과목: 알고리즘

교수: 정민영 교수님

Algorithm [Assignment #4]

- Genetic Algorithm with Tree Search for TSP -

홍지훈

이름: 홍지훈

학과: 소프트웨어학부

분반: 나

학번 : 20201777

과제: TSP 문제의 해를 유전 알고리즘을 이용하여 구함. 해의 공간의 부분, 계층, 혹은 전체를 트리(tree) 구조로 구성하여 활용하고 제안한 유전 알고리즘과 트리 탐색 기법의 설명 및 실험적 결과를 보고

주제 및 목표: 유전 알고리즘과 트리 탐색 기법을 활용한 TSP 문제 최적화

1. 개요

과제는 1000 개의 City 를 가지고 있는 Map 에서 TSP(Travelling Salesperson Problem)의 Shortest Path 를 찾는 문제이다. 이과정에서 유전알고리즘과 트리탐색기법을 사용해여 결과를 확인해 봐야한다.

먼저 TSP 란 한 노드에서 다른 모든 노드를 방문하는 최소 경로를 구하는 문제이다. TSP 는 NP-hard 의 대표적인 문제인데, 일반적인 다항식으로 답을 풀 수 없는 문제를 말한다. 일반적인 브루트포스 알고리즘을 사용하면 O(n!)의 시간복잡도가 걸리기 때문에 Heuristic 한 알고리즘을 주로 사용하여 문제를 해결한다.

Heuristic Algorithm 의 대표적인 것으로는 GA(Genetic Algoritim, 유전 알고리즘)이 있다. 유전 알고리즘은 실제 생물들의 진화 과정을 모방하여 문제를 해결하는 방법으로, 말그대로 유전을 통해 문제를 푸는 것을 말한다.

Tree Search 는 Tree 구조를 탐색하는 알고리즘을 의미하며, 대표적으로 DFS(깊이 우선 탐색)과 BFS(너비 우선 탐색)등이 있다.

2. 구현언어 및 방법

구현언어는 Python 을 사용하였다. 다양한 패키지들을 사용할 수 있으므로 훨씬 유리하다고 생각되어 결정하게 되었다.

처음에는 전체를 통째로 구하려고 했지만, 유의미한 결과가 나오지 않았다. 과제 명세서를 다시 읽어보고 일반적인 방법이 아닌 트리구조를 활용하여 알고리즘을 짜야 한다는 것을 알게 되었고, 그에 관련된 자료들을 찾아보았다.

사용한 패키지는 그래프를 출력하기 위해 matplotlib 를 사용하였다.

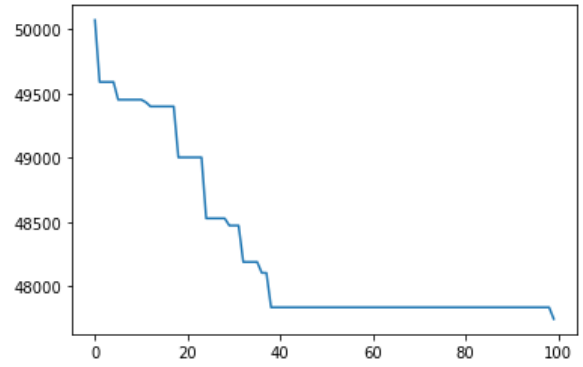
2-1. 유전 알고리즘 (Genetic Algorithm)

(1) 선택연산

일단 유전 알고리즘을 직접 구현하는데 아직 익숙하지 않아서 기본적인 연산을 시작으로 하나씩 시도해보기로 하였다. 먼저 선택연산부터 비교해보기로 하였다. 선택연산은 crossover 할 두 부모 유전자를 선택하는 과정인데, 대표적으로 룰렛휠 선택방법과 토너먼트 선택방법을 사용하여 비교해보기로 하였다. 룰렛휠 방법은 해당 염색체의 Fitness 가 높을수록 선택될 확률이 높아지는 방식이고, Fitness 는 각 Population 의 모든 거리 합으로 계산하였고 룰렛휠 방법은 Fitness 의 비율로 결정하게 되었다.

테스트를 하는데 사용한 교차연산은 이점교차를 사용하였으며, 변이확률은 0.05, 무작위변이를 사용하였다. 유전자는 100 개를 사용하였고, crossover 비율은 1.0 으로 전체를 교체하는 방법을 사용하였다. 또한 엘리트주의를 사용하였는데, 전세대에서 결과가 가장 좋은 n 가지를 다음세대로 들고가는 연산이다. 엘리트주의 수치는 2 를 사용하였다. 다음은 두 연산의 100 세대 결과이다.

