Haskell에서 외판원 문제를 해결하기 위한 진화형 알고리즘 사용 및 평가

정재용[O1], 변석우[2], 우균[34*]
[1]부산대학교 전기전자컴퓨터공학과, [2]경성대학교 컴퓨터공학과,
[3]부산대학교 전기컴퓨터공학부, [4]LG전자 스마트제어센터
jjysam319@pusan.ac.kr, swbyun@ks.ac.kr, woogyun@pusan.ac.kr

Using and Evaluating Evolutionary Algorithm for Solving Traveling Salesman Problem in Haskell

JaeYong Jeong[O1], Sugwoo Byun[2], Gyun Woo[34*]
[1]Department of Electrical and Computer Engineering, Pusan National University,
[2]School of Computer Science & Engineering, Kyungsung University,
[3]School of Electrical and Computer Engineering, Pusan National University,
[4]Smart Control Center of LG Electronics

요 약

Haskell은 일반적이고 순수한 함수형 프로그래밍 언어이다. Haskell은 절차형 언어처럼 구문 순서에 따라서 해당 순서가 되면 해당 라인에 있는 계산을 무조건 실행하는 것이 아니라, 해당하는 값이 필요한 순간에 계산하는 '느긋한 계산법'을 사용한다. '느긋한 계산법'으로 인해 동일한 기능을 반복하거나 공유할 때에 빠른 처리 속도를 보여준다. 따라서 이러한 장점이 실제 문제 해결에서 유용한 지를 확인하기 위해 동일한 계산이 반복되는 진화형 알고리즘을 사용한 외판원 문제 해결 프로그램을 Haskell과 C++로 각각 작성하였다. 그리고 집단의 크기와 진화 횟수에 따라 비교실험을 했다. 실험 결과 집단의 크기와 진화횟수가 4배일 때 C++가 Haskell보다 각각 5배, 1.3배 높은 증가폭을 보였다. 실험결과를 통해서 진화형 알고리즘과 같이 유사한 형태의 계산이 반복되는 문제를 해결하는 데 있어서 Haskell이 유용함을 알 수 있다.

1. 서 론

Haskell은 1990년에 완성된 함수형 프로그래밍 언어이다. Haskell의 특징은 C, C++, Java와 같은 절차형 프로그래밍언어와 달리 반복문을 이용한 순차적 구조가 아니라 수많은 작은 함수들로 이루어지며 이 과정에서 느긋한 계산법(lazy evaluation)을 쓴다는 점이다[1]. '느긋한 계산법'은 절차형 언어처럼 구문 순서에 따라서 그 순서가 되면 무조건 계산하는 것이 아니라 해당하는 연산의 결과가 필요한 순간이 되어서야 계산한다. 따라서 동일한 기능을 반복하거나 공유할 때에 불필요한 계산을 줄일 수 있다. 본 논문에서는 Haskell의 이러한 장점을 통해 진화형 알고리즘과 같이 많은 양의 동일한 형태의 계산을 반복해서 해야 하는 문제에서 처리 시간이 급격히 늘어나는 문제를 해결할 수 있음을 보이고자 한다.

진화형 알고리즘은 최적화 문제에 주로 쓰인다. 때로 유전자의 길이가 길어지고 많은 세대를 거쳐야만 원하는 결과를 얻을 수 있어서 많은 시간적 비용이 드는 경우가 많다. 하지만 진화형 알고리즘에서 사용되는 대부분의 연산은 유사한 형태의 계산을 n번 반복하는 형태이다. 따라서 Haskell을 이용해서 작성한다면 유전자의 길이가 길어지고 세대가 커짐에 따라서 증가하는 시간 비용의 증가 폭이 크지 않으리라고 기대한다. 진화형 알고리즘을 적용하기 위한 대상으로 가장 널리 알려진 최적화 문제인 외판원 문제(Traveling Salesman Problem)를 대상으로 실험한다. 사용한 진화형 알고리즘은 진화 전략(Evolution Strategy)($\mu + \lambda$)[2]이다.

