<2024 Algorithm Assignment 03: Max-cut with Pure GA and Sorting>

2022-1146 강연주

1. Assignment02와 비교 해의 표현, 연산자, 수정 사항과 이유

1) 해의 표현

과제2와 마찬가지로 0과 1로 구성된 문자열로 구현했다. 0과 1은 그래프의 정점을 나타내고, 그래프의 각 정점을 두 집합 중 하나로 분류했다. 예를 들어 문자열 '10101010'은 여덟 개의 정점을 가진 그래프에 대한 하나의 해이다. 이 경우 홀수 번째 정점이 한 집합에 속하고, 짝수 번째 정점이 또 다른 집합에 속하는 것을 의미한다.

2) Selection

과제2에선 rouletteWheel 방식을 사용했다. 과제3에선 tournament 방식으로 구현했다. uniform_int_distribution<>를 사용해 두 개체를 랜덤하게 선택하고, 더 높은 적합도를 가진 개체가 선택될 확률을 60%로 줬다. 만약 두 개체의 fitness가 같으면 50%의 확률로 하나의 개체가 선택되게 했다. tournament는 다양성 보존을 위해 선택했다. rouletteWheel에서 generation 수를 2500으로 설정하니 1500세대 때부터 같은 값만 도출되는 일이 발생했다. 이를 방지하고자 tournament를 선택했다.

과제2의 rouletteWheel 선택 방식은 적합도의 총합이 음수일 가능성을 고려하지 않았다. 여기서 tournament와 비교한 rouletteWheel은 이 문제를 해결하기 위해 모든 적합도가 음수이거나 0인 특수한 경우에도 알고리즘이 계속 실행될 수 있도록 변경했다. 과제2에서 chimera 데이터를 run하면 코어덤프가 발생하지만 변경된 룰렛휠로 교체하고 과제2에서 run했더니 결괏값이 나오는 것을 확인할 수 있었다.

3) Crossover

기존에는 1-point crossover를 사용했다. 과제3에선 Uniform Crossover로 구현했다. 부모 해의 각 유전자 위치에서 무작위로 유전자를 선택하여 자손을 생성하고 특정 확률(70%)로 부모 1의 유전자를 선택하고 그렇지 않으면 부모2의 유전자를 선택하게 했다. Uniform은 부모 해들의 특성을 더 균일하게 조합할 수 있어 다양성을 증가시키고 새로운 해의 탐색 공간을 넓히기 때문에 선택했다.

4) Mutation

기존과 동일한 돌연변이 확률(mutationRate)을 pow(10, -2) 즉, 1%의 확률로 설정했다. 입력으로 부모 해를 받으면 문자열의 각 문자를 순회하고 각 위치에 대해 돌연변이 확률에 따라 난수를 생성한다. 이 난수가 돌연변이 확률보다 작은 경우 해당 유전자의 값을 변경하도록 구현했다. mutation이 발생하면 현재 유전자 값이 '0'인 경우 '1'로, '1'인 경우 '0'으로 변경된다.

5) Replacement

기존의 Genitor-style 방식에서 새로운 자손이 기존 population 중 가장 열등한 150개의 개체를 대체할 수 있도록 변경했다. 이 방식은 최악의 해들을 제거함으로써 알고리즘의 효율을 개선할 수 있다. population을 처음부터 끝까지 순회하거나 replaced가 150에 도달할 때까지 for 루프를 돌며 각 반복에서 현재 population의 적합도가 더 낮다면 그 위치에 새로운 자손으로 교체하고 그 위치의 적합도도 새 자손의 적합도로 업데이트하게끔 구현했다. 교체가 발생하면 replaced 변수를 1 증가시켜 교체 횟수를 업데이트한다.

2. GA를 선택한 과정과 사용한 연산자(selection, crossover, mutation, replacement)

- GA를 선택하는 과정에서 정렬은 사용하지 않았다.

1) selection 선택 과정 (rouletteWheel, tournament)

- one-point crossover, mutaion 확률 0.01, replacement하는 인구 수 200, population 수 500, generation 수 1000, 사용한 데이터 chimera_946으로 동일한 조건에서 비교했다.
- 토너먼트에서 더 적합한 개체가 선택될 확률은 60%로 설정했다.

selection	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
rouletteWheel	10	25516	21008	23146.22	1288.26
tournament	10	24952	21460	22602.22	1357.12

결과값이 룰렛휠이 더 좋음에도 토너먼트를 선택한 이유는 세대 수를 2000까지(제한 시간 180초 마지노선) 증가시켰더니 룰렛휠을 이용했을 때 쉽게 local optima에 빠졌기 때문이다. 세대 수가 적다면 룰렛휠이 좋을지라도 세대 수가 크다면 토너먼트를 선택하는 게 더 좋은 값을 낼 수 있어 토너먼트를 선택했다.

