Міністерство освіти і науки України
Національний університет “Львівська політехніка”

На правах рукопису

Муляревич Олександр Володимирович

УДК 004.89:004.942

Розв’язання динамічної задачі комівояжера з використанням поведінкової моделі колонії мурах в багатоагентних системах

05.13.05 - комп’ютерні системи та компоненти

Дисертація на здобуття наукового ступеня
кандидата технічних наук

Науковий керівник –
кандидат технічних наук,
доцент **Голембо В.А.**

Львів – 2016

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення, термін, позначення	Пояснення
GPS	Global Positioning System (Система глобального позиціонування)
LKH	Lin-Kernighan Heuristic (метод)
ГП	Генетичне програмування
ЕОМ	Електронна обчислювальна машина
ЕП	Еволюційне програмування
ЗК	Задача комівояжера
КМ	Комунікаційна мережа
МЗП	Метод зграї птахів
МРБ	Метод рою бджіл

ВСТУП

Актуальність теми. Під час опису транспортних чи комунікаційних процесів за допомогою теорії графів, проблема оптимізації витрат зводиться до розв’язання задачі однократного обходження точок (пунктів, міст, вузлів) графу (циклу Гамільтона) найкоротшим шляхом, яка називається задачею комівояжера (ЗК). Оскільки за найменших витрат ресурсів (часу, енергії та інших) для досягнення поставленого завдання досягаються найбільші продуктивність та відповідно прибуток, тому розв’язок ЗК за прийнятний час та точність (наближеність до оптимального розв’язку) є предметом багатьох наукових досліджень. Одним з найбільш перспективних напрямів досліджень є перехід до застосування децентралізованих колективів автономних агентів з метою автоматизації вимірювальних, обчислювальних та технологічних процесів. Розв’язання динамічної (вхідні дані змінюються в процесі розв’язання задачі: кількості точок, вартості та доступності з’єднань) ЗК, в тому числі в умовах частково невідомих вхідних даних є важливим завданням для взаємодії агентів в колективі мобільних вимірювально-обчислювальних апаратів, для проектування робототехнічних систем військового, космічного та промислового призначення, для маршрутизації в комунікаційних мережах (КМ), для моделювання складських процесів. Більшість існуючих методів розв’язання ЗК не здатні розв’язувати динамічну ЗК, а розв’язання динамічної асиметричної ЗК в умовах частково невідомих даних практично не було досліджено. Тому **актуальним науковим завданням** є підвищення ефективності розв’язання динамічної ЗК шляхом вдосконалення існуючих та розробки нових методів, моделей та засобів.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота виконувалася відповідно до напрямку наукової діяльності кафедри електронних обчислювальних машин Національного університету “Львівська політехніка”: “Питання теорії, проектування та реалізації комп’ютерних систем та мереж, а також комп’ютерних засобів, вузлів, приладів і пристроїв вимірювальних, інформаційних, керуючих, телекомунікаційних та кіберфізичних систем” та науково-дослідної роботи “Інтеграція методів і засобів

вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем" (номер державної реєстрації 0115U000446).

Мета і задачі дослідження. Метою дисертаційної роботи є підвищення ефективності розв'язання динамічної ЗК шляхом вдосконалення існуючих та розробки нових методів, моделей та засобів, що базуються на використанні поведінкової моделі колонії мурах в багатоагентних системах.

Для досягнення поставленої мети необхідно розв'язати наступні **задачі:**

1. аналіз існуючих та пошук нових методів, моделей та засобів для розв'язання динамічної задачі комівояжера, в тому числі динамічної асиметричної задачі комівояжера в умовах частково невідомих даних;

2. дослідження можливості використання поведінкової моделі колонії мурах для розв'язання динамічної ЗК;

3. розроблення моделі багатоагентної системи з використанням технологій паралельних обчислень та вдосконалення існуючих методів для розв'язання динамічної ЗК з кількістю пунктів обходу 100 і більше;

4. розроблення нових засобів та моделі багатоагентної системи з метою реалізації можливості розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних;

5. оцінювання ефективності запропонованих методів, засобів та розроблених моделей багатоагентних систем для розв'язання динамічної ЗК, в тому числі динамічної асиметричної ЗК в умовах частково невідомих даних.

Об'єктом дослідження є процес розв'язання узагальненої багатокритеріальної динамічної ЗК, в тому числі в умовах частково невідомих вхідних даних.

Предметом дослідження є методи, засоби та моделі для розв'язання узагальненої багатокритеріальної динамічної ЗК, в тому числі в умовах частково невідомих вхідних даних.

Методи дослідження. Під час вирішення наукових задач використовувались основні положення дискретної математики, теорії імовірності, теорії графів та математичного моделювання. Методи досліджень

базуються на принципах системного аналізу. Використання методів теорії паралельних обчислень, теорії багатоагентних систем дозволило розробити ефективні засоби для розв'язання динамічної ЗК.

Наукова новизна одержаних результатів полягає в тому, що:

- вперше розроблено та апробовано модель багатоагентної системи, яка базується на використанні поведінкової моделі колонії мурах при розміщенні цифрових міток на комунікаційних вузлах, що дозволило розв'язати динамічну асиметричну задачу комівояжера в умовах частково невідомих вхідних даних;

- вперше розроблено метод і отримано експериментальні результати подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток (пам'яті колонії мурах), який базується на використанні адаптивної верхньої межі значення цифрової мітки, що дозволило розробленій багатоагентній системі відновити пошук маршрутів меншої вартості навіть після тривалого статичного стану вхідних даних;

- вперше розроблено метод опрацювання результуючого маршруту при розв'язанні динамічної задачі комівояжера, який базується на використанні алгоритмів локальної оптимізації 2-opt, 2.5-opt, 3-opt в залежності від інтенсивності динамічних змін вхідних даних, що дозволило зменшити вартість результуючих маршрутів;

- удосконалено метод розв'язання задачі комівояжера, який базується на використанні поведінкової моделі колонії мурах, шляхом зміни початкової установки значень міток та імовірнісного вибору наступного вузла для переходу мурахи-агента, що дозволило зменшити кількість ітерацій циклу пошуку маршрутів агентами та відповідно час розв'язання задачі комівояжера.

Достовірність та обгрунтованість викладених в дисертації наукових положень, висновків та рекомендацій забезпечується коректністю постановки розглянутих задач, використанням методів дискретної математики, математичного моделювання, теорії графів, системного аналізу, теорії паралельних обчислень, теорії імовірності та теорії багатоагентних систем, збіжністю результатів, отриманих аналітичними методами з результатами

комп’ютерного моделювання й експериментальних досліджень.

Практичне значення одержаних результатів досліджень полягає у використанні для розробки:

- класифікації існуючих методів розв’язання динамічної ЗК, аналіз якої показав, що для великої кількості пунктів (N >100) потрібно орієнтуватись на методи розв’язання із застосуванням колективної поведінки агентів, серед яких найбільш перспективний – метод колонії мурах;

- додаткового програмного модуля на базі методів локальної оптимізації, застосування якого дозволило збільшити точність отримуваних результатів, зберігаючи можливість розв’язання динамічної ЗК;

- багатоагентної системи з використанням поведінкової моделі колонії мурах та технологій паралельних обчислень для розв’язання динамічної ЗК з кількістю вузлів до 65536;

- багатоагентної системи для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних, наближених до умов в реальній комунікаційній мережі робототехнічних систем космічного та військового призначення.

Основні результати дисертаційної роботи впроваджено:

- в науково-дослідну роботу «Інтеграція методів і засобів вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем» (шифр ДБ/КІБЕР, реєстраційний номер 0115U000446) Національного університету «Львівська політехніка» (Додаток А);
- при розробці програмного забезпечення та оптимізаційних рішень в міжнародній компанії “Logivations GmbH” (Гребенцель, Німеччина) (Додаток Б);
- при розробці обчислювальних засобів в міжнародній компанії “Eleks” (Львів, Україна) (Додаток В);
- в навчальному процесі на кафедрі електронних обчислювальних машин Національного університету «Львівська політехніка» (Додаток Г).

Особистий внесок здобувача. Усі основні наукові та практичні

результати дисертаційної роботи автор отримав самостійно. У роботах, опублікованих у співавторстві, здобувачу належать: розроблена нова модифікація алгоритму колонії мурах [8,9]; систематизовано та розроблено класифікацію існуючих методів розв’язання ЗК [24]; розроблена багатоагентна система для розв’язання динамічної ЗК з використанням додаткового програмного модуля на базі методів локальної оптимізації [26,27]; розроблені засоби та модель багатоагентної системи для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних [25,90,91].

Апробація результатів дисертації. Основні положення та результати дисертаційної роботи було апробовано у доповідях на 3 науково-технічних конференціях, а саме: 1) 5-а Міжнародна науково-технічна конференція ACSN-2011 "Сучасні комп'ютерні системи та мережі: розробка та використання", м.Львів, 29 вересня – 01 жовтня 2011 р.; 2) IX Міжнародна науково-технічна конференція «Методи і засоби вимірювань фізичних величин» - «Температура-2012», м.Львів, 25 – 28 вересня 2012 р.; 3) XIII Міжнародна науково-технічна конференція TCSET’2016 «Сучасні проблеми радіоелектроніки, телекомунікацій, комп'ютерної інженерії», Львів – Славське, 23 – 26 лютого 2016.

Публікації. За результатами виконаних наукових досліджень опубліковано 8 наукових праць, з них 4 статей у наукових журналах, що входять до переліку періодичних фахових видань, 1 стаття у закордонному науковому журналі, що входить до наступних наукометричних баз даних: Google Scholar, ERIH, Index Copernicus, Ulrichsweb Global Serials Directory, DOAJ, EBSCOhost, GetInfo, Research Bible; 3 тези доповідей у збірниках міжнародних науково-технічних конференцій.

Структура та обсяг роботи. Дисертаційна робота складається із вступу, чотирьох розділів, що містять 53 рисунки і 9 таблиць, висновків, списку використаних джерел (112 найменувань) і 10 додатків. Загальний обсяг дисертації складає 227 сторінок, з яких основний зміст викладено на 129 сторінках.

1. АНАЛІЗ МЕТОДІВ РОЗВ'ЯЗАННЯ ЗАДАЧІ КОМІВОЯЖЕРА (ЗК)

1.1. Формулювання ЗК. Аналіз існуючих методів, моделей та засобів

Серед завдань теорії комбінаторики видатне місце займає задача комівояжера (ЗК). ЗК була вперше сформульована в 1934 році та служить своєрідним полігоном, на якому математики випробовують нові методи [35,39].

Формулювання класичної задачі комівояжера: комівояжер (торговець - розповсюджувач) має вийти з певного початкового міста, відвідати одноразово в невідомому порядку N міст і повернутися в початкове місто. Відстані між містами відомі. Метою задачі є знаходження порядку відвідування міст, який необхідно обрати, щоб замкнутий шлях (тур, маршрут) комівояжера був найкоротшим. В рамках КМ, в тому числі комп'ютерній, формулювання класичної ЗК набуде наступного вигляду: пошук результуючого маршруту найменшої вартості (це може бути відстань, час, навантаження, і т.і.), який включає одноразове відвідування N вузлів КМ комівояжером та повернення в початковий вузел. ЗК відноситься до задач комбінаторної оптимізації NP [102] складності. Розв'язання різних типів ЗК є важливим завданням в дослідженні математичних операцій та теорії комп'ютерних наук.

Аналіз завдання, перелік початкових даних: позначимо вузли мережі [99] як P(i), де i – індекс вузла в послідовності вузлів [0,N]. Маршрут комівояжера розпочинається з визначеного початкового вузла P(0). Маршрут комівояжера може бути описаний циклічною перестановкою R=(P(0),…,P(i), P(i+1), …,P(0)); індекси вузлів в послідовності повторюються тільки на початку й у кінці маршруту, оскільки маршрут комівояжера розпочинається і закінчується в визначеному початковому вузлі – P(0), тобто перестановка зациклена. Вартість переміщення використовуючи з'єднання P(i) → P(j), тобто з вузла P(i) до вузла P(j), позначимо як Cij. В сукупності вартості переміщень між вузлами КМ утворюють матрицю C (Costs), розмірністю N x N. Задача полягає в тому, щоб знайти такий маршрут R, який мав би найменшу сумарну вартість (відстань, час і т.д.) [43].

Також слід зазначити математичні умови, які встановлюються згідно

формулюваня ЗК:

1) Згідно формулювання відстань між будь-якими вузлами i та j – Cij має бути невід'ємною, а переміщення в той самий вузел забороненим, тобто для усіх $i,j \in N$, $i \neq j$:

$$Cij \geq 0; Cjj = \infty; \quad (1.1)$$

2) Для класичного формулювання ЗК властива умова заборони петлі в маршруті, а також вартості протилежно направлених з'єднань P(i) → P(j) та P(j) → P(i) мають бути однаковими – умова симетричності ЗК, тобто для усіх i,j виконується:

$$Cij = Cji \quad (1.2)$$

Отже ЗК бувають симетричними та асиметричними. Умова асиметричності ЗК відповідно формулюється наступним чином:

$$Cij \neq Cji \quad (1.3)$$

3) Виконання нерівності трикутника, тобто для усіх значень відстаней між вузлами виконується наступна рівність:

$$Cij + Cjk \geq Cik \quad (1.4)$$

З метою розв'язання ЗК в умовах КМ введемо поняття доступності для використання з'єднання P(i) → P(j) та позначимо її як Aij. Змінна Aij набуває значення 0 (з'єднання недоступне) та 1 (з'єднання доступне). В сукупності доступності з'єднань між вузлами мережі складають матрицю доступностей A (Accessability). Більш детальний огляд ЗК представлено у Додатку Д.

Якщо протягом розв'язання задачі значення N, Cij та Aij є незмінними, то ЗК є статичного характеру – статична ЗК. Якщо в процесі розв'язання задачі виникають динамічні зміни вхідних даних: з'єднання між вузлами стає недоступим або вартість проходження по ньому міняється, додається новий або відключається існуючий вузел – N, Cij та Aij можуть мінятись, тоді ЗК є динамічного характеру – **динамічна ЗК** [4,11]. В ході даної роботи розглядається розв'язання статичної ЗК (як симетричної так і асиметричної), розв'язання ЗК в умовах виникнення динамічних змін та розв'язання асиметричної динамічної ЗК з частково невідомими вхідними даними,

максимально наближеної до умов в реальній КМ. В рамках реальної КМ комівояжер володіє лише даними про доступності з'єднань та вартості переміщення по з'єднанням тільки відносно поточного вузла. Тобто в кожен момент обчислення ЗК – знаходження машруту обходження вузлів, комівояжер отримує лише наступну інформацію: [Aim] та [Cim], де i – індекс поточного вузла, m – список індексів вузлів, під'єднаних до поточного вузла, в якому знаходиться комівояжер. Також припускається в рамках КМ можливість порушення нерівності трикуника (рівняння 1.4), крім того в випадках відсутності ейлерового циклу [58] припускається порушення умови одноразового відвідування кожного вузла, тобто ЗК зводиться до розв'язання задачі обходження, аналогічної до розсилання широкомовного повідомлення в комп'ютерній мережі.

Вхідними даними є множина вузлів P, кількістю N, також задані матриці C та A. Необхідно знайти маршрут R, що проходить по одному разу через кожен вузел та повертається в початковий, довжина якого Lmin є мінімальною серед усіх можливих інших маршрутів Lk – найоптимальніший маршрут – точне рішення ЗК. В процесі розв'язання ЗК може бути отримано квазі-оптимальний маршрут R' – евристичний розв'язок наближений до оптимального. Для оцінки точності розв'язку задачі комівояжера в класичній теорії застосовується межа Хелда-Карпа [3], що ґрунтується на методах релаксації. Межа Хелда-Карпа є стандартом оцінювання точності отриманого маршруту (визначення похибки – різниці вартості отриманого від теоретично можливого оптимального, мінімального розв'язку). Для тестування існують відкриті бази даних ЗК [64], що вже є розв'язаними, з наявними даними по вартості найоптимальнішого шляху, що знаходяться у відкритому доступі в Інтернеті [107, 108].

З метою визначення переваг та недоліків застосування існуючих методів для розв'язання ЗК, було проведено дослідження, на базі якого розроблено класифікацію методів розв'язання ЗК, представлену на рис.1.1..

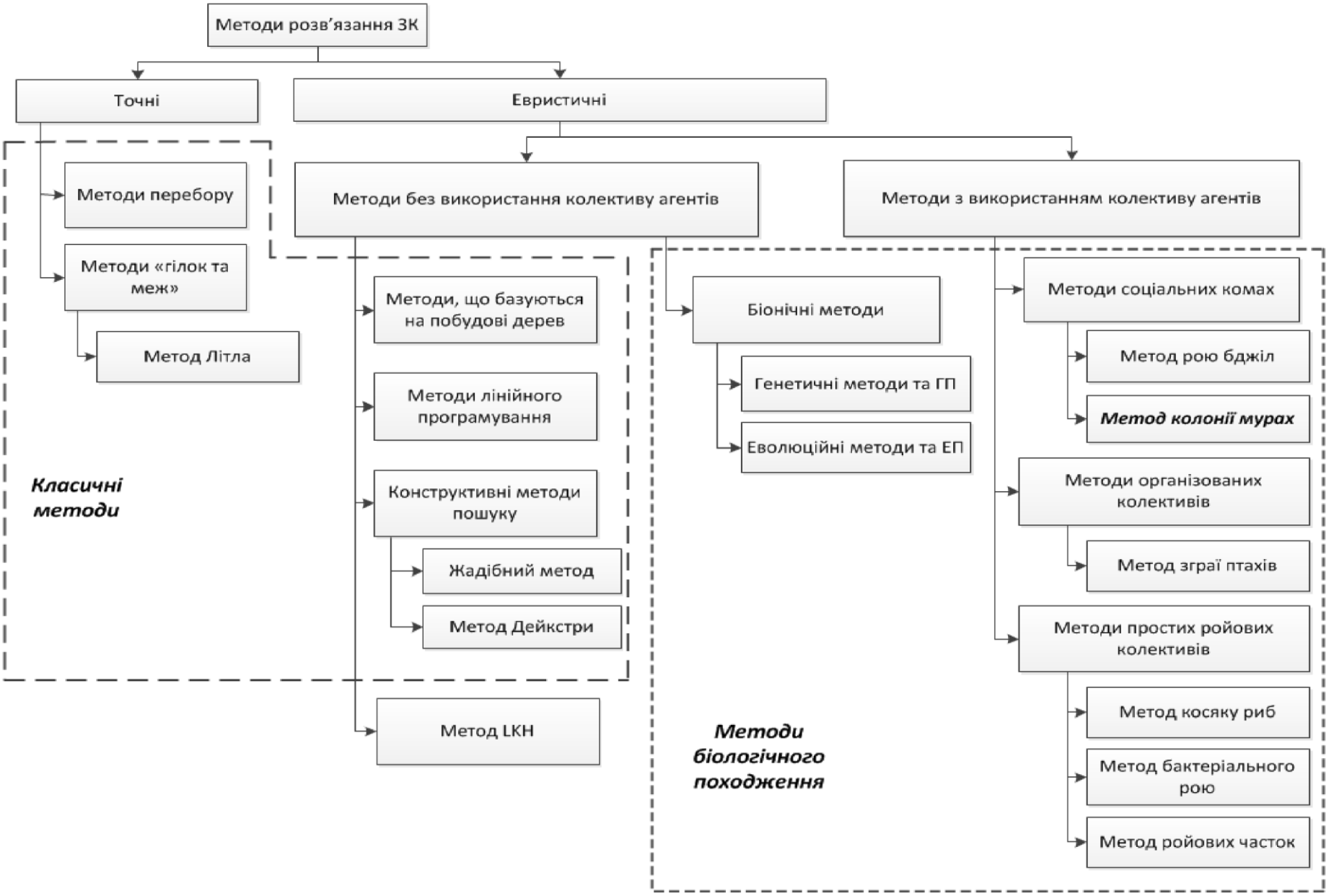


Рис.1.1. Запропонована класифікація методів розв'язання ЗК

Традиційно методи [1] для розв’язання ЗК поділяють на:

1) Точні методи (потребують значних часових витрат і підходять лише для задач відносно малих розмірностей, до 100 пунктів, з використанням розпаралелення до 1000);

2) Евристичні методи, які забезпечують знаходження маршруту, наближеного до оптимального, квазі-оптимального розв’язку. До еврестичних методів відносяться методи розв’язку без використання колективу агентів (при застосуванні одного агента) та методи з використанням колективу агентів [24].

Крім розподілу за точністю, було введено два умовних групування: «класичні» методи та методи біологічного походження. До загальновідомих так званих «класичних» методів розв’язання ЗК відносяться: 1) метод перебору та метод гілок і меж; 2) методи лінійного програмування [46,47], конструктивні методи пошуку та методи що базуються на побудові дерев («дерев’яні» методи). Конструктивні методи пошуку, або методи конструктивного характеру – методи, в яких поступово формується кінцевий результат – квазі-оптимальний шлях, згідно проходження по ньому. До конструктивних методів відносяться жадібний метод, метод «найближчого сусіда», метод Дейкстри та інші. До «дерев’яних» методів належить клас методів, що використовують структуру типу дерева для знаходження результуючого шляху. Спочатку формується граф типу дерева, після чого в ньому шукається оптимальний шлях.

Методи біологічного походження відносяться до сучасних методів розв’язання ЗК, в цю групу входять біонічні методи (евристичні, без використання колективу агентів) та методи з використанням колективу агентів. Біонічні методи поділяються на дві групи: 1) еволюційні методи та еволюційне програмування (ЕП) [7,33,69,70,111] 2) генетичні методи та генетичне програмування (ГП) [34,94]. Біонічні методи можуть бути реалізовані також з використанням колективу агентів, проте це є необов’язковим [73,74,75]. До методів з використанням колективу агентів належать: 1) методи соціальних комах: метод рою бджіл (МРБ, Artificial Bee Colony) [30,77], метод колонії мурах (МКМ, Ant Colony Optimization) [14,22,32]; 2) методи організованих

колективів: метод зграї птахів (МЗП, Migrating Birds Optimization) [106]; 3) методи простих ройових колективів: метод ройових часток [81,89] (Particle Swarm Optimization), метод бактеріального рою [48,51] (Bacterial Foraging Optimization), метод косяку риб [56] (Artificial Fish Swarm algorithm) та інші [18,28,41,56].

Окреме місце займають методи локального пошуку [63], або як їх ще називають – методи локальної оптимізації. Для обчислення результату їм необхідно на вхід подати згенерований якимось чином або отриманий іншим методом маршрут. Кількість з'єднань між вузлами, які приймають участь в перестановці, визначають складність методу локальної оптимізації. Дану кількість з'єднань (ребер, при представленні результуючого маршруту у вигляді графу) ще називають коефіцієнтом складності методів локальної оптимізації – k. Розрізняють методи низької та високої слкадності згідно кількості з'єднань (рис.1.2), що приймають участь в перестановці, а також метод локальної оптимізації з динамічно змінюваним коефіцієнтом складності k [73], який змінюється в процесі обчислення результату. Найпоширенішими є методи локальної оптимізації низької складності: алгоритми 2-opt та 3-opt, а також проміжна форма *2.5*-opt [3, 54], які позначаються згідно загально прийнятої концепції у формі k-opt.

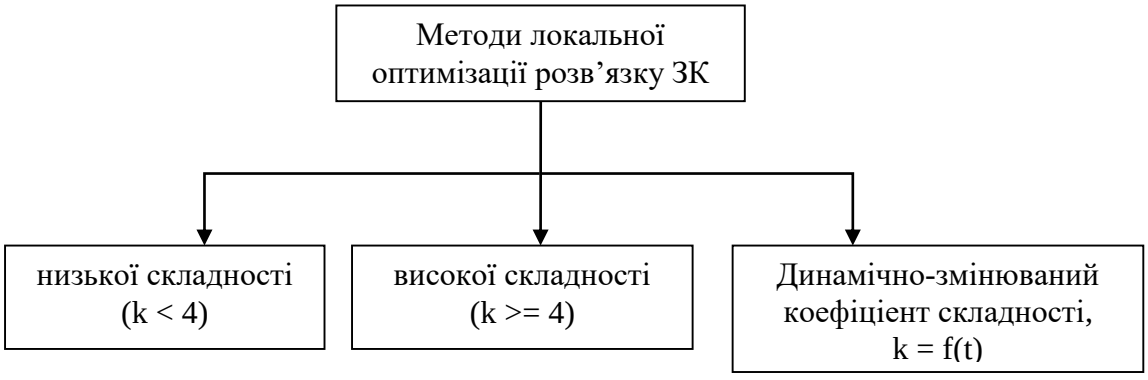


Рис.1.2. Класифікація методів локальної оптимізації розв'язку ЗК

У класичній реалізації [84], при представленні результуючого маршруту у вигляді графу, алгоритм 2-opt вилучає 2 ребра, що входять до маршруту і додає

два нові так, щоб новий маршрут став коротшим. Причому існує лише один варіант заміни двох ребер на нові два, при якому новий маршрут залишиться маршрутом, а не утворяться 2 цикли.

Аналогічно використовуються інші алгоритми локальної оптимізації з різницею лише у тому, що замість двох ребер використовується 3-и ребра для оптимізації у 3-opt алгоритмі, або ж **k** ребер у найбільш гнучкому і ефективному k-opt алгоритмі, що був розроблений як модифікація алгоритмів локальної оптимізації та відомий під назвою - метод Ліна-Кернігана, у якому коефіцієнт складності k динамічна величина, що змінюється в процесі обчислення кінцевого результату.

Огляд класичних методів розв'язання ЗК. Аналіз класичних методів розв'язання ЗК, розглянутих в розробленій класифікації (див. рис.1.1), представлено в Додатку Е. Серед усіх методів слід відзначити метод гілок та меж, що є точним методом, за допомогою якого можливо вирішити ЗК при N = 100 та до N = 1000 з використанням модифікацій [23,42,61].

В таблиці 1.1. наведено порівняльні характеристики методів. На практиці серед розглянутих методів найбільш використовуються метод гілок та меж та «дерев'яний» метод [52,53,82]. Аналіз класичних методів, які наведені в табл. 1.1., показав, що більшість з них використовується при кількості вузлів меншої за 100.

Алгоритми складають основу функціонування систем та суттєво впливають на їх архітектуру. Саме тому вони є основними об'єктами вивчення, аналізу та вдосконалення. Оптимізація алгоритму дозволяє значно зменшити ресурси, що витрачаються на розв'язання завдання, а також при цьому зазвичай спрощується апаратна та програмна реалізація [43]. Дослідження алгоритмів потребує сукупності декількох підходів: творчого – для утворення ідеї рішення задачі; логічного – для аналізу правильності рішення; математичного – для аналізу продуктивності та ефективності алгоритму порівняно з існуючими.

Алгоритми, які базуються на «дерев'яному» методі розв'язання ЗК, мають меншу обчислювальну складність — O(nlog(n)), порівняно з O(n2) для інших

Таблиця 1.1.

Порівняння класичних методів розв’язання ЗК

Назва методу	Переваги	Недоліки	Кількість вузлів N
Метод повного перебору	1) точний алгоритм, дає однозначну відповідь про існування циклу Гамільтона, та результат є найкоротшим шляхом. 2) простий метод, легко реалізується.	1) надзвичайно багато обчислень, при збільшенні кількості вхідних пунктів. 2) можливий розв’язок лише до 13 вузлів, оскільки час необхідний на обчислення досягає астрономічних значень.	<13
Жадібний метод	1) простий метод, легко реалізується.	результат більший від оптимального за вартістю у декілька разів, знайдений результат часто не прийнятний як оптимальний.	<50
Дерев’яний метод	1) має дещо нижчу складність, а отже і час обчислення результату. 2) отримані результати потім часто використовуються для подальшої локальної оптимізації.	дає результат не більший ніж у 2 рази за вартістю за найоптимальніший, однак такий результат часто є не прийнятним на практиці.	<100
Метод Дейкстри	1) поетапне формування результуючого шляху, тобто є можливість корекції в процесі пошуку. 2) отриманий результат є наближеним до оптимального.	1) необхідність збереження великої кількості даних. 2) обов’язкова умова наявності координат пунктів. 3) багато обчислень, різке збільшення часу обчислення при великих обсягах даних.	<100
Метод гілок і меж	1) ефективний метод, що дає максимально наближені до оптимального результати. 2) менший час необхідний для обчислень порівняно з іншими класичними методами.	1) велика кількість обчислень при кількості початкових даних від 100 вузлів та відповідно стрімке збільшення часу обчислення.	< 100 (<1000 враховуючи модифікації)

алгоритмів розв’язання ЗК [46,47]. Комп’ютерні системи для розв’язання ЗК, що базуються на використанні «дерев’яних» алгоритмів демонструють високу швидкодію, проте результуючий маршрут до 2-х разів більший за вартістю від оптимального при розмірності ЗК більше 100, що є неприйнятним на практиці [52,82].

Обчислювальні засоби та комп’ютерні системи для розв’язання ЗК з використанням алгоритму Дейкстри використовуються лише для задач з геометричним представленням вхідних даних у вигляді координат пунктів, крім того потребують великих часових затрат при обчислення ЗК з розмірністю більше 100 та перезапуску процесу обчислення при виникненні динамічних змін вхідних даних [43].

Комп’ютерні системи для розв’язання ЗК з використанням алгоритму гілок та меж здатні розв’язувати ефективно ЗК при розмірності до 1000 вузлів, проте потребують частковий перерахунок знайдених меж при виникненні динамічних змін вхідних даних [42,61]. Для обчислення ЗК з розмірністю більше 1000, виникає потреба застосування методів декомпозиції задачі, оскільки метод гілок та меж потребує великих часових витрат, неприйнятних на практиці. Застосування методів декомпозиції задачі призводить до потреби перезапуску процесу обчислення при виникненні динамічних змін.

Впродовж останніх років К. Хельсгаун (Helsgaun) удосконалив класичну версію методу Ліна-Кернігана [73], запропонувавши удосконалений метод LKH (Lin-Kernighan Heuristic), який вважається найкращим на даний час евристичним методом розв’язування ЗК за точністю [3]. Метод LKH [43,73] займає окреме місце в розробленій класифікації (див. рис. 1.1).

Даний метод базується на поєднанні методу декомпозиції задачі [1,18], методу гілок та меж, а також k-opt методу. Проте LKH метод має високу обчислювальну складність, що призводить до великих часових затрат на обчислення для задач великих розмірностей [3]. Крім того даний метод не є конструктивним, а отже при виникненні динамічних змін вхідних даних, необхідно виконувати повторне обчислення результату.

Вдосконалення класичних алгоритмів розв’язання ЗК вже не призводить до бажаних результатів, особливо коли мова йде про великий обсяг вхідних даних, або розв’язання динамічної ЗК. Комп’ютерні системи для розв’язання ЗК, що базуються на використанні класичних методів, незалежно від типу необхідних початкових даних та принципу знаходження найкоротшого шляху мають наступні недоліки:

1) Пошук розв’язку виконує один комівояжер (агент), що робить неможливим розподілене обчислення ЗК.

2) Розв’язання лише статичної ЗК, без можливості розв’язання динамічної ЗК.

3) Слабка або неефективна здатність до розпаралелення виконання алгоритму для більшості методів.

4) Спроможність більшості методів знайти розв’язки ЗК наближені до оптимального результату за допустимий на практиці час лише для кількості вузлів $N < 100$.

Розглянуті недоліки комп’ютерних систем для розв’язання ЗК, що базуються на використанні класичних алгоритмів, роблять їх застосування на практиці для розв’язання ЗК великої розмірності N неефективним. Особливо коли мова йде про динамічні ЗК, які є більш поширеними в реальному житті.

Огляд методів біологічного походження. До методів біологічного походження відносяться біонічні методи, що включають в себе генетичні та еволюційні методи, а також ГП та ЕП [69,74,94], які функціонують за принципом відбору кращої «популяції» - класу маршрутів на певній ділянці. Відбувається пошук проблемних ділянок, які не вийшло ефективно оптимізувати, після чого застосовуються алгоритми локальної оптимізації. Дані методи дають непогані результати, проте мають низьку точність при розв’язанні ЗК розмірності більше 1000 [7,12], потребують генерації «початкової популяції» – набору маршрутів за допомогою інших алгоритмів, та проведення налаштування мутації популяції для різних типів даних ЗК, що робить такий клас алгоритмів досить складним у використанні та налаштуванні під кожний

окремий випадок ЗК для досягнення максимальної ефективності та мінімального часу на отримання результату. Для динамічної ЗК процес зведення до кращої популяції потребує значно більше часу.

До методів біологічного походження відносяться методи розв'язання ЗК з використанням колективу агентів, які поділяються на: методи соціальних комах, до яких відносяться метод колонії мурах та метод рою бджіл (МРБ); методи організованих колективів, до яких відноситься МЗП; методи простих ройових колективів, до яких відносяться метод ройових часток, метод бактеріального рою та метод косяку риб.

Методи розв'язання ЗК з використанням колективу агентів застосовуються для проектування децентралізованих комп'ютерних систем, які є більш ефективними при вирішенні складних задач, мають більші показники надійності та живучості в порівнянні з централізованими комп'ютерними системами, забезпечують більш високу гнучкість і плавність (точність) реагування на зовнішні впливи (краще пристосування до динамічних змін вхідних даних) [5,87]. Використання методів організованих колективів та методів простих ройових колективів є поки що недостатньо ефективним при N > 100 та в умовах динамічних змін вхідних даних.

Досить перспективним виявилось застосування поведінки «природних агентів» – соціальних комах, здатних щоденно вирішувати складні задачі, які по суті близькі до задач з комбінаторної оптимізації, в тому числі і ЗК. Такі методи отримали назву «методи соціальних комах», дані методи відносяться до методів ***ройового інтелекту*** (англ. swarm intelligence) [88], або як їх прийнято називати інтелектуальними методами оптимізації [56]. Ройовий інтелект є результатом колективної поведінки агентів децентралізованої (від терміну «рій» залишається лише поняття сім'ї, колективу, без врахування матки – центру такого колективу) самоорганізуючої (здатної самостійно розв'язувати поставлені завдання) системи. Більшість задач комбінаторної оптимізації успішно вирішуються в природі колонією мурах та бджолиним роєм. Єдиний центр відсутній, тобто комахи діють незалежно, самоорганізовано, узгоджено з колективом. Алгоритми

цих самоорганізованих істот перспективно дослідити та реалізувати на практиці.

Метод Рою Бджіл (МРБ) в оригіналі розв'язує задачі пов'язані з пошуком місцевості з найбільшою густиною квітів [76], тобто ділянки з найбільшою концентрацією цільових об'єктів. МРБ базується на існуванні двох орієнтирів: Індивідуальна Найкраща Позиція (ІНП) – ділянка, яку окрема бджола відмічає з власного досвіду пошуку як найкращу, з найбільшим показником необхідних об'єктів; Глобальна Найкраща Позиція (ГНП) – це ділянка, яка прийнята за найкращу усім колективом бджіл – роєм.

Усі бджоли отримують дані про поточний стан ГНП, якщо ІНП є гіршим за знайдену роєм ділянку індивідуум прямує до ГНП, при чому шляхом до неї він ретельно аналізує ділянки, що зустрічає на шляху. Якщо бджола знаходить ділянку, що є кращою за знайдену роєм, вона приймається новою ГНП, до якої будуть спрямовуватись усі інші бджоли. Коли більш оптимальних ділянок знайдено не буде весь рій бджіл знаходитиметься в ГНП.

Проте МРБ не підходить для рішення ЗК, сам механізм рішення завдання комівояжера бджолами та джмелями вчені на даний момент так і не змогли дослідити до кінця. Проте механізм пошуку найбільшого скупчення об'єктів, що ефективно реалізує МРБ [30,77], вже широко застосовується в багатоагентних системах.

ЗК вирішується за допомогою методу відтвореного на базі інших соціальних комах – мурах. Метод колонії мурах [79,112] відноситься до унікальних інтелектуальних багатоагентних методів, що здатні ефективно та досить швидко вирішувати складні задачі маршрутизації [6,19]. На сьогоднішній день цей метод ефективно застосовується в програмній реалізації для розв'язання завдання маршрутизації у телефонних комунікаціях [65], а також при GPS (Global Positioning System) навігації складних транспортних технологічних процесів. Вперше поведінкову модель колонії мурах, аналогічну до природньої, разом з алгоритмом колонії мурах було запропоновано Марко Доріго [56].

1.2. Розв’язання ЗК з використанням поведінкової моделі колонії мурах

Метод колонії мурах [66,101,103,105] здатен розв’язувати ЗК, отримуючи за початкові дані тільки початковий вузел P(0). Матриця вартостей C та матриця доступностей A в повному обсязі є необов’язковими для даного методу, тобто дані можуть бути частково невідомими на початок обчислень або евристичними, що є **неможливою задачею для усіх інших розглянутих методів**. Також в реальному житті будь-яка мережа не є ідеальною, тобто є імовірність виникнення перевантаження певних ділянок мережі, що призводить до збільшення часу необхідного на передачу даних через перевантажені з’єднання, та навпаки – з’єднання з меншим навантаженням між вузлами можна подолати швидче. Згодом ситуація може змінитись, а отже знадобиться повторне розв’язання ЗК з новими даними. Інакше кажучи на практиці показники та дані по мережі динамічно змінюються, з такими змінами легко впорається метод колонії мурах [66,101], здатний за таких умов швидко знайти новий найкоротших шлях.

Опис поведінкової моделі колонії мурах. Різноманітна діяльність мурах має яскраво виражений колективний характер. Працюючи разом, група мурах здатна затягнути в мурашник шматок їжі або будівельного матеріалу, в 10 разів більше за самих працівників. Вчені давно знають про це, але тільки останнім часом замислилися про корисне застосування досвіду мурах в повсякденному житті.

Сам по собі мураха - досить примітивна істота. Усі його дії, по суті, зводяться до елементарних інстинктивних реакцій на навколишнє оточення та інших мурах. **Стігмергія (stigmergy)** [67,85] – механізм опосередкованої непрямої взаємодії між індивідумами, одна з форм самоорганізації складних, децентралізованих, незалежних систем без прямої взаємодії між агентами [37,38,44,45,62], що реалізовується шляхом використання міток. Колектив мурах за рахунок явища стігмергії утворює складну багатоагентну систему [21,60,67,40,80] на базі ройової поведінкової моделі. Колектив мурах здатен

ефективно знаходити найкоротший маршрут до їжі. Якщо яка-небудь перешкода – палиця, камінь, нога людини - встає на використовуваному шляху, мурахи швидко знаходять новий оптимальний маршрут. Мітки в випадку апаратно-програмної реалізації алгоритму є цифрові (хімічні у справжніх мурах). Значення мітки, що відноситься до з'єднання з вузла i до вузла j позначимо як Mij, а сукупність значень міток складає матрицю M (marks). Дану сукупність значень міток ще називають **пам'ять колонії мурах**, яка є акумулятором накопиченого мурахами досвіду в процесі розв'язання ЗК.

Мурахи вирішують проблеми пошуку шляхів за допомогою хімічної регуляції. Кожна мураха залишає за собою на землі доріжку особливих речовин - феромонів. Інший мураха, відчувши слід на землі, відправляється за ним. Чим більше по одному шляху пройшло мурах - тим чіткіше слід, а чим чіткіше слід - тим більше «бажання» піти в ту ж сторону виникає у інших мурах. Оскільки мурахи, що знайшли найкоротший шлях до «об'єкту їжі», витрачають менше часу на шлях туди і назад, їх слід швидко стає найпомітнішим. Він притягує все більшу кількість мурах, і коло замикається. Інші шляхи - менш використовувані - поступово зникають, завдяки механізму випаровування феромону, тобто зменшення його кількості.

Алгоритм колонії мурах. Алгоритм колонії мурах [66] базується на застосуванні декількох агентів і має специфічні властивості, характерні мурахам, які використовуються для орієнтації у фізичному просторі. Алгоритм колонії мурах [97] потребує ряд керуючих вхідних параметрів, таких як:

1) Кількість агентів (мурах) – позначимо як **k**;

2) Доцільність використання допоміжних засобів підсилення знайдених квазі-оптимальних маршрутів шляхом збільшення значення мітки (додаткового «нанесення феромону») з метою пришвидчення розв'язання ЗК.

3) Параметри процесів «накладання феромонів» та «випарювання феромонів».

4) Критерій зупинки обчислень: гранична довжина оптимального маршруту, кількість ітерацій циклу пошуку маршрутів мурахами або час

обчислення.

5) Коефіцієнти, що визначають співвідношення ваги вартостей ребер – L (Length) та ваги феромону (значень міток) – E (Experience);

Дослідження оптимальності цих параметрів для певних мереж ЗК лише починають проводитись. Чітких відповідей немає, ведеться розробка модифікацій алгоритму (Asrank, елітні мурахи, Max-Min, пропорційні псевдовипадкові правила та інші модифікації алгоритму [101,103,110]), проте розв'язання динамічної ЗК було недостатньо досліджено [4,40,59]. Тому необхідно провести аналіз з метою виявлення вхідних оптимальних параметрів.

Алгоритм колонії мурах особливо цікавий тому, що його можна використовувати для вирішення не лише статичних, але і динамічних проблем, наприклад, в мережах [16], структура яких змінюється. Отже розглянемо загальний принцип алгоритму.

Ідея алгоритму колонії мурах полягає у використанні міток – феромонів, що можна побачити на наступному класичному прикладі [103]: дві мурахи з мурашника повинні дістатися до певного об'єкту «їжа», яка знаходиться за перешкодою (рис.1.3). Під час переміщення кожна мураха виділяє порцію феромону, використовуючи його як мітку.

Якщо початкові параметри вірні, тобто ще немає ніяких встановлених пріоритетів, кожна мураха вибере свій шлях. Припустимо перша мураха вибирає верхній шлях, а друга - нижній. Оскільки нижній шлях в два рази коротше за верхній, друга мураха досягне мети за час ***t1***. Перша мураха у цей момент пройде лише половину шляху (рис. 1.4.а). Коли мураха досягає їжі, вона бере один з об'єктів і повертається до мурашника по тому ж шляху. За час ***t2*** друга мураха повернулася в мурашник з їжею, а перша мураха лише досягла їжі (рис. 1.4.б).

При переміщенні кожної мурахи на шляху залишається порція феромону. Для першої мурахи за час ***t2*** шлях був покритий феромоном тільки один раз. У той же самий час друга мураха покрила шлях феромоном двічі. За час ***t4*** перша мураха повернулася в мурашник, а друга мураха вже встигла ще раз сходити до

їжі і повернутися. При цьому концентрація феромону на нижньому шляху буде в два рази вище, ніж на верхньому. Тому перша мураха наступного разу вибере нижній шлях, оскільки там концентрація феромону вища. У цьому полягає базова ідея алгоритму мурах – самоорганізуюча оптимізація шляхом непрямого зв'язку між автономними агентами.

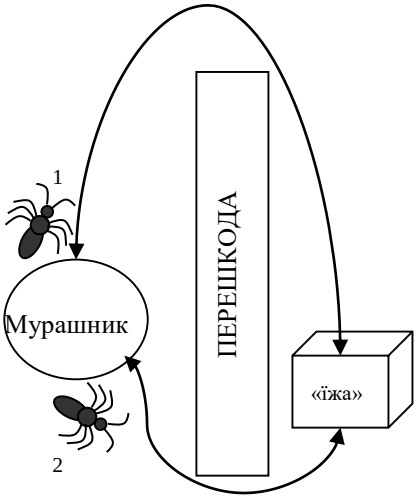
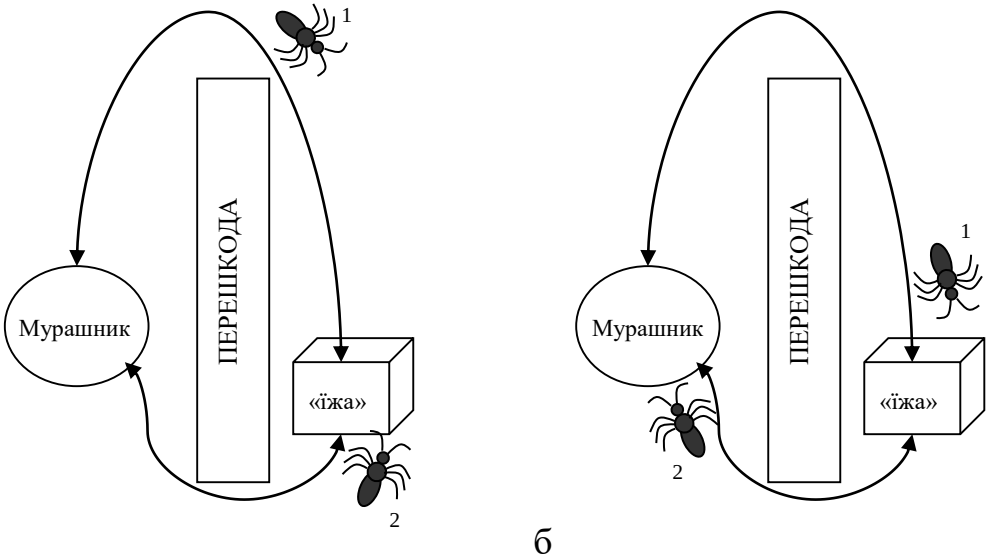


Рис.1.3. Класична схема пошуку їжі мурахами. Початкова стадія



а б

Рис.1.4. Класична схема пошуку їжі мурахами. Етапи руху мурах:
а – через час ***t1***; б – через час ***t2***

Покроковий опис загальної схеми. Припустимо, що довкілля для мурах є повний неорієнтований граф, зображений на рис.1.5. Кожне ребро має вагу, яка позначається як відстань між двома вершинами, сполученими ним. Граф

двонаправлений, тому мураха може подорожувати по грані у будь-якому напрямі.

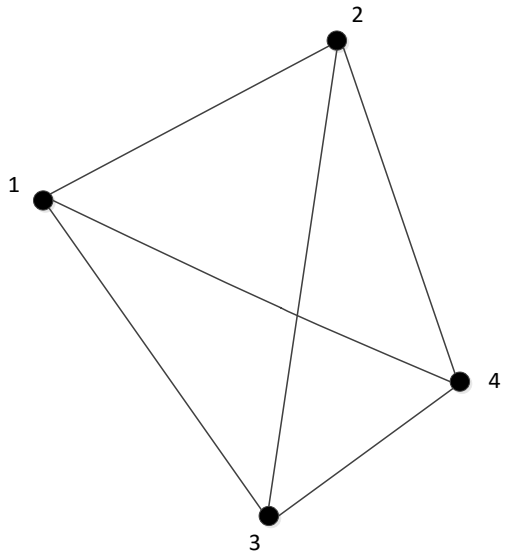


Рис.1.5. Граф побудований по розглянутій класичній схемі пошуку їжі

Імовірність включення ребра в маршрут окремої мурахи пропорційна кількості феромону на цьому ребрі, а кількість феромону, що відкладається, пропорційна довжині маршруту. Чим коротше маршрут тим більше феромону буде відкладено на його ребрах, отже, більша кількість мурах включатиме його складові в синтез власних маршрутів. Моделювання такого підходу, що використовує тільки позитивний зворотний зв'язок, призводить до передчасної збіжності – більшість мурах рухаються по локально-оптимальному маршруту. Уникнути цього можна моделюючи негативний зворотний зв'язок у вигляді випаровування феромону. Причому, якщо феромон випаровується швидко, то це призводить до втрати накопиченого досвіду, що зберігається в пам'яті колонії мурах, та відомостей про оптимальні рішення, з іншого боку, великий час випаровування може призвести до отримання стійкого локального оптимального рішення, без можливості пошуку інших більш оптимальних маршрутів, що також знижує ефективність алгоритму.

Мураха як агент. Будемо розглядати мураху як програмно-апаратного агента, який є членом колективу агентів (колонії мурах) і використовується для вирішення поставленого завдання. Мураха-агент забезпечений набором простих правил та функцій, які дозволяють йому вибирати шлях в КМ. Він підтримує

список вузлів, які вже відвідав. Таким чином, мураха-агент повинен проходити через кожен вузол тільки один раз. Шлях між двома вузлами мережі, по якому мураха відвідав кожен вузол тільки один раз, називається шляхом Гамільтона [95], що і є розв'язком поставленої ЗК.

Вузли в списку відвіданих розташовуються в тому порядку, в якому мураха-агент їх проходив. Пізніше список може використовуватись для визначення протяжності шляху між вузлами. Справжній мураха під час переміщення по дорозі залишає за собою феромон. У алгоритмі колонії мурах агент залишає цифрову чисельну мітку на з'єднаннях між вузлами після завершення подорожі. Початковий вузел, куди поміщається мураха-агент, залежить від обмежень, що накладаються умовами ЗК, оскільки для кожного завдання спосіб розміщення агентів є визначальним. Або усі вони поміщаються в одну точку, або в різні з повтореннями, або без повторень. З метою порівняння результатів з іншими методами та при розв'язанні класичних ЗК, всіх агентів будемо розміщувати в одному початковому вузлі, який можна обирати випадковим чином, або він є заданим в умові ЗК. Тепер розглянемо більш детально класичний алгоритм колонії мурах запропонований Марко Доріго [56,103,105].

1.3. Дослідження алгоритму колонії мурах

Для кращого розуміння класичного алгоритму колонії мурах та його аналізу, з метою проведення, в подальшому, модифікації для збільшення точності та швидкодії, було виконано дослідження основних етапів базового алгоритму, згідно якого алгоритм колонії мурах можна розбити на наступні процедури [8]:

Отримання вхідних даних по ЗК. На вході алгоритм отримує вхідні дані по ЗК: визначений початковий вузел P(0), кількість вузлів N, які необхідно відвідати; матриця вартостей C та матриця доступностей A.

Ініціалізація початкових параметрів. Для алгоритму колонії мурах необхідно вказати: 1) закон нанесення феромону, 2) закон випаровування

феромону, 3) кількість агентів, 4) місця розміщення агентів, 5) параметри для процедури знаходження наступного вузла для переходу мурахи-агента. Усі ці параметри встановлюються з урахуванням особливості завдання на основі експериментальних досліджень (евристики). На цьому ж етапі задаються початкові значення усіх міток Mij, розташованих на з'єднаннях між вузлами, що в сукупності складають матрицю M. Початкове значення для ініціалізації є невеликим позитивним числом, тобто на початковому кроці ймовірності переходу до наступного вузла будуть рівні та не нульові. Фактично пам'ять колонії мурах є онуленої (без наявного досвіду).

Процедура знаходження наступного вузла для переходу мурахи-агента, також її можна позначити як **Процедуру D** (decision – вибір та прийняття рішення стосовно переходу до наступного вузла). Рух мурахи-агента базується на одному досить простому імовірнісному рівнянні. Якщо мураха-агент ще не закінчив шлях, тобто не відвідав усі вузли мережі, для визначення наступного вузла для переходу використовується імовірнісна вибірка. Імовірність вибору для переходу мурахою-агентом вузла j з вузла i обчислюється наступним чином:

$$PS(i,j) = \frac{Mij^{E} \times \eta(i,j)^{L}}{\sum_{k} (Mik^{E} \times \eta(i,k)^{L})}, \qquad (1.5)$$

де **Mij** - поточне значення мітки на з'єднанні з вузла i до вузла j,

η(i,j) - функція, яка відображає вимір зворотної вартості з'єднання,

E - коефіцієнт значимості досвіду пам'яті колонії мурах,

L - коефіцієнт евристики, коефіцієнт значимості вартостей переміщень,

k – індекси доступних для переходу вузлів з вузла **i**, тобто для яких **Aik**=1, відповідно значення мітки та функція зворотнього відображення вартості з'єднання з вузла **i** до вузла **k**: **Mik** та **η(i,k)**.

Вузли, які мураха-агент відвідав, заносяться до списку заборонених для відвідування, який ще називають списком табу (taboo list). Оскільки агент подорожує тільки по вузлах, які ще не були відвідані (відповідно до списку табу), імовірність розраховується тільки для **k** з'єднань, які ведуть до ще не відвіданих, доступних вузлів. Процедура знаходження наступного вузла

відбувається *доки не буде пройдено усі вузли та мураха-агент не повернеться в початковий вузел*.

Процедура «нанесення феромону» - процедура збільшення мурахами-агентами значень міток на з'єднаннях між вузлами мережі. Позначимо її як **Процедуру С** (covering). Після завершення маршруту, вартість шляху може бути підрахована - вона дорівнює сумі вартостей усіх з'єднань, по яких проходив мураха-агент. Зміна значень цифрових міток (кількості феромону) на $\Delta Mij^k(t)$, яка відбувається на кожному з'єднанні, що складає маршрут, для k-ого мурахи-агента обчислюється за наступним рівнянням:

$$\Delta M_{ij}^{k}(t) = \frac{Q}{C^{k}(t)}, \tag{1.6}$$

де **Q** – константа пропорційна до очікуваної сумарної вартості маршруту,

$\mathbf{C^k(t)}$ – сумарна вартість маршруту на момент часу **t** для **k**-ого мурахи-агента.

Результатом рівняння 1.6 є отримання досвіду для пам'яті колонії мурах, причому з'єднання між вузлами мережі, що складають маршрут меншої вартості, отримають більші значення міток ніж з'єднання, що є складовими частинами маршрутів більшої вартості. Отриманий результат $\Delta Mij^k(t)$, використовується в рівнянні (1.7), щоб збільшити значення цифрових міток на кожному з'єднанні між вузлами пройденого мурахою-агентом маршруту. Збільшення значення мітки (накладання феромону) відбувається в кінці кожної ітерації ***циклу пошуку маршрутів мурахами-агентами***, після процедури «випаровування феромону».

$$M_{ij}(t+1) = M_{ij}(t) + \sum_{k} \Delta M_{ij}^{k}(t). \tag{1.7}$$

Рівняння 1.6 застосовується до усього маршруту, при цьому значення мітки кожного з'єднання збільшується пропорційно до вартості маршруту. Тому необхідно дочекатися, доки мураха-агент закінчить подорож і тільки потім оновити значення цифрових міток (рівні феромону) згідно отриманого агентами досвіду, інакше істинна довжина шляху залишиться невідомою.

Процедура «випаровування феромону», позначимо як **Процедуру E** (evaporation). На початку шляху у кожного доступного з'єднання між вузлами є шанс бути обраним мурахою-агентом. Щоб поступово видалити з'єднання, які входять в маршрути великої вартості, до усіх з'єднань застосовується процедура «випаровування феромону». Використовуючи константу P, що визначає інтенсивність випаровування, отримується наступне рівняння для зменшення значення мітки (випаровування феромону):

$$M_{ij}(t+1) = M_{ij}(t) \times (1-P). \quad (1.8)$$

Для «випаровування феромону» використовується зворотній коефіцієнт оновлення маршруту. Константа **P** має значення в межах від 0 до 1.

Цикл пошуку маршрутів мурахами-агентами. Протягом кожної ітерації циклу мурахи-агенти, починаючи свій шлях з початкового вузла P(0), переміщаються по мережі відвідуючи інші вузли, переходячи по з'єднанням між вузлами, що обираються згідно рівняння 1.4. Кожна ітерація циклу пошуку маршрутів мурахами-агентами закінчується поверненням усіх агентів до початкового вузла та збиранням результатів, після чого для кожного k-ого агента очищується список табу та стирається вартість пройденого маршруту разом з записаним результуючим маршрутом R^k. Цикл виконується за однією або декількома з наступних умов: виконано визначену кількість ітерацій; утворення постійної кількості маршрутів; до моменту, коли упродовж декількох запусків не було відмічено повторних змін; досягнення заданої вартості оптимального маршруту; вичерпано визначений ліміт часу обчислення ЗК. Після того, як шлях завершено та мураха-агент повернувся в початковий вузел, проводиться процедура оновлення значень цифрових міток на з'єднаннях між вузлами мережі, яка складається з наступних етапів: 1) «випаровування феромону» на усіх з'єднаннях; 2) збільшення значень міток («нанесення феромону») на пройдених з'єднаннях відповідно до вартості маршруту.

Аналіз пройдених маршрутів мурахами-агентами, вибір серед них найоптимальнішого. Після завершення циклу пошуку маршрутів мурахами-агентами визначається маршрут найменшої вартості серед пройдених мурахами-

агентами, який і є отриманим розв’язком ЗК. Після чого відбувається збереження та *вивід результату*.

Розглянутий алгоритм відноситься до класичного алгоритму колонії мурах, на базі нього було розроблено декілька загальновідомих модифікацій алгоритму, однією з них є застосування *елітних мурах*. Крім основних початкових параметрів алгоритму додатково можна встановити застосування «елітних» мурах-агентів, їх кількість, частота запуску. Їх застосування в певні моменти розглядається як модифікація алгоритму колонії мурах. «Елітні» мурахи-агенти нічим не відрізняються від звичайних агентів, крім того вони пересуваються по мережі так само як звичайні агенти. Мета їх запуску – це збільшення значень міток на з’єднаннях, що входять до знайденої сукупності кращих маршрутів, та, відповідно, відсіювання інших маршрутів, більших за вартістю. Але кількість та частота запуску елітних мурах-агентів ще не є до кінця дослідженими питаннями, оскільки при частому їх використанні можуть залишитись нерозглянутими з’єднання, що входять до потенційно оптимальних машрутів.

Особливості класичного алгоритму колонії мурах. Характерні особливості алгоритму колонії мурах полягають у керуючих параметрах, які необхідно вказати на початку, для успішної роботи алгоритму. Велика кількість існуючих модифікацій класичного алгоритму [43,68] свідчить про продовження пошуку шляхів його вдосконалення, з метою збільшення швидкості отримання кінцевого результату та його точності.

Загальні характеристики алгоритму колонії мурах:

1) реалізує пошук наближених, квазі-оптимальних рішень, є евристичним.

2) здатний розв’язувати задачі поліноміальної складності.

3) є одним з видів імовірнісних алгоритмів.

4) має конструктивний характер знаходження кінцевого маршруту, а отже здатен швидко адаптуватись та коректувати результати при розв’язанні динамічних ЗК.

5) теоретично здатний розв'язувати ЗК з частково невідомими вхідними даними.

Сфери застосування. Алгоритм колонії мурах може бути успішно застосований для вирішення складних завдань оптимізації. Метою такої оптимізації є пошук і визначення найбільш відповідного рішення для оптимізації (знаходження мінімуму або максимуму) цільової функції (ціни, точності, часу, відстані і тому подібне) з дискретної безмежності можливих рішень. Типовий приклад рішення подібної задачі – завдання календарного планування, завдання про призначення, завдання маршрутизації транспорту, різного роду мереж (GPS, телефонні, комп'ютерні і тому подібне), розподіл ресурсів та обов’язків. Ці завдання виникають у бізнесі, інженерії, виробничій, інформаційній, транспортній сферах та багатьох інших областях.

Дослідження [101,103] показали, що алгоритм колонії мурах може давати результати, навіть кращі чим при використанні генетичних алгоритмів і нейронних мереж [15,34], а коли йде мова про можливі динамічні зміни вхідних даних, алгоритм колонії мурах через конструктивний характер обчислення розв'язку є найбільш перспективним для застосування. Що стосується розв’язання ЗК, то алгоритм колонії мурах дає кращі результати за МЗП та еволюційні методи [94,111], які потребують фактично повторного старту після зміни кількості вузлів N, або значного часу для отримання результату наближеного до оптимального при кожній зміні вхідної матриці вартостей C. На практиці алгоритм колонії мурах вже успішно застосовується для децентралізованої маршрутизації в телефонних мережах (British Telecom - BT) та національній цифровій мережі обміну (SDH) Великобританії, також на базі методу колонії мурах було розроблено систему маршрутизації комп’ютерних мереж – AntNet [50], яка ще є прототипом та знаходиться на стадії дослідження [65]. Дана розроблена система не перебуває у відкритому доступі, проте існують викладені ідея та результати досліджень.

Розглянуті особливості евристичного алгоритму колонії мурах можуть бути успішно застосовані для вирішення складних комбінаторних завдань оптимізації. Крім розглянутих аспектів, слід зазначити ще одну ідею, що лежить в основі алгоритму колонії мурах, яка полягає у використанні механізму

позитивного зворотнього зв'язку, який допомагає знайти найкраще наближене рішення в складних завданнях оптимізації. Тобто, якщо в поточному вузлі мураха-агент повинен вибрати між різними варіантами, і якщо фактично вибрані варіанти (наступні вузли для переходу) будуть мати кращі результати, то в майбутньому такий вибір буде бажаніший, ніж попередні. Цей підхід є багатообіцяючим через його ефективність у виявленні оптимальних рішень складних проблем.

Проте класичний алгоритм колонії мурах потребує багато часу для отримання результату максимально наближеного до оптимального. З метою збільшення точності отриманих результатів при розв’язанні динамічної ЗК та зменшення часу обчислення результуючого маршруту, найбільш наближеного до оптимального, доцільно розглянути можливість вдосконалення – модифікації базового класичного алгоритму мурашиної колонії.

1.4. Перспективність застосування методів локальної оптимізації

Методи локальної оптимізації [63,73] застосовуються для розв'язання складних у обчисленні оптимізаційних задач. Методи локальної оптимізації, або як їх ще називають методи локального пошуку, можуть використовуватись для задач, що формулюються як знаходження розв'язку, максимального за певним критерієм, серед існуючих можливих рішень. Ідея методів локальної оптимізації полягає у перебиранні можливих розв'язків шляхом виконання локальних змін, доки результат не зведеться до оптимального або не буде вичерпано певний ліміт часу чи кількість спроб.

Методи локальної оптимізації застосовуються для розв'язання великої кількості задач в області комп'ютерних наук, штучного інтелекту, математики, операційних досліджень, інженерії та біоінформатики. Як приклади можна навести GSAT та WalkSat [76,104], але саме для розв'язання ЗК найбільш популярними є k-opt алгоритми [63,71].

Більшість задач може бути сформульована в термінах «простору» та «цілі» декількома різними шляхами. Так наприклад, для ЗК розв'язок є маршрут

одноразового відвідування усіх N вузлів та повернення до початкового вузла P(0) – цикл Ейлера, а критерієм максимізації буде комбінація певної кількості вузлів та довжина маршруту. Але розв'язком може також бути маршрут, а утворити цикл Ейлера буде ще одною частиною завдання для локального пошуку, такий принцип використовується в LKH методі [63] при «склеюванні» окремих маршрутів.

Методи локальної оптимації починають аналіз від вхідного певного розв'язку або його частини та послідовно переходять до сусіднього, що є можливим тільки тоді, коли відношення сусідства між розв'язками чи його частинами визначено в просторі пошуку. Як приклад, сусідній машрут від існуючого для ЗК, є інший маршрут, що відрізняється лише перестановкою одного вузла. Отже для ЗК кількість сусідніх розв'язків залежить від кількості вузлів N. Відповідно до коефіцієнту складності k методу локальної оптимізації, збільшується кількість з’єднань між вузлами, що буде переставлено, та відповдіно кількість сусідніх розв'язків для аналізу.

Умова припинення виконання обчислень зазвичай визначається як обмеження по часу або ж у кількості спроб проведення локальної оптимізації. Методи локальної оптимізації можуть бути зупинені у будь-який час виконання, надаючи при цьому коректний результат при будь-якій ситуації. По завершенню свого виконання, тобто коли можливих локальних покращень не виявлено, дані методи не гарантують найоптимальнішого результату. Це зазвичай відбувається у випадках, коли складності методу локального пошуку недостатньо для виявлення кращого сусіднього розв'язку. Методи локальної оптимізації є евристичними.

Алгоритм Ліна-Кернігана (k-opt алгоритм) потребує знаходження ефективного коефіцієнта складності k, коли його значення динамічно змінюється та може набувати значення більші за 5, такі методи зазвичай називають методи пошуку розширенного сусідства, що є досить ефективними, проте в деяких випадках приймають експоненційну складність та часові витрати. Особливістю методу LKH, що включає алгоритм Ліна-Кернігана, є те,

що він працює зі змінною k, значення якої перевизначається в кожній ітерації локального пошуку. На базі методу LKH було розроблено систему для розв'язання ЗК – LKH 2.0 [83]. При встановленні списку сусідів для вузла метод локального пошуку зменшує складність з $O(n^2)$ до O(m*n), однак якщо m взяти недостатньо великим ефективність алгоритму локальної оптимізації зменшується.

Згідно твердження С.Ліна [71] 3-opt алгоритм продемонстрував найкращі результати у співвідношенні з обчислювальним часом, також досить часто використовуються й інші алгоритми низької складності: 2-opt та 2,5-opt. Отже методи локальної оптимізації з коефіцієнтом складності $k < 4$ можуть суттєво покращити точність результатів, що отримуються алгоритмом колонії мурах, без значного збільшення часу обчислення. Аналіз на можливість оптимізації ділянки маршруту не анулює можливість обчислення ЗК динамічного характеру.

1.5. Вибір засобів розробки. Формулювання вимог до апаратних засобів

Для розробки багатоагентних систем та моделювання базового алгоритму, а також проведення тестування було використано середовище розробки Microsoft Visual Studio 2011 [109], обраного як одного з надійних ефективних засобів розробки програмного забезпечення на мові C та C++ [20]. З метою зберігання вхідних та вихідних даних необхідна наявність накопичувача інформації, у якому будуть зберігатись текстові файли з даними по кожному етапу та результати. Для системи розв'язання динамічної ЗК використовуються наступні файли: 1) файл з вхідними даними та параметрами (формалізовані матриці A, C та розмірністю N); 2) файл з координатами вузлів (якщо така інформація присутня); 3) файл з послідовністю вузлів (результуючий маршрут);

Для реалізації графічного відображення вхідних та вихідних даних, а також для створення графіків, було використано засоби MATLAB R2010b [100]. Більш детальний опис програмних засобів, що були використані для розробки системи, наведений в додатку Є. Для початкової розробки, налагодження та

тестування системи було використано один персональний комп’ютер (тактова частота процесора Celeron 2,93 ГГц, одноядерний, 4 ГБ ОЗУ, NVIDIA GeForce 6600 PCI-E графічний адаптер, операційна система – Windows 7), отже мінімальні вимоги до комп’ютера для нормального режиму роботи з використовуваними середовищами розробки наступні:

Вимоги Visual Studio 2011 щодо програмного забезпечення. Його можна встановити в наступних операційних системах:

-- Windows XP Service Pack 3;

-- Windows Server 2003 R2 Service Pack 2;

-- Windows Vista Service Pack 1/2;

-- Windows Server 2008 Service Pack 2/R2;

-- Windows 7;

-- Windows 8;

Архітектури, що підтримуються:

-- 32-розрядна (x86).

