

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение высшего образования «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина» (УрФУ)

Физико-технологический институт

Кафедра «Технической физики»

Оценка

Преподаватель

Кашин И.В.

**ПАРАЛЛЕЛИЗАЦИЯ АЛГОРИТМА ПОИСКА   
РЕШЕНИЯ ЛАБИРИНТА**

ОТЧЕТ

по лабораторной работе №7

|  |  |  |
| --- | --- | --- |
| Студент | Черняков Матвей Сергеевич | ФИО студента |

|  |
| --- |
| Специальность (направление подготовки) |
| 09.03.02 Информационные системы и технологии | |

|  |  |
| --- | --- |
| Группа | Фт-420008 |

Екатеринбург

2025

**ОГЛАВЛЕНИЕ**

[ОПИСАНИЕ ЗАДАЧИ 3](#_Toc212600700)

[ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ 6](#_Toc212600701)

[РЕШЕНИЕ ЗАДАЧИ 7](#_Toc212600702)

[ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ 9](#_Toc212600703)

[ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА 11](#_Toc212600704)

ОПИСАНИЕ ЗАДАЧИ

Целью данной лабораторной работы является изучение одного из подходов с использованием **эвристических алгоритмов поиска**, основанных на динамике агентов в пространстве. Основная задача заключается в исследовании зависимости времени выполнения алгоритма от количества используемых процессов.

Эвристические (или метаэвристические) алгоритмы представляют собой класс методов оптимизации, которые используются для решения задач в пространствах с большим числом возможных вариантов, где полный перебор невозможен или непрактичен. Основная идея таких алгоритмов заключается в использовании локальной информации для принятия решений, которые в совокупности приводят к глобально оптимальному или близкому к оптимальному решению.

Алгоритм основан на фундаментальной идее: превращение независимых чисел во взаимосвязанные через **нормирование** (преобразование в проценты или вероятности).

Рассмотрим простой пример. Если у нас есть четыре независимых числа: 2, 3, 6, 8, они никак не связаны между собой. Изменение одного не влияет на другие. Но если мы превратим их в распределение (проценты от суммы):

Сумма = 2 + 3 + 6 + 8 = 19

Распределение: 2/19, 3/19, 6/19, 8/19

Теперь любое изменение любого числа автоматически влияет на все остальные через изменение знаменателя.

В алгоритме используется мультиагентный подход. Агенты – это абстрактные сущности, которые: имеют позицию в пространстве (для 2D изображения: координаты x, y). Могут перемещаться по дискретной сетке (8 направлений: вверх, вниз, влево, вправо и по диагоналям). Принимают решения на основе локальной информации. Оставляют след в виде увеличения значения количества посещений в точке посещения.

Ключевой задачей является моделирование движения множества агентов по лабиринту, представленному в виде чёрно-белого изображения. Белые пиксели обозначают свободное пространство (1), чёрные — стены (0). Вход в лабиринт находится слева, выход – справа. Каждому агенту разрешено перемещаться на одну клетку в любую из восьми соседних (по вертикали, горизонтали или диагонали), при условии, что новая клетка проходима.

Для того чтобы агенты могли ориентироваться в лабиринте, на его основе вычисляется специальная метрика α – «**градиент направленности к выходу**». Она рассчитывается при помощи обхода в ширину от точки выхода и характеризует расстояние до выхода: чем ближе клетка к выходу, тем больше значение α. Таким образом, каждый агент на своём шаге стремится выбрать одно из направлений, увеличивающее значение α, то есть приближающее его к выходу.

Ключевой элемент алгоритма – правило перемещения агента. На каждом шаге агент:

1. Оценивает соседние позиции (до 4 направлений)

2. Вычисляет коэффициент K для каждой пробной позиции ():

*где n(x, y) – количество посещений точки, β – степень награды (>1 усиливает движение вправо), γ – степень штрафа. Для примера брал β=1.5, γ=0.8*

3. Выбирает направление с минимальным значением K

4. Перемещается в выбранную позицию

5. Обновляет динамическое распределение:

K близкое к 0 (недобор): В данной точке динамическое распределение меньше эталонного – агент стремится туда переместиться

K >> 0 (перебор): В данной точке динамическое распределение больше эталонного → агент избегает этой точки

Важная деталь реализации: 4 возможных направления проверяются в случайном порядке на каждом шаге. Это достигается через:

1. Предварительную генерацию массива случайных перестановок   
   чисел 0-7
2. Использование новой перестановки на каждом шаге агента
3. Циклическое переиспользование массива перестановок

Несмотря на то, что каждый агент принимает решения на основе локальной информации (только значения в соседних клетках), коллективное поведение всех агентов приводит к глобальному результату – воспроизведению всего изображения.

Механизм достижения глобального эффекта:

1. Нормирование создает взаимосвязь: изменение M (общей нормы) влияет на все вычисления K во всех точках пространства
2. Отрицательная обратная связь: если агенты слишком долго находятся в одной области, то агенты уходят
3. Самоорганизация: система естественным образом стремится к равномерному покрытию согласно эталонному распределению

Это позволяет алгоритму избегать локальных экстремумов – классической проблемы эвристических методов.

ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ

Параллелизация реализована с использованием механизма multiprocessing в Python. Каждый процесс обрабатывает подмножество агентов: группы агентов делятся по диапазонам индексов, и каждый процесс независимо вычисляет новое положение агентов своего батча.

Основной проблемой многопроцессной обработки является необходимость обмена состоянием между процессами. Для минимизации накладных расходов в программе применяется shared memory (RawArray) – общий разделяемый массив, благодаря чему позиции агентов, сама карта лабиринта и метрика альфа не копируются между процессами, а используются совместно.

Архитектура параллельной обработки выглядит следующим образом:

1. Главный процесс загружает лабиринт, вычисляет метрику α и создаёт разделяемые буферы.
2. Создаётся пул процессов (Pool), каждому из которых передаются дескрипторы shared memory.
3. На каждом шаге симуляции агенты разбиваются на батчи, и пул процессов параллельно вычисляет новое положение для каждой группы агентов.
4. После завершения шага главный процесс синхронизирует обновлённые координаты агентов и увеличивает счётчик итераций.
5. Пул создаётся один раз и переиспользуется в течение всего эксперимента, что значительно снижает накладные расходы.

Таким образом, каждый процесс работает автономно над своей частью агентов, не вмешиваясь в чужие данные, а доступ к общим массивам осуществляется без копирования. За счёт этого достигается ускорение вычислений, особенно при большом количестве агентов.

