УДК 004.021

Решение задачи коммивояжёра с помощью генетического алгоритма, реализованного на Python

В.В. Картофлицкий, **Г.И. Горемыкина**

Российский экономический университет им. Г.В. Плеханова, г. Москва, Россия; v_neo_l@mail.ru

Аннотация. *В статье рассматривается применение генетических алгоритмов (ГА) для решения задачи коммивояжёра (TSP), являющейся важной задачей оптимизации с широким спектром практических применений. Реализация ГА осуществляется на языке программирования Python. Исследуется влияние ключевых этапов работы ГА, таких, как формирование начальной популяции, операторы скрещивания и мутации, а также стратегии отбора особей. Особое внимание уделяется анализу поведения алгоритма на различных этапах эволюции популяции, включая график изменения функции приспособленности. Также проводится замер производительности алгоритма с точки зрения времени выполнения. В результате работы продемонстрирована эффективность применения ГА для нахождения приближённых решений задачи коммивояжёра, предложены рекомендации по настройке параметров алгоритма для оптимизации его работы.*

Ключевые слова: генетический алгоритм, задача коммивояжёра, эвристические методы, оптимизация, эволюционные алгоритмы, Python.

Solving The Traveling Salesman Problem Using A Genetic Algorithm Implemented In Python

V.V. Kartoflitskiy, **G.I. Goremykina**

Plekhanov Russian University of Economics, Moscow, Russia; v_neo_l@mail.ru

Abstract. *The article examines the application of genetic algorithms (GA) to solving the Traveling Salesman Problem (TSP), an important optimization problem with a wide range of practical applications. The GA implementation is carried out using the Python programming language. The study explores the impact of key GA stages, such as the formation of the initial population, crossover and mutation operators, and selection strategies. Particular attention is given to analyzing the algorithm's behavior at different stages of population evolution, including a graph of*

fitness function changes. Additionally, the algorithm's performance is measured in terms of execution time. As a result, the study demonstrates the effectiveness of GA in finding approximate solutions to the TSP and provides recommendations for tuning algorithm parameters to optimize its performance.

Keywords: genetic algorithm, traveling salesman problem, heuristic methods, optimization, evolutionary algorithms, Python.

JEL: C63.

1. Введение

Одной из наиболее известных задач, решаемых с помощью ГА, является задача коммивояжёра (TSP – Traveling Salesman Problem). Она заключается в поиске кратчайшего пути, проходящего через заданное множество городов и возвращающегося в исходную точку. Данная задача имеет множество практических приложений, таких как маршрутизация транспортных средств, оптимизация логистических цепочек и планирование передвижения автономных систем.

Традиционные методы решения задачи коммивояжёра, включая жадные алгоритмы и методы динамического программирования, оказываются неэффективными при увеличении числа городов. В таких случаях эвристические и метаэвристические подходы, включая ГА, позволяют находить приближённые, но достаточно качественные решения за приемлемое время.

2. Цель исследования

Данное исследование посвящено изучению и применению генетического алгоритма для решения задачи коммивояжёра. Основное внимание уделяется ключевым этапам работы ГА: формированию начальной популяции, операторам скрещивания и мутации, а также стратегии отбора особей. Также рассматривается влияние параметров алгоритма на скорость сходимости и качество найденных решений. Показана реализация алгоритма на языке программирования Python.

Кроме того, в рамках исследования был проведён анализ поведения алгоритма на различных этапах эволюции популяции, включая построение графика изменения функции приспособленности. Одной из целей работы является определение итерации, на которой прекращаются улучшения решения, а также замер производительности алгоритма с точки зрения времени выполнения.

Цель работы – продемонстрировать эффективность генетического алгоритма в решении задачи коммивояжёра, оценить его производительность и предложить рекомендации по настройке параметров для достижения наилучших результатов.

3. Основная часть

Математическая постановка задачи коммивояжёра заключается в следующем. Есть множество городов $C = (c_1, c_2, \ldots, c_n)$, где n – количество городов. Расстояние между городами i и j обозначим через d_{ij}. Требуется найти перестановку городов $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, такую, что полный путь по следующей формуле (1):

$$D(\pi) = d_{\pi_n\ \pi_1} + \sum_{k=1}^{n-1} d_{\pi_k\ \pi_{k+1}} \quad (1)$$

является минимальным [1].

