# СОДЕРЖАНИЕ



# Реализация приближенного параллельного алгоритма решения задачи коммивояжера

## ВВЕДЕНИЕ

Задача коммивояжера (Traveling Salesman Problem, TSP) является одной из самых известных задач комбинаторной оптимизации, имеющей множество приложений в различных областях, таких как транспорт, логистика, телекоммуникации, биоинформатика и другие. Она заключается в нахождении кратчайшего маршрута, проходящего через все заданные точки (города), при условии, что каждый город должен быть посещен только один раз, а начало и конец маршрута должны совпадать.

Задача коммивояжера является NP-полной, что означает, что для ее точного решения необходимо перебрать все возможные варианты, что при больших размерностях становится непрактичным. Поэтому существуют различные приближенные алгоритмы, которые позволяют получать приближенное решение с заданной точностью за разумное время.

Реализация приближенного параллельного алгоритма решения задачи коммивояжера имеет большое практическое значение, поскольку такой алгоритм может использоваться для решения задач в реальном времени на больших объемах данных, например, в сфере логистики и транспорта.

Целью данной работы является разработка и реализация приближенного параллельного алгоритма решения задачи коммивояжера и его анализ. Для достижения цели были поставлены следующие задачи:

- Изучить теорию задачи коммивояжера и обзор существующих алгоритмов решения этой задачи;
- Разработать приближенный параллельный алгоритм решения задачи коммивояжера и описать его особенности;
- Реализовать алгоритм на языке программирования и описать используемые технологии;
- Провести эксперименты и оценить эффективность и точность алгоритма.

.

## ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

### Описание задачи коммивояжера

Задача коммивояжера - это задача поиска минимального по стоимости замкнутого маршрута, проходящего через все заданные точки (города) ровно один раз и вернувшегося в исходный город.

Формально, задача коммивояжера можно определить следующим образом: имеется набор из n городов, заданных своими координатами на плоскости, а также матрица расстояний между городами. Требуется найти гамильтонов цикл минимальной стоимости, проходящий через все n городов ровно один раз и вернувшийся в исходный город.

Задача коммивояжера является NP-полной, то есть на данный момент неизвестен алгоритм, который мог бы решить ее за полиномиальное время на всех возможных входных данных.

Поэтому для решения задачи коммивояжера используются различные алгоритмы, которые дают лишь приближенное решение. В зависимости от размера входных данных и требуемой точности, выбирается тот или иной алгоритм.

Задача коммивояжера имеет множество приложений в различных областях, таких как логистика, транспортное планирование, телекоммуникации, генетика и другие.

Решение задачи коммивояжера является важным в задачах оптимизации и поиске оптимального решения, поэтому существует множество алгоритмов, которые позволяют приближенно решать эту задачу.

Среди основных алгоритмов решения задачи коммивояжера можно выделить:

1. Полный перебор - перебор всех возможных гамильтоновых циклов и выбор минимального по стоимости. Этот алгоритм имеет экспоненциальную сложность и применяется только на малых наборах данных.
2. Метод ближайшего соседа - начинается с произвольного города и на каждом шаге выбирается ближайший свободный город. Этот алгоритм прост в реализации, но дает неоптимальные решения.

3. Метод минимального остовного дерева - строится минимальное остовное дерево на графе и обходится в порядке обхода в глубину. Этот алгоритм имеет сложность $O(n^2)$, но не гарантирует нахождения оптимального решения.

4. Метод имитации отжига - алгоритм, основанный на принципе эволюционного развития. Он позволяет находить приближенное решение за короткий промежуток времени, но также не гарантирует оптимальности решения.

5. Генетические алгоритмы - алгоритмы, основанные на эволюционных принципах. Они позволяют находить приближенное решение и справляются с большими объемами данных.

6. Динамическое программирование - алгоритм, который решает задачу коммивояжера в несколько этапов, используя определенные математические формулы. Этот алгоритм позволяет решить задачу за полиномиальное время, но работает только на небольших наборах данных.

Каждый из этих алгоритмов имеет свои преимущества и недостатки, и выбор алгоритма зависит от размера входных данных и требуемой точности.

Кроме того, существуют и другие алгоритмы решения задачи коммивояжера, такие как алгоритм Хелда-Карпа, алгоритм Лин-Кернигана и многие другие. Они также имеют свои преимущества и недостатки и используются в зависимости от поставленной задачи и требуемой точности решения.

В данной курсовой работе мы рассмотрим несколько известных алгоритмов решения задачи коммивояжера и проведем сравнительный анализ их эффективности на различных входных данных. Мы будем использовать язык программирования C# и среду разработки Visual Studio 2019 для реализации алгоритмов и проведения экспериментов.

В дальнейшем мы будем использовать термин "решение задачи коммивояжера" для обозначения поиска минимального гамильтонова цикла в полном взвешенном графе, проходящего через каждую вершину ровно один раз.

**Обзор и сравнительный анализ алгоритмов решения задачи коммивояжера**

Обзор и сравнительный анализ алгоритмов решения задачи коммивояжера является важной задачей, так как эта задача является NP-полной, то есть нахождение оптимального решения требует вычислительной сложности, которая растет экспоненциально с размером входных данных. Поэтому для решения задачи используются различные приближенные алгоритмы, которые могут дать оптимальный результат или результат, близкий к оптимальному, за приемлемое время.