-- 64-розрядна (x64).

Вимоги щодо обладнання:

-- Процесор з частотою 1,8 ГГц або вище.

-- 1024 МБ ОЗП.

-- 2,2 ГБ вільного місця на диску.

-- Жорсткий диск з швидкістю 5400 об/хв..

-- Відеоадаптер з підтримкою DirectX 9 і вище та розширенням від 1024 x 768 (або більш високим).

Системні вимоги MATLAB R2010b:

-- Операційні системи:

-- Windows XP Service Pack 3;

-- Windows Server 2003 R2 Service Pack 2;

-- Windows Vista Service Pack 1/2;

-- Windows Server 2008 Service Pack 2/R2;

-- Windows 7;

-- Linux / UNIXOS;

-- ЦП (CPU): Будь-який процесор Intel або AMD x86 з підтримкою SSE2.

-- ОЗП: 1024 МВ (рекомендовано 2048 МБ)

-- Жорсткий диск (HDD): 1 ГБ тільки для базового комплекту MATLAB, 3–4 ГБ для стандартного встановлення.

З метою збільшення швидкодії багатоагентної системи з використанням поведінкової моделі колонії мурах можливим є застосування технологій паралельних обчислень. Виконання обчислення обходження мурахою-агентом вузлів може бути виконано в окремому потоці кожної ітерації циклу пошуку маршрутів мурахами-агентами. Тобто кількість потоків, в яких паралельно обчислюються маршрути, дорівнює кількості задіяних мурах-агентів. «Кількість ядер» в процесорі використовуваного персонального комп'ютера визначає кількість можливих для одночасного виконання потоків. При чому сума «ядер» (одноядерний комп'ютер рахується як одне ядро) має дорівнювати кількості агентів, які запускаються для досягнення максимальної швидкодії. Загалом вимоги до обчислювального вузла співпадають з вище описаними вимогами до середовища Visual Studio 2011.

Для застосування технологій паралельного програмування було вирішено використати OpenMP та _pthread API, які є включено в пакет бібліотек C++ 11-го стандарту та доступні при застосуванні Visual Studio 2011. Оголошення та запуск паралельного обчислення маршрутів потоками, які виступають як агенти в багатоагентній системі, відбувається за допомогою компілятора. Така стратегія паралельного пошуку маршрутів агентами суттєво збільшує продуктивність системи колективної поведінки агентів для розв'язання динамічної ЗК з використанням методів локальної оптимізації без потреби застосування більш потужної апаратної складової. Детальні дослідження проводились на ЕОМ з наступними характеристиками апаратної складової: Intel Core i7-5960X Haswell-E 8-Core 3.0 GHz, 8 ядер, 16 ГБ ОЗУ, NVIDIA GeForce 6600 PCI-E графічний адаптер. При застосуванні ЕОМ з більшою кількістю ядер очікується ще більше прискорення процесу обчислення результату.

1.6. Висновки до розділу

У цьому розділі проведено аналіз існуючих методів, засобів та моделей для розв’язання багатокритеріальної ЗК. Отримано такі результати:

- розглянуто основні характеристики багатокритеріальної ЗК та динамічної асиметричної ЗК в умовах частково невідомих вхідних даних;

- запропоновано класифікацію існуючих методів розв’язання ЗК, аналіз якої показав, що для розв’язання динамічної ЗК при великій кількості пунктів (N > 100) потрібно орієнтуватись на методи із застосуванням колективної поведінки агентів, серед яких найбільш перспективниий – метод колонії мурах.

- проаналізовано можливість застосування поведінкової моделі колонії мурах при розв’язанні динамічної асиметричної ЗК в умовах частково невідомих вхідних даних, розглянуто основні процедури алгоритму колонії мурах та набір необхідних вхідних параметрів та даних;

- проведений аналіз існуючих обчислювальних засобів та комп’ютерних систем для розв’язання ЗК показав, що доцільно використати методи локальної оптимізації для опрацювання результуючого маршруту, з метою зменшення його вартості в умовах динамічних змін вхідних даних.

2. ВДОСКОНАЛЕННЯ БАЗОВОГО МЕТОДУ ТА ЗАПРОПОНОВАНІ НОВІ МЕТОДИ ТА ЗАСОБИ ДЛЯ РОЗВ'ЯЗАННЯ ДИНАМІЧНОЇ ЗК

2.1. Вдосконалення базового методу

З метою вдосконалення існуючого базового методу колонії мурах [56], в процесі моделювання алгоритму та аналізу колективної поведінки агентів, було запропоновано нову модифікацію алгоритму [3,9], яка забезпечує зменшення часу обчислення ЗК при застосуванні алгоритму колонії мурах, без збільшення вартості отримуваних результуючих маршрутів. На рис.2.1 представлено дві блок-схеми алгоритмів: класичний алгоритм колонії мурах (рис.2.1.а) та алгоритм колонії мурах з врахуванням запропонованої модифікації (рис.2.1.б). Процедура D (decision) – процедура знаходження наступного вузла для переходу мурахи-агента (див. стор. 28), Процедура E (evaporation) – випаровування феромону (див. стор. 30) та Процедура C (covering) – нанесення феромону (див. стор. 29), як і інші блоки було описано в першому розділі під час огляду класичного алгоритму колонії мурах (див. розділ 1.3). Метод розв'язання ЗК, який базується на використанні поведінкової моделі колонії мурах, було удосконалено шляхом зміни початкової установки значень міток (додано нову процедуру ініціалізації - Процедура I (initialization)) та імовірнісного вибору наступного вузла для переходу мурахи-агента (змінено існуючу процедуру D на процедуру mD (modified decision)).

Додавання початкової установки значень міток (рівня феромону (Процедура I на рис.2.1.б). В класичному алгоритмі колонії мурах вказано, що початкова установка рівня феромону має проводитись заданою невеликою константою, більшою за 0. Це призведе до початкової рівномірної ненульової ймовірності вибору з'єднання між вузлами для переходу агентом. Однак, якщо ввести початкову ініціалізацію не константою, а значеннями згідно початкового аналізу вартості з'єднання, які обчислюватимуться за рівнянням 1.6 при k = 1, початкова імовірність вибору також буде ненульовою. Всі значення ймовірності знаходяться в межах від 0 до 1. Сума імовірностей доступних виборів дорівнює

1. Фактично відбудеться занесення до пам’яті колонії мурах (матриця M) початкового стану мережі, що дає ряд вагомих переваг:

1) Фактичний «пробіг» мурахами-агентами з метою аналізу та оцінки одразу по всім з’єднанням, жодне не залишиться без уваги. Відповідно, виявлення найменш придатних або не доступних з’єднань для прокладання маршруту в момент ініціалізації.
2) Від самого старту виконання алгоритму агенти орієнтуються у виборі шляху не на просту випадковість, а на вже сформований певний досвід – початкову ініціалізовану пам'ять колонії мурах, що дає змогу значно пришвидшити алгоритм знаходження оптимального шляху, особливо при розв’язанні статичної ЗК.
3) Зменшення кількості ітерацій циклу пошуку машрутів мурахами-агентами необхідної для визначення оптимального шляху та збільшення важливості іншого параметру – кількості мурах-агентів. Як показує практика, запуск більшої кількісті мурах-агентів практично не збільшує час обчислення в багатоядерних системах та є менш ресурсо затратним, ніж збільшення кількості ітерацій циклу пошуку машрутів мурахами-агентами.

Слід зазначити, що Процедура I стосується більше випадку з відомими наперед вхідними даними, а також трохи збільшує, відповідно до розмірності матриці вартостей C, час ініціалізації.

Зміна імовірнісного вибору наступного вузла для переходу мурахи-агента (Процедура mD на рис.2.1). За класичним алгоритмом колонії мурах [56,101] зазвичай обирається наступне найбільш імовірнісне з’єднання для переходу, згідно величини ймовірності його вибору. А якщо врахувати, що і значення мітки після ініціалізації також залежить від вартості з’єднання через застосування Процедури I, то існує імовірність зведення алгоритму до жадібного алгоритму, або як його ще називають – алгоритму «найближчого сусіда» (greedy algorithm [95]).

Для реалізації імовірністного вибору було вирішено застосовувати генератор випадкових чисел з нормальним розподілом, на далі *common random*,

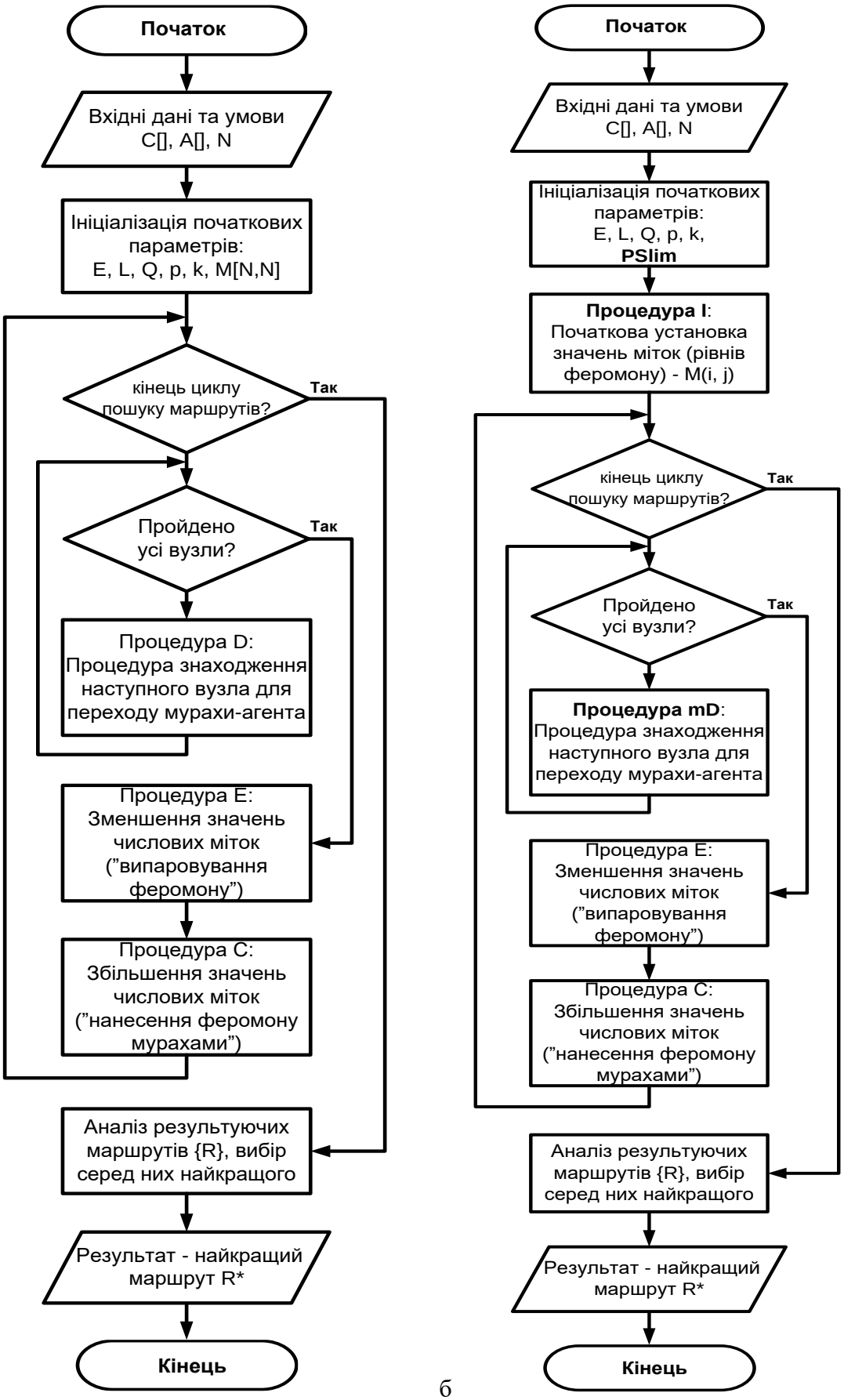


Рис.2.1. Порівняння блок-схем алгоритмів: а) класичний алгоритм колонії мурах, б) алгоритм колонії мурах з врахуванням запропонованої модифікації

та «взваженний» генератор псевдовипадкових чисел з рівномірним розподілом, на далі weighted random [84], який забезпечує умову вибору з’єднань згідно величини ймовірності їх обирання, тобто чим більше імовірність, тим більший шанс обирання. Обидва генератора функціонують на базі лінійного конгуентного методу генерації псевдовипадкових чисел (LCRNG – Linear Congruential Random Number Generator) [92].

З метою вдосконалення базового методу та усунення імовірності зведення результатів алгоритму мурашиної колонії до результатів жадібного алгоритму підраховані ймовірності аналізуються наступним чином:

1) Встановлюється нижня імовірнісна межа – **PSlim**, як додатковий початковий параметр алгоритму.
2) Аналізуються усі можливі переходи до ще не відвіданих вузлів та створюється список — пул (*BestNodes* список), з найбільш придатних для переміщення, імовірність обирання яких вища за **PSlim**. Інші доступні вузли для переходу потрапляють до списку *LowChanceNodes*.
3) Якщо *BestNodes* список є порожнім, то за допомогою «взваженого» генератора обирається наступний вузел для переходу.
4) В інакшому випадку, для визначення списку вузлів, який буде застосовуватись, обчислюється значення граничної ймовірності **PSb** наступним чином:

$$\mathbf{PSb = (PSmin + 1 / NA) / 2}, \quad (2.1)$$

де **PSmin** – мінімальне значення імовірності, яке обчислюється для вузлів з списку *LowChanceNodes*,

NA – кількість доступних вузлів для переходу (сумарна розмірність *LowChanceNodes* та *BestNodes* списків). Фактично це середнє значення імовірності, що обчислюється для всіх доступних вузлів для переходу.

5) Генерується за допомогою генератора псевдовипадкових чисел з нормальним розподілом число **PSs**. Якщо **PSs ≥ PSb**, то використовується *BestNodes* список вузлів, а з’єднання для переходу вибирається шляхом використання «взваженого» генератора псевдовипадкових чисел (з

рівномірним розподілом), інакше використовується *LowChanceNodes* список та генератор псевдовипадкових чисел з нормальним розподілом.

Такий метод визначення мурахою-агентом наступного вузла забезпечує гнучкість алгоритму, постійний розгляд можливих більш оптимальних відхилень від вже існуючих знайдених оптимальних маршрутів. Регулювання значення **PSlim** дозволяє визначати межу ігнорування низьких значень імовірностей, а також дозволяє уникнути зведення до жадібного алгоритму.

Запропоновану нову модифікацію базового алгоритму було використано при розробленні багатоагентної системи з використанням поведінкової моделі колонії мурах для розв'язання динамічної ЗК. Проведені дослідження (розділ 4.1) функціонування розробленої багатоагентної системи продемонстрували ефективність запропонованої модифікації алгоритму колонії мурах.

2.2. Застосування методів локальної оптимізації для опрацювання результатів при розв'язанні динамічної ЗК

Передбачається, що використання методів локальної оптимізації, ідею яких розглянуто в розділі 1.4 при розробці багатоагентної системи, дозволить збільшити точність результату, отриманого алгоритмом колонії мурах з врахуванням запропонованої модифікації алгоритму. Для розробки додаткового програмного модуля [26,27] на базі методів локальної оптимізації необхідно розробити програмну реалізацію обраних через невелику складність та відповідно час обчислення 3-х алгоритмів: 2-opt, 2.5-opt, 3-opt. Оскільки для даних методів було знайдено лише теоретичне обгрунтування ідеї, а більшість вже створених реалізацій закриті для ознайомлення, вирішено розробити блок-схеми перелічених алгоритмів локальної оптимізації для кращого розуміння та імплементації в систему. Представимо КМ у вигляді графу, вершини якого – вузли (міста в блок-схемах), з'єднання – ребра.

На рис.2.2 представлено розроблену блок-схему програмної реалізації 2-opt алгоритму. В програмі введено наступні позначення: **tour** (R) – масив, що

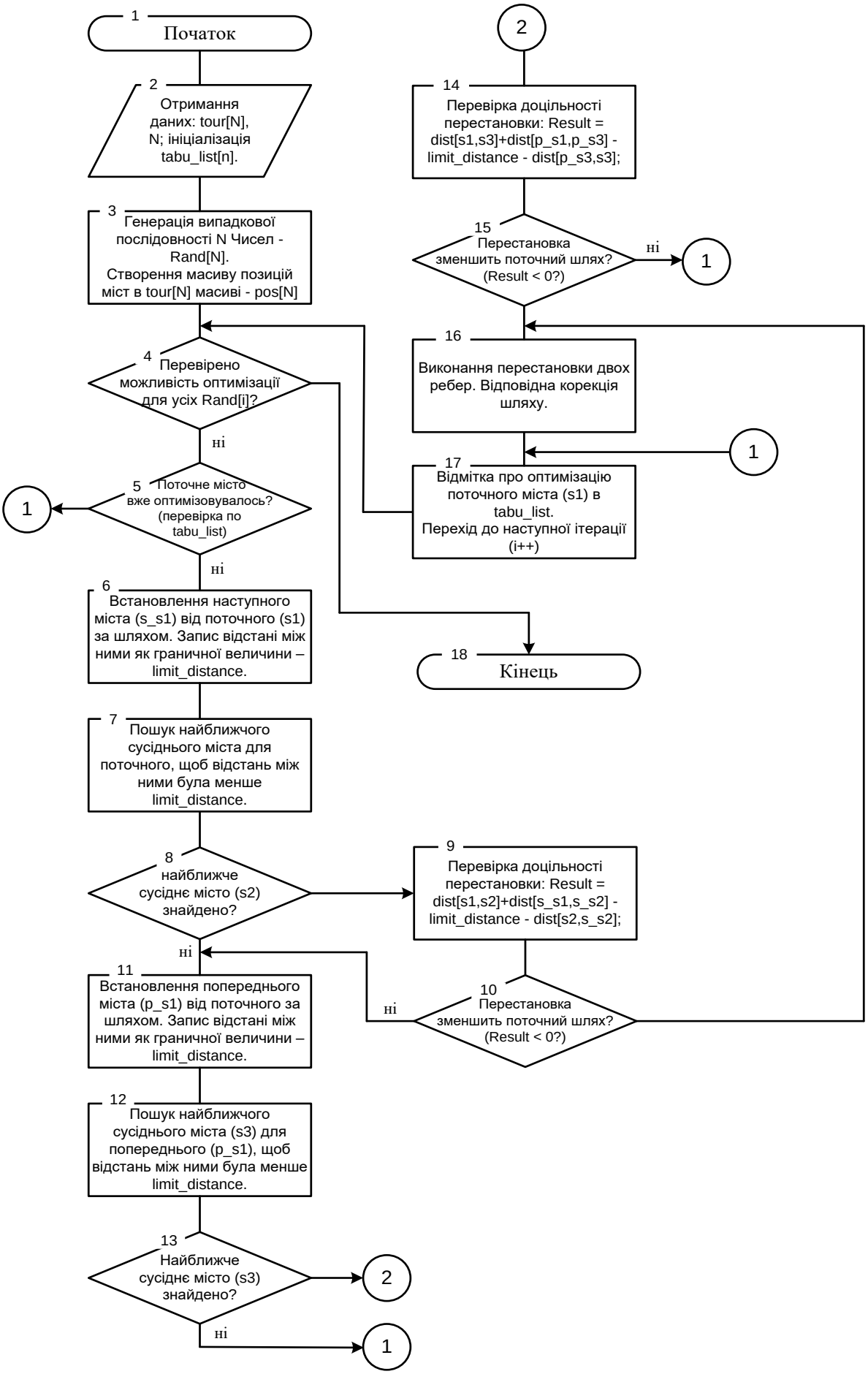


Рис.2.2. Блок-схема програмної реалізації 2-opt алгоритму

включає в себе отриманий маршрут, в випадку успішного розв'язання ЗК, розмірністю N, **tabu_list** – масив, розмірністю N, що зберігає індекси вузлів, які були відвідані мурахою-агентом.

Етап генерації випадкової послідовності N чисел - **Rand[N]** та створення **Pos[N]**, масиву позицій вузлів в масиві tour, реалізований для випадкового перебору вузлів при локальній оптимізації.

Розглянемо довільно обрану ділянку графу, з вже прокладеними з'єднаннями між вершинами (частина маршруту). Вузли, що приймають участь у перевірці на доцільність перестановки, позначимо як s1 та s2 (station). Попередній до s1 вузел згідно маршруту – p_s1 (previous to s1), наступний – s_s1 (sequent after s1). Аналогічно і до вузла s2: p_s2 та s_s2.

Якщо поточний вузел s1 ще не був оптимізований, проводимо пошук вузла s2 так, щоб вартість переходу з s1 до s2 була менше за вартість переходу з s1 до s_s1, яка позначена на блок-схемі (див. рис. 2.2) як limit_distance, або як R– радіус на рис. 2.3. Після чого виконується перевірка доцільності перестановки: вилучення ребер s1-s_s1 та s2-s_s2, додавання ребер s1-s2 та s_s1-s_s2. В подальшому будемо називати дану перестановку як 2-opt перестановку 1-го типу, яка відбувається з задіяними наступними вузлами за маршрутом відносно поточного s1 та знайденого s2 вузлів. Виконання даної перестановки для певної ділянки маршруту зображено на рис. 2.3.

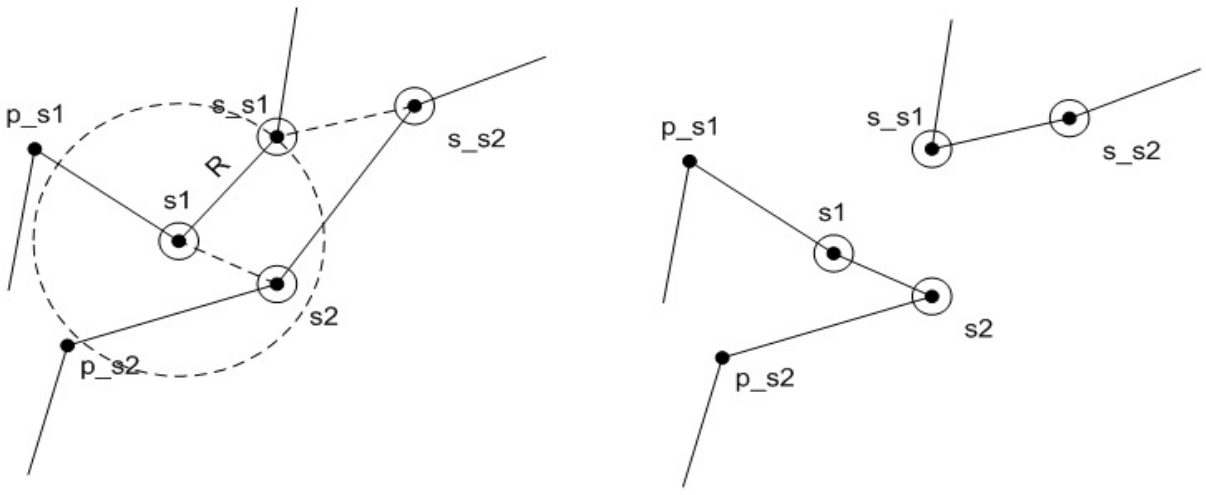


Рис.2.3. Виконання в 2-opt алгоритмі перестановки 1-го типу на довільно обраній ділянці графу

Тут і надалі зліва зображено початковий стан, справа – стан маршруту після перестановки. Колами обведено вузли, що приймають участь у перестановці. Якщо дана перестановка (див. рис. 2.3.) зменшить поточний маршрут за вартістю, то відбувається перехід до етапу виконання перестановки, тобто відповідної корекції маршруту, поточний вузел відмічається як оптимізований, та відбувається перехід до наступного. Якщо ж дана перестановка не доцільна, то відбувається пошук вузла s3 (див. рис. 2.2), або фактично попереднього для деякого вузла s2 так, щоб тепер вартість переходу з p_s1 до p_s2 була менше за вартість переходу з p_s1 до s1, що позначена як R – радіус на рис. 2.4. Після чого виконується перевірка доцільності перестановки: вилучення ребер p_s1-s1 та p_s2-s2, додавання ребер p_s1-p_s2 та s1-s2. Будемо називати її як 2-opt перестановку 2-го типу, в якій задіяні попередні за маршрутом вузли відносно поточного s1 та деякого s2, причому p_s2 фактично є вузлом s3 згідно блок-схеми (див. рис. 2.2). Такий тип перестановки для деякої ділянки маршруту зображено на рис. 2.4.

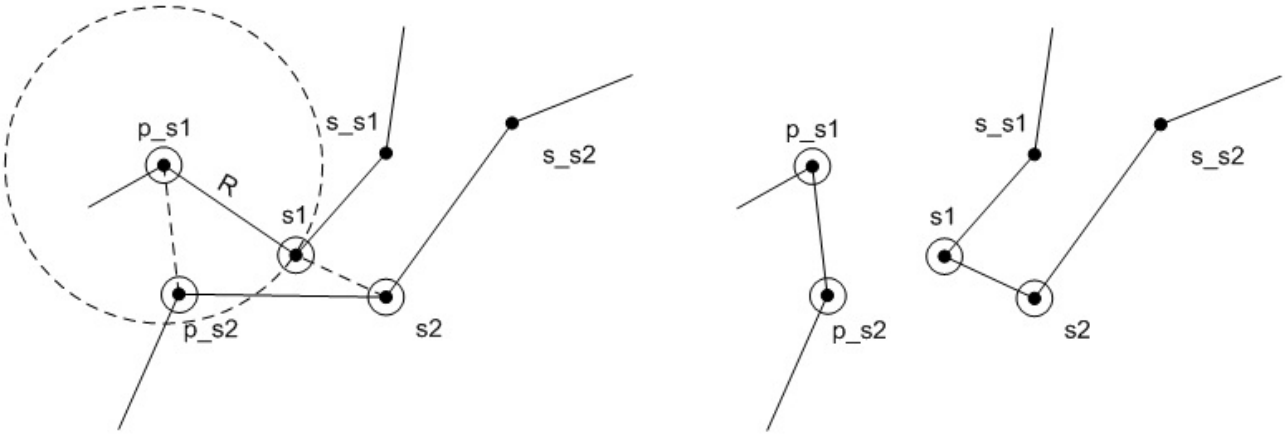


Рис.2.4. Виконання в 2-opt алгоритмі перестановки 2-го типу на довільно обраній ділянці графу

Якщо дана перестановка зменшить поточний маршрут за вартістю аналогічно як і до першого типу перестановки виконується перехід до корекції шляху та продовження циклу локальної оптимізації.

Фактично даний алгоритм при потребі можна зупинити на будь-якому етапі. Аналогічно до розглянутого алгоритму 2-opt реалізовується 2.5-opt

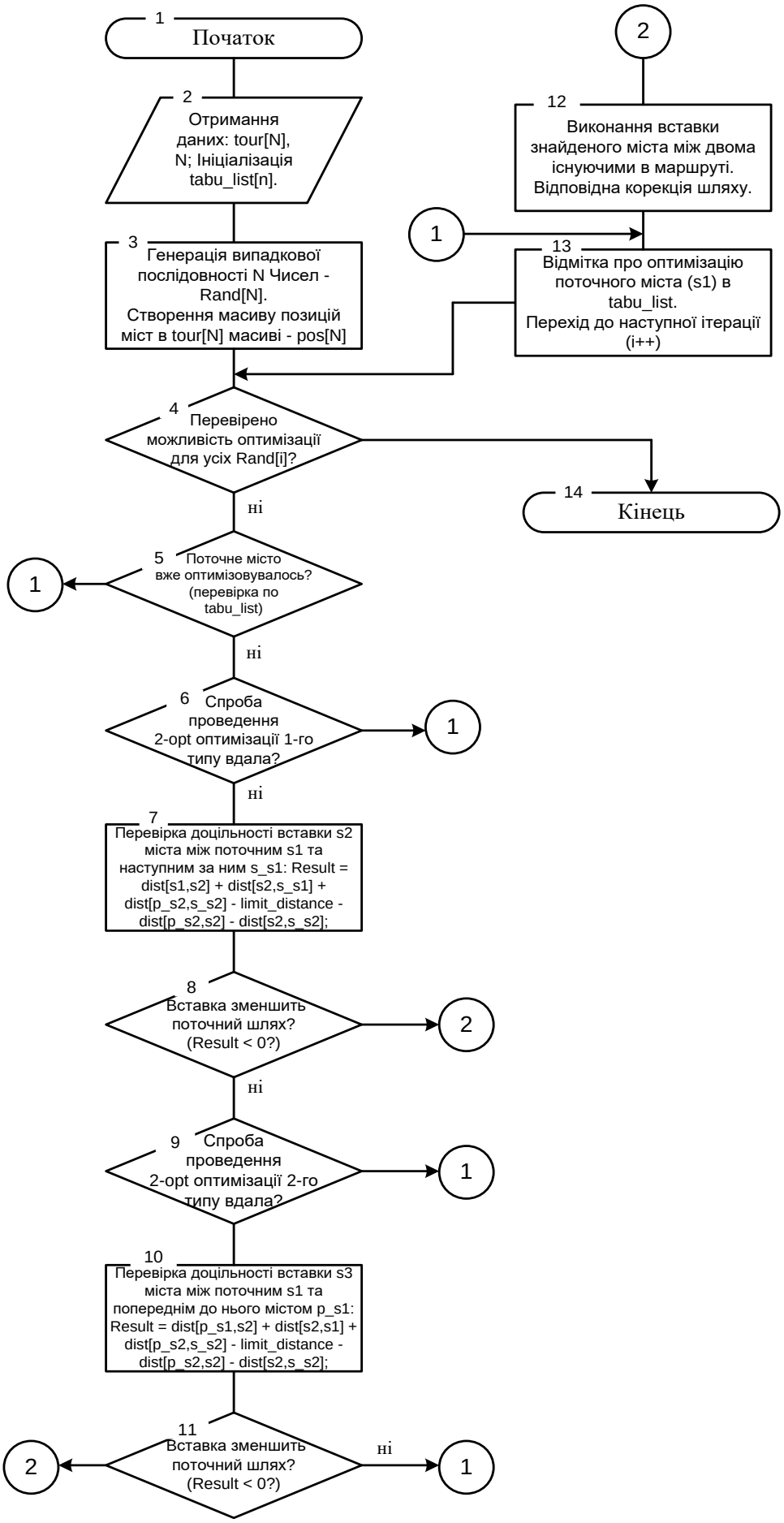


Рис.2.5. Блок-схема програмної реалізації 2.5-opt алгоритму

алгоритм (рис.2.5), який частково включає в себе етапи 2-opt алгоритму та елементи 3-opt алгоритму, оскільки при вилученні вузла з одного місця маршруту та вставки його в іншу ділянку також задіяні три ребра. Як бачимо з розробленої блок-схеми алгоритму (див. рис. 2.5) принцип організації локальної оптимізації схожий до методу 2-opt, з аналогічними позначками. Отже виконуються спроби оптимізації 2-opt перестановкою 1-го та 2-го типу, аналогічно до 2-opt алгоритму. Якщо перестановки не дають покращення маршруту для поточного вузла s1, відбувається аналіз доцільності вставки вузла s2 з пулу доступних для переходу з вузла s1, між вузлами s1 та s_s1 – будемо називати дану вставку 2.5-opt вставкою 1-го типу. Дану вставку для певної ділянки маршруту ЗК зображено на рис. 2.6. Якщо даний тип вставки не зменшує маршрут за вартістю, то виконується аналіз на доцільність вставки в маршрут деякого вузла s3 між вузлами p_s1 та s1. Даний тип вставки називатимемо 2.5-opt вставкою 2-го типу, яка зображена на рис. 2.7, де вузлу s3 на рисунку відповідає вузел s2.

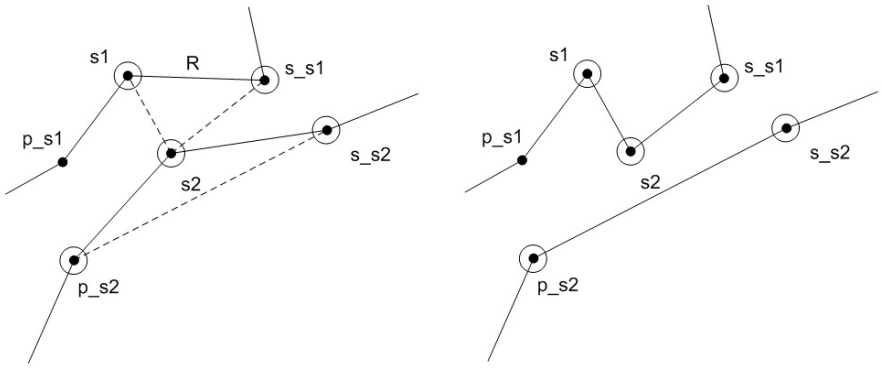


Рис.2.6. Виконання в 2.5-opt алгоритмі вставки 1-го типу на довільно обраній ділянці графу

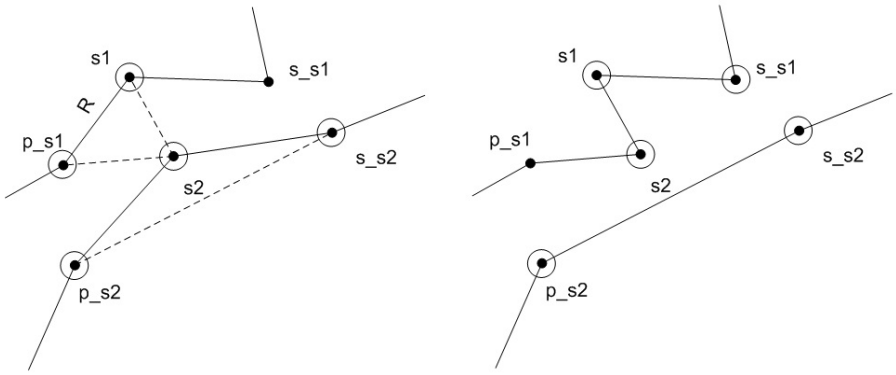


Рис.2.7. Виконання в 2.5-opt алгоритмі вставки 2-го типу на довільно обраній ділянці графу

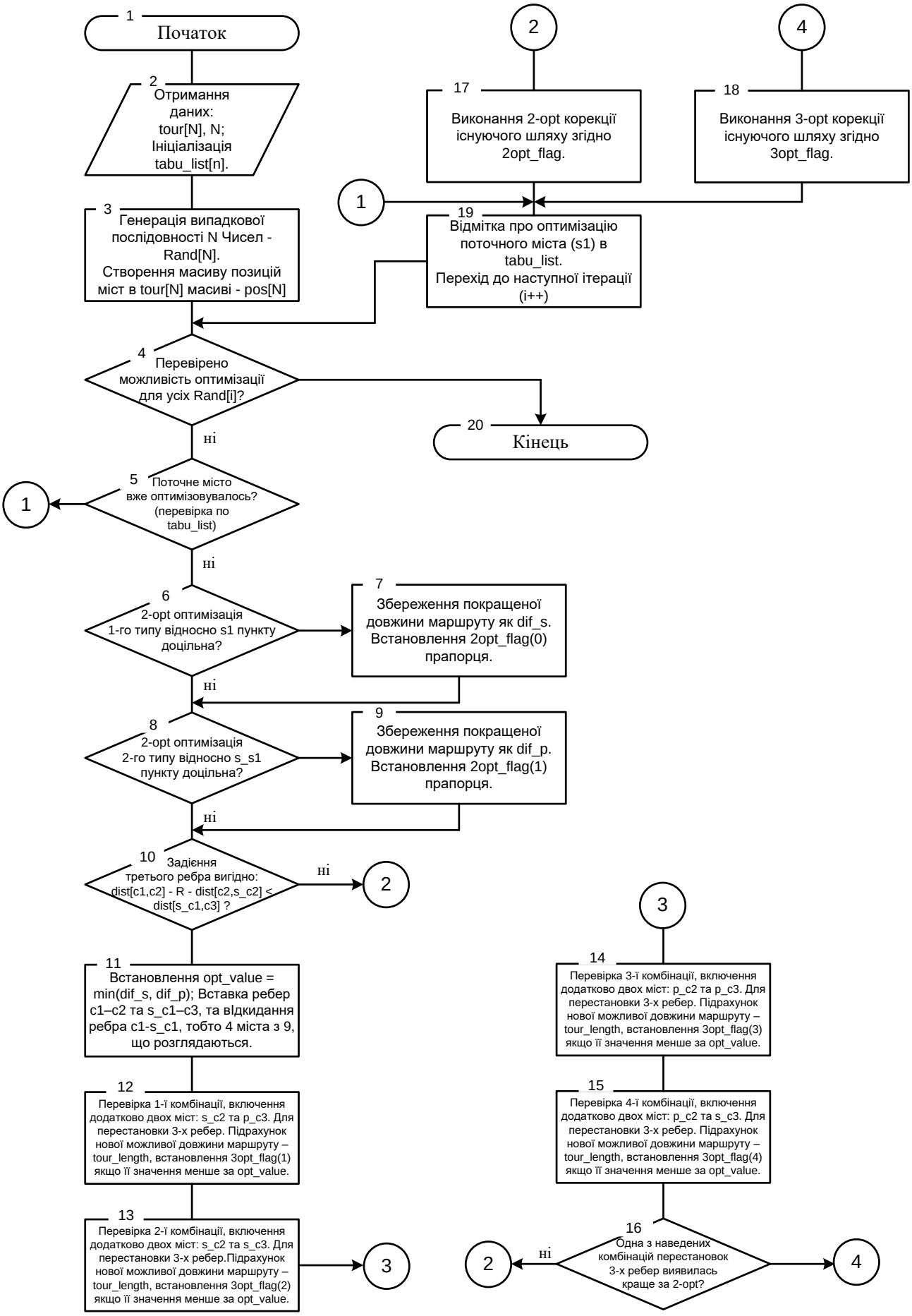


Рис.2.8. Блок-схема програмної реалізації 3-opt алгоритму

На рис.2.8 представлено розроблену блок-схему програмної реалізації 3-opt алгоритму. Даний алгоритм функціонує схоже до двох попередньо розглянутих з аналогічними позначками та вхідними даними. Етап оптимізації є набагато складнішим відмінно від 2-opt та 2.5-opt, де було лише 2 та 4 відповідно перевірки.

Для 3-opt алгоритму виконуються спочатку дві 2-opt перевірки зображені на рис.2.9, при чому якщо вони доцільні, встановлюється прапорець 2opt_flag. На рис.2.9 зліва зобажено 2-opt перестановки 1-го типу відносно вузла s1, справа – 2-opt перестановку 2-го типу відносно вузла s_s1. Вузли обведені колами приймають участь у перестановці ребер. Пунктиром наведено ребра, що утворюються після перестановки. В першому випадку вилучається ребро s2-s_s2, в другому – p_s2-s2, ребро s1-s_s1 вилучається в обох випадках.

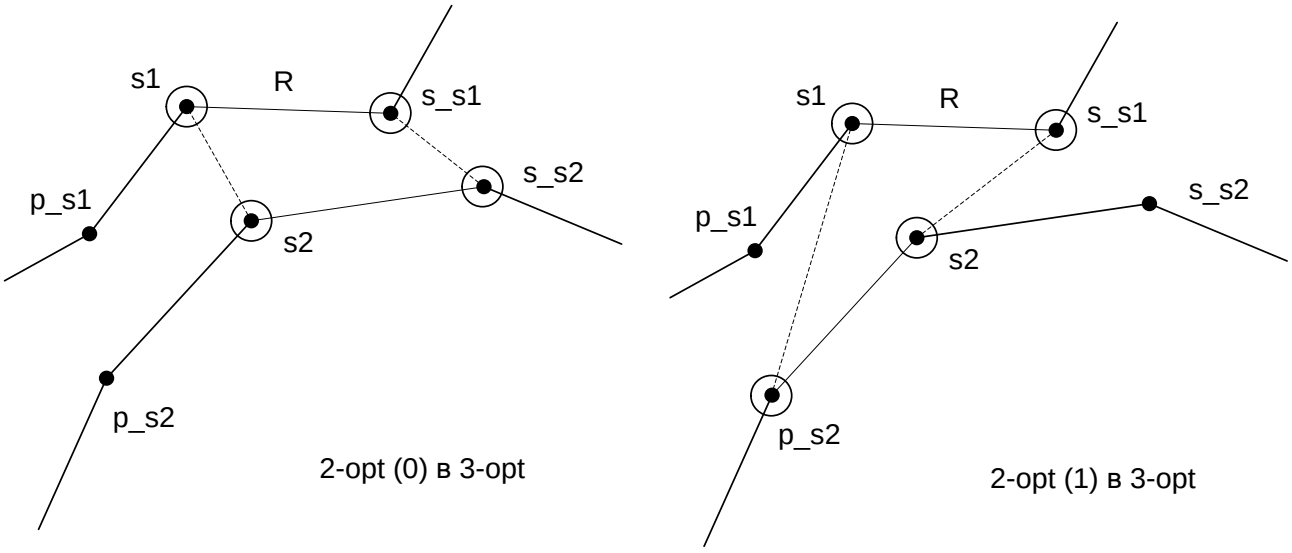


Рис.2.9. 2-opt перестановки в 3-opt алгоритмі

Після чого відбувається подальша перевірка на доцільність перестановок 2-х ребер з обраним при 2-opt аналізі вузлом s2 та знайденим вузлом s3, що має можливість для переходу до вузла s_s1. Причому вузли s2 та s3 обрані так, щоб ребра s3-s_s1 та s1-s2 були менші за ребро s-s_s1, яке позначено як R–радіус.

Якщо перестановки з вилученням ребра s1-s_s1 та додаванням ребер s1-s2 та s_s1-s3 є доцільними йде аналіз, як саме необхідно провести переставноку. В розробленій реалізації 3-opt алгоритму пропонується перевірка 4-х варіантів комбінацій перестановок ребер, зображені на рис.2.10.

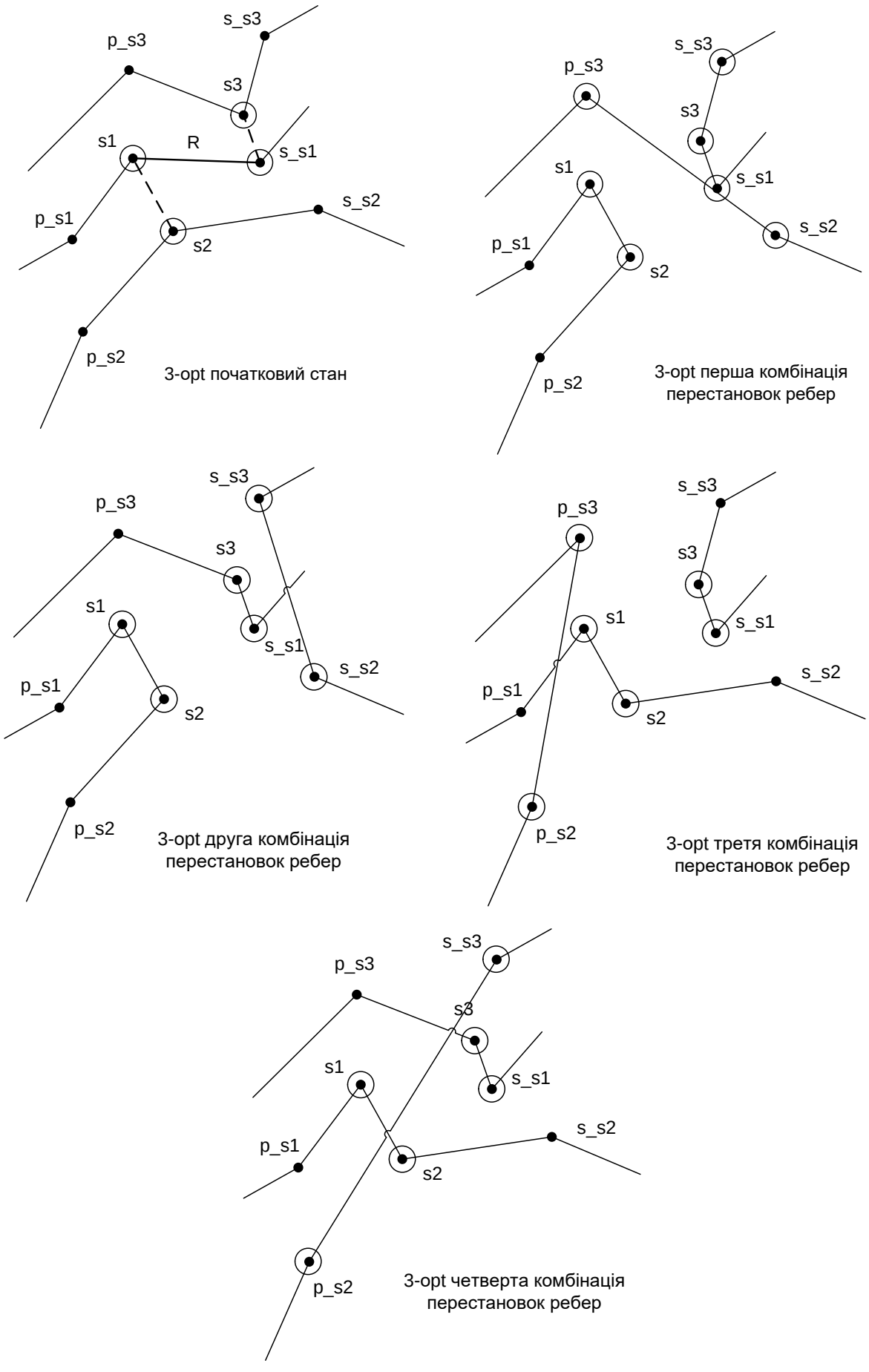


Рис.2.10. Комбінації перестановок в 3-opt алгоритмі

Колами обведено вузли, задіяні в перестановці в кожній комбінації. Ребра позначені пунктирами додаються, ребро s1-s_s1 вилучається у кожній з наведених кобінацій. Після перевірки кожної з комбінацій зберігається можлива довжина покращеного маршруту та встановлюється прапорець 3opt_flag, який визначає доцільність проведення перестановок 3-х ребер.

Запропонований **метод опрацювання результуючого маршруту** при розв'язанні динамічної задачі комівояжера, який базується на використанні алгоритмів локальної оптимізації 2-opt, 2.5-opt, 3-opt, полягає в зміні режиму опрацювання результуючих маршрутів в залежності від інтенсивності динамічних змін вхідних даних. Тобто складність алгоритму локальної оптимізації **k** визначається інтенсивністю динамічних змін вхідних даних: при високій інтенсивності використовується 2-opt алгоритм, при низькій інтенсивності або відсутності динамічних змін – 3-opt алгоритм.

Після закінчення перевірки доцільності всіх розглянутих перестановок відбувається перехід до корекції маршруту згідно встановлених прапорців 2opt_flag чи 3opt_flag, відповідно до активного режиму опрацювання результуючих маршрутів. Після проведення корекції чи якщо покращити не вдалось (відсутні вище описані прапорці) поточний вузел s1 відмічається в списку оптимізованих та береться наступний за випадковою послідовністю, ще не оптимізований вузел з результуючого маршруту ЗК. Після завершення виконання алгоритму локальної оптимізації отриманий оптимізований результуючий маршрут передається на обробку результатів, після чого відбувається процедура «нанесення феромону» та запускається якщо потрібно нова ітерація циклу пошуку маршрутів мурахами-агентами.

2.3. Нові методи та засоби для подолання труднощів розв'язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних

Застосування багатоагентної системи для розв'язання практичного завдання: пошуку оптимального маршруту (маршрутизація) обходження всіх

вузлів в КМ, потребує додавання до формулювання класичної ЗК наступних умов:

- динамічні зміни вхідних;

- можлива відсутність шляху з одноразовим проходженням усіх вузлів мережі – неможливість розв’язку ЗК;

- частково невідомі вхідні дані – при знаходженні в будь-якому вузлі мережі агент володіє лише поточною інформацією про вартість відправки до інших вузлів мережі, доступних з поточного вузла;

- вартість проходження по одному сполученню в різних напрямах може відрізнятись, тобто ЗК є асиметричною.

Фактично необхідним є розробити модель багатоагентної системи, здатну вирішувати таке практичне завдання, що зводиться до розв’язання динамічної асиметричної ЗК в умовах частково невідомих даних. Описані умови роблять неможливим гарантування розв’язання ЗК, не дають доступ до оцінки стану мережі в повному обсязі – інформація лише про вузли доступні з вузла знаходження мурахи-агента. В початковому вузлі до завершення ітерації циклу пошуку маршрутів мурахами-агентами та повернення агентів назад немає жодної інформації про процес обчислення ЗК. В цьому полягає складність обчислення в умовах частково невідомих динамічно змінюваних вхідних даних. Умова частково невідомих вхідних даних ЗК робить неможливим використання методів локальної оптимізації, тобто при розв’язанні ЗК за таких умов застосувати розроблений додатковий програмний модуль на базі методів локальної оптимізації буде неможливо. Можливість повторного проходження знайденого мурахою-агентом маршруту не гарантується. Крім того повернення мурахи-агента в початковий вузел може бути неможливим, тому було вирішено обмежити ***час існування мурахи-агента***, для цього було введено додаткові параметри: 1) час початку подорожі; 2) час існування агента; 3) кількість спроб (максимальна кількість допустимих переходів між вузлами).

З переходом до розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних, виникають наступні складності, що потребують розробки нових методів та засобів для їх подолання [25, 90, 91]:

1) Необхідність уникнення та виходу з ситуацій ***«пасток»*** та ***«зациклень»***;

2) Неможливість виконання звичайної процедури «нанесення феромону», яка виконується після завершення маршруту та повернення в початковий вузел усіх мурах-агентів.

3) Недоцільність повторного використання знайдених результуючих маршрутів після їх отримання, через постійні динамічні зміни вхідних даних та, відповідно, стану мережі, без можливості своєчасного виявлення цих змін.

4) Надмірне збільшення значень цифрових міток, розташованих на вузлах мережі – ***надмірне укріплення «пам’яті колонії мурах»,*** яке було виявлено в процесі аналізу та тестування розробленої моделі багатоагентної системи, актуальне для розв’язання ЗК динамічного характеру.

Для виявлення та виходу з ситуацій «пасток» та «зациклень» було запропоновано методи та розроблено на базі запропонованих методів засоби опрацювання критичних ситуацій. Складність пов’язана з тим, що частково невідомі вхідні дані та їх динамічна змінюваність роблять неможливим гарантувати розв’язання ЗК, тобто проходження через усі вузли мережі через їх недоступність або неможливість проходження однократно через структуру мережі [2]. Це все призводить до того, що мураха-агент, в процесі розв’язання ЗК, здатний потрапити в одну з двох критичних ситуацій.

Ситуація «пастки». Коли агент-комівояжер, потрапивши в поточний вузел мережі після опрацювання інформації виявляє, що даний вузел «від’єднано» від мережі, або ж за певних інших умов не існує жодного доступного вузла для подальшого переходу (наприклад вихідний трафік даних, крім службової інформації заборонений або обмежений налаштуваннями мережового екрану), тобто мураха-агент ніби потрапляє в «пастку» на поточному вузлі. Ситуацію "пастки" показано на рис.2.11, після переходу з вузла В до вузла А, виявляється що повернутись назад до вузла В або перейти

до наступного вузла є неможливим. Це означає, що мураха-агент потрапив у "пастку" на вузлі A.

Для розв’язання даної складності було розроблено наступний алгоритм:

A.1) Перевірка згідно значень доступностей з’днань для переходу у матриці A, чи є доступні для переходу вузли з поточного вузла i: A(i, j) = 1. Якщо доступні вузли є – продовження шляху, інакше – встановлення відмітки ***"Пастка"*** (**Trap**) на поточному вузлі для інших мурах-агентів. Даний стан на вузлі буде вставновлено до тих пір, поки агент не зможе вибратись з поточного вузла. Інші агенти що звертаються до вузла i, будуть отримувати значення доступності A(k, i) = 0, навіть якщо з’єднання для переходу є доступним.

A.2) Спроба пошуку доступних для переходу вузлів. Якщо вузли для переходу знайдено, продовження подорожі мурахи-агента та зняття відмітки **Trap**, після завершення успішного переходу до наступного вузла. Якщо доступних вузлів немає, то перехід до пункту A.3.

A.3) Очікування заданого інтервалу часу та перехід до пункту A.2, якщо визначений ліміт спроб або час існування мурахи-агента не вичерпано, інакше «смерть» мурахи-агента, тобто його самоліквідація та зняття відмітки **Trap** на поточному вузлі.

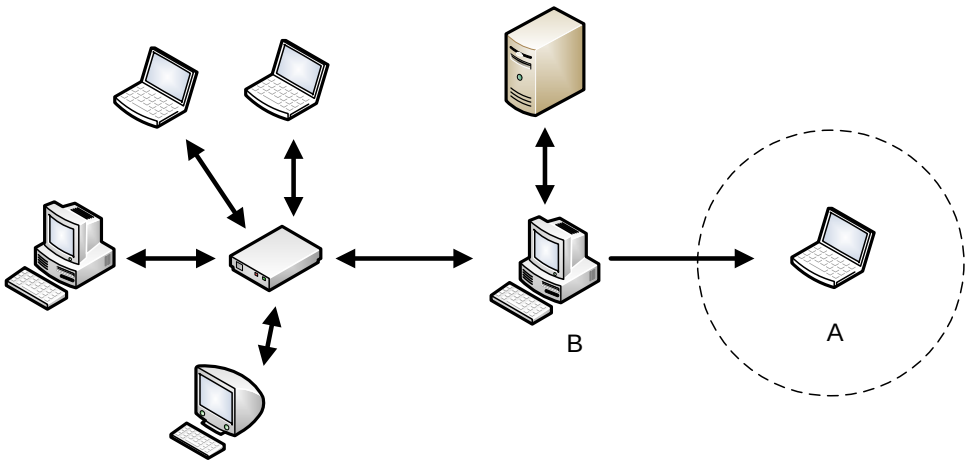


Рис.2.11. Ситуація "пастки" на вузлі A

Ситуація «зациклення». Ситуація потрапляння мурахи-агента в «петлю» – набір з декількох вузлів, які циклічно відвідуються по колу в процесі подорожі агента по мережі нескінченну кількість раз.

На рис.2.12 представлено комп'ютерну мережу з можливими для утворення ділянками ситуації "зациклення". Цикли можуть виникати у різних ситуаціях:

а) між двома двосторонньо сполученими вузлами, наприклад на рис.2.12 між M та N: M → N → M → N ..., між B та M: B → M → B →M ... та інші;

б) ланцюжкове зациклення, вузли A, C, D на рис.2.12, циклічні переходи по вузлам A→C→D→A нагадують ситуацію "пастки" з кількістю вузлів 3 до моменту появи нового сполучення чи підключення нового вузла;

в) відокремлена ділянка мережі – вузли E, F, G на рис.2.12, потрапивши в цю ділянку, агент може переходити циклічно по даним вузлам: E → F → G → E → G → F → E.

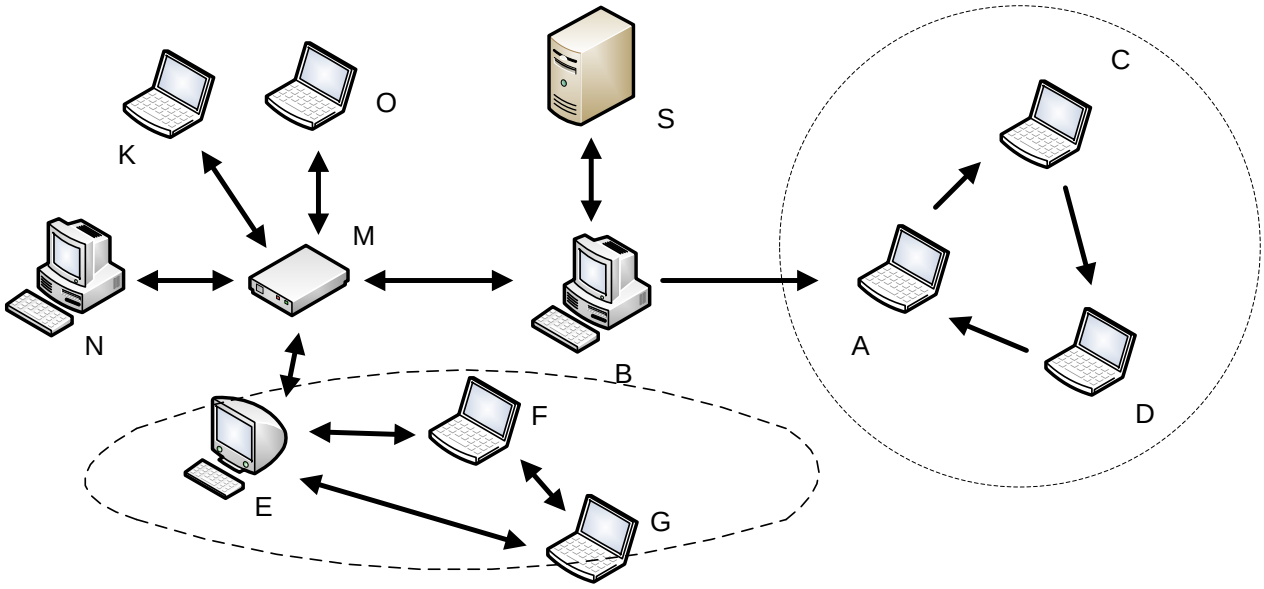


Рис.2.12. Комп'ютерна мережа з ділянками можливими для утворення ситуації "зациклення"

Пройдений шлях R – послідовність вузлів, які було пройдено, що зберігаються мурахою-агентом, тепер включає крім ідентифікатора вузла ще й час його відвідування та додаткові відмітки.

Для розв’язання даної задачі було розроблено наступний алгоритм:

Б.1) Обирається наступний вузел для подорожі лише серед тих вузлів, які не були відвідані до сих пір, тобто відсутні у списку табу. Якщо серед доступних для переходу вузлів немає ще не відвіданих, то перехід до пункту Б.2.

Б.2) Пошук доступного для переходу початкового вузла P(0) з метою завершення маршруту. Якщо його не знайдено, то перехід до пункту Б.3.

Б.3) Пошук наступного вузла для переходу P(i+1) серед відвідуваних, який відповідає наступній умові:

$$P(i+1) != P(i-1), \quad (2.2)$$

де P(i-1) попередній за маршрутом вузел.

Таким чином йде уникання короткого зациклення на двох вузлах. Перехід до пункту Б.4.

Б.4) Перевірка чи обраний наступний вузел в пункті Б.3 – P(i+1) не є частиною циклу: аналіз на спільні ділянки в пройденому маршруті з метою виявлення зациклень. При виявленні потрапляння агентом в такий цикл, продовження подорожі по ньому з метою пошуку будь-якого вузла Pj, який не належить до множини вузлів, що складають зациклену ділянку, тобто мураха-агент відправляється до обраного вузла P(i+1) та процедура виходу з циклу продовжується з кроку Б1.

Збільшення значень цифрових міток (процедура «нанесення феромону») проводиться для кожного відвіданого мурахою-агентом з'єднання лише один раз, незалежно від кількості його відвідування, це робиться з метою уникання додаткового збільшення значення міток на з'єднаннях (відповідно збільшення імовірності їх вибору мурахами-агентами), які використовуються вимушено багаторазово через структуру мережі чи розглянуті критичні ситуації.

Неможливість звичайного оновлення значень цифрових міток. В алгоритмі колонії мурах оновлення значень міток — «нанесення феромону», відбувалось після завершення ітерації циклу пошуку маршрутів мурахами-агентами та повернення в початковий вузел усіх агентів. Це давало змогу збільшувати значення міток згідно вартості отриманого кінцевого маршруту. В умовах частково невідомих динамічно змінюваних вхідних даних, повторне проходження маршруту та збільшення значень міток пропорційно до його підрахованої вартості може бути недоцільним або навіть неможливим, через

існуючу імовірність відключення вузла від мережі, чи недоступності деяких з’єднань.

Запропоновано новий метод для вирішення даної складності: значення цифрових міток збільшуються тепер в процесі проходження сполучення між вузлами на момент досягнення вузла призначення, ідея реалізації такого процесу збільшення значень міток була частково запозичена з природньої поведінки мурах, оскільки вони відкладають феромон (збільшують значення мітки) постійно, протягом усього свого руху.

Додатково для підкріплення кращих досягнутих результатів після повернення агентів відправляється визначена кількість нових агентів (аналог розглянутої існуючої модифікації з використанням «елітних» мурах), мета яких лише спроба проходження знайдених квазіоптимальних маршрутів найменшої вартості з метою додаткового збільшення значень цифрових міток на пройдених з’єднаннях. Повернення додаткових агентів є необов'язковим, проте при успішному проходженні шляху гарантується підкріплення лише найкращих поточно знайдених маршрутів.

Збільшення значення мітки відбувається тепер за наступним запронованим рівнянням:

$$\Delta M_{ij}^{k}(t) = \frac{(2 \times Q / N)}{C_{ij}^{k}(t)} \times ExpF, \qquad (2.3)$$

де $\mathbf{C^k(t)}$ – сумарна вартість маршруту на момент часу **t** для **k**-ого мурахи-агента.

ExpF – коефіцієнт, що визначає відношення прогнозованої кінцевої вартості поточного маршруту мурахи-агента до вартості знайденого колективом агентів квазі-оптимального результуючого маршруту.

Процедура «випаровування феромону», тобто зменшення значень цифрових міток відбувається згідно рівняння 1.8, де t – час початку поточної ітерації, а t+1 – час початку наступної ітерації (закінчення поточної ітерації), тепер це незалежні від циклу запуску мурах-агентів відліки часу. Тобто

зменшення значень міток відбувається за заданим періодом часу $\mathbf{T_{evp}}$, який було додано як ще один вхідний параметр.

Недоцільність повторного використання знайдених результуючих маршрутів після їх отримання, через постійні динамічні зміни вхідних даних та, відповідно, стану мережі, без можливості своєчасного виявлення цих змін.

В більшості розроблених систем для розв’язання ЗК, що базуються на використанні LKH, МЗП, генетичних, еволюційних, класичних (перелічених в розділі 1.1) та інших методів [59,68,73,106], застосовується наступний режим функціонування систем: спочатку обчислюється розв'язок ЗК згідно вхідних умов, після чого отриманий розв'язок — результуючий квазіоптимальний маршрут використовується для виконання практичного завдання. В межах КМ це буде розсилання даних. В випадку з динамічно змінюваними частково невідомими вхідними даними, в процесі проходження вузлів за знайденим маршрутом може виникнути ситуація, коли перехід за обчисленим маршрутом є неможливим або кінцева вартість після проходження не буде відповідати порахованій вартості обчисленого маршруту. При чому дана ситуація буде визначена лише при спробі використання знайденого машруту, в процесі його проходження.

Для вирішення даної задачі було запопоновано новий засіб організації функціонування системи в режимі **поєднання процесу пошуку маршрутів з процесом передачі даних**.

Процес обчислення та пошук оптимальних шляхів має поєднуватись з розсиланням даних по мережі, тобто в процесі передачі даних мурахами-агентами відбувається постійний пошук найкращих, мінімальних за вартістю шляхів з метою зменшення витрат.

В межах такого режиму функціонування системи, маршрути, що знаходяться, є актуальними на момент їх обчислення, а після повернення мурах-агентів до початкового вузла – завершення ітерації циклу пошуку маршрутів мурахами-агентами, зібрана інформація про мережу є доступною для

подальшого аналізу. Для передачі дані діляться на порції – блоки даних відповідно до кількості мурах-агентів – k, та розмірності даних – DataVolume.

Якщо деякі мурахи-агенти не повернулись за визначений відлік часу через розглянуті ситуації "пастки" або "зациклення", блоки даних які передавались ними, можна відправити повторно новими агентами, розпочавши нову ітерацію циклу пошуку маршрутів мурахами-агентами.

Виявлення та метод подолання негативних наслідків надмірного укріплення «пам'яті колонії мурах». В процесі дослідження (див. розділ 4.2) розробленої моделі багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних було виявлено ще одну складність — надмірне збільшення значень цифрових міток, розташованих на вузлах мережі.

Чим більше значення цифрової мітки на з'єднанні між вузлами, тим більше ймовірність його обирання для проходження мурахами-агентами. Після знаходження мурахами-агентами квазі-оптимального шляху в умовах, коли динамічні зміни відсутні, або не впливають на знайдений шлях протягом тривалого періоду, відбувається збільшення значень міток (процедура «нанесення феромону») на тих самих з'єднаннях, що входять до знайденого маршруту.

Як наслідок, постійно зростаючі значення цифрових міток на цих з'єднаннях, за умови їх доступності, не дозволяють вільно шукати інші альтернативні, можливо більш оптимальні, з'єднання та муршрути. Тобто сформована «пам'ять колонії мурах», в якій зберігається знайдений квазі-оптимальний шлях, продовжує надмірно укріплюватись.

Для вирішення даної задачі було розроблено новий метод, що базується на застосуванні адаптивної верхньої межі значення мітки Mmax — максимальне значення цифрової мітки. Значення цифрової мітки M(i,j) на з'єднаннях між вузлами i та j визначається як:

$$M(i,j) = T * V(i,j), \quad (2.4)$$

де T – час пересилання даних по мережі в процесі обчислення ЗК,

V(i,j) – швидкість накопичення значення цифрової мітки між вузлами і та j.

