2021년 겨울학기 연구참여 보고서

: TSP 알고리즘 구현

2022.01.03. ~ 2022.02.11. (6주)

Logistics Lab

지도 교수 : 김병인 교수님

지도 선배 : 이승엽

참여 학생 : 홍성균

목차

I. 서론

1.1. 연구참여 목표

TSP (Traveling salesman problem) 문제는 조합 최적화 문제의 일종으로, NP-난해에 속한다. 흔히 계산 복잡도 이론에서 해를 구하기 어려운 문제의 대표적인 예로 많이 다룬다. 여러 도시들이 있고 한 도시에서 다른 도시로 이동하는 비용이 모두 주어졌을 때, 모든 도시들을 단 한 번만 방문하고 원래 시작점으로 돌아오는 최소 비용의 이동 순서를 구하는 것이다.[1] 일반적으로 다항 시간 내에는 최적해를 구하는 알고리즘이 없기 때문에, 근사 해를 구하는 것이 일반적이다.

TSP 문제는 물류시스템 문제와 함께 수많은 분야에서 활용할 수 있다. 어느 분야라도 비용을 최소화하는 것은 중요하기 때문이다. 이미 선행 연구로 다양한 알고리즘이 이미 제시되었지만, 문제를 해결하기 위해 직접 알고리즘을 구현하면서 학습에 도움이 될 것으로 보인다. 또한 이번 연구 참여를 바탕으로 알고리즘을 다른 문제에 적용할 수 있기 때문에, 본 연구의 주제로 채택하게 되었다.

본 연구에서는 TSP 문제를 해결하기 위한 알고리즘을 Python으로 구현하고, 각 알고리즘의 성능을 비교하기 위해 실행 결과를 시각화 하는 것을 목표로 한다.

II. 연구내용

2.1. TSP 알고리즘

연구에 앞서, TSP 문제에 조건을 주고자 한다. 본 연구에서는 한 도시에서 다른 도시로 이동하는 비용을 두 도시 사이의 거리로 정의한다. 따라서 본 TSP 문제는 최소 이동 거리를 구하는 것을 목표로 한다. 또한 각 도시를 Node로 정의하며, 각 Node는 x-y 좌표를 가지게 된다. 첫 Node는 임의의 Node에서 시작한다.

Python 파일을 작성하기에 앞서, 기능이 비슷한 모듈끼리 파일을 분리하였다. 이는 본 연구 이후 추가적인 연구가 이루어질 때 코드의 가시성을 높이기 위함이다.

먼저, Algorithm 파일에서는 각 알고리즘을 정의한다. Algorithm.py 에서는 ABC를 정의하며, 그 외 다른 .py에서는 문제를 해결하기 위한 알고리즘을 정의한다. 각 알고리즘은 Solution 객체를 return한다. 본 연구에서는 두 가지 알고리즘만을 제시하였지만, 추후 알고리즘을 쉽게 추가할 수 있는 이점이 있다.

---

[1] https://ko.wikipedia.org/wiki/%EC%99%B8%ED%8C%90%EC%9B%90_%EB%AC%B8%EC%A0%9C

Generator 파일에서는 문제 상황을 구현한다. 각 Node의 좌표가 저장되어 있는 Txt 파일을 읽거나, 입력한 Node 수에 맞추어 random한 좌표를 만든다.

Interface 파일에서는 2.2에서 소개할 Dash에 사용된다. 각각의 layout.py는 html을 import 하며, 실행결과를 시각적으로 보여주기 위해 사용된다.

Structure 파일에서는 Problem과 Solution의 구조를 정의한다. 그 구조는 아래와 같다. Node는 문제에서 제공하는 node의 수이며, coord는 node의 좌표를 뜻한다. 좌표는 tuple로 구성된 list의 형태이다. Route는 node들 간의 순서를 뜻하며, TSP 문제의 해가 된다. 각 node들의 index가 저장되어 있는 list 형태이다.

끝으로 viewer 파일에서는 Problem graph와 Solution graph를 출력한다. 두 graph 공통으로 node의 숫자를 출력한다. graph 상의 점은 node의 위치를 의미하고, Graph의 선은 node의 이동 경로가 된다. Solution graph 에서는 걸린 시간 (elapsed time)과 거리(distance)를 함께 출력한다.

