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

П. К. Ніколюк

МОДЕЛЮВАННЯ СИСТЕМ

Методичні вказівки для самостійної роботи
та виконання лабораторних робіт
здобувачами спеціальності 122 Комп’ютерні науки
освітньої програми «Комп’ютерні науки»

Вінниця
2024

УДК 378.147.091.31.22:004](075.8)
Н 643

Рекомендовано до друку вченою радою факультету інформаційних і прикладних технологій ДонНУ імені Василя Стуса (протокол № 8 від 20 березня 2024 р.)

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

Рецензенти: *Крижановський В. Г.,* д-р техн. наук, професор, професор кафедри прикладної математики та кібербезпеки ДонНУ імені Василя Стуса.

Ніколюк П. К.

Н 643 Методичні вказівки для самостійної роботи та виконання лабораторних робіт здобувачами спеціальності 122 Комп’ютерні науки освітньої програми «Комп’ютерні науки». Вінниця: ДонНУ імені Василя Стуса, 2024. 84 с.

Методичні вказівки є навчально-методичним документом, який містить рекомендації для отримання навичок оптимального розв’язання прикладних задач під час виконання лабораторного практикуму з дисципліни «Системний аналіз та моделювання систем».

Для здобувачів ОС «Бакалавр» спеціальності 122 Комп’ютерні науки факультету інформаційних і прикладних технологій ДонНУ імені Василя Стуса.

УДК 378.147.091.31.22:004](075.8)

© Ніколюк П. К., 2024
© ДонНУ імені Василя Стуса, 2024

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 моделювання дасть позитивні результати для використання їх у роботі об'єкта (системи). Із сказаного випливає, що до процедури моделювання систем треба ставитись дуже відповідально: якщо модель не задовольняє необхідні вимоги, 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. Вона 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.

ЛАБОРАТОРНА РОБОТА 1
МОДЕЛЬ ІНТЕЛЕКТУАЛЬНОГО МІСЬКОГО ПЕРЕХРЕСТЯ – ПРОГРАМА ANYLOGIC NORTH AMERICA LLC

Мета роботи: ознайомлення з особливостями програми візуального імітаційного моделювання AnyLogic North America LLC; виконати побудову і запуск моделі інтелектуального регульованого міського перехрестя.

Теоретичні відомості. 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. 1.1).

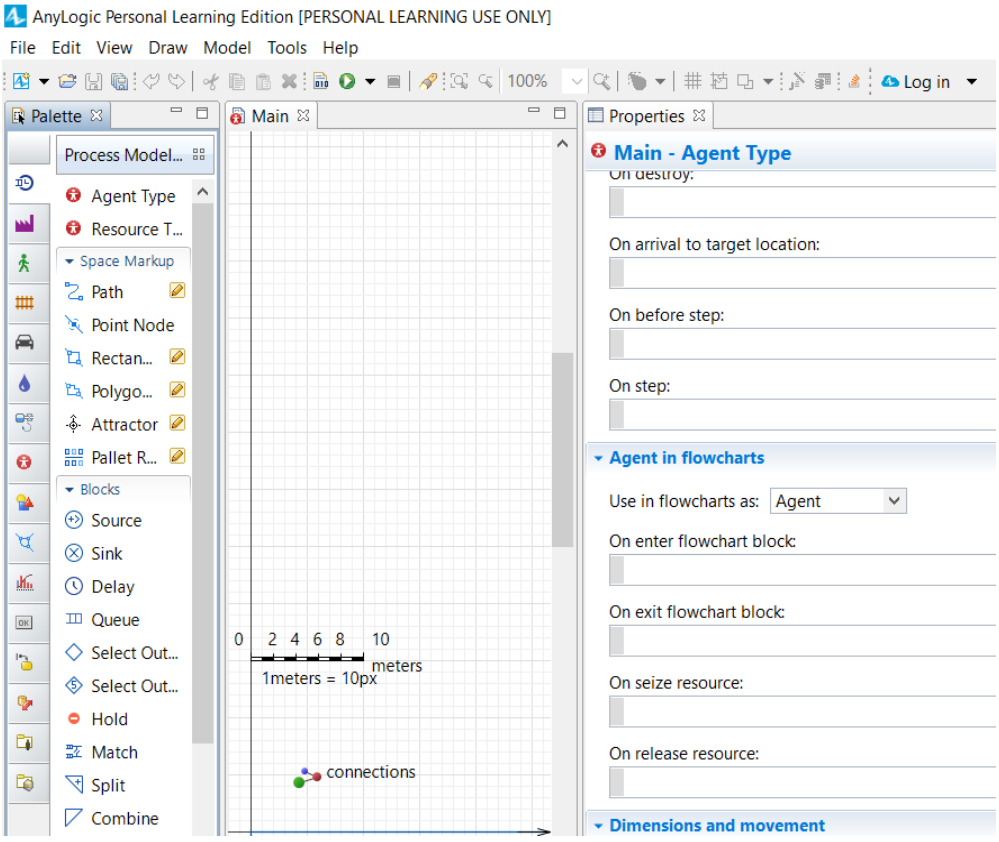


Рис. 1.1. Інтерфейс програми AnyLogic North America LLC

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. 1.2) і натиснувши Model.

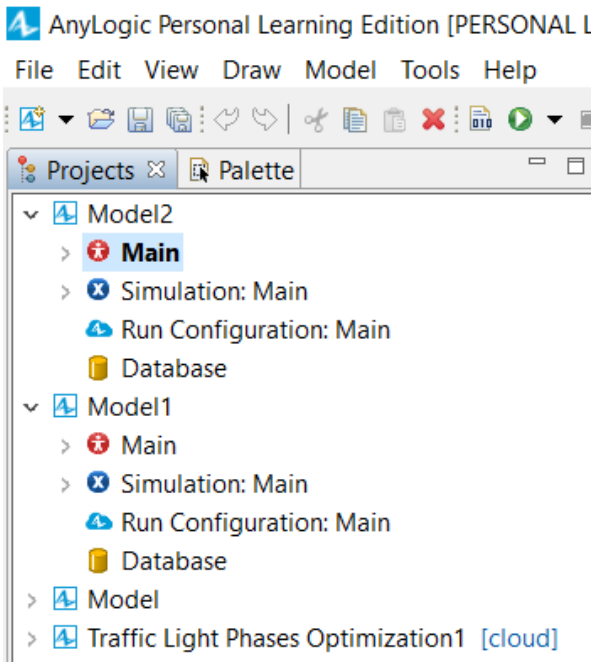


Рис. 1.2. Створення нової моделі

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. 1.3).

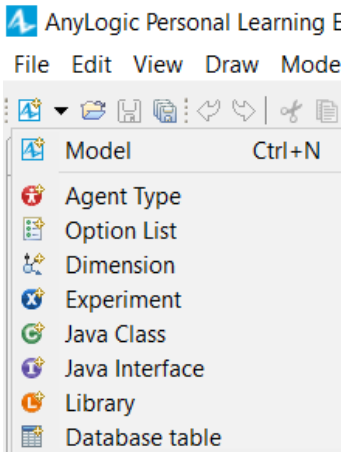


Рис. 1.3. Панель Projects. Представлено перелік назв створених проєктів

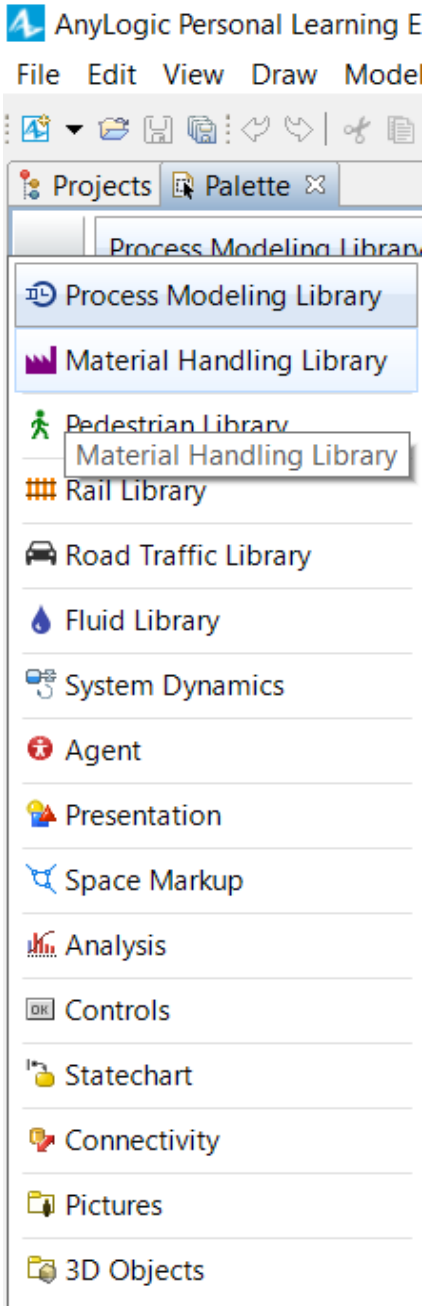


Рис. 1.4. Перелік вкладок панелі Palette

- Панель Palette (палітри) складається з декількох вкладок (палітр), кожна з яких містить елементи, що належать до певної задачі (рис. 1.4): 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.
- Material Handling Library (бібліотека обробки матеріалів) спрощує моделювання складних виробничих систем і операцій. Її використовують для розробки детальних моделей виробничих і складських приміщень та 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 відповідність 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 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 і можуть анімувати їх діяльність. Щоб анімувати агентів, які знаходяться у 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. 1.5.

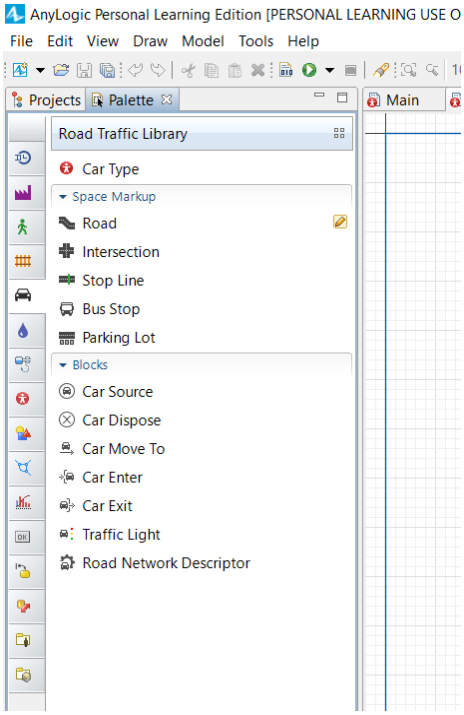


Рис. 1.5. Конструктори дорожньої бібліотеки

Виконання роботи. 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. 1.6). Вверху є віконце з масштабом: реалізована можливість змінити розміри, як вам потрібно.

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, розташованій 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.

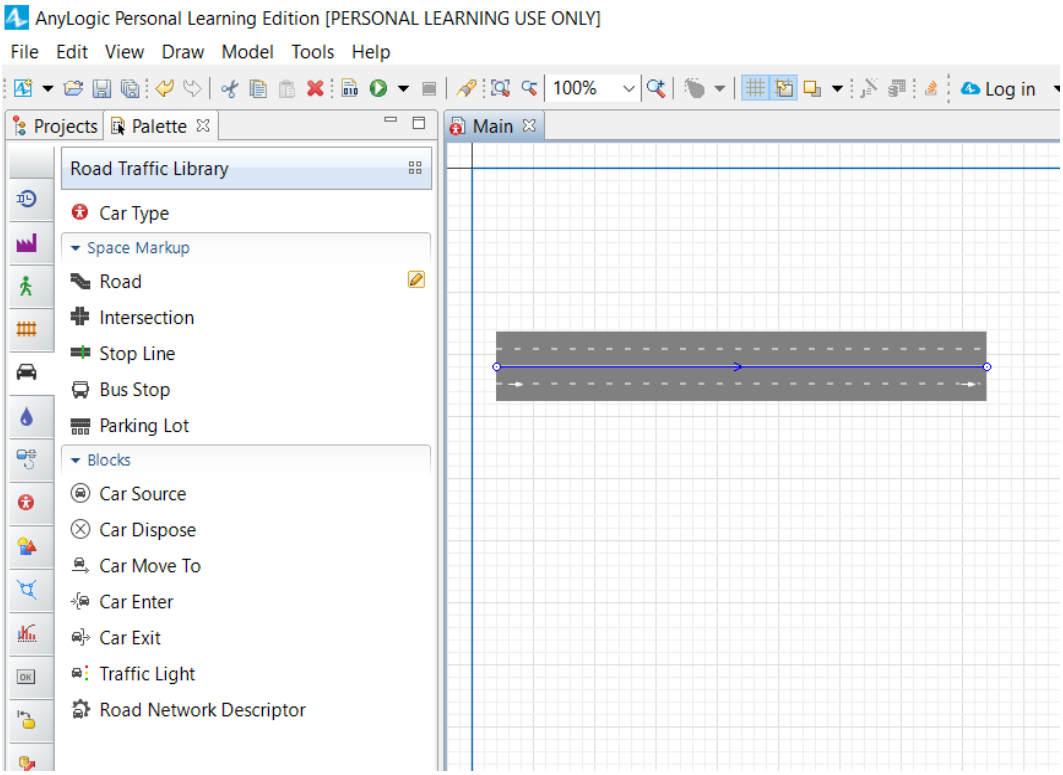


Рис. 1.6. Модель прямолінійної ділянки дороги

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. 1.7. 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.

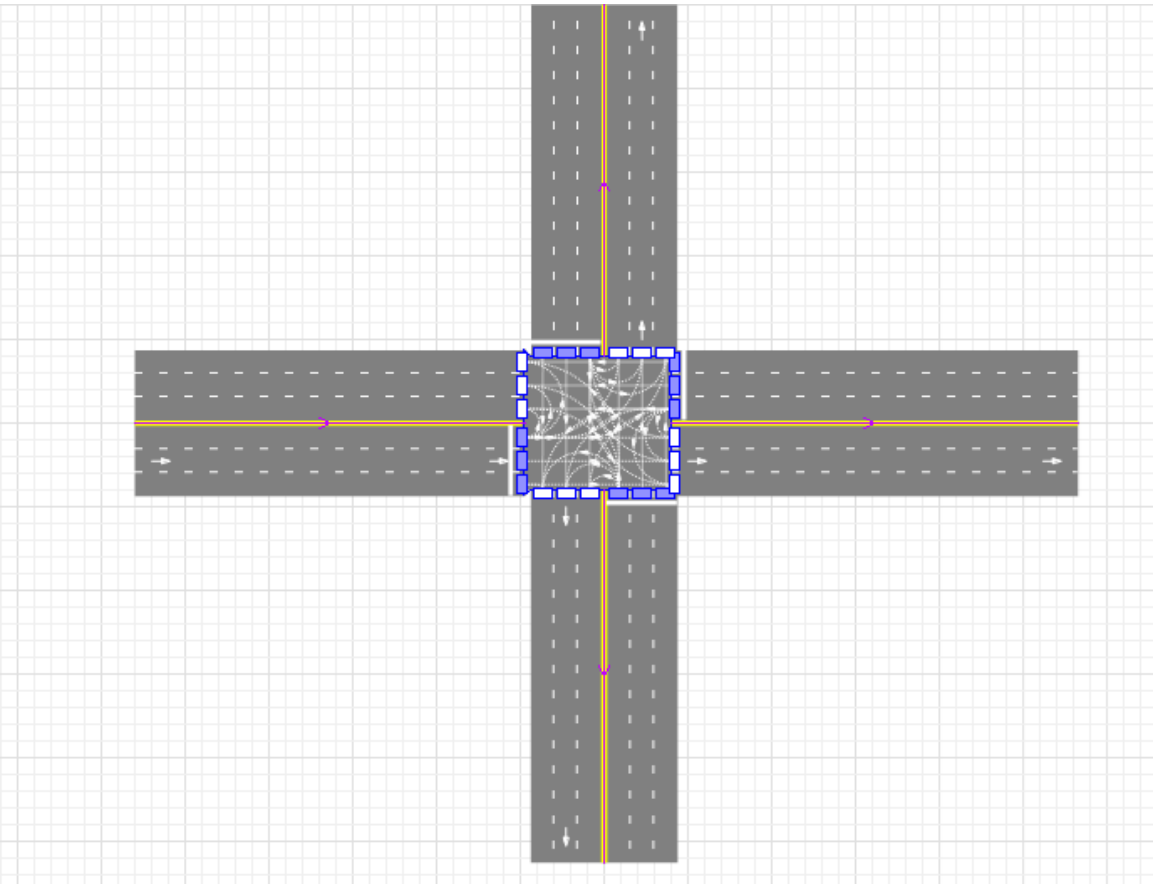


Рис. 1.7. Модель хрестоподібного перехрестя

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 цього 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. 1.8).

