ДОДАТОК А

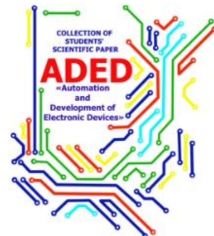
Апробація результатів роботи

Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

кафедра комп'ютерно-інтегрованих технологій, автоматизації та робототехніки

(КІТАР)



ЗБІРНИК

студентських наукових статей

«Автоматизація та приладобудування»

«Automation and Development of Electronic Devices»

ADED-2025

(Випуск 1)

[електронне видання]

Харків 2025

УДК 681.5; 004.4`6

АНАЛІЗ АЛГОРИТМІВ ПЛАНУВАННЯ ШЛЯХУ МОБІЛЬНОГО РОБОТА

Ільєнков Г.О.
Харківський національний університет радіоелектроніки
Україна, 61166, Харків, пр. Науки 14
E-mail: hlib.ilienkov@nure.ua

В роботі розглянуто актуальне питання автоматизації процесу пошуку оптимального шляху роботизованої платформи. Проведено аналіз існуючих алгоритмів для пошуку шляху, їх актуальності та особливостей застосування, також проаналізовано переваги та недоліки цих алгоритмів.

Ключові слова: автоматизація, пошух шляху, роботизавана платформа, алгоритм.

ANALYSIS OF MOBILE ROBOT PATH PLANNING ALGORITHMS

Ilienkov H.O.
Kharkiv National University of Radio Electronics
Ukraine, 61166, Kharkiv, Nauky ave. 14
E-mail: hlib.ilienkov@nure.ua

The paper considers the current issue of automating the process of finding the optimal path of a robotic platform. An analysis of existing algorithms for path finding, their relevance and application features, and an analysis of the advantages and disadvantages of these algorithms are also presented.

Keywords: automation, pathfinding, robotic platform, algorithm.

В даний момент на різних підприємствах зростає кількість роботизованих платформ відповідно до цього зростає необхідність у різних системах руху для цих платформ. Автоматизація цього процесу дозволяє знизити вплив людського фактора: коли роботизована платформа має можливість самостійно орієнтуватися у просторі і оператору не потрібно його постійно направляти, це дозволяє зменшити навантаження на людину а також це дозволяє економити ресурси та час [1].

Автономна навігація є цінним активом для мобільних роботів. Вона допомагає пом'якшити їхню залежність від втручання людини. Однак вона також спричиняє безліч завдань або проблем для вирішення, наприклад, планування шляху.

Для ефективної побудови маршруту роботизованої платформи потрібно дослідити методи та алгоритми автоматизації побудови шляху. Для цієї задачі алгоритми були класифіковані таким чином [2]:

1. Алгоритми пошуку на основі сітки, які знаходять шлях на основі мінімальної вартості переміщення на карті сітки. Їх можна використовувати для таких програм як мобільні роботи в 2D-середовищі. Однак вимоги до пам'яті реалізації алгоритмів на основі сітки можуть збільшуватися з числом вимірювань.

2. Алгоритм пошуку на основі вибірки які створюють дерево пошуку шляхом випадкової вибірки нових вузлів або конфігурації робота у просторі станів. Алгоритми на основі вибірки можуть бути придатними для багатовимірних просторів пошуку, такі як ті, які використовуються для пошуку допустимого набору конфігурацій для руки робота, щоб підібрати об'єкт. Генерація динамічно можливих шляхів для різних практичних програм робить планування на основі вибірки популярним, хоча воно і не забезпечує повного рішення.

3. Алгоритми оптимізації траєкторії які формують завдання планування шляху як завдання оптимізації, яка враховує бажану продуктивність транспортного засобу, відповідні

обмеження на динаміку транспортного засобу. Вони можуть застосовуватися для онлайн-планування шляху в невизначених умовах. Проте, залежно від складності завдання оптимізації, планування у реальному часі може бути нездійсненним.

