FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
«NATIONAL RESEARCH UNIVERSITY
«HIGHER SCHOOL OF ECONOMICS»

Faculty of Computer Sciences

A Heuristic Algorithm for the Traveling Salesman Problem, based on Nicos Christofides' lower bound algorithm

Educational program "Applied Mathematics and Informatics"
Educational standard ***01.03.02*** "Applied Mathematics and Informatics"

Reviewer — Dr. Boris Goldengorin
Professor of Operations Research,
University of Baltimore

Reporters — Nikiforov Alexey Vladimirovich
3rd year student, Higher School of Economics

Popov Ilya Ivanovich
3rd year student, Higher School of Economics

MOSCOW, 2020

Contents

Introduction and problem formulation

The traveling salesman problem (TSP) is a well known combinatorial optimization problem. Basically, to solve a traveling salesman problem means to find the shortest possible route that visits each city and returns to the origin city for a given list of cities and the distances between each pair of cities. Optimal solutions to small instances can be found in reasonable time by linear programming. But TSP is also known to be a NP-hard problem, which leads to the fact that solving larger instances with guaranteed optimality is very time-consuming.

Putting the optimality aside, there are a bunch of heuristic methods to find a near-optimal solution. Speaking about the traveling salesman problem, there are a lot of methods we can use: the *greedy algorithm*, the *Nearest Neighbor* algorithm, insertion heuristics (*nearest addition, nearest insertion, farthest insertion, cheapest insertion, arbitrary insertion*), tour improvement heuristics (*k-opt, Lin-Kernighan algorithm*), methods combining heuristics above (*Double MST, Christofides algorithm*) and others. [1]

There is also another very interesting Christofides algorithm that provides us with a lower bound to the solution of TSP which is only 4.7 percent below the optimum and requires 9 percent greater computation time than the time required to solve an equivalent assignment problem. [2] The general idea behind this algorithm is to solve an assignment problem for the source adjacency matrix, replace each found subtour with a single node and continue solving smaller dimension assignment problems until the whole graph is compressed into a single node. [2]

Christofides lower bound algorithm, $O(1.143kN^3)$:

Step 1. Set a matrix M equal to the initial distance matrix $[d_{ij}]$, and set $L=0$

Step 2. If the matrix M satisfies the triangularity condition of metric space, go to *step 3*; if not, compress M until $m_{ij} \leq m_{ik} + m_{kj}$ for any k;

Step 3. Solve the assignment problem using matrix *M* and let *V(AP)* be the value of this solution. Set *L=L+V(AP)*.

Step 4. Contract the matrix *M* by replacing subtours (formed as a result of the solution to the assignment problem at *step 3*) by single nodes.

Step 5. If the contracted matrix *M* is a 1 by 1 matrix, go to *step 6*; otherwise, return to *step 2*.

Step 6. End. The value of *L* is a lower bound to the value of the TSP.

The goal of our work is to modify the algorithm above and come up with yet another heuristic algorithm for solving the traveling salesman problem. We are going to describe a recursive algorithm of finding near-optimal solution of TSP and run some tests using Python implementation of the algorithm.

Description of the algorithm and code

The algorithm we made takes distance matrix *M* as input and consists of these steps:

Step 1. Let *T = rec(M)*, where T – near-optimal tour from our recurrent algorithm.

Step 2. Use 2-opt tour optimization algorithm for the tour *T* from *step 1* and distance matrix *M*.

Function rec(M):

Step 1. Check if matrix *M* satisfies the triangularity condition of metric space using Floyd-Warshall algorithm. If not, compress *M* until $m_{ij} \leq m_{ik} + m_{kj}$ for any *k*;

Step 2. Solve the assignment problem using matrix *M*. The output of this step will be a dictionary of pairs *L*, each pair describing edge.

Step 3. Construct cycles from *L*. The output of this step will be a list of cycles *P*.

Step 4. If *P* contains only one cycle than return *P*. Otherwise – build matrix *M'* from *M* by replacing subtours (formed as a result of the solution to the assignment problem at *step 3*) by single nodes and let *S* = *rec(M')*, where *S* – sequence of vertices from the step after the current one.