실험은 같은 알고리즘으로 Haskell과 C++로 작성한 프로그램에 대해서 집단의 크기와 진화 횟수를 조정해가면서 실행시간을 측정하는 것이다. 이를 통해 집단의 크기와 진화 횟수에 따른 실행 시간 변화의 크기를 비교해보고자 한다. C++ 프로그램은 주로 리스트를 사용하는 Haskell 프로그램과 최대한 유사한 구조를 이루게 하려고 유전자 표현을 위한 자료 구조로 STL의 List를 사용하였다.

2장에서는 관련 연구로 외판원 문제와 진화 전략($\mu + \lambda$)에 대한 설명과 각각에 관한 연구들을 담고 있다. 3장에서는 실제 작성한 Haskell과 C++ 프로그램에 대한 설계 내용을 담고 있다. 4장에서는 두 프로그래밍 언어로 작성한 프로그램을 비교 실험한 결과를 담고 있다. 5장에서는 결론 및 향후 연구과제에 대해서 다룬다.

2. 관련 연구

2.1 외판원 문제

외판원 문제는 외판원이 출발지에서 다른 모든 도시를 순서에 상관없이 순회 방문하고 돌아오는 문제이다. 중요한 것은 같은 도시를 다시 방문하지 않고 최단 이동 시간으로 순회해야 한다는 점이다. 일반적으로 비방향성 가중치 그래프에서의 해밀턴 경로를 구하는 문제[3]로 표현할 수 있다. 외판원 문제의 시간 복잡도는 기본적으로O(n!)이기 때문에 들려야 하는 도시 수가 많아지면 답을 구하는 것이 사실상 불가능하다. 따라서 진화형 알고리즘 등을 통해서 근사한 답을 구해서 사용한다. 외판원 문제를 해결하기 위한 연구로는 협동 학습을 이용한 해결 방법[4], 휴리스틱 알고리즘을 이용한 해결방법[5], 최소 묶음 트리(MST: minimum spanning tree)를 이용한 해결방법[6] 등이 있었다.

2.2 진화 전략

진화 전략(μ+λ)은 1973년 Rechenberg에 의해서 제안된 진화형 알고리즘의 일종이다. 그 알고리즘은 그림 1과 같다. 전체적인 구성은 유전자 알고리즘과 같은 일반적인 진화형 알고리즘과 유사하다.

진화 전략(μ + λ)-ES: (Rechenberg, 1965)
1. μ개의 개체를 임의로 생성하고 이들의 적합도를 평가한다.
2. λ개의 개체가 생성될 때까지 아래를 반복한다.
   - 임의로 2개의 개체를 복원 추출한다.
   - 크로스 오버로 새로운 개체를 생성한다.
   - 돌연변이를 일으킨다.
3. λ개의 새로운 개체의 적합도를 계산한다.
4. μ개의 현재 세대와 λ개의 새로운 개체 중 μ개의 다음 세대를 뽑는다.
5. 종료조건을 만족하지 않으면 2로 돌아간다.

(그림 1) 진화 전략 알고리즘

부모 선택에서 진화 전략은 적합도와 상관없이 임의로 고른다. 또한, 자식을 생성한 후 다음 세대를 고를 때 우리가 사용한 진화 전략(μ+λ) 알고리즘은 현재 세대(u)와 생성한 개체(λ) 중 다음 세대가 될 적합도가 높은 λ개의 개체를 고른다. 따라서 진화 전략(μ+λ) 알고리즘은 부모 생성 부분에서는 폭넓게 접근(diversify)하며 후대 결정 부분에서는 강하게 접근(intensify)한다. 이에 관한 연구[7-8]도 다양하게 진행되고 있다.

3. 시스템 설계

두 프로그램의 전체적인 설계 방법은 같다. 정확한 비교를 위해서 동일한 유전자 표현과 동일한 평가 함수를 사용한다. 그밖에 세부적인 알고리즘 또한 일치한다. 다음은 각 부분의 설계 내용이다.

3.1 유전자 설계

테스트를 위한 유전자 표현은 임의키 코드 방법[9]을 사용하였다. 각 유전자는 0에서 1 사이의 임의의 실수가 된다. 그림 2는 임의키를 사용한 염색체의 크로스 오버 예제이다.

