2022 겨울학기 연구참여 보고서

: Python을 이용한 TSP 프로그램 구현

2022.01.03. ~ 02.11. (6주)

Logistics Lab.

지도교수 김병인

멘토 권오현

참여학생 김상원

작성일자: 2022.02.11

목차

1. 서론

1.1. 연구 배경

Traveling Salesman Problem(TSP, 외판원 문제)는 방문해야 할 여러 도시가 주어졌을 때, 출발 지점으로부터 시작하여 각 도시를 한 번씩만 방문하여 다시 출발점으로 돌아오는 경로를 찾는 문제로, 가장 길이가 적은 경로를 찾아내는 것이 목적이다. 이 문제는 NP-hard 문제로, 다항 시간에 문제를 해결하는 알고리즘이 존재하지 않는다. 생각할 수 있는 가장 단순한 방법은 존재할 수 있는 모든 순열을 구한 후, 가장 짧은 경로를 찾아내는 것이다. 그러나 이 방법은 복잡도가 O(n!)에 달하기 때문에 경우의 수가 지나치게 많아지므로 현실성이 없는 방법이다. 그렇기 때문에 도시의 수가 많아지면 정확한 해를 찾는 것이 불가능해지므로, 다른 알고리즘을 통해 근사적인 해를 구할 필요가 있다. 이번 연구참여의 목표는 TSP의 해를 구할 수 있는 여러 종류의 알고리즘을 Python을 사용하여 구현하고, 이를 시각화하는 프로그램을 만드는 것이다.

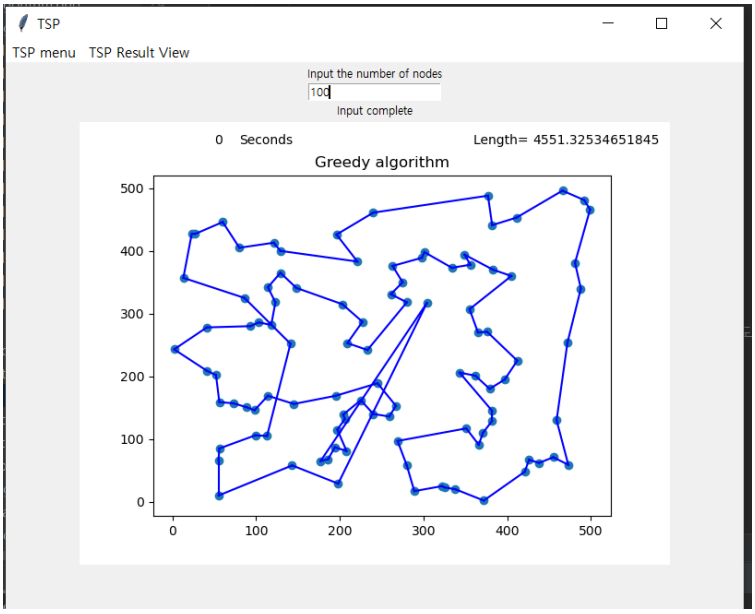
2. 연구 방법.

2.1. 알고리즘의 종류

연구에서 구현한 것은 Greedy algorithm, (Greedy) 2-opt, (Greedy) 3-opt, Genetic algorithm, Simulated annealing 이 있다.

2.1.1. Greedy algorithm

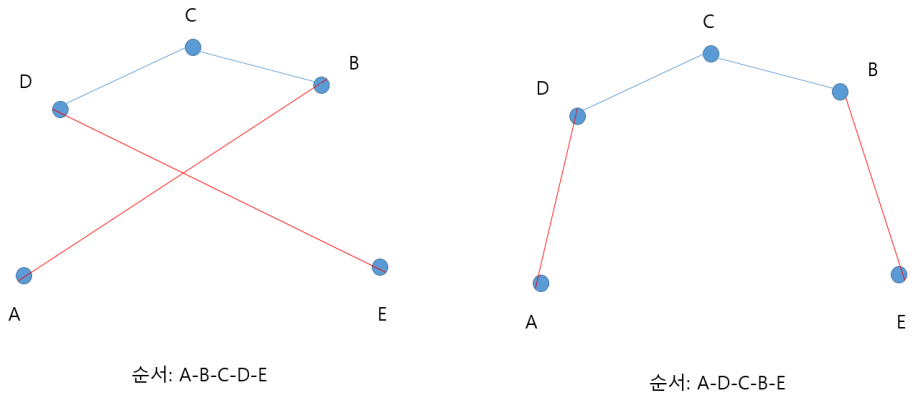
Greedy algorithm은 먼저 출발점에서 가장 가까운 노드를 찾아, 그곳을 다음 지점으로 정한다. 그 후 그 지점에서 이미 지났던 노드는 제외하고, 가장 가까운 노드를 다시 찾아 다음 지점으로 만든다. 이런 식으로 계속해서 제일 가까운 노드를 이어간 후 마지막에는 출발점으로 돌아오도록 하는 알고리즘이다. 구현이 간단하며, 계산 과정에서 경로의 변화가 없으므로 해를 찾는 데 시간이 적게 걸리는 장점이 있으나, 노드의 수가 많아짐에 따라 좋은 해를 구하기가 어려워진다.



[그림 1] Greedy algorithm 실행 결과

2.1.2. 2-opt algorithm

2-opt algorithm은 2개의 edge가 이어진 방식을 변경했을 때, 원래의 경로보다 거리가 짧아질 경우 그 변경된 경로를 택하는 방식이다. 자세한 구현 방법은 다음 그림과 같다.



[그림 2] 경로 역전을 통한 경로 변경

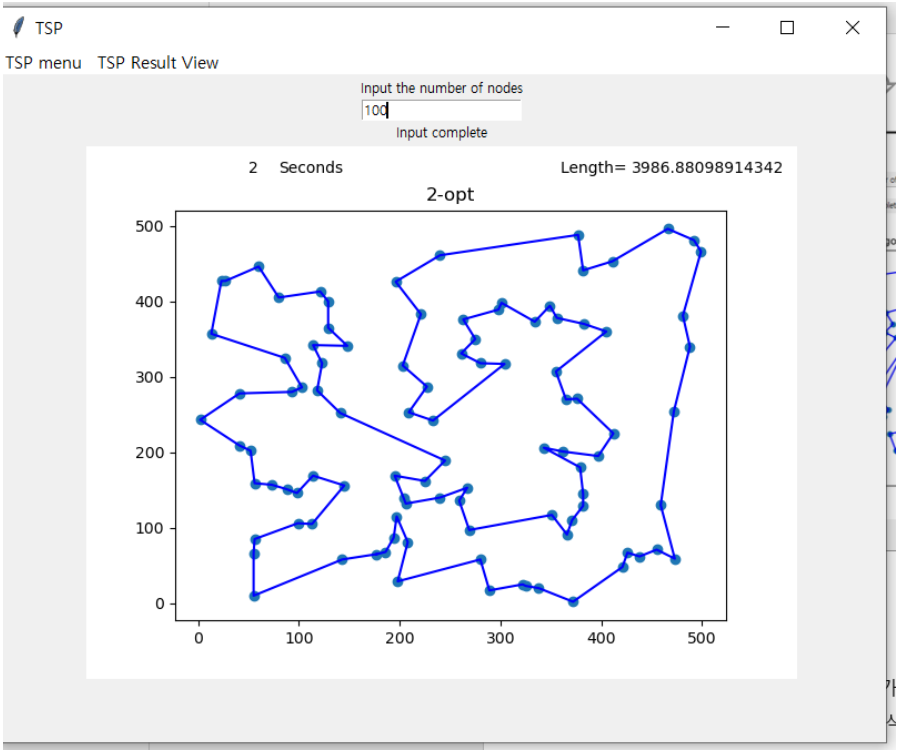
위 사진은 부분적인 경로의 모습으로, 원래 순서는 A-**B-C-D**-E 이다. 임의의 점 A와 E가 선택되었다고 할 때, A와 E 사이의 노드들의 경로 순서를 역전시키면 순서는 A-**D-C-B**-E가 된다. 이 때, 순서를 역전시킴으로써 위의 그림처럼 A-B, D-E의 경로가 서로 교차하던 것이 해결되면서 총 경로의 길이가 짧아지는 효과가 생겼다. 이렇게 모든 두 노드의 조합에서 그 사이의 경로를 역전시켰을 때 원래보다 경로가 짧아지는 경우에만 바꿔주고, 그렇지 않으면 원래대로 둔다. 자세한 알고

리즘의 방식을 정리하면 다음과 같다.

