МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Ніколюк П. К.

МОДЕЛЮВАННЯ СИСТЕМ

Навчальний посібник

Вінниця
ДонНУ імені Василя Стуса
2023

УДК 004.94 (075.8)
Н18

Затверджено Вченою радою Донецького національного університету імені Василя Стуса (протокол № 6 від 22.12.2023 р.)

Автор:
Ніколюк П. К., професор кафедри інформаційних технологій ДонНУ імені Василя Стуса.

Рецензенти:
Лужецький В. А., д-р техн. наук, професор, завідувач кафедри захисту інформації Вінницького національного технічного університету.
Ковтун В. В., д-р техн. наук, професор, завідувач кафедри комп'ютерних систем управління Вінницького національного технічного університету.
Матвійчук В. А. д-р техн. наук, професор, завідувач кафедри електроенергетики, електротехніки та електромеханіки Вінницького національного аграрного університету.

Ніколюк П. К.
Н 18 Моделювання систем: навчальний посібник для здобувачів вищої освіти спеціальності 122 Комп'ютерні науки. Вінниця: ДонНУ, 2023. 228 с.

Навчальний посібник призначений для здобувачів вищої освіти спеціальності 122 Комп'ютерні науки.

У посібнику розглянуті основні принципи моделювання систем різної природи та проілюстрована їх актуальність. Здобувачі мають можливість застосовувати модельний підхід під час проєктування складних систем ІТ-напряму.

Увага приділена різним видам моделювання – математичному, комп'ютерному та імітаційному. Розглянуті також мережі масового обслуговування та графи як засоби моделювання мереж різної природи.

Мета посібника – дати змогу читачеві застосовувати методологію та інструментарій моделювання як у теоретичних дослідженнях, так і для практичної діяльності. З цією метою розглянуті, зокрема, клітинні автомати, програмні моделі (патерни) та уніфікована мова моделювання Unified Modeling Language, яка являє собою збірку найкращих інженерних практик, що виявилися успішними під час моделювання великих і складних систем та є важливою частиною розробки моделей об'єктно-орієнтованого програмного забезпечення. Розглянуті особливості програми візуального імітаційного моделювання AnyLogic North America LLC, мережі Петрі та модель керування запасами.

Оскільки посібник рекомендований для здобувачів вищої освіти спеціальності 122 Комп'ютерні науки, то у ньому наведено низку комп'ютерних програм, що дає змогу здобувачам скопіювати лістинг програми та інсталювати у програмне поле мови програмування Java. Після цього програма запускається та може використовуватись як базова модель досліджуваної проблеми.

УДК 004.94 (075.8)

ISBN

ЗМІСТ

ВСТУП

Моделювання систем – особливий і дуже дієвий метод досліджень систем різної природи. За такого підходу предмет дослідження являє собою на першому етапі модель системи, яка досліджується, а не саму систему. Дослідник замість досліджуваного об'єкта (системи) вивчає створений ним образ (модель) у матеріальній чи реальній формі. Отже, дослідження проводяться не на самій системі, а на її відображенні (образі). Жодних змін сама досліджувана система поки що не зазнає. Навпаки, модель цієї системи, створена дослідником, піддається різним впливам доти поки не буде віднайдений спосіб покращення функціонування власне самої системи. Після проведення серії експериментів над прототипом системи, тобто моделлю, отримані нові дані переносяться на об'єкт-оригінал. Що це дає? По-перше, економляться колосальні ресурси, адже для внесення змін у систему зазвичай потрібно вкладати величезні кошти. І може виявитися так, що така пряма модернізація системи не тільки не покращить функціональні можливості системи, а, навпаки, погіршить. А кошти вже витрачені безповоротно! З метою уникнення вищеозначених колізій якраз і потрібно проводити спочатку експерименти над моделлю. Лише переконавшись у ефективності модельних експериментів, перевірених багаторазово, можна інсталювати їх у систему з метою покращення її функціональних можливостей. По-друге, правильно проведені модельні експерименти після застосування їх результатів на системі гарантують зростання ефективності системи.

Отже, між дослідником та системою, що вивчається, стоїть деяка проміжна ланка – модель. Дуже важливо щоб модель відображала характерні особливості системи та не привела до хибних висновків. Лише в цьому випадку результати моделювання дадуть позитивні результати під час використання їх у роботі об'єкта (системи). Із сказаного випливає, що до процедури моделювання систем треба ставитись дуже відповідально: якщо модель не відповідає необхідним вимогам, то можна отримати хибні висновки і результати. Застосувавши їх до системи, отримаємо не покращення, а погіршення результатів її роботи.

Мета посібника – представити інструмент, що надає унікальні можливості дослідження систем різної природи, використовуючи, зокрема, методи розробки програмного забезпечення та уніфікованої мови моделювання Unified Modeling Language (UML). UML важлива для здобувачів спеціальності 122 Комп'ютерні технології та являє собою збірку найкращих інженерних практик, які виявилися успішними в моделюванні великих і складних систем та є дуже важливою частиною розробки об'єктно-орієнтованого програмного забезпечення.

РОЗДІЛ 1
СИСТЕМИ, МОДЕЛІ, МОДЕЛЮВАННЯ

1.1. Модель і система

Модель можна визначити як певний спосіб відображення реальних явищ, процесів або об'єктів у деякій абстрактній чи фізичній формі, придатній для наукового дослідження. Отже, моделювання є специфічним способом дослідження реальних природніх процесів. Зауважимо також, що процес моделювання передбачає процес відображення характерних особливостей системи у вигляді певного образу. Зрозуміло, що модель є абстракцією системи. Ступінь цієї абстракції може бути різним. Завдання моделювання визначає дослідник, застосовуючи потрібний ступінь деталізації моделі.

Система – це цілісний об'єкт, що складається із деяких об'єднаних між собою частин, що функціонують як єдине ціле. Системою може бути планета Земля, атом, виробниче підприємство, комп'ютер тощо. Для дослідника система виглядає як складник іншої більш складної системи. Системи пов'язані між собою. Тому необхідно виокремити якийсь компонент складного системного комплексу та досліджувати його з допомогою модельного підходу.

Дуже важливо усвідомлювати взаємодію між моделлю та системою, яка моделюється. Модель і система в певному сенсі знаходяться у взаємодії, яку називають ізоморфізмом. Ізоморфізм – взаємнооднозначне відображення (бієкція) елементів однієї структури на іншу. Отже, між моделлю та системою потрібно встановити ізоморфне відображення. Система та модель можуть бути ізоморфними лише у разі спрощення системи. Такий спрощений варіант системи власне і є моделлю.

Отже, система – це реальна сутність, яку ми досліджуємо. Проілюструємо на конкретному прикладі взаємодію між моделлю та системою. Розглянемо цунамі – гігантські морські хвилі, що народжуються у Світовому океані внаслідок землетрусів чи виверження вулканів та поширюються зі швидкістю кількасот кілометрів на годину і можуть завдати колосальних руйнувань, дійшовши до берега. Саме такий апокаліпсис стався 13 березня 2011 року на атомній електростанції «Фукусіма-1» в Японії у місті Окума (префектура Фукусіма). Внаслідок цього хвиля цунамі зруйнувала атомну електростанцію. У трьох із чотирьох реакторів станції почали плавитися активні зони, що призвело до другої за наслідками ядерної аварії в історії (перша така аварія сталася у Чорнобилі в 1986 році). Поставимо запитання: чи можна було уникнути

такої ядерної аварії? Уявіть, що ви – головний конструктор (архітектор, проєктант) атомної електростанції «Фукусіма-1» у Японії, де спостерігається сильна сейсмічна активність, і найчастіше небезпека приходить з моря у вигляді гігантських морських хвиль – цунамі. Перед вами стоїть завдання так спроєктувати атомну електростанцію, щоб у разі будь-яких катаклізм реактори АЕС витримали вплив зовнішніх потужних сейсмічних та морських хвиль. Перше і найголовніше у цьому випадку – створити модель, яка відтворює ситуацію з цунамі. Результати моделювання потрібно проводити доти, поки модельні об'єкти за будь-яких зовнішніх впливів типу штучних сейсмічних та морських хвиль не приведуть до виведення з ладу модельних функціональних систем (реакторів) станції-моделі та не вплинуть на робочий процес її функціонування. Отже, процес моделювання в цьому випадку потрібно проводити доти, поки станція-модель не буде забезпечена таким рівнем стійкості, що витримає найвищий рівень магнітуди (відоме значення магнітуди становить величину 9,5). Провести такий модельний експеримент, імітуючи стійкість станції-моделі до штучно створених сейсмічних та морських хвиль, не становить великої проблеми. У цьому випадку потрібно досягнути високої динамічної стійкості моделі, а інші її характеристики несуттєві – для них будуть створюватись інші моделі. Річ утім, що система, яку ми моделюємо, зазвичай дуже складна і моделювати всі її особливості надзвичайно складно і практично неможливо. Та, власне кажучи, цього робити і не потрібно. Перед дослідником системи стоїть завдання створити модель, яка вирішує конкретні унітарні завдання. Отже, дуже важливо не відійти від принципу ізоморфізму – модель повинна строго відображати модельовану характеристику системи, і в цьому її цінність. Якщо ця умова не виконується, то модель може бути шкідливою та навіть небезпечною.

1.2. Види моделювання

Умовно, схематично і в загальному вигляді моделювання можна класифікувати за такою схемою (рис. 1.1).

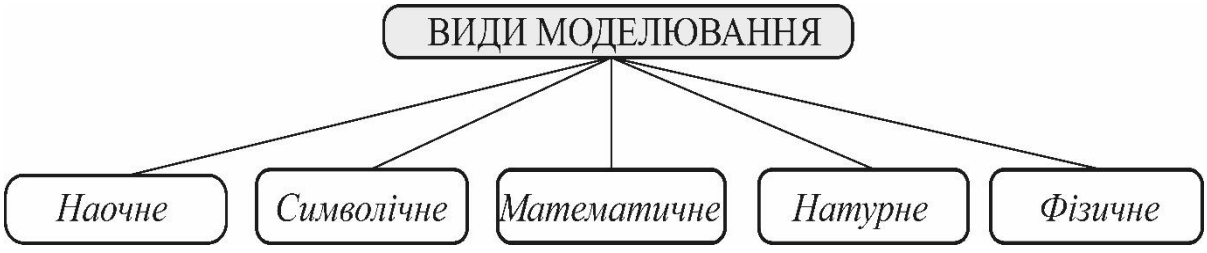


Рис. 1.1. Умовне представлення видів моделювання

Наочне моделювання – це перші спроби досліджень різних систем. Створюється певний макет об'єкта-оригіналу, що досліджується. Під макетом слід розуміти певну зменшену (або навпаки, збільшену) копію будівлі, механізму, пристрою чи якогось складного комплексу. Макет може відрізнятись від оригіналу і розмірами, і матеріалами, і зовнішнім виглядом. У якості прикладу можна запропонувати карту місцевості (мапу), що відтворює рельєф цієї місцевості.

Символічне моделювання широко застосовується у соціальній сфері, якщо потрібно коригувати поведінку людей у колективі чи інші подібні соціальні аспекти людських стосунків. У якості дієвого механізму такий підхід часто використовується менеджерами компаній, психіатрами, психологами та прогнозистами. Отже, символічне моделювання дає змогу краще пізнати людину як соціальну одиницю. Таке моделювання допомагає знаходити відповіді на важливі питання людського життя. Застосовуючи принципи символічного моделювання, особа за допомогою запитань, спрямованих до свого внутрішнього світу (его) має можливість розібратися у своїх почуттях, фобіях, страхах та емоціях. Саме про такий принцип самовдосконалення говорив відомий австрійський психолог і невролог Зигмунд Фройд.

Отже, використовуючи символічне моделювання, людина чи лікар-психолог, ставить запитання, націлені на підсвідомість (внутрішній світ). Усе це дає змогу розкрити внутрішній стан особистості та виявити причини, що перешкоджають людині активно і щасливо існувати у соціумі. Дуже показовим прикладом символічного моделювання є діяльність французького аптекаря і лікаря Еміля Куе, який лікував людей по всьому світу виключно словом і досягнув у цьому відношенні вражаючих результатів. Психолог, що володіє технікою символічного моделювання, модифікує внутрішній світ людини за допомогою спеціально підібраних слів і фраз, спрямованих на підсвідомість.

Символічне моделювання може використовуватися для:

- зміни стилю поведінки людини;
- позбавлення негативних психологічних станів;
- моделювання бажаного стилю і способу життя;
- зміни свого внутрішнього емоційного стану з негативного на позитивний;
- формування позитивних звичок та рис характеру;
- розвитку свого внутрішнього творчого потенціалу;
- досягнення вагомих показників у сфері діяльності кожної конкретної людини;
- поліпшення стосунків із соціумом, особливо у трудовому колективі;
- досягнення цілковитої гармонії внутрішнього світу.

Математичне моделювання – це створення математичної моделі системи, що досліджується, різними математичними методами – аналітичними, число-

вими, емпіричними чи напівемпіричними. Мета математичного моделювання – вивчення характеристик функціонування системи шляхом її формалізації, побудувавши математичну модель.

Завдання математичного моделювання – встановлення аналітичної відповідності між реальним об'єктом та його математичним описом. Дослідження математичної моделі різноманітними аналітичними методами дає змогу одержувати числові характеристики реального об'єкта. Тип та структура математичної моделі залежать насамперед від природи реального об'єкта. До того ж важливим моментом є також поставлені перед дослідником завдання, їх ступінь точності та діапазон застосування. Зрозуміло, що будь-яка модель, зокрема і математична, описує реальний об'єкт лише з деяким ступенем наближення – більшим чи меншим. Умовно математичне моделювання можна формально розділити на три види – аналітичне, імітаційне та комбіноване.

Отже, математична модель – це такий математичний опис системи, який відображає характерні властивості системи чи процесу, що моделюється. Треба підкреслити, що будь-яка модель – це бієктивне відображення реально наявних закономірностей, що дають змогу створити уявлення про реальні природні процеси, явища та закономірності. У цьому принципова вимога накладається на спрощення, які застосовуються під час створення моделі (не тільки математичної). Ці спрощення не повинні спотворювати уявлення про систему, що вивчається, а навпаки, мають допомагати краще розібратись у спектрі закономірностей, притаманних об'єкту. Зауважимо також, що охоплення всіх аспектів реальності не тільки неможливе, а і недоцільне. Однак наявні моделі постійно удосконалюються, наближаючись до реальних систем.

Натурне моделювання – відтворення оригіналу, тобто реального об'єкта, у будь-якому матеріальному вигляді – збільшеному, зменшеному чи такому, як і сам об'єкт. Іншими словами, ми створюємо аналог-прототип об'єкта з метою його дослідження. Ідея полягає в тому, щоб у перспективі перенести властивості із моделі на об'єкт, базуючись на принципах теорії подібності. Скажімо, потрібно побудувати міст через провалля. У цьому випадку створюють зменшений аналог мосту, використовуючи різні варіанти зміцнення конструкції прототипу з метою досягнення його максимальної міцності та здатності витримувати критичні навантаження (ураганні вітри, землетруси чи повені). Дослідивши всі варіанти, вибирають найкращий, і на його основі, використовуючи знову ж таки теорію подібності, будують реальний міст.

Фізичне моделювання – такий метод експериментального вивчення різних об'єктів, процесів, механізмів або явищ, що базується на використанні моделі, яка за своєю фізичною сутністю така сама, як і досліджуваний об'єкт. Специфіка фізичного моделювання базується на подібності між лабораторною

фізичною моделлю та досліджуваним явищем-феноменом. Фізик-експериментатор має можливість проводити різноманітні фізичні експерименти, змінюючи режими проведення таких досліджень. Це, власне кажучи, дає змогу не тільки всебічно вивчити явище, а і використати його для практичних потреб людей чи створення наукових теорій. Отримані на фізичній моделі результати експерименту поширюються потім на явище в реальних масштабах. Зрозуміло, що точно відтворити реальне природне явище практично неможливо. Так само неможливий точний математичний опис явища, але досить наближений до реалій експеримент провести можливо з використанням сучасної наукової апаратури та дослідницьких методик високого рівня.

Експериментальне відтворення досліджуваного фізичного явища в реальних умовах та масштабах часто неможливе. Скажімо, сьогодні неможливо відтворити природне явище, як-от кульова блискавка. Тому перед фізиками-теоретиками та експериментаторами стоїть важливе наукове завдання – так вдосконалити систему рівнянь Максвела, щоб вони описували феномен кульової блискавки. Проте часто метод фізичного моделювання дає достовірні результати. Отже, будь-який лабораторний фізичний експеримент є, власне кажучи, процесом моделювання, оскільки в експерименті вивчається явище природи у лабораторних умовах. Це дає змогу у перспективі отримати загальні закономірності для всього спектру подібних природних явищ у широкому діапазоні умов. Творчий та фаховий підхід фізика-експериментатора полягає в тому, щоб досягнути реальної подібності між досліджуваним явищем, спостережуваним у лабораторних умовах, і всім класом явищ, які вивчаються. Отже, з'являється можливість використати природні явища та закономірності для практичних потреб людства.

Формально в широкому сенсі класифікацію видів моделювання за сферою застосування можна представити так:

- математичне моделювання;
- комп'ютерне моделювання;
- імітаційне моделювання;
- логічне моделювання;
- психолого-педагогічне моделювання;
- статистичне моделювання;
- структурне моделювання;
- фізичне моделювання;
- еволюційне моделювання;
- геометричне моделювання;
- натурне моделювання.

Для класифікації представлених видів моделювання важливо виділити характерні їх ознаки, а саме: особливості процесів, що досліджуються у системі,

та ступінь достовірності моделі. Досліджувані процеси можна поділити на детерміновані та дискретні, стохастичні, статичні та динамічні, безперервні та дискретно-безперервні (рис. 1.2).

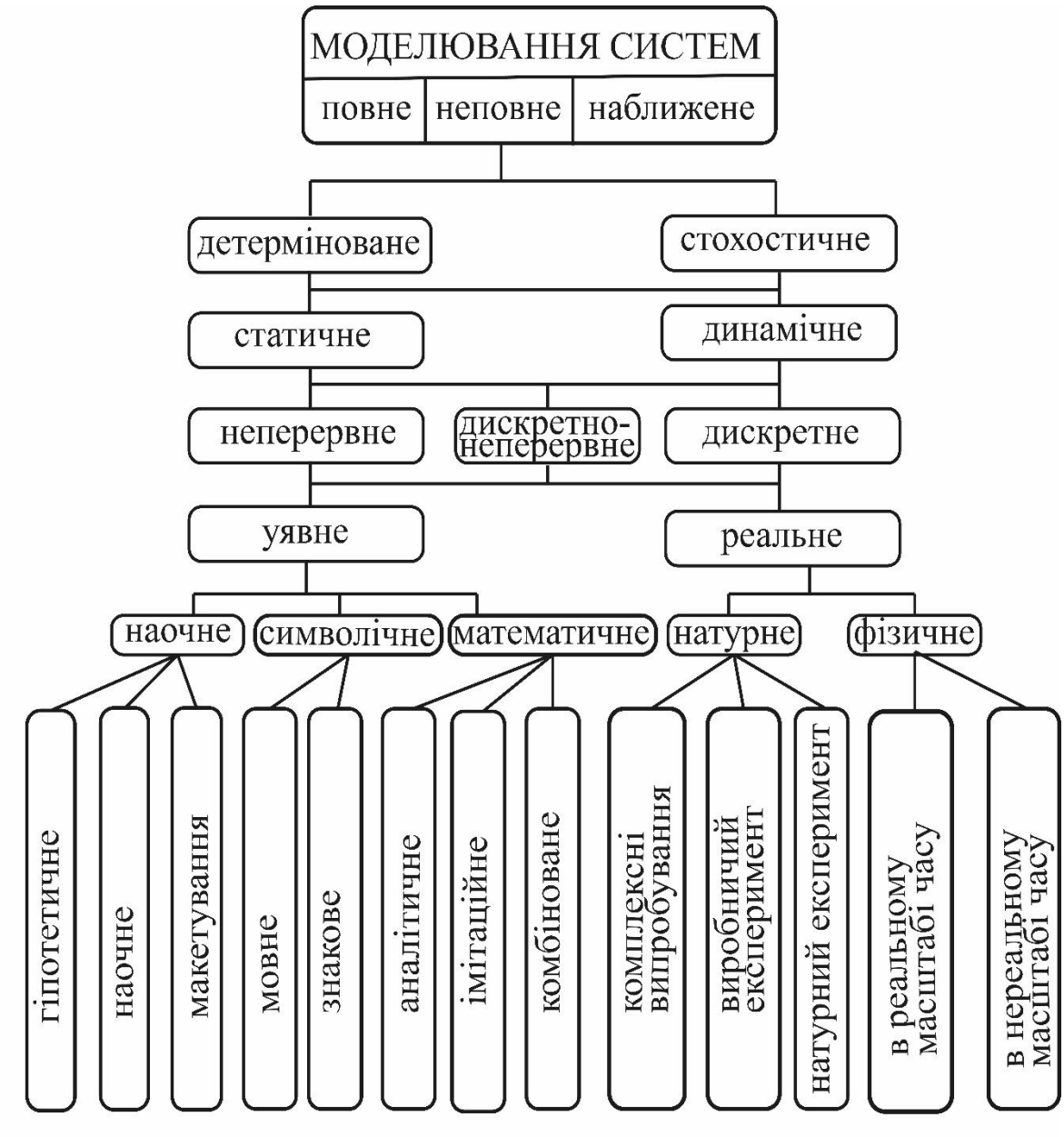


Рис. 1.2. Наочне представлення видів моделювання

Детерміноване моделювання розглядає лише такі особливі процеси, в яких відсутні будь-які випадкові фактори-впливи. Навпаки, стохастичне моделювання реалізує випадкові (ймовірнісні) явища та процеси, що змінюються у випадковий, неконтрольований спосіб (як броунівський рух молекул). Під час аналізу таких процесів до уваги беруться не точні числові дані, а ймовірнісні, що реалізуються у певному діапазоні варіантів. Статичне моделювання слугує для опису поведінки об’єкта в будь-який момент часу, а динамічне, навпаки, відображає зміну поведінки об’єкта у часі. Прикладом динамічного моделю-

вання є, скажімо, процедура опису міського автомобільного трафіку. Динаміка змін трафіку є швидкозмінною функцією від часу. Як один із вагомих принципів класифікації видів моделювання моделі можна умовно розділити на повні, неповні та наближені. Суть повного моделювання полягає у досягненні принципів повної подібності. Така характеристика має проявлятись як у часі, так і у просторі. Якщо ж говорити про неповне моделювання, то природно, має спостерігатись лише деяка, тобто неповна подібність моделі до об'єкта, що розглядається. Отже, сутність неповного (наближеного) моделювання базується на наближеній (приблизній) подібності. За такого підходу деякі особливості та характеристики функціонування реальної системи не моделюються взагалі.

Особливістю дискретного моделювання є опис таких специфічних процесів, що за своєю суттю є дискретними. Скажімо, дискретними є електронні стани у фермі-системах. Водночас електрони займають строго визначені енергетичні рівні та підкоряються статистиці Фермі-Дірака, яка, власне, і є моделлю таких специфічних систем. Під час переходу електрона з одного енергетичного рівня на інший поглинається або навпаки, випромінюється квант енергії (фотон). Навпаки, неперервне моделювання відтворює нерозривні з плином часу процеси в системах. Консенсусом між дискретним на неперервним моделюваннями є дискретно-неперервне моделювання. Такий підхід використовується, коли система проявляє дуальні особливості – як дискретні, так і безперервні. В якості прикладу приведемо поведінку квантових систем за різних температур. За нормальних температур система (матеріал) володіє квазінеперервними квантовими станами, а за зниження температури до так званої критичної відбувається перехід у надпровідний стан – відбувається так звана бозе-конденсація.

Зауважимо далі, що об'єкт (систему) можна представити уявно (наприклад, Всесвіт) або реально (скажімо, автомобіль). Відповідно треба розрізняти реальне та уявне моделювання. Уявне моделювання дуже часто є єдиним можливим засобом моделювання об'єктів, які існують у таких станах і у такій формі, що фізичне їх створення неможливе. Наприклад, засобами уявного моделювання можна відтворити макросвіт (Всесвіт), проте представити цей об'єкт неможливо. Отже, уявне моделювання можна реалізувати з допомогою описового, наочного, символічного або математичного підходів.

Під час наочного моделювання створюються різноманітні описові моделі, що у певний спосіб відображають процеси і зміни, які відбуваються в об'єкті.

Важливим методом наукового пізнання навколишнього світу є гіпотеза. Власне гіпотеза являє собою неперевірену, непідтверджену модель реального світу. Спочатку формулюється власне гіпотеза, а потім на її основі створюється наукова теорія. Отже, гіпотеза як основа наукового пізнання навколишнього світу є специфічним прогнозом – гіпотетичною моделлю, науковим передбаченням. В основу гіпотетичного моделювання вченим закладається науково

обґрунтована гіпотеза щодо закономірностей та динаміки протікання змін у реальному об’єкті. Водночас наукова глибина гіпотези виявляє глибину знань дослідника про об’єкт. Наведемо приклад. Наприкінці 19-го сторіччя вчений-фізик де Бройль сформулював гіпотезу про те, що матеріальні об’єкти мікросвіту (електрони, протони, атоми) володіють хвильовими властивостями. Спочатку науковий світ, зокрема і А. Ейнштейн, виступили проти такої гіпотези. Але не пройшло і десяти років, як гіпотеза де Бройля отримала експериментальне підтвердження. Як результат, була створена теорія (інакше кажучи, модель) корпускулярно-хвильового дуалізму матерії. Можна також сказати, що гіпотетичне моделювання використовується у тих випадках, коли знань про об’єкт замало для побудови моделей.

Аналогове моделювання застосовує аналогії різних видів і рівнів. Найвищим рівнем є, звичайно, повна аналогія. Аналогію такого типу можна застосувати лише для достатньо простих об’єктів. У міру ускладнення об’єкта необхідно використовувати аналогії більш високих рівнів. Водночас аналогова модель відображає лише одну сторону досліджуваного об’єкта.

Всі знайомі з інтерфейсом комп’ютера. На вкладці «Рецензування» зліва розташований сервіс «Тезаурус». Власне це синонімічний словник, який надає приклади синонімів для конкретного слова. Отже, в основі мовного моделювання лежить певний тезаурус. Останній утворюється з набору синонімів цього слова. Відмітимо далі, що між тезаурусом і типовим словником є принципові відмінності. Річ у тім, що тезаурус – специфічний словник, очищений від неоднозначності, тобто кожному слову відповідає лише єдине слово-синонім. Для типового словника така ситуація нехарактерна – одному і тому самому слову можуть відповідати кілька понять.

Аналітичне моделювання (рис. 1.2) характеризується тим, що процеси функціонування системи формулюються за допомогою аналітичних виразів – інтегральних, диференціальних, алгебраїчних тощо. Дослідження аналітичної моделі можна провести як чисельними, так і аналітичними методами.

Фундаментальне вивчення особливостей функціонування системи можна здійснити, використовуючи відомі аналітичні залежності. За допомогою таких залежностей встановлюється зв’язок між шуканими характеристики та параметрами і змінними системи, що вивчається. Однак такі залежності можна отримати лише для нескладних систем. Навпаки, під час дослідження складних об’єктів аналітичним методом можуть виникнути суттєві труднощі, які інколи бувають непереборними. Отже, маючи на меті застосувати аналітичний метод, прагнемо суттєво спростити стартову модель. Це дає змогу вивчити деякі загальні характеристики системи. Дослідження на спрощеній моделі аналітичним методом допомагає отримати приблизні результати. Далі процес пізнання

системи базується на більш точних оцінках або альтернативними методами досліджень, або модифікованим аналітичним методом. Звернемо увагу, що чисельний метод як частковий випадок аналітичного володіє спектром розроблених методів досліджень. Проте отримані з допомогою цього методу розв'язки мають обмежений (частковий) характер. Особливо ефективним чисельний метод стає під час використання програмних засобів, що реалізується на сучасних комп'ютерах з використанням пріоритетних мов програмування, як-от Java, Python, JavaScript або C#.

Якісні методи аналізу математичної моделі в особливих випадках можуть задовольнити ті висновки, які зроблені під час її використання. Згадані методи широко використовують для оцінки ефективності різних способів керування у теорії автоматичного керування.

Особливо вагоме значення відіграють методи реалізації дослідження характеристик процесу функціонування великих систем. Тут потрібно звернути увагу на реалізацію програмних алгоритмів з допомогою Java EE. Для реалізації математичної моделі на комп'ютері необхідно побудувати відповідний моделювальний алгоритм, що складається з послідовності «описовий алгоритм → блок-схемний алгоритм → псевдо-алгоритм → програмний алгоритм».

Імітаційне моделювання реалізує таку специфічну модель, яка відтворює процеси динаміки системи з плином часу. На першому етапі імітуються досить прості процеси зі збереженням їх функціональної структури та темпів протікання. Все це дає змогу одержати відомості про стани системи в дискретні моменти часу і оцінити поведінку системи в динаміці.

Основною перевагою імітаційного моделювання, порівняно з аналітичним, є можливість розв'язання складніших завдань. Імітаційні моделі дають змогу ефективно враховувати вплив факторів, як-от дискретні і безперервні елементи, нелінійні елементи та часті випадкові впливи, що створюють проблеми для аналітичних досліджень. Сьогодні імітаційне моделювання – найефективніший метод дослідження великих систем, а часто і єдиний практично доступний метод отримання інформації про поведінку системи, особливо на етапі її проєктування.

Результати, отримані під час вивчення імітаційної моделі, є реалізаціями випадкових величин і функцій. Отже, для знаходження характеристик процесу потрібне його багаторазове повторення з наступним статистичним опрацюванням інформації. Досить зручно і практично доцільно як засіб реалізації імітаційної моделі використовувати метод статистичного моделювання (інакше – метод статистичних випробувань), що являє собою чисельний метод, який застосовують для моделювання статистичних (випадкових) величин і функцій.

Ймовірнісні характеристики системи дуже зручно досліджувати, використовуючи, зокрема, метод Монте-Карло. Цей метод ефективно застосовують для таких процесів функціонування систем, де проявляються випадкові збурення.

Отже, метод статистичного моделювання дає змогу здійснювати реалізацію імітаційної моделі, а метод статистичних випробувань Монте-Карло дає змогу чисельного розв'язування аналітичної задачі.

Звернемо увагу на те, що метод імітаційного моделювання дає змогу розв'язувати проблеми аналізу складних систем, зокрема оцінювання варіантів структури системи та дієвості різних алгоритмів керування системою. Також імітаційне моделювання може бути покладено в основу алгоритмічного, структурного та параметричного синтезу складних систем. Це буває потрібно для створення системи із заданими параметрами з деякими обмеженнями, яка є оптимальною за певними критеріями ефективності.

Окрім розроблення моделювальних алгоритмів для аналізу фіксованої системи, під час розв'язання проблем синтезу систем на основі їх імітаційних моделей буває потрібно також розробити принципи пошуку оптимального варіанта системи. Водночас необхідно розрізняти два основні моменти – динаміку і статику, основним сенсом яких є відповідно питання синтезу та аналізу систем, заданих моделювальними алгоритмами.

Під час аналізу та синтезу систем аналітико-імітаційне (комбіноване) моделювання дає змогу об'єднати переваги аналітичного та імітаційного моделювання. Будуючи комбіновані моделі, проводять попередню декомпозицію процесу функціонування об'єкта на підпроцеси і для деяких використовують аналітичні моделі, а для інших – будують імітаційні моделі. Такий комплексний (комбінований) підхід дає змогу охопити нові та складніші класи систем, які не можуть бути досліджені з використанням тільки аналітичного чи імітаційного методів.

Під час реального моделювання використовується можливість дослідження різних характеристик або на реальному об'єкті цілком, або на його частині. Дослідження подібного роду можуть проводитися як на об'єктах, що працюють як у звичних режимах, так і під час особливих умов роботи. Реальне моделювання є особливо ефективним, але все ж таки його можливості з урахуванням специфіки реальних об'єктів обмежені. Проведення реального моделювання інформаційної системи підприємства, наприклад, потребуватиме як створення автоматизованої системи управління, так і проведення експериментів з підприємством, що в більшості випадків складно здійснити, а в основному – неможливо. Існує спектр видів реального моделювання.

Розгляд розпочнемо із натурного моделювання, яке являє собою проведення дослідження на конкретному об'єкті з наступним опрацюванням результатів експерименту, базуючись на принципах теорії подібності. Під час функціонування об'єкта відповідно до поставленої мети вдається виявити закономірності протікання реального процесу. Треба зазначити, що різновиди натурного експерименту, як-от виробничий експеримент чи комплексні випробування, мають високий ступінь достовірності.

Технічна оснащеність сучасного наукового експерименту постійно зростає. Експеримент такого типу характеризується широким використанням засобів автоматизації та застосуванням досить різноманітних способів опрацювання інформації. Враховуючи вищесказане, можна констатувати появу нового наукового напряму – автоматизації наукових експериментів.

Принципова відмінність експерименту від реального протікання процесу полягає в тому, що у реальності можуть виникнути непередбачувані критичні ситуації. Так трапилося, зокрема, під час проведення експериментів на Чорнобильській АЕС. З метою імітації таких критичних ситуацій під час експерименту інсталюються такі фактори, що вводять експеримент у критичну область. Так моделюються екстремальні ситуації у процесі функціонування модельованого об'єкта. Один із різновидів експерименту – комплексні випробування, які також можна віднести до натурного моделювання, коли внаслідок повторення випробувань виробів виявляють загальні закономірності щодо надійності цих виробів та їх функціональної придатності. У цьому разі моделювання здійснюється шляхом опрацювання та узагальнення даних, що охоплюють весь можливий діапазон функціонування модельованого об'єкта. Поряд зі спеціально організованими випробуваннями можлива реалізація натурного моделювання шляхом узагальнення досвіду, накопиченого під час виробничого процесу. У цьому випадку ідеться про виробничий експеримент. Тут на базі теорії подібності обробляють статистичний матеріал щодо виробничого процесу й отримують його узагальнені характеристики.

Іншим видом реального моделювання є фізичне, яке відрізняється від натурного тим, що дослідження проводять на установках, які відтворюють природу явищ. У процесі фізичного моделювання експериментатори задають деякі характеристики зовнішнього середовища і досліджують поведінку або реального об'єкта, або його моделі за заданих або створюваних штучно впливів. Фізичне моделювання може здійснюватися як з урахуванням фактора часу, так і без такого. В останньому випадку вивченню підлягають стаціонарні процеси, зафіксовані в деякий момент часу. Найбільшу важливість з погляду достовірності одержуваних результатів являє собою фізичне моделювання в реальному масштабі часу.

З погляду математичного опису об'єкта моделі умовно можна розділити на цифрові, аналогові і аналого-цифрові. Під цифровою розуміють таку модель, яка описується рівняннями, що зв'язують дискретні величини, представлені у цифровому вигляді. Під аналоговою моделлю розуміють модель, яка описується рівняннями, що зв'язують безперервні величини. Аналого-цифровою називається модель, яку можна описати рівняннями, що зв'язують як безперервні, так і дискретні величини.

Особливе місце посідає кібернетичне моделювання, у якому відсутня безпосередня подібність фізичних процесів, що здійснюються в моделях, до реальних процесів. У цьому разі прагнуть відобразити лише деяку функцію і розглядають реальний об'єкт як «чорну скриньку», що має низку входів і виходів. Моделюванню підлягають деякі зв'язки між виходами і входами. Найчастіше під час використання кібернетичних моделей проводять аналіз поведінкової сторони об'єкта за різних впливів зовнішнього середовища.

Отже, в основі кібернетичних моделей лежить відображення деяких інформаційних процесів управління, що дає змогу оцінити поведінку реального об'єкта. Для побудови імітаційної моделі необхідно виокремити досліджувану функцію реального об'єкта та спробувати формалізувати цю функцію у вигляді деяких операторів зв'язку між входом і виходом. Потім відбувається процес відтворення на імітаційній моделі цієї функції на основі інших математичних співвідношень.

Особливим видом моделювання є макетування. Макет може застосовуватися у випадках, коли процеси, що протікають у реальному об'єкті, не піддаються фізичному моделюванню. Можлива ситуація, коли дослідження макета передує проведенню інших видів моделювання. В основі побудови макетів лежать аналогії, які базуються на причинно-наслідкових зв'язках між явищами і процесами в об'єкті.

1.3. Процес моделювання

Процес моделювання включає три елементи: об'єкт моделювання, суб'єкт моделювання та модель, що відображає взаємодію між об'єктом та суб'єктом.

Моделювання даних у широкому сенсі – це створення віртуального образу про систему або її окремий елемент. Мета полягає у тому, щоб відобразити особливості функціонування системи загалом, відношення між компонентами системи, принципи організації та зберігання даних.

Моделі часто використовуються, зокрема, для потреб бізнес-структур. Правила і вимоги до моделі даних у цьому випадку визначаються заздалегідь на основі зворотного зв'язку з бізнесом. Внаслідок цієї особливості їх (моделі) можна включити в розробку нової системи або адаптувати до наявної.

Процедуру моделювання даних для бізнесу можна здійснювати на різних рівнях абстракції. Процес починається зі збору бізнес-вимог від зацікавлених сторін і кінцевих користувачів. Ці бізнес-правила потім перетворюються на структури даних. Модель даних можна порівняти з дорожньою картою, планом

архітектора або будь-якою формальною схемою, яка сприяє глибшому розумінню того, що розробляється.

Моделювання даних використовує стандартизовані схеми та формальні методи. Це забезпечує послідовний і передбачуваний спосіб управління даними в організації або за її межами.

В ідеалі моделі даних – це живі документи, які розвиваються разом із потребами бізнесу. Вони відіграють важливу роль у підтримці бізнес-процесів і плануванні IT-архітектури та стратегії. Моделями даних можна ділитися з постачальниками, партнерами та колегами.

Переваги моделювання даних полягають у організації взаємозв'язків між даними для розробників, архітекторів даних, бізнес-аналітиків та інших зацікавлених осіб. До того ж моделювання даних допомагає:

- зменшити кількість помилок під час розроблення програмного забезпечення та баз даних;
- уніфікувати документацію на підприємстві;
- підвищити продуктивність додатків і баз даних;
- спростити відображення даних в усій організації;
- поліпшити взаємодію між розробниками та командами бізнес-аналітиків;
- спростити і прискорити процес проєктування бази даних на концептуальному, логічному і фізичному рівнях.

За типами моделей даних відбувається проєктування та розробка баз даних та інформаційних систем. Усе починається з високого рівня абстракції, що з кожним кроком стає дедалі точнішим та конкретнішим. Залежно від ступеня абстракції моделі даних можна розділити на три категорії. Процес починається з концептуальної моделі, переходить до логічної і завершується фізичною моделлю.

Концептуальні моделі даних описують загальну картину – що міститиме система, як її буде організовано та які процеси буде задіяно. Концептуальні моделі створюють під час збору вихідних вимог до проєкту. Зазвичай вони включають класи сутностей, їх характеристики та обмеження, відношення між сутностями, вимоги до безпеки та цілісності даних.

Перший етап побудови моделі передбачає наявність деяких знань про систему. Особливості моделі зумовлюються тим, що вона відображає (відтворює, імітує) деякі істотні риси системи. Питання про необхідну і достатню міру подібності оригіналу і моделі потребує конкретного аналізу. Очевидно, модель втрачає свій сенс як у разі тотожності з оригіналом (тоді вона перестає бути моделлю), так і в разі повної відмінності в усьому від оригіналу. Отже, вивчення одних сторін модельованого об'єкта здійснюється ціною відмови від дослідження інших сторін. Тому будь-яка модель заміщує оригінал лише в суворо обмеженому сенсі. Із цього випливає, що для одного об'єкта може бути побудовано

кілька «особливих» моделей, що концентрують увагу на певних сторонах досліджуваного об'єкта або ж характеризують об'єкт із різним ступенем деталізації.

На другому етапі модель виступає як самостійний об'єкт дослідження. Однією з форм такого дослідження є проведення модельних експериментів, під час яких свідомо змінюються умови функціонування моделі та систематизуються дані про її особливості. Кінцевим результатом цього етапу є спектр знань про модель.

На важливому третьому етапі здійснюється перенесення знань з моделі на оригінал. Це основний етап. Одночасно відбувається консолідація моделі та оригіналу. Процес трансформації знань у напрямках «модель ↔ оригінал» проводиться за певними правилами. Знання про модель мають бути скориговані з урахуванням тих властивостей об'єкта-оригіналу, які не знайшли відображення або були змінені під час побудови моделі.

Четвертий етап – практична перевірка одержуваних за допомогою моделей знань і їх використання для побудови узагальнюючої теорії об'єкта.

Моделювання – багатокроковий та циклічний процес. Це означає, що за першим циклом може настати другий, третій тощо. Водночас знання про досліджуваний об'єкт розширюються, а вихідна модель поступово вдосконалюється. Недоліки, виявлені після першого циклу моделювання, зумовлені недостатнім знанням об'єкта або помилками в побудові моделі, потрібно скоригувати в наступних циклах.

Зараз важко вказати сферу людської діяльності, де не застосовувалося б моделювання. Розроблено, наприклад, моделі цехів виробничих підприємств, вирощування гречки, функціонування окремих систем життєдіяльності людини тощо. У перспективі для кожної системи можуть бути створені свої моделі, так що перед реалізацією кожного технічного або організаційного проєкту має проводитися моделювання.

Показовим прикладом є наукове моделювання прямих вимірювань та експериментів. Такого типу моделі зазвичай використовують, коли неможливо або непрактично створювати експериментальні умови, за яких вчені можуть безпосередньо отримувати результати. Пряме вимірювання у контрольованих умовах завжди буде надійнішим, ніж змодельовані оцінки результатів.

Під час моделювання здійснюється цілеспрямоване спрощення й абстрагування реальності, зумовлене фізичними та когнітивними обмеженнями. Моделювання – акція, орієнтована на розв'язання певних заданих проблем або завдань.

Будь-яка модель неможлива без спрощень. Важливо встановити суть таких спрощень, щоб дослідження моделі не привели до неправильних висновків про систему. Спрощення не враховують деякі сутності системи, які не є важливими

для задачі, що розглядається. Абстракція сепарує інформацію, що важлива для вивчення системи з погляду вирішення поставленого завдання. Обидві дії, спрощення й абстракція, виконуються цілеспрямовано. Однак вони зроблені на основі сприйняття реальності. Це сприйняття вже саме по собі є моделлю, оскільки воно пов'язане з фізичними обмеженнями.

Існують також обмеження на те, що ми можемо формально спостерігати за допомогою нашого інструментарію та застосовуваних методів. Часто розглядається модель, яка включає сутності, їх поведінку та формальні відношення між сутностями. Така модель носить назву концептуальної. Щоб створити цю модель, потрібно здійснити процедуру комп'ютерного моделювання. Для цього потрібна велика вибірка через застосування, наприклад, чисельної апроксимації або використання евристики. Незважаючи на всі ці обчислювальні обмеження, моделювання було визнано як один із трьох ключових компонентів наукових методів. Назви цих методів – теорія, моделювання та експериментування.

Одним із важливих способів моделювання є симуляція, що досліджує комплексні процеси поведінки моделі в межах заданих умов моделювання. Статична симуляція надає інформацію про систему у певний заданий момент часу. Навпаки, динамічна симуляція надає інформацію у часовому режимі. Симуляція активізує модель і показує, як поводитиметься конкретний об'єкт або явище. Згаданий спосіб моделювання може бути корисним для тестування, аналізу або навчання в тих випадках, коли концепції реального світу можуть бути представлені у вигляді їх моделей. Наведемо приклад із життя нашої країни, що відвойовує свою свободу у боротьбі із північним ворогом – Росією. Українській армії вкрай потрібні американські винищувачі F-16. Проте для підготовки українських пілотів, які зможуть вести активні бойові дії на винищувачах подібного виду, потрібен досить значний проміжок часу – можливо, пів року. Як можна швидко та ефективно підготовити пілотів, які зможуть воювати на F-16? Суттєво прискорити процес підготовки пілотів можна з допомогою використання спеціальних тренажерів-авіасимуляторів. Наші друзі із Чехії вирішили надати Україні такі пристрої (див. https://m.youtube.com/ watch?v= 5POsKaULYsE). Панель керування симулятором показана на рис. 1.3. Тренажер імітує реальний політ винищувача F-16, що дає змогу здійснювати ефективний тренінг український пілотів, які навчаються керувати винищувачами, вкрай необхідними українській армії для проведення наступальних дій. Важливо, що термін підготовки наших пілотів завдяки використанню авіасимуляторів F-16 може суттєво скоротитись – із шести до двох місяців. Тепер стає зрозумілим, наскільки життєво важливим є питання симуляції. Звичайно, такі тренажери повинні бути високого класу та повною мірою імітувати реальні бойові дії винищувача F-16.

Рис. 1.3. Авіасимулятор F-16

Створення моделі являє собою процес концептуального представлення деякого явища. Зазвичай модель враховує тільки деякі аспекти розглядуваного явища, і дві моделі одного й того самого явища можуть істотно відрізнятися. Такі відмінності можуть бути викликані різними вимогами кінцевих користувачів цієї моделі або концептуальними чи естетичними уподобаннями конструкторів моделі та їхніми рішеннями, прийнятими під час процесу моделювання. Міркування конструкторів, які можуть вплинути на структуру моделі, можуть бути викликані особистими професійними уподобаннями. У будь-якому разі користувачам моделі необхідно зрозуміти зроблені припущення.

Для побудови моделі потрібна абстракція. В моделюванні використовуються різноманітні припущення. Наприклад, спеціальна теорія відносності стосується лише інерційної системи відліку. Модель робить точні передбачення, коли її припущення дійсні і, з великою ймовірністю, не дає точних прогнозів, коли її припущення не виконуються. Такі припущення часто містять у собі елементи попередніх варіантів моделі. Отже, коли старі теорії змінюються новими, то має місце генезис старого з новим. Зокрема, загальна теорія відносності працює і в неінерційних системах відліку, але як частковий випадок використовує також інерційні системи.

Модель оцінюють насамперед за її узгодженістю з емпіричними даними: будь-яка модель, несумісна з відтворюваними спостереженнями, має бути змінена або відхилена. Один зі способів змінити модель – це обмеження сфери її застосування так, щоб спостерігався збіг зі спостереженнями з високим ступенем достовірності. Наприклад, ньютонівська фізика, яка коректно описує рух та взаємодію тіл за винятком дуже малих та дуже швидких природніх процесів. Проте відповідності тільки емпіричним даним недостатньо для того,

щоб модель була прийнята як дійсна. Інші фактори, важливі під час оцінки моделі, включають:

- можливість пояснення минулих спостережень;
- можливість прогнозування майбутніх спостережень;
- вартість використання, особливо в поєднанні з іншими моделями;
- застосовність, що дає змогу оцінити ступінь достовірності моделі;
- ефективність під час використання.

З огляду на перераховані критерії користувач моделі може спробувати кількісно оцінити її за допомогою функції корисності, визначивши для себе фактори пріоритетності.

Візуалізація – це будь-який спосіб створення зображень, діаграм або анімацій для комунікаційного повідомлення. Візуалізація за допомогою образів є ефективним способом комунікаційного обміну як між абстрактними, так і майже конкретними сутностями. Наочні та яскраві візуальні моделі дають змогу створювати згадувана вже програма AnyLogic, про яку йтиметься буде йти далі. Просторовий маппінг (рис. 1.4) також певною мірою дає змогу створювати моделі різної складності.

В інженерній практиці маппінг відображає модель місцевості і реальних дій (процесів) на цій місцевості. Прикладам такого маппінгу може бути карта DeepState (рис. 1.4).

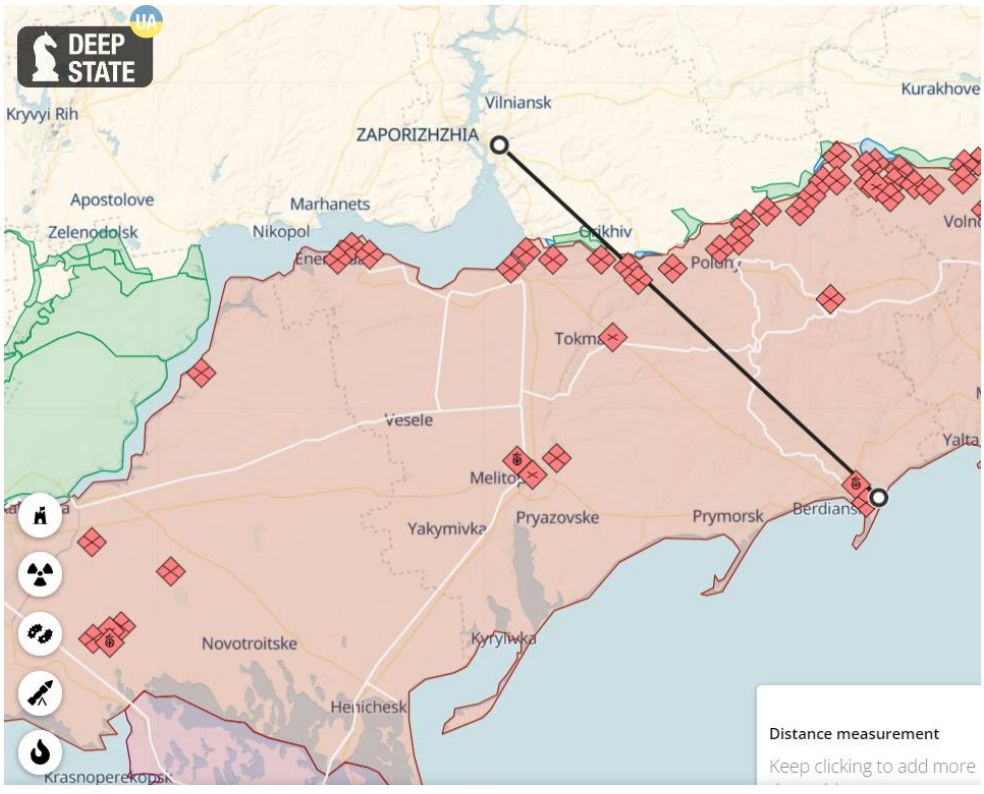


Рис. 1.4. Мапа deepState російсько-української війни,
що ілюструє ситуацію на фронті на 18.06.2023

1.4. Ізоморфні та гомоморфні моделі

Модель за своєю природою і призначенням відтворює лише деяку частину системи. У цьому випадку важливе адекватне перенесення відтворюваних властивостей системи на модель. Загалом формально можна стверджувати, що модель і система перебувають у деяких співвідношеннях, від яких залежить ступінь адекватності моделі. Оцінити ступінь відповідності між моделлю та системою можна, використовуючи формальні поняття ізоморфізму та гомоморфізму. Система і модель є ізоморфними, якщо встановлена певна однозначність між цими сутностями. Реально потрібно адекватно відтворити лише деякі характеристики системи, а не весь спектр її властивостей. Тому ізоморфізм є лише формальним показником, і на практиці досягати виконання повної ізоморфності між моделлю і системою не потрібно, та зробити це зазвичай неможливо. Але тоді виникає питання: в чому цінність моделей? Сутність моделювання і його практична цінність у тому, щоб адекватно (ізоморфно) відобразити (відтворити) лише вибрані характеристики системи. Це може бути всього лише одна така характеристика. Проте вона повинна точно відбиватись у моделі, тобто вести себе у процесі моделювання точно так, як ця особливість проявляє себе у системі. У цьому випадку йдеться про гомоморфні зв'язки. Вони організовують менш тісну відповідність між моделлю та системою. Проте це не означає, що такого типу зв'язки менш цінні. Все залежить від точності відображення характеристик системи у моделі. І якщо ми відображаємо лише одну характеристику, але робимо це з великим ступенем достовірності, то наша модель загалом заслуговує високої оцінки. Така модель не дасть хибних результатів, і її використання на практиці принесе велику користь. І навпаки, ізоморфна модель, яка не завжди відтворює поведінку системи, може привести до дуже згубних наслідків у випадку використання її на практиці.

Не існує ідеальних моделей. Модель не може повністю відповідати системі. Вона не може бути клоном системи. Для того, щоб досягти ізоморфності між сутностями, потрібно розглядати спрощену систему, тобто таку, у якій розглядається зменшений спектр модельованих характеристик; їх ще називають атрибутами.

Отже, узагальнюючи сказане, можна стверджувати, що процес моделювання оперує категоріями спрощення, аналогії та абстракції. Отже, аналогія, абстракція та спрощення – це основні поняття, які характеризують взаємовідносини типу «модель – система». Загалом між моделлю та системою можуть бути встановлені такі типи відношень:

1. ***Детерміновані відношення*** – встановлюють взаємнооднозначну відповідність між моделлю та системою. Створюється така специфічна модель, у

якій представлена скінченна множина детермінованих станів. Типовим прикладом у цьому випадку може бути мережа Петрі.

2. ***Відношення з обмеженим спектром станів,*** що реалізуються з певною величиною ймовірності. Модель реалізується лише з деяким ступенем ймовірності відносно системи. Ймовірнісний автомат може бути наочним прикладом подібної моделі

3. ***Між системою та моделлю установлюються ймовірнісні відношення,*** так що стани системи і моделі визначають стани один одного лише з деяким ступенем достовірності. До такого типу належать стохастичні моделі та моделі систем масового обслуговування.

РОЗДІЛ 2
МАТЕМАТИЧНЕ МОДЕЛЮВАННЯ

2.1. Етапи математичного моделювання

Математичне моделювання певною мірою можна вважати і мистецтвом, і наукою. Відомий фахівець в області імітаційного моделювання Роберт Шеннон так назвав свою широко відому в науковому та інженерному світі книгу: «Імітаційне моделювання систем – мистецтво і наука». Отже, неможливо запропонувати строгий алгоритм створення моделей різних систем. Проте аналіз прийомів, які використовують розробники моделей, дає змогу виділити ряд етапів моделювання.

Перший етап: визначення і з'ясування цілей моделювання. Це головний етап будь-якої діяльності. Мета істотно визначає зміст інших етапів моделювання. Зауважимо, що різниця між простою і складною системами породжується не тільки їх сутністю, але і цілями, які ставить дослідник.

Зазвичай цілями моделювання є:

- прогноз поведінки об'єкта за нових режимів і змінних факторів впливу;
- підбір та поєднання факторів, що забезпечують оптимальні показники ефективності процесу;
- аналіз чутливості системи до зміни факторів;
- перевірка різних гіпотез про характеристики випадкових параметрів досліджуваного процесу;
- визначення функціональних зв'язків між поведінкою (реакцією) системи і факторами, які впливають;
- отримання вторинної моделі у вигляді регресійної залежності методами, які розглядатимуться пізніше;
- з'ясування суті, краще розуміння об'єкта дослідження, а також формування перших навичок для експлуатації модельованої або діючої системи.

Другий етап: побудова концептуальної моделі. Концептуальна модель формується під час вивчення модельованого об'єкта. На цьому етапі досліджуються всі процеси, що протікають в об'єкті, та встановлюються необхідні спрощення і допущення. Виявляються істотні аспекти, виключаються другорядні. Якщо можливо, концептуальна модель представляється у вигляді відомих і добре розроблених систем: масового обслуговування, управління, авторегулювання і под. Концептуальна модель повністю підводить підсумок процесу вивчення проєктної документації або експериментального обстеження об'єкта, що

моделюється. Результатом другого етапу є узагальнена схема моделі, повністю підготовлена для математичного опису – побудови математичної моделі.

Третій етап: вибір мови програмування або моделювання, розробка алгоритму і програми моделі. Модель може бути аналітичною або імітаційною чи їх поєднанням.

В історії математики є багато прикладів того, коли необхідність моделювання різноманітних процесів приводила до нових відкриттів. Наприклад, необхідність моделювання нерівномірного руху привела до відкриття та розробки диференціального числення. Проблеми аналітичного моделювання стійкості кораблів привели вчених до створення теорії наближених обчислень і аналогової обчислювальної машини.

Результатом третього етапу моделювання є комп'ютерна програма, складена на найбільш прийнятній для моделювання мові.

Четвертий етап: планування експерименту. Математична модель є об'єктом експерименту. Експеримент повинен бути максимально інформативним, задовольняти обмеження, забезпечувати отримання даних з необхідною точністю і достовірністю. Для цієї мети створена теорія планування експерименту. Отже, результат четвертого етапу – план експерименту.

П'ятий етап: виконання експерименту з моделлю. Якщо модель аналітична, то експеримент зводиться до виконання розрахунків за умови варійованих вихідних даних. Під час імітаційного моделювання модель реалізується на комп'ютері з фіксацією і наступною обробкою одержуваних даних статистичними та іншими методами. Експерименти проводяться відповідно до плану, який може бути включений в алгоритм моделі. У сучасних системах моделювання така можливість є.

Шостий етап: обробка, аналіз та інтерпретація даних експерименту. Відповідно до мети моделювання застосовуються різноманітні методи обробки: визначення різноманітних характеристик випадкових величин і процесів, виконання аналізу – дисперсійного, регресійного, кореляційного та ін. Багато з цих методів входять у системи моделювання (GPSS World, AnyLogic North America LLC та ін.) і можуть застосовуватися автоматично. Не виключено, що під час аналізу отриманих результатів модель може бути уточнена, доповнена або навіть повністю переглянута.

Після аналізу результатів моделювання здійснюється їх інтерпретація, тобто адаптація результатів до сфери застосування. Це необхідно, тому що зазвичай фахівець предметної області (той, кому потрібні результати досліджень) не володіє термінологією математики та моделювання або володіє, але недостатньо, і може виконувати свої завдання, оперуючи лише добре знайомими йому поняттями.

2.2. Приклади математичних моделей

З метою застосування математики до реалій світу вчені повинні працювати з науковцями та інженерами, перетворювати реальні життєві проблеми на математичні, а потім розв'язувати отримані рівняння. Це і є процес математичного моделювання. Розглянемо основні ідеї, що лежать в основі математичного моделювання, і спробуємо описати його можливості та обмеження.

Математичне моделювання – це процес опису реальної проблеми в математичних термінах, зазвичай у вигляді рівнянь, а потім використання цих рівнянь як для того, щоб допомогти зрозуміти вихідну проблему, так і для того, щоб виявити нові особливості проблеми. Моделювання лежить в основі більшої частини нашого розуміння світу, а також дає змогу інженерам розробляти технології майбутнього. За допомогою моделювання ми можемо подорожувати до краю Всесвіту, зазирнути в глибину атома і зрозуміти майбутнє нашої планети.

Всі добре знайомі з одним із застосувань математичного моделювання, а саме з прогнозом погоди. Сучасний прогноз погоди (рис. 2.1) базується на таких кроках:

- використання законів фізики;
- кодування їх у вигляді диференціальних рівнянь, зокрема рівнянь Нав'є–Стокса;
- отримання даних із супутників і метеостанцій для точного визначення погоди;
- використовуючи ці дані як початкову умову, розв'язати рівняння за лічені хвилини, щоб отримати погоду на наступний день;
- постійне оновлення прогнозу;
- представлення результатів прогнозу.

На рис. 2.1 показаний прогноз погоди для значної частини планети Земля. Проте жителів сіл і міст в основному цікавить прогноз погоди місцевого характеру: люди ставлять прості запитання – чи буде дощ або сніг і коли? Яка буде температура повітря? Буде хмарна чи вітряна погода? Відповіді на поставлені запитання дає регіональний прогноз погоди, який використовує дані, отримані від метеоцентру країни чи регіону.

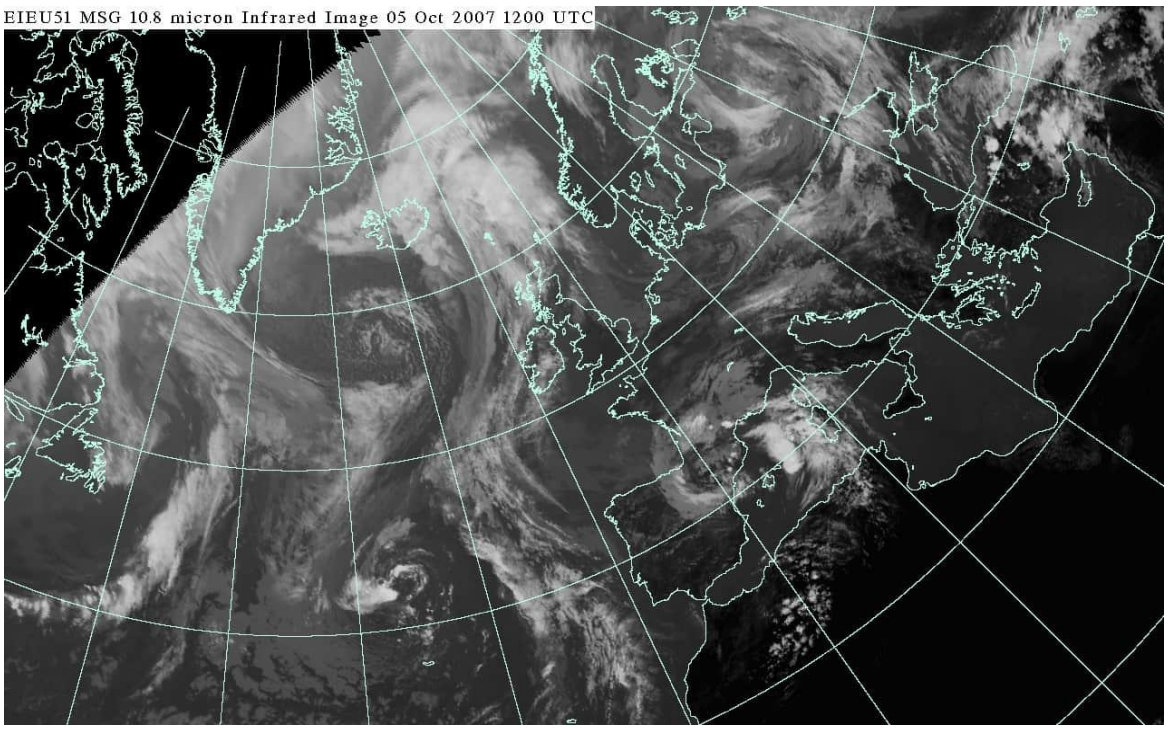


Рис. 2.1. Мапа прогнозованої погоди на планеті Земля

Те, що ми описали вище, є симуляцією. Різниця між симуляцією та моделлю полягає в тому, що в симуляції намагаються максимально точно врахувати всі деталі, щоб висновки були якомога точнішими. Використовуючи симуляції, можна, наприклад, заздалегідь визначити, чи витримає міст максимально допустиме розраховане аналітично навантаження після того, як його побудували. Ми також можемо протестувати міст на руйнування, навіть не будуючи його, просто змінюючи параметри в комп'ютерній симуляції. Ще одне важливе застосування симуляції – це навчання пілотів на авіаційних тренажерах, які розроблені так, щоб бути максимально наближеними до реальності. Використовуючи їх, пілот може навчитися керувати літаком і діяти в небезпечних ситуаціях задовго до того, як йому доведеться сісти в кабіну. Одним із наочних прикладів симулятора є комп'ютерна програма.

Хоча симулятори дуже корисні, проте вони володіють суттєвими недоліками. Потреба у високій точності означає, що рівняння зазвичай занадто складні для аналітичного розв'язання. Натомість їх часто доводиться розв'язувати за допомогою великих суперкомп'ютерів. Чим потужніший комп'ютер, тим краще. Такі симуляції часто займають багато часу, споживають багато енергії і створюють величезні обсяги даних. Настільки багато даних, що часто буває важко зрозуміти, що є важливим, а що неважливим.

Другий недолік полягає у тому, що симуляції мають тенденцію працювати і застосовуватися лише до тих проблем, які добре вивчені фундаментальною

наукою. Однією з причин цього є те, що написання і кодування симулятора – це значний обсяг роботи.

У певному сенсі симуляція протиставляється математичній моделі. Це спрощення проблеми та зведення її до невеликої системи рівнянь, які відображають основну суть і, що важливо, є достатньо простими, щоб дати змогу зробити аналітичні розрахунки. Формула, отримана внаслідок аналітичних розрахунків, може дати чітке уявлення про роль параметрів у цій системі без необхідності виконувати дуже велику кількість обчислень.

Можливо, найпершим прикладом математичної моделі з величезною прогностичною силою був закон тяжіння Ньютона, застосований до Сонячної системи. Замість того, щоб моделювати всю Систему у всій її складності, Ньютон розглядав Сонце і планети як окремі точки. Це дало змогу записати основні рівняння руху всієї Сонячної системи (рис. 2.2).

$$F_{1,2} = \gamma \frac{m_1 m_2}{r^2} \tag{2.1}$$

Тут m_1 та m_2 – маси матеріальних точок, γ – гравітаційна стала, r – відстань між матеріальними точками.

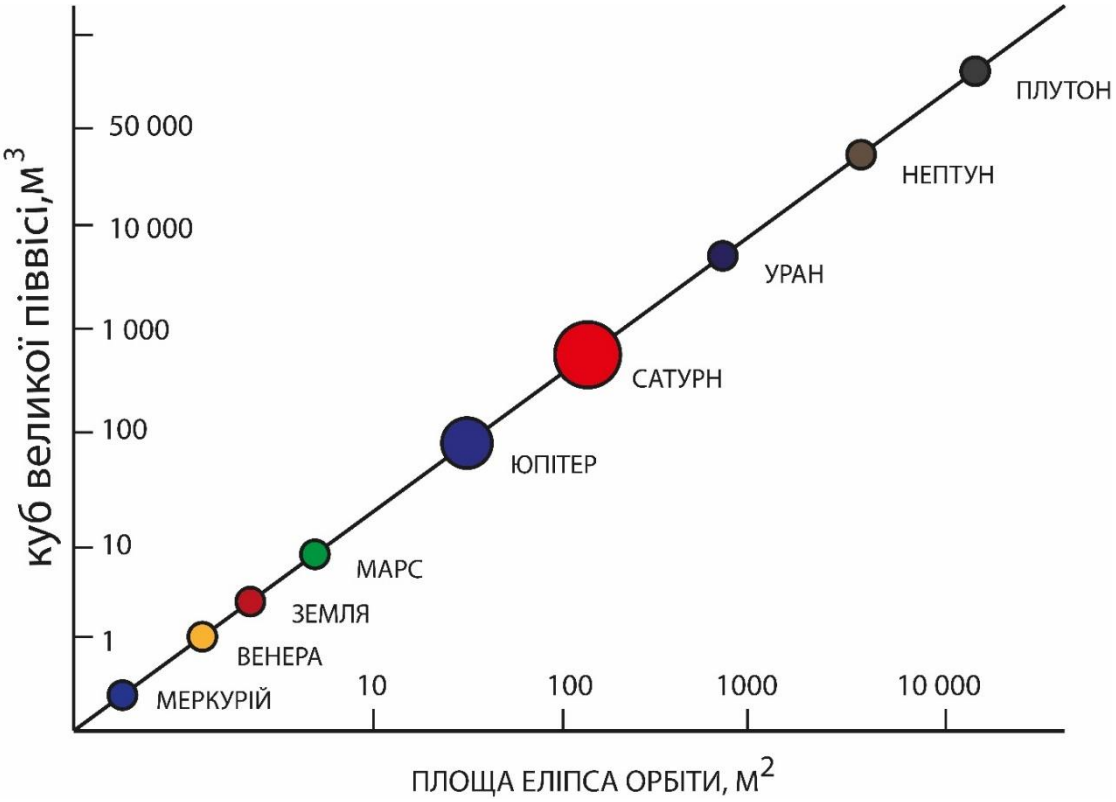


Рис. 2.2. Моделювання Сонячної системи

На цьому рисунку представлені наочно планети Сонячної системи, які Ньютон розглядав як матеріальні точки, оскільки розміри планет у діаметрі становили тисячні частки відсотка відносно розмірів самої Сонячної системи. Це дало змогу вченому використати закон всесвітнього тяжіння (2.1), застосовний лише до матеріальних точок:

2.3. Засоби математичного моделювання

Перелік засобів математичного моделювання великий. Для прикладу наведемо деякі з них: **Maple, VisSim, Mathematica, MathCad, DERIVE, MatLab, DinSYS, Genius.** Названі інструменти суттєво збільшують можливості математичного моделювання.

