Αλγόριθμος #1 διαφοροποιείται από τον αλγόριθμο Dijkstra είναι ότι με την κατάλληλη τροποποίηση στον τρόπο υπολογισμού των μηκών των μονοπατιών, μπορεί να καταλήξει σε λύση ακόμη και όταν στο γράφο υπάρχουν αρνητικά βάρη σε αντίθεση με τον αλγόριθμο Dijkstra ο οποίος σε περίπτωση που στον γράφο υπάρχει κύκλος αρνητικού βάρους, το συντομότερο μονοπάτι θα ελαττώνεται συνεχώς, οπότε ο αλγόριθμος δεν καταλήγει σε λύση. Η απαραίτητη τροποποίηση για τον υπολογισμό των μηκών των νέων μονοπατιών απαιτεί τα νέα μήκη να υπολογίζονται μέσω του πίνακα με τα βάρη και όχι με τον τρόπο που ακολουθήσαμε εμείς κατά την υλοποίηση του αλγορίθμου μας (δημιουργία αντικειμένου LineString από τις συντεταγμένες των κόμβων, εύρεση μήκους με χρήση του line.length, κλπ.). Το αρνητικό βάρος στο πρόβλημα εύρεσης του πιο σύντομου μονοπατιού θα μπορούσε για παράδειγμα να συμβολίζει μια ακμή με κατηφορική κλίση ή κάποιο άλλο χαρακτηριστικό της ακμής.

Επίσης ο Γενετικός Αλγόριθμος #1 βρίσκει κάθε στιγμή πολλές διαφορετικές λύσεις (διαφορετικές διαδρομές), οι οποίες ενδέχεται να διαφέρουν ελάχιστα ως προς το μήκος τους. Έτσι για παράδειγμα όταν θέλουμε να δρομολογήσουμε πολλά οχήματα-ρομπότ από έναν κόμβο εκκίνησης προς ένα κόμβο προορισμού προσέχοντας όμως να μην υπερφορτωθεί η καλύτερη διαδρομή, μπορούμε να τα κατευθύνουμε μέσω των αμέσως επόμενων καλύτερων διαδρομών, οδηγώντας έτσι το δίκτυο σε εξισορρόπηση φορτίου.

Μια ακόμη εφαρμογή στην οποία ο Γενετικός Αλγόριθμος #1 μπορεί να φανεί χρήσιμος είναι στο σύστημα πλοήγησης οχημάτων (Automotive Navigation System). Το σύστημα αυτό έχει ως στόχο την καθοδήγηση του οδηγού του οχήματος ώστε μέσω ηλεκτρονικού χάρτη να φτάσει στον προορισμό του μέσα σε κάποια προθεσμία. Όμως σε συνθήκες

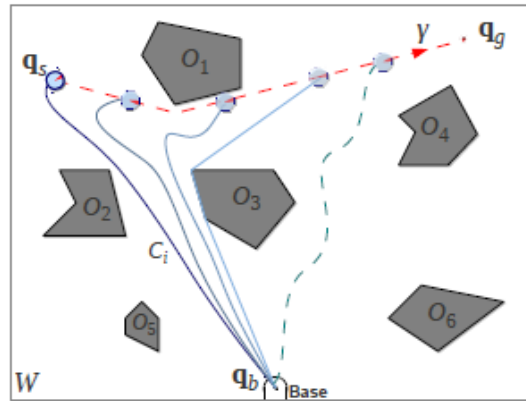
συμφόρησης η δρομολόγηση του οχήματος θα πρέπει να επαναυπολογιστεί όσο το δυνατόν πιο σύντομα, προτού το όχημα φτάσει στην επόμενη διασταύρωση. Οι Γενετικοί Αλγόριθμοι μπορούν να δώσουν λύση στο παραπάνω πρόβλημα αφού κάθε στιγμή έχουν αποθηκευμένες εναλλακτικές διαδρομές μέσα στον πληθυσμό (Kumar, 2009).

Η ιδιότητα του Γενετικού Αλγορίθμου #1 να μπορεί να επιστρέφει κάθε στιγμή πολλαπλές έγκυρες λύσεις (any-time algorithm), μπορεί να φανεί χρήσιμη στο πρόβλημα της πλοήγησης ενός ρομπότ προσδεμένου σε μια σταθερή βάση μέσω ενός καλωδίου μήκους L , μέσα σε ένα περιβάλλον με εμποδία. Όπως φαίνεται και στην παρακάτω εικόνα ο προσιτός χώρος εργασίας περιορίζεται λόγω της ύπαρξης του καλωδίου:



Εικόνα 172 Ο προσιτός χώρος εργασίας για ένα ρομπότ προσδεμένο μέσω ενός καλωδίου μήκους L . Η μπλε περιοχή είναι ο χώρος εργασίας χωρίς την παρουσία εμποδίων (όλος ο χώρος W) ενώ η γραμμοσκιασμένη περιοχή είναι ο χώρος εργασίας με εμποδία. (πηγή: [researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot](https://www.researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot))

Τα καλώδια χρησιμοποιούνται συχνά ως ένας τρόπος για να παρέχεται ενέργεια στο ρομπότ. Επίσης χρησιμοποιούνται και ως επικοινωνιακή σύνδεση κατά τον τηλεχειρισμό του ρομπότ (Salzman, 2015). Ρομπότ αυτού του είδους χρησιμοποιούνται συχνά σε αποστολές αναζήτησης επιζώντων μέσα σε κατεστραμμένους χώρους. Κατά τη σχεδίαση λοιπόν της συντομότερης διαδρομής θα πρέπει να ληφθούν υπόψη τόσο το περιορισμένο μήκος του σχοινιού όσο και η τρέχουσα θέση του ρομπότ. Όπως φαίνεται και στην παρακάτω εικόνα, αν και η διαδρομή γ είναι η συντομότερη διαδρομή μεταξύ των σημείων q_s και q_g , η τελική θέση (q_g) δεν είναι προσβάσιμη λόγω του περιορισμένου μήκους του καλωδίου:

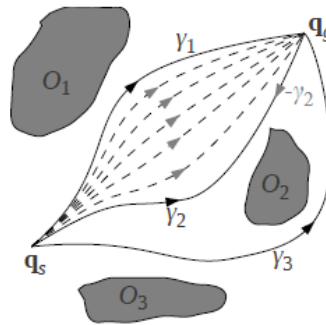


Εικόνα 173 Το ρομπότ δεν μπορεί να διασχίσει τη συντομότερη διαδρομή (γ) μεταξύ των σημείων q_s και q_g με αρχική διαμόρφωση καλωδίου C_i λόγω του περιορισμένου μήκους του καλωδίου. (πηγή: researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot)

Για αυτό θα πρέπει να αναζητηθεί μια εναλλακτική διαδρομή η οποία να είναι όσο το δυνατόν πιο σύντομη και η οποία να λαμβάνει υπόψη τη διαμόρφωση (C_i) του καλωδίου. Με κατάλληλη τροποποίηση της αντικειμενικής συνάρτησης ο Γενετικός Αλγόριθμος #1 θα μπορούσε σε αυτή την περίπτωση να φανεί χρήσιμος. Η αντικειμενική συνάρτηση θα μπορούσε να χρησιμοποιήσει την έννοια της ομοτοπίας (H)²⁰ για να περιγράψει τη διαμόρφωση του καλωδίου κάθε στιγμή. Αν θεωρήσουμε τη δεύτερη παράμετρο της H ως το χρόνο τότε η συνάρτηση H περιγράφει τη συνεχή παραμόρφωση της f στην g δηλαδή στο χρόνο $t=0$ έχουμε τη συνάρτηση f και στο χρόνο $t=1$ έχουμε τη συνάρτηση g (Wikipedia, 2021). Στην παρακάτω εικόνα οι καμπύλες γ_1 και γ_2 είναι ομοτοπικές αφού:

- 1) συνδέουν τα ίδια σημεία αρχής – τέλους,
- 2) η μία μπορεί να παραμορφωθεί συνεχώς στην άλλη χωρίς να ακουμπήσει κανένα εμπόδιο.

²⁰ Έστω $f, g : X \rightarrow Y$ συνεχείς. Λέμε ότι η f είναι ομοτοπική με την g αν υπάρχει συνεχής συνάρτηση $H : X \times [0, 1] \rightarrow Y$ τέτοια ώστε: $H(x, 0) = f(x)$ και $H(x, 1) = g(x)$ (Πλατής, 2017).



Εικόνα 174 Η καμπύλη γ_1 είναι ομοτοπική με την καμπύλη γ_2 αφού υπάρχει μια συνεχής παραμόρφωση από τη μια στην άλλη αλλά όχι με την καμπύλη γ_3 . (πηγή: researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot)

8.2 Σύνοψη και συμπεράσματα για τον Γενετικό Αλγόριθμο #2

Αρχικά ο Αλγόριθμος #2 δέχεται ως είσοδο τον επιθυμητό από το χρήστη αριθμό εμποδίων καθώς και το επιθυμητό πλήθος κορυφών (≥ 3) ανά εμπόδιο. Ο τρόπος κατασκευής των πολυγωνικών εμποδίων είναι ίδιος όπως στον Αλγόριθμο #1 δηλαδή γίνεται καλώντας επαναληπτικά τη συνάρτηση `generatePolygon()`, ενώ καλώντας την `check_polygons()` απορρίπτει όσα πολύγωνα τέμνονται μεταξύ τους.

Στη συνέχεια συνδέονται οι κόμβοι που αντιστοιχούν στις κορυφές των πολυγώνων οπότε δημιουργούνται τόσα ανεξάρτητα τμήματα όσα και το πλήθος των έγκυρων πολυγώνων. Έπειτα προσθέτουμε επαναληπτικά τυχαίες ακμές έως ότου όλα τα ανεξάρτητα τμήματα συνδεθούν για πρώτη φορά και σχηματίσουν ένα γράφο. Κατά τη διαδικασία αυτή ελέγχουμε ώστε οι ακμές που προστίθενται να είναι έγκυρες.

Όταν ολοκληρωθεί η παραπάνω διαδικασία καλούμε τις συναρτήσεις `visibility_matrix()` και `matrix_with_weights()` που έχουμε κατασκευάσει στον Γενετικό Αλγόριθμο #1, οπότε κατασκευάζουμε τον πίνακα γειτνίασης του αρχικού γράφου και υπολογίζουμε τη συντομότερη διαδρομή ώστε να έχουμε ένα μέτρο σύγκρισης. Από τον αρχικό αυτό γράφο θα προέλθουν τα μέλη του πληθυσμού.

Κατά την αρχικοποίηση του πληθυσμού δημιουργούμε έναν επαναληπτικό βρόχο με τόσες επαναλήψεις όσες η μεταβλητή `pop_size`. Ξεκινώντας από τον αρχικό γράφο, σε κάθε επανάληψη προσθέτουμε μια ακμή με τέτοιο τρόπο ώστε το ένα άκρο της να είναι είτε ο κόμβος εκκίνησης είτε ο κόμβος τερματισμού ενώ το άλλο άκρο ένας οποιοσδήποτε κόμβος (εκτός των κόμβων εκκίνησης-τερματισμού). Με αυτό τον τρόπο συνδέουμε όσους περισσότερους γείτονες των κόμβων εκκίνησης-τερματισμού οπότε βοηθούμε τον

αλγόριθμο ώστε να συγκλίνει πιο γρήγορα σε λύση. Όταν μια ακμή περάσει τον έλεγχο εγκυρότητας, συνδέεται στο γράφο και δημιουργείται ένα καινούριο μέλος του αρχικού πληθυσμού. Συμπεραίνουμε ότι κάθε μέλος είναι συνέχεια του προηγούμενου και διαφέρουν είτε ως προς μια ακμή είτε είναι αντίγραφα. Σε κάθε μέλος που δημιουργείται υπολογίζουμε την τιμή της αντικειμενικής συνάρτησης²¹ και στη συνέχεια βρίσκουμε το μέσο όρο των τιμών της κάθε γενιάς.