Step 1: Greedy algorithm에서 구했던 경로를 초기 해로 정한다.

Step 2: 이중 for문을 통해 모든 두 점의 조합에 대해 위의 과정을 거친다. 경로를 역전시켜 짧아졌을 경우, 그 경로로 변경한다. For문을 전부 끝냈을 때의 해를 저장한다.

Step 3: 다시 Step 2를 진행한다. Step 2가 마무리되었을 때 이전에 저장한 해와 동일할 경우(즉, 개선이 더 이상 안 될 경우), 이를 최적의 해로 정하고, 알고리즘을 마친다. 완료 시점에서 edge가 교차되는 모든 지점이 사라지게 된다.



[그림 3] 2-opt algorithm 실행 결과

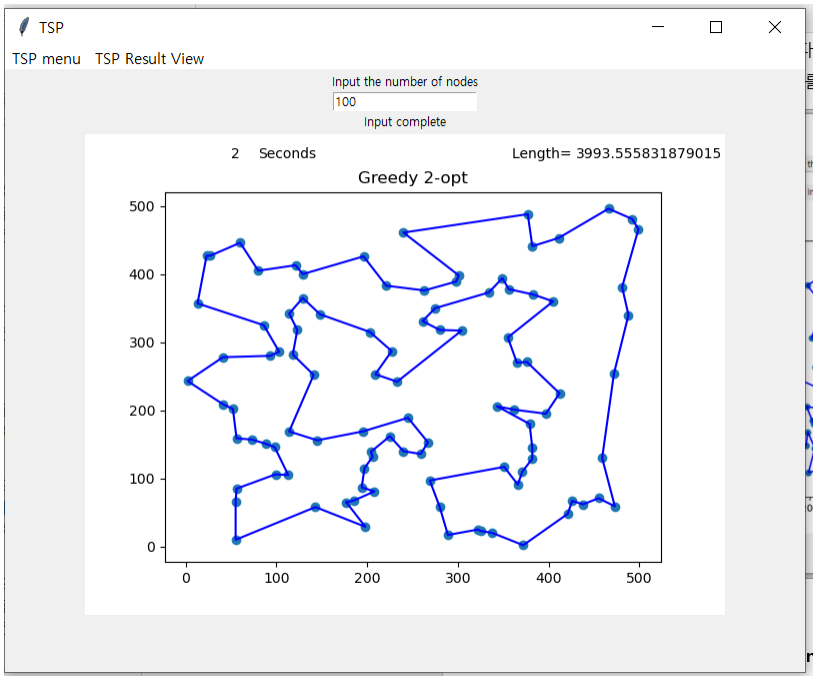
2.1.3. Greedy 2-opt algorithm

Greedy 2-opt algorithm은 위의 2-opt 알고리즘과 비슷하지만 방식이 약간 다르다. 두 점의 조합을 고르고 그 사이의 경로를 역전시키는 것은 동일하지만, 개선이 이루어졌을 경우에 곧바로 Step 2로 돌아간다는 점이 차이점이다.

Step 1: 마찬가지로 Greedy algorithm에서 구했던 경로를 초기 해로 정한다.

Step 2: 위와 동일한 이중 for문을 진행하다가 경로의 개선이 이루어졌을 경우, 경로를 변경하고 해를 저장한 뒤, 즉시 Step 2로 되돌아가 다시 이중 for문을 진행한다.

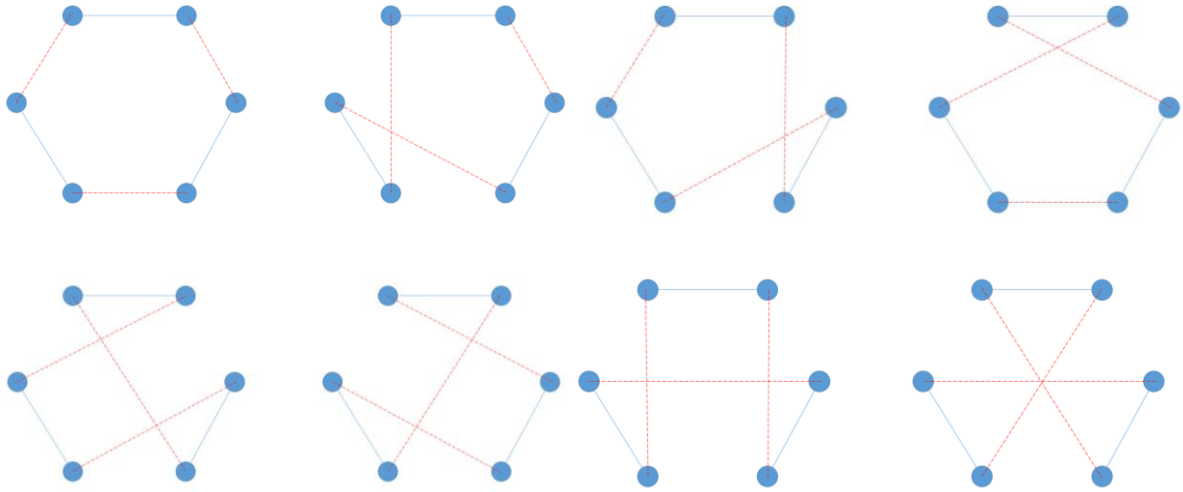
Step 3: 더 이상 경로의 개선이 불가능하면, 이전에 저장한 해를 최적의 해로 정하고 알고리즘을 마친다. 2-opt와 마찬가지로 모든 edge 교차점이 사라진다.



[그림 4] Greedy 2-opt algorithm 실행 결과

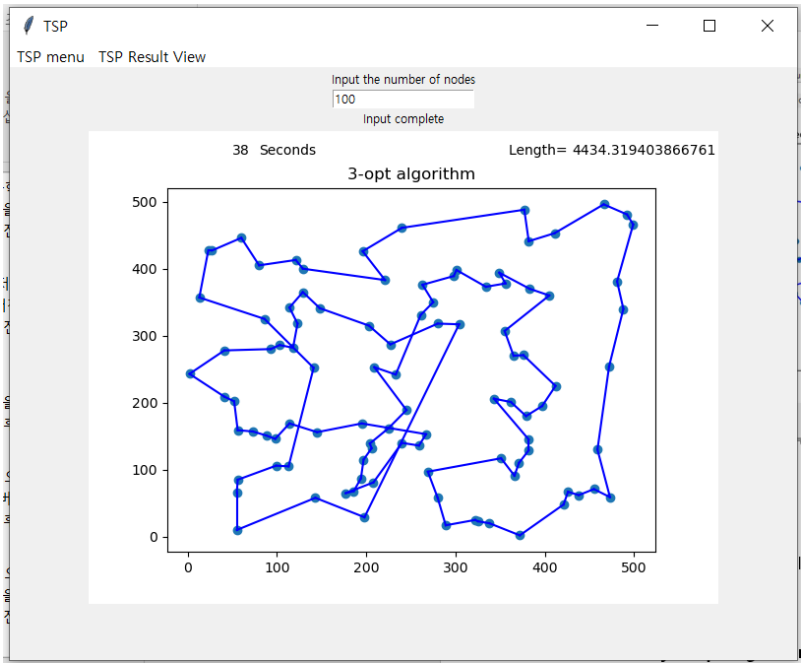
2.1.4. 3-opt algorithm

3-opt algorithm은 위의 2-opt algorithm의 연장선으로, 총 3개의 edge의 연결 방식을 변경했을 때 나오는 8가지 조합에서 각각 경로를 역전시키고, 총 거리를 비교하여 가장 경로 거리가 짧은 해를 구하는 방식이다.