Сервіс **Maple** – досить зручний математизований програмний пакет, що дає змогу проводити широкий спектр математичних розрахунків, будувати графіки, обчислювати інтеграли та загалом проводити математизовані комп'ютерні експерименти. Пакет дає змогу створювати інтегровані середовища, використовуючи мови програмування високого рівня. Візуалізація даних – також один із можливих варіантів використання цього пакету. Робота з Maple має творчий характер, оскільки проходить інтерактивно, бо користувач має можливість контролювати на екрані результати своєї роботи. Характерно, що Maple не вимагає строгої формалізації змінних, тому з ним досить просто працювати. Інтерфейс простий та зручний. Зокрема, натиснувши Help, маємо можливість знайти у базі даних будь-який потрібний нам приклад чи підказку. Maple ефективно організовує зворотний зв'язок з користувачем, перевіряючи правильність введення даних. Ядро символьних обчислень Maple містить цілий спектр систем комп'ютерної математики – Programming, Curve Fitting, Differential Equation, Discrete Mathematics, Finance, Linear Algebra, Statistic and Probability, Optimization, Signal Processing and Visualization.

Звичайно, кваліфікований користувач, який достатньою мірою володіє однією з мов програмування (C#, Java, JavaScript, Python та іншими), може самостійно створити окрему програму або комплекс програм, що дасть змогу реалізувати на комп'ютері алгоритм його задачі. Проте такий підхід потребує зазвичай великих працезатрат на програмування, налагодження та тестування кожної програми. Тому для скорочення часу програмування було створено згадані прикладні програмні пакети, області використання яких суттєво перетинаються. Для найбільш ефективного використання обчислювальної техніки необхідно правильно вибрати найкращий пакет програм на ранній стадії розв'язання прикладної задачі. Адже реальна мета полягає у вирішенні певної проблеми, а обчислення – лише проміжний етап на шляху до цього вирішення.

Під час дослідження систем автоматичного регулювання та обчислювальних математичних задач найбільш ефективним є використання програмної системи MatLab з широким класом предметно-орієнтованих бібліотек та інструментом візуального моделювання Simulink. У системі MatLab також існують широкі можливості для програмування. Її бібліотека є об'єктною і містить понад 300 процедур обробки даних. Усередині пакета можна використовувати як процедури самої MatLab, так і стандартні процедури мови С, що робить цей інструмент наймогутнішою підмогою під час розробки різних додатків.

Для візуалізації моделювання система MatLab має бібліотеку Image Processing Toolbox, що забезпечує широкий спектр функцій, які підтримують візуалізацію проведених обчислень безпосередньо із середовища MatLab, а також можливість побудови алгоритмів обробки зображень. Систему MatLab можна використовувати для обробки зображень, сконструювавши власні алгоритми, які будуть працювати з масивами графіки як з матрицями даних. Оскільки мова MatLab оптимізована для роботи з матрицями, забезпечується простота використання, висока швидкість і економічність проведення операцій над зображеннями.

Серед інших бібліотек системи MatLab можна також зазначити System Identification Toolbox – набір інструментів для створення математичних моделей динамічних систем, заснованих на спостережуваних даних. Особливістю цього інструменту є наявність гнучкого користувацького інтерфейсу, що дає змогу організувати дані. Бібліотека System Identification Toolbox підтримує як параметричні, так і непараметричні методи. Інтерфейс системи полегшує попередню обробку даних, роботу з ітеративним процесом створення моделей для одержання оцінок і виділення найбільш значимих даних. Стосовно математичних обчислень MatLab надає доступ до величезної кількості підпрограм, які містяться в бібліотеці NAG Foundation Library. Цей інструмент має сотні функцій з різних областей математики. Це унікальна колекція реалізацій сучасних чисельних методів комп'ютерної математики, створених за останні роки. Додану до системи велику кількість документації цілком можна розглядати як фундаментальний багатотомний електронний довідник із математичного забезпечення. Сьогодні система MatLab широко використовується в техніці, науці та освіті, але все-таки вона більше підходить для аналізу даних і організації обчислень.

На відміну від потужного та орієнтованого на високоефективні обчислення під час аналізу даних пакету MatLab, програма MathCad – це, імовірніше, редактор математичних текстів із широкими можливостями символьних обчислень і прекрасним інтерфейсом. MathCad не має мови програмування як такої, а можливості символьних обчислень запозичені з пакету Maple. Проте інтер-

фейс програми MathCad дуже простий, а можливості візуалізації широкі. Всі обчислення тут здійснюються на рівні візуального запису виразів у загальновживаній математичній формі. Пакет має зрозумілі підказки, докладну документацію, цілу низку додаткових модулів та вбудованих функцій. Однак поки математичні можливості MathCad поступаються системам Maple, Mathematica та MatLab. Незважаючи на це, за програмою MathCad випущено багато книг і навчальних курсів. Сьогодні ця система стала буквально міжнародним стандартом для технічних обчислень. Розробники MathCad зробили все можливе, щоб користувач, який не володіє спеціальними знаннями у програмуванні, міг реалізувати велику кількість обчислювальних методів та досягнути значного результату в області математичних розрахунків.

Звичайно, кожен з математичних пакетів має свої переваги та недоліки та є зручним для розв'язання конкретних завдань. Наведемо практичні задачі, які можна розв'язувати з допомогою потужного сервісу **Maple,** вже згаданого вище. Нехай поставлена задача апроксимації таблично заданої функції:

x	–2	–1	0	1	2	3	4	5	6	7	8	9
y	–5	–3	–1	0	2	4	6	8	7	2	–3	–11

Програмний код Maple має такий вигляд:

```
>with(CurveFitting):
>xydata:=[[-2,-5],[-1,-3],[0,-1,[1,0],[2,2],[3,4],[4,6],[5,8],[6,7],[7,2],[8,-3],
[9,-11]]
xydata := [[-2, -5], [-1, -3], [0, -1], [1, 0], [2, 2], [3, 4], [4, 6], [5, 8], [6, 7],
[7, 2], [8, -3], [9, -11]]
c1:=BSplineCurve(xydata, v)
```

Далі задана таблично функція інтерполюється кубічними сплайнами. Сплайном називається кусочно-поліноміальна функція, визначена на відрізку (x_1, x_2) і має на цьому відрізку не менше двох неперервних похідних. Інтерполяція з допомогою сплайнів називається сплайн-інтерполяцією. Приклад сплайн-інтерполяції кубічного типу показано далі (Лістинг 2.1).

Внаслідок процедури моделювання отримаємо графік, представлений на рис. 2.3. Показано модельовану кусково-подібну ламану криву (голубий колір) та моделюючу згладжену лінію коричневого кольору. Моделювання здійснюється, як уже згадувалось, за допомогою так званих сплайнів. У нашому випадку використовуються кубічні сплайни.

Лістинг 2.1

$$\begin{cases} 0 & v < 0 \\ -\frac{v^3}{3} & v < 1 \\ \frac{2}{3} - v - (v-1)^2 + \frac{5\,(v-1)^3}{6} & v < 2 \\ -\frac{1}{2} + \frac{3\,(v-2)^2}{2} - \frac{(v-2)^3}{2} - \frac{v}{2} & v < 3 \\ v - 4 & v < 12 \\ -4 + v - \frac{5\,(v-12)^3}{3} & v < 13 \\ \frac{178}{3} - 4\,v - 5\,(v-13)^2 + \frac{19\,(v-13)^3}{6} & v < 14 \\ \frac{129}{2} - \frac{9\,v}{2} + \frac{9\,(v-14)^2}{2} - \frac{3\,(v-14)^3}{2} & v < 15 \\ 0 & 15 \le v \end{cases}$$

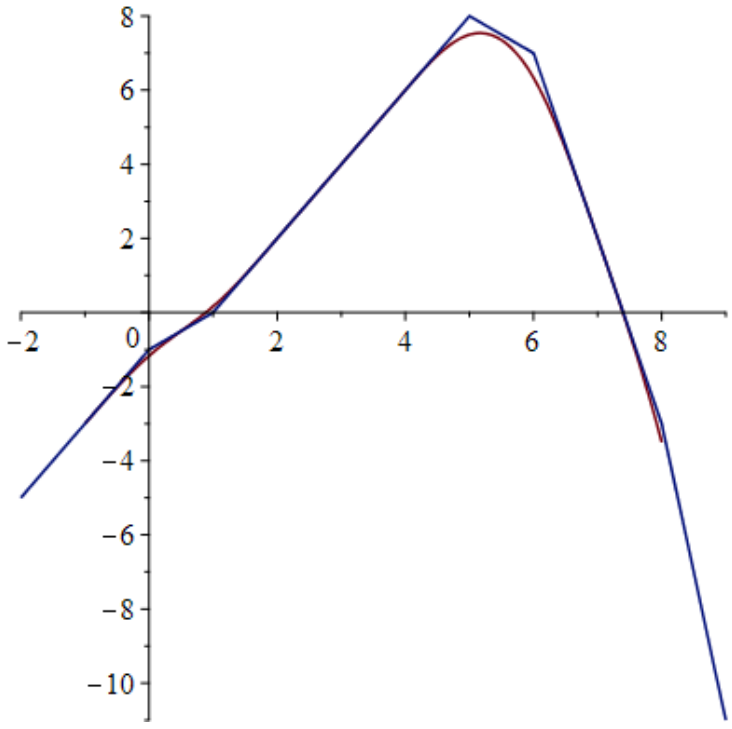


Рис. 2.3. Моделювання функції, заданої таблично, кубічними сплайнами

Далі наведемо просту програму Maple, що дає змогу побудувати графік функції $y=2\cos^2 t-4/5$ (рис. 2.4):

```
> with(plots):
> polarplot(2*cos(t)*cos(t)-4/5, t= -Pi..Pi)
```

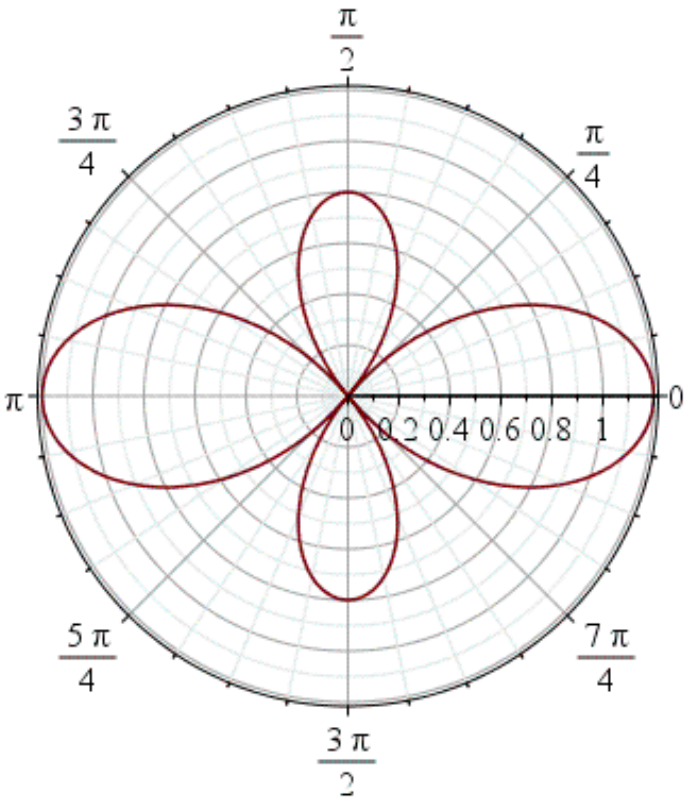


Рис. 2.4. Графік функції $y=2\cos^2 t-4/5$, побудований з допомогою Maple

Загалом сервіс Maple володіє досить широким спектром варіантів математичного моделювання.

2.4. Моделювання даних

Одним із важливих напрямів моделювання є моделювання даних. Завдання такого моделювання – створення візуального представлення про інформаційну систему. Потрібно проілюструвати типи даних, які застосовуються під час функціонування системи як цілісного об’єкта, та проаналізувати відношення між такими типами даних. Важливо також сформувати засоби групування та організації даних, їх формати та атрибути.

Для чого потрібні моделі такого типу? В основному для реалізації потреб бізнес-структур та різноманітних підприємств. Правила та вимоги до моделей даних формуються на основі зворотного зв’язку з бізнес-структурами. Отже, генератором для модернізації моделей даних є надійний зворотний зв’язок типу «бізнес ↔ моделі даних».

Глибина моделювання даних залежить від потреб бізнесу. Зрозуміло, що моделювати можна на різних рівнях абстракції. На першому етапі моделювання даних необхідно встановити тісний зв’язок типу «бізнес ↔ моделі даних».

Мета – зібрати максимум інформації про бізнес-потреби, тісно співпрацюючи зі всіма зацікавленими сторонами. Такого типу бізнес-алгоритми інсталюються у структуру даних. Модифіковану модель даних можна порівняти з мапою, планом архітектора або будь-якою електричною схемою, яка дає змогу зрозуміти принцип роботи складного пристрою чи системи загалом.

Ефективність моделі даних базується на тих нормативних документах, які змінюються синхронно з потребами бізнесу. Такого типу документи сфокусовані на підтримку бізнес-процесів та сприяють плануванню IT-архітектури та стратегії розвитку. Моделі даних слугують загальною базою для взаємодії між постачальниками, партнерами та бізнес-аналітиками. Загалом моделювання в якості стандартних інструментів використовує загальні схеми та алгоритмізовані методи. Це дає змогу керувати базами даних як у певній бізнес-структурі, так і поза її межами.

У чому полягають переваги моделювання даними? Такий спосіб оперування набором різноманітних даних сприяє розумінню взаємозв'язків між різними даними для розробників, архітекторів бізнес-структур та бізнес-аналітиків. Загалом моделювання дає змогу:

- модифікувати відображення даних в усій бізнес-структурі;
- покращити співпрацю між розробниками та бізнесом;
- мінімізувати помилки під час розробки програмного забезпечення та баз даних;
- стандартизувати бізнес-документацію на підприємстві;
- максимізувати ефективність програмних додатків і баз даних;
- уніфікувати та спростити процеси створення баз даних на аналітичному, логічному та структурному рівнях.

Інформаційні системи та бази даних потрібно починати розробляти, абстрагуючись від конкретних деталей. Проте крок за кроком процес абстрагування стає все більш деталізованим, а тому – більш точним і практичним. Рівні абстракції моделі даних умовно можна розділити на три види. Зазвичай процедура створення моделі бази даних починається з концептуальної моделі, яка згодом трансформується у логічну модель, і нарешті процес завершується робочою фізичною моделлю.

Концептуальні моделі даних називаються моделями предметної області та описують загальну картину: що буде містити система, як вона буде організована та які бізнес-правила будуть задіяні. Концептуальні моделі зазвичай створюються в процесі збору вихідних вимог до проєкту. Зазвичай, вони включають класи сутностей (речі, які бізнесу важливо представити в моделі даних), їх характеристики та обмеження, відношення між сутностями, вимоги до безпеки та цілісності даних. У загальному випадку концептуальна модель даних може

бути представлена, як показано на рис. 2.5. Така модель включає у себе чотири важливі компоненти – продукт (product), продаж (sales), зберігання (store) та час (time). Включення останнього компоненту є важливим моментом, оскільки час визначає динаміку всіх бізнес-процесів у реальній економічній структурі.

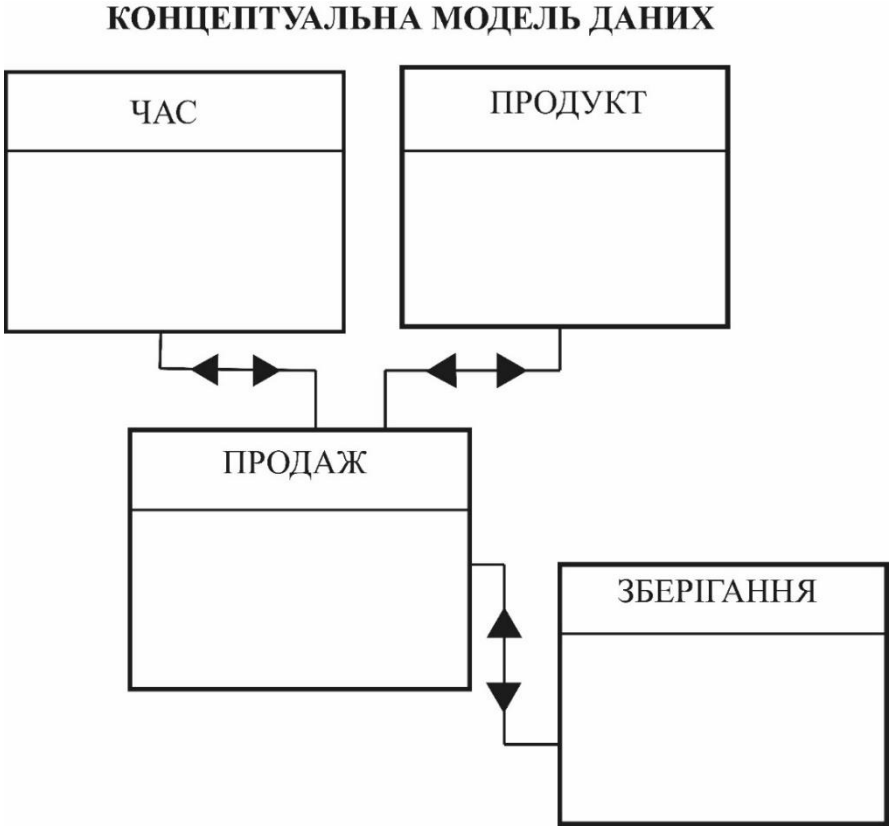


Рис. 2.5. Концептуальна модель даних

Зараз потрібно перейти до аналізу завершальної процедури моделювання даних. Такою завершальною віхою є фізичні моделі даних. У них вирішується питання того, яка реалізується модель зберігання даних у базі. Фізичні моделі даних є найбільш адаптовані до вирішення реальних проблем, і тому найменш абстрактні з усіх моделей. Ці моделі пропонують остаточний варіант, який можна реально реалізувати, наприклад, у вигляді реляційної бази даних. Така база включає в себе асоціативні таблиці, які демонструють як відношення між сутностями, так і первинні та зовнішні ключі для зв'язку даних. Фізичну модель даних і її структуру можна представити, як показано на рис. 2.6. Тут фактично концептуальна модель даних (рис. 2.5) наповнюється реальним змістом.

Процедура моделювання даних стартує із постановки задачі моделювання. Дуже бажано, щоб цю задачу сформулював сам замовник. Насамперед потрібно стандартизувати набір позначень та символів, які будуть використовуватися для представлення даних. Потрібно також узгодити спосіб розміщення моделі та процедуру передачі бізнес-пропозицій. Такий спеціалізований робочий

процес має також включати конкретний перелік завдань, які повинен розробити виконавець. Процес розробки загалом має ітеративну форму, що передбачає послідовну модифікацію розробки моделі даних. Поетапні стадії процесу можна представити в такому вигляді:

Рис. 2.6. Взаємодія між компонентами фізичної моделі даних

1. На першому етапі відбувається конкретизація концепції моделювання даних з формулюванням конкретних завдань та постановки цілей розробникам.

2. На цьому етапі встановлюються головні властивості кожного складника моделі. Кожна така сутність відрізняється від інших, оскільки має певну кількість унікальних властивостей. Ці властивості називаються атрибутами. Наприклад, сутність «клієнт» володіє такими атрибутами: прізвище, ім’я, місце проживання, номер телефону тощо. Сутність «адреса» включає в себе назву вулиці та номер будинку чи квартири, країну, місто та поштовий індекс.

3. Далі встановлюється взаємозв’язок між сутностями. Нульовий варіант моделі даних визначатиме спосіб відношень, які кожна сутність встановлює з іншими сутностями. Зрозуміло, що кожен клієнт живе за певною адресою. Зрозуміло, що модель має передбачати сутність «замовлення», яке, природно, пов’язане з адресом. Дуже зручним способом документування подібних відношень є уніфікована мова моделювання (UML).

4. Дуже важливо співставити атрибути з сутностями. Це дає гарантію того, що модель відображає спосіб використання даних бізнесом. Можна ефективно

застосовувати кілька формальних еталонів (патернів) моделювання даних. Розробники часто застосовують патерни проєктування.

5. Важливо встановити рівень нормалізації. Загалом під терміном нормалізація потрібно розуміти спосіб організації моделей даних так, що ідентифікатори (ключі) призначаються групам даних. Зокрема, якщо клієнту призначено ключ, тоді його можна пов'зати з адресою клієнта та з історією замовлень. Водночас немає потреби дублювати цю інформацію у таблиці з іменами клієнтів. Отже, нормалізація допомагає зменшити об'єм пам'яті на диску, де зберігається база даних. Все вищеозначене може суттєво впливати на швидкість виконання запитів.

6. На завершальному етапі модель даних потрібно протестувати у різних режимах роботи. Нагадаємо ще раз, що моделювання даних – це ітеративний творчий процес, який потрібно постійно повторювати та адаптувати під потреби замовника.

Моделювання даних удосконалювалося паралельно із розвитком системами управління базами даних (СУБД). Звернемо увагу на те, що моделі даних формують певну ієрархічну структуру. Їх у певному сенсі можна порівняти із графами типу «дерево», де чітко прослідковується саме ієрархічна структура. У моделях такого типу кожен запис має єдиний корінь вищого рівня, який зіставляється з кількома дочірніми таблицями. Модель подібного типу була розроблена компанією Information Management System (IMS). Така модель широко застосовується, зокрема, у банківській сфері. Цей модельний підхід не є найоптимальнішим. Розроблені сьогодні моделі баз даних є більш ефективними, проте IMS також ефективно використовується.

Дослідник з компанії IBM Ф. Кодд запропонував реляційні моделі даних, які досить часто зустрічаються у деяких базах даних. Зазвичай такі бази часто використовуються в корпоративних обчисленнях. Реляційне моделювання не вимагає глибокого розуміння фізичних особливостей використовуваного сховища даних. У ньому сегменти даних поєднуються за допомогою таблиць, що спрощує базу даних.

Реляційні бази даних часто використовують мову структурованих запитів (SQL) з метою керування даними. Ці бази використовуються, зокрема, у касових системах а також інших видах обробки транзакцій.

Моделі даних типу «сутність-зв'язок» (ER-модель) використовують діаграми представлення зв'язків між сутностями у базі даних. ER-модель є формальною конструкцією, яка не використовує конкретних графічних засобів її візуального представлення. Графічно таку модель можна представити за допомогою діаграми «сутність-зв'язок» (Entity-Relationship diagram). ER-модель дає змогу описувати концептуальні схеми за допомогою узагальнених конструкцій блоків.

Одним із найзручніших інструментів уніфікованого представлення даних, незалежного від програмного забезпечення, що його реалізує, є саме ER-модель. Важливим є той факт, що з такої моделі можуть бути породжені всі наявні моделі даних (ієрархічна, мережева, реляційна, об'єктна), тому вона є найзагальнішою.

Об'ктно-орієнтовані моделі даних почали інтенсивно розвиватися з середини 90-х років 20-го сторіччя. В якості об'єктів виступають абстрактні сутності реального світу. Об'єкти згруповані в класи та мають подібні особливості. Об'єктно-орієнтовані бази даних містять таблиці, але загалом підтримують і більш складні зв'язки. Підхід такого типу використовується, зокрема, у мультимедійних та гіпертекстових базах даних.

Ральф Кімбол розробив розмірні моделі даних з метою швидкого пошуку інформації у сховищі. Загалом реляційні і ER-моделі спрямовані на ефективне зберігання даних та оптимізують масиви даних. Дві широко розповсюджені розмірні моделі даних – це схеми «зірка» та «сніжинка». У схемі «зірка» дані організовані у факти та виміри, де кожен факт оточений пов'язаним з ним виміром у вигляді зірочки. Схема «сніжинка» нагадує схему «зірка», але включає додаткові шари пов'язаних вимірювань, що ускладнює схему розгалуження.

Зупинимось тепер на інструментах для моделювання даних. Сьогодні інтенсивно застосовуються численні комерційні та CASE-рішення, а також різні інструменти моделювання даних, побудови діаграм та візуалізації. Розглянемо коротко особливості деяких систем проєктування даних:

- ER/Studio – програма для проєктування баз даних, яку можна сумістити з відомими СУБД. Вона підтримує як реляційне, та і розмірне моделювання даних;

- Enterprise Architect – досконалий інструмент візуального моделювання, який підтримує моделювання архітектур, програмних програм та баз даних корпоративних інформаційних систем;

- Erwin Data Modeler – відомий інструмент моделювання даних, що використовує мову IDEF1X;

- Open ModelSphere, ChenToolkit або EasyCASE – безкоштовні інструменти моделювання даних.

Для того, щоб адаптувати дані до структури, що відповідає вимогам моделі, потрібно скористатись вмонтованим механізмом регулярних запитів. Такі запити виконуються в Scheduled Queries, AppScript Google або BigQuery.

Розроблені спеціалізовані інструменти для управління SQL-запитами, якот data build tool (dbt) і Dataform. Перший із названих інструментів (dbt) являє собою фреймворк із відкритим вихідним кодом для виконання, тестування та документування SQL-запитів. Цей фреймворк допомагає оптимізувати роботу

з SQL-запитами: використовувати макроси та шаблони JINJA, щоб не дублювати повторювані фрагменти коду.

2.5. Клітинні автомати

Клітинний автомат (КА) являє собою набір комірок, упорядкованих у сітку заданої форми, так що кожна комірка змінює стан як функцію часу відповідно до визначеного набору правил, керованих станами сусідніх комірок. Кожна клітина знаходиться в одному зі скінченної кількості станів. Ця обчислювальна модель одночасно є абстрактною та дискретною у просторі та часі. Еволюція КА відбувається на основі набору правил, заснованих на станах сусідніх комірок.

Комірки в КА знаходяться на сітці, яка має певну форму (квадрат, трикутник, шестикутник тощо). Кожна клітинка на сітці має стан. Хоча існує багато варіантів, що описують стан клітинки, найпростішою формою є стани типу увімкнено / вимкнено, істина / хиба або 1 / 0.

Клітини, що прилягають до певної вибраної комірки, складають її околиці. Сусідні клітини мають властивість впливати одна на одну. А оскільки всі клітини в КА мають сусідів, то зміна стану однієї клітини спричиняє хвилеподібний процес, що являє собою передачу збудження від однієї комірки до іншої.

Загалом КА мають спектр різновидів. Найпростіший КА є одновимірним, з клітинками на прямій лінії, де кожна клітинка може мати лише два можливі стани (наприклад, високий / низький, чорний / білий або true / false). Однак можливі й інші форми. У двовимірному просторі поширеними формами клітин є квадрати та шестикутники.

Взагалі КА може мати будь-яку кількість вимірів, і кожна комірка може мати будь-яку кількість можливих станів. Стан кожної клітини змінюється дискретними кроками через рівні проміжки часу. У будь-який момент часу цей стан залежить від такого: власний стан на попередньому часовому кроці та стани своїх безпосередніх сусідів на попередньому часовому кроці. Коли переглядається графічне відтворення КА, воно виглядає як «квантований» анімований об’єкт.

Отже, КА може бути побудований у довільній кількості вимірів. Спочатку модель КА було запропоновано для використання в криптографії з відкритим ключем, у географії, антропології, політології, соціології та фізиці.

Особливістю КА є їх здатність еволюціонувати (змінювати свій стан) відповідно до стану сусідніх клітин і певних правил, які залежать від принципів моделювання.

Модель КА була запропонована американськими математиками Джоном фон Нейманом і Станіславом Уламом. Найвідоміший клітинний автомат «Гра в життя» Джона Конвея моделює процеси життя, смерті та динаміку популяції.

Існує багато типів КА. Найпростішим типом є двійковий одновимірний автомат. Припустимо, що у цій моделі клітинки можуть мати два можливі значення: 0 або 1. Цей КА можна описати за допомогою таблиці, яка визначає стан клітинки в наступному поколінні на основі значення: *клітинка ліворуч від вибраної, сама клітинка, клітинка праворуч*. Існує вісім можливих двійкових станів для трьох комірок, які знаходяться поруч із певною коміркою. Перерахуємо ці стани: <0,0,0>, <1,0,0>, <0,0,1>, <1,0,1>, <0,1,0>, <1,1,0>, <0,1,1>, <1,1,1>. У випадку КА, що складається із восьми клітинок, спектр можливих станів розраховується за формулою комбінаторики (2.2), що описує розміщення з повтореннями:

$$\widetilde{A_n^r} = n^r, \tag{2.2}$$

де n – кількість елементів, які розміщуються за заданими r позиціями. В нашому випадку $n = 2$, а $r = 3$. Якщо, наприклад, маємо ситуацію, як показано на рис. 2.7, то отримаємо кількість можливих станів КА, рівне $2^8 = 256$.

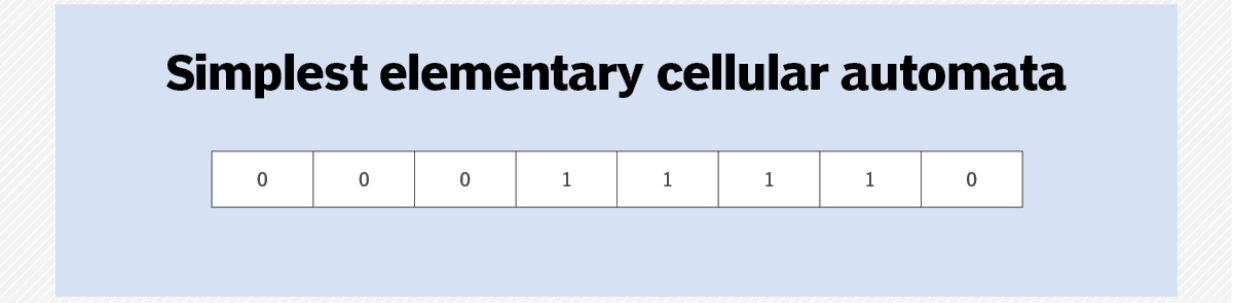


Рис. 2.7. Двійковий одновимірний автомат, що складається з восьми комірок

Найбільш відомою інтерпретацією КА є ГРА ЖИТТЯ (GAME Of LIFE). Для наочної інтерпретації цієї гри створимо таку модель. Нехай маємо сітку клітин розміром 10×10, які можуть бути живими або мертвими (рис. 2.8).

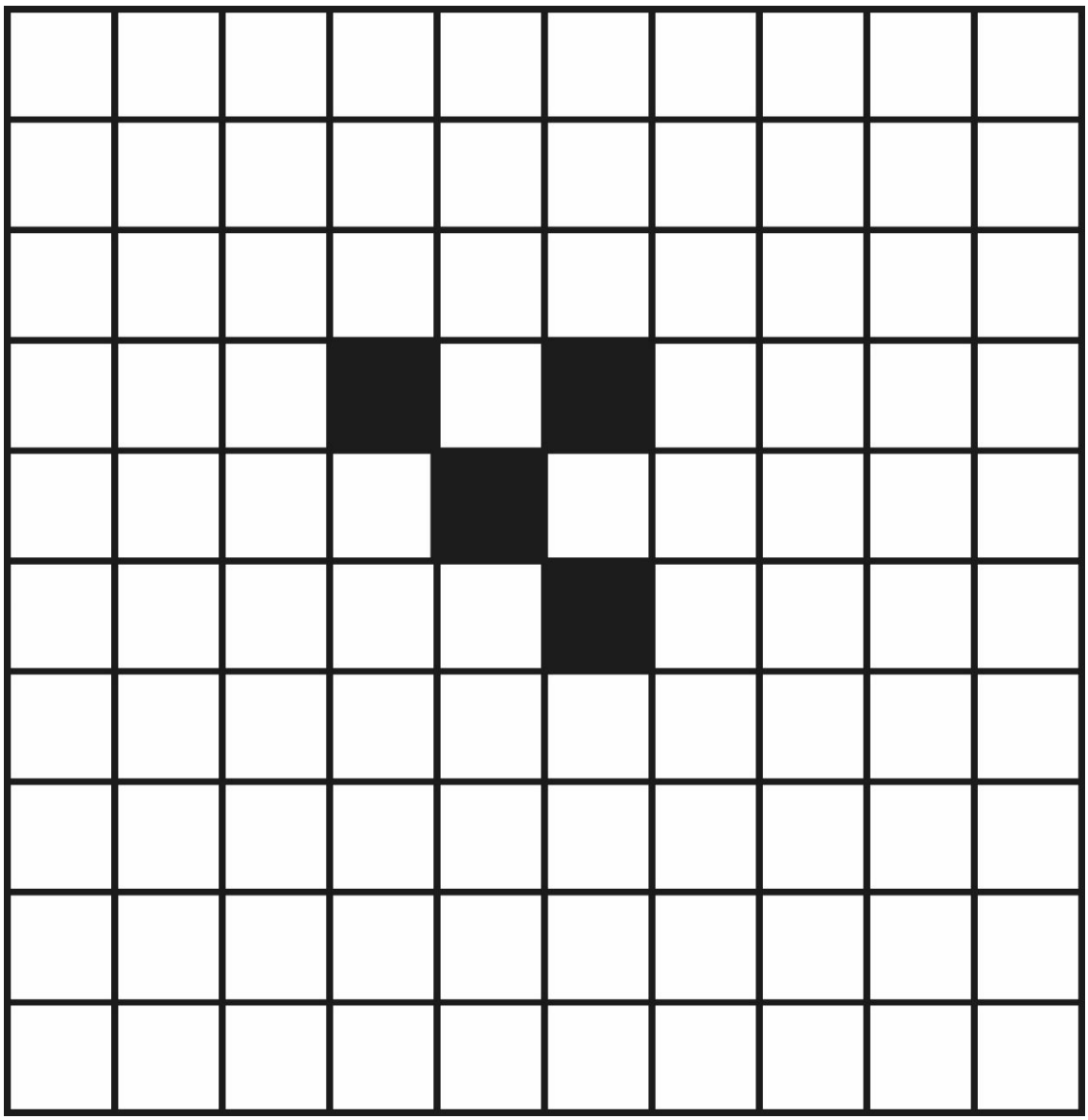


Рис. 2.8. Двовимірний автомат розміру 10×10

Завдання полягає у тому, щоб створити нове покоління клітин на основі таких ПРАВИЛ:

1) жива клітина з менш ніж двома живими сусідами гине, оскільки це викликано недостатнім населенням;

2) жива клітина з двома-трьома живими сусідами живе до наступного покоління;

3) жива клітина з більш ніж трьома живими сусідами гине, оскільки наявне перенаселення;

4) мертва клітина з рівно трьома живими сусідами стає живою клітиною, оскільки як відбувається процес розмноження.

Нехай у наступній схемі «**1**» означає живу клітинку, а «●» – мертву (рис. 2.9). Початкове розташування живих та мертвих клітин задається ВХІДНИМИ ДАНИМИ. Нове покоління клітин буде представлено, як зображено

на діаграмі ВИХІДНІ ДАНІ. Народження нової клітини на діаграмі ВИХІДНІ ДАНІ обумовлено дією пункту 4 наведених вище правил.

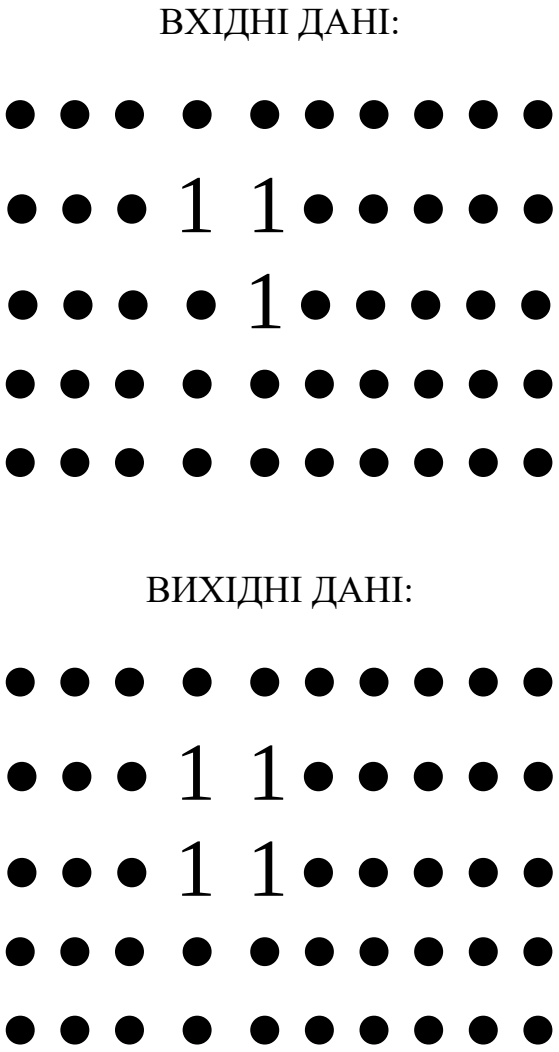


Рис. 2.9. Народження нової клітини у позиції (3,4), де мертва клітина стає живою згідно з пунктом 4 наведених вище ПРАВИЛ

Представимо далі програмну реалізацію Game Of Life на Java (Лістинг 2.2). Сітка, що являє собою двомірну матрицю, ініціалізується нулями та одиницями. Перші представляють мертві клітини, а другі – живі. Функція generate() проходить по кожній комірці та підраховує її сусідів. На основі цих значень реалізуються вищезгадані правила. Наступна реалізація ігнорує крайові комірки, оскільки має відтворюватися на нескінченній площині.

Лістинг 2.2

```
// клас, що реалізує гру Game of Life
public class GameOfLife
```

```
{
    public static void main(String[] args)
    {
        int M = 10, N = 10;
        // ініціалізація решітки
        int[][] grid = {
                { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
                { 0, 0, 1, 1, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0 }
        };
        // Displaying the grid
        System.out.println("Стартова генерація");
        for (int i = 0; i < M; i++)
        {
            for (int j = 0; j < N; j++)
            {
                if (grid[i][j] == 0)
                    System.out.print("  .  ");
                else
                    System.out.print("  1  ");
            }
            System.out.println();
        }
        System.out.println();
        nextGeneration(grid, M, N);
    }
    //наступна генерація
    static void nextGeneration(int grid[][], int M, int N)
    {
        int[][] future = new int[M][N];

        // цикл, що переглядає кожну комірку
```

```
for (int l = 0; l < M; l++)
{
   for (int m = 0; m < N; m++)
   {
      // не знайдено жодного живого сусіда
      int aliveNeighbours = 0;
      for (int i = -1; i <= 1; i++)
         for (int j = -1; j <= 1; j++)
            if ((l+i>=0 && l+i<M) && (m+j>=0 && m+j<N))
               aliveNeighbours += grid[l + i][m + j];
      //  клітинку потрібно видалити
      aliveNeighbours -= grid[l][m];
      //застосування правил життя
                  if ((grid[l][m] == 1) && (aliveNeighbours < 2))
         future[l][m] = 0;
         //комірка гине через перенаселення
      else if ((grid[l][m] == 1) && (aliveNeighbours > 3))
         future[l][m] = 0;
         // нова комірка народжується
      else if ((grid[l][m] == 0) && (aliveNeighbours == 3))
         future[l][m] = 1;
         // залишається незмінною
      else
         future[l][m] = grid[l][m];
   }
}
System.out.println("Наступна генерація");
for (int i = 0; i < M; i++)
{
   for (int j = 0; j < N; j++)
   {
      if (future[i][j] == 0)
         System.out.print("  .   ");
      else
         System.out.print("  1   ");
   }
   System.out.println();
}   }  }
```

Результат роботи програми виглядає так (рис. 2.10 та 2.11):

Стартова генерація

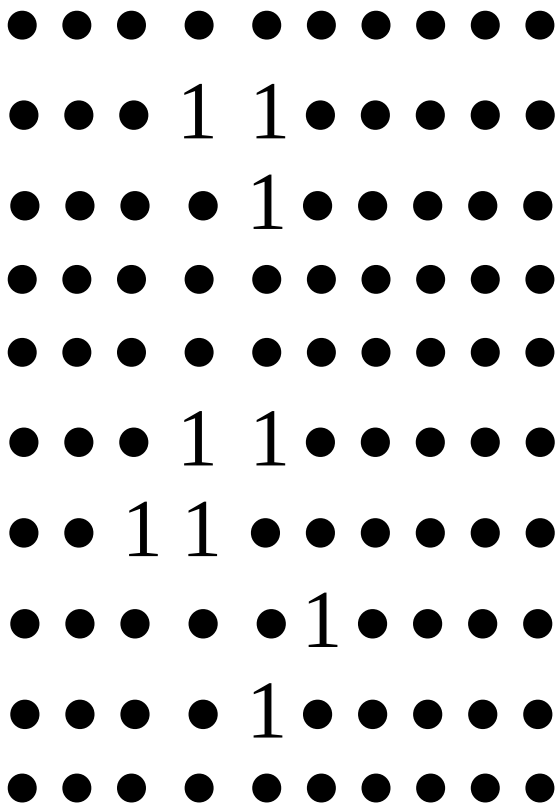


Рис. 2.10. Стартове розташування живих та мертвих клітин

Наступна генерація

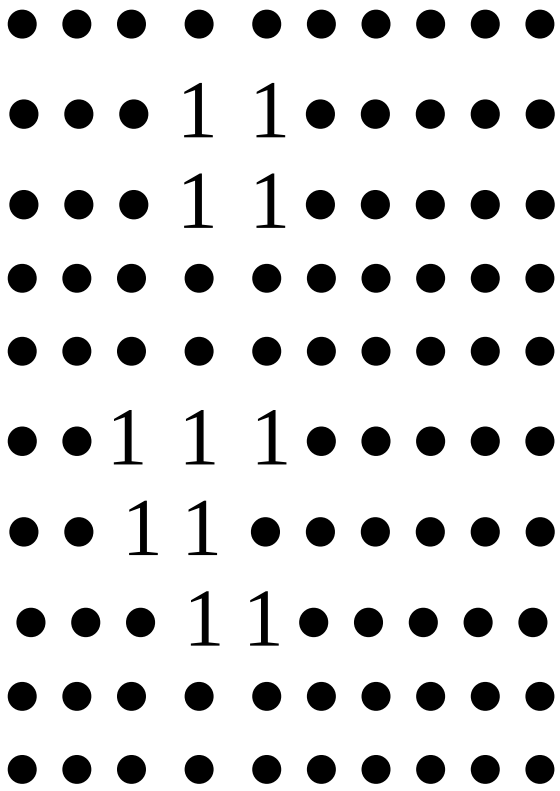


Рис. 2.11. Розташування живих та мертвих клітин у наступній генерації

Розташування живих та мертвих клітин, представлене на рис. 2.11, слід розуміти так. Поява живої клітини в позиції (3,4) обумовлена дією пункту 4 ПРАВИЛ. Так само трансформація «мертва клітина → жива клітина» у позиціях (6,3), (8,4) та (8,5) є результатом дії того самого пункту. Навпаки, трансформація «жива клітина → мертва клітина» для клітин з координатами (8,6) та (9,5) обумовлена дією пункту 1 ПРАВИЛ (рис. 2.10).

Спочатку гра Game Of Life розглядала біологічні аспекти, тобто процеси життєдіяльності клітин, але згодом вона почала застосовуватись у різних сферах, як-от графіка, картографія тощо.

Застосуємо тепер програму Лістингу 2.2 для ситуації, зображеної на рис. 2.8. Для цього у програмі стартову матрицю представимо так (рис. 2.12):

```
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 1, 0, 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, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Рис. 2.12. Матриця, що демонструє розташування живих та мертвих клітин у програмі Лістинг 2.2: 1 – жива клітина, 0 – мертва клітина

Запустимо програму та отримаємо таке розташування живих та мертвих клітин, як у стартовій позиції (рис. 2.13), так і у наступній генерації клітин (рис. 2.14).

Стартова генерація

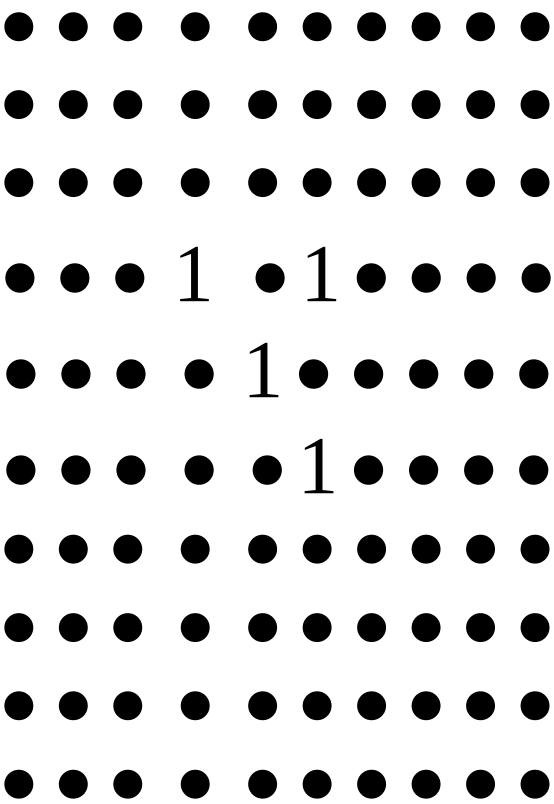


Рис. 2.13. Розташування живих та мертвих клітин у стартовій позиції

Наступна генерація

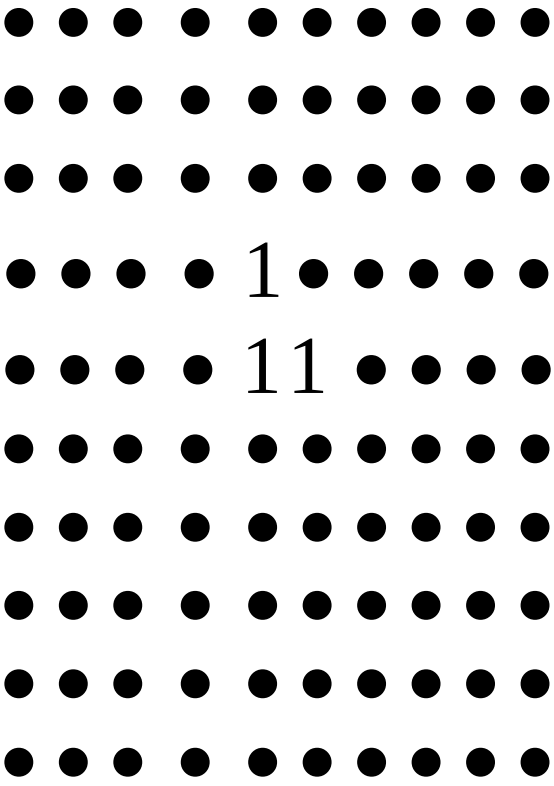


Рис. 2.14. Розташування живих та мертвих клітин у наступній генерації

Проаналізуємо отриманий результат. Клітина з координатами (4,4) гине, згідно з пунктом 1 ПРАВИЛ (рис. 2.13). Так само гине клітина з координатами (4,6). Аналогічно зникає клітина з координатами (6,6). Навпаки, клітина з координатами (5,5) залишається живою, оскільки вона має трьох живих сусідів. Мертві клітини з координатами (4,5) та (5,6) народжуються згідно з пунктом 4 ПРАВИЛ. Внаслідок цього отримуємо картину, представлену на рис. 2.14.

Ознайомитись із особливостями гри Game Of Life можна за посиланням https://playgameoflife.com/

РОЗДІЛ 3
КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ

3.1. Види і особливості комп'ютерного моделювання

Комп'ютерне моделювання – процес створення комп'ютерної моделі на одному або декількох обчислювальних вузлах. Цей процес реалізує представлення об'єкта, системи чи поняття у формі, відмінній від реальної, але наближеній до алгоритмічного опису, та включає набір даних, що характеризують властивості системи та динаміку їх зміни з часом.

Представимо загальну класифікацію видів комп'ютерного моделювання (рис. 3.1).

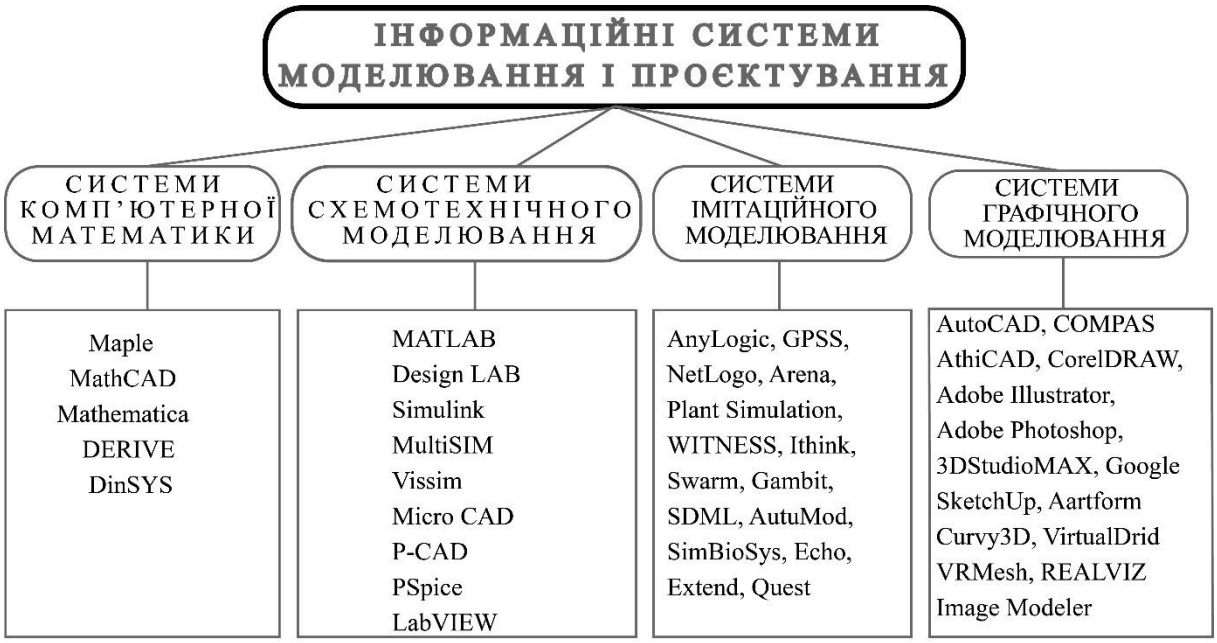


Рис. 3.1. Комп'ютерні інструментальні засоби моделювання

На рис. 3.1 представлено далеко не повний перелік засобів інструментального комп'ютерного моделювання. Сьогодні спектр таких засобів постійно розширюється та оновлюється новими інструментами. Серед наведених засобів моделювання виділимо такі широко відомі:

- AnyLogic North America LLC (далі – AnyLogic) включає в себе графічну мову моделювання, можливості якої можуть бути розширені з використанням мови програмування Java. Поєднання компілятора Java в AnyLogic із можливостями програми створює спектр нових можливостей під час реалізації різного

роду моделей. Загалом ми маємо справу з комплексним інструментом візуального комп'ютерного моделювання;

- GPSS – система імітаційного моделювання, що використовується переважно для моделювання систем масового обслуговування.

Комп'ютерні моделі стали звичним інструментом математичного моделювання і застосовуються у фізиці, астрофізиці, механіці, хімії, біології, економіці, соціології, метеорології, інших науках і прикладних задачах та у різних галузях радіоелектроніки, машинобудування, автомобілебудування тощо. Комп'ютерні моделі використовуються для отримання нових знань про об'єкт або для наближеної оцінки поведінки систем, занадто складних для аналітичного дослідження.

Комп'ютерне моделювання є одним із ефективних методів вивчення складних систем. Комп'ютерні моделі простіше і зручніше досліджувати через можливість проводити обчислювальні експерименти в тих випадках, коли реальні експерименти ускладнені через фінансові або фізичні перешкоди або можуть дати неконтрольований результат. Формалізованість комп'ютерних моделей дає змогу визначити основні чинники, що визначають властивості досліджуваного об'єкта, зокрема дають змогу досліджувати відгук модельованої фізичної системи на зміни її параметрів і початкових умов.

Побудова комп'ютерної моделі, як і будь-якої іншої, базується на абстрагуванні від конкретної природи явищ або досліджуваного об'єкта і складається з двох етапів – створення якісної, а потім і кількісної моделі. Чим більше значущих властивостей буде виявлено і перенесено на комп'ютерну модель, тим більш наближеною вона виявиться до реальної системи. Водночас система, що досліджується, стане більш зрозумілою і матиме більші можливості. Комп'ютерне моделювання полягає у проведенні серії обчислювальних експериментів на комп'ютері, метою яких є аналіз, інтерпретація та зіставлення результатів моделювання з реальною поведінкою досліджуваного об'єкта і, за необхідності, подальше уточнення моделі.

Розрізняють аналітичне та імітаційне моделювання. Під час аналітичного моделювання вивчають математичні (абстрактні) моделі реального об'єкта у вигляді алгебраїчних, диференціальних та інших рівнянь, а також тих, що передбачають здійснення однозначної обчислювальної процедури, яка приводить до їхнього точного розв'язання. Під час імітаційного моделювання досліджуються математичні моделі у вигляді алгоритму, що відтворює функціонування досліджуваної системи шляхом послідовного виконання великої кількості елементарних операцій.

Комп'ютерне моделювання дає змогу:

- розширити коло дослідницьких об'єктів і зробити можливим вивчення явищ, що не повторюються, та об'єктів, які не відтворюються в реальних умовах;

- візуалізувати об’єкти будь-якої природи, зокрема й абстрактні;
- досліджувати явища і процеси в динаміці їх розгортання;
- керувати часом (прискорювати, сповільнювати тощо);
- здійснювати багаторазові випробування моделі, щоразу повертаючи її в первинний стан;
- отримувати різні характеристики об’єкта в числовому або графічному вигляді;
- знаходити оптимальну конструкцію об’єкта, не виготовляючи його пробних екземплярів;
- проводити експерименти без ризику негативних наслідків для здоров’я людини або навколишнього середовища.

Розглянемо та проаналізуємо основні етапи комп’ютерного моделювання:

1. Постановка завдання та його аналіз.

1.1. З’ясувати, з якою метою створюється модель.

1.2. Уточнити, які вихідні результати і у якому вигляді слід отримати.

1.3. Визначити, які вихідні дані потрібні для створення моделі.

2. Побудова інформаційної моделі

2.1. Визначити параметри моделі та виявити взаємозв’язок між ними.

2.2. Оцінити, які з параметрів впливові, а якими можна нехтувати.

2.3. Математично описати залежність між параметрами моделі.

3. Розробка методики та алгоритму реалізації комп’ютерної моделі

3.1. Обрати або розробити метод отримання вихідних результатів.

3.2. Скласти алгоритм отримання результатів за обраними методами.

3.3. Перевірити правильність алгоритму.

4. Розробка комп’ютерної моделі

4.1. Вибрати засоби програмної реалізації алгоритму на комп’ютері.

4.2. Розробити комп’ютерну модель.

4.3. Перевірити правильність створеної комп’ютерної моделі.

5. Проведення експерименту

5.1. Розробити план дослідження.

5.2. Провести експеримент на базі створеної комп’ютерної моделі.

5.3. Проаналізувати отримані результати.

5.4. Зробити висновки щодо властивостей прототипу моделі.

У процесі проведення експерименту може з’ясуватися, що потрібно:

- скоригувати план дослідження;
- обрати інший метод розв’язання задачі;
- удосконалити алгоритм отримання результатів;
- уточнити інформаційну модель;
- внести зміни у постановку задачі.

У такому разі відбувається повернення до відповідного етапу і процес починається знову.

Комп'ютерне моделювання застосовують для широкого кола завдань, як-от:

- аналіз поширення забруднювальних речовин в атмосфері;
- проєктування шумових бар'єрів для боротьби з шумовим забрудненням;
- конструювання транспортних засобів;
- симуляція польоту на авіаційному тренажері для тренування льотчиків;
- прогнозування погоди;
- емуляція роботи інших електронних пристроїв;
- прогнозування цін на фінансових ринках;
- дослідження поведінки будівель, конструкцій і деталей під механічним навантаженням;
- прогнозування міцності конструкцій і механізмів їх руйнування;
- проєктування виробничих процесів, наприклад, хімічних;
- стратегічне управління організацією;
- дослідження поведінки гідравлічних систем: нафтопроводів, водопроводів;
- моделювання роботів і автоматичних маніпуляторів;
- моделювання варіантів розвитку міст;
- моделювання транспортних систем;
- моделювання краш-тестів;
- моделювання результатів пластичних операцій.

Різні сфери застосування комп'ютерних моделей висувають різні вимоги до надійності одержуваних за їх допомогою результатів. Наприклад, для моделювання будівель і деталей літаків потрібна висока точність і ступінь достовірності, тоді як моделі еволюції міст і соціально-економічних систем використовуються для отримання наближених або якісних результатів.

Алгоритми комп'ютерного моделювання:

- метод скінченних елементів;
- метод скінченних різниць;
- метод скінченних об'ємів;
- метод рухомих клітинних автоматів;
- метод класичної молекулярної динаміки;
- метод компонентних ланцюгів;
- метод вузлових потенціалів.

3.2. Варіанти застосування комп'ютерного моделювання

Інформаційні технології значною мірою змінили життя людей, тому майже неможливо уявити будь-яку діяльність, яка б не залежала від комп'ютерів. Як

тільки з'явилися перші комп'ютерні системи, люди намагалися скористатися перевагами комп'ютерів для вирішення складних завдань у різних сферах. Із розвитком промисловості зростала потреба в комп'ютерах, і тому обчислювальні програми з часом еволюціонували. На зміну традиційним методам конструювання прийшли комп'ютерні програми, які мають можливість прогнозувати поведінку конструкцій за різних умов навантаження. Отже, дорогі експерименти, випробування та обстеження замінюються дешевшими і потужнішими обчислювальними методами, які не потребують руйнування самої конструкції для визначення її стійкості.

Комп'ютерні програми допомагають у вирішенні різного роду проблем. Спочатку проводиться моделювання реальної системи, а потім, якщо моделювання дає задовільні результати, можна приступати до реалізації розглянутої системи. Комп'ютерне моделювання або комп'ютерна модель має завдання імітувати абстрактну модель конкретної або еквівалентної системи. Комп'ютерна симуляція стала корисною частиною математичного моделювання багатьох природних систем у фізиці, механіці, хімії, біології, економіці, психології та соціальних науках, а також в усіх галузях інженерії, щоб отримати краще уявлення про роботу досліджуваних систем.

Для того, щоб мати корисну модель, необхідно визначити її поведінку для конкретного та обмеженого набору змінних. Це означає, що для деяких випадкових вхідних параметрів спостерігаються відповідні вихідні значення.

Моделювання в повсякденному житті може бути пов'язане з різними видами діяльності. Якщо це слово використовується в комп'ютерних технологіях, то під терміном «симуляція» розуміють процес створення абстрактних моделей систем з реального середовища та проведення відповідної кількості експериментів над ними. Коли експерименти проводяться на комп'ютері, то вони називаються комп'ютерним моделюванням чи комп'ютерною симуляцією.

Під час моделювання та імітації вхідні дані можуть бути різними і залежать від багатьох факторів. Наприклад, деякі моделі вимагають дуже простих вхідних даних (наприклад, вхідні дані для синусоїди змінного струму базуються на декількох числах), водночас інші моделі вимагають терабайт вхідних даних (наприклад, моделювання погоди або кліматичних змін).

Вхідні дані надаються різними пристроями, якими є:

- датчики та інші фізичні пристрої, які підключаються до моделі;
- панель управління, яка безпосередньо впливає на перебіг самої симуляції;
- поточні або старіші дані, введені вручну;
- значення, що представляють вихідні продукти інших процесів або операцій.

Слід зазначити, що системи, які отримують дані із зовнішніх джерел, повинні враховувати тип представлених даних. Необхідно також оцінювати

точність і не допускати помилок. Якщо помилки з'являються, вони повинні бути зведені до мінімуму. Математика, інтегрована в комп'ютер, не є досконалою, тому приблизні результати, скорочення результатів або нейтралізація невеликих помилок можуть призвести до збільшення потенційних ризиків. У деяких випадках необхідно проаналізувати отриману похибку, щоб переконатися, що результат моделювання є достовірним і може бути використаний у подальших розрахунках і моделюванні. Навіть невеликі помилки у вхідних даних можуть накопичуватися у значні похибки в подальших моделюваннях.

Для чого потрібне комп'ютерне моделювання? Для чого ми використовуємо моделювання та імітацію? Чи є вони необхідними? Ці питання ставляться дуже часто, і причин є досить багато, але серед них найважливішими є такі:

- неможливо отримати математичний розв'язок аналітичної моделі;
- система занадто складна і її неможливо описати аналітично;
- експеримент у реальній системі в більшості випадків або нерентабельний, або занадто складний; моделювання та імітація можуть показати, чи виправдані подальші інвестиції в експеримент, чи ні;
- часто метою моделювання та імітації є сприйняття функціональності наявної реальної системи, структура якої мало відома або до якої неможливо наблизитися;
- коли потрібне оптимальне або оптимізоване функціонування системи, необхідно змінювати різні параметри; якщо брати до уваги реальну систему, то часто це неможливо, тому що такої системи не існує; іншими словами, така система ще не побудована, або ціна такого експерименту надто висока; у таких ситуаціях моделювання є найкращим рішенням.

Конкретно у яких випадках потрібно застосувати комп'ютерне моделювання? Цей підхід необхідно здійснити у таких ситуаціях:

- коли необхідно змоделювати умови, які призводять до руйнування системи; руйнування реальної системи в більшості випадків не допускається, тому комп'ютерне моделювання в таких ситуаціях є єдиним рішенням;
- коли йдеться про довготривалі процеси в реальній системі або всередині реальної системи, то час може бути проблематичним фактором – у таких ситуаціях комп'ютерне моделювання може прискорити процес або скоротити його штучно;
- коли йдеться про надзвичайно швидкі процеси всередині реальної системи, тоді комп'ютерне моделювання – це рішення, яке дає змогу контролювати такі процеси;
- іноді експеримент доводиться зупиняти з різних причин, а часто це неможливо реально – комп'ютерне моделювання такого експерименту вирішує проблему просто, тому що симуляцію можна зупинити та продовжити, коли це необхідно.

Як і все в житті, комп'ютерне моделювання не ідеальне, і тут виникають різні проблеми. Симуляції загалом дуже корисні, але вони мають як переваги, так і недоліки. Основними перевагами комп'ютерного моделювання є:

- коли модель створена, її можна використовувати багаторазово для аналізу необхідного процесу чи конструкції;
- комп'ютерне моделювання може бути надзвичайно корисним, навіть якщо вхідні дані є неповними;
- у більшості випадків простіше та дешевше отримати вихідні дані моделювання, ніж вихідні дані реальної системи;
- комп'ютерне моделювання генерує необхідні дані, які можуть бути використані для оцінки будь-якої характеристики системи і без обмежень;
- у деяких випадках комп'ютерне моделювання може бути єдиним способом вирішення проблем;
- комп'ютерне моделювання може описувати та вирішувати складні проблеми за допомогою динамічного випадкового вибору змінних, які недоступні в математичному моделюванні.

Основними недоліками комп'ютерного моделювання є:

- створення імітаційних моделей, а також комп'ютерне моделювання може бути дорогим і тривалим процесом, адже потрібен час, необхідний для розробки, тестування та перевірки;
- під час використання комп'ютерного моделювання не завжди можуть бути отримані оптимальні рішення;
- для реалізації комп'ютерної моделі необхідні знання різних інструментів і методів;
- оцінка моделі є досить складним процесом і вимагає додаткових експериментів.

Незважаючи ні на що, комп'ютерне моделювання є дуже корисною річчю, і його використання є дуже поширеним. Очевидно, що застосування комп'ютерного моделювання має набагато більше переваг, ніж недоліків, і безсумнівно, що такий механізм буде домінувати майже в кожній сфері.

3.3. Моделюючі комп'ютерні програми

Будь-яка комп'ютерна програма представляє собою, по суті, модель, яка відтворює поведінку реальної системи. Програмні засоби моделювання базуються на різноманітних імітаційних моделях, як-от генератори випадкових чисел і різні математичні розподіли. В якості прикладу комп'ютерної моделі проведемо програмний варіант обчислення означеного інтегралу.

Геометрична інтерпретація означеного інтегралу – це криволінійна фігура, зображена на площині у декартовій системі координат і обмежена зверху кривою підінтегральної функції $f(x)$, знизу – віссю *x*-ів, справа та зліва – вертикальними прямими, що відповідають межам інтегрування. Таку криволінійну замкнуту фігуру називають криволінійною трапецією. Існує кілька способів знаходження площі такої криволінійної трапеції. Для чисельного обрахування інтегралу потрібно площу цієї фігури S (рис. 3.2) розбити на нескінченно малі прямокутники, і тоді:

$$S = \int_a^b f(x)dx. \tag{3.1}$$

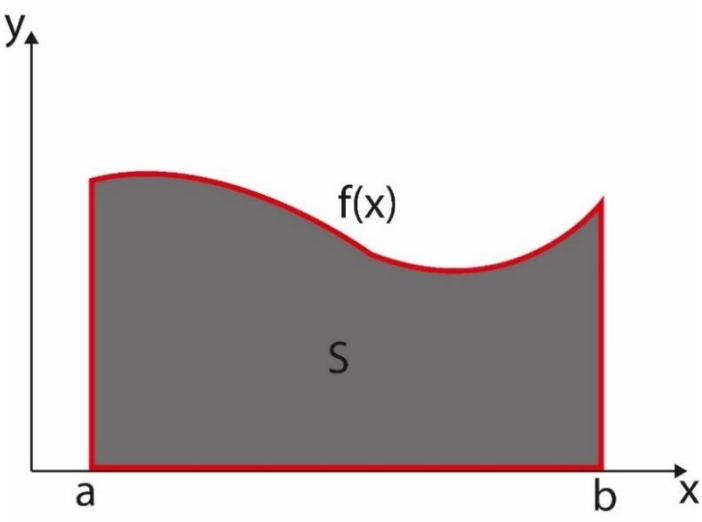


Рис. 3.2. Криволінійна трапеція. Червоний контур охоплює площу, яка обраховується

Для практичних розрахунків нескінченно малі значення *dx* неприйнятні, і тому використовують наближені чисельні обчислення, застосовуючи метод прямокутників, метод трапецій чи метод Сімпсона. Почнемо з методу прямокутників. Розділимо криволінійну трапецію на тоненькі смужки – прямокутники. Чим їх буде більше, тим точнішим буде результат.

В якості точок поділу виберемо спектр величин $x_0 = a, x_1, x_2, ..., x_{n-1}, x_n = b$. Нехай довжина проміжку розбиття $x_{i+1} - x_i = h$. Тоді можна записати таку послідовність площ елементарних прямокутників:

$h \times f(a + 0 \cdot h)$;

$h \times f(a + 1 \cdot h)$;

$h \times f(a + 2 \cdot h)$;

$h \times f(a + i \cdot h)$.

Сумуючи ці площі, отримаємо площу криволінійної трапеції (рис. 3.3):

$$S = h \sum_{i=0}^{i=\frac{b-a}{h}} f(a + i \cdot h). \qquad (3.2)$$

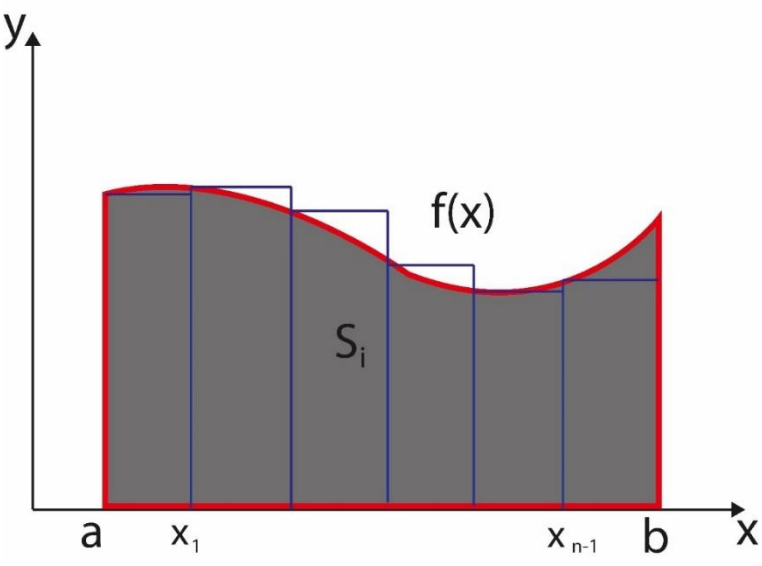


Рис. 3.3. Метод прямокутників обчислення означеного інтегралу

Розглянемо приклад комп'ютерної програми, що дає змогу, використовуючи (3.3), чисельно обраховувати інтеграл:

$$\int_0^2 \sqrt{x^3 + 7}\,dx.$$

Програма (Лістинг 3.1), написана на Java, виглядає так:

Лістинг 3.1

