CS492 Search Based Software Engineering

20165501 박준영

1. TSP 구현체 실행법

1) Unzip **tsp.tar.gz** 파일
$ tar –xzvf tsp.tar.gz
2) Makefile 실행
$ make clean; make
3) **tsp** 실행파일을 원하는 옵션과 함께 실행(-p PopulationSize, -f # of Fitness)
$ ./tsp (–p N) (–f N) {INPUT.tsp} Ex) $ ./tsp rl11849.tsp //입력 파일 경로만 입력한 경우 default p, f 값 사용 -p 의 default 는 30 이다. -f 의 default 는 -1 로 무한을 의미한다.

2. Euclidean TSP 문제 관찰

2.1 꼬인 경로 풀기(untwist)

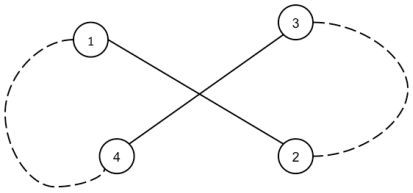


Figure 1 꼬인경로

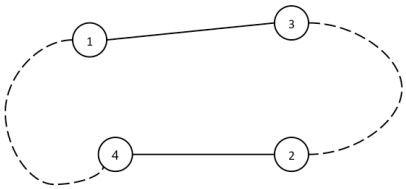


Figure 2 풀린 경로

TSP 의 경로 중 위 그림처럼 edge 가 교차하는 점이 있으면,
교차하지 않도록 방문 순서를 변경하면 경로합이 항상 줄어든다.
이 꼬인 경로를 푸는 함수를 untwist 라고 이름 지었다.
untwist 함수를 활용한 hill climbing 알고리즘을 구상할 수 있다.
Neighbor 는 꼬인 edge 쌍 하나를 푼 것으로 생각할 수 있고 꼬인 곳이 없는 경로를 local optimum 으로 볼 수 있다.
Cross 라는 함수를 정의해서 입력 경로에서 꼬인 edge 쌍의 개수를 리턴하도록 하였다.
선분 2 개가 교차하는지 판단하는 코드는 아래 링크에서 가져다 사용하였다.
http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/

2.2 Two-approximation 알고리즘

주어진 입력의 minimum spanning tree 를 구하고 tree 는 M, 그 edge 의 합을 m 이라고 하자.
M 의 edge 만을 사용해서 모든 노드를 한 번 이상 traverse 하는 알고리즘으로 roundtrip 경로를 찾을 수 있다. 이 경로의 합은 2m 임을 쉽게 알 수 있다.
Minimum spanning tree 의 정의에 의해 m 은 TSP optimum 값(opt)보다 작다.
(어떤 node 에서 도달할 수 있는 가장 짧은 다른 node 와 연결되어 있기 때문)
따라서 아래 부등식이 성립한다.
m <= opt <= 2m <= 2opt
따라서 2m 은 TSP 의 2-approximation 알고리즘이다.
간단한 2-approximation 알고리즘이 존재하기 때문에 search-based 알고리즘의 seed 값으로 사용하기 용이하다.

3. Search-based 알고리즘

3.1 사용한 알고리즘 – GA(유전자 알고리즘)

선택이유
1) TSP 문제가 오래되고 유명하다보니 대회 또는 연구 결과 내용을 접할 수 있는데, 단기간에 직접 구현하기 복잡한 것들이 많고 다른 사람의 것을 답습하는 대신 직접 시행착오를 겪으면서 부딪혀보고 싶었다.
2) 수업시간에 배운 local-search 로는 좋은 해를 구하기 힘들 것 같았다.
Local minimum 이 많은 ruggedness 가 큰 문제라고 판단하였다.
3) 적어도 SA 정도 되는 random search 알고리즘을 사용하는 것도 생각해보다가 GA 가 이곳 저곳 튜닝할 수 있는 곳이 많을 것으로 판단하고 선택하게 되었다.

3.2 Fitness 함수

1) 정의 : Cycle(p)
가장 흔하게 사용할 경로 edge 의 합을 fitness 함수로 쓰게 되었다.
작을 수록 최적이다.

2) 시행착오 – Multi-objective Optimisation
2.1 의 꼬인 경로 풀기를 활용하여 fitness 함수를 2 개의 objective 로 만드는 시도를 했었다.
(Cycle(p), Cross(p)) = (경로합, 꼬인개수)

꼬인 개수가 크다는 것은 앞으로 꼬인 것을 풀어나가면 경로합이 줄어든다는 것을 “암시한다”. 꼬인 개수가 크다는 것은 앞으로 이 경로의 성장 가능성이 크다고 생각해서 population selection 에 multi-objective optimisation 알고리즘을 사용해보려고 했으나 실제로 dominate 하는 것과 “암시하는” 것이 달라서 의미가 없다는 것을 깨닫고 population monitoring 할 때 local optimum 에 갇혔는지 확인하는 지표로만 사용하였다.

