Метрическая задача коммивояжёра–2

Алексеев Николай, Б05-023

14 декабря 2022 г.

Содержание

1 Введение

В данной статье предлагается несколько алгоритмов, позволяющих найти приближенное решение задачи поиска гамильтонова пути, которая принадлежит классу **NP-полных** (гамильтонов цикл можно считать сертификатом). Эта задача при большом числе вершин не может быть решена простым перебором, поэтому необходимо испольпользовать различные алгоритмы оптимизации.

2 Постановка задачи

Пусть теперь надо найти гамильтонов путь между двумя фиксированными вер- шинами минимального веса.

Задание

(а) Постройте алгоритм, дающий 3 5 -приближение для указанной задачи, также на основе остовного дерева и паросочетания.

(б) Имплементируйте этот алгоритм.

Если имеем дело с циклом: коммивояжеру необходимо посетить все города по одному разу по построенным дорогам и вернуться в исходный.

Пусть дан граф $G = G(V, E)$, где $n = |V|$ вершин с весами и $e = |E|$ ребер. Назовем путь между вершинами s и t **гамильтоновым**, если он проходит по всем вершинам ровно пути ровно один раз таким образом, чтобы сумма весов ребер, принадлежащих пути, была минимальной. Мы будем решать **метрическую** задачу, то есть будем работать в пространстве с метрикой. Иными словами, выполнено неравенство треугольника.

3 NP-полнота задачи поиска гамильтонова пути в графе.

3.1 Лемма 1

Язык TSP =$(G, w, s, t)|$ **во взвешенном графе (G,w) есть путь коммивояжера из** s **в** t **длины не более** l **– NP-полный.**

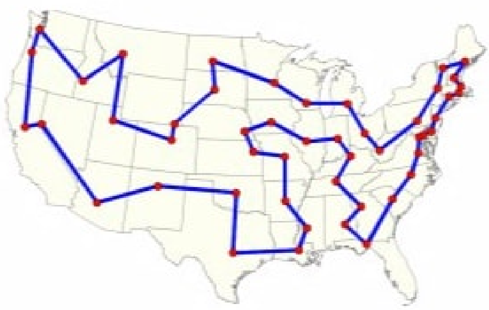
Доказательство: сведем $UHAMPATH$ к TSP. Для перевода одной задачи в другую сопоставим каждому ребру во взвешенном графе одинаковые веса, тогда $l = n - 1$, здесь n – число вершин. Если нужно найти цикл, то берем ровно по количеству вершин. Действительно, путь из $n - 1$ ребра, проходящий через все вершины, обязан быть гамильтоновым (известное утверждение из курса дискретного анализа).

3.2 Замечание

Иногда для формулировки задачи коммивояжера требуется полнота графа. Заметим, что это никак не повлияет на суть задачи – достаточно поставить добавленным ребрам очень большие веса и тогда оптимальный путь через них заведомо не пройдет.

4 Приближенное решение задачи. Оптимизация с помощью паросочетания и минимального остовного дерева

Данная задача может быть решена переборным алгоритмом. Рассмотрим, например, карту обхода столиц штатов США:



Посчитаем число комбинаций для перебора – $\frac{(49-1)!}{2}$, что примерно равно$6,2\cdot 10^{60}$. Кажется, это не очень простая задача даже для самого современного компьютера. Поэтому будем решать приближенные, но более оптимальные варианты задачи о поиске гамильтонова пути во взвешенном графе. В данной статье мы рассматриваем только один метод оптимизации. Существуют и другие. (например, метод ветвей и границ).
$\frac{5}{3}$ – приближение для указанной задачи на основе остовного дерева и паросочетания.

4.1 План работы

Прежде чем приступать к написанию кода, опишем алгоритм, дающий $\frac{5}{3}$ – приближение.

1. Посмотрим минимальное остовное дерево MST. В данной задаче будем использовать алгоритм Крускала
   Этот алгоритм работает за полиномиальное время от $|V|$.
   Напомним как работает алгоритм Крускала[1]:
   (a) Изначально создаем n деревьев, в каждом по одной вершине из графа
   (b) Сортируем все ребра по весу в порядке неубывания.
   (c) Начинаем объединять деревья, созданные в пункте 1. Проходимся по всем рёбрам от первого до последнего. В случае, если у некоторого ребра концы лежат в различных поддеревьях, то поддеревья объединяются и ребро добавляется к ответу.
   (d) после окончания перебора все вершины окажутся в одном дереве и MST для данного графа будет построено.
2. Формируем список вершин W, содержащий вершины нечетных степеней.