РЕШЕНИЕ ЗАДАЧИ

**MazeNavigationAgent** – класс агента, который перемещается по лабиринту размером 20×20 клеток, находит кратчайший путь от стартовой позиции до цели с помощью алгоритма, учитывая препятствия (стены), и имитирует задержки при движении, чтобы создать вычислительную нагрузку для тестирования параллелизма.

class MazeNavigationAgent:  
 *"""  
 Алгоритм навигации агентов по лабиринту с использованием градиента альфа  
 """  
…*

**init\_worker\_maze** – функция инициализации процесса-воркера, которая создает глобальный экземпляр лабиринта и агента для каждого рабочего процесса, чтобы избежать дублирования данных и повторной инициализации при обработке каждого батча агентов.

def init\_worker\_maze(maze\_array, alpha\_array, agents\_x\_array, agents\_y\_array,  
 agents\_finished\_array, maze\_shape, n\_agents, directions,  
 random\_orders, exit\_point):  
 *"""Инициализация worker процесса"""  
…*

**process\_agent\_batch\_maze** – функция обработки батча агентов, которая выполняется в отдельном процессе, запускает навигацию для каждого агента в батче и возвращает статистику (количество успешных навигаций и общее количество шагов), используя предварительно инициализированный агент из глобальной переменной воркера.

def process\_agent\_batch\_maze(args):  
 *"""Обработка батча агентов"""*  
…

**ParallelMazeNavigator** – класс-менеджер для параллельного выполнения навигации множества агентов, создает пул процессов с заданным количеством воркеров, распределяет агентов по батчам, отправляет их на обработку в разные процессы и собирает результаты, измеряя общее время выполнения.

class ParallelMazeNavigator(MazeNavigationAgent):  
 *"""Параллельная версия навигатора по лабиринту"""*

…

**plot\_maze\_scalability** – функция визуализации результатов экспериментов, строит два графика: зависимость времени выполнения от количества агентов при фиксированном числе воркеров и зависимость времени от числа воркеров при фиксированном количестве агентов, чтобы продемонстрировать эффективность параллелизации.

def plot\_maze\_scalability(results: dict, save\_path: str = 'experiment\_scalability.png'):  
 *"""График масштабируемости"""*

…

**experiment\_scalability** – главная экспериментальная функция, которая проводит серию тестов с различными комбинациями количества агентов (от 10 до 1000) и воркеров (от 1 до 8), измеряет время выполнения для каждой комбинации, собирает статистику и визуализирует результаты для анализа масштабируемости параллельной обработки.

def experiment\_scalability(maze\_path: str, n\_agents: int = 100, max\_processes: int = 24):  
 *"""  
 Эксперимент: Масштабируемость от 1 до 24 процессов  
 Задача - первый агент достигает выхода  
 """* …

Исходный лабиринт представлен на рисунке 1. Он сгенерирован с помощью специализированного интернет-ресурса.



Рисунок 1 – Лабиринт

ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ

Ключевыми параметрами является количество ядер (12) и количество потоков (24). Для данных характеристик было проведено тестирование сравнения идеального и условного времени при M от 1 до 24. Результаты тестирования представлены на графике (рисунок 2). Точные значения представлены в приложении Б;

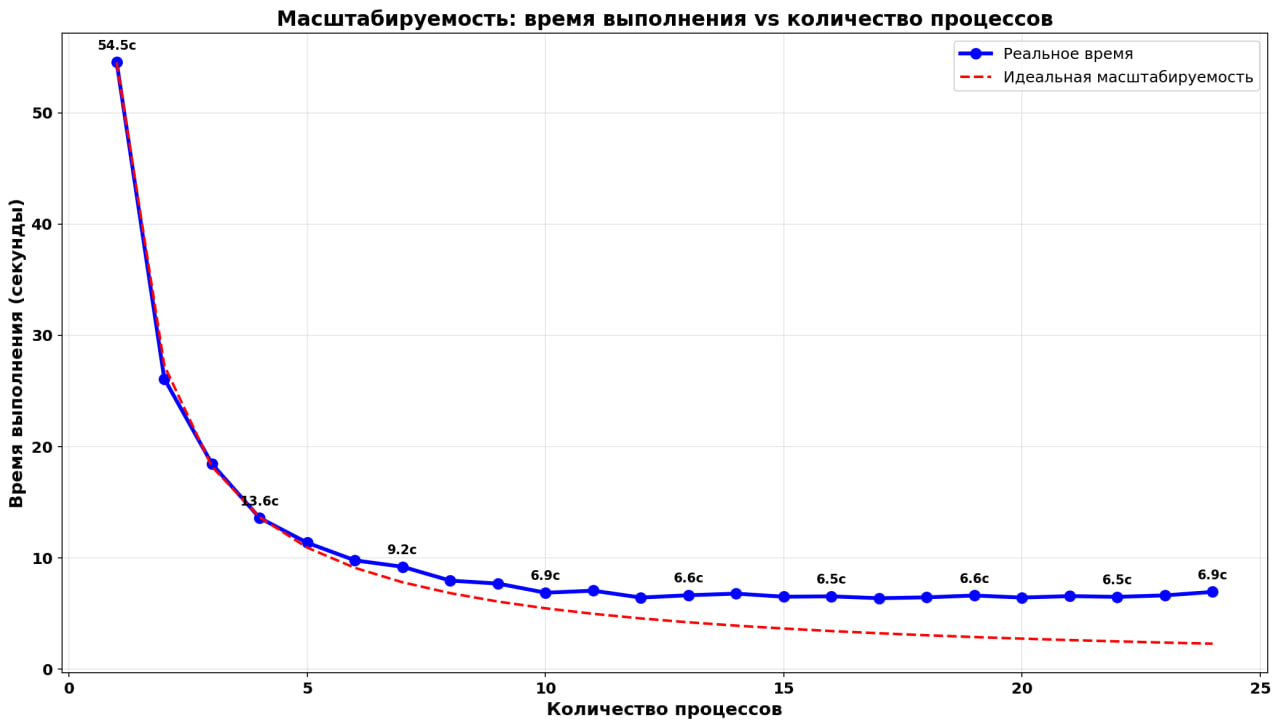


Рисунок 2 – График эксперимента 1

По графику для эксперимента 1 видно, что реальное время достаточно близко к идеальному. Отсюда можно понять, что можно выбрать оптимальное количество потоков для распараллеливания данного алгоритма сортировки (здесь это 12 потоков).