[그림 5] 경로 변경의 경우의 수

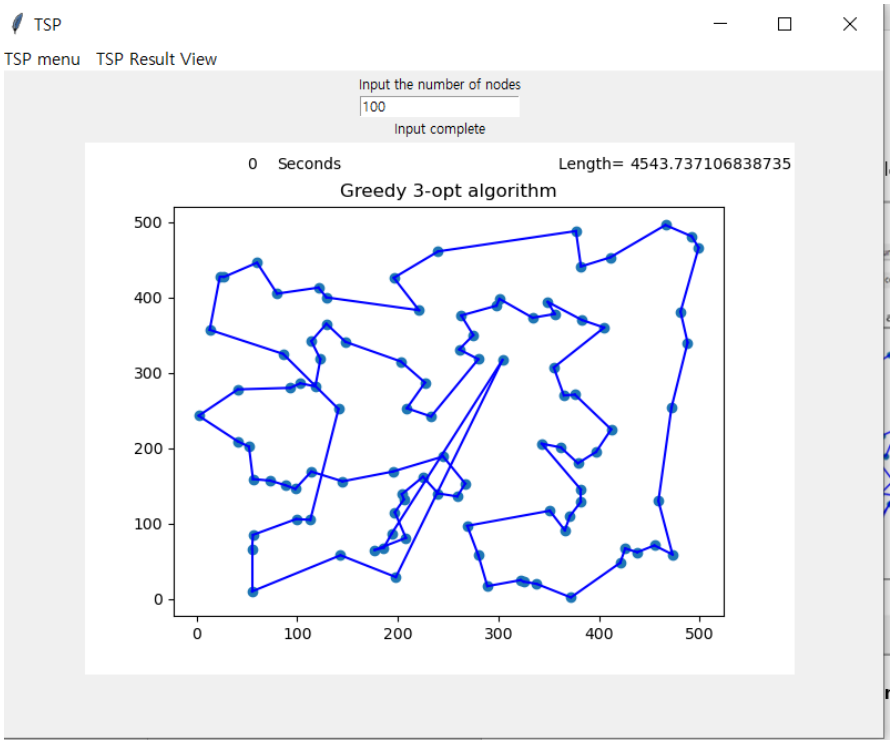
그림에서 첫 번째는 변화가 없을 때, 2~4번째는 2-opt와 마찬가지로 하나의 경로만 역전되었을 때, 5~7번째는 2개의 경로를 역전했을 때, 8번째는 3개의 경로가 역전된 경우이다. 3중 for문을 돌려 모든 조합에 대해 비교하고, 거리가 줄어들었을 경우 경로를 변경해주면 된다.



[그림 6] 3-opt algorithm 실행 결과

2.1.5. Greedy 3-opt algorithm

3-opt와 비슷하지만, 3중 for문에서 경로 변경이 이루어졌을 경우, 해를 저장하고 즉시 3중 for문을 다시 시작한다. 따라서 3-opt에 비해 계산 횟수가 적어지므로 시간이 짧게 걸리지만, 더 안 좋은 해를 얻게 된다.



[그림 7] Greedy 3-opt algorithm 실행 결과

2.1.6. Genetic algorithm

Genetic algorithm은 염색체와 유전자의 특성을 응용, 염색체로부터 새로운 자손이 만들어지는 과정을 모방한 메타휴리스틱 알고리즘이다. 이 방법은 특정 문제에만 국한되지 않고, 원하는 문제에 맞게 적용하여 문제를 해결할 수 있다. 이 연구에서는 TSP에 Genetic algorithm을 적용한 것이다. 구체적인 과정은 다음과 같다.

Step 1: 총 개체 수인 P를 정하고, P개의 무작위 경로 해를 만들어낸다. 이 개체들은 첫 번째 세대에 해당된다.

Step 2: 각 해에 대해 적합도를 계산한다. 여기서 적합도는 경로의 총 거리의 역을 사용하였다. 즉, 적합도가 높은 해가 좋은 해이다.

Step 3: 다음의 과정을 거쳐 다음 세대를 만들어낸다.

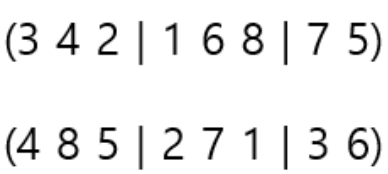
1. 이전 세대의 해 중, 적합도가 가장 높은 0.1P 만큼의 해를 저장한다.

2. 이전 세대의 해로부터, 자손 염색체를 형성할 두 개의 해를 고른다. 해가 선택될 확률은 (특정 해의 적합도/모든 해의 적합도의 합)이 된다. 즉, 적합도가 높은 해일수록 더 높은 확률로 선택된다. 선택된 두 개의 해를 Partial-Mapped Crossover(PMX) 방법으로 교차시켜 두 개의 자손 해를 생성한다. PMX의 방식은 다음과 같다.

먼저, 두 개의 염색체 (3 4 8 2 7 1 6 5)와 (4 2 5 1 6 8 3 7)이 있다고 하자. 무작위로 두 지점을 골라 염색체를 분할한 뒤, 가운데 부분을 서로 교차해준다.

[그림 8] PMX 교차 과정

교차되었을 때, 위쪽은 8과 6이, 아래쪽은 2와 7이 중복된다. 따라서 유전자의 위치를 다시 바꿔야 할 필요가 있다. 이 예시에서 교차된 부분은 (1, 8), (2, 1), (7, 6)인데, 이를 X가 표시된 위치에 넣어서 중복이 안 되는지 확인한다. 위쪽의 염색체에서 (1, 8)은 전부 중복되고, (2, 1)에서 2가 중복이 아니므로 왼쪽 X에 2를 넣는다. 그 다음 (7, 6)에서 7이 중복이 아니므로 오른쪽 X에 7을 넣는다. 마찬가지로 아래쪽 염색체에서는 (1, 8)에서 8이 중복이 아니므로 왼쪽 X에 8이 들어가고, (7, 6)에서 7이 중복이 아니므로 오른쪽 X에 7이 들어간다.

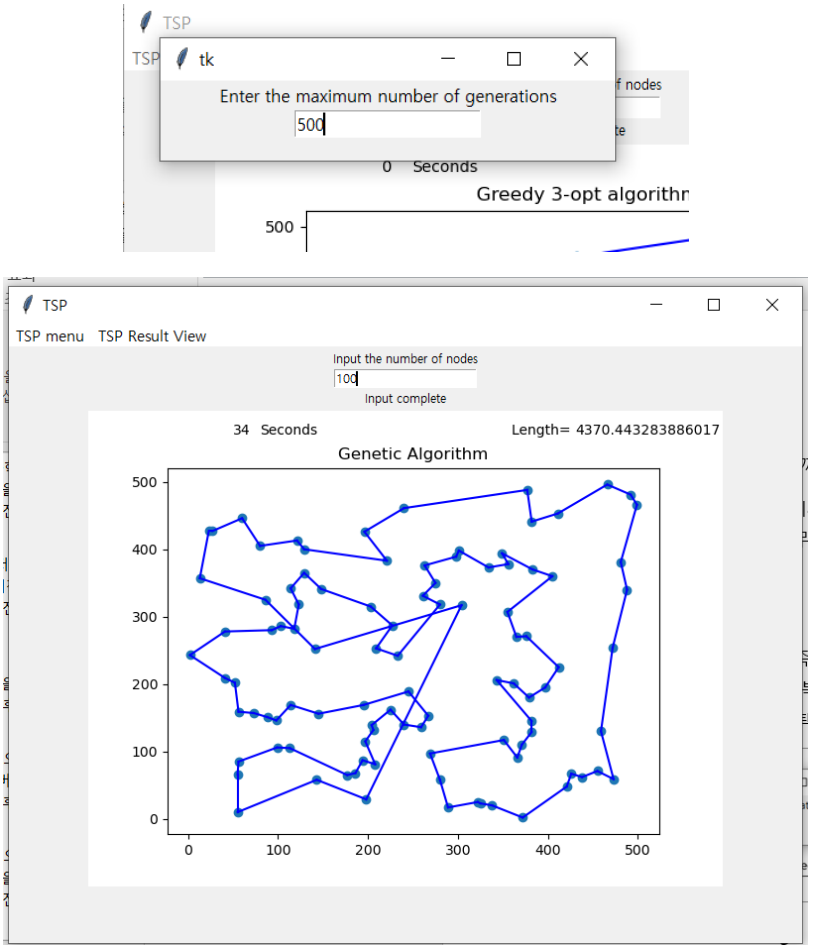


[그림 9] PMX 교차 결과

자손 해가 0.89P개만큼 생길 때까지 이를 반복하고, 생성된 모든 해를 저장한다.

3. 이전 세대의 해 중에서 0.01P개를 무작위로 선택한다. 이후 선택된 해들에 돌연변이를 일으킨다. 돌연변이는 해에서 무작위로 선택된 두 개의 유전자를 바꾸는 것이다. 이렇게 변경된 해들을 저장한다.

Step 4: 0.1P+0.89P+0.01P = P, 즉 다음 세대의 해들이 P개만큼 형성되었다. 나아갈 세대 수를 N이라고 했을 때, Step 3을 N번 반복한다. 마지막으로 형성된 해 중 적합도가 가장 높은(경로 길이가 가장 짧은) 해를 최종적으로 선택한다.



