**ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ**
**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ**

# Ανάπτυξη αλγορίθμων μετακίνησης αντικειμένων σε κατανεμημένα συστήματα

**Πανάγου Ναταλία**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

**Υπεύθυνος**

**Λουκόπουλος Αθανάσιος**

**Επίκουρος Καθηγητής**

**Λαμία, Ιούνιος 2017**

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ**
**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗ ΒΙΟΙΑΤΡΙΚΗ**

# Τίτλος: Ανάπτυξη αλγορίθμων μετακίνησης αντικειμένων σε κατανεμημένα συστήματα

**Πανάγου Ναταλία**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

**Επιβλέπων**
**Λουκόπουλος Αθανάσιος**
**Επίκουρος Καθηγητής**

**Λαμία, Ιούνιος 2017**

Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. *Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί* ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. *Δέχομαι ότι η αυτολεξεί* ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. *Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια*

*4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.*

Ημερομηνία: ……/…../20……

Ο – Η Δηλ.

(Υπογραφή)

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.

# Τίτλος: Ανάπτυξη αλγορίθμων μετακίνησης αντικειμένων σε κατανεμημένα συστήματα

**Πανάγου Ναταλία**

**Τριμελής Επιτροπή:**

Λουκόπουλος Αθανάσιος, Επίκουρος Καθηγητής (επιβλέπων)

Κακαρούντας Αθανάσιος, Επίκουρος Καθηγητής

Πλαγιανάκος Βασίλειος, Αναπληρωτής Καθηγητής

# Περιεχόμενα

# Περίληψη

Τα Ασύρματα Δίκτυα Αισθητήρων(ΑΔΑ) αποτελούν ένα συνεχώς εξελισσόμενο πεδίο με μια πληθώρα εφαρμογών όπως είναι η παρατήρηση βιοτόπων, ο εντοπισμός αντικειμένων, η εποπτεία περιβάλλοντος, η υγειονομική περίθαλψη, η παρακολούθηση κτιρίων κ.τ.λ.

Λόγω των ιδιαίτερων χαρακτηριστικών τους καθώς και των περιορισμών από τους οποίους πάσχουν τα ΑΔΑ καθιστούν την ανάπτυξη εφαρμογών περίπλοκη. Η ύπαρξη, λοιπόν, ενός ενδιάμεσου λογισμικού (middleware) κρίνεται απαραίτητη για να γεμίσει το κενό μεταξύ των απαιτήσεων υψηλού επιπέδου των διάφορων εφαρμογών που τρέχουν σε ΑΔΑ και την πολυπλοκότητα των διαφορετικών λειτουργιών στο υλικό του αισθητήριου κόμβου. Μία από τις αρχιτεκτονικές που έχουν χρησιμοποιηθεί για την ανάπτυξη ενός αποδοτικού middleware είναι αυτή των κινητών πρακτόρων (mobile agents). Η προσέγγιση αυτή επιτρέπει την ανάπτυξη της εφαρμογής σαν ένα σύνολο από agents, οι οποίο αφού αρχικοποιηθούν σε έναν κόμβο, έχουν τη δυνατότητα να μετακινούνται μέσα στο δίκτυο. Αυτό παρέχει τη δυνατότητα στο middleware να αποφασίσει μία καλύτερη τοποθέτηση των agents επιτυγχάνοντας έτσι τη μείωση του συνολικού φόρτου του δικτύου που οφείλεται στην μεταξύ τους επικοινωνία.

Ένας αλγόριθμος που έχει προταθεί για τη μετακίνηση και τοποθέτηση των agents είναι ο AMA[1] (Agent Migration Algorithm), ο οποίος μετακινεί έναν agent την φορά.

Σκοπός της συγκεκριμένης πτυχιακής εργασίας είναι η παρουσίαση του GRAL[2] (Grouping Algorithm), ο οποίος μετακινεί ομάδες από agents. Για την αξιολόγηση του αλγορίθμου υλοποιήθηκε σε γλώσσα C πρόγραμμα που προσομοιώνει την ομαδοποίηση και μετακίνηση των πρακτόρων.

[1] N. Tziritas, T. Loukopoulos, S. Lalis and P. Lampsas, "On Deploying Tree Structured Agent Applications in Embedded Systems" in Proc. EUROPAR 2010.

[2] N. Tziritas, S. Lalis, T. Loukopoulos and P. Lampsas, "GRAL: A Grouping Algorithm to Optimize Application Placement in Wireless Embedded Systems" in 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011).

Στο Κεφάλαιο 1 γίνεται μια εισαγωγή στα Ασύρματα Δίκτυα Αισθητήρων, καθώς και στο Ενδιάμεσο Λογισμικό που συνοδεύει την ανάπτυξη εφαρμογών σε αυτά. Δίνεται ο ορισμός του προβλήματος της τοποθέτησης των πρακτόρων, καθώς και το μοντέλο της εφαρμογής και του δικτύου πάνω στο οποίο αναπτύσσεται, και αναφέρονται παρόμοιες με την παρούσα μελέτες.

Στο Κεφάλαιο 2 περιγράφεται ο αλγόριθμος GRAL.

Στο Κεφάλαιο 3 περιγράφεται η υλοποίηση της προσομοίωσης και παρατίθενται τα αποτελέσματα που έδωσε.

Στο Κεφάλαιο 4 παρουσιάζονται τα συμπεράσματα που εξάγονται σχετικά με την απόδοση του αλγορίθμου και γίνεται μια πρόταση για μελλοντική μελέτη.

Ο κώδικας της προσομοίωσης παρατίθεται στο Παράρτημα.

# Κεφάλαιο 1

## 1.1 Ασύρματα Δίκτυα Αισθητήρων

Τα «έξυπνα» περιβάλλοντα αποτελούν το επόμενο βήμα ανάπτυξης στις κατοικίες, στην αυτοματοποίηση των μεταφορικών συστημάτων και στη βιομηχανία. Όπως κάθε οργανισμός που αισθάνεται, έτσι και το «έξυπνο» περιβάλλον στηρίζεται κυρίως στη συλλογή ενός συνόλου από αισθητήρια στοιχεία του πραγματικού κόσμου που το περιβάλλει. Αισθητήρια δεδομένα προέρχονται από πολλαπλούς κόμβους διαφορετικών λειτουργιών, οι οποίοι βρίσκονται κατανεμημένοι στο χώρο. Τα δεδομένα που ένα «έξυπνο» περιβάλλον συλλέγει από τον πραγματικό κόσμο συντελούν είτε σε εσωτερικές είτε σε εξωτερικές λειτουργίες του. Οι πληροφορίες που ένα τέτοιο περιβάλλον χρειάζεται παρέχονται σε αυτό από κατανεμημένα Ασύρματα Δίκτυα Αισθητήρων (ΑΔΑ).

Τα ΑΔΑ, κάποιες φορές αποκαλούμενα Ασύρματα Δίκτυα Αισθητήρων/Ενεργοποιητών (WSAN -Wireless Sensor/Actuator Networks) αποτελούνται από ένα σύνολο αισθητήριων κόμβων, οι οποίοι βρίσκονται κατανεμημένοι στο χώρο και έχουν την ικανότητα να συλλέγουν πληροφορίες που αφορούν φυσικές ή περιβαλλοντικές συνθήκες όπως είναι η θερμοκρασία, η ατμοσφαιρική πίεση, η υγρασία, η φωτεινότητα, η ανίχνευση χημικών στοιχείων, η πίεση του αίματος καθώς και μια σειρά από πολλές άλλες παραμέτρους. Οι αισθητήριοι αυτοί κόμβοι δρώντας συνεργατικά μπορούν να μεταφέρουν τις πληροφορίες που συλλέγουν σε μια συγκεκριμένη τοποθεσία χωρίς να χρησιμοποιούνται καλώδια.

Η πρόοδος που έχει σημειωθεί στην τεχνολογία των μικρό-ηλεκτρομηχανικών συστημάτων (ΜΗΜΣ), στην ασύρματη επικοινωνία και στα ψηφιακά ηλεκτρονικά έχει δώσει την δυνατότητα ανάπτυξης χαμηλού κόστους, χαμηλής κατανάλωσης ενέργειας και πολλών λειτουργιών αισθητήριων κόμβων οι οποίοι είναι μικροί σε μέγεθος και έχουν την ικανότητα να επικοινωνούν σε μικρές αποστάσεις.

Κάθε αισθητήριος κόμβος αποτελείται από έναν ή περισσότερους αισθητήρες, έναν μικροεπεξεργαστή, έναν πομποδέκτη (ο οποίος προσφέρει ασύρματη συνδεσιμότητα) και μια μονάδα ενέργειας η οποία είναι συνήθως μια μπαταρία.

## 1.2 Ενδιάμεσο Λογισμικό και Κινητοί Πράκτορες στα ΑΔΑ

Ο όρος «ενδιάμεσο λογισμικό» αναφέρεται στο στρώμα λογισμικού το οποίο κρύβει την υποκείμενη πολυπλοκότητα και την ετερογένεια του υλικού των κόμβων, καθώς και της πλατφόρμας του δικτύου. Έχει τον ρόλο του μεταφραστή που γεμίζει το κενό μεταξύ των απαιτήσεων υψηλού επιπέδου των διάφορων εφαρμογών που αναπτύσσονται σε ένα ΑΔΑ και την πολυπλοκότητα των διαφορετικών λειτουργιών στο υλικό του αισθητήριου κόμβου. Οι περιορισμοί σε πόρους, οι συχνές αλλαγές στην τοπολογία του δικτύου καθώς και τα ενσωματωμένα λειτουργικά συστήματα χαμηλού επιπέδου συνθέτουν την πολυπλοκότητα των λειτουργιών σε ένα Ασύρματο Δίκτυο Αισθητήρων.

Η τεχνολογία των κινητών πρακτόρων αποτελεί μια από τις πολλές προσεγγίσεις που αφορούν την κατασκευή ενδιάμεσου λογισμικού για την ανάπτυξη, τη συντήρηση και την εκτέλεση εφαρμογών στα ΑΔΑ.

Στην επιστήμη των υπολογιστών ένας κινητός πράκτορας (mobile agent) είναι ένας συνδυασμός λογισμικού και δεδομένων ο οποίος είναι σε θέση να μετακινείται αυτόνομα από έναν υπολογιστή σε έναν άλλον και να συνεχίζει την εκτέλεσή του στον προορισμό.

Ένας κινητός πράκτορας αποτελεί ουσιαστικά μια διεργασία η οποία μπορεί να μεταφέρει την κατάστασή της από ένα σύστημα σε ένα άλλο διατηρώντας τα δεδομένα της άθικτα και διαθέτοντας την ικανότητα να εκτελείται καταλλήλως στο καινούριο περιβάλλον. Όταν ένας κινητός πράκτορας αποφασίσει να μετακινηθεί αποθηκεύει την κατάσταση στην οποία βρίσκεται, την μεταφέρει στον προορισμό του και συνεχίζει την εκτέλεσή του από αυτή την αποθηκευμένη κατάσταση.

Ενσωματωμένα συστήματα κινητών πρακτόρων όπως το POBICOS[8] και το ROVERS [20] δίνουν στον προγραμματιστή την δυνατότητα να σχεδιάσει την εφαρμογή ως ένα σύνολο από συνεργαζόμενους agents. Η τοποθέτηση των agents στους κόμβους γίνεται βάσει των πόρων που απαιτούνται αλλά και παρέχονται. Ωστόσο, η τοποθέτηση αυτή θα πρέπει να λαμβάνει υπόψη και τον φόρτο του δικτύου που οφείλεται στην επικοινωνία των συνεργαζόμενων agents. Κάτι τέτοιο είναι ζωτικής σημασίας για ένα δίκτυο το οποίο αποτελείται από κόμβους που είναι εξοπλισμένοι με μπαταρία και μια κακή τοποθέτηση θα οδηγούσε στην γρήγορη εξάντλησή της.

## 1.3 Το Μοντέλο της Εφαρμογής, το Μοντέλο του Συστήματος και Διατύπωση του Προβλήματος

Σε αυτή την ενότητα περιγράφεται το μοντέλο των εφαρμογών στις οποίες εστιάζουμε, το μοντέλο του συστήματος και διατυπώνεται το Πρόβλημα Τοποθέτησης των Πρακτόρων (Agent Placement Problem-APP).

### *Το μοντέλο της εφαρμογής*

Όπως προαναφέρθηκε, η παρούσα εργασία επικεντρώνεται σε εφαρμογές οι οποίες δομούνται σαν ένα σύνολο από agents οι οποίοι δρουν συνεργατικά με σκοπό την συλλογή και την επεξεργασία δεδομένων.

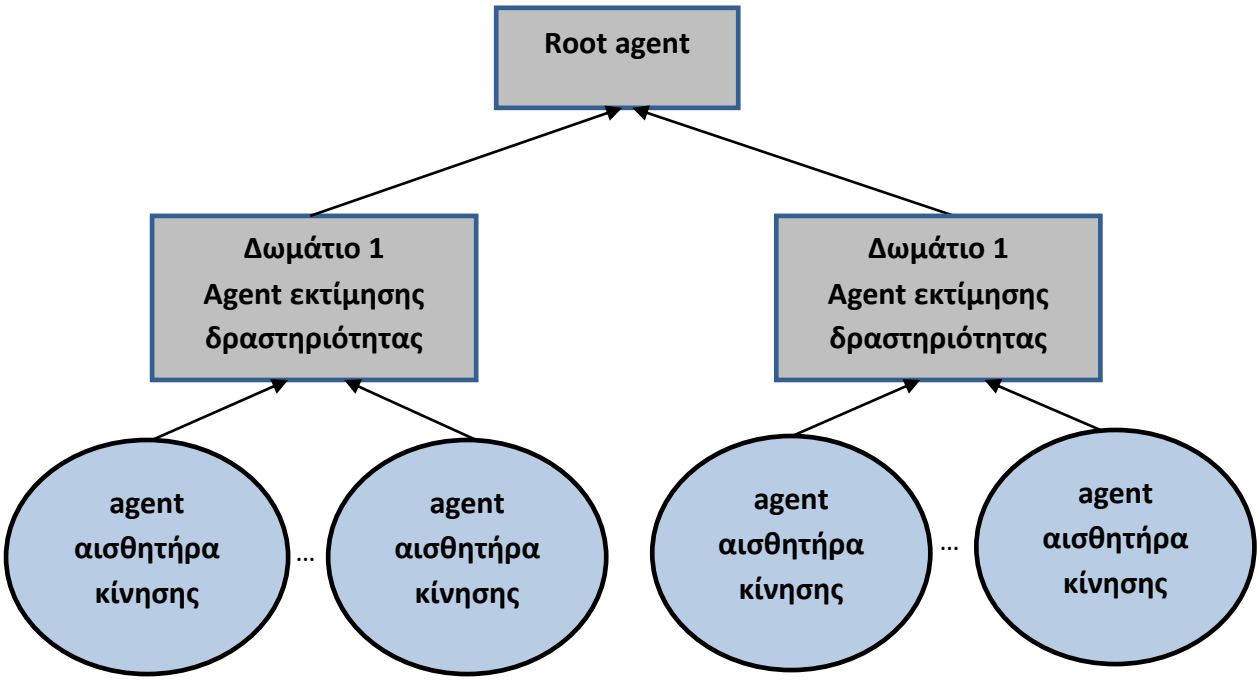
Οι agents αυτοί είναι οργανωμένοι σε μια ιεραρχική, δενδρικού τύπου, δομή όπου κάθε agent μπορεί να επικοινωνεί μόνο με τον πατέρα και τα παιδιά του. Η επικοινωνία γίνεται μέσω ανταλλαγής μηνυμάτων μεταξύ των agents, οι οποίοι διακρίνονται σε δυο κατηγορίες ανάλογα με τον ρόλο που αναλαμβάνουν.

Η πρώτη κατηγορία είναι αυτή των *agents συγκεκριμένων κόμβων (node-specific)*, οι οποίοι βρίσκονται στο κατώτερο επίπεδο της ιεραρχίας. Απαρτίζουν, δηλαδή, τα φύλλα του δένδρου. Ονομάζονται έτσι γιατί εξαρτώνται από την θέση τους μέσα στο δίκτυο, καθώς αλληλεπιδρούν με το περιβάλλον, εκμεταλλευόμενοι τους αισθητήρες που φέρουν οι κόμβοι, με σκοπό την συλλογή πληροφοριών από αυτό. Είναι, επομένως, άρρηκτα συνδεδεμένοι με τους κόμβους οι οποίοι προσφέρουν τους κατάλληλους αισθητήρες ή ενεργοποιητές για να επιτελέσουν οι agents την εργασία για την οποία δημιουργήθηκαν.

Η δεύτερη κατηγορία είναι αυτή των *agents ανεξαρτήτων κόμβων (node-neutral)*. Είναι ανώτεροι σε ιεραρχία και δρουν σαν συνθέτες των πληροφοριών που οι κατώτεροί τους συλλέγουν. Επεξεργάζονται, δηλαδή, τις πληροφορίες αυτές και λαμβάνουν αποφάσεις. Για να επιτευχθούν τέτοιες ενέργειες χρειάζονται πόροι γενικού σκοπού (επεξεργαστική ισχύς, μνήμη) οι οποίοι προσφέρονται από όλους τους κόμβους. Συμπερασματικά, οι agents αυτής της κατηγορίας μπορούν να τοποθετηθούν σε οποιονδήποτε κόμβο προσφέρει επαρκείς πόρους γενικού σκοπού. Από την στιγμή που θα δημιουργηθούν σε έναν κόμβο, μπορούν να μετακινηθούν σε κάποιον άλλο με σκοπό να έρθουν πιο κοντά σε agents με τους οποίους συνεργάζονται.

Ένα παράδειγμα μιας τέτοιας εφαρμογής φαίνεται στην Εικόνα 1 και αφορά ένα σύστημα συναγερμού για μια κατοικία. Η εφαρμογή λειτουργεί εκμεταλλευόμενη τους αισθητήρες κίνησης που βρίσκονται στο χώρο για την ενεργοποίηση του συναγερμού.

Στα φύλλα της δενδρικής δομής βρίσκονται οι *node specific agents*, οι οποίοι τοποθετημένοι στους αισθητήρες κίνησης συλλέγουν δεδομένα από το περιβάλλον. Οι node neutral agents οι οποίοι είναι ανώτεροι σε ιεραρχία, επεξεργάζονται τα δεδομένα αυτά και ενημερώνουν με τη σειρά τους τον root agent, ο οποίος αποφασίζει για την ενεργοποίηση του συναγερμού.




**Εικόνα 1. Παράδειγμα εφαρμογής συστήματος ασφαλείας**

### *Το μοντέλο του συστήματος*

Το δίκτυο πάνω στο οποίο αναπτύσσεται μια εφαρμογή, αποτελείται από κόμβους οι οποίοι διαθέτουν δυνατότητες ανίχνευσης/ενεργοποίησης και επικοινωνούν μέσω κάποιας ασύρματης τεχνολογίας. Ακόμη, οι κόμβοι διαθέτουν μια χωρητικότητα η οποία σχετίζεται με την επεξεργαστική ισχύ, την μνήμη ή ακόμα και το εύρος ζώνης. Η χωρητικότητα αυτή θέτει έναν περιορισμό σχετικά με τον αριθμό των agents που ένας κόμβος μπορεί να φιλοξενήσει. Ένας κόμβος μπορεί να φιλοξενήσει από κανέναν μέχρι και τον μέγιστο αριθμό από agents που ο περιορισμός χωρητικότητα επιβάλλει. Επίσης, τα μονοπάτια μέσα από τα οποία επικοινωνούν οι κόμβοι είναι μοναδικά.

