ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΠΙΛΥΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ ΤΟΥ ΠΛΑΝΟΔΙΟΥ ΠΩΛΗΤΗ ΜΕ ΤΟΝ ΑΛΓΟΡΙΘΜΟ ΤΗΣ ΑΠΟΚΟΙΑΣ ΤΩΝ ΜΥΡΜΗΓΚΙΩΝ

ΚΑΛΑΝΤΖΗΣ ΜΙΧΑΗΛ-ΑΡΗΣ

Επιβλέπων: Αλέξανδρος Τζάλλας

Επίκουρος Καθηγητής

Άρτα, Οκτώβριος, 2021

ΕΠΙΛΥΣΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ ΤΟΥ ΠΛΑΝΟΔΙΟΥ ΠΩΛΗΤΗ ΜΕ ΤΟΝ ΑΛΓΟΡΙΘΜΟ ΤΗΣ ΑΠΟΚΟΙΑΣ ΤΩΝ ΜΥΡΜΗΓΚΙΩΝ

Εγκρίθηκε από τριμελή εξεταστική επιτροπή

Άρτα, 07/10/2021

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. Επιβλέπων καθηγητής

Όνομα Επίθετο, **Αλέξανδρος Τζάλλας**

τίτλος, βαθμίδα **Επίκουρος Καθηγητής**

2. Μέλος επιτροπής

Όνομα Επίθετο, **Ιωάννης Τσούλος**

τίτλος, βαθμίδα **Αναπληρωτής Καθηγητής**

3. Μέλος επιτροπής

Όνομα Επίθετο, **Νικόλαος Γιαννακέας**

τίτλος, βαθμίδα **Επίκουρος Καθηγητής**

Δήλωση μη λογοκλοπής

Δηλώνω υπεύθυνα και γνωρίζοντας τις κυρώσεις του Ν. 2121/1993 περί Πνευματικής Ιδιοκτησίας, ότι η παρούσα μεταπτυχιακή εργασία είναι εξ ολοκλήρου αποτέλεσμα δικής μου ερευνητικής εργασίας, δεν αποτελεί προϊόν αντιγραφής ούτε προέρχεται από ανάθεση σε τρίτους. Όλες οι πηγές που χρησιμοποιήθηκαν (κάθε είδους, μορφής και προέλευσης) για τη συγγραφή της περιλαμβάνονται στη βιβλιογραφία.

Καλαντζής Μιχαήλ-Άρης
Υπογραφή

ΕΥΧΑΡΙΣΤΙΕΣ

Η παρούσα πτυχιακή εργασία πραγματοποιήθηκε στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου των Ιωαννίνων κατά το έτος 2021

Η ολοκλήρωση της πτυχιακής αυτής θα ήταν αδύνατη χωρίς την υποστήριξη του καθηγητή μου κου. Τζάλλα Αλεξάνδρου. Του εκφράζω ένα βαθύ ευχαριστώ για όλη την βοήθεια του που προσέφερε.

ΠΕΡΙΛΗΨΗ

Οι αλγόριθμοι βελτιστοποίησης με βάση την λειτουργία των αποικιών των μυρμηγκιών μελετούνται από την Επιστήμη Υπολογιστών και την περιοχή της Επιχειρησιακής Έρευνας. Πρόκειται για μια πιθανολογική τεχνική για την επίλυση υπολογιστικών προβλημάτων τα οποία αφορούν στην εύρεση βέλτιστων μονοπατιών σε γράφους. Ο συγκεκριμένος αλγόριθμος ανήκει στην οικογένεια των αλγορίθμων "Αποικιών μυρμηγκιών" και στην κατηγορία μεθόδων γνωστές ως "Μέθοδοι Ευφυίας Σμήνους", αποτελεί δε μια μεταευριστική βελτιστοποίηση.

Ο αλγόριθμος αυτός προτάθηκε αρχικά μέσα από τη διδακτορική διατριβή του Marco Dorigo το 1992. Είναι ο πρώτος αλγόριθμος που αποσκοπεί στην αναζήτηση μιας βέλτιστης διαδρομής σε ένα γράφο με βάση την συμπεριφορά των μυρμηγκιών που αναζητούν μια διαδρομή από την αποικία προς την τροφή τους. Η αρχική αυτή ιδέα διαφοροποιήθηκε και επεκτάθηκε ώστε να υπηρετήσει την επίλυση μιας ευρύτερης κατηγορίας υπολογιστικών προβλημάτων, έχοντας σαν αποτέλεσμα την δημιουργία αρκετών προβλημάτων τα οποία βασίζονται στις διαφορετικές πτυχές της συμπεριφοράς των μυρμηγκιών

Λέξεις-κλειδιά: Σύστημα Αποικίας Μυρμηγκιών, Σύστημα Μυρμηγκιών, Πρόβλημα Πλανόδιου Πωλητή, Ευφυία Σμήνους

ABSTRACT

Optimization algorithms based on the function of ant colonies are studied by Computer Science and the area of Operations Research. It is a probabilistic technique for solving computational problems which involve finding optimal paths in graphs. This algorithm belongs to the family of 'Ant Colony' algorithms and to the class of methods known as 'Swarm Intelligence Methods', and is a meta-heuristic optimisation.

This algorithm was originally proposed through Marco Dorigo's PhD thesis in 1992. It is the first algorithm that aims to find an optimal path on a graph based on the behavior of ants seeking a path from the colony to their food. This original idea was diversified and extended to serve the solution of a broader class of computational problems, resulting in the creation of several problems based on different aspects of ant behavior

Keywords: Ant Colony System (ACS), Travel Saleman Problem(TSP), Ant System(AS), Swarm Intelligence(SI)

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΥΡΕΤΗΡΙΟ ΕΙΚΟΝΩΝ

ΕΙΣΑΓΩΓΗ

Σε αυτή την πτυχιακή εργασία παρουσιάζονται αλγόριθμοι που ορίζονται στην μοντελοποίηση συστημάτων που βασίζονται στην συνεργασία μεταξύ των ατόμων από ένα πληθυσμό, στην περίπτωση που τα άτομα του πληθυσμού δεν έχουν πλήρη γνώση του προς την επίλυση προβλήματος. Το κάθε μέλος του πληθυσμού ενεργεί με βάση τις πληροφορίες που έχει και την συνέχεια μοιράζεται την γνώση που απέκτησε με άλλα μέλη του πληθυσμού. Συνήθως ο πληθυσμός ονομάζεται σμήνος και όλοι αυτοί οι μέθοδοι ονομάζονται αλγόριθμοι νοημοσύνης σμήνους .Σε αυτή την εργασία θα δούμε τον αλγόριθμο βελτιστοποίησης των μυρμηγκιών. Ο αλγόριθμος αυτός είναι κατάλληλος για επίλυση προβλημάτων συνδυαστικής βελτιστοποίησης.

Η αυτό-οργάνωση είναι ένα φαινόμενο που περιγράφεται σε πολλά πεδία και είναι μια διαδικασία όπου μια ολοκληρωμένη δραστηριότητα ενός συστήματος αναδεικνύεται από ένα σύνολο από αλληλεπιδράσεις μεταξύ διαφορετικών παραγόντων του συστήματος. Έτσι αυτό πρέπει να κατανοηθεί είναι το πώς διαφορετικοί παράγοντες ενός συστήματος αλληλεπιδρούν μεταξύ τους για να πράξουν ένα ποιο σύνθετο σύστημα.

1. ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ ΠΛΑΝΟΔΙΟΥ ΠΩΛΗΤΗ

Ένα από τα πιο γνωστά προβλήματα πάνω στην συνδυαστική βελτιστοποίηση αλλά και στην επιστήμη των υπολογιστών είναι το Πρόβλημα του Πλανόδιου Πωλητή. Η απλότητα και ταυτόχρονα η δυσκολία της επίλυσης του το έχουν κάνει ένα ελκυστικό πρόβλημα που έχει οδηγήσει τους ερευνητές να ασχοληθούν με την επίλυση του κατά τις τελευταίες δεκαετίες.

Στο οποίο τα δεδομένα είναι ένα σύνολο από πόλεις και από αποστάσεις μεταξύ κάθε δεδομένου ζεύγους πόλεων και το ζητούμενο είναι η εύρεση εκείνου του μονοπατιού που επισκέπτεται κάθε πόλη μια φορά και επιστρέφει στην αρχική πόλη,με τη συνολική απόσταση που διανύθηκε να είναι η ελάχιστη.

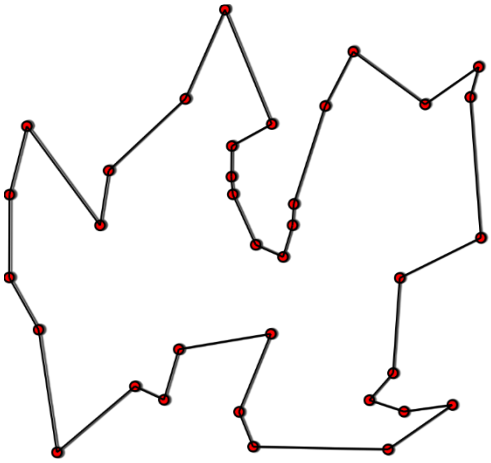

Figure 1.1 Μη Κατευθυνόμενο γράφημα

Το Πρόβλημα του Πλανόδιου πωλητή από την εποχή της καθιέρωσης του την δεκαετία του 1930 είναι σημαντικό και παράλληλα δύσκολο (NP-hard) και αντιπροσωπευτικό(NP-completed). Τα προβλήματα NP–hard (Non–deterministic Polynominal hard) είναι δυσεπίλυτα υπολογιστικά προβλήματα υπό την έννοια ότι δεν έχουν βρεθεί γνωστές λύσεις σε πολυωνυμικό χρόνο, σε αντίθεση με την οικογένεια των προβλημάτων P που μπορούν να λυθούν σε ακριβή πολυωνυμικό χρόνο. Ως υποσύνολο των προβλημάτων NP, τα προβλήματα NP–complete είναι εκείνα των οποίων οι λύσεις είναι επαρκείς για να αντιμετωπίσουν οποιοδήποτε άλλο πρόβλημα NP σε πολυωνυμικό χρόνο. Έτσι αν μπορούσαν να βρεθούν αποτελεσματικές λύσεις για κάθε πρόβλημα NP–complete, συμπεριλαμβανομένου και του Προβλήματος του Πλανόδιου Πωλητή (TSP), τότε θα είμαστε σε θέση να επιλύσουμε κατηγορηματικά το αναπάντητο ερώτημα, αν P = NP.

1.1. Ιστορική αναδρομή

Τα ιστορικά στοιχεία για την πρώτη αναφορά στο συγκεκριμένο πρόβλημα δεν

είναι σαφή. Υπάρχουν ενδείξεις ότι οι πρώτες αναφορές στο Πρόβλημα του Πλανόδιου Πωλητή έγιναν τον 18ο αιώνα. Στην ουσία, η πρώτη αναφορά περιέχει παραδείγματα περιηγήσεων, τα οποία δεν θεμελιώθηκαν μαθηματικά και δεν είχαν κανέναν επιστημονικό υπόβαθρο. Συγκεκριμένα, το 1832 τυπώθηκε ένα εγχειρίδιο το οποίο αναφέρει το πρόβλημα TSP, περιλαμβάνονται και κάποια παραδείγματα διαδρομών μεταξύ Γερμανίας και Ελβετίας.

Το πρόβλημα του Πλανόδιου Πωλητή(TSP) η πρώτη σαφής αναφορά έγινε το 1800 από τον Ιρλανδό μαθηματικό W.R. Hamilton και από τον Βρετανό μαθηματικό Thomas Kirkman. Ο Hamilton είχε δημιουργήσει ένα παζλ που βασίστηκε στην εύρεση ενός κύκλου Χάμιλτον, το

οποίο έχει αρκετές ομοιότητες με το πρόβλημα του πλανόδιου πωλητή (TSP).Η γενική μορφή του TSP φαίνεται να μελετήθηκε για πρώτη φορά από μαθηματικούς κατά τη διάρκεια της δεκαετίας του 1930 στη Βιέννη και στο Harvard, ιδίως από τον Karl Menger,ο οποίος παρατήρησε τη μη βελτιστότητα της ευρετικής μεθόδου του πλησιέστερου γείτονα (Nearest Neighbor Heuristic).

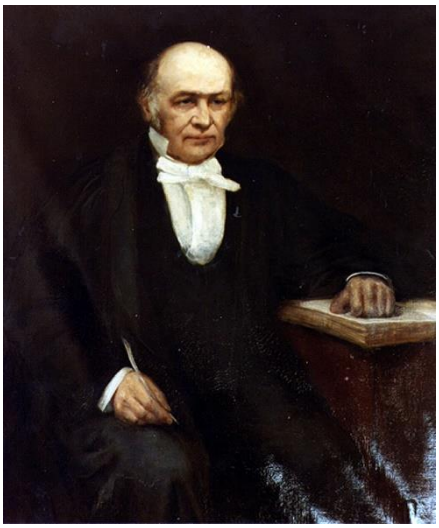

Figure 1.1.1 Γουίλιαμ Ρόουαν Χάμιλτον

Θεωρήθηκε για πρώτη φορά μαθηματικά τη δεκαετία του 1930 από τον Merrill M. Flood, ο οποίος έψαχνε να λύσει ένα πρόβλημα δρομολόγησης σχολικών λεωφορείων. Hassler Whitney στο Πανεπιστήμιο του Princeton προκάλεσε ενδιαφέρον για το πρόβλημα, το οποίο ονόμασε «πρόβλημα 48 κρατών» Η πρώτη δημοσίευση που χρησιμοποιεί τη φράση "Πρόβλημα Πλανόδιου Πωλητή" ήταν στην έκθεση RAND Corporation του 1949 της Julia Robinson, "On the Hamiltonian game (a traveling salesman problem)"

Στις δεκαετίες του 1950 και του 1960, το πρόβλημα έγινε όλο και πιο δημοφιλές σε επιστημονικούς κύκλους της Ευρώπης και των ΗΠΑ, αφού η RAND Corporation στη Santa Monica προσέφερε βραβεία για την επίλυση του προβλήματος. Αξιοσημείωτες συνεισφορές έγιναν από τους George Dantzig, Delbert Ray Fulkerson και Selmer M. Johnson από την RAND Corporation, οι οποίοι εξέφρασαν το αντιμετώπισαν το πρόβλημα ως πρόβλημα ακέραιου γραμμικού προγραμματισμού για την επίλυση ενός προβλήματος που περιλάμβανε τη σύνδεση 49 πόλεων. Αργότερα, τις δεκαετίες του 1970 και 1980, αρκετοί επιστήμονες χρησιμοποιώντας τεχνικές, όπως branch–and–bound και cutting–planes, κατάφεραν να δώσουν λύση σε προβλήματα που είχαν ως κόμβους αρκετές χιλιάδες πόλεις (Grötschel, Padberg και Rinaldi).

