ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ ΠΑΡΑΓΩΓΗΣ & ΔΙΟΙΚΗΣΗΣ

ΜΕΘΕΥΡΕΤΙΚΟΣ ΑΛΓΟΡΙΘΜΟΣ ΓΙΑ ΤΟ ΠΡΟΒΛΗΜΑ ΔΡΟΜΟΛΟΓΗΣΗΣ ΟΧΗΜΑΤΩΝ

ΤΡΕΜΜΑΣ ΒΙΚΤΩΡ

2015010148

Χανιά,2025

Περιεχόμενα

Ευχαριστίες

Με την ολοκλήρωση αυτής της διπλωματικής εργασίας, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες σε όλους όσους συνέλαβαν στην επιτυχία αυτής της προσπάθειας.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Μαρινάκη Ιωάννη, για την καθοδήγηση και την υποστήριξή του καθ' ολη τη διάρκεια της εκπόνησης της εργασίας αυτής.

Ιδιαίτερες ευχαριστίες οφείλονται στους γονείς μου για την αμέριστη υποστήριξη και την υπομονή κατά τη διάρκεια των σπουδών μου. Χωρίς την στήριξή τους, δεν θα είχα κατορθώσει να φτάσω σε αυτό το σημείο. Επίσης θα ήθελα να ευχαριστήσω τον αδερφό μου για τη στήριξη και τη πολύτιμη βοήθειά του αυτά τα χρόνια.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους και τους συμφοιτητές μου, για τη συνεργασία και τις στιγμές που μοιραστήκαμε κατά τη διάρκεια των σπουδών μου.

Εισαγωγή

Η εφοδιαστική αλυσίδα είναι ένας θεμελιώδης τομέας της σύγχρονης επιχειρηματικότητας, που συνδυάζει τη διαχείριση της προμήθειας, της παραγωγής, της αποθήκευσης και της διανομής αγαθών. Η ιστορία της εφοδιαστικής αλυσίδας είναι μια μακρά και πολύπλοκη διαδρομή που αποτυπώνει τις εξελίξεις στο εμπορικό και παραγωγικό τοπίο από τις αρχαίες εποχές μέχρι σήμερα.

Η αρχαία εποχή προσφέρει τα πρώτα δείγματα εφοδιαστικών διαδικασιών, όπως οι εμπορικές οδοί που συνδέουν διαφορετικές περιοχές για την ανταλλαγή αγαθών. Οι πολιτισμοί όπως οι Αιγύπτιοι, οι Ρωμαίοι και οι Κινέζοι είχαν αναπτύξει πρωτόγονες μορφές αποθήκευσης και μεταφοράς αγαθών, αναγνωρίζοντας τη σημασία της οργάνωσης και της διαχείρισης του εμπορίου. Με την ανάπτυξη των πρώτων πολιτισμών και τη δημιουργία μεγάλων αυτοκρατοριών, η εφοδιαστική αλυσίδα εξελίχθηκε από απλές μεταφορές σε πιο οργανωμένα δίκτυα διανομής. Η βιομηχανική επανάσταση του 18ου και 19ου αιώνα σηματοδότησε μια καθοριστική αλλαγή στην εφοδιαστική αλυσίδα. Η μαζική παραγωγή, η ανάπτυξη των σιδηροδρόμων και η βελτίωση των τεχνολογιών μεταφοράς δημιούργησαν νέες προκλήσεις και ευκαιρίες στη διαχείριση της αλυσίδας εφοδιασμού. Οι επιχειρήσεις άρχισαν να εφαρμόζουν πιο σύνθετες μεθόδους για την αποθήκευση, την παρακολούθηση και τη μεταφορά προϊόντων, προκειμένου να καλύψουν τις αυξανόμενες ανάγκες των αγορών. Στον 20ό αιώνα, η τεχνολογική επανάσταση έφερε νέες αλλαγές στην εφοδιαστική αλυσίδα. Η εισαγωγή υπολογιστικών συστημάτων, η ανάπτυξη λογισμικού για τη διαχείριση αποθεμάτων και οι προηγμένες μέθοδοι ανάλυσης δεδομένων επαναστατούν τη βιομηχανία. Η παγκοσμιοποίηση, η αύξηση της πολυπλοκότητας των αλυσίδων εφοδιασμού και η ανάγκη για γρήγορη και ακριβή εξυπηρέτηση του πελάτη, ανέδειξαν την εφοδιαστική αλυσίδα σε στρατηγικό πλεονέκτημα για τις επιχειρήσεις. Οι σύγχρονες επιχειρήσεις αναγνωρίζουν ότι η αποδοτική διαχείριση της εφοδιαστικής αλυσίδας είναι κρίσιμη για την ανταγωνιστικότητα και την επιτυχία τους σε έναν συνεχώς μεταβαλλόμενο οικονομικό και εμπορικό περιβάλλον. Στο πλαίσιο της σύγχρονης εφοδιαστικής αλυσίδας, ο προγραμματισμός δρομολόγησης οχημάτων (Vehicle Routing Problem - VRP) παίζει έναν κρίσιμο ρόλο στην αποτελεσματική διαχείριση των μεταφορών. Το VRP αναφέρεται στη διαδικασία σχεδίασης και βελτιστοποίησης των διαδρομών που πρέπει να ακολουθήσουν τα οχήματα για τη διανομή προϊόντων από τα κέντρα παραγωγής ή αποθήκευσης μέχρι τους τελικούς καταναλωτές. Ο σωστός προγραμματισμός δρομολογίων όχι μόνο μειώνει το κόστος μεταφοράς και βελτιώνει την αποδοτικότητα των επιχειρησιακών διαδικασιών, αλλά συμβάλλει επίσης στη μείωση του περιβαλλοντικού αποτυπώματος και στην αύξηση της ικανοποίησης των πελατών.

Στην παρούσα εργασία, θα αναπτύξουμε έναν αλγόριθμο για την επίλυση του προβλήματος δρομολόγησης οχημάτων χρησιμοποιώντας τη γλώσσα προγραμματισμού Python. Σκοπός μας είναι να βελτιστοποιήσουμε τη διαδικασία δρομολόγησης οχημάτων λαμβάνοντας υπόψη τους διάφορους περιορισμούς που σχετίζονται με την εφοδιαστική αλυσίδα, όπως η διαθεσιμότητα των οχημάτων, οι χωρητικότητες των φορτίων, και οι χρονικοί περιορισμοί.

Κεφάλαιο 1: Logistics

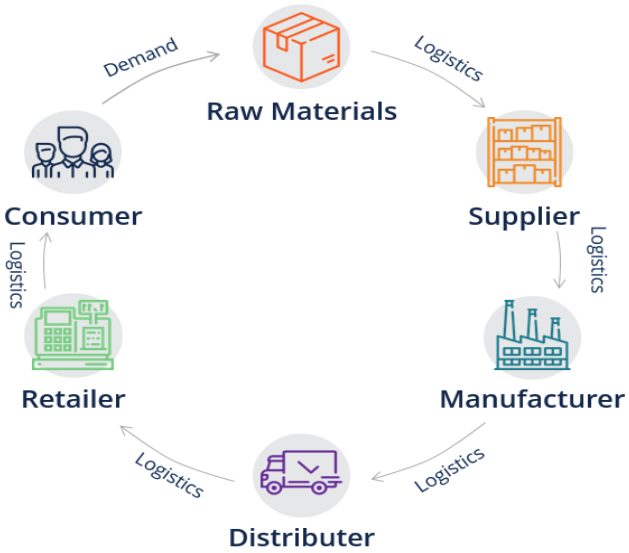
1.1 Εφοδιαστική Αλυσίδα (Supply Chain)

Η εφοδιαστική αλυσίδα είναι το σύστημα μέσω του οποίου προϊόντα και υπηρεσίες μετακινούνται από την αρχική τους πηγή μέχρι τον τελικό καταναλωτή. Σκοπός της είναι να διασφαλίσει ότι οι κατάλληλες ποσότητες προϊόντων φτάνουν στον προορισμό τους με τον πιο αποτελεσματικό, γρήγορο και οικονομικό τρόπο. Στην εφοδιαστική αλυσίδα περιλαμβάνονται όλα τα στάδια από την απόκτηση πρώτων υλών, την παραγωγή, τη διανομή και την παράδοση στον πελάτη, καθώς και η διαδικασία της επιστροφής προϊόντων.

Η εφοδιαστική αλυσίδα αποτελεί βασικό συστατικό της οικονομίας, καθώς υποστηρίζει τη ροή των αγαθών και υπηρεσιών και συνδέει τις επιχειρήσεις με τους καταναλωτές. Η αποτελεσματική διαχείρισή της συμβάλλει σημαντικά στη μείωση κόστους, στη βελτίωση της ποιότητας των προϊόντων και στην ικανοποίηση του πελάτη. Η λειτουργικότητα της αλυσίδας εξαρτάται από τη συνεργασία και την επικοινωνία μεταξύ των εταιρειών και των προμηθευτών τους, καθιστώντας την εφοδιαστική αλυσίδα έναν ζωτικής σημασίας τομέα για την επιχειρηματική επιτυχία.

Η εφοδιαστική αλυσίδα αποτελείται από τα παρακάτω βασικά στοιχεία:

- **Προμήθεια πρώτων υλών:** Η αρχική φάση περιλαμβάνει την απόκτηση των απαραίτητων πρώτων υλών και εξαρτημάτων που είναι αναγκαία για την παραγωγή ενός προϊόντος.
- **Παραγωγή:** Ακολουθεί η διαδικασία παραγωγής, κατά την οποία οι πρώτες ύλες και τα υλικά μετατρέπονται σε τελικά προϊόντα μέσα από διαδικασίες επεξεργασίας και συναρμολόγησης.
- **Διανομή:** Μετά την παραγωγή, τα προϊόντα μεταφέρονται σε αποθήκες και στη συνέχεια στους πελάτες. Η διανομή περιλαμβάνει τον σχεδιασμό της μεταφοράς, την αποθήκευση, και την τελική διανομή.
- **Επιστροφή προϊόντων:** Η διαδικασία επιστροφής αφορά είτε τα ελαττωματικά προϊόντα είτε εκείνα που δεν πληρούν τις προδιαγραφές και πρέπει να επανεισαχθούν στην αλυσίδα παραγωγής ή να αποσυρθούν.



Εικόνα 1: Τραβήχτηκε από https://corporatefinanceinstitute.com/

Οι βασικές λειτουργίες της διαχείρισης της εφοδιαστικής αλυσίδας περιλαμβάνουν την πρόβλεψη ζήτησης, τη διαχείριση αποθεμάτων, την επιλογή και αξιολόγηση προμηθευτών, την παρακολούθηση των παραγγελιών, καθώς και την οργάνωση των μεταφορών. Ένα σημαντικό στοιχείο είναι η συνεχής βελτίωση των λειτουργιών με στόχο τη μείωση κόστους, την αύξηση της παραγωγικότητας και την επίτευξη των ποιοτικών στόχων.

Συμπερασματικά, η εφοδιαστική αλυσίδα είναι ένα κρίσιμο εργαλείο για τη βιωσιμότητα και την ανάπτυξη των επιχειρήσεων. Μέσω αποτελεσματικών πρακτικών διαχείρισης, οι εταιρείες μπορούν να επιτύχουν βελτιωμένη απόδοση, χαμηλότερα κόστη και υψηλότερη ικανοποίηση πελατών. Η συνεχής πρόοδος στον τομέα, με τη χρήση καινοτόμων τεχνολογιών και βιώσιμων πρακτικών, ενισχύει τη σημασία της για τις επιχειρήσεις και την κοινωνία.

1.2 Τι είναι τα Logistics και που τα συναντάμε;

Τα logistics αποτελούν έναν κρίσιμο τομέα της σύγχρονης επιχειρηματικής δραστηριότητας που περιλαμβάνει τη διαδικασία σχεδίασης, εκτέλεσης και ελέγχου της αποτελεσματικής διαχείρισης και αποθήκευσης αγαθών και υπηρεσιών από το σημείο προέλευσης έως το σημείο κατανάλωσης. Ο κύριος στόχος των logistics είναι να εξασφαλίσει ότι τα προϊόντα παραδίδονται στον τελικό καταναλωτή στην κατάλληλη ποσότητα, στην κατάλληλη ποιότητα, στη σωστή τοποθεσία και στον κατάλληλο χρόνο, με το μικρότερο δυνατό κόστος για την επιχείρηση.

Βασικές Λειτουργίες των Logistics:

Διαχείριση Αποθεμάτων: Η διαχείριση αποθεμάτων επικεντρώνεται στην παρακολούθηση και έλεγχο των ποσοτήτων των αγαθών που διατηρούνται στις αποθήκες. Περιλαμβάνει τη λήψη αποφάσεων για την παραγγελία νέων προϊόντων, την αποθήκευση και την οργάνωση των αποθεμάτων, με στόχο τη βελτιστοποίηση των επιπέδων αποθεμάτων ώστε να αποφεύγονται οι ελλείψεις ή οι υπερβολικές ποσότητες που αυξάνουν το κόστος αποθήκευσης.

Μεταφορά και Διανομή: Αυτή η λειτουργία περιλαμβάνει τη μεταφορά των προϊόντων από το σημείο παραγωγής ή αποθήκευσης έως το σημείο κατανάλωσης. Η διαχείριση της μεταφοράς καλύπτει την επιλογή των μεταφορικών μέσων (όπως φορτηγά, πλοία, αεροπλάνα), τη σχεδίαση δρομολογίων και την εξασφάλιση της ασφαλούς και έγκαιρης παράδοσης.

Αποθήκευση: Περιλαμβάνει τη διαδικασία αποθήκευσης των προϊόντων σε αποθήκες. Η αποθήκευση αφορά την παραλαβή, τη διαχείριση και την προετοιμασία των προϊόντων για αποστολή, με στόχο την αποδοτικότητα και τη μείωση του κόστους.

Σχεδίαση Εφοδιαστικής Αλυσίδας: Η σχεδίαση της εφοδιαστικής αλυσίδας περιλαμβάνει την ανάπτυξη στρατηγικών για την ολοκληρωμένη διαχείριση όλων των βημάτων της εφοδιαστικής διαδικασίας, από την προμήθεια πρώτων υλών μέχρι την τελική παράδοση των προϊόντων στον καταναλωτή.

Διαχείριση Παραγγελιών: Επικεντρώνεται στην επεξεργασία παραγγελιών, από τη λήψη τους μέχρι την εκπλήρωσή τους. Περιλαμβάνει την επεξεργασία και την αποστολή των παραγγελιών, τη διαχείριση των αποθεμάτων και την επικοινωνία με τους πελάτες για την ενημέρωση σχετικά με την κατάσταση των παραγγελιών.

Επιστροφές και Διαχείριση Αντικαταστάσεων: Αφορά τη διαδικασία διαχείρισης επιστροφών προϊόντων και τη διαδικασία αντικατάστασης ή αποζημίωσης. Σκοπός είναι να εξασφαλιστεί η ικανοποίηση του πελάτη και η αποδοτική διαχείριση των επιστροφών.

Που Συναντάμε τα Logistics:

Εφοδιαστική Αλυσίδα Σούπερ Μάρκετ: Στα σούπερ μάρκετ, τα logistics διαχειρίζονται την προμήθεια προϊόντων από τους προμηθευτές, τη μεταφορά τους στα καταστήματα και την αποθήκευση τους μέχρι να φτάσουν στον καταναλωτή.

Ηλεκτρονικό Εμπόριο (E-commerce): Στον τομέα του ηλεκτρονικού εμπορίου, τα logistics περιλαμβάνουν τη διαχείριση των παραγγελιών, τη συσκευασία, τη μεταφορά και την παράδοση των προϊόντων στους πελάτες.

Βιομηχανία Εφοδιασμού: Στη βιομηχανία, τα logistics διαχειρίζονται τη ροή πρώτων υλών από τους προμηθευτές, την αποθήκευση και την προετοιμασία τους για παραγωγή, καθώς και τη διανομή των τελικών προϊόντων στους πελάτες. Η αποδοτική διαχείριση των logistics μπορεί να οδηγήσει σε σημαντική μείωση του κόστους παραγωγής και αύξηση της αποδοτικότητας.

Εφοδιαστική Στρατηγική Επιχειρήσεων: Οι επιχειρήσεις σε πολλούς τομείς, από την αυτοκινητοβιομηχανία μέχρι τη φαρμακευτική βιομηχανία, στηρίζονται σε σύνθετα δίκτυα logistics για τη διαχείριση της προμήθειας και της διανομής.

Τουρισμός και Ξενοδοχεία: Στον τομέα του τουρισμού, τα logistics διαχειρίζονται τη μεταφορά τουριστών, την προμήθεια και αποθήκευση προμηθευτών για ξενοδοχεία και εστιατόρια, καθώς και τις κρατήσεις και τις διαχειρίσεις υπηρεσιών για τους πελάτες.

Δημόσια Υγειονομική Υποστήριξη: Στον τομέα της δημόσιας υγείας, τα logistics περιλαμβάνουν τη διαχείριση των προμηθευτών ιατρικών υλικών και φαρμάκων, την αποθήκευση και τη διανομή τους σε νοσοκομεία και κλινικές. Ο σωστός σχεδιασμός και η διαχείριση των logistics μπορούν να έχουν κρίσιμο αντίκτυπο στην ποιότητα των παρεχόμενων υπηρεσιών υγείας.

Διαχείριση Καταστροφών και Ανθρωπιστική Βοήθεια: Τα logistics παίζουν σημαντικό ρόλο στην οργάνωση και διαχείριση της βοήθειας σε καταστάσεις έκτακτης ανάγκης, όπως φυσικές καταστροφές ή ανθρωπιστικές κρίσεις. Περιλαμβάνουν τη διανομή τροφίμων, φαρμάκων και άλλων αναγκαίων αγαθών στις πληγείσες περιοχές.

Τα logistics είναι απαραίτητα για την ομαλή λειτουργία πολλών τομέων της σύγχρονης κοινωνίας και επιχειρηματικότητας. Επηρεάζουν την αποτελεσματικότητα, το κόστος και την ικανοποίηση των πελατών, προσδιορίζοντας την επιτυχία ή την αποτυχία επιχειρηματικών στρατηγικών. Με την ανάπτυξη τεχνολογιών και την εξέλιξη των διαδικασιών, τα logistics συνεχώς προσαρμόζονται για να καλύψουν τις αυξανόμενες ανάγκες και προκλήσεις της παγκόσμιας αγοράς.

Κεφάλαιο 2: Προβλήματα Δρομολόγησης Οχημάτων

2.1 Πρόβλημα Δρομολόγησης Οχημάτων (Vehicle Routing Problem)

Το VRP (Vehicle Routing Problem) είναι ενα σύνθετο πρόβλημα στην εφοδιαστική αλυσίδα. Το πρόβλημα δρομολόγησης οχημάτων εστιάζει στη βελτιστοποίηση των διαδρομών-μονοπατιών που πρέπει να ακολουθούν τα οχήματα, ώστε να εξυπηρετήσουν τους εκάστοτε πελάτες, με σκοπό τη μείωση του συνολικού κόστους του δρομολογίου. Το δρομολόγιο αυτό ανάλογα με την πολυπλοκότητα και τις απαιτήσεις μπορεί να εξαρτάται από διάφορους περιορισμούς όπως την απόσταση, το χρόνο εξυπηρέτησης, το μέγιστο φορτίο των μεταφορικών μέσων και άλλα πολλά.

Το VRP περιλαμβάνει πολλές εκδοχές. Σε γενικές γραμμές αν προσπαθήσουμε να το επεξηγήσουμε, βλέπουμε ότι πρακτικά εξαρτάται από τρία βασικά στοιχεία. Τα οποία είναι τα εξής:

Οχήματα: Πρόκειται για τα μεταφορικά μέσα τα οποία εκτελούν την διανομή των αγαθών. Κάθε όχημα έχει τους δικούς του περιορισμούς, όπως χωρητικότητα και διαθέσιμο χρόνο.

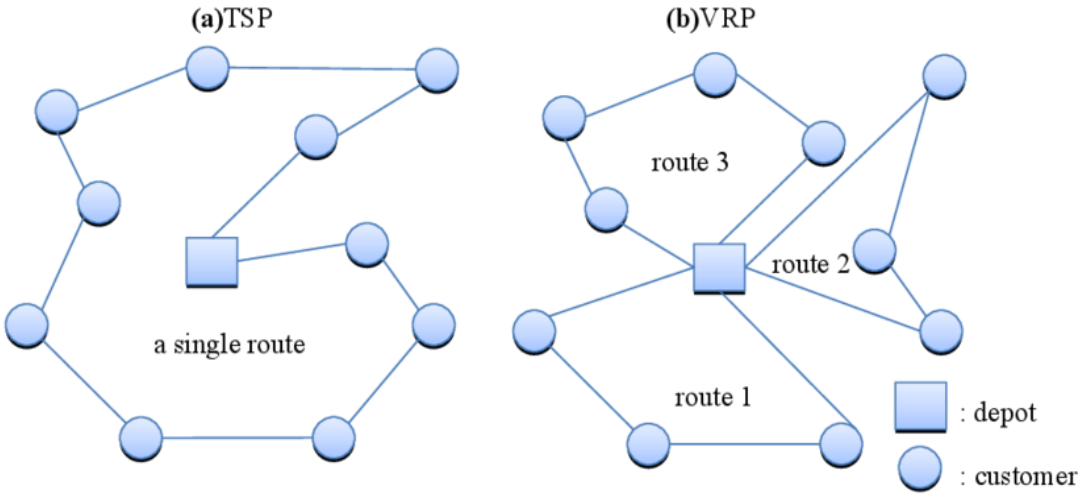
Περιορισμοί: Υπάρχουν διάφοροι περιορισμοί που πρέπει να ληφθούν υπόψιν, όπως η χωρητικότητα των οχημάτων, τα χρονικά παράθυρα για την παράδοση, οι περιορισμοί δρόμων ή κυκλοφορίας, οι συνολικές αποστάσεις που πρέπει να καλυφθούν κ.α.