3.3 Initial Population - Random

Fisher-Yates shuffle 알고리즘으로 random permutaion 을 population size limit 만큼 생성하였다.

3.4 Parent Matching

1) 시행착오 – 3-tier Matching
Population 을 fitness 순서로 정렬하고 3 개의 tier 로 나누었다.
1st : 상위 20%
2nd : 상위 20~80%
3rd : 상위 80~100%
3 개의 tier 마다 다른 matching 방법을 사용하였다.
1st : (i, i+1), (i, i+2)
i 번째 person 은 i+1, i+2 와 match 시켜서 약 4 개의 자손을 생성한다.
2nd : (i, i+1)
i 번째 person 은 i+1 와 match 시켜서 약 2 개의 자손을 생성한다.
2nd : (i, popLimit-i)
i 번째 person 은 1st-tier 에 있는 popLimit-i 와 match 시켜서 1 개의 자손을 생성한다.

fitness 가 좋을 수록다음 세대에 많은 자손을 만들기 때문에 evolutionary pressure 를 높이는 방법이다.

2) 1-N Mathcing
(i, popLimit-i)를 매칭한다.

3) Adjacent Mathcing
(i, i+1)을 매칭하고 (popLimit,1)을 매칭한다.

3.5 Crossover
1) 시행착오 – Noncommutative Order1 Crossover
가장 기본적인 permutation crossover 알고리즘 중 하나인 Order 1 Crossover 를 변형하여 사용했다.
(http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/Order1CrossoverOperator.aspx)
두 개의 경로 p1, p2 가 왔을 때,
p1 의 fitness 가 좋은 경우(1st, 2nd tier 에서 parent match 된 경우)
  p1 의 꼬인개수가 더 큰 경우: 90~95%의 길이만큼 보존하도록 했다.
  p1 의 꼬인개수가 작은 경우 : 75~85%의 길이만큼 보존하도록 했다.
p2 의 fitness 가 좋은 경우 의 50%의 길이만큼 보존하도록 했다.

2) N/2 Order1 Crossover
Order1 crossover 를 시작 위치는 랜덤, 길이는 N/2 로 고정한 버전이다.

3) Hill Climbing
정석 GA 의 crossover 는 아니지만 (자가향상이기 때문) 3.8 Hill Climbing 을 하나의 step 적용한 것으로 offspring 을 만들었다.

3.6 Mutation
두 종류의 mutation 을 사용하였다.
1) Random

Initial population 생성에 사용한 random permutation 을 똑같이 사용하였다.
완전히 새로운 permutation 을 생성한다.
2) Reverse
꼬인 경로를 푸는 방법이 1-2 와 3-4 가 꼬여있을때, 2 부터 3 사이의 경로를 뒤집는 것이다.(1-2-p-3-4..를 1-3-rp-2-4.. 처럼)
랜덤한 두 위치를 선택하여 뒤집는 것을 mutation 으로 넣어 랜덤하게 꼬인 것이 풀리거나 꼬이도록 하였다.