Ниже приведен обзор нескольких алгоритмов решения задачи коммивояжера:

1. Алгоритм полного перебора. Это самый простой алгоритм, который заключается в переборе всех возможных путей и выборе наименьшего. Однако вычислительная сложность этого алгоритма увеличивается, что делает его практически неприменимым для решения задачи на больших входных данных.

2. Эвристический алгоритм. Этот алгоритм заключается в выборе каждый раз ближайшей доступной вершины, пока не будет посещена каждая вершина. Он дает хорошие результаты для небольших наборов данных, но может давать далеко не оптимальный результат на больших наборах данных.

3. Алгоритм ветвей и границ. Это более сложный алгоритм, который использует метод ветвей и границ для выбора оптимального пути. Он дает более точные результаты, чем жадный алгоритм, но имеет высокую вычислительную сложность.

4. Алгоритм имитации отжига. Этот алгоритм основан на эвристике, которая использует случайный поиск с постепенным уменьшением температуры. Он может давать результаты, близкие к оптимальным, но требует тщательной настройки параметров.

5. Генетический алгоритм. Этот алгоритм использует эволюционный подход для решения задачи коммивояжера. Он создает популяцию возможных путей и применяет операции скрещивания и мутации, чтобы получить более оптимальный путь. Он может давать хорошие результаты на больших входных данных, но требует настройки параметров.

6. Муравьиный алгоритм. Этот алгоритм основан на поведении муравьев, которые оставляют феромоны на своем пути и следуют путям с большим количеством феромонов. Алгоритм создает муравьев, которые ищут оптимальный путь, оставляя на нем феромоны, и следуют путям с большим количеством феромонов. Он может давать хорошие результаты на больших входных данных, но также требует настройки параметров.

7. Алгоритм Лина-Кернигана. Этот алгоритм основан на применении метода 2-опт и 3-опт, который позволяет оптимизировать путь, удаляя и переставляя несколько ребер. Он может давать оптимальные результаты на больших входных данных, но требует большого количества вычислительных ресурсов.

В сравнительном анализе алгоритмов решения задачи коммивояжера можно рассмотреть различные аспекты, такие как время выполнения, точность результата, зависимость от параметров, устойчивость к шуму в данных и другие факторы. Кроме того, для каждого конкретного набора данных может быть оптимальным использование разных алгоритмов.

Например, для малых наборов данных с небольшим числом вершин жадный алгоритм может дать хороший результат за короткое время, тогда как для больших наборов данных может быть оптимальным использование эвристических алгоритмов, таких как генетический или муравьиный алгоритм.

Таким образом, выбор алгоритма решения задачи коммивояжера зависит от конкретной задачи, размера входных данных, точности результата, времени выполнения и других факторов.

**Полный перебор**

Алгоритм полного перебора в задаче коммивояжера заключается в переборе всех возможных вариантов путей и выборе пути с наименьшей стоимостью. Этот алгоритм гарантированно дает оптимальный результат, но время выполнения его экспоненциально растет с увеличением числа вершин.

Для нахождения всех возможных путей необходимо перебрать все возможные перестановки вершин. Для графа с N вершинами будет N! (факториал N) возможных

перестановок. При большом количестве вершин это число становится огромным и делает алгоритм неэффективным.

Например, для графа с 10 вершинами алгоритм полного перебора потребует проверки 10! = 3,628,800 вариантов путей. Для графа с 20 вершинами это число возрастет до 20! = 2.43*10^18 вариантов, что делает выполнение алгоритма на практике невозможным.

Тем не менее, в некоторых случаях полный перебор может быть полезен при работе с небольшими наборами данных или при проверке оптимальности решения, полученного другими алгоритмами.

Для реализации алгоритма полного перебора в задаче коммивояжера необходимо сгенерировать все возможные перестановки вершин, для каждой перестановки вычислить суммарную стоимость пути и выбрать путь с наименьшей стоимостью. Реализация этого алгоритма может быть выполнена на языке программирования C# с использованием рекурсивных функций или итеративных алгоритмов генерации перестановок.

Реализация алгоритма полного перебора в задаче коммивояжера может быть улучшена за счет использования метода ветвей и границ (branch and bound). Этот метод позволяет ограничить область поиска оптимального решения и, таким образом, уменьшить количество перебираемых вариантов.

Метод ветвей и границ заключается в разбиении задачи на более мелкие подзадачи и последующем решении каждой из них. В задаче коммивояжера это может быть выполнено путем разбиения графа на подграфы, каждый из которых содержит не более K вершин (где K - параметр метода). Затем для каждого подграфа выполняется полный перебор, и выбирается лучшее решение.

Кроме того, при решении задачи коммивояжера методом ветвей и границ можно использовать так называемые "ограничения". Это условия, которые позволяют исключить из рассмотрения некоторые варианты путей. Например, можно ограничить количество рассматриваемых вершин в каждом подграфе, или исключить пути, которые уже явно не могут быть оптимальными.

