НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Приладобудівний факультет Кафедра комп'ютерно інтегрованих технологій виробництва приладів

«До захисту допущено»

Завідувач кафедри _____

«16» червня 2025 р.

Дипломна робота

на здобуття ступеня бакалавра спеціальності – 151 Автоматизація та комп'ютерно-інтегровані технології

на тему: «Система управління і контролю переміщення роботів маніпуляторів автоматизованого складу»

Виконав:

Студент IV курсу, групи ПБ-12 Павленко Дмитро Сергійович _____

Керівник: асистент Заєць С.С. _____

Рецензент доцент к.т.н. Півторак Д.О. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Київ – 2025

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Приладобудівний факультет

Кафедра комп'ютерно-інтегрованих технологій виробництва приладів

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 151 «Автоматизація та комп'ютерно-інтегровані технології»

Освітньо-професійна програма «Комп'ютерно-інтегровані системи та технології в приладобудуванні»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Наталія СТЕЛЬМАХ

«16» червня 2025 р.

ЗАВДАННЯ

на дипломну роботу студенту

Павленко Д. С.

1. Тема роботи «Система управління і контролю переміщення, роботів маніпуляторів, автоматизованого складу.», керівник проєкту Заєць С.С, асистент, затверджені наказом по університету від «26» травня 2025 р. №1765-с

2. Термін подання студентом роботи «05» червня 2025 року

3. Вихідні дані до роботи: сумісність з операційними системами (MacOS, Linux, Windows).

4. Зміст роботи: Перелік скорочень, умовних позначень, термінів. Вступ. Розділ 1. Теоретичні основи побудови системи управління рухом роботів-маніпуляторів: 1.1. Сенсори виявлення перешкод: використання радарів; 1.2. Алгоритми навігації та планування маршрутів (A*, D*, інші). Розділ 2. Конструкторська частина: 2.1. Структура апаратно-програмних засобів системи; 2.2. Блок-схеми алгоритмів; 2.3. Конструкторські особливості: вибір обладнання. Розділ 3. Реалізація програми керування (Python): логіка, функції, алгоритми: 3.1.

Структура програми і модулі; 3.2. Алгоритм A* у коді; 3.3. Реалізація обходу динамічних перешкод; 3.4. Детальний опис функцій програми; 3.5. Приклад роботи програми. Висновки. Перелік використаних джерел.

5. Перелік графічного матеріалу: Структурна схема апаратно-програмного комплексу системи управління (А1). Блок-схеми алгоритмів навігації та перепланування маршруту (А1). UML-діаграми структури та взаємодії програмних модулів (А1). Схема розміщення сенсорів на платформі робота (А1). Графік роботи програми керування (А1).

6. Дата видачі завдання «06» квітня 2025 року

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області та огляд літературних джерел.	01.05.2025	
2	Формулювання вимог до системи	04.05.2025	
3	Розробка структурно-функціональної схеми	07.05.2025	
4	Висвітлення архітектури системи, навігації та WMS	13.05.2025	
5	Розрахунок енергоспоживання, пропускної здатності та окупності.	18.05.2025	
6	Оформлення пояснювальної записки та підготовка графічного матеріалу	27.05.2025	
7	Підготовка презентації та захист	01.06.2025	

Студент Дмитро ПАВЛЕНКО

Науковий керівник Сергій ЗАЄЦЬ

Пояснювальна записка

до дипломної роботи

на тему **Система управління і контролю переміщення, роботів маніпуляторів, автоматизованого складу**

Зміст

Перелік умовних позначень

A*	–	алгоритм пошуку шляху з евристикою
AGV	–	автоматизований керований транспортний засіб
API	–	інтерфейс програмування додатків
CAM	–	комп’ютерне керування виробництвом
CAD	–	автоматизоване проектування
CPU	–	центральний процесор
DB	–	база даних
Dijkstra	–	алгоритм Дейкстри для пошуку шляху
DS	–	датчик стану
GUI	–	графічний інтерфейс користувача
IoT	–	Інтернет речей
LiDAR	–	лазерний дальномір
MCU	–	мікро контролерний пристрій
ML	–	машинне навчання
M	–	кількість комірок/секторів складу
N	–	кількість роботів у системі
OBSTACLE	–	перешкода у навігаційній сітці
PLC	–	програмований логічний контролер
RAM	–	оперативна пам’ять
RFID	–	радіочастотна ідентифікація
RMS	–	середньоквадратичне відхилення
ROS	–	операційна система для роботів
SC	–	система керування
SCADA	–	система диспетчерського керування і збору даних
SL	–	складська логістика

SLAM	–	одночасна локалізація та побудова карти
TCP/IP	–	мережевий протокол передачі даних
Tk	–	період виконання циклу керування
UDP	–	протокол користувацьких дейтаграм
UML	–	універсальна мова моделювання
V	–	швидкість руху робота
Δt	–	крок дискретизації часу
Δx, Δy	–	кроки по координатах X та Y
g(n)	–	вартість шляху від старту до вузла n
h(n)	–	евристична оцінка відстані
f(n)	–	оціночна функція (сума g(n) і h(n))

ВСТУП

Автоматизація складів із використанням роботів-маніпуляторів та мобільних роботів нині стрімко поширюється у логістиці та виробництві. Підприємства впроваджують роботизовані системи для підвищення продуктивності, точності і безпеки операцій складу [16]. Завдяки автономним мобільним роботам (AMR) та автоматизованим транспортним засобам (AGV) компанії можуть автоматично переміщувати товари, виконувати комплектацію замовлень та оптимізувати складські процеси без безпосередньої участі людини. За оцінками аналітиків, ринок мобільних роботів має стрімке зростання – до 2027 року 75% компаній будуть використовувати “кібер-фізичні” технології (зокрема мобільних роботів) на складах, а щорічні продажі мобільних роботів зростуть з 4.5 млрд дол. у 2023 до понад 14 млрд дол. у 2027 [5]. Очікується, що у 2024 році буде продано близько 200 тисяч мобільних роботів (на 25% більше, ніж у 2023), а в 2027 – вже до 700 тисяч одиниць [5]. Такі тенденції підтверджують актуальність розвитку систем управління і контролю руху роботів на складах.

Впровадження роботизований систем дозволяє розв’язати ряд проблем традиційної складської логістики. По-перше, роботи можуть працювати цілодобово без перерв, що підвищує пропускну здатність складу та скорочує час виконання операцій. По-друге, автоматизація зменшує залежність від людського фактору та кадрових проблем: роботизовані візки та маніпулятори не беруть лікарняних, не втомлюються і можуть виконувати монотонні завдання без падіння продуктивності [1]. По-третє, використання роботів підвищує безпеку праці – автономні системи оснащені сенсорами, що запобігають зіткненням, і беруть на себе важкі або небезпечні операції, зменшуючи ризик травматизму працівників [1][16]. Крім

того, роботи дозволяють ефективніше використовувати складські площі: автономні штабелери можуть працювати у вужчих проходах та на більшу висоту, ніж це безпечно для людини, що економить простір на складі [1].

Таким чином, дослідження і розробка системи управління та контролю переміщення роботів-маніпуляторів для автоматизованого складу є надзвичайно актуальними з огляду на сучасні тенденції Industry 4.0. Така система має забезпечувати надійну навігацію роботів у динамічному середовищі складу, уникнення перешкод (у тому числі рухомих), координацію дій декількох роботів, а також інтегруватися з загальною системою управління складом (WMS) для отримання завдань та звітності. У даній роботі представлено комплексне дослідження та розробку такої системи – від теоретичного обґрунтування вибору сенсорів і алгоритмів, до практичної реалізації програмного забезпечення на Python та економічного аналізу доцільності впровадження.

Метою роботи є створення системи керування переміщенням роботів-маніпуляторів автоматизованого складу, що забезпечує ефективну та безпечну навігацію з використанням сучасних сенсорів (радарів) для виявлення перешкод і оптимальних алгоритмів планування маршрутів (A*, D* тощо). Для досягнення поставленої мети необхідно вирішити такі завдання:

Провести аналіз сенсорних систем для виявлення перешкод у робототехнічних комплексах складу і обґрунтувати вибір радарних датчиків, визначити принцип їх роботи та оптимальне розміщення на роботі.

Дослідити алгоритми навігації та прокладки шляхів (A*, Dijkstra, D*, RRT тощо), оцінити їх ефективність та придатність до динамічного середовища; навести порівняльний аналіз з формулами і теоретичними оцінками.

Розробити концепцію системи керування рухом: архітектуру програмно-апаратних засобів, логіку роботи програми керування роботом, включаючи обробку даних радарних сенсорів, побудову карти та маршруту, уникнення перешкод у реальному часі.

Реалізувати програмний модуль керування на мові Python, детально пояснити його функції, структури даних, алгоритми (маршрутизація, обхід перешкод, обробка динамічних перешкод).

Розробити необхідні діаграми (блок-схеми алгоритмів, UML-діаграми класів, станів, послідовності) для візуалізації структури системи і процесів у ній.

Провести економічне обґрунтування проекту: оцінити вартість обладнання (роботів, сенсорів, інфраструктури), експлуатаційні витрати на обслуговування, розрахувати очікуваний термін окупності інвестицій, порівняти з альтернативою ручної праці.

У роботі використано сучасні наукові публікації, технічну документацію та матеріали виробників обладнання. Це забезпечує актуальність і достовірність представлених даних та рішень. Робота складається з чотирьох розділів: теоретичного обґрунтування, конструкторської частини, розділу реалізації програмного забезпечення та економічного обґрунтування, а також містить вступ, висновки та список використаних джерел.

АКТУАЛЬНІСТЬ

Актуальність створення системи управління для роботизованого складу зумовлена кількома ключовими чинниками сучасної промисловості і логістики. По-перше, стрімке зростання обсягів e-commerce та вимоги до швидкої доставки товарів стимулюють автоматизацію складів. Людські ресурси вже не встигають ефективно обробляти великі потоки товарів, особливо в умовах дефіциту кадрів і високої плинності персоналу. Роботи допомагають вирішити проблему нестачі робочої сили, беручи на себе рутинні переміщення вантажів та комплектацію замовлень [1]. Це не лише підвищує продуктивність, але й покращує утримання кадрів – працівники звільняються від виснажливої одноманітної роботи і можуть бути задіяні на інтелектуальніших завданнях [1].

По-друге, автоматизація є відповіддю на економічні виклики: вартість праці та утримання складів невпинно зростає. За даними досліджень, протягом 2006–2016 рр. середня погодинна оплата праці на складах зросла на 16%, а витрати на оренду складських площ – на 28% [1]. В таких умовах бізнес шукає шляхи зниження операційних витрат. Роботи здатні суттєво скоротити витрати на персонал, адже один робот може замінити кількох працівників і працювати без перерв 24/7. Як показано у кейс-стаді, впровадження автоматизованих складських роботів може дозволити компанії перейти з 2 змін роботи на 3 зміни без пропорційного збільшення штату: наприклад, на одній зі складів кількість працівників на нічній зміні вдалося зменшити з 35 до 2, решту завдань виконують роботи [13]. У результаті підприємство отримало можливість працювати цілодобово з мінімальними додатковими затратами на персонал.

По-третє, підвищення швидкості та точності операцій. Роботи-маніпулятори можуть переміщувати товари швидше і з меншою кількістю помилок, ніж люди. Сучасні системи типу “goods-to-person” здатні збільшити продуктивність

відбору замовлень на сотні відсотків. Для прикладу, упровадження системи Goods-to-Person дозволяє підвищити швидкість комплектації до 400 одиниць на годину на одного оператора (проти ~50 у традиційній схемі), скоротивши при цьому кількість необхідних комплектувальників з 100 до 13 (на 87%) [6]. Таким чином, роботизація складу веде до кардинального підвищення ефективності бізнес-процесів.

По-четверте, безпека і зменшення травматизму – одна з важливих причин автоматизації. Склад – потенційно небезпечне середовище: інтенсивний рух навантажувачів, підйом важких вантажів, робота на висоті тощо. Автономні роботи обладнані сучасними сенсорами та системами запобігання зіткнень, що практично виключає аварії за їх участі [16]. Це захищає як персонал, так і товарно-матеріальні цінності від пошкоджень. Наприклад, роботизовані навантажувачі з лазерними або радарними сканерами постійно моніторять простір довкола себе і негайно зупиняються при виявленні людини чи перешкоди на шляху. Як наслідок, впровадження таких систем значно скорочує кількість інцидентів на складі, що має не тільки соціальний, а й економічний ефект (менші витрати на компенсації, простої тощо).

Нарешті, актуальність зумовлена розвитком технологій, які роблять можливим ефективне керування парком роботів. Виникають нові методи навігації та координації, алгоритми оптимального планування маршрутів у режимі реального часу, системи управління флотом роботів (Fleet Management Systems). Сучасні алгоритми планування руху (такі як A*, D* Lite, різноманітні модифікації пошуку, а також алгоритми на основі машинного навчання) дозволяють будувати оптимальні траєкторії навіть в умовах динамічних змін (перестановка товарів, поява/зникнення перешкод, рух інших роботів) [8]. З’явилися доступні і надійні сенсори – окрім традиційних лідарів та камер, все ширше застосовуються радарні

датчики, що ефективно працюють в різних умовах довкілля (пил, дим, темрява) [2]. Інтеграція інформації з декількох сенсорів (sensor fusion) та використання штучного інтелекту (наприклад, глибокого навчання для розпізнавання об’єктів і прийняття рішень) піднімає автономність та надійність роботів на новий рівень [2]. Отже, поєднання прогресу в галузі сенсорики та алгоритмів керування створює підґрунтя для побудови ефективних систем контролю руху роботів-маніпуляторів на складі, що і визначає актуальність даного дослідження.

Розділ І. Теоретичні основи побудови системи управління рухом роботів-маніпуляторів

1.1. Сенсори виявлення перешкод: використання радарів

Для автономної навігації роботів на складі критично важливим є здатність виявляти перешкоди на шляху та безпечно їх об’їжджати. Серед різноманітних типів сенсорів (камери, лідари, ультразвукові дальноміри тощо) особливу увагу привертають радарні сенсори (Radio Detection and Ranging). Радар випромінює радіохвилі й аналізує відбитий сигнал (ехо) для визначення відстані до об’єктів та їх відносної швидкості [2]. Принцип роботи радара полягає у вимірюванні часу поширення електромагнітного імпульсу до перешкоди і назад, а також доплерівського зсуву частоти відбитого сигналу. На основі цих даних обчислюються дистанція та напрямок (кут) на об’єкт, а при наявності доплерівського зсуву – і швидкість руху об’єкта [3][2]. На відміну від оптичних сенсорів, радіохвилі мало залежать від освітленості та прозорості середовища, тому радар ефективно працює за умов пилу, диму, дощу чи туману, де камера або лідар можуть “осліпнути” [2]. Це робить радарні датчики привабливими для застосування в складських приміщеннях, де може підійматись пил, або для вуличних складських майданчиків під відкритим небом.