Припустимо, що час обчислення в умовах відсутності динамічних змін вхідних даних набуває надзвичайно великих значень T → ∞; тоді згідно з рівнянням 2.4 значення цифрової мітки необмежено збільшується M(i,j) → ∞.

Згідно розробленого методу подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток, що базується на застосуванні адаптивної верхньої межі цифрової мітки Mmax вираз (2.4) можна представити у вигляді:

$$M(i,j) = \min(T * V(i,j), Mmax). \quad (2.5)$$

Значення верхньої адаптивної межі цифрової мітки Mmax є динамічно змінюваним в процесі обчилення ЗК, встановлюється в залежності від наступних факторів: кількості агентів – k; швидкості накопичення значення цифрової мітки V(i,j) — різниця між значеннями цифрової мітки в результаті виконання процедури «нанесення феромону» та процедури «випаровування феромону» відносно часу обчислення T; інтенсивності динамічних змін вхідних даних – If. Тобто Mmax можна представити залежністю:

$$Mmax = Function(k, V(i,j), If). \quad (2.6)$$

Згідно з виразом (2.5) значення цифрової мітки M(i,j) тепер може набувати лише значення в межах від 0 до Mmax.

За допомогою застосування адаптивної верхньої межі значення цифрової мітки, збільшення значень міток на з’єднаннях між вузлами відбувається до певної межі, залишаючі можливість для пошуку альтернативних більш оптимальних маршрутів мурахами-агентами, навіть коли динамічних змін вхідних даних тривалий час не відбувається.

А коли виникають динамічни зміни вхідних даних, при яких зберігається доступність використовуваних з’єднань вже знайденого квазі-оптимального шляху, система здатна швидко адаптуватись до нових умов, що було практично неможливо до цього.

2.4. Висновки до розділу

У цьому розділі проведено вдосконалення базового методу шляхом розробки нової модифікації алгоритму колонії мурах для розв’язання динамічної ЗК та запропоновано нові методи та засоби для розв’язання динамічної асиметричної ЗК в умовах частково невідомих даних. Отримано такі результати:

- удосконалено метод розв’язання задачі комівояжера, який базується на використанні поведінкової моделі колонії мурах, шляхом зміни початкової установки значень міток та імовірнісного вибору наступного вузла для переходу мурахи-агента, що дозволило зменшити кількість ітерацій циклу пошуку маршрутів мурахами-агентами та відповідно зменшити час обчислення ЗК;

- запропоновано новий метод опрацювання результуючого маршруту при розв’язанні динамічної задачі комівояжера, який базується на використанні алгоритмів локальної оптимізації 2-opt, 2.5-opt, 3-opt в залежності від інтенсивності динамічних змін вхідних даних, що дозволило зменшити вартість результуючого маршруту;

- запропоновано новий метод подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток (пам’яті колонії мурах), який базується на використанні адаптивної верхньої межі значення цифрової мітки, що дозволило відновити пошук маршрутів меншої вартості навіть після тривалого статичного стану вхідних даних;

- запропоновано новий метод виявлення та виходу мурахи-агента з критичних ситуацій «пастки» та «зациклення», що дозволило отримати розв’язок ЗК навіть за умови виникнення критичних ситуацій;

- запропоновано новий метод оновлення значень цифрових міток в процесі проходження сполучення між вузлами, що дозволило забезпечити ефективне оновлення пам’яті колонії мурах в умовах частково невідомих вхідних даних;

- запропоновано новий засіб організації функціонування системи в режимі поєднання процесу пошуку маршрутів з процесом передачі даних, що дозволило забезпечити розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних.

3. БАГАТОАГЕНТНІ СИСТЕМИ З ВИКОРИСТАННЯМ ПОВЕДІНКОВОЇ МОДЕЛІ КОЛОНІЇ МУРАХ

3.1. Моделі багатоагентних систем

При дослідженні базового алгоритму колонії мурах виявилось, що отримані результати мають недостатньо високу точність (до 12% різниці з оптимальним розв'язком отриманим точними методами для ЗК до 10000 вузлів представлені у розділі 4.1), що також підтверджується іншими дослідженнями [49,57,78]).

З метою підвищення точності отримуваних результатів розв'язання ЗК, було вирішено застосувати додатково методи локальної оптимізації як доповнення до запропонованої модифікації базового алгоритму (див. розділ 2.1). Ідею та програмну реалізацію алгоритмів локальної оптимізації було розглянуто у розділі 2.2. На базі методів локальної оптимізації було створено програмний модуль, який можливо використовувати при розв'язанні динамічної ЗК.

В роботі було розроблено дві моделі багатоагентних систем на базі алгоритму колонії мурах (з врахуванням запропонованої модифікації базового алгоритму):

1) модель багатоагентної системи з використанням технологій паралельного обчислення та методів локальної оптимізації для розв'язання динамічної ЗК з кількістю пунктів обходу 100 і більше;

2) модель багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних, наближених до умов в реальній КМ.

При обчисленні ЗК вхідні та вихідні дані мають суттєвий об'єм, тому було вирішено використовувати збереження та читання через файлову систему, оскільки застосування баз даних потребує встановлення додаткового програмного забезпечення та призводить до високих часових витрат при виконанні операцій читання та запису даних.

Самоорганізація, як процес і як результат, не може розглядатись як властивість будь-якої централізованої системи. Багатоагентна система – це децентралізована система, функціональні можливості якої вище за суму можливостей окремих сутностей (агентів), що входять до її складу [5,10,87,96,98]. Багатоагентна система складається з колективу автономних агентів, які взаємодіючи між собою функціонують в середовищі, за допомогою чого реалізовується виконання поставленого завдання. Мурахи-агенти є незалежними, без наявної прямої взаємодії між собою, напряму комунікація між ними відсутня. У жодного з агентів немає уявлення про всю систему.

Застосування багатоагентних систем є гнучким рішенням, оскільки система може бути доповнена й модифікована без переписування значної частини програми. Багатоагентні системи мають здатність до самовідновлення (поторного запуску агентів), мають стійкість до збоїв завдяки достатньому запасу компонентів і самоорганізації. Узагальнену структуру моделі багатоагентної системи для розв’язання ЗК, що базується на використанні алгоритму колонії мурах представлено на рис.3.1.

Як видно з рис.3.1, кожен з k агентів взаємодіє з середовищем, з якого агенти отримують дані про кількість вузлів, доступні з’єднання для переходу між ними, вартість переходу по з’єднанню, а також значення цифрових міток, які забезпечують явище стігмергії, що було розглянуто в розділі 1.2, непряму комунікацію між агентами, яка досягається шляхом накопичення колективного досвіду (пам’яті колонії мурах) в процесі руху агентів по мережі.

Середовище включає в себе наступні дані: матрицю доступностей (матриця A), матрицю вартостей (матриця C), пам’ять колонії мурах (матриця M) та вхідні параметри ЗК (див. розділ 1.3). Кожен мураха-агент включає в себе список пройдених вузлів (taboo список); ідентифікатор; блок читання та виводу даних для взаємодії з середовищем; блок прийняття рішень, в якому визначається, згідно отриманих даних, наступний вузел для переходу агента.

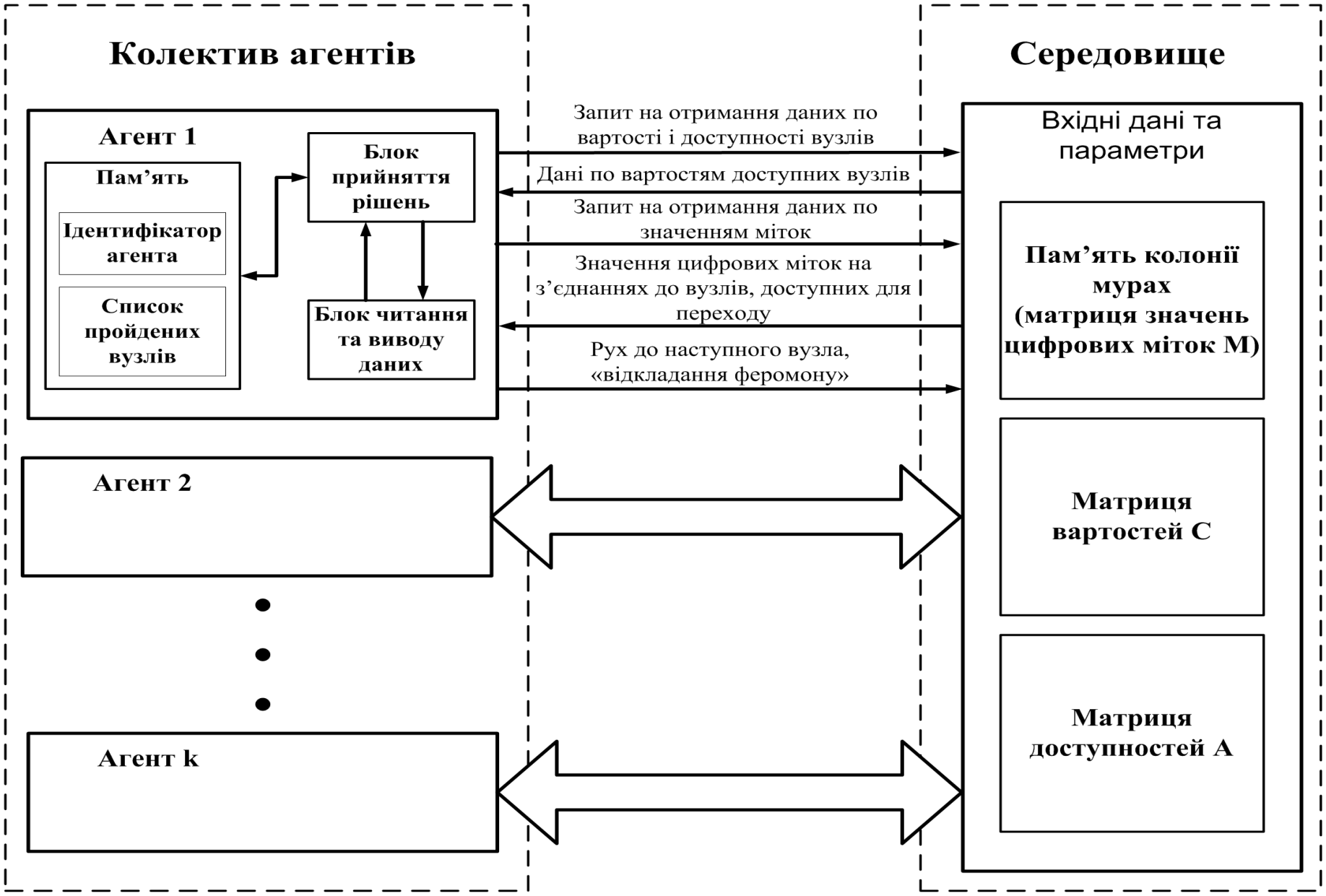


Рис.3.1. Узагальнена структура моделі багатоагентної системи для розв'язання ЗК

Мета агента – знаходження маршруту обходження одноразово усіх N вузлів, що входять до мережі, сформованої на базі вхідних даних ЗК. В кожен момент часу агент володіє інформацією про поточний вузел, в якому він знаходиться, для переходу до наступного вузла, агенту необхідно виконати запит до середовища з метою отримання даних про доступні вузли. Після чого агент приймає рішення про виконання переходу до наступного вузла, та передає ці дані в середовище, в якому відповідно до задіяних агентами з'єднань, відбувається процедура збільшення значень міток («нанесення феромону») та зменшення з часом значень міток («випаровування феромону»). Таким чином мурахи-агенти впливають на середовище в процесі свого руху по мережі.

В середовищі можуть відбуватись зміни, оскільки вхідні умови ЗК є динамічно змінюваними, тобто кількість вузлів, вартість або доступність з'єднань між вузлами може змінитись. При розв'язанні динамічної ЗК без умови частково невідомих даних є можливість аналізу вхідних даних та змін на потребу повторного запуску ітерації циклу пошуку шляхів мурахами-агентами, залежно від етапу на якому знаходиться кожен агент. При втраті агента, є можливість запустити поторно агентів. Мінімальна кількість агентів для функціонування системи – 2, як і для будь-якої класичної багатоагентної системи. При розв'язанні динамічної асиметричної ЗК в умовах частково невідомих даних виникає потреба додаткового застосування запропонованих нових методів та засобів, які було розглянуто в розділі 2.3.

3.2. Багатоагентна система розв'язання динамічної ЗК

При виникненні змін вхідних даних відбувається запуск наступної ітерації циклу пошуку машрутів мурахами-агентами без потреби перезапуску системи. При такій реалізації система буде здатна розв'язувати динамічну ЗК з інтенсивністю динамічних змін вхідних даних менше ніж час виконання однієї ітерації циклу пошуку маршрутів: виходу агентів з початкового вузла, проходження всіх вузлів та повернення в початковий вузел.

На рис.3.2 показано структурну схему розробленої моделі багатоагентної системи, яка складається з двох підсистем:

1) ***підсистема графічного відображення результатів*** – генератор координат при відсутності файлу з координатами вузлів, та можливістю графічного відображення результуючих маршрутів у вигляді графу, вершини якого – вузли, ребра – з'єднання між вузлами;

2) ***підсистема обчислення результуючого маршруту ЗК*** – основний інтерфейс та взаємопов'язані програмні модулі, за допомогою яких відбувається обчислення вхідної ЗК та отримання результуючих маршрутів.

В ***підсистемі графічного відображення результатів*** генерується послідовність координат в обраній координатній сітці (від 0 до 900 по координатам **x** і **y**) задана кількість точок, відповідно до кількості вузлів, перша обирається як початкова та відповідно позначається на графіку (маркери * та + чорного кольору).

Відповідно до кожного вузла генеруються нові або використовуються зчитані з файлу координати. Отримані вузли відображаються за визначеними для них координатами, позначаються на графіку (маркером o). Згідно вхідного маршруту отриманого з файлу від підсистеми обчислення результуючого маршруту ЗК, будується граф за визначеною послідовністю вузлів, згідно якої проводяться з'єднання – ребра графу.

В ***підсистемі обчислення результуючого машруту ЗК*** реалізовано процедури, розглянуті в розробленій граф-схемі алгоритму колонії мурах з врахуванням запропонованої нової модифікації базового алгоритму (рис.2.1, схема праворуч), та розроблених блок-схем алгоритмів локальної оптимізації (рис.2.2, 2.5, 2.8). Код програмної реалізації розробленої багатоагентної системи представлений в Додатку Ж.

Програмний інтерфейс має наступний функціональний набір:

1) читання матриці вартостей C та матриці доступностей A з файлів та вхідних параметрів, перевірка вхідних даних на коректність;

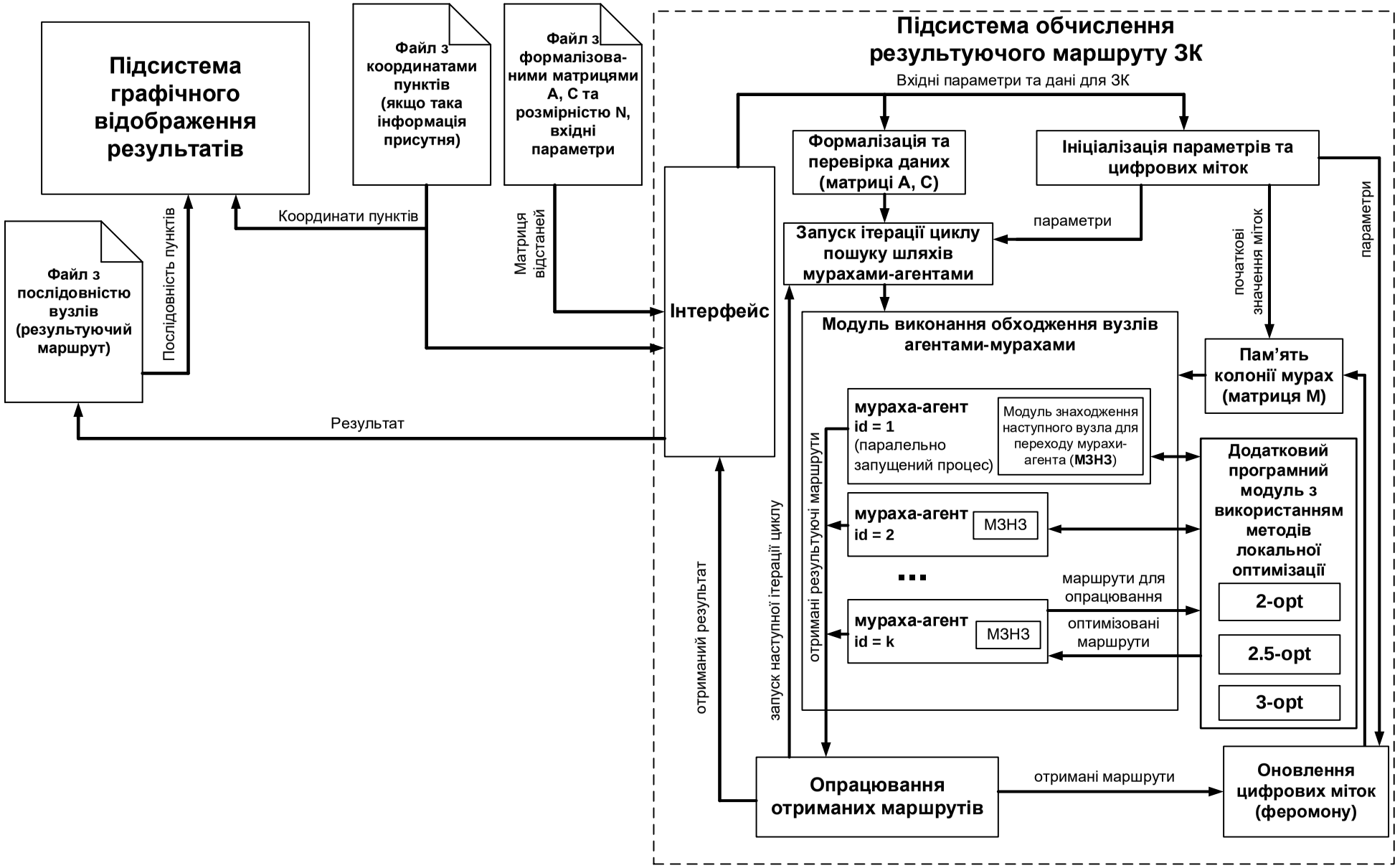


Рис.3.2. Структура розробленої моделі багатоагентної системи для розв'язання динамічної ЗК

2) зчитування з стандартизованого вхідного файлу даних по ЗК в різній формі представлення (матриця вартостей С або набір координат);

3) запис проміжних результатів, вхідних параметрів та результуючого маршруту разом з показниками часу до файлу результатів.

Після опрацювання вхідних даних та ініалізації початкових параметрів та системи міток відбувається запуск циклу пошуку шляхів мурахами-агентами. Протягом кожної ітерації циклу виконується процес обходження вузлів мережі мурахами – агентами, кількістю k. Кожен агент – незалежний потік виконання (thread), який має доступ до спільної пам'яті — даних об'єкта “середовища”. При визначенні вузла для переміщення агент використовує модуль знаходження наступного вузла для переходу мурахи-агента, який є частиною агента (див. рис.3.2). Крім того агент локально зберігає в адресному просторі потоку, в якому він запущений, маршрут (список пройдених вузлів) та свій ідентифікатор. Для прийняття рішення щодо обирання вузла для переходу мурахою-агентом використовується процедура знаходження наступного вузла.

Після закінчення кожної ітерації циклу пошуку шляхів мурахами-агентами отримані результати в різних потоках передаються на опрацювання, після чого запускається оновлення значень цифрових міток. Також, в залежності від встановлених вхідних параметрів та інтенсивності динамічних змін вхідних даних ЗК, агенти передають отримані маршрути на додатковий програмний модуль з використанням методів локальної оптимізації, з метою їх покращення за точністю, тобто наближеністю за вартістю до оптимального маршруту шляхом перестановки з’єднань між вузлами. Виконання даної процедури також відбувається паралельно, як і пошук маршрутів мурахами-агентами, у незалежних потоках.

В додатковому програмному модулі [27] включено усі 3-и, розглянуті в розділі 2.2, алгоритми локальної оптимізації. Отже система здатна функціонувати у 4-х режимах:

1) без використання методів локальної оптимізації;

2) з використанням 2-opt алгоритму локальної оптимізації;

3) з використанням 2.5-opt алгоритму локальної оптимізації;

4) з використанням 3-opt алгоритму локальної оптимізації.

Для визначення режиму функціонування було введено додатковий вхідний параметр ***opt***, який вставновлюється відповідно до інтенсивності динамічних змін вхідних даних. Оптимізований за допомогою методів локальної оптимізації маршрут використовується для подальшого опрацювання – виклику процедури оновлення значень цифрових міток та збереження проміжних результатів.

Після чого відбувається запуск наступної ітерації циклу пошуку маршрутів мурахами-агентами. Після закінчення даного циклу зібрані результати записані в файл передаються до підсистеми відображення та доступні для подальшого аналізу.

3.3. Багатоагентна система розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних

Розроблений засіб, що базується на об’єднанні процесу обчислення з виконанням практичного завдання – передачі даних в мережі (див. розділ 2.3), було використано при розробці моделі багатоагентної системи для розв’язання динамічної асиметричної ЗК в умовах частково невідомих даних, структурну схему якої представлено на рис.3.3. Код програмної реалізації розробленої багатоагентної системи представлений в Додатку З.

Розроблена багатоагентна система включає в себе симуляцію динамічного невідомого середовища – аналог реальної КМ, та віртуальні вузли (надалі просто вузли, оскільки вони є аналогами вузлів в реальній КМ), з встановленим програмним забезпеченням, що забезпечує обчислення ЗК та передачу даних – агентів-пакетів, які є вмістилищем даних.

Програмний модуль, який було названо «Сканер» (див. структурну схему на рис.3.3), встановлено на кожному вузлі КМ. Даний модуль дозволяє виконувати звернення до об’єкту динамічного невідомого середовища, з метою

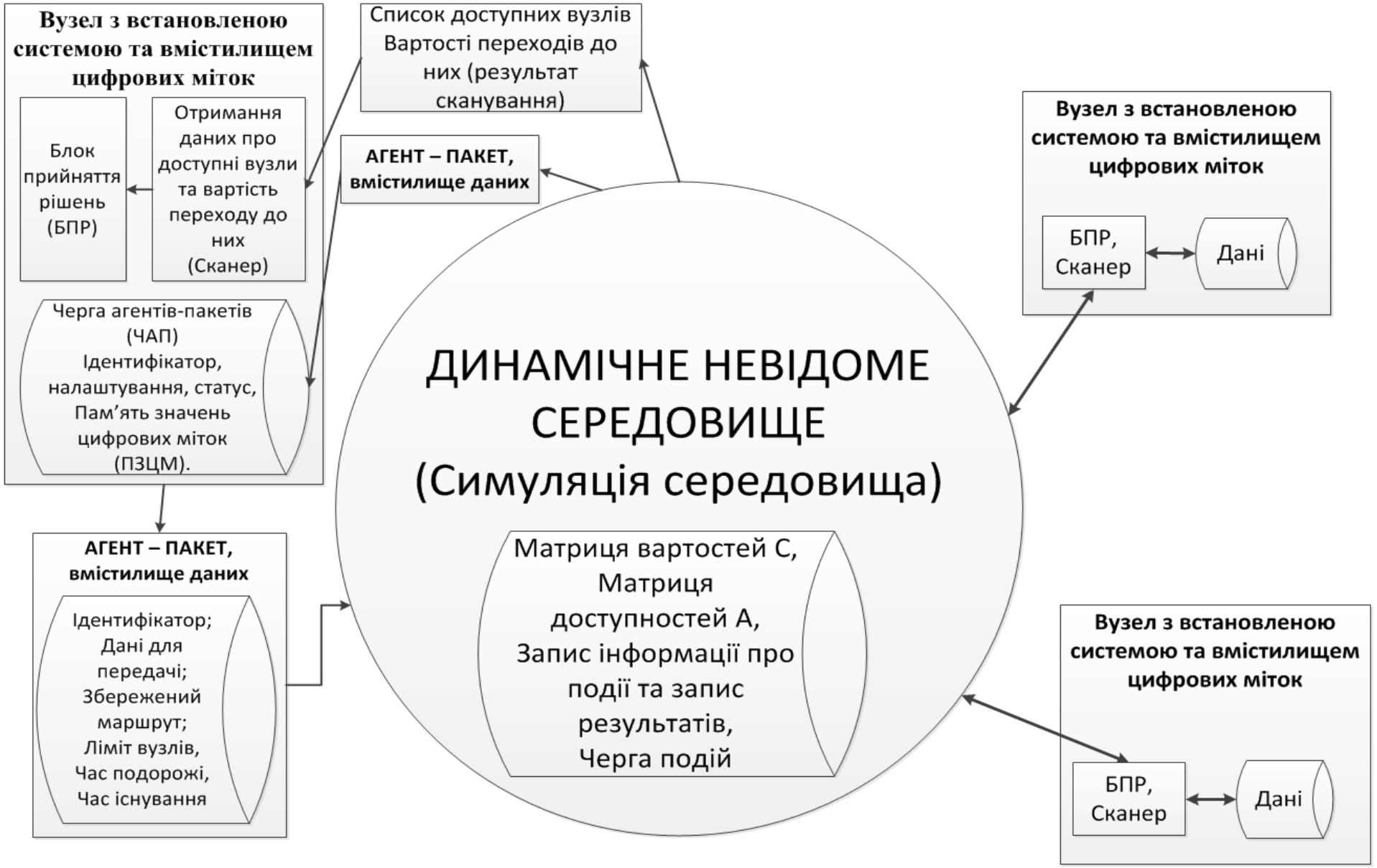


Рис. 3.3. Структура розробленої моделі багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних

отримання доступних для переходу вузлів, включаючи вартість переходу агента до них. Пам’ять колонії мурах тепер не входить до даних, які включено в об’єкт середовища.

В процесі розробки системи було додано отримання статусів доступних вузлів. Кожен статус відповідає певній логіці по обробці та взаємодії з вузлом. Статус може приймати наступні значення:

1) «нормальний» – активний доступний вузел;

2) «пастка» – статус пастки на вузлі, більш детально було розглянуто в розділі 2.3;

3) «відключений» – вузел від’єднано від мережі, опрацювання даних та агентів-пакетів неможливе.

Список доступних вузлів після отримання з середовища передається на блок прийняття рішень, що працює на базі розглянутої в другому розділі процедури mD (див. рис.2.1), з врахуванням розроблених алгоритмів опрацювання ситуацій «пастки» та «зациклення».

Кожен вузел крім статусу, зберігає наступні дані:

1) ідентифікатор – унікальний ключ користувача, який користується вузлом;

2) набір параметрів, що відносяться до алгоритму колонії мурах з врахуванням запропонованої модифікації базового алгоритму (E, L, Q, PSlim, P, T_{evp});

3) черга агентів-пакетів – агенти, що були успішно прийняті на вузлі, опрацьовані та готові для подальшої передачі. Черга працює за принципом FIFO (first input first output) – перший отримано, перший відіслано.

Якщо агент немає доступних для переходу вузлів після аналізу в блоку прийнятті рішень, або передачу до наступного вузла було перервано, агент потрапляє знову в цю чергу в кінець. Тобто його буде знову опрацьовано через визначений момент часу, з повторною спробою передачі до наступних вузлів.

4) пам’ять значень цифрових міток на з’єднаннях з поточного вузла до доступних для переходу вузлів – M(i,j), фактично рядок з матриці значень

цифрових міток. Кожен запис в цій пам’яті включає в себе наступну інформацію: ідентифікатор вузла для переходу; значення цифрової мітки; час останньої зміни або створення запису;

Після успішного завершення переходу агента до наступного вузла ***j*** з поточного вузла ***i*** відбувається збільшення значення цифрової мітки на пройденому з’єднанні M(i, j) в пам’яті значень цифрових міток згідно рівняння 2.3:

$$\Delta M_{ij}^{k}(t) = \frac{(2 \times Q / N)}{C_{ij}^{k}(t)} \times ExpF.$$

У відповідності з визначеним періодом T_{evp} та константи P (див. розділ 2.3) запускається зменшення значень цифрових міток (процедура «випаровування феромону») згідно рівняння 1.8:

$$M_{ij}(t + T_{EVP}) = M_{ij}(t) \times (1 - P).$$

В якості мурахи-агента використовується пакет даних – *агент-пакет*, який зображено на рис.3.4. Цифрами на рисунку позначено кількість бітів, за якими можна визначити розмір кожного поля пакета даних.

Крім користувацьких даних пакет включає в себе наступні поля:

- ідентифікатор агента (64 біта – 8 байт);
- ідентифікатор початкового вузла – вузла ініціатора розсилання агентів (32 біти – 4 байти);
- лічильник пройдених вузлів мережі (20 бітів та 12 бітів зарезервованих);
- лічильник часу (64 біта – 8 байт);
- максимальна кількість кроків (кількість пересилань) та час існування агента, які обмежують час існування агента, а також зібрані в процесі шляху агента по мережі службові дані (кожне поле по 32 біти – 4 байти);
- послідовність пройдених вузлів мережі, включаючи час їх відвідування (4 байти ідентифікатор вузла та 4 байти час відвідування, тобто кожен запис по 8 байт), список "табу" — список відвіданих (або заборонених

для відвідування) агентом вузлів мережі формується шляхом вибірки з послідовності пройдених вузлів унікальних ідентифікаторів вузлів.

Поле «розмір користувацьких даних» має розмір 24 біти, тобто максимально допустимий розмір даних для передачі в пакеті 15 МБ.

Поле «розмір зібраних службових даних» має розмір 20 біт, тобто максимально допустимий розмір службових даних 1023 КБ, враховуючи розмір кожного запису (8 байт), це складає 130944 записів про відповідну кількість пройдених вузлів.

Рис.3.4. Пакет даних – мураха-агент в розробленій системі

В рамках розробленої багатоагентної системи було встановлено обмеження, з можливістю розширення розмірності в майбутньому, в 65535 вузлів, тобто для значення N було виділено 2 байти.

Якщо в процесі передачі агента до наступного вузла виник збій – з'єднання стало не доступним, відключився вузел, відбувається повторне сканування середовища з метою подальшої передачі агента з поточного вузла.

Об'єкт динамічного невідомого середовища включає в себе матрицю доступностей A, матрицю вартостей C, та чергу подій. На рис. 3.5 відображено структуру матриці вартостей C, розмірністю N x N, згідно кількості вузлів в вхідній мережі ЗК. Аналогічну структуру має матриця доступностей A, зображена на рис. 3.6.

Кожна комірка матриці C містить в собі 4-х байтове значення вартості відповідного з'єднання. Кожна комірка матриці A містить в собі 1-о байтове значення статусу доступності відповідного з'єднання.

Пам'ять колонії мурах в багатоагентній системі, що було розглянуто в розділі 3.2, для розв'язання динамічної ЗК має вигляд матриці цифрових міток M. Структура пам'яті колонії мурах – матриці значень цифрових міток M розмірності N x N зображено на рис.3.7. Кожна комірка матриці M містить в собі 4-х байтове значення цифрової мітки відповідного з'єднання. Матриця M зберігається в підсистемі обчислення результату (див. рис.3.2).

При розробці моделі багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних (див. рис.3.3) матрицю M було розподілено між вузлами. В кожному вузлі зберігається один рядок матриці значень цифрових міток.

Структуру пам'яті значень цифрових міток розмірністю N, що зберігаються на i -му вузлі представлено на рис.3.8. Згідно запропонованих нових засобів (див. розділ 2.3) для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних, до кожної комірки пам'яті колонії мурах було додано поле «час останньої зміни», яке визначає час встановлення початкового значення чи час останньої зміни значення цифрової мітки.

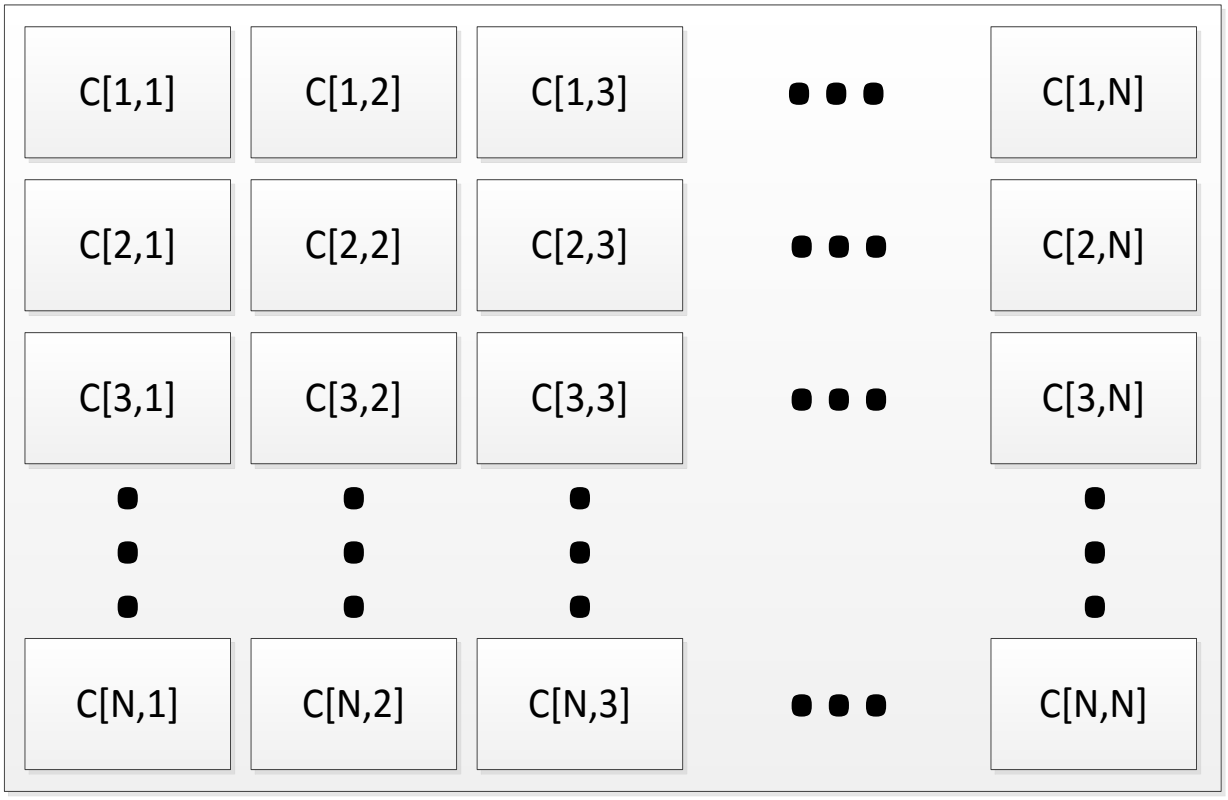


Рис.3.5. Структура матриці вартостей C розмірності N x N

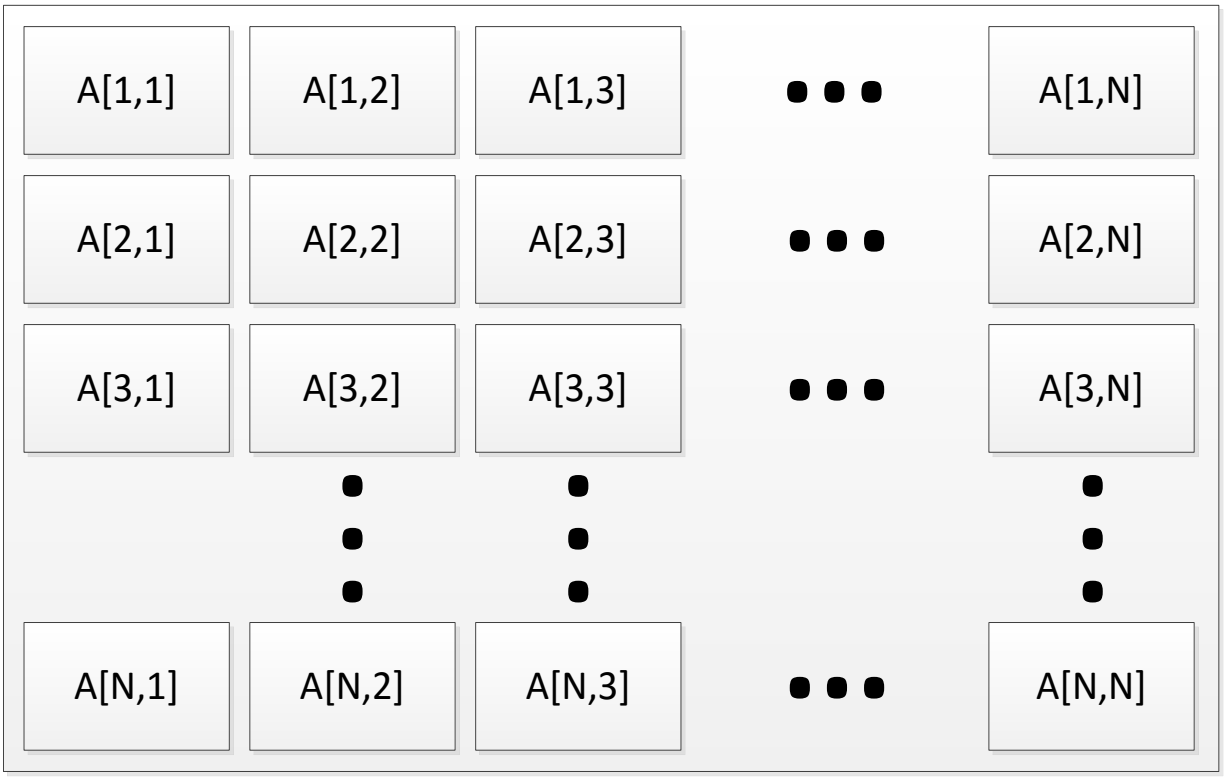


Рис.3.6. Структура матриці доступностей A розмірності N x N

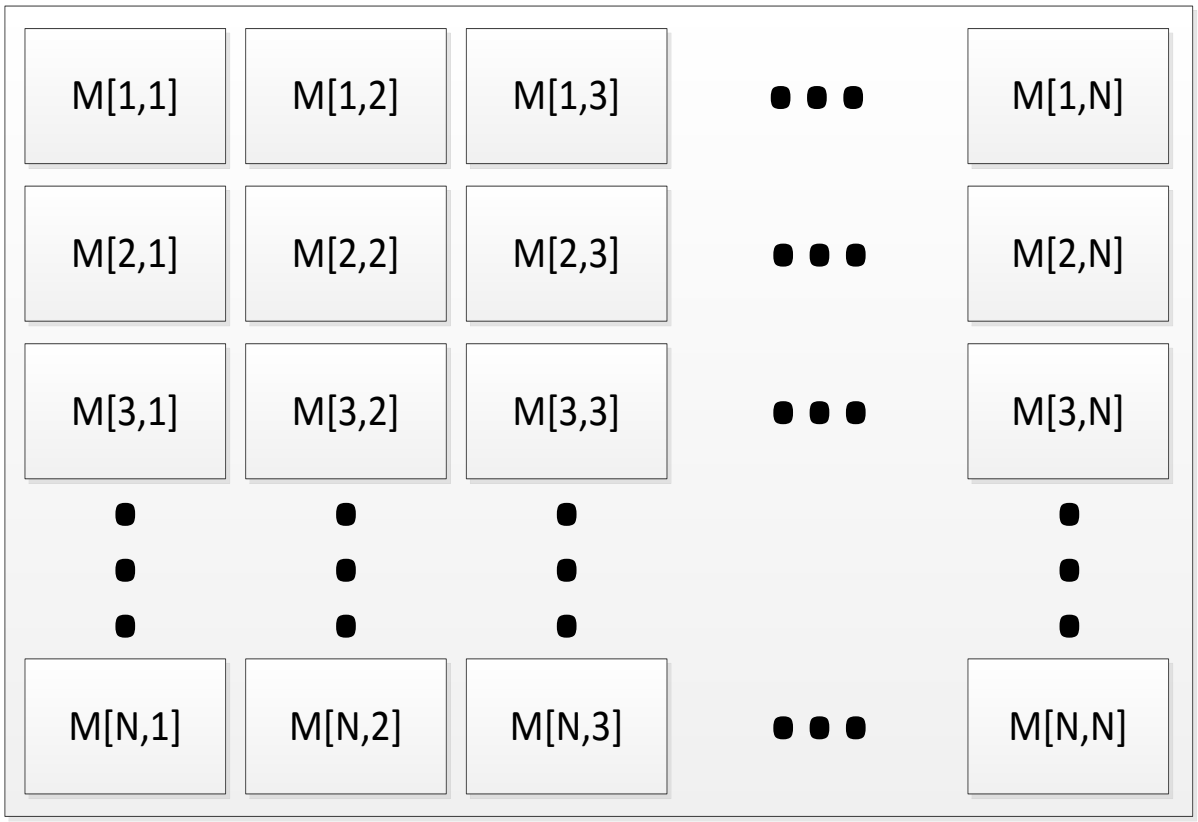


Рис.3.7. Структура пам’яті колонії мурах – матриці значень цифрових міток M розмірності N x N

Час останньої зміни	Значення цифрової мітки
T_1	M[i,1]
T_2	M[i,2]
⋮	⋮
T_N	M[i,N]

Рис.3.8. Структура пам’яті значень цифрових міток розмірністю N, що зберігаються на вузлі *i*

Функціонування середовища комп'ютерної мережі симулюється за допомогою черги подій. Кожна подія включає в себе час виконання. Події можуть бути наступних видів:

1) зміна статуса вузла;

2) зміна доступності чи вартості з'єднання між вузлами;

3) спроба передачі агента з одного вузла до іншого;

4) закінчення симуляції та збір даних;

5) запуск спеціальних агентів (аналог застосування «елітних мурах»);

Події впорядковуються за часом виконання та зберігаються у черзі подій. Кожна подія зберігається у вигляді струтури даних з 5 полів, вигляд якої представлено на рис.3.9.

Зверху позначено розмірність кожного поля в байтах. Кожна подія включає в себе наступні поля: час виконання події (4 байти); тип події (1 байт), типи подій було описано вище; ідентифікатор вузла 1 (4 байти), через який визначається вузел-джерело при передачі, або вузел над яким виконується подія; ідентифікатор вузла 2 (4 байти), що використовується при потребі як вузел призначення для визначення події передачі; значення (4 байти) (вартість, новий статус вузла, значення доступності, і т.д.);

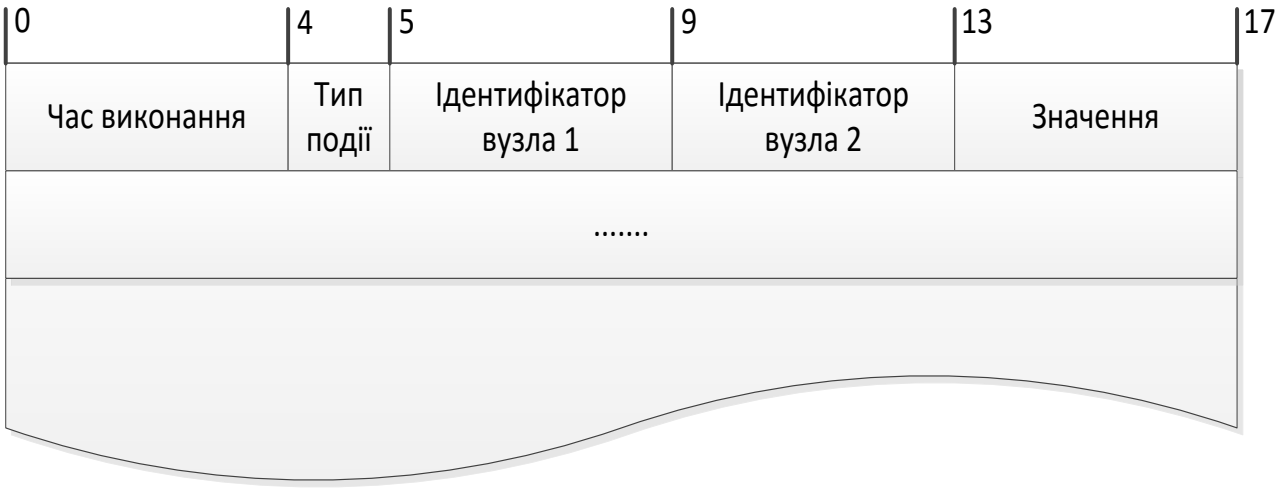


Рис.3.9. Структура даних, що зберігається в черзі подій

Після завершення передачі агента відбувається опрацювання даних, яке також потребує деякий час. Перехід мурахи-агента з одного вузла до іншого

відбувається як в реальній комп'ютерній мережі згідно вартості з'єднання (вартість з'єднання з вузла А до вузла В – час передачі агента з вузла А до вузла В).

3.4. Оцінка об'єму споживання пам'яті при застосуванні запропонованих моделей багатоагентних систем

В процесі розробки багатоагентних систем було визначено типи даних, та обсяг одного з основних ресурсів – пам'яті, необхідної для зберігання вхідних даних та параметрів, з метою забезпечення необхідних ресурсів та обчислення можливих обмежень при розв'язанні ЗК великої розмірності. Представимо методику розрахунку об'єму споживання пам'яті та опис використовуваних даних для кожної системи окремо.

Підрахунок об'єму споживання пам'яті ***багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних***:

1) N – розмірність вхідних даних по ЗК, тобто кількість вузлів, що необхідно відвідати, зчитується з вхідного файлу з поля розмірності, необхідно 2 байти для зберігання.
2) Матриця вартостей C (4 байти кожне значення вартості) розмірністю N × N, тобто сумарно об'єм необхідної пам'яті для зберігання – 4 × N × N байт;
3) Матриця доступностей A (1 байт кожне значення «доступності») розмірністю N × N, тобто сумарно об'єм необхідної пам'яті для зберігання – N × N байт;
4) Мапа значень цифрових міток що розташовані на вузлах мережі – локальна пам'яті колонії мурах, які представляють собою фактично рядок матриці M. Ключем мапи є ідентифікатор вузла (4 байт) доступного для переходу, за ключем мапи зберігається значення цифрової мітки (4 байт), що відноситься до з'єднання між вузлами, та часу останньої зміни (4 байт) чи створення даного запису в мапі. Тобто максимальний розмір, який може набувати така мапа на кожному з N

вузлів – (4 + 4 + 4) × N = 12 × N байт, оскільки таких локальних мап буде відповідно до кількості існуючих вузлів, то загалом на зберігання пам’яті колонії мурах необхідно 12 × N × N байт;

5) Статуси вузлів: 4 × N байт;
6) Вхідні параметри, що зчитуються з файлу:

   k – кількість агентів (мурах) (4 байти);

   kA – кількість спеціальних агентів (аналогія використання «елітних мурах») (4 байти);

   E – коефіцієнт значимості досвіду пам’яті колонії мурах (2 байти); L – коефіцієнт евристики, коефіцієнт значимості вартостей переміщень (2 байти);

   Коефіцієнти E та L мають значення в межах від 0 до 10, дослідження їх впливу на процес обчислення ЗК представлено у розділі 4 (див. табл.4.5).

   Q – константа, має значення в межах від 10 до 10^{24}, саме значення визначається пропорційно до очікуваної сумарної вартості маршруту згідно значень вхідної матриці вартостей C (4 байти);

   P – константа, має значення в межах від 0 до 1, визначає швидкість зменшення («випаровування феромону») значень міток (4 байти);

   PSlim – нижня імовірнісна межа (4 байти);

   Tevp – періодичність виконання зменшення значень міток (4 байти);

   Загалом на зберігання вхідних параметрів необхідно 28 байт;
7) Кожен агент-пакет (див. рис. 3.4) для зберігання потребує наступний об’єм (при максимально допустимому N): заголовок пакету – 5 слів по 8 байт, 15 МБ користувацьких даних та 130944 слів по 8 байт службових даних; Відповідно до кількості агентів – k, зберігається:

   k × (130944 × 8 + 5 × 8 + 1966080 × 8) = 16776232 × k байт; (кожен агент приблизно по 16 МБ);

Для підрахунку необхідного обсягу для збереження даних по пунктам 1 – 6, використаємо підстановку максимально допустимо значення N = 65535:

Volume = 2 + 4 × N × N + N × N + 12 × N × N + 4 × N + 28 = 73012477995 байт (**приблизно 68 ГБ**), це без врахування збереження агентів-мурах (кожен по майже 16 МБ) та черги подій (кожна подія потребує 17 байт).

Підрахунок об’єму споживання пам’яті ***багатоагентної системи для розв’язання статичної та динамічної ЗК з використанням методів локальної оптимізації*** відбувається аналогічно описаним значенням до попередньої системи з наступними відмінностями:

1) Мапи значень цифрових міток немає, використовується загальна матриця значень цифрових міток (4 байти кожне значення) M, без зберігання часу останньої зміни чи створення запису. Відповідно розмірність зберігаємих даних менша – 4 × N × N байт;
2) Відсутні статуси вузлів;
3) Вхідні параметри відрізняються:

   k – кількість мурах-агентів (4 байти);

   opt – режим використання додаткового програмного модуля на базі методів локальної оптимізації (1 байт); Значення параметра opt визначає тип локальної оптимізації з 3-х реалізованих, тобто встановлює один з 4-х режимів функціонування системи: 1) opt = 0 – без локальної оптимізації; 2) opt = 1 – застосування 2-opt оптимізації; 3) opt = 2 – застосування 2,5-opt оптимізації; 4) opt = 3 – застосування 3-opt оптимізації;

   E, L, Q – аналогічно до описаних значень до попередньої багатоагентної системи (2, 2 та 4 байти відповідно);

   P – константа, має значення в межах від 0 до 1, визначає швидкість зменшення («випаровування феромону») значень міток (4 байти);

   PSlim – нижня імовірнісна межа (4 байти);

   Загалом на зберігання вхідних параметрів необхідно 21 байт;
4) Кожен мураха-агент потребує масив розмірністю N для збереження пройденого шляху R, ідентифікатор кожного вузла займає 4 байти, отже

для кожного мурахи необхідно 4 × N байт, а сумарно для всіх мурах потрібно 4 × N × k байт.

5) замість матриці вартостей C, може бути визначений файл з координатами (2 значення по 4 байти) вузлів-пунктів, розмір такої матриці буде 8 × N байти.

Для підрахунку необхідного обсягу для збереження даних, використаємо підстановку максимально допустимо значення N = 65535:

Volume1 = 2 + 4 × N × N + N × N + 4 × N × N + 21 = 9 × N × N + 23 = 38653526048 байт (**приблизно 36 ГБ**);

При умові використання координатів замість вартостей та без використання матриці доступностей (всі з'єднання між вузлами є доступними) необхідний об'єм буде значно меншим:

Volume2 = 2 + 4 × N × N + 8 × N + 21 = 4 × N × N + 8 × N + 23 = 17179869203 байт (**приблизно 16 ГБ**);

Крім того для збереження результуючих маршрутів відповідно до кількості агентів, необхідно 262140 байт (приблизно 256 КБ) на кожного агента.

В обох випадках витрати на збереження даних мають характер $O(N^2)$ відповідно до розмірності ЗК – N та констант o1, o2, o3, що відображено наступним узагальненим виразом:

$$Volume = o1 \times N \times N + o2 \times N + o3 \rightarrow (o1 + 1) \times N \times N;$$

Після розробки багатоагентних систем з використанням поведінкової моделі колонії мурах, залишилось проаналізувати отримані результати обчислень ЗК, дослідити доцільність використання розробленої модифікації, перспективність використання методів локальної оптимізації та розроблених методів та засобів для розв'язання динамічної асиметричної ЗК в умовах частково невідомих даних. Також у зв'язку з великою кількістю вхідних параметрів, необхідно забезпечити знаходження значень за замовчуванням, що будуть підходити для різних умов ЗК, з метою спрощення застосування розроблених багатоагентних систем з використанням поведінкової моделі колонії мурах для розв'язання динамічної ЗК.

3.5. Висновки до розділу

У цьому розділі виконано розробку моделей багатоагентних систем з використанням поведінкової моделі колонії мурах для розв’язання динамічної ЗК, проведено обчислення ресурсних затрат та представлено структурні схеми розроблених моделей систем з описом їх функціонування. Отримано такі результати:

- розроблено модель багатоагентної системи з використанням поведінкової моделі колонії мурах та технологій паралельних обчислень для розв’язання динамічної ЗК з кількістю вузлів до 65536;

- розроблено додатковий програмний модуль на базі методів локальної оптимізації, застосування якого дозволило зменшити вартість отримуваних результатів, зберігаючи можливість розв’язання динамічної ЗК;

- розроблено модель багатоагентної системи, яка базується на використанні поведінкової моделі колонії мурах при розміщенні цифрових міток на комунікаційних вузлах, що дозволило розв’язати динамічну асиметричну задачу комівояжера в умовах частково невідомих вхідних даних;

- за запропонованою методикою проведено розрахунок об’єму споживання пам’яті для збереження даних, необхідних для розроблених багатоагентних систем. Було виявлено, що при кількості вузлів N = 65535 для розроблених систем необхідно від 16 до 68 ГБ пам’яті.

4. РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ БАГАТОАГЕНТНИХ СИСТЕМ З ВИКОРИСТАННЯМ ПОВЕДІНКОВОЇ МОДЕЛІ КОЛОНІЇ МУРАХ

4.1. Результати досліджень багатоагентної системи для розв'язання динамічної ЗК

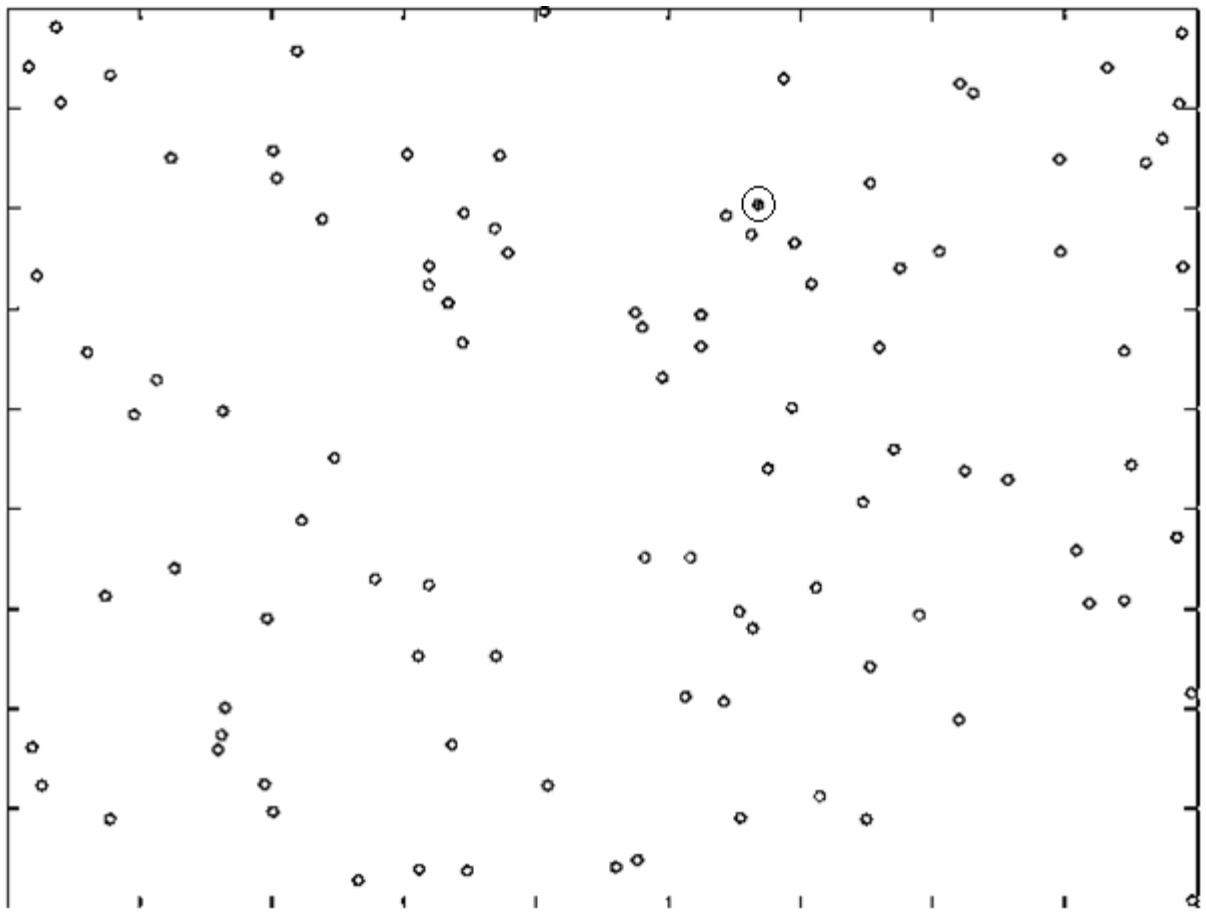
Для зручності демонстрації працездатності розробленої багатоагентної системи та доцільності використання запропонованої модифікації базового алгоритму колонії мурах було сформовано випадковим чином вхідні мережі для ЗК на 100, 500, 1000 та 10000 вузлів. Тестування розробленої модифікації проводилось ще до застосування технологій паралельних обчислень та імплементації додаткового програмного модуля з використанням методів локальної оптимізації. Зображення даних мереж у вигляді графу за допомогою підсистеми графічного представлення показано на рис. 4.1.а, рис. 4.2.а, рис. 4.3.а та рис. 4.4.а. Для демонстрації обрано двомірну площину, на якій кружечками відображаються згенеровані випадковим чином вузли. Початковий вузел позначається додатковими маркерами та є обведеним колом.

Дані задані мережі ЗК знаходяться в вихідних файлах l_1.txt (розмір 38,9 КБ) для ЗК на 100 вузлів, l_2.txt (розмір 970 КБ) для ЗК на 500 вузлів, l_3.txt (розмір 3,78 МБ) для ЗК на 1000 вузлів, l_4.txt (розмір 377,86 МБ) для ЗК на 10000 вузлів, після перетворення їх з матриці координат до матриці вартостей C. Дані файли мають знаходитись в директорії вказаній як джерело для розробленої системи. Отриманий файл з результатом відповідно буде зберігатись в файлі res1.txt, res2.txt, res3.txt, res4.txt.

Розмірність вхідних даних для ЗК, як видно з обсягу файлів, є великою, тому графічне відображення є більш зручним для сприйняття людиною. Фрагмент числового представлення (два перші рядки матриці відстаней) вмісту файлу l_1.txt для ЗК на 100 вузлів (рис. 4.1) має наступний вигляд:

0 261 769 370 127 26 701 571 550 195 644 469 255 333 170 495 681 295 330 297 709 587 786 264 668 537 620 712 204 223 424 772 471 536 265 197 263 613 356 661 408 467 251 364 309 334 794 363 506 195 327 529 483 568 593 144 151 232 421 30 446 556 454 559 516 407 313 454 200 459 384 88 363 427 269 48 381 740 385 142 492 497 124 148 187 118 670 630 308 587 256 313 519 233 602 202 689 487 474 87 ;

а)



б)

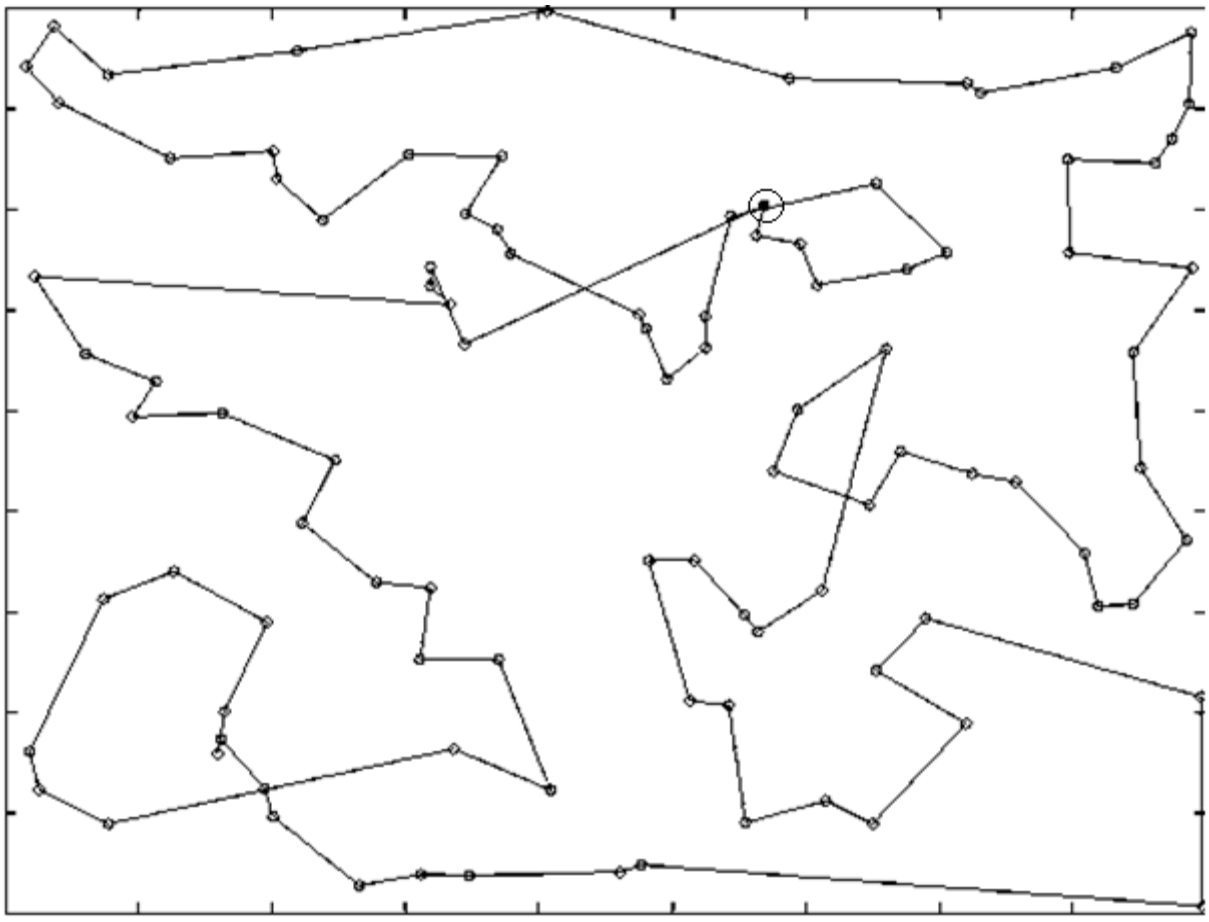


Рис. 4.1. ЗК на 100 вузлів: а) граф вхідної мережі, б) результуючий маршрут

261 0 844 178 338 235 586 342 297 68 450 507 23 479 346 455 491 555 104 557 540 460 586 316 477 333 629 585 301 75 422 551 593 591 388 454 63 582 337 596 187 254 286 156 446 596 580 317 319 449 571 267 613 373 591 387 166 493 557 249 232 355 201 381 258 402 530 300 75 619 561 289 622 496 130 280 254 598 421 158 374 472 356 215 199 208 598 395 393 706 19 573 371 479 509 139 515 226 297 349 ;

В ході проведених досліджень та тестувань було отримано квазі-оптимальний маршрут для ЗК на 100 вузлів (рис. 4.1.б), довжиною – 8311, отриманий за наступних параметрів:

N = 100, L = 2, E = 2, P = 0.5, Q = 1000, k = 6.

Даний результат був досягнутий:

1) ***без врахування розробленої модифікації базового алгоритму*** за час 8,6 с після виконання 19 ітерацій циклу пошуку маршрутів мурахами-агентами.

2) ***з врахуванням розробленої модифікації базового алгоритму*** за час 0,9 с після виконання 3-х ітерацій циклу пошуку маршрутів мурахами-агентами. Отриманий результуючий квазі-оптимальний маршрут відрізняється від оптимального на 7,3 % , має наступний вигляд:

0 59 75 71 82 55 99 5 85 83 84 56 79 9 68 29 95 74 18 43 3 60 25 53 63 48 76 42 4 49 35 19 72 45 91 17 57 93 50 66 70 69 52 32 58 13 44 34 88 23 28 14 78 30 65 38 47 15 81 37 54 26 33 11 73 89 2 39 86 6 27 77 20 96 24 16 10 61 7 87 31 46 22 21 94 80 92 67 98 41 40 62 64 97 51 8 12 1 90 36 0;

На рис. 4.2.б аналогічно зображено отриманий результуючий маршрут для ЗК на 500 вузлів, який має довжину 18342, отриманий за тих самих параметрів як і для ЗК на 100 вузлів, тільки при Q = 10000.

