인공지능 4주차 과제

20186889 권용한

1.Level 1: beginning

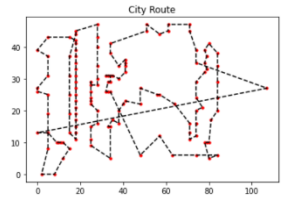
Greedy

-data1

Execution Time: 0.19710969924926758

Path: [0, 5, 13, 14, 15, 16, 24, 17, 12, 4, 11..., 84, 85, 79, 71, 72, 89, 90, 94, 95, 96, 103, 102, 110, 109, 130, 0]

Cost: 687.7892183230354



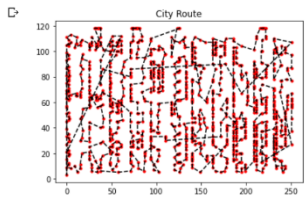
Execution Time: 0.19710969924926758
Path: [0, 5, 13, 14, 15, 16, 24, 17, 12, 4, 11, 6, 7, 1, 2, 8, 9, 3, 10, 23, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 18, 19, 20, 21, 22, 42, 43, 60, 59, 58, 57, 56, 55, 50, 51, 49, 48, 47,
Cost: 687.7892183230354

-data2

Execution Time: 0.3041253089904785

Path: [0, 22, 21, 54, 23, 1, 55, 63, 62, 60, 59.... 822, 722, 296, 295, 294, 293, 169, 161, 160, 163, 209, 254, 3, 5, 0]

Cost: 4683.17962574481



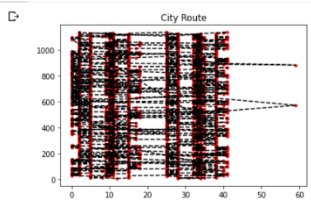
Execution Time: 0.3041253089904785
Path: [0, 22, 21, 54, 23, 1, 55, 63, 62, 60, 59, 67, 68, 78, 81, 80, 83, 118, 119, 121, 131, 137, 138, 130, 127, 148, 149, 144, 151, 152, 147, 146, 145, 138, 132, 128, 126, 170, 173, 204, 206, 207, 208, 175, 174, 226
Cost: 4683.17962574481

-data3

Execution Time: 1.1661574840545654

Path: [0, 209, 208, 582, 581, 580, 579, 578, 577...674, 317, 3355, 3584, 3583, 3585, 3586, 3356, 3357, 3358, 3359, 0]

Cost: 12695.859851064295



Execution Time: 1.1661574840545654
Path: [0, 209, 208, 582, 581, 580, 579, 578, 577, 208, 207, 204, 576, 575, 574, 573, 572, 571, 188, 189, 200, 202, 203, 864, 978, 1216, 1479, 1215, 977, 976, 1214, 863, 975, 1213, 1212, 974, 861, 1478, 1537, 1805, 18
Cost: 12695.859851064295

HC(Hill-climbing)

-data1

Execution Time: 1.3342955112457275

Path: [0, 5, 11, 4, 12, 17, 24, 16, 15, 14, 13… 51, 55, 56, 57, 58, 59, 60, 43, 42, 40, 41, 23, 10, 3, 9, 8, 2, 1, 7, 6, 0]

Cost: 618.4810805291258

-data2

Execution Time: 6.66622257232666

Path: [0, 22, 21, 54, 23, 1, 55, 63, 62, 60, 59, 67… 453, 462, 296, 295, 294, 293, 169, 161, 160, 163, 209, 254, 3, 5, 0]

Cost: 4665.70374420122

-data3

Execution Time: 23.173314332962036

Path: [0, 209, 208, 582, 581, 580, 579, 578, 577, 206… 1403, 1509, 1860, 1861, 1706, 1707, 1510, 1511, 0]