### *Διατύπωση του προβλήματος της τοποθέτησης των πρακτόρων (Agent Placement Problem-APP)*

Έχοντας τοποθετηθεί οι agents της εφαρμογής πάνω στο δίκτυο, χρησιμοποιούν τους πόρους που οι κόμβοι προσφέρουν για να επικοινωνήσουν ανταλλάσοντας μηνύματα μεταξύ τους. Για να θεωρηθεί έγκυρη μια τοποθέτηση των agents πάνω στο δίκτυο των κόμβων, θα πρέπει να τηρούνται οι περιορισμοί χωρητικότητας καθώς επίσης και κάθε agent να ανήκει σε ακριβώς έναν κόμβο. Η τεχνική δρομολόγησης δεδομένων που χρησιμοποιείται επιτρέπει την αναπαραγωγή των μηνυμάτων πάνω σε ένα μονοπάτι. Τα μηνύματα αυτά αναπηδούν από κόμβο σε κόμβο έως ότου φτάσουν στον προορισμό τους (multihop routing) και με αυτό τον τρόπο μπορούν να ανταλλάξουν μηνύματα μεταξύ τους, agents οι οποίοι βρίσκονται σε απομακρυσμένους κόμβους.

Κάθε κόμβος που ενεργοποιείται για να μεταδώσει κάποιο μήνυμα καταναλώνει ενέργεια. Στην περίπτωση οπού δυο συγγενείς agents φιλοξενούνται σε διαφορετικούς κόμβους οι οποίοι βρίσκονται μακριά ο ένας από τον άλλο, για την μετάδοση των μεταξύ τους μηνυμάτων ενεργοποιούνται πάνω στο μονοπάτι που τους χωρίζει ενδιάμεσοι κόμβοι, άσχετοι με τα μηνύματα αυτά, καταναλώνοντας ενέργεια.

Το πρόβλημα της τοποθέτησης των πρακτόρων μπορεί να περιγραφεί ως εξής:

Ξεκινώντας από μια έγκυρη τοποθέτηση των agents μέσα στο δίκτυο και υποθέτοντας ότι η τοποθέτηση αυτή είναι μη βέλτιστη, δηλαδή οι συγγενείς agents βρίσκονται μακριά ο ένας από τον άλλο δημιουργώντας έτσι έναν αυξημένο συνολικό φόρτο, εκτελώντας μια σειρά από μετακινήσεις πρακτόρων, αναμένεται να δημιουργηθεί μια καινούρια έγκυρη τοποθέτηση η οποία να μειώνει κατά το δυνατό τον αρχικό συνολικό φόρτο. Επίσης έγκυρη θεωρείται μια μετακίνηση αν δεν παραβιάζει τις συνθήκες μιας έγκυρης τοποθέτησης και αν ο πράκτορας που μετακινείται είναι πράκτορας ανεξαρτήτων κόμβων (node neutral agent).

## 1.4 Παρόμοιες Μελέτες

Το πρόβλημα της καλύτερης τοποθέτησης με στόχο την ελαχιστοποίηση της κατανάλωσης ενέργειας έχει εμφανιστεί και στο παρελθόν και τρόποι αντιμετώπισής του έχουν προταθεί. Για παράδειγμα, στα [9], [10] και [11] εξετάζεται το πρόβλημα της τοποθέτησης διεργασιών, που επικοινωνούν μεταξύ τους, σε ομογενή περιβάλλοντα με στόχο την μείωση του χρόνου εκτέλεσής τους, ενώ στο [12] το ίδιο πρόβλημα εξετάζεται, αυτή τη φορά όμως σε ετερογενές περιβάλλον. Στο [13], βλέπουμε επίσης το πρόβλημα της κατανομής των διεργασιών αυτή τη φορά σε ένα δίκτυο Torus, με στόχο την μείωση του φόρτου της μεταξύ τους επικοινωνίας αλλά και της συμφόρησης του δικτύου.

Στο [14], οι συγγραφείς έχουν αναπτύξει έναν αλγόριθμο, με στόχο την κατανομή των διεργασιών σε συστήματα που επικοινωνούν με το περιβάλλον, ο οποίος λαμβάνει υπόψη του τις εξωτερικές μεταβλητές κατά την διαδικασία ανάθεσης των διεργασιών και έτσι δεν χρειάζεται κάποια ανακατανομή τους.

Στο [15] εξετάζεται η αποδοχή ενός καινούριου agent σε ένα δίκτυο κόμβων, λαμβάνοντας υπόψη τόσο την διαθέσιμη μνήμη όσο και την μπαταρία του κόμβου, καθώς και η μεγιστοποίηση της ζωής του δικτύου. Και τα δυο αυτά προβλήματα αντιμετωπίζονται με μεταναστεύσεις των agents.

Σε ότι αφορά τα ΑΔΑ στο [16] εξετάζεται επίσης η κατανομή διεργασιών με στόχο την ελαχιστοποίηση της κατανάλωσης ενέργειας, όμως το μοντέλο της εφαρμογής διαφέρει από αυτό της παρούσας εργασίας.

Ακόμη, στο [17] στόχος είναι ο ορισμός ενός βέλτιστου δένδρου διάδοσης δεδομένων κατά μήκος ενός δενδρικού δικτύου και στο [18] χρησιμοποιούνται τεχνικές μετανάστευσης για την συλλογή δεδομένων μέσω της δημιουργίας βέλτιστων μονοπατιών.

Οι προαναφερθείσες μελέτες διαφέρουν όμως από την παρούσα είτε ως προς το πεδίο στο οποίο εφαρμόζονται, είτε ως προς το μοντέλο του δικτύου και των εφαρμογών είτε ως προς τον τρόπο προσέγγισης του προβλήματος.

Παρόμοια με την παρούσα μελέτη μπορεί να θεωρηθεί η [19] , όπου προτείνεται ένα κατανεμημένος αλγόριθμος ο οποίος μετακινεί αντικείμενα ως προς το κέντρο βάρους του φόρτου επικοινωνίας τους, διαφέρει όμως ως προς το γεγονός ότι εκτελείται μετακίνηση ενός αντικειμένου τη φορά και τα αντικείμενα αυτά είναι ανεξάρτητα μεταξύ τους.

Η παρούσα εργασία αφορά ενσωματωμένα συστήματα κινητών πρακτόρων και συγκεκριμένα το POBICOS[8], στο οποίο την τοποθέτηση των agents αναλαμβάνει

το middleware . Υπάρχουν αρκετά ακόμα συστήματα τα οποία υποστηρίζουν κινητό κώδικα, όπως τα Rover [20], Mate [21], Agilla [22], one.world [23], Smart Messages [24], Olympus [25], SensorWar [26], Pushpin [27], Mobile-C [28], MagnetOS [29], Pleiades [30], και DFuse [31], τα περισσότερα όμως από αυτά δεν προσφέρουν αυτόματη τοποθέτηση κώδικα επιβαρύνοντας έτσι του έργο του προγραμματιστή και επίσης κρίνονται ακατάλληλα σε περιπτώσεις που η τοπολογία του δικτύου δύναται να διαφοροποιηθεί. Ακόμα όμως και εκείνα που υποστηρίζουν μια δυναμική τοποθέτηση των agents ( MagnetOS, Pleiades, DFuse) δεν λαμβάνουν υπόψη τους ομάδες από agents όταν εκτελούνται μετακινήσεις αυτών.

# Κεφάλαιο 2

# Ο αλγόριθμος GRAL (Grouping Algorithm)

Σε αυτό το κεφάλαιο παρουσιάζεται ο GRAL για την περίπτωση που οι κόμβοι μπορούν να φιλοξενήσουν οποιονδήποτε αριθμό από agents και άρα δεν υπάρχουν περιορισμοί χωρητικότητας.

Ο αλγόριθμος μετακινεί agents με ομαδοποιημένο τρόπο. Αρχικά, ταυτοποιούνται υπόδενδρα της εφαρμογής των agents σε επίπεδο κόμβου. Κάθε υπόδενδρο συνιστά μια ομάδα από agents και για κάθε τέτοια ομάδα επιλέγεται ένας προορισμός, ο οποίος συνιστά τον πιο επωφελή προς μετανάστευση γειτονικό κόμβο. Δεδομένου του ότι διατίθενται απεριόριστοι πόροι γενικού σκοπού, κάθε κόμβος αξιοποιεί τις πληροφορίες που του παρέχονται τοπικά για την λήψη απόφασης σχετικά με την μετακίνηση του υπόδενδρου των agents. Μια πιο ρεαλιστική προσέγγιση του αλγορίθμου όπου υπάρχουν περιορισμοί χωρητικότητας, θα απαιτούσε ο ελεύθερος χώρος που κάθε κόμβος διαθέτει να εκτιμηθεί.

Η ταυτοποίηση του υποδένδρου, η επιλογή του προορισμού καθώς και ο τρόπος με τον οποίο αποφασίζεται μια επωφελής μετακίνηση παρουσιάζονται αναλυτικότερα παρακάτω.

### *1. Ταυτοποίηση υποδένδρου*

Ένα υπόδενδρο αποτελεί ένα υποσύνολο του δένδρου της εφαρμογής και περιέχει node-neutral agents οι οποίοι φιλοξενούνται στον ίδιο κόμβο και επικοινωνούν μεταξύ τους.

Αν συμβολίσουμε το σύνολο των node-neutral agents ως A και το σύνολο των node-specific agents ως S, τότε ως $a_i$ θα συμβολίζεται κάθε agent της εφαρμογής με $i \in [1, A+S]$.

Στην συνέχεια, κάθε ταυτοποιημένο υπόδενδρο θα συμβολίζεται ως G και κάθε agent που ανήκει σε αυτό ως $a_k$ , $k \in [1,A]$.

## *2. Επιλογή προορισμού*

Αφού ταυτοποιηθούν ένα ή περισσότερα υπόδενδρα G της εφαρμογής, για κάθε ένα από αυτά επιλέγεται ο κόμβος στον οποίο θα μεταφερθούν.

Ο κόμβος στον οποίο ανήκει κάθε υπόδενδρο G θα συμβολίζεται ως $n_i$ .

Κάθε ταυτοποιημένο υπόδενδρο δύναται να μεταναστεύσει μόνο σε γειτονικό κόμβο του δικτύου.

Για την επιλογή του υποψήφιου προς μετανάστευση κόμβου υπολογίζονται τα εξής:

Α. Ο φόρτος που προκύπτει από την επικοινωνία του κάθε ενός από τους agents του G με τους node specific agents που ανήκουν στον ίδιο κόμβο και τον οποίο συμβολίζουμε ως $\boldsymbol{\ell}_{ak,ni}(S)$.

Β. Για κάθε έναν από τους γειτονικούς κόμβους ο φόρτος ο οποίος οφείλεται στην επικοινωνία του κάθε ενός από τους agents του G με απομακρυσμένους agents (node neutral και node specific).

Ως πιθανός προορισμός επιλέγεται ο κόμβος με τον μεγαλύτερο συνολικό φόρτο και στο εξής θα αναφερόμαστε σε αυτόν ως $n_j$ και στον φόρτο επικοινωνίας που προκύπτει για κάθε $a_κ$ σε σχέση με αυτόν ως $\boldsymbol{L}_{ak,nj}$, ενώ οι υπόλοιποι γειτονικοί κόμβοι θα συμβολίζονται ως $n_y$ ,με y ≠ i,j, και ο φόρτος επικοινωνίας που προκύπτει για κάθε $a_κ$ σε σχέση με αυτούς ως $\boldsymbol{L}_{ak,ny}$.

Τελικά, για να είναι μια μεταφορά του G από τον $n_i$ στον $n_j$ επωφελής, θα πρέπει να ισχύει:

$$\sum_{\forall ak \in G} \boldsymbol{L}_{ak,,nj} > \sum_{\forall ak \in G} \boldsymbol{\ell}_{ak,ni}(S) \quad (1)$$

## *3. Υπολογισμός μερικού κέρδους (partial benefit)*

Αφού έχει εντοπιστεί ο πιο επωφελής προορισμός για το υπόδενδρο G, για κάθε έναν από τους agents $a_k$ υπολογίζεται η τιμή του affinity και του partial benefit.

*Υπολογισμός affinity*

Αν συμβολίσουμε το affinity ως $aff_{ak,nj}$, ο τύπος μέσω του οποίου υπολογίζεται είναι ο εξής:

$$aff_{ak,nj} = \boldsymbol{\ell}_{ak,nj} - \boldsymbol{\ell}_{ak,ni}(S) - \sum_{\forall y \neq i,j} \boldsymbol{\ell}_{ak,ny} \quad (2)$$

Το αποτέλεσμα του affinity περιγράφει τον αντίκτυπο που θα έχει η μετανάστευση του $a_k$ από τον $n_i$ στον $n_j$. Αν είναι θετικό, τότε η μετακίνησή του είναι ωφέλιμη.

Αν όλοι οι agents του G έχουν αρνητικό affinity, τότε δεν υπάρχει υποψήφια προς μετανάστευση ομάδα και έτσι αποφασίζεται το G να παραμείνει στον κόμβο $n_i$.

Σε αντίθετη περίπτωση, για κάθε agent του G υπολογίζεται η τιμή του partial benefit.

*Υπολογισμός partial benefit*

Η τιμή του partial benefit υπολογίζεται διαφορετικά για την ρίζα του G, έστω $a_r$ και διαφορετικά για τους υπόλοιπους agents του G, έστω $a_m$.

Για την ρίζα $a_r$, η τιμή του partial benefit προσδιορίζει το όφελος που θα προκύψει αν μόνο ο $a_r$ μεταναστεύσει στον $n_j$, ενώ όλοι οι υπόλοιποι agents του G παραμείνουν στον ni.

Για όλους τους υπόλοιπους agents, $a_m$, το partial benefit προσδιορίζει το όφελος που θα προκύψει αν μεταναστεύσει ο $a_m$ μαζί με τον πατέρα του, έστω $a_v$, ενώ τα παιδιά του παραμείνουν στον $n_i$.

Αν συμβολίσουμε ως $pb_{ar,nj}$ το partial benefit της ρίζας $a_r$ και ως $pb_{am,nj}$ το partial benefit για κάθε έναν από τους υπόλοιπους agents, $a_m$, τότε οι σχέσεις που χρησιμοποιούνται για τον υπολογισμό τους είναι οι εξής:

$$pb_{ar,nj} = aff_{ar,nj} - \boldsymbol{\ell}_{ar,ni}(A) \quad (3)$$

$$pb_{am,nj} = aff_{am,nj} - \boldsymbol{\ell}_{am,ni}(A) + 2 * \boldsymbol{\ell}_{am,av} \quad (4)$$

Ο συμβολισμός $\ell_{ar,ni}$ (A) αναφέρεται στον φόρτο που προκύπτει από την επικοινωνία του $a_r$ με τους node-neutral agents του G και αντίστοιχα, ο συμβολισμός $\ell_{am,ni}$ (A) στον φόρτο που προκύπτει από την επικοινωνία του $a_m$ με τους node-neutral agents του G.

Ακόμη, ο συμβολισμός $\ell_{am,av}$ αναφέρεται στον φόρτο επικοινωνίας μεταξύ του agent $a_m$ και του πατέρα του $a_v$.

## *4. Επιλογή ομάδας (group selection)*

Αφού υπολογιστούν τα partial benefits για όλους τους agents του G, τότε γίνεται επεξεργασία του υποδένδρου η οποία παράγει τον τελικό προς μετανάστευση συνδυασμό. Η διαδικασία της επεξεργασίας ξεκινάει από τα φύλλα του G. Όποιος agent έχει αρνητικό partial benefit αφαιρείται από το G, ενώ σε αντίθετη περίπτωση συγχωνεύεται με τον πατέρα του δημιουργώντας έτσι ένα καινούριο φύλλο που περιέχει και τους δυο και έχει partial benefit το άθροισμα των partial benefits πατέρα και παιδιού. Η ίδια διαδικασία επαναλαμβάνεται μέχρι την ρίζα του G όπου και σταματάει και επιστρέφεται ο καλύτερος προς μετανάστευση συνδυασμός από agents.

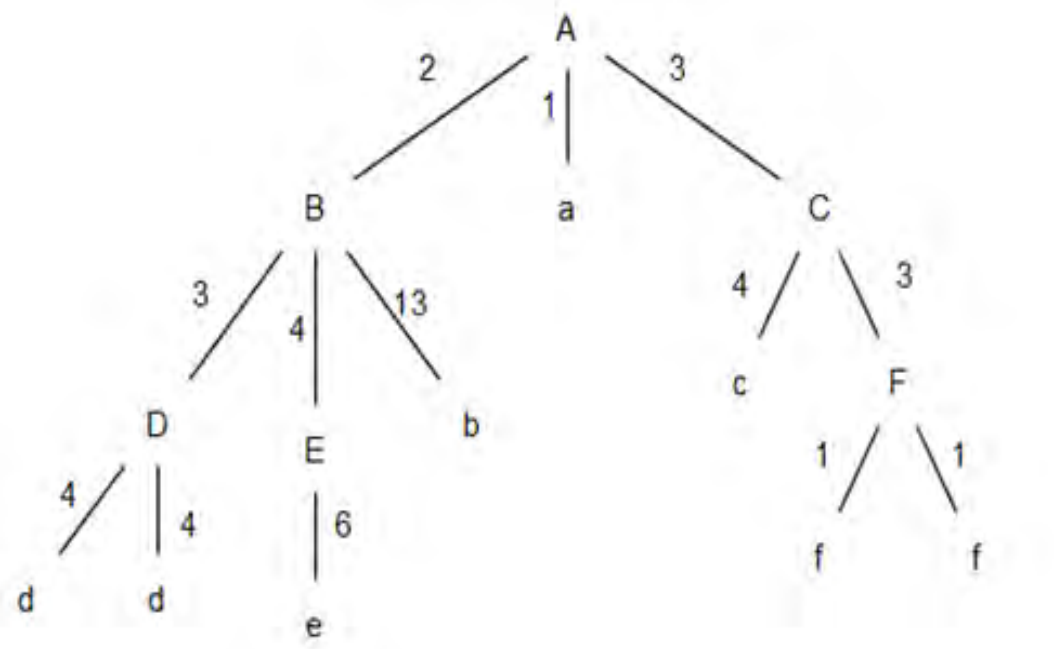
Η παραπάνω διαδικασία εφαρμοζόμενη σε όλους του κόμβους του δικτύου αναμένεται να προκαλέσει μια ελαχιστοποίηση στον συνολικό του φόρτο, ο οποίος μπορεί να περιγραφεί από τον παρακάτω τύπο:

$$Total\ Load = \sum_{i=0}^{A+S} \sum_{j=0}^{A+S} load(ai, aj) * hops(ai, aj) \quad (5)$$

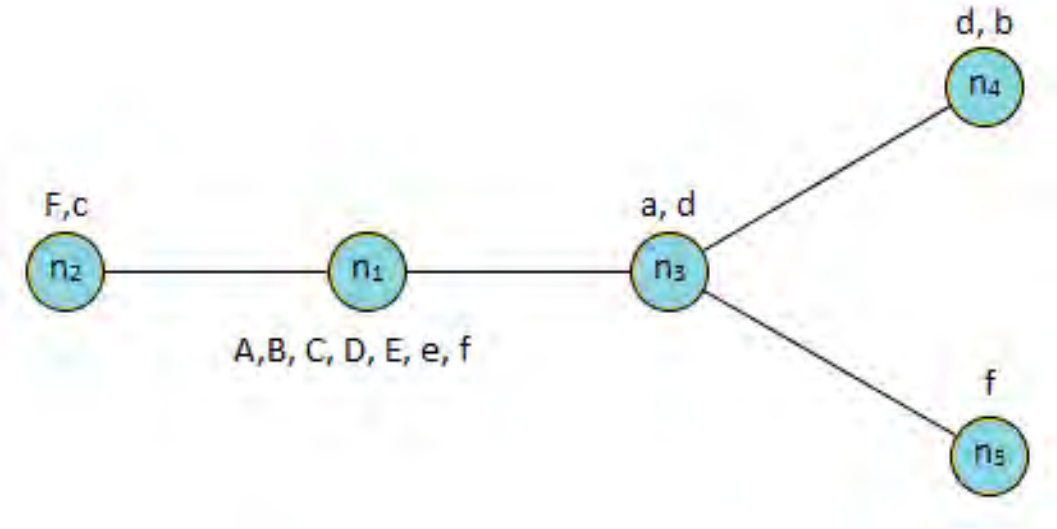
Το load($a_i$,$a_j$) συμβολίζει τον φόρτο επικοινωνίας μεταξύ συνεργαζόμενων agents, ενώ το hops(ai,aj) συμβολίζει την απόσταση μεταξύ τους και στην περίπτωση που οι agents φιλοξενούνται στον ίδιο κόμβο είναι ίσο με μηδέν.

