

Разработка интерактивного приложения «Пятнашки» с решателем A* и расширенным пользовательским интерфейсом

Акентьев Данила Денисович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлен процесс создания интерактивного приложения для классической головоломки «Пятнашки» (15 Puzzle) с интегрированным алгоритмом поиска A* для автоматического решения. Особое внимание уделено реализации удобного графического интерфейса, обеспечивающего ручное управление, визуализацию решения, а также расширенные возможности — выбор размера поля, генерация уровней сложности, подсчёт ходов, таймер, подсказки и сохранение/загрузку игры. В работе описываются методы оптимизации производительности и удобства пользователя. Результаты показывают, что предложенный подход повышает вовлечённость игроков и способствует лучшему пониманию алгоритмов поиска на практике.

Ключевые слова: Пятнашки, 15 Puzzle, алгоритм A*, поиск пути, интерактивное приложение, графический интерфейс, визуализация, головоломка, Python, Tkinter

Development of an interactive Tag application with the A* solver and an extended user interface

Akentev Danila Denisovich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents the process of creating an interactive application for the classic "Tag" puzzle (15 Puzzle) with an integrated A* search algorithm for automatic solutions. Special attention is paid to the implementation of a user—friendly graphical interface that provides manual control, visualization of solutions, as well as advanced features such as field size selection, difficulty level generation, counting moves, timer, hints, and saving/loading the game. The paper describes methods for optimizing performance and user convenience. The results show that the proposed approach increases the involvement of players and contributes to a better understanding of search algorithms in practice.

Keywords: Tag games, 15 Puzzle, Algorithm A*, pathfinding, interactive application, Graphical interface, Visualization, Puzzle, Python, Tkinter

1 Введение

1.1 Актуальность

Головоломка «Пятнашки» является классическим примером задачи перестановок и широко используется для изучения алгоритмов поиска и эвристик в информатике. Несмотря на доступность множества решений, существует необходимость создания приложений с интуитивно понятным интерфейсом и расширенным функционалом для повышения образовательной и развлекательной ценности. Интеграция алгоритма A^* с визуализацией решений позволяет не только ускорить процесс нахождения решения, но и демонстрирует работу алгоритмов поиска в реальном времени.

1.2 Обзор исследований

В работе Н.Н.Х. Куана, В.Н. Нечаева и В.Б. Малыгина рассмотрено применение алгоритма A^* для оптимизации маршрутов обслуживания воздушного движения в районе диспетчерского центра Хошимина. Авторы разработали математическую модель, в которой целевая функция минимизирует общую длину маршрутов с учётом ограничений воздушного пространства. Результаты показали, что использование алгоритма A^* способствует снижению перегрузки, увеличению пропускной способности и повышению эффективности маршрутизации воздушных судов, сохраняя необходимый уровень безопасности [1].

Статья Н.Н. Хтун, Л.В. Маркина и А.А. Соседко посвящена применению рецепторных геометрических моделей в задачах автоматизированной компоновки авиационной техники. Авторы используют алгоритм A^* в процессе телесной трассировки — построения соединительных трасс между объектами в ограниченном пространстве с заданной степенью гладкости. За счёт модификации эвристической функции достигается сокращение числа лишних обходов, что делает подход эффективным при проектировании компоновки объектов сопоставимых размеров [2].

В исследовании Т.С. Домрачевой, В.В. Гусева, Н.А. Романовой и Р.П. Христофорова представлены алгоритмы поиска пути в дискретном пространстве. Особое внимание уделено алгоритму A^* , его структуре, принципу работы и применимости в различных задачах. Авторы проводят сравнительный анализ с другими методами и демонстрируют, что A^* является универсальным инструментом благодаря настраиваемой эвристике и устойчивой эффективности в решении навигационных задач [3].

В статье И.Д. Хирьянова рассмотрена реализация алгоритма A^* в навигационных сетках с использованием шаблона проектирования ECS для игрового движка Unity. Автор описывает основные этапы разработки, включая структурирование данных и оптимизацию процессов. Сравнительный анализ показывает, что ECS-реализация обеспечивает более высокую производительность и скорость поиска по сравнению со стандартной, что особенно актуально для создания кроссплатформенных игровых приложений [4].