```
public class Square {
  //Обрахунок інтеграла з допомогою методу трапецій
  public static void main(String[] args) {
    System.out.println(SquareRact(0, 2, x-> Math.sqrt((Math.pow(x,3)+7)));
  }
  public static double SquareRact(double a, double b, function f) {
    double S = 0;
    double h = 0.0000001;
     for (int i = 0; i < (b-a)/h; i++) {
      S += h * f.func(a + i * h);
    }
```

```
        return S;
    }
    interface  function {
        double func(double x);
    }
}
```

У наведеній програмі процедура обчислення зводиться до обрахунку величини S завдяки використанню циклу:

```
for (int i = 0; i < (b – a)/h; i++) {
    S += h * f.func(a + i * h);
}
```

Чим меншою буде величина h у приведеній вище програмі, тим точнішим буде результат. Запустимо програму за умови h = 0,0000001. Отримаємо значення інтегралу 5,956640871110381. Збільшимо точність обчислень. Нехай h = 0,00000001. Результат обчислень – 5,956640926334963. Як бачимо, відмінність проявляється лише у сьомому знакові після коми.

Програма Maple також належить до засобів комп'ютерного моделювання. Це багатофункціональна програма, компоненти інтерфейсу якої представлені на рис. 3.4.

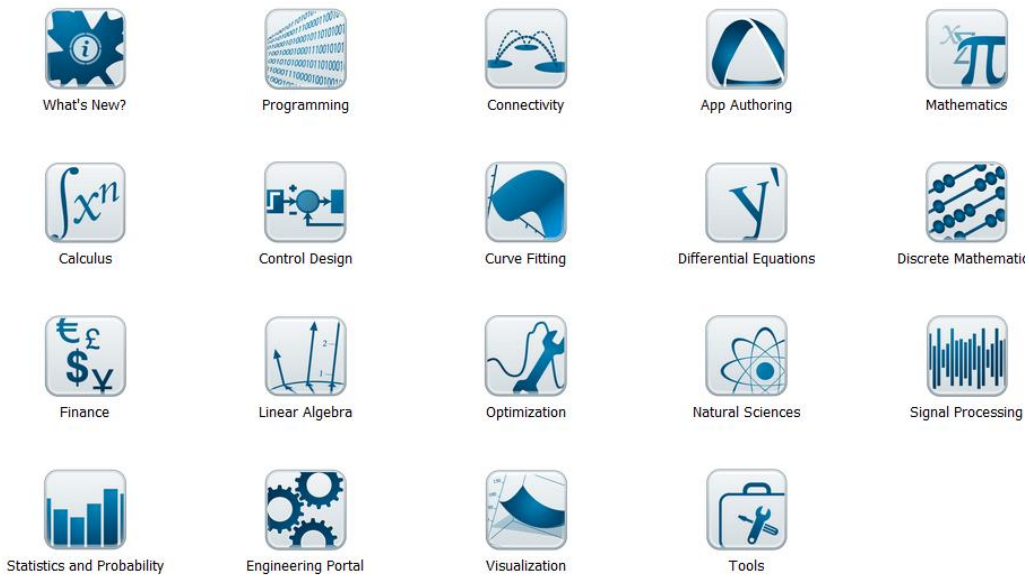


Рис. 3.4. Інтерфейс програми комп'ютерного моделювання Maple

Для прикладу розрахуємо з допомогою цієї програми інтеграл, який ми розраховували з використанням мови програмування Java. Результат обчислень співпадає з точністю до п’ятого знаку після коми.

Метод трапецій аналогічний методу прямокутників з тією відмінністю, що в якості елементарних апроксимуючих фігур використовуються трапеції, які більш точно співпадають із функцією $f(x)$. Ситуація в цьому випадку виглядає, як показано на рис. 3.5.

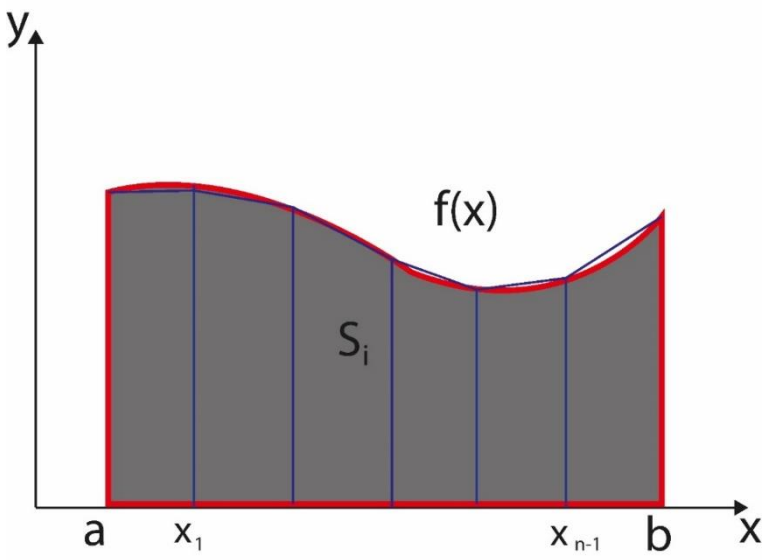


Рис. 3.5. Графічне представлення методу трапецій

Отже, у випадку використання цього методу криволінійна фігура (рис. 3.2) розбивається на елементарні трапеції, площа кожної з яких представляється у вигляді:

$$S_i = h \cdot \frac{1}{2} \cdot (f(a + i \cdot h) + f(a + (i+1) \cdot h)). \qquad (3.3)$$

Додаючи такі елементарні площі, отримаємо наближене значення інтегральної суми:

$$S = h \cdot \frac{1}{2} \cdot \sum_{i=0}^{(b-a)/h} (f(a + i \cdot h) + f(a + (i+1) \cdot h)). \qquad (3.4)$$

Програмний варіант методу трапецій базується на обрахунку величини:

```
for (int i = 0; i < (b-a)/h; i++) {
S += h * (0.5 * (f.func(a + i * h)+ f.func(a+(i+1)*h)))
},
```

як показано у наступній програмі:

Лістинг 3.2

```
public class Square {
   public static void main(String[] args) {
      System.out.println(SquareTrap(0, 2, x-> Math.sqrt((Math.pow(x,3)+7))));
   }
   public static double SquareTrap(double a, double b, function f) {
      double S = 0;
      double h = 0.0000001;
      for (int i = 0; i < (b-a)/h; i++) {
         S += h * (0.5 * (f.func(a + i * h)+ f.func(a+(i+1)*h)));
      }
      return S;
   }
   interface  function {
      double func(double x);
   }
}
```

Результат обчислень: 5,956640932471819. Розбіжність із результатом, отриманим з допомогою методу прямокутників, проявляється лише у сьомому знаку після коми.

Проведені обчислення дають підстави стверджувати, що і методи прямокутників, і методи трапецій цілком можна використовувати для наближеного (але з високою точністю) обчислення інтегралів. До речі, як відомо, існує багато інтегралів, які не беруться в аналітичних функціях. Прикладом інтеграла, що не береться в елементарних функціях, є інтеграл Пуассона:

$$\int e^{-x^2} dx. \tag{3.5}$$

Для того, щоб розрахувати цій інтеграл як визначений у певних межах, потрібно скористатись наведеними вище програмами. Скористаємось методом трапецій як більш точним. Нехай потрібно обрахувати визначений інтеграл:

$$\int_0^{15} e^{-x^2} dx. \tag{3.6}$$

Для цього нам потрібно у програмі (Лістинг 3.2) змінити всього один запис, вказавши новий вираз для функції, інтеграл від якої ми бажаємо розрахувати, а саме:

System.*out*.println(*SquareTrap*(0, 15, x-> Math.*exp*(Math.*pow*(-x,2)))).

Аналогічно розраховуються інші інтеграли, що не беруться в елементарних функціях, наприклад:

$$\int \frac{\sin x}{x} dx . \tag{3.7}$$

Інакше кажучи, це значить, що потрібно використовувати наближені методи обчислень. Саме так ми зробили, використавши два методи – метод прямокутників та метод трапецій – як для функцій, первісні від яких виражаються в елементарних функціях, так і для випадку, коли первісні не можна представити у вигляді елементарних функцій.

Комп'ютерні моделі можна також використовувати для побудови графіків. Нехай потрібно побудувати графік функції:

$$y = \frac{1+\sin x}{1+|x|} . \tag{3.8}$$

Програма (Лістинг 3.3), написана на мові програмування Java, дає змогу побудувати графік будь-якої функції у заданому діапазоні. Далі приведемо повний код програми, який починається із імпорту пакетів java.awt та java.lang.

Лістинг 3.3

```
// Підключення пакетів:
import java.awt.*;
import java.awt.event.*;
import static java.lang.Math.cos;
import static java.lang.Math.tan;
import static java.lang.StrictMath.abs;
import static java.lang.StrictMath.sin;
   // Клас фрейму:
   class PlotFrame extends Frame{
      // Конструктор (аргументи – висота і ширина вікна):
      PlotFrame(int H,int W){
// Заголовок вікна:
         setTitle("Графік функції");
```

```
        setBounds(900,700,W,H); // Положення і розмір вікна
        setBackground(Color.GRAY); // Колір  фону вікна
        setLayout(null); // Відключення  менеджера разташування елементів
        Font f=new Font("Arial",Font.BOLD,11); // Визначення шрифту
        setFont(f); // Застосування шрифту
        BPanel BPnl=new BPanel(6,25,W/4,H-30); // Створення панелі з
кнопками
        add(BPnl); // Додавання панелі в головне вікно
// Панель для відтворення  графіка (створення):
        PPanel PPnl=new PPanel(W/4+8,25,5*W/4-15,H-12,BPnl);
// Додавання панелі в головне вікно:
        add(PPnl);
// Третя панель для відтворення довідки:
        HPanel HPnl=new HPanel(W/4+10,H-90,3*W/4-15,85);
// Додавання панелі в головне вікно:
        add(HPnl);
// Регистрація обробника у  вікні (закриття вікна):
        addWindowListener(new WindowAdapter(){
           public void windowClosing(WindowEvent ve){
              System.exit(0);} // Закрытие окна
        });
// Реєстрація обробника для першої кнопки:
        BPnl.B1.addActionListener(new Button1Pressed(BPnl,PPnl));
// Реєстрація обробника для другої кнопки:
        BPnl.B2.addActionListener(new Button2Pressed());
// Реєстрація обробника для прапорця виведення сітки:
        BPnl.Cb[3].addItemListener(new cbChanged(BPnl));
// Розмір вікна (фрейма) не змінюється:
        setResizable(false);
// Значок для вікна програми:
        setIconImage(getToolkit().getImage("C:/icons/icon.png"));

        setVisible(true); // Відтворення вікна
      }}
   // Клас панелі з кнопками:
   class BPanel extends Panel{
      // Мітки панелі:
      public Label[] L;
      // Група перемикачів панелі:
      public CheckboxGroup CbG;
```

```
        // Перемикачі панелі:
        public Checkbox[] Cb;
        // Список, що розкривається:
        public Choice Ch;
        // Текстове поле:
        public TextField TF;
        // Кнопки панелі:
        public Button B1,B2;
 // Конструктор
// (аргументи – координати і розміри панелі):
        BPanel(int x,int y,int W,int H){
// Відключення  менеджера розміщення елементів на панелі:
            setLayout(null);
            setBounds(x,y,W,H); // Положення і розмір панелі
            setBackground(Color.LIGHT_GRAY); // Колір фону панелі
// Масив міток:
            L=new Label[3];
// Текстова мітка:
            L[0]=new Label("Вибір кольору:",Label.CENTER);
// Шрифт для текстової мітки:
            L[0].setFont(new Font("Arial",Font.BOLD,12));
// Розмір  мітки:
            L[0].setBounds(15,15,getWidth()-10,30);
// Додавання мітки на панель:
            add(L[0]);
// Група перемикачів:
            CbG=new CheckboxGroup();
            Cb=new Checkbox[4];
// Перемикачі групи:
            Cb[0]=new Checkbox(" червоний ",CbG,true); // Червоний
            Cb[1]=new Checkbox(" синій ",CbG,false); // Синій
            Cb[2]=new Checkbox(" чорний ",CbG,false); // Чорний
// Прапорець виводу сітки:
            Cb[3]=new Checkbox(" Сітка ",true);
// Розміри перемикачів і прапорця так и додавання їх на панель:
            for(int i=0;i<4;i++){
               Cb[i].setBounds(5,30+i*25,getWidth()-10,30); // Розмір
               add(Cb[i]);

            }
```

```
// список вибору кольору для ліній сітки:
        Ch=new Choice();
// Додавання елементу «Зелений»:
        Ch.add("Red");
// Додавання елементу «Жовтий»:
        Ch.add("Blue");
// Додавання елементу «Сірий»:
        Ch.add("Grey");
// Розмір і положення  списку:
        Ch.setBounds(220,240,getWidth()-25,30);
// Додавання списку на панель:
        add(Ch);
// Друга текстова мітка:
        L[1]=new Label("Інтервал по x:",Label.CENTER);
// Шрифт для мітки:
        L[1].setFont(new Font("Arial",Font.BOLD,12));
// Розмір і положення мітки:
        L[1].setBounds(250,820,getWidth()-10,30);
// Додавання мітки на панель:
        add(L[1]);
// Третя текстова мітка:
        L[2]=new Label("От x=0 до x=",Label.LEFT);
// Розмір і положення мітки:
        L[2].setBounds(15,250,170,120);
// Додавання мітки на панель:
        add(L[2]);
// Текстове поле для введення границі інтервалу:
        TF=new TextField("10");
// Розмір і положення поля:
        TF.setBounds(275,350,245,220);
// Додавання поля на панель:
        add(TF);
// Перша кнопка («Намалювати»):
        B1=new Button("Намалювати");
// Друга кнопка («Закрити»):
        B2=new Button("Закрити");
// Розміри і положення першої кнопки:
        B1.setBounds(5,getHeight()-75,getWidth()-10,30);
// Розмір і положення другої кнопки:
        B2.setBounds(5,getHeight()-35,getWidth()-10,30);
```

```
            add(B1); // Додавання першої кнопки на панель
            add(B2); // Додавання другої кнопки на панель
        }}
    // Клас панелі для відображення графіка:
    class PPanel extends Panel{

        // Посилання на об'єкт реалізації графіка функції:
        public Plotter G;
        // Внутрішній клас для реалізації графіка функції:
        class Plotter{
            // Границі діапазону зміни координат:
            private double Xmin=0,Xmax,Ymin=0,Ymax=1.0;
            // Стан прапорця виведення сітки:
            private boolean status;
            // Колір для лінії графіка:
            private Color clr;
            // Колір для відтворення лінії сітки:
            private Color gclr;
     // Конструктор класу
    // (аргументи – панель з кнопками і панель для вітворення графіка):
            Plotter(BPanel P){
    // Зчитування значення текстового поля і перетворення в число:
                try{
                    Xmax=Double.valueOf(P.TF.getText());}
                catch(NumberFormatException e){
                    P.TF.setText("10");
                    Xmax=10;}
                status=P.Cb[3].getState();
    // Визначення кольору ліній сітки:
                switch(P.Ch.getSelectedIndex()){
                    case 0:
                        gclr=Color.GREEN;
                        break;
                    case 1:
                        gclr=Color.YELLOW;
                        break;
                    default:
                        gclr=Color.GRAY;}
    // Колір лінії графіка:
                String name=P.CbG.getSelectedCheckbox().getLabel();
```

```
            if(name.equalsIgnoreCase(" червоний ")) clr=Color.RED;
            else if(name.equalsIgnoreCase(" синій ")) clr=Color.BLUE;
            else clr=Color.BLACK;
        }
        // Відтворювана  на графіку функція:
        private double f(double x){
            return (1 + sin(x))/(1+ abs(x));}
        // Метод для зчитування і запам'ятовування налаштувань:
        public Plotter remember(BPanel P){
            return new Plotter(P);}
// Метод для відтворення графіка і сітки
// (Fig – об'єкт графічного контекста):

        public void plot(Graphics Fig){
// Параметри області відтворення графіка:
            int H,W,h,w,s=20;
            H=getHeight();
            W=getWidth();
            h=H-2*s;
            w=W-2*s;
// Очистка області графіка:
            Fig.clearRect(0,0,W,H);
// Індексна змінна і кількість  ліній сітки:
            int k,nums=10;
// Колір кординатних осей – чорний:
            Fig.setColor(Color.BLACK);
// Відтворення координатних осей:
            Fig.drawLine(s,s,s,h+s);
            Fig.drawLine(s,s+h,s+w,s+h);
// Відображення засічок і числових значень на координатних осях:
            for(k=0;k<=nums;k++){
               Fig.drawLine(s+k*w/nums,s+h,s+k*w/nums,s+h+5);
               Fig.drawLine(s-5,s+k*h/nums,s,s+k*h/nums);
               Fig.drawString(Double.toString(Xmin+k*(Xmax-
Xmin)/nums),s+k*w/nums-5,s+h+15);
               Fig.drawString(Double.toString(Ymin+k*(Ymax-Ymin)/nums),s-
17,s+h-1-k*h/nums);
            }
// Відображення сітки (якщо встановлений прапорець):
            if(status){
```

```
                Fig.setColor(gclr);
// Відображення ліній сітки:
                for(k=1;k<=nums;k++){
                   Fig.drawLine(s+k*w/nums,s,s+k*w/nums,h+s);
                   Fig.drawLine(s,s+(k-1)*h/nums,s+w,s+(k-1)*h/nums);
                }}
// Відображення графіка:
             Fig.setColor(clr); // Установка кольору лінії
// Масштаб на один піксель по кожній із координат:
             double dx=(Xmax-Xmin)/w,dy=(Ymax-Ymin)/h;
// Змінні для запису декартових координат:
             double x1,x2,y1,y2;
// Змінні для запису координат у вікні відображення графіка:
             int h1,h2,w1,w2;
// Початкові  значення:
             x1=Xmin;
             y1=f(x1);
             w1=s;
             h1=h+s-(int)Math.round(y1/dy);
// Крок у пікселях для базових точок:
             int step=5;

// Відображення базових точок і з'єднання їх лініями:
             for(int i=step;i<=w;i+=step){
                x2=i*dx;
                y2=f(x2);
                w2=s+(int)Math.round(x2/dx);
                h2=h+s-(int)Math.round(y2/dy);
// Лінія:
                Fig.drawLine(w1,h1,w2,h2);
// Базова точка (квадрат):
                Fig.drawRect(w1-2,h1-2,4,4);
// Нові значення для координат:
                x1=x2;
                y1=y2;
                w1=w2;
                h1=h2;}
          }}
      // Конструктор панелі
// (аргументи – координати і розміри панелі,
```

```
// а також посилання на панель з кнопками):
      PPanel(int x,int y,int W,int H,BPanel P){
// Створення об'єкта реалізації графіка функції:
         G=new Plotter(P);
// Колір фону панелі:
         setBackground(Color.WHITE);
// Розмір і положення панелі:
         setBounds(x,y,W,H);
      }
      // Перевизначення методу фарбування панелі:
      public void paint(Graphics g){
// Під час фарбування панелі викликається метод
// для відображення графіка:
         G.plot(g);
      }}
   // Клас для панелі довідки:
   class HPanel extends Panel{
      // Мітка:
      public Label L;
      // Текстова область:
      public TextArea TA;
      // Конструктор створення панелі
// (аргументи – координати і розміри панелі):
      HPanel(int x,int y,int W,int H){
// колір фона панелі:
         setBackground(Color.LIGHT_GRAY);
// Розмір і положення панелі:
         setBounds(x,y,W,H);

// Відключення менеджера розташування і компонентів панелі:
            setLayout(null);
// Мітка для панелі довідки:
         L=new Label("ДОВІДКА",Label.CENTER);
// Розмір і положення мітки:
         L.setBounds(0,0,W,20);
// Додавання мітки на панель:
         add(L);
// Текстова область для панелі довідки:
         TA=new TextArea(" График функции y(x)=((1 + sin(x))/(1+
abs(x)))");
```

```
// Шрифт для текстової області:
        TA.setFont(new Font("Serif",Font.PLAIN,15));
// Розмір і положення текстової області:
        TA.setBounds(5,20,W-10,60);
// Область недоступна для редагування:
        TA.setEditable(false);
// Додавання текстової області на панель довідки:
        add(TA);
      }}
   // Клас обробника для першої кнопки:
   class Button1Pressed implements ActionListener{
      // Панель з кнопками:
      private BPanel P1;
      // Панель для відображення графіки:
      private PPanel P2;
      // Конструктор класу (аргументи – панелі):
      Button1Pressed(BPanel P1,PPanel P2){
        this.P1=P1;
        this.P2=P2;}
      // Метод для обробки клацання по кнопці:
      public void actionPerformed(ActionEvent ae){
// Оновлення параметрів (налаштувань) для відображення графіка:
        P2.G=P2.G.remember(P1);
// Реакція на клацання (малювання графіка):
        P2.G.plot(P2.getGraphics());
      }}
   // Клас обробника для другої кнопки:
   class Button2Pressed implements ActionListener{
      // Метод для обробки клацання по кнопці:
      public void actionPerformed(ActionEvent ae){
// Реакція на клацання:
        System.exit(0);
      }}
   // Клас обробника для прапорця виведення сітки:
   class cbChanged implements ItemListener{

      // Список вибору кольору для сітки:
      private Choice ch;
      // Конструктор класу (аргумент – панель з кнопками):
      cbChanged(BPanel P){
```

```
        this.ch=P.Ch;}
      // Метод для обробки зміни стану прапорця:
      public void itemStateChanged(ItemEvent ie){
// Реакція на зміну стану прапорця:
        ch.setEnabled(ie.getStateChange()==ie.SELECTED);
      }}
   // Клас з головним методом програми:
   class PlotDemo{
      public static void main(String args[]){
// Створення вікна:
        new PlotFrame(400,500);} }
```

Особливості наведеної вище програми детально проаналізовані у ній у вигляді коментарів до кожного структурного програмного блоку. Результат роботи цієї програми (рис. 3.6) має такий вигляд:

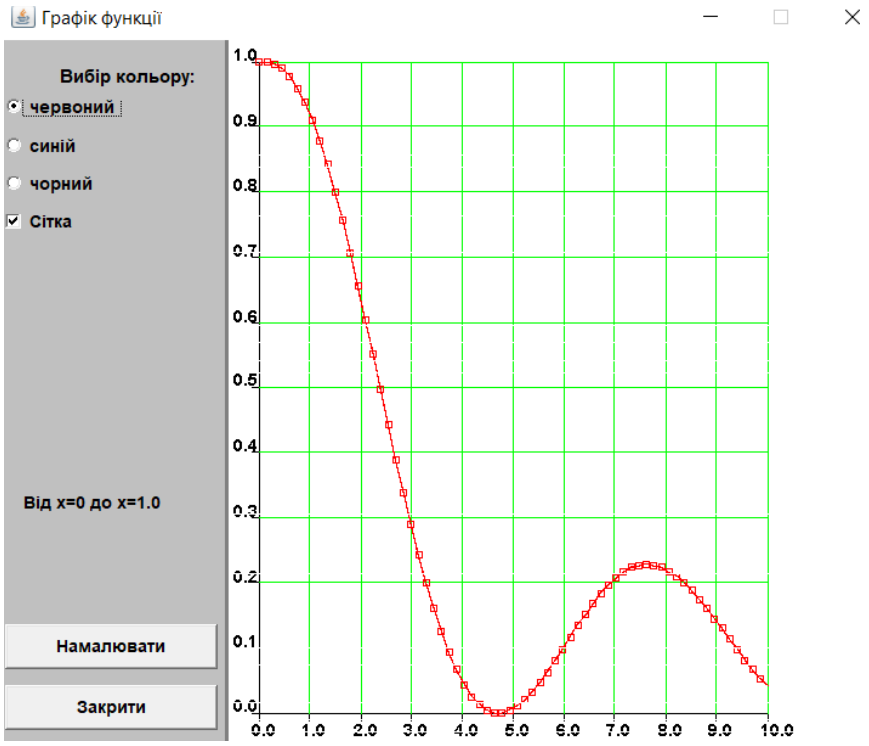


Рис. 3.6. Графік функції $y=\frac{1+\sin x}{1+abs(x)}$, побудований з допомогою методів комп'ютерного моделювання

У цьому випадку наведена вище програма відтворює графік функції (у програмі вираз для функції виділений червоним кольором):

$$y=\frac{1+\sin x}{1+abs(x)}. \qquad (3.9)$$

Наведені у програмі (Лістинг 3.3) коментарі дають змогу розібратись із особливостями використовуваних методів. Але тут важливо також зазначити, що дуже просто побудувати графік будь-якої іншої функції. Для цього просто потрібно ввести інший аналітичний вираз для функції. Наприклад, введемо у програмне поле програми запис:

$$\frac{sqrt(x)}{2+x^2} \tag{3.10}$$

Знову запустимо програму. Отримаємо таке зображення (рис. 3.7):

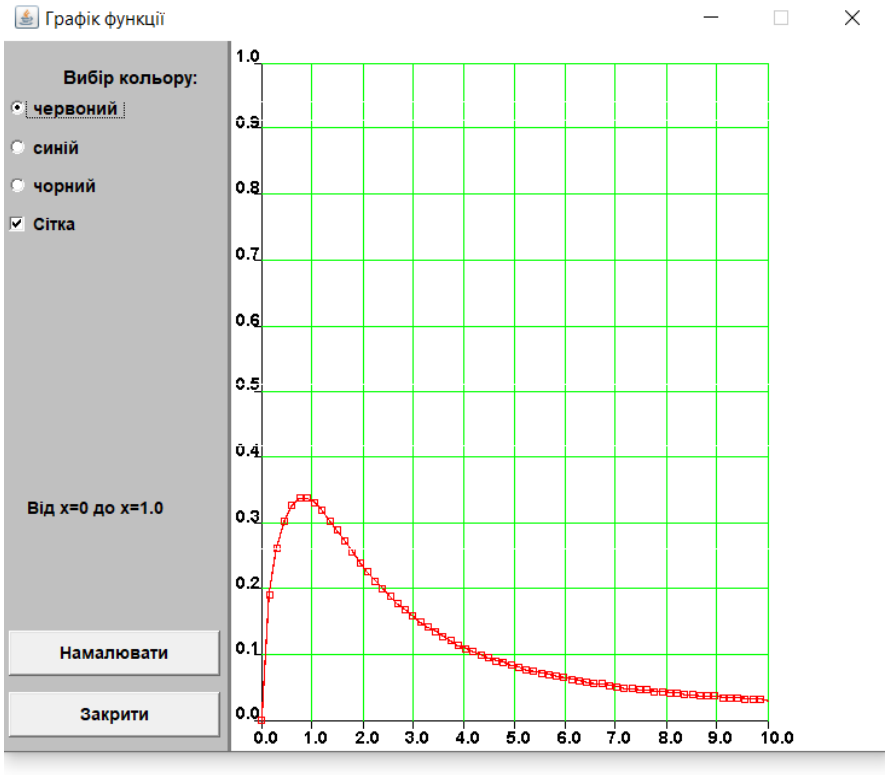


Рис. 3.7. Графік функції $y=\frac{sqrt(x)}{2+x^2}$.

Аналогічно можна побудувати графік будь-якої аналітичної і не тільки аналітичної функції. На «побудову» графіка буде витрачено секунди. Дослідник має можливість будувати графіки будь-яких функцій, витрачаючи секунди. І звичайно, такий підхід дає суттєві переваги.

Розглянемо тепер комп'ютерну модель, що дає змогу працювати з бінарними деревами. Такі графові структури широко використовуються для зберігання та швидкого пошуку збережених даних. Суть полягає у застосуванні бінарного пошуку, який порівняно з лінійним пошуком, є набагато швидшим, і тому – ефективним. Для прикладу наведемо програму (Лістинг 3.4), що реалізує бінарний пошук на Java:

Лістинг 3.4

```
import java.util.*;
/**
 * Created by пк on 03.05.2023.
 */
public class BinSearch {
   public static void main(String[] args) {

      int[] arr = {23, 45, 24, 90,90, 43, 0, -23, 89, 266, 789, 86};
      Arrays.sort(arr);
      int SV = 90;
       int RV =  Arrays.binarySearch(arr, SV);

      {
         for (int i = 0; i < arr.length; i++)


         System.out.print(arr[i]+ "   ");
         System.out.println();
         System.out.println(RV);
      }
   }
}
```

Принцип роботи програми дуже простий – потрібно спочатку упорядкувати заданий масив чисел, використовуючи метод *Arrays.sort(arr)*, а потім у відсортованому масиві знайти індекс шуканого елементу SV. Суть бінарного пошуку дуже проста – ділимо відсортований масив чисел навпіл і встановлюємо, в якій із отриманих половин міститься шукане число. Переходимо до діапазону чисел знайденої половини. Знову здійснюємо процедуру поділу і порівняння і под., поки не відшукаємо потрібне число. З огляду на описану процедуру, легко припустити доречність зберігання даних у графі типу «дерево», де номер кожної вершини впорядкований у певному порядку – або в числовому, або у лексикографічному. Якщо вершини упорядковані у числовому порядку, то це значить, що ліві вершини менші, ніж праві, як показано на рис. 3.8.

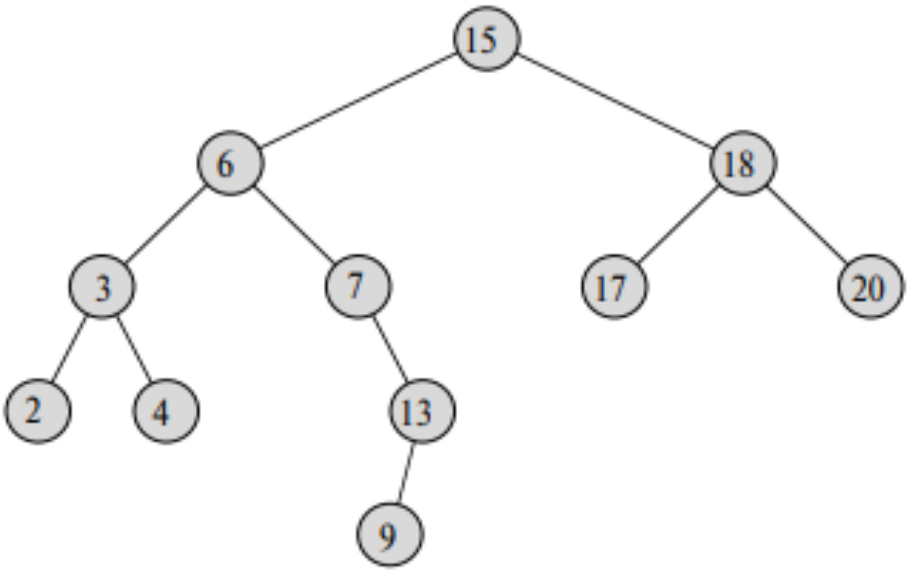


Рис. 3.8. Дерево, вершини якого упорядковані – ключ лівої вершини завжди менший за ключ правої

Оскільки ключ лівої вершини завжди менший за ключ правої, то це дає змогу дуже швидко відшукати потрібний ключ і дістатись до інформації, записаної у вершині зі знайденим ключем

Наведемо приклад програми (Лістинг 3.5), що дає змогу показувати, відшукувати, вставляти, видаляти чи переміщати ключі.

Лістинг 3.5

```
import java.io.*;
import java.util.*; // Для використання класу Stack
    class Node
  {
    public int iData; // Дані, що використовуються в якості ключа
    public double dData; // Інші дані
    public Node leftChild; // Лівий нащадок вузла
    public Node rightChild; // Правий нащадок вузла
    public void displayNode() // Виведення вузла
    {
      System.out.print('{');
      System.out.print(iData);
      System.out.print(", ");
      System.out.print(dData);
      System.out.print("} ");
    }
  } // Кінець класу Node
```

```
class Tree {
private Node root; // first node of tree
public Tree() // Конструктор
{
   root = null;
} // Поки немає жодного вузла
public Node find(int key) // Пошук вузла із заданим ключем
{ // (припускається, що дерево не пусте)
   Node current = root; // Почати з кореневого вузла
   while (current.iData != key) // Поки не знайдено співпадіння
   {
      if (key < current.iData) // Рухатись наліво?
         current = current.leftChild;
      else // чи  направо?
         current = current.rightChild;
      if (current == null) // Якщо нащадка немає,
         return null; // пошук завершився невдачею
   }
   return current; // Елемент знайдено
}
public void insert(int id, double dd) {
   Node newNode = new Node(); // Створення нового вузла
   newNode.iData = id; // Вставка даних
   newNode.dData = dd;
   if (root == null) // Корневого вузла не існує
      root = newNode;
   else // Корневий вузол зайнятий
   {
      Node current = root; // Почати з кореневого вузла
      Node parent;
      while (true) // (внутрішній вихід із циклу)
      {
         parent = current;
         if (id < current.iData) // Рухатись наліво?
         {
            current = current.leftChild;
            if (current == null) //Якщо досягнутий кінець ланцюга,
            { // вставити зліва
               parent.leftChild = newNode;
```

```
                return;
            }
        } else // чи справа?
        {
            current = current.rightChild;
            if (current == null) // Якщо досягнуто кінця ланцюга,
            { // вставити справа
                parent.rightChild = newNode;
                return;
            }
        }
    }
}
public boolean delete(int key) // Видалення вузла з заданим ключем
{ // (припускається, що дерево не пусте)
    Node current = root;
    Node parent = root;
    boolean isLeftChild = true;
    while (current.iData != key) // Пошук вузла

    {
        parent = current;
        if (key < current.iData) // Рухатись наліво?
        {
            isLeftChild = true;
            current = current.leftChild;
        } else // Чи направо?
        {
            isLeftChild = false;
            current = current.rightChild;
        }
        if (current == null) // Кінець ланцюга
            return false; // Вузол не знайдено
    }
// Вузол, що видаляється, знайдено
// Якщо вузол не має нащадків, він просто видаляється
    if (current.leftChild == null &&
            current.rightChild == null) {
```

```
            if (current == root) // Якщо вузол кореневий,
               root = null; // дерево очищається
            else if (isLeftChild)
               parent.leftChild = null; // Вузол від'єднується від батька
                  else
               parent.rightChild = null;
         }
// Якщо немає правого нащадка, вузол заміняється лівим піддеревом
         else if (current.rightChild == null)
            if (current == root)
               root = current.leftChild;
            else if (isLeftChild)
               parent.leftChild = current.leftChild;
            else
               parent.rightChild = current.leftChild;
// Якщо немає лівого нащадка, вузол замінюється правим піддеревом
         else if (current.leftChild == null)
            if (current == root)
               root = current.rightChild;
            else if (isLeftChild)
               parent.leftChild = current.rightChild;
            else
               parent.rightChild = current.rightChild;
         else // Два нащадки, вузол змінюється наступним
         {
// Пошук наступного вузла для вузла, що видаляється (current)
            Node successor = getSuccessor(current);
// Батько current зв'язується з посередником
            if (current == root)
               root = successor;
            else if (isLeftChild)
               parent.leftChild = successor;
            else
               parent.rightChild = successor;
// Наступний вузол зв'язується з лівим нащадком  current
             // Прикмета успішного  завершення
         }
         return true;
      }
```

```
// Метод повертає вузол із наступним значенням після delNode.
// Для цього він спочатку переходить до правого нащадка а потім
// відслідковує ланцюг лівих нащадків цього вузла.
        private Node getSuccessor (Node delNode) {
           Node successorParent = delNode;
           Node successor = delNode;
           Node current = delNode.rightChild; // Перехід до правого нащадка
           while (current != null) // Поки залишаються ліві нащадки
           {
              successorParent = successor;
              successor = current;
              current = current.leftChild; // Перехід до лівого нащадка
           }
// Якщо наступний вузол не є
           if (successor != delNode.rightChild) // правим нащадком,
           { // створити зв'язки між вузлами
              successorParent.leftChild = successor.rightChild;
              successor.rightChild = delNode.rightChild;
                        }
           return successor;
        }
        public void traverse(int traverseType)
        {
           switch (traverseType) {
              case 1:
                 System.out.print("\nPreorder traversal: ");
                 preOrder(root);
                 break;
              case 2:
                 System.out.print("\nInorder traversal: ");
                 inOrder(root);
                 break;

              case 3:
                 System.out.print("\nPostorder traversal: ");
                 postOrder(root);
                 break;
           }
           System.out.println();
```

```
}
private void preOrder (Node localRoot)
{
   if (localRoot != null) {
      System.out.print(localRoot.iData + " ");
      preOrder(localRoot.leftChild);
      preOrder(localRoot.rightChild);
   }
}
private void inOrder (Node localRoot)
{
   if (localRoot != null) {
      inOrder(localRoot.leftChild);
      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
   }
}
private void postOrder (Node localRoot)
{
   if (localRoot != null) {
      postOrder(localRoot.leftChild);
      postOrder(localRoot.rightChild);
      System.out.print(localRoot.iData + " ");
   }
}

public void displayTree ()
{
   Stack globalStack = new Stack();
   globalStack.push(root);
   int nBlanks = 32;
   boolean isRowEmpty = false;
   System.out.println(
         ".....");
   while (isRowEmpty == false) {
      Stack localStack = new Stack();
      isRowEmpty = true;

      for (int j = 0; j < nBlanks; j++)
```

```
            System.out.print(' ');
         while (globalStack.isEmpty() == false) {
            Node temp = (Node) globalStack.pop();
            if (temp != null) {
               System.out.print(temp.iData);
               localStack.push(temp.leftChild);
               localStack.push(temp.rightChild);
               if (temp.leftChild != null ||
                     temp.rightChild != null)
                  isRowEmpty = false;
            } else {
               System.out.print("--");
               localStack.push(null);
               localStack.push(null);
            }
            for (int j = 0; j < nBlanks * 2 - 2; j++)
               System.out.print(' ');
         }
         System.out.println();
         nBlanks /= 2;
         while (localStack.isEmpty() == false)
            globalStack.push(localStack.pop());
      }
      System.out.println(
            "...................................................");
   }

} // Кінець  класу Tree
class TreeApp {
   public static void main(String[] args) throws IOException {
      int value;
      Tree theTree = new Tree();
      theTree.insert(50, 1.5);
      theTree.insert(25, 1.2);
      theTree.insert(75, 1.7);
      theTree.insert(12, 1.5);
      theTree.insert(37, 1.2);
      theTree.insert(43, 1.7);
      theTree.insert(30, 1.5);
```

```
      theTree.insert(33, 1.2);
      theTree.insert(87, 1.7);
      theTree.insert(93, 1.5);
      theTree.insert(97, 1.5);
      while (true) {
         System.out.print("Enter first letter of show, ");
         System.out.print("insert, find, delete, or traverse: ");
         int choice = getChar();
         switch (choice) {
            case 's':
               theTree.displayTree();
               break;
            case 'i':
               System.out.print("Enter value to insert: ");
               value = getInt();
               theTree.insert(value, value + 0.9);
               break;
            case 'f':
               System.out.print("Enter value to find: ");
               value = getInt();
               Node found = theTree.find(value);
               if (found != null) {
                  System.out.print("Found: ");
                  found.displayNode();
                  System.out.print("\n");
               }
         }
      }
   }
               public static String getString() throws IOException
            {
               InputStreamReader isr = new InputStreamReader(System.in);
               BufferedReader br = new BufferedReader(isr);
               String s = br.readLine();
               return s;
            }
            public static char getChar () throws IOException
            {
```

```
        String s = getString();
        return s.charAt(0);
    }
    public static int getInt () throws IOException
    {
        String s = getString();
        return Integer.parseInt(s);

    }
}
```

Запустимо програму. На консолі отримаємо

```
Enter first letter of show, insert, find, delete, or traverse: s
.....................................................
                        50
                25                      75
            12          37          --          87
          --    --    30    43    --    --    --    93
        --  --  --  --  --  33  --  --  --  --  --  --  --  --  --  97
```

Видно, що ключі умовного графа впорядковані у числовому сенсі. Якщо потрібно, скажімо, вставити ключ, то інсталюємо у консоль літеру *i*. Натискаємо enter і програма відтворює позицію виставленого ключа. Введемо, наприклад, число 86. Отримаємо:

```
Enter first letter of show, insert, find, delete, or traverse: s
.....................................................
                        50
                25                      75
            12          37          --          87
          --    --    30    43    --    --    86    93
        --  --  --  --  --  33  --  --  --  --  --  --  --  --  --  97
```

Аналогічно можна здійснювати інші операції.

3.4. Патерни як комп’ютерні моделі

Патерн проєктування – це комп’ютерна модель, необхідна для проєктування архітектури комп’ютерних програм. Патерн є схемою, скелетом, заготовкою

майбутньої комп'ютерної програми. Патерн не можна просто скопіювати в програму. Це загальна ідеологія чи концепція, ескіз тієї чи іншої проблеми. Патерни – це не алгоритми, хоча обидва ці поняття описують типові рішення якихось відомих проблем. І якщо алгоритм – це точний набір дій, то патерн – це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Патерни як моделі комп'ютерних програм формально можна охарактеризувати так:

- проблеми, які вирішує патерн;
- обґрунтування до вирішення проблеми, яке пропонує патерн;
- структури класів, що становлять рішення;
- приклад на одній з мов програмування;
- особливості під час реалізації у різних контекстах;
- зв'язки із іншими патернами.

Такий підхід дає змогу зібрати великий каталог патернів, орієнтованих на вирішення багатьох проблем програмування шляхом створення комп'ютерних модельних систем.

Патерни відрізняються за рівнем складності, деталізації та охоплення проєктованої системи. Проводячи аналогію з дорожнім міським трафіком, можна підвищити безпеку перехрестя, поставивши світлофор, а можна замінити перехрестя цілою автомобільною розв'язкою з підземними переходами. Найбільш низькорівневі та прості патерни називаються ідіомами. Вони не дуже універсальні, тому що застосовні тільки в межах однієї мови програмування. Найуніверсальнішими є архітектурні патерни, які можна реалізувати практично будь-якою мовою. Вони потрібні для проєктування (створення моделі) всієї програми, а не окремих її елементів. До того ж патерни відрізняються за призначенням. Загалом патерни умовно поділяються на три групи:

- породжуючі патерни – створюють об'єкти без внесення до програми власних змін;
- структурні патерни – реалізують різні способи побудови зв'язків між об'єктами;
- поведінкові патерни – організовують ефективну комунікацію між об'єктами.

Патерни являють собою комп'ютерні моделі, що враховують весь накопичений досвід у програмуванні складних систем. Отже, використовуючи патерн, ви базуєтесь на вже перевірених та апробованих рішеннях. Ви витрачаєте менше часу, використовуючи вже готові рішення замість по-новому конструювати свою власну програмну конструкцію. До деяких рішень ви змогли б додуматися і самі, але багато що може бути для вас відкриттям.

Патерни дають змогу стандартизувати програмний код. Програміст здійснює значно менше кроків під час проєктування, а рухається вже пройденим маршрутом, використовуючи типові уніфіковані способи вирішення, оскільки всі приховані проблеми в них уже давно знайдено.

Загальні програмні рішення дають змогу уніфікувати весь процес створення нових програмних пакетів.

Процедура проєктування програм має свої специфічні особливості. Однією з них є повторне використання коду. Зрозуміло, що вартість та час розробки – це найбільш важливі критерії під час розробки будь-яких програмних продуктів. Чим менші обидва ці показники, тим конкурентніший продукт буде на ринку і тим більше прибутку отримає розробник. Отож, повторне використання програмної архітектури та коду – це якраз один з найпоширеніших способів як зниження вартості розробки, так і часу, витраченого на цю розробку. Логіка проста: замість того, щоб розробляти щось по-новому, чому б не використати минулі напрацювання у новому проєкті? Ідея виглядає чудово на папері, але, на жаль, не всякий код можна пристосувати до роботи у нових умовах. Річ у тім, що у конкретній програмі спостерігаються надто тісні зв'язки між компонентами та є залежність коду від конкретних класів. Все це зменшує гнучкість програмної архітектури та перешкоджає її повторному використанню. І тут на допомогу приходять патерни проєктування, які ціною ускладнення коду програми підвищують гнучкість її частин, спрощуючи подальше повторне використання коду.

Існує три рівні повторного використання коду. На найнижчому рівні знаходяться класи. Фреймворки стоять на верхньому рівні. У них важлива лише архітектура. Вони визначають ключові абстракції для вирішення деяких бізнес-завдань, представлених у вигляді класів та відносин між ними. Є ще середній рівень, до якого належать патерни проєктування. Вони насправді – просто опис того, як пара класів відноситься і взаємодіє один з одним. Рівень повторного використання підвищується, коли програміст рухаєтеся в напрямі від конкретних класів до патернів, а потім до фреймворків. Водночас патерни дають змогу повторно використовувати ідеї та концепції у відриві від конкретного коду.

Базові принципи створення ефективних моделей проєктування формуються на певних критеріях. Програміст має ставити перед собою запитання: за якими критеріями оцінювати програму і яких правил дотримуватися під час розробки? Також важливо забезпечити достатній рівень гнучкості, пов'язаності, керованості, стабільності та зрозумілості коду. Все це – правильні питання, але для кожної конкретної програми відповідь буде трохи відрізнятися. Тепер розглянемо універсальні принципи проєктування, які допоможуть дати відповіді на поставлені запитання.

Перший принцип проєктування добре відомий: інкапсулюйте те, що змінюється. Для цього визначте аспекти програми, класу чи методу, які змінюються найчастіше, і відокремте їх від того, що залишається незмінним. Цей принцип має єдину мету – зменшити наслідки, що викликаються змінами. Ізолюючи мінливі частини програми в окремих модулях, класах чи методах, ви зменшуєте об'єм коду, який торкнеться наступних змін.

Отже, вам потрібно буде витратити менше зусиль на те, щоб надати програмі робочого стану, налагодити і протестувати код, що змінився. Де менше роботи, там менша вартість розробки. А де менша вартість, там і перевага перед конкурентами.

Другий принцип проєктування: програмуйте на рівні інтерфейсу. Це означає, що програмний код повинен залежати від абстракцій, а не конкретних класів. Гнучкість програмної архітектури повинна полягати у тому щоб її можна було розширювати, не ламаючи наявний код. Коли є необхідність організувати взаємодію між двома об'єктами різних класів, найпростіше у цьому випадку зробити один клас залежним від іншого. Але є і інший, більш ефективний, спосіб. Для його реалізації потрібно визначити, що саме потрібно одному об'єкту від іншого, які методи він викликає. Після цього необхідно описати ці методи в окремому інтерфейсі. Далі треба зробити так, щоб клас-залежність дотримувався цього інтерфейсу. Зазвичай у цьому випадку потрібно буде лише додати цей інтерфейс в описі класу. Остаточно можна зробити другий клас залежним від інтерфейсу, а не конкретного класу.

Розглянемо ще п'ять принципів проєктування, які відомі як SOLID. Ці принципи було вперше викладено Робертом Мартіном. Головна мета цих принципів – підвищити гнучкість архітектури, зменшити пов'язаність між її компонентами та полегшити повторне використання коду. Проте дотримання принципів SOLID має свої особливості. В основному це виражається в ускладненні коду програми. Реально коду, у якому б дотримувалися всі ці принципи одночасно, немає. Тому треба пам'ятати про баланс під час дотримання принципів SOLID.

Перший принцип звучить як ***Single Responsibility Principle.*** Перекладається як принцип єдиної відповідальності. Суть полягає у тому, щоб кожен клас відповідав тільки за одну частину функціональності програми, причому вона має бути повністю інкапсульована в цей клас. Сформульований принцип призначений для боротьби зі складністю. Загалом проблеми програмування виникають, коли система зростає і збільшується у масштабах. Клас розростається і починає володіти певною деструктивністю. Навігація утруднюється і загалом втрачається контроль над кодом. Якщо клас надто багатогранний, тоді доводиться змінювати його щоразу, коли одна із граней змінюється. Водночас складно зберегти

непорушними інші частини класу, які не потрібно було змінювати. Отже, перший принцип SOLID – принцип єдиної відповідальності – полягає у тому, щоб розділяти класи на частини.

Другий принцип формулюється як ***Open / Closed Principle*** – принцип відкритості / закритості. Потрібно прагнути до того, щоб класи були відкриті для розширення, але закриті для зміни. Головна ідея цього принципу у тому, щоб не ламати наявний код під час внесення змін до програми. Клас можна назвати відкритим, якщо він доступний для розширення. Водночас є можливість розширити набір його операцій або додати до нього нові поля, створивши власний підклас. Клас можна назвати закритим, якщо він готовий для використання іншими класами. Це означає, що інтерфейс класу вже остаточно визначений і не змінюватиметься у майбутньому. Якщо клас уже був написаний, схвалений та протестований, то після цього намагатися модифікувати його вміст не бажано. Замість цього можна створити підклас і розширити у ньому базову поведінку, не змінюючи код батьківського класу безпосередньо.

Третій принцип програмного моделювання має назву ***Liskov Substitution Principle*** – принцип підстановки Ліскова. Суть цього принципу зводиться до такої формули: підкласи повинні доповнювати, а не заміщати поведінку базового класу. Потрібно прагнути створювати підкласи так, щоб їх об’єкти можна було підставляти замість об’єктів базового класу, не ламаючи функціональності клієнтського коду. Принцип підстановки – це низка перевірок, які допомагають передбачити, чи залишиться підклас сумісний з рештою коду програми, який до цього успішно працював, використовуючи об’єкти базового класу. Це особливо важливо під час розробки бібліотек та фреймворків. На відміну від інших принципів, які визначені дуже вільно і мають масу трактувань, принцип підстановки має низку формальних вимог до підкласів, а точніше до перевизначених у них методів. Принцип підстановки зводиться до того, що типи параметрів методу підкласу повинні збігатися або бути більш абстрактними.

Четвертий принцип моделювання програм має назву ***Integrate Segregation Principle*** – принцип розділення інтерфейсу. Потрібно прагнути щоб інтерфейси були достатньо вузькими. Тоді класам не доведеться реалізовувати складні, багатогранні програми. Принцип поділу інтерфейсів вказує на те, що надто розширені за функціоналом інтерфейси необхідно розділяти на менші та специфічні.

П’ятий принцип моделювання програм має назву ***Dependency Inversion Principle*** – принцип інверсії залежностей. Класи верхніх рівнів не повинні залежати від класів нижніх рівнів. І перші, і другі мають залежати лише від абстракцій. Абстракції, зі свого боку, не повинні залежати від деталей. Лише деталі мають залежати від абстракцій.

Під час моделювання проєктів програм можна виділити два рівні класів. Класи нижнього рівня реалізують базові операції типу роботи з диском, передачі даних по мережі, підключення до бази даних та інше. Навпаки, класи високого рівня містять складну бізнес-логіку програми, що спирається на класи низького рівня для більш простих операцій. Найчастіше спочатку проєктуються класи нижнього рівня, а лише потім класи верхнього рівня. Отже, принцип інверсії залежностей пропонує змінити напрям, у якому відбувається проєктування програм.

Розглянемо тепер найбільш широко вживані патерни моделювання комп'ютерних програм. Почнемо з аналізу так званих породжувальних патернів. До них належать ***Factory method*** (Фабричний метод), ***Abstract Factory*** (Абстрактна фабрика), ***Builder*** (Будівельник), ***Prototype*** (Прототип) та ***Singleton*** (Одиночка).

Коротка характеристика породжувальних патернів:

Factory method – це породжувальний патерн проєктування, що створює загальний інтерфейс для формування об'єктів у суперкласі, даючи змогу підкласам змінювати тип створюваних об'єктів.

Abstract Factory – це породжувальний патерн проєктування, що дає змогу створювати сукупності пов'язаних об'єктів, не прив'язуючись до конкретних класів створюваних об'єктів.

Builder – це породжувальний патерн проєктування, що дає змогу створювати складні об'єкти крок за кроком. Цей патерн дає змогу використовувати один і той самий код конструювання різних програмних об'єктів.

Prototype – це такий патерн проєктування, який дає змогу копіювати об'єкти, не вдаючись у подробиці реалізації.

Singleton – це патерн проєктування, який гарантує, що клас має лише один екземпляр і надає до нього глобальну точку доступу.

Проаналізуємо більш детально ***Factory method.*** Цей патерн пропонує створювати об'єкт не безпосередньо, використовуючи оператор new, а через виклик особливого фабричного методу. Зауважимо, що всі об'єкти будуть створюватися за допомогою new, але виконувати таку процедуру буде саме фабричний метод. Водночас підкласи можуть змінювати клас об'єктів, що створюються. З'являється можливість перевизначити фабричний метод у підкласі, щоб змінити тип продукту. Для реалізації фабричного методу всі об'єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть створювати об'єкти різних класів, доступних одному й тому ж інтерфейсу.

У всіх фабричних патернах проєктування є дві групи учасників – творці (самі фабрики) та продукти (об'єкти, що створюються фабриками). Уявімо ситуацію: у нас є піцерія, що випускає один вид піци під назвою MOZARELLA. Підприємство розширилось і поглинуло дві інші піцерії, що випускали піци

BACON та CHILI_PEPPER. Отже, потрібно розширюватись та випускати інші види піци, щоб не втрачати клієнтів. Тому стоїть завдання реструктуризувати виробництво так, щоб можна було випускати всі три види піци. Зрозуміло, що рецептура нових продуктів додає нові харчові компоненти. Програмну реалізацію описаної проблеми (Лістинг 3.6) можна представити у вигляді такого коду:

Лістинг 3.6

```
enum PizzaType {
   MOZZARELLA,
   BACON,
   CHILI_PEPPER
}
 abstract class Pizza {
   public void takeDough() {}
   public void addIngredients() {}
   public void bake() {}
}
 class Mozzarella_UA extends Pizza {}
 class Mozzarella_EU extends Pizza {}
 class Bacon_UA extends Pizza {}
 class Bacon_EU extends Pizza {}
 class Chili_Pepper_UA extends Pizza {}
 class Chili_Pepper_EU extends Pizza {}
 abstract class PizzaShop {
   public Pizza orderPizza(PizzaType type) {
      Pizza pizza = createPizza(type);
      pizza.takeDough();
      pizza.addIngredients();
      pizza.bake();
      System.out.println("Bon Appetit!");
      return pizza;
   }
   protected abstract Pizza createPizza(PizzaType type);
}
 class PizzaShop_UA extends PizzaShop {
   @Override
   public Pizza createPizza(PizzaType type) {
      Pizza pizza = null;
      switch (type) {
```

```
            case MOZZARELLA -> pizza = new Mozzarella_UA();
            case BACON -> pizza = new Bacon_UA();
            case CHILI_PEPPER -> pizza = new Chili_Pepper_UA();
        }
        return pizza;
    }
}
class PizzaShop_EU extends PizzaShop {
    @Override
    public Pizza createPizza(PizzaType type) {
        Pizza pizza = null;
        switch (type) {
            case MOZZARELLA -> pizza = new Mozzarella_EU();
            case BACON -> pizza = new Bacon_EU();
            case CHILI_PEPPER -> pizza = new Chili_Pepper_EU();
        }
        return pizza;
    }
}
class Test {
    public static void main(String[] args) {
        PizzaShop pizzaShop_UA = new PizzaShop_UA();
        pizzaShop_UA.orderPizza(PizzaType.MOZZARELLA);
        pizzaShop_UA.orderPizza(PizzaType.BACON);
        pizzaShop_UA.orderPizza(PizzaType.CHILI_PEPPER);
        PizzaShop pizzaShop_EU = new PizzaShop_EU();
        pizzaShop_EU.orderPizza(PizzaType.MOZZARELLA);
        pizzaShop_EU.orderPizza(PizzaType.BACON);
        pizzaShop_EU.orderPizza(PizzaType.CHILI_PEPPER);
    }
}
```

Патерн ***Abstract Factory*** у певному сенсі пов’язаний з розглянутою вище моделлю проєктування. Цей новий патерн пропонує створити інтерфейс або абстрактний клас для створення родин пов’язаних (або залежних) об’єктів, але без визначення їх конкретних підкласів. Сказане означає, що ***Abstract Factory*** дозволяє класу повертати фабрику класів. Отже, патерн ***Abstract Factory*** на один рівень вищий, ніж патерн ***Factory,*** який ми розглянули вище. ***Abstract Factory*** полегшує обмін між об’єктами і сприяє узгодженості між ними.

Пояснимо загальні речі, висловлені вище, на конкретному прикладі. Припустимо, деякий підприємець вирішив взяти під свій контроль ринок автомобілів. Як це зробити? Можна створити власну марку автомобіля і своє широкопрофільне виробництво. Проте у цьому випадку доведеться конкурувати з гігантами, як-от Mersedes, Opel або Ford. Дуже малоймовірно, що згаданий підприємець буде переможцем у цій боротьбі. Набагато потужнішим рішенням буде скупити заводи всіх цих компаній та продовжити випускати автомобілі під їх власними марками. Іншими словами – створити холдинг. Саме цей холдинг і буде Abstract Factory. Отриману структуру можна ще умовно назвати «фабрикою фабрик». У нашій програмі Абстрактна фабрика (холдинг) буде представлена інтерфейсом або абстрактним класом. Підприємства, що входять у холдинг, представлені класами, що реалізують цей інтерфейс.

```
public interface CarsFactory { }
public class ToyotaFactory implements CarsFactory {}
public class FordFactory implements CarsFactory {}
```

Керівник холдингу дає розпорядження випускати автомобілі з двома типами кузова – седан і купе. Японія буде випускати ToyotaSedan і ToyotaCoupe, а США – FordSedan і FordCoupe. Отже, інтерфейс CarsFactory буде містити такі два методи:

```
public interface CarsFactory {
    Sedan createSedan();
    Coupe createCoupe();
}
```

Відповідно, в дочірніх класах інтерфейсу CarsFactory ці методи теж повинні бути реалізовані.

```
public class ToyotaFactory implements CarsFactory {
   @Override
   public Sedan createSedan() {
      return new  ToyotaSedan();
   }

   @Override
   public Coupe createCoupe() {
      return new ToyotaCoupe();
   }
}
public class FordFactory implements CarsFactory {
   @Override
   public Sedan createSedan() {
```

```
        return new FordSedan();
    }
    @Override
    public Coupe createCoupe() {
        return new FordCoupe();
    }
}
```

Значення, що повертаються в методах, мають типи Sedan і Coupe. Як видно з наведеного коду, у програмі повинні з'явитися деякі сутності, що описують конкретні типи кузова, а саме Sedan і Coupe. Такою сутністю будуть інтерфейси.