Τα βήματα που ακολουθεί ο αλγόριθμος γίνονται πιο κατανοητά με το παράδειγμα που παρατίθεται στην συνέχεια.

## Παράδειγμα εφαρμογής του GRAL




**Σχήμα 1. Το δένδρο των agents της Εφαρμογής.**




**Σχήμα2. Το δένδρο των κόμβων του Δικτύου.**

Στο Σχήμα 1 απεικονίζεται το δένδρο των agents της εφαρμογής που θα χρησιμοποιήσουμε στο παράδειγμά μας. Τα κεφαλαία γράμματα αναπαριστούν τους node-neutral agents ενώ τα πεζά τους node-specific agents. Τα νούμερα ανάμεσα στους agents αναπαριστούν τον φόρτο επικοινωνίας μεταξύ τους.

Στο Σχήμα 2 απεικονίζεται το δένδρο των κόμβων του δικτύου στο οποίο έχει αναπτυχθεί η εφαρμογή.

Έστω ότι εφαρμόζουμε τον GRAL στον κόμβο $n_1$.

Αρχικά, ο αλγόριθμος ξεκινά ταυτοποιώντας τυχόν υπόδενδρα που υπάρχουν στον κόμβο.

Στον τρέχοντα κόμβο(δηλ. τον $n_1$) ταυτοποιείται μόνο το υπόδενδρο (A,B,C,D,E).

Στη συνέχεια, για κάθε ταυτοποιημένο υπόδενδρο επιλέγεται ο πιο επωφελής, εκ των γειτονικών, προς μετανάστευση κόμβος.

Για κάθε agent του υποδένδρου (A,B,C,D,E) υπολογίζονται τα εξής:

| Agent | $\ell_{ak,n1}$(A) | $\ell_{ak,n1}$(S) | $L_{ak,n2}$ | $L_{ak,n3}$ |
|---|---|---|---|---|
| A | 5 | 0 | 0 | 1 |
| B | 9 | 0 | 0 | 13 |
| C | 3 | 0 | 7 | 0 |
| D | 3 | 0 | 0 | 8 |
| E | 4 | 6 | 0 | 0 |
| Total | | 6 | 7 | 15 |

Ενδεικτικά, για τον agent A το $\ell_{A,n1}$(A), είναι 5 λόγω της επικοινωνίας του με τους B, C. Ενώ, για τον agent E το $\ell_{E,n1}$(S), είναι 6 λόγω της επικοινωνίας του με τον e.

Για τον agent C το $L_{C,n2}$, είναι 7 λόγω της επικοινωνίας του με τους F, c στον $n_2$.

Για τον agent B το $L_{B,n3}$, είναι 13 λόγω της επικοινωνίας του με τον b από τον $n_4$ μέσω του $n_3$.

Πιο επωφελής προορισμός είναι ο κόμβος $n_3$, καθώς ο συνολικός του φόρτος επικοινωνίας (=15) είναι μεγαλύτερος από αυτόν του $n_2$ (=7) και επίσης η συνθήκη (1) ,αφού ο συνολικός του φόρτος είναι μεγαλύτερος και από τον συνολικό φόρτο που προκύπτει τοπικά από την επικοινωνία με τους node specific agents.

Έπειτα, για την επιλογή της καλύτερης προς μετανάστευση ομάδας, θα υπολογίσουμε για κάθε έναν από τους agents την τιμή του affinity, βάσει της σχέσης (2) και του partial benefit, βάσει των σχέσεων (3) και (4).

Ενδεικτικά πάλι, υπολογίζουμε το affinity για τον agent A:

$aff_{A,n3} = \boldsymbol{\ell}_{A,n3} - \boldsymbol{\ell}_{A,n1}(S) - \boldsymbol{\ell}_{A,n2} = 1 - 0 - 0 = 1$

Εύκολα υπολογίζεται με τον ίδιο τρόπο η τιμή του affinity για τους υπόλοιπους agents. Οι τιμές που προκύπτουν είναι:

$aff_{B,n3} = 13$

$aff_{C,n3} = -7$

$aff_{D,n3} = 8$

$aff_{E,n3} = -6$

Έχοντας υπολογίσει τις τιμές του affinity, θα υπολογίσουμε τις τιμές του partial benefit, το οποίο όπως προαναφέρθηκε υπολογίζεται διαφορετικά για την ρίζα του υποδένδρου και διαφορετικά για τους υπόλοιπους agents.

Για την ρίζα:

$pb_{A,n3} = aff_{A,n3} - \boldsymbol{\ell}_{A,n1}(A) = 1 - 5 = -4$

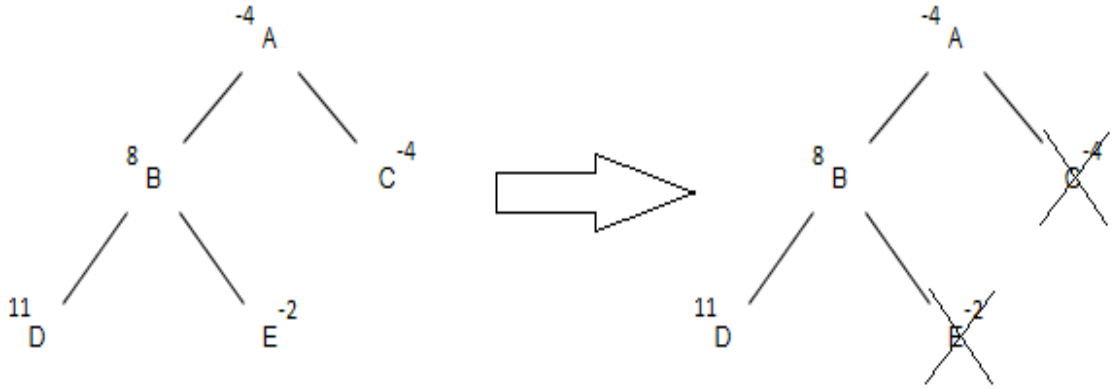
Για τους υπόλοιπους agents του υποδένδρου:

$pb_{B,n3} = aff_{B,n3} - \boldsymbol{\ell}_{B,n1}(A) + 2 * \boldsymbol{\ell}_{B,A} = 13 - 9 + 2 * 2 = 8$

$pb_{C,n3} = -4$

$pb_{D,n3} = 11$

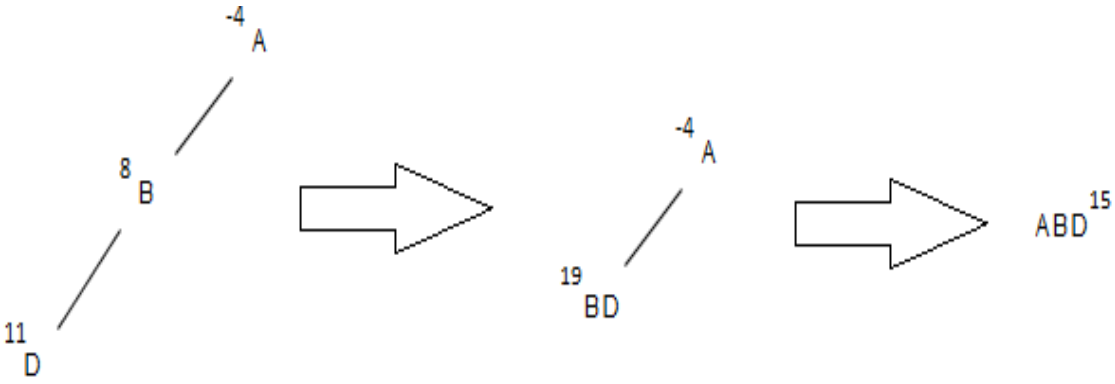
$pb_{E,n3} = -2$




**Σχήμα 3. Τα φύλλα με αρνητικό partial benefit διαγράφονται.**

Όπως φαίνεται στο παραπάνω σχήμα, τα φύλλα με αρνητικό partial benefit διαγράφονται.

Στη συνέχεια, γίνεται από κάτω προς τα πάνω συγχώνευση των agents. Δηλαδή, ο agent D συγχωνεύεται με τον B, δημιουργώντας το φύλλο-ομάδα BD με partial benefit 19. Τα φύλλα-ομάδες υπακούν επίσης στον κανόνα που επιβάλλει τα φύλλα με αρνητικό partial benefit να διαγράφονται. Αφού 19>0 η προς τα πάνω συγχώνευση συνεχίζεται και η τελική ομάδα που προκύπτει είναι η ABD με συνολικό partial benefit 15.

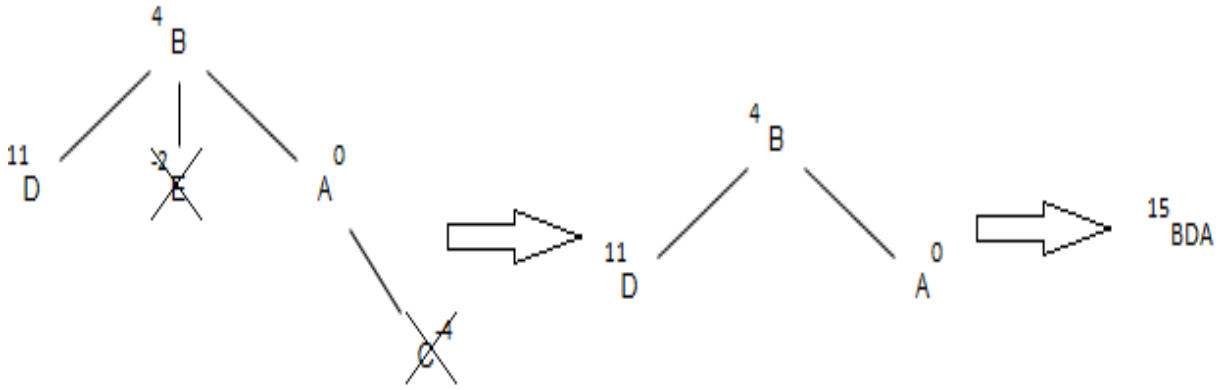



**Σχήμα 4. Η διαδικασία μέχρι τη τελική ομαδοποίηση.**

Για να παραχθεί η καλύτερη δυνατή ομαδοποίηση, το υπόδενδρο (A, B, C, D, E) θα πρέπει να ελεγχθεί για την περίπτωση όπου κάθε ένας από τους agents που το απαρτίζουν αποτελεί την ρίζα του.

Στην περίπτωση όπου η ρίζα του υποδένδρου είναι ο B προκύπτει η παρακάτω ομαδοποίηση.

Να τονισθεί ότι οι τιμές των partial benefits ανανεώνονται εφόσον αλλάζουν και οι σχέσεις πατέρα-παιδιού μεταξύ των agents.

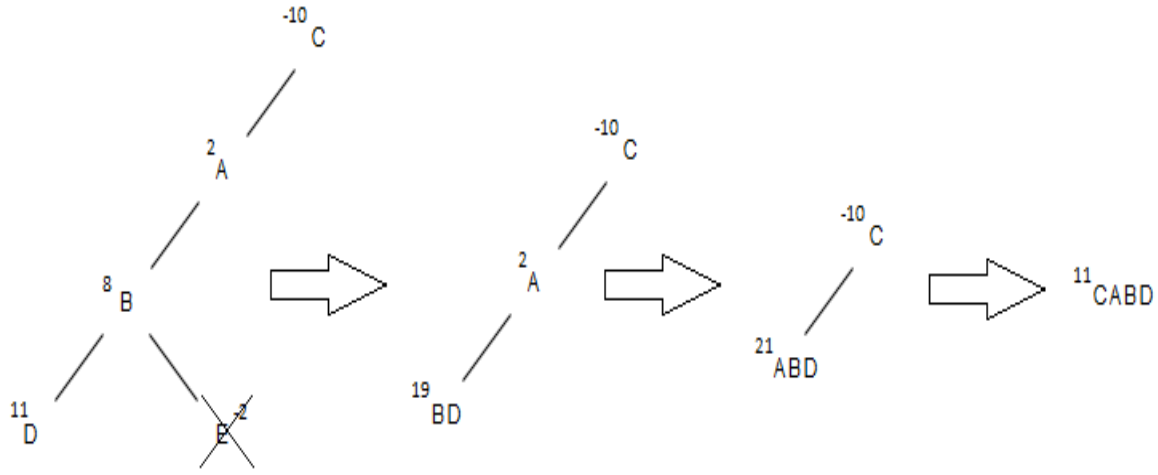



**Σχήμα 5. Η ομαδοποίηση που προκύπτει στην περίπτωση που ρίζα του υποδένδρου είναι ο B.**

Όπως φαίνεται στο παραπάνω σχήμα, και πάλι τα φύλλα με αρνητικό partial benefit διαγράφονται.

Στη συνέχεια, γίνεται από κάτω προς τα πάνω συγχώνευση των agents. Ο agent D και ο agent A συγχωνεύονται με τον B, δημιουργώντας την τελική ομάδα BDA με συνολικό partial benefit 15.

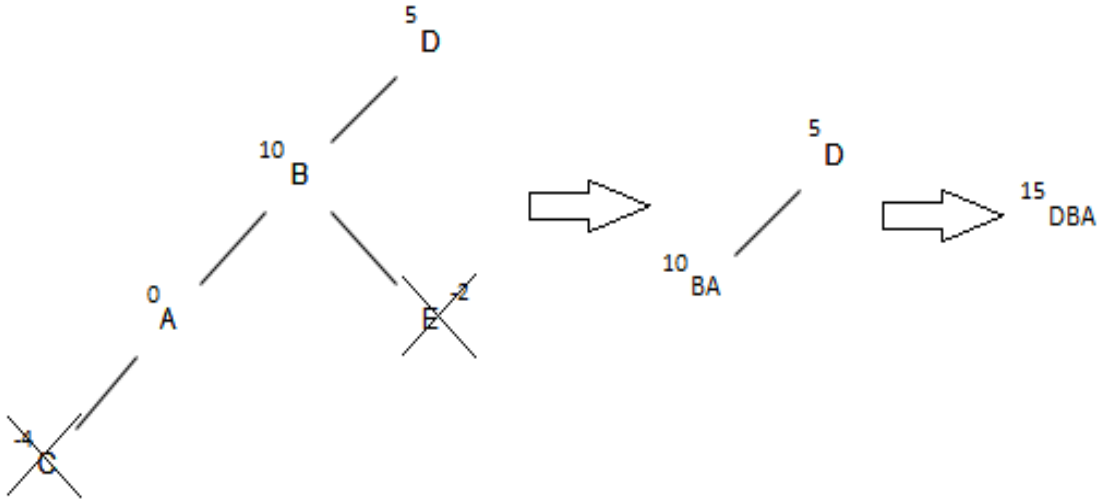
Παρακάτω, παρατίθενται οι αντίστοιχες ομαδοποιήσεις για τους C, D και E.




**Σχήμα 6. Η ομαδοποίηση που προκύπτει στην περίπτωση που ρίζα του υποδένδρου είναι ο C.**

Όπως φαίνεται στο παραπάνω σχήμα, τα φύλλο E με αρνητικό partial benefit διαγράφεται και ξεκινά η διαδικασία της από κάτω προς τα πάνω συγχώνευσης των agents. Δηλαδή, ο agent D συγχωνεύεται με τον B, δημιουργώντας το φύλλο-ομάδα BD με partial benefit 19. Στη συνέχεια, το φύλλο-ομάδα BD συγχωνεύεται με τον agent A και σαν ομάδα αποκτούν partial benefit ίσο με 21 το οποίο είναι μεγαλύτερο του μηδενός και οδηγεί έτσι στη συγχώνευση του ABD με τον agent C, δημιουργώντας την τελική ομάδα CABD με συνολικό partial benefit ίσο με 11.

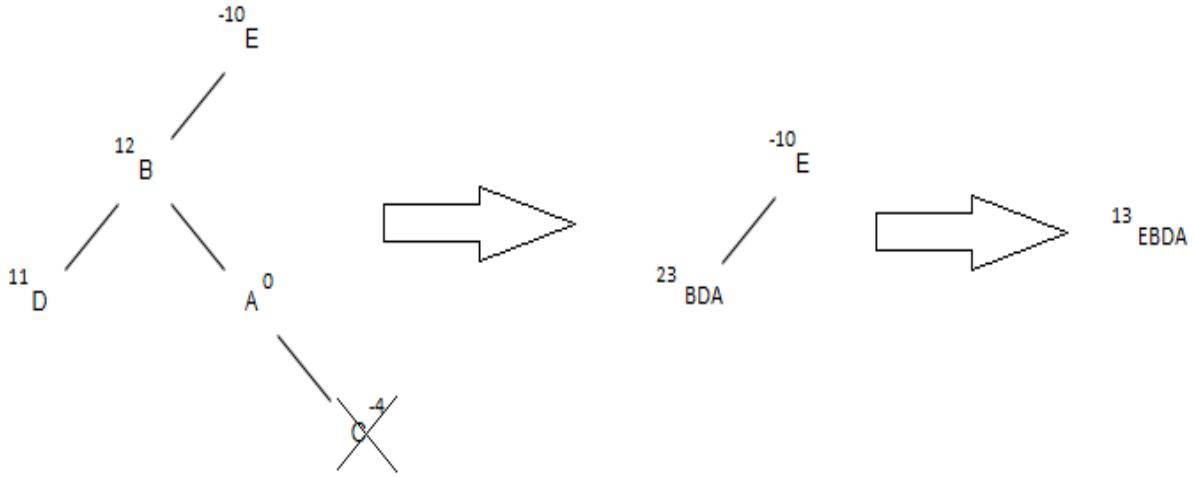
Παρακάτω απεικονίζεται η περίπτωση όπου ρίζα του υποδένδρου είναι ο D. Ακολουθώντας την ίδια διαδικασία, οι C και E με αρνητικό partial benefit διαγράφονται και οι A, B και D συγχωνεύονται και αποκτούν θετικό συνολικό partial benefit ίσο με 15.




**Σχήμα 7. Η ομαδοποίηση που προκύπτει στην περίπτωση που ρίζα του υποδένδρου είναι ο D.**

Τέλος, την θέση της ρίζας του υποδένδρου παίρνει και ο Ε. Στην περίπτωση αυτή, αρνητικό partial benefit έχει ο agent C ο οποίος και διαγράφεται. Στην συνέχεια οι A, B και D συγχωνεύονται και αποκτούν θετικό partial benefit ίσο με 23 και έπειτα συγχωνεύονται με τον Ε, δημιουργώντας την τελική ομάδα EBDA με partial benefit ίσο με 13.

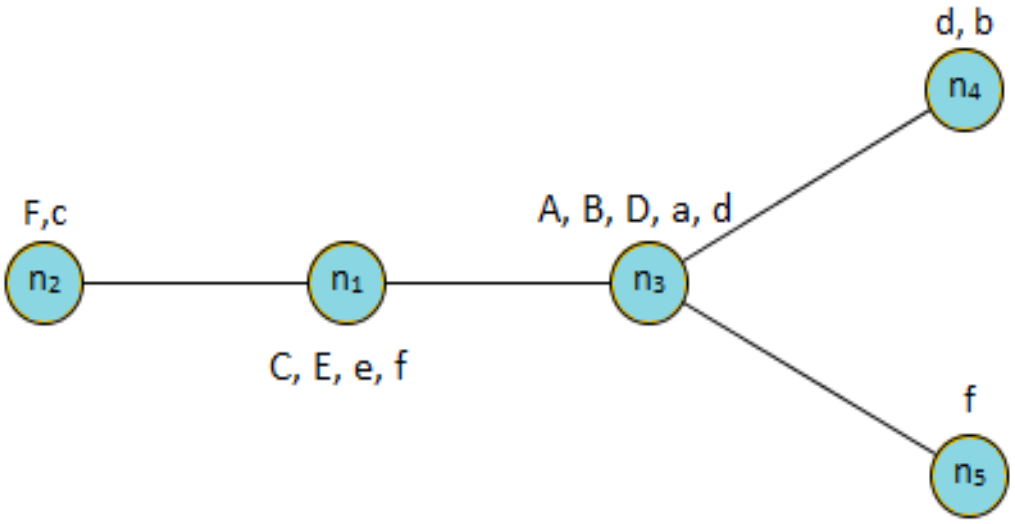
Η διαδικασία που ακολουθεί η ομαδοποίηση στην περίπτωση όπου ρίζα του υποδένδρου είναι ο Ε φαίνεται στο παρακάτω σχήμα.




