INHA UNIVERSITY

알고리즘 설계 프로젝트

n-Queens 문제 (Backtracking, Genetic Algorithm)

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.
2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.
3. 나는 보고서의 내용을 조작하지 않겠습니다.
4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.
5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2020년 6월 28일

학부 정보통신공학과

학년 3

성명 심규환

학번 12181793

개요

(1) 전체 코드

(2) Backtracking

i. 구현상 특징

(3) Genetic Algorithm

i. 구현상 특징

a) Encoding

b) Selection

c) Crossover

d) Mutation

(4) 성능 비교 및 분석

i. N 의 값을 달리하여 비교

ii. Generation 수를 달리하여 비교

iii. 분석

(1) 전체 코드

```cpp
#include <chrono>
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <vector>
#include <ctime>
#define MAX 15 // 최대 N
#define POPULATION 100 // 염색체 집단의 수
#define CrossoverRate 0.7 // Crossover 확률
#define MutationRate 0.01 // Mutation 확률
#define GenerationNum 5000 // Generation 수
using namespace std;

int col[MAX];
int fit[POPULATION] = { 0, };
int N, total_Backtracking, total_GA = 0;

// 오차 측정 함수
float error(float exval, float real) {
    float error;
    error = ((real - exval) / real) * 100;
    return error;
}

// 퀸의 배치가 유효한지 확인하는 함수
bool promising(int y){

    for (int i = 0; i < y; i++)
        // col[y] == col[i]: x좌표가 같은 경우 => 같은 열인 경우
        // abs(col[y] - col[i]) == y - i : x좌표의 차이와 y좌표의 차이가 같은 경우 => 같은 대각선 인경우
        if (col[y] == col[i] || abs(col[y] - col[i]) == y - i)
            return false; // Backtracking 이용. false이므로 되돌아 온다.
    return true;
}
// y좌표가 y일때 퀸 배치하는 함수
// y == N이라면 배치가 완료되었으므로 방법의 수 +1
void nqueen(int y)
{
    if (y == N)
        total_Backtracking++;
    else
    {
        for (int i = 0; i < N; i++)
        {
            col[y] = i; // 체스판 위치를 좌표로 나타내면 퀸이 놓인 자리는 (col[y], y) 이다.

            if (promising(y))  // 유효하다면 다음행의 퀸 배치, 유효하지않다면 다른 위치로 퀸 배치 변경
                nqueen(y + 1);
        }
    }
}
```

```cpp
// GA 알고리즘
class NQueensGA
{
public:
    vector<vector<int>> vecPopulation; // 염색체 집단
    vector<vector<int>> answer; // 정답 염색체 집단
    bool isAnswer(vector<int>& vec); // 어떤 염색체가 정답 염색체 집단에 속하는지 확인하는 함수
    void InitPopulation(); // 초기 염색체 집단 생성 함수
    vector<int> PermutationEncoding(int&); // 순열 기반 Encoding 함수
    void CalFitness(); // fitness 계산 함수
    void elitism(); // elitism 구현 함수
    int RouletteWheelSelection(); // 룰렛 휠 선택 함수
    void CrossoverPMX(const vector<int>&, const vector<int>&); // PMX 교배 함수
    void Mutation(vector<int>&); // 돌연변이 함수
    void Epoch();
    int index1, index2; // index를 복사하는 parameter
    int elite1, elite2; // elite의 index를 복사하는 parameter
    int max1, max2 = 0;
    int MAXFIT; // fitness 최댓값
    vector<int> mom, dad, baby1, baby2; // 부모 염색체와 자식 염색체를 복사하는 parameter
};
// x~y 사이의 랜덤한 integer return
int RandInt(int x, int y)
{
    return rand() % (y - x + 1) + x;
}

// 0~1 사이의 랜덤한 float return
float RandFloat() {

    return (rand()) / (RAND_MAX + 1.00);
}

// number가  벡터 vec에 포함되어 있는지 확인
// 포함 되어 있으면 true
// 포함 안되어 있으면 false
bool TestNumber(const vector<int>& vec, const int& number)
{
    for (int i = 0; i < vec.size(); ++i)
    {
        if (vec[i] == number)
        {
            return true;
        }
    }

    return false;
}
```

```cpp
// 어떤 염색체가 정답 염색체 집단에 속해있는지 확인 하는 함수
bool NQueensGA::isAnswer(vector<int>& vec)
{
    for (int i = 0;i < answer.size();i++) {