Cost: 12683.39009844431

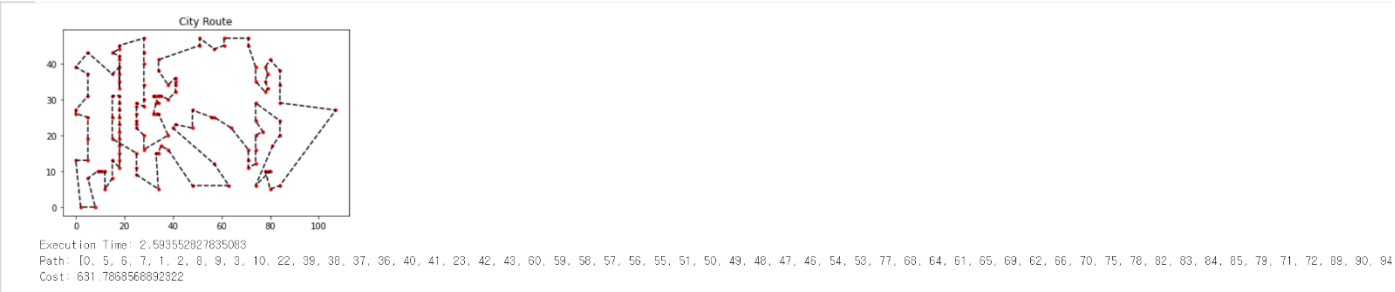
SA(Simulated Annealing)

-data1

Execution Time: 2.593552827835083

Path: [0, 5, 6, 7, 1, 2, 8, 9, 3, 10, 22, 39, 38, 37… 30, 29, 28, 27, 26, 25, 18, 24, 17, 16, 15, 14, 13, 11, 12, 4, 0]

Cost: 631.7868568892322

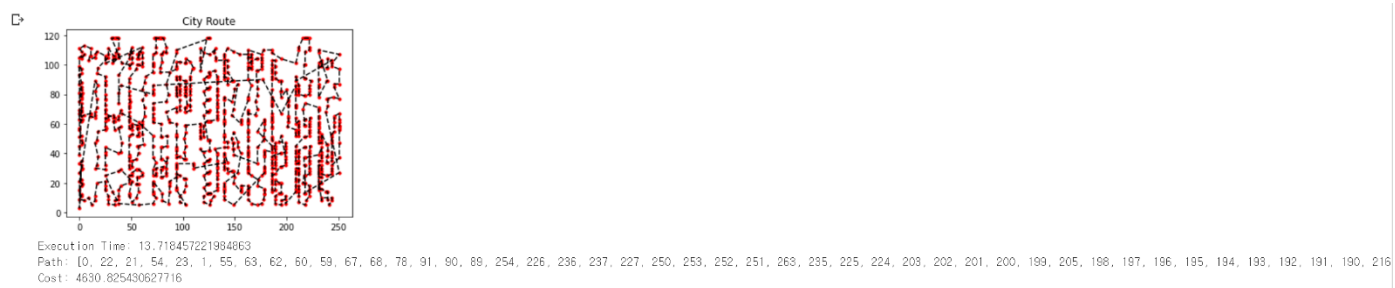


-data2

Execution Time: 13.718457221984863

Path: [0, 22, 21, 54, 23, 1, 55, 63, 62, 60, 59, 67, 68... 20, 48, 47, 19, 46, 15, 41, 42, 43, 53, 44, 45, 17, 0]

Cost: 4630.825430627716



-data3

Execution Time: 45.05232810974121

Path: [0, 209, 208, 582, 581, 580, 579, 578, 577, 206, 207... 1772, 1773, 1886, 1887, 1774, 1527, 1475, 850, 0]

Cost: 12663.926448617798

2. Lever2

-Greedy: TSP는 조합최적화 문제이다 보니 local optimal이 global optimal이 될수 있음을 증명해주지 못한다.