Οι Dantzig, Fulkerson και Johnson, ωστόσο, εικάζουν ότι, δεδομένης μιας σχεδόν βέλτιστης λύσης, ενδέχεται να είμαστε σε θέση να βρούμε τη βέλτιστη ή να αποδείξουμε τη βέλτιστη προσθήκη ενός μικρού αριθμού επιπλέον ανισοτήτων(περικοπές). Χρησιμοποίησαν αυτήν την ιδέα για να λύσουν το αρχικό πρόβλημα των 49 πόλεων χρησιμοποιώντας ένα μοντέλο χορδών(String Model). Βρήκαν ότι χρειάζονταν μόνο 26 περικοπές για να βρουν λύση στο πρόβλημα των 49 πόλεών τους. Ενώ αυτό το έγγραφο δεν έδωσε μια αλγοριθμική προσέγγιση σε προβλήματα TSP, οι ιδέες που περιέχονται σε αυτό ήταν απαραίτητες για τη δημιουργία μετέπειτα ακριβών μεθόδων λύσης για το TSP, αν και θα χρειαστούν 15 χρόνια για να βρεθεί μια αλγοριθμική προσέγγιση στη δημιουργία αυτών των περικοπών. Εκτός από τις μεθόδους κοπής επιπέδων, οι Dantzig, Fulkerson και Johnson χρησιμοποίησαν αλγόριθμους κλάδου και δέσμευσης ίσως για πρώτη φορά.

Το 1972 ο Richard M. Karp έδειξε ότι το πρόβλημα του Χαμιλτονιανού κύκλου ήταν NP–complete, πράγμα που συνεπάγεται την NP–δυσεπιλυσιμότητα του TSP. Το γεγονός αυτό παρείχε μια επιστημονική εξήγηση για την υπολογιστική δυσκολία εύρεσης βέλτιστων διαδρομών.

Το 1959, Jillian Beardwood, J.H. Ο Halton και ο John Hammersley δημοσίευσαν ένα άρθρο με τίτλο "The Shortest Path Through Many Points" στο περιοδικό της Φιλοσοφικής Εταιρείας του Cambridge.Το θεώρημα Beardwood – Halton – Hammersley παρέχει μια πρακτική λύση στο Πρόβλημα του Πλανόδιου Πωλητή(TSP). Οι συγγραφείς προέβησαν σε μια ασυμπτωτική φόρμουλα για να προσδιορίσουν το μήκος της συντομότερης διαδρομής για έναν πωλητή που ξεκινά από ένα σπίτι ή ένα γραφείο και επισκέπτεται έναν καθορισμένο αριθμό τοποθεσιών πριν επιστρέψει στην αρχή.

Τις επόμενες δεκαετίες, το πρόβλημα μελετήθηκε από πολλούς ερευνητές από μαθηματικά, επιστήμη υπολογιστών, χημεία, φυσική και άλλες επιστήμες. Στη δεκαετία του 1960, ωστόσο, δημιουργήθηκε μια νέα προσέγγιση, ότι αντί να αναζητούν βέλτιστες λύσεις, θα παρήγαγε μια λύση της οποίας το μήκος αποδεικνύεται αποδεδειγμένα από ένα πολλαπλάσιο του βέλτιστου μήκους, και με αυτόν τον τρόπο δημιουργούν χαμηλότερα όρια για το πρόβλημα. Αυτά μπορούν στη συνέχεια να χρησιμοποιηθούν στην μέθοδο branch and bound.

Το 1976, ο Christofides και ο Serdyukov ανεξάρτητα ο ένας από τον άλλον σημείωσαν μεγάλη πρόοδο προς αυτήν την κατεύθυνση: Ο αλγόριθμος Christofides-Serdyukov αποδίδει μια λύση που, στη χειρότερη περίπτωση, είναι το πολύ 1,5 φορές μεγαλύτερη από τη βέλτιστη λύση. Καθώς ο αλγόριθμος ήταν τόσο απλός και γρήγορος, πολλοί ήλπιζαν ότι θα παραχωρούσε σε μια σχεδόν βέλτιστη μέθοδο λύσης.

Ο Richard M. Karp έδειξε το 1972 ότι το πρόβλημα του Hamiltonian cycle ήταν NP-complete, πράγμα που συνεπάγεται το NP-hardness του TSP. Αυτή είναι η μαθηματική εξήγηση για την φαινομενική υπολογιστική δυσκολία εύρεσης βέλτιστων διαδρομών.

Μεγάλη πρόοδος σημειώθηκε στα τέλη της δεκαετίας του 1970 και του 1980, όταν οι Grötschel, Padberg, Rinaldi και άλλοι κατάφεραν να λύσουν με ακρίβεια περιπτώσεις με έως και 2.322 πόλεις, χρησιμοποιώντας "cut-planes" και "branch and bound".

Την δεκαετία του 1990 οι David Applegate, Robert E. Bixby, Vasek Chvatal και William J. Cook δημιούργησαν ένα λογισμικό επίλυσης του συγκεκριμένου προβλήματος, το οποίο ονομάσθηκε «Concorde TSP Problem». Το 1991, ο Gerhard Reinelt δημοσίευσε το TSPLIB, μια συλλογή προβλημάτων αναφοράς μεταβλητής δυσκολίας, η οποία έχει χρησιμοποιηθεί από πολλές ερευνητικές ομάδες για τη σύγκριση αποτελεσμάτων.

Τέλος, το 2004, οι ίδιοι επιστήμονες, Applegate, Bixby, Chvatal και Cook, μαζί με τον Keld Helsgaun κατάφεραν να λύσουν ένα πρόβλημα με 24978 κόμβους – πόλεις της Σουηδίας. Λίγο αργότερα, το 2006 ο Cook και η επιστημονική του ομάδα επίλυσαν ένα πρόβλημα 85900 κόμβων, οι οποίοι αναπαριστούσαν τη σύνδεση των τρανζίστορ σε ένα μικροεπεξεργαστή.

1.2. Μαθηματική μοντελοποίηση του TSP

Μία από τις πολλές τροποποιήσεις του προβλήματος του πλανόδιου πωλητή είναι η επακόλουθη:

Ένα πλήρες, σταθμισμένο, μη προσανατολισμένο γράφημα (G) το οποίο καθορίζεται από τα ζεύγη (N,d).

- N: σύνολο των κόμβων.
- d: η συνάρτηση απόστασης μεταξύ των κορυφών (κόμβων)

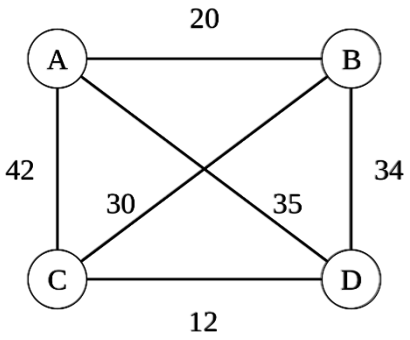

Figure 1.2.1 Μη Κατευθυνόμενο γράφημα με βάρη

τα παραπάνω ικανοποιούν τις παραπάνω σχέσεις:

1. d(i,j) = d(j,i) για όλα τα i και j που ανήκουν στο N.

2. d(i,j) ≥ 0 για όλα τα i και j που ανήκουν στο N.

3. d(i,j) + d(j,k) ≥ d(i,k) για όλα τα i και j που ανήκουν στο N (τριγωνική ανισότητα).

- Ο αριθμός d(i,j) ονομάζεται μήκος ή βάρος της απόστασης (i,j).

Αν ορίσουμε τις μεταβλητές:

$$xij = \{^{1,\Lambda\ \varepsilon\acute{\alpha}\nu\tau o\acute{\alpha}\tau o\mu o\kappa\acute{\alpha}\nu\varepsilon\iota\chi\rho\acute{\eta}\sigma\eta\tau o\upsilon\sigma\upsilon\nu\delta\acute{\varepsilon}\sigma\mu o\upsilon\{i,j\},}_{0,\Lambda\ \alpha\lambda\lambda\iota\acute{\omega}\varsigma,\forall i,\forall j,i\neq j}$$

Οπότε έχουμε:

min

$$\sum_{i=1}^{n}\sum_{j=1\square_{i\neq j}}^{n} c_{ij}x_{ij} \quad (2.1)$$

υπό

$$\sum_{j=1}^{n} x_{ij} = 2, i = 1,\ldots,n (2.2)$$

$$\sum_{i\in C}\sum_{j\in C} x_{ij} \geq 1, \nabla \subset C\{1, \ldots . n\}, C \neq 0 (2.3)$$

$$x_{ij} \in \{0,1\}, \forall i, \forall ji \neq j. (2.4)$$

Οι περιορισμοί (2.2) επιβάλλουν την λύση να έχει δυο συνδυασμούς σε κάθε κόμβο, έτσι ώστε το άτομο να εισέλθει κατά μήκος του ενός και να εξέλθει κατά μήκος του άλλου, ενώ οι περιορισμοί (2.3) αποβλέπουν στην εξάλειψη κυκλικών διαδρομών (υπο-κύκλων) που δεν διέρχονται από όλους τους κόμβους απαιτώντας από κάθε πιθανό υποκύκλο, που αντιπροσωπεύεται από ένα κατάλληλο μη-κενό υποσύνολο C των κόμβων, να διαθέτει στην λύση τουλάχιστον ένα σύνδεσμο που οδηγεί στο συμπληρωματικό του υποσύνολο C = {1,....,n}/C.

Όταν η φορά του ατόμου είναι σημαντική, εάν δηλαδή κινείται από το i στο j ή από το j στο i, τότε στο γράφημα έχουμε τόξα που αντιστοιχούν στα διατεταγμένα ζεύγη (i,j) και (j,i) και αντίστοιχα (i,j) ≠ (j,i). Οπότε και η συνθήκη συμμετρίας δεν ισχύει και έχουμε $c_{ij} \neq c_{ji}$τουλάχιστον για κάποια i και j. Το μαθηματικό πρωτότυπο σε αυτή την περίπτωση διαμορφώνεται όπως έχουμε ορίσει προηγουμένους στις (2.1) και (2.4), έχοντας αντικαταστήσει τον περιορισμό με δυο νέους περιορισμούς:

$$\sum_{j=1}^{n} x_{ij} = 1, i = 1, \ldots . n (2.5)$$

$$\sum_{j=1}^{n} x_{ij} = 1, j = 1, \ldots . n (2.6)$$

όπου ο (2.5) εγγυάται ότι το άτομο εξέρχεται από τον κόμβο i χρησιμοποιώντας ένα ακριβώς τόξο προς κάποιον άλλο κόμβο j. Ενώ ο δεύτερος εγγυάται ότι το άτομο εισέρχεται στον κόμβο j χρησιμοποιώντας ένα ακριβώς τόξο από κάποιον άλλο κόμβο i.

2. ΜΕΘΕΥΡΕΤΙΚΟΙ ΑΛΓΟΡΙΘΜΟΙ

Κάθε μεθευρετικός αλγόριθμος αποτελείται από δύο χαρακτηριστικά:

- επίταση (intensification) : είναι η εστίαση στην τοπική αναζήτηση εκμεταλλευόμενοι την πληροφορία ότι βρίσκουμε μια καλή λύση σε αυτήν την περιοχή.

- διαφοροποίηση (diversification) ή εκμετάλλευση (exploitation) : είναι η εξερεύνηση του χώρου των λύσεων προς αναζήτηση νέας βέλτιστης λύσης, η οποία και εξασφαλίζει ότι ο αλγόριθμος δεν θα παγιδευτεί σε ένα τοπικό ελάχιστο

Με τη σωστή αναλογία των δύο παραπάνω στοιχείων του αλγορίθμου εξασφαλίζεται η εύρεση μιας πολύ καλής λύσης. Όλοι οι αλγόριθμοι της οικογένειας των μεθευρετικών χρησιμοποιούν μεθόδους προκειμένου να συνεχίσουν να ψάχνουν για λύσεις κοντά σε προηγούμενες (exploitation) ή να μη δώσουν μεγάλη βαρύτητα σε ήδη δοκιμασμένες λύσεις (exploration).

Ένας από τους πολλούς τρόπους που μπορούν να κατηγοριοποιηθούν οι μεθευρετικοί αλγόριθμοι, είναι οι αλγόριθμοι που χρησιμοποιούν πληθυσμούς (population-based) και εκείνοι που χρησιμοποιούν την τροχιά (trajectory based). Ένα από τα κυριότερα παραδείγματα του πρώτου είδους είναι η μέθοδος βελτιστοποίησης σμήνους σωματιδίων (particle swarm optimization) χρησιμοποιώντας έναν πληθυσμό υποψηφίων λύσεων που τον μετακινεί στο χώρο αναζήτησης. Όλοι οι αλγόριθμοι αυτής της ομάδας διαθέτουν εργαλεία που επιτρέπουν την ανταλλαγή πληροφοριών μεταξύ των ατόμων του πληθυσμού προκειμένου να αποφύγουν κακές λύσεις. Αντίθετα ένας αλγόριθμος βασισμένος στην τροχιά, είναι η προσημειωμένη ανόπτηση (simulated annealing), που χρησιμοποιεί ένα στοιχείο –υποψήφια λύση – την οποία μετακινεί τμηματικά στον χώρο των λύσεων. Μια μετακίνηση σε καλύτερη λύση γίνεται πάντα αποδεκτή, ενώ μια μέτρια μετακίνηση γίνεται αποδεκτή με κάποια πιθανότητα.