        if (vec == answer[i]) {
            return true;
        }
    }
    return false;
}

// 초기 염색체 집단을 만드는 함수
void NQueensGA::InitPopulation()
{
    for (int k = 0; k < POPULATION; k++) {
        vecPopulation.push_back(PermutationEncoding(N));
        // 순열을 기반으로 Encoding한 후 Encoding한 염색체를 초기집단에 push
    }
}
// Encoding
// 순열 기반 염색체 Encoding
vector<int> NQueensGA::PermutationEncoding(int& limit)
{
    vector<int> vecPerm;

    for (int i = 0; i < limit; i++)
    {
        int NextPossibleNumber = RandInt(0, limit - 1); // 무작위 수 가져오기

        while (TestNumber(vecPerm, NextPossibleNumber)) // vector에 NextPossibleNumber포함 되어 있으면
        {
            NextPossibleNumber = RandInt(0, limit - 1); // 다시 무작위 수 가져오기
        }
        // vector에 NextPossibleNumber포함 되어 있지 않으면
        vecPerm.push_back(NextPossibleNumber); // 가져온 무작위 수 push_back
    }
    return vecPerm;
}
// 각 염색체의 fitness를 계산하는 함수
// fitness는 서로 공격하지 않는 퀸의 쌍들의 수로 한다.
void NQueensGA::CalFitness()
{
    for (int k = 0; k < POPULATION; k++) {
        for (int i = 0;i < N - 1;i++) {
            for (int j = i + 1; j < N;j++) {
                // 같은 대각선에 있지 않으면 => 서로 공격하지 않음
                if (abs(vecPopulation[k][i] - vecPopulation[k][j]) != j - i)
                    fit[k]++; // 적합도 증가
            }
        }
    }
}
```

```cpp
// Selection
// 룰렛 휠 선택 함수
int NQueensGA::RouletteWheelSelection()
{
    int a = 0;
    int b = 1;
    int FitnessSum = 0;

    // 적합도 총합 구하기
    for (int i = 0; i < POPULATION; i++) {
        if (i == elite1 || i == elite2) continue; // 엘리트 유전자 제외
        FitnessSum = FitnessSum + fit[i];
    }

    // 적합도 총합 만큼 룰렛 배열 할당
    int* roulette = new int[FitnessSum];

    // 적합도 만큼 룰렛 배열에 염색체의 인덱스를 저장한다
    for (int i = 0; i < POPULATION; i++) {
        if (i == elite1 || i == elite2) continue; // 엘리트 유전자 제외
        a = fit[i];
        for (int j = b;j < a + b;j++) {
            roulette[j] = i;
        }
        b = b + a;
    }

    // 임의로 point를 지정함
    int point = (rand() % FitnessSum) + 1;

    // point가 가리키는 인덱스를 선택
    return roulette[point];
}
// Crossover
// 교배후에도 염색체가 순열에 기반하여야 하므로 PMX Crossover operator를 이용
// 두 염색체 교배후 baby1, baby2 염색체 생성
void NQueensGA::CrossoverPMX(const vector<int>& mom, const vector<int>& dad)
{
    baby1 = mom;
    baby2 = dad;

    if ((RandFloat() > CrossoverRate))
    {
        return;
    }

    //염색체 Crossover 구간 설정
    int beg = RandInt(0, N - 2); // 시작 위치 설정

    int end = beg;

    // 마지막 위치 설정
    while (end <= beg)
    {
        end = RandInt(0, N - 1);
    }

    vector<int>::iterator posGene1, posGene2;
```

```
    for (int pos = beg; pos < end + 1; ++pos)
    {
        // 교체할 두 유전자 gene1, gene2
        int gene1 = mom[pos];
        int gene2 = dad[pos];

        if (gene1 != gene2)
        {
            // 염색체 baby1에서 교체할 유전자의 위치를 찾고 유전자 교체
            posGene1 = find(baby1.begin(), baby1.end(), gene1);
            posGene2 = find(baby1.begin(), baby1.end(), gene2);
            swap(*posGene1, *posGene2);

            // 염색체 baby1에서 교체할 유전자의 위치를 찾고 유전자 교체
            posGene1 = find(baby2.begin(), baby2.end(), gene1);
            posGene2 = find(baby2.begin(), baby2.end(), gene2);
            swap(*posGene1, *posGene2);
        }

    }
}
// Mutation