**Σχήμα 8. Η ομαδοποίηση που προκύπτει στην περίπτωση που ρίζα του υποδένδρου είναι ο Ε.**

Όπως φαίνεται από τις παραπάνω ομαδοποιήσεις, η καλύτερη που προέκυψε είναι η ABD με migration benefit ίσο με 15. Επομένως, η ομάδα ABD μεταναστεύει στον κόμβο $n_3$.

Το δίκτυο, μετά την εφαρμογή του GRAL στον κόμβο $n_1$, αποκτά την εξής μορφή:




**Σχήμα 9. Το δένδρο των κόμβων του Δικτύου, μετά την εφαρμογή του GRAL στον κόμβο $n_1$.**

Η διαδικασία που περιγράφηκε για τον $n_1$, εφαρμόζεται για όλους τους κόμβους του δικτύου και επαναλαμβάνεται έως ότου δεν εντοπίζεται κάποια ωφέλιμη μετανάστευση.

# Κεφάλαιο 3

## 3.1 Υλοποίηση προσομοίωσης

Στα πλαίσια αυτής της πτυχιακής εργασίας δεν κατεστάθη δυνατή η εφαρμογή του GRAL σε πραγματικές συνθήκες. Για τον λόγο αυτό, το δίκτυο των αισθητήρων, οι εφαρμογές οι οποίες αναπτύσσονται πάνω σε αυτό, καθώς και η λειτουργία του GRAL προσομοιώθηκαν σε γλώσσα C.

Συγκεκριμένα, δημιουργήθηκαν δίκτυα μεγέθους 20 και 50 κόμβων καθώς και εφαρμογές με διαφορετικούς αριθμούς από agents, (10, 5) , (25, 12) και (50, 22), node specific και node neutral αντίστοιχα.

Για την δημιουργία του δικτύου των κόμβων ακολουθήθηκε η παρακάτω διαδικασία:

Αρχικά, οι κόμβοι αναπαραστάθηκαν σαν ψευδοτυχαία σημεία στο επίπεδο, με εύρος τιμών [0, 80] για την περίπτωση του δικτύου 20 κόμβων και [0, 120] για την περίπτωση του δικτύου 50 κόμβων και θεωρήθηκε πως συνδέονται κόμβοι οι οποίοι απείχαν κατά ευκλείδεια απόσταση μικρότερη του 30. Αυτή η διαδικασία δημιούργησε έναν γράφο πάνω στον οποίο εφαρμόστηκε ο αλγόριθμος του Prim, για την εύρεση του ελάχιστου γεννητικού δένδρου( minimum spanning tree ), ώστε οι κόμβοι να συνδέονται μεταξύ τους με μοναδικά μονοπάτια.

Αντίστοιχα, η δημιουργία του δένδρου των εφαρμογών υλοποιήθηκε ως εξής:

Σε πρώτο στάδιο, οι node specific agents χωρίστηκαν σε ομάδες των 5 και για κάθε τέτοια ομάδα, 2-5 agents επιλέγονταν τυχαία ως παιδιά ενός node neutral agent. Στη συνέχεια, οι node specific agents που δεν ανατέθηκαν σε κάποιον πατέρα, μαζί με τους node neutral agents που απέκτησαν παιδία χωριζόντουσαν πάλι σε ομάδες των 5 επαναλαμβάνοντας την διαδικασία ανάθεσής τους σε κάποιον καινούριο node neutral agent μέχρι και την ρίζα του δένδρου. Όσον αφορά το μέγεθος της πληροφορίας που οι agents ανταλλάσουν μεταξύ τους, θεωρήθηκε πως κάθε node specific agent στέλνει στον πατέρα του τυχαία από 1 έως 5 μηνύματα ανά μονάδα χρόνου, ενώ για τους node neutral agents, θεωρήθηκαν τρεις περιπτώσεις. Στην πρώτη, όλοι οι node neutral agents στέλνουν στον πατέρα του τον μέσο όρο του όγκου της πληροφορίας που λαμβάνουν από τα παιδιά τους, στην δεύτερη στέλνουν το άθροισμα της πληροφορίας που λαμβάνουν από τα παιδιά τους, ενώ στην τρίτη οι μισοί node neutral agents στέλνουν το άθροισμα του φόρτου που

λαμβάνουν από τα παιδιά τους και οι άλλοι μισοί τον μέσο όρο του φόρτου που λαμβάνουν. Στη συνέχεια, η πρώτη περίπτωση θα αναφέρεται ως load(avg), η δεύτερη ως load(sum) και η τρίτη ως load(mix).

Ακολούθως, οι agents της εφαρμογής σκορπίστηκαν τυχαία μέσα στο δίκτυο και σε κάθε κόμβο εφαρμόστηκε ο αλγόριθμος.

Τα βασικά structs για την υλοποίηση του δένδρου των agents της εφαρμογής, του δικτύου των κόμβων καθώς και της ανάθεσης των πρώτων στους δεύτερους είναι τα εξής:

```
struct Agent
{
   int agent;
   int load;
   struct Agent* next;
};
```

```
struct netNode
{
   int id;
   …
   struct netNode*
next;
};
```

```
//agents' assignments on
nodes
struct nodeA
{
        int id;
        …
        struct nodeA* next;
}
```

Το **struct nodeA** χρησιμοποιείται για την δημιουργία μιας λίστα γειτνίασης η οποία δείχνει τις αναθέσεις των agents πάνω στους κόμβους. Η κεφαλή της λίστας αναπαριστά κάθε φορά το id του κόμβου, ενώ τα υπόλοιπα δεδομένα της λίστας τους agents που έχουν ανατεθεί στους κόμβους.

Για την δημιουργία του δένδρου της εφαρμογής, καθώς και του δένδρου του δικτύου έχουν χρησιμοποιηθεί επίσης λίστες γειτνίασης.

Ο αλγόριθμος αξιοποιεί τις παραπάνω λίστες γειτνίασης για τον εντοπισμό υποδένδρων, την επεξεργασία τους και την μεταφορά τους στον πιο επωφελή προορισμό. Εφαρμόζεται σε κάθε κόμβο του δικτύου και τερματίζει αφού έχουν εξεταστεί όλοι οι κόμβοι και δεν έχει εντοπιστεί καμία δυνατή μετανάστευση.

Κάθε φορά που τρέχει το πρόγραμμα που προσομοιώνει τα παραπάνω, παράγεται ένα διαφορετικό δίκτυο, μια διαφορετική εφαρμογή, καθώς και μια διαφορετική ανάθεση των agents της εφαρμογής πάνω στους κόμβους του εκάστοτε δικτύου.

## 3.2 Αποτελέσματα Προσομοίωσης

Για την δοκιμή της απόδοσης του αλγορίθμου σε σχέση με την μείωση του συνολικού φόρτου του δικτύου, ο οποίος υπολογίζεται σύμφωνα με την σχέση (5), εκτελέστηκαν ενδεικτικά 100 πειράματα για όλους τους πιθανούς συνδυασμούς δικτύου (20 και 50 κόμβων) και εφαρμογής ( (10, 5) , (25, 12) και (50, 22), node specific και node neutral αντίστοιχα), καθώς και για τις αντίστοιχες περιπτώσεις ανταλλαγής δεδομένων μεταξύ των agents (load(avg), load(sum) και load(mix)).

Τα αποτελέσματα των πειραμάτων παρουσιάζονται αναλυτικά παρακάτω. Ως app-10 συμβολίζονται οι εφαρμογές με (10,5) agents, node neutral και node specific αντίστοιχα, ως app-25 οι εφαρμογές με (25, 12) agents και ως app-50 συμβολίζονται οι εφαρμογές με (50, 22) agents.

### *Πειράματα σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(avg):*

Οι παρακάτω πίνακες αναφέρονται στην περίπτωση όπου οι node neutral agents στέλνουν στον πατέρα τους τον μέσο όρο του φόρτου που λαμβάνουν από τα παιδιά τους (load(avg)). Τα αποτελέσματα δείχνουν ότι ανεξαρτήτως του αριθμού των κόμβων του δικτύου και της εφαρμογής που αναπτύσσεται πάνω σε αυτό η ποσοστιαία μείωση είναι σχεδόν σταθερή και κυμαίνεται μεταξύ 55% έως και 59%.

Ακόμη, όσον αφορά τον μέσο αριθμό των μεταναστεύσεων και στις δύο περιπτώσεις δικτύου παρατηρείται μια σταθερή αύξηση, ανάλογη του μεγέθους της εφαρμογής.

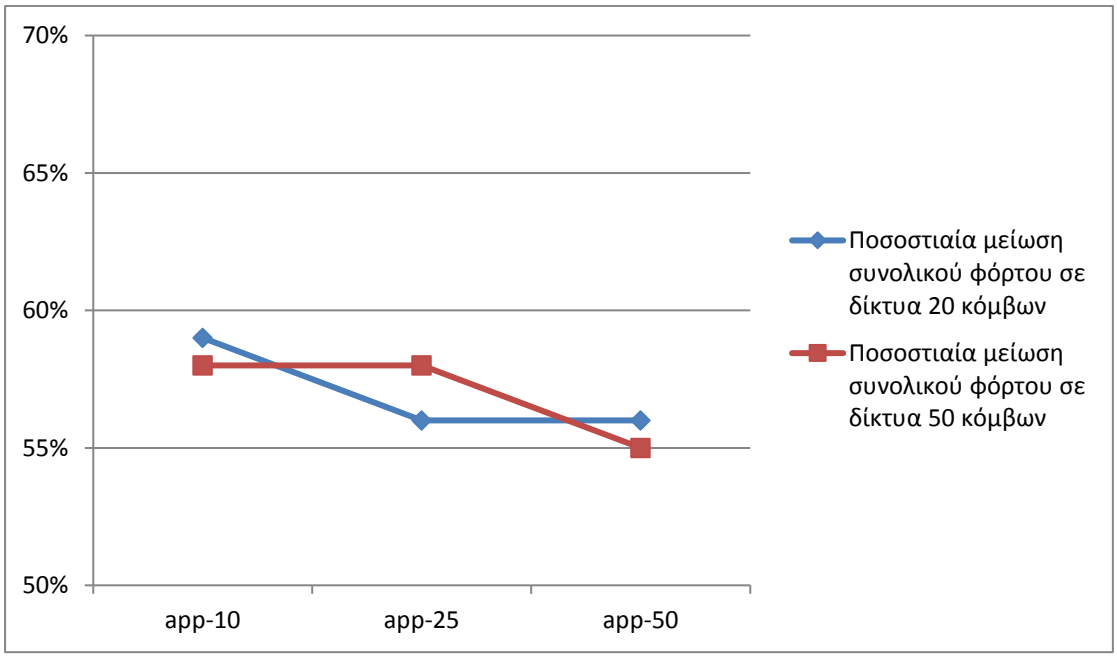
| | Δίκτυο 20 κόμβων (load(avg)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 235,8 | 552,38 | 1131 |
| Μέσος τελικός συνολικός φόρτος | 96,54 | 245,22 | 503,21 |
| Ποσοστιαία μείωση συνολικού φόρτου | 59% | 56% | 56% |
| Μέσος αριθμός μεταναστεύσεων | 17,48 | 33,4 | 63,12 |

**Πίνακας 1. Αποτελέσματα πειράματος σε δίκτυο 20 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(avg).**

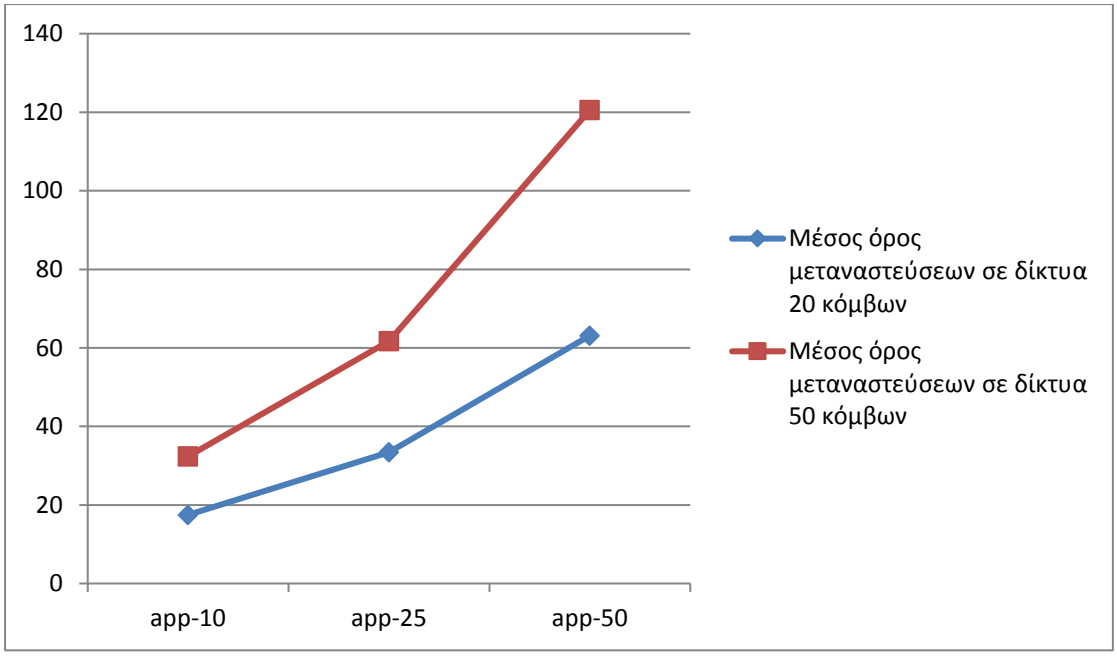
| | Δίκτυο 50 κόμβων (load(avg)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 429,74 | 1031,44 | 2214,38 |
| Μέσος τελικός συνολικός φόρτος | 181,64 | 434 | 1001,9 |
| Ποσοστιαία μείωση συνολικού φόρτου | 58% | 58% | 55% |
| Μέσος αριθμός μεταναστεύσεων | 32,38 | 61,68 | 120,55 |

**Πίνακας 2. Αποτελέσματα πειράματος σε δίκτυο 50 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(avg).**

Στα παρακάτω διαγράμματα, φαίνεται παραστατικά η ποσοστιαία μείωση του συνολικού φόρτου καθώς και ο μέσος όρος των μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(avg).




**Διάγραμμα 1. Ποσοστιαία μείωση συνολικού φόρτου σε δίκτυο 20 και 50 κόμβων για την περίπτωση load(avg).**




**Διάγραμμα 2. Μέσος όρος μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(avg).**

### *Πειράματα σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(sum):*

Οι παρακάτω πίνακες αναφέρονται στην περίπτωση όπου οι node neutral agents στέλνουν στον πατέρα τους το άθροισμα του φόρτου που λαμβάνουν από τα παιδιά τους (load(sum)). Τα αποτελέσματα δείχνουν και σε αυτή την περίπτωση ότι ανεξαρτήτως του αριθμού των κόμβων του δικτύου και της εφαρμογής που αναπτύσσεται πάνω σε αυτό η ποσοστιαία μείωση είναι επίσης σχεδόν σταθερή και κυμαίνεται μεταξύ 81% έως και 88%.

Το ποσοστό μείωσης εδώ φαίνεται να είναι κατά πολύ μεγαλύτερο από την προηγούμενη περίπτωση, όμως αυτό οφείλεται στο γεγονός ότι οι node neutral agents ανταλλάσουν μεταξύ τους μεγαλύτερο όγκο πληροφορίας (στέλνουν το άθροισμα του φόρτου που λαμβάνουν από τα παιδιά τους και όχι τον μέσο όρο) δίνοντας έτσι έναν αρκετά μεγαλύτερο αρχικό συνολικό φόρτο, όντας τοποθετημένοι αρχικά μακριά ο ένας από τον άλλον στο δίκτυο των κόμβων. Η εφαρμογή του αλγορίθμου φέρνει πολύ κοντά τους συνεργαζόμενους agents, αφού δεν υπάρχει κάποιος περιορισμός στον αριθμό των agents που μπορεί κάποιος κόμβος να φιλοξενήσει, επιφέροντας έτσι μια σημαντική μείωση τον συνολικό φόρτο του δικτύου.

Ακόμη, όσον αφορά τον μέσο αριθμό των μεταναστεύσεων και στις δύο περιπτώσεις δικτύου παρατηρείται και εδώ μια σχεδόν σταθερή αύξηση, ανάλογη του μεγέθους της εφαρμογής.

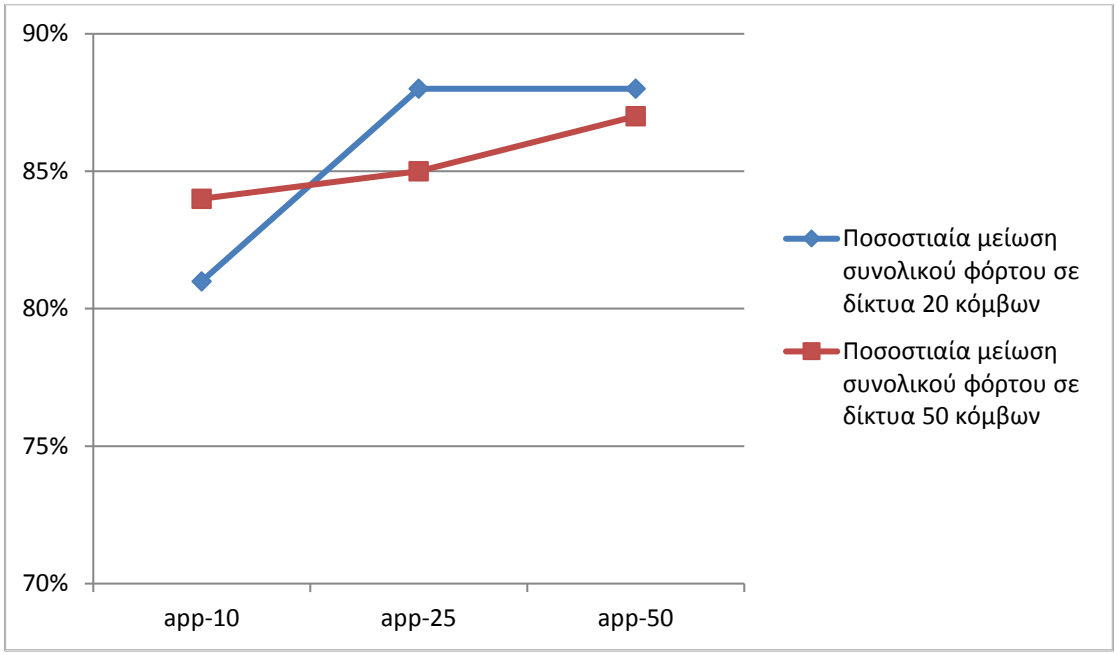
| | Δίκτυο 20 κόμβων (load(sum)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 522,72 | 2208,22 | 4612,19 |
| Μέσος τελικός συνολικός φόρτος | 101,42 | 266,46 | 548,14 |
| Ποσοστιαία μείωση συνολικού φόρτου | 81% | 88% | 88% |
| Μέσος αριθμός μεταναστεύσεων | 20,18 | 51,51 | 91,06 |

**Πίνακας 3. Αποτελέσματα πειράματος σε δίκτυο 20 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(sum).**

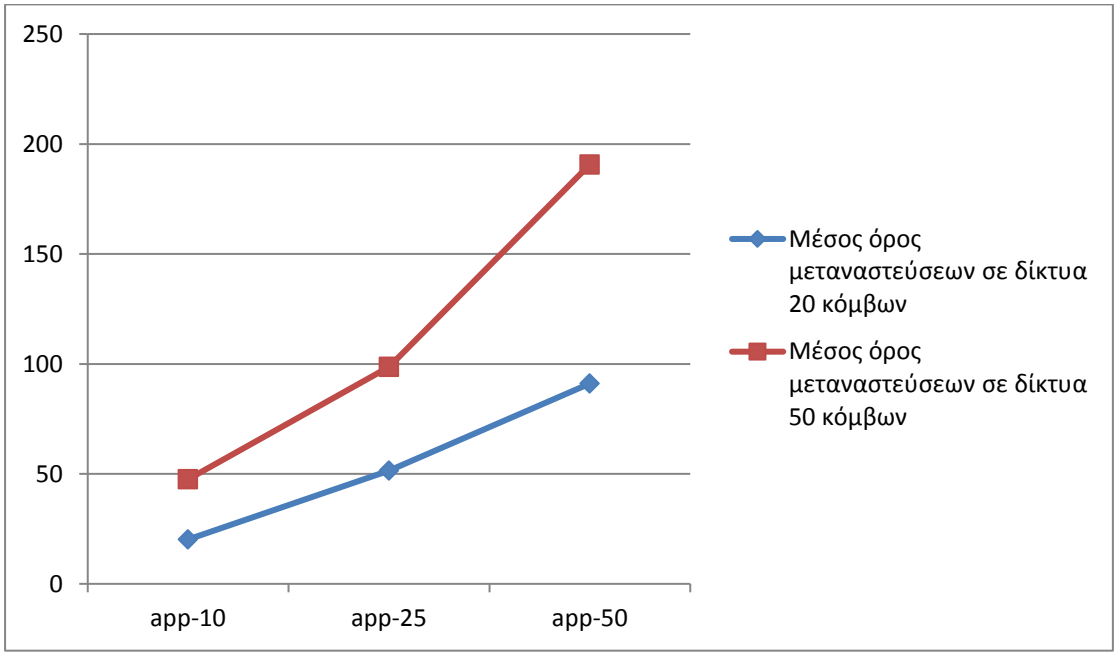
| | Δίκτυο 50 κόμβων (load(sum)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 1318 | 3429,72 | 8611,88 |
| Μέσος τελικός συνολικός φόρτος | 204,76 | 498,02 | 1155,22 |
| Ποσοστιαία μείωση συνολικού φόρτου | 84% | 85% | 87% |
| Μέσος αριθμός μεταναστεύσεων | 47,55 | 98,7 | 190,68 |

**Πίνακας 4. Αποτελέσματα πειράματος σε δίκτυο 50 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(sum).**

Στα παρακάτω διαγράμματα, φαίνεται παραστατικά η ποσοστιαία μείωση του συνολικού φόρτου καθώς και ο μέσος όρος των μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(sum).