[1]https://e-maxx.ru/algo/

3. Строим совершенное паросочетание $Match$ на подграфе, содержащем вершины W, с помощью алгоритма Эдмонса

4. Теперь составим новый граф G‘, состоящий из ребер из MST U $Match$. Заметим, что полученный граф будет *эйлеровым*[2]., начало – наша вершина s, а конец – наша вершина t. Полученный граф назовем $EulerGraph$

5. Совершим обход по $EulerGraph$. Путь, который мы найдем, обозначим $EulerPath$.

6. Сконструируем гамильтонов цикл $AlgoHamPath$ по вершинам исходного графа G в порядке, в котором они встречаются в списке вершин $EulerPath$

7. вернем полученный список $AlgoHamPath$

[2]*– в графе G‘ окажутся две вершины нечетной степени по построению (пункты 1-3) Описанный выше алгоритм, по сути, является модификацией алгоритма Кристофидеса, предложенной в 2002 году [4]

4.2 Доказательство оценки $\frac{5}{3}$

Доказательство:

Пусть оптимальный путь – $OptimalHamPath$. Функция $weight(Path)$ возвращает вес пути $Path$. Так как имеем дело с евклидовым пространством, то известно, что $weight(AlgoHamPath) <= weight(MST) + weight(Match)$. В результате выделения вершин нечетной степени получили, что их $|V`| = 2k$ от исходного графа, так как сумма всех степеней вершин должна быть четной. На множестве $|V`|$ находится идельное паросочетание минимального веса: k ребер, имеющие минимальный суммарный вес и покрывающие все вершины. Эта задача решается полиномиальным алгоритмом. Также вес паросочетания не превосходит веса половины ребер гамильтонова цикла. Теперь рассмотрим три таких подмножества, в которых содержатся идеальные паросочетания и их объединение есть MST U $OptimalHamPath$. Вершины обозначим v_i. Пусть хотим найти путь из s в t. Составим подграф S составим из ребер, которые ведут из v_{2l-1} в v_{2l} для всех допустимых l от 1 до |V`|-2.Удалим этот подграф. Рассмотрим граф на оставшихся вершинах, в нем содержится Эйлеров цикл, который мы можем разбить на два подграфа – $S`, S``$. Следуя описанному выше алгоритму, получим два совершенных паросочетания на W. Итого разбили исходное есть MST U $OptimalHamPath$ на три подмножества и тогда верно, что $weight(Match) <= \frac{2}{3} weight(OptimalHamPath)$, откуда $weight(AlgoHamPath) <= weight(MST) + weight(Match) <= \frac{2}{3} weight(OptimalHamPath) + weight(OptimalPath) = \frac{5}{3} weight(OptimalHamPath)$
Что и требовалось показать.

4.3 Проверка времени работы и эффективности

В качестве примера возьмем с сайта данные [3] о городах Катара. Как мы видим, их 194. С того же сайта возьмем длину оптимального цикла – 9352km[4].. Таким образом, наш алгоритм должен выдавать гамильтонов путь, вес которого не превосходит $9352 \cdot \frac{5}{3} = 15586.(6)$ **анализ на основе алгоритма Кристофидеса-Сердюкова** Будем реализовывать алгоритм на языке *Python*: **Реализация алгоритма**:

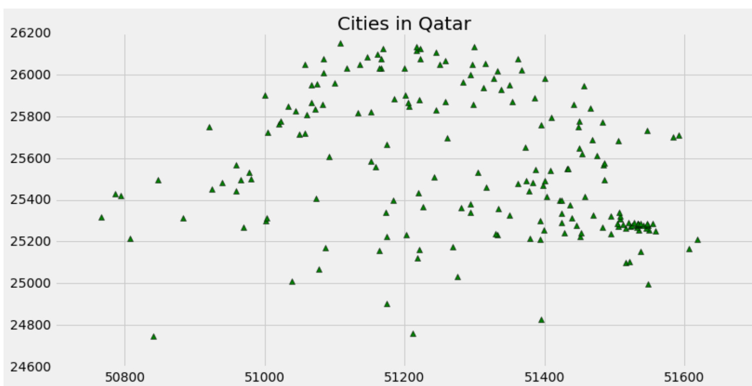
1. используемые библиотеки

```
import networkx as nx
import matplotlib.pyplot as plt
import networkx.algorithms.approximation
import networkx.algorithms.euler
import numpy as np
from scipy.spatial import distance
```

[3]http://www.math.uwaterloo.ca/tsp/world/qa194.tsp
[4]http://www.math.uwaterloo.ca/tsp/world/qatour.html

