2021 겨울학기 연구참여 보고서

: PYTHON 을 이용한 TSP 구현

2022.01.03. ~ 2022.02.11. (6 주)

Logistics Lab.

지도교수 김병인

멘토 김한솔

참여학생 한선진

작성일자 : 2022.02.10

목차

그림 목차

I. 서론

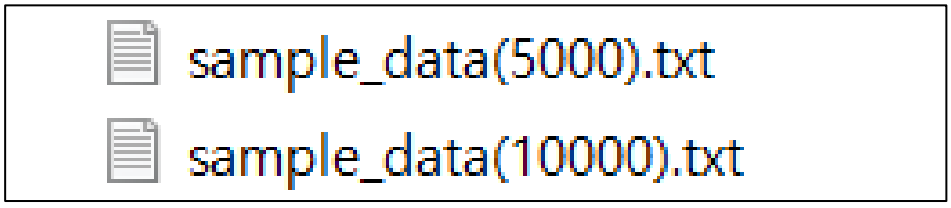
1.1. 연구 배경 및 목적

Travelling Salesman Problem (TSP) 문제는 대표적인 조합 최적화 문제로 NP-hard 에 속한다. 즉, 다항 시간 안에 풀 수 없는 문제로, 문제를 풀기 위한 다양한 Algorithm 이 존재한다. TSP 문제는 방문 판매원이 N 개의 도시를 한 번씩 방문하는 데에 걸리는 시간을 최소화시키는 방문 경로를 구하는 문제이다. TSP 문제는 물류 문제, 설비 계획 등의 산업 현장에서 다양하게 적용가능하다. 다만, 다항 시간 내에 풀 수 있는 알고리즘이 존재하지 않기 때문에 문제의 규모, 상황에 따라 알맞은 Algorithm 을 사용해야한다. 대표적인 TSP Algorithm 으로는 Nearest Neighbor Algorithm, K-Opt Algorithm, Simulated Annealing Algorithm, Genetic Algorithm, Neural Network Algorithm 등이 존재한다. 최근에는 AI (Artificial Intelligence), 강화학습 등을 TSP 에 적용시킨 Algorithm 도 연구되고 있다.

본 연구에서는 TSP Algorithm 중 가장 기본적인 Nearest Neighbor Algorithm 과 2-Opt Algorithm (Node 기준, Edge 기준), Greedy 2-Opt Algorithm 을 Python 을 활용하여 구현하고자 한다. 같은 입력값에 대해 각 Algorithm 의 효율성, 결과의 정확도를 비교해보고, Algorithm 을 개선해보는 것을 목표로 한다.

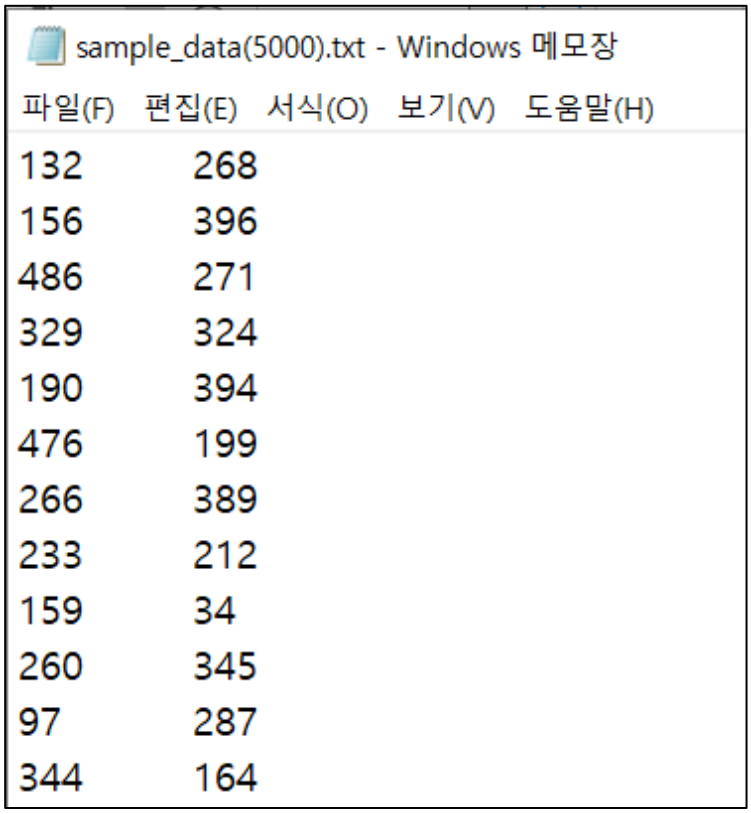
1.2. 연구 문제

Txt 형태의 파일을 통해 입력값으로 주어진 5000 개, 10000 개 도시의 x,y 좌표를 읽어 3 개의 Algorithm (Nearest Neighbor Algorithm, 2-Opt Algorithm, Greedy 2-Opt Algorithm)을 활용하여 TSP 문제를 해결한다. 추가적으로 도시의 x,y 좌표를 Random Number Generator 을 통해 설정하여 입력값으로 사용한다.