때문에 시작지점에서 끝지점까지는 최적의 경로로 이동하지만, 끝지점에서 다시 시작지점으로 돌아오는(왕복하기 때문) 경로가 더해진다면 최적의 수가 아니게 된다.

최소 길이의 link만 선택하며 가기 때문에 가면 갈수록 link의 길이가 길어짐은 물론, 마지막에서 시작지점으로 돌아오는 경우의 cost는 고려되지 못해 값이 크게 나온다고 생각된다.

출발지와 끝 지점과의 관계에 초점을 맞추거나 혹은 가장 작은 cost가 아닌 다른 선택 방법을 고안해야겠다는 생각이 들었다.

출발지점이 다른 노드들과 비교했을 때 가장 편차가 작은 값이 있는 곳을 선정한다면 동선이 효율적일 것으로 생각된다.

-HC: 2-opt search에서 랜덤 2개의 노드를 선택하여 교체하는 방법은 생각보다 비효율적이라 생각되었다.

가능하다면 가장 큰 cost의 link를 가진 노드를 선택하여 교체한다면 cost를 줄일 확률이 높지 않을까? 라는 의문이 생긴다. 가장 높은 cost와 그 다음 높은 cost를 가진 node를 교체하면 너무 단순해지고, 개선되지 않는 경우에 대한 상황에 대해서도 고민이 필요하다. 단순히 높은 cost순으로 탐색해 나가기엔 경우의 수와 시간제약이 있어 단순히 높은 순으로만 결정하는 방법 이외에 다른 방법도 고민이 필요해 보인다.

가장 높은 cost를 가진 link를 없애기 가장 좋은 위치는 다음 높은 cost를 가진 노드나 작은 cost의 link를 가진 노드 보다는 중간 cost의 link를 가지는 노드가 가장 확률이 높을 것으로 생각된다. 가장 높은 cost와 작은 cost를 가진 node가 교체되어 중간 값을 가진 link로 변화한다 해도 그 값의 차이는 미미할 것으로 생각되고, 반대로 높은 cost의 link끼리의 교환은 성공시 큰 효과를 낼 수 있을지 모르지만, 그건 특이 케이스에 대해서만 성공할 것으로 생각된다. 때문에 가장 높은 cost를 가진 link와 중간 혹은 random한 link를 교체해 그 효과를 꾸준하게 보는 것이 가장 효율적인 방안으로 생각된다.

-SA: 2-opt search의 경우 HC와 유사한 생각이 들었으며, Temperature를 조정하는 방법에 따라 효율이 다르게 나올 것이라 생각이 들었다. Bad case에 대한 선택이 일어날 수도 있기 때문에, HC와 비교해 cost가 더 큰 경우의 수가 등장하는 경우도 많이 있었다. 때문에 Bad case가 선택되었다면, 충분히 cost를 감소시키는 과정이 필요할 수 있겠다는 생각이 들었다. Bad case를 받아들이는 것은 물론 global optimal을 찾기 위해선 좋은 행동일수 있지만, 확률을 잘 설정하지 못한다면 제한 시간 내에 발견된 최적의 선택지가 아닌 더 나쁜 선택지를 최선으로 선택할 가능성이 있기 때문에 이를 고려해 적절한 수치 조정이 필요해 보인다.