Планування шляху, поряд із сприйняттям та системами управління, складають три основні будівельні блоки навігації для будь якої роботизованої платформи чи транспортного засобу. Планування шляху додає автономності до таких систем, як безпілотні автомобілі, роботизовані платформи, безпілотні наземні транспортні засоби та безпілотні літальні апарати. Алгоритми планування шляху зазвичай намагаються отримати найкращий шлях або принаймні допустиме наближення до нього. Найкращий шлях тут відноситься до оптимального, у тому сенсі, що результуючий шлях виходить шляхом мінімізації однієї або кількох функцій оптимізації. Наприклад, цей шлях може бути тим, який вимагає найменшої кількості часу. Це є вирішальне значення в таких місіях, як пошуково-рятувальні операції: жертви катастроф можуть звернутися за допомогою в ситуаціях, коли йдеться про життя чи смерть. Іншою функцією оптимізації, яку слід враховувати, може бути енергія роботизованої платформи.

Були досліджені наступні алгоритми [3-5]:

1. Dijkstra's Algorithm

Алгоритм Дейкстри може бути надзвичайно успішною стратегією для визначення найкоротшого шляху до цільової точки. Цей алгоритм працює ітераційним методом, обчислюючи найкоротшу відстань від початкової точки до всіх інших точок графа. Запропонований алгоритм паралелізує початковий процес: він починається з пошуку найкоротших шляхів від першого вузла до всіх інших вузлів, дозволяючи їм використовувати різні шляхи в найкращому випадку та той самий шлях до додаткових вузлів у гіршому випадку. Головним недоліком є те, що алгоритм, починає обчислення безпосередньо з центрального вузла всіх інших вузлів, і тому використовує інформацію незалежно від пройденого шляху.

Переваги:
– гарантує найкоротший шлях;
– простий для реалізації та розуміння.

Недоліки:
– повільний при великих просторах тому, що перевіряє всі можливі шляхи;
– не використовує евристик тому неефективний у складних динамічних ситуаціях.

2. A*(A-star)

A* – це алгоритм обходу графа та пошуку шляху, який використовується в багатьох галузях комп'ютерних наук завдяки своїй повноті, оптимальності та ефективності.

Його можна розглядати як розширення алгоритму Дейкстри. A* досягає кращої продуктивності в порівнянні з алгоритмом Дейкстри. Алгоритм A* – це алгоритм на основі евристичної функції для правильного планування шляху. Він обчислює значення евристичної функції у кожному вузлі робочої зони і включає перевірку надто великої кількості сусідніх вузлів знаходження оптимального рішення з нульовою ймовірністю зіткнення. Але це займає багато часу для обробки і сильно знижує швидкість роботи.

Порівняно з алгоритмом Дейкстри, алгоритм A* знаходить лише найкоротший шлях від зазначеного джерела до зазначеної мети, а не дерево найкоротших шляхів від зазначеного джерела до всіх можливих цілей. Це необхідний компроміс для використання евристики, спрямованої на конкретну мету.

Переваги:
– швидший за Дейкстра завдяки використанню евристики;
– гарантує оптимальний шлях, якщо евристика допустима;

– гнучкий та широко використовуваний в робототехніці.

Недоліки:
– якість залежить від правильно підібраної евристичної функції;
– велике споживання пам'яті при складному середовищі.

3. Artificial Bee Colony (ABC)

Основна ідея алгоритму була натхнена завданням збору меду бджолиними колоніями. Алгоритм ABC був схвалений багатьма вченими через його переваги, такі як швидка збіжність і відмінна продуктивність алгоритму при вирішенні задачі.

Алгоритм штучної колонії бджіл був розширений для вирішення багатоцільових задач, алгоритм MOABC потім часто використовувався в різних галузях. Проте про планування шляху з урахуванням багатоцільового штучного алгоритму рою рідко повідомляється.