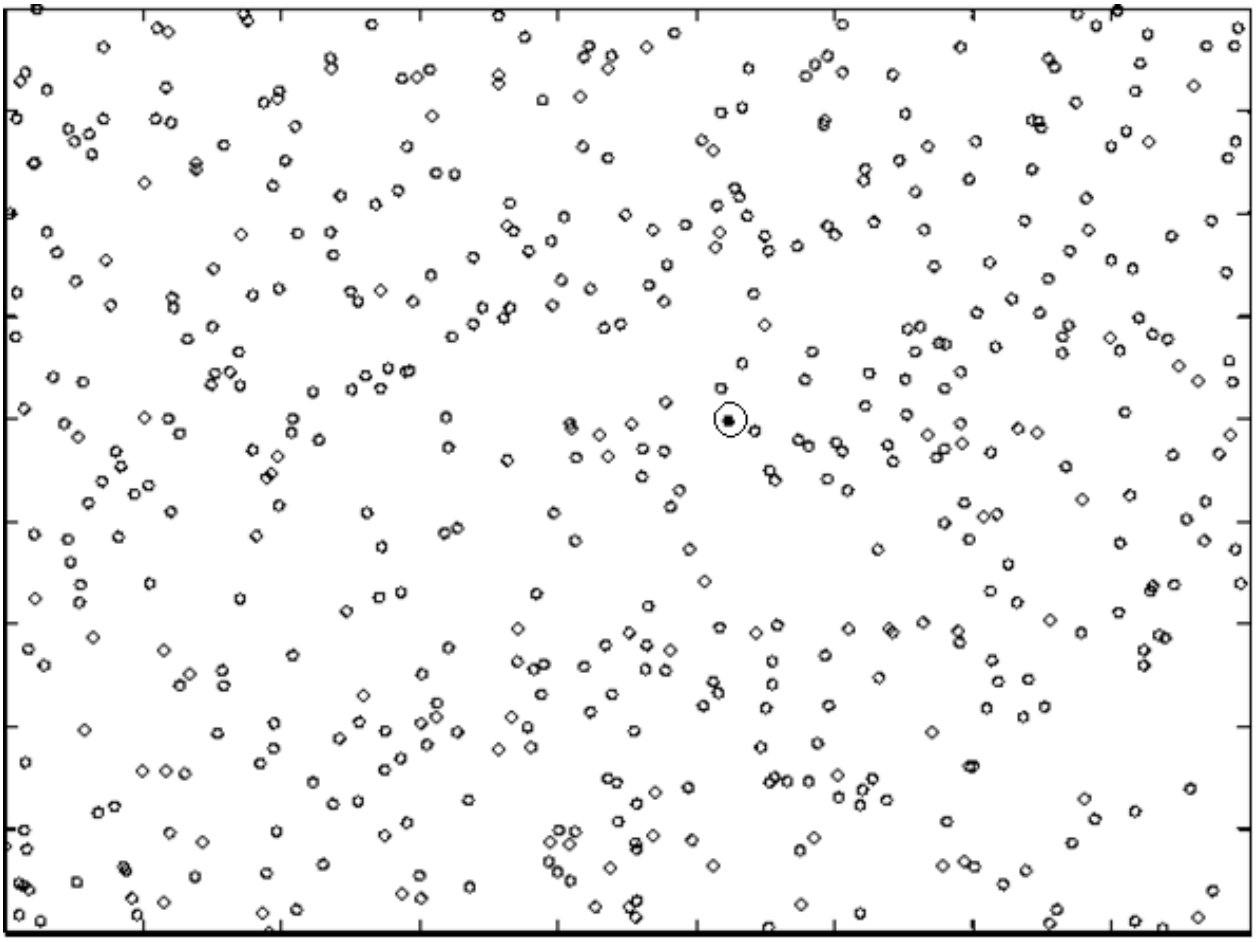
Даний результат, що відрізняється на 9,1% від оптимального, був досягнутий:

1) ***без врахування розробленої модифікації базового алгоритму*** за час 27,7 с після виконання 32 ітерацій циклу пошуку маршрутів мурахами-агентами;

2) ***з врахуванням розробленої модифікації базового алгоритму*** за час 4,2 с після виконання 7-и ітерацій циклу пошуку маршрутів мурахами-агентами.

Результуючий маршрут отриманий для ЗК на 1000 вузлів (рис. 4.3.б) має довжину 24922, отриманий при аналогічних параметрах як і для ЗК на 500 вузлів.

а)



б)

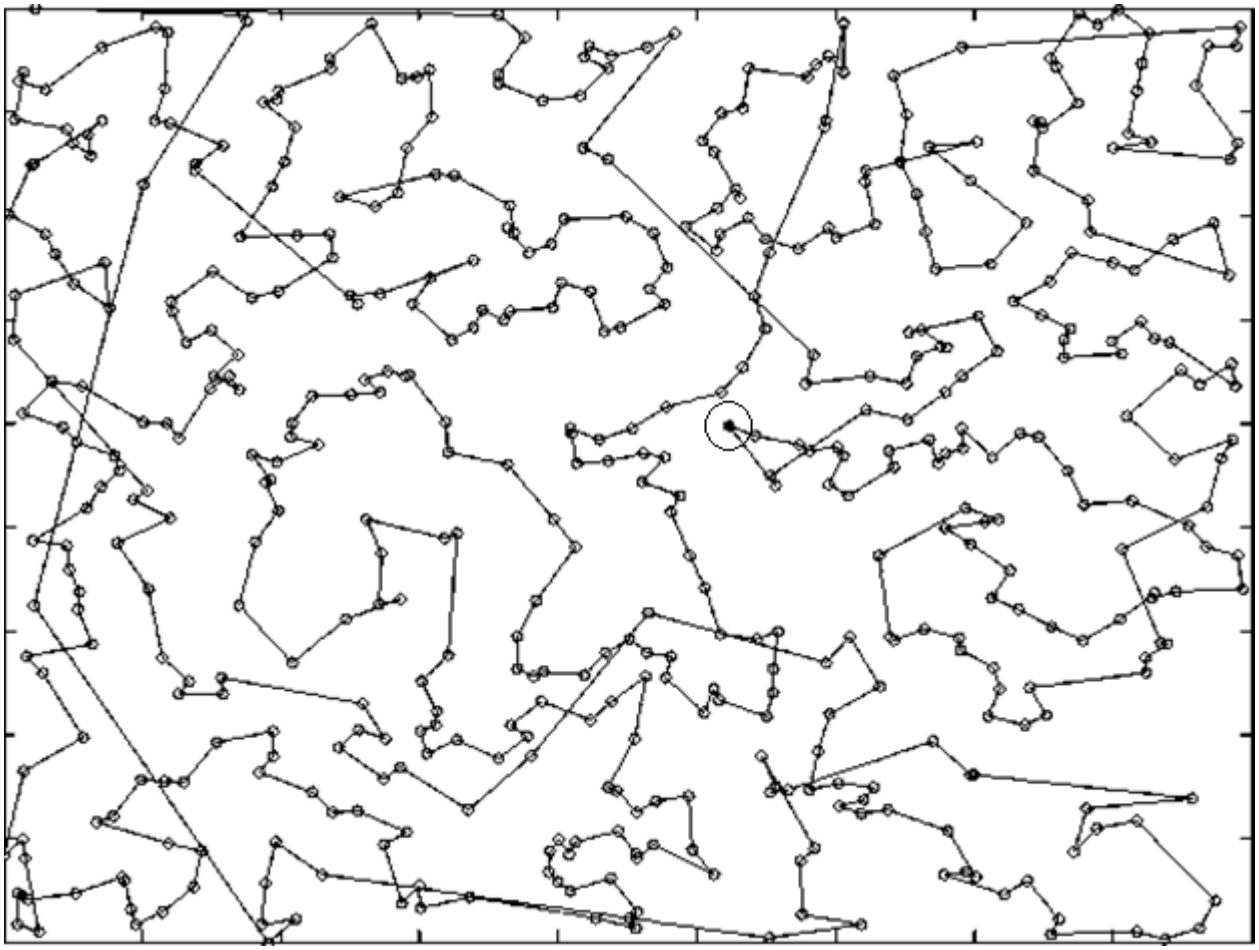
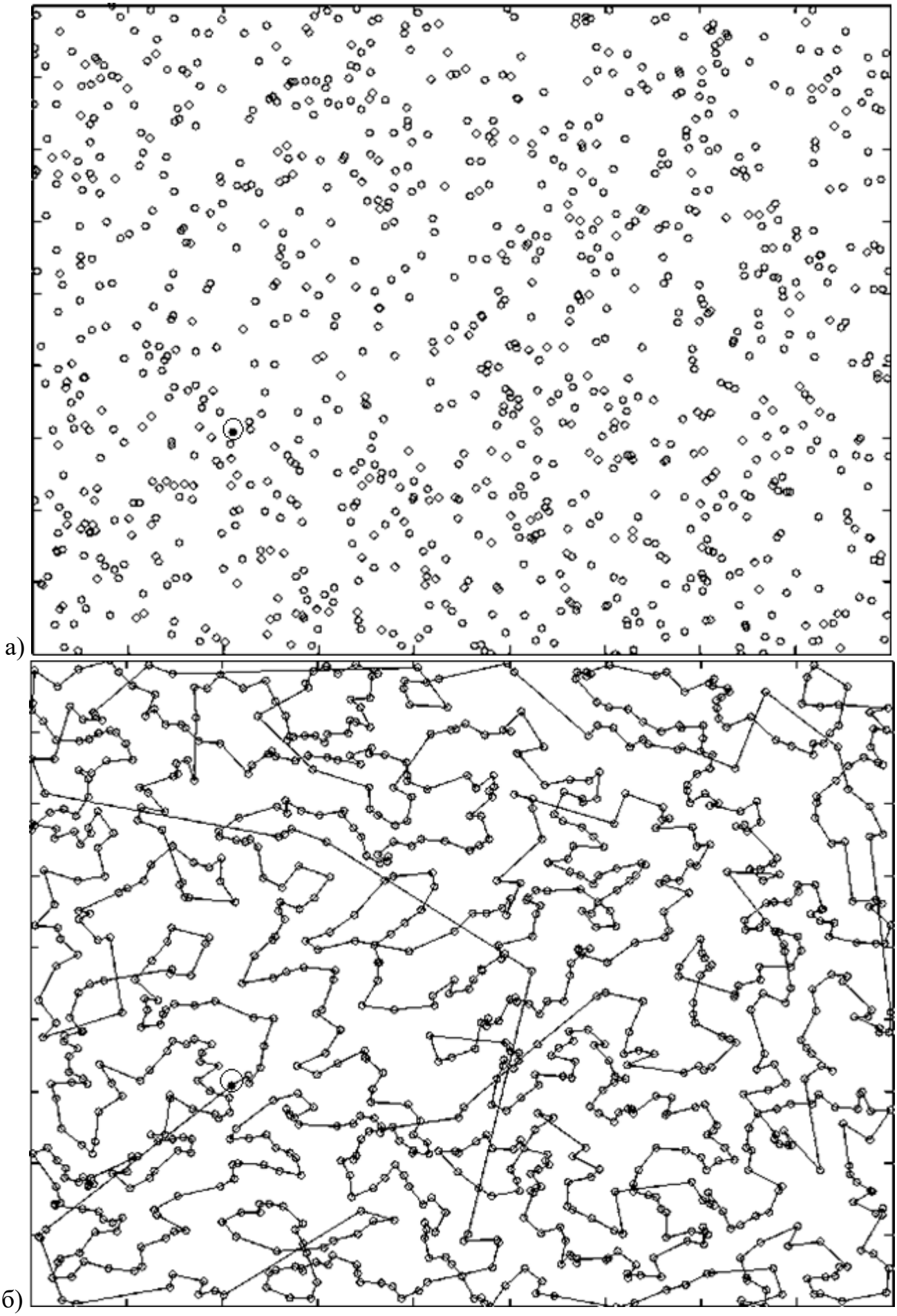


Рис. 4.2. ЗК на 500 вузлів: а) граф вхідної мережі, б) результуючий маршрут



а)

б)

Рис. 4.3. ЗК на 1000 вузлів: а) граф вхідної мережі, б) результуючий маршрут

Даний результат, що відрізняється на 11,5% від оптимального був досягнутий:

1) ***без врахування розробленої модифікації базового алгоритму*** за час 51,8 с після виконання 24-х ітерацій циклу пошуку маршрутів агентами***;***

2) ***з врахуванням розробленої модифікації базового алгоритму*** за час 11,1 с після виконання 14-и ітерацій циклу пошуку маршрутів мурахами-агентами.

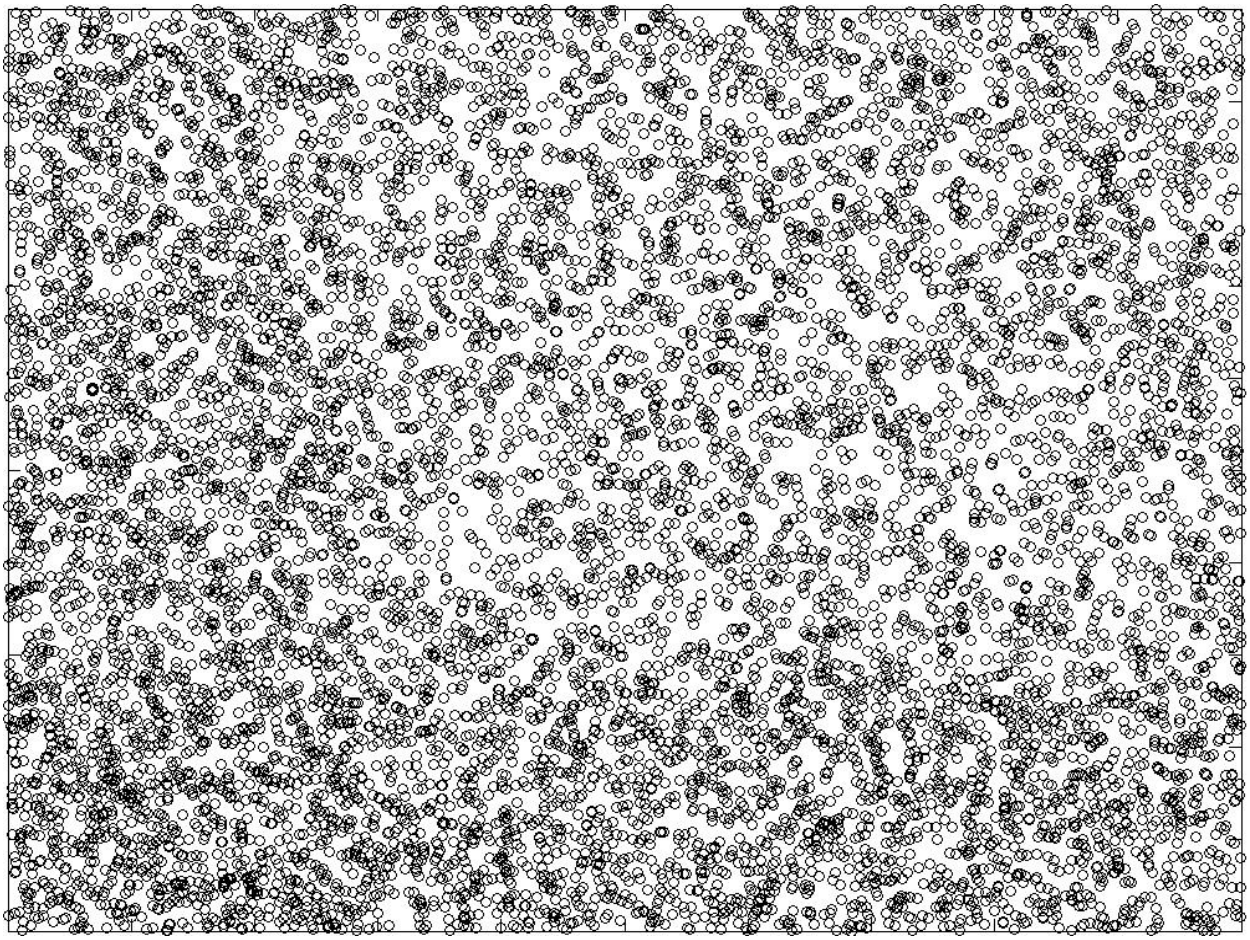
Результуючий маршрут отриманий для ЗК на 10000 вузлів (рис. 4.4.б) має довжину 84133, отриманий за аналогічних параметрів як і для ЗК на 500 вузлів. Даний результат був досягнутий:

1) ***без врахування розробленої модифікації базового алгоритму*** за час 3827 с (менше 2-х годин) після виконання 87 ітерацій циклу пошуку маршрутів мурахами-агентами; 2) ***з врахуванням розробленої модифікації базового алгоритму*** за час 2986 с після виконання 58 ітерацій циклу пошуку маршрутів мурахами-агентами.

Отримані результати розв’язання згенерованих ЗК на 100, 500, 1000 та 10000 вузлів продемонстрували те, що застосування розробленої модифікації базового алгоритму дозволяє зменшити час отримання результуючого маршруту майже у 4 рази для ЗК розмірністю до 1000 вузлів та на 20% для ЗК розмірністю до 10000 вузлів, зменшення часу обчислення відповідає зменшенню кількості ітерацій циклу пошуку шляхів мурахами-агентами.

Згідно проведених тестувань час виконання, в загальному випадку, зростає згідно складності алгоритму за закономірністю $O(N^2)$. Залежність часу виконання від розмірності N, для ЗК до 1000 вузлів зображено на рис. 4.5, кількість агентів k = 7, на 8-и ядерній машині, без використання розпаралелення при виконанні обчислень (задіяне лише 1 ядро), кількість виконаних ітерацій циклу пошуку маршрутів мурахами-агентами – 3. На рис. 4.6 зображено аналогічну залежність часу виконання з застосуванням технологій паралельного обчислення, при запуску на ЕОМ з 8-ма ядрами, тобто обчислення кожним з 7-ми агентів результуючого маршруту виконується паралельно, незалежно від

а)



б)

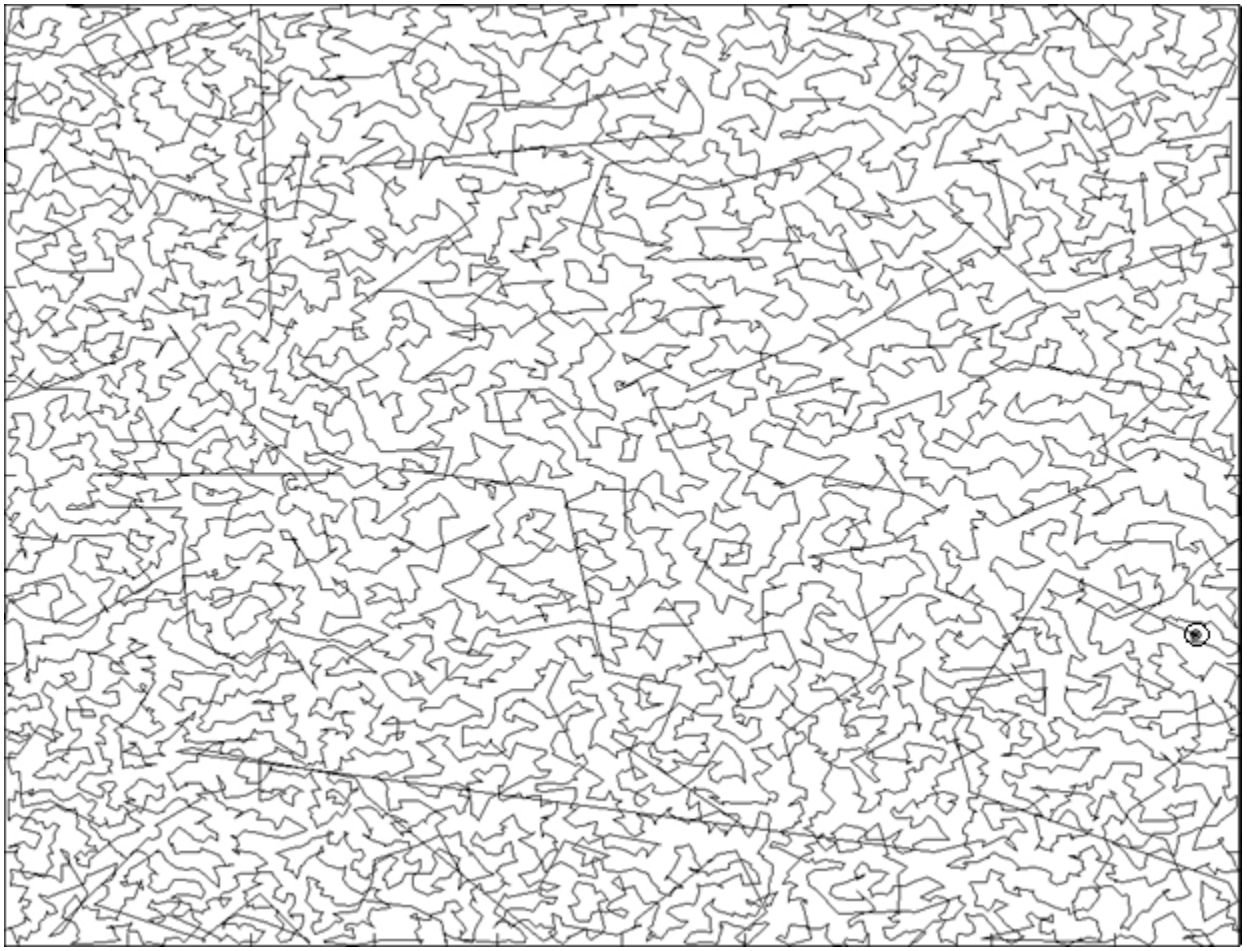


Рис. 4.4. ЗК на 10000 вузлів: а) граф вхідної мережі, б) результуючий маршрут

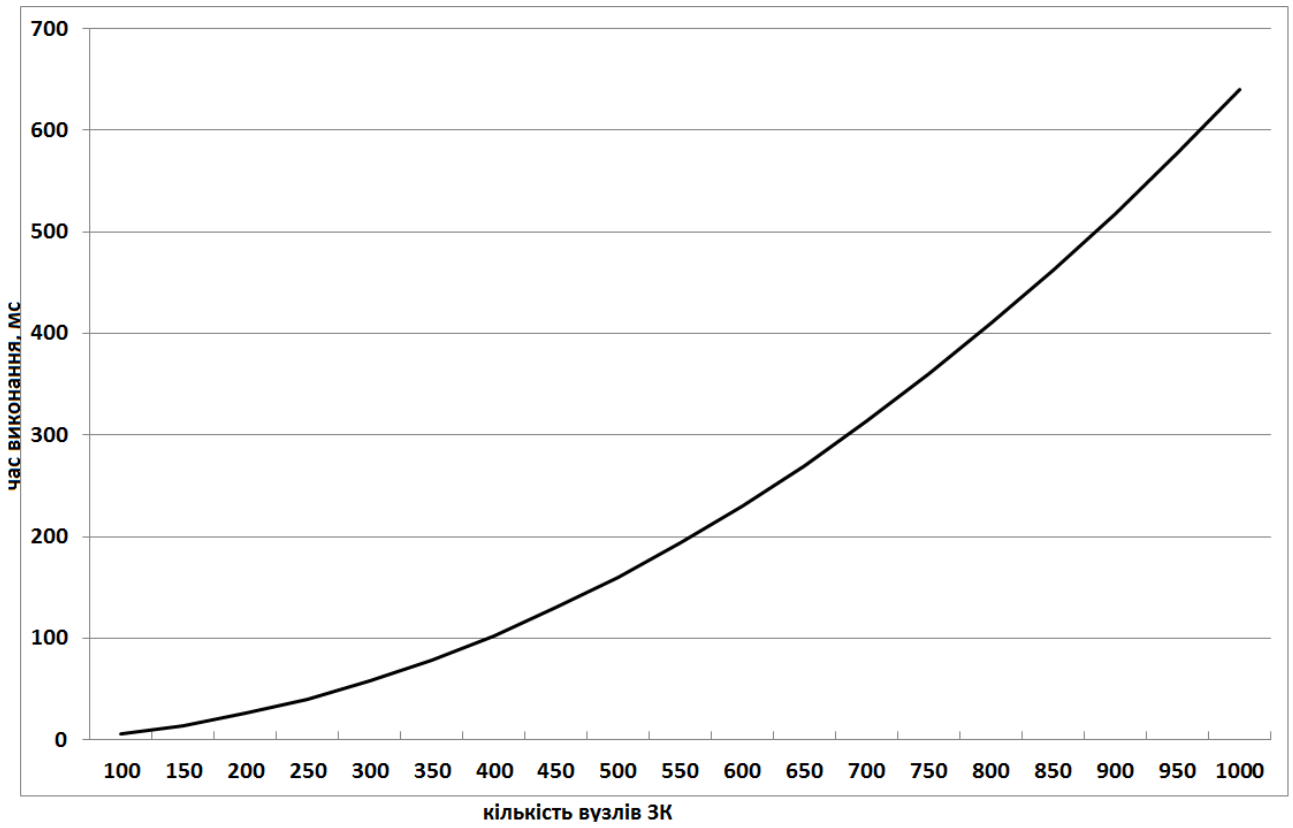


Рис. 4.5. Графік залежності часу виконання обчислень від кількості вузлів ЗК (без розпаралелення) на одноядерному комп’ютері

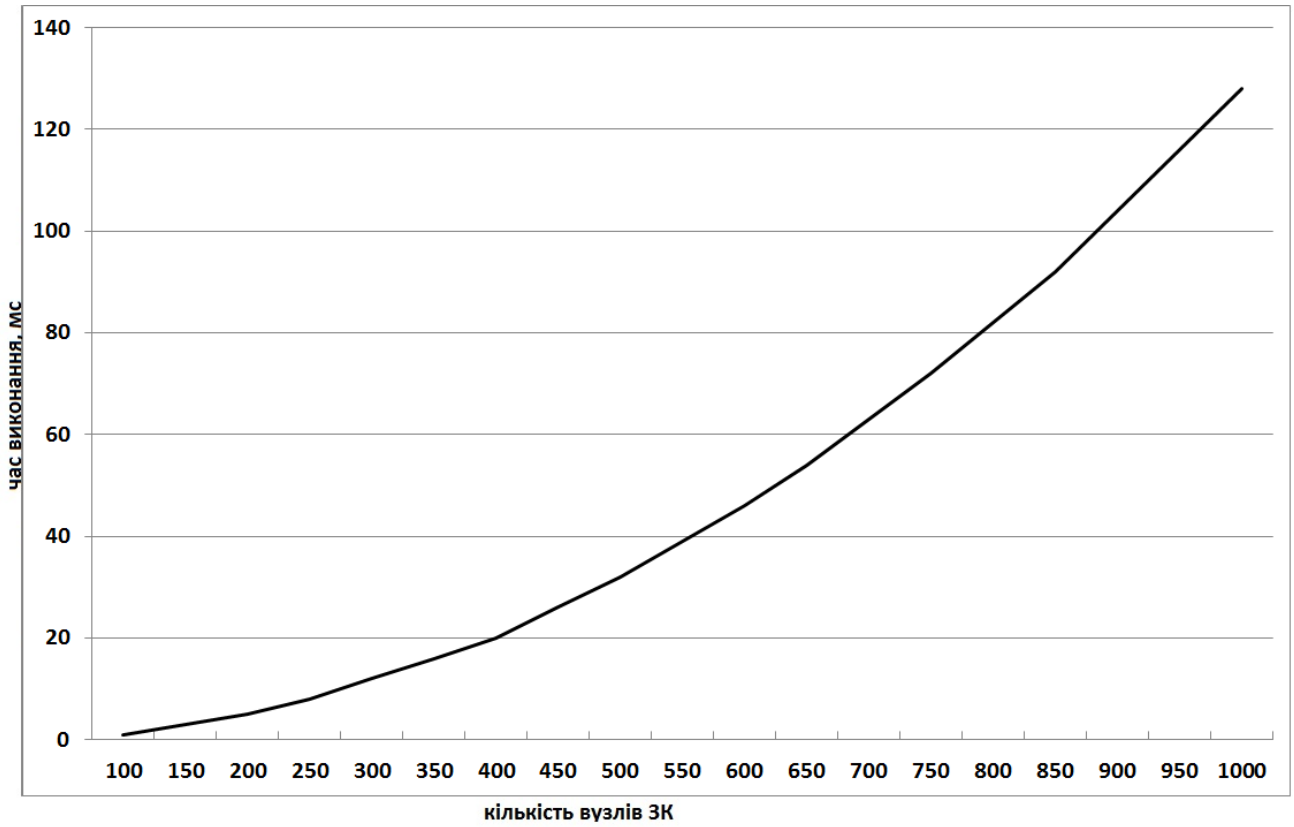


Рис. 4.6. Графік залежності часу виконання обчислень від кількості вузлів ЗК (з розпаралеленням на 7 потоків)

інших, послідовно виконується ініціалізація та обробка результатів після кожної ітерації циклу пошуку маршрутів.

Розглянуті залежності часу виконання обчислень від розмірності задачі N (кількість вузлів ЗК) як видно з рис.4.5, рис.4.6 мають стрімко зростаючий характер, який відповідає параболі в першій чверті координат, що підтверджує описану вище залежність часу обчислення від розмірності N як $O(N^2)$. Залежність часу виконання від кількості ітерацій циклу пошуку маршрутів мурахами-агентами пропорційна до часу необхідного на виконання однієї ітерації циклу.

Доцільним буде дослідити залежність швидкодії розробленої системи від кількості мурах-агентів, для зручності при виконанні 1000 ітерацій циклу пошуку маршрутів. Отримані результати потім можна буде використовувати як базові для розрахунків при більшій кількості ітерацій циклу, просто помножуючи отриманий час для одної ітерації на кількість ітерацій. Залежність часу виконання від кількості агентів зображено на рис. 4.7. Дослідження проводилось в межах 30 агентів, для ЗК на 1000 вузлів, процес обчислення відбувався без розпаралелення, на одноядерній ЕОМ.

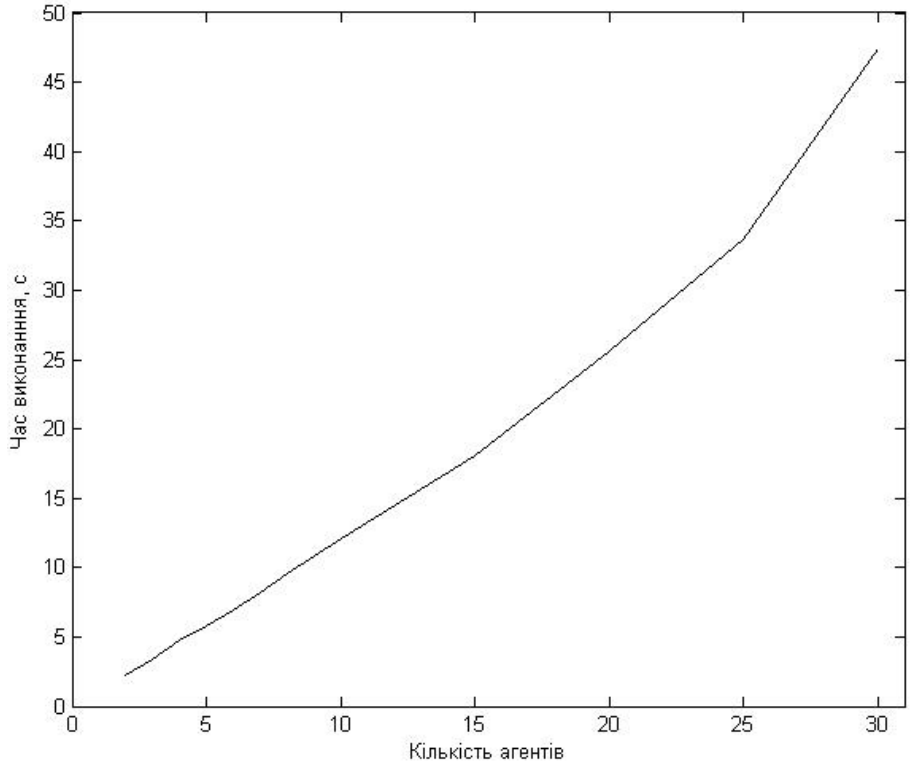


Рис. 4.7. Графік залежності часу виконання обчислень від кількості агентів

Результати були отримані в межах від 2 до 30 агентів, 2 агенти – це мінімальна кількість, необхідна для працездатності розробленої системи колективної поведінки агентів. Як видно з графіка (див. рис. 4.7) залежність часу необхідного для виконання розв’язання ЗК від кількості агентів має лінійний характер, без використання технологій паралельних обчислень.

На рис.4.8 показано залежність часу виконання від кількості агентів на багатоядерній ЕОМ, при використанні 1, 2 та 3-х ядер. При кожному вимірюванні запускалось обчислення ЗК на 1000 вузлів, при кількості ітерацій 250. Використання 2-х ядер дозволяє різко зменшити час обчилення ЗК. Як видно з графіка найкраще розпаралелення досягається при умові кратності кількості агентів до кількості ядер, за якої досягається рівномірний розподіл агентів між обчислювальними ресурсами ЕОМ. Графіки залежності мають ступінчастий характер, різкий зріст часу виконання обчислень ЗК відбувається коли максимальна кількість агентів, які запускаються на одному з ядер, збільшується. Аналогічний характер графіку видно на рис.4.9, на якому зображено графік залежності часу виконання обчислень ЗК від кількості використовуваних ядер, до 8-ми доступних на задіяній ЕОМ. При кожному вимірюванні запускалось обчислення ЗК на 1000 вузлів, при кількості ітерацій циклу пошуку маршрутів мурахами-агентами – 250, кількість агентів k – 8.

Максимальна ефективність була досягнута при кількості використовуваних ядер – 8, при використанні 2 та 4-х ядер досягається стрімке зниження часу виконання, що відповідає найбільшій ефективності при зменшенні середньої кількості агентів, які виконуються на одному ядрі.

Подальші дослідження було проведено на базі розробленої багатоагентної системи з застосуванням розробленого додаткового програмного модуля та з врахуванням запропонованої модифікації базового алгоритму. Для тестування та аналізу можливості розв’язання статичних та динамічних ЗК було вирішено використати бібліотечні файли, що включають вхідні дані для ЗК [107,108] та є тестувальним полігоном для порівняння ефективності різних методів та розроблених систем для розв’язання ЗК.

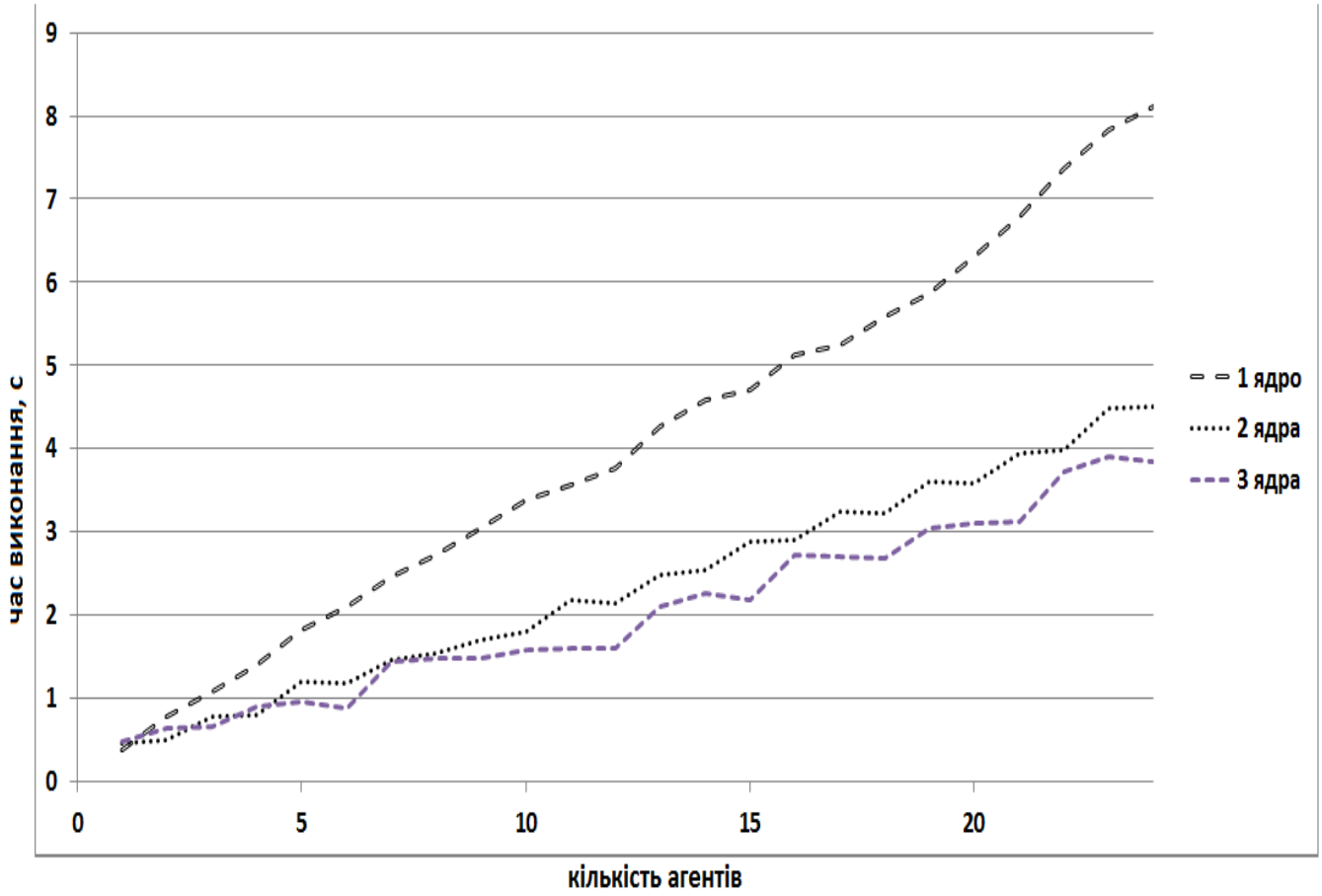


Рис. 4.8. Графік залежності часу виконання від кількості агентів при умові використання технологій паралельних обчислень

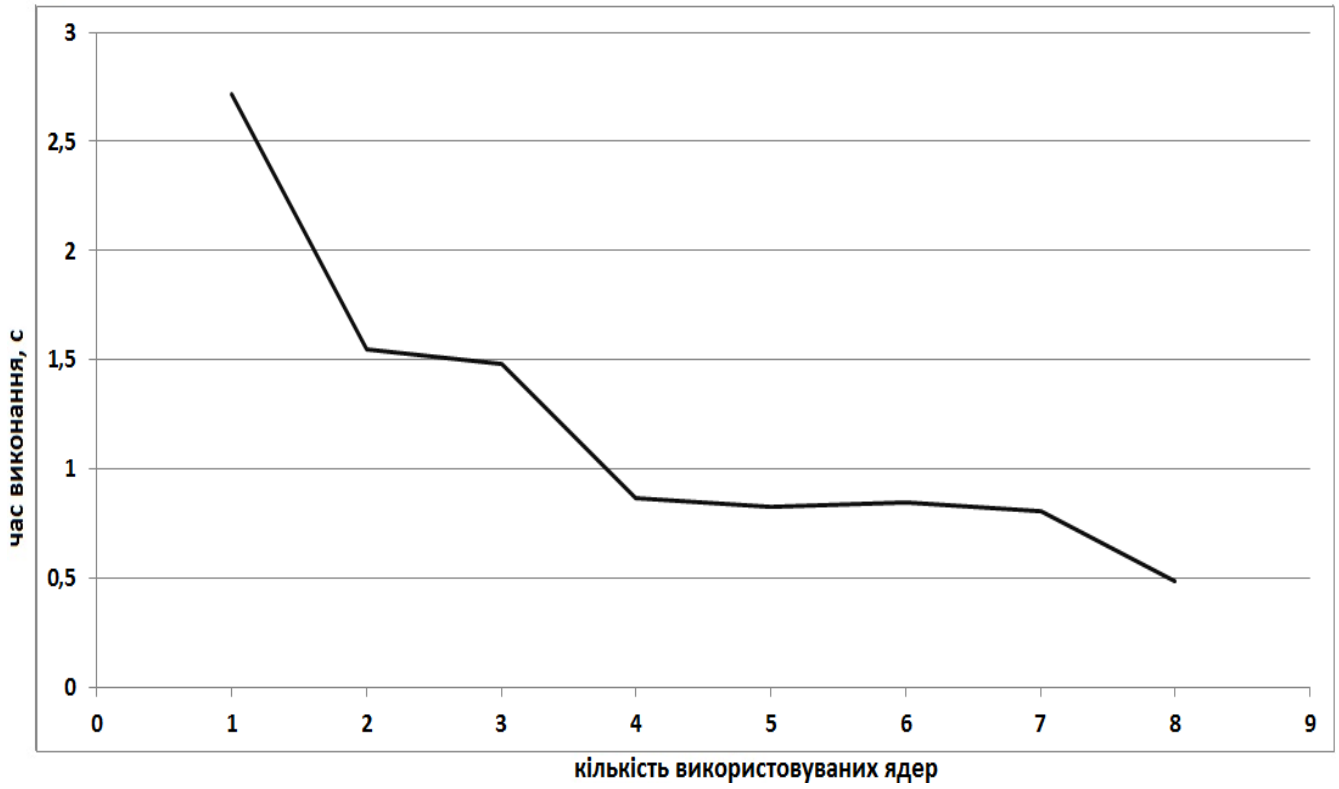


Рис. 4.9. Графік залежності часу виконання від кількості використовуваних ядер

Для початку роботи з розробленою багатоагентною системою достатньо запустити виконавчий файл ACS.exe та вказати вхідним аргументом ім’я бібліотечного файлу ЗК. Після чого з заданого вхідного файлу, наприклад «atr532.tsp», яка включає в себе 532 вузли – відповідно до основних крупних міст в США, або «lin318.tsp», яка включає в себе 318 вузлів – відповідно отворів на друкованій платі, буде зчитано всю інформацію по ЗК, яка включає в себе тип задачі TSP (симетрична ЗК) або ATSP (асиметрична ЗК), тип вхідних даних: матриця вартостей (відстаней) або координати; назва ЗК; розмірність ЗК; оптимальний розв'язок, якщо відомий; опис задачі. Загальні відомості по ЗК є необов'язковими. Файл має розташовуватись в одній директорії визначеній як джерело для зчитування даних, після його завантаження відбувається ініціалізація даних, виділення пам'яті та формування матриці вартостей.

Нижче наведено фрагмент (початок) вигляду бібліотечного файлу atr532.tsp, що вміщує симетричну ЗК на 532 вузли:

```
NAME : att532
TYPE : TSP
COMMENT : 532-city problem (Padberg/Rinaldi)
DIMENSION : 532

EDGE_WEIGHT_TYPE : ATT
NODE_COORD_SECTION
1 7810 6053
2 7798 5709
3 7264 5575
4 7324 5560
5 7547 5503
.....
529 1790 429
530 1503 362
531 5393 355
532 5469 10
EOF
```

Проведені дослідження вхідних параметрів для ЗК, дозволяють визначити оптимальні значення за замовчуванням для різного типу вхідних умов, з метою зменшення кількості обов’язкових вхідних параметрів.

Дослідження параметрів проводились без використання технологій паралельних обчислень на декількох випадково обраних ЗК, наведемо результати для двох з них ATT532 та LIN318, симетричні ЗК на 532 та 318 вузлів відповідно. Загальна кількість ітерацій циклу пошуку маршрутів мурахами-агентами – 1000, кількість запусків для отримання статистики – 20.

В таблицях 4.1 та 4.2 наведено дослідження введеного, згідно розробленої модифікації базового алгоритму, параметру PSlim – нижня імовірнісна межа. В таблиці 4.1 представлено результати для ATT532, в таблиці 4.2 – для LIN318.

Таблиця 4.1.

Дослідження впливу параметру PSlim на результати обчислення для ATT532

Значення PSlim	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час на знаходження квазі-оптимального результату, с
0.0	27686	27720	27699.10	582.90	19.44	11.73
0.1	27686	27718	27701.60	420.50	18.99	8.27
0.2	27686	27706	27700.40	286.90	18.92	7.92
0.3	27686	27709	27699.60	381.10	18.87	7.53
0.4	27686	27726	27704.10	366.70	18.29	7.02
0.5	27686	27732	27706.30	357.50	18.14	6.73
0.6	27686	27705	27698.40	328.40	17.54	6.07
0.7	27686	27714	27702.70	530.30	17.73	9.60
0.8	27693	27728	27708.10	453.90	17.33	8.06
0.9	27705	27738	27709.40	337.50	16.49	5.71
1.0	27709	27760	27724.90	604.80	16.40	10.10

Слід зазначити, що результати 27686 та 42029 є оптимальними для відповідних ЗК. Як бачимо найгріші результати досягаються при граничних значеннях параметра PSlim – 0 та 1.

Тобто постійно випадкове обирання, чи обирання найкращого для переходу вузла, дає гірші результати. Найкращі результати для двох різних розглянутих ЗК знаходяться в межах 0.2 – 0.3, як за часовими параметрами, так і за параметрами точності.

Таблиця 4.2.

Дослідження впливу параметру PSlim на результати обчислення для LIN318

Значення PSlim	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час на знаходження квазі-оптимального результату, с
0.0	42029	42143	42051.80	158.80	11.22	1.97
0.1	42029	42029	42029.00	231.90	11.44	2.81
0.2	42029	42029	42029.00	307.90	11.32	3.61
0.3	42029	42029	42029.00	190.70	11.13	2.22
0.4	42029	42143	42040.40	273.40	11.03	3.15
0.5	42029	42029	42029.00	398.70	11.02	4.50
0.6	42029	42143	42046.60	377.80	10.76	4.16
0.7	42029	42029	42029.00	548.90	10.58	5.89
0.8	42029	42143	42069.60	498.80	10.44	5.27
0.9	42029	42143	42065.20	272.30	10.25	2.88
1.0	42080	42165	42120.30	417.20	10.05	4.25

В таблицях 4.3 та 4.4 представлено аналогічні дослідження для параметру P для тих самих ЗК, що визначає інтенсивність зменшення значень міток («випаровування феромону»). Як видно з таблиці аналогічно до попереднього випадку граничні значення параметру дають найгірші результати.

Результати оптимальні за співвідношенням точності та часу обчислення для двох ЗК отримуються при значенні вхідного параметра P, що знаходиться в межах 0.2 – 0.5. Значення 0.2 обирається як значення за замовчуванням.

Таблиця 4.3.

Дослідження впливу параметру P на результати обчислення для LIN318

Значення P	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час знаходження квазі-оптимального результату, с
0.0	42129	42310	42258.50	533.50	24.23	9.82
0.1	42029	42029	42029.00	215.40	12.39	2.92
0.2	42029	42029	42029.00	144.90	11.62	1.82
0.3	42029	42143	42051.80	89.20	11.23	1.12
0.4	42029	42143	42040.40	361.90	11.45	4.22
0.5	42029	42091	42035.20	286.60	11.64	3.39
0.6	42029	42091	42041.40	449.60	11.35	5.16
0.7	42029	42163	42042.40	245.80	11.28	2.80
0.8	42029	42143	42040.40	163.80	11.31	1.89
0.9	42029	42163	42053.80	195.40	11.53	2.24
1.0	42029	42143	42068.60	256.50	11.24	2.91

Таблиця 4.4.

Дослідження впливу параметру P на результати обчисленнядля ATT532

Значення P	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час знаходження квазі-оптимального результату, с
0.0	27988	28059	28017.60	514.90	31.94	16.45
0.1	27686	27706	27698.50	619.20	21.19	13.56
0.2	27686	27716	27699.70	642.20	20.14	13.19
0.3	27686	27727	27703.10	446.40	19.47	8.99
0.4	27686	27705	27696.00	471.50	19.18	9.24
0.5	27686	27705	27698.40	396.10	18.63	7.61
0.6	27686	27735	27704.00	648.70	18.67	12.16
0.7	27686	27721	27699.70	417.80	18.43	7.80
0.8	27686	27725	27700.10	420.40	18.21	7.70
0.9	27686	27730	27700.90	555.20	18.19	10.15
1.0	27693	27738	27708.60	500.20	17.98	9.04

В таблиці 4.5 наведено дослідження співвідношення параметрів E та L, що відповідають вазі значення цифрової мітки на з’єднанні та фактичної вартості переходу по з’єднанню між вузлами при визначенні ймовірності переходу до наступного вузла. Окремо розглядати ці параметри немає змісту, оскільки важливо саме їх співвідношення.

Таблиця 4.5.

Дослідження впливу співвідношення параметрів E та L на результати обчислення для ATT532

Значення E L	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час знаходження квазі-оптимального результату, с
1 1	27686	27729	27703.70	455.90	20.77	9.91
1 1.5	27686	27708	27700.90	570.30	20.39	12.05
1 2	27686	27715	27701.80	495.00	19.92	10.28
1 3	27686	27705	27701.50	460.60	19.68	9.67
1.5 1	27686	27713	27700.20	566.00	18.19	10.50
2 1	27693	27727	27711.80	583.00	17.84	10.58
3 1	27686	27727	27711.80	627.30	17.44	11.04
0 1	27866	28037	27980.90	370.10	32.34	12.01
1 0	27686	27726	27704.70	640.30	24.32	15.86

Отримані результати для різних ЗК, дещо відрізняються, тому встановлення співвідношення даних параметрів за замовчуванням є найбільшою

складністю. Так для ЗК на 532 точки під назвою ATT532, отримані результати мають найкращі показники при співвідношенні E до L, як 1 до 2 або 1 до 3.

В таблиці 4.6 наведено дослідження результатів від кількості агентів – k. Дослідження проведено на кількості агентів від 1 до 100, оскільки більша кількість агентів призводить до суттєвих часових затрат.

Як видно з таблиці кількість агентів більше 30-ти істотно не змінює точність результату, лише значно збільшує час отримання квазі-оптимального маршруту (при використанні лише 1-го ядра). При кількості агентів в межах від 15 до 25 отримуються результати найкращі за співвідношенням точність та час обчислення. За замовчування будемо встановлювати кількість агентів – 20.

Таблиця 4.6.

Дослідження впливу кількості агентів на результати обчислення для ATT532

Значення k	Кращий результат	Гірший результат	Середній результат	Середня кількість ітерацій	Середній загальний час, с	Час знаходження квазі-оптимального результату, с
1	27700	27778	27727.90	733.40	2.33	1.74
2	27709	27795	27734.80	581.00	3.11	1.83
3	27693	27746	27714.40	653.30	3.92	2.59
4	27705	27732	27722.00	610.10	4.63	2.86
5	27693	27732	27712.50	526.30	5.46	2.92
6	27686	27734	27706.80	627.10	6.12	3.92
7	27696	27738	27712.20	503.80	6.95	3.59
8	27703	27741	27712.60	494.00	7.67	3.85
9	27693	27763	27712.40	543.80	8.43	4.69
10	27693	27715	27702.00	382.20	9.07	3.65
12	27686	27726	27704.30	624.10	10.44	6.65
15	27686	27717	27698.60	550.00	12.62	7.16
17	27686	27725	27703.20	570.00	14.27	8.47
20	27686	27705	27699.50	369.80	16.37	8.38
25	27686	27705	27693.00	392.20	20.10	8.36
30	27686	27705	27695.70	454.20	24.22	11.46
40	27686	27706	27698.20	353.00	30.87	11.64
50	27686	27705	27691.20	491.60	38.36	19.52
60	27686	27705	27694.60	537.90	45.89	25.28
70	27686	27705	27694.00	501.50	54.08	27.98
100	27686	27705	27694.00	382.20	73.79	29.68

В процесі розв’язання ЗК розробленою багатоагентною системою створюється та накопичується пам’ять колонії мурах, тобто отримується та

зберігається досвід колективу агентів, на який в подальшому орієнтуються мурахи-агенти при пошуку маршрутів. За допомогою пам’яті колонії мурах агенти здатні легко пристосовуватись до динамічних змін початкових вхідних даних.

При виникненні динамічних змін в процесі виконання поточної ітерації, методи локальної оптимізації запускати вже є недоцільним, тому відбувається запуск наступної ітерації циклу пошуку маршрутів. Пам’ять колонії мурах зберігає накопичений досвід, який після перезапуску ітерації та пошуку шляхів в нових умовах, зазнає змін: нові знайдені квазі-оптимальні маршрути менші за вартістю зберігаються в пам’яті, збережені вже неактуальні маршрути поступово «забуваються» – втрачають свою значимість при виборі вузлів для переходу мурахами-агентами, через процедуру «випаровування феромону».

На рис.4.10 та рис.4.11 представлено пам’ять колонії мурах у вигляді графу для ЗК (ATT532). Товщина ребра графу тим більша, чим більше значення мітки, яке відноситься до цього ребра – з’єднання між вузлами.

На рис.4.10 показано стан на ранньому етапі – після виконання 5-и ітерацій циклу пошуку маршрутів мурахами-агентами. На рис.4.11 представлено стан пам’яті після 30-и ітерацій циклу пошуку маршрутів. Як видно з рисунку багато ребер зникло, усунено як непридатні, через періодичне зменшення значення мітки («випаровування феромону») та відсутність збільшення значень міток на не використовуваних з’єднаннях («нанесення феромону») через відсутність проходження по них мурахами-агентами.

Фактично на рис.4.10 та рис.4.11 графічно відображено матрицю М в різні моменти часу обчислення ЗК на 532 вузли, ділянки з найбільшою товщиною використовуються мурахами-агентами частіше, через те що входять в більшість знайдених квазі-оптимальних маршрутів. Ребра з мінімально встановленими значеннями (значення мітки згідно описаних в розділі 1.3 правил мають бути не нульовими) на рисунку не позначено та вважаються як «зниклі».

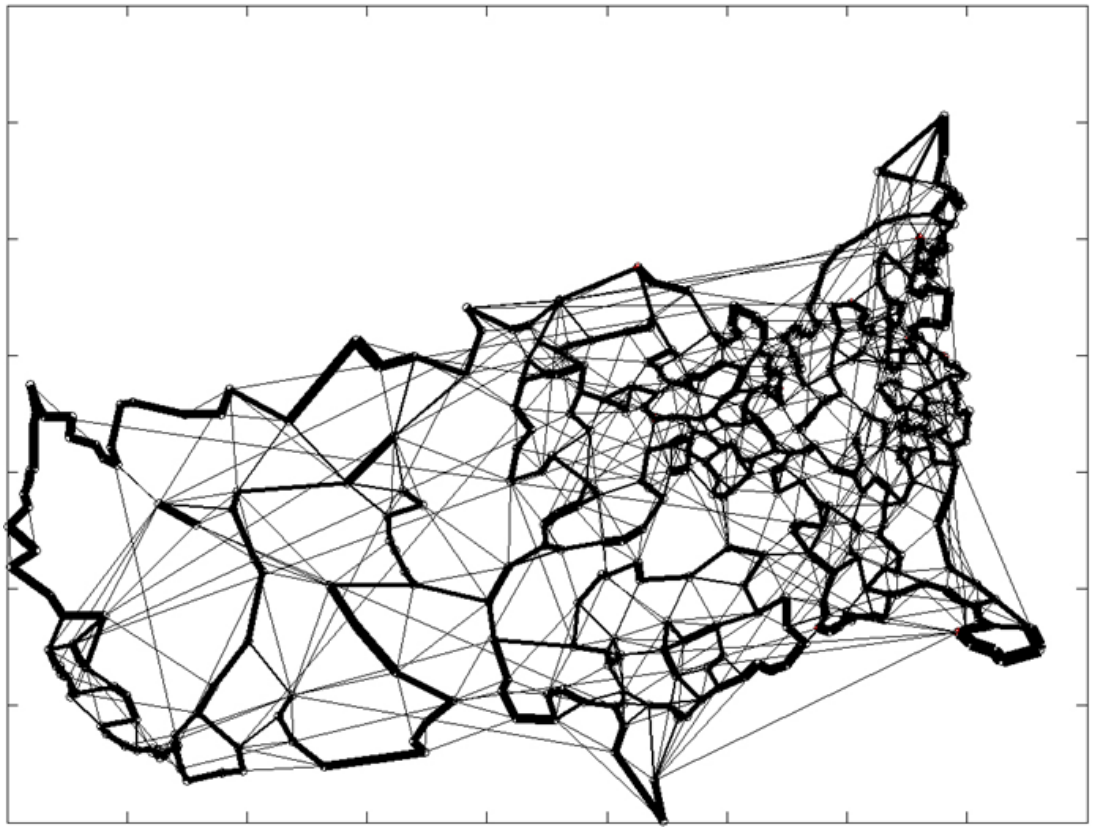


Рис. 4.10. Вигляд пам’яті колонії мурах – матриці значень міток після 5 ітерацій циклу пошуку маршрутів при розв’язанні ЗК на 532 вузла, ATT532

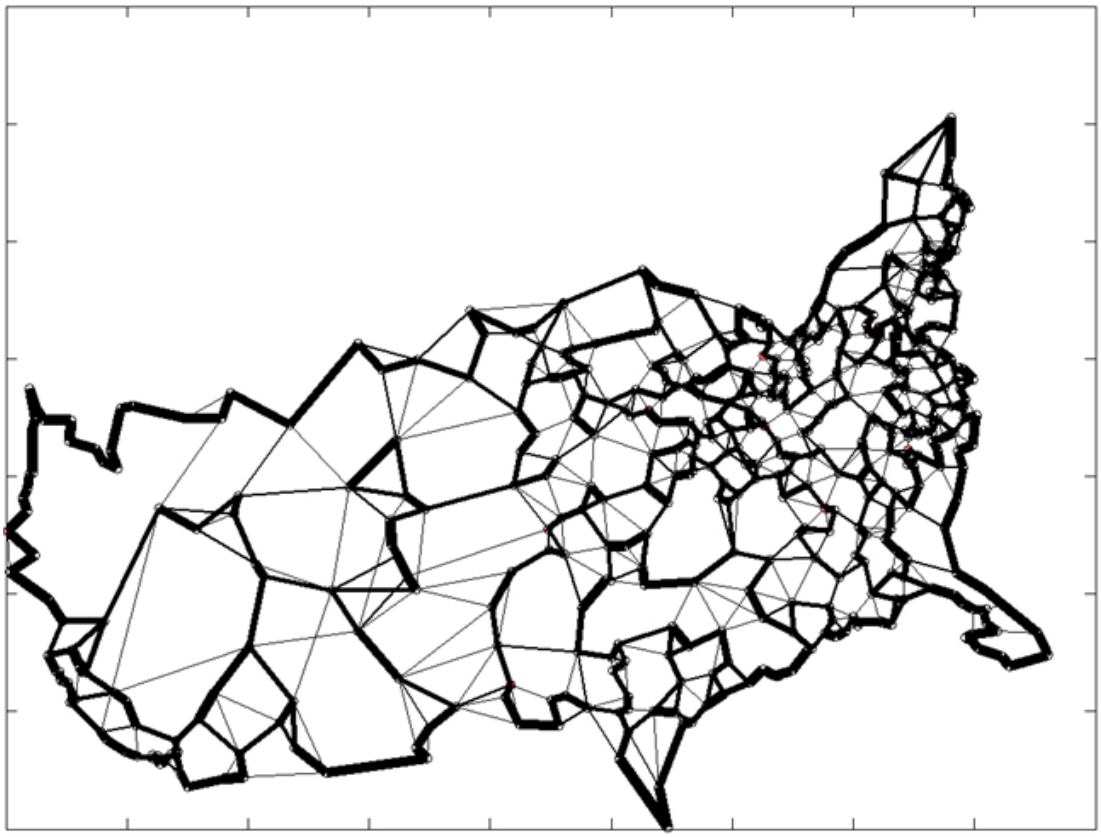


Рис. 4.11. Вигляд пам’яті колонії мурах – матриці значень міток після 30 ітерацій циклу пошуку маршрутів при розв’язанні ЗК на 532 вузла, ATT532

Доцільність використання методів локальної оптимізації. Розроблені 4-и режима функціонування системи дозволяють визначати необхідне співвідношення часу обчислення та точності результату. Також для порівняння було включено аналіз альтернативної системи LKH 2.0 [83] для розв'язання ЗК на базі LKH – методу, що був розглянутий у першому розділі.

Тестування проводилось на ЕОМ з процесором Intel Core i7-5960X Haswell-E 8-Core 3.0 GHz, 8 ядер, 16 ГБ ОЗУ. Крім обчислення ЗК було додано запис усіх проміжних результатів у файл (логування усіх етапів та подій) для обох систем, яке також займає певний додатковий час крім виконання обчислення ЗК (до 60% сумарного часу).

Кількість запусків для збору статистики – 100. Результати тестування розробленої багатоагентної системи та системи на базі LKH-методу для порівняння представлено в таблиці 4.7. MIN, AVG, MAX – позначення відповідно для мінімального, середнього та максимального результатів. Оптимальний маршрут для даної ЗК має довжину 27686 (виділено в таблиці), його було досягнуто з застосуванням розробленої системи в режимі використання методу 3-opt локальної оптимізації та за допомогою системи на базі LKH-методу.

При запуску розробленої багатоагентної системи використовувались 8 агентів, в таблиці 4.7 час дослідження показано при виконанні на одному ядрі, в дужках значення часу при виконанні обчислень з використанням 8-и доступних ядер. Режим роботи розробленої багатоагентної системи визначається вхідним параметром ***opt,*** який змінюється в залежності від інтенсивності динамічних змін в процесі обчислення**.** Система на базі методу LKH має найкращі показники за точністю отримуваних результатів.

Однак крім точності слід враховувати ще й час обчислення ЗК, а також здатність пристосовуватись до динамічних змін. При виникненні динамічних змін систему на базі LKH методу необхідно перезапускати. Розроблену багатоагентну систему для розв'язання динамічної ЗК перезапускати не потрібно, відбувається виконання наступної ітерації циклу пошуку маршрутів

мурахами-агентами з метою пошуку нового маршруту після виникнення динамічних змін, накопичений досвід (пам’ять колонії мурах) не втрачається та корегується агентами.

Таблиця 4.7.

Характеристики точності та часу обчислення результату ЗК (ATT532)

Досліджувана система	Результат за вартістю			Час знаходження результату, с			Відхилення середнього від оптимального результату, %
	MIN	AVG	MAX	MIN	AVG	MAX	
Розроблена система з використанням 3-opt методу локальної оптимізації	**27686**	27700.3	27705	1,96 (0,39)	4,95 (0,88)	8,7 (1,55)	0,07
Розроблена система з використанням 2,5-opt методу локальної оптимізації	27693	27712.8	27751	2.3 (0,41)	4.6 (0,82)	7.1 (1,27)	0.1
Розроблена система з використанням 2-opt методу локальної оптимізації	27717	27733.4	27762	1.9 (0,33)	4.4 (0,78)	7.1 (1,26)	0.2
Розроблена система без використання методів локальної оптимізації	30255	30450,7	33835	0,6 (0,11)	0,9 (0,16)	1,4 (0,25)	9,98
Система на базі методу LKH	**27686**	**27686**	**27686**	3,3	7,0	10,9	0

Так для системи на базі методу LKH необхідно в середньому 7 секунд для розв’язання даної ЗК (3.5 секунди без врахування збережнення проміжних результатів в файли), тобто більше ніж для усіх представлених режимів функціонування розробленої системи. Також слід врахувати, що наявність динамічних змін вхідних даних ЗК з періодом меншим ніж час обчислення результату (3 секунди), робить розв’язання ЗК методом LKH не можливим, оскільки потрібно буде кожного разу перезапускати систему на базі цього методу, а отриманий результат через динамічні зміни вхідних даних може бути вже недоцільним для використання. При застосуванні 8-ми ядер розроблена багатоагентна система здатна отримувати результуючий маршрут ЗК на порядок швидче ніж система на базі методу LKH, навіть при використанні методів локальної оптимізації. Розроблена багатоагентна система на базі алгоритму колонії мурах з врахуванням розробленої модифікації базового алгоритму дозволяє отримати розв’язок з високим показником за точністю. При різниці з

оптимальним результатом, що складає приблизно 10%, розроблена система здатна видати результат вже за 1 секунду на одноядерному ЕОМ (190 мс при використанні 8-ми ядер). Більш того в режимах 3-opt, 2.5-opt, 2-opt, при потребі, можна зупинити виконання обчислень вже після першої ітерації циклу пошуку маршрутів мурахами-агентами

Через конструктивний характер знаходження маршруту в кожний момент часу після кожної, починаючи з першої, ітерації циклу пошуку маршрутів мурахами-агентами, яка відбувається за декілька мілісекунд можна отримати результуючий маршрут для розглянутої тестової ЗК на 532 вузли.

Отже залежно від інтенсивності динамічних змін доцільно використовувати певний режим функціонування розробленої системи. За замовчуванням використовується ***opt*** = 3 режим (використання 3-opt алгоритму локальної оптимізації). Крім того, при виникненні динамічних змін час необхідний для знаходження нового квазі-оптимального маршруту значно менший ніж час повторного запуску обчислення, як у випадку використання системи на базі LKH-методу. Для розробленої системи достатньо продовжити виконання ітерацій циклу пошуку машрутів мурахами-агентами.

Розв’язки ЗК ATT532 представлені графічно, отримані в наступних режимах функціонування багатоагентної системи: без використання методів локальної оптимізації на рис.4.12, з використанням 3-opt методу локальної оптимізації на рис.4.13. Слід також зазначити, що маршрут отриманий без використання методів локальної оптимізації подається на вхід модуля методу локальної оптимізації для опрацювання.

Використання методів локальної оптимізації для опрацювання результуючого маршруту дозволяє позбавитись з’єднань великої вартості, які виникають через пропускання існуючих вузлів в процесі проходження агентів.

В ході досліджень було розв’язано набір бібліотечних ЗК з метою аналізу можливостей розробленої багатоагентної системи з використанням поведінкової моделі колонії мурах, технологій паралельних обчислень та методів локальної оптимізації для розв’язання ЗК динамічного характеру.