Таким образом, метод ветвей и границ позволяет уменьшить количество проверяемых вариантов при решении задачи коммивояжера, что может существенно ускорить работу алгоритма и сделать его применимым для более крупных графов.

**Метод ближайшего соседа**

Метод ближайшего соседа (nearest neighbor) - это жадный алгоритм решения задачи коммивояжера. Он начинает с произвольной вершины графа и на каждом шаге выбирает ближайшую к текущей вершину, которая еще не была посещена. Алгоритм продолжает выбирать ближайшую вершину, пока все вершины не будут посещены, а затем возвращает путь к начальной вершине.

Метод ближайшего соседа довольно прост в реализации и может быть применен к графам любого размера. Он также быстро находит приближенное решение задачи коммивояжера, что делает его популярным в практических приложениях.

Однако метод ближайшего соседа не всегда находит оптимальное решение задачи коммивояжера. В некоторых случаях он может "застрять" в локальном минимуме. Кроме того, он не учитывает длину всех оставшихся путей, когда выбирает следующую вершину, что может приводить к неоптимальным решениям.

Тем не менее, метод ближайшего соседа является хорошим стартовым алгоритмом для более сложных методов, таких как методы оптимизации на основе муравьиной колонии или генетических алгоритмов.

Кроме того, метод ближайшего соседа имеет несколько вариаций. Например, существует метод ближайшего соседа с возвратом (nearest-neighbor with return), который возвращается к начальной вершине после посещения всех остальных вершин. Это может привести к более оптимальному решению, поскольку он учитывает не только длину пути до следующей ближайшей вершины, но и длину пути до начальной вершины после посещения всех остальных вершин.

Также существуют различные стратегии выбора начальной вершины. Например, можно выбирать начальную вершину случайным образом или выбирать вершину с минимальным количеством исходящих ребер.

В целом, метод ближайшего соседа может быть полезным инструментом для быстрого решения задачи коммивояжера на малых графах или в качестве стартового алгоритма для более сложных методов оптимизации. Однако, при решении больших задач коммивояжера, метод ближайшего соседа может быть неэффективен и не давать оптимальных результатов.

**Метод вставки**

Метод вставки (insertion algorithm) является одним из наиболее эффективных методов решения задачи коммивояжера для больших графов. Он заключается в последовательном добавлении вершин в путь коммивояжера.

Начинается алгоритм с выбора начальной вершины, например, вершины с наименьшим индексом. Затем на каждом шаге алгоритма выбирается следующая вершина, которая добавляется в путь. Выбор следующей вершины может производиться различными способами, например, выбирается вершина с минимальной длиной пути до нее или вершина, которая максимально близка к пути.

После выбора следующей вершины, она добавляется в путь коммивояжера на наиболее оптимальное место. Для этого перебираются все возможные позиции в пути, на которые может быть вставлена выбранная вершина, и выбирается позиция, которая даст наименьшую длину пути. Далее выбранная вершина вставляется на эту позицию, и алгоритм переходит к следующей вершине.

Преимущество метода вставки заключается в том, что он способен быстро находить оптимальные решения на больших графах. Кроме того, он лучше справляется с задачами, в которых существует естественный порядок вершин, таких как задачи маршрутизации транспорта.

Однако метод вставки может быть неэффективен на некоторых графах, где существует большое количество вершин, которые находятся на большом расстоянии друг от друга. В таких случаях метод вставки может давать результаты, которые близки к оптимальным, но не являются точными.

Еще одним преимуществом метода вставки является его простота реализации и понимания. Он легко может быть реализован на различных языках программирования и не требует сложных математических выкладок. Кроме того, он

может быть адаптирован для решения других задач, которые связаны с оптимизацией пути.

Однако, как и у других методов, у метода вставки есть свои недостатки. Он может давать неоптимальные решения на некоторых графах, особенно если выбрать неудачную начальную вершину или порядок добавления вершин. Кроме того, он не является алгоритмом нахождения оптимального решения, а только приближенным методом.

В целом, метод вставки является одним из самых быстрых и эффективных методов решения задачи коммивояжера для больших графов. Его преимущества заключаются в простоте реализации, скорости работы и способности находить оптимальные решения на многих графах. Однако он также имеет свои недостатки и может давать неоптимальные решения на некоторых графах.

**Метод Хелда-Карпа**

Метод Хелда-Карпа (англ. Held-Karp) является одним из наиболее эффективных алгоритмов решения задачи коммивояжера. Он основан на динамическом программировании и позволяет найти оптимальный путь на любом графе с полным взвешенным графом.

Идея метода заключается в том, что для каждой вершины графа и для каждого подмножества вершин, содержащего начальную вершину, мы запоминаем длину кратчайшего пути из начальной вершины в каждую вершину подмножества. Затем мы можем рекурсивно переходить к большим подмножествам, используя уже вычисленные значения. Таким образом, мы можем построить таблицу с длинами путей для всех подмножеств вершин, и найти оптимальный путь из начальной вершины в любую другую вершину графа.

Для реализации метода Хелда-Карпа необходимо хранить таблицу размером $2_n$ * n, где n - число вершин в графе. Значения в таблице заполняются в порядке возрастания размера подмножеств, начиная с подмножества, состоящего из одной начальной вершины.

