알고리즘\_과제\_10

(BackTracking, Monte Carlo)



학부: 컴퓨터학부

학번: 20162518

출석번호 : 156번

이름 : 최승서

11. n queen algorithm by monte carlo

< 소스코드>

//

// main.c

// algo\_10

//

// Created by James Choi on 2019/11/18.

// Copyright © 2019 James Choi. All rights reserved.

//

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#define MAX 8

#define TRUE 1

#define FALSE 0

int \*col;

int promising(int i) {

int k;

int result;

k = 1;

result = TRUE;

while (k < i&&result){

if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)

result = FALSE;

k++;

}

return result;

}

int estimate\_n\_queens (int n) {

int i, j, k, m, v;

int mprod, numnodes,index;

int prom\_children[MAX+1]= {0,};

i=0;

numnodes=1;

m=1;

mprod=1;

index = 0;

while (m != 0 && i!=n) {

mprod = mprod\*m;

numnodes = numnodes + mprod\*n;

i++;

m = 0;

for(k=1; k<=MAX; k++){

prom\_children[k]=0;

}

for (j=1; j<=n; j++) {

col[i] = j;

if (promising(i)) {

prom\_children[m] =j;

m++;

}

}

if(m!=0){

for(k=1; k<=MAX; k++){

if(prom\_children[k] != 0)

index++;

}

if(index == 0)

index=1;

v = (rand()%index) + 1;

j = prom\_children[v];

col[i] = j;

index = 0;

}

}

return numnodes;

}

int main() {

int i, cnt = 0;

int total = 0;

col = (int\*)malloc(sizeof(int) \* (MAX+1));

printf("monte carlo를 이용하여 n queen 문제 효율 추정하기\n");

for (i = 0; i < 20; ++i) {

int nodes = estimate\_n\_queens(MAX);

printf("%2d run : %d개\n", i+1, nodes);

if (nodes) {

total += nodes;

cnt++;

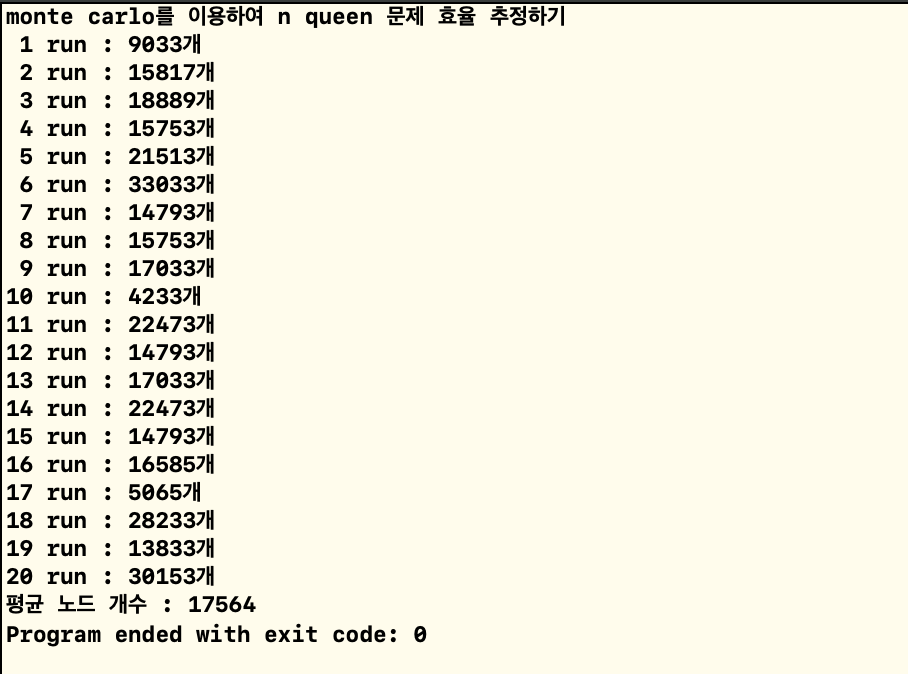
}

}

printf("평균 노드 개수 : %d\n", total/20);

free(col);

}



monte carlo는 효율을 추정하기 위한 알고리즘인데, 추정치는 해답을 모두 찾기 위해 검사할 것이라 생각되는 노드의 총 개수를 추정한 것이다.