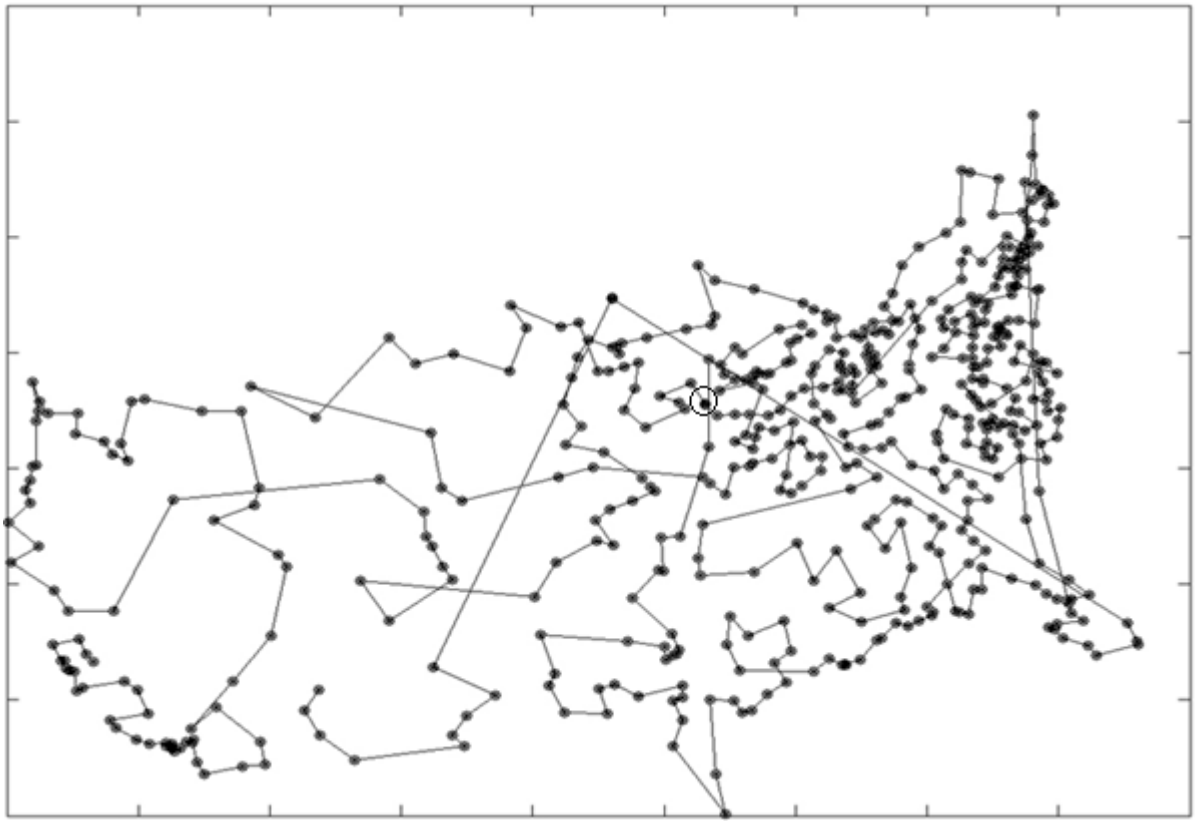


Рис. 4.12. Результуючий маршрут ЗК (ATT532) отриманий без використання методів локальної оптимізації

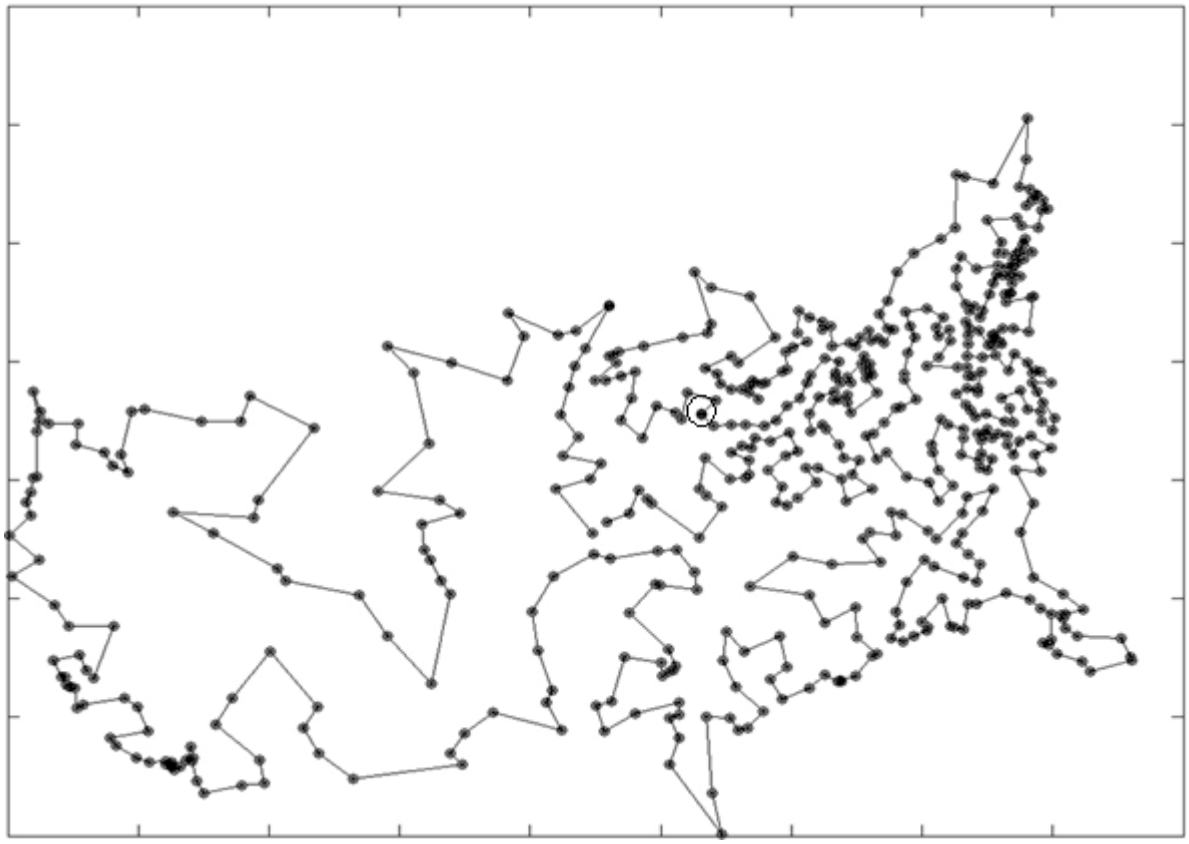


Рис. 4.13. Результуючий маршрут ЗК (ATT532) отриманий з використанням 3-opt методу локальної оптимізації

В таблиці 4.8 представлено результати розв’язання бібліотечних ЗК, в назві цифрами позначають розмірність ЗК. Результати представлені в абстрактних величинах без одиниць вимірювання. Середня кількість ітерацій – кількість запусків агентів, які знадобилось для знаходження квазі-оптимального маршруту. Виділені результати є оптимальними для даної ЗК, якщо оптимального результату досягнуто не було його вказано в дужках знизу.

Кожна ЗК розв’язувалась 100 разів, при кожному розв’язку зберігалися наступні дані:

- отриманий результуючий маршрут мінімальної вартості;
- номер ітерації циклу в якій було отримано маршрут мінімальної вартості;
- обчислену вартість найгіршого та найкращого результату;
- середню вартість результату;
- середню кількість ітерацій, за яку можна отримати кінцевий результат;
- середній загальний час розв’язання ЗК – час необхідний на виконання 1000 ітерацій (для кожної ЗК) циклу пошуку маршрутів мурахами-агентами.

Описані дані представлено в таблиці 4.8 для обраних бібліотечних ЗК з розмірністю до 6000. Середній час знаходження квазі-оптимального результату та середній загальний час виконання 1000 ітерацій циклу пошуку маршрутів мурахами-агентами представлено в таблиці 4.8 в секундах при виконанні на одноядерній ЕОМ, в дужках позначено час при використанні 8-ми ядер.

Розроблена багатоагентна система для розв’язання динамічної ЗК дозволяє гнучко пристосовуватись до різної частоти динамічних змін. Чим більше частота, тим краще використовувати режим без застосування локальної оптимізації або режим з використанням 2-opt методу локальної оптимізації. В будь-якому випадку при появі динамічних змін розроблену систему не потрібно перезапускати. Для розв’язання статичної ЗК з високою точністю застосовується режим з використанням 3-opt методу локальної оптимізації.

Таблиця 4.8.

Результати розв’язання біблотечних ЗК

Назва ЗК	Вартість кращого результату	Вартість гіршого результату	Середня вартість результату	Середня кількість ітерацій	Середній загальний час виконання, с	Середній час знаходження квазі-оптимального результату, с
a280	**2579**	**2579**	2759.0	28.5	6.59 (1.19)	0.29 (0.06)
ali535	**202339**	202454	202390.1	482.9	24.02 (4.32)	10.02 (1.94)
att532	**27686**	27705	27698.4	396.1	18.63 (3.32)	5.61 (1.02)
att48	**10628**	**10628**	10628.0	3.5	1.95 (0.35)	0.01 (0.002)
berlin52	**7542**	**7542**	7542.0	1.0	2.17 (0.44)	0.006 (0.002)
bier127	**118282**	**118282**	118282.0	17.1	5.47 (1.05)	0.13 (0.029)
ch130	**6110**	**6110**	6110.0	10.9	4.81 (0.88)	0.08 (0.03)
ch150	**6528**	**6528**	6528.0	15.2	4.93 (0.94)	0.11 (0.05)
d198	**15780**	15781	15780.4	348.6	8.66 (1.61)	3.06 (0.89)
d493	35004 (35002)	35033	35015.8	548.6	19.78 (3.53)	11.05 (2.27)
d657	48930 (48912)	48989	48967.0	563.2	27.07 (4.93)	15.50 (3.79)
d1291	**50801**	50825	50814.8	350.3	40.17 (7.30)	14.87 (3.11)
d1655	62169 (62128)	62411	62286.4	857.5	51.94 (10.38)	45.14 (10.12)
dsj1000	18665854 (18660188)	18699764	18689299.2	866.2	48.92 (9.23)	42.78 (9.98)
eil51	**426**	**426**	426.0	2.9	1.75 (0.33)	0.01 (0.007)
eil76	**538**	**538**	538.0	7.2	2.51 (0.46)	0.03 (0.01)
eil101	**629**	**629**	629.0	12.4	3.29 (0.67)	0.06 (0.02)
fl417	11861	11861	11861.0	302.9	14.88 (3.94)	4.65 (1.03)
fl1400	20188 (20127)	20259	20220.8	715.1	60.62 (13.11)	43.25 (9.05)
fl1577	22329 (22249)	22485	22360.8	599.7	46.75 (9.17)	28.10 (6.67)
gil262	**2378**	**2378**	2378.0	72.9	8.84 (1.81)	0.78 (0.21)
gr96	**55209**	**55209**	55209.0	8.8	3.90 (0.84)	0.04 (0.03)
gr137	**69853**	**69853**	69853.0	12.1	5.30 (1.10)	0.09 (0.02)
gr202	**40160**	**40160**	40160.0	124.0	8.82 (1.74)	1.17 (0.39)
gr431	**171414**	171463	171426.8	588.8	18.11 (3.26)	10.90 (2.28)
gr666	**294382**	**295059**	294584.5	565.3	27.21 (5.21)	15.60 (3.19)
lin318	**42029**	**42029**	42029.0	405.3	11.86 (2.22)	4.91 (1.22)
nrw1379	56717 (56638)	56812	56762.6	848.9	53.68 (11.76)	46.91 (11.31)
pr2392	378599 (378032)	379939	379173.7	752.0	91.42 (19.04)	71.59 (16.88)
rl5915	568473 (565530)	576503	571753.0	900.2	366.64 (76.16)	330.33 (69.77)

При виникнені динамічних змін в процесі розв’язання ЗК, достатньо продовжити виконання обчислень. При різниці з оптимальним результатом, що складає лише 10%, розроблена система здатна видати результат вже за 1 секунду на одноядерному ЕОМ (190 мс при використанні 8-ми ядер), а за менше ніж 2 секунди отримати оптимальний розв’язок при використанні 8-ми ядерної ЕОМ. При виникненні динамічних змін розроблена система здатна видати результат вже після виконання першої ітерації циклу запуску мурах-агентів, що складає декілька мілісекунд. Розроблена система здатна обчислювати ЗК в умовах динамічних змін без потреби перезапуску системи, що є неможливим для альтернативної розглянутої системи на базі LKH-методу. Зауважимо, що LKH-метод недостатньо підходить для розв’язання ЗК в умовах динамічних змін з інтенсивністю меншої за час обчислення ЗК даним методом – для ЗК на 532 вузли це складає 3,3 с (див. табл.4.7).

4.2. Результати досліджень багатоагентної системи для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних

Вхідні дані та вихідні результати для розробленої багатоагентної системи для розв’язання динамічної асиметричної ЗК в умовах частково невідомих даних зберігаються в файлах. Всі файли мають однакове ім’я – назву ЗК, міняється лише розширення файлу відповідно до його призначення, використовуються наступні розширення:

1) .time – вхідний файл, який містить матрицю вартостей C.
2) .acs – вхідний файл, який містить матрицю доступностей A.
3) .cnf – вхідний файл, який містить вхідні параметри.
4) .snc – вхідний файл, який містить чергу очікуваних подій по зміні вхідних даних (вартості або доступності з’єднання, вузла, тощо), є необов’язковим.
5) .res – вихідний файл, який містить кінцеві маршрути отримані кожним агентом після кожної ітерації циклу пошуку маршрутів мурахами-агентами.

6) .log – вихідний файл, який містить інформацію про виконання усіх подій протягом розв’язання вхідної ЗК.
7) .mem – вихідний файл, який містить інформацію про стан «пам’яті колонії мурах» (матриці M) після кожної ітерації циклу пошуку маршрутів мурахами-агентами.

З метою демонстрації здатності виходити з ситуації «пастки» було використано ЗК розмірністю 10 вузлів з назвою “trap_task”. Представлення у вигляді графа даної ЗК зображено на рис.4.14. Вузли, тут і надалі, позначаються колами, цифра в колі – ідентифікатор вузла, стрілками позначено з’єднання та можливі напрями руху агента по ньому.

На рис.4.15 представлено вмістиме вхідних файлів для даної ЗК, зліва – файл “trap_task.acs” з матрицею A, справа – файл “trap_task.time” з матрицею C. Початковий вузел – 1-й, кількість агентів – 6.

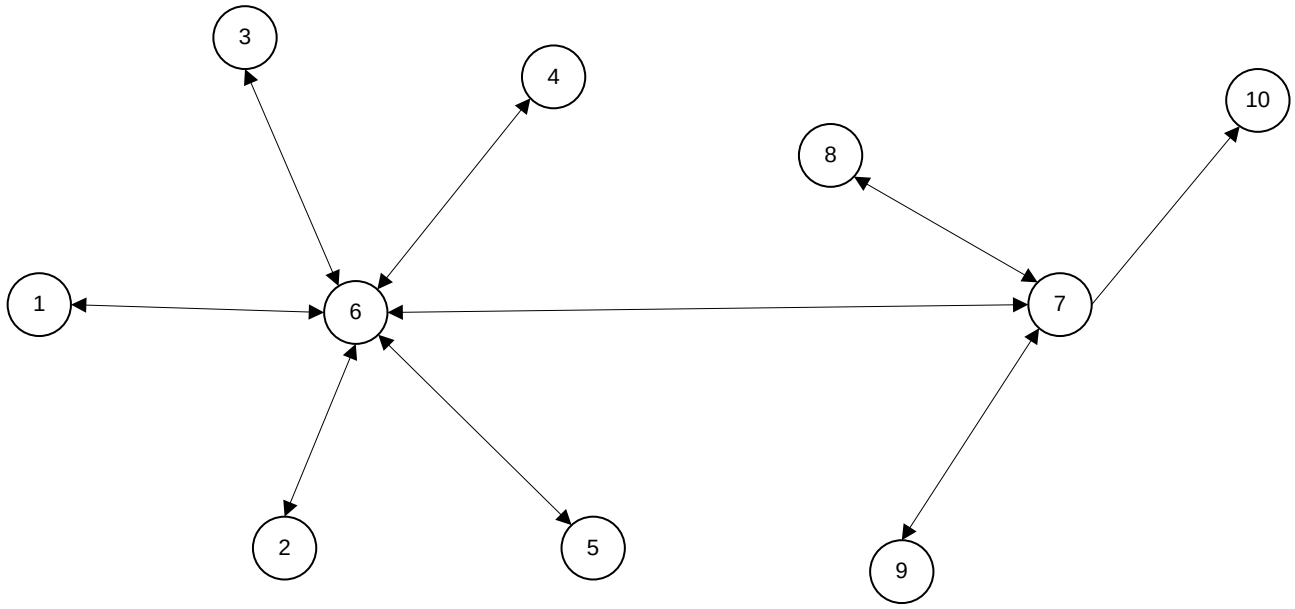


Рис.4.14. Графічне представлення вхідної мережі ЗК з ситуацією «пастки»

```
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
1 1 1 1 1 0 1 0 0 0
0 0 0 0 0 1 0 1 1 1
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 210 0 0 0 0
0 0 0 0 0 140 0 0 0 0
0 0 0 0 0 415 0 0 0 0
0 0 0 0 0 170 0 0 0 0
0 0 0 0 0 300 0 0 0 0
201 130 421 161 310 0 950 0 0 0
0 0 0 0 0 900 0 200 250 180
0 0 0 0 0 0 210 0 0 0
0 0 0 0 0 0 190 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Рис.4.15. Вмістиме вхідних файлів для “trap_task” ЗК з ситуацією «пастки»

Через структуру з’язків, розв’язати “trap_task” ЗК неможливо, оскільки буде порушено умову одноразового обходження усіх вузлів. Проте задача обходження всіх вузлів при мінімальній вартості машруту має бути розв’язано. Однак на вузлі з ідентифікатором 10 (див. рис. 4.10) виникає ситуація, коли є неможливим вибратись для будь-якого агента, вузли для переходу відсутні.

Розглянемо хід виконання обчислення до імплементації розробленого алгоритму виходу з ситуації «пастки». Вмістиме файлу «trap_task.res» наведено в Додатку І. Рядок результуючого маршруту одного з агентів з цього файлу наведено нижче:

Route of [5] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Результати кожної ітерації відокремлюються, кожний рядок представляє собою маршрут агента, в квадратних дужках вказано ідентифікатор агента, сам маршрут представлений послідовністю ідентифікаторів вузлів, в процесі їх відвідування. «SPENT TIME» – вартість пройденого маршруту, «currently at MarkId» – ідентифікатор вузла розташування агента-пакета, «STATUS» – статус завершення подорожі, тобто чи повернувся агент за визначений час до початкового вузла (FINISHED) чи ні (UNFINISHED).

Як видно з отриманих результатів, усі агенти потрапляють до вузла з ідентифікатором 10, та залишаються там в «пастці», до початкового вузла не повернувся жоден агент за 2 ітерації циклу пошуку маршрутів мурахами-агентами.

Після імплементації розроблених засобів (див. розділ 2.3) на базі запропонованого алгоритму виходу з ситуації «пастки» агенти в системі отримала здатність повернутись до початкового вузла, завершуючи шлях. Отримані результати представлено в Додатку І – вмістиме файлу «trap_task2.res».

Згідно отриманих результатів видно, що тепер протягом першої ітерації циклу 3 агенти одночасно потрапляють до вузла з ідентифікатором 10. Після чого агенти виявляють, що потрапили до ситуації «пастки» та згідно

розробленого алгоритму забезпечують уникання даного вузла іншими агентами. Тобто з 12 агентів запущених протягом 2 ітерацій циклу пошуку маршрутів мурахами-агентами повернулось 9 до початкового вузла.

В процесі розв'язання розглянутої ЗК (див. рис. 4.14) також було виявлено явище «зациклення». Для прикладу приведемо декілька маршрутів пройдених агентами, які потрапляли в ситуацію «зациклення» на даній вхідній мережі з 10 вузлів, нижче наведено 3 маршрути з 3-ї ітерації циклу пошуку маршрутів мурахами-агентами, при виникненні ситуації «зациклення»:

Route of [16] agent : 1 6 2 6 4 6 5 6 3 6 7 8 7 9 7 8 7 9 7 8 7 9 7 || SPENT TIME = 5757 currently at MarkId = 7 STATUS = UNFINISHED

Route of [14] agent : 1 6 4 6 2 6 3 6 5 6 7 8 7 9 7 8 7 9 7 8 7 9 7 || SPENT TIME = 5757 currently at MarkId = 7 STATUS = UNFINISHED

Route of [18] agent : 1 6 2 6 5 6 3 6 4 6 7 8 7 9 7 8 7 9 7 8 7 9 7 || SPENT TIME = 5757 currently at MarkId = 7 STATUS = UNFINISHED

Крім того, що агенти з ідентифікаторами 14, 16 та 18 на 3-й ітерації циклу не змогли повернутись до початкового вузла, значення цифрових міток (матриця M) на відповідних з'єднаннях продовжували збільшувались. Нижче представлено стан «пам'яті колонії мурах» через 10 ітерацій з наявністю агентів, які потрапили в ситуацію зациклення на вузлах з ідентифікаторами 7,8,9 (файл «trap_task.mem»):

[1] : 0.01 0.01 0.01 0.01 0.01 13.01 0.01 0.01 0.01 0.01

[2] : 0.01 0.01 0.01 0.01 0.01 12.27 0.01 0.01 0.01 0.01

[3] : 0.01 0.01 0.01 0.01 0.01 12.55 0.01 0.01 0.01 0.01

[4] : 0.01 0.01 0.01 0.01 0.01 12.27 0.01 0.01 0.01 0.01

[5] : 0.01 0.01 0.01 0.01 0.01 12.58 0.01 0.01 0.01 0.01

[6] : 0.01 12.19 12.3 12.2 12.4 0.01 13.01 0.01 0.01 0.01

[7] : 0.01 0.01 0.01 0.01 0.01 0.01 0.01 **1123 1150.9** 11.5

[8] : 0.01 0.01 0.01 0.01 0.01 0.01 **1155** 0.01 0.01 0.01

[9] : 0.01 0.01 0.01 0.01 0.01 0.01 **1325** 0.01 0.01 0.01

[10] : 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01

В квадратних дужках позначено ідентифікатор вузла, на якому зберігаються значення цифрових міток в порядку відповідно до існуючих сусідніх вузлів. Потовщеним шрифтом виділено найбільші значення цифрових

міток, стрімке збільшення яких пов’язано з виявленими негативними наслідками надмірного укріплення «пам’яті колонії мурах».

Розв’язати розглянуту ЗК «trap_task» на 10 вузлів дозволило проведення імплементації розроблених засобів (див. розділ 2.3):

1) виходу мурахи-агента з критичних ситуацій ***«пастки»*** та ***«зациклення»***;

2) оновлення значень цифрових міток в процесі проходження сполучення між вузлами;

3) організації функціонування системи в режимі поєднання процесу пошуку маршрутів з процесом передачі даних функціонування системи;

4) подолання негативних наслідків надмірного укріплення «пам’яті колонії мурах» шляхом застосування адаптивної верхньої межі цифрової мітки Mmax.

Розроблена система забезпечує вимогу обходження вузлів та повернення в початковий вузел агентів, умова однократного проходження вузлів виконується тоді, коли це можливо. В результаті було усунуто явище «зациклення», значення цифрових міток для даної ЗК не перевищували 50. Розглянемо здатність розробленої багатоагентної системи до розв’язання ЗК в умовах динамічних змін. Для цього використаємо створену ЗК на 10 вузлів “mesh_task”, вхідна мережа представлена у вигляді графу, який зображено на рис.4.16. Дана ЗК має сітчасту фізичну топологію мережі, яку ще називають mesh мережею [72].

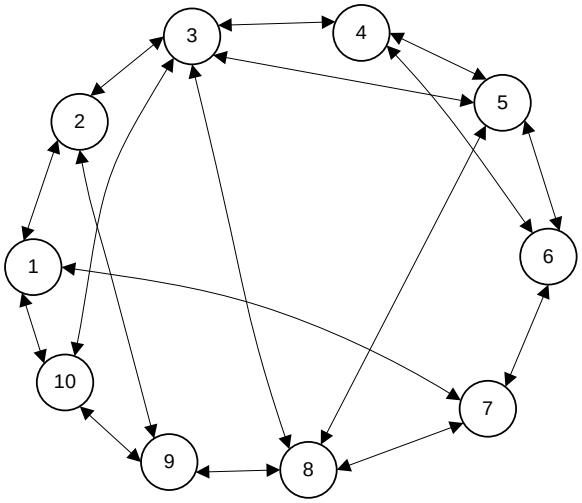


Рис.4.16. Графічне представлення вхідної мережі “mesh_task” ЗК

Кожен вузел має більше 2-х з’єднань з іншими вузлами, в даному прикладі всі з’єднання двонаправлені, ЗК є асиметричною, матриці А та С представлено на рис.4.17.

```
0 80 0 0 0 0 90 0 0 70
90 0 30 0 0 0 0 0 50 0
0 50 0 40 50 0 0 70 0 110
0 0 110 0 70 70 0 0 0 0
0 0 80 150 0 80 0 100 0 0
0 0 0 80 60 0 60 0 0 0
110 0 0 0 0 80 0 50 0 0
0 0 120 0 120 0 70 0 80 0
0 60 0 0 0 0 0 150 0 60
90 0 80 0 0 0 0 0 110 0
```

```
0 1 0 0 0 0 1 0 0 1
1 0 1 0 0 0 0 0 1 0
0 1 0 1 1 0 0 1 0 1
0 0 1 0 1 1 0 0 0 0
0 0 1 1 0 1 0 1 0 0
0 0 0 1 1 0 1 0 0 0
1 0 0 0 0 1 0 1 0 0
0 0 1 0 1 0 1 0 1 0
0 1 0 0 0 0 0 1 0 1
1 0 1 0 0 0 0 0 1 0
```

Рис.4.17. Вмістиме вхідних файлів для “mesh _task” ЗК

При розв’язанні використаної ЗК на 10 вузлів “mesh_task” при зведенні агентів до проходження вже після 17-ї ітерації по знайденому квазіоптимальному результуючому маршруту, значення цифрових міток Mij на з’єднаннях лише зростають, процедура «випаровування феромону» лише усуває значення на не використовуваних з’єднаннях. В момент часу коли починається 1050-а ітерація циклу пошуку маршрутів мурахами-агентами відбуваються задані динамічні зміни вхідної мережі ЗК, які призводять до встановлення вартостей в знайденому маршруті R = {1 2 3 4 5 6 7 8 9 10 1} на значення 2000, що робить даний маршрут найбільшим за вартістю та відповідно найменш придатним для використання.

На рис.4.18 продемонстровано графік зміни сумарного значення цифрових міток (ордината на графіку) на з’єднаннях знайденого квазіоптимального маршруту R протягом запуску ітерацій циклу пошуку маршрутів мурахами-агентами (абсциса на графіку).

На рис.4.18 наведено дві послідовності: M(i) – без використання розробленого засобу, що базується на застосуванні адаптивної верхньої межі значення цифрової мітки – Mmax; M’(i) – з застосуванням адаптивної верхньої межі значення цифрової мітки Mmax, яка для даного випадку дорівнює 376. Як видно з графіку на рисунку запропонований метод дозволяє адаптуватись до динамічних змін навіть після тривалого періоду відсутності динамічних змін

вхідної мережі ЗК та зведенності вибору до єдиного знайденого квазіоптимального маршруту. Так вже після приблизно 200-х ітерацій з використанням Mmax було отримано новий квазі-оптимальний маршрут, без застосування розробленого засобу знадобилось приблизно 1500 ітерацій.

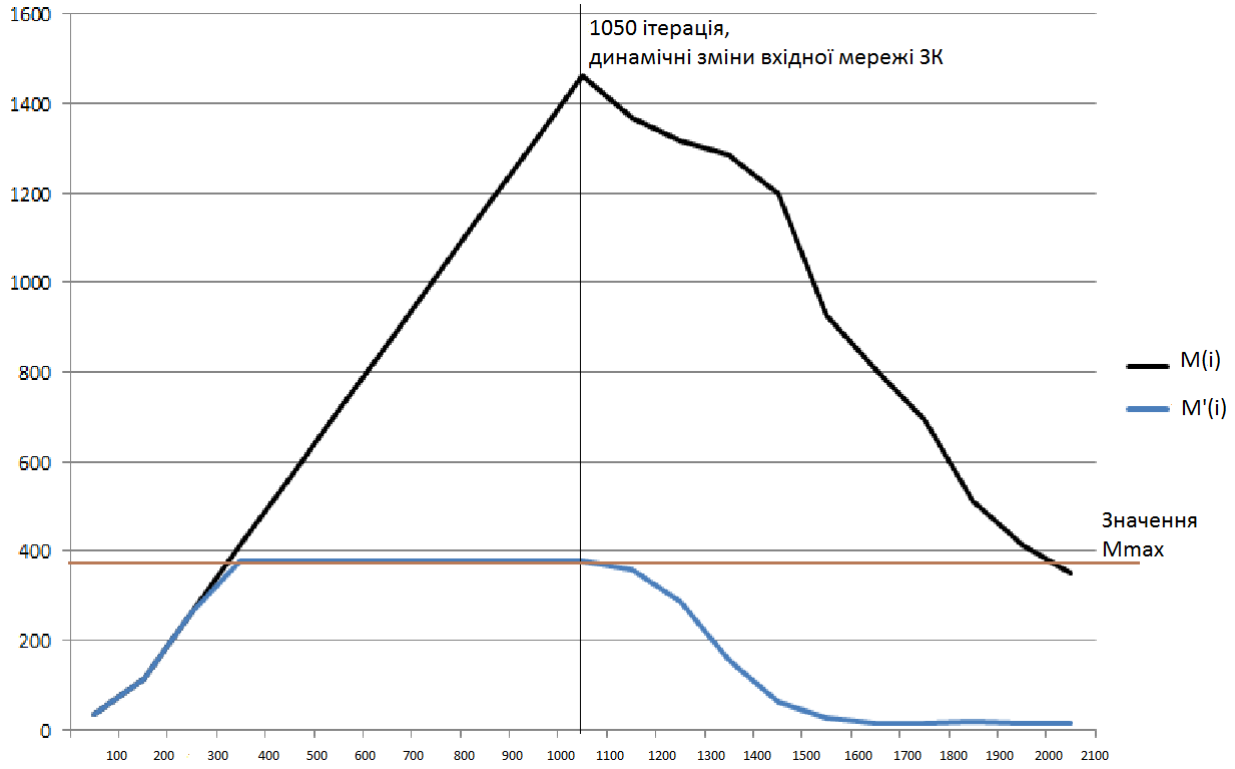


Рис.4.18. Графік зміни сумарного значення цифрових міток протягом обчислення ЗК

А при виконанні 5000 ітерацій та аналогічних динамічних змінах, для розробленої багатоагентної системи без застосування розробленого засобу подолання негативних наслідків надмірного укріплення «пам'яті колонії мурах» шляхом застосування адаптивної верхньої межі цифрової мітки Mmax, було неможливим отримати новий квазіоптимальний маршрут, агенти продовжували використовувати «найгірший», попередньо знайдений маршрут.

Результати розв'язання "mesh _task" ЗК протягом перших 2-х ітерацій циклу пошуку маршрутів мурахами-агентами (вмістиме файлу "mesh _task.res") представлено в Додатку І.

Як видно з представлених результатів, агенти починають сходитись до «кільцевого» послідовного проходження пунктів, маршрут «1 2 3 4 5 6 7 8 9 10

1», який включає в себе всі вузли, виконує умову одноразового проходження всіх вузлів та має меншу вартість ніж знайдені інші маршрути. Розглянемо більш детально хід подій в проміжок часу з 310 до 380 мс з метою аналізу стану мережі ЗК на даний період часу. Відповідний фрагмент з файлу подій має наступний вигляд (вмістиме файлу “mesh_task.log”):

310 -> action=Node to node send finish, agentId = 3, st_node = 9, end_node = 8, newValue = 0
310 -> action=Send all messages of node, agentId = 0, st_node = 5, end_node = 0, newValue = 0
320 -> action=Send all messages of node, agentId = 0, st_node = 8, end_node = 0, newValue = 0
340 -> action=Node to node send finish, agentId = 6, st_node = 8, end_node = 5, newValue = 0
350 -> action=Node to node send finish, agentId = 2, st_node = 5, end_node = 4, newValue = 0
350 -> action=Send all messages of node, agentId = 0, st_node = 5, end_node = 0, newValue = 0
360 -> action=Node to node send finish, agentId = 5, st_node = 4, end_node = 5, newValue = 0
360 -> action=Send all messages of node, agentId = 0, st_node = 4, end_node = 0, newValue = 0
370 -> action=Node to node send finish, agentId = 1, st_node = 5, end_node = 8, newValue = 0
370 -> action=Send all messages of node, agentId = 0, st_node = 5, end_node = 0, newValue = 0
380 -> action=Send all messages of node, agentId = 0, st_node = 8, end_node = 0, newValue = 0

Значення ліворуч в кожному рядку з файлу подій – значення часу коли трапляється подія (в даному прикладі в мс, загалом це може бути абстрактна величина). Далі йде опис події: «Node to node send finish» – завершення пересилання агента-пакета та його опрацювання на кінцевому вузлі, «Send all messages of node» – початок розсилання всіх агентів-пакетів з черги повідомлень, яка знаходиться на вузлі. Після опису події послідовно наведено наступні дані: ідентифікатор агента, якщо дія виконується над пакетом-агентом («agentId»); початковий вузел або вузел в якому відбувається подія («st_node»); кінцевий вузел («end_node»), наприклад для події пересилання агента-пакета; нове значення («newValue»), відноситься до подій змінення доступності або вартості з’єднання.

Як видно з представленої інформації, до вузла з ідентифікатором 5 надходять декілька агентів-пакетів, один з них в момент часу 360 мс успішно надходить з вузла з ідентифікатором 5 до вузла з ідентифікатором 4. З метою експеременту та перевірки працездатності розробленої системи в умовах динамічних змін внесемо декілька подій в сценарій.

Дані події (динамічні зміни) задаються в файлі сценарію подій “mesh_task.scn”. Внесемо наступні події:

1) В момент часу 355 (мс), після того як в момент часу 350 (мс) відбувається розсилання агентів-пакетів з вузла з ідентифікатором 5, з’єднання з вузла з ідентифікатором 5 до вузла з ідентифікатором 4 стає недоступним, вже в процесі передачі по ньому агента-пакета. Дане з’єднання знову стає доступним в момент часу 800 мс.
2) В момент часу 1160 (мс) змінемо вартість з’єднань так, щоб за вартістю оптимальним став протилежний маршрут до знайденого в 2-й ітерації, тобто агенти мають знайти новий маршрут: «1 10 9 8 7 6 5 4 3 2 1».
3) Крім того в момент часу 1200 (мс) відключається вузел з ідентифікатором 5.

Для реалізації описаних подій в файл “mesh_task.scn”. внесено наступні події:

```
SWITCH_OFF_ARC 355 5 4
SWITCH_ON_ARC 800 5 4
CHANGE_NODE_TO_NODE_TIME 1160 1 10 10
CHANGE_NODE_TO_NODE_TIME 1160 10 9 10
CHANGE_NODE_TO_NODE_TIME 1160 9 8 10
CHANGE_NODE_TO_NODE_TIME 1160 8 7 10
CHANGE_NODE_TO_NODE_TIME 1160 7 6 10
CHANGE_NODE_TO_NODE_TIME 1160 6 5 10
CHANGE_NODE_TO_NODE_TIME 1160 5 4 10
CHANGE_NODE_TO_NODE_TIME 1160 4 3 10
CHANGE_NODE_TO_NODE_TIME 1160 3 2 10
CHANGE_NODE_TO_NODE_TIME 1160 2 1 10
SWITCH_OFF_NODE 1200 5
```

Кожна подія в файлі сценарію складається з назви – типу події, далі йдуть час виконання події (в мс), ідентифікатор вузла джерела, ідентифікатор вузла призначення та нове значення (для події зміни вартості з’єднання).

Після повторного запуску багатоагентної системи вже з врахуванням даних подій з файлу сценарію, отримуються результати представлені в файлі “mesh _task2.res” (див. Додаток І). Як видно з результатів, протягом проходження вузлів агентами під час першої ітерації відключається з’єднання з

вузла з ідентифікатором 5 до вузла з ідентифікатором 4, через це жоден результуючий маршрут не включає дане з’єднання.

Перед початком виконання 2-ї ітерації циклу пошуку маршрутів мурахами-агентами відбувається зміна вартостей з’єднань, та відновлюється відключене з’єднання з вузла з ідентифікатором 5 до вузла з ідентифікатором 4. Як результат бачимо в отриманих маршрутах знову використовується дане з’єднання (маршрути агентів з ідентифікаторами 7 та 8). Крім того двома агентами був виявлений новий оптимальний маршрут, що виник після динамічних змін: «1 10 9 8 7 6 5 4 3 2 1», вартістю 100.

Перед початком виконання 3-ї ітерації відключається вузел з ідентифікатором 5. Як бачимо більшість агентів використали маршрут мінімальної вартості 160 – «1 10 9 8 7 6 4 3 2 1», в даному маршруті як і в іншому знайденому відсутній відключений вузел з ідентифікатором 5.

Наведені результати демонструють здатність розробленої системи пристосовуватись до виникнення динамічних змін в процесі розв’язання асиметричної ЗК в умовах частково невідомих даних без потреби перезапуску обчислень. Процес передачі даних за допомогою агентів-пакетів не припиняється, агенти швидко адаптуються до нових умов завдяки розробленим засобам.

Крім сітчастої топології було проаналізовано здатність розробленої багатоагентної системи розв’язувати ЗК на базі інших фізичних топологій мереж. Для демонстрації було вирішено використати невеликі мережі розмірністю до 10 вузлів, з’єднання між якими організовані відповідно до існуючих та застосовуваних в реальному житті топологій КМ.

На рис.4.19 представлено у вигляді графу вхідну мережу “full_top_task” ЗК, яка відповідає ***топології повного з’єднання.*** Кожен вузел з’єднаний з кожним іншим вузлом, що забезпечує високу надійність такого типу мереж, також в таких мережах є найбільша кількість варіантів можливих маршрутів, а також практично відсутній ризик виникнення ситуації «зациклення».

На рис.4.20 представлено вхідні матриці A та C для даної ЗК, розмірністю 10 вузлів. Початковий вузел – вузел з ідентифікатором 1.

Дана асиметрична ЗК була розв’язана практично вже після 1-ї ітерації циклу пошуку маршрутів мурахами-агентами. Результати розв’язання протягом 3-х перших ітерацій циклу представлено в Додатку І (вмістиме файлу “full_top_task.res”). Квазі-оптимальний результат: маршрут мінімальної вартості 640 «1 2 3 4 6 5 7 10 8 9 1» було знайдено вже протягом першої ітерації циклу, в 3-й ітерації більшість агентів почали користуватись знайденим маршрутом.

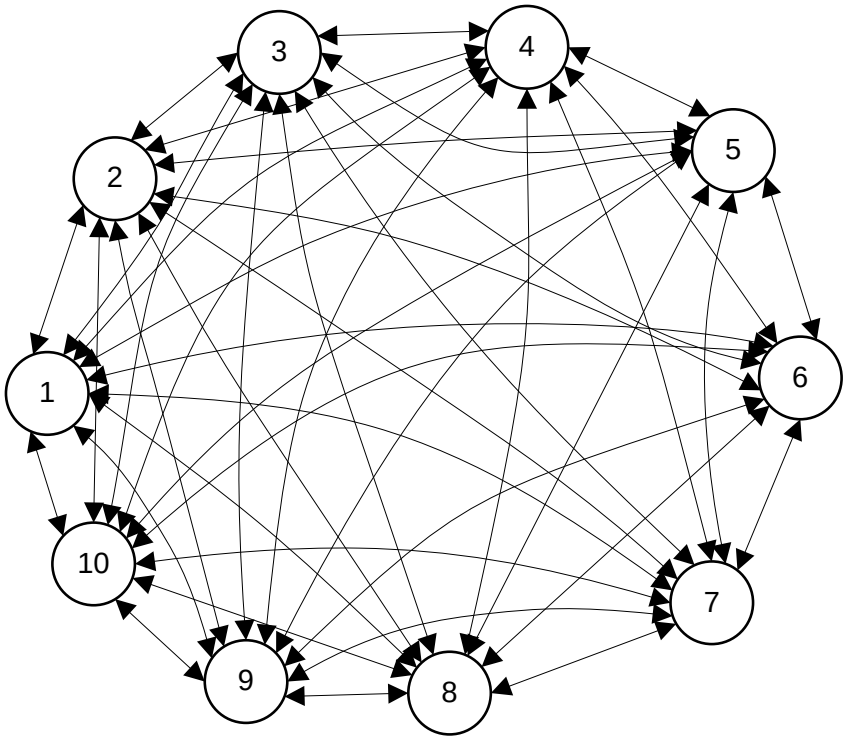


Рис.4.19. Графічне представлення вхідної мережі “full_top_task” ЗК

```
0 80 200 170 50 80 90 70 110 70
90 0 30 100 40 70 60 60 50 50
90 50 0 40 50 60 50 70 120 110
120 240 110 0 70 70 90 110 90 60
50 110 80 150 0 80 60 100 50 80
70 60 210 80 90 0 60 110 80 90
110 50 60 70 70 80 0 50 50 50
60 50 110 60 120 90 70 0 80 60
90 60 90 70 30 130 210 150 0 60
90 70 80 50 60 60 90 50 110 0
```

```
0 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 0 1 1
1 1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1 0
```

Рис.4.20. Вмістиме вхідних файлів для “full_top_task ” ЗК

При розв’язанні ЗК топології повного з’єднання, при умові відсутності відключень вузлів чи з’єднань гарантується отримання кожним агентом маршрутів одноразового проходження усіх вузлів.

На рис.4.21 представлено у вигляді графу вхідну мережу “cycle_top_task” ЗК, яка відповідає ***топології «кільця».*** Вузли підключаються до кабелю, замкнутого в коло, сигнал передається по колу, для даної асиметричної ЗК в двох напрямках, тобто всі з’єднання є двонаправленими.

На рис.4.22 представлено вхідні матриці А та С для даної ЗК, розмірністю 10 вузлів. Початковий вузел – вузел з ідентифікатором 1. При розв’язанні даної ЗК існує лише два можливих маршрути, через структуру вхідної мережі (див. рис. 4.21).

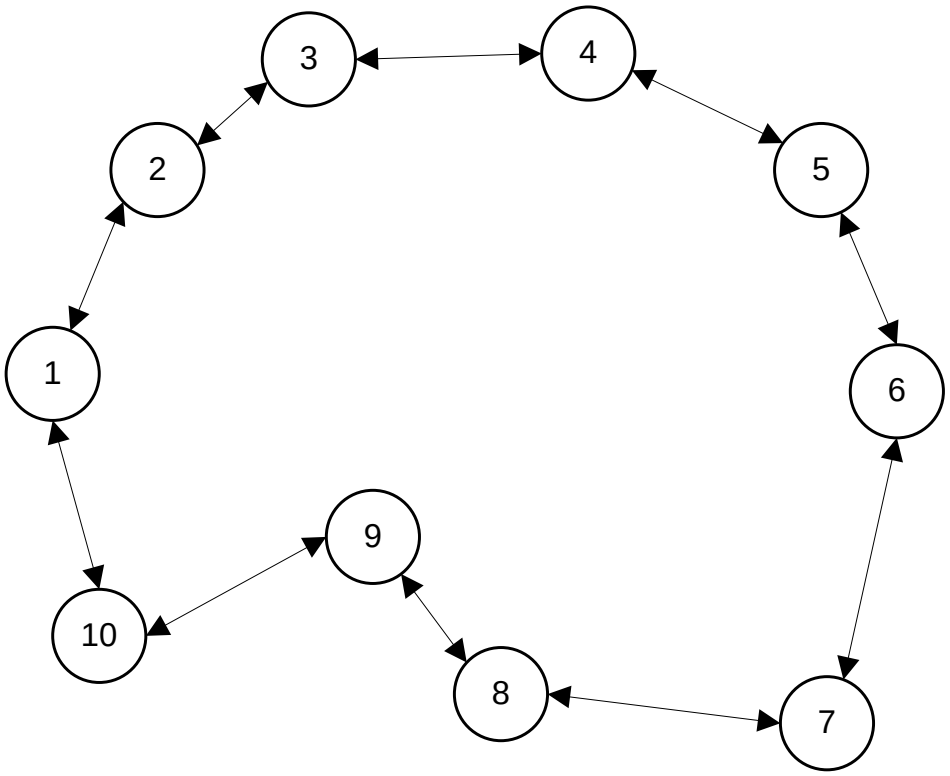


Рис.4.21. Графічне представлення вхідної мережі “ cycle_top_task” ЗК

Отримані результати протягом 3-х ітерацій циклу пошуку маршрутів мурахами-агентами при розв’язанні даної ЗК представлено в Додатку І (вмістиме файлу “cycle_top_task.res”). Для “cycle_top_task” ЗК існує два розв’язки, обидва маршрути були знайдені агентами вже на 1-й ітерації циклу

пошуку маршрутів мурахами-агентами: 1) «1 10 9 8 7 6 5 4 3 2 1» вартістю 879; 2) «1 2 3 4 5 6 7 8 9 10 1» вартістю 1483. Вже на 3-й ітерації циклу більшість агентів обирає маршрут меншої вартості.

```
0 120 0 0 0 0 0 0 0 150
130 0 130 0 0 0 0 0 0 0
0 150 0 400 0 0 0 0 0 0
0 0 127 0 150 0 0 0 0 0
0 0 0 125 0 330 0 0 0 0
0 0 0 0 19 0 67 0 0 0
0 0 0 0 0 81 0 58 0 0
0 0 0 0 0 0 71 0 198 0
0 0 0 0 0 0 0 15 0 14
16 0 0 0 0 0 0 0 11 0
```

```
0 1 0 0 0 0 0 0 0 1
1 0 1 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0
0 0 0 0 1 0 1 0 0 0
0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 1 0 1
1 0 0 0 0 0 0 0 1 0
```

Рис.4.22. Вмістиме вхідних файлів для “cycle_top_task” ЗК

При виникненні розриву з’єднання, тобто коли одне з з’єднань стає недоступним, вхідна мережа “cycle_top_task” ЗК відповідає **лінійній («ланцюговій») топології**. На рис.4.23 зображено вхідну мережу “chain_top_task” ЗК, що утворюється шляхом відключення доступності з’єднання між вузлами з ідентифікаторами 1 та 10 в “cycle_top_task” ЗК.

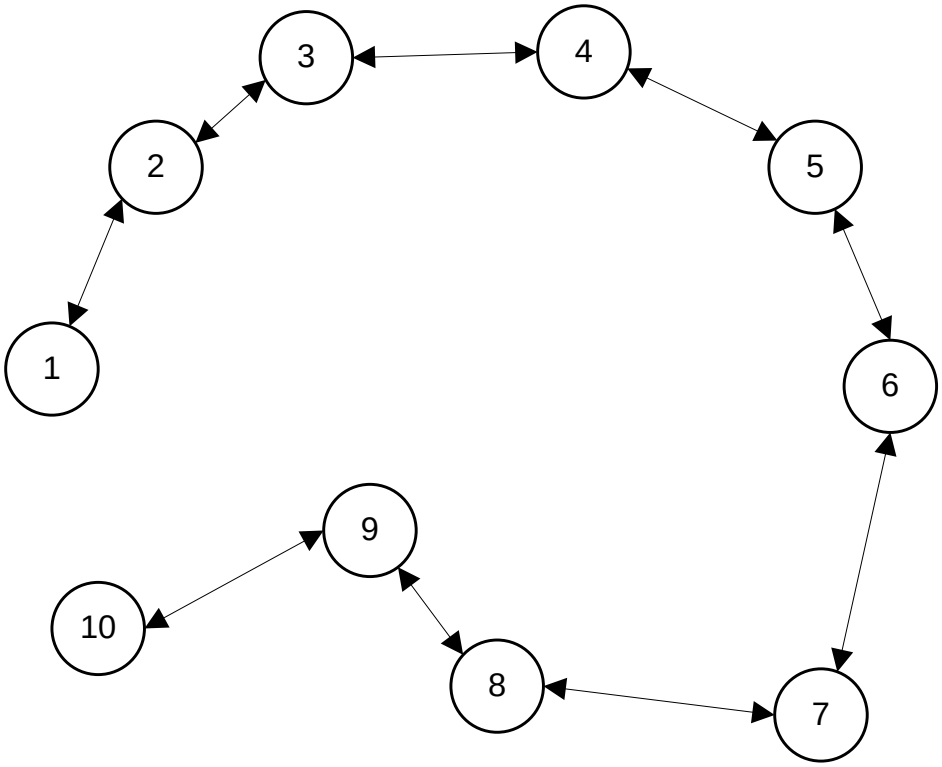


Рис.4.23. Графічне представлення вхідної мережі “chain_top_task” ЗК

Вхідна матриця вартостей C для “chain_top_task” ЗК аналогічна до матриці C у “cycle_top_task” ЗК (див. рис.4.22 зліва). Було розглянуто декілька ситуацій, пов’язаних з початковим розташуванням агентів:

1) Результати після 1-ї ітерації циклу, при початковому вузлі з ідентифікатором 1 (вмістиме файлу “chain_top_task.res”):

Route of [1] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1
Route of [2] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1
Route of [3] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1
Route of [4] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1
Route of [5] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1
Route of [6] agent : 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 2196 currently at MarkId = 1

2) Результати після 1-ї ітерації циклу, при початковому вузлі з ідентифікатором 5 (вмістиме файлу “chain_top_task2.res”):

Route of [2] agent : 5 6 7 8 9 10 9 8 7 6 5 || SPENT TIME = 864 currently at MarkId = 5 S = FINISHED
Route of [5] agent : 5 6 7 8 9 10 9 8 7 6 5 || SPENT TIME = 864 currently at MarkId = 5 S = FINISHED
Route of [6] agent : 5 6 7 8 9 10 9 8 7 6 5 || SPENT TIME = 864 currently at MarkId = 5 S = FINISHED
Route of [1] agent : 5 4 3 2 1 2 3 4 5 || SPENT TIME = 1332 currently at MarkId = 5 S = FINISHED
Route of [3] agent : 5 4 3 2 1 2 3 4 5 || SPENT TIME = 1332 currently at MarkId = 5 S = FINISHED
Route of [4] agent : 5 4 3 2 1 2 3 4 5 || SPENT TIME = 1332 currently at MarkId = 5 S = FINISHED

Як видно з результатів, отримати розв’язок ЗК при таких вхідних даних є неможливим, тому розв’язання зводиться до відвідування вузлів мережі без умови одноразового проходження. При будь-якому початковому розташуванні, агенти здатні виконати обходження існуючих вузлів та повернутись до початкового вузла.

Аналогічна проблема – неможливість розв’язання ЗК, виникає і при вхідних даних, що відповідають фізичним топологіям «дерева» або «зірки». Розглянемо розв’язання “star_top_task” ЗК, структура вхідної мережі якої відповідає **топології «зірка»**. На рис.4.24 зображено у вигляді графу вхідну мережу даної ЗК. Центральним вузлом «зірки» є вузел з ідентифікатором 5. Всі інші вузли з’єднані двосторонніми з’єднаннями з центральним вузлом «зірки».

На рис.4.25 зображено вхідні дані – матриці C (зліва) та A (справа) для “star_top_task” ЗК. Початковим вузлом для розташування агентів визначено вузел з ідентифікатором 1.

Нижче наведено результати отримані після 1-ї ітерації циклу пошуку маршрутів мурахами-агентами (“star_top_task.res”):

Route of [4] agent : 1 5 2 5 3 5 6 5 4 5 9 5 10 5 8 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Route of [5] agent : 1 5 3 5 2 5 4 5 6 5 9 5 10 5 8 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Route of [2] agent : 1 5 4 5 3 5 2 5 6 5 9 5 10 5 8 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Route of [1] agent : 1 5 3 5 6 5 2 5 4 5 9 5 8 5 10 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Route of [3] agent : 1 5 2 5 3 5 6 5 4 5 9 5 8 5 10 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Route of [6] agent : 1 5 3 5 4 5 6 5 2 5 8 5 9 5 10 5 7 5 1 || SPENT TIME = 5525 currently at MarkId = 1

Як видно з отриманих результатів кожен агент виконав відвідування всіх вузлів мережі ЗК, без умови одноразового проходження, що є неможливим при даних умовах ЗК.

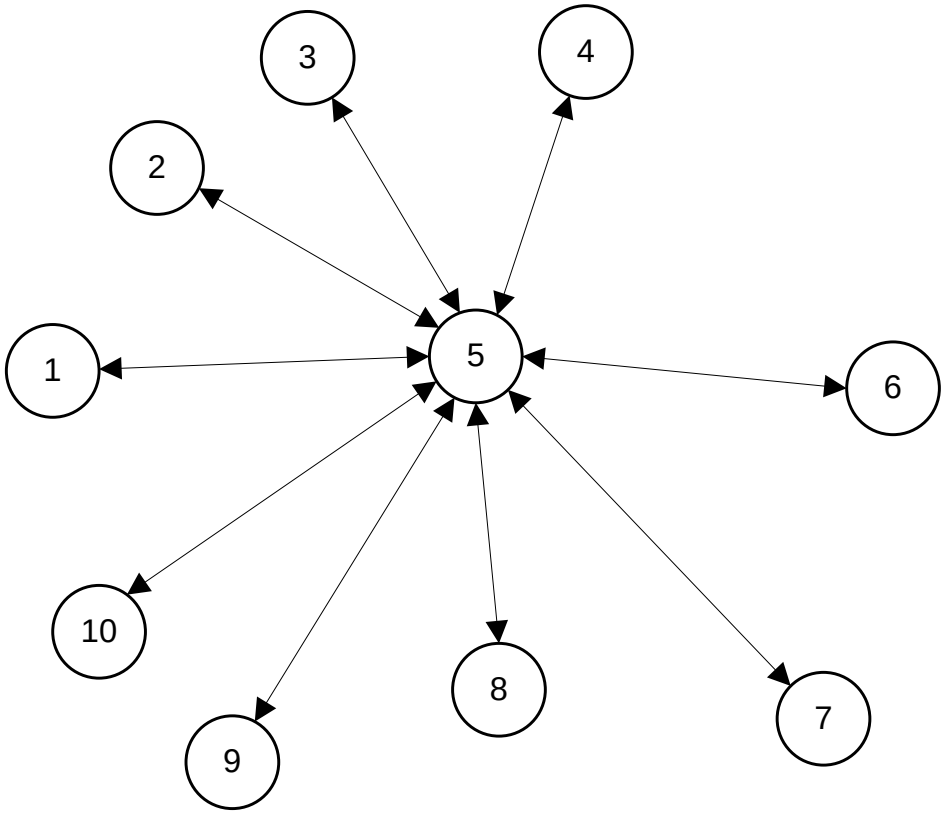


Рис.4.24. Графічне представлення вхідної мережі “star_top_task” ЗК

```
0 0 0 0 152 0 0 0 0 0
0 0 0 0 324 0 0 0 0 0
0 0 0 0 123 0 0 0 0 0
0 0 0 0 423 0 0 0 0 0
123 214 441 414 0 441 22 123 743 35
0 0 0 0 73 0 0 0 0 0
0 0 0 0 345 0 0 0 0 0
0 0 0 0 742 0 0 0 0 0
0 0 0 0 453 0 0 0 0 0
0 0 0 0 334 0 0 0 0 0
```

```
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1 1 1 1 0 1 1 1 1 1
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
```

Рис.4.25. Вмістиме вхідних файлів для “star _top_task” ЗК

Усі отримані маршрути мають однакову вартість через те, що включають в себе однаковий набір з'єднань. Розглянемо розв'язання "tree_top_task" ЗК, структура вхідної мережі якої відповідає **топології «дерево»**. На рис. 4.26 зображено у вигляді графу вхідну мережу даної ЗК. Всі з'єднання між вузлами є двосторонніми. На рис.4.27 зображено вхідні дані – матриці С (зліва) та А (справа) для "tree_top_task" ЗК. Було розглянуто декілька ситуацій, пов'язаних з початковим розташуванням агентів:

1) Результати після 1-ї ітерації циклу, при початковому вузлі з ідентифікатором 1 (вмістиме файлу "star_top_task.res"):

Route of [4] agent : 1 10 1 || SPENT TIME = 433 currently at MarkId = 1 STATUS = FINISHED

Route of [5] agent : 1 10 1 || SPENT TIME = 433 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 2 5 8 5 9 5 2 3 4 6 7 6 4 3 2 1 || SPENT TIME = 3203 currently at MarkId = 1

Route of [1] agent : 1 2 3 4 6 7 6 4 3 2 5 8 5 9 5 2 1 || SPENT TIME = 3203 currently at MarkId = 1

Route of [2] agent : 1 2 3 4 6 7 6 4 3 2 5 9 5 8 5 2 1 || SPENT TIME = 3203 currently at MarkId = 1

Route of [6] agent : 1 2 3 4 6 7 6 4 3 2 5 9 5 8 5 2 1 || SPENT TIME = 3203 currently at MarkId = 1

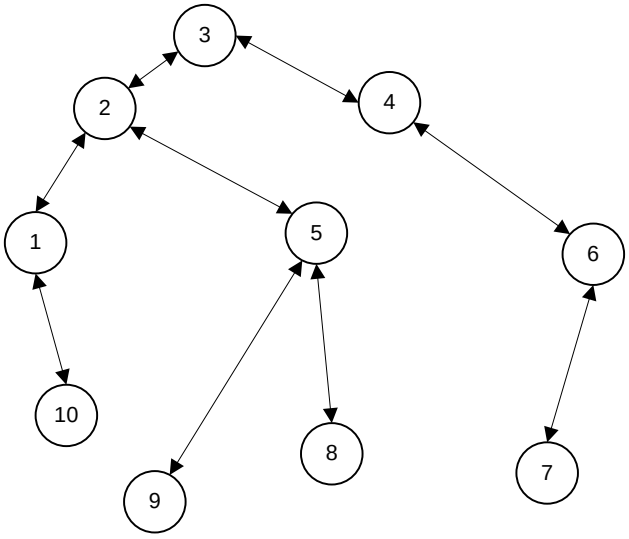


Рис.4.26. Графічне представлення вхідної мережі "star_top_task" ЗК

```
0 120 0 0 0 0 0 0 0 110
130 0 245 0 112 0 0 0 0 0
0 350 0 400 0 0 0 0 0 0
0 0 127 0 0 589 0 0 0 0
0 231 0 0 0 0 0 57 123 0
0 0 0 198 0 0 167 0 0 0
0 0 0 0 0 81 0 0 0 0
0 0 0 0 98 0 0 0 0 0
0 0 0 0 175 0 0 0 0 0
323 0 0 0 0 0 0 0 0 0
```