- 룰렛휠 연산

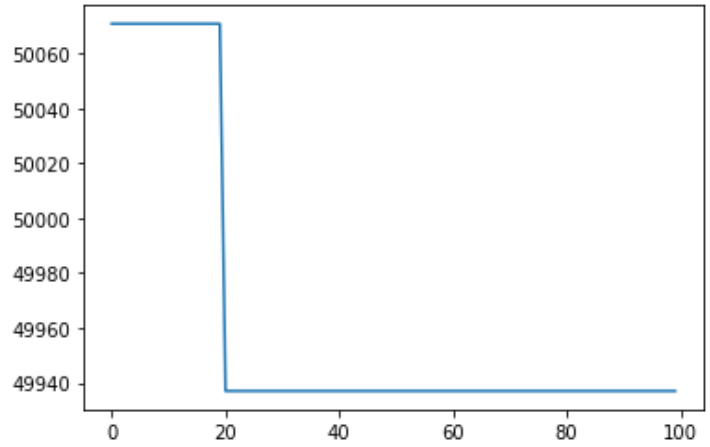


(룰렛휠 연산, 최종연산: 47743.68545576272)

룰렛휠 같은 경우는 대부분의 무작위 Population 이 비슷한 수치를 가지고 있기 때문에 엘리트주의의 영향으로 코스트가 늘어나는 일은 없지만 줄어드는 수가 많지 않은 것을 확인할 수 있다.

- 토너먼트 연산

토너먼트 선택 연산 같은 경우는 2^k 개의 값을 임의로 선택하여 토너먼트 형식으로 두 값을 계속 비교해가면서 확률에 따라 높은확률로 Fitness 가 높은 염색체가, 낮은 확률로 Fitness 가 낮은 염색체가 선택되는 방식이다. K 를 4 로, 토너먼트 비율을 0.85 로 설정한 뒤 돌린 결과이다.

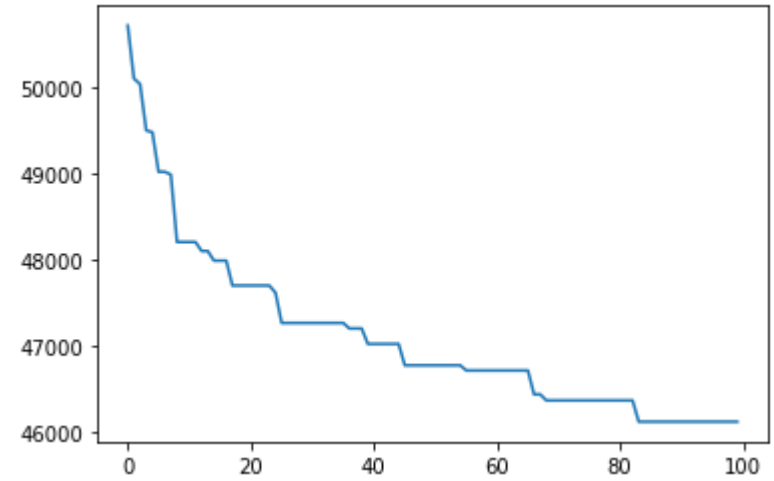


(토너먼트 연산, 최종연산: 49937.051126323226)

토너먼트 연산은 룰렛휠 연산보다 더 처참한 결과로 단 한 번의 갱신만 일어났다. 이는 최적의 해가 아닌 다른 값이 많이 선택되어 일어난 현상이라고 생각된다.

- 간단한 토너먼트 연산

마지막으로 자료조사를 하면서 토너먼트선택의 방식은 다양하게 사용한다는 사실을 알게 되었는데, 그중 토너먼트 과정을 대부분 생략하고, 임의의 k 개의 수를 뽑아서 그중 가장 우수한 해를 꺼내는 토너먼트방식이 존재하였다.



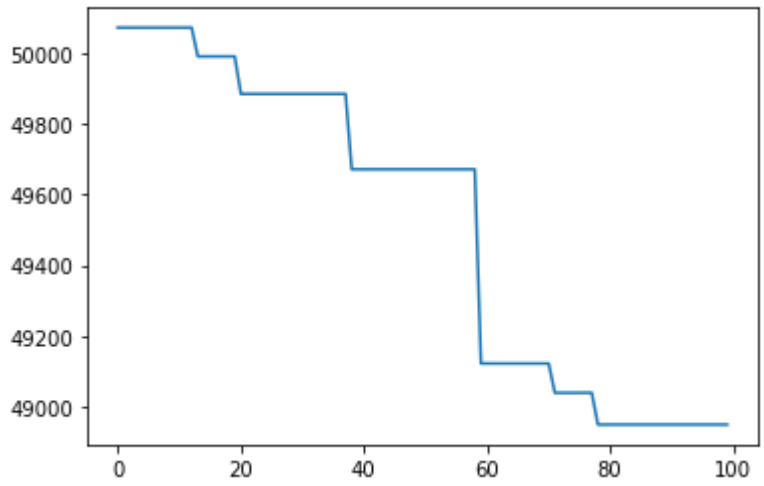
(간단한 토너먼트 연산, 최종연산: 46111.650358717736)

현재 상태에서는 오히려 이 방법이 더 좋은 결과를 가져왔다.

(2) 교차연산

교차연산도 대표적인 2 가지를 가지고 시도해보았다.