n queen문제는 n x n 크기의 체스판에 n개의 여왕말이 있다고 했을때 이 n개의 여왕말이 서로를 죽일 수 없는 위치 구하는 알고리즘인데, backtracking 을 이용해 자식 노드가 유망한지 판별하여 유망하지 않다면 다시 부모 노드로 돌아가 다음 자식노드에 대한 탐색을 계속 진행한다.

monte carlo를 이용하여 n Queen algorithm의 효율을 추정하기 위해 20번 반복 수행한 후 평균 탐색 노드의 개수를 확인한 결과 17564회가 나오는 것을 확인하였다.

13,14,15, 16

Backtracking 알고리즘을 사용하여 Sum of Subsets 문제를 해결하는 프로그램이다. n개의 아이템에 각각의 무게가 2, 10, 13, 17, 22, 42인데 이 아이템들의 부분집합의 total weight가 52가 되는 경우를 찾는다.

노드마다 자식노드가 유망한지 확인하고, 유망하면 계속 진행되고 그렇지 않다면 다시 뒤로 돌아가 다른 자식 노드를 검사하는 방법, 즉 Backtracking 방법을 사용하여 원하는 결과를 얻는다.

<소스코드>

#include <stdio.h>

#include<stdlib.h>

#include<time.h>

#define TRUE 1

#define FALSE 0

const int w[6] = {2, 10, 13, 17, 22, 42};

int tmp\_w[6] = { 2, 10, 13, 17, 22,42 };

int include[6] = { 0,};

int tmp\_include[6]={0,};

int W =52;

int chs; //1개의 부분집합을 구할때와 전부 구할때를 구분하기 위해 선언

int promissing(int i, int weight, int total){

if((weight+total > W) && (weight == W || weight + w[i+1] <=W))

return TRUE;

else

return FALSE;

}

void print(){

printf("sum of sub of set\n");

for(int i=0; i<6; i++){

if(include[i] != 0)

printf("incldue[%d]: %d\n", i, include[i]);

}

printf("--------------\n");

chs++; // 1개의 부분집합 탐색을 완료하고 출력하면 증가

}

void sum\_of\_subsets(int i, int weight, int total){

if(promissing(i, weight, total)){ //유망 여부를 확인하여 탐색을 실행한다.

if(chs == 3) //초기 입력값 2에서 1개의 부분집합의 합을 구했을때 3으로 증가하여 함수 종료

return;

if(W==weight)//탐색을 완료했을때 출력한다.

print();

else{ //재귀적으로 실행하여 backtracking한다.

//가중치를 포함했을 경우

include[i+1]=w[i+1];

sum\_of\_subsets(i+1, weight+w[i+1], total-w[i+1]);

include[i+1]=0;

//가중치를 포함 안했을 경우

sum\_of\_subsets(i+1, weight, total-w[i+1]);

}

}

}

int estimate\_sum\_of\_subsets(int n) {

int i;

int j;

int k;

int m;

int mprod;

int numnodes; //탐색에 소비한 노드 개수를 담는다

int weight = 0;

int total = 106;

int prom\_children[6] = { 0,};//현재깊이에서 유망한자식노드값 저장

int F = FALSE; //유망한지 판별

i = 0;

numnodes = 1;

m = 1;

mprod = 1;

while (m != 0 && i != 6) {//level 이n 의안에있을때동작하도록

mprod \*= m;

numnodes += mprod \* 2;

i++;

m = 0;

for (k = 0; k < 6; k++) {//초기화시켜주는과정.

if (k == 0) {

prom\_children[k] = 0;// 0으로초기화

}

}//자식마디초기화

for (j = 0; j < 6; j++) {

if (tmp\_w[j] != 0) {

tmp\_include[i] = j;

if (promissing(i, weight, total)) {//promising확인 후 포함

m++;

prom\_children[j] = tmp\_w[j];

}

}

}

if (m != 0) { //monte carlo핵심인 난수 지정하여 경로 설정

while (F == FALSE) {

j = rand() % 5 + 1; //랜덤으로수결정

if (prom\_children[j] != 0) { //promissing할때 F값 true 반환

F = TRUE;