Распределение метрики K представлено на рисунке 3.

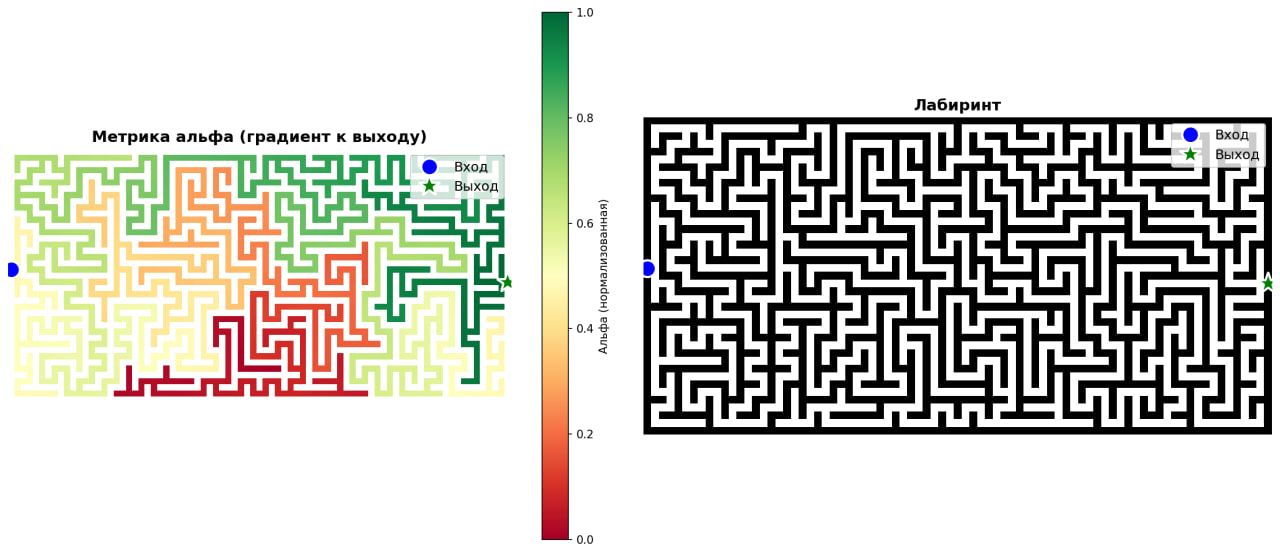


Рисунок 3 – Градиент к выходу

Путь агента от начала к выходу представлен на рисунке 4.



Рисунок 4 – Решение лабиринта

ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА

import numpy as np  
from PIL import Image  
import matplotlib.pyplot as plt  
from multiprocessing import Pool, cpu\_count, RawArray  
import time  
from typing import Tuple, List  
from collections import deque  
import ctypes  
import atexit  
  
  
class MazeNavigationAgent:  
 *"""  
 Алгоритм навигации агентов по лабиринту с использованием градиента альфа  
 """* def \_\_init\_\_(self, maze\_path: str, n\_agents: int = 100, seed: int = 42):  
 *"""  
 Инициализация алгоритма  
  
 Args:  
 maze\_path: путь к изображению лабиринта  
 n\_agents: количество агентов  
 seed: seed для воспроизводимости  
 """* np.random.seed(seed)  
  
 *# Загрузка лабиринта* self.load\_maze(maze\_path)  
  
 *# Параметры* self.n\_agents = n\_agents  
 self.height, self.width = self.maze.shape  
  
 *# Вычисляем метрику альфа (градиент к выходу)* self.compute\_alpha\_metric()  
  
 *# Инициализация агентов у входа* self.initialize\_agents()  
  
 *# 8 направлений: (dx, dy)* self.directions = [  
 (-1, -1), (0, -1), (1, -1),  
 (-1, 0), (1, 0),  
 (-1, 1), (0, 1), (1, 1)  
 ]  
  
 *# Подготовка случайных порядков проверки направлений* self.prepare\_random\_orders(100000)  
 self.current\_order\_idx = 0  
  
 *# История для визуализации* self.history = []  
  
 *# Статистика* self.successful\_agents = 0  
 self.total\_steps = 0

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def load\_maze(self, maze\_path: str):  
 *"""Загрузка лабиринта из изображения"""* img = Image.open(maze\_path).convert('L')  
  
 *# Преобразуем в numpy массив* img\_array = np.array(img, dtype=np.uint8)  
  
 *# Бинаризация: темные пиксели (< 128) = стены (0), светлые = проходы (1)* self.maze = (img\_array > 128).astype(np.int32)  
  
 self.height, self.width = self.maze.shape  
  
 print(f"Лабиринт загружен: {self.maze.shape}")  
 print(f"Проходимых клеток: {np.sum(self.maze)}")  
 print(f"Стен: {np.sum(1 - self.maze)}")  
  
 def find\_entry\_exit(self) -> Tuple[Tuple[int, int], Tuple[int, int]]:  
 *"""  
 Поиск входа (слева) и выхода (справа) в лабиринте  
  
 Returns:  
 ((entry\_x, entry\_y), (exit\_x, exit\_y))  
 """  
 # Ищем вход в левой колонке* entry = None  
 for y in range(self.height):  
 if self.maze[y, 0] == 1:  
 entry = (0, y)  
 break  
  
 *# Ищем выход в правой колонке* exit\_point = None  
 for y in range(self.height):  
 if self.maze[y, self.width - 1] == 1:  
 exit\_point = (self.width - 1, y)  
 break  
  
 if entry is None:  
 *# Если нет прохода на границе, ищем ближайший к левому краю* passable\_left = np.where(self.maze[:, :self.width // 4] == 1)  
 if len(passable\_left[0]) > 0:  
 idx = np.argmin(passable\_left[1])  
 entry = (passable\_left[1][idx], passable\_left[0][idx])  
  
 if exit\_point is None:  
 *# Если нет прохода на границе, ищем ближайший к правому краю* passable\_right = np.where(self.maze[:, 3 \* self.width // 4:] == 1)  
 if len(passable\_right[0]) > 0:  
 idx = np.argmax(passable\_right[1])  
 exit\_point = (3 \* self.width // 4 + passable\_right[1][idx], passable\_right[0][idx])