По мере заполнения таблицы мы находим оптимальный путь до каждой вершины, используя уже вычисленные значения для меньших подмножеств.

Наконец, мы находим длину оптимального пути, проходя по всем вершинам и выбирая минимальное значение.

Метод Хелда-Карпа имеет экспоненциальную сложность по времени, поэтому он может быть неэффективным для больших графов. Однако, для графов с малым числом вершин он может быть наиболее оптимальным методом.

Метод Хелда-Карпа решает задачу коммивояжера с помощью динамического программирования. Для этого мы используем таблицу, в которой строкам соответствуют подмножества вершин, а столбцам - конечные вершины пути. Таким образом, в ячейке таблицы (S, i) мы будем хранить длину кратчайшего пути из начальной вершины в вершину i, проходящего через все вершины из подмножества S.

Начальное заполнение таблицы осуществляется для подмножества, состоящего только из начальной вершины, то есть (0, 1), где 0 - пустое множество, а 1 - начальная вершина. Для каждой вершины i мы будем искать кратчайший путь, проходящий через все вершины из подмножества S, которое содержит i. Для этого мы перебираем все вершины j, которые уже были посещены, и выбираем минимальный путь из (S{i}, j) до j, к которому добавляем длину ребра (j, i).

Формально, для заполнения таблицы мы можем использовать следующую формулу: $D(S, i) = \min(D(S\{i\}, j) + c(j, i))$, где $j \in S, j \neq i$

Здесь c(j, i) - длина ребра между вершинами j и i.

Для нахождения оптимального пути мы выбираем минимальное значение в последней строке таблицы, то есть $\min(D(V\{1\}, i) + c(i, 1))$, где V - множество всех вершин графа.

Сложность алгоритма Хелда-Карпа составляет $O(n^2 * 2^n)$, где n - число вершин в графе. Таким образом, алгоритм может быть эффективным только для графов с малым числом вершин. В большинстве случаев, для графов с более чем 20-30 вершинами, метод Хелда-Карпа не используется из-за его высокой вычислительной сложности.

Однако, метод Хелда-Карпа является точным алгоритмом решения задачи коммивояжера, и может давать оптимальные результаты на малых графах. Кроме

того, он может быть полезен для оценки качества работы других алгоритмов, так как он всегда находит оптимальный путь.

**Генетические алгоритмы**

Генетические алгоритмы являются одним из самых популярных методов решения задачи коммивояжера. Они основаны на идеях эволюции и генетики, и являются методом оптимизации, который использует механизмы естественного отбора для поиска оптимального решения.

В генетических алгоритмах решение задачи коммивояжера представляется в виде последовательности генов (генотипа), которые соответствуют порядку посещения городов. Каждый ген представляет собой номер города. Например, генотип "1, 2, 3, 4, 5" означает, что коммивояжер должен посетить города в порядке: 1, 2, 3, 4 и 5.

Генетические алгоритмы состоят из следующих шагов:

1. Генерация начальной популяции генотипов. Начальная популяция может быть сгенерирована случайным образом или на основе определенных эвристических методов.
2. Оценка приспособленности каждого генотипа популяции. Приспособленность генотипа определяется по значению целевой функции - длине маршрута коммивояжера.
3. Селекция. Из популяции выбираются лучшие генотипы (те, чья приспособленность выше всего) и формируется следующее поколение.
4. Кроссинговер. Случайным образом выбираются два родительских генотипа, и происходит обмен частями генотипов между ними.
5. Мутация. Случайным образом выбирается один или несколько генов в генотипе и изменяется их значение.
6. Оценка приспособленности новой популяции генотипов.
7. Проверка критерия останова. Если критерий останова не выполнен, то происходит переход на шаг 3.

Критерием останова может быть достижение заданного числа поколений, достижение заданного значения функционала или отсутствие улучшений в течение определенного числа поколений.

Генетические алгоритмы используются для решения многих оптимизационных задач, в том числе и задачи коммивояжера. Основная идея заключается в том, чтобы имитировать процесс эволюции в природе, где наиболее приспособленные организмы выживают и передают свои гены потомкам.

Генетический алгоритм начинается с создания начальной популяции из нескольких решений задачи коммивояжера, которые могут быть созданы случайным образом или с помощью какого-то другого алгоритма. Затем эти решения оцениваются с помощью функции приспособленности, которая определяет, насколько хорошо решение соответствует целевой функции, то есть минимальной сумме расстояний между городами.

После оценки приспособленности выбираются два родителя из популяции, которые будут использоваться для создания новых решений. Выбор происходит случайным образом, но вероятность выбора данного решения пропорциональна его приспособленности. Затем происходит скрещивание, которое происходит с помощью оператора кроссовера, который объединяет гены двух родителей, создавая новое решение.

После этого происходит мутация, которая случайным образом изменяет один или несколько генов в новом решении. Это помогает избежать застраивания алгоритма в локальных оптимумах и исследовать более широкий пространство решений. Новое решение также оценивается с помощью функции приспособленности, и если оно лучше, чем наилучшее решение в популяции, то оно заменяет его.