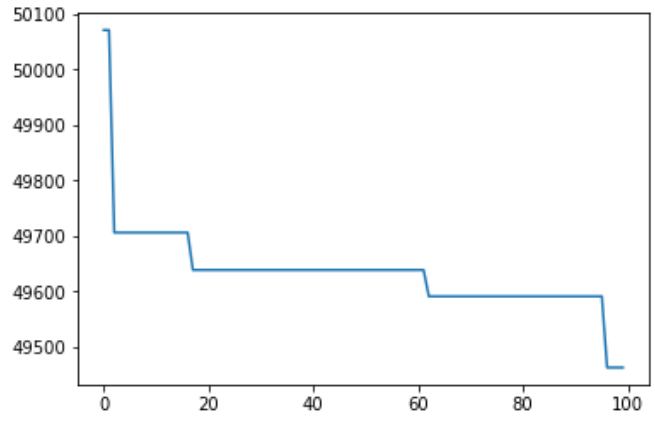
2 점교차와 순서 교차인데 두 교차방법은 매우 유사하지만, 2 점교차는 그 포지션에 있는 값들만 바꾸기 때문에 중복인 노드가 발생할 수 있다. 그 문제를 해결하기 위해 순서교차와 유사하지만, IDX 의 처음부분부터 적용하는 방법으로 교차연산을 구현하였다.



(순서 교차, 최종연산: 48950.07671924877)

결국 순서교차와 2 점교차의 방식이 비슷하다 보니 유의미한 차이를 얻진 못하였다. 그렇게 다른 교차방식을 알아보다가 두 교차방식과 유사하지만 또 다른 방법인 PMX(Partially mapped Crossover, 부분사상교차)방법을 찾았다.

PMX 방식은 순서교차와 이점교차를 섞은 방식으로, 이점교차처럼 한 부모에서 일정 부분을 복사하고 자식에게 붙여 넣는데, 두번째 부모에서 위치를 고정한 채로 중복되지 않는 값만 자식에게 물려준다. 그리고 남은 공백부분을 순서교차를 사용하여 순서대로 물려주는 방식을 사용한다.



(PMX)

하지만 역시 유의미한 해를 구하진 못했다. 결국 간선재조합 방법을 알아보았다.

간선 재조합 방법은 TSP 를 위해 개발된 방법으로, 휴리스틱 교차 방법 중 하나이다. 부모해를 분석하여 각 도시에 연결된 인접 도시 목록을 만든다. 그리고 시작도시를 정하는데, 본 코드에는 첫번째 부모해의 첫번째 노드를 시작 노드로 설정하였다. 하지만 간선 재조합 방법은 city 의 개수가 큰 상태에서는 별 큰 성능을 가지진 못하였다. 1000 개의 노드에서 이웃 노드가 3 개 이하인 경우는 시작 도시를 제외하면 3 개뿐인데, 인접 노드의 개수를 가지고 판단하는 건 시간 복잡도만 늘리는 행위라고 판단하였다.

변이연산은 균등변이를 사용하였고, 염색체 내에서 유전자들이 변이가 일어나면 swap mutation 이 일어나도록 만들었다.
여기서 깨달은 사실은 단순한 GA 만 가지곤 제대로 된 해결책을 가져올 수 없다는 사실이었다.

3. 실험 결과 및 분석

3-1 Genetic Algorithm 분석

GA 만 사용한 결과는 큰 변화를 가져오지 못했지만, 각 수치를 변화시킬 때마다 어떠한 변화가 일어나는지 확인해보도록 하였다.

각 수치의 기본값은 다음과 같다.

Gencnt: 유전알고리즘을 수행할 세대 수 = 100

M_rate: 변이과정에서 유전자가 변이될 확률 = 0.05 (5%)

Elitism: 이전 세대에서 가져올 우수한 염색체 개수 = 10

Selection: Simple tournament Selection (20)

(1) Selection 형태의 변화에 따른 분석

- Roulette Wheel Selection (k)

k = 3	K = 4	K = 5
47938	48129	47585

룰렛휠 선택에서는 k 라는 수치를 사용하여 선택압을 조절한다. k 값을 높이면 선택압이 높아지며, 일반적으로 사용하는 k 값은 3~4 라고 하였기에 3~5 값을 사용해가면서 비교해보았다.

물론 룰렛휠 선택에서는 랜덤성이 강하기 때문에 제대로 된 비교는 어렵고 초기 Population 이 대부분 임의로 정해진 최적의 해와 전혀 상관없는 염색체의 집합이기 때문에 더욱 여기서 비교하는 것은 의미가 없다고 판단된다.

- Tournament Selection (k, tournament_rate)

k = 4			k = 5			K = 6		
0.7	0.8	0.9	0.7	0.8	0.9	0.7	0.8	0.9
50130	50592	50389	49953	50710	50710	50644	50710	50710

토너먼트 선택은 k(2^k 개의 난수 생성)와 t_rate 를 사용하여 선택압을 조절한다. 여러 조합을 사용하여 비교해보려고 노력하였으나, 위의 룰렛휠 연산과 마찬가지로 랜덤성과 초기 Population 의 비적합성 때문에 유의미한 비교는 불가능하다시피 하였다.

여기서 재밌는 사실은 K 가 6(모집단이 64)인 결과를 보면 거의 처음 최소값에서 변하지 않는 사실을 확인할 수 있다. 또한 t_rate 의 값이 낮을수록 오히려 더 좋은 결과를 가져오는 것을 확인해볼 수 있다.