Переваги:
– висока здатність до глобальної оптимізації;
– гнучкий і стійкий до умов середовища;
– підходить для багатокритеріальних задач.

Недоліки:
– висока складність у тонкому налаштуванні параметрів;
– не завжди стабільна продуктивність при великій кількості перешкод.

4. Rapidly-exploring Random Tree (RRT)

Класичні алгоритми планування шляху містять стратегію пошуку представлену A*, та стратегію вибірки, представлену ймовірнісною дорожньою картою, перша покладається на поточну позицію та історичну інформацію про позицію, і таким чином, схильна до локальних оптимальних рішень. Алгоритм Rapidly-exploring Random Tree поєднує в собі характеристики пошуку та вибірки:

– його характеристики пошуку виявляются в тому, що алгоритм починається з кореневого вузла і продовжує розгалуджуватися, ростячи як дерево, поки не знайде цільовий вузол;

– його характеристики вибірки виявляются у тому, що на алгоритм впливають випадкові точки вибірки у процесі розгалуження, тому алгоритм поліпшення Rapidly-exploring Random Tree має потенціал досягнення кращої продуктивності, ніж інші алгоритми.

Переваги:
– дуже ефективний для задач з високою розмірністю (наприклад 3Д-простори);
– працює добре в середовищах з багатьма перешкодами;
– легко реалізується, можна використовувати для рухомих роботів.

Недоліки:
– шлях може бути не оптимальним;
– потребує додаткових алгоритмів для згладжування маршруту.

5. Firefly Algorithm (FA)

Алгоритм світлячка (FA) був розроблений доктором Сінь-Ше Янгом в Кембриджському університеті в 2007 році. Цей алгоритм імітує шлюбну поведінку світлячків. Світлячки спілкуються змінюючи інтенсивність свого світла. Ця інтенсивність світла також має зворотну залежність від відстані. Чим більша відстань тим нижча інтенсивність світла.

Основна концепція алгоритму, описаного, полягає у наступному :

– між світлячками немає гендерних відмінностей, а привабливість залежить тільки від інтенсивності світіння кожного світлячка;

– притягання між двома світлячками пропорційно їхній яскравості і обернено пропорційно відстані між світлячками;

– яскравість світлячка визначається значенням цільової функції.

Алгоритм починається з отримання інформації про навколишнє середовище чи завдання. Після завершення світлячок ініціалізується, і його місцезнаходження на карті записується. Передатна функція обчислює яскравість кожного світляка. Після встановлення яскравості кожен світлячок порівнюється на основі основі його яскравості із сусідніми світлячками. Якщо цільової функції досягнуто, ітерація закінчується. В іншому випадку він продовжить обчислювати яскравість кожного світлячка і повторювати кроки знову.

Переваги:

– хороший для глобального пошуку та уникнення локальних мінімумів;
– простий у реалізації та параметризації;
– може адаптуватися до складного середовища.

Недоліки:

– повільна збіжність до точної траєкторії;
– вимагає багато ітерацій для точного результату;
– результати можуть варіюватись залежно від налаштувань

6. Particle Swarm Optimization

Алгоритм Particle Swarm Optimization починається з великої кількості можливих рішень. Ці частинки безперервно переміщуються у просторі пошуку для пошуку оптимального рішення. У цьому процесі кожна частка визначає свій рух у просторі пошуку. Вона робить це, поєднуючи свою поточну позицію з історичною найкращою позицією. Крім того вона включає деякі випадкові збурення. Коли знаходяться покращені позиції, ці позиції спрямовують рух рою. Процес повторюється багато разів у спробі знайти задовільне рішення, яке, хоч і не гарантується, як очікується, знайде глобальну або майже глобальну оптимальну відповідь наприкінці всіх циклів алгоритму.

Алгоритм можна узагальнити в чотири основні кроки, які повторюються доти, доки не буде виконано умову зупинки :

– призначте початкові випадкові положення на швидкості всім частинкам у просторі пошуку;
– оцініть придатність кожної частки;
– поновіть індивідуальні та світові найкращі позиції;
– обновіть швидкість та положення кожної частки.

