CS454 coursework 1

20120888 조민기

1. Description

사용한 알고리즘 : 탐욕 알고리즘, 로컬서치, Simulated annealing

python tsp_solver.py rl11849.tsp 으로 실행할 경우 디폴트로 어닐링 과정이 수행되며
각각의 실행 방법은
python tsp_solver.py rl11849.tsp greedy
python tsp_solver.py rl11849.tsp local
python tsp_solver.py rl11849.tsp anneal
을 입력하면 된다.

각 데이터는 Node 라는 클래스에 의해 관리된다. main() 함수의 초반 부분은 .tsp파일을 읽어 번호, x좌표, y좌표를 각각 self.id, self.x, self.y에 담은 Node 객체를 생성한다. 그리고 이들을 node_lst라는 리스트에 담는다. node_lst의 타입은 list of Node 이다.

2. Algorithm & implementation

아래의 세가지 함수 모두 list of Node → list of Node 타입이다. 즉 파일로부터 입력된 리스트와 출력 형태가 항상 같으므로 anneal(local_opt(Greedy(node_lst))) 이렇게 모두 사용할 수도 있다. 이런 함수들을 통해 생성된 list of Node는 path_dst(node_lst) 함수를 통해 총 거리를 알 수 있다. (순환 거리이다.)

그리고 main 함수 내부에 simple이라는 (0,200) 사이의 랜덤한 정수를 좌표로 갖는 100개의 노드 리스트가 있다. 이를 통해 성능 테스트를 할 수 있다.

2.1 Greedy Algorithm

Greedy(node_lst) 로 실행하면 탐욕 알고리즘에 의해 정렬된 리스트가 출력된다. 이는 도시의 수가 적을 때는 적당한 성능으로 동작하지만, 도시의 수가 많은 rl11849에 대해 적용할 경우 무려 5천만이 넘는 코스트를 기록하는 등 매우 성능이 나쁘다. 하지만 어느 정도 경로를 정리하여 이후 로컬 서치를 하는데 시간을 단축 시키는 효과가 있다.

2.2 Local Search

local_opt(node_lst) 로 실행하면 로컬 서치를 실행한 뒤 경로를 담은 리스트가 출력된다. 로컬서치를 사용하여 초기로 입력된 순서로 정해진 경로를 기반으로 Local optima를 탐색해낸다. 사용한 방법은 2-opt라고 알려져 있는데, 경로에서 특정 두 지점의 경로를 서로 교환 하였을 때의 코스트가 원래 코스트보다 나은지 비교, 더 좋은 성능을 갖는 경로를 선택하는 작업이다.
경로를 바꾸는 디테일은 1,2,3,4,5… i, i+1, i+2, … , k-1, k, k+1, … , n 의 수열상에서 i와 k점에서의 간선을 서로 교차하는 것이다. 이 때 코스트가 더 낮아진다면 경로를 1,2,3, … i, k, k-1, k-2, … , i+2, i+1, k+1, … n 이런 순서로 바꾸게 된다. 이 작업을 전체 경로에 대해서 수행하며, 더 나은 코스트를 확보할 수 있는 교환 쌍이 발견되지 않을 때 까지 반복한다.
최종적으로는 모든 경로에서 어떤 두 점의 간선을 서로 교차하였을 때 더 좋은 코스트를 갖는 경로를 찾을 수 없는 상태가 되므로 Optima라고 할 수 있겠으나, 이는 global optima가 아니다.
(본인의 컴퓨터 스펙이 32비트여서 rl11849를 로컬 서치로 도저히 수행하지 못해 결과를 내지 못하고 있다. Greedy 알고리즘을 수행하는데 13분이 걸렸고, 따라서 해당 문제에 대해 얼마나 발전이 있었는지 기술하기 힘들다.)

2.3 Simulated Annealing

anneal(node_lst)로 실행하면 Simulated Annealing을 사용한 경로를 담은 리스트가 출력된다. 최대 온도 T에서 T_min까지 냉각되도록 변수를 설정해두었고 default로 각각 1.0e+100과 0.1로 설정되어있다. 코스트는 현재 솔루션에서의 여행 거리이며 이웃 솔루션은 위의 local search에서와 비슷하게 랜덤하게

기존 경로상의 두 간선을 선택해서 서 교차한 경로를 사용했다. alpha는 디폴트로 0.9로 설정되어 있고 내부의 num_iter를 통해 iteration 횟수를 설정할 수 있게 두었다.
(마찬가지로 rl11849를 Simulated Annealing으로 도저히 수행하지 못해 결과를 내지 못하고 있다. 매우 낮은 초기 온도, 높은 알파 등의 조건에서 3천만까지 떨어지는 것을 확인했으나 그 이상의 스펙에서 프로그램을 실행하면 결과를 얻을 수가 없었다.)

3. Experiment Result

Greedy의 경우 Node의 수가 적은 문제에 대해선 어느 정도 성능을 보여주었으나, 1만개 이상의 Node를 가진 문제에 대해서는 Random search보다 나을 것이 없을 정도로 성능이 나빴다. 그리고 Local search의 경우 프로그램을 가동 한지 수 시간이 지났으나 아직 결과를 출력하지 못한 상태이고, Annealing 역시 적절한 스펙을 입력해주었을 때 본인의 컴퓨터가 1만개의 도시를 가진 셋을 소화할 수가 없어서 rl11849.tsp에 적용한 결과를 확인하지 못했다.
해서 100개의 랜덤 도시를 생성한 뒤 이를 토대로 실험을 하였다. 이 랜덤도시는 프로그램 내 simple 이라는 리스트에 등록이 되어있으며 python -i tsp_solver.py 로 들어가서 사용할 수 있다. 동일 리스트에 대해 Greedy는 11000대의 경로를 찾아내었다. 그리고 Greedy에 비해 Local search가 좋은 성능을 보여주었다. 하지만 Local search의 경우 코스트가 2000부터 9000까지 불균일하게 나왔는데, 초기 실행값에 따라 출력 값이 많은 영향을 받는 모습을 보여줬다. 이는 Local optima에 도달하면 더이상의 개선이 없고 알고리즘이 멈추는 구조이기 때문이다. 마지막으로 annealing을 적용한 결과 수차례 Local search로 얻을 수 없던 코스트 1140의 경로를 단번에 찾아내었다.
사실 이 문제에 대해서 Particle swarm optimization등의 알고리즘을 적용해보았으나 위의 알고리즘보다 더 느리게 동작하여 역 답을 얻을 수가 없었다.