- Simple Tournament Selection (k)

K = 5	K = 10	K = 20	K = 30	K = 50
46633	46111	45412	45075	45169

놀라운 사실은 위의 정형화된 두 방법보다 단순히 랜덤선택에 최적의 해를 가지고 구하는 이 선택방법이 가장 좋은 결과를 가져왔다는 것이다. 특히 K 의 값이 높아지면 높은 코스트의 값들이 선택될 확률이 낮아지는 데에도 불구하고 K 가 클수록 더 좋은 결과를 가져오고 있었다.

선택연산에서 분석을 하는 과정에서 깨달은 점은 무작위 값을 사용하여 Population 을 생성한 경우, 초기 모집단 상태가 최적과 떨어져 있으면 다양한 값으로 변이를 보는 것보다 우수한 염색체를 위주로 교배를 하는게 훨씬 더 좋은 결과를 가져올 수 있다는 것을 알게 되었다. 그러나 이러한 법칙이 조금의 간격을 줄이는 데에만 사용할 수 있는지, 나중에 최적의 해를 찾을 때 문제가 생길 수도 있으므로 확실한 생각은 할 순 없다.

(2) Crossover 형태의 변화에 따른 분석

Crossover 방식은 세가지 방식 모두 큰 차이가 없어서 지금처럼 City 의 수가 큰 상황에선 큰 변화를 가지지 못할 것 이라 예상하였다.

2 Point Crossover	Order Crossover	PMX
45499	45325	45757

(3) 세대 수의 변화에 따른 분석

50	100	500	1000
45894	45421	44505	44019

세대수의 변화에 따른 shortest path 의 변화도 분명 존재한다고 생각한다. 애초에 유전 알고리즘은 세대를 거듭하면서 발전해 나가는 방법이기 때문에 세대수가 커질수록 shortest path 또한 작아지는 게 당연하다.

하지만 세대수가 늘어나면 늘어날수록 실행 시간은 늘어난다는 점이 존재한다.

(4) Population 개수의 변화에 따른 분석

50	100	500
46016	45699	45294

Population 의 개수 또한 세대 수와 같은 맥락으로, 더 많은 번식과 변이를 진행할 수 있기 때문에 Population 이 커질수록 세대 수만큼의 차이는 아니지만, 어느정도 차이를 가질 수 있을 것이다. 하지만 Population 이 늘어나면 연산양도 많아 지기 때문에 실행시간 또한 늘어난다.

100 개의 Population 을 100 세대동안 번식시키는데 약 7 분,

500 개의 Population 을 100 세대동안 번식시키는데 약 30 분가량이 걸렸다.

(5) Mutate rate 의 변화에 따른 분석

0.001	0.005	0.01	0.05	0.1
35416	38559	40055	45596	47220

Mutation rate 는 변이 확률로 적을수록 변이될 확률이 줄어든다. 변이 확률을 0.001(0.1%)에서 10%까지 다양한 확률을 가지고 분석을 시도해보았다.

처음에는 위의 결과들을 토대로 mutation rate 또한 초기 랜덤 값을 줄이기 위해 높을수록 좋을 것이라 생각하였다. 하지만 실제로 알고리즘을 돌려본 결과 mutate rate 가 낮을 때 훨씬 더 좋은 결과를 가져오게 되었다. 특히 mutation rate 를 조절할 때 다른 수치를 조절할 때보다 훨씬 큰 변화를 가져왔다.

이는 교차과정에서 잘 짜인 염색체가 변이과정에서 오히려 경로의 길이가 늘어나 버리는 일이 많이 발생하는 것 같다. 특히 city 의 수가 1000 개이므로 일반적으로 낮다고 생각하는 확률도 많은 돌연변이를 만들 수 있기 때문에 city 의 수가 많아지면 많아질수록 돌연변이 확률도 그만큼 줄어들어야 한다는 사실을 확인할 수 있다. (1000 개의 city 에서 0.05 의 확률로 돌연변이를 만들면 한 염색체에서 약 50 개의 돌연변이가 나온다)

(6) Elitism 의 수의 변화에 따른 분석

1	2	5	10	20
40434	40519	40475	39677	40473

엘리트 주의는 부모세대에서 자식세대에 물려줄 최적 결과물을 의미하는데, 조금 미세한 차이는 있었지만, 결국 다음 세대가 갱신하는 내용이 더 크기 때문에 큰 차이를 가지진 못하였다. 세대 수가 많아지면 유의미한 차이를 가질 순 있겠으나 많은 엘리트를 가지고 다음세대로 넘어가면 다음 세대에서 번식을 할 확률이 줄어들기 때문에 큰 값을 가지고 가는 건 좋지 않은 방향이라 생각된다.

결과

지금까지 분석한 요소들을 종합하여 가장 최적의 결과를 가져올 수 있는 구조를 형성하였다.

