Министерство науки и образования Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информатики

Кафедра теоретических основ информатики

УДК 519.178

ДОПУСТИТЬ К ЗАЩИТЕ В ГАК

Зав. кафедрой, д.т.н., проф.

_____ Костюк Ю. Л.

«___»_____ 2011 г.

Доскоч Павел Сергеевич

РАЗРАБОТКА И ИССЛЕДОВАНИЕ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧ КОМИВОЯЖЕРА ДЛЯ КОНТУРОВ

Выпускная квалификационная работа бакалавра

Научный руководитель, зав. каф. ТОИ,
д.т.н., профессор — Ю. Л. Костюк

Исполнитель,
студ. гр. 1471 — П. С. Доскоч

Электронная версия выпускной квалификационной работы помещена в электронную библиотеку. Файл

Администратор

Томск – 2011

Реферат

Выпускная квалификационная работа 38 с., 2 рисунок, 3 таблицы, 1 источник.

ЗАДАЧА КОММИВОЯЖЕРА, КОНТУРЫ, АЛГОРИТМЫ, МЕТОД ВЕТВЕЙ И ГРАНИЦ, C#, АЛГОРИТМ ОСТОВНОГО ДЕРЕВА, ЭЙЛЕРОВ ЦИКЛ.

Объект исследования – алгоритмы решения задачи коммивояжера для контуров.

Цель работы – разработать и исследовать алгоритмы для решения задачи коммивояжера для контуров.

Метод исследования – теоретический и практический на ЭВМ.

Результаты работы – разработаны алгоритмы для решения задачи коммивояжера на контурах, разработана демонстрационная программа, реализующая данные алгоритмы, проведено тестирование алгоритмов на практике.

Содержание

Введение

Задача коммивояжера - одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. Существует большое количество научных работ и исследований по этой теме [1, разд. 10.5].

Если представить города в виде контуров, возникает новая модификация задачи коммивояжера. Такая вариация задачи еще не была исследована, как следствие, не существует алгоритма её решения.

Данная задача позволяет решить множество прикладных задач. Например, задача нахождения оптимального пути для лазера, вырезающего детали из плоской заготовки. Кольцевые линии метро можно представить в виде контуров, тогда решением задачи будет являться общая кольцевая магистраль.

Целью данной работы является изучение существующих алгоритмов решения стандартной задачи коммивояжера и их модификация с целью применения к задаче на контурах.

К задаче коммивояжера для контуров можно добавить дополнительное требование – неразрывность контуров. Мы будем рассматривать оба варианта задачи:

- Задача, не допускающая разрывы контуров
- Задача, допускающая разрывы контуров

В первой главе дано формальное описание задачи для задачи коммивояжера для контуров.

Во второй и третьей главах описаны соответствующие подзадачи и разработанные для них алгоритмы решения.

В четвертой главе описываются особенности реализации алгоритмов и результаты их работы.

1. Определения и постановка задачи

Определим некоторые понятия, необходимые для понимания задачи:

Определение 1: Вершиной будем называть точку на плоскости, заданную в декартовых координатах.

Определение 2: Отрезок – это множество точек, находящихся на прямой, образованной двумя вершинами между этими точками.

Определение 3: Ломаная – множество отрезков последовательно соединенных своими концами.

Длину ломаной будем считать как сумму длин его отрезков. Длина отрезков определяется метрикой, заданной в задаче. В данной работе используется обычное эвклидово расстояние.

Определение 4: Контур - определим как ломаную, у которой конец совпадает с началом. В случае, когда требуется использовать кривые линии, например, дуги круга, мы будем их разбивать на части, которые можно аппроксимировать ломаными линиями.

Удобным способом представления контура является перечисление вершин ломаной в порядке её обхода.

Длина контура – длинна соответствующей ломаной.

Вырожденные контуры:

1. Одну вершину можно считать вырожденным контуром.
2. Отрезок – еще один вырожденный контур – два отрезка ломаной наложены друг на друга.
3. Если пользоваться представлением контура в виде последовательности вершин, то в контуре могут появляться такие эффекты, как наложение и самопересечение.

Постановка задачи: Пусть задано множество контуров и начальная вершина. Решением задачи коммивояжера считается контур, включающий в себя все отрезки из которых состоят заданные контуры, проходящий через начальную точку и имеющий минимальную длину. Этот контур будем называть результирующим.

Существует усиленное требование к результирующему контуру: для каждого исходного контура, в результирующей последовательности вершин между вершинами из этого контура не должно быть других вершин.

Это условие назовем условием неразрывности контура – пока один контур не обошли полностью, не приступаем к обходу следующего.

Если каждый из исходных контуров будут состоять только из одной вершины, то мы получим стандартную задачу коммивояжера – соединить конечное множество точек одной ломаной линией минимальной длины[1, разд. 10.5]. То есть, результирующий контур должен будет включать в себя все заданные вершины.

Таким образом, поставленная задача является обобщением стандартной задачей коммивояжера.

2. Решение задачи с неразрывными контурами.

2.1. Особенности задачи

Для задачи с неразрывными контурами необходимо учитывать следующие факты:

Во-первых, сумма длин всех исходных контуров меньше длины контура ответа и все эти контуры входят в решение. Следовательно, минимизировать требуется лишь расстояния между вершинами, соединяющими контуры.

Во-вторых, если убрать из результирующего контура все исходные контуры, мы получим еще один контур. Так получается из-за того, что исходные контуры входят в результирующий контур как петли, убрав которые контур не разрывается.

Таким образом задачу с поиска результирующего контура можно свести к задачи поиска контура, проходящего ровно через одну вершину каждого контура и через начальную вершину и имеющего минимальную длину.