3. Level3

-Greedy:

```
def greedy(coord_list):
    cnt_cities = len(coord_list)
    # Initialize path and insert first city index to the first and last elements
    best_path = np.zeros(cnt_cities + 1, dtype=np.int32)


    # Euclidean distance map between cities
    path_map = euclidean_distances(coord_list, coord_list)

    cities_tovisit = np.ones((cnt_cities), dtype=np.bool_)


    path_map_copy=path_map.copy()
    longest_list = np.zeros(cnt_cities, dtype=np.int32)
    for i in range(0,cnt_cities):
        longest=np.argsort(-path_map_copy[i])
        longest_list[i]=path_map_copy[i][longest[0]]
    longest_list.sort()


    best_path[0], best_path[-1] = longest_list[0], longest_list[0]
    cities_tovisit[longest_list[0]] = False
    # Iteratively Connect nearest cities
    for i in range(1, cnt_cities):
        start_idx = best_path[i - 1]
        distance_from_start = path_map[start_idx, :]
        nearest_list = np.argsort(distance_from_start)
        for idx in range(len(nearest_list)):
            # check the nearest city is visited
            if cities_tovisit[nearest_list[idx]]:
                nearest_city = nearest_list[idx]
                break
        cities_tovisit[nearest_city] = False
        best_path[i] = nearest_city

    cost_arr = path_cost(path_map, best_path)
    best_cost = cost_arr.sum()
```

위에서 언급한 내용 중에서, 출발 지점이 다른 노드들과 편차가 가장 적은 경우를 생각하여 작성한 코드이다.

여기서 편차가 가장 적은 위치는 일반적으로 생각하면 가장 중간 지점, 즉 노드들 중 제일 중간, 혹은 그 노드에서 가장 먼 노드까지의 거리가 가장 짧은 노드를 기점으로 시작하는 것이다. 일반적으로 생각하면 greedy 알고리즘의 결과는 시작지점과 끝 지점 사이의 거리 차이가 크지 않을 것이라 생각했고, 때문에 greedy 알고리즘을 사용하면서 TSP를 짧게 만들어 낼 수 있는 경우는 시작지점과 끝 지점 사이의 거리가 가장 짧은 경우가 TSP에서 cost가 제일 적을 확률이 높다고 생각했다.

때문에 우선 각 노드에서 가장 먼 노드와의 거리를 측정하고, 이 값들을 정렬해 가장 작은 값을 보여주는 노드부터 시작하면 TSP가 짧을 확률이 높다는 것이다. 그 이후엔, 기존의 greedy 알고리즘과 같은 메커니즘으로 동작하면서 탐색을 이어가고, 마지막 지점에서 출발 지점으로 연결되면서 탐색을 마치고 최종 cost를 측정하게 된다.

-HC:

위에서 언급한 내용을 토대로 가장 긴 link를 가진 노드가 선택되도록 cost를 내림차순으로 정렬해 그 중 가장 긴 값을 가지는 경우를 선택하는 경우를 가지게 했다.

나머지 한 노드를 랜덤 혹은 cost가 중간에 위치한 노드를 선택하도록 하였으며, 위 두 경우 모두 효율적으로 동작하지 못하는 경우를 대비해 두 노드 모두 랜덤으로 선택하는 경우의 수를 남겨두었다.

위 3가지 경우가 충분히 선택되도록 sub iteration을 100으로 증가시켰으며, 이에 따라 max evaluation을 100으로 감소시켰다.

```python
def two_opt(path_map, path, iterations, coord_list):
    cnt_cities = path_map.shape[0]
    # Save the best path

    cost_arr = path_cost(path_map, path)
    best_path = path.copy()
    best_cost = cost_arr.sum()
    sel_idx=np.zeros(2,dtype=int)

    for i in range(iterations):
        curr_path = best_path.copy()

        if i%3==0:
            sel_idx = np.sort(np.random.choice(np.arange(1, cnt_cities + 1), 2))

        elif i%3==1:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1] = curr_path[cost_idx[int(cnt_cities/2)]]+1
            sel_idx.sort()
        else:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1]=np.random.choice(np.arange(1, cnt_cities + 1), 1)
            sel_idx.sort()
        # Select two indices of flip points


        # Path Flip and update cost array
        curr_path[sel_idx[0]:sel_idx[1]] = np.flip(curr_path[sel_idx[0]: sel_idx[1]])
        cost_arr = path_cost(path_map, curr_path)

        # Compare to the best path
        curr_cost = cost_arr.sum()
        if curr_cost < best_cost:
            best_path = curr_path
            best_cost = curr_cost

    return best_path, best_cost
```

```python
[401] # package list
      import numpy as np
      import sys
      from sklearn.metrics.pairwise import euclidean_distances
      import matplotlib.pyplot as plt
      import time

      # Global Variables
      # Hill Climbing
      SUB_ITERATIONS = 100 # Iteration of 2-opt search in each evaluation
      MAX_EVALUATION = 100 # Max hill climbing iterations

      # Plot Settings
      PLOT_MODE = True # Draw Route
      plt.ion()

      # First City Index
      FIRST_IDX = 0
```