선택연산: Simple tournament Selection

교차연산: PMX

세대 수: 1000

Population 수: 100

Mutation rate: **0.005 (약 0.5%)**

Elitism: 5

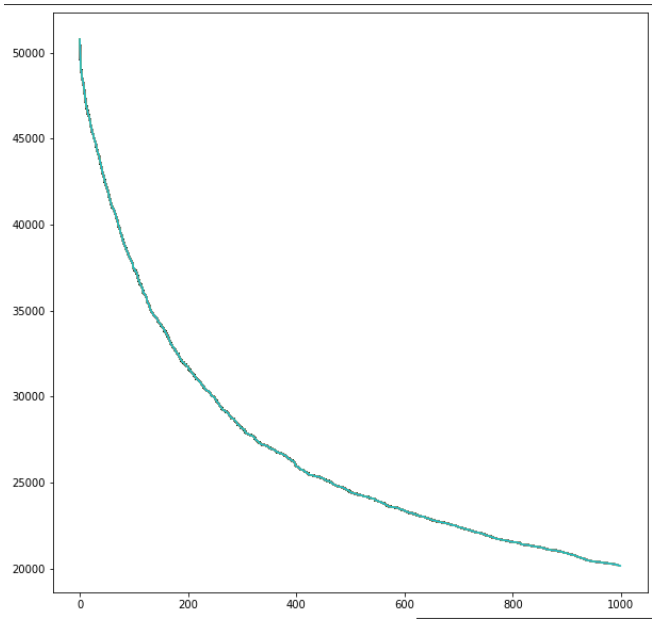
실행시간: 약 2 시간
Result: 20157.804569027037

최적 경로

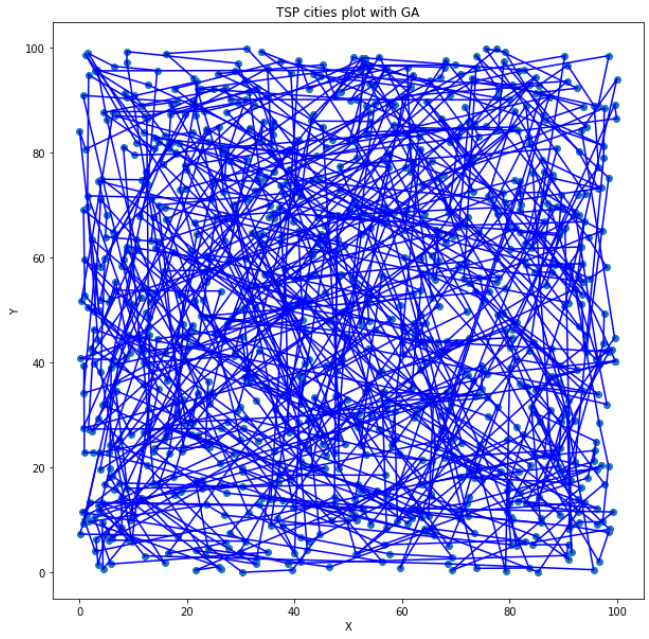
[76, 201, 87, 395, 35, 234, 720, 150, 127, 730, 812, 988, 158, 576, 629, 731, 906, 634, 401, 433, 263, 946, 592, 550, 120, 773, 594, 937, 873, 69, 209, 892, 578, 519, 165, 985, 805, 23, 897, 593, 835, 42, 509, 295, 407, 400, 618, 895, 9, 445, 14, 337, 750, 623, 597, 104, 214, 987, 506, 531, 657, 861, 795, 536, 403, 926, 233, 72, 813, 167, 456, 168, 171, 280, 740, 986, 51, 79, 989, 540, 925, 696, 885, 329, 950, 97, 777, 442, 685, 843, 152, 264, 421, 701, 880, 369, 682, 960, 727, 670, 262, 279, 913, 325, 394, 157, 799, 502, 360, 966, 677, 376, 129, 111, 853, 53, 384, 473, 846, 249, 633, 939, 746, 11, 566, 250, 389, 355, 114, 994, 661, 749, 148, 584, 821, 449, 237, 849, 482, 679, 741, 851, 616, 45, 310, 967, 742, 689, 131, 572, 196, 662, 123, 399, 20, 641, 965, 415, 538, 66, 785, 259, 681, 970, 991, 469, 922, 585, 521, 614, 815, 602, 911, 862, 446, 423, 247, 507, 649, 705, 340, 29, 241, 372, 736, 391, 549, 713, 958, 139, 811, 574, 377, 25, 206, 692, 778, 274, 443, 722, 440, 972, 702, 852, 82, 278, 260, 933, 493, 126, 334, 86, 420, 450, 212, 466, 595, 172, 761, 801, 227, 410, 755, 971, 639, 318, 774, 159, 3, 156, 309, 128, 439, 667, 784, 494, 135, 526, 422, 975, 176, 137, 147, 532, 883, 787, 728, 893, 779, 575, 598, 824, 328, 698, 894, 451, 38, 523, 191, 465, 580, 654, 914, 528, 300, 261, 354, 868, 143, 269, 142, 630, 174, 27, 246, 791, 386, 370, 704, 803, 419, 455, 896, 380, 477, 622, 666, 724, 447, 927, 396, 529, 552, 691, 807, 605, 770, 229, 559, 504, 830, 338, 556, 802, 91, 491, 460, 100, 579, 535, 715, 476, 980, 119, 907, 760, 859, 244, 766, 530, 125, 478, 437, 561, 252, 31, 7, 133, 637, 387, 647, 534, 902, 721, 817, 266, 613, 390, 901, 258, 823, 918, 879, 108, 22, 953, 866, 673, 271, 284, 358, 923, 16, 541, 659, 956, 957, 458, 95, 238, 840, 222, 462, 311, 243, 809, 140, 268, 304, 461, 789, 216, 964, 472, 505, 842, 62, 256, 909, 544, 672, 920, 55, 669, 418, 417, 690, 517, 486, 737, 658, 226, 636, 796, 792, 107, 161, 626, 149, 628, 28, 674, 166, 164, 804, 272, 392, 257, 492, 496, 617, 320, 224, 113, 546, 408, 475, 577, 607, 48, 413, 474, 656, 276, 912, 609, 941, 0, 940, 92, 361, 678, 352, 342, 50, 515, 170, 231, 828, 590, 19, 638, 75, 18, 603, 599, 708, 772, 324, 199, 219, 254, 322, 707, 17, 716, 122, 362, 189, 383, 26, 959, 930, 213, 983, 291, 154, 786, 49, 834, 56, 888, 632, 10, 539, 43, 240, 32, 99, 839, 640, 405, 412, 385, 71, 67, 917, 867, 34, 524, 457, 203, 554, 881, 74, 497, 586, 115, 863, 944, 228, 999, 438, 810, 192, 915, 762, 571, 684, 118, 979, 186, 928, 21, 573, 305, 215, 874, 973, 780, 193, 481, 655, 588, 339, 919, 70, 404, 581, 480, 220, 511, 351, 814, 427, 558, 371, 221, 60, 808, 116, 739, 869, 211, 188, 782, 942, 854, 754, 756, 949, 364, 197, 611, 665, 931, 625, 317, 729, 267, 495, 357, 886, 169, 441, 459, 910, 827, 217, 190, 106, 752, 947, 363, 788, 935, 498, 969, 488, 836, 675, 218, 604, 381, 393, 596, 870, 978, 775, 533, 47, 543, 961, 621, 759, 878, 179, 345, 565, 110, 564, 648, 101, 144, 891, 981, 294, 890,