Αφού αρχικοποιηθεί ο πληθυσμός, στη συνέχεια ξεκινάει ο κυρίως βρόχος του Γενετικού Αλγορίθμου #2. Επιλέγονται τυχαία τόσοι γονείς, όσο το πλήθος των μελών του πληθυσμού. Σημειώνουμε ότι το πλήθος των ατόμων του πληθυσμού διατηρείται σταθερό σε κάθε γενιά, όπως συμβαίνει και με τον Αλγόριθμο #1. Οι συναρτήσεις που χρησιμοποιούνται για την επιλογή γονέων (`begin_tournamnet()`) και την επιλογή των ζευγαριών προς διασταύρωση (`create_pool()`), είναι ίδιες με αυτές του Αλγορίθμου #1. Έχουμε επιλέξει ώστε από κάθε ζευγάρι να προκύπτει μόνο ένα παιδί²². Μετά την ολοκλήρωση της διαδικασίας της διασταύρωσης ο Αλγόριθμος #2 δεν χρειάζεται να ελέγξει αν το παιδί αντιστοιχεί σε συνδεδεμένο γράφο αφού ο αρχικός γράφος από τον οποίο προέρχεται ήταν ήδη συνδεδεμένος. Επίσης όπως έχουμε ήδη εξηγήσει²³ ο τρόπος με τον οποίο γίνεται η διασταύρωση, εγγυάται ότι τα παιδιά είναι και αυτά έγκυροι γράφοι ορατότητας.

Στη συνέχεια, εφόσον ικανοποιείται η συνθήκη όπου ένας τυχαία παραγόμενος αριθμός είναι μεγαλύτερος από τη συχνότητα μετάλλαξης, ξεκινά η διαδικασία της μετάλλαξης. Κατά την υλοποίηση του Αλγορίθμου #2 σχεδιάσαμε μόνο ένα είδος μετάλλαξης (συνάρτηση `cluster_swap2()`) η οποία προέρχεται και αυτή από τον Αλγόριθμο #1. Η συνάρτηση αυτή προχωρεί σε συσταδοποίηση των εμποδίων και υπολογίζει ένα σύνολο κόμβων (`best_nodes`) οι οποίοι είναι περισσότερο πιθανοί να αποτελούν τμήμα της συντομότερης διαδρομής. Η διαφορά με την αντίστοιχη συνάρτηση του αλγορίθμου #1 είναι ότι τώρα αντί να προστίθεται ένας κόμβος, προστίθεται στο γράφο μια ακμή με άκρα δύο τυχαίους κόμβους από το σύνολο `best_nodes`, υπό την προϋπόθεση ότι η ακμή αυτή συνδέει δύο διαφορετικές συστάδες. Με αυτό τον τρόπο αυξάνουμε την πιθανότητα η μεταλλαγμένη ακμή να αποτελεί τμήμα του συντομότερου μονοπατιού.

²¹ Ως τιμή αντικειμενικής συνάρτησης ορίζουμε το μήκος του μονοπατιού του αλγορίθμου Dijkstra.

²² Όπως και στον Γενετικό Αλγόριθμο #1.

²³ Βλέπε ενότητα 6.5.3

Ο Γενετικός αλγόριθμος #2 όπως είδαμε στο προηγούμενο κεφάλαιο, έχει καλύτερη επίδοση από την απλοϊκή μέθοδο κατασκευής γράφου ορατότητας. Όμως η επίδοσή του είναι χειρότερη από την επίδοση του αλγόριθμου Lee. Επιπλέον, σε κανένα από τα επτά σενάρια δεν καταφέρνει να υπολογίσει τη βέλτιστη λύση, αλλά σε πολλές περιπτώσεις την προσεγγίζει σε ικανοποιητικό βαθμό.

Όπως είδαμε στο σενάριο 2.7 (232 nodes) αν και υπολογίζει μόνο 346 ακμές για να κατασκευάσει το γράφο ορατότητας (έναντι 3983 ακμών του πλήρους γράφου), δηλαδή ενώ υπολογίζει:

$$\frac{3983 - 346}{3983} \times 100 = 91.3\%$$

λιγότερες ακμές, η επίδοσή του Αλγορίθμου #2 (155.8 sec) είναι αυξημένη σε σχέση με την επίδοση του αλγορίθμου Lee (4.2 sec).

8.3 Μελλοντικές κατευθύνσεις-βελτιώσεις Γενετικού Αλγορίθμου #1

Στη συνέχεια ακολουθούν κάποια σημεία τα οποία θα μπορούσαν να βελτιώσουν τον Γενετικό Αλγόριθμο #1:

1. Τροποποίηση κατά το στάδιο της διασταύρωσης ώστε να προκύπτουν περισσότερα του ενός παιδιά. Με αυτό τον τρόπο περισσότερα καινούρια άτομα θα εισάγονται μέσα στον πληθυσμό σε κάθε γενιά μειώνοντας πιο γρήγορα τον μέσο όρο των τιμών της αντικειμενικής συνάρτησης με αντίστοιχο όμως κόστος ελέγχου εγκυρότητας των νέων μονοπατιών.
2. Τροποποίηση αντικειμενικής συνάρτησης. Στον αλγόριθμο που παρουσιάσαμε ορίσαμε ως κόστος κάθε ακμής του γράφου το μήκος του. Έπειτα ορίσαμε ως τιμή της αντικειμενικής συνάρτησης το άθροισμα των μηκών των ακμών του μονοπατιού. Μια εναλλακτική προσέγγιση θα ήταν να προσθέσουμε στην τιμή αυτή εκτός από το άθροισμα των μηκών και το άθροισμα των κλίσεων των ευθυγράμμων τμημάτων που ορίζουν δύο κόμβοι. Όσο μικρότερο είναι αυτό το άθροισμα τόσο λιγότερα ζιγκ-ζαγκ θα έχει το τελικό μονοπάτι. Επίσης όσο μικρότερο είναι το άθροισμα αυτό τόσο μικρότερη θα είναι και η τιμή της αντικειμενικής συνάρτησης την οποία θέλουμε να ελαχιστοποιήσουμε.
3. Διαφορετικά είδη διασταύρωσης. Στον Γενετικό Αλγόριθμο #1 χρησιμοποιήσαμε διασταύρωση ενός σημείου (one-point crossover) σύμφωνα με την οποία το χρωμόσωμα σπάει σε δύο τμήματα. Χρήση της διασταύρωσης πολλών σημείων

(*n*-point crossover) σύμφωνα με την οποία το χρωμόσωμα χωρίζεται σε περισσότερα των δύο τμημάτων ή χρήση της ομοιόμορφης διασταύρωσης (uniform crossover) σύμφωνα με την οποία κάθε γονίδιο αντιμετωπίζεται ξεχωριστά (είτε κληρονομείται από τον πρώτο είτε από τον δεύτερο γονέα), θα μπορούσαν να καταλήξουν σε καλύτερα αποτελέσματα.

4. Διαφορετική πολιτική αντικατάστασης πληθυσμού. Στον Γενετικό Αλγόριθμο #1 μετά το στάδιο της μετάλλαξης και εφόσον οι απόγονοι (μονοπάτια) αναπαριστούσαν έγκυρες λύσεις τότε αφαιρούσαμε τυχαία από τον πληθυσμό τόσα παλιά άτομα όσοι και οι απόγονοι χωρίς να εφαρμόζουμε κάποια σύγκριση μεταξύ τους. Μια διαφορετική μέθοδος αντικατάστασης του πληθυσμού είναι η αντικατάσταση βάση παλαιότητας (age-based replacement). Σύμφωνα με αυτή ο αριθμός των απογόνων σε κάθε γενιά είναι ίσος με τον αριθμό των γονιών οπότε σε κάθε επανάληψη ολόκληρος ο πληθυσμός αντικαθίσταται από νέα άτομα. Συνέπεια των παραπάνω είναι ότι σε περίπτωση που η καλύτερη λύση βρίσκεται ήδη μέσα στον πληθυσμό, με την τεχνική αυτή η βέλτιστη λύση θα αντικατασταθεί από μια άλλη. Άρα στο διάγραμμα *Mean Fitness Value/Generations* θα παρατηρούνται αυξομειώσεις. Μια ακόμη μέθοδος αντικατάστασης πληθυσμού είναι αυτή στην οποία επιλέγουμε τόσα άτομα για αντικατάσταση όσος και ο αριθμός των απογόνων με την προϋπόθεση ότι τα άτομα που θα απορριφθούν έχουν τη χειρότερη τιμή αντικειμενικής συνάρτησης. Αν και αυτή η μέθοδος μπορεί να οδηγήσει σε ραγδαία πτώση του μέσου όρου της τιμής της αντικειμενικής συνάρτησης, ενδέχεται να καταλήξει σε πρόωρη λύση χωρίς να προλάβει να εξετάσει όλο τον χώρο των πιθανών λύσεων. Επίσης θα μπορούσε να χρησιμοποιηθεί η μέθοδος κατά την οποία σε κάθε γενιά εντοπίζεται το άτομο με την καλύτερη τιμή αντικειμενικής συνάρτησης (fittest member) και συγκρίνεται με την αντικειμενική συνάρτηση των απογόνων. Αν κανένας από τους απογόνους δεν έχει μεγαλύτερη τιμή τότε το συγκεκριμένο άτομο διατηρείται στον πληθυσμό.
5. Διαφορετική συνθήκη τερματισμού του Γενετικού Αλγορίθμου #1. Στον αλγόριθμο που κατασκευάσαμε ο αριθμός των γενεών επιλέγεται από το χρήστη κατά την παραμετροποίηση. Άλλες συνθήκες τερματισμού θα μπορούσαν να είναι:
 - a. Σταθεροποίηση μέσου όρου της τιμής της αντικειμενικής συνάρτησης για συγκεκριμένο αριθμό γενιών (π.χ., 10 γενιές). Δεν είναι όμως λίγες οι περιπτώσεις όπου ενώ ο μέσος όρος της αντικειμενικής συνάρτησης

παραμένει σταθερός για μεγάλο χρονικό διάστημα (π.χ., στο Σενάριο 1.10 παρατηρήσαμε ότι μέχρι την 100^η γενιά ο μέσος όρος δεν είχε μεταβληθεί) στη συνέχεια λόγω των γενετικών διαδικασιών ο αλγόριθμος ενδέχεται να βρει τη βέλτιστη λύση.

b. Όταν ο γενετικός αλγόριθμος πλησιάσει κοντά στην περιοχή της καλύτερης λύσης, τότε στο τρέχον καλύτερο άτομο του πληθυσμού εφαρμόζεται ο αλγόριθμος Hill Climbing (Veach, M.A., 1996).

