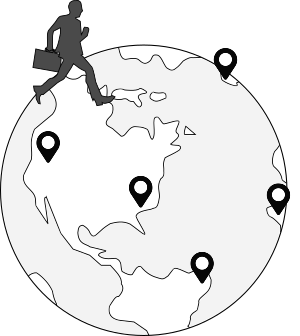
[Project #1]

Traveling Salesman Problem

[우과연의 말하는 감자(3팀)]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **이름** | **학번** | **학년** | **E-mail** |
| 팀장 | 지석훈 | 20192877 | 3 | gaenyeom37@naver.com |
| 조원1 | 기영빈 | 20192802 | 3 | youngbin218@naver.com |
| 조원2 | 장수정 | 20192869 | 3 | jo9392@naver.com |
| 조원3 | 조성현 | 20192875 | 3 | tjdgus726@naver.com |

# Introduction



외판원 순회 문제(TSP)란 여러 도시를 한 번씩 여행해서 돌아올 때까지의 최단 경로를 구하는 문제다. 해당 문제는 접근법에 대한 새로운 솔루션이 개발될 때 벤치마크 문제로 자주 사용되기에, 상당한 학술적 가치가 있는 어려운 문제이다.

본 문서에선 **1000-TSP 문제를 해결하기 위한 방안**에 관해 서술하며, 최적화 기법 중 **유전 알고리즘**에 초점을 맞춘다. 추가로, 다른 대표적인 방법인 Dynamic programming, branch and bound, Greedy Search와 2-opt Algorithm을 언급 및 활용하여 기술했다.

# Algorithms of Solving TSP

외판원 순회 문제를 해결하는 방법론은 여러 가지가 있으며, 본 문서에서는 유전 알고리즘에 초점을 두어 해당 알고리즘과 다른 알고리즘 간의 비교 분석을 기술한다.

# 2.1 Genetic Algorithm

유전 알고리즘은 1975년, John Holland가 "Adaptation on Natural and Artificial Systems"에서 처음 소개한 최적화 기법이다. 해당 알고리즘은 생물의 진화를 모방하여 문제를 해결하는 진화 연산의 대표적인 방법으로, 이론적으로 전역 최적(Global Optimal) 지점을 찾을 수 있다.

본 목차에선 사용한 유전 알고리즘의 구조와 해당 알고리즘의 python code를 기술하며, 그에 따른 실험 결과도 서술한다.

# 2.1.1 Chromosome Design

* 각각의 도시노드를 INDEX 순서인 0~999로 표현한다.
* List에 각 City의 INDEX 순서를 저장한다.
* e.g., 만약 Tour= [1, 5, 10, 999, 7, 20] 라면,  
  1 -> 5 -> 10 -> 999 -> 7 -> 20 -> 1 순서대로 도시를 방문한다

# 2.1.2 Genetic Operators

1. **Selection**

* Fitness

|  |
| --- |
| 1 / float(self.getDistance())  // Fitness 식 |

* Tournament Selection, Elitism Selection

|  |
| --- |
| elitismOffset = 0  if self.elitism:  newPopulation.saveTour(0, pop.getFittest())  elitismOffset = 4  // elitism 값에 따라 Elitism Selection 사용할 것인지 결정  // 사용한다면 elitismOffset의 값 설정 (상위 유전자 중 몇 개를 보존할 것인지)  for i in range(elitismOffset, newPopulation.populationSize()):  parent1 = self.tournamentSelection(pop)  parent2 = self.tournamentSelection(pop)  // parent1, parent2 생성  tournament = Population(self.tourmanager, self.tournamentSize, False)  for i in range(0, self.tournamentSize):  randomId = int(random.random() \* pop.populationSize())  tournament.saveTour(i, pop.getTour(randomId))  fittest = tournament.getFittest()  return fittest  // Tournament Selection 사용  // random 값 중 fitness 값이 가장 큰 값을 parent에 추가 |

1. **Crossover**

* Multipoint Crossover (다점 교차)
  + e.g., point = 2

|  |
| --- |
| child = Tour(self.tourmanager)  startPos = int(random.random() \* parent1.tourSize())  endPos = int(random.random() \* parent1.tourSize())  for i in range(0, child.tourSize()):  if startPos < endPos and i > startPos and i < endPos:  child.setCity(i, parent1.getCity(i))  elif startPos > endPos:  if not (i < startPos and i > endPos):  child.setCity(i, parent1.getCity(i))  // startPos와 endPos 사이에 해당하는 parent1의 도시를 child에 추가  for i in range(0, parent2.tourSize()):  if not child.containsCity(parent2.getCity(i)):  for ii in range(0, child.tourSize()):  if child.getCity(ii) == None:  child.setCity(ii, parent2.getCity(i))  break  // 그외 부분에 해당하는 parent2의 도시를 child에 추가  // parent1에서 이미 추가된 도시는 parent2에서 넣지 않음 (중복 x)  return child |

* e.g., point = 10

|  |
| --- |
| child = Tour(self.tourmanager)  my\_lst = list(random.sample(range(1,100),10))  my\_lst.sort()  for i in range(0, child.tourSize()):  if i < my\_lst[0] or my\_lst[1]<= i < my\_lst[2] or my\_lst[3]<= i < my\_lst[4]:  child.setCity(i, parent1.getCity(i))  elif my\_lst[5]<= i < my\_lst[6] or my\_lst[7]<= i < my\_lst[8] or my\_lst[9]<= i:  child.setCity(i, parent1.getCity(i))  // 위 조건에 해당하는 parent1의 도시를 child에 추가  for i in range(0, parent2.tourSize()):  if not child.containsCity(parent2.getCity(i)):  for ii in range(0, child.tourSize()):  if child.getCity(ii) == None:  child.setCity(ii, parent2.getCity(i))  break  // 그외 부분에 해당하는 parent2의 도시를 child에 추가  // parent1에서 이미 추가된 도시는 parent2에서 넣지 않음 (중복 x)  return child |