// Insertion Mutation
// 하나의 유전자의 위치를 바꿈
void NQueensGA::Mutation(vector<int>& vec)
{
    if ((RandFloat() > MutationRate))
    {
        return;
    }

    vector<int>::iterator cur;  // vector 요소의 위치를 가리킬 iterator cur 선언

    cur = vec.begin() + RandInt(0, vec.size() - 1); // 옮길 유전자의 위치 랜덤으로 설정
    int x = *cur; // 옮길 유전자 x
    vec.erase(cur); // 염색체로부터 유전자 삭제
    cur = vec.begin() + RandInt(0, vec.size() - 1); // 유전자 삽입 위치 랜덤으로 설정
    vec.insert(cur, x); // 유전자 삽입

}
```

```cpp
void NQueensGA::Epoch()
{
    // 적합도의 최댓값: MAXFIT
    // fit[i] == MAXFIT면 i번 염색체는 정답 염색체가 된다.
    MAXFIT = N * (N - 1) / 2;

    // 초기 염색체 집단 생성
    InitPopulation();

    //Generation 시작
    for (int i = 0; i < GenerationNum; i++) {
        // 적합도 계산
        CalFitness();

        //적합도 == MAXFIT 이고 answer에 염색체가 없다면 push하고 방법의 수 증가
        for (int j = 0;j < POPULATION;j++) {
            if (fit[j] == MAXFIT && !isAnswer(vecPopulation[j])) {
                answer.push_back(vecPopulation[j]); // answer에 정답 염색체 push
                total_GA++; // 방법의 수 증가
            }
        }

        // Seletion 두 염색체 선택
        index1 = RouletteWheelSelection();
        mom = vecPopulation[index1];
        for (;;) {
            index2 = RouletteWheelSelection();
            dad = vecPopulation[index2];
            if (mom != dad) break;
        }

        // Crossover 선택된 두 염색체 교배
        CrossoverPMX(mom, dad);


        // Mutation 교배 후 생성된 염색체 돌연변이 과정 진행
        Mutation(baby1);
        Mutation(baby2);
        vecPopulation[index1] = baby1;
        vecPopulation[index2] = baby2;

        for (int j = 0;j < POPULATION;j++) {
            fit[j] = 0;
        }
        //Generation 끝
    }
}
int main() {
    cout << "12181793 심규환" << endl;
    cout << "------------------------------" << endl;
    cout << "N Queen Problem" << endl;
    cout << "------------------------------" << endl;
    cout << "N을 입력하세요" << endl;
    cin >> N;
    srand(time(NULL));

    NQueensGA a;

    cout << "------------------------------" << endl;
    // Backtracking 수행 시간 μs단위로 측정
    chrono::system_clock::time_point StartTime = chrono::system_clock::now();
    nqueen(0);
    chrono::system_clock::time_point EndTime = chrono::system_clock::now();
    chrono::microseconds micro = chrono::duration_cast<chrono::microseconds>(EndTime - StartTime);
```

```
    cout << "Backtracking\n" << endl;
    cout << "수행 시간 : " << micro.count() << " μs" << endl; //수행시간 μs단위로 출력
    cout << "방법의 수 : " << total_Backtracking << endl; // GA 사용시 방법의 수 출력

    cout << "------------------------------" << endl;
    // Genetic Algorithm 수행 시간 s단위로 측정
    chrono::system_clock::time_point StartTime1 = chrono::system_clock::now();
    a.Epoch();
    chrono::system_clock::time_point EndTime1 = chrono::system_clock::now();
    chrono::duration<double> DefaultSec = (EndTime1 - StartTime1);

    cout << "Genetic Algorithm\n" << endl;
    cout << "수행 시간 : " << DefaultSec.count() << " s" << endl; //수행시간 s단위로 출력
    cout << "방법의 수 : " << total_GA << endl; // GA 사용시 방법의 수 출력
    cout << "Genearation 수 : " << GenerationNum << endl; // Geration 수 출력
    cout << "오차 : " << error(total_GA, total_Backtracking) << "%" << endl; // 오차 계산후 출력
    cout << "------------------------------" << endl;

    return 0;
}
```