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

if entry is None or exit\_point is None:  
 raise ValueError("Не удалось найти вход или выход в лабиринте")  
  
 print(f"Вход найден: {entry}")  
 print(f"Выход найден: {exit\_point}")  
  
 return entry, exit\_point  
  
 def compute\_alpha\_metric(self):  
 *"""  
 Вычисление метрики альфа через волновой алгоритм (BFS) от выхода.  
 Альфа = -расстояние\_до\_выхода, то есть чем ближе к выходу, тем больше альфа.  
 """* entry, exit\_point = self.find\_entry\_exit()  
 self.entry = entry  
 self.exit = exit\_point  
  
 *# Инициализация: все клетки с -inf, кроме выхода* self.alpha = np.full((self.height, self.width), -np.inf, dtype=np.float64)  
  
 *# BFS от выхода* queue = deque([exit\_point])  
 self.alpha[exit\_point[1], exit\_point[0]] = 0  
 visited = set([exit\_point])  
  
 *# 8 направлений для BFS* directions\_8 = [  
 (-1, -1), (0, -1), (1, -1),  
 (-1, 0), (1, 0),  
 (-1, 1), (0, 1), (1, 1)  
 ]  
  
 while queue:  
 x, y = queue.popleft()  
 current\_dist = self.alpha[y, x]  
  
 for dx, dy in directions\_8:  
 nx, ny = x + dx, y + dy  
  
 *# Проверка границ и проходимости* if (0 <= nx < self.width and 0 <= ny < self.height and  
 self.maze[ny, nx] == 1 and (nx, ny) not in visited):  
 *# Расстояние с учетом диагоналей* if dx != 0 and dy != 0:  
 dist = np.sqrt(2) *# диагональ* else:  
 dist = 1 *# прямое движение* self.alpha[ny, nx] = current\_dist - dist *# отрицательное расстояние* visited.add((nx, ny))  
 queue.append((nx, ny))  
  
 *# Нормализация альфа к диапазону [0, 1] для наглядности* valid\_alpha = self.alpha[self.alpha > -np.inf]

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

if len(valid\_alpha) > 0:  
 min\_alpha = np.min(valid\_alpha)  
 max\_alpha = np.max(valid\_alpha)  
 self.alpha\_normalized = np.where(  
 self.alpha > -np.inf,  
 (self.alpha - min\_alpha) / (max\_alpha - min\_alpha + 1e-10),  
 0  
 )  
 else:  
 self.alpha\_normalized = np.zeros\_like(self.alpha)  
  
 print(f"Метрика альфа вычислена")  
 print(f"Диапазон альфа: [{np.min(valid\_alpha):.2f}, {np.max(valid\_alpha):.2f}]")  
 print(f"Достижимых клеток: {len(valid\_alpha)}")  
  
 def initialize\_agents(self):  
 *"""Инициализация агентов у входа в лабиринт"""* entry\_x, entry\_y = self.entry  
  
 *# ВСЕ агенты стартуют строго в точке входа* self.agents\_x = np.full(self.n\_agents, entry\_x, dtype=np.int32)  
 self.agents\_y = np.full(self.n\_agents, entry\_y, dtype=np.int32)  
 self.agents\_finished = np.zeros(self.n\_agents, dtype=bool)  
  
 *# Вычисляем альфа для стартовой позиции* start\_alpha = self.alpha[entry\_y, entry\_x]  
  
 print(f"Агенты инициализированы: {self.n\_agents}")  
 print(f"Стартовая позиция: ({entry\_x}, {entry\_y})")  
 print(f"Альфа на старте: {start\_alpha:.2f}")  
  
 def prepare\_random\_orders(self, n\_orders: int = 100000):  
 *"""Подготовка случайных порядков проверки направлений"""* self.random\_orders = np.zeros((n\_orders, 8), dtype=np.int32)  
 for i in range(n\_orders):  
 self.random\_orders[i] = np.random.permutation(8)  
  
 def get\_next\_order(self) -> np.ndarray:  
 *"""Получить следующий случайный порядок проверки"""* order = self.random\_orders[self.current\_order\_idx]  
 self.current\_order\_idx = (self.current\_order\_idx + 1) % len(self.random\_orders)  
 return order  
  
 def move\_agent(self, agent\_idx: int) -> Tuple[int, int]:  
 *"""  
 Перемещение одного агента  
  
 Returns:  
 новые координаты (x, y)  
 """* if self.agents\_finished[agent\_idx]:  
 return self.agents\_x[agent\_idx], self.agents\_y[agent\_idx]

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

x, y = self.agents\_x[agent\_idx], self.agents\_y[agent\_idx]  
  
 *# Проверяем, достиг ли агент выхода (в радиусе 2 клеток)* exit\_x, exit\_y = self.exit  
 if abs(x - exit\_x) <= 2 and abs(y - exit\_y) <= 2:  
 self.agents\_finished[agent\_idx] = True  
 self.successful\_agents += 1  
 return x, y  
  
 current\_alpha = self.alpha[y, x]  
  
 *# Собираем все возможные направления с их альфами* possible\_moves = []  
  
 for dir\_idx in range(8):  
 dx, dy = self.directions[dir\_idx]  
 new\_x, new\_y = x + dx, y + dy  
  
 *# Проверка границ и проходимости* if (0 <= new\_x < self.width and 0 <= new\_y < self.height and  
 self.maze[new\_y, new\_x] == 1):  
 alpha\_val = self.alpha[new\_y, new\_x]  
 possible\_moves.append((new\_x, new\_y, alpha\_val))  
  
 if not possible\_moves:  
 *# Застряли - остаемся на месте* return x, y  
  
 *# Сортируем по альфе (по убыванию)* possible\_moves.sort(key=lambda m: m[2], reverse=True)  
  
 *# Выбираем из топ-K лучших направлений (K=3 или меньше если вариантов мало)* top\_k = min(3, len(possible\_moves))  
 candidates = possible\_moves[:top\_k]  
  
 *# Случайный выбор среди топ-K с уникальным seed* np.random.seed(self.total\_steps \* 1000 + agent\_idx)  
 best\_x, best\_y, \_ = candidates[np.random.randint(len(candidates))]  
  
 return best\_x, best\_y  
  
 def step(self, n\_steps: int = 1):  
 *"""Выполнить n\_steps шагов алгоритма"""* for \_ in range(n\_steps):  
 *# Перемещаем всех агентов последовательно* for agent\_idx in range(self.n\_agents):  
 if not self.agents\_finished[agent\_idx]:  
 new\_x, new\_y = self.move\_agent(agent\_idx)  
  