* Order Crossover (순서 교차)

|  |
| --- |
| child = Tour(self.tourmanager)  startPos = int(random.random() \* parent1.tourSize())  endPos = int(random.random() \* parent1.tourSize())  for i in range(0, child.tourSize()):  if startPos < endPos and i > startPos and i < endPos:  child.setCity(i, parent1.getCity(i))  elif startPos > endPos:  if not (i < startPos and i > endPos):  child.setCity(i, parent1.getCity(i))  // startPos와 endPos 사이에 해당하는 parent1의 도시를 child에 추가  for i in range(0, parent2.tourSize()):  i2=i+endPos  if(i2>=parent2.tourSize()):  i2=i2-parent2.tourSize()  if not child.containsCity(parent2.getCity(i2)):  for ii in range(0, child.tourSize()):  ii=ii+i2  if(ii>=child.tourSize()):  ii=ii-child.tourSize()  if child.getCity(ii) == None:  child.setCity(ii, parent2.getCity(i))  break  // 순서 교차에 의해 해당하는 parent2의 도시를 child에 추가  return child |

1. **Mutation**

* Swap

|  |
| --- |
| for i in range(elitismOffset, newPopulation.populationSize()):  self.mutate(newPopulation.getTour(i))  // 각각 도시에 대해 mutate 진행  for tourPos1 in range(0, tour.tourSize()):  if random.random() < self.mutationRate:  tourPos2 = int(tour.tourSize() \* random.random())  city1 = tour.getCity(tourPos1)  city2 = tour.getCity(tourPos2)  tour.setCity(tourPos2, city1)  tour.setCity(tourPos1, city2)  // random 값이 미리 지정한 mutate 확률보다 작으면 mutate 진행 |

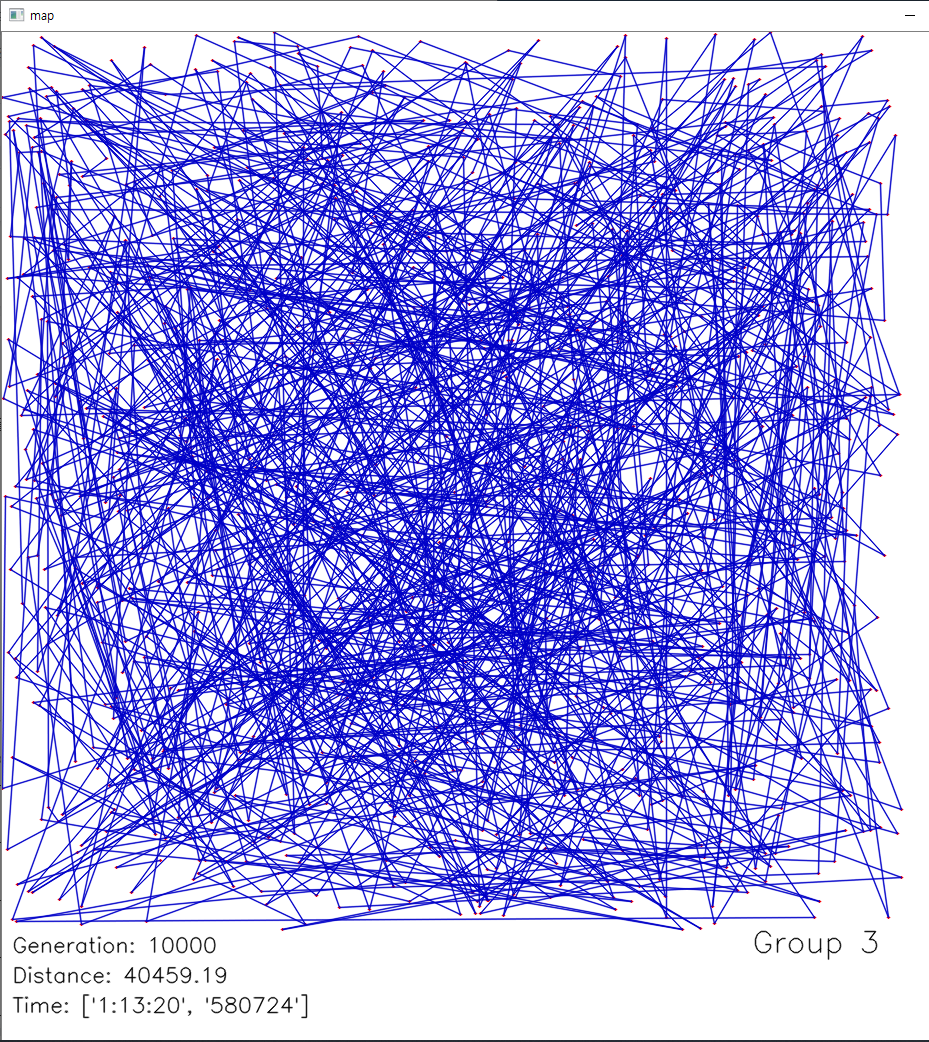
# 2.1.3 Experimental Results

## 2.1.3.1 Test Environment

* PyCharm 2019.3.5 (python 3.5)
* Windows 10(x64)
* Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
* NVIDIA GeForce GTX 1060 3GB
* 16GB RAM

## 

## 2.1.3.2 Result



* 경로 최종 길이 : 40459.19
* 탐색 시간 : 4400 seconds
* 탐색 경로 :