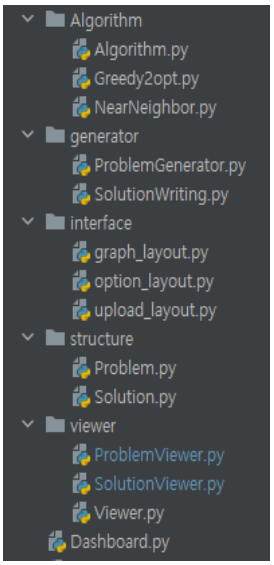

Figure 1. Python File

```python
class Problem:
    """
    :var node: 문제의 노드 수
    :var edge: 노드의 최대 좌표 (정사각형)
    :var coord: 노드의 좌표
    """

    def __init__(self, node: int, coord=None, edge=1000, name: str = ""):
        self.node: int = node
        self.edge: int = edge
        self.name: str = name

        if coord is None:
            self.coord: list[tuple] = self.randomize_coordination()
            self.name = 'Problem (Randomized)'
        else:
            self.coord: list[tuple] = coord
            self.name = 'Problem (Saved)'
```

Figure 2. Problem Structure

```python
class Solution:
    def __init__(self, problem: Problem, route: list[int], name: str = "", elapsed_time: float = 0):
        self.problem = problem
        self.route = route
        self.algorithm = name
        self.elapsed_time = elapsed_time
        self.dist = self.calculate_distance()
```

Figure 3. Solution Structure

2.1.1. Nearest Neighbor Algorithm

Nearest Neighbor 알고리즘은 (이하 NN 알고리즘) 다음 Node로 이동하기 위해 그 Node에 가장 근접한 Node를 선택하는 알고리즘이다. TSP 문제의 근사해를 찾기 위해 생각할 수 있는 가장 기본적인 알고리즘이다. 다음 node로 넘어갈 때 방문하지 않은 node들의 거리를 모두 계산한 후 가장 최소가 되는 node와 연결한다. 직관적으로 쉽게 이해할 수 있다는 장점이 있지만, 문제 상황에 따라 graph의 line이 cross 되는 경우가 많이 생기는 단점이 있다.

```python
@staticmethod
def solve_obsolete(problem: Problem) -> Solution:
    start_time = time.perf_counter()
    visited = [False for i in range(problem.node)]  # 방문 이력 초기화
    route = [0]  # 방문 루트 초기화. 초기 노드 랜덤 추출
    visited[route[0]] = True
    now_idx = route[0]

    while True:
        min_dist = problem.edge ** 2
        min_idx = -1
        first = problem.coord[now_idx]
        for order in range(0, problem.node):
            if visited[order]:
                continue
            dist = math.dist(first, problem.coord[order])
            if min_dist > dist:  # 방문하지 않았고 기존 거리보다 짧은 곳일 경우
                min_dist = dist
                min_idx = order

        if min_idx == -1:
            break
        else:
            visited[min_idx] = True
            route.append(min_idx)
            now_idx = min_idx

    return Solution(problem, route, NearNeighbor.__name__, round(time.perf_counter() - start_time, 3))
```

Figure 4. Nearest Neighbor Algorithm Code

2.1.2. Nearest Neighbor Algorithm with K-d tree

2.1.1에서 사용한 알고리즘은 시간이 방문하지 않은 node들에 대해 모두 계산이 이루어지기 때문에, 시간이 오래 걸린다는 단점이 있다. 이를 개선하기 위해, 본 연구에서는 K-d tree를 적용하고자 한다. K-d tree (K-dimensional tree) 란 k-차원 공간에서 점을 구성하기 위한 공간 분할 데이

터 구조이다.[2] 본 문제에서는 node가 x, y 좌표를 가지기 때문에, k=2 인 tree를 구성할 수 있다. 이에 대한 예시는 Figure 5와 같다. 즉, tree의 depth가 1씩 증가할 때마다, 정렬하는 기준이 바뀌게 된다. Root node에서 x 좌표에 대해서 정렬한 후, 기준이 되는 location을 기준으로 x 좌표가 더 작다면 left node에, 더 크다면 right node에 저장한다. 이후 두 번째로는 y 좌표에 대해서 다시 정렬한 후, 기준 node의 y 좌표 보다 작으면 left, 크다면 right node에 저장한다. 이번 알고리즘에서는 좌표가 저장되어 있는 list 개수의 절반을 기준으로 location를 설정하였다.