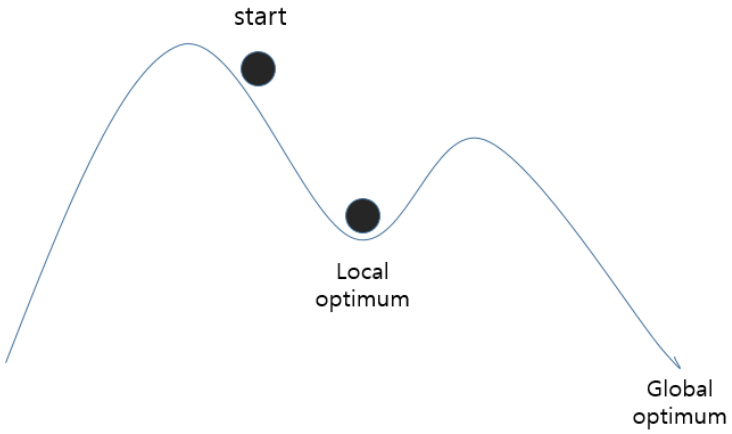
[그림 10] Genetic algorithm 실행 결과

2.1.7. Simulated annealing

Simulated annealing은 함수의 전역 최적점(Global optimum)의 근사치를 찾기 위한 메타휴리스틱 알고리즘이다. 금속재료를 가열했다가 냉각시키는 담금질(annealing)과 비슷하다고 하여 이런 이름이 붙여졌다. 이 알고리즘의 기본적인 원리는 다음과 같다.

1. 현재 해의 근방에 있는 해를 찾아, 더 좋은 지점으로 계속해서 이동한다. 더 이상 이동하지 못하면, 지역 최적점(Local optimum)에 닿은 것이다. 이는 전역 최적점과는 차이가 있기 때문에 좋은 해라고 할 수 없다.

2. 현재 온도에 따라 크게, 혹은 작게 해를 변화시킨다. 해는 온도가 높은 상황에서 변화가 더 커진다. 초기에는 온도가 높아서 해의 변화가 크기 때문에 지역 최적점에서 벗어날 확률이 높다. 시간이 지남에 따라 온도가 낮아지면서 최적점을 벗어날 확률은 점점 낮아지고, 최종적으로는 전역 최적점에 근사한 지점에 도달하게 된다.



[그림 11] Simulated annealing 예시

비유하자면, start 지점에 공을 놓을 경우, 공은 local optimum으로 이동한 후 멈춘다. 더 이상 이동하지 않기 때문에, 일정 세기로 공을 무작위 방향으로 밀어버린다. 왼쪽은 높기 때문에 공이 올라가더라도 다시 local optimum으로 내려올 확률이 높다. 반대로 오른쪽으로 민다면 local optimum을 벗어나 global optimum으로 이동할 가능성이 크다. 공을 미는 세기는 현재 온도에 비례한다. 온도가 점점 낮아지면서 언덕을 넘을 가능성은 줄어들고, 최종적으로는 한 지점에 멈추게 될 것이다.

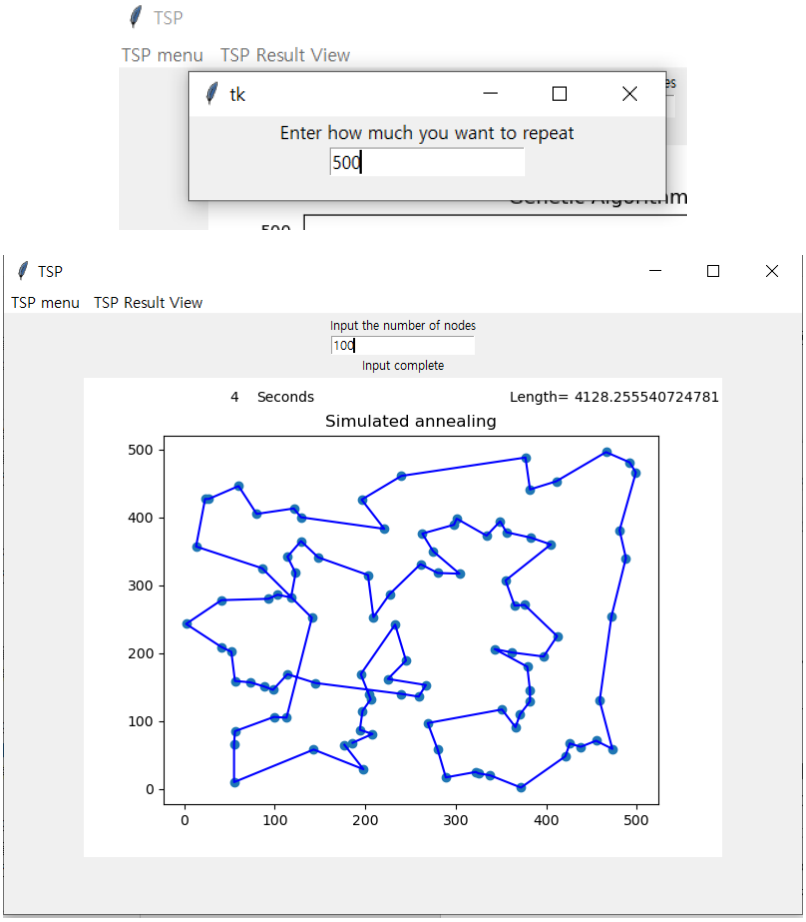
이 알고리즘 역시 TSP에 적용이 가능하다. 알고리즘의 과정은 다음과 같다.

Step 1: 처음 온도 T와, cooling factor를 결정한다. 연구에서는 T=30, cooling factor =0.9로 사용하였다. 초기의 해는 greedy algorithm에서 구한 해를 사용하였다.

Step 2: 초기 해에서, 무작위로 두 개의 지점을 골라 순서를 바꾼다. 변경된 해와 원래 해의 경로 길이를 비교하여, 변경된 해가 더 좋을 경우(경로 길이가 더 짧을 경우) 그 해의 경로 길이를 저장하고, 변경된 해를 초기 해로 지정한다. 그렇지 않을 경우, 아래의 과정으로 넘어간다.

X를 (0, 1) 균등분포에서 무작위로 구해진 난수라고 하자. X >= $e^{-delta/T}$ 일 때, 변경된 해를 다시 원래대로 돌려놓는다. 그렇지 않을 경우, 해를 변경된 상태 그대로 두고 그 해의 경로 길이를 저장하며, 변경된 해를 초기 해로 지정한다. 이후, Step2를 99번 더 반복한다. 결론적으로, 온도가 높을 때 더 많은 변경이 이루어지게 된다.

Step 3: T에 cooling factor를 곱해 온도를 낮추고, Step 2를 다시 실행한다. 온도가 낮아지면서 무작위 변경의 정도는 줄어들게 된다. 사용자가 정한 횟수 동안 Step 3을 반복하고, 최종적으로 나온 해를 최적해로 저장한다.

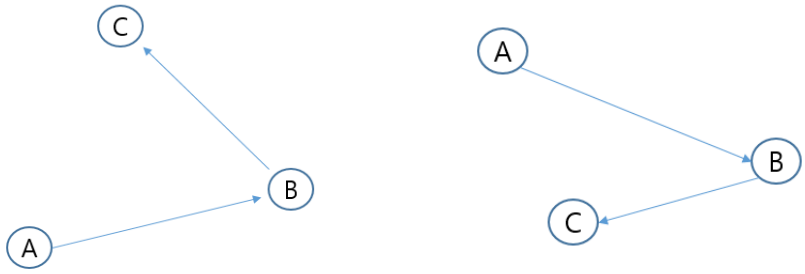


[그림 12] Simulated annealing 실행 결과

2.1.8. Improvement of algorithm

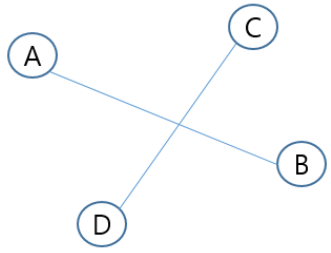
위에서 설명한 알고리즘으로 구한 해를 개선하여 더 좋은 해를 만드는 방법이다. 만들어진 경로에서 edge 2개가 서로 교차되어 있을 경우, 두 edge가 교차되지 않도록 위치를 바꿔주면 총 길이를 줄일 수 있다. 모든 교차점에 대해 이를 실행하여 교차되는 부분이 없는 closed-loop가 만들어지도록 한다. 예를 들어 i+1 < j이고, (i ~ i+1), (j ~ j+1)의 edge가 교차되었을 경우, i+1에서 j까지의 노드 순서를 역전시키면 교차점이 사라진다. 순서를 역전시킨다는 점에서 위에서 설명한 2-opt algorithm과 비슷하지만, 2-opt는 노드 순서를 역전시켰을 때 경로 길이가 줄어드는지를 확인한 후 바꾸는 것인 반면, 이 과정은 기하학적으로 교차 여부를 판단 후 교차 지점을 풀어주는 것이므로 원리가 다르다고 할 수 있다.