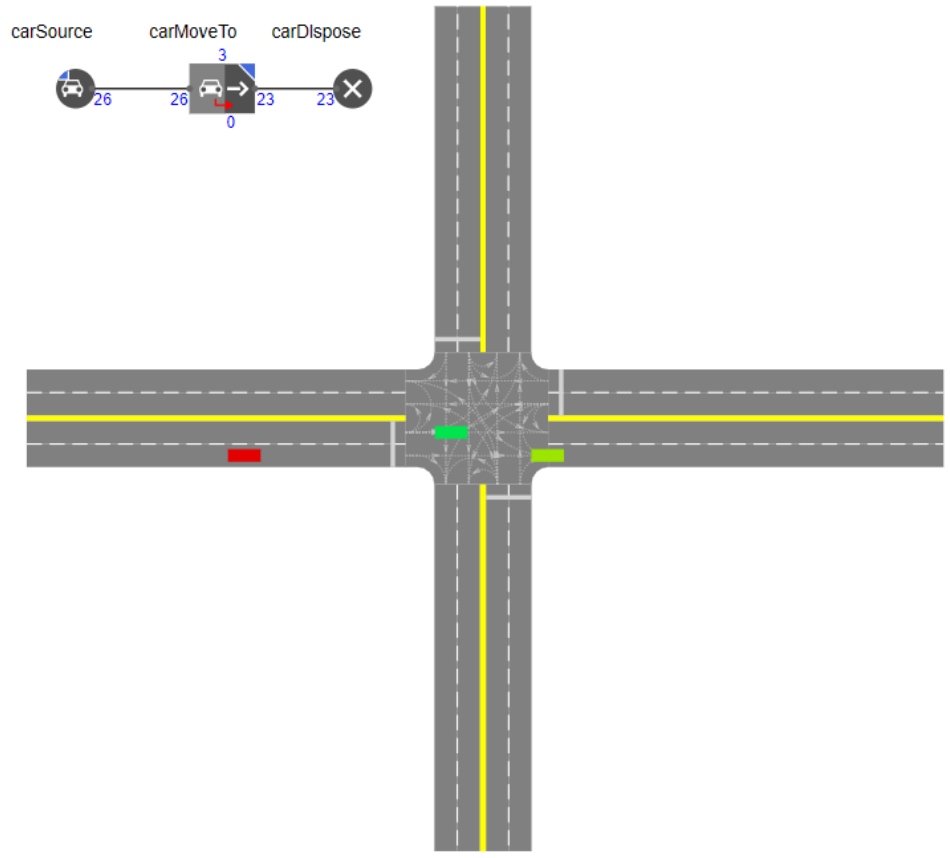


Рис. 1.8. Логіка динамічної моделі під час руху автомобілів зліва направо

Вгорі зліва представлена логіка руху: показано кількість автомобілів, що генеровані carSource. На іконці carMoveTo кількість 23 означає кількість транс-

портних 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. 1.9). Для цього створюємо послідовність carSourse1→ carMoveTo1→carDispose1. 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.

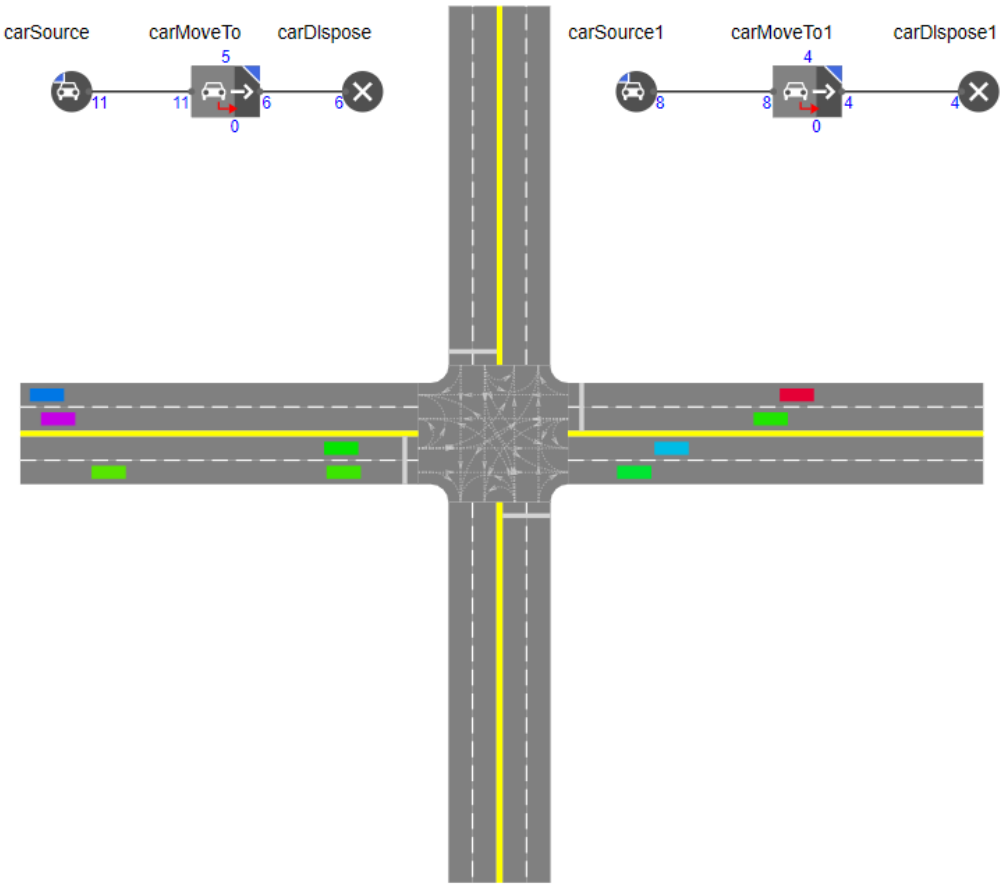


Рис. 1.9. Модель руху автомобілів у горизонтальному напрямку

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. 1.10): з кожного із чотирьох напрямків транспортний засіб має змогу рухатись у чотирьох напрямках – прямо, наліво, направо та розворот у зворотному напрямку.

Отже, остаточно отримуємо варіант моделі, представлений на рис. 1.11. Фінальна модель інтелектуального регульованого перехрестя створена так, щоб синхронізувати протяжність горіння зелених фаз світлофору на конкретному. Якщо

всі регульовані у місті світлофори перейдуть у такий режим, то міський трафік підніметься на якісно новий рівень. Це фактично означає оптимізацію режиму світлофорного регулювання.

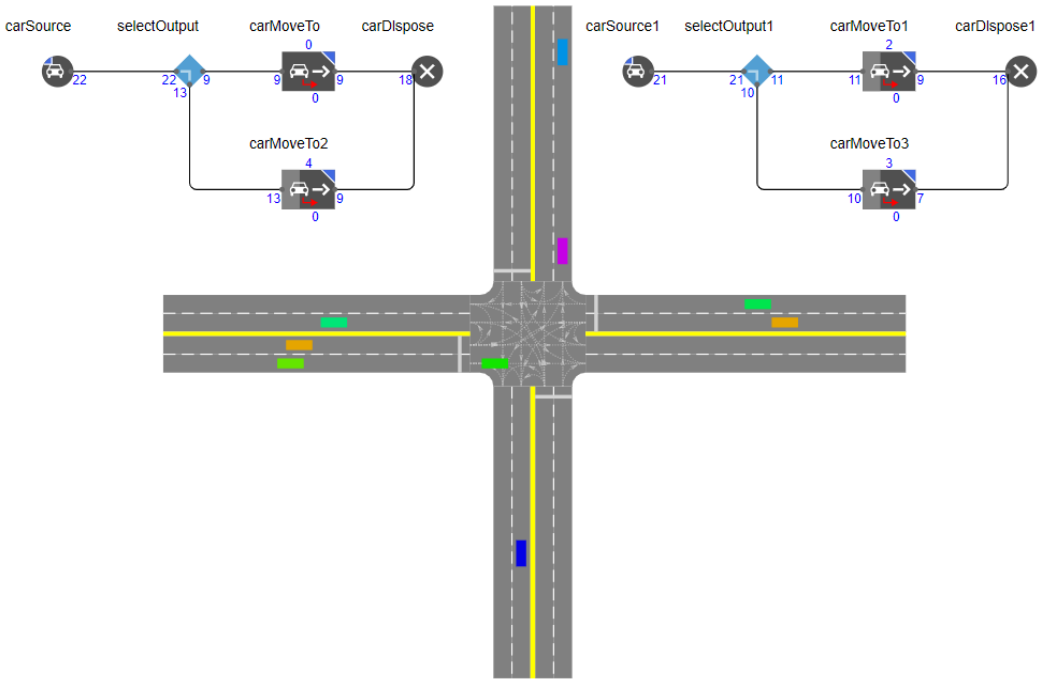


Рис. 1.10. Модель руху автомобілів у двох напрямках

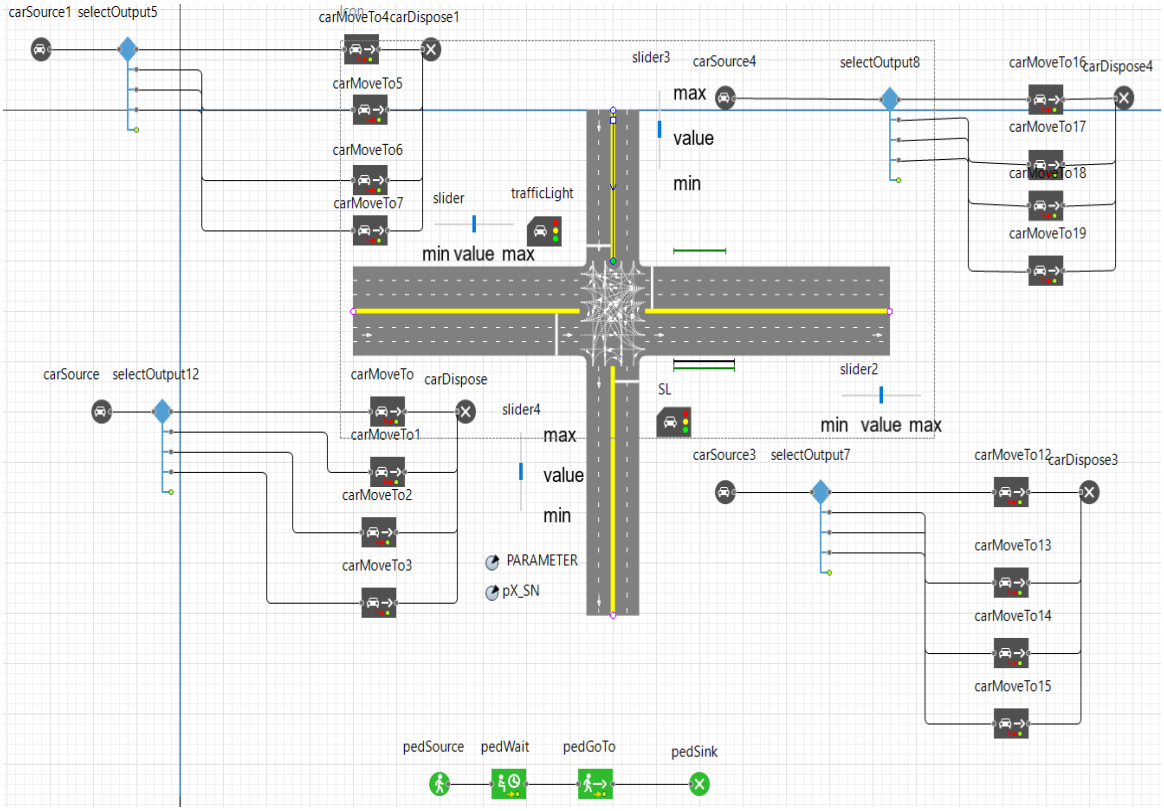


Рис. 1.11. Фінальна модель інтелектуального перехрестя. Зображене перехрестя та діаграми логіки. Показані також світлофори, що регулюють потоки транспортних засобів

Завдання. Створити модель інтелектуального світлофорного регулювання потоків транспортних засобів через хрестоподібне міське перехрестя.

ЛАБОРАТОРНА РОБОТА 2
МОДЕЛЮВАННЯ ПОЛЬОТУ БПЛА: СИМУЛЯТОР DJI

Мета роботи: ознайомитися з особливостями симуляції польотів на БПЛА; проаналізувати різні варіанти польотів; виробити навички керування БПЛА в умовах, наближених до реальних.

Теоретичні відомості. Опишемо загальні кроки, які можуть бути типовими для більшості симуляторів дронів, включно з тими, що розроблені компанією DJI.

1. Встановлення програми: зазвичай ви маєте завантажити та встановити програмне забезпечення симулятора на ваш комп'ютер або мобільний пристрій.

2. Підключення контролера: підключіть контролер дрона до вашого комп'ютера або мобільного пристрою. Деякі симулятори можуть підтримувати бездротові з'єднання через Wi-Fi або Bluetooth.

3. Вибір дрона та середовища: оберіть модель дрона, яку ви хочете симулювати, і виберіть середовище, в якому ви будете літати (наприклад, міська місцевість, ліс або пустеля).

4. Навчання управління: після запуску симулятора ви зможете навчатися керувати дроном за допомогою вашого контролера; треба ретельно дотримуватися інструкцій, щоб отримати розуміння процесу керування.

5. Виконання різних завдань: багато симуляторів надають різноманітні завдання, як-от обліт певних точок, посадка на визначений майданчик або навіть моделювання певних сценаріїв аварій та їх розбір.

6. Налаштування параметрів: деякі симулятори дають змогу змінювати параметри дрона, як-от швидкість, чутливість контролів та інші характеристики, щоб краще відповідати вашим потребам або вирішити конкретні проблеми.

7. Експериментування та вдосконалення навичок: використовуйте симулятор для експериментів та вдосконалення вашого вміння керувати дроном у різних умовах та ситуаціях.

8. Оцінка результатів: деякі симулятори надають звіти або оцінки вашої продуктивності, які можуть допомогти вам визначити ваші сильні та слабкі сторони і покращити навички керування.

Зауважте, що точні кроки та можливості можуть відрізнятися залежно від конкретного симулятора, тому завжди рекомендується ознайомлюватися з документацією користувача або відеоуроками, наданими розробниками програмного забезпечення.

Детальну інструкцію з користування симулятором DJI можна знайти за посиланням: https://i.citrus.world/uploads/files/dji-mini-2-user-manual-uapdf.pdf

Перед початком виконання роботи необхідно ознайомитись з інструкцією.

Звернемо увагу на особливості симулятора DJI. Симулятор DJI є потужним інструментом для тренування та навчання пілотів дронів. Ось деякі особливості, які можуть бути характерними для симулятора DJI.

1. Моделювання реальних дронів DJI: симулятор DJI має можливість моделювати реальні дрони компанії DJI, як-от DJI Phantom, Mavic, Inspire тощо. Це дає змогу пілотам відчути реалістичність керування та характеристики польоту своїх конкретних моделей дронів.

2. Різноманітність сценаріїв: симулятор може містити різні сценарії для тренування, як-от політ у різних погодних умовах, у різних локаціях або навіть у симуляції аварійних ситуацій.

3. Деталізоване середовище: симулятор може включати детально відтворені території та ландшафти, що дають змогу пілотам випробувати свої навички в різних умовах.

4. Реалістична фізика польоту: однією з важливих особливостей симулятора DJI може бути реалістична модель фізики польоту, яка відтворює реальність керування дроном.

5. Модульність та розширені можливості: симулятор може мати можливість розширення та модифікації через додаткові модулі або розширення, що дає змогу користувачам адаптувати його до своїх конкретних потреб та цілей.

6. Інтерактивні навчальні матеріали: в симуляторі є навчальні режими та матеріали, які допомагають новачкам ознайомитися з основами керування дроном та розвивати свої навички.

7. Підтримка аксесуарів DJI: симулятор може інтегруватися з різними аксесуарами та додатковими пристроями компанії DJI, що робить тренування більш реалістичним та корисним.

8. Аналіз результатів: симулятор може надавати звіти або аналітичні інструменти, що дають змогу користувачам оцінювати свої успіхи та вдосконалювати навички керування.

Зауважте, що конкретні функції та особливості можуть змінюватися залежно від версії симулятора DJI та його налаштувань. Рекомендується ознайомитися з офіційною документацією та матеріалами навчання, які надаються розробниками для отримання докладної інформації.

Виконання роботи. Для запуску та симуляції польоту на симуляторі DJI можна використовувати загальний алгоритм:

1. Встановлення програмного забезпечення. Запустіть симулятор DJI на вашому комп’ютері або мобільному пристрої (рис. 2.1).

2. Запуск симулятора. Натисніть Start. Відкриється вікно програми (рис. 2.2).

