A HEURISTIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM, BASED ON NICOS CHRISTOFIDES' LOWER BOUND ALGORITHM

PROBLEM FORMULATION & EXAMPLES,

ILYA POPOV, NATIONAL RESEARCH UNIVERSITY «HIGHER SCHOOL OF ECONOMICS», BAMI-174

CONTENTS

PROBLEM FORMULATION, COMPLEXITY

Given:

- n cities
- Distances or costs between pair or a metric on the space

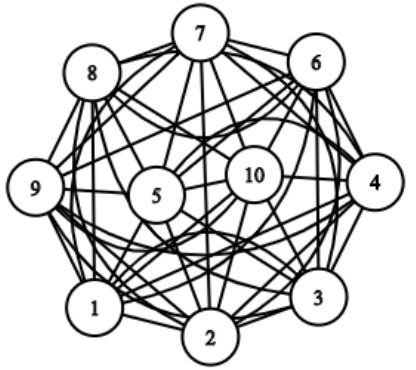
Find:

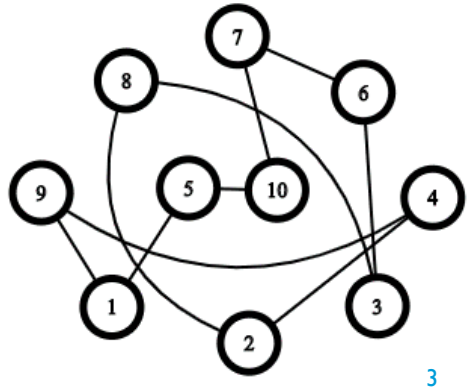
- Permutation such that tour length is minimized

Problem:

- There are n! different permutations
- The TSP belongs to class of NP-complete problems

	1	2	3	4	5	6	7	8	9	10
1	∞	32	41	22	20	57	54	32	22	45
2	32	∞	22	30	42	51	61	20	54	31
3	41	22	∞	63	41	30	45	10	60	36
4	22	30	63	∞	36	78	72	54	20	66
5	20	42	41	36	∞	45	36	32	22	28
6	57	51	30	78	45	∞	22	32	67	20
7	54	61	45	72	36	22	∞	41	57	10
8	32	20	10	54	32	32	41	∞	50	32
9	22	54	60	20	22	67	57	50	∞	50
10	45	31	36	64	28	20	10	32	50	∞