```
public interface Sedan {}
public interface Coupe {}
```

Конкретні марки автомобілів, створюваних на тій чи іншій фабриці, повинні відповідати заданим типам кузовів, що реалізується так:

```
public class ToyotaCoupe implements Coupe {
    public ToyotaCoupe() {
        System.out.println("Create ToyotaCoupe");
    }
}
public class ToyotaSedan implements Sedan {
    public ToyotaSedan() {
        System.out.println("Create ToyotaSedan");
    }
}
public class FordCoupe implements Coupe {
    public FordCoupe () {
        System.out.println("Create FordCoupe");
    }
}
public class FordSedan implements Sedan {
    public FordSedan() {
        System.out.println("Create FordSedan");
    }
}
```

Отже, ***Abstract Factory*** готова. Загалом патерни проєктування – дуже зручний та ефективний механізм, створений з метою моделювання комп'ютерних програм будь-якої складності. Як приклад широкого використання патернів наведемо корпоративну платформу Java EE, що являє собою стандартизоване програмне середовище для створення корпоративних та вебзастосунків. Зас-

тосунки цього типу мають багаторівневу архітектуру. Все вищеозначене вимагає застосування патернів як стратегії організації та моделювання таких складних програмних середовищ.

3.5. Уніфікована мова моделювання

Процес розвитку уніфікованої мови моделювання (unified modeling language – UML) стартував наприкінці XX сторіччя. Однією з цілей UML було забезпечити взаємодію між програмістами шляхом створення загальної, але специфічної мовної конструкції. У перспективі таку структуру можна було б використовувати для розвитку і створення комп'ютерних програм. Взагалі ситуація нагадує історію із програмними моделями типу патернів, які ми розглядали у попередньому підрозділі. Отож, розробники UML створили уніфікований стандарт моделювання, до якого IT-фахівці прагнули роками. Тепер, використовуючи UML, IT-спеціалісти можуть формалізувати і розуміти складні програмні структури. Інакше кажучи, UML стала об'єднуючою, стандартизованою мовою створення програмних продуктів, що дає можливість моделювати комп'ютерні програми.

UML використовує специфічну діаграмну техніку, що дає змогу зрозуміти принципи створення програм. Розміщенням стандартних діаграм UML у робочому програмному коді створюється реальна можливість полегшити роботу програміста.

У мові UML є 12 типів діаграм:

- 5 типів діаграм описують поведінкові аспекти системи;
- 4 діаграми характеризують статичну структуру додатку;
- 3 діаграми мають назву діаграм реалізації.

Деякі з видів діаграм є специфічними. Найбільш доступними з них є:

- Class diagram (діаграма класів);
- Activity diagram (діаграма активностей);
- Use-case diagram (діаграма прецедентів);
- Sequence diagram (діаграма послідовності);
- Deployment diagram (діаграма розгортання);
- Object diagram (діаграма об'єктів);
- Collaboration diagram (діаграма співробітництва);
- Statechart diagram (діаграма станів).

Class diagram представлена схематично на рис. 3.9 та являє собою набір загальних атрибутів та операцій.

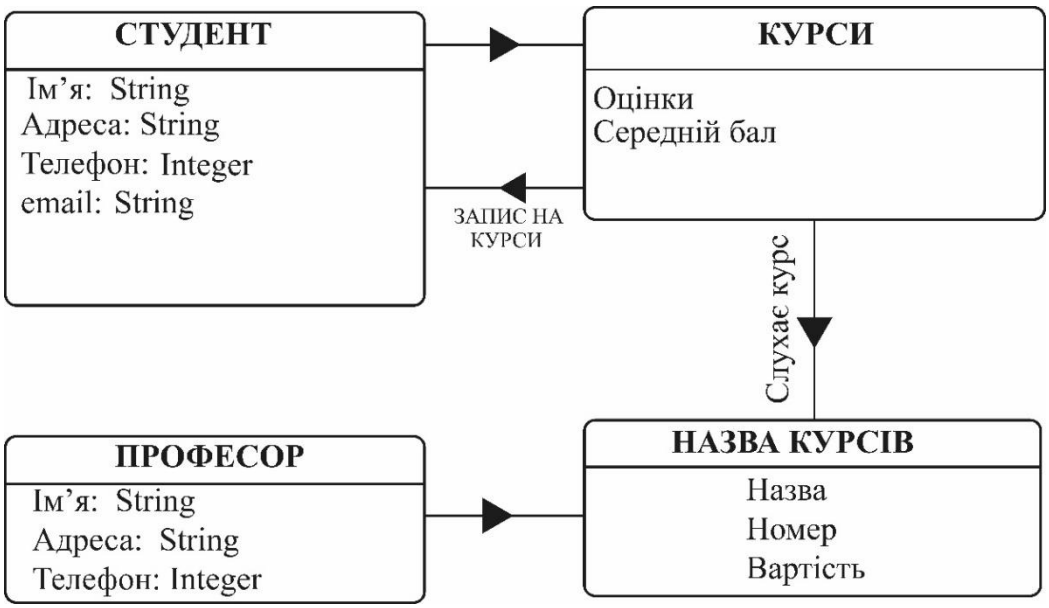


Рис. 3.9. Специфіка діаграми класів

Фактично діаграма класів – це набір статичних елементів моделі. Вона дає найбільш повне і розгорнуте уявлення про зв'язки в програмному коді, функціональність та інформацію про окремі компоненти. Додатки генеруються часто саме з діаграми класів. Для класу **СТУДЕНТ** сформовані записи, що містять атрибути: ім'я, адреса, телефон, email. Схематично зображені зв'язки цієї сутності з іншими: проходженням курсу, який курс слухає, хто професор. У цьому прикладі також представляються функції, які можуть бути застосовані до атрибута **ПРОФЕСОР.**

Activity diagram досить часто використовується на практиці (рис. 3.10). Діаграма подібного типу описує динамічні аспекти поведінки системи, використовуючи блок-схеми, що наочно відображають ті чи інші процеси та структуру логіки процедур. Взагалі, йдеться про процедуру алгоритму дій щодо взаємодії елементів у межах системи або взаємодії між кількома системами. Приклад такої діаграми доцільно навести для інтернет-магазину. У цьому випадку діаграма Activity diagram для сайта магазину максимально доступно пояснює, які функціонують координаційні зв'язки у системі. Рушійним елементом системи у цьому випадку є **Actor** (у якості такого об'єкта у нашому випадку виступає покупець), що відкрив сайт та здійснює процедуру замовлення (рис. 3.10). Далі відбувається розгалуження, а саме: перевіряється, чи є **Actor** оптовиком («Так» або «Ні»). Якщо «Ні», то замовлення відправляється в **retail.CRM.** Дамо пояснення наведеному вище запису. Слово *retail* перекладається як «роздрібна торгівля». Визначення **CRM** розшифровується як **Customer Relationship Management,** що означає «управління відносинами з клієнтами» і стосується всіх стратегій, методів, інструментів і технологій, які використовує

бізнес для розвитку, утримання і залучення клієнтів. **Customer Relationship Management** – це особливий підхід до ведення бізнесу, за якого на перше місце діяльності компанії ставиться клієнт. Основна мета впровадження CRM-стратегії – створення єдиної системи з залучення нових і розвитку наявних клієнтів. Управляти відносинами означає залучати нових клієнтів, а нейтральних покупців перетворювати в лояльних клієнтів і з постійних клієнтів формувати бізнес-партнерів.

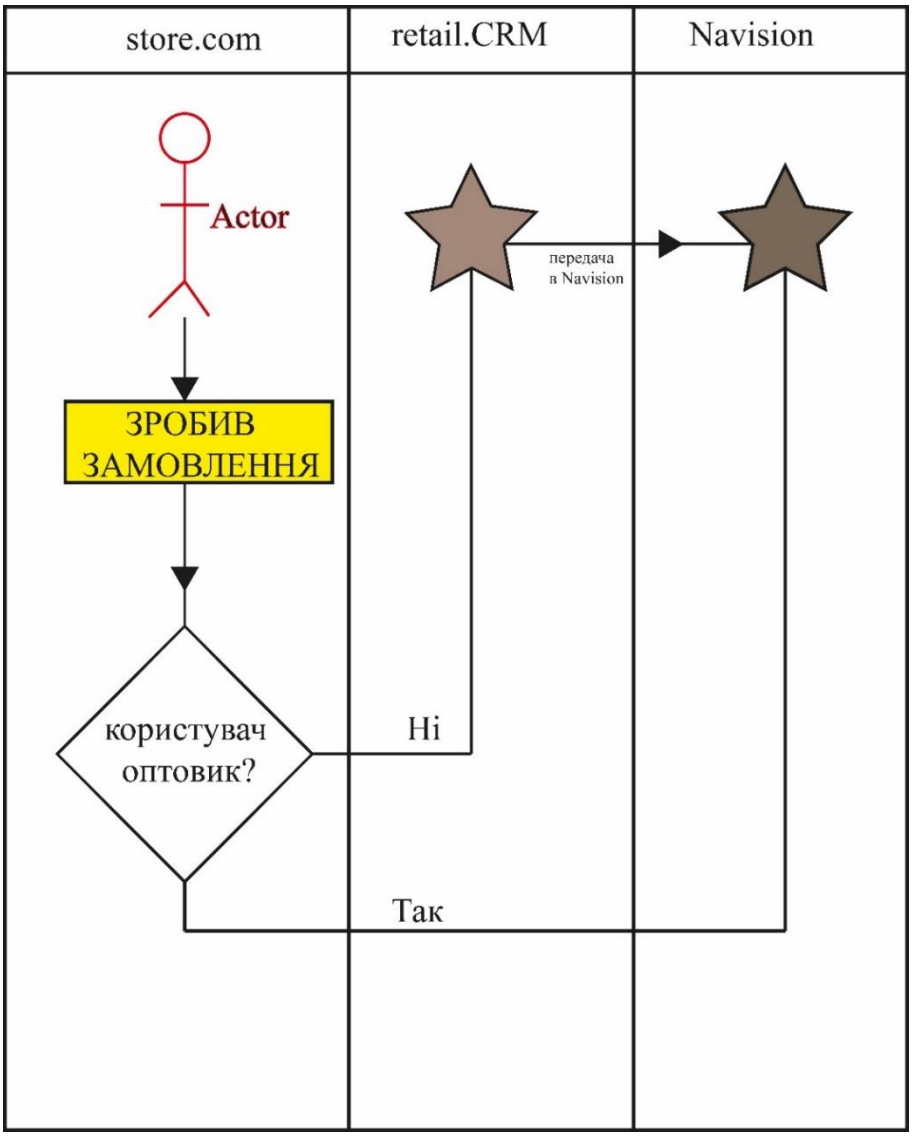


Рис. 3.10. Activity diagram (діаграма активностей), спроєктована для інтернет-магазину

Продовжимо аналіз логіки згідно з рис. 3.10. Якщо логічний блок видає «Так», то його замовлення потрапляє в **Navision** – програма для управління підприємством малого та середнього розміру. Зокрема, до складу Microsoft Dynamics NAV 2016 входять додаткові оптимізовані програми для пристроїв усіх форм-факторів, які працюють під керуванням Android, iOS або Windows. Контроль бізнес-процесів реалізується за допомогою конфігурованих інфор-

маційних панелей Microsoft Power BI, що забезпечують миттєвий доступ до ключових показників ефективності в інтуїтивному середовищі, призначеному для безпечної групової роботи. Реалізовано також інтеграцію з Microsoft Dynamics CRM Online, що дає змогу реалізувати можливості збільшення продажів, покращення обслуговування клієнтів та оптимізації ланцюжка операцій від замовлення до отримання оплати. На рис. 3.10 між **retail.CRM** і **Navision** зображено взаємодію між цими програмними компонентами, зокрема з метою здійснення процедури синхронізації залишків.

Звичайно, що подібні діаграми можна і потрібно доповнювати та розширювати. Такі діаграми стають важливим складником документації і створюють загальне уявлення про роботу системи в цілому.

Use-case diagram (рис. 3.10) містить два важливих атрибути:

1) **Actor** (учасник) – спектр логічно пов'язаних ролей, виконуваних під час взаємодії з прецедентами або сутностями. Учасником **(Actor)** може бути система загалом, людина, підсистема, клас або будь-яка інша сутність;

2) **Use-case** (прецедент) – опис окремого аспекту поведінки системи з погляду користувача. Прецедент не показує, як досягається певний результат, а тільки показує, що саме виконується.

Розберемо класичний приклад студента у бібліотеці, де є два учасники: бібліотекар і студент (рис. 3.11). Прецеденти для студента: пошук книг у електронному каталозі з використанням комп'ютерів зі спеціальним програмним забезпеченням та інстальованою базою даних, пошук книг у тематичному каталозі, пошук книг у алфавітному каталозі і, нарешті, замовлення книг чи іншої навчальної або художньої літератури. Прецеденти для бібліотекаря: робота у читальному залі у плані організації роботи читального залу загалом, видача замовлення студентам, консультації з усіх питань бібліотечного обслуговування (консультації щодо використання електронної пошукової системи, рекомендації книг із теми, заповнення бланків замовлення та ін.).

Отже, Use-case-діаграми спрямовані допомогти командам розробників візуалізувати функціональні вимоги системи. Такі діаграми зазвичай показують спектр варіантів використання – або всі випадки використання для всієї системи, або лише для окремої групи. Щоб показати варіант використання на Use-case діаграми, потрібно намалювати овал посередині діаграми та ввести назву варіанта використання у центрі овалу або під ним (рис. 3.11). Потрібно також застосовувати просту та зрозумілу діаграмну техніку для зображення взаємозв'язків між «акторами» та варіантами використання.

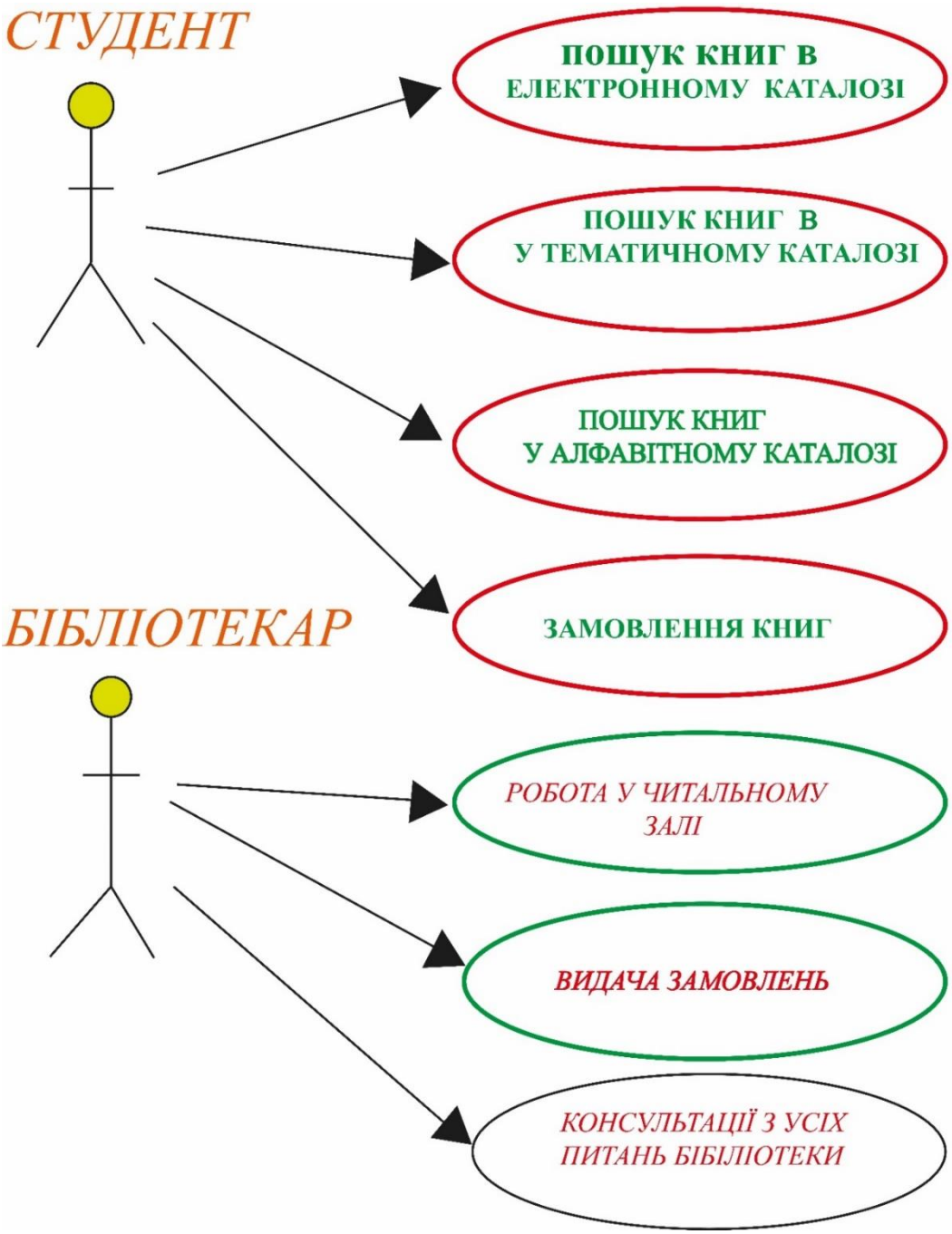


Рис. 3.11. Типовий варіант Use-case diagram

Use-case діаграма зазвичай використовується для того, щоб пов’язати функції високого рівня системи з областю застосування системи. Подивившись на діаграму Use-case (рис. 3.11), можна легко визначити функції, які розглядувана система може виконувати. Зокрема, ця система дає змогу бібліотекарю вести статистику відвідування студентами читального залу та статистику використання книг.

Sequence Diagram (діаграма послідовності) використовується для уточнення діаграм прецедентів та описує деякі поведінкові аспекти системи. Така діаграма відображає взаємодію об’єктів у динамічному режимі, інакше – у часі. Водно-

час інформація набуває вигляду повідомлень, а взаємодія об’єктів передбачає обмін цими повідомленнями.

Deployment Diagram (діаграма розгортання) відображає графічне представлення інфраструктури, для якої буде застосовано додаток, а саме: структуру системи загалом та розподіл її компонентів по вузлах. Зображуються також з’єднання – маршрути передачі даних між вузлами. Діаграма допомагає ефективніше організувати компоненти, від чого залежить продуктивність системи, а також вирішити допоміжні проблеми, зокрема пов’язані з безпекою.

Як бачимо, на перший погляд банальний набір фігур і стрілок може значно спростити вирішення складних завдань у програмуванні, допомогти під час вибору оптимального рішення і під час розробки технічної документації. Можемо зробити такі висновки:

- будувати діаграми нескладно;
- діаграми легко зчитуються та зрозуміло спроєктовані;
- діаграми – чудовий інструмент для реалізації архітектури системи загалом;
- необхідні для документування будь-якої нетривіальної системи та дають змогу зрозуміти зв’язки між її модулями.

РОЗДІЛ 4
ІМІТАЦІЙНЕ МОДЕЛЮВАННЯ

4.1. Види імітаційного моделювання

Моделювання є ефективним і надзвичайно дієвим засобом вивчення об’єктів, явищ і процесів реального світу. Цю процедуру використовують під час вивчення реальних об’єктів у випадку, коли ці об’єкти безпосередньо важко або неможливо дослідити або процес дослідження вимагає великих затрат коштів, часу та ресурсів. Отже, моделювання використовують для спрощення і здешевлення розробки та оптимізації складних і дорогих систем. Важлива специфіка моделювання полягає у виокремленні тих характеристик системи, які цікавлять науковців, розробників чи дослідників. Саме ця обставина робить моделювання основним і необхідним етапом дослідження будь-яких систем, процесів чи явищ реального світу. Модель обов’язково повинна точно описувати особливості системи та відтворювати ті з них, вивченню яких приділяється увага. Наведемо приклад. Нехай потрібно удосконалити надійність автомобіля, додавши додаткові засоби для гарантування безпеки водія у випадку дорожньо-транспортної пригоди. З цією метою проводять так званий краш-тест (рис. 4.1).

Рис. 4.1. Лобовий краш-тест автомобіля Hyundai Solaris (2017)

На місці водія розташовують манекен. Під час проведення краш-тесту манекен отримує пошкодження у певній своїй частині. Значить, у цьому напрямі повинні працювати конструктори, щоб унеможливити виникнення пошкоджень, коли за кермом автомобіля буде знаходитись жива людина. Треба зазначити, що манекен, використаний для проведення краш-тесту, повинен володіти всього однією особливістю, що поєднує його з людиною, – його фізико-механічні характеристики повинні строго відповідати таким для людини. Все інше неважливо.

Деякі сучасні методики моделювання дають змогу на підставі заданих параметрів системи імітувати її поведінку в часі та отримувати необхідні характеристики. Загалом моделювання дає змогу отримати точні результати лише у випадку, коли модель строго відтворює модельовану характеристику системи. Тут необхідно звернути увагу на те, що розрахувати аналітично всі можливі варіанти поведінки системи буває вкрай складно, а іноді просто неможливо. Наприклад, нехай потрібно розрахувати взаємодію між атомами та їх електронами у молекулі вуглекислого газу CO_2 та отримати спектр збуджених електронних станів. На перший погляд система сама по собі не складна. Але фактично описати квантово-механічну взаємодію між електронами та ядрами у цій молекулі нереально. Саме тому фізики та хіміки застосовують різні моделі з метою отримання наближених розрахунків, що моделюють поведінку такого об’єкта у збудженому стані.

Масштабний розвиток комп’ютерних технологій дав змогу розширити діапазон можливостей моделювання реальних об’єктів. З’явилися нові специфічні методи та технології, що дають змогу здійснювати комп’ютерне моделювати складних об’єктів та процесів у різних сферах людської діяльності – економіці, промисловості, охороні здоров’я, соціальній сфері, науці та інших областях. Поява нових видів моделювання привела до створення нового типу комп’ютерних моделей – імітаційних. Під імітаційним моделюванням розуміється розробка моделі системи у вигляді програми для комп’ютера. Водночас різноманітні «експерименти» проводяться з програмою, а не з реальною системою чи об’єктом. Це зручно, дешево, ефективно та надзвичайно дієво. Отримавши багатократно перевірену комп’ютерну модель, можна застосувати її у реальних умовах. Наведемо приклад. Нехай поставлена задача – розробити комп’ютерно-інформаційну систему під назвою «ВИБОРИ». Ідеться про те, що населення України буде брати участь у виборах Президента чи депутатів Верховної Ради не на виборчих дільницях, а через електронні засоби (комп’ютери, мобільні гаджети та ін.). Створивши комп’ютерну модель зазначеної вище процедури, можна досягти важливих результатів, а саме: 1) суттєвого скорочення витрат на організацію виборів; 2) збільшення кількості виборців, що братимуть участь у таких акціях, а значить – досягнення більшої дієвості та ефективності. Отже,

можна зробити висновок, що імітаційне моделювання – досить ефективний механізм і застосовується для отримання більш точних результатів та зменшення витрат під час роботи з системою. До того ж часто буває неможливо побудувати аналітичну модель системи, яка б враховувала всі причинно-наслідкові зв'язки та ймовірнісні змінні. Комп'ютерно-імітаційна модель актуальна також, коли необхідно імітувати поведінку системи у часі, розглядаючи різні можливі сценарії її розвитку під час зміни зовнішніх і внутрішніх умов. Отже, імітаційне моделювання – це високорівнева інформаційна технологія із застосуванням комп'ютерів, яка найчастіше використовується під час моделювання складних систем.

Поява нових сучасних програмних продуктів істотно спрощує вимоги до розробника моделі, відкриваючи широкі можливості для фахівців різних спеціальностей, які не мають навичок програмування. Це створює можливості для розробки моделей як у різних галузях, так і для досить складних систем. Вищесказане дає надзвичайно широкі можливості щодо застосування методів імітаційного моделювання стосовно соціальних явищ, освітньої діяльності, під час навчання управлінських кадрів тощо. Однак існує певна небезпека у розробці та інтерпретації результатів імітаційних моделей складних систем. Адже відомо, що модель є спрощеним відображенням реальності – це менш детальне, менш складне, менш докладне відтворення реального наявного об'єкта, системи або феномену чи процесу. Водночас, коли ми говоримо про моделювання соціальних явищ, модель – це не просто спрощення реальності, а відображення реальності через призму певного теоретичного підходу або думки експерта. Імітаційне моделювання умовно може бути представлено різними видами або напрямами, що відповідно мають свої методології (рис. 4.2). Охарактеризуємо тепер наявні види імітаційного моделювання. *Статистичне моделювання* є різновидом імітаційного моделювання. Спочатку воно з'явилося в теорії випадкових процесів та математичної статистики як спосіб обчислення статистичних характеристик випадкових процесів шляхом багаторазового відтворення перебігу процесу за допомогою моделі цього процесу. Цей підхід до дослідження реального процесу був названий методом статистичних випробувань (методом Монте-Карло). Моделі тут будуються для явищ і систем об'єктів; входи та функціональні співвідношення між різними компонентами містять елементи повністю випадкових процесів, що підпорядковуються ймовірнісним законам.

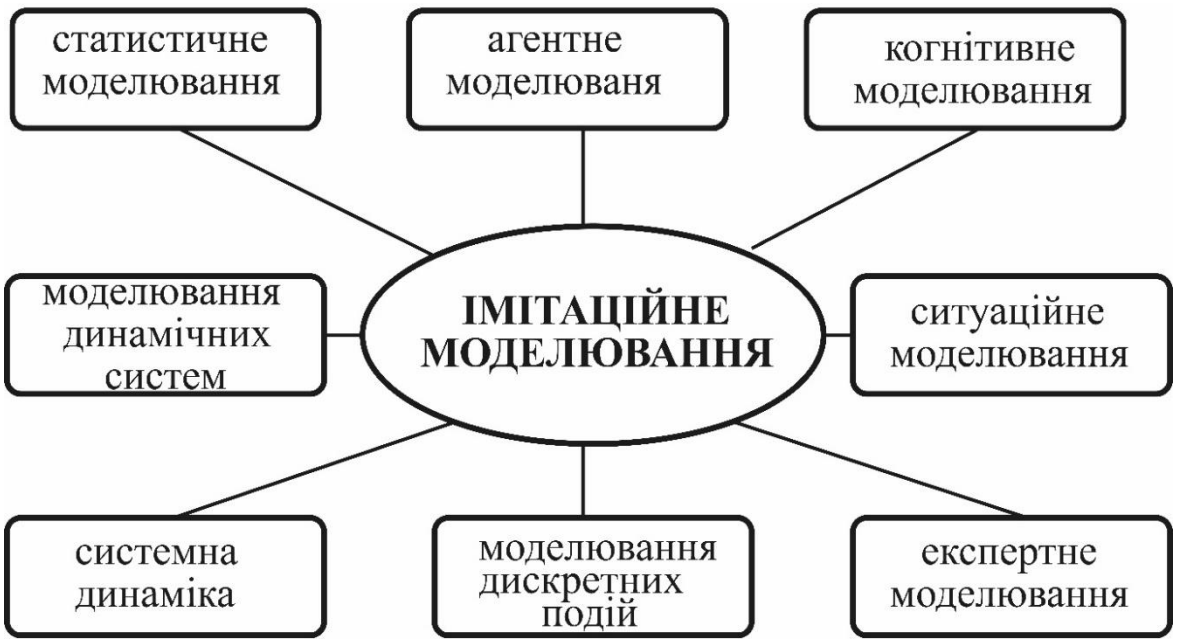


Рис. 4.2. Види імітаційного моделювання

Найбільш ефективним засобом імітаційного моделювання є реалізація ймовірнісної моделі реального об'єкта, здійснювана на комп'ютері. Така процедура дає змогу досліджувати модель як у певні моменти часу, так і протягом тривалих періодів часу. Для знаходження стійких рішень під час чисельного статистичного моделювання потрібне його багаторазове відтворення з наступною статистичною обробкою. Тут проводиться імітація впливу численних випадкових чинників на різні елементи моделі. Кожен такий вплив на процес у моделі представляється у вигляді випадкового явища за допомогою процедури, що дає ймовірнісний результат. Безліч таких реалізацій під час одного варіанта імітації дає одну реалізацію процесу. Потім обчислюються середні статистичні характеристики з багатьох проведених імітацій. Статистичне моделювання залежно від сфери застосування має кілька напрямів (рис. 4.3).

Рис. 4.3. Класифікація видів статистичного моделювання

Ймовірнісне моделювання, в основі якого лежить метод Монте-Карло, спрямоване на розв'язання складних математичних проблем, а саме: обчислення нестандартних інтегралів, розв'язання інтегро-диференціальних рівнянь тощо.

Метод Монте-Карло можна назвати ймовірнісно-імітаційним моделюванням. Фактично мова йдеться про використання теорії ймовірностей для побудови імітаційних моделей у різних науках, у моделях масового обслуговування та прикладних алгоритмах.

Статистичне моделювання (СМ) орієнтоване на застосування законів математичної статистики для оцінювання, прогнозування та оптимізації систем і управлінських рішень. СМ найчастіше застосовується в економічних та соціальних науках. Під час розробки СМ-моделі вирішуються завдання створення алгоритмів для розв'язання складних проблем, які не можуть бути вирішені іншими засобами. Водночас поставлена проблема формулюється в аналітичному вигляді. До того ж проводяться статистичні дослідження на основі законів статистики з метою формування теоретичної концепції. Сама по собі подібна формалізація є важливим етапом моделювання і виступає як теоретична концепція процедури СМ. Результатом застосування СМ є можливість здійснення прогнозу. Можливість прогнозування є вагомим результатом застосування процедури СМ.

Зазвичай системи, що моделюються, є змінними у часі, інакше – динамічними. Під динамічною системою будемо розуміти будь-який об'єкт, процес або явище, для якого однозначно визначено поняття стану як сукупності деяких показників чи параметрів та задано закон, за яким протікають процеси. Інакше кажучи, описується зміна початкового стану з плином часу. Динамічними об'єктами можуть бути фізичні, хімічні, біологічні, механічні, виробничі і взагалі будь-які природні об'єкти. Динамічні системи описуються різними засобами: рівняннями руху, диференційними рівняннями, графічними образами та ін. В основі теорії моделювання динамічних систем та побудови об'єктно-орієнтованих моделей лежить агрегативний підхід. Такий підхід був започаткований вченими у 70-х роках XX сторіччя. Водночас складна система представлялася у вигляді агрегату, що має спектр вихідних, вхідних та керуючих сигналів. Математично подібний агрегат можна задати сукупністю множин T, X, Г, У, Z з використанням операторів H і G, де T – фактор часу, а X, Г, У – множини вхідних, керуючих та вихідних сигналів відповідно; H і G – оператори переходів. Цей підхід широко використовується під час дослідження складних управлінських систем, до яких належать автоматичні системи управління (АСУ). Агрегативні системи дають змогу описати широке коло об'єктів з відображенням системного характеру цих складних структур. Існує також можливість декомпозиції складної системи на певну кількість підсистем зі збереженням

зв’язків між ними. Основним об’єктом вивчення є системи управління складними динамічними об’єктами та їх елементами. Математичні моделі АСУ представляються зазвичай у вигляді рівнянь динаміки, які записуються у формі диференціальних, інтегральних чи інтегро-диференціальних рівнянь. Завдяки вищеозначеним особливостям згадані моделі знайшли широке застосування не тільки в інженерній практиці. Опис динамічних систем і елементів у просторі станів дає змогу перейти до рівнянь для моделювання на комп’ютерах, а також провести моделювання АСУ у вигляді структурних схем.

Для моделювання динамічних систем використовуються такі програми: VISSIM, SIMULINK, MATLAB, PowerSim, Multisim, LabView, Easy5, MvStudium. Особливу роль займає система дискретно-подійного програмування Global Purpose Simulation System (GPSS). Основний об’єкт у цій системі – пасивна заявка на обслуговування, яка може взаємодіяти із працівниками, клієнтами та покупцями. Також подібна заявка може стосуватись різних деталей системи.

Загалом GPSS-модель можна розглядати як специфічну систему обслуговування заявок. Аналітичні результати функціонування такої моделі розглядаються в теорії масового обслуговування. Цей підхід використовується для опису переходу системи з одного стану до іншого дискретно у вигляді події. Схема побудови імітаційних моделей, що пропонує моделювати реальні процеси такими подіями, саме і називається *дискретно-подійним* моделюванням. Цей вид моделювання найчастіше використовується для виробничих процесів, де динаміка системи може бути представлена як послідовність операцій. GPSS-моделювання застосовується, зокрема, в теорії масового обслуговування. Водночас досліджується широкий клас випадкових процесів у системах поширення інформації та у різних галузях масового обслуговування.

Дамо тепер коротку характеристику *системній динаміці* (рис. 4.2). Це така технологія моделювання, де для досліджуваної системи будуються графічні діаграми причинних зв’язків. Досліджуються, зокрема, і дієві впливи одних параметрів на інші в часі, а потім створена на основі цих діаграм модель імітується на комп’ютері. Метод системної динаміки почав розроблятись Дж. Форрестером у 1950-х роках і зараз використовується для аналізу складних систем з від’ємними зворотними зв’язками. Такий вид моделювання, на відміну від інших парадигм моделювання, допомагає зрозуміти причинно-наслідкові зв’язки між об’єктами і явищами. За допомогою системної динаміки будують моделі виробництва, світової та регіональної економіки а також екології чи поширення епідемій. Отож, системна динаміка – це такий специфічний підхід до процесу імітаційного моделювання, що своїми інструментами досліджує динаміку складних систем. Також системна динаміка – це метод моделювання, що використовується для створення точних комп’ютерних моделей. Далі такі моделі можуть

ефективно використовуватись для реалізації більш ефективної організації роботи системи. Переважний вектор спрямування системної динаміки – довгострокові та стратегічні моделі. Водночас події, компоненти, люди та інші дискретні елементи представлені в моделях системної динаміки не як окремі складники, а як система загалом.

Тепер зупинимось коротко на *агентному моделюванні*. Це специфічний різновид імітаційного моделювання – метод, що дає змогу досліджувати роботу децентралізованих агентів і те, як їх поведінка визначає динаміку всієї системи загалом. На відміну від системної динаміки, аналітик визначає поведінку агентів на індивідуальному рівні, а глобальна поведінка проявляється як результат діяльності численних агентів. Основною дійовою одиницею агентного моделювання є сам агент. Це така специфічна сутність, що володіє активністю та автономною поведінкою. Агент може приймати рішення відповідно до деякого набору правил та взаємодіяти з оточенням. Агент також володіє властивістю самостійно вдосконалюватись. Агентами можуть бути: люди (покупці, водії, споживачі, жителі, працівники, пацієнти, клієнти, солдати, токарі, слюсарі та ін.); машини (літаки, автомобілі, підводні човни, вагони, гвинтокрили, верстати); об’єкти інтелектуальної власності (винаходи, статті, інновації, гіпотези, інвестиції); організації (бізнес-структури, компанії, політичні партії). Крім агентів, потрібно фокусувати увагу і на середовищі. Це деякий простір, у якому функціонують агенти. Така сутність характеризується деякими станами і факторами. Агенти знаходяться в певному місці цього простору з можливістю орієнтування та пересування в ньому. Існують певні правила взаємодії між агентами. Ці правила являють собою закони взаємодії агентів у навколишньому середовищі. Завдання імітаційного моделювання в агентному підході полягає у визначенні характеристик стану агентів. Проводиться також вивчення поведінки агентів у разі різних ситуацій взаємодії та динамічних станів середовища. Через вивчення поведінки безлічі агентів у певному просторі здійснюється прогнозування поведінки системи загалом. Мета створення агентних моделей – отримати уявлення про загальну поведінку системи з огляду на припущення про індивідуальні особливості її окремих активних об’єктів та взаємодію цих об’єктів у системі. Агентний підхід дає змогу досліджувати завдання колективної взаємодії, ефективно вирішувати питання прогнозування. Агентні системи дають змогу досліджувати процеси самоорганізації та дають змогу природного опису складних систем і володіють високою гнучкістю.

Когнітивне моделювання (рис. 4.2) призначене для аналізу та прийняття рішень у нечітко визначених ситуаціях. Цей вид моделювання був запропонований американським дослідником Р. Аксельродом. Спочатку когнітивний аналіз формувався в межах соціальної психології, а саме – когнітивізму. Ця

наука займається вивченням процесів сприйняття та пізнання. Застосування досліджень соціальної психології у теорії управління призвело до формування специфічної галузі знань – когнітології. Ця наука фокусує свої дослідження на проблемах управління. Загалом під когнітивною методологією розуміється широкий спектр технологій формалізації інтелектуальних систем, функціонування знань та прийняття рішень. Когнітивні інформаційні технології являють собою сукупність засобів, прийомів та процесів, що здійснюються у певній послідовності. Такі технології дають змогу перетворювати вхідну інформацію у варіанти управлінського рішення. У тріаді «теорія – натурний експеримент – машинний імітаційний експеримент», останній блок є науковим методом, що швидко розвивається та застосовується практично у всіх високотехнологічних галузях для моделювання надскладних систем. Когнітивна наука в широкому сенсі слова – сукупність наукових напрямів із формування знань про системи. Сьогодні когнітивний підхід є напрямом дослідження великих систем, до яких належить соціально-економічні, політичні та екологічні системи. Методологія когнітивного моделювання розвивається у напрямі вдосконалення апарату аналізу, моделювання та пошуку рішень у неформалізованих та слабо структурованих ситуаціях за відсутності або неповної інформації про процеси, що відбуваються в таких системах у разі факторів впливу швидких змін.

Охарактеризуємо тепер *ситуаційне моделювання* (СМ) – такий напрям досліджень та прийняття рішень, що інтенсивно розвивається та вдосконалюється. СМ – метод управління складними технічними та організаційними системами, заснований на ідеях теорії штучного інтелекту, завдяки якому знання про об’єкт управління та способи керування ним здійснюється на рівні логіко-лінгвістичних моделей. СМ використовує навчання та узагальнення як основні процедури під час побудови схем управління та реалізації багатокрокових рішень. СМ можна розглядати також як метод дослідження, що включає в себе побудову моделі реальної системи. СМ проводить із системою різноманітні уявні експерименти: прогнозування напрямів її розвитку та імітації рішень з управління ситуацією з метою вибору оптимального варіанта. Для дослідження ситуацій різного виду розробляють комп’ютерні, аналітичні та імітаційні моделі для прикладних застосувань: ігрові моделі для бізнесу, виробництва, суспільства чи освіти. За допомогою таких моделей можна вирішувати конфліктні ситуації та моделювати поведінку об’єкта (системи) в різних ситуаціях. Особливо цінного практичного значення набуває створення багатоваріантних ситуаційних моделей як перспективної, так і ретроспективної спрямованості. Багатоваріантне ситуаційне моделювання надає його суб’єкту широкий спектр можливостей. Відбувається багатоваріантний вибір – обрання того чи іншого тактичного прийому або застосування того чи іншого варіанта. Таке багатоваріантне моде-

лювання постійно створює потребу шукати альтернативи і на цій основі знаходити кращі рішення та засоби їх реалізації. Основними перевагами багатоваріантного моделювання є його багатофакторність і багатофункціональність, гнучкість і продуктивність. Розробка такої комп'ютерної імітаційної моделі для прийняття рішення на багатоваріантному ситуаційному полі є одним з важливих напрямів імітаційних досліджень. Логічна послідовність процедури ситуаційного моделювання може бути зведена до таких його етапів: а) формулювання завдань моделювання; б) моделювання ситуації; в) абстрагування від несуттєвих обставин; г) діагностика ситуації; д) облік динамічних факторів; е) визначення спектру можливих альтернативних рішень; ж) прогони різних варіантів моделей і на цій основі вибір оптимального варіанта.

Зауважимо, що завжди процес ситуаційного моделювання включає у себе вищезгадані етапи. Для формалізації та опису моделей використовують різні підходи, зокрема дискретні ситуаційні мережі. Одним із напрямків ситуаційного моделювання є розробка комп'ютерних та ігрових імітаційних тренажерів, що дають змогу проводити навчання у виробничих, транспортних, психологічних, педагогічних та управлінських структурах. В основі цих тренажерів лежить імітаційна модель, яка дає змогу учням застосовувати певний набір інструментів впливу та управління і відтворює реакцію соціальної системи на відповідний керуючий вплив. Такі тренажери (віртуальні ігрові імітаційні моделі) вже набули широкого поширення в зарубіжній практиці. Результатом розвитку напряму ситуаційного моделювання є ситуаційне управління як науковий підхід до аналізу та вирішення завдань ситуаційного відображення інформації. Для цієї мети розроблено спеціальні системи відображення інформації. Системи подібного типу дають змогу розв'язувати проблеми керування динамічними системами, у якості яких можуть виступати штаби армій, де проводяться командно-штабні ігри за участю протидіючих військових формувань. Наочним прикладом високодинамічної широкомасштабної системи є міський трафік, що характеризує рух транспортних засобів по місту: кожної секунди міський трафік змінюється досить помітно.

Отже, дискретні ситуаційні моделі дають змогу здійснювати бієктивне відображення ситуацій типу «система → модель». Отримавши результати такого перетворення, можна застосовувати управлінські рішення у межах поставленої проблеми. Важливість розвитку цього напряму підтверджує та обставина, що у кожному великому місті функціонують ситуаційні центри (СЦ) для фіксації, опрацювання та моделювання різних життєвих явищ (рис. 4.4).

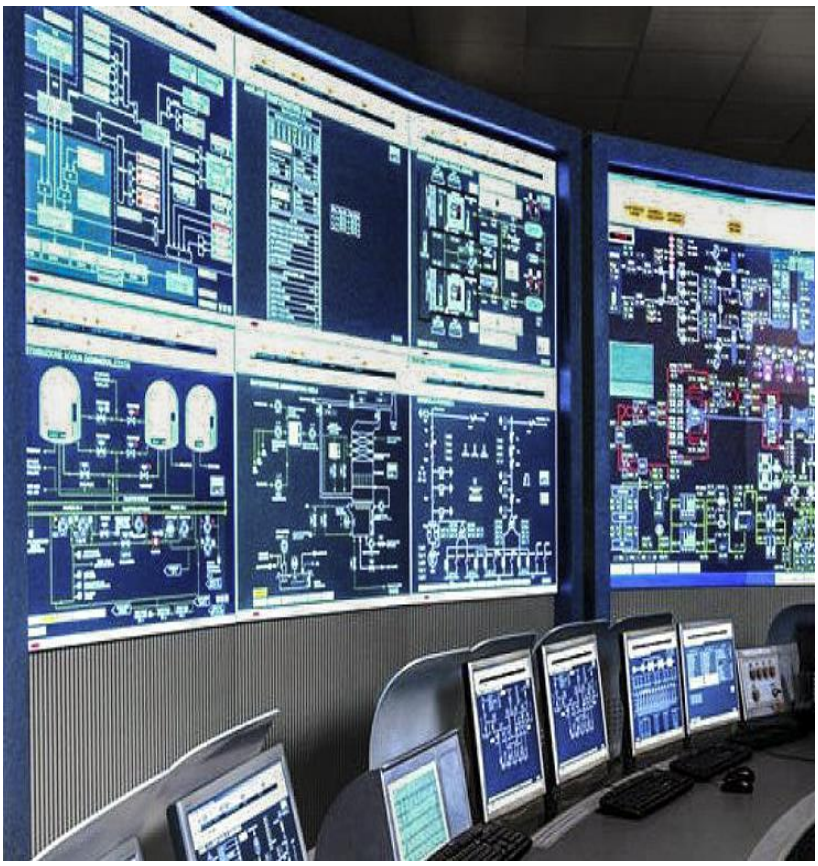


Рис. 4.4. Ситуаційний центр як сукупність робочих місць

СЦ (рис. 4.4) містить у своєму складі: 1) оперативний штаб, спроможний у разі необхідності приймати екстрені рішення; 2) приміщення спостереження та управління, які є центами збору різноманітної інформації з усіх місць системи (країни, регіону, міста тощо). Як видно (рис. 4.4), з технічної сторони СЦ являє собою робочі місця, підключені до мережі. Оператори робочих місць мають можливість отримувати всю вхідну інформацію, аналізувати її та за необхідності представляти керівництву. Відповідно до поставлених СЦ завдань розробляються різні ситуаційні моделі. Зрозуміло, що СЦ можуть мати різний статус, який визначає масштаб завдань і їх специфіку. Інакше кажучи, СЦ можуть бути як оперативного, так і стратегічного рівня. Перед оперативними СЦ стоїть задача збору та аналізу оперативної вхідної інформації. На основі такої проаналізованої та зібраної інформації створюється ситуаційна модель. Модель розглядається штабом. Керівник штабу, який приймає остаточне рішення, має змогу застосувати на практиці створену ситуаційну модель. Саме так діяли СЦ під час коронавірусної епідемії COVID-19. Оперативні СЦ діють у межах великих підприємств чи бізнес-структур. Конкретні завдання таких центрів – створити імітаційну модель, покликану вступити у дію під час екстраординарних подій (повеней, ураганів, пожеж, воєн тощо). Якщо йдеться про СЦ підприємства, компанії або галузі, то у цьому випадку проводиться оперативний контроль та

аналіз стану компанії, що суттєво покращує процес управління. Для ситуаційного моделювання можна ефективно застосувати також засоби когнітивного моделювання.

Стратегічні СЦ розв'язують набагато більш масштабні та складніші проблеми, орієнтовані на реорганізацію масштабних об'єктів, як-от регіон країни, міністерство чи галузь промисловості. Слід згадати, наприклад, реорганізацію міністерства охорони здоров'я України (наказ № 1636 «Про деякі питання реорганізації державних установ МОЗ України» від 07.09.2018 року).

4.2. Метод Монте-Карло

Метод Монте Карло працює з випадковими процесами. По суті, цей метод базується на моделюванні реальних явищ. Побудова моделі стартує зі встановлення функціональних особливостей у модельованій системі. Кількісні закономірності встановлюються шляхом використанням теорії ймовірностей. Отже, цей метод може бути ефективно використаний для вирішення таких задач, де результат залежить від випадкових (стохастичних) процесів. Зокрема, метод можна використовувати для прогнозування курсу валют. Моделювання названим методом дає змогу обчислити спектр випадкових величин, значення яких змінюються випадково. Здійснюючи усереднення отриманих стохастичних значень, отримуємо надійні достовірні величини.

Метод Монте-Карло можна, зокрема, використовувати для прогнозування бізнес-процесів, ступеня економічної динаміки, процесів валідності та деяких інших показників. Схема застосування методу працює приблизно так: ймовірність події визначається шляхом вибору комбінації випадкових значень та отримання усередненого результату. Оскільки мова йдеться про статистично-ймовірнісні дослідження, то симуляцію слід повторити декілька разів.

Яке використовується програмне забезпечення? Досить часто застосовуються таблиці Excel а також спектр спеціалізованих програмних продуктів, розроблених для використання програмістами, фінансовими аналітиками, фізиками, маркетологами тощо.

Імітаційне моделювання за методом Монте-Карло зі сторони суто математичного підходу являє собою процедуру знаходження величини математичного очікування шляхом проведення певної кількості випробувань.

Нехай потрібно оцінити математичне очікування для випадкової величини X. Позначимо його як M(X) = α. Формула розрахунку M(X) записується так:

$$M(X) = \sum_{i=1}^{n} X_i \cdot p_i, \qquad (4.1)$$

де X_i – величина, що набуває спектру значень від 1 до n; p_i – спектр величин ймовірностей, що змінюється в межах від 1 до n.

Отже, технічно моделювання методом Монте-Карло полягає у проведенні серії n випробувань. Внаслідок експерименту отримуємо спектр значень X. Потім розраховується їх середнє арифметичне, яке і буде приблизним значенням α.

Виникає природнє запитання: скільки експериментів потрібно здійснити? Загалом кількість імітаційних експериментів залежить від поставленої мети дослідження. Взагалі процес моделювання у режимі Монте-Карло може повторюватись сотні, тисячі або десятки тисяч разів. Тут важливо зауважити, що чим більше випробувань, тим достовірніший результат буде отримано. Зауважимо, що під час реалізації комп'ютерної симуляції багаторазове повторення операцій не є проблемою.

Метод Монте-Карло володіє спектром переваг і недоліків. До переваг слід віднести такі:

1) універсальність та простота – метод достатньо апробований у багатьох сферах та може застосовуватися до різних типів даних;

2) метод завжди можна застосовувати там, де реалізуються типові математичні методи розрахунків;

3) метод має змогу працювати з різнорідними типами даних.

До недоліків потрібно віднести:

1) великі витрати часу під час проведення значної кількості випробувань;

2) метод не може ефективно працювати з подіями, що характеризуються дуже низькою або дуже високою ймовірністю реалізації;

3) метод через свою стохастичну природу допускає певну похибку, яка частково нейтралізується великою кількістю проведених експериментів.

Отже, метод Монте-Карло ґрунтується на моделюванні випадкових процесів і може успішно застосовується там, де звичні математичні розрахунки можуть дати недостовірні результати.

За допомогою наочних прикладів спробуємо зрозуміти, які завдання можна вирішувати із застосуванням методу Монте-Карло. Припустимо, потрібно розрахувати значення числа π. Розглянемо квадрат зі стороною a. Його площа $S = a^2$, а площа круга, вписаного у цей квадрат, $S_0 = \pi R^2$. Зрозуміло, що $R = a / 2$. За допомогою генератора випадкових чисел усю площу цього квадрата (рис. 4.5) заповнюватимемо випадково згенерованими числами (далі у програмі Лістинг 4.1 у якості такого генератора використовується *Math.random*()). Деякі генеровані значення потраплять у круг (зелена зона на рис. 4.5), а інші – поза ним, але у межі квадрата (червона зона на рис. 4.5).

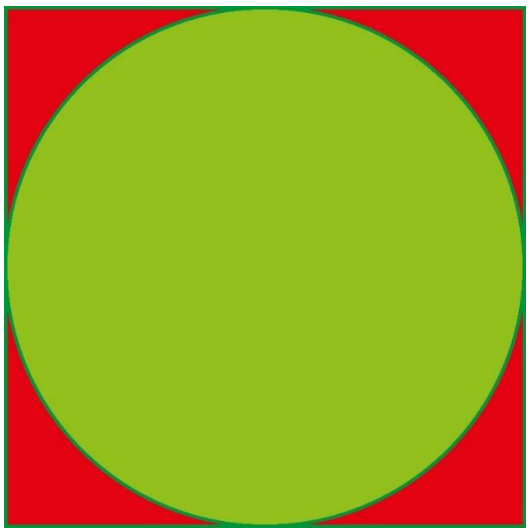


Рис. 4.5. Обчислення числа π методом Монте-Карло

Число значень, що потраплять у межі кола, позначимо K.

Із (4.2) отримуємо:

$$S_0 = \frac{\pi a^2}{4} = \frac{\pi S}{4} \tag{4.2}$$

$$\pi = \frac{4S_0}{S} = \frac{4K}{S} \tag{4.3}$$

Лістинг 4.1

```
import java.util.Random;
public class MonteKarlo {
   public static void main(String[] args) {
      double r = 100;
      double n_0 = 0;
      double n = 100000000;
      for (int i = 0; i < n; i++) {
         double x = -r +  Math.random()*(2*r) ;
         double y = -r +  Math.random()*(2*r) ;
         if(x*x + y*y <r*r)
            n_0++;
      }
      System.out.println("real Pi = 3.141592653");
      System.out.println("calculated Pi = "+ 4*n_0/(n));
   }
}
```

Число генерованих значень сто мільйонів – досить велике. Запустимо програму і отримаємо результат:

real Pi = 3.141592653
calculated Pi = 3.1414368

Як бачимо, результат обчислення дуже точний – розбіжність спостерігається лише у четвертому знаку після коми.

$$\int_0^5 \sqrt{2x^2 + x^3}\,dx \tag{4.4}$$

Тепер використаємо метод Монте-Карло для розрахунку інтеграла Геометричне тлумачення інтеграла полягає у такому: це площа криволінійної трапеції, зверху обмеженої графіком підінтегральної функції, зліва і справа – вертикальними прямими, що проходять через точки меж інтегрування, і знизу – віссю x-ів. Саме така криволінійна трапеція представлена на рис. 4.5. Завдання – обчислити площу цієї криволінійної трапеції, тобто розрахувати інтеграл виду:

$$S = \int_{x_1}^{x_2} f(x)dx\,. \tag{4.5}$$

Розкидаємо у випадковий спосіб, але максимально густо, точки в прямокутнику, зображеному на рис. 4.6. Нехай N – кількість точок, які потрапили у прямокутник; N_s – кількість точок, що потрапили під криву $f(x)$, тобто у зафарбовану ділянку (на рис. 4.6 такі точки зображені кольоровими червоними кружечками). Очевидно, що кількість точок, які розміщені під кривою, порівняно із загальною кількістю точок у прямокутнику, пропорційна площі під $f(x)$, тобто величині інтеграла. Представимо сказане у вигляді такого відношення:

$$\frac{N_S}{N} = \frac{S}{(y_2 - y_1)(x_2 - x_1)} \tag{4.6}$$

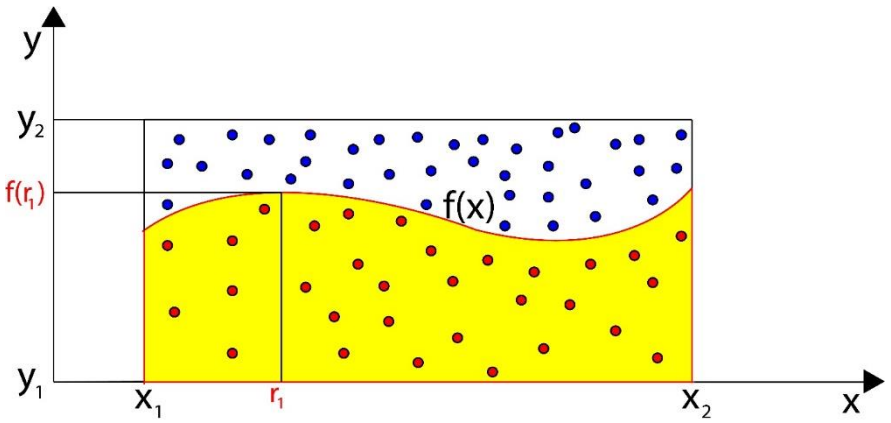


Рис. 4.6. Забарвлена криволінійна трапеція, площа якої обчислюється, заповнена точками червоного кольору. Величина забарвленої площі площа дорівнює інтегралу (4.5). Загальна кількість точок дорівнює кількості точок, що потрапили у прямокутник

Із (4.6) отримуємо вираз для обчислення інтеграла:

$$S=(y_2-y_1)(x_2-x_1)\frac{N_S}{N} \qquad (4.7)$$

У Лістингу 4.2 описаний вище словесний алгоритм програмно представлений у такий спосіб:

Лістинг 4.2