2.1. Αυτές είναι οι ιδιότητες που χαρακτηρίζουν έναν μεθευρετικό αλγόριθμο

- Οι μεθευρετικοί αλγόριθμοι χρησιμοποιούν ως στρατηγική να καθοδηγούν τη διαδικασία αναζήτησης.
- Στόχος είναι να εξερευνήσουν αποτελεσματικά τον χώρο αναζήτησης για να βρεθούν οσο ποιό κοντά στην βέλτιστη λύση.
- Οι τεχνικές που αποτελούν τους μεθευρετικούς αλγόριθμους μπορεί να είναι από απλές τοπικές διαδικασίες αναζήτησης έως πολύπλοκες διαδικασίες μάθησης.
- Οι μεθευρετικοί αλγόριθμοι είναι κατά προσέγγιση και συνήθως μη-ντετερμινιστικοί.
- Οι μεθευρετικοί αλγόριθμοι δεν αφορούν συγκεκριμένα προβλήματα.

2.2. Local search vs. global search

Μια προσέγγιση για τον χαρακτηρισμό αυτού του τύπου της αναζήτησης. Ένας τύπος αναζήτησης είναι η βελτίωση των απλών τοπικών αλγορίθμων αναζήτησης. Ένας γνωστός αλγόριθμος τοπικής αναζήτησης (ocal search) είναι η μέθοδος αναρρίχησης (hill climbing) που χρησιμοποιείται για την εύρεση τοπικών βέλτιστων τιμών. Ωστόσο, αυτός ο αλγόριθμος δεν εγγυάται την εύρεση βέλτιστων λύσεων (global).

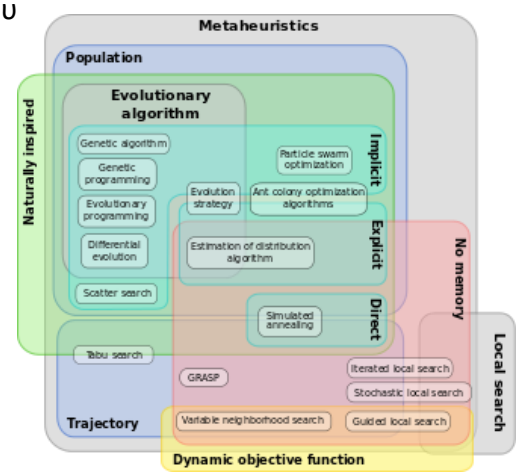

Figure 2.2.1Διάγραμμα Euler των διαφόρων ταξινομήσεων των μεταευρετικών μεθόδων

Προτάθηκαν πολλές μεθευρετικες ιδέες για τη βελτίωση της ευρετικής τοπικής αναζήτησης (heuristic local search), προκειμένου να βρεθούν καλύτερες λύσεις. Τέτοια μεταεπισκόπηση περιλαμβάνουν προσομοιωμένη ανόπτηση (simulated annealing), αναζήτηση tabu (tabu search), επαναλαμβανόμενη τοπική αναζήτηση (iterated local search), αναζήτηση μεταβλητής γειτονιάς (variable neighborhood search) και GRASP. Αυτά τα μεταεπιστημιακά στοιχεία μπορούν να ταξινομηθούν τόσο ως τοπικά βασισμένα στην αναζήτηση είτε ως παγκόσμια μετα-ευρετήρια αναζήτησης.

Άλλοι μεθευρετικοί (Global search) που δεν βασίζονται στην τοπική αναζήτηση (local search)αλλά βασίζονται σε κάποιο πληθυσμό (population-based metaheuristics). Τετοιοι μεθευρετικοί αλγόριθμοι είναι:η βελτιστοποίηση με την αποικία των μυρμηγκιών(Ant Colony Optimization), οι εξελικτικοί υπολογισμοί (Evolutionary computation),η βελτιστοποίηση των σωματιδίων(particle swarm), γενετικοί αλγόριθμοι(Genetic Algorithm) και τον αλγόριθμο βελτιστοποίησης αναβάτη (Rider Optimization Algorithm)

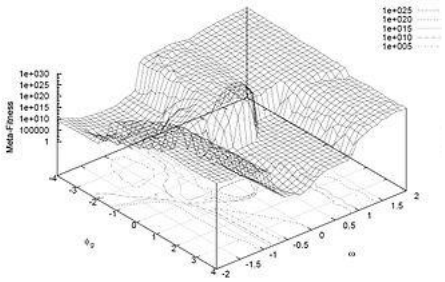

Figure 2.3.1 Population-based vs. Single-solution

2.3. Single-solution vs. population-based

Άλλη μια άλλη μέθοδος ταξινόμησης είναι μία λύση (Single-solution) και αναζητήσεων με βάση τον πληθυσμό (Population-based). Οι μέθοδοι μιας λύσης επικεντρώνονται στην τροποποίηση και τη βελτίωση μιας μοναδικής υποψήφιας λύσης. Η μεθευρετική μιας λύσης περιλαμβάνει προσομοιωμένη ανόπτηση (simulated annealing), επαναλαμβανόμενη τοπική αναζήτηση (iterated local search), μεταβλητή αναζήτηση γειτονιάς (variable neighborhood search) και καθοδηγούμενη τοπική αναζήτηση (and guided local search). Οι μέθοδοι με βάση τον πληθυσμό διατηρούν και βελτιώνουν πολλαπλές υποψήφιες λύσεις, συχνά χρησιμοποιώντας χαρακτηριστικά πληθυσμού για να καθοδηγήσουν την αναζήτηση. Οι μεθευρετικοί αλγόριθμοι βάση τον πληθυσμό περιλαμβάνουν εξελικτικούς υπολογισμούς (evolutionary computation), γενετικούς αλγόριθμους(genetic algorithms) και βελτιστοποίηση σωματιδίων (particle swarm optimization). Μια άλλη κατηγορία μεθευρετικών αλγορίθμων είναι η νοημοσύνη σμήνους (Swarm Intelligence) που είναι μια συλλογική συμπεριφορά αποκεντρωμένων, αυτο-οργανωμένων παραγόντων σε έναν πληθυσμό ή σμήνος. Η βελτιστοποίηση της αποικίας των μυρμηγκιών (Ant Colony Optimization), βελτιστοποίηση σωματιδίων (Patricle Swarm Optimization),η κοινωνική γνωστική βελτιστοποίηση (Social Cognitive Optimization) είναι παραδείγματα αυτής της κατηγορίας

2.4. Αλγόριθμοι υβριδοποίησης και μνήμης

Ένας υβριδικός μεθευρετικός αλγόριθμος, συνδυάζει ένα μεθευρετικό αλγόριθμο με άλλους αλγόριθμους βελτιστοποίησης, όπως αλγόριθμοι από μαθηματικό προγραμματισμό

(Mathematical Programming), προγραμματισμός με περιορισμους (constraint programming) και μηχανική μάθηση (machine learning).Και τα δύο συστατικά ενός υβριδικού μεθευτετικού αλγορίθμου μπορούν να λειτουργήσουν ταυτόχρονα και να ανταλλάσσουν πληροφορίες για να καθοδηγήσουν την αναζήτηση.

Από την άλλη πλευρά, οι αλγόριθμοι μνήμης αντιπροσωπεύουν τη συνέργεια (synergy) της εξέλιξης ή οιασδήποτε άλλη μέθοδο βάσει τον πληθυσμού με ξεχωριστές ατομικές διαδικασίες μάθησης ή τοπικής βελτίωσης . Ένα παράδειγμα αλγορίθμου μνήμης είναι η χρήση ενός τοπικού αλγορίθμου αναζήτησης αντί κάποιου εξελικτικού αλγόριθμου.

2.5. Μεθευρετικοί αλγόριθμοι που βασίζονται στη φύση

Ένας πολύ ενεργός τομέας της έρευνας είναι ο σχεδιασμός των μεθευρετικών αλγορίθμων οι οποίοι είναι εμπνευσμένοι από τη φύση. Πολλοί πρόσφατοι μεθευρετικοί αλγόριθμοι, ειδικά εξελικτικοί αλγόριθμοι που βασίζονται σε υπολογισμούς που είναι εμπνευσμένοι από την φύση. Η φύση λειτουργεί ως πηγή ιδεών, μηχανισμών και αρχών για το σχεδιασμό τεχνητών υπολογιστικών συστημάτων για την αντιμετώπιση πολύπλοκων υπολογιστικών προβλημάτων. Τέτοιοι μεθευρετικοί αλγόριθμοι είναι:προσομοιωμένη ανόπτηση (Simulated Annealing), εξελικτικοί αλγόριθμοι (Evolutionary Algorithms), βελτιστοποίηση αποικίας μυρμηγκιών (Ant Colony Optimization) και βελτιστοποίηση σμήνους σωματιδίων (Particle Swarm Optimization).

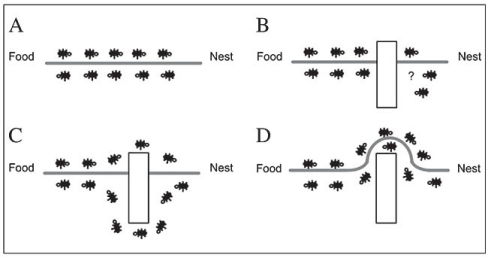

Figure 2. A. Ants in a pheromone trail between nest and food; B. an obstacle interrupts the trail; C. ants find two paths to go around the obstacle; D. a new pheromone trail is formed along the shorter path.

Figure 2.5.1 Συμπεριφορά μυρμηγκιών

3. ΝΟΗΜΟΣΥΝΗ ΣΜΗΝΟΥΣ (SWARM INTELLIGENCE)

Η νοημοσύνη σμήνους (Swarm Intelligence) είναι η συλλογική συμπεριφορά αποκεντρωμένων, αυτο-οργανωμένων συστημάτων, φυσικών ή τεχνητών. Η ιδέα χρησιμοποιείται στην τεχνητή νοημοσύνη. Η έκφραση εισήχθη από τους Gerardo Beni και Jing Wang το 1989, στο πλαίσιο των κυτταρικών ρομποτικών συστημάτων.

Τα συστήματα νοημοσύνης σμήνους (SI) αποτελούνται συνήθως από έναν πληθυσμό απλών agents ή ομάδων που αλληλεπιδρούν τοπικά μεταξύ τους και με το περιβάλλον τους. Η έμπνευση προέρχεται συχνά από τη φύση, ειδικά από βιολογικά συστήματα. Οι agents ακολουθούν πολύ απλούς κανόνες, και παρόλο που δεν υπάρχει κεντρική δομή ελέγχου που να υπαγορεύει τον τρόπο με τον οποίο πρέπει να συμπεριφέρονται μεμονωμένα οι agents, τοπικά και σε κάποιο βαθμό τυχαίες, οι αλληλεπιδράσεις μεταξύ των agents οδηγούν στην εμφάνιση «έξυπνης» συμπεριφοράς global, άγνωστης σε κάθε agent. Παραδείγματα νοημοσύνης σμήνος σε φυσικά συστήματα περιλαμβάνουν αποικίες μυρμηγκιών, αποικίες μελισσών, σμήνη πουλιών, κυνήγι των γερακιών, κτηνοτροφία, ανάπτυξη βακτηρίων, και μικροβιακή νοημοσύνη.

Figure 3.1 Νοημοσυνη σμηνους

Η εφαρμογή των αρχών του σμήνους στα ρομπότ ονομάζεται ρομποτική σμήνος ενώ η νοημοσύνη σμήνος αναφέρεται στο πιο γενικό σύνολο αλγορίθμων. Η πρόβλεψη σμήνος έχει χρησιμοποιηθεί στο πλαίσιο των προβλημάτων πρόβλεψης. Παρόμοιες προσεγγίσεις με εκείνες που προτείνονται για τη ρομποτική σμήνος λαμβάνονται υπόψη για γενετικά τροποποιημένους οργανισμούς στη συνθετική συλλογική νοημοσύνη.

3.1. Μοντέλα με συμπεριφορά σμήνους

1. Boids (Reynolds 1987)

Το Boids είναι ένα πρόγραμμα τεχνητής ζωής, που αναπτύχθηκε από τον Craig Reynolds το 1986, το οποίο προσομοιώνει τη συρρέουσα συμπεριφορά των πουλιών. Η εργασία του για αυτό το θέμα δημοσιεύθηκε το 1987 στις εργασίες του συνεδρίου ACM SIGGRAPH.Το όνομα "boid" αντιστοιχεί σε μια συντομευμένη έκδοση του "bird-oid object", το οποίο αναφέρεται σε ένα αντικείμενο που μοιάζει με πουλί.

Όπως με τις περισσότερες προσομοιώσεις τεχνητής ζωής, το Boids είναι ένα παράδειγμα αναδυόμενης συμπεριφοράς. Δηλαδή, η πολυπλοκότητα των Boids προκύπτει από την αλληλεπίδραση των μεμονωμένων agents (τα boilds, σε αυτήν την περίπτωση) τηρώντας ένα σύνολο απλών κανόνων. Οι κανόνες που εφαρμόζονται στον απλούστερο Boids είναι οι εξής:

- **separation**: κατευθυνθείτε για να αποφύγετε τη συσσώρευση τοπικών agents.
- **alignment**: κατευθυνθείτε προς τη μέση κατεύθυνση των τοπικών agents.
- **cohesion**:κατευθυνθείτε για να κινηθείτε προς τη μέση θέση (κέντρο μάζας) του τοπικού agent.

Μπορούν να προστεθούν πιο περίπλοκοι κανόνες, όπως η αποφυγή εμποδίων και η αναζήτηση στόχων.

2. Αυτοκινούμενα σωματίδια (Self-propelled particles(SPP)) (Vicsek et al. 1995)

Τα αυτοκινούμενα σωματίδια (Self-propelled particles(SPP)), που αναφέρονται επίσης ως μοντέλο Vicsek, εισήχθησαν το 1995 από τους Vicsek et al. ως ειδική περίπτωση του μοντέλου boids που παρουσιάστηκε το 1986 από τον Reynolds. [3] Ένα σμήνος που μοντελοποιείται στο SPP από μια συλλογή σωματιδίων που κινούνται με σταθερή ταχύτητα αλλά ανταποκρίνονται σε μια τυχαία διαταραχή υιοθετώντας κάθε φορά την αύξηση της μέσης κατεύθυνσης κίνησης των άλλων σωματιδίων στην τοπική τους γειτονιά. Τα μοντέλα SPP προβλέπουν ότι τα σμήνη ζώων μοιράζονται ορισμένες ιδιότητες σε επίπεδο ομάδας, ανεξάρτητα από τον τύπο των ζώων στο σμήνος. Τα συστήματα σμήνους δημιουργούν συμπεριφορές που εμφανίζονται σε πολλές διαφορετικές κλίμακες, μερικές από τις οποίες αποδεικνύονται καθολικές και στιβαρές. Έχει γίνει πρόκληση στη θεωρητική φυσική να βρεθούν ελάχιστα στατιστικά μοντέλα που καταγράφουν αυτές τις συμπεριφορές.