-SA:

```python
def two_opt_swap(path_map, path, iterations, coord_list):
    cnt_cities = path_map.shape[0]
    # Save the best path

    cost_arr = path_cost(path_map, path)
    best_path = path.copy()
    best_cost = cost_arr.sum()

    for i in range(iterations):
        curr_path = best_path.copy()
        # Select two indices of flip points
        if i%3==0:
            sel_idx = np.sort(np.random.choice(np.arange(1, cnt_cities + 1), 2))

        elif i%3==1:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1] = curr_path[cost_idx[int(cnt_cities/2)]]+1
            sel_idx.sort()
        else:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1]=np.random.choice(np.arange(1, cnt_cities + 1), 1)
            sel_idx.sort()

        # Path Flip and update cost array
        curr_path[sel_idx[0]:sel_idx[1]] = np.flip(curr_path[sel_idx[0]: sel_idx[1]])
        cost_arr = path_cost(path_map, curr_path)

        # Compare to the best path
        curr_cost = cost_arr.sum()
        if curr_cost < best_cost:
            best_path = curr_path
            best_cost = curr_cost
```

```python
    temperature = TEMPERATURE
    i=0

    while temperature > TEMP_LIMIT:
        curr_path = best_path.copy()
        # Select two indices of flip points

        if i%3==0:
            sel_idx = np.sort(np.random.choice(np.arange(1, cnt_cities + 1), 2))

        elif i%3==1:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1] = curr_path[cost_idx[int(cnt_cities/2)]]+1
            sel_idx.sort()
        else:
            cost_idx = np.argsort(-cost_arr)
            sel_idx[0] = curr_path[cost_idx[0]]+1
            sel_idx[1]=np.random.choice(np.arange(1, cnt_cities + 1), 1)
            sel_idx.sort()
        i=i+1
        # Path Flip and update cost array
        curr_path[sel_idx[0]:sel_idx[1]] = np.flip(curr_path[sel_idx[0]: sel_idx[1]])
        cost_arr = path_cost(path_map, curr_path)
        curr_cost = cost_arr.sum()

        if curr_cost <= best_cost:
            best_path, best_cost = curr_path, curr_cost

            temperature = temperature * COOLING_RATIO
        else:
            prob = 1 / np.exp((curr_cost - best_cost) / float(temperature))
            if prob > np.random.rand(1):
                best_path, best_cost = curr_path, curr_cost

        temperature = temperature * COOLING_RATIO
    return best_path, best_cost
```

```python
[380] # package list
      import numpy as np
      import sys
      from sklearn.metrics.pairwise import euclidean_distances
      import matplotlib.pyplot as plt
      import time

      # Global Variables
      # SA
      MAX_EVALUATION = 600
      SUB_ITERATIONS =25
      TEMPERATURE = 100
      COOLING_RATIO = 0.5
      COOLING_RATIO_2 = 0.6
      TEMP_LIMIT = 1

      # Plot Settings
      PLOT_MODE = True # Draw Route
      plt.ion()

      # First City Index
      FIRST_IDX = 0
```

SA의 경우 2-opt search의 경우 위의 HC 코드를 참고하여 작성하였으며, 여기에 temperature변수를 사용하는 while문 부분이 추가되었다. 기존에 0.5씩 곱하여 7번 반복동안 bad case를 받아들일 가능성을 둔 것에서 발전시켜 good case로 받아들여 진다면 0.5를 한번 더 곱하여 최선의 상황이 기록될 확률을 높였으며, 동시에 bad case가 받아들여 질 가능성 또한 남겨두었다.