Алгоритм Particle Swarm Optimization успішно застосовувався у багатьох галузях, таких як планування шляху та завдання оптимізації функцій. Було показано, що алгоритм Particle Swarm Optimization забезпечує кращі результати швидше та простіше порівняно з іншими методами.

Переваги:

– швидка збіжність у простих задачах;
– простота реалізації, мінімум параметрів;
– добре підходить для неперервних середовищ.

Недоліки:

– може застрягати в локальних мінімумах;
– погано працює у складних середовищах з багатьма перешкодами;
– не гарантує оптимальності рішення.

ВИСНОВКИ. Таким чином була зібрана та структурована інформація щодо алгоритмів для розроблення автоматизованої системи визначення мінімальної траєкторії руху роботизованої платформи. Після проведеного аналізу існуючих алгоритмів для розроблення автоматизованої системи визначення мінімальної траєкторії руху роботизованої платформи а також проведеного аналізу переваг та недоліків перелічених алгоритмів. Було виявлено, що для поставленої задачі більш за все підходять наступні алгоритми :

–Firefly Algorithm (FA);
–Rapidly-exploring Random Tree (RRT);
– A*(A-star).

ВИКОРИСТАНА ЛІТЕРАТУРА

1. 3rd International Conference on Innovations in Automation and Mechatronics Engineering, ICIAME 2016 Time-Efficient A* Algorithm for Robot Path Planning Akshay Kumar Guruji, Himansh Agarwal, D. K. Parsediya* https://www.sciencedirect.com/science/article/pii/S2212017316300111?ref=pdf_download&fr=RR-7&rr=933f1b1d785a1907 (дата звернення 19.04.2025)

2. Mobile Robot Path Planning Based on Improved Particle Swarm Optimization and Improved Dynamic Window Approach https://onlinelibrary.wiley.com/doi/10.1155/2023/6619841 (дата звернення 19.04.2025)

Recent advances in Rapidly-exploring random tree: A review https://www.sciencedirect.com/science/article/pii/S2405844024084822 (дата звернення 20.04.2025)

3. Firefly Algorithm: Bio-Inspired Decision Making Algorithm https://medium.com/@shantanuparab99/firefly-algorithm-bio-inspired-decision-making-algorithm-d25941725d14 (дата звернення 21.04.2025)

4. Невлюдов І. Ш., Андрусевич А. О., Євсєєв В. В., Новоселов С. П., Демська Н. П. Проектування мобільних маніпуляційних роботів: Монографія. – Х. :, 2022.

5. Path Planning for Autonomous Mobile Robots: A Review by José Ricardo Sánchez-Ibáñez https://www.mdpi.com/1424-8220/21/23/7898 (23.04.2025)

6. Yevsieiev, V., Abu-Jassar, A., Maksymova, S., & Demska, N. (2025). Development of a model for recognizing various objects and tools in a collaborative robot workspace. *ACUMEN: International journal of multidisciplinary research*, *2*(1), 224-239.

7. Yevsieiev V. Comparison of Functional Capabilities of Classic Manipulator Robots and Collaborative Robots / V. Yevsieiev, N. Demska // Digital innovation & sustainable development 2024 : Proceedings of I st International Conference, November 15, 2024. - Kharkiv, 2024. - P.16-17.

8. Yevsieiev V. Using Contouring Algorithms to Select Objects in the Robots' Workspace / V. Yevsieiev, S. Maksymova, N. Demska // Technical science research in Uzbekistan. – 2024. – Vol. 2(2). – P. 32–42.

9. Nevliudov, I., Yevsieiev, V., Maksymova, S., Demska, N., Starodubcev, N., & Klymenko, O. (2023, September). Monitoring System Development for Equipment Upgrade for IIoT. In *2023 IEEE 5th International Conference on Modern Electrical and Energy System (MEES)* (pp. 1-5). IEEE.