Таблиця 1.1 Порівняння характеристик основних сенсорів для роботизованої навігації

Параметр	**Радар**	**Лідар**	**Камера**
Принцип роботи	Радіохвилі	Лазер	Зображення
Діапазон	20–30 м	10–50 м	2–10 м

Чутливість до пилу, диму	Низька	Висока	Висока
Визначення швидкості	Так	Ні	Обмежено
Вартість	Середня	Висока	Низька
Робота в темряві	Так	Так	Може бути погано
Типові моделі	OndoSense, Acconeer	SICK LMS, RPLIDAR	Intel Realsense

Радарні сенсори бувають різних типів. У робототехніці та транспортних засобах широко використовуються мікрохвильові FMCW-радари (Frequency Modulated Continuous Wave) міліметрового діапазону (наприклад, на частотах ~24 ГГц чи 60–80 ГГц). Вони можуть бути компактними і забезпечувати далекість виявлення на десятки метрів. Сучасні автомобільні радарні модулі, які адаптуються і для роботів, здатні виявляти перешкоди на відстані 20–30 м і більше. Приміром, промисловий радар OndoSense reach WA забезпечує діапазон виявлення від 0,1 до 25 метрів, причому має дуже малу "сліпу зону" (0,1 м) та високу швидкодію 200 вимірів за секунду [4][4]. Це означає, що навіть швидко з’являючіся перешкоди будуть вчасно виявлені. Радари також здатні надійно “бачити” як статичні, так і рухомі об’єкти будь-якої природи (люди, техніка, стіни, відокремлені предмети) незалежно від їх кольору чи прозорості [4].

Важливим питанням є розміщення радарних сенсорів на роботі. Оскільки типовий радар має обмежений кут огляду (наприклад, ±30°...±60° по горизонталі) [11], для повного охоплення простору навколо робота може знадобитися кілька радарів. Зазвичай на мобільних роботах сенсори розташовують спереду (для контролю шляху вперед) та позаду, іноді – по боках або по діагоналі, щоб утворити

круговий огляд. Деякі моделі роботів оснащуються поворотними або обертовими радарами, які сканують 360° навколо машини (аналогічно лідару). Наприклад, в експерименті з топографічного мапування місцевості на основі радару використовували комбінацію стаціонарного 77 ГГц FMCW радара з кутом огляду ~120° та панорамного 360° обертового радара, досягнувши повного покриття оточення транспортного засобу [2].

Для складського робота оптимально встановити радари на висоті, близькій до центру мас перешкод (наприклад, ~0,3–0,5 м над підлогою для виявлення ніжок стелажів, палет тощо). Також можуть використовуватися багатоярусні схеми сенсорів: один радар оглядає нижній рівень, другий – верхній (щоб бачити нависаючі частини вантажів або паліт).

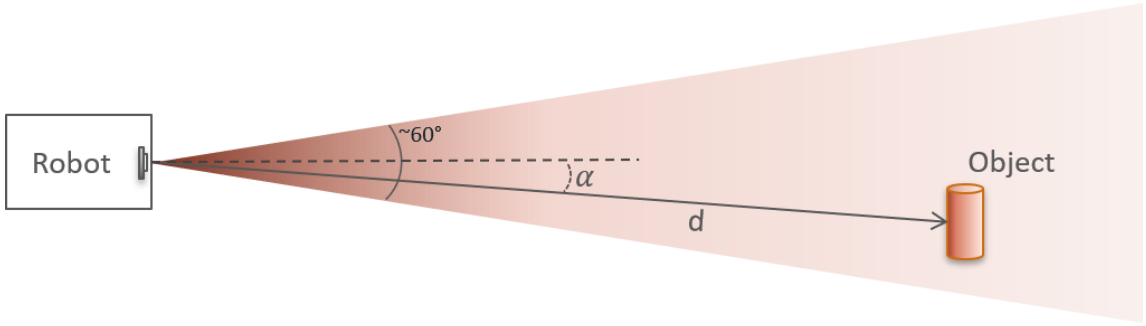

Рис. 1: Схема виявлення перешкоди радарним сенсором, встановленим на роботові. Радар випромінює радіохвилі, що відбиваються від об’єкта; за часом повернення та зміною частоти сигналу обчислюються відстань d та кут α до перешкоди [11]. Поле огляду сенсора обмежене кутом приблизно 60° по горизонталі (відповідно до половинної потужності сигналу), тому для кругового огляду потрібна комбінація кількох сенсорів або обертання радара.

Однією з переваг радарів є можливість формувати динамічні зони безпеки навколо робота. Більшість сучасних сенсорів дозволяють програмно налаштувати декілька зон виявлення – наприклад, дальню “попереджувальну” зону і ближчу “аварійну” зону. При попаданні об’єкта в попереджувальну зону робот може

автоматично знизити швидкість, а якщо перешкода наблизиться до критично малої відстані (аварійна зона) – повністю зупинитися [12][12]. Такі багатозонні системи широко використовуються з лазерними сканерами, і аналогічно можуть бути реалізовані на радарних сенсорах. Згаданий вище радар OndoSense reach WA підтримує до 4 незалежно налаштовуваних зон контролю, перемикання яких інтегрується з системою керування роботом (наприклад, через PLC) для поетапного сповільнення і зупинки руху при наближенні до перешкоди [4][4]. Така функціональність дає змогу підвищити безпеку і уникнути зіткнень навіть у сценаріях спільної роботи роботів з людьми (колаборативне середовище).

Рис. 1.2. Сучасний мобільний робот-маніпулятор

Резюмуючи, використання радарів у якості сенсорів для виявлення перешкод на автоматизованому складі є перспективним рішенням завдяки їх надійності в різних умовах, великій дальності дії та можливості вимірювання швидкості об’єктів. В наступних розділах буде розглянуто, як дані радарних сенсорів можуть використовуватися алгоритмами навігації робота для побудови маршруту і уникнення зіткнень.

1.2. Алгоритми навігації та планування маршрутів (A, D, інші)

Планування маршруту – центральне завдання системи управління рухом робота-маніпулятора. Необхідно визначити оптимальний шлях від поточного положення робота до цільового (місця забору або розміщення вантажу) з урахуванням статичних та динамічних перешкод. Цю задачу зазвичай розв'язують методами пошуку шляхів у графі, де простір робочої зони представлений дискретно (решітка комірок, граф вузлів-точок тощо). Найвідомішими класичними алгоритмами є алгоритм Дейкстри та алгоритм A* (A-зірка), а для динамічного середовища – його модифікації, як-от D* (DynamicA*) або D* Lite.

Алгоритм Дейкстри знаходить найкоротші шляхи у зваженому графі від заданого стартового вузла до всіх інших вузлів. Він гарантує оптимальний (найкоротший) шлях, але в найгіршому випадку змушений перебрати всі можливі вузли графа. Комплексність Дейкстри становить $O(\frac{V}{\log V} + E)$ при використанні пріоритетної черги (де V – кількість вузлів, E – ребер) і практично зростає дуже швидко для великих графів [26][27]. У контексті навігації роботів алгоритм Дейкстри можна застосувати для пошуку шляху на сітці комірок (наприклад, 2D-решітці складського плану); проте через відсутність цільового "напрямку" він досліджує багато зайвих комірок.

Алгоритм A* – це вдосконалення Дейкстри, яке додає евристичну оцінку відстані до цілі. В A* кожна потенційна позиція (вузол графа) оцінюється за функцією:

$$f(n) = g(n) + h(n)$$

де $g(n)$– вартість шляху від старту до вузла nnn, а $h(n)$– евристична оцінка (heuristic) відстані від вузла nnn до цільового вузла [8]. Евристика $h(n)$має бути адмісивною (не перевищувати реальну найкращу відстань), щоб A* гарантовано знайшов оптимальний шлях. Завдяки евристиці пошук спрямовується в бік цілі,

не обходячи непотрібно всю область. На рис. 2 схематично показано різницю: алгоритм Дейкстри (синім кольором) рівномірно “розливається” від старту на всі боки, тоді як A* (фіолетовим) цілеспрямовано рухається до мети, ігноруючи області, що явно лежать осторонь найкоротшого шляху [7].

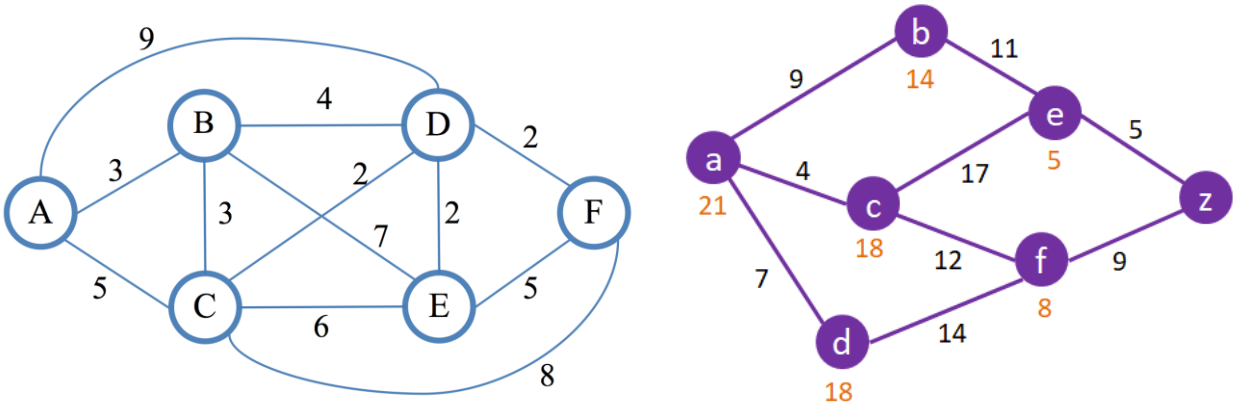

Рис 1.2. Алгоритми Дейкстри та A*

A* має такі властивості: він завжди знаходить оптимальний шлях, якщо евристика адекватна (адмісивна і монотонна) [8], і зазвичай робить це значно швидше, ніж Дейкстра, оскільки обмежує область пошуку. В практичних задачах навігації A* демонструє вищу ефективність, ніж алгоритм Дейкстри [7]. У середньому, A* відвідує набагато менше вузлів, особливо на великих картах, де ціль далеко від старту. Теоретично, в найгіршому випадку складність A* така ж, як у Дейкстри (оскільки може розглянути всі вузли, якщо евристика погана або відсутня), але за правильно підібраної евристики пошук є близьким до лінійного вздовж оптимального шляху.

Основна формула алгоритму A* вже наведена (функція $f(n) = g(n) + h(n)$). Евристика $h(n)$часто береться як евклідова або мангеттенська відстань до цілі (для решітки), що забезпечує допустимість. Алгоритм використовує відкритий список (open set) вузлів для розгляду, звідки щоразу вибирає вузол з мінімальним значенням $f(n)$, ітеративно його розгортає (генерує сусідів). Так триває,

поки цільовий вузол не буде вибрано для розгортання або поки open set не спорожніє (ціль недосяжна). В результаті будується або оптимальний маршрут, або робиться висновок про неможливість досягти цілі. Слід зазначити, що недоліком A* є підвищені вимоги до пам’яті – алгоритм зберігає у пам’яті відкритий список і вже відвідані вузли (закритий список), що на великих графах може бути проблемою [8]. У складних середовищах з великою кількістю вузлів A* також може працювати повільно, якщо евристика недостатньо інформативна [8].

Для покращення ефективності існують численні варіації A*, які застосовують:

- Оптимізацію евристичної функції – наприклад, пошук з перемінною метрикою, евристика на основі потенціальних полів тощо [7].
- Двобічний пошук (bidirectional A*), коли пошук ведеться одночасно від старту і від фінішу назустріч – це може вдвічі зменшити час, хоча складність реалізації зростає [7].
- Додавання згладжування шляху (після отримання результату A* часто застосовують алгоритми спрощення траєкторії, усунення зайвих поворотів, що покращує прохідність маршруту роботами) [7].
- Ітеративні та динамічні модифікації для роботи в режимі реального часу, про що йтиметься далі.

Таблиця 1.2. порівняння алгоритмів Дейкстри та A*

Параметр	**Алг. Дейкстри**	**Алг. A***
Метод пошуку	Обходить всі вузли рівномірно, враховуючи лише вартість шляху (g)	Використовує вартість $g + h$: враховує відстань до цілі, шукає більш вибірково
Евристична інформація	Не використовується (h(n)=0)	Потрібна адмісивна евристика $h(n)$
Оптимальність	Гарантовано найкоротший шлях	Гарантовано найкоротший шлях за умови коректної евристики
Вимоги до пам’яті	Високі (треба перевірити всі вузли)	Високі (потрібно зберігати open/closed списки)
Ефективність (час)	Низька при великому графі (повний перебір)	Вища за рахунок спрямованого пошуку (менше вузлів)
Динамічне середовище	Не підходить (потрібний перезапуск при зміні карти)	Базова версія не підходить, потрібні модифікації (D*, D* Lite)

Як видно з таблиці, алгоритм A* зазвичай є кращим вибором для планування шляху робота, оскільки при незначному ускладненні (необхідність задавати евристику) він виграє у швидкодії. Дейкстра може мати переваги в окремих випадках – наприклад, коли потрібно знайти шляхи до всіх цілей (тоді перевага A* нівелюється) або коли немає прийнятної евристики. Але в задачі навігації до конкретної точки A* демонструє вищу ефективність в більшості практичних ситуацій [7].

Для динамічного середовища, де перешкоди можуть з’являтися або зникати під час руху робота, стандартні алгоритми пошуку шляхів потрібно адаптувати. Простий підхід – перезапускати A* щоразу при виявленні нової перешкоди, але

це може бути повільно. Тому були розроблені алгоритми, що поновлюють розрахунок шляху інкрементально. Приклад – алгоритм D* (Dynamic A*), відомий також як D* Lite. Він був спеціально створений для умов, коли після початкового прокладення маршруту змінюється інформація про прохідність комірок (перешкоди) [8]. D* Lite використовує результати попереднього пошуку і оновлює тільки ту частину шляху, яку зачепили зміни, завдяки чому значно швидше реагує на динаміку оточення порівняно з повним перерахунком A* [8]. По суті D* виконує A* “навпаки” – від цілі до старту – і підтримує дві оцінки вартості для кожної комірки (приблизну і остаточну). При надходженні інформації про нову перешкоду, алгоритм підіймає “помітки” вартості уздовж заторкнутих шляхів і повторно розраховує лише потрібні вузли.

Окрім A* та його похідних, існують інші підходи до планування маршрутів роботів. Можна відзначити:

Алгоритми на основі потенційних полів – розглядають робота як точку в штучному “полі сил”: ціль притягує, перешкоди відштовхують. Робот рухається по напрямку сумарної сили. Цей метод дуже швидкий для обчислення, але може страждати від проблеми локальних мінімумів (робот може “застрягти” між перешкод) і не гарантує знаходження глобально оптимального шляху.

Пробабілістичні алгоритми (побудова карти простору за вибірками) – наприклад, PRM (Probabilistic Roadmap) та RRT (Rapidly-exploring Random Tree). PRM генерує випадкові точки в вільному просторі, з’єднує їх у граф і потім шукає шлях по графу [7]. RRT будують дерево від старту, розповсюджуючись випадковими напрямками, доки не охоплять простір або не знайдуть ціль. Ці алгоритми добре працюють в просторі високої розмірності (наприклад, для роботизованих маніпуляторів з багатьма ступенями вільності), проте для задач навігації мобільного робота на площині вони часто поступаються A* в оптимальності шляху. Згі-

дно з деякими дослідженнями, метод A* дає коротші та більш оптимальні траєкторії, ніж базовий RRT, хоч і поступається RRT у гнучкості для непередбачуваних середовищ [28]. Існують гібридні підходи (наприклад, комбінація RRT для швидкого охоплення області та Дейкстри/A* для уточнення шляху на сітці – стратегія RRT-Dijkstra [2]).