2) crossover 선택 과정 (one-point, two-point, uniform)

- tournament, mutaion 확률 0.01, replacement하는 인구 수 200, population 수 500, generation 수 1000, 사용한 데이터 chimera_946으로 동일한 조건에서 비교했다.
- 유니폼에서 부모1을 선택하는 특정 확률은 70%로 설정했다.

crossover	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
one-point	5	21964	18816	20336	1275.74
two-point	5	22963	20672	21829.4	1057.85
uniform	5	24852	22356	23174.4	966.00

two-point crossover가 기존에 사용했던 one-point보다 더 다양한 변이를 제공할 것이라고 생각해 two-point crossover도 구현했다. 표를 보면 one-point보다 성능이 좋은 것을 알 수 있다. 그러나 유니폼의 평균 품질이 독보적으로 높고, 다양성을 최대화할 필요가 있는 문제에 적합하다고 판단해 유니폼을 선택했다.

3) mutaion 선택 과정

- tournament, uniform crossover, replacement하는 인구 수 200, population 수 500, generation 수 1000, 사용한 데이터 chimera_946으로 동일한 조건에서 비교했다.

mutationRate	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
0.01	5	25048	22356	23810.4	1090.01

mutationRate를 0.1로 주었더니 결괏값이 음수로 나왔다. 이처럼 돌연변이 확률을 높이면 개체의 유전적 다양성이 급격하게 증가하지만 기존의 적합한 유전자 set을 망가트릴 수 있다. 유리한 해들을 잃을 수 있기 때문에 돌연변이 확률을 그대로 0.01로 설정했다.

4) replacement 선택 과정

tournament, uniform crossover, mutaion 확률 0.01, population 수 500, generation 수 800, 사용한 데이터 chimera_946으로 동일한 조건에서 비교했다.

교체 인구 수	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
100	5	23308	19720	21612	1517.08
150	5	22960	20352	21859.2	1071.33
200	5	22720	19428	20689.6	1296.63
300	5	22460	18488	20646.4	1477.95
400	5	23132	17508	19500	2257.38

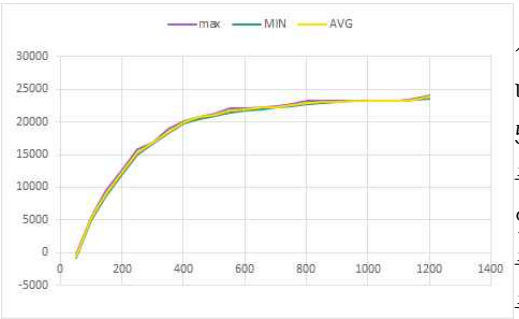
교체하는 인구수를 늘릴수록 안 좋은 값이 교체되어 값이 좋아질 줄 알았는데 오히려 100~150만큼 교체했을 때 가장 값이 좋았다. 충분한 다양성을 유지하면서 유리한 유전적 특성이 잘 퍼질 수 있도록 해주는 평균 품질이 가장 높은 150을 선택했다.

5) 내가 선택한 GA

- Selection: tournament
- Cross-over: uniform
- Mutation: random mutation (rate: 0.01)
- Replacement: generational GA(k/p = 1), 교체되는 population 수 150
- generation 수: 1200
- population 수: 500

데이터	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
chimera_946	20	27124	21260	24617	2128.71

[chimera_946, 세대 진행에 따른 population 분석]



다음 그래프는 chimera_946 데이터의 21번째 실행 결과이다. x축은 세대, y축은 cost(품질)를 나타낸다. 총 1200세대가 기록되었다. 초기의 50세대(-324)와 마지막 1200세대(24024)에서의 품질을 비교하면 지속적인 적합도 개선이 관찰되고 있다. 그러나 1000세대쯤에 이르렀을 때, 최고 품질과 최소 품질이 겹치는 것과 그래프의 상승 폭이 좁아지는 것을 볼 수 있다. 인구 내 해들이 비슷한 적합도를 갖는 해들로 수렴하고 있음을 의미한다. 이는 유전 알고리즘이 시간이 지남에 따라 유전 다양성이 감소하는 것을 보여준다. 다양성 보존을 위해 tournament 방식과 uniform 방식을 사용했음에도 다음과 같은 결과가 나왔다. tournament에서 더 좋은 해가 이길 확률을 낮추고, replacement 방식을 변경하면 다양성이 보존될 수 있을 것이다.