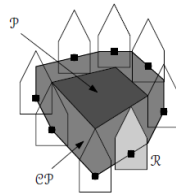
6. Νέα είδη μετάλλαξης θα μπορούσαν να βελτιώσουν το χρόνο εκτέλεσης του αλγορίθμου ώστε αυτός να συγκλίνει πιο γρήγορα σε λύση .
7. Διαφορετικός τρόπος υπολογισμού του γράφου ορατότητας. Στην υλοποίηση που παρουσιάσαμε καλούμε τη συνάρτηση `check_crosses()` τόσες φορές όσες είναι τα πολυγωνικά εμπόδια και στη συνέχεια με τη βοήθεια του πίνακα `non_zero[]` ο αλγόριθμος εντοπίζει ποιες ακμές διέρχονται διαμέσου των εμποδίων. Μια βελτιωμένη τεχνική στον τρόπο εύρεσης των έγκυρων ακμών θα συντόμευε το χρόνο υπολογισμού του γράφου ορατότητας.
8. Διαφορετική κωδικοποίηση των ατόμων του πληθυσμού π.χ., ως υπογράφοι (subgraphs) ενδεχομένως να προσέφερε πιο γρήγορη επεξεργασία αποτελεσμάτων.
9. Χρήση του πακέτου `multiprocessing` της Python ώστε να πραγματοποιείται παράλληλη επεξεργασία των δεδομένων.
10. Ο αλγόριθμος μπορεί να επεκταθεί ώστε να λειτουργεί σε δυναμικά περιβάλλοντα αφού είναι έτσι σχεδιασμένος ώστε να δέχεται ως είσοδο σημεία οι συντεταγμένες των οποίων δεν είναι γνωστές εκ των προτέρων. Επομένως όταν οι συντεταγμένες των εμποδίων μεταβάλλονται ανά τακτά χρονικά διαστήματα, ο γενετικός αλγόριθμος μπορεί να καλείται ισάριθμες φορές ώστε να υπολογίζει τη βέλτιστη διαδρομή.
11. Τροποποίηση της μεθόδου αρχικοποίησης του πληθυσμού. Στην υλοποίηση μας λάβαμε υπόψη μόνο τους γείτονες του κόμβου εκκίνησης. Υπολογίζοντας από το γράφο ορατότητας και τους γείτονες του κόμβου προορισμού μπορούμε να τοποθετούμε στον αρχικό πληθυσμό μονοπάτια τέτοια ώστε η πρώτη ακμή να συνδέει τον κόμβο εκκίνησης με κάποιον γείτονα του και η τελευταία ακμή να συνδέει τον κόμβο προορισμού με κάποιον γείτονα του. Με αυτό τον τρόπο διευκολύνουμε ακόμη περισσότερο τον αλγόριθμο ώστε να καταλήξει στη

βέλτιστη λύση με αντίστοιχη όμως χρονική επιβάρυνση αφού κατά το στάδιο της διασταύρωσης θα πρέπει τώρα να γίνονται επιπλέον έλεγχοι .

12. Εύρεση βέλτιστου μεγέθους πληθυσμού σε κάθε περίπτωση. Όπως αναφέραμε όταν ο αριθμός των γειτόνων του αρχικού κόμβου είναι μικρός, τότε ο αλγόριθμος λαμβάνει μέτρα και τριπλασιάζει τον αρχικό πληθυσμό ώστε να δημιουργείται κάθε φορά ικανός αριθμός ζευγαριών. Όμως όταν το πλήθος των γειτόνων είναι μεγάλο, ο πληθυσμός αυξάνεται σε μεγάλο βαθμό (βλέπε Σενάριο 1.12). Αυτό επιβαρύνει το χρόνο εκτέλεσης αφού πραγματοποιούνται άσκοπες διασταυρώσεις μεταξύ των μελών οι οποίες δεν συμβάλλουν στη μείωση του μέσου όρου της αντικειμενικής συνάρτησης. Άρα με μια κατάλληλη τροποποίηση, όταν ο αλγόριθμος εντοπίσει ότι ο κόμβος εκκίνησης έχει πολλούς γείτονες, θα μπορούσε να μειώνει το πλήθος τους στο κατάλληλο μέγεθος ώστε να συγκλίνει γρήγορα σε λύση.
13. Τροποποίηση της μεθόδου επιλογής γονέων που συμμετέχουν στο στάδιο της διασταύρωσης. Στον αλγόριθμο που παρουσιάσαμε σε κάθε επανάληψη διεξάγουμε ένα τουρνουά μεταξύ των ατόμων του πληθυσμού, συγκρίνοντάς τους ανά δύο, και επιλέγουμε το άτομο με την καλύτερη τιμή αντικειμενικής συνάρτησης. Με αυτό τον τρόπο όμως τα άτομα με τη μεγαλύτερη τιμή θα επιλέγονται συνεχώς μη δίνοντας την ευκαιρία στα υπόλοιπα άτομα του πληθυσμού να συμμετάσχουν στη διαδικασία της διασταύρωσης ώστε να προσφέρουν ποικιλία κατά τον ανασυνδυασμό των γονιδίων. Η επιλογή κατάταξης (ranking selection) αντιμετωπίζει ακριβώς αυτό το πρόβλημα. Σύμφωνα με τη μέθοδο αυτή στα άτομα του πληθυσμού αποδίδεται ένας βαθμός από το 1 έως το N , όπου N το μέγεθος του πληθυσμού. Έτσι το άτομο με τη χειρότερη τιμή αντικειμενικής συνάρτησης έχει βαθμό ίσο με 1 ενώ το άτομο με την καλύτερη τιμή έχει βαθμό ίσο με N . Με αυτό τον τρόπο όλα τα άτομα του πληθυσμού έχουν πιθανότητα να επιλεγούν ως γονείς.
14. Υλοποίηση του αλγορίθμου ώστε να εφαρμόζεται σε μια περιοχή με πολλά εμπόδια. Προσπαθήσαμε να κατασκευάσουμε τον αλγόριθμο με τέτοιο τρόπο ώστε να είναι εφικτή η απεικόνιση των εμποδίων που βρίσκονται στο χώρο διαμόρφωσης αλλά και ο χρόνος εκτέλεσης (t) του αλγορίθμου να βρίσκεται σε ανεκτά όρια π.χ., $1 \leq t \leq 120 \text{ sec}$. Όταν όμως ο χώρος αποτελείται από περισσότερα εμπόδια (π.χ., 100 εξάγωνα) τότε ο πίνακας γειτνίασης έχει

διαστάσεις 600×600 άρα ο χρόνος εκτέλεσης αυξάνεται σημαντικά. Τότε διάφορες τεχνικές μείωσης των διαστάσεων μπορούν να εφαρμοστούν όπως η Ανάλυση Κυρίων Συνιστωσών (*Principal Component Analysis-PCA*) αλλά και η Διάσπαση Μοναδιαίων Συνιστωσών (*Singular Value Decomposition-SVD*).

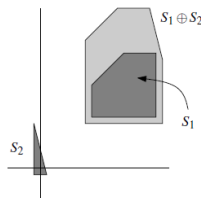
15. Στον Γενετικό Αλγόριθμο #1 υποθέσαμε ότι η κίνηση διαμέσου πολυγωνικών εμποδίων (P) πραγματοποιείται από ένα σημειακό ρομπότ (R). Αν όμως το ρομπότ δεν είναι πλέον σημειακό τότε ο ελεύθερος χώρος μέσα στον οποίο μπορεί να κινηθεί περιορίζεται. Όπως αναφέραμε στην ενότητα 2.1 το σύνολο των σημείων για τα οποία η μετατόπιση $R(x,y)$ του ρομπότ τέμνει ένα εμπόδιο ονομάζεται χώρος διαμόρφωσης (CP). Μπορούμε να φανταστούμε το σχήμα του χώρου CP αν μετακινήσουμε το ρομπότ R κατά μήκος του συνόρου του P , όπως φαίνεται στην παρακάτω εικόνα:



Σε αυτή την περίπτωση ο ελεύθερος χώρος μετακίνησης του ρομπότ υπολογίζεται μέσω των αθροισμάτων Minkowski. Το άθροισμα Minkowski δύο συνόλων $S_1 \subset R^2$ και $S_2 \subset R^2$ συμβολίζεται με $S_1 \oplus S_2$ και ορίζεται ως:

$$S_1 \oplus S_2 := \{p + q, p \in S_1, q \in S_2\}$$

όπου $p + q$ το διανυσματικό άθροισμα των διανυσμάτων p και q . Το άθροισμα Minkowski για δύο σύνολα σημείων S_1 και S_2 φαίνεται παρακάτω:

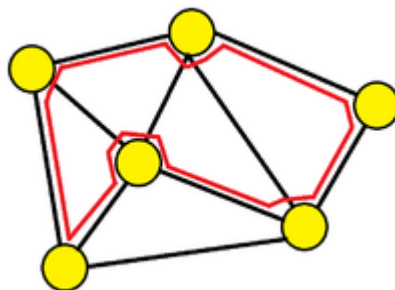


Εικόνα 175 Άθροισμα Minkowski (πηγή: Computational Geometry, 2008)

16. Απόδοση της χαμηλότερης (μηδενικής) τιμής στην αντικειμενική συνάρτηση των μη αποδεκτών λύσεων. Με αυτό τον τρόπο η συνάρτηση `max_weight()` καθίσταται περιττή οπότε μειώνουμε τον υπολογιστικό φόρτο και η εύρεση του συντομότερου

μονοπατιού μετατρέπεται σε πρόβλημα μεγιστοποίησης της αντικειμενικής συνάρτησης.

17. Εφαρμογή του Γενετικού Αλγορίθμου #1 σε προβλήματα δρομολόγησης στα δίκτυα αισθητήρων²⁴ (Tsanikidis, 2018).
18. Συνδυασμός Γενετικών Αλγορίθμων με DNA computing. Το 1994 ο Leonard Adleman στο άρθρο του με τίτλο *Molecular Computation of Solutions to Combinatorial Problems* προσπάθησε να υπολογίσει τον κύκλο Hamilton για ένα γράφο με επτά κόμβους χρησιμοποιώντας τεχνικές που εφαρμόζονται στη μοριακή βιολογία (πηγή: Wikipedia). Ο κύκλος Hamilton είναι το μονοπάτι που επισκέπτεται τους κόμβους ενός γράφου ακριβώς μια φορά. Ένας κύκλος Hamilton απεικονίζεται παρακάτω:



Εικόνα 176 Κύκλος Hamilton για γράφο με έξι κόμβους (πηγή: Wikipedia)

Στο άρθρο του ο Adleman κωδικοποίησε τους κόμβους και τις ακμές του γράφου χρησιμοποιώντας 20 βάσεις DNA. Για παράδειγμα αν ο κόμβος N_1 αναπαρασταθεί ως :

GCATGGCCAT AATCCAATTC

και ο κόμβος N_2 αναπαρασταθεί ως:

ATTACCGGAA CGATCGGCAT

τότε η ακμή N_1-N_2 αναπαρίσταται ως το συμπληρωματικό δεξί μέρος του κόμβου N_1 (TTAGGTTAAG) συν το συμπληρωματικό αριστερό μέρος του κόμβου N_2 (TAATGGCCTT) δηλαδή ως:

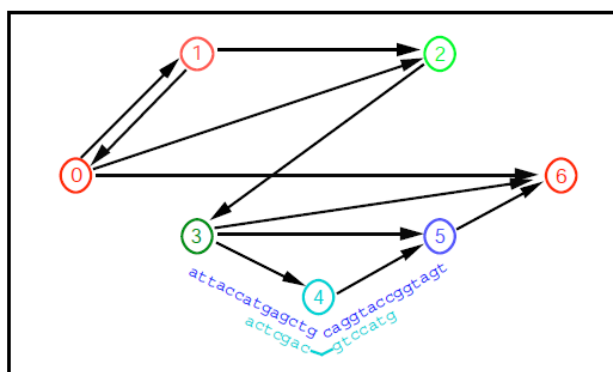
TTAGGTTAAG TAATGGCCTT

²⁴ Βλέπε Παράρτημα Γ

όπου A = Αδερίνη, T = Θυμίνη, G = Γουανίνη και C = Κυτοσίνη. Υπενθυμίζουμε ότι οι τέσσερις βάσεις σχηματίζουν διπλούς ή τριπλούς δεσμούς Υδρογόνου ως εξής:

$A-T$ και $C-G$

Στην παρακάτω εικόνα φαίνεται ένας κατευθυνόμενος γράφος όπου η κωδικοποίηση των ακμών 3-4 και 4-5 αναπαρίσταται με σκούρο μπλε χρώμα, ενώ η κωδικοποίηση του κόμβου 4 αναπαρίσταται με ανοιχτό μπλε:

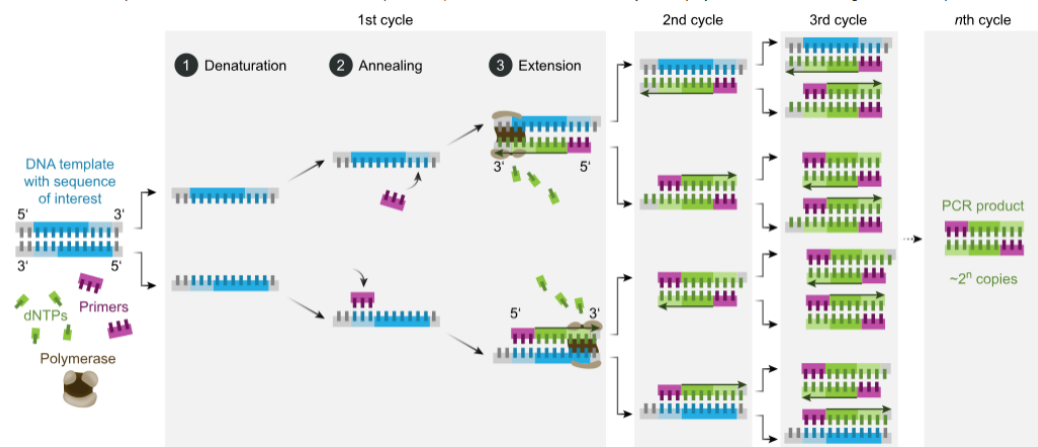


Εικόνα 177 Κύκλος Hamilton για κατευθυνόμενο γράφο με επτά κόμβους. Η μόνη διαδρομή η οποία επισκέπτεται όλους τους κόμβους ακριβώς μια φορά είναι η $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$. Παρατηρούμε επίσης τη συμπληρωματικότητα μεταξύ των βάσεων που αναπαριστούν τον κόμβο 4 και εκείνων που αναπαριστούν τις ακμές 3-4 και 4-5 (πηγή: Molecular computing: Does DNA compute?, 1996)

Ο Adleman ακολούθησε τα τρία παρακάτω βήματα για να υπολογίσει τον κύκλο Hamilton ενός γράφου (DNA computing based RNA genetic algorithm with applications in parameter estimation of chemical engineering processes, 2007):

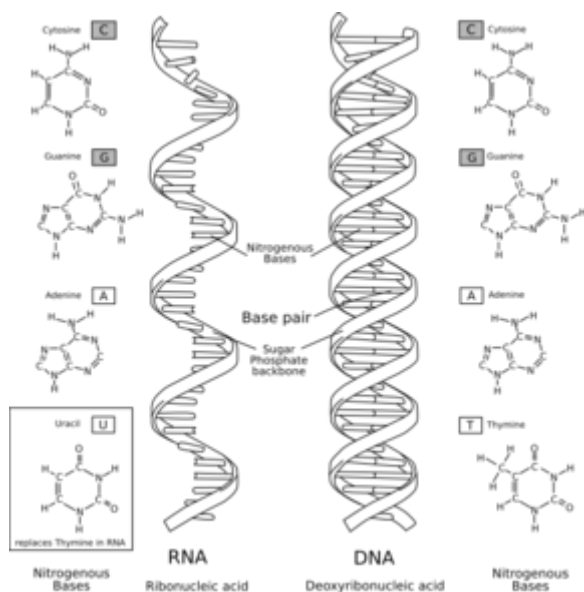
- κατασκευή μιας δεξαμενής με μόρια DNA τα οποία αναπαριστούν όλες τις πιθανές λύσεις,
- χρήση τεχνικών μοριακής βιολογίας ώστε να απομακρυνθούν οι λύσεις οι οποίες δεν είναι σύμφωνες με τους περιορισμούς του προβλήματος,
- επιλογή των μορίων DNA που απέμειναν μέσα στη δεξαμενή.

Μια από αυτές τις τεχνικές είναι και η Αλυσιδωτή Αντίδραση Πολυμεράσης (PCR-Polymerase Chain Reaction) η οποία χρησιμοποιείται για την απομόνωση και τον πολλαπλασιασμό μιας αλληλουχίας DNA (πηγή: Wikipedia). Στην παρακάτω εικόνα παρουσιάζεται η τεχνική PCR κατά την οποία χρησιμοποιώντας κατάλληλους εκκινητές (primers) καταφέρνουμε να πολλαπλασιάσουμε τα μόρια DNA:



Εικόνα 178 Τεχνική PCR (πηγή: Wikipedia)

Επειδή κατά τις γενετικές διαδικασίες η δομή διπλής έλικας του μορίου DNA δεν είναι κατάλληλη για να χρησιμοποιηθεί ως χρωμόσωμα, οι συγγραφείς του παραπάνω άρθρου προτείνουν ως κατάλληλη δομή αυτή του μονόκλωνου μορίου RNA. Η δομή RNA χρησιμοποιεί αντί για τη βάση Θυμίνη (T) την Ουρακίλη (U), όπως φαίνεται και στην παρακάτω εικόνα:



Εικόνα 179 Το μονόκλωνο RNA αριστερά και το δίκλωνο DNA στα δεξιά (πηγή: Wikipedia)

Κωδικοποιώντας λοιπόν τις πιθανές λύσεις ως μόρια RNA μπορούμε να εφαρμόσουμε σε αυτά τρεις βασικούς τελεστές:

- 1) Μετατόπιση: αν η αρχική ακολουθία RNA είναι $R=R_5R_4R_3R_2R_1$ (όπου R_i αποτελείται από τέσσερις βάσεις) τότε μετά τη μετατόπιση η νέα ακολουθία γίνεται: $R'=R_5R_2R_4R_3R_1$.
- 2) Μετατροπή: πραγματοποιείται διαμέσου ανταλλαγής τμημάτων μέσα σε μια ακολουθία RNA, για παράδειγμα η νέα ακολουθία μετά την εφαρμογή του τελεστή μετατόπισης είναι: $R'=R_5R_2R_3R_4R_1$.
- 3) Μετάθεση: πραγματοποιείται με αντικατάσταση $\pi\chi$ της υπο-ακολουθίας R_2 από μια άλλη υπο-ακολουθία R'_2 η οποία προέρχεται από την ίδια ή κάποια άλλη ακολουθία RNA και έτσι προκύπτει η νέα ακολουθία $R'=R_5R_4R_3R'_2R_1$.

Τα οφέλη από το συνδυασμό Γενετικών αλγορίθμων με τις τεχνικές που εφαρμόζονται κατά τη μέθοδο DNA/RNA computing είναι:

- ο η αποφυγή των χρονοβόρων διαδικασιών της διασταύρωσης και της μετάλλαξης,
- ο η αποφυγή να μετατραπεί ο γενετικός αλγόριθμος σε έναν αλγόριθμο τυχαίας αναζήτησης όταν η πιθανότητα μετάλλαξης είναι μεγάλη,
- ο η αποφυγή σύγκλισης του γενετικού αλγορίθμου σε τοπικό ελάχιστο όταν η πιθανότητα μετάλλαξης είναι μικρή.

8.4 Μελλοντικές κατευθύνσεις-βελτιώσεις Γενετικού Αλγορίθμου #2

Στη συνέχεια ακολουθούν κάποια σημεία τα οποία θα μπορούσαν να βελτιώσουν τον Γενετικό Αλγόριθμο #2:

1. Κωδικοποίηση των μελών του πληθυσμού ως άνω τριγωνικοί πίνακες. Μπορούμε να εκμεταλλευτούμε το γεγονός ότι οι πίνακες γειτνίασης είναι συμμετρικοί ως προς την κύρια διαγώνιο. Έτσι κερδίζουμε χρόνο αφού κατά τη διάρκεια κάποιου ελέγχου, διατρέχουμε μόνο τον μισό πίνακα.
2. Νέα είδη μετάλλαξης. Τροποποίηση της ενσωματωμένης συνάρτησης `double-edge_swap()` της βιβλιοθήκης `Networkx`, έτσι ώστε η εναλλαγή των ακμών να οδηγεί σε έγκυρες ακμές, δηλαδή ακμές που να μην τέμνουν τα εμπόδια.
3. Περισσότεροι από ένας απόγονοι. Όπως και στον Αλγόριθμο #1 έτσι και στον Αλγόριθμο #2 μόνο ένας απόγονος δημιουργείται κατά το στάδιο της

διασταύρωσης. Μεγαλύτερος αριθμός παιδιών ίσως επιταχύνει τον χρόνο σύγκλισης του αλγορίθμου.

4. Τροποποίηση της μεθόδου αρχικοποίησης του πληθυσμού. Το μειονέκτημα κατά την αρχικοποίηση του πληθυσμού στον Γενετικό Αλγόριθμο #2 είναι ότι η απαίτηση τα μέλη του αρχικού πληθυσμού να είναι όχι μόνο συνδεδεμένοι αλλά και έγκυροι γράφοι ορατότητας, αυξάνει το πλήθος των ελέγχων και συνεπώς αυξάνει το χρόνο εκτέλεσης του αλγορίθμου. Η αρχικοποίηση πληθυσμού με τυχαίους γράφους, οι οποίοι μπορεί να είναι είτε ασύνδετοι είτε μη έγκυροι, μπορεί να βελτιώνει το χρόνο εκτέλεσης του γενετικού αλγορίθμου. Επίσης, στον αλγόριθμο που κατασκευάσαμε κάθε γράφος διαφέρει ως προς μια ακμή σε σχέση με το προηγούμενο μέλος ή δεν διαφέρει καθόλου. Μια πιθανή βελτίωση θα μπορούσε να είναι ότι η διαφορά ακμών από το ένα μέλος στο άλλο να καθορίζεται με τυχαίο τρόπο.
5. Εύρεση βέλτιστου μεγέθους πληθυσμού. Όπως είδαμε και στο Κεφάλαιο 7, όταν το πλήθος των κόμβων είναι μικρό, τότε ο πληθυσμός που δημιουργείται είναι μεγάλος, ενώ όσο το πλήθος των κόμβων αυξάνεται, το μέγεθος του πληθυσμού ελαττώνεται. Συγκεκριμένα, για τα επτά σενάρια του Κεφαλαίου 7 έχουμε:

Κόμβοι	Πληθυσμός
77	29
92	35
102	7
127	9
147	9
202	13
232	7

Το αυξημένο μέγεθος πληθυσμού στα δύο πρώτα σενάρια έχει ως συνέπεια αυξημένο χρόνο εκτέλεσης, κάτι που απεικονίζεται και στο διάγραμμα της Εικόνας 174. Από την άλλη, όσο περισσότερα είναι τα εμπόδια, τόσο λιγότερες ακμές εισάγονται στον αρχικό γράφο κατά την αρχικοποίηση οπότε δημιουργείται ένας μικρός σε μέγεθος πληθυσμός. Επομένως, θα μπορούσαν να παρθούν τα κατάλληλα μέτρα ώστε το μέγεθος του πληθυσμού να ρυθμίζεται αυτόματα είτε σε σχέση με το πλήθος των κόμβων είτε σε σχέση με το πλήθος των εμποδίων.