Другим важным исследованием является работа Д.Ф. Ахметгалиевой и Г.Р. Галиаскаровой, в которой рассматривается создание игры "Змейка" на Python с использованием библиотеки tkinter. В статье акцентируется внимание на необычном подходе к разработке игры, в котором традиционную змейку заменяет персонаж Гомер Симпсон, поедающий пончики. Игра включает элементы геймплея, где с каждым съеденным пончиком персонаж удлиняется, а скорость игры увеличивается. Такое решение подчеркивает возможности tkinter для создания простых, но увлекательных игр с графическим интерфейсом [5].

1.3 Цель исследования

Целью исследования является разработка и реализация интерактивного приложения для игры «Пятнашки» с решателем A*, оснащённое расширенным графическим интерфейсом и функционалом, обеспечивающим удобство пользователя и демонстрацию работы алгоритма поиска.

2 Материалы и методы

В качестве инструментария использовался язык программирования Python и библиотека Tkinter для создания графического интерфейса. Алгоритм поиска пути A* реализован с использованием эвристики Манхэттена для эффективного решения головоломки. Для повышения удобства пользователя реализованы функции выбора размера поля, генерации уровней сложности, подсчёта ходов, таймера, подсказок на основе решения и сохранения/загрузки состояния игры. Многоступенчатое тестирование и оптимизация обеспечили корректную работу приложения при различных условиях.

3 Результаты и обсуждения

1. Инициализация и базовые параметры игры (рис.1).

```
import tkinter as tk
from tkinter import messagebox, filedialog
import random
import time
import threading
import heapq
import json

class FifteenPuzzleApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Пятнашки [A* и расширенными возможностями]")

        # Параметры игры
        self.size = 4 # размер поля (3-5)
        self.tiles = [] # плитки на Canvas
        self.tile_size = 80
        self.margin = 20
        self.board_state = []
        self.empty_pos = None
        self.moves_count = 0
        self.timer_running = False
        self.start_time = None
        self.timer_id = None
        self.solution = None # хранит решение
        self.animating = False

        # UI
        self.create_widgets()
        self.reset_game()
```

Рисунок 1. Код

Задаются основные параметры игры — размер поля, размеры плиток и отступы. Инициализируется состояние доски и вспомогательные переменные для контроля ходов, таймера, решения и анимации. Запускается построение интерфейса и стартовое сбрасывание игры.

2. Создание графического интерфейса (рис.2).

```

def create_widgets(self):
    control_frame = tk.Frame(self.root)
    control_frame.pack(pady=10)

    tk.Label(control_frame, text="Размер поля:").grid(row=0, column=0)
    self.size_var = tk.IntVar(value=self.size)
    size_menu = tk.OptionMenu(control_frame, self.size_var, 3, 4, 5, command=self.change_size)
    size_menu.grid(row=0, column=1)

    tk.Label(control_frame, text="Сложность (кол-во перемешиваний):").grid(row=1, column=0)
    self.mix_var = tk.IntVar(value=20)
    mix_entry = tk.Entry(control_frame, textvariable=self.mix_var, width=5)
    mix_entry.grid(row=1, column=1)

    self.moves_label = tk.Label(control_frame, text="Ходов: 0")
    self.moves_label.grid(row=0, column=2, padx=15)
    self.timer_label = tk.Label(control_frame, text="Время: 00:00")
    self.timer_label.grid(row=1, column=2, padx=15)

    btn_frame = tk.Frame(self.root)
    btn_frame.pack(pady=10)

    self.shuffle_btn = tk.Button(btn_frame, text="Перемешать", command=self.shuffle_board)
    self.shuffle_btn.grid(row=0, column=0, padx=5)
    self.solve_btn = tk.Button(btn_frame, text="Решить", command=self.solve_puzzle)
    self.solve_btn.grid(row=0, column=1, padx=5)
    self.hint_btn = tk.Button(btn_frame, text="Подсказка", command=self.show_hint)
    self.hint_btn.grid(row=0, column=2, padx=5)
    self.save_btn = tk.Button(btn_frame, text="Сохранить игру", command=self.save_game)
    self.save_btn.grid(row=0, column=3, padx=5)
    self.load_btn = tk.Button(btn_frame, text="Загрузить игру", command=self.load_game)
    self.load_btn.grid(row=0, column=4, padx=5)

    self.canvas = tk.Canvas(self.root,
                             width=self.size * self.tile_size + 2 * self.margin,
                             height=self.size * self.tile_size + 2 * self.margin,
                             bg='white')
    self.canvas.pack(pady=10)
    self.canvas.bind("<Button-1", self.click_tile)

```

Рисунок 2. Код

Создаются элементы управления: выбор размера поля и сложности, отображение текущих ходов и времени, кнопки для перемешивания, решения, подсказки, сохранения и загрузки. Основная игровая область — холст (canvas), на котором отображается доска, и к которому привязывается обработка кликов мыши.