2.2. Простейший метод решения

Рассмотрим простейший метод решения полученной ранее задачи: перебрать все перестановки контуров и для каждого контура определить опорную вершину.

Данное решение имеет очень большую асимптотику - O(n!*k^n), где n - количество контуров, k - максимальное количество вершин в контуре. Но оно полезно тем, что показывает нам необходимость разбить задачу на 2 подзадачи:

- Определение порядка обхода контуров
- Выбор вершины в контуре, который будет входить в результирующий контур.

2.3. Метод ветвей и границ и алгоритм Дейкстры

Наиболее популярным алгоритмом для решения задачи коммивояжера является метод ветвей и границ. Для решения задачи на контурах модифицируем данный алгоритм, используя его совместно с алгоритмом Дейкстры. Метод ветвей и границ будет давать нам последовательность контуров, а алгоритм Дейкстры вершины из этого контура.

Для начала покажем уместность алгоритма Дейкстры. Предположим, что мы имеем некоторую последовательность контуров. Наша задача узнать, какие вершины необходимо выбрать, причем контур, проходящий через них, должен быть минимальным.

Для этого построим граф: вершинами в нем будут все вершины из контуров, а также вершины начала и конца, которые будут иметь одну и ту же координату. Ребра в графе будут проходить между вершинами, если они принадлежат контурам, стоящим рядом в последовательности контуров, а также между начальной вершиной и вершинами из первого контура в последовательности и конечной вершиной и последним контуром в последовательности. Веса ребер будут определяться как эвклидово расстояние между вершинами.

Если мы найдем кратчайший путь от начальной вершины до конечной, то заметим, что он проходит через ровно одну вершину, принадлежащую исходному контуру, что и требуется от этой подзадачи. Искать этот кратчайший путь можно классическим алгоритмом Дейкстры.

Обозначим:

- V – множество вершин;
- E – множество ребер;
- W_{ij} – вес ребра между вершинами i и j;

- s – начальная вершина;
- e – конечная вершина;
- U – множество уже посещенных вершин;
- D_i – кратчайшее расстояние от вершины i до начальной вершины через вершины из множества U;
- P_i – кратчайший путь из s в i проходящий через множество U.

Положим $D_s \leftarrow 0$, $P_s \leftarrow s$

Для всех $u \in V$ и $u \neq s$ положим $D_u \leftarrow \infty$

Пока $e \notin U$

Пусть $v \notin U$ – вершина с минимальным D_v

Если e = v – конец алгоритма

Добавим v в U

Для всех $u \notin U$ таких, что $(v,u) \in E$

Если $D_u > D_v + W_{vu}$

Положим $D_u \leftarrow D_v + W_{vu}$

Положим $P_u \leftarrow P_v,u$.

Конец если

Конец цикла

Конец цикла

В результате ответ можно получить из P_e и D_e. Асимптотика алгоритма может отличаться от способа выбора минимума и формата графа, но не превосходит $O(n^2)$[1, раздел 8.2].

Однако составленный нами граф специфичен, и алгоритм Дейкстры можно изменить так, что он будет понятнее и будет позволять проводить частичные вычисления. В графе не будет циклов. Когда мы просмотрим все вершины, из которых можно попасть в данную, для этой вершины будет известен кратчайший путь. Из формата графа можно увидеть, что это все вершины из предыдущего контура. Это значит, что мы можем поэтапно посчитать для каждого контура D. Первым и последним контуром сделаем вырожденный контур из начальной вершины.

Обозначим:

- V_{kj} – вершина jв контуре k
- S_k – количество вершин в контуре k
- N – количество контуров
- O_i – номер контура на i-й позиции в последовательности контуров.
- O_1, O_N – номера контуров полученных из начальной вершины
- p – предыдущий контур.
- c – текущий контур
- D_{kj} – кратчайшее расстояние до вершины j из контура k
- P_{kj} – кратчайший путь из начальной вершины в j-ю вершину контура k
- Dist - функция, возвращающая евклидово расстояние между вершинами.

Положим $p \leftarrow O_1 D_{p1} \leftarrow 0$;

Цикл по n от 2 до N

Цикл по i от 1 до S_c

Положим c←O_n, D_{ci}←∞

Конец цикла

Конец цикла

Цикл по n от 2 до N

Положим c←O_n

Цикл по i от 1 до S_c

Цикл по j от 1 до S_p

Если D_{ci}> D_{pj} + dist(V_{ci},V_{pj})

Положим D_{ci} ← D_{pj} + dist(V_{ci},V_{pj}), P_{ci}← P_{pj},i

Конец если

Конец цикла

Конец цикла

Положим p←c

Конец цикла

Теорема: трудоемкость данного алгоритма O(N*S^2), где N – количество контуров, S – максимальное количество вершин в контуре.

Доказательство: Если развернуть внешний цикл основной части алгоритма, и рассмотреть внутренние, то получим $T = \sum_{i=2}^{N} S_{o_i} * S_{o_{i-1}} * b \leq \sum_{i=2}^{N} (\max_{1 \leq i \leq N} S_i)^2 * b \leq N * b * (\max_{1 \leq i \leq N} S_i)^2 = O(N * S^2)$. Что и требовалось доказать.

Простейший способ получить последовательность O – полный перебор. Трудоемкость такого перебора O(n!).

При переборе можно использовать отсечения. Из упрощенного алгоритма можем заметить, что между обработкой контуров мы имеем расстояние от начальной вершины до текущего контура. Таким образом, зная лишь часть последовательности можно оценить будущий ответ снизу. Данную оценку можно использовать при работе с методом ветвей и границ: при выборе очередного контура мы вычисляем расстояния для каждой из его вершин и оцениваем их. Эти расстояния мы используем для вычисления расстояний следующих вершин. В итоге, совмещая метод ветвей и границ и упрощенный алгоритм Дейкстры, получаем алгоритм, который решает поставленную задачу.