4.Level4

-Greedy:

	1	2	3
기존	687.7892	4683.1796	12695.8598
향상	723.1339	4678.1615	12572.2701

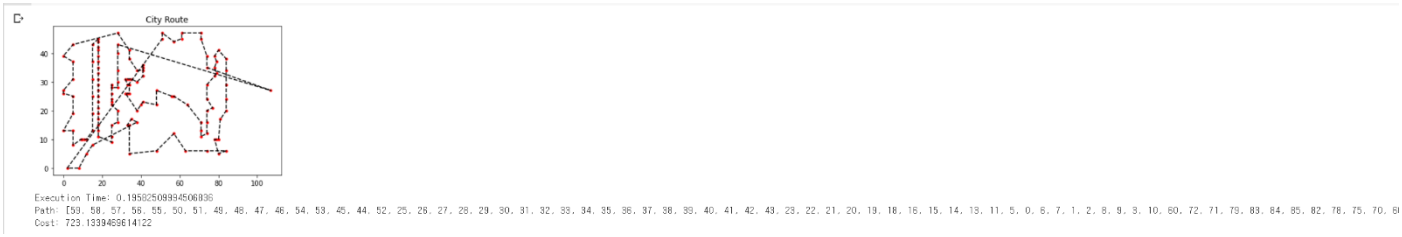
안타깝게도 data1의 경우엔 cost가 증가하였지만, data2와 data3의 경우엔 기존의 greedy 알고리즘과 비교해 보았을 때 눈에 띄는 향상이 있었음을 확인할 수 있다. Data 2와 3의 경우엔 내가 기대한 효과를 볼 수 있었던 것 같지만, data1의 경우엔 내 기대와는 달리 greedy하게 진행하면서 연결 되지 못한 노드들 사이에 큰 공백이 생긴 것 같다.

-data1:

Execution Time: 0.19582509994506836

Path: [59, 58, 57, 56, 55, 50, 51, 49, 48, 47, 46…. 17, 12, 4, 89, 90, 94, 95, 96, 103, 102, 110, 109, 130, 59]

Cost: 723.1339469614122



-data2:

Execution Time: 0.35297083854675293

Path: [136, 130, 131, 137, 144, 149, 148, 170, 173, 204…. 969, 1069, 822, 722, 296, 295, 294, 293, 254, 209, 3, 5, 136]

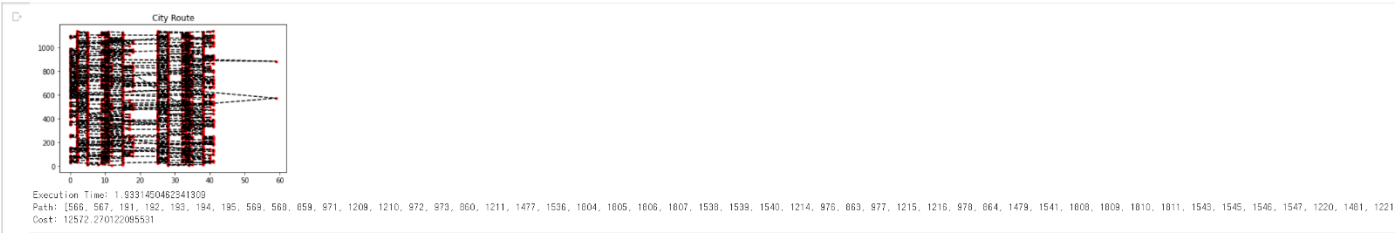
Cost: 4678.161542406278

-data3:

Execution Time: 1.9331450462341309

Path: [566, 567, 191, 192, 193, 194, 195, 569, 568…317, 3355, 3584, 3583, 3585, 3586, 3356, 3357, 3358, 3359, 566]

Cost: 12572.270122095531



-HC:

기존 알고리즘과 개선 알고리즘에 대해 6회분의 실행을 하였다. Orig은 기존의 알고리즘, update는 개선 알고리즘에 대한 결과값이며, 각각 위에는 시간, 밑은 minimum cost값, 7번째 값은 1~6번째 까지의 값들에 대한 평균값 data이다.

