유전 알고리즘 기반 Max-cut 문제 해결 및 해의 품질 정렬 알고리즘 비교

20201015 컴퓨터공학과 최유림

1. 과제2 비교 수정사항 및 이유

A. 코드 내 cost, fitness, weight 용어 확립

기존의 코드를 작성할 때, cost를 한 개의 해가 가질 수 있는 가중치 총합으로 정의하여 사용했다. Max-cut의 문제는 가중치 총합이 클수록 좋은 해이므로, cost가 클수록 좋은 해임을 가정하고 코드를 작성했다. 그러나 cost 용어에 대한 정의는 해를 구하기 위한 비용이므로, cost는 작을수록 좋은 해이다. 기존 코드에서는 cost가 클수록 fitness 식에 따라 fitness가 작게 나오는 문제가 있어 cost가 클수록 fitness가 크게 나올 수 있도록 식을 변형해서 사용했다. 이러한 용어에 대한 정의를 확립하고, 코드를 재작성했다. 따라서 해가 가진 weight의 합을 나타내는 sumWeight가 클수록 cost는 작아지고, fitness는 커지는 코드가 되었다.

B. 그래프 edge 저장 방법 변경

기존 코드를 작성할 때는 그래프의 간선을 ArrayList에 순서대로 저장했다. 따라서 특정 노드에 연결된 간선을 찾기 위해서는 모든 간선을 순회해야지만 특정 노드에 연결된 간선 리스트를 찾아낼 수 있었다. 이러한 자료구조의 선택이 sumWeight를 찾는 시간을 증가시키는 것을 알 수 있었다. 따라서 자료구조의 변경이 필요할 것 같다고 생각하여 HashMap을 도입했다. 시작점을 key로 가지고 있어 전체 간선을 순회하지 않아도 시작점과 연결된 간선들의 정보를 바로 가져올 수 있다. 해당 자료구조로 변환해보니 훨씬 더 빠른 속도로 연산이 진행됨을 확인할 수 있었다. 또한 시간 내에 큰 데이터도 충분히 수렴 과정을 거칠 수 있게 되었다.

C. Uniform Crossover 방식 도입

기존 코드는 1 point Crossover 방식을 사용했다. 기존 방식으로 세 개 샘플 인스턴스를 비교한 결과에서 그래프의 노드가 많은 인스턴스의 경우, 최적의 해 결과 표준편차가 38까지 올라가는 것을 확인할 수 있었다. 당시 추측한 원인으로는, 노드 수는 결국 해의 길이를 의미하므로 해의 길이가 길어질수록 하나의 지점을 기준으로만 크로스 오버하는 방법은 자손의 다양성 보존이 어렵다는 점이었다. 따라서 해당 방식으로부터 Uniform Crossover 방법으로 변화를 택했다.

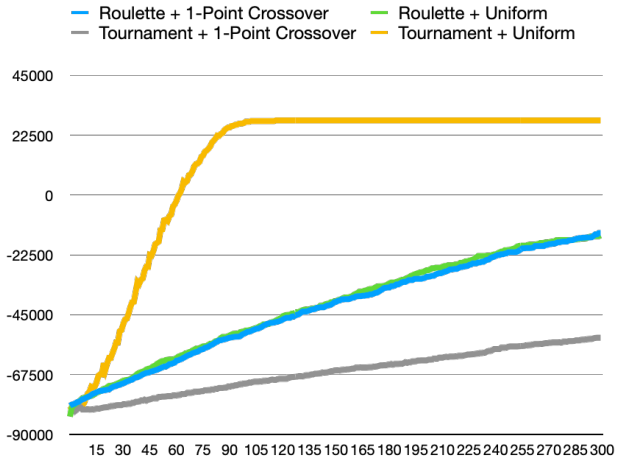
D. Chimera_946 인스턴스에 대한 해결 방법 고안

기존 코드로는 시간 내에 양수의 weight값을 얻어낼 수 없다는 것을 알 수 있었다. 수렴 속도의 문제가 있다는 것으로 파악했는데, 다양한 개선 방식을 고안해봤다. 방금 언급한 Crossover 방식의 변화가 해당 해결 방법 중 하나가 될 것이라고 생각했다. 또한 기존 룰렛 휠 선택 방식에서 토너먼트 선택 방식으로의 변화도 주었다.

기존의 코드에서 변경한 방식을 도입했을 때의 변화를 살펴보기 위해, 선택 연산 두 개와 Crossover 방식 두 개를 조합한 총 4개의 방식을 직접 실험해 봤다.

나머지 연산은 모두 동일하며, Generaitonal GA 비율을 1로 고정한 채 세대별 해의 변화를 살펴봤다.

연산 방식에 따른 총 300개 세대의 해의 변화를 나타낸 그래프이다. Roulette Wheel Selection 방식을 이용한 두 가지의 경우는 거의 비슷한 성능을 보이는 것을 알 수 있으나 Tournament Selection은 크로스오버 방식에 따라 큰 차이를 보였다.