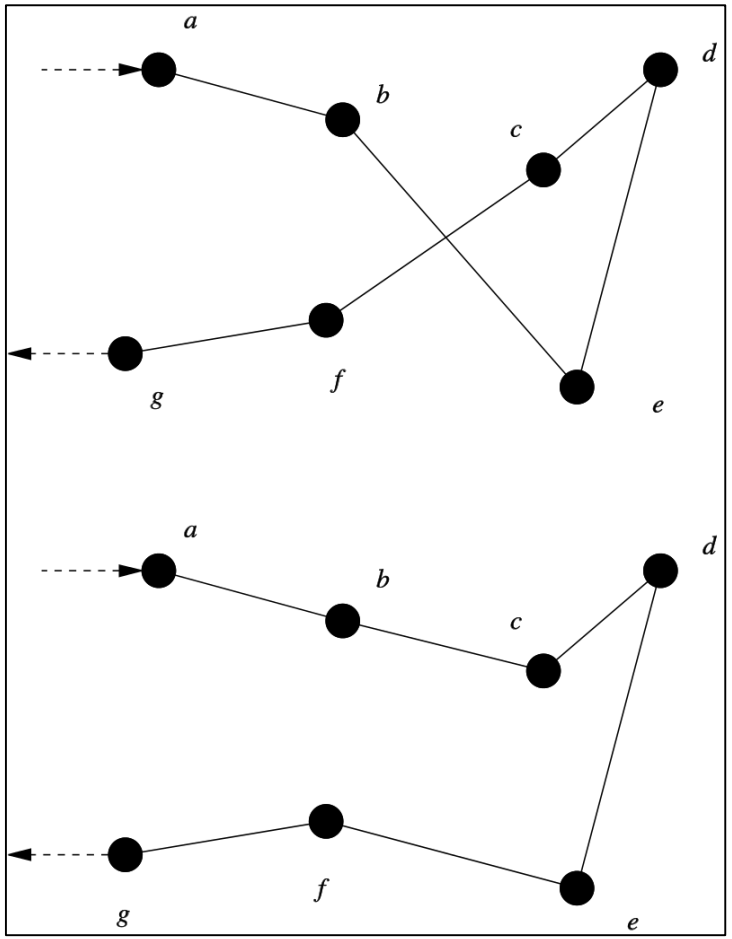




HEURISTIC METHODS OF SOLVING TSP

There is a bunch of algorithms offering comparably fast running time and still yielding near-optimal solutions:

- Tour Construction Algorithms:
  - Nearest Neighbor, $O(n^2)$
  - Greedy Algorithm, $O(n^2 \log_2(n))$
  - Insertion heuristics:
    - Nearest Insertion, $O(n^2)$
    - Convex Hull, $O(n^2 \log_2(n))$
  - Double MST, $O(n^2 \log_2(n))$
  - Christofides Algorithm, $O(n^3)$
- Tour Improvement Algorithms:
  - k-opt Algorithms
  - Lin-Kernighan Algorithm
  - Taboo-Search Algorithm
  - Simulated Annealing
  - Genetic Algorithms
- Branch & Bound Algorithm
- Ant Colony Optimization