```
package MS;
public class MonteKarloInt {
   public static void main(String[] args) {
      double x1 = 0, x2 = 5, y1 = 0, y2 = 15;
      double NS = 0;
      double N = 100000000;
      for (int i = 0; i < N; i++) {
         double x = x1 + (x2 - x1) * Math.random();
         double y = y1 + (y2 - y1) * Math.random();
         double f = (Math.sqrt(2*Math.pow(x,2)+ Math.pow(x,3)));
         if (f > y) {
            NS++;
         }
       }
           double S = (NS/N) * (x2 - x1) * ((y2 - y1));
            System.out.println(S);
         }
      }
```

Кількість точок для імітаційного експерименту, використана у Лістингу 4.2, досить велика – сто мільйонів, але для комп'ютера це не становить проблеми: обчислення виконуються за час приблизно одну секунду. Результат обчислень для функції 4.5 виглядає так: S = 28,665867. Запустивши програму по-новому, отримаємо наступне значення інтеграла: S = 28,665792. Як бачимо, розбіжність спостерігається лише у четвертому знаку після коми.

Подібні обчислення можна виконувати також іншими методами, зокрема використовуючи таблиці Excel.

4.3. Особливості програми AnyLogic North America LLC як засобу візуального імітаційного моделювання

Аналіз програми імітаційного моделювання AnyLogic розпочнемо із розгляду процедури завантаження цього сервісу. Введемо у вікно браузера **AnyLogic North America LLC** (далі – **AnyLogic**). Відкриється сайт Downloads – AnyLogic Simulation Software. Вибираємо Free DownLoad. Скачуємо програму, наприклад, версію AnyLogic 8.8.3 Personal Learning Edition. Під час запуску програми потрібно ввести свої персональні дані, що займає менше однієї хвилини, і все готово до роботи. Для початку – короткий огляд інтерфейсу програми AnyLogic (рис. 4.7).

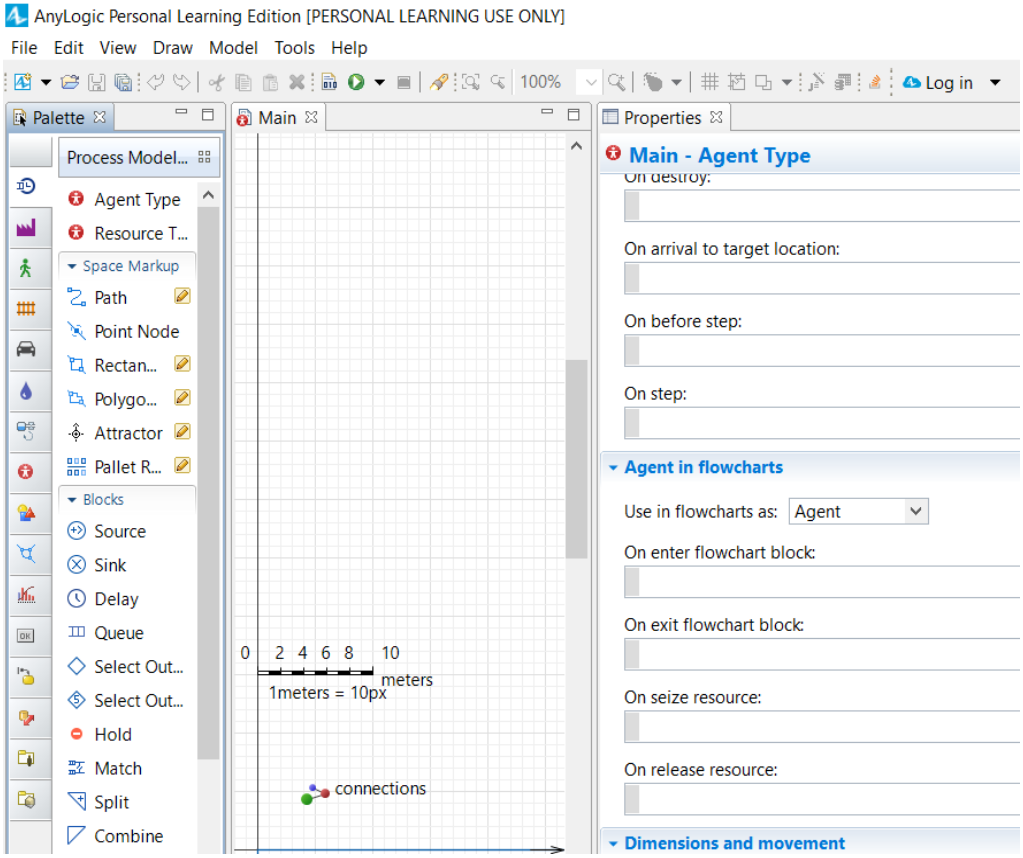


Рис. 4.7. Інтерфейс програми AnyLogic North America LLC

Вікно редактора AnyLogic має такі елементи управління програмою:

- Панель інструментів (File, Edit, View, Draw, Model, Tools, Help), розташована зверху.
- Панель проєктів, розташована зліва.
- Вікно властивостей (Properties) – справа.
- Вікно графічного редактора – у центрі.

Панель інструментів забезпечує швидкий доступ до більшості функцій. Створити нову модель (New model) можна, розкривши випадний список New (рис. 4.8) і натиснувши Model.

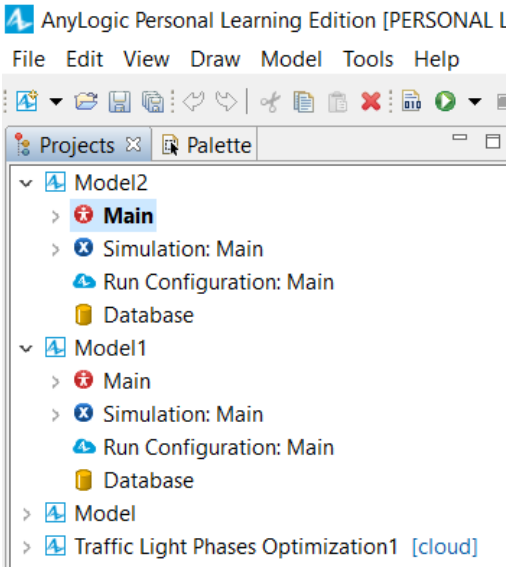


Рис. 4.8. Створення нової моделі

Правіше від ярлика New знаходиться Open Model, що дає змогу відкрити модель. Поряд розташований ярлик Save all models, що дає змогу зберігати розроблені моделі. Панель Projects (проєкти) представляє перелік всіх розроблених проєктів та їх внутрішню структуру (рис. 4.9).

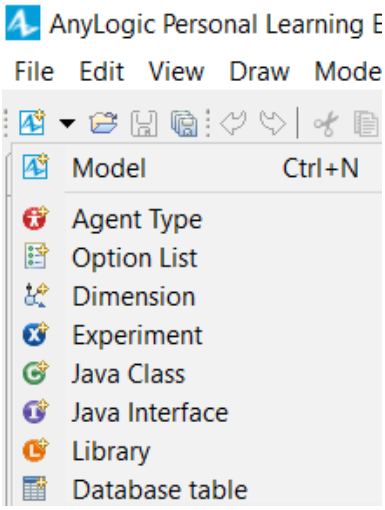


Рис. 4.9. Панель Projects. Представлено перелік назв створених проєктів

- Панель Palette (палітри) складається з декількох вкладок (палітр), кожна з яких містить елементи, що відносяться до певної задачі (рис. 4.10): Process Modelling Library (бібліотека моделювання процесів) – основна вкладка, позначена символом DL – містить головні елементи, за допомогою яких можна задати динаміку моделі, її структуру та дані;

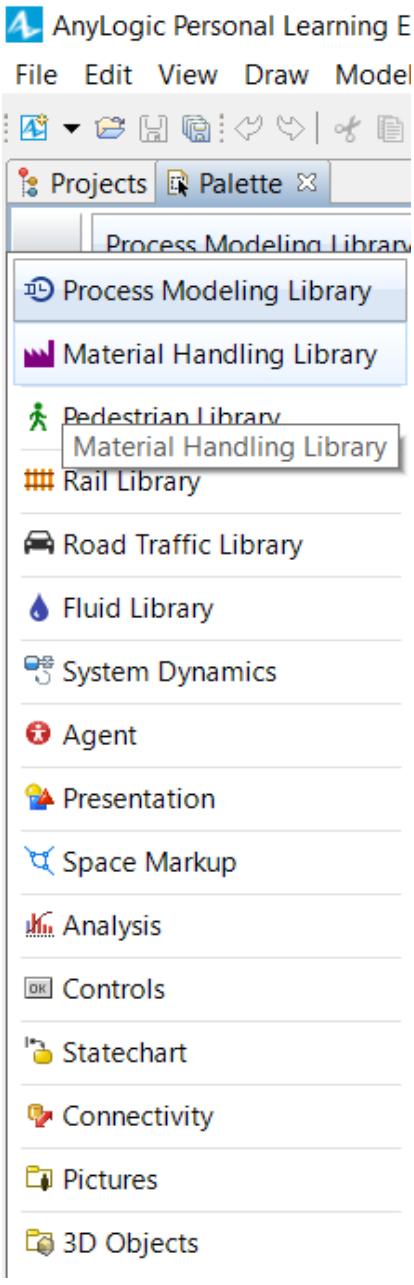


Рис. 4.10. Перелік вкладок панелі Palette

- Material Handling Library (бібліотека обробки матеріалів) спрощує моделювання складних виробничих систем і операцій. Її використовують для розробки детальних моделей виробничих і складських приміщень і керування виробничими операціями, транспортування та інвентаризації, а також для уникнення затримок потоку матеріалів на виробництві;

- Pedestrian Library (пішохідна бібліотека) використовується для моделювання динаміки пішоходів у міських ландшафтах, музеях, торгових центрах і транспортних вузлах. На етапі попереднього проєктування імітаційні моделі пішоходів допоможуть оцінити здатність об'єкта впоратися із запланованим навантаженням і відповідності вимогам безпеки. Можливості пішохідної бібліотеки будуть корисними під час оцінки пропускної здатності та мобільності. Під час реконструкції інструмент управління натовпом підтримуватиме тестування запланованих змін і визначення найкращого рішення.

- Rail Library (залізнична бібліотека) дає змогу користувачам ефективно моделювати, імітувати та візуалізувати роботу залізничних станцій і залізничного транспорту будь-якої складності та масштабу. За допомогою цієї бібліотеки можна моделювати класифікаційні станції, парки великих заводів, залізничні станції, вагоноремонтні підприємства, станції метрополітену, маршрутні потяги до аеропортів і навіть трамвайні мережі. Цей інструмент допомагає користувачам планувати операції, керувати автопарком, а також планувати розклад руху поїздів та їх технічне обслуговування.

- Road Traffic Library (бібліотека дорожнього руху) дає змогу планувати, проєктувати та моделювати транспортні потоки на детальному фізичному рівні. Бібліотека ідеально підходить для явного моделювання поведінки кожного водія та для аналізу і вивчення динаміки транспортних потоків.

- Fluid Library (бібліотека рідин) моделює процеси транспортування та зберігання сипучих матеріалів, рідин і газу, включно з моделюванням операцій функціонування трубопроводів, процесів видобутку, а також виробництва та транспортування води, нафти чи палива. За допомогою компонентів бібліотеки є змога створювати точні копії резервуарів, труб, конвеєрів та їх мереж, а також виконувати пакетне відстеження потоків. Бібліотека реєструє різні характеристики потоків, як-от швидкість і пропускна здатність, та оптимізує операційні процеси.

- System Dynamic (системна динаміка) підтримує проєктування та моделювання структур зворотного зв'язку, як-от фондові та потокові діаграми у спосіб, знайомий більшості системно-динамічних моделей, та пропонує всі переваги об'єктно-орієнтованого підходу до моделювання. Шаблон системної динаміки можна зберегти як об'єкт бібліотеки та повторно використовувати в іншій імітаційній моделі. Вона дає змогу отримувати переваги від таких процесів, як-от експорт моделі, виконання хмарної моделі, складна анімація та взаємодія з іншими програмними інструментами. System Dynamics містить елементи діаграми потоків і накопичувачів, а також з'єднувач та табличну функцію. Системна динаміка є дуже абстрактним методом моделювання. Ігноруються тонкі деталі системи, як-от індивідуальні властивості людей, продуктів або подій,

і створюється загальне представлення складної системи. Ці абстрактні імітаційні моделі можна використовувати для довгострокового стратегічного моделювання та імітації. Наприклад, телефонна мережа, яка обслуговує маркетингову кампанію, може симулювати та аналізувати успішність нових ідей без необхідності моделювати взаємодію окремих клієнтів.

- Agent (агент) може представляти дуже різні об'єкти: транспортні засоби, елементи обладнання, проєкти, продукти, ідеї, організації, інвестиції, ділянки землі, людей у різних ролях тощо. Агенти є основними конструктивними блоками моделі AnyLogic. Агент – це одиниця дизайну моделі, яка може мати поведінку, пам'ять, історію, час, контакти тощо. Всередині агента можна визначати змінні, події, діаграми станів, фондові та потокові діаграми System Dynamics. Можна також вбудовувати інших агентів, додавати блок-схеми процесів та визначати, скільки типів агентів потрібно задіяти у своїй моделі. Проєктування агента зазвичай починається з визначення його атрибутів, поведінки та взаємодії із зовнішнім світом. У випадку великої кількості агентів з динамічними зв'язками (як-от соціальні мережі) агенти можуть спілкуватися за допомогою виклику функцій. Внутрішній стан і поведінка агента можуть бути реалізовані декількома способами. Стан агента може бути представлений кількома змінними, діаграмою станів тощо. Поведінка агентів буває як пасивною (наприклад, є агенти, які реагують лише на надходження повідомлень або виклики функцій) так і активною, коли внутрішня динаміка змушує його діяти.

- Presentation (презентація) AnyLogic пов'язана з компонентами моделі, якими є агенти (Agents), і формується відповідно до ієрархії моделі. Презентація розробляється модульно, окремо для кожного об'єкта. Їх можна включити в будь-яку сцену презентації вищого рівня, пов'язану з об'єктом-контейнером.

- Space Markup (розмітка простору) – блоки, що виконують різноманітні операції над агентами та ресурсами та можуть анімувати їх діяльність. Щоб анімувати агентів, які знаходяться в блоці, використовуються форми розмітки простору з розділу «розмітки простору» палітри бібліотеки моделювання процесів – шляхи та вузли. Шляхи та вузли – це елементи розмітки простору, які визначають розташування агентів у просторі.

- Analysis (аналіз) допомагає дослідити, наскільки результати моделювання чутливі до змін параметрів моделі. Експеримент запускає модель кілька разів, змінюючи один із параметрів, і показує, як результат моделювання залежить від нього. Для одного типу виводу відображається діаграма «вихід проти параметра». Якщо результатом моделювання є набір даних (наприклад, динаміка певного процесу в часі), серія кривих відображається на одній діаграмі для порівняння.

- Controls (контроль) дає змогу зробити моделі AnyLogic інтерактивними, застосувавши різні елементи керування (кнопки, повзунки, введення тексту тощо) у інтерфейс моделі, а також шляхом визначення реакції на клацання мишею. Елементи керування можна використовувати як для попереднього налаштування параметрів до виконання моделі, так і змінювати модель у міру роботи над нею. Ці елементи можна знайти на палітрі елементів керування, створювати їх та редагувати так само, як і фігури. Елементи керування можна згрупувати за допомогою фігур та інших елементів керування. Ці елементи мають динамічні властивості, які можуть бути використані для зміни їх розміру, положення, доступності та видимості під час виконання. Вони завжди з'являються поверх будь-якої іншої графіки (фігур, елементів моделі тощо). Потрібно уникати накладання цих елементів, оскільки це може призвести до небажаних візуальних ефектів.

- Statechart (діаграма станів) складається із блоків діаграм, що дають змогу графічно задавати поведінку об'єкта. Хоча використання подій досить зрозуміле, іноді потрібно визначити більш складну поведінку. У такому випадку використовують діаграму стану. Діаграма станів – це найдосконаліша конструкція для опису поведінки, керованої подіями та часом. Для деяких об'єктів таке впорядкування операцій за подіями та часом є настільки поширеним, що ви можете найкраще охарактеризувати поведінку таких об'єктів за допомогою діаграми станів. Така діаграма володіє станами та переходами. Переходи можуть бути викликані умовами, визначеними користувачем. Виконання переходу часто призводить до зміни стану, коли новий набір переходів стає активним. Стани в діаграмі станів можуть бути ієрархічними, тобто містити інші стани та переходи. Використовуючи діаграми станів, можна візуально охопити широкий спектр дискретних дій, набагато більш насичених, ніж просто стан «неактивний / зайнятий» або «відкрито / закрито».

- Connectivity (підключення) – кожна модель AnyLogic має вбудовану повністю інтегровану базу даних для читання вхідних даних і запису вихідних даних моделювання. База даних із моделлю така ж портативна та кросплатформна, як і сама модель. З новою базою даних можна: 1) читати значення параметрів і здійснювати налаштування моделей; 2) створювати параметризовані групи агентів; 3) генерувати надходження сутностей у моделях процесів; 4) імпортувати дані з інших баз або електронних таблиць Excel; 5) переглядати використання ресурсів; 6) зберігати та експортувати статистику, набори даних і спеціальні журнали; 7) взаємодіяти із зовнішніми джерелами даних.

- Pictures (зображення) – ця палітра містить часто використовувані зображення в масштабованому векторному графічному форматі AnyLogic. Тепер не потрібно малювати людину, машину чи будинок з нуля кожного разу, коли вам

потрібно додати ці елементи до моделі – просто треба перетягнути їх із палітри «Зображення» та інсталювати у полотно. Зображення є групами стандартних форм AnyLogic, і їх можна масштабувати, змінювати кольори, змінювати внутрішні елементи, розгруповувати та навіть керувати ними програмно. Палітра включає людей, транспортні засоби, літаки, карти, будинки та промислові будівлі.

- 3D Objects дає змогу користувачам AnyLogic імпортувати готові до використання 3D-моделі, створені за допомогою будь-яких пакетів 3D-графіки сторонніх виробників, у свої моделі.

Практику моделювання з допомогою AnyLogic North America LLC краще отримувати на конкретних прикладах. Змоделюємо міське автомобільне перехрестя. Для цього вмикаємо Road Traffic Library (бібліотеку дорожнього руху) на панелі Palette, яка дає змогу планувати, проєктувати та моделювати транспортні потоки на фізичному рівні. Бібліотека підходить для моделювання динаміки транспортних потоків. Тут є спектр конструкторів, представлених на рис. 4.11.

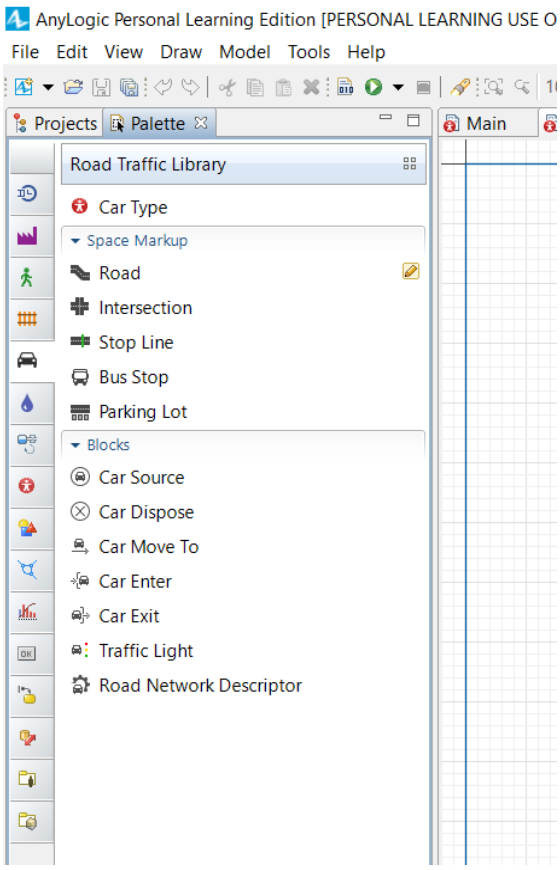


Рис. 4.11. Конструктори дорожньої бібліотеки

Щоб почати моделювати дорогу, потрібно двічі клацнути лівою кнопкою миші на значку Road, потім навести курсор на те місце робочої області (графічного редактора), де ми бажаємо прокласти дорогу. Клацаємо у позиції, де

має починатись дорога, а потім двічі клацаємо у позиції, де вона має закінчуватись. З'являється таке зображення (рис. 4.12). Вверху є віконце з масштабом: реалізована можливість змінити розміри, як вам потрібно.

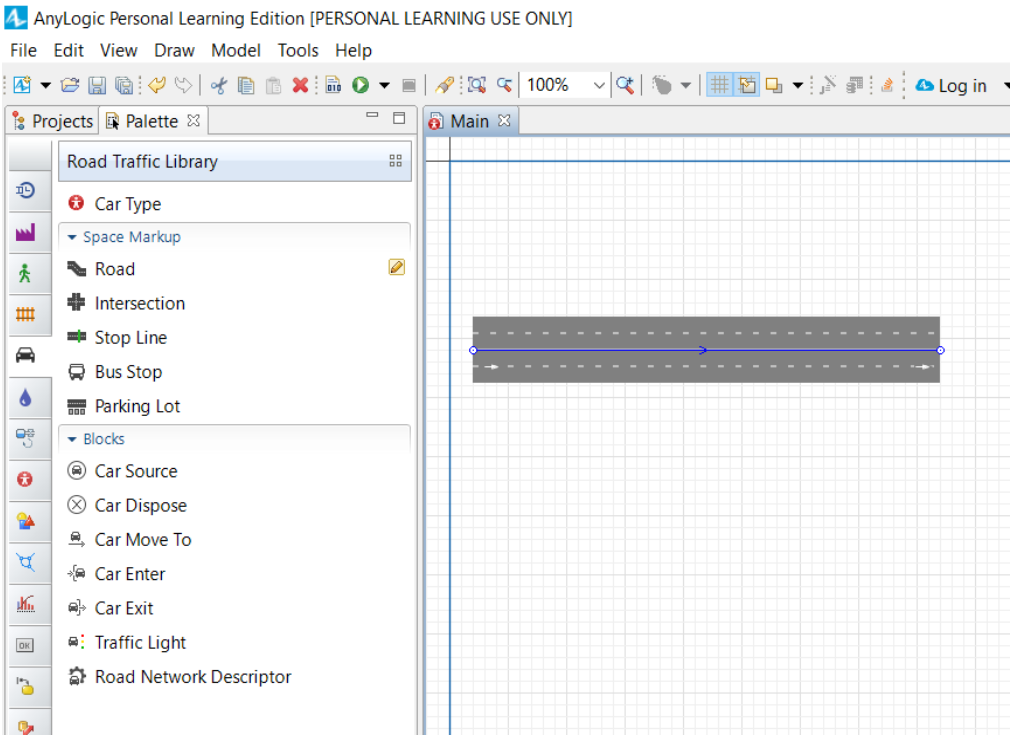


Рис. 4.12. Модель прямолінійної ділянки дороги

Існує другий варіант створення ділянки дороги – для цього просто потрібно зображення значка дороги перетягнути у робочу область.

Дорогу у будь-який момент моделювання можна редагувати – змінювати її довжину, ширину, напрямок, переносити паралельно сама собі, видовжувати чи скорочувати.

Коли дорога виділяється, то активізується панель Properties – з'являються такі параметри дороги: Number of forward lanes (кількість смуг основного руху), Number of backward lanes (кількість смуг зворотного руху),

Median strip width (ширина розділювальної смуги) та Median strip color (колір розділювальної смуги). Можна поставити галочку у віконці One way (односторонній рух). Існує можливість також поміняти розташування дороги по осях *x*, *y* та *z*. Для цього потрібно розкрити закладку Position and size, де виставляються координати початку дороги. До речі, початок координат розташований зверху робочої зони, вісь *x* спрямована вправо, а вісь *y* – вниз. Двічі клацнувши по осьовій лінії дороги, можна додавати точки зламу – маленькі квадратики. Потягнувши за квадратик, можна створити зигзагоподібну дорогу. Самі квадратики можна переміщати вздовж осьової лінії дороги. Якщо змінювати положення дороги, то спостерігаються відповідні зміни і в таблиці, роз-

ташованій у вікні Position and size. Також положення дороги можна змінювати, вводячи параметри безпосередньо у таблицю Position and size. Так само дорогу можна перейменовувати, наприклад, road → highway. Існує спосіб збільшувати кількість смуг руху, тобто робити дорогу повноцінною.

Перейдемо тепер до побудови перехрестя, яке можна побудувати двома способами. Можна витягнути із панелі готове перехрестя і продовжувати моделювати дороги, які формуються із центру. Для цього потрібно Road ввести в режим малювання. Але є інший спосіб моделювання перехрестя. Спочатку малюємо дорогу і створюємо звичайне Т-подібне перехрестя. Додаємо ще одне плече дороги, внаслідок чого отримуємо зображення, представлене на рис. 4.13. Створеною моделлю можна керувати: якщо затиснути Ctrl і крутити колесо миші, то спостерігається регулювання масштабу. Якщо затиснути праву кнопку миші, то можна рухати модель. У центрі перехрестя виділені смуги руху та контурні лінії, що з'єднують смуги руху на різних плечах цього хрестоподібного перехрестя.

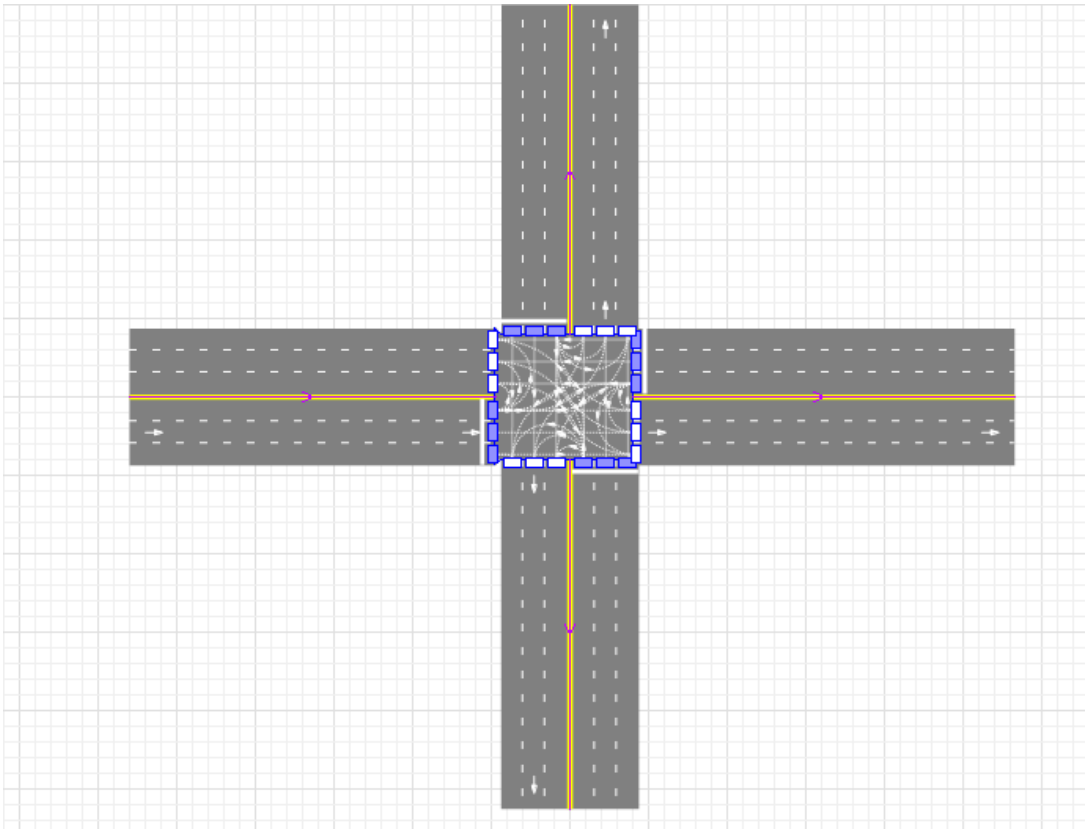


Рис. 4.13. Модель хрестоподібного перехрестя

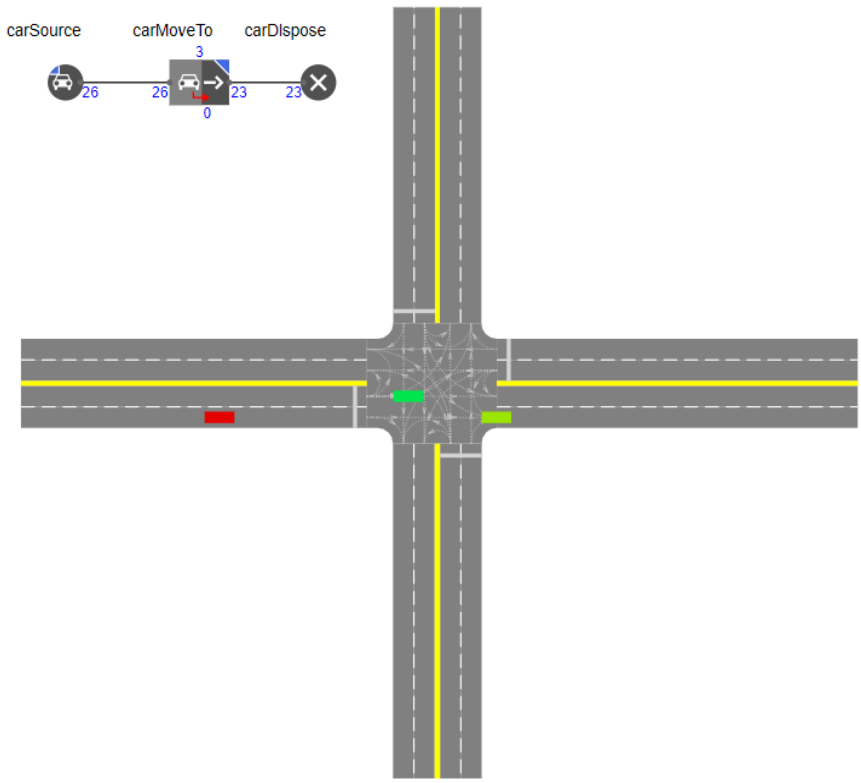
Центральний квадрат перехрестя заповнений контурними лініями руху автомобілів: потрібно налаштувати ці лінії відповідно до правил дорожнього руху (ПДР). Натиснемо Ctrl, виділимо перехрестя та натиснемо на один із прямокутників, що символізує собою смугу руху. Виділяться суцільні білі лінії,

які є активними, та пунктирні – неактивні. Активуємо центральну частину перехрестя і зробимо активними лінії на кожній смузі руху, виділяючи відповідні прямокутниками. Для видалення активних ліній руху, що не відповідають ПДР, виділяємо лінію, клацаємо по цій лінії, і після цього зникає стрілочка, що вказує напрямок. Це значить, що ця лінія стала неактивною. Якщо ж потрібно, навпаки, активувати лінію, що світиться пунктиром, то клацаємо по цій лінії, і вона стає активною.

У позиції Blocks наявна опція Car Source, що відповідальна за процедуру створення автомобілів: клацаємо по цій опції, затискаємо ліву клавішу миші та витягуємо зображення у робочу зону. Далі аналогічно активуємо у робочу область CarMoveTo та CarDispose. Ці три опції мають бути з'єднані лініями: якщо підводити зображення одне до одного, то такі з'єднувачі формуються автоматично. Для активації Car Source натискаємо на його іконку у робочій зоні: під час цьогомо активується вкладка Properties справа. Навпроти запису Road знаходимо віконце, натискаємо галочку: з'являється випадний список – road, road1, road2, road3. Якщо активувати кожне плече дороги, то відповідно у полі Properties з'являються назви активованого плеча дороги: лівому плечу відповідає road (з'являється відповідний запис у полі Name), нижньому – road1, правому – road2 і верхньому – road3.

Активувати рух автомобілів можна з будь-якої сторони. Виберемо ліве плече, ввівши у віконце Road запис road. У позиції Appears ввімкнемо опцію on road, а не parking lot. У позиції Enters включаємо forward lane. Це означає, що автомобілі будуть з'являтись на головній смузі дороги road (ця смуга позначена двома білими стрілочками). Навпаки, смуга backward lane (зворотна смуга) не позначена ніяк. Тепер ввімкнемо опцію CarMoveTo, що відповідає за напрямок руху автомобілів, тобто на яке плече має рухатись автомобіль. Нехай автомобілі на першому етапі моделювання рухаються зліва направо, тобто у напрямку road → road2. Напрямок руху автомобіля співпадає з головною дорогою на road2, тому вибираємо forward lane. Опція CarDispose потрібна для того, щоб автомобілі видалялись із робочої зони.

Тепер можна запустити нашу просту модель, натиснувши на зелений трикутник на панелі інструментів. Запускається програма і здійснюється динамічна процедура руху автомобілів зліва направо (рис. 4.14). Вгорі зліва представлена логіка руху: показано кількість автомобілів, що генеровані carSource. На іконці carMoveTo кількість 23 означає кількість транспортних засобів, що вийшли з робочої зони та потрапили у CarDispose. Цифра 3 означає число автомобілів, що знаходяться у робочій зоні. Створимо тепер зустрічний рух: автомобілі рухатимуться із road2 у напрямку road (рис. 4.15). Для цього створюємо послідовність carSourse1 → carMoveTo1 → carDispose1.



4.14. Логіка динамічної моделі під час руху автомобілів зліва направо

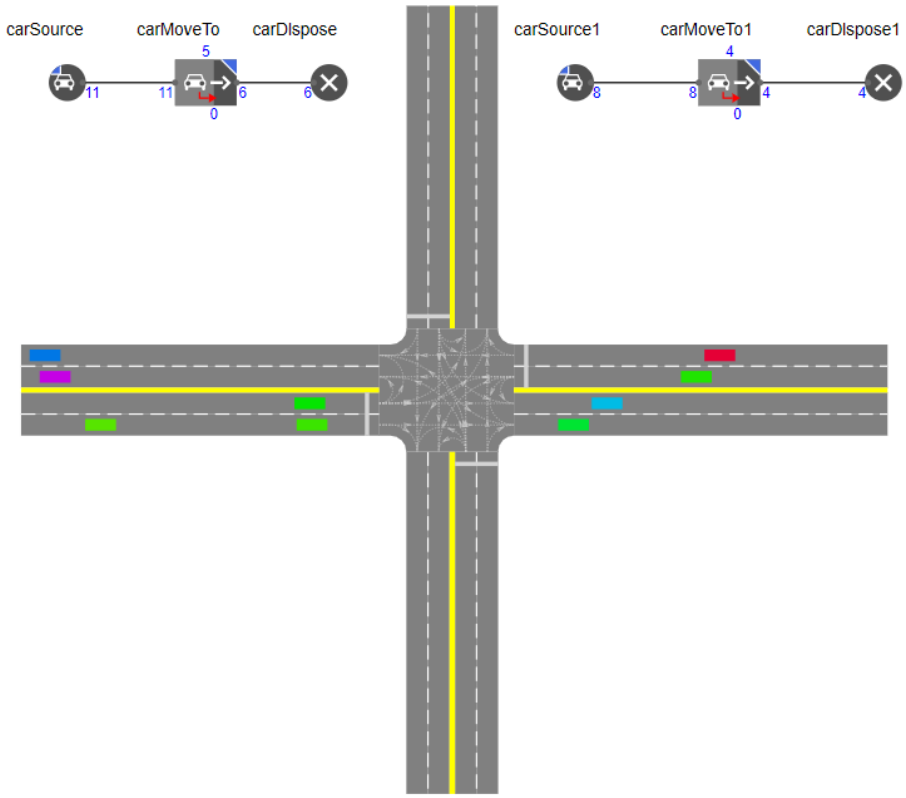


Рис. 4.15. Модель руху автомобілів у горизонтальному напрямку

Автомобілі, що під'їжджають до перехрестя, можуть рухатись у чотирьох можливих напрямках – прямо, направо, наліво і на розворот. Нехай потрібно змоделювати ситуацію повороту із road направо на road1.

Це означає, що потік автомобілів із road потрібно розділити на дві частини – одна частина автомобілів прямуватиме прямо на road2, а інша повертатиме вниз на road1.

З метою поділу потоку автомобілів потрібно встановити функцію Select Output, що знаходиться у Process Modelling Library (закладка DL). Продовжуючи моделювання перехрестя, моделюємо всі можливі режими руху автомобілів (рис. 4.16): з кожного із чотирьох напрямків транспортний засіб має змогу рухатись у чотирьох напрямках – прямо, наліво, направо та розворот у зворотному напрямку.

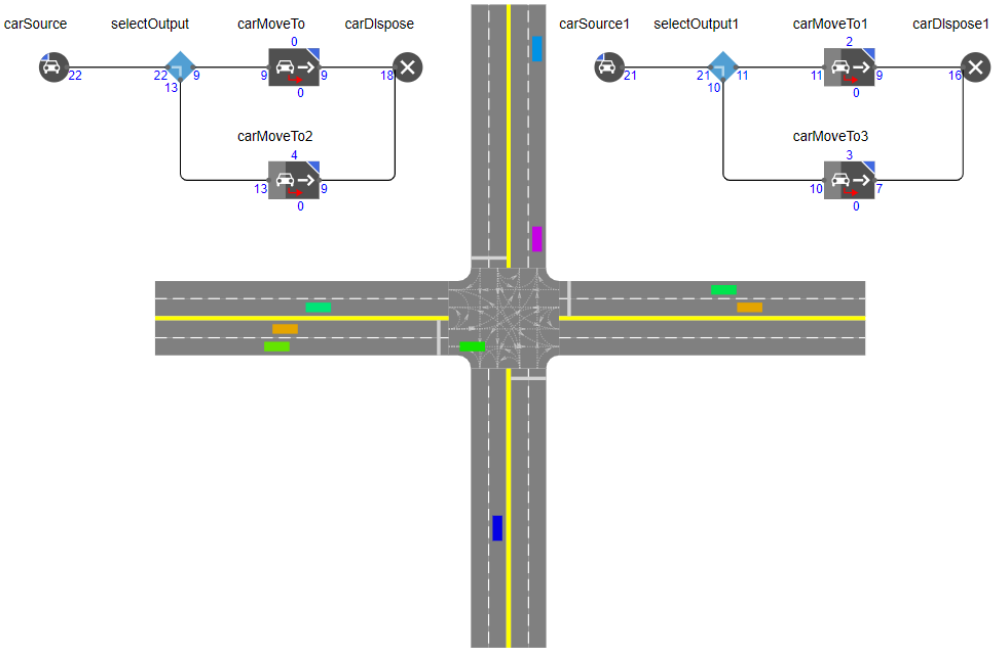


Рис. 4.16. Модель руху автомобілів у двох напрямках

Отже, остаточно отримуємо варіант моделі, представлений на рис. 4.17. Фінальна модель інтелектуального регульованого перехрестя створена так, щоб синхронізувати протяжність горіння зелених фаз світлофору на конкретному напрямку, узгодивши його з кількістю автомобілів на цьому напрямку. Це фактично означає оптимізацію режиму світлофорного регулювання. Якщо всі регульовані у місті світлофори перейдуть у такий режим, то міський трафік підніметься на якісно новий рівень.

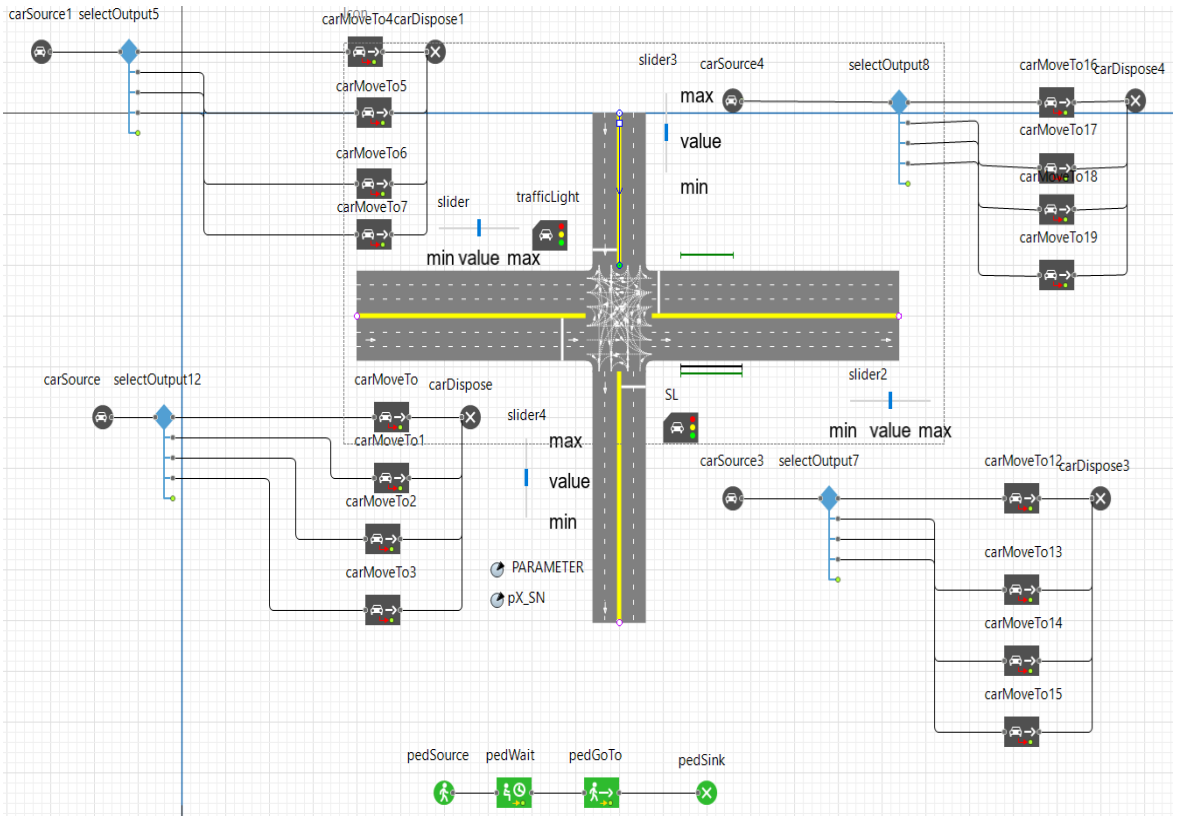


Рис. 4.17. Фінальна модель інтелектуального перехрестя. Зображене перехрестя та діаграми логіки. Показані також світлофори, що регулюють потоки транспортних засобів

4.4. Моделювання фізичних систем

Фізична модель – представлення об’єкта, системи, явища або процесу за допомогою іншого діючого фізичного об’єкта, який відтворює у тому чи іншому аспекті динаміку і характер поведінки досліджуваного феномену. Можна також сказати, що фізична модель реалізується шляхом створення експериментальної установки, що дає змогу проводити фізичне моделювання шляхом заміщення реального фізичного процесу подібним до нього процесом тієї ж фізичної природи.

Фізичне дослідницьке обладнання, на якому проводяться експерименти, фактично дає змогу моделювати реальне фізичне явище чи процес за умови, що є фізична подібність до реальної досліджуваної системи. Фізична подібність фактично означає повну відповідність між параметрами об’єкта і моделі. Під моделлю тут слід розуміти фізичне устаткування, створене для вивчення системи, процесу, явища або об’єкта.

Прикладом фізичної моделі може бути масштабна модель. Наприклад, потрібно дослідити механічну міцність батискафа – апарата для вивчення океанських глибин. Для цього конструюють його зменшену копію. Потім таку

фізичну модель занурюють у спеціально сконструйований басейн з водою. Далі створюють тиск, величина якого досягає 1 500 кГ/см^2 або більше. Реально вказана величина тиску на морських глибинах не досягається. Проте для глибоководних апаратів потрібен запас міцності. Практика показує, що недостатній запас міцності може приводити до трагедій, як це трапилось з глибоководним апаратом «Титан» 18 червня 2023 року під час занурення до уламків лайнера «Титанік».

Загалом фізичне моделювання спрямоване на експериментальне вивчення фізичних явищ та процесів, які у природніх умовах дослідити дуже проблематично або взагалі неможливо. Для прикладу розглянемо кульову блискавку. Для її вивчення потрібно створити лабораторну установку, яка відтворює необхідні умови для «штучного» виникнення вказаного виду утримання електромагнітної енергії. Далі дослідження кульової блискавки проводиться шляхом багаторазового повторення експерименту у лабораторних умовах.

Одним із прикладів застосування фізичного моделювання є дослідження процесів обтікання автомобілів, літальних апаратів чи снарядів газовими потоками у так званих аеродинамічних трубах. Такі дослідження проводяться з метою підбору особливої форми досліджуваного об’єкта за якої сила опору (так звана сила Стокса) стає мінімальною. На рис. 4.18 зображено макет літального апарата, розташований в аеродинамічній трубі.

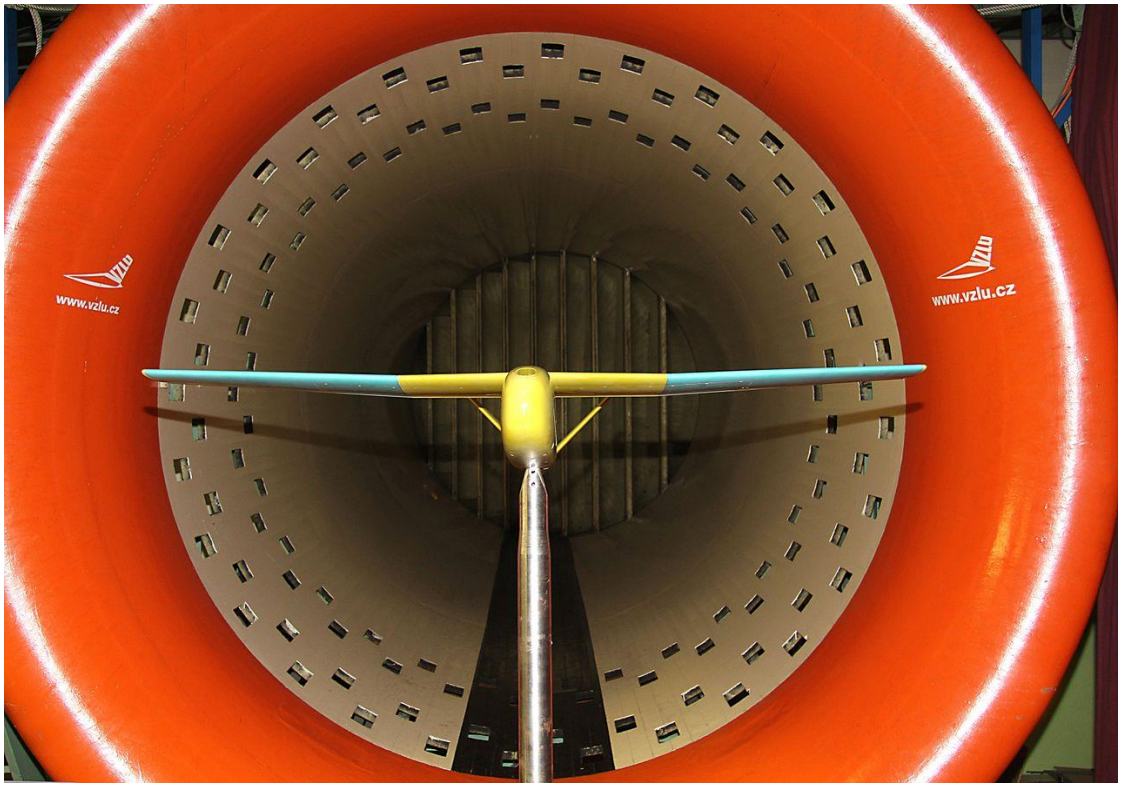


Рис. 4.18. Макет літального апарата, розташований в аеродинамічній трубі. Дослідження проводяться з метою створення такої форми літального апарата, за якої величина сили Стокса є мінімальною

Наведемо приклад фізичної моделі, розглядаючи так звану ланцюгову реакцію. Мова буде йти про розпад радіоактивних ядер урану. Одним із прикладів ділення ядер урану є така реакція:

$$U_{92}^{235} + n_0^1 \to Cs_{55}^{140} + Rb_{37}^{94} + 2n_0^1 + Q. \qquad (4.8)$$

Записану реакцію слід розуміти так: нейтрони n_0^1, опромінюючи ядра урану, спричиняють їх поділ, що супроводжується утворенням двох осколків – ядер цезію і рубідію, а також виділенням енергії Q. Виявляється, що під час поділу ядер урану відбувається виділення вторинних нейтронів, як видно зі схеми реакції (4.8).

Випромінювання під час поділу ядер урану вторинних нейтронів має принципове значення, оскільки робить можливим існування ланцюгової реакції. Дійсно, випускання під час реакції поділу одного ядра атома урану (4.8) z нейтронів може викликати поділ z інших ядер урану. Внаслідок цього буде випущено z^2 нейтронів нового покоління. У кожному новому поколінні кількість нейтронів буде збільшуватися в геометричній прогресії. Тут доцільно ввести поняття коефіцієнта розмноження нейтронів k, що являє собою відношення кількості нейтронів наступного покоління до кількості нейтронів попереднього покоління у всьому об'ємі активного нейтронного середовища, наприклад, в активній зоні ядерного реактора. За умови $k > 1$ самоплинно буде продовжуватись реакція поділу, як це схематично показано на рис. 4.19. Загалом ланцюгова ядерна реакція може бути трьох видів: 1) затухаючою, коли коефіцієнт розмноження нейтронів $k < 1$; 2) незатухаючою контрольованою, коли $k \approx 1$; 3) незатухаючою неконтрольованою, коли $k > 1$.

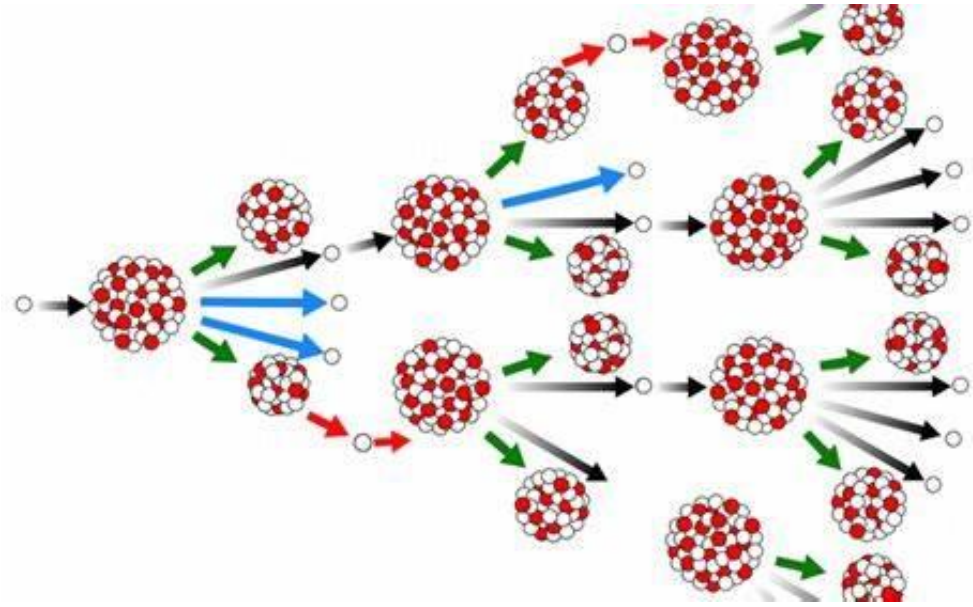


Рис. 4.19. Схема ланцюгової реакції поділу ядер радіоактивних елементів

У першому випадку реакція швидко затухає. Це відбувається, коли маса збагаченого урану менше критичної або концентрація ізотопу урану U_{92}^{235} незначна. У другому випадку ми маємо справу із контрольованою атомною реакцією, що реалізується у атомних реакторах. Третій випадок відповідає ситуації, коли реакція відбувається швидкоплинно та лавиноподібно та, як результат, виділяється колосальна енергія. Така ситуація реалізується у атомній бомбі.

Певним аналогом описаного тріадного атомного процесу є епідемії та пандемії. Останніми роками добре відома пандемія – COVID-19. Якщо у момент виникнення інфекції у певний час у певному місці створити умови, за яких коефіцієнт поширення інфекції R_0 (своєрідний аналог коефіцієнта розмноження нейтронів k у атомній реакції) стане меншим одиниці ($R_0^L < 1$), то епідемію можна швидко подавити. За умови $R_0^M \approx 1$ епідемія розповсюджується, але повільно, не затихає і навпаки, не прогресує. У цьому випадку епідемію можна подавити шляхом проведення медичних профілактичних заходів. Така ситуація характерна для гострої вірусної інфекції на зразок грипу. Особливо небезпечною є ситуація, коли $R_0^H > 1$. У цьому випадку інфекція швидко розповсюджується, епідемія трансформується у пандемію, що еквівалентно переходу $R_0^M \to R_0^H$.

Між розглянутою моделлю атомної реакції та процесом розповсюдження інфекційного захворювання будь-якого типу існує виражена подібність. На рис. 4.20 представлена модель SIR, запропонована Кермаком-МакКендріком. Згідно з цією моделлю населення будь-якого регіону планети поділяється на три групи: перша група – особи, сприйнятливі до захворювання S (susceptible), інфіковані патогеном особи I (infected) та люди, що одужали R (recovered). Величини S, I та R змінюються з часом, тобто є функціями від *t*. Наочно демонстрація залежностей *S(t), I(t) та R(t)* та їх динаміка представлені за посиланням https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0% BB:SIR_model_anim.gif

Аналіз динаміки вищезгаданих функцій дає змогу спрогнозувати можливі спалахи поширення епідемій та пандемій, а значить – контролювати їх і швидко подавити.

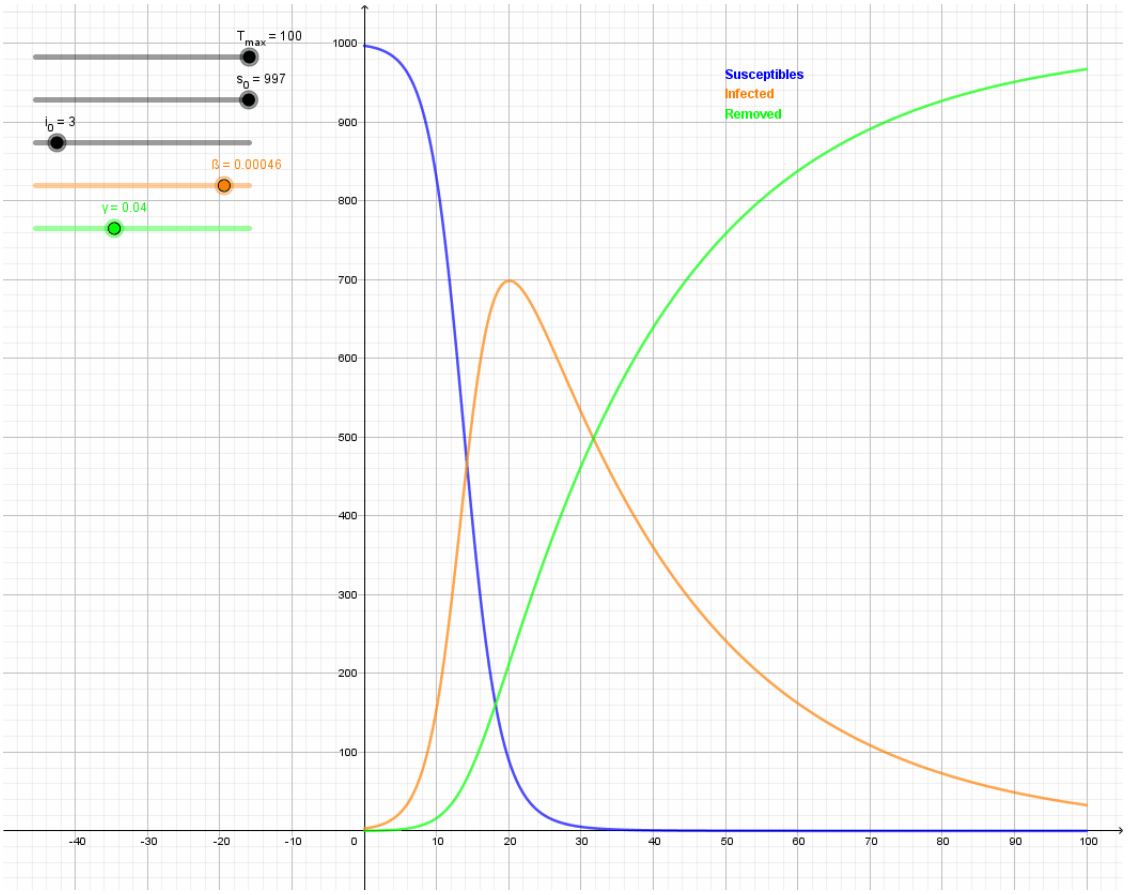


Рис. 4.20. Модель SIR розповсюдження інфекції

Особливо важливе значення має вигляд функції I(*t*). Як видно із рис. 4.20, зростаюча ділянка функції I(*t*) характеризується різким зростанням, що свідчить про швидке збільшення кількості осіб, які захворіли та є носіями інфекції. Ці ж особи інфікують навколишніх людей, що, власне, і спричиняє активацію процесу лавиноподібного розповсюдження інфекції. Аналогічна ситуація спостерігається під час ланцюгової реакції поділу ядер радіоактивних елементів (рис. 4.19). Лавиноподібний процес у першому, і у другому випадках можна зупинити шляхом зменшення так званої критичної маси (якщо йдеться про ланцюгову реакцію) або завдяки превентивним заходам, спрямованим не недопускання скупчення людей (якщо йдеться про епідемію). Важливо зауважити, що вирішальний вплив на поведінку функції I(*t*) відіграє фактор часу *t* (назвемо його *t*-фактор). Фактично це означає *своєчасне* прийняття комплексних заходів захисту населення від інфекції. Тут дорога кожна хвилина. Наочним прикладом є лісова (і не тільки) пожежа: загасити її можна швидко лише у перші хвилини виникнення. Далі процес стає неконтрольованим, і потрібно витратити колосальні ресурси для приборкання такої природної стихії. Так само і у випадку епідемії: чим дієвішими та активнішими будуть дії уряду тієї чи іншої країни у момент виникнення епідемії, тим швидше буде приборкане таке стихійне

лихо. Якщо тепер розглянути ситуацію з аварією світового масштабу на Чорнобильській АЕС, то варто зазначити, що горизонтальне розташування уранових стержнів не дало змоги миттєво ввести графітові стержні, що гасять атомну реакцію. У випадку вертикального розташування уранових стержнів у атомних реакторах введення графітових стержнів відбувається за частки секунди: вони просто падають під дією сили тяжіння. Саме так сконструйовані сучасні атомні реактори. Сформулюємо підсумок.

Означення 4.1. У швидкоплинних процесах типу атомних реакцій, епідемій, пандемій та пожеж вирішальне значення відіграє t-фактор.

РОЗДІЛ 5
МЕРЕЖІ МАСОВОГО ОБСЛУГОВУВАННЯ

5.1. Мережі Петрі

Теорія мереж Петрі – формалізм, призначений для імітації роботи динамічних системам. Зараз розроблена велика кількість різноманітних моделей, методів та засобів аналізу, а також різноманітних додатків практично у всіх галузях. Мережі Петрі є типовим прикладом.

Мережа Петрі – математичний апарат, що використовується для моделювання динамічних дискретних систем. Згаданий апарат був розроблений Карлом Петрі у 1962 році. Мережу Петрі по-іншому можна визначити як дводольний орієнтований мультиграф, у якому допускається існування кратних дуг, що пролягають від однієї вершини графа до іншої. Такий граф із кратними дугами є антиподом простого графа, який ще можна назвати монографом. Мультиграф складається з вершин двох типів – позицій та переходів. Позиціям відповідають вершини, що представляються у вигляді кіл, а переходам відповідають вершини, що являють собою потовщені риски-смужки (або просто прямокутники). Позиції і переходи з'єднані між собою дугами (направленими ребрами) так, що кожна дуга спрямована від елемента однієї множини (позиції або переходу) до елемента іншої множини – переходу або позиції. Наочно ситуацію з позиціями, переходами і дугами, що їх з'єднують можна представити у такому вигляді (рис. 5.1).

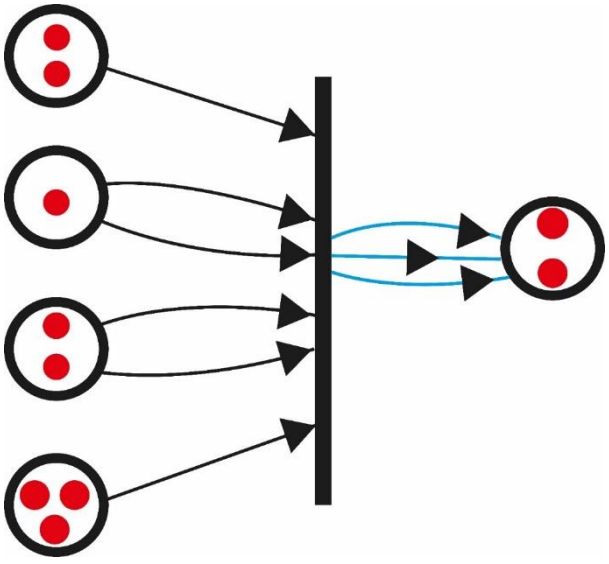


Рис. 5.1. Чорними кільцями зображені позиції P (зліва – вхідні, а справа – вихідні). Переходи T представлені прямокутною чорною смужкою. Вхідні функції I зображуються дугами чорного кольору та розміщені зліва; вихідні функції O розміщені праворуч та зображені дугами блакитного кольору. Червоні кружечки всередині позицій – це так звані мітки

Зазвичай, мережа Петрі містить чотири складники – сукупність позицій P, множина переходів T, вхідна функція I та вихідна функція O. Отже, мережу Петрі можна визначити як квартет вигляду $\langle P,T,I,O\rangle$, де P і T – позиції і переходи відповідно, водночас I та O символізують собою вхідні та вихідні функції відповідно. Позиціям відповідають вершини, що зображаються кружечками, а переходам відповідає потовщена смужка. Функціям I відповідають дуги, які спрямовані від позицій до переходів. Дуги типу O символізують собою вихідні функції. Мітки зображені як червоні кружечки всередині позицій.

Мітки ще називають маркерами, а розподіл маркерів за позиціями – маркуванням. Маркери мають можливість переміщатись у такій специфічній мережі. Переміщення маркера в мережі називають подією. A priori вважається, що події відбуваються миттєво і у різні моменти часу. Вершини (позиції і переходи) одного типу не можуть бути з'єднані безпосередньо (між собою). Коли відбувається подія, то кажуть, що відбулося спрацювання переходу; мітки з вхідних позицій переміщуються у вихідні позиції (зліва направо, якщо розглядати рис. 5.1). Зауважимо, що події відбуваються миттєво, у різні моменти часу і за виконання певних умов.

Мережа Петрі функціонує шляхом здійснення запусків переходів. Запуск переходу може бути здійснений тільки у випадку, коли у всіх вхідних позиціях є хоча б одна фішка. Водночас маркери з вхідних позицій переміщуються у вихідні (маркери зліва із вхідних позицій переміщуються у вихідну позицію справа – рис. 5.1). Послідовність таких переміщень маркерів являє собою моделюваний процес, конкретний приклад якого буде наведено далі.

Існують суворі правила спрацювання переходів. Перехід спрацьовує, якщо для кожної із його вхідних позицій виконується умова $N_i \geq K_i$. Тут N_i – кількість маркерів, що знаходяться у вхідній позиції i; K_i – кількість дуг, що йдуть від вхідної позиції i до переходу. Якщо перехід спрацьовує, то кількість маркерів у вхідній позиції стає меншою на величину K_i. Навпаки, у вихідній позиції кількість маркерів збільшується на величину M_j – це кількість дуг, що пов'язують перехід з вихідною позицією j. Чи відбудеться спрацювання переходу у випадку ситуації, представленої на рис. 5.1? Відповідь – ні. Річ у тім, що для другої зверху вхідної позиції умова $N_i \geq K_i$ не виконується.

Розглянемо тепер ситуацію, представлену на рис. 5.2. Для такої мережі Петрі спрацювання переходу відбудеться, оскільки умова $N_i \geq K_i$ у цьому випадку виконується. Маркування (розподіл маркерів за позиціями – рис. 5.2) представляють у вигляді послідовності (1,3,3,3,2). Після реалізації переходу маркування (рис. 5.2) матиме такий вигляд: (0,1,1,2,5). У конкретний момент часу може спрацювати лише один перехід, що вирішує проблему конфліктів.

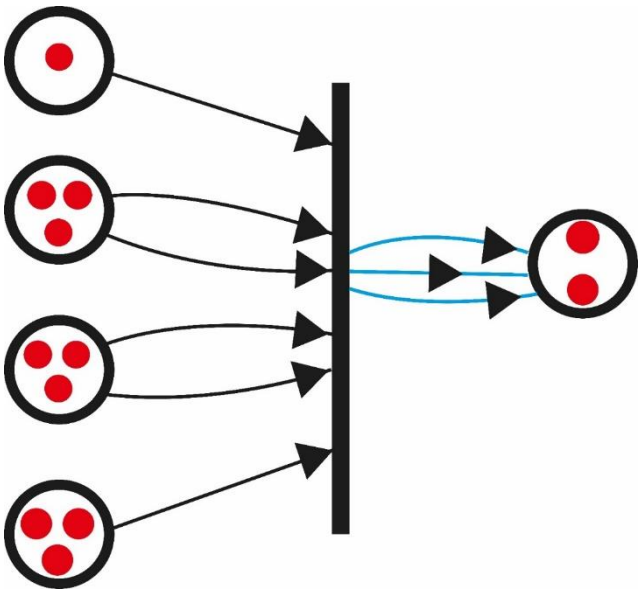


Рис. 5.2. Мережа Петрі, для якої відбувається спрацювання переходу

Існують різні види мереж Петрі. Одним із таких видів є часові мережі Петрі, у яких вводиться так званий модельний час, який використовується для встановлення часу затримки переходу. Якщо ж такі затримки набувають випадкових значень, то мережу розглядають як стохастичну (ймовірнісну). За такого підходу з'являється можливість моделювати не тільки послідовність подій, але і розглядати їх динаміку у часі. Подібні мережі розглядаються як сукупність взаємодіючих процесів, за яких ймовірності переходів з одного стану в інший залежать від поточного стану всієї системи. Інакше кажучи, у стохастичних мережах Петрі взаємодіючі процеси мають ймовірнісний характер, для характеристики яких вводиться клас стохастичних показників, зокрема функція щільності ймовірності. До того ж у стохастичних мережах вводиться поняття ймовірності спрацювання збуджених переходів (на рис. 5.2 зображений саме збуджений перехід). Тобто переходи здобувають вагу, у якості якої виступає тривалість затримки спрацювання переходу. Вага переходу може залежати від спектру факторів, зокрема від кількості маркерів у вихідних позиціях чи їх розподілу по цих позиціях. Подібну мережу називають функціональною. Загалом розрізняють такі види мереж Петрі:

- тимчасова мережа Петрі – переходи мають вагу, що визначає тривалість спрацьовування (затримку);
- стохастична мережа Петрі – затримки є випадковими величинами;
- функціональна мережа Петрі – затримки визначаються як функції деяких аргументів, наприклад, кількості міток у будь-яких позиціях або стану деяких переходів;
- WF-мережі;
- алгебраїчні мережі Петрі;

- кольорова мережа Петрі – мітки можуть бути різних типів, що позначаються кольорами; тип мітки може бути використаний як аргумент у функціональних мережах;

- інгібіторна мережа Петрі – можливі інгібіторні дуги, які забороняють спрацьовування переходу, якщо у вхідній позиції, пов'язаній із переходом інгібіторною дугою, знаходиться мітка;

- ієрархічна мережа – містить не миттєві переходи, в які вкладені інші, можливо, також ієрархічні мережі; спрацювання такого переходу характеризує виконання повного життєвого циклу вкладеної мережі.

Для прикладу розглянемо функціонування інгібіторної мережі Петрі. У такої мережі наявними є так званні заборонні (інгібіторні) дуги. Присутність маркера у вхідній позиції, пов'язаній з переходом інгібіторною дугою, означає заборону спрацювання переходу.

Опишемо принцип роботи мережі Петрі, представленої на рис. 5.3. Нехай існує необхідність моделювання роботи заводу, що складається з трьох цехів: два цехи заводу є виробничі, а третій цех – складальний. Виробничі цехи поставляють певні комплектуючі, що по конвеєру надходять у складальний цех, де збираються і, як результат, отримується кінцевий продукт (машина, агрегат, пристрій, апарат тощо). Нехай для збирання кінцевого виробу потрібно одна комплектуюча X_1 та дві комплектуючі типу X_2. На рис. 5.3 виробничим цехам відповідають переходи t_1 та t_2, а складальному цеху – перехід t_3. Очевидно, що спрацювання переходу t_3 можливе тільки в тому випадку, якщо в позиції p_1 присутня мітка (яка символізує собою комплектуючу типу X_1), а в позиції p_2 наявними є не менше двох міток (які символізують собою комплектуючі типу X_2). Але це ще не все: у позиції p_4 також має бути мітка, яка символізує собою той факт, що складальний цех завершив збирання виробу, і тому конвеєр вільний та готовий поставити комплектуючі X_1 та X_2 у складальний цех. Як тільки складальний цех закінчить зборку виробу, у позицію p_4 надійде мітка. Якщо ж мітки у p_4 не буде, запити, що надходитимуть у вхідні позиції p_1 та p_2, зобов'язані чекати спрацювання переходу t_4. Отже, переходи t_1, t_2 та t_3 функціонують у режимі затримок, які у перших двох переходах рівні проміжкам часу між появами готових комплектуючих, а затримка t_3 рівна часу збирання виробу.

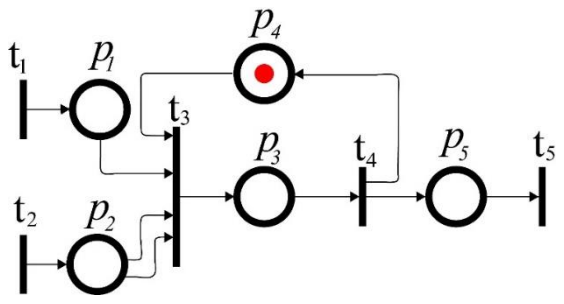


Рис. 5.3. Інгібіторна мережа Петрі

Запустимо модель, використовуючи Petry.NET simulator 2.0, що реалізує описану вище модель. Представлений на рис. 5.4 скрин екрана симулятора відповідає моменту завершення роботи моделі.

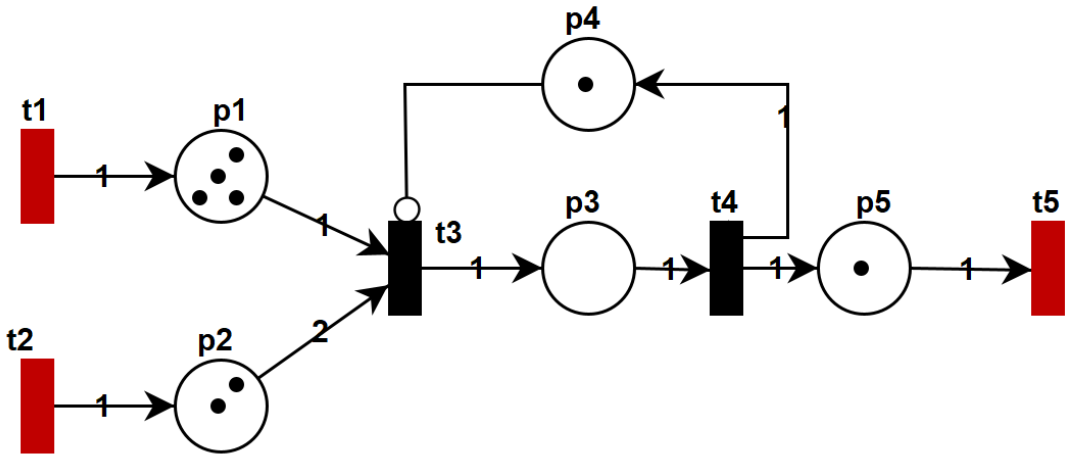


Рис. 5.4. Робота Petry.NET simulator 2.0.
Кількість вхідних та вихідних функцій зображується цифрою, виставленою на дугах

Як і стандартні UML-діаграми, мережі Петрі дають можливість графічно ілюструвати процеси, що передбачають функціонування складної мережі, у якій можливі виробничі затримки. На відміну від UML-діаграм, мережі Петрі моделюють динамічні процеси і базуються на розвиненій математичній теорії.

Сьогодні для моделювання мереж Петрі доступні програмні середовища моделювання, як-т PIPE2 (Platform Independent Petri net Editor) – платформонезалежний редактор мереж Петрі. Ця розробка ведеться за принципом відкритого програмного забезпечення, що дає змогує будь-кому зробити свій внесок у реалізацію. PIPE2 дає широкі можливості редагувати та моделювати ймовірнісні мережі Петрі з дугами, що забороняють переходи. Існує також можливість переглядати матриці інцидентності для поточного маркування, а в режимі моделювання реалізується перелік переходів, що спрацювали, та здійснюється відображення щодо змін кількості міток у позиціях.

Завдяки використанню віртуальної машини для мови програмування JAVA програму можна запускати на різних операційних системах, що дає змогу використовувати PIPE2 на звичній для дослідника платформі. Розглянемо, наприклад, Petri.Net Simulator 2.0 (рис. 5.5) – додаток, створений для моделювання процесів різної природи за допомогою сервісу мереж Петрі. Цей додаток може бути використаний також і для інших дискретних систем. Особливості PIPE2 полягають у тому, що інтерфейс програми зрозумілий та дає змогу швидко створювати імітаційні моделі динамічних процесів та систем самої різної природи.

Звернемо увагу на те що математичний апарат мереж Петрі досить гнучкий. Це дає можливість адаптувати його для опису дискретно-подієвих систем. Для прикладу: існує можливість тимчасової затримки спрацювання позиції. Така особливість дає змогу реалізувати більш складні алгоритми руху міток по мережі. Також можна вводити власні правила для спрацьовування позиції. Ця процедура реалізується через спеціальне вікно Rules Editor. За допомогою простого синтаксису реалізована можливість поміщувати маркери у певні позиції.

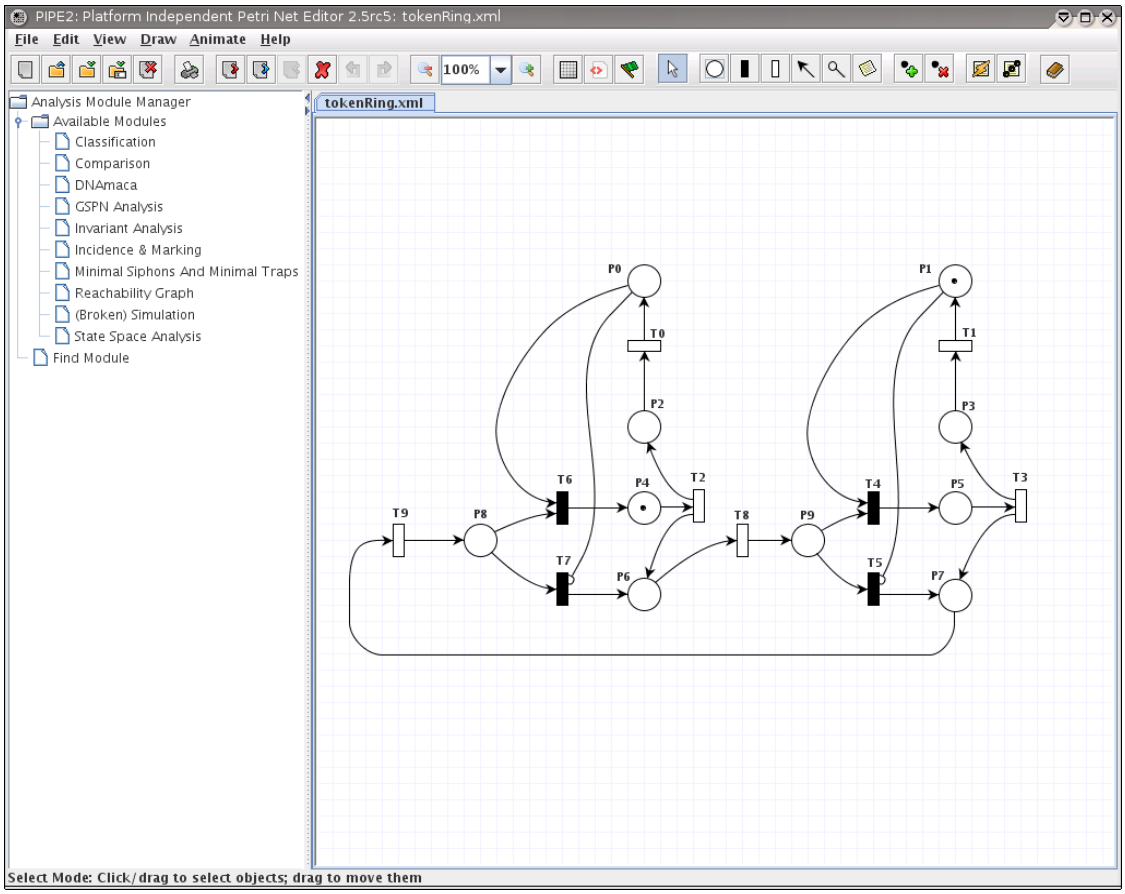


Рис. 5.5. Інтерфейс та робоча зона Platform Independent Petri net Editor (PIPE2)

Для кожного переходу створені умови, що дають змогу задавати вагові коефіцієнти. Існує також можливість реалізувати режим пріоритету – це дає змогу уникнути колізій. За однакових значень пріоритету перехід працює як у типовій стохастичній мережі Петрі. Реалізована можливість задання інгібіторної (зворотної) дуги на певні типи переходів. Також є функція об'єднання кількох елементів у групу, де задані лише входи та виходи.

Інший актуальний сервіс має назву DCNet. Цей засіб моделювання володіє спектром можливостей для активації мереж Петрі. Існує також можливість будувати на їх основі дискретно-безперервні системи з керованою структурою. DCNet призначена:

- для моделювання та дослідження різних виробничих процесів;

- для проєктування різноманітних тренажерів;
- для розробки експертних систем.

Основними режимами роботи програми є:

- редагування;
- маркування мережі;
- моделювання.

Режим редагування призначений для графічного введення топологічної структури, елементів та параметрів дискретно-безперервної мережі. Також цей режим дає змогу оперативно отримувати інформацію про структуру схеми та характеристики параметрів її елементів, додавати та видаляти елементи, змінювати їх параметри, кількість входів та виходів, а також з'єднання між ними. Існує також можливість будувати ієрархічні моделі та здійснювати копіювання чи переміщення окремих блоків та груп зі збереженням зв'язків, що прискорює конструювання схеми.

5.2. Модель керування запасами

Теорія управління матеріальними запасами має давню історію. Насамперед такі моделі дуже сильно впливають на режим системності постачання матеріальних ресурсів виробничим підприємствам та організаціям. Важливо оптимізувати рівень наявних матеріальних запасів, і водночас організувати процедуру безперебійного постачання наявних сировинних ресурсів. Вплив управління запасами на ланцюг поставок тісно пов'язаний з постачальниками, виробниками, центрами розподілу, роздрібною торгівлею, наявністю торговельних центрів та покупців. Усі вони мають свої специфічні вимоги до наявних сировинних ресурсів. Отже, постачальник має справу із сировиною або комплектуючими чи виробами, а також з матеріалами для технічного обслуговування, ремонту та експлуатації. Переважно ці ресурси використовуються для підтримки режиму нормального функціонування економіки. Коли постачальник відправляє товари замовнику, то зазвичай запаси перебувають певний час у дорозі. Аналогічна ситуація складається і для виробника, який виготовляє товари для технічного обслуговування, ремонту та експлуатації. Коли виробник відвантажує товар, то цей товар за звичай надходить до розподільного центру. Цей центр може реалізувати свої товари безпосередньо у режимі роздрібної торгівлі або зразу відправляти замовнику. Ресурси такого типу називаються транзитними. У розподільному центрі зазвичай матеріали, комплектуючі чи сировина не зберігаються. У такому центрі можлива процедура пакування товарів. Зазвичай центр розподілу товарів відправляє їх на ринок. У торгових підприємствах здійснюється передпродажна перевірка товарів, що надходять.

Припустимо, існує необхідність провести оцінку системи керування запасами товарів деякого торгового підприємства. Із практики відомо, що попит на товари виникає через певні інтервали часу. За наявності товару в магазині покупець має можливість здійснити покупку. Якщо його вимоги не задовольняються, то зростає негативне ставлення до цього торговельного підприємства. З часом такий негатив може наростати, що, врешті-решт, може стати причиною банкрутства підприємства. Зауважимо, що торговельне підприємство має обмежені можливості зберігати товар у складських приміщеннях. Припустимо, що максимальний рівень запасу товарів, які зберігаються на складах, становить N одиниць. Стратегія прийняття рішень про поповнення запасів полягає у періодичному перегляді їх рівня. Якщо під час оцінки ситуації із запасами товару виявилось, що кількість товарів на складі менша за m штук, то приймається рішення про поповнення запасу товарів і, очевидно, здійснюється замовлення на додаткову доставку необхідної продукції. Процедура доставки товарів здійснюється протягом певного часу і періодично. Об'єм товарних одиниць, що замовляються, дає змогу збільшити запаси до певного рівня.

У цьому випадку завданням моделювання є встановлення такої стратегії прийняття рішень, яка дає змогу забезпечити оптимально ефективне функціонування торгового підприємства з погляду максимізації у задоволенні потреб споживачів. Саме максимізація задоволення потреб населення є цільовою функцією сформульованої стратегії.

Зауважимо, що процедура реалізації товару складається із ланцюга «замовлення товару → реалізація товару → замовлення товару». Проте товар може бути відсутній, тому спрацьовує інша конструкція «замовлення товару → відмова у продажу товару через його відсутність». Внаслідок здійснення першої послідовності підраховується кількість проданих товарів та рівень задоволення попиту на товар. Якщо ж подія «реалізація товару» не здійснюється, то здійснюється подія «відмова у продажу товару через його відсутність». Через здійснення такої негативної події підраховується рівень нереалізованого попиту на необхідну продукцію. Зрозуміло, що рівень задоволення споживацького попиту населення повинен бути значно вищий за рівень нереалізованого попиту.

Ясно, що торговельна організація прагне до оптимізації процесу задоволення попиту споживачів, в ідеалі доводячи його до 100 %. Отже, потрібні своєчасні управлінські вказівки в системі безперервного поповнення торгового запасу. Для цього необхідно реалізувати такі функції:

1) постійний електронний моніторинг стану товарних запасів;

2) оцінка рівня достатності товарних запасів;

3) оцінка ступеня недостатності товарних запасів;

4) поповнення товарних запасів до необхідного рівня.

На початковій стадії моделювання будемо вважати, що запас достатній, тому показник знаходиться у позиції «достатній запас». Оскільки умова «постійний електронний моніторинг стану товарних запасів» виконана, то можливий розвиток подій такий. За позитивного виконання умови 2 (рівень запасів достатній) торговельна організація запасів не поповнює та працює у своєму нормальному режимі. Якщо ж згідно з функцією 3 виявляється, що рівень запасів недостатній, то відбувається перехід для реалізації четвертої позиції. Рішення про достатній стан запасу товарів приймається, якщо позиція «запас» містить не менше m маркерів. У протилежному випадку приймається рішення про недостатній стан запасу і здійснюється подія «поповнення запасів до необхідного рівня». Внаслідок реалізації цієї функції здійснюється доставка товарів, і в позицію «запас» надходить відповідна кількість маркерів. Поповнення запасу відбувається до максимального рівня N штук товарів. Отже, кількість товарів, що доставляються, розраховується як максимальна кількість товарів мінус кількість товарів у запасі M, тобто N–M одиниць товару. Подібний аналіз дає змогу отримати мережу Петрі, яка представлена на рис. 5.6. Наведена модель мережі Петрі дає реальну можливість активно керувати запасами: не накопичувати їх на складах, з одного боку, а з іншого – максимально оперативно та у повному обсязі задовольняти потреби споживачів.

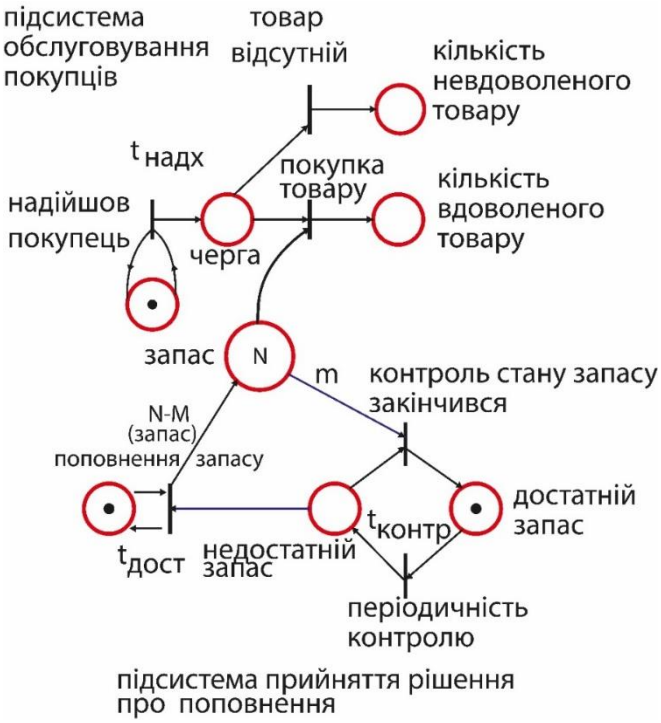


Рис. 5.6. Модель мережі Петрі керування запасати торгового підприємства

5.3. Специфіка комп'ютерних мереж

Комп'ютерні мережі можна класифікувати за різними прикметами:

- за топологією – кільцеві, деревоподібні, шинні чи зіркоподібні;
- за типом технічних засобів передавання даних – можуть використовуватись, наприклад, волоконно-оптичні кабелі;
- за типом застосування – міжконтинентальні, комерційні чи промислові мережі;
- за сферою функціонування – локальні, регіональні чи глобальні.

Навіщо взагалі використовують комп'ютерну мережу? По-перше, комп'ютери, об'єднані у мережу, можуть використовувати одне і те саме програмне забезпечення а також обмінюватись копіями файлів. По-друге, з'являється можливість більш ефективно використовувати вивільнений об'єм дискового простору на комп'ютерах. І нарешті, зменшується вартість обслуговування всіх персональних комп'ютерів, оскільки легко здійснити адміністрування всієї мережі.

Охарактеризуємо тепер різні види мереж з погляду їх локалізації:

- Локальні мережі (Local Area Network) орієнтовані на передавання інформації в межах недалеко розташованих офісних приміщень чи житлових будинків. У якості прикладів таких мереж можна назвати Token Ring, Ethernet, Gigabit Ethernet та Fast;
- Корпоративні мережі являють собою об'єднання декількох локальних мереж. Для організації такого об'єднання використовуються зазвичай телефонні чи супутникові канали;
- Регіональні мережі (Metropolitain Area Network) охоплюють місто, область, регіон;
- Глобальні мережі (Wide Ares Network) охоплюють всю планету Земля. Такі мережі володіють високою швидкодією, передаючи десятки терабіт за секунду.

Локальна обчислювальна мережа (ЛОМ) – це сукупність кабелів, що з'єднують комп'ютери та мережеві адаптери (рис. 5.7). Останні працюють під управлінням мережевої операційної системи з використанням прикладного програмного забезпечення. Кожен комп'ютер в ЛОМ називається робочою станцією. Крім робочих станцій, у ЛОМ обов'язково знаходиться спеціально виділений потужний комп'ютер, який виконує функції файл-сервера. У межах ЛОМ функціонує мережеве програмне забезпечення. Взаємодія між ПК в ЛОМ здійснюється через реалізацію методу доступу. Зокрема, методом доступу в Ethernet є Carrier Sense Multihle Access with Collision Detection (CSMA/CD). Цей метод дає змогу кожній станції передавати інформацію у будь-який потрібний момент.

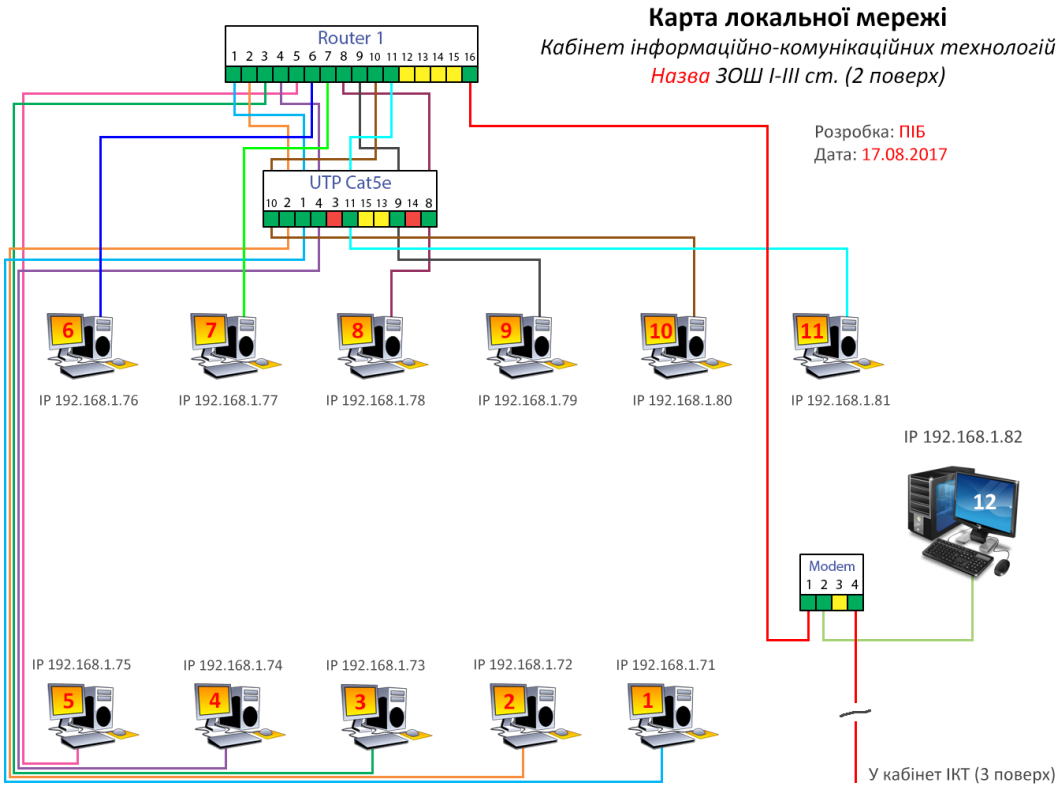


Рис. 5.7. Наочна схема ЛОМ

Метод доступу CSMA/CD базується на двох правилах. Перше правило полягає у тому, що станція постійно «прослуховує» канал: якщо канал вільний, то станція починає передавання, у протилежному випадку – чекає своєї черги. Друге правило стосується ситуації, яка має назву колізії, що фактично означає взаємне блокування передавань станціями своїх повідомлень. Якщо виявлена колізія, то станція через деякий інтервал затримки, дочекавшись звільнення середовища, передає своє повідомлення.

Корпоративні мережі являють собою об'єднання локальних мереж. Засобами об'єднання ЛОМ є, наприклад, повторювачі (repeaters). Вони пропускають через себе потоки між мережами. Їх просто розміщують між різними ЛОМ.

Функціонально близькими до повторювачів є концентратори і так звані хаби (hubs).

Мости та комутатори являють собою апаратно-програмні блоки, спрямовані на те, щоб сполучати різні ЛОМ з різними середовищами передавання та діючими у них протоколами. Розрізняють внутрішні та зовнішні мости. Внутрішній міст являє собою кілька адаптерних плат, встановлених на одному сервері. Така конструкція дає можливість сполучати декілька ЛОМ з різними середовищами та протоколами. Для мосту використовується спеціальне програмне забезпечення.

Зовнішній міст реалізується у спеціальній машині. Розрізняють два види таких мостів – призначені та непризначені (dedicated or nondedicated). Перші, крім безпосередньо функції мосту, виконують спектр прикладних функцій. Збої в роботі прикладного процесу перешкоджають роботі непризначеного мосту і впливають на транзакції.

Комутатор (switch) є функціонально близьким до мосту. Цей пристрій працює як високошвидкісний багатопортовий міст. Механізм комутації дає можливість здійснювати сегментування ЛОМ та виділяти смугу пропускання кожній станції.

Маршрутизатори (routers) являють собою такі апаратно-програмні пристрої, що дають змогу сполучати різні ЛОМ та виконують також функції інформаційних потоків. Маршрутизатори поділяються на статичні та динамічні. Статичні мають стабільні таблиці (шляхи) маршрутизації. Динамічні базуються на принципах оптимізації мережі, тобто використовують алгоритми типу Дейкстри чи Флойда. Ці алгоритми прокладають оптимальні маршрути у мережах так само як і у графах.

Шлюзи (gateway) являють собою порт колективного доступу, що об'єднує кілька мереж.

Зупинимось тепер коротко на характеристиках глобальних мереж. Однією з найпоширеніших таких мереж є Asynchronous Transfer Mode (ATM). У перекладі це звучить як режим асинхронного передавання. Головні властивості означеної мережі такі:

- це мережа з інтеграцією послуг; це означає, що інформація передається єдиним потоком з різними вимогами до затримок у режимі передавання;

- це пакетна мережа з віртуальними каналами – у цій мережі використовується пакет з фіксованим розміром 53 байти; назва цього пакету – комірка (cell): це дає змогу реалізувати спектр функцій маршрутизації та суттєво зменшити протяжність опрацювання кожної комірки;

- ця мережа описує тільки інтерфейсні характеристики і з метою передавання даних використовує багато реальних каналів та комунікаційних мереж;

- якість та швидкість передавання інформації задається за потребою користувача; мережа працює з двома видами каналів – вузькосмуговими (діапазон від 9 800 біт/с до 2 Мбіт/с) та широкосмуговими (діапазон від 2 до 622 Мбіт/с).

Насамкінець додамо, що АТМ-мережа досить гнучка в експлуатації. У випадку виникнення у ній збою вмикається механізм автоматичного відшукування альтернативних шляхів передачі інформації, що гарантує безперебійність роботи такої мережі.

РОЗДІЛ 6
ГРАФИ. МОДЕЛЮВАННЯ МЕРЕЖ РІЗНОЇ ПРИРОДИ

6.1. Основні поняття теорії графів

Що таке граф? Уявімо площину, на якій зображені крапки або кільця, з'єднані лініями, що називаються ребрами графа. Крапки (кільця) називаються вершинами. Це і є граф. Символічно позначатимемо граф як G(V,E). Тут $V = (v_1,v_2,v_3, \ldots\ldots,v_n)$ – множина вершин графа. Позначення V символізує собою слово *Vertice,* що означає англійською мовою «вершина». $E = (e_1,e_2,e_3,\ldots.., e_m)$ – множина ребер графа (E – стартова буква слова Edge, що перекладається з англійської як ребро). Означення графа можна сформулювати так:

Означення 6.1. Граф G(V,E) – це геометрична фігура, що складається із множини вершин V та множини ребер E.

Для прикладу на рис. 6.1 представлений так званий зважений граф. Чому «зважений»? Очевидно, тому, що біля ребер цього графа проставлені цифри – їх ще називають вагами. Взагалі є досить багато різних видів графів.

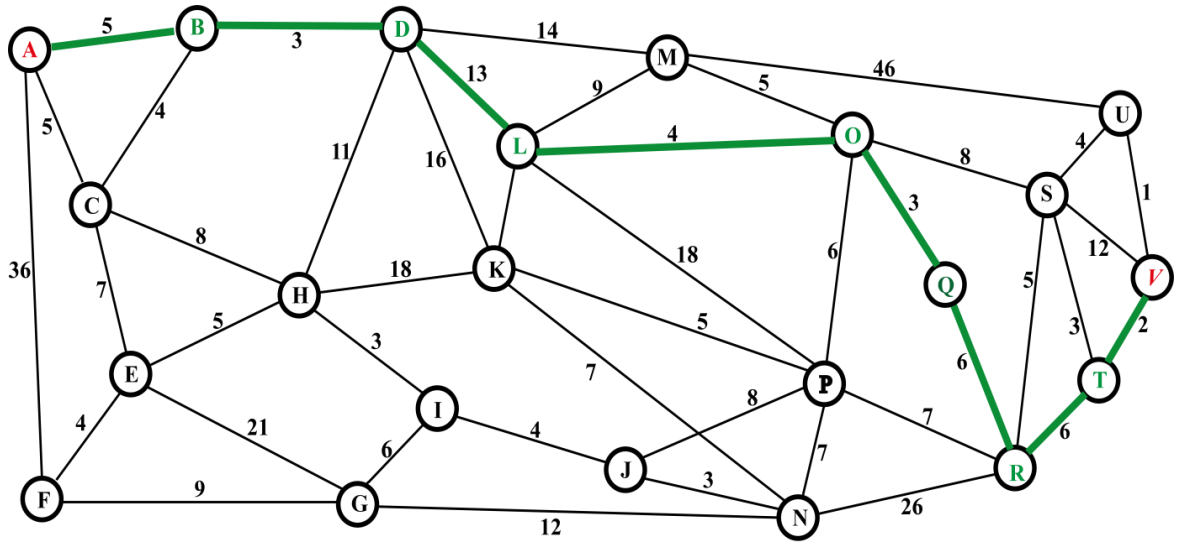


Рис. 6.1. Неорієнтований зважений плоский монограф

Розберемо, наприклад, визначення «орієнтований плоский мультиграф». На рис. 6.2 представлений саме такий граф. Що означає ця назва? Що значить «орієнтований»? Якщо кожне ребро графа буде мати напрямок, то це буде орієнтований граф; такий напрямок зображується стрілкою, що ставиться на самому ребрі. Слово «плоский» означає граф, що лежить в одній площині, так що ребра його не перетинаються між собою. І нарешті, ознака «мультиграф»

визначається префіксом «мульти», тобто «багато». Але багато чого? Це стосується ребер: дві сусідні вершини з'єднуються не одним, а кількома ребрами.

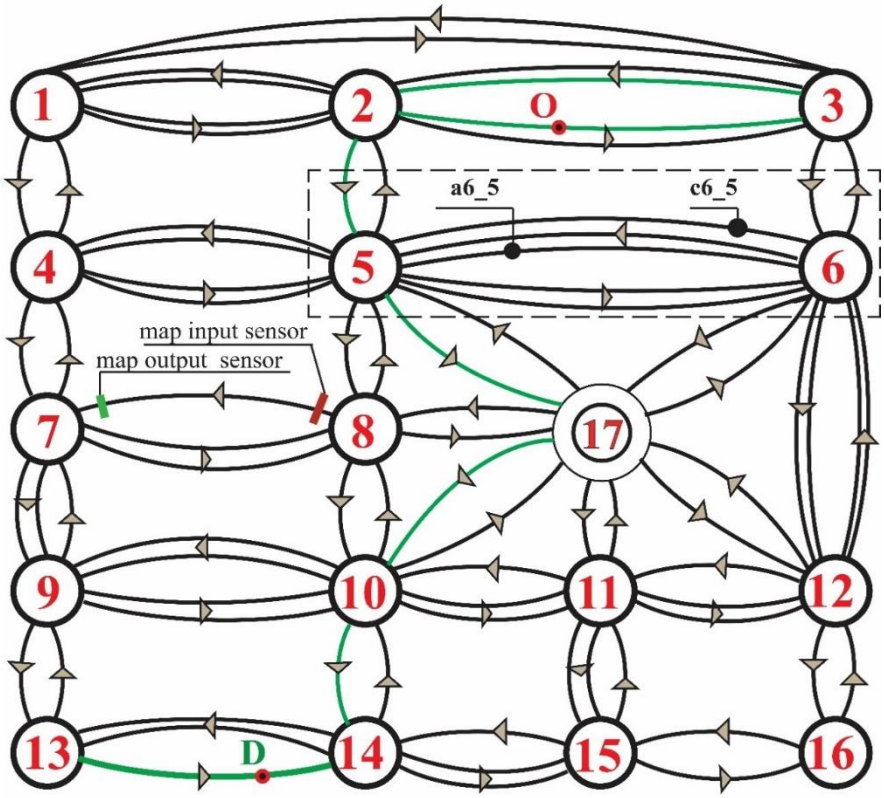


Рис. 6.2. Орієнтований плоский мультиграф

Продовжуючи аналіз назв графів як геометричних зображень, представлений на рис. 6.1, граф можна назвати неорієнтованим плоским навантаженим монографом. Зробимо деякі коментарі до рис. 6.1. Вершини зображено у вигляді кілець, всередині яких проставлені літери латинського алфавіту – це зроблено просто для зручності.

Чому графи важливі з погляду зору моделювання систем і мереж? Річ у тім, що графи моделюють різноманітні мережі – транспортні, електричні, газорозподільні, інтернет-мережі, комунальні тощо. Отже, графи є теоретичною основою логістики. Для графів створена потужна аналітична та програмна бази. Це означає, що використовуючи різні програмні алгоритми, ми можемо, зокрема, прокладати оптимальні маршрути у графах і відповідно в реальних діючих мережах, які моделюються цими графами. Наприклад, потрібно прокласти маршрут у графі. Але якщо ми завели мову про маршрут, то тут потрібно зауважити, що граф може, наприклад, зображувати певний реальний транспортний маршрут. Дамо означення маршруту:

Означення 6.2. Послідовність вершин і ребер, що утворюють нерозривну лінію, називається маршрутом. Маршрут можна представити, просто вказавши послідовність вершин.

Наприклад, маршрут на рис. 6.1 можна зобразити у такий спосіб:

$$L \to B \to D \to L \to O \to Q \to R \to T \to V .$$

Серед спектру можливих маршрутів нас цікавить оптимальний, на проходження якого витрачається, наприклад, найменший час. Якщо граф представляє транспортну мережу – країни, міста, континенту тощо, то вибір оптимального маршруту проїзду між будь-якими об'єктами є досить актуальною проблемо. І тут на допомогу приходить теорія графів. Отже, проклавши оптимальний маршрут у графі шляхом використання програмних алгоритмів можна прокладати такі маршрути в реальних мережах різної природи. Ось чому теорія графів – це реальна практична наука, що являє собою фундаментальну основу логістики і не тільки. Спектр застосувань цієї науки будемо розглядати далі.

Знову проаналізуємо рис. 6.1. Тут представлений виділений маршрут від вершини A до вершини V. Взагалі прокласти маршрут типу A→V можна дуже великою кількістю способів. Але нас цікавить найбільш оптимальний маршрут. Отож, чому маршрут A→V є особливим на фоні інших можливих варіантів? Річ у тім, що сума ваг ребер цього маршруту є мінімальною величиною з усіх можливих варіантів прокладання шляху між вказаними позиціями. Інакше кажучи, цей маршрут є найкоротшим (геометрично) зі всіх можливих. Розрахунок виділеного шляху (рис. 6.1) проведений з допомогою так званого A*-алгоритму (читається A-стар). Про нього йтиметься далі.

Під час використання алгоритмів теорії графів часто доводиться стикатися з так званими матрицями інциденцій та суміжності. Охарактеризуємо спочатку матрицю інциденцій графа, представленого на рис. 6.3, де зображений простий неорієнтований незважений граф. Слово «простий» символізує той факт, що цей граф не має кратних ребер на противагу графу, зображеному на рис. 6.2.

Матриця інциденцій графа, представленого на рис. 6.3, представляється так:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}
1	1	0	1	0	0	0	0	0	0	0	0
2	1	1	0	1	1	0	0	0	0	0	0
3	0	1	0	0	0	0	1	0	0	0	0
4	0	0	0	1	0	1	0	0	0	1	1
5	0	0	0	0	1	0	0	0	1	1	0
6	0	0	0	0	0	1	0	1	1	0	0
7	0	0	0	0	0	0	1	1	0	0	1

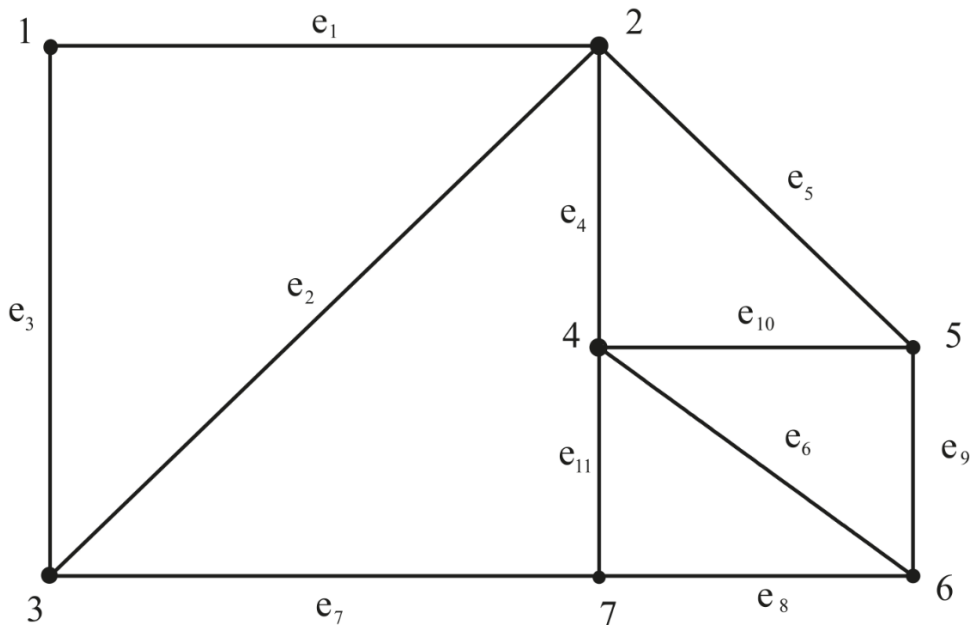


Рис. 6.3. Простий плоский неорієнтований незважений граф

Визначимо матрицю інциденцій простого неорієнтованого графа так:

$$m_{ij} = \begin{cases} 1, \textit{ якщо вершина } v_i \textit{ інцидентна до ребра } e_j \\ 0, \textit{ в протилежному випадку} \end{cases}$$

По горизонталі у наведеній вище матриці інциденцій виписані позначення всіх ребер графа, а по вертикалі у крайньому лівому стовпчику – позначення всіх вершини графа. Отже, матрицю інциденцій слід розуміти так: якщо ребро підходить до вершини (кажуть, що вершина є інцидентною до ребра), то на перетині відповідних рядків та стовпчиків виставляють 1, у протилежному випадку ставлять нуль. Звернемо також увагу на те, що матриця інциденцій є так званою булевою матрицею, оскільки складається лише з нулів та одиниць.

Далі охарактеризуємо дуже часто використовувану в програмних алгоритмах теорії графів матрицю суміжності.

Означення 6.3. Матриця суміжності – це квадратна матриця, стовпцям і рядкам якої відповідають вершини графа.

Визначимо матрицю суміжності так:

$$m_{ij} = \begin{cases} 1, \textit{ якщо } (v_i, v_j) \in E \\ 0, \textit{ у протилежному випадку} \end{cases}.$$

З допомогою матриці суміжності граф на рис. 6.3 представляється у такому вигляді:

	1	***2***	***3***	***4***	***5***	***6***	***7***
1	0	1	1	0	0	0	0
2	1	0	1	1	1	0	0
3	1	1	0	0	0	0	1
4	0	1	0	0	1	1	1
5	0	1	0	1	0	1	0
6	0	0	0	1	1	0	1
7	0	0	1	1	0	1	0

Звернемо увагу, що кількість одиниць у кожному рядку матриці дорівнює степеню відповідної вершини, адже номер рядка відповідає номеру вершини. Наприклад, вершина ***7*** має степінь три: відповідно і у сьомому рядку матриці суміжності стоїть три одиниці.

У програмних алгоритмах теорії графів часто використовується так звана вагова матриця. Для ілюстрації представлення цієї матриці розглянемо зважений граф на рис. 6.4.

Елементи вагової матриці задаються такою формулою:

$$a_{ij} = \begin{cases} 0, \text{ якщо } v_i = v_j \\ \infty, \text{ якщо } v_i \text{ та } v_j \text{ не з'єднані дугою} \\ d, \text{ якщо дуга } v_i v_j \text{ має вагу } d \end{cases}.$$

Отже, для графа на рис. 6.4 матимемо вагову матрицю, зображену зразу за цим рисунком:

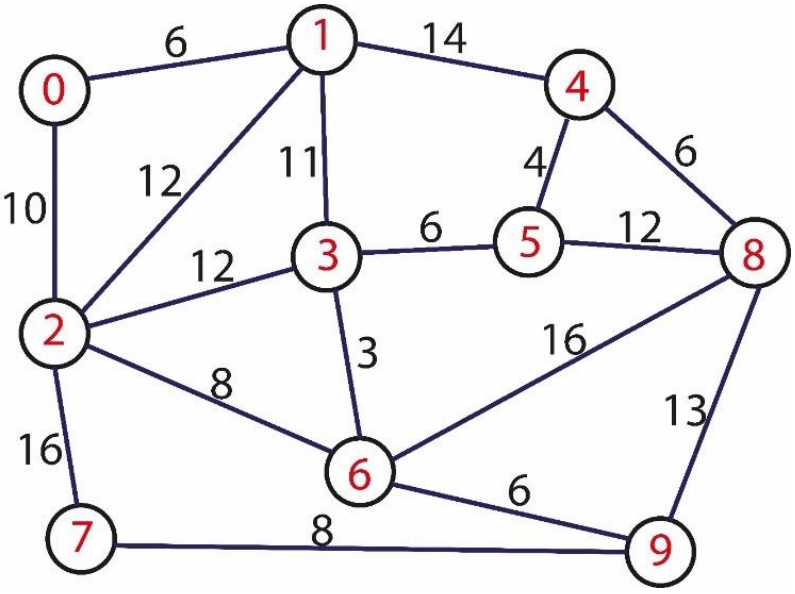


Рис. 6.4. Плоский зважений неорієнтований граф

Звернемо увагу на такі обставини:

- всі елементи головної діагоналі матриці рівні нулю;
- матриця симетрична відносно головної діагоналі;
- стовпчики цифр зліва і зверху, розташовані за межами самої матриці, записані для зручності зчитування елементів матриці. Ці цифри символізують собою номери вершин, з одного боку, а з іншого – задають індекси елементів матриці.

Вагова матриця часто використовується в програмній реалізації алгоритмів. Проте в таких ситуаціях зрозуміло, що символ нескінченності не може бути використаний. Тому у матрицю вводять заміну символу ∞ на достатньо велике число, наприклад, 999. Для ілюстрації практики використання програмних алгоритмів у теорії графів розглянемо так званий алгоритм Прима. Цей алгоритм є дуже ефективним для знаходження мінімального кістякового дерева. Кістякове дерево зображує лише ті ребра заданого графа, сума довжин яких є мінімальною. Знання такого дерева дає змогу розв'язати надзвичайно актуальні логістичні задачі. Тепер коротко про дерева (рис. 6.5).

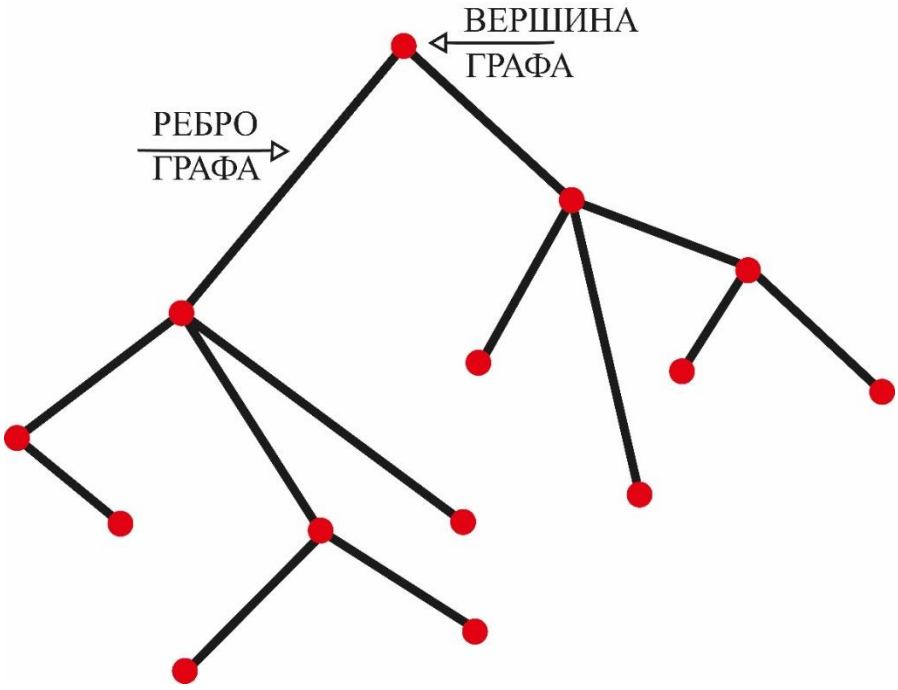


Рис. 6.5. Дерево – специфічний ациклічний граф

Дерева – найпоширеніший і специфічний клас графів. Цьому класу притаманна низка особливих властивостей, що відрізняє їх від інших видів графів. Специфіка дерев ще і у тому, що вони найчастіше використовуються в програмуванні. Дерево – це зв'язний ациклічний граф, тобто граф, у якому немає циклів. На рис. 6.5 показано саме такий граф. Тепер розглянемо зв'язний граф, показаний на рис. 6.6, що містить 6 вершин.

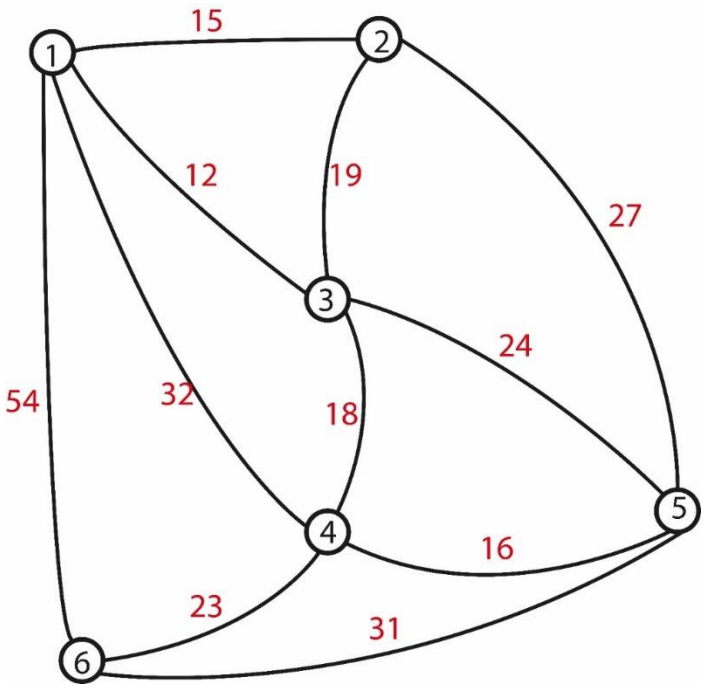


Рис. 6.6. Зв'язний граф, у якому з допомогою алгоритму Прима знаходять мінімальне кістякове дерево (каркас)

Логічним буде представити програмний варіант алгоритму Прима (Лістинг 6.1).

Лістинг 6.1

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class Prims {
   private boolean unsettled[];
   private boolean settled[];
   private int numberofvertices;
   private int adjacencyMatrix[][];
   private int key[];
   public static final int INFINITE = 999;
   private int parent[];
   public Prims(int numberofvertices) {
      this.numberofvertices = numberofvertices;
      unsettled = new boolean[numberofvertices + 1];
      settled = new boolean[numberofvertices + 1];
      adjacencyMatrix = new int[numberofvertices + 1][numberofvertices + 1];
      key = new int[numberofvertices + 1];
      parent = new int[numberofvertices + 1];
   }
```

```
    public int getUnsettledCount(boolean unsettled[]) {
        int count = 0;
        for (int index = 0; index < unsettled.length; index++) {
            if (unsettled[index]) {
                count++;
            }
        }
        return count;
    }
    public void primsAlgorithm(int adjacencyMatrix[][]) {
        int evaluationVertex;
        for (int source = 0; source <= numberofvertices; source++) {
            for (int destination = 1; destination <= numberofvertices; destination++)
{

this.adjacencyMatrix[source][destination] = adjacencyMatrix[source][destination
];
            }
        }
        for (int index = 1; index <= numberofvertices; index++) {
            key[index] = INFINITE;
        }
        key[1] = 0;
        unsettled[1] = true;
        parent[1] = 1;
        while (getUnsettledCount(unsettled) != 0) {
            evaluationVertex = getMimumKeyVertexFromUnsettled(unsettled);
            unsettled[evaluationVertex] = false;
            settled[evaluationVertex] = true;
            evaluateNeighbours(evaluationVertex);
        }
    }
    private int getMimumKeyVertexFromUnsettled(boolean[] unsettled2) {
        int min = Integer.MAX_VALUE;
        int node = 0;
        for (int vertex = 1; vertex <= numberofvertices; vertex++) {
            if (unsettled[vertex] == true && key[vertex] < min) {
                node = vertex;
                min = key[vertex];
            }
```

```
        }
        return node;
    }
    public void evaluateNeighbours(int evaluationVertex) {
        for (int destinationvertex = 1; destinationvertex <= numberofvertices;
destinationvertex++) {
            if (settled[destinationvertex] == false) {
                if (adjacencyMatrix[evaluationVertex][destinationvertex] !=
INFINITE) {
                    if (adjacencyMatrix[evaluationVertex][destinationvertex] <
                            key[destinationvertex]) {

key[destinationvertex] = adjacencyMatrix[evaluationVertex][destinationvertex];
                        parent[destinationvertex] = evaluationVertex;
                    }
                    unsettled[destinationvertex] = true;
                }
            }
        }
    }
```