3.2. Μεθευρετικοί αλγόριθμοι

Οι εξελικτικοί αλγόριθμοι ((Evolutionary algorithms)EA)), η βελτιστοποίηση σωματιδίων(Particle swarm optimization(PSO)), η διαφορική εξέλιξη (differential evolution)(DE), η βελτιστοποίηση των αποικιών ((ant colony optimization)ACO) και οι παραλλαγές τους κυριαρχούν στον τομέα των μεθευρετικών οι οποίοι είναι εμπνευσμένοι από τη φύση. Αυτή η λίστα περιλαμβάνει αλγόριθμους που δημοσιεύθηκαν μέχρι το έτος 2000. Θα πρέπει επίσης να σημειωθεί ότι οι μεθευρετικοί, όσο καλοί και αν είναι,μπορεί να μην βρουν λύση. Όταν προσδιορίζονται οι κατάλληλες παράμετροι και όταν επιτυγχάνεται επαρκές στάδιο σύγκλισης, συχνά βρίσκουν μια βέλτιστη λύση ή πλησιάζει τη βέλτιστη - ωστόσο, εάν κάποιος δεν γνωρίζει εκ των προτέρων τη βέλτιστη λύση, τοτε δεν είναι γνωστή η ποιότητα μιας λύσης που βρήκε. Παρά αυτό το προφανές μειονέκτημα έχει αποδειχθεί ότι αυτοί οι τύποι αλγορίθμων λειτουργούν καλά στην πράξη, και έχουν διερευνηθεί εκτεταμένα και αναπτυχθεί. Από την άλλη πλευρά, είναι δυνατόν να αποφευχθεί αυτό το μειονέκτημα υπολογίζοντας την ποιότητα της λύσης για κάποια ειδική περίπτωση όπου είναι δυνατός ένας τέτοιος υπολογισμός και μετά από αυτήν την εκτέλεση είναι γνωστό ότι κάθε λύση που

είναι τουλάχιστον τόσο καλή όσο η λύση που είχε αυτή η ειδική περίπτωση. Ένα τέτοιο παράδειγμα είναι ο αλγόριθμος Monte Carlo για το Minimum Feedback Arc Set, όπου αυτό επιτεύχθηκε πιθανώς μέσω υβριδισμού του αλγορίθμου Monte Carlo με την τεχνική Ant Colony Optimization.

1. Αναζήτηση στοχαστικής διάχυσης(Stochastic diffusion search(SDS)) (Bishop 1989)

   Δημοσιεύτηκε για πρώτη φορά το 1989 Η αναζήτηση στοχαστικής διάχυσης (Stochastic diffusion search (SDS)) ήταν ο πρώτος μεθευρετικός με νοημοσύνη σμήνους. Το SDS είναι μια τεχνική παγκόσμιας αναζήτησης και βελτιστοποίησης agents-based που ταιριάζει καλύτερα σε προβλήματα όπου η αντικειμενική συνάρτηση μπορεί να αποσυντεθεί σε πολλαπλές ανεξάρτητες μερικές συναρτήσεις. Κάθε πράκτορας διατηρεί μια υπόθεση που ελέγχεται επαναληπτικά αξιολογώντας μια τυχαία επιλεγμένη μερική αντικειμενική συνάρτηση που παραμετροποιείται από την τρέχουσα υπόθεση του agent. Στην τυπική έκδοση του SDS, αυτές οι αξιολογήσεις μερικής συνάρτησης είναι δυαδικές, με αποτέλεσμα κάθε agent να είναι ενεργός ή ανενεργός. Οι πληροφορίες για τις υποθέσεις διαχέονται σε ολόκληρο τον πληθυσμό μέσω της επικοινωνίας μεταξύ των agents. Σε αντίθεση με τη στιγματική επικοινωνία που χρησιμοποιείται στο ACO, οι agents στον SDS επικοινωνούν υποθέσεις μέσω μιας στρατηγικής επικοινωνίας ένας προς έναν ανάλογος με τη διαδικασία παράλληλης λειτουργίας που παρατηρείται στο Leptothorax acervorum. Ένας θετικός μηχανισμός ανατροφοδότησης διασφαλίζει ότι, με την πάροδο του χρόνου, ένας πληθυσμός agent σταθεροποιείται γύρω από την global καλύτερη λύση. Ο SDS είναι ένας αποτελεσματικός και ισχυρός αλγόριθμος global αναζήτησης και βελτιστοποίησης, ο οποίος έχει περιγραφεί εκτενώς μαθηματικά. Η πρόσφατη εργασία περιελάμβανε τη συγχώνευση των ιδιοτήτων global αναζήτησης του SDS με άλλους αλγόριθμους συλλογής πληροφοριών

2. Βελτιστοποίηση με την αποικία των μυρμηγκιών(Ant Colony Optimization 1992(ACO)) Dorigo 1992

   Η βελτιστοποίηση της αποικίας μυρμηγκιών (ACO), που εισήγαγε ο Dorigo στη διδακτορική του διατριβή, είναι μια κατηγορία αλγορίθμων βελτιστοποίησης με βάση τις δράσεις μιας αποικίας μυρμηγκιών. Ο ACO είναι μια τεχνική πιθανοτήτων χρήσιμη σε προβλήματα που σχετίζονται με την εύρεση καλύτερων διαδρομών μέσω γραφημάτων. Τα τεχνητά «μυρμήγκια» - προσομοιώνουν agents - εντοπίζουν βέλτιστες λύσεις μετακινώντας έναν χώρο παραμέτρων που αντιπροσωπεύει όλες τις

πιθανές λύσεις. Τα φυσικά μυρμήγκια καθορίζουν φερομόνες που κατευθύνουν ο ένας τον άλλο σε πόρους ενώ εξερευνούν το περιβάλλον τους. Τα τεχνητά «μυρμήγκια» καταγράφουν επίσης τις θέσεις τους και την ποιότητα των λύσεών τους, έτσι ώστε σε μεταγενέστερες προσομοιώσεις να επαναλαμβάνονται από περισσότερα μυρμήγκια για καλύτερες λύσεις.

3. Βελτιστοποίηση σωματιδίων(Particle swarm optimization (PSO)) Kennedy, Eberhart & Shi 1995

Η βελτιστοποίηση σωματιδίων είναι ένας αλγόριθμος παγκόσμιας(global) βελτιστοποίησης για την αντιμετώπιση προβλημάτων στα οποία μια καλύτερη λύση μπορεί να αναπαρασταθεί ως σημείο ή επιφάνεια σε ένα μη δισδιάστατο χώρο. Οι υποθέσεις σχεδιάζονται σε αυτόν τον χώρο και σπέρνονται με αρχική ταχύτητα, καθώς και ένα κανάλι επικοινωνίας μεταξύ των σωματιδίων. Στη συνέχεια, τα σωματίδια κινούνται μέσω της αντικειμενικής συνάρτησης και αξιολογούνται σύμφωνα με κάποιο κριτήριο φυσικής κατάστασης μετά από κάθε χρονικό βήμα. Με την πάροδο του χρόνου, τα σωματίδια επιταχύνονται προς εκείνα τα σωματίδια εντός της ομαδοποίησης επικοινωνίας τους που έχουν καλύτερες τιμές. Το κύριο πλεονέκτημα μιας τέτοιας προσέγγισης έναντι άλλων στρατηγικών παγκόσμιας (global)ελαχιστοποίησης όπως η προσομοίωση ανόπτησης (simulated annealing) είναι ότι ο μεγάλος αριθμός μελών που απαρτίζουν το σμήνος σωματιδίων κάνουν την τεχνική εντυπωσιακά ανθεκτική στο πρόβλημα των τοπικών ελαχίστων.

4. Τεχνητή νοημοσύνη σμήνους(Artificial Swarm Intelligence(ASI (2015))

Η τεχνητή νοημοσύνη σμήνους είναι μια μέθοδος ενίσχυσης της συλλογικής νοημοσύνης των δικτύων ανθρώπινων ομάδων χρησιμοποιώντας αλγόριθμους ελέγχου που έχουν διαμορφωθεί σύμφωνα με τα φυσικά σμήνη. Μερικές φορές αναφέρεται ως Human Swarming ή Swarm AI,η τεχνολογία συνδέει ομάδες ανθρώπινων συμμετεχόντων σε συστήματα πραγματικού χρόνου που συζητούν και συγκλίνουν σε λύσεις ως δυναμικά σμήνη όταν παρουσιάζονται ταυτόχρονα με μια ερώτηση. Η ASI έχει χρησιμοποιηθεί για ένα ευρύ φάσμα εφαρμογών, από τη δυνατότητα των επιχειρηματικών ομάδων να παράγουν πολύ ακριβείς οικονομικές προβλέψεις έως τη δυνατότητα παίκτες να ξεπεράσουν τις αγορές στοιχημάτων Vegas. Η ASI έχει επίσης χρησιμοποιηθεί για να επιτρέψει σε ομάδες ιατρών να δημιουργήσουν διαγνώσεις με σημαντικά μεγαλύτερη ακρίβεια από τις παραδοσιακές μεθόδους.

4. ΑΛΓΟΡΙΘΜΟΣ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ ΑΠΟΙΚΙΑΣ ΜΥΡΜΗΓΚΙΩΝ

Η Βελτστοποίηση Αποικίας Μυρμηγκιών (Ant Colony Optimization(ACO))

είναι σχετικά μια νέα στρατιγική για την επίλυση προβλημάτων συνδυαστικής

βελτιστοποίησης που παρουσιάστηκε από τους Dorigo, Maniezzo και Colorni. Η βελτιστοποίηση αποικίας μυρμηγκιών είναι ένα σύστημα που αντιγράφει την συμπεριφορά των μυρμηγκιών κατά την διαδικασία της αναζήτησης της τροφής τους. Τα μυρμήγκια διαθέτουν και αναπτύσσουν μια τεχνική για την εύρεση της συντομότερης διαδρομής από την φωλιά τους προς την τροφή τους, και αντιθέτως. Τα μυρμήγκια αρχίζουν την αναζήτηση της τροφής τους γύρω από την φωλιά τους με τυχαίο τρόπο και καθώς κινούνται αφήνουν πίσω τους μία ποσότητας μιας ούσιάς που ονομάζεται φερομόνη και με αυτό τον τρόπο 'μαρκάρουν' το μονοπάτι που έχουν διανύσει. Η ποσότητα της φερομόνης απο κάθε μονοπάτι εξαρτάται από την απόσταση, την ποιότητα και την ποσότητα της τροφής που βρέθηκε. Το επόμενο μυρμήγκι που θα φύγει απο την φωλιά του είναι πιο πιθανό να ακολουθήσει την φερομόνη που θα υπάρχει σε κάποιο μονοπάτι, αφήνοντας μια επιπλέον ποσότητα φερομόνης στο ίδιο μονοπάτι. Καθώς η ποσότητα φερομόνης στο συγκεκριμένο μονοπάτι όλο και αυξάνεται, όλο και περισσότερα μυρμήγκια θα ακολουθούν αυτό το μονοπάτι. Όπως είναι λογικό η φερομόνη ελαττώνεται καθώς η ώρα περνάει, από τα μονοπάτια που δεν πηγαίνουν πολλα μυρμήγκια. Ως εκ' τουτου σε όλα τα άλλα μονοπάτια η φερομόνη εξαφανίζεται με αποτέλεσμα όλα τα μυρμήγκια να ακολουθούν το ίδιο μονοπάτι, το οποίο είναι και η βέλτιστη ή πολύ κοντά στην βέλτιστη λύση.

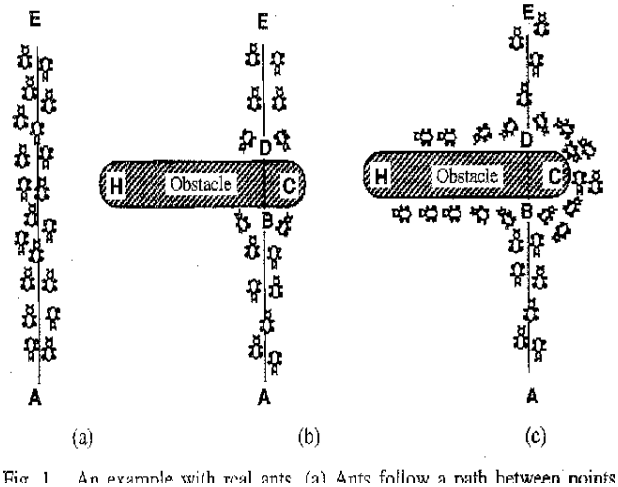

Figure 4.1 Αλγόριθμος βελτιστοποίησης αποικιών των μυρμηγκιών

Η βασικότερη ιδέα της βελτιστοποίησης αποικίας μυρμηγκιών είναι να μοντελοποιούν το πρόβλημα ως πρόβλημα εύρεσης μονοπατιού με το ελάχιστο κόστος σε ένα γράφημα.
Κάθε μυρμήγκι είναι μια πιθανή λύση για το πρόβλημα. Ο αλγόριθμος ακολουθεί έναν αριθμό επαναλήψεων, όπου σε κάθε επανάληψη κάθε μυρμήγκι αρχίζει να κατασκευάζει την λύση του με βάση τις λύσεις που δημιουργήθηκαν από τα προηγούμενα μυρμήγκια. Έπειτα απο έναν αριθμό επαναλήψεων όλα ή σχεδόν όλα τα μυρμήγκια ακολουθούν την ίδια διαδρομή η οποία πιθανών είναι και η βέλτιστη.