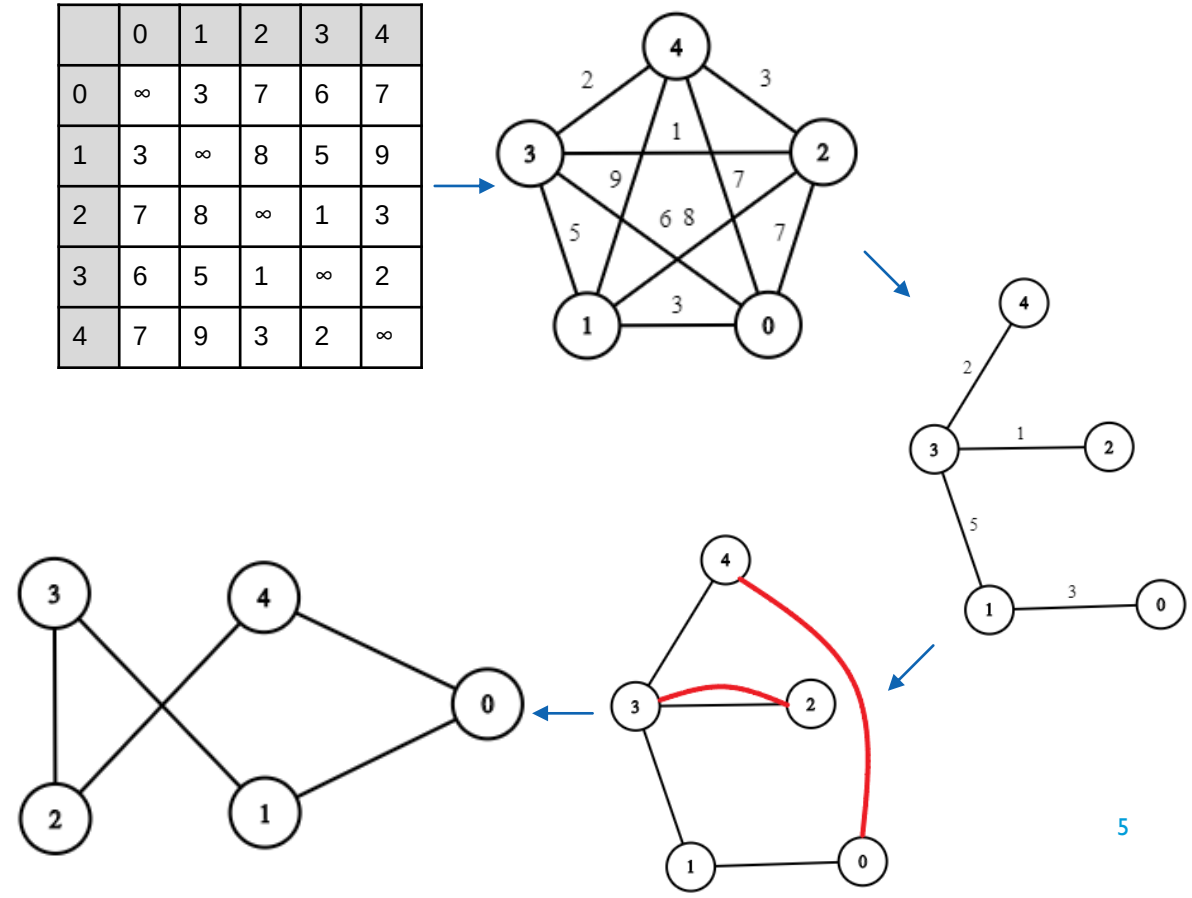


CLASSIC CHRISTOFIDES ALGORITHM

Christofides Algorithm, worst-case ratio 3/2, $O(n^3)$:

- Build a minimal spanning tree (MST) from the set of all cities
- Create a minimum-weight matching (MWM) on the set of nodes having an odd degree. Add the MST together with the MWM
- Create an Euler cycle from the combined graph, and traverse it taking shortcuts to avoid visited nodes

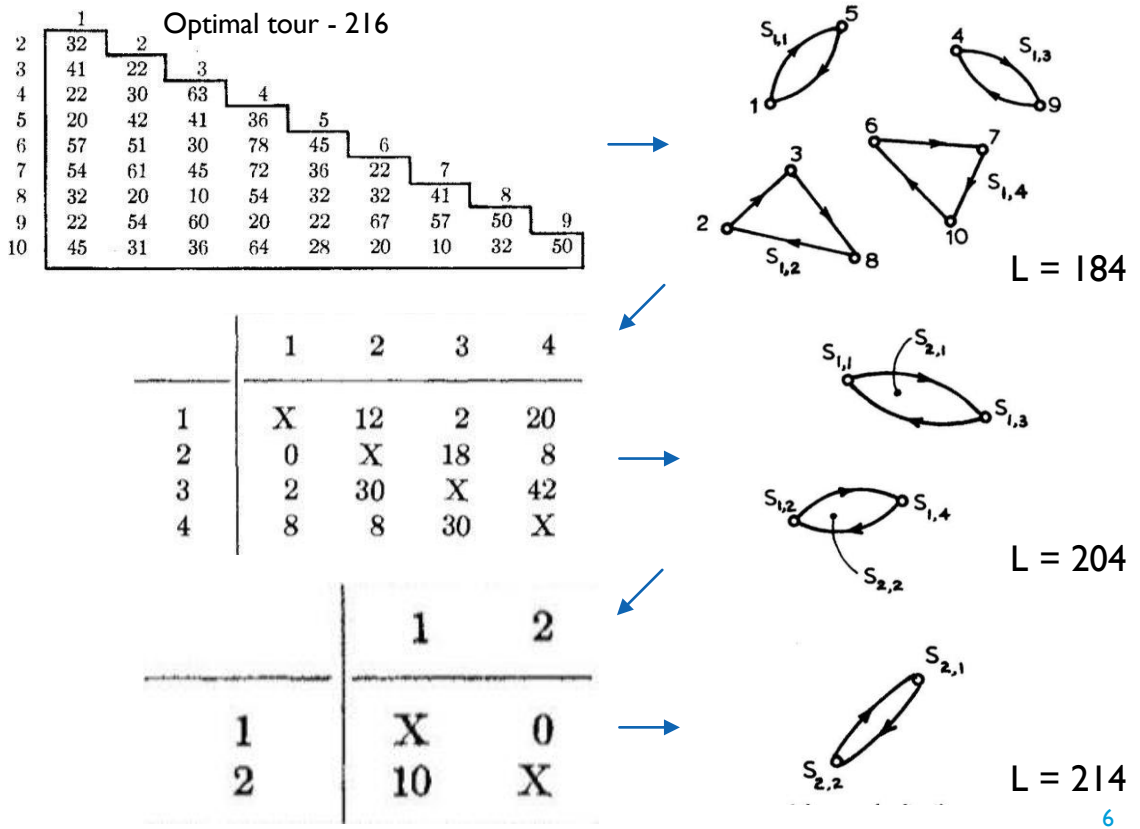
	0	1	2	3	4
0	∞	3	7	6	7
1	3	∞	8	5	9
2	7	8	∞	1	3
3	6	5	1	∞	2
4	7	9	3	2	∞