367, 542, 738, 646, 771, 723, 717, 277, 323, 414, 606, 645, 694, 652, 15, 864, 500, 431, 711, 373, 582, 955, 624, 30, 612, 314, 587, 175, 378, 54, 242, 483, 709, 712, 270, 845, 365, 841, 865, 132, 938, 837, 5, 464, 281, 232, 236, 411, 555, 41, 453, 141, 525, 195, 664, 520, 290, 751, 610, 335, 313, 831, 898, 832, 121, 767, 343, 207, 484, 680, 187, 899, 877, 359, 428, 470, 77, 379, 856, 615, 570, 503, 289, 331, 875, 194, 876, 781, 105, 327, 434, 905, 288, 984, 463, 163, 583, 424, 687, 996, 448, 995, 89, 198, 299, 921, 726, 826, 134, 900, 600, 130, 764, 793, 943, 24, 173, 321, 838, 471, 743, 934, 40, 683, 650, 589, 245, 700, 790, 553, 366, 557, 59, 200, 591, 204, 748, 601, 202, 33, 346, 882, 308, 844, 253, 145, 102, 138, 398, 468, 301, 332, 96, 61, 527, 763, 454, 64, 697, 887, 73, 265, 513, 699, 819, 884, 718, 153, 275, 8, 651, 319, 181, 568, 783, 347, 46, 569, 432, 968, 508, 952, 255, 992, 620, 644, 768, 816, 435, 998, 136, 806, 514, 501, 1, 608, 375, 117, 676, 93, 798, 78, 81, 205, 356, 776, 734, 671, 112, 444, 753, 499, 330, 416, 567, 180, 63, 146, 303, 627, 653, 990, 88, 287, 903, 39, 479, 948, 860, 818, 951, 296, 297, 409, 425, 631, 155, 757, 94, 302, 547, 516, 871, 733, 326, 710, 223, 58, 298, 693, 820, 341, 847, 688, 548, 703, 12, 80, 945, 182, 98, 388, 974, 312, 822, 487, 208, 426, 635, 292, 747, 36, 248, 512, 745, 800, 963, 997, 68, 230, 382, 225, 2, 744, 758, 560, 336, 177, 735, 353, 924, 282, 993, 333, 916, 695, 103, 65, 797, 619, 522, 889, 452, 430, 37, 510, 273, 151, 725, 765, 686, 57, 518, 857, 563, 124, 429, 976, 160, 932, 962, 397, 855, 349, 307, 982, 368, 85, 210, 344, 239, 954, 13, 402, 293, 732, 706, 537, 850, 794, 316, 251, 315, 769, 84, 872, 109, 936, 858, 545, 489, 719, 90, 929, 829, 660, 4, 562, 350, 83, 183, 825, 6, 184, 283, 833, 904, 44, 643, 977, 908, 374, 348, 285, 306, 286, 467, 642, 663, 848, 406, 185, 490, 52, 235, 485, 436, 551, 178, 714, 162, 668]

각 세대를 지나면서 변한 최소거리 시각화 그래프



GA 로 구한 TSP 경로 시각화



결과는 그렇게 만족스러운 편은 아니지만 처음보다 그럭저럭 최적화가 된 모습을 볼 수 있었다. 또한 속도가 너무 느렸는데 최적화과정이 부족했던 것을 알 수 있다. 세대수를 더 늘렸으면 시간은 더 오래 걸렸겠지만 조금 더 최적화된 모습을 볼 수 있었을 것이다. 하지만 세대수에 따른 최적경로 그래프를 보면 점점 감소폭이 줄어드는 모습을 확인할 수 있고, 결국 GA 만으로는 1000-TSP 를 효과적으로 풀 수 없다는 사실을 알 수 있다.