[그림 1] 도시 좌표 파일

도시의 x,y 좌표는 임의의 정수이고, Txt 파일에 x 좌표, y 좌표의 순서로 입력되어 있다. 도시의 x,y 좌표를 이용하여 모든 도시를 한번씩 방문하고 다시 출발점으로 돌아오는 데에 걸리는 총 거리와 Algorithm 실행 시간, 방문 경로를 결과값으로 출력한다.



[그림 2] 도시의 x,y 좌표

II. 연구 방법

Python 의 Networkx 라이브러리를 통해 TSP 문제를 Network 로 구현하였다. 도시의 x,y 좌표를 읽어 도시 (Node)의 Position 을 지정하고, Edge (경로)에 Node 사이의 거리를 계산하여 Distance 로 입력하였다.

2.1. Nearest Neighbor Algorithm 구현

Nearest Neighbor Algorithm 은 가장 기초적인 TSP Algorithm 으로 첫번째 도시(Node)에서 출발하여 가장 가까운 도시로 이동한 뒤, 모든 도시를 다 한 번씩 방문하고 다시 첫번째 도시로 돌아오는 방법이다. 코드 실행시간은 짧지만 도시들의 Position 에 따라 최적의 Solution 과 유사한 Solution 이 도출될 수도 있고, 부정확한 Solution 이 도출될 수도 있다.

```python
def nearest_neighbor():
    object = 1
    visited_node = [1]
    unvisited_node = list(setting.G.nodes)
    total_distance = 0
    del unvisited_node[0]
    while len(unvisited_node) != 0:
        min_object = unvisited_node[0]
        min_value = setting.G1[object][min_object]['distance']
        for i in range(len(unvisited_node)):
            distance = setting.G1[object][unvisited_node[i]]['distance']
            if distance < min_value:
                min_object = unvisited_node[i]
                min_value = distance
        total_distance += min_value
        setting.G.add_edge(object, min_object)
        visited_node.append(min_object)
        unvisited_node.remove(min_object)
        object = min_object
    total_distance += setting.G1[1][visited_node[-1]]['distance']
    setting.G.add_edge(1,visited_node[-1])
    visited_node.append(visited_node[0])
    print("<result of nearest neighbor algorithm>")
    print("total_distance is {}".format(total_distance))
    #print("visited_order is {}".format(visited_node))
    nx.draw_networkx(setting.G, pos=pos, arrows=None, with_labels=True, node_size=10, font_size=1)
    plt.savefig('nearest neighbor.png')
    #plt.show()
    return setting.G,visited_node,total_distance
```

[그림 3] Nearest Neighbor Algorithm Code

Nearest Neighbor :
Repeat
    calculate_distance(p1, p2)
    if distance is minimum
        put the node in visited_node
        delete the node from unvisited_node
        plus the distance
Until unvisited_node is null
Return G, visited_node, total_distance

[그림 4] Nearest Neighbor Algorithm Pseudocode

2.2. 2-Opt Algorithm 구현

2-Opt Algorithm 은 Nearest Neighbor Algorithm 의 결과를 바탕으로 2 개의 Node 또는 Edge 를 교환하여 총 거리가 감소하면 교환을 유지하고, 그렇지 않으면 기존의 결과를 유지하는 방법이다. Nearest Neighbor Algorithm 을 통해 도출된 결과에는 꼬인 경로가 존재할 수 있는데, 2-Opt Algorithm 을 사용하면 꼬인 경로의 순서를 변경하여 총 거리를 감소시킬 수 있다.

```
def two_opt(visited_node2,total_distance2):
    improvement = 0
    for p in range(1, len(visited_node2)-4): # 원래 - 1
        for q in range(p+2, len(visited_node2)-2): #원래 (p+1, -1)
            original = setting.G1[visited_node2[p]][visited_node2[p+1]]['distance'] \
                       + setting.G1[visited_node2[q]][visited_node2[q+1]]['distance']
            new = setting.G1[visited_node2[p]][visited_node2[q]]['distance'] \
                  + setting.G1[visited_node2[p+1]][visited_node2[q+1]]['distance']
            if original > new:
                G2.remove_edge(visited_node2[p], visited_node2[p+1])
                G2.remove_edge(visited_node2[q], visited_node2[q+1])
                G2.add_edge(visited_node2[p], visited_node2[q], color='r')
                G2.add_edge(visited_node2[p+1], visited_node2[q+1], color='r')
                list1 = visited_node2[0:p+1]
                list2 = visited_node2[p+1:q+1]
                list2.reverse()
                list3 = visited_node2[q+1:]
                visited_node2 = list1 + list2 + list3
                total_distance2 = total_distance2 - original + new
                improvement = improvement - original + new
                break
    if improvement != 0:
        two_opt(visited_node2,total_distance2)
    else:
        print("<result of 2-opt algorithm>")
        print("total_distance is {}".format(total_distance2))
        #print("visited_order is {}".format(visited_node2))
        nx.draw_networkx(G2, pos=pos, arrows=None, with_labels=True, node_size=10, font_size=1)
        plt.savefig('2-opt.png')
        #plt.show()
```