|  |
| --- |
| 855 352 718 756 600 889 274 836 86 369 853 978 780 251 278 546 36 776 252 341 764 22 895 292 485 259 42 102 609 877 182 995 696 796 357 886 225 208 202 420 940 77 317 667 986 181 882 134 304 298 58 132 335 826 755 376 574 987 216 421 577 570 802 140 604 646 769 139 25 383 706 825 207 107 342 829 828 961 416 125 339 631 462 870 519 295 153 813 564 354 8 301 330 494 922 953 428 380 662 495 699 971 26 963 345 326 351 639 670 47 896 789 526 266 324 438 387 165 821 325 698 608 768 637 490 303 661 784 702 884 151 996 714 51 378 0 617 887 547 742 186 943 439 18 456 932 959 400 833 866 659 518 944 695 120 220 579 240 114 841 363 44 283 558 914 997 651 255 539 54 935 785 758 332 402 731 967 34 672 175 162 880 52 308 159 10 413 927 299 668 35 941 688 720 2 730 648 629 583 188 729 893 976 812 931 550 743 902 955 211 763 626 779 249 979 569 94 436 591 174 306 867 783 123 6 344 682 128 717 281 183 552 771 149 575 253 999 705 554 664 657 76 860 224 733 13 515 396 950 241 881 601 192 915 610 408 517 364 318 650 514 725 722 786 738 499 568 405 806 981 83 97 327 810 947 92 426 27 641 233 528 23 170 632 262 727 24 556 470 830 353 939 493 100 16 576 103 989 767 673 726 745 777 801 84 196 686 643 50 741 477 952 993 757 423 964 66 787 464 41 443 331 954 381 508 907 917 734 919 838 936 693 746 990 390 256 305 242 496 235 45 260 309 145 164 565 856 691 747 248 204 488 973 90 709 430 553 173 446 290 500 992 962 302 535 974 343 817 584 819 243 839 858 236 437 138 529 865 291 523 232 593 219 588 960 773 73 715 277 484 790 356 425 816 956 900 520 501 778 845 732 221 824 217 834 473 728 509 71 560 988 491 176 998 711 154 486 594 848 388 545 169 969 239 541 804 684 753 616 272 760 49 820 375 949 925 160 141 370 638 61 461 261 890 167 96 397 878 487 851 451 307 359 850 724 372 573 723 645 475 748 677 772 316 831 33 91 349 124 401 360 857 792 991 534 605 770 775 751 849 504 38 121 690 108 3 938 585 704 965 218 190 551 458 106 447 55 782 562 524 926 903 675 282 46 191 130 864 805 250 578 101 832 113 4 630 31 942 161 19 284 30 80 9 749 126 474 166 460 754 542 536 229 793 238 459 453 566 854 701 177 111 781 948 88 432 863 209 431 842 811 623 972 69 279 336 213 906 373 32 482 595 506 226 155 561 665 510 489 797 597 586 109 223 531 310 814 379 361 286 983 846 532 200 5 393 1 980 559 471 611 572 719 15 606 334 143 198 640 527 567 680 311 156 297 516 287 708 246 20 642 398 206 410 975 692 293 427 945 807 445 892 469 613 288 117 118 371 818 909 888 951 937 654 505 861 843 409 144 384 411 180 247 346 394 465 289 280 68 70 314 319 522 368 11 452 338 29 313 658 418 74 212 503 985 48 136 934 958 366 275 119 184 694 513 899 977 581 348 823 419 739 137 483 201 131 441 913 350 511 837 320 263 563 14 231 406 480 285 918 920 822 521 194 803 676 333 946 105 147 679 422 40 847 267 329 230 276 766 93 434 467 655 911 710 671 875 502 62 355 85 81 894 296 794 602 872 744 582 315 99 809 178 540 681 923 78 59 908 897 636 417 660 862 644 898 135 82 548 37 873 365 478 625 358 678 189 587 392 620 968 245 697 389 450 703 512 43 115 222 273 759 685 799 133 442 112 994 883 244 215 407 874 39 321 340 237 148 507 538 982 270 403 929 615 79 386 901 116 385 53 622 666 736 152 737 912 347 444 95 835 498 885 762 652 791 689 557 89 580 627 367 924 12 669 683 492 337 933 210 227 228 984 382 868 707 65 713 891 322 265 619 537 740 28 172 129 448 808 395 399 687 294 67 735 377 543 264 800 970 75 150 142 555 412 214 599 63 457 859 530 603 904 391 815 876 110 98 203 312 472 852 774 656 481 716 374 455 429 17 598 905 7 87 647 628 197 916 146 788 440 653 596 449 424 415 328 635 674 466 205 930 60 199 57 193 269 624 621 72 752 414 589 840 592 435 163 966 590 362 468 700 663 433 634 179 454 56 921 614 798 633 618 479 195 404 649 158 257 544 234 157 549 122 258 928 612 268 497 844 607 765 571 761 721 171 525 323 869 271 871 795 185 476 168 879 127 463 957 21 254 533 104 187 750 827 712 910 300 64 |

**Crossover 방식에 따른 결과 분석**

(Cities : 1000, Population : 100, generation : 1000, mutationRate : 0.05 , tournament\_size : 25, , elitism : 2)

|  |  |  |  |
| --- | --- | --- | --- |
| crossover | 3분할 | 11분할 | order |
| 시작 cost | 49944 | 50670 | 50314 |
| 결과 cost | 44270 | 44273 | 44150 |
| 감소 비율 | 0.8864 | 0.8737 | 0.8775 |

**MutationRate에 따른 결과 분석**

(Cities : 1000, Population : 100, generation : 1000, tournament\_size : 25, cross : 2분할, elitism : 2)

|  |  |  |  |
| --- | --- | --- | --- |
| mutationRate | 0.025 | 0.05 | 0.1 |
| 시작 cost | 50709 | 49944 | 50715 |
| 결과 cost | 41293 | 44270 | 46126 |
| 감소 비율 | 0.8143 | 0.8864 | 0.9095 |

**Generation 개수에 따른 결과 분석**

(Cities : 1000, Population : 20, mutationRate : 0.025, cross : 2분할, tournament\_size : 10 elitism : 2)

|  |  |  |
| --- | --- | --- |
| generation | 1000 | 10000 |
| 시작 cost | 50637 | 50815 |
| 결과 cost | 42028 | 39328 |
| 감소 비율 | 0.8300 | 0.7739 |
| 시간 | 0:17:28 | 7:22:20 |

**Population 개수에 따른 결과분석**

(Cities : 1000, generation : 10000, mutationRate : 0.025, cross : 2분할, elitism : 2)

|  |  |  |
| --- | --- | --- |
| **Population** | **10** | **20** |
| **tournament\_size** | **5** | **10** |
| **시작 cost** | **51198** | **50815** |
| **결과 cost** | **40026** | **39328** |
| **감소 비율** | **0.7818** | **0.7739** |
| **시간** | **1:32:00** | **7:22:20** |