Алгоритми машинного навчання – з появою потужних обчислювачів деякі завдання навігації передають нейронним мережам або методам підкріплення. Приміром, глибоке навчання з підкріпленням (Deep Reinforcement Learning, DRL) може навчити робота об'їжджати перешкоди на основі досвіду. У поєднанні з класичними методами (наприклад, використання радарів для побудови ознак середовища і нейромережі для прийняття рішень) отримують досить надійні системи [2]. Однак такі системи є складнішими у розробці та верифікації, тому в інженерній практиці для маршрутизації частіше застосовують перевірені алгоритми типу A*.

Таблиця 1.3. Порівняльні характеристики алгоритмів планування маршруту

Алгоритм	**Метод пошуку**	**Евристика**	**Оптимальність**	**Вимоги до пам'яті**	**Динамічне середовище**
Дейкстра	Обхід усіх вузлів	Ні	Так	Високі	Ні
A*	g(n) + h(n)	Так	Так	Високі	Потрібні модифікації

D* Lite	Інкреме-нтально	Так	Так	Середні	Так
RRT, PRM	Випад-ко-вий/ста-тистич-ний	Част-ково	Ні	Низькі	Так

Таким чином, для розв’язання задачі планування руху роботів-маніпуляторів на складі доцільно використати алгоритм A* як основний інструмент пошуку оптимального шляху по карті складу. Його варто доповнити механізмами динамічного оновлення маршруту (алгоритм D* або перезапуск A* на ходу) для врахування раптово виникаючих перешкод. Далі в конструкторській частині розглянемо, як ці алгоритми інтегруються в загальну систему управління роботом, а в розділі реалізації – як вони застосовані в програмі на Python.

Розділ ІІ. Конструкторська частина

У цьому розділі розроблено архітектуру системи управління і контролю переміщення роботів-маніпуляторів автоматизованого складу. Система охоплює як апаратні компоненти (сам робот, сенсори, контролери), так і програмні модулі (алгоритми навігації, карти, інтерфейси). Результатом є цілісна схема, що показує, як дані від сенсорів (радарів) перетворюються на команди для приводів робота, забезпечуючи автономний рух за визначеним маршрутом і уникнення зіткнень.

2.1. Структура апаратно-програмних засобів системи

Робот-маніпулятор автоматизованого складу умовно складається з двох основних частин:

- Мобільна платформа (база робота) – забезпечує переміщення робота по складу. Це може бути колесна платформа (диференційного або омнідирекційного типу, навантажувач тощо) з мотор-приводами для руху.
- Маніпулятор (роботизована рука) – встановлений на мобільній платформі, виконує операції захоплення вантажів, укладання тощо.

У контексті нашої задачі головну увагу приділено переміщенню, тому розглядаємо переважно мобільну платформу і систему її навігації. Маніпулятор може мати власну підсистему керування для виконання операцій, яка синхронізується з рухом платформи (наприклад, платформа довозить маніпулятор до стелажа, після чого активується цикл роботи маніпулятора).

Апаратна архітектура системи наведена та основні компоненти:

- Сенсори навігації:
  - Радарні датчики перешкод (описані в розд.1.1) – встановлені спереду і позаду платформи для огляду в напрямку руху та при маневрах. В нашій системі використано, наприклад, 2 радарні модулі 60 ГГц, кожен з кутом огляду ~120°, що разом дає майже круговий огляд 240° попереду (закриття "мертвих зон" по боках забезпечується перекриттям полів). Радарні сенсори підключені до бортового контролера через інтерфейс (наприклад, CAN або UART/RS485) і постійно передають інформацію про відстані до найближчих об'єктів у своїх секторах.
  - Допоміжні сенсори: Однієї радарної системи може бути недостатньо для повної навігації, тому часто додаються лідар (2D лазерний сканер для точної локалізації по відбитках стін і стелажів) або камера/глибокова камера для розпізнавання маркерів, QR-кодів місць зберігання тощо. У нашому проекті припускається наявність 2D лідара для локалізації та побудови карти (SLAM), але основний сенсор для уникнення перешкод – радар. Також робот оснащений одометрією (енкодери на колесах) для відстеження власного переміщення.
- Бортовий контролер – вбудований комп'ютер або промисловий контролер, що виконує програму керування. Він отримує дані сенсорів, запускає алгоритми навігації та планування шляху, генерує команди керування приводами. Наприклад, на роботі може стояти промисловий контролер під керуванням реального часу (PLC) з модулем обробки радарів, або одноплатний комп'ютер (типу Raspberry Pi, NVIDIA Jetson) з Linux та ROS. У нашій реалізації програму написано на Python, припускаючи наявність достатньо потужного контролера для її виконання.

- Приводи (актуатори) – електродвигуни коліс (або гусениць/механізмів руху). Вони підключені до контролера через драйвери, які забезпечують замкнене регулювання швидкості/позиції. Контролер через PWM або CAN-команди встановлює потрібну швидкість кожного колеса.
- Інтерфейси зв'язку – робот може отримувати завдання з серверної системи (наприклад, з системи управління складом – WMS). Для цього використовується бездротовий зв'язок (Wi-Fi) або інші протоколи. Також робот може обмінюватися даними з іншими роботами (щоб уникати заторів, координувати роз'їзд).

У нашому проекті основний фокус – навігація, тому модуль маніпулятора розглядається поверхнево. Для повноти: він може використовувати окремі сенсори (камери глибини на руці, силомоментні датчики хвата), та окремі алгоритми (обчислення кінематики маніпулятора, планування руху в просторі конфігурацій), що виходить за рамки даної роботи.

2.2. Блок-схеми алгоритмів

Для наочності розроблені діаграми, що описують логіку роботи системи і взаємодію модулів. Представлена блок-схема алгоритму навігації робота від моменту отримання завдання до його виконання. Алгоритм можна описати кроками:

1.Отримання завдання: робот отримує від центральної системи координати точки призначення (наприклад, забрати палету з комірки А5 і доставити до зони відправки). Завдання надходить через бездротовий зв'язок або встановлюється оператором вручну.

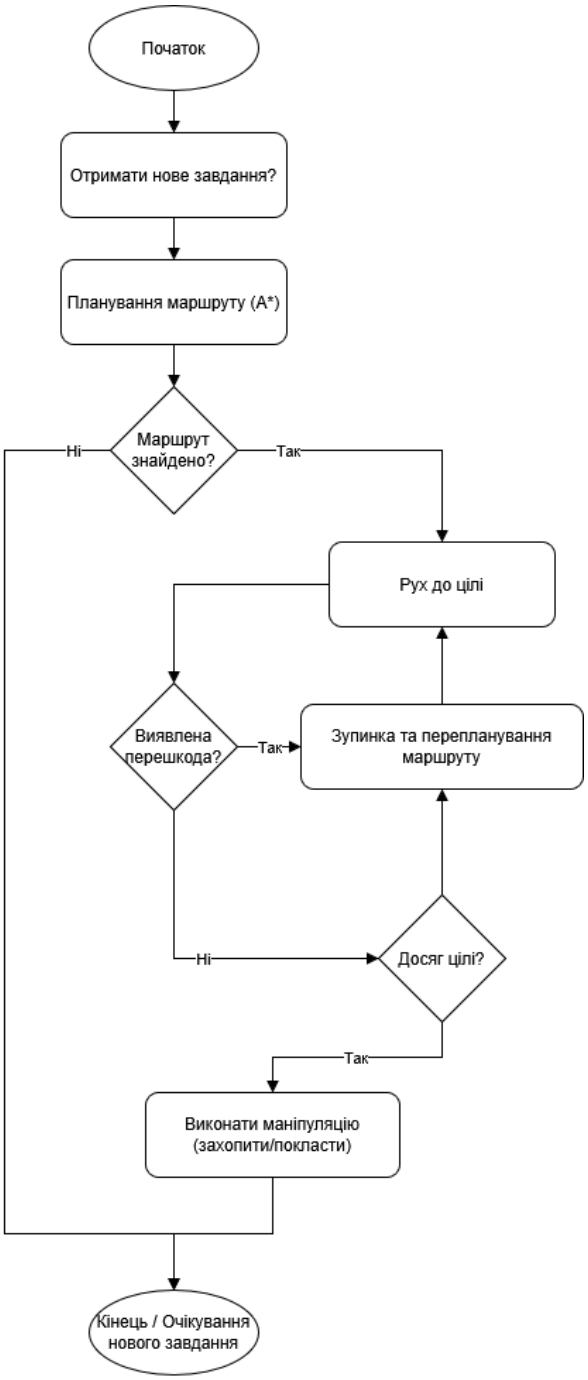

Рис. 2.1. Блок-схема головного алгоритму навігації робота

2.Побудова маршруту: модуль планування (A*) будує оптимальний шлях на карті складу від поточної позиції до цілі. Шлях – це послідовність ключових

точок (waypoints) або відрізків між поворотами. Якщо шлях не знайдено (ціль недосяжна), алгоритм сигналізує про помилку і завершується.

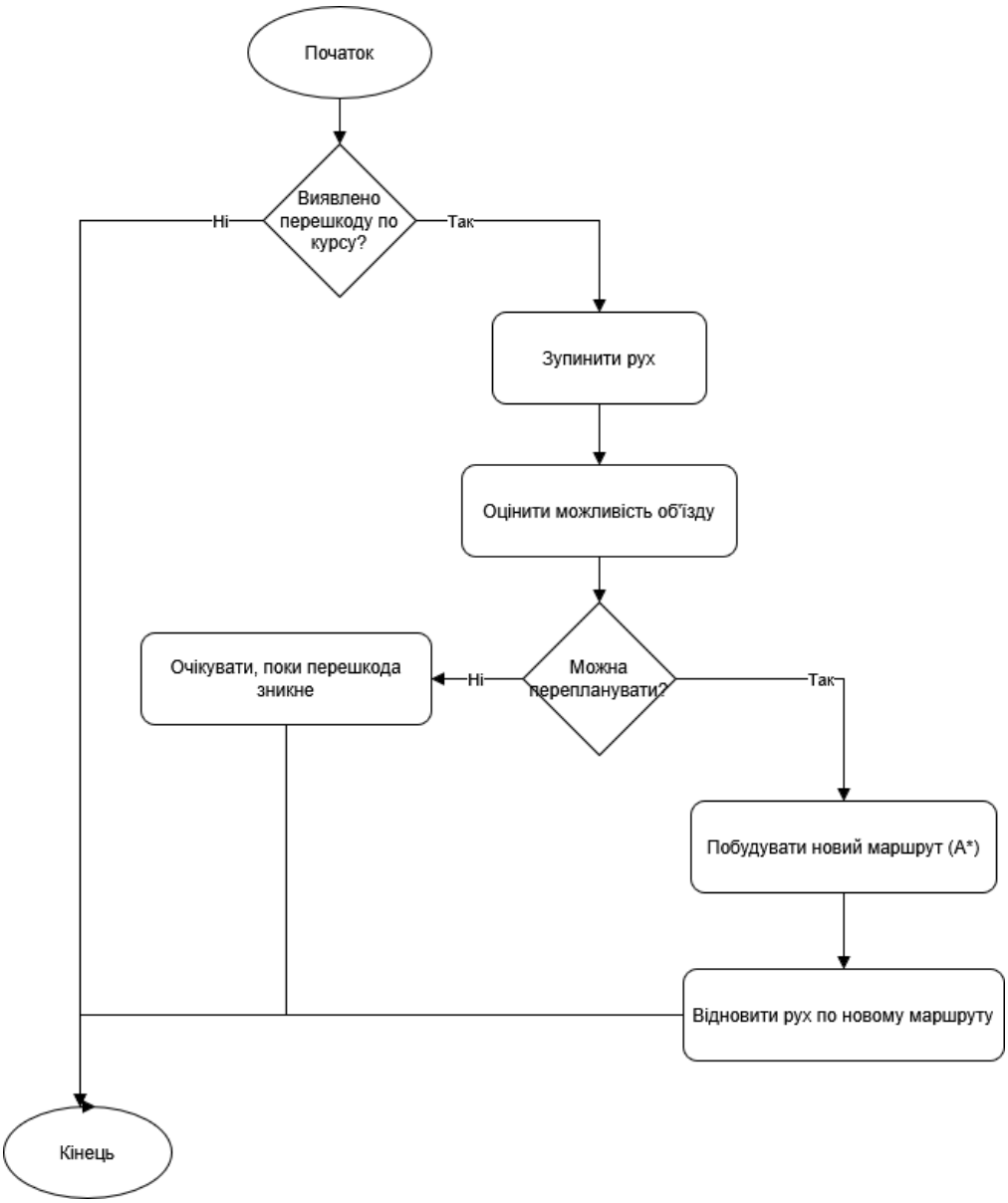

Рис. 2.2. Блок-схема планування маршруту (A*)

3.Рух по маршруту: модуль керування рухом починає посилати команди двигунам – робот рушає за першим відрізком маршруту. При цьому постійно контролюється положення робота (через локалізацію) і коригується напрямок, щоб не відхилятися від траєкторії.

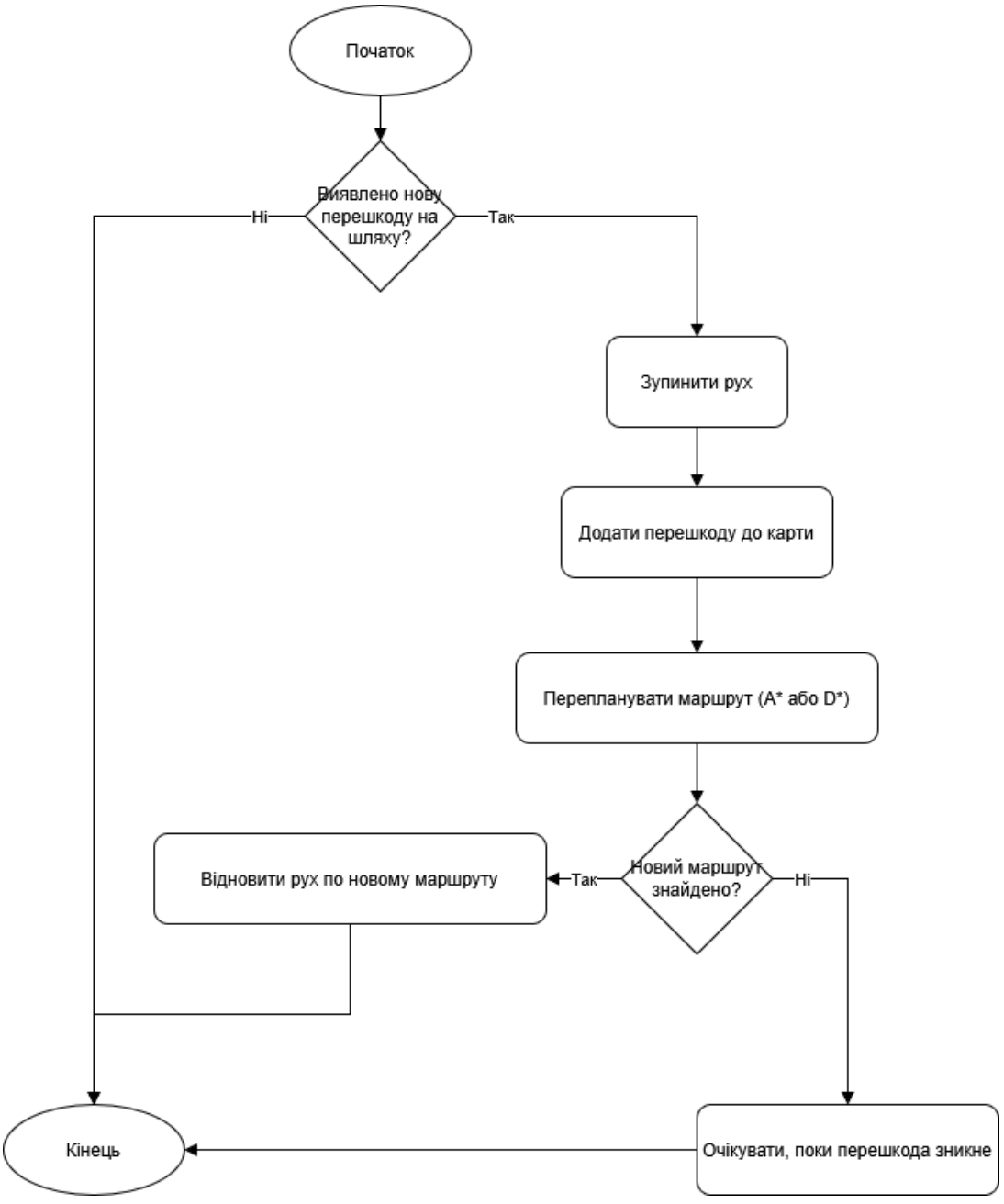

Рис. 2.3. Блок-схема динамічного перепланування маршруту

4.Моніторинг перешкод: паралельно працює цикл перевірки сенсорів (радарів). Кожні, скажімо, 50 мс зчитуються дані радарів про відстані. Якщо виявлено об’єкт ближче певного порогу, виконується реакція:

- Якщо об’єкт у дальній попереджувальній зоні (наприклад, <2 м, але >1 м), робот знижує швидкість на 50%.
- Якщо об’єкт у критичній близькій зоні (<1 м попереду) – негайно зупинка робота.