 *# Обновляем позицию агента* self.agents\_x[agent\_idx] = new\_x  
 self.agents\_y[agent\_idx] = new\_y  
  
 self.total\_steps += 1

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def get\_statistics(self) -> dict:  
 *"""Получить статистику о прогрессе"""* return {  
 'total\_steps': self.total\_steps,  
 'successful\_agents': self.successful\_agents,  
 'active\_agents': np.sum(~self.agents\_finished),  
 'success\_rate': self.successful\_agents / self.n\_agents \* 100  
 }  
  
 def save\_checkpoint(self, iteration: int):  
 *"""Сохранить контрольную точку"""* stats = self.get\_statistics()  
  
 *# Вычисляем метрики альфа для активных агентов* active\_mask = ~self.agents\_finished  
 if np.any(active\_mask):  
 active\_x = self.agents\_x[active\_mask]  
 active\_y = self.agents\_y[active\_mask]  
 active\_alphas = self.alpha[active\_y, active\_x]  
 max\_alpha = np.max(active\_alphas)  
 avg\_alpha = np.mean(active\_alphas)  
 else:  
 max\_alpha = 0  
 avg\_alpha = 0  
  
 self.history.append({  
 'iteration': iteration,  
 'agents\_x': self.agents\_x.copy(),  
 'agents\_y': self.agents\_y.copy(),  
 'finished': self.agents\_finished.copy(),  
 'max\_alpha': max\_alpha,  
 'avg\_alpha': avg\_alpha,  
 \*\*stats  
 })  
  
 print(f"Checkpoint - Итерация {iteration:,}: "  
 f"max\_α={max\_alpha:.2f}, avg\_α={avg\_alpha:.2f}")  
  
 def visualize\_progress(self, save\_path: str = 'maze\_progress.png'):  
 *"""Визуализация прогресса навигации"""* if not self.history:  
 print("История пуста, нечего визуализировать")  
 return  
  
 n\_checkpoints = len(self.history)  
  
 *# Создаем фигуру с 7 subplot'ами: 6 для лабиринта + 1 для графика альфа* fig = plt.figure(figsize=(20, 12))  
 gs = fig.add\_gridspec(3, 3, hspace=0.3, wspace=0.3)  
  
 *# Первые 6 ячеек для визуализации лабиринта* axes\_maze = [fig.add\_subplot(gs[i // 3, i % 3]) for i in range(6)]  
  
 *# Последняя ячейка (внизу справа) для графика альфа* ax\_alpha = fig.add\_subplot(gs[2, :])

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Выбираем ключевые контрольные точки* indices = [0, n\_checkpoints // 5, 2 \* n\_checkpoints // 5,  
 3 \* n\_checkpoints // 5, 4 \* n\_checkpoints // 5, n\_checkpoints - 1]  
  
 for idx, ax in zip(indices, axes\_maze):  
 if idx >= n\_checkpoints:  
 idx = n\_checkpoints - 1  
  
 checkpoint = self.history[idx]  
  
 *# Создаем изображение для визуализации* viz\_img = np.ones((self.height, self.width, 3), dtype=np.float32)  
  
 *# Рисуем лабиринт (стены черные)* viz\_img[self.maze == 0] = [0, 0, 0]  
  
 *# Рисуем градиент альфа* for y in range(self.height):  
 for x in range(self.width):  
 if self.maze[y, x] == 1:  
 alpha\_val = self.alpha\_normalized[y, x]  
 viz\_img[y, x] = [1 - alpha\_val \* 0.3, 1 - alpha\_val \* 0.3, 1]  
  
 *# Рисуем вход и выход (оба зеленые)* viz\_img[self.entry[1], self.entry[0]] = [0, 1, 0] *# Зеленый вход* viz\_img[self.exit[1], self.exit[0]] = [0, 1, 0] *# Зеленый выход  
  
 # Рисуем агентов с градиентом цвета* agents\_x = checkpoint['agents\_x']  
 agents\_y = checkpoint['agents\_y']  
 finished = checkpoint['finished']  
  
 *# Подсчитываем активных агентов для градиента* n\_active = np.sum(~finished)  
  
 for x, y, is\_finished in zip(agents\_x, agents\_y, finished):  
 if 0 <= x < self.width and 0 <= y < self.height:  
 if is\_finished:  
 viz\_img[y, x] = [0.3, 0.3, 0.3] *# Темно-серые (завершили)* else:  
 *# Градиент от темно-красного (мало агентов) до ярко-красного (много агентов)* intensity = 0.5 + 0.5 \* (n\_active / self.n\_agents)  
 viz\_img[y, x] = [intensity, 0, 0]  
  
 ax.imshow(viz\_img, interpolation='nearest')  
 ax.set\_title(f"Итерация {checkpoint['iteration']:,}\n"  
 f"Успешных: {checkpoint['successful\_agents']}, "  
 f"Активных: {checkpoint['active\_agents']}",  
 fontsize=10, fontweight='bold')  
 ax.axis('off')

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# График прогресса альфа-метрики* iterations = [h['iteration'] for h in self.history]  
 max\_alphas = [h.get('max\_alpha', 0) for h in self.history]  
 avg\_alphas = [h.get('avg\_alpha', 0) for h in self.history]  
  
 ax\_alpha.plot(iterations, max\_alphas, 'b-', linewidth=3, marker='o',  
 markersize=6, label='Максимальная α', alpha=0.8)  
 ax\_alpha.plot(iterations, avg\_alphas, 'g-', linewidth=3, marker='s',  
 markersize=6, label='Средняя α', alpha=0.8)  
 ax\_alpha.fill\_between(iterations, avg\_alphas, max\_alphas, alpha=0.2, color='cyan')  
  
 ax\_alpha.set\_xlabel('Итерация', fontsize=14, fontweight='bold')  
 ax\_alpha.set\_ylabel('Метрика α', fontsize=14, fontweight='bold')  
 ax\_alpha.set\_title('Прогресс метрики α активных агентов', fontsize=14, fontweight='bold')  
 ax\_alpha.grid(True, alpha=0.3, linestyle='--')  
 ax\_alpha.legend(fontsize=12, loc='lower right')  
  
 plt.savefig(save\_path, dpi=150, bbox\_inches='tight')  
 print(f"Визуализация сохранена: {save\_path}")  
 plt.close()  
  
 def visualize\_alpha\_metric(self, save\_path: str = 'alpha\_metric.png'):  
 *"""Визуализация метрики альфа"""* fig, axes = plt.subplots(1, 2, figsize=(16, 7))  
  