Запустимо цю програму. На консолі отримаємо такий результат:

```
Enter the number of vertices
 6
Enter the Weighted Matrix for the graph
0     15     12     32     999     54
15     0     19     999     27     999
12     19     0     18     24     999
32     999     18     0     16     23
999     27     24     16     0     31
54     999     999     23     31     0
SOURCE : DESTINATION = WEIGHT
1       :       2       =       15
1       :       3       =       12
3       :       4       =       18
4       :       5       =       16
4       :       6       =       23
Process finished with exit code 0
```

На рис. 6.7 показано мінімальне кістякове дерево графа, зображеного вище (рис. 6.6). Отож, спочатку у консоль запущеної програми вводимо кількість вершин графа, потім уже знайому нам вагову матрицю. Внаслідок цього отримуємо 5 ребер та їх ваги, що власне формують спектр ребер, сума ваг яких є мінімальною зі всіх можливих. Представимо тепер зображення мінімального кістякового дерева (каркасу). Ребра отриманого з допомогою алгоритму Прима дерева виділені зеленим кольором. Сума їх довжин є мінімальною зі всіх можливих варіантів. Тонкі лінії разом з товстими утворюють сам граф, у якому виділено кістякове дерево. Зрозуміло, що кістякових дерев для будь-якого графа може бути декілька. Але для нас особливо цінним є мінімальне кістякове дерево.

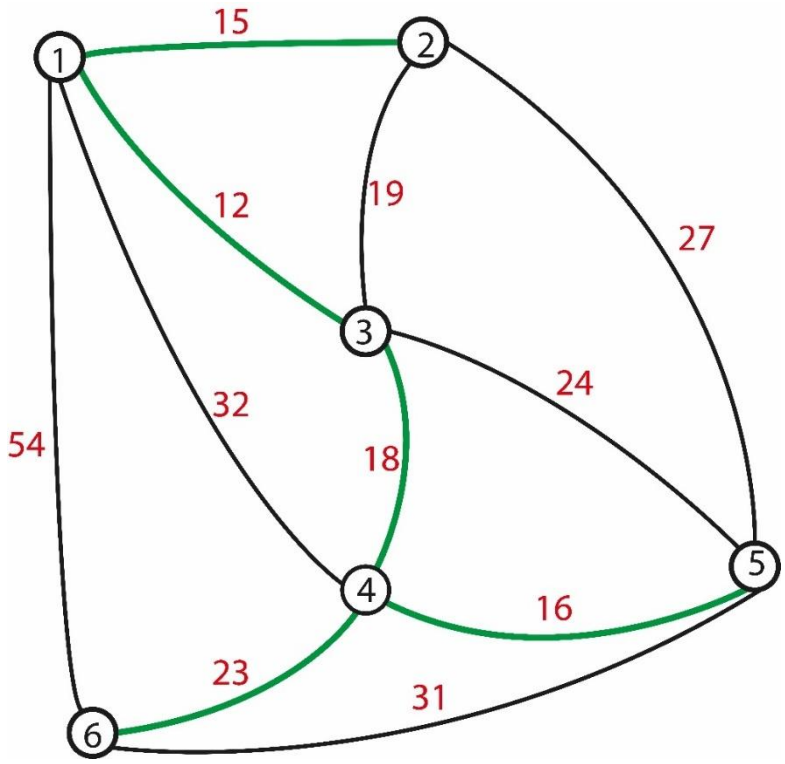


Рис. 6.7. Мінімальне кістякове дерево (каркас) графа.
Потовщені зелені лінії є ребрами каркасу графа, представленого на рис. 6.6

6.2. Графи як моделі систем та мереж

Звернемо увагу на одну з останніх розробок науки про складні системи. Йдеться про мережі. Наука про складні мережі стрімко зростає і напрацьовує нові перспективи, дослідницькі варіанти та аналітичні інструменти з метою вивчення різних видів систем. Ця наука застосовується для наукових дисциплін різноманітного спектру (економіка, політологія, прикладна фізика, біологія, екологія, соціологія, наука управління, інженерія, медицина тощо).

Історичне коріння мережевої науки можна шукати в декількох дисциплінах. Однією з них є, очевидно, дискретна математика, особливо *теорія графів,* де математики вивчають різні властивості абстрактних структур, які називаються *графами,* сформованими з *вузлів* і *ребер*. Важливо також у цьому сенсі згадати статистичну фізику, яка розглядає властивості колективних систем, що складаються з великої кількості сутностей (наприклад, фазових переходів). Вивчення таких фізичних сутностей здійснюється саме за допомогою аналітичних засобів. Важливий аспект застосування мережевої науки – соціальні мережі. Необхідно також акцентувати на динамічних системах, що використовуються в біології та роботі штучних нейронних мереж. У всіх цих дослідженнях розглядаються зв'язки та взаємодії між компонентами системи, а не лише на якомусь вибраному компоненті.

Звернемо увагу на те, що мережеві моделі відрізняються від інших більш відомих динамічних моделей за низкою важливих параметрів. По-перше, компоненти системи можуть не з'єднуватися рівномірно і регулярно, як це відбувається для клітин у клітинних автоматах (розділ 2, п. 2.5), що утворюють регулярні однорідні сітки. Сказане свідчить про те, що у деякій мережі певні компоненти можуть бути надійно з'єднані, а інші – ні. Така неоднорідна зв'язність ускладнює математичний аналіз властивостей системи. Зокрема, наближення середнього поля непросто застосовати до подібного роду мереж. Водночас сказане дає моделі більшу можливість для представлення з'єднань між компонентами системи більш тісно з реальністю. Можна представляти будь-яку структуру мережі, вказавши докладно, які компоненти підключені до інших складників і як здійснено це підключення. Це робить мережеве моделювання актуальним. Незалежно від того, генерується мережа за допомогою певного математичного алгоритму чи отримана з аналізу реальних даних, створена мережева модель буде містити об'ємний діапазон інформації про те, як саме підключені компоненти. Ось чому необхідно здобути навички ефективно створювати, керувати та маніпулювати заданими фрагментами інформації.

Кількість компонентів мережі може суттєво збільшуватися або спадати з часом у деяких динамічних моделях мережі. Подібні збільшення або спадання є основним припущенням, яке зазвичай постулюється в генеративних мережевих моделях. Саме ці моделі проливають світло на процеси самоорганізації деяких мережевих топологій. Звернемо увагу, що така динамічна зміна кількості компонентів у певній системі реалізує процедуру відходу від інших більш звичайних моделей динамічних систем. Це відбувається тому, що під час розгляду станів компонент системи наявність ще однієї компоненти означає, що фазовий простір системи збільшується на одиницю. З погляду типових динамічних систем це виглядає не зовсім логічно, оскільки змінювати розміри фазового простору сис-

теми з плином часу досить проблематично. Однак, такі ситуації відбуваються у багатьох реальних складних системах. Мережеві моделі дають змогу закономірно описати такі складні процеси.

6.3. Моделювання мереж: алгоритм Дейкстри

Існує багато способів знаходження мінімального шляху у графі. Точніше, оптимального маршруту. Під оптимальністю можна розуміти, наприклад, мінімальний час проходження маршруту, чи мінімальні витрати палива, витраченого на проходження маршруту, або мінімальну відстань. У цьому сенсі алгоритм Дейкстри досить ефективний та близький по суті і по принципу роботи до алгоритму Флойда. Але на відміну від останнього, дає змогу **знайти спектр найкоротших відстаней між вибраною вершиною графа та всіма іншими його вершинами.** Суть алгоритму Дейкстри полягає у порівнянні відстаней до сусідніх (інцидентних) вершин та знаходженні оптимального вибору на кожному кроці. Розглянемо рис. 6.8. Тут представлений зважений плоский простий неорієнтований граф. Вершини цього графа позначені літерами латинського алфавіту. Біля ребер графа виставлені їх ваги. Звернемо увагу, що ваги ребер зовсім не корелюють з їх геометричною довжиною, тому що їх визначення залежить від постановки конкретної задачі. На цю обставину ми звернемо особливу увагу далі. А зараз лише зазначимо, що вага ребра може являти собою будь-яку сутність. Важливо зауважити, що вага одного і того ж ребра часто буває змінною з часом величиною. Розглянемо для прикладу транспортні магістралі між містами. У погожий літній день такі транспортні артерії володіють великою пропускною здатністю, і автомобілі ними рухаються з великими швидкостями. Але в зимовий період під час снігопадів та заметілей ситуація докорінно змінюється – транспортні засоби застрягають у снігових заметах, і тому прохідність такої дороги (тобто ребра графа) різко зменшується. Із цього випливає, що потрібно створити базу даних, у якій зберігаються ваги ребер та постійно оновлювати і базу даних, і ваги ребер відповідно до реалій. Наочно динамічність ваг ребер проявляється у випадку із транспортною мережею у місті, де ситуація змінюється щосекундно та з’являються затори на перехрестях.

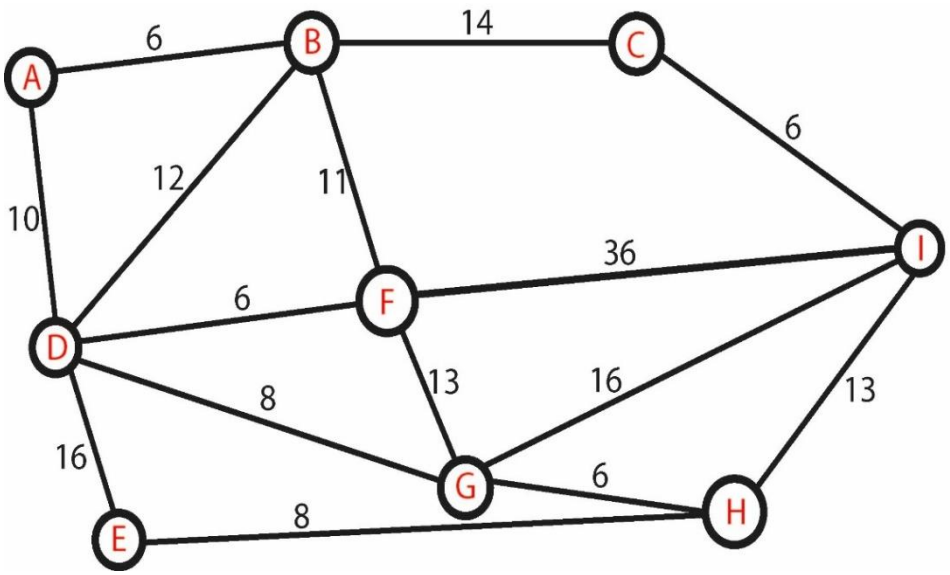


Рис. 6.8. Навантажений неорієнтований граф

Нехай за допомогою алгоритму Дейкстри необхідно знайти найкоротші відстані між вершиною A та всіма іншими вершинами (рис. 6.8). Запишемо для початку вагову матрицю для названого графа. Її вигляд представляється так:

$$a_{ij} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \end{array} \begin{array}{c} \begin{array}{ccccccccc} A & B & C & D & E & F & G & H & I \end{array} \\ \begin{pmatrix} 0 & 6 & \infty & 10 & \infty & \infty & \infty & \infty & \infty \\ 6 & 0 & 14 & 12 & \infty & 11 & \infty & \infty & \infty \\ \infty & 14 & 0 & \infty & \infty & \infty & \infty & \infty & 6 \\ 10 & 12 & \infty & 0 & 16 & 6 & 8 & \infty & \infty \\ \infty & \infty & \infty & 16 & 0 & \infty & \infty & 8 & \infty \\ \infty & 11 & \infty & 6 & \infty & 0 & 13 & \infty & 36 \\ \infty & \infty & \infty & 8 & \infty & 13 & 0 & 6 & 16 \\ \infty & \infty & \infty & \infty & 8 & \infty & 6 & 0 & 13 \\ \infty & \infty & 6 & \infty & \infty & 36 & 16 & 13 & 0 \end{pmatrix} \end{array}$$

По головній діагоналі цієї матриці стоять нулі. Символ Г свідчить про те, що вибрані вершини – не інцидентні. Звернемо також увагу на те, що ця мат-

риця симетрична, тобто $a_{ij} = a_{ji}$. Наприклад, $a_{24} = a_{42} = 12$. Виконання алгоритму Дейкстри будемо здійснювати покроково, заповнюючи поступово табл. 6.1.

Таблиця 6.1. Словесний алгоритм Дейкстри

Крок	**Відмічені вершини**	**Відстань до вершини**									**Невідмічені вершини**
		A	B	C	D	E	F	G	H	I	
1	A	0	6	∞	10	∞	∞	∞	∞	∞	B, C, D, E, F, G, H, I
2	B	0	6	20	10	∞	17	∞	∞	∞	C, D, E, F, G, H, I
3	D	0	6	20	10	26	16	18	∞	∞	C, E, F, G, H, I
4	F	0	6	20	10	26	16	18	∞	52	C, E, G, H, I
5	G	0	6	20	10	26	16	18	24	34	C, E, H, I
6	C	0	6	20	10	26	16	18	24	26	E, H, I
7	H	0	6	20	10	26	16	18	24	26	E, I
8	E	0	6	20	10	26	16	18	24	26	I
9	I	0	6	20	10	26	16	18	24	26	

Крок 1. Оскільки в задачі стоїть завдання знайти відстані від вершини A до всіх інших вершин (рис. 6.8), то це означає, що вершина A пройдена, і тому на першому кроці відмічаємо цю вершину як пройдену, а всі інші вершини графа записуємо у рядок невідмічених вершин (табл. 6.1). До того ж у першому рядку таблиці виставляємо відстані від вершини A до всіх інших вершин. Зрозуміло, що цей рядок буде співпадати із першим рядком матриці ваг.

Крок 2. Відмічаємо вершину B, оскільки вона є найближчою до A. Далі обчислюємо довжини шляхів, що ведуть від вершини A до невідмічених вершин через вершину B. Якщо нові значення виявляються меншими від старих, то змінюємо останні на нові. Через вершину B є три варіанти проходження: ABD, ABF та ABC. Спектр відповідних відстаней буде: 18, 17 та 20. Отже, попередні значення до вершин F та C замінюються на нові, а відстань до вершини D не змінюється. Значить, тепер другий рядок таблиці, порівняно з першим, зміниться у двох клітинках: 2-C та 2-F.

Крок 3. Звернемо увагу тепер на рядок № 2 табл. 6.1. Із вершин, що залишились невідміченими, тобто C, D, E, F, G, H, I, найближче до A знаходиться вершина D. Із A через вершину D у невідмічені вершини ідуть три ланцюги (ADF, ADG та ADE) з відповідними довжинами – 16, 18 та 26. Вносимо належні зміни у рядок № 3, врахувавши, що вершина D вже відмічена і прокладати до неї маршрут не можна. Знову ж таки знаходимо найменше значення серед невідмічених на цей момент вершин: це буде вершина F.

Крок 4. Через вершини D і F до непройдених вершин пролягає два шляхи – ADFI та ADFG з відповідними вагами 52 та 29. Відповідно у рядку № 4 змінюємо лише відстань до вершини I та знаходимо мінімальне значення для

непройдених вершин – воно дорівнює 18 і стосується вершини G. У цю вершину шлях пролягав через вершину D – відповідно маємо два ланцюги: ADGI, ADGH з довжинами 34 та 24 відповідно. Вносимо тепер зміни у рядок № 5. Аналізуючи цей рядок, доходимо висновку, що вершині C відповідає мінімальна величина – 20.

Крок 5. Через вершину C проходить лише один ланцюг – ABCI довжиною 26. Тому вносимо відповідні зміни у рядок № 6. У цьому самому рядку знаходимо мінімальне значення для непройдених вершин E, H, I. Воно виявляється рівним 24 та стосується вершини H. Проглядаємо відповідні ланцюги – ADGHI та ADGHE. Їм відповідають довжини 37 та 32 відповідно. Бачимо, що у цьому випадку ніяких змін вносити не треба.

Крок 6. Вибираємо вершину E. Через цю вершину проходить лише один маршрут – ADEH з довжиною 34. Тому змін теж немає ніяких. Далі вибираємо вершину I. У цьому випадку ситуація теж незмінна. Взагалі для цього графа останні чотири рядки повторюються. Усі вершини тепер пройдені. Результат записаний в останньому рядку таблиці.

Зрозуміло, що приведений вище словесний алгоритм Дейкстри важливий з погляду розуміння його роботи. На практиці застосовують програмні варіанти. Один із таких варіантів задається лістингом 6.2. Але оскільки названа програма працює з числовими символами, то спочатку необхідно перепозначити вершини графа на рис. 6.8, замінивши $A \to 1,\ B \to 2$ та ін. Отже, матимемо граф, представлений на рис. 6.9, повністю ідентичний графу рис. 6.8 (у теорії графів такі графи називають ізоморфними).

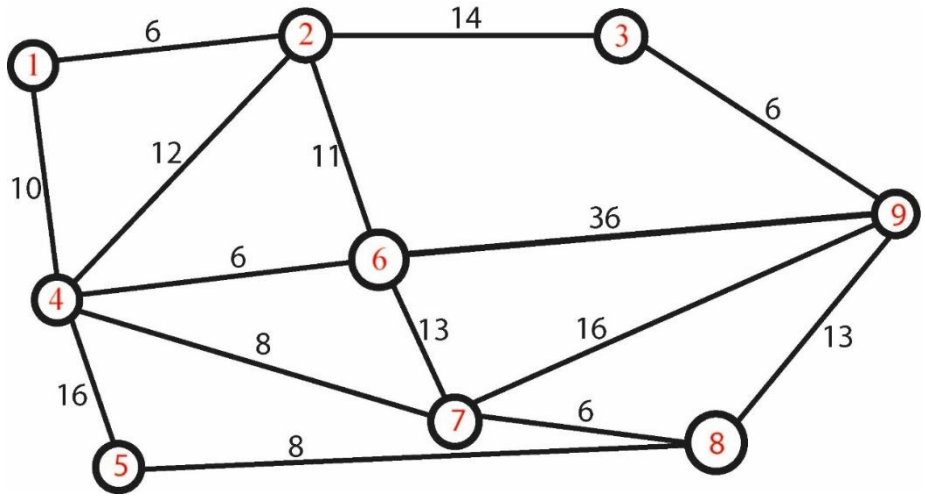


Рис. 6.9. Зважений неорієнтований плоский простий граф, у якому за допомогою алгоритму Дейкстри знаходяться відстані між вершиною 1 та всіма іншими вершинами

Обчислювальне ядро алгоритму перебирає усі можливі варіанти вибору маршруту у графі та відшукує оптимальний. Далі наводимо усю програму пов-

ністю. Після виконання цієї програми достатньо лише ввести список суміжності для ваг ребер заданого графа, і програма обрахує оптимальний маршрут між вибраною вершиною (вершина 1) та всіма іншими вершинами графа.

Лістинг 6.2

```
import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;
    import java.io.PrintWriter;
    import java.util.ArrayList;
    import java.util.Arrays;
    import java.util.StringTokenizer;
public class MiniWay {
   private static int INF = Integer.MAX_VALUE/2;
   double weightU;
   int u;
   private int n; //кількість вершин у графі
   private int m; //кількість дуг у графі
   private ArrayList adj[]; //список суміжності
   private ArrayList weight[]; //вага ребра в орграфі
   private boolean used[]; //масив для зберігання інформації про пройдені
   // та не пройдені вершини
   private double dist[]; //массив для зберігання відстані від стартової
вершини
   private int[] pred;//масив предків, необхідних  для відновлення
   // найкоротшого шляху від стартової вершини
   int start; //стартова вершина, від якої знаходимо відстань до всіх інших
   private BufferedReader cin;
   private PrintWriter cout;
   private StringTokenizer tokenizer;
   private void dejkstra(int s) {//процедура запуску алгоритму Дейкстри зі
      // стартової вершини
      dist[s] = 0; //найкоротша відстань до стартової вершини рівна 0
      for (int iter = 0; iter < n; ++iter) {
         int v = -1;
         double distV = INF;
         for (int i = 0; i < n; ++i) {
            if (used[i]) {
               continue;
            }
```

```
            if (distV < dist[i]) {
                continue;
            }
            v = i;
            distV = dist[i];
        }
        for (int i = 0; i < adj[v].size(); ++i) {
            int u = (int) adj[v].get(i);
            double weightU = (double) weight[v].get(i);
            if (dist[v] + weightU < dist[u]) {
                dist[u] = dist[v] + weightU;
                pred[u] = (int) v;  }}
        used[v] = true;           }}
private void readData() throws IOException {
    cin = new BufferedReader(new InputStreamReader(System.in));
    cout = new PrintWriter(System.out);
    tokenizer = new StringTokenizer(cin.readLine());
    n = Integer.parseInt(tokenizer.nextToken());
    m = Integer.parseInt(tokenizer.nextToken());
    start = Integer.parseInt(tokenizer.nextToken()) - 1;
    adj = new ArrayList[n];
    for (int i = 0; i < n; ++i) {
        adj[i] = new ArrayList();
    }
    //ініціалізація списку, в якому зберігаються ваги ребер
    weight = new ArrayList[n];
    for (int i = 0; i < n; ++i) {
        weight[i] = new ArrayList();
    }
    //зчитуємо граф, заданий списком ребер
    for (int i = 0; i < m; ++i) {
        tokenizer = new StringTokenizer(cin.readLine());
        int u = Integer.parseInt(tokenizer.nextToken());
        int v = Integer.parseInt(tokenizer.nextToken());
        double w = Double.parseDouble(tokenizer.nextToken());
        u--;
        v--;
        adj[u].add(v);
        weight[u].add(w);
    }
```

```
    used = new boolean[n];
    Arrays.fill(used, false);
    pred = new int[n];
    Arrays.fill(pred, -1);
    dist = new double[n];
    Arrays.fill(dist, INF);
}
void printWay(int v) {
    if (v == -1) {
        return;
    }
    printWay(pred[v]);
    cout.print((v + 1) + " ");
}
private void printData() throws IOException {
    for (int v = 0; v < n; ++v) {
        if (dist[v] != INF) {
            cout.print(dist[v] + " ");
        } else {
            cout.print("-1 ");
        }
    }
    cout.println();
    for (int v = 0; v < n; ++v) {
        cout.print((v + 1) + ": ");
        if (dist[v] != INF) {
            printWay(v);
        }
        cout.println();
    }
    cin.close();
    cout.close();
}
private void run() throws IOException {
    readData();
    dejkstra(start);
    printData();
    cin.close();
    cout.close();
}
```

```
    public static void main(String[] args) throws IOException {
        MiniWay solution = new MiniWay();
        solution.run();
    }
}
```

Результат роботи програми:

```
   9   14   1
1    2    6
1    4   10
2    4   12
4    5   16
4    6    6
4    7    8
5    8    8
7    8    6
7    9   16
6    9   36
2    3   14
3    9   6
8    9   13
0.0  6.0  20.0  10.0  26.0  16.0  18.0  24.0  26.0
1: 1
2: 1 2
3: 1 2 3
4: 1 4
5: 1 4 5
6: 1 4 6
7: 1 4 7
8: 1 4 7 8
9: 1 2 3 9
Process finished with exit code 0
```

Проаналізуємо програму Лістингу 6.2 та результат її роботи. Перша тріада чисел задає відповідно кількість вершин графа, кількість ребер графа та номер вершини, від якої розраховуються відстані до усіх інших вершин (у нашому випадку це вершина 1). Тепер з другого рядка у консолі вводимо тріади чисел у послідовності: номер однієї вершини, далі номер інцидентної вершини і третє число – вага відповідного ребра. І так вводимо всі 14 ребер графа (рис. 6.9) та натискаємо клавішу Enter. Після відпрацювання програма видає результат:

1) послідовність із 9 чисел, що і являють собою найкоротші відстані від вершини 1 до всіх інших вершин;

2) дев'ять рядків, що символізують собою оптимальний маршрут.

Скажімо, запис *9: 1 2 3 9* слід розуміти так – маршрут до вершини 9 пролягає вздовж ланцюга $1 \to 2 \to 3 \to 9$. Відповідно довжина цього оптимального ланцюга становить 26,0 одиниць.

6.4. Моделювання мереж: алгоритм Флойда

По суті, це досить простий алгоритм. Він дає змогу знайти спектр найкоротших відстаней *між усіма вершинами графа*. Принцип роботи алгоритму Флойда полягає у визначенні мінімальної відстані шляхом порівняння двох величин, а саме відстані між сусідніми вершинами графа, взятої напряму, та відстані між цими ж вершинами, але пройденої через інші два ребра, тобто обхідним шляхом. Ситуація наочно представлена на рис. 6.10.

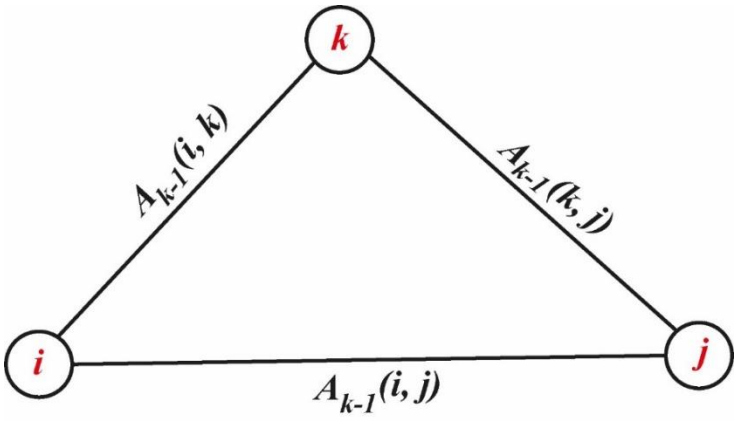


Рис. 6.10. Елементарний фрагмент у алгоритмі пошуку оптимального шляху у графі

Записати аналітично процедуру знаходження мінімальної відстані між сусідніми вершинами графа можна так:

$$A_k(i, j) = \min(A_{k-1}(i, j), A_{k-1}(i, k) + A_{k-1}(k, j)). \qquad (6.1)$$

Графічно ситуація виглядає, як показано на рис. 6.11. Виникає питання: який обрати шлях від i до j – напряму чи застосувати обхідний маневр – пройти через вершину k? Відповідь на поставлене питання дає формула (6.1) – із двох порівнюваних варіантів шляху треба вибрати коротший. Алгоритм перебирає всі можливі варіанти, тобто проходить по всіх вершинах k та знаходить найкоротший шлях між i та j. Далі описана процедура повторюється аж поки не буде знайдений оптимальний маршрут між усіма вершинами графа. Зрозумілою

тепер стає практична значимість графа, адже він дає можливість прокладати оптимальні маршрути, наприклад, між населеними пунктами області.

Нехай для графа, представленого на рис. 6.11, необхідно знайти мінімальні відстані між усіма вершинами. Для цієї мети використаємо алгоритм Флойда. Сформуємо для такого графа вагову матрицю A_{ij} відповідно до визначення (6.1).

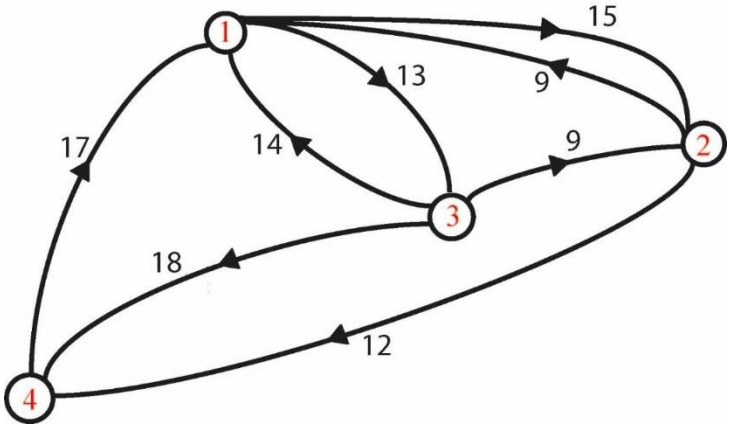


Рис. 6.11. Орієнтований навантажений планарний мультиграф

На нульовому кроці алгоритму це буде просто вагова матриця графа:

$$A_0 = \begin{pmatrix} 0 & 15 & 13 & \infty \\ 9 & 0 & \infty & 12 \\ 14 & 9 & 0 & 18 \\ 17 & \infty & \infty & 0 \end{pmatrix}.$$

На першому кроці ітераційного процесу зі співвідношення (6.1) отримаємо:

$$A_1(i, j) = \min\,(A_0(i, j), A_0(i,1) + A_0(1, j)) \ . \qquad (6.2)$$

Як бачимо, у цьому виразі змінна k приймає значення 1. Відповідна матриця матиме вигляд:

$$A_1 = \begin{pmatrix} 0 & 15 & 13 & \infty \\ 9 & 0 & 22 & 12 \\ 14 & 9 & 0 & 18 \\ 17 & 32 & 30 & 0 \end{pmatrix}.$$

Порівнюючи матрицю A_1 з матрицею A_0, бачимо, що деякі елементи матриці A_1 змінились: внаслідок проходження по всіх елементах матриці з допомогою виразу (6.2) змінюємо ті елементи матриці, які виявляються меншими за

попередні: наприклад, елемент $(A_1)_{2,3}$ став рівним 22, а в матриці A_0 цей елемент був рівним ∞. Аналогічно проходимо другий, третій та четвертий ітераційні кроки, для яких k змінюється відповідно у напрямку $2 \to 4$ (у формулі (6.2) замість 1 виставляємо послідовно 2, 3 і 4). Остаточно матимемо матрицю:

$$A_4 = \begin{pmatrix} 0 & 15 & 13 & 27 \\ 9 & 0 & 22 & 12 \\ 14 & 9 & 0 & 18 \\ 17 & 32 & 30 & 0 \end{pmatrix}.$$

Власне, це і є матриця найкоротших відстаней. Наприклад, від вершини 1 до вершини 4 мінімальний шлях буде рівним 27 – це буде елемент матриці $(A_4)_{1,4}$. До речі, дістатися від вершини 1 до вершини 4 можна двома маршрутами: $1 \to 3 \to 4$ та $1 \to 2 \to 4$. Проте останній маршрут – коротший, тому алгоритм вибирає саме цей маршрут.

Отже, якщо A_0 – матриця ваг графа, то згідно з алгоритмом Флойда пробігаємо по всіх вершинах цього графа і шукаємо коротший шлях через вершину k, реалізуючи всі варіанти, що відповідають співвідношенню (6.1). Фактично для графа з V вершинами пройдені вище кроки зводяться до виконання потрійного циклу *for*:

```
{for (int k = 0; k <V; k++)//пробігаємо по всіх вершинах графа
for (int i = 0; i < V; i++)
for (int j = 0; j < V; j++)
if (dist[i][k] + dist[k][j] < dist[i][j]) {//знаходимо найкоротший шлях
dist[i][j] = dist[i][k] + dist[k][j];
next[i][j] = next[i][k];}.
```

Звичайно, використовувати наведений вище формульно-словесний алгоритм Флойда для об'ємних графів нереально. Наприклад, для графа із 16 ребрами та 6 вершинами, зображеного на рис. 6.12, наведена знизу програма (Лістинг 6.3) дає змогу знаходити відстані за частки секунди (водночас як використання описового алгоритму в цьому випадку вимагатиме набагато більше часу).

Ваги ребер у Лістингу 6.3 задаються в основному коді цієї програми. Зверніть увагу, що ваги ребер у наведеному графі не корелюють зовсім із геометричними відстанями між вершинами ребер. У програмі Лістинг 6.4 двовимірний масив слід розуміти так: перше число – це номер вершини, із якої виходить ребро, друге – це номер вершини, у яку входить ребро, і третє – це вага такого ребра. І так задається будь-який граф. Запустимо програму та проаналізуємо результат її роботи:

int[][] weights = {{1, 2, 7}, {2, 1, 15}, {1, 3, 4}, {3, 1, 9}, {2, 3, 9}, {2, 5, 10}, {5, 2, 11}, {2, 4, 18}, {4, 2, 14}, {5, 4, 12}, {6, 5, 4}, {3, 4, 8}, {4, 3, 9}, {3, 6, 12}, {4, 6, 7}, {6, 4, 4}};

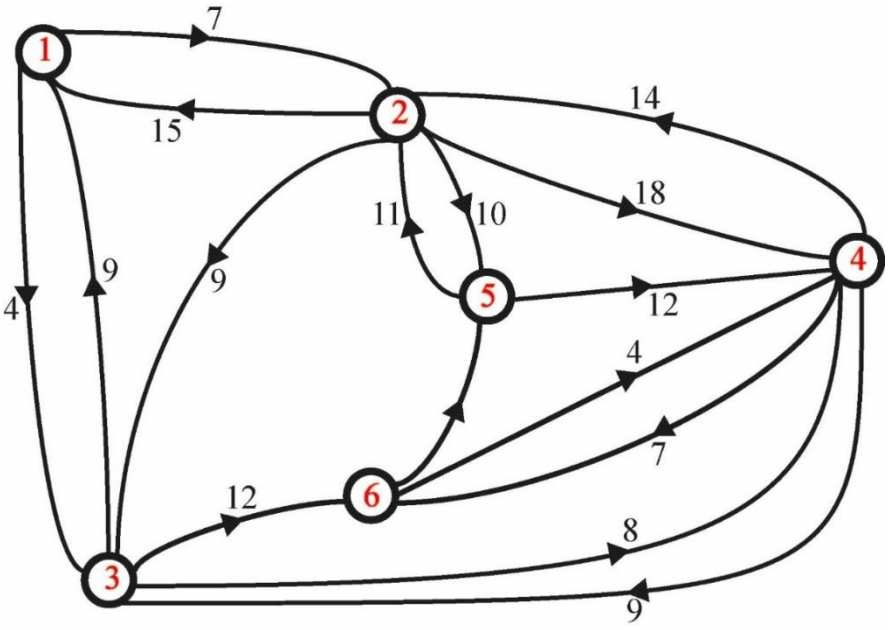


Рис. 6.12. Зважений орієнтований граф

Лістинг 6.3

```
import static java.lang.String.format;
import java.util.Arrays;
public class FloydWarshall {
   public static void main(String[] args) {
      int[][] weights = {{1, 2, 15}, {2, 1, 9}, {1, 3, 13}, {3, 1, 14},
            {2, 4, 12}, {3, 4, 18},{4,1,17}, {3, 2, 9},{1,4,22}, {4,2,36}
            };
      int numVertices = 4;
      floydWarshall(weights, numVertices);
   }
   static void floydWarshall(int[][] weights, int numVertices) {
      double[][] dist = new double[numVertices][numVertices];
      for (double[] row : dist)
         Arrays.fill(row, Double.POSITIVE_INFINITY);
      for (int[] w : weights)
         dist[w[0] - 1][w[1] - 1] = w[2];
      int[][] next = new int[numVertices][numVertices];
      for (int i = 0; i < next.length; i++) {
         for (int j = 0; j < next.length; j++)
```

```
            if (i != j)
               next[i][j] = j + 1;
         }
         for (int k = 0; k < numVertices; k++)
            for (int i = 0; i < numVertices; i++)
               for (int j = 0; j < numVertices; j++)
                  if (dist[i][k] + dist[k][j] < dist[i][j]) {
                     dist[i][j] = dist[i][k] + dist[k][j];
                     next[i][j] = next[i][k];
                  }
         printResult(dist, next);
      }
      static void printResult(double[][] dist, int[][] next) {
         System.out.println("pair     dist    path");
         for (int i = 0; i < next.length; i++) {
            for (int j = 0; j < next.length; j++) {
               if (i != j) {
                  int u = i + 1;
                  int v = j + 1;
                  String path = format("%d -> %d    %2d     %s", u, v,
                        (int) dist[i][j], u);
                  do {
                     u = next[u - 1][v - 1];
                     path += " -> " + u;
                  } while (u != v);
                  System.out.println(path);
               }
            }
         }
      }
   }
```

Результат виконання програми:

```
Pair    dist    path
1 ® 2     7     1® 2
1 ® 3     4     1® 3
1 ® 4    12     1 ® 3 ® 4
1 ® 5    17     1 ® 2 ® 5
1 ® 6    16     1® 3 ® 6
2 ® 1    15     2® 1
```

```
2 ® 3    9    2 ® 3
2 ® 4   17    2 ® 3 ® 4
2 ® 5   10    2 ® 5
2 ® 6   21    2 ® 3® 6
3 ® 1    9    3 ® 1
3 ® 2   16    3 ® 1® 2
3 ® 4    8    3® 4
3 ® 5   16    3® 6® 5
3 ® 6   12    3 ® 6
4 ® 1   18    4 ® 3® 1
4 ® 2   14    4 ® 2
4 ® 3    9    4® 3
4 ® 5   11    4 ® 6 ® 5
4 ® 6    7    4® 6
5 ® 1   26    5 ® 2 ® 1
5 ® 2   11    5 ® 2
5 ® 3   20    5 ® 2 ® 3
5 ® 4   12    5 ® 4
5 ® 6   19    5 ® 4® 6
6 ® 1   22    6 ® 4® 3 ® 1
6 ® 2   15    6 ® 5® 2
6 ® 3   13    6 ® 4® 3
6 ® 4    4    6® 4
6 ® 5    4    6® 5
Process finished with exit code 0
```

Тут представлені найкоротші маршрути між усіма вершинами графа, а також довжини цих маршрутів. Скажімо, найкоротший шлях у напрямку від вершини 6 до вершини 1, тобто маршрут $6 \to 1$, дорівнює 22 та пролягає вздовж простого ланцюга $6 \to 4 \to 3 \to 1$.