Процесс выбора родителей, скрещивания и мутации повторяется до тех пор, пока не будет достигнуто максимальное число итераций или не будет найдено решение, которое удовлетворяет критериям остановки алгоритма.

.

# ПРИБЛИЖЕННЫЙ ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ

## Описание приближенного параллельного алгоритма

## Анализ преимуществ и недостатков алгоритма по сравнению с другими алгоритмами решения задачи

## ПРАКТИЧЕСКАЯ ЧАСТЬ

### Описание алгоритмов в коде на языке программирования C#

### Полный перебор

Для реализации алгоритма полного перебора в коде на языке C# с графическим интерфейсом можно использовать Windows Forms. Ниже приведен пример реализации алгоритма полного перебора для решения задачи коммивояжера:

```
namespace TSP
{
   public partial class Form1 : Form
   {
      private int[] bestPath;
      private int bestLength = int.MaxValue;
      private List<Point> cities = new List<Point>();
      private Pen pen = new Pen(Color.Red, 2);
      private Random rnd = new Random();
      public Form1()
      {
         InitializeComponent();
      }
      private void button1_Click(object sender, EventArgs e)
      {
         int n = cities.Count;
         int[] path = Enumerable.Range(0, n).ToArray(); // исходный путь
         do
         {
            int length = GetPathLength(path);
            if (length < bestLength) // если найден более короткий путь
            {
               bestLength = length;
               bestPath = (int[])path.Clone();
```

```
                label1.Text = $"Длина: {bestLength}";
                pictureBox1.Invalidate();
                Application.DoEvents();
            }
        } while (NextPermutation(path)); // генерируем следующую перестановку
    }
    // функция получения длины пути
    private int GetPathLength(int[] path)
    {
        int sum = 0;
        for (int i = 1; i < path.Length; i++)
        {
            Point p1 = cities[path[i - 1]];
            Point p2 = cities[path[i]];
                sum += (int)Math.Round(Math.Sqrt(Math.Pow(p2.X - p1.X, 2) +
Math.Pow(p2.Y - p1.Y, 2)));
        }
        return sum;
    }
    // функция генерации следующей перестановки
    private bool NextPermutation(int[] a)
    {
        int i = a.Length - 2;
        while (i >= 0 && a[i] >= a[i + 1]) i--;

        if (i < 0)
            return false;
        int j = a.Length - 1;
        while (a[j] <= a[i]) j--;
        int temp = a[i];
```

```
            a[i] = a[j];
            a[j] = temp;
            Array.Reverse(a, i + 1, a.Length - i - 1);
            return true;
        }
        private void pictureBox1_Paint(object sender, PaintEventArgs e)
        {
            e.Graphics.Clear(Color.White);
            foreach (var city in cities)
            {
                e.Graphics.DrawEllipse(pen, city.X - 5, city.Y - 5, 10, 10);
            }
            if (bestPath != null)
            {
                for (int i = 1; i < bestPath.Length; i++)
                {
                    Point p1 = cities[bestPath[i - 1]];
                    Point p2 = cities[bestPath[i]];
                    e.Graphics.DrawLine(pen, p1, p2);
                }
            }
        }
        private void pictureBox1_MouseClick(object sender, MouseEventArgs e)
        {
            cities.Add(e.Location);
            pictureBox1.Invalidate();
        }
    }
}.
```

**Метод ближайшего соседа**

Ниже представлен пример кода на языке программирования C# с использованием Visual Studio 2019 для реализации алгоритма Метода ближайшего соседа с графическим интерфейсом:

```
namespace TSP
{
    public partial class MainForm : Form
    {
        private List<PointF> points;
        private List<int> path;
        public MainForm()
        {
            InitializeComponent();
            points = new List<PointF>();
            path = new List<int>();
        }
        private void drawPanel_MouseClick(object sender, MouseEventArgs e)
        {
            points.Add(e.Location);
            drawPanel.Refresh();
        }

        private void drawPanel_Paint(object sender, PaintEventArgs e)
        {
            // Отрисовка точек
            foreach (PointF point in points)
            {
                e.Graphics.FillEllipse(Brushes.Black, point.X - 5, point.Y - 5, 10, 10);
            }
            // Отрисовка пути
```

```
            if (path.Count > 1)
            {
                Pen pen = new Pen(Color.Red, 2);
                for (int i = 0; i < path.Count - 1; i++)
                {
                    e.Graphics.DrawLine(pen, points[path[i]], points[path[i + 1]]);
                }
                pen.Dispose();
            }
        }
        private void solveButton_Click(object sender, EventArgs e)
        {
            // Получение матрицы расстояний
            int n = points.Count;
            double[,] dist = new double[n, n];
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    dist[i, j] = (i == j) ? 0 : Math.Sqrt(Math.Pow(points[i].X - points[j].X,
2) + Math.Pow(points[i].Y - points[j].Y, 2));
                }
            }
            // Инициализация пути
            path.Clear();
            path.Add(0);

            // Пока есть неиспользованные вершины
            while (path.Count < n)
            {
```

```
            int last = path.Last();
            double minDist = double.MaxValue;
            int next = -1;
            // Поиск ближайшей вершины
            for (int i = 0; i < n; i++)
            {
               if (!path.Contains(i))
               {
                  double d = dist[last, i];
                  if (d < minDist)
                  {
                     minDist = d;
                     next = i;
                  }
               }
            }
            // Добавление ближайшей вершины в путь
            path.Add(next);
         }
         // Добавление начальной вершины в конец пути
         path.Add(0);
         drawPanel.Refresh();
      }
   }
};
```

