УДК 004.021:004.41 URL: https://ptsj.bmstu.ru/catalog/icec/inf_tech/1066.html

СОВРЕМЕННЫЕ АЛГОРИТМЫ И ЦИФРОВЫЕ ИНСТРУМЕНТЫ ДЛЯ РЕШЕНИЯ ЗАДАЧ ДИСКРЕТНОЙ ОПТИМИЗАЦИИ

Е.А. Кирюхин egorkiruhin5707@gmail.com

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация

Проанализированы процессы постановки задач дискретной оптимизации и методы их решения. В качестве примера рассмотрена задача коммивояжера. Для постановки ее условия введен ряд определений и классификаций, а затем построена ее математическая модель. На примере данной задачи выполнено сравнение двух распространенных способов решения оптимизационных задач с помощью современных цифровых инструментов. В качестве первого способа рассмотрено решение с использованием надстройки Microsoft Excel «Поиск решения». В качестве второго решения рассмотрена разработка программного продукта, реализующего один из методов решения задач дискретной оптимизации, а именно эволюционную стратегию популяционного алгоритма. Данный метод также был использован и в первом способе решения. Стоит учитывать, что программная реализация такого решения может быть осуществлена на любом современном языке программирования. В статье приведен пример программного решения, разработанного на языке программирования C++. Решение с помощью разработанного программного обеспечения определило оба верных маршрута с равной минимальной длиной для заданных исходных данных, в отличие от решения с использованием надстройки «Поиск решения», которое ввиду специфики своей организации выдало только одно верное решение. На основе результатов сравнения двух решений обоснована целесообразность решения оптимизационных задач путем разбиения условия поставленной задачи на входные и выходные данные с последующим составлением математической модели задачи, в частности, заданием целевой функции и разработкой универсального программного решения на любом современном языке программирования.

Ключевые слова: оптимизационные задачи, задачи дискретной оптимизации, задача коммивояжера, методы решения оптимизационных задач, популяционный алгоритм, надстройка Microsoft Excel «Поиск решения», язык программирования C++

Введение. Каждый человек сталкивался с ситуацией, когда результата можно достичь несколькими способами. В таком случае приходится выбирать наилучший вариант, который позволит получить желаемый результат при минимальных затратах. Выбор наилучшего варианта из всех возможных называется *оптимизацией* (от лат. *optimus* — наилучший), а задача, которая требует поиска наилучшего варианта, — *оптимизационной задачей* [1].

Одна из основных задач алгоритма решения оптимизационной задачи — обеспечение баланса между скоростью поиска решения и полнотой описания

 ISSN 2541-8009

значимых факторов. Многие задачи, возникающие в прикладных областях промышленности, энергетики и транспорта, можно свести к задачам нахождения экстремумов целевых функций. Так, широко известная задача коммивояжера сводится к реализации алгоритма поиска минимального пути при разработке схемы перевозок транспортным средством. Особенно актуальны такие задачи для маршрутов с высокой себестоимостью перевозок, например, для фидерных перевозок на морских бассейнах с вариативной маршрутизацией и линейкой транспортов — фидерных контейнеровозов различной грузовместимости.

В статье рассмотрены алгоритмы поиска оптимальных маршрутов на примере фидерных контейнерных перевозок в Черноморском бассейне, в котором транспортное средство осуществляет перевозки с вариативными схемами погрузки-выгрузки контейнеров в основных портах.

Основными методами решения оптимизационных задач маршрутизации являются методы перебора, генетических и популяционных алгоритмов, методы жадных алгоритмов и т. д. В качестве алгоритмов для решения задачи коммивояжера могут быть выбраны методы полного и случайного перебора — универсальные инструменты условной оптимизации в статических детерминированных задачах. Эти инструменты реализованы в процедурах и функциях таких программных продуктов, как Microsoft Excel, Mathcad, Matematica и других, что облегчает реализацию программного кода алгоритма оптимизации. Сравнение скорости схождения итерационных алгоритмов перебора в цикле однокритериальной оптимизации в различных программных средах позволяет выбирать наиболее эффективные инструменты вычислений для конкретных прикладных задач.

В данной статье описана разработка схемы перевозок — порядка перемещения между пунктами назначения, обеспечивающего минимальный суммарный пробег транспортного средства. Постановка задачи предусматривает разработку схемы движения транспортного средства между определенными пунктами загрузки-выгрузки, структура связей и расстояние между которыми известно.

Решать оптимизационные задачи, используя возможности цифровых инструментов, можно двумя способами:

1) с помощью надстройки Microsoft Excel «Поиск решения»;

2) с помощью разработанного программного обеспечения, реализующего один из методов решения конкретной задачи оптимизации.

Рассмотрим эти два способа решения оптимизационных задач на примере решения *задачи коммивояжера (Traveling Salesman Problem)*. Она заключается в нахождении самого выгодного маршрута, проходящего через указан-

ISSN 2541-8009

ные города хотя бы по одному разу с последующим возвратом в исходный город [2]. В условиях задачи указывают критерий выгодности маршрута и соответствующие матрицы расстояний, стоимости перевозок и т. п.

Элементы теории графов для решения задач оптимизации маршрута. Для постановки задачи коммивояжера следует рассмотреть ряд определений из раздела дискретной математики под названием «*Теория графов*» [3].

Граф — это математическая структура (рис. 1), состоящая множества вершин, представляющих объекты или сущности и множества ребер, представляющих связи или отношения между парами вершин.

Взвешенный граф — это граф (рис. 2), в котором каждому ребру присвоено числовое значение — вес. Это может быть расстояние, время, стоимость или любая другая характеристика.

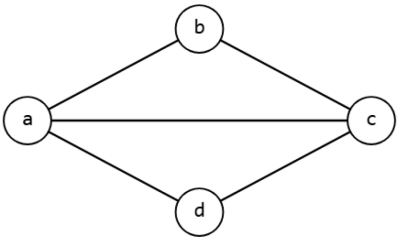

Рис. 1. Пример графа

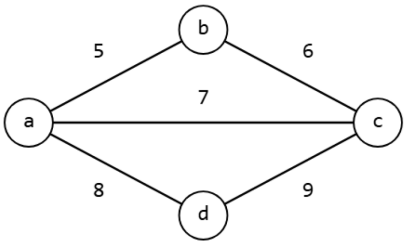

Рис. 2. Пример взвешенного графа

Матрица расстояний графа — это квадратная матрица (рис. 3), в которой строки и столбцы матрицы обозначены вершинами графа, а элемент на пересечении строки и столбца равен весу ребра между вершинами (например, расстоянию). Элементы, расположенные на главной диагонали матрицы, всегда заполняются нулями. Если между двумя вершинами графа нет связи, то элемент на пересечении строки и столбца этих двух вершин также заполняется нулем.

	a	b	c	d
a	0	5	7	8
b	5	0	6	0
c	7	6	0	9
d	8	0	9	0

Рис. 3. Пример матрицы расстояний графа

 ISSN 2541-8009

Маршрут в графе — это чередующаяся последовательность вершин и ребер, начинающаяся и заканчивающаяся вершиной.

Длина маршрута в графе — это количество ребер в рассматриваемом маршруте.

Цепь — это маршрут (рис. 4), у которого все ребра различны.

Цикл — это цепь (рис. 5), которая начинается и заканчивается в одной и той же вершине.

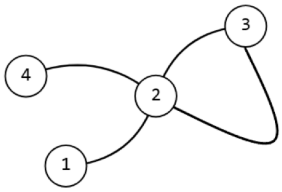

Рис. 4. Пример цепи

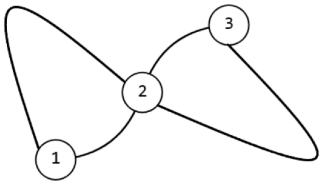

Рис. 5. Пример цикла

Гамильтонов граф — это граф, в котором существует цикл, проходящий каждую вершину графа ровно один раз, такой цикл называется гамильтоновым.

Математическое программирование. Чтобы определить, к какому типу задач оптимизации относится задача коммивояжера, необходимо рассмотреть их классификацию.

Раздел прикладной математики, изучающий теории и методы решения задач о нахождении экстремумов функций (задач оптимизации) на множествах, определяемых линейными и нелинейными ограничениями (равенствами и неравенствами), называют математическим программированием (МП) [1]. Поэтому в любой постановке оптимизационной задачи выделяют объекты и процедуры, по которым строят математическую модель. В МП можно выделить два направления [4]:

1) детерминированные задачи — задачи, в которых вся исходная информация является полностью определенной;

2) стохастические задачи — задачи, в которых либо исходная информация содержит элементы неопределенности, либо некоторые параметры имеют случайный характер с известными вероятностными характеристиками.

Целевая функция — это вещественная или целочисленная функция одной или нескольких переменных, подлежащая оптимизации в целях решения некоторой оптимизационной задачи [4].

Общая задача МП формулируется следующим образом.

Необходимо найти вектор $x=(x_1,x_2,...,x_n)$, удовлетворяющий системе ограничений в виде равенств

$$g_i(x_1,x_2,...,x_n)=b_i,\ i\in[1;k],$$

и неравенств

$$g_j(x_1,x_2,...,x_n)\le b_j, j\in[k+1;m],$$

и достигающий экстремума целевой функции $f(x_1,x_2,...,x_n)$. Предполагается, что функции $f(x_1,x_2,...,x_n)$, $g_l(x_1,x_2,...,x_n)$ и постоянные $b_l, l\in[1;m]$, известны. Если дана зависимость

$$g_i(x_1,x_2,...,x_n)=\sum_{j=1}^{n}a_{ij}x_j, i\in[1;m],$$

$$f(x_1,x_2,...,x_n)=\sum_{j=1}^{n}c_jx_j,$$

где a_{ij}, c_j — известные постоянные, то получаем задачу линейного программирования (ЗЛП). Если хотя бы одна из функций $f(x_1,x_2,...,x_n)$, $g_i(x_1,x_2,...,x_n)$ нелинейна, то соответствующая задача является задачей нелинейного программирования (ЗНП) [4].

Наиболее изученным разделом МП является ЗЛП. Для решения таких задач разработан целый ряд эффективных методов, алгоритмов и программ. Среди ЗНП наиболее глубоко изучены задачи выпуклого программирования. Это задачи минимизации, в которых целевая функция и допустимое множество являются выпуклыми. Такие задачи имеют единственный глобальный экстремум. Также среди ЗНП выделяют задачи невыпуклого программирования. В таких задачах часто присутствует множество локальных минимумов [4].

Если в задаче МП целевая функция и ограничения не зависят от времени или каких-либо динамических изменений, т. е. остаются неизменными в процессе поиска решения, то такую задачу называют статической. В ином случае, если какие-либо параметры изменяются с течением времени, такую задачу относят к динамическим задачам оптимизации [5].

Стоит учитывать, что в рассматриваемой задаче МП переменные могут быть различных типов. Так, если переменные принимают значение из непрерывного множества (например, вещественные числа), то такая задача называется непрерывной. Если переменные ограничены дискретными значениями (например, целыми числами), то дискретной. Также существуют смешанные задачи, в которых часть переменных непрерывная, а часть — дискретная [5].

 ISSN 2541-8009

Общие сведения о задаче коммивояжера и ее постановка. Комбинаторика — раздел математики, посвященный решению задач выбора и расположения элементов некоторого, обычно конечного множества в соответствии с заданными правилами [1].

Задача коммивояжера — одна из самых известных задач комбинаторики и комбинаторной оптимизации [1]. Она служит упрощенной моделью многих других задач оптимизации, а также часто является подзадачей.

Особенность задачи коммивояжера — необходимость дополнительно учитывать расстояния между городами, которые считаются известными. Суммарное пройденное расстояние может быть заменено на количество затраченного времени, стоимость проезда или предполагать другие произвольные значения [2].

Классическая постановка задачи коммивояжера выглядит следующим образом [6].

Имеется *N* городов; выезжая из первого города, коммивояжер должен побывать во всех городах по одному разу и вернуться в исходный город. Задача заключается в построении пути, при котором суммарное пройденное расстояние (количество затраченного времени, стоимость проезда и т. д.) будет минимальным.

Постановка задачи коммивояжера с использованием терминов теории графов [3] может звучать следующим образом.

Задан взвешенный граф. Известна начальная вершина. Необходимо найти в нем гамильтонов цикл кратчайшей длины.

Математическая модель задачи коммивояжера. Задача коммивояжера может быть сформулирована как целочисленная введением переменных $x_{ij} = 1,$ если маршрут включает ребро, ведущее из *i*-й в *j*-ю вершину, и $x_{ij} = 0$ в противном случае [7]. Тогда можно задать математическую модель задачи, т. е. записать ее целевую функцию и систему ограничений. Из постановки задачи коммивояжера можно выделить ряд значений, среди которых *N* — число вершин, связанных ребрами; C — матрица затрат на переход из *i*-й в *j*-ю вершину; x — бинарная матрица.

Также следует наложить на эти значения характерные ограничения:

$$N, C_{ij} \in [0; +\infty),$$
$$x_{ij} \in \{0, 1\},$$
$$\forall i, j \in [1; N].$$

Пусть $x_{ij} = 1,$ когда совершается переход из *i*-й в *j*-ю вершину, и $x_{ij} = 0,$ если *j*-я вершина не была пройдена [7].

Зададим функцию *f*, являющуюся функцией одной или нескольких переменных. Она будет определять критерий выгодности маршрута (кратчайший, самый дешевый и т. д.) [7]. В рассматриваемой задаче функция *f* определяет минимальное суммарное пройденное расстояние. Тогда *целевая функция* будет иметь следующий вид:

$$\min_{x\in\delta} f(C,x) = \min_{x\in\delta} \sum_{i=1}^{N}\sum_{j=1}^{N} C_{ij}x_{ij},$$

где δ — это множество допустимых альтернатив, на которые накладываются следующие *ограничения*:

$$\sum_{i=1}^{N} x_{ij} = 1,\ \forall i \in [1;N],$$

$$\sum_{j=1}^{N} x_{ij} = 1,\ \forall j \in [1;N].$$

Это ограничение задает условие единственности прохода через каждую вершину искомым маршрутом.

Поскольку известны число вершин и значения матрицы расстояний, которые не изменяются с течением поиска решения и не носят случайный характер, задачу коммивояжера можно отнести к *статическим детерминированным задачам линейного программирования*. Далее будем рассматривать данную задачу в целочисленной, т. е. *дискретной*, постановке.

Методы решения задачи коммивояжера. Перед просмотром методов решения оптимизационных задач, в частности, задачи коммивояжера, введем определение сходимости оптимизации.

Сходимость оптимизации — это показатель метода оптимизации, который определяет, как быстро на каждой итерации уменьшается расстояние до точного значения решения [8].

Различают *глобальную* и *асимптотическую сходимость*. *Глобальная сходимость* означает, что при любом выборе начальной точки последовательность сходится к точке, удовлетворяющей необходимым условиям оптимизации. *Асимптотическая сходимость* подразумевает поведение последовательности в окрестности предельной точки [8].

Существует множество методов, с помощью которых можно решить задачу коммивояжера. Рассмотрим основные из них.

1. *Полный перебор* — метод решения задачи путем перебора всех возможных вариантов [9]. Сложность полного перебора зависит от количества всех возможных решений задачи.

 ISSN 2541-8009

2. *Жадный алгоритм* — алгоритм нахождения наикратчайшего расстояния путем выбора самого короткого, еще не выбранного ребра, при условии, что оно не образует цикла с уже выбранными ребрами [9].

3. *Метод минимального остовного дерева (деревянный алгоритм)* — алгоритм нахождения наикратчайшего расстояния путем построения графа и выделения в нем минимального остова [9].

Дерево — это связный граф без циклов [3]. Между любыми двумя вершинами дерева существует единственный путь.

Остов — это подграф графа, задающий дерево на каждой компоненте связности графа [3]. Для связного графа остов — это дерево, покрывающее все вершины исходного графа. Остов получается, если из исходного графа удалить максимальное число ребер, входящих в циклы, но при этом не нарушать связность графа.

4. *Метод имитации отжига* — метод решения различных оптимизационных задач, основанный на моделировании реального физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твердое, в том числе при отжиге металлов [9]. В процессе работы алгоритма хранится текущее решение, которое является промежуточным результатом. А после работы алгоритма текущее решение становится итоговым.

5. *Метод ветвей и границ* — метод решения различных оптимизационных задач, заключающийся в упорядоченном переборе вариантов и рассмотрении лишь тех из них, которые оказываются по определенным признакам перспективными, и отбрасывании бесперспективных вариантов [9].

6. *Популяционные алгоритмы* — класс методов оптимизации, которые работают с множеством решений (популяцией) одновременно [9]. Эти алгоритмы вдохновлены биологическими, социальными или природными процессами, такими как эволюция, поведение стай или колоний. Они используются для поиска приближенных решений сложных задач, где точные методы неэффективны из-за большого пространства поиска или высокой вычислительной сложности.

Рассмотрим классификацию популяционных алгоритмов, поскольку в дальнейшем будем использовать один из видов данного алгоритма для решения задачи коммивояжера.

Популяционные алгоритмы. Популяционные алгоритмы можно различать по типу операторов [9], необходимых для реализации конкретного метода поиска решения.

1. Наличие *оператора обмена информацией*. Из названия этого оператора следует, что он отвечает за передачу знаний между решениями. Он служит основным оператором *муравьиного алгоритма*.

2. Наличие *оператора перемещения*. Данный оператор используется для изменения позиций решений в пространстве поиска и используется, например, в *роевом интеллекте*.

3. Наличие *оператора скрещивания*. Примером такого алгоритма может послужить *дифференциальная эволюция*, которая для поиска лучшего решения использует комбинацию из двух или ранее найденных решений.

4. Наличие *оператора мутации*. К такому классу алгоритмов можно отнести *эволюционные стратегии*. В них поиск решения осуществляется путем создания начальной популяции и рассмотрения мутаций каждой особи по отдельности с целью определения кандидата на лучшее решение. Далее рассмотрим решение задачи коммивояжера с использованием данного алгоритма.

Отметим, что есть алгоритмы, использующие одновременно несколько операторов. К таким алгоритмам относится, например, *генетический алгоритм*, который основан на принципах естественного отбора в процессе эволюции. В нем используются операторы мутации и скрещивания для создания новых решений, лучшие из которых отбирает функция приспособленности.

Рассмотрим механизм работы популяционного алгоритма, реализованного на основе эволюционной стратегии, на примере задачи коммивояжера.

Популяция — это совокупность всех предложенных решений, рассмотренных на одной итерации.

Особь — это член популяции, в соответствии с биологической моделью для подражания.

Потомок — это результат мутации особи.

Агент — это термин, который используется для общего обозначения членов популяции. В разных видах популяционных алгоритмов агенты могут быть представлены частицами, муравьями и т. д.

Локальные данные — это данные, принадлежащие конкретной особи, например положение, скорость, значение целевой функции.

Глобальные данные — это данные, принадлежащие общей среде, к ним относятся целевая функция, направление оптимизации, параметры и т. д.

Выберем в качестве *популяции* множество путей. В таком случае в роли *особи* будет выступать отдельный путь. Тогда к *локальным данным* каждой особи будет относиться ограниченное множество городов $(N + 1)$. В таком случае в качестве оценки пригодности, используемой для определения популяции, будет принята длина пути между всеми городами. Чем меньше длина пути, тем лучше особь. Самые приспособленные особи из популяции проходят отбор и переходят к следующей итерации. Число итераций зависит от значения длины пути. В ходе поиска решения длина пути с каждой итера-

 ISSN 2541-8009

цией уменьшается и достигает искомого экстремума после определенного количества итераций.

Приведем *пример работы рассматриваемого алгоритма*. Предположим, что есть пять городов: 0, 1, 2, 3, 4. Продавец находится в городе 0 и должен найти кратчайший маршрут, чтобы проехать через все города и вернуться в город 0. В данной постановке рассмотрим, что пути есть только между двумя соседними городами. Пример особи, представляющей выбранный путь, показан на рис. 6.

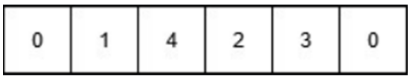

Рис. 6. Пример особи

Эта особь подвергается мутации. Во время мутации положение двух городов меняется местами, образуя новую конфигурацию, за исключением первой и последней клеток, поскольку они представляют собой начало и конец.

Поскольку пути между первым и четвертым городами не существует, исходная особь имела длину пути, равную *INT_MAX*, т. е. максимально возможному целочисленному значению. После мутации (рис. 7) у нового потомка длина пути уже равна некому конечному целочисленному значению, что является гораздо более оптимизированным ответом, чем первоначальное предположение. Именно так популяционный алгоритм оптимизирует решения сложных задач.

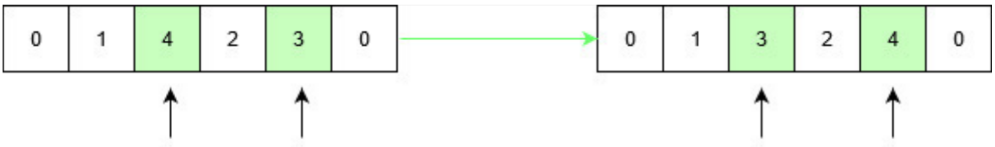

Рис. 7. Вариант организации перестановок (мутаций)

Анализ исходных данных. Чтобы сравнить два способа решений оптимизационных задач с помощью цифровых инструментов, необходимо протестировать оба решения на исходных данных, для которых заранее известен результат.

Рассмотрим решение задачи коммивояжера на примере матрицы морских расстояний между портами Черноморского бассейна (рис. 8), в которой все расстояния выражены в морских милях. В качестве начального порта выберем Стамбул «0». Стоит отметить, что поскольку исходная матрица симметрична, т. е. элементы, расположенные выше главной диагонали, и элементы, расположенные ниже главной диагонали, равны, пути могут быть разнонаправленными. В таком случае нам известно, что между рассматриваемыми портами Черноморского бассейна реализованы два разнонаправленных цикличных круговых маршрута, длины которых равны и составляют 1498 морских миль. Это значит, что для предложенных исходных данных существуют два равносильных решения, различающиеся направлением обхода портов, но имеющие равную длину.

			0	1	2	3	4	5	6	7	8	9
Стамбул	0	0	0	146	196	299	363	510	593	553	479	452
Варна	1	1	146	0	50	265	402	545	605	577	485	444
Констанца	2	2	196	50	0	216	386	528	579	534	408	402
Севастополь	3	3	299	265	216	0	248	367	403	355	260	219
Самсун	4	4	363	402	386	248	0	158	240	226	206	213
Трабзон	5	5	510	545	528	367	158	0	108	131	186	239
Поти	6	6	583	605	579	403	240	108	0	57	163	227
Сухуми	7	7	553	577	534	355	226	131	57	0	114	177
Туапсе	8	8	479	485	408	260	206	186	163	114	0	67
Новороссийск	9	9	452	444	402	219	213	239	227	177	67	0

Рис. 8. Матрица морских расстояний между портами Черноморского бассейна

Решение задачи коммивояжера с помощью надстройки Microsoft Excel «Поиск решения». *Надстройка Microsoft Excel «Поиск решения»* служит мощным средством поиска решений и применяется при решении задач оптимизации. Процедура поиск решения позволяет находить оптимальное значение формулы, содержащейся в ячейке, которую называют *целевой*. Процедура работает с группой ячеек, связанных с формулой в целевой ячейке [10].

Надстройка «Поиск решения» является стандартной надстройкой Microsoft Excel. Для получения заданного результата по формуле процедура изменяет значения во влияющих ячейках. Для уменьшения интервала значений, используемых в модели, используются ограничения значений [11].

Надстройка «Поиск решения» поставляется вместе с Microsoft Excel, но по умолчанию отключена. Ее можно подключить в разделе «*Надстройки*», меню «*Параметры Excel*». В результате подключения во вкладке «*Данные*» появится новая группа «*Анализ*» с кнопкой запуска процедуры «*Поиск решения*».

 ISSN 2541-8009

В процедуре «Поиск решения» для решения задач оптимизации предусмотрено три метода: «*Поиск решения нелинейных задач методом ОПГ*», «*Поиск решения лин. задач симплекс-методом*», «*Эволюционный поиск решения*». Для решения поставленной задачи будем использовать «*Эволюционный поиск решения*». Он основан на *эволюционной стратегии популяционного алгоритма* и предназначен для решения оптимизационных задач. Принцип работы данного алгоритма был рассмотрен ранее.

Алгоритм решения задачи коммивояжера реализуем следующим образом [10]:

1) создадим три колонки (рис. 9): «*Путь*», «*Расст*», «*Длинна пути*», где «*Путь*» — итоговый маршрут, суммарное расстояние которого самое минимальное;

«*Расст*» — длина промежуточного пути;

«*Длина пути*» — длина итогового маршрута;

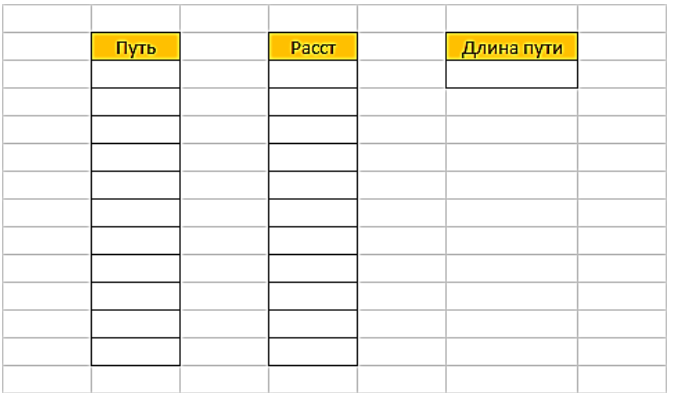

Рис. 9. Набор искомых данных

2) ячейкам *Q4* и *Q14* присвоим значение «0», поскольку итоговый маршрут должен начинаться и заканчиваться 0-м городом, а в ячейку *S5* запишем формулу «*=ИНДЕКС($F$4:$O$13;Q4+1;Q5+1)*» и скопируем ее на диапазон *S5:S14* (рис. 10);

3) в ячейку *U4* запишем целевую формулу «*=СУММ(S5:S14)*» (рис. 11);

4) вызовем процедуру «*Поиск решения*» и в открывшемся окне «*Параметры поиска решения*» выставим значения, показанные на рис. 12, а именно: зададим целевую функцию, расположив ее в ячейке *U4*, наложим ограничения на варьируемые значения, в совокупности образующие итоговый путь, а также укажем метод решения поставленной задачи;

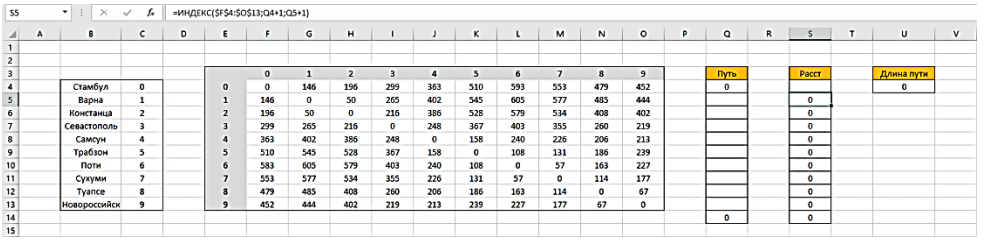

Рис. 10. Заполнение колонки «Расст»

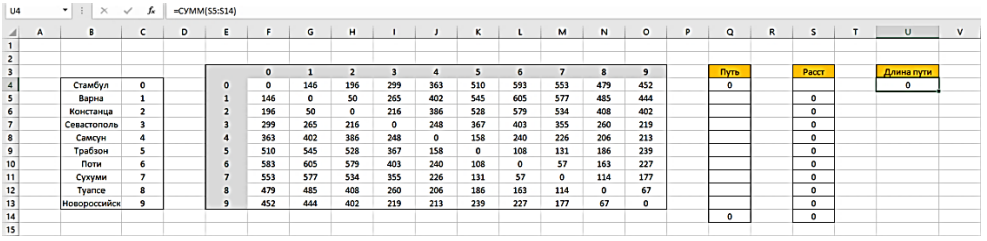

Рис. 11. Заполнение колонки «*Длина пути*»

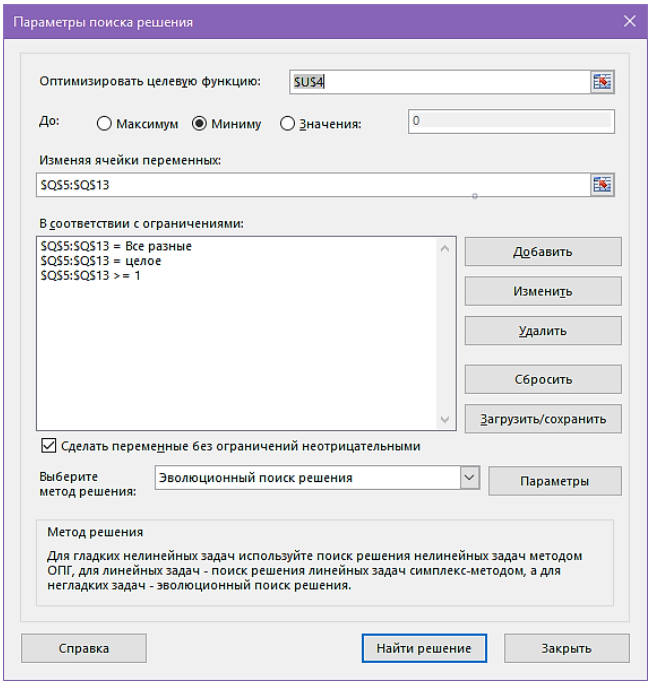

Рис. 12. Процедура «*Поиск решения*»

5) введем значения параметров (рис. 13), необходимых для работы выбранного метода, нажмем на кнопку «*Найти решение*» и будем ожидать выполнения процедуры.

Результат выполнения процедуры представлен на рис. 14. На выполнение процедуры была затрачена 51 с. Было создано 48 тыс. поколений особей и совершено 720 тыс. итераций перестановок в особях. Поиск решения остановился по истечении 30 с отсутствия улучшения в решении с момента определения лучшего маршрута.

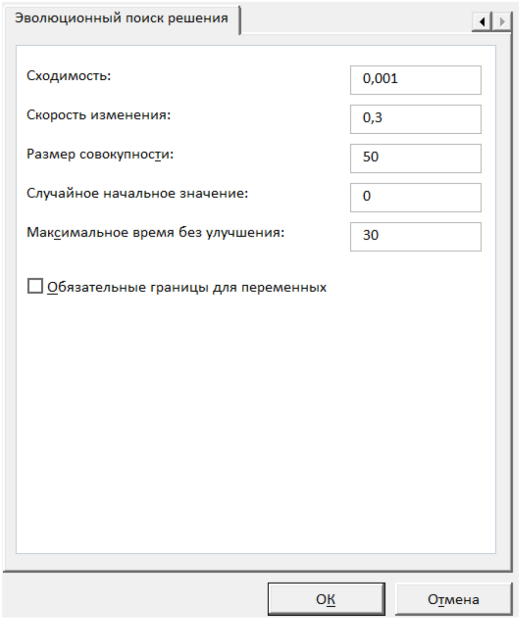

Рис. 13. Параметры популяционного алгоритма

Путь	Расст	Длина пути
0		1498
4	363	
5	158	
6	108	
7	57	
8	114	
9	67	
3	219	
2	216	
1	50	
0	146	

Рис. 14. Результат выполнения процедуры «Поиск решения»

Разработка программного решения задачи коммивояжера на языке программирования C++. Ввиду ресурсозатратности решения задачи в Microsoft Excel сделан вывод о необходимости разработки программного обеспечения, реализующего аналогичный метод решения задачи коммивояжера, а именно популяционного алгоритма, основанного на эволюционной стратегии.

Для обеспечения многовариантности постановки задачи и скорости схождения результатов решение задачи коммивояжера было реализовано на языке программирования C++. Отметим, что данный язык программирования широко используется для обучения программированию в старших классах школы и на первых курсах высших учебных заведений. Поэтому учащийся, освоивший данные ступени образования, без трудностей может разобраться в принципе работы программного решения поставленной задачи.

Алгоритм работы программного решения задачи коммивояжера построен на основе *эволюционной стратегии популяционного алгоритма*. Данный вид популяционного алгоритма позволяет эффективно исследовать пространство решений и избегать «застревания» в локальных минимумах. Программное обеспечение построено таким образом, что пользователь сам может задавать или вносить изменения в существующие значения исходных данных (количества городов и матрицы расстояний), а также параметров оптимизации (размера популяции, порога сходимости, вероятности мутации и максимального времени без улучшения). Некорректный ввод проверяется по условию превышения заданного значения. По исходным данным осуществляется поиск искомого маршрута с минимальной длиной путем задания начальной популяции с последующими перестановками пунктов, формирующих особь популяции. У пользователя на экране отображаются все поколения популяций, одна или несколько особей с минимальной длиной маршрута и время выполнения программного решения.

Стоит учитывать, что на время работы программного решения сильно влияет вывод промежуточных результатов, т. е. *логирование*. Поскольку при решении задачи при помощи надстройки *Microsoft Excel «Поиск решения»* информация о текущей генерации (номер генерации, минимальное расстояние и список всех особей) не выводилась, стоит сравнивать оба решения без логирования промежуточной информации. Таким образом, на выполнение программного решения было затрачено чуть больше 30 с. Было создано 100 712 поколений особей и суммарно совершено 1 510 680 итераций перестановок городов. Поиск решения был прерван по истечению 30 с отсутствия улучшения найденного решения. Сам результат представлен на рис. 15. Также было рассмотрено выполнение программного решения с логированием про-

 ISSN 2541-8009

межуточных результатов, таких как номер текущего поколения, минимальная длина маршрута в этом поколении и все его особи с длинами их маршрутов. Результаты такого решения аналогичны прошлому, но время выполнения составило почти 728 с (рис. 16).

```
Search stopped due to timeout (no improvement for 30 seconds)

Best route(s) with minimum distance (1498):
01239876540
04567893210
Total time: 30251 ms
```

Рис. 15. Результат работы программного решения без логирования

```
Search stopped due to timeout (no improvement for 30 seconds)

Best route(s) with minimum distance (1498):
01239876540
04567893210
Total time: 727919 ms
```

Рис. 16. Результат работы программного решения с логированием

Программное решение задачи коммивояжера разработано в соответствии с принципами структурного программирования и состоит из четырех основных узлов.

1. При поиске решения используется набор исходных данных и параметров оптимизации, представляющих собой константы и переменные значения. К ним относятся:

а) *NUM_CITIES* — целочисленная константа, задающая число городов;

б) *distanceMatrix* — целочисленный двумерный массив (матрица), содержащий в себе значения расстояний между городами;

в) *POPULATION_SIZE* — целочисленная константа, определяющая размер популяции геномов (маршрутов);

г) *CONVERGENCE_THRESHOLD* — вещественная константа, определяющая порог сходимости;

д) *MUTATION_RATE* — вещественная константа, определяющая вероятность мутации особи;

е) *MAX_STAGNATION_TIME* — целочисленная константа, определяющая максимальное время без улучшения (в секундах);

ж) *maxGenerations* — искусственно введенная целочисленная переменная, по значению которой происходит ограничение на количество поколений популяций.

Ограничение на количество поколений популяций было введено для того, чтобы отследить ошибки в работе алгоритма и избежать бесконечного цикла. Начальные значения всех параметров оптимизации для решения задачи коммивояжера с учетом известных исходных данных представлены на рис. 17.

```
#define NUM_CITIES 10               // Количество городов
#define POPULATION_SIZE 50          // Размер популяции
#define CONVERGENCE_THRESHOLD 0.001 // Порог сходимости
#define MUTATION_RATE 0.3           // Вероятность мутации
#define MAX_STAGNATION_TIME 30      // Максимальное время без улучшения (в секундах)

int maxGenerations = 2000000; // Максимальное количество поколений

// Матрица расстояний между городами
int distanceMatrix[NUM_CITIES][NUM_CITIES] = {
    {0, 146, 196, 299, 363, 510, 593, 553, 479, 452},
    {146, 0, 50, 265, 402, 545, 605, 577, 485, 444},
    {196, 50, 0, 216, 386, 528, 579, 534, 408, 402},
    {299, 265, 216, 0, 248, 367, 403, 355, 260, 219},
    {363, 402, 386, 248, 0, 158, 240, 226, 206, 213},
    {510, 545, 528, 367, 158, 0, 108, 131, 186, 239},
    {583, 605, 579, 403, 240, 108, 0, 57, 163, 227},
    {553, 577, 534, 355, 226, 131, 57, 0, 114, 177},
    {479, 485, 408, 260, 206, 186, 163, 114, 0, 67},
    {452, 444, 402, 219, 213, 239, 227, 177, 67, 0}
};
```

Рис. 17. Начальные значения параметров оптимизации и исходных данных

2. Для хранения информации о маршрутах используется структура *Path* (рис. 18). Она содержит два поля, одно из которых представляет собой строку (*route*), определяющую последовательность городов в маршруте, а второе — целочисленную переменную (*total_distance*), содержащую длину маршрута. Данная структура упрощает обработку данных о маршрутах, их сортировку и отбор лучших вариантов.

```
struct Path {
    string route;        // Маршрут
    int total_distance;  // Длина маршрута
};
```

Рис. 18. Листинг структуры Path

 ISSN 2541-8009

3. Для создания особей и работы с ними используются специальные функции. Перечислим их ниже и опишем их действие:

а) функция *getRandomNumber* служит для генерации случайного числа в заданном диапазоне (рис. 19). Она используется для выбора случайных городов при создании и мутации маршрутов;

```
// Функция для генерации случайного числа в диапазоне [start, end)
int getRandomNumber(int start, int end) {
    int range = end - start;
    int randomNumber = start + rand() % range;
    return randomNumber;
}
```

Рис. 19. Листинг функции getRandomNumber

б) функция *isCharacterInString* осуществляет проверку на наличие символа в строке, то есть города в маршруте (рис. 20);

```
// Функция для проверки, содержится ли символ в строке
bool isCharacterInString(string str, char ch) {
    for (int i = 0; i < str.size(); i++) {
        if (str[i] == ch)
            return true; // Символ найден
    }
    return false; // Символ не найден
}
```

Рис. 20. Листинг функции isCharacterInString

в) функция *mutatePath* проводит перестановку двух любых городов в заданном маршруте (рис. 21). Во время ее выполнения дважды вызывается функция *getRandomNumber* для получения двух случайных городов;

```
// Функция для мутации пути (перестановка двух городов)
string mutatePath(string path) {
    while (true) {
        int index1 = getRandomNumber(1, NUM_CITIES); // Случайный индекс первого города
        int index2 = getRandomNumber(1, NUM_CITIES); // Случайный индекс второго города
        if (index1 != index2) { // Если индексы разные, меняем города местами
            char temp = path[index1];
            path[index1] = path[index2];
            path[index2] = temp;
            break;
        }
    }
    return path; // Возвращаем изменённый путь
}
```

Рис. 21. Листинг функции mutatePath

г) функция *generateRandomPath* создает случайный маршрут (рис. 22). Во время ее выполнения вызывается функция *getRandomNumber* для получения случайного города;

```
// Функция для генерации случайного маршрута
string generateRandomPath() {
    string path = "0"; // Начинаем с города 0
    while (true) {
        if (path.size() == NUM_CITIES) { // Если все города добавлены
            path += path[0]; // Возвращаемся в начальный город
            break;
        }
        int city = getRandomNumber(1, NUM_CITIES); // Генерация случайного города
        if (!isCharacterInString(path, (char)(city + '0'))) // Если город ещё не добавлен
            path += (char)(city + '0'); // Добавляем город в путь
    }
    return path; // Возвращаем сгенерированный путь
}
```

Рис. 22. Листинг функции generateRandomPath

д) функция *calculatePathDistance* вычисляет длину заданного маршрута по исходной матрице расстояний (рис. 23);

```
// Функция для вычисления длины маршрута
int calculatePathDistance(string path, int distanceMatrix[NUM_CITIES][NUM_CITIES]) {
    int totalDistance = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        int city1 = path[i] - '0'; // Текущий город
        int city2 = path[i + 1] - '0'; // Следующий город
        if (distanceMatrix[city1][city2] == INT_MAX) // Если путь невозможен
            return INT_MAX; // Возвращаем максимальное значение
        totalDistance += distanceMatrix[city1][city2]; // Добавляем расстояние к общей длине
    }
    return totalDistance; // Возвращаем общую длину маршрута
}
```

Рис. 23. Листинг функции calculatePathDistance

4. Основная функция имитации эволюционного развития особей *solveTSP*. Она реализует логику эволюционной стратегии популяционного алгоритма. Сначала создается начальная популяция случайных маршрутов размером *POPULATION_SIZE*. Затем в цикле выполняются сортировка маршрутов по длине, создание нового поколения путем мутации старых особей и запись лучших значений во множество решений (*bestRoutes*). Организация сортировки происходит при помощи компаратора *comparePaths* (рис. 24).

 ISSN 2541-8009

Мутация каждой особи происходит с заданной вероятностью, значение которой определяет константа *MUTATION_RATE* (рис. 25). Работа главного цикла данной функции завершается, если 30 поколений подряд прошли проверку на сходимость (рис. 26), т. е. длина их минимального маршрута монотонно уменьшалась и отношение изменения решения к уже определенному всегда было строго меньше заданного значения порога сходимости, т. е. значения константы *CONVERGENCE_THRESHOLD*. Поиск решения также может завершиться из-за превышения ограничения на максимальное количество поколений (*maxGenerations*), которое было искусственно введено для того, чтобы отследить ошибки в работе алгоритма и избежать бесконечного цикла (рис. 27). Помимо этого предусмотрен выход по истечении 30 с программного времени (*MAX_STAGNATION_TIME*), в течение которого не произошло улучшения найденного решения (рис. 28). Во время поиска решения опционально предусмотрено логирование промежуточных результатов. Поэтому для корректной работы выхода из главного цикла по истечению 30 с безрезультатного поиска наилучшего решения предусмотрена коррекция таймера (рис. 29), которая позволяет не учитывать в данном временном промежутке время, потраченное на вывод информации. В конце выводится минимальная длина маршрута и перечень маршрутов, длины которых равны найденному значению (рис. 30).

```
// Компаратор для сортировки путей по длине
bool comparePaths(Path path1, Path path2) {
    return path1.total_distance < path2.total_distance; // Сравниваем пути по длине
}
```

Рис. 24. Листинг компаратора comparePaths

```
// Создание новой популяции путём мутации
for (int i = 0; i < POPULATION_SIZE; i++) {
    Path parent = population[i]; // Родительский путь

    // Мутация пути с заданной вероятностью
    if ((double)rand() / RAND_MAX < MUTATION_RATE) {
        string newRoute = mutatePath(parent.route); // Мутация пути
        Path child;
        child.route = newRoute;
        child.total_distance = calculatePathDistance(child.route, distanceMatrix); // Вычисление длины нового пути
        newPopulation.push_back(child); // Добавляем новый путь в популяцию
    } else {
        newPopulation.push_back(parent); // Сохраняем родительский путь без изменений
    }
}
```

Рис. 25. Листинг блока кода функции solveTSP, отвечающего за мутацию маршрутов


```
// Проверка сходимости
dDistance = (minDistance - population[0].total_distance) / minDistance;
if ((dDistance < CONVERGENCE_THRESHOLD) && (dDistance > 0)) {
    stagnationCounter++; // Увеличиваем счётчик застоя
} else {
    stagnationCounter = 0; // Сбрасываем счётчик застоя
}

// Проверка условий завершения
if (stagnationCounter >= 30) {
    cout << "Search stopped due to stagnation (no improvement for 30 generations)\n";
    break; // Завершаем алгоритм, если достигнут застой
}
```

Рис. 26. Листинг блока кода функции solveTSP, отвечающего за прекращение поиска решения по достижении значения порога сходимости

```
// Основной цикл алгоритма
while (generation <= maxGenerations) {
    //...
    if (generation == maxGenerations) {
        cout << "Search stopped due to reaching maximum generations (" << maxGenerations << ")\n";
        break; // Завершаем алгоритм, если достигнуто максимальное число поколений
    }

    generation++; // Переход к следующему поколению
}
```

Рис. 27. Листинг блока кода функции solveTSP, отвечающего за прекращение поиска решения по достижении максимального количества поколений

```
// Обновление лучших маршрутов
for (int i = 0; i < POPULATION_SIZE; i++) {
    if (population[i].total_distance < minDistance) { // Если найден новый лучший путь
        minDistance = population[i].total_distance; // Обновляем минимальную длину
        bestRoutes.clear(); // Очищаем список лучших маршрутов
        bestRoutes.push_back(population[i].route); // Добавляем новый лучший маршрут
        lastImprovementTime = chrono::high_resolution_clock::now(); // Обновляем время последнего улучшения
        stagnationCounter = 0; // Сбрасываем счётчик застоя
    } else if (population[i].total_distance == minDistance) { // Если маршрут такой же длины
        // Проверяем, не содержится ли маршрут уже в списке лучших
        if (find(bestRoutes.begin(), bestRoutes.end(), population[i].route) == bestRoutes.end()) {
            bestRoutes.push_back(population[i].route); // Добавляем его в список лучших
            lastImprovementTime = chrono::high_resolution_clock::now(); // Обновляем время последнего улучшения
        }
    }
}

if (chrono::duration_cast<chrono::seconds>(chrono::high_resolution_clock::now() - lastImprovementTime).count() >= MAX_STAGNATION_TIME) {
    cout << "Search stopped due to timeout (no improvement for " << MAX_STAGNATION_TIME << " seconds)\n";
    break; // Завершаем алгоритм, если превышено время без улучшений
}
```

Рис. 28. Листинг блока кода функции solveTSP, отвечающего за обновление лучших маршрутов и завершение поиска решения по истечении времени MAX_STAGNATION_TIME

 ISSN 2541-8009

```
auto logStart = chrono::high_resolution_clock::now(); // Начало логирования

// Логирование текущего поколения
cout << "Generation: " << generation << "\n";
cout << "Best distance: " << population[0].total_distance << "\n";
cout << "Population:\n";
cout << "Route\t\tDistance" << "\n";
for (const auto& p : population) {
    cout << p.route << "\t" << p.total_distance << "\n";
}
cout << "------------------------\n";

auto logEnd = chrono::high_resolution_clock::now(); // Конец логирования
auto logDuration = chrono::duration_cast<chrono::milliseconds>(logEnd - logStart); // Время, затраченное на логирование

// Коррекция времени последнего улучшения с учётом времени логирования
lastImprovementTime += logDuration;

if (chrono::duration_cast<chrono::seconds>(chrono::high_resolution_clock::now() - lastImprovementTime).count() >= MAX_STAGNATION_TIME) {
    cout << "Search stopped due to timeout (no improvement for " << MAX_STAGNATION_TIME << " seconds)\n";
    break; // Завершаем алгоритм, если превышено время без улучшений
}
```

Рис. 29. Листинг блока кода функции solveTSP, отвечающего за коррекцию таймера и завершение поиска решения по истечении времени MAX_STAGNATION_TIME

```
// Вывод лучших маршрутов
cout << "\nBest route(s) with minimum distance (" << minDistance << "):\n";
for (const string& route : bestRoutes) {
    cout << route << "\n";
}
```

Рис. 30. Листинг блока кода функции solveTSP, отвечающего за вывод результатов

```
auto start = chrono::high_resolution_clock::now(); // Запуск таймера
//...
auto end = chrono::high_resolution_clock::now(); // Остановка таймера
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start); // Вычисление времени выполнения
```

Рис. 31. Листинг организации работы таймера

Для работы таймера используется библиотека *chrono*. В необходимом месте фиксируется время старта таймера (*start*), а по завершению — время окончания (*end*). Вычисляется разница между этим значениями (*duration*) и полученное значение далее используется (рис. 31), например для вывода пользователю на экран в миллисекундах времени работы алгоритма.

Результаты сравнения двух решений задачи коммивояжера представлены в таблице.

Сравнение двух решений задачи коммивояжера

Характеристики		Решение через Microsoft Excel с помощью надстройки *«Поиск решения»*	Программное решение на языке программирования C++
Параметры оптимизации	Количество городов	10 0 – начальный город	
	Размер популяции	50	
	Порог сходимости	0,001	
	Вероятность мутации	0,3	
	Максимальное время без улучшения, с	30	
Время, затраченное на получение результата, с		51	30,251
Количество поколений		48000	100712
Количество итераций перестановок		720000	1510680
Результат	Путь	0-4-5-6-7-8-9-3-2-1-0	
		–	0-1-2-3-9-8-7-6-5-4-0
	Длина пути, морских миль	1498	1498

Заключение. По итогам исследования получены следующие результаты.

1. Представлен анализ постановки задачи коммивояжера и методов ее решения:

а) рассмотрена классификация задач математического программирования и отношение к этим задачам задачи коммивояжера;

б) введены основные определения из раздела дискретной математики *«Теория графов»* для постановки задачи коммивояжера;

в) приведены основные методы решения оптимизационных задач, а также более детально рассмотрены классификация популяционных алгоритмов и принцип их работы.

2. На примере решения задачи коммивояжера рассмотрены два способа решения оптимизационных задач: решение при помощи надстройки Microsoft Excel «Поиск решения» и решение при помощи разработки программного обеспечения на языке программирования C++. Но только программное решение определило оба верных маршрута, а также показало лучшее быстродействие.

 ISSN 2541-8009

3. Оба решения задачи коммивояжера обосновали целесообразность организации кругового рейса транспортного судна между портами Черноморского бассейна (рис. 32).

Рис. 32. Графическое отображение полученных результатов

В статье рассмотрены два способа решения задачи коммивояжера с помощью современных средств цифровой техники. Оба способа решения основаны на эволюционной стратегии популяционного алгоритма. Решение с помощью разработанного программного обеспечения определило оба верных маршрута с равной минимальной длиной для заданных исходных данных, в отличие от решения при помощи надстройки *«Поиск решения»*, которое ввиду специфики своей организации выдало только одно верное решение. Помимо этого программное решение показало наилучшее время выполнения поставленной задачи. Также стоит учесть, что программное обеспечение предоставляет возможность логирования промежуточной информации, но следует понимать, что при активации данной функции время поиска решения сильно увеличивается. Промежуточная информация может использоваться, например, для оценки эффективности алгоритма или может выступать в роли статистических данных.

Таким образом, разработанный программный продукт ни в чем не уступает коммерческому решению от компании Microsoft. Отсюда можно сделать вывод о том, что самым простым способом решения оптимизационных задач для пользователя является использование программных продуктов, например от компании Microsoft, а самым эффективным — разработка собственного программного обеспечения, реализующего один из методов решения той или иной оптимизационной задачи, или использование уже созданных для решения конкретных программных задач.

Литература

[1] Гашков С.Б. *Дискретная математика*. Санкт-Петербург, Лань, 2025, 520 с.

[2] Шевляков А.Н. *Основы дискретной математики*. Омск, ОмГУ, 2020, 98 с.

[3] Белоусов А.И., Ткачев С.Б. *Дискретная математика*. Москва, Изд-во МГТУ им. Н.Э. Баумана, 2020, вып. 19, 703 с.

[4] Ржевский С.В. *Математическое программирование*. Санкт-Петербург, Лань, 2022, 608 с.

[5] Рогова Н.В., Старожилова О.В. *Математическое программирование*. Самара, ПГУТИ, 2024, 177 с.

[6] Борознов В.О. Исследование решения задачи коммивояжера. *Вестник Астраханского государственного технического университета. Серия: Управление, вычислительная техника и информатика*, 2009, № 2, с. 147–151.

[7] Рахмангулов А.Н., Цыганов А.В., Пикалов В.А., Муравьев Д.С. *Математическое моделирование транспортных систем и процессов*. Магнитогорск, Изд-во Магнитогорского гос. техн. ун-та, 2021, 190 с.

[8] Перлюк В.В. *Методы оптимизации проектных решений*. Санкт-Петербург, ГУАП, 2023, 121 с.

[9] Гапанович В.С., Гапанович И.В. *Методы решения оптимизационных задач*. Тюмень, ТИУ, 2014, 272 с.

[10] *Поиск решения в Excel: пример использования функции для решения задачи с неизвестными параметрами.* URL: https://exceltut.ru/poisk-resheniya-v-excel-primer-ispolzovaniya-funktsii-dlya-resheniya-zadachi-s-neizvestnymi-parametrami/ (дата обращения 20.03.2025).

[11] *Функция Поиск решения в Excel. Включение, пример использования со скриншотами*. URL: https://office-guru.ru/excel/funkciya-poisk-resheniya-v-excel-vklyuchenie-primer-ispolzovaniya-so-skrinshotami.html (дата обращения 20.03.2025).

Поступила в редакцию 31.03.2025

 ISSN 2541-8009

Кирюхин Егор Александрович — студент кафедры «Системы автоматизированного проектирования», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Ссылку на эту статью просим оформлять следующим образом:
Кирюхин Е.А. Современные алгоритмы и цифровые инструменты для решения задач дискретной оптимизации. *Политехнический молодежный журнал*, 2025, № 05 (100). URL: https://ptsj.bmstu.ru/catalog/icec/inf_tech/1066.html

ISSN 2541-8009

MODERN ALGORITHMS AND DIGITAL TOOLS FOR SOLVING DISCRETE OPTIMIZATION PROBLEMS

E.A. Kiryukhin egorkiruhin5707@gmail.com

Bauman Moscow State Technical University, Moscow, Russian Federation

The article presents an analysis of the formulation of discrete optimization problems and methods for their solution. The traveling salesman problem was considered as an example. To formulate its condition, a number of definitions and classifications were introduced, and then its mathematical model was constructed. Using the example of this task, we compared two common ways of solving optimization problems using modern digital tools. As the first method, a solution using the Microsoft Excel add-in "Solution Search" is considered. As a second solution, we considered the development of a software product that implements one of the methods for solving discrete optimization problems, namely, the evolutionary strategy of a population algorithm. This method was also used in the first solution method. It should be borne in mind that the software implementation of such a solution can be implemented in any modern programming language. The article provides an example of a software solution developed in the C++ programming language. Using the software developed, the solution identified both correct routes with the same minimum length for the specified source data, in contrast to the solution using the "Solution Search" add-on, which, due to the specifics of its organization, provided only one correct solution. Based on the results of comparing the two solutions, the expediency of solving optimization problems was substantiated by dividing the conditions of the task into input and output data, followed by the creation of a mathematical model of the problem, in particular, setting the objective function and developing a universal software solution in any modern programming language.

Keywords: optimization problems, discrete optimization problems, traveling salesman problem, methods for solving optimization problems, population algorithm, Microsoft Excel add-in "Search for solutions", C++ programming language

Received 31.03.2025

Kiryukhin E.A. — Student of Department of Computer-Aided Design Systems, Bauman Moscow State Technical University, Moscow, Russian Federation.

Please cite this article in English as:
Kiryukhin E.A. Modern algorithms and digital tools for solving discrete optimization problems. *Politekhnicheskiy molodezhnyy zhurnal*, 2025, no. 05 (100). (In Russ.). URL: https://ptsj.bmstu.ru/catalog/icec/inf_tech/1066.html

 ISSN 2541-8009