```
0 1 0 0 0 0 0 0 0 1
1 0 1 0 1 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0
0 1 0 0 0 0 0 1 1 0
0 0 0 1 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
```

Рис.4.27. Вмістиме вхідних файлів для "star _top_task" ЗК

2) Результати після 1-ї ітерації циклу, при початковому вузлі з ідентифікатором 3 (вмістиме файлу “star_top_task2.res”):

Route of [3] agent : 3 4 6 7 6 4 3 || SPENT TIME = 1562 currently at MarkId = 3 STATUS = FINISHED
Route of [4] agent : 3 4 6 7 6 4 3 || SPENT TIME = 1562 currently at MarkId = 3 STATUS = FINISHED
Route of [5] agent : 3 4 6 7 6 4 3 || SPENT TIME = 1562 currently at MarkId = 3 STATUS = FINISHED
Route of [6] agent : 3 4 6 7 6 4 3 || SPENT TIME = 1562 currently at MarkId = 3 STATUS = FINISHED
Route of [1] agent : 3 2 1 10 1 2 5 8 5 9 5 2 3 || SPENT TIME = 2074 currently at MarkId = 3
Route of [2] agent : 3 2 1 10 1 2 5 9 5 8 5 2 3 || SPENT TIME = 2074 currently at MarkId = 3

Як видно з представлених результатів, отримати розв’язок ЗК є неможливим. Колектив агентів здатен відвідати усі вузли, шляхом утворення декількох маршрутів обходження, які сумарно покривають усі існючі вузли “star_top_task” ЗК. При будь-якому початковому розташуванні в мережі побудованій на базі топології «дерева», агенти здатні виконати обходження існуючих, доступних в «гілці дерева», вузлів та повернутись до початкового вузла.

4.3. Впровадження результатів роботи

Основні результати дисертаційної роботи було впроваджено в науково-дослідну роботу «Інтеграція методів і засобів вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем», шифр ДБ/КІБЕР, реєстраційний номер №0115U000446, Національного університету «Львівська політехніка» (див. Додаток А) за науковим напрямом «Дослідження та розроблення принципів побудови засобів збору та доставки інформації кіберфізичних систем на основі автономних вимірювально-обчислювальних вузлів з використанням багатоагентних технологій». Наукові результати, які було використано в ході науково-дослідної роботи: розроблена багатоагентна система з використанням поведнікової моделі колонії мурах; розроблені засоби, що базуються на запропонованому методі виявлення та виходу мурахи-агента з критичних ситуацій «пастки» та «зациклення»; розроблений метод подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток (пам’яті колонії мурах); розроблений метод оновлення значень цифрових

міток в процесі проходження сполучення між вузлами; розроблений новий засіб організації функціонування системи в режимі поєднання процесу пошуку маршрутів з процесом передачі даних, який дозволив забезпечити розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних.

Результати проведених досліджень використано при розробці спеціалізованих обчислювальних засобів в міжнародній компанії “Eleks” (Львів, Україна) (див. Додаток В). Зокрема було використано програмну реалізацію розробленої багатоагентної системи з застосуванням методів локальної оптимізації та технологій паралельних обчислень для розв'язання динамічної задачі комівояжера. Впровадження результатів дисертаційного дослідження дозволило суттєво збільшити швидкодію обчислювальних засобів при розв’язанні динамічної задачі комівояжера, з різницею від оптимального результату не більше 10% для задач розмірністю до 10 000 вузлів.

Розроблені засоби та моделі багатоагентних систем для розв’язання динамічної ЗК, в тому числі динамічної асиметричної ЗК в умовах частково невідомих даних, були використані співробітниками кафедри електронних обчислювальних машин Інституту комп’ютерних технологій автоматики та метрології Національного університету «Львівська політехніка» при підготовці методичних матеріалів до лабораторних робіт “Поведінкова модель на базі алгоритму мурашиної колонії”, “Організація паралельних обчислювальних процесів засобами OpenMP” та викладанні навчальних курсів: «Теорія інтелектуальних систем» для освітньо-кваліфікаційного рівня «Магістр»; «Організація обчислювальних процесів у паралельних системах», для освітньо-кваліфікаційного рівня «Магістр»; при підготовці бакалаврських та магістерських кваліфікаційних робіт (див. Додаток Г).

Розроблені методи для розв’язання динамічної ЗК було використано при розробці оптимізаційних рішень в міжнародній компанії “Logivations GmbH” (Гребенцель, Німеччина) (див. Додаток Б). Оптимізаційний сервіс компанії “Logivations GmbH” доступний як веб аплікація, надає широкий набір засобів для контролю та розрахунків по складській логістиці: керування та розрахунок

потоку замовлень, розрахунку витрат людино-годин, оптимізації розміщення товарів, розрахунок послідовності транспортування, проектування нових та покращення організації існуючих складів, а також симуляція виконання замовлень від етапу постачання товарів до етапу пакування та відправки до кінцевого замовника.

Задача розрахунку витрат людино-годин для відправки замовлень зі складу при кількості робітників більше сотні, які «подорожують», збираючи товари, по приміщенню складу, потребує ефективного розв'язку динамічної ЗК. Запропоновані в дисертаційній роботі методи та засоби для розв’язання динамічної ЗК було використано при розробці симуляційних програмних засобів оптимізаційного сервісу компанії “Logivations GmbH”, що дозволило забезпечити кращу адаптацію до динамічних змін та мінімізацію людино-годин.

4.4. Висновки до розділу

У цьому розділі проведено тестування розроблених багатоагентних систем, оцінювання ефективності запропонованих методів, засобів та розроблених моделей багатоагентних систем. Отримано такі результати:

- застосування розробленої модифікації алгоритму колонії мурах дозволяє зменшити час отримання результуючого маршруту у майже 4 рази для ЗК розмірністю до 1000 вузлів та на 20% для ЗК розмірністю до 10000 вузлів;

- розроблена багатоагентна система здатна розв’язувати динамічну ЗК без потреби перезапуску процесу обчислення при виникненні змін вхідних даних. В процесі розв’язання симетричної ЗК розмірністю 532 вузли розроблена система здатна видати результат з різницею від оптимального, що складає приблизно 10%, вже за 1 секунду та оптимальний результат, за відсутності динамічних змін, через 5 с при використанні одноядерної ЕОМ; та за час 190 мс і 880 мс відповідно при використанні 8-ми ядерної ЕОМ;

- розроблена багатоагентна система з використанням поведінкової моделі колонії мурах, технологій паралельних обчислень та методу опрацювання результуючого маршруту, який базується на застосуванні алгоритмів локальної

оптимізації 2-opt, 2.5-opt, 3-opt в залежності від інтенсивності динамічних змін вхідних даних, здатна видати оптимальний результат або квазі-оптимальний результат з різницею від оптимального не більше ніж 2% для симетричних та асиметричних ЗК з кількістю вузлів до 6000;

- розроблена багатоагентна система для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних забезпечує знаходження результуючих маршрутів ЗК при зв’язках вхідної мережі, що відповідають будь-якій фізичній топології. Якщо однократне проходження вузлів неможливе, забезпечується обходження усіх вузлів та повернення агентів в початковий вузел. Розроблена система здатна функціонувати в умовах частково невідомих вхідних даних та адаптуватись до динамічних змін вхідних даних в процесі розв'язання ЗК без потреби перезапуску ітерації циклу пошуку маршрутів мурахами-агентами.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальне наукове завдання – підвищення ефективності розв’язання динамічної ЗК шляхом вдосконалення існуючих та розробки нових методів, моделей та засобів, що базуються на використанні поведінкової моделі колонії мурах в багатоагентних системах. Отримано такі наукові та практичні результати:

1. Розроблено класифікацію існуючих методів розв’язання динамічної ЗК, аналіз якої показав, що при великій кількості пунктів (N > 100) потрібно орієнтуватись на методи розв’язання із застосуванням колективної поведінки агентів, серед яких найбільш перспективний – метод колонії мурах. Проведено дослідження можливості застосування поведінкової моделі колонії мурах для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних.

2. Удосконалено метод розв’язання задачі комівояжера, який базується на використанні поведінкової моделі колонії мурах, шляхом зміни початкової установки значень міток та імовірнісного вибору наступного вузла для переходу мурахи-агента, що дозволило зменшити кількість ітерацій циклу пошуку маршрутів агентами та відповідно час розв’язання ЗК у майже 4 рази при розмірності до 1000 вузлів та на 20% при розмірності до 10000 вузлів.

3. Вперше розроблено метод опрацювання результуючого маршруту при розв’язанні динамічної задачі комівояжера, який базується на використанні алгоритмів локальної оптимізації 2-opt, 2.5-opt, 3-opt в залежності від інтенсивності динамічних змін вхідних даних, що дозволило зменшити вартість результуючих маршрутів.

4. Розроблено багатоагентну систему для розв’язання динамічної ЗК. Система базується на використанні поведінкової моделі колонії мурах, технологій паралельних обчислень та методу опрацювання результуючого маршруту. Система здатна видати оптимальний результат або квазі-оптимальний результат з різницею від оптимального не більше, ніж 2% для симетричних та асиметричних ЗК з кількістю вузлів до 6000.

5. Вперше розроблено метод і отримано експериментальні результати подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток (пам’яті колонії мурах), який базується на використанні адаптивної верхньої межі значення цифрової мітки, що дозволило розробленій багатоагентній системі відновити пошук маршрутів меншої вартості навіть після тривалого статичного стану вхідних даних.

6. Вперше розроблено та апробовано модель багатоагентної системи, яка базується на використанні поведінкової моделі колонії мурах при розміщенні цифрових міток на комунікаційних вузлах, що дозволило розв’язати динамічну асиметричну задачу комівояжера в умовах частково невідомих вхідних даних.

7. Розроблено багатоагентну систему для розв’язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних з кількістю вузлів до 65536. Система базується на використанні розробленого методу подолання виявлених негативних наслідків нескінченного збільшення значень цифрових міток, розробленого методу виявлення та виходу мурахи-агента з критичних ситуацій «пастки» та «зациклення», запропонованого методу оновлення значень цифрових міток в процесі проходження сполучення між вузлами, запропонованого засобу організації функціонування системи в режимі поєднання процесу пошуку маршрутів з процесом передачі даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ананий Левитин. Алгоритмы. Введение в разработку и анализ (2-е издание). – Пирсон, 2012 — 565 с.

2. Андон П.І., Ігнатенко О.П. Протидія атакам на відмову в мережі інтернет: концепція підходу // Проблеми програмування, № 2-3. – Київ. – 2008. — с. 564 – 574.

3. Базилевич Р., Кутельмах Р. Дослідження ефективності існуючих алгоритмів для розв’язання задачі комівояжера // Вісник НУ “Львівська політехніка”. – 2009. – № 650: Комп’ютерні науки та інформаційні технології. – с. 235 – 245.

4. Белоусов А.А., Бердышев Ю.И., Ченцов А.Г., Чикрий А.А. О решении игровой задачи динамического коммивояжера // Кибернетика и системный анализ, № 5. — 2010. — с. 40 – 45.

5. Бочкарев А.Ю., Голембо В.А. Самоорганизация коллектива мобильных измерительных агентов в задаче распределенных контактных измерений // Штучний інтелект, №3. – Донецьк. – 2005. – с. 723 – 231.

6. Вейтман В. Разговор о маршрутизации не окончен // Компьютерное обозрение, №27 (546), июль 2009.

7. Глибовець М.М., Гулаєва Н.М. Еволюційне програмування // Проблеми програмування, № 4. — 2013. — № 4. — с. 3 – 13.

8. Голембо В.А., Бочкарьов О.Ю., Муляревич О.В. Нові підходи до розв’язку задач комбінаторної оптимізації колективом автономних агентів // Матеріали 5-ої Міжнародної науково-технічної конференції ACSN-2011 "Сучасні комп'ютерні системи та мережі: розробка та використання". — Львів. — 2011. — с. 227 – 230.

9. Голембо В.А., Муляревич О.В. Модифікація методу мурашиної колонії для розв’язання задачі комівояжера колективом автономних агентів // Вісник Національного університету "Львівська політехніка". – 2011. – №717: Комп’ютерні системи та мережі. – с. 24 – 30.

10. Гриценко В.И., Гладун А.Я., Журавлев Ю.Д., Несен М.В. Модель мультиагентной системы для е-бизнеса и технология ее программной реализации // Проблеми програмування, № 2-3. — 2004. — с. 510 – 519.

11. Данчук В.Д., Сватко В.В., Оптимізації пошуку шляхів по графу в динамічній задачі комівояжера методом модифікованого мурашиного алгоритму // Системні дослідження та інформаційні технології, №2, 2012.— с. 78 – 86.

12. Добронравин Ю.В. Структура программы интеллектуальных агентов, решающих несколько задач одновременно // Збірник наукових праць Інституту проблем моделювання в енергетиці ім.Г.Є.Пухова НАН України. — К.: ІПМЕ ім. Г.Є.Пухова НАН України, 2009. — Вип. 53. — с. 21 – 27.

13. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення: ДСТУ 3008-95.

14. Желдак Т.А. Застосування методу моделювання колонії мурах до розв’язання комбінаторних задач планування виконання замовлень металургійними підприємствами // Математичні машини і системи, № 4. — Київ. – 2013. — с. 95 – 106.

15. Зайко Т.А., Олійник А.О., Субботін С.О. Побудова нейро-нечітких моделей на основі неструктурованих даних // Штучний інтелект, №4. — Донецьк. – 2012. — с. 546 – 556.

16. Згуровский М.З., Павлов О.А. Принятие решений в сетевых системах с ограниченными ресурсами . – К.: Наукова думка, 2010. – 576 с.

17. Інформація та документація. Скорочення слів в українській мові та бібліографічному описі. Загальні вимоги та правила: ДСТУ 3582-97.

18. Карпенко А.П. Популяционные алгоритмы глобальной оптимизации. Обзор новых и малоизвестных алгоритмов // Приложение к журналу «Информационные технологии», № 7. – 2012. – с. 1 – 32.

19. Козуб А.Н., Кучеров Д.П. Интеграционный подход к задаче выбора маршрута группы БПЛА // Штучний інтелект, №4. – Донецьк — 2013. — с. 333 – 343.

20. Лафоре Р. Объектно-ориентированное программирование в С++. Основы алгоритмизации и программирования, – 4-е изд., – СПб.: Питер, 2011. – 928 с.

21. Литвинов В.В., Задорожний А.А. Агентные модели операций // Математичні машини і системи. – Київ. — 2014. — № 1. — с. 114 – 121.

22. Матренин П.В., Секаев В.Г. Оптимизация адаптивного алгоритма муравьиной колонии на примере задачи календарного планирования // Програмна інженерія, № 4. – 2013. – с. 34 – 40.

23. Морозов А.В., Панішев А.В. Метод гілок та меж у гальмітоновій задачі про сільського листоношу // Системи дослідження та інформаційні технології, № 2. — 2012. — с. 57 – 66.

24. Муляревич О. Переваги застосування колективної поведінки агентів для розв’язку задачі комівояжера динамічного характеру // Тези доповідей студентської секції «Кібер фізичні системи в метрології» IX Міжнародної науково-технічної конференції «Методи і засоби вимірювань фізичних величин» - «Температура-2012», 25-28 вересня 2012р. – Львів: ПП Сорока Т.Б., 2012. – с. 165 – 168.

25. Муляревич О.В., Голембо В.А. Вирішення проблем при розв'язанні динамічної асиметричної задачі комівояжера в умовах частково невідомих вхідних даних // Науковий вісник Чернівецького університету. - 2015. Комп'ютерні системи та компоненти. Т.6, Вип.1. – с. 21 – 26.

26. Муляревич О.В., Голембо В.А. Імплементація методів локальної оптимізації у комп’ютерній системі для розв’язання динамічної задачі комівояжера з використанням моделі ройової поведінки агентів // Вісник НУ "Львівська політехніка". – 2013. – №771: Комп’ютерні науки та інформаційні технології. – с. 245 – 252.

27. Муляревич О.В., Голембо В.А. Розробка додаткового програмного модуля з використанням методів локальної оптимізації у комп’ютерній системі для розв’язання динамічної задачі комівояжера // Вісник Національного університету "Львівська політехніка". – 2014. – №806: Комп’ютерні системи та мережі. – с. 181 – 186.

28. Наукові прориви 2014 року за версією журналу Science // Вісн. НАН України. — 2015. — № 2. — с. 104 – 111.

29. Національна стандартизація. Правила побудови, викладення, оформлення та вимоги до змісту нормативних документів: ДСТУ 1.5:2003.

30. Олейник А.А. Мультиагентный метод оптимизации с адаптивными параметрами // Штучний інтелект, № 1. — 2011. — с. 83 – 90.

32. Олейник А.А., Субботин С.А. Мультиагентный метод с непрямой связью между агентами для выделения информативных признаков // Штучний інтелект, №4. — Донецьк. — 2009. — с. 75 – 82.

33. Олейник А.А., Субботин С.А., Гофман Е.А. Эволюционный метод синтеза деревьев решений, // Штучний інтелект, № 2. — Донецьк. – 2011. — с. 6 – 14.

34. Олейник Д.В., Шинкаренко В.И. Мультиагентная адаптация гибридного генетического алгоритма для обучения нейросетей // Штучний інтелект, № 4. — — Донецьк. – 2008. — с. 463 – 470.

35. Оре О. Графы и их применение. Пер. с англ. под ред. И.М. Яглома. – М.:Мир, 1965. – 174 с.

36. Основні одиниці фізичних величин Міжнародної системи одиниць: ДСТУ 3651.0-97.

37. Парасюк И.Н., Ершов С.В. Нечеткие модели мультиагентных систем в распределенной среде // Проблеми програмування, № 2-3. — 2010. — с. 330 – 339.

38. Пахомова В.М., Міщанюк Л.О. Інтелектуальна підсистема вибору раціональних маршрутів вантажних потягів // Штучний інтелект, №1. — Донецьк. — 2014. — с. 119 – 125.

39. Потопахин В.В. Искусство алгоритмизации. - М.: ДМК Пресс, 2011. – 320 с.

40. Ручкин К.А., Данилов А.В. Разработка многоагентной системы для прогнозирования поведения динамической системы в режиме реального времени // Штучний інтелект, № 4. – Донецьк — 2011. — с. 449 – 460.

41. Субботін С.О., Олійник А.О., Олійник О.О. Неітеративні, еволюційні та мультиагентні методи синтезу нечіткологічних і нейромережних моделей: Монографія / Під заг. ред. С.О. Субботіна. – Запоріжжя: ЗНТУ, 2009. – 375 с.

42. Тимофієва Н.К. Про подібність задач комбінаторної оптимізації та універсальність алгоритмів // Системні дослідження та інформаційні технології, № 4. — 2013. — с. 27 – 37.

43. Черноруцкий И.Г. Методы оптимизации. Компьютерные технологии. – СПб.: БХВ-Петербург, 2011. – 384 с.

44. Чураков М., Якушев А. Муравьиные алгоритмы // електронний ресурс, режим доступу: http://rain.ifmo.ru/cat/data/theory/unsorted/ant-algo-2006/article.pdf. — 2006. – 15 с.

45. Шуть В.Н. Мультиагентное управление движением транспортных средств в улично-дорожной сети города // Штучний інтелект, № 4 (66). – Донецьк. — 2014. — с. 123 – 128.

46. Юдин Д.Б., Гольштейн Е.Г. Линейное программирование: Теория, методы и приложения. – М.: КРАСАНД, 2012. – 424 с.

47. Юдин Д.Б., Гольштейн Е.Г. Задачи и методы линейного программирования: Конечные методы. – М.: КРАСАНД, 2010. – 264 с.

48. Aghajarian M., Kiani K., Fateh M. Design of Fuzzy Controller for Robot Manipulators Using Bacterial Foraging Optimization Algorithm // Journal of Intelligent Learning Systems and Applications, Vol. 4, № 1. – 2012. – pp. 53 – 58.

49. Agrawal K., Begoria R. “Ant Colony Optimization: Efficient Way To Find The Shortest Path” // ISSN No: 2250-3536 – IJTATER, Vol. 4 (2014), Iss. 3, pp.18-21.

50. AntNet by Di Caro G. A.. URL:http://www.idsia.ch/~gianni/antnet.html

51. Arunkumar G., Gnanambal D. Utilization of Bacterial Foraging Algorithm for Optimization of Boost Inverter Parameters // Circuits and Systems, Vol.7, № 8. – 2016. – pp. 1430 – 1440.

52. Askali M., Azouaoui A., Nouh S. and Belkasmi M. On the Computing of the Minimum Distance of Linear Block Codes by Heuristic Methods // International

Journal of Communications, Network and System Sciences, Vol. 5, № 11. – 2012. – pp. 774 – 784.

53. Barach G., Fort H., Mehlman Y., Zypman F. Information in the Traveling Salesman Problem // Applied Mathematics, Vol. 3, № 8. – 2012. – pp. 926 – 930.

54. Battiti, Roberto, Mauro Brunato, Franco Mascia Reactive Search and Intelligent Optimization, Springer Verlag, 2008. – 182 p.

55. Blum C., Roli A. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison // ACM Computing Surveys. – Vol.35, № 3, 2003. – pp. 268 – 308.

56. Bonabeau E., Dorigo M., Theraulaz G. Swarm intelligence: From Natural to Artificial Systems. – Oxford University Press, 1999. – 320 p.

57. Brezina I., Čičková Jr. Z. “Solving the Travelling Salesman Problem Using the Ant Colony Optimization” // Publ.: University of Novi Sad – Management Information Systems, Vol. 6 (2011), No. 4. – pp. 10 – 14.

58. Brightwell G.R., Winkler P., Note on Counting Eulerian Circuits, May, CDAM Research Report LSE-CDAM-2004-12, 2004. – 7 p.

59. Changhe Li, Ming Yang, Lishan Kang, A New Approach to Solving Dynamic Traveling Salesman Problems, - Simulated Evolution and Learning: 6th International Conference, SEAL 2006. – pp. 236 – 243.

60. Chen T., Guo W., Gao Z. Artificial Searching Swarm Algorithm and Its Performance Analysis // Applied Mathematics, Vol. 3, № 10A. – 2012. – pp. 1435 – 1441.

61. Cheremisinov D.I. The real difference between linear and branching temporal logics // Workshop on Discrete-Event System Design DESDes’04, University of Zielona Gora Press, Poland. – 2004. – pp. 103 – 108.

62. Cheremisinov D.I., Cheremisinova L. Specifying agent interaction protocols with Parallel control algorithms // Proceedings of 11th International Conference of Knowledge-Dialog-Solution (KDS’05), Varna, Bulgaria. – 2005. – pp. 496 – 503.

63. Dai J., Ni L., Wang X., Chen W. “A VaR Algortihm for Warrants Portfolio” // Algorithmic Aspects in Information and Management: 6th International Conference, AAIM 2010, Springer Verlag Publ.. – pp. 103 – 111.

64. David L. Applegate,Robert E. Bixby,Vasek Chvátal,William J. Cook “The Traveling Salesman Problem: A Computational Study”, Princeton University Press, 2011. – 583 p.

65. Di Caro G. A. Ant Colony Optimization and its application to adaptive routing in telecommunication networks, PhD thesis in Applied Sciences, Polytechnic School, Université Libre de Bruxelles, Brussels, Belgium, 2004. – 374 p.

66. Dorigo M., Dario Floreano, Luca M Gambardella and others, “Swarmanoid: A Novel Concept for the Study of Heterogeneous Robotic Swarms”, // IEEE Robotics & Automation Magazine, Vol.20, Iss.4, 2013. – pp. 60 – 71.

67. Dorigo M., Stützle T. Ant Colony Optimization // Cambridge, MA: MIT. Press / Bradford Books. – 2004. – 321 p.

68. Ducatelle F., Di Caro G.A., Gambardella L.M., Principles and applications of swarm intelligence for telecommunications networks, Swarm Intelligence Journal Vol. 4, N. 3, 2010. – pp. 173 – 198.

69. Eiben A.E., Smith J.E. “Introduction to Evolutionary Computing”, Second Edition, Springer, 2015. – 283 p.

70. Emelyanov V.V., Kureychyk V.V., Kureychyk V.M. The theory and practice of evolutionary modeling. – M.: Fizmatlit, 2003. – 432 p.

71. Gendreau M., Potvin J.-Y. “Handbook of Metaheuristics”, Second edition, Springer Verlag, 2010. – 641 p.

72. Groth D., Skandier T. “Network+”. – Study Guide. Fourth Edition, Sybex Inc., 2005. – 491 p.

73. Helsgaun K. General k-opt submoves for the Lin-Kernighan TSP heuristic // Mathematical Programming Computation, 2009. – pp. 119 – 163.

74. Hokamoto and J. Murakami. Genetic-algorithm-based randezvous trajectory design for multiple active debris removal. In 28th International Symposium on Space Technology and Science, 2011. – 26 p.

75. Hussain S., Islam O. Genetic Algorithm for Energy Efficient Trees in Wireless Sensor Networks // In Advanced Intelligent Environments, Springer: Boston, MA, USA. – 2008. – pp. 1 – 14.

76. Kautz H., Selman B. Pushing the envelope: planning, propositional logic, and stochastic search. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), 1996. – pp. 1194 – 1201.

77. Kiran M., Gündüz M. The Analysis of Peculiar Control Parameters of Artificial Bee Colony Algorithm on the Numerical Optimization Problems // Journal of Computer and Communications, Vol.2, No.4. – 2014. – pp. 127 – 136.

78. Krishna H. Hingrajiya, Ravindra Kumar Gupta, Gajendra Singh Chandel “An Ant Colony Optimization Algorithm for Solving Travelling Salesman Problem” // ISSN 2250 - 3153 – International Journal of Scientific and Research Publications, Vol. 2 (2012), Iss. 8. – pp. 293 – 299.

79. Krishna H. Hingrajiya, Ravindra Kumar Gupta, Gajendra Singh Chandel “An Approach for Solving Multiple Travelling Salesman Problem using Ant Colony Optimization” // ISSN 2222-2863 – Computer Engineering and Intelligent Systems, IISTE, Vol. 6 (2015), No. 2. – pp. 13 – 17.

80. Lesser V. Cooperative multiagent systems: A personal view of the state of the art // IEEE Transactions of Knowledge and Data Engineering, Vol. 11, No. 1. – 1999. – pp. 133 – 142.

81. Li X., Xu H., Guan X. Quantum-Inspired Particle Swarm Optimization Algorithm Encoded by Probability Amplitudes of Multi-Qubits // Open Journal of Optimization, Vol.4 No.2. – 2015. – pp. 21 – 30.

82. Liu Z., Jlang B. The Design of the Minimum Spanning Tree Algorithms // Intelligent Information Management, Vol. 1, № 1. – 2009. – pp. 56 – 59.

83. LKH Keld Helsgaun official page: http://www.akira.ruc.dk/~keld/research/LKH/

84. Marsaglia G., Tsang W. W., Wang Jingbo "Fast Generation of Discrete Random Variables". – Journal of Statistical Software 11 (3), 2004. – pp. 1 – 11.

85. Martinez-Zeron E., Aceves-Fernandez M., Gorrostieta-Hurtado E., Sotomayor-Olmedo A., Ramos-Arreguín J. Method to Improve Airborne Pollution Forecasting by

Using Ant Colony Optimization and Neuro-Fuzzy Algorithms // International Journal of Intelligence Science, Vol.4, №4. – 2014. – pp. 81 – 90.

86. Martino G. Solving a Traveling Salesman Problem with a Flower Structure // Journal of Applied Mathematics and Physics, Vol. 2, № 7. – 2014. – pp. 718 – 722.

87. Melnyk A., Golembo V., Bochkaryov A. Multiagent approach to the distributed autonomous explorations // Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2007), Edinburgh, UK. – 2007. – pp. 606 – 610.

88. Meng K., Dong Z.Y., Qiao Y.C. Swarm Intelligence in Power System Planning // International Journal of Clean Coal and Energy, Vol. 2, № 2B. – 2013. – pp. 1 – 7.

89. Mousa A., El-Shorbagy M. Enhanced Particle Swarm Optimization Based Local Search for Reactive Power Compensation Problem // Applied Mathematics, Vol. 3, № 10A. – 2012. – pp. 1276 – 1284.

90. Muliarevych O., Golembo V. New approaches for solving travelling salesman problems using agents swarm intelligence behavior model // European Cooperation: Scientific Approaches and Applied Technologies. - CLM Consulting Publ., Warsaw 2015, Vol 5(5). – pp. 131 – 143.

91. Muliarevych O. Solving dynamic asymmetrical Travelling Salesman Problem in conditions of partly unknown data // Матеріали XIII-ої міжнародної конференції “Сучасні проблеми радіоелектроніки, телекомунікацій, комп'ютерної інженерії — TCSET'2016, 23 - 26 лютого 2016. — Львів - Славське: Видавництво Львівської політехніки, 2016. — с. 446 – 448.

92. Neil Gershenfeld, The Nature of Mathematical Modeling. – First Edition, Section 5.3.2: Linear Feedback. – Cambridge University Press, 1999. – 59 p.

93. Norman Matloff “Parallel Computing for Data Science”, CRC Press, USA. – 2015. – 328 p.

94. Poli R., Langdon W.B., McPhee N. F.A Field Guide to Genetic Programming, Lightning Source Inc, 2008. – 233 p.

95. Sedgewick R., Flajolet P. “An Introduction to the Analysis of Algorithms”, 2nd Edition, Publ.: Addison-Wesley Professional, 2013. – 604 p.

96. Shi Z., Zhang J., Yue J., Yang X. A Cognitive Model for Multi-Agent Collaboration // International Journal of Intelligence Science, Vol. 4, №. 1. – 2014. – pp. 1 – 6.

97. Shtovba S. D. Ant Colony Algorithms: Theory and Use // Programming, №4. – 2005. – pp. 1 – 16.

98. Subrahmanian V.S., Bonatti P., Dix J. Heterogeneous Agent Systems. – MIT Press, 2000. – 441 p.

99. Stallings W. High-Speed Networks and Internets – SPb.: Piter. 2003. – 783 p.

100. Stormy Attaway “MATLAB. A practical introduction to programming and problem solving” 3-d edition, - Publ.: Elsevier, USA, 2013. – 538 p.

101. Stützle T., López-Ibáñez M., Pellegrini P., Maur M., Oca M., Birattari M., Michael Maur, Dorigo M. “Parameter Adaptation in Ant Colony Optimization” // Technical Report, IRIDIA, Université Libre de Bruxelles, 2010. – 26 p.

102. Thomas H. Cormen Algorithms Unlocked, MIT Press, 2013. – 216 p.

103. Tianjun Liao, Krzysztof Socha, Marco Montes de Oca, Thomas Stutzle, Marco Dorigo “Ant colony optimization for mixed-variable optimization problems” // IEEE Evolutionary Computation, Transactions on, – Vol.18, Iss.4, 2014. – pp. 503 – 518.

104. Tianjun Liao, Marco A. Montes de Oca, Dogan Aydin, Thomas Stutzle, Marco Dorigo “An Incremental Ant Colony Algorithm with Local Search for Continuous Optimization” // ISSN 1781-3794, – Technical Report TR/IRIDIA/2011-005. – pp. 1 – 17.

105. Tianjun Liao, Thomas Stützle, Marco A Montes de Oca, Marco Dorigo “A unified ant colony optimization algorithm for continuous optimization” // European Journal of Operational Research, Publ.: North-Holland, Vol.234, Iss.3, 2014. – pp. 597 – 609.

106. Tongur V., Ulker E. Migrating Birds Optimization for Traveling Salesman Problem // Матеріали 6-ої Міжнародної науково-технічної конференції ACSN-2013 "Сучасні комп'ютерні системи та мережі: розробка та використання". — Львів. — 2013. — с. 219 – 223.

107. TSP Test data. URL: http://www.tsp.gatech.edu/data/index.html

108. TSPLIB official homepage. Exist from 1995, © Copyright Universität Heidelberg. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

109. Visual studio Official Site URL: https://www.visualstudio.com/

110. Wang R., Jiang H. Two-Dimension Path Planning Method Based on Improved Ant Colony Algorithm // Advances in Pure Mathematics, Vol.5, №9. – 2015. – pp. 571 – 578.

111. Yang X.-S. Nature-Inspired Metaheuristic Algorithms, 2nd Edition. – UK: Luniver Press, 2010. – 160 p.

112. Zar Chi Su Su Hlaing, May Aye Khine “An Ant Colony Optimization Algorithm for Solving Traveling Salesman Problem” // International Conference on Information Communication and Management, – IACSIT Press, Singapore, – IPCSIT, Vol.16. – 2011. – pp. 54 – 59.

ДОДАТКИ

Додаток А

Акт використання в НДР «Інтеграція методів і засобів вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем» (шифр ДБ/КІБЕР, реєстраційний номер № 0115U000446) Національного університету «Львівська політехніка»

ЗАТВЕРДЖУЮ
Проректор з наукової роботи
Національного університету
"Львівська політехніка"
проф. _____ Н.І. Чухрай
"__" _____ 2015 р.

АКТ

Про використання результатів дисертаційної роботи Муляревича Олександра Володимировича «Розв'язання динамічної задачі комівояжера з використанням поведінкової моделі колонії мурах в багатоагентних системах», представленої на здобуття наукового ступеня кандидата технічних наук,
при виконанні НДР «Інтеграція методів і засобів вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем» (шифр ДБ/КІБЕР, реєстраційний номер № 0115U000446)
Національного університету «Львівська політехніка»

Комісія в складі голови – начальника НДЧ к.т.н. Жук Л.В. та членів: завідувача кафедри електронних обчислювальних машин д.т.н., професора Мельника А.О., зав. відділом науково-організаційного супроводу наукових досліджень к.т.н. Лазько Г.В. та заст. нач. планово-фінансового відділу Чулой Т.М. цим актом підтверджують, що результати дисертаційного дослідження Муляревича О.В. щодо використання поведінкової моделі колонії мурах в багатоагентних системах для розв'язання динамічної задачі комівояжера використані при виконанні НДР «Інтеграція методів і засобів вимірювання, автоматизації, опрацювання та захисту інформації в базисі кібер-фізичних систем» (шифр ДБ/КІБЕР, реєстраційний номер № 0115U000446).

Зокрема, Муляревич О.В. апробував розроблену багатоагентну систему з використанням поведінкової моделі колонії мурах, методи виходу з критичних ситуацій «пастки» та «зациклення», метод збільшення значень міток під час переміщення агентів, метод подолання негативних наслідків надмірного збільшення значень цифрових міток, використання яких дозволяє розв'язувати динамічну асиметричну задачу комівояжера в умовах частково невідомих вхідних даних.

Голова комісії:

Начальник НДЧ
к.т.н. Л.В. Жук

Члени комісії:

Зав. кафедри ЕОМ, д.т.н., проф. А.О. Мельник

Зав. відділом НОСНД Г.В. Лазько

Заст. нач. ПФВ Т.М. Чулой

Додаток Б

Акт впровадження в компанії “Logivations GmbH” (Гребенцель,Німеччина)

Logivations GmbH
Oppelner Straße 5
82194 Groebenzell / Munich, Germany
+49 89 21909750
info@logivations.com

from 01.12.2015

An Act

about implementation of PhD research results

of PhD student Muliarevych Oleksandr Volodimirovych, who studying at National University “Lviv Polytechnic”, Department of Computer Engineering, on theme: “Solving dynamic travelling salesman problem using ant colony behavior model in multiagent systems”. Proposed methods and new developed solutions, which are based on swarm behavior model, were used for improvement of simulation algorithms in scope of innovative world-known W2MO simulation and optimization solution. Result of implementation is better evaluation of logistics process workflows, especially in dynamic conditions.

Managing Partner
of “Logivations GmbH”,

Dr. Christoph-Ulrich Plapp

Додаток В

Акт впровадження в компанії “Eleks” (Львів, Україна)

eleks®
тел.: +380 (32) 297-12-51, email: office@eleks.com, www.eleks.com
ТзОВ «ЕЛЕКС» ЄДРПОУ 13806807, вул. Наукова 7, корп. Г, м.Львів 79060, Україна

11.12.2015 р.

АКТ

впровадженя результатів дисертаційного дослідження
Муляревича Олександра Володимировича за темою
«Розв’язання динамічної задачі комівояжера з використанням поведінкової моделі колонії мурах в багатоагентних системах»

Результати дисертаційного дослідження з логістичних оптимізаційних рішень, виконаних асистентом кафедри електронних обчислювальних машин НУ “Львівська політехніка” Муляревичем О.В., використано при розробці спеціалізованих обчислювальних засобів. Зокрема було використано програмну реалізацію розробленої багатоагентної системи з застосуванням методів локальної оптимізації та технологій паралельних обчислень для розв'язання динамічної задачі комівояжера.

Впровадження результатів дисертаційного дослідження дозволило суттєво збільшити швидкодію обчислювальних засобів при розв’язанні динамічної задачі комівояжера, з різницею від оптимального результату не більше 10% для задач розмірністю до 10 000 вузлів.

Директор з розробки Харитонов С.Ю.

Компанія «Елекс»

Товариство з обмеженою відповідальністю * Україна * Львів * ЕЛЕКС ELEKS № 13806807

Додаток Г

Акт використання в навчальному процесі кафедри електронних обчислювальних машин Національного університету «Львівська політехніка»

Затверджую
Проректор з науково-
педагогічної роботи
Національного університету
«Львівська політехніка»
О.Р. Давидчак
"23" грудня 2015 р.

Акт

про використання результатів дисертаційної роботи
Муляревича Олександра Володимировича
на тему «Розв'язання динамічної задачі комівояжера з використанням поведінкової моделі колонії мурах в багатоагентних системах»

Комісія у складі: голови – завідувача кафедри електронних обчислювальних машин, д.т.н., професора Мельника А.О. та членів: професора кафедри електронних обчислювальних машин, д.т.н. Глухова В.С., доцента кафедри електронних обчислювальних машин, к.т.н. Березко Л.О. цим Актом засвідчує, що на кафедрі електронних обчислювальних машин Національного університету «Львівська політехніка» було впроваджено у навчальний процес результати дисертаційних досліджень Муляревича О.В., зокрема:

- у курсі «Теорія інтелектуальних систем» для освітньо-кваліфікаційного рівня «Магістр» при підготовці методичних вказівок до лабораторної роботи «Поведінкова модель на базі алгоритму мурашиної колонії» було використано дослідження та опис поведінкової моделі на базі алгоритму мурашиної колонії, опис засобів проектування багатоагентних систем, програмну реалізацію розробленої багатоагентної системи для розв'язання динамічної задачі комівояжера;
- у курсі «Організація обчислювальних процесів у паралельних системах» для освітньо-кваліфікаційного рівня «Магістр» при підготовці методичних вказівок до лабораторної роботи «Організація паралельних обчислювальних процесів засобами OpenMP» було використано опис засобів організації обчислень в паралельних системах, програмну реалізацію паралельного запуску агентів в розробленій багатоагентній системі;
- при підготовці бакалаврських та магістерських кваліфікаційних робіт.

Завідувач кафедри ЕОМ,
д.т.н., професор — А.О. Мельник

професор кафедри ЕОМ,
д.т.н., професор — В.С. Глухов

доцент кафедри ЕОМ,
к.т.н., доцент — Л.О. Березко

Додаток Д

Характеристика ЗК. Практичне застосування ЗК

Задача комівояжера відноситься до задач комбінаторики. Комбінаторика – це розділ математики, присвячений рішенню завдань вибірки і розташування елементів деякої, зазвичай кінцевої множини відповідно до встановлених правил [55].

Кожне правило визначає комбінаторну конфігурацію – спосіб побудови певної конструкції з великої кількості початкових елементів. Метою комбінаторного аналізу є вивчення комбінаторних конфігурацій. Це вивчення включає в себе питання існування комбінаторних конфігурацій, алгоритми їх побудови, оптимізацію таких алгоритмів, а також рішення завдань перераховування, зокрема визначення числа конфігурацій цього класу. Прості приклади комбінаторних конфігурацій – перестановки, об’єднання і розміщення.

Класичні комбінаторні завдання - це завдання вибору і розміщення елементів великої кінцевої кількості. Початковою умовою такого завдання є деяке формулювання, що відповідає за типом до головоломок.

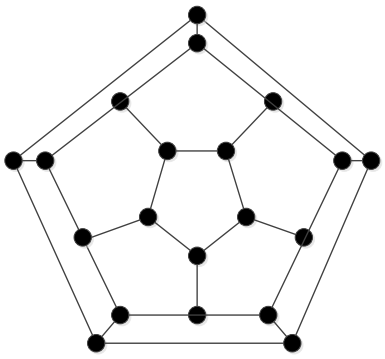


Рис.Д.1. Граф для гри «Кругосвітня подорож»

У 1859 році У. Гамільтон придумав гру «Кругосвітня подорож», що полягає у пошуку такого шляху, який проходить через усі вершини (міста, пункти призначення) графа, який зображено на рис.Д.1., проходження кожної вершини відбувається одноразово, після чого шлях завершується на початкову вершину. Шляхи, що мають таку властивість, називаються гамільтоновими циклами [58]. Завдання знаходження гамільтонових циклів в графі отримало

різні варіації. Одна з таких варіацій – **задача комівояжера.**

У математичному формулюванні матриця вартостей визначається як довільна. Це зроблено через те, що існує багато прикладних завдань, які підпадають під опис основної моделі, проте не задовольняють описані умови (рівняння 1.1 – 1.3). Досить часто порушується друга умова (наприклад, якщо Cij - не відстань, а плата за проїзд: часто туди квиток коштує одну ціну, а назад - іншу), що ще більше ускладнює завдання. Тому розрізняють два варіанти ЗК: симетрична задача, коли виконується умова з рівняння 1.2, та асиметрична – коли не виконується. Умови описані в рівняннях 1.1 та 1.3 вважаються виконаними за замовчуванням [86].

Наступне зауваження стосується кількості усіх можливих маршрутів. У асиметричній ЗК усі маршрути R = (P(0), P(1), … , P(N), P(0)) і R' = (P(0), P(N), … , P(1), P(0)) мають різну довжину і повинні враховуватися обидва. У цьому випадку кількість можливих маршрутів складає: (N - 1)!. Зафіксуємо на першому і останньому місці в циклічній перестановці початковий вузел P(0), а N–1 вузлів, що залишилися, переставимо усіма (N - 1)! можливими способами. В результаті отримаємо усі асиметричні маршрути. Симетричних маршрутів є в два рази менше, оскільки при асиметричній ЗК – маршрут зараховується двічі: як R і як R'.

Припустимо що матриця вартостей C складається лише з одиниць і нулів. Тоді C можна інтерпретувати, як граф, де ребро (i, j) проведене, при Cij=0 та не проведено, при Cij=1. Тоді, якщо існує маршрут довжиною 0, то він пройде по циклу, який включає усі вершини по одному разу. Такий цикл називається гамільтоновим циклом. Незамкнутий цикл Гамільтона називається гамільтоновим ланцюгом (гамільтоновим шляхом).

В теорії графів симетрична ЗК формулюється наступним чином: Дана повна мережа з N вершинами, довжина ребра (i, j) = Cij. Знайти гамільтоновий цикл мінімальної довжини. У асиметричній ЗК замість «цикл» потрібно говорити «контур», а замість «ребра» - «дуги» або «стрілки».

Деякі прикладні завдання формулюються як ЗК, але в них треба

мінімізувати довжину не гамільтонового циклу, а гамільтонового ланцюга. Такі завдання називаються незамкнутими, тобто умова повернення до початкової вершини є необов'язковою. Деякі моделі наближені до реальних зводяться до завдання про сукупність комівояжерів, що відносяться вже до задач з використанням колективу агентів. Надалі методи розв'язання ЗК, що є централізованими та вирішуються за допомогою одного комівояжера, будемо називати класичними методами розв'язку ЗК. Більша частина досліджень сконцентрована на отриманні розв'язків, близьких до оптимальних [92,111].

Задача комбінаторної оптимізації, що виникає при спробі вдосконалити існуючі рішення класичної ЗК, відноситься до числа NP-складних. **Клас складності NP** (Complexity class NP) — клас складності, до якого належать задачі, що можна розв'язати недетермінованими алгоритмами за поліноміальний час; тобто, недетермінованими алгоритмами, в яких завжди існує шлях успішного обчислення за поліноміальний час відносно довжини вхідного рядка; очевидно, що $\mathcal{P} \subseteq \mathcal{NP}$. Власне розв'язок ЗК – послідовність пунктів, що утворюють гамільтоновий цикл, як сертифікат, буде визначати, що вхідний рядок (пункти) належать до класу графів Гамільтона.

Задача комбінаторної оптимізації через можливі похибки в заданих початкових даних та значну кількість локальних мінімумів цільової функції, робить непридатним використання точних алгоритмів рішення через значні обчислювальні затрати. Ці та інші аспекти, а також прогрес в розробці високопродуктивних засобів обчислювальної техніки обумовили інтенсивний розвиток в останні роки класу наближених методів, які отримали назву евристичних.

Практичне застосування задачі комівояжера. Одержані розв'язки ЗК широко використовуються для оптимізації систем керування складними технологічними процесами, перш за все пов'язаними з транспортом та маршрутизацією у комп'ютерних та інформаційних мережах, телефонних комунікаціях, а також й інших технологічних процесів, які зводяться до ЗК – виробництво фарб, діропробивний прес, розведення електричних схем, завдання

рентгенівської кристалографії та інші [35,39,41].

В кожній розглянутій ЗК в наявності може бути як один агент, так і декілька. Наприклад в діропробивному пресі може використовуватись одразу два преси, або в транспортній системі декілька автомобілів розвозять кожен різний товар усім клієнтам по мінімальному маршруту.

Задача про виробництво фарб: Є виробнича лінія для виробництва n фарб різного кольору; позначимо ці фарби номерами 1,2, .. n. Усю виробничу лінію вважатимемо одним процесом. Вважатимемо також, що одноразово процес проводить тільки одну фарбу, тому фарби треба проводити в деякому порядку. Оскільки виробництво циклічне, то фарби потрібно проводити в циклічному порядку π=(j1,j2,..,jn,j1). Після закінчення виробництва фарби i та перед початком виробництва фарби j потрібно відмити устаткування від фарби i. Для цього потрібно час C[i, j]. Очевидно, що C[i, j] залежить як від фарби i, так і від фарби j, також в даному випадку C[i,j]≠C[j,i]. При певному обраному порядку доведеться на цикл виробництва фарб згаяти час. Таким чином, ЗК і завдання про мінімізацію часу переналаштування – одне й те саме завдання, просто описані різними варіантами.

Задача про діропробивний прес: Діропробивний прес проводить велике число однакових панелей - металевих листів, в яких послідовно по одному пробиваються отвори різної форми і величини. Схематично прес можна представити у вигляді столу, що рухається незалежно по координатах x, y, і диска, що обертається над столом, по периметру якого розташовані діропробивні інструменти різної форми і величини. Кожен інструмент є присутнім в одному екземплярі. Диск може обертатися однаково в двох напрямах (координата обертання z). Є власне прес, який натискає на підвішений під нього інструмент тоді, коли під інструмент підведена потрібна точка листа.

Операція пробивки j-того отвору характеризується четвіркою чисел (x_j, y_j, z_j, t_j), де x_j, y_j - координати потрібного положення столу, z_j - координата потрібного положення диска й t_j - час пробивки j-того отвору.

Виробництво панелей носить циклічний характер: на початку й кінці

обробки кожного листа стіл повинен знаходитися в положеннях (x_0, y_0) диск в положенні z_0 причому в цьому положенні отвір не пробивається. Цей початковий стан системи можна вважати пробивкою фіктивного нульового отвору. З параметрами (x_0, y_0, z_0, 0).

Щоб пробити j-ий отвір безпосередньо після i-того необхідно виконати наступні дії:

1. Перемістити стіл по осі x з положення x_i в положення x_j, витрачаючи при цьому час $t(|x_i - x_j|)=t_{i,j}(x)$

2. Виконати те ж саме по осі y, витративши час $t_{i,j}(y)$

3. Повернути голівку по найкоротшій з двох дуг з положення z_i в положення z_j, витративши час $t_{i,j}(z)$.

4. Пробити j-ий отвір, витративши час t_j.

Конкретний вигляд функцій t(x), t(y), t(z) залежить від механічних властивостей пресу та є досить громіздким. Явно виписувати ці функції немає необхідності. Дії 1-3 (переналагодження з i-того отвору до j-того) відбуваються одночасно, і пробивання відбувається негайно після завершення найтривалішої з цих дій. Тому$T_3[i, j] = \max(t(x), t(y), t(z))$. Тепер, як і у попередньому випадку, завдання складання оптимальної програми для діропробивного пресу зводиться до сиемтричної ЗК.

Додаток Е

Аналіз класичних методів розв’язання ЗК

Метод повного перебору. Метод повного перебору полягає у тому, що виконується перевірка усіх можливих комбінацій пунктів призначення в пошуках найоптимальнішого. Як відомо з математики, кількість таких перестановок дорівнює n!, де n – кількість пунктів. Але оскільки в ЗК початковий пункт приймається зазвичай той самий (наприклад перший), то залишається перебрати ті що залишились перестановки, тобто кількість перестановок буде дорівнювати (n - 1)!. Цей алгоритм відноситься до точних алгоритмів, тобто знаходить найоптимальніший шлях, якщо він є, проте час необхідний на обчислення цього маршруту може приймати астрономічні значення. Аналіз показав, що при значеннях n > 12, сучасний комп’ютер не зміг би вирішити ЗК навіть за час існування людства – навіть якщо припустити, що при n = 13 і комп’ютер формує 1-у послідовність за 1-у секунду, знадобилось би майже 198 років (13!) на розв’язання ЗК.Тому даний метод практично майже не використовується.

Жадібний алгоритм. Жадібний алгоритм - алгоритм знаходження найкоротшої відстані шляхом вибору найкоротшого, ще не вибраного ребра, за умови, що воно не утворює циклу із вже вибраними ребрами. «Жадібним» цей алгоритм названий тому, що на останніх кроках алгоритму доводиться жорстоко розплачуватися за «жадібність» – неоптимальні вибори.

Принцип поведінки жадібного алгоритму при розв’язанні ЗК полягає у стратегії «йти до найближчого (у який ще не входив) міста». Розглянемо для прикладу мережу (рис.Е.1), що представляє собою звужений ромб. Нехай комівояжер стартує з міста 1. Алгоритм за принципом «йти до найближчого міста» виведе його в місто 2, потім 3, потім 4; на останньому кроці доведеться платити за жадібність, повертаючись по довгій діагоналі ромба. В результаті вийде не найкоротший, а найдовший маршрут. На перевагу алгоритму слід відмітити, що стратегія «йти до найближчого» при старті з одного міста не поступиться стратегією «йти до найдальшого». Та безперечно кращий за

алгоритм перебору.

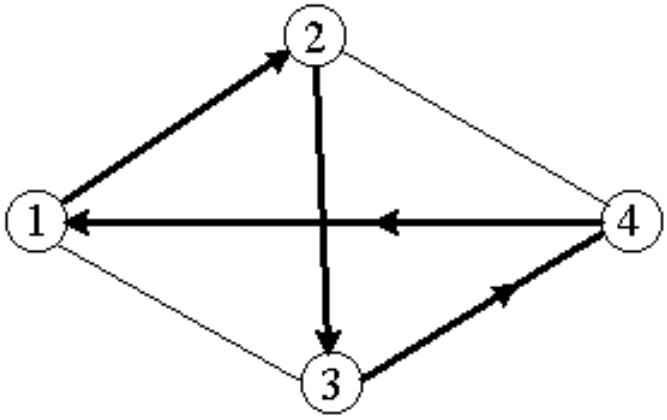


Рис.Е.1. Демонстрація розв’язання ЗК жадібним алгоритмом

Статистика помилки алгоритмів: Жадібний алгоритм не дає оптимальний результат для даної ЗК. Виникає питання стосовно знаходження імовірності помилки, тобто характеристики знайдених маршрутів відносно мінімального, тобто найкращого варіанту. Доведемо, що цього визначити не можливо, причому не лише для жадібного алгоритму, а й для алгоритмів набагато складніших. Для початку треба визначити, як оцінювати похибку евристичних алгоритмів для визначеної в завданнях мінімізації. Нехай fB - справжній мінімум, а fA – квазімінімум, який отриманий за певним алгоритмом. Ясно, що $fA / fB \geq 1$, але це - тривіальне твердження, за визначенням похибки. Щоб оцінити її, треба обмежити відношення верхньою оцінкою:

$$fA / fB \geq 1 + n\varepsilon, \qquad (E.1)$$

де, як завжди у вищій математиці, $\varepsilon \geq 0$, та може приймати будь-яке як завгодно мале значення, так і може бути дуже великим. Величина ε є мірою похибки. Якщо алгоритм мінімізації задовольняє нерівність (Е.1), це означає, що він має похибку ε.

Припустимо тепер, що є алгоритм А розв’язання ЗК, похибку якого треба оцінити. Візьмемо довільний граф G (V, E) і по ньому складемо вхідну матрицю С для ЗК:

$$C[i, j] = \begin{cases} 1, \text{ якщо ребро } (i, j) \text{ належить E.} \\ 1+ n\varepsilon, \text{ в інакшому випадку.} \end{cases} \qquad (E.2)$$

Якщо в графі G є гамільтоновий цикл, то мінімальний маршрут проходить по цьому циклу, тобто fB = n. Якщо алгоритм А теж завжди знаходитиме цей шлях, то за результатами алгоритму можна судити, чи є гамільтоновий цикл в довільному графі. Проте, безвідмовного алгоритму, який міг би відповісти, чи є гамільтоновий цикл в довільному графі, досі ще не винайшли. Таким чином, алгоритм А повинен іноді помилятися і включати в маршрут хоча б одне ребро довжини 1+ nε. Але тоді fA ≥ (n - 1) + (1 + nε) так що fA / fB = 1 + nε тобто перевершує похибку ε на задану нерівністю (Е.1). Величина ε може бути довільно великого значення.

Таким чином доведена наступна теорема: Алгоритм А визначає чи існує в довільному графі гамільтоновий цикл, інакше похибка алгоритму А при рішенні ЗК може бути довільно велика. Це визначення вперше було опубліковане Сані та Гонзалесом в 1980 р [39, 95]. Теорема Сані-Гонзалеса заснована на тому, що немає ніяких обмежень на довжину ребер. Теорема не підтверджується, якщо відстані підкоряються нерівності трикутника (1.4).

Цикл Гамільтона є фактично частковим випадком ще однієї класичної задачі, в якій метою ставилось намалювати однією лінією відкритий конверт. Це завдання вирішив згодом Ейлер, який довів що замкнута лінія, яка покриває усі ребра графа, тепер називається ейлеревим циклом, існує тільки тоді, коли:

1) граф зв’язаний;

2) усі його вершини мають парні міри;

Для розв’язання ЗК методом дерева – або дерев’яним алгоритмом, будування ейлеревого циклу є досить важливою умовою.

Дерев'яний алгоритм. Метод полягає в побудові «найкоротшого» кістякового дерева. Дерево формується так, що його корінь буде початковий пункт маршруту, далі гілки формуються за проходженням до найближчих пунктів за відстанню, з поверненням до кореня, і так поки до дерева не будуть включені усі вершини графу.

На рис.Е.2. зображено початкова вхідна мережа для ЗК на 6 пунктів. В таблиці Е.1 представлено початкові дані даної ЗК – матриця вартостей С, де

кожне значення – довжина між пунктами. В межах даного графу формуємо кістякове дерево і подвоїмо усі його ребра. Отримаємо в результаті граф G з вершинами, що мають тільки парні міри, як зображено на рис.Е.3. пунктиром. Побудуємо ейлерів цикл, починаючи з вершини 1, цикл задається переліком вершин: 1-2-1-3-4-3-5-6-5-3-1.

Проглянемо перелік вершин, починаючи з 1-ї та закреслюючі кожну вершину, яка повторює вже існуючу в послідовності. Залишиться тур, який і є результатом алгоритму: 1-2-3-4-5-6-1. Згідно теореми про те, що похибка дерев'яного алгоритму дорівнює 1-ці, з нерівності (Е.1) випливає, що дерев'яний алгоритм помиляється менш, ніж в два рази.

Таблиця Е.1.

Початкові дані для ЗК на 6 пунктів.

-	**1**	**2**	**3**	**4**	**5**	**6**
1	-	6	4	8	7	14
2	6	-	7	11	7	10
3	4	7	-	4	3	10
4	8	11	4	-	5	11
5	7	7	3	5	-	7
6	14	10	10	11	7	-

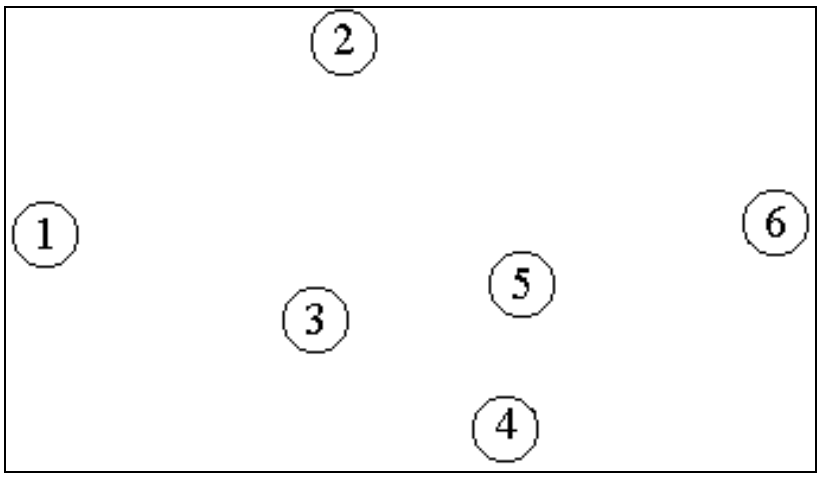


Рис.Е.2. Початкова мережа для ЗК

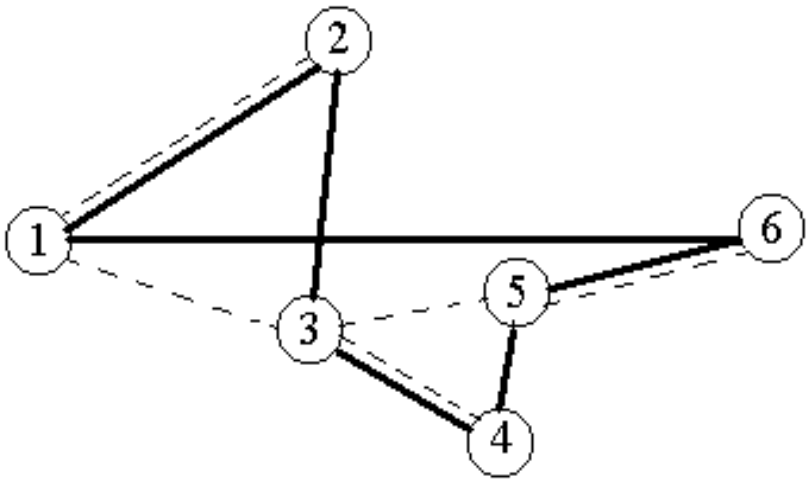


Рис.Е.3. Розв’язання ЗК дерев’яним алгоритмом

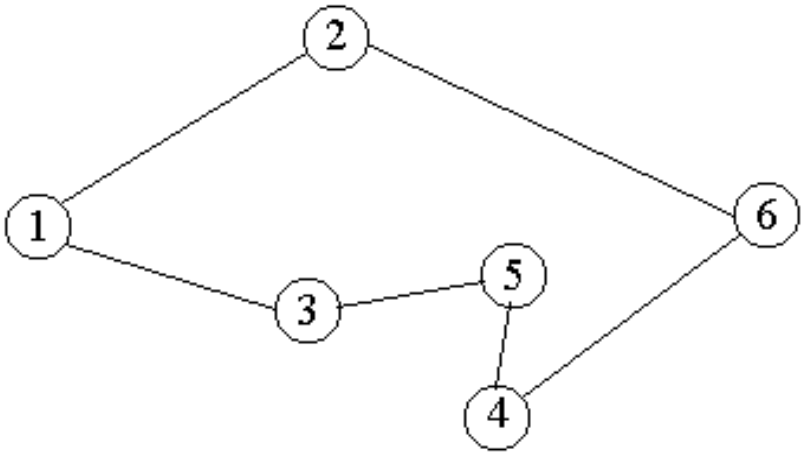


Рис.Е.4. Розв’язання ЗК жадібним алгоритмом

В даному практичному прикладі довжина знайденого шляху дерев’яним алгоритмом складатиме – 43. Для порівняння розв’яжемо цю ж мережу жадібним алгоритмом як показано на Рис.Е.4. Довжина шляху отриманого жадібним методом – дає тур 1(4) - 3(3) - 5(5) - 4(11) - 6(10) - 2(6) - 1, який має довжину – 39. Для порівняння методом перебору найоптимальніший шлях має довжину – 36.

Конструктивні алгоритми до яких відносяться жадібний алгоритм, алгоритм найближчого сусіда, алгоритм вставки, забезпечують найгіршу якість отриманих розв’язків задачі комівояжера при обчислювальній складності в середньому $O(n^2)$ [95]. Існують методи пришвидшення роботи описаних вище

підходів. Зокрема завдяки методу, запропонованому Бентлі – k-d деревам, або як вище розглянутий варіант дерев'яного алгоритму, що ґрунтуються на побудові ієрархічної декомпозиції вхідної області точок на частини, можна пришвидшити роботу алгоритмів до часу, еквівалентному O(n*log(n)).

Алгоритм Дейкстри. Одним з варіантів рішення ЗК є варіант знаходження найкоротшого ланцюга, що містить усі міста. Потім отриманий ланцюг доповнюється початковим містом - виходить шуканий тур.

Можна запропонувати багато процедур рішення цієї задачі, наприклад, фізичне моделювання. На плоскій дошці малюється карта місцевості, в міста, що розташовані на розгалуженні доріг, забиваються цвяхи, на кожен цвях одягається кільце, дороги прокладаються мотузками, які прив'язують до відповідних кілець. Щоб знайти найкоротшу відстань між i та k пунктами, треба взяти кінці мотузки на пункті i в одну руку та на пункті k в іншу та розтягнути. Ті мотузки, які натягатимуться і не дадуть розводити руки ширше і утворюють найкоротший шлях між i та k пунктами. Проте математична процедура, яка відповідає цій фізичній, виглядає досить складно. Але є алгоритми, що використовують цю ідею хоча б частково, один з них - алгоритм Дейкстри, запропонований Дейкстром ще у 1959 р. Цей алгоритм вирішує загальну задачу: в орієнтованій, неорієнтованій або змішаній (тобто де частина ребер є дугами) мережі знайти найкоротший шлях між двома заданими вершинами.

Алгоритм використовує три масиви з N чисел кожен. Перший масив A містить мітки з двома значеннями: 0 (вершина ще не розглянута) і 1 (вершина вже розглянута). Другий масив B містить відстані - поточні найкоротші відстані від якоїсь обраної початкової вершини v_i до відповідної вершини; третій масив C містить номери вершин - k-й елемент C_k є номер передостанньої вершини на поточному найкоротшому шляху з v_i в v_k. Матриця відстаней D_{ik} задає довжини дуг d_{ik}; якщо такої дуги немає, то d_{ik} привласнюється велике число X.

Опис алгоритма Дейкстри:

1) Ініціалізація: в циклі від одного до N заповнити нулями масив A; заповнити числом i масив C: перенести i-тий рядок матриці D в масив B;

A[i]:=1; C[i]:=0; {i - номер стартової вершини}

2) Загальний крок: знайти мінімум серед невідмічених (тобто тих k, для яких A[k] = 0);

3) Видача відповіді: шлях $v_i.v_k$ видається в зворотному порядку наступною процедурою: z = C[k]; Видати z; z = c[z]; Якщо z = 0, то кінець, інакше перейти до видачі z.

Для виконання алгоритму потрібно N разів проглянути масив B з N елементів, тобто алгоритм Дейкстри має квадратичну складність – $O(n^2)$. На рис.Е.5. відображено роботу алгоритму Дейкстри чисельним прикладом (для більшої складності, вважаємо, що деякі міста (вершини) i, j не сполучені між собою, тобто D[i, j] = ∞). Початкова вершина маршруту буде пункт 3. В таблиці Е.2 відображено стан 3-х масивів, згідно виконання першого кроку алгоритму, фактично першого проходження.

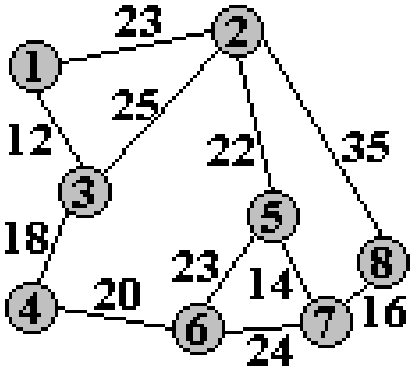


Рис.Е.5. Мережа ЗК для алгоритму Дейкстри

Таблиця Е.2.

Масиви A,B,C, після виконання першого загального кроку алгоритму Дейкстри

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
A	0	0	1	0	0	0	0	0
B	12	25	0	18	∞	∞	∞	∞
C	3	3	0	3	3	3	3	3

Таким чином, для розв’язання ЗК потрібно N раз застосувати алгоритм

Дейкстри. З початку обирається довільна пара вершин між якими знаходиться найкоротший маршрут. Далі аналогічно знаходимо найкоротший шлях між парами вершин алгоритмом Дейкстри, до тих пір, поки усі вершини не будуть задіяні. З'єднаємо останню вершину з першою і отримаємо кінцевий маршрут. Найчастіше це останнє ребро виявляється дуже великим, і результат виходить з похибкою, проте алгоритм Дейкстри можна віднести до евристичних алгоритмів, які дають непогані результати при використанні в сукупності з методами локальної оптимізації.

Одним з важливих недоліків такого алгоритму є необхідність знати не матрицю відстаней, а координати кожного міста на площині. Якщо відома матриця відстаней між містами, але невідомі їх координати, то для їх знаходження треба буде вирішити N систем квадратних рівнянь з N невідомими для кожної координати. Вже для 6 міст це зробити досить складно. Якщо ж, навпаки, є координати усіх міст, але немає матриці відстаней між ними, то створити цю матрицю простіше. Її можна без складностей обрахувати використовуючи можливості комп’ютера, тоді як промоделювати рішення системи квадратних рівнянь на комп'ютері досить складно і займає багато часу.

Метод гілок і меж. Ідея методу розбиття завдання на певні етапи тобто на декілька менш об’ємних завдань виникала багато разів. Але найбільш вдалий алгоритм рішення ЗК методом гілок і меж розробив Літтл. Алгоритм Літтла досить швидко набув популярності та має багато версій модифікацій та покращень. Метод гілок і меж був успішно застосований до багатьох завдань. Загальна ідея тривіальна: треба розділити величезне число варіантів перебору на класи і отримати оцінки (знизу – в завданні мінімізації, згори – в завданні максимізації) для цих класів, щоб мати можливість відкидати варіанти не по одному, а цілими класами. Складність полягає в тому, щоб знайти такий розподіл на класи (гілки) і такі оцінки (межі), щоб процедура була ефективною.

Алгоритм діє згідно проведення операцій над таблицею даних, в якій записані – вартість, час чи довжина маршруту. Постійно проводиться оцінка нижньої границі. Так Літтл довів, що віднімаючи будь-яку константу з усіх

елементів будь-якого рядка або стовпця вхідної матриці, ми залишаємо мінімальний тур мінімальним. На вже обрані ділянки – класи, що суттєво наближені до нижньої межі накладається заборона використання. Та відбувається подальший аналіз та операції над таблицею, для виявлення нових класів та визначення їх верхньої та нижньої межі.

Так якщо вважати таблицю Е.1 як початкові дані для алгоритму Літтла, результат розбиття на класи та визначення найоптимальнішого туру набуде вигляду зображеного на рис.Е.6. Класи, що мають високу нижню оцінку відкидаються з усіма дочірніми гілками (на рис.Е.6. позначені лінією зверху над класом). Зверху над класом позначена нижня границя оцінки шляху.

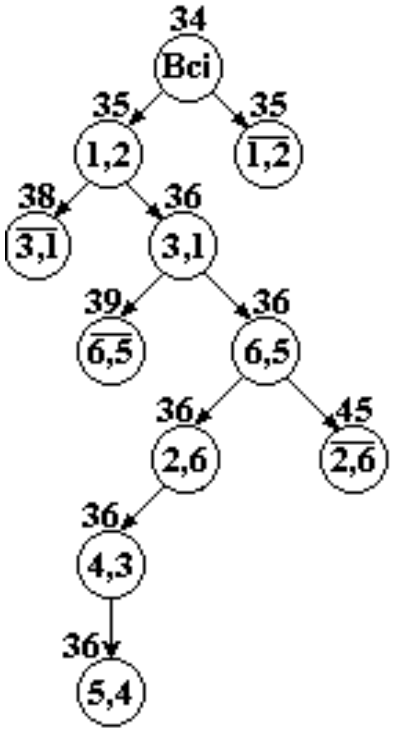


Рис.Е.6. Розв’язок ЗК алгоритмом гілок і меж

Задовільних теоретичних оцінок швидкодії алгоритму Літтла і споріднених алгоритмів немає, але практика показує, що на сучасних ЕОМ вони дозволяють вирішити ЗК з N до 100 за прийнятний на практиці час. Це величезний прогрес в порівнянні з повним перебором. Даний метод також відноситься до евристичних методів.

Для розв’язання ЗК було розроблено багато алгоритмів, проте тільки

алгоритм глобального перебору дає однозначну відповідь про існування циклу Гамільтона – тобто найкоротшого маршруту. Однак даний алгоритм є й найменш ефективним.

Додаток Є

Опис обраних засобів розробки та технологій

Для «графічної частини» був використаний пакет програми **Mathworks Matlab R2010a**, вікно середовища даного пакету програм зображено на рис.Є.1. MATLAB — це мова технічних розрахунків високого рівня, як інженерних, так і математичних; інтерактивне середовище розробки алгоритмів і сучасний засіб аналізу даних [100]. Створена компанією The MathWorks, це досить простий засіб для роботи з математичними матрицями, малювання функцій, роботи з алгоритмами, створення робочих оболонок (user interfaces) з програмами в інших мовах програмування. Хоча цей продукт спеціалізується на чисельному обчисленні, спеціальні інструментальні засоби працюють з програмним забезпеченням Maple, що робить його повноцінною системою для роботи з алгеброю. MATLAB надає користувачеві велику кількість функцій для аналізу даних, які покривають майже усі області математики.

Обчислювальна програма в більшості випадків призначена для дослідження деяких явищ, поведінки пристроїв що розроблюються. Інформація що отримується в результаті виконання таких програм, як правило, має вигляд ряду чисел, кожне з яких відповідає значенню конкретного параметра (аргументу). Таку інформацію зручно представляти в графічному вигляді. В даному випадку це два цілочисельні масиви згенерованих координат по осям x та y. Для створення графічного інженерного документа в MATLAB необхідно використовувати функцію subplot: *subplot (m, n, p)*, де m – вказує на скільки частин розбивається графічне вікно по вертикалі, n – по горизонталі, p – номер підвікна в якому буде будуватись графік.

Сама ж функція виведення графіку досить елементарна, має наступний вигляд: *plot (x(1),y(1),'k*')*, де перші два значення вказують координати точок, а третій вказує тип маркеру точок, колір точок та ліній, та тип ліній, що з’єднують точки послідовно (у випадку масиву точок).

Microsoft Visual Studio — лінійка продуктів фірми Майкрософт, що включають інтегроване середовище розробки програмного забезпечення і ряд

інших інструментальних засобів. Для розробки системи моделювання алгоритму та допоміжної програми використовувалась Visual Studio 2005, вікно середовища даної програми зображено на рис.Є.2, використовувані мови програмування C та C++ [20].

Visual Studio включає один або декілька з наступних компонентів:

* Visual Basic .NET, а до його появи — Visual Basic
* Visual C++
* Visual C#
* Visual J#

А також:

* Microsoft SQL Server, або
* MSDE Visual Source Safe — файл-серверна система управління версіями.

Visual Studio 2005 (кодове ім'я Whidbey; внутрішня версія 8.0) — випущена в кінці жовтня 2005 (включає .NET Framework 2.0). На початку листопада 2005 також вийшла серія продуктів в редакції Express: Visual C++ 2005 Express, Visual Basic 2005 Express, Visual C# 2005 Express і інше. 19 квітня 2006 редакція Express стала безоплатною.

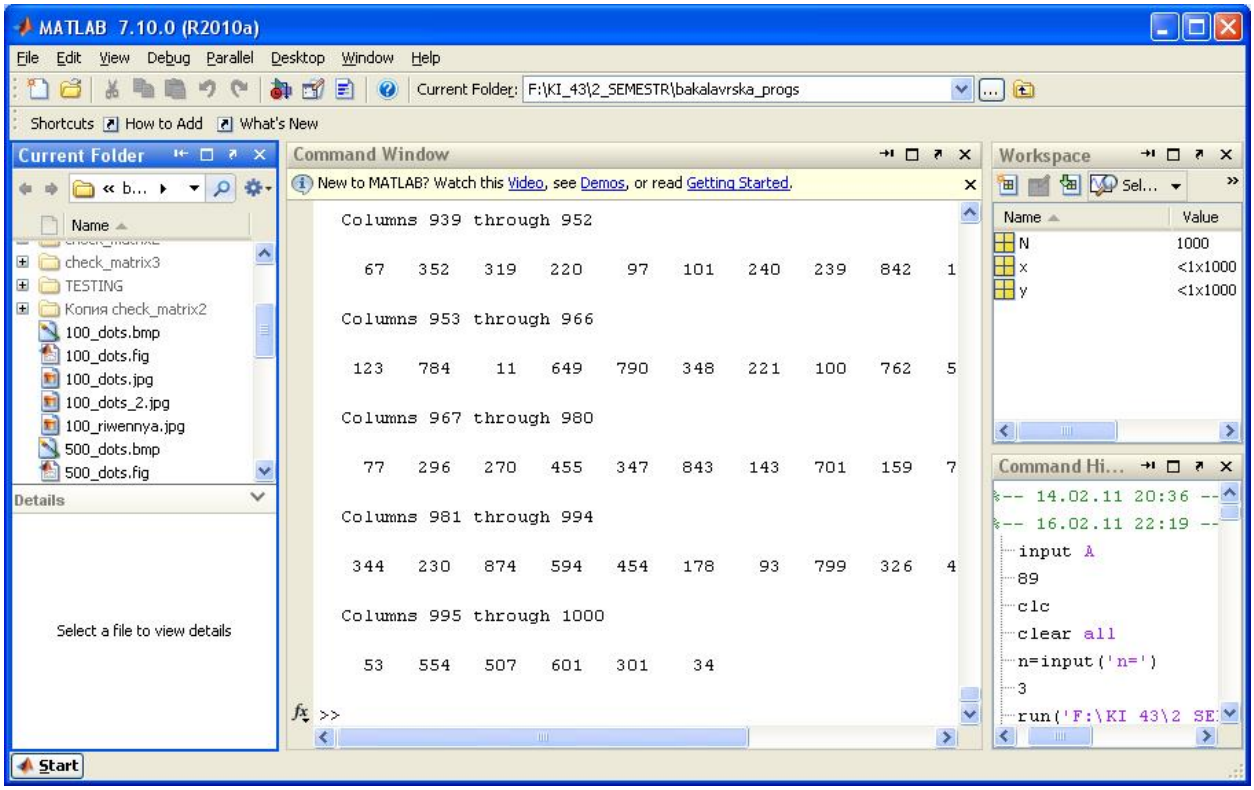


Рис.Є.1. Середовище MATLAB R2010a

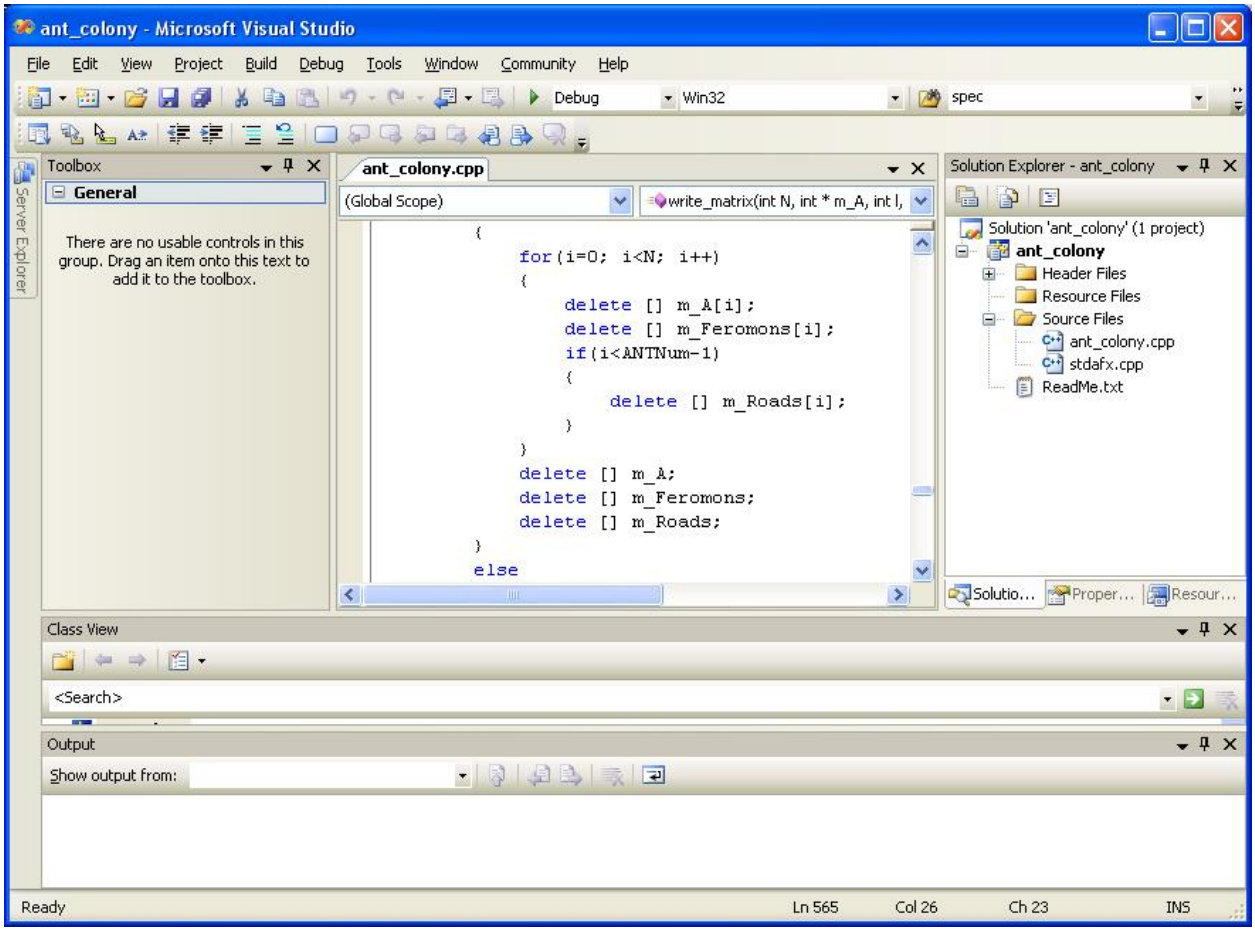


Рис.Є.2. Середовище Microsoft Visual Studio 2005

Мова програмування C++ — універсальна мова програмування високого рівня з підтримкою декількох парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. Стандартна бібліотека C++ включає стандартну бібліотеку C [20] з невеликими змінами. Інша велика частина бібліотеки C++ заснована на Стандартній Бібліотеці Шаблонів (STL). Вона надає такі важливі інструменти, як контейнери (наприклад, вектори і списки) і ітератори (узагальнені вказівники), що надають доступ до цих контейнерів як до масивів. Крім того, STL дозволяє схожим чином працювати і з іншими типами контейнерів, наприклад, асоціативними списками, стеками, чергами.

Мова C++ багато в чому є надмножиною C. Нові можливості C++ включають оголошення у вигляді виразів, перетворення типів у вигляді функцій, оператори new і delete, тип bool, посилання, розширене поняття константності та

змінності, функції, що підставляються, аргументи за умовчанням, перевизначення, простори імен, класи (включаючи і всі пов'язані з класами можливості, такі як успадкування, функції-члени (методи), віртуальні функції, абстрактні класи і конструктори), перевизначення операторів, шаблони, оператор ::, обробку винятків, динамічну ідентифікацію і багато що інше. C++ є також мовою строгого типування і накладає більше вимагань щодо дотримання типів, порівняно з C.

Технології паралельного програмування. З метою змешення часу обчислення було вирішено застосувати основи паралельного програмування, де кожен агент функціонує як окремий потік (thread). Було використано стандартні бібліотеки **OpenMP** та **_pthread** стандарту POSIX [93]**,** які включає в себе API для створення так званих тасків – потоків, які в подальшому запускаються в виконавчому класі – Executor. Для застосування OpenMP достатньо оголосити директиви omp визначаючи для компілятора ділянки коду, які буде ініціалізовано в окремих потоках, та виконано паралельно, наприклад:

```
#include <omp.h>

int main()
{
  int th_id, nthreads;
  #pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
    #pragma omp critical

    #pragma omp barrier

    #pragma omp master
    {
      nthreads = omp_get_num_threads();
    }
  }
  return 0;
}
```

Для застосування _pthread достатньо включити наступну директиву: #include <pthread.h>. Клас потоку має наступний вигляд:

```
class Thread
{
private:
    pthread_t thread;

    Thread(const Thread& copy);          // copy constructor denied
    static void *thread_func(void *d)    { ((Thread *)d)->run(); return NULL; }

public:
```

```
    Thread()              {}
    virtual ~Thread()     {}

    virtual void run() = 0;

    int start()           { return pthread_create(&thread, NULL,
                            Thread::thread_func, (void*)this); }
    int wait ()           { return pthread_join  (thread, NULL); }
};

typedef std::auto_ptr<Thread> ThreadPtr;
```

В подальшому функцію run потоку можна перевизначати конкретно при виклику. Кожен мураха-агент запускається як окремий потік, що дозволяє зменшити час на виконання одної ітерації циклу пошуку маршрутів мурахами-агентами.

Додаток Ж

Лістинг програмної реалізації багатоагентної системи для розв'язання динамічної ЗК з використанням додаткового програмного модуля на базі методів локальної оптимізації

Представлено лістинг програмної реалізації розробленої багатоагентної системи з використанням поведінкової моделі колонії мурах, технологій паралельних обчислень та методів локальної оптимізації.

Лістинг програми підсистеми графічного відображення, макрос inputm Mathlab середовища для ЗК на 100 точок:

```
clc   %очистка пам'яті
clear all   %закриття всіх графічних вікон
close all   %очистка екрану
N=100; % розмір матриць
x = [568   319   897   201   587   544   348   127    22   379   165   653   333
757 660   513   159   861   238   832   201   336    78   576   162    40   650
312 594   345   564    19   819   720   671   731   344   555   517   477   247
223 406   204   724   887    26   482    78   721   890    60   845    16   615
705 480   796   809   563   124   196   163    37    96   554   845   319   369
885 851   608   889   690   303   596   219   265   612   475   370   542   675
525 496   525   460    74   647   896   319   874   311   797   409   372   195
113 278   653]
y = [704   624     8   758   830   693    38   340   633   656   201   242   605
429 561   212   159   746   690   841    97   164    89   440   173   806    89
39 501   695   280   161   305   189   459   815   566    91   351    49   451
388 897   730   437   805   123   351   834   825   642   556   308   842   112
657 581   749   358   674   751   290   497   881   494   297   557   324   680
372 443   625   875   294   754   665   858    28   321   596   253   207   640
562 531   594    42   313   406   216   643   770   253   657   123   753   124
529 329   726]
plot (x(1),y(1),'k+'), title('Вхідний граф для ЗК'); %вивід графу
hold on
plot (x(1),y(1),'k*')
plot (x,y,'ro')   %вивід функцій на спільному графіку
axis([0 900 0 900])
xlabel( ' X координати ' )
ylabel( ' Y координати ' )
```

макрос outputm:

```
clc   %очистка пам'яті
clear all   %закриття всіх графічних вікон
close all   %очистка екрану
N=100; % розмір матриць
x = [568   319   897   201   587   544   348   127    22   379   165   653   333
757 660   513   159   861   238   832   201   336    78   576   162    40   650
312 594   345   564    19   819   720   671   731   344   555   517   477   247
223 406   204   724   887    26   482    78   721   890    60   845    16   615
705 480   796   809   563   124   196   163    37    96   554   845   319   369
885 851   608   889   690   303   596   219   265   612   475   370   542   675
525 496   525   460    74   647   896   319   874   311   797   409   372   195
113 278   653];
y = [704   624     8   758   830   693    38   340   633   656   201   242   605
```

```
429 561   212   159   746   690   841    97   164    89   440   173   806    89
39 501   695   280   161   305   189   459   815   566    91   351    49   451
388 897   730   437   805   123   351   834   825   642   556   308   842   112
657 581   749   358   674   751   290   497   881   494   297   557   324   680
372 443   625   875   294   754   665   858    28   321   596   253   207   640
562 531   594    42   313   406   216   643   770   253   657   123   753   124
529 329   726];
t = [1 60 76 72 83 56 100 6 86 84 85 57 80 10 69 30 96 75 19 44 4 61 26 54 64 49
77 43 5 50 36 20 73 46 92 18 58 94 51 67 71 70 53 33 59 14 45 35 89 24 29 15 79
31 66 39 48 16 82 38 55 27 34 12 74 90 3 40 87 7 28 78 21 97 25 17 11 62 8 88 32
47 23 22 95 81 93 68 99 42 41 63 65 98 52 9 13 2 91 37 1];
plot (x(1),y(1),'k+'), title('Вхідний граф для ЗК'); %вивід графу
hold on
plot (x(1),y(1),'k*')
plot (x(t),y(t),'-ro')  %вивід функцій на спільному графіку
axis([0 900 0 900])
xlabel( ' X координати ' )
ylabel( ' Y координати ' )
```

Лістинг програми формалізації даних check_matrix.c:

```
#include "conio.h"
#include <direct.h>
#include <stdlib.h>
#include <math.h>
#include "string.h"
#include "stdio.h"
#include "time.h"
#include "iostream.h"
//using namespace std;

void write_matrix(int N, int **m_A, FILE *pF2);

void correct_matrix(int N, int **m_A);

void read_matrix(int **m_A, FILE *pF1);

char* buf = _getcwd( NULL, 0);

void main()
{
      srand ( time(NULL) );

      int i,j;
      int N;

      char c_sel;
      char* buf1 = _getcwd( NULL, 0);
      char fname[40];

      bool s_err = false;

      FILE *pF2;
      FILE *pF1;

      cout<<"Choose program mode:"<<endl;
      cout<<"For randomize matrix N x N input 'R',"<<endl;
      cout<<"For  check  of  input  matrix  of  size  N  x  N  of  distance  input
'D'"<<endl;
cout<<"For creation of matrix N x N from matrix of coordinates input 'C'"<<endl;
      cin >> c_sel;

      if((c_sel != 'R')&&(c_sel != 'D')&&(c_sel != 'C'))
```

```
	{
		s_err = true;
	}

	cout << "Input N - size of matrix" << endl;
	cin >> N;

	if((N < 0)||(N > 100000))
	{
		s_err = true;
	}

	int **m_Cor = new int*[2];
	for(i=0; i<2; i++)
	{
	 m_Cor[i]=new int[N];
	}

	int **m_A = new int*[N];
	for(i=0; i<N; i++)
	{
		 m_A[i]=new int[N];
	}
//	cout<<"Create matrix"<<endl;

	if(s_err == false)
	{
		cout << "Input file name:"<<endl;
		cin >> fname;
		if(c_sel != 'R')
		{
			strcat(buf,"\\");
			strcat(buf,fname);
		//	strcat(buf,".txt");
			pF1 = fopen (buf, "rt");
		//	cout<<"op - "<<buf<<endl<<pF1<<endl;
		}
		strcat(buf1,"\\l_");
		strcat(buf1,fname);
		strcat(buf1,".txt");
		pF2 = fopen (buf1, "wb");
	}
	if((!pF2)||((c_sel!='R')&&(!pF1)))
	{
		s_err = true;
	}
	if(s_err == false)
	{
		if(c_sel == 'R')
		{
			for(i = 0; i < N; i++)
			{
				for(j = 0; j < N; j++)
				{
					m_A[i][j] = rand() % 900;
					if(i == j)
					{
						m_A[i][j] = 0;
					}
				}
			}
			cout<<"randomize succesfully"<<endl;

			//check and correct matrix;
```

```
                  correct_matrix(N,m_A);
                  cout<<"correct succesfully"<<endl;

                  //writing matrix:
                  write_matrix(N,m_A,pF2);
                  cout<<"writed succesfully"<<endl;

            }
            if(c_sel == 'D')
            {
                  read_matrix(m_A,pF1);

                  correct_matrix(N,m_A);

                  write_matrix(N,m_A,pF2);

            }

            if(c_sel == 'C')
            {
                  read_matrix(m_Cor,pF1);

                  for(i = 0; i < N; i++)
                  {
                        for(j = 0; j < N; j++)
                        {
                              if(i == j)
                              {
                                    m_A[i][j] = 0;
                              }
                              else
                              {
                              m_A[i][j] = sqrt((m_Cor[0][i]-m_Cor[0][j])*
(m_Cor[0][i]-m_Cor[0][j]) + (m_Cor[1][i]-m_Cor[1][j])
                                          *(m_Cor[1][i]-m_Cor[1][j]));
                              }
                        }
                  }

                  //writing matrix:
                  write_matrix(N,m_A,pF2);
            }
            fclose(pF2);
            if(c_sel != 'R')
            {
                  fclose(pF1);
            }
      }
      else
      {
            cout<<"Can't open file or bad input!"<<endl;
      }
      cout<<"THE END OF PROGRAM"<<endl;
      getch();
}
void write_matrix(int N, int **m_A, FILE *pF2)
{
      char itoa_buf[6];
      int i,j;
      for(i = 0; i < N; i++)
      {
            for(j = 0; j < N; j++)
            {
                  itoa((m_A[i][j]),itoa_buf,10);
```

```
                fputs(itoa_buf,pF2);
                fputs(" ",pF2);
            }
            fputs(" ; ",pF2);
        }
}
void correct_matrix(int N, int **m_A)
{
        int i,j,k;
        for(i = 0; i < N; i++)
        {
            for(j = 0; j < N; j++)
            {
                for(k = 0; k < N; k++)
                {
                    if((k > j) && (j > i))
                    {
                        if(m_A[i][j]+m_A[j][k] < m_A[i][k])
                        {
                            m_A[i][k] = m_A[i][j]+m_A[j][k];
                        }
                    }
                }
            }
        }
}
void read_matrix(int **m_A, FILE *pF1)
{
        char atoi_buf[5];
        bool b_per = false;
        int i=0;
        int j=0;
        char pl;
        memset(atoi_buf,0,sizeof(atoi_buf));
        int ll = 0;
        while(pl != EOF)
        {
            pl = fgetc(pF1);
            if((pl == ' ')||(pl == ';'))
            {
                ll = 0;
                if(pl == ';')
                {
                    i++; j = 0;
                    b_per = true;
                }
                else if(b_per != true)
                {
                    m_A[i][j] = atoi(atoi_buf);
                    j++;
                    b_per = true;
                }
                memset(atoi_buf,0,sizeof(atoi_buf));
            }
            else if((pl >= '0')&&(pl <= '9'))
            {
                b_per = false;
                atoi_buf[ll] = pl;
                ll++;
            }
        }
}
```

Лістинг програмної реалізації підсистеми обчислення результуючого маршруту ЗК:

Лістинг реалізації алгоритму колонії мурах з врахуванням запропонованої модифікації алгоритму – AntTSPSolver.c:

```
#include<time.h>
#include<windows.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>
#include<limits.h>
#include<assert.h>
#include<string.h>
#include<stdlib.h>

#include"ANT_utils.h"
#include"Utils.h"
#include"Reader.h"
#include"TSP_Data.h"
#include"TimerProcessor.h"
#include"LocalSearchModule.h"

void my_pheromon_intital(void);

void my_route_search(void);

void my_update_pher(void);


FILE* File;

longint terminate_search_cycle ( void )
{
	return (n_tours >= max_tours);
}

void construct_solutions( void )
{
	long temp;
	longint k;		/* counter variable */
	longint step;	/* counter of the number of construction steps */

	TRACE ( printf("construct solutions for all ants\n"); );

/* Mark all cities as unvisited */
for ( k = 0 ; k < n_ants ; k++) {
	ant_empty_memory( &ant[k] );
}

	step = 0;
/* Place the ants on same initial city */
for ( k = 0 ; k < n_ants ; k++ ) {
	place_ant( &ant[k], step);
}


while ( step < n-1 ) {
	step++;
	for ( k = 0 ; k < n_ants ; k++ ) {
	    choose_and_move_to_next( &ant[k], step);
	    pheromone_update( &ant[k], step );
	}
}
```

```
    step = n;
for ( k = 0 ; k < n_ants ; k++ ) {
       ant[k].tour[n] = ant[k].tour[0];
       ant[k].tour_length = compute_tour_length( ant[k].tour );
       pheromone_update( &ant[k], step );
       n_tours += n_ants;
}

void init_try( longint try )
/*
      FUNCTION: initilialize new try
      INPUT:    try number
      OUTPUT:   none
      COMMENTS: none
*/
{

TRACE ( printf("INITIALIZE TRY\n"); );

    start_timers();
    time_used = elapsed_time( VIRTUAL );
    time_passed = time_used;

    if (comp_report) {
        fprintf(comp_report,"seed %ld\n", seed);
        fflush(comp_report);
    }

/* Initialize variables concerning statistics etc. */
    n_tours      = 1;
    iteration    = 1;
    restart_iteration = 1;
    best_so_far_ant->tour_length = INFTY;
    found_best   = 0;
    init_pheromone_trails();

/* Calculate combined information pheromone times heuristic information */
    compute_total_information();

       if (comp_report) fprintf(comp_report,"begin try %li \n",ntry);
       if (stat_report) fprintf(stat_report,"begin try %li \n",ntry);
}


long local_search( void )
{

       longint k;
       longint fetch_num = 0;

       TRACE ( printf("call for local search module \n"); );

       opt = checkLocalSearchMode();
       for ( k = 0 ; k < n_ants ; k++ ) {
              switch (opt) {
       case 1:
          two_opt_fetch[k] = call_two_opt( ant[k].tour );    /* 2-opt local search */
          fetch_num += two_opt_fetch[k];
              break;
       case 2:
          two_hopt_fetch[k] =  call_two_half_opt ( ant[k].tour );  /* 2.5-opt local search */
           fetch_num += two_hopt_fetch[k];
              break;
       case 3:
```

```
            three_opt_fetch[k] = call_three_opt( ant[k].tour );  /* 3-opt local search */
             fetch_num += three_opt_fetch[k];
                break;
          default:
              exit(1);
          }
          ant[k].tour_length = compute_tour_length( ant[k].tour );
          if (termination_condition()) return fetch_num;
      }
      return fetch_num;
}



void update_final_statistics( void ) {

longint iteration_best_ant;
iteration_best_ant = find_best();
if ( ant[iteration_best_ant].tour_length < best_so_far_ant->tour_length ) {

       time_used = elapsed_time( VIRTUAL ); /* best sol found after time_used */
       copy_from_to( &ant[iteration_best_ant], best_so_far_ant );
       copy_from_to( &ant[iteration_best_ant], restart_best_ant );

       found_best = iteration;
       restart_found_best = iteration;
       found_branching = node_branching();
       branching_factor = found_branching;
       write_reports();
}
if ( ant[iteration_best_ant].tour_length < restart_best_ant->tour_length ) {
       copy_from_to( &ant[iteration_best_ant], restart_best_ant );
       restart_found_best = iteration;
       printf("restart best: %ld, restart_found_best %ld, time %.2f\n",restart_best_ant-
>tour_length, restart_found_best, elapsed_time ( VIRTUAL ));
    }
}



void controll_statistics( void )
{
if (!(iteration % 100)) {
       population_statistics();
       branching_factor = node_branching(lambda);
       printf("\nbest so far %ld, iteration: %ld, time %.2f, b_fac %.5f\n",best_so_far_ant-
>tour_length,iteration,elapsed_time( VIRTUAL),branching_factor);

       if ( mmas_flag && (branching_factor < branch_fac)
              && (iteration - restart_found_best > 250) ) {
          printf("INIT TRAILS!!!\n"); restart_best_ant->tour_length = INFTY;
          init_pheromone_trails( trail_max );
          compute_total_information();
          restart_iteration = iteration;
          restart_time = elapsed_time( VIRTUAL );
       }
       printf("try %li, iteration %li, b-fac %f \n\n", n_try,iteration,branching_factor);
    }
}

void run_TSP_calculator() {
       for(k = 0; k < k_Cycle; k++)
       {
              if(ANT != 0)
              {
```

```
      for(i = 0; i < N; i++)
      {
            m_TABU[i] = 0;
      }
      cur_city = 0;
}
ii = 0;
gran = gran + koef_V;
if(ANT != 0)
{
      ch_rand = rand() % (ANTNum);
      ch_rand = ch_rand + ANT;
      if(ch_rand == 0)
            ch_rand = 10;
            znam = 0;
            for(i = 0; i < N; i++)
            {
                  if(i != cur_city)
                  {
                  znam = znam + powl((m_FER[i]),koef_A)
                  *powl((1/double(m_Dist[i])),koef_B);
                  }
            }
            buff[1] = 0;
            for(i = 0; i < N; i++)
            {
            if(i == cur_city)
            {
                  m_Imov[i] = 0;
            }
            else
            {
                  m_Imov[i] = (powl((m_FER[i]),koef_A)
                  *powl((1/double(m_Dist[i])),koef_B))
                  / znam;
                  for(j = 0; j<= ii; j++)
                  {
                        if(i == m_TABU[j])
                              m_Imov[i] = 0;
                  }
            }
            if(m_Imov[i] != 0)
                  buff[1] = 1;
                  if((m_Imov[i] < 0)||(m_Imov[i] >= 1))
            }
            j = 0; i_max = 0; buff[0] = 0;
            while(j <= ch_rand)
            {
            for(i = 0; i < N; i++)
            {
                  if(m_Imov[i] > 0)
                  {
                        if(i_max <= m_Imov[i])
                        {
                              i_max = m_Imov[i];
                              spec_res = i;
                              buff[0] = 1;
                        }
                  }
                  if((m_Imov[i] >= gran)&&(j<=ch_rand))
                  {
                        result = i;
                        j++;
                  }
```

```
                        }
                        if((j == 0)&&(buff[0] == 1))
                        {
                              result = spec_res;
                              j = ch_rand+1;
                        }
                        if(buff[0] == 0)
                        {
                              buff[0] = 1;
                              j = ch_rand+1;
                        }
                  }

                  cur_city = result;
            }
            ii++;
            if((ii == (N-1))&&(ANT!=0))
            {
                  m_TABU[ii] = cur_city;
            }
      }

}


void write_matrix(int N, int *m_A, int l, int m, long int T1, long int T2)
{
      FILE *pF2;
      pF2 = fopen ("res.txt", "w");
      if(!pF2)
      {
            cout<<"Can't create file!"<<endl;
      }
      else
      {
      char itoa_buf[9];
      int i;

      fputs("N = ",pF2);
      _itoa(N,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", k_Cycle = ",pF2);
      _itoa(k_Cycle,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", koef_A = ",pF2);
      _itoa(koef_A,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", koef_B = ",pF2);
      _itoa(koef_B,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", koef_P = ",pF2);
      _itoa(koef_P,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", koef_Q = ",pF2);
      _itoa(koef_Q,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", ANT_num = ",pF2);
      _itoa(m,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(", Time real = ",pF2);
      _itoa(T1,itoa_buf,10);
      fputs(itoa_buf,pF2);
      fputs(" ms, Time coud be = ",pF2);
      _itoa(T2,itoa_buf,10);
```

```
	fputs(itoa_buf,pF2);
	fputs(" ms \n Best way length = ",pF2);
	_itoa(l,itoa_buf,10);
	fputs(itoa_buf,pF2);
	fputs(" \n",pF2);

	for(i = 0; i < N; i++)
	{
			_itoa((m_A[i]),itoa_buf,10);
			fputs(itoa_buf,pF2);
			fputs(" ",pF2);
	}
	_itoa(Start_city,itoa_buf,10);
	fputs(itoa_buf,pF2);
	fputs("; \n",pF2);
	fclose(pF2);
	}
}


void read_matrix(int **m_A)
{
	FILE *pF1;

	pF1 = fopen ("l_1.txt", "r");
	if(!pF1)
	{
		cout<<"Can't open file!"<<endl;
	}
	else
	{
	char atoi_buf[5];
	bool b_per = false;
	int i=0;
	int j=0;
	char pl = '1';
	memset(atoi_buf,0,sizeof(atoi_buf));
	int ll = 0;
	while(pl != EOF)
	{
		pl = fgetc(pF1);
		if((pl == ' ')||(pl == ';'))
		{
			ll = 0;
			if(pl == ';')
			{
				i++; j = 0;
				b_per = true;
			}
			else if(b_per != true)
			{
				m_A[i][j] = atoi(atoi_buf);
				j++;
				b_per = true;
			}
			memset(atoi_buf,0,sizeof(atoi_buf));
		}
		else if((pl >= '0')&&(pl <= '9'))
		{
			b_per = false;
			atoi_buf[ll] = pl;
			ll++;
		}
	}
```

```
        fclose(pF1);
        }
}


void pheromone_update( void )
{
longint   k;
for ( k = 0 ; k < n_ants ; k++ )
        specified_pheromone_update( &ant[k] );
}

void pheromone_recalculation ( void )
{
        evaporation();
        pheromone_update();
        check_pheromone_trail_limits();
        compute_total_information();

}


/* --- main program ------------------------------------------------------ */

int main(int argc, char *argv[]) {
/*
      FUNCTION:       main control for running the ACO algorithms
      INPUT:          none
      OUTPUT:         none
      (SIDE)EFFECTS:  none
      COMMENTS:       this function controls the run of "max_tries" independent trials
*/
        long my_aco_flag = 0;
        long opt;
        longint i,j,k;
        int myiter;
        char fname[150];
        char* chptr = fname;

        long temp;

        double min_fetch, aver_fetch, max_fetch;
        long all_fetch, min_aver, max_aver, total_aver, total_all;

        FILE* fnoopt;
        FILE* nopt;
        FILE* topt;
        FILE* hopt;
        FILE* ropt;

        longint time2opt = 0, time2hopt = 0, time3opt = 0, timecycle;
        int start,end;

        start_timers();

        init_program(argc, argv);
        elitist_ants = n_ants * 0.1;
        if(elitist_ants < 3) elitist_ants = 3;
        instance.nn_list = compute_nn_lists();
        pheromone = generate_double_matrix( n, n );
        cost = generate_double_matrix( n, n );
        time_used = elapsed_time( VIRTUAL );
        printf("Initialization took %.10f seconds\n",time_used);

        sprintf(chptr,"TSP(%d)tryN.txt",n,n_try);
```

```
        File = fopen (fname, "w");

for ( n_try = 0 ; n_try < max_tries ; n_try++ ) {
        create_try_task();
}

fputs("\n=================================================\n",File);
sprintf(chptr,"best_found->tour_lenght=%d\n",best_so_far_ant->tour_length);
fputs(fname,File);
fputs("[",File);
for(myiter = 0; myiter < n; myiter++)
{
        sprintf(chptr,"%d ",best_so_far_ant->tour[myiter]);
        fputs(fname,File);
}
fputs("]",File);

fclose(File);

exit_program();
free( instance.distance );
free( instance.nn_list );
free( pheromone );
free( total );
free( best_in_try );
free( best_found_at );
free( time_best_found );
free( time_total_run );
for ( i = 0 ; i < n_ants ; i++ ) {
        free( ant[i].tour );
        free( ant[i].visited );
}
free( ant );
free( best_so_far_ant->tour );
free( best_so_far_ant->visited );
free( try_of_selection );

return(0);
}

void create_try_task() {

        // initialise pheromone lvl at the beginning, write to files.
        init_try(n_try);
        memset(fname,0,sizeof(fname));
        sprintf(chptr,"stat_opt_%s_try=%d.txt",instance.name,n_try);
        chptr = &fname[0];
        fnoopt = fopen(chptr,"wt");
        sprintf(chptr,"all_opt_way_%s_try=%d.txt",instance.name,n_try);
        nopt = fopen(chptr,"wt");
        sprintf(chptr,"\"iter\";\"ant number\";\"noopt tour length\";\"2opt tour
length\";\"2,5opt tour length\";\"3opt tour length\";\"2opt improves\";\"2,5opt
improves\";\"3opt improves\";\n");
        fputs(fname,fnoopt);
        max_tours = n_ants * 1000;
while ( !termination_condition() ) {
        ThreadTask[] tasks = construct_solutions();
        run_tasks(tasks);
        opt = recalculate_opt();
        local_search_check();
        update_statistics();
        pheromone_update();
        update_statistics();
        iteration++;
}
```

```
fclose(fnoopt);
fclose(nopt);
exit_try(n_try);

}


void my_pheromon_intital(void) {
	double koef_Q = instance.distance[1][2] * n;
	int i,j;
	for(i = 0; i < n; i++) {
		for(j = 0; j < n; j++) {
			if(i != j) {
				pheromone[i][j] = koef_Q / (double)(instance.distance[i][j]);
			} else {
				pheromone[i][j] = 0;
			}
		}
	}
}



void route_creation(void)
{
	int ch_rand, ch_rand2, sel, result;
	longdouble chance, chance2;
	longdouble gran = 0;
	longdouble* m_Imov;
	longdouble* m_pool;
	int* pool_city_list;
	int size_of_city_pool;
	int Start_city = 0;
	int ii = 0;
	int k, i, j, jj;
	int cur_city;
	longdouble znam;
	long b_found;
	double koef_A, koef_B;
	int im_count;
	koef_A = alpha; koef_B = beta;
	if((m_Imov = malloc(sizeof( longdouble ) * n)) == NULL){
			printf("Out of memory, exit.");
			exit(1);
	}
	if((m_pool = malloc(sizeof( longdouble ) * n)) == NULL){
		printf("Out of memory, exit.");
		exit(1);
	}
	if((pool_city_list = malloc(sizeof( int ) * n)) == NULL){
		printf("Out of memory, exit.");
		exit(1);
	}
	srand(seed);

	// make total information table:
	for(i = 0 ; i < n ; i++)
		for(k = 0 ; k < n ; k++)
		total[k][i] = powl((pheromone[k][i]), koef_A)
			*powl((1/(double)(instance.distance[k][i])),koef_B);

	for ( k = 0 ; k < n_ants ; k++) {
		ant_empty_memory( &ant[k] );
		ant[k].tour[0] = Start_city;
		ant[k].tour[n] = Start_city;
```

```
		ant[k].visited[Start_city] = 1;
	}

	while(ii != (n-1)) {

		//////.....CHOOSE NEXT CITY.....////////

		for ( k = 0 ; k < n_ants ; k++) {
			cur_city = ant[k].tour[ii];
			znam = 0;
			im_count = 0;

			// count znam value;
			for(i = 0; i < n; i++) {
				if(i != cur_city) {
					if(ant[k].visited[i] != 1) {
						znam = znam + total[cur_city][i];

						pool_city_list[im_count] = i;
						im_count++;
					}
				}
			}
			size_of_city_pool = im_count;
			im_count = 0;

			for(i = 0; i < n; i++) {
				// CALCULATE POSSIBILITIES:
				if (i == cur_city || ant[k].visited[i] != 0) {
					m_Imov[i] = -1;
				} else {
				//	if(ant[k].visited[i] == 1)
				//		printf("bad check %d ant %d city\n", k, sel);
					m_Imov[i] = (total[cur_city][i])/ znam;

				}
				// CALCULATE POOL FOR AIMING NEXT CITY:

				if(m_Imov[i] > 0) {
					if(im_count == 0) {
						m_pool[im_count] = m_Imov[i];
					} else {
						m_pool[im_count] = m_pool[im_count-1] + m_Imov[i];
					}
					im_count++;
				}
			}

			if (znam == 0) {
				printf("znam - is zero\n");
			}

			gran = 1.00 / im_count;
			b_found = 0;
			while(b_found == 0) {
				ch_rand = rand() % 101; //(ANTNum);
				chance = ((double)(ch_rand) / 100.00);// * gran;
				if(chance < gran)
					chance = gran / 1.5;
				for(i = 0; i < size_of_city_pool; i++) {
					if(chance > m_pool[i])	{
						sel = i;

					} else {
						sel = i;
```

```
                                        i=n;
                                }
                        }
                        if(chance < gran) {
                                ch_rand2 = rand() % 101;
                                chance2 = ((double)(ch_rand2) / 100.00);
                                if((chance2) < m_Imov[(pool_city_list[sel])]) {
                                        b_found = 1;
                                } else {
                                        b_found = 0;
                                }
                                result = sel;
                        } else {
                                b_found = 1;
                                result = sel;
                        }
                }
                if(ii < (n-1)) {
                        ant[k].tour[ii+1] = pool_city_list[sel];
                        if(ant[k].visited[pool_city_list[sel]] == 1)
                                printf("again %d ant %d city\n", k, pool_city_list[sel]);
                        ant[k].visited[pool_city_list[sel]] = 1;
                }
        }
        ii++;
    }
    for ( k = 0 ; k < n_ants ; k++) {
        ant[k].tour_length = compute_tour_length( ant[k].tour );
    }
    free(m_Imov);
    free(m_pool);
    free(pool_city_list);
}


void specific_update_pheromone(void)
{
    longdouble dbuff;
    int i,j,k;
    int city1, city2;
    int lenght;
    for ( k = 0 ; k < n_ants ; k++) {
        lenght = ant[k].tour_length;
        for ( i = 0; i < n; i++) {
            city1 = ant[k].tour[i];
            city2 = ant[k].tour[i+1];
            pheromone[city1][city2] =
(double)(instance.distance[city1][city2])/(double)(lenght);
        }
    }

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i != j) {
                dbuff = pheromone[i][j] *(1 - rho);
                pheromone[i][j] = dbuff; //koef_Q / double(m_A[i][j]);
            }
        }
    }
}
```

Лістинг програмного модуля на базі методів локальної оптимізації

LocalSearchModule.c:

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

#include"ls.h"
#include"InOut.h"
#include"TSP.h"
#include"ants.h"
#include"utilities.h"


longint opt;
longint count_search_checks;

longint d_option = TRUE;

longint *generate_permutation(longint n)
{
    longint  i, buffer, cnode, total_nodes_assigned = 0;
    double   random;
    longint  *r;

    r = malloc(n * sizeof(longint));

    for ( i = 0 ; i <n; i++)
        r[i] = i;

    for ( i = 0 ; i <n ; i++ ) {
        random  = randomVal1( &seed );
        cnode = (longint) (random  * (n - total_nodes_assigned));
        assert( i + cnode <n );
        buffer = r[i];
        r[i] = r[i+cnode];
        r[i+cnode] = buffer;
        total_nodes_assigned++;
    }
    return r;
}

long call_two_opt_search( longint *route )
{
    longint node1, node2;
    longint s_node1, s_node2;
    longint p_node1, p_node2;
    longint node_position_node1, node_position_node2;
    longint i, j, h, l;
    longint swap_option, swap_cnode, buffer, swap_count = 0,swap_try=0;
    longint hiracle1=0, hiracle2=0, hiracle3=0, hiracle4=0;
    longint R_dist;
    longint graduler = 0;
    longint *random_numbers_vector;
    longint *node_position;
    longint *doubleVal;

    node_position = malloc(n * sizeof(longint));
    doubleVal = malloc(n * sizeof(longint));
    for ( i = 0 ; i < n ; i++ ) {
        node_position[route[i]] = i;
        doubleVal[i] = FALSE;
    }

    swap_option = TRUE;
    random_numbers_vector = generate_permutation(n);

    while ( swap_option ) {
```

```
        swap_option = FALSE;

        for (l = 0 ; l < n; l++) {

            node1 = random_numbers_vector[l];
            DEBUG ( assert ( node1 < n && node1 >= 0); )
            if ( d_option && doubleVal[node1] )
                continue;
            swap_cnode = FALSE;
            node_position_node1 = node_position[node1];
            s_node1 = route[node_position_node1+1];
            R_dist = instance.costs[node1][s_node1];

            for ( h = 0 ; h < count_search_checks ; h++ ) {
                node2 = instance.pool_nodes[node1][h];
                if ( R_dist > instance.costs[node1][node2] ) {
                    s_node2 = route[node_position[node2]+1];
                    graduler =  - R_dist + instance.costs[node1][node2] +
                            instance.costs[s_node1][s_node2] -
instance.costs[node2][s_node2];
                    if ( graduler < 0 ) {
                        hiracle1 = node1; hiracle2 = s_node1; hiracle3 = node2; hiracle4
= s_node2;
                        swap_cnode = TRUE;
                        goto swapExecutor2opt;
                    }
                }
                else
                    break;
            }
            if (node_position_node1 > 0)
                p_node1 = route[node_position_node1-1];
            else
                p_node1 = route[n-1];
            R_dist = instance.costs[p_node1][node1];
            for ( h = 0 ; h < count_search_checks ; h++ ) {
                node2 = instance.pool_nodes[node1][h];
                if ( R_dist > instance.costs[node1][node2] ) {
                    node_position_node2 = node_position[node2];
                    if (node_position_node2 > 0)
                        p_node2 = route[node_position_node2-1];
                    else
                        p_node2 = route[n-1];
                    if ( p_node2 == node1 )
                        continue;
                    if ( p_node1 == node2 )
                        continue;
                    graduler =  - R_dist + instance.costs[node1][node2] +
                            instance.costs[p_node1][p_node2] -
instance.costs[p_node2][node2];
                    if ( graduler < 0 ) {
                        hiracle1 = p_node1; hiracle2 = node1; hiracle3 = p_node2;
hiracle4 = node2;
                        swap_cnode = TRUE;
                        goto swapExecutor2opt;
                    }
                }
                else
                    break;
            }
            if (swap_cnode) {
                swapExecutor2opt:
               swap_try++;
                swap_option = TRUE;
                doubleVal[hiracle1] = FALSE; doubleVal[hiracle2] = FALSE;
                doubleVal[hiracle3] = FALSE; doubleVal[hiracle4] = FALSE;
                if ( node_position[hiracle3] < node_position[hiracle1] ) {
                    buffer = hiracle1; hiracle1 = hiracle3; hiracle3 = buffer;
                    buffer = hiracle2; hiracle2 = hiracle4; hiracle4 = buffer;
                }
```

```
                if ( node_position[hiracle3] - node_position[hiracle2] < n / 2 + 1) {
                    i = node_position[hiracle2]; j = node_position[hiracle3];
                    while (i < j) {
                        node1 = route[i];
                        node2 = route[j];
                        route[i] = node2;
                        route[j] = node1;
                        node_position[node1] = j;
                        node_position[node2] = i;
                        i++; j--;
                    }
                }
                else {
                    i = node_position[hiracle1]; j = node_position[hiracle4];
                    if ( j > i )
                        buffer = n - (j - i) + 1;
                    else
                        buffer = (i - j) + 1;
                    buffer = buffer / 2;
                    for ( h = 0 ; h < buffer; h++ ) {
                        node1 = route[i];
                        node2 = route[j];
                        route[i] = node2;
                        route[j] = node1;
                        node_position[node1] = j;
                        node_position[node2] = i;
                        i--; j++;
                        if ( i < 0 )
                            i = n-1;
                        if ( j >= n )
                            j = 0;
                    }
                    route[n] = route[0];
                }
            } else {
                doubleVal[node1] = TRUE;
            }

        }
        if ( swap_option ) {
            swap_count++;
        }
    }

    free( random_numbers_vector );
    free( doubleVal );
    free( node_position );
    return swap_try;
}

long call_two_half_opt_search( longint *route )
{

    longint node1, node2;
    longint s_node1, s_node2;
    longint p_node1, p_node2;
    longint node_position_node1, node_position_node2;
    longint i, j, h, l;
    longint swap_option, swap_cnode;
    longint hiracle1=0, hiracle2=0, hiracle3=0, hiracle4=0, hiracle5=0, buffer;
    longint R_dist;
    longint graduler = 0;
    longint *random_numbers_vector;
    longint two_opt_step, cnode_move;
    longint two_half_opt_selector = 0;

    longint *node_position;
    longint *doubleVal;

    node_position = malloc(n * sizeof(longint));
    doubleVal = malloc(n * sizeof(longint));
```

```
    for ( i = 0 ; i < n ; i++ ) {
        node_position[route[i]] = i;
        doubleVal[i] = FALSE;
    }

    swap_option = TRUE;
    random_numbers_vector = generate_permutation(n);

    while ( swap_option ) {

        swap_option = FALSE; two_opt_step = FALSE; cnode_move = FALSE;

        for (l = 0 ; l < n; l++) {

            node1 = random_numbers_vector[l];
            DEBUG ( assert ( node1 < n && node1 >= 0); )
            if ( d_option && doubleVal[node1] )
                continue;
            swap_cnode = FALSE;
            node_position_node1 = node_position[node1];
            s_node1 = route[node_position_node1+1];
            R_dist = instance.costs[node1][s_node1];

            for ( h = 0 ; h < count_search_checks ; h++ ) {
                node2 = instance.pool_nodes[node1][h];
                if ( R_dist > instance.costs[node1][node2] ) {
                    node_position_node2 = node_position[node2];
                    s_node2 = route[node_position_node2+1];
                    graduler =  - R_dist + instance.costs[node1][node2] +
                                instance.costs[s_node1][s_node2] -
instance.costs[node2][s_node2];
                    if ( graduler < 0 ) {
                        hiracle1 = node1; hiracle2 = s_node1; hiracle3 = node2; hiracle4
= s_node2;
                        swap_cnode = TRUE; two_opt_step = TRUE; cnode_move = FALSE;
                        goto swapExecutor;
                    }
                    if (node_position_node2 > 0)
                        p_node2 = route[node_position_node2-1];
                    else
                        p_node2 = route[n-1];
                    graduler = - R_dist + instance.costs[node1][node2] +
instance.costs[node2][s_node1]
                           + instance.costs[p_node2][s_node2] -
instance.costs[node2][s_node2]
                           - instance.costs[p_node2][node2];
                    if ( node2 == s_node1 )
                        graduler = 0;
                    if ( p_node2 == s_node1 )
                        graduler = 0;

                    graduler = 0;

                    if ( graduler < 0 ) {
                        hiracle1 = node1; hiracle2 = s_node1; hiracle3 = node2; hiracle4
= p_node2; hiracle5 = s_node2;
                        swap_cnode = TRUE; cnode_move = TRUE; two_opt_step = FALSE;
                        goto swapExecutor;
                    }
                }
                else
                    break;
            }

            if (node_position_node1 > 0)
                p_node1 = route[node_position_node1-1];
            else
                p_node1 = route[n-1];
            R_dist = instance.costs[p_node1][node1];
            for ( h = 0 ; h < count_search_checks ; h++ ) {
                node2 = instance.pool_nodes[node1][h];
```

```
                if ( R_dist > instance.costs[node1][node2] ) {
                    node_position_node2 = node_position[node2];
                    if (node_position_node2 > 0)
                        p_node2 = route[node_position_node2-1];
                    else
                        p_node2 = route[n-1];
                    if ( p_node2 == node1 )
                        continue;
                    if ( p_node1 == node2 )
                        continue;
                    graduler =  - R_dist + instance.costs[node1][node2] +
                            instance.costs[p_node1][p_node2] -
instance.costs[p_node2][node2];
                    if ( graduler < 0 ) {
                        hiracle1 = p_node1; hiracle2 = node1; hiracle3 = p_node2;
hiracle4 = node2;
                        swap_cnode = TRUE; two_opt_step = TRUE; cnode_move = FALSE;
                        goto swapExecutor;
                    }
                    s_node2 = route[node_position[node2]+1];
                    graduler = - R_dist + instance.costs[node2][node1] +
instance.costs[p_node1][node2]
                           + instance.costs[p_node2][s_node2] -
instance.costs[node2][s_node2]
                            - instance.costs[p_node2][node2];
                    if ( p_node1 == node2 )
                        graduler = 0;
                    if ( p_node1 == s_node2 )
                        graduler = 0;

                    if ( graduler < 0 ) {
                        hiracle1 = p_node1; hiracle2 = node1; hiracle3 = node2; hiracle4
= p_node2; hiracle5 = s_node2;
                        swap_cnode = TRUE; cnode_move = TRUE; two_opt_step = FALSE;
                        goto swapExecutor;
                    }
                }
                else
                    break;
            }
            swapExecutor:
            if (swap_cnode) {
                if ( two_opt_step ) {
                    swap_option = TRUE;
                    doubleVal[hiracle1] = FALSE; doubleVal[hiracle2] = FALSE;
                    doubleVal[hiracle3] = FALSE; doubleVal[hiracle4] = FALSE;

                    if ( node_position[hiracle3] < node_position[hiracle1] ) {
                        buffer = hiracle1; hiracle1 = hiracle3; hiracle3 = buffer;
                        buffer = hiracle2; hiracle2 = hiracle4; hiracle4 = buffer;
                    }
                    if ( node_position[hiracle3] - node_position[hiracle2] < n / 2 + 1) {

                        i = node_position[hiracle2]; j = node_position[hiracle3];
                        while (i < j) {
                            node1 = route[i];
                            node2 = route[j];
                            route[i] = node2;
                            route[j] = node1;
                            node_position[node1] = j;
                            node_position[node2] = i;
                            i++; j--;
                        }
                    }
                    else {

                        i = node_position[hiracle1]; j = node_position[hiracle4];
                        if ( j > i )
                            buffer = n - (j - i) + 1;
                        else
                            buffer = (i - j) + 1;
```

```
                        buffer = buffer / 2;
                        for ( h = 0 ; h < buffer; h++ ) {
                            node1 = route[i];
                            node2 = route[j];
                            route[i] = node2;
                            route[j] = node1;
                            node_position[node1] = j;
                            node_position[node2] = i;
                            i--; j++;
                            if ( i < 0 )
                                i = n-1;
                            if ( j >= n )
                                j = 0;
                        }
                        route[n] = route[0];
                    }
                } elseif ( cnode_move ) {
                    swap_option = TRUE;
                    doubleVal[hiracle1] = FALSE; doubleVal[hiracle2] = FALSE;
doubleVal[hiracle3] = FALSE;
                    doubleVal[hiracle4] = FALSE; doubleVal[hiracle5] = FALSE;

                    if ( node_position[hiracle3] < node_position[hiracle1] ) {
                        buffer = node_position[hiracle1] - node_position[hiracle3];
                        i = node_position[hiracle3];
                        for ( h = 0 ; h < buffer; h++ ) {
                            node1 = route[i+1];
                            route[i] = node1;
                            node_position[node1] = i;
                            i++;
                        }
                        route[i] = hiracle3;
                        node_position[hiracle3] = i;
                        route[n] = route[0];
                    } else {

                        buffer = node_position[hiracle3] - node_position[hiracle1];

                        i = node_position[hiracle3];
                        for ( h = 0 ; h < buffer - 1 ; h++ ) {
                            node1 = route[i-1];
                            route[i] = node1;
                            node_position[node1] = i;
                            i--;
                        }
                        route[i] = hiracle3;
                        node_position[hiracle3] = i;
                        route[n] = route[0];
                    }
                } else {
                    exit(0);
                }
                two_opt_step = FALSE; cnode_move = FALSE;
            } else {
                doubleVal[node1] = TRUE;
            }

            if(swap_option)
                two_half_opt_selector++;
        }
    }
    free( random_numbers_vector );
    free( doubleVal );
    free( node_position );
    return two_half_opt_selector;
}

long call_three_opt_search( longint *route )
{

    longint   node1, node2, node3;
```

```
longint   s_node1, s_node2, s_node3;
longint   p_node1, p_node2, p_node3;
longint   node_position_node1, node_position_node2, node_position_node3;
longint   i, j, h, g, l, k;
longint   swap_option, buffer;
longint   hiracle1=0, hiracle2=0, hiracle3=0, hiracle4=0, hiracle5=0, hiracle6=0;
longint   diffs, diffp;
longint   between = FALSE;
longint   opt_two_option;
longint   move_step_option;
longint graduler, move_value, R_dist, add_buffer1, add_buffer2;
longint dbreak_count;
longint val[3];
longint nBuffer1, nBuffer2, nBuffer3;
longint *node_position;
longint *doubleVal;
longint *h_route;
longint *hh_route;
longint *random_numbers_vector;

longint three_opt_fetch = 0;

node_position = malloc(n * sizeof(longint));
doubleVal = malloc(n * sizeof(longint));
h_route = malloc(n * sizeof(longint));
hh_route = malloc(n * sizeof(longint));

for ( i = 0 ; i < n ; i++ ) {
    node_position[route[i]] = i;
    doubleVal[i] = FALSE;
}
swap_option = TRUE;
random_numbers_vector = generate_permutation(n);

while ( swap_option ) {
    move_value = 0;
    swap_option = FALSE;

    for ( l = 0 ; l < n ; l++ ) {

        node1 = random_numbers_vector[l];
        if ( d_option && doubleVal[node1] )
            continue;
        opt_two_option = FALSE;

        move_step_option = 0;
        node_position_node1 = node_position[node1];
        s_node1 = route[node_position_node1+1];
        if (node_position_node1 > 0)
            p_node1 = route[node_position_node1-1];
        else
            p_node1 = route[n-1];

        h = 0;
        while ( h < count_search_checks ) {

            node2   = instance.pool_nodes[node1][h];
            node_position_node2 = node_position[node2];
            s_node2 = route[node_position_node2+1];
            if (node_position_node2 > 0)
                p_node2 = route[node_position_node2-1];
            else
                p_node2 = route[n-1];

            diffs = 0; diffp = 0;

            R_dist = instance.costs[node1][s_node1];
            add_buffer1   = instance.costs[node1][node2];

            if ( R_dist > add_buffer1 ) {
                dbreak_count = - R_dist - instance.costs[node2][s_node2];
```

```
                    diffs =  dbreak_count + add_buffer1 +
instance.costs[s_node1][s_node2];
                    diffp =  - R_dist - instance.costs[node2][p_node2] +
                             instance.costs[node1][p_node2] +
instance.costs[s_node1][node2];
                }
                else
                    break;
                if ( p_node2 == node1 )
                    diffp = 0;
                if ( (diffs < move_value) || (diffp < move_value) ) {
                    swap_option = TRUE;
                    if (diffs <= diffp) {
                        hiracle1 = node1; hiracle2 = s_node1; hiracle3 = node2; hiracle4
= s_node2;
                        move_value = diffs;
                        opt_two_option = TRUE; move_step_option = 0;

                    } else {
                        hiracle1 = node1; hiracle2 = s_node1; hiracle3 = p_node2;
hiracle4 = node2;
                        move_value = diffp;
                        opt_two_option = TRUE; move_step_option = 0;

                    }
                }

                g = 0;
                while (g < count_search_checks) {

                    node3   = instance.pool_nodes[s_node1][g];
                    node_position_node3 = node_position[node3];
                    s_node3 = route[node_position_node3+1];
                    if (node_position_node3 > 0)
                        p_node3 = route[node_position_node3-1];
                    else
                        p_node3 = route[n-1];

                    if ( node3 == node1 ) {
                        g++;
                        continue;
                    }
                    else {
                        add_buffer2 = instance.costs[s_node1][node3];

                        if ( dbreak_count + add_buffer1 < add_buffer2 ) {

                            if ( node_position_node2 > node_position_node1 ) {
                                if ( node_position_node3 <= node_position_node2 &&
node_position_node3 > node_position_node1 )
                                    between = TRUE;
                                else
                                    between = FALSE;
                            }
                            elseif ( node_position_node2 < node_position_node1 )
                            if ( node_position_node3 > node_position_node1 ||
node_position_node3 < node_position_node2 )
                                between = TRUE;
                            else
                                between = FALSE;
                            else {

                                for (k = 0 ; k < n+1; k++ )
                                    printf("%d ",route[k]);
                                printf("\n");
                            }

                            if ( between ) {

                                graduler = dbreak_count - instance.costs[node3][p_node3]
+
```

```
                                    add_buffer1 + add_buffer2 +
                                    instance.costs[p_node3][s_node2];


                                if ( graduler < move_value ) {
                                    swap_option = TRUE;
                                    move_value = graduler;
                                    opt_two_option = FALSE;
                                    move_step_option = 1;

                                    hiracle1 = node1; hiracle2 = s_node1; hiracle3 =
node2; hiracle4 = s_node2; hiracle5 = p_node3; hiracle6 = node3;
                                    goto swapExecutor;
                                }
                            }
                            else {

                                graduler = dbreak_count - instance.costs[node3][s_node3]
+
                                    add_buffer1 + add_buffer2 +
                                    instance.costs[s_node2][s_node3];

                                if ( node_position_node2 == node_position_node3 ) {
                                    graduler = 20000;
                                }

                                if ( graduler < move_value ) {
                                    swap_option = TRUE;
                                    move_value = graduler;
                                    opt_two_option = FALSE;
                                    move_step_option = 2;
                                    hiracle1 = node1; hiracle2 = s_node1; hiracle3 =
node2; hiracle4 = s_node2; hiracle5 = node3; hiracle6 = s_node3;
                                    goto swapExecutor;
                                }

                                graduler = - R_dist - instance.costs[p_node2][node2]
                                    - instance.costs[p_node3][node3] +
                                    add_buffer1 + add_buffer2 +
                                    instance.costs[p_node2][p_node3];

                                if ( node3 == node2 || node2 == node1 || node1 == node3
|| p_node2 == node1 ) {
                                    graduler = 2000000;
                                }

                                if ( graduler < move_value ) {
                                    swap_option = TRUE;
                                    move_value = graduler;
                                    opt_two_option = FALSE;
                                    move_step_option = 3;
                                    hiracle1 = node1; hiracle2 = s_node1; hiracle3 =
p_node2; hiracle4 = node2; hiracle5 = p_node3; hiracle6 = node3;
                                    goto swapExecutor;
                                }

                                 graduler = - R_dist - instance.costs[p_node2][node2] -
                                    instance.costs[node3][s_node3]
                                    + add_buffer1 + add_buffer2 +
instance.costs[p_node2][s_node3];

                                     if ( graduler < move_value ) {
                                    swap_option = TRUE;
                                    move_value = graduler;
                                    opt_two_option = FALSE;
                                    move_step_option = 4;
                                    swap_option = TRUE;
                                    hiracle1 = node1; hiracle2 = s_node1; hiracle3 =
p_node2; hiracle4 = node2; hiracle5 = node3; hiracle6 = s_node3;
                                    goto swapExecutor;
                                }
```

```
                        }
                    }
                    else
                        g = count_search_checks + 1;
                }
                g++;
            }
            h++;
        }
        if ( move_step_option || opt_two_option ) {
            swapExecutor:
            move_value = 0;

            if ( move_step_option ) {
                doubleVal[hiracle1] = FALSE; doubleVal[hiracle2] = FALSE;
doubleVal[hiracle3] = FALSE;
                doubleVal[hiracle4] = FALSE; doubleVal[hiracle5] = FALSE;
doubleVal[hiracle6] = FALSE;
                node_position_node1 = node_position[hiracle1]; node_position_node2 =
node_position[hiracle3]; node_position_node3 = node_position[hiracle5];

                if ( move_step_option == 4 ) {

                    if ( node_position_node2 > node_position_node1 )
                        nBuffer1 = node_position_node2 - node_position_node1;
                    else
                        nBuffer1 = n - (node_position_node1 - node_position_node2);
                    if ( node_position_node3 > node_position_node2 )
                        nBuffer2 = node_position_node3 - node_position_node2;
                    else
                        nBuffer2 = n - (node_position_node2 - node_position_node3);
                    if ( node_position_node1 > node_position_node3 )
                        nBuffer3 = node_position_node1 - node_position_node3;
                    else
                        nBuffer3 = n - (node_position_node3 - node_position_node1);

                    val[0] = nBuffer1; val[1] = nBuffer2; val[2] = nBuffer3;

                    h = 0;
                    buffer = LONG_MIN;
                    for ( g = 0; g <= 2; g++) {
                        if ( buffer < val[g] ) {
                            buffer = val[g];
                            h = g;
                        }
                    }


                    if ( h == 0 ) {
                        j = node_position[hiracle4];
                        h = node_position[hiracle5];
                        i = 0;
                        h_route[i] = route[j];
                        nBuffer1 = 1;
                        while ( j != h) {
                            i++;
                            j++;
                            if ( j  >= n )
                                j = 0;
                            h_route[i] = route[j];
                            nBuffer1++;
                        }

                        j = node_position[hiracle4];
                        i = node_position[hiracle6];
                        route[j] = route[i];
                        node_position[route[i]] = j;
                        while ( i != node_position_node1) {
                            i++;
                            if ( i >= n )
                                i = 0;
```

```
            j++;
            if ( j >= n )
                j = 0;
            route[j] = route[i];
            node_position[route[i]] = j;
        }

        j++;
        if ( j >= n )
            j = 0;
        for ( i = 0; i<nBuffer1 ; i++ ) {
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }
        route[n] = route[0];
    }
    elseif ( h == 1 ) {

        j = node_position[hiracle6];
        h = node_position[hiracle1];
        i = 0;
        h_route[i] = route[j];
        nBuffer1 = 1;
        while ( j != h) {
            i++;
            j++;
            if ( j  >= n )
                j = 0;
            h_route[i] = route[j];
            nBuffer1++;
        }

        j = node_position[hiracle6];
        i = node_position[hiracle2];
        route[j] = route[i];
        node_position[route[i]] = j;
        while ( i != node_position_node2) {
            i++;
            if ( i >= n )
                i = 0;
            j++;
            if ( j >= n )
                j = 0;
            route[j] = route[i];
            node_position[route[i]] = j;
        }

        j++;
        if ( j >= n )
            j = 0;
        for ( i = 0; i<nBuffer1 ; i++ ) {
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }
        route[n] = route[0];
    }
    elseif ( h == 2 ) {

        j = node_position[hiracle2];
        h = node_position[hiracle3];
        i = 0;
        h_route[i] = route[j];
        nBuffer1 = 1;
        while ( j != h) {
            i++;
```

```
                    j++;
                    if ( j  >= n )
                        j = 0;
                    h_route[i] = route[j];
                    nBuffer1++;
                }

                j = node_position[hiracle2];
                i = node_position[hiracle4];
                route[j] = route[i];
                node_position[route[i]] = j;
                while ( i != node_position_node3) {
                    i++;
                    if ( i >= n )
                        i = 0;
                    j++;
                    if ( j >= n )
                        j = 0;
                    route[j] = route[i];
                    node_position[route[i]] = j;
                }

                j++;
                if ( j >= n )
                    j = 0;
                for ( i = 0; i<nBuffer1 ; i++ ) {
                    route[j] = h_route[i];
                    node_position[h_route[i]] = j;
                    j++;
                    if ( j >= n )
                        j = 0;
                }
                route[n] = route[0];
            }
        }
        elseif ( move_step_option == 1 ) {

            if ( node_position_node3 < node_position_node2 )
                nBuffer1 = node_position_node2 - node_position_node3;
            else
                nBuffer1 = n - (node_position_node3 - node_position_node2);
            if ( node_position_node3 > node_position_node1 )
                nBuffer2 = node_position_node3 - node_position_node1 + 1;
            else
                nBuffer2 = n - (node_position_node1 - node_position_node3 +
1);
            if ( node_position_node2 > node_position_node1 )
                nBuffer3 = n - (node_position_node2 - node_position_node1 +
1);
            else
                nBuffer3 = node_position_node1 - node_position_node2 + 1;

            val[0] = nBuffer1; val[1] = nBuffer2; val[2] = nBuffer3;
            h = 0;
            buffer = LONG_MIN;
            for ( g = 0; g <= 2; g++) {
                if ( buffer < val[g] ) {
                    buffer = val[g];
                    h = g;
                }
            }

            if ( h == 0 ) {

                j = node_position[hiracle5];
                h = node_position[hiracle2];
                i = 0;
                h_route[i] = route[j];
                nBuffer1 = 1;
                while ( j != h ) {
                    i++;
```

```
            j--;
            if ( j < 0 )
                j = n-1;
            h_route[i] = route[j];
            nBuffer1++;
        }

        j = node_position[hiracle1];
        h = node_position[hiracle4];
        i = 0;
        hh_route[i] = route[j];
        nBuffer2 = 1;
        while ( j != h) {
            i++;
            j--;
            if ( j < 0 )
                j = n-1;
            hh_route[i] = route[j];
            nBuffer2++;
        }

        j = node_position[hiracle4];
        for ( i = 0; i< nBuffer2 ; i++ ) {
            route[j] = hh_route[i];
            node_position[hh_route[i]] = j;
            j++;
            if (j >= n)
                j = 0;
        }

        for ( i = 0; i< nBuffer1 ; i++ ) {
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }
        route[n] = route[0];
    }
    elseif ( h == 1 ) {

        j = node_position[hiracle3];
        h = node_position[hiracle6];
        i = 0;
        h_route[i] = route[j];
        nBuffer1 = 1;
        while ( j != h) {
            i++;
            j--;
            if ( j  < 0 )
                j = n-1;
            h_route[i] = route[j];
            nBuffer1++;
        }

        j = node_position[hiracle6];
        i = node_position[hiracle4];

        route[j] = route[i];
        node_position[route[i]] = j;
        while ( i != node_position_node1) {
            i++;
            j++;
            if ( j >= n)
                j = 0;
            if ( i >= n)
                i = 0;
            route[j] = route[i];
            node_position[route[i]] = j;
        }
```

```
            j++;
            if ( j >= n )
                j = 0;
            i = 0;
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            while ( j != node_position_node1 ) {
                j++;
                if ( j >= n )
                    j = 0;
                i++;
                route[j] = h_route[i];
                node_position[h_route[i]] = j;
            }
            route[n] = route[0];
        }

        elseif ( h == 2 ) {

            j = node_position[hiracle2];
            h = node_position[hiracle5];
            i = 0;
            h_route[i] =  route[j];
            nBuffer1 = 1;
            while ( j != h ) {
                i++;
                j++;
                if ( j >= n )
                    j = 0;
                h_route[i] = route[j];
                nBuffer1++;
            }
            j = node_position_node2;
            h = node_position[hiracle6];
            i = 0;
            hh_route[i] = route[j];
            nBuffer2 = 1;
            while ( j != h) {
                i++;
                j--;
                if ( j < 0 )
                    j = n-1;
                hh_route[i] = route[j];
                nBuffer2++;
            }

            j = node_position[hiracle2];
            for ( i = 0; i< nBuffer2 ; i++ ) {
                route[j] = hh_route[i];
                node_position[hh_route[i]] = j;
                j++;
                if ( j >= n)
                    j = 0;
            }

            for ( i = 0; i< nBuffer1 ; i++ ) {
                route[j] = h_route[i];
                node_position[h_route[i]] = j;
                j++;
                if ( j >= n )
                    j = 0;
            }
            route[n] = route[0];
        }
    }
    elseif ( move_step_option == 2 ) {

        if ( node_position_node3 < node_position_node1 )
            nBuffer1 = node_position_node1 - node_position_node3;
        else
            nBuffer1 = n - (node_position_node3 - node_position_node1);
```

```
if ( node_position_node3 > node_position_node2 )
    nBuffer2 = node_position_node3 - node_position_node2;
else
    nBuffer2 = n - (node_position_node2 - node_position_node3);
if ( node_position_node2 > node_position_node1 )
    nBuffer3 = node_position_node2 - node_position_node1;
else
    nBuffer3 = n - (node_position_node1 - node_position_node2);

val[0] = nBuffer1; val[1] = nBuffer2; val[2] = nBuffer3;
h = 0;
buffer = LONG_MIN;
for ( g = 0; g <= 2; g++) {
    if ( buffer < val[g] ) {
        buffer = val[g];
        h = g;
    }
}

if ( h == 0 ) {

    j = node_position[hiracle3];
    h = node_position[hiracle2];
    i = 0;
    h_route[i] = route[j];
    nBuffer1 = 1;
    while ( j != h ) {
        i++;
        j--;
        if ( j < 0 )
            j = n-1;
        h_route[i] = route[j];
        nBuffer1++;
    }

    j = node_position[hiracle5];
    h = node_position[hiracle4];
    i = 0;
    hh_route[i] = route[j];
    nBuffer2 = 1;
    while ( j != h ) {
        i++;
        j--;
        if ( j < 0 )
            j = n-1;
        hh_route[i] = route[j];
        nBuffer2++;
    }

    j = node_position[hiracle2];
    for ( i = 0; i<nBuffer1 ; i++ ) {
        route[j] = h_route[i];
        node_position[h_route[i]] = j;
        j++;
        if ( j >= n )
            j = 0;
    }

    for ( i = 0; i < nBuffer2 ; i++ ) {
        route[j] = hh_route[i];
        node_position[hh_route[i]] = j;
        j++;
        if ( j >= n )
            j = 0;
    }
    route[n] = route[0];
}
elseif ( h == 1 ) {

    j = node_position[hiracle2];
    h = node_position[hiracle3];
```

```
    i = 0;
    h_route[i] = route[j];
    nBuffer1 = 1;
    while ( j != h ) {
        i++;
        j++;
        if ( j >= n  )
            j = 0;
        h_route[i] = route[j];
        nBuffer1++;
    }

    j = node_position[hiracle1];
    h = node_position[hiracle6];
    i = 0;
    hh_route[i] = route[j];
    nBuffer2 = 1;
    while ( j != h ) {
        i++;
        j--;
        if ( j < 0 )
            j =  n-1;
        hh_route[i] = route[j];
        nBuffer2++;
    }
    j = node_position[hiracle6];
    for ( i = 0; i<nBuffer1 ; i++ ) {
        route[j] = h_route[i];
        node_position[h_route[i]] = j;
        j++;
        if ( j >= n )
            j = 0;
    }
    for ( i = 0; i < nBuffer2 ; i++ ) {
        route[j] = hh_route[i];
        node_position[hh_route[i]] = j;
        j++;
        if ( j >= n )
            j = 0;
    }
    route[n] = route[0];
}

elseif ( h == 2 ) {

    j = node_position[hiracle1];
    h = node_position[hiracle6];
    i = 0;
    h_route[i] = route[j];
    nBuffer1 = 1;
    while ( j != h ) {
        i++;
        j--;
        if ( j < 0 )
            j = n-1;
        h_route[i] = route[j];
        nBuffer1++;
    }

    j = node_position[hiracle4];
    h = node_position[hiracle5];
    i = 0;
    hh_route[i] = route[j];
    nBuffer2 = 1;
    while ( j != h ) {
        i++;
        j++;
        if ( j >= n  )
            j = 0;
        hh_route[i] = route[j];
        nBuffer2++;
```

```
            }

            j = node_position[hiracle4];
            for ( i = 0; i<nBuffer1 ; i++ ) {
                route[j] = h_route[i];
                node_position[h_route[i]] = j;
                j++;
                if ( j >= n )
                    j = 0;
            }

            for ( i = 0; i < nBuffer2 ; i++ ) {
                route[j] = hh_route[i];
                node_position[hh_route[i]] = j;
                j++;
                if ( j >= n )
                    j = 0;
            }
            route[n] = route[0];
        }
    }
    elseif ( move_step_option == 3 ) {

        if ( node_position_node3 < node_position_node1 )
            nBuffer1 = node_position_node1 - node_position_node3;
        else
            nBuffer1 = n - (node_position_node3 - node_position_node1);
        if ( node_position_node3 > node_position_node2 )
            nBuffer2 = node_position_node3 - node_position_node2;
        else
            nBuffer2 = n - (node_position_node2 - node_position_node3);
        if ( node_position_node2 > node_position_node1 )
            nBuffer3 = node_position_node2 - node_position_node1;
        else
            nBuffer3 = n - (node_position_node1 - node_position_node2);

        val[0] = nBuffer1; val[1] = nBuffer2; val[2] = nBuffer3;
        h = 0;
        buffer = LONG_MIN;
        for ( g = 0; g <= 2; g++) {
            if ( buffer < val[g] ) {
                buffer = val[g];
                h = g;
            }
        }

        if ( h == 0 ) {

            j = node_position[hiracle3];
            h = node_position[hiracle2];
            i = 0;
            h_route[i] = route[j];
            nBuffer1 = 1;
            while ( j != h ) {
                i++;
                j--;
                if ( j < 0 )
                    j = n-1;
                h_route[i] = route[j];
                nBuffer1++;
            }

            j = node_position[hiracle2];
            h = node_position[hiracle5];
            i = node_position[hiracle4];
            route[j] = hiracle4;
            node_position[hiracle4] = j;
            while ( i != h ) {
                i++;
                if ( i >= n )
                    i = 0;
```

```
            j++;
            if ( j >= n )
                j = 0;
            route[j] = route[i];
            node_position[route[i]] = j;
        }
        j++;
        if ( j >= n )
            j = 0;
        for ( i = 0; i < nBuffer1 ; i++ ) {
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }
        route[n] = route[0];
    }
    elseif ( h == 1 ) {

        j = node_position[hiracle3];
        h = node_position[hiracle2];
        i = 0;
        h_route[i] = route[j];
        nBuffer1 = 1;
        while ( j != h ) {
            i++;
            j--;
            if ( j < 0  )
                j = n-1;
            h_route[i] = route[j];
            nBuffer1++;
        }

        j = node_position[hiracle6];
        h = node_position[hiracle1];
        i = 0;
        hh_route[i] = route[j];
        nBuffer2 = 1;
        while ( j != h ) {
            i++;
            j++;
            if ( j >= n )
                j = 0;
            hh_route[i] = route[j];
            nBuffer2++;
        }

        j = node_position[hiracle6];
        for ( i = 0; i<nBuffer1 ; i++ ) {
            route[j] = h_route[i];
            node_position[h_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }

        for ( i = 0 ; i < nBuffer2 ; i++ ) {
            route[j] = hh_route[i];
            node_position[hh_route[i]] = j;
            j++;
            if ( j >= n )
                j = 0;
        }
        route[n] = route[0];
    }

    elseif ( h == 2 ) {

        j = node_position[hiracle5];
        h = node_position[hiracle4];
```

```
                i = 0;
                h_route[i] = route[j];
                nBuffer1 = 1;
                while ( j != h ) {
                    i++;
                    j--;
                    if ( j < 0 )
                        j = n-1;
                    h_route[i] = route[j];
                    nBuffer1++;
                }

                j = node_position[hiracle1];
                h = node_position[hiracle6];
                i = 0;
                hh_route[i] = route[j];
                nBuffer2 = 1;
                while ( j != h ) {
                    i++;
                    j--;
                    if ( j < 0 )
                        j = n-1;
                    hh_route[i] = route[j];
                    nBuffer2++;
                }

                j = node_position[hiracle4];
                for ( i = 0; i< nBuffer1 ; i++ ) {
                    route[j] = h_route[i];
                    node_position[h_route[i]] = j;
                    j++;
                    if ( j >= n )
                        j = 0;
                }
                for ( i = 0; i< nBuffer2 ; i++ ) {
                    route[j] = hh_route[i];
                    node_position[hh_route[i]] = j;
                    j++;
                    if ( j >= n )
                        j = 0;
                }
                route[n] = route[0];
            }
        }
        else {
            printf(" Some very strange error must have occurred !!!\n\n");
            exit(0);
        }
    }
    if (opt_two_option) {

        doubleVal[hiracle1] = FALSE; doubleVal[hiracle2] = FALSE;
        doubleVal[hiracle3] = FALSE; doubleVal[hiracle4] = FALSE;
        if ( node_position[hiracle3] < node_position[hiracle1] ) {
            buffer = hiracle1; hiracle1 = hiracle3; hiracle3 = buffer;
            buffer = hiracle2; hiracle2 = hiracle4; hiracle4 = buffer;
        }
        if ( node_position[hiracle3]-node_position[hiracle2] < n / 2 + 1) {
            i = node_position[hiracle2]; j = node_position[hiracle3];
            while (i < j) {
                node1 = route[i];
                node2 = route[j];
                route[i] = node2;
                route[j] = node1;
                node_position[node1] = j;
                node_position[node2] = i;
                i++; j--;
            }
        }
        else {
            i = node_position[hiracle1]; j = node_position[hiracle4];
```

```
                        if ( j > i )
                            buffer = n - (j - i) + 1;
                        else
                            buffer = (i - j) + 1;
                        buffer = buffer / 2;
                        for ( h = 0 ; h < buffer ; h++ ) {
                            node1 = route[i];
                            node2 = route[j];
                            route[i] = node2;
                            route[j] = node1;
                            node_position[node1] = j;
                            node_position[node2] = i;
                            i--; j++;
                            if ( i < 0 )
                                i = n - 1;
                            if ( j >= n )
                                j = 0;
                        }
                        route[n] = route[0];
                    }
                }
            }
            else {
                doubleVal[node1] = TRUE;
            }

            if(swap_option)
                three_opt_fetch++;

        }
    }
    free( random_numbers_vector );
    free( h_route );
    free( hh_route );
    free( node_position );
    free( doubleVal );
    return three_opt_fetch;
}
```