3. Вибір режиму польоту. Можливими режимами польоту є **Skills Training, FreeFlight** та **Entertainment.** Переключення між режимами здійснюється шляхом натискання на клавіатурі клавіш **S + X.**

Рис. 2.1. Інтерфейс симулятора DJI

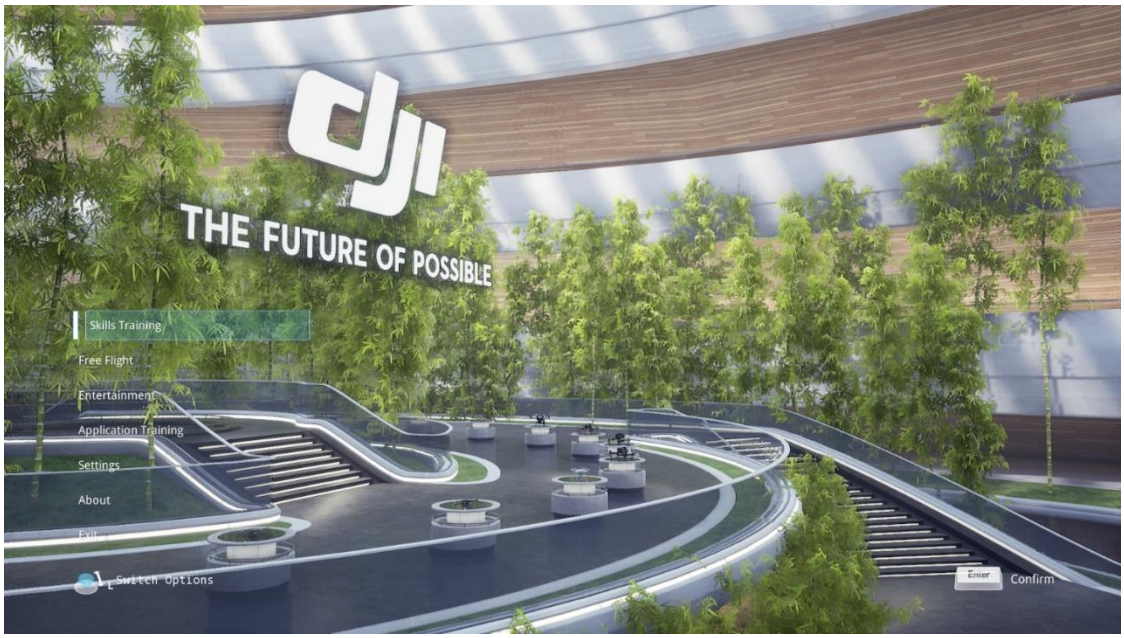


Рис. 2.2. Робоче середовище симулятора

4. Вибір моделі дрона і середовища. Оберіть модель дрона, яку ви хочете симулювати, і виберіть середовище, в якому ви будете літати. Деякі симулятори можуть надавати можливість вибору різних моделей та локацій для польоту.

5. Налаштування параметрів. Залежно від вашого рівня навичок та вимог ви можете налаштувати параметри симуляції, як-от швидкість вітру, погодні умови, чутливість контролів тощо.

6. Початок симуляції. Після налаштування параметрів та вибору моделі дрона та середовища натисніть одночасно чотири кнопки: **A + S** зліва на клавіатурі та **нижню і праву кнопки,** що показані на рис. 2.3 справа:

Рис. 2.3. Стіки для запуску БПЛА за допомогою пульта керування та кнопки запуску БПЛА за допомогою клавіатури

7. Управління дроном. Використовуйте контролер дрона, щоб керувати польотом у симуляторі. Екран симулятора відображатиме зображення з камери дрона та інші важливі дані про політ.

8. Виконання завдань та тренування. Виконуйте різні завдання та тренуйтеся в управлінні дроном, використовуючи симулятор DJI. Завдяки цьому ви зможете поліпшити свої навички та набути досвіду без ризику для реального дрона.

9. Завершення симуляції. Після завершення тренування або виконання завдань завершіть симуляцію та збережіть результати, якщо потрібно.

10. Аналіз та вдосконалення. Оцініть свій політ, проаналізуйте результати та вдосконалюйте свої навички управління дроном для подальших симуляцій або реальних польотів.

Зауважте, що конкретні кроки можуть відрізнятися залежно від версії симулятора DJI та його налаштувань. Рекомендується докладно ознайомитися з інструкціями користувача та навчальними матеріалами, що надаються розробниками, для отримання докладної інформації.

Завдання. Здійснити пілотування БПЛА у різних режимах польоту з використанням симулятора DJI.

ЛАБОРАТОРНА РОБОТА 3
МОДЕЛЮВАННЯ: ПРОГРАМА NETLOGO

Мета роботи: ознайомитися з особливостями візуальної програми моделювання NetLogo та інтегрованого середовища моделювання складних систем.

Теоретичні відомості. NetLogo – це візуальна програмна мова та інтегроване середовище для моделювання складних систем. Ось деякі з основних особливостей програми NetLogo.

1. Візуальне середовище: NetLogo має візуальний інтерфейс, який дає змогу користувачам побудувати та візуалізувати агент-орієнтовані моделі. Це робить його дуже доступним для новачків у моделюванні, а також корисним для досліджень і навчання.

2. Моделювання агентів: однією з основних концепцій NetLogo є моделювання агентів. Користувачі можуть створювати моделі, де окремі агенти (наприклад, тварини, люди або частинки) діють та взаємодіють один з одним та з оточенням.

3. Простота використання: мова програмування NetLogo доволі проста для вивчення та використання. Вона базується на агент-орієнтованому підході, що полегшує реалізацію складних моделей.

4. Моделі зв'язку: NetLogo дає змогу користувачам легко створювати моделі зв'язків, які досліджують взаємодію між різними складниками системи.

5. Підтримка різних типів моделювання: NetLogo підтримує різні типи моделювання, включно з моделями росту, дифузії, вирішенням проблем трафіка та ін.

6. Інтегроване середовище для візуалізації: NetLogo має інтегровану систему візуалізації, яка дає змогу користувачам спостерігати за розвитком своїх моделей у реальному часі.

7. Можливості розширення: NetLogo дає змогу розширювати свої можливості за допомогою розширень і додаткових компонентів, що дає змогу розробникам створювати власні модулі та функції.

Загалом NetLogo є потужним інструментом для моделювання складних систем, який широко використовується в науці, освіті та дослідженнях.

Виконання роботи. Алгоритм створення моделі з допомогою NetLogo – це цікавий і важливий процес, який дає змогу моделювати різноманітні системи та процеси, використовуючи агентні моделі. Нижче подано загальний алгоритм створення моделі за допомогою NetLogo:

1. Розробка концепції моделі:

- Спочатку необхідно визначити, що саме ви хочете моделювати та які агенти будуть брати участь у цій системі.
- Сформулюйте базові правила поведінки агентів і їх взаємодії.

2. Запуск NetLogo:

- Відкрийте NetLogo та створіть новий проєкт. Інтерфейс програми виглядає так (рис. 3.1).

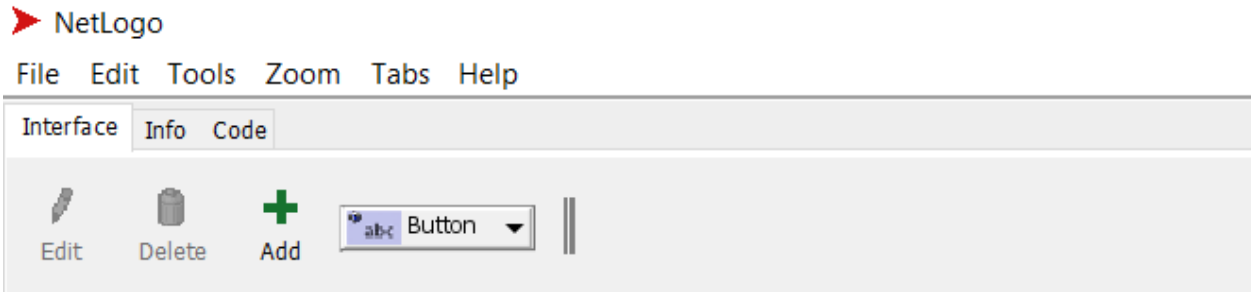


Рис. 3.1. Інтерфейс програми NetLogo

3. Створення агентів:

- Визначте типи агентів, які будуть наявні у вашій моделі (наприклад, люди, тварини, частинки).
- Створіть агентів і задайте їх початкові параметри та властивості.

4. Визначення правил поведінки:

- Додайте правила, які керують поведінкою агентів у вашій моделі.
- Визначте, як агенти реагують на зовнішні події та взаємодіють один з одним.

5. Створення середовища:

- Визначте середовище, у якому будуть існувати агенти.
- Задайте параметри середовища, як-от розмір поля, наявність перешкод, ресурсів тощо.

6. Візуалізація:

- Додайте елементи візуалізації, які дають змогу вам спостерігати за розвитком моделі.
- Розгляньте можливість використання кольорів, форм, ліній тощо для представлення агентів та їх взаємодії.

7. Тестування та налагодження:

- Перевірте модель на відповідність та реалізм.
- Вносьте зміни до параметрів та правил, якщо потрібно, щоб забезпечити бажаний результат.

8. Аналіз результатів:

- Після того, як модель відпрацює, проаналізуйте її результати.
- Вивчіть вплив різних факторів на систему та взаємодію агентів.

9. Документування:

- Документуйте вашу модель, включно з описом її функціональності, основних параметрів, правил поведінки, результатів та висновків.

10. Розповсюдження:

- Поділіться вашою моделлю з іншими користувачами NetLogo або використовуйте її для освітніх або дослідницьких цілей.

Це загальний алгоритм, який можна використовувати для створення моделей за допомогою NetLogo. Залежно від конкретної задачі деякі кроки можуть вимагати більш детального або специфічного підходу.

Завдання. Створити власну модель NetLogo. Продемонструвати роботу моделі.

ЛАБОРАТОРНА РОБОТА 4
ЕКСТРАПОЛЯЦІЯ ЯК ЗАСІБ МОДЕЛЮВАННЯ

Мета роботи: ознайомитися з особливостями різних засобів моделювання і екстраполяції як головного методу моделювання складних систем.

Теоретичні відомості. Моделювання – це процес створення спрощених абстракцій реальних систем або процесів для аналізу їх поведінки, прогнозування подій або розуміння основних закономірностей. Існує кілька різних типів моделювання, які використовуються для вивчення різних типів систем. Ось деякі з них:

1. Дискретне моделювання:

- У цьому типі моделювання час розглядається у вигляді послідовності дискретних моментів.
- Система представляється як набір станів, які змінюються у відповідь на дискретні події або кроки часу.

2. Континуальне моделювання:

- У цьому типі моделювання час розглядається як неперервний.
- Система представляється у вигляді диференціальних рівнянь або інших математичних виразів, які описують зміни у часі.

3. Статистичне моделювання:

- Використовується для аналізу випадкових явищ та подій, враховуючи їх ймовірнісні характеристики.
- Моделі побудовані на основі статистичних методів та імітують випадкові процеси.

4. Агентне моделювання:

- У цьому підході система розглядається як набір окремих агентів, які взаємодіють один з одним та з навколишнім середовищем.
- Агенти мають власні правила поведінки та можуть адаптуватися до змін у середовищі.

5. Системне динамічне моделювання:

- Використовується для аналізу змін у системах з часом, враховуючи їх взаємодію та залежності.
- Моделі побудовані на основі зв’язків між різними елементами системи та їх впливом один на одного.

6. Фізичне моделювання:

- Використовує фізичні принципи для моделювання систем та процесів у реальному світі.
- Може включати моделювання руху тіл, теплопередачу, електромагнітні явища тощо.

Ці види моделювання можуть бути використані окремо або в поєднанні для аналізу різних аспектів систем та процесів. Кожен з них має свої переваги та обмеження і використовується залежно від конкретних цілей дослідження.

Розглянемо тепер особливості екстраполяції як засобу моделювання.

Екстраполяція – це процес прогнозування значень величини поза межами доступних даних на основі відомих даних та деяких припущень про їх подальший розвиток. У контексті моделювання екстраполяція може використовуватися для прогнозування поведінки системи або процесу в майбутньому, особливо коли доступні дані обмежені або не можуть бути отримані в майбутньому. Ось деякі особливості екстраполяції як засобу моделювання:

1. Заснована на наявних даних. Екстраполяція використовує наявні дані для прогнозування значень величини за межами цих даних. Чим більше даних доступно для аналізу, тим більш точним і надійним може бути прогноз.

2. Потенційно неправильна. Екстраполяція може бути небезпечною, оскільки вона базується на припущеннях про подальший розвиток системи або процесу. Якщо ці припущення неправильні, екстрапольовані результати можуть бути неточними або навіть неправильними.

3. Залежить від контексту. Успішність екстраполяції залежить від контексту і характеристик системи чи процесу, які аналізуються. У деяких випадках екстраполяція може бути доволі точною, особливо якщо система має стабільні та прогнозовані закономірності.

4. Ризики великих відхилень. Екстраполяція може призвести до значних відхилень, особливо якщо взята модель неадекватна або якщо недостатньо враховано можливі фактори, які можуть вплинути на систему.

5. Необхідність перевірки. Перед використанням екстраполяції важливо перевірити її адекватність та достовірність. Це може включати тестування моделі на даних, які не використовувалися для її побудови, або порівняння прогнозів з реальними спостереженнями.

6. Використання як інструменту прогнозування. Екстраполяція широко використовується як інструмент прогнозування в різних галузях, як-от наука, економіка, фінанси тощо. Вона дає змогу здійснювати прогнози та приймати рішення на основі доступних даних і знань.

Використання екстраполяції у моделюванні може бути корисним інструментом для прогнозування подальшого розвитку системи чи процесу, але вимагає обережності та перевірки адекватності результатів.

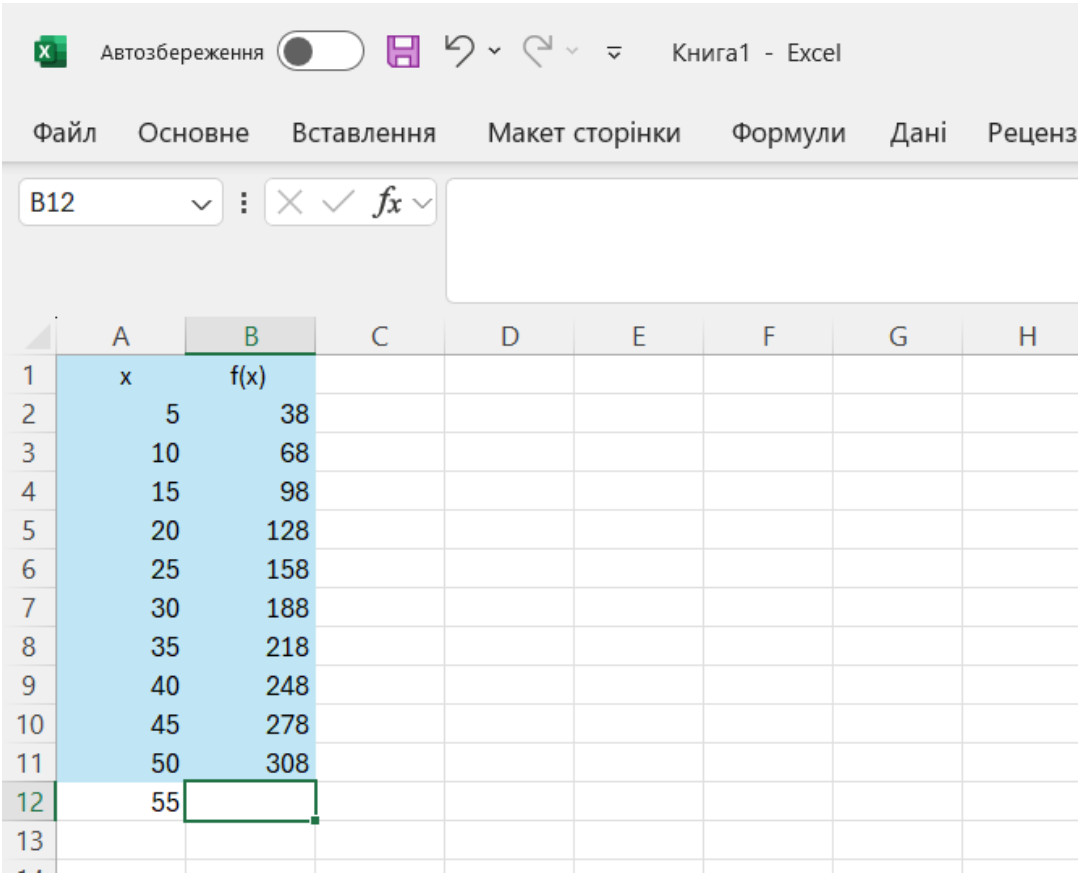
Виконання роботи. Існують випадки, коли потрібно дізнатися результати обчислення функції за межами відомої області. Особливо актуальне це питання для процедури прогнозування. У Екселі є кілька способів, за допомогою яких можна здійснити цю операцію. Давайте розглянемо їх на конкретних прикладах.

Використання екстраполяції

На відміну від інтерполяції, завданням якої є знаходження значення функції між двома відомими аргументами, екстраполяція передбачає пошук рішення за межами відомої області. Саме тому цей метод настільки затребуваний для прогнозування. У Екселі можна застосовувати екстраполяцію як для табличних значень, так і для графіків.