Ο αλγόριθμου βελτιστοποίησης αποικίας των μυρμηγκιών θα εφαρμοστεί στο **πρόβλημα του πλανόδιου πωλητή**(**TSP**) το οποίο είναι και το πρώτο πρόβλημα που εφαρμόστηκε η

συγκεκριμένη μέθοδος και που είναι το κλασικότερο πρόβλημα για οποιοδήποτε αλγόριθμο που εφαρμόζεται σε προβλήματα συνδυαστικής βελτιστοποίησης. Στο πρόβλημα του πλανόδιου πωλητή, η φερομόνη συσχετίζεται με τα τόξα και για αυτό το λόγο η φερομόνη συμβολίζεται με τ_{ij}, όπου είναι η επιθυμία του πωλητή να επισκεφτεί απευθείας μετά την πόλη i την πόλη j.
Ως ευρευτική συνάρτηση για το συγκεκριμένο πρόβλημα ακολουθούμε την παρακάτω:

$$n_{ij}\frac{1}{c_{ij}} \text{(4.1)}$$

Όπου c_{ij} είναι η απόσταση απο το τόξο i στο τόξο j και η εύρετική συνάρτηση μας λέει πως η επιθυμία του πωλητή να πάει απο το τόξο i στο τόξο j έιναι αντιστρόφως ανάλογη με την απόσταση ανάμεσα στις πόλεις. Αν τον κόστος c_{ij} είναι μηδέν, τότε δίνεται στην ευρετική συνάρτηση μια πολύ μικρή τιμή. Γενικά, η ευρετική συνάρτηση ή αλλιώς πληροφορία, αντιπροσωπεύει μια εκ των προτέρων πληροφορία η οποία παρέχεται απο μία ανεξάρτητη πηγή πέρα απο τα μυρμήγκια. Θα μπορούσε να είναι το κόστος ή μια εκτίμηση του κόστους. Αυτή η πληροφορία, όπως θα δούμε στην συνέχεια είναι πολύ σημαντική γιατί χρησιμοποιείται απο τα μυρμήγκια για να πάρουν πιθανοτικές αποφάσεις για το πώς θα κινηθούν μέσα στο γράφημα.

Οι κύκλοι στο πρόβλημα του πλανόδιου πωλητή κατασκευάζονται ως εξής:

Αρχικά έχοντας έναν αριθμό απο μυρμήγκια τα αφήνουμε σε διαφορετική πόλη το καθένα, στην συνέχεια χρησιμοποιούμε τη φερομόνη και την ευρετική συνάρτηση για την κατασκευή, με βάση τις πιθανότητες ενός κύκλου που θα ξεκινάει και θα τελειώνει στην πόλη που αφήσαμε το μυρμήγκι. Όταν όλα τα μυρμήγκια έχουν κατασκευάσει τον κύκλο τους μπορούν να αφήσουν φερομόνη στις πόλεις που επισκέφθηκαν. Βεβαίως όπως θα δούμε και παρακάτω, έχουν γίνε πολλές παραλλαγές του αρχικού αλγορίθμου σε αυτό το σημείο. Δηλαδή στο σημείο που μας λέει ποια μυρμήγκια και την ποσότητα της φερομόνης που θα αφήσουν και σε ποια τόξα. Αυτό είναι κομβικό ερώτημα που τίθεται στον αλγόριθμο γιατί αν όλα τα μυρμήγκια αφήσουν την ίδια ποσότητα φερομόνης στους κύκλους που έχουν επιλέξει - όπως είναι λογικό -, δεν θα ξεχωρίσει εύκολα και γρήγορα μια καλή διαδρομή. Επιπλέον με αυτόν τον τρόπο δεν θα μπορέσει ο αλγόριθμος να βρει καλύτερες λύσεις. Μια ιδιαίτερα καλή στρατηγική είναι να αφήσει φερομόνη μόνο το καλύτερο μυρμήγκι. Από την άλλη, πολλές φορές πριν ξεκινήσει η επόμενη επανάληψη και πριν αποφασιστεί ποιο μυρμήγκι και και πόση ποσότητα φερομόνης θα αφήσει εφαρμόζεται μια διαδικασία τοπικής αναζήτησης με στόχο την όσο το δυνατό βελτίωση της λύσης.

Ένας τρόπος να υπολογιστεί η αρχική φερομόνη είναι:

$$\tau_{ij} = \frac{m}{TD} \quad \text{(4.2)}$$

Όπου m είναι ο συνολικός αριθμός των μυρμηγκιών και TD είναι το κόστος ενός αρχικού κύκλου που έχει κατασκευαστεί απο έναν απλούστερο τρόπο. Στην περίπτωση που η αρχική φερομόνη που δίνεται στο πρόβλημα έχει πολύ μικρή τιμή, τότε υπάρχει πιθανότητα ένα μυρμήγκι (που θα έχει σχετικά καλή αρχικη λύση) να κυριαρχήσει πάνω στην αναζήτηση. Απο την άλλη μεριά αν έχουμε αρχική τιμή πολύ μεγάλη, θα περάσει μεγάλος αριθμός από επαναλήψεις μέχρι να επηρεάσει η αλλαγή της φερομόνης τη λύση.

Για να κατασκευαστούν οι κύκλοι των μυρμηγκιών αρχικά αφήνεται ένας αριθμός απο μυρμήγκια σε διάφορες πόλεις καλό είναι να μην αφήσουμε περισσότερα από ένα μυρμήγκι πάνω σε μία πόλη γιατί υπάρχει πιθανότητα να κατασκευάσουν δυο φορές τον ίδιο κύκλο. Σε κάθε βήμα κάθε μυρμήγκι εφαρμόζει έναν πιθανοτικό κανόνα για να επιλέξει ποια πόλη θα είναι η επόμενη που θα επιλέξει. Κάθε μυρμήγκι θυμάται ποιες πόλεις έχει επισκεφτεί και δεν μπορεί να τις επισκεφτεί ξανά. Το κάθε μυρμήγκι που βρίσκεται σε μια πόλη i επιλέγει αν θα πάει ή όχι την πόλη j με βάση την πιθανότητα:

$$p_{ij} = \frac{[\tau_{ij}]^a [n_{ij}]^\beta}{\sum_{l=1}^{M} [\tau_{il}]^a [n_{il}]^\beta} \quad (4.3)$$

Όπου M το σύνολο των πόλεων και α,β δύο παράμετροι που καθορίζουν αν επηρεάζει περισσότερο την επιλογή η φερομόνη που βρίσκεται πάνω στον τόξο ή η ευρετική πληροφορία που έχει υπολογιστεί για κάθε τόξο. Έχοντας αυτό ως πιθανοτικό κανόνα η πιθανότητα επιλογής ενός τόξου αυξάνει όταν αυξάνει η φερομόνη του τόξου και η ευρετική πληροφορία του τόξου. Αν το $\alpha = 0$ οι κοντινότερες πόλεις είναι ποιο πιθανό να επιλεγούν, και μετατρέπεται στον αλγόριθμο του πλησιέστερου γείτονα και γίνεται Greedy:

$$p_{ij} = \frac{[n_{ij}]^\beta}{\sum_{l=1}^{M} [n_{il}]^\beta} \quad (4.4)$$

Αν χρησιμοποιήσουμε μόνο την φερομόνη δηλαδή β=0, τότε δεν θα έχουμε αξιόλογα αποτελέσματα διότι ο αλγόριθμος δεν θα λαμβάνει καθόλου υπόψη του τις αποστάσεις. Οπότε θα καταλήξουμε σε έναν καθαρά πιθανοτικό αλγόριθμο ο οποίος θα έχει τυχαία βήματα και άρα η εξίσωση θα είναι της μορφής:

$$p_{ij} = \frac{[\tau_{ij}]^a}{\sum_{l=1}^{M} [\tau_{il}]^a} \quad (4.5)$$

Εφόσον έχουν κατασκευαστεί οι αρχικοί κύκλοι απο όλα τα μυρμήγκια, σε κάθε τόξο ενημερώνεται και η ποσότητα φερομόνης, έτσι προστίθεται μια νέα ποσότητα φερομόνης στα τόξα που διασχίζουν τα μυρμήγκια

Η αρχική μείωση της φερομόνης μειώνεται σε όλα τα τόξα με την παρακάτω ποσότητα:

$$\tau_{ij} = (1 - \rho)\tau_{ij} \text{ (4.6)}$$

Για εξασφαλίσουμε ότι ο αλγόριθμος θα 'ξεχάσει' τις όχι και τόσο καλές αποφάσεις που έχουν ληφθεί κατά το πέρας των επαναλήψεων, χρησιμοποιούμε τον συντελεστή 'εξάτμισης' 0 < ρ < 1. Όσο ένα τόξο δεν διασχίζεται από τα μυρμήγκια τόσο και η φερομόνη του τόξου κατα το πέρας των επαναλήψεων μειώνεται εκθετικά μέχρι να φτάσει στο μηδέν. Η φερομόνη από όσα τόξα χρησιμοποιήθηκαν από τα μυρμήγκια αυξάνεται κατά:

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \text{ (4.7)}$$

Ορίζεται ως $\Delta\tau_{ij}^{k}$ η ποσότητα της φερομόνης που αφήνει το κάθε μυρμήγκι(k) από τόξα που πέρασε και ορίζεται από τον ακόλουθο τύπο:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{c^k}, \text{εαν το τόξο έχει επιλεγεί απο μυρμήγκι κ} \\ 0, \text{αλλιως} \end{cases} \quad \text{(4.8)}$$

Το κόστος του κύκλου που έχει δημιουργήσει κάθε μυρμήγκι (k) ορίζεται ως c^k.

Ο κύκλος με το μικρότερο κόστος θεωρείται ως πιο βέλτιστος, και είναι αυτός που τα μυρμήγκια έχουν αφήσει την περισσότερη φερομόνη στα τόξα, και όπως είναι επακόλουθο είναι και αυτός που έχει τις περισσότερες πιθανότητες να επιλέγει απο τα μηρμήγκια ξανά στην επόμενη επανάληψη.

Αν τυχών τα μυρμήγκια αφήσουν την ίδια ποσότητα φερομόνης στα τόξα που διασχίζουν, τότε θα έχουμε πολύ αργήσει να βρει μια καλή λύση.

Αυτό έχει σαν αποτέλεσμα να έχουν παρουσιαστεί πολλές παραλλαγές την διαδικασίας της ενημέρωσης της φερομόνης. Για παράδειγμα θα μπορούσε μόνο το βέλτιστο μυρμήγκι να αφήνει φερομόνη ή εμείς να λαμβάνουμε μόνο αυτό υπόψιν ή να επιλέξουμε στον αλγόριθμο το καλύτερο μυρμληγκι να αφήνει περισσότερη φερομόνη από τα υπόλοιπα

Στην συνέχεια δίνεται ένας ψευδοκώδικας για την γενική ιδέα του αλγορίθμου:

Αλγόριθμος:Βελτιστοποίηση αποικίας μυρμηγκιών

1. Αρχικοποίηση
    - Υπολογισμός της αρχικής φερομόνης t_{ij} για κάθε τόξο.
    - Δημιουργία του αρχικού πληθυσμού των μυρμηγκιών.

- Υπολογισμός της ευρετικής συνάρτησης n_{ij} για κάθε τόξο.
- Τοποθέτηση μυρμηγκιών στις αρχικές πόλεις.
- Σε κάθε μυρμήγκι τοποθέτησε την αρχική σε μια λίστα που αντιπροσωπεύει την μνήμη του κάθε μυρμηγκιού
- Όρισε τον μέγιστο αριθμό των επαναλήψεων

2. Μέχρι όλα τα μυρμήγκια να επισκεφτούν όλες τις πόλεις του γραφήματος
    - Για κάθε πόλη
        - Για κάθε μυρμήγκι

          Επέλεξε την επόμενη πόλη με την πιθανότητα απο τον τύπο 4.3

          Μετακίνησε το μυρμήγκι στην πόλη j

          Πρόσθεσε στην μνήμη του μυρμηγκιού (λίστα) την πόλη j

          Υπολόγισε το Δt_{ij}^{k} 4.6

3. Ενημέρωση φερομόνης:
    - Ενημέρωση φερομόνης στην ακμή i, j 4.7
4. Αποθήκευσε τη συντομότερη διαδρομή μέχρι τώρα
    - Αν το (NC<NC_MAX) ή όλα τα μυρμήγκια **δεν** κάνουν την ίδια διαδρομή
        - Άδειασε τις λίστες από τα μυρμήγκια.
        - Πρόσθεσε σε κάθε λίστα την πρώτη πόλη που είναι το κάθε μυρμήγκι.
        - Για κάθε ακμή i, j τότε το $\Delta t_{ij}^{k} = 0$.
        - Πήγαινε στο βήμα 2
    - Αλλιώς
        - Κάνε Print την συντομότερη διαδρομή.

4.1. Παραλλαγές Βασικού Αλγορίθμου Βελτιστοποίησης Αποικίας Μυρμηγκιών

Κατά την διάρκεια των τευλευταίων χρόνων ο αλγόριθμος βελτιστοποίησης αποικίας μυρμηγκιών εφαρμόζεται με πολύ καλά αποτελέσματα σε δύσκολα προβλήματα συνδυαστικής βελτιστοποίησης. Πάρα πολλοί ερευνητές έχουν δημοσιεύσει διάφορες παραλλαγές των εξισώσεων που εφαρμόζονται στον κλασσικό αλγόριθμο.

Ο πρώτος ολοκληρωμένος αλγόριθμός ονομάστηκε **Σύστημα Μυρμηγκιών(Ant System)** σε αυτόν τον αλγόριθμο παρουσιάστηκαν για πρώτη φορά τα εκλεκτά (ελίτ) μυρμήγκια. Τα καλύτερα μυρμήγκια αφήνουν μεγαλύτερη ποσότητα φερομόνης με στόχο να αυξήσουν την πιθανότητα των άλλων μυρμηγκιών για εξερεύνηση στις υποσχόμενες περιοχές του χώρου λύσεων η εξίσωση ενημέρωσης της φερομόνης είναι:

$$\tau_{ij} = \tau_{ij}(t) + \Delta\tau_{ij}(t) + n_e * \Delta\tau_{ij}^{e}(\tau) \quad (4.9)$$

Όπου e το πλήθος των εκλεκτών μυρμηγκιών. Σε αυτόν τον αλγόριθμο δόθηκαν και τρεις διαφορετικοί τρόποι για την ποσότητα της φερομόνης που αφήνει κάποιο η κάποια μυρμήγκια στα τόξα που ανήκουν στην διαδρομή του.