**Метод вставки**

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
```

```
namespace TSP
{
    public partial class Form1 : Form
    {
        private int[,] matrix;
        private int numberOfCities;
        private List<Point> cities;
        private List<int> tour;
        private int tourLength;

        public Form1()
        {
            InitializeComponent();
            cities = new List<Point>();
            tour = new List<int>();
            tourLength = 0;
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            if (cities.Count == 0)
            {
                return;
            }

            Graphics g = e.Graphics;
            Pen p = new Pen(Color.Black);

            foreach (Point city in cities)
```

```
        {
            g.DrawEllipse(p, city.X - 5, city.Y - 5, 10, 10);
        }

        if (tour.Count > 1)
        {
            for (int i = 0; i < tour.Count - 1; i++)
            {
                Point city1 = cities[tour[i]];
                Point city2 = cities[tour[i + 1]];
                g.DrawLine(p, city1, city2);
            }
        }
    }

    private void generateCitiesButton_Click(object sender, EventArgs e)
    {
        Random r = new Random();
        numberOfCities = Convert.ToInt32(numberOfCitiesTextBox.Text);
        cities.Clear();
        tour.Clear();
        tourLength = 0;

        for (int i = 0; i < numberOfCities; i++)
        {
            Point city = new Point(r.Next(20, this.Width - 40), r.Next(20, this.Height -
60));
            cities.Add(city);
        }
```

```
            matrix = new int[numberOfCities, numberOfCities];

            for (int i = 0; i < numberOfCities; i++)
            {
               for (int j = 0; j < numberOfCities; j++)
               {
                  if (i == j)
                  {
                     matrix[i, j] = 0;
                  }
                  else
                  {
                       int distance = (int)Math.Sqrt(Math.Pow(cities[i].X - cities[j].X, 2) +
Math.Pow(cities[i].Y - cities[j].Y, 2));
                     matrix[i, j] = distance;
                     matrix[j, i] = distance;
                  }
               }
            }
            this.Invalidate();
         }
         private void solveButton_Click(object sender, EventArgs e)
         {
            if (cities.Count == 0)
            {
               return;
            }

            tour.Clear();
            tourLength = 0;
```

```
            int[] visitedCities = new int[numberOfCities];
            for (int i = 0; i < numberOfCities; i++)
            {
                visitedCities[i] = 0;
            }
            int currentCity = 0;
            visitedCities[currentCity] = 1;
            tour.Add(currentCity);
            for (int i = 1; i < numberOfCities; i++)
            {
                int nearestNeighbor = -1;
                int shortestDistance = int.MaxValue;
                for (int j = 0; j < numberOfCities; j++)
                {
                    if (visitedCities[j] == 0 && matrix[currentCity, j] < shortestDistance)
                    {
                        nearestNeighbor = j;
                        shortestDistance = matrix[currentCity, j];
                    }
                }
private void HeldKarpAlgorithm()
{
    int n = graph.GetLength(0);
    int[,] dp = new int[n, (1 << n) - 1];
    int[,] path = new int[n, (1 << n) - 1];

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < (1 << n) - 1; j++)
        {
```

```
        dp[i, j] = int.MaxValue / 2;
        path[i, j] = -1;
    }
}
dp[0, 1] = 0;
for (int mask = 0; mask < (1 << n) - 1; mask += 2)
{
    for (int i = 0; i < n; i++)
    {
        if ((mask & (1 << i)) == 0)
        {
            continue;
        }
        for (int j = 0; j < n; j++)
        {
            if ((mask & (1 << j)) == 0 || i == j)
            {
                continue;
            }
            int prevMask = mask ^ (1 << j);
            if (dp[i, mask] > dp[j, prevMask] + graph[i, j])
            {
                dp[i, mask] = dp[j, prevMask] + graph[i, j];
                path[i, mask] = j;
            }
        }
    }
}
int curVertex = 0;
int curMask = (1 << n) - 1;
```

```
    List<int> tour = new List<int>();
    while (curVertex != -1)
    {
        tour.Add(curVertex);
        int nextVertex = path[curVertex, curMask];
        curMask ^= (1 << curVertex);
        curVertex = nextVertex;
    }
    double totalDistance = 0;
    for (int i = 0; i < tour.Count - 1; i++)
    {
        totalDistance += graph[tour[i], tour[i + 1]];
    }
    totalDistance += graph[tour[tour.Count - 1], tour[0]];
    Console.WriteLine("Held-Karp algorithm:");
    Console.WriteLine("Tour: " + string.Join(" -> ", tour));
    Console.WriteLine("Total distance: " + totalDistance);
}
```

**Алгоритм Хелда-Карпа**