[그림 5] 2-Opt Algorithm Code

2-Opt(visited_node2, total_distance2) :

Repeat for all edges

    Exchange two edges

    if the total distance is reduced

        keep the exchange

    else

        put it back to original order

[그림 6] 2-Opt Algorithm Pseudocode

2.3. Greedy 2-Opt Algorithm 구현

Greedy 2-Opt Algorithm 은 2-Opt Algorithm 과 방식은 동일하지만, 한 번 교환된 Node 또는 Edge 는 최선의 Solution 이라고 생각하고 더 이상 변경하지 않는 방법이다. 따라서 2-Opt Algorithm 보다 효율성은 높지만 정확도는 낮을 수 있다.

```
def greedy_two_opt(visited_node3,total_distance3):
    greedy_list = []
    for p in range(1, len(visited_node3)-4):  # 원래 - 1
        for q in range(p+2, len(visited_node3)-2): #원래 (p+1, -1)
            if p in greedy_list:
                continue
            for q in range(p + 1, len(visited_node3) - 1):
                if q in greedy_list:
                    continue
            original = setting.G1[visited_node3[p]][visited_node3[p+1]]['distance'] \
                    + setting.G1[visited_node3[q]][visited_node3[q+1]]['distance']
            new = setting.G1[visited_node3[p]][visited_node3[q]]['distance'] \
                  + setting.G1[visited_node3[p+1]][visited_node3[q+1]]['distance']
            if original > new:
                greedy_list.append(p)
                greedy_list.append(q)
                G3.remove_edge(visited_node3[p], visited_node3[p+1])
                G3.remove_edge(visited_node3[q], visited_node3[q+1])
                G3.add_edge(visited_node3[p], visited_node3[q])
                G3.add_edge(visited_node3[p+1], visited_node3[q+1])
                list1 = visited_node3[0:p+1]
                list2 = visited_node3[p+1:q+1]
                list2.reverse()
                list3 = visited_node3[q+1:]
                visited_node3 = list1 + list2 + list3
                total_distance3 = total_distance3 - original + new
                break
    print("<result of greedy-2-opt algorithm>")
    print("total_distance is {}".format(total_distance3))
    #print("visited_order is {}".format(visited_node3))
```

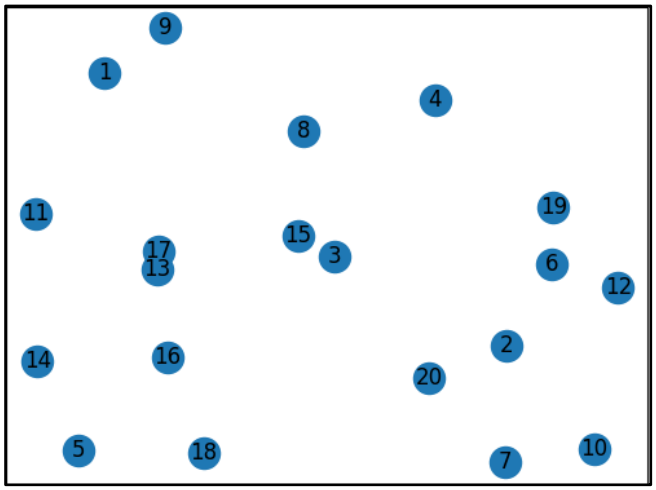
[그림 7] Greedy 2-Opt Algorithm Code

```
Greedy 2-Opt(visited_node2, total_distance2) :
  Repeat for all edges
      if the edge is not in greedy_list
            Exchange two egdes
            if the total distance is reduced
                  keep the exchange
            else
                  put it back to original order
```