Обозначим:

- V_{kj} – вершина j в контуре k
- S_k – количество вершин в контуре k
- N – количество контуров
- O_i – номер контура на i-й позиции в последовательности контуров
- O_1, O_N – номера контуров полученных из начальной вершины
- U_i – 1 если i-й контур уже в последовательности, иначе 0
- p – предыдущий контур
- c – текущий контур
- D_{kj} – кратчайшее расстояние до вершины j из контура k
- P_{kj} – кратчайший путь из начальной вершины в j-ю вершину контура k

- Dist – функция, возвращающая евклидово расстояние между вершинами.
- Best – текущее лучшее решение
- BestLen – длина лучшего решения

Функция Проверить(n)

Положим len← ∞,c ← O_n, p ← O_{n-1}

Цикл по i от 1 до S_c

Положим D_{ci} ←∞

Цикл по jот 1 до S_p

Если D_{ci}> D_{pj} + dist(V_{ci},V_{pj})

Положим D_{ci} ← D_{pj} + dist(V_{ci},V_{pj}), P_{ci}← P_{pj},i

Конец если

Если D_{ci}<len

Положим len ← D_{ci}

Конец если

Конец цикла

Конец цикла

Вернуть len, как результат функции

Конец функции

Функция Решить(n)

Если (n = N)

Положим len← Проверить(n)

Если BestLen>len

Положим Best← { O, P_{N1}}, BestLen ← len

Конец если

Иначе

Цикл по i от 2до N - 1

Положим O_n←i

Если U_i = 0 и Проверить(n) >BestLen

Положим U_c← 1

Решить(n + 1)

Положим U_c← 0

Конец если

Конец цикла

Конец если

Конец функции

Выполнив Решить(2), мы получим ответ в Best и BestLen, а также из P.

Теорема: Трудоемкость данного алгоритма $O(N * S^2)$ – где N – количество контуров, а S – максимальное количество вершин в контуре.

Время работы функции «Проверить» $T_c = S^2 * b$, так как мы имеем два вложенных цикла фиксированной длинны. b–константное количество действий внутри циклов.

Время работы функции «Решить» можно описать следующим рекуррентным соотношением:

$$T_s(n) = \begin{cases} T_c, \text{при } n = 1 \\ n * \left(T_c + T_s(n-1)\right), \text{при } n \geq 2 \end{cases}$$

Развернув данное рекуррентное соотношение, мы получим:

$$T_s(N) = T_c * \left(N + N * (N-1) + \cdots + \frac{N!}{2} + \frac{N!}{1}\right) = T_c * N! * \left(\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots \frac{1}{n!}\right)$$
$$\leq S^2 * b * N! * (e - 1) = O(N! * S^2)$$

Теорема доказана.

Данная теорема дает асимптотическую трудоемкость, но в ней не учитываются отсечения и начальное приближение. Однако, даже без этих оценок она асимптотически эффективнее, чем алгоритмы, упомянутые выше.

2.4. Начальное приближение для метода ветвей и границ и приближенные методы нахождения ответа

Вопрос наличия эффективных приближенных методов для решения задачи коммивояжера для контуров остается не до конца определенным. Проблема заключается в том, что не понятно, как построить граф, с которым работают приближенные алгоритмы решения задачи коммивояжера.

Все алгоритмы, так или иначе, пользуются расстоянием между вершинами, под которыми в нашей задаче подразумеваются контуры. Но мы не можем строго определить расстояние между контурами, поскольку при выборе разной последовательности контуров у нас будет разное расстояние.

Задача коммивояжера обычно определяется на плоскости, и там действует правило треугольника. Этим пользуются некоторые приближенные методы, например метод остовного дерева [1, раздел 10.6]. Наша задача также определена на плоскости, но граф, в котором вершины – контуры уже не соответствует этому условию, даже если использовать простейшие эвристики расстояния между контурами.

Это не значит, что приближенные методы неприменимы. Это значит, что мы не сможем так же качественно оценить ответ от этих приближенных методов.

2.6. Решение исходной задачи

Имея последовательность вершин, полученных в ответе, необходимо восстановить исходное решение.

Для этого достаточно для каждой вершины из последовательности найти контур, которому она принадлежит, и добавить следом за этой вершиной все вершины этого контура в порядке его обхода. При этом последней должна быть добавлена эта же самая вершина. Полученная в итоге последовательность вершин образует результирующий контур.

3. Решение общей задачи коммивояжера для контуров

3.1. Особенности задачи

В разделе 2.1. были сделаны выводы, что следует минимизировать лишь то, что соединяет исходные контуры в результирующем контуре. Эти выводы справедливы и для общей задачи. Однако, следует определиться, чем является то, что мы будем минимизировать. Для этого необходимо узнать, что останется, если убрать из результирующего контура все отрезки, принадлежащие исходным контурам.

Представим результирующий контур в виде графа. В этом графе все вершины имеют четную степень. Вершины в исходных контурах тоже имеют четные степени. Если уберем из результирующего контура ребра, которые есть в исходных контурах, то степени также останутся четными. Теперь уберем из полученного графа вершины, степень которых 0 – это вершины, не участвующие в соединении контуров. В итоге мы по-прежнему имеем граф, в котором степени вершин четные. Разобьем его на компоненты связанности. В каждой компоненте связанности можно найти Эйлеров цикл, а это значит, что каждая компонента связанности – контур. Таким образом, получаем, что исходные контуры соединены такими же контурами.

Воспользовавшись предыдущим выводом можно переформулировать условие задачи: необходимо в мульти-граф, где вершины – вершины контура, а ребра – линии между вершинами в контуре, добавить ребра, так чтобы в графе существовал Эйлеров цикл, и сумма этих ребер была минимальна.