Для решения задачи применим генетический алгоритм, заключающийся в прохождении следующих этапов:

1. Генерация начальной популяции. Формируется множество возможных решений, называемых индивидами. В данном случае индивид – это последовательность посещения городов. Каждый индивид представлен в виде перестановки множества городов: $X = (x_1, x_2, \ldots, x_n, x_1)$, где $x_1 = 1$ – начальный и конечный город, а остальные x_i являются перестановкой $\{2, \ldots, n\}$. Реализация генерации начальной популяции в Python приведена на Рисунке 1.

2. Выбор функция приспособленности (fitness function). Приспособленность (или целевая функция) измеряет, насколько хорош индивид.[2] В данном случае это суммарное расстояние маршрута, рассчитанное по формуле (2):

$$F(X) = \sum_{i}^{n} d(x_i, x_{i+1}), \quad (2)$$

где $d(x_i, x_{i+1})$ – расстояние между городами x_i и x_{i+1}. Реализация этой части в Python приведена на Рисунке 2.

```python
def generate_population(num_cities, num_individuals):
    population = []
    for _ in range(num_individuals):
        individual = list(range(2, num_cities + 1))
        random.shuffle(individual)
        individual = [1] + individual + [1]
        if individual not in population:
            population.append(individual)
    return population
```

Рисунок 1. Генерация начальной популяции

Figure 1. Generation of the initial population

Примечание: Сложность алгоритма: O(P^2N).

```python
def calculate_fitness(individual, distance_matrix):
    total_distance = 0
    for i in range(len(individual) - 1):
        total_distance += distance_matrix[individual[i] - 1][individual[i + 1] - 1]
    return total_distance
```

Рисунок 2. Функция приспособленности

Figure 2. Fitness function

Примечание: Сложность алгоритма: O(N), так как вычисляется расстояние для каждого города.

3. Выбор родителей (selection). Из популяции случайным образом выбираются несколько особей для скрещивания [3]. Используется метод случайного отбора (random sampling). Реализация этой части в Python приведена на Рисунке 3.

```python
def selection(population, num_parents):
    return random.sample(population, num_parents)
```

Рисунок 3. Выбор родителей

Figure 3. Parent selection

Примечание: Сложность: O(P), где P – число выбираемых родителей.

4. Скрещивание (crossover). Создаются новые индивиды (потомки) путём комбинации генетического материала родителей. Для этого выбирают точку разреза k и комбинируют родителей A и B: $C_1 = (A_1, A_2, \dots, A_k, B'_{k+1}, \dots, B'_n, 1)$ и $C_2 = (B_1, B_2, \dots, B_k, A'_{k+1}, \dots, A'_n, 1)$ [4]. Реализация скрещивания в Python приведена на Рисунке 4.

```
def crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        parent1 = parents[i]
        parent2 = parents[i + 1]
        crossover_point = random.randint(1, len(parent1) - 2)
        child1 = parent1[:crossover_point] + [city for city in parent2 if city not in parent1
        child2 = parent2[:crossover_point] + [city for city in parent1 if city not in parent2
        offspring.extend([child1, child2])
    return offspring
```

Рисунок 4. Скрещивание

Figure 4. Crossover

Примечание: Сложность: O(P·N) = O(PN).

5. Мутация (mutation). Происходит случайный обмен двух городов с вероятностью mutation_rate. Для индивида X выбираются два случайных индекса i, j и производится обмен: $X' = (x_1, \dots, x_i, \dots, x_j, \dots, x_n, x_1)$, где $x_i \leftrightarrow x_j$ [5]. Реализация мутации в Python приведена на Рисунке 5.

```
def mutation(offspring, mutation_rate):
    for i in range(len(offspring)):
        if random.random() < mutation_rate:
            idx1, idx2 = random.sample(range(1, len(offspring[i]) - 1), 2)
            offspring[i][idx1], offspring[i][idx2] = offspring[i][idx2], offspring[i][idx1]
    return offspring
```

Рисунок 5. Мутация

Figure 5. Mutation

Примечание: Сложность: O(1) в среднем для одного индивида.