10. Nevliudov, I., Yevsieiev, V., Maksymova, S., Demska, N., Kolesnyk, K., & Miliutina, O. (2023, September). Mobile Robot Navigation System Based on Ultrasonic Sensors. In *2023 IEEE XXVIII International Seminar/Workshop on Direct and Inverse Problems of Electromagnetic and Acoustic Wave Theory (DIPED)* (Vol. 1, pp. 247-251). IEEE.

11. Chala, O., Yevsieiev, V., Maksymova, S., & Abu-Jassar, A. (2025). USING THE HUMAN FACE RECOGNITION METHOD BASED ON THE MOBILENETV2 NEURAL NETWORK IN AUTHENTICATION SYSTEMS. *Multidisciplinary Journal of Science and Technology*, *5*(3), 882-895.

Науковий керівник: *Демська Наталія Павлівна, доцент, кандидат технічних наук, доцент кафедри КІТАР Харківського національного університету радіоелектроніки*

ДОДАТОК Б

Код програми

```
import numpy as np
import matplotlib.pyplot as plt
import random
import time
import tkinter as tk
from tkinter import filedialog

grid_size = 20
start = (0, 0)
goal = (18, 15)
directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
num_obstacles = 30
max_generation_time = 120
n_fireflies = 30
n_iter = 50

obstacle_map = np.zeros((grid_size, grid_size))
main_path = []
alt_path = []
best_path = []

def is_free(cell, obstacle_map, allow_buffer=False):
    x, y = cell
    if not (0 <= x < grid_size and 0 <= y < grid_size):
        return False
    value = obstacle_map[x, y]
    return value == 0 or (allow_buffer and value == 0.5)
```

```
def clear_zone_around(cell, obstacle_map, radius=2):
    x0, y0 = cell
    for dx in range(-radius, radius + 1):
        for dy in range(-radius, radius + 1):
            x, y = x0 + dx, y0 + dy
            if 0 <= x < grid_size and 0 <= y < grid_size:
                obstacle_map[x, y] = 0

def bfs_check_path(start, goal, obstacle_map):
    from collections import deque
    queue = deque([start])
    visited = set([start])
    while queue:
        current = queue.popleft()
        if current == goal:
            return True
        for dx, dy in directions:
            next_cell = (current[0] + dx, current[1] + dy)
            if is_free(next_cell, obstacle_map) and next_cell not in visited:
                queue.append(next_cell)
                visited.add(next_cell)
    return False

def generate_valid_map():
    start_time = time.time()
    while time.time() - start_time < max_generation_time:
        obstacle_map = np.zeros((grid_size, grid_size))
        obstacles = set()
        while len(obstacles) < num_obstacles:
```

```
            x, y = random.randint(0, grid_size - 1), random.randint(0, grid_size - 1)
            cell = (x, y)
            if cell != start and cell != goal:
                obstacles.add(cell)
        for (x, y) in obstacles:
            obstacle_map[x, y] = 1
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < grid_size and 0 <= ny < grid_size:
                        if obstacle_map[nx, ny] == 0:
                            obstacle_map[nx, ny] = 0.5
        clear_zone_around(start, obstacle_map)
        clear_zone_around(goal, obstacle_map)
        if bfs_check_path(start, goal, obstacle_map):
            return obstacle_map
    return obstacle_map

def generate_maze_with_multiple_paths():
    maze = np.ones((grid_size, grid_size))
    visited = np.zeros((grid_size, grid_size), dtype=bool)
    def in_bounds(x, y): return 0 <= x < grid_size and 0 <= y < grid_size
    def dfs(x, y):
        visited[x, y] = True
        maze[x, y] = 0
        dirs = [(-2, 0), (2, 0), (0, -2), (0, 2)]
        random.shuffle(dirs)
        for dx, dy in dirs:
            nx, ny = x + dx, y + dy
            if in_bounds(nx, ny) and not visited[nx, ny]:
```

```
            wall_x = x + dx // 2
            wall_y = y + dy // 2
            maze[wall_x, wall_y] = 0
            dfs(nx, ny)