2. считываем граф из датасета с сайта с городами Катара и рисуем график

```
Coord_X, Coord_Y = [], []
i = 0
for line in open('input.txt', 'r').readlines():
    if (i<7):
        i += 1
        pass
    else:
        i += 1
        line = line.strip()
        #print(line)
        str = line.split(' ')
        try:
            if (len(str)>2):
                Coord_Y.append(str[1])
                Coord_X.append(str[2])
        except ValueError:
            pass
print("Всего городов:", i)
plt.figure(figsize=(12, 6))
plt.plot(Coord_X, Coord_Y, 'g^')
plt.title("Cities in Qatar")
```



3. Преобразуем граф в объект типа Graph из библиотеки Networkx:

```
    for i in range(len(Coord_X)):
    for j in range(len(Coord_Y)):
        G.add_edge(i,j,weight=np.sqrt((float(Coord_X[i])-float(Coord_X[j]))**2+
            (float(Coord_Y[i])-float(Coord_Y[j]))**2))
```

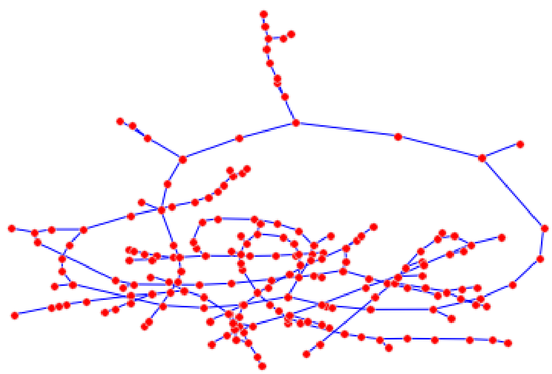
таким образом, все города занумерованы в порядке входа, а веса на соответствующих ребрах – евклидово расстояние между городами.

4. алгоритм Крускала

```
def Kruskal(G, pos):
    #вспомогательная функция для поиска минимального ребра:
    def minimum_edge(G, mst_bool):
        min = sys.maxsize  # assigning largest numeric value to min
        for i in [(u, v, len(G.nodes())) for u, v in G.edges( data = True)]:
            if mst_bool[i] == False and i[2] < min:
                min_edge = i
                min = i[2]

        return min_edge
    #вспомогательная вершина для поиска корня:
    def Root(p, i):
        if p[i] == i:
            return i
    return findRoot(parent, parent[i])
    p,o = [None] * l, [None] * l # вершина, предшествующая вершине i в MST,
    # порядок #следования вершин в MST
    l = len(G.nodes()) #число вершин
    mst_edges,mst_bool = [] # mst -- ребра остова, mst_bool[i] true,
    #если i-я вершина содержится в MST
    for i in [ (u, v, len(G.nodes())) for u, v in G.edges(data = True)]:
        mst_bool[i] = False
    for v in range(l):
        p[v] = v
        o[v] = 0
        # шаг алгоритма до тех пор, пока не получим полное дерево,
        #т.е. не обойдем ребра:
    while len(mst_edges) < l - 1 :
        curr = minimum_edge(G, mst_bool) # находим ребро минимального веса
        mst_bool[curr_edge] = True
        y = Root(p, curr[1])
        x = Root(p, curr[0])
        # добавляем в остов, если они не образуют цикл (не лежат в одном дереве)
        if x != y:
            mst_edges.append(curr_edge)
    return nx.Graph(mst_edges, mst_bool)
```

Кстати, вот как выглядит **MST для городов Катара:**



5. идеальное паросочетание
   Этот алгоритм реализуется при помощи анонсированного выше **алгоритма Эдмондса** Напомним основные положения:

   (a) Сначала введем несколько вспомогательных определений.

      i. **Паросочетание** – это набор попарно несмежных рёбер.
      ii. **Совершенное паросочетание** – это паросочетание, в котором участвуют все вершины графа.
      iii. **Чередующаяся цепь** – простая цепь, где чередуются рёбра, принадлежащие/не принадлежащие пути
      iv. **Увеличивающая цепь** – чередующая цепь, такая, что ее первая и последняя вершины не принадлежат паросочетанию.
      v. **Теорема Бержа** Паросочетания является наибольшим тогда и только тогда, когда для него не существует увеличивающей цепи[5]
      vi. G‘ обозначим симметрическую разность максимального паросочетания в G и некоторого фиксированного паросочетания
      vii. **Теорма Эдмондса** В графе G‘ существует увеличивающая цепь тогда и только тогда, когда она существует в G [6]
      viii. **цветок** в алгоритме Эдмондса – подграф, образованный циклом нечетной длины. В цикле есть ровно одна вершина, не насыщенная ребрами цикла, ее назовем **стеблем**