Πελάτες/Σημεία Εξυπηρέτησης: Είναι τα σημεία τα οποία καταλήγουν τα αγαθά. Ανάλογα τον πελάτη ή το σημείο εξυπηρέτησης υπάρχουν συγκεκριμένες προϋποθέσεις, όπως η ποσότητα των αγαθών που πρέπει να παραδοθούν, ο χρονικός περιορισμός της παράδοσης και άλλοι περιορισμοί που μπαίνουν στην εξίσωση του VRP προς ανάλυση.

Οι στόχοι ενός σύνθετου και μη VRP είναι ξεκάθαροι:

Μείωση του κόστους: Ο κύριος στόχος του VRP είναι η μείωση του συνολικού κόστους διανομής, το οποίο μπορεί να περιλαμβάνει το κόστος καυσίμων, τη συντήρηση οχημάτων, και τις εργατικές δαπάνες.

Εξυπηρέτηση Πελατών: Διασφάλιση της έγκαιρης και σωστής παράδοσης των προϊόντων στους πελάτες, ικανοποιόντας τις απαιτήσεις και τις προτιμήσεις τους.

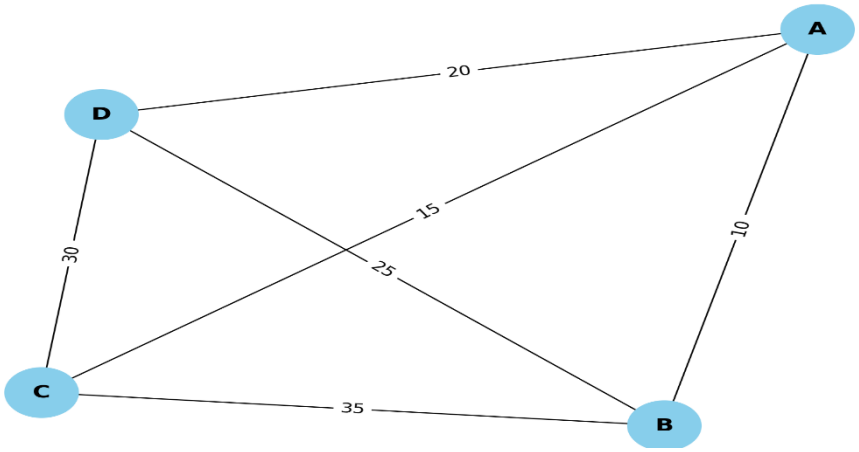


Εικόνα 2: Τραβήχτηκε από https://www.researchgate.net/

Στην παραπάνω εικόνα βλέπουμε αριστερά ένα από τα πιο γνωστά προβλήματα στην κατηγορία προβλημάτων δρομολόγησης οχημάτων. Το πρόβλημα του πλανόδιου πωλητή (TSP) - το οποίο θα αναφερθεί παρακάτω - και δεξιά ενα σύνθετο VRP με κάποιους περιορισμούς, οι οποίοι οδηγούν στη δημιουργία τριών διαφορετικών διαδρομών.

2.2 Πρόβλημα πλανόδιου πωλητή (Traveling Salesman problem)

Το πρόβλημα που αναφέρθηκε και παραπάνω. Ένα από τα πιο ευρέως γνωστά προβλήματα δρομολόγησης οχημάτων. Το Traveling Salesman Problem (TSP) αφορά τη διαδρομή που πρέπει να ακολουθήσει ένας πωλητής ώστε να επισκεφθεί μια σειρά από σημεία εξυπηρέτησης μόνο μία φορά, επιστρέφοντας στο σημείο από όπου ξεκίνησε, με στόχο φυσικά να πραγματοποιήσει την μικρότερη διαδρομή είτε χιλιομετρικά είτε χρονικά. Το TSP είναι σημαντικό όχι μόνο λόγω της πρακτικής του εφαρμογής, αλλά και γιατί αποτελεί πρότυπο για πολλά άλλα προβλήματα βελτιστοποίησης, όπως το Vehicle Routing Problem (VRP), που αποτελεί γενίκευση του TSP.



Εικόνα 3: Παράδειγμα TSP με τέσσερα σημεία εξυπηρέτησης

Στην εικόνα βλέπουμε ένα πολύ απλό παράδειγμα του πλανόδιου πωλητή με τέσσερεις κόμβους και τις αποστάσεις μεταξύ των κόμβων. Ο πωλητής ξεκινάει από την πόλη Α και πρέπει να επισκεφτεί τις υπόλοιπες πόλεις (B, C, D) και να επιστρέψει στην Α με τον μικρότερο δυνατό αριθμό χιλιομέτρων. Ο πωλητής έχει πολλές πιθανές διαδρομές. Εν τέλη ο πωλητής θα επιλέξει μία διαδρομή για να ελαχιστοποιήσει την απόσταση που θα διανύσει και να επιστρέψει το σημείο εκκίνησης Α.

2.3 Πρόβλημα δρομολόγησης οχημάτων μέσα σε χρονικούς περιορισμούς – Vehicle Routing Problem with time windows

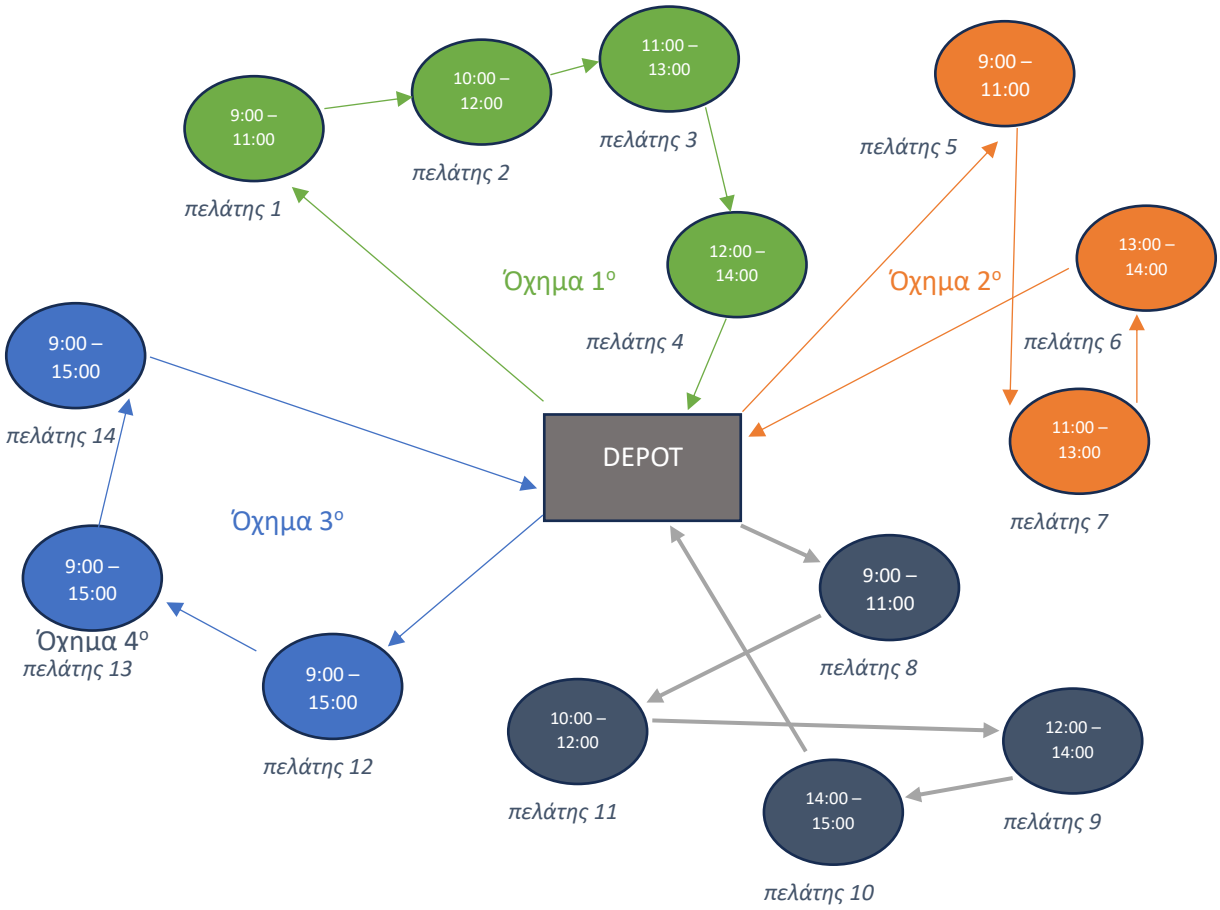
Το πρόβλημα δρομολόγησης οχημάτων με χρονικά παράθυρα αποτελεί μία από τις πιο σύνθετες εκδοχές του προβλήματος δρομολόγησης. Σε αυτό το σενάριο, ένα σύνολο από οχήματα αναλαμβάνει να εξυπηρετήσει πελάτες που βρίσκονται σε διαφορετικές τοποθεσίες, ενώ κάθε πελάτης έχει συγκεκριμένα χρονικά περιθώρια (παράθυρα) μέσα στα οποία μπορεί να εξυπηρετηθεί. Τα οχήματα, τα οποία ξεκινούν από μια κεντρική αποθήκη, πρέπει να τηρήσουν τα χρονικά όρια κάθε πελάτη, ενώ ταυτόχρονα έχουν περιορισμένη χωρητικότητα, πράγμα που περιορίζει την ποσότητα των προϊόντων που μπορούν να μεταφέρουν.

Το πρόβλημα περιλαμβάνει δύο βασικά είδη χρονικών παραθύρων:

- **Χαλαρά χρονικά παράθυρα**: Σε αυτή την περίπτωση, αν ένα όχημα φτάσει στον πελάτη εκτός του συγκεκριμένου διαστήματος, μπορεί να τον εξυπηρετήσει άμεσα. Δηλαδή, δεν υπάρχει αυστηρός χρονικός περιορισμός για την εξυπηρέτηση, αλλά προτιμάται να γίνεται εντός του καθορισμένου παραθύρου.
- **Σκληρά χρονικά παράθυρα:** Τα οχήματα πρέπει να φτάσουν και να εξυπηρετήσουν τον πελάτη μόνο εντός του προκαθορισμένου διαστήματος. Αν φτάσουν νωρίτερα,

πρέπει να περιμένουν μέχρι να ξεκινήσει το χρονικό παράθυρο, και αν φτάσουν αργότερα, δεν μπορούν να εξυπηρετήσουν τον πελάτη.

Ο στόχος είναι η εύρεση της βέλτιστης διαδρομής για κάθε όχημα, ώστε να ελαχιστοποιηθεί το συνολικό κόστος ή η απόσταση που θα διανύσουν τα οχήματα, διατηρώντας ταυτόχρονα την τήρηση των χρονικών παραθύρων και τη χωρητικότητα των οχημάτων. Έτσι, απαιτείται η εύρεση της καλύτερης δυνατής σειράς επισκέψεων, που θα επιτρέπει τη μέγιστη δυνατή εξυπηρέτηση πελατών χωρίς καθυστερήσεις και με τον βέλτιστο δυνατό προγραμματισμό των πόρων.



Στο παραπάνω γράφημα παρουσιάζεται μια προσέγγιση για την εξυπηρέτηση 14 πελατών, όπου στους κόμβους-πελάτες φαίνονται τα χρονικά παράθυρα τους. Η διαδρομή περιλαμβάνει διασταυρούμενα τόξα, λόγω της δομής των χρονικών παραθύρων των πελατών στην περιοχή, η οποία δεν επιτρέπει μια πιο άμεση διαδρομή για το όχημα.

2.4 Πρόβλημα δρομολόγησης οχημάτων με εξυπηρέτηση πελατών με παραπάνω από ένα όχημα (Split Delivery Vehicle Routing Problem)

Το πρόβλημα δρομολόγησης οχημάτων με δυνατότητα εξυπηρέτησης ενός πελάτη από πολλαπλά οχήματα (Split Delivery Vehicle Routing Problem - SDVRP) αποτελεί μια επέκταση του κλασικού προβλήματος δρομολόγησης οχημάτων (VRP). Σε αυτό το σενάριο, η ζήτηση των πελατών δεν χρειάζεται να καλυφθεί εξ ολοκλήρου από ένα μόνο όχημα. Αντίθετα, κάθε πελάτης μπορεί να εξυπηρετηθεί από περισσότερα του ενός οχήματα, ανάλογα με τη διαθεσιμότητα και τη χωρητικότητα των οχημάτων.

Η δυνατότητα αυτή προσφέρει αυξημένη ευελιξία, ειδικά όταν η ζήτηση των πελατών υπερβαίνει τη χωρητικότητα ενός οχήματος ή όταν υπάρχουν περιορισμοί στους πόρους. Ωστόσο, η πολυπλοκότητα του προβλήματος αυξάνεται, καθώς απαιτείται επιπλέον σχεδιασμός και συντονισμός για να βρεθεί η βέλτιστη κατανομή της ζήτησης κάθε πελάτη στα διαθέσιμα οχήματα.

Περιορισμός Διασπάσεων και Βελτιστοποίηση

Σε αυτή την προσέγγιση, το πρόβλημα αντιμετωπίζεται με στόχο τη μείωση του αριθμού των διασπάσεων ανά πελάτη, εξυπηρετώντας τον με όσο το δυνατόν λιγότερα οχήματα, αλλά διατηρώντας τη δυνατότητα διάσπασης αν αυτό απαιτείται. Η διαδικασία περιλαμβάνει:

- **Ελαχιστοποίηση Διασπάσεων**: Προτιμάται η εξυπηρέτηση ενός πελάτη από ένα μόνο όχημα όταν είναι εφικτό, ενώ η διάσπαση της ζήτησης εφαρμόζεται μόνο όταν δεν υπάρχουν άλλες επιλογές.
- **Συνδυαστική Βελτιστοποίηση**: Γίνεται συνδυασμός του κλασικού VRP με το SDVRP, διατηρώντας ισορροπία μεταξύ της ανάγκης για διάσπαση και της βελτιστοποίησης του συνολικού κόστους.

Παράδειγμα

Ας υποθέσουμε ότι έχουμε τρεις πελάτες και δύο οχήματα με χωρητικότητα 10 μονάδων φορτίου το καθένα. Οι πελάτες έχουν τις εξής απαιτήσεις:

- **Πελάτης Α**: 15 μονάδες φορτίου
- **Πελάτης Β**: 7 μονάδες φορτίου
- **Πελάτης Γ**: 8 μονάδες φορτίου

Στο κλασικό VRP, κάθε πελάτης πρέπει να εξυπηρετηθεί από ένα μόνο όχημα. Όμως, η ζήτηση του Πελάτη Α υπερβαίνει τη χωρητικότητα των οχημάτων, επομένως δεν μπορεί να εξυπηρετηθεί από ένα μόνο όχημα.

Με το SDVRP, μπορούμε να χωρίσουμε τη ζήτηση του Πελάτη Α ανάμεσα στα δύο οχήματα. Έτσι, η λύση μπορεί να είναι ως εξής:

- **Όχημα 1**:
    - Πελάτης Α: 10 μονάδες (όλο το φορτίο του οχήματος)
    - Πελάτης Β: 0 μονάδες (αυτός ο πελάτης θα εξυπηρετηθεί από το άλλο όχημα)
- **Όχημα 2**:
    - Πελάτης Α: 5 μονάδες (υπόλοιπο φορτίου που δεν χωράει στο Όχημα 1)
    - Πελάτης Β: 7 μονάδες (όλη η ζήτηση του πελάτη)
    - Πελάτης Γ: 8 μονάδες (όλη η ζήτηση του πελάτη)

Ανάλυση

Με αυτόν τον τρόπο:

- Ο Πελάτης Α εξυπηρετείται από δύο οχήματα, καλύπτοντας τις 15 μονάδες που χρειάζεται συνολικά.
- Ο Πελάτης Β και ο Πελάτης Γ εξυπηρετούνται από το Όχημα 2.

Η κατανομή αυτή επιτρέπει την πλήρη εξυπηρέτηση όλων των πελατών, αξιοποιώντας στο έπακρο τη χωρητικότητα των οχημάτων, χωρίς να απαιτείται επιπλέον όχημα.

2.5 Το πρόβλημα δρομολόγησης οχημάτων με πολλαπλές επιστροφές στην αποθήκη

Το πρόβλημα δρομολόγησης οχημάτων με πολλαπλές επιστροφές στην αποθήκη (Multitrip Vehicle Routing Problem - MTVRP) αποτελεί μια παραλλαγή του κλασικού προβλήματος δρομολόγησης οχημάτων (VRP). Στο MTVRP, τα οχήματα δεν περιορίζονται σε ένα μόνο ταξίδι μέσα στην ημέρα, αλλά μπορούν να επιστρέφουν στην αποθήκη για ανεφοδιασμό ή εκφόρτωση και να συνεχίζουν με νέα δρομολόγια για να εξυπηρετήσουν επιπλέον πελάτες. Αυτό προσφέρει μεγαλύτερη ευελιξία και επιτρέπει την εξυπηρέτηση περισσότερων πελατών με έναν περιορισμένο αριθμό οχημάτων.

Χαρακτηριστικά του MTVRP είναι ότι, σε αντίθεση με το κλασικό VRP, όπου κάθε όχημα ολοκληρώνει μία μόνο διαδρομή, στο MTVRP επιτρέπονται πολλαπλές διαδρομές από κάθε όχημα μέσα στην ίδια ημέρα. Αυτό αυξάνει τη δυνατότητα εξυπηρέτησης πελατών και μειώνει την ανάγκη για περισσότερα οχήματα. Επιπροσθέτως η τα οχήματα μπορούν να επιστρέφουν στην αποθήκη για να ανεφοδιαστούν ή να εκφορτώσουν προϊόντα. Αυτό είναι

ιδιαίτερα χρήσιμο όταν η ζήτηση των πελατών είναι μεγαλύτερη από τη χωρητικότητα του οχήματος, επιτρέποντας στο όχημα να εξυπηρετήσει περισσότερους πελάτες. Τέλος, προσφέρει ευελιξία στη χρήση πόρων. Με την επαναχρησιμοποίηση των οχημάτων μέσα στην ημέρα, το MTVRP μειώνει την ανάγκη για μεγάλο αριθμό οχημάτων. Αυτό μπορεί να προσφέρει σημαντικά οικονομικά και λειτουργικά οφέλη, ειδικά σε περιπτώσεις όπου οι πόροι είναι περιορισμένοι.

Ο κύριος στόχος του MTVRP είναι η ελαχιστοποίηση του συνολικού κόστους ή της απόστασης των διαδρομών, λαμβάνοντας υπόψη την ανάγκη για πολλαπλές επιστροφές στην αποθήκη. Επιπλέον, το MTVRP στοχεύει στη βέλτιστη κατανομή των πόρων (δηλαδή, οχημάτων και χρόνου), έτσι ώστε να εξυπηρετούνται όλοι οι πελάτες εντός του καθορισμένου χρόνου λειτουργίας, χωρίς να υπερβαίνεται η χωρητικότητα των οχημάτων.

Η μεγαλύτερη πρόκληση του MTVRP είναι η πολυπλοκότητα της διαχείρισης πολλαπλών διαδρομών, καθώς και η ανάγκη για αποτελεσματικό συντονισμό των επιστροφών στην αποθήκη. Η πολυπλοκότητα αυξάνεται καθώς τα οχήματα πρέπει να βελτιστοποιούν όχι μόνο την εξυπηρέτηση των πελατών, αλλά και τον χρόνο και τη συχνότητα των επιστροφών. Αυτό απαιτεί εξειδικευμένους αλγορίθμους και στρατηγικές βελτιστοποίησης που μπορούν να λάβουν υπόψη όλες αυτές τις παραμέτρους ταυτόχρονα.

Το MTVRP παρέχει μια ευέλικτη προσέγγιση στη διαχείριση δρομολογίων, επιτρέποντας στα οχήματα να εξυπηρετούν περισσότερους πελάτες μέσω πολλαπλών επιστροφών στην αποθήκη. Η παραλλαγή αυτή είναι ιδιαίτερα χρήσιμη για εταιρείες με υψηλή ζήτηση ή περιορισμένους πόρους, καθώς μεγιστοποιεί την αποδοτικότητα των οχημάτων και μειώνει το συνολικό κόστος.