(2) Backtracking

i. 구현상특징

```
// 퀸의 배치가 유효한지 확인하는 함수
bool promising(int y){

    for (int i = 0; i < y; i++)
        // col[y] == col[i]: x좌표가 같은 경우 => 같은 열인 경우
        // abs(col[y] - col[i]) == y - i : x좌표의 차이와 y좌표의 차이가 같은 경우 => 같은 대각선 인경우
        if (col[y] == col[i] || abs(col[y] - col[i]) == y - i)
            return false; // Backtracking 이용. false이므로 되돌아 온다.
    return true;
}
```

promising 함수는 (col[y], y) 위치의 퀸이 다른 위치의 퀸을 공격하는지 즉 같은 대각선이거나 같은 행, 열인지 확인하고 만약 공격한다면(유효하지 않다면) Backtracking을 한다.

```
// y좌표가 y일때 퀸 배치하는 함수
// y == N이라면 배치가 완료되었으므로 방법의 수 +1
void nqueen(int y)
{
    if (y == N)
        total_Backtracking++;
    else
    {
        for (int i = 0; i < N; i++)
        {
            col[y] = i; // 체스판 위치를 좌표로 나타내면 퀸이 놓인 자리는 (col[y], y) 이다.

            if (promising(y))  // 유효하다면 다음행의 퀸 배치, 유효하지않다면 다른 위치로 퀸 배치 변경
                nqueen(y + 1);
        }
    }
}
```

nqueen 함수는 0행부터 하나씩 퀸을 놓고 유효한지 promising 함수로 확인한 다음 유효하다면 퀸을 배치하고 그렇지 않으면 재귀를 통해 퀸의 배치를 변경하도록 한다.

(2) Genetic Algorithm

i. 구현상특징

a) Encoding

```cpp
// Encoding
// 순열 기반 염색체 Encoding
vector<int> NQueensGA::PermutationEncoding(int& limit)
{
    vector<int> vecPerm;

    for (int i = 0; i < limit; i++)
    {
        int NextPossibleNumber = RandInt(0, limit - 1); // 무작위 수 가져오기

        while (TestNumber(vecPerm, NextPossibleNumber)) // vector에 NextPossibleNumber포함 되어 있으면
        {
            NextPossibleNumber = RandInt(0, limit - 1); // 다시 무작위 수 가져오기
        }
        // vector에 NextPossibleNumber포함 되어 있지 않으면
        vecPerm.push_back(NextPossibleNumber); // 가져온 무작위 수 push_back
    }
    return vecPerm;
}
```

			O
O			
		O	
	O		

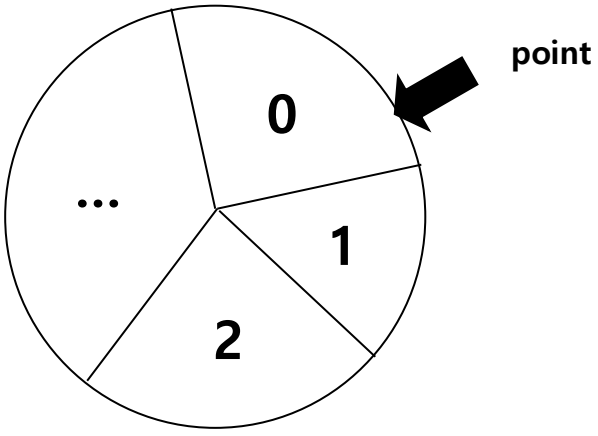
vec염색체 (1, 2, 0, 3)

다음과 같은 퀸의 배치가 있다고 할 때 우리는 염색체 vector<int> vec를 (1, 2, 0, 3)로 설정한다. 즉 vector의 index가 행이 되고 열이 요소가 된다. 이 때 각 요소는 겹치지 않고 0~3 까지의 정수이므로 순열을 기반으로 염색체를 Encoding 한다. 순열 기반으로 염색체를 Encoding하면 다음 과정들을 진행하면서도 염색체가 순열을 기반으로 유지 되어야한다.

b) Selection

```
// Selection
// 룰렛 휠 선택 함수
int NQueensGA::RouletteWheelSelection()
{
    int a = 0;
    int b = 1;
    int FitnessSum = 0;

    // 적합도 총합 구하기
    for (int i = 0; i < POPULATION; i++) {
        if (i == elite1 || i == elite2) continue; // 엘리트 유전자 제외
        FitnessSum = FitnessSum + fit[i];
    }

    // 적합도 총합 만큼 룰렛 배열 할당
    int* roulette = new int[FitnessSum];

    // 적합도 만큼 룰렛 배열에 염색체의 인덱스를 저장한다
    for (int i = 0; i < POPULATION; i++) {
        if (i == elite1 || i == elite2) continue; // 엘리트 유전자 제외
        a = fit[i];
        for (int j = b;j < a + b;j++) {
            roulette[j] = i;
        }
        b = b + a;
    }

    // 임의로 point를 지정함
    int point = (rand() % FitnessSum) + 1;

    // point가 가리키는 인덱스를 선택
    return roulette[point];
}
```