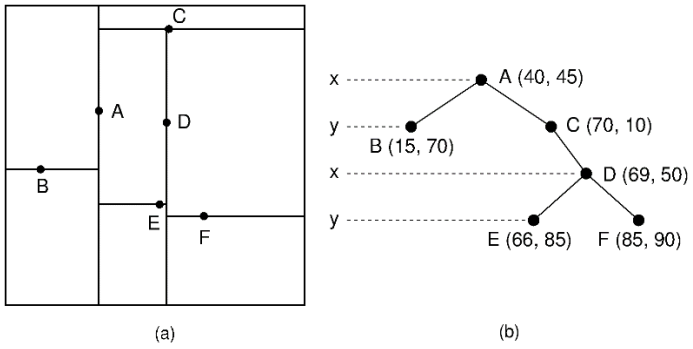

Figure 5. Example of K-d tree (k=2)

K-d tree를 적용하면 모든 node에 대해 탐색하지 않아도 되는 이점을 가지게 된다. K-d tree에서 가장 가까운 node를 찾는 과정은 다음과 같다.

1) 만약 방문하지 않은 node 이면서 현재 위치한 node 가 아닐 경우 그 node 의 위치와 node 간의 거리를 list 에 저장한다.

2) 이전 node 의 x (또는 y) 좌표보다 현재 위치한 node 의 좌표가 작을 경우, right node 로 이동한다. 큰 경우, left node 로 이동한다.

3) 2)의 경우와 반대의 node 로 이동할 경우, 그 두 node 의 좌표의 차이가 저장되어 있는 거리의 최솟값보다 작을 경우 그 방향으로 이동한다.

이 과정은 leaf node에서부터 recursive 하게 이루어진다. 알고리즘에서는 Node의 좌표에 따라 Figure 5의 영역으로 구분할 수 있다. 2)의 과정은 node가 속한 영역으로 들어가는 것을 의미하고, 3)의 과정은 node의 위치와 다른 영역의 최소가 되는 거리(수선의 발을 내릴 때의 거리)가 최솟값보다 작을 때 탐색하는 것을 의미한다. 이 방법을 사용하면 멀리 떨어져 있는 영역에 속해 있는 node들과의 거리 계산이 제외되기 때문에, 실행 시간이 오래 걸리는 기존 알고리즘의 단점을 극복할 수 있다. 이에 대한 코드는 Figure 6과 Figure 7에서 확인할 수 있다.

[2] https://en.wikipedia.org/wiki/K-d_tree

```python
@staticmethod
def kdtree(lst, depth: int = 0):
    if not lst:
        return None
    axis = depth % 2
    lst.sort(key=itemgetter(axis))
    med = len(lst) // 2

    return Node(
        location=lst[med],
        left=NearNeighbor.kdtree(lst[:med], depth + 1),
        right=NearNeighbor.kdtree(lst[med+1:], depth + 1)
    )
```

Figure 6. K-d tree Code

```python
@staticmethod
def solve_tree(node: Node, target: tuple, visited: list, depth: int = 0) -> (int, int):
    axis = depth % 2
    if node is None:
        return -1, MAX_NUM
    res = []
    if not visited[node.location[2]] and node.location[2] != target[2]:
        res.append((node.location[2], math.dist(node.location[:2], target[:2])))

    next_node = node.right if node.location[axis] < target[axis] else node.left
    if next_node is not None:
        res.append(NearNeighbor.solve_tree(next_node, target, visited, depth + 1))

    next_node = node.left if node.location[axis] < target[axis] else node.right
    if next_node is not None and (len(res) == 0 or abs(node.location[axis] - target[axis]) < min(res, key=itemgetter(1))[1]):
        res.append(NearNeighbor.solve_tree(next_node, target, visited, depth+1))

    return min(res, key=itemgetter(1)) if len(res) > 0 else (-1, MAX_NUM)
```

Figure 7. K-d tree code (solve)

2.1.3. Greedy 2-opt Algorithm

Greedy 2-opt 알고리즘이란, 1958년 Croes가 제안한 지역 탐색 알고리즘으로, 경로가 서로 cross 되어 있을 경우 이를 풀어주게 되면 이동 거리가 줄어들게 되는 아이디어이다.[3] 오른쪽 그림과 같이 1->2->3->4 로 이동할 때 그 경로가 서로 cross 될 경우, 이를 1->3->2->4 로 변경하면 이동거리를 최소화할 수 있다. 이에 대한 코드는 아래 Figure 9에 있다.