3. Обработка изменения размера поля и сброс игры (рис.3).

```

def change_size(self, val):
    if self.animating:
        messagebox.showinfo("Подождите", "Пожалуйста, дождитесь окончания анимации.")
        self.size_var.set(self.size)
        return
    self.size = int(val)
    self.reset_game()

def reset_game(self):
    self.board_state = list(range(1, self.size*self.size))
    self.board_state.append(0) # пустая клетка
    self.empty_pos = self.board_state.index(0)
    self.moves_count = 0
    self.update_moves_label()
    self.stop_timer()
    self.update_timer_label(0)
    self.solution = None
    self.animating = False
    self.resize_canvas()
    self.draw_board()

```

Рисунок 3. Код

Обрабатывается смена размера поля: если нет текущей анимации, игра сбрасывается — доска возвращается в исходное состояние с упорядоченными

плитками и пустой клеткой, счётчики и таймеры обнуляются, интерфейс обновляется.

4. Подгонка холста и отрисовка игрового поля (рис.4).

```
def resize_canvas(self):
    self.canvas.config(width=self.size * self.tile_size + 2 * self.margin,
                       height=self.size * self.tile_size + 2 * self.margin)

def draw_board(self):
    self.canvas.delete("all")
    self.tiles = []
    for i in range(self.size*self.size):
        r, c = divmod(i, self.size)
        x1 = self.margin + c * self.tile_size
        y1 = self.margin + r * self.tile_size
        x2 = x1 + self.tile_size
        y2 = y1 + self.tile_size
        tile_val = self.board_state[i]
        if tile_val == 0:
            # пустой квадрат
            self.canvas.create_rectangle(x1, y1, x2, y2, fill="lightgray", outline="black")
            self.tiles.append(None)
        else:
            rect = self.canvas.create_rectangle(x1, y1, x2, y2, fill="white", outline="black")
            text = self.canvas.create_text((x1 + x2)//2, (y1 + y2)//2,
                                          text=str(tile_val), font=("Arial", 24, "bold"))
            self.tiles.append((rect, text))
```

Рисунок 4. Код

Автоматически подстраиваем размер холста под текущий размер поля с учётом отступов. Затем полностью перерисовываем поле: пустую ячейку выделяем серым, остальные — белыми с крупными номерами, выстраивая аккуратную сетку.

5. Перемешивание плиток и запуск таймера (рис.5).

```
def shuffle_board(self):
    if self.animating:
        return
    moves = self.mix_var.get()
    if moves < 5 or moves > 200:
        messagebox.showerror("Ошибка", "Количество перемешиваний от 5 до 200")
        return
    self.reset_game()
    # Перемешиваем, делая случайные ходы
    for _ in range(moves):
        neighbors = self.get_neighbors(self.empty_pos)
        next_pos = random.choice(neighbors)
        self.swap_tiles(self.empty_pos, next_pos)
        self.empty_pos = next_pos
    if not self.is_solvable():
        # Если получилось нерешаемое, поменяем пару плиток
        if self.size > 2:
            self.board_state[0], self.board_state[1] = self.board_state[1], self.board_state[0]
    self.moves_count = 0
    self.update_moves_label()
    self.draw_board()
    self.start_timer()
    self.solution = None
```

Рисунок 5. Код

Для перемешивания игровое поле сбрасывается, после чего выполняется серия случайных ходов пустой клетки, имитируя реальное смешивание. Проверяется, разрешима ли новая позиция; при необходимости вносятся правки для гарантии решения. Счётчики и таймеры обнуляются, интерфейс обновляется, и запускается отсчёт времени.