**Tournament\_size에 따른 결과분석**

(Cities : 1000, Population : 100, generation : 1000, mutationRate : 0.05, cross : 2분할, elitism : 2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| tournament\_size | 25 | 50 | 75 | 100 |
| 시작 cost | 49944 | 50637 | 50687 | 50342 |
| 결과 cost | 44270 | 43456 | 42982 | 43078 |
| 감소 비율 | 0.8864 | 0.8582 | 0.8480 | 0.8557 |

**Elitism 개수에 따른 결과분석**

(Cities : 1000, Population : 100, generation : 1000, mutationRate : 0.05, tournament\_size=25, cross : 2분할)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| elitism | 0 | 1 | 2 | 10 |
| 시작 cost | 50872 | 50482 | 49944 | 50040 |
| 결과 cost | 47774 | 44253 | 44270 | 43511 |
| 감소 비율 | 0.9391 | 0.8766 | 0.8864 | 0.8695 |

# 2.2 Typical Algorithms

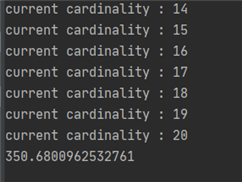
본 목차에선 1000-TSP에 대해 최적의 해결책을 도출하는 (트리로 표현 가능한) 일반적인 탐색 알고리즘을 적용하며, 결과에 관해서도 기술한다.

# 2.2.1 Dynamic Programming

첫째로, 동적 계획법(Dynamic programming)을 사용하여 TSP 문제에서 최적의 답을 찾고, 유전 알고리즘과 비교하고자 했다.

|  |
| --- |
| def tsp\_dist(self):  #base case  for m in range(2,len(self.G.keys())+1):  self.prev\_A=self.curr\_A  self.prev\_A[(1,0)]=0  self.curr\_A={}  print('current cardinality :',m)  for S in self.setS(m):  for j in self.G.keys():  temp=float("inf")  if 1<<j & S:  #A[s,j]=min(A[s-j,k]+dist\_kj  for k in self.G.keys():  if (1<<k & S) and (k!=j):  S\_j=S^(1<<j)  temp=min(self.prev\_A.get((S\_j,k),float("inf")) + self.dist[(k,j)],temp)  self.curr\_A[S,j]=temp  #last hop  S=2\*\*(len(self.G.keys()))-1  for j in range(1,len(self.G.keys())):  self.min\_dist=min(self.curr\_A[S,j]+self.dist[(j,0)],self.min\_dist)  print(self.min\_dist) |

탐색 中에서 최단 경로를 부분적으로 집합에 저장한 후, 저장된 집합을 재사용한다. 그렇기에 완전 탐색(brute-force)보다 효율이 높고, 최적(Optimal)이다. 그러나 노드의 개수가 10개 이하일 때만 높은 성능을 보였을 뿐, 30개 이상의 노드를 가진 문제는 시간복잡도 O(N^2\*M) 상 결과가 출력되지 않았다.



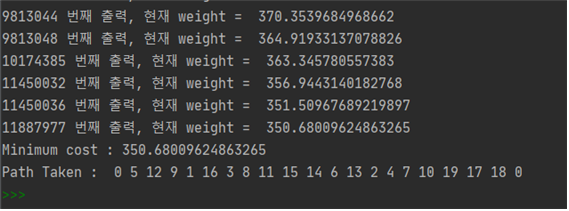
\* 노드를 20개로 설정한 후 출력한 값

# 2.2.2 Branch-and-Bound

두 번째로, 분기한정법(branch-and-bound)을 활용하여 이상적인 최솟값인 lower bound를 설정했고, 가지치기를 반복하는 코드를 작성했다.

|  |
| --- |
| def TSP(adj):  curr\_bound = 0  curr\_path = [-1] \* (N + 1)  visited = [False] \* N  for i in range(N):  curr\_bound += (firstMin(adj, i) +  secondMin(adj, i))  curr\_bound = math.ceil(curr\_bound / 2)  visited[0] = True  curr\_path[0] = 0  TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited) |

본 코드는 bound가 가장 낮은 가지부터 탐색하다가, 상위 노드의 lower bound보다 큰 가지를 발견하면 탐색하지 않는 방식으로 Search를 시도한다. 이는 불필요한 시간 소모를 줄이게 하지만, 최적 보장을 위한 시간 복잡도가 O(2^N)로 크며, 따라서 노드의 개수를 20개 이상으로 늘릴 수 없었다.



\* 노드를 20개로 설정한 후 출력한 값

# 2.3 Representative Algorithm of TSP (Greedy Search & 2-opt)

위의 서술한 알고리즘 외에 TSP 문제를 해결할 수 있는 대표적인 알고리즘을 적용해보았다.

탐욕 알고리즘은 최적인 해를 찾는 데에 사용되는 근사적 방법으로, 결정을 내리는 순간마다 최적이라고 생각되는 것을 선택하는 방식으로 진행하여 최종적인 해답에 도달한다.

한편, 2-opt 알고리즘은 외판원 문제를 해결하기 위해 1958년 Croes가 제안한 간단한 지역 탐색(Local Search) 알고리즘으로, 꼬인 두 간선을 풀어주는 방식을 취하여 최종 해답에 도달한다.

# 2.3.1 Greedy Search Algorithm

Greedy Search는 다음 단계로 구현할 수 있다.

1. 임의의 도시에서 시작함
2. 방문하지 않은 가장 가까운 도시로 이동함
3. 모든 도시를 방문할 때까지 (2)를 반복하고 경로의 길이를 저장
4. 다른 초기 도시로 (1)에서 다시 시작함
5. 모든 가능성이 사라질 때까지 (1-4)를 반복하고 최단 경로로 돌아감

(후에 사용할 Github 알고리즘 기준으로) 시간복잡도는 대략 O(N^3)이다. (N=도시 개수)

# 2.3.2 2-opt Algorithm

2-opt는 다음 단계로 구현할 수 있다.