두 edge가 교차되는지를 알아보기 위한 방법으로 CCW 알고리즘을 이용하였다.



[그림 13] CCW 알고리즘 예시 1

A->B 벡터를 x, B->C 벡터를 y라고 하자. 왼쪽의 경우 A->B->C가 반시계 방향인데, 이 때 x와 y의 외적은 양수다. 오른쪽의 경우 시계 방향이고, x와 y의 외적이 음수다. 이렇게 외적을 구하는 것을 임의로 ccw(A, B, C)이라고 해본다.

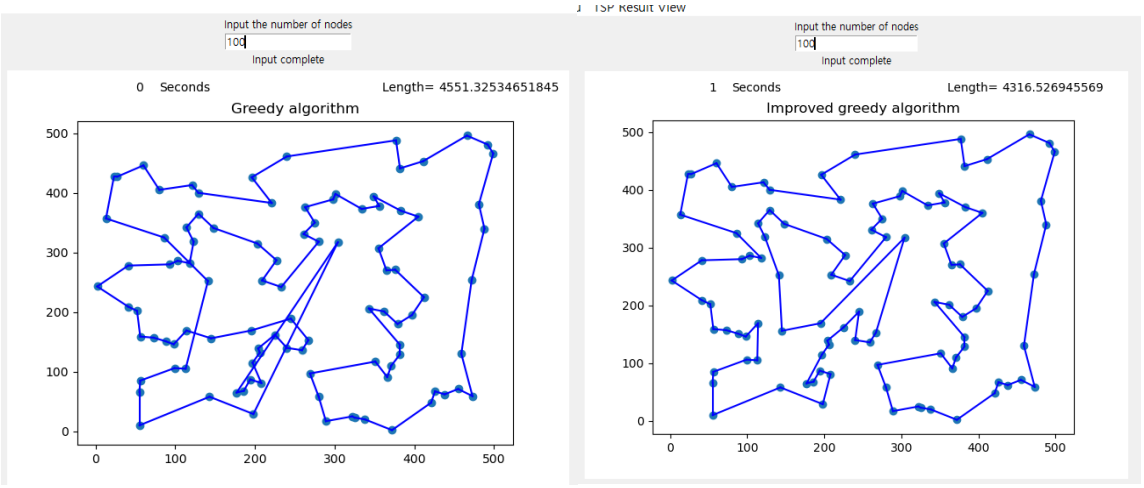


[그림 14] CCW 알고리즘 예시 2

선분 AB, CD가 교차하기 위해서는 두 가지 조건을 만족해야 한다.

```
ccw(A,B,C)*ccw(A,B,D) < 0, ccw(C,D,A)*ccw(C,D,B) < 0
```

선분 AB에 대해 C와 D가 각각 반대 방향에 있어야 음수 값이 나오며, 선분 CD에 대해 A와 B가 반대 방향에 있어야 음수 값이 나온다. 이를 이용해 선분 두 개의 교차 여부를 알 수 있다.



[그림 15] Improved algorithm 비교

위는 Greedy algorithm과 Improved greedy algorithm을 비교한 것이다. 교차 지점이 전부 없어졌음을 볼 수 있으며, 경로의 길이도 4551 -> 4316으로 줄어들었다.

2.2. 알고리즘 간 비교

2.2.1. 50-nodes, 100-nodes, 200-nodes problem

<경로 길이 비교> (파란색은 최선의 결과)

	50	**100**	**200**
Greedy	3069	5204	7240
Improved greedy	2962	4514	6606
Greedy 2-opt	2735	4147	5568
2-opt	2617	4054	5832
Greedy 3-opt	2857	5119	7132
Improved greedy 3-opt	2718	4604	6610
3-opt	2924	5031	6139
Improved 3-opt	2794	4510	6009
Genetic algorithm	3023	4964	7177
Improved genetic algorithm	2956	4542	6442
Simulated annealing	3011	4612	6794
Improved simulated annealing	2915	4408	6127

[표 1] 알고리즘 경로 길이 비교

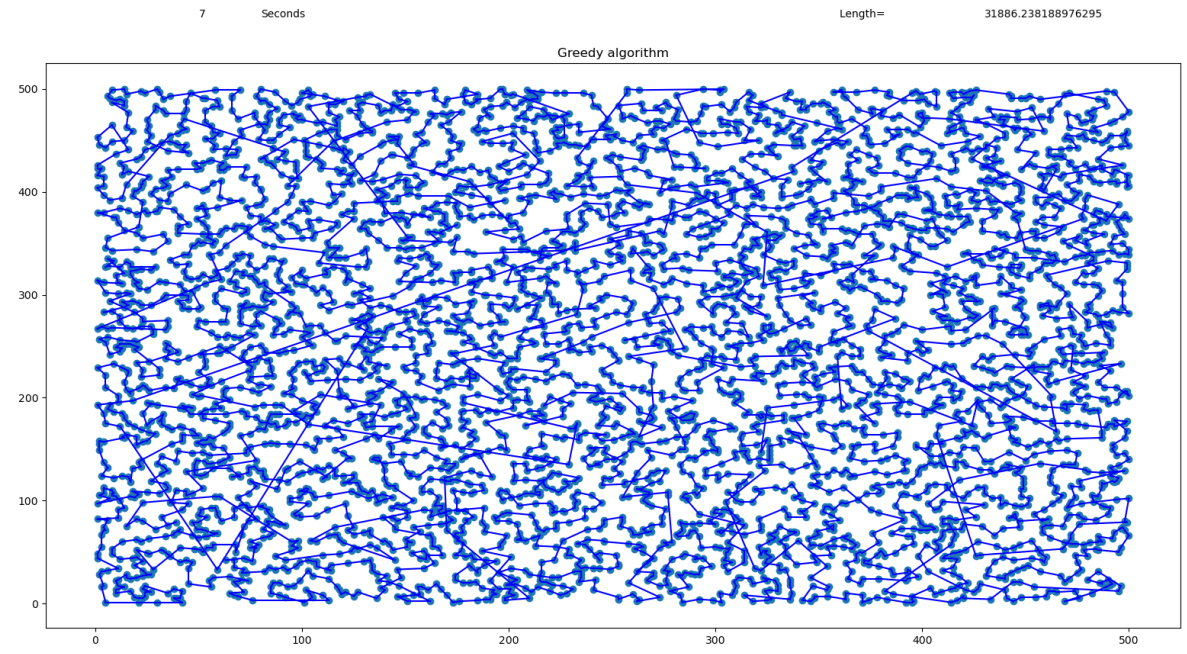
<소요 시간(Second) 비교> (빨간색은 가장 오래 걸린 시간)

	50	**100**	**200**
Greedy	0	0	0
Greedy 2-opt	0	8	108
2-opt	0	1	15
Greedy 3-opt	0	0	0
3-opt	5	67	390
Genetic algorithm	20	36	86
Simulated annealing	2	3	5