(0.9501, 0.2311, 0.4568, 0.4860, 0.8913, 0.7621) (2, 3, 4, 6, 5, 1)
(0.4565, 0.0185, 0.8214, 0.6154, 0.4447, 0.7919) (2, 5, 1, 4, 6, 3)

⬇

(0.9501, 0.2311, 0.4568, 0.6154, 0.4447, 0.7919) (2, 5, 3, 4, 6, 1)
(0.4565, 0.0185, 0.8214, 0.4860, 0.8913, 0.7621) (2, 1, 4, 6, 3, 5)

(그림 2) 임의키를 사용한 염색체의 크로스 오버

왼쪽은 유전자 표현이며 오른쪽은 각 도시를 들리는 순서이다. 각각의 임의키는 각 도시에 대응하는 임의키이며 임의키가 작은 도시부터 들린다고 정의한다. 그림2의 첫 번째 염색체와 같이 만약 염색체가 (0.9501, 0.2311, 0.4568, 0.4860, 0.8913, 0.7621)로 이루어져 있다면 가장 작은 숫자인 0.2311이 두 번째 자리에 있으므로 2번 도시를 가장 먼저 들리게 되고, 다음으로 작은 0.4568이 세 번째 자리에 있으므로 3번 도시를 두 번째로 들리게 된다.

3.2 부모 선택

그림 1에 나와 있는 진화 전략(μ+λ)의 알고리즘과 같이 임의로 두 개의 부모 염색체를 랜덤하게 복원 추출한다. 그리고 이를 크로스 오버 및 돌연변이 해서 자손을 생성한다. 단, 이때 생성하는 자손의 수는 초기 집단의 수와 항상 일치하도록 조정하여서 구현하였다(μ=λ). 따라서 실험을 위해서 초기 집단의 수를 증가시키면 생성하는 자손의 수인 λ도 같이 증가하게 된다.

3.3 크로스 오버 및 돌연변이

최대한 확률적으로 발생하는 연산 시간의 차이를 줄이기 위해서 크로스 오버와 돌연변이는 간단하게 구현하였다. 크로스 오버는 항상 절반이 되는 지점을 기준으로 크로스 오버를 시행하게 된다. 또한, 돌연변이는 1/2 확률로 염색체의 첫 번째 유전자를 맨 뒤로 보내도록 구현을 하였다. 이를 통해 전체적인 방문 순서가 바뀌는 효과가 있다.

3.4 평가함수 및 세대 결정

다음 세대를 결정하기 위해서 적합도를 이용한다. 적합도를 구하기 위한 평가 함수는 각 도시를 들리는 데 걸리는 시간으로 구할 수 있다. 각 도시에서 다음 도시로 이동하는 소요시간이 미리 N x N 형태로 저장되어 있다. 먼저 염색체마다 도시 방문 순서, 즉 표현형을 구한다. 그리고 표현형을 통해서 해당 순서로 도시를 거칠 때의 소요시간의 합을 구한다. 따라서 적합도(fitness) 값이 작을수록 좋은 유전자가 된다. (u+λ)개 중에서 가장 우수한 u개를 선정하여 다음 세대로 결정한다.

3.5 자료구조 및 시스템 분석

Haskell 프로그램에서는 별도의 Array 패키지 라이브러리 등을 사용하지 않고 기본적인 리스트를 사용해서 모든 유전자를 표현하였다. 따라서 C++ 프로그램에서 또한 동등한 비교를 위해서 STL의 List를 사용해서 이를 구현하였다. 앞서 설명한 대부분 기능은 O(n) 안에서 모두 이루어진다. 하지만 평가 함수를 사용하여 다음 세대를 결정하는 부분에서는 적어도 정렬 수준의 시간 복잡도를 갖는 연산이 필요하므로 O(nlogn)에서 최악의 경우에 $O(n^2)$의 시간 복잡도가 발생한다. 따라서 시스템의 시간 복잡도는 O(nlogn) ~ $O(n^2)$으로 예상할 수 있다.