    sx, sy = start
    if sx % 2 == 0: sx += 1
    if sy % 2 == 0: sy += 1
    dfs(sx, sy)
    clear_zone_around(start, maze)
    clear_zone_around(goal, maze)
    def simple_path(from_point):
        path = []
        current = from_point
        visited_local = set()
        while current != goal and len(path) < 300:
            path.append(current)
            visited_local.add(current)
            next_moves = [(current[0] + dx, current[1] + dy) for dx, dy in directions]
            next_moves = [m for m in next_moves if in_bounds(*m) and maze[m[0],
m[1]] == 0 and m not in visited_local]
            if not next_moves:
                break
            current = min(next_moves, key=lambda c: np.linalg.norm(np.array(goal) -
np.array(c)))
        if current == goal:
            path.append(goal)
        return path
    main_path = simple_path(start)
    alt_start = (min(start[0] + 4, grid_size - 1), start[1])
    alt_path = simple_path(alt_start) if maze[alt_start[0], alt_start[1]] == 0 else []
```

```
    return maze, main_path, alt_path


def smart_path():
    path = [start]
    current = start
    visited = set([start])
    while current != goal and len(path) < 100:
        primary_moves, buffer_moves = [], []
        for dx, dy in directions:
            next_cell = (current[0] + dx, current[1] + dy)
            if not (0 <= next_cell[0] < grid_size and 0 <= next_cell[1] < grid_size):
                continue
            if obstacle_map[next_cell[0], next_cell[1]] == 0 and next_cell not in
visited:
                dist = np.linalg.norm(np.array(goal) - np.array(next_cell))
                primary_moves.append((dist, next_cell))
            elif obstacle_map[next_cell[0], next_cell[1]] == 0.5 and next_cell not in
visited:
                dist = np.linalg.norm(np.array(goal) - np.array(next_cell))
                buffer_moves.append((dist, next_cell))
        if primary_moves:
            next_cell = min(primary_moves)[1]
        elif buffer_moves:
            next_cell = min(buffer_moves)[1]
        else:
            break
        path.append(next_cell)
        visited.add(next_cell)
        current = next_cell
    return path
```

```
def smart_path_maze(maze):
    path = [start]
    current = start
    visited = set([start])
    while current != goal and len(path) < 300:
        possible_moves = [
            (current[0] + dx, current[1] + dy) for dx, dy in directions
            if 0 <= current[0] + dx < grid_size and 0 <= current[1] + dy < grid_size
        ]
        valid_moves = [
            cell for cell in possible_moves
            if maze[cell[0], cell[1]] == 0 and cell not in visited
        ]
        if not valid_moves:
            break
        current = min(valid_moves, key=lambda c: np.linalg.norm(np.array(goal) -
np.array(c)))
        path.append(current)
        visited.add(current)
    return path

def path_has_collision(path):
    return any(obstacle_map[x, y] >= 0.5 for x, y in path)

def fitness(path):
    if path[-1] != goal:
        return 1e6
    return len(path)
```

```
def load_map_from_file():
    file_path = filedialog.askopenfilename(title="Select map file (CSV)", 
filetypes=[("CSV Files", "*.csv")])
    if file_path:
        try:
            loaded_map = np.loadtxt(file_path, delimiter=',')
            if loaded_map.shape != (grid_size, grid_size):
                raise ValueError("Incorrect map size")
            return loaded_map
        except Exception as e:
            print("Error loading map:", e)
    return None

def start_gui():
    def on_generate():
        nonlocal maze_var
        global obstacle_map, main_path, alt_path
        if maze_var.get():
            obstacle_map, main_path, alt_path = generate_maze_with_multiple_paths()