3-2. 탐욕적 알고리즘을 사용한 solution 비교

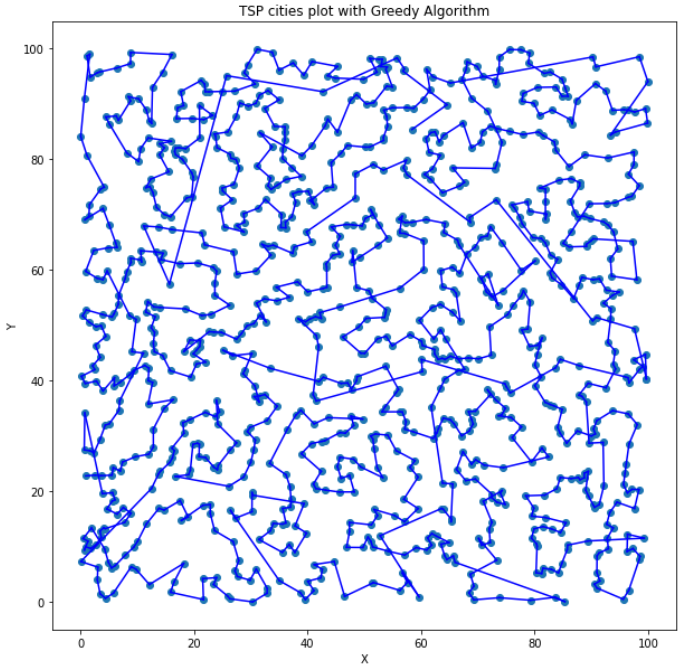
탐욕적 알고리즘을 사용하여 Large-scale TSP 를 해결해볼 수 있다.

임의의 한 도시를 시작도시로 설정하고, 그 도시에서 가장 거리가 가까우면서 아직 연결되지 않은 도시로 이동해가면서 탐욕적인 경로를 구하는 과정이다. 이과정에서 n^2 의 시간 복잡도가 소모되며, 모든 도시를 시작도시로 설정해본다면 n^3 의 시간 복잡도를 가지게 된다.

총 1000 개의 도시를 가지고 구하는 시간은 4 시간이 소요되었으며, 2705 라는 길이의 경로가 나왔다. 연산 시간은 오래 걸린 편이였지만, GA 와 비교하여 훨씬 유의미한 결과를 가져왔다. 하지만 완벽한 해를 구하지는 못하였다.

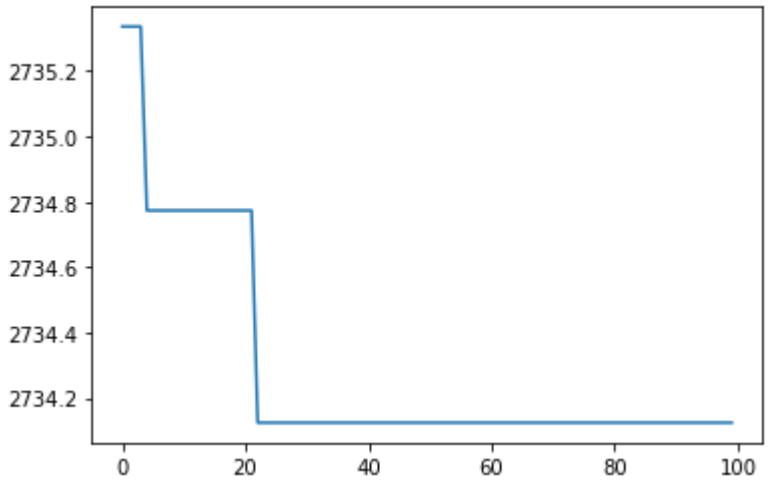
Greedy Algorithm 을 사용한 Shortest Path

실행시간: 약 4 시간 Result: 2705.6800670610314



3-3 탐욕적 알고리즘을 Population 을 만드는 과정에 삽입하여 hybrid Solution 을 제안

유전 알고리즘에서 최적화된 결과값이 나오지 못하는 가장 큰 이유 중 하나가 초기 Population 을 만드는 과정에서 완전한 무작위로 Tour 를 만들어서 초기 Cost 값이 매우 큰 상태로 시작하는 점이다. 이러한 점을 해결하기 위해 첫 Population 을 만드는 과정에서 Greedy 알고리즘을 사용하여 첫 염색체를 만들고, 그 염색체를 사용하여 유전 알고리즘을 돌려보는 방향을 생각해보게 되었다.