6. Обработка кликов по плиткам и игровая логика ходов (рис.6).

```
def click_tile(self, event):
    if self.animating:
        return
    x, y = event.x, event.y
    c = (x - self.margin) // self.tile_size
    r = (y - self.margin) // self.tile_size
    if 0 <= r < self.size and 0 <= c < self.size:
        pos = r * self.size + c
        if pos in self.get_neighbors(self.empty_pos):
            self.swap_tiles(pos, self.empty_pos)
            self.empty_pos = pos
            self.moves_count += 1
            self.update_moves_label()
            self.draw_board()
        if self.is_solved():
            self.stop_timer()
            messagebox.showinfo("Поздравляем!", f"Вы решили головоломку за {self.moves_count} ходов и {self.elapsed_time_str()}!")
```

Рисунок 6. Код

Реализована реакция на клики: если нажата плитка, соседняя с пустой, происходит обмен, обновляется счёт ходов и поле. Если после хода головоломка решена — таймер останавливается и игроку выводится поздравление.

7. Управление счётчиком ходов и таймером (рис.7).

```
def update_moves_label(self):
    self.moves_label.config(text=f"Ходов: {self.moves_count}")

def start_timer(self):
    if self.timer_running:
        return
    self.start_time = time.time()
    self.timer_running = True
    self.update_timer()

def update_timer(self):
    if not self.timer_running:
        return
    elapsed = time.time() - self.start_time
    self.update_timer_label(elapsed)
    self.timer_id = self.root.after(500, self.update_timer)

def update_timer_label(self, elapsed):
    mins = int(elapsed // 60)
    secs = int(elapsed % 60)
    self.timer_label.config(text=f"Время: {mins:02d}:{secs:02d}")

def stop_timer(self):
    if self.timer_running:
        self.timer_running = False
        if self.timer_id:
            self.root.after_cancel(self.timer_id)
            self.timer_id = None
```

Рисунок 7. Код

Отдельно оформлен контроль за ходами и временем: вывод текущего количества ходов, запуск и регулярное обновление таймера с точностью до полсекунды, а также его остановка.

8. Проверка разрешимости и победы (рис.8).

```
def is_solvable(self):
    inv_count = 0
    flat = [x for x in self.board_state if x != 0]
    for i in range(len(flat)):
        for j in range(i + 1, len(flat)):
            if flat[i] > flat[j]:
                inv_count += 1
    zero_row = self.board_state.index(0) // self.size
    row_from_bottom = self.size - zero_row
    if self.size % 2 == 1:
        return inv_count % 2 == 0
    else:
        if row_from_bottom % 2 == 0:
            return inv_count % 2 == 1
        else:
            return inv_count % 2 == 0
```

Рисунок 8. Код

Классическая логика пятнашек: считаем количество инверсий для оценки разрешимости, учитывая размер поля и позицию пустой клетки. Также реализована простая проверка — собраны ли плитки в нужном порядке.

9. Поиск решения с помощью алгоритма A^* (рис.9).

```

def solve_puzzle(self):
    if self.animating:
        return
    self.solve_btn.config(state='disabled')
    threading.Thread(target=self._solve_thread).start()

def _solve_thread(self):
    start = tuple(self.board_state)
    goal = tuple(range(1, self.size*self.size)) + (0,)
    frontier = []
    heapq.heappush(frontier, (0, start, []))
    cost_so_far = {start: 0}
    size = self.size

    def neighbors(state):
        zero_idx = state.index(0)
        r, c = divmod(zero_idx, size)
        positions = []
        if r > 0:
            positions.append(zero_idx - size)
        if r < size - 1:
            positions.append(zero_idx + size)
        if c > 0:
            positions.append(zero_idx - 1)
        if c < size - 1:
            positions.append(zero_idx + 1)
        for pos in positions:
            new_state = list(state)
            new_state[zero_idx], new_state[pos] = new_state[pos], new_state[zero_idx]
            yield tuple(new_state), pos

    def heuristic(state):
        dist = 0
        for idx, val in enumerate(state):
            if val == 0:
                continue
            goal_r, goal_c = divmod(val - 1, size)
            cur_r, cur_c = divmod(idx, size)
            dist += abs(goal_r - cur_r) + abs(goal_c - cur_c)
        return dist

    while frontier:
        priority, current, path = heapq.heappop(frontier)
        if current == goal:
            self.solution = path
            self.root.after(0, self.animate_solution)
            self.solve_btn.config(state='normal')
            return
        for next_state, move_pos in neighbors(current):
            new_cost = cost_so_far[current] + 1
            if next_state not in cost_so_far or new_cost < cost_so_far[next_state]:
                cost_so_far[next_state] = new_cost
                priority = new_cost + heuristic(next_state)
                heapq.heappush(frontier, (priority, next_state, path + [next_state]))
    messagebox.showinfo("Решение не найдено", "К сожалению, решение не было найдено.")
    self.solve_btn.config(state='normal')