Спосіб 1: екстраполяція для табличних даних

Насамперед можна застосувати метод екстраполяції до вмісту табличного діапазону. Для прикладу візьмемо таблицю, в якій є ряд аргументів (x) від *5* до *50* (рис. 4.1) і ряд відповідних їм значень функції $f(x)$. Нам потрібно знайти значення функції для аргументу $X = 55$, який знаходиться за межею зазначеного масиву даних.

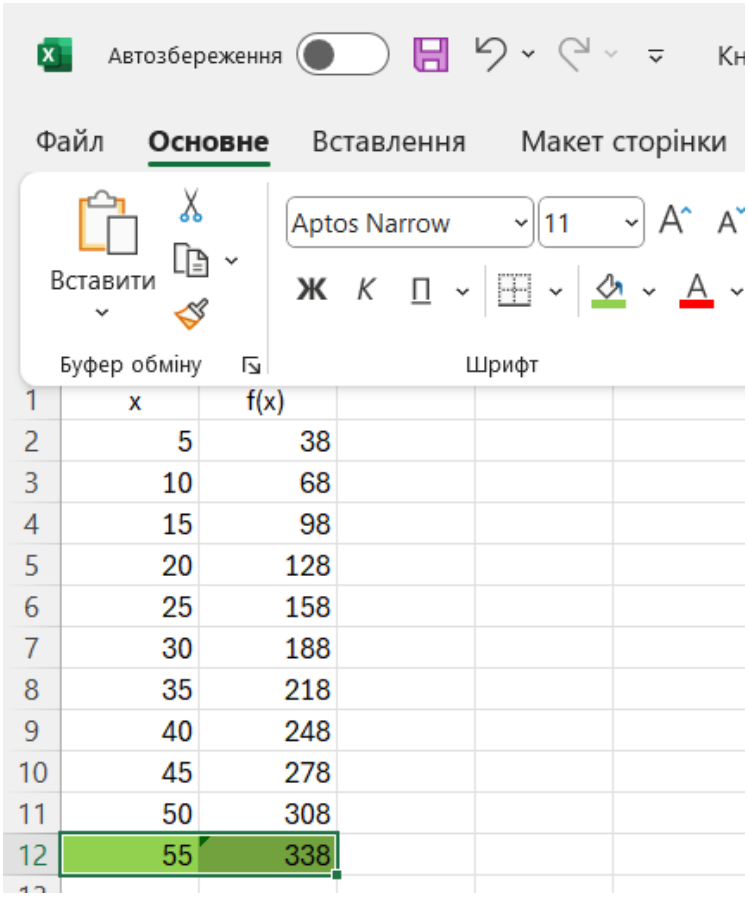


	A	B	C	D	E	F	G	H
1	x	f(x)						
2	5	38						
3	10	68						
4	15	98						
5	20	128						
6	25	158						
7	30	188						
8	35	218						
9	40	248						
10	45	278						
11	50	308						
12	55							
13								

Рис. 4.1. Виділено відомий масив значень x і відповідний масив значень $f(x)$

1. Виділяємо клітинку, у якій буде відображатися результат проведених обчислень. Клікаємо по значку *«Вставити функцію»,* який розміщений у рядку формул.

2. Запускається вікно *Майстра функцій.* Виконуємо перехід у категорію *«Статистичні»* або *«Повний алфавітний перелік».* У списку проводимо пошук найменування «FORECAST.LINEAR». Знайшовши його, виділяємо, а потім клацаємо по кнопці *«ОК»* у нижній частині вікна.



	x	f(x)
2	5	38
3	10	68
4	15	98
5	20	128
6	25	158
7	30	188
8	35	218
9	40	248
10	45	278
11	50	308
12	55	338

Рис. 4.2. Результат лінійного прогнозування $f(55) = 338$

Спосіб 2: екстраполяція для графіка

Виконати процедуру екстраполяції для графіка можна шляхом побудови лінії тренду.

1. Насамперед будуємо сам графік. Для цього курсором із затиснутою лівою кнопкою миші виділяємо всю область таблиці, включно з аргументами і відповідними значеннями функції. Потім послідовність дій така: *«Вставлення»* → *«Діаграми»* → *«Усі діаграми»* → *«Лінійчаста діаграма»*.

2. Після того, як графік побудований (рис. 4.3), видаляємо з нього додаткову лінію аргументу, виділивши її і натиснувши на кнопку *Delete* на клавіатурі комп'ютера. Отримуємо графік, представлений на рис. 4.4.

3. Далі нам потрібно поміняти поділки горизонтальної шкали, оскільки в ній відображаються не значення аргументів, які представлені на рис. 4.4. Для цього натискаємо правою кнопкою миші по діаграмі, і у списку, що з'явився, зупиняємося на значенні *«Вибрати дані»*.

4. У вікні *«Вибір джерела даних»* натискаємо кнопку *«Редагувати»,* вибравши попередньо діапазон даних x. Отримаємо графік показаний на рис. 4.5.

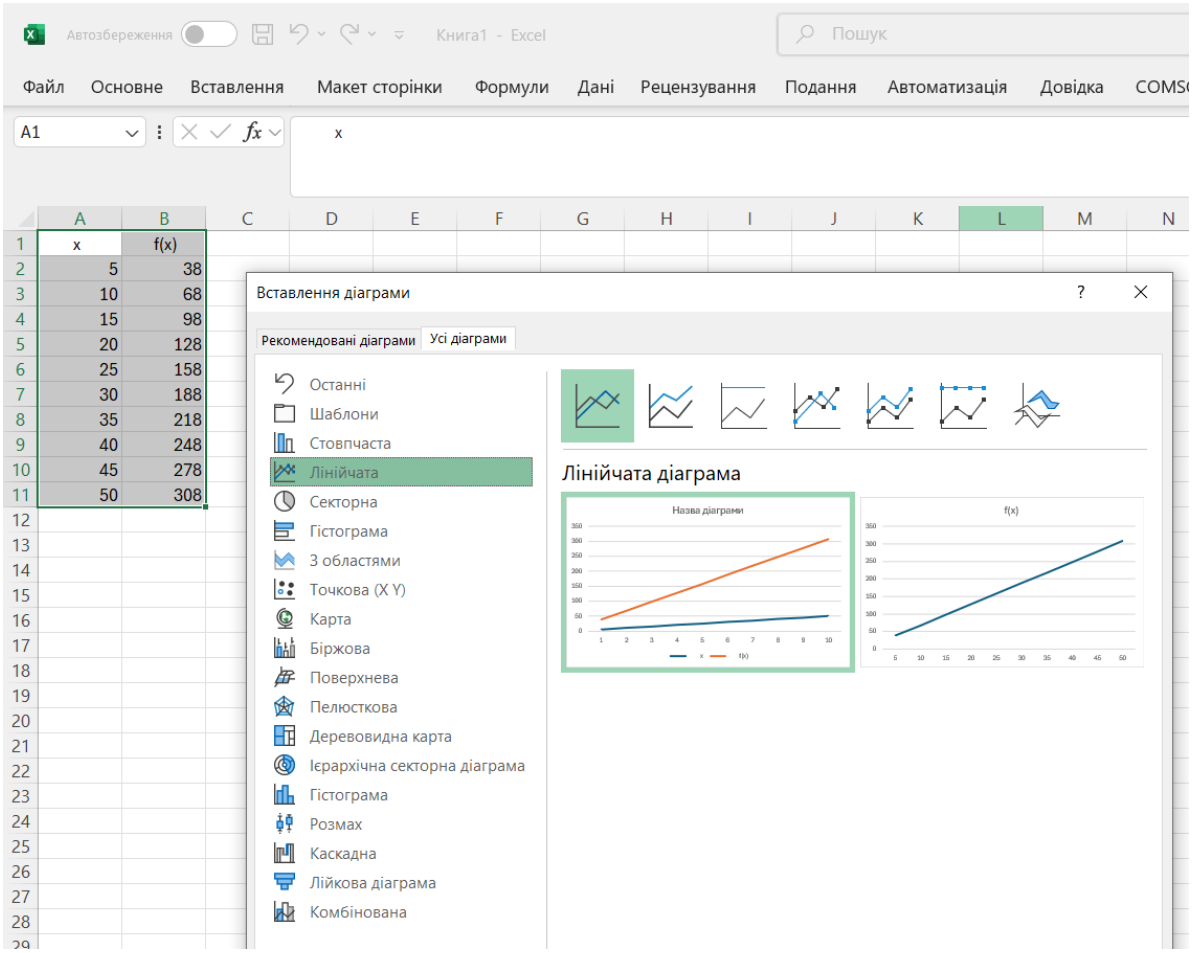


Рис. 4.3. Представлено таблично задану функцію (ліворуч) та вкладку з лінійчастою діаграмою (праворуч)

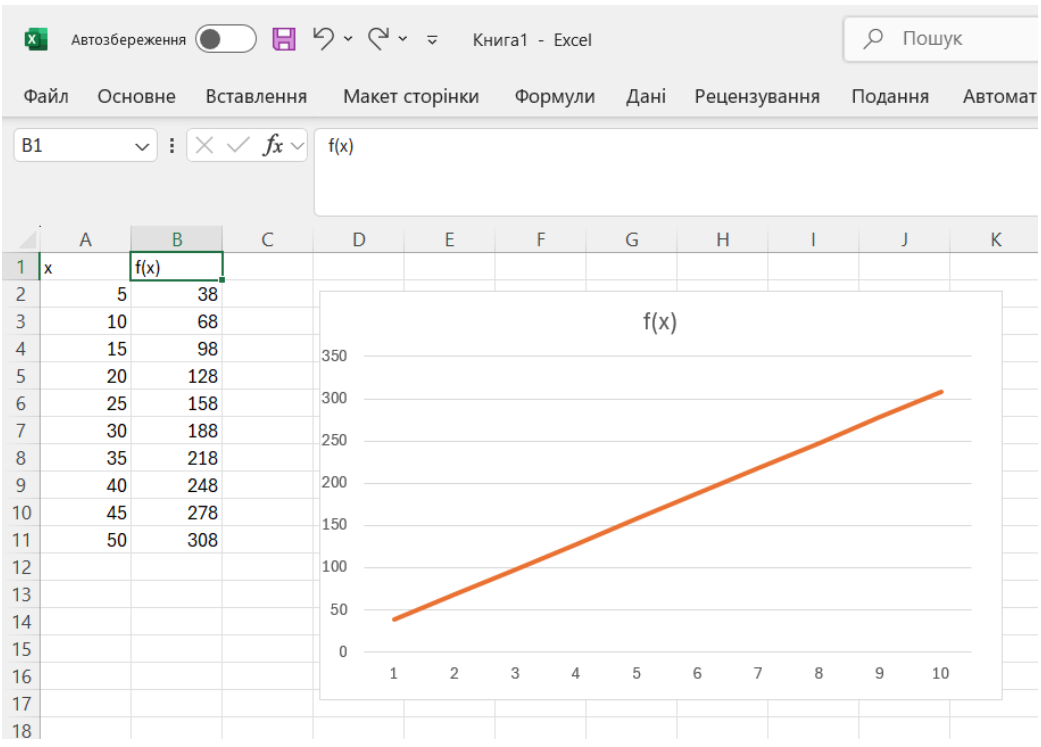


Рис. 4.4. Графік прямої лінії (праворуч) для даних, представлених ліворуч

5. Тепер наш графік підготовлений, і можна безпосередньо приступати до побудови лінії тренду. Клікаємо по графіку, після чого праворуч від рамки графіку активується додатковий набір вкладок – *«Елементи діаграми», «Стилі діаграми» і «Фільтри діаграми».* Переміщаємося у вкладку *«Елементи діаграми»* і тиснемо на кнопку *«Лінія тренду»*. Далі вибираємо *«Лінійний прогноз».* Остаточний графік приведений на рис. 4.6.

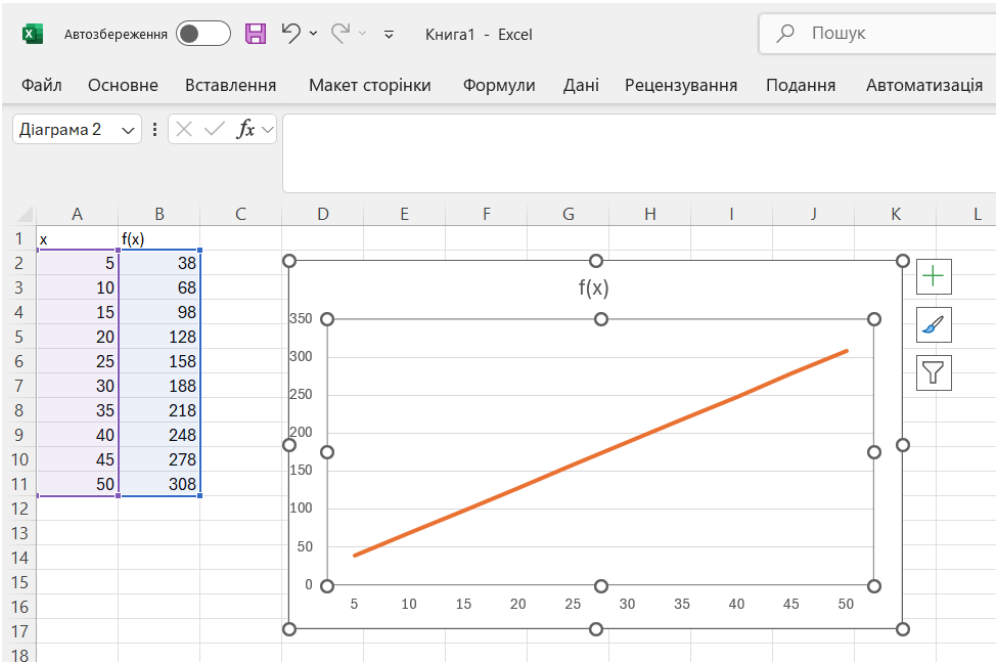


Рис. 4.5. Графік функції f(x)

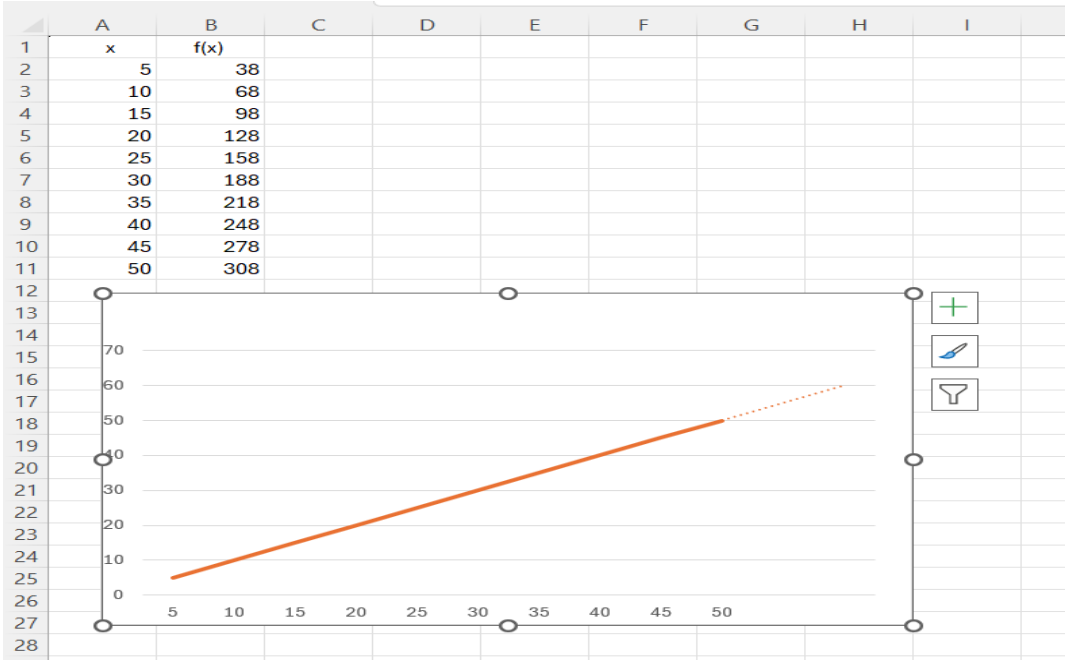


Рис. 4.6. Графік функції f(x): лінійний прогноз

Завдання: роботу виконати відповідно до представленого вище алгоритму.

ЛАБОРАТОРНА РОБОТА 5
СТВОРЕННЯ МОДЕЛІ З ДОПОМОГОЮ МЕТОДУ МОНТЕ-КАРЛО

Мета роботи: ознайомитися з особливостями моделювання за допомогою методу Монте-Карло.

Теоретичні відомості. Метод Монте Карло має справу з випадковими процесами. Цей метод базується на моделюванні реальних явищ. Побудова моделі стартує зі встановлення функціональних особливостей у модельованій системі. Кількісні закономірності встановлюються шляхом використання 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 йдеться про статистично-ймовірнісні дослідження, то симуляцію треба повторити декілька разів.