6.5. Моделювання мереж: A*-алгоритм

Нехай потрібно знайти оптимальний маршрут у мережі, представленій у вигляді графа на рис. 6.13. Почнемо розгляд поставленої задачі з практичної реалізації алгоритму, що знаходить найкоротший шлях у графі *між двома вибраними вершинами*. Це так званий A*-алгоритм, або його ще називають евристичним. Особливість полягає у введенні поняття евристичної відстані.

Позначимо цю відстань h(x), де x – номер вибраної вершини графа. Це відстань, взята по прямій, від усіх вершин графа до вибраної кінцевої вершини. Далі вводиться функція f(x) = g(x) + h(x), де g(x) – відстань від стартової вершини до поточної вершини x, але пройдена вздовж ребер графа. Завдяки введенню евристики зменшується кількість можливих варіантів вибору шляху і тому алгоритмічна складність A*-алгоритму невелика. На відміну від алгоритмів Дейкстри та Флойда, цей алгоритм має лінійну алгоритмічну складність (алгоритм Дейкстри – квадратичну, а алгоритм Флойда – кубічну).

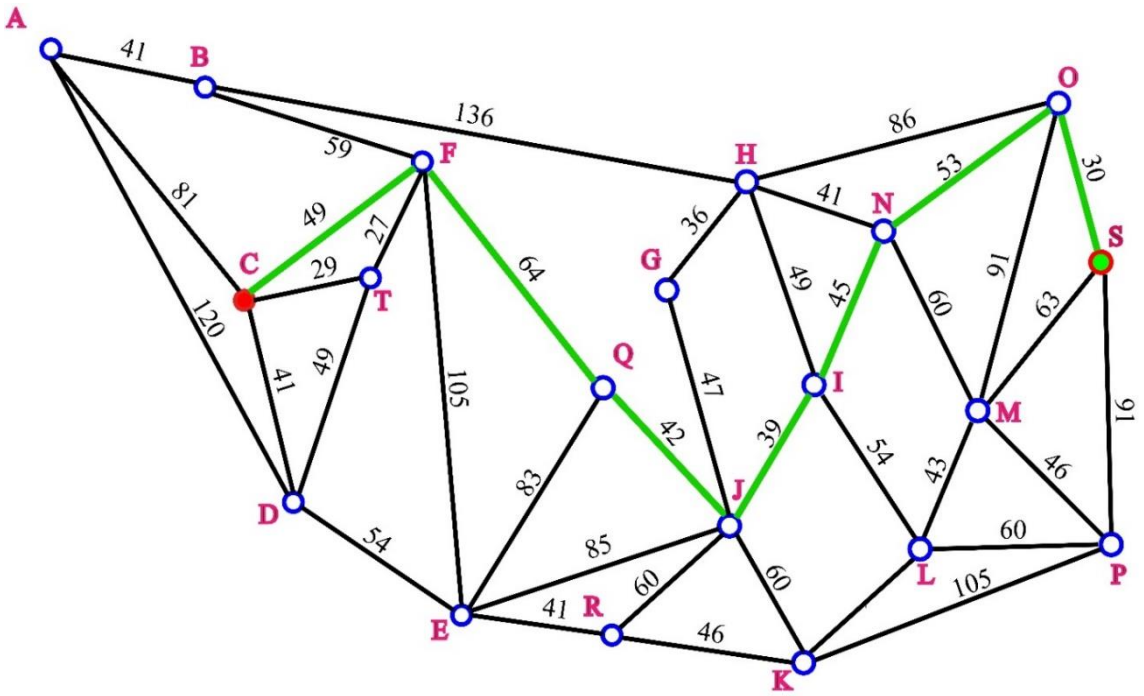


Рис. 6.13. Зважений граф, у якому за допомогою A*-алгоритму знаходиться оптимальний маршрут між вершинами C (стартова вершина) і S (фінішна вершина).
Цей маршрут виділений потовщеною зеленою ламаною лінією.
Біля кожного ребра проставлені ваги, що відповідають геометрії задачі

Представимо тепер приведений описовий варіант, а також програмну реалізацію A*-алгоритму та покажемо шляхи його використання під час прокладання оптимальних маршрутів у різноманітних мережах, зокрема і міських транспортних. Розглянемо граф, представлений на рис. 6.13. Припустимо, треба знайти оптимальний маршрут між вершинами C та S. Скористаємось A*-алгоритмом. Сформуємо таблицю, у яку спочатку занесемо згадані вже евристичні відстані h(x). Фактично це особливі відстані, що вимірюються від поточної вершини x до кінцевої вершини маршруту по прямій. Отже, такі величини є найкоротшими з усіх можливих. Для чого вводиться евристична функція? Така процедура необхідна для знаходження мінімальних величин функції f(x) і на цій основі прокладання оптимального маршруту. Інакше кажучи, вибирається

найкоротший ланцюг, що пролягає від стартової вершини через поточну вершину *x* до кінцевої. Для графа, представленого на рис. 6.13, оптимальний маршрут між вершинами C та S та його довжина прораховані та представлені у табл. 6.2.

Таблиця 6.2. Прокладання оптимального маршруту з допомогою A*-алгоритму

Вершина	**Відстань від C, g(*x*)**	**Евристична відстань, h(*x*)**	**f(*x*)=g(*x*)+h(*x*)**	**Попередня вершина**
A	81,161	262	343,423	C, D
B		221	329	F
C	0	207	207	
D	41,78	208	249,285	C, T
E	154,196,96,240	201	355,397,297,441	F, Q, D, J
F	49,56	165	214,221	C, T
G	202	109	311	J,
H	243,280,238,378	86	329,366,324,464	I, N, G, O
I	194	79	273	J
J	155	117	272	Q
K	215,183	157	372,340	J, R
L	248	105	353	I
M	299,383	63	362,446	N, O
N	239	45	284	I
O	292	30	322	N
P		91		
Q	120	128	248	F
R	215, 137	176	391,313	J, E
S	322	0	322	O
T	29	178	207	C

Технічно процес прокладання найкращого шляху у павутинні графа виглядає так. На першому кроці визначаємо вершини, інцидентні до вершини C. Це будуть вершини A, F, T та D. Записуємо відповідні дані g(*x*) у табл. 6.2 у стовпчик для значень функції g(*x*). Потім додаємо відповідні величини g(*x*) та h(*x*) і результат записуємо у стовпчик для функції f(x). Порівнюємо між собою чотири отримані значення та вибираємо найменше число, рівне 214. Значить, наш маршрут має пролягати до вершини F. У вершини A, F, T та D перехід відбувся із вершини C, тому у стовпчику «Попередня вершина» виставляємо C. Продовжуючи процедуру вибору оптимального маршруту, далі аналогічним способом заповнюємо табл. 6.2. Аналізуючи отримані результати, бачимо, що протяжність оптимального маршруту C → S складає величину 322 умовні одиниці. З допомогою табл. 6.2 легко можна відтворити оптимальний маршрут у графі (рис. 6.13). Для цього зауважимо, що у вершину S перехід відбувся із

вершини O. У вершину O перехід був здійснений із N. Продовжуючи подальший аналіз, відтворюємо сам оптимальний маршрут C → F → Q → J → I → N → O → S.

Звернемо увагу, що у табл. 6.2 у стовпчику «Вершина» пройдені вершини зсовуються вправо. Це робиться для того, щоб під час оцінки функції f(*x*) розглядати спектр величин f(*x*) лише для непройдених вершин.

Представимо тепер програмний варіант сформульованого вище описового алгоритму, використовуючи програмний Java-код (Лістинг 6.4).

Лістинг 6.4

```
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
import java.util.List;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.Collections;
public class AstarSearchAlgo {
 // евристична відстань h являє собою відстані по прямій між
 // всіма вершинами графа та фінішною вершиною S
 public static void main(String[] args) {
    //створення об'єктів – вузлів графа
    Node A = new Node("A", 262);
    Node B = new Node("B", 221);
    Node C = new Node("C", 207);
    Node D = new Node("D", 207);
    Node E = new Node("E", 201);
    Node F = new Node("F", 165);
    Node G = new Node("G", 109);
    Node H = new Node("H", 86);
    Node I = new Node("I", 79);
    Node J = new Node("J", 117);
    Node K = new Node("K", 157);
    Node L = new Node("L", 105);
    Node M = new Node("M", 63);
    Node N = new Node("N", 45);
    Node O = new Node("O", 30);
    Node P = new Node("P", 91);
    Node Q = new Node("Q", 128);
    Node R = new Node("R", 176);
```

```
Node S = new Node("S", 0);
Node T = new Node("T", 178);
//створення об'єктів – ребер графа
A.adjacencies = new Edge[]{
        new Edge(B, 41),
        new Edge(C, 81),
        new Edge(D, 120),};
B.adjacencies = new Edge[]{
        new Edge(A, 41),
        new Edge(F, 59),
        new Edge(H, 136) };
C.adjacencies = new Edge[]{
        new Edge(A, 81),
        new Edge(F, 49),
        new Edge(T, 29),
        new Edge(D, 41),  };
D.adjacencies = new Edge[]{
        new Edge(A, 120),
        new Edge(C, 41),
        new Edge(T, 49),
        new Edge(E, 54),  };
E.adjacencies = new Edge[]{
        new Edge(D, 54),
        new Edge(F, 105),
        new Edge(Q, 83),
        new Edge(J, 85),
        new Edge(R, 41), };
F.adjacencies = new Edge[]{
        new Edge(B, 59),
        new Edge(Q, 64),
        new Edge(E, 105),
        new Edge(T, 27),
        new Edge(C, 49),  };
G.adjacencies = new Edge[]{
        new Edge(H, 36),
        new Edge(J, 47),  };
H.adjacencies = new Edge[]{
        new Edge(B, 136),
        new Edge(O, 86),
        new Edge(N, 41),
```

```
        new Edge(I, 49),
        new Edge(G, 36) };
I.adjacencies = new Edge[]{
        new Edge(H, 49),
        new Edge(N, 45),
        new Edge(L, 54),
        new Edge(J, 39),   };
J.adjacencies = new Edge[]{
        new Edge(Q, 42),
        new Edge(G, 47),
        new Edge(I, 39),
        new Edge(K, 60),
        new Edge(R, 60),
        new Edge(E, 85),  };
K.adjacencies = new Edge[]{
        new Edge(R, 46),
        new Edge(J, 60),
        new Edge(L, 53),
        new Edge(P, 105),  };
L.adjacencies = new Edge[]{
        new Edge(K, 53),
        new Edge(I, 54),
        new Edge(M, 43),
        new Edge(P, 60),   };
M.adjacencies = new Edge[]{
        new Edge(N, 60),
        new Edge(O, 91),
        new Edge(S, 63),
        new Edge(P, 46),
        new Edge(L, 43),   };
N.adjacencies = new Edge[]{
        new Edge(H, 41),
        new Edge(O, 53),
        new Edge(M, 60),
        new Edge(I, 45),   };
O.adjacencies = new Edge[]{
        new Edge(H, 86),
        new Edge(S, 30),
        new Edge(M, 91),
        new Edge(N, 53), };
```

```
        P.adjacencies = new Edge[]{
                new Edge(K, 105),
                new Edge(L, 60),
                new Edge(M, 46),
                new Edge(S, 91),  };
        Q.adjacencies = new Edge[]{
                new Edge(F, 64),
                new Edge(J, 42),
                new Edge(E, 83), };
        R.adjacencies = new Edge[]{
                new Edge(E, 41),
                new Edge(J, 60),
                new Edge(K, 46),  };
        S.adjacencies = new Edge[]{
                new Edge(O, 30),
                new Edge(P, 91),
                new Edge(M, 63),  };
        T.adjacencies = new Edge[]{
                new Edge(C, 29),
                new Edge(F, 27),
                new Edge(D, 49),  };
        AstarSearch(C, S);
        List<Node> path = printPath(S);
        System.out.println("Path: " + path);}
    public static List<Node> printPath(Node target) {
        List<Node> path = new ArrayList<Node>();
        for (Node node = target; node != null; node = node.parent) {
            path.add(node);}
        Collections.reverse(path);
        return path;}
    public static void AstarSearch(Node source, Node goal) {
        Set<Node> explored = new HashSet<Node>();
        PriorityQueue<Node> queue = new PriorityQueue<Node>(30,new
Comparator<Node>() {
            //реалізація методу порівняння
            public int compare(Node i, Node j) {
                if (i.f_scores > j.f_scores) {
                    return 1;
                } else if (i.f_scores < j.f_scores) {
                    return -1;
```

```
            } else {
                return 0;
            }}});
        //вага на старті
        source.g_scores = 0;
        queue.add(source);
        boolean found = false;
        while ((!queue.isEmpty()) && (!found)) {
            //цей вузол має найнижчу величину f_score
            Node current = queue.poll();
            explored.add(current);
            //мета знайдена
            if (current.value.equals(goal.value)) {
                found = true;}
            //перевірка кожного інцидентного ребра поточного вузла
            for (Edge e : current.adjacencies) {
                Node child = e.target;
                double cost = e.cost;
                double temp_g_scores = current.g_scores + cost;
                double temp_f_scores = temp_g_scores + child.h_scores;
                //якщо інцидентний вузол оцінений і
               // оновлена f_score є більшою, то здійснюємо перехід
                if ((explored.contains(child)) &&
                        (temp_f_scores >= child.f_scores)) {
                    continue;}
                else if ((!queue.contains(child)) ||
                        (temp_f_scores < child.f_scores)) {
                    child.parent = current;
                    child.g_scores = temp_g_scores;
                    child.f_scores = temp_f_scores;
                    if (queue.contains(child)) {
                        queue.remove(child);}
                    queue.add(child);
                }}}}}
class Node {
    public final String value;
    public double g_scores;
    public final double h_scores;
    public double f_scores = 0;
    public Edge[] adjacencies;
```

```
    public Node parent;
    public Node(String val, double hVal) {
        value = val;
        h_scores = hVal;}
    public String toString() {
        return value;
    }}
class Edge {
    public final double cost;
    public final Node target;
    public Edge(Node targetNode, double costVal) {
        target = targetNode;
        cost = costVal;
    }}
```

Результат роботи програми виглядає так: Path: [A, C, H, I, J, P]. Process finished with exit code 0.

Як бачимо, отриманий шлях співпадає із знайденим раніше за допомогою «ручного» алгоритму. Важливість A*-алгоритму полягає у тому, що відсікаються неперспективні варіанти маршрутів завдяки аналізу значень функції f(x). Зі свого боку, величини f(x) залежать від значень функції h(x).

6.6. Моделювання міського перехрестя

Моделювання міського перехрестя викладене за посиланням https://r.donnu.edu.ua/bitstream/123456789/1074/1/%D0%9D%D0%86%D0%9A%D0%9E%D0%9B%D0%AE%D0%9A%20%D0%9F.%D0%9A.%20%D0%86%D0%9D%D0%A2%D0%95%D0%9B%D0%95%D0%9A%D0%A2%20%D0%9F%D0%95%D0%A0%D0%95%D0%A5%D0%A0%D0%95%D0%A1%D0%A2%D0%AF.pdf

Слідуючи наведеному дослідженню, проведемо детальний аналіз особливостей такого міського транспортного об’єкта. Загалом моделювання міського трафіку, зокрема і моделювання міського перехрестя, є надзвичайно актуальною проблемою сьогодення. Стартовою задачею у вирішенні поставленої проблеми є оптимізація (максимізація) пропускної здатності згаданого об’єкта міської транспортної мережі. Отже, базовим елементом, а заодно і базовою проблемою регулювання трафіку у мегаполісі є окреме перехрестя. Дійсно, насамперед така проблема пов’язана із заторами на перехрестях і, як наслідок, складністю проїзду кожним транспортним засобом (ТЗ) за вибраним маршрутом. З метою запобігання виникненню заторів, які в основному виникають на перехрестях,

пропонується застосувати мережу інтелектуальних перехресть, споряджених спеціальними датчиками. Припускається, що датчики сусідніх перехресть взаємодіють між собою. Така взаємодія фактично означає, що не тільки окремі перехрестя знаходяться під контролем центру керування трафіком (ЦКТ), а під таким контролем знаходяться і ділянки дороги між сусідніми перехрестями. Отже, з'являється можливість моніторити всю транспортну мережу міста, а це означає спроможність контролювати маршрути усіх ТЗ на дорогах міста. Враховуючи усе вищесказане, головна проблема, що виникає у цьому випадку, полягає у поєднанні алгоритмів регулювання руху ТЗ як через окреме перехрестя *(перша стадія)*, так і по всьому місту *(друга стадія)*.

Отже, організувавши ефективний рух через усі перехрестя міста, досягнемо високої ефективності трафіку вже на *першій стадії*. Для керування трафіком ТЗ необхідно їх реєструвати. Причому способів реєстрації є багато. Для розв'язання нашої проблеми найбільш ефективним засобом реєстрації є використання п'єзокристалічних датчиків, вмонтованих у полотно дороги.

Компанія International Road Dynamics Inc. розробила досить ефективну конструкцію п'єзоелектричного датчика дорожнього руху RoadTrax BL9. Такий пристрій легко монтується в дорожнє полотно і досить чутливий. Під час наїзду колісної пари на зону розташування датчика вихідного сигналу величиною 250 мВ цілком достатньо для реєстрації ТЗ. Узагалі датчики дорожнього руху за своїм типом поділяються на дві групи (рис. 6.14).

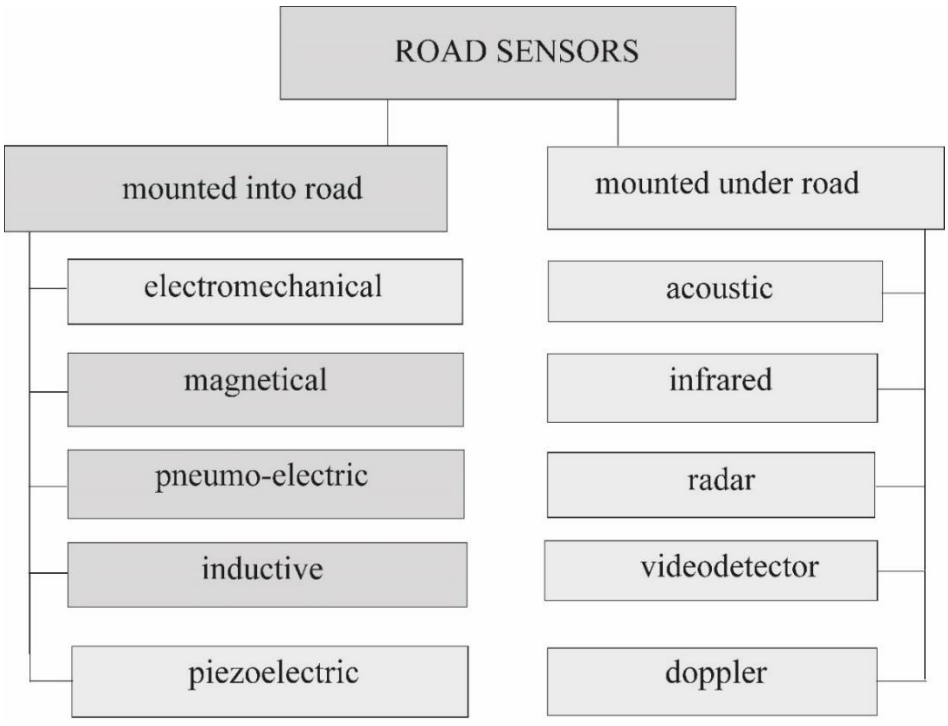


Рис. 6.14. Класифікація датчиків дорожнього руху

До першої групи належать датчики, що монтуються в полотно дороги. До цих датчиків належать: 1) електромеханічні; 2) магнітні; 3) пневмоелектричні; 4) індуктивні та 5) п'єзоелектричні. До другої групи належать датчики, що монтуються над дорогою. До цієї групи належать такі датчики: 1) акустичні; 2) інфрачервоні; 3) радарні; 4) відеодетекторні; 5) допплерівські.

Найпридатніші для розв'язання поставленої у нашій роботі задачі є датчики, що монтуються в дорожнє полотно, а серед цієї групи – п'єзокристалічні та індуктивні.

Монтування індуктивних датчиків у полотно дороги є більш проблематичним. Ось чому доцільно зупинитись на простих, але досить ефективних п'єзоелектричних датчиках. Серед цього класу датчиків є сенс згадати про п'єзоелектричний датчик типу 12-DOF. Це новий високочутливий пристрій, спроможний реєструвати сили у широкому діапазоні величин. Апарат працює на частоті 11 кГц з похибкою не більше 1 %.

Теоретичні аспекти оптимізації трафіку на окремому перевантаженому перехресті розглядаються у так званій дискретно-часовій моделі (discrete-time model). Для хрестоподібного перехрестя математична модель, що описує рух ТЗ через таке перехрестя, представляється у вигляді системи рівнянь такого виду:

$$J_D = \sum_{k=0}^{N} [q_1(k) + q_2(k)] + \frac{a_1 + a_2}{2} \cdot T \cdot \sum_{k=0}^{N-1} u(k) \to \min, \tag{6.3}$$

$$q_1(k+1) \geq \max\{q_1(k) + d_1 \cdot T \cdot (u(k) - u_L), a_1 \cdot T \cdot (1 - u(k))\}, \tag{6.4}$$

$$q_2(k+1) \geq \max\{q_2(k) + d_2 \cdot T \cdot (u_H - u(k)), 0)\}, \tag{6.5}$$

$$u_{\min} \leq u(k) \leq u_{\max}, \tag{6.6}$$

$$q_i(0) = q_{i,\text{int}},\ i = 1,2;\ N = 0,1,2 \ldots \text{N} - 1. \tag{6.7}$$

У наведених рівняннях фігурують такі змінні: J_D – модельна величина, що являє собою число ТЗ на перевантаженому перехресті, які не перетнули його; N – число циклів перемикання світлофора; k – нумератор циклів; $q_1(k), q_2(k)$ – число ТЗ на перехресті у k -му циклі переключення світлофора відповідно на горизонтальному та вертикальному напрямках хрестоподібного перехрестя; a_1, a_2 – кількість ТЗ, що прибули за одиницю часу відповідно на горизонтальний та вертикальний напрямок; T – цикл роботи світлофора; $u(k)$ – відношення величини фази горіння зеленого світла до аналогічної

величини для червоної фази; d_1, d_2 – кількість ТЗ, що перетнули перехрестя за одиницю часу; $u_H = 1 - a_2 / d_2$, $u_L = a_1 / d_1$; $u_{\min}, u_{\max}$ – відповідно мінімальне та максимальне значення $u(k)$; $q_{i,\text{int}}$ – початкова величина числа ТЗ на перехресті для горизонтального ($i = 1$) та вертикального напрямків ($i = 2$).

Цільова функція системи рівнянь (6.4)–(6.7) представляється виразом (6.3) і являє собою число ТЗ, що не перетнули перехрестя. Зрозуміло, що таку величину треба мінімізувати. Шуканими змінними є величини $u(k)$. Рівняння (6.4) і (6.5) представляють еволюцію з часом обох автомобільних черг. Рівняння (6.6) дає верхню і нижню границі $u(k)$ у кожному циклі. Рівняння (6.6) представляє стартові (початкові) довжини автомобільних черг на перехресті. Загалом система рівнянь (6.3)–(6.7) являє собою задачу математичного програмування. Графічний розв'язок цієї задачі представлений на рис. 6.15.

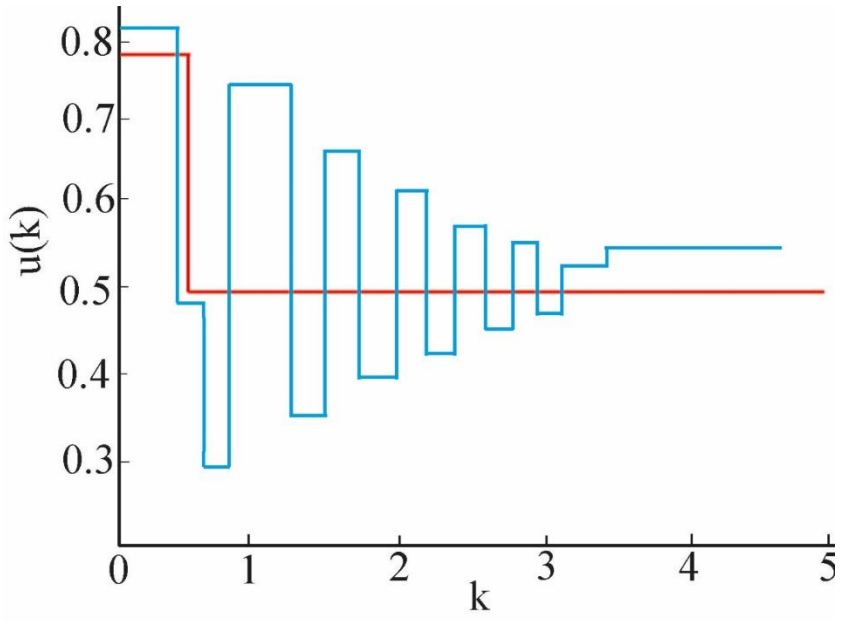


Рис. 6.15. Розв'язок системи рівнянь (6.3)–(6.7):
зигзагоподібна лінія – точний розв'язок; ступінчаста – наближений

Отриманий результат має, імовірніше, академічний, а не практичний характер, і не дає змоги використати отримані результати для практичної мети – інтелектуальної регуляції трафіку через перехрестя з метою покращення його пропускної здатності.

Широкомасштабне дослідження методів об'єднаної інформації, отримуваної із різноманітних дорожніх датчиків, як-от детектори, відеокамери і радари, досліджувалось у багатьох роботах. Особливу увагу привертає технологія зв'язаних ТЗ (connected vehicles), яка дає можливість збирати й аналізувати зв'язки типу «ТЗ – Інфраструктура» (V2I) та «ТЗ – ТЗ» (V2V), які дають можливість

зменшити ймовірність заторів, збільшити безпеку руху та зменшити витрати палива.

На рис. 6.16 зображена схема взаємодії автомобілів, що проїжджають перехрестя, із придорожньою інфраструктурою. Система передачі даних OBU забезпечує V2X комунікацію, тобто взаємодію автомобілів із придорожніми комірками – Road Side Unit (RSU). Організація наведеної на рис. 6.16 взаємодії між автомобілями та придорожньою інфраструктурою недостатньо ефективна та вимагає складної технічної організації і значних фінансових витрат. Через це пропонується простіша, але ефективніша система організації дорожнього руху. Така модель базується на:

1) алгоритмі та комп'ютерній програмі, що сумісно забезпечують ефективне перемикання світлофорних фаз відповідно до завантаженості дорожніх напрямків;

2) реєстрації ТЗ на кожному перехресті за допомогою датчиків, які фіксують кількість колісних пар, пропорційну кількості автомобілів, що або в'їжджають на проїзну частину дороги між сусідніми перехрестями (так звані вхідні датчики), або виїжджають із цієї частини дороги – вихідні датчики.

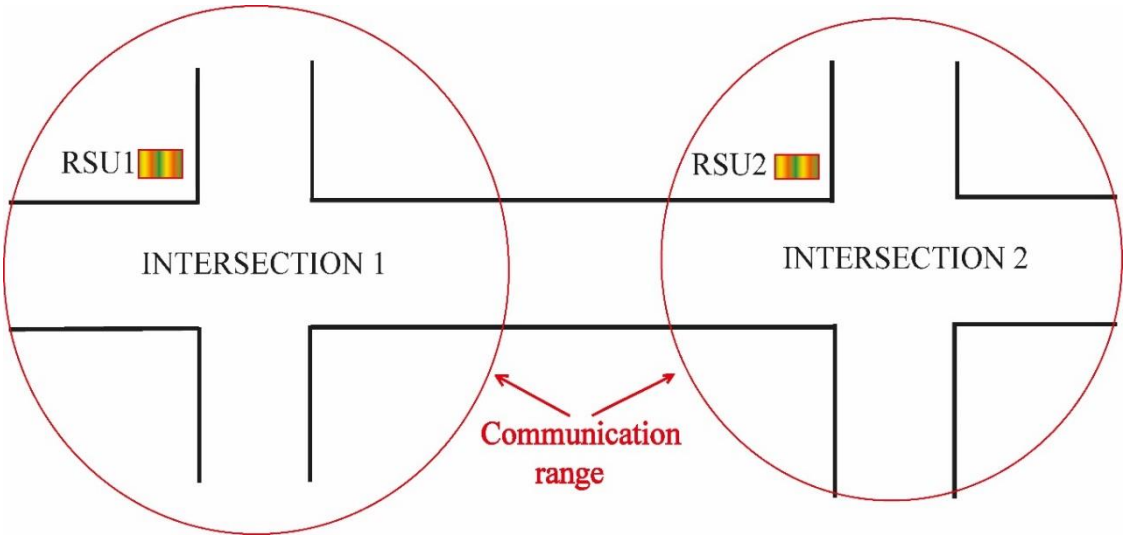


Рис. 6.16. Технологія зв'язаних автомобілів, де RSU – Road Side Unit, тобто придорожня ячейка, яка взаємодіє і зі світлофором, і з автомобілем. Діапазон взаємодії RSU зображений у вигляді колової траєкторії. На автомобілях встановлена система передачі даних OBU (On Board Unit)

Реєстрація ТЗ, що перетинають перехрестя, організована, як показано на рис. 6.17. Кожне окреме перехрестя обладнане п'єзокристалічними датчиками типу RoadTrax BL9. Ці датчики монтуються перпендикулярно до поздовжньої осі дороги на відстані 10 м після стоп-лінії ближче до центру перехрестя. Це дає змогу реєструвати автомобілі, що залишили ділянку дороги одного напрямку руху між сусідніми перехрестями.

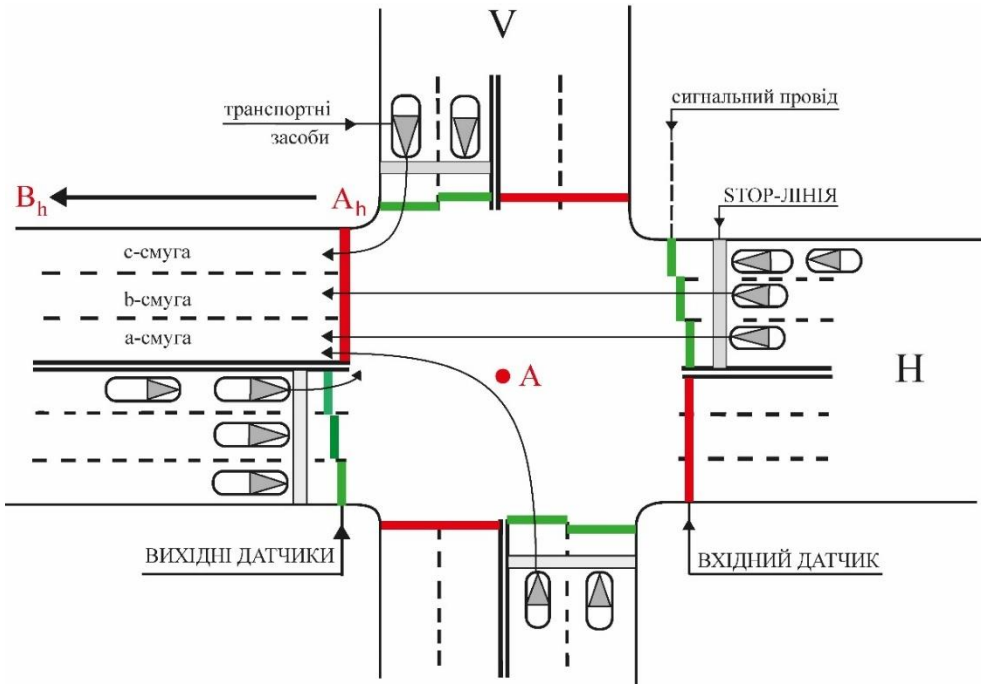


Рис. 6.17. Зображено окреме перехрестя А. Округленими прямокутниками зображені ТЗ, маршрути яких пролягають у напрямку $A_h \to B_h$. Показані також вхідні датчики (прямокутна смужка) та вихідні датчики (три поруч розташовані смужки). Вхідні та вихідні датчики з'єднані з дорожнім контролером (на рисунку зображений лише один сигнальний провід, що виходить від одного із датчиків – це пунктирна лінія). Символ V символізує вертикальний напрямок, H – горизонтальний

Колісна пара автомобіля створює тиск на полотно дороги, у якому на глибині близько 5 см вмонтована п'єзокристалічна смужка. Стиск полотна приводить до стиску смужки, на поверхні якої з'являються електричні заряди завдяки явищу п'єзоефекту. Інакше кажучи, виникають електричні імпульси, які реєструються. Під час проїзду ТЗ через зону розташування датчика (рис. 6.17) реєструються колісні пари, що є більш точним показником завантаженості транспортної магістралі, ніж просто кількість ТЗ. Адже, наприклад, ТЗ великої протяжності (long vehicle) можуть мати 5–7 колісних пар. Сама кількість колісних пар є точнішим критерієм завантаженості трафіку, порівняно з кількістю автомобілів (або ТЗ).

Вважатимемо, що цикл роботи світлофорів на перехресті – це величина, що зазвичай містить такі складники:

$$T = trh + tgh + tyh + tp, \tag{6.8}$$

де *trh* – час горіння червоного світла у горизонтальному напрямку; *tgh* – час горіння зеленого світла у горизонтальному напрямку; *tyh* – час горіння жовтого світла також у горизонтальному напрямку; *tp* – час горіння зеленого світла для пішоходів.

У співвідношенні (6.8) цикл роботи світлофора формується із часових інтервалів, взятих щодо горизонтального напрямку. Таке саме співвідношення можна записати стосовно вертикального напрямку перехрестя, тобто:

$$T = trv + tgv + tyv + tp\,, \tag{6.9}$$

де trv – протяжність червоної фази у вертикальному напрямку; tgv – час горіння зеленого світла у вертикальному напрямку; tyv – час горіння жовтого світла так само у вертикальному напрямку; tp – час горіння зеленого світла для пішоходів.

Із використанням наведених змінних програму регуляції трафіку на окремому перехресті можна записати у такому вигляді:

Лістинг 6.5

```
import java.util.Random;
import static java.lang.StrictMath.abs;
interface Lights{
   int REDH =0;
   int YELLOWH =1;
   int GREENH =2;
   int REDV =3;
   int YELLOWV =4;
   int GREENV =5;
   int GREENP = 6;
   int ERROR = -1;
}
class T implements Lights {
   private int delay;
   private static int light = REDH;
   T(int sec) {
      delay = 1000 * sec;
   }
   public int shift() {
      int count = (light++) % 7;
      try {
         switch (count) {
            case REDH:
               Thread.sleep(delay);
               break;
```

```
            case YELLOWH:
               Thread.sleep(delay / 3);
               break;
            case GREENH:
               Thread.sleep(delay / 2);
               break;
            case REDV:
               Thread.sleep(delay);
               break;
            case YELLOWV:
               Thread.sleep(delay / 3);
               break;
            case GREENV:
               Thread.sleep(delay / 2);
               break;
            case GREENP:
               Thread.sleep(delay );
               break;
         }
      } catch (Exception e) {
         return ERROR;
      }
      return count;
   } }
class TrafficRegulator {
   static int T = 96;
   private static T t = new T(1);
   static Random gn1 = new Random();
   static Random gn2 = new Random();
   static Random gn3 = new Random();
   static Random gn4 = new Random();
   TrafficRegulator() {
   }
   public static void main(String[] args) {
      double k = Math.abs ((gn1.nextDouble()  + gn2.nextDouble() ) /
            (gn3.nextDouble() + gn4.nextDouble()));
      double tg = 35.0;
      double tp = 23.0;
      double tyh = 2.0;
```

```
double tgp = 15.0;
double tgh = k * tg;
double trh = (double)(T - tyh - tgh - tp);
double tgv = abs(2.0 * tg - tgh);
double trv = (double)T - trh;
int tyv = 2;
for(int j = 0; j < 7; ++j) {
   switch(t.shift()) {
      case -1:
         System.out.println("Time error!");
         break;
      case 0:
         System.out.println("red horizontal!");
         System.out.format("%.1f%n", trh);
         break;
      case 1:
         System.out.println("yellow horizontal!");
         System.out.println(tyh);
         break;
      case 2:
         System.out.println("green horizontal!");
         System.out.format("%.1f%n", tgh);
         break;
      case 3:
         System.out.println("red vertical!");
         System.out.format("%.1f%n", trv);
         break;
      case 4:
         System.out.println("yellow vertical!");
         System.out.println(tyv);
         break;
      case 5:
         System.out.println("green vertical!");
         System.out.format("%.1f%n", tgv);
         break;
      case Lights.GREENP:
         System.out.println("green pedestrian!");
         System.out.println(tgp);
         break;
```

```
            default:
                System.err.println("Unknown light.");
                return;
            }
        }
    }
}
```

Внаслідок роботи програми під час першого запуску отримуємо такі результати: *red horizontal! – 29,6; yellow horizontal! – 2.0; green horizontal! – 41,4; red vertical! – 66,4; yellow vertical!– 2; green vertical! – 28,6; green pedestrian! – 23.0. Process finished with exit code 0.*

Наступний запуск програми видасть нові результати: *red horizontal! – 49,6; yellow horizontal! – 2.0; green horizontal! – 21,4; red vertical!– 46,4; yellow vertical! – 2; green vertical! – 48,6; green pedestrian! – 23.0. Process finished with exit code 0.*

Отже, програма щоразу видаватиме нові результати залежно від даних, що надходять від генераторів випадкових величин. Саме з метою моделювання реальної ситуації на перехресті в програмі ввімкнено чотири генератори випадкових величин **static** Random *gn* = **new** Random(), що імітують завантаженість перехрестя автомобілями. Кожен із цих генераторів задає число, що відповідає рівню завантаженості автомобілями певного напрямку на перехресті. Власне, нас цікавить відношення кількості автомобілів, що розташовані на горизонтальному H та вертикальному V напрямках (рис. 6.17). У програмі таке відношення задається величиною коефіцієнта k. Варіація величини k якраз імітує зміни завантаженості напрямків руху ТЗ. Тому спектр чисел на виході програми щоразу змінюється випадково. Змінюються, зокрема, величини, що задають час горіння зеленого світла у горизонтальному та вертикальному напрямках. У якості констант вибираються величини типу REDH – червоне світло в горизонтальному напрямку. Принципово те, що програма реагує на завантаженість ТЗ напрямків перехрестя: чим більш завантажений напрямок – горизонтальний чи вертикальний – тим довше горітиме зелене світло у відповідному напрямку у межах періоду перемикання світлофора, який у цьому випадку вибраний рівним 96 с.

Отже, презентована комп'ютерна модель дає можливість суттєво покращити пропускну здатність перехрестя завдяки «розумному» режиму його роботи у плані кореляції між протяжністю горіння різних фаз та завантаженістю напрямків на перехресті.

6.7. Моделювання міського трафіку

Важливим завданням програмування є створення прикладних програм, які розв'язують конкретні проблеми. Багато проблем виникає під час оптимізації різних видів трафіку. Що таке трафік? У перекладі з англійської мови *traffic* означає «потік транспорту». У широкому розумінні трафік може означати:

1) інтернет-трафік – об'єм мегабайт, що передаються на смартфон, планшет чи комп'ютер. Пакети такого виду трафіка включаються у тарифи сотових операторів;

2) мобільний трафік – протяжність у хвилинах сотового зв'язку, що споживається абонентом;

3) трафік відвідувачів сайта – число осіб, що зайшли на сайт;

4) реферальний трафік – це кількість переходів на певний сайт за посиланнями, що знаходяться на інших інтернет-ресурсах.

Часто кажуть: вебтрафік, мультимедіа-трафік, трафік вантажних (пасажирських) перевезень тощо. Отже, в широкому розумінні трафік – це інтенсивність руху, транспортування, потік даних через комунікаційну систему чи мережу. Можна, наприклад, сказати «нічний трафік електроенергії», маючи водночас на увазі передачу електроенергії у нічний час, коли вартість її споживання зменшується вдвічі стосовно денного часу.

З огляду на це розглядатимемо трафік як потік транспортних засобів (ТЗ) по міських магістралях. Річ у тім, що сьогодні міські транспортні мережі неспроможні здійснювати ефективний трафік ТЗ. Наслідком цієї неспроможності є транспортні затори у великих і не дуже великих містах. Проблема заторів далеко не нова. Ось що писав у своєму романі «Здобич» знаменитий французький письменник Еміль Золя: «По дорозі додому, серед скупчення екіпажів, які поверталися берегом озера, їхати можна було тільки ступою. Нарешті коляска потрапила у такий затор, що довелося навіть зупинитися». Як бачимо, затори були і тоді, коли ще не було автомобілів. З тих пір проблема набула значно гострішого характеру.

Оптимізація трафіку у мегаполісах є актуальною проблемою всесвітнього характеру. А найголовнішою проблемою є затори. Як уникнути цього негативного явища? Як організувати проїзд кожного окремого ТЗ (а таких об'єктів у великому місті може бути понад мільйон) до заявленого водієм пункту призначення з розрахунком, щоб поїздка минула якнайшвидше? Технологія, що пропонується, дає змогу ефективно розв'язувати поставлені задачі.

Взагалі згадані проблеми стоять на порядку денному для багатьох великих міжнародних компаній, що розробляють технічні засоби організації дорожнього руху. Найбільш близькою системою до розглядуваної нами технології є система

навігації типу GPS. Кожного року GPS-навігація модифікується. Деяке автомобільне навігаційне обладнання може повідомляти про затори на вулицях міста та пропонувати альтернативний маршрут об'їзду таких місць. Інтеграція із автомобілем стає глибшою, і це дає змогу задавати кожному конкретному водієві голосові маршрут-команди. Особливе значення має взаємодія між автомобілем та дорожньою інфраструктурою. Ця взаємодія здійснюється за допомогою так званих точок доступу (access points), розташованих вздовж автомобільної дороги та на перехрестях. Взагалі в основі технології регулювання трафіку лежать чотири складники: світлофори, детектори черги (queue detectors), дорожні відеокамери і центральна контрольна система (central control system).

Пропонована технологія спрямована на реєстрацію ТЗ, що проїжджають перехрестя. Водночас кожні 10 секунд система знімає дані зі спеціальних дорожніх контрольних датчиків та оновлює базу даних, яка використовується для прокладання оптимальних маршрутів, а також для коригування протяжності фаз горіння світлофорів відповідно до завантаженості напрямків на перехресті. Завдяки цьому досягається оптимізація трафіку.

Важливою проблемою є розташування датчиків, оскільки їх локалізація суттєво впливає на те, які транспортні потоки реєструються і тому можуть бути керованими. Сьогодні застосовується модифікована модель стільникових автоматів (modified cellular automata model) для вивчення процесів взаємодії між автомобілями. Здійснюється широкомасштабне моделювання потоків ТЗ. Часто застосовується «розумна» мережева імітаційна модель, що використовує у якості об'єкта дослідження центральні райони мегаполісів – їх найбільш завантажені транспортні артерії. У таких випадках на карті міст зображуються автомагістралі, звичайні дороги, автобусні зупинки, комерційні зони, перехрестя та автомобільні розв'язки. Для проведення імітаційних досліджень часто застосовується імітаційна модель PARAMICS. Із метою реєстрації потоків ТЗ в деяких випадках використовуються петлеві детектори (loop detectors) та камери спостереження (surveillance cameras). Можливим варіантом покращення трафіку є координоване регулювання автомобільних потоків за допомогою бездротового зв'язку між автомобілями. З розвитком автотранспорту все більшого поширення набуває теорія стільникових автоматів. Ефективним засобом моделювання руху через перехрестя є платформа NetLogo, що є багатоагентним програмним середовищем для моделювання різних динамічних процесів. Тут також слід згадати програму інтерактивного візуального моделювання AnyLogic North America LLC, яка дає змогу оцінити ситуацію із трафіком у великому місті візуально.

Широкомасштабні дослідження проводяться провідними автомобільними компаніями у плані інтелектуальної системи керування режимом роботи світлофорів через двосторонній обмін інформацією із центром керування трафіком

(ЦКТ). Перехрестя використовується переважно для перетворення інформації світлофорів на бездротові сигнали та реалізації інтелектуального управління світлофорною системою. ЦКТ здійснює управління міськими дорогами та надає інформацію водіям автомобілів, а також передає електронні карти маршрутів. Інтелектуальна система управління роботою світлофорів на базі інтелектуального терміналу дає змогу водіям своєчасно отримувати інформацію про перемикання світлофорів. Система світлофорів автоматично визначає кількість автомобілів на дорогах, щоб миттєво змінювати час проїзду для автомобілів різних напрямків. Так досягається оптимізація маршрутів руху кожного ТЗ.

Важливим питанням проблеми трафіку є проблема паркінгу у великому місті. Ефективна технологія для цього запропонована компанією Siemens. Система контролює завантаженість вулиць і передає інформацію автомобілістам, використовуючи для кожної окремої стоянки інформацію, зчитувану за допомогою наземних датчиків або на основі кількості проданих паркувальних дозволів. В обох випадках система направляє водіїв безпосередньо на наявні місця для паркування, що запобігає переповненню вулиць та зменшує навантаження на трафік. Відбувається інтеграція в загальну систему керування трафіком. Це дає змогу використовувати бази даних паркування для надання автомобілістам рекомендацій щодо маршрутизації вже під час в'їзду у межі міста.

Масштабні дослідження проблем трафіку проводяться по всьому світу. Так, у штаті Юта (США) у розгляд введена система показників ефективності регулювання руху потоків ТЗ на основі аналізу даних, отриманих зі спеціальних мікрохвильових датчиків.

Зауважимо, що для вирішення проблеми дорожнього руху використовують різні наукові підходи, зокрема динаміку рідин. Водночас потоки рідин у трубах асоціюються із потоками ТЗ у містах. Такі модельні підходи досить ефективні та дають змогу певною мірою вирішити проблеми, пов'язані із дорожнім трафіком.

Важливою проблемою трафіку є питання, присвячене взаємодії приватних автомобілів та громадського транспорту. Для розв'язання проблем такої взаємодії вводиться поняття двотипних (бімодальних) міських мереж (bi-modal urban networks). Для організації ефективної взаємодії вказаних типів ТЗ вводиться у розгляд бімодальна Макроскопічна Фундаментальна Діаграма (МФД), що моделює змішаний трафік ТЗ згаданих видів. Результати показують, що запропонована технологія може значно: 1) зменшити затори у мережі; 2) поліпшити показники трафіку автобусів з погляду часу проїзду маршруту руху; 3) знизити рівень скупченості ТЗ на критичних ділянках транспортної мережі. У якості об'єкта досліджень вибрана транспортна мережа Сан-Франциско.

Загалом координована взаємодія між ТЗ та ЦКТ із зворотнім зв'язком має особливі перспективи з огляду на наявну тенденцію поширення автомобілів з так званими системами автопілота та появою на дорогах безпілотних ТЗ, що у майбутньому дасть змогу вивести керування транспортними потоками на кардинально новий, більш ефективний рівень зі скоординованими діями усіх учасників руху та упередженням проблемних ситуацій.

Стратегічна мета, яку ставлять перед собою дослідники транспортних мереж, полягає у побудові оптимальних маршрутів для кожного ТЗ та синхронізації потоків ТЗ. Інакше кажучи, ставиться завдання провести кожен ТЗ оптимальним маршрутом із урахуванням можливої корекції такого маршруту на основі оновлюваної кожні 10 секунд бази даних. Тобто система регулювання трафіку всієї сукупності автомобілів на трасах міста повинна прокладати динамічний (у режимі реального часу) і оптимальний маршрут кожному ТЗ, що замовляє лише кінцеву позицію маршруту (стартова позиція фіксується автоматично під час підключення до системи навігації). Центральний комп'ютер на ЦКТ, використовуючи комп'ютерну програму, записану далі, співпрацює із кожним водієм та передає йому голосові команди щодо маршруту руху до заявленого водієм пункту призначення, як у звичайній GPS-навігації. Особливість полягає у тому, що програма аналізує динамічну ситуацію на кожному перехресті, на ділянках дороги між сусідніми перехрестями і по всьому місту та відповідно прокладає маршрут з урахуванням ситуації на кожен конкретний момент часу. Водночас використовується комп'ютерна програма, що реалізує алгоритм Дейкстри щодо знаходження оптимального шляху. Отже, остаточною метою цієї технології є синхронізація транспортних потоків, оптимальне використання транспортних артерій всього міста (через перенаправлення потоків ТЗ на менш завантажені вулиці міста), запобігання утворенню заторів, а також супровід кожного ТЗ до місця призначення із розрахунком, щоб витрачений на поїздку час був мінімальним.

Технологія являє собою автоматизовану інтелектуальну систему регуляції дорожнього руху у великих містах, яку умовно можна розділити на два етапи. На першому етапі здійснюється регулювання трафіку через одне окреме перехрестя, що взаємодіє із сусіднім, а на другому – через усе місто. Розглянемо тепер детально перший етап регулювання трафіку.

Алгоритм Дейкстри використовується для знаходження оптимальних маршрутів між вибраною вершиною графа та всіма іншими його вершинами. Проте коли потрібно знаходити найкоротший маршрут *лише між двома конкретними вершинами графа* (що власне і потрібно кожному водію), тоді більш доцільно використати A*-алгоритм. До того ж цей алгоритм має меншу обчислювальну складність, ніж алгоритм Дейкстри (обчислювальна складність алгоритму

Дейкстри $O(n^2)$, а обчислювальна складність A*-алгоритму пропорційна кількості n вузлів графа, тобто $O(n)$).

Проблема трафіку у великому місті є надзвичайно актуальною сьогодні, але не вирішеною. Насамперед така проблема пов'язана із заторами на перехрестях і, як наслідок, складністю проїзду кожним ТЗ по вибраному маршруту.

Як уже підкреслювалось, базовим елементом у технології регулювання міського трафіку є перехрестя. Саме цей об'єкт, де перетинаються міські дороги, є основною причиною та джерелом заторів. Тому першочергово необхідно здійснити інтелектуальну регуляцію проїзду ТЗ через окреме перехрестя. Організувавши ефективний рух через такий об'єкт, досягнемо вищої ефективності трафіку по всьому місту.

З метою запобігання колапсу трафіку у вигляді довготривалих заторів пропонується застосувати мережу «розумних» перехресть, що охоплюють усі основні транспортні вузли міста. Але цього мало. Потрібно організувати оптимальний проїзд кожного ТЗ вздовж вибраного маршруту (нехай стартова позиція ТЗ з номером i позначається як S_i, а фінішна – F_i). Отже, кожному ТЗ приписується своєрідний маркер (S_i, F_i). Водій кожного такого об'єкта за допомогою мобільного телефону зі спеціальним додатком чи GPS-навігатором задає свій позиційний маркер та координує маршрут із ЦКТ. Так з'являється можливість контролювати та прокладати оптимальні маршрути для всіх ТЗ (водіїв таких ТЗ називатимемо IR-водіями). З огляду на вищесказане головна проблема полягає у поєднанні алгоритмів регулювання руху ТЗ як через окреме перехрестя, так і по всьому місту. За такого підходу затори у великих містах виникатимуть значно рідше, ніж це відбувається сьогодні, а трафік перейде на якісно новий рівень функціонування.

З одного боку, необхідно створити алгоритм і комп'ютерну програму, що забезпечать ефективне перемикання світлофорних фаз відповідно до завантаженості дорожніх напрямків. З іншого боку, необхідно контролювати не тільки перехрестя, а й усі смуги руху проїзної частини дороги одного напрямку. На рис. 6.18 схематично зображено траєкторію руху окремого ТЗ. На другому етапі необхідно прокласти оптимальний маршрут для кожного ТЗ, використовуючи, наприклад, A*-алгоритм та спектр даних, отримуваних із інфраструктури міської дорожньої мережі. Внаслідок застосування описаних двох фаз регулювання міського трафіку досягається оптимальний режим руху всього рухомого транспорту міста. Отже, пропонована технологія має практичний характер та спрямована на вирішення конкретної проблеми – проблеми трафіку у великому місті. Принципово найважливіша проблема трафіку зводиться до того, щоб кожен IR-водій проїжджав заявлений ним маршрут за мінімально короткий час. Для

цієї мети потрібно вирахувати такий маршрут на певний конкретний момент часу з урахуванням відповідної завантаженості перехресть та смуг руху, що їх проїжджатиме кожен автомобіль.

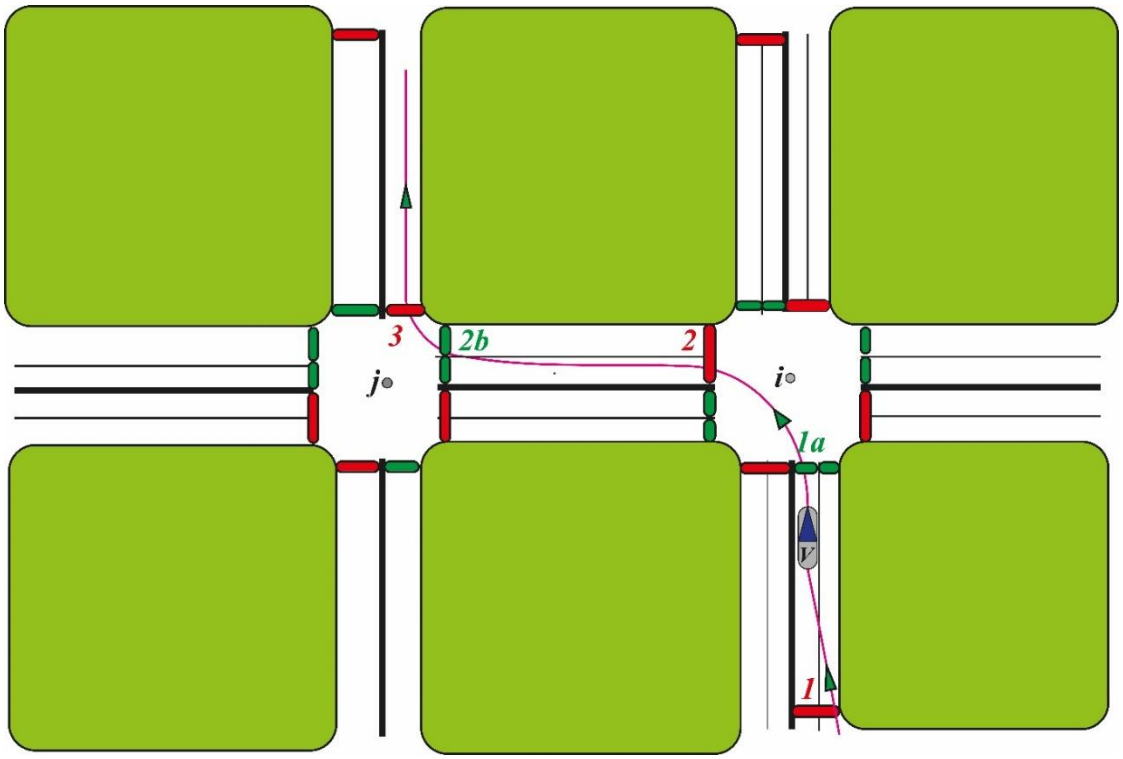


Рис. 6.18. Траєкторія руху автомобіля V, який послідовно перетинає перехрестя i та j і реєструється датчиками, розташованими на цих перехрестях

Як і у випадку ситуації з алгоритмом Дейкстри, так і під час використання A*-алгоритму, представимо транспортну мережу міста як навантажений мультиграф (рис. 6.19). Це перший ключовий меседж. Тепер можна використати теорію графів з метою прокладання оптимальних маршрутів. Другий меседж полягає у тому, що ваги ребер є величинами швидкозмінними, високо динамічними, тому їх треба реєструвати у режимі онлайн, постійно оновлюючи базу даних, де зберігаються ці величини. Вхідні та вихідні датчики видають спектр величин $N^a_{i\to j}$ та $n^b_{i\to j}$. Перша величина – це кількість автомобілів, що в'їхали на ділянку дороги між сусідніми перехрестями (на рис. 6.19 це ділянка дороги $(i\to j)_a$) протягом циклу роботи світлофора, а друга – це кількість автомобілів, що виїхали із i-ої смуги цієї ділянки дороги на сусідньому перехресті B за той самий час. Чим ближчим є відношення $N^a_{i\to j}\big/n^b_{i\to j}$ до одиниці, тим динаміка руху на вибраній ділянці дороги між перехрестями є кращою. У випадку, коли це відношення близьке до нуля, програма, шукаючи у графі оптимальний маршрут, омине таку завантажену ділянку дороги.

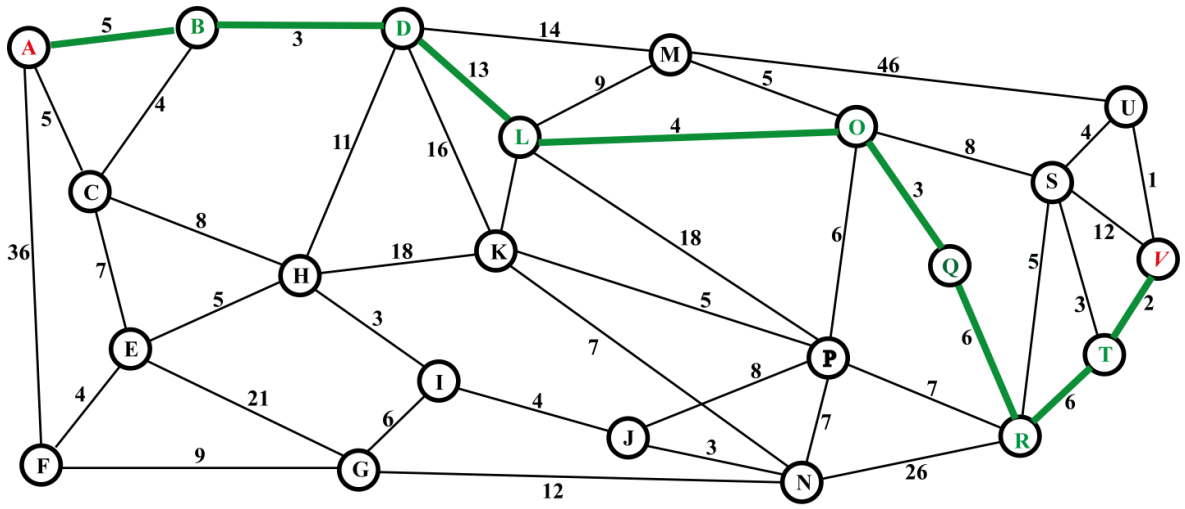


Рис. 6.19. Зважений граф, у якому з допомогою A*-алгоритму прокладений оптимальний маршрут між вершинами *A* і *V*. Оптимальний маршрут руху з урахуванням ваг ребер на конкретний момент часу виділений та пролягає через перехрестя у послідовності $A \to B \to D \to L \to O \to Q \to R \to T \to V$

Ваги ребер графа – величини динамічні (швидкозмінні) і відповідно система інтелектуального регулювання трафіку прокладатиме маршрути для всіх IR-водіїв із урахуванням дорожньої ситуації на конкретний момент часу. Інакше кажучи, система працює в режимі онлайн, що докорінно відрізняє її від подібних наявних систем. Для забезпечення роботи такого режиму використовується передача даних, отриманих на ЦКТ від датчиків міської мережі, до кожного користувача цими даними, тобто IR-водія. Але оскільки дані від міських датчиків оновлюються кожні 10 с, то і отримувані IR-водіями дані також матимуть квазі-онлайн-режим. Java-програма, що відповідає графу на рис. 6.19, виглядає так:

Лістинг 6.6

```
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
import java.util.List;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.Collections;

public class AstarSearchAlgo {
 // програма розраховує маршрут в неорієнтованому навантаженому графі
   public static void main(String[] args) {
      /ініціалізація графа
      Node A = new Node("A", 34);
```

```
Node B = new Node("B", 31);
Node C = new Node("C", 30);
Node D = new Node("D", 27);
Node E = new Node("E", 29);
Node F = new Node("F", 33);
Node G = new Node("G", 27);
Node H = new Node("H", 24);
Node I = new Node("I", 23);
Node J = new Node("J", 19);
Node K = new Node("K", 19);
Node L = new Node("L", 19);
Node M = new Node("M", 20);
Node N = new Node("N", 16);
Node O = new Node("O", 13);
Node P = new Node("P", 13);
Node Q = new Node("Q", 9);
Node R = new Node("R", 8);
Node S = new Node("S", 6);
Node T = new Node("T", 3);
Node U = new Node("U", 3);
Node V = new Node("V", 0);
/ініціалізація ребер
A.adjacencies = new Edge[]{
        new Edge(B, 5),
        new Edge(C, 5),};
B.adjacencies = new Edge[]{
        new Edge(D, 3),
        new Edge(C, 4),
        new Edge(F, 36)};
C.adjacencies = new Edge[]{
        new Edge(B, 4),
        new Edge(D, 7),
        new Edge(E, 7),
        new Edge(H, 8)};
D.adjacencies = new Edge[]{
        new Edge(M, 14),
        new Edge(L, 13),
        new Edge(K, 16),
        new Edge(H, 11),
```

```
        new Edge(C, 7),
        new Edge(B, 3),};
E.adjacencies = new Edge[]{
        new Edge(C, 7),
        new Edge(H, 5),
        new Edge(F, 4),
        new Edge(G, 21),};
F.adjacencies = new Edge[]{
        new Edge(E, 4),
        new Edge(G, 9),
        new Edge(A, 36),};
G.adjacencies = new Edge[]{
        new Edge(F, 9),
        new Edge(N, 12),
        new Edge(I, 6),
        new Edge(E, 21),};
H.adjacencies = new Edge[]{
        new Edge(D, 11),
        new Edge(E, 5),
        new Edge(I, 3),
        new Edge(C, 8),
        new Edge(K, 18)};
I.adjacencies = new Edge[]{
        new Edge(J, 4),
        new Edge(H, 3),
        new Edge(I, 6),};
J.adjacencies = new Edge[]{
        new Edge(P, 8),
        new Edge(N, 3),
        new Edge(I, 4)};
K.adjacencies = new Edge[]{
        new Edge(L, 5),
        new Edge(P, 5),
        new Edge(N, 7),
        new Edge(D, 16),
        new Edge(H, 18)};
L.adjacencies = new Edge[]{
        new Edge(M, 9),
        new Edge(O, 4),
```

```
        new Edge(K, 5),
        new Edge(D, 13),
        new Edge(P, 18)};
M.adjacencies = new Edge[]{
        new Edge(O, 5),
        new Edge(L, 9),
        new Edge(D, 14),
        new Edge(U, 46),};
N.adjacencies = new Edge[]{
        new Edge(K, 7),
        new Edge(P, 7),
        new Edge(G, 12),
        new Edge(J, 3),
        new Edge(R, 26)};
O.adjacencies = new Edge[]{
        new Edge(M, 5),
        new Edge(L, 4),
        new Edge(Q, 3),};
P.adjacencies = new Edge[]{
        new Edge(K, 5),
        new Edge(J, 8),
        new Edge(N, 7),
        new Edge(Q, 4),
        new Edge(R, 7),
        new Edge(L, 18),};
Q.adjacencies = new Edge[]{
        new Edge(P, 4),
        new Edge(O, 3),
        new Edge(R, 6),};
R.adjacencies = new Edge[]{
        new Edge(P, 7),
        new Edge(Q, 6),
        new Edge(S, 5),
        new Edge(T, 6),
        new Edge(N, 26),};
S.adjacencies = new Edge[]{
        new Edge(O, 8),
        new Edge(R, 5),
        new Edge(U, 4),
```

```
            new Edge(T, 3),
            new Edge(V, 12),};
        T.adjacencies = new Edge[]{
            new Edge(V, 2),
            new Edge(R, 6),
            new Edge(S, 3),};
        U.adjacencies = new Edge[]{
            new Edge(S, 4),
            new Edge(V, 1),
            new Edge(M, 46),};
        V.adjacencies = new Edge[]{
            new Edge(U, 1),
            new Edge(T, 2),
            new Edge(S, 12),};
        AstarSearch(A, V);
        List<Node> path = printPath(V);
        System.out.println("Path: " + path);
    }

    public static List<Node> printPath(Node target) {
        List<Node> path = new ArrayList<Node>();
        for (Node node = target; node != null; node = node.parent) {
            path.add(node);
        }
        Collections.reverse(path);
        return path;
    }

    public static void AstarSearch(Node source, Node goal) {
        Set<Node> explored = new HashSet<Node>();
        PriorityQueue<Node> queue = new PriorityQueue<Node>(30, new
Comparator<Node>() {
            public int compare(Node i, Node j) {
            if (i.f_scores > j.f_scores) {
                return 1;
            } else if (i.f_scores < j.f_scores) {
                return -1;
            } else {
                return 0;
```

```
            }
        }
    });
    source.g_scores = 0;
    queue.add(source);
    boolean found = false;
    while ((!queue.isEmpty()) && (!found)) {
        /вузол, що має найменшу величину f_score
        Node current = queue.poll();
        explored.add(current);
        //знаходження цілі
        if (current.value.equals(goal.value)) {
            found = true;
        }
        //перевірка кожного нащадка поточного вузла
        for (Edge e : current.adjacencies) {
            Node child = e.target;
            double cost = e.cost;
            double temp_g_scores = current.g_scores + cost;
            double temp_f_scores = temp_g_scores + child.h_scores;
             if ((explored.contains(child)) &&
                 (temp_f_scores >= child.f_scores)) {
              continue;
            } else if ((!queue.contains(child)) ||
                 (temp_f_scores < child.f_scores)) {
              child.parent = current;
              child.g_scores = temp_g_scores;
              child.f_scores = temp_f_scores;
              if (queue.contains(child)) {
                 queue.remove(child);
              }
              queue.add(child);
            }
        }
    }
}
class Node {
   public final String value;
```

```
    public double g_scores;
    public final double h_scores;
    public double f_scores = 0;
    public Edge[] adjacencies;
    public Node parent;

    public Node(String val, double hVal) {
        value = val;
        h_scores = hVal;
    }

    public String toString() {
        return value;
    }
}
class Edge {
    public final double cost;
    public final Node target;

    public Edge(Node targetNode, double costVal) {
        target = targetNode;
        cost = costVal;
    }
}
```

На початку програми використовується ряд конструкторів, що вводять евристичні відстані, тобто значення функції h(*n*) для кожного вузла графа. Далі ініціалізуються усі вузли графа та описуються сусідні до кожного вузла ребра, тобто вводяться їх ваги. Аналізуючи введені дані, програма прокладе оптимальний маршрут ***Path: [A, B, D, L, O, Q, R, T, V],*** що є оптимальним лише на певний проміжок часу за відповідної завантаженості ребер графа. Такий маршрут виділено на рис. 6.19 – це потовщена ламана лінія. Дані щодо такого маршруту будуть передаватись IR-водієві із ЦКТ, поки ситуація на трасі руху не зміниться. У такому випадку програма прорахує новий маршрут. І так для всіх ТЗ по усьому місту. Внаслідок цього відбудеться цілковита синхронізація транспортних потоків, що приведе до повного зникнення заторів у транспортній мережі та дасть змогу кожному водієві прибувати до місця призначення за мінімально короткий проміжок часу. Отже, міський трафік перейде на якісно новий рівень.