[표 2] 알고리즘 수행 시간 비교

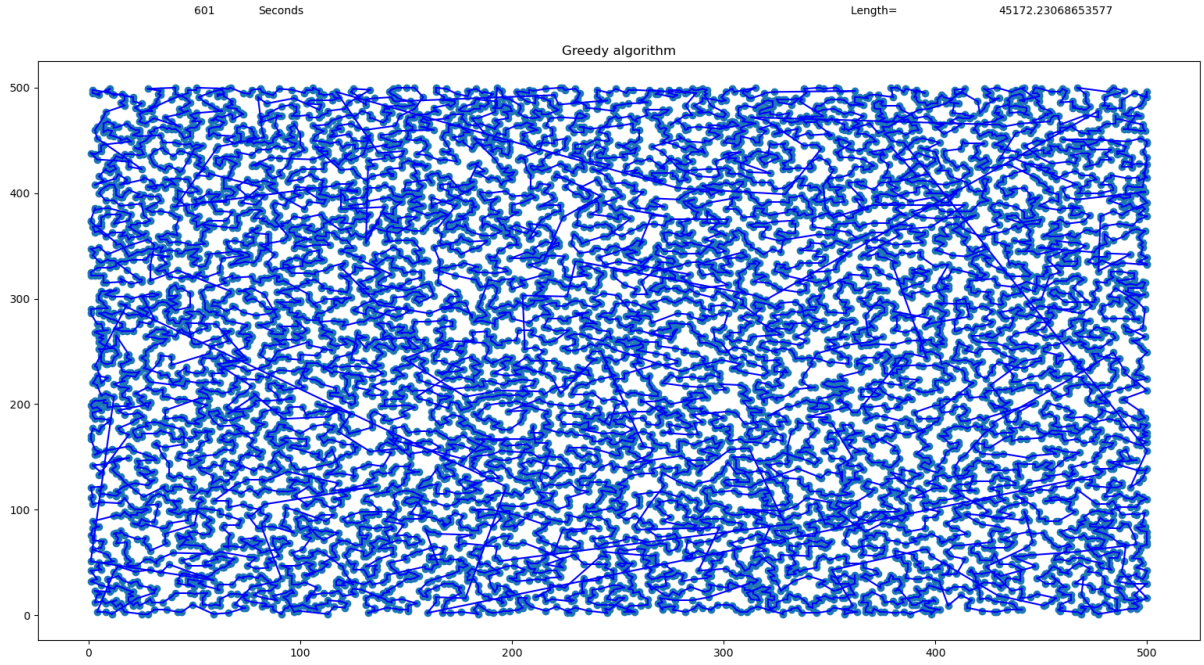
노드 수를 50, 100, 200개로 변화시키며 시뮬레이션을 한 결과이다. 경로 길이 면에서는 Greedy 2-opt와 2-opt가 가장 좋은 결과를 도출했다. 다른 알고리즘으로 만든 결과를 Improve시켰을 경우에도 앞의 두 경우보다 길이가 줄어들지 않았다. 알고리즘 소요 시간 면에서는 3-opt 가 일반적으로 가장 오랜 시간이 걸렸으며, 200개 문제에서는 390초가 소요되었다. Genetic algorithm은 50개 문제에서는 가장 오래 걸렸지만, 노드 수의 증가에 따른 시간 증가 폭이 크지 않음을 알 수 있다. Simulated annealing 역시 시간 증가 폭이 크지 않다. 반면 Greedy 2-opt와 2-opt의 경우 노드 수의 증가에 따라 소요 시간도 크게 증가했다. 알고리즘의 특성상 Edge가 교차하는 부분을 전부 풀어줘야 하기 때문에 오래 걸리는 것으로 보인다.

2.2.2. 5000-nodes, 10000 nodes problem



[그림 16] 5000개 sample: Greedy algorithm 실행 결과

주어진 5000개의 sample로 Greedy algorithm을 구했을 때, 경로 길이는 총 31886, 시간은 7초 경과되었다.



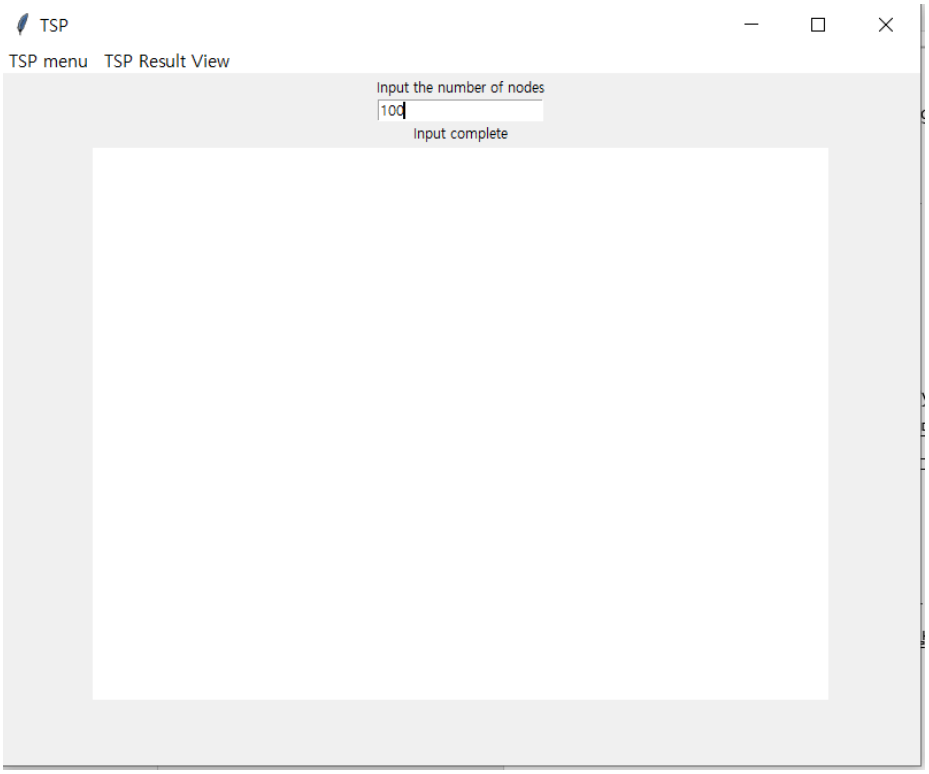
[그림 17] 10000개 sample: Greedy algorithm 실행 결과

10000개의 sample로 Greedy algorithm을 구했을 때는 경로 길이가 45172, 시간은 601초가 경과되었다.

2-opt 알고리즘도 실행해보려고 했으나, 시간이 너무 오래 걸려서 그런지 결과를 추출하지 못했다.

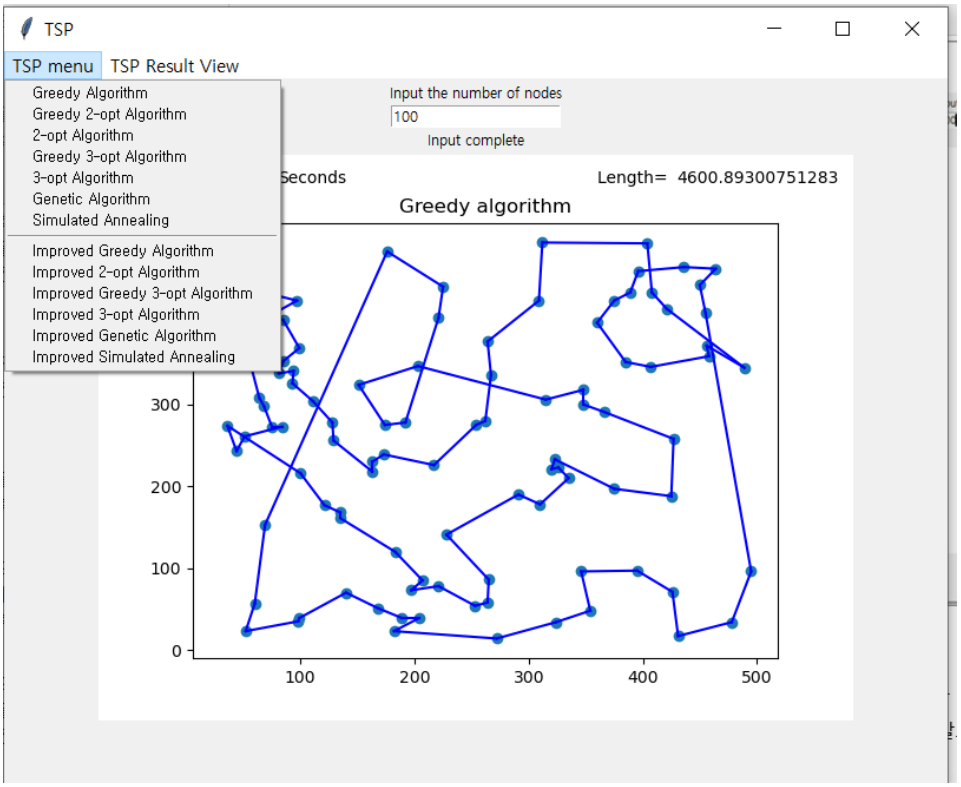
2.3. GUI 프로그램 구현

GUI 프로그램을 만들기 위해 Python에서 제공하는 라이브러리인 Tkinter를 사용하였다. 이 프로그램은 처음에 Input으로 총 노드의 개수 n을 입력한다. 입력 후 x, y좌표가 (1, 500) 사이에 있는 n개의 노드가 무작위로 생성된다. 이후 각 2개의 노드의 거리가 담긴 행렬이 만들어지면 완료됐다는 메시지가 출력된다.