Κεφάλαιο 3: Αλγόριθμοι που θα μελετήσουμε

3.1 Αλγόριθμος πλησιέστερου γείτονα

Ο αλγόριθμος του πλησιέστερου γείτονα (Nearest Neighbor Algorithm) είναι μια βασική μέθοδος που χρησιμοποιείται κυρίως για ταξινόμηση και παλινδρόμηση, αλλά και σε προβλήματα βελτιστοποίησης, όπως η εύρεση της συντομότερης διαδρομής, για παράδειγμα Traveling Salesman Problem - TSP). Λειτουργεί με βάση την αρχή της εγγύτητας, όπου ο αλγόριθμος προσδιορίζει τα «k» πλησιέστερα σημεία δεδομένων στον χώρο χαρακτηριστικών σε ένα δεδομένο σημείο εισόδου. Η απόσταση μεταξύ των σημείων μετριέται συνήθως χρησιμοποιώντας μετρήσεις όπως η Ευκλείδεια απόσταση, η απόσταση του Μανχάταν ή άλλες, ανάλογα με τη φύση των δεδομένων. Σε αυτή τη διπλωματική θα υπολογίσουμε την ευκλείδεια απόσταση των πόλεων.

Ένα από τα βασικά πλεονεκτήματα του αλγόριθμου του πλησιέστερου γείτονα είναι η ευκολία εφαρμογής του. Το συναντάμε σε προβλήματα όπως το K-Nearest Neighbors (K-NN), όπου το νέο δεδομένο ταξινομείται με βάση την κατηγορία της πλειοψηφίας των πλησιέστερων σημείων. Σε παλινδρόμηση όπου η τιμή του νέου δεδομένου εκτιμάται με βάση τον μέσο όρο των τιμών των πλησιέστερων σημείων. Τέλος σε προβλήματα όπως το Travelling Salesman Problem όπου βρίσκει τη συντομότερη διαδρομή μεταξύ σημείων.

Η βασική διαδικασία της μεθόδου αποτελείται από τα παρακάτω βήματα:

1. Αφού υπολογίσουμε τις ευκλείδειες αποστάσεις των σημείων εξυπηρέτησης, θέτουμε ως αφετηρία έναν οποιονδήποτε κόμβο ως ξεκίνημα του μονοπατιού.
2. Αναζητούμε τον κόμβο που είναι πλησιέστερος στο τελευταίο σημείο που βρισκόμασταν. Προσθέτουμε τον κόμβο αυτόν στο μονοπάτι.
3. Επαναλαμβάνουμε το δεύτερο βήμα έως ότου όλοι οι κόμβοι ενταχθούν στο μονοπάτι ακριβώς μία φορά.

Αρχικά λοιπόν, το μονοπάτι μας αποτελείται από έναν μονάχα κόμβο «i» . Αναζητούμε τον κόμβο «k» του οποίου το κόστος c_{ik}

είναι το ελάχιστο και δημιουργούμε το μονοπάτι i-k-i. Στη συνέχεια βρίσκουμε τον κόμβο «k» που δεν είναι ήδη στη διαδρομή και είναι στην ελάχιστη απόσταση από οποιονδήποτε κόμβο της διαδρομής μας. Βρίσκουμε το τόξο που ελαχιστοποιεί το $c_{ik} + c_{kj} - c_{ij}$ και εισάγουμε το «k» ανάμεσα στα i και j. Επαναλαμβάνουμε τη διαδικασία αναζήτησης του «k» που δεν υπάρχει στη διαδρομή και τη διαδικασία τοποθέτησης του τόξου έως ότου έχουμε κύκλο Hamilton δηλαδή να έχουμε περάσει από όλους τους κόμβους ακριβώς μία φορά.

Παρακάτω θα αναλύσουμε ένα απλό πρόβλημα δρομολόγησης οχημάτων με δεδομένες τις συντεταγμένες σε ευκλείδεια μορφή. Δίνονται για παράδειγμα οι παρακάτω συντεταγμένες των 10 πόλεων.

Table 3.1.1

Πόλη	**x^{cord}**	**y^{cord}**
1	37	52
2	49	49
3	52	64
4	20	26
5	40	30
6	21	47
7	17	63
8	31	62
9	52	33
10	51	21

Πρώτο βήμα είναι να υπολογίσουμε και να αποθηκεύσουμε το κόστος κάθε τόξου σε έναν πίνακα. Το κόστος c_{ij} υπολογίζεται από:

$$\sqrt{\left(x_i^{cord} - x_j^{cord}\right)^2 + \left(y_i^{cord} - y_j^{cord}\right)^2}$$

Να σημειωθεί ότι το κόστος από τον κόμβο i στον j και το ανάποδο είναι ίσο.

Ο πίνακας κόστους

Table 3.1.2

Πόλη	1	2	3	4	5	6	7	8	9	10
1	0	12,36	19,20	31,06	22,20	16,76	22,82	11,66	24,20	34,01
2	12,36	0	15,29	37,01	21,02	28,07	34,92	22,20	16,27	28,07
3	19,20	15,29	0	49,67	36,05	35,35	35,01	21,09	31,00	43,01
4	31,06	37,01	49,67	0	20,39	21,02	37,12	37,64	32,75	31,40
5	22,20	21,02	36,05	20,39	0	25,49	40,22	33,24	12,36	14,21
6	16,76	28,07	35,35	21,02	25,49	0	16,49	18,02	34,01	39,69
7	22,82	34,92	35,01	37,12	40,22	16,49	0	14,03	46,09	54,03
8	11,66	22,20	21,09	37,64	33,24	18,02	14,03	0	35,80	45,61
9	24,20	16,27	31,00	32,75	12,36	34,01	46,09	35,80	0	12,04
10	34,01	28,07	43,01	31,40	14,21	39,69	54,03	46,61	12,04	0

Ξεκινάμε από τον κόμβο 1. Ο κοντινότερος κόμβος στον 1 είναι ο κόμβος 8, οπότε δημιουργούμε τον κύκλο 1-8-1. Στη συνέχεια, βρίσκουμε τον κόμβο που δεν ανήκει ακόμα στον κύκλο και είναι πιο κοντά σε κάποιον από τους κόμβους του υπάρχοντος κύκλου. Ο πλησιέστερος κόμβος είναι ο κόμβος 2.

Αξιολογούμε το πού θα εισαχθεί ο κόμβος 2 στη διαδρομή. Υπάρχουν δύο πιθανότητες: είτε η διαδρομή θα γίνει 1-2-8-1 είτε 1-8-2-1. Για να αποφασίσουμε, χρησιμοποιούμε τον τύπο κόστους: $C_{12}+C_{21}-C_{81}=12.36+22.20-11.66=22.9$ και $C_{12} + C_{28} - C_{81} = 12.36 + 22.20 - 11.66 = 22.9$. Και στις δύο περιπτώσεις, το κόστος είναι το ίδιο επειδή το πρόβλημα είναι συμμετρικό. Επομένως, επιλέγουμε την πρώτη πιθανότητα.

Συνεχίζοντας, εντοπίζουμε τον πλησιέστερο κόμβο σε οποιονδήποτε κόμβο της διαδρομής που δεν ανήκει ακόμα στη διαδρομή. Αυτός είναι ο κόμβος 7. Οι πιθανές θέσεις για την εισαγωγή του κόμβου 7 στη διαδρομή είναι:

- 1-7-2-8-1,
- 1-2-7-8-1,
- 1-2-8-7-1.

Με βάση την ανάλυση, η τρίτη επιλογή είναι η πιο συμφέρουσα. Στην παρακάτω εικόνα παρουσιάζεται η τελική μορφή του αλγορίθμου σε πίνακα.

Επανάληψη	Κόμβοι	Συνολικό Κόστος
1η	1	-
2η	1 8 1	23.32
3η	1 2 8 7 1	46.22
4η	1 2 8 7 1	71.41
5η	1 2 3 8 7 1	85.59
6η	1 9 2 3 8 7 1	113.7
7η	1 10 9 2 3 8 7 1	135.55
8η	1 5 10 9 2 3 8 7 1	137.95
9η	1 5 10 9 2 3 8 7 6 1	148.38
10η	1 4 5 10 9 2 3 8 7 6 1	177.63

Εικόνα 4: Συνοπτική παρουσίαση της λύσης του παραπάνω προβλήματος

Όπως φαίνεται παραπάνω αλγόριθμος του πλησιέστερου γείτονα είναι μια απλή και αποδοτική μέθοδος για την επίλυση προβλημάτων δρομολόγησης, όπως το Πρόβλημα του Πλανόδιου Πωλητή (Traveling Salesman Problem - TSP). Η εφαρμογή του βασίζεται στη σταδιακή αναζήτηση και εισαγωγή κόμβων με βάση την ελάχιστη απόσταση, δημιουργώντας έναν κύκλο που περνά από όλους τους κόμβους ακριβώς μία φορά. Τα αποτελέσματα που παρουσιάστηκαν αναδεικνύουν την αποτελεσματικότητα του αλγορίθμου σε συμμετρικά και σχετικά μικρά προβλήματα. Επιπλέον, ο πίνακας κόστους παρέχει μια σαφή και χρήσιμη βάση για την κατανόηση και την υλοποίηση της διαδικασίας. Συνολικά, ο αλγόριθμος είναι ιδανικός για προβλήματα με μέτριο μέγεθος, ενώ μπορεί να συνδυαστεί με βελτιστοποιητικές τεχνικές και μεθόδους για την επίλυση πιο σύνθετων προβλημάτων. Στην συνέχεια θα δούμε τον αλγόριθμο του πλησιέστερου γείτονα πως διαμορφώνεται εντάσσοντας περιορισμούς ως προς τη δρομολόγηση των οχημάτων. Κάτι τέτοιο ουσιαστικά το φέρνει πιο κοντά στην πραγματικότητα και στις καθημερινές προκλήσεις που αντιμετωπίζουν οι εταιρείες.

3.2 Πρόβλημα δρομολόγησης οχημάτων με αλγόριθμο πλησιέστερου γείτονα και περιορισμό χωρητικότητας

Ο αλγόριθμος του πλησιέστερου γείτονα με περιορισμό χωρητικότητας του οχήματος αποτελεί μια παραλλαγή που προσαρμόζεται σε προβλήματα δρομολόγησης όπου υπάρχουν

περιορισμοί στη μεταφορική ικανότητα. Η προσαρμογή αυτή επιβάλλει έναν επιπλέον περιορισμό στον αλγόριθμο, ώστε να λαμβάνει υπόψη τόσο την απόσταση όσο και την τρέχουσα χωρητικότητα του οχήματος. Για να δώσουμε τη βάση του προβλήματος και να γίνει κατανοητό έχουμε τα εξής δεδομένα.

- Ένα όχημα έχει δεδομένη μέγιστη χωρητικότητα Q.
- Κάθε πελάτης (ή κόμβος) έχει μια ζήτηση demand (π.χ. βάρος, όγκο προϊόντων).
- Στόχος: Να βρεθεί μια διαδρομή που επισκέπτεται όλους τους κόμβους, διατηρώντας το συνολικό φορτίο του οχήματος κάτω από το Q και ελαχιστοποιώντας τη συνολική απόσταση.

Ο αλγόριθμος του πλησιέστερου γείτονα με περιορισμό χωρητικότητας έχει κάποια πολύ απλά βήματα. Καθώς το μόνο που μας ενδιαφέρει είναι να πηγαίνουμε στο πιο κοντινό πελάτη χωρίς να παραβιάζουμε τη χωρητικότητα του οχήματός μας.

Βήμα 1: Υπολογισμός των αποστάσεων

- Υπολογίζουμε τον πίνακα κόστους (*table 3.1)* όπως γίνεται και στον αλγόριθμο χωρίς περιορισμούς.

Βήμα 2: Έναρξη διαδρομής

- Θέτουμε την αρχή της διαδρομής στον αποθηκευτικό χώρο ή αποθήκη (depot).
- Το όχημα ξεκινά με χωρητικότητα Q.

Βήμα 3: Επιλογή πλησιέστερου κόμβου

- Εντοπίζουμε τον πλησιέστερο κόμβο που ταυτόχρονα:
    - Δεν έχει εξυπηρετηθεί ακόμα.
    - Η ζήτησή του demand είναι μικρότερη ή ίση με την υπολειπόμενη χωρητικότητα του οχήματος.
- Ενημερώνουμε τη διαδρομή και αφαιρούμε τη ζήτηση demand από την υπολειπόμενη χωρητικότητα Q

Βήμα 4: Επιστροφή στην αποθήκη

- Αν δεν υπάρχει κόμβος που να ικανοποιεί τις συνθήκες του Βήματος 3, το όχημα επιστρέφει στην αποθήκη (depot) για επαναφόρτωση.
- Ενημερώνεται η διαδρομή και το όχημα ξαναξεκινάει τη διαδρομή με πλήρη χωρητικότητα Q.

Βήμα 5: Επαναληπτικότητα

- Η διαδικασία συνεχίζεται μέχρι να εξυπηρετηθούν όλοι οι κόμβοι.

Με τα δεδομένα που έχουμε από την απλή εκδοχή του πλησιέστερου γείτονα θα δούμε πως διαμορφώνεται η τελική διαδρομή με τον νέο αυτόν περιορισμό. Θα λάβουμε υπόψιν και τον παρακάτω πίνακα της ζήτησης ανά πελάτη:

Table 3.2.1

Πόλη	**Ζήτηση**
1	0
2	20
3	10
4	15
5	10
6	20
7	15
8	10
9	5
10	25

Έχοντας λοιπόν τον πίνακα κόστους διαδρομών έχουμε την ακόλουθη διαδικασία βήμα βήμα:

- Αρχική χωρητικότητα οχήματος 50 μονάδες. Θέση εκκίνησης Depot (1)
- Εντοπισμός πλησιέστερου κόμβου στην αποθήκη. Ο κόμβος 8. Δεν έχει εξυπηρετηθεί ακόμα και η ζήτησή του ισούται με 10 μονάδες.
  demand = demand +10 = 10 < 50. Η νέα διαδρομή είναι 1 → 8.
- Ενημερώνουμε και το συνολικό κόστος της διαδρομής:
  routecost = routecost + 11,66 = 11,66
- Εντοπισμός πλησιέστερου κόμβου στον κόμβο 8. Ο κόμβος 7. Δεν έχει εξυπηρετηθεί ακόμα και η ζήτηση του ισούται με 15.
  demand = demand +15 = 25 < 50. Η νέα διαδρομή είναι 1 → 8 → 7
- Ενημερώνουμε και το συνολικό κόστος της διαδρομής:
  routecost = routecost + 14,03 = 25,69
- Εντοπισμός πλησιέστερου κόμβου στον κόμβο 7. Ο κόμβος 6. Δεν έχει εξυπηρετηθεί ακόμα και η ζήτησή του ισούται με 20 μονάδες.
  demand = demand +20 = 45 < 50. Η νέα διαδρομή είναι 1 → 8 → 7 → 6
- Ενημερώνουμε και το συνολικό κόστος της διαδρομής:
  routecost = routecost + 16,49 = 42,18
- Εντοπισμός πλησιέστερου κόμβου στον κόμβο 6. Ο κόμβος 4. Δεν έχει εξυπηρετηθεί ακόμα **αλλά** όπως θα δούμε παραβιάζεται η χωρητικότητα του οχήματος καθώς η ζήτησή του είναι 15.
  demand = demand + 15 = 60 > 50. Η νέα διαδρομή είναι 1 → 8 → 7 → 6 → 1

- Ενημερώνουμε και το συνολικό κόστος της διαδρομής:
  routecost = routecost + 16,76 = 58,94

Με την ίδια λογική συνεχίζεται όλος ο αλγόριθμος μέχρι να εξυπηρετηθούν όλοι οι πελάτες έως ότου φτάσουμε σε μία τελική μορφή που δείχνει κάπως έτσι:

Table 3.2.2

Στον παραπάνω πίνακα έχουμε εν συντομία μια εικόνα για το πως κινείται ο αλγόριθμος. Όπου, **V** συμβολίζεται η εκάστοτε διαδρομή, **S** ο κόμβος που βρισκόμαστε τώρα, **E** ο πιο κοντινός κόμβος σε σχέση με αυτόν που βρισκόμαστε τώρα, **D** η ζήτηση του εκάστοτε πελάτη που είναι πιο κοντά, **RC** το κόστος της εκάστοτε διαδρομής και **TC** το συνολικό κόστος όλων των διαδρομών. Στη τελευταία στήλη φυσικά **R** αναγράφεται η διαδρομή που θα

V	S	E	D	RC	TC	R
1	1	8	10	11	11	1→8
1	8	7	25	25	25	1→8→7
1	7	6	45	41	41	1→8→7→6
1	6	4	60	---	---	---
1	6	1	---	57	57	1→8→7→6→1
2	1	2	20	12	69	1→2
2	2	3	30	27	84	1→2→3
2	3	9	35	58	115	1→2→3→9
2	9	5	45	70	127	1→2→3→9→5
2	5	10	70	---	---	---
2	5	1	---	92	149	1→2→3→9→5→1
3	1	4	15	31	180	1→4
3	4	10	40	62	211	1→4→10
3	10	1	---	96	245	1→4→10→1

ακολουθήσει το φορτηγό. Όπως παρατηρούμε το φορτηγό μας σε αυτή τη περίπτωση έχει καθοριστεί να γυρνάει στην αποθήκη κάθε φορά που παραβιάζεται ο περιοερισμός. Υπάρχει η δυνατότητα να συνεχιστεί η αναζήτηση ώστε να βρεθεί ποιος πελάτης δεν παραβιάζει τον περιορισμό ώστε να εξυπηρετηθεί πρώτα εκείνος, χωρίς όμως να αποτελεί την πιο κοντινή επιλογή. Απλά θέλουμε να εξαντλήσουμε την χωρητικότητα που μας έχει απομείνει έως εκείνη τη στιγμή.

Παρατηρούμε επίσης ότι σε σχέση με την απλή μορφή του προβλήματος χωρίς περιορισμούς, όπως ήταν επακόλουθο το συνολικό κόστος για την εξυπηρέτηση όλων των πελατών αυξήθηκε στις 245 μονάδες έναντι 178 μονάδων στη προηγούμενο παράδειγμα. Εν συνεχεία θα δούμε το πρόβλημα αυτό να περιπλέκεται ακόμα περισσότερο με ακόμα έναν περιορισμό.

3.3 Πρόβλημα δρομολόγησης οχημάτων με αλγόριθμο πλησιέστερου γείτονα με περιορισμό χωρητικότητας και χρόνο εξυπηρέτησης.

Ο αλγόριθμος του πλησιέστερου γείτονα με περιορισμό χωρητικότητας του οχήματος αλλά και το χρόνο εξυπηρέτησης είναι ακόμα μία από τις πολλές παραλλαγές που μπορούμε να συναντήσουμε σε VRP. Η προσαρμογή αυτή επιβάλλει έναν επιπλέον περιορισμό στον αλγόριθμο, ώστε να λαμβάνει την απόσταση, την τρέχουσα χωρητικότητα του οχήματος, τον χρόνο μετάβασης μεταξύ των κόμβων καθώς και το χρόνο που απαιτείται για την εξυπηρέτηση του εκάστοτε πελάτη. Πιο αναλυτικά σε σχέση με το προηγούμενο παράδειγμα:

- Ένα όχημα έχει δεδομένη μέγιστη χωρητικότητα Q.
- Κάθε πελάτης (ή κόμβος) έχει μια ζήτηση demand (π.χ. βάρος, όγκο προϊόντων).
- Κάθε μετάβαση από έναν κόμβο σε έναν άλλον πέρα από κόστος έχει και χρόνο.
- Κάθε πελάτης έχει χρόνο εξυπηρέτησης.
- Στόχος: Να βρεθεί μια διαδρομή που επισκέπτεται όλους τους κόμβους, διατηρώντας το συνολικό φορτίο του οχήματος κάτω από το Q, να μη παραβιάζει τους χρονικούς περιορισμούς και να ελαχιστοποιεί τη συνολική απόσταση.

Σε σχέση με το προηγούμενο παράδειγμα υπάρχουν κάποιες αλλαγές. Πλέον δεν μας ενδιαφέρει απλά να ρυθμίζεται η χωρητικότητα. Μας ενδιαφέρει σε συνδυασμό αυτής να είμαστε μέσα στα χρονικά παράθυρα. Αναλύοντας τα βήματα του αλγορίθμου έχουμε τα εξής:

Βήμα 1: Υπολογισμός των αποστάσεων

- Υπολογίζουμε τον πίνακα κόστους (*table 3.1)* όπως γίνεται και στους προηγούμενους αλγόριθμους.

Βήμα 2: Έναρξη διαδρομής

- Θέτουμε την αρχή της διαδρομής στον αποθηκευτικό χώρο ή αποθήκη (depot).
- Το όχημα ξεκινά με χωρητικότητα Q και μέγιστο χρόνο παραμονής του οχήματος MaxRouteTime μία τιμή ας πούμε 150 μονάδες.

Βήμα 3: Επιλογή πλησιέστερου κόμβου

- Εντοπίζουμε τον πλησιέστερο κόμβο που ταυτόχρονα:
    - Δεν έχει εξυπηρετηθεί ακόμα.
    - Η ζήτησή του demand είναι μικρότερη ή ίση με την υπολειπόμενη χωρητικότητα του οχήματος.

- Ο χρόνος της διαδρομής RouteTime είναι μικρότερος από το maxroutetime- (την απόσταση του κόμβου που θέλουμε να πάμε από την αποθήκη), καθώς πρέπει να συνυπολογίζεται ότι το φορτηγό θα προλάβει να πάει και πίσω στην αποθήκη σε περίπτωση παραβίασης των περιορισμών.