Step 5. Reconstruct the cycle with sequence *S* with *P* from *step 3*, forming sequence of nodes Q. For each element *s* of *S* we add elements of *P[s]* to *Q*.

Step 6. Return *Q*

end function

Demonstration of the algorithm

Here is an example how the algorithm works for solving symmetrical TSP problem with known optimal tour. Consider the 10 city TSP whose distance matrix is given in Table 1.

	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
a1	∞	32	41	22	20	57	54	32	22	45
a2	32	∞	22	30	42	51	61	20	54	31
a3	41	22	∞	63	41	30	45	10	60	36
a4	22	30	63	∞	36	78	72	54	20	66
a5	20	42	41	36	∞	45	36	32	22	28
a6	57	51	30	78	45	∞	22	32	67	20
a7	54	61	45	72	36	22	∞	41	57	10
a8	32	20	10	54	32	32	41	∞	50	32
a9	22	54	60	20	22	67	57	50	∞	50
a10	45	31	36	64	28	20	10	32	50	∞

Table 1. Initial distance matrix of the example

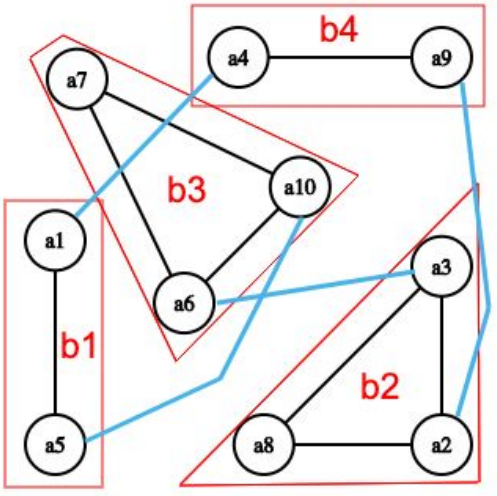

Figure 1. The first solution of the AP for the example

Green cells in Table 1 represent the solution of AP. On Figure 1 group of nodes in red boxes are the cycles formed from the solution of assignment problem, and blue edges between them are the shortest edges connecting two cycles. Let P_1 = [[a_1, a_5],

$[a_2, a_3, a_8]$, $[a_6, a_7, a_{10}]$, $[a_4, a_9]]$. Now we construct the new distance matrix M_2 (see Table 2) and solve another assignment problem.

	b1	b2	b3	b4
b1	∞	32	22	28
b2	32	∞	30	30
b3	22	30	∞	50
b4	28	30	50	∞

Table 2. Compressed distance matrix M_2

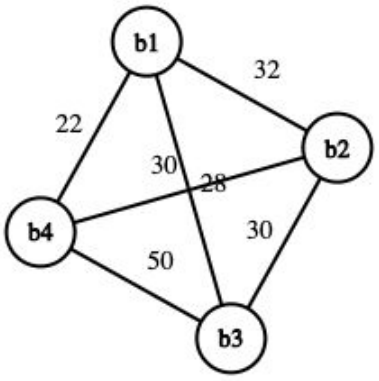

Figure 2. A graph described by distance matrix M_2

After solving an AP from matrix given in Table 2, we let $P_2 = [[b_1, b_3], [b_2, b_4]]$ and contract the matrix M_2 to the state given on Figure 3. As there are two cycles in the solution of AP – we construct the new distance matrix M_3 and solve another assignment problem.

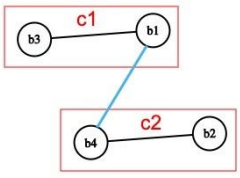

Figure 3. The second solution of the AP for the example

	c1	c2
c1	∞	28
c2	28	∞

Table 3. Compressed distance matrix M_3

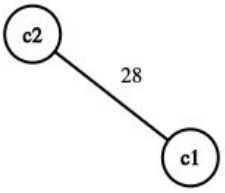

Figure 4. A graph described by distance matrix M_3

It is obvious that in this case the solution of an AP will consist of only one cycle, so we have no need to solve it. Recursion finishes, and now we can calculate the results for all the previous steps.