Яке використовується програмне забезпечення? 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 $M(X) = \alpha$. Формула розрахунку $M(X)$ записується так:

$$M(X) = \sum_{i=1}^{n} X_i \cdot p_i, \qquad (5.1)$$

де X_i – величина, що набуває спектр 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. За результатами експерименту отримуємо спектр значень X. Потім розраховується їх середнє арифметичне, яке і буде приблизним значенням α.

Виникає природне запитання: скільки експериментів потрібно здійснити? Загалом кількість імітаційних експериментів залежить від поставленої мети дослідження. Взагалі процес моделювання у режимі Монте-Карло може повторю-

ватись 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 має переваги і недоліки. До переваг належать 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 можливість працювати з різнорідними типами даних.

До недоліків потрібно віднести:

1) великі затрати часу для 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.

Виконання роботи. 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 всю площу цього квадрата (рис. 5.1) заповнюватимемо випадково згенерованими числами (далі у програмі Лістинг 5.1 у якості такого генератора використовується *Math.random*()). Деякі генеровані значення потраплять у круг (зелена зона на рис. 5.1), а інші – поза нього, але у межі квадрата (червона зона на рис. 5.1).

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

$$S_0 = \frac{\pi a^2}{4} = \frac{\pi S}{4}. \tag{5.2}$$

Із (5.2) отримуємо:

$$\pi = \frac{4S_0}{S} = \frac{4K}{S}. \tag{5.3}$$

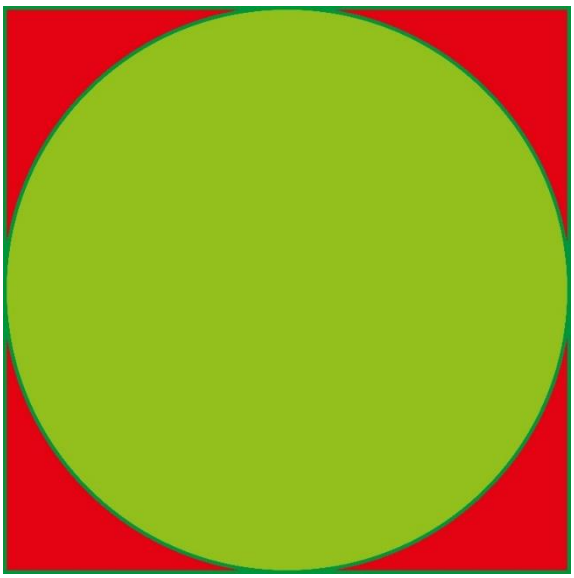


Рис. 5.1. Обчислення числа π методом Монте-Карло

Лістинг 5.1

```
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.

Тепер використаємо метод Монте-Карло для розрахунку інтегралу:

$$\int_0^5 \sqrt{2x^2 + x^3}\, dx. \tag{5.4}$$

Геометричне тлумачення інтегралу таке: це площа криволінійної трапеції, зверху обмеженої графіком підінтегральної функції, зліва і справа – вертикальними прямими, що проходять через точки меж інтегрування, і знизу – віссю *x*. Саме така криволінійна трапеція представлена на рис. 4.5. Завдання – обчислити площу даної криволінійної трапеції, тобто розрахувати інтеграл виду:

$$S = \int_{x_1}^{x_2} f(x)dx. \tag{5.5}$$

Розкидаємо випадковим способом, але максимально густо, точки у прямокутнику, зображеному на рис. 5.2. Нехай N – кількість точок, які потрапили у прямокутник; N_s – кількість точок, що потрапили під криву *f*(*x*), тобто у зафарбовану ділянку (на рис. 5.2 такі точки зображені кольоровими червоними колами). Очевидно, що кількість точок, які розміщені під кривою, порівняно з загальною кількістю точок у прямокутнику, пропорційна площі під *f*(*x*), тобто величині інтеграла. Представимо сказане у вигляді такого відношення:

$$\frac{N_S}{N} = \frac{S}{(y_2 - y_1)(x_2 - x_1)}. \tag{5.6}$$

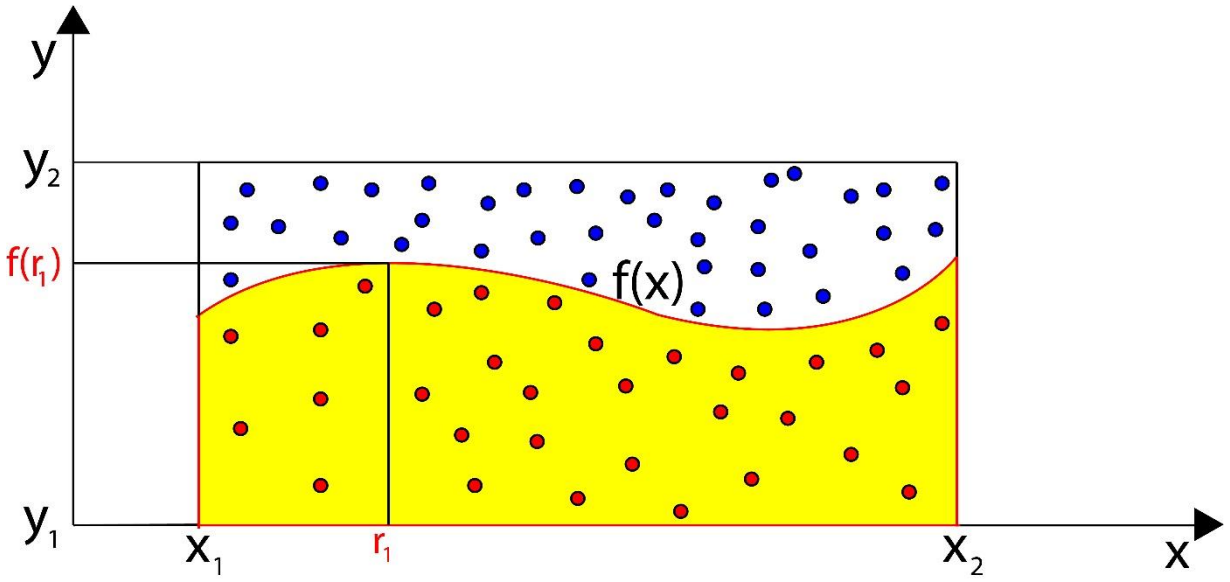


Рис. 5.2. Забарвлена криволінійна трапеція, площа якої обчислюється, заповнена точками червоного кольору. Величина забарвленої площі площа дорівнює інтегралу (5.5). Загальна кількість точок дорівнює кількості точок, що потрапили у прямокутник

Із (5.6) отримуємо вираз для обчислення інтеграла:

$$S = (y_2 - y_1)(x_2 - x_1)\frac{N_S}{N}. \qquad (5.7)$$

У Лістингу 5.2 описаний вище словесний алгоритм програмно представлений так:

Лістинг 5.2

```
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 5.2, доволі велика – сто мільйонів, але для комп'ютера це не становить проблеми: обчислення виконуються приблизно за одну секунду. Результат обчислень для функції (5.4) виглядає так: S = 28.665867. Запустивши програму знову, отримаємо таке 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.

Завдання: за допомогою методу Монте-Карло обчислити площу заданої викладачем фігури.

ЛАБОРАТОРНА РОБОТА 6
КОМП'ЮТЕРНА МОДЕЛЬ ТРАНСПОРТНОЇ МЕРЕЖІ МІСТА

Мета роботи: ознайомитися з особливостями моделювання транспортних мереж; навчитися прокладати оптимальні маршрути у транспортних мережах різної природи.

Теоретичні відомості. Одним з ефективних засобів моделювання транспортних (і не тільки) мереж є алгоритм Флойда. Це доволі 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 графу, взятої напряму, та відстані між цими ж вершинами, але пройденої через інші два ребра, тобто обхідним шляхом. Ситуація наочно представлена на рис. 6.1.

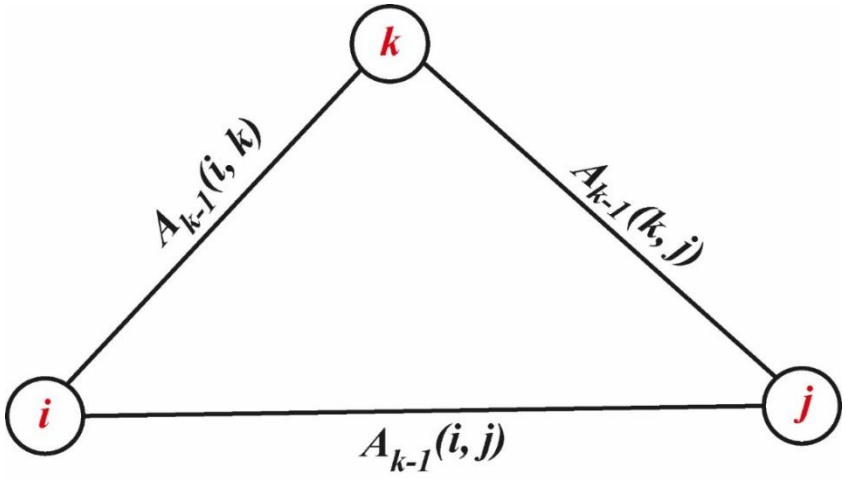


Рис. 6.1. Елементарний фрагмент у алгоритмі пошуку оптимального шляху у графі

Записати аналітично процедуру знаходження мінімальної відстані між сусідніми вершинами графу 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. Під час аналізу цього рисунка 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 графу.

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

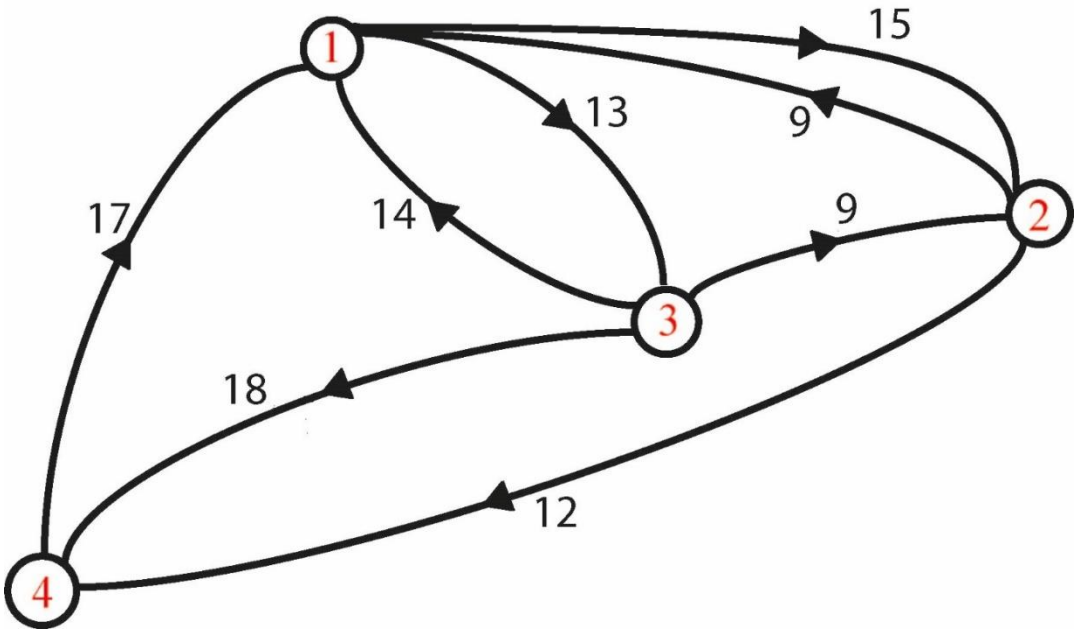


Рис. 6.2. Орієнтований навантажений планарний мультиграф

Нехай для графу, представленого на рис. 6.2, необхідно знайти мінімальні відстані між усіма вершинами. Для цього використаємо алгоритм Флойда. Сформуємо для такого графу вагову матрицю A_{ij} відповідно до визначення (6.1). На нульовому кроці алгоритму це буде просто вагова матриця графу:

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

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

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

Як бачимо, у цьому виразі змінна k набуває 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 за 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 графу, то відповідно до алгоритму Флойда, пробігаємо по всіх вершинах цього графу і шукаємо більш короткий 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];}.
```

Виконання роботи. Для графа із 16 ребрами та 6 вершинами, зображеного на рис. 6.1, наведена далі програма (Лістинг 6.1). У програмі Лістинг 6.1 двовимірний масив треба розуміти так: перше число – це номер:

```
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}};
```

вершини, з якої виходить ребро, друге – це номер вершини, у яку входить ребро, і третє – це вага такого ребра. І в такий спосіб задається будь-який граф. Запустимо програму та проаналізуємо результат її роботи.

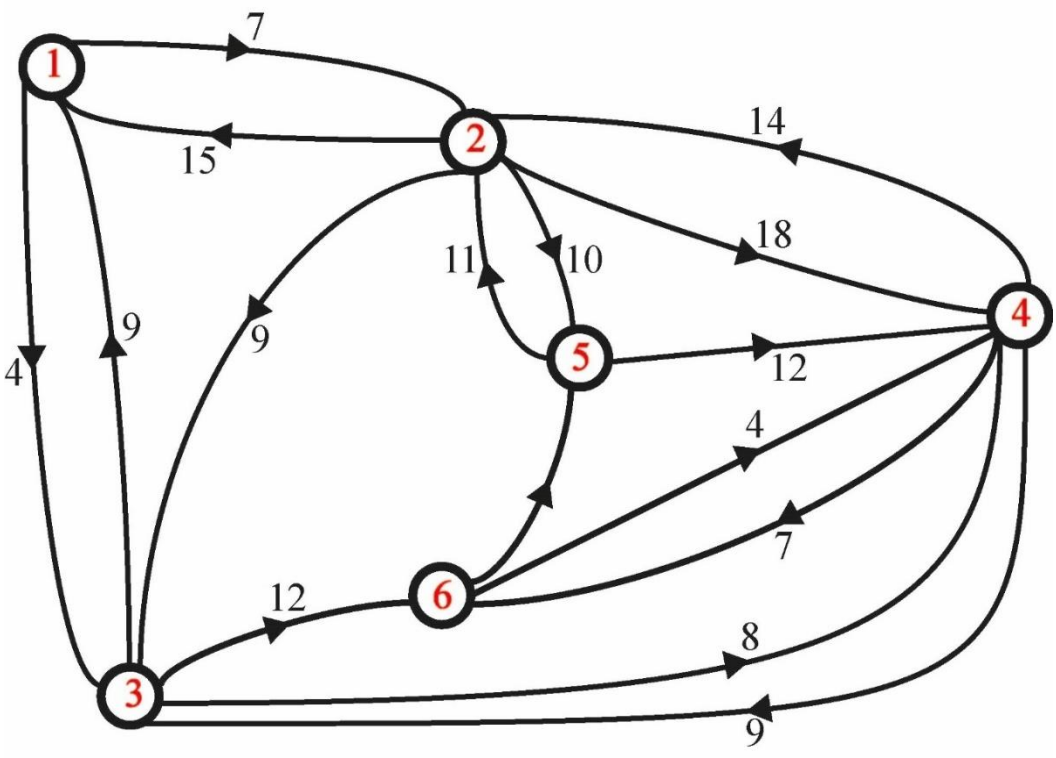


Рис. 6.3. Зважений орієнтований граф

Лістинг 6.1

```
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 графу, а також довжини цих маршрутів. Скажімо, найкоротший шлях у напрямку від вершини 6 до вершини 1, тобто маршрут 6 → 1, дорівнює 22 та пролягає вздовж простого ланцюга 6 → 4 → 3 → 1.

Ефективним алгоритмом прокладання оптимальних маршрутів у мережах різної природи є A^*-алгоритм. Нехай потрібно знайти оптимальний маршрут у мережі, представленій у вигляді графу на рис. 6.4. 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 алгоритмом. Особливість полягає у введенні поняття еврис-

тичної відстані. Позначимо цю відстань h(x), де x – номер вибраної вершини графу. Це відстань, взята по прямій, від всіх вершин графу 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 графу. Завдяки введенню евристики зменшується кількість можливих варіантів вибору шляху, 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).

Представимо тепер наведений описовий варіант, а також програмну реалізацію A*-алгоритму, та шляхи його використання під час прокладання оптимальних маршрутів у різних мережах, зокрема і міських транспортних.

Розглянемо граф, представлений на рис. 6.4. Припустимо, треба знайти оптимальний маршрут між вершинами C та S. Скористаємось A*-алгоритмом. Сформуємо таблицю, у яку від початку 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.