- Ενημερώνουμε τη διαδρομή και αφαιρούμε τη ζήτηση demand από την υπολειπόμενη χωρητικότητα Q
- Ενημερώνουμε το Routetime, το RouteCost και το TotalCost.

Βήμα 4: Επιστροφή στην αποθήκη

- Αν δεν υπάρχει κόμβος που να ικανοποιεί τις συνθήκες του Βήματος 3, το όχημα επιστρέφει στην αποθήκη (depot) για επαναφόρτωση.
- Ενημερώνεται η διαδρομή και το όχημα ξαναξεκινάει τη διαδρομή με πλήρη χωρητικότητα Q και μέγιστο χρόνο παραμονής MaxRouteTime

Βήμα 5: Επαναληπτικότητα

- Η διαδικασία συνεχίζεται μέχρι να εξυπηρετηθούν όλοι οι κόμβοι.

Με τα δεδομένα που έχουμε από την απλή εκδοχή του πλησιέστερου γείτονα θα δούμε πως διαμορφώνεται η τελική διαδρομή με τον νέο αυτόν περιορισμό. Θα λάβουμε υπόψιν και τον παρακάτω πίνακα με το χρόνο εξυπηρέτησης καθώς και έναν ακόμα πίνακα των χρόνων μετάβασης από κόμβο σε κόμβο:

Table 3.3.1

Πόλη	**Ζήτηση**	**Χρόνος Εξυπηρέτησης**
1	0	0
2	20	10
3	10	15
4	15	10
5	10	5
6	20	10
7	15	15
8	10	5
9	5	10
10	25	15

Table 3.3.2

Πόλη	1	2	3	4	5	6	7	8	9	10
1	0	10	12	9	11	15	12	17	16	11
2	10	0	13	15	12	11	15	13	14	16
3	12	13	0	11	14	8	5	17	12	13
4	9	15	11	0	13	11	9	8	12	15

5	11	12	14	13	0	10	7	7	11	12
6	15	11	8	11	10	0	12	11	10	9
7	12	15	5	9	7	12	0	12	8	10
8	17	13	17	8	7	11	12	0	9	9
9	16	14	12	12	11	10	8	9	0	11
10	11	16	13	15	12	9	10	9	11	0

Έχοντας λοιπόν τον πίνακα κόστους διαδρομών και χρόνου μετάβασης έχουμε την ακόλουθη διαδικασία βήμα βήμα:

- Αρχική χωρητικότητα οχήματος 50 μονάδες. Θέση εκκίνησης Depot (1)
- Αρχικοποίηση μέγιστης παραμονής MaxRouteTime = 150 μονάδες.
- Εντοπισμός πλησιέστερου κόμβου στην αποθήκη. Ο κόμβος 8. Δεν έχει εξυπηρετηθεί ακόμα και η ζήτησή του ισούται με 10 μονάδες. Ο χρόνος μετάβασης είναι 17 και χρόνος εξυπηρέτησης 5.
  demand = demand +10 = 10 < 50. Η νέα διαδρομή είναι 1 → 8.
  Routetime = Routetime + 17 + 5 = 22 < 150 – 17 = 133
  Άρα η κίνηση είναι εφικτή.

Με την ίδια λογική συνεχίζεται ο αλγόριθμος έως ότου εξυπηρετηθούν όλοι οι πελάτες. Είναι σημαντικό να προσέξουμε ότι ο συνολικός χρόνος από έναν κόμβο σε έναν άλλον θα πρέπει στον περιορισμό να συμπεριλαμβάνεται και ο χρόνος για να γυρίσει στην αποθήκη (depot). Η σύντομη παρουσίαση του αλγόριθμου επιλυμένου θα δείχνει ως εξής

Table 3.3.3

V	S	E	D	RC	TC	R
1	1	8	10	11	11	1→8
1	8	7	25	25	25	1→8→7
1	7	6	45	41	41	1→8→7→6
1	6	4	60	---	---	---
1	6	5	55	---	---	---
1	6	2	65	---	---	---
1	6	9	50	75	75	1→8→7→6→9
1	9	1	---	99	99	1→8→7→6→9→1
2	1	2	20	12	111	1→2
2	2	3	30	27	126	1→2→3
2	3	5	40	63	162	1→2→3→5
2	5	10	65	---	---	---
2	5	4	55	---	---	---
2	5	1	40	85	184	1→2→3→9→5→1
3	1	4	15	31	215	1→4
3	4	10	40	62	246	1→4→10
3	10	1	---	96	280	1→4→10→1

Παρατηρείται ότι δύο φορές το φορτηγό αναγκάστηκε να γυρίσει πίσω για μηδενισμό των περιορισμών. Το συνολικό κόστος των διαδρομών είναι 280 μονάδες. Όπως ήταν αναμενόμενο και πάλι έχουμε αύξηση του κόστους. Όλα αυτά τα παραδείγματα είναι μικρής κλίμακας και βλέπουμε ότι δεν είναι περίπλοκο να βρούμε τη τελική λύση. Ωστόσο, όσο τα προβλήματα μεγαλώνουν αυτός τρόπος αναζήτησης του πλησιέστερου γείτονα μπορεί να μη βγάζει επιθυμητά αποτελέσματα λόγο πολυπλοκότητας. Σε αυτή τη περίπτωση μπορεί να υπάρχουν άλλα δεδομένα, όπως πολλαπλές αποθήκες και παραπάνω από 1 όχημα εξυπηρέτησης. Στη συγκεκριμένη διπλωματική δεν χρησιμοποιήθηκαν τέτοιου είδους δεδομένα.

3.4 Βελτιστοποίηση με τη μέθοδος τοπικής αναζήτησης 2-opt του προβλήματος δρομολόγησης οχημάτων με αλγόριθμο πλησιέστερου γείτονα, υπό τους περιορισμούς χωρητικότητας και χρόνου εξυπηρέτησης.

Η ανάπτυξη ευρετικών μεθόδων, όπως ο αλγόριθμος πλησιέστερου γείτονα και η 2-opt (που προτάθηκε από τον Croes το 1958 για το TSP και αργότερα επεκτάθηκε στο VRP), έδωσε τη δυνατότητα επίλυσης μεγαλύτερων προβλημάτων. Η 2-opt θεωρείται μια σχετικά απλοϊκή μέθοδος τοπικής αναζήτησης για τους εξής λόγους:

- **Απλή Λογική:** Η βασική ιδέα της ανταλλαγής δύο ακμών είναι εύκολη στην κατανόηση και στην υλοποίηση και δεν απαιτεί σύνθετους υπολογισμούς ή δομές δεδομένων.
- **Τοπική Εστίαση:** Η 2-opt εξετάζει μόνο μικρές αλλαγές στη διαδρομή (ανταλλαγή δύο ακμών). Αυτό σημαίνει ότι εστιάζει σε μια μικρή γειτονιά της τρέχουσας λύσης και δεν εξερευνά τον χώρο λύσεων σε βάθος.
- **Εγκλωβισμός σε Τοπικά Ελάχιστα:** Λόγω της τοπικής της εστίασης, η 2-opt είναι επιρρεπής στον εγκλωβισμό σε τοπικά ελάχιστα. Δηλαδή, μπορεί να βρει μια λύση που είναι καλή σε σχέση με τις γειτονικές της, αλλά όχι η καλύτερη δυνατή (ολικό ελάχιστο).

3.4.1 Που συναντάμε τον αλγόριθμο βελτιστοποίησης 2-opt;

Πέρα από τις συνηθισμένες χρήσεις της σε απλοϊκούς αλγορίρθμος (TSP, VRP, Logistics), τη μέθοδο αυτή τη συναντάμε στον σχεδιασμό δικτύων (π.χ., τηλεπικοινωνιακά δίκτυα, δίκτυα μεταφοράς). Η 2-opt μπορεί να χρησιμοποιηθεί για τη βελτιστοποίηση της τοποθέτησης κόμβων και των συνδέσεων μεταξύ τους, με στόχο τη μείωση του κόστους κατασκευής. Στη ρομποτική, η 2-opt μπορεί να χρησιμοποιηθεί για τον σχεδιασμό διαδρομών για ρομπότ που εκτελούν εργασίες σε ένα περιβάλλον. Για παράδειγμα, σε ένα εργοστάσιο, ένα ρομπότ μπορεί να χρησιμοποιεί την 2-opt για να βρει την πιο αποδοτική διαδρομή για τη συλλογή αντικειμένων από διάφορες θέσεις. Ακόμα και στη βιοπληροφορική, η 2-opt μπορεί να

χρησιμοποιηθεί για την ανάλυση αλληλουχιών DNA ή πρωτεϊνών, με στόχο την εύρεση ομοιοτήτων και την ταξινόμηση βιολογικών μορίων. Είναι σημαντικό να σημειωθεί ότι η 2-opt, ως μέθοδος τοπικής αναζήτησης, μπορεί να μην βρίσκει πάντα την ολική βέλτιστη λύση. Ωστόσο, σε πολλές περιπτώσεις, προσφέρει ικανοποιητικές λύσεις σε λογικό υπολογιστικό χρόνο. Όταν απαιτείται υψηλότερη ακρίβεια, μπορούν να χρησιμοποιηθούν πιο προηγμένες μέθοδοι, όπως οι μετα-ευρετικές.

3.4.2 Επεξήγηση του αλγόριθμου.

Για λόγους παρουσίασης και πολυπλοκότητας δεν θα αναλυθεί ο αλγόριθμος βήμα βήμα καθώς στο επόμενο κεφάλαιο θα παρουσιαστεί ο κώδικας αυτούσιος για να γίνει κατανοητή η διαδικασία. Παρ΄ όλα αυτά παρακάτω παρουσιάζεται εν συντομία η αλληλουχία που ακολουθεί ο αλγόριθμος:

- **Αρχικοποίηση**:

  Ξεκινήστε με την αρχική λύση από τον αλγόριθμο πλησιέστερου γείτονα.

- **Βελτίωση**:

  Επιλέγει ο αλγόριθμος τυχαία δύο ακμές (i, j) που ανήκουν στην ίδια διαδρομή.

  Αντιστρέφει τη σειρά των κόμβων μεταξύ αυτών των ακμών.

- **Επαλήθευση**:

  Υπολογίζουμε αν η νέα διαδρομή είναι εφικτή:

  - Ικανοποιεί τη χωρητικότητα του οχήματος.
  - Ικανοποιεί τα χρονικά παράθυρα.

  Αν η νέα λύση είναι καλύτερη, γίνεται αποδεκτή η αλλαγή.

Επανάληψη:

- Ο αλγόριθμος συνήθως κατά επιλογή του προγραμματιστή εφαρμόζει την τοπική αναζήτηση μέχρι να μην μπορεί να βελτιώσει περαιτέρω τη λύση έως ότου όμως φτάσει έναν αριθμό επαναλήψεων πχ. 50.

Παρακάτω θα δούμε ενα τυχαίο απλό παράδειγμα για την κατανόηση του αλγόριθμου:

1. **Επιλογή Δύο Ακμών**:
   - Από τη διαδρομή επιλέγονται δύο ακμές (π.χ., μεταξύ των κόμβων A−B και C).
2. **Αντικατάσταση Ακμών**:
   - Αντί για τη σύνδεση A−B και C−D, συνδέουμε A−C και B−D.
   - Αυτό συνεπάγεται αντιστροφή της σειράς των κόμβων μεταξύ B και C.

3. **Υπολογισμός Κόστους**:
    - Υπολογίζουμε τη συνολική απόσταση της νέας διαδρομής.
    - Αν η νέα διαδρομή είναι καλύτερη (μικρότερη απόσταση), την αποδεχόμαστε.
4. **Επανάληψη**:
    - Συνεχίζουμε με νέους συνδυασμούς ακμών.

Στη πράξη το παραπάνω παράδειγμα:

Ας υποθέσουμε ότι έχουμε την εξής αρχική διαδρομή: Αρχική Διαδρομή: [0,1,2,3,4,5,0]

- Οι ακμές είναι:
    - 0→1, 1→2, 2→, 3→, 4→, 5→0
- Επιλέγουμε τις ακμές:
    - 1→4 και 3→4.

Αντικατάσταση:

- Αντί για 1→2 και 3→4, συνδέουμε 1→3 και 2→4
- Η διαδρομή γίνεται: [0,1,3,2,4,5,0]

  Υπολογίζουμε την απόσταση της νέας διαδρομής:
    - Αν η νέα απόσταση είναι μικρότερη, την αποδεχόμαστε.

Η 2-opt είναι μια απλή και αποτελεσματική μέθοδος τοπικής αναζήτησης που μπορεί να βελτιώσει σημαντικά τις λύσεις του VRP, ειδικά όταν συνδυάζεται με μια καλή αρχική λύση. Ωστόσο, λόγω της απλοϊκότητάς της, μπορεί να μην βρίσκει πάντα τις καλύτερες δυνατές λύσεις, ειδικά σε πολύ μεγάλα και σύνθετα προβλήματα. Σε αυτές τις περιπτώσεις, πιο προηγμένες μέθοδοι είναι απαραίτητες.

Η 3-opt και η Lin-Kernighan για παράδειγμα, εξετάζουν ανταλλαγές τριών ή περισσότερων ακμών, αντίστοιχα, και επομένως είναι πιο ισχυρές από την 2-opt, αλλά και πιο υπολογιστικά απαιτητικές.

3.5 Αλγόριθμος Βελτιστοποίησης Αποικίας Μυρμηγκιών (Ant Colony Optimization)

Ο αλγόριθμος βελτιστοποίησης αποικίας μυρμηγκιών (ACO) βασίζεται στη συμπεριφορά των πραγματικών μυρμηγκιών και συγκεκριμένα στον τρόπο που αναζητούν τροφή. Τα μυρμήγκια χρησιμοποιούν μια χημική ουσία, τη φερομόνη, για να επικοινωνούν μεταξύ τους και να χαράσσουν διαδρομές. Όταν ένα μυρμήγκι βρίσκει τροφή, επιστρέφει στη φωλιά αφήνοντας

ένα ίχνος φερομόνης. Άλλα μυρμήγκια είναι πιο πιθανό να ακολουθήσουν διαδρομές με ισχυρότερο ίχνος φερομόνης, ενισχύοντας περαιτέρω τις πιο αποδοτικές διαδρομές. Αυτή η φυσική διαδικασία έδωσε την έμπνευση για την ανάπτυξη του ACO ως μεθόδου επίλυσης προβλημάτων βελτιστοποίησης.

Ο αλγόριθμος εισήχθη για πρώτη φορά από τον Ιταλό επιστήμονα Marco Dorigo. Ο Marco Dorigo, στο πλαίσιο της διδακτορικής του διατριβής στις αρχές της δεκαετίας του 1990, εμπνεύστηκε από την ικανότητα των μυρμηγκιών να βρίσκουν την συντομότερη διαδρομή μεταξύ της φωλιάς τους και μιας πηγής τροφής. Τα μυρμήγκια επικοινωνούν μεταξύ τους μέσω της εναπόθεσης μιας χημικής ουσίας που ονομάζεται φερομόνη. Όσο περισσότερα μυρμήγκια ακολουθούν μια διαδρομή, τόσο ισχυρότερο γίνεται το ίχνος της φερομόνης, καθιστώντας την διαδρομή πιο ελκυστική για τα επόμενα μυρμήγκια. Το 1992, ο Dorigo παρουσίασε τον πρώτο επίσημο αλγόριθμο ACO, γνωστό ως "Ant System" (AS). Ο AS εφαρμόστηκε στο πρόβλημα του περιοδεύοντος πωλητή (TSP), ένα κλασικό πρόβλημα συνδυαστικής βελτιστοποίησης. Ο αλγόριθμος ACO πέρασε πολλά στάδια για να φτάσει στο σημείο που τον συναντάμε σήμερα. Παρακάτω θα δούμε την ιστορική του αναδρομή μέχρι να φτάσει στις εφαρμογές που συναντάται σήμερα.

Πρώιμες Εφαρμογές και Εξέλιξη:

- Κατά τη διάρκεια της δεκαετίας του 1990, η έρευνα για τον ACO συνέχισε με την ανάπτυξη διαφόρων παραλλαγών του AS, με στόχο τη βελτίωση της απόδοσης και την αντιμετώπιση διαφόρων προβλημάτων βελτιστοποίησης.
- Προτάθηκαν διάφορες βελτιώσεις, όπως:
    - Ant Colony System (ACS): Παρουσιάστηκε από τον Dorigo και τον Gambardella το 1997. Εισήγαγε πιο εξελιγμένες τεχνικές, όπως η τοπική ενημέρωση φερομόνης και ο ψευδο-τυχαίος κανόνας μετάβασης, βελτιώνοντας σημαντικά την απόδοση σε σχέση με τον AS.
    - MAX-MIN Ant System (MMAS): Προτάθηκε από τον Stützle και τον Hoos το 2000. Εστιάζει στον έλεγχο των ορίων της φερομόνης, αποτρέποντας την πρόωρη σύγκλιση σε υπο-βέλτιστες λύσεις.

Ανάπτυξη και Εφαρμογές (2000s - Σήμερα):

- Από τις αρχές της δεκαετίας του 2000, ο ACO έχει γνωρίσει σημαντική ανάπτυξη και έχει εφαρμοστεί σε ένα ευρύ φάσμα προβλημάτων βελτιστοποίησης, πέρα από το TSP και το VRP.
- Παραδείγματα εφαρμογών περιλαμβάνουν:
    - Προβλήματα δρομολόγησης και χρονοδρομολόγησης (VRP, Job Shop Scheduling)
    - Προβλήματα βελτιστοποίησης δικτύων (routing σε τηλεπικοινωνιακά δίκτυα, βελτιστοποίηση δικτύων αισθητήρων)

- Βιοπληροφορική (ανάλυση αλληλουχιών DNA, προβλέψεις πρωτεϊνικής δομής)
- Ρομποτική (έλεγχος ρομπότ, σχεδιασμός διαδρομών)
- Εξόρυξη δεδομένων και μηχανική μάθηση

Ο ACO έχει αποδειχθεί μια ισχυρή και ευέλικτη μέθοδος για την επίλυση προβλημάτων βελτιστοποίησης. Η έμπνευσή του από τη φύση και η ικανότητά του να εξερευνά αποτελεσματικά τον χώρο λύσεων τον καθιστούν μια δημοφιλή επιλογή για την αντιμετώπιση σύνθετων προβλημάτων. Ο ACO αποτελεί μία από τις πρώτες βιολογικά εμπνευσμένες προσεγγίσεις βελτιστοποίησης. Παρουσίασε το πλεονέκτημα της συνεργατικής συμπεριφοράς, όπου πολλά τεχνητά μυρμήγκια συνεργάζονται για την εύρεση καλύτερων λύσεων και παρά την αρχική του εφαρμογή στο TSP, ο αλγόριθμος έχει προσαρμοστεί σε μεγάλο εύρος προβλημάτων.

3.5.1 Που συναντάμε τον Αλγόριθμο Βελτιστοποίησης Αποικίας Μυρμηγκιών (ACO);

1. Δρομολόγηση και Logistics

Ο ACO είναι ιδιαίτερα δημοφιλής στην επίλυση προβλημάτων δρομολόγησης, όπου απαιτείται ελαχιστοποίηση αποστάσεων ή κόστους:

- Πρόβλημα του Πλανόδιου Πωλητή (TSP): Βελτίωση δρομολογίων για ελαχιστοποίηση αποστάσεων.
- Πρόβλημα Δρομολόγησης Οχημάτων (VRP): Βελτιστοποίηση παραδόσεων με περιορισμούς χωρητικότητας και χρόνου.
- Logistics και εφοδιαστική αλυσίδα: Αποδοτική δρομολόγηση φορτηγών και παραδόσεων προϊόντων.
- Αποθήκες: Βελτιστοποίηση διαδρομών συλλογής προϊόντων μέσα σε αποθήκες.

2. Δικτυακά Συστήματα και Τηλεπικοινωνίες

Ο ACO χρησιμοποιείται για την αποδοτική διαχείριση δικτύων και ροών δεδομένων:

- Δρομολόγηση πακέτων δεδομένων: Εξασφάλιση γρήγορης και αξιόπιστης μετάδοσης δεδομένων στο Διαδίκτυο.
- Διαχείριση δικτύων: Βελτιστοποίηση ροής δεδομένων και εύρεση εναλλακτικών διαδρομών σε περιπτώσεις συμφόρησης.
- Ασύρματα δίκτυα αισθητήρων (WSN): Βελτίωση της διάρκειας ζωής του δικτύου μέσω βελτιστοποίησης δρομολόγησης.

- Δίκτυα 5G και IoT: Αποτελεσματική διαχείριση των πολλαπλών συνδέσεων και απαιτήσεων.

3. Παραγωγή και Κατασκευές

Η βελτιστοποίηση της παραγωγικής διαδικασίας και του σχεδιασμού είναι κρίσιμη:

- Προγραμματισμός παραγωγής: Βελτίωση χρονοδιαγραμμάτων εργασιών σε βιομηχανικές μονάδες.
- Κατανομή πόρων: Αποδοτική κατανομή μηχανημάτων, προσωπικού και υλικών.
- Ρομποτική: Δρομολόγηση ρομπότ σε γραμμές παραγωγής ή αποθήκες.