6. Διασταύρωση περισσότερων των δύο γονέων. Με αυτό τον τρόπο και χρησιμοποιώντας την ενσωματωμένη συνάρτηση `np.logical_or()` του πακέτου

Numpy, δημιουργούνται παιδιά με περισσότερες ακμές, άρα αυξάνεται η πιθανότητα να βρεθεί μια λύση που να προσεγγίζει όσο το δυνατόν περισσότερο τη βέλτιστη.

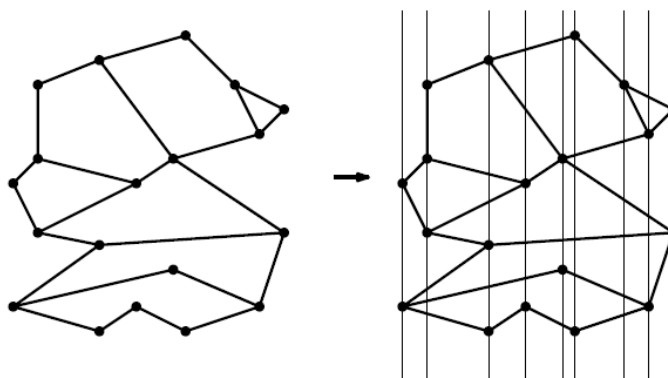
7. Διαφορετικός τρόπος υπολογισμού του μέσου όρου των τιμών της αντικειμενικής συνάρτησης των μελών κάθε γενιάς. Υλοποιήσαμε με τέτοιο τρόπο τον Γενετικό Αλγόριθμο #2 ώστε μετά το τέλος των γενετικών διαδικασιών ο αλγόριθμος να επαναυπολογίζει τις τιμές της αντικειμενικής συνάρτησης όλων των μελών του πληθυσμού. Μια βελτίωση θα μπορούσε να είναι ο υπολογισμός να γίνεται μόνο για τα νέα μέλη που εισήχθησαν στον πληθυσμό. Αυτό θα επιτάχυνε την εκτέλεση του αλγορίθμου ειδικά σε περιπτώσεις όπου ο πληθυσμός αποτελείται από πολλά μέλη.

Παράρτημα Α: Τραπεζοειδής χάρτης $T(S)$

Σκοπός της ενότητας είναι να διευκολύνει τον αναγνώστη στην κατανόηση των βασικών εννοιών του τραπεζοειδούς χάρτη (όψεις, τραπέζια, κλπ.).

Ας υποθέσουμε ότι S είναι μια υποδιαίρεση του επιπέδου με n ακμές. Αναζητούμε να βρούμε την όψη του S στην οποία ανήκει ένα σημείο q . Θα πρέπει να κατασκευάσουμε μια δομή δεδομένων στην οποία να αποθηκεύσουμε το S ώστε να μπορούμε αποτελεσματικά να εντοπίζουμε ένα οποιοδήποτε σημείο.

Αν φέρουμε κάθετες από όλες τις κορυφές του επιπέδου S τότε αυτό θα χωριστεί σε κάθετους τομείς όπως φαίνεται στην παρακάτω εικόνα:

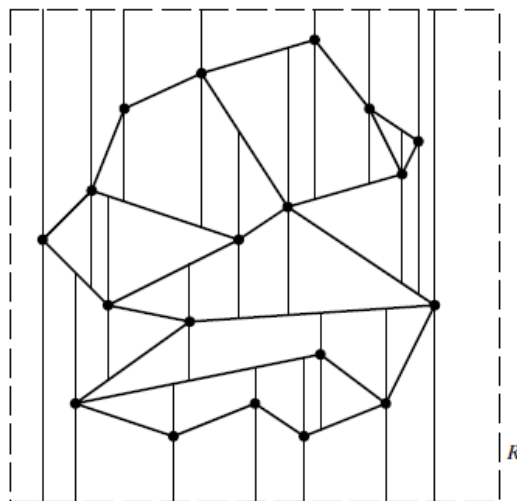


Εικόνα 180 Διαίρεση του S σε τομείς (πηγή: Computational Geometry, 2008)

Στη συνέχεια αποθηκεύουμε τις τετμημένες x των κορυφών σε έναν ταξινομημένο πίνακα. Με αυτό τον τρόπο μπορούμε σε χρόνο $O(\log n)$ να εντοπίσουμε τον τομέα που περιέχει το σημείο q . Μέσα σε κάθε τομέα δεν υπάρχουν κορυφές του S . Αυτό σημαίνει ότι όλες οι ακμές που τέμνουν έναν τομέα τον διαπερνούν πλήρως, δεν υπάρχουν άκρα των ακμών μέσα σε έναν τομέα και επίσης οι ακμές δεν τέμνονται. Επομένως μπορούμε να διατάξουμε τις ακμές από πάνω προς τα κάτω. Στη συνέχεια αποθηκεύουμε κάθε ακμή σε έναν ταξινομημένο πίνακα και την σημειώνουμε με το όνομα της όψης του S που βρίσκεται ακριβώς από πάνω. Έτσι λοιπόν όταν θέλουμε να εντοπίσουμε ένα σημείο q πρώτα εκτελούμε μια δυαδική αναζήτηση στον πίνακα με τις τετμημένες x , εντοπίζουμε τον τομέα που περιέχει το q και στη συνέχεια εκτελούμε πάλι μια δυαδική αναζήτηση στον πίνακα όπου έχουμε αποθηκεύσει τις ακμές κάθε τομέα. Από εκεί μπορούμε να εντοπίσουμε την όψη του S στην οποία ανήκει το σημείο q .

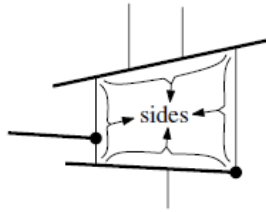
Αν και ο χρόνος αναζήτησης μιας όψης του S είναι $O(\log n)$, επειδή χρειαζόμαστε δύο πίνακες για την αποθήκευση των δεδομένων, έναν για τις τετμημένες των κορυφών και έναν πίνακα για κάθε τομέα, η λειτουργία αποθήκευσης απαιτεί χώρο ίσο με $O(n^2)$. Για αυτό το λόγο χρειαζόμαστε μια καλύτερη υποδιαίρεση του S . Υποθέτουμε ότι το S είναι ένα σύνολο από ακμές οι οποίες δεν τέμνονται. Έπειτα τοποθετούμε αυτό το σύνολο μέσα σε ένα ορθογώνιο παραλληλόγραμμο R . Με αυτό τον τρόπο απαλλασσόμαστε από τις μη οριοθετούμενες περιοχές. Στη συνέχεια κάνουμε μια ακόμη απλοποίηση. Υποθέτουμε ότι κανένα άκρο ευθυγράμμου τμήματος δεν έχει την ίδια τετμημένη. Επομένως κανένα άκρο δεν θα βρίσκεται στην ίδια κατακόρυφο με κάποιο άλλο.

Λέμε τότε ότι τα ευθύγραμμα τμήματα βρίσκονται σε γενική θέση. Για να σχεδιάσουμε τον τραπεζοειδή χάρτη $T(S)$ του S προεκτείνουμε τα άκρα κάθε τμήματος του S προς τα πάνω και προς τα κάτω. Οι προεκτάσεις σταματούν όταν συναντήσουν ένα άλλο τμήμα του S ή το όριο του R . Ο τραπεζοειδής χάρτης είναι η υποδιαίρεση του επιπέδου που προκύπτει από το S , το παραλληλόγραμμο R καθώς και τις κάθετες προεκτάσεις όπως φαίνεται στην παρακάτω εικόνα:



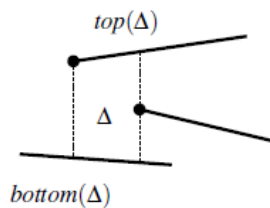
Εικόνα 181 Τραπεζοειδής χάρτης (πηγή: Computational Geometry, 2008)

Κάθε όψη στον τραπεζοειδή χάρτη περικλείεται από έναν αριθμό ακμών του $T(S)$. Κάποιες από αυτές τις ακμές μπορεί να είναι γειτονικές και συγγραμμικές, δηλαδή να βρίσκονται πάνω στην ίδια διεύθυνση. Την ένωση αυτών των ακμών την αποκαλούμε πλευρά της όψης. Με άλλα λόγια, πλευρά μιας όψης είναι τα τμήματα που περιέχονται στο όριο της όψης αυτής, όπως φαίνεται και στην παρακάτω εικόνα:



Εικόνα 182 Πλευρές μιας όψης ενός Τραπεζοειδούς χάρτη (πηγή: Computational Geometry, 2008)

Ισχύει ότι κάθε όψη ενός τραπεζοειδούς χάρτη ενός συνόλου ευθυγράμμων τμημάτων S τα οποία βρίσκονται σε γενική θέση, έχει μία ή δύο κάθετες πλευρές και ακριβώς δύο μη κάθετες πλευρές. Επομένως κάθε όψη ενός τραπεζοειδούς χάρτη είναι ή τραπέζιο ή τρίγωνο. Το ευθύγραμμο τμήμα το οποίο περικλείει το τραπέζιο Δ από το πάνω μέρος ονομάζεται $top(\Delta)$, ενώ το ευθύγραμμο τμήμα που περικλείει το τραπέζιο από το κάτω μέρος ονομάζεται $bottom(\Delta)$ όπως δείχνει η παρακάτω εικόνα:



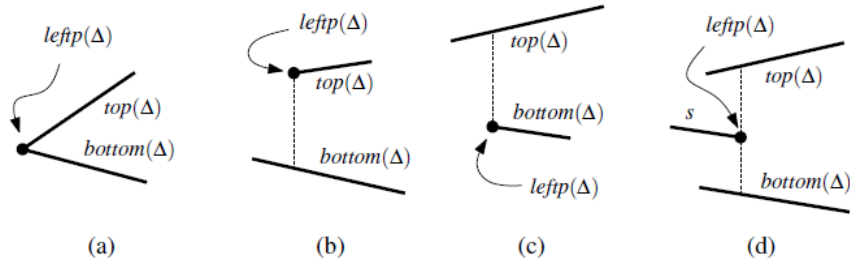
Εικόνα 183 Ονομασία ευθυγράμμων τμημάτων που περικλείουν ένα τραπέζιο Δ (πηγή: Computational Geometry, 2008)

Οι δύο κάθετες πλευρές ενός τραπέζιου Δ ανήκουν σε μία από τις πέντε παρακάτω περιπτώσεις (εδώ αναφερόμαστε μόνο στην αριστερή πλευρά):

- Εκφυλίζεται σε ένα μόνο σημείο το οποίο είναι το κοινό αριστερό άκρο των $bottom(\Delta)$ και $top(\Delta)$.
- Πρόκειται για την κατακόρυφη προέκταση του αριστερού άκρου του $top(\Delta)$ η οποία καταλήγει στο $bottom(\Delta)$.
- Πρόκειται για την κατακόρυφη προέκταση του αριστερού άκρου του $bottom(\Delta)$ το οποίο καταλήγει στο $top(\Delta)$.
- Πρόκειται για τις προς τα πάνω και προς τα κάτω προεκτάσεις του δεξιού άκρου ενός άλλου ευθυγράμμου τμήματος s οι οποίες καταλήγουν στο $top(\Delta)$ και $bottom(\Delta)$ αντίστοιχα.

ε) Πρόκειται για την αριστερή πλευρά του παραλληλογράμμου R .