- Ο πρώτος αλγόριθμος ονομάστηκε κύκλος μυρμηγκιών (Ant-Cycle) δίνεται από την εξίσωση:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{f(x^k)(t)}, εαν το τόξο έχει επιλεγεί απο μυρμήγκι κ \\ 0, αλλιως \end{cases} \quad (4.10)$$

Στην ουσία μοιάζει με την εξίσωση 4.8 όπου Q είναι μια σταθερά

- Ο δεύτερος τρόπος που ονομάστηκε πυκνότητα μυρμηγκιών (Ant-Destiny) δίνεται από την εξίσωση:

$$\tau_{ij}^{k} = \begin{cases} Q, \varepsilon\alpha\nu\tau\sigma\tau\acute{o}\xi\sigma\acute{\varepsilon}\chi\varepsilon\iota\varepsilon\pi\iota\lambda\varepsilon\gamma\varepsilon\acute{\iota}\alpha\pi\sigma\mu\upsilon\rho\mu\acute{\eta}\gamma\kappa\iota\kappa \\ 0, \alpha\lambda\lambda\iota\omega\varsigma \end{cases} \quad (4.11)$$

Σε αυτή την περίπτωση κάθε μυρμήγκι αφήνει την ίδια ποσότητα φερομόνης σε κάθε τόξο της διαδρομής του. Στην ουσία αυτή η μέθοδος μετράει το πλήθος των μυρμηγκιών που πέρασαν από το τόξο i, j σε μια επανάληψη. Έτσι όσο μεγαλύτερη πυκνότητα της κίνησης σε ένα τόξο τόσο περισσότερη φερομόνη θα έχει αυτό το τόξο στο τέλος μιας επανάληψης και τόσο περισσότερες πιθανότητες έχει αυτό το τόξο να γίνει η τελική λύση.

- Ο τρίτος τρόπος ονομάστηκε ποσότητα των μυρμηγκιών (Ant-Quantity) δίνεται απο την εξίσωση:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{d_{ij}}, \varepsilon\alpha\nu\tau\sigma\tau\acute{o}\xi\sigma\acute{\varepsilon}\chi\varepsilon\iota\varepsilon\pi\iota\lambda\varepsilon\gamma\varepsilon\acute{\iota}\alpha\pi\sigma\mu\upsilon\rho\mu\acute{\eta}\gamma\kappa\iota\kappa \\ 0, \alpha\lambda\lambda\iota\omega\varsigma \end{cases} \quad (4.12)$$

Όπου d_{ij} συμβολίζουμε την απόσταση μεταξύ των τόξων i,j. Σε αυτή την περίπτωση μόνο η τοπική πληροφορία χρησιμοποιείται κάνοντας με αυτό τον τρόπο πιο επιθυμητά τα τόξα που έχουν μικρότερο κόστος.

5. ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ

5.1. Εισαγωγή

Σε αυτή την ενότητα θα σας παρουσιάσω την δική μου υλοποίηση του αλγορίθμου Ant System.

Για την υλοποίηση χρησιμοποιήθηκαν τα εξής:

- **IDE**:PyCharm by JetBrains
- **Γλώσσα προγραμματισμού**:Python(3.5)
- **Λειτουργικό Σύστημα**:Debian 9 “Stretch”

Και DataSets για την δοκιμή του Αλγορίθμου από πάνω στο πρόβλημα του Πλανόδιου Πωλητή(TSP) απο την βιβλιοθήκη του πανεπιστημίου **Universität Heidelberg.**

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Τα dataset που χρησιμοποίησα είναι τα εξής:

- gr17
- gr21
- ulysses16
- att48

Τέλος ο κώδικας υπάρχει αποθετήριο στο **github** στον παρακάτω σύνδεσμο:

https://github.com/ArisKal/AntSystem-TSP-/tree/devel

5.2. Λειτουργικό σύστημα

Το λειτουργικό σύστημα που χρησιμοποιήθηκε στο Project είναι Debian GNU/LINUX 9 “Stretch”.

Δεν υπάρχει κάποιος συγκεκριμένος λόγος που έγινε σε Linux απλά έχω χρησιμοποιήσει τα τελευταία χρόνια διάφορες διανομές Linux και τα 5 τελευταία χρόνια Debian. Με έχει

βοηθήσει στην ανάπτυξη κώδικά το Linux και βεβαίως και σε άλλες εργασίες εκτός από την ανάπτυξη κώδικα.

5.3. Γλώσσα προγραμματισμού

Η γλώσσα προγραμματισμού που χρησιμοποίησα ήταν Python(3.5) δυο ήταν οι λόγοι που χρησιμοποίησα Python. Ο πρώτος λόγος είναι γιατί δεν είχα ασχοληθεί πολύ και ήταν μια ευκαιρία να γράψω σε Python και ο δεύτερος λόγος είναι ότι είναι μια γλώσσα εύκολη και με πολλές δυνατότητες.

5.4. IDE

Ο IDE που χρησιμοποίησα ήταν το PyCharm της JetBrains ένας πανίσχυρος IDE και μάλιστα υπάρχει και το community edition που διατίθεται δωρεάν.

5.5. DataSets

Για το πρόβλημα του πλανόδιου πωλητή(TSP) χρησιμοποίησα δεδομένα από την βιβλιοθήκη TSPLIB http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ στην οποία υπάρχουν μικρά και μεγάλα προβλήματα για τον Πλανόδιο Πωλητή έτσι μπόρεσα να δοκιμάσω τον κωδικά μου σε ορθότητα και επίδοση.

5.6. Ανάλυση Κώδικα

Στον Κώδικα θα δούμε ότι υπάρχουν 4 κλάσεις:

- main.py
- ReadData.py
- Colony.py
- Ant.py

Θα πάρουμε τις κλάσεις μια μια και θα κάνουμε μια πολύ σύντομη ανάλυση:

ReadData.py

Δημιούργησα αυτή την κλάση η οποία διαβάσει το .xml που περιέχει να δεδομένα απο το πρόβλημα το αρχείο μπορεί να κατέβει από τον παρακάτω σύνδεσμο:

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/.

και τα επιστρέφει σε ένα πίνακα δισδιάστατο, πληροφορίες για την δομή του .xml .

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/Description.pdf

Ant.py

Η κλάση αυτή περιέχει πληροφορίες για κάθε μυρμήγκι

Την πόλη από την οποία ξεκινάει την διαδρομή του.

- Την τρέχουσα θέση που βρίσκεται κάθε χρονική στιγμή το μυρμήγκι.
- Τις πόλεις που επιτρέπεται να επισκεφτεί το μυρμήγκι δηλαδή αυτές που δεν έχει επισκεφτεί ακόμα.
- Η διαδρομή που έχει κάνει το μυρμήγκι.
- Και το συνολικό κόστος της διαδρομής.
- Το id του μυρμηγκιού δηλαδή κάθε μυρμήγκι έχει έναν μοναδικό αριθμό.

Colony.py

Η κλάση Colony περιέχει τις πληροφορίες της αποικίας :

- Τον συνολικό αριθμό των μυρμηγκιών.
- Την σταθερά α(alpha).
- Την σταθερά β(beta).
- Τον ρυθμό εξάτμισης της φερομόνης.
- Τον συνολικό αριθμό των κύκλων που θα κάνει η αποικία.
- Την φερομόνη που υπάρχει.
- Το μικρότερο κόστος(Global).

- Την διαδρομή με το μικτότερο κόστος (Global).

main.py

Στην **main** δημιουγείται η αποικία,και δίνονται οι κατάλληλες μεταβλητές για την λύση του προβλήματος.

5.7. Δοκιμές και αποτελέσματα

Σε αυτή την ενότητα θα κάνουμε δοκιμές πάνω σε προβλήματα με γνωστά αποτελέσματα για τα δούμε ποσό αποτελεσματικός και αποδοτικός είναι ο αλγόριθμος που κατασκεύασα.

Τα προβλήματα τα βρήκα από την βιβλιοθήκη TSPLIB http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Τα προβλήματα δοκιμάστηκαν τουλάχιστον 5 φορές το καθένα με μικρές διαφορές στους παραμέτρους:

Ulysses16

Μέγεθος προβλήματος 16 πόλεις

Γράφος: Συμμετρικός

Καλύτερη γνώστη λύση:6859

Link:http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/ulysses16.xml.zip

Trials	Number of Cycles	Number Of Ants	Aplha	Beta	Ρυθμός Εξάτμισης	Μέσο Κόστος	Βέλτιστο
1	700	16	1	5	0.5	6.901	6859
2	700	16	1	5	0.9	6.908,6	6903
3	700	16	1	7	0.9	6.897,4	6859

Gr17

Μέγεθος προβλήματος 17 πόλεις

Γράφος: Συμμετρικός

Καλύτερη γνώστη λύση: 2085

Link:http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/gr17.xml.zip

Trials	Number of Cycles	Number Of Ants	Aplha	Beta	Ρυθμός Εξάτμισης	Μέσο Κόστος	Βέλτιστο
1	700	17	1	5	0.5	2.109	2085
2	700	17	1	5	0.9	2.097,8	2085
3	700	17	1	7	0.9	2.093,4	2085

Gr21

Μέγεθος προβήματος 21 πόλεις

Γράφος: Συμμετρικός

Καλύτερη γνώστη λύση: 2707

Link:http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/gr21.xml.zip

Trials	Number of Cycles	Numbe r Of Ants	Aplha	Beta	Ρυθμός Εξάτμισης	Μέσο Κόστος	Βέλτιστο
1	700	21	1	5	0.5	2.772,4	2707
2	700	21	1	5	0.9	2.724,6	2707
3	700	21	1	7	0.9	2.766,6	2707

Att48

Μέγεθος προβήματος 48 πόλεις

Γράφος: Συμμετρικός

Καλύτερη γνώστη λύση: 10628

Link:http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-TSPLIB/instances/att48.xml.zip

Trials	Number of Cycles	Numbe r Of Ants	Aplha	Beta	Ρυθμός Εξάτμισης	Μέσο Κόστος	Βέλτιστο
1	700	48	1	5	0.5	11.103,8	11014
2	700	48	1	5	0.9	11.066,4	10927
3	700	48	1	7	0.9	11.157,6	11060

6. ΣΥΜΠΕΡΑΣΜΑ

Συμπεραίνουμε από τους πίνακες στα παραπάνω προβλήματα παίρνουμε πολύ καλά αποτελέσματα ακόμη και το βέλτιστο με τις παραπάνω τιμές στις παραμέτρους, παρόλο που το σύστημα μυρμηγκιών (Ant System) είναι χρήσιμο για την ανακάλυψη καλών ή βέλτιστων λύσεων για μικρά TSP (έως 30 πόλεις), ο χρόνος που απαιτείται για την εύρεση τέτοιων αποτελεσμάτων το καθιστούσε ανέφικτο για μεγαλύτερα προβλήματα. Πραγματοποιήθηκαν τρεις βασικές αλλαγές για να βελτιωθεί η απόδοσή του, οι οποίες οδήγησαν στον ορισμό του AntColonySystem (ACS), που παρουσιάζεται στην επόμενη ενότητα

Ο **αλγόριθμος Σύστημα Αποικίας Μυρμηγκιών** (Ant Colony System(ACS)) στον οποίο έχει προστεθεί μια παράμετρος q_0) η οποία δίνει μία ισορροπία μεταξύ διασποράς της αναζήτησης και εντατικοποίησης της αναζήτησης. Ένα μυρμήγκι k που βρίσκεται στην πόλη i θα επιλέξει να πάει στην πόλη j σύμφωνα με την εξίσωση:

$$j = \begin{cases} argmax_{u \in J_i^k}[(\tau_{iu}(t))(n_i j)^\beta], εάν q \leq q_0 \\ J αλλιώς \end{cases} (5.1)$$

Όπου το q είναι μία τυχαία μεταβλητή στο διάστημα (0,1) και το J είναι μια πόλη που επιλέγεται τυχαία με τη χρήση της εξίσωσης πιθανοτήτων:

$$p_{iJ} = \frac{[\tau j][n_i j]^\beta}{\sum_{l=1}^{M}[\tau_{il}][n_{il}]^\beta} \text{ (5.2)}$$

Με αυτό τον τρόπο αν επιλέγει η περίπτωση όπου το $q \geq q_0$ τότε η επιλογή γίνεται με τον ίδιο τρόπο που γίνεται και στο βασικό αλγόριθμο πράγμα που βοηθάει στην διασπορά των λύσεων. Ενώ αν επιλέγει το $q \leq q_0$ τότε τα μυρμήγκια έχουν την δυνατότητα να εξερευνήσουν περιοχές γύρω από το ένα τοπικό βέλτιστο που δεν έχουν εξερευνηθεί πλήρως, γεγονός που βοηθάει στην εντατικοποίηση της αναζήτησης.

Η άλλη αλλαγή είναι στην φάση ενημέρωσης της φερομόνης. Κάθε μυρμήγκι αρχικά ενημερώνει τη φερομόνη των τόξων που χρησιμοποίησε στην κατασκευή των λύσεων του βάση του τύπου:

$$\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t) + \rho\tau_0 \text{ (5.3)}$$

Όταν όλα τα μυρμήγκια έχουν ολοκληρώσει την διαδρομή τους τότε το καλύτερο μυρμήγκι αφήνει επιπλέον φερομόνη στην διαδρομή του βάση της εξίσωσης:

$$\tau_{ij}(t) = (1-\rho)\tau_{ij}(t) + \rho\Delta\tau_{ij} \quad (5.4)$$

Η Τρίτη αλλαγή είναι η χρήση μια λίστας υποψηφίων. Στην λίστα αυτή αποθηκεύονται οι πλησιέστεροι γείτονες του κάθε μυρμηγκιού,βάση των αποστάσεων τους και το κάθε μυρμήγκι θα επιλέξει την πλησιέστερη του πόλη βάσει της λίστας. Ένα μυρμήγκι θα επιλέξει μια πόλη εκτός της λίστας μόνο αν η λίστα έχει εξερευνηθεί. Σε αυτή την περίπτωση το μυρμήγκι θα επιλέξει με βάση την εξίσωση(5.2).