}

}

tmp\_include[i] = j;

tmp\_w[j] = 0;

weight += prom\_children[j];

total -= prom\_children[j];

F = FALSE;

}

}

return numnodes; // 탐색에 소비한 노드 개수 반환

}

int main() {

int numnode =0, total\_numnode = 0;

int total =0;

srand((unsigned int)time(NULL));

clock\_t start, end;

float run\_time, run\_sum=0;

printf("1. 모든 부분 집합의 합 구하기\n2. 한개의 부분 집합의 합 구하기\n3. monte carlo를 이용하여 효율 추정\n입력 : ");

scanf("%d", &chs);

switch (chs) {

case 1:

case 2:

printf("\nBackTracking을 이용하여 부분 집합의 합 구하기\n");

for(int i =0; i<6; i++)

total+=w[i];

start = clock();

sum\_of\_subsets(0, 0, total);

end = clock();

run\_time = (float)(end - start);

printf("run time : %.3f ms\n", run\_time);

break;

default:

printf("monte carlo를 이용하여 효율 추정하기\n");

for (int i = 0; i < 20; i++){ //n queen과 동일하게 더욱 정확한 효율 추정을 위해 20번 반복한다

start = clock();

numnode = estimate\_sum\_of\_subsets(6);

end= clock();

run\_time = (float)(end - start);

run\_sum +=run\_time;

printf("%d run : %d개\n", i + 1, numnode);

for (int p = 0; p < 6; p++){//추정값 획득을 위해 함수 실행시 변환되는 값을 다시 원상태로 돌려준다.

tmp\_w[p] = w[p];

tmp\_include[p] = include[p];

}

total\_numnode += numnode;

}

printf("평균 노드 개수 : %d개\n", total\_numnode / 20);

printf("평균 수행시간 : %.3fms\n", run\_sum/20);

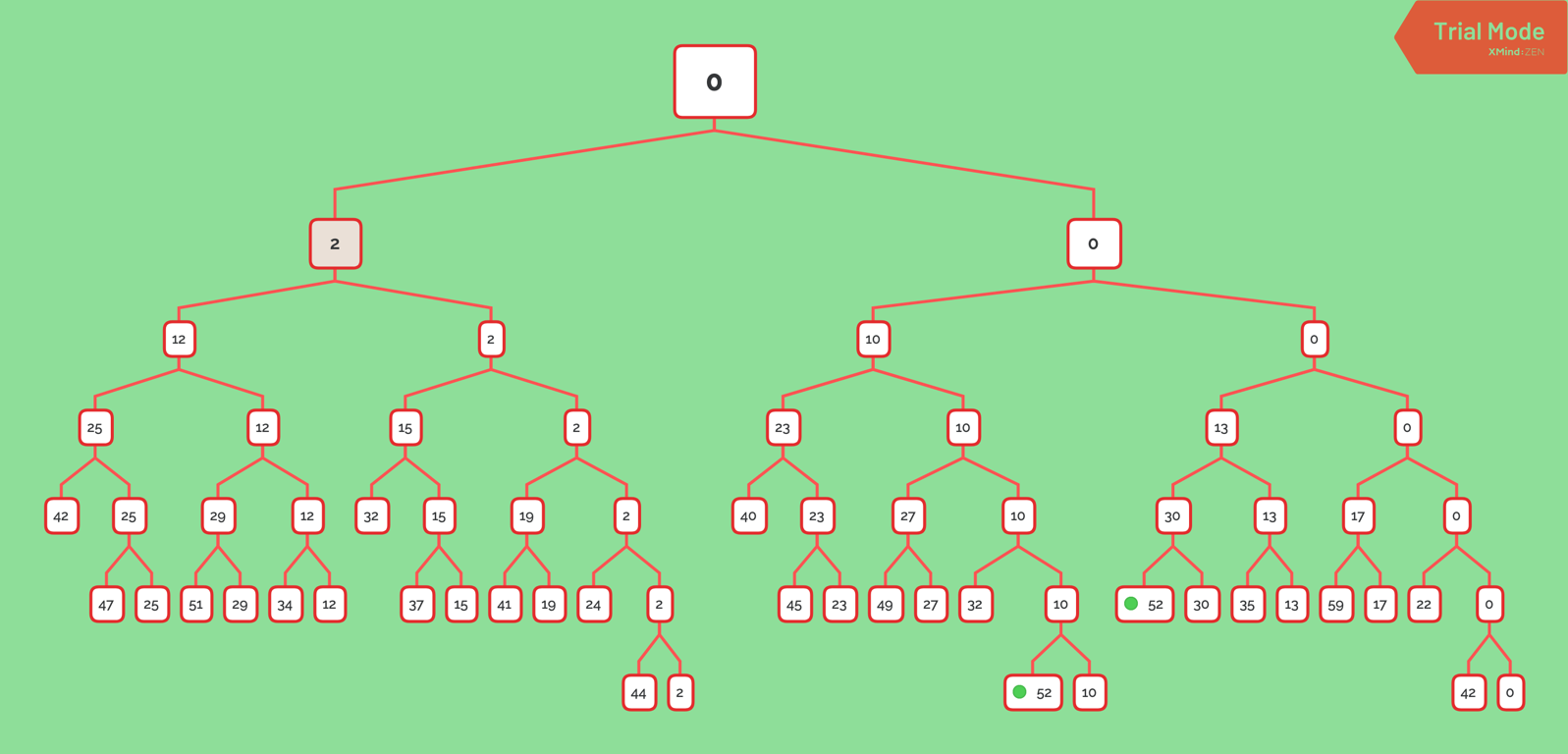
break;