Seletion은 Elitism을 이용한 Roulette Wheel Seletion을 선택했다. Elite 염색체를 제외한 fitness의 총합을 계산하고 총합 사이즈만큼 배열을 동적으로 할당한다. 그리고 fitness 값만큼 동적으로 할당한 배열에 index값을 저장한다. 그리고 임의로 point를 정하고 point가 가리키는 index를 선택하면 index가 가리키는 염색체를 선택하게 된다. 아래와 같은 경우는 0번 염색체를 선택한다.



c) Crossover

```
// Crossover
// 교배후에도 염색체가 순열에 기반하여야 하므로 PMX Crossover operator를 이용
// 두 염색체 교배후 baby1, baby2 염색체 생성
void NQueensGA::CrossoverPMX(const vector<int>& mom, const vector<int>& dad)
{
    baby1 = mom;
    baby2 = dad;

    if ((RandFloat() > CrossoverRate))
    {
        return;
    }

    //염색체 Crossover 구간 설정
    int beg = RandInt(0, N - 2); // 시작 위치 설정

    int end = beg;

    // 마지막 위치 설정
    while (end <= beg)
    {
        end = RandInt(0, N - 1);
    }

    vector<int>::iterator posGene1, posGene2;
   for (int pos = beg; pos < end + 1; ++pos)
   {
       // 교체할 두 유전자 gene1, gene2
       int gene1 = mom[pos];
       int gene2 = dad[pos];

       if (gene1 != gene2)
       {
           // 염색체 baby1에서 교체할 유전자의 위치를 찾고 유전자 교체
           posGene1 = find(baby1.begin(), baby1.end(), gene1);
           posGene2 = find(baby1.begin(), baby1.end(), gene2);
           swap(*posGene1, *posGene2);

           // 염색체 baby1에서 교체할 유전자의 위치를 찾고 유전자 교체
           posGene1 = find(baby2.begin(), baby2.end(), gene1);
           posGene2 = find(baby2.begin(), baby2.end(), gene2);
           swap(*posGene1, *posGene2);
       }

   }
}
```

70%의 확률로 교배를 진행하도록 하였다.

순열을 기반으로 Encoding한 염색체를 일반적인 방법으로 교배를 진행하게 되면 교배 후 자손 염색체가 순열을 기반으로 하지 않은 염색체가 나올 수도 있다. 따라서 PMX crossover operator를 이용하여 Crossover를 진행한다. PMX는 임의로 지정한 두 지점에서 치환 규칙을 정하고 한 염색체에서 치환 규칙에 따라 염색체의 유전자 교체를 진행한다.

↓ ↓

염색체1 (2, 5, 0, 3, 6, 1, 4, 7)

염색체2 (3, 4, 0, 7, 2, 5, 1, 6)

이렇게 임의의 두지점이 선택되면 1과 5를 교체하고, 7과 6을 교체하는 규칙이 생기고

자식 염색체 3과 4가 아래와 같이 생기게 된다.

염색체3 (2, 1, 0, 3, 7, 5, 4, 6)

염색체4 (3, 4, 0, 6, 2, 1, 5, 7)

d) Mutation

```
// Mutation
// Insertion Mutation
// 하나의 유전자의 위치를 바꿈
void NQueensGA::Mutation(vector<int>& vec)
{
    if ((RandFloat() > MutationRate))
    {
        return;
    }

    vector<int>::iterator cur;  // vector 요소의 위치를 가리킬 iterator cur 선언

    cur = vec.begin() + RandInt(0, vec.size() - 1); // 옮길 유전자의 위치 랜덤으로 설정
    int x = *cur; // 옮길 유전자 x
    vec.erase(cur); // 염색체로부터 유전자 삭제
    cur = vec.begin() + RandInt(0, vec.size() - 1); // 유전자 삽입 위치 랜덤으로 설정
    vec.insert(cur, x); // 유전자 삽입


}
```

교배를 진행한 후 1%의 확률로 돌연변이를 생성하게 된다.

Insertion Mutation을 이용하여 돌연변이를 생성하게 만들었다.

Insertion Mutation은 임의의 한 지점을 선정한 후에 그 위치의 유전자를 다른 위치로 이동하도록 하여 돌연변이를 생성한다.