Для данной задачи используется мульти-граф. Это обосновано тем, что существуют вырожденные контуры, состоящие из двух вершин. Для представления их в графе потребуется два ребра между вершинами.

В результате получаем, для того чтобы решить общую задачу коммивояжера для контуров, необходимо найти множество контуров минимальной длинны, такое, что вершина этого контура либо является вершиной исходного конура, либо начальной вершиной. При этом не должно быть такого, что нет пути из одного контура в другой либо в начальную вершину.

3.2. Полный перебор с отсечениями

Полный перебор в данном случае будет представлять собой: во-первых перебор групп контуров, между которыми будут проходить контуры, во-вторых перебор последовательности контуров в этих группах. Поскольку группы обязаны пересекаться, и граф, построенный из контуров и связей между ними, должен быть связан, мы получаем очень трудоемкий перебор.

3.3. Приближенное решение

Идея приближенного решения, которое я хочу изложить, очень простая и основывается на идее остовного дерева для задачи коммивояжера.

Построим граф, в котором вершинами будут контуры, а ребра – кратчайшие расстояние между ними, запомнив при этом вершины, через которые проходит это кратчайшее расстояние. Найдем в этом графе остовное дерево. В граф, состоящий из вершин и линий, добавим ребра, соответствующие ребрам из полученного остовного дерева, при этом добавим дважды. Получится полностью связанный граф. Эти удвоенные ребра и будут образовывать соединительные контуры, вырожденные в удвоенную линию.

Для нахождения остовного дерева будем использовать алгоритм Прима. Его трудоемкость $O(n^2)$, где n – количество вершин[1, с. 162]. В нашем случае эта трудоемкость к $O(N^2S^2)$, поскольку остовное дерево будет искаться для всех вершин всех контуров.

Теорема: Приближенное решение, полученное методом остовного дерева, не превосходит точное решение более чем в два раза.

Доказательство:

L_b – длина точного решения, L_t – длина остовного дерева. L_a – длина приближенного ответа.

Точное решение соединяет все контуры в одну компоненту связанности. Ребра из остовного дерева тоже соединяют контуры в одну компоненту связанности. Но остовное дерево по определению имеет кратчайшую длину. Это значит, что $L_b \geq L_t$. Поскольку в ответ каждое ребро из остовного дерева входит дважды, то мы имеем длину ответа $L_a = 2 * L_t$. Это решение является приближенным, а значит его длина больше приближенного решения. В итоге получаем $L_t \leq L_b \leq L_a = 2 * L_t \leq 2 * L_b$, из чего следует, что $L_a \leq 2 * L_b$. Теорема доказана.

В оригинальной варианте применения метода остовного дерева, сокращаются все дублирующие ребра, но в нашем случае это не подходит. Во-первых, мы имеем дело с Эйлеровым циклом, а не с оригинальной задачей коммивояжера. Во-вторых, в графе, в котором мы ищем остовное дерево, не выполняется правило треугольника.

3.4. Решение исходной задачи

Для того чтобы получить решение исходной задачи, необходимо добавить ребра остовного дерева в исходный граф, удвоив их, и найти в этом графе Эйлеров цикл[1, глава 9].

4. Реализация алгоритмов

Выбор языка программирования для задач такого рода большой значимости не имеет. Язык C# был выбран лишь потому, что на нем с помощью .Net очень просто нарисовать демонстрационную форму.

4.1. Руководство пользователя

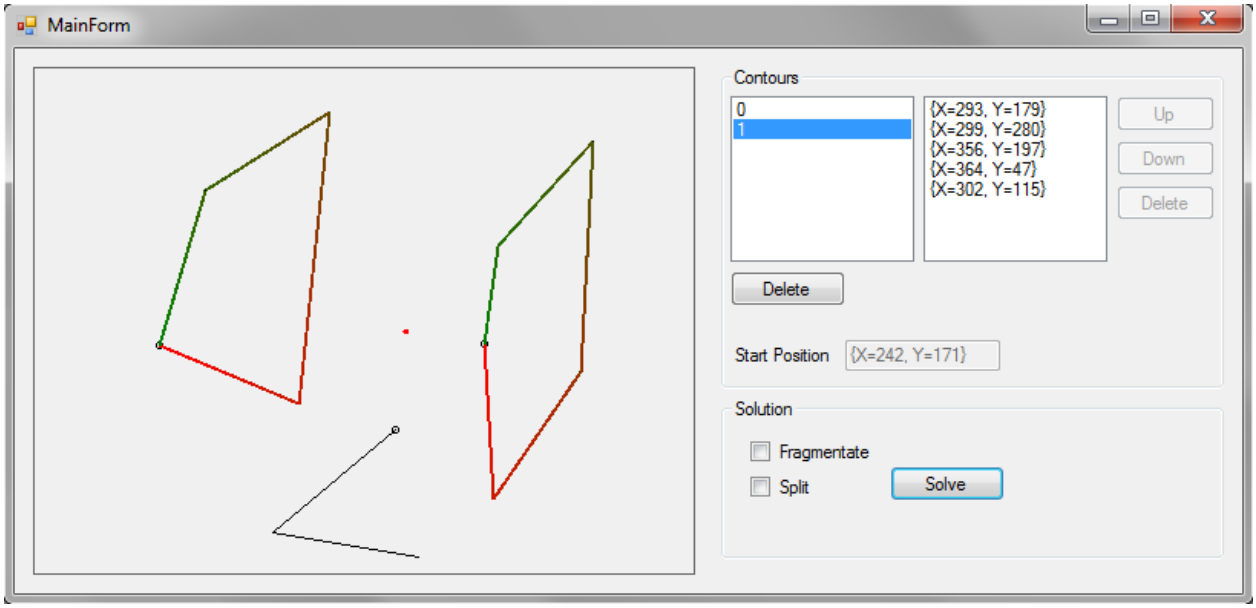

Рисунок 4.1 Демонстрационная форма алгоритмов

На Рисунке 4.1 представлена демонстрационная форма, как задавать контуры, редактировать их и просматривать решения различных алгоритмов для заданных контуров.