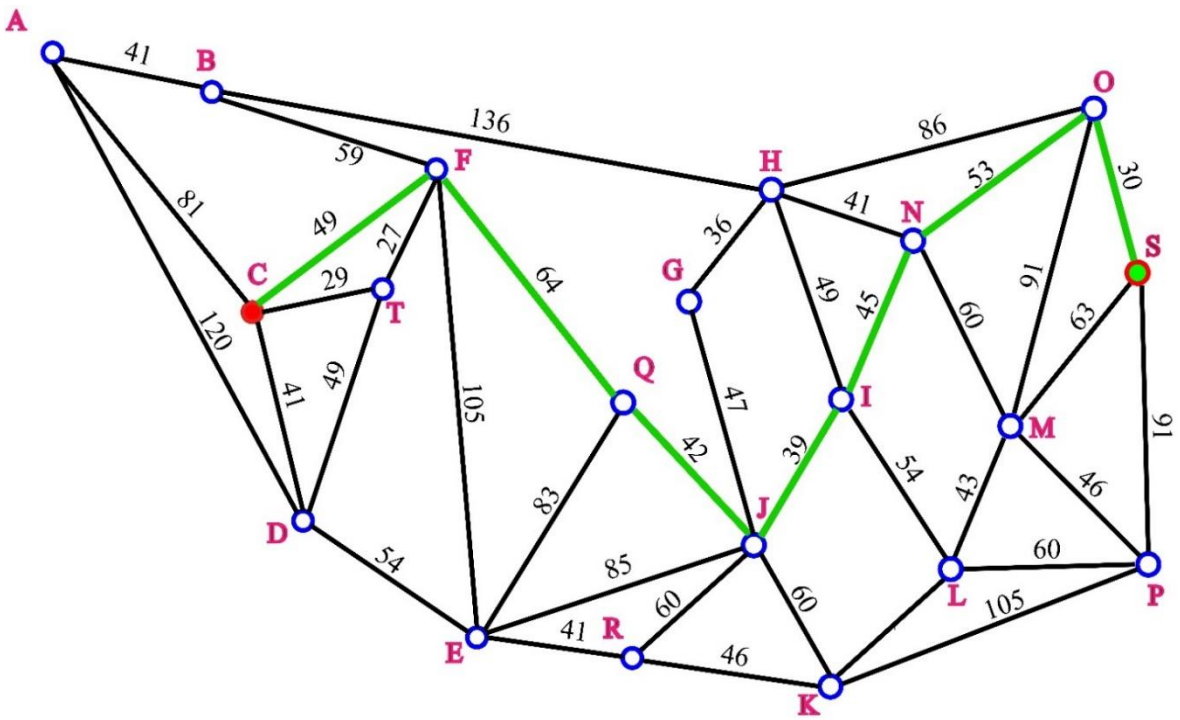


Рис. 6.4. Зважений граф, у якому за допомогою A-алгоритму знаходиться оптимальний маршрут між вершинами C (стартова вершина) і S (фінішна вершина). Цей маршрут виділений потовщеною зеленою ламаною лінією. Біля кожного ребра проставлені ваги, що відповідають геометрії задачі*

Для графу, представленого на рис. 6.4, оптимальний маршрут між вершинами C та S і його довжина прораховані та представлені у табл. 6.1.

Таблиця 6.1 – 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 графу 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. 6.1 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 з вершини C, тому у стовпчику «Попередня вершина» виставляємо C. Продовжуючи процедуру вибору оптимального маршруту, далі аналогічним способом заповнюємо табл. 6.1. Аналізуючи отримані результати, бачимо, що протяжність оптимального маршруту C→S становить 322 умовні одиниці. За допомогою табл. 6.2 легко можна відтворити оптимальний маршрут у графі (рис. 6.14). Для цього зауважимо, що у вершину S перехід відбувся з вершини O. У вершину O перехід був здійснений із N. Продовжуючи аналіз, відтворюємо оптимальний маршрут C→ F → Q → J → I → N → O →S.

Звернемо увагу, що у табл. 6.1 у стовпчику «Вершина» пройдені вершини зсовуються правіше. 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 6.2).

Лістинг 6.2

```
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)) {
            //даний 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 зі знайденим раніше за допомогою «ручного» алгоритму. Важливість A*-алгоритму полягає у тому, що відсікаються неперспективні варіанти маршрутів завдяки аналізу значень функції $f(x)$. Натомість величини $f(x)$ залежать від значень функції $h(x)$.

Завдання: використовуючи наведені вище алгоритми, прокласти оптимальні маршрути для наведеного викладачем графу. Запустити програму та отримати результат у вигляді прокладеного маршруту.

ЛАБОРАТОРНА РОБОТА 7
СТВОРЕННЯ МОДЕЛІ ЗА ДОПОМОГОЮ МЕРЕЖ ПЕТРІ

Мета роботи: ознайомитися з особливостями моделювання за допомогою мереж Петрі; створити власну мережу Петрі; запустити мережу; пояснити принцип роботи мережі.

Теоретичні відомості. Мережі Петрі – це процес моделювання конкурентних та паралельних процесів, що включає в себе елементи стану, переходів та зв'язків між ними. Теорія мереж Петрі – формалізм, призначений для імітації роботи динамічних систем. Зараз розроблена велика кількість різноманітних моделей, методів та засобів аналізу, а також різноманітних додатків практично в усіх галузях. Мережі Петрі є типовим прикладом.

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 зображуються 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 зі вхідних позицій переміщуються у вихідні позиції (зліва направо, якщо розглядати рис. 7.1). Зауважимо, що події відбуваються миттєво, у різні моменти часу і у разі виконання певних умов.

Мережа Петрі функціонує шляхом здійснення запусків переходів. Запуск переходу може бути здійснений тільки у випадку, коли в усіх вхідних позиціях є хоча б одна фішка. Водночас маркери зі вхідних позицій переміщуються у вихідні (маркери зліва зі вхідних позицій переміщуються у вихідну позицію справа – рис. 7.1). Послідовність таких переміщень маркерів являє собою модельований процес, конкретний приклад якого буде наведено далі.

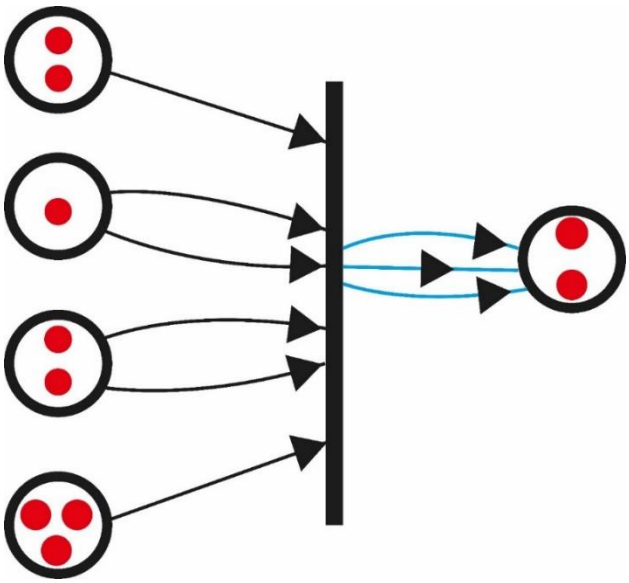


Рис. 7.1. Чорними кільцями зображені позиції P (зліва – вхідні, а справа – вихідні). Переходи T представлені прямокутною чорною смужкою. Вхідні функції I зображуються дугами чорного кольору та розміщені зліва; вихідні функції O розміщені праворуч та зображені дугами блакитного кольору. Червоні кружечки всередині позицій – це так звані мітки

Існують суворі правила спрацювання переходів. Перехід спрацьовує, якщо для кожної з 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. І 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, якщо для другої зверху вхідної позиції умова $N_i \geq K_i$ не виконується? Розглянемо тепер ситуацію, представлену на рис. 7.2. Для такої мережі Петрі спрацювання переходу відбудеться, оскільки умова $N_i \geq K_i$ у цьому випадку виконується. Маркування (розподіл маркерів за позиціями – рис. 7.2) представляють у вигляді послідовності (1,3,3,3,2). У конкретний момент часу може спрацювати лише один перехід, що вирішує проблему конфліктів.

Існують різні види мереж Петрі. Одним із таких видів є часові мережі Петрі, у яких вводиться так званий модельний час, що 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, під час 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. 7.2 зображений саме збуджений перехід).

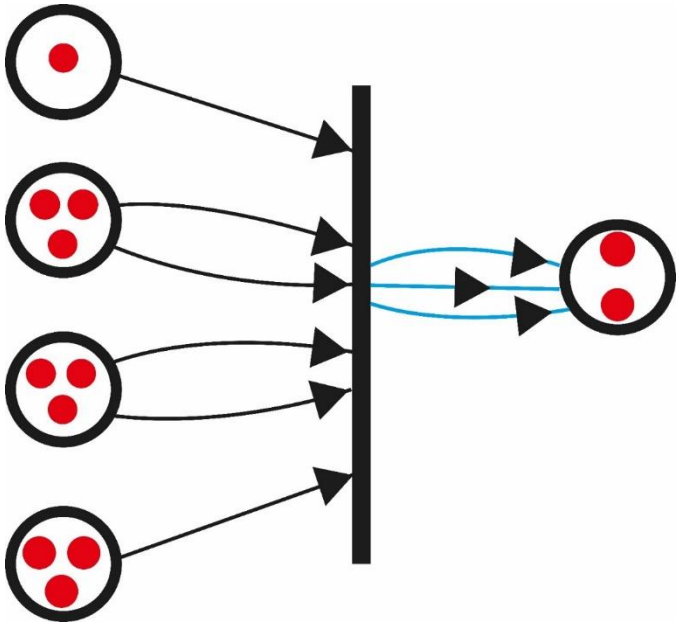


Рис. 7.2. Мережа Петрі, для якої відбувається спрацювання переходу

Тимчасова мережа Петрі – переходи мають вагу, що визначає тривалість спрацьовування (затримку), тобто переходи здобувають вагу, у якості якої виступає тривалість затримки спрацювання переходу. Вага переходу може залежати від спектра факторів, зокрема від кількості маркерів у вихідних позиціях чи їх розподілу по цих позиціях. Подібну мережу називають функціональною. Загалом розрізняють такі види мереж Петрі:

- 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;
- 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;

- WF-мережі;
- алгебраїчні мережі Петрі.

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

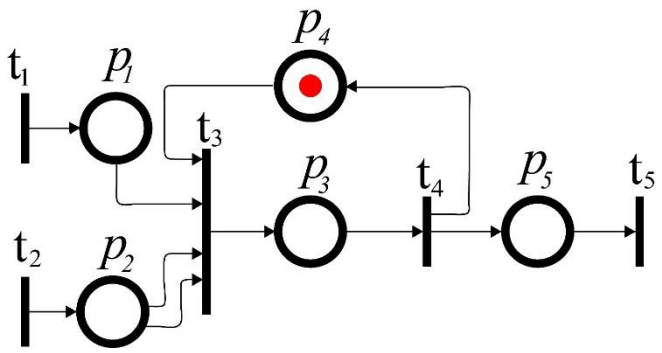


Рис. 7.3. Інгібіторна мережа Петрі

Опишемо принцип роботи мережі Петрі, представленої на рис. 7.3. Нехай існує необхідність моделювання роботи заводу, що складається з трьох цехів: два цехи заводу 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 X_2. На рис. 7.3 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.

Запустимо модель, використовуючи Petry.NET simulator 2.0, що реалізує описану вище модель. Представлений на рис. 7.4 скрин екрана симулятора відповідає моменту завершення роботи моделі.

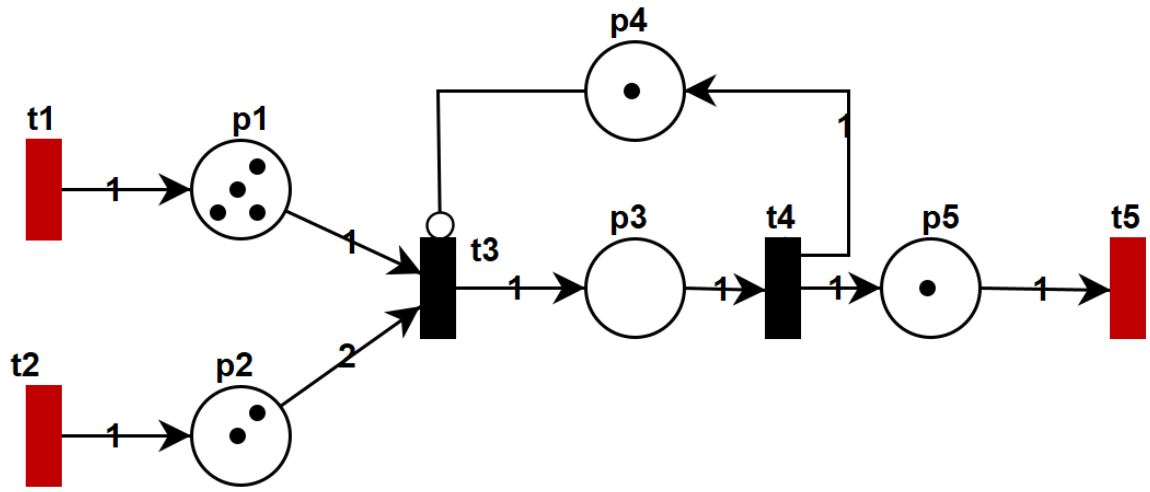


Рис. 7.4. Робота Petry.NET simulator 2.0.
Кількість вхідних та вихідних функцій зображується цифрою, виставленою на дугах

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-от PIPE2 (Platform Independent Petri net Editor) – платформонезалежний редактор мереж Петрі. Ця розробка ведеться за принципом відкритого програмного забезпечення, що дає змогу 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. 7.5) – додаток, створений для моделювання різних процесів 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 найрізноманітніших систем. Звернемо увагу на те, що математичний апарат мереж Петрі доволі 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.

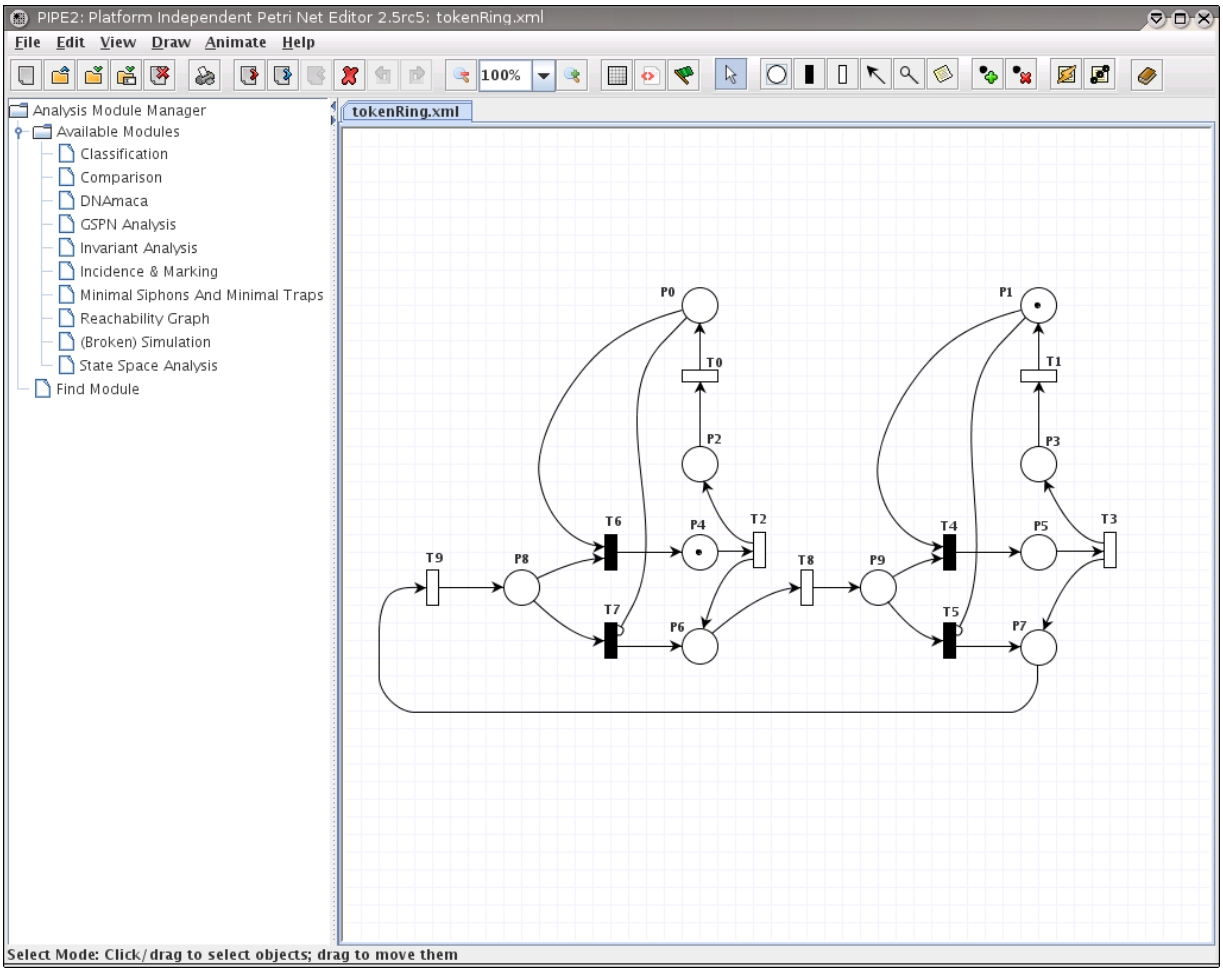


Рис. 7.5. Інтерфейс та робоча зона Platform Independent Petri net Editor (PIPE2)

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.

Виконання роботи. Алгоритм створення мережі Петрі виглядає так:

1. Визначення множини місць (P): місця представляють стани системи. Кожне місце має певну кількість маркерів, які вказують на поточний стан системи.

2. Визначення множини переходів (T): переходи представляють події або дії, які можуть змінити стан системи. Вони можуть бути активовані лише у випадку, якщо всі вхідні місця мають достатню кількість маркерів.

3. Визначення функцій переходів (δ): функції переходів визначають, як переходи впливають на місця. Вони вказують, яку кількість маркерів додавати або видаляти з кожного місця під час активації переходу.

4. Визначення початкового стану мережі: це включає в себе розстановку маркерів у місцях у початковий момент часу.

5. Визначення правил переходів: це визначає умови, за яких переходи можуть бути активовані. Ці умови пов'язані з кількістю маркерів у вхідних місцях.

6. Симуляція роботи мережі Петрі: проведення послідовності переходів на основі правил активації та функцій переходів.

7. Аналіз результатів: після симуляції аналізується стан системи та її поведінка, а також ефективність процесів.

8. Модифікація мережі за необхідності: зміна мережі Петрі для вдосконалення моделювання або вирішення виявлених проблем.

Цей алгоритм може варіюватися залежно від конкретного завдання та потреб моделювання. Важливо мати на увазі, що мережа Петрі є потужним інструментом для аналізу та моделювання паралельних та конкурентних систем.

Завдання: створити реально функціонуючу мережу Петрі. Запустити мережу. Пояснити принцип роботи.

ЛАБОРАТОРНА РОБОТА 8
СТВОРЕННЯ МОДЕЛІ З ВИКОРИСТАННЯМ КЛІТИННИХ АВТОМАТІВ

Мета роботи: ознайомитися з особливостями моделі клітинних автоматів; вивчити особливості функціонування клітинних автоматів у різних сферах; навчитися моделювати природні процеси з використанням клітинних автоматів.

Теоретичні відомості. 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 і 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 у 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 з восьми клітинок, спектр можливих станів розраховується за формулою комбінаторики (8.1), що описує розміщення з повтореннями:

$$\widetilde{A_n^r} = n^r, \tag{8.1}$$

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. 8.1, то отримаємо кількість можливих станів КА, рівну $2^8 = 256$.

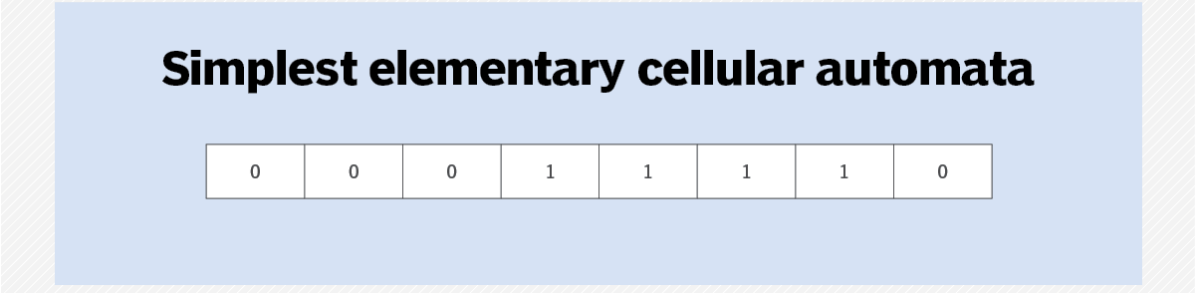


Рис. 8.1. Двійковий одновимірний автомат, що складається з восьми комірок

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. 8.2).

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 відбувається процес розмноження.

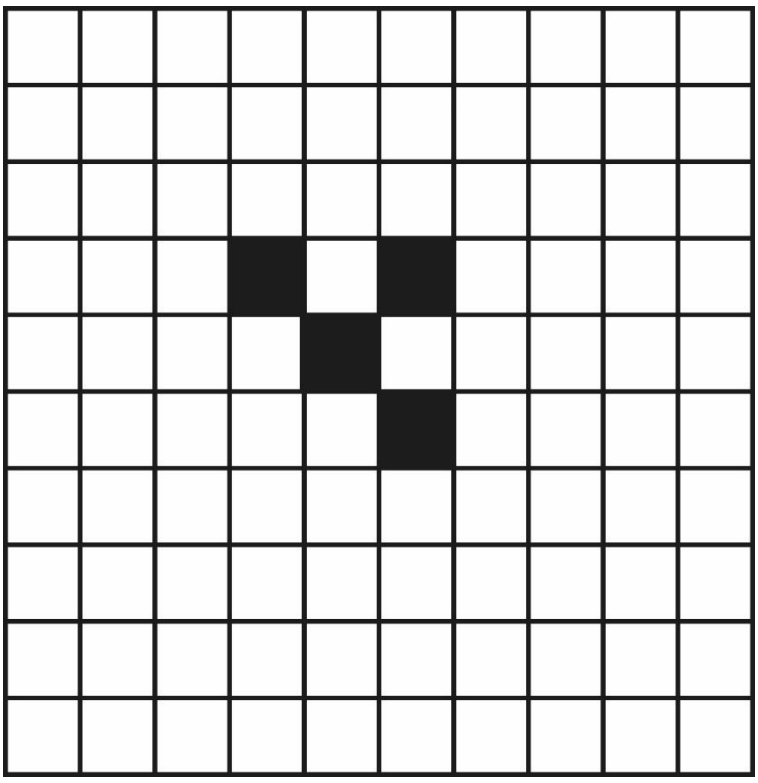


Рис. 8.2. Двовимірний автомат розміру 10 × 10

Нехай у наступній схемі **«1»** означає живу клітинку, а **«•»** – мертву (рис. 8.3). 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.

ВХІДНІ ДАНІ:

••••••••••
••• 11 •••••
•••• 1 •••••
••••••••••
••••••••••

ВИХІДНІ ДАНІ:

••••••••••
••• 11 •••••
••• 11 •••••
••••••••••
••••••••••

Рис. 8.3. Народження нової клітини у позиції (3,4), де мертва клітина стає живою, згідно з пунктом 4 наведених вище ПРАВИЛ

Виконання роботи. Представимо далі програмну реалізацію Game of Life на Java (Лістинг 8.1). 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 8.1