**Διάγραμμα 3. Ποσοστιαία μείωση συνολικού φόρτου σε δίκτυο 20 και 50 κόμβων για την περίπτωση load(sum).**




**Διάγραμμα 4. Μέσος όρος μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(sum).**

### *Πειράματα σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(mix):*

Οι παρακάτω πίνακες αναφέρονται στην περίπτωση όπου μισοί οι node neutral agents στέλνουν στον πατέρα τους το άθροισμα του φόρτου που λαμβάνουν από τα παιδιά τους, ενώ οι άλλοι μισοί τον μέσο όρο του φόρτου που λαμβάνουν (load(mix)). Τα αποτελέσματα δείχνουν και εδώ ότι ανεξαρτήτως του αριθμού των κόμβων του δικτύου και της εφαρμογής που αναπτύσσεται πάνω σε αυτό η ποσοστιαία μείωση είναι σχεδόν σταθερή και κυμαίνεται μεταξύ 74% έως και 75%.

Επίσης, το ποσοστό μείωσης εδώ φαίνεται να είναι ο μέσος όρος των ποσοστών μείωσης που απέδωσαν οι δυο προηγούμενες περιπτώσεις και είναι λογικό, αφού οι μισοί node neutral agents της εφαρμογής στέλνουν στον πατέρα τους το άθροισμα του φόρτου που λαμβάνουν από τα παιδιά τους, ενώ οι άλλοι μισοί τον μέσο όρο του φόρτου που λαμβάνουν.

Ακόμη, όσον αφορά τον μέσο αριθμό των μεταναστεύσεων και στις δύο περιπτώσεις δικτύου παρατηρείται παρόμοια με τις άλλες δυο περιπτώσεις μια σχεδόν σταθερή αύξηση, ανάλογη του μεγέθους της εφαρμογής που εφαρμόζεται στο εκάστοτε δίκτυο.

| | Δίκτυο 20 κόμβων (load(mix)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 375,8 | 1036,95 | 1980,43 |
| Μέσος τελικός συνολικός φόρτος | 94,5 | 274,2 | 524,05 |
| Ποσοστιαία μείωση συνολικού φόρτου | 75% | 74% | 74% |
| Μέσος αριθμός μεταναστεύσεων | 19,07 | 43,95 | 72 |

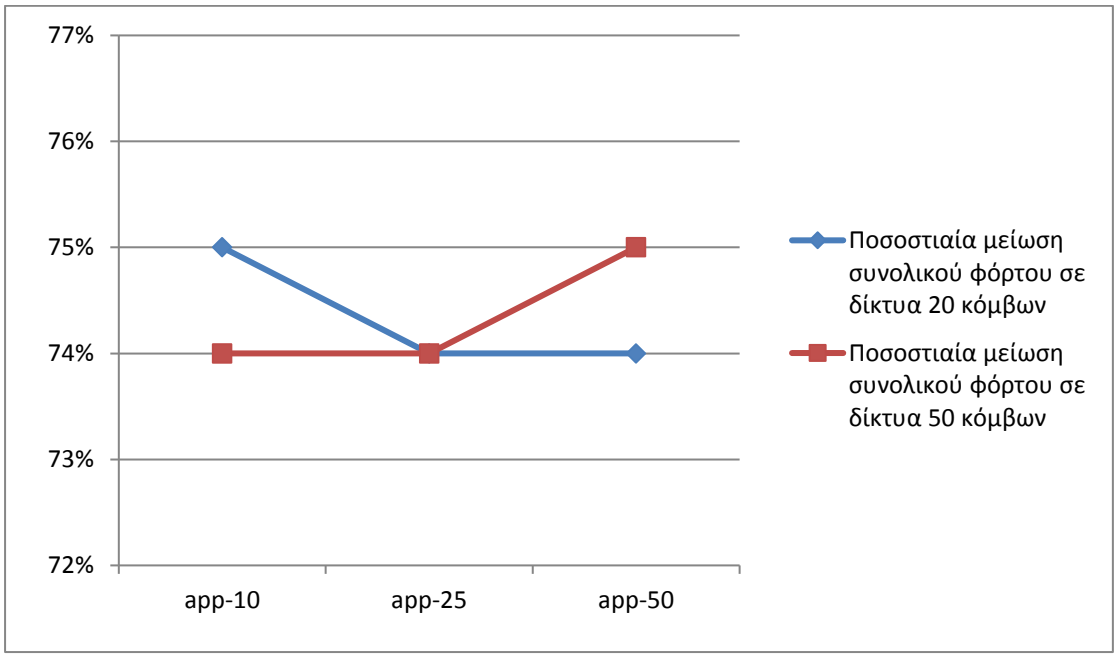
**Πίνακας 5. Αποτελέσματα πειράματος σε δίκτυο 20 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(mix).**

| | Δίκτυο 50 κόμβων (load(mix)) | | |
|---|---|---|---|
| | app-10 | app-25 | app-50 |
| Μέσος αρχικός συνολικός φόρτος | 793 | 2041,68 | 3984 |
| Μέσος τελικός συνολικός φόρτος | 202,7 | 525,42 | 995,96 |
| Ποσοστιαία μείωση συνολικού φόρτου | 74% | 74% | 75% |
| Μέσος αριθμός μεταναστεύσεων | 40,2 | 83,56 | 149,63 |

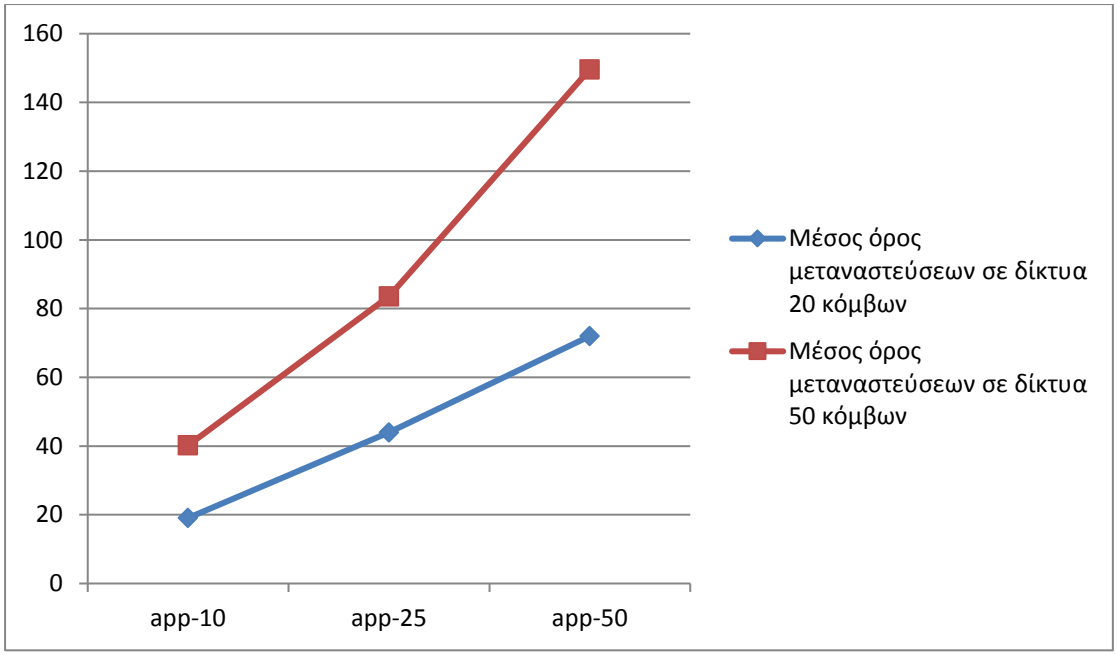
**Πίνακας 6. Αποτελέσματα πειράματος σε δίκτυο 50 κόμβων με εφαρμογές των (10,5), (25,12) και (50,22) node specific και node neutral agents αντίστοιχα για την περίπτωση load(mix).**

Στα παρακάτω διαγράμματα, φαίνεται παραστατικά η ποσοστιαία μείωση του συνολικού φόρτου καθώς και ο μέσος όρος των μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(mix).

.




**Διάγραμμα 5. Ποσοστιαία μείωση συνολικού φόρτου σε δίκτυο 20 και 50 κόμβων για την περίπτωση load(mix).**




**Διάγραμμα 6. Μέσος όρος μεταναστεύσεων σε δίκτυα 20 και 50 κόμβων για την περίπτωση load(mix).**

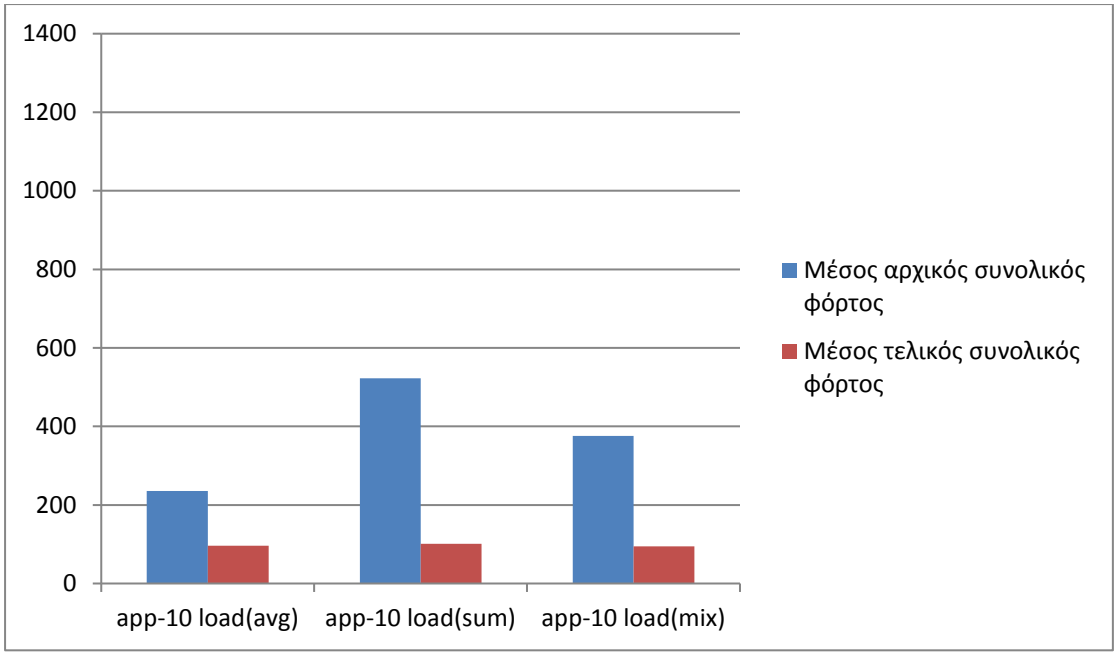
# Κεφάλαιο 4

## 4.1 Συμπεράσματα

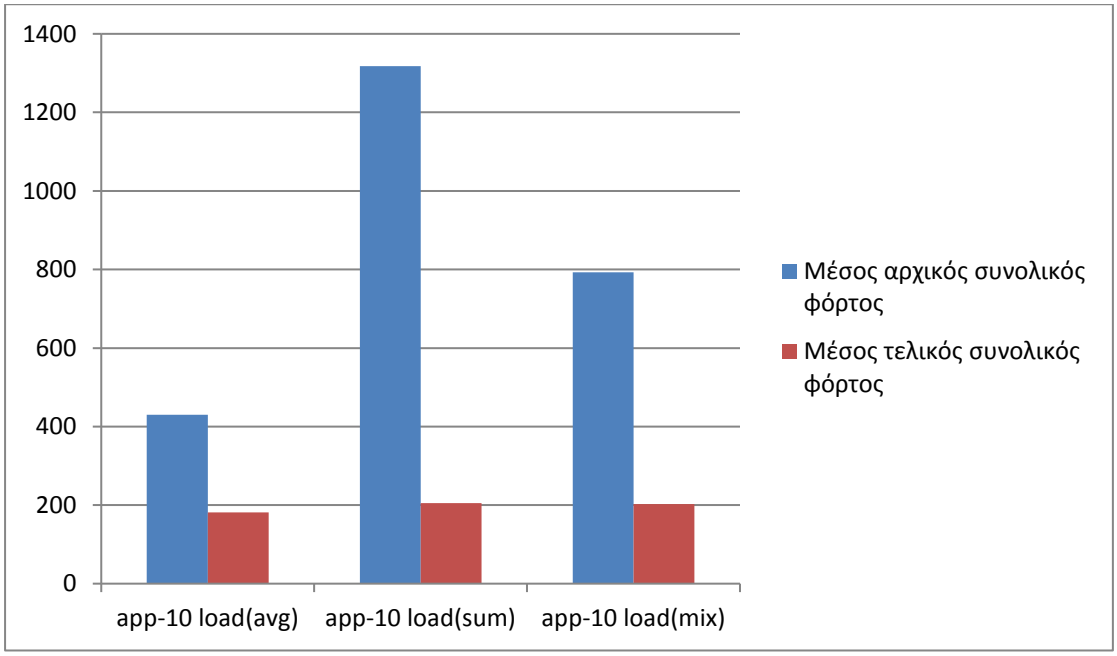
Τα πειράματα που διενεργήθηκαν έδειξαν ότι ο GRAL όντως επιφέρει μια σημαντική μείωση στον συνολικό φόρτο του δικτύου, στην περίπτωση που δεν υπάρχουν περιορισμοί χωρητικότητας και οι agents μπορούν να μετακινούνται ελεύθερα σε οποιονδήποτε κόμβο του δικτύου. Επίσης, λαμβάνοντας υπόψη διαφορετικές περιπτώσεις ανταλλαγής δεδομένων μεταξύ των node neutral agents (load(avg), load(sum) και load(mix)), το ποσοστό μείωσης του συνολικού φόρτου διαφοροποιήθηκε ανάλογα με τον όγκο της πληροφορίας που ανταλλάσσονταν κάθε φορά. Στην περίπτωση του load(sum) αυξήθηκε αρκετά, φτάνοντας μέχρι και το 88% σε δίκτυα 20 κόμβων και μέχρι και το 87% σε δίκτυα 50 κόμβων, ενώ στην περίπτωση load(avg) που έφτασε μέχρι και το 59% σε δίκτυα 20 κόμβων και μέχρι και το 58% σε δίκτυα 50 κόμβων. Επίσης, στην περίπτωση του load(mix), το ποσοστό μείωσης του τελικού συνολικού φόρτου έφτασε μέχρι και το 75% και για τα δύο μεγέθη δικτύων. Στις περιπτώσεις όπου παρατηρείται πιο αυξημένο ποσοστό μείωσης, ο φόρτος που οι συγγενείς node neutral ανταλλάσουν μεταξύ τους είναι μεγαλύτερος (load(sum) και load(mix)) οδηγώντας έτσι σε μεγαλύτερο αρχικό συνολικό φόρτο, ενώ επειδή οι εξαρτήσεις μεταξύ των συγγενών node neutral agents είναι ισχυρότερες, η εφαρμογή του GRAL έχει ως αποτέλεσμα οι node neutral agents να έρχονται πιο κοντά, επιφέροντας έτσι μια σημαντική μείωση στον συνολικό φόρτο του δικτύου.

Στα παρακάτω διαγράμματα απεικονίζονται ενδεικτικά οι μέσοι αρχικοί και τελικοί συνολικοί φόρτοι σε δίκτυα 20 και 50 κόμβων για εφαρμογές app-10.

Αυτό που παρατηρείται είναι ότι ενώ οι μέσοι αρχικοί συνολικοί φόρτοι διαφοροποιούνται, καθώς αλλάζει το μοντέλο ανταλλαγής δεδομένων μεταξύ των node neutral agents, η εφαρμογή του GRAL, στην περίπτωση που δεν υπάρχουν περιορισμοί χωρητικότητας, έχει ως αποτέλεσμα οι μέσοι τελικοί συνολικοί φόρτοι να κυμαίνονται περίπου στα ίδια επίπεδα και στις δύο περιπτώσεις δικτύου.




**Διάγραμμα 7. Δένδρο εφαρμογής με (10,5) node specific και node neutral αντίστοιχα σε δίκτυα 20 κόμβων.**




**Διάγραμμα 8. Δένδρο εφαρμογής με (10,5) node specific και node neutral αντίστοιχα σε δίκτυα 50 κόμβων.**

## 4.2 Πρόταση για μελλοντική έρευνα

Στην παρούσα πτυχιακή εργασία, έγινε μελέτη του αλγορίθμου GRAL για την περίπτωση που οι κόμβοι του δικτύου στο οποίο αναπτύσσεται μια εφαρμογή δεν έχουν περιορισμούς χωρητικότητας και οι agents μπορούν να μετακινούνται ελεύθερα σε οποιονδήποτε κόμβο. Η εφαρμογή και η δοκιμή του GRAL, στην περίπτωση που οι κόμβοι του δικτύου διαθέτουν έναν περιορισμό σχετικά με τον αριθμό των agents που μπορούν να φιλοξενήσουν, μπορεί να αποτελέσει αντικείμενο για μελλοντική μελέτη.

