기본 유전 알고리즘

20201015 컴퓨터공학과 최유림

1. Genetic algorithm의 correctness와 efficiency

[Correctness]

Statement : 유전 알고리즘이 사용하는 연산은 모두 올바르게 구현되어 있음

Base Case: 초기의 세대에서 가장 좋은 해가 선택된다.

Inductive Step: 새로운 n개의 세대를 지나면 이전 세대보다 더 나은 해가 선택된다.

Inductive Hypothesis : 따라서 n+1개의 세대를 지나면 가장 좋은 해가 선택됨을 알 수 있다.

Conclusion: 유전 알고리즘은 최적 해에 수렴한다.

[Efficiency : O(N^2)]

유전 알고리즘 주요 단계의 시간 복잡도를 계산해서 총 시간 복잡도를 고려하고자 한다.

1. 초기화
   초기화는 초기 해 집단을 생성해야 하므로 총 population개(N)의 해를 만들어야 하고, 하나의 해는 gene으로 구성되어있기 때문에 gene의 개수(G)만큼 필요하다. 따라서 시간복잡도는 O(N * G)이다.

2. 선택
   선택의 과정은 룰렛 휠 알고리즘을 사용하는데, 총 N개의 해에서 2개를 룰렛의 크기에 따라 랜덤으로 선택하는 과정을 거친다. 이를 구현할 때 랜덤으로 숫자를 뽑고 어떤 룰렛의 범위에 들어가는지 확인하기 때문에 O(1)의 시간 복잡도를 가진다.

3. 교차
   one-crossover 방식으로 offspring의 해를 만든다고 생각했을 때, 2개의 부모 해에서 일정 값을 가져오고 합치는 과정이므로 O(1)의 시간 복잡도가 소요된다.

4. 변이
   희박한 확률 내에서 특정 gene이 random하게 바뀌는 과정이므로 조건을 확인하는 시간 O(1)과 특정 값을 바꾸는 시간 O(1)을 더해 총 O(1)의 시간 복잡도를 가진다.

2~4의 과정은 generation 수만큼 반복하는데 이를 K라고 하자. generational GA라고 생각했을 때 K는 보통 population의 값으로 설정한다. 따라서 K는 N이다. 2~4의 과정은 따라서 O(N), O(N), O(N)의 시간 복잡도를 가지게 된다.

5. 대치

   해 집단을 자식해로 대체하는 과정이다. Generational GA를 선택했을 때, generation 수만큼 대체되기 때문에 O(N)의 시간 복잡도가 소요된다. 따라서 O(N)의 시간복잡도를 가진다.

모든 과정의 시간복잡도를 합치면 O(N * G + N^2 + N + N + N)이므로 시간복잡도는 O(N^2)으로 볼 수 있다.

2. 해의 표현

해당 문제는 집합 S와 S′으로 나누어 둘을 잇는 간선의 가중치 합을 최대화시키는 문제이다. 따라서 해는 주어진 노드를 집합 S와 S′으로 나누어서 표현할 수 있게끔 구성했다. 노드가 S에 속하면 1이라는 값을 가지고, S′에 속하면 0이라는 값을 가지게 하여 각각의 gene을 집합 관계에 따라 binary한 값을 가질 수 있도록 구성했다. 한 개의 노드가 하나의 gene으로 표현되기 때문에 하나의 유전자는 노드 개수만큼의 길이를 가진다.

따라서 S = {1,6,8,9}, S′ = {2,3,4,5,7,10}이라면 1000010110으로 표현된다.

3. 사용한 연산자에 대한 설명

selection : 룰렛 휠 알고리즘을 사용한다. 해당 알고리즘은 각각의 해의 fitness 값을 구하여 값에 따라 룰렛의 면적을 정하고, 랜덤값을 발생시켰을 때(다트 던지는 경우) 룰렛의 어떤 면적에 속하는지 확인하여 부모 해를 선택한다. 적합도 값을 구하는 적합도 함수는 calcFitness 에서 수행하고 있다. 해당 함수는 적합도 계산 식에 따라 가장 cost가 낮은 해와 cost가 높은 해의 cost값을 요구하므로, cost에 따라 해의 집단을 정렬하는 과정을 먼저 수행하고 있다. 적합도 함수에서 selection pressure는 3으로 선택했다. selection pressure는 작을수록 열등한 해도 선택되는 기회가 생기기 때문에 3으로 선택하게 되었다.