실험 결과 시간과 cost 모두 대략 1%가 안되는 향상을 이루어냈음을 확인할 수 있었다.

비록 수치적으로는 적어 보이지만, 이미 greedy 알고리즘이나 random하게 많은 수를 돌려 찾은 값에서 조금의 추가적인 향상을 얻을 수 있었던 것을 생각하면 유의미한 결과라고 생각된다.

orig-1

	1	2	3	4	5	6	평균
시간	1.33	1.31	1.29	1.30	1.33	1.32	1.3133
cost	618.48	633.40	628.96	595.46	618.60	618.39	618.83

update-1

	1	2	3	4	5	6	평균
시간	1.31	1.30	1.27	1.30	1.27	1.28	1.2883
cost	627.98	614.94	624.57	622.27	615.38	598.67	617.3016

orig-2

	1	2	3	4	5	6	평균
시간	6.66	6.73	6.83	6.68	6.68	6.79	6.7283
cost	4665.70	4655.45	4659.85	4647.21	4658.78	4663.62	4658.435

update-2

	1	2	3	4	5	6	평균
시간	4.96	4.97	4.95	4.98	4.91	4.95	4.953
cost	4647.96	4648.97	4639.14	4667.64	4626.43	4647.32	4646.243

orig-3

	1	2	3	4	5	6	평균
시간	23.17	22.29	22.63	22.26	22.74	22.28	22.5616
cost	12683.39	12672.84	12685.15	12677.71	12690.59	12677.20	12681.1466

update-3

	1	2	3	4	5	6	평균
시간	16.89	16.82	16.73	16.85	17.07	16.89	16.875
cost	12568.47	12568.30	12556.91	12553.81	12565.40	12524.26	12556.1916

실행 결과 예시

-data1:

Execution Time: 1.276667594909668

Path: [0, 5, 13, 11, 4, 12, 17, 24, 16, 15, 14, 18, 19…. 55, 56, 57, 58, 59, 60, 43, 42, 23, 10, 3, 9, 8, 2, 1, 7, 6, 0]

Cost: 615.3831839729837

-data2:

Execution Time: 4.967122316360474

Path: [136, 130, 131, 137, 144, 149, 148, 170, 173, 204….75, 66, 58, 57, 18, 24, 25, 2, 26, 49, 27, 3, 5, 136]

Cost: 4647.967071826004

-data3:

Execution Time: 16.8949134349823

Path: [566, 567, 191, 192, 193, 194, 195, 569, 568, 859, 971… 2415, 2064, 2065, 2067, 2068, 2069, 2070, 2071, 566]

Cost: 12568.473159649224

-SA:

기존 알고리즘과 개선 알고리즘에 대해 6회분의 실행을 하였다. Orig은 기존의 알고리즘, update는 개선 알고리즘

에 대한 결과값이며, 각각 위에는 시간, 밑은 minimum cost값, 7번째 값은 1~6번째 까지의 값들에 대한 평균값 data이다.

개선시킨 알고리즘에서 앞의 HC에서와 같이 미세한 향상을 볼 수 있다. 이때 data1은 기존의 HC보다 안좋은 데이터를 확인하게 되었는데, 이는 Bad Case를 수용하는 것이 더 손해를 본 경우로, 아마 다른 data보다 경우의 수가 작아 local optimal을 찾기가 더 쉬우며 이때 bad case를 선택하는 과정에서 손해를 본 것으로 예상된다. 이후 data2나 data3의 경우엔 SA가 더 좋은 결과를 확인할 수 있는데, 이는 경우의 수가 너무 많아 한가지 경우에서 local optimal을 찾는 것 보다 Bad case도 수용하고 random하게 탐색하는 과정에서 기존의 local optimal보다 더 좋은 값을 발견 가능한 것으로 추측된다.