왜 Tournament 방식과 1 point Cross Over 방식을 조합해서 사용하면 최악의 성능이 나오는지에 대해서 생각해보았다. Tournament Selection 방식은 Roulette Wheel Selection 방식과 다르게 적합도에 따라 선택되지 않고 랜덤 방식으로 선택된다. 따라서 데이터를 보면 알 수 있듯이 초기의 세대에서는 수렴하지 않고 해가 좋아졌다가 나빠지기도 하는 모습을 보여주기도 한다. 그러나 수렴 방향을 잡게 되면 다른 선택 방식보다 훨씬 더 빠르게 수렴하는 것을 확인할 수 있었다. 1 point crossover 방식을 사용하게 되면 초기의 해에서 좋은 gene을 버리게 되어 최악의 성능을 가질 수도 있다는 가능성을 생각하게 되었다. 그러나 Uniform Crossover는 각각의 Gene이 독립적으로 선택되며 다양성을 유지할 수 있기 때문에 Tournament 방식과 시너지를 낸다고 생각할 수 있었다. 이러한 다양성을 기반으로 랜덤 선택하여 수렴 방향을 잡는다면 더 좋은 해로의 수렴 방향을 잡을 수 있다.

2. 가장 좋은 GA 선택 과정 및 선택한 연산자

A. 선택 과정

'가장 좋은'이라는 조건의 의미부터 고민하였다. 유전 알고리즘은 세대를 거칠수록 좋은 결과를 내지만, 가끔 지역해에 이끌려 최적의 해에 도달하지 못하거나 특정 결과에 도달하기까지 시간도 오래 걸린다. 특히 수행 시간과 가장 좋은 해 도출은 반대의 관계를 가진다. 데이터의 크기가 클수록 오랜 시간이 소요된다. 따라서 제한시간이라는 조건이 걸려있다면 데이터의 크기에 상관없이 시간 내에 최적의 답안을 도출하는 것이 가장 중요하다고 생각했다. 따라서 이런 조건에 가장 부합하는 GA를 선택하게 되었다.

A. 선택 결과

해당 기준으로 비교한 주요 연산자를 소개하고자 한다.

selection : 토너먼트 선택 방식을 사용한다. 처음 구현할 때에는 population 중 2개를 랜덤 선택하여 토너먼트를 진행했다. 100개의 노드를 가질 때에는 문제없이 좋은 성능을 보였으나 500개, 946개의 데이터와 같이 큰 데이터들의 경우에는 거의 수렴하지 못하는 모습을 보였다. 왜 이러한 경향을 보이는지 고민해보았을 때, 선택되는 확률의 문제라고 생각했다. 100개 중의 2개와 500개 중의 2개는 확률 차이가 있다. 좋은 해가 선택될 수 있는 확률이 너무 적어져서 문제라고 생각하였다. 따라서 토너먼트에 참여할 해의 개수

를 6개로 늘렸다. 현재 해당 연산은 랜덤으로 6개의 해를 추출하고, 랜덤 확률이 0.7 안에 들 경우 가장 좋은 해를 반환하고 그렇지 않은 경우 토너먼트에 참여한 해 중 랜덤으로 선택된다. 위에 언급한 실험 결과는 이 방식을 도입한 것으로 해당 선택 방식을 도입하는 경우, 더 빠른 수렴 속도를 보이는 것을 확인했다.

crossover : Uniform Crossover 방식을 선택했다. 각각의 Gene은 Random 확률에 따라 부모 1의 유전자를 받기도 하고, 부모 2의 유전자를 받기도 한다. 해당 방식으로 진행하게 되면 Gene은 서로 독립성을 가져 다양성을 유지할 수 있다. 해당 방식과 토너먼트 선택 방식을 함께 사용하면 훨씬 더 좋은 수렴 속도를 가지는 것을 확인할 수 있었다.

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 1로 고정한 Genitor-Style 방식으로 구현했다. Population이 500개로 고정되어 있기 때문에 자손 또한 500개를 생성하고, 하나의 세대가 끝나면 모든 population이 새롭게 생성된 해로 모두 대체된다. 저번에 작성한 코드는 0.6의 비율을 사용했는데 1로 변경하니 수렴 속도가 더 느려지는 것을 확인할 수 있었다. 아마 0.6일 때는 가장 안 좋은 해만 대체할 수 있었기 때문에 더 빠른 수렴을 할 수 있었던 듯하다.

3. 5가지 정렬 알고리즘을 GA를 이용해 비교 분석한 과정과 결과

A. GA 내에 적용한 Sorting 부분

- 기존 코드에서 Sorting 사용하던 부분은 세 곳이었다. 룰렛 휠 연산을 위한 적합도를 계산하기 위해 정렬하는 부분과 Generational-Gap을 0.6으로 설정하여 안 좋은 해 60%를 알아내기 위해 정렬하는 부분, 마지막으로 모든 세대를 거치고 나온 최적 해를 알아내기 위한 부분으로 이루어져 있었다. 그러나 이번 과제 과정을 거치며 토너먼트 선택 연산으로 변경되고, Generational-gap 또한 1로 고정되며 Sorting 연산이 빠지게 되었다. 따라서 마지막으로 최적 해를 알아내기 위한 부분의 코드로 5가지 정렬 알고리즘을 비교하게 되었다.