crossover : 1-point crossover를 선택했다. 선택된 부모 해 중 첫번째로 선택된 것의 절반의 gene과 두번째로 선택된 것의 절반의 gene을 합쳐서 자식해를 생성하고 있다.

mutation : 돌연변이 확률을 0.015로 선택하여 수행하게끔 했다. 랜덤으로 확률을 생성하고 0.015 내의 확률을 가지면 자식 해 중 하나의 gene을 랜덤하게 골라 반대의 값을 가지게 하고 있다.(0의 값을 가지면 1을, 1의 값을 가지면 0을 가진다)

replace : generation gap을 0.6으로 유지하면서 Genitor-Style 방식으로 구현했다. 예를 들어 1000개의 해 집단에서 다음 세대의 자손 600개를 생성하여 부모해 집단에서 가장

안 좋은 600개를 대체하는 것이다.

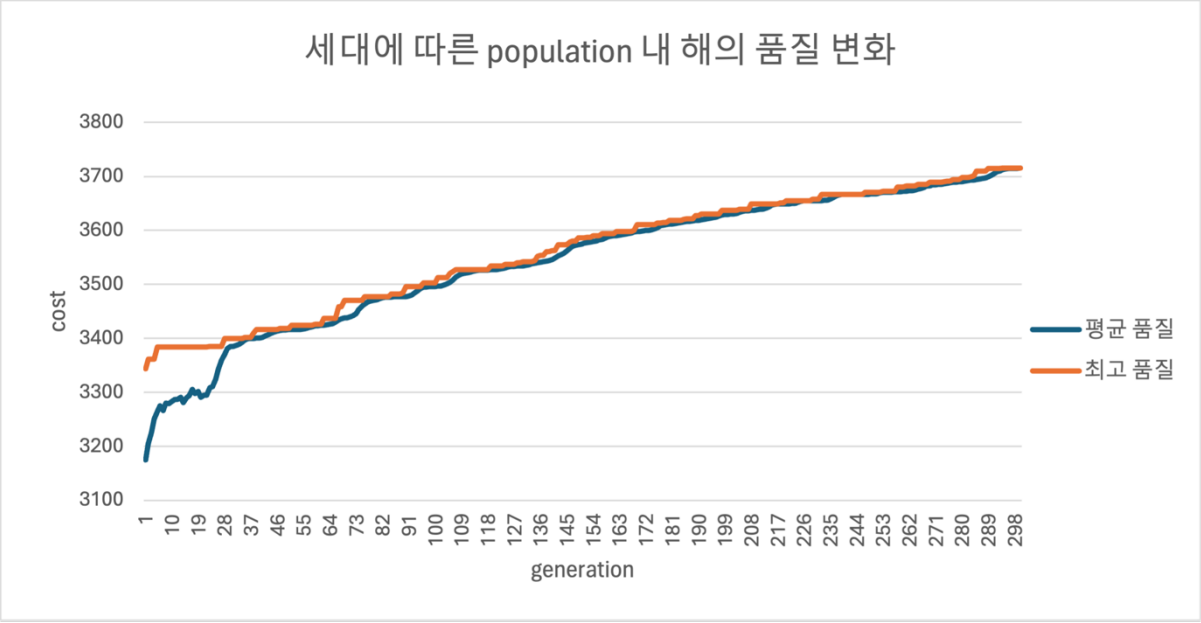
4. 세 개의 샘플 인스턴스에 대한 최소 30번의 GA 수행 결과

	가장 좋은 결과	평균 결과	표준편차
Unweighted_50	96	90.4666	3.1699
Unweighted_100	334	327.5333	4.7939
Weighted_500	3753	3676.5	38.5008

각각의 인스턴스에 대해 30번씩 수행해본 결과 위와 같은 표를 얻을 수 있었다. 그래프의 노드 수가 적은 인스턴스의 경우에는 표준편차가 3~4 정도로 안정적인 결과를 도출했지만, Weighted_500과 같이 노드 수가 많은 인스턴스의 경우에는 표준편차가 38까지 올라가는 것을 확인할 수 있다.