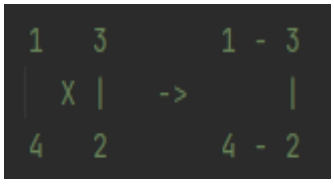

Figure 8. Cross line

[3] https://ko.wikipedia.org/wiki/2-OPT

```python
@staticmethod
def solve(problem: Problem) -> Solution:
    init_solution = NearNeighbor.solve(problem)
    start_time = time.perf_counter()

    coord = np.array([complex(c[0], c[1]) for c in problem.coord])
    out = abs(coord[..., np.newaxis] - coord).tolist()

    for i in range(problem.node-3):
        idx1 = init_solution.route[i]
        idx2 = init_solution.route[i + 1]
        for j in range(i+2, problem.node-1):
            idx3 = init_solution.route[j]
            idx4 = init_solution.route[j+1]

            """
            1   3       1 - 3
              X |   ->      |
            4   2       4 - 2
            """
            if out[idx1][idx3] + out[idx2][idx4] < out[idx1][idx2] + out[idx3][idx4]:
                init_solution.route = Greedy2opt.swap_2opt(init_solution.route, i+1, j)
                idx2 = idx3

    init_solution.elapsed_time = round(time.perf_counter() - start_time, 3)
    init_solution.algorithm = Greedy2opt.__name__
    return init_solution
```

Figure 9. Greedy 2 opt code

2.2. 알고리즘 실행결과

각 알고리즘을 실행하기에 앞서, 실험에 사용된 pc의 정보는 다음과 같다.

프로세서 : 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz 2.59 GHz

RAM : 16.0GB

시스템 종류 : 64비트 운영 체제, x64 기반 프로세서

각 알고리즘의 전제조건은 다음과 같다.

만약 random 하게 node를 생성한다면, 좌표의 최대값을 입력 받는다. 각 node의 x, y 좌표는 0에서 최댓값 사이의 임의의 실수 값을 가지게 된다. 좌표의 distribution은 uniform 하다고 가정하였다. 본 연구에 사용된 txt 파일의 경우 0에서 500 사이의 임의의 정수 값으로 작성하였다.

2.2.1. K-d tree 적용 결과

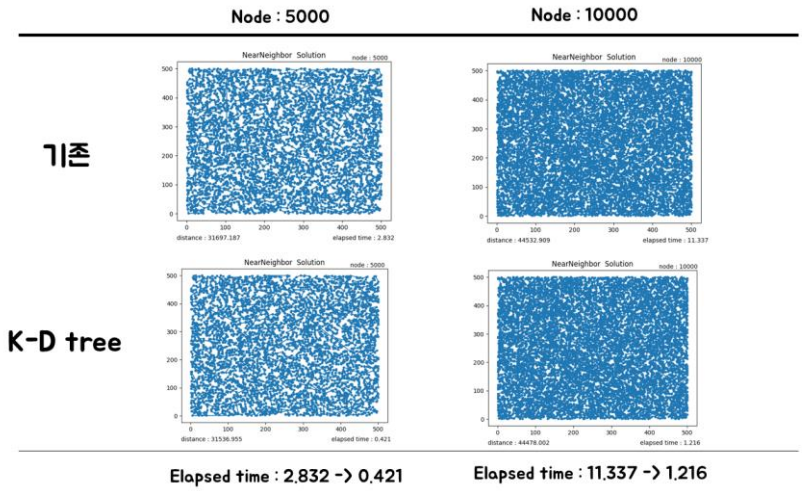

Figure 10. 알고리즘 간 실행시간 비교

위의 Figure 10 은 각각 Node의 개수가 5,000개와 10,000개 일 때 Graph와 실행 시간을 나타낸 것이다. 기존에 사용된 (2.1.1에서 사용된) NN 알고리즘의 경우 node의 개수가 증가할수록 계산해야 하는 node가 증가하기 때문에 다소 느린 결과를 보여주었다. 그러나 K-d tree가 적용된 (2.1.2에서 사용된) NN 알고리즘의 경우 기존 알고리즘 보다 실행시간이 단축되어 더 실행시간이 빨라짐을 알 수 있다.