```
// 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. 8.4 та 8.5):

Стартова генерація

••••••••••
••• 11 •••••
•••• 1 •••••
••••••••••
••••••••••
••• 11 •••••
•• 11 ••••••
••••• 1 ••••
•••• 1 •••••
••••••••••

Рис. 8.4. Стартове розташування живих та мертвих клітин

Наступна генерація

••••••••••
••• 11 •••••
••• 11 •••••
••••••••••
••••••••••
•• 111 •••••
•• 11 ••••••
••• 11 •••••
••••••••••
••••••••••

Рис. 8.5. Розташування живих та мертвих клітин у наступній генерації

Розташування живих та мертвих клітин, представлене на рис. 8.4, 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. 8.3).

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 8.1 для ситуації, зображеної на рис. 8.2. Для цього у програмі стартову матрицю представимо так (рис. 8.6):

```
{ 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. 8.6. 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. 8.7), так і у наступній генерації клітин (рис. 8.8).

Стартова генерація

```
••••••••••
••••••••••
••••••••••
••• 1 • 1 ••••
•••• 1 •••••
••••• 1 ••••
••••••••••
••••••••••
••••••••••
••••••••••
```

Рис. 8.7. Розташування живих та мертвих клітин у стартовій позиції

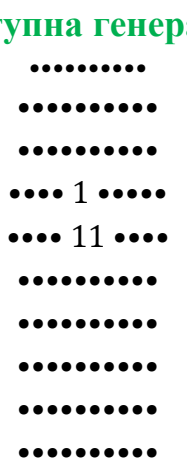


Рис. 8.8. Розташування живих та мертвих клітин у наступній генерації

Проаналізуємо отриманий результат. Клітина з координатами (4,4) гине, згідно з пунктом 1 ПРАВИЛ (рис. 8.7). 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. 8.8.

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

Створення клітинного автомату – це процес створення моделі, що складається з сітки клітин, кожна з яких може знаходитися в різних станах, та визначення правил для зміни станів цих клітин з плином часу. Ось алгоритм створення клітинного автомату:

1. Визначення розмірів сітки: виберіть розмір сітки, тобто кількість рядків і стовпців, на якій буде розгорнута ваша модель клітинного автомату.

2. Ініціалізація початкових станів клітин: встановіть початкові стани клітин, які можуть бути живими або мертвими (або в інших станах, залежно від конкретної задачі).

3. Визначення правил переходу: визначте правила, за якими стани клітин змінюються з часом. Це може бути здійснено шляхом встановлення правил для кожного типу клітини на основі стану її сусідів.

4. Виконання ітераційного процесу: повторюйте процес зміни станів клітин відповідно до визначених правил. Це може бути здійснено для кожного кроку часу в моделі.

5. Аналіз результатів: аналізуйте стан клітинної сітки після кожної ітерації, спостерігаючи за динамікою зміни станів та будь-якими властивостями або патернами, які виникають.

6. Модифікація правил та параметрів: виправлення або зміна правил клітинного автомату для досягнення бажаних результатів або для вирішення конкретних завдань.

7. Додатковий аналіз і візуалізація: проведення додаткового аналізу результатів, а також візуалізація динаміки системи за допомогою графіків або анімацій.

8. Тестування та перевірка: перевірте, чи відображає модель реальні явища або задачі, що вона моделює, і внесіть необхідні зміни, якщо це необхідно.

Цей алгоритм може бути модифікований та адаптований залежно від конкретного застосування клітинного автомату.

Завдання: використовуючи представлені теоретичні відомості та алгоритм, скласти реально функціонуючий клітинний автомат. Запустити модель та описати її функціональні особливості і практичне застосування.

ЛАБОРАТОРНА РОБОТА 9
МОДЕЛЮВАННЯ ФІЗИЧНИХ СИСТЕМ

Мета роботи: ознайомитися з особливостями моделювання фізичних систем різної природи; вивчити особливості моделювання швидкоплинних фізичних процесів, моделювання природних процесів.

Теоретичні відомості. 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 наявна фізична подібність реальній досліджуваній системі. Фізична подібність фактично означає повну відповідність між параметрами об'єкта і моделі. Під моделлю тут треба розуміти фізичне устаткування, створене для вивчення системи, процесу, явища або об'єкта.

Прикладом фізичної моделі може слугувати 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 із глибоководним апаратом «Титан» 18 червня 2023 р. під час занурення до уламків лайнера «Титанік».

Загалом фізичне моделювання спрямоване на експериментальне вивчення таких фізичних явищ та процесів, які у природних умовах дослідити дуже проблематично або взагалі неможливо. Наприклад, розглянемо кульову блискавку. Для її вивчення потрібно створити таку 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, коли сила опору (так звана сила Стокса) стає мінімальною. На рис. 9.1 зображено макет літального апарата, розташований в аеродинамічній трубі.

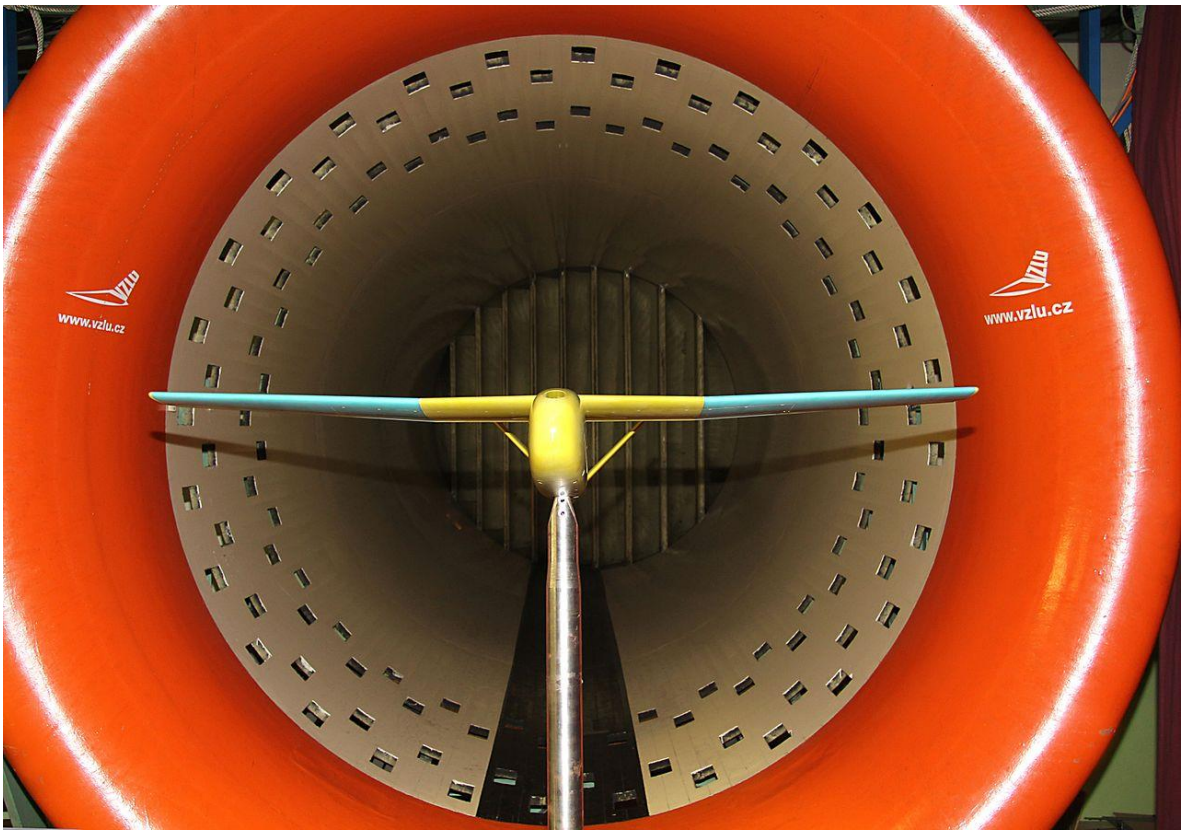


Рис. 9.1. Макет літального апарата, розташований в аеродинамічній трубі. Дослідження проводяться з метою створення такої форми літального апарата, за якої величина сили Стокса є мінімальною

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

$$U_{92}^{235} + n_0^1 \rightarrow Cs_{55}^{140} + Rb_{37}^{94} + 2n_0^1 + Q. \qquad (9.1)$$

Записану реакцію треба 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 (9.1).

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 (9.1) z нейтронів може викликати поділ z інших ядер урану. Отже, буде випущено z^2 нейтронів нового покоління. В кожному новому поколінні кількість нейтронів буде наростати

в геометричній прогресії. Тут доцільно ввести поняття коефіцієнта розмноження нейтронів k, що являє собою відношення числа нейтронів наступного покоління до числа нейтронів попереднього покоління у всьому об'ємі активного нейтронного середовища, наприклад, в активній зоні ядерного реактора. Так от, при $k > 1$ самоплинно буде продовжуватись реакція поділу, як це схематично показано на рис. 9.2.

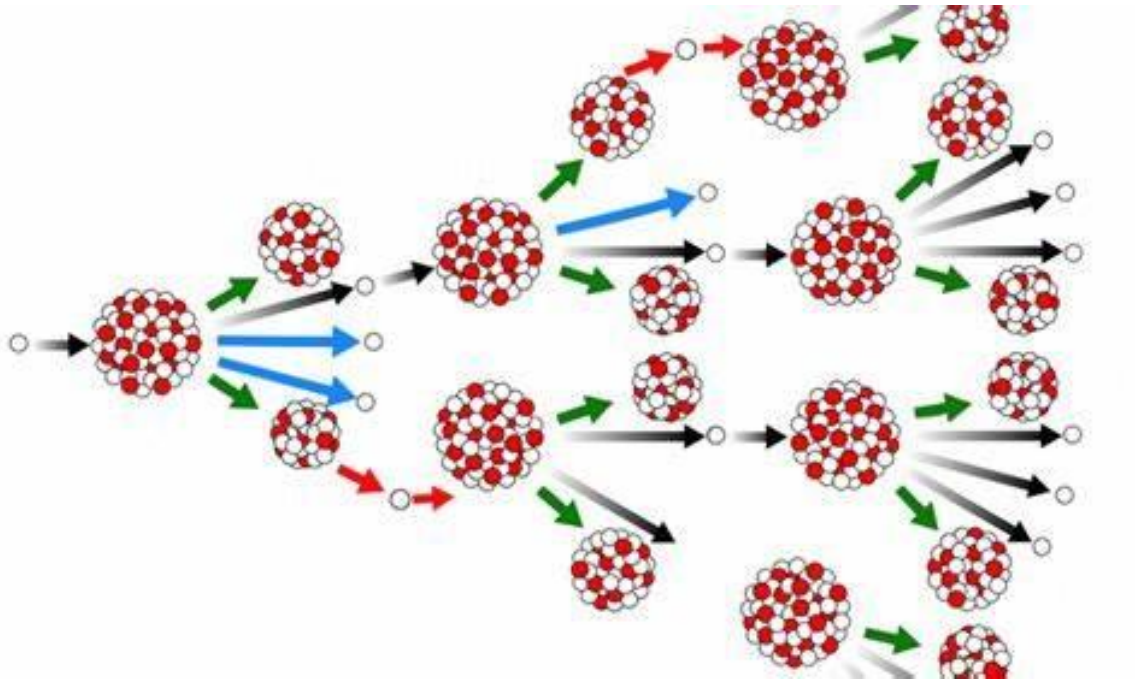


Рис. 9.2. Схема ланцюгової реакції поділу ядер радіоактивних елементів

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 $k > 1$.

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

Певним аналогом описаного тріадного атомного процесу є епідемії та пандемії. Останніми роками добре відомою пандемією є 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 в 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 такому випадку епідемію такого типу можна подавити шляхом проведення медичних профілактичних заходів. Така ситуація характерна для гострої вірусної інфекції на кшталт грипу. Особливо небезпечною є ситуація, коли $R_0^H > 1$. У цьому випадку інфекція швидко розповсюджується, водночас 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. 9.3 представлена модель SIR, запропонована Кермаком і МакКендріком. Відповідно до цієї моделі 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.

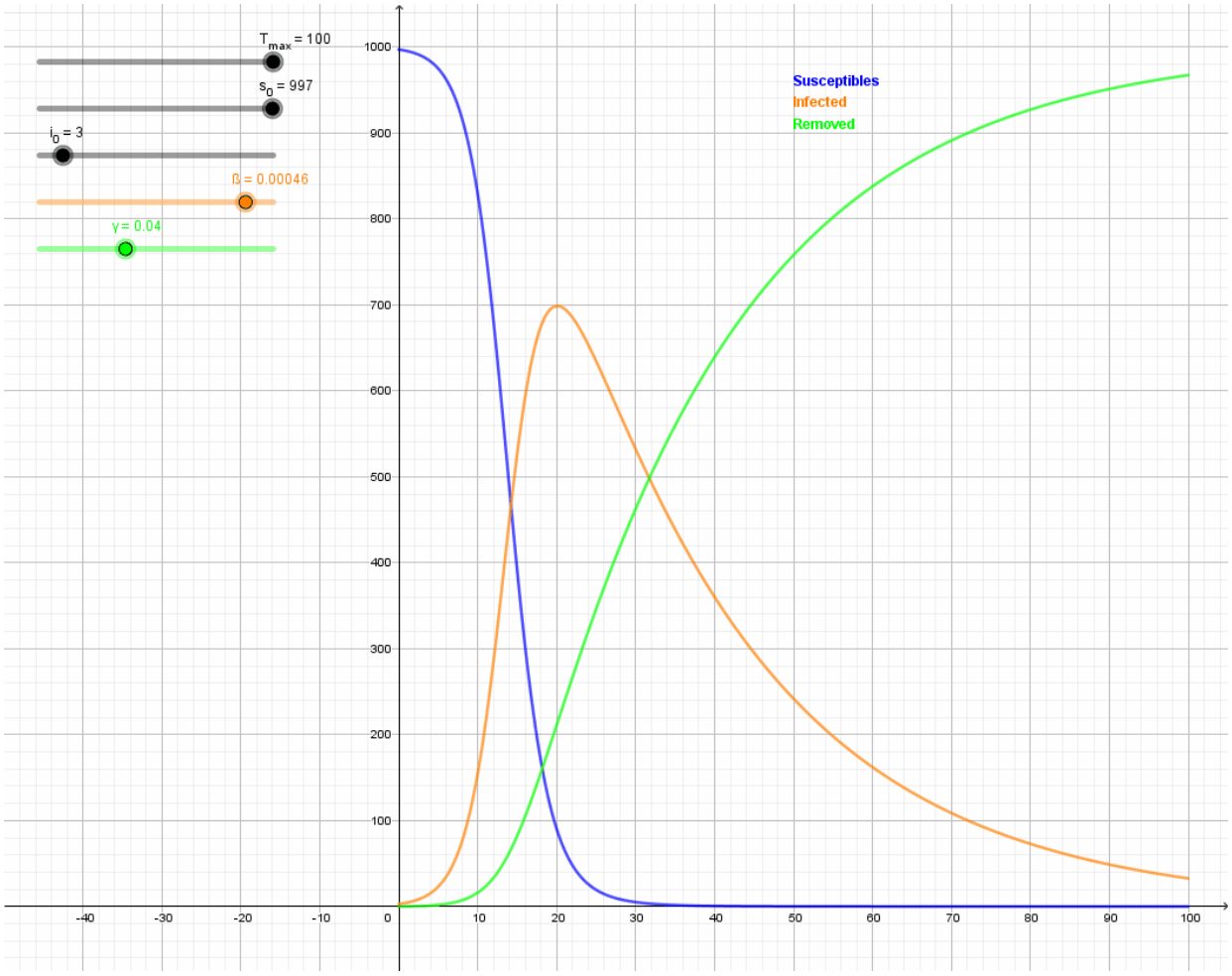


Рис. 9.3. Модель SIR розповсюдження інфекції

Особливо важливе значення має вигляд функції $I(t)$. Як видно з рис. 9.2, 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. 9.1). Лавиноподібний процес як у першому, так 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 у цьому сенсі зауважити, що вирішальний вплив на поведінку функції $I(t)$ відіграє фактор часу t (назвемо його t-фактор). Фактично це означає *своєчасне* прийняття комплексних заходів щодо 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 затратити 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 9.1. У швидкоплинних процесах типу атомних реакцій, епідемій, пандемій та пожеж вирішальне значення відіграє t-фактор.