1. 무작위 경로로 시작함
2. 두 Edge를 교체함
3. 교체한 두 Edge가 더 짧은 경우, 해당 경로를 취함
4. 가능한 모든 경우에 대한 (2)~(3)을 반복함
5. M 개의 가능한 초기 구성에 대해 반복적으로 (1)~(5) 과정을 반복함

(후에 사용할 Github 알고리즘 기준으로) 시간복잡도는 대략 O(M\*N^2)이다. (N=도시 개수)

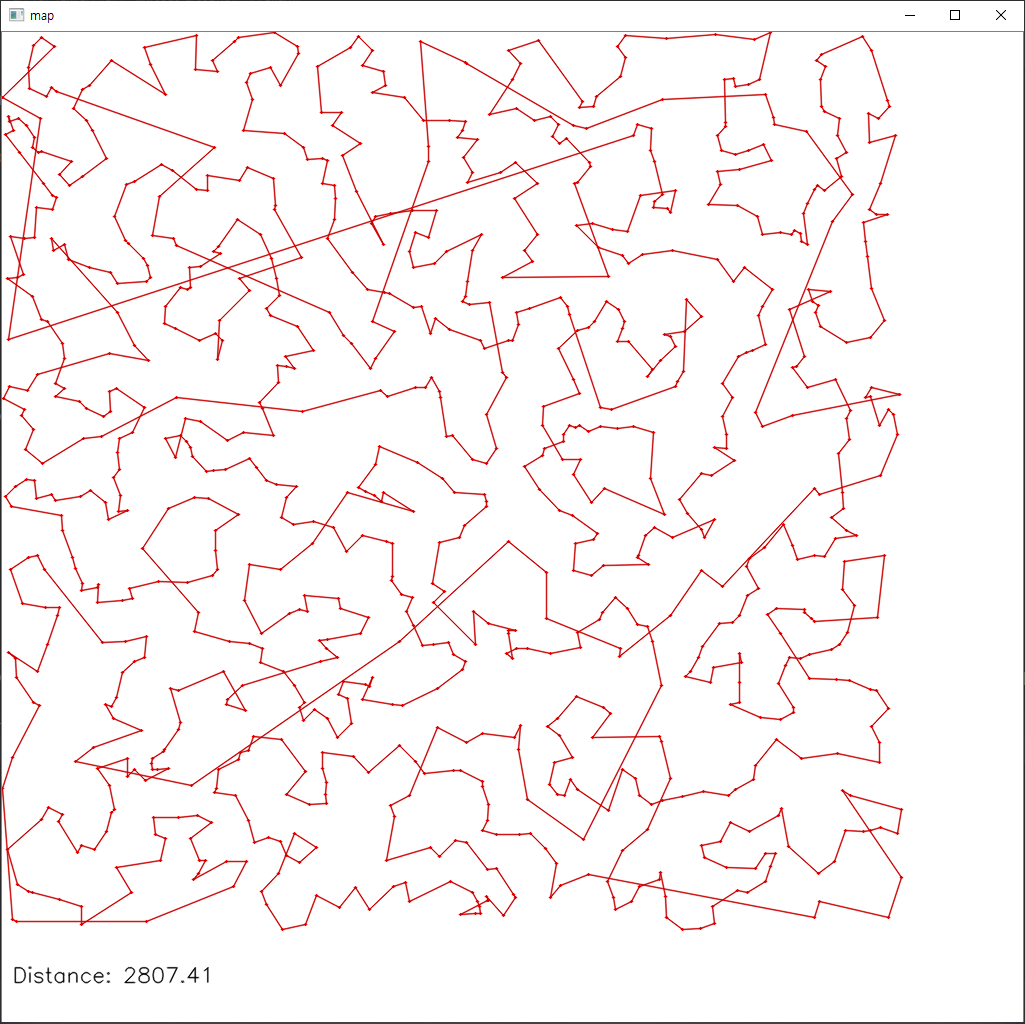
# 3 Experimental Results

## 2.3.3.1 Test Environment

* Anaconda 2019.10, (Python 2.7)
* Windows 10(x64)
* Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz, 2.81 GHz
* NVIDIA GeForce GTX 1060 6GB
* 16GB RAM

## 2.3.3.2 Result (Greedy Search)

Github 코드[[1]](#footnote-1)를 참조하여 Greedy Search 알고리즘으로 1000-TSP 문제를 실행한 결과 다음과 같은 경로를 구할 수 있었다.



경로 최종 길이 : 2807.41

탐색 시간 : 4787.46 seconds

탐색 경로 :