```

Рисунок 9. Код

Запуск поиска решения с использованием алгоритма А*. Поиск происходит в отдельном потоке, чтобы не блокировать интерфейс. Используется манхэттенская эвристика, которая эффективно оценивает расстояния плиток до их целей. Если решение найдено, запускается анимация

10. Анимация решения (рис.10).

```

def animate_solution(self):
    if not self.solution:
        return
    self.animating = True

    def step(index=0):
        if index >= len(self.solution):
            self.animating = False
            return
        self.board_state = list(self.solution[index])
        self.empty_pos = self.board_state.index(0)
        self.draw_board()
        self.root.after(300, step, index + 1)

    step()

```

Рисунок 10. Код

Пошагово воспроизводится решение — каждые 300 мс состояние доски обновляется, показывая игроку, как можно решить головоломку. Во время анимации блокируются остальные действия.

11. Сохранение и загрузка состояния игры (рис.11).

```

def save_game(self):
    if self.animating:
        return
    file_path = filedialog.asksaveasfilename(defaultextension=".json",
                                             filetypes=[("JSON files", "*.json")])
    if not file_path:
        return
    elapsed = 0
    if self.timer_running:
        elapsed = time.time() - self.start_time
    data = {
        "size": self.size,
        "board_state": self.board_state,
        "empty_pos": self.empty_pos,
        "moves_count": self.moves_count,
        "elapsed_time": elapsed
    }
    with open(file_path, "w") as f:
        json.dump(data, f)

def load_game(self):
    if self.animating:
        return
    file_path = filedialog.askopenfilename(defaultextension=".json",
                                             filetypes=[("JSON files", "*.json")])
    if not file_path:
        return
    with open(file_path, "r") as f:
        data = json.load(f)
    self.size = data.get("size", 4)
    self.size_var.set(self.size)
    self.board_state = data.get("board_state", list(range(1, self.size*self.size)) + [0])
    self.empty_pos = data.get("empty_pos", self.board_state.index(0))
    self.moves_count = data.get("moves_count", 0)
    elapsed = data.get("elapsed_time", 0)
    self.update_moves_label()
    self.resize_canvas()
    self.draw_board()
    if elapsed > 0:
        self.start_time = time.time() - elapsed
        self.timer_running = True
        self.update_timer()
    else:
        self.stop_timer()

```

Рисунок 11. Код

Реализован функционал сохранения текущего состояния игры в JSON-файл и загрузки из него. Это включает размер доски, положение плиток, количество ходов и прошедшее время. Игрок может прервать игру и продолжить позже.

12. Запуск приложения (рис.12).

```
if __name__ == "__main__":  
    root = tk.Tk()  
    app = FifteenPuzzleApp(root)  
    root.mainloop()
```

Рисунок 12. Код

13. Внешний вид приложения (рис.13).