Так как мутация каждой особи – O(1), общая сложность мутации: O(P·1) = O(P)O(P·1) = O(P).

6. Замена популяции (replacement). Новые особи заменяют старую популяцию. Для этого комбинируют старых и новых индивидов и сортируют их по приспособленности [6].

Математически это описывается следующим образом. Пусть P_t – популяция в поколении t, а C_t – потомки. Тогда $P_{t+1} = Best\ (P_t \cup C_t)$. Реализация замены популяции в Python представлена на Рисунке 6.

7. Основной цикл генетического алгоритма. В данном коде было выбрано 100 поколений (это компромиссное значение, которое часто даёт достаточно хорошее решение за разумное время) [7] (Рисунок 7).

```python
def replace_population(population, offspring, distance_matrix):
    combined_population = population + offspring
    combined_population.sort(key=lambda x: calculate_fitness(x, distance_matrix))
    unique_population = [combined_population[0]]
    for ind in combined_population[1:]:
        if ind != unique_population[-1]:
            unique_population.append(ind)
    return unique_population[:len(population)]
```

Рисунок 6. Замена популяции

Figure 6. Population replacement

Примечание: Сложность*:* O(PN + PlogP).

```python
for generation in range(100):
    parents = selection(population, num_parents)
    offspring = crossover(parents)
    offspring = mutation(offspring, mutation_rate)
    population = replace_population(population, offspring, distance_matrix)
```

Рисунок 7. Основной цикл генетического алгоритма

Figure 7. Main loop of the genetic algorithm

Примечание: Сложность алгоритма: Основная сложность генетического алгоритма (за G поколений): O(G·(P^2N+PN+PlogP)), где G – число поколений; P – размер популяции.

8. Вывод лучшего маршрута. После 100 поколений алгоритм возвращает маршрут с минимальной длиной (Рисунок 8).

```python
best_route = min(population, key=lambda x: calculate_fitness(x, distance_matrix))
best_fitness = calculate_fitness(best_route, distance_matrix)

print("Best route:", best_route)
print("Best fitness:", best_fitness)
```

Рисунок 8. Вывод лучшего маршрута

Figure 8. Output of the best route

Примечание: Сложность: O(n·m) + O(m) = O(n·m), где n – это размер популяции, m – это количество городов в маршруте.

Размер популяции (num_individuals в коде) – это число маршрутов (особей) в каждом поколении. Он является гиперпараметром, который выбирается эмпирически, чтобы сбалансировать поиск хорошего решения и производительность алгоритма [8].

Алгоритм жертвует точностью ради скорости, но способен находить приближённые решения для больших графов.

4. Полученные результаты

В качестве примера для иллюстрации работы алгоритма были взяты следующие параметры (Рисунки 10 и 11).

```
# Параметры алгоритма
num_cities = 10
num_individuals = 50
num_parents = 40
mutation_rate = 0.1
generations = 100
```

Рисунок 10. Параметры алгоритма

Figure 10. Algorithm Parameters

```
# Генерация матрицы расстояний
distance_matrix = [[0, 3, 4, 2, 7, 5, 6, 8, 9, 1],
                   [3, 0, 1, 4, 2, 7, 9, 6, 5, 8],
                   [4, 1, 0, 5, 3, 6, 2, 9, 8, 7],
                   [2, 4, 5, 0, 8, 9, 3, 7, 6, 1],
                   [7, 2, 3, 8, 0, 4, 5, 1, 9, 6],
                   [5, 7, 6, 9, 4, 0, 1, 3, 2, 8],
                   [6, 9, 2, 3, 5, 1, 0, 4, 8, 7],
                   [8, 6, 9, 7, 1, 3, 4, 0, 5, 2],
                   [9, 5, 8, 6, 9, 2, 8, 5, 0, 1],
                   [1, 8, 7, 1, 6, 8, 7, 2, 1, 0]]
```

Рисунок 11. Матрица расстояний между городами

Figure 11. Distance Matrix Between Cities

После работы алгоритма получили такие результаты (Рисунок 12).

```
Лучший маршрут: [1, 4, 7, 3, 2, 5, 8, 6, 9, 10, 1]
Длина лучшего маршрута: 18
Algorithm execution time: 0.0246 seconds
```

Рисунок 12. Результаты работы алгоритма

Figure 12. Algorithm Results

График изменения фитнес-функции приведён на Рисунке 13.