        else:
            obstacle_map = generate_valid_map()
            main_path = []
            alt_path = []
        run_firefly()

    def on_load():
        global obstacle_map, main_path, alt_path
        loaded = load_map_from_file()
        if loaded is not None:
            obstacle_map[:] = loaded
```

```
        main_path = []
        alt_path = []
        run_firefly()

    def run_firefly():
        global best_path
        if maze_var.get():
            fireflies = [smart_path_maze(obstacle_map) for _ in range(n_fireflies)]
        else:
            fireflies = [smart_path() for _ in range(n_fireflies)]

        for _ in range(n_iter):
            for i in range(n_fireflies):
                for j in range(n_fireflies):
                    if fitness(fireflies[j]) < fitness(fireflies[i]):
                        cut = random.randint(1, min(len(fireflies[i]), len(fireflies[j])) - 2)
                        new_path = fireflies[i][:cut]
                        current = new_path[-1]
                        visited = set(new_path)
                        while current != goal and len(new_path) < 100:
                            next_moves = [(current[0] + dx, current[1] + dy) for dx, dy in
directions]
                            next_moves = [m for m in next_moves if is_free(m,
obstacle_map) and m not in visited]
                            if not next_moves:
                                break
                            current = min(next_moves, key=lambda c:
np.linalg.norm(np.array(goal) - np.array(c)))
                            new_path.append(current)
                            visited.add(current)
```

```
                if not path_has_collision(new_path) and new_path[-1] == goal:
                    fireflies[i] = new_path
        best_path = min(fireflies, key=fitness)
        draw_result()

    def draw_result():
        plt.figure(figsize=(8, 8))
        plt.imshow(obstacle_map.T, origin='lower', cmap=plt.cm.gray_r)
        best_path_np = np.array(best_path)
        plt.plot(best_path_np[:, 0], best_path_np[:, 1], 'r-o', label='Best Path')
        if main_path:
            plt.plot(np.array(main_path)[:, 0], np.array(main_path)[:, 1], 'g--',
label='Maze Path')
        if alt_path:
            plt.plot(np.array(alt_path)[:, 0], np.array(alt_path)[:, 1], color='purple',
linestyle='--', label='Alt Path')
        plt.plot(start[0], start[1], 'go', label='Start')
        plt.plot(goal[0], goal[1], 'bo', label='Goal')
        plt.title("Path")
        plt.grid(True)
        plt.legend()
        plt.show()

    root = tk.Tk()
    root.title("Path Planner")
    frame = tk.Frame(root)
    frame.pack(padx=10, pady=10)
    maze_var = tk.BooleanVar()
    tk.Button(frame, text="Генерувати карту",
command=on_generate).pack(fill="x", pady=5)
```

```
    tk.Button(frame, text="Завантажити карту з файлу",
command=on_load).pack(fill="x", pady=5)
    tk.Checkbutton(frame, text="Maze режим",
variable=maze_var).pack(anchor='w')
    tk.Label(frame, text="* Карта повинна бути CSV з розміром 20x20").pack()
    root.mainloop()

if __name__ == "__main__":
    start_gui()
```

ДОДАТОК В

Демонстраційний матеріал

Формат	Позиція.	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кільк.	Прим.
			Текстові документи		
А4	1	ГЮИК 442561.008 ПЗ	Пояснювальна записка	54 с.	
			Інші документи		
А4	2		Апробація результатів	2 с.	
			роботи		
А4	3		Код програми	10 с.	
А4	4		Демонстраційний матеріал	12 с.	

Змін.	Арк.	Номер докум.	Підпис	Дата	ГЮИК 442561.008 ВД		
Розроб.		Ільєнков Г. О.			Розроблення гібридного механізму пересування для малого мобільного робота	Літера	Арку	Ар
Перевір.		Демська Н.П.				Н	1	1
Н.контр.		Демська Н.П.				Кафедра КІТАР ХНУРЕ		
Затв.		Невлюдов І.Ш.						