4. Μεταφορές

Ο ACO χρησιμοποιείται για τη βελτιστοποίηση μεταφορικών συστημάτων:

- Κυκλοφοριακή διαχείριση: Εύρεση αποδοτικών διαδρομών για μείωση της κυκλοφοριακής συμφόρησης.
- Δρομολόγηση δημόσιων συγκοινωνιών: Βελτίωση χρονοδιαγραμμάτων και διαδρομών.
- Αυτόνομα οχήματα: Βελτιστοποίηση της πλοήγησης για αποφυγή εμποδίων.

5. Επιστήμη Υπολογιστών

Ο ACO έχει εφαρμογές σε πολλά προβλήματα που σχετίζονται με την επιστήμη των υπολογιστών:

- Βελτιστοποίηση αλγορίθμων: Ρύθμιση υπερπαραμέτρων αλγορίθμων μηχανικής μάθησης.
- Πρόβλημα κατανομής πόρων: Αποδοτική κατανομή επεξεργαστικής ισχύος σε κατανεμημένα συστήματα.
- Εξόρυξη δεδομένων: Βελτιστοποίηση της εξαγωγής κανόνων συσχέτισης.

6. Ρομποτική

Στη ρομποτική, ο ACO χρησιμοποιείται για τον προγραμματισμό κινήσεων και τη συνεργασία πολλών ρομπότ:

- Δρομολόγηση ομάδων ρομπότ (swarm robotics): Συντονισμός ομάδων για αποστολές, όπως συλλογή αντικειμένων ή εξερεύνηση περιοχών.

- Αποφυγή συγκρούσεων: Βελτιστοποίηση διαδρομών για να αποφευχθούν συγκρούσεις.

7. Εφαρμογές σε Βιολογία και Ιατρική

- Γονιδιωματική: Βελτιστοποίηση στην ευθυγράμμιση αλληλουχιών DNA.
- Βελτιστοποίηση ακτινοθεραπείας: Εξασφάλιση της σωστής στόχευσης όγκων.
- Πρωτεϊνική δομή: Αναζήτηση ενεργειακά σταθερών διαμορφώσεων.

8. Παιχνίδια και Τεχνητή Νοημοσύνη

- Ανάπτυξη AI: Βελτιστοποίηση στρατηγικών για παιχνίδια.
- Πολύπλοκα περιβάλλοντα: Βελτιστοποίηση κινήσεων σε περιβάλλοντα που αλλάζουν δυναμικά.

9. Οικονομικά και Χρηματοοικονομικά

- Βελτιστοποίηση χαρτοφυλακίου: Εύρεση βέλτιστων συνδυασμών επενδύσεων.
- Διαχείριση ρίσκου: Αποδοτική κατανομή πόρων για ελαχιστοποίηση κινδύνων.

10. Πολυκριτήρια Προβλήματα

- Βελτιστοποίηση πολλαπλών στόχων: Εφαρμογή σε προβλήματα όπου απαιτείται ταυτόχρονη βελτιστοποίηση διαφορετικών κριτηρίων, π.χ. κόστους και χρόνου.

Στο σημείο αυτό θα παρουσιαστεί ο αλγόριθμος αυτός σε μορφή ψευδοκώδικα και στο επόμενο κεφάλαιο θα αναλυθεί σε βάθος σε περιβάλλον Python. Αυτό γίνεται για να γίνει πιο κατανοητός ο αλγόριθμος καθώς έχει αρκετή πληροφορία και πολυπλοκότητα. Ακολουθεί ο ψευδοκώδικας:

Αλγόριθμος Βελτιστοποίησης αποικίας μυρμηγκιών

Αρχικοποίηση
Δημιουργία του αρχικού πληθυσμού των μυρμηγκιών
Υπολογισμός της ευρετικής συνάρτησης n_{ij} *για κάθε τόξο του γραφήματος*
Υπολογισμός της αρχικής φερομόνης τ_{ij} *για κάθε τόξο του γραφήματος*

Επιλογή του μέγιστου αριθμού επαναλήψεων
Main
Do until *δεν έχει φθάσει ο μέγιστος αριθμός των επαναλήψεων*
Κάθε μυρμήγκι ξεκινάει τη δική του διαδρομή από διαφορετικό σημείο στο χώρο
For *κάθε μυρμήγκι στον πληθυσμό* ***do***
Do while *τα κριτήρια τερματισμού δεν ικανοποιούνται*
Επέλεξε το επόμενο σημείο στο γράφημα που θα πάει το μυρμήγκι βάσει της φερομόνης και της ευρετικής συνάρτησης
Υπολόγισε τη συνάρτηση ποιότητας για κάθε μυρμήγκι
Εφάρμοσε μία διαδικασία τοπικής αναζήτησης
Enddo
EndFor
Ενημέρωσε *τη φερομόνη βάσει της λύσης του βέλτιστου μυρμηγκιού στον πληθυσμό*
Enddo

Επέστρεψε *το βέλτιστο μυρμήγκι (βέλτιστη λύση).*

Κεφάλαιο 4: Παρουσίαση κώδικα

4.1.1 Επεξήγηση διπλωματικής

Σε αυτή τη διπλωματική εργασία όπως αναφέραμε θα αναπτύξουμε έναν αλγόριθμο για την επίλυση του προβλήματος δρομολόγησης οχημάτων χρησιμοποιώντας τη γλώσσα προγραμματισμού Python. Σκοπός μας είναι να βελτιστοποιήσουμε τη διαδικασία δρομολόγησης οχημάτων λαμβάνοντας υπόψη τους διάφορους περιορισμούς. Θα «τρέξουμε» διάφορα αρχεία με διαφορετικούς περιορισμούς σε κάθε ένα από αυτά και θα δούμε τι αποτελέσματα έχουμε σχετικά με τα βέλτιστα που έχουμε.

Για την πλήρη κατανόηση των αλγορίθμων και την καλύτερη παρουσίαση γράφτηκαν οι αλγόριθμοι ένας προς έναν και σταδιακά δημιουργήθηκαν όλοι οι απαιτούμενοι αλγόριθμοι για το τελικό ζητούμενο με τους απαραίτητους περιορισμούς και τις βελτιστοποιήσεις που μελετάει η διπλωματική. Έτσι παρατηρούσαμε και τη διαδικασία βήμα προς βήμα καθώς και τα αποτελέσματα πως βελτιώνονταν ανάλογα τον αλγόριθμο.

Τα αρχεία από τη βιβλιογραφία είχαν όλη τη πληροφορία που χρειάζεται ο αλγόριθμος. Ο αλγόριθμος διαβάζει κάθε φορά το αρχείο που του δηλώνουμε , δημιουργεί τους πίνακες που χρειάζεται και υλοποιεί τον αλγόριθμο. Παρουσιάζει επίσης όλα τα γραφήματα των διαδρομών που κάνει και στο τέλος αναφέρει το συνολικό κόστος των διαδρομών. Παρακάτω θα γίνει επεξήγηση του αλγορίθμου. Αρχικά της main συνάρτησης και έπειτα όλες τις υπόλοιπες που τρέχουν όταν τις καλούμε.

Τα αρχεία της βιβλιογραφίας έχουν τις εξής πληροφορίες:

Αριθμός κόμβων

Χωρητικότητα οχήματος

Μέγιστος χρόνος παραμονής του οχήματος στη διαδρομή

Χρόνος εξυπηρέτησης

Συντεταγμένες κόμβων (X,Y)

Ζήτηση του εκάστοτε κόμβου

4.1.2 Επεξήγηση συνάρτησης main

Η συνάρτηση Main όπως είπαμε περιλαμβάνει τις αρχικοποιήσεις πινάκων, την αρχικοποίηση μεταβλητών και είναι ο κώδικας που καλεί όποια συνάρτηση επιθυμούμε. Θα παρουσιαστεί σε κομμάτια ώστε να δίνεται και επεξήγηση.

```
import time
import matplotlib.pyplot as plt
plt.close('all')

from Algorithms.NearestNeighbor import *
from Algorithms.NearestNeighborWithCapacityConstrain import *
from Algorithms.NearestNeighborWithCapacityAndServiceTimeConstrains import *
from Algorithms.NearestNeighborWithCapacityAndServiceTimeConstrains2Opt import *
from Algorithms.ACO import *
```

Εικόνα 4.1

Σε αυτό το κομμάτι του κώδικα:

1. Εισαγωγή βιβλιοθηκών:
    - import time: Εισάγουμε τη βιβλιοθήκη time, η οποία παρέχει συναρτήσεις για τη μέτρηση και τον χειρισμό του χρόνου.
    - import matplotlib.pyplot as plt: Εισάγουμε το matplotlib.pyplot, που χρησιμοποιείται για τη δημιουργία γραφημάτων και οπτικοποίησης δεδομένων.
    - plt.close('all'): Κλείνουμε όλα τα ανοιχτά παράθυρα γραφημάτων, ώστε να αποφύγουμε επαναλήψεις όταν τρέχει ο κώδικας.
2. Εισαγωγή αλγορίθμων από το module Algorithms:
    - from Algorithms.NearestNeighbor import *: Εισάγουμε όλες τις συναρτήσεις από τον αλγόριθμο "Nearest Neighbor" (Πλησιέστερου Γείτονα), που είναι μια απλή προσέγγιση για προβλήματα δρομολόγησης.
    - from Algorithms.NearestNeighborWithCapacityConstrain import *: Εισάγουμε μια παραλλαγή του αλγορίθμου, που λαμβάνει υπόψη περιορισμούς χωρητικότητας.
    - from Algorithms.NearestNeighborWithCapacityAndServiceTimeConstrains import *: Μια πιο σύνθετη εκδοχή που περιλαμβάνει περιορισμούς χωρητικότητας και χρόνου εξυπηρέτησης.
    - from Algorithms.NearestNeighborWithCapacityAndServiceTimeConstrains2Opt import *: Ακόμα μία παραλλαγή που ενδέχεται να περιλαμβάνει τη βελτιστοποίηση 2-opt για τη βελτίωση των λύσεων.

- from Algorithms.ACO import *: Εισαγωγή του αλγορίθμου "Ant Colony Optimization" (ACO), που βασίζεται στη συμπεριφορά των μυρμηγκιών για εύρεση βέλτιστων διαδρομών.
-

```
if __name__ == '__main__':

    with open('files/par8.txt', 'r') as file:
        num_cities = int(file.readline().strip())
        capacity = int(file.readline().strip())
        max_route_time = int(file.readline().strip())
        service_time = int(file.readline().strip())
```

Εικόνα 4.2

Αυτό το κομμάτι κώδικα κάνει τα εξής:

1. Έλεγχος κύριας εκτέλεσης του script: Αυτό διασφαλίζει ότι ο κώδικας θα εκτελεστεί μόνο αν το αρχείο τρέχει ως κύριο πρόγραμμα και όχι αν εισαχθεί ως module σε άλλο script.

2. Άνοιγμα αρχείου και ανάγνωση παραμέτρων: Ανοίγει το αρχείο par8.txt(για παράδειγμα) από τον φάκελο files σε λειτουργία ανάγνωσης ('r').

   Χρησιμοποιεί with open(...) as file, που διασφαλίζει ότι το αρχείο θα κλείσει αυτόματα μετά την ανάγνωση.

3. Ανάγνωση και αποθήκευση δεδομένων: Διαβάζει τις πρώτες τέσσερις γραμμές του αρχείου και αποθηκεύει τις τιμές σε μεταβλητές. (Οι πρώτες τέσσερις γραμμές είναι τα δεδομένα του εκάστοτε αρχείου)
   file.readline() διαβάζει μία γραμμή κάθε φορά.
   .strip() αφαιρεί τυχόν κενά ή newline χαρακτήρες (\n).
   int(...) μετατρέπει το περιεχόμενο σε ακέραιο αριθμό.

   num_cities: Ο αριθμός των πόλεων που θα χρησιμοποιηθούν στον αλγόριθμο.
   capacity: Η χωρητικότητα του οχήματος
   max_route_time: Ο μέγιστος επιτρεπτός χρόνος για μια διαδρομή.
   service_time: Ο χρόνος εξυπηρέτησης σε κάθε πόλη.

```
coordinates = []
for _ in range(num_cities):
    _, x, y = map(float, file.readline().strip().split())
    coordinates.append([x, y])
coordinates = np.array(coordinates)
```

Εικόνα 4.3

1. Αρχικοποιούμε μια λίστα coordinates για να αποθηκεύσουμε τις συντεταγμένες των πόλεων.
2. Διαβάζουμε num_cities γραμμές από το αρχείο και αποθηκεύουμε τις τιμές x, y (οι οποίες μετατρέπονται σε float).
3. Οι συντεταγμένες προστίθενται στη λίστα και στη συνέχεια μετατρέπονται σε numpy array (np.array(coordinates)) για πιο αποδοτικούς υπολογισμούς.

```
demands = []
for _ in range(num_cities):
    _, demand = map(int, file.readline().strip().split())
    demands.append(demand)
demands = np.array(demands)
```

Εικόνα 4.4

1. Δημιουργούμε μια λίστα demands για να αποθηκεύσουμε τις απαιτήσεις κάθε πόλης.
2. Διαβάζουμε num_cities γραμμές και εξάγουμε την απαίτηση φορτίου demand (μετατρέπεται σε int).
3. Μετατρέπουμε τη λίστα σε numpy array για καλύτερη διαχείριση δεδομένων.

```
nearest_neighbor(num_cities, coordinates)
nearest_neighbor_with_capacity_constrain(num_cities, capacity, coordinates, demands)
nearest_neighbor_with_capacity_and_time(num_cities, capacity, max_route_time, service_time, coordinates,
                                        demands)
```

Εικόνα 4.5

Καλούμε διάφορες εκδόσεις του αλγορίθμου **Nearest Neighbor**:

1. **nearest_neighbor**: Τρέχει τη βασική εκδοχή του αλγορίθμου.
2. **nearest_neighbor_with_capacity_constrain**: Λαμβάνει υπόψη περιορισμούς χωρητικότητας.
3. **nearest_neighbor_with_capacity_and_time**: Προσθέτει περιορισμούς μέγιστου χρόνου διαδρομής και χρόνου εξυπηρέτησης.

```
paths, total_cost_of_entire_path = nearest_neighbor_with_capacity_time_2opt(num_cities, capacity, max_route_time, service_time, coordinates, demands)

print(len(paths))
```

Εικόνα 4.6

Χρησιμοποιούμε την εκδοχή **2-opt**, η οποία προσπαθεί να βελτιώσει τις διαδρομές μέσω ανταλλαγών ακμών για μείωση του κόστους.

Εκτυπώνουμε τον αριθμό των διαδρομών που προέκυψαν από τον βελτιστοποιημένο αλγόριθμο.

```
best_route, best_cost = aco(paths, total_cost_of_entire_path,
                            coordinates, demands, service_time, capacity, max_route_time,
                              alpha = 0.5, beta = 0,5, evaporation_rate = 0.1, num_ants = 20, max_iterations = 100)
```

Εικόνα 4.7

Εφαρμόζουμε τον αλγόριθμο **Ant Colony Optimization (ACO)** για περαιτέρω βελτιστοποίηση των διαδρομών.

Οι παράμετροι του ACO:

- alpha, beta: Ρυθμίζουν την επιρροή της φερομόνης και της τυχαίας ευρετικής πληροφορίας.
- evaporation_rate: Ρυθμός εξάτμισης της φερομόνης.
- num_ants: Αριθμός μυρμηγκιών στη διαδικασία αναζήτησης.
- max_iterations: Μέγιστος αριθμός επαναλήψεων του αλγορίθμου.

4.2 Κώδικας πλησιέστερου γείτονα

Στην ενότητα αυτή θα επεξηγηθεί η απλούστερη μορφή του αλγόριθμου ώστε να γίνει κατανοητός βήμα βήμα.

Αυτή είναι η αρχή της υλοποίησης ενός αλγορίθμου που:

- Ξεκινά από μια πόλη
- Σε κάθε βήμα επιλέγει την πλησιέστερη, μη επισκεφθείσα πόλη,
- Επαναλαμβάνει μέχρι να επισκεφθεί όλες τις πόλεις.

Στόχος: Δημιουργία μιας αρχικής διαδρομής με χαμηλό κόστος, η οποία μπορεί αργότερα να βελτιστοποιηθεί με πιο πολύπλοκους αλγορίθμους (π.χ. ACO, 2-opt).

```python
import numpy as np

from Tools.Tools import *


1 usage
def nearest_neighbor(num_cities, coordinates):

    # Initialization
    path = [0]  # Starting from the first city
    unvisited = list(range(1, num_cities))  # Cities that have to be visited
```

Εικόνα 4.8

Εισάγεται η βιβλιοθήκη NumPy για αριθμητικές πράξεις.

Εισάγονται εργαλεία από το αρχείο Tools/Tools.py.

Ορίζει μια συνάρτηση που υλοποιεί τον απλό αλγόριθμο του πλησιέστερου γείτονα (Nearest Neighbor). Παίρνει δύο παραμέτρους:

- num_cities: Πλήθος πόλεων.
- coordinates: Συντεταγμένες των πόλεων.

Η διαδρομή (path) ξεκινάει από την πρώτη πόλη (πόλη με index 0) όπου είναι η αποθήκη μας.

Η λίστα unvisited περιέχει όλους τους υπόλοιπους δείκτες πόλεων που πρέπει να επισκεφθεί το όχημα (από 1 έως num_cities-1).

```python
    # Execute the nearest neighbor algorithm
    for _ in range(1, num_cities):
        last_city = path[-1]
        distances = np.sqrt((coordinates[last_city, 0] - coordinates[unvisited, 0]) ** 2 +
                            (coordinates[last_city, 1] - coordinates[unvisited, 1]) ** 2)
        nearest_city_idx = np.argmin(distances)
        path.append(unvisited[nearest_city_idx])
        del unvisited[nearest_city_idx]
```

Εικόνα 4.9

- Επαναλαμβάνουμε για κάθε πόλη εκτός της αρχικής (συνολικά num_cities - 1 φορές).
- Παίρνει την τελευταία πόλη από την διαδρομή που έχουμε φτιάξει μέχρι στιγμής.

- Υπολογίζει την ευκλείδεια απόσταση από την τελευταία πόλη προς όλες τις μη επισκεφθείσες πόλεις.
- Βρίσκει το index της πλησιέστερης πόλης στη λίστα unvisited.
- Προσθέτει τη νέα πόλη στο τέλος της διαδρομής και τη διαγράφει από τη λίστα των μη επισκεφθέντων πόλεων.

Με λίγα λόγια ο βρόγχος αυτός ακολουθεί τα εξής βήματα:

Βρίσκει την κοντινότερη, μη επισκεφθείσα πόλη.

Την προσθέτει στη διαδρομή.

Αφαιρεί την πόλη από τη λίστα των πόλεων που απομένουν.

```
path.append(0)

# Display the path and the total cost
print('\n Nearest Neighbor')
print("Path:", path)

# Calculate the total cost
total_cost = 0
for i in range(len(path) - 1):
    total_cost += np.sqrt((coordinates[path[i], 0] - coordinates[path[i + 1], 0]) ** 2 +
                          (coordinates[path[i], 1] - coordinates[path[i + 1], 1]) ** 2)

print("Total Cost:", total_cost)

# Plotting the cities and the path
# printPathOnFigure(coordinates, path)
return total_cost
```

Εικόνα 4.10

Προσθέτει την πόλη με index 0 στο τέλος της διαδρομής, δηλαδή επιστρέφει στην αφετηρία για να κλείσει ο κύκλος.

Εκτυπώνει το όνομα του αλγορίθμου και την διαδρομή που βρήκε.

Στη συνέχεια υπολογίζει το συνολικό κόστος της διαδρομής και το εκτυπώνει.

Οι γραμμές σχολιασμού αν ενεργοποιηθούν παρουσιάζουν σε γράφημα τις διαδρομές του φορτηγού. Θα παρουσιαστούν ενδεικτικά κάποια γραφήματα στο επόμενο κεφάλαιο.

4.3 Κώδικας πλησιέστερου γείτονα με περιορισμό χωρητικότητας

```
import numpy as np

from Tools.Tools import printPathOnFigure


1 usage
def nearest_neighbor_with_capacity_constrain(num_cities, capacity, coordinates, demands):
```

Εικόνα 4.11

Εισάγεται η βιβλιοθήκη NumPy για αριθμητικές πράξεις και η συνάρτηση printPathOnFigure για την απεικόνιση της διαδρομής.

Η συνάρτηση αυτή είναι επέκταση του βασικού Nearest Neighbor, προσθέτοντας τον περιορισμό χωρητικότητας του οχήματος.

```
    # Initialization
    unvisited = list(range(1, num_cities))  # Cities that have to be visited
    paths = []
    while unvisited:
        path = [0]  # Starting from the first city (base)
        current_capacity = capacity
        while unvisited:
            last_city = path[-1]
            distances = np.sqrt((coordinates[last_city, 0] - coordinates[unvisited, 0]) ** 2 +
                                (coordinates[last_city, 1] - coordinates[unvisited, 1]) ** 2)
            nearest_city_idx = np.argmin(distances)

            # Check if adding the next city would exceed the vehicle's capacity
            if current_capacity - demands[unvisited[nearest_city_idx]] >= 0:
                current_capacity -= demands[unvisited[nearest_city_idx]]
                path.append(unvisited[nearest_city_idx])
                del unvisited[nearest_city_idx]
            else:
                break

        # End current path by returning to base
        path.append(0)
        paths.append(path)
```

Εικόνα 4.12

Δημιουργεί μια λίστα με τις πόλεις που δεν έχουν επισκεφθεί ακόμα (εκτός της βάσης, που είναι η πόλη 0). Φτιάχνει μια κενή λίστα για να αποθηκεύσει όλες τις διαδρομές (paths).