Οι τέσσερις από τις πέντε παραπάνω περιπτώσεις φαίνονται στην εικόνα που ακολουθεί:

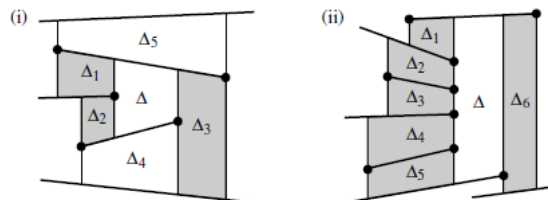


Εικόνα 184 Τέσσερις από τις πέντε περιπτώσεις για την αριστερή πλευρά του τραπέζιου Δ (πηγή: Computational Geometry, 2008)

Συμβολίζουμε με $leftp(\Delta)$ το αριστερό άκρο που ορίζει την αριστερή πλευρά του Δ και με $rightp(\Delta)$ το δεξιό άκρο που ορίζει τη δεξιά πλευρά του Δ . Ως εκ τούτου ένα τραπέζιο ορίζεται μοναδικά από τα τμήματα $top(\Delta)$, $bottom(\Delta)$ και τα άκρα $leftp(\Delta)$, $rightp(\Delta)$.

Ισχύει ότι ο τραπεζοειδής χάρτης $T(S)$ ενός συνόλου S το οποίο αποτελείται από n ευθύγραμμα τμήματα σε γενική θέση περιέχει το πολύ $6n + 4$ κορυφές και $3n + 1$ τραπέζια.

Καλούμε δυο τραπέζια Δ και Δ' γειτονικά αν έχουν μια κοινή κάθετη ακμή. Για παράδειγμα στην Εικόνα 185(i) το τραπέζιο Δ είναι γειτονικό με τα $\Delta_1, \Delta_2, \Delta_3$ αλλά όχι με τα Δ_4 και Δ_5 .



Εικόνα 185 Τα γειτονικά τραπέζια με το Δ είναι σκιασμένα με γκρι (πηγή: Computational Geometry, 2008)

Επειδή τα ευθύγραμμα τμήματα βρίσκονται σε γενική θέση ένα τραπέζιο έχει το πολύ τέσσερα γειτονικά τραπέζια αλλιώς μπορεί να έχει αυθαίρετο αριθμό γειτονικών τραπέζιων όπως φαίνεται στην Εικόνα 185(ii).

Για να αναπαραστήσουμε έναν τραπεζοειδή χάρτη εκμεταλλευόμαστε τη γειννίαση των τραπέζιων, ώστε να συνδέσουμε όλο το χάρτη. Αφού κάθε ευθύγραμμο τμήμα στο χάρτη θα είναι είτε $top(\Delta)$ είτε $bottom(\Delta)$ και κάθε σημείο θα είναι είτε $leftp(\Delta)$ είτε $rightp(\Delta)$ αποθηκεύουμε κάθε εγγραφή τραπέζιου Δ ως τέσσερις δείκτες στα τέσσερα παραπάνω στοιχεία. Αυτό σημαίνει ότι μπορούμε να εξάγουμε την πληροφορία για κάθε τραπέζιο σε σταθερό χρόνο $O(1)$.

Παράρτημα Β: Αλγόριθμος Trapezoidal Map (S)

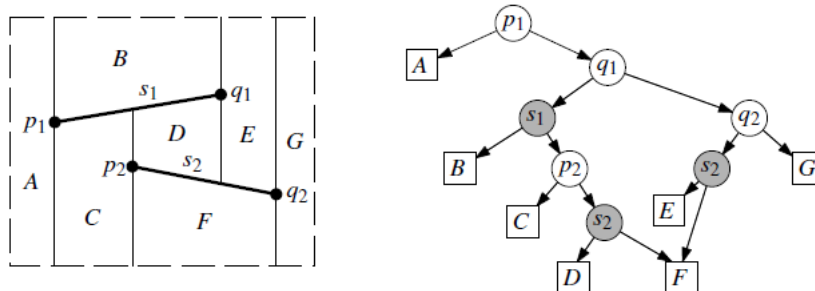
Σε αυτή την ενότητα θα παρουσιάσουμε τον αλγόριθμο κατασκευής του τραπεζοειδούς χάρτη $T(S)$ ενός συνόλου S το οποίο αποτελείται από n ευθύγραμμα τμήματα. Σημειώνουμε ότι ο αλγόριθμος Trapezoidal Map χρησιμοποιείται από τον αλγόριθμο Compute Path που παρουσιάσαμε στην υποενότητα 2.5.1.

Κατά τη διάρκεια της κατασκευής ο αλγόριθμος δημιουργεί μια δομή δεδομένων D την οποία μπορούμε να χρησιμοποιήσουμε για να αναζητήσουμε σημεία μέσα στο χάρτη $T(S)$. Η δομή δεδομένων D είναι ένας κατευθυνόμενος ακυκλικός γράφος του οποίου τα φύλλα αντιστοιχούν στα τραπέζια Δ του χάρτη. Οι εσωτερικοί κόμβοι του γράφου έχουν βαθμό 2, δηλαδή κάθε κόμβος ενώνεται με 2 ακμές. Υπάρχουν δυο είδη κόμβων, οι x -κόμβοι στους οποίους τοποθετούμε τα άκρα κάποιου τμήματος s και οι y -κόμβοι στους οποίους τοποθετούμε τα τμήματα s_i .

Η αναζήτηση ενός σημείου q ξεκινά από τη ρίζα και καταλήγει μέσω ενός μονοπατιού σε κάποιο φύλλο το οποίο αντιστοιχεί σε ένα τραπέζιο $\Delta \in T(S)$. Σε κάθε x -κόμβο ο αλγόριθμος εξετάζει αν το σημείο q βρίσκεται δεξιά ή αριστερά του άκρου το οποίο είναι αποθηκευμένο στον κόμβο και ανάλογα πηγαίνει στο δεξί ή στο αριστερό παιδί. Σε κάθε y -κόμβο ο αλγόριθμος ελέγχει αν το σημείο q βρίσκεται πάνω ή κάτω από το τμήμα το οποίο είναι αποθηκευμένο στον κόμβο αυτόν.

Η δομή δεδομένων D και ο χάρτης $T(S)$ συνδέονται ως εξής: κάθε τραπέζιο $\Delta \in T(S)$ έχει έναν δείκτη στο αντίστοιχο φύλλο της δομής D και κάθε φύλλο της δομής D έχει έναν

δείκτη στο αντίστοιχο τραπέζιο του χάρτη $T(S)$. Η Εικόνα 186 δείχνει τον χάρτη ενός συνόλου δύο τμημάτων s_1 και s_2 καθώς και την δομή δεδομένων D .



Εικόνα 186 Ο τραπεζοειδής χάρτης δύο τμημάτων και η δομή δεδομένων του (πηγή: Computational Geometry, 2008)

Παρατηρούμε ότι οι x -κόμβοι απεικονίζονται με άσπρο χρώμα και περιέχουν την ονομασία του άκρου του τμήματος στο οποίο ανήκουν, ενώ οι y -κόμβοι απεικονίζονται με γκρι χρώμα και περιέχουν την ονομασία του τμήματος. Τα φύλλα της δομής απεικονίζονται με τετράγωνο και περιέχουν την ονομασία του αντίστοιχου τραpezίου.

Ο αλγόριθμος κατασκευής του χάρτη $T(S)$ ονομάζεται αυξητικός, διότι προσθέτει στο χάρτη ένα-ένα τα τμήματα s_i και στη συνέχεια ανανεώνει τη δομή δεδομένων D . Η απόδοση το αλγορίθμου επηρεάζεται από τη σειρά με την οποία προσθέτονται τα τμήματα κάθε φορά για αυτό και άλλες φορές παρουσιάζει ικανοποιητικούς χρόνους αναζήτησης και άλλες όχι. Επίσης ο αλγόριθμος επιλέγει τυχαία κάθε φορά ποια τμήματα s_i θα προσθέσει στο χάρτη. Στην επόμενη εικόνα παραθέτουμε τον αλγόριθμο Trapezoidal Map:

Algorithm TRAPEZOIDALMAP(S)

Input. A set S of n non-crossing line segments.

Output. The trapezoidal map $\mathcal{T}(S)$ and a search structure \mathcal{D} for $\mathcal{T}(S)$ in a bounding box.

1. Determine a bounding box R that contains all segments of S , and initialize the trapezoidal map structure \mathcal{T} and search structure \mathcal{D} for it.
2. Compute a random permutation s_1, s_2, \dots, s_n of the elements of S .
3. **for** $i \leftarrow 1$ **to** n
4. **do** Find the set $\Delta_0, \Delta_1, \dots, \Delta_k$ of trapezoids in \mathcal{T} properly intersected by s_i .
5. Remove $\Delta_0, \Delta_1, \dots, \Delta_k$ from \mathcal{T} and replace them by the new trapezoids that appear because of the insertion of s_i .
6. Remove the leaves for $\Delta_0, \Delta_1, \dots, \Delta_k$ from \mathcal{D} , and create leaves for the new trapezoids. Link the new leaves to the existing inner nodes by adding some new inner nodes, as explained below.

Εικόνα 187 Αλγόριθμος Trapezoidal Map (πηγή: Computational Geometry, 2008)

Ο αλγόριθμος αρχικοποιείται με το κενό σύνολο δηλαδή $T(S_0) = T(\emptyset)$ οπότε ο χάρτης του κενού συνόλου περιέχει μόνο ένα τραπέζιο, το οριοθετημένο παραλληλόγραμμο R και η δομή δεδομένων του $T(\emptyset)$ περιέχει μόνο ένα φύλλο. Στη συνέχεια με κάθε ευθύγραμμο τμήμα s_i που προστίθεται στο χάρτη ο αλγόριθμος εξετάζει ποια τραπέζια από αυτά που βρίσκονται ήδη μέσα σε αυτόν τέμνει. Εδώ θα πρέπει να παρατηρήσουμε ότι ένα τραπέζιο του χάρτη $T(S_{i-1})$ δεν είναι παρόν στο χάρτη $T(S_i)$ αν και μόνο αν το τμήμα s_i τέμνει το τραπέζιο αυτό. Για να βρει ο αλγόριθμος ποια τραπέζια $\Delta_0, \Delta_1, \dots, \Delta_k$ τέμνει κάθε τμήμα s_i που εισέρχεται στο χάρτη εξετάζει τα τέσσερα στοιχεία που αναφέραμε στο Παράρτημα Α δηλαδή τα άκρα $leftp(\Delta)$, $rightp(\Delta)$ και τα ευθύγραμμα τμήματα $top(\Delta)$, $bottom(\Delta)$. Για παράδειγμα αν το άκρο $rightp(\Delta_j)$ βρίσκεται πάνω από το τμήμα s_i τότε το τραπέζιο Δ_{j+1} είναι το κάτω δεξιά γειτονικό τραπέζιο του Δ_j , αλλιώς είναι το πάνω δεξιά γειτονικό του τραπέζιο. Εν συντομία, γνωρίζοντας το τραπέζιο Δ_0 μπορούμε να βρούμε και τα υπόλοιπα $\Delta_1, \Delta_2, \dots, \Delta_k$. Για να βρούμε το Δ_0 το οποίο περιέχει το αριστερό άκρο p του νέου τμήματος s_i εξετάζουμε αν το p περιέχεται στο προηγούμενο τμήμα s_{i-1} . Αν όχι τότε θα βρίσκεται εντός του τραπεζίου Δ_0 οπότε διεξάγουμε μια αναζήτηση του σημείου q μέσα στον χάρτη $T(S_{i-1})$. Σε αυτό μας βοηθάει η δομή δεδομένων D η οποία περιέχει όλα τα άκρα p και q των προηγούμενων ευθυγράμμων τμημάτων s . Συνοπτικά χρησιμοποιούμε τον παρακάτω αλγόριθμο για να βρούμε τα $\Delta_0, \dots, \Delta_k$:

Algorithm FOLLOWSEGMENT($\mathcal{T}, \mathcal{D}, s_i$)

Input. A trapezoidal map \mathcal{T} , a search structure \mathcal{D} for \mathcal{T} , and a new segment s_i .

