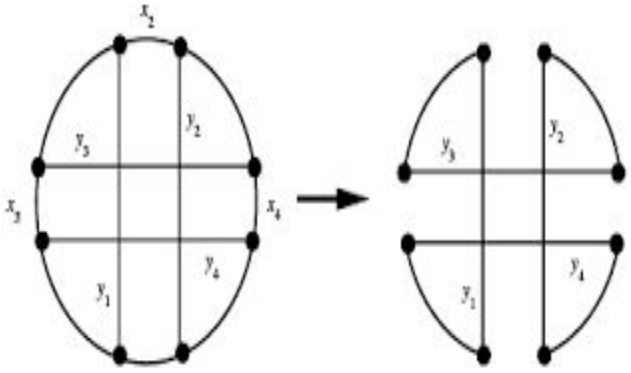
Algorithms Coursework

Orges Skura(s1813106)

Part C : Lin-Kerninghan Algorithm:

For part C, I chose to implement Lin-Kernighan heuristic. Lin-Kernighan Algorithm relies on the concept of k-opt algorithm.A tour is said to be k-optimal (or simply k-opt) if it is impossible to obtain a shorter tour by replacing any k of its current links by any other set of k links.From this definition it is obvious that any k-opt tour is also k'-optimal for $1 \leq k' \leq k$. Lin and Kernighan used this concept and introduced a powerful variable-opt algorithm. The algorithm changes the value of k during its execution,deciding at each iteration what the value of k should be. At each iteration step the algorithm examines, for ascending values of k, whether an interchange of k links may result in a shorter tour. Given that the exchange of r links is being considered, a series of tests is performed to determine whether r+1 link exchanges should be considered. This continues until some stopping conditions are satisfied.Hence, Lin-Kernighan Algorithm belongs to the class of local optimization algorithms. The algorithm is specified in terms of exchanges (or moves) that can convert one tour into another. Given a feasible tour, the algorithm repeatedly performs exchanges that reduce the length of the current tour, until a tour is reached for which no exchange gives an improvement.This process may be repeated many times from initial tours generated in a random way. The way the algorithm works is given below:
Let T be the current tour. At each iteration step the algorithm attempts to find two sets of links, X = {x1, ..., xr} and Y = {y1, ..., yr}, such that, if the links of X are deleted from T and replaced by the links of Y, the result is a better tour. This interchange of links is called a r-opt move.



For example this figure depicts a 4-opt move.The two sets X and Y are constructed element by element. Initially X and Y are empty. In step i a pair of links, xi and yi, are added to X and Y, respec-tively.In order to achieve a sufficient efficient algorithm, only links that fulfill the following criteria may enter X and Y:
1) The sequential exchange criterion:
x[i] and y[i] must share an endpoint, and so must y[i] and x[i+1].If t[1] denotes one of the two endpoints of x[1], we have in general: x[i] = (t[2i-1],t[2i]), y[i] = (t[2i],t[2i+1]) and x[i+1] = (t[2i+1],t[2i+2]) for $i \geq 1$.the sequence (x[1], y[1], x[2], y[2], x[3], ..., x[r], y[r]) constitutes a chain of ad-joining links.A necessary (but not sufficient) condition that the exchange of links X with links Y results in a tour is that the chain is closed, i.e., y[r] = (t[2r],t[1]). Such an exchange is called sequential.

(2) The feasibility criterion:
It is required that x[i]= (t[2i-1],t[2i]) is chosen so that, if t[2i] is joined to t[1], the resulting configuration is a tour. This feasibility criterion is used for $i \geq 3$ and guarantees that it is possible to finish with a tour. This criterion was included in the algorithm both to reduce running time and to simplify the coding
(3) The positive gain criterion:
It is required that y[i] is always chosen so that the gain, Gi, from the proposed set of exchanges is positive. Suppose gi = c(x[i]) - c(y[i]) is the gain from ex-changing x[i] with y[i]. Then Gi is the sum g[1] + g[2] + ... + g[i].This stop criterion plays a great role in the efficiency of the algorithm. The demand that every partial sum, Gi, must be positive seems immediately to be too restrictive. That this, however, is not the case, follows from the following simple fact: If a sequence of numbers has a positive sum, there is a cyclic permutation of these numbers such that every partial sum is positive. This is proved in the original paper.
(4) The disjunctivity criterion:
Finally, it is required that the sets X and Y are disjoint. This simplifies coding, reduces running time and gives an effective stop criterion.