# Βιβλιογραφία


**[1] Wireless Sensor Networks, Ian F. Akyildiz, Mehmet Can Vuran**

**[2] Wireless Sensor Networks, last edited on 22 May 2017, https://en.wikipedia.org/wiki/Wireless_sensor_network**

**[3] Prim's MST for Adjacency List Representation, http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-mst-for-adjacency-list-representation/**

**[4] F. L. Lewis, Wireless Sensor Networks, 2004**

**[5] Tziritas N. et al. (2012) Middleware Mechanisms for Agent Mobility in Wireless Sensor and Actuator Networks. In: Martins F., Lopes L., Paulino H. (eds) Sensor Systems and Software. S-CUBE 2012. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 102. Springer, Berlin, Heidelberg**

**[6] Mobile agent, last edited on 6 February 2017, https://en.wikipedia.org/wiki/Wireless_sensor_network**

**[7] A Survey on Middleware for Wireless Sensor Networks Bhaskar Bhuyan1 , Hiren Kumar Deva Sarma1,* , Nityananda Sarma2**

**[8] STREP/FP7-ICT: Platform for Opportunistic Behaviour in Incompletely Specified, Heterogeneous Object Communities (POBICOS), http://www.ictpobicos.eu/index.htm.**

**[9] H.U. Heiss and M. Schmitz, "Decentralized dynamic load balancing: The particles approach," in Proc. 8th Int. Symp. on Computer and Information Sciences (ISCIS 95).**

**[10] T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," in IEEE Transactions on Software Engineering., Vol. 17 (7) , pp. 725-730, 1991.**

**[11] V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," in IEEE Transactions on Computers, Vol. 31 (11), pp. 1384-1397, 1988.**

**[12] K. Taura and A. Chien, "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources," in Proc. 9th Heterogeneous Computing Workshop (HCW 00).**

**[13] T. Agarwal, A. Sharma, A. Laxmikant and L.V. Kale, "Topologyaware task mapping for reducing communication contention on large parallel machines," in Proc. 20th Int. Parallel and Distributed Processing Symp. (IPDPS 06).**

**[14] D. Gu, F. Drews and L.R. Welch, "Robust task allocation for dynamic distributed real-time systems subject to multiple environmental parameters," in Proc. 25th Int. Conf. on Distributed Computing Systems (ICDCS 05).**

**[15] N. Tziritas, T. Loukopoulos, S. Lalis, P. Lampsas : Agent placement in wireless embedded systems: memory space and energy optimizations. Proc. 9th Int. Workshop on Performance Modeling, Evaluation and Optimization of Ubiquitous Computing and Networked Systems (PMEO-UCNS10).**

**[16] Y. Tian, J. Boangoat, E. Ekici and F. Özgüner, "Real-time task mapping and scheduling for collaborative in-network processing in DVS-enabled wireless sensor networks," in Proc. 20th Int. Parallel and Distributed Processing Symp. (IPDPS 06).**

**[17] H.S. Kim, T.F. Abdelzaher and W.H. Kwon, "Dynamic Delay- Constrained Minimum-Energy Dissemination in Wireless Sensor Networks," in ACM Trans. on Embedded Computing Systems, Vol. 4 (3), pp. 679-706, 2005.**

**[18] Q. Wu, N. S. V. Rao, J. Barhen, et al., "On computing mobile agent routes for data fusion in distributed sensor networks," in IEEE Transactions on Knowledge and Data Engineering, Vol. 16 (6), pp. 740–753, 2004.**

**[19] J. Domaszewicz, M. Roj, A. Pruszkowski, M. Golanski and K. Kacperski, "ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks," in Proc. WoWMoM 2006.**

**[20] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in Proc. ASPLOS 2002.**

**[21] C.L. Fok, G.C. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," in Proc. 25th Int. Conf. on Distributed Computing Systems (ICDCS 05).**

**[22] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall, "System support for pervasive applications", ACM Transactions on Computer Systems, 22(4), 2004.**

**[23] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, L. Iftode, "Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems", The Computer Journal, 47(4), 2004, British Computer Society, Oxford University Press.**

**[24] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R.H. Campbell, M.D. Mickunas, "Olympus: A High-Level Programming Model for Pervasive Computing Environments," 3rd IEEE International Conference on Pervasive Computing and Communications (PERCOM), 2005.**

**[25] A. Boulis, C.-C. Han, R. Shea, M.B. Srivastava, "SensorWare: Programming sensor networks beyond code update and querying", Pervasive and Mobile Computing Journal, 3(4), 2007, Elsevier.**

**[26] J. Lifton, D. Seetharam, M. Broxton, J.A. Paradiso, "Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks", 1st international Conference on Pervasive Computing, Lecture Notes In Computer Science, 2414,2002, Springer.**

**[27] B. Chen, H.H. Cheng, J. Palen, "Mobile-C: a mobile agent platform for mobile C-C++ agents", Software Practice and Experience, 36(15), 2006.**

**[28] H. Liu, T. Roeder, K. Walsh, R. Barr, E.G. Sirer, "Design and implementation of a single system image operating system for ad hoc networks", 3rd International Conference on Mobile Systems, Applications and Services (MOBISYS), 2005.**

**[29] N. Kothari, R. Gummadi, T. Millstein, R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks", ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007.**

**[30] U. Ramachandran, R. Kumar, M. Wolenetz, B. Cooper, B. Agarwalla, J. Shin, P. Hutto, A. Paul, "Dynamic data fusion for future sensor networks", ACM Transactions on Sensor Networks, 2(3), 2006.**

# Παράρτημα

## Network_generator.h

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <time.h>
#include <stdbool.h>


//A structure to represent each vertex
 typedef struct vertex{
        int vid;
        int x;
        int y;

}vertex;

// A structure to represent a node in adjacency list
struct netNode
{
   int id;//dest;
   int weight;
   int relationship;
   struct netNode* next;
};

// A structure to represent an adjacency list
struct netNodeList//AdjListA
{
   struct netNode *head;  // pointer to head node of list
};

// A structure to represent a graph.
struct Graph
{
   int V;//number of vertices
   struct netNodeList* array;//adjacency lists array
};

// create a new adjacency list node
struct netNode* newAdjListNode(int id, int weight,int relationship)
```

```
{
   struct netNode* newNode =
         (struct netNode*) malloc(sizeof(struct netNode));
   newNode->id = id;
   newNode->weight = weight;
   newNode->relationship=relationship;
   newNode->next = NULL;
   return newNode;
}

// creates a graph of V vertices
struct Graph* createGraph(int V)
{
   struct Graph* graph=(struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;

   // Create an array of adjacency lists.
   graph->array=(struct netNodeList*) malloc(V*sizeof(struct netNodeList));

    // Initialize each adjacency list as empty by making head as NULL
   int i;
          for (i=0;i<V;i++)
      graph->array[i].head=NULL;

   return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src,int id, int weight,int relationship)
{
   //The node is added at the begining
   struct netNode* newNode = newAdjListNode(id, weight,relationship);
   newNode->next = graph->array[src].head;
   graph->array[src].head = newNode;

}

// Structure to represent a min heap node
struct MinHeapNode
{
   int  v;
   int key;
};

// Structure to represent a min heap
struct MinHeap
{
   int size;      // Number of heap nodes present currently
   int capacity;  // Capacity of min heap
   int *pos;     // This is needed for decreaseKey()
   struct MinHeapNode **array;
```

```
};

//create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
{
   struct MinHeapNode* minHeapNode =
         (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
   minHeapNode->v = v;
   minHeapNode->key = key;
   return minHeapNode;
}


//create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
   struct MinHeap* minHeap =
      (struct MinHeap*) malloc(sizeof(struct MinHeap));
   minHeap->pos = (int *)malloc(capacity * sizeof(int));
   minHeap->size = 0;
   minHeap->capacity = capacity;
   minHeap->array =
      (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
   return minHeap;
}

//Swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
   struct MinHeapNode* t = *a;
   *a = *b;
   *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
   int smallest, left, right;
   smallest = idx;
   left = 2 * idx + 1;
   right = 2 * idx + 2;

   if (left < minHeap->size &&
      minHeap->array[left]->key < minHeap->array[smallest]->key )
     smallest = left;

   if (right < minHeap->size &&
      minHeap->array[right]->key < minHeap->array[smallest]->key )
     smallest = right;
```

```
    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        struct MinHeapNode *smallestNode = minHeap->array[smallest];
        struct MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is empty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
```

```
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
   // Get the index of v in  heap array
   int i = minHeap->pos[v];

   // Get the node and update its key value
   minHeap->array[i]->key = key;
   // Travel up while the complete tree is not hepified.
   // This is a O(Logn) loop
   while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key)
   {
      // Swap this node with its parent
      minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
      minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
      swapMinHeapNode(&minHeap->array[i],  &minHeap->array[(i - 1) / 2]);

      // move to parent index
      i = (i - 1) / 2;
   }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
  if (minHeap->pos[v] < minHeap->size)
    return true;
  return false;
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
int PrimMST(struct Graph* graph,int parent[],int V,int key[])
{
   // minHeap represents set E
   struct MinHeap* minHeap = createMinHeap(V);

   // Initialize min heap with all vertices. Key value of
   // all vertices (except 0th vertex) is initially infinite
   int v;
          for (v = 1; v < V; ++v)
   {

                    parent[v] = -1;
      key[v] = INT_MAX;
      minHeap->array[v] = newMinHeapNode(v, key[v]);
      minHeap->pos[v] = v;
   }

   // Make key value of 0th vertex as 0 so that it
   // is extracted first
```

```
   key[0] = 0;
   minHeap->array[0] = newMinHeapNode(0, key[0]);
   minHeap->pos[0]   = 0;

   // Initially size of min heap is equal to V
   minHeap->size = V;

   // In the followin loop, min heap contains all nodes
   // not yet added to MST.
   while (!isEmpty(minHeap))
   {
      // Extract the vertex with minimum key value
      struct MinHeapNode* minHeapNode = extractMin(minHeap);
      int u = minHeapNode->v; // Store the extracted vertex number

      // Traverse through all adjacent vertices of u (the extracted
      // vertex) and update their key values
      struct netNode* pCrawl = graph->array[u].head;
      while (pCrawl != NULL)
      {
         int v = pCrawl->id;

         // If v is not yet included in MST and weight of u-v is
         // less than key value of v, then update key value and
         // parent of v
         if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])
         {
            key[v] = pCrawl->weight;
            parent[v] = u;
            decreaseKey(minHeap, v, key[v]);
         }
         pCrawl = pCrawl->next;
      }
   }
   // print edges of MST
   return *parent;
   return *key;
}



struct Graph* net_gen(int N,int m,int l)

{
   //create the graph
   struct Graph* graph=createGraph(N);
  vertex *varray=(vertex*)malloc(graph->V*sizeof(vertex));

int i,j;
int mstArray[N];
int weights[N];
```

```
for(i=0;i<graph->V;i++){

        varray[i].vid=i;
        varray[i].x=rand()%m;
        varray[i].y=rand()%m;
        printf("%d %d %d\n",varray[i].vid,varray[i].x,varray[i].y);
}


int k=0;

int weightl;
for(i=0;i<graph->V;i++){
        for(j=0;j<graph->V;j++){
                weightl=sqrt(((varray[i].x-varray[j].x)*(varray[i].x-varray[j].x))+((varray[i].y-
varray[j].y)*(varray[i].y-varray[j].y)));
                if(weightl<l && i!=j)
                {
                  addEdge(graph,i,j,weightl,0);

                }


        }

}

PrimMST(graph,mstArray,N,weights);

for(i=1;i<N;i++){
        printf("%d-%d\n",mstArray[i],i);
}

struct Graph* network=createGraph(N);

for(i=0;i<N;i++){
        for(j=1;j<N;j++){
        if(i==mstArray[j]){
           addEdge(network,i,j,weights[j],0);
        }
        if(i==j){
                addEdge(network,j,mstArray[j],weights[j],1);
        }
}
}
for(i=0;i<network->V;i++){
addEdge(network,i,i,0,1);
}


return network;
```

```
}
```

## Application_generator.h

```
//app_generator
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include <time.h>
#include<assert.h>
#include <ctype.h>

//a structure to represent each agent
typedef struct Agent
{
   int agent;
   int load;
   struct Agent* next;
}*AgentPtr;

// A structure to represent an adjacency list
struct Agent_List
{
   struct Agent *head;
};


//A structure to represent an application tree
struct App_tree
{
   int K;//number of node-neutral agents
   struct Agent_List* array;//list array
};

struct Applcation{
   struct App_tree tree;
};

// create a new adjacency list node
struct Agent* newAdjListNode(char agent,int size)
{
   struct Agent* newNode =
         (struct Agent*) malloc(sizeof(struct Agent));
   newNode->agent = agent;
   newNode->load=size;
```

```
   newNode->next = NULL;
   return newNode;
}

struct App_tree* create_app(int k)
{
   struct App_tree* adj_tree=(struct App_tree*)malloc(sizeof(struct App_tree));
   adj_tree->K = k;

   // Create an array of adjacency lists.
   adj_tree->array=(struct Agent_List*) malloc(k*sizeof(struct Agent_List));

    // Initialize each adjacency list as empty by making head as NULL
   int i;
         for ( i=0;i<k;i++)
      adj_tree->array[i].head=NULL;

   return adj_tree;
}

// Adds an agent to agents' adjacency list
void addEdge(struct App_tree* app_tree, char agent, int size,int j)
{
   struct Agent* newNode = newAdjListNode(agent, size);
   newNode->next = app_tree->array[j].head;
   app_tree->array[j].head = newNode;

}

// Adds an agent to agents' adjacency list
void add(struct App_tree* app_tree, char agent, int size,int j)
{
         struct Agent* cur;
         struct Agent* newNode = newAdjListNode(agent, size);

         cur=app_tree->array[j].head;
         while(cur->next!=NULL){
                  cur=cur->next;
         }
         cur->next=newNode;
}

//generates a random number
int random_number(int *x){
         do{
         *x=rand()%6;
   }while(*x<2 );
```

```
        return *x;

}

//pick a random agent
int random_agent(int vector[],int x){


  int SIZE=5;
  int in,im;

  im = 0;

for (in = 0; in <SIZE && im <x; ++in) {
 int rn = SIZE - in;
 int rm = x - im;
 if (rand() % rn < rm)
   vector[im++] = in;

}
assert(im == x);
}


//create a new application tree
struct App_tree* app_gen(int a, int b)
{

   int *S;//node-specific array
   int *N;//node-neutral array
        int i,j;
   int n,x=0;
   int vector[x];

   struct App_tree *application=create_app(a+b);
   struct Agent* newNode; //points to agent node

   N=(int *)malloc(b*sizeof(int));//node neutral array
        printf("the array of node-neutral agents is:\n");
   for(i=0;i<b;i++){
                N[i]=i;
        }


   j=-1;
```

```
S=(int *)malloc(a*sizeof(int)); //node specific array
printf("the array of node-specific agents is:\n");
for(i=b;i<a+b;i++){
        j++;
          S[j]=i;
}
for(i=0;i<a;i++){
                printf("%d\n",S[i]); //print node specific array
}



         for(i=0;i<b;i++){
                printf("%d\n",N[i]); //print node neutral array
}



        int l=0,u=0,k=0,s=0,w;
        int *array,*array_temp;
        int  v=-1,counter=0,length,n_length=b,piece,m,nl=0,repeat=0;  //n_length:the
        length of node-neutral agents' array

        while(repeat<b){
        if(counter==0){
                length=a;//length:the length of temp array of orphan agents
                m=length;
                array_temp=(int*)malloc(length*sizeof(int));
                for(i=0;i<length;i++)
                {
                        array_temp[i]=S[i];
                }


        for(i=0;i<length/5;i++)
        {

    random_number(&x);
                random_agent(vector,x);
                l=l+x;//counts the number of agents that acquire a father
                nl++;
                repeat++;
         for(j=0;j<x;j++){

                  vector[j]=vector[j]+(i*5);
                  addEdge(application,array_temp[vector[j]],rand()%5+1,n_length-nl);
                  array_temp[vector[j]]=-1;
 }
```

```
          addEdge(application,N[n_length-nl],rand()%5+1,n_length-nl);
    }
        }
        else
        {

                length=k;
                m=length;
                array_temp=(int*)malloc(length*sizeof(int));
                for(i=0;i<length;i++){
                        array_temp[i]=array[i];

                }

                free(array);

                if(m/5!=0){
        for(i=0;i<length/5;i++)
         {
                random_number(&x);
                random_agent(vector,x);
                l=l+x;//counts the number of agents that acquire a father
                nl++;
                for(j=0;j<x;j++){
                  vector[j]=vector[j]+(i*5);
                  addEdge(application,array_temp[vector[j]],rand()%5+1,n_length-nl);
                  array_temp[vector[j]]=-1;
         }


                if(N[n_length-nl]!=0){
                        addEdge(application,N[n_length-nl],rand()%5+1,n_length-nl);
                }
                                m=m-5;
                   repeat++;
                }
        }

           if(m%5!=0){
                nl++;
                for(i=0;i<m%5;i++){
                        addEdge(application,array_temp[(length-
length%5)+i],rand()%5+1,n_length-nl);
        array_temp[length-(length%5)+i]=-1;
        l++;
                        }
                        if(N[n_length-nl]!=0){
```

```
                        addEdge(application,N[n_length-nl],rand()%5+1,n_length-nl);
                }
                        repeat++;
            }
                if(N[n_length-nl]==0){
                        for(i=0;i<length;i++){
                                if(array_temp[i]!=-1){

        addEdge(application,array_temp[i],rand()%5+1,n_length-nl);
                                }
                        }
                }

        i=0;
        }
        k=length/5+(length-l);
        if(length%5!=0){
                k++;
         }
        array=(int*)malloc(k*sizeof(int));

   for(i=0;i<length;i++){
      if(array_temp[i]!=-1){
         v++;
                        array[v]=array_temp[i];
         u++;
      }
   }

   for(i=0;i<length/5;i++){
      v++;
      s++;
      if(N[b-s]!=0){
         array[v]=N[b-s];
      }
      if(N[n_length-nl]==0 && N[b-s]!=0){

         addEdge(application,N[b-s],rand()%5+1,n_length-nl);


                }
}


if(length%5!=0){
         v++;
         s++;
```

```
        if(N[b-s]!=0){
        array[v]=N[b-s];
   }
   if(N[n_length-nl]==0 && N[b-s]!=0){

                addEdge(application,N[b-s],rand()%5+1,n_length-nl);
   }

}
if(N[n_length-nl]==0){
   addEdge(application,N[n_length-nl],rand()%5+1,n_length-nl);
}
v=-1;

free(array_temp);

l=0;
counter++;
}


int t=b;
j=t-1;
for(i=0;i<a;i++){
        j++;
        addEdge(application,S[i],0,j);
}

int sum=0,avg,count=0;
AgentPtr tmp,tmp2;
for(i=b-1;i>=0;i--){
        count=0;
        sum=0;
        tmp=application->array[i].head;
        tmp->load=0;
        tmp2=application->array[tmp->agent].head;
        if(tmp2->next!=NULL){
                        tmp2=tmp2->next;

while(tmp2!=NULL){
 count=count+1;
 sum=sum+tmp2->load;
 tmp2=tmp2->next;
}

avg=sum/count;
}
```

```
for(j=0;j<b;j++){
	tmp2=application->array[j].head;
	while(tmp2!=NULL){
		if(tmp2->agent==tmp->agent){
			printf("avg is:%d\n",avg);
		//	tmp2->load=sum; // load(sum)
	/*if(tmp2->agent==tmp->agent){ // load(mix)
			//printf("avg is:%d\n",avg);
			tmp2->load=sum;
			for(k=0;k<b/2;k++){
				if(tmp->agent==mix_array[k]){
					tmp2->load=avg;
					printf("%d\n",tmp2->agent);
					printf("avg is:%d\n",avg);
				}

			}
		//	tmp2->load=sum;
			//tmp2->load=avg;
		}*/
			tmp2->load=avg; // load(avg)
		}
		tmp2=tmp2->next;
	}
}
tmp->load=0;
}return application;

system("pause");

}
```

## Network_placement.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <list>


struct nodeA
{
	int id;
	int check;
	struct nodeA* next;
```

```
};

struct nodeAList
{
        struct nodeA* head;
};

struct Placement_List{
        int N;
        struct nodeAList* array;
};

struct nodeA* createNode(int id,int check){
        struct nodeA* newNode;
        newNode=(struct nodeA*) malloc(sizeof(struct nodeA));
        newNode->id=id;
        newNode->check=check;
        newNode->next = NULL;
   return newNode;
}

struct Placement_List* create(int n){
        struct Placement_List* p_list=(struct Placement_List*
)malloc(sizeof(Placement_List));
        p_list->N=n;
        p_list->array=(struct nodeAList*)malloc(n*sizeof(nodeAList));
        int i;
        for(i=0;i<n;i++){
                p_list->array[i].head=NULL;
        }
        return p_list;
}

void addNode(struct Placement_List* Plist,int id,int j,int check){
                struct nodeA* newNode=createNode(id,check);
                newNode->next= Plist->array[j].head;
                Plist->array[j].head=newNode;
}

// A utility function to swap to integers
void swap (int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