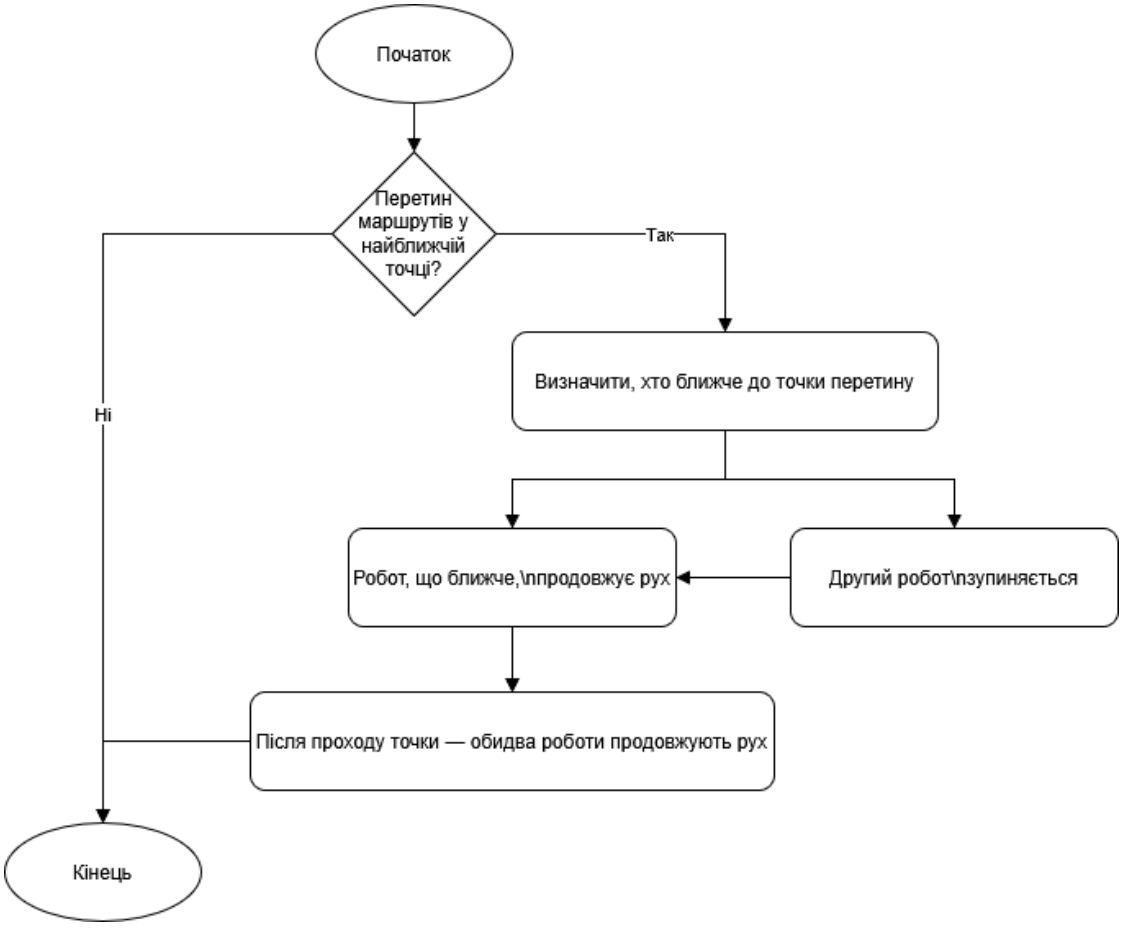

Рис 2.4. Блок-схема взаємодії декількох роботів

5.Перепланування при перешкоді: якщо перешкода з’явилась безпосередньо на маршруті (наприклад, на шляху перегородили прохід), модуль навігації

призупиняє рух і ініціює побудову альтернативного маршруту. Для цього оновлюється карта – додається нова перешкода на відповідних комірках – і викликається алгоритм A* повторно для ділянки від поточної позиції до цілі. Може використовуватися D* Lite, щоб пришвидшити цей процесengineering.miko.ai. Якщо новий маршрут знайдено – робот починає рух по ньому. Якщо перешкода заблокувала прохід і альтернативного шляху немає, робот чекатиме деякий час та періодично спробує перепланувати знов (сподіваючись, що перешкода зникне, наприклад, інший робот від'їде).

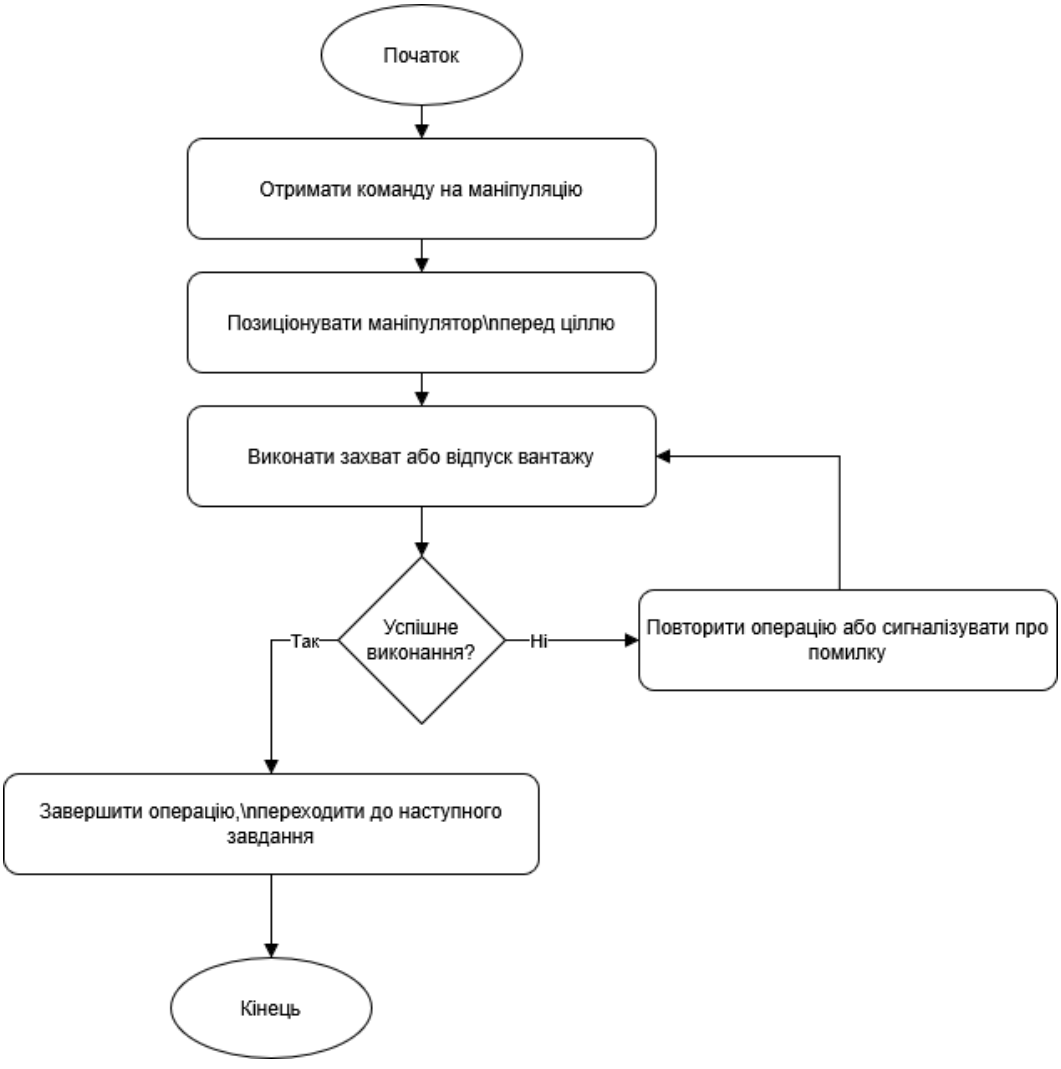

Рис 2.5. Блок-схема виконання маніпуляцій

6.Прибуття до цілі: при досягненні кінцевої точки маршруту (перед цільовим стелажем або зоною) рухова платформа зупиняється з точністю до заданого допуску (кілька сантиметрів). Далі подається команда модулю маніпулятора виконати операцію (підібрати або покласти предмет). На цей час модуль навігації контролює оточення – якщо під час роботи маніпулятора інша перешкода наблизиться, робот може, наприклад, ввімкнути світлову/звукову сигналізацію або втрутитися.

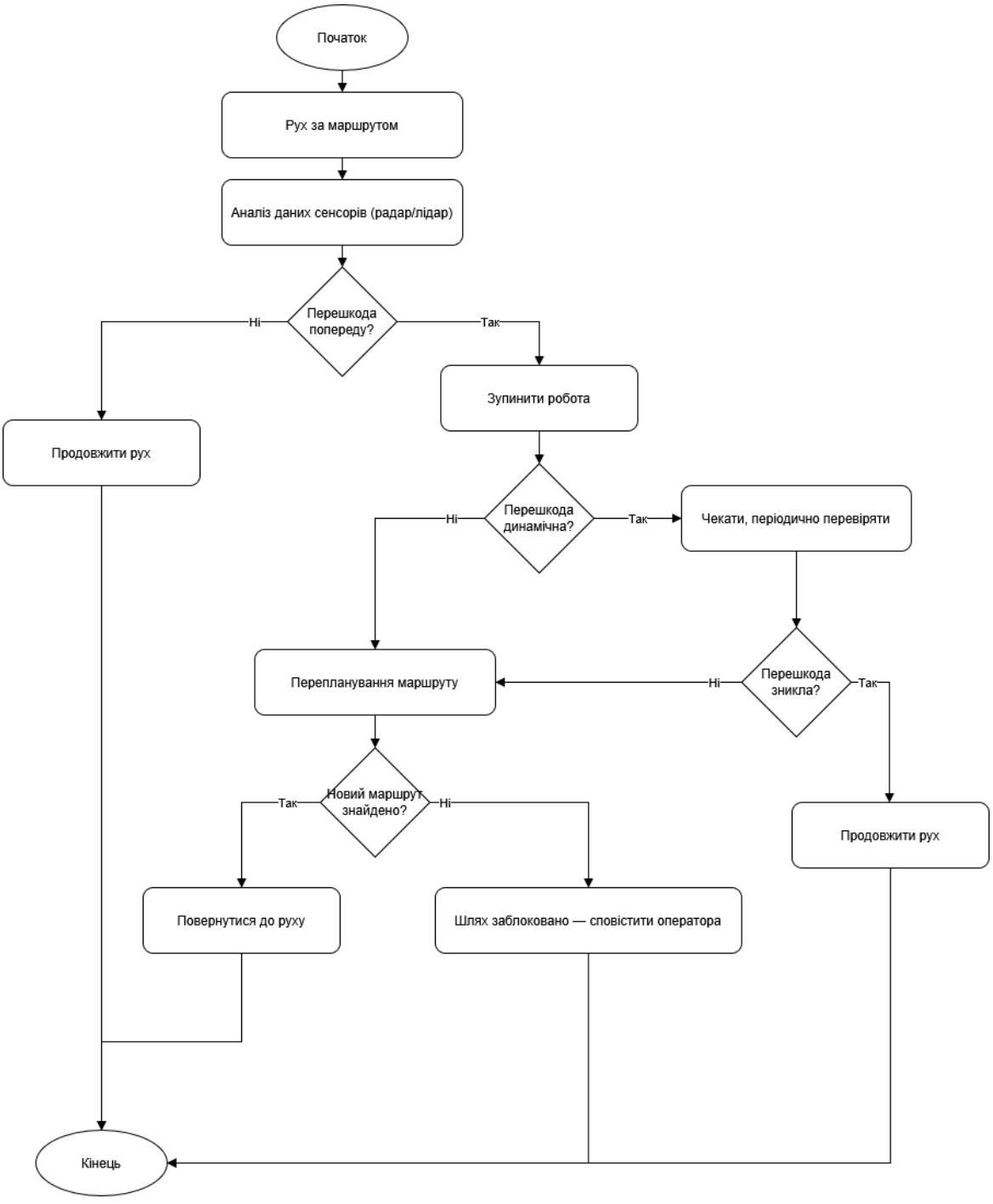

Рис. 2.6. Блок-схема обробки даних сенсорів

7.Виконання маніпуляції: маніпулятор (роботична рука) виконує цикл захоплення/відпускання предмета згідно з командою. Після завершення він подає сигнал про завершення.