Виконання роботи. Моделювання фізичного процесу – це процес створення математичної моделі, яка описує рух, взаємодію та динаміку фізичних систем або явищ. Ось загальний алгоритм моделювання фізичного процесу:

1. Визначення системи та її меж: це включає в себе розуміння об’єкта або системи, яку ви хочете моделювати, і визначення меж, у яких відбуваються фізичні процеси.

2. Вибір математичної моделі: виберіть або розробіть математичну модель, яка найкраще відображає фізичні властивості системи. Це може бути здійснено на основі фізичних законів, емпіричних даних або комбінації обох.

3. Формалізація початкових умов: встановіть початкові значення для всіх змінних, що входять до математичної моделі, щоб почати моделювання з певного початкового стану.

4. Визначення вхідних параметрів та обмежень: це включає в себе визначення параметрів, що можуть змінюватися з часом, а також будь-яких обмежень або умов, які повинні бути враховані в моделі.

5. Вибір методу чисельного розв'язку: виберіть або розробіть метод чисельного розв'язку, який дасть змогу обчислити значення змінних у моделі з плином часу. Це можуть бути методи диференціальних рівнянь, методи скінченних елементів або інші чисельні методи.

6. Виконання чисельного моделювання: застосуйте обраний метод чисельного розв'язку для обчислення значень змінних у моделі з плином часу. Це може вимагати використання комп'ютерних програм або пакетів для чисельного моделювання.

7. Аналіз результатів: аналізуйте отримані дані та візуалізуйте їх, щоб зрозуміти поведінку системи в часі та відносно різних умов.

8. Валідація та верифікація моделі: перевірте, наскільки точно ваша модель відображає реальні фізичні явища, і впевніться, що вона відповідає вимогам та обмеженням, встановленим для задачі моделювання.

9. Уточнення та модифікація моделі: виправлення або зміна моделі на основі аналізу результатів та валідації, якщо це необхідно.

10. Застосування моделі: використовуйте модель для прогнозування поведінки системи в різних умовах або для оптимізації процесів на основі отриманих результатів.

Цей алгоритм може бути модифікований залежно від конкретної задачі моделювання та особливостей системи, яку ви хочете дослідити.

Завдання: змоделювати фізичний процес, використавши в якості об'єкта моделювання будь-яке явище, процес або систему.

ЛАБОРАТОРНА РОБОТА 10
МОДЕЛЮВАННЯ МЕРЕЖ: A*-АЛГОРИТМ

Мета роботи: ознайомитися з особливостями моделювання транспортних мереж різної природи; вивчити особливості моделювання міських транспортних мереж.

Теоретичні відомості. Нехай потрібно знайти оптимальний маршрут у мережі, представленій у вигляді графу на рис. 10.1. 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 графу. Це відстань, взята по прямій, від усіх вершин графу 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 графу. 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).

Представимо тепер наведений описовий варіант, а також програмну реалізацію A*-алгоритму та шляхи його використання під час прокладання оптимальних маршрутів у різноманітних мережах, зокрема і міських транспортних. Розглянемо граф, представлений на рис. 10.1.

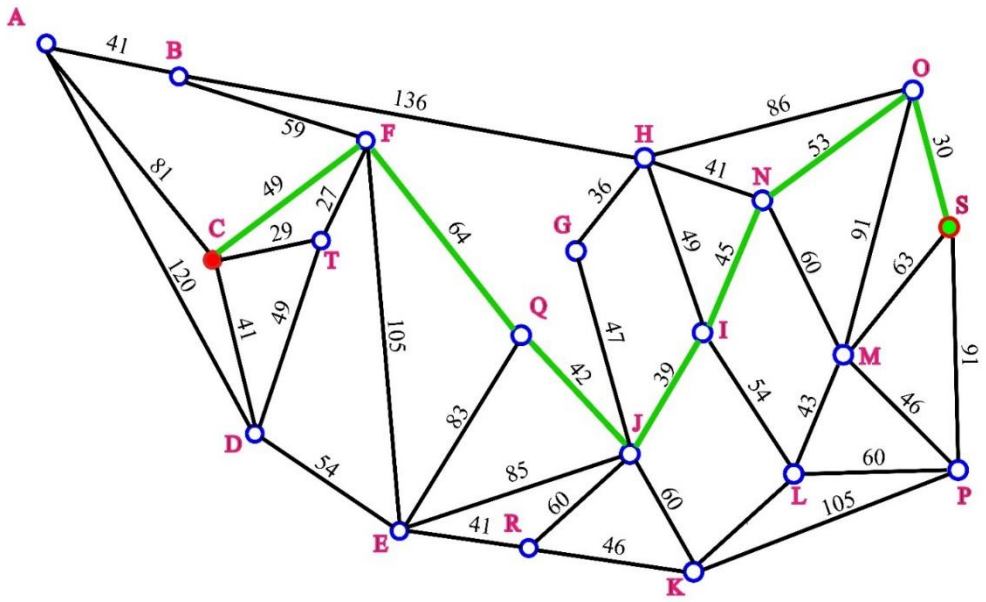


Рис. 10.1. Зважений граф, у якому за допомогою A-алгоритму знаходиться оптимальний маршрут між вершинами C (стартова вершина) і S (фінішна вершина). Цей маршрут виділений потовщеною зеленою ламаною лінією. Біля кожного ребра проставлені ваги, що відповідають геометрії задачі*

Припустимо, треба знайти оптимальний маршрут між вершинами C та S. Скористаємось A*-алгоритмом. Сформуємо таблицю, в 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 графу, представленого на рис. 10.1, оптимальний маршрут між вершинами C та S та його довжина прораховані та представлені у табл. 10.1.

Таблиця 10.1. 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 графу 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. 10.1 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. 10.1. Аналізуючи отримані результати, бачимо, що протяжність оптимального маршруту C→S складає 322 умовні одиниці. За допомогою табл. 10.1 можна легко відтворити оптимальний маршрут у графі (рис. 10.1). Для цього зауважимо, що у вершину S перехід відбувся з вершини O. У вершину O перехід був здійснений із N. Продовжуючи подальший аналіз, відтворюємо оптимальний маршрут C→ F → Q → J → I → N → O →S.

Звернемо увагу, що у табл. 10.1 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 10.1).

Лістинг 10.1

```
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 зі знайденим раніше за допомогою «ручного» алгоритму. Важливість A*-алгоритму полягає у тому, що відсікаються неперспективні варіанти маршрутів завдяки аналізу значень функції $f(x)$. Натомість величини $f(x)$ залежать від значень функції $h(x)$.

Завдання: прокласти оптимальний маршрут у графі, представленому на рис. 10.2.

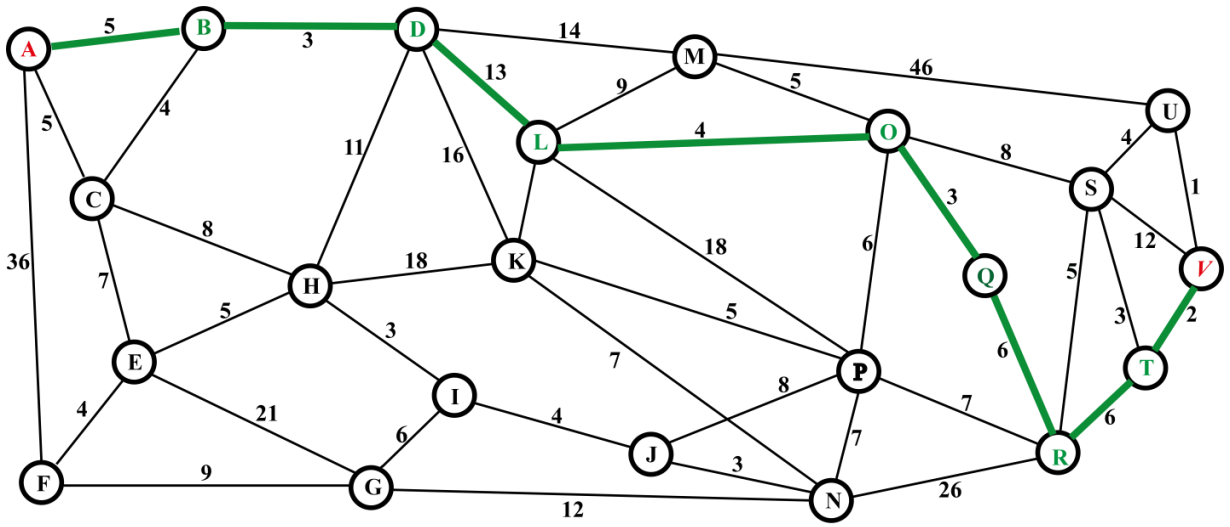


Рис. 10.2. Зважений граф, у якому з допомогою A^*-алгоритму прокладений оптимальний маршрут між вершинами A і V. Оптимальний маршрут руху з урахуванням ваг ребер на конкретний момент часу виділений та пролягає через перехрестя у послідовності

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

Вершини, між якими потрібно прокласти оптимальний маршрут, задаються викладачем. Під час виконання завдання необхідно дотримуватись такого алгоритму:

1. Ініціалізація: встановіть початкову вершину та кінцеву вершину. Початкова вершина матиме величину g (значення шляху від початкової вершини до поточної) рівну 0, тоді як оцінка вартості шляху до кінцевої вершини, яка ще не відома, може бути оцінена за допомогою функції оцінки *h* (евристична оцінка від поточної вершини до кінцевої).

2. Відкриті та закриті вершини: створіть дві множини – відкриті та закриті вершини. Початкова вершина додається до множини відкритих вершин.

3. Цикл пошуку: повторюйте доти, доки множина відкритих вершин не стане порожньою або не буде знайдено кінцеву вершину:

1) виберіть вершину з мінімальною загальною вартістю *f* ($f = g + h$);

2) перевірте, чи є ця вершина кінцевою. Якщо так, то ми знайшли шлях;

3) в іншому випадку переходьте до всіх сусідніх вершин поточної вершини та оновіть їх вартості *g* та *f*, якщо новий шлях коротший або кращий;

4) додайте поточну вершину до множини закритих вершин та видаліть її з множини відкритих.

4. Відтворення шляху: якщо кінцева вершина знайдена, відтворіть шлях, переходячи від кінцевої вершини до початкової за допомогою зворотних посилань.

5. Завершення: шлях буде знайдений, якщо множина відкритих вершин стане порожньою або якщо кінцева вершина буде знайдена.

Цей алгоритм ефективний, оскільки він використовує евристичну інформацію, щоб пришвидшити пошук, але водночас гарантує знаходження оптимального шляху.

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.

ДЛЯ НОТАТОК

Навчальне видання

Ніколюк Петро Карпович

МОДЕЛЮВАННЯ СИСТЕМ

Методичні вказівки для самостійної роботи
та виконання лабораторних робіт
здобувачами спеціальності 122 Комп'ютерні науки
освітньої програми «Комп'ютерні науки»

Редактор О. А. Солдатова
Технічний редактор Т. О. Важеніна-Гопрак

Підписано до друку 16.12.2024
Формат 60 × 84/16. Папір офсетний.
Друк – цифровий. Умовн. друк. арк. 4,88.
Тираж 30. Зам. 35.

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