2.2.2. 개선 알고리즘 결과

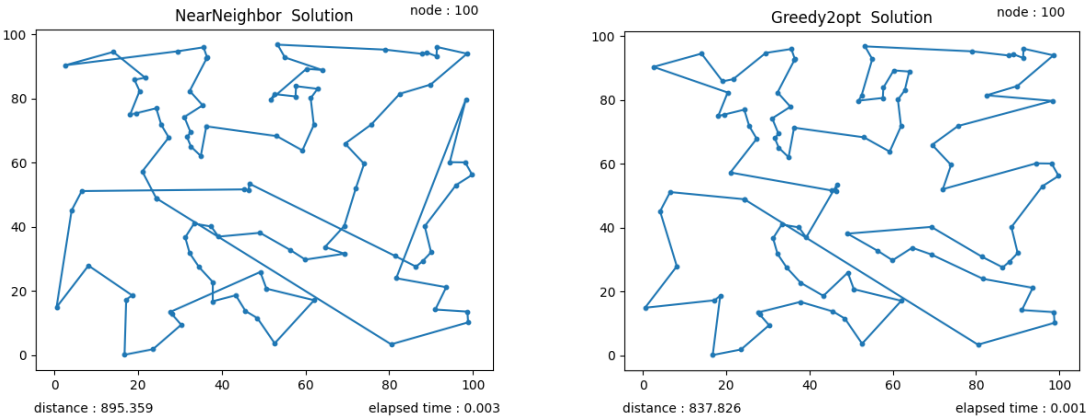

Figure 11. NN 알고리즘(좌), Greedy 2-opt(우) 실행결과 (n=100)

```
C:\Users\Hong\AppData\Local\Programs\Python\Python310\python.exe
NearNeighbor
distance : 31536.955  elapsed time : 0.407
Greedy2opt
distance : 29302.324  elapsed time : 4.172

Process finished with exit code 0
```

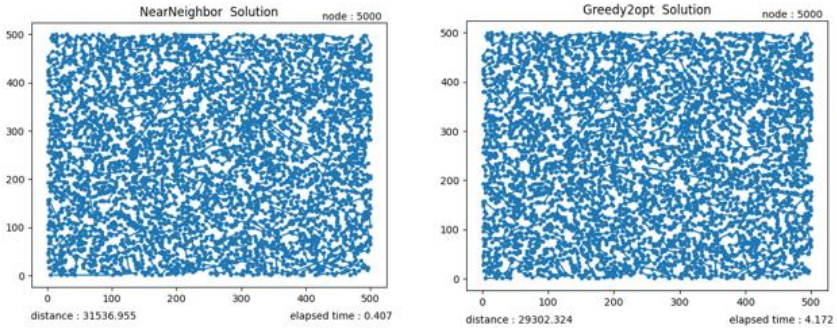

Figure 12. 코드 실행 결과 (node = 5000)

```
C:\Users\Hong\AppData\Local\Programs\Python\Python310\python.exe
NearNeighbor
distance : 44478.002  elapsed time : 1.183
Greedy2opt
distance : 40881.846  elapsed time : 19.457

Process finished with exit code 0
```

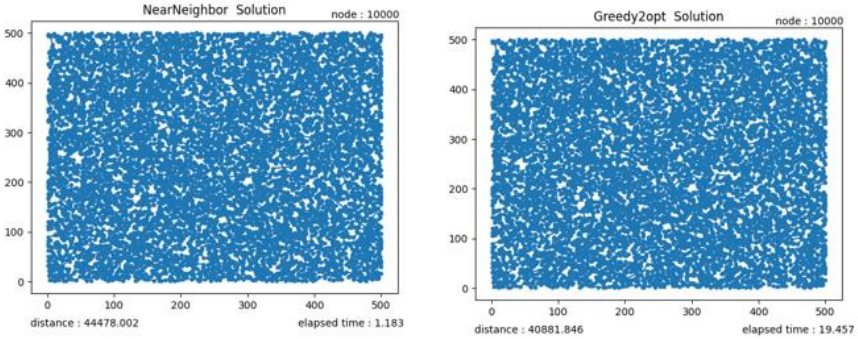

Figure 13. 코드 실행 결과 (node = 10000)

Distance	Node = 100	Node = 5000	Node = 10000
NN + K-d tree	859.359	31536.955	44478.002
Greedy 2 opt	837.826	29302.324	40881.846