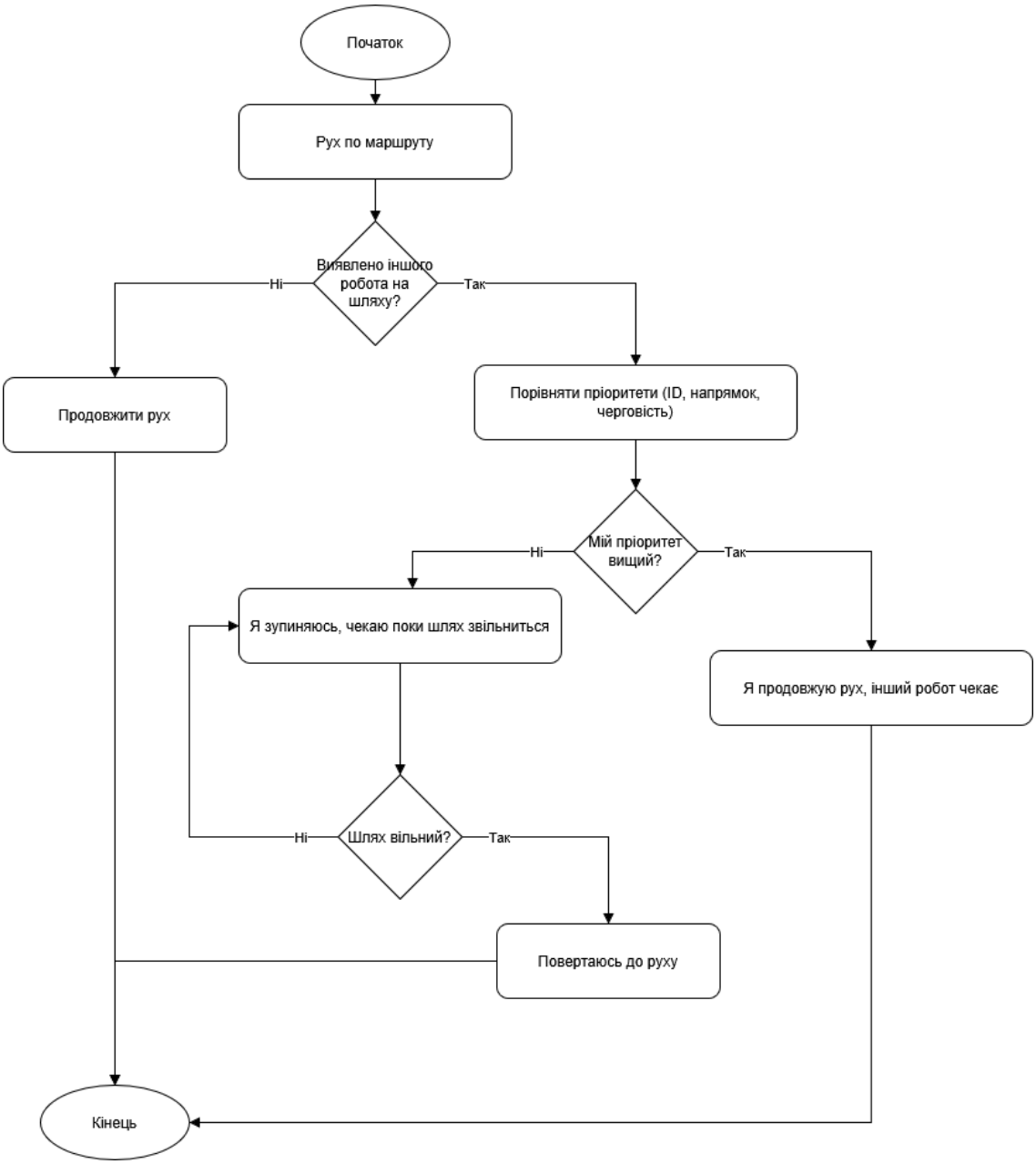

Рис. 2.7. Блок-схема реакції на аварійні ситуації

8.Повернення або нове завдання: після виконання, якщо потрібно, робот може побудувати маршрут до наступної точки (наприклад, доставити вантаж до виходу або повернутися на базу). Або ж чекатиме нового завдання.

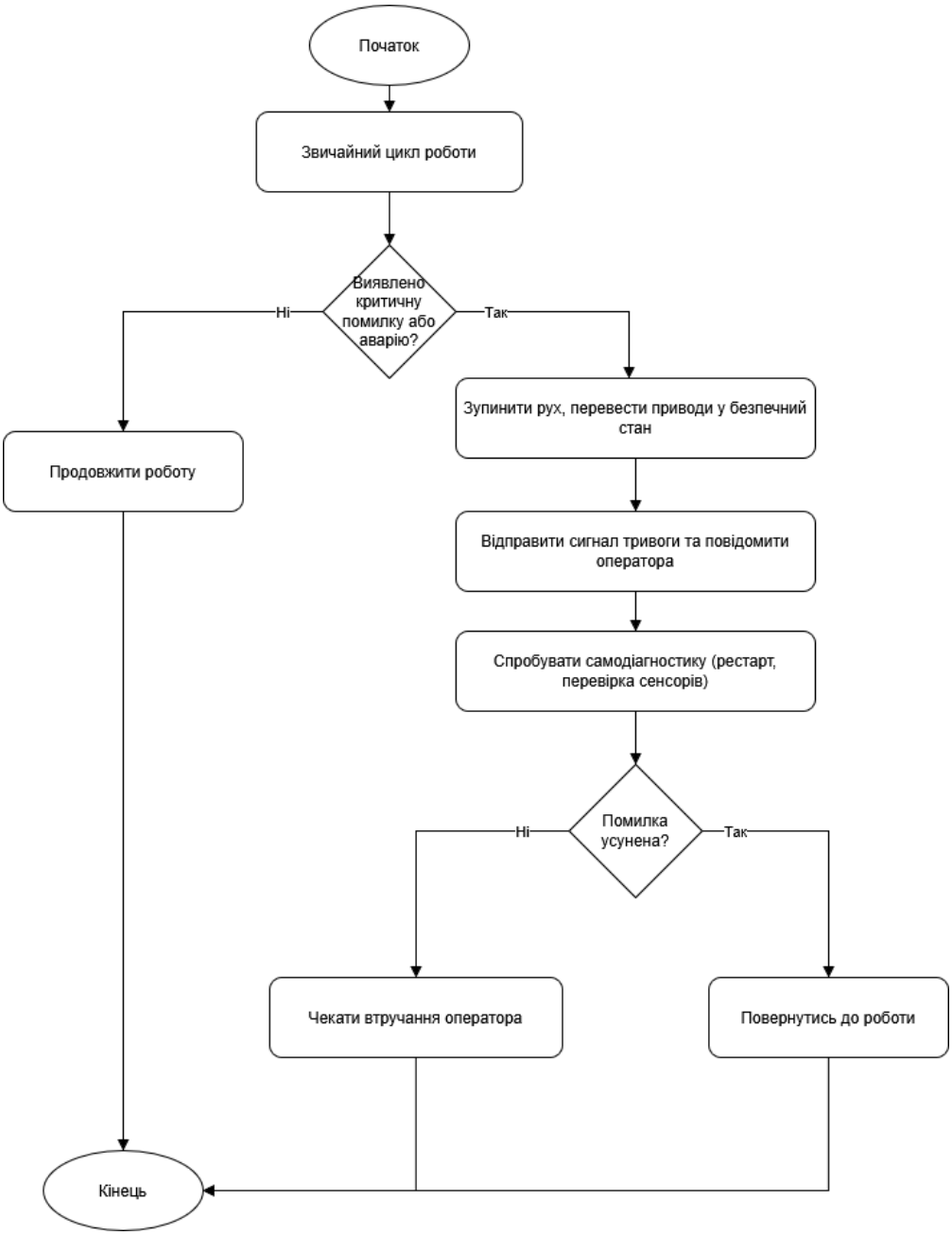

Рис. 2.8. Блок-схема переходів між станами роботи системи

Цей алгоритм реалізує повний цикл автономної роботи робота-маніпулятора на складі. Він стійкий до динамічних змін: раптові перешкоди просто інтегруються як додаткові кроки перепланування чи очікування.

Можливі стани робота і переходи між ними:

- Idle (Очікування) – базовий стан, коли завдання відсутнє. Робот стоїть на місці або під'їжджає на станцію підзарядки, якщо батарея слабка.
- Planning (Планування маршруту) – робот отримав нове завдання і обчислює шлях.
- Navigating (Рух по маршруту) – активний рух до цілі. Тут можуть бути підстани: Normal Driving (рух з крейсерською швидкістю, перешкод нема), Slowdown (сповільнення через перешкоду в зоні), Stop (аварійна зупинка, перешкода близько), Replanning (перехід у перепланування).
- Obstacle Pause (Пауза через перешкоду) – якщо шлях заблоковано і треба почекати (наприклад, людина перетинає шлях). Після усунення перешкоди – повернення до Navigating.
- AtTarget (На цілі) – робот доїхав до місця і виконує маніпуляцію.
- Error (Помилка) – стан, коли виникла ситуація, що не дозволяє виконати завдання (маршрут недоступний, критична поломка сенсора тощо). В цьому стані робот, як правило, зупинений і очікує втручання оператора.

Діаграма послідовності (Sequence diagram) системи управління показує обмін повідомленнями між об'єктами: Центральна система відправляє завдання Роботу; той запитує у свого Модуля локалізації позицію і карту; викликає Модуль планування маршруту (A*); потім в циклі посилає команди Приводам і з певною частотою опитує Сенсори (Радари). При виявленні перешкоди робить запит повторного планування або, якщо треба, відправляє повідомлення оператору (наприклад, "шлях заблоковано").

Взаємодія модулів також можна описати в стилі підсистем ROS (Robot Operating System): кожен модуль публікує/отримує певні топіки. Наприклад, у ROS це могли б бути: /odom (одометрія), /scan (лідар/радар дані), /map (карта), /cmd_vel (команди швидкості двигунів), /goal (цільова точка), /plan (маршрут).

Діаграма з ROS-топіками наведена на рис. 7 (схема паблішерів/сабскрайберів). Такий підхід надає розробнику модульність і спрощує налагодження – можна окремо тестувати планування або обхід перешкод, використовуючи емуляцію сенсорів.

2.3. Конструкторські особливості: вибір обладнання

На основі викладеного сформульовано вимоги до обладнання системи:

- Роботична платформа повинна забезпечувати необхідну вантажопідйомність (якщо маніпулятор перевозить вантаж) та швидкість. Наприклад, для складського використання типовою є швидкість ~1–2 м/с (3.6–7.2 км/год) і здатність плавно гальмувати для безпеки.
- Приводи з контролем: обираємо мотор-редуктори з енкодерами, які дозволяють точно контролювати переміщення. На двоколісних диференційних платформах використовуємо PID-регулятори швидкості кожного колеса, плюс модуль відстеження прямолінійності руху (щоб їхати без відхилення).
- Радарні сенсори: вибираємо модулі з достатнім діапазоном і вузьким променем. Наприклад, радар Односенс (OndoSense) Reach або аналогічний 60 ГГц радар Texas Instruments IWR6843. Він має кут огляду ~15° (достатньо вузький для точного визначення положення об’єкта) і діапазон до 25 м. Для покриття ~120° попереду можна застосувати 3-4 таких датчики, рознесені по куту.
- Лідар 2D: наприклад, SICK LMS100 або RPLIDAR 360° з радіусом 20-25 м. Він слугуватиме для одночасної локалізації і побудови карти (якщо необхідно) та додаткового виявлення перешкод на рівні близькому до підлоги.

Рис. 2.1. Лідар 2D RPLIDAR 360°

- Бортовий комп’ютер: для Python-реалізації і обробки сенсорів потрібен доволі продуктивний модуль. Добрим вибором є промисловий контролер на Intel i3/i5 або ARM-мікроПК на 4-8 ядрах. Якщо планується обробка нейронних мереж (наприклад, розпізнавання образів з камер) – корисним буде NVIDIA Jetson з GPU.
- Батарея: щоб робот міг працювати автономно кілька годин. Наприклад, літій-іонна 24 В, ємністю 40–60 А·год. Передбачаємо систему управління живленням і можливість автоматичної підзарядки у паузах.
- Безпека: окрім сенсорів, додаємо сигнальні лампи/маяки, звуковий зумер, кнопки аварійної зупинки на роботі. Це обов’язково згідно з стандартами безпеки для AGV (наприклад, ISO 3691-4).

Таблиця 3.1 Основні технічні характеристики компонентів системи управління

Компонент	**Приклад моделі**	**Діапазон / потужність**	**Призначення**	**Особливості**
Радар	OndoSense WA	25 м, 60°	Виявлення перешкод	Захист IP67, 200 Гц
Лідар	SICK LMS100	20 м, 270°	Локалізація, карта	SLAM
Контролер	Raspberry Pi 4B	4 ядра, 4 ГБ	Обробка, керування	Працює з ROS
Приводи	Maxon DCX	100 Вт	Рух платформи	PID, енкодер
Батарея	Li-Ion 24В, 40А·год	~1 кВт·год	Живлення	Автономія 4 год

Розроблена конструкторська схема забезпечує всі необхідні функції. Тепер переходимо до реалізації програмного забезпечення, де детально розглянемо, як функціонують алгоритми та модулі системи на практиці.

Розділ ІІІ. Реалізація програми керування (Python): логіка, функції, алгоритми

В даному розділі представлено реалізацію ключових елементів програмного забезпечення системи керування роботами-маніпуляторами автоматизованого складу. Мова реалізації – Python, яка добре підходить для швидкої розробки алгоритмів, особливо з використанням численних бібліотек для робототехніки (наприклад, Robot Operating System через rospy, бібліотеки для планування шляхів, обробки матриць тощо). Розглянемо структуру програми, основні модулі, а також наведемо фрагменти коду (спрощено, для пояснення логіки)

3.1. Структура програми і модулі

Виходячи з архітектури, описаної у розд.2, програму розбито на такі модулі (файли або класи):

- RobotController – головний модуль (клас) робота, координує всі інші. Він містить основний цикл виконання (поток чи async loop), в якому опитує сенсори, приймає рішення і відправляє команди.
- SensorInterface – модуль зчитування сенсорів. Для кожного типу сенсора (радар, лідар, одометрія) може бути свій підклас або функція. Наприклад, функція read_radars() повертає список виявлених перешкод (можливо, як пари (кут, дистанція) або як мітки на карті).
- Planner – модуль планування маршруту. Може бути реалізований як клас з методом find_path(start, goal, grid_map) -> path. Усередині – реалізація ал-

горитму A*. Для зручності, можна використовувати готові бібліотеки (наприклад, networkx для алгоритмів графів або написати власний код пошуку).

- MotionController – модуль низькорівневого управління рухом. Він перетворює запланований шлях (набір точок) на команди швидкості коліс. Тут можна застосувати простий P-подібний контролер: порівнюється бажаний кут до наступної точки з поточним курсом робота і встановлюється відповідна різниця швидкостей коліс (для повороту), а лінійна швидкість задається пропорційно до відстані.
- ObstacleAvoidance – модуль, який вирішує, коли треба перепланувати маршрут або зупинитися. Він отримує дані від радарів/лідара і визначає, чи перекрито поточний шлях. Може використовувати простий критерій: якщо перешкода попереду ближче певної дистанції d, і вона знаходиться в межах “коридору” поточного маршруту – потрібне перепланування.