Output. The sequence $\Delta_0, \dots, \Delta_k$ of trapezoids intersected by s_i .

1. Let p and q be the left and right endpoint of s_i .
2. Search with p in the search structure \mathcal{D} to find Δ_0 .
3. $j \leftarrow 0$;
4. **while** q lies to the right of $rightp(\Delta_j)$
5. **do if** $rightp(\Delta_j)$ lies above s_i
6. **then** Let Δ_{j+1} be the lower right neighbor of Δ_j .
7. **else** Let Δ_{j+1} be the upper right neighbor of Δ_j .
8. $j \leftarrow j + 1$
9. **return** $\Delta_0, \Delta_1, \dots, \Delta_j$

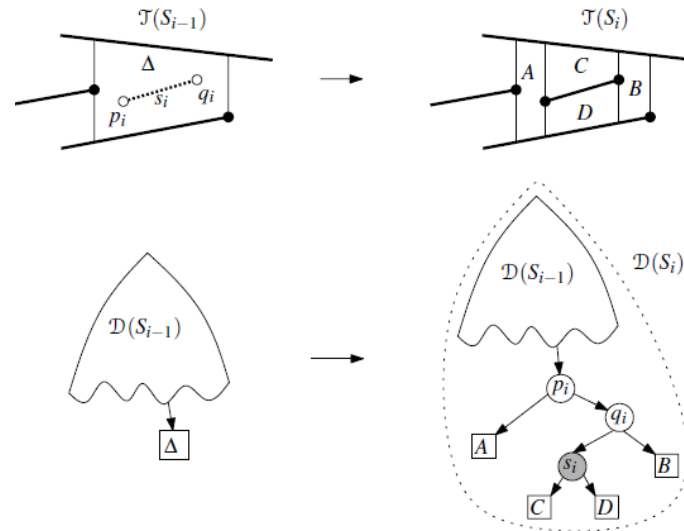
Εικόνα 188 Αλγόριθμος Follow Segment (πηγή: Computational Geometry, 2008)

Παρατηρούμε ότι ο αλγόριθμος εξετάζει τους δεξιούς γείτονες γιατί προχωρούμε από τα αριστερά προς τα δεξιά.

Αφού είδαμε πως βρίσκουμε τα τραπέζια τα οποία τέμνονται από το τμήμα s_i στη συνέχεια εξετάζουμε το πώς γίνεται η ανανέωση τόσο στο χάρτη T όσο και στη δομή δεδομένων D . Θα ασχοληθούμε με τη πιο απλή περίπτωση όπου το νέο τμήμα s_i βρίσκεται εξ ολοκλήρου μέσα στο τραπέζιο Δ .

Για να ενημερώσουμε το δέντρο T διαγράφουμε το τραπέζιο Δ γιατί όπως αναφέραμε προηγουμένως ένα τραπέζιο του χάρτη $T(S_{i-1})$ δεν είναι παρόν στο χάρτη $T(S_i)$, και το αντικαθιστούμε με τέσσερα νέα τραπέζια A, B, C, D διότι όπως γνωρίζουμε ένα τραπέζιο μπορεί να έχει το πολύ τέσσερις γείτονες. Όλες οι απαραίτητες πληροφορίες για να αρχικοποιήσουμε τις εγγραφές των τεσσάρων νέων τραpezίων ($leftp(\Delta)$, $rightp(\Delta)$, $top(\Delta)$, $bottom(\Delta)$) είναι διαθέσιμες από τις πληροφορίες οι οποίες είναι αποθηκευμένες για το τραπέζιο Δ . Επισημαίνουμε ότι όλες αυτές οι πληροφορίες είναι διαθέσιμες σε σταθερό χρόνο $O(1)$ αφού χρησιμοποιούμε δείκτες.

Για να ενημερώσουμε τη δομή δεδομένων D αντικαθιστούμε το φύλλο το οποίο αντιστοιχεί στο τραπέζιο Δ με ένα δέντρο με τέσσερα φύλλα. Το δέντρο διαθέτει δυο εσωτερικούς x -κόμβους στους οποίους αποθηκεύονται το δεξιό και αριστερό άκρο του τμήματος s_i και ένα y -κόμβο στον οποίο αποθηκεύεται το τμήμα s_i . Οι παραπάνω τροποποιήσεις στο δέντρο T και στη δομή D φαίνονται στην Εικόνα 189:



Εικόνα 189 Το νέο τμήμα s_i βρίσκεται εξ ολοκλήρου μέσα στο τραπέζιο Δ (πηγή: Computational Geometry, 2008)

Παρατηρούμε ότι το τραπέζιο A βρίσκεται αριστερά του σημείου p , το τραπέζιο B βρίσκεται δεξιά του σημείου q , το τραπέζιο C βρίσκεται πάνω από το τμήμα s_i και τέλος το τραπέζιο D βρίσκεται κάτω από το τμήμα s_i . Επίσης παρατηρούμε ότι όταν εισάγονται στο δέντρο 4 τραπέζια, εισάγονται ταυτόχρονα και $4-1=3$ εσωτερικοί κόμβοι.

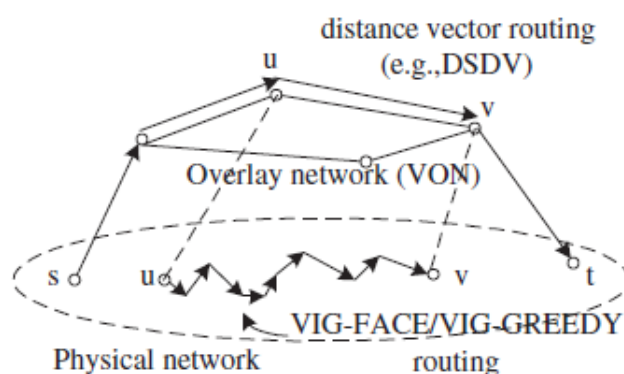
Όπως αναφέραμε και προηγουμένως η απόδοση του αλγορίθμου εξαρτάται από τη σειρά με την οποία εισάγονται τα τμήματα στο χάρτη T . Για παράδειγμα ας θεωρήσουμε ένα σύνολο S από n ευθύγραμμα τμήματα τα οποία δεν τέμνονται μεταξύ τους. Αφού υπάρχουν $n!$ δυνατές μεταθέσεις των n τμημάτων, θα υπάρχουν και $n!$ δυνατές διαδρομές που μπορεί να ακολουθήσει ο αλγόριθμος. Ο αναμενόμενος χρόνος εκτέλεσης του αλγορίθμου είναι η μέση τιμή του χρόνου εκτέλεσης καθεμιάς από τις $n!$ πιθανές διαδρομές που μπορεί να ακολουθήσει. Τελικά μπορεί να αποδειχθεί ότι ο αλγόριθμος Trapezoidal Map μπορεί να υπολογίζει τον τραπεζοειδή χάρτη $T(S)$ ενός συνόλου S το οποίο αποτελείται από n ευθύγραμμα τμήματα σε γενική θέση και μια δομή δεδομένων D σε χρόνο $O(n \log n)$. Επίσης το μέγεθος της δομής δεδομένων είναι $O(n)$ και τέλος μια αναζήτηση του σημείου q διαρκεί χρόνο $O(\log n)$.

Παράρτημα Γ: Γράφος Ορατότητας και Δίκτυα Αισθητήρων

Στα στατικά δίκτυα αισθητήρων οι αποφάσεις δρομολόγησης λαμβάνονται σύμφωνα με τη διαθέσιμη πληροφορία σε γειτονικούς κόμβους. Όμως σύμφωνα με παρατηρήσεις οι

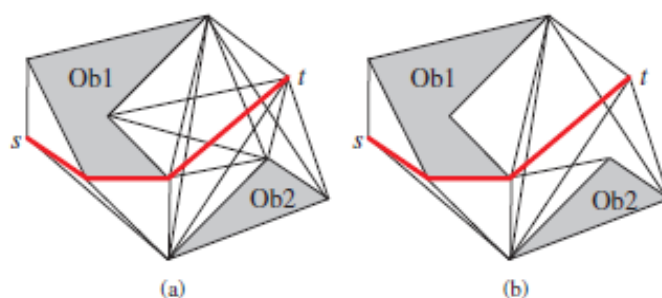
περισσότεροι αλγόριθμοι δρομολόγησης αδυνατούν να βρουν την καλύτερη λύση στο πρόβλημα υπολογισμού της πιο σύντομης διαδρομής. Πιο συγκεκριμένα έχει αποδειχτεί ότι αν η καλύτερη διαδρομή έχει μήκος c , τότε η λύση την οποία δίνουν οι περισσότεροι αλγόριθμοι έχει μήκος $\Omega(c^2)$.

Μια έρευνα που δίνει λύση στο παραπάνω πρόβλημα (Tan, Bertier, Kermarrec, 2009) προτείνει τον αλγόριθμο VIGOR (Visibility Graph-based Routing Protocol) η σχεδίαση του οποίου βασίζεται στην κατασκευή ενός Γράφου Ορατότητας. Σε πολλά ασύρματα δίκτυα αισθητήρων μέσα στο χώρο του δικτύου συναντάμε εμπόδια αυθαίρετου σχήματος και μεγέθους. Ο αλγόριθμος δρομολόγησης χρησιμοποιεί αρχικά την τεχνική greedy forwarding επιλέγοντας κάθε φορά τους κόμβους εκείνους οι οποίοι βρίσκονται σε πιο κοντινή απόσταση από τον κόμβο προορισμού. Όταν δεν βρεθεί τέτοιος κόμβος τότε οι περισσότεροι αλγόριθμοι ακολουθούν τη διαδρομή κατά μήκος της συνοριακής γραμμής του εμποδίου σύμφωνα με τη φορά των δεικτών του ρολογιού μη λαμβάνοντας υπόψη το γεγονός ότι μπορεί να υπάρχει πιο σύντομη διαδρομή από την αντίθετη φορά. Το πρωτόκολλο VIGOR κατασκευάζει ένα μικρό δίκτυο επικάλυψης (overlay network) που ονομάζεται Visibility-based Overlay Network (VON) και βρίσκεται πάνω από το αρχικό δίκτυο όπως φαίνεται στην παρακάτω εικόνα:



Εικόνα 190 Overlay network VON (πηγή: www.researchgate.net/publication/224500413_Visibility-Graph-Based_Shortest-Path_Geographic_Routing_in_Sensor_Networks)

Για να μειώσουμε το πλήθος των ακμών σε έναν γράφο ορατότητας χρησιμοποιούμε μια τεχνική για να κατασκευάσουμε τον μειωμένο γράφο ορατότητας (Reduced Visibility Graph-RVG). Λέμε ότι το ευθύγραμμο τμήμα \overline{xy} είναι εφαπτόμενο αν προεκτείνοντάς το αυτό εφάπτεται στις κορυφές x και y των πολυγωνικών εμποδίων. Αν από το Γράφο Ορατότητας VG αφαιρέσουμε όλα τα μη εφαπτόμενα ευθύγραμμοι τμήματα τότε προκύπτει ο Μειωμένος Γράφος Ορατότητας RVG, όπως φαίνεται στην Εικόνα 191(b).