Из Рисунка 13 видно, что после 22-го поколения алгоритм перестал улучшать результат.

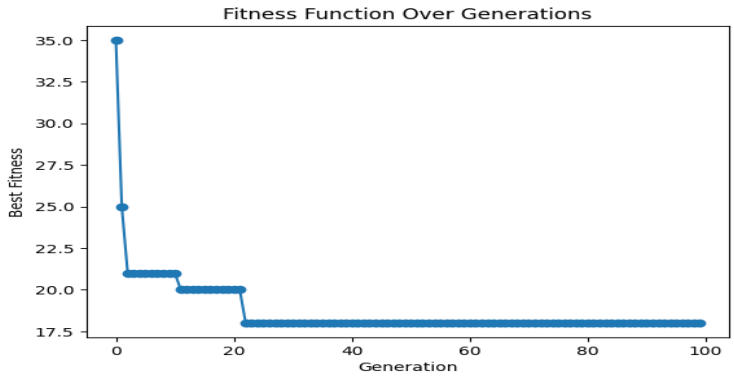

Рисунок 13. Фитнесс-функция

Figure 13. Fitness Function

По сравнению с методом полного перебора, генетический алгоритм значительно уменьшил время поиска приемлемого решения.

5. Заключение

Генетический алгоритм показал свою эффективность в решении задачи коммивояжера. Несмотря на его стохастическую природу и невозможность гарантировать нахождение глобального оптимума, он позволяет находить хорошие приближённые решения за разумное время [9]. Количество поколений и размер популяции являются ключевыми параметрами, влияющими на сходимость алгоритма. Дальнейшие улучшения могут быть достигнуты за счёт адаптивных операторов мутации и селекции, а также использования гибридных подходов, совмещающих генетический алгоритм с методами локального поиска [10].

Авторский вклад. *Концептуализация, методология, В.В. Картофлицкий; программное обеспечение, В.В. Картофлицкий; валидация, В.В. Картофлицкий и Г.И. Горемыкина; формальный анализ, В.В. Картофлицкий; эксперимент, В.В. Картофлицкий; ресурсы, Г.И. Горемыкина; обработка данных, В.В. Картофлицкий; подготовка и написание оригинального проекта, В.В. Картофлицкий; написание и редактирование, Г.И. Горемыкина; визуализация результатов, В.В. Картофлицкий; администрирование проекта, Г.И. Горемыкина. Все авторы ознакомились с опубликованной версией рукописи и согласились с ней.*

СПИСОК ИСТОЧНИКОВ

1. Вирсански Э. *Генетические алгоритмы на Python*. Москва: ДМК Пресс, 2020, 286.
2. Кувшинова Е.А. *Решение задачи коммивояжера методами ветвей и границ и ближайшего соседа на языке Python*. Тольятти: ТГУ, 2019, 57.
3. Метаэвристические подходы к решению задачи коммивояжёра. *Tproger*. Режим доступа: https://tproger.ru/articles/metaevristicheskie-podhody-k-reweniyu-zadachi-kommivoyazhyora (дата обращения 14 апреля 2025).
4. Решение задачи коммивояжёра методом ближайшего соседа на Python. *Habr*. Режим доступа: https://habr.com/ru/articles/329604/ (дата обращения 14 апреля 2025).
5. Collection of metaheuristic solutions for Travelling Salesman Problem. GitHub. Available online: https://github.com/r4nd0lph-c/travelling_salesman_problem (accessed on 14 April 2025).
6. Nguyen V.T. A collection of the state-of-the-art MEta-heuristics ALgorithms in PYthon (MEALPY). *Zenodo*. 2020. DOI: https://doi.org/10.5281/zenodo.3711948.
7. Liang D. Intro – Python Algorithms: Traveling Salesman Problem. *Medium*. 2024, 17 Jul. Available online: https://medium.com/@davidlfliang/intro-python-algorithms-traveling-salesman-problem-ffa61f0bd47b (accessed on 14 April 2025).
8. Traveling Salesman Problem using Branch And Bound. *GeeksforGeeks*. 2023, 30 Apr. Available online: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/ (accessed on 14 April 2025).
9. Shunji-umetani/tsp-solver: Metaheuristics for Traveling Salesman Problem. *GitHub*. Available online: https://github.com/shunji-umetani/tsp-solver (accessed on 14 April 2025).
10. JooZef315/TSP-by-Metaheuristics: This Python package provides implementations of three metaheuristic algorithms to solve the Traveling Salesman Problem. *GitHub*. Available online: https://github.com/JooZef315/TSP-by-Metaheuristics (accessed on 14 April 2025).

REFERENCES

1. Wirsanski E. *Genetic algorithms in Python*. Moscow: DMK Press, 2020, 286. (In Russ)
2. Kuvshinova E.A. *Solving the traveling salesman problem using branch, boundary, and nearest neighbor methods in Python*. Tolyatti: TSU, 2019, 57. (In Russ)
3. Metaheuristic approaches to solving the traveling salesman problem. *Tproger*. Available online: https://tproger.ru/articles/metaevristicheskie-podhody-k-reweniyu-zadachi-kommivoyazhyora (accessed on 14 April 2025). (In Russ)
4. Solving the traveling salesman problem using the nearest neighbor method in Python. *Habr*. Available online: https://habr.com/ru/articles/329604/ (accessed on 14 April 2025). (In Russ)

5. Collection of metaheuristic solutions for Travelling Salesman Problem. GitHub. Available online: https://github.com/r4nd0lph-c/travelling_salesman_problem (accessed on 14 April 2025).
6. Nguyen V.T. A collection of the state-of-the-art MEta-heuristics ALgorithms in PYthon (MEALPY). *Zenodo*. 2020. DOI: https://doi.org/10.5281/zenodo.3711948.
7. Liang D. Intro – Python Algorithms: Traveling Salesman Problem. *Medium*. 2024, 17 Jul. Available online: https://medium.com/@davidlfliang/intro-python-algorithms-traveling-salesman-problem-ffa61f0bd47b (accessed on 14 April 2025).
8. Traveling Salesman Problem using Branch And Bound. *GeeksforGeeks*. 2023, 30 Apr. Available online: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/ (accessed on 14 April 2025).
9. Shunji-umetani/tsp-solver: Metaheuristics for Traveling Salesman Problem. *GitHub*. Available online: https://github.com/shunji-umetani/tsp-solver (accessed on 14 April 2025).
10. JooZef315/TSP-by-Metaheuristics: This Python package provides implementations of three metaheuristic algorithms to solve the Traveling Salesman Problem. *GitHub*. Available online: https://github.com/JooZef315/TSP-by-Metaheuristics (accessed on 14 April 2025).

ИНФОРМАЦИЯ ОБ АВТОРАХ / INFORMATION ABOUT THE AUTHORS

Картофлицкий Владимир Валерьевич, студент, Российский экономический университет им. Г.В. Плеханова, г. Москва, Россия.
e-mail: v_neo_l@mail.ru

Vladimir Valerievich Kartoflitskiy, student, Plekhanov Russian University of Economics, Moscow, Russia.

Горемыкина Галина Ивановна, кандидат физико-математических наук, доцент кафедры математических методов в экономике, Российский экономический университет им. Г.В. Плеханова, г. Москва, Россия.
e-mail: Goremykina.GI@rea.ru

Galina Ivanovna Goremykina, Candidate of Physical and Mathematical Sciences, Associate Professor of the Department of Mathematical Methods in Economics, Plekhanov Russian University of Economics, Moscow, Russia.

Статья поступила в редакцию 22.02.2025; одобрена после рецензирования 30.03.2025; принята к публикации 30.04.2025.

The article was submitted 22.02.2025; approved after reviewing 30.03.2025; accepted for publication 30.04.2025.