CHRISTOFIDES LOWER BOUND ALGORITHM

Christofides lower bound Algorithm, $O(1.143k \cdot n^3)$:

- Set a matrix *M* equal to the initial matrix $[d_{ij}]$ and set *L=0*
- If *M* satisfies the triangularity condition of metric space – go to *step 3*, otherwise – compress *M* until $\forall\ k\ m_{ij} \leq m_{ik} + m_{kj}$
- Solve the assignment problem using matrix *M* and let V(AP) be the value of this solution. *L = L + V(AP)*
- Contract the matrix *M* by replacing sub tours by single nodes
- If contracted matrix *M* is a 1 by 1 matrix – go to *step 6*, otherwise – go to *step 2*
- The end. The value of *L* is a lower bound of the TSP.



THE GOAL OF OUR WORK

The goal of our work was to:

- Come up with heuristic algorithm for solving the Travelling Salesman Problem based on Nicos Christofides' Lower Bound Algorithm
- Write a program to find a solution of the Travelling Salesman Problem using the algorithm we came up with
- Test our algorithm on TSPs with different number of cities and see if we can find optimal tour or find a tour which is better than known optimal tour

THE NEW ALGORITHM

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

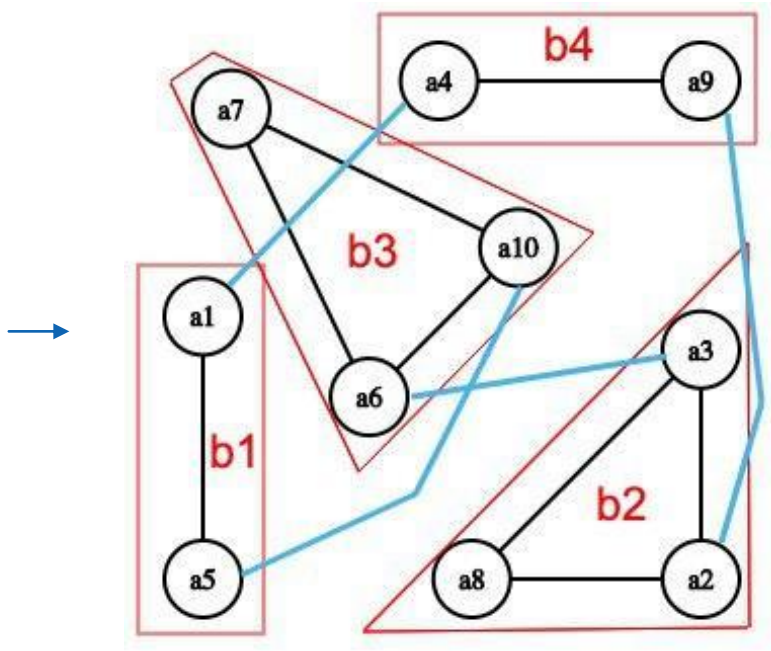
The algorithm we made takes distance matrix *M* as input and consists of these steps:

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*.