|  |
| --- |
| [964, 512, 926, 341, 970, 555, 676, 776, 204, 39, 369, 384, 682, 984, 678, 48, 722, 186, 613, 176, 248, 321, 998, 340, 44, 163, 536, 42, 668, 385, 232, 287, 41, 211, 695, 714, 835, 660, 247, 643, 392, 567, 174, 763, 252, 257, 178, 527, 223, 710, 742, 539, 217, 632, 281, 272, 10, 648, 893, 989, 586, 707, 418, 851, 417, 103, 627, 303, 925, 973, 561, 594, 449, 690, 188, 19, 238, 554, 17, 638, 772, 540, 829, 773, 464, 796, 440, 125, 734, 708, 137, 344, 821, 741, 205, 66, 793, 335, 675, 146, 435, 356, 147, 657, 916, 610, 593, 696, 488, 81, 78, 993, 751, 836, 218, 58, 482, 144, 994, 520, 941, 764, 816, 155, 865, 100, 166, 114, 262, 713, 135, 420, 336, 9, 422, 91, 241, 40, 661, 814, 504, 895, 77, 86, 559, 845, 126, 471, 229, 608, 334, 838, 971, 351, 338, 84, 452, 466, 428, 425, 268, 830, 353, 721, 493, 359, 140, 769, 212, 639, 584, 743, 111, 873, 937, 902, 802, 933, 755, 270, 315, 533, 69, 615, 570, 339, 202, 421, 427, 666, 558, 924, 47, 251, 815, 896, 588, 316, 297, 899, 831, 543, 85, 430, 477, 906, 187, 371, 383, 614, 105, 26, 264, 129, 580, 854, 850, 537, 565, 391, 549, 963, 206, 481, 794, 762, 152, 915, 959, 654, 552, 372, 254, 386, 149, 758, 852, 462, 930, 944, 702, 193, 628, 389, 280, 228, 302, 362, 154, 189, 746, 777, 442, 663, 803, 704, 349, 808, 275, 250, 128, 23, 45, 106, 113, 57, 826, 124, 190, 312, 492, 955, 208, 429, 976, 387, 500, 686, 310, 255, 582, 912, 123, 328, 55, 967, 164, 51, 122, 669, 863, 65, 744, 541, 546, 712, 393, 780, 192, 416, 368, 740, 600, 130, 94, 278, 877, 291, 988, 28, 602, 768, 898, 585, 995, 16, 757, 911, 674, 571, 355, 611, 607, 363, 597, 394, 157, 577, 949, 216, 913, 209, 191, 282, 945, 260, 807, 465, 523, 521, 410, 756, 381, 279, 33, 604, 526, 692, 817, 501, 266, 801, 379, 1, 450, 631, 770, 313, 856, 227, 475, 900, 408, 325, 789, 622, 197, 629, 307, 304, 560, 982, 461, 409, 738, 919, 889, 731, 503, 380, 601, 691, 799, 748, 655, 210, 646, 134, 448, 511, 986, 701, 667, 716, 891, 161, 810, 882, 22, 107, 531, 446, 636, 929, 120, 237, 469, 849, 940, 679, 0, 318, 438, 529, 5, 999, 361, 290, 494, 784, 684, 800, 922, 346, 104, 939, 881, 12, 276, 749, 618, 365, 491, 74, 460, 29, 703, 841, 670, 778, 80, 987, 975, 474, 550, 798, 735, 727, 118, 437, 958, 273, 508, 952, 765, 979, 53, 785, 151, 443, 847, 960, 745, 177, 93, 333, 298, 413, 633, 548, 534, 62, 117, 842, 578, 214, 991, 688, 872, 165, 75, 903, 18, 505, 295, 519, 249, 787, 683, 728, 985, 603, 687, 198, 215, 305, 766, 288, 650, 996, 969, 880, 599, 553, 259, 244, 173, 591, 463, 705, 411, 557, 904, 473, 412, 846, 532, 472, 299, 640, 514, 405, 407, 317, 400, 935, 109, 136, 806, 883, 839, 243, 89, 809, 509, 366, 820, 99, 59, 200, 583, 32, 71, 162, 423, 180, 936, 625, 24, 236, 649, 681, 63, 36, 795, 551, 453, 234, 436, 138, 102, 485, 441, 885, 326, 314, 292, 240, 587, 378, 459, 977, 910, 175, 858, 871, 34, 374, 870, 145, 133, 637, 116, 235, 253, 119, 954, 169, 203, 239, 990, 859, 867, 490, 256, 457, 458, 21, 31, 7, 917, 524, 635, 30, 869, 54, 612, 95, 729, 88, 49, 908, 733, 329, 630, 834, 390, 267, 909, 907, 720, 901, 434, 414, 544, 516, 760, 56, 689, 957, 294, 920, 286, 730, 2, 569, 285, 184, 868, 980, 406, 342, 956, 141, 840, 968, 812, 222, 398, 142, 782, 767, 844, 354, 382, 201, 76, 196, 499, 753, 624, 96, 97, 358, 572, 61, 150, 495, 771, 357, 672, 673, 207, 225, 918, 131, 717, 306, 581, 127, 887, 73, 52, 888, 886, 323, 352, 404, 723, 277, 258, 284, 327, 4, 562, 343, 700, 245, 271, 823, 950, 13, 439, 468, 737, 879, 617, 658, 866, 828, 367, 921, 890, 848, 590, 293, 476, 525, 350, 563, 8, 518, 331, 965, 415, 857, 11, 289, 566, 783, 220, 347, 538, 467, 932, 680, 938, 719, 864, 265, 641, 759, 855, 399, 15, 732, 974, 195, 388, 715, 320, 875, 697, 224, 445, 517, 486, 694, 480, 168, 444, 931, 484, 171, 606, 337, 332, 402, 50, 231, 230, 170, 269, 6, 515, 301, 645, 14, 665, 726, 664, 671, 112, 547, 897, 160, 121, 659, 348, 143, 575, 426, 923, 487, 598, 804, 43, 824, 261, 981, 942, 653, 108, 822, 156, 545, 132, 79, 308, 3, 226, 324, 609, 115, 953, 983, 213, 779, 792, 373, 978, 775, 319, 37, 322, 576, 644, 158, 626, 70, 620, 662, 311, 568, 592, 221, 805, 972, 709, 774, 483, 159, 510, 616, 832, 496, 786, 992, 370, 489, 542, 419, 46, 621, 455, 623, 837, 651, 818, 642, 183, 811, 619, 574, 60, 813, 685, 83, 819, 884, 153, 634, 397, 966, 167, 433, 401, 997, 962, 914, 699, 376, 360, 456, 528, 927, 182, 98, 447, 300, 64, 677, 139, 20, 194, 946, 68, 454, 750, 825, 263, 579, 878, 513, 652, 535, 396, 718, 345, 179, 479, 27, 961, 502, 25, 736, 246, 377, 706, 791, 781, 754, 110, 876, 522, 797, 219, 330, 199, 752, 82, 724, 860, 948, 843, 90, 951, 296, 72, 309, 101, 233, 185, 432, 87, 827, 596, 739, 589, 242, 283, 790, 35, 874, 833, 67, 395, 573, 747, 861, 507, 943, 928, 905, 424, 530, 498, 403, 788, 725, 478, 892, 693, 274, 375, 647, 656, 431, 92, 711, 497, 181, 862, 506, 947, 364, 605, 38, 934, 761, 470, 698, 451, 853, 894, 595, 148, 556, 172, 564] |