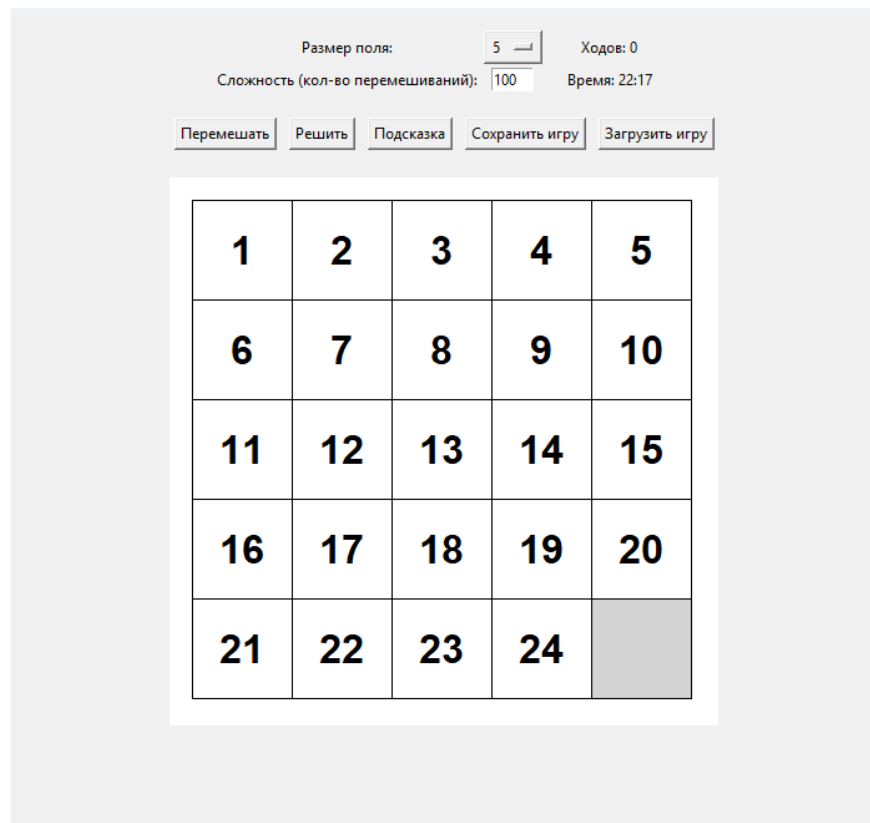


Рисунок 13. Внешний вид игры

4 Выводы

В ходе работы было разработано интерактивное приложение для классической головоломки «Пятнашки» с реализацией алгоритма поиска A* для автоматического решения. Приложение обладает расширенным графическим интерфейсом, который обеспечивает удобство ручного управления, визуализацию процесса решения, а также дополнительные

функции — выбор размера поля, настройку сложности, подсчёт ходов, таймер, подсказки и сохранение прогресса.

Реализованный алгоритм A^* с эвристикой Манхэттена продемонстрировал эффективность и надёжность при поиске оптимальных решений в различных конфигурациях. Использование многопоточности позволило избежать зависаний интерфейса во время вычислений, что значительно улучшило пользовательский опыт.

Таким образом, представленное решение сочетает в себе образовательные и развлекательные компоненты: пользователи могут как самостоятельно решать головоломку, так и наблюдать за работой алгоритма поиска. Это способствует лучшему пониманию принципов алгоритмического поиска и повышает интерес к изучению компьютерных наук.

В дальнейшем планируется расширение функционала за счёт реализации плавной анимации перемещения плиток, а также адаптация приложения под мобильные платформы для увеличения аудитории пользователей.

Библиографический список

1. Куан Н.Н.Х., Нечаев В.Н., Малыгин В.Б. Математическая модель и применение алгоритма $a\text{-star}$ для оптимизации маршрутов ОВД в воздушном пространстве районного диспетчерского центра Хошимаина // Crede Experto: транспорт, общество, образование, язык. 2025. №.1(44). С. 1-22.
2. Хтун Н.Н., Маркин Л.В., Соседко А.А. Применение рецепторных геометрических моделей в задачах автоматизированной компоновки авиационной техники // Труды МАИ. 2014. №. 72. С. 1-26.
3. Домрачева Т.С., Гусев В.В., Романова Н.А., Христофорова Р.П. Алгоритмы поиска пути в дискретном пространстве // Вестник современных исследований. 2018. №. 12.5(27). С. 103-105.
4. Хирьянов И.Д. Реализация алгоритма поиска пути в навигационных сетках A^* с использованием шаблона проектирования ecs для кроссплатформенной среды разработки игровых приложений $unity$ // Информационно-коммуникационные технологии: достижения, проблемы, инновации (ИКТ-2024): электронный сборник статей III международной научно-практической конференции, Полоцк, 29 марта 2024 г. Новополоцк, 2024. С. 25-28.
5. Ахметгалеева Д.Ф., Галиаскарова Г.Р. Создание игры-змейка на языке программирования Python «Гомер ест пончики» // Colloquium-journal. 2019. № 17-2(41). С. 24-27.