СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Люди в реальном мире сталкиваются с переплетенными и связанными задачами каждый день. Некоторые исследователи утверждают, что существует разрыв между теорией и практикой, потому что ни теоретические задачи, ни критерии не отражают характеристик реального мира [1]. Поэтому, чтобы увеличить сложность тестовых данных, они масштабируются, чтобы приблизиться к реальному миру. Например, ученые исследовали известную задачу коммивояжера с 25 000 000 городов [2].

Тем не менее, очень часто делаются предположения и упрощения, когда реальные задачи сводятся к известным и хорошо изученным NP-сложным задачам оптимизации. По этой причине теоретиков обвиняют в том, что они решают «игрушечные», не приспособленные к реальным условиям задачи. Поэтому было исследовано множество новых проблем, с которыми сталкиваются люди и организации в реальной жизни, и, например, была предложена задача маршрутизации транспортных средств (VRP) [3]. Кроме того, существуют многочисленные вариации этой задачи, где добавляются либо новые ограничения, либо новые цели. Например, уже введены грузоподъемность ТС в качестве ограничения (Capacitated VRP) или удовлетворение потребителей в определенный промежуток времени в качестве дополнительной цели (VRP with Time Windows), а также множество других задач VRP с добавочными критериями. Варианты VRP уже можно рассматривать как начало для моделирования реальных задач проводить исследования, более приближенные к промышленным проблемам. Хотя эти задачи являются продолжением существующих, в реальном мире существуют случаи, когда некоторые из этих задач взаимозависимы. Из-за взаимозависимости решение каждой проблемы самостоятельно не всегда может привести к оптимальному решению. В целом, пока мало исследований по взаимосвязанным и взаимозависимым задачам. Чтобы научиться решать задачи с характеристиками и проблемами реального мира, в этой работе исследуется задача о воре (Travelling Thief Problem, TTP), в которой взаимодействуют известные задача коммивояжера (TSP) и задача о

рюкзаке (KP). Исследования задачи о воре дадут представление о том, как решать сложные взаимосвязанные задачи и, следовательно, как решать проблемы реального мира.

Задача о воре (Travelling Thief Problem, TTP) была предложена учеными Bonyadi, Michalewicz, и Barone в 2013 году. Они провели исследование, в рамках которого сравнили метаэвристики для решения некоторых NP-трудных проблем. Но они утверждают, что между реальными задачами и классическими существует большой разрыв. И его рост объясняется тем, что практические задачи, с которыми люди сталкиваются в мире, становятся все более сложными, в то время как классические проблемы за 50 лет остались прежними.

После анализа ряда реальных задач они выявили две их важные характеристики:

1) Комплексность – промышленные задачи обычно включают в себя несколько подзадач, каждая из которых имеет свои критерии оптимизации или ограничения.
2) Взаимозависимость – решение одной из подзадач влияет на ограничения и критерии оптимизации другой подзадачи.

За короткое время после публикации TTP были предложены различные алгоритмы ее решения. Но нет никаких оснований для использования тех или иных моделей решения задачи, проведены лишь только сравнения между различными алгоритмами. Поэтому я хотел бы заняться двумя вещами в своей диссертации: исследовать и понять существующие модели решения задачи и алгоритмы, а также попытаться улучшить их.

В главе 1 приводятся постановки подзадач, их связь с основной задачей, а также распространенные варианты их решения.

В главе 2 рассматривается сама задача и существующие алгоритмы ее решения. Проводится сравнение и анализ стратегий решения задачи данными алгоритмами, и обосновывается необходимость их оптимизации.

Глава 3 содержит предложения по оптимизации алгоритмов, их описание.

В главе 4 описываются тестовые данные, на которых проводится вычислительный эксперимент, экспериментальная установка и результаты вычислений.

ГЛАВА 1. ПОДЗАДАЧИ ЗАДАЧИ О ВОРЕ

Задача о воре содержит в себе две взаимосвязанные подзадачи – задачу коммивояжера и задачу о рюкзаке. Данные задачи являются очень известными задачами оптимизации и очень глубоко исследованы. В данной главе описаны наиболее важные результаты, полученные для решения данных задач.

1.1. ЗАДАЧА КОММИВОЯЖЕРА

Уже долгое время одной из самых популярных задач оптимизации является задача коммивояжера [4].

Содержательная постановка задачи. Имеется N городов, для каждой пары городов заданы расстояния между ними. Коммивояжер должен обойти все города ровно по одному разу так, чтобы суммарная длина маршрута была минимальной.

Формальная постановка задачи. Пусть $G(V, E)$ - полный ориентированный граф с заданным множеством вершин $V = \{1, \dots, N\}$ и множеством ребер E. Каждому ребру $(i, j) \in E$ сопоставлено число c_{ij} - расстояние между парой городов.

Пусть

$$x_{i,j} = \begin{cases} 1, \text{если из города } i \text{ движемся в город } j \\ 0, \text{в противном случае} \end{cases}, i = \overline{1, N}, j = \overline{1, N}$$

Требуется найти замкнутый маршрут (цикл) виде матрицы $X = (x_{ij}), i = \overline{1, N}$, $j = \overline{1, N}$, удовлетворяющий следующим ограничениям и критерию оптимальности:

1) Маршрут должен проходить через каждый город ровно один раз:

$$\sum_{i=1}^{N} x_{ij} = 1,\ j = \overline{1, N} \qquad (1.1)$$

2) Маршрут должен иметь минимальное суммарное пройденное расстояние или затраченное время:

$$\min \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij} \qquad (1.2)$$

1.2. Методы решения задачи коммивояжера

В данном разделе рассмотрены наиболее популярные подходы для решения задачи коммивояжера, которые также применимы для решения задачи о воре и используются в данной работе.

1.2.1. Точные алгоритмы

Можно выделить два наиболее популярных подхода для решения TSP точными алгоритмами – метод ветвей и границ и метод динамического программирования [5]. Главное достоинство данной группы алгоритмов – они гарантированно находят оптимальные решения. Однако, они неприменимы для решения задач с большим числом городов, так как имеют высокую вычислительную сложность.

Метод ветвей и границ (Branch and Bound). Метод ветвей и границ является наиболее известным и общепринятым точным алгоритмом для решения задачи коммивояжера. Основная идея данного метода состоит в том, чтобы разделить задачу на множество подзадач и отсекать те, что гарантированно не дадут оптимального решения.

Самой эффективной реализацией метода ветвей и границ для решения TSP является реализация Concorde [6]. Concorde – программное обеспечение, которое основано на исследовании Дэвида Эпплгейта и коллег [4]. Оно включает в себя различные методы, которые были придуманы за 60 лет исследования в области линейного программирования для решения TSP. Concorde смогло решать задачи с 85900 городами, например, задачу *pla85900* из TSPLIB [7]. Однако, на это ушло очень много времени и вычислительных ресурсов. Задачи с около 1000 городами данный алгоритм умеет решать за пару минут.

Главным недостатком данного алгоритма является необходимость полностью решать задачу (вычислять целевую функцию) на все области

допустимых решений. Поэтому поиск решения для задач с большим числом городов занимает очень много времени.

Метод динамического программирования (Dynamic Programming). Методом полного перебора задача TSP может быть решена за время $O(n!)$ но метод динамического программирования дает возможность уменьшить сложность решения задачи.

На данный момент наиболее известным алгоритмом, решающим задачу коммивояжера методом динамического программирования, является алгоритм Хелда-Карпа [8]. Его временная сложность – $O(n^2 2^n)$. Данный метод требует много памяти для хранения промежуточных результатов, что повышает скорость пересчета целевой функции. Однако, сложность все равно экспоненциальная, и поэтому данный алгоритм тратит много времени для решения задач даже с небольшим количеством городов.

1.2.2. Алгоритм Лина-Кернигана

Наибольший успех в решении практических задач был достигнут реализациями эвристического алгоритма Лина–Кернигана [9]. Эвристика Лина–Кернигана обобщает так называемый «2-OPT» поиск и является основой для многих других метаэвристик, решающих задачу TSP. Данный алгоритм работает следующим образом: на каждой итерации происходит поиск некоторой последовательности обмена дуг местами, которые суммарно улучшают маршрут. Если такая последовательность не найдена, то алгоритм останавливается, и выдается найденный локальный оптимум. При этом показано, что трудоемкость данного алгоритма равна $O(n^{2.2})$ [10].

Одной из довольно эффективных реализаций этой эвристики является реализация Хельсгауна, сокращенно LKH [11]. LKH имеет различные оптимизации, которые ускоряют алгоритм, а также некоторые существенные изменения, которые изменяют структуру алгоритма. Вместо использования обмена с 2-OPT в качестве базового перемещения, он использует обмен 5-opt, учитывая 10 ребер одновременно. Еще одно изменение заключается в том, что

LKH рассматривает только перспективные границы для обмена. Перспективными ребрами являются те, которые могут быть найдены в оптимальном решении. Он находит эти ребра с помощью меры, называемой α-мерой, которая определяется как расстояние до соответствующего минимального 1-дерева – остовного дерева с одним дополнительным ребром. Эмпирически доказано, что минимальное 1-дерево содержит 70–80% ребер оптимального обхода [11]. LKH установил рекорд для нескольких экземпляров из TSPLIB и уже нашел глобальный оптимум для задачи *pla85900*.

1.2.3. Эволюционные алгоритмы

Хотя раньше генетические алгоритмы для TSP не показывали хороших результатов в решении TSP [12], относительно новые генетические алгоритмы с кроссоверами EAX и GPX дают сравнительные результаты с эвристикой Лина–Кернигана и LKH [13].

Важным аспектом генетического алгоритма является оператор кроссовера. Найти хороший кроссовер для перестановок нетривиально. Например, традиционный 1-точечный оператор кроссовера не применим к задаче коммивояжера (рисунок 1.1).

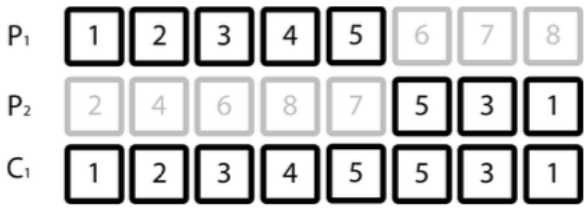

Рисунок 1.1 – Пример, где 1-точечный кроссовер неприменим для TSP

За последние 30 лет были предложены многочисленные операторы кроссовера для перестановок в задаче TSP. Различные кроссоверы фокусируются на разных аспектах проблемы. Одни основаны на абсолютных позициях городов, например, циклический (CX), позиционный (PX), частично сопоставляющий кроссовер (PMX) и в некоторой степени порядковый кроссовер (OX). Другие кроссоверы оценивают относительный порядок, например, максимально сохраняющий кроссовер (MPX) и порядковый (OX). Последняя группа

определяет информацию о смежности, это кроссовер рекомбинации дуг (ERX), разделяющий кроссовер (PX), обобщенное разделяющий кроссовер (GPX) и собирающий ребра кроссовер (EAX). Информация о смежности и относительный порядок городов являются важной информацией для TSP, в отличие от абсолютного положения города [14]. В общем случае для задачи TSP показано, что CX < PMX < OX < ERX [15].

Циклический оператор кроссовера (CX)

Циклический кроссовер (англ. Cycle Crossover) был предложен Оливером, Смитом и Холландом [16]. Он работает следующим образом: сначала циклы строятся для обоих родителей. Чтобы создать цикл, нужно начать со случайного города первой родительской особи P_1, которого в настоящее время нет ни в одном цикле. Следующий город выбирается так, чтобы быть в положении C во второй родительской особи P_2. Это повторяется, пока не встретится начальный город, создавая цикл. Город в каждой позиции потомка, получаемого кроссовером CX, соответствует либо P_1, либо P_2. Циклический кроссовер соответствует выполнению равномерного кроссовера по циклам. Пересечение цикла только случайным образом сохраняет абсолютное положение двух родителей, относительный порядок только в пределах цикла и информацию о смежности. Пример работы CX показан на рисунке 1.2.

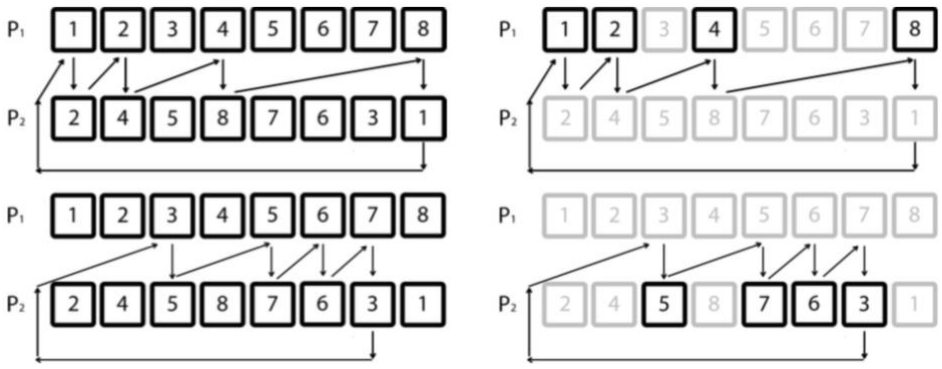

Рисунок 1.2 – Пример двух циклов (слева) и работы CX (справа)

Порядковый оператор кроссовера (OX)

Порядковый кроссовер (англ. Ordered Crossover) был впервые представлен Дэвисом [17]. Это работает следующим образом: сначала отрезок копируется от одного родителя к

потомку. Этот отрезок определяется как города между двумя случайно определенными точками пересечения. Затем остальные города добавляются после второй точки пересечения. Они добавляются в порядке их появления в другой родительском особи, начиная со второй точки кроссовера.

Порядковый кроссовер сохраняет информацию об абсолютном положении, относительном порядке и смежности скопированного отрезка, но для остальных городов запоминает только относительный порядок.

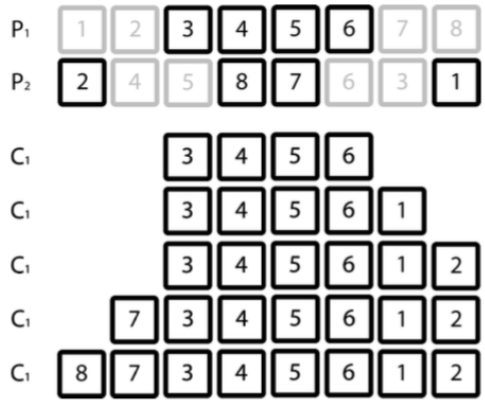

Рисунок 1.3 – Пример работы порядкового кроссовера

Частично отображающий оператор кроссовера (PMX)

Частично отображающий оператор кроссовера был представлен Голдбергом, Линглом и их коллегами [18]. Он работает следующим образом: сначала о случайный отрезок копируется из одного родителя (так же, как в OX). Затем оставшиеся позиции копируются с другого родителя. Это приводит к дублированию городов. Чтобы исправить это, используется отображение между узлами родителей. Если город все еще повторяется, то сопоставление применяется снова, пока город не станет уникальным в туре.

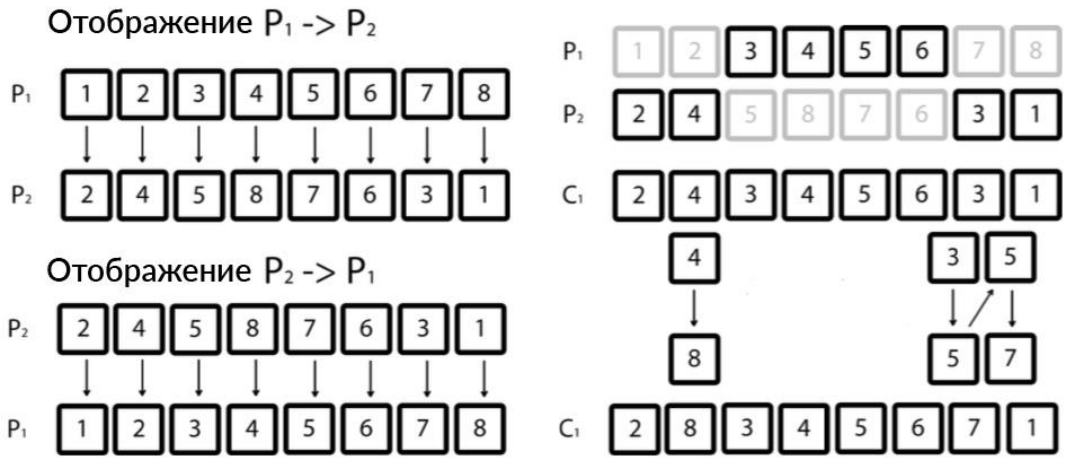

Рисунок 1.4 – Пример отображения между родителями (слева) и работы PMX (справа)

Максимально сохраняющий оператор кроссовера (MPX)

Максимально сохраняющий оператор кроссовера (англ. Maximal Preservation Crossover) был введен ученым Muhlenbein [19] и работает следующим образом: сначала случайный отрезок копируется от одного родителя к началу маршрута. После копирования отрезка города добавляются от другого родителя в порядке их появления. Отрезок изначально определяется как города между двумя случайно определенными точками пересечения с ограничением, что его длина больше 10, но меньше n [14]. Другое исследование показало, что использование фиксированной длины 1/3 дало лучшие результаты [20]. Информация об относительном положении и смежности одного из родителей сохраняется, в то время как только относительный порядок между городами другого родителя наследуется. MPX имеет хорошие результаты по сравнению с другими классическими кроссоверами для перестановок в сочетании с локальным поиском [21].

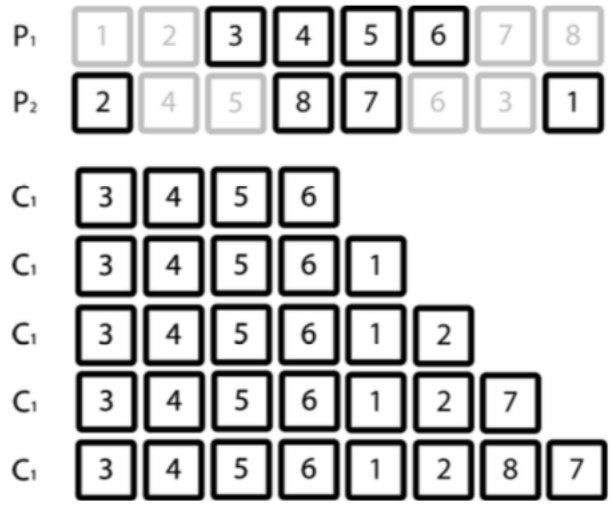

Рисунок 1.5 – Пример работы максимально сохраняющего кроссовера.

Кроссовер рекомбинации дуг (ERX)

Кроссовер рекомбинации (перестановки) дуг был введен Уитли и Старквезером [22]. Он основывается на перемещении общих дуг родителей к потомку. Для каждой из родительских особей составляется список смежности, а затем эти списки объединяются. В итоге выбирается та дуга, которая чаще всего встречалась в родительских особях. Если таковых нет – выбирается случайная.

Данный способ удобен тем, что в качестве родителей для новой особи может выступать любое количество особей.

Кроссовер рекомбинации дуг сохраняет информацию о смежности обоих родителей, но относительный порядок может быть нарушен, поскольку ERX может изменить направление.

1.3. Задача о рюкзаке

В данной задаче необходимо найти набор предметов с наибольшей суммарной стоимостью так, чтобы они поместились в рюкзак. Более формальная классическая постановка описана ниже.

Имеется M предметов, каждый из которых имеет вес w_i и стоимость p_i, а грузоподъемность рюкзака W.

Пусть

$$x_i = \begin{cases} 1, \text{если предмет включен в набор} \\ 0, \text{в противном случае} \end{cases}, i = \overline{1, M}$$

Требуется найти бинарный вектор (x_i), $i = \overline{1, M}$, удовлетворяющий следующим ограничениям и критерию оптимальности:

1о. Суммарный вес предметов, находящихся в рюкзаке, не должен превышать его грузоподъемность:

$$\sum_{i=1}^{M} x_i w_i \leq W \tag{1.3}$$

2о. Общая стоимость предметов в рюкзаке должна быть максимальна:

$$\max \sum_{i=1}^{M} x_i p_i \tag{1.4}$$

1.4. Методы решения задачи о рюкзаке

Задача о рюкзаке относится к классу NP-полных, и для нее нет полиномиального алгоритма, решающего ее за разумное время. Поэтому для решения задачи существуют два типа алгоритмов – точные и приближенные. Каждые из них имеют свои плюсы и минусы, которые будут рассмотрены ниже.

1.4.1. Полный перебор

Метод полного перебора применим к данной задаче так же, как и ко многим другим оптимизационным задачам. Пусть дано N предметов, которыми можно заполнять рюкзак. Необходимо собрать предметы таким образом, что их суммарная стоимость была максимальна, а вес бы не превышал W.

Очевидно, что для каждого предмета можно выделить 2 состояния: предмет можно либо положить в рюкзак, либо нет. Тогда всего существует 2^N вариантов упаковки рюкзака, и соответственно, временная сложность алгоритма составляет $O(2^N)$. Таким образом, экспоненциальная сложность алгоритма не позволяет решать задачи с большим количеством предметов за приемлемое время.

На рисунке 1.6 отображено дерево перебора для задачи с тремя предметами. Каждый лист означает некоторое подмножество предметов. Чтобы

найти решения задачи, нужно по данному дереву выбрать среди листов с весом, меньшим W, тот, что имеет максимальную стоимость [23].

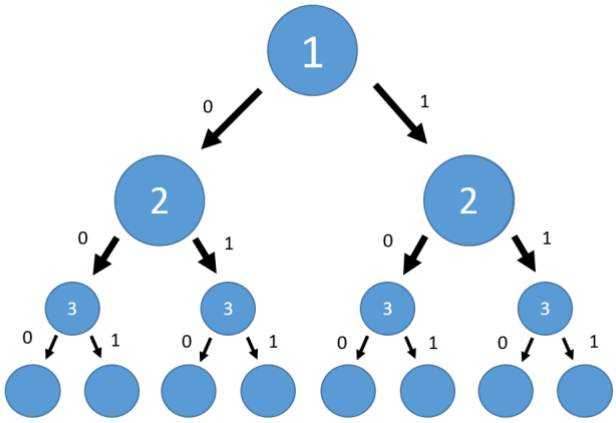

Рисунок 1.6 – Пример дерева полного перебора для трех предметов

1.4.2. Динамическое программирование

Этот метод заключается в следующем: изначально задача сильно упрощается, для нашей задачи, например, возьмем рюкзак емкостью 1 и всего 1 вещь. После чего будем увеличивать емкость рюкзака и смотреть какую максимальную цену мы можем получить. Потом добавляем 2 вещь и проделываем то же самое со 2-й вещью, учитывая результаты с первым грузом.

Сложность данного решения $O(NW)$ [24]. Метод динамического программирования все равно не позволяет решать задачу за полиномиальное время, потому что задача о рюкзаке — одна из NP-полных задач комбинаторной оптимизации.

1.4.3. Алгоритм Combo

Наилучшим алгоритмом для решения задачи о рюкзаке является алгоритм Combo [25]. В нем используются алгоритмы, которые рассматривают только интересные элементы, называемые ядром. Ядро определяется как набор элементов вокруг элемента разрыва. Все предметы с соотношением вес/стоимость выше всех основных предметов включены в рюкзак, и все предметы с соотношением меньше всех основных предметов исключены. Если ядро выбрано правильно, это решает проблему до оптимальности, а если нет, то

оно может использоваться в качестве верхней границы. Выбор правильного ядра может быть трудным. Дэвид Пизингер в алгоритме Combo решает эту проблему, находя правильное ядро на лету, итеративно добавляя элементы в ядро или удаляя их из ядра. Само ядро ищется либо с помощью целочисленного линейного программирования, либо с помощью динамического программирования.

Выводы по главе 1

Задача коммивояжера и задача о рюкзаке очень популярны и для их решения предложено множество алгоритмов. Данные задачи принадлежат классу NP-полных задач, и поэтому алгоритмы не умеют находить оптимальное решение за полиномиальное время. В связи с этим было предложено множество приближенных алгоритмов для решения данных задач.

Однако существующие методы решения подзадач не всегда применимы к решению задачи о воре: их сложность значительно возрастает при пересчете целевой функции оптимизации. Поэтому необходимо учитывать специфику задачи о воре для того, чтобы грамотно применять известные методы для решения подзадач TSP и KP. Так, например, относительный порядок и абсолютная позиция города в маршруте могут не играть роли в TSP, но данные критерии очень важны в TTP.

ГЛАВА 2. ЗАДАЧА О ВОРЕ И МЕТОДЫ ЕЕ РЕШЕНИЯ

В данной главе рассмотрены содержательная и математическая постановки задачи о воре. За пять лет предложено множество способов решения данной задачи, но в данной главе рассмотрены наиболее важные алгоритмы и подходы для решения задачи о воре.

2.1. Содержательная постановка задачи о воре

Имеется n городов (заданы попарные расстояния между ними d_{ij}). Также имеется m предметов; для каждого предмета заданы его стоимость p_k и вес w_k. Для каждого предмета указано, в каких городах он находится, а в каких - нет.

У вора имеется рюкзак с ограниченной грузоподъемностью W.

Требуется обойти все города по одному разу, украв в них предметы таким образом, чтобы суммарный вес предметов не превышал W, и при этом оптимизировать целевую функцию G.

Создатели задачи в своей статье предложили различные критерии оптимизации, поэтому представлено две вариации задачи о воре: TTP_1 и TTP_2 [1].

2.1.1. Вариация TTP_1

Требуется максимизировать общий доход, который состоит из стоимости украденных предметов за вычетом стоимости аренды рюкзака. Стоимость аренды рюкзака зависит от времени прохождения маршрута t и имеет определенную цену R за единицу времени. Время маршрута вычисляется как сумма расстояний переходов от одного города в другой, поделенных на скорость на данном участке маршрута. Скорость вора при передвижении от одного города к другому зависит от суммарного веса рюкзака на данный момент:

$v_c = v_{max} - W_c \frac{v_{\max} - v_{\min}}{W}$, где

W_c – текущий вес рюкзака,

$v_{\max}$ – максимальная скорость вора,

$v_{\min}$ – минимальная скорость вора.

2.1.2. Вариация TTP2

Требуется одновременно максимизировать общий доход и минимизировать время прохождения по маршруту.

Текущая скорость вычисляется так же, как в условиях вариации задачи TTP1. Однако стоимость украденного предмета уменьшается с течением времени. Скорость снижения стоимость определяется как $Dr^{\left\lceil \frac{T_i}{C} \right\rceil}$, где T_i – время с момента похищения предмета до окончания маршрута, а C – константа. Таким образом, стоимость украденного предмета в конце маршрута вычисляется как $p_i \times Dr^{\left\lceil \frac{T_i}{C} \right\rceil}$.

Наибольшую популярность в литературе и исследованиях получила постановка задачи TTP1, и в моей работе будет исследоваться именно она.

2.2. Математическая постановка задачи о воре TTP1

Дано:

- Доступность предмета I_i в каждом городе: $A_i \subseteq \{1, \dots, n\}$;
- Скорость вора: $v_c = v_{\max} - W_c \frac{v_{\max} - v_{\min}}{W}$, где W_c – текущий вес рюкзака;
- Стоимость аренды рюкзака: $\$R$ за ед. времени.

Найти:

- $\overline{\Pi} = (\pi_1, \dots, \pi_n)$ – маршрут передвижения вора;
- $\bar{P} = (p_1, \dots, p_m)$, $p_i \in \{0 \cup A_i\}$ – план упаковки предметов, где z_i показывает, из какого города предмет I_i должен быть «украден».

Критерий оптимизации:

$$\max Z(\Pi, P) = profit(P) - R \cdot time(\Pi, P), \text{ где}$$
$$profit(\text{P}) = \sum_{i=1}^{m} p_i,$$
$$time(\Pi, P) = \sum_{i=1}^{n} \frac{d(\pi_i, \pi_{i+1})}{v_{\max} - W_c \frac{v_{\max} - v_{\min}}{W}} \quad (2.1)$$

2.3. Существующие алгоритмы решения задачи о воре

С момента создания задачи о воре было предложено множество алгоритмов для ее решения. В данном разделе упомянуты наиболее популярные из них и описаны по мере появления в литературе.

2.3.1. Процедуры локального поиска

Прежде, чем перейти к описанию самих алгоритмов, я хочу кратко описать процедуры локального поиска, которые часто используются в нижеописанных алгоритмах: 2-OPT, Insertion, BitFlip, Exchange.

2-OPT. Данная процедура может быть применена к любому виду перестановки. Она разворачивает определенный отрезок перестановки. Если рассматривать процедуру в рамках маршрута в TSP – создаются и удаляются две дуги.

Всего существует $\frac{n(n-1)}{2}$ возможных применений данной процедуры.

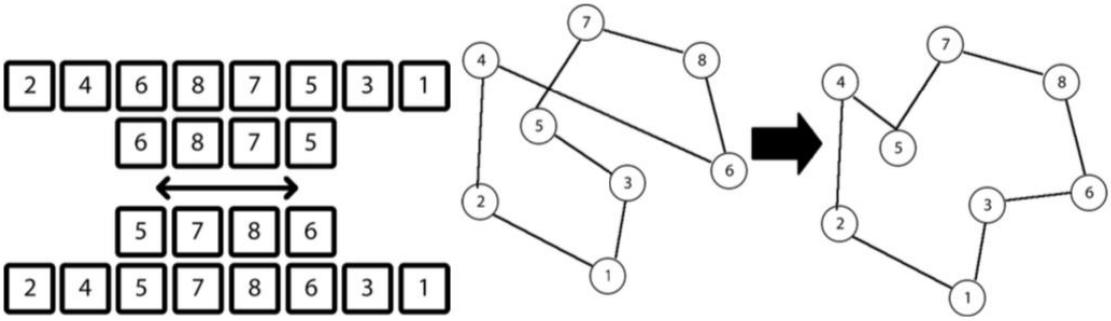

Рисунок 2.1 – Процедура 2-OPT для перестановки (слева) и соответствующего маршрута (справа)

Insertion. В рамках данной процедуры выбирается случайный элемент перестановки, удаляется из нее и вставляется в другую позицию. Эта процедура также применяется в подзадаче TSP.

Всего существует $\frac{n(n-1)}{2}$ возможных применений данной процедуры.

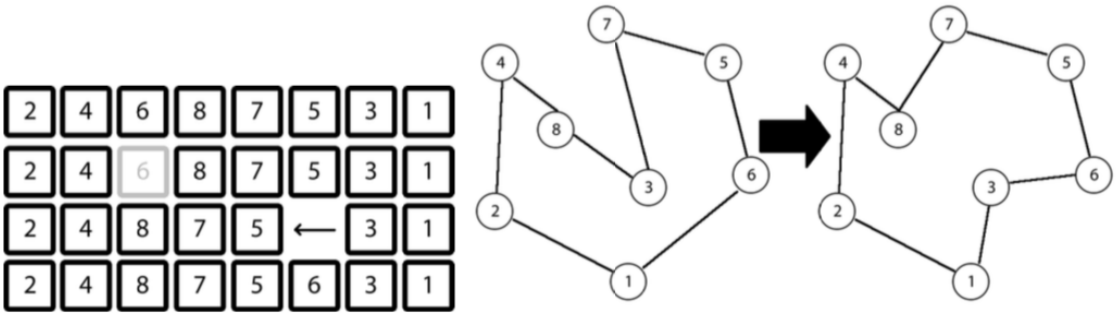

Рисунок 2.2 – Процедура Insertion для перестановки (слева) и соответствующего маршрута (справа)

BitFlip. Случайным образом выбирает один элемент бинарного вектора и инвертирует его. В терминах плана упаковки выполнение процедуры включение или исключение предмета из рюкзака.

Для вектора длины n всего существует n возможных применений процедуры.

Рисунок 2.3 – Процедура BitFlip

Exchange. В рамках данной процедуры выбираются два случайных элемента бинарного вектора с различными значениями и их значения меняются на противоположные. В терминах плана упаковки это означает, что один предмет вынимается из рюкзака, а другой добавляется.

Для вектора длины n всего существует $\frac{n(n-1)}{2}$ возможных применений процедуры.

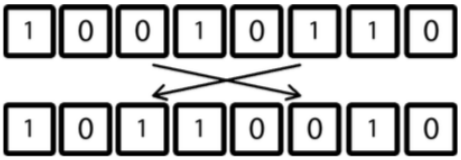

Рисунок 2.4 – Процедура Exchange

2.3.2. SH, RLS, (1+1) EA

В статье [26] описаны три эвристических алгоритма. В каждом из них способ обхода городов вычисляется алгоритмом Лина–Кернигана для задачи коммивояжера, так как он достаточно быстро вычисляет хороший путь. Ниже представлены описания данных алгоритмов:

Simple Heuristic (SH). Для каждого предмета, украденного из определенного города, ищется оценка $score$, показывающая общую выгоду, если вор украдет только данный предмет в данном городе, и фитнесс-оценка u. Затем полученный массив оценок сортируется по убыванию $score$, и проходом по массиву кладем предмет в рюкзак, если его фитнесс-оценка $u > 0$, до тех пор, пока рюкзак не будет наполнен. Затем остается лишь сравнить полученную полезность с полезностью, которая была бы, если бы не брали ни один предмет.

Random Local Search (RLS). Алгоритм случайного локального поиска итеративно модифицирует план упаковки. На каждой итерации изменяется статус одного случайно выбранного предмета (под статусом понимается наличие предмета в плане упаковки), затем вычисляется целевая функция и проверяется, улучшилось ли найденное решение по сравнению с предыдущими; если лучше – то результат обновляется. Критерий остановки – заранее заданное количество итераций без улучшений.

(1+1) EA. Эволюционный алгоритм (1+1) аналогичен алгоритму случайного поиска, только на каждой итерации изменяется статус каждого предмета с заданной вероятностью $\frac{1}{m}$.

2.3.3. DH и CoSolver

M. R. Bonyadi с коллегами представили два алгоритма для решения задачи о воре: Density Heuristic (DH) и CoSolver [27].

Density Heuristic (DH). Этот алгоритм является жадным для процесса упаковки предметов. Общая схема алгоритма аналогична Simple Heuristic (SH). Разница лишь в том, что DH вычисляет, увеличилась ли целевая функция при добавлении предмета в текущий план упаковки, а SH основан на вычислении

прироста при добавлении предмета в рюкзак по сравнению с пустым планом упаковки.

CoSolver. Главная идея данного алгоритма – разделить задачу о воре на подзадачи и решать каждую по-отдельности с некоторой связью между ними. А в конце решения подзадач объединяются в общее решение задачи о воре.

В подзадаче TSP зафиксированы предметы, которые нужно положить в рюкзак, а минимизируется общее время в пути. Эта подзадача решается точным алгоритмом.

А в подзадаче KP зафиксирован маршрут обхода городов, а необходимо найти такой план загрузки рюкзака, чтобы суммарная ценность предметов была максимальна. Однако в данной подзадаче используются «облегченные» ценности предметов $\overline{p_i} = p_i - R(t - t')$, где t – время до конца тура, если брать с собой предмет, а t' – время до конца тура, если не брать предмет. Данная подзадача решается методом динамического программирования.

Данный алгоритм решает задачу лучше, чем DH, однако сравнение проводилось только на задачах с числом городов $n \leq 25$, так как подзадача TSP решается точным алгоритмом.

2.3.4. MATLS

Mei, Li и Yao [28] также изучали взаимозависимость подзадач. Они проанализировали математическую постановку задачи и показали, что она не разделима. Поскольку цели подзадач TSP и KP не полностью взаимосвязаны, нельзя ожидать достижения конкурентоспособных результатов путем решения каждой подзадачи по-отдельности. Авторы использовали два отдельных подхода для решения TTP: подход, основанный на совместной коэволюции, похожий на CoSolver, и меметический алгоритм под названием Memetic Algorithm with a Two-stage Local Search (MATLS), который решает задачу в целом [28].

В данном алгоритме сначала популяция генерируется случайным образом: маршрут обхода городов вычисляется алгоритмом Лина–Кернигана или эвристикой минимального остовного дерева, а план упаковки – жадной

эвристикой, аналогичной SH. Отличие от SH в фитнесс-оценке u : она аппроксимируется между худшим случаем – когда все предметы положены в рюкзак до города, в котором находится текущий предмет – и ожидаемым случаем – когда все ранее собранные предметы подобраны в течение маршрута с равномерным распределением.

Далее на каждой итерации для двух случайных родительских особей применяется кроссовер порядка OX (Ordered crossover) для получения маршрута потомка, а затем ищется его улучшение при помощи процедуры 2-OPT (причем значение целевой функции пересчитывается за константное время, в отличие от SH, где это занимает $O(n+m)$). План упаковки предметов для него генерируется так же, как описано было описано ранее.

Данный алгоритм по сравнению с RLS и (1+1) EA показал лучшие результаты.

2.3.5. S1-S5, C1-C6

В статье [29] для описания работы алгоритмов сначала описана процедура упаковки *«PackIterative»* – она вычисляет план упаковки рюкзака на основе оценок для каждого предмета, украденного в заданном городе, получаемых следующим образом: $p_{ik}^{\alpha}/(\,w_{ik}^{\alpha} \times d_i)$, где p_{ik} и w_{ik} – ценность и вес i -го предмета в k -ом городе, а d_i – расстояние от i-го города до конца маршрута. Экспонента α используется для увеличения значимости определенных параметров в формуле и подбирается итеративно.

Также здесь используются процедуры *«Bitflip»* и *«Insertion»*, которые описаны выше.

Далее предложены простые алгоритмы локального поиска, в которых изначально вычисляется путь обхода городов (решение TSP) по алгоритму Лина–Кернигана, а затем решение ищется различными способами:

- S1: запустить «*PackIterative*».
- S2: запустить «*PackIterative*», затем повторять «*Bitflip*», пока алгоритм не сойдется.

- S3: запустить « *PackIterative* », затем (1+1) EA.
- S4: запустить «*PackIterative*», затем повторять «*Insertion*», пока алгоритм не сойдется.
- S5: запустить S1, пока не выйдет время.

Также предложены более сложные комбинации вышеописанных процедур:

- C1: запустить алгоритм Лина–Кернигана, «*PackIterative*», затем повторять группу процедур «*Bitflip*» и «Insertion», пока алгоритм не сойдется.
- C2: запустить алгоритм Лина–Кернигана, «*PackIterative*», затем повторять группу процедур «*Bitflip*», проход (1+1) EA и «*Insertion*», пока алгоритм не сойдется.
- C3: запустить алгоритм Лина–Кернигана на 10% времени, «*PackIterative*», выбрать лучший из полученный результатов, и повторять группу процедур «*Bitflip*» и «*Insertion*».
- C4: запустить алгоритм Лина–Кернигана на 10% времени, «*PackIterative*», выбрать лучший из полученный результатов, и повторять группу процедур «*Bitflip*», проход (1+1) EA и «*Insertion*».
- C5: повторять C1, пока не выйдет время.
- C6: повторять C2, пока не выйдет время.

Наилучшим образом на большом наборе задач показала себя вариация S5.

Стоит также отметить, что процедура «*PackIterative*» для генерации плана упаковки также используется во многих других алгоритмах, предложенных позднее.

2.3.6. MA2B, CS2SA

В другой статье [30] El Yafrani и Ahiod представили два алгоритма: меметический алгоритм MA2B, и CS2SA, который вдохновлен фреймворком CoSolver и использует алгоритм имитации отжига.

CS2SA работает следующим образом: во-первых, он инициализирует маршрут, используя алгоритм Лина–Кернигана, и создает план упаковки, используя жадную эвристику поиска плана упаковки, аналогичную используемой в MATLS. Затем он продолжает пытаться улучшить маршрут при помощи процедуры 2-OPT с уменьшенной окрестностью, полученной при помощи триангуляции Делоне. После этого он пытается улучшить план упаковки с помощью алгоритма имитации отжига. Этот процесс улучшения маршрута и плана упаковки повторяется до тех пор, пока не будет сделано ни одно улучшение. Они обнаружили, что CS2SA конкурирует с S5 и MATLS и работает лучше в случаях с рюкзаком большой вместимости.

Для алгоритма MA2B изначально определена процедура «ограниченного» локального поиска, которая заключается в следующем: сначала оптимизируется маршрут при помощи процедуры 2-OPT, затем план упаковки улучшается при помощи процедуры BitFlip. Данный локальный поиск ограничен малым количеством итераций – 50 для инициализации начальной популяции, и 10 – во время поиска решения.

В MA2B начальная популяция инициализируется следующим образом: сначала создается маршрут при помощи алгоритма Лина–Кернигана, и создает план упаковки, используя жадную эвристику поиска плана упаковки, аналогичную используемой в MATLS. Затем применяется алгоритм локального поиска, ограниченный 50 итерациями. В качестве кроссовера для маршрута используется MPX, а план упаковки устанавливается следующим образом: статус предмета берется от того родителя, от которого унаследовался город в маршруте. Мутация происходит только в том случае, если в популяции уже есть аналогичная особь – генерируется новый план упаковки при помощи жадного алгоритма, аз затем выполняется локальный поиск, ограниченный 10 итерациями. Критерии остановки – число итераций 10_6, число итераций без улучшений 10_4, время работы алгоритма 10 минут.

Размер популяции ограничен 40 особями, на каждой итерации 75% особей заменяются новыми.

Они также провели сравнение с алгоритмами S5 и MATLS. MA2B показал лучшие результаты на задачах меньшей размерности, а S5 и CS2SA решают лучше задачи большей размерности. В MATLS большую часть времени занимает инициализация популяции.

Чуть позже авторы данных алгоритмов провели более глубокое тестирование своих алгоритмов, и представили новую модификации алгоритма CS2SA – CS2SA-R [31]. Он устроен так, что он перезапускает CS2SA в течение всего предоставляемого лимита времени. Также они провели некоторую настройку внутренних параметров алгоритма, и этот алгоритм в некоторых задачах небольшой размерности показал лучшие результаты.

2.3.7. MMAS

Wagner предложил алгоритм MAX-MIN ant colony optimization (MMAS) [32], использующий муравьиный алгоритм. Пошагово он выглядит следующим образом:

1. Построить маршрут, используя муравьев.
2. Запустить процедуры локального поиска маршрута: 2-OPT, BitFlip или 3-OPT.
3. Для каждого маршрута сгенерировать план упаковки при помощи процедуры *«PackIterative»*.
4. Произвести локальный поиск маршрута: сначала (1+1) EA, затем один запуск Insertion и один запуск *«BitFlip»*.
5. Обновить феромонную тропу.

Автор показал, что данный алгоритм находит более длинные маршруты по сравнению с алгоритмами S1-S5 и C1-C6, но при этом с лучшими значениями целевой функции задачи.

Выводы по главе 2

В некоторых работах утверждается, что один из их алгоритмов работает лучше, поскольку он рассматривает взаимозависимость или специфичные для TTP маршруты больше, чем другие алгоритмы. CoSolver лучше, чем DH, MA

лучше, чем CA, а MMAS работает лучше, чем S1-S5, C1-C6 и MATLS в некоторых случаях. Авторы этих работ утверждают, что их алгоритмы лучше других, потому что они учитывают важные аспекты TTP, в то время как другие этого не делают. Эти утверждения, однако, подтверждаются только их хорошими результатами по сравнению с другими, а не какими-либо другими аналитическими выводами или эмпирическими исследованиями.

Что значит для алгоритма учитывать взаимозависимость подзадач TTP? Проще говоря, взаимозависимость заключается в том, что если вы выберете определенные предметы, то города, содержащие эти предметы, в свою очередь, скорее всего, будут в конце тура. Если у вас есть города в конце тура, то предметы, принадлежащие этим городам, с большей вероятностью будут включены в план упаковки, поскольку время в пути до конца тура короче. Но все же нет четкого понимания, какие правила применять для учета взаимозависимости подзадач.

В результате полного сравнения алгоритмов выявлено, что CS2SA, MATLS и MMAS, очевидно, решают задачу лучше, и, следовательно, наиболее применимы для решения задачи о воре [33]. Однако причина этих результатов на самом деле существенно не обоснована – каждый автор объясняет свои результаты рассмотрением взаимозависимости подзадач TTP и результатами сравнения алгоритмов.

ГЛАВА 3. РАЗРАБОТКА ЭВОЛЮЦИОННОГО АЛГОРИТМА ДЛЯ РЕШЕНИЯ ЗАДАЧИ О ВОРЕ

В данной главе описывается предлагаемый эволюционный алгоритм для решения задачи о воре, а также некоторые особенности ее реализации для оптимизации производимых вычислений.

3.1. Общая схема алгоритма

При выборе типа алгоритма я опирался на сложность вычислений и применимость существующих решений для подзадач TSP и KP.

Первое, на что стоит обратить внимание – сложность кодирования решения задачи и вычисления целевой функции. Для того, чтобы записать решения задачи, необходимо учесть, какой предмет вор должен украсть, а какой – нет. Если n – количество городов, а m – количество предметов, то размер памяти, требуемой для кодирования одного решения, равен $O(n+m)$. Соответственно, сложность вычисления фитнесс-функции также пропорциональна $O(n+m)$ – необходимо пройти по всем городам, посчитать суммарный вес предметов в каждом городе, рассчитать текущую скорость и время в пути до следующего города. Это довольно дорогая операция, поэтому необходимо стремиться не вычислять данную целевую функцию лишний раз.

Также стоит учесть взаимозависимость решения подзадач на итоговое решение задачи. Так, например, небольшое изменение маршрута, которое является локальным улучшением решения задачи TSP, может значительно повлиять на значение целевой функции TTP. Аналогично и в задаче KP: если добавить в рюкзак предмет в начале пути, то данное изменение может довольно сильно повлиять на конечный результат TTP, хотя для задачи KP данное изменение является локальным.

Однако пространство поиска решений в данной задаче очень большое – $n! \cdot 2^m$, поэтому выбранной стратегией довольно сложно искать поиск лучшего решения. Поэтому для ускорения поиска было решено ввести процедуры

локального улучшения конечного решения, когда решение одной из подзадач остается неизменным.

Как говорилось ранее, на данный момент не существует наилучшего алгоритма для решения задачи о воре, поэтому в качестве базового алгоритма выбран алгоритм CS2SA и его модификация с множественным запуском CS2SA-R, так как это один из последних предложенных алгоритмов, показывающий сравнимые и даже лучшие результаты с популярными S5 и MATLS [30, 31].

Мною же предлагается построить меметический алгоритм на основе CS2SA, добавив к нему операторы локального поиска. Итоговое формальное описание предлагаемого алгоритма представлено в листинге 3.1.

Листинг 3.1 – Общая схема предлагаемого алгоритма

```
   Результат: П - маршрут, P - план упаковки
 1 Сгенерировать начальное решение (П, P);
 2 до тех пор, пока есть улучшение выполнять
 3 |  (П*, P*) := CS2SA(П, P);
 4 |  если Z(П*, P*) > Z(П, P) тогда
 5 |  |  П := П*;
 6 |  |  P := P*;
 7 |  иначе
 8 |  |  Получить П** применением процедуры локального поиска,
   |  |   изменяя П* при неизменном P*;
 9 |  |  Получить P** применением процедуры локального поиска,
   |  |   изменяя P* при неизменном П**;
10 |  |  если Z(П**, P**) > Z(П, P) тогда
11 |  |  |  П := П**;
12 |  |  |  P := P**;
13 |  |  конец
14 |  конец
15 конец
```

Более подробно каждая часть алгоритма будет рассмотрена далее.

3.2. Процедуры локального поиска

Как говорилось выше, в большом пространстве поиска хотелось бы быстрее приближаться к оптимуму. Поэтому предлагаются процедуры локального улучшения и маршрута, и плана упаковки рюкзака.

3.2.1. Процедура локального поиска маршрута Optimal Subtour Search

Для локального улучшения маршрута на каждой итерации алгоритма можно проводить следующую операцию: брать случайный отрезок маршрута небольшой длины (например, из 5-ти городов), делать всевозможные перестановки и пересчитывать целевую функцию. В наивной реализации это займет $O(k! \cdot m)$, где k – длина выбранного отрезка.

Однако здесь можно учесть тот факт, что при неизменном плане упаковки рюкзака стоимость целевой функции можно пересчитывать быстрее, аналогично тому, как это предлагалось делать в процедуре *«Insertion»*. Случайны выберем отрезок маршрута длины k. Заранее найдем все суммарные и накопленные суммы весов предметов в каждом городе. Затем будем менять города в отрезке всеми возможными способами и запоминать наилучшую перестановку. Так как план упаковки не меняется, а маршрут будем менять только на определенном отрезке, то на каждой итерации можно пересчитать разницу целевой функции только на измененном отрезке маршрута (формула 3.1). Стоимость пересчета пропорциональна длине отрезка, поэтому суммарно сложность данной операции можно уменьшить до $O(k! \cdot k)$.

Формально данная процедура отображена в листинге 3.2.

Листинг 3.2 – Процедура локального поиска *«Optimal Subtour Search»*

Входные данные: Π - маршрут, P - план упаковки, k - длина отрезка

Предпосчитать суммарные веса в каждом городе $w[]$ и накопленные веса $w_{acc}[]$;
цикл $left := 2; left < length(\Pi); left := left + k$ **выполнять**
    $right := min(left + k; length(\Pi))$;
    $segment := [\Pi_{left}, ..., \Pi_{right}]$;
    $Z_{oldSeg} :=$ стоимость отрезка $segment$;
    $bestSegment := segment$;
    $Z := Z(\Pi, P); Z_{best} := Z$;
    до тех пор, пока *существует следующая перестановка segment* **выполнять**
        $segment :=$ следующая перестановка $segment$;
        $Z_{newSeg} :=$ стоимость отрезка $segment$;
        $Z := Z - Z_{oldSeg} + Z_{newSeg}$;
        если $Z_{newSeg} > Z_{oldSeg}$ **тогда**
            $Z_{best} := Z - Z_{oldSeg} + Z_{newSeg}$;
            $bestSegment := segment$;
        конец
    конец
    $[\Pi_{left}, ..., \Pi_{right}] := bestSegment$;
конец
Выходные данные: Π, P, Z

3.2.2. Процедура локального поиска маршрута Insertion

Как уже было сказано ранее, любое изменение маршрута может привести к сильному изменению целевой функции, что плохо скажется на поиске лучшего решения.

В качестве операторов мутации маршрута в алгоритме CS2SA используется процедура «*2-OPT*», которая была описана в главе 2.3. Учитывая, что данная процедура разворачивает случайную часть маршрута, это приводит к довольно сильным изменениям, так как целевая функция задачи TTP очень сильно привязана к последовательности городов в маршруте. По этой причине процедура *«Insertion»* выглядит более локальной, и именно она была взята в качестве основы для предлагаемого оператора локального поиска маршрута.

Хочется каждый город в текущем маршруте поставить в наилучшую позицию – так, чтобы целевая функция задачи стала больше. Количество возможных результатов применения *«Insertion»* равно n^2, так как каждый из n городов может быть перемещен в любую из n позиций. Если для каждого возможного результата пересчитывать целевую функцию (ее сложность – $O(n+m)$), то сложность алгоритма будет довольно большой – $O(n^2(n+m))$.

Стоит обратить внимание на вычислительную сложность пересчета целевой функции после применения выбранного оператора мутации. Ее вычисление может быть ускорено следующим образом: для каждого города можем предподсчитать суммарные накопленные стоимость и вес украденных вещей. Это сократит сложность подсчета с $O(n+m)$ до $O(n)$. Но мы можем сократить сложность вычисления фитнесс-функции еще больше, применяя последовательные инкрементальные вычисления целевой функции. Значение фитнесс-функции для любой возможной операции вставки может быть эффективно вычислен для каждого города из текущего маршрута следующим способом:

1. Поместить выбранный город в конец маршрута;
2. Посчитать значение целевой функции за $O(n)$;
3. Поменять выбранный город с предшествующим (последующим) ему городом и пересчитать значение функции за $O(1)$;
4. Повторить шаг 3, пока невыбранный город не попадет в необходимую позицию – $O(n)$.

Шаг 3 данном случае занимает время $O(1)$ только потому, что изменяется маленькая часть маршрута – всего 3 дуги. Обмен местами двух соседних городов влияет только на три ребра (см. рисунок 10), а так как накопленная сумма весов предметов в городе до и после измененного отрезка не изменяется, то и значение фитнесс-функции изменится только на измененном отрезке маршрута:

$$f(\text{П}^*) = f(\text{П}) - \text{стоимость старого отрезка} + \text{стоимость нового отезка} \quad (3.1)$$

Это позволяет сократить стоимость пересчета функции для всех возможных применений процедуры *«Insertion»* до $O(n^2)$.

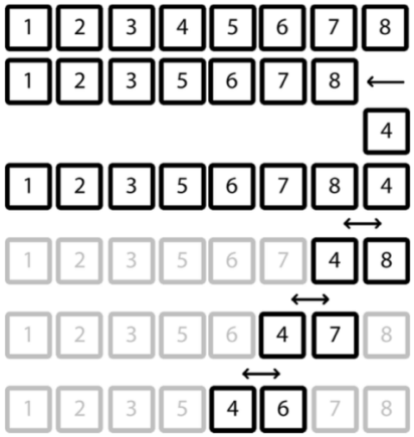

Рисунок 3.1. Использование инкрементального вычисления целевой функции для оператора *«Insertion»*

Предлагаемый оператор локального поиска маршрута формально расписан в листинге 3.3.

Листинг 3.3 – Процедура локального поиска *«Insertion»*

Входные данные: Π - маршрут, P - план упаковки
Предпосчитать суммарные веса в каждом городе $w[]$ и накопленные $w_{acc}[]$;
$Z := Z(\Pi, P); Z_{best} := Z;$
$\Pi_{best} := \Pi;$
цикл $pos := 2; pos < length(\Pi); pos := pos + 1$ **выполнять**
  повторять
    $pos_{neighbour} := pos - 1;$ (или $pos_{neighbour} = pos + 1$);
    $left := min(pos, pos_{neighbour});$
    $right := max(pos, pos_{neighbour});$
    $Z_{oldSeg} :=$ стоимость отрезка $[left, right];$
    $swap(\Pi_{pos}, \Pi_{pos_{neighbour}});$
    $Z_{newSeg} :=$ стоимость отрезка $[left, right];$
    $Z := Z - Z_{oldSeg} + Z_{newSeg};$
    если $Z > Z_{best}$ **тогда**
      $Z_{best} := Z;$
      $\Pi_{best} := \Pi;$
    конец
  до тех пор, пока $pos \neq 1$ *(или* $pos \neq n$*)*;
конец
Выходные данные: Π_{best}, P, Z_{best}

3.2.3. Локальный поиск плана упаковки

Вернемся к постановке задачи: сказано, что в одинаковые предметы могут находиться в разных городах. Очевидно, что если мы хотим украсть один из одинаковых предметов, то лучше всего положить его в рюкзак в городе, находящемся ближе всего к концу маршрута. Исходя из этого факта предлагается процедура улучшения плана упаковки рюкзака *«PackLater»*.

Для каждого предмета изначально запомним, сколько аналогичных по весу и стоимости предметов имеется в других городах. Тогда при фиксированном маршруте можно проверить каждый предмет, находящийся в плане упаковки, можно ли своровать такой же предмет в другом городе, который находится ближе к концу маршрута. Таким образом, значение целевой функции будет как минимум не меньше исходного значения.

Сгруппировать одинаковые предметы можно на этапе инициализации задачи. Тогда временная сложность данной процедуры будет равна $O(m \cdot \log(m))$ – необходимо проверить каждую группу предметов, и отсортировать предметы в группе по позиции города в текущем маршруте. Такая сложность получается, когда абсолютно все предметы одинаковы, что очень маловероятно, поэтому реальная сложность данной процедуры еще меньше.

Но если городов и предметов мало, то вероятность встретить одинаковые предметы очень мала, поэтому вышеописанная процедура будет неэффективна. Для данных случаев можно поступить иначе: для каждого города будем находить не только одинаковые города, но и те, что абсолютно лучше. При фиксированном маршруте один предмет лучше другого в том случае, если выполняются три условия:

а) вес первого предмета меньше или равен весу второго;

б) стоимость предмета больше или равна стоимости второго;

в) первый предмет украден в городе, находящемся ближе к концу маршрута, чем город со вторым предметом.

Листинг 3.4 – Процедура локального поиска плана упаковки рюкзака *«PackLater»*

Входные данные: П - маршрут, P - план упаковки, G - группы одинаковых предметов(G_i содержит номера предметов, одинаковых по весу и стоимости i-му предмету)

```
1  для каждого equalItems ∈ G выполнять
2      packedItems := 0;
3      для каждого item ∈ equalItems выполнять
4          если P_item = 1 тогда
5              packedItems := packedItems + 1;
6          конец
7      конец
8      sort(equalItems) ;          // По убыванию положения города в П
9      для каждого item ∈ equalItems выполнять
10         если packedItems > 0 тогда
11             P_item := 1;
12             packedItems := packedItems − 1;
13         иначе
14             P_item := 0;
15         конец
16     конец
17 конец
```

Выходные данные: P

С таким условием сформировать группы предметов можно только на каждой итерации алгоритма, так как они зависят от текущего маршрута вора. Сложность такого алгоритма $O(m^2)$ – необходимо каждый предмет, который входит в план упаковки рюкзака, сравнить со всеми другими из его группы, и если найден лучший – исходный предмет убрать из рюкзака и добавить новый предмет.

Данная процедура получила название *«PackBetter»*. Так как она более затратна, чем *«PackLater»*, то ее целесообразно запускать, когда количество предметов не более 10_4.

Формально же процедура *«PackBetter»* отличается от *«PackLater»* только на этапе сортировки предметов.

Выводы по главе 3

У алгоритмов, решающих данную задачу, есть проблема поиска локальных приближений, так как многие используют операторы мутации или кроссовера из задач TSP или KP. Поэтому мною предлагаются новые операторы локального поиска, которые бы производили поиск в более локальных областях относительно найденного решения.

Вышеописанные операторы можно использовать как совместно друг с другом, так и по-отдельности. В предложенном алгоритме я их запускаю последовательно в том порядке, в каком они представлены в данной работе.

Предложенные процедуры локального поиска вычислительно не сложны, однако могут быстро улучшить решение. Поэтому они также могут быть применены для ускорения поиска в других алгоритмах для решения TTP.

ГЛАВА 4. ВЫЧИСЛИТЕЛЬНЫЙ ЭКСПЕРИМЕНТ И АНАЛИЗ РЕЗУЛЬТАТОВ

В данной главе рассматриваются предложенные авторами задачи тестовые данные для сравнения алгоритмов. Далее описаны условия запуска алгоритмов и статистический анализ полученных результатов.

4.1. Тестовые данные

Так как данная задача является NP-полной, необходим универсальный способ оценивания решения различных реализаций задачи о воре. В статье [26] авторами самой задачи приведен способ генерации тестовых наборов данных для задачи о воре.

Тестовые данные основаны на экземплярах известной библиотеки задач TSP – TSPLIB [7]. Рассматриваются только случаи TSP, в которых расстояние между двумя городами определяется как евклидово расстояние. Экземпляр TSP из TSPLIB определяет количество городов и расстояния между ними. Чтобы превратить экземпляр задачи TSP в экземпляр задачи TTP, нужно сделать несколько вещей. Должны быть сгенерированы предметы, они должны быть назначены городам, а также должны быть заданы вместимость и стоимость аренды рюкзака. В этом разделе я объясню, как генерируются эти экземпляры, и я включил некоторые критические замечания о том, как они генерируются. Я считаю, что есть еще возможности для улучшения этих эталонных примеров, чтобы он в полной мере отражал остроту проблемы путешествующего вора.

4.1.1. Тип рюкзака

Корреляция между весом и стоимостью предметов может сильно влиять на сложность решения задачи о рюкзаке [25]. Поэтому Поляковский и соавторы выбрали три разных типа предметов в зависимости от корреляции весов и стоимостей предметов: некоррелированные, некоррелированные с похожими весами и сильно коррелированные с небольшим разбросом. Тот факт, что веса предметов очень схожи в экземплярах с сильно коррелированными предметами, сильно затрудняет решение. Однако то, что трудно решить в КР, не обязательно

трудно в TTP. В TTP отношение стоимость/вес предмета имеет меньшее значение, поскольку вклад в целевую функцию зависит не только от прибыли, но и от расстояний между городами. Мы можем сортировать предметы в соответствии с соотношением прибыли, веса и расстояния. Это делается практически всеми существующими алгоритмами TTP в форме жадной эвристики упаковки предметов (см. главу 2.3).

4.1.2. Распределение предметов

Задачи TTP генерируются так, что в каждом городе содержится одинаковое количество предметов, а какое именно - определяется специальным "множителем" предметов $F_i \in \{1, 3, 5, 10\}$.

4.1.3. Ограничение на вместимость рюкзака

Вместимость рюкзака задается в виде доли от общего веса всех предметов, где доля выбирается из набора $C \in \{1, 2, \dots, 10\}$. Итоговая вместимость W расчитывается как $W = \frac{C}{11} \sum w_i$. В результате получается 10 различных категорий вместимости рюкзака.

4.1.4. Стоимость аренды рюкзака

Стоимость аренды рюкзака имеет решающее значение для взаимозависимости проблемы [32]. Оно должно быть выбрано таким образом, чтобы и прибыль предмета, и время в пути вносили равный вклад в целевую функцию. Вклад в целевую функцию должен быть примерно таким же, чтобы гарантировать, что одно не доминирует над другим. Этот баланс обеспечивается путем установки арендной ставки $R = \frac{optimal\ sol.\ of\ KP}{near\ optimal\ sol.\ of\ TSP}$, где оптимальная стоимость маршрута аппроксимируется решением, найденным через эвристику Лина–Кернигана.

4.1.5. Сгенерированный набор тестовых данных

Набор тестовых данных основан на 81 экземпляре TSP, с 3 типами корреляции стоимости и весов предметов, 4 множителями предметов и 10

категориями вместимости рюкзака. Таким образом всего сгенерировано $81 \times 3 \times 4 \times 10 = 9720$ различных наборов входных данных для задачи.

Для проведения вычислительного эксперимента мною было выбрано подмножество данного набора следующим образом:

1) отсортированы по увеличению количество городов 81 экземпляр задачи TSP, и из них равномерно выбран каждый восьмой – итого 11 экземпляров;
2) все типы рюкзака включены в используемый набор;
3) из четырех множителей F_i взяты два: 3 и 10;
4) для ограничения на вместимость рюкзака выбраны 2 из 10 видов: $C \in \{3, 7\}$.

Итого мною для сравнительного анализа алгоритмов выбрано $11 \times 3 \times 2 \times 2 = 132$ экземпляра задачи TTP.

4.2. Схема проведения эксперимента

Для сравнения с предложенным мною алгоритмом использовался CS2SAR – множественный запуск алгоритма CS2SA в течение всего предоставленного лимита по времени. Данный выбор обусловлен тем, что по результатам сравнения алгоритмов [33] CS2SA находит наилучшие решения по сравнению с остальными в 1043 случаях из 9720, поэтому данный эволюционный алгоритм является одним из лучших, и он превосходит (1+1) EA и RLS.

Вычислительный эксперимент был проведен параллельным запуском каждого алгоритма по 10 раз (так как данные алгоритмы имеют случайную природу) на кластере, где каждому исполнителю выдавалось два потока и 4Гб оперативной памяти. Каждый запуск был ограничен 10 минутами и прерывался с найденным результатом.

4.3. Результаты вычислительного эксперимента

В результате запуска были получены результаты 1320 запусков (132 задачи, каждая по 10 раз) для каждого алгоритма. Для общего сравнения на

диаграмме 4.1 представлены сравнение по средним значениям и медианам. Так, по средним значениям предлагаемый алгоритм лучше в 74 случаях, хуже – в 35 случаях, а на 23 задачах были показаны одинаковые результаты. Если рассматривать медианы, то в 59 случаях предлагаемый алгоритм находил лучшее решение, в 50 случаях – результаты совпадали, а на 23 задачах – был хуже.

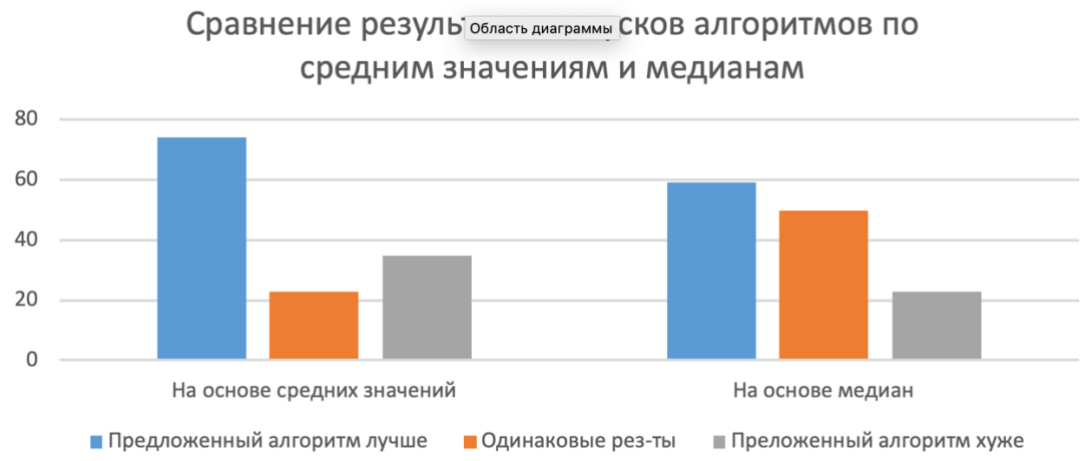

Диаграмма 4.1 – Сравнение результатов запусков алгоритмов по средним значениям и медианам

Что же касается эффективности предложенных локальных операторов, то оператор *«Optimal Subtour Search»* нашла улучшение для 53 задач из 132, процедура *«Insertion»* для 47 задач, а процедуры *«PackLater»* и *«PackBetter»* – только на 21 задаче. Данная статистика отображена на диаграмме 4.2. В данной статистике учитывались операторы, которые нашли улучшения хотя бы в 1 из 10 запусков.

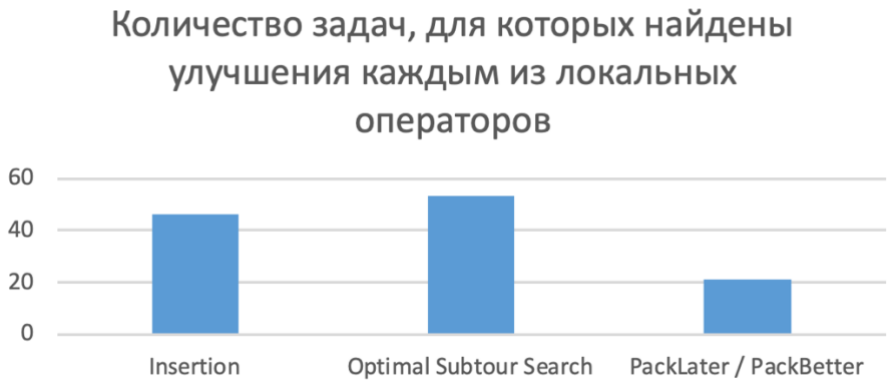

Диаграмма 4.2 – Статистика улучшений по каждому локальному оператору

Также проведена проверка на нормальность распределения полученных результатов для каждого набора запусков алгоритма. Этот тест выполнялся следующим образом: сначала вычисляются средние значения (mean) и медианы (median) по каждому набору запусков для двух алгоритмов. Далее эти значения сравниваются попарно. Если медиана одного алгоритма больше медианы другого, но среднее значение – меньше (или же наоборот), то считалось, что распределение в данной выборке не совсем близко к нормальному. Данным образом выявлено 9 из 132 таких случаев, что довольно немного. Поэтому можно предположить, что в целом распределение полученных результатов близко к нормальному.

Прежде чем проводить более глубокий статистический анализ, необходимо выделить некоторые особенности тестовой выборки (полученных результатов). Во-первых, выборки результатов запусков двух алгоритмов на каждой из задач независимы. Во-вторых, произведено всего 10 запусков каждого алгоритма, поэтому следует учитывать небольшой объем выборки.

4.3.1. Анализ результатов с помощью критерия Уилкоксона-Манна Уитни

Критерий Уилкоксона-Манна-Уитни (англ. Mann–Whitney U test) [35] – непараметрический тест, который проверяет, что случайно выбранное значение из одной выборки будет меньше или больше случайно выбранного значений из второй выборки. Данный тест применяется для небольших (с объемом меньше 20 значений) и независимых выборок, поэтому он подходит для анализа полученных результатов. Чем меньше значение критерия, тем вероятность того, что одна выборка различается от другой (или же доминирует над другой), выше.

Под нулевой гипотезой H_0 понимается, что распределения двух выборок одинаковы. Альтернативной гипотезой H_1 является то, что одна выборка доминирует над другой.

Пусть X – выборка с результатами работы предлагаемого алгоритма на некоторой задаче, а Y – выборка с результатами работы алгоритма CS2SA-R на той же задаче.

Сначала был запущен тест с альтернативной гипотезой $X < Y$. По результатам данного теста с уровнем значимости $\alpha = 0{,}0005$ выявлено всего 9 ошибок первого рода.

Затем запущен тест с альтернативной гипотезой $X > Y$. Результаты данного теста показали с уровнем значимости $\alpha = 0{,}0005$, что нашлось 28 ошибок первого рода.

Исходя из результатов данного теста можно сделать вывод, что предложенный алгоритм примерно в 3 раза чаще выдает лучший результат.

4.3.2. Применение поправки Холма-Бонферрони

Поправка Холма–Бонферрони (англ. Holm-Bonferroni correction) – поправка на множественную проверку гипотез. Она необходима при построении некоторого множества статистических выводов, потому что с ростом числа выводов вероятность того, что хотя бы один из них будет некорректным, значительно увеличивается.

Алгоритм данной поправки достаточно прост. Пусть даны заранее отсортированные по неубыванию уровни значимости $p_1 \leq \cdots \leq p_m$, и соответствующие им гипотезы $H_1, \ldots, H_m$. Фиксируется некоторый уровень значимости α. До тех пор, пока выполняется $p_i \leq \frac{\alpha}{m-i+1}$ – гипотеза H_i отвергается, и проверяется следующая гипотеза; иначе – все последующие гипотезы (включая текущую) $H_i, \ldots, H_m$ принимаются.

В качестве входных данных использовались уровни значимости, полученные тестом Уилкоксона-Манна-Уитни. Уровень значимости зафиксирован $\alpha = 0{,}05$.

Данная поправка также применялась к двум наборам данных – уровням значимости, полученным тестом Уилкоксона-Манна-Уитни с альтернативными гипотезами $X < Y$ и $X > Y$. Полученные результаты совпали предыдущими –

отвергнуто 9 гипотез в случае, когда проверялось, что предложенный алгоритм хуже существующего. При этом аналогично отвергнуто 28 гипотез в случаях проверки, что предложенный алгоритм хуже существующего.

Выводы по главе 4

Для проведения вычислительного эксперимента были выбраны случайным образом 132 экземпляра задач из множества, предложенного авторами задачи. Также описаны ограничения и условия запуска алгоритмов, самое главное из которых – ограничение в 10 минут на время работы каждого запуска алгоритма.

В результате анализа проведенного эксперимента было выявлено, что в среднем на более 56% задач найдены лучшие решения, но в 26% случаев полученные решения оказались хуже. В остальных же случаях результаты приблизительно одинаковы. Если рассматривать полезность предложенных операторов локального приближения, то те, что изменяют маршрут – давали приближение примерно в 2,5 раза чаще, нежели локальные операторы изменения плана упаковки рюкзака. Однако, в рамках данной задачи не стоит их разделять, так как они взаимосвязаны и использовались совместно.

Статистический анализ показал, что с вероятностью $p = 0{,}05$ примерно в 3 раза чаще алгоритм решает задачу лучше в сравнении с CS2SA-R, нежели хуже. Это также подтверждает результаты предварительного анализа в том, что предложенный алгоритм в целом работает лучше относительно CS2SA-R.

ЗАКЛЮЧЕНИЕ

Задача в воре – одна из новых задач оптимизации, которая комбинирует две известные задачи – задачу коммивояжера и задачу о рюкзаке. Но самая главная проблема данной задачи в том, что применение существующих алгоритмов для решения каждой из подзадач не приводят к успешному решению исходной задачи. Анализ существующих решений показал, что на данный момент нет алгоритма, который бы решал задачу лучше всех остальных.

В данной работе предложены операторы локального поиска решения, потому что применение аналогичных операторов, разработанных для каждой из подзадач, зачастую дает нелокальные изменения. Предложены по два оператора для локального изменения маршрута и плана упаковки рюкзака. Данные операторы были адаптированы к существующему алгоритму CS2SA, таким образом был сконструирован меметический алгоритм.

Предложенный алгоритм показал прирост в поиске более оптимальных решений относительно другой модификации алгоритма CS2SA, представленной в 2018 году – CS2SA-R. По усредненным значениям запусков для более 56% задач найдены лучшие решения, но в 26% случаев полученные результаты оказались хуже. В остальных же случаях результаты приблизительно одинаковы. Результаты статистического анализа показали, что с уровнем значимости $\alpha = 0{,}05$ примерно в 3 раза чаще алгоритм решает задачу лучше в сравнении с CS2SA-R, нежели хуже.

К сожалению, остаются экземпляры задач, которые абсолютно не получили никакого улучшения после использования предлагаемых операторов, а также есть случаи, когда работал хуже. Поэтому есть большой задел для будущий исследований и поиска новый подходов к решению данной задачи.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. M. R. Bonyadi, Z. Michalewicz, and L. Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems // IEEE Congress on Evolutionary Computation. – 2013. – P. 1037–1044.

2. D. Applegate, W. Cook, A. Rohe. Chained Lin–Kernighan for large traveling salesman problems // INFORMS Journal on Computing. – 2003. – Vol. 2. – P. 82–92.

3. P. Toth, D. Vigo. The vehicle Routing Problem. – Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. – 2001.

4. David L. Applegate. The Traveling Salesman Problem: A Computational Study. – Princeton University Press. – 2007.

5. H. Kona, A. Burde, D. R. Zanwar. A Review of Traveling Salesman Problem with Time Window Constraint // International Journal for Innovative Research in Science & Technology. – 2015. – Vol. 2. – P. 253–254.

6. M. Hahsler, K. Hornik. TSP-Infrastructure for the traveling salesperson problem // Journal of Statistical Software. – 2007. – Vol. 23, no. 2. – P. 1–21.

7. G. Reinelt. TSPLIB – A traveling salesman problem library // ORSA journal on computing. – 1991. – Vol. 3. – P. 376–384.

8. M. Held, R.M. Karp. A dynamic programming approach to sequencing problems // Journal of the Society for Industrial and Applied Mathematics. – 1962. – Vol. 10, no. 1. – P. 196–210.

9. S. Lin, B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem // Operations Research. – 1973. – Vol. 21, no. 2. – P. 498-516.

10. C. Nilsson. Heuristics for the Traveling Salesman Problem // Journal of Theoretical Computer Science Reports. – 1996. – P. 473–480

11. K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic // European Journal of Operational Research. – 2000. – Vol. 126, no. 1.– P. 106–130.

12. W. Cook. In pursuit of the traveling salesman: mathematics at the limits of computation. – Princeton University Press. – 2012.

13. Y. Nagata, S. Kobayashi. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. // INFORMS Journal on Computing. – 2013. – Vol. 25, no. 2. – P. 346–363.

14. P. Larranaga et al. Genetic algorithms for the travelling salesman problem: A review of representations and operators // Artificial Intelligence Review. – 1999. – Vol. 13, no. 2. – P. 129–170.

15. Timothy Starkweather et al. “A Comparison of Genetic Sequencing Operators // International Conference on Genetic Algorithms. – 1991. – P. 69–76.

16. I. Oliver, D. Smith, J. Holland. Study of permutation crossover operators on the traveling salesman problem // Proceedings of International Conference on Genetic Algorithms. – 1987.

17. L. Davis. Applying adaptive algorithms to epistatic domains // International Joint Conferences on Artificial Intelligence. – 1985. – Vol. 85. – P. 162–164.

18. D. Goldberg, R. Lingle, et al. Alleles, loci, and the traveling salesman problem // Proceedings of an international conference on genetic algorithms and their applications. – 1985. – Vol. 154. – P. 154–159.

19. H. Muhlenbein, M. Gorges-Schleuter, O. Kramer. Evolution algorithms in combinatorial optimization // Parallel Computing. – 1988. – Vol. 7, no. 1. – P. 65–85.

20. K. Mathias, D. Whitley. Genetic operators, the fitness landscape and the traveling salesman problem // Parallel Problem Solving from Nature Conference. – 1992. – P. 221–230.

21. D. Whitley, D. Hains, A. Howe. Tunneling between optima: partition crossover for the traveling salesman problem // Proceedings of Genetic and Evolutionary Computation Conference. – 2009. – P. 915–922.

22. D. Whitley, L. Darrell et al. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator // International Conference on Genetic Algorithms. – 1989.

23. С. Окулов. Программирование в алгоритмах. – Бином. Лаборатория знаний. – 2007. – С. 384.

24. S. Martello, D. Pisinger, P. Toth. New trends in exact algorithms for the 0–1 knapsack problem // European Journal of Operational Research. – 2000. – P. 325–332.

25. S. Martello, D. Pisinger, P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem // Management Science. – 1999. – P. 414–424.

26. S. Polyakovskiy, M. R. Bonyadi, M. Wagner, F. Neumann, Z. Michalewicz. A Comprehensive Benchmark Set and Heuristics for the Travelling Thief Problem // Proceedings of Genetic and Evolutionary Computation Conference. – 2014. – P. 477–484.

27. M. R. Bonyadi, Z. Michalewicz, M. R. Przybylek, A. Wierzbicki. Socially inspired algorithms for the TTP // Proceedings of Genetic and Evolutionary Computation Conference. – 2014. – P. 421–428.

28. Y. Mei, X. Li, X. Yao. On investigation of interdependence between sub-problems of the travelling thief problem // Soft Computing. – 2016. – Vol. 2, no. 1. – P. 157–172.

29. H. Faulkner et al. Approximate approaches to the traveling thief problem // Proceedings of Genetic and Evolutionary Computation Conference. – 2015. P. 385–392.

30. M. El Yafrani, B. Ahiod. Population-based vs. single-solution heuristics for the travelling thief problem // Proceedings of Genetic and Evolutionary Computation Conference. – 2016. – P. 317–324.

31. M. El Yafrani, B. Ahiod. Efficiently Solving the Traveling Thief Problem using Hill Climbing and Simulated Annealing // Information Sciences, Elsevier. – 2018. – P. 231–244.

32. M. Wagner. Stealing items more efficiently with ants: a swarm intelligence approach to the travelling thief problem // International Conference on Swarm Intelligence. – 2016. – P. 273–281.

33. M. Wagner et al. A case study of algorithm selection for the traveling thief problem // Journal of Heuristics. – 2017. – P. 1–26.

34. H. B. Mann, D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other // Annals of Mathematical Statistics. – 1947. – P. 50-60.

35. S. Holm. A simple sequentially rejective multiple test procedure // Scandinavian Journal of Statistics. – 1979. – Vol. 6, no. 2. – P. 65-70.