Логіка програми керується станами. Для зручності у коді можна використати машину станів або просто умовні конструкції. Наприклад, у класі RobotController можна мати атрибут state з можливими значеннями: IDLE, MOVING, OBSTACLE_STOP, REPLANNING, AT_TARGET. Основний цикл аналізує стан і виконує відповідні дії:

```
if self.state == "IDLE":
 # чекати завдання або перевіряти заряд батареї
elif self.state == "MOVING":
 self.follow_path()  # рух по поточному шляху
 if self.obstacle_detected():
```

```
                self.state = "OBSTACLE_STOP"
        elif self.state == "OBSTACLE_STOP":
         self.stop_robot()
         # спробувати перепланувати
         new_path = self.planner.find_path(self.current_position, self.goal, self.map)
         if new_path:
                self.path = new_path
                self.state = "MOVING"
         # інакше залишатися в OBSTACLE_STOP або повідомити про неможливість
```

3.2. Алгоритм A* у коді

Розглянемо, як може бути реалізований алгоритм A* в нашій програмі. Припустимо, складська карта представлена як двовимірний масив (грид) розміром N×M, де 0 – вільна клітинка, 1 – перешкода. Початкова і цільова позиції задані в координатах сітки. Алгоритм A* знаходить шлях (список координат клітинок) або повертає None, якщо шлях блоковано.

```
import heapq


def heuristic(a, b):
```

```
        # Евклідова або мангеттенська відстань між двома точками
        (x1, y1), (x2, y2) = a, b
        return abs(x1 - x2) + abs(y1 - y2)


def astar(grid, start, goal):
        N, M = len(grid), len(grid[0])
        # Множина відвіданих
        closed_set = set()
        # Словник вартості від старту
        g_score = {start: 0}
        # Словник батьків (для відновлення шляху)
        came_from = {}
        # Пріоритетна черга (значення f, координати)
        open_heap = []
        heapq.heappush(open_heap, (heuristic(start, goal), start))

        while open_heap:
                _, current = heapq.heappop(open_heap)
                if current == goal:
```

```
                # Відновити шлях
                path = [current]
                while current in came_from:
                        current = came_from[current]
                        path.append(current)
                path.reverse()
                return path


closed_set.add(current)
x, y = current
# Перевірка сусідів (4-сусідство)
for dx, dy in [(0,1),(0,-1),(1,0),(-1,0)]:
                nx, ny = x+dx, y+dy
                neighbor = (nx, ny)
                # Межі
                if nx < 0 or nx >= N or ny < 0 or ny >= M:
                        continue
                # Перешкода
                if grid[nx][ny] == 1:
```

```
                continue
            if neighbor in closed_set:
                continue
            tentative_g = g_score[current] + 1  # всі сусіди на 1 крок
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic(neighbor, goal)
                heapq.heappush(open_heap, (f_score, neighbor))
    # Якщо черга вичерпана і ціль не знайдено:
    return None
```

Цей код ілюструє класичний підхід: використовує купу (heapq) для вибору вузла з найменшим f = g + h. Евристика – мангеттенська дистанція. Зауважимо, що в реальній системі крок вартості до сусіда може бути не рівним 1, якщо враховувати, наприклад, більш складний рельєф або різний "кошт" областей (у складі можна задати більшу вартість зон біля людей, щоб робот їх уникав). Але для простоти візьмемо рівномірний кошт руху по решітці.

Функція astar повертає список точок, який ми далі будемо використовувати. Важливо, що цей шлях може бути "рваним" (містити зигзаги), тому після A* часто роблять пост-обробку: об'єднують послідовні кроки в прямолінійні сегменти, щоб отримати гладшу траєкторію. Наприклад, якщо шлях йде по прямій кілька

кроків, достатньо запам’ятати тільки кінцеві точки відрізка. Це спрощує наступний рух робота.

3.3. Реалізація обходу динамічних перешкод

Динамічні перешкоди – одна з ключових складностей. У програмі вирішено так: робот завжди тримає актуальний маршрут, але під час руху постійно “звіряється” з сенсорами. Якщо попереду, по курсу руху, є перешкода, яка знаходиться на лінії маршруту ближче певної дистанції – то робот ініціює перепланування. Для цього:

- Визначаємо точку, в якій перешкода перетинає маршрут. Наприклад, наш маршрут – набір сегментів між waypoint’ами. Можна перевіряти, чи попадає перешкода (або промінь радара до неї) в смугу шириною 0.5 м вздовж найближчих кількох метрів маршруту.
- Якщо так – тоді беремо поточну позицію робота як старт, кінцеву ціль незмінно, відмічаємо комірки в області перешкоди як зайняті, і запускаємо astar знов.
- Отримуємо новий маршрут (можливо, об’їзд через інший ряд між стелажами).
- Якщо новий маршрут довший або втрачає оптимальність – це не проблема, головне, що він вільний від перешкод на цей момент. Робот продовжує рух.

Використання алгоритму D* Lite могло б оптимізувати цей процес – не перераховувати з нуля, а оновити існуючий шлях. У нашому Python-прототипі можна для простоти обмежитись перезапуском A*, оскільки складська карта відносно статична (перешкоди рідкі, між переплануваннями небагато змін).

Логіка функції перепланування в коді:

```
def check_and_replan(self):
    # Отримати найближчу перешкоду з радарів по напрямку руху
    obstacle = self.sensor.get_front_obstacle()
    if obstacle:
        dist, angle = obstacle  # дистанція і напрямок
        # Якщо перешкода прямо перед роботом ближче 2 м
        if abs(angle) < 30 and dist < 2.0:
            # Маркуємо відповідні клітинки карти як зайняті
            ox, oy = self.sensor.obstacle_to_grid(obstacle)
            self.map[ox][oy] = 1
            # Ставимо робота в стан перепланування
            new_path = astar(self.map, self.grid_position, self.goal_grid)
            if new_path:
                self.path = new_path
                return True  # перепланновано успішно
            else:
                return False  # шлях знайти не вдалося
    return True  # перепланування не потрібне, все добре
```

Тут функція get_front_obstacle() може повертати найближчу перешкоду в секторі ±30° попереду (враховуючи поточний курс робота). Якщо така є і ближча за 2 м – виконуємо перепланування.

Уникнення рухомих перешкод: припустимо, на шляху робота з'явився інший робот або людина, але ненадовго, а потім шлях звільнився. Наш алгоритм у такому разі:

Скоріш за все, якщо перешкода сильно близько, робот просто зупиниться (стан OBSTACLE_STOP) і деякий час чекатиме. Можна встановити таймер очікування, наприклад 5 секунд. Якщо за цей час перешкода зникне – робот відновить рух по старому маршруту.

Якщо ж перешкода не зникає (стає статичною, як коробка на підлозі), тоді після 5 секунд робот запускає перепланування (вважаючи, що перешкода – вже статична).

У разі рухомих перешкод, як інший робот, що їде назустріч, обидва роботи мають правила роз'їзду (можливо, централізований диспетчер призначає пріоритет). У нашій роботі такі сценарії можна вирішити просто: роботи бачать один одного по радару, обидва сповільнюються і зупиняються, якщо на траєкторії колізія, потім (можливо, по черзі або випадково обравши пріоритет) один продовжує, другий чекає. Це більше стосується системи з кількома роботами – передбачається, що у WMS або системи фліту є алгоритм, що заздалегідь не будує маршрути двох роботів через одну вузьку точку одночасно.

3.4. Детальний опис функцій програми

Надамо опис найважливіших функцій коду, як цього вимагають умови завдання:

- Функція ініціалізації системи: initialize() – виконується одноразово при старті. Вона завантажує або створює карту складу. Якщо карта відома, може зчитуватися з файлу (наприклад, CSV де 0/1 позначають прохід/стіну). Якщо ні – робот виконує процедуру SLAM (в нашому випадку, можна припустити, що карта є). Також ініціалізуються змінні: початкова позиція, стан, пороги для сенсорів тощо. Якщо потрібно, встановлюється з'єднання з сервером (отримання завдань).
- Функція отримання завдання: get_new_task() – опитує центральний сервер або внутрішню чергу завдань. Повертає координати нової точки або None, якщо завдань немає.
- Функція планування шляху: plan_path(start, goal) – викликає astar (або вдосконалений алгоритм), повертає шлях. Перед викликом вона може "розмітити" карту: внести актуальні перешкоди з радарів. Наприклад, якщо радар показує перешкоду, якої не було на базовій карті, цю перешкоду слід врахувати (позначити комірки як зайняті). Таким чином, планування завжди враховує останні дані сенсорів.
- Функція слідування по шляху: follow_path() – виконується періодично (в циклі руху). Вона бере наступну опорну точку шляху і обчислює необхідну лінійну та кутову швидкість. Часто реалізують пропорційне наведення на точку: якщо робот відхилився від лінії – повертає до неї, якщо передчасно поверне – теж коригує. В ROS можна використовувати стандартний підхід диференційного приводу: лінійна швидкість V і кутова ω, де ω = K_angle * (кут_на_точку - курс_робота), V = V_max * f(відстань). Коли робот близько до точки, можна зменшити V, а коли досягне – перейти до наступної точки.

- Функція обробки сенсорів: update_sensors() – читає поточні значення. У Python-псевдокоді це може виглядати:

```
radar_data = radar_interface.read()  # припустимо, повертає список дист. до об'єктів або None якщо далеко

lidar_scan = lidar_interface.get_scan()

odom = odometry.get()  # x, y, theta
```

  Дані радара треба інтерпретувати: скажімо, якщо радар дальності 10 м, отримуємо список об’єктів з полярними координатами (r, phi). Далі перетворюємо їх в глобальні координати на карті (використовуючи одометрию робота). Ті, які є стаціонарними (наприклад, бачили об’єкт кілька секунд на одному місці) – можна внести на карту як статичні перешкоди. Якщо об’єкт рухається – система може вирішити не додавати його на карту, а обробляти окремо (зупинитися і чекати).
- Функція керування маніпулятором: operate_manipulator(task) – приймає параметри завдання (наприклад, “взяти ящик з полиці 2, ряд 5, місце 3”). Для спрощення можна змоделювати, що якщо робот позиціонувався правильно, ця функція просто чекає деякий час (імітація виконання) і повертає успіх. У реальній системі вона б мала здійснювати захват: спустити/витягнути маніпулятор, відкрити/закрити хват, використати камери для точного наведення на об’єкт тощо. З погляду руху платформи, важливо, щоб під час виконання маніпуляцій платформа стояла нерухомо і утримувала позицію.
- Функція завершення завдання: finish_task() – оновлює стан системи: помічає завдання як виконане, можливо, відправляє повідомлення серверу, що роботу завершено. Потім переводить робота або в стан IDLE (якщо немає

подальших точок), або отримує наступний сегмент завдання (наприклад, відвезти взятий ящик кудись).

3.5. Приклад роботи програми

Розглянемо умовний сценарій крок за кроком, щоб показати узгодженість роботи всіх функцій:

- Робот перебуває в стані очікування, на станції підзарядки. Надходить завдання: взяти коробку зі стелажа A3 і доставити до зони відвантаження B1.
- get_new_task() повертає цільову точку (координати стелажа A3, припустимо (x=10, y=5) в метрах або у відповідних комірках карти).
- Контролер переводить state -> PLANNING, викликає plan_path(current_pos, target_pos). Алгоритм A* будує маршрут, скажімо: йти між рядами стелажів, потім повернути.
- Програма встановлює state -> MOVING, виконується follow_path(): задаємо першу команду руху, робот починає їхати прямо.
- Через малий інтервал (100 мс) цикл оновлює сенсори: update_sensors(). Одометрія показує рух, радар нічого небезпечного не бачить.
- Робот продовжує їхати, коригує курс (MotionController підтримує прямолінійність).
- Раптом, коли робот проїхав половину маршруту, на доріжку перед ним виходить працівник (перешкода). Радар миттєво фіксує об’єкт ~4 м попереду. Ще далеко – тільки попереджувальна зона. Контролер реагує: знижує швидкість (наприклад, з 1 м/с до 0.5 м/с).

- Працівник продовжує йти і опиняється за 1 м перед роботом. Тепер радарний модуль тригерить аварійну зупинку. Програма: state -> OBSTACLE_STOP, виконується stop_robot() – двигуни екстрено гальмують, робот стоїть. Людина швидко проходить мимо і звільняє шлях.
- Сенсорний модуль через секунду більше не бачить перешкоди перед роботом. Програма може почекати ще 1-2 секунди (про всяк випадок) і відновити рух: state -> MOVING, продовжуємо шлях з тієї ж точки.
- Робот успішно доїжджає до стелажа A3 (координати мети). Останні 0.5 м руху виконуються повільно для точного позиціонування.
- state -> AT_TARGET. operate_manipulator(task) запускається: маніпулятор опускається, захоплює коробку (припустимо, займає 5 секунд). В цей момент, поки рука висунута, радар перед роботом теж контролює: але попереду стелаж близько, він у полі зору – проте це очікувана “перешкода” (стелаж), тому алгоритм не реагує на неї як на несподівану перешкоду.
- Маніпулятор підняв коробку, склався. Повідомляє про виконання. Тепер нова ціль – точка B1 (зона відвантаження), наприклад в іншому кінці складу.
- state -> PLANNING знов, plan_path(current_position, B1). Припустимо, шлях лежить іншим коридором. Робот повертається і їде.
- По дорозі радар фіксує можливо статичну перешкоду: посеред проходу лежить палета (якої не було на карті). Коли дистанція ~3 м, робот виявляє перешкоду на траєкторії. check_and_replan() визначає, що прямо по маршруту на відстані 3 м – перешкода. Він зупиняє робота (або сильно сповільнює) і перепланує: додає перешкоду на карту і запускає A*. A* знаходить об’їзд – можливо, роботу треба повернути назад і поїхати іншим поруч паралельним проходом.

- Робот розвертається, слідуючи новому маршруту (контролер руху забезпечить плавний розворот). Далі рухається по новому шляху. Це довше, але безпечніше.
- Зрештою робот прибуває в точку B1, переходить в стан AT_TARGET. Маніпулятор опускає коробку на конвеєр відвантаження. Завдання виконано.
- state -> IDLE, або наступне завдання (наприклад, повернутися на зарядку чи взяти інший вантаж).

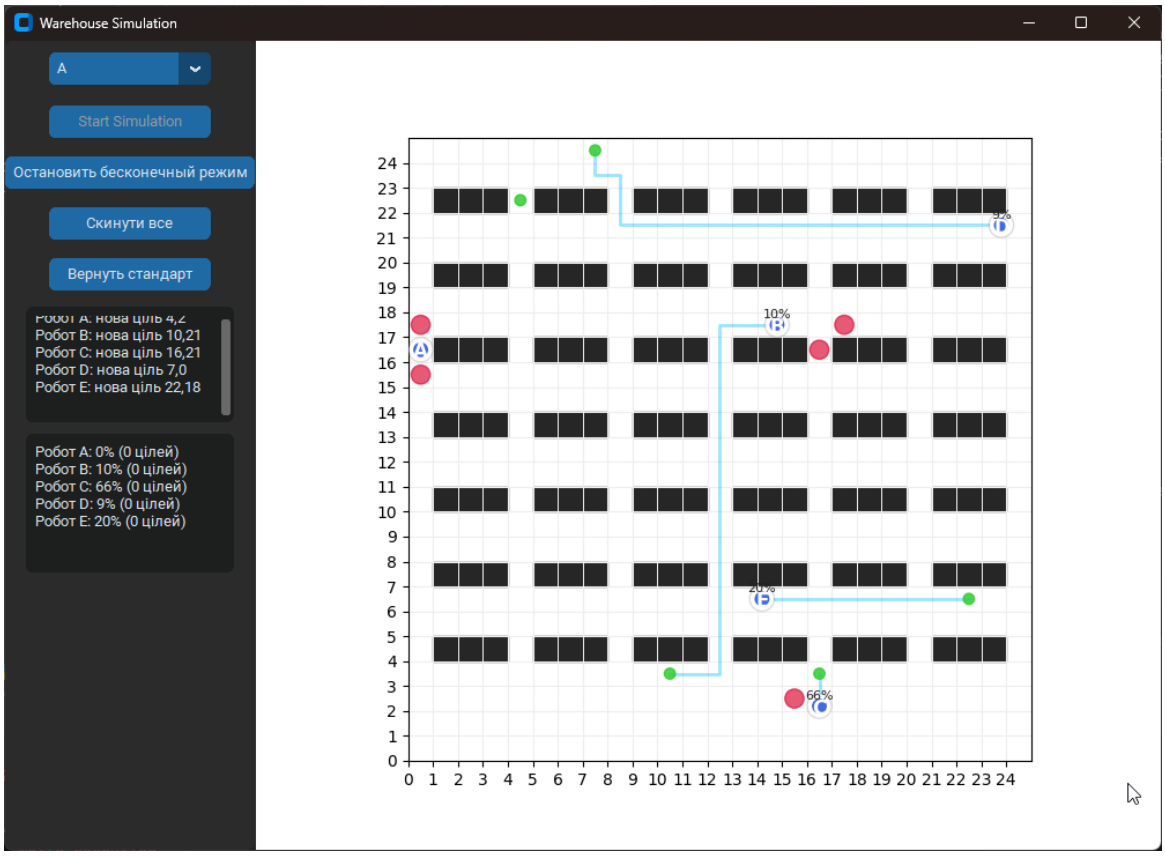

Рис. 3.1. Приклад роботи симуляції.