[그림 8] Greedy 2-Opt Algorithm Pseudocode

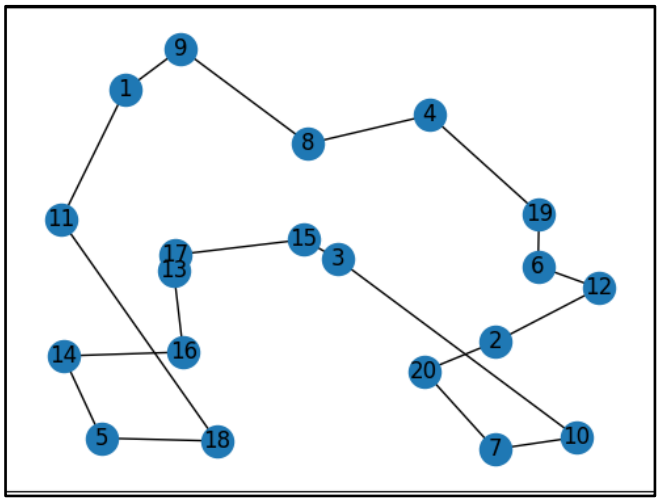
III. 연구 결과

N = 20 일 때의 결과를 바탕으로 Nearest Neighbor Algorithm, 2-Opt Algorithm, Greedy 2-Opt Algorithm 의 차이를 볼 수 있다.



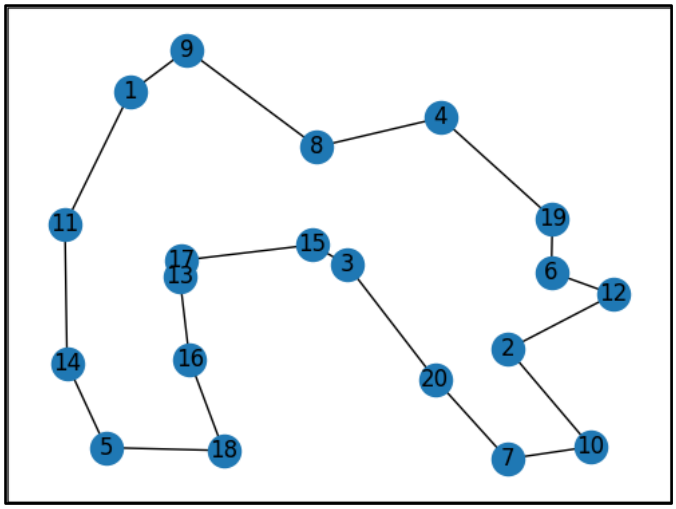
[그림 9] Initial Setting

Nearest Neighbor Algorithm 의 경우 최적의 Solution 과 동일한 경로도 존재하지만, N 값이 커질수록 꼬인 경로의 수가 증가한다. 이로 인해 전체 거리가 증가한다.

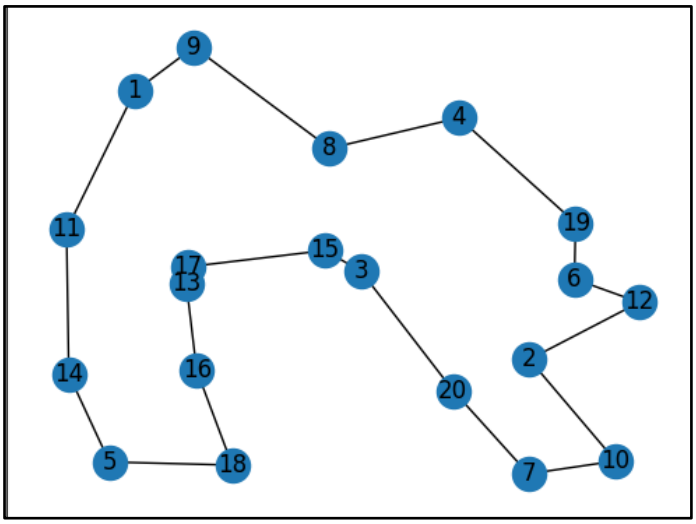


[그림 10] Result of Nearest Neighbor Algorithm

2-Opt Algorithm 과 Greedy 2-Opt Algorithm 은 Nearest Neighbor Algorithm 의 Solution 에 존재하는 꼬인 경로를 풀어 전체 거리를 감소시킨 것을 확인할 수 있다. N 값이 작을 때는 2-Opt Algorithm 과 Greedy 2-Opt Algorithm 이 비슷하거나 같은 결과를 도출하지만, N 값이 커질수록 2-Opt Algorithm 이 더 정확한 결과를 도출한다.