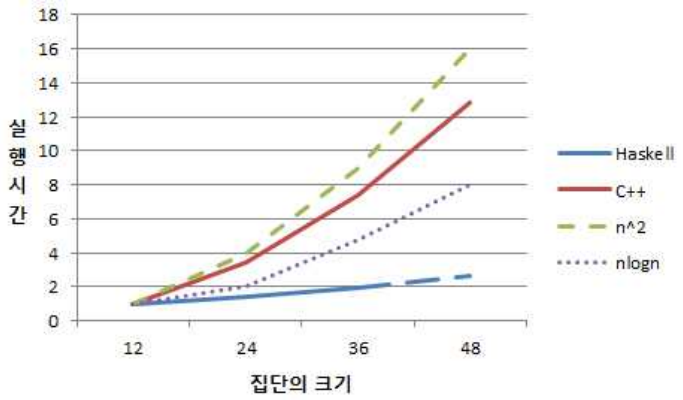
4. 비교 실험

실험의 가정은 C++로 작성한 프로그램은 설계상에서 예상한 시간 복잡도인 O(nlogn) ~ $O(n^2)$에 근접할 것이며 Haskell로 작성한 프로그램은 동일한 형태의 연산이 반복되기 때문에 비슷한 구현에도 n이 증가했을 때 이보다 훨씬 적은 실행시간의 증가를 보이리라는 것이다. 실험은 Intel i7-4770 Quard Core Processor, 8GB RAM에서 이루어졌다. 염색체의 크기는 항상 12로 고정해서 실험을 진행하였다. 표1은 집단의 크기를 다르게 했을 때 Haskell로 작성한 프로그램과 C++로 작성한 프로그램의 실행시간을 나타낸 것이다.

(표 1) 집단의 크기에 따른 실행 시간 비교표

집단의 크기	Haskell	C++
12	1.9	72.2
24	2.7	248.3
36	3.7	532.0
48	5.0	926.4

표 1을 살펴보면 C++ 프로그램과 비교하면 Haskell 프로그램의 실행 시간이 적게 걸린다는 것을 알 수 있다. 이는 Haskell 프로그램은 Haskell에 맞게 효율적인 프로그램을 작성하였지만, C++ 프로그램은 Haskell 프로그램과 유사하게 작성하기 위해서 STL의 List를 사용하는 등 비효율적인 구조가 되었기 때문이다. 따라서 단순한 실행 시간이 아니라 변화율을 기준으로 비교해보았다. 그림 3은 집단의 크기를 다르게 했을 때 집단의 크기에 따른 실행시간 변화를 나타낸다. 표시된 값은 Haskell, C++, $O(n^2)$인 프로그램, O(nlogn)인 프로그램에 대해서 각각 n(집단의 크기)이 12일 때의 실행시간 값을 1로 잡고 n을 증가시키면서 실행시간의 비를 구한 것이다.



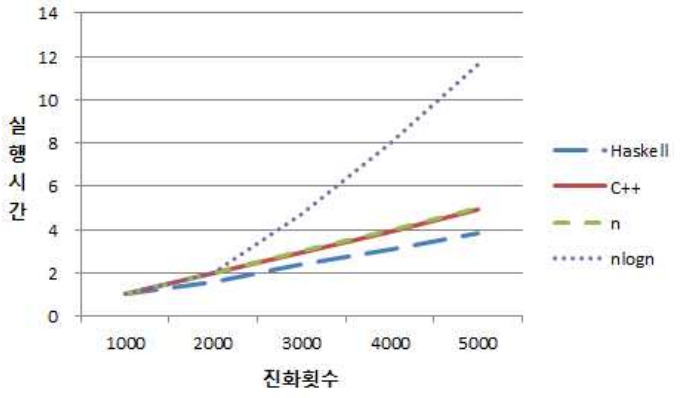
(그림 3) 집단의 크기와 실행시간 간의 그래프

그림 3과 같이 집단 크기의 증가에 따른 영향을 Haskell 프로그램이 덜 받고 있는 것을 볼 수 있다. Haskell 프로그램은 집단이 12에서 24로 2배 증가했을 때 실행시간이 1.41배 증가했지만, C++의 경우 3.44배 증가하고 있다. 이는 nlogn에서 n^2 사이의 값으로 설계상에서의 시스템 분석과 유사하다. 표 2는 진화횟수를 다르게 했을 때 Haskell로 작성한 프로그램과 C++로 작성한 프로그램의 실행시간을 나타낸 것이다.