3.7 Population Selection
1)시행착오 – Simple
간단하게 현재 population 을 fitness 순서로 정렬하고 population size limit 을 넘어간 것들을 제거하였다.

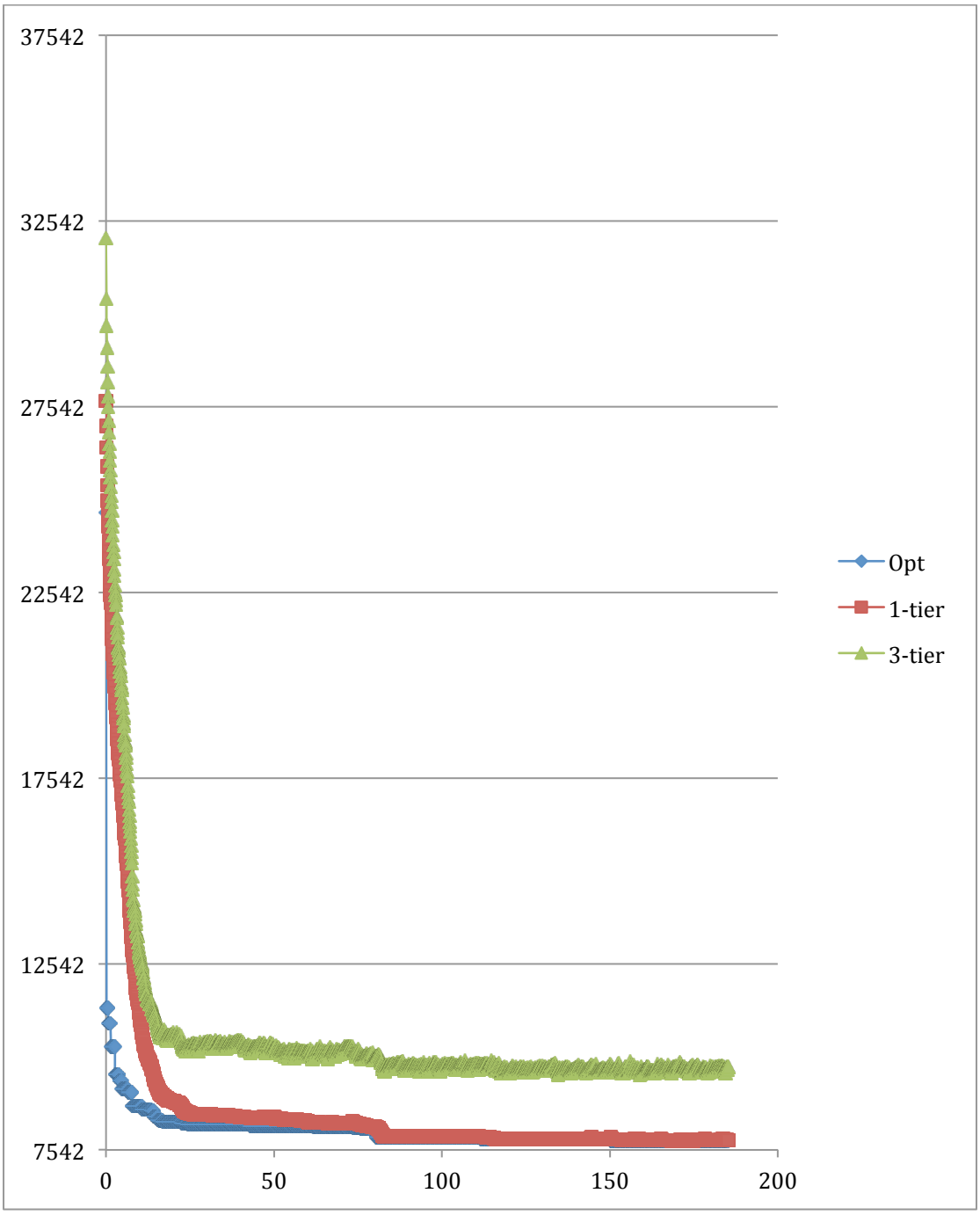
2) LCS
Longest Common Sequence 를 fitness 정렬 후인접한 population 개체끼리 비교하여 유사도를 확인한다. 이때 LCS/N 을 유사도로 정의한다.
비슷한 개체들을 제거하여 population 의 다양성을 확보할 수 있다.

3.8 Hill Climbing
untwist 함수를 이용하여 현재 population 에서 fitness 가 가장 좋은 한 경로의 꼬인 것들을 random 한 순서로 모두 푼 뒤에 현재 까지 찾은 solution 과 비교하도록 했다.

4. 실험
4.1 작은 입력에서 실험
1) 실험 환경

Input	berline52.tsp
Termination	2000 iterations
-p(Population Limit)	1000
GA settings	
Parent Matching	3-tier
Crossover	Noncom. Order 1
Mutation - Random	3.5%
Mutation - Reverse	1.5%

2) 실험 결과



X 축은 시간, Y 축은 경로합 이다. 빨간선과 초록선은 population 의 1-tier 와 3-tier 의 평균값이다.
2000 iteration 에서 optimum 은 7797 이었다.
같은 setting 으로 기록은 못했지만 global optimum 인 7542 에 도달하기도 했었다.
대체로 2000 iteration 을 돌리면 7800 근처의 값을 찾는다.

3) 필요한 수정사항
GA 를 새로 돌릴때 마다 수렴하는 지점이 제각각이다. 현재의 알고리즘은 local optimum 에 도달했을 때 모든 population 이 다양성을 잃는다는 결론을 내렸다.
Mutation 으로 local minimum 을 탈출하는 것도 잘 되지 않았다.