```
// A utility function to print an array
void printArray (int arr[], int n)
{
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}

// A function to generate a random permutation of arr[]
void randomize ( int arr[], int n )
{
   // Use a different seed value so that we don't get same
   // result each time we run this program

   // Start from the last element and swap one by one. We don't
   // need to run for the first element that's why i > 0
   for (int i = n-1; i > 0; i--)
   {
      // Pick a random index from 0 to i
      int j = rand() % (i+1);

      // Swap arr[i] with the element at random index
      swap(&arr[i], &arr[j]);
   }
}

//delete an agent from adjacency list
struct Placement_List* delete_agent(struct Placement_List* Plist,int id,int n){
          struct nodeA *temp;
          struct nodeA *previous;
          int i;
          previous=Plist->array[n].head;
            temp=previous->next;
            while(temp!=NULL){
                    if(temp->id==id){
                              previous->next=temp->next;
                              free(temp);

                    }
                    previous=temp;
                    temp=temp->next;
            }
return Plist;
}

//adds an agent in agents' adjacency list
struct Placement_List* add_agent(struct Placement_List *Plist,int id,int n){
```

```
        nodeA *cur;
                struct nodeA* newNode;
                        newNode=(struct nodeA*) malloc(sizeof(struct nodeA));
        newNode->id=id;
        newNode->next = NULL;
        cur=Plist->array[n].head;
        while(cur->next!=NULL){
                cur=cur->next;
        }
        cur->next=newNode;
        return Plist;

}

//assigns agents in network nodes
struct Placement_List* assignments_gen(struct Graph *net, struct App_tree *app,
int a, int b,int N){


   struct Placement_List* Pl_list=create(N);//Agent placement on network

        struct netNode *net_temp;
        struct Agent *app_temp;

        int net_array[N];//array with netwok's nodes ids
        int app_array[a+b];//array with app's agents ids

        int i;
        for(i=0;i<N;i++){
          net_temp=net->array[i].head;
          net_array[i]=net_temp->id;
        }

  for(i=0;i<a+b;i++){
            app_temp=app->array[i].head;
            app_array[i]=app_temp->agent;
  }

  randomize(net_array, N);

  randomize(app_array, a+b);

        int group,count=0,count_n=0;
        int n=a+b,j;

    while(n>0){
         if(count<19){
```

```
            group = rand() % n+1;
            for(i=0;i<group;i++){
                    addNode(Pl_list,app_array[count_n],count,0);
                    count_n++;
                     }
            addNode(Pl_list,net_array[count],count,0);
            n=n-group;
            count++;

    }
    else{//in case more than 20 repetition are executed
          group=n;
          break;
    }
           }
           for(i=count;i<N;i++){
                    addNode(Pl_list,net_array[i],i,0);
           }

            nodeA* s;
            for (i = 0; i < N; ++i)
   {
      for (j = i + 1; j < N; ++j)
      {
         if (Pl_list->array[i].head->id >Pl_list->array[j].head->id)
         {
            s =  Pl_list->array[i].head;
            Pl_list->array[i].head = Pl_list->array[j].head;
            Pl_list->array[j].head = s;
         }
      }
   }

    nodeA* current_p;

    for(i=0;i<N;i++){
            current_p=Pl_list->array[i].head;
           while(current_p!=NULL){
      current_p=current_p->next;
           }
           }
  return Pl_list;
}
```

## Subtree.h

```
#include <stdio.h>
#include <stdlib.h>

//a structure to represent an agent
struct s_node{
	int n_id;
	int parent;
	int check;//-1:root,1:leaf,-1:intermediate
	int dest;
	struct s_node *next;
};

//a structure to represent an adjacency list of agents
struct s_List{
	struct s_node *head;
};

//a structure to represent a subtree of agents
struct Subtree{
	int size;
	struct s_List *array;
};


//creates a new node in adjacency list
struct s_node* newNode(int n_id, int parent,int check,int dest){
	struct s_node *newnode;

	newnode=(s_node*)malloc(sizeof(s_node));

	newnode->n_id=n_id;
	newnode->parent=parent;
	newnode->check=check;
	newnode->dest=dest;
	newnode->next=NULL;

	return newnode;
}

//allocates memory for a new subtree
struct Subtree *createSubtree(int size){

	struct Subtree * subtree;
```

```
        subtree=(struct Subtree*)malloc(sizeof(struct Subtree));

        subtree->size=size;
            subtree->array=(s_List*)malloc(size*sizeof(s_List));


        int i;
        for(i=0;i<size;i++){
                subtree->array[i].head=NULL;
        }
        return subtree;

}

//adds an agent in adjacency list
struct s_node *add_agent(struct Subtree *stree,int n_id,int parent,int check,int i,int
dest){
        struct s_node* newnode= newNode(n_id,parent,check,dest);

        newnode->next=stree->array[i].head;
        stree->array[i].head=newnode;
   return newnode;
}

//creates a new subtree
struct Subtree * subtree_generation(int size){

        struct Subtree *subtree=createSubtree(size);

        return subtree;
}
```

## Temporary_subtree.h

```
#include <stdio.h>
#include <stdlib.h>

// a struct to represent an agent
struct node{
        int n_id;
        int parent;
        int former_parent;
        int pb;
        int tmp_pb;
        int check;
        int pruned;
        int root;
```

```
	int affinity;
	int load_neutral;
	struct node *next;
};

//a struct to represent an adjacency list of agents
struct nodeList
{
	struct node *head;
};

//a struct to represent a temporary subtree
struct tmp_Subtree{
	int size;
	struct nodeList *array;
};


//creates a new node in adjacency list
struct node* Newnode(int n_id, int parent,int former_parent,int pb,int tmp_pb,int
check,int pruned,int root,int affinity,int load_neutral){
	struct node *newnode;

	newnode=(node*)malloc(sizeof(node));

	newnode->n_id=n_id;
	newnode->parent=parent;
	newnode->former_parent=former_parent;
	newnode->pb=pb;
	newnode->tmp_pb=tmp_pb;
	newnode->check=check;
	newnode->pruned=pruned;
	newnode->root=root;
	newnode->affinity=affinity;
	newnode->load_neutral=load_neutral;
	newnode->next=NULL;

	return newnode;
}

struct tmp_Subtree *createS(int size){

	struct tmp_Subtree * subtree;

	subtree=(struct tmp_Subtree*)malloc(sizeof(struct tmp_Subtree));

	subtree->size=size;
```

```
   subtree->array=(nodeList*)malloc(size*sizeof(nodeList));


        int i;
        for(i=0;i<size;i++){
                subtree->array[i].head=NULL;
        }
        return subtree;

}

struct node *add(struct tmp_Subtree *stree,int n_id,int parent,int former_parent,int
pb,int tmp_pb,int i,int check,int pruned,int root,int affinity,int load_neutral)
{
        struct   node*  newnode=
Newnode(n_id,parent,former_parent,pb,tmp_pb,check,pruned,root,affinity,load_ne
utral);

        newnode->next=stree->array[i].head;
        stree->array[i].head=newnode;
   return newnode;
}

struct loads{
        int agent_specific;
        int agent_neutral;
        int agent_adest;
        int agent_dest;
        int affinity;
        int load_parent;
        int parent;
};



struct tmp_Subtree * subtree_gen(int size){

        struct tmp_Subtree *subtree=createS(size);

        return subtree;
}
```

## Gral_implementation.h

```
#include "app_generator.h"
#include "network_generator.h"
#include "Network_placement.h"
#include "Subtree.h"
#include "Temporary_subtree.h"
#define MAX 100
const int N=20;    //network nodes
const int m=80;    //network distance units
const int l=30;    //minimum Euclidean distance


struct array_node{
        int dest;
        int load;
};

struct agent_node{
        int id;
        int pos;
};

struct neutral_node{
        int id;
        int pos;
        int parent;
};

typedef struct agent_load{
        int id;
        int check;
        int parent;
       int pb;
       int aff;
       int l_neutral;
       int l_parent;
}agent_load;

struct agent{
        int id;
        int parent;
        int pb;
};
```

```
struct agent_review{
            int id;
            int check;
            int parent;
           int pb;
};


//queue for BFS
typedef struct Q
{
 int data[MAX];
 int R,F;
}Q;

int empty(Q *P)
{
 if(P->R==-1)
  return(1);
 return(0);
}

int full(Q *P)
{
 if(P->R==MAX-1)
  return(1);
 return(0);
}

void enqueue(Q *P,int x)
{
 if(P->R==-1)
 {
  P->R=P->F=0;
  P->data[P->R]=x;
 }
 else
 {
  P->R=P->R+1;
  P->data[P->R]=x;
 }
}

int dequeue(Q *P)
{
 int x;
 x=P->data[P->F];
```

```
 if(P->R==P->F)
 {
  P->R=-1;
  P->F=-1;
 }
 else
  P->F=P->F+1;
 return(x);
}

//BFS algorithm
int BFS(int s,int d,struct Graph * n)
{
 netNode* current;
 Q q;
 int i,w;
 q.R=q.F=-1;
 if(s==d){
          return s;
 }

 bool visited[N];
 for(i=0;i<N;i++){
          visited[i]=false;
 }

 visited[s]=true;
 enqueue(&q,s);
 while(!empty(&q))
 {
  s=dequeue(&q);
  //insert all unvisited,adjacent vertices of v into queue
  current=n->array[s].head;
  while(current!=NULL)
  {
   w=current->id;
   if(w==d)
   {
    return s;

   }
   if(!visited[w]){
          visited[w] = true;
          enqueue(&q,w);
   }
   current=current->next;
  }
```

```
 }
free(current);
}

void temp_arrays(struct agent_node temp_app[], int n, struct neutral_node temp_neutral[] , int b, struct App_tree *app, struct Graph *net, struct Placement_List *Pl_list){

        Agent* current_a;//pointer to application
        nodeA* current_p;//pointer to f_network
        netNode * current_n;

        int i,j,position;

        for(i=0;i<n;i++){
                current_a=app->array[i].head;
                temp_app[i].id=current_a->agent;
        }

        for(i=0;i<b;i++){
                current_a=app->array[i].head;
                temp_neutral[i].id=current_a->agent;
        }

        //position review for app's agents
   for(i=0;i<N;i++){

                 position=Pl_list->array[i].head->id;//to position tou agent sto teliko network
                 current_p=Pl_list->array[i].head;
                 current_p=current_p->next;

            while( current_p!=NULL){
                        for(j=0;j<n;j++){
                                if(temp_app[j].id==current_p->id){
                                  temp_app[j].pos=position;
                                }
                                }
                        if(current_p->id<b){
                                for(j=0;j<b;j++){
                                        if(temp_neutral[j].id==current_p->id){
                                                temp_neutral[j].pos=position;
                                        }
                                }
                        }
                        current_p=current_p->next;
                        }
```

```
   }
}

//fills review_array with the load exchanged between app's agents
void position_review(struct array_node **review_array, int s, int b, int a,struct App_tree *app,struct agent_node temp_app[], int x, struct neutral_node temp_neutral[] , int y,struct Graph *net){

        Agent* current_a;//pointer to application

        int i,j;
        int t,m,n=a+b;


        for(i=0;i<s;i++)
        {
                for(j=0;j<b;j++)
                {
                        review_array[i][j].dest=-1;
                        review_array[i][j].load=-1;
                }
        }

          //print application tree
                for(i=0;i<a+b;i++){
                        current_a=app->array[i].head;
                        while(current_a!=NULL){
                                current_a=current_a->next;
                        }
                        printf("\n");
                }

                int parent,pos_p,pos_c,tmp,d_vertex;

                temp_neutral[0].parent=-1;


                        for(i=0;i<b;i++){
                        parent=app->array[i].head->agent;
                        current_a=app->array[i].head;
                        current_a=current_a->next;
                        for(j=0;j<b;j++){
                                if(temp_neutral[j].id==parent){
                                        pos_p=temp_neutral[j].pos;
                                        tmp=j;
                                }
                        }
```

```
                while(current_a!=NULL){
                for(j=0;j<n;j++){
                                if(temp_app[j].id==current_a->agent){
                                        pos_c=temp_app[j].pos;
        d_vertex=BFS(pos_c,pos_p,net);
        review_array[j][tmp].dest=d_vertex;
        review_array[j][tmp].load=current_a->load;
          if(current_a->agent<b){
             d_vertex=BFS(pos_p,pos_c,net);
                for(t=0;t<n;t++){
                    if(temp_app[t].id==parent){
                        for(m=0;m<b;m++){
                           if(temp_neutral[m].id==current_a->agent){
                                temp_neutral[m].parent=parent;
                                review_array[t][m].dest=d_vertex;
                                review_array[t][m].load=current_a>load;
                           }
                     }
                }
        }
     }
   }
 }
current_a=current_a->next;
 }
 }
}

//sets the destination for every identified subtree
void destination_review(struct Subtree* subtree,int size,struct array_node
**review_array,int x,int b,struct Graph *net,struct neutral_node temp_neutral[],int
y,int n_id){

   int i,j,max,dest,sum,sum_2,l,count,sum_specific,sum_neutral,sum_adest;


   struct s_node *agent;
   netNode* current_n;//points to network tree

   for(i=0;i<size;i++){//check every subtree

        max=-1;
        sum_2=0;
        dest=-1;

        //track the destination for each subtree
   current_n=net->array[n_id].head;
```

```
   current_n=current_n->next;//points to nodes that communicate with the node we
check
      while(current_n!=NULL){
         sum=0;
         sum_specific=0;
         sum_neutral=0;

         agent=subtree->array[i].head;//points to the subtree
         count=0;
         while(agent!=NULL){//check every agent on the subtree

          for(j=0;j<b;j++){
                   if(temp_neutral[j].id==agent->n_id){
                            for(l=0;l<x;l++){
                                     if(review_array[l][j].dest==current_n->id){
                                              sum=sum+review_array[l][j].load;
                                     }
                                     if(l>=b && review_array[l][j].dest==n_id){

         sum_specific=sum_specific+review_array[l][j].load;
                                     }
                                     if(l<b && review_array[l][j].dest==n_id){

         sum_neutral=sum_neutral+review_array[l][j].load;
                                     }
                            }
                   }
         }
   count++;
         agent=agent->next;

}//end_of while for agents
if(sum>max){//max represents the node that the subtree will be transfered
         dest=current_n->id;//dest represents the node that the subtree will be
transfered
         max=sum;
}
current_n=current_n->next;
}//end_of_while

sum_adest=sum_2-max;
if(max>sum_specific){
         agent=subtree->array[i].head;
         while(agent!=NULL){
                   agent->dest=dest;
                   agent=agent->next;
          }
```

```
}
}

//checks if an agent has children
int has_no_children(int i,struct tmp_Subtree *tmp_subtree){
        struct node *tmp;
        int child=0;
        tmp=tmp_subtree->array[i].head;
        tmp=tmp->next;
        while(tmp!=NULL){
         if(tmp->check==0){
                child++;
                 }
         tmp=tmp->next;
         }
         if(child>0){
                return 0;
         }
         else{
           return 1;//if there are no children
         }

}

//prunes subtree
void prune_subtree(struct tmp_Subtree *tmp_subtree,int b){
 struct node* tmp;
 struct node* tmp2;
 int i,action,j,count=0;

do{
        action=0;
        count++;
for(i=0;i<b;i++){
        tmp=tmp_subtree->array[i].head;
        if(tmp!=NULL){
        if(tmp->next==NULL && tmp->check==0){
                action=1;
                if(tmp->pb>0){
                        tmp->pruned=1;
                        tmp->check=1;

                                tmp2=tmp_subtree->array[tmp->parent].head;
                                tmp2->pb=tmp2->pb+tmp->pb;
                        while(tmp2!=NULL){
                                if(tmp2->n_id==tmp->n_id)
                                {
```

```
                                tmp2->check=1;
                                tmp2->pruned=1;
                        }
                        tmp2=tmp2->next;
                }
        }
        else{
                action=1;
                tmp->check=1;
                tmp2=tmp_subtree->array[tmp->parent].head;
                while(tmp2!=NULL){
                        if(tmp2->n_id==tmp->n_id)
                        {
                                tmp2->check=1;
                        }
                        tmp2=tmp2->next;
                }
        }
}
if(tmp->next!=NULL && tmp->check==0){
        action=1;
        if(has_no_children(i,tmp_subtree)==1){
                if(tmp->pb>0){
                        tmp->check=1;
                        tmp->pruned=1;
                        if(tmp->parent!=-1){
                        tmp2=tmp_subtree->array[tmp->parent].head;
                        tmp2->pb=tmp2->pb+tmp->pb;
                while(tmp2!=NULL){
                        if(tmp2->n_id==tmp->n_id)
                        {
                                tmp2->check=1;
                                tmp2->pruned=1;
                        }
                        tmp2=tmp2->next;
                }
        }
                }
                else{
                        tmp->check=1;
                        if(tmp->parent!=-1){
                        tmp2=tmp_subtree->array[tmp->parent].head;
                while(tmp2!=NULL){
                        if(tmp2->n_id==tmp->n_id)
                        {
                                tmp2->check=1;
                        }
```

```
                                        tmp2=tmp2->next;
                                }
                        }
                                }
                        }
                }

}
}
}while(action==1);

for(i=0;i<b;i++){
        tmp=tmp_subtree->array[i].head;
        if(tmp!=NULL){
        if(tmp->pruned==0){
                tmp=tmp->next;
                while(tmp!=NULL){
                        tmp->pruned=0;
                        tmp2=tmp_subtree->array[tmp->n_id].head;
                        tmp2->pruned=0;
                        tmp=tmp->next;
                }
        }
}
}


}

//deletes a node from tmp_subtree
struct tmp_Subtree* deleteNode(struct tmp_Subtree *tmp_subtree,int id,int i,int b){
        struct node *temp;
        struct node *previous;

        previous=tmp_subtree->array[i].head;
          temp=previous->next;
          while(temp!=NULL){
                if(temp->n_id==id){
                        previous->next=temp->next;
                        free(temp);

                }
                previous=temp;
                temp=temp->next;
          }

return tmp_subtree;
```

```
}

//adds a node to tmp_subtree
struct tmp_Subtree* add_a(struct tmp_Subtree *tmp_subtree,int i,int id,int
parent,int pb,int check,int pruned){
        struct node *cur;
                struct node* newNode;
                        newNode=(struct node*) malloc(sizeof(struct node));
        newNode->n_id=id;
        newNode->parent=parent;
        newNode->pb=pb;
        newNode->check=check;
        newNode->pruned=pruned;
        newNode->next = NULL;
        cur=tmp_subtree->array[i].head;
        while(cur->next!=NULL){
                cur=cur->next;
        }
        cur->next=newNode;
        return tmp_subtree;

}

//rotates tmp_subtree to define a new root
void rotate(struct tmp_Subtree *tmp_subtree,int b){
        struct node *tmp;
        struct node *tmp2;
        int i,count=0,former_root,former_parent;

        for(i=0;i<b;i++){
                tmp=tmp_subtree->array[i].head;
                if(tmp!=NULL){
                        if(tmp->root==1){
                                tmp->root=-1;
                                former_root=tmp->n_id;
                        }
                        if(tmp->root==0){
                                tmp->root=1;
                                do{
            tmp2=tmp_subtree->array[tmp->former_parent].head;
            deleteNode(tmp_subtree,tmp->n_id,tmp2->n_id,b);
            add_a(tmp_subtree,tmp->n_id,tmp->former_parent,tmp->n_id,tmp2-
>pb,0,0);
              if(tmp->root==1){
                 tmp->parent=-1;
              }
            tmp2->parent=tmp->n_id;
```

```
                tmp=tmp_subtree->array[tmp2->n_id].head;
                        }while(tmp->former_parent!=-1);
                break;
                        }
                }

        }


}



void resetting(struct tmp_Subtree *tmp_subtree,int b){

        struct node *tmp;
   int i,pb;

   for(i=0;i<b;i++){
        tmp=tmp_subtree->array[i].head;
        while(tmp!=NULL){
                if(tmp!=NULL){
                tmp->pruned=0;
                tmp->check=0;
                pb=tmp->tmp_pb;
                tmp->pb=pb;
                tmp->former_parent=tmp->parent;
                tmp=tmp->next;
        }
   }
   }