EXAMPLE

	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	∞

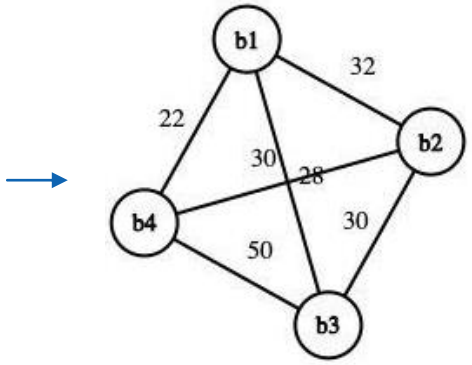


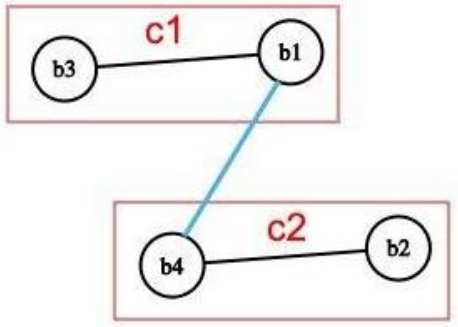
- Given matrix satisfies triangularity condition
- Solve AP for a given matrix (cells in AP's solution are colored in green)
- Get contracted matrix M_2 by replacing subtours by single nodes
  - Replace cycles with nodes
  - Find shortest edges connecting cycles – these will be the edges of the new graph

$P_1 = [\,[a_1, a_5], [a_2, a_3, a_8], [a_6, a_7, a_{10}], [a_4, a_9]]$

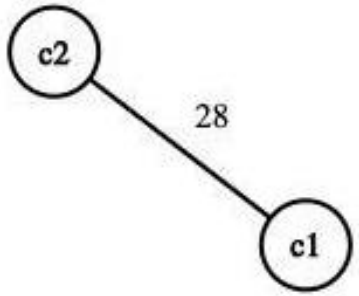
EXAMPLE

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





	c1	c2
c1	∞	28
c2	28	∞



- Do the same steps for matrix M_2
- AP solution for M_2 gives us 2 cycles, so we find contracted matrix M_3 and solve AP for it.

$P_2 = [[b_1, b_3], [b_2, b_4]]$

- AP solution for M_2 gives only 1 cycle – 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 is 236.

After applying 2-opt we get a tour $[a_3, a_8, a_2, a_4, a_9, a_1, a_5, a_{10}, a_7, a_6]$, it's length equals optimal, which is 214.

A HEURISTIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM, BASED ON NICOS CHRISTOFIDES' LOWER BOUND ALGORITHM

ALGORITHM & PYTHON PROGRAM,
ALEXEY NIKIFOROV, NATIONAL RESEARCH UNIVERSITY «HIGHER SCHOOL OF ECONOMICS», BAMI-174

THE NEW ALGORITHM

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

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*.

For out program we used pure Python 3.6

CODE AND COMPUTATIONAL COMPLEXITY

```
1.Christofides_recurrent(matrix):
2.    matrix = floyd_warshall_algorithm(matrix)
3.    assignments = hungarian_algorithm(matrix)
4.    cycles = []
5.    repeat until all nodes were used
6.        new_cycle = []
7.        i = unused node
8.        while i unused
9.            i used
10.            new_cycle.append(i)
11.            i = assignments[i]
12.        cycles.append(new_cycle)
13.    if only 1 cycle then return cycles[0]
14.    for i, j in cycles (i != j)
15.        new_matrix[i][j] = shortest edge between i and j in matrix
16.    new_matrix diagonal elements = infinity
17.    external_cycle = Christofides_recurrent(new_matrix)
18.    perfect_cycle = []
19.    for i in external_cycle
20.        (a, b) = shortest edge between cycle i and i+1 in matrix (a in i, b in i+1)
21.        add i in perfect_cycle in such order to end in a
22.    return perfect_cycle

23.2-opt(Christofides_recurrent(matrix))
```

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
```

CODE AND COMPUTATIONAL COMPLEXITY

```
1.Christofides_recurrent(matrix):
2.    matrix = floyd_warshall_algorithm(matrix) O(n^3)
3.    assignments = hungarian_algorithm(matrix)
4.    cycles = []
5.    repeat until all nodes were used
6.        new_cycle = []
7.        i = unused node
8.        while i unused
9.            i used
10.            new_cycle.append(i)
11.            i = assignments[i]
12.        cycles.append(new_cycle)
13.    if only 1 cycle then return cycles[0]
14.    for i, j in cycles (i != j)
15.        new_matrix[i][j] = shortest edge between i and j in matrix
16.    new_matrix diagonal elements = infinity
17.    external_cycle = Christofides_recurrent(new_matrix)
18.    perfect_cycle = []
19.    for i in external_cycle
20.        (a, b) = shortest edge between cycle i and i+1 in matrix (a in i, b in i+1)
21.        add i in perfect_cycle in such order to end in a
22.    return perfect_cycle