}

}

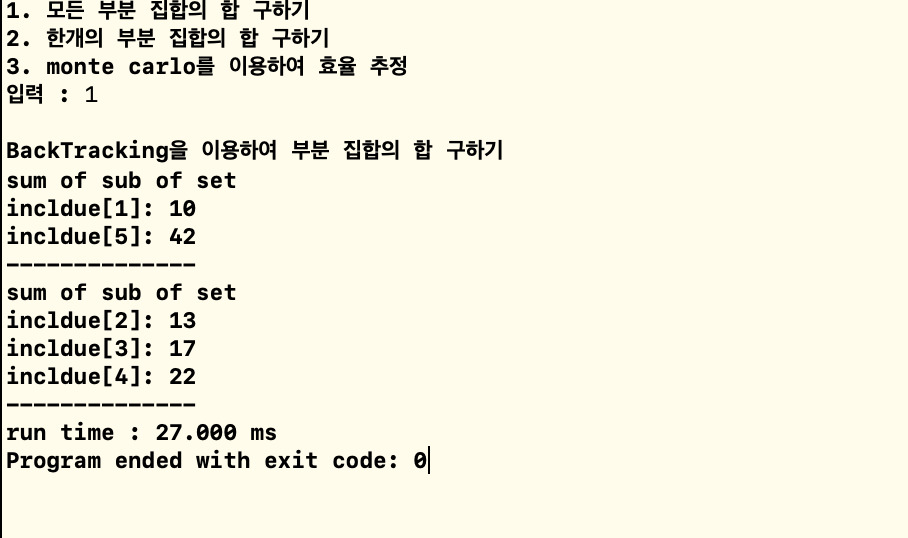
먼저 13번 풀이로

부분집합의 합 구하기를 backtracking algorithm을 사용하여 단계적으로 만들어 보았다.



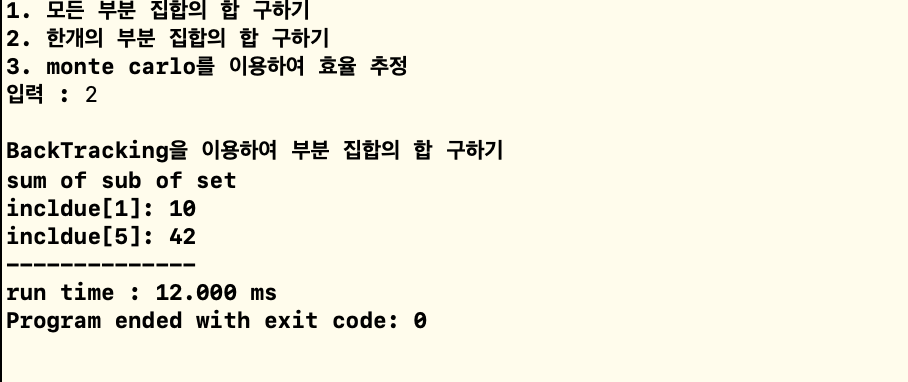
다음 마인드맵 프로그램을 이용하여 작성한 표를 확인하면 6개의 정수의 부분합이 W값을 만족하는 부분 집합의 합은 2가지가 존재하는 것을 확인하였다.