 *# Левый график - метрика альфа* ax = axes[0]  
 alpha\_display = np.where(self.maze == 1, self.alpha\_normalized, np.nan)  
 im = ax.imshow(alpha\_display, cmap='RdYlGn', interpolation='nearest')  
 ax.plot(self.entry[0], self.entry[1], 'bo', markersize=15, label='Вход', markeredgecolor='white',  
 markeredgewidth=2)  
 ax.plot(self.exit[0], self.exit[1], 'g\*', markersize=20, label='Выход', markeredgecolor='white',  
 markeredgewidth=2)  
 ax.set\_title('Метрика альфа (градиент к выходу)', fontsize=14, fontweight='bold')  
 ax.legend(fontsize=12)  
 ax.axis('off')  
 plt.colorbar(im, ax=ax, label='Альфа (нормализованная)')  
  
 *# Правый график - лабиринт* ax = axes[1]  
 ax.imshow(self.maze, cmap='gray', interpolation='nearest')  
 ax.plot(self.entry[0], self.entry[1], 'bo', markersize=15, label='Вход', markeredgecolor='white',  
 markeredgewidth=2)  
 ax.plot(self.exit[0], self.exit[1], 'g\*', markersize=20, label='Выход', markeredgecolor='white',  
 markeredgewidth=2)  
 ax.set\_title('Лабиринт', fontsize=14, fontweight='bold')  
 ax.legend(fontsize=12)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

ax.axis('off')  
  
 plt.tight\_layout()  
 plt.savefig(save\_path, dpi=150, bbox\_inches='tight')  
 print(f"Визуализация метрики альфа сохранена: {save\_path}")  
 plt.close()  
  
  
*# === ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ ===  
  
# Глобальные переменные для shared memory*\_shared\_maze = None  
\_shared\_alpha = None  
\_shared\_agents\_x = None  
\_shared\_agents\_y = None  
\_shared\_agents\_finished = None  
\_maze\_shape = None  
\_n\_agents = None  
\_directions = None  
\_random\_orders = None  
\_exit\_point = None  
  
  
def init\_worker\_maze(maze\_array, alpha\_array, agents\_x\_array, agents\_y\_array,  
 agents\_finished\_array, maze\_shape, n\_agents, directions,  
 random\_orders, exit\_point):  
 *"""Инициализация worker процесса"""* global \_shared\_maze, \_shared\_alpha, \_shared\_agents\_x, \_shared\_agents\_y  
 global \_shared\_agents\_finished, \_maze\_shape, \_n\_agents, \_directions  
 global \_random\_orders, \_exit\_point  
  
 \_shared\_maze = maze\_array  
 \_shared\_alpha = alpha\_array  
 \_shared\_agents\_x = agents\_x\_array  
 \_shared\_agents\_y = agents\_y\_array  
 \_shared\_agents\_finished = agents\_finished\_array  
 \_maze\_shape = maze\_shape  
 \_n\_agents = n\_agents  
 \_directions = directions  
 \_random\_orders = random\_orders  
 \_exit\_point = exit\_point  
  
  
def process\_agent\_batch\_maze(args):  
 *"""Обработка батча агентов"""* start\_idx, end\_idx, step\_num, exit\_point = args  
  
 *# Получаем numpy массивы из shared memory* maze = np.frombuffer(\_shared\_maze, dtype=np.int32).reshape(\_maze\_shape)  
 alpha = np.frombuffer(\_shared\_alpha, dtype=np.float64).reshape(\_maze\_shape)  
 agents\_x = np.frombuffer(\_shared\_agents\_x, dtype=np.int32)  
 agents\_y = np.frombuffer(\_shared\_agents\_y, dtype=np.int32)  
 agents\_finished = np.frombuffer(\_shared\_agents\_finished, dtype=ctypes.c\_bool)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

height, width = \_maze\_shape  
 new\_positions = []  
 newly\_finished = []  
  
 for i in range(start\_idx, end\_idx):  
 if agents\_finished[i]:  
 new\_positions.append((agents\_x[i], agents\_y[i]))  
 newly\_finished.append(False)  
 continue  
  
 x, y = agents\_x[i], agents\_y[i]  
  
 *# Проверка достижения выхода* exit\_x, exit\_y = exit\_point  
 if abs(x - exit\_x) <= 2 and abs(y - exit\_y) <= 2:  
 new\_positions.append((x, y))  
 newly\_finished.append(True)  
 continue  
  
 *# Собираем все возможные направления с их альфами* possible\_moves = []  
  
 for dir\_idx in range(8):  
 dx, dy = \_directions[dir\_idx]  
 new\_x, new\_y = x + dx, y + dy  
  
 if (0 <= new\_x < width and 0 <= new\_y < height and  
 maze[new\_y, new\_x] == 1):  
 alpha\_val = alpha[new\_y, new\_x]  
 possible\_moves.append((new\_x, new\_y, alpha\_val))  
  
 if not possible\_moves:  
 *# Застряли - остаемся на месте* new\_positions.append((x, y))  
 newly\_finished.append(False)  
 continue  
  
 *# Сортируем по альфе (по убыванию)* possible\_moves.sort(key=lambda m: m[2], reverse=True)  
  
 *# Выбираем из топ-K лучших направлений* top\_k = min(3, len(possible\_moves))  
 candidates = possible\_moves[:top\_k]  
  
 *# Случайный выбор с уникальным seed* np.random.seed(step\_num \* 1000 + i)  
 best\_x, best\_y, \_ = candidates[np.random.randint(len(candidates))]  
  
 new\_positions.append((best\_x, best\_y))  
 newly\_finished.append(False)  
  
 return new\_positions, newly\_finished

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

class ParallelMazeNavigator(MazeNavigationAgent):  
 *"""Параллельная версия навигатора по лабиринту"""* def \_\_init\_\_(self, maze\_path: str, n\_agents: int = 100,  
 n\_processes: int = None, seed: int = 42):  
 super().\_\_init\_\_(maze\_path, n\_agents, seed)  
  
 if n\_processes is None:  
 n\_processes = cpu\_count()  
 self.n\_processes = n\_processes  
  
 *# Создаем shared memory* self.setup\_shared\_memory()  
  