Όσο υπάρχουν αδιάβατες πόλεις, ξεκινάει νέα διαδρομή. Αρχικοποιεί την τρέχουσα διαδρομή από τη βάση. Ρυθμίζει τη χωρητικότητα στην αρχική τιμή.

Υπολογίζει την ευκλείδεια απόσταση από την τελευταία πόλη σε όλες τις αδιάβατες. Βρίσκει την πιο κοντινή πόλη.

Ελέγχει αν η χωρητικότητα του οχήματος επαρκεί για την επόμενη πόλη. Αν επαρκεί:

- Ενημερώνει τη χωρητικότητα.
- Προσθέτει την πόλη στη διαδρομή.
- Τη διαγράφει από τις αδιάβατες. Αν όχι, τερματίζει τη διαδρομή και επιστρέφει στη βάση.

```
# Display the paths and the total cost
print('\n Nearest Neighbor with Capacity Constrains')

total_cost_of_entire_path = 0
for idx, path in enumerate(paths):
    print(f"Path {idx + 1}:", path)
    total_cost = sum(np.sqrt((coordinates[path[i], 0] - coordinates[path[i + 1], 0]) ** 2 +
                             (coordinates[path[i], 1] - coordinates[path[i + 1], 1]) ** 2) for i in
                     range(len(path) - 1))
    print(f"Total Cost for Path {idx + 1}:", total_cost)
    total_cost_of_entire_path += total_cost
    printPathOnFigure(coordinates, path, title: '')

print(f"Total Cost for the Entire Path :", total_cost_of_entire_path)
```

Εικόνα 4.13

Εκτυπώνει ότι ξεκινάει ο αλγόριθμος. Αρχικοποιεί μεταβλητή για να κρατήσει το συνολικό κόστος όλων των διαδρομών.

Για κάθε διαδρομή που έχει βρει ο αλγόριθμος εκτυπώνει ποια διαδρομή είναι και ποιες πόλεις περιλαμβάνει.

Υπολογίζει την Ευκλείδεια απόσταση από κάθε πόλη στην επόμενη για ολόκληρη τη διαδρομή και τις προσθέτει.

Εκτυπώνει το κόστος για τη συγκεκριμένη διαδρομή. Προσθέτει το κόστος στη συνολική μεταβλητή. Κάνει οπτικοποίηση της διαδρομής με την printPathOnFigure.

Τέλος, εκτυπώνει το συνολικό κόστος για όλες τις διαδρομές.

4.4 Κώδικας πλησιέστερου γείτονα με περιορισμό χωρητικότητας και χρόνου

```
import numpy as np

from Tools.Tools import printPathOnFigure


1 usage
def nearest_neighbor_with_capacity_and_time(num_cities, capacity, max_route_time, service_time, coordinates, demands,
                                            speed=1.0):
```

Εικόνα 4.14

Όπως και στις δύο προηγούμενες περιπτώσεις εδώ έχουμε τις αρχικοποιήσεις μας και την εκκίνηση του αλγόριθμου, οπότε δε θα αναλυθεί κάτι περαιτέρω.

```
    # Initialization
    unvisited = list(range(1, num_cities))  # Cities that have to be visited
    paths = []
    while unvisited:
        path = [0]  # Starting from the first city (base)
        current_capacity = capacity
        current_time = 0
        while unvisited:
            last_city = path[-1]
            distances = np.sqrt((coordinates[last_city, 0] - coordinates[unvisited, 0]) ** 2 +
                                (coordinates[last_city, 1] - coordinates[unvisited, 1]) ** 2)
            travel_time = distances / speed
            nearest_city_idx = np.argmin(distances)

            # Check if adding the next city would exceed the vehicle's capacity or the time constraint
            if (current_capacity - demands[unvisited[nearest_city_idx]] >= 0 and
                    current_time + travel_time[nearest_city_idx] + service_time <= max_route_time):

                current_capacity -= demands[unvisited[nearest_city_idx]]
                current_time += travel_time[nearest_city_idx] + service_time
                path.append(unvisited[nearest_city_idx])
                del unvisited[nearest_city_idx]
            else:
                break

        # End current path by returning to base
        path.append(0)
        paths.append(path)
```

Εικόνα 4.15

Δημιουργεί μια λίστα με τις πόλεις που δεν έχουν επισκεφθεί ακόμα, εξαιρώντας την πόλη 0 (που είναι η βάση).

Κρατά όλες τις διαδρομές που θα βρεθούν στο [paths].

Αρχικοποιεί μια νέα διαδρομή ξεκινώντας από την πόλη 0 (που είναι η βάση).

Ορίζει:

current_capacity: Χωρητικότητα του οχήματος.

current_time: Χρόνος που έχει διανυθεί.

Υπολογίζει την Ευκλείδεια απόσταση από την τελευταία πόλη σε όλες τις μη επισκέψιμες πόλεις.

Μετατρέπει την απόσταση σε χρόνο ταξιδιού ορίζοντας σταθερή ταχύτητα στο πρόβλημά μας 1.0

Επιλέγει την πλησιέστερη πόλη.

Ελέγχει αν: Η πόλη μπορεί να εξυπηρετηθεί χωρίς να ξεπεραστεί η χωρητικότητα και αν ο συνολικός χρόνος δεν υπερβαίνει το μέγιστο όριο. Στη λούπα αυτή:

Αφαιρεί τη ζήτηση της πόλης από τη διαθέσιμη χωρητικότητα.

Αυξάνει τον χρόνο που πέρασε.

Προσθέτει την πόλη στη διαδρομή.

Αφαιρεί την πόλη από τη λίστα των μη επισκέψιμων.

Εάν δεν πληροί τους περιορισμούς καμία πόλη ο αλγόριθμος τερματίζει τη διαδρομή επιστρέφει στη βάση ενώ ταυτόχρονα αποθηκεύει τη διαδρομή που μόλις ολοκλήρωσε στο paths[].

Με λίγα λόγια περιληπτικά:

1. Ξεκινάει από τη βάση (πόλη 0).
2. Επιλέγει την πλησιέστερη πόλη που πληροί τους περιορισμούς.
3. Αν ξεπερνά χωρητικότητα ή μέγιστο χρόνο, τερματίζει τη διαδρομή.
4. Επιστρέφει στη βάση και αποθηκεύει τη διαδρομή.
5. Επαναλαμβάνει μέχρι να επισκεφθεί όλες τις πόλεις.

```
    # Display the paths and the total cost
    print('\n Nearest Neighbor with Capacity Service and Time Constrains')
    total_cost_of_entire_path = 0

    for idx, path in enumerate(paths):
        print(f"Path {idx + 1}:", path)
        total_cost = sum(np.sqrt((coordinates[path[i], 0] - coordinates[path[i + 1], 0]) ** 2 +
                                 (coordinates[path[i], 1] - coordinates[path[i + 1], 1]) ** 2) for i in
                         range(len(path) - 1))
        print(f"Total Cost for Path {idx + 1}:", total_cost)
        total_cost_of_entire_path += total_cost
        printPathOnFigure(coordinates, path)

    print(f"Total Cost for the Entire Path :", total_cost_of_entire_path)
```

Εικόνα 4.16

Ορίζει το αρχικό συνολικό κόστος της διαδρομής σε μηδέν

Διατρέχει κάθε διαδρομή (path) στη λίστα paths, με το idx να αντιπροσωπεύει τον αριθμό της διαδρομής.

Εκτυπώνει τη διαδρομή με αύξοντα αριθμό. Υπολογίζει τη συνολική απόσταση της διαδρομής, εφαρμόζοντας τον Ευκλείδειο τύπο απόστασης μεταξύ διαδοχικών σημείων της διαδρομής.

Εκτυπώνει το συνολικό κόστος της τρέχουσας διαδρομής.

Προσθέτει το κόστος της διαδρομής στο συνολικό κόστος όλων των διαδρομών.

Καλεί μια συνάρτηση printPathOnFigure() για την οπτικοποίηση της διαδρομής σε ένα γράφημα.

Και τέλος εκτυπώνει το συνολικό κόστος όλων των διαδρομών.

4.5 Κώδικας πλησιέστερου γείτονα με δύο περιορισμούς και τοπική αναζήτηση 2-opt

```
import numpy as np                                                    ⚠ 3

from Tools.Tools import *


1 usage
def is_route_valid(route, demands, capacity):
    """Check if the total demand of a route doesn't exceed the capacity."""
    total_demand = sum(demands[city] for city in route[1:-1])  # Exclude start and end (base)
    return total_demand <= capacity
```

Ελέγχει αν το άθροισμα των απαιτήσεων των πελατών σε μία διαδρομή δεν ξεπερνά τη χωρητικότητα του οχήματος.

- Υπολογίζει το συνολικό demand όλων των κόμβων της διαδρομής, εξαιρώντας τη βάση (αρχή και τέλος).
- Επιστρέφει True αν το total demand είναι μικρότερο ή ίσο από τη χωρητικότητα, αλλιώς False.

```
1 usage
def two_opt_swap_with_time(route, demands, coordinates, speed, capacity, max_route_time, service_time):
    best_route = route.copy()
    best_distance = compute_route_distance(route, coordinates)

    improved = True
    while improved:
        improved = False
        for i in range(1, len(route) - 2):  # Avoid start and end (base)
            for j in range(i + 1, len(route) - 1):
                # Swap edges
                new_route = route[:i] + route[i:j + 1][::-1] + route[j + 1:]

                # Check constraints
                if (is_route_valid(new_route, demands, capacity) and
                        is_route_time_valid(new_route, coordinates, speed, max_route_time, service_time)):

                    new_distance = compute_route_distance(new_route, coordinates)
                    if new_distance < best_distance:
                        best_distance = new_distance
                        best_route = new_route
                        improved = True
                        break
            if improved:
                break

    return best_route
```

Βελτιστοποιεί μία υπάρχουσα διαδρομή εφαρμόζοντας τη μέθοδο 2-opt, τηρώντας τους περιορισμούς χωρητικότητας και χρόνου.

1. Αρχικά αποθηκεύει τη διαδρομή που δόθηκε ως best_route.
2. Υπολογίζει την αρχική απόσταση της διαδρομής.
3. Επαναλαμβάνει:
    - Κάνει όλους τους δυνατούς 2-opt swaps (αντιστροφή υποδιαδρομής).
    - Για κάθε νέα διαδρομή που προκύπτει:
        - Ελέγχει αν η χωρητικότητα και ο χρόνος της διαδρομής είναι έγκυρα.

- Αν η νέα απόσταση είναι μικρότερη από την προηγούμενη, κρατάει τη νέα διαδρομή ως καλύτερη.

4. Επαναλαμβάνει μέχρι να μην υπάρχει καμία βελτίωση.

Στόχος της να μειώσει την απόσταση της διαδρομής χωρίς να παραβιάζει τους περιορισμούς.

```
3 usages
def compute_route_distance(route, coordinates):
    return sum(np.sqrt((coordinates[route[i], 0] - coordinates[route[i + 1], 0]) ** 2 +
                       (coordinates[route[i], 1] - coordinates[route[i + 1], 1]) ** 2) for i in range(len(route) - 1))


1 usage
def is_route_time_valid(route, coordinates, speed, max_route_time, service_time):
    total_time = 0
    for i in range(len(route) - 1):
        distance = np.sqrt((coordinates[route[i], 0] - coordinates[route[i + 1], 0]) ** 2 +
                           (coordinates[route[i], 1] - coordinates[route[i + 1], 1]) ** 2)
        total_time += distance / speed + service_time
    return total_time <= max_route_time
```

Στη πρώτη def υπολογίζει τη συνολική απόσταση μίας διαδρομής. Υπολογίζει την ευκλείδεια απόσταση μεταξύ κάθε ζεύγους διαδοχικών κόμβων της διαδρομής και αθροίζει όλες τις αποστάσεις.

Στη δεύτερη def:
Ελέγχει αν η συνολική διάρκεια μίας διαδρομής δεν ξεπερνά τον μέγιστο επιτρεπόμενο χρόνο.

1. Για κάθε διαδοχικό ζεύγος κόμβων υπολογίζει:

   Την απόσταση.

   Τον χρόνο μετακίνησης (απόσταση / ταχύτητα).

   Προσθέτει και τον χρόνο εξυπηρέτησης.

2. Επιστρέφει True αν ο συνολικός χρόνος ≤ max_route_time.

```
1 usage
def nearest_neighbor_with_capacity_time_2opt(num_cities, capacity, max_route_time, service_time, coordinates, demands,
                                             speed=1.0):
    # Initialization
    unvisited = list(range(1, num_cities))  # Cities that have to be visited
    paths = []
    while unvisited:
        path = [0]  # Starting from the first city (base)
        current_capacity = capacity
        current_time = 0
        while unvisited:
            last_city = path[-1]
            distances = np.sqrt((coordinates[last_city, 0] - coordinates[unvisited, 0]) ** 2 +
                                (coordinates[last_city, 1] - coordinates[unvisited, 1]) ** 2)
            travel_time = distances / speed
            nearest_city_idx = np.argmin(distances)
                    nearest_city_idx: Union[signedinteger, long] = np.argmin(distances)
            # Chec                                                    or the time constraint
            if (current_capacity - demands[unvisited[nearest_city_idx]] >= 0 and
                    current_time + travel_time[nearest_city_idx] + service_time <= max_route_time):

                current_capacity -= demands[unvisited[nearest_city_idx]]
                current_time += travel_time[nearest_city_idx] + service_time
                path.append(unvisited[nearest_city_idx])
                del unvisited[nearest_city_idx]
            else:
                break
```

```
        # End current path by returning to base
        path.append(0)

        # Apply 2-opt swapping to the path
        optimized_path = two_opt_swap_with_time(path, demands, coordinates, speed, capacity, max_route_time,
                                                service_time)
        paths.append(optimized_path)
```

Αρχικά δημιουργεί λίστα με όλες τις πόλεις που πρέπει να επισκεφθεί (εκτός της βάσης). Έπειτα δημιουργεί κενή λίστα για να αποθηκεύσει τις διαδρομές.

Δημιουργία Διαδρομής (Nearest Neighbor με Περιορισμούς)

Όσο υπάρχουν πόλεις που δεν έχουν επισκεφθεί:

- Ξεκινάει νέα διαδρομή από τη βάση.
- Επαναλαμβάνει:

    Βρίσκει την πιο κοντινή αζήτητη πόλη.

    Ελέγχει αν, προσθέτοντας την πόλη:

        Δεν παραβιάζεται η χωρητικότητα.

        Ο συνολικός χρόνος διαδρομής παραμένει εντός του ορίου.

Αν πληρούνται οι προϋποθέσεις, προσθέτει την πόλη στη διαδρομή και την αφαιρεί από τις αζήτητες.

Αν όχι, τερματίζει τη διαδρομή και επιστρέφει στη βάση.

Όταν δεν μπορεί να προστεθεί άλλη πόλη, επιστρέφει στη βάση (δηλαδή προσθέτει ξανά το 0 στο τέλος της διαδρομής).

Για κάθε διαδρομή που κατασκευάστηκε, εφαρμόζει τη συνάρτηση **two_opt_swap_with_time** για περαιτέρω μείωση της απόστασης.

```
# Display the paths and the total cost
print('\n Nearest Neighbor with Capacity and Time Constraints, and 2-opt Optimization')

total_cost_of_entire_path = 0
for idx, path in enumerate(paths):
    print(f"Path {idx + 1}:", path)
    total_cost = compute_route_distance(path, coordinates)
    total_cost_of_entire_path += total_cost
    print(f"Total Cost for Path {idx + 1}:", total_cost)
    printPathOnFigure(coordinates, path, title: 'Nearest neighbor 2opt figure')

print(f"Total Cost for the Entire Path :", total_cost_of_entire_path)
print(f"Entire Path: ", paths)

flattened_paths = []
counter = 0
for sublist in paths:
    for item in sublist:
        if item == 0 and counter == 0:
            flattened_paths.append(item)
            counter += 1
        elif item == 0 and counter == 1:
            counter = 0
        else:
            flattened_paths.append(item)
flattened_paths.append(0)
return flattened_paths, total_cost_of_entire_path
```

Υπολογίζει την απόσταση κάθε διαδρομής και εμφανίζει το συνολικό κόστος για κάθε μία και συνολικά.

Ενώνει όλες τις επιμέρους διαδρομές σε μία ενιαία λίστα, κρατώντας τις επιστροφές στη βάση.

Η ενιαία διαδρομή ξεκινά και τελειώνει στη βάση και περιλαμβάνει όλους τους κόμβους.

4.6 Κώδικας πλησιέστερου γείτονα με δύο περιορισμούς, τοπική αναζήτηση 2-opt και βελτιστοποίηση αποτελεσμάτων με τον αλγόριθμο Ant Colony

```python
# Improved ACO
import numpy as np
import random
from Tools.Tools import *


1 usage
def compute_route_distance(route, coordinates):
    return sum(np.sqrt((coordinates[route[i], 0] - coordinates[route[i + 1], 0]) ** 2 +
                       (coordinates[route[i], 1] - coordinates[route[i + 1], 1]) ** 2) for i in range(len(route) - 1))

# Συνάρτηση για τον υπολογισμό των αποστάσεων
1 usage
def calculate_distances(coordinates):
    num_cities = len(coordinates)
    distances = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                distances[i, j] = np.sqrt((coordinates[i, 0] - coordinates[j, 0]) ** 2 +
                                          (coordinates[i, 1] - coordinates[j, 1]) ** 2)
    return distances

# Συνάρτηση για τον υπολογισμό της ευρετικής πληροφορίας (ορατότητα)
1 usage
def calculate_visibility(distances):
    num_cities = len(distances)
    visibility = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                visibility[i, j] = 1 / distances[i, j]
    return visibility
```

calculate_distances(coordinates)
Υπολογίζει τον πίνακα αποστάσεων μεταξύ όλων των πόλεων (Ευκλείδεια απόσταση).

calculate_visibility(distances)
Υπολογίζει την ευρετική πληροφορία (συνάρτηση) = 1 / distance (όσο πιο κοντά, τόσο πιο ορατή η πόλη).

```
# Συνάρτηση για την αρχικοποίηση της φερομόνης
1 usage
def initialize_pheromone(num_cities, initial_pheromone):
    return np.full( shape: (num_cities, num_cities), initial_pheromone)
```

Όλος ο πίνακας αρχικοποιείται με μια τιμή initial_pheromone που υπολογίζεται ως:

initial_pheromone = num_ants / starting_solution_cost

Άρα, όσο καλύτερη είναι η αρχική λύση (Nearest Neighbor + 2-Opt), τόσο μεγαλύτερη η αρχική φερομόνη.

```
# Συνάρτηση για την επιλογή της επόμενης πόλης
1 usage
def select_next_city(current_city, pheromone, visibility, taboo_list, alpha, beta):
    num_cities = len(pheromone)
    pheromone_power = np.power(pheromone[current_city], alpha)
    visibility_power = np.power(visibility[current_city], beta)
    probabilities = pheromone_power * visibility_power
    for city in taboo_list:
        probabilities[city] = 0

    probabilities /= probabilities.sum()

    next_city = np.random.choice(np.arange(num_cities), p=probabilities)
    return next_city
```

Η συγκεκριμένη def επιλέγει την επόμενη πόλη με βάση μια πιθανότητα επηρεαζόμενη από τη φερομόνη την ευρετική πληροφορία και τη τυχαιότητα.

1. def select_next_city(current_city, pheromone, visibility, taboo_list, alpha, beta):

Η συνάρτηση δέχεται τα εξής ορίσματα:

- current_city: Η πόλη στην οποία βρίσκεται το μυρμήγκι αυτή τη στιγμή.
- pheromone: Πίνακας που περιέχει τις ποσότητες φερομόνης μεταξύ όλων των ζευγαριών πόλεων.
- visibility: Πίνακας με τις αποστάσεις ή ορατότητες από την τρέχουσα πόλη προς όλες τις υπόλοιπες πόλεις.
- taboo_list: Λίστα με τις πόλεις που έχουν ήδη επισκεφτεί (δηλαδή, αυτές που δεν μπορεί να επισκεφτεί ξανά το μυρμήγκι).
- alpha: Παράμετρος που καθορίζει την επίδραση της ποσότητας φερομόνης στην πιθανότητα επιλογής μιας πόλης.

- beta: Παράμετρος που καθορίζει την επίδραση της ορατότητας στην πιθανότητα επιλογής μιας πόλης.

2. num_cities = len(pheromone)

   Υπολογίζει τον αριθμό των πόλεων, βασιζόμενο στον αριθμό των στοιχείων στον πίνακα pheromone.

3. pheromone_power = np.power(pheromone[current_city], alpha)

   Υπολογίζει την ποσότητα φερομόνης για την τρέχουσα πόλη, επηρεασμένη από την παράμετρο alpha. Η τιμή της φερομόνης υπολογίζεται με τη χρήση της εκθετικής συνάρτησης (np.power).

4. visibility_power = np.power(visibility[current_city], beta)

   Υπολογίζει την ορατότητα για την τρέχουσα πόλη, επηρεασμένη από την παράμετρο beta. Όπως και με τη φερομόνη, η ορατότητα υπολογίζεται με εκθετική συνάρτηση.