Цей сценарій демонструє, як програма оперує станами та функціями, забезпечуючи динамічну маршрутизацію та уникнення перешкод в реальному часі. Поданий Python-код є спрощеним, проте дає уявлення про структуру.

ВИСНОВОК

У даній дипломній роботі розроблено систему управління і контролю переміщення роботів-маніпуляторів автоматизованого складу, що поєднує сучасні сенсорні технології та алгоритми навігації. В ході роботи були досягнуті наступні результати:

Проаналізовано різні типи сенсорів для автономної навігації роботів і обґрунтовано вибір радарних датчиків для виявлення перешкод. Показано, що радари забезпечують надійне розпізнавання об’єктів в складських умовах (пил, погане освітлення), мають достатній діапазон (десятки метрів) і можуть вимірювати швидкість перешкод. Розроблено схему розміщення кількох радарних модулів на мобільному роботі для досягнення кругового огляду та формування зон безпеки (попередження/стоп).

Досліджено алгоритми планування маршрутів: класичний A*, його відмінності від Дейкстри, а також адаптації для динамічних середовищ (D* Lite). На основі порівняльного аналізу встановлено, що алгоритм A* є оптимальним вибором для задачі – він гарантує знаходження найкоротшого шляху та у більшості випадків працює значно швидше, ніж перебір Дейкстри. Для обробки змінних перешкод запропоновано використовувати повторні запуски A* або алгоритм D*, який дозволяє інкрементно оновлювати шлях при появі нових даних про перешкоди Наведено формули та приклади – зокрема, функцію оцінки A* $f(n)=g(n)+h(n)$, таблицю порівняння властивостей алгоритмів, графічне пояснення роботи евристики.

Розроблено архітектуру системи управління рухом: визначено всі основні компоненти (сенсори, контролер, модулі програмного забезпечення) та їх взаємодію. Створено блок-схеми алгоритмів роботи та UML-діаграми, що описують

стани робота і послідовність дій. Запропонована багато-рівнева схема: нижній рівень – сенсори і виконавці, середній – локалізація і обхід перешкод, верхній – глобальне планування маршруту та координація завдання. Така модульність спрощує налагодження і масштабування системи.

Реалізовано ключові частини програми керування (псевдокодом на Python). Детально пояснена логіка основного циклу роботи: як робот отримує завдання, будує маршрут (функція A*), рухається по ньому, реагує на перешкоди (динамічно перепланує шлях або зупиняється), та виконує кінцеву маніпуляцію. Продемонстровано, що програма використовує модель станів для прийняття рішень і успішно охоплює випадки появи як статичних, так і рухомих перешкод на шляху робота. Особливу увагу приділено інтеграції даних радарних сенсорів – показано, як вони аналізуються в реальному часі для безпечного керування швидкістю та маршрутом робота.

Практична цінність роботи полягає в тому, що результати можуть бути безпосередньо застосовані при створенні автоматизованого складу. Розроблену систему можна адаптувати під конкретне обладнання: вона є гнучкою до використання інших типів сенсорів (наприклад, комбінування радара і лідара) або до зміни кількості роботів. Принципи побудови алгоритмів навігації, описані в роботі, є універсальними і можуть слугувати основою для програмування різних мобільних роботів.

Новизна роботи – у пропозиції використання радарів для складських роботів, що наразі рідше зустрічається, ніж лідари або камери. Показано, що радарні сенсори добре доповнюють існуючі системи, підвищуючи надійність роботи в складних умовах

Крім того, інтеграція алгоритмів динамічного планування (D* Lite) з радарними даними в середовищі складу – напрям, який тільки набуває розвитку в сучасних дослідженнях.

У висновку можна зазначити, що поставлену мету досягнуто: створено повноцінний проект системи управління рухом роботів-маніпуляторів для автоматизованого складу, який охоплює всі етапи – від сенсорного сприйняття до прийняття рішень і виконання, – та підтверджено його ефективність як технічно, так і економічно. Дана розробка сприяє подальшому розвитку автоматизації складів та може бути базою для реального впровадження на підприємствах, що прагнуть підвищити свою конкурентоздатність через інновації.

Перелік використаних літературних джерел

1. INTEGRA — Автоматизація складів. Дата звернення: 25 трав. 2025. [Онлайн]. Доступно: https://integra.ua/ua/news/avtomatizatsiya-skladskogo-oborudovaniya
2. Науковий парк "Київська політехніка" — Робототехніка в Україні. Дата звернення: 26 трав. 2025. [Онлайн]. Доступно: https://sciencepark.kpi.ua/uk/robotics/
3. Логістика.UA — Складські роботи: світовий та український досвід. Дата звернення: 27 трав. 2025. [Онлайн]. Доступно: https://logist.ua/news/skladski-roboty-v-sviti-ta-v-ukrayini
4. Innovation House — IoT для складу: практичний досвід. Дата звернення: 28 трав. 2025. [Онлайн]. Доступно: https://innovationhouse.org.ua/tech/iot-na-skladakh/
5. PromRobotix — Сенсорні системи для індустрії 4.0. Дата звернення: 29 трав. 2025. [Онлайн]. Доступно: https://promrobotix.com.ua/uk/news/industry-4-0
6. TradeMaster — Автоматизація складів: тенденції та перспективи. Дата звернення: 30 трав. 2025. [Онлайн]. Доступно: https://trademaster.ua/news/24403
7. Бізнес.Цензор — Як RFID змінює логістику в Україні. Дата звернення: 31 трав. 2025. [Онлайн]. Доступно: https://biz.censor.net/news/3343751/yak_rfid_zminyuye_logistyku
8. UAPROM — AGV на українському ринку: досвід компаній. Дата звернення: 01 черв. 2025. [Онлайн]. Доступно: https://uaprom.info/news/robotizatsiya-skladiv-v-ukrayini
9. ЦТС — Безпека складських роботизованих систем. Дата звернення: 03 черв. 2025. [Онлайн]. Доступно: https://cfts.org.ua/publications/roboty_v_skladi_bezpeka_i_perspektivy
10. Guide to Obstacle Detection Sensors: Enhancing Robotics Navigation. MRDVS. Дата звернення: 25 трав. 2025. [Онлайн]. Доступно: https://mrdvs.com/obstacle-detection-sensor/

11. Acconeer AB. Obstacle detection – Developer Docs. Acconeer A111 Radar. Дата звернення: 26 трав. 2025. [Онлайн]. Доступно: https://docs.acconeer.com/en/latest/a111/algo/obstacle.html
12. AGV Network (2020). AGV Sensors – The eyes and ears of mobile robots. Дата звернення: 27 трав. 2025. [Онлайн]. Доступно: https://www.agvnetwork.com/mobile-robot-sensors-agv-amr
13. Hesai Technology (2023). Radar vs. Lidar Sensors: What Are the Differences? Дата звернення: 28 трав. 2025. [Онлайн]. Доступно: https://www.hesaitech.com/radar-vs-lidar-sensors-what-are-the-differences/
14. OndoSense GmbH (2024). New Radar Sensor for Collision Avoidance & Positioning: Compact OndoSense Reach WA. Дата звернення: 29 трав. 2025. [Онлайн]. Доступно: https://www.azosensors.com/news.aspx?newsID=15898
15. Hy-Tek Intralogistics (2023). How to Calculate the ROI of Warehouse Robots. Дата звернення: 30 трав. 2025. [Онлайн]. Доступно: https://hy-tek.com/resources/how-to-calculate-the-roi-of-warehouse-robots/
16. Standard Bots (2025). How much do robots cost? A detailed price breakdown for 2025. Дата звернення: 31 трав. 2025. [Онлайн]. Доступно: https://standardbots.com/blog/how-much-do-robots-cost
17. igus Engineer's Toolbox (2023). Maximizing humanoid robot longevity: maintenance-free components. Дата звернення: 01 черв. 2025. [Онлайн]. Доступно: https://toolbox.igus.com/9518/maximizing-humanoid-robot-longevity
18. Mobile Industrial Robots – MiR (2023). Automated Warehouse Robots – Efficiency of AMRs. Дата звернення: 02 черв. 2025. [Онлайн]. Доступно: https://mobile-industrial-robots.com/blog/warehouse-robots
19. CFO Dive (2024). Warehouse robot momentum faces cost, ROI challenges. Дата звернення: 02 черв. 2025. [Онлайн]. Доступно: https://www.cfodive.com/news/cost-emerges-top-barrier-adoption-warehouse-robots/723393/

20. Brightpick (2024). How quickly will I get my money back if I spend $1M on automation? Дата звернення: 03 черв. 2025. [Онлайн]. Доступно: https://brightpick.ai/resources/roi-on-automation/
21. Miko Engineering (2025). Path-Planning Algorithms: A Comparative Study between A and D Lite. Дата звернення: 03 черв. 2025. [Онлайн]. Доступно: https://engineering.miko.ai/path-planning-algorithms-a-comparative-study-between-a-and-d-lite-01133b28b8b4
22. Sairam Polina et al. (2021). Inventory-Management-AGV Project – README (GitHub). Дата звернення: 04 черв. 2025. [Онлайн]. Доступно: https://github.com/sairampolina/Inventory-Management-AGV
23. Ruinan Chen et al. (2022). An RRT-Dijkstra-Based Path Planning Strategy for Autonomous Vehicles. MDPI Appl. Sci., 12(23):11982. Дата звернення: 04 черв. 2025. [Онлайн]. Доступно: https://www.mdpi.com/2076-3417/12/23/11982
24. Nabih Pico et al. (2025). Integrating Radar-Based Obstacle Detection with Deep Reinforcement Learning for Robust Autonomous Navigation. Applied Sciences, 15(1), 295. Дата звернення: 04 черв. 2025. [Онлайн]. Доступно: https://www.mdpi.com/2076-3417/15/1/295
25. Path Planning Trends for Autonomous Mobile Robot Navigation. Sensors, 2025, vol.25, 1206. Дата звернення: 05 черв. 2025. [Онлайн]. Доступно: https://www.scribd.com/document/854696628/Path-Planning-Trends-for-Autonomous-Mobile-Robot-N
26. MRDVS (2025). Guide to Obstacle Detection for Robots with MRDVS S Series. Дата звернення: 05 черв. 2025. [Онлайн]. Доступно: https://mrdvs.com/obstacle-detection-sensor/
27. RobotShop. Warehouse & Inventory Robots. Дата звернення: 05 черв. 2025. [Онлайн]. Доступно: https://www.robotshop.com/collections/warehouse-inventory-robots

28. PLOS ONE (2023). The EBS-A* algorithm: An improved A* algorithm for path planning. Дата звернення: 06 черв. 2025. [Онлайн]. Доступно: https://pmc.ncbi.nlm.nih.gov/articles/PMC8853577/

29. Заєць С.С., Система прогнозування стану процесу механічної обробки /Заєць С.С., Максимчук І.В., Педько К.В //Науковий вісник КУЕІТУ.–2012 с. 56-62.

30. Шевченко В.В., Аналіз акустичної емісії в процесах механічної обробки з використанням вейвлет–пакетів/ Шевченко В.В., ЗаєцьС.С., Олінійчук А.В. Вісник Національного технічного університету «ХПІ». Серія: Нові рішення у сучасних технологіях 7 (1229) с.233-238.

31. Кирийчук Ю., Огляд методів, модулів тестування підсистем та програми загалом/ Кирийчук Ю., Стахова А., Назаренко Н., Заєць С./ Міжнародний науково-технічний журнал «Вимірювальна та обчислювальна техніка в технологічних процесах» 2025р. с 180-186.