 *# КЛЮЧЕВОЕ ИСПРАВЛЕНИЕ: создаем Pool один раз!* self.pool = None  
 if self.n\_processes >= 1:  
 self.pool = Pool(  
 processes=self.n\_processes,  
 initializer=init\_worker\_maze,  
 initargs=(  
 self.shared\_maze, self.shared\_alpha,  
 self.shared\_agents\_x, self.shared\_agents\_y,  
 self.shared\_agents\_finished,  
 self.maze.shape, self.n\_agents,  
 self.directions, self.random\_orders,  
 self.exit  
 )  
 )  
 *# Регистрируем закрытие пула при завершении программы* atexit.register(self.close\_pool)  
  
 def close\_pool(self):  
 *"""Закрытие процессного пула"""* if self.pool is not None:  
 self.pool.close()  
 self.pool.join()  
 self.pool = None  
  
 def \_\_del\_\_(self):  
 *"""Деструктор - закрываем пул"""* self.close\_pool()  
  
 def setup\_shared\_memory(self):  
 *"""Настройка shared memory для параллельной обработки"""  
 # Shared memory для лабиринта и альфа (read-only для workers)* self.shared\_maze = RawArray(ctypes.c\_int32, self.maze.flatten())  
 self.shared\_alpha = RawArray(ctypes.c\_double, self.alpha.flatten())  
  
 *# Shared memory для агентов (read-write)* self.shared\_agents\_x = RawArray(ctypes.c\_int32, self.agents\_x)  
 self.shared\_agents\_y = RawArray(ctypes.c\_int32, self.agents\_y)  
 self.shared\_agents\_finished = RawArray(ctypes.c\_bool, self.agents\_finished)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def step\_parallel(self, n\_steps: int = 1):  
 *"""Параллельное выполнение шагов"""  
 # if self.n\_processes <= 1 or self.pool is None:  
 # # Если 1 процесс - используем обычный метод  
 # self.step(n\_steps)  
 # return* for step in range(n\_steps):  
 *# Разбиваем агентов на батчи* batch\_size = max(1, self.n\_agents // self.n\_processes)  
 batches = []  
  
 for i in range(self.n\_processes):  
 start = i \* batch\_size  
 end = min((i + 1) \* batch\_size, self.n\_agents)  
 if start < end:  
 batches.append((start, end, self.total\_steps, self.exit))  
  
 *# Параллельная обработка - используем существующий pool!* results = self.pool.map(process\_agent\_batch\_maze, batches)  
  
 *# Обновляем позиции - используем view вместо копирования* agents\_x\_np = np.frombuffer(self.shared\_agents\_x, dtype=np.int32)  
 agents\_y\_np = np.frombuffer(self.shared\_agents\_y, dtype=np.int32)  
 agents\_finished\_np = np.frombuffer(self.shared\_agents\_finished, dtype=ctypes.c\_bool)  
  
 batch\_idx = 0  
 for start, end, \_, \_ in batches:  
 positions, finished\_flags = results[batch\_idx]  
 for i, (new\_x, new\_y) in enumerate(positions):  
 agent\_idx = start + i  
 agents\_x\_np[agent\_idx] = new\_x  
 agents\_y\_np[agent\_idx] = new\_y  
  
 *# Обновляем флаг завершения* if finished\_flags[i] and not agents\_finished\_np[agent\_idx]:  
 agents\_finished\_np[agent\_idx] = True  
 self.successful\_agents += 1  
  
 batch\_idx += 1  
  
 self.total\_steps += 1  
  
 *# Обновляем только если нужно сохранить checkpoint  
 # Убираем постоянное копирование для производительности*def plot\_maze\_scalability(results: dict, save\_path: str = 'experiment\_scalability.png'):  
 *"""График масштабируемости"""* n\_processes = sorted(results.keys())  
 times = [results[n] for n in n\_processes]

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))  
  
 *# График 1: Время выполнения* ax1.plot(n\_processes, times, 'b-', linewidth=3, marker='o', markersize=10, label='Время выполнения')  
  
 *# Идеальная масштабируемость* t1 = times[0]  
 ideal\_times = [t1 / n for n in n\_processes]  
 ax1.plot(n\_processes, ideal\_times, 'r--', linewidth=2, alpha=0.5, label='Идеальная масштабируемость')  
  
 ax1.set\_xlabel('Количество процессов', fontsize=14, fontweight='bold')  
 ax1.set\_ylabel('Время выполнения (секунды)', fontsize=14, fontweight='bold')  
 ax1.set\_title('Время выполнения vs Количество процессов', fontsize=14, fontweight='bold')  
 ax1.grid(True, alpha=0.3)  
 ax1.legend(fontsize=12)  
  
 *# График 2: Ускорение и эффективность* speedups = [t1 / t for t in times]  
 efficiencies = [(s / n) \* 100 for s, n in zip(speedups, n\_processes)]  
  
 ax2\_twin = ax2.twinx()  
  
 line1 = ax2.plot(n\_processes, speedups, 'g-', linewidth=3, marker='o', markersize=10, label='Ускорение')  
 line2 = ax2.plot(n\_processes, n\_processes, 'r--', linewidth=2, alpha=0.5, label='Линейное ускорение')  
 line3 = ax2\_twin.plot(n\_processes, efficiencies, 'purple', linewidth=3, marker='s', markersize=8,  
 label='Эффективность')  
  
 ax2.set\_xlabel('Количество процессов', fontsize=14, fontweight='bold')  
 ax2.set\_ylabel('Ускорение', fontsize=14, fontweight='bold', color='g')  
 ax2\_twin.set\_ylabel('Эффективность (%)', fontsize=14, fontweight='bold', color='purple')  
 ax2.set\_title('Ускорение и Эффективность', fontsize=14, fontweight='bold')  
 ax2.grid(True, alpha=0.3)  
 ax2.tick\_params(axis='y', labelcolor='g')  
 ax2\_twin.tick\_params(axis='y', labelcolor='purple')  
  
 *# Объединяем легенды* lines = line1 + line2 + line3  
 labels = [l.get\_label() for l in lines]  
 ax2.legend(lines, labels, fontsize=10)  
  
 plt.tight\_layout()  
 plt.savefig(save\_path, dpi=150, bbox\_inches='tight')  
 print(f"\nГрафик масштабируемости сохранен: {save\_path}")  
 plt.close()

