«Задача коммивояжера»

Описание предлагаемого решения

Выполнил: ст. гр. ПЗПИ-16-1, Асландуков Матвей

Как известно, задача коммивояжера является NP-трудной, а значит даже при небольших *N* уже невозможно найти гарантированно оптимальный маршрут за разумное время. Однако, при совсем маленьких *N* возможно применение экспоненциальных алгоритмов.

Экспоненциальные алгоритмы

Самым простым экспоненциальным решением является полный перебор всех *(N – 1)!* возможных маршрутов и выбор самого короткого из них. Такое решение имеет временную сложность *O(N!)*, из-за чего может быть применено при *N* <= 10.

Чуть более трудным является решение с помощью динамического программирования по маскам. А именно, пусть *dp[mask][last]* – минимальная длина маршрута, посещающего все вершины из *mask* и заканчивающегося в городе *last.* Такую динамику можно довольно просто пересчитывать. Для этого необходимо перебрать следующий город *to* и сделать обновление: *dp[mask | (1 << to)][to] min= dp[mask][last] + d[last][to].* Это решение требует уже *O(2nn2)* времени и *O(2nn)* памяти, поэтому может быть применено при *N <= 20.* Стоит отметить, что данное решение работает для незамкнутого варианта задачи, однако замкнутый вариант можно свести к незамкнутому. Для этого не обходимо добавить мнимую вершину с номером (*N + 1)* и провести следующие ребра:

* *(N + 1) => v* ребро стоимостью *3 \* inf;*
* *1 => (N + 1)* ребро стоимостью *3 \* inf;*
* *v => (N + 1)* ребро стоимостью *dv,1 + 2 \* inf (*здесь *inf*  - это число, заведомо большее длины оптимального маршрута).

Нетрудно заметить, что в таком графе оптимальный путь всегда будет заканчиваться в вершине *(N + 1)* и иметь стоимость ровно на *2 \* inf* большую оптимальной стоимости замкнутого пути.

Приближенные решения

Т.к. при больших *N,* экспоненциальные решения уже не смогут отработать, то задачу можно пытаться решить только приближенно.

Во всех приближенных решениях можно выделить два этапа:

* нахождение «стартового» маршрута;
* последовательное улучшение стартового маршрута методом локальных оптимизаций.

Алгоритмы выбора «стартового» маршрута.

Самым первым и тривиальным алгоритмом является выбор **случайного маршрута** среди всех *(N – 1)!* возможных. Небольшой его модификацией является выбор лучшего маршрута среди *K* случайных, где параметр *K* можно варьировать, в зависимости от требуемого времени работы. Т.к. большинство дальнейших методов будут иметь сложность порядка *O(N2)*, то мы положим *K = N,* получив тем самым такую же асимптотику времени работы.

**Метод ближайшего соседа** – на каждом из *(N – 1)* шагов будем переходить в такую вершину, расстояние до которой от текущей минимально Этот алгоритм очень просто реализуется и требует *O(N2)* времени.

**Битонический коммивояжер**. Т.к. поиск оптимального маршрута среди всех является *NP-*трудной задачей, можно попытаться как-то ограничить множество маршрутов, среди которых мы ищем оптимум. Одним способом это сделать – является рассмотрение только «битонических» путей. «Битонический» путь – это путь, который сначала идет слева направо (нижняя часть пути), а затем справа налево (верхняя часть). Примечательно то, что оптимальный «битонический» путь уже можно найти за полиномиальное время. Сделать это можно за время *O(N2)* с помощью динамического программирования. А именно, давайте отсортируем точки по возрастанию *x*-координаты и будем последовательно их добавлять то ли к нижней, то ли к верхней части пути. Получаем следующую динамику: *dp[i][j]* – минимальная дина пути, верхняя часть которого заканчивается в вершине *i*, а правая – в *j*. Довольно просто ее можно пересчитывать за *O(1):* из каждого состояния будет ровно два перехода, соответствующих добавлению очередной вершины *max(i, j)* то ли в верхнюю, то ли в нижнюю часть пути.

**Метод минимального остовного дерева**. Давайте выделим минимальное остовное дерево в нашем графе, проведем каждое ребро в двух направлениях, и рассмотрим эйлеров цикл в этом дереве (он всегда будет существовать, т.к. после раздвоения ребер, степень каждой вершины стала четной). Далее оставим в этом цикле только первое вхождение каждой вершины и вернем полученный замкнутый путь в качестве искомого.

Теперь ответим на вопрос, почему этот метод будет хорошо работать. Обозначим суммарный вес минимального остовного дерева как *W*, а длину кратчайшего маршрута – *Lopt*. Заметим, что если из кратчайшего маршрута удалить произвольное ребро, то мы получим дерево, а значит *W* <= *Lopt.* Осталось понять, что после того, как мы удалили некоторые вершины из нашего обхода, суммарная длина не увеличилась, что следует из неравенства треугольника. Таким образом, итоговая длина пути *L* <= *Le* = *2W* <= *2 Lopt.* Это значит, что данный алгоритм дает 2-приближение исходной задачи (т.е. ошибается не более, чем в 2 раза).

Оценим затраты по времени и памяти этого алгоритма. Минимальное остовное дерево можно построить за время *O(N2)* алгоритмом Прима, а получить эйлеров обход можно с помощью обычного *dfs-а* за *O(N)*. Итоговые затраты времени – *O(N2)*. В свою очередь памяти нужно всего *O(N)* для хранения итогового пути и вспомогательных массивов для работы алгоритма Прима.