Для того чтобы начать задавать контур следует кликнуть левой кнопкой мыши в точку, где будет первая вершина. Затем продолжать кликать в порядке обхода контура. Когда все вершины контура установлены, еще раз кликнуть на первую вершину контура.

После того, как заканчивается ввод контура, в первом списке появляется очередной номер, символизирующий этот контур. Если этот контур выбрать в списке, он подсветится и во втором списке можно увидеть координаты вершин этого контура в порядке обхода.

Выбранный контур можно удалить, нажав на кнопку *Delete* под списком контуров.

Редактировать контур можно с помощью кнопок управления: изменять порядок вершин в этом контуре с помощью кнопок *Up* и *Down*; удалять вершины кнопкой *Delete*, продолжить формирование контура в поле рисования.

Чтобы установить начальную точку, следует кликнуть правой кнопкой мыши по полю рисования в том месте, где необходимо её установить. В поле *StartPosition* можно увидеть координаты этой точки. Изначально координаты начальной точки выбраны (0,0).

Чтобы увидеть решение для заданных контуров, следует нажать кнопку *Solve*.

Тип решения определяется флагами *Fragmentate* и *Split*. Флаг *Split* определяет, какой тип задачи мы решаем: если он установлен, то в задаче допускаются разрывы, если не установлен – не допускаются. Флаг *Fragmentate* отвечает за фрагментацию контуров. Если он установлен, то в исходные контуры будут добавлены дополнительные вершины, так,

чтобы контур не изменился, а между соседними вершинами было расстояние не более одного пикселя. Кнопка *Solve* не будет работать, если задано менее трех контуров.

4.2. Руководство программиста

Проект разрабатывался в среде Microsoft Visual Studio 10.

В состав проекта входит 4 библиотеки:

1. Библиотека, описывающая модель данных
2. Библиотека алгоритмов
3. Библиотека тестов
4. Библиотека демонстрационной формы

Диаграмма классов 1-ой и 2-ой библиотеке расположена на рис 4.1.

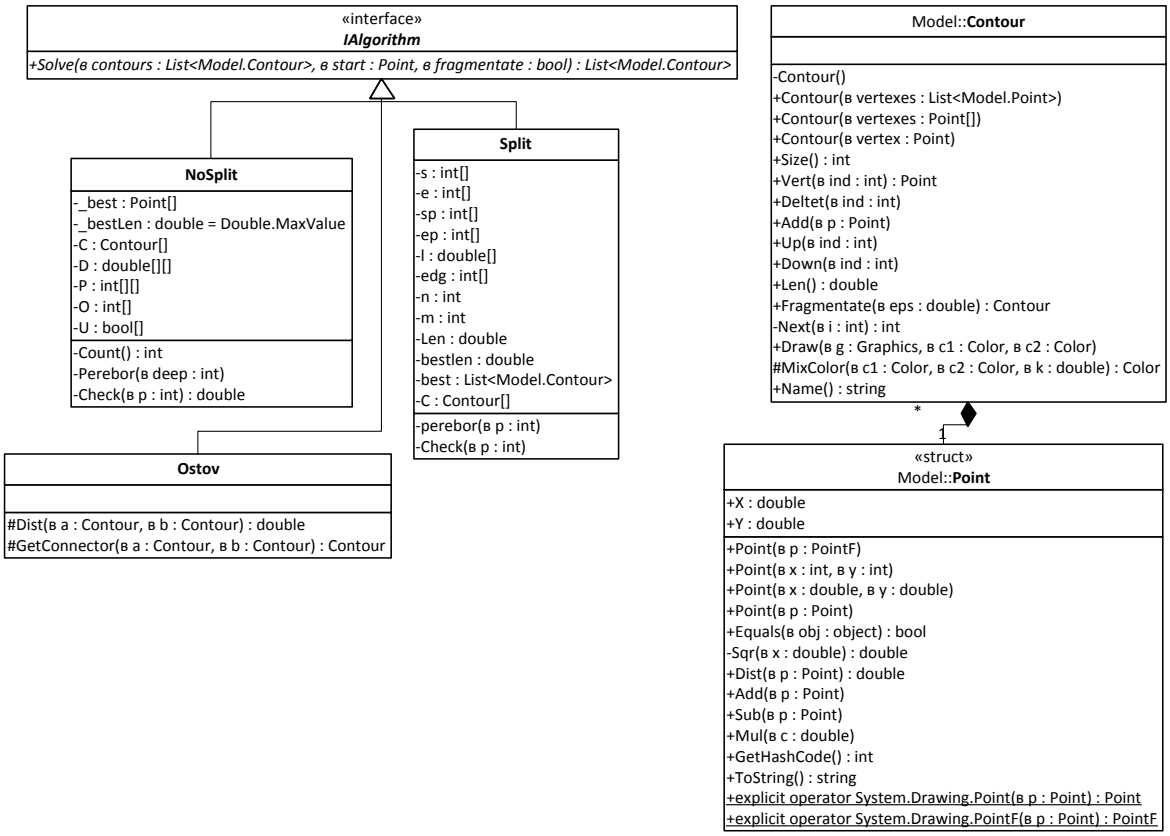

Рис. 4.1

4.2.1. Библиотека данных

Структура *Point* – структура, представляющая вершину на плоскости.

Публичные Поля:

double X – абсцисса вершины.

double Y – ордината вершины.