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Выводим таблицу* speedups = [t1 / t for t in times]  
 print("\n" + "=" \* 70)  
 print("РЕЗУЛЬТАТЫ МАСШТАБИРУЕМОСТИ")  
 print("=" \* 70)  
 print(f"{'Процессы':<12} {'Время (с)':<12} {'Ускорение':<12} {'Эффективность':<15}")  
 print("-" \* 70)  
 for n, t, s in zip(n\_processes, times, speedups):  
 efficiency = (s / n) \* 100  
 print(f"{n:<12} {t:>10.2f} {s:>10.2f}x {efficiency:>12.1f}%")  
 print("=" \* 70)  
  
  
def experiment\_scalability(maze\_path: str, n\_agents: int = 100, max\_processes: int = 24):  
 *"""  
 Эксперимент 2: Масштабируемость от 1 до 24 процессов  
 Задача - первый агент достигает выхода  
 """* print("\n" + "=" \* 70)  
 print("ЭКСПЕРИМЕНТ 2: Масштабируемость (1-24 процесса)")  
 print("Критерий остановки: первый агент достигает выхода")  
 print("=" \* 70)  
  
 results = {}  
 iterations\_per\_run = {}  
  
 for n\_proc in range(1, max\_processes + 1):  
 print(f"\n{'─' \* 60}")  
 print(f"Тест с {n\_proc} процесс{'ом' if n\_proc == 1 else 'ами'}")  
 print(f"{'─' \* 60}")  
  
 algo = ParallelMazeNavigator(maze\_path, n\_agents=n\_agents,  
 n\_processes=n\_proc, seed=42)  
  
 iteration = 0  
 start\_time = time.time()  
 last\_print = 0  
  
 while algo.successful\_agents < 1:  
 algo.step\_parallel(1)  
 iteration += 1  
  
 *# Печатаем прогресс каждые 100 итераций* if iteration - last\_print >= 100:  
 *# Синхронизируем массивы для отображения статистики* algo.agents\_x = np.array(np.frombuffer(algo.shared\_agents\_x, dtype=np.int32))  
 algo.agents\_y = np.array(np.frombuffer(algo.shared\_agents\_y, dtype=np.int32))  
 algo.agents\_finished = np.array(np.frombuffer(algo.shared\_agents\_finished, dtype=ctypes.c\_bool))

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

active\_mask = ~algo.agents\_finished  
 if np.any(active\_mask):  
 active\_alphas = algo.alpha[algo.agents\_y[active\_mask], algo.agents\_x[active\_mask]]  
 max\_alpha = np.max(active\_alphas)  
 avg\_alpha = np.mean(active\_alphas)  
 print(f" Итерация {iteration}: max\_α={max\_alpha:.2f}, avg\_α={avg\_alpha:.2f}")  
 else:  
 print(f" Итерация {iteration}: все агенты завершили")  
 last\_print = iteration  
  
 *# Защита от бесконечного цикла* if iteration > 100000:  
 print(f"Достигнут лимит итераций: {iteration}")  
 break  
  
 elapsed = time.time() - start\_time  
  
 results[n\_proc] = elapsed  
 iterations\_per\_run[n\_proc] = iteration  
  
 stats = algo.get\_statistics()  
 print(f"✓ Время до первого успеха: {elapsed:.2f}с")  
 print(f"✓ Итераций: {iteration}")  
 print(f"✓ Успешных агентов на момент остановки: {stats['successful\_agents']}")  
 print(f"✓ Скорость: {iteration / elapsed:.1f} итераций/с")  
  
 *# Закрываем пул* algo.close\_pool()  
  
 return results, iterations\_per\_run  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 import sys  
 import os  
  
 maze\_path = "maze.png"  
  
 *# Проверка существования файла* if not os.path.exists(maze\_path):  
 print(f"Файл {maze\_path} не найден!")  
 sys.exit(1)  
  
 print("\n" + "=" \* 70)  
 print("НАВИГАЦИЯ АГЕНТОВ ПО ЛАБИРИНТУ (ИСПРАВЛЕННАЯ ВЕРСИЯ)")  
 print("=" \* 70)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Загрузка лабиринта и визуализация метрики альфа* print("\nЗагрузка лабиринта и вычисление метрики альфа...")  
 algo\_init = MazeNavigationAgent(maze\_path, n\_agents=100, seed=42)  
 algo\_init.visualize\_alpha\_metric('outputs/alpha\_metric.png')  
  
 *# ЭКСПЕРИМЕНТ 1: Масштабируемость* max\_proc = min(24, cpu\_count())  
 print(f"\nМаксимальное количество процессов для тестирования: {max\_proc}")  
  
 scalability\_results, iterations\_results = experiment\_scalability(  
 maze\_path, n\_agents=100, max\_processes=max\_proc  
 )  
  
 *# Построение графика масштабируемости* plot\_maze\_scalability(scalability\_results, save\_path='outputs/experiment\_scalability.png')  
  
 *# Итоговая сводка* print("\n" + "=" \* 70)  
 print("ИТОГОВАЯ СВОДКА")  
 print("=" \* 70)  
  
 t1 = scalability\_results[1]  
 t\_max = scalability\_results[max\_proc]  
 speedup = t1 / t\_max  
 efficiency = (speedup / max\_proc) \* 100  
  
 print(f"\nЭксперимент 1. Масштабируемость:")  
 print(f" Время на 1 процессе: {t1:.2f}с")  
 print(f" Время на {max\_proc} процессах: {t\_max:.2f}с")  
 print(f" Ускорение: {speedup:.2f}x")  
 print(f" Эффективность: {efficiency:.1f}%")  
  
 *# Находим оптимальное количество процессов* best\_speedup = max(t1 / scalability\_results[n] for n in scalability\_results.keys())  
 best\_n\_proc = [n for n in scalability\_results.keys()  
 if abs((t1 / scalability\_results[n]) - best\_speedup) < 0.01][0]  
 print(f" Оптимальное число процессов: {best\_n\_proc} (ускорение {best\_speedup:.2f}x)")  
  
 print("\n" + "=" \* 70)  
 print("ВСЕ ЭКСПЕРИМЕНТЫ ЗАВЕРШЕНЫ!")  
 print("=" \* 70)  
 print("\nСохраненные файлы:")  
 print(" - alpha\_metric.png - визуализация метрики альфа")  
 print(" - experiment\_scalability.png - график масштабируемости")  
 print("=" \* 70)