Table 1. 알고리즘 간 Distance 비교

Elapsed time	Node = 100	Node = 5000	Node = 10000
NN + K-d tree	0.003	0.407	1.183
Greedy 2 opt	0.001	4.172	19.457

Table 2. 알고리즘 간 Elapsed time 비교

Figure 11~13의 결과에 따라 NN 알고리즘보다 2opt 과정을 거쳤을 때 그 거리가 줄어들었음을 알 수 있다. 하지만 node의 개수가 증가함에 따라 실행 시간이 Greedy 2 opt 알고리즘에서 매우 차이가 남을 알 수 있다. 또한, Greedy 2 opt Solution의 Graph에서 아직 cross가 남아 있기 때문에, 2-opt 과정이 완전히 돌아가지 않았음을 알 수 있다.

2.3. Dash 실행결과

2.2의 code 실행결과는 각 그래프를 png 그래프로 저장할 수 있지만, 일일이 비교해야 하는 수고로움이 존재한다. 본 연구에서는 Plotly와 Dash를 이용하여 이 그래프들을 web에 표시하고, 그 결과를 비교할 수 있게 하고자 한다.

Plotly란, Python으로 만들어진 Grapy Library이다. 이와 유사한 기능을 가진 Library로는 matplotlib가 있다. Dash란 이 Plotly를 기반으로 Web Service를 개발할 수 있는 Library이다.[4] Dash를 만들어 실행하게 되면 다음과 같이 출력 되며, 로컬호스트에 웹 페이지가 하나 열린다.

```
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app 'Dashboard' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Figure 14. Dash 실행 결과

[4] https://en.wikipedia.org/wiki/Plotly

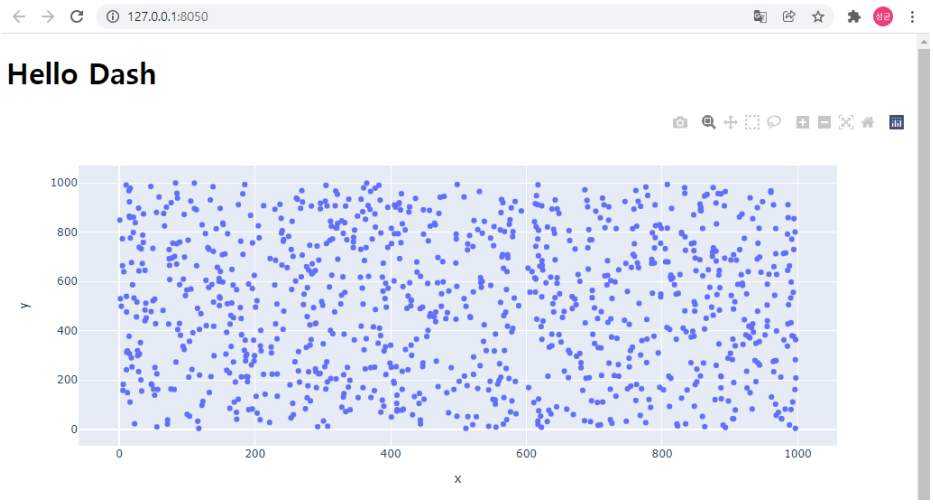

Figure 15. Graph layout

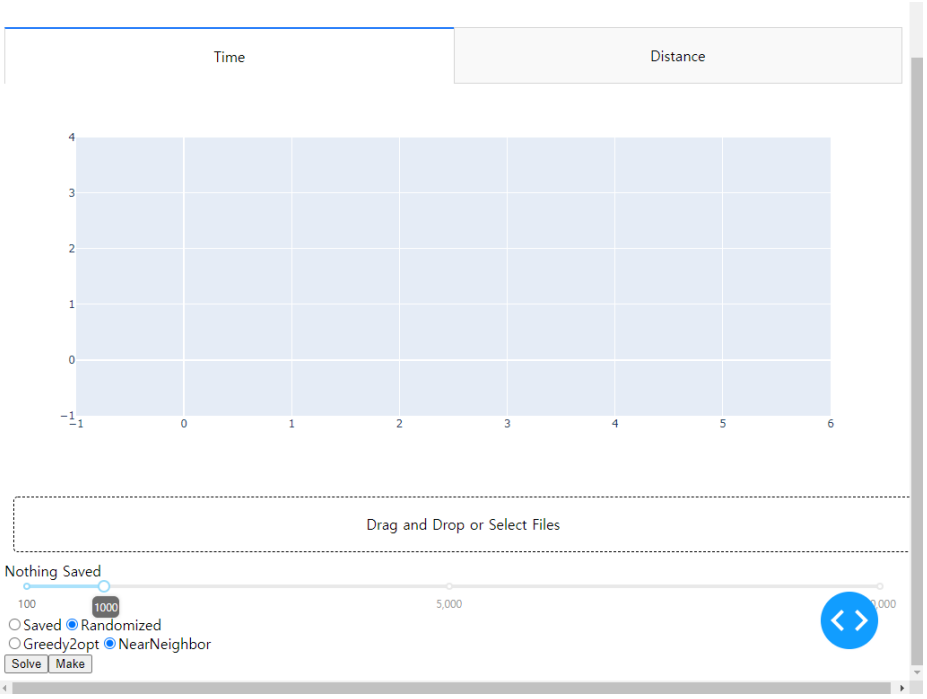