}

//checks which agent hasn't been root yet in tmp_subtree
int rooting(struct tmp_Subtree *tmp_subtree,int b,int root){
        struct node *tmp;
        int i;

        for(i=0;i<b;i++){
                tmp=tmp_subtree->array[i].head;
                if(tmp!=NULL && tmp->root==0){
                        return 1;
                }
        }

        return 0;
```

```
}

void print_tmp_subtree(struct tmp_Subtree *tmp_subtree,int b){
	struct node *tmp;
	int j;

		for(j=0;j<b;j++){
		tmp=tmp_subtree->array[j].head;
		while(tmp!=NULL){
			printf("%d-%d-%d-%d-%d-%d-%d-%d->",tmp->n_id,tmp-
>parent,tmp->former_parent,tmp->pb,tmp->tmp_pb,tmp->check,tmp->pruned,tmp-
>root);
			fprintf(fp,"%d-%d-%d-%d-%d-%d-%d-%d->",tmp->n_id,tmp-
>parent,tmp->former_parent,tmp->pb,tmp->tmp_pb,tmp->check,tmp->pruned,tmp-
>root);
			tmp=tmp->next;
		}
		printf("\n");

	}


}

//calculates total migration benefit of a subtree
void find_total_benefit(struct tmp_Subtree *tmp_subtree,int b,int* total_benefit){
	struct node *tmp;
	int j;

	for(j=0;j<b;j++){
		tmp=tmp_subtree->array[j].head;
		if(tmp!=NULL && tmp->parent==-1){
			*total_benefit=tmp->pb;
		}
	}

}

//checks an agent's migration
void migration_check(struct tmp_Subtree *tmp_subtree,int b,struct Subtree
*subtree,int size,int i){
	struct s_node *agent;
	struct node *tmp;

	agent=subtree->array[i].head;
	while(agent!=NULL){
		tmp=tmp_subtree->array[agent->n_id].head;
```

```
                if(tmp->pruned==1){
                        agent->check=1;
                }
                else{
                        agent->check=0;
                }
                agent=agent->next;
        }

}

//checks the case of a single agent's migration
void check_single_agent(struct tmp_Subtree *tmp_subtree,struct Subtree
*subtree,int i){
   struct s_node *agent;
        struct node *tmp;

        agent=subtree->array[i].head;

        tmp=tmp_subtree->array[agent->n_id].head;
        if(tmp->pb>0){
                agent->check=1;
        }

}

//calculates the partial benefit of a subtree with a new root
void find_new_pb(struct tmp_Subtree *tmp_subtree,struct array_node
**review_array,int x,int b){

        struct node *tmp;
        int i,r,load_parent;

        for(i=0;i<b;i++){
                tmp=tmp_subtree->array[i].head;
                if(tmp!=NULL){
                        for(r=0;r<x;r++){
                           if(r==tmp->parent){
                                        load_parent=review_array[r][tmp->n_id].load;
                                }

                        }
                        if(tmp->parent==-1){
                                tmp->pb=tmp->affinity-tmp->load_neutral;
                        }
                        else{
```

```
                                tmp->pb=tmp->affinity-tmp-
>load_neutral+2*load_parent;
                            }
                    }
            }


}

//finds the best subtree
void subtree_identification(struct Subtree *subtree,int size,struct array_node
**review_array,int x,int b,int n_id){

   struct tmp_Subtree *tmp_subtree;
   struct s_node *agent;
   struct node *tmp_agent;
   struct node *tmp_a;
   struct agent agent_r[b];

   netNode* current_n;//points to network tree

   struct loads tmp_array[b];
          int i,root,r,agent_id,j;
          int
agent_neutral,agent_spec,agent_dest,agent_adest,agent_parent,pb,count,total_ben
efit,f_total_benefit=0;


          for(i=0;i<b;i++){
                    tmp_array[i].agent_adest=0;
                    tmp_array[i].agent_dest=0;
                    tmp_array[i].agent_neutral=0;
                    tmp_array[i].agent_specific=0;
                    tmp_array[i].affinity=0;
                    tmp_array[i].load_parent=0;
                    agent_r[i].id=-1;
                    agent_r[i].parent=-2;
                    agent_r[i].pb=-1;
          }

          for(i=0;i<size;i++)//check every subtree
          {
                    tmp_subtree=subtree_gen(b);
                    agent=subtree->array[i].head;

                    if(agent->dest!=-1){
```

```
                count=0;
            while(agent!=NULL){
                agent_neutral=0;
                agent_spec=0;
                agent_dest=0;
                agent_adest=0;
                agent->check=0;
                pb=0;

        agent_r[agent->n_id].id=agent->n_id;
        agent_r[agent->n_id].parent=agent->parent;


                for(r=0;r<x;r++){
                    if(r<b && review_array[r][agent->n_id].dest==n_id){

    agent_neutral=agent_neutral+review_array[r][agent->n_id].load;
                    }
                    if(r>=b && review_array[r][agent->n_id].dest==n_id){
                        agent_spec=agent_spec+review_array[r][agent-
>n_id].load;
                    }
                    if(review_array[r][agent->n_id].dest==agent->dest){
                        agent_dest=agent_dest+review_array[r][agent-
>n_id].load;
                    }
                    if(review_array[r][agent->n_id].dest!=agent->dest    &&
review_array[r][agent->n_id].dest!=n_id && review_array[r][agent->n_id].dest!=-1){

    agent_adest=agent_adest+review_array[r][agent->n_id].load;
                    }
                    if(r==agent->parent){
                        tmp_array[agent-
>n_id].load_parent=review_array[r][agent->n_id].load;
                    }

                }

                tmp_array[agent->n_id].agent_neutral=agent_neutral;
                tmp_array[agent->n_id].agent_specific=agent_spec;
                tmp_array[agent->n_id].agent_dest=agent_dest;
                tmp_array[agent->n_id].agent_adest=agent_adest;

                tmp_array[agent->n_id].affinity=agent_dest-agent_spec-
agent_adest;//calculate affinity
                fprintf(fp,"%s""%d\n","affinity          is:",tmp_array[agent-
>n_id].affinity);
```

```
                        tmp_array[agent->n_id].parent=agent->parent;

                        if(agent->parent==-1){
                                root=agent->n_id;
                                pb=tmp_array[agent->n_id].affinity-agent_neutral;
                        }
                        else{
                                pb=tmp_array[agent->n_id].affinity-
agent_neutral+2*tmp_array[agent->n_id].load_parent;
                        }

                        add(tmp_subtree,agent->n_id,agent->parent,agent-
>parent,pb,pb,agent->n_id,0,0,0,tmp_array[agent->n_id].affinity,tmp_array[agent-
>n_id].agent_neutral);
                        agent_r[agent->n_id].pb=pb;

                        if(agent->parent!=-1){
                        add(tmp_subtree,agent->n_id,agent->parent,agent-
>parent,pb,pb,agent->parent,0,0,0,tmp_array[agent-
>n_id].affinity,tmp_array[agent->n_id].agent_neutral);//add agent to parent's list
                }

                    agent=agent->next;
                    count++;
                    }
          tmp_a=tmp_subtree->array[root].head;
          tmp_a->root=1;//check the first root of tmp_subtree


          struct agent_review l_array[count];
          if(count==1){
                    check_single_agent(tmp_subtree,subtree,i);

          }
          if(count>1){
       prune_subtree(tmp_subtree,b);
             find_total_benefit(tmp_subtree,b,&total_benefit);

          if(total_benefit>f_total_benefit){
                    f_total_benefit=total_benefit;
                    migration_check(tmp_subtree,b,subtree,size,i);
          }
          //prune subtree
          //rotate

          while(rooting(tmp_subtree,b,root)==1){
          resetting(tmp_subtree,b);
```

```
            rotate(tmp_subtree,b);
            find_new_pb(tmp_subtree,review_array,x,b);
            prune_subtree(tmp_subtree,b);
            find_total_benefit(tmp_subtree,b,&total_benefit);
            if(total_benefit>f_total_benefit){
                    f_total_benefit=total_benefit;
                    migration_check(tmp_subtree,b,subtree,size,i);
            }
}
   print_tmp_subtree(tmp_subtree,b);
}
            }

            }




}

void print_Pl_list(struct Placement_List *f_network,int N){
            nodeA *current;
            int i;
            FILE *fp;



            for(i=0;i<N;i++){
            current=f_network->array[i].head;
            while(current!=NULL){
                    current=current->next;
            }
            printf("\n");
   }
}

void print_network_tree(struct Graph *net){
            netNode* current_n;
            int i;

            for(i=0;i<N;i++){
                    current_n=net->array[i].head;
                    while(current_n!=NULL){
                            printf("%d-%d->",current_n->id,current_n->relationship);
                            current_n=current_n->next;
                    }
```

```
                printf("\n");
        }
}

void print_app_tree(struct App_tree *app,int x){
        Agent* current_a;
        int i;


        for(i=0;i<x;i++){
                current_a=app->array[i].head;
                while(current_a!=NULL){
                        printf("%d-%d->",current_a->agent,current_a->load);
                        current_a=current_a->next;
                }
                printf("\n");
        }

}


void print_subtree(struct Subtree *subtree,int b){
        struct s_node *agent;
        int i;

        for(i=0;i<b;i++){
                agent=subtree->array[i].head;
                if(agent!=NULL){
                while(agent!=NULL){
                        printf("%d,%d,%d->",agent->n_id,agent->check,agent->dest);
                        agent=agent->next;
                }
                printf("\n");
        }
}

}

int parent[N];
bool visited[N];

void initialize_visited(bool visited[],int N){
        int i;
        for(i=0;i<N;i++){
                visited[i]=false;
        }
}
```

```
void getParents(struct Graph *net){
        netNode* current_n;
        int i;

        for(i=0;i<N;i++){
                current_n=net->array[i].head;
                current_n=current_n->next;
                while(current_n!=NULL){
                        if(current_n->relationship==1){
                                parent[i]=current_n->id;
                        }
                        current_n=current_n->next;
                }
        }
}

int LCA(int u,int v){
        int lca;

        while(1){
                visited[u]=true;
                if(parent[u]==-1){
                        break;
                }
                u=parent[u];
        }
        while(1){
      if(visited[v]){ /*Intersection of paths found at this node.*/
         lca = v;
         break ;
      }
      v = parent[v] ;
   }
   return lca ;
}
bool visited1[N];
int getrootDistance(int v){
        int dist=0;
        while(1){

                if(parent[v]==-1){
                        return dist;
                }
                else
                {
                        v=parent[v];
```

```
                        dist++;
                }
        }
}

//finds the distance between two nodes
int getDistance(int n1,int n2){
        int lca,dist1,dist2,lca_dist,dist;

  lca=LCA(n1,n2);
  dist1=getrootDistance(n1);
  dist2=getrootDistance(n2);
  lca_dist=getrootDistance(lca);

  dist=dist1+dist2-2*lca_dist;

  return dist;
}

//finds the parent of an agent
int getParent(struct App_tree *app,int agent_id,int b){
        Agent* current_a;//points to application tree
        int i,pos_p;

        if(agent_id==0){
                pos_p=-1;
        }
        else{
        for(i=0;i<b;i++){
                current_a=app->array[i].head;
                current_a=current_a->next;
                while(current_a!=NULL){
                        if(current_a->agent==agent_id){
                                pos_p=i;
                        }
                        current_a=current_a->next;
                }
        }
   }
        return pos_p;
}

//calculates the load that an agent exchanges with parent and children
int getLoad(struct App_tree *app,struct Placement_List *f_network,int id,int n_id,int
b){//id:current node
        Agent* current_a;//points to application tree  //n_id:node's position
        nodeA *current;//points to final network (network with -agents)
```

```
	int i,pos,hops,load=0,pos_p,parent,agent_load;

	//load from children
	current_a=app->array[id].head;
	if(current_a->next!=NULL){//if current agent has children
		current_a=current_a->next;
	while(current_a!=NULL){
		for(i=0;i<N;i++){
			current=f_network->array[i].head;
			if(current->next!=NULL){
				current=current->next;

			while(current!=NULL){
				if(current->id==current_a->agent){
					pos=i;
				}
				current=current->next;
			}
		}
		}
			initialize_visited(visited,N);
			hops=getDistance(pos,n_id);
			load=load+hops*current_a->load;
		current_a=current_a->next;
	}
}

 //load from parent
 parent=getParent(app,id,b);
if(parent==-1){
	agent_load=0;
}
 else{
	for(i=0;i<N;i++){
			current=f_network->array[i].head;
			if(current->next!=NULL){
				current=current->next;
			while(current!=NULL){
				if(current->id==parent){
					pos_p=i;
				}
				current=current->next;
			}
		}
		}
```

```
                current_a=app->array[parent].head;
                while(current_a!=NULL){
                        if(current_a->agent==id){
                                agent_load=current_a->load;
                        }
                        current_a=current_a->next;
                }
                        initialize_visited(visited,N);
                        hops=getDistance(pos_p,n_id);
                        load=load+hops*agent_load;
        }
        return load;
}

//calculates total network load
int getTotalLoad(struct Placement_List *f_network,struct App_tree *app,int b){
        nodeA *current;//points to final network (network with -agents)
        int i,total_load=0;

        for(i=0;i<N;i++){
                current=f_network->array[i].head;

                if(current->next!=NULL){
                        current=current->next;
                        while(current!=NULL){
                        total_load=total_load+getLoad(app,f_network,current->id,i,b);
                        current=current->next;
                }
                }

        }
        return total_load;
}


int check_network(struct Graph *net,int N){
        netNode* current_n;
        int i;

        for(i=0;i<N;i++){
                current_n=net->array[i].head;
                while(current_n!=NULL){
                        if(current_n->id==-1){
                                return 1;
                        }
                        current_n=current_n->next;
                }
```

```
        }
        return 0;
}

//GRAL's algorithm implementation
void GRAL(int a,int b, int *final_total_load,int *former_total_load,int *migrations){

        struct Graph *net;//network
   struct App_tree *app;//application
   struct Placement_List *f_network;
   struct Subtree *subtree;



   Agent* current_a;//points to application tree
   netNode* current_n;//points to network tree
   nodeA *current;//points to final network (network with -agents)


   int count=-1,counter=0;


        net=net_gen(N,m,l);//generation of network
        app=app_gen(a,b);//generation of application
        f_network=assignments_gen(net,app,a,b,N);

        struct array_node **review_array;
        struct agent_node temp_app[a+b];
        struct neutral_node temp_neutral[b];//array of node-neutral agents


   if(check_network(net,N)==0){
    reps++;
        int n_id,l=0,k;

        getParents(net);
  for(int i=0;i<N;i++){
          if(i==parent[i]){
                  parent[i]=-1;
        }
   }
        print_Pl_list(f_network,N);
        print_app_tree(app,a+b);
        int f_total_load;
        f_total_load=getTotalLoad(f_network,app,b);
        int migrations=0;
   while(count!=0){
```

```
        counter++;
        count=0;
   for(k=0;k<N;k++){
        //agents' positions in f_network
   temp_arrays(temp_app, a+b, temp_neutral, b, app, net, f_network);

        int x=a+b,i,j;

        review_array =(array_node**) malloc(x* sizeof(array_node*));
        for (i=0; i<x; i++)
   {
     review_array[i] =(array_node*) malloc(b * sizeof (array_node));
   }


   position_review(review_array,x,b,a,app,temp_app, a+b, temp_neutral, b,net);

   n_id=k;

   struct s_node *agent;
   subtree=subtree_generation(b);

//BFS to track subtrees in selected node
    Q q;
 int w,pos=-1,check,s,p,size=0;
 q.R=q.F=-1;

 bool visited[x];
 for(i=0;i<x;i++){
        visited[i]=false;
 }
 s=app->array[0].head->agent;//the root agent
 for(i=0;i<b;i++){//b:the number of node-specific agents
        if(temp_neutral[i].id==s && temp_neutral[i].pos==n_id){//if this node-neutral
agent exists in the node place it into subtree
                        pos++;
                        size++;
                        add_agent(subtree,s,-1,-1,pos,-1);
        }
 }
visited[s]=true;
 enqueue(&q,s);
while(!empty(&q))//while queue is not empty
 {
  s=dequeue(&q);

  //insert all unvisited,adjacent vertices of v into queue
```

```
 current_a=app->array[s].head;//points to application tree
 current_a=current_a->next;
 while(current_a!=NULL)
 {
          check=0;
  w=current_a->agent;
  for(i=0;i<b;i++){
         if(temp_neutral[i].id==w && temp_neutral[i].pos==n_id && s!=w){
                  for(j=0;j<b;j++){
                           agent=subtree->array[j].head;//points to subtree
                           while(agent!=NULL){
                                    if(agent->n_id==s){
                                             add_agent(subtree,w,s,0,j,-1);//1:agent is a leaf
                                             if(agent->check!=-1){
                                                      agent->check=0;
                                             }
                                             check=1;
                                    }
                                    agent=agent->next;
                           }
                  }
                  if(check==0){
                           pos++;
                           size++;
                           add_agent(subtree,w,-1,-1,pos,-1);

                  }

         }
   }
   if(!visited[w]){
         visited[w] = true;
         enqueue(&q,w);
   }
   current_a=current_a->next;
  }
 }

printf("%d subtrees detected\n",size);


  //set the destination for every subtree
destination_review(subtree,size,review_array,x,b,net,temp_neutral,b,n_id);


subtree_identification(subtree,size,review_array,x,b,n_id);
```

```
if(size>0){
print_subtree(subtree,b);
}
for(i=0;i<b;i++){
        agent=subtree->array[i].head;
        if(agent!=NULL && agent->dest!=-1){
                while(agent!=NULL){
                        if(agent->check!=0 && agent->dest!=n_id){
                                add_agent(f_network,agent->n_id,agent->dest);
                                delete_agent(f_network,agent->n_id,n_id);
                                printf("o agent that migrated is :%d\n new position
is:%d\n",agent->n_id,agent->dest);
                                count++;
                                migrations++;
                        }
                        agent=agent->next;
                }
        }
}
}
}


getParents(net);
for(int i=0;i<N;i++){
        if(i==parent[i]){
                parent[i]=-1;
        }
}
int total_load;
total_load=getTotalLoad(f_network,app,b);

printf("initial total load is:%d\n",f_total_load);
printf("final total load is:%d\n",total_load);
printf("migrations are:%d\n",migrations);
}
return reps;
}
```

## Main_simulation.c

```
#include<stdio.h>
#include<stdlib.h>
```

```
#include <math.h>
#include "Gral_implementation.h"

int main(){

        int a,b,i,reps,sum_reps=0;
        int final_load, former_load, migrations;
        int sum_final_load=0,sum_former_load=0,sum_migrations=0;
        double avg_final_load,avg_former_load;
        double avg_reduction,avg_migrations;

        printf("Give number of node-specific agents:");
        scanf("%d", &a);
        printf("Give number of node-neutral agents:");
        scanf("%d", &b);

  srand(time(NULL));
  while(sum_reps<100){
        final_load=0;
                former_load=0;
                migrations=0;
     reps=GRAL(a,b,& final_load,& former_load,& migrations);
        if(former_load!=0 && former_load!=final_load){
        sum_reps=sum_reps+reps;
        sum_final_load=sum_final_load+final_load;
        printf("sum load:%d\n",sum_final_load);
        sum_former_load=sum_former_load+former_load;
        printf("migrations:%d\n",migrations);
        sum_migrations=sum_migrations+migrations;
        }
  }
  avg_final_load=double(sum_final_load)/double(sum_reps);
  avg_former_load=double(sum_former_load)/double(sum_reps);
```

```
   avg_reduction=double(((avg_former_load-
avg_final_load)/avg_former_load)*100);
   avg_reduction=round(avg_reduction);
   avg_migrations=double(sum_migrations)/double(sum_reps);

         printf("avg former load:%f\n",avg_former_load);
         printf("avg final load:%f\n",avg_final_load);
         printf("avg_reduction:%f\n",avg_reduction);
         printf("avg migrations:%f",avg_migrations);

}
```