(표 2) 진화 횟수에 따른 실행 시간 비교표

진화 횟수	Haskell	C++
1000	1.9	72.2
2000	3.1	142.3
3000	4.6	214.6
4000	6.0	280.4
5000	7.4	353.1

표 1과 마찬가지로 Haskell로 작성한 프로그램의 실행시간이 전체적으로 적게 걸린다. 앞선 실험과 마찬가지로 Haskell 스타일로 작성한 프로그램이기 때문으로 생각한다. 마찬가지로 변화율을 기준으로 비교해 보았다. 그림 4는 진화횟수를 다르게 했을 때 실행시간 변화를 나타낸다. 표시된 값은 마찬가지로 Haskell, C++, O(n)인 프로그램, O(nlogn)인 프로그램에 대해서 각각 n이 1,000일 때의 실행시간 값을 1로 잡고 실행시간의 비를 구한 것이다.



(그림 4) 진화 횟수와 실행시간 간의 그래프

진화 횟수에 따른 영향은 C++의 경우 거의 O(n)에 근접함을 볼 수 있다. 진화 횟수에 따른 그래프에서는 Haskell도 그래프의 형태가 큰 차이를 보이지는 않는다. 하지만 C++에 비해서는 그 증가폭이 작음을 알 수 있다.

5. 결론 및 향후 연구과제

동일한 알고리즘으로 구현한 Haskell 프로그램과 C++ 프로그램의 실행시간 비교 실험을 통해서 진화 전략과 같이 동일한 형태의 계산이 반복되는 알고리즘은 Haskell을 사용했을 때 시간복잡도의 증가폭이 C++에 비해서 적음을 알 수 있었다. 각각의 프로그램이 각각의 언어에서 진화전략 알고리즘을 사용하기 위한 최적화된 형태라고 볼 수는 없으므로 실행시간 자체가 아니라, 실행시간의 비로 비교하였다. 집단의 크기를 4배로 증가시켰을 때 C++가 Haskell보다 약 5배의 증가폭을 보였으며, 진화횟수를 4배로 증가시켰을 때는 C++가 Haskell 약 30% 정도 높은 증가폭을 보였다.

이러한 결론을 통해서 단순히 진화 전략과 같은 진화형 알고리즘이 아니더라도 n이 몹시 커질 수 있고 동일한 형태의 연산이 반복적으로 이루어지는 다양한 알고리즘에서 Haskell을 사용한 프로그램 개발이 효과적일 수 있다는 결론을 얻을 수 있다. 다만 본 문서에서의 실험은 어디까지나 List를 사용한 연산에 국한될 수 있으므로 유사한 알고리즘에 대해서 Array를 통해서 구현하거나 Haskell의 경우 병렬 Haskell, C++의 경우 쓰레드를 이용한 프로그램에서의 비교가 더해진다면 더 도움이 될만한 결과를 얻을 수 있을 것으로 기대한다. 따라서 향후 연구과제로 병렬 Haskell을 이용한 진화형 알고리즘 프로그램과 동일한 알고리즘에 대해서 쓰레드를 이용한 C++ 프로그램을 통해 실험을 해보고자 한다.

ACKNOWLEDGMENT

본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-15-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr).

참고문헌

[1] H. Graham, *Programming in Haskell,* Cambridge University Press, 2007.
[2] I. Rechenberg, "Evolution Strategy: Optimization of Technical systems by means of biological evolution," Fromman-Holzboog, Stuttgart 104, 1973.
[3] J. A. Bondy and U. Murty, "Graph theory with applications," London: Macmillan, 1976.
[4] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," Evolutionary Computation, IEEE Transactions on, Vol. 1, No. 1, pp. 53-66, 1997.
[5] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," Operations research, Vol. 21, No. 1, pp. 498-516, 1973.
[6] M. Held and R. M. Karp. "The traveling -salesman problem and minimum spanning trees," Operations Research, Vol. 18, No. 6, pp. 1138-1162, 1970.
[7] J. D. Knowles and D. W. Corne, "Approximating the nondominated front using the Pareto archived evolution strategy," Evolutionary computation, Vol. 8, No. 2, pp. 149-172, 2000.
[8] N. Hansen and S. Kern, "Evaluating the CMA evolution strategy on multimodal test functions," In: Parallel problem solving from nature-PPSN VIII, Springer Berlin Heidelberg, pp. 282-291, 2004.
[9] L. V. Snyder and M. S. Daskin, "A ran dom-key genetic algorithm for the generalized traveling salesman problem," European Journal of Operational Research, Vol. 174, No. 1, pp. 38-53, 2006.