Data1의 경우 약 5%, 나머지는 약 0.1~5% 정도의 향상을 이루어 냈음을 확인할 수 있다.

orig-1

	1	2	3	4	5	6	평균
시간	2.59	2.53	2.54	2.66	2.56	2.52	2.5666
cost	631.78	639.64	625.09	650.04	643.32	651.23	640.1833

update-1

	1	2	3	4	5	6	평균
시간	2.40	2.48	2.49	2.49	2.49	2.60	2.4916
cost	628.04	624.63	639.33	616.85	633.31	641.89	630.675

orig-2

	1	2	3	4	5	6	평균
시간	13.71	13.18	13.25	13.18	13.71	13.74	13.4616
cost	4630.82	4632.49	4656.36	4631.80	4639.52	4657.04	4641.3383

update-2

	1	2	3	4	5	6	평균
시간	12.73	12.72	12.68	12.97	12.79	12.94	12.805
Cost	4640.29	4637.01	4639.94	4630.95	4631.25	4641.79	4636.8716

orig-3

	1	2	3	4	5	6	평균
시간	45.05	44.17	42.32	43.06	41.81	42.61	43.17
cost	12663.92	12677.90	12667.02	12687.01	12673.45	12667.68	12672.83

update-3

	1	2	3	4	5	6	평균
시간	44.02	42.31	42.66	42.18	42.96	42.26	42.7316
cost	12550.43	12515.92	12569.31	12541.12	12545.75	12567.01	12548.256

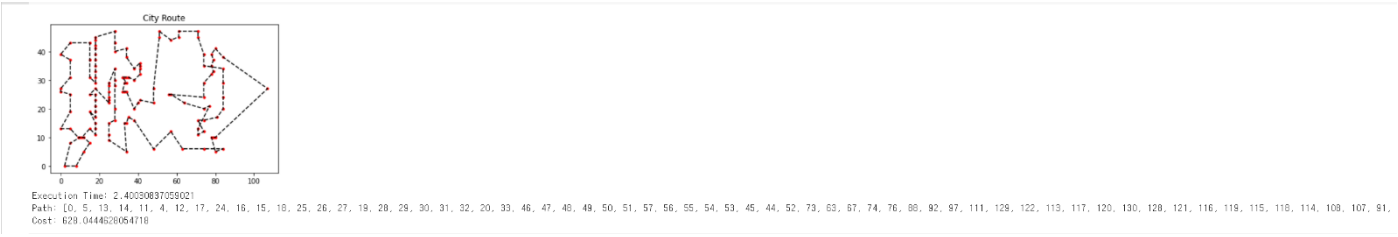
실행 결과

-data1:

Execution Time: 2.40030837059021

Path: [0, 5, 13, 14, 11, 4, 12, 17, 24, 16, 15, 18, 25….40, 39, 38, 37, 36, 35, 34, 21, 22, 23, 10, 3, 9, 8, 2, 1, 7, 6, 0]

Cost: 628.0444628054718



-data2:

Execution Time: 12.735992670059204

Path: [136, 130, 131, 137, 144, 149, 148, 170, 173, 204…133, 134, 142, 135, 124, 123, 105, 104, 103, 117, 102, 136]

Cost: 4640.295325772442

-data3:

Execution Time: 44.028634548187256

Path: [566, 567, 191, 192, 193, 194, 195, 569, 568, 859…3641, 3642, 3643, 3644, 3480, 3481, 3482, 2910, 2673, 566]

Cost: 12550.43795495229