5. probabilities = pheromone_power * visibility_power

   Υπολογίζει την πιθανότητα για κάθε πόλη, συνδυάζοντας τη φερομόνη και την ορατότητα. Πόλεις με υψηλή φερομόνη και ορατότητα θα έχουν υψηλότερες πιθανότητες επιλογής.

6. for city in taboo_list:

   Η παρακάτω εντολή θέτει τις πιθανότητες για τις πόλεις που είναι στη taboo_list (δηλαδή, τις πόλεις που το μυρμήγκι έχει ήδη επισκεφτεί) σε μηδέν, εξασφαλίζοντας ότι αυτές οι πόλεις δεν θα επιλεγούν ξανά.

7. probabilities[city] = 0

   Για κάθε πόλη στη λίστα taboo_list, η πιθανότητα της πόλης αυτής γίνεται μηδέν.

8. probabilities /= probabilities.sum()

   Κανονικοποιεί τις πιθανότητες, έτσι ώστε το άθροισμα των πιθανοτήτων να είναι 1. Αυτό εξασφαλίζει ότι πρόκειται για έγκυρες πιθανότητες.

9. next_city = np.random.choice(np.arange(num_cities), p=probabilities)

   Επιλέγει τυχαία την επόμενη πόλη, με βάση τις κανονικοποιημένες πιθανότητες. Η συνάρτηση np.random.choice επιλέγει την επόμενη πόλη από τις διαθέσιμες πόλεις, λαμβάνοντας υπόψη τις πιθανότητες που υπολογίστηκαν προηγουμένως.

10. **return next_city**

    Επιστρέφει την επόμενη πόλη που επιλέχθηκε.

```
# Συνάρτηση για τον υπολογισμό του κόστους της διαδρομής
2 usages
def calculate_route_cost(route, distances):
    total_cost = 0
    for i in range(len(route) - 1):
        total_cost += distances[route[i], route[i + 1]]
    total_cost += distances[route[-1], route[0]]  # Επιστροφή στην αποθήκη
    return total_cost
```

Εδώ η διαδικασία είναι γνώστη για τον υπολογισμό κόστους διαδρομής.

```
# Συνάρτηση για την ενημέρωση της φερομόνης
2 usages
def update_pheromone(pheromone,distances, best_route, evaporation_rate, Q= 1):
    pheromone *= (1 - evaporation_rate)
    route_cost = calculate_route_cost(best_route, distances)
    pheromone_deposit = Q / route_cost
    for i in range(len(best_route) - 1):
        pheromone[best_route[i], best_route[i + 1]] += pheromone_deposit
    pheromone[best_route[-1], best_route[0]] += pheromone_deposit  # Ενημέρωση επιστροφής
    return pheromone
```

Αυτή η συνάρτηση ενημερώνει την ποσότητα φερομόνης στον πίνακα pheromone με βάση τη διαδρομή που βρέθηκε και το κόστος αυτής. Η φερομόνη ενημερώνεται με δύο βασικούς τρόπους: μέσω της εξάτμισης και μέσω της προσθήκης νέας φερομόνης από τη βέλτιστη διαδρομή.

def update_pheromone(pheromone, distances, best_route, evaporation_rate, Q= 1):

- Η συνάρτηση δέχεται τα εξής ορίσματα:

  pheromone: Πίνακας που περιέχει τις ποσότητες φερομόνης μεταξύ όλων των ζευγαριών πόλεων.

  distances: Πίνακας με τις αποστάσεις ή τα κόστη μεταξύ των πόλεων.

  best_route: Η καλύτερη διαδρομή που βρέθηκε (μια λίστα με τις πόλεις στη σειρά που ακολουθήθηκαν).

  evaporation_rate: Ο ρυθμός εξάτμισης της φερομόνης (προσομοιώνει τη φθορά της φερομόνης με το χρόνο).

  Q: Σταθερά που καθορίζει πόση φερομόνη θα προστίθεται για κάθε βέλτιστη διαδρομή (προαιρετική παράμετρος, με default τιμή 1).

pheromone *= (1 - evaporation_rate)

- Ενημερώνει την φερομόνη μειώνοντας τις υπάρχουσες ποσότητες φερομόνης για κάθε ζεύγος πόλεων, με βάση τον ρυθμό εξάτμισης (evaporation_rate). Αυτό προσομοιώνει την εξάτμιση ή "σβήσιμο" της φερομόνης με το χρόνο.

route_cost = calculate_route_cost(best_route, distances)

- Υπολογίζει το κόστος της βέλτιστης διαδρομής, καλώντας τη συνάρτηση calculate_route_cost, η οποία υπολογίζει το άθροισμα των αποστάσεων ή του κόστους μεταξύ των πόλεων στη διαδρομή best_route.

pheromone_deposit = Q / route_cost

- Υπολογίζει την ποσότητα φερομόνης που πρέπει να προστεθεί για κάθε ζεύγος πόλεων στην καλύτερη διαδρομή. Ο υπολογισμός γίνεται με τη διαίρεση του Q (μια σταθερά) με το κόστος της διαδρομής.

for i in range(len(best_route) - 1):

- Ξεκινά έναν βρόχο για να περάσει από όλες τις πόλεις της βέλτιστης διαδρομής, εκτός από την τελευταία, για να ενημερώσει την ποσότητα φερομόνης για κάθε ζεύγος πόλεων.

pheromone[best_route[i], best_route[i + 1]] += pheromone_deposit

- Προσθέτει την ποσότητα φερομόνης (pheromone_deposit) για το ζεύγος πόλεων που είναι διαδοχικές στη διαδρομή (best_route[i] και best_route[i + 1]).

pheromone[best_route[-1], best_route[0]] += pheromone_deposit

- Ενημερώνει την φερομόνη και για την τελευταία πόλη στην διαδρομή που επιστρέφει στην αρχική πόλη (κλείνει τον κύκλο).

return pheromone

- Επιστρέφει τον ενημερωμένο πίνακα φερομόνης.

Ας δούμε τώρα την κύρια συνάρτηση ACO

```python
def aco(start_solution, starting_solution_cost, coordinates, demand, service_time, vehicle_capacity, time_limit, alpha, beta, evaporation_rate, num_ants, max_iterations ):
    num_cities = len(coordinates)
    distances = calculate_distances(coordinates)
    visibility = calculate_visibility(distances)
    initial_pheromone = num_ants / starting_solution_cost #(num_cities * np.mean(distances))
    pheromone = initialize_pheromone(num_cities, initial_pheromone)

    # Update Pheromone using the route of the solution of Nearest Neighbor 2-OPT
    pheromone = update_pheromone(pheromone,distances, start_solution, evaporation_rate, Q= 1)
    best_route = None
    best_cost = np.inf

    for iteration in range(max_iterations):
        for ant in range(num_ants):
            current_city = 0  # Αρχική πόλη (αποθήκη)
            taboo_list = [current_city]
            load = 0
            time = 0
            return_to_wh = 0

            while len(taboo_list) < num_cities + return_to_wh:
                next_city = select_next_city(current_city, pheromone, visibility, taboo_list, alpha, beta)
                if (load + demand[next_city] <= vehicle_capacity) and (time + service_time + distances[current_city, next_city] <= time_limit):
                    taboo_list.append(next_city)
                    load += demand[next_city]
                    time += service_time + distances[current_city, next_city]
                    current_city = next_city
                else:
                    taboo_list.append(0)  # Επιστροφή στην αποθήκη
                    load = 0
                    time = 0
                    current_city = 0
                    return_to_wh += 1

            route_cost = calculate_route_cost(taboo_list, distances)
            if route_cost < best_cost:
                best_cost = route_cost
                best_route = taboo_list

        pheromone = update_pheromone(pheromone,distances, best_route, evaporation_rate, Q= 1)
```

Για κάθε επανάληψη (iteration), ο αλγόριθμος εκτελεί τα εξής:

- **Για κάθε μυρμήγκι (ant)**:

  Ξεκινά από την αποθήκη (current_city = 0).

  Η λίστα taboo_list αποθηκεύει τις πόλεις που έχουν επισκεφθεί τα μυρμήγκια (ώστε να μην επισκέπτονται πόλεις περισσότερες από μία φορά).

  Ο υπολογισμός της φόρτωσης (load) και του χρόνου (time) γίνονται με βάση τα δεδομένα της ζήτησης και του χρόνου εξυπηρέτησης.

  Το μυρμήγκι επιλέγει την επόμενη πόλη με τη συνάρτηση select_next_city, ενώ ελέγχεται αν μπορεί να προχωρήσει (βάσει της χωρητικότητας του οχήματος και του χρόνου).

  Όταν το όχημα φτάσει σε πλήρη ικανότητα ή εξαντληθεί ο χρόνος, επιστρέφει στην αποθήκη (μετά το πρώτο γύρο, η αποθήκη είναι ο επόμενος προορισμός).

Στη συνέχεια αξιολογείται η λύση

route_cost = calculate_route_cost(taboo_list, distances):

Υπολογίζει το κόστος της διαδρομής που βρήκε το μυρμήγκι.

Αν η διαδρομή είναι καλύτερη από την προηγούμενη καλύτερη λύση (best_cost), τότε αποθηκεύεται αυτή η διαδρομή ως best_route και το κόστος ενημερώνεται.

Μετά από κάθε επανάληψη, η φερομόνη ενημερώνεται με την καλύτερη διαδρομή (best_route).

Η καλύτερη λύση αποθηκεύεται και ενημερώνεται μετά από κάθε επανάληψη, και ο αλγόριθμος συνεχίζει τις επαναλήψεις για τον αριθμό των μέγιστων επαναλήψεων (max_iterations).

```
# Display the paths and the total cost
print('\n ACO with Capacity and Time Constraints')

unflattenedpaths = []
temp_list = []
temp_list.append(0)
for i in best_route[1:]:
    if i == 0:
        temp_list.append(i)
        unflattenedpaths.append(temp_list)
        temp_list = []
        temp_list.append(0)
    else:
        temp_list.append(i)
temp_list.append(0)
print(unflattenedpaths)

total_cost_of_entire_path = 0
for idx, path in enumerate(unflattenedpaths):
    print(f"Path {idx + 1}:", path)
    total_cost = compute_route_distance(path, coordinates)
    total_cost_of_entire_path += total_cost
    print(f"Total Cost for Path {idx + 1}:", total_cost)
    printPathOnFigure(coordinates, path, title: 'ACO figures')

print(f"Total Cost for the Entire Path :", total_cost_of_entire_path)
print(f"Entire Path: ", unflattenedpaths)

print("Best Route:", best_route)
print("Best Cost:", best_cost)

print("\n total nodes:", len(taboo_list))
```

Στο τέλος βρίσκεται η οπτικοποίηση και η επιστροφή της καλύτερης διαδρομής.

Κεφάλαιο 5: Αποτελέσματα και Συμπεράσματα

5.1 Σύγκριση αποτελεσμάτων αλγόριθμου με την βιβλιογραφία

Παρακάτω θα δούμε τα αποτελέσματα του αλγόριθμου σε σχέση με την βιβλιογραφία. Ακολουθεί πίνακας παρουσίασης αποτελεσμάτων:

	Βέλτιστες Λύσεις	Λύσεις αλγόριθμου χωρίς βελτιστοποίηση	Λύσεις αλγορίθμου με 2-OPT	Λύσεις αλγόριθμου ACO
Par1	524.61	710.3382045035324	673.3342578184906	629.5115179676283
Par2	835.26	1000.7239858142979	985.9016568374763	1087.6209985707908
Par3	826.25	1096.0812322662512	1051.8516003479526	1227.9588834670592
Par6	**555.43**	**657.9209627806433**	**652.1888595042565**	**609.7065011060436**
Par7	909.68	1129.002765238201	1108.1894764007072	1048.6628474052982
Par8	866.76	1148.9365376058304	1138.9106963661543	1137.9187748725576
Par12	820.01	1146.3792363157668	1121.0446804787787	1120.2592484760535
Par14	866.37	1085.146697864725	1072.5629531459376	1070.3157244928377

Τα αποτελέσματα αυτά έχουν εξαχθεί με τα εξής ορίσματα:

alpha=1 ,beta=1,evaporation rate=0.1, ants=20, max_reps=500

5.2 Ψηφιακή απεικόνιση των αποτελεσμάτων

Παρακάτω θα δούμε κάποια αποτελέσματα από κάποια αρχεία που τρέξαμε:

Par6 Entire Path:

[[0, 27, 1, 8, 26, 7, 23, 43, 24, 6, 0],

[0, 46, 32, 22, 2, 29, 50, 21, 34, 9, 30, 0],

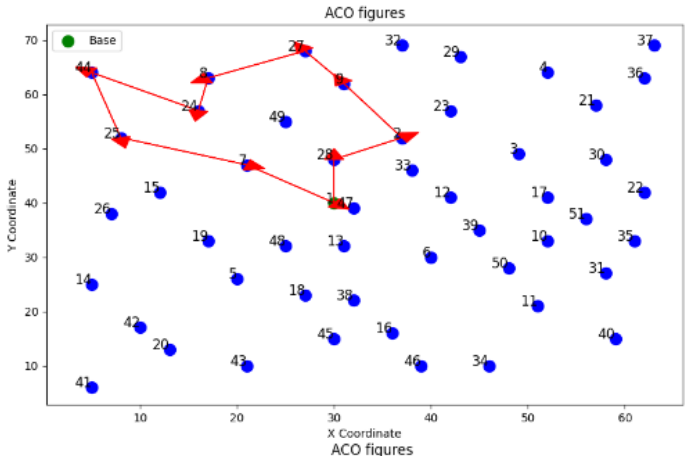
[0, 12, 37, 17, 44, 42, 19, 40, 41, 13, 4, 0],

[0, 11, 16, 20, 35, 36, 3, 28, 31, 48, 0],

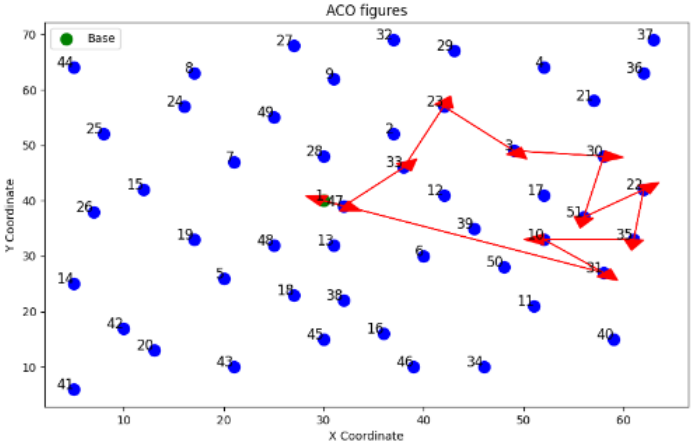
[0, 5, 38, 49, 10, 39, 33, 45, 15, 47, 0],

[0, 18, 14, 25, 0]]

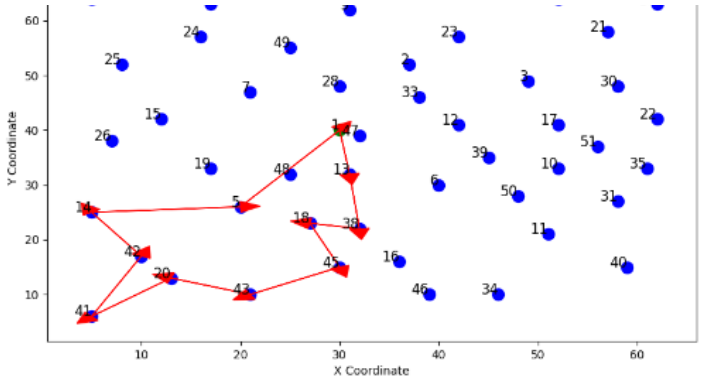
Total Cost for the Entire Path : 609.7065011060437



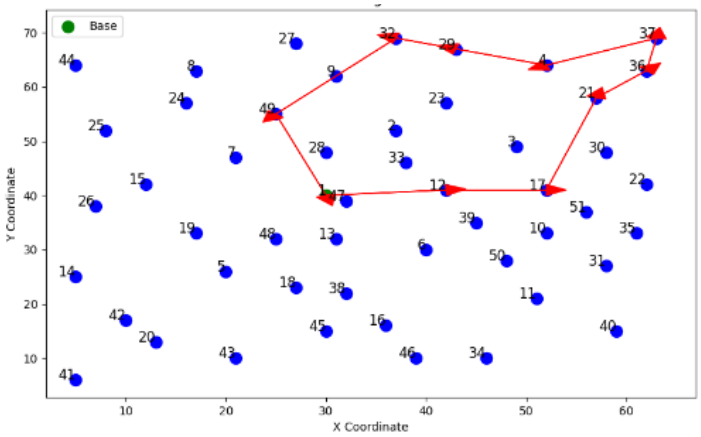
path1



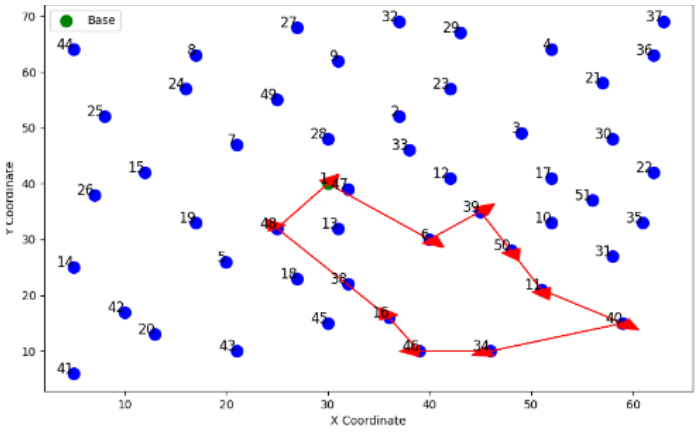
path2



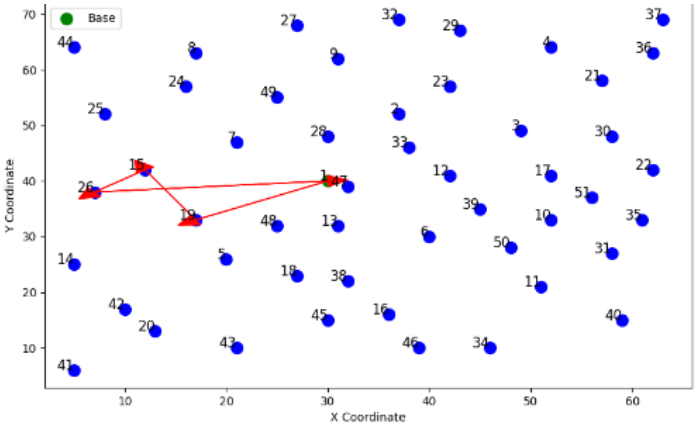
path3



path4



path5



path6

Par1 Entire Path:

[[0, 2, 20, 29, 16, 50, 39, 33, 45, 15, 37, 17, 0],

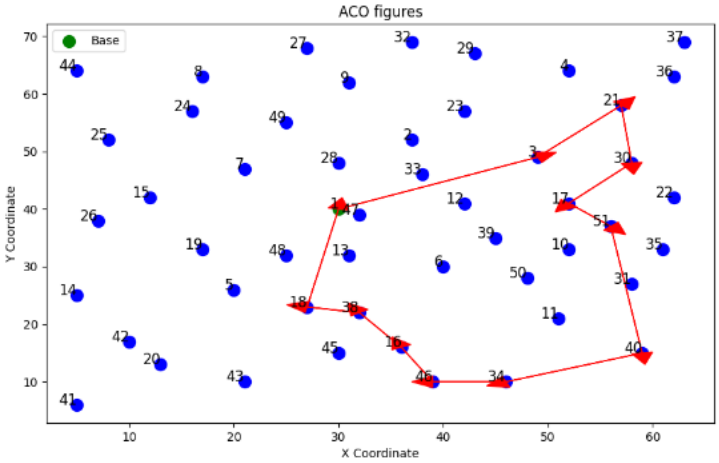
[0, 11, 38, 9, 21, 34, 30, 10, 49, 5, 32, 0],

[0, 47, 13, 40, 41, 19, 42, 44, 12, 0],

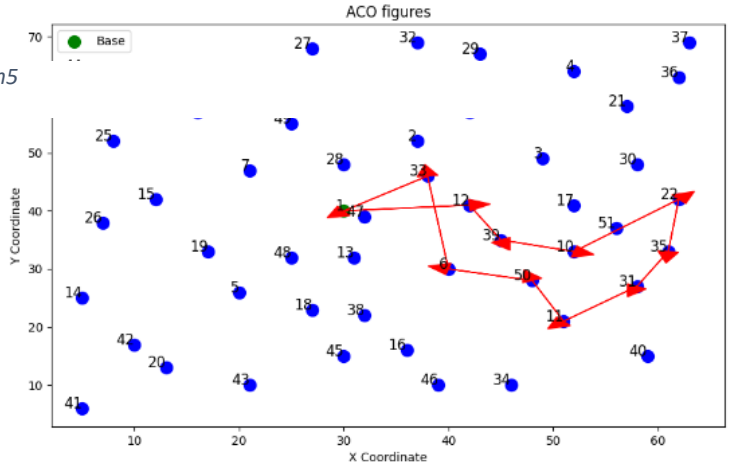
[0, 6, 48, 27, 8, 31, 36, 35, 3, 28, 26, 1, 22, 0],