## 2.3.3.3 Result (2-opt)

탐욕법과 동일한 Github를 참조하여 2-opt Algorithm을 통해 1000-TSP 문제를 풀어보려 했으나, 15시간 동안 (2.3.3.1)의 조건으로 돌려봐도 구할 수 없었다. 이유를 다음과 같이 추론해볼 수 있었다.

|  |
| --- |
| * 많은 도시 수로 인한 길이 비교 시간의 오래 걸림 (꼬인 선 풀기 시간이 오래 걸림) * Github에 올라온 해당 알고리즘의 기능적 한계 존재 |

그러나 도시의 수를 100개, 200개로 줄여서 실험해본 결과, 각각의 실행시간이 10초, 100초 정도로 나왔다. 아마도 실행에 많은 시간을 투자했다면 본 알고리즘으로 1000-TSP 문제를 해결할 수 있었을 거라 생각한다.

# 3 Discussion (Comparison)

유전 알고리즘을 바탕으로 1000-TSP 문제를 풀었을 때 나온 결과(2.1.3.2)를 분석해보았다.

* crossover를 3가지 방식으로 나누어보았을 때는 큰 차이가 나타나지 않았으며, mutate 비율을 2.5%로 설정하였을 때가 약 81%로 가장 큰 감소율을 보였다.
* Generation 수를 늘릴 수록 점점 값이 감소하는 특징을 보이나 이와 동시에 값이 굳어지기(고착화)에, 10000까지 작성했다.
* Population 개수를 늘릴 수록 결과는 감소하였지만 큰 차이를 보이지는 않았다.
* tournament\_size는 population에서 선택하는 개체의 개수로, population에 대한 3/4의 비율일 때 가장 높은 감소율을 보였다.
* 마지막으로 elitism의 개수를 조절하여 낸 결과표에 의하면, elitism이 10일 때 가장 효과적이었으나 유의미한 차이를 보이지 않았다.

따라서 단순 유전 알고리즘으로는 cost가 감소하는 데 기하학적인 시간이 필요할 것으로 판단되어, 2.3.3의 두 결과를 부모 유전자로 활용하여 유전 알고리즘을 새로이 돌려보려 했다. 그러나 2-opt 알고리즘의 결과 도출 실패에 따라 해볼 수 없던 점이 아쉬웠다.

조금 더 몇 가지 추론을 해보자면, Greedy Search로 구한 Cost가 2807 정도임을 알 수 있다. 또한, 휴리스틱으로 최소 cost(lower bound)가 1583이므로 탐욕법과 다른 알고리즘을 기반으로 유전 알고리즘을 돌린다면 휴리스틱 결괏값에 도달(Global Optimal)할 수 있을 것으로 예상한다.

한편, 본 문서를 작성하기 위해 “A Powerful Genetic Algorithm for Traveling Salesman Problem”[[2]](#footnote-2) 논문 또한 참고했다. 해당 논문은 ES(Edge-Swapping) 알고리즘을 사용했고, 1000-TSP 문제를 대입한 결과, 2284의 Cost를 얻을 수 있었다. 유전 알고리즘의 경우 Cost와 상관없이 사용자가 목표로 한 세대까지 탐색해 나가는 반면, 논문에서는 세대 수를 정하지 않고 한 세대마다 탐색한 Cost 중에서 Best Cost와 Average Cost를 비교한다. 만약 둘의 차가 0.001 이하면, 탐색을 종료하는 방식을 사용하여 기존 유전 알고리즘보다 훨씬 빠르게 탐색이 가능하다. 실제로 프로젝트를 진행하며 사용한 코드는 기본적으로 1시간 이상 걸렸는데, 논문의 코드 같은 경우 한 번 탐색할 때마다 약 1분 30초의 시간이 소요됐다. 이러한 접근 방식을 프로젝트에서 사용한 코드에 적용했다면, 시간적으로 더 효율적인 코드가 되지 않았을까 생각한다.

유전 알고리즘 논문을 활용해 얻은 최소 탐색 경로[[3]](#footnote-3) :