Конструкторы:

Имеет 4 вида конструкторов: из целых координат, из вещественных координат, конструктор копирования и конструктор от типа *System.Drawing.PointF.*

Публичные методы:

bool Equals(object obj) – возвращает истину, если параметр совпадает с точкой.

double Dist(Point P) – возвращает евклидово расстояние между точкой и параметром.

void Add(Point p) – добавляет к точке координаты точки из параметра.

void Sub(Point p) – отнимает от точки координаты точки из параметра.

void Mul(double c) – умножает точку на коэффициент *c.*

int GetHashCode() – возвращает хеш-функцию от точки.

string ToString() – возвращает строковое представление точки.

Операторы:

Имеет 2 оператора приведения типа: к типу *System.Drawing.PointF* и к типу *System.Drawing.Point.*

Класс *Contour* – класс, представляющий контур.

Приватные поля:

List<Point> _vert – список вершин контура в порядке его обхода.

Публичные поля:

int Size – количество вершин в контуре

string Name – именование контура.

Конструкторы:

Имеет 4 конструктора: без параметров, от *List<Point>*, от *Point[]* и от *Point.* Последние 3 добавляют вершины в поле *_vert.*

Публичные методы:

void Point Vert(int ind) – возвращает вершину по индексу

void Delete(int ind) – удаляет вершину по индексу

void Add(Point P) – добавляет вершину в конец списка

void Up(int ind) – смещает вершину на одну позицию вверх в *_vert*

void Down(int ind) – смещает вершину на одну позицию вниз в *_vert*

double Len() – возвращает длинну контура.

Contour Fragmentate(double eps) –возвращает фрагментированный контур, который полностью совпадает с текущим, но расстояния между вершинами которого меньше *eps*.

Void Draw(Graphis g, Color c1, Color c2) – рисует контур на *g* как градиентную линию цветов *c1* и *c2*

4.2.2. Библиотека алгоритмов

Интерфейс *IAlgorithm* – интерфейс для классов, реализующих алгоритм решения задачи коммивояжера для контуров.

Методы:

List<Countour> Solve(List<Countour> countours, Point start, bool fragmentate) – возвращает решение задачи коммивояжера для контуров заданной в параметрах.

Класс *NoSplit* – реализация алгоритма коммивояжера для неразрывных контуров. Реализует интерфейс *IAlgorithm*

Приватные поля:

Point[] _best – текущая лучшая последовательность вершин

double _bestLen – длина текущего лучшего решения.

Contour[] C – массив исходных контуров.

double[][] d – массив расстояний до вершин

int[][] p –массив предыдущих вершин

int[] O – текущая последовательность контуров.

bool[] U – флаг использования контура.

int Count – количество контуров.

Приватные методы:

void Perebor(int deep) – реализация метода ветвей и границ

double Check(int p) – пересчет расстояний для текущего контура.

Класс Ostov – реализация приближенного алгоритма решения общей задачи коммивояжера для контуров. Реализует интерфейс IAlgorithm.

Защищенные методы:

Contour GetConnector(Contour a, Contour b) – возвращает вырожденный контур из 2-х вершин, наименьшей длинны, соединяющий контуры a и b.

Double Dist(Contour a, Contour b) – возвращает расстояние между двумя контурами.

4.2.3. Библиотека тестов

Класс *Test* – реализация тестов для алгоритмов из второй библиотеки.

Публичные методы:

void TestOstovTime() – проводит тестирование времени работы приближенного алгоритма решения общей задачи и сохраняет результаты в файл.

void TestOstovLen() – проводит тестирование коэффициента удлинения приближенного решения общей задачи относительно точного решения и сохраняет результат в файл.

void TestNoSplitTime() – проводит тестирование времени работы алгоритма решения задачи для неразрывных контуров и сохраняет результаты в файл.

4.2.4. Библиотека демонстрационной формы

Класс *DemoForm* – класс-форма, предоставляющий интерфейс для демонстрации работы алгоритмов решения задачи коммивояжера для контуров.

4.3. Результаты тестов

Все тесты проводились на произвольных контурах. Контуры создавались как последовательность произвольных точек в квадрате 20 на 20.

В таблице 4.1 содержатся длины ответов для точного и приближенного решения общей задачи. Максимальное количество вершин (S) – 10.

Таблица 4.1. Сравнение длин ответов приближенного и точного алгоритмов для общей задачи.

N	3	4	3	4	3	5
Точное решение	5,454888	7,420671	6,536405	8,195335	9,352841	9,705982
Метод остовного дерева	6,977977	8,94376	9,335096	10,99403	10,38874	13,65355
% от точного	127,92%	120,52%	142,82%	134,15%	111,08%	140,67%

В таблице 4.2 представлено время работы алгоритма приближенного решения задачи для контуров. Время выражено в секундах. Тесты проводились на контурах с количеством вершин 10 и 200.

Таблица 4.2. Время работы приближенного алгоритма в секундах.

N	4	5	4	5	8	10	12	15
S < 10	0,000123	0,000126	0,000123	0,000126	0,000236	0,000349	0,000521	0,000858
s < 200	0,007124	0,007386	0,007124	0,000126	0,018066	0,025916	0,034248	0,056462
N	18	20	30	50	100	200	500	1000
s < 10	0,001043	0,001518	0,002554	0,008822	0,026887	0,114482	0,720842	2,772478
s <200	0,078254	0,088695	0,173918	0,532516	2,081616	8,528433	58,21035	225,0969

В таблице 4.3 содержится время работы алгоритма задачи для неразрывных контуров. Количество вершин в контурах не превышало 10.

Таблица 4.3. Время работы алгоритма для неразрывных контуров в секундах.