Figure 16. Option layout

Figure 15는 node들이 표시된 graph를, Figure 16는 각각 시간과 거리에 대한 graph를 출력한다. 아래의 bar를 조절하면 원하는 node의 문제를 만들고 각 알고리즘을 실행할 수 있다. 실행한 알고리즘은 Figure 15에 표시되고, 걸린 시간과 거리를 Figure 16에서 표시한다. Figure 16의 x축은 node의 수를, y축은 각각 시간과 거리를 나타낸다. 본 연구에서는 node의 수를 1,000부터 시작하여 +1,000씩 증가하여 총 10,000 개까지 실행 결과를 출력해 보았다.

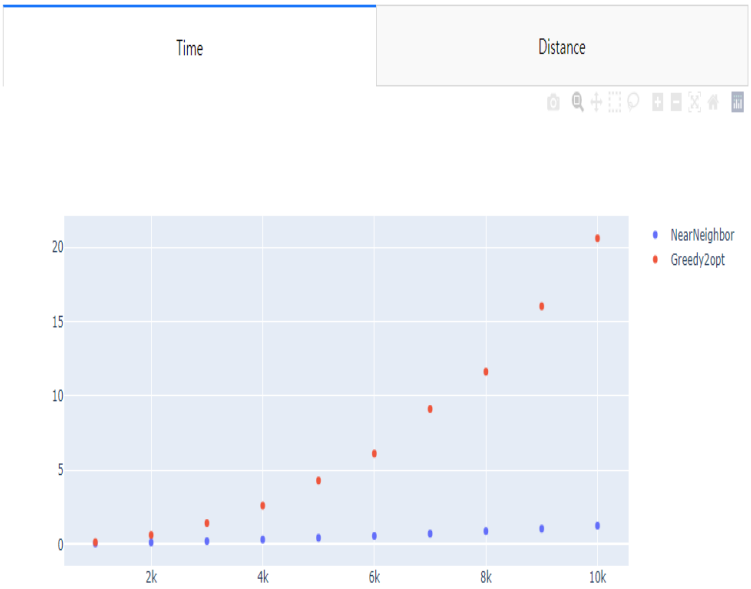

Figure 17. Time Plot

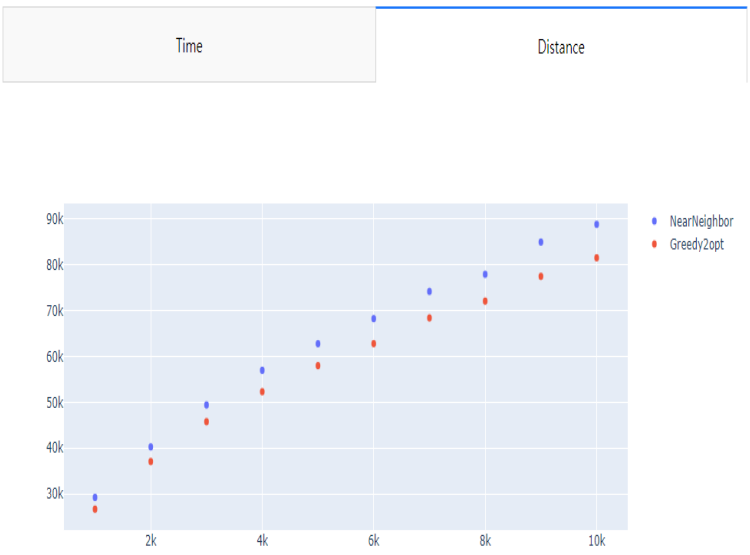

Figure 18. Distance Plot

Figure 17는 node 수에 따른 걸린 시간을, Figure 18은 node 수에 따른 거리를 나타낸다. 붉은색 점은 Greedy 2 opt 알고리즘을, 푸른색 점은 NN 알고리즘을 나타낸다. 그래프에 따르면, NN 알고리즘은 실행시간이 5초 이내로 2-opt 알고리즘과 비교할 때 더 빠른 것을 알 수 있다. Greedy 2 opt 알고리즘의 경우 NN 알고리즘보다 모든 node에 대해 거리는 더 짧아졌지만, 실행시간이 linear 하게 증가하지 않았다.

III. 후기

3.1. 개선점

본 연구에서는 TSP 문제를 해결하기 위해 두 가지 알고리즘을 구현해보았다. 그 중 NN 알고리즘은 데이터에 K-d tree 구조를 적용하여 실행 시간을 단축시켰다. 또한 Greedy 2-opt 알고리즘을 통해 NN 알고리즘에서 더 개선된 결과를 얻을 수 있었다. 또한 Dash를 사용하여 알고리즘의 실행시간과 결과를 비교할 수 있도록 시각화 하였다