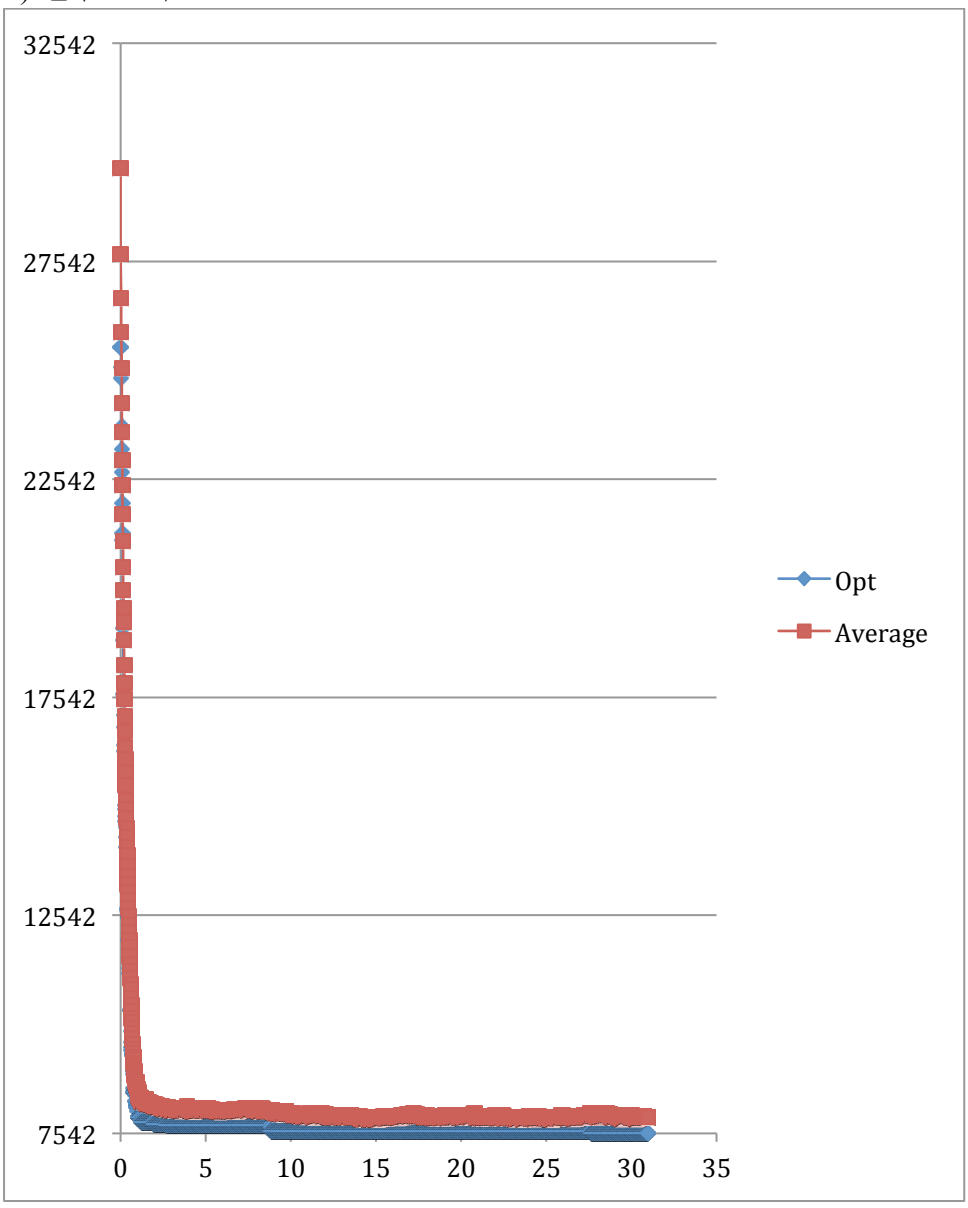
우선 population 의 다양성을 보장할 수 있는 population selection 을 추가해보기로 하였다.