The following is the pseudocode taken from the original paper:
1: Generate a random starting tour T:
In my algorithm my main part applies the Lin-Kernighan algorithm to the current tour. In the end of the file I generate random tours applying the Drunken Sailor Algorithm. Then, I test it on 15 different random starting tours since the algorithm will yield different results depending on the starting Tour. This method in my graph file is runRandomTours().
2: Set G = 0 (G is the best improvement so far). Choose any node t[1] and let x[1] be one of the edges of T adjacent to T[1]. Let i = 1
3: From the other endpoints t of x[i] choose y[1] to t[3] such that g[1]>0. If not possible go to step 6d)
4:Let i = i +1. Choose x[i] and y[i] such that :
a)They comply with the feasibility criterion.
b)y[i] is some available at the endpoint t[2i], shared with x[i] subject to c), d) and e). If not possible go to Step 5.
c)Guarantee the Disjunctive Criterion
d)Guarantee the Gain Criterion
e) In order to ensure the feasibility criterion, the y[i] chosen must permit the breaking of x[i+1]
f) Before y[i] is constructed, we check if closing up by joining t[2i] to t[1] will give a gain value better than the best seen previously.
5: Terminate the construction of x[i] and y[i] in Steps 2 through 4 when either no further links x[i] , y[i] satisfy 4c) -e) or when Gi <=G. If G>0, take the tour T' with f(T') = f(T) - G and repeat the whole process from Step 2, using T' as initial tour.
6: If G = 0 revoke following steps:
a)Repeat steps 4 and 5 choosing y's in order of increasing lengths as long as they satisfy gain criterion.
b) if all choices of y[2] are taken from Step 4b) with no profit from them, return to step 4a) and try for x[2]
c) If this also fails to give improvement, a further backup is performed to Step 3 where y[1] 's are examined in order of increasing length.
d) If the y[1]'s are also exhausted with no profit, we try to alternate x[1] in Step 2.
e) If this fails, a new t[1] is selected, and we repeat at Step 2.
7: The procedure terminates when all n values of t[1] have been used without profit. At this time, we may consider further random tours in Step 1.

Part D: Experiments

In this part I am doing the experiments for my algorithm. Since my algorithm does not actually depend on the metric or the triangle inequality, I have only made experiments on the general graphs as it works for any type of graph. Before I continue my experiments with random graphs, these are the results I get for the provided files:cities25,cities50,cities75,and sixnodes. I am only providing the integer part.

Graphs	Identity	Swap	2-opt	Swap &2opt	Greedy	My algorithm
cities25	6489	5027	2211	2233	2587	2021
cities50	11842	8707	2781	2686	3011	2622
cities75	18075	13126	3354	3291	3412	3095
sixnodes	9	9	9	9	8	9

Concerning the running time of my algorithm, it is not a bad one. It takes a few seconds for cities50 and cities75. According to the paper, the algorithm has an asymptotic runtime of n^2.2, which is polynomial.

Firstly, I try to generate small graphs with 6-10 nodes. The distances between the nodes I generate are between 10 and 100. Since there are a few nodes, by brute forcing all the permutations I find the optimal path and then compare it against the algorithms in graph.py file. These are the results I get for 5 random small graphs:

nr Orig Swap TwoOPt Togeth Greed MyPart Optimal
10 483 387 308 308 305 299 299
8 436 334 283 283 348 283 283
7 527 364 318 352 318 318 318
6 269 201 201 201 188 188 188
7 331 224 246 224 262 224 224

In this table nr denotes nr of nodes. My algorithm does better than greedy but cannot beat the swap and 2-opt together which seems that it works perfectly for a small number of nodes.

For a large number of nodes, my idea is to generate a dists table where the optimal path lies on the diagonal and so i precompute it. Then, I apply the drunken sailor algorithm to create a random dists table with each row being a permutation of the original row and the matrix will preserve its symmetry.