7. ΠΑΡΑΡΤΗΜΑ

7.1. Main.py

```
from AntSystem.ReadData import ReadData
from AntSystem.Colony import Colony


def main():
    read_data = ReadData("../data/ulysses16.xml")
    data = read_data.get_tree()
    dimension = ReadData.get_dimension(data)
    graph = ReadData.create_graph(data, dimension)
    # colony = Colony(dimension, a_graph, number_of_ants, alpha, beta,
evaporation_rate, number_of_cycles)
    print(read_data.get_name_of_file())
    colony = Colony(dimension, graph, 30, 0.7, 0.7, 0.5, 100)
    colony.run()

if __name__ == "__main__":
    main()
```

7.2. ReadData.py

```
import xml.etree.ElementTree as ET
import numpy as np


class ReadData:
    """
    Κλάση ReadData η οποία θα διαβάζει τα προβλήματα (TSP) σε μορφή *.xml
και τα μετατρέπει σε πίνακα (n*n) n = διάσταση
    προβλήματος.
    Τα προβλήματα μπορούν να κατέβουν από τον ιστότοπο:
    http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-
TSPLIB/instances/.
    Πληροφορίες για τη δομή του xml στον ιστότοπο:
    http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/XML-
TSPLIB/Description.pdf

    -self.name_of_file: Το όνομα του αρχείου.
```

```
    -self.tree:Το δέντρο του xml
    """

    def __init__(self, name_of_file):
        """
        Ορίζει το όνομα του αρχείου *.xml
        :param name_of_file: το όνομα του αρχείου.
        """
        self.name_of_file = name_of_file
        try:
            self.tree = ET.parse(name_of_file)
        except FileNotFoundError:
            print("Το αρχείο δεν υπάρχει")

    def set_name_of_file(self, name_of_file):
        """
        Ορίζει το όνομα του αρχείου *.xml
        :param name_of_file: το όνομα του αρχείου.
        """
        self.name_of_file = name_of_file

    def set_tree(self, name_of_file):
        """
        Διάβασμα αρχείου *.xml και επιστροφή του δέντρου.
        :param name_of_file:Το όνομα του αρχείου
        :return:Επιστοφή δέντρου
        """
        try:
            self.tree = ET.parse(name_of_file)
        except FileNotFoundError:
            print("Το αρχείο δεν υπάρχει")
        return self.tree

    def get_name_of_file(self):
        """
        Επιστροφή του ονόματος του αρχειου.
        :return: Όνομα αρχείου
        """
        return self.name_of_file

    def get_tree(self):
        """
        Επιστροφή του δέντρου του xml.
        :return: Επιστροφή του δέντρου του xml.
        """
        return self.tree

    @staticmethod
    def get_dimension(data):
        """
            Μέθοδος που επιστέφει την διάσταση του προβλήματος,
            Κάθε αρχείο .xml περιέχει το tag "vertex" όσα "vertex" έχει το
αρχείο τόσοι είναι και οι κόμβοι άρα και η διάσταση
            του προβλήματος είναι οι κόμβοι.
            :param data:Το δεδομένα του αρχείου.
            :return:Επιστρέφει την διάσταση του προβλήματος
            """
        count = 0
        tree = data
        root = tree.getroot()
```

```
        # Για όλο το αρχείο όταν βρίσκει την λέξη "vertex" αύξησε τον
μετρητή κατά 1.
        for vertex in root.iter('vertex'):
            count += 1

        return count

    @staticmethod
    def create_graph(data, dimension):
        """
        Δημιουργία του γράφου σε αυτή την μέθοδο από το αρχείο βρίσκουμε τα
κόστη και να εισχωρούνται σε ένα πίνακα d*d
        οι διαγώνιες τιμές είναι 0
        Απο το αρχείο xml από το tag edge μπορούμε να πάρουμε τα κόστη.
        :param data: τα δεδομένα του αρχείου
        :param dimension: Διάσταση προβλήματος
        :return:
        """
        tree = data
        root = tree.getroot()
        # Δημιουργία πίνακα dimension * dimension γεμάτος με 0
        arr = np.zeros((dimension, dimension))
        # Ορίζω τον δείκτη της γραμμής με 0
        row = 0
        # Ορίζω τον δείκτη της στήλης με 0
        column = 0
        # Για κάθε tag edge:
        for edge in root.iter('edge'):
            """
            Όταν ο δείκτης την γραμμής είναι ίδιος με τον δείκτη της στήλης
            τότε στον πίνακα γίνεται εισχώρηση του 0 και ο δείκτης της
στήλης αυξάνεται κατά 1
            """
            if row == column:
                arr[row][column] = 0
                column += 1

            #  Όταν ο δείκτης της γραμμής και της στήλης δεν είναι όμοιοι
τότε εισχωρείτε το κόστος
            if row != column:
                # Από το tag 'vertex' πάρε το attribute 'cost' (ένα ένας
πραγματικός αριθμός)
                arr[row][column] = edge.get('cost')
                column += 1
                # Όταν ο δείκτης της στήλης φτάσει στο τέλος τοτε αλλάζουμς
γραμμή αυξάνοντας κατά 1 και μηδενίζεται η
                # στήλη
            if column == dimension:
                column = 0
                row += 1
            # if row == dimension:
            #     break
        return arr

# read_data = ReadData("data/gr17.xml")
# # read_data.set_name_of_file("data/att48.xml")
# name = read_data.get_name_of_file()
#
# file = ReadData.read_file(name)
# dimesion = ReadData.get_dimesion(file)
# print(dimesion)
```

```
# graph = ReadData.create_graph(file, dimesion)
# print(graph)
```

7.3. Ant.py

```
class Ant:
    """
    Κλάση Ant περιέχει όλα τα χαρακτηριστικά του μυρμηγκιού
    - self.ant_id:Το id που έχει το μυρμήγκι(integer).
    - self.starting_town: Η αρχική πόλη από τον οποία θα ξεκινήσει την
διαδρομή το μυρμήγκι(integer).
    - self.located_town: Είναι η πόλη που βρίσκεται το μυρμήγκι(integer).
    - self.allowed_town: Είναι μια λίστα με τις πόλεις που επιτρέπεται το
μυρμήγκι να επισκεφτεί(tabu list).
    - self.tour: Είναι μια λίστα με τις πόλεις που έχει επισκεφτεί το
μυρμήγκι(list).

    :param ant_id: Το id του μυρμηγκιού (integer).
    :param starting_town: Η αρχική πόλη (integer)
    """

    # Κατασκευαστής
    def __init__(self, ant_id, starting_town):
        self.ant_id = ant_id
        self.starting_town = starting_town
        # Στον κατασκευαστή η τρέχων πόλη είναι και η αρχική πόλη.
        self.located_town = starting_town
        # Κενή λίστα που θα περιέχει όλες τις πόλεις που επιτρέπεται να
επισκεφτεί το μυρμήγκι.
        self.allowed_towns = list()
        # Κενή λίστα με τις πόλεις που έχει επισκεφτεί το μυρμήγκι.
        self.tour = list()

    def set_ant_id(self, ant_id):
        """
        Ορίζει το id του μυρμηγκιού.
        :param ant_id: integer
        """
        self.ant_id = ant_id

    def set_starting_town(self, starting_town):
        """
        Ορίζει την αρχική πόλη
        :param starting_town: integer
        """
        self.starting_town = starting_town

    def set_located_town(self, located_town):
        """
        Οριζεί την τρέχων πόλη.
        :param located_town: integer
        """
        self.located_town = located_town

    def set_allowed_towns(self, allowed_towns):
        """
        Ορίζει τις πόλεις που επιτρέπεται το μυρμήγκι να επισκεφτεί.
        :param allowed_towns: list
```

```
        """
        self.allowed_towns = allowed_towns

    def set_tour(self, town):
        """
        Προσθέτει στην λίστα με την διαδρομή μια πόλη.
        :param town: integer
        """
        self.tour.append(town)

    def get_ant_id(self):
        """
        Επιστρέφει το id του μυρμηγκιού.
        :return: integer
        """
        return self.ant_id

    def get_starting_town(self):
        """
        Επιστρέφει την αρχική πόλη
        :return: integer
        """
        return self.starting_town

    def get_located_town(self):
        """
        Επιστρέφει την τρέχον πόλη.
        :return: integer
        """
        return self.located_town

    def get_allowed_towns(self):
        """
        Επιστροφή λίστας με τους επιτρεπόμενους κόμβους.
        :return: list
        """
        return self.allowed_towns

    def get_tour(self):
        """
        Επιστρέφει την λίστα με την διαδρομή που έχει ακολουθήσει το
μυρμήγκι.
        :return: list
        """
        return self.tour

    @staticmethod
    def initialize_allowed_towns(dimension):
        """
        Αρχικοποίηση των πόλεων που επιτρέπεται να επισκεφτεί το μυρμήγκι.
        :param dimension: Η διάσταση του προβλήματος(integer).
        :return: Μια λίστα με τους κόμβους που μπορεί να επισκεφτεί το
μυρμήγκι.
        """
        allowed_towns = []
        for town in range(0, dimension):
            allowed_towns.append(town)
        return allowed_towns

    @staticmethod
```

```
    def transition_probability(located_town, next_town, alpha, beta,
pheromone, visibility, allowed_towns):
        """
        Σε αυτή την μέθοδο υπολογίζεται η πιθανότητα να πάει ένα μυρμήγκι
απο την πόλη i στην πόλη j :
        p(i|j) = pheromone[i,j] ^ α  *  visibility[i,j] ^ β / Σ
pheromone[i,allowed_towns] ^  *
                 visibility[i,allowed_towns] jE allowed towns
        p(i|j) = 0
        :param located_towns:Η τρέχον πόλη που βρίσκεται το μυρμήγκι
(ακέραιος).
        :param next_town:Η επόμενη πόλη που μπορεί να επισκεφτεί
(ακέραιος).
        :param alpha:Μια παράμετρος πραγματικός αριθμός.
        :param beta:Μια παράμετρος πραγματικός αριθμός.
        :param pheromone:Η φερομόνη στις ακμές μεταξύ των πόλεων(πίνακας).
        :param visibility:Η "ορατότητα" στις ακμές μεταξύ των
πόλεων(πίνακας)
        :param allowed_towns: Οι πόλεις που επιτρέπεται να επισκεφτεί το
μυρμήγκι(Λίστα)
        :return: Πιθανότητα πραγματικός αριθμός.
        """
        # Αν ο επόμενη πόλη βρίσκεται στη λίστα με τις πόλεις που
επιτρέπεται να επισκεφτεί το μυρμήγκι.
        if next_town in allowed_towns:
            # Η φερομόνη στην ακμή από την πόλη i στον j
            pheromone_i_j = pheromone[located_town][next_town] ** alpha
            # "ορατότητα" στην ακμή από την πόλη i στον j
            visibility_i = visibility[located_town][next_town] ** beta
            a = pheromone_i_j * visibility_i

            # Η φερομόνη σε όλες τις ακμές από την πόλη i προς στις πόλεις
που επιτρέπεται
            pheromone_all = pheromone[located_town][allowed_towns] ** alpha
            # Η ορατότητα σε όλες τις ακμές από την πόλη i προς στις πόλεις
που επιτρέπεται
            visibility_all = visibility[located_town][allowed_towns] **
beta
            b = sum(pheromone_all * visibility_all)

            return a / b

        else:
            return 0

    @staticmethod
    def compute_tour_length(ant, a_graph):
        """
        Υπολογισμός διαδρομής κάθε μυρμηγκιού.
        :param ant: Αντικείμενο τύπου Ant (Ant).
        :param a_graph: Γράφος σε μορφή πίνακα (array).
        :return: συνολικό μήκος διαδρομής (integer).
        """
        # Αρχικοποίηση
        length_tour = 0
        # Υπολογισμός από την λίστα με τις ακμές το μήκος της συνολικής
διαδρομής
        for i in range(0, len(a_graph)):
            length_tour +=
a_graph[ant.get_tour()[i][0]][ant.get_tour()[i][1]]
        return length_tour
```