↓

(2, 5, 0, 3, 6, 1, 4, 7) => 돌연변이 염색체 (2, 5, 1, 0, 3, 6, 4, 7)

(4) 성능 분석 및 비교

i. N의 값을 달리하여 비교

Generation 수는 1000으로 고정 후 N만 변화시켜가며 비교

N=3

```
12181793 심규환
-----------------------------
N Queen Problem
-----------------------------
N을 입력하세요
3
-----------------------------
Backtracking

수행 시간 : 2 μs
방법의 수 : 0
-----------------------------
Genetic Algorithm

수행 시간 : 0.0540488 s
방법의 수 : 0
Genearation 수 : 1000
오차 : 0%
-----------------------------
```

N=4

```
12181793 심규환
-----------------------------
N Queen Problem
-----------------------------
N을 입력하세요
4
-----------------------------
Backtracking

수행 시간 : 5 μs
방법의 수 : 2
-----------------------------
Genetic Algorithm

수행 시간 : 0.102014 s
방법의 수 : 2
Genearation 수 : 1000
오차 : 0%
-----------------------------
```

N=5

```
12181793 심규환
-----------------------------
N Queen Problem
-----------------------------
N을 입력하세요
5
-----------------------------
Backtracking

수행 시간 : 16 μs
방법의 수 : 10
-----------------------------
Genetic Algorithm

수행 시간 : 0.134588 s
방법의 수 : 10
Genearation 수 : 1000
오차 : 0%
-----------------------------
```

N=6

```
12181793 심규환
-----------------------------
N Queen Problem
-----------------------------
N을 입력하세요
6
-----------------------------
Backtracking

수행 시간 : 120 μs
방법의 수 : 4
-----------------------------
Genetic Algorithm

수행 시간 : 0.202983 s
방법의 수 : 4
Genearation 수 : 1000
오차 : 0%
-----------------------------
```

N=7

```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 289 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 0.432508 s
방법의 수 : 11
Genearation 수 : 1000
오차 : 72.5%
------------------------------
```

ii. Generation 수를 달리하여 비교

N은 7로 고정 후 Generation 수만 변화시켜가며 비교

Generation 수 = 1000

```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 286 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 0.326648 s
방법의 수 : 8
Genearation 수 : 1000
오차 : 80%
------------------------------
```

Generation 수 = 5000

```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 418 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 2.78727 s
방법의 수 : 29
Genearation 수 : 5000
오차 : 27.5%
------------------------------
```

Generation 수 = 10000

```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 241 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 4.90193 s
방법의 수 : 36
Genearation 수 : 10000
오차 : 10%
------------------------------
```

Generation 수 = 20000

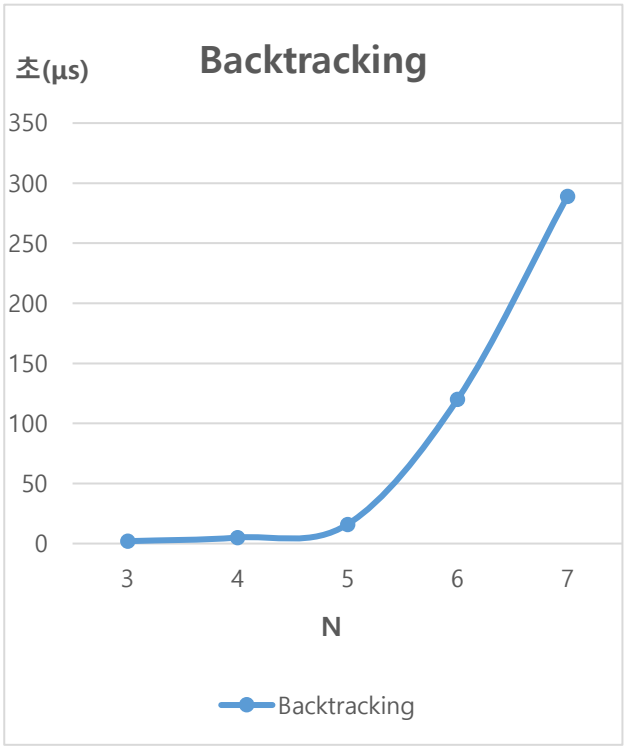
```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 206 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 7.89798 s
방법의 수 : 39
Genearation 수 : 20000
오차 : 2.5%
------------------------------
```