Εικόνα 191 (α) Γράφος Ορατότητας (β) Μειωμένος Γράφος Ορατότητας. Με γκρι χρώμα απεικονίζονται τα εμπόδια και η κόκκινη γραμμή απεικονίζει τη συντομότερη διαδρομή από το s στο t . (πηγή: www.researchgate.net/publication/224500413_Visibility-Graph-Based_Shortest-Path_Geographic_Routing_in_Sensor_Networks)

Η βασική ιδέα πίσω από τη δρομολόγηση με βάση το Γράφο Ορατότητας (VG-based routing) είναι να εντοπίσει ποιοι από τους κόμβους του αρχικού δικτύου αντιπροσωπεύουν κορυφές πολυγωνικών εμποδίων και στη συνέχεια να τους οργανώσει σε ένα όσο το δυνατόν μικρότερο δίκτυο που ονομάζεται VON (Visibility-based Overlay Network). Τότε, όταν ένας κόμβος-πηγή θέλει να μεταφέρει ένα μήνυμα σε κάποιον κόμβο προορισμού δεν έχει παρά να γίνει μέλος αυτού του δικτύου ώστε να υπολογιστεί στη συνέχεια η συντομότερη διαδρομή προς τον κόμβο προορισμού.

Βιβλιογραφία

Ακολουθούν οι βιβλιογραφικές αναφορές (πηγές) της Εργασίας.

- 1) Eiben, A.E. & Smith. J.E. (2015). *Introduction to Evolutionary Computing.*, Springer-Verlag Berlin Heidelberg, 2nd Edition.
- 2) Berg, M. & Cheong, O. & Kreveld, M. & Overmars, M. (2008). *Computational Geometry -Algorithms and Applications.*, Springer-Verlag Berlin Heidelberg, 3rd Edition.
- 3) Δημητρίου, Π. (2020) *Υπολογισμός Τριγωνοποίησης Delaunay με χρήση Γενετικών Αλγορίθμων* (Αδημοσίευτη Μεταπτυχιακή Εργασία). Ελληνικό Ανοικτό Πανεπιστήμιο, Πάτρα.
- 4) An $O(n^2 \log n)$ Algorithm for Computing Visibility Graphs. (2020, Οκτώβριος 8) Ανακτήθηκε από <http://www.science.smith.edu>
- 5) Gray Code. (2020, Οκτώβριος 12) Ανακτήθηκε από https://en.wikipedia.org/wiki/Gray_code
- 6) Ανασυνδυασμός γονιδίων (2020, Οκτώβριος 12) Ανακτήθηκε από https://el.wikipedia.org/wiki/%CE%91%CE%BD%CE%B1%CF%83%CF%85%CE%BD%CE%B4%CF%85%CE%B1%CF%83%CE%BC%CF%8C%CF%82_%CE%B3%CE%BF%CE%BD%CE%B9%CE%B4%CE%AF%CF%89%CE%BD
- 7) Visibility-Graph-Based Shortest-Path Geographic Routing in Sensor Networks (2021, Ιανουάριος, 14) Ανακτήθηκε από https://www.researchgate.net/publication/224500413_Visibility-Graph-Based_Shortest-Path_Geographic_Routing_in_Sensor_Networks
- 8) <https://shapely.readthedocs.io/en/latest/manual.html?highlight=Polygon#shapely.geometry.polygon.orient>
- 9) Tan P.N., Steinbach M., Karpathe A. & Kumar V. (2020), *Εξόρυξη Δεδομένων*. Θεσσαλονίκη. Εκδόσεις Τζιόλα
- 10) DNA Computing (2021, Ιούνιος 13) Ανακτήθηκε από https://en.wikipedia.org/wiki/DNA_computing
- 11) Polymerase chain reaction (2021, Ιούνιος 13) Ανακτήθηκε από https://en.wikipedia.org/wiki/Polymerase_chain_reaction
- 12) Hamiltonian path (2021, Ιούνιος 13) Ανακτήθηκε από https://en.wikipedia.org/wiki/Hamiltonian_path

- 13) DNA computing based RNA genetic algorithm with applications in parameter estimation of chemical engineering processes (2021, Ιούνιος 13) Ανακτήθηκε από www.science direct.com
- 14) RNA (2021, Ιούνιος 13) Ανακτήθηκε από <https://el.wikipedia.org/wiki/RNA>
- 15) Molecular computing: Does DNA compute? (2021, Ιούνιος 13) Ανακτήθηκε από www.science direct.com
- 16) M. S. Veach, “Recognition and Reconstruction of Visibility Graphs Using a Genetic Algorithm”, Proc. of the 1st annual conference on genetic programming, pp. 491–496, July 1996.
- 17) *Path Planning for a Tethered Mobile Robot* (2021, Ιούλιος 6) Ανακτήθηκε από researchgate.net/publication/286680267_Path_planning_for_a_tethered_mobile_robot
- 18) Salzman O., Halperin D., (2015) *Optimal motion planning for a tethered robot: Efficient preprocessing for fast shortest paths queries*. IEEE International Conference on Robotics and Automation (ICRA), May 26-30, 2015 (pp. 4161-4166). Washington State Convention Center, Seattle, Washington
- 19) Πλατής Ι.Δ. (2017), *MEM232-Τοπολογία: Πρόχειρες Σημειώσεις*. Τμήμα Μαθηματικών και Εφαρμοσμένων Μαθηματικών, Πανεπιστήμιο Κρήτης
- 20) Homotopy (2021, Ιούλιος 6) Ανακτήθηκε από <https://en.wikipedia.org/wiki/Homotopy>
- 21) Song B., Wang Z., Sheng L.(2016) A new genetic algorithm approach to smooth path planning for mobile robots. *Assembly Automation* 36(2), pp 138-145. DOI:10.1108/AA-11-2015-094
- 22) Andayesh M., Sadeghpour F. (2014) A comparative study of different approaches for finding the shortest path on construction sites. *Creative Construction Conference*, pp 33-41
- 23) Ma J., Liu Y., Zang S., Wang L. (2020) Robot path planning based on genetic algorithm fused with continuous Bezier optimization. *Computational Intelligence and Neuroscience*, Volume 2020, Article ID 9813040, 10 pages
- 24) https://en.wikipedia.org/wiki/B%C3%A9zier_curve (2021, Ιούλιος 7) Ανακτήθηκε από https://en.wikipedia.org/wiki/B%C3%A9zier_curve
- 25) Salzman O. (2019) Sampling-based robot motion planning. *Communications of the ACM*. Vol. 62. No. 10, pp. 54-63 doi: 10.1145/3318164

- 26) Givigi S., Carvalho E. A. N., Freire E. O.E, Molina L. (2017) Heuristics for the multi-robot worst-case pursuit-evasion problem. IEEE Access. DOI: 10.1109/ACCESS.2017.2739641
- 27) L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*. IEEE Trans. on Robotics & Automation , 12(4) : 566 - 580, June 1996.
- 28) Garrido S., Abderrahim M., Moreno L. (2006) *Path planning and navigation using Voronoi diagram and Fast Marching*.(2021, Ιούλιος 10) Ανακτήθηκε από https://www.researchgate.net/publication/260287339_Mobile_Robot_Path_Planning_using_Voronoi_Diagram_and_Fast_Marching
- 29) Nagib G., Gharieb W. (2004) *Path planning for a mobile robot using genetic algorithms* (2021, Ιούλιος 10) Ανακτήθηκε από www.researchgate.net/publication/4113354
- 30) Ηλιακοπούλου Α., Καργάκος Α. (2012) *Ανάπτυξη μεθόδων εξερεύνησης και πλήρους κάλυψης αγνώστου χώρου από ρομπότ με εφαρμογή σε αναζήτηση θυμάτων* (Διπλωματική εργασία). Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης, Θεσσαλονίκη.
- 31) Zhang H., Lin W., Chen A. (2018) Path planning for a mobile robot: a preview. *Symmetry* 2018, 10, 450; DOI: 10.3390/sym10100450
- 32) Goerzen C., Kong Z., Mettler B. (2010) A survey of motion planning algorithms from the perspective of autonomous UAV guidance. *J Intel Robot Syst* (2010) 57:65-100 DOI 10.1007/s10846-009-9383-1
- 33) Rohnert H. (1986) Shortest paths in the plane with convex polygonal obstacles. *Information Processing Letters* 23 (1986) 71-76
- 34) Wein R., Van den Berg J. P., Halperin D. (2005) The visibility-Voronoi complex and its applications. *Computational Geometry* 36 (2007) 66-87
- 35) Hwang Y. K. (1992) Gross Motion Planning-A Survey (2021, Ιούλιος 16) Ανακτήθηκε από <https://dl.acm.org/doi/10.1145/136035.136037>
- 36) Sridharan K., Priya T. K. (2004) A parallel algorithm for constructing reduced visibility graph and its FPGA implementation. *Journal of Systems Architecture* 50 (2004) 635-644.
- 37) Lozano-Perez T., Wesley M. A. (1979) An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Communications of the ACM*, vol. 22, no 10 (Oct 1979), pp. 560-570
- 38) Geometric K Shortest Paths (2021, Ιούλιος 18) Ανακτήθηκε από: <https://weber.itn.liu.se/~valpo40/pages/kspSlides.pdf>

- 39) Scharir M., Schorr A. (1984) On Shortest Paths in Polyhedral Spaces. (2021, Ιούλιος 18) Ανακτήθηκε από: <https://dl.acm.org/doi/pdf/10.1145/800057.808676>
- 40) Storer J., Reif J., H. (1985) Shortest Paths in the Plane with Polygonal Obstacles. *Journal of the ACM*, 41:5, (September, 1994), pp. 982-1012
- 41) Inkulu R., Kapoor S., Maheshwari S. N. (2010) A near optimal algorithm for finding Euclidean shortest path in polygonal domain. (2021, Ιούλιος 18) Ανακτήθηκε από: <https://arxiv.org/abs/1011.6481>
- 42) Kapoor S., Maheshwari S. N. (2000) Efficiently Constructing the Visibility Graph of a Simple Polygon with Obstacles. *SIAM Journal on Computing*, 30(3): 847-871, (2000)
- 43) Dhaenens C., Espinouse M., Penz B. (2008). Classical Combinatorial Problems and Solution Techniques. *Operations Research and Networks*. pp.71-103 (2021, Ιούλιος 18) Ανακτήθηκε από: <https://onlinelibrary.wiley.com/doi/book/10.1002/9780470611753>
- 44) Masehian E., Sedighizadeh D. (2007) Classic and Heuristic Approaches in Robot Motion Planning-A Chronological Review. *International Journal of Mechanical, Industrial Science and Engineering*. Vol:1 No:5, pp.255-260
- 45) Rapidly-Exploring Random Tree (2021, Ιούλιος 23).Ανακτήθηκε από: https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree
- 46) AL-Taharwa I., Sheeta A., Al-Weshah M. (2008) A Mobile Robot Path Planning using Genetic Algorithm in Static Environment. *Journal of Computer Science* 4(4). pp. 341-344
- 47) Kumar A., Arunadevi J., Mohan V. (2009) Intelligent Transport Route Planning Using Genetic Algorithms in Path Computation Algorithms. *European Journal of Scientific Research* Vol.25 No.3, pp. 463-468
- 48) networkx.algorithms.swap.double_edge_swap (2021, Αύγουστος 10). Ανακτήθηκε από: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.swap.double_edge_swap.html
- 49) Pyvisgraph-Python Visibility Graph (2021, Αύγουστος 11). Ανακτήθηκε από: github.com/TaipanRex/pyvisgraph/blob/master/README.md
- 50) Lee D. T., *Proximity and reachability in the plane*, Ph.D. thesis and Tech. Report ACT-12, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL (1978)
- 51) LaValle S. (2006) *Planning Algorithms*. Cambridge University Press

- 52) Christos Tsanikidis, Margarita Vitoropoulou, Vasileios Karyotis, Symeon Papavassiliou, On the Energy-Efficient Coverage of Network Regions with Convex Opaque Obstacles, Proc. IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), Sept. 2018.

Υπεύθυνη Δήλωση Συγγραφέα:

Δηλώνω ρητά ότι, σύμφωνα με το άρθρο 8 του Ν.1599/1986, η παρούσα εργασία αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης.