Санкт-Петербургский государственный университет

Кафедра математического моделирования энергетических систем

Гуров Василий Максимович

Выпускная квалификационная работа бакалавра

Управление товарными потоками и распределение запасов в дистрибьюторской сети

Направление 010400
«Прикладная математика, фундаментальная информатика и программирование»

Научный руководитель,
доктор физ. - мат. наук,
профессор
Захаров В. В.

Санкт-Петербург
2016

ОГЛАВЛЕНИЕ

Стр.

ГЛАВА 1

ВВЕДЕНИЕ

1.1 Дистрибьюторские сети в наше время

В наше время очень сложно представить себе крупную компанию, будь то производитель продуктов питания, или транспортная компания без сети региональных представителей. Для крупных компаний, особенно за границей, даже свойственно иметь целое множество дистрибьюторов. Каждый из таких дистрибьюторов представляет продукцию своей компании в какой-нибудь из стран или регионов.

Таким образом, изготовители продукции считают, что без отлаженной системы поставок их продукт не попадет в розничные точки и, следовательно, не дойдет до рук клиента.[1]

Цепи поставок, организующиеся в наше время, становятся все более сложными, они уже включают множество ключевых звеньев как внутри компании, так и за ее пределами. Поэтому изменение цепи поставок, особенно в сторону увеличения количества звеньев оказывает существенное влияние на суммарные затраты поставок. А ведущими критериями эффективности цепи поставок становятся такие критерии как качество обслуживания потребителей. Требования, как к качеству продукции, так и к ассортименту увеличиваются, потребители становятся все более требовательными к скорости и качеству доставки и другим критериям сервиса. Такие требования существенно влияют на уровень логистических затрат, что в конечном счете вынуждает руководителей крупных организаций задуматься о том, как оптимизировать цепь поставок так, чтобы сохранилось качество обслуживания клиентов, но при этом свести к минимуму затраты на транспортировку и хранение продукции.[2]

Такие задачи являются актуальными для среднего и крупного бизнеса в течение многих лет, и многие исследователи изучают различные математические модели логистических структур и осуществляют поиск и разработку способов их оптимизации, в том числе и наиболее эффективных.

Наш выбор остановился на дистрибьюторских сетях, так как распространение продукции является одним из ключевых факторов в получении прибыли компании, и во многих компаниях ожидания клиентов относительно скоординированности и синхронизации материальных потоков не оправдываются.

1.2 Что такое дистрибьюторская сеть

Дистрибьюторская сеть представляет собой множество посредников компании - производителя продукции, помогающих доставить её продукцию конечному потребителю. Чаще всего она состоит из оптовых компаний - дистрибьюторов.

Для распределения продукции в цепях поставок используются каналы. Каналы - это набор посредников, участвующих в процессе доставки продукции компании от производителя к конечному потребителю.

Каналы классифицируют согласно количеству промежуточных узлов в цепи поставок(посредников) между потребителем и производителем продукции.

В каналах нулевого уровня посредники отсутствуют. Дистрибьюторские сети обычно содержат одного посредника(назовем его дистрибьютором) и являются каналами первого уровня.

В данной работе рассматривается дистрибьюторская сеть с каналами первого уровня.

1.3 Цели и задачи данной работы

В цепях поставок между производителями и потребителями находятся посредники, цель которых - сократить свои издержки, максимизируя при этом прибыль. Для них расчёт оптимальных объемов поставок, при которых минимизируются издержки, является одним из ключевых действий для выживания на рынке и дальнейшей деятельности. Как следствие, цены, по

которым посредник закупает партию продукта, влияют на розничную цену для конечного потребителя.

В данной работе мы проведём исследование модели дистрибьюторской сети, её анализ, решим задачу оптимизации дистрибьюторской сети, рассмотрим алгоритмы решения такой задачи и оценим их производительность.

Объектом исследования является дистрибьюторская сеть.

Предметами исследования – математическая модель дистрибьюторской сети, алгоритмы решения оптимизационной задачи для данной модели, результаты расчетов данных алгоритмов на различных наборах входных данных.

Цель исследования состоит в получении решений оптимизационной задачи для математической модели дистрибьюторской сети путем реализации различных алгоритмов, анализе полученных экспериментальных данных, выявлении наиболее оптимальных с точки зрения полученных решений и точки зрения затраченных ресурсов алгоритмов.

Для достижения указанной цели решаются следующие задачи:

а) Построение математической модели, описывающей дистрибьюторскую сеть.
б) Обзор и изучение существующих алгоритмов на применение к решению оптимизационной задачи на основе данной математической модели.
в) Программная реализация изученных алгоритмов.
г) Получение приближённых решений задачи и прочих экспериментальных данных на основе различных наборов исходных данных.
д) Анализ полученных данных. Построение выводов.

1.4 Дистрибьюторская сеть компании BMW

Дистрибьюторская сеть, рассматриваемая в данной работе, имеет сходство с цепями поставок крупнейших производителей автомобилей. К таким относится и немецкий автомобильный завод BMW. На примере дистрибью-

торской сети данной компании мы построим математическую модель и решим ряд задач.

Дистрибьюторская стратегия компании основывается на величине спроса на продукцию, периодически оцениваемой специалистами компании. Основная идея стратегии заключается в поставке продукции в те места, где сосредоточено большинство клиентов. И это, несомненно, влияет как на сроки доставки, так и на издержки в процессе поставок.

Компания постоянно оптимизирует свою дистрибьюторскую сеть для достижения наибольшей эффективности и сокращения издержек.

Дистрибьюторская сеть компании BMW имеет множество конфигураций, основанных на расположении дистрибьюторов и клиентов, спросе на продукцию и вместимости складских комплексов дистрибьюторов и дилеров.

Механизмы доставки продукции от производителя к конечному клиенту также бывают разными. Одна из конфигураций поставки - непосредственная прямая поставка продукта с завода клиенту. Кроме того, цепь поставок может включать дилеров и дистрибьюторов. Три основные формы дистрибуции в компании BMW AG приведены на рис.1.1.

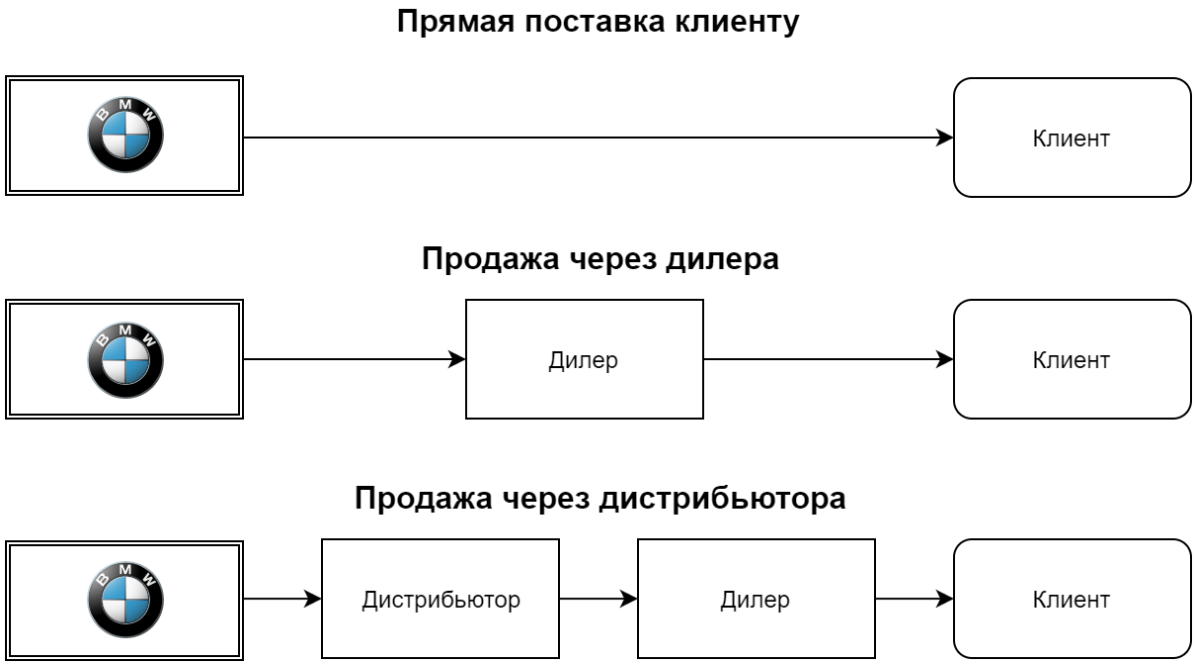

Рисунок 1.1 — Конфигурации дистрибьюторской сети компании BMW

В некоторых регионах, таких как Южная Америка, компания предпочитает использовать так называемые дистрибьюторские пункты, куда с завода поставка автомобилей осуществляется в первую очередь. Кроме того, данные центры участвуют в распределении продаваемых транспортных средств по дилерам. Пока автомобиль не доставлен дилеру для розничной продажи, он удерживается в центральном пункте, причем BMW в данном случае оплачивает все затраты на хранение. Поэтому издержки хранения, в первую очередь, несет компания, а не сторонние организации. Такой подход помогает компании сэкономить значительные средства на транспортировке своей продукции, что также доказывает преимущество распространения продукции через дистрибьюторов над прямыми поставками. [4]

Значительная экономия может быть достигнута в случае транспортировки автомобилей от дистрибьютора до дилера, если дилеру подобран такой дистрибьютор, стоимость поставки от которого будет наименьшей.

Таким образом, наиболее эффективное построение дистрибьюторской сети для таких стран как Великобритания, Ирландия и Россия позволило компании увеличить объем выручки с данных рынков. Оптимизация поставок в других страны также является одной из приоритетных задач компании для дальнейшего развития.

1.5 Обзор литературы: дистрибьюторские сети

Очевидно, что построение и оптимизация дистрибьюторских сетей играет важную роль в развитии бизнеса. Поэтому многие исследователи занимаются изучением данной области. Модели дистрибьюторских сетей также довольно часто оказываются в поле зрения не только менеджеров, но и математиков. Ведь исследования таких моделей достаточно интересны и актуальны. Приведем несколько примеров.

Например, в материале [5] авторы исследования осуществляют математическое моделирование дистрибьюторской сети автосервисов. Основная задача исследователей - максимизировать комплексный критерий, который включает в себя несколько свойств модели.

Другие российские исследователи [6] разработали принципы организации дистрибуции внутри торговой сети. В данной работе было показано, что наличие дистрибьюторских распределительных центров может значительно сократить общие затраты цепи поставок.

Исследования в области логистики и цепей поставок также проводятся и зарубежными учеными и исследователями.

В работе [3] рассматривается модель цепи поставок в дистрибьюторской сети со спросом, зависящем от цены товара. Одной из особенностей данной модели является наличие уровней цен (markup levels). Задача относится к классу смешанных задач нелинейного целочисленного программирования и решается алгоритмами недифференцируемой оптимизации. При решении был рассмотрен способ ослабления ограничений и сведения задачи к задаче нелинейного целочисленного программирования. Решение такой задачи подразумевает нахождение распределения запасов, цену и размер поставок для максимизации прибыли, полученной в результате продажи продукции.

В исследовании [7] того же автора был описан эвристический алгоритм решения задачи максимизации прибыли для многопродуктовой дистрибьюторской сети.

В исследовании [8] авторами была решена задача оптимизации двухслойной дистрибьюторской сети для Ирландской цепи поставок в молочной отрасли пищевой промышленности. Одним из критериев оптимизации, помимо затрат сети, была минимизация выбросов углекислого газа в атмосферу.

Также широкий спектр приложений дистрибьюторских сетей в реальной жизни рассматривается и в других статьях. Ведь дистрибуция используется не только в задачах бизнеса.

В статье [9] рассмотрена задача управления потоками опасных отходов и расположения пунктов их переработки. Основная цель данной модели - уменьшение общей стоимости организации и транспортировки опасных отходов, а также минимизация различных рисков. Данная модель внедряется в Турции.

В другом материале [10] рассмотрена математическая модель сети для

сбора и вывоза мусора в городе Брюссель. Одной из особенностей данной модели является то, что для транспортировки мусора используются несколько типов транспортных средств - автомобили, железнодорожный и водный транспорт. Авторы решают задачу линейного программирования под набором ограничений.

Таким образом, задачи дистрибуции и их математические модели актуальны и интересны исследователям, причем решение подобных задач имеет спрос не только в бизнесе, но и в сфере общественных услуг.

ГЛАВА 2

МАТЕМАТИЧЕСКАЯ МОДЕЛЬ ДИСТРИБЬЮТОРСКОЙ СЕТИ

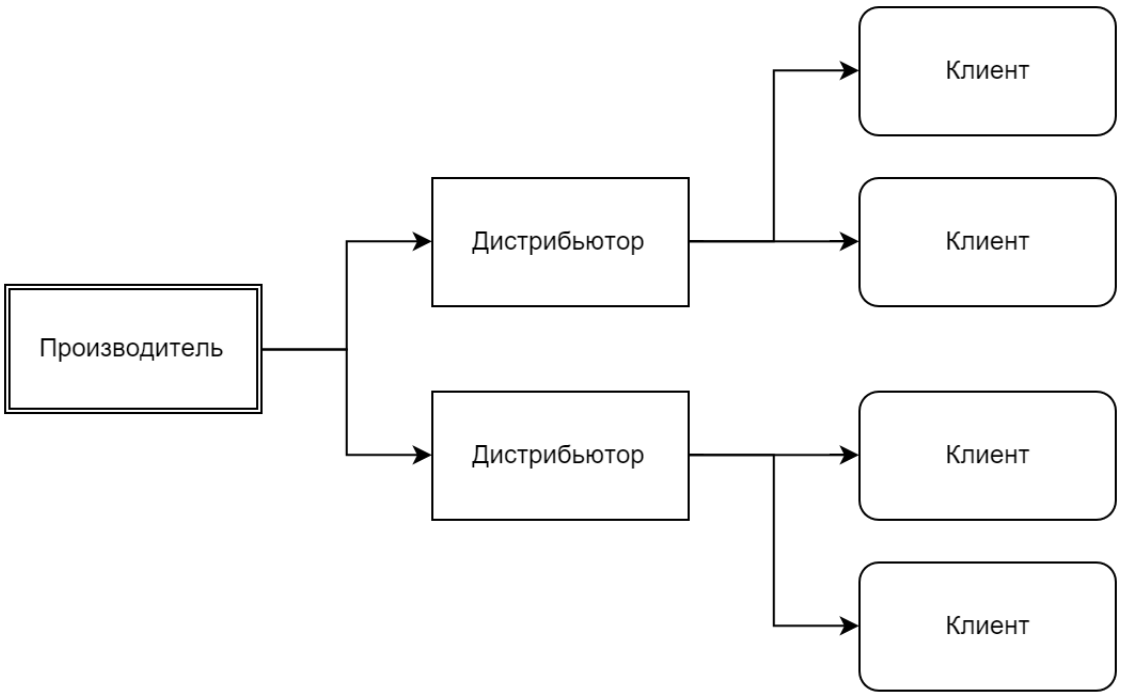

Рисунок 2.1 — Общая схема рассматриваемой дистрибьюторской сети

В данной работе рассматривается дистрибьюторская сеть как множество взаимосвязанных организаций трех типов: производителя, дистрибьюторских центров (посредников) и ритейлеров.

Считаем, что в третьей конфигурации сети компании BMW (рис.1.1) нас не интересует спрос отдельных клиентов, при этом каждый дилер знает необходимое ему для продажи количество автомобилей. Данную величину назовем спросом клиента j. Далее она будет обозначаться как d_j.

Таким образом, в предложенной сети, дилер является клиентом, для которого известен и заранее определен его спрос.

Продукция поступает от производителя в дистрибьюторский центр, где перепродается по оптовой цене клиентам. Клиенты (ритейлеры), в свою очередь, реализуют полученную продукцию по розничной цене. В процессе поставок дистрибьюторские центры несут определенные издержки, например, затраты на хранение продукции, перевозку товаров от производителя в дистрибьюторский центр, организационные издержки.

Наша задача - математическими методами определить, нужно ли включать в цепь поставок каждый из возможных дистрибьюторских центров и каким клиентам должен поставлять продукцию каждый из дистрибьюторов (и должен ли вообще), чтобы максимизировать прибыль всей сети в целом.

2.1 Основные предположения

1. Каждый дистрибьютор осуществляет поставку только одного вида товара, который является общим для всех дистрибьюторов. Рассматривается модель с одним продуктом.

2. Каждый ритейлер осуществляет заказ только у одного дистрибьютора. Деление заказов ритейлера на поставки от разных дистрибьюторов в данной модели недопустимо.

3. Вместимость складов каждого дистрибьютора ограничены.

4. Спрос каждого из клиентов постоянный в течение всего периода планирования.

5. Время выполнения поставки продукции клиентам постоянно для всех дистрибьюторских центров. Считаем, что поставка осуществляется мгновенно.

2.2 Основные обозначения

- $I = \{i_1, ..., i_n\}$ - множество дистрибьюторов.
- $J = \{j_1, ..., j_m\}$ - множество клиентов / ритейлеров.
- f_i - стоимость включения дистрибьюторского центра в сеть. (издержки на обеспечение работы дистрибьюторского центра).
- t_{ij} - стоимость доставки единицы товара от дистрибьютора i до клиента j.
- c_i - вместимость склада дистрибьюторского центра i (в единицах продукции).
- e_i - стоимость осуществления поставки от производителя к дистрибьютору i (фиксированная).

- a_i - стоимость перевозки единицы продукции от производителя к дистрибьютору i.
- h_i - стоимость хранения единицы продукции в дистрибьюторском центре i в течение всего периода планирования.
- wp_i - оптовая стоимость единицы продукции в дистрибьюторском центре i
- p_j - розничная цена товара, установленная ритейлером j.
- d_j - спрос клиента j (ед.)
- q_i - объем поставки от производителя дистрибьютору i (ед.)

$$q_i = \sum_{j \in J} d_j y_{ij}$$

В силу того, что спрос клиента постоянный, считаем, что средний объём хранимой на складе дистрибьютора продукции равен

$$\frac{q_i h_i}{2}.$$

Данная величина будет использоваться при подсчёте суммарных издержек дистрибьюторской сети.

2.3 Искомые значения

а) Бинарная переменная x_i, которая показывает, включен ли дистрибьютор i в сеть.

$$x_i = \begin{cases} 1 & \text{если дистрибьютор i включен в сеть} \\ 0 & \text{в противном случае} \end{cases}$$

б) Бинарная переменная y_{ij}, показывающая, связаны ли поставками дистрибьютор i и ритейлер j.

$$y_{ij} = \begin{cases} 1 & \text{поставка от дистрибьютора i ритейлеру j осуществляется} \\ 0 & \text{в противном случае} \end{cases}$$

Исходя из описанных выше параметров, еще раз сформулируем задачу: необходимо определить, организовать работу какого подмножества дистрибьюторских центров из множества потенциальных дистрибьюторов, назначив на каждый из них некоторое подмножество клиентов для осуществления поставок необходимо, чтобы максимизировать прибыль всей дистрибьюторской системы.

Стоит заметить, что, на самом деле, решение задачи может определяться только значениями переменных y_{ij}, так как переменные x_i зависят от y_{ij}.

Данная переменная была введена для удобства записи целевой функции и ее задания в программном виде. Связь этих переменных x_i и y_{ij} рассматривается в разделе 2.5.

Функция прибыли дистрибьюторской сети предложена далее в разделе 2.4.

2.4 Целевая функция

Так как основная цель дистрибьюторской сети в поставленной задаче - максимизация прибыли, то построим классическую функцию прибыли, состоящую из доходов за счет розничных продаж сети за вычетом всех основных затрат и издержек.

$$TotalProfit = TotalIncome - TotalCosts \rightarrow max \tag{2.1}$$

Total Income - есть доходы. К ним относится общая сумма, полученная в результате реализации всех товаров ритейлерами по розничным ценам.

$$TotalIncome = \sum_{i \in I} \sum_{j \in J} p_j d_j y_{ij}. \tag{2.2}$$

Далее рассмотрим, из каких частей складывается Total Costs.

Для включения дистрибьюторского центра в цепь поставок необходимы некоторые средства. Стоимость открытия и обеспечения работоспособности каждого из дистрибьюторов определяется значением f_i. Суммарная

стоимость открытия всех дистрибьюторских центров:

$$LaunchCosts = \sum_{i \in I} f_i x_i. \tag{2.3}$$

Величина wp_i характеризует оптовую стоимость покупки товара для i-го дистрибьютора. Суммарный объем затрат на закупку продукции дистрибьюторами есть

$$WholesalePurchaseCosts = \sum_{i \in I} \sum_{j \in J} wp_i d_j y_{ij} \tag{2.4}$$

Суммарная стоимость транспортировки продукции товаров между дистрибьюторами и клиентами есть

$$DistributorTransportationCosts = \sum_{i \in I} \sum_{j \in J} t_{ij} d_j y_{ij} \tag{2.5}$$

Каждый из дистрибьюторов также несет следующие расходы:

Затраты на хранение:

$$HoldingCosts = \sum_{i \in I} \frac{h_i q_i}{2} = \sum_{i \in I} \sum_{j \in J} \frac{h_i d_j y_{ij}}{2} \tag{2.6}$$

Затраты на доставку от производителя дистрибьютору:

$$ProducerTransportationCosts = \sum_{i \in I} (e_i + a_i q_i) \tag{2.7}$$

В результате получаем следующее выражение прибыли для дистрибьюторской сети.

$$\max \sum_{i \in I} \sum_{j \in J} p_j d_j y_{ij} - \sum_{i \in I} f_i x_i - \sum_{i \in I} \sum_{j \in J} wp_i d_j y_{ij} - \sum_{i \in I} \sum_{j \in J} t_{ij} d_j y_{ij} - \\ - \sum_{i \in I} \left(\frac{1}{2} \sum_{j \in J} h_i d_j y_{ij} + e_i + a_i q_i \right) \tag{2.8}$$

Первое слагаемое в данной сумме есть общая стоимость проданного ритейлерами товара по розничной цене. Это единственное положительное слагаемое в данной сумме, далее из него вычитаются все основные затраты.

Второе слагаемое - суммарная стоимость открытия всех дистрибьюторских центров. Третье слагаемое - общая стоимость проданного ритейлерам товара по оптовой цене. Четвертое - суммарные затраты на транспортировку между дистрибьюторскими центрами и ритейлерами. Далее для каждого из дистрибьюторов мы вычисляем стоимость хранения, стоимость доставки партии товаров к дистрибьютору от производителя и вычитаем их также из выручки ритейлеров в данной дистрибьюторской сети.

2.5 Ограничения

Не все возможные вектора решений x_i и значения y_{ij} характеризуют вполне адекватное распределение товарных потоков. Поэтому для того чтобы получить только допустимые решения, а также ограничить область поиска, введем некоторые ограничения, предложенные далее.

а) Ограничение указывает на то, что обслуживание клиента возможно только одним дистрибьютором, либо, в случае, если данная сумма равна нулю, кто клиент не обслуживается ни одним дистрибьюторским центром.

$$\sum_{i \in I} y_{ij} \leq 1 (j \in J) \tag{2.9}$$

б) Следующее ограничение означает, что если дистрибьютор не включен в сеть и не обслуживает клиентов, то никакой из клиентов не может обслуживаться у него.

$$y_{ij} - x_i \leq 0; (i \in I) \tag{2.10}$$

в) Ограничение на объем поставки: поставка дистрибьютору i не может превышать вместимости дистрибьюторского центра.

$$q_i - c_i \leq 0 (i \in I) \tag{2.11}$$

г) Ограничение на бинарные переменные

$$x_i \in \{0,1\}; y_{ij} \in \{0,1\}; (i \in I)(j \in J) \tag{2.12}$$

д) Ограничения на объем поставки дистрибьюторам

$$q_i \geq 0 (i \in I) \tag{2.13}$$

ГЛАВА 3

МЕТОДЫ РЕШЕНИЯ

Модель, описанная выше, относится к классу задач смешанного нелинейного целочисленного программирования (MINLP). Решения таких задач зачастую трудны, и такие задачи классифицируются как NP-сложные задачи.

Приведем несколько примеров ранее решенных задач, относящихся к данному классу. Множество моделей, относящихся к MINLP, описывают потоки в сетях. Одна из статей на эту тему [11] описывает построение оптимальных водопроводных сетей. Другие исследователи применяют решение подобных задач к системе метрополитена Нью-Йорка. [12] Авторы решают задачу о минимизации функционала, описывающего суммарное потребление электроэнергии в сети подземного транспорта. Одно из основных направлений решения таких задач - цепи поставок в логистике, например, поставки воды и газа, применение в компьютерной маршрутизации в задаче отражения кибератак. Также решения задач подобного типа находятся в химической промышленности и атомной энергетике.

Поэтому решение таких задач особенно актуально, как и поиск способов их решения.

3.1 Обзор литературы: методы решения

Для решения задач подобного типа существуют целое множество алгоритмов. Основные и часто применяемые в подобных задачах будут рассмотрены ниже, а некоторые из них будут далее реализованы нами на практике и изучена их производительность.

Как мы знаем, алгоритмы могут осуществлять построение, как точного решения, так и приближенного (эвристические алгоритмы).

Самым популярным точным алгоритмом для решения данной задачи является алгоритм Branch and Bound, иначе известный как метод ветвей и границ. С сутью данного алгоритма и его применением к задачам нелиней-

ного программирования можно ознакомиться в материале [13].

В статье [14] для решения похожей задачи рассматривается эвристический алгоритм VNS. Его суть заключается в последовательной смене районов поиска. В каждом районе происходит локальный поиск. Если в районе находится лучшее решение, то поиск продолжается в данном районе. Если же нет, то происходит смена района. Подробнее с данным алгоритмом, а также с его реализацией мы познакомимся далее - в разделе 3.2.

Кроме того, к подобным задачам применим целый класс генетических алгоритмов.[15] Генетические алгоритмы могут быть применены к широкому классу задач с многоканальными дистрибьюторскими сетями, но не стоит забывать о том, что для всех эвристических алгоритмов свойственно отсутствие гарантии нахождения оптимального решения в пространстве решений за разумное время. В исследовании [15] авторы предлагают применение генетических алгоритмов к задаче определения оптимальных уровней запаса в дистрибьюторской сети с использованием скидок. Авторы также приходят к выводу, что использование суммарных издержек и стоимости сети как основных критериев оптимизации не всегда является разумным подходом к оптимизации.

3.2 Алгоритм VNS

Алгоритм VNS является популярным эвристическим алгоритмом, который очень часто применяется для решений различных задач в области логистики и управления запасами. Одним из примеров решения такой задачи является применение данного алгоритма к задаче IRP в цепи топливных поставок. [16] Кроме того, часто алгоритм VNS применяется в задачах маршрутизации и согласно исследованию [17], в котором рассматривается задача маршрутизации в многопортовой системе с разнородным флотом, успешно с ними справляется.

3.2.1 Кратко об алгоритме VNS

Алгоритм VNS (Variable Neighborhood Search) был разработан в 1997 году. Основная идея заключается в систематическом изменении района в ходе работы алгоритма локального поиска. Улучшение решения производится в нескольких районах. Каждый последующий район покрывает большую область поиска. Процедура перемешивания (Shaking) отвечает за получение новой отправной точки для локального поиска, в пределах каждого из районов. Когда в результате локального поиск было найдено лучшее решение, чем то, что было зафиксировано ранее, мы его запоминаем, и алгоритм VNS начинает свой поиск с первого района. Для завершения работы алгоритма можем использовать различные критерии остановки: время вычисления, число повторений, и т.д. Шаги базового алгоритма VNS представлены далее.

Алгоритм 1: Базовый VNS-алгоритм

1 **Определение начального решения** v
2 **Присвоим** $t = 1$
3 **до тех пор, пока** не выполнено условие остановки алгоритма **выполнять**
4 **Shaking.** Выбор параметра t. Построение решения v' в $N^{(t)}$
5 **Локальный поиск.** Применяется метод локального поиска с начальным решением v'. На выходе получаем локальный оптимум v''
6 **Сохранение.** Сохранение полученного решения и переход к следующей окрестности.
7 **конец**

3.2.2 Применение алгоритма VNS

Ключевые компоненты решения проблемы следующие. Для любого решения v существует подмножество $I(v) \in I$ - множество открытых дистрибьюторских центров. Таким образом, решение v частично характеризуется подмножеством $I(v)$.

Для того чтобы адаптировать VNS-алгоритм к поставленной ранее задаче, мы должны выполнить следующую последовательность действий:

а) Создать структуру окрестностей.

б) Выбрать способ выбора решения в окрестности в пункте 4 алгоритма.

в) Выбрать способ локального поиска в пункте 5 алгоритма.

г) Определить условие остановки работы алгоритма.

Для решения задачи, возьмем за основу предложенную в исследовании [14] модификацию алгоритма VNS.

Аналогично данному методу, мы введем окрестности трех типов. Обозначим их $N^{(1)}$, $N^{(2)}$ и $N^{(3)}$.

Решение v' в окрестности типа $N^{(1)}$ получается добавлением элемента из подмножества решений $I \setminus I(v)$ в $I(v)$, то есть открываем дополнительный дистрибьюторский центр в дополнение к уже работающим.

Решение v' в окрестности типа $N^{(2)}$ получаем путем удаления элемента из $I(v)$, то есть, мы закрываем один из дистрибьюторских центров.

Решение v' в окрестности типа $N^{(3)}$ получается заменой элемента из подмножества решений $I \setminus I(v)$ элементом из множества $I(v)$, то есть открывается дополнительный дистрибьюторский центр, при этом закрывается один из ранее открытых.

Процедура обмена элементов в множествах $I \setminus I(v)$ и $I(v)$ будет также использоваться в работе локального поиска.

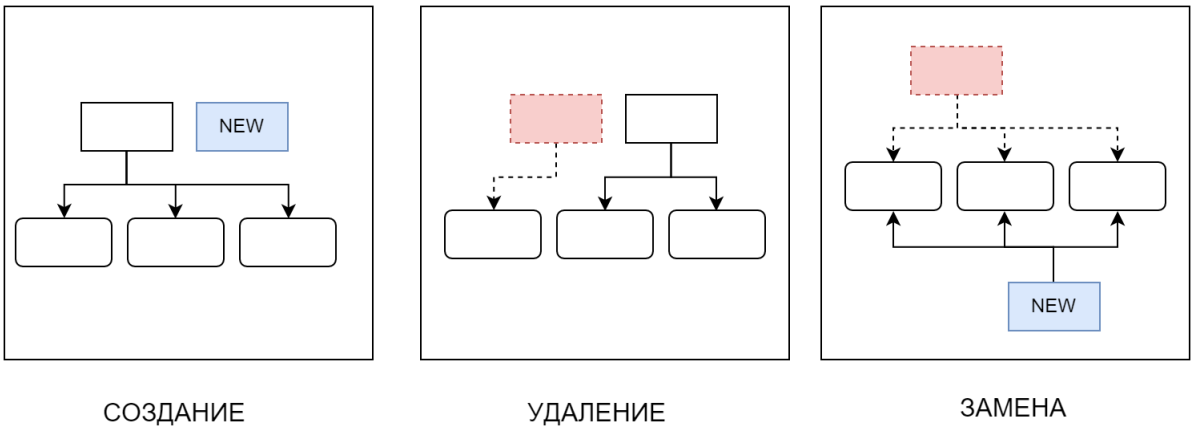

Рисунок 3.1 — Иллюстрация получения новых окрестностей в алгоритме VNS

Относительно данных структур окрестностей должны быть решены две ключевые подзадачи:

а) Какие ритейлеры должны быть связаны с дистрибьюторскими центрами, когда он включен в дистрибьюторскую сеть.

б) К каким дистрибьюторам должны быть подключены ритейлеры когда дистрибьюторы в процессе алгоритма удаляются из сети после удаления или перемещения.

В обоих случаях мы должны обращать внимание на затраты на транспортировку груза между ритейлерами и РЦ, а также затраты на организацию дистрибьюторского центра, затраты на хранение.

Мы решим первую подзадачу следующим образом. Предположим, что мы хотим включить дистрибьютора $s' \in I$ в сеть. Будет уместно подключить ритейлера $d \in J$ к данному дистрибьютору взамен текущего клиента s, если суммарные издержки от такой замены уменьшаются. Однако, мы можем наблюдать, что такая замена может привести к несоблюдению ограничений вместимости для терминала дистрибьюторского центра, к которому был подключен данный клиент. То есть к s'. Данное ограничение соблюдается путем простой проверки на вместимость дистрибьютора s' перед подключением к нему клиента после осуществления проверки на уместность данной операции (проверки Total Costs).

Для решения второй подзадачи мы последовательно рассматриваем каждый из подключенных в данный момент дистрибьюторов и выбираем наилучший в плане минимизации затрат и подключаем данного клиента к нему.

В обоих случаях далее выбирается наилучшая конфигурация дистрибьюторской сети, то есть удовлетворяющая всем условиям ограничений и при этом имеющая наименьшие суммарные затраты.

Опишем, как будут использоваться те три типа окрестностей, которые были введены ранее.

Пусть v^* - вектор, содержащий лучшее решение, которое на текущий момент было вычислено в ходе работы алгоритма. Процесс начинается с множества $N = \{N^{(1)}, N^{(2)}, N^{(3)}\}$

Вместо обычного выбираемого $t = 1$ на шаге 4 VNS - алгоритма мы случайным образом выбираем $t \in \{1, 2, 3\}$. На шаге 5 выполняется локальный поиск, в ходе которого мы получаем решение v''. Затем на шаге 6 мы, если

решение v'' лучше, чем v^*, то мы обновляем лучшее решение $v^* = v''$. В противном же случае:

а) Если $|N| > 1$, мы удаляем $N^{(t)}$ из N

б) Если $|N| = 1$, то $N = \{N^{(1)}, N^{(2)}, N^{(3)}\}$

Следующая выбранная окрестность будет случайно выбираться из N.

Алгоритм будет остановлен в тот момент, когда решение не было улучшено в течение δ итераций.

Псевдокод предложенного алгоритма предложен ниже:

Алгоритм 2: Модифицированный VNS-алгоритм

```
1  Построение начального решения. Построим начальное решение v согласно
     шагам, описанным в разделе 3.2.3.
2  Присвоим TC' = TC(v), v* = v и N = {N^(1), N^(2), N^(3)}, i = 0
3  до тех пор, пока i < δ выполнять
4      Shaking. Случайный выбор N^(t) в N. Строим лучшее возможное решение в
         данной окрестности.
5      Локальный поиск. Применяем к полученному в предыдущем шаге решению
         метод локального поиска, описанного в разделе 3.2.4. Полученный оптимум
         обозначим как v''
6      Сохранение. Сохранение полученного решения и переход к следующей
         окрестности.
7      если TC(v'') < TC(v*) тогда
8          v* = v'';
9          N = {N^(1), N^(2), N^(3)};
10         i = 0;
11     иначе
12         если |N| > 1 тогда
13             удаляем N^(t) из N;
14         иначе
15             N = {N^(1), N^(2), N^(3)};
16         конец
17     конец
18     Присвоим i = i + 1
19 конец
```

3.2.3 Построение начального решения

Построение начального решения v происходит в два этапа.

На первом этапе на каждом шаге мы, используя метод Монте-Карло, выбираем ритейлера d еще не подключённого к дистрибьюторскому центру и подключаем его к тому дистрибьюторскому центру, при выборе которого издержки минимальны, при этом, должны выполняться ограничения на вместимость склада дистрибьютора.

На втором этапе мы на каждом шаге аналогичным образом, случайно выбираем ритейлера и пытаемся сократить суммарные расходы путем переключения ритейлера с одного дистрибьютора на другой. На данном этапе каждый из ритейлеров может быть перераспределен лишь единожды. Затратами, которые могут изменить свое значения при подобных перераспределениях будут: затраты на перевозку, затраты на хранение, затраты на открытие дистрибьютора. Если быть точным - изменение поставщика для ритейлера может привести к следующим изменениям:

а) Сократить организационные затраты на открытие того или иного дистрибьютора

б) Изменить затраты на хранение как для последующего поставщика клиента, так и для того, кто работал с данным клиентом ранее.

3.2.4 Локальный поиск

Как уже известно, после шага 4 будет получено некоторое решение v'. Для локального поиска используем идею замены дистрибьюторов, то есть в процессе работы локального поиска будет производиться перестановка уже открытого дистрибьютора $i \in I$ с в данный момент закрытым $i' \in I \setminus I(i)$. Количество итераций локального поиска является заранее определенным параметром θ - после выполнения θ операций без улучшения текущего наилучшего решения v'' работа локального поиска прекращается, и далее происходит переход к следующему шагу алгоритма VNS.

Одна из основных задач локального поиска является поиск соседнего

решения на каждой из итераций. Так как целевая функция $TC(v)$ является довольно сложной для вычисления, то работа алгоритма при условии выполнения подсчета значения целевой функции на каждом шаге локального поиска потребует значительных временных ресурсов. Поэтому необходимо сократить количество вычислений такой функции, или, как минимум, упростить её.

Для этого введём упрощенную функцию TC', которая будет вычисляться намного быстрее.

Данная функция будет показывать, какие дистрибьюторы должны быть закрыты, а какие из ранее не включенных в сеть должны быть открыты. Таким образом, при открытии одного дистрибьютора взамен другого мы получим возможность сократить расходы, и, тем самым, улучшить решение.

Другими словами, функция TC' используется для определения подмножества $I^K(v) \in I(v)$, которое содержит K худших на данный момент дистрибьюторов.

Определим функцию $TC'(i)$ для всех $i \in I$.

$$TC'(i) = \min_{i' \in I \setminus I(v)} \left[\sum_{j \in J} (t_{i'j} - t_{ij}) + \sum_{j \in J} (h_{i'} - h_i) \right] \tag{3.1}$$

Данная функция характеризует изменения в издержках дистрибьюторской сети, которые могут произойти, в случае если некоторый отдельный дистрибьютор i будет заменен другим - i'.

Если данная величина отрицательная, то это означает, что, по крайней мере дистрибьютор i' может снабжать клиентов, ранее обслуживаемых центром i, продукцией с меньшими издержками.

Таким образом, мы получаем подмножество $I^K(v)$ путем упорядочивания дистрибьюторов $I(v)$ в возрастающем порядке согласно функции $TC'(i)$ и выделением среди них K первых элементов.

Затем мы вычисляем стоимость каждой замены дистрибьютора $i \in I^K(v)$ на дистрибьютора $i' \in I \setminus I(v)$, используя уже оригинальную функцию $TC(v)$.

Выполнив замены, которые имеют набольшую стоимость, мы получаем

новое решение, которое может улучшить текущее.

Если изменения лучшего решения в процессе локального поиска в данном районе не происходят в течение θ итераций, то процедура локального поиска прекращает свою работу, а алгоритм после проверки и возможного обновления наилучшего решения переходит к следующему району поиска.

Псевдокод данного алгоритма локального поиска представлен ниже.

Алгоритм 3: Локальный поиск для модиф. VNS-алгоритма

```
1  Пусть v - решение, поданное на вход локального поиска. Т.е. оно было получено
     ранее.
2  Присвоим i = 0, v'' = v и TC'' = +∞
3  до тех пор, пока i < θ выполнять
4      i = i + 1
5      Вычисляем множество I^K(v) тех дистрибьюторов, которые первыми попадают
         под закрытие по значению TC'
6      Находим лучшее решение v* среди всех v, получающихся путем замены одного
         дистрибьютора на другой
7      если TC(v*) < TC'' тогда
8          v'' = v*
9          TC'' = TC(v*)
10         i = 0
11     конец
12 конец
```

3.3 Метод ветвей и границ

Одним из самых успешных методов решения задач целочисленного программирования является метод ветвей и границ (Branch and Bound). Он был одним из первых методов, решающим подобные задачи.

В данном разделе мы опишем решение задачи путем применения алгоритма ветвей и границ.

Метод ветвей и границ одновременно решает как целочисленную, так и комбинаторную части задачи. Комбинаторная часть задачи решается путем обхода двоичного дерева, в узлах которых мы ищем решения путем отбрасывания условия целочисленности.

Общий алгоритм решения задачи целочисленного программирования предложен в [18]. Приведем адаптированный алгоритм ветвей и границ для решения поставленной ранее задачи.

Первое правило решения задач целочисленного программирования - это ослабление условий целочисленности. Затем мы рассматриваем данную задачу как задачу нелинейного программирования и решаем её. Если получается так, что оптимальное решение задачи нелинейного программирования является целочисленным, то такое решение также будет оптимальным и для задачи целочисленного программирования[19].

Таким образом, мы пытаемся решить задачу целочисленного программирования путем преобразования к задаче нелинейного программирования и решения таковой.

Отбросим условия целочисленности задачи, обозначив целевую функцию через TC, Получаем задачу LP_0. Далее алгоритм ветвей и границ начинает свою работу, решая именно данную задачу.

Нижняя грань, обозначенная в начале работы алгоритма, показывает числовую характеристику лучшего на данный момент решения. Она выражена значением целевой функции в полученном решении.

В процесс работы метода ветвей и границ происходит процесс ветвления - разбиение текущей задачи нелинейного программирования на две задачи, но уже с другими ограничениями на какие-то определенные переменные. Полученные задачи будут помещены в стек и позднее рассмотрены (прозондированы) алгоритмом.

Ниже приведен алгоритм, который будет применен к решению описанной выше задачи[13].

Алгоритм 4: Метод ветвей и границ

Присваиваем $i = 0$, и нижнюю границу $TC^* = -\infty$
до тех пор, пока В стеке есть хотя бы одна задача LP_i, и время работы алгоритма не закончилось **выполнять**
    Выбираем задачу из стека
    Решаем данную задачу без ограничений целочисленности произвольным методом
    Удаляем данную задачу из стека.
    Пусть получено решение (X', Y'), значение функции TC'.
    если решения данной задачи ЛП не существует **тогда**
        далее мы данный узел не рассматриваем и не применяем ветвление.
    конец
    иначе
        если $TC' \leq TC^*$ **тогда**
            Прекращаем поиск решения для данной задачи. Дальнейший поиск лучших решений не принесет.
        конец
        иначе
            если Решение целочисленное **тогда**
                Обновляем лучшее решение и нижнюю границу TC^*.
                Удаляем из стека все задачи с верхней границей меньшей, чем TC^*
            конец
            иначе
                Делим задачу по переменной, которая ранее не участвовала в ветвлении
                Добавляем данные задачи в стек
                Переходим к выбору задачи из стека
            конец
        конец
    конец
конец

В результате получаем решение задачи. При небольших размерностях (количествах ритейлеров и дистрибьюторов) такое решение будет точным решением задачи. Но при увеличении размерности задачи ее решение становится все более трудоемким, а сложность алгоритма мало чем отличается от полного перебора или полного обхода по двоичному дереву, так как, казалось бы, при небольших значениях $|I| = 6$ и $|J| = 14$, мы получаем обход

по 2^{90} узлам, что недопустимо при вычислениях на пользовательских ПК. А метод ветвей и границ, хоть и упрощает задачу, отсекая лишние ветви, но он также требует время на решение задачи нелинейного программирования.

Более точную информацию о размерностях, на которых данный алгоритм применим, мы получим при исследовании работы программной реализации данного алгоритма на ПК.

3.4 Двоичный перебор решений

Двоичный перебор является версией полного перебора всех возможных решений. В силу того, что математическая модель, рассматриваемая в данной работе, имеет только двоичные искомые переменные, и при этом количество решений ограничено, то данный алгоритм также можно использовать для поиска точного решения задачи. Кроме того, вычислительная сложность перебора практически совпадает со сложностью алгоритма ветвей и границ и равна $O(2^{N+NM})$.

Применение алгоритма полного перебора для решения поставленной задачи является нецелесообразным (в этом мы убедимся в ходе эксперимента, описанного в следующей главе). Количество возможных и допустимых решений в предложенной модели велико. Кроме того, количество решений зависит и от входных параметров. Даже незначительное увеличение одной из входных постоянных, характеризующих количество дистрибьюторов или клиентов, уже значительно увеличит время перебора.

Программный код реализации данного алгоритма приведен в приложении 2.

ГЛАВА 4

ИССЛЕДОВАНИЕ АЛГОРИТМОВ

В данном разделе мы проведем вычислительные исследования, необходимые для оценки производительности алгоритмов, предложенных в предыдущем разделе.

4.1 Программная реализация

Все алгоритмы мы реализовали, используя самые современные средства разработки и языки программирования.

Результаты данных исследований приведены далее, а код программной реализации доступен для просмотра в приложениях 1-3.

4.1.1 Требования к аппаратному и программному обеспечению

Программная реализация алгоритма ветвей и границ для обоих вариантов постановки задачи осуществлялось в среде математического моделирования MATLAB на ПК с четырехъядерным 1.9 ГГц процессором и 4 Гб оперативной памяти в 64-разрядной операционной системе Windows 10.

Другие два алгоритма были реализованы на языке C++ и тестировались в среде Microsoft Visual Studio 2012.

4.1.2 Требования к входным данным

Входные данные должны содержать информацию обо всех необходимых для моделирования параметрах (массив розничных цен, массив оптовых цен, массив стоимостей хранения по дистрибьюторам, массив спроса по клиентам и т.д.). Импорт данных осуществляется из файла формата .txt.

Входные данные были сгенерированы случайно и автоматически, исходя из вполне реальных ограничений на показатели стоимостей и спроса для

автомобильного завода (стоимость продукции, стоимость транспортировки, стоимость хранения, спрос на продукцию).

4.1.3 Требования к выходным данным

Выходные данные должны содержать массив полученных значений x_i, двумерный массив y_{ij}, время выполнения алгоритмической части программы. Экспорт полученных результатов осуществляется в файл формата .txt.

4.2 Анализ работы алгоритмов

Для того чтобы понять какие алгоритмы применимы к решению данной задачи, и какие из них имеют перспективы для работы на больших размерностях, рассмотрим результаты исполнения реализаций алгоритмов, приведенных в данной работе на нескольких наборах входных данных.

В таблицах, приведенных ниже, символами N и M обозначены количество дистрибьюторов и клиентов соответственно. Символом Σ - суммарное количество искомых переменных. TC^* - полученное оптимальное значение целевой функции $TotalCosts$. В колонке «Отклонение» мы запишем отклонение значение целевой функции для полученного решения от значения этой функции для точного решения для данного набора входных данных. Очевидно, что для точных алгоритмов эта величина равна нулю.

Тестирование на входных данных, в результате которых задача решается более, чем для 20 переменных не было полностью завершено в силу того, что выполнение программы превысило максимально допустимое время на выполнение, ранее объявленное в 500 секунд.

Таким образом, в таблице 4.1 мы видим, что при дальнейшем увеличении количества искомых переменных применение алгоритмов ветвей и границ нецелесообразно на пользовательском ПК, так как количество операций, выполняемых компьютером для такого количества решений довольно велико.

Таблица 4.1 — Сравнение работы точных алгоритмов на различных наборах входных данных

N	M	Σ	Метод ветвей и границ (Branch and Bound)			Переборный алгоритм		
			Решение TC^*	Отклонение	Время работы	Решение TC^*	Отклонение	Время работы
1	3	4	252,650,800	0	0,47	252,650,800	0	0,02
2	2	5	267,601,620	0	0,62	267,601,620	0	0,02
2	3	8	305,205,560	0	0,86	305,205,560	0	0,04
2	4	10	394,850,740	0	1,47	394,850,740	0	0,18
3	5	18	588,001,050	0	93,10	588,001,050	0	49,61
4	4	20	519,892,340	0	464.99	519,892,340	0	283,96
7	13	98	-	-	-	-	-	-
10	27	280	-	-	-	-	-	-
16	44	720	-	-	-	-	-	-
25	50	1275	-	-	-	-	-	-
35	75	2660	-	-	-	-	-	-
50	100	5050	-	-	-	-	-	-

Рассмотрим выполнение реализации алгоритма VNS на пользовательском ПК и запишем все полученные результаты в таблицу 4.2.

Таблица 4.2 — Сравнение работы алгоритма Branch and Bound и алгоритма VNS на различных наборах входных данных

N	M	Σ	Метод ветвей и границ (Branch and Bound)			VNS		
			Решение TC^*	Отклонение	Время работы	Решение TC^*	Отклонение	Время работы
1	3	4	252,650,800	0	0,47	252,650,800	0	0,2
2	2	5	267,601,620	0	0,62	267,601,620	0	0,3
2	3	8	305,205,560	0	0,86	305,205,560	0	0,3
2	4	10	394,850,740	0	1,47	394,850,740	0	0,3
3	5	18	588,001,050	0	93,10	588,001,050	0	0,5
4	4	20	519,892,340	0	464.99	519,892,340	0	0,4
7	13	98	-	-	-	1,205,994,610	-	1,1
10	27	280	-	-	-	3,299,122,398	-	2,3
16	44	720	-	-	-	5,249,525,820	-	5,2
25	50	1275	-	-	-	5,757,285,744	-	13,1
35	75	2660	-	-	-	6,380,371,150	-	25,8
50	100	5050	-	-	-	8,618,480,106	-	47,4

Как мы можем видеть, время работы алгоритма VNS значительно меньше. Несмотря на то, что данный алгоритм не гарантирует нам получение точного решения, мы можем его использовать для получения наиболее оптимального. Кроме того, стоит заметить, что на входных данных небольших объемов решения, полученные в ходе работы алгоритма VNS, совпадают с точными решениями.

Проведем еще один, более глубокий анализ работы алгоритма VNS. Мы знаем, что время работы алгоритма зависит от:

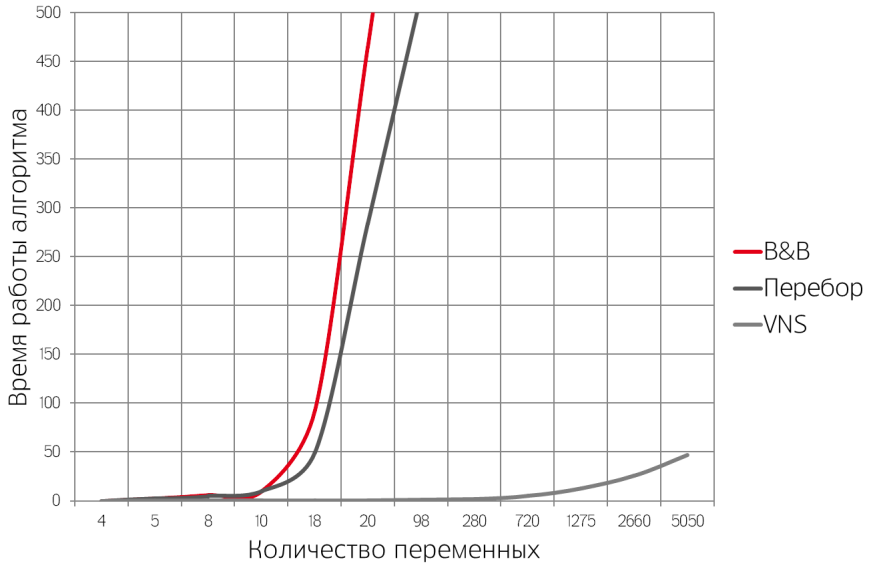

Рисунок 4.1 — График зависимости времени работы алгоритма от количества искомых переменных

- Количества итераций без изменения оптимального решения. (Значение δ). Данное значение устанавливается в алгоритме.
- Значения θ в процедуре локального поиска.

В описанных выше экспериментах наш алгоритм прекращал свою работу, в случае если в течение $\delta = 73$ итераций не происходило улучшения решения, а значение $\theta = 9$.

Попробуем регулировать данные параметры, и посмотрим, как будет зависеть полученное решение от данных параметров. Пусть тестирование алгоритма проходит на одном наборе данных, в котором $N = 10$ дистрибьюторов, $M = 27$ клиентов. Всего переменных - 280.

Исходя из результатов данного эксперимента, можем сделать вывод, что необязательно заранее устанавливать большие значения параметров δ и θ, так как их увеличение значительно увеличивает время работы программы, при этом решение остается тем же. Таким образом, подобрав наиболее подходящие параметры, мы можем значительно сократить время работы программы, не теряя при этом в эффективности.

Таблица 4.3 — Время работы алгоритма VNS на одинаковых входных данных при изменеии параметров θ и δ. $N = 10$, $M = 27$, $\Sigma = 280$.

δ	θ	TC^*	$Profit$	$CPU(s)$
1	1	3,264,599,100	0	0,00
1	5	3,264,599,100	0	0,00
5	1	3,264,599,100	0	0,01
10	5	3,299,122,398	+1.1%	0,03
10	10	3,299,122,398	+1.1%	0,4
50	10	3,299,122,398	+1.1%	1,7
10	50	3,299,122,398	+1.1%	1,7
50	50	3,299,122,398	+1.1%	8,7
100	100	3,299,122,398	+1.1%	34,9
250	250	3,299,122,398	+1.1%	218,3

4.3 Выводы

Используя программные реализации описанных в предыдущей главе алгоритмов, был осуществлён эксперимент, в результате которого мы сделали выводы относительно возможности использования данных алгоритмов на реальных данных.

Мы заметили, что алгоритм ветвей и границ и общий переборный алгоритм не могут быть использованы для моделирования процессов в реальных дистрибьюторских сетях, несмотря на то, что они дают возможность получить точное решение поставленной задачи.

Алгоритм VNS показал свою производительность на входных данных, как малой размерности, так и таких входных данных, искомых переменных для которых превышает 5000. Стоит заметить, что на входных данных малой размерности полученное данным алгоритмом решение полностью совпадало с точным решением, что говорит о том, что на входных данных большей размерности может быть также получено точное решение, хоть алгоритмом это не гарантировано.

В ходе дополнительного исследования реакции работы алгоритма VNS на внутренние параметры δ и θ было выяснено, что влияние данных параметров на полученное решение незначительно, что означает, что можно ускорить работу алгоритма, изменив параметры до минимально допустимого значения. Данное значение может быть получено для каждого набора

входных данных в ходе дополнительного эксперимента.

Таким образом, алгоритм VNS может применяться для моделирования процессов реальных дистрибьюторских сетей с входными данными средней и большой размерности.

ГЛАВА 5

ЗАКЛЮЧЕНИЕ

В данной работе была исследована задача максимизации прибыли для дистрибьюторской сети с одноканальными поставками.

В ходе работы:

- Построены математические модели дистрибьюторских сетей с каналами первого уровня
- Изучены алгоритмы решения задачи оптимизации дистрибьюторских сетей
- Осуществлена программная реализация алгоритмов
- Проведено исследование реализованных алгоритмов на применение к решению задач с входными данными различных размеров

Задача, которая была сформулирована для решения, относится к типу смешанной задачи нелинейного программирования. Решение данной задачи осуществлялось с использованием алгоритмов VNS (Variable Neigborhood Search) и метода ветвей и границ.

Исследование программной реализации данных алгоритмов показало, что **алгоритм VNS существенно более производителен на возрастающих входных данных в сравнении с методом ветвей и границ**.

Подобный подход можно применить и к оптимизации многоканальных дистрибьюторских сетей, а также к решению других подобных задач. А алгоритмы, рассмотренные при решении данных задач и приведенные в данной работе, могут быть интересны для дальнейшего изучения.

Стоит также заметить, что данную задачу можно решать не только для определения расположения региональных или международных представительств, но и на уровне города или, даже, небольшого района. В качестве дистрибьюторов может выступать множество торговых точек, гипермаркетов, расположение которых нам необходимо определить для получения наибольшей выгоды.

Таким образом, применение решения данной задачи не ограничено только одной системой, а может использоваться в различных компаниях и городских службах.

СПИСОК ЛИТЕРАТУРЫ

1. Деловой мир: Построение дистрибьюторской сети http://delovoymir.biz/ru/articles/view/?did=10594
2. Голубин Е. Дистрибуция. Формирование и оптимизация каналов сбыта. М.: Вершина, 2006. 136 с.
3. Ahmadi-Javid A., Hoseinpour P. A location-inventory-pricing model in a supply chain distribution network with price-sensitive demands and inventory-capacity constraints // Transportation Research, 2015. Vol. 82, P. 238–255.
4. BMW GROUP FINANCIAL SERVICES IN THE AMERICAS https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/en/2015/events-&-presentations/capital-markets/Mr_Robinson_ BMW_ GROUP_ Financial_ Services_ Americas.pdf
5. Алексеев А. А., Ширяев С. А., Гудков В. А., Гронин Д. П. Математическое моделирование дистрибьюторских сетей автосервиса // Известия Волгоградского государственного технического университета, 2013. Т. 6, Вып. 10(113). С. 47-50.
6. Федорова Л. П., Тимофеев С. В. Логистика в торговой сети региональной потребительской кооперации // Вестник Чувашского университета, 2010. Вып. 2.
7. Ahmadi-Javid A., Hoseinpour P. Incorporating location, inventory and price decisions into a supply chain distribution network design problem // Computers & Operations Research, 2015. Vol. 56, P. 110–119.
8. Validi S., Bhattacharya, A., Byrne P. J. A solution method for a two-layer sustainable supply chain distribution model // Computers & Operations Research, 2015. Vol. 54, P. 204–217.
9. Alumur S., Kara B.Y. A new model for the hazardous waste location-routing problem // Computers & Operations Research, 2007. Vol. 34, No 5. P. 1406–1423.

10. Kulcar T. Optimizing solid waste collection in Brussels // European Journal of Operational Research, 1996. Vol. 90, P. 71-77.
11. Burgschweiger, J., Gnadig, B., and Steinbach, M. Optimization models for operative planning in drinking water networks // Optimization and Engineering, 2008. Vol. 10, P. 43–73.
12. Bock, H. and Longman, R. Computation of optimal controls on disjoint control sets for minimum energy subway operation. // Proceedings of the American Astronomical Society Symposium on Engineering Science and Mechanics, 1982. P. 949–972.
13. Leyffe S. A. Deterministic Methods for Mixed Integer Nonlinear Programming, PhD Thesis, Department of Mathematics and Computer Science, University of Dundee, 1993
14. Houssam A., Alain M., Zufferey N., Makeeva P., A variable neighborhood search heuristic for the design of multicommodity production–distribution networks with alternative facility configurations // Operations Research-Spektrum, 2008
15. Celebi D. Inventory control in a centralized distribution network using genetic algorithms: A case study // Computers & Industrial Engineering, 2015. Vol. 87, P. 532–539.
16. Popovic D., Vidovic M., Radivojevic G. Variable Neighborhood Search heuristic for the Inventory Routing Problem in fuel delivery // Expert Systems with Applications, 2012. No. 39, P. 13390-13398.
17. Salhi S., Imran A., Wassan N.A. The multi-depot vehicle routing problem with heterogeneous vehicle fleet: Formulation and a variable neighborhood search implementation // Computers & Operations Research, 2014. Vol. 52, P. 315–325.
18. Таха Х.А. Введение в исследование операций. М.: Вильямс, 2005. 912 с.
19. Косоруков О. А., Мищенко А. В., Исследование операций. М.: Экзамен, 2003. 448 с.
20. OPTI Toolbox Wiki http://www.i2c2.aut.ac.nz/Wiki/OPTI/index.php

МЕТОД ВЕТВЕЙ И ГРАНИЦ

Пример программной реализации алгоритма ветвей и границ для заданных начальных данных при $n = 2, m = 2$, где $n = 2$ - количество дистрибьюторов, $m = 2$ - количество клиентов. Для реализации алгоритма был использован пакет для оптимизации "MATLAB OPTI ToolBox"[20].

```
clear;
clc;

% Source Data

p = [190 175];
f = [100 120];
wp = [120 130];
t = [7 9; 6 8];
h = [1 1];
a = [0.5 0.7];
e = [100 100];
c = [100 200];
d = [24 29];

% Object Function

Income = @(x) x(3)*d(1)*p(2)+x(4)*d(2)*p(2)+x(5)*d(1)*p(1)+x(6)*d(2)*p(2);
LC = @(x) f(1)*x(1) + f(2)*x(2);
WPC = @(x) x(3)*wp(1)*d(1)+x(4)*wp(1)*d(2)+x(5)*wp(2)*d(1)+x(6)*wp(2)*d(2);
DTC = @(x) x(3)*t(1,1)*d(1)+x(4)*t(1,2)*d(2) + x(5)*t(2,1)*d(2) + x(6)*t(2,2)*
    d(2);
HC = @(x) 0.5*(x(3)*h(1)*d(1)+x(4)*h(1)*d(2)+x(5)*h(2)*d(1)+x(6)*h(2)*d(2));
PTC = @(x) e(1)+e(2)+a(1)*(d(1)*x(3)+d(2)*x(4))+a(2)*(d(1)*x(5)+d(2)*x(6));
fun = @(x) -(Income(x) - LC(x) - WPC(x) - DTC(x) - HC(x) - PTC(x));

% Constraints

A = [0 0 1 0 1 0;
     0 0 0 1 0 1;
     -1 0 1 0 0 0;
     -1 0 0 1 0 0;
     0 -1 0 0 1 0;
     0 -1 0 0 0 1;
     0 0 d(1) d(2) 0 0;
     0 0 0 0 d(1) d(2);];
b = [1; 1; 0; 0; 0; 0; c(1); c(2)];

nlcon = @(x) 0;
nlrhs = 0;
nle = 0;

lb = [0;0;0;0;0;0];
ub = [1;1;1;1;1;1];
```

```
% Type of constraints

xtype = 'IIIIII';

% Initial Point
x0 = [99;99;99;99;99;99];

Opt = opti('fun',fun,'nlmix',nlcon,nlrhs,nle,'ineq',A,b,'bounds',lb,ub,...
           'xtype',xtype)
[x,fval,exitflag,info] = solve(Opt,x0)
```

ПЕРЕБОРНЫЙ АЛГОРИТМ

В данном приложении приведена упрощенная реализация алгоритма двоичного перебора на языке программирования C++ для общего случая.

```
#include <iostream>
#include <vector>
#include <cstdio>

using namespace std;

int N,M;
vector<int> p, f, wp, h, a, e, c, d;
vector<vector<int>> t;

double TotalCosts(vector<int> x, vector<vector<int>> y)
{
        double TC = 0;

        double TotalIncome = 0,
                LaunchCosts = 0,
                WholesalePurchaseCosts = 0,
                DistributorTransportationCosts = 0,
                HoldingCosts = 0,
                ProducerTransportationCosts = 0;

        for (int i=0; i<N; i++)
                for (int j=0; j<M; j++)
                        TotalIncome += p[j]*d[j]*y[i][j];

        for (int i=0; i<N; i++)
                LaunchCosts += f[i]*x[i];

        for (int i=0; i<N; i++)
                for (int j=0; j<M; j++)
                        WholesalePurchaseCosts += wp[i]*d[j]*y[i][j];

        for (int i=0; i<N; i++)
                for (int j=0; j<M; j++)
                        DistributorTransportationCosts += t[i][j]*d[j]*y[i][j
                           ];

        for (int i=0; i<N; i++)
                for (int j=0; j<M; j++)
                        HoldingCosts += (double)0.5*h[i]*d[j]*y[i][j];

        for (int i=0; i<N; i++)
        {
                ProducerTransportationCosts += e[i];
                for (int j=0; j<M; j++)
                        ProducerTransportationCosts += a[i]*d[j]*y[i][j];
        }
```

```
        TC = TotalIncome - LaunchCosts - WholesalePurchaseCosts -
           DistributorTransportationCosts -
                HoldingCosts - ProducerTransportationCosts;
        return TC;
}
bool CheckConstraints(vector<int> x, vector<vector<int>> y)
{
        bool flag = 0;

        ...

        return flag;
}
int main()
{
        ...

        ReadData();

        ...

        int j;
        while(1)
        {
                j = 0;
                while (temp[j] == 1)
                temp[j++] = 0;
                ++temp[j];
                if (j >= N+N*M) break;

                x.clear();
                y.clear();
                for (int i = 0; i < N; i++)
                        x.push_back(temp.at(i));

                int pos = N;
                for (int i = 0; i < N; i++)
                {
                        vector<int> tmp;
                        for (int k = 0; k < M; k++)
                        {
                                tmp.push_back(temp.at(pos));
                                pos++;
                        }
                        y.push_back(tmp);
                }

                if (!CheckConstraints(x,y) && TotalCosts(x,y)>maxf)
                        {
                                maxf = TotalCosts(x,y);
                                maxx = x;
                                maxy = y;
                        }
        }

        ResultExport(maxf,maxx,maxy)
}
```

АЛГОРИТМ VNS

В данном приложении приведена реализация алгоритма VNS на языке программирования C++ для общего случая.

```
#include <iostream>
#include <vector>
#include <cstdio>
#include <algorithm>
#include <ctime>
#include <iomanip>

using namespace std;

int N,M;
vector<int> p, f, wp, h, a, e, c, d;
vector<vector<int>> t;

int Random(int a, int b)
{
        return (b - a == 0)?0:(rand() % (b - a) + a);
}

void toReadData()
{
        ...
}

double TotalCosts(vector<int> x, vector<vector<int>> y)
{
        double TC = 0;

        double TotalIncome = 0,
                   LaunchCosts = 0,
                   WholesalePurchaseCosts = 0,
                   DistributorTransportationCosts = 0,
                   HoldingCosts = 0,
                   ProducerTransportationCosts = 0;

        for (int i = 0; i < N; i++)
                for (int j = 0; j < M; j++)
                        TotalIncome += p[j] * d[j] * y[i][j];

        for (int i = 0; i < N; i++)
                LaunchCosts += f[i] * x[i];

        for (int i = 0; i < N; i++)
                for (int j = 0; j < M; j++)
                        WholesalePurchaseCosts += wp[i] * d[j] * y[i][j];

        for (int i = 0; i < N; i++)
                for (int j = 0; j < M; j++)
```

```
                        DistributorTransportationCosts += t[i][j] * d[j] * y[i
                            ][j];

        for (int i = 0; i < N; i++)
                for (int j = 0; j < M; j++)
                        HoldingCosts += (double) 0.5 * h[i] * d[j] * y[i][j];

        for (int i = 0; i < N; i++)
        {
                ProducerTransportationCosts += e[i];
                for (int j = 0; j < M; j++)
                        ProducerTransportationCosts += a[i] * d[j] * y[i][j];
        }

        TC = TotalIncome - LaunchCosts - WholesalePurchaseCosts -
                DistributorTransportationCosts - HoldingCosts
                - ProducerTransportationCosts;
        return TC;
}

pair<double, int> TotalCosts2(vector<int> x, vector<vector<int>> y, int
    openedDistr)
{
        double min_value = 0;
        int min_ndist = -1;

        for (int i = 0; i < N; i++)
                if (i != openedDistr && x.at(i) == 0)
                {
                        double nakop = 0;
                        for (int j = 0; j < M; j++)
                        {
                                nakop += (t.at(i).at(j) - t.at(openedDistr).at
                                    (j));
                                nakop += (h.at(i) - h.at(openedDistr));
                        }
                        if (nakop < min_value)
                        {
                                min_value = nakop;
                                min_ndist = i;
                        }
                }

        pair<double, int> result;
        result.first = min_value;
        result.second = min_ndist;

        return result;
}

bool CheckConstraints(vector<int> x, vector<vector<int>> y)
{
        bool flag = 0;
        for (int j = 0; j < M; j++)
        {
                int yy = 0;

                for (int i = 0; i < N; i++)
```

```
                        yy += y[i][j];

                if (yy > 1) flag = 1;
        }

        for (int i = 0; i < N; i++)
                for (int j = 0; j < M; j++)
                        if (y[i][j] > x[i]) flag = 1;

        for (int i = 0; i < N; i++)
        {
                double q = 0;

                for (int j = 0; j < M; j++)
                        q += d[j] * y[i][j];

                if (q > c[i]) flag = 1;
        }

        return flag;
}

void ChangeDistr(int retailer, int newDistributor, int oldDistributor,
                                vector<int>* x, vector<vector<int>>* y)
{
        swap(y->at(newDistributor).at(retailer),
                y->at(oldDistributor).at(retailer));

        for (int i = 0; i < N; i++)
        {
                int k=0;

                for(int j = 0; j < M; j++)
                        k += y->at(i).at(j);

                x->at(i) = (k)?1:0;
        }
}

void BuildInitialSolution(vector<int>* x, vector<vector<int>>* y)
{
        for (int j = 0; j < M; j++)
        {
                int tempLinking = -1;
                double tempBest = 0;

                for (int i = 0; i < N; i++)
                {
                        int wasZero = 0;
                        if (x->at(i) == 0)
                        {
                                wasZero = 1;
                                x->at(i) = 1;
                        }

                        y->at(i).at(j) = 1;

                        double tempCost = TotalCosts(*x,*y);
```

```
                        if (tempCost > tempBest && !CheckConstraints(*x,*y))
                        {
                                tempBest = tempCost;
                                tempLinking = i;
                                y->at(i).at(j) = 0;
                                if (wasZero)
                                        x->at(i) = 0;
                        }
                        else
                        {
                                y->at(i).at(j) = 0;
                                if (wasZero)
                                        x->at(i) = 0;
                        }
                }
                if (tempLinking >= 0)
                {
                        x->at(tempLinking) = 1;
                        y->at(tempLinking).at(j) = 1;
                }
        }

        double currentBestTC = TotalCosts(*x,*y);
        for (int j = 0; j < M; j++)
        {
                int currentDistr = -1;

                for (int i=0; i<N; i++)
                        if (y->at(i).at(j)) currentDistr = i;

                for (int i = 0; i < N; i++)
                {
                        if (!y->at(i).at(j) && currentDistr != -1)
                        {
                                ChangeDistr(j, currentDistr, i, x, y);

                                if (TotalCosts(*x,*y) > currentBestTC)
                                        currentBestTC = TotalCosts(*x,*y);
                                else ChangeDistr(j, i, currentDistr, x, y);
                        }
                }
        }
}

void RestoreNeigboursArray(vector<int>* Neighbours)
{
        Neighbours->clear();
        Neighbours->push_back(1);
        Neighbours->push_back(2);
        Neighbours->push_back(3);
}

void FullDistrExchange(int a, int b, vector<int>* x, vector<vector<int>>* y)
{
        if (a != -1 && b != -1)
        {
                swap(x->at(a), x->at(b));
                for (int i = 0; i < M; i++)
```

```
                        swap(y->at(a).at(i), y->at(b).at(i));
        }
}

void LocalSearch(vector<int>* x, vector<vector<int>>* y)
{
        int theta = 0;
        int k = 1;

        while (theta < 9)
        {
                theta++;
                vector<double> tc2Total;
                vector<int> tc2Zamena;

                for (int i =0; i<N; i++)
                        if (x->at(i)==1)
                        {
                                pair<double,int> result = TotalCosts2(*x,*y,i)
                                    ;
                                tc2Total.push_back(result.first);
                                tc2Zamena.push_back(result.second);
                        }
                        else
                        {
                                tc2Total.push_back(0);
                                tc2Zamena.push_back(-1);
                        }

                double minTC = 0;
                int minPosOpen = -1, minPosClose = -1;
                for (int i = 0; i<N; i++)
                {
                        if (minTC > tc2Total.at(i))
                        {
                                minTC = tc2Total.at(i);
                                minPosOpen = tc2Zamena.at(i);
                                minPosClose = i;
                        }
                }

                double TCBeforeExchanging = TotalCosts(*x,*y);
                if (!CheckConstraints(*x,*y))
                        FullDistrExchange(minPosClose, minPosOpen, x, y);
                double TCAfterExchanging = TotalCosts(*x, *y);

                if (TCBeforeExchanging >= TCAfterExchanging)
                        FullDistrExchange(minPosOpen, minPosClose, x, y);
                else theta = 0;
        }
}

void Shaking(vector<int>* x, vector<vector<int>>* y, vector<int>* nb)
{
        int randPos = Random(0, nb->size()-1);
        swap(nb->at(0), nb->at(randPos));
        int action = nb->at(0);
        nb->erase(nb->begin());
```

```
if (action == 1)
{
        bool f = 0;
        for (int i = 0; i < N; i++)
                if (x->at(i) == 0 && !f)
                {
                        x->at(i) = 1;
                        f = 1;
                }
}
else if (action == 2)
{
        int posToClose = Random(0,N-1);

        int i = 0;
        while (posToClose > 0)
        {
                if (x->at(i) == 1)
                        posToClose--;

                if (!posToClose)
                {
                        x->at(i) = 0;
                        for (int j = 0; j < M; j++)
                        y->at(i).at(j) = 0;
                        break;
                }

                i++;
                if (i == N)
                        i = 0;
        }
}
else
{
        int a = -1, b = -1, i = 0;

        int posToOpen = Random(0,N-1);

        int posToClose = Random(0,N-1);

        while (posToOpen > 0)
        {
                if (x->at(i) == 0)
                        posToOpen--;

                if (!posToOpen)
                {
                        a = i;
                        break;
                }

                i++;
                if (i == N)
                        i = 0;
        }
```

```
                while (posToClose > 0)
                {
                        if (x->at(i) == 1)
                                posToClose--;

                        if (!posToClose)
                        {
                                b = i;
                                break;
                        }

                        i++;
                        if (i == N)
                                i = 0;
                }

                if (a != -1 && b != -1)
                        FullDistrExchange(a,b,x,y);
        }
}

int main()
{
        toReadData();

        vector<int> InitialX, AlgorithmicX,
                                AlgorithmicBestX, OptimalX;

        vector<vector<int>> InitialY, AlgorithmicY,
                AlgorithmicBestY, OptimalY;

        double InitialTC = 0, AlgorithmicTC = 0,
                   AlgorithmicBestTC = 0, OptimalTC = 0;

        for (int i = 0; i < N; i++)
                InitialX.push_back(0);

        for (int i = 0; i < N; i++)
        {
                vector<int> tempX;

                tempX.clear();

                for (int j = 0; j < M; j++)
                        tempX.push_back(0);

                InitialY.push_back(tempX);
        }

        BuildInitialSolution(&InitialX, &InitialY);
        InitialTC = TotalCosts(InitialX, InitialY);

        AlgorithmicBestX = InitialX;
        AlgorithmicBestY = InitialY;
        AlgorithmicBestTC = InitialTC;

        vector<int> Neighbours;
        RestoreNeigboursArray(&Neighbours);
```

```
        unsigned int start_time =  clock();

        int theta = 0;
        while(theta < 73)
        {
                AlgorithmicX = AlgorithmicBestX;
                AlgorithmicY = AlgorithmicBestY;
                AlgorithmicTC = AlgorithmicBestTC;

                Shaking(&AlgorithmicX, &AlgorithmicY, &Neighbours);

                LocalSearch(&AlgorithmicX, &AlgorithmicY);

                AlgorithmicTC = TotalCosts(AlgorithmicX, AlgorithmicY);

                if (AlgorithmicTC > AlgorithmicBestTC)
                {
                        AlgorithmicBestX = AlgorithmicX;
                        AlgorithmicBestY = AlgorithmicY;
                        AlgorithmicBestTC = AlgorithmicTC;

                        RestoreNeigboursArray(&Neighbours);

                        theta = 0;
                }

                if (Neighbours.size() == 0)
                        RestoreNeigboursArray(&Neighbours);

                theta ++;
        }
        unsigned int end_time = clock();
        unsigned int search_time = end_time - start_time;

        OptimalX = AlgorithmicBestX;
        OptimalY = AlgorithmicBestY;
        OptimalTC = AlgorithmicBestTC;

        cout << std::fixed << std::setprecision(1) << OptimalTC << endl <<
           endl;

        for (int i = 0; i < N; i++)
                cout << OptimalX[i] << " ";
        cout << endl << endl;

        for (int i = 0; i<N; i++)
        {
                for (int j = 0; j<M; j++)
                        cout << OptimalY[i][j] << " ";
                cout << endl;
        }

        cout << endl << "Execution time: " << search_time * 0.001;

        return 0;
}
```