|  |
| --- |
| 1 319 439 530 6 1000 362 291 495 785 923 801 685 347 136 421 714 263 115 605 598 395 158 527 693 818 502 267 380 2 802 935 41 662 815 505 896 78 242 92 423 337 10 105 940 882 277 13 704 30 842 671 648 657 976 779 81 988 414 475 551 736 799 728 119 959 438 54 786 152 444 965 513 726 927 342 971 385 777 677 556 370 205 40 249 177 187 614 683 985 679 49 723 980 735 709 138 345 742 822 126 441 797 465 774 830 541 773 419 708 587 990 894 649 282 273 11 239 20 18 555 639 852 418 104 628 304 926 974 697 594 562 595 450 691 644 248 661 715 836 696 212 42 929 944 508 552 454 64 37 796 43 669 386 233 288 658 917 611 67 206 794 336 676 147 436 357 148 322 341 999 45 537 164 682 237 650 25 937 626 181 424 163 72 33 584 201 60 906 413 474 905 558 412 592 706 464 174 245 260 554 600 881 970 997 651 289 216 767 306 199 688 604 986 729 788 684 250 520 296 789 404 499 531 406 515 641 300 473 533 847 425 367 821 100 510 90 810 840 244 884 807 137 110 936 318 408 401 506 19 904 76 166 873 689 992 215 579 118 843 893 479 848 961 746 178 94 334 299 634 549 63 535 694 275 376 461 75 492 366 619 750 87 560 846 127 472 839 230 609 335 339 352 972 85 453 467 149 596 895 854 452 557 874 744 112 938 903 803 141 770 213 640 585 173 699 471 762 429 153 426 269 831 360 354 722 494 934 756 271 316 534 571 616 70 983 561 305 308 630 198 623 462 410 739 920 544 832 900 298 647 211 317 589 816 897 252 48 925 559 667 340 203 422 428 565 890 732 504 381 692 602 800 749 656 566 392 550 207 964 538 851 855 482 795 581 130 265 27 615 106 384 372 188 907 478 431 86 135 757 411 522 382 524 466 808 261 790 326 409 901 476 228 857 314 771 632 451 39 606 365 948 946 192 283 210 914 950 578 217 34 280 364 608 612 356 572 675 758 17 912 996 586 899 769 449 603 29 989 292 878 131 601 95 279 741 369 417 193 781 394 713 547 542 745 66 864 670 123 663 52 165 968 56 329 124 913 583 256 311 227 610 325 309 80 546 157 133 186 825 44 805 599 488 924 427 828 576 88 433 262 982 943 654 823 109 4 209 956 430 977 388 501 687 310 511 617 160 484 710 806 973 775 312 569 593 222 190 747 778 443 664 371 497 833 787 993 107 46 24 125 114 827 58 191 313 493 672 351 526 477 665 666 727 564 9 519 251 129 276 809 804 705 350 784 567 332 966 416 858 12 290 733 16 400 975 856 760 642 266 865 720 939 681 933 468 348 539 221 47 420 543 490 622 155 363 303 853 463 931 945 703 194 629 390 281 229 759 150 387 255 553 655 373 960 916 763 523 877 111 755 782 792 707 378 247 26 503 962 737 28 480 180 346 915 963 998 700 377 361 457 529 719 397 928 183 99 536 448 301 168 402 434 967 398 635 154 885 297 952 844 861 949 725 83 753 798 220 331 200 624 456 838 652 91 820 84 686 814 620 575 812 184 643 819 61 73 947 69 455 195 21 140 678 65 580 264 751 826 879 514 653 196 389 716 321 876 698 446 225 15 518 487 695 607 172 485 932 445 403 338 333 51 232 231 171 270 169 481 302 646 516 7 829 368 867 922 891 849 294 591 113 548 161 898 122 841 813 223 399 143 783 768 845 969 407 981 343 957 142 660 659 618 880 738 469 440 14 500 754 625 97 98 573 359 62 151 496 772 358 673 674 208 226 919 132 718 307 582 128 888 74 53 889 887 324 353 724 405 824 951 272 246 5 563 328 285 259 278 344 701 545 415 435 902 721 908 910 268 391 835 32 8 22 459 458 257 491 868 860 761 517 57 690 958 295 921 287 202 731 3 77 197 349 869 570 286 185 383 355 144 236 117 638 134 120 254 991 240 204 955 170 146 871 375 35 872 330 734 631 50 909 89 730 525 918 636 96 31 870 55 613 315 327 442 486 284 791 834 36 875 862 748 574 68 396 235 437 139 103 590 243 886 293 241 588 379 460 978 911 176 859 740 597 764 253 175 568 393 189 258 179 528 224 743 711 540 218 633 234 102 780 793 214 984 954 116 71 627 159 645 577 621 323 38 320 776 979 374 23 108 532 447 883 811 162 892 717 668 702 987 512 498 712 182 863 507 167 101 156 866 432 93 817 765 521 942 995 145 483 930 637 219 59 837 752 994 489 82 79 953 274 509 766 121 238 850 470 941 680 |

# 4 Conclusion

트리구조를 이용하는 분기한정법이나 동적 계획법을 1000-TSP 문제에 적용하기에는 시간적 한계가 있었다. 아마 해당 알고리즘들은 해 공간에 부분적으로 트리구조를 적용했을 경우 효과적일 것으로 예상한다. 또한, 많은 시간이 주어진다면, 탐욕법과 2-opt 알고리즘을 활용하여 1000-TSP 문제를 해결할 수 있다는 것을 실험/예상해볼 수 있었다.

한편, 유전 알고리즘에서 결과를 향상하기 위해 총 6가지(Crossover 방식, Mutation 확률, 토너먼트 크기, elitismOffset 크기, Population 크기, Generation 수)의 변화를 주며 프로젝트를 진행했다. 직접 변수 조건에 맞게 코드를 수정하고 실행 과정을 거친 대부분의 요인(mutation=0.025, tournament\_size=population\*¾, population 증가, generation 증가 일 때 최솟값을 가진다.)들은 결괏값에 영향을 미쳤다. 그러나 Crossover 방식과 elitismOffset의 크기는 결괏값에 거의 영향을 미치지 못했다. 아마도 Crossover 방식에서 선택한 다점 교차의 점 개수는 결과에 큰 영향을 주지 않는 것으로 보인다. 또한 elitismOffset도 값이 0일 때를 제외하고 1~10일 때에는 결괏값에 영향을 주지 못했다. 이를 바탕으로 Population의 개수 대비 elitismOffset의 크기를 너무 작게 설정한 부분이 문제인 것으로 생각한다.

즉, 본 주제(1000-TSP)를 해결하고 연구함으로써 다양한 알고리즘을 사용해본 것이 의미 있게 다가왔다. 단순한 유전 알고리즘만을 사용하여 해당 문제를 풀기에는 시간적인 한계가 존재함을 알 수 있었고, 다양한 알고리즘 활용은 문제해결의 실마리를 제공해준다는 것을 알았다.

# 5 Reference

* <https://gist.github.com/turbofart/3428880> (유전 알고리즘)
* <https://gist.github.com/kavyakalidindi/5eddebc2f7c3f03ca921582ede3760da>(동적 계획법 알고리즘)
* <https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/>(분기한정법 알고리즘)
* <https://github.com/rohanp/travelingSalesman> (탐욕법과 2-opt 알고리즘)
* <https://www.wikipedia.org/> (각 알고리즘 별 개념 정의)
* Shujia Liu.(2014). A Powerful Genetic Algorithm for Traveling Salesman Problem. arXiv:1402.4699v1 [cs.NE], 19, Feb, 2014. (<https://arxiv.org/pdf/1402.4699v1.pdf>)

1. <https://github.com/rohanp/travelingSalesman> [↑](#footnote-ref-1)
2. <https://arxiv.org/pdf/1402.4699v1.pdf> [↑](#footnote-ref-2)
3. 시작 노드가 1번, 끝 노드가 1000일 때를 가정함. [↑](#footnote-ref-3)