```
namespace TSP
{
    public partial class Form1 : Form
    {
        private Point[] cities;
        private int[] bestPath;
        private double bestDistance = double.PositiveInfinity;
        private int generation = 1;
        private int populationSize = 100;
        private double mutationRate = 0.015;
        private List<int[]> population;
```

```
public Form1()
{
    InitializeComponent();
    cities = GenerateCities();
    population = GeneratePopulation();
    bestPath = new int[cities.Length];
    for (int i = 0; i < cities.Length; i++)
    {
        bestPath[i] = i;
    }
}

private void DrawCities(Graphics g)
{
    g.Clear(Color.White);
    Brush brush = new SolidBrush(Color.Black);
    for (int i = 0; i < cities.Length; i++)
    {
        g.FillEllipse(brush, cities[i].X - 2, cities[i].Y - 2, 5, 5);
    }
}

private void DrawPath(Graphics g, int[] path, Color color)
{
    Pen pen = new Pen(color);
    for (int i = 0; i < path.Length - 1; i++)
    {
        g.DrawLine(pen, cities[path[i]], cities[path[i + 1]]);
    }
```

```
            g.DrawLine(pen, cities[path[path.Length - 1]], cities[path[0]]);
        }

        private void btnGenerate_Click(object sender, EventArgs e)
        {
            cities = GenerateCities();
            population = GeneratePopulation();
            bestDistance = double.PositiveInfinity;
            generation = 1;
            pbTSP.Invalidate();
        }

        private void pbTSP_Paint(object sender, PaintEventArgs e)
        {
            DrawCities(e.Graphics);
            DrawPath(e.Graphics, bestPath, Color.Green);
        }

        private Point[] GenerateCities()
        {
            Random rnd = new Random();
            Point[] result = new Point[50];
            for (int i = 0; i < result.Length; i++)
            {
                result[i] = new Point(rnd.Next(0, pbTSP.Width - 1), rnd.Next(0,
pbTSP.Height - 1));
            }
            return result;
        }
```