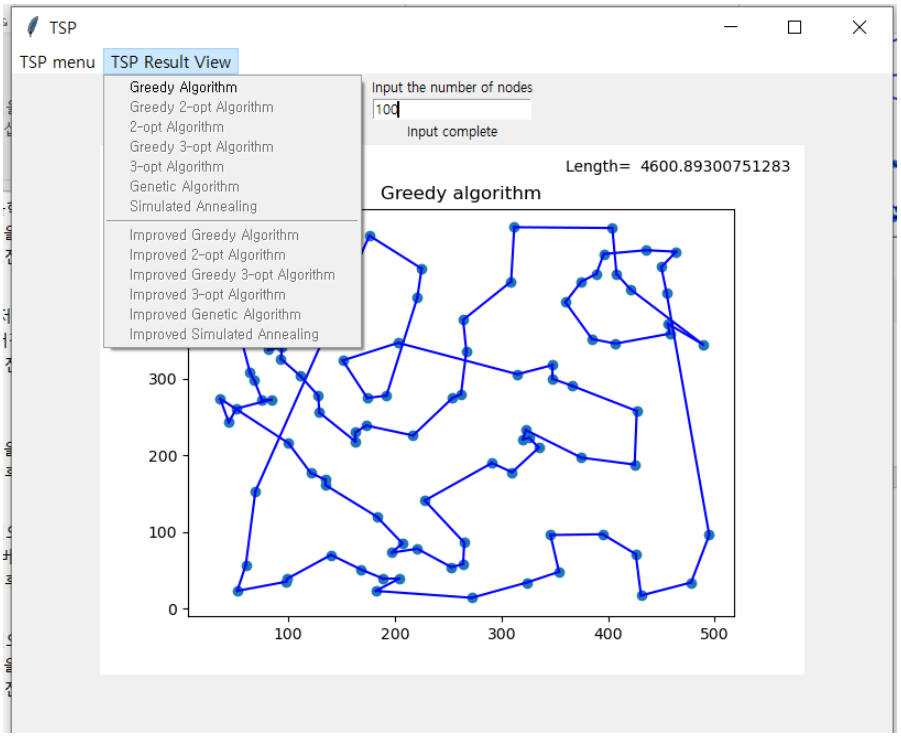
[그림 18] node 개수 입력

메뉴 창에서 원하는 알고리즘을 선택하면 결과를 출력할 수 있다. 출력되는 결과로는 총 경로의 모습, 경로의 총 길이, 그리고 알고리즘을 수행하는 데 소요된 시간이 있다. Improved algorithm들은 위쪽의 해당하는 알고리즘을 먼저 수행해야 이를 실행할 수 있다.



[그림 19] 메뉴 화면

<TSP Result View> 메뉴는 실행했던 알고리즘 결과를 다시 볼 수 있다. 실행하지 않았던 알고리즘 창은 비활성화되어 있다.



[그림 20] 메뉴 화면(결과 확인)

3. 결론

본 연구에서는 TSP의 각종 알고리즘을 Python으로 구현해보고, 최종 해의 길이와 소요 시간을 비교해 보았다. 코드를 작성하면서 데이터 구조를 주로 리스트로 사용하였는데, 이 때문인지 다수의 데이터를 입력했을 때 시간이 상당히 오래 걸린 것 같다. Greedy algorithm을 실행할 때 5000개에서 7초가 걸렸지만, 10000개에서는 무려 601초가 소요되었다. 2-opt 알고리즘을 실행했을 때는 5000개에서도 결과를 출력할 수 없었다. 나중에서야 데이터 구조로 numpy를 사용할 때 속도가 빨라진다는 것을 알았다. 추후 기회가 된다면 numpy를 사용하여 더 빠르게 알고리즘을 개선해 볼 생각이다.

4. 후기

이번에 처음으로 연구참여를 진행해 보았다. 처음 연구실에 들어가보니 다들 자신의 일에 집중하고 있었고, 나는 무엇을 해야 할지 몰라 당황하기도 했다. 다들 바쁘다 보니 도움은 그리 많이 요청하지 못했고, 혼자서 생각하고 프로그램을 만드는 시간이 대부분이었다. 시간이 지나고 나니 최종적인 구현에서 몇 가지 아쉬운 부분이 있었기에, 진작에 도움을 자주 청했다면 어땠을지 아쉽기도 했다. 또한, 이번 연구에서 나는 알고리즘의 속도를 개선하기보다는 메타휴리스틱 알고리즘을 포함해서 최대한 많은 알고리즘을 직접 구현하고, 이를 표현할 수 있는 GUI 프로그램을 만드는 데에 초점을 맞췄기에, 5000개나 10000개의 데이터를 입력했을 때 속도가 너무 느리거나 아예 실행이 되지 않은 경우가 많았던 것이 많이 아쉬웠다. 지금은 연구참여 시간이 다 끝났기에 속도를 개선한 결과를 바로 낼 수는 없겠지만, 나중에 개인적으로 알고리즘을 개선하여 빠른 속도를 내 보도록 구현하는 것도 좋은 시간이 될 것 같다. 만약 대학원에 진학하게 된다면 이번의 경험을 교훈 삼아 연구에 전념할 수 있도록 할 것이다.

마지막으로 연구참여를 허락해 주시고 저희의 의욕을 북돋워 주신 김병인 교수님, 그리고 TSP에 대해 친절하게 가르쳐 주신 권오현 선배님께 진심으로 감사드리겠습니다.

Appendix(python code)

```python
def greedy(data, matrix, nodes):
    go = time.time()
    plt.figure()
    sample = np.array(data)
    plt.scatter(sample[:, 0], sample[:, 1])
    nearest_addition = Tour()  # greedy algorithm 노드 순서
    nearest_addition.tour_node_order.append(0)
    start = 0
    tmp_list = list(range(0, len(data) - 1))

    total_distance_nearest = 0  # nearest addition 총 거리
    for j in range(len(data) - 1):
        shortest_distance = 10000000
        short = 0
        for k in tmp_list:

            if matrix[start][k + 1] < shortest_distance:
                shortest_distance = matrix[start][k + 1]
                short = k + 1
        nearest_addition.tour_node_order.append(short)

        plt.plot([nodes[start].x_coord, nodes[short].x_coord], [nodes[start].y_coord, nodes[short].y_coord], 'b')
        start = short
        total_distance_nearest += shortest_distance
        tmp_list.remove(short - 1)
```

[코드 1] Greedy algorithm

```python
def two(data, matrix, tour_node_order):
    go = time.time()
    array = np.array(data)
    solution = np.array(tour_node_order)

    distance_calculation = lambda r: sum(matrix[r[p]][r[p - 1]] for p in range(len(r)))
    swap_algorithm = lambda r, i, k: np.concatenate((r[0:i], r[k:-len(r) + i - 1:-1], r[k + 1:len(r)]))
    current_best_distance = distance_calculation(solution)

    swap = True
    while swap == True:
        swap = False
        for swap1 in range(1, len(solution) - 2):
            for swap2 in range(swap1 + 1, len(solution)):
                new_solution = swap_algorithm(solution, swap1, swap2)
                new_distance = distance_calculation(new_solution)
                if new_distance < current_best_distance:
                    solution = new_solution
                    current_best_distance = new_distance
                    swap = True
```

[코드 2] 2-opt algorithm

```python
def greedy_two(data, matrix, tour_node_order):
    go = time.time()
    array = np.array(data)
    solution = np.array(tour_node_order)

    distance_calculation = lambda r: sum(matrix[r[p]][r[p - 1]] for p in range(len(r)))
    swap_algorithm = lambda r, i, k: np.concatenate((r[0:i], r[k:-len(r) + i - 1:-1], r[k + 1:len(r)]))
    current_best_distance = distance_calculation(solution)

    swap = True
    while swap == True:
        ex_break = True
        swap = False
        for swap1 in range(1, len(solution) - 2):
            for swap2 in range(swap1 + 1, len(solution)):
                new_solution = swap_algorithm(solution, swap1, swap2)
                new_distance = distance_calculation(new_solution)
                if new_distance < current_best_distance:
                    solution = new_solution
                    current_best_distance = new_distance
                    swap = True
                    ex_break = False
                    break
            if ex_break == False:
                break
```