$P_3 = [c_1, c_2] \rightarrow$

$S_2 = [c_1, c_2] \rightarrow$

$Q_2 = [b_3, b_1, b_4, b_2] \rightarrow$

$S_1 = [b_3, b_1, b_4, b_2] \rightarrow$

$Q_1 = [a_6, a_7, a_{10}, a_5, a_1, a_4, a_9, a_2, a_8, a_3]$ - the tour found by the algorithm. It's length equals 236. After applying 2-opt tour optimization algorithm we get a tour $[a_3, a_8, a_2, a_4, a_9, a_1, a_5, a_{10}, a_7, a_6]$. The length of this tour is equal to the length of the optimal tour, which is 214.

Computational complexity and code implementation

Here's pseudo code of our algorithm once again:

```
function rec(matrix):
    matrix = floyd_warshall_algorithm(matrix)
    assignments = hungarian_algorithm(matrix)
    cycles = []
    repeat until all nodes were used
        new_cycle = []
        i = unused node
        while i unused
            i used
            new_cycle.append(i)
            i = assignments[i]
        cycles.append(new_cycle)
    if only 1 cycle then return cycles[0]
    for i, j in cycles (i != j)
        new_matrix[i][j] = shortest edge between i and j in matrix
    new_matrix diagonal elements = infinity
    external_cycle = rec(new_matrix)
    perfect_cycle = []
    for i in external_cycle
        (a, b) = shortest edge between cycle i and i+1 in matrix (a in i, b in
i+1)
        add i in perfect_cycle in such order to end in a
    return perfect_cycle
end function

2-opt(rec(matrix))
```

As the first step of our recursive algorithm, we use Floyd-Warshall algorithm to check if input matrix satisfies the triangularity condition of metric space and fix if it does not.

Floyd-Warshall Algorithm, $O(n^3)$:

```
function floyd_warshall_algorithm(matrix):
  for i, j, k in range(n):
    if (matrix[i][j] < infinity):
  matrix_new[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
  return matrix_new
end function
```

Then we need to solve an assignment problem with Hungarian Algorithm. Here we switch to alternative formulation of the assignment problem: find minimal perfect matching in bipartite graph with n nodes in each part [4].

Let *potential* be arrays $u[1…n]$, $v[1…n]$, such that $u[i]+v[j] \leq matrix[i][j]$. If $u[i]+v[j] = matrix[i][j]$, then the edge (i, j) is called a hard edge. We are trying to find matching only among hard edges. At the beginning $u[i] = v[i] = 0$ and matching M is empty. Let Z_1 be edges in left part included in matching, Z_2 – edges in right part included in matching and $minv[j] = min(matrix[i][j] – u[i] – v[j])$ *(among all i in* Z_1*)*.

Add to considered part one row from matrix. Until we find increasing chain starting in this row we recalculate potential:

$delta = min(minv[j])$ *(for all j in* Z_2*)*

for all i in Z_1: $u[i] = u[i] + delta$

for all j in Z_2: $v[j] = v[j] – delta$

$minv[j] = min(matrix[i][j] – u[i] – v[j])$ *(among all i in* Z_1*)*

After that for each hard edge added mark it’s right end as achievable and continue bypass from it in case it’s left end is achievable. When we find increasing chain we remember matching and add next row. The computational complexity of this algorithm is $O(n^3)$ [4].

With every call of our recursive function each dimension of input matrix becomes at least two times smaller, which gives us the total computational complexity of our recurrent algorithm of $O(n^3 \log_2(n))$.

And in the end, we apply 2-opt algorithm [1] on tour we received from our recursive algorithm.

2-opt Tout Optimization Algorithm, $O(n^2)$:

```
Function Swap(route, i, k):
    new_route.append(route[0] to route[i-1])
    new_route.append(reversed route[i] to route[k])
    new_route.append(route[k+1] to route[end])
End function

Function 2opt(tour, matrix):
     repeat until no improvement made
         for i, j in nodes eligible to swap do
             new_route = Swap(route, i, j)
             if RouteDistance(new_route) < RouteDistance(route):
                 route = new_route
             end if
         end for
     end
     return route
End function
```

See Appendix 1 for the full code used in experiments.