23.2-opt(Christofides_recurrent(matrix))
```

Hungarian Algorithm, $O(n^3)$:

Here we switch to alternative formulation of assignment problem: find minimal perfect matching in bipartite graph with n nodes in each part.

Potential - arrays *u[1..n]*, *v[1..n]*, such that *u[i]+v[j] <= matrix[i][j]*

If *u[i]+v[j] = matrix[i][j]*, then call *(i, j)* 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.

Z_1 – edges in left part included in matching
Z_2 – edges in right part included in matching
minv[j] = min(matrix[i][j] – u[i] – v[j]) (for all *i* in Z_1)

Add to considered part one row from matrix. Until no increasing chain starting in this row 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 Z2 v[j] = v[j] – delta
recalculate minv

After that for each added hard edge if its left end was achievable mark his right end as achievable and continue bypass from them.

CODE AND COMPUTATIONAL COMPLEXITY

```
1.Christofides_recurrent(matrix):
2.    matrix = floyd_warshall_algorithm(matrix) O(n^3)
3.    assignments = hungarian_algorithm(matrix) O(n^3)
4.    cycles = []
5.    repeat until all nodes were used
6.        new_cycle = []
7.        i = unused node
8.        while i unused
9.            i used
10.            new_cycle.append(i)
11.            i = assignments[i]
12.        cycles.append(new_cycle)
13.    if only 1 cycle then return cycles[0]
14.    for i, j in cycles (i != j)
15.        new_matrix[i][j] = shortest edge between i and j in matrix
16.    new_matrix diagonal elements = infinity
17.    external_cycle = Christofides_recurrent(new_matrix)
18.    perfect_cycle = []
19.    for i in external_cycle
20.        (a, b) = shortest edge between cycle i and i+1 in matrix (a in i, b in i+1)
21.        add i in perfect_cycle in such order to end in a
22.    return perfect_cycle

23.2-opt(Christofides_recurrent(matrix))
```

On each iteration of our algorithm each dimension of given matrix becomes at least two times smaller.

Considering all written above, the total computational complexity of our recurrent algorithm is $O(n^3 \log_2(n))$

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

```
1.Swap(route, i, k):
2.    new_route.append(route[0] to route[i-1])
3.    new_route.append(reversed route[i] to route[k])
4.    new_route.append(route[k+1] to route[end])
5.
6.repeat until no improvement made
7.    for i,j in nodes eligible to swap do
8.        new_route = Swap(route, i, j)
9.        if RouteDistance(new_route) < RouteDistance(route):
10.            route = new_route
11.        end if
12.    end for
13.end
14.return route
```

TESTS

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

- To test our algorithm, we did some tests on well known problems from TSPLIB.
- The algorithm itself seems to be not the most accurate
- Fortunately, 2-opt tour optimization saves the day

REFERENCES

- Christian Nilsson, Heuristics for the Traveling Salesman Problem, Jan 2003 – https://www.researchgate.net/publication/228906083_Heuristics_for_the_Traveling_Salesman_Problem
- Nicos Christofides Bounds for the Travelling-Salesman Problem // Operations Research, Vol. 20, No. 5 (Sep. - Oct., 1972), p. 1044-1056
- Gerhard Reinelt, 1991. "TSPLIB—A Traveling Salesman Problem Library," INFORMS Journal on Computing, INFORMS, vol. 3(4), pages 376-384, November.
- Ahuja, Ravindra & Magnanti, Thomas & Orlin, James. (1993). Network Flows.
- Kuhn, H.W. (1955), The Hungarian method for the assignment problem. Naval Research Logistics, 2: 83-97. doi:10.1002/nav.3800020109
- Munkres, J. (1957) Algorithms for the Assignment and Transportation Problems. Journal of the Society for Industrial and Applied Mathematics, 5, 32-38.