[코드 3] Greedy 2-opt algorithm

```python
def three_opt_transposition(tour, i, j, k, matrix):
    a, b, c, d, e, f = tour[i-1], tour[i], tour[j-1], tour[j], tour[k-1], tour[k % len(tour)]
    d0 = matrix[a][b] + matrix[c][d] + matrix[e][f]
    d1 = matrix[a][c] + matrix[b][d] + matrix[e][f]
    d2 = matrix[a][b] + matrix[c][e] + matrix[d][f]
    d3 = matrix[f][b] + matrix[c][d] + matrix[e][a]
    d4 = matrix[a][e] + matrix[b][d] + matrix[c][f]
    d5 = matrix[a][d] + matrix[b][f] + matrix[e][c]
    d6 = matrix[a][c] + matrix[f][d] + matrix[b][e]
    d7 = matrix[a][d] + matrix[b][e] + matrix[c][f]

    d_list = [d0, d1, d2, d3, d4, d5, d6, d7]
```

```
    if min(d_list) == d0:
        return 0
    elif min(d_list) == d1:
        tour[i:j] = list(reversed(tour[i:j]))
        return -d0+d1
    elif min(d_list) == d2:
        tour[j:k] = list(reversed(tour[j:k]))
        return -d0+d2
    elif min(d_list) == d3:
        tour[i:k] = list(reversed(tour[i:k]))
        return -d0+d3
    elif min(d_list) == d4:
        tmp = list(reversed(tour[j:k])) + tour[i:j]
        tour[i:k] = tmp
        return -d0+d4
    elif min(d_list) == d5:
        tmp = tour[j:k] + list(reversed(tour[i:j]))
        tour[i:k] = tmp
        return -d0+d5
    elif min(d_list) == d6:
        tmp = list(reversed(tour[i:j])) + list(reversed(tour[j:k]))
        tour[i:k] = tmp
        return -d0+d6
    elif min(d_list) == d7:
        tmp = tour[j:k] + tour[i:j]
        tour[i:k] = tmp
        return -d0+d7
```

```
def three(data, tour_node_order, matrix, nodes):
    go = time.time()
    sample = np.array(data)
    t_opt_order = tour_node_order.copy()
    for i in range(10):
        delta = 0
        for a in range(len(data)):
            for b in range(len(data)):
                if a == b:
                    continue
                for c in range(len(data)):
                    if c == a or c == b:
                        continue
                    delta += three_opt_transposition(t_opt_order, a, b, c, matrix)
    print(t_opt_order)
    new_t_opt_order = []
    for v in t_opt_order:
        if v not in new_t_opt_order:
            new_t_opt_order.append(v)


    total_distance_nearest_3opt = 0
```

[코드 4] 3-opt algorithm

```python
def genetic(data, tour_node_order, matrix, nodes, generations):
    go = time.time()
    def generate_solutions(population):
        set = []
        for i in range(population):
            sol_i = np.random.choice(list(range(len(data))), len(data), replace=False)
            new_sol_i = sol_i.tolist()
            set.append(new_sol_i)
        return set

    def other_solutions(population):
        other_set = []
        for i in range(population):
            other_order = tour_node_order.copy()
            integers = np.random.choice(len(other_order), 2, replace=False)
            left = min(integers)
            right = max(integers)
            while left < right:
                other_order[left], other_order[right] = other_order[right], other_order[left]
                left += 1
                right -= 1
            other_set.append(other_order)

        return other_set
```

```python
def find_fitnesses(set, population):
    fitness_list = []
    for i in range(population):
        total_fitness = 0
        for j in range(len(data) - 1):
            total_fitness += matrix[set[i][j]][set[i][j + 1]]
        total_fitness += matrix[set[i][j + 1]][set[i][0]]
        fitness_list.append(total_fitness)
    return fitness_list

def make_array(set, fitness_list):
    tmp_array = []
    for i in range(len(fitness_list)):
        tmp_array.append([set[i], fitness_list[i]])
    new_array = np.array(tmp_array)
```

```
sample = np.array(data)
population = 200

initial_set = other_solutions(population).copy()

initial_array = make_array(initial_set, find_fitnesses(initial_set, population)).copy()

a = next_generation(initial_array).copy()

for i in range(generations):
    b = find_fitnesses(a, len(a)).copy()
    a = next_generation(make_array(a, b)).copy()

final_fitness = find_fitnesses(a, len(a)).copy()
min_index = np.argmin(final_fitness)
final_order = a[min_index]

total_distance_genetic = 0
```

```
def next_generation(arr):
    #retain 0.1p solutions
    sort = arr[arr[:, 1].argsort()]
    new_sort = sort.tolist()
    new_generation = []   #next generation list
    for i in range(round(len(arr)*0.1)):
        new_generation.append(new_sort[i][0])

    #generate 0.89p solutions
    fitness_list = []
    new_arr = arr.tolist()
    for i in range(len(new_arr)):
        fitness_list.append(1/new_arr[i][1])
    total_fit = sum(fitness_list)
    prob_list = []
    for i in range(len(fitness_list)):
        prob_list.append(fitness_list[i]/total_fit)

    tmp = list(range(len(fitness_list)))
    for i in range(round(len(fitness_list)*0.445)):
        sample = np.random.choice(tmp, 2, p=prob_list, replace=False)
        new_sample = []
        new_sample.append(new_arr[sample[0]])

        new_sample.append(new_arr[sample[1]])
        c = crossover(new_sample[0][0], new_sample[1][0])

        new_generation.append(c[0])
        new_generation.append(c[1])
```

```
#select 0.01p solutions
tmp2 = list(range(len(fitness_list)))
mutant = np.random.choice(tmp2, round(len(new_arr)/100), replace=False)
new_mutant = []
for i in range(round(len(new_arr)/100)):
    new_mutant.append(new_arr[mutant[i]])
for i in range(len(new_mutant)):
    integers = np.random.choice(len(new_mutant[i][0]), 2, replace=False)
    a = new_mutant[i][0]
    a[integers[0]], a[integers[1]] = a[integers[1]], a[integers[0]]
    new_generation.append(a)

return new_generation
```

[코드 5] Genetic algorithm

```
def sa(data, tour_node_order, matrix, total_distance_nearest, number):
    go = time.time()
    sample = np.array(data)
    sa_order = tour_node_order.copy()
    cost0 = total_distance_nearest
    sa_data = []
    for w in range(len(data)):
        sa_data.append(data[w])
    T = 30
    factor = 0.9
```

```
for i in range(number):
    print(i, 'cost =', cost0)

    T = T * factor  # T값 줄이기
    for j in range(100):
        # exchange two coordinates and get a new neighbor solution
        r1, r2 = np.random.randint(0, len(sa_data), size=2)

        temp = sa_order[r1]
        sa_order[r1] = sa_order[r2]
        sa_order[r2] = temp

        cost1 = 0
        for l in range(len(sa_data) - 1):
            cost1 += matrix[sa_order[l]][sa_order[l + 1]]
        cost1 += matrix[sa_order[l + 1]][sa_order[0]]
        if cost1 < cost0:
            cost0 = cost1
        else:
            x = np.random.uniform()
            if x < np.exp((cost0 - cost1) / T):  # 온도가 높을 때 무작위 변경 가능성 증가
                cost0 = cost1
            else:
                temp = sa_order[r1]
                sa_order[r1] = sa_order[r2]
                sa_order[r2] = temp
```

[코드 6] Simulated annealing

```python
def improved_greedy(data, tour_node_order, matrix, nodes):
    go = time.time()
    sample = np.array(data)
    aga_order = tour_node_order.copy()

    for k in range(10):

        for i in range(len(data) - 1):
            for j in range(len(data) - 1):

                if intersect(data[aga_order[i]], data[aga_order[i + 1]],
                             data[aga_order[j]], data[aga_order[j + 1]]) == 1:

                    short = i + 1
                    long = j
                    while short < long:
                        aga_order[short], aga_order[long] = aga_order[long], aga_order[short]
                        short += 1
                        long -= 1
        for l in range(len(data) - 1):

            if intersect(data[aga_order[i + 1]], data[aga_order[0]], data[aga_order[l]], data[aga_order[l + 1]]) == 1:
                transpose = True
                short = 0
                long = l
                while short < long:
                    aga_order[short], aga_order[long] = aga_order[long], aga_order[short]
                    short += 1
                    long -= 1
```

```python
def ccw(a, b, c):
    return a[0] * b[1] + b[0] * c[1] + c[0] * a[1] - (b[0] * a[1] + c[0] * b[1] + a[0] * c[1])

def intersect(A, B, C, D):
    if ccw(A, B, C) * ccw(A, B, D) <= 0 and ccw(C, D, A) * ccw(C, D, B) <= 0:
        if ccw(A, B, C) * ccw(A, B, D) == 0 and ccw(C, D, A) * ccw(C, D, B) == 0:
            if (max(A[0], B[0]) >= min(C[0], D[0]) and max(C[0], D[0]) >= min(A[0], B[0])) \
                    and (max(A[1], B[1]) >= min(C[1], D[1]) and max(C[1], D[1]) >= min(A[1], B[1])):
                return 1
            else:
                return 0
        else:
            return 1

    else:
        return 0
```

[코드 7] Algorithm improvement