Tests

To test our algorithm, we did some tests on well known problems from TSPLIB [3]. See results in table 4.

Name	Number of cities	Solution to TSP	Solution by recursive algorithm	% difference	Solution by recursive + 2-opt	% difference
FIVE	5	19	25	31.50	21	10.53
P 01	15	291	358	23.02	305	4.81

GR 17	17	2085	2456	17.79	2191	5.08
FRI 26	26	937	1021	8.96	1021	8.96
DANTZIG 42	42	699	831	18.88	780	11.59
ATT 48	48	33523	43984	31.20	36308	8.30
XQF 131	131	564	800	41.84	680	20.57
XQG 237	237	1019	1390	36.40	1203	18.06
PBM 436	436	1443	2034	40.95	1635	13.31
XQL 662	662	2513	3764	49.78	2912	15.88
DKG 813	813	3199	4278	33.72	3565	11.44
FRA 1488	1488	4264	6106	43.19	4787	12.27
XSC 6880	6880	21537	30642	42.23	23665	9.88
			Average	32.27	Average	11.59

Table 4. Tests' results.

As we can see from Table 4, the recursive algorithm we came up with seems to be not the most accurate (tours are 32.27% longer than the optimal in average), but 2-opt helps us to get a better tour. In total, the whole pipeline with recursive algorithm and 2-opt optimization provides us with the tour which is ~12% bigger than optimal in average.

Conclusion

We presented a heuristic algorithm for solving TSP problem with $O(n^3\log_2(n))$ time complexity. Unfortunately for us, the algorithm turned out to be pretty slow comparing to other heuristic methods (for example, for Double Minimum Spanning Tree algorithm computational complexity is $O(n^2log_2(n))$ [1], for the Christofides Algorithm - $O(n^3)$ [1]) and also not very accurate. However, we do believe that there is a space for improvement, and there are lots of interesting things for us to discover in this field of knowledge.

References

[1] Christian Nilsson, Heuristics for the Traveling Salesman Problem, Jan 2003 – https://www.researchgate.net/publication/228906083_Heuristics_for_the_Traveling_Salesman_Problem

[2] Nicos Christofides Bounds for the Travelling-Salesman Problem // Operations Research, Vol. 20, No. 5 (Sep. - Oct., 1972), p. 1044-1056

[3] Gerhard Reinelt, 1991. "TSPLIB—A Traveling Salesman Problem Library," INFORMS Journal on Computing, INFORMS, vol. 3(4), pages 376-384, November.

[4] Ahuja, Ravindra & Magnanti, Thomas & Orlin, James. (1993). Network Flows.

[5] Kuhn, H.W. (1955), The Hungarian method for the assignment problem. Naval Research Logistics, 2: 83-97. doi:10.1002/nav.3800020109

[6] Munkres, J. (1957) Algorithms for the Assignment and Transportation Problems. Journal of the Society for Industrial and Applied Mathematics, 5, 32-38.