**Метод минимального паросочетания.** Этот метод является небольшой модификацией предыдущего. Посмотрим, как можно сделать граф эйлеровым чуть более «умным» способом. Для этого рассмотрим все вершины нечетной степени. Т.к. общая степень всех вершин – четная, то кол-во нечетных вершин – четно. Поэтому можем добавить совершенное паросочетание минимального веса на этих нечетных вершин, после чего степень каждой вершины станет четной.

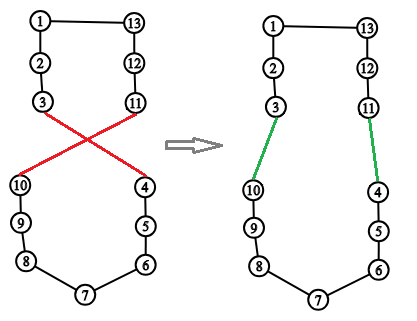
Докажем, что этот метод дает *1.5-приближение* исходной задачи. Для этого докажем, что вес найденного паросочетания *Lm* не будет превосходить 0.5 *Lopt*. Действительно, рассмотрим сокращенный оптимальный маршрут, т.е. такой, в котором останутся только нечетные вершины. Заметим, что его длина *Ls* <= *Lopt*. Также заметим, что его можно разбить на 2 составляющие – четные и нечетные ребра. Каждая из частей будет являться паросочетанием, а значит его вес будет *Ls* >= *Lm* + *Lm*. Получили, что *Lm* <= 0.5 *Ls* <= 0.5 *Lopt*, ч.т.д.

Оценим затраты по времени и памяти этого алгоритма. Минимальное остовное дерево можно построить за время *O(N2)* все тем же алгоритмом Прима, а получить совершенное паросочетание минимального веса в полном графе на нечетных вершинах можно с помощью жадного алгоритма также за время *O(N2)*. Дальнейший поиск эйлерового цикла требует линейного времени, а значит итоговые затраты времени – *O(N2).* Памяти нужно всего лишь *O(N)* для хранения итогового пути и вспомогательных массивов для работы алгоритма Прима / выделения минимального паросочетания.

Методы локальных оптимизаций

После того, как мы нашли стартовое решение, его можно попытаться улучшить методом локальных оптимизаций. А именно, будем итеративно «немного» изменять найденный маршрут таким образом, чтобы его итоговая длина постоянно уменьшалась.

**2-opt.** Этот метод заключается в следующем. Попытаемся удалить какие-то два ребра и заменить их двумя более короткими. Заметим, что если рассматривать путь как массив из *N* чисел, то такая замена будет означать простое переворачивание подотрезка этого массива.



Например, в данном случае, будет такое изменение массива:

1, 2, 3, [4, 5, 6, 7, 8, 9, 10], 11, 12, 13

1, 2, 3, [10, 9, 8, 7, 6, 5, 4], 11, 12, 13

Таким образом, на каждой итерации мы будем пытаться находить пару ребер, дающее максимальное улучшение и применять его. Если же все пары только ухудшают итоговую длину, то алгоритм заканчивает свою работу. Заметим, что т.к. одна итерация заключается в переворачивании подотрезка массива, то с помощью определенной последовательности итераций мы сможем получить абсолютно любой маршрут, и в частности – оптимальный. Это дает нам повод надеется, что данный алгоритм приведет нас к неплохому результату.

Оценим затраты по времени и памяти этого алгоритма. В наивной реализации каждая итерация будет требовать квадратичного времени работы для нахождения оптимальной пары ребер и линейного времени для переворачивания подотрезка. Таким образом, общее время работы будет равняться *O(Iter \* N2)*, где *Iter* – количество итераций до тех пор, пока метод не сойдется. Памяти же необходимо *O(N)* для хранения массива городов и текущего пути.

Однако, данный алгоритм помогает ускорить следующая идея Легко увидеть, что переворачивание одного подотрезка, никак не влияет практически на все остальные пары ребер (кроме тех, которые соприкасаются с границами отрезка) Поэтому можно заранее найти все пары ребер, замена которых приносит улучшение, отсортировать их в порядке убывания по улучшению, и поочередно пытаться их добавлять В таком случае практически все хорошие пары, которые мы найдем, мы сможем заменить, а значит за *O(N2)* мы сделаем сразу множество улучшений. На практике, это дает ускорение примерно в *O(N)* раз.

**k-opt.** Нетрудно понять, что вместо того, чтобы пытаться заменять одну пару ребер другой, можно также заменить тройки, четверки, и т.д. Таким образом, k-opt – это обобщение метода 2-opt на случай произвольных k. Стоит отметить, что при увеличении k на 1, метод будет работать в N раз больше, что на практике означает непригодность метода при k > 3.

Итого

Таким образом, всего предлагается 5 алгоритмов поиска стартового маршрута, каждый из которых можно либо не улучшать, либо улучшить методом *2-opt,* либо улучшить методом *3-opt*. Итого получаем 5 \* 3 = 15 возможных комбинаций итогового алгоритма.

Также стоит отметить, что в случае маленьких значений *N (1 <= N <= 20),* у нас всегда есть возможно применить экспоненциальный алгоритм для нахождения гарантированно оптимального ответа.