[그림 11] Result of 2-Opt Algorithm



[그림 12] Result of Greedy 2-Opt Algorithm

N=5000 일 때와 N=10000 일 때의 결과를 비교해보면, N=5000 일 때보다 N=10000 일 때 코드의 실행시간이 급격히 커진 것을 확인할 수 있다. 세 개의 Algorithm 이 모두 모든 Node, Edge 를 하나씩

탐색하기 때문에 N 값이 커질수록 코드의 실행시간이 급격히 증가한다. Greedy 2-Opt Algorithm 은 Nearest Neighbor Algorithm 보다는 정확한 결과를 도출하지만, 2-Opt Algorithm 에 비해서는 부정확한 결과를 도출한다.

```
Number of nodes : 5000
<result of nearest neighbor algorithm>
total_distance is 31169.634260374132
Elapsed Time : 1.263993740081787
<result of 2-opt algorithm>
total_distance is 27384.615981144874
Elapsed Time : 7.630005836486816
<result of greedy-2-opt algorithm>
total_distance is 30771.432558792454
Elasped Time : 6.852996349334717
```

[그림 13] Result of N = 5000

```
Number of nodes : 10000
<result of nearest neighbor algorithm>
total_distance is 43291.62184262445
Elapsed Time : 4.261674642562866
<result of 2-opt algorithm>
total_distance is 37448.8646602636
Elapsed Time : 37.24859380722046
<result of greedy-2-opt algorithm>
total_distance is 42106.59189133461
Elasped Time : 30.19001340866089
```

[그림 14] Result of N = 10000

IV. 결론

본 연구에서는 TSP 문제를 Nearest Neighbor Algorithm, 2-Opt Algorithm (Node, Edge 기준), Greedy 2-Opt Algorithm 을 Python 을 통해 구현하였다. N 개의 도시 좌표를 담은 입력 파일을 읽어 계산하고, Nearest Neighbor Algorithm 을 통해 Basic Solution 을 도출하고, Basic Solution 을 2-Opt Algorithm 과 Greedy 2-Opt Algorithm 을 통해 발전된 Solution 을 도출하였다. Algorithm 의 효율성은 Nearest Neighbor Algorithm, Greedy 2-Opt Algorithm, 2-Opt Algorithm 순서로 좋았고, 정확도는 2-Opt Algorithm, Greedy 2-Opt Algorithm, Nearest Neighbor Algorithm 순서로 좋았다. 즉, 효율성과 정확도가 반비례했고, 이 이유 때문에 문제의 규모와 상황에 맞는 Algorithm 을 사용해야 하는 것이다.

Algorithm 을 개선하기 위해 초기에는 Node 를 기준으로 구현했던 Algorithm 을 Edge 기준으로 수정하였다. Edge 를 기준으로 구현한 Algorithm 은 Node 를 기준으로 구현한 Algorithm 에 비해 효율성과 정확도가 높았고, Node 의 개수가 작을 때는 큰 차이가 없었으나 Node 의 개수가 커질수록 차이도 커졌다.

AI, 강화학습을 이용해 TSP 를 구현한 다양한 Algorithm 도 구현해보고 싶었지만, 기존에 존재하는 기본적인 Algorithm 을 구현해보고, 개선하는 데에서 마무리를 하게 되었다. POMO (Policy Optimization with Multiple Optima)와 같은 Algorithm 을 사용하면 기존의 Algorithm 보다 효율성, 정확도를 향상시킬 수 있고, TSP 뿐만 아니라 다양한 조합 최적화 문제에도 적용할 수 있다. TSP 문제를 추가적으로 다루게 된다면 이러한 Algorithm 도 구현해보고 싶다.

V. 후기

교내 코로나 확산으로 인해 재택근무를 한 기간이 길었지만, 연구실에서 열심히 연구하시는 선배님들과 교수님을 보면서 많은 것을 배울 수 있었다. 연구 참여를 진행한 주제도 흥미로웠고, 선배님들이 진행하시는 프로젝트의 주제도 흥미로워 보였지만, 랩미팅을 진행하시는 모습을 보며 개인적으로 부족함도 많이 느꼈다. 1 학기에도 연구참여를 이어서 할까 고민했지만, 교수님의 이산최적화 강의를 수강하고 관련 논문을 읽으며 부족함을 먼저 채울 계획이다. 마지막으로 6 주의 시간동안 연구참여를 도와 주신 멘토 김한솔 선배님과 연구참여의 기회를 제공해주시고 많은 조언해주신 김병인 교수님 감사합니다.