Додаток З

Лістинг програмної реалізації багатоагентної системи для розв'язання динамічної асиметричної ЗК в умовах частково невідомих вхідних даних

Лістинг "EnvironmentData.h":

```
#include <vector>
#include <list>
#include "DigitalComplexMark.h"

#ifndef ENVIRONMENTDATA_H_
#define ENVIRONMENTDATA_H_
using namespace std;

class EnvironmentData {

private:
    vector<long> accessibilityData;
    vector<long> abstractTransferTimeData;
    long smartMarkSystemCount;

public:

    EnvironmentData(vector<long> accessibilityData, vector<long>
abstractTransferTimeData, long smartMarkSystemCount) {
        this->accessibilityData = accessibilityData;
        this->abstractTransferTimeData = abstractTransferTimeData;
        this->smartMarkSystemCount = smartMarkSystemCount;
    }

    vector<int> getAccessibleSmartMarkIndexes(int currentSmartMarkStationId) {
        vector<int> accessibleIds;
        for (int id = 1; id <= smartMarkSystemCount; id++) {
            if (id != currentSmartMarkStationId) {
                if (accessibilityData[(currentSmartMarkStationId - 1) *
smartMarkSystemCount + id - 1] == 1) {
                    accessibleIds.push_back(id);
                }
            }
        }
        return accessibleIds;
    }

    int getSystemCount() {
        return smartMarkSystemCount;
    }

    vector<long> getAbstractTransferTimeData(long currentMarkIndex) {
        vector<long>::const_iterator first = abstractTransferTimeData.begin() +
smartMarkSystemCount * currentMarkIndex;
        vector<long>::const_iterator last = abstractTransferTimeData.begin() +
smartMarkSystemCount * currentMarkIndex + smartMarkSystemCount;
        vector<long> rowData(first, last);
        return rowData;
    }

    void changeTransferTimeValue(long startMarkIndex, long endMarkIndex, long newValue) {
        abstractTransferTimeData[smartMarkSystemCount * startMarkIndex + endMarkIndex] =
newValue;
    }

    void changeAccessOfArc(long startMarkIndex, long endMarkIndex, long access) {
        accessibilityData[smartMarkSystemCount * startMarkIndex + endMarkIndex] = access;
    }

    void changeAccessOfNode(long startMarkIndex, long access) {
        for (int nodeIndex = 0; nodeIndex < smartMarkSystemCount; nodeIndex++) {
```

```
            accessibilityData[smartMarkSystemCount * startMarkIndex + nodeIndex] =
access;
            accessibilityData[smartMarkSystemCount * nodeIndex + startMarkIndex] =
access;
        }
    }

};

#endif /* ENVIRONMENTDATA_H_ */
```

Лістинг “EnvironmentSimulator.h”:

```
#include <vector>
#include <map>
#include "DigitalComplexMark.h"
#include "AgentPackageDataContainer.h"
#include "SimulationAction.h"
#include "LogFileWriter.h"

#ifndef ENVIRONMENTSIMULATOR_H_
#define ENVIRONMENTSIMULATOR_H_
using namespace std;

class EnvironmentSimulator {

private:
    EnvironmentData *data;
    long globalSpentTime;
    map<long, DigitalComplexMark *> mappedComplexMarks;
    map<long, vector<AgentPackageDataContainer *> > mappedResultPackages;
    map<long, AgentPackageDataContainer *> mappedAgentPackages;
    multimap<long, SimulationAction *> actionQueue;
    LogFileWriter *logFileWriter;
    LogFileWriter *routeFileWriter;
    LogFileWriter *markValueFileWriter;
    InputParameterData *inputParameterData;
    long bestFoundResult;
    long agentIdentificator;

public:
    EnvironmentSimulator(EnvironmentData *data, LogFileWriter *logFileWriter,
LogFileWriter *routeFileWriter,
            LogFileWriter *markValueFileWriter, InputParameterData
*inputParameterData, vector<SimulationAction *> scenarioActions) {
        this->data = data;
        this->logFileWriter = logFileWriter;
        this->routeFileWriter = routeFileWriter;
        this->markValueFileWriter = markValueFileWriter;
        this->inputParameterData = inputParameterData;
        globalSpentTime = 0;
        long stationCount = data->getSystemCount();
        for (long markId = 1; markId <= stationCount; markId++) {
            mappedComplexMarks.insert(std::pair<long, DigitalComplexMark
*>(markId, new DigitalComplexMark(markId, data, inputParameterData)));
        }
        for (int actionIndex = 0; actionIndex < scenarioActions.size();
actionIndex++) {
            SimulationAction *scenarioAction = scenarioActions[actionIndex];
            actionQueue.insert(std::pair<long, SimulationAction
*>(scenarioAction->executionTimeFromStart,
```

```
scenarioActions[actionIndex]));
        }
        for (map<long, SimulationAction *>::iterator action =
actionQueue.begin(); action != actionQueue.end();) {
            SimulationAction *nextAction = action->second;
            if (nextAction->actionType == NODE_TO_NODE_SEND_ACTION) {
                int a = 0;
            }
            action++;
        }
        bestFoundResult = 0;
        agentIdentificator = 0;
    }

    void startSimulation() {
        long cycleIndex = 0;
        char message[] = "Test message for wide sending";
        long agentCount = inputParameterData->getAgentCount();
        long startStationId = inputParameterData->getStartMarkId();
        long stationCount = data->getSystemCount();
        // lets assume to avoid dead locks as setting max node count as
totalCount * 2;
        long MAX_NODE_AT_ROUTE_LIMIT = data->getSystemCount() * 2;
        // initialize placing of agent packages in start Station.
        long startCycleTime = 0;
        placePackagesToStartSmartMark(message, agentCount, startStationId,
MAX_NODE_AT_ROUTE_LIMIT);
        while (globalSpentTime <= inputParameterData->getTotalSimulationTime())
{
            long nextTime = executeNextAction(globalSpentTime);
            // Id all packages returns:

            // Added condition of force reset after some period of time
(packages could not return yet in critical situation, otherway it will not get
here)
            if (globalSpentTime - startCycleTime >= inputParameterData-
>getWaitTime()) {
                nextTime = 0;
               // actionQueue.clear();
            }
            if (nextTime > globalSpentTime) {
                globalSpentTime = nextTime;
            } else {
                if (nextTime == 0) {
                    writeCycleSeparatorToFiles(cycleIndex);
                    // add collecting packages from all nodes and free data:
                    writeCycleMarkValuesMatrixToFile();
                    vector<AgentPackageDataContainer *> collectedPackages;
                    vector<AgentPackageDataContainer *> outputMessagesInProcess;
                    if (!actionQueue.empty()) {
                        for (map<long, SimulationAction *>::iterator action =
actionQueue.begin(); action != actionQueue.end();) {
                            SimulationAction *nextAction = action->second;
                            if (nextAction->actionType ==
NODE_TO_NODE_SEND_ACTION) {
                                outputMessagesInProcess.push_back(nextAction-
>agentPackage);
                            }
                            action++;
                        }
                        for (int packageIndex = 0; packageIndex <
outputMessagesInProcess.size(); packageIndex++) {
                            AgentPackageDataContainer *resultMessage =
outputMessagesInProcess[packageIndex];
```

```
                            vector<long> route = resultMessage->getRouteInfo();
                            writeResultCycleRoutesToFile(resultMessage-
>currentNodeId(), resultMessage->getPackageIdentificator(), resultMessage,
route, false);
                        }
                    }
                    for (long markId = 1; markId <= stationCount; markId++) {
                        vector<AgentPackageDataContainer *> outputMessages =
mappedComplexMarks[markId]->retrieveAllMessagesAndResetQueue();
                        pair<long, long> bestTimeResults =
bestTimeResult(outputMessages);
                        bool routeFinished = markId == startStationId;
                        for (int packageIndex = 0; packageIndex <
outputMessages.size(); packageIndex++) {
                            AgentPackageDataContainer *resultMessage =
outputMessages[packageIndex];
                            vector<long> route = resultMessage->getRouteInfo();
                            if (routeFinished) {

boostUpdateBestAndGoodResultRoutes(bestTimeResults, resultMessage, route);
                            }
                            writeResultCycleRoutesToFile(markId, resultMessage-
>getPackageIdentificator(), resultMessage, route, routeFinished);
                        }
                        if (routeFinished) {
                            if (bestFoundResult == 0 || bestFoundResult >
bestTimeResults.first) {
                                bestFoundResult = bestTimeResults.first;
                            }
                        }
                        if (!outputMessages.empty()) {
                            collectedPackages.insert(collectedPackages.end(),
outputMessages.begin(), outputMessages.end());
                        }

                    }

                    mappedResultPackages.insert(std::pair<long,
vector<AgentPackageDataContainer *> >(cycleIndex, collectedPackages));
                    // run cycle from start point again
                    placePackagesToStartSmartMark(message, agentCount,
startStationId, MAX_NODE_AT_ROUTE_LIMIT);
                    startCycleTime = globalSpentTime;
                    cycleIndex++;
                    continue;
                }
                break;
            }
        }
        // finalize searching:
        int createdPackages = mappedResultPackages[0].size();
        int finalization = 0;
    }

    void boostUpdateBestAndGoodResultRoutes(pair<long, long> bestTimeResults,
AgentPackageDataContainer *resultMessage, vector<long> route) {
        long bestCycleTime = bestTimeResults.first;
        long bestSecondTime = bestTimeResults.second;
        if (resultMessage->isRouteFinished() && resultMessage-
>getTotalTravelTime() <= bestSecondTime) {
            double boostCoefficient = inputParameterData-
>getGoodBoostResultCoefficient();
            if (resultMessage->getTotalTravelTime() < bestSecondTime ||
bestSecondTime == bestCycleTime) {
```

```
                boostCoefficient = inputParameterData-
>getBestBoostResultCoefficient();
                if (bestCycleTime < bestFoundResult) {
                    boostCoefficient += inputParameterData-
>getBestBoostResultCoefficient();
                }

            }
            vector<NodeConnection> uniqueConnections = resultMessage-
>getUniqueNodeConnections();
            for (int connectionIndex = 0; connectionIndex <
uniqueConnections.size(); connectionIndex++) {
                NodeConnection connection = uniqueConnections[connectionIndex];
                DigitalComplexMark *smartMarkToUpdate =
mappedComplexMarks[connection.firstNodeId];
                smartMarkToUpdate-
>updateMarkValueToTargetMarkNet(connection.secondNodeId, globalSpentTime,
boostCoefficient);
            }
        }
    }

    pair<long, long> bestTimeResult(vector<AgentPackageDataContainer *>
outputMessages) {
        long bestSecondResult = 0;
        long bestResult = 0;
        for (int packageIndex = 0; packageIndex < outputMessages.size();
packageIndex++) {
            AgentPackageDataContainer *package = outputMessages[packageIndex];
            if (package->isRouteFinished()) {
                long totalTravelTime = package->getTotalTravelTime();
                if (bestResult == 0 || bestResult > totalTravelTime) {
                    bestSecondResult = bestResult;
                    bestResult = totalTravelTime;
                    continue;
                }
                if ((bestSecondResult == 0 || bestSecondResult >
totalTravelTime) && totalTravelTime != bestResult) {
                    bestSecondResult = totalTravelTime;
                }
            }
        }
        int bestResultCount = 0;
        for (int packageIndex = 0; packageIndex < outputMessages.size();
packageIndex++) {
            AgentPackageDataContainer *package = outputMessages[packageIndex];
            if (bestResult == package->getTotalTravelTime()) {
                bestResultCount++;
            }
        }
        if (bestResultCount > outputMessages.size() / 3.0) {
            bestSecondResult = bestResult;
        }
        if (bestSecondResult > 0) {
            return pair<long, long>(bestResult, bestSecondResult);
        }
        return pair<long, long>(bestResult, bestResult);
    }

    void writeCycleSeparatorToFiles(long cycleIndex) {
        stringstream logMessage;
        logMessage << "CYCLE END = " << cycleIndex + 1 << endl;
        stringstream routeMessage;
        routeMessage << "CYCLE " << cycleIndex + 1 << " ROUTES:";
```

```
        stringstream markMessage;
        markMessage << endl << "CYCLE " << cycleIndex + 1 << " MARK VALUE
MARTIX:";
        logFileWriter->writeToLog(logMessage.str().c_str());
        routeFileWriter->writeToLog(routeMessage.str().c_str());
        markValueFileWriter->writeToLog(markMessage.str().c_str());
    }

    void writeResultCycleRoutesToFile(long markId, long packageId,
AgentPackageDataContainer *resultMessage, vector<long> route, bool
routeFinished) {
        stringstream routeMessage;
        routeMessage << "Route of [" << packageId << "] agent :";
        for (int nodeIndex = 0; nodeIndex < route.size(); nodeIndex++) {
            routeMessage << " " << route[nodeIndex];
        }
        routeMessage << " || SPENT TIME = " << resultMessage-
>getTotalTravelTime()
                << " currently at MarkId = " << markId;
        if (routeFinished) {
            routeMessage << " STATUS = FINISHED";
        } else {
            routeMessage << " STATUS = UNFINISHED";
        }
        routeFileWriter->writeToLog(routeMessage.str().c_str());
    }

    void writeCycleMarkValuesMatrixToFile() {
        stringstream routeMessage;
        for (map<long, DigitalComplexMark *>::iterator mark =
mappedComplexMarks.begin(); mark != mappedComplexMarks.end(); ++mark) {
            routeMessage << "[" << mark->first << "] :";
            DigitalComplexMark *smartMark = mark->second;
            vector<double> markValues = smartMark->getMemoryMarkValuesRow();
            for (int markIndex = 0; markIndex < markValues.size(); markIndex++)
{
                routeMessage << " " << markValues[markIndex];
            }
            routeMessage << endl;
        }
        markValueFileWriter->writeToLog(routeMessage.str().c_str());
    }

    void placePackagesToStartSmartMark(char message[], long agentCount, long
startStationId, long MAX_NODE_AT_ROUTE_LIMIT) {
        // initialize agent packages for sending;
        DigitalComplexMark *startComplexMark =
mappedComplexMarks[startStationId];
        for (long agentId = 1; agentId <= agentCount; agentId++) {
            // place them at start node
            AgentPackageDataContainer *agentPackageDataContainer = new
AgentPackageDataContainer(agentId + agentIdentificator, -1, startStationId,
                    MAX_NODE_AT_ROUTE_LIMIT, message);
            agentPackageDataContainer->placeAtNode(startStationId, 0);
            // retrieved accessible not visited smartMarks:
            startComplexMark->addMessageToQueue(agentPackageDataContainer);
            mappedAgentPackages.insert(std::pair<long, AgentPackageDataContainer
*>(agentId + agentIdentificator, agentPackageDataContainer));

        }
        agentIdentificator += agentCount;
        SimulationAction *startAction = new SimulationAction(globalSpentTime +
inputParameterData->getProcessingDataDelay(), SEND_FROM_NODE_QUEUE_ACTION,
                startStationId, 0, 0, NULL);
```

```
        actionQueue.insert(std::pair<long, SimulationAction *>(startAction-
>executionTimeFromStart,
                startAction));
    }

    long executeNextAction(long currentTime) {
        if (actionQueue.empty()) {
            cout << "Execute actions running, but no actions found" << endl;
            return 0;
        }
        while (actionQueue.begin()->first == currentTime) {
            vector<SimulationAction *> newActions;
            SimulationAction *nextAction = actionQueue.begin()->second;
            writeCurrentActionInfoToLog(nextAction);
            long startMarkId = nextAction->startSmartMarkId;
            long endSmartMarkId = nextAction->endSmartMarkId;
            DigitalComplexMark *startMark = mappedComplexMarks[startMarkId];
            switch (nextAction->actionType) {
                case SEND_FROM_NODE_QUEUE_ACTION:
                    newActions = runActionSendMessageFromQueue(startMark);
                    break;
                case NODE_TO_NODE_SEND_ACTION:
                    newActions = runNodeToNodeSendingAction(nextAction);
                    break;
                case CHANGE_NODE_TO_NODE_TIME:
                    data->changeTransferTimeValue(startMarkId - 1,
endSmartMarkId - 1, nextAction->newValue);
                    break;
                case SWITCH_ON_NODE:
                    data->changeAccessOfNode(startMarkId - 1, 1);
                    startMark->switchOnSystem();
                    break;
                case SWITCH_ON_ARC:
                    data->changeAccessOfArc(startMarkId - 1, endSmartMarkId - 1,
1);
                    break;
                case SWITCH_OFF_ARC:
                    data->changeAccessOfArc(startMarkId - 1, endSmartMarkId - 1,
0);
                    // revert sending by blocked arc:
                    revertSendingByArcActions(newActions, startMarkId,
endSmartMarkId, startMark);
                    break;
                case SWITCH_OFF_NODE:
                    data->changeAccessOfNode(startMarkId - 1, 0);
                    startMark->switchOffSystem();
                    // revert sending from \ to blocked node:
                    revertSendingByBlockedNode(newActions, startMarkId,
startMark);
                    break;
            }
            for (int index = 0; index < newActions.size(); index++) {
                long actionStartTime = newActions[index]-
>executionTimeFromStart;
                pair<long, SimulationAction *> timedActionInfo = std::pair<long,
SimulationAction *>(actionStartTime,
                        newActions[index]);

                bool found = false;
                std::pair<std::multimap<long, SimulationAction *>::iterator,
std::multimap<long, SimulationAction *>::iterator> ret;
                ret = actionQueue.equal_range(actionStartTime);
                for (std::multimap<long, SimulationAction *>::iterator it =
ret.first; it != ret.second; ++it) {
```

```
                    if (it->second->actionType == timedActionInfo.second-
>actionType
                            && it->second->agentPackage ==
timedActionInfo.second->agentPackage
                            && it->second->startSmartMarkId ==
timedActionInfo.second->startSmartMarkId
                            && it->second->endSmartMarkId ==
timedActionInfo.second->endSmartMarkId
                            && it->second->newValue == timedActionInfo.second-
>newValue) {
                        found = true;
                        break;
                    }
                }

                if (!found) {
                    actionQueue.insert(timedActionInfo);
                }
            }
            if (actionQueue.empty()) {
                return 0;
            }
            actionQueue.erase(actionQueue.begin());
        }
        long nextActionTime = actionQueue.begin()->first;
        bool transferActive = false;
        for (map<long, SimulationAction *>::iterator action =
actionQueue.begin(); action != actionQueue.end();) {
            SimulationAction *simulationAction = action->second;
            if (simulationAction->actionType == NODE_TO_NODE_SEND_ACTION ||
                    simulationAction->actionType == SEND_FROM_NODE_QUEUE_ACTION)
{
                transferActive = true;
            }
            if (nextActionTime > action->first) {
                nextActionTime = action->first;
            }
            action++;
        }
        if (!transferActive) {
            return 0;
        }
        return nextActionTime;
    }

    void revertSendingByBlockedNode(vector<SimulationAction *> newActions, long
startMarkId, DigitalComplexMark *startMark) {
        for (map<long, SimulationAction *>::iterator action =
actionQueue.begin(); action != actionQueue.end();) {
            SimulationAction *simulationAction = action->second;
            if (simulationAction->actionType == NODE_TO_NODE_SEND_ACTION) {
                long simulationStartMarkId = simulationAction->startSmartMarkId;
                if (simulationStartMarkId == startMarkId) {
                    startMark->addMessageToQueue(simulationAction-
>agentPackage);
                    newActions.push_back(new SimulationAction(globalSpentTime +
inputParameterData->getProcessingDataDelay(), SEND_FROM_NODE_QUEUE_ACTION,
                            startMarkId, 0, 0, NULL));
                    actionQueue.erase(action);
                    action = actionQueue.begin();
                    continue;
                } else if (simulationAction->endSmartMarkId == startMarkId) {
                    mappedComplexMarks[simulationStartMarkId]-
>addMessageToQueue(simulationAction->agentPackage);
```

```
                    newActions.push_back(new SimulationAction(globalSpentTime +
inputParameterData->getProcessingDataDelay(), SEND_FROM_NODE_QUEUE_ACTION,
                            simulationStartMarkId, 0, 0, NULL));
                    actionQueue.erase(action);
                    action = actionQueue.begin();
                    continue;
                }
            }
            ++action;
        }
    }

    void revertSendingByArcActions(vector<SimulationAction *> newActions, long
startMarkId, long endSmartMarkId, DigitalComplexMark *startMark) {
        for (map<long, SimulationAction *>::iterator action =
actionQueue.begin(); action != actionQueue.end();) {
            SimulationAction *simulationAction = action->second;
            if (simulationAction->actionType == NODE_TO_NODE_SEND_ACTION) {
                if (simulationAction->startSmartMarkId == startMarkId &&
simulationAction->endSmartMarkId == endSmartMarkId) {
                    startMark->addMessageToQueue(simulationAction-
>agentPackage);
                    newActions.push_back(new SimulationAction(globalSpentTime +
inputParameterData->getProcessingDataDelay(), SEND_FROM_NODE_QUEUE_ACTION,
                            startMarkId, 0, 0, NULL));
                    actionQueue.erase(action);
                    action = actionQueue.begin();
                    continue;
                }
            }
            ++action;
        }
    }

    void writeCurrentActionInfoToLog(SimulationAction *nextAction) {
        stringstream actionMessage;
        long agentId = nextAction->agentPackage == NULL ? 0 : nextAction-
>agentPackage->getPackageIdentificator();
        actionMessage << nextAction->executionTimeFromStart <<
                " -> action=" << nextAction->getActionTypeName()
                << ", agentId = " << agentId
                << ", st_node = " << nextAction->startSmartMarkId <<
                ", end_node = " << nextAction->endSmartMarkId <<
                ", newValue = " << nextAction->newValue;
        logFileWriter->writeToLog(actionMessage.str().c_str());
    }

    vector<SimulationAction *> runActionSendMessageFromQueue(DigitalComplexMark
*digitalComplexMark) {
        long markId = digitalComplexMark->getId();
        vector<long> availableMarkIds =
getAvailableToSendSmartMarkStations(markId);
        vector<SimulationAction *> actions;
        vector<AgentPackageDataContainer *> outputPackages = digitalComplexMark-
>startAllAgentPackagesSending(availableMarkIds, globalSpentTime);
        vector<AgentPackageDataContainer *> sentPackages;
        vector<AgentPackageDataContainer *> leftPackages;
        for (int packageIndex = 0; packageIndex < outputPackages.size();
packageIndex++) {
            AgentPackageDataContainer *currentPackage =
outputPackages[packageIndex];
            long nextMarkId = currentPackage->getNextPossibleNodeId();
            if (nextMarkId != 0) {
                actions.push_back(new SimulationAction(globalSpentTime +
```

```
currentPackage->getNextNodeReachTime(), NODE_TO_NODE_SEND_ACTION,
                        markId, nextMarkId, 0, currentPackage));
            } else {
                digitalComplexMark->addMessageToQueue(currentPackage);
                leftPackages.push_back(currentPackage);
            }
        }
        if (!leftPackages.empty()) {
            actions.push_back(new SimulationAction(globalSpentTime +
inputParameterData->getNextTryDelay(), SEND_FROM_NODE_QUEUE_ACTION,
                    markId, 0, 0, NULL));
        }
        return actions;
    }

    vector<SimulationAction *> runNodeToNodeSendingAction(SimulationAction
*action) {
        vector<SimulationAction *> actions;
        AgentPackageDataContainer *sentPackage = action->agentPackage;
        long previousMarkId = action->startSmartMarkId;
        long targetMarkId = action->endSmartMarkId;
        // update mark values in DSM:
        if (!sentPackage->isVisitedNodeConnection(previousMarkId, targetMarkId))
{
            DigitalComplexMark *previousSmartMark =
mappedComplexMarks[previousMarkId];
            previousSmartMark->updateMarkValueToTargetMarkNet(targetMarkId,
globalSpentTime, inputParameterData->getUpdateMarkValueCoefficient());
        }
        // place agent package to next DSM:
        DigitalComplexMark *targetSmartMark = mappedComplexMarks[targetMarkId];
        sentPackage->placeAtNode(targetMarkId, sentPackage-
>getNextNodeReachTime());
        targetSmartMark->addMessageToQueue(sentPackage);
        // if current node was start, end of route
        if (sentPackage->getStartNodeId() == targetMarkId) {
            return actions;
        }

        // add delay for data processing and sending further
        actions.push_back(new SimulationAction(globalSpentTime +
inputParameterData->getProcessingDataDelay(),
                SEND_FROM_NODE_QUEUE_ACTION, targetMarkId, 0, 0, NULL));
        return actions;
    }

    vector<long> getAvailableToSendSmartMarkStations(long currentSmartMarkIndex)
{
        vector<long> accessibilityData;
        vector<int> availableMarkIndexes = data-
>getAccessibleSmartMarkIndexes(currentSmartMarkIndex);
        for (long index = 0; index < availableMarkIndexes.size(); index++) {
            long smartMarkIndex = availableMarkIndexes[index];
            if (smartMarkIndex == currentSmartMarkIndex) {
                continue;
            }
            if (mappedComplexMarks[smartMarkIndex]->isAccessible() &&
(!mappedComplexMarks[smartMarkIndex]->isTrap()
                    || mappedComplexMarks[smartMarkIndex]->isTrap() &&
mappedComplexMarks[smartMarkIndex]->getId() == inputParameterData-
>getStartMarkId())) {
                accessibilityData.push_back(smartMarkIndex);
            }
        }
```

```
        return accessibilityData;
    }

};

#endif /* ENVIRONMENTSIMULATOR_H_ */
```

Лістинг “InputParameterData.h”:

```
#include <vector>
#include <algorithm>
#include "DigitalComplexMark.h"

#ifndef INPUTPARAMETERDATA_H_
#define INPUTPARAMETERDATA_H_
using namespace std;

class InputParameterData {

    double coefficientA;
    double coefficientB;
    long startMarkId;
    // neighbor list in classical ant colony optimization algorithm
    long decisionPoolSize;
    long agentCount;
    long totalSimulationTime;
    long nextTryDelay;
    long processingDataDelay;
    double bestBoostResultCoefficient;
    double goodBoostResultCoefficient;
    double updateMarkValueCoefficient;
    double minStartDigitalMarkValue;
    double evaporationSpeed;
    double markValueBoost;
    double maxMarkValue;
    double waitTime;
    double coefficientQ;

public:

    InputParameterData(double coefficientA, double coefficientB, double coefficientQ,
long startMarkId, long decisionPoolSize,
            long agentCount, long totalSimulationTime, long nextTryDelay, long
processingDataDelay,
            double bestBoostResultCoefficient, double goodBoostResultCoefficient, double
updateMarkValueCoefficient,
            double minStartDigitalMarkValue, double evaporationSpeed, double
markValueBoost, double maxMarkValue, double waitTime) {
        this->coefficientA = coefficientA;
        this->coefficientB = coefficientB;
        this->coefficientQ = coefficientQ;
        this->startMarkId = startMarkId;
        this->decisionPoolSize = decisionPoolSize;
        this->agentCount = agentCount;
        this->totalSimulationTime = totalSimulationTime;
        this->nextTryDelay = nextTryDelay;
        this->processingDataDelay = processingDataDelay;
        this->bestBoostResultCoefficient = bestBoostResultCoefficient;
        this->goodBoostResultCoefficient = goodBoostResultCoefficient;
        this->updateMarkValueCoefficient = updateMarkValueCoefficient;
        this->minStartDigitalMarkValue = minStartDigitalMarkValue;
        this->evaporationSpeed = evaporationSpeed;
        this->markValueBoost = markValueBoost;
        this->maxMarkValue = maxMarkValue;
        this->waitTime = waitTime;
    }
```

```
    double getCoefficientA() {
        return coefficientA;
    }

    double getCoefficientB() {
        return coefficientB;
    }

    double getCoefficientQ() {
        return coefficientQ;
    }

    long getStartMarkId() {
        return startMarkId;
    }

    long getDecisionPoolSize() {
        return decisionPoolSize;
    }

    long getAgentCount() {
        return agentCount;
    }

    long getTotalSimulationTime() {
        return totalSimulationTime;
    }

    long getNextTryDelay() {
        return nextTryDelay;
    }

    long getProcessingDataDelay() {
        return processingDataDelay;
    }

    double getBestBoostResultCoefficient() {
        return bestBoostResultCoefficient;
    }

    double getGoodBoostResultCoefficient() {
        return goodBoostResultCoefficient;
    }

    double getUpdateMarkValueCoefficient() {
        return updateMarkValueCoefficient;
    }

    double getMinStartDigitalMarkValue() {
        return minStartDigitalMarkValue;
    }

    double getEvaporationSpeed() {
        return evaporationSpeed;
    }

    double getMarkValueBoost() {
        return markValueBoost;
    }

    double getMaxMarkValue() {
        return maxMarkValue;
    }

    double getWaitTime() {
        return waitTime;
    }

};

#endif /* INPUTPARAMETERDATA_H_ */
```

Лістинг “AgentPackageContainer.h”:

```
#include <bits/stl_vector.h>

#ifndef AGENTPACKAGEDATACONTAINER_H_
#define AGENTPACKAGEDATACONTAINER_H_

using namespace std;

struct VisitedNodeInfo {
    long nodeId;
    long lastVisitTime;

    bool operator<(const VisitedNodeInfo &nodeInfo) const {
        return (lastVisitTime < nodeInfo.lastVisitTime);
    }

};

struct NodeConnection {
    long firstNodeId;
    long secondNodeId;
};

class AgentPackageDataContainer {

private:
    // agent id
    long packageIdentificator;
    // visited nodes memory will be calculated as data.
    // visitedNodeData - timed visited nodes for logic of cycles overcoming
    vector<VisitedNodeInfo> visitedNodeData;
    vector<long> visitedNodeIds;
    vector<long> routeNodeIds;
    long totalTravelTime;
    int targetIndex;
    int sourceIndex;
    int maxNodeLimit;
    char *dataToSend;
    long nextPossibleNodeId;
    long nextNodeReachTime;


public:
    AgentPackageDataContainer(long packageIdentificator, int targetIndex, int
sourceIndex, int maxNodeLimit, char *dataToSend) {
        this->packageIdentificator = packageIdentificator;
        this->targetIndex = targetIndex;
        this->sourceIndex = sourceIndex;
        this->maxNodeLimit = maxNodeLimit;
        this->dataToSend = dataToSend;
        totalTravelTime = 0;
        nextPossibleNodeId = 0;
        nextNodeReachTime = 0;
    }

    bool reverseCompareVisitedNodeInfo(VisitedNodeInfo n1, VisitedNodeInfo n2) {
        return (n1.lastVisitTime > n2.lastVisitTime);
    }

    vector<long> getSortedByVisitNodeIds() {
        std::sort(visitedNodeData.begin(), visitedNodeData.end());
        vector<long> sortedNodeIds;
        for (int index = 0; index < visitedNodeData.size(); index++) {
            sortedNodeIds.push_back(visitedNodeData[index].nodeId);
        }
        return sortedNodeIds;
    }

    bool isVisitedNodeConnection(long previousNodeId, long currentNodeId) {
        for (int index = 0; index < routeNodeIds.size() - 1; index++) {
```

```
            if (routeNodeIds[index] == previousNodeId) {
                if (routeNodeIds[index + 1] == currentNodeId) {
                    return true;
                }
            }
        }
        return false;
    }

    void placeAtNode(long nodeId, long wastedTimeOnSend) {
        if (!containValue(visitedNodeIds.begin(), visitedNodeIds.end(), nodeId)) {
            visitedNodeIds.push_back(nodeId);
            VisitedNodeInfo nodeInfo = {nodeId, totalTravelTime + wastedTimeOnSend};
            visitedNodeData.push_back(nodeInfo);
        } else {
            for (int index = 0; index < visitedNodeData.size(); index++) {
                if (visitedNodeData[index].nodeId == nodeId) {
                    visitedNodeData[index].lastVisitTime == totalTravelTime +
wastedTimeOnSend;
                }
            }
        }
        routeNodeIds.push_back(nodeId);
        totalTravelTime += wastedTimeOnSend;
    }

    bool wasVisitedNode(long nodeId) {
        return containValue(visitedNodeIds.begin(), visitedNodeIds.end(), nodeId);
    }

    vector<long> getRouteInfo() {
        return routeNodeIds;
    }

    vector<NodeConnection> getUniqueNodeConnections() {
        vector<NodeConnection> nodeConnections;
        for (int index = 0; index < routeNodeIds.size() - 1; index++) {
            NodeConnection newConnection = {routeNodeIds[index], routeNodeIds[index +
1]};
            bool found = false;
            for (int i = 0; i < nodeConnections.size(); i++) {
                NodeConnection existConnection = nodeConnections[i];
                if (existConnection.firstNodeId == newConnection.firstNodeId &&
existConnection.secondNodeId == newConnection.secondNodeId) {
                    found = true;
                }
            }
            if (!found) {
                nodeConnections.push_back(newConnection);
            }
        };
        return nodeConnections;
    }

    long getStartNodeId() {
        if (visitedNodeIds.size() > 0) {
            return visitedNodeIds[0];
        }
        return 0;
    }

    long getPreviousNodeId() {
        if (routeNodeIds.size() > 1) {
            return routeNodeIds[routeNodeIds.size() - 2];
        } else {
            return currentNodeId();
        }
    }

    long getDataSize() {
        // + 4 cause of additional fields data.
```

```
        return sizeof(*dataToSend) + routeNodeIds.size() + 4;
    }

    bool isRouteFinished() {
        return getStartNodeId() == currentNodeId() && totalTravelTime != 0;
    }

    long currentNodeId() {
        if (routeNodeIds.size() > 0) {
            return routeNodeIds[routeNodeIds.size() - 1];
        }
        return 0;
    }


    long getNextPossibleNodeId() {
        return nextPossibleNodeId;
    }


    void setNextPossibleNodeId(long nextPossibleNodeId) {
        this->nextPossibleNodeId = nextPossibleNodeId;
    }

    long getNextNodeReachTime() {
        return nextNodeReachTime;
    }

    void setNextNodeReachTime(long nextNodeReachTime) {
        this->nextNodeReachTime = nextNodeReachTime;
    }

    long getTotalTravelTime() {
        return totalTravelTime;
    }

    long getPackageIdentificator() {
        return packageIdentificator;
    }

private :

    bool containValue(vector<long>::iterator startIterator, vector<long>::iterator
endIterator, long value) {
        for (; startIterator != endIterator; ++startIterator) {
            if (*startIterator == value) {
                return true;
            }
        }
        return false;
    }

};

#endif /* AGENTPACKAGEDATACONTAINER_H_ */
```

Лістинг “SimulationAction.h”:

```
#include <vector>
#include <algorithm>
#include "DigitalComplexMark.h"

#ifndef SIMULATIONACTION_H_
#define SIMULATIONACTION_H_
using namespace std;

enum SimulationActionType {
    NODE_TO_NODE_SEND_ACTION,
```

```
    SEND_FROM_NODE_QUEUE_ACTION,
    SWITCH_OFF_NODE,
    SWITCH_ON_NODE,
    SWITCH_OFF_ARC,
    SWITCH_ON_ARC,
    CHANGE_NODE_TO_NODE_TIME
};

class SimulationAction {

public:
    long executionTimeFromStart;
    SimulationActionType actionType;
    long startSmartMarkId;
    long endSmartMarkId;
    long newValue;
    AgentPackageDataContainer *agentPackage;

    SimulationAction(long executionTimeFromStart, SimulationActionType actionType, long
startSmartMarkId, long endSmartMarkId,
            long newValue, AgentPackageDataContainer *agentPackage) {
        this->executionTimeFromStart = executionTimeFromStart;
        this->actionType = actionType;
        this->startSmartMarkId = startSmartMarkId;
        this->endSmartMarkId = endSmartMarkId;
        this->newValue = newValue;
        this->agentPackage = agentPackage;
    }

    string getActionTypeName() {
        switch(actionType) {
            case NODE_TO_NODE_SEND_ACTION:
                return "Node to node send finish";
            case SEND_FROM_NODE_QUEUE_ACTION:
                return "Send all messages of node";
            case SWITCH_OFF_NODE:
                return "Switch off node";
            case SWITCH_OFF_ARC:
                return "Switch off arc";
            case CHANGE_NODE_TO_NODE_TIME:
                return "Change node to node value";
        }
    }

};

bool simulationActionTypeComparator(SimulationAction action1, SimulationAction action2) {
    return (action1.executionTimeFromStart < action2.executionTimeFromStart);
}

void sortSimulationActionsByStartTime(vector<SimulationAction> actions) {
    sort(actions.begin(), actions.end(), simulationActionTypeComparator);
}

#endif /* SIMULATIONACTION_H_ */
```

Лістинг “DigitalComplexMark.h”:

```
#ifndef DIGITALCOMPLEXMARK_H_
#define DIGITALCOMPLEXMARK_H_

#include <queue>
#include <map>
#include <algorithm>

#include "AgentPackageDataContainer.h"
#include "EnvironmentData.h"
#include "InputParameterData.h"
```

```
#include "SmartMarkUtils.h"

const long MIN_VALUE_COST = 1;

class NextNodeCostInfo {

public:
    long nextMarkId;
    double cost;

    NextNodeCostInfo(long nextMarkId, double cost) {
        this->nextMarkId = nextMarkId;
        this->cost = cost;
    }

};

class MarkTimedValueInfo {

public:
    long lastTimeChanged;
    double markValue;

    MarkTimedValueInfo(long lastTimeChanged, double markValue) {
        this->lastTimeChanged = lastTimeChanged;
        this->markValue = markValue;
    }

    void updateTime(long lastTimeChanged) {
        this->lastTimeChanged = lastTimeChanged;
    }

    void updateMarkValue(double markValue) {
        this->markValue = markValue;
    }

};

void reverseSortNextActionsByStartTime(vector<NextNodeCostInfo *> costs);

class DigitalComplexMark {

private:
    // id \ index of current smart mark system
    long markId;
    // station is disabled
    bool isOffline;
    // can receive but can't send anything
    bool isClosed;
    // flag of trap situation on current figital mark
    bool trap;

    long randomSeedValue;

    // DCRM of this smart mark system
    map<long, MarkTimedValueInfo *> digitalConnectionRatioMemory;

    // agent messages queue
    queue<AgentPackageDataContainer *> messageQueue;


    // link to environment, which include total data:
    EnvironmentData *environment;
    InputParameterData *parameters;

public:
    DigitalComplexMark(long markId, EnvironmentData *environment, InputParameterData
*parameters) {
        this->markId = markId;
        this->environment = environment;
        this->parameters = parameters;
```

```
        isOffline = false;
        isClosed = false;
        trap = false;
        for (long markIndex = 1; markIndex <= environment->getSystemCount(); markIndex++)
{
            digitalConnectionRatioMemory.insert(pair<long, MarkTimedValueInfo
*>(markIndex, new MarkTimedValueInfo(0,

parameters->getMinStartDigitalMarkValue())));
        }
        randomSeedValue = markId;
    }

    vector<AgentPackageDataContainer *> retrieveAllMessagesAndResetQueue() {
        vector<AgentPackageDataContainer *> allAgentPackages, leftAgentPackages;
        while (!messageQueue.empty()) {
            AgentPackageDataContainer *agent = messageQueue.front();
            allAgentPackages.push_back(agent);
            if (agent->currentNodeId() != agent->getStartNodeId()) {
                leftAgentPackages.push_back(agent);
            }
            messageQueue.pop();
        }
        for (int index = 0; index < leftAgentPackages.size(); index++) {
            messageQueue.push(leftAgentPackages[index]);
        }
        return allAgentPackages;
    }

    void addMessageToQueue(AgentPackageDataContainer *message) {
        messageQueue.push(message);
    }

    vector<AgentPackageDataContainer *> startAllAgentPackagesSending(vector<long>
availableMarkIds, long currentTime) {
        vector<AgentPackageDataContainer *> packages;
        vector<long> wideTimeDataValues = environment->getAbstractTransferTimeData(markId
- 1);
        if (availableMarkIds.empty()) {
            // Trap situation appears
            trap = true;
        }
        while (!messageQueue.empty()) {
            AgentPackageDataContainer *agent = messageQueue.front();
            long nextMarkId = findNextMarkId(agent, wideTimeDataValues, availableMarkIds,
currentTime);
            if (nextMarkId != 0) {
                agent->setNextNodeReachTime(wideTimeDataValues[nextMarkId - 1]);
                trap = false;
            }
            agent->setNextPossibleNodeId(nextMarkId);
            packages.push_back(agent);
            messageQueue.pop();
        }
        return packages;
    }

    void updateMarkValueToTargetMarkNet(long targetMarkId, long currentTime, double
inputCoefficient) {
        MarkTimedValueInfo *timedMarkValue = digitalConnectionRatioMemory[targetMarkId];
        double newMarkValue = 0;
        double timedValue = timedMarkValue->markValue;
        if (timedMarkValue->lastTimeChanged < currentTime) {
            long duration = currentTime - timedMarkValue->lastTimeChanged;
            newMarkValue += max(timedValue - parameters->getEvaporationSpeed() *
duration,
                                parameters->getMinStartDigitalMarkValue());
        } else {
            newMarkValue += timedValue;
        }
        newMarkValue += parameters->getMarkValueBoost() * inputCoefficient;
```

```
//        cout << "CurId =" << markId << ", targetId=" << targetMarkId << ", markValue="
<< timedValue << ", newValue=" << newMarkValue << endl;
        newMarkValue = min(newMarkValue, parameters->getMaxMarkValue());
        timedMarkValue->updateTime(currentTime);
        timedMarkValue->updateMarkValue(newMarkValue);
    }

    long findNextMarkId(AgentPackageDataContainer *agentPackageDataContainer,
vector<long> wideTimeDataValues,
                        vector<long> availableMarkIds, long currentTime) {
        vector<NextNodeCostInfo *> possibilitiesPool;
        long startNodeId = agentPackageDataContainer->getStartNodeId();
        bool availableReturnToEndRoute = false;
        vector<NextNodeCostInfo *> visitedPossibilitiesPool;
        vector<long> visitedNodeIds;
        // get max value from cost for reverse value set the.
        // Less travel time - more possible to choose next node;
        /*
        long maxTimeValue = 0;
        for (int availableMarkId = 0; availableMarkId < availableMarkIds.size();
availableMarkId++) {
            long currentMarkId = availableMarkIds[availableMarkId];
            if (wideTimeDataValues[currentMarkId - 1] > maxTimeValue) {
                // to avoid 0 chance value:
                maxTimeValue = wideTimeDataValues[currentMarkId - 1] + 1;
            }
        }
        */
        for (int availableMarkId = 0; availableMarkId < availableMarkIds.size();
availableMarkId++) {
            long currentMarkId = availableMarkIds[availableMarkId];
            MarkTimedValueInfo *timedMarkValue =
digitalConnectionRatioMemory[currentMarkId];
            if (timedMarkValue->lastTimeChanged < currentTime) {
                long duration = currentTime - timedMarkValue->lastTimeChanged;
                double newMarkValue = max(timedMarkValue->markValue - parameters-
>getEvaporationSpeed() * duration,
                                          parameters->getMinStartDigitalMarkValue());
                timedMarkValue->updateTime(currentTime);
                timedMarkValue->updateMarkValue(newMarkValue);
            }
            double sendCostValue = powValue(parameters->getCoefficientQ() /
wideTimeDataValues[currentMarkId - 1], (int) parameters->getCoefficientA()) *
                    powValue(timedMarkValue->markValue, (int) parameters-
>getCoefficientB());
            NextNodeCostInfo *info = new NextNodeCostInfo(currentMarkId, sendCostValue);
            if (agentPackageDataContainer->wasVisitedNode(currentMarkId)) {
                if (currentMarkId == startNodeId) {
                    availableReturnToEndRoute = true;
                }
                visitedPossibilitiesPool.push_back(info);
                visitedNodeIds.push_back(currentMarkId);
                continue;
            }
            possibilitiesPool.push_back(info);
        }
        if (possibilitiesPool.size() > 0) {
            return getNextSmartMarkIdFromPossiblePool(possibilitiesPool);
        } else if (availableReturnToEndRoute) {
            return startNodeId;
        } else if (visitedPossibilitiesPool.size() > 0) {
            if (visitedPossibilitiesPool.size() > 1) {
                for (int index = 0; index < visitedPossibilitiesPool.size(); index++) {
                    if (visitedPossibilitiesPool[index]->nextMarkId ==
agentPackageDataContainer->getPreviousNodeId()) {
                        visitedPossibilitiesPool[index]->cost = MIN_VALUE_COST;
                    }
                }
            }
            // currently go just by choose possibility, that is wrong.
            // 1) Check by new connection that will appear if it haven't be passed yet;
```

```
If such exist - use one of them by possibility; Use start node in last order.
            vector<NextNodeCostInfo *> notVisitedConnectionsPool;
            for (int nodeIndex = 0; nodeIndex < visitedPossibilitiesPool.size();
nodeIndex++) {
                if (agentPackageDataContainer->isVisitedNodeConnection(startNodeId,

visitedPossibilitiesPool[nodeIndex]->nextMarkId)) {

notVisitedConnectionsPool.push_back(visitedPossibilitiesPool[nodeIndex]);
                }
            }
            if (!notVisitedConnectionsPool.empty()) {
                return getNextSmartMarkIdFromPossiblePool(notVisitedConnectionsPool);
            }
            // 2) All connections were used. Search by visited nodes order, choose first
visited one with aim to get out from cycle and get closer to start node.
            vector<long> sortedByTimeNodeIds = agentPackageDataContainer-
>getSortedByVisitNodeIds();
            for (int index = 0; index < sortedByTimeNodeIds.size(); index++) {
                long selectedNodeId = sortedByTimeNodeIds[index];
                std::vector<long>::iterator found = find(visitedNodeIds.begin(),
visitedNodeIds.end(), selectedNodeId);
                if (found != visitedNodeIds.end()) {
                    return selectedNodeId;
                }
            }
            // 3) Nothing to do just go not into previous, continue to search. Selection
by possibility.
            return getNextSmartMarkIdFromPossiblePool(visitedPossibilitiesPool);
        } else {
            // return not valid Id to generate check again after some time. Current node
is blocked;
            // will set situation of 'TRAP' by flag and stay in node for change
situation.
            return 0;
        }
    }

    double powValue(double value, int step) {
        double result = 1;
        for (int i = 0; i < step; i++) {
            result = result * value;
        }
        return result;
    }

    bool isAccessible() {
        return !isOffline && !isClosed;
    }

    bool isTrap() {
        return trap;
    }

    void switchOffSystem() {
        isOffline = true;
        isClosed = true;
    }

    void switchOnSystem() {
        isOffline = false;
        isClosed = false;
    }

    long getId() {
        return this->markId;
    }

    vector<double> getMemoryMarkValuesRow() {
        vector<double> markValues;
        for (map<long, MarkTimedValueInfo *>::iterator mark =
```

```
digitalConnectionRatioMemory.begin();
             mark != digitalConnectionRatioMemory.end(); ++mark) {
            markValues.push_back(mark->second->markValue);
        }
        return markValues;
    }

private:

    long getNextSmartMarkIdFromPossiblePool(vector<NextNodeCostInfo *> possibilitiesPool)
{
        // sort by cost to get only decision pool size variants:
        double totalPossibilitySum = 0;
        // more cost - more possibility for selection
        reverseSortNextActionsByStartTime(possibilitiesPool);
        // calculate total cost sum:
        for (int index = 0; index < min((long) possibilitiesPool.size(), parameters-
>getDecisionPoolSize()); index++) {
            totalPossibilitySum += possibilitiesPool[index]->cost;
        }
        double randomValue = randomGeneratorOfParkAndMiller(&randomSeedValue) *
totalPossibilitySum;
        for (int index = 0; index < min((long) possibilitiesPool.size(), parameters-
>getDecisionPoolSize()); index++) {
            double currentCostValue = possibilitiesPool[index]->cost;
            if (randomValue < currentCostValue) {
                return possibilitiesPool[index]->nextMarkId;
            }
            randomValue -= currentCostValue;
        }
        return 0;
    }

};

bool reverseNextNodeInfoComparator(NextNodeCostInfo *cost1, NextNodeCostInfo *cost2) {
    return (cost1->cost > cost2->cost);
}

void reverseSortNextActionsByStartTime(vector<NextNodeCostInfo *> costs) {
    sort(costs.begin(), costs.end(), reverseNextNodeInfoComparator);
}

#endif /* DIGITALCOMPLEXMARK_H_ */
```

Додаток І

Результати досліджень, розв'язки ЗК

Вмістиме файлу «trap_task.res»:

CYCLE 1 ROUTES:

Route of [5] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [1] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [4] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [3] agent : 1 6 4 6 2 6 5 6 3 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [2] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [6] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

CYCLE 2 ROUTES:

Route of [12] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [10] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [7] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [8] agent : 1 6 4 6 2 6 5 6 3 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [9] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [1] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [4] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [11] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [5] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [3] agent : 1 6 4 6 2 6 5 6 3 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [2] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [6] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Вмістиме файлу «trap_task2.res»:

CYCLE 1 ROUTES:

Route of [1] agent : 1 6 2 6 4 6 3 6 5 6 7 9 7 8 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [4] agent : 1 6 3 6 4 6 2 6 5 6 7 9 7 8 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [5] agent : 1 6 2 6 4 6 5 6 3 6 7 8 7 9 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 6 4 6 2 6 5 6 3 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [2] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [6] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

CYCLE 2 ROUTES:

Route of [12] agent : 1 6 2 6 4 6 5 6 3 6 7 9 7 8 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [7] agent : 1 6 2 6 5 6 3 6 4 6 7 9 7 8 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [8] agent : 1 6 4 6 5 6 3 6 2 6 7 9 7 8 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [10] agent : 1 6 4 6 2 6 3 6 5 6 7 8 7 9 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [11] agent : 1 6 4 6 5 6 3 6 2 6 7 8 7 9 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [9] agent : 1 6 3 6 4 6 5 6 2 6 7 8 7 9 7 6 1 || SPENT TIME = 5158 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 6 4 6 2 6 5 6 3 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [2] agent : 1 6 4 6 3 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Route of [6] agent : 1 6 3 6 4 6 2 6 5 6 7 10 || SPENT TIME = 3387 currently at MarkId = 10 STATUS = UNFINISHED

Результати розв'язання "mesh _task" ЗК, вмістиме файлу "mesh _task.res":

CYCLE 1 ROUTES:

Route of [6] agent : 1 2 3 8 5 4 6 7 1 || SPENT TIME = 690 currently at MarkId = 1 STATUS = FINISHED

Route of [4] agent : 1 10 3 4 5 6 7 8 9 2 1 || SPENT TIME = 680 currently at MarkId = 1 STATUS = FINISHED

Route of [2] agent : 1 2 3 5 4 6 7 8 9 10 1 || SPENT TIME = 720 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 2 9 8 3 5 4 6 7 1 || SPENT TIME = 840 currently at MarkId = 1 STATUS = FINISHED

Route of [5] agent : 1 7 6 4 5 8 3 10 9 2 1 || SPENT TIME = 910 currently at MarkId = 1 STATUS = FINISHED

Route of [1] agent : 1 2 3 4 5 8 7 6 4 3 10 9 2 1 || SPENT TIME = 1030 currently at MarkId = 1 STATUS = FINISHED

CYCLE 2 ROUTES:

Route of [7] agent : 1 2 3 8 9 10 1 || SPENT TIME = 410 currently at MarkId = 1 STATUS = FINISHED

Route of [11] agent : 1 2 3 8 5 6 7 1 || SPENT TIME = 550 currently at MarkId = 1 STATUS = FINISHED

Route of [12] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [9] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [8] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [10] agent : 1 10 3 4 5 6 7 8 9 2 1 || SPENT TIME = 680 currently at MarkId = 1 STATUS = FINISHED

Вмістиме файлу "mesh _task2.res":

CYCLE 1 ROUTES:

Route of [2] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [4] agent : 1 10 3 4 5 6 7 8 9 2 1 || SPENT TIME = 680 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 2 9 8 3 4 5 6 7 1 || SPENT TIME = 760 currently at MarkId = 1 STATUS = FINISHED

Route of [5] agent : 1 7 6 4 5 8 3 10 9 2 1 || SPENT TIME = 910 currently at MarkId = 1 STATUS = FINISHED

Route of [6] agent : 1 2 3 5 6 4 3 10 9 8 7 1 || SPENT TIME = 980 currently at MarkId = 1 STATUS = FINISHED

Route of [1] agent : 1 2 3 4 5 8 7 6 4 3 10 9 2 1 || SPENT TIME = 1030 currently at MarkId = 1 STATUS = FINISHED

CYCLE 2 ROUTES:

Route of [7] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 100 currently at MarkId = 1 STATUS = FINISHED

Route of [8] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 100 currently at MarkId = 1 STATUS = FINISHED

Route of [12] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [9] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [10] agent : 1 10 9 2 3 4 5 8 7 6 4 3 10 1 || SPENT TIME = 630 currently at MarkId = 1 STATUS = FINISHED

Route of [11] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

CYCLE 3 ROUTES:

Route of [13] agent : 1 10 9 8 7 6 4 3 2 1 || SPENT TIME = 160 currently at MarkId = 1 STATUS = FINISHED

Route of [14] agent : 1 10 9 8 7 6 4 3 2 1 || SPENT TIME = 160 currently at MarkId = 1 STATUS = FINISHED

Route of [15] agent : 1 10 9 8 7 6 4 3 2 1 || SPENT TIME = 160 currently at MarkId = 1 STATUS = FINISHED

Route of [16] agent : 1 10 9 8 7 6 4 3 2 1 || SPENT TIME = 160 currently at MarkId = 1 STATUS = FINISHED

Route of [17] agent : 1 10 9 8 7 6 4 3 2 1 || SPENT TIME = 160 currently at MarkId = 1 STATUS = FINISHED

Route of [18] agent : 1 2 3 4 6 7 8 9 10 1 || SPENT TIME = 560 currently at MarkId = 1 STATUS = FINISHED

Вмістиме файлу “full_top_task.res”:

CYCLE 1 ROUTES:

Route of [1] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [2] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

Route of [3] agent : 1 2 5 3 4 6 9 8 7 10 1 || SPENT TIME = 750 currently at MarkId = 1 STATUS = FINISHED

Route of [6] agent : 1 2 3 5 4 6 8 7 10 9 1 || SPENT TIME = 810 currently at MarkId = 1 STATUS = FINISHED

Route of [4] agent : 1 4 3 5 7 8 2 9 6 10 1 || SPENT TIME = 850 currently at MarkId = 1 STATUS = FINISHED

Route of [5] agent : 1 3 5 2 4 6 7 10 8 9 1 || SPENT TIME = 860 currently at MarkId = 1 STATUS = FINISHED

CYCLE 2 ROUTES:

Route of [9] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [11] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [12] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [7] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

Route of [8] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

Route of [10] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

CYCLE 3 ROUTES:

Route of [13] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [14] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [17] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [18] agent : 1 2 3 4 6 5 7 10 8 9 1 || SPENT TIME = 640 currently at MarkId = 1 STATUS = FINISHED

Route of [15] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

Route of [16] agent : 1 2 3 5 7 4 6 10 8 9 1 || SPENT TIME = 670 currently at MarkId = 1 STATUS = FINISHED

Вмістиме файлу “ cycle_top_task.res”:

CYCLE 1 ROUTES:

Route of [4] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [5] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [1] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

Route of [2] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

Route of [3] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

Route of [6] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

CYCLE 2 ROUTES:

Route of [7] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [8] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [9] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [10] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [11] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

Route of [12] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED

CYCLE 3 ROUTES:

Route of [13] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [14] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [15] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [16] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [17] agent : 1 10 9 8 7 6 5 4 3 2 1 || SPENT TIME = 879 currently at MarkId = 1 S = FINISHED

Route of [18] agent : 1 2 3 4 5 6 7 8 9 10 1 || SPENT TIME = 1483 currently at MarkId = 1 S = FINISHED