B. 초기 아이디어

- 하나의 세대의 해를 모두 확인해본 결과, 중복된 데이터가 많음을 알 수 있었다. 수렴하는 단계에서 바로 수렴하지 않는 구간이 있기 때문이다. 또한 작은 데이터의 경우 많이 수렴된 상태에서 정렬이 진행되기 때문에 데이터의 범위도 넓지 않다. 따라서 이러한 데이터를 정렬할 경우 Counting Sort가 제일 적합할 것 같다고 생각했다. Counting Sort는 동일한 데이터들의 개수를 세서 정렬하는 것이기 때문에 중복 데이터가 많은 데이터에 적합하기 때문이다.

C. Sorting 비교

아래의 표는 961개의 노드를 가지는 그래프 instance를 수행한 해를 Sorting하는데 걸린 시간을 비교한 것이다. 각각의 Sort 방법에 따라 네 번씩 반복하였다.

Generation : 55, Node : 961	1st	2nd	3rd	4th
Counting Sorted	2ms	2ms	2ms	2ms
Merge Sorted	1ms	1ms	1ms	1ms
Quick Sorted	2ms	1ms	2ms	2ms
Intelligent Quick Sorted	2ms	2ms	2ms	2ms
Paranoid Quick Sorted	1ms	1ms	2ms	1ms

초기에 생각했던 것과 달리 Counting Sort는 네 번의 시도 내내 계속 좋지 않은 성능을 보였다. 이유를 찾기 위해 마지막 세대의 해를 살펴본 결과, 작은 데이터와 달리 큰 데이터의 경우에는 중복된 데이터가 거의 없고, 완벽한 수렴의 결과가 아니라 데이터의 범위가 넓기 때문이라고 예측했다. Counting Sort는 데이터의 크기에 영향을 받기 때문에 최적의 Sorting 방법에서 제외했다.

또한 Intelligent Quick Sort 역시 네 번의 시도 내내 좋지 않은 결과를 보였다. 그 이유로는 아마 pivot의 값을 median of medians를 통해 구하는 과정이 데이터 크기가 큰 경우에 오래 걸리기 때문일 것이라고 예상했다. Quick Sort는 pivot으로 결정되는 첫 번째 데이터가 어떤 값을 가지는지에 따라 다른 결과를 보이지만, 안 좋은 성능을 더 많이 보여 최적의 방법에서 제외했다.

그렇다면 해당 문제에 가장 적절한 해의 후보로는 Merge Sort와 Paranoid Quick Sort가 있다. Merge Sort는 꾸준히 좋은 성능을 보였으나 in-place 방식이 아니기에 데이터 크기에 따라 데이터 메모리 문제를 발생시킬 수 있을 것 같다는 생각이 들었다. 따라서 데이터의 크기에 상관없이 일관된 성능을 보여주고, 추가 메모리를 사용하지 않는 Paranoid Quick Sort를 선택했다.

4. Discussion

처음 시간 제한을 생각했을 때는, 빠른 수렴 속도로만 해결하려고 했으나 여러 번 직접 실행을 해보며 문제는 연산 속도에 있다는 것을 깨달을 수 있었다. 하나의 GA 연산을 수행할 때 너무 오랜 시간이 걸린다는 것을 알게 되고, 어떤 자료구조가 적합할지 고민하는 시간을 가지게 되었다. 자료구조 하나의 변화가 코드 전체에 큰 영향을 미친다는 것을 체감할 수 있는 시간이었다.

또한 해당 과제를 하며 정말 다양한 방법을 시도해봤고, 그 중에 가장 좋은 방법을 찾아낼 수 있었던 것 같아 뿌듯하기도 하다. 연산 방법을 조합해서 총 4개의 방법을 실험해 보았다. 또한 토너먼트도 2개만 경쟁하는 것에서 4개, 6개, 8개 등등 여러 번의 시도를 했고, 적은 데이터의 경우에는 너무 많은 후보군이 있을 경우 제대로 수렴하지 않는 것을 확인하여 6개가 가장 적절하다는 것을 알아낼 수 있었다. 이를 통해서 유전 알고리즘은 연산 방법도 다양하지만 특정 연산 내에 parameter들도 중요하다는 것을 깨달을 수 있는 시간이 되었다. 또한 생각해내는 힘을 기를 수 있었다. 데이터의 개수, 연산 방법, 파라미터 등에 따라 나오는 결과를 단순히 받아들이는 것 뿐만 아니라 왜 이런 결과가 나왔을지 고민하는 시간을 많이 가졌다. 이러한 과정이 유전 알고리즘에 대한 더 깊은 이해를 가져올 수 있지 않았나 라는 생각이 든다.