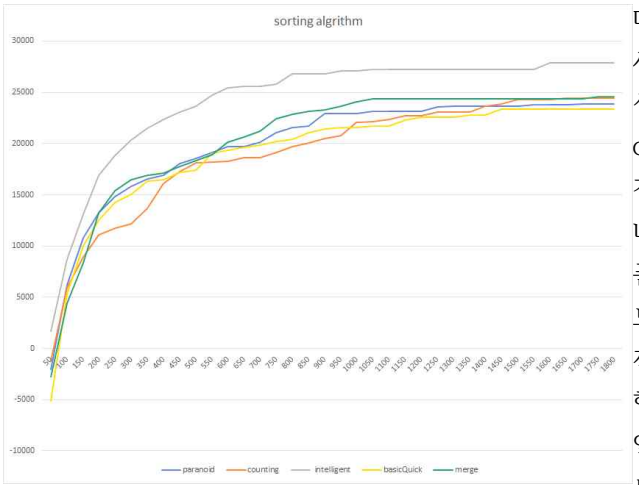
3. 5가지 정렬 알고리즘 비교 분석 과정, 결과

다음은 내가 선택한 GA에 데이터는 chimera_946, population 수 500, generation 수를 1800으로 설정하고, population을 해의 품질로 정렬한 후 run 함수 내에서 sorting 함수를 호출해 나온 값이다.

정렬 알고리즘	실행 횟수	최고 품질	최소 품질	평균 품질	표준편차
merge	5	25212	24780	24173.33	1176.72
BasicQuick	5	24780	23896	25204	1276.78
IntelligentQuick	5	27260	24196	25282.67	1400.47
ParanoidQuick	5	26940	23368	25241.33	1463.48
Counting	5	26712	24920	26080	821.34

- 다음 표를 통해 볼 때 counting sort는 평균 품질과 최소 품질에서 우수하며, 표준편차가 가장 낮아 가장 안정적인 결과를 보여준다. 그러나 가장 높은 최고 품질을 달성한 것은 intelligent quick sort이다. 품질 차이가 548이나 난다.

[chimera_946, 세대 진행에 따른 population 분석 - sorting]



다음 그래프는 chimera_946 데이터를 사용한 각 sorting 알고리즘의 6번째 실행 결과이다. x축은 세대, y축은 cost(품질)를 나타낸다. 총 1800세대에 걸쳐 각 정렬 알고리즘에 대한 cost를 나타내며 이것은 각 세대에서의 해의 품질을 의미한다. Merge sort는 가장 느린 초기 성장률을 보이지만, 세대가 진행됨에 따라 안정적으로 성능이 향상되고 있으며, 마지막에는 다른 알고리즘들과 유사한 수준의 품질에 도달하는 것을 볼 수 있다. basic quick sort는 시작할 때 낮은 cost로 시작했으나 시간이 지남에 따라 꾸준히 성장하는 것을 보여줬다. 그러나 다른 알고리즘에 비해 최종 품질이 약간 떨어지는 것을 확인할 수 있다. intelligent quick sort는 이 그래프에 나타난 알고리즘 중 최고의 성능을 보여준다. 고른 속도로 꾸준히 품질이 향상되고, 마지막 세대에 가장 높은 품질에 도달한 것을 볼 수 있다. paranoid quick sort는 초기에 빠른 성능 향상을 보인다. 이후로 성장 속도가 느려지긴 했지만, 꾸준히 품질이 향상되는 것을 볼 수 있다. counting sort는 전반적으로 안정적인 성장을 보여준다. 성장 속도가 가장 느리지만 마지막 세대에 가서는 다른 정렬 알고리즘들과 비슷한 수준의 품질을 달성했다. 모든 알고리즘이 시간에 따라 품질이 향상되는 것을 볼 수 있으며, 각기 다른 성장률과 안정화가 되는 시점을 갖고 있다. 결과에 따르면, intelligent quick sort가 일관되게 높은 성능을 유지하고 있으며, 특히 후반부에서는 다른 알고리즘에 비해 더 높은 품질을 달성했다. 또한 sorting을 하기 전과 후의 결과를 비교해 보면 품질 값이 상승한 것을 확인할 수 있다.

4. Discussion

과제2 코드에 키메라 데이터를 넣고 run 했을 때 코어덤프 에러가 발생하여 selection과 replacement 연산자를 수정했다. 코드가 돌아가긴 하는데 시작 품질과 끝 품질이 유사한 음수 값을 나타내어 이것의 원인이 음수 가중치를 고려하지 않고 cost 계산을 했기 때문이라 판단했다. evaluateFitness 라는 cost를 평가하는 함수를 가장 낮은 음수 가중치를 찾아 그 가중치의 절댓값만큼 모든 가중치에 더해주는 방식으로 수정했더니 수업 시간에 들었던 것처럼 그래프의 값이 완전히 달라졌다. longest를 찾는 것이라 생각하고 벨만포드를 구현하려 했는데 음수 cycle이 발생하여 계속 중간에 break 됐다. 결국 replacement 연산자를 열등한 population과 수정하는 방식으로 수정하고 해결할 수 있었다. 이를 통해 전체 population을 replacement 하는 것은 좋은 해를 얻지 못한다는 것을 알 수 있게 됐다.

참고자료: https://github.com/williamfiset/algorithms?tab=readme-ov-file, chatGPT