N	3	4	5	6	8	10	12
Время	0,006521	0,000488	0,001713	0,010214	0,384601	15,4385	784,0465

Заключение

Результатом данной работы являются:

1. Два алгоритма решения задачи коммивояжера для контуров. Один для задачи с неразрывными контурами, другой для приближенного решения общей задачи.
2. Теоретическое исследование этих алгоритмов: доказательство трудоемкости, доказательство коэффициента отклонения от точного решения для приближенного решения.
3. Реализация библиотеки классов для решения этих задач на языке C# и демонстрационной формы. Исходный код представлен в Приложении А.
4. Проведение тестов полученных алгоритмов. Таблица с результатами представлена в Приложении Б.
5. Выводы, сделанные на основе полученных результатов.

Выводы:

В результате исследования было выявлено:

1. Переборные алгоритмы проще и эффективнее для задачи с неразрывными контурами, чем для задачи с контурами, допускающими разрывы.
2. Приближенные алгоритмы для задачи с неразрывными контурами имеют существенные препятствия. Задача для контуров с разрывами не встречает подобных препятствий.
3. Алгоритмы нахождения точного решения при существующих мощностях компьютера непригодны для большого количества контуров.

Список литературы

1. *Кристофидес Н.* Теория графов. Алгоритмический подход. - М.: Мир, 1978.- 432с.