4.2 작은 입력에서 실험 - LCS

1) 실험 환경

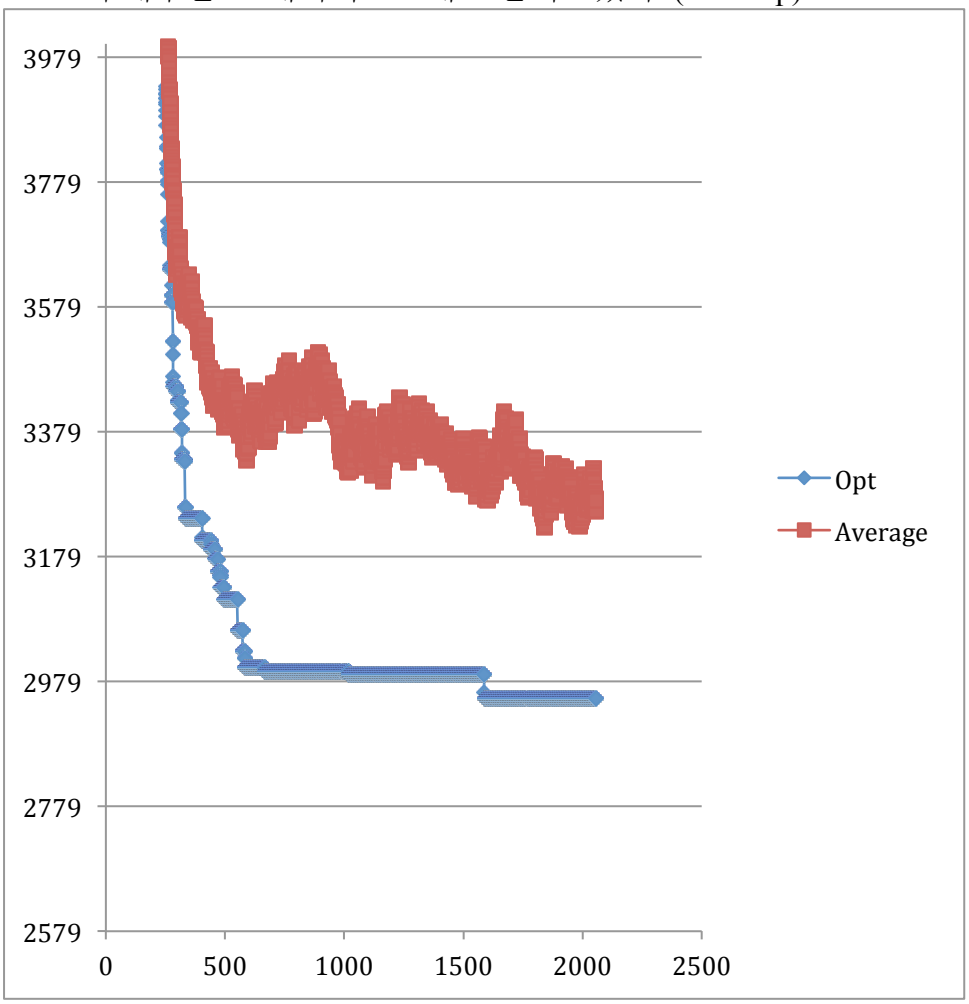
Input	berline52.tsp
Termination	Finding Optimum
-p(Population Limit)	1000
GA settings	
Parent Matching	1-N, Adjacent, One step Hill Climbing
Crossover	N/2 Order 1
Mutation - Random	3.5%
Mutation - Reverse	1.5%
Population Selection	LCS

2) 실험 결과



30 초만에 global optimum 에 도달하였다. Population 의 다양성을 갖추는 것이 효과적인 것을 확인하였다.

Node 의 개수를 52 개에서 280 개로 늘려보았다. (a280.tsp)



500 초 까지 빠르게 좋은 solution 을 찾아나간다.
500 초 이후부터 1000 초 가까이 더 좋은 solution 을 찾지 못하고 local optimum 에 갇혀있다.
최종적으로 2951 까지 구하였다.

3) 필요한 수정사항
Initial population 을 minimum spanning tree 와 그 variant 들로 구성해보려고 했다.

5. Heuristic 알고리즘

5.1 구현

2-approximation 알고리즘을 시작 경로로 random hill climbing 알고리즘을 적용한 결과 GA 로 random initial population 을 사용한 것 보다 월등한 결과를 얻을 수 있었다.

5.2 결과

rl11849.tsp 에 적용한 결과 GA 는 초기 결과에서 거의 향상이 없었지만,

5.1 heuristic 알고리즘을 적용한 결과 1357660 정도에서 계속 향상되고 있는 중이다.

6. 결론

GA 를 적용해보기에는 입력의 크기가 너무 커서 GA 가 잘 동작하는 parameter 설정값을 찾을 수가 없었다.
반면에 간단하게 적용가능한 heuristic 이 좋은 성능을 보였다.
GA 로도 node 개수가 52 개인 것에서는 global optimum 을 찾은 것을 보면 parameter 설정을 할 수 있는 충분한 시간이 있다면 큰 입력에서도 잘 동작하는 구현체를 찾을 수 있지 않을까 싶다.