7.4. Colony.py

```
from AntSystem.Ant import Ant
import numpy as np
import random as rand


class Colony:
    """
    Κλάση AntSystem περιέχει όλες τις λειτουργίες την αποικίας για να βρεθεί η συντομότερη διαδρομή στον πρόβλημα TSP
    """

    def __init__(self, dimension, a_graph, number_of_ants, alpha, beta, evaporation_rate, number_of_cycles):
        """
        Στον κατασκευαστή γίνεται η αρχικοποίηση των δεδομένων που είναι απαραίτητα για την λειτουργία της αποικίας.
        - self.dimension: Διάσταση του προβλήματος.
        - self.a_graph: Ο γράφος σε μορφή πίνακα.
        - self.ants: Λίστα με τα μυρμήγκια της αποικίας.
        - self.number_of_ants: Αριθμος μυρμηγκιών.
        - self.alpha: Μια παράμετρος a ελέγχει την σχετική σημασία της φερομόνης.
        - self.beta: Μια παράμετρος β ελέγχει την σχετική σημασία της "ορατότητας".
        - self.evaporation_rate: Ο ρυθμός εξάτμισης της φερομόνης σε κάθε κύκλο
        - self.number_of_cycles: Ο αριθμός κύκλων που θα εκτελέσει η αποικία.
        :param dimension: Διάσταση του προβλήματος.
        :param a_graph: Γράφος σε μορφή πίνακα.
        """
        self.dimension = dimension
        self.a_graph = a_graph
        self.number_of_ants = number_of_ants
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.number_of_cycles = number_of_cycles
        self.ants = list()

    def set_dimension(self, dimension):
        """
        Ορίζει την διάσταση του προβήματος.
        :param dimension: integer
        """
        self.dimension = dimension

    def set_graph(self, a_graph):
        """
        Ορίζει το γράφο σε μορφή πίνακα.
        :param a_graph: array
        """
        self.a_graph = a_graph

    def set_number_of_ants(self, number_of_ants):
        """
        Ορίζει τον αριθμό τον μυρμηγκιών στην αποικία.
        :param number_of_ants: integer
        """
```

```
    def set_alpha(self, alpha):
        """
        Ορίζει την παράμετρο a που ελέγχει την σχετική σημασία της
φερομόνης.
        :param alpha: real
        """
        self.alpha = alpha

    def set_beta(self, beta):
        """
        Ορίζει την παράμετρο β που ελέγχει την σχετική σημασία της
"ορατότητας".
        :param beta: real
        """
        self.beta = beta

    def set_evaporation_rate(self, evaporation_rate):
        """
        Ορίζει τον  ρυθμό εξάτμισης της φερομόνης σε κάθε κύκλο
        :param evaporation_rate: real
        """
        self.evaporation_rate = evaporation_rate

    def set_number_of_cycles(self, number_of_cycles):
        """
        Ορίζει τον αριθμό των κύκλων που θα εκτελέσει η αποικία
        :param number_of_cycles: integer
        """
        self.number_of_cycles = number_of_cycles

    def set_ants(self, ants):
        """
        Ορίζει την λίστα με τα μυρμήγκια της αποικίας.
        :param ants: list
        """
        self.ants = ants

    def get_dimension(self):
        """
        Επιστρέφει την διάσταση του προβλήματος (integer).
        :return: integer
        """
        return self.dimension

    def get_graph(self):
        """
        Επιστρέφει το γράφο σε μορφή πίνακα.
        :return: Πίνακας είναι ο γράφος
        """
        return self.a_graph

    def get_number_of_ants(self):
        """
        Επιστρέφει τον αριθμό τον μυρμηγκιών στην αποικία.
        :return: integer
        """
        return self.number_of_ants()

    def get_alpha(self):
        """
```

```
        Επιστρέφει το α μια παράμετρος που ελέγχει την σχετική σημασία της
φερομόνης.
        :return: real
        """
        return self.alpha

    def get_beta(self):
        """
        Επιστρέφει το β μια παράμετρος που ελέγχει την σχετική ορατότητα
της φερομόνης.
        :return: real
        """
        return self.beta

    def get_evaporation_rate(self):
        """
        Επιστρέφει το ρυθμό εξάτμισης της φερομόνης της αποικίας σε κάθε
κύκλο
        :return: real
        """
        return self.evaporation_rate

    def get_number_of_cycles(self):
        """
        Επιστρέφει τον αριθμό των κύκλων που θα εκτελέσει η αποικία.
        :return: integer
        """
        return self.number_of_cycles

    def get_ants(self):
        """
        Επιστρέφει την λίστα με τα μυρμήγκια της αποικίας.
        :return: list
        """
        return self.ants

    @staticmethod
    def set_visibility(a_graph):
        """
        Ορίζει την "ορατότητα" του γράφου.
        :param a_graph: array
        :return: array
        """
        visibility = 1 / a_graph
        return visibility

    @staticmethod
    def initialize_pheromone(dimension, initial_value):
        """
        Αρχικοποίηση της φερομόνης.
        :param dimension: Η διάσταση του προβλήματος.
        :param initial_value: Αρχική τιμή που θα οριστεί(integer).
        :return: Επιστροφή πίνακα με τις αρχικές τιμές την
φερομόνης(array).
        """
        initial_pheromone = np.zeros((dimension, dimension))
        for row in range(0, dimension):
            for column in range(0, dimension):
                if row == column:
                    initial_pheromone[row][column] = 0
                else:
```

```
                    initial_pheromone[row][column] = initial_value
        return initial_pheromone

    @staticmethod
    def initialize_ants(ants, number_of_ants, dimension):
        """
        Αρχικοποίηση μυρμηγκιών σε τυχαίες αρχικές θέσεις.
        :param ants: Λίστα άδεια
        :param number_of_ants: Ο συνολικός αριθμός των μυρμηγκιών στην
αποίκια
        :param dimension: ακέραιος με την διάσταση του
προβλήματος(integer).
        :return: Λίστα αντικειμένων τύπου Ant με τα μυρμήγκια της αποικίας
αρχικοποιημένα στις αρχικές θέσεις(list).
        """
        for ant_id in range(0, number_of_ants):
            # Τυχαίος αριθμός είναι  dimension -1 για το ξεκινάει από το 0
            starting_town = rand.randint(0, dimension - 1)
            # Δημιουργία ενός αντικειμένου τύπου Ant ορίζω το id του
μυμρμηγκιού, και την τυχαία αρχική πόλη
            ants.append(Ant(ant_id, starting_town))
            # Για κάθε μυρμήγκι ορίζω τους κόμβους που επιτρέπεται να πάει

ants[ant_id].set_allowed_towns(Ant.initialize_allowed_towns(dimension))
            # Αφαιρώ από την λίστα με τους επιτρεπόμενους κόμβους τον
αρχικό κόμβο

ants[ant_id].get_allowed_towns().remove(ants[ant_id].get_starting_town())
        return ants

    @staticmethod
    def built_tour(ant, alpha, beta, pheromone, visibility, dimension):
        """
        Μέθοδος που κατασκευάζει την διαδρομή που θα ακολουθήσει το
μυρμήγκι.
        :param ant: Αντικείμενο τύπου Ant.
        :param alpha: Μια σταθερή παράμετρος α.
        :param beta: Μια σταθερή παράμετρος β.
        :param pheromone: Η φερομόνη στις ακμές μεταξύ των πόλεων.
        :param visibility: Η "ορατότητα" στις ακμές μεταξύ των πόλεων.
        :param dimension: Η διάσταση του προβλήματος
        :return: Ένας πίνακα με την διαδρομή που ακολούθησε το μυρμήγκι
        """
        # Μέχρι να αδειάσει η λίστα με τις πόλεις που επιτρέπεται να πάει
το μυρμήγκι
        # είναι το σύνολο των πόλεων -1 γιατί ήδη έχουμε αφαιρέσει από την
λίστα την αρχική πόλη.
        global next_town
        for iteration in range(0, dimension - 1):
            located_town = ant.get_located_town()
            allowed_towns = ant.get_allowed_towns()
            max_probability = 0
            # Για κάθε τιμή της λίστας
            for index in range(0, len(allowed_towns)):
                # Υπολογισμός πιθανότητας να μετακινηθεί από την πόλη που
βρίσκεται αυτή την στιγμή σε μια άλλη πόλη που
                # βρίσκεται στην λίστα με τις πόλεις που επιτρέπεται να
μετακινηθεί.
                probability = Ant.transition_probability(located_town,
allowed_towns[index], alpha, beta, pheromone,
                                                         visibility,
```

```
                                                        allowed_towns)
                if probability > max_probability:
                    max_probability = probability
                    next_town = allowed_towns[index]

            # Προθέτει στην λίστα την πόλη με την μεγαλύτερη πιθανότητα.
            ant.set_tour((ant.located_town, next_town))
            # Γίνεται τρέχον πόλη η πόλη με την μεγαλύτερη πιθανότητα.
            ant.set_located_town(next_town)
            # Την αφαιρεί από την λίστα με τις επιτρεπόμενες πόλεις
            ant.get_allowed_towns().remove(ant.get_located_town())
        # Προσθέτει στην λίστα με την διαδρομή την αρχική πόλη
        ant.set_tour((ant.get_located_town(), ant.get_starting_town()))
        # Γίνεται τρέχον πόλη η αρχική.
        ant.set_located_town(ant.get_starting_town())
        return ant.get_tour()

    @staticmethod
    def quantity_per_unit_of_length_of_pheromone(ants, tour_length,
quantity):
        """
        Υπολογισμός της ποσότητας φερομόνης ανά μονάδα συνολικού μήκους
διαδρομής
        κάθε ακμής που την  επισκέφτηκε κάθε μυρμήγκι.

        :param quantity: Πίνακας με την ποσότητα φερομόνης… σε κάθε ακμή
        :param ants: Λίστα με τα μυρμήγκια που βρίσκονται στην αποικία.
        :param tour_length:Συνολικό μήκος διαδρομής
        :return: ποσότητας φερομόνης ανά μονάδα συνολικού μήκους διαδρομής
σε κάθε ακμή.
        """
        for i in range(0, len(quantity)):
            for j in range(0, len(quantity)):
                for k in range(len(ants)):
                    if (i, j) in ants[k].get_tour():
                        d = 1 / tour_length[k]
                    else:
                        d = 0
                    quantity[i][j] += d
        return quantity

    @staticmethod
    def update_pheromone(pheromone, quantity, evaporation_rate):
        """
        Ενημέρωση φερομόνης στις ακμές του γράφου

        :param evaporation_rate: Ρυθμός εξάτμισης φερομόνης(real)
        :param pheromone: Πίνακας με τις τιμές της φερομόνης πάνω στις
ακμές του γράφου(array)
        :param quantity: Πίνακας με τις τιμές της φερομόνης ανά μονάδα
συνολικού μήκους διαδρομής
            πάνω στις ακμές του γράφου(array)
        :return: Πίνακας με την ενημερωμένη φερομόνη(array).
        """
        for i in range(0, len(pheromone)):
            for j in range(0, len(pheromone)):
                pheromone[i][j] = evaporation_rate * pheromone[i][j] +
quantity[i][j]
        return pheromone

    def run(self):
```

```
        """
        Τρέχει την αποικία.
        """
        # Αρχικοποίηση
        # Αρχικοποιώ την συντομότερη διαδρομή άπειρο.
        min_length_all_time = np.inf
        min_tour = 0
        # Αρχικοποίηση μίας λίστας που θα περιέχει το μήκος των διαδρομών
των μυρμηγκιών.
        length_tours = list()
        # Αρχικοποίηση της αρχικής φερομόνης στις ακμές μεταξύ των πόλεων.
        initial_pheromone = Colony.initialize_pheromone(self.dimension, 1)
        pheromone = initial_pheromone
        # Αρχικοποίηση ενός πίνακα n * n διαστάσεων που θα περέχει μετα τον
υπολογισμό την ποσότητας φερομόνης ανά
        # μονάδα συνολικού μήκους διαδρομής κάθε ακμής που την  επισκέφτηκε
κάθε μυρμήγκι.
        quantity = np.zeros((self.dimension, self.dimension))
        # Αρχικοποίηση ενός πίνακα n * n που θα περιέχει μετά τον
υπολογισμό την "ορατότητα" στις ακμές
        # μεταξύ των πόλεων.
        visibility = Colony.set_visibility(self.a_graph)
        # Αρχικοποίση μυρμήγκιών στην αποικία.
        ants = Colony.initialize_ants(self.ants, self.number_of_ants,
self.dimension)

print("============================Initialize============================
==========")
        # for k in range(0, self.number_of_ants):
        #     print("ant id:", ants[k].get_ant_id())
        #     print("starting node:", ants[k].get_starting_town())
        #     print("allowed nodes:", ants[k].get_allowed_towns())

        #
print("=========================TOURS=================================
============")
        # Για κάθε κύκλο.
        nc = 0
        while nc < self.number_of_cycles:
            # Για κάθε μυρμήγκι.
            for k in range(0, self.number_of_ants):
                # print("ant", k)
                ant = ants[k]
                # Δημιουργία διαδρομής του μυρμηγκιού
                tours = Colony.built_tour(ant, self.alpha, self.beta,
pheromone, visibility, self.dimension)
                # print(tours)
                # Υπολογισμός κόστους της διαδρομής και εισαγωγή στην λίστα
                length_tours.append(Ant.compute_tour_length(ant,
self.a_graph))
                # print("length:", length_tours)
            # Υπολογισμός ποσότητας φερομόνης ανά μονάδα συνολικού μήκους
διαδρομης κάθε ακμής
            quantity =
Colony.quantity_per_unit_of_length_of_pheromone(self.ants, length_tours,
quantity)
            # Ενημέρωση φερομόνης.
            pheromone = Colony.update_pheromone(pheromone, quantity,
self.evaporation_rate)

            min_length = min(length_tours)
```

```
            if min_length < min_length_all_time:
                min_length_all_time = min_length
                min_tour =
ants[length_tours.index(min(length_tours))].get_tour()
                print(min_length_all_time)
                print(min_tour)

            for k in range(0, self.number_of_ants):

ants[k].set_allowed_towns(Ant.initialize_allowed_towns(self.dimension))
                ants[k].get_tour().clear()
                length_tours.clear()
                ants[k].set_located_town(ants[k].get_starting_town())

ants[k].get_allowed_towns().remove(ants[k].get_starting_town())
            nc += 1
        print("Parameters:")
        print("alpha", self.alpha)
        print("beta", self.beta)
        print("evaporation_rate", self.evaporation_rate)
        print("Minimum length:", min_length_all_time, ":", min_tour)
```

8. ΒΙΒΛΙΟΓΡΑΦΙΑ

1. "Αλγόριθμοι βελτιστοποίησης αποικιών των μυρμηγκιών." *Βικιπαίδεια*, 21 Dec. 2020. *Wikipedia*, https://el.wikipedia.org/w/index.php?title=%CE%91%CE%BB%CE%B3%CF%8C%CF%81%CE%B9%CE%B8%CE%BC%CE%BF%CE%B9_%CE%B2%CE%B5%CE%BB%CF%84%CE%B9%CF%83%CF%84%CE%BF%CF%80%CE%BF%CE%AF%CE%B7%CF%83%CE%B7%CF%82_%CE%B1%CF%80%CE%BF%CE%B9%CE%BA%CE%B9%CF%8E%CE%BD_%CF%84%CF%89%CE%BD_%CE%BC%CF%85%CF%81%CE%BC%CE%B7%CE%B3%CE%BA%CE%B9%CF%8E%CE%BD&oldid=8586075.

2. "Ant Colony Optimization Algorithms." *Wikipedia*, 30 Mar. 2021. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Ant_colony_optimization_algorithms&oldid=1015016073.

3. "Travelling Salesman Problem." *Wikipedia*, 19 Apr. 2021. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1018647826.

4. "Metaheuristic Algorithms: A Comprehensive Review." *Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications*, Jan. 2018, pp. 185–231. *www.sciencedirect.com*, doi:10.1016/B978-0-12-813314-9.00010-4.

5. "Swarm Intelligence." *Wikipedia*, 25 Mar. 2021. *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Swarm_intelligence&oldid=1014145492.

6. M. Dorigo, V. Maniezzo, et A. Colorni, *Ant system: optimization by a colony of cooperating agents*, IEEE Transactions on Systems, Man, and Cybernetics--Part B , volume 26, numéro 1, pages 29-41, 1996.

7. M. Dorigo et L.M. Gambardella, *Ant Colony System : A Cooperative Learning Approach to the Traveling Salesman Problem*, IEEE Transactions on Evolutionary Computation, volume 1, numéro 1, pages 53-66, 1997.

8. Ιωάννης Μαρινάκης,Μαγδαλινή Μαρινάκη,Νικόλαος Φ. Ματσατσινης και Κωνσταντίνος Ζμπουνίδης Μεθευρετικοί και Εξελικτικοί αλγόριθμοι σε Πρβληματα Διοικητηκής Επιστήμης.