   (b) алгоритм Эдмондса

[5] http://www.mscs.dal.ca/ janssen/4115/presentations/Poppy.pdf
[6] J. Edmonds. Path, trees, and flowers. Canadian J. Math., 17:449–467, 1965.

i. найти все "цветки"в графе/

ii. производится **сжатие цветков** – сжатие всего нечетного цикла в одну псевдо-вершину с сохранением всех инцидентных ребер. Сжать все найденный в п.1 цветки. Теперь не осталось циклов нечетной длины.

iii. ищем увеличивающую цепь при помощи DFS (обхода в глубину).

iv. Производим "разворачивание"всех цветков, тем самым восстанавливая цепь в исходном графе.

Замечание этот алгоритм реализован в языке **Python** – minmaximalmatching(G)

6. Алгоритм Кристофидеса переходим к алгоритму, описанному в параграфе **4.1**. Поскольку нам достоверно известен циклический обход городов, то будем осуществлять его, то есть на вход алгоритму дадим $s = t = 1$:

   (a) MST построено при помощи алгоритма Крускала.

   (b) Сформируем список вершин W

```
W = [] # полагаем список пустым
MST = Kruskal(G,nx.draw(G)
for n in (Kruskal(G,nx.draw(G)).nodes():
    #заметим, что для нашего конкретного случая s=t
    if len(MST[n]) % 2 == 1 and not( n== s or n == t):
        W.append(n)
        #также надо проверить крайние ситуации
if len(MST[s]) % 2 == 0:
    MST.append(s)
if len(T[t]) % 2 == 0:
    MST.append(t)
    #отсортируем на $O(n log(n))$ с помощью $TeamSort$.
    #Также можно использовать $RadixSort$
MST = np.sort(MST)
```

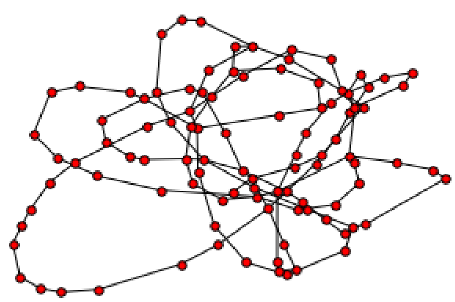
   (c) Теперь перейдем к поиску совершенного паросочетания на подграфе, содержащем вершины W при помощи алгоритма Эдмондса

```
G_ = nx.subgraph(G, W)
 #так как nx.subgraph передает неизменяемый объект
G_ = G_.copy()
#удалим ребра согласно алгоритму
G_.remove_edges_from((MST).edges())
#тут можно использовать реализацию алгоритма Эдмондса в Питоне
Ideal_Matching = Edmonds(G_)
```

теперь найдем объединение MST и $IdealMatch$. Этот граф будет Эйлеровым

```
_G = np.concatenate(np.array(MST.edges()),np.array(Ideal_Match))
for edge in _G.edges():
    if not(edge in Q):
      _G.remove(egde[0],edge[1])
```

Получили:
(0, 34), (0, 91), (1, 32), (1, 3), (1, 44), (2, 18), (3, 40), (4, 24), (4, 23), (5, 48), (5, 54), (6, 81), (6, 67), (6, 95), (7, 65), (7, 82), (8, 88), (8, 39), (9, 84), (9, 94), (10, 47), (10, 85), (10, 37), (11, 77), (11, 87), (12, 96), (13, 18), (13, 53), (14, 57), (15, 40), (16, 65), (16, 61), (16, 94), (17, 68), (17, 22), (18, 62), (19, 96), (19, 83), (19, 63), (20, 25), (20, 28), (20, 60), (21, 97), (21, 30), (21, 71), (23, 47), (24, 31), (24, 95), (25, 78), (26, 96), (26, 54), (27, 44), (27, 78), (28, 51), (28, 29), (29, 79), (30, 45), (31, 35), (32, 36), (33, 59), (34, 89), (36, 56), (37, 83), (38, 98), (38, 69), (39, 42), (40, 97), (41, 58), (41, 50), (42, 46), (43, 75), (45, 86), (46, 75), (49, 50), (49, 60), (52, 74), (52, 62), (55, 56), (55, 61), (56, 73), (57, 74), (58, 98), (59, 90), (62, 69), (64, 80), (64, 98), (66, 80), (67, 90), (68, 81), (70, 82), (71, 88), (72, 81), (76, 79), (84, 89), (85, 93), (86, 93), (87, 96), (92, 93), (95, 99), (16, 81), (33, 98), (40, 92), (43, 48), (56, 62), (22, 24), (35, 99), (6, 70), (20, 21), (93, 95), (10, 12), (14, 15), (18, 19), (77, 72), (2, 66), (91, 28), (73, 76), (51, 53)