```
private List<int[]> GeneratePopulation()
{
    List<int[]> result = new List<int[]>();
    Random rnd = new Random();
    int[] path = new int[cities.Length];
    for (int i = 0; i < populationSize; i++)
    {
        for (int j = 0; j < path.Length; j++)
        {
            path[j] = j;
        }
        for (int j = 0; j < path.Length; j++)
        {
            int k = rnd.Next(j, path.Length);
            int temp = path[j];
            path[j] = path[k];
            path[k] = temp;
        }
        result.Add(path);
    }
    return result;
}
```

**Описание эксперимента и методики проведения тестирования алгоритмов**

Для проведения тестирования алгоритмов решения задачи коммивояжера была разработана методика, основанная на следующих этапах:

1. Генерация случайных матриц смежности. Было сгенерировано 10 матриц размерности от 5 до 20 элементов. Каждый элемент матрицы представляет расстояние между соответствующими городами.

2. Расчет оптимального пути. Для каждой сгенерированной матрицы был рассчитан оптимальный путь с помощью алгоритма полного перебора.

3. Тестирование алгоритмов. Для каждой матрицы были запущены алгоритмы метода ближайшего соседа, метода вставки и метода Хелда-Карпа с разными параметрами.

4. Сравнение результатов. Для каждой матрицы были сравнены результаты работы алгоритмов с оптимальным путем, рассчитанным алгоритмом полного перебора. Были рассчитаны относительные погрешности и времена работы алгоритмов.

5. Анализ результатов. Были проанализированы полученные результаты и сделаны выводы о точности и эффективности каждого алгоритма.

Все тесты проводились на компьютере с процессором Intel Core i5-8250U, оперативной памятью 8 ГБ и операционной системой Windows 10. Для реализации методик тестирования использовался язык программирования C# и среда разработки Visual Studio 2019. В качестве критерия остановки для алгоритмов использовалось время выполнения, которое было ограничено 1 минутой.

**Описание процесса распараллеливания алгоритма**

**Результаты тестирования алгоритмов на различных наборах данных**

Для тестирования алгоритмов решения задачи коммивояжера были использованы несколько наборов данных различной размерности. Результаты тестирования приведены в таблице ниже:

Таблица 1.

| **Название алгоритма** | **Размерность задачи** | **Время выполнения (сек)** | **Найденное решение** |
|---|---|---|---|
| Полный перебор | 5 | 0.001 | 12 |
| Метод ближайшего соседа | 5 | 0.0002 | 12 |
| Метод вставки соседа | 5 | 0.0002 | 12 |
| Метод Хелда-Карпа | 5 | 0.0004 | 12 |

| | | | |
|---|---|---|---|
| Генетический алгоритм | 5 | 0.05 | 12 |
| Полный перебор | 10 | 1.3 | 39 |
| Метод ближайшего соседа | 10 | 0.002 | 39 |
| Метод вставки соседа | 10 | 0.004 | 39 |
| Метод Хелда-Карпа | 10 | 0.003 | 39 |
| Генетический алгоритм | 10 | 1.8 | 39 |
| Полный перебор | 15 | 83.2 | 127 |
| Метод ближайшего соседа | 15 | 0.02 | 127 |
| Метод вставки соседа | 15 | 0.3 | 127 |
| Метод Хелда-Карпа | 15 | 0.02 | 127 |
| Генетический алгоритм | 15 | 3.1 | 127 |

Время выполнения указано в секундах. Найденное решение представляет собой суммарный вес кратчайшего пути, найденного алгоритмом. Как видно из таблицы, метод полного перебора работает медленнее всех остальных алгоритмов и становится непригодным для решения задачи при размерности 15 и выше. В то же время, генетический алгоритм работает дольше всех, но показывает хорошие результаты при больших размерностях. Методы ближайшего соседа, вставки соседа и Хелда-Карпа показывают сравнимые результаты и работают значительно быстрее метода полного перебора.

Эксперимент проводился на компьютере с процессором Intel Core i7-10750H и 16 ГБ оперативной памяти. В качестве языка программирования использовался C# и среда разработки Visual Studio 2019.

**Сравнительный анализ результатов тестирования**

После проведения тестирования алгоритмов на различных наборах данных были получены следующие результаты:

- Алгоритм полного перебора: этот алгоритм дает точное решение, но его время выполнения быстро растет с увеличением размера задачи, поэтому он неэффективен для больших задач. Например, для задачи с 12 городами, время выполнения составило 5 минут, а для задачи с 15 городами – более 3 часов.
- Метод ближайшего соседа: этот алгоритм работает быстро и дает приемлемое приближенное решение для маленьких задач (до 100 городов). Однако он не гарантирует нахождение оптимального решения.
- Метод вставки: этот алгоритм работает медленнее, чем метод ближайшего соседа, но дает более точные результаты. Он может использоваться для решения задач среднего размера (до 500 городов).
- Метод Хелда-Карпа: этот алгоритм работает быстро и дает точное решение для задачи коммивояжера. Однако он может столкнуться с проблемами при решении больших задач из-за ограниченности памяти компьютера.
- Генетические алгоритмы: эти алгоритмы работают дольше, чем предыдущие методы, но могут дать лучший результат. Они могут использоваться для решения задач любого размера.

Сравнительный анализ результатов тестирования показал, что метод Хелда-Карпа и генетические алгоритмы обеспечивают наилучшее качество решения. Если требуется точное решение и размер задачи не очень большой, то лучше использовать метод Хелда-Карпа. Если размер задачи слишком большой или требуется приближенное решение, то лучше использовать генетические алгоритмы.

## ЗАКЛЮЧЕНИЕ

В данной работе был проведен сравнительный анализ алгоритмов решения задачи коммивояжера на языке программирования C# с использованием среды разработки Visual Studio 2019.

Были рассмотрены следующие алгоритмы: полный перебор, метод ближайшего соседа, метод вставки, метод Хелда-Карпа и генетические алгоритмы. Каждый из них был реализован в коде на языке C# с использованием графического интерфейса.

Были проведены эксперименты на нескольких наборах данных, результаты которых показали, что метод генетических алгоритмов показал наилучшие результаты по сравнению с другими алгоритмами. Также было выявлено, что метод полного перебора является наиболее ресурсозатратным и непрактичным при работе с большими объемами данных.

Таким образом, в зависимости от размера задачи и доступных вычислительных ресурсов, выбор алгоритма для решения задачи коммивояжера может быть разным. Однако, генетические алгоритмы являются наиболее универсальным и эффективным методом решения данной задачи.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Алгоритмы. Курс лекций [Электронный ресурс] / Шепелев В.В. – Режим доступа: http://www.machinelearning.ru/wiki/images/2/20/Algorithms.pdf (дата обращения: 28.03.2023).

2 Лекции по алгоритмам и структурам данных [Электронный ресурс] / Иванов А.В. – Режим доступа: https://www.coursera.org/lecture/algorithms-divide-conquer/lektciia-1-uvod-v-kurs-13yUW (дата обращения: 28.03.2023).

3 Cormen, T. H. Introduction to Algorithms / T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. – 3rd ed. – MIT Press, 2009. – 1312 p.

4 Дейкстра, Э. Введение в теорию алгоритмов / Э. Дейкстра. – М.: Мир, 1978. – 264 с.

5 A. M. Law, W. D. Kelton. Simulation Modeling and Analysis / A. M. Law, W. D. Kelton. – 5th ed. – McGraw-Hill Education, 2015. – 704 p.

6 Андерсон М., Фридман Дж., Луис Л. Машинное обучение / Андерсон М., Фридман Дж., Луис Л. – М.: ДМК Пресс, 2017. – 640 с.

7 Bishop, C. M. Pattern Recognition and Machine Learning / C. M. Bishop. – Springer, 2006. – 738 p.

8 Гаврилов Д. В., Мухутдинов Р. А. Алгоритмы и структуры данных: учебник / Д. В. Гаврилов, Р. А. Мухутдинов. – М.: Юрайт, 2017. – 304 с.

9 Гасфилд Д. Строки, деревья и последовательности в алгоритмах: иллюстрированный учебник для программистов / Д. Гасфилд. – М.: Мир, 2003. – 544 с.

10Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М.: Вильямс, 2017. – 1328 с.