결과는 Greedy Algorithm 으로 초기값을 설정하여도 GA 의 성능이 거의 변하지는 않는다는 사실을 알게 되었다. 이는 City 의 수가 너무 크기 때문에 초기 Population 의 적합성과 관계없이 GA 의 기능을 전부 끌어내지 못하는 것이라 생각할 수 있다.

3-4 부르트포스 알고리즘과 비교

부르트포스 알고리즘의 경우에는 어떤 경우에도 최적의 해를 구할 수 있다. 모든 경우의 수를 전부 탐색하기 때문에 언젠간 최적의 해가 나오긴 하겠지만, TSP 문제에서 부르트포스 알고리즘을 사용하기에는 셀 수없이 많은 시간이 필요하다.

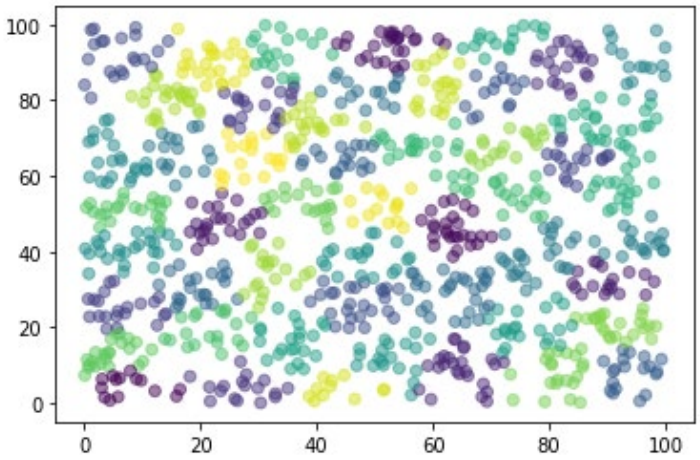
TSP 에서 부르트포스 알고리즘을 사용하면 O(n!)의 시간 복잡도가 필요하다. 이는 n 의 개수가 늘어나면 늘어날수록 시간 복잡도 또한 기하급수적으로 늘어나게 되어, n 이 10 만 넘어도 많은 시간이 걸리게 된다. 본 과제에서는 N(도시의 수)가 1000 개인 Large-Scale TSP 이기 때문에 부르트포스를 사용하려면 O(1000!) 의 시간 복잡도가 필요한데, 이는 셀 수없이 큰 수이다.

이 때문에 부르트포스(완전 무작위 탐색)을 사용하면 언젠간 최적의 해를 구할 수 있겠지만 1000 개의 도시를 가지고 있는 TSP 문제에서는 거의 무한에 가까운 시간이 걸리기 때문에 비교자체가 성립이 안된다고 생각할 수 있다.

이에 반해 탐욕적 알고리즘이나 유전 알고리즘을 사용하면 부르트포스를 사용한 만큼의 결과를 가져오진 못하지만, 현실적인 시간 내에서 탐색이 가능한 방법이기 때문에 더 의미가 있다.

3-5 Tree search 의 활용

사실 GA 와 Tree search 를 같이 활용할 만한 아이디어가 도저히 떠오르지 않았다. 가장 단순하게 떠오른 아이디어는 커다란 1000 개의 City 들을 N 개의 자식들로 계속 나누어서 트리구조를 만들고, 여러 area 를 만들어서 거기서 최적의 노드를 구하고, 다시 merge 시켜서 최적의 알고리즘을 구하는 방법을 생각하긴 했으나 merge 시키는 방법이 마땅히 떠오르지 않았고, 결국 구현하지 못하였다.



(K-클러스터링을 사용하여 City 들을 구역별로 나눈 것)

4. 결론

유전 알고리즘만 사용해서 Large-scale TSP 문제를 해결하려면 많은 시간이 필요하다. City 의 개수가 늘어날수록 유전 알고리즘의 효율성은 낮아지고, 1000 개의 city 를 가진 작업을 진행하면 짧은 시간 안에 유의미한 진행을 할 수 없다. 많은 시간을 들여도 GA 만 사용해선 한계가 있을 것이다. GA 에선 City 의 개수가 늘어나면 변의 확률을 줄여야 한다.

또한 초기 Population 을 Greedy 알고리즘을 사용하여 어느정도 최적의 해를 잡아도, city 의 개수가 너무 많아서 제대로 된 변이를 구하는 게 불가능에 가까웠다.

Large-Scale TSP 문제를 유전알고리즘으로 해결하기 위해선 다른 방법과 병행하여 사용하여야 한다.

다양한 해결법을 찾기 위해 여러 방법을 찾아보았지만 트리 탐색을 활용할 수 있는 마땅한 방법은 생각해내지 못하였고, 구현하지 못하였다. K-클러스터링을 활용하여 노드들을 여러 area 로 나누어서 각자 문제를 푸는 방법도 생각을 해봤지만, merge 를 시키는 과정에서 아이디어가 고갈되어 완벽하게 구현해내지는 못하였다. 해당 방법을 구현했으면 지금보다 훨씬 최적화된 해를 구할 수 있었을 것이다.