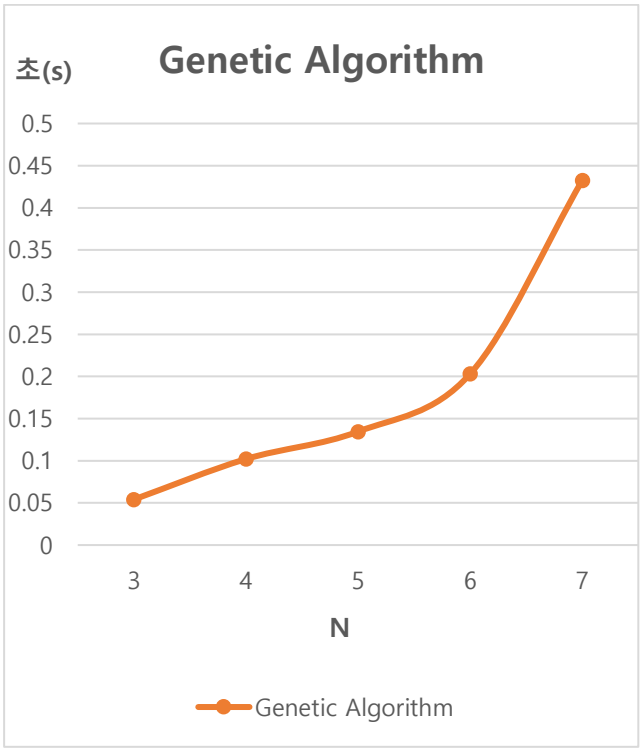
Generation 수 = 50000

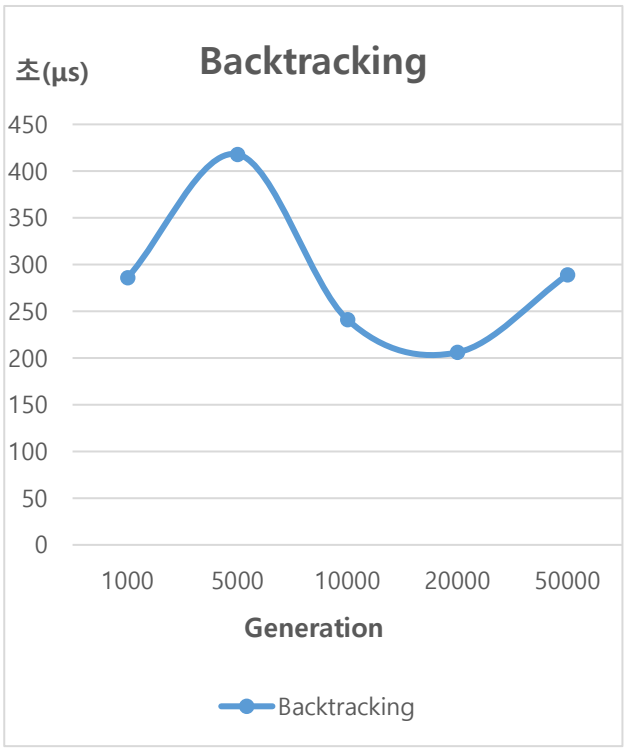
```
12181793 심규환
------------------------------
N Queen Problem
------------------------------
N을 입력하세요
7
------------------------------
Backtracking

수행 시간 : 417 μs
방법의 수 : 40
------------------------------
Genetic Algorithm

수행 시간 : 24.8256 s
방법의 수 : 40
Genearation 수 : 50000
오차 : 0%
------------------------------
```