(d) Теперь надо реализовать **эйлеров обход графа**.
Напомним, что **Эйлеров обход в графе** – путь, который проходит по всем ребрам ровно один раз. Напомним алгоритм обхода:
   i. Запускаем алгоритм из вершины с нечетной степенью (таковых не больше двух).

ii. Действуем аналогично поиску в ширину, только помечаем не пройденные вершины, а ребра: Начинаем со старотовой вершины s и добавляем на каждом шаге не открытое ребо, исходящее из текущей вершины, которые мы накапливаем в стеке $CurrenPath$.

iii. Когда для текущей вершины открыты все инцидентные ребра, записываем вершины из $CurrentPath$ в $EulerPath$ до тех пор, пока не встретится вершина, из которой исходят неоткрытые ребра.

iv. Обход продолжается до тех пор, пока все ребра не будут посещены.

Реализация:

```
def EulerPath(G):
    ans=[]
    for u in G.nodes():
        if nx.deg(u) % 2 == 1:
            v = u
            break
    CurrentPath.append(v)
    while not (len(CurrentPath)==0):
        q=CurrentPath.pop()
        for r in G.nodes():
            if (q,r) in G.edges():
                CurrentPath.append(r)
                break
            if q == CurrentPath[len(CurrentPath)-1]
                CurrentPath.pop()
                ans.append(q)
    return ans
```

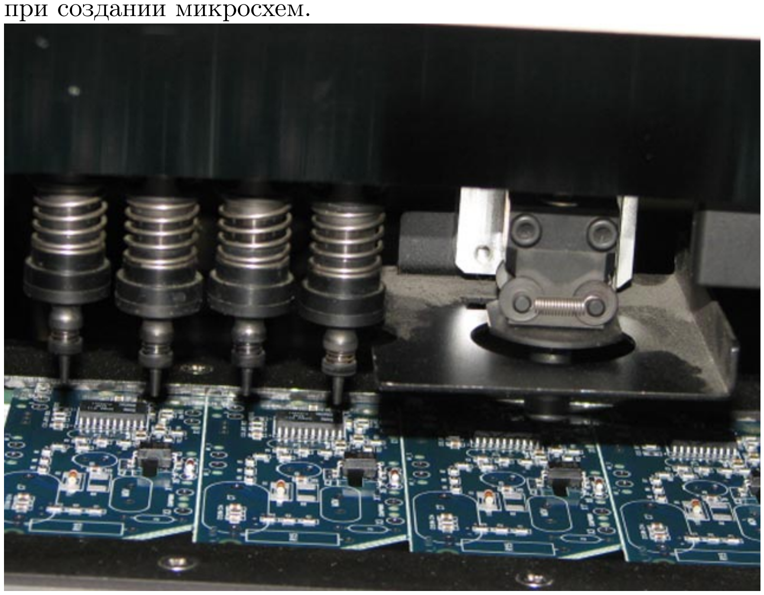
Запустим эйлеров обход на нашем графе и предпосчитаем общую длину обхода:

```
EP = EulerPath(_G)
sum_distance=0
for i in len(EP):
    sum_distance += _G[i][(i+1)%len(EP)]['weight']
```

Итого, получили суммарную протяженность пути – 14433km. Как мы видим, это меньше, чем 15587. Следовательно, мы подтвердили на практике, что наш алгоритм действительно работает корректно на примере городов Катара.

5 Заключение и результаты

Задачи, связанные с TSP, сегодня не имеют оптимального решения. Ученые годами ищут новые алгоритмы, позволяющие уточнить приближение на 1-2.Последнее значимое достижение произошло в 2006 году. Решение этой задачи очень важно – оно носит не просто теоретический характер, но и имеет важное прикладное значение. Например, при создании микросхем.



[7].

6 Литература

(a) https://habrahabr.ru/post/125898/

(b) http://e-maxx.ru/index.php

(c) https://arxiv.org/abs/1310.1896

(d) J. Edmonds. Path, trees, and flowers. Canadian J. Math., 17:449–467, 1965.

(e) Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

(f) http://www.mscs.dal.ca/ janssen/4115/presentations/Poppy.pdf

(g) http://mathworld.wolfram.com/EulerianCycle.html

[7]https://en.wikipedia.org/wiki/Travelling$_{s}alesman_{p}roblem$