하지만 본 연구에는 개선해야할 점이 존재한다. 첫째, Greedy 2-opt 알고리즘에서 cross 된 부분이 존재한다. Cross가 존재한다는 뜻은 아직 더 최적화된 해가 존재한다는 뜻이기 때문에, 이를 해결하기 위해서 코드를 개선하고 긴 실행시간을 줄여야 한다. 둘째, 다양한 알고리즘이 부족하다. 본 연구가 TSP 문제와 그 알고리즘을 이해하는 것을 목표로 하기 때문에, 앞서 소개한 두 가지 알고리즘과 더불어 휴리스틱 알고리즘을 추가로 구현할 필요가 있다. 다만 작성한 Python 코드에서 새로운 알고리즘을 쉽게 추가할 수 있도록 구현한 것에 의의를 둔다.

TSP 문제는 그 역사가 오래된 만큼 유명한 문제이고, 그동안 연구가 이루어져 수많은 해결책이 존재하며, 다양한 분야에 활용되고 있다. 문제의 종류에 따라 최적인 알고리즘은 다르기 때문에, 다양한 알고리즘을 연구하고 조사하는 것은 중요하다. 본 연구에 추가적인 알고리즘을 조사하고 고안한다면 학습에 도움이 될 것이라 기대한다.

3.2. 소감

지난 학기 군 전역 후 복학하였고, 첫 연구 참여가 되었습니다. 이전에 배웠던 전공 지식의 대부분을 잊어버린 상태였기에 연구 참여에 대한 기대보다는 '잘 할 수 있을까'에 대한 걱정이 먼저 앞섰습니다. 여러가지 이유들로 대면으로 연구실에 출근한 날 보다 비대면이 많았기 때문에 아쉬움이 남았지만, 이번 연구 참여를 통해 앞으로의 공부에 많은 도움이 될 것이라 느꼈습니다.

이번 연구 참여를 통해 TSP 알고리즘을 처음 접해보았으며, 이를 해결하기 위해 알고리즘을 어떻게 작성해야 하는지 등을 배울 수 있었습니다. 특히 문제를 시각화 하기 위해 dash를 접하면서, 연구 참여 전에는 생각하지 못한 html에 대해서도 찾아보게 되었습니다. 학부 수업에서는 쉽게 배우지 못할 경험을 할 수 있어 좋았습니다.

끝으로, 대면 연구를 허락해주신 김병인 교수님께 감사드립니다. 이번 겨울학기는 가장 뜻 깊은 계절학기로 기억될 것 같습니다. 면담과 랩미팅을 통해 주신 말씀으로 연구 방향을 설정할 수 있었고 연구를 개선하는데 많은 도움이 되었습니다. 또한 이승엽 사수님께, 많은 것을 보고 배울 수 있었고, 항상 응원해 주셔서 제가 중간에 낙오하지 않고 끝까지 올 수 있었습니다.

이만 후기를 마치겠습니다. 감사합니다.