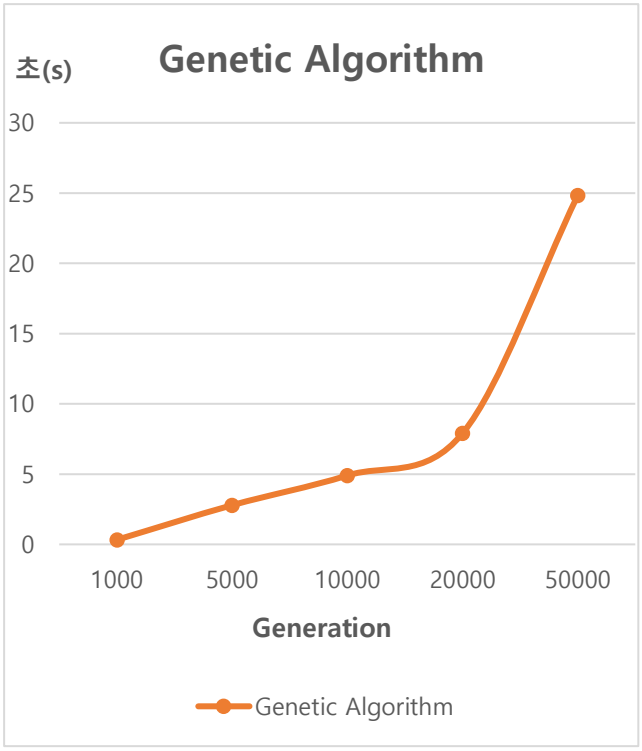
iii. 분석

수행시간 그래프

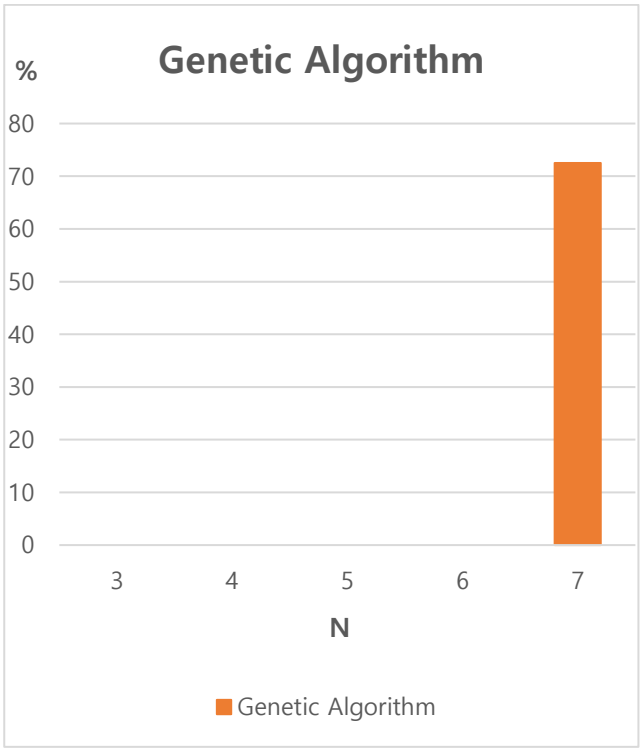


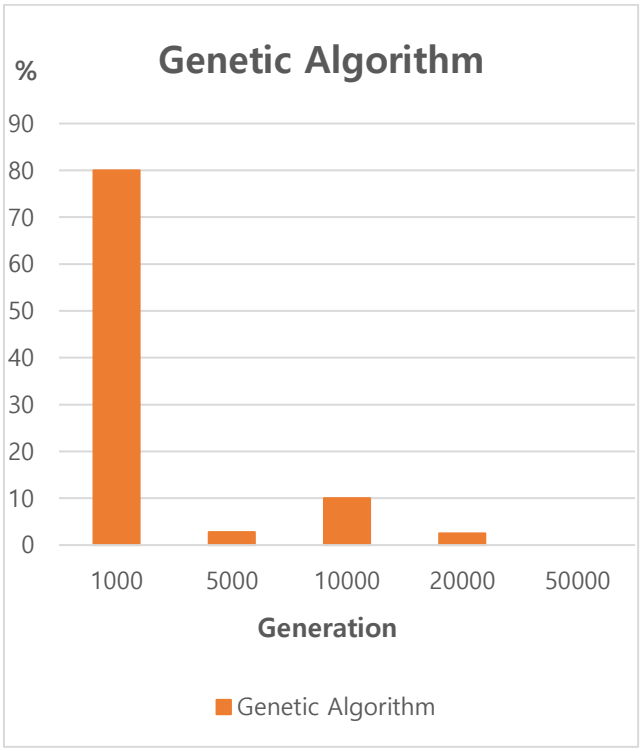






Genetic Algorithm 오차 그래프





Backtracking의 수행시간 그래프를 살펴보면 N이 커질수록 수행시간이 증가하는 것을 볼 수 있다. 그리고 Generation의 수와는 관계없이 Backtracking이 작동하므로 N=7일 때 약 200 μs ~ 450 μs 내에서 동작함을 추측할 수 있다.

Genetic Algorithm의 수행시간 그래프를 살펴보면 N이 커질수록 수행시간이 증가하며 Generatrion이 커질수록 마찬가지로 수행시간이 증가하는 것을 확인할 수 있다. 오차 그래프를 보면 Generatrion이 커질수록 Genetic Algorithm의 오차가 작아지는 것을 확인할 수 있는데 이를 통해 Generatrion가 많이 지날수록 수행시간은 증가하지만 더욱 정확한 결과값을 얻어 낼 수 있음을 알 수 있다.

Backtracking과 Genetic Algorithm을 비교하였을 때는 Backtracking이 수행시간이 압도적으로 빠를 뿐만 아니라 오차도 없으므로 Backtracking이 압도적으로 성능이 좋은 것을 확인할 수 있다.