이러한 차이에 대한 원인을 분석해보자면, 노드 수는 결국 해의 길이를 의미하므로 해의 길이가 길어질수록 도출하는 cost값의 범위가 넓어진다는 것을 추측할 수 있다. 이러한 변화에 대한 가장 큰 원인은 cross-over에 있을 것으로 예상한다. 해당 코드에서는 one point cross over 방식을 사용중인데, 해의 길이가 길어질수록 하나의 지점을 기준으로만 크로스 오버하는 방식은 더 다양한 자손의 형태를 만드는 것이 어렵기 때문일 것으로 추측해볼 수 있었다.

5. Weighted_500 인스턴스 수행시 세대 진행에 따른 population 분석



500개의 노드를 가진 그래프의 인스턴스로 GA를 실행했을 때의 세대별 population 분

석 그래프이다. 평균 품질의 측면을 보면 세대수가 30일 때까지, 최고 품질에 비해 많이 떨어지는 것을 확인할 수 있다. 그러나 빠른 속도로 수렴해 나가는 것 또한 확인할 수 있다. 30개의 세대 이후부터는 평균품질이 최고품질과 큰 차이가 나지 않는다.

최고품질 그래프는 계속 완만한 상승 곡선을 그리며, 세대에 따라 해의 품질을 개선하고 있다는 것을 보여주고 있다. 마지막 세대수에 가까워질수록 점점 더 완만한 곡선을 그리며 cost 최댓값에 수렴하고 있다.

6. Discussion

수업 시간에 개념과 간단한 코드 구조만을 봤을 때는 이해가 안 가는 부분이 많았는데, 직접 많은 부분을 고민하고 코드를 작성해보며 유전 알고리즘을 이해할 수 있게 된 것 같다. 제출한 코드와는 달리 많은 실패도 있었는데, 특히 값이 수렴되지 않은 채로 계속 안 좋은 cost 값을 도출하여 어떤 부분에 문제가 있는지 탐색해보는 과정을 가졌다. 처음에는 부모 세대를 완전히 새롭게 만들어진 자식으로 모두 대체하는 방식을 도입했는데 해당 방식으로는 거의 수렴하지 않는 것을 알게 되었고, replace에서 generation gap의 변화를 주어 이를 해결했다. 또한 시간 제한이 있다보니 노드 수가 많은 인스턴스의 경우에는 원하는 수치까지 제대로 수렴하지 않아 아쉬움이 남았다.

의외의 현상으로는 작은 노드 수의 경우에 오랜 시간 유전 알고리즘을 돌릴 경우 제대로 수렴하지 않는 것을 확인할 수 있었다. 해당 현상에 대한 원인을 고민해봤다. local attractor에 끌려서 그런 것일수도 있겠다는 생각을 했고, 시간을 꽉 채워서 실행하는 것보다는 세 개의 인스턴스 모두 적당한 cost 값을 도출할 수 있는 실행 횟수를 조절하여 while 종료 조건을 도입했다. While 종료 조건은 다양한 방식을 고민해봤는데, 처음 도입한 것은 cost가 특정 세대수를 거쳤음에도 불구하고 더 발전하지 않을 때 종료하는 것이었다. 그러나 해당 방법은 노드 수가 적은 인스턴스에는 최적의 경우였지만, 노드수가 500인 경우에는 타임오버가 발생하여 도입할 수 없었다. 두 번째는 시간을 제한했지만, 앞에서 언급한 이유로 제대로 수렴하지 않아 도입할 수 없었다. 마지막으로는 실행 횟수를 임의로 조절한 경우를 도입했고, 모든 예시에 가장 적합하다고 생각하여 해당 방법으로 코드를 제출하게 되었다.

이렇게 유전 알고리즘을 짜본 것은 처음이라 잘 안되는 부분이 많았지만 부족한 부분은 검색해보고 찾아보며 해결해보려고 노력한 점이 뿌듯한 것 같다. 또한 이 기회를 통해 자바의 자료구조를 다양하게 활용해볼 수 있어서 좋았다.