Appendix 1 - Full program code

```
import copy
inf = 999999

def floyd_warshall_algorithm(matrix):
    n = len(matrix)
    matrix_new = copy.deepcopy(matrix)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                if (matrix_new[i][j] < inf):
                    matrix_new[i][j] = min(matrix_new[i][j],
matrix_new[i][k]+matrix_new[k][j])
    return matrix_new

def hungarian_algorithm(matrix):
    n = len(matrix)
    a = [[inf]*(n+1)]
    for i in range(n):
        a = a + [[inf] + matrix[i]]
    u = [0] * (n+1)
    v = [0] * (n+1)
    p = [0] * (n+1)
    way = [0] * (n+1)
    for i in range(1, n+1):
        p[0] = i
        j_0 = 0
        minv = [inf] * (n+1)
        used = [0] * (n+1)
        while(True):
            used[j_0] = 1
            i_0 = p[j_0]
            delta = inf
            for j in range(1, n+1):
                if (used[j] == 0):
                    cur = a[i_0][j] - u[i_0] - v[j]
                    if (cur < minv[j]):
                        minv[j] = cur
                        way[j] = j_0
                    if (minv[j] < delta):
                        delta = minv[j]
                        j_1 = j
            for j in range(n+1):
                if (used[j] != 0):
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    minv[j] -= delta
            j_0 = j_1
```

```
            if (p[j_0] == 0):
                break
        while(True):
            j_1 = way[j_0]
            p[j_0] = p[j_1]
            j_0 = j_1
            if (j_0 == 0):
                break
    ans = [(-1, -1)] * (n)
    for j in range(1, n+1):
        ans[p[j]-1] = (p[j]-1, j-1)
    return ans

def Christofides_recurrent(input_matrix, flag_skip_floyd_warshall = False):
    if (flag_skip_floyd_warshall == True):
        matrix = floyd_warshall_algorithm(input_matrix)
    n = len(matrix)
    idx = hungarian_algorithm(matrix)   #Решаем задачу о
назначениях
    way = dict(idx)
    #восстанавливаем циклы
    flags = [0]*n
    cycles = [[]]
    while sum(flags) < n:
        one_cycle = []
        i = flags.index(0)
        while flags[i] == 0:
            flags[i] = 1
            one_cycle.append(i)
            i = way[i]
        cycles.append(one_cycle)
    cycles = cycles[1:]
    if len(cycles) == 1:
        return cycles[0]
    #Объединяем циклы в точку и повторяем
    new_matrix = [[inf] * len(cycles) for i in range(len(cycles))]
    shortcuts = [[(-1, -1)] * len(cycles) for i in range(len(cycles))]
    for i in range(len(cycles)):
        for j in range(len(cycles)):
            if(i != j):
                for p in range(len(cycles[i])):
                    for q in range(len(cycles[j])):
                        x = cycles[i][p]
                        y = cycles[j][q]
                        if(matrix[x][y] < new_matrix[i][j]):
                            new_matrix[i][j] = matrix[x][y]
                            shortcuts[i][j] = (x, y)
    external_cycle = Christofides_recurrent(new_matrix)
    #Объединяем циклы, выписывая порядки обхода
малых циклов по порядку обхода большого
    external_cycle = external_cycle + external_cycle[0:2]
    perfect_cycle = []
```

```
    for i in range(1, len(external_cycle)-1):
        sequence_in = shortcuts[external_cycle[i-1]][external_cycle[i]][1]
        next_in = shortcuts[external_cycle[i]][external_cycle[i+1]][1]
        this_cycle = copy.deepcopy(cycles[external_cycle[i]])
        this_cycle = this_cycle + this_cycle[0:2]
        j = this_cycle[1:-1].index(sequence_in)+1
        direct_out = this_cycle[j-1]
        reverse_out = this_cycle[j+1]
        if(matrix[direct_out][next_in]-matrix[direct_out][sequence_in] <=
matrix[reverse_out][next_in]-matrix[reverse_out][sequence_in]):
            perfect_cycle = perfect_cycle + this_cycle[j:-1]
            perfect_cycle = perfect_cycle + this_cycle[1:j]
        else:
            part_1 = this_cycle[1:j+1]
            part_1.reverse()
            part_2 = this_cycle[j+1:-1]
            part_2.reverse()
            perfect_cycle = perfect_cycle + part_1 + part_2
    return perfect_cycle

def algorithm_2opt(matrix, input_sequence):
    sequence = copy.deepcopy(input_sequence)
    flag_is_improved = 1
    while flag_is_improved == 1:
        flag_is_improved = 0
        for i in range(len(sequence)-2):
            for j in range(i+2,min(len(sequence)+i-1, len(sequence))):
                if (matrix[sequence[i]][sequence[i+1]] +
matrix[sequence[j]][sequence[(j+1)%len(sequence)]] > matrix[sequence[i]][sequence[j]]
+ matrix[sequence[i+1]][sequence[(j+1)%len(sequence)]]):
                    rev_part = sequence[i+1:j+1]
                    rev_part.reverse()
                    sequence = sequence[:i+1] + rev_part + sequence[j+1:]
                    flag_is_improved = 1
    return sequence
```