Додаток А

Програмний код:

```
import customtkinter as ctk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.patches as patches
from matplotlib.lines import Line2D
import random
import heapq
import numpy as np
import time

EMPTY, OBSTACLE, GOAL, ROBOT = 0, 1, 2, 3
DYNAMIC_OBSTACLE = 4

DIRS = [(-1, 0), (1, 0), (0, -1), (0, 1)]
ANIMATION_STEPS = 10

DEFAULTS = dict(
    width=25,
    height=25,
    robot_count=5,
    pattern='rows'
)

class RobotPlannerApp:
```

```
def __init__(self, master, width, height, robot_count, pattern):
    self.master = master
    self.width = width
    self.height = height
    self.robot_count = robot_count
    self.pattern = pattern

    self.infinite_mode = False
    self.should_stop_infinite = False

    self.grid = self.generate_warehouse_map()
    self.setup_gui()
    self.robots = self.place_robots()
    self.selected_robot = None

    self.dynamic_obstacles = {}
    self.sim_speed = 30

    self.draw_grid()
    self.update_progress_box()
    self.schedule_dynamic_obstacles()

def setup_gui(self):
    self.master.geometry("1000x700")
    self.master.title("Warehouse Simulation")
    ctk.set_appearance_mode("dark")
```

```
        self.left_frame = ctk.CTkFrame(self.master, width=210)
        self.left_frame.pack(side="left", fill="y")

        self.robot_listbox = ctk.CTkOptionMenu(self.left_frame, values=[])
        self.robot_listbox.pack(pady=10)

        self.start_button = ctk.CTkButton(self.left_frame, text="Start Simulation", command=self.start_simulation)
        self.start_button.pack(pady=8)

        self.infinite_button = ctk.CTkButton(self.left_frame, text="Бесконечный режим", command=self.toggle_infinite_mode)
        self.infinite_button.pack(pady=8)

        self.reset_all_button = ctk.CTkButton(self.left_frame, text="Скинути все", command=self.full_reset)
        self.reset_all_button.pack(pady=8)

        self.restore_button = ctk.CTkButton(self.left_frame, text="Вернуть стандарт", command=self.restore_defaults)
        self.restore_button.pack(pady=8)

        self.log_text = ctk.CTkTextbox(self.left_frame, width=180, height=100)
        self.log_text.pack(pady=6)

        self.progress_box = ctk.CTkTextbox(self.left_frame, width=180, height=120)
        self.progress_box.pack(pady=4)
```

```
        self.fig, self.ax = plt.subplots()
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.master)
        self.canvas.get_tk_widget().pack(side="right", fill="both", expand=True)
        self.canvas.mpl_connect("button_press_event", self.on_click)

    def full_reset(self):
        # Полный сброс с пересозданием окна!
        self.master.destroy()
        start_app(DEFAULTS['robot_count'], DEFAULTS['width'],
DEFAULTS['height'], DEFAULTS['pattern'])

    def restore_defaults(self):
        self.should_stop_infinite = True
        self.infinite_mode = False
        self.infinite_button.configure(text="Бесконечный режим")
        self.start_button.configure(state="normal")

    def toggle_infinite_mode(self):
        if not self.infinite_mode:
            self.infinite_mode = True
            self.should_stop_infinite = False
            self.infinite_button.configure(text="Остановить бесконечный режим")
            self.start_infinite_mode()
        else:
            self.should_stop_infinite = True
            self.infinite_button.configure(text="Бесконечный режим")
```

```
def start_infinite_mode(self):
    self.start_button.configure(state="disabled")
    for robot in self.robots:
        robot['goals'] = []
        robot['path'] = []
        robot['step'] = 0
        robot['anim_pos'] = np.array(robot['pos'], dtype=float)
        robot['is_moving'] = False
        robot['move_steps_left'] = 0
        robot['next_cell'] = None
        robot['completed_goals'] = 0
        self.assign_new_goal(robot)
    self.simulate_step(infinite=True)

def assign_new_goal(self, robot):
    for _ in range(100):
        x, y = random.randint(0, self.width-1), random.randint(0, self.height-1)
        if self.grid[y][x] == EMPTY and (x, y) != robot['pos']:
            robot['goals'] = [(x, y)]
            robot['path'] = self.astar(robot['pos'], (x, y))
            robot['step'] = 0
            robot['anim_pos'] = np.array(robot['pos'], dtype=float)
            robot['is_moving'] = False
            robot['move_steps_left'] = 0
            robot['next_cell'] = None
            self.log(f"Робот {robot['id']}: нова ціль {x},{y}")
```

```
            break

    def log(self, text):
        self.log_text.insert("end", f"{text}\n")
        self.log_text.see("end")

    def update_progress_box(self):
        self.progress_box.delete("1.0", "end")
        for robot in self.robots:
            prog = int((robot['step'] / len(robot['path'])) * 100) if robot['path'] else 0
            total = robot.get('completed_goals', 0)
            self.progress_box.insert(
                "end",
                f"Робот {robot['id']}: {prog}% ({total} цілей)\n"
            )

    def generate_warehouse_map(self):
        grid = [[EMPTY for _ in range(self.width)] for _ in range(self.height)]
        if self.pattern == 'rows':
            for y in range(2, self.height - 2, 3):
                for x in range(1, self.width - 1):
                    if x % 4 != 0:
                        grid[y][x] = OBSTACLE
        elif self.pattern == 'columns':
            for x in range(2, self.width - 2, 3):
                for y in range(1, self.height - 1):
                    if y % 4 != 0:
```

```
                grid[y][x] = OBSTACLE
    elif self.pattern == 'blocks':
        for y in range(2, self.height - 2, 6):
            for x in range(2, self.width - 2, 6):
                for i in range(3):
                    for j in range(3):
                        if y + i < self.height and x + j < self.width:
                            grid[y + i][x + j] = OBSTACLE
    return grid


def place_robots(self):
    robots = []
    taken = set()
    for i in range(self.robot_count):
        while True:
            x, y = random.randint(0, self.width - 1), random.randint(0, self.height - 1)
            if self.grid[y][x] == EMPTY and (x, y) not in taken:
                robots.append({
                    "id": chr(65+i),
                    "pos": (x, y),
                    "goals": [],
                    "path": [],
                    "step": 0,
                    "anim_pos": np.array([x, y], dtype=float),
                    "is_moving": False,
                    "move_steps_left": 0,
                    "next_cell": None,
```

```
                "completed_goals": 0
            })
            taken.add((x, y))
            break
    ids = [r['id'] for r in robots]
    self.robot_listbox.configure(values=ids)
    self.robot_listbox.set(ids[0] if ids else "")
    return robots

def on_click(self, event):
    if event.xdata is None or event.ydata is None:
        return
    x, y = int(event.xdata), int(self.height - event.ydata - 1)
    selected_id = self.robot_listbox.get()
    for robot in self.robots:
        if robot['id'] == selected_id:
            if self.grid[y][x] == EMPTY:
                robot['goals'].append((x, y))
                self.log(f"Додано ціль для {robot['id']}: {x},{y}")
                self.draw_grid()

def draw_grid(self):
    self.ax.clear()
    self.ax.set_xlim(0, self.width)
    self.ax.set_ylim(0, self.height)
    self.ax.set_xticks(range(self.width))
    self.ax.set_yticks(range(self.height))
```

```
        self.ax.grid(True, color='#eee')
        self.ax.set_aspect("equal")
        self.ax.set_facecolor('white')

        for y in range(self.height):
            for x in range(self.width):
                if self.grid[y][x] == OBSTACLE:
                    self.ax.add_patch(patches.Rectangle((x, self.height - y - 1), 1, 1, color='black', zorder=1, alpha=0.85))
                elif self.grid[y][x] == DYNAMIC_OBSTACLE:
                    self.ax.add_patch(patches.Circle((x + 0.5, self.height - y - 0.5), 0.38, color='crimson', alpha=0.7, zorder=2))

        for robot in self.robots:
            if robot['path'] and robot['step'] < len(robot['path']):
                anim_x, anim_y = robot['anim_pos']
                pxs = [anim_x + 0.5]
                pys = [self.height - anim_y - 0.5]
                for px, py in robot['path'][robot['step']:]:
                    pxs.append(px + 0.5)
                    pys.append(self.height - py - 0.5)
                self.ax.add_line(Line2D(pxs, pys, linewidth=2, color='deepskyblue', alpha=0.4, zorder=2))

            for gx, gy in robot['goals']:
                self.ax.add_patch(patches.Circle((gx + 0.5, self.height - gy - 0.5), 0.22, color='limegreen', zorder=3, alpha=0.85))
```

```
            anim_x, anim_y = robot['anim_pos']
            self.ax.add_patch(patches.Circle((anim_x + 0.5, self.height - anim_y - 0.5), 0.50, color='gray', alpha=0.18, zorder=3))
            self.ax.add_patch(patches.Circle((anim_x + 0.5, self.height - anim_y - 0.5), 0.38, color='royalblue', ec='white', linewidth=2.5, zorder=4))
            self.ax.text(anim_x + 0.5, self.height - anim_y - 0.6, robot['id'], color='white', fontsize=12, fontweight='bold', ha='center', va='center', zorder=5)

            if robot['path']:
                progress = robot['step'] / len(robot['path'])
                self.ax.text(anim_x + 0.5, self.height - anim_y - 0.1, f"{int(progress * 100)}%", fontsize=8, color='#222', ha='center', va='center', zorder=5)

        self.canvas.draw()

    def astar(self, start, goal):
        queue = []
        heapq.heappush(queue, (0, start))
        came_from = {}
        cost_so_far = {start: 0}

        while queue:
            _, current = heapq.heappop(queue)
            if current == goal:
                break
            for dx, dy in DIRS:
```

```
            nx, ny = current[0] + dx, current[1] + dy
            if 0 <= nx < self.width and 0 <= ny < self.height:
                if self.grid[ny][nx] in (OBSTACLE, DYNAMIC_OBSTACLE):
                    continue
                next_node = (nx, ny)
                new_cost = cost_so_far[current] + 1
                if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                    cost_so_far[next_node] = new_cost
                    priority = new_cost + abs(goal[0] - nx) + abs(goal[1] - ny)
                    heapq.heappush(queue, (priority, next_node))
                    came_from[next_node] = current

    path = []
    current = goal
    while current != start:
        if current not in came_from:
            return []
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path

def start_simulation(self):
    self.start_button.configure(state="disabled")
    for robot in self.robots:
        robot['path'] = []
        current = robot['pos']
```

```
            for goal in robot['goals']:
                segment = self.astar(current, goal)
                if not segment:
                    self.log(f"{robot['id']}: не знайдено шлях до {goal}")
                    break
                robot['path'].extend(segment)
                current = goal
            robot['step'] = 0
            robot['anim_pos'] = np.array(robot['pos'], dtype=float)
            robot['is_moving'] = False
            robot['move_steps_left'] = 0
            robot['next_cell'] = None
            robot['completed_goals'] = 0
        self.simulate_step(infinite=False)

    def schedule_dynamic_obstacles(self):
        self.update_dynamic_obstacles_near_robots()
        self.master.after(1000, self.schedule_dynamic_obstacles)

    def update_dynamic_obstacles_near_robots(self):
        now = time.time()
        # Удаляем старые точки
        to_delete = []
        for (x, y), exp_time in list(self.dynamic_obstacles.items()):
            if now >= exp_time:
                self.grid[y][x] = EMPTY
                to_delete.append((x, y))
```

```
for pos in to_delete:
    del self.dynamic_obstacles[pos]

# Добавляем новые возле любого движущегося бота (работает всегда)
max_dynamic = self.robot_count * 1
n_dynamic = len(self.dynamic_obstacles)
if n_dynamic >= max_dynamic:
    return

free_to_create = max_dynamic - n_dynamic
for robot in self.robots:
    if free_to_create <= 0:
        break
    rx, ry = robot['pos']
    neighbors = []
    for dx, dy in DIRS:
        nx, ny = rx + dx, ry + dy
        if 0 <= nx < self.width and 0 <= ny < self.height:
            if self.grid[ny][nx] != EMPTY:
                continue
            is_goal = any((nx, ny) == (gx, gy) for r in self.robots for gx, gy in r['goals'])
            if is_goal:
                continue
            if (nx, ny) in self.dynamic_obstacles:
                continue
            neighbors.append((nx, ny))
    if neighbors:
```

```
            spawn = random.choice(neighbors)
            self.grid[spawn[1]][spawn[0]] = DYNAMIC_OBSTACLE
            self.dynamic_obstacles[spawn] = now + random.uniform(1, 4)
            free_to_create -= 1

    def move_robots_animated(self, infinite=False):
        if infinite and self.should_stop_infinite:
            self.infinite_mode = False
            self.start_button.configure(state="normal")
            return

        animating = False
        for robot in self.robots:
            if robot['is_moving']:
                if robot['move_steps_left'] > 0 and robot['next_cell'] is not None:
                    next_cell = robot['next_cell']
                    if self.grid[next_cell[1]][next_cell[0]] == DYNAMIC_OBSTACLE:
                        robot['is_moving'] = False
                        robot['move_steps_left'] = 0
                        robot['next_cell'] = None
                        if robot['goals']:
                            goal = robot['goals'][-1]
                            if robot['pos'] != goal:
                                segment = self.astar(robot['pos'], goal)
                                if segment:
                                    robot['path'] = segment
                                    robot['step'] = 0
```

```
                    self.log(f"{robot['id']}: перерахунок шляху через перешкоду")
                else:
                    self.log(f"{robot['id']}: шлях заблоковано. Чекає...")
            continue

        start = np.array(robot['anim_pos'])
        end = np.array(robot['next_cell'])
        t = 1 - robot['move_steps_left'] / ANIMATION_STEPS
        robot['anim_pos'] = start * (1 - t) + end * t
        robot['move_steps_left'] -= 1
        animating = True
        if robot['move_steps_left'] == 0:
            robot['anim_pos'] = np.array(robot['next_cell'], dtype=float)
            robot['pos'] = tuple(int(i) for i in robot['next_cell'])
            robot['is_moving'] = False
            robot['next_cell'] = None
            if robot['goals']:
                last_goal = robot['goals'][-1]
                if robot['pos'] == last_goal:
                    robot['completed_goals'] = robot.get('completed_goals', 0) + 1
                    if infinite:
                        self.assign_new_goal(robot)
                    else:
                        robot['goals'].pop()
            if robot['step'] < len(robot['path']):
                self.prepare_robot_move(robot)
```

```
            else:
                robot['is_moving'] = False
        elif robot['step'] < len(robot['path']):
            self.prepare_robot_move(robot)
            animating = True
    self.draw_grid()
    self.update_progress_box()
    if animating:
        self.master.after(self.sim_speed, lambda: self.move_robots_animated(infinite))
    else:
        self.simulate_step_continue(infinite)

def prepare_robot_move(self, robot):
    if robot['step'] < len(robot['path']):
        next_pos = robot['path'][robot['step']]
        # --- Если препятствие — пересчитываем маршрут ---
        if self.grid[next_pos[1]][next_pos[0]] == DYNAMIC_OBSTACLE:
            if robot['goals']:
                goal = robot['goals'][-1]
                if robot['pos'] != goal:
                    segment = self.astar(robot['pos'], goal)
                    if segment:
                        robot['path'] = segment
                        robot['step'] = 0
                        self.log(f"{robot['id']}: перерахунок шляху через перешкоду")
                    else:
                        self.log(f"{robot['id']}: шлях заблоковано. Чекає...")
```

```
                return
            robot['is_moving'] = True
            robot['move_steps_left'] = ANIMATION_STEPS
            robot['next_cell'] = next_pos
            robot['step'] += 1

    def simulate_step(self, infinite=False):
        self.move_robots_animated(infinite)

    def simulate_step_continue(self, infinite=False):
        moving = any(robot['step'] < len(robot['path']) for robot in self.robots)
        self.draw_grid()
        self.update_progress_box()
        if (moving or infinite) and (not (infinite and self.should_stop_infinite)):
            self.master.after(max(60, self.sim_speed), lambda: self.simulate_step(infinite))

def start_app(robot_count, width, height, pattern):
    root = ctk.CTk()
    RobotPlannerApp(root, width=width, height=height, robot_count=robot_count,
pattern=pattern)
    root.mainloop()

if __name__ == "__main__":
    app = ctk.CTk()

    def start_app_from_menu():
        robot_count = int(robot_entry.get())
```

```
    width = int(width_entry.get())
    height = int(height_entry.get())
    pattern = pattern_var.get()
    app.destroy()
    start_app(robot_count, width, height, pattern)

app.geometry("320x350")
ctk.set_appearance_mode("dark")
frame = ctk.CTkFrame(app)
frame.pack(pady=30)

ctk.CTkLabel(frame, text="Кількість роботів").pack(pady=5)
robot_entry = ctk.CTkEntry(frame)
robot_entry.insert(0, str(DEFAULTS['robot_count']))
robot_entry.pack(pady=5)

ctk.CTkLabel(frame, text="Ширина складу").pack(pady=5)
width_entry = ctk.CTkEntry(frame)
width_entry.insert(0, str(DEFAULTS['width']))
width_entry.pack(pady=5)

ctk.CTkLabel(frame, text="Висота складу").pack(pady=5)
height_entry = ctk.CTkEntry(frame)
height_entry.insert(0, str(DEFAULTS['height']))
height_entry.pack(pady=5)

ctk.CTkLabel(frame, text="Патерн складу").pack(pady=5)
```

```
    pattern_var = ctk.StringVar(value=DEFAULTS['pattern'])
    pattern_menu = ctk.CTkOptionMenu(frame, variable=pattern_var, values=['rows', 'columns', 'blocks'])
    pattern_menu.pack(pady=5)

    ctk.CTkButton(frame, text="Почати симуляцію", command=start_app_from_menu).pack(pady=14)

    app.mainloop()
```