Приложение А. Исходный код алгоритмов

```
//Файл Point.cs

   public struct Point
    {
        public Point(System.Drawing.PointF p)
        {
            X = p.X;
            Y = p.Y;
        }

        public Point(int x, int y)
        {
            X = x;
            Y = y;
        }

        public Point(double x, double y)
        {
            X = x;
            Y = y;
        }

        public Point(Point p)
        {
            X = p.X;
            Y = p.Y;
        }

        public override bool Equals(object obj)
        {
            if (!(obj is Point))
            {
                return false;
            }

            Point p = (Point)obj;

            return X == p.X && Y == p.Y;
        }

        public double X;// { get { return _x; } set { _x =
value; } }
        public double Y;// { get { return _y; } set { _y =
value; } }

        private double Sqr(double x)
        {
            return x * x;
        }

        public double Dist(Point p)
```

```
        {
            return Math.Sqrt(Sqr(p.X - X) + Sqr(p.Y - Y));
        }

        public void Add(Point p)
        {
            X += p.X;
            Y += p.Y;
        }

        public void Sub(Point p)
        {
            X -= p.X;
            Y -= p.Y;
        }

        public void Mul(double c)
        {
            X *= c;
            Y *= c;
        }

        public override int GetHashCode()
        {
            return (int)(X * 10000 + Y * 1000000);
        }

        public override string ToString()
        {
            return "{X=" + X.ToString() + ", Y=" + Y.ToString()
+ "}";
        }

        public static explicit operator
System.Drawing.Point(Point p)
        {
            return new System.Drawing.Point((int)p.X, (int)p.Y);
        }

        public static explicit operator
System.Drawing.PointF(Point p)
        {
            return new System.Drawing.PointF((float)p.X,
(float)p.Y);
        }
    }

//Файл Contour.cs

     public class Contour
     {
          private List<Point> _vert;
```

```
  private Contour()
  {
    _vert = new List<Point>();
  }

  public Contour(List<Point> vertexes)
       : this()
  {
       _vert.AddRange(vertexes);
  }

  public Contour(Point[] vertexes)
       : this()
  {
       _vert.AddRange(vertexes);
  }

  public Contour(Point vertex)
       : this()
  {
       _vert.Add(vertex);
  }

  public int Size
  {
       get
       {
            if (_vert == null)
            {
                 return 0;
            }

            return _vert.Count;
       }
  }

public Point Vert(int ind)
{
    return _vert[ind];
}

public void Deltet(int ind)
{
    _vert.RemoveAt(ind);
}

public void Add(Point p)
{
    _vert.Add(p);
}

public void Up(int ind)
{
```

```
            if (ind <= 0)
                return;
            Point p = _vert[ind];
            _vert[ind] = _vert[ind - 1];
            _vert[ind - 1] = p;
        }

        public void Down(int ind)
        {
            Up(ind + 1);
        }

          public double Len()
          {
               if (_vert == null)
               {
                    return 0;
               }

               double l = 0;

               for (int i = 0; i < Size; i++)
               {
                    l += _vert[i].Dist(_vert[Next(i)]);
               }

               return l;
          }


          public Contour Fragmentate(double eps)
          {
               if (_vert == null)
               {
                    return new Contour();
               }

               List<Point> result = new List<Point>();
               for (int i = 0; i < Size; i++)
               {
                    int k =
(int)Math.Ceiling(_vert[i].Dist(_vert[Next(i)]) / eps);
                    Point d = new Point(_vert[Next(i)]);
                    d.Sub(_vert[i]);
                    d.Mul(1 / k);

                    Point p = _vert[i];

                    for (int st = 0; st < k; st++)
                    {
                         result.Add(p);
                         p = new Point(p);
                         p.Add(d);
```

```
                    }
                }
                return new Contour() { _vert = result };
            }

            private int Next(int i)
            {
                return (i + 1) % Size;
            }

        public void Draw(Graphics g, Color c1, Color c2)
        {
            double l = 0;
            for (int i = 0; i < _vert.Count; i++)
                l += _vert[i].Dist(_vert[Next(i)]);
            double other = 0;
            for (int i = 0; i < _vert.Count; i++)
            {
                double d = _vert[i].Dist(_vert[Next(i)]);
                Color C1 = MixColor(c1, c2, other / l);
                other += d;
                Color C2 = MixColor(c1, c2, other / l);
                g.DrawLine(new Pen(new
LinearGradientBrush((PointF)_vert[i], (PointF)_vert[Next(i)],
C1, C2), 2), (PointF)_vert[i], (PointF)_vert[Next(i)]);
            }
        }

        protected Color MixColor(Color c1, Color c2, double k)
        {
            double r = c1.R * k + c2.R * (1 - k);
            double g = c1.G * k + c2.G * (1 - k);
            double b = c1.B * k + c2.B * (1 - k);
            return Color.FromArgb((int)r, (int)g, (int)b);
        }

        public String Name { get; set; }
    }

//Файл IAlgrotihm.cs

    public interface IAlgorithm
    {
        System.Collections.Generic.List<Model.Contour>
Solve(System.Collections.Generic.List<Model.Contour> contours,
Model.Point start, bool fragmentate);
    }
}

//Файл NoSplit.cs

    public class NoSplit :IAlgorithm
    {
```

```
        private Point[] _best;
        private double _bestLen = Double.MaxValue;
        private Contour[] C;
        private double[][] D;
        private int[][] P;

        private int[] O;
        private bool[] U;

        private int Count { get { return C.Length; } }

        private void Perebor(int deep)
        {
            if (deep == 1)
            {
                if (Check(Count - 1) < _bestLen)
                {
                    _bestLen = D.Last().Last();//
C.Last().D.Last();
                    _best[0] = C.Last().Vert(0);//.First());
                    int p = 0;
                    for (int i = Count - 1; i >= 1; i--)
                    {
                        p = P[O[i]][p];// C[O[i]].P[p];
                        _best[Count - i] = (C[O[i -
1]].Vert(p));
                    }
                }
                return;
            }
            for (int i = 1; i < Count - 1; i++)
                if (!U[i])
                {
                    U[i] = true;
                    O[Count - deep] = i;
                    if (Check(Count - deep) < _bestLen)
                    {
                        Perebor(deep - 1);
                    }
                    U[i] = false;
                }
        }

        private double Check(int p)
        {
            int s = O[p - 1];
            int e = O[p];
            double ans = Double.MaxValue;
            for (int i = 0; i < C[e].Size; i++)
            {
                D[e][i] = Double.MaxValue;
                for (int j = 0; j < C[s].Size; j++)
                {
```

```
                    double d = D[s][j] +
C[s].Vert(j).Dist(C[e].Vert(i));
                    if (d < D[e][i])
                    {
                        D[e][i] = d;
                        P[e][i] = j;
                    }
                    ans = Math.Min(ans, d);
                }
            }
            return ans;
        }


        public List<Contour> Solve(List<Contour> contours, Point
start, bool fragmentate)
        {
            C = new Contour[contours.Count + 2];

            _best = new Point[Count];// new List<PointF>();
            _bestLen = Double.MaxValue;

            for (int i = 0; i < contours.Count; i++)
            {
                if (fragmentate)
                    C[i + 1] = contours[i].Fragmentate(1.0);
                else
                    C[i + 1] = contours[i];
            }

            C[0] = new Contour(start);
            C[Count - 1] = new Contour(start);

            D = new double[Count][];
            P = new int[Count][];
            for (int i = 0; i < Count; i++)
            {
                D[i] = new double[C[i].Size];
                P[i] = new int[C[i].Size];
            }

            U = new bool[Count];
            O = new int[Count];
            O[0] = 0;
            O[Count - 1] = Count - 1;
            D[0][0] = 0;

            Perebor(Count - 1);

            List<Contour> result = new List<Contour>();
            result.Add(new Contour(_best));
            return result;
        }
```

```
    }

//Файл Ostov.cs

    public class Ostov : IAlgorithm
    {
        protected double Dist(Contour a, Contour b)
        {
            var conn = GetConnector(a, b);
            return conn.Vert(0).Dist(conn.Vert(1));
            //return conn.Vert(0).Dist(conn.Vert(1));
        }

        protected Contour GetConnector(Contour a, Contour b)
        {
            double min = Double.MaxValue;
            Point st = new Point();
            Point en = new Point();
            for (int i = 0; i < a.Size; i++)
                for (int j = 0; j < b.Size; j++)
                {
                    Point p = a.Vert(i);
                    Point q = b.Vert(j);
                    double d = p.Dist(q);
                    if (min > d)
                    {
                        st = p;
                        en = q;
                        min = d;
                    }

                }
            return new Contour(new Point[] { st, en });
        }

        public List<Contour> Solve(List<Contour> contours,
Model.Point start, bool fragmentate)
        {
            List<Contour> C = new List<Contour>();
            foreach (Contour c in contours)
                if (fragmentate)
                    C.Add(c.Fragmentate(1.0));
                else
                    C.Add(c);
            C.Add(new Contour(start));

            int[] p = new int[C.Count];
            double[] d = new double[C.Count];
            double[,] M = new double[C.Count, C.Count];
            for (int i = 0; i < C.Count; i++)
            {
                for (int j = i + 1; j < C.Count; j++)
                {
```

```
                M[i, j] = M[j, i] = Dist(C[i], C[j]);
            }
            d[i] = M[i, 0];
            p[i] = 0;
        }
        p[0] = 0;

        bool[] u = new bool[C.Count];
        u[0] = true;

        for (int t = 1; t < C.Count; t++)
        {
            int b = 0;
            while (u[b])
                b++;

            for (int i = b + 1; i < C.Count; i++)
                if (!u[i] && d[i] < d[b])
                    b = i;
            u[b] = true;

            for (int i = 0; i < C.Count; i++)
                if (!u[i] && d[i] > M[b, i])
                {
                    p[i] = b;
                    d[i] = M[b, i];
                }
        }
        List<Contour> result = new List<Contour>();
        for (int i = 1; i < C.Count; i++)
            result.Add(GetConnector(C[i], C[p[i]]));
        C.RemoveAt(C.Count - 1);
        return result;
    }
}
```