[0, 46, 4, 18, 25, 14, 24, 43, 7, 23, 0]]

Total Cost for the Entire Path : 629.5115179676283

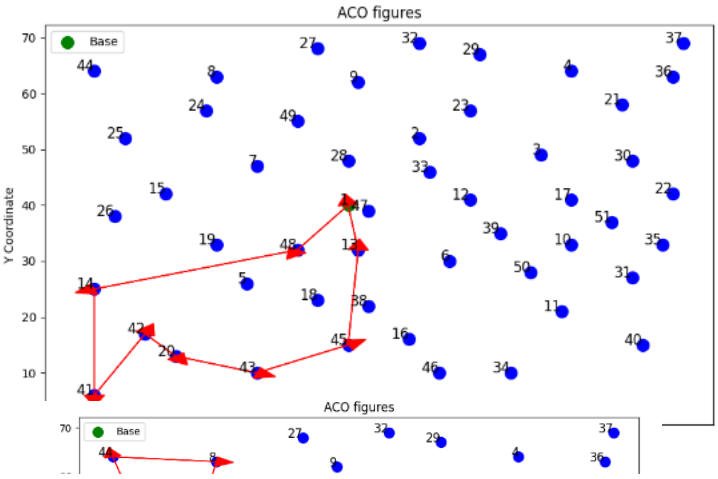


path1

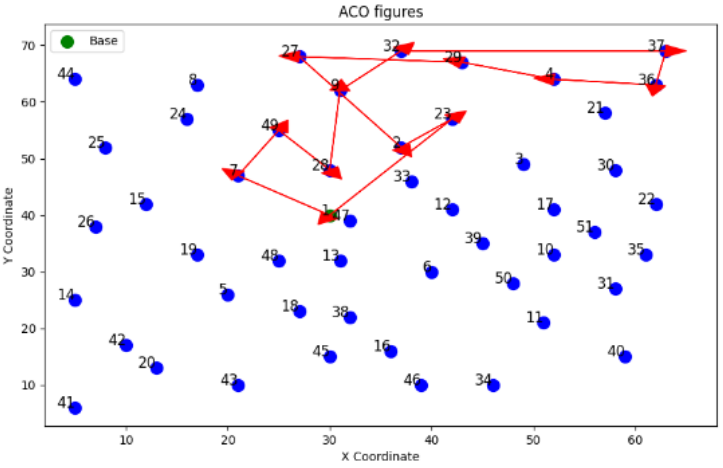


path5

path2



path3



path4

Par7 Entire Path:

[[0, 30, 21, 61, 69, 36, 47, 2, 0],

[0, 4, 45, 29, 48, 5, 74, 22, 73, 0],

[0, 16, 24, 23, 56, 41, 42, 64, 0],

[0, 75, 51, 49, 3, 44, 32, 17, 6, 0],

[0, 12, 40, 58, 10, 38, 65, 0],

[0, 26, 7, 14, 35, 8, 46, 34, 0],

[0, 67, 11, 66, 31, 72, 0],

[0, 68, 62, 1, 43, 63, 33, 28, 0],

[0, 18, 55, 25, 50, 9, 39, 0],

[0, 53, 59, 19, 54, 52, 27, 0],

[0, 13, 57, 15, 37, 20, 70, 60, 71, 0]]

Total Cost for the Entire Path : 1048.6628474052982

Par8 Entire Path:

[[0, 28, 76, 79, 81, 33, 51, 20, 71, 35, 34, 78, 29, 0],

[0, 26, 22, 74, 75, 39, 67, 25, 55, 54, 77, 3, 0],

[0, 89, 94, 96, 99, 59, 93, 5, 60, 47, 90, 62, 0],

[0, 12, 9, 65, 66, 32, 30, 70, 69, 1, 50, 0],

[0, 31, 10, 19, 11, 36, 49, 64, 63, 88, 0],

[0, 58, 2, 57, 87, 14, 43, 15, 42, 100, 16, 44, 61, 0],

[0, 53, 40, 21, 73, 72, 41, 23, 4, 56, 24, 80, 68, 0],

[0, 52, 82, 7, 48, 46, 45, 17, 84, 83, 8, 18, 0],

[0, 13, 95, 97, 37, 98, 86, 38, 91, 85, 92, 6, 27, 0]]

Total Cost for the Entire Path : 1137.9187748725576

Par14 Entire Path: [[0, 75, 7, 8, 9, 10, 6, 3, 5, 1, 2, 0],

[0, 90, 86, 87, 89, 88, 82, 83, 84, 85, 91, 0],

[0, 21, 22, 27, 25, 23, 28, 30, 24, 29, 26, 0],

[0, 69, 68, 53, 54, 60, 58, 56, 61, 62, 72, 0],

[0, 63, 65, 67, 43, 42, 41, 44, 45, 46, 47, 49, 0],

[0, 64, 79, 77, 70, 73, 71, 76, 78, 81, 80, 0],

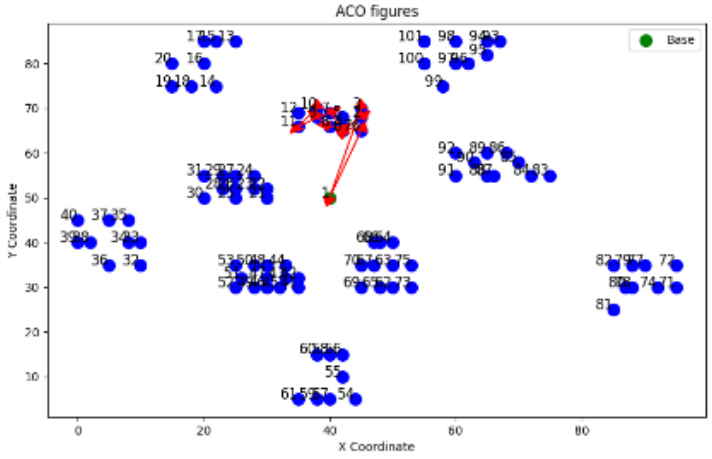
[0, 40, 48, 51, 52, 50, 55, 57, 59, 74, 66, 0],

[0, 39, 35, 37, 38, 32, 34, 36, 33, 31, 0],

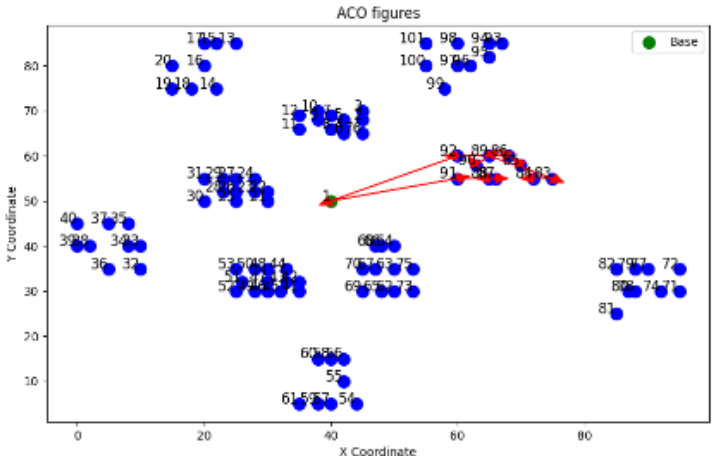
[0, 4, 99, 96, 97, 92, 93, 94, 95, 100, 98, 0],

[0, 19, 12, 14, 16, 15, 18, 13, 17, 11, 0],

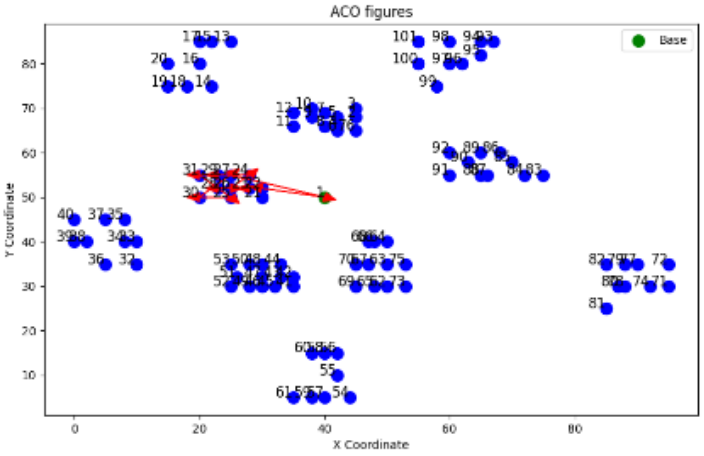
[0, 20, 0]] **Total Cost for the Entire Path** : 1070.3157244928377



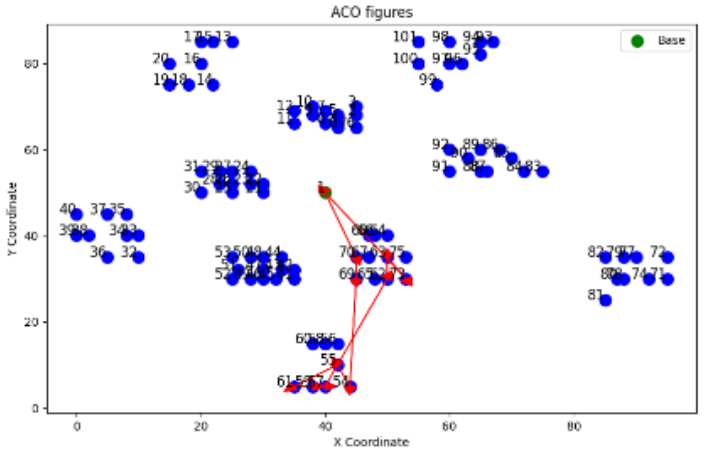
path1



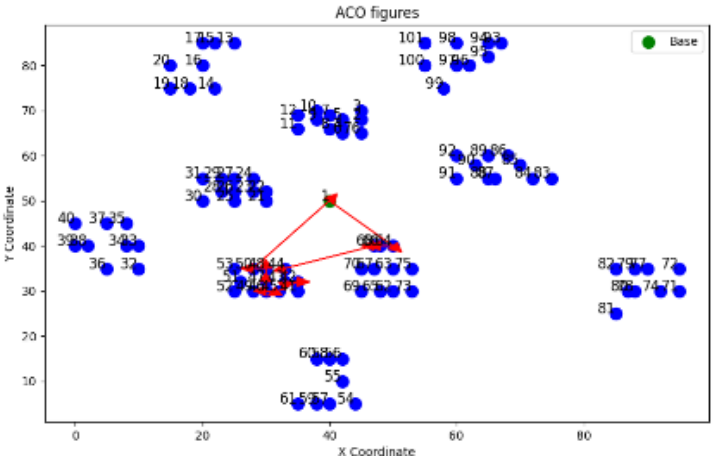
path2



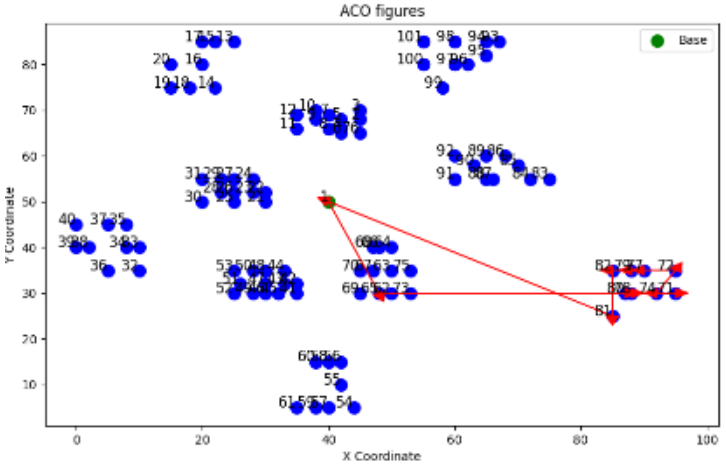
path3



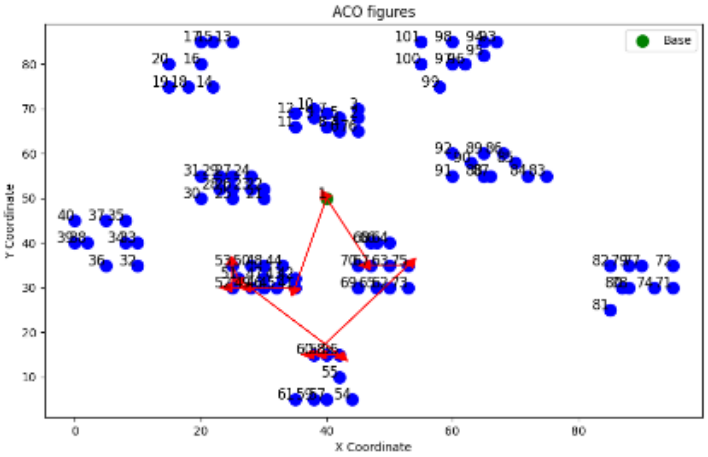
path4



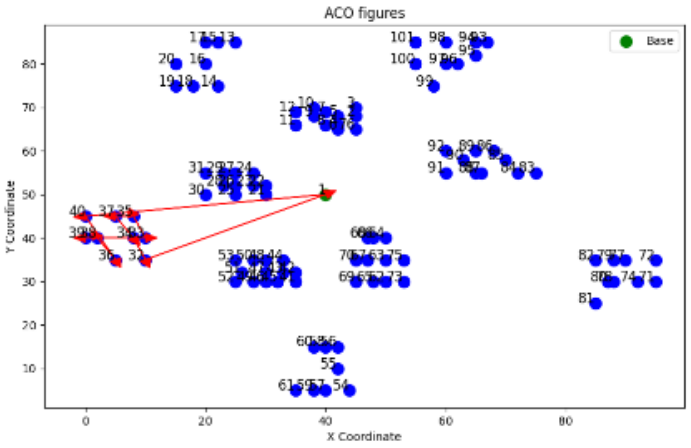
path5



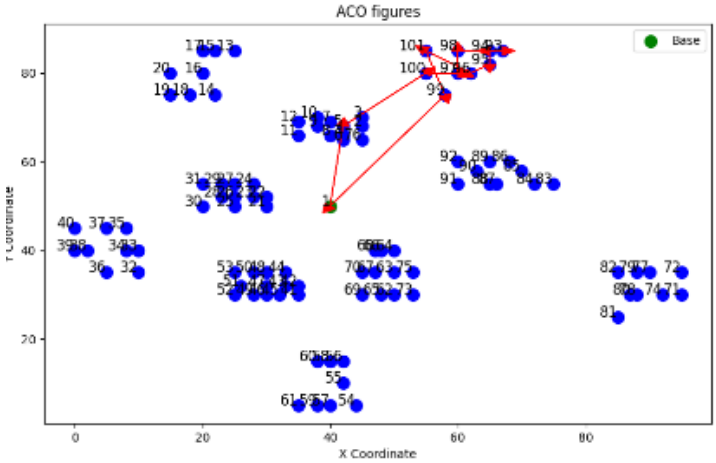
path6



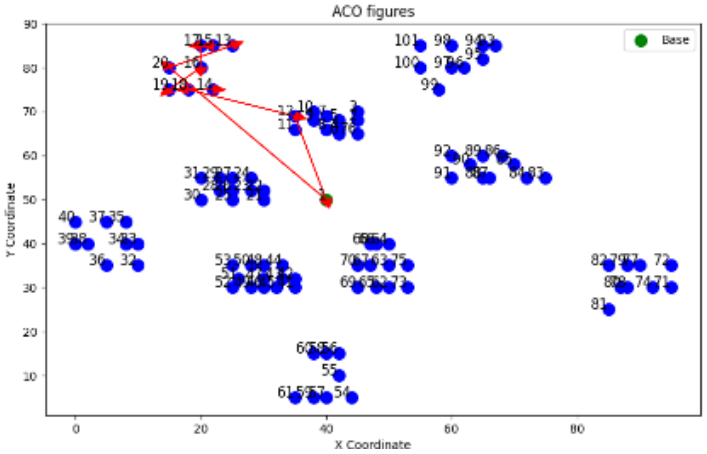
path7



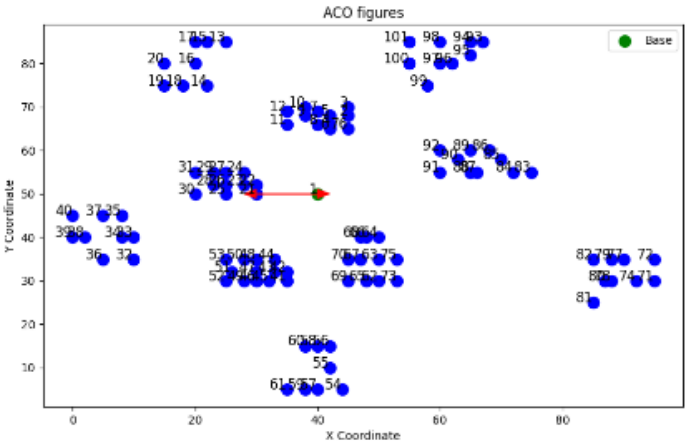
path8



path9



path10



path11

5.3 Συμπεράσματα

Ο αλγόριθμος βελτιστοποίησης αποικίας μυρμηγκιών (ACO) που αναπτύχθηκε στο πλαίσιο της διπλωματικής εργασίας αποτελεί μια αποδοτική προσέγγιση για την επίλυση του προβλήματος δρομολόγησης οχημάτων (Vehicle Routing Problem - VRP) με περιορισμούς χωρητικότητας και μέγιστου χρόνου διαδρομής. Ο αλγόριθμος ξεκινά από μια αρχική λύση που παράγεται μέσω του Nearest Neighbor με βελτιστοποίηση 2-Opt, γεγονός που μειώνει σημαντικά τον χρόνο σύγκλισης του ACO και επιτρέπει την καθοδήγηση των πρώτων επαναλήψεων προς καλύτερες λύσεις. Το μοντέλο ενσωματώνει με επιτυχία περιορισμούς που σχετίζονται με τη χωρητικότητα των οχημάτων και τον μέγιστο επιτρεπτό χρόνο διαδρομής, εξασφαλίζοντας ότι οι προτεινόμενες λύσεις είναι εφαρμόσιμες στην πράξη. Η αρχικοποίηση της φερομόνης με την αρχική λύση του Nearest Neighbor και η δυναμική ενημέρωσή της μέσα από τις καλύτερες διαδρομές συμβάλλουν στην ταχύτερη εύρεση βελτιστοποιημένων λύσεων.

Παρά τη βελτιστοποίηση, ο χρόνος εκτέλεσης του αλγορίθμου αυξάνεται σημαντικά με την αύξηση του αριθμού των πόλεων, γεγονός που καθιστά απαραίτητη τη μελλοντική βελτίωση της απόδοσής του μέσω παράλληλης επεξεργασίας ή υβριδικών τεχνικών. Η ποιότητα των λύσεων επηρεάζεται από την αρχική ρύθμιση των παραμέτρων , απαιτώντας πειραματική ανάλυση για τη βέλτιστη ρύθμιση.

Συνολικά, η υλοποίηση ACO απέδειξε ότι αποτελεί μια ισχυρή και αποτελεσματική προσέγγιση για τη δρομολόγηση οχημάτων, συνδυάζοντας ευρετικές μεθόδους με αλγοριθμικές βελτιστοποιήσεις για την παραγωγή ποιοτικών και εφικτών λύσεων.

Βιβλιογραφία

[1] **Bullnheimer, Bernd, Richard F. Hartl, and Christine Strauss.** 1999. "An Improved Ant System Algorithm for the Vehicle Routing Problem." *Annals of Operations Research* 89: 319–328.

[2] **Dorigo, Marco, and Luca Maria Gambardella.** 1997. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." *IEEE Transactions on Evolutionary Computation* 1 (1): 53–66.

[3] **Gendreau, Michel, and Jean-Yves Potvin.** 2010. "Metaheuristics in Combinatorial Optimization." In *Handbook of Metaheuristics*, edited by Michel Gendreau and Jean-Yves Potvin, 227–263. New York: Springer.

[4] **Lin, Shen.** 1965. "Computer Solutions of the Traveling Salesman Problem." *Bell System Technical Journal* 44: 2245–2269.

[5] **Toth, Paolo, and Daniele Vigo.** 2002. *The Vehicle Routing Problem*. Philadelphia: Society for Industrial and Applied Mathematics (SIAM) (pp. 1–26, 29–51).

[6] **Marinakis, Y., & Marinaki, M.** (2010). *A hybrid genetic–particle swarm optimization algorithm for the vehicle routing problem*. Expert Systems with Applications, 37(2), 1446–1455.

[7] **Vlachos, Nikolaos. 2006.** *Μετα-ευρεστικοί αλγόριθμοι βελτιστοποίησης και εφαρμογές σε προβλήματα δρομολόγησης οχημάτων. Πανεπιστήμιο Πειραιώς, Διδακτορική Διατριβή, Σελ. 90–97.*

[8] **Marinakis, Ioannis, Magdalene Marinaki, and Athanasios Mygdalas.** 2008. Σχεδιασμός και Βελτιστοποίηση της Εφοδιαστικής Αλυσίδας. Chania: Εκδόσεις Νέων Τεχνολογιών, 61–459.