다음으로 14번 문제를 프로그래밍을 통해 구현을 해보면



{w1,w5}와 {w2,w3,w4}가 W가 52가 되는 부분집합임을 알 수있다. 이때 실행시간은 27ms가 나온다.

14번 문제가 가능한 모든 해답을 생성하는 방법이었다면 하나만 구했을 경우 출력 결과는 다음과 같다.

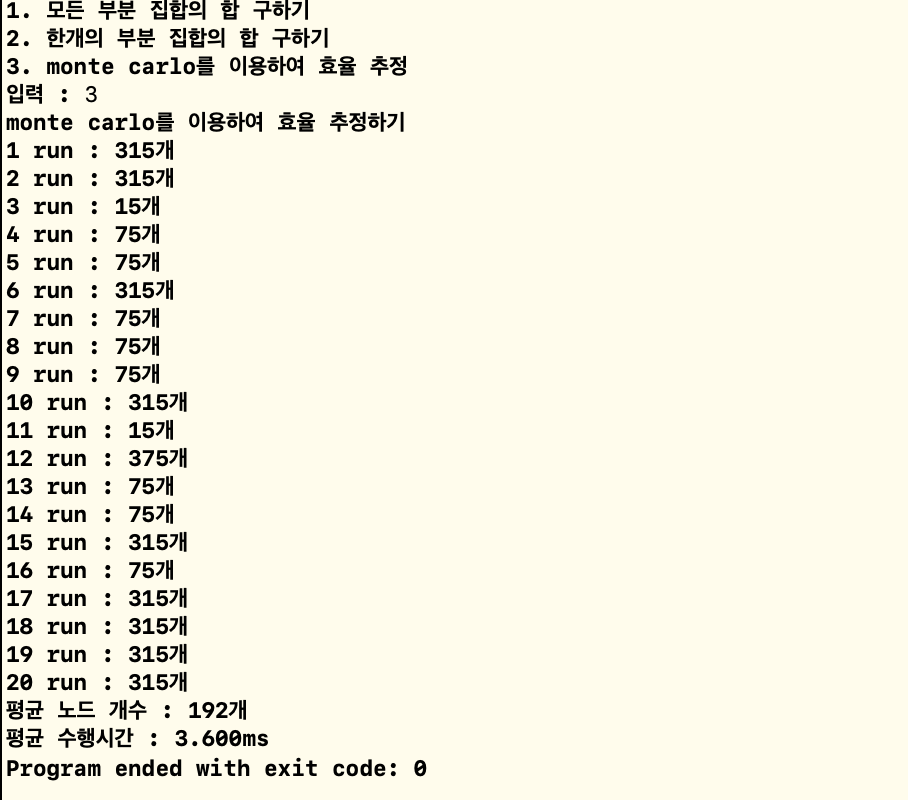


앞선 방법에서 첫번째로 구해진 부분 집합 {w1, w5}가 출력됨을 확인했다.

실행 시간이 12ms인데 사실 측정 단위가 ms 인 것을 생각하면 크게 차이가 나는 수치는 아니라고 생각한다.

마지막으로 앞선 n queen problem처럼 monte carlo를 이용해서 sum of subsets algorithm의 효율을 추정해 보았다.

결과는 다음과 같다.



Monte-carlo 기법의 근본적인 성격인 랜덤을 사용 하여 접근해본 결과 총 3가지 복잡도로 수행되는 경우를 볼 수 있었다. 원하는 해답을 얻기 위한 평균적으로 탐색에 사용되는 노드의 개수는 192개이고, 알고리즘의 평균 수행 시간은 3.6ms가 나온다. 앞선 방법에 비해 속도가 눈에 띄게 향상 됨을 알 수 있다. 이 기법을 통해 우리가 원하는 정확한 평균값은 아니지만 대략적인 평균값을 예측할 수 있었다. 또 수행 횟수를 더 늘리면 더욱 정확한 평균값을 구할 수 있을것이라 생각한다.