Тестування приведеної програми проводилось на графах із кількістю вершин до 150 та кількістю ребер 430. Час виконання становив 2 с 580 мс. Як згадувалось, обчислювальна складність A*-алгоритму лінійна $O(n)$, що робить його досить ефективним для задач із великим об'ємом вхідних даних.

6.8. Ігрові моделі

Ігрове поле у комп'ютерній ігровій моделі часто використовується для переміщення гравця (ігрового агента) з одного місця на інше, скажімо, з точки А в точку В. Тому важливо поставити питання: «Як зробити таке переміщення оптимальним з погляду геометричної протяжності маршруту переміщення?». Інакше кажучи, необхідно прокласти найкоротший маршрут в ігровому полі. Загалом переміщення об'єкта із однієї ігрової позиції в іншу можливе за допомогою кількох варіантів, але у будь-якому випадку необхідно розв'язати такі проблеми:

1. Як дістатись із позиції А в Б?
2. Як обійти перешкоди під час руху?
3. Як знайти найкоротший (оптимальний) варіант шляху з усіх можливих варіантів?
4. Як знайти оптимальний маршрут якнайшвидше?

Найефективнішим способом досягнення поставлених цілей є використання A*-алгоритму, який застосовується для прокладання оптимальних маршрутів у графах. Нехай ігрове поле складене з квадратних комірок (плиток). Плитки пронумеровані: їх номери проставлені всередині плиток зверху зліва (рис. 6.20). **Завдання полягає у тому, щоб в оптимальний спосіб провести ігрового агента зі стартової ігрової позиції у фінішну.** Для вирішення цієї проблеми на першому етапі скористаємось описовим варіантом A*-алгоритму. Для цього знову розглянемо функцію $f(x) = g(x) + h(x)$, де $g(x)$ – функція, що визначає вагу (відстань) маршруту від стартової позиції до прохідної плитки з номером x, а $h(x)$ – функція евристичної відстані. Зауважимо, що визначення цієї функції неоднозначне та залежить від постановки задачі і характеристик ігрового поля. Стандартна евристична функція для квадратної ігрової решітки являє собою так звану манхеттенську відстань. Введення такого типу евристичної функції обумовлене тим, що у багатьох випадках ігровий агент може рухатись тільки вздовж горизонталей та вертикалей – саме так розташовані квартали Манхеттена – центральної частини міста Нью-Йорка. Якраз таку евристику ми будемо використовувати у нашій задачі.

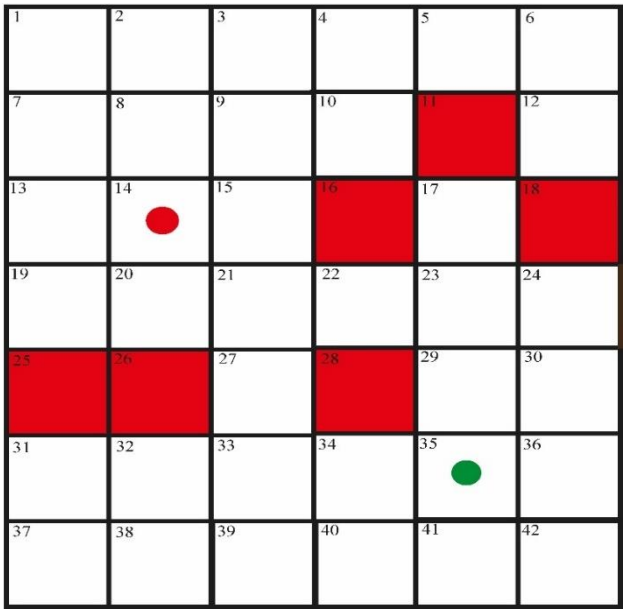


Рис. 6.20. Ігрове поле з перешкодами – замальованими комірками.
Зліва зверху у кожній комірці проставлені їх номери.
У комірці 14 знаходиться стартова позиція ігрового агента, а у комірці 35 – фінішна

Проте досить часто евристична відстань вимірюється по прямій, оскільки пряма є найкоротшою відстанню між двома точками. Зокрема, використовуючи A*-алгоритм для попередньої задачі (рис. 6.21), евристика являла собою пряму між кожною поточною вершиною n та кінцевою вершиною маршруту P. Зауважимо також, що у деяких випадках виміряти евристичну відстань проблематично або взагалі неможливо (скажімо, коли йдеться про інтернет-мережу, то розташування серверів та окремих комп'ютерів може бути невідомим). У цьому випадку необхідно використовувати алгоритми без застосування евристики.

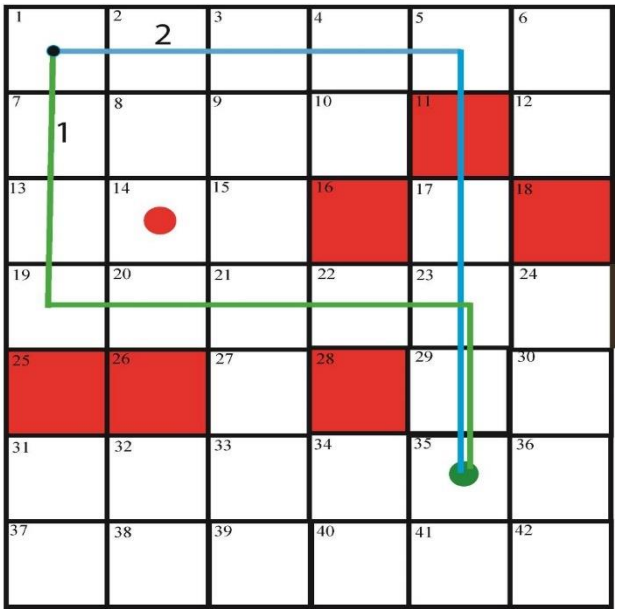


Рис. 6.21. Маршрути, що з'єднують комірки 1 та 35,
демонструють метод визначення евристичної відстані

На рис. 6.21 показано два можливі манхеттенські маршрути. Маршрут 1 проходить виключно по відкритих плитках. Але часто ігрове поле має високу густину закритих плиток, і тому треба прокладати маршрут через закриті ігрові плитки за допомогою маршруту під номером 2.

Як підкреслювалось вище, на противагу манхеттенській евристиці доцільно розраховувати відстань між поточною вершиною та фінішною вершиною по прямій. Інакше кажучи, потрібно використовувати так звану евклідову евристику, вимірюючи саме найкоротшу відстань між двома точками. Водночас необхідно щоразу обчислювати квадратний корінь, але за відповідної швидкодії процесора така перешкода не є суттєвою. Фактично треба розрахувати спектр відстаней виду $h(n)= Math.sqrt((x_n - x_f)^2 + (y_n - y_f)^2)$, де x_n – координата поточної вершини, взята по осі абсцис; x_f УДК 658.5:005.4(075.8) аналогічна координата, але для фінішної вершини; y_n – координата поточної вершини, взята по осі ординат; y_f – y-координата фінішної вершини.

Інколи евристика може являти собою суму двох відрізків. Це може трапитись у випадку, коли граф розділений на дві частини, які з'єднані між собою лише одним ребром (у теорії графів таке ребро називається мостом). Припустимо, необхідно прокласти оптимальний маршрут між вершинами графа, одна із яких знаходиться в одній частині графа, а інша – у частині графа, відділеній від першої частини мостом. У такому випадку треба застосувати евристику у вигляді двох прямих відрізків: один із них пролягає від стартової вершини до мосту, а інший – від мосту до фінішної вершини.

Отже, вибір евристичної функції залежить від виду конкретного графа (інакше кажучи, від геометрії розташування об'єктів, що моделюються графом).

Нехай кожна плитка (tile) має розміри 10×10. В ігровому полі зобразимо перешкоди – це зафарбовані плитки. У комірках такого типу ігровий агент перебувати не може, на відміну від відкритих комірок, через які можливе пересування ігрових об'єктів. Стартова позиція ігрового агента виділена та являє собою кружечок, що знаходиться у комірці під номером 14. Фінішна позиція – це кружечок, що знаходиться у комірці 35 (рис. 6.21). Необхідно провести ігрового агента зі стартової позиції у фінішну найкоротшим шляхом. Вважається, що ігровий агент може рухатись по горизонтальних відкритих плитках, по вертикальних відкритих плитках, а також у діагональному напрямку плиток такого типу. Оскільки в іграх зазвичай ставиться завдання пройти зі стартової позиції у фінішну оптимальним маршрутом, то необхідно визначити принцип обрахування евристичної відстані. У цій задачі використаємо обрахунок евристичних відстаней, як показано на рис. 6.21. Для конкретного випадку евристичну функцію можна записати так:

$$h(n)=abs(x_n - x_f)+abs(y_n - y_f), \qquad (6.10)$$

де x_n – x-координата однієї із прохідних плиток; x_f – x-координата фінішної плитки; abs – позначення абсолютної величини; y_n – y-координата прохідної плитки, y_f – y-координата фінішної плитки.

Якщо рух ігрового агента дозволений по діагоналях плиток, тоді евристика має значно складнішу структуру. У такому випадку, очевидно, більш доцільним варіантом буде використання у якості евристичної відстані або пряму лінію, що з'єднує початкову та кінцеву точку ігрового агента, або кілька прямолінійних відрізків, що проходять між забороненими для проходження плитками.

Вважаючи, що відстань між центрами сусідніх плиток по горизонталі та вертикалі складає 10 умовних одиниць, можна обрахувати всі евристичні відстані між будь-якою коміркою та коміркою 35, де знаходиться кінцева позиція руху ігрового агента. Обрахунок здійснюється за формулою $h(n) = 10 \times k$, де k – кількість проміжків між центрами початкової та кінцевої плиток (наприклад, вздовж ламаних ліній на рис. 6.21 кількість таких проміжків дорівнює 9).

Сформуємо відкритий список – список плиток, що оточують стартову позицію (рис. 6.22). Цей список спочатку буде складатись із 8 плиток з номерами 7, 8, 9, 13, 15, 19, 20 та 21. Закритий список буде містити всього одну плитку за номером 14, тобто стартову плитку. Наступний крок – визначення величин f(n) = g(n) + h(n). Тут n – номер проміжної плитки, тобто плитки, що лежить між стартовою та фінішною плитками. Функція g(n) являє собою реальну відстань між стартовою та проміжною плитками, а h(n) – це так звана евристична відстань, яка в нашій задачі вимірюється, як показано на рис. 6.21 – або вздовж маршруту 1, або вздовж маршруту 2.

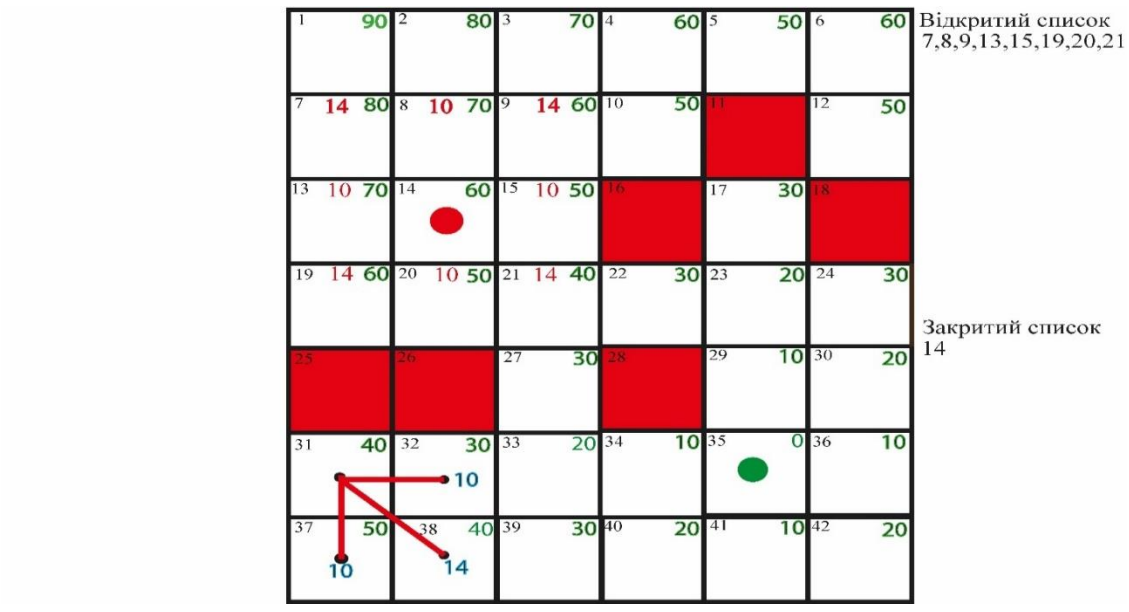


Рис. 6.22. Навколо стартової позиції агента (плитка 14) розташовані вісім плиток, які формують так званий відкритий список. У відкритих плитках зверху справа виставлені величини евристичних відстаней

Зупинимось більш детально на поняттях прохідних та непрохідних плиток. Зрозуміло, що це певною мірою абстракція. Реально маршрут в ігровому полі може складатися із плиток різної прохідності, або словами теорії графів – із ребер з різними вагами. Така ситуація є природнішою, оскільки моделює реальний земний ландшафт із його рівнинами, горами, лісами та річками. У цьому випадку доцільно ввести поняття коефіцієнта складності маршруту k_{ij}. Для кожної ігрової плитки (або для кожного відрізка маршруту) матричні елементи k_{ij} приймають значення, що відповідають реальній складності проходження конкретного елемента шляху. Інакше кажучи, якщо ігрове поле складається із плиток з різною прохідністю, то ситуацію можна графічно представити у вигляді рис. 6.23.

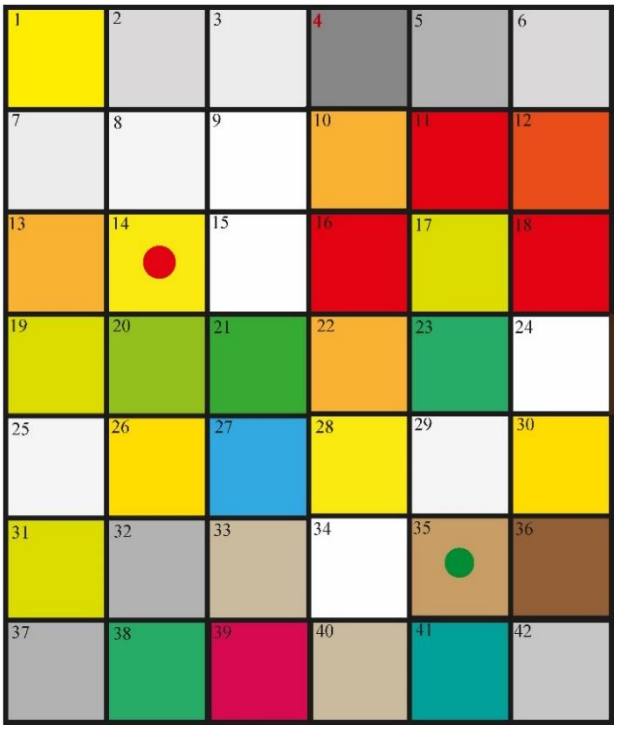


Рис. 6.23. Ігрове поле з різною прохідністю плиток

Якщо умовно встановити відповідність між кольором плитки та її прохідністю (наприклад, білий колір – $k=1$, червоний – $k=2$, оранжевий – $k=3$, жовтий – $k=4$, зелений – $k=5$, блакитний – $k=6$, синій – $k=7$, фіолетовий – $k=8$, сірий – $k=9$, чорний – $k=10$), тоді будь-якому ігровому полю можна у відповідність із прохідністю його плиток зіставити матрицю:

$$k_{ij} = \begin{pmatrix} k_{11}\ k_{12}\ k_{13} \ldots k_{1i}\ \ldots k_{1n} \\ k_{21}\ k_{22}\ k_{23} \ldots k_{2i}\ \ldots k_{2n} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ k_{j1}\ k_{j2}\ k_{j3} \ldots k_{ji} \ldots k_{jn} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ k_{n1}\ k_{n2}\ k_{n3} \ldots k_{ni} \ldots k_{nn} \end{pmatrix}. \qquad (6.11)$$

Після такого співставлення довжина маршруту ігрового агента в горизонтальному або вертикальному напрямках може бути представлена у вигляді:

$$W = \sum_{ij} k_{ij} l_{ij}\,, \qquad (6.12)$$

де l_{ij} – матриця, що відповідає геометричній довжині кожної плитки; якщо всі плитки ігрового поля еквівалентні, то тоді із (6.12) отримуємо:

$$W = l \sum_{ij} k_{ij} \qquad (6.13)$$

Далі розглянемо спрощену задачу прокладання оптимального маршруту в ігровому полі, вважаючи, що прохідність плиток приймає лише два значення: повна прохідність (k = 1) – це незабарвлені плитки та нульова прохідність (k = Ґ) – це забарвлені червоним кольором плитки. Для обчислення значення функції f(n) у нашому спрощеному варіанті ігрового поля скористаємось сталістю геометричних розмірів плиток (10×10) та незмінною величиною коефіцієнта прохідності (k = 1). Після цього значення функції f(n) виставимо у центрі кожної плитки з відкритого списку, як показано на рис. 6.24. Вважатимемо, що ігровий агент має змогу рухатись по діагоналі між прохідними плитками, а також по діагоналі між непрохідними плитками. Наприклад, ігровий агент може пройти між плитками 11 та 16 (рис. 6.24). Припускаємо також, що евристичний маршрут може проходити як через прохідні плитки, так і через непрохідні. На рис. 6.21 показано два принципово відмінні маршрути, що визначають евристичну відстань: маршрут 1 проходить виключно через прохідні плитки, водночас маршрут під номером 2 перетинає непрохідну плитку під номером 11.

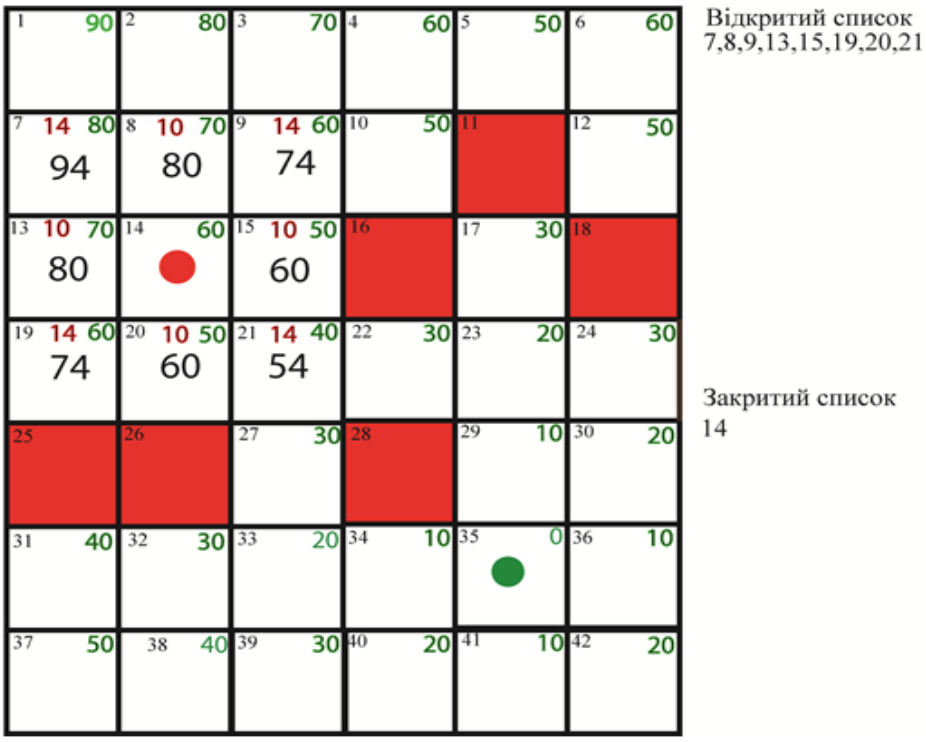


Рис. 6.24. У центах плиток відкритого списку, що оточують стартову позицію ігрового агента, проставлені значення функції f(n)

Аналіз значень функції f(n) дає змогу встановити її мінімальну величину, рівну 54 (рис. 6.25). Наступний крок полягає у розширенні закритого списку плиток, вибираючи плитки із відкритого списку, але із мінімальними величинами f(n). Це дає змогу приєднати до списку закритих плиток комірку із номером 21, що характеризується величиною функції f(n), рівною 54 – мінімальному значенню серед представленого спектра величин.

Сусіди вказаної плитки мають номери 22 та 27. Визначимо значення функції f(n) для цих плиток та приєднаємо їх до списку відкритих плиток. Тепер знову серед відкритих плиток встановимо плитки з мінімальними значеннями функції f(n). Це будуть щойно відкриті плитки, тобто плитки за номерами 22 та 27 – саме значення 54 є мінімальним серед усіх відкритих плиток (рис. 6.25). Розширимо спектр закритих плиток так само, як і відкритих (рис. 6.26). Відповідні дані занесемо у списки, представлені справа від ігрового поля. Внаслідок цього отримаємо 4 закриті плитки та до списку відкритих плиток додамо 6 плиток. Як і у попередніх варіантах, проведемо аналіз значень функції f(n) для відкритих плиток. Коли працюємо з невеликим масивом відкритих плиток, таку сепарацію плиток за значенням функції f(n) провести просто. Але у випадку великого масиву краще названу операцію доручити комп'ютерній програмі, яка виставляє величини у порядку їх зростання. Наведемо код цієї програми (Лістинг 6.7).

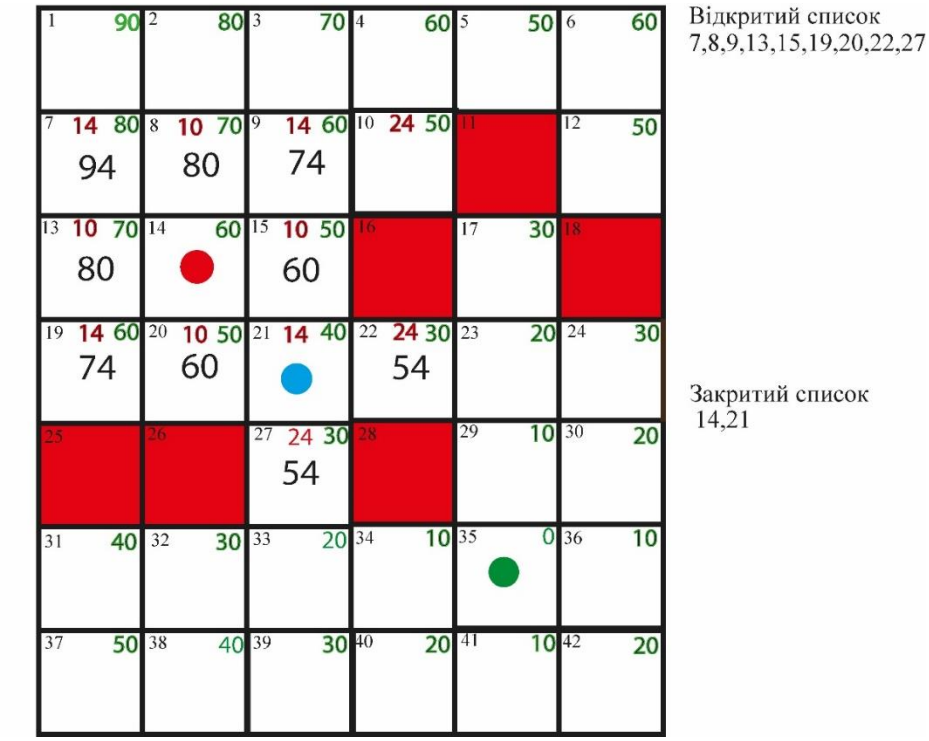


Рис. 6.25. Навколо закритих плиток виставлені значення функції f(*n*) = g(*n*) + h(*n*) та величини її складників – g(*n*) та h(*n*). Значення функції f(*n*) наведені в центрі плитки, а значення величин g(n) та h(*n*) – відповідно зверху в центрі та зверху справа

Лістинг 6.7

```
package sorting;

    import java.util.*;
/**
 * Created by pk on 07.06.2023.
 */
public class SortDemo {
   public static void main(String[] args) {
      int[] a = {94, 80, 74, 80, 60, 68, 74, 60, 54, 48, 68, 54, 48};
      Arrays.sort(a);
      for (int n : a) {
         System.out.print( "  " +n);
      }
   }
}
```

Внаслідок роботи програми отримуємо відсортований список: 48 48 54 54 60 60 68 68 74 74 80 80 94. Із наведеного списку вибираємо мінімальне / мінімальні значення функції f(*n*) – одне або кілька – та заносимо ці дані у закритий список і паралельно оновлюємо відкритий список. Для нового відкритого списку повторюємо процедуру ранжування, поки не дійдемо до фінішної

плитки. Така процедура, повторена за допомогою циклу for, дасть змогу досить швидко прокласти оптимальний маршрут, якщо йдеться про одноагентну ігрову ситуацію. У випадку багатоагентної ігрової ситуації, коли необхідно прокласти оптимальні маршрути кількох ігровим агентам, необхідно скористатись алгоритмом Флойда, який застосовується для прокладання оптимальних маршрутів між усіма вершинами графа.

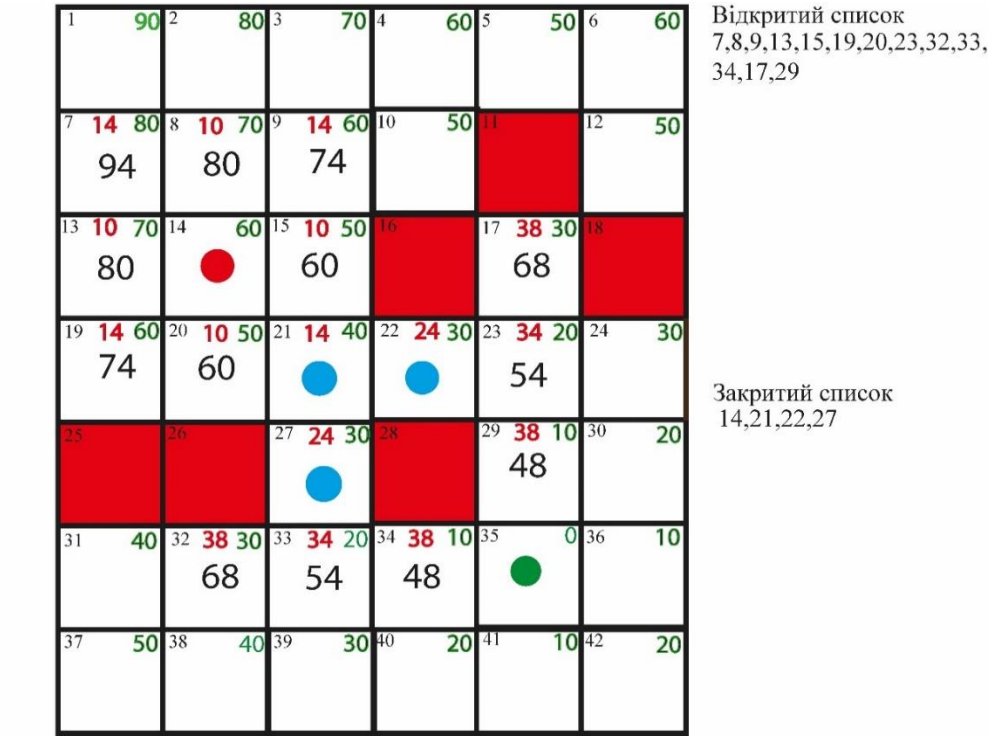


Рис. 6.26. Закритий список складається з чотирьох плиток, навколо яких розташовуються плитки із відкритого списку

На рис. 6.27 наведено остаточний результат прокладання маршруту у випадку одного ігрового агента. Звернемо увагу, що у плитці 21 маршрут розділяється на дві рівновеликі вітки – A та B. Кожен із цих двох маршрутів має однакову мінімальну вагу, рівну 48 умовним одиницям довжини.

У своїй роботі «Pathfinding in Games» Adi Botea зі співавторами приводить висновки, які у перекладі звучать так: «Комерційні ігри можуть бути чудовим тестовим майданчиком для дослідження штучного інтелекту, який є середнім між синтетичними, дуже абстрактними академічними тестами та більш складними проблемами із реального життя. Серед численних технологій штучного інтелекту та проблем, пов’язаних з іграми, як-от навчання, планування та налаштування мов спілкування, прокладання шляху в ігровому полі є одним із найпоширеніших застосувань штучного інтелекту для ігор. Почнемо із алгоритму популярного, базового підходу до пошуку шляхів. Три основні елементи такого підходу – представлення ігрового поля у вигляді графа, пошук алгоритму та

евристичної функції для здійснення пошуку. Ігрові поля у вигляді решітки є популярним способом представлення ігрового поля в пошуковому графі. За такого підходу ігрове поле розділяється на квадратні клітинки, які часто називаються плитками. Залежно від стратегії гри деякі ігрові плитки позначаються як прохідні, інші як заблоковані, або непрохідні. Ігровий агент може займати лише одну прохідну плитку на певний момент часу. Прохідні плитки стають вузлами у графі пошуку. Ребрами графа стають лінії, що з'єднують суміжні прохідні плитки. Залежно від типу ігрового поля, можна визначити або 4 напрямки руху ігрового агента (вверх-вниз та вправо-вліво) або 8 напрямків (до вказаних напрямків додаються ще напрямки по діагоналях плиток).

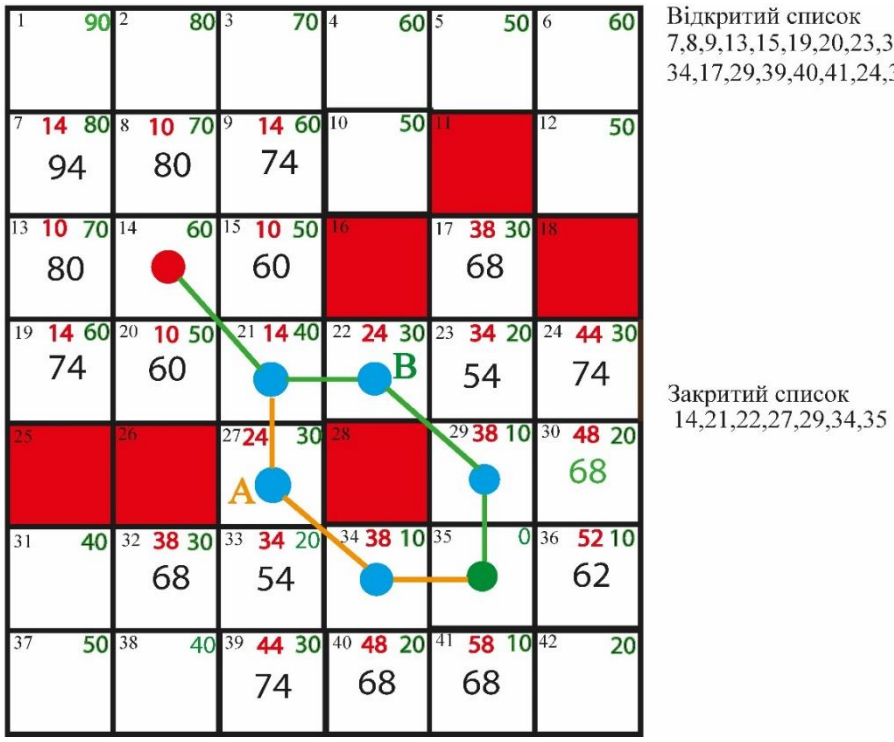


Рис. 6.27. Показані два оптимальні рівноцінні маршрути A та B, довжина кожного з яких складає 48 одиниць шляху

Найбільш прийнятним варіантом алгоритму, що знаходить оптимальний шлях в ігровому полі, є A*-алгоритм. Вище ми вже розглядали конкретний приклад використання цього алгоритму. Принциповим питанням, як уже підкреслювалось, є питання вибору евристичної функції h(n). Важливо також зіставити ігрове поле, складене з плиток, із графом. Вважатимемо, що центр плитки (tile) являє собою вузол графа, а проміжок між центрами сусідніх плиток – ребро графа. Отже, на доступні плитки ігрового поля накладаємо граф (рис. 6.28). Тепер можна скористатись алгоритмом пошуку оптимального маршруту за допомогою програми. Саме цей алгоритм знаходить оптимальний шлях між двома вибраними вершинами графа, витрачаючи під час цього мінімальні

обчислювальні ресурси. В ігровому варіанті використання A*-алгоритму вузли графа можуть являти собою комірки 2D-решітки ігрового поля. Тут слід зауважити, що графи, на яких використовується A*-алгоритм, можуть мати різні ребра, наприклад, ненаправлені, однонаправлені, або двонаправлені. В нашій задачі вважатимемо, що працюємо з навантаженим неорієнтованим графом. Ваги ребер вздовж горизонталі та вертикалі рівні і складають величину 10. Ваги ребер між сусідніми плитками по горизонталі визначаються як $10 \cdot \sqrt{2} \approx 14$. Тепер можна ввести дані про граф, представлений на рис. 6.28, у код програми. Треба лише у цій програмі перевизначити номери вузлів та їх евристичні відстані, як це зроблено у лістингу 6.8.

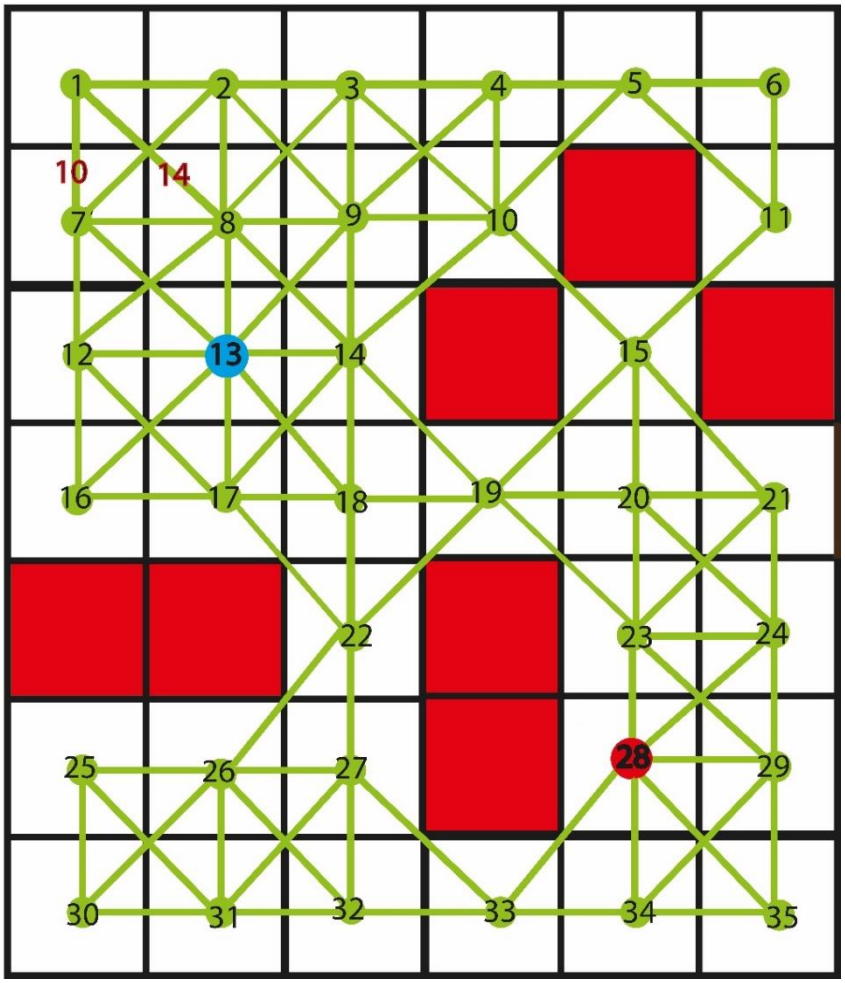


Рис. 6.28. На відкриті плитки ігрового поля накладено зважений граф. Вага кожного ребра між сусідніми вершинами по горизонталі та вертикалі дорівнює 10, а по діагоналі – 14 (показані ваги лише для двох ребер – інші аналогічні ребра мають такі ж ваги). Стартова позиція (вершина 13) та фінішна (вершина 28) виділені

Внаслідок виконання програми отримаємо маршрут $14 \to 21 \to 27 \to 34 \to 35$.

Цей маршрут співпадає із маршрутом А, що був отриманий у випадку виконання словесного алгоритму, результати роботи якого приведені на рис. 6.27.

Лістинг 6.8

```
import java.util.PriorityQueue;
import java.util.HashSet;
import java.util.Set;
import java.util.List;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.Collections;
public class AstarSearchAlgo {
public static void main(String[] args) {
Node n1 = new Node("1", 90);
Node n2 = new Node("2", 80);
Node n3 = new Node("3", 70);
Node n4 = new Node("4", 60);
Node n5 = new Node("5", 50);
Node n6 = new Node("6", 60);
Node n7 = new Node("7", 80);
Node n8 = new Node("8", 70);
Node n9 = new Node("9", 60);
Node n10 = new Node("10",50);
Node n11 = new Node("11", 50);
Node n12 = new Node("12", 70);
Node n13 = new Node("13", 60);
Node n14 = new Node("14", 50);
Node n15 = new Node("15", 30);
Node n16 = new Node("16", 60);
Node n17 = new Node("17", 50);
Node n18 = new Node("18", 40);
Node n19 = new Node("19", 30);
Node n20 = new Node("20", 20);
Node n21 = new Node("21", 30);
Node n22 = new Node("22", 30);
Node n23 = new Node("23", 10);
Node n24 = new Node("24", 20);
Node n25 = new Node("25", 40);
Node n26 = new Node("26",30);
```

```
Node n27 = new Node("27", 20);
Node n28 = new Node("28", 0);
Node n29 = new Node("29", 10);
Node n30 = new Node("30", 50);
Node n31 = new Node("31", 40);
Node n32 = new Node("32", 30);
Node n33 = new Node("33", 20);
Node n34 = new Node("34", 10);
Node n35 = new Node("35", 20);
n1.adjacencies = new Edge[]{
        new Edge(n2, 10),
        new Edge(n8, 14),
        new Edge(n7, 10),};
n2.adjacencies = new Edge[]{
        new Edge(n1, 10),
        new Edge(n3, 10),
        new Edge(n9, 14) ,
        new Edge(n8, 10) ,
        new Edge(n7, 14),

};

n3.adjacencies = new Edge[]{
        new Edge(n2, 10),
        new Edge(n4, 10),
        new Edge(n10, 14),
        new Edge(n9, 10),
        new Edge(n8, 14),
};
n4.adjacencies = new Edge[]{
        new Edge(n3, 10),
        new Edge(n5, 10),
        new Edge(n10, 10),
        new Edge(n9, 14),
};
n5.adjacencies = new Edge[]{
        new Edge(n4, 10),
        new Edge(n6, 10),
        new Edge(n11, 14),
```

```
            new Edge(n10, 14),

};
n6.adjacencies = new Edge[]{
            new Edge(n5, 10),
            new Edge(n11, 10),

};
n7.adjacencies = new Edge[]{
            new Edge(n1, 10),
            new Edge(n2, 14),
            new Edge(n8, 10),
            new Edge(n13, 14),
            new Edge(n12, 10),

};
n8.adjacencies = new Edge[]{
            new Edge(n1, 14),
            new Edge(n2, 10),
            new Edge(n3, 14),
            new Edge(n9, 10),
            new Edge(n14, 14),
            new Edge(n13, 10),
            new Edge(n12, 14),
            new Edge(n7, 10)
    };
n9.adjacencies = new Edge[]{
            new Edge(n2, 14),
            new Edge(n3, 10),
            new Edge(n4, 14),
            new Edge(n10, 10),
            new Edge(n14, 10),
            new Edge(n13, 14),
            new Edge(n8, 10),
};
n10.adjacencies = new Edge[]{
            new Edge(n3, 14),
            new Edge(n4, 10),
            new Edge(n5, 14),
```

```
            new Edge(n15, 14),
            new Edge(n14, 14),
            new Edge(n9, 10),
    };
    n11.adjacencies = new Edge[]{
            new Edge(n5, 14),
            new Edge(n6, 10),
            new Edge(n15, 14),
              };
    n12.adjacencies = new Edge[]{
            new Edge(n7, 10),
            new Edge(n8, 14),
            new Edge(n13, 10),
            new Edge(n17, 14),
            new Edge(n16, 10),
    };
    n13.adjacencies = new Edge[]{
            new Edge(n7, 14),
            new Edge(n8, 10),
            new Edge(n9, 14),
            new Edge(n14, 10),
            new Edge(n18, 14),
            new Edge(n17, 10),
            new Edge(n16, 14),
            new Edge(n12, 10),
    };
    n14.adjacencies = new Edge[]{
            new Edge(n8, 14),
            new Edge(n9, 10),
            new Edge(n10, 14),
            new Edge(n19, 14),
            new Edge(n18, 10),
            new Edge(n17, 14),
            new Edge(n13, 10),
    };
    n15.adjacencies = new Edge[]{
       new Edge(n10, 14),
            new Edge(n11, 14),
            new Edge(n20, 10),
```

```
        new Edge(n19, 14),
};
n16.adjacencies = new Edge[]{
   new Edge(n12, 10),
        new Edge(n13, 14),
        new Edge(n17, 10),
};
n17.adjacencies = new Edge[]{
        new Edge(n12, 14),
        new Edge(n13, 10),
        new Edge(n14, 14),
        new Edge(n18, 10),
        new Edge(n16, 10),
        new Edge(n12, 14),
};
n18.adjacencies = new Edge[]{
        new Edge(n13, 14),
        new Edge(n14, 10),
        new Edge(n19, 10),
        new Edge(n17, 10),
        new Edge(n22, 10),
};
n19.adjacencies = new Edge[]{
        new Edge(n14, 14),
        new Edge(n15, 14),
        new Edge(n20, 10),
        new Edge(n23, 14),
        new Edge(n22, 14),
        new Edge(n18, 10),
};
n20.adjacencies = new Edge[]{
       new Edge(n19, 10),
        new Edge(n15, 10),
        new Edge(n21, 10),
        new Edge(n24, 14),
        new Edge(n23, 10),
          };
   n21.adjacencies = new Edge[]{
        new Edge(n20, 10),
```

```
        new Edge(n15, 14),
        new Edge(n24, 10),
        new Edge(n23, 14),
};
n22.adjacencies = new Edge[]{
        new Edge(n18, 10),
        new Edge(n19, 14),
        new Edge(n27, 10),

};
n23.adjacencies = new Edge[]{
        new Edge(n19, 14),
        new Edge(n20, 10),
        new Edge(n21, 14),
        new Edge(n24, 10),
        new Edge(n29, 14),
        new Edge(n28, 10),
};
n24.adjacencies = new Edge[]{
        new Edge(n20, 14),
        new Edge(n21, 10),
        new Edge(n29, 10),
        new Edge(n28, 14),
        new Edge(n23, 10),
};
n25.adjacencies = new Edge[]{
        new Edge(n26, 10),
        new Edge(n31, 14),
        new Edge(n30, 10),
};
n26.adjacencies = new Edge[]{
        new Edge(n25, 10),
        new Edge(n27, 10),
        new Edge(n32, 14),
        new Edge(n31, 10),
        new Edge(n30, 14),
};
n27.adjacencies = new Edge[]{
        new Edge(n26, 10),
```

```
        new Edge(n33, 14),
        new Edge(n32, 10),
        new Edge(n31, 14),
};
n28.adjacencies = new Edge[]{
        new Edge(n23, 10),
        new Edge(n24, 14),
        new Edge(n29, 10),
        new Edge(n35, 14),
        new Edge(n34, 10),
        new Edge(n33, 14),
};
n29.adjacencies = new Edge[]{
        new Edge(n28, 10),
        new Edge(n23, 14),
        new Edge(n24, 10),
        new Edge(n35, 10),
        new Edge(n34, 14),
};
n30.adjacencies = new Edge[]{
        new Edge(n25, 10),
        new Edge(n26, 14),
        new Edge(n31, 10),
};
n31.adjacencies = new Edge[]{
        new Edge(n30, 10),
        new Edge(n25, 14),
        new Edge(n26, 10),
        new Edge(n27, 14),
        new Edge(n32, 10),
};
n32.adjacencies = new Edge[]{
        new Edge(n31, 10),
        new Edge(n26, 14),
        new Edge(n27, 10),
        new Edge(n33, 10),
};
n33.adjacencies = new Edge[]{
        new Edge(n32, 10),
```

```
            new Edge(n27, 14),
            new Edge(n28, 14),
            new Edge(n34, 10),
        };
        n34.adjacencies = new Edge[]{
            new Edge(n33, 10),
            new Edge(n28, 10),
            new Edge(n29, 14),
            new Edge(n35, 10),
        };
        n35.adjacencies = new Edge[]{
            new Edge(n34, 10),
            new Edge(n28, 14),
            new Edge(n29, 10),
        };
    AstarSearch(n13, n28);
    List<Node> path = printPath(n28);
        System.out.println("Path: " + path);}
    public static List<Node> printPath(Node target) {
        List<Node> path = new ArrayList<Node>();
        for (Node node = target; node != null; node = node.parent) {
            path.add(node);}
        Collections.reverse(path);
        return path;}
    public static void AstarSearch(Node source, Node goal) {
        Set<Node> explored = new HashSet<Node>();
        PriorityQueue<Node> queue = new PriorityQueue<Node>(30,new
Comparator<Node>() {
            //override compare method
            public int compare(Node i, Node j) {
                if (i.f_scores > j.f_scores) {
                    return 1;
                } else if (i.f_scores < j.f_scores) {
                    return -1;
                } else {
                    return 0;
                }}});
        //cost from start
        source.g_scores = 0;
```

```
        queue.add(source);
        boolean found = false;
        while ((!queue.isEmpty()) && (!found)) {
            //the node in having the lowest f_score value
            Node current = queue.poll();
            explored.add(current);
            //goal found
            if (current.value.equals(goal.value)) {
                found = true;}
            //check every child of current node
            for (Edge e : current.adjacencies) {
                Node child = e.target;
                double cost = e.cost;
                double temp_g_scores = current.g_scores + cost;
                double temp_f_scores = temp_g_scores + child.h_scores;
  /*if child node has been evaluated and
    the newer f_score is higher, skip*/
                if ((explored.contains(child)) &&
                        (temp_f_scores >= child.f_scores)) {
                    continue;}
                else if ((!queue.contains(child)) ||
                        (temp_f_scores < child.f_scores)) {
                    child.parent = current;
                    child.g_scores = temp_g_scores;
                    child.f_scores = temp_f_scores;
                    if (queue.contains(child)) {
                        queue.remove(child);}
                    queue.add(child);
                }}}}}
class Node {
    public final String value;
    public double g_scores;
    public final double h_scores;
    public double f_scores = 0;
    public Edge[] adjacencies;
    public Node parent;
    public Node(String val, double hVal) {
        value = val;
        h_scores = hVal;}
```

```
    public String toString() {
        return value;
    }}
class Edge {
    public final double cost;
    public final Node target;
    public Edge(Node targetNode, double costVal) {
        target = targetNode;
        cost = costVal;
    }}
```

Ігрове поле може виглядати по-різному. Представимо таке ігрове поле (рис. 6.29):

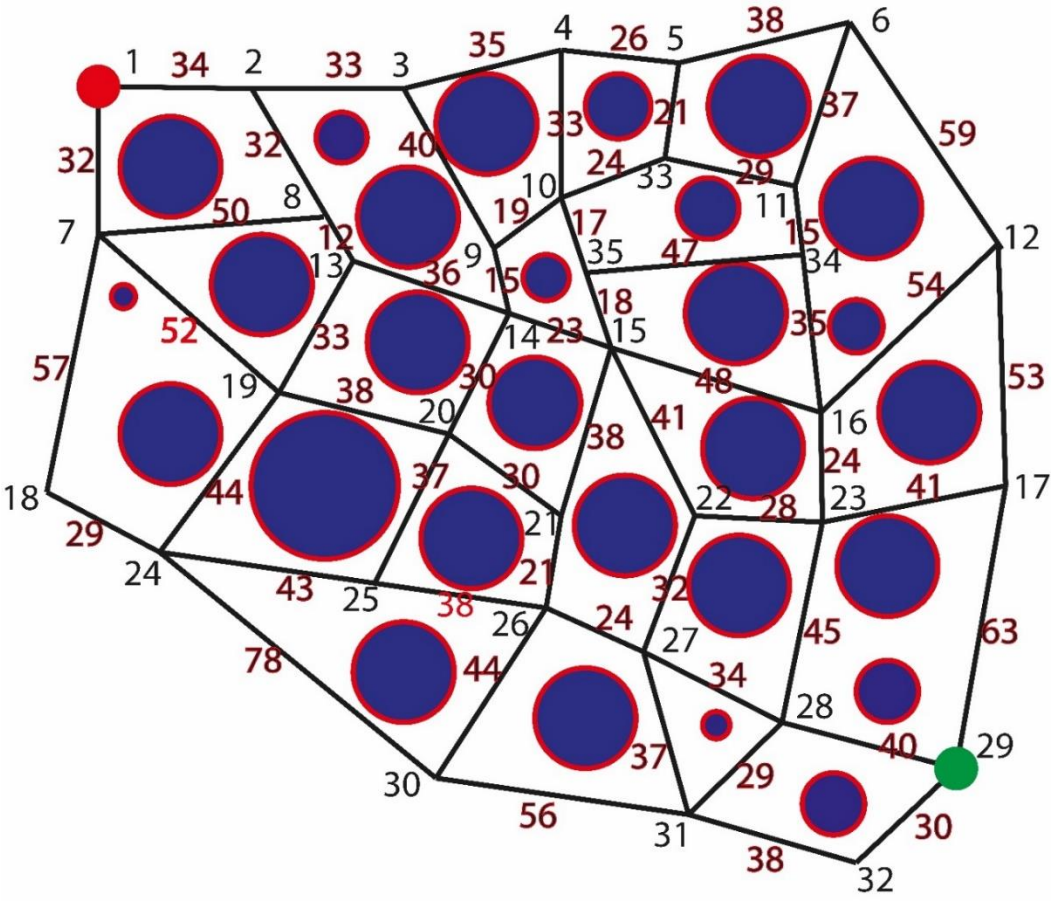


Рис. 6.29. Ігрове поле, сформоване із кругів різного діаметру

Прокладання оптимального маршруту між будь-якими двома вершинами у цьому ігровому полі доцільно здійснити, використовуючи A-star алгоритму (Лістинг 6.9). Для прикладу розглянемо прокладання маршруту між виділеними вершинами 1 та 29. Програмний варіант прокладання маршруту виглядає так:

Лістинг 6.9

```
import java.util.*;
public class AstarSearchAlgo {
    // евристична відстань h являє собою відстані по прямій між
    // всіма вершинами графа та фінішною вершиною 29
    public static void main(String[] args) {
        //створення об'єктів – вузлів графа
        Node n1 = new Node("1", 239);
        Node n2 = new Node("2", 213);
        Node n3 = new Node("3", 191);
        Node n4 = new Node("4", 179);
        Node n5 = new Node("5", 166);
        Node n6 = new Node("6", 165);
        Node n7 = new Node("7", 221);
        Node n8 = new Node("8", 184);
        Node n9 = new Node("9", 153);
        Node n10 = new Node("10", 152);
        Node n11 = new Node("11", 133);
        Node n12 = new Node("12", 115);
        Node n13 = new Node("13", 172);
        Node n14 = new Node("14", 140);
        Node n15 = new Node("15", 119);
        Node n16 = new Node("16", 84);
        Node n17 = new Node("17", 63);
        Node n18 = new Node("18", 207);
        Node n19 = new Node("19", 169);
        Node n20 = new Node("20", 133);
        Node n21 = new Node("21", 103);
        Node n22 = new Node("22", 79);
        Node n23 = new Node("23", 62);
        Node n24 = new Node("24", 180);
        Node n25 = new Node("25", 134);
        Node n26 = new Node("26", 97);
        Node n27 = new Node("27", 73);
        Node n28 = new Node("28", 39);
        Node n29 = new Node("29", 0);
        Node n30 = new Node("30", 113);
        Node n31 = new Node("31", 59);
        Node n32 = new Node("32", 31);
```

```
Node n33 = new Node("33", 148);
Node n34 = new Node("34", 118);
Node n35 = new Node("35", 135);
//створення об'єктів – ребер графа
n1.adjacencies = new Edge[]{
        new Edge(n2, 41),
        new Edge(n7, 32),
};
n2.adjacencies = new Edge[]{
        new Edge(n1, 34),
        new Edge(n3, 33),
        new Edge(n8, 32) };
n3.adjacencies = new Edge[]{
        new Edge(n2, 33),
        new Edge(n4, 35),
        new Edge(n9, 40),
};
n4.adjacencies = new Edge[]{
        new Edge(n3, 35),
        new Edge(n5,26),
        new Edge(n10, 33),
};
n5.adjacencies = new Edge[]{
        new Edge(n4, 26),
        new Edge(n6, 38),
        new Edge(n33, 21),
};
n6.adjacencies = new Edge[]{
        new Edge(n5, 38),
        new Edge(n12, 59),
        new Edge(n11, 37),
};
n7.adjacencies = new Edge[]{
        new Edge(n1, 32),
        new Edge(n8, 50),
        new Edge(n19, 52),
        new Edge(n18, 57),
};
n8.adjacencies = new Edge[]{
```

```
        new Edge(n7, 50),
        new Edge(n2, 32),
        new Edge(n13, 12),
};
 n9.adjacencies = new Edge[]{
        new Edge(n3, 40),
        new Edge(n10, 19),
        new Edge(n14, 15),
 };
 n10.adjacencies = new Edge[]{
        new Edge(n9, 19),
        new Edge(n4, 33),
        new Edge(n33, 24),
        new Edge(n35, 17),
 };
 n11.adjacencies = new Edge[]{
        new Edge(n33, 29),
        new Edge(n6, 37),
        new Edge(n34, 15),
 };
 n12.adjacencies = new Edge[]{
        new Edge(n16, 54),
        new Edge(n6, 59),
        new Edge(n17, 53),
 };
 n13.adjacencies = new Edge[]{
        new Edge(n19, 33),
        new Edge(n8, 12),
        new Edge(n14, 36),
 };
 n14.adjacencies = new Edge[]{
        new Edge(n13, 36),
        new Edge(n9, 15),
        new Edge(n15, 23),
        new Edge(n20, 30),
  };
 n15.adjacencies = new Edge[]{
        new Edge(n14, 23),
        new Edge(n35, 18),
```

```
        new Edge(n16, 48),
        new Edge(n22, 41),
        new Edge(n21, 38),
    };
    n16.adjacencies = new Edge[]{
        new Edge(n15, 48),
        new Edge(n34, 35),
        new Edge(n12, 54),
        new Edge(n23, 24),
    };
    n17.adjacencies = new Edge[]{
        new Edge(n23, 41),
        new Edge(n12, 53),
        new Edge(n29, 63),  };
    n18.adjacencies = new Edge[]{
        new Edge(n7, 57),
        new Edge(n24, 29),
    };
    n19.adjacencies = new Edge[]{
        new Edge(n7, 52),
        new Edge(n13, 33),
        new Edge(n20, 38),
        new Edge(n24, 44),
    };
    n20.adjacencies = new Edge[]{
        new Edge(n19, 38),
        new Edge(n14, 30),
        new Edge(n21, 30),
        new Edge(n25, 37),
    };
    n21.adjacencies = new Edge[]{
        new Edge(n20, 30),
        new Edge(n15, 38),
        new Edge(n26, 21),
    };
    n22.adjacencies = new Edge[]{
        new Edge(n15, 41),
        new Edge(n23, 28),
        new Edge(n27, 32),  };
```

```
n23.adjacencies = new Edge[]{
        new Edge(n22, 28),
        new Edge(n16, 24),
        new Edge(n17, 41),
        new Edge(n28, 45),
};
n24.adjacencies = new Edge[]{
        new Edge(n18, 29),
        new Edge(n19, 44),
        new Edge(n25, 43),
        new Edge(n30, 78),
};
n25.adjacencies = new Edge[]{
        new Edge(n24, 43),
        new Edge(n20, 37),
        new Edge(n26, 38),
 };
n26.adjacencies = new Edge[]{
        new Edge(n25, 38),
        new Edge(n21, 21),
        new Edge(n27, 24),
        new Edge(n30, 44),
};
n27.adjacencies = new Edge[]{
        new Edge(n26, 24),
        new Edge(n22, 32),
        new Edge(n28, 34),
        new Edge(n31, 37),
};
n28.adjacencies = new Edge[]{
        new Edge(n27, 34),
        new Edge(n23, 45),
        new Edge(n29, 40),
        new Edge(n31, 29),
};
n29.adjacencies = new Edge[]{
        new Edge(n28, 40),
        new Edge(n17, 63),
        new Edge(n32, 38),
```

```
    };
    n30.adjacencies = new Edge[]{
            new Edge(n24, 78),
            new Edge(n26, 44),
            new Edge(n31, 56),
    };
    n31.adjacencies = new Edge[]{
            new Edge(n30, 56),
            new Edge(n27, 37),
            new Edge(n28, 29),
            new Edge(n32, 38),
    };
    n32.adjacencies = new Edge[]{
            new Edge(n31, 38),
            new Edge(n29, 30),
    };
    n33.adjacencies = new Edge[]{
            new Edge(n10, 24),
            new Edge(n5, 21),
            new Edge(n11, 29),
    };
    n34.adjacencies = new Edge[]{
            new Edge(n35, 47),
            new Edge(n11, 15),
            new Edge(n16, 35),
    };
    n35.adjacencies = new Edge[]{
            new Edge(n10, 17),
            new Edge(n34, 47),
            new Edge(n15, 18),
    };
    AstarSearch(n1, n29);
    List<Node> path = printPath(n29);
    System.out.println("Path: " + path);}
public static List<Node> printPath(Node target) {
    List<Node> path = new ArrayList<Node>();
    for (Node node = target; node != null; node = node.parent) {
       path.add(node);}
    Collections.reverse(path);
```

```
        return path;}
    public static void AstarSearch(Node source, Node goal) {
        Set<Node> explored = new HashSet<Node>();
        PriorityQueue<Node> queue = new PriorityQueue<Node>(40,new
Comparator<Node>() {
            //реалізація методу порівняння
            public int compare(Node i, Node j) {
                if (i.f_scores > j.f_scores) {
                    return 1;
                } else if (i.f_scores < j.f_scores) {
                    return -1;
                } else {
                    return 0;
                }}});
        //вага на старті
        source.g_scores = 0;
        queue.add(source);
        boolean found = false;
        while ((!queue.isEmpty()) && (!found)) {
            /цей вузол має найнижчу величину f_score
            Node current = queue.poll();
            explored.add(current);
            //мета знайдена
            if (current.value.equals(goal.value)) {
                found = true;}
            //перевірка кожного інцидентного ребра поточного вузла
            for (Edge e : current.adjacencies) {
                Node child = e.target;
                double cost = e.cost;
                double temp_g_scores = current.g_scores + cost;
                double temp_f_scores = temp_g_scores + child.h_scores;
                //якщо інцидентний вузол оцінений і
                // оновлена f_score є більшою, то здійснюємо перехід
                if ((explored.contains(child)) &&
                        (temp_f_scores >= child.f_scores)) {
                    continue;}
                else if ((!queue.contains(child)) ||
                        (temp_f_scores < child.f_scores)) {
                    child.parent = current;
```

```
                child.g_scores = temp_g_scores;
                child.f_scores = temp_f_scores;
                if (queue.contains(child)) {
                   queue.remove(child);}
                queue.add(child);
             }}}}}
class Node {
   public final String value;
   public double g_scores;
   public final double h_scores;
   public double f_scores = 0;
   public Edge[] adjacencies;
   public Node parent;
   public Node(String val, double hVal) {
      value = val;
      h_scores = hVal;}
   public String toString() {
      return value;
   }}
class Edge {
   public final double cost;
   public final Node target;
   public Edge(Node targetNode, double costVal) {
      target = targetNode;
      cost = costVal;
   }
}
```

Результат роботи програми виглядає так. Path: [1, 7, 19, 20, 21, 26, 27, 28, 29]. Це, власне кажучи, і є оптимальний маршрут.

Цю програму, як і інші, представлені у посібнику, можна просто скопіювати та інсталювати у програмне поле мови програмування Java. Після цього програма запускається та може використовуватись як базова модель досліджуваної проблеми.

СПИСОК ЛІТЕРАТУРИ

1. Томашевський В. М. Моделювання систем: навч. посіб. Київ: Видавнича група BHV, 2015. 349 с.

2. Жерновий Ю. В. Імітаційне моделювання систем масового обслуговування: практикум. Львів: Видавничий центр ЛНУ імені Івана Франка, 2017. 307 с.

3. Задачин В. М., Конюшенко І. Г. Лабораторний практикум з навчальної дисципліни «Моделювання систем»: навч.-практ. посіб. Харків: Вид. ХНЕУ, 2019. 212 с.

4. Ситник В. Ф., Орленко Н. С. Імітаційне моделювання: навч. посіб. Київ: КНЕУ. 2014. 230 с.

5. Стеценко І. В., Батора Ю. В. Інформаційна технологія визначення оптимальних параметрів управління транспортним рухом через світлофорні об'єкти міста. *Математичні машини і системи*. Київ, 2007. № 3, 4. С. 211–217.

6. Стеценко І. В. Бойко О. В. Технологія імітаційного моделювання систем управління засобами сіток Петрі. *Вісник Черкаського державного технологічного університету.* Черкаси, 2016. № 4. С. 29–32.

7. Стеценко І. В., Бойко О. В. Система імітаційного моделювання засобами сіток Петрі. *Математичні машини і системи*. Київ, 2009. № 1. С. 117–124.

8. Стеценко І. В., Данилюк А. А. Імітаційне моделювання систем управління засобами сіток Петрі. *Вісник Черкаського державного технологічного університету.* Черкаси, 2015. № 3. С. 293–295.

9. Стеценко І. В., Стеценко В. Г., Дифучин Ю. М. Оптимізація імітаційних моделей систем методами групового врахування аргументів. *Питання прикладної математики та математичного моделювання*. Видавництво Дніпропетровського університету, 2004. С. 172–177.

10. Теорія статистики: навч. посіб. / П. Г. Вашків, П. І. Пастер, В. П. Сторожук, Є. І. Ткач. Київ: Либідь, 2001. 320 с.

11. Тимченко А. А. Основи системного проектування та системного аналізу складних об'єктів: підручник для студентів вищих закладів освіти / за ред. В. І. Бикова. Київ: Либідь, 2010. 270 с.

12. Тимченко А. А. Основи системного проєктування та системного аналізу об'єктів: навч. посіб. / за ред. Ю. Г. Леги. Київ: Либідь, 2004. 288 с.

13. Томашевський В. М., Жданова О. Г., Жолдакова О. О. Вирішення практичних завдань методами комп'ютерного моделювання: навч. посіб. Київ: Корнійчук, 2019. 214 с.

14. Ямпольський Л. С., Лавров О. А. Штучний інтелект у плануванні та управлінні виробництвом. Київ: Вища школа, 2013. 254 с.

15. Kelton W. D., Sadowski R. P., Sadowski D. A. Simulation with Arena, New York: McGraw–Hill. 2014.

16. Systems Modeling Corporation: Arena User's Guide, Version 4.0, Sewickly, Pennsylvania. 2013.

17. Васильєв В. В., Кузьмук В. В. Мережі Петрі, паралельні алгоритми та моделі мультипроцесорних систем. Київ: Наук. думка, 1990. 212 с.

18. Горбачов В. А. Моделювання систем: навч. посіб. Київ: ІСДО, 1996. 120 с.

19. Моделювання та оптимізація систем: підручник / В. М. Дубовой, Р. Н. Квєтний, О. І. Михальов, А. В. Усов. Вінниця, ВНТУ, 2017. 803 с.

20. Комп'ютерне моделювання систем та процесів. Методи обчислень. Частина 1: навч. посіб. / Р. Н. Квєтний, І. В. Богач, О. Р. Бойко та ін. Вінниця, ВНТУ, 2013. 190 с.

21. Комп'ютерне моделювання систем та процесів. Методи обчислень. Частина 2: навч. посіб. / Р. Н. Квєтний, І. В. Богач, О. Р. Бойко та ін. Вінниця, ВНТУ, 2013. 234 с.

Навчальне видання

Ніколюк *Петро Карпович*

МОДЕЛЮВАННЯ СИСТЕМ

Навчальний посібник

Редактор О. А. Солдатова
Технічний редактор Т. О. Важеніна-Гопрак

Підписано до друку 26.12.2023.
Формат 60×84/16. Папір офсетний.
Друк – цифровий. Умовн. друк. арк. 13,25.
Тираж 30. Зам. 85.

Донецький національний університет імені Василя Стуса
21021, м. Вінниця, 600-річчя, 21
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру
серія ДК № 5945 від 15.01.2018