2020년 1학기

프로그래밍언어 과제1 : 돌 게임 (Picking Stones)

컴퓨터학부

20162481 안승훈

1. 개요

- 프로그래밍을 이용해 ‘돌 게임’ 프로그램을 작성한다.

- ‘돌 게임’ 프로그래밍을 통해 Alpha-Beta pruning 알고리즘에 대한 이해와 능력을 향상시킨다.

- 프로그램 상에서, 평가 함수를 사용하여 현재 플레이어의 최선의 돌을 선택할 수 있도록 한다.

- 선택이 완료된 경우, 선택한 돌, 선택하게 된 평가 점수, 탐색 깊이 등을 출력하도록 한다.

2. 구현 방법

- Eclipse 프로그램 상에서 C언어 또는 C++언어를 이용하여 설계하였다.

- Eclipse 프로그램과 minGW의 gcc, g++ 컴파일러를 연동하여 프로그래밍에 이용하였다.

- Alpha-Beta pruning 알고리즘과 minimax 알고리즘을 이용하여 평가 함수를 구현하였다.

- 사용자로부터 게임의 진행 정보를 받아올 수 있도록 설계하였다.

3. 게임 규칙

- 1부터 n까지 숫자가 매겨진 n개의 돌로 게임을 시작한다.

- Max 플레이어와 Min 플레이어가 번갈아 가며 돌을 하나씩 가져가며 게임이 진행된다.

- 첫 번째 플레이어는 n/2보다 작은 숫자 중에서 홀수인 돌을 선택해야 한다.

- 다음 플레이어는 마지막으로 제거된 돌의 배수 또는 인수만 가져올 수 있다.

- 플레이어의 차례에 가져갈 돌이 없는 경우에 패배하게 된다.

- 가져간 돌이 없는 경우, Max 플레이어가 게임을 먼저 시작한다.

- 프로그램에서 구현한 평가 함수를 기반으로 플레이어를 위한 최선의 돌을 선택한다.

4. 사용한 알고리즘 설명

- Mini-max 알고리즘

Mini-Max 알고리즘은 예상되는 최대의 손실(Maximum loss)을 최소화(minimize)시키기 위해 사용하는 의사결정 이론의 한 방법이다. 이 알고리즘의 처음 시작은 2명이 진행하는 게임 이론에서 시작되었고, 현재는 어떠한 불확실성이 존재하는 상황에서의 의사경정을 하는 경우에까지 적용되고 있다. 시간과 용량이 허용되는 한도 내에서 깊이 우선 탐색을 실행하면서, 상대에 대한 최솟값을 찾음으로써 플레이어 자신에게 대한 최댓값을 얻는 게임 트리의 검색 방법을 Mini-Max 알고리즘이라 한다. Mini-max 알고리즘은 내가 선택하는 모든 경우에 대해서 각각의 경우에 대해 상대방의 결정을 모두 고려하여 자신의 선택을 결정하는 과정을 거친다. 이로 인해 시간복잡도를 계산하여보면, 게임 트리의 마디에서 뻗어 나오는 가지의 수를 b 라 하고, 플레이어 들이 움직일 수 있는 경우의 수를 p 라고 할 때, O(b^p)의 시간복잡도를 가진다. 이 알고리즘을 의사 코드로 작성하면 대략 이런 모양이 된다.

**function** minimax(node, depth, maxPlayer) **is**

**if** depth = 0 **or** node is a terminal node **then**

**return** the heuristic value of node

**if** maxPlayer **then**

value := −∞

**for each** child of node **do**

value := max(value, minimax(child, depth − 1, FALSE))

**return** value

**else** *(\* minPlayer \*)*

value := +∞

**for each** child of node **do**

value := min(value, minimax(child, depth − 1, TRUE))

**return** value

mini-max 알고리즘의 단점으로는, 앞에서 언급한 바와 마찬가지로, 트리의 가지가 많아질수록, 그리고 탐색하는 트리의 깊이가 깊어질수록, 시간복잡도가 기하급수적으로 증가하는 것이라고 할 수 있다. 이러한 mini-max 알고리즘의 문제점을 해결하기 위해서 alpha-beta pruning 알고리즘이 개발 되었다.

- Alpha-Beta Pruning 알고리즘

Alpha-Beta Pruning 알고리즘은, 이름에서 알 수 있다시피, pruning, 즉 가지치기를 하는 알고리즘이다. 초기에 등장한 mini-max 알고리즘의 단점인 시간복잡도를 줄이기 위해서, 계산할 필요가 없는 노드는 과감하게 삭제하고 탐색을 하지 않음으로써, 시간복잡도를 줄이고 탐색에 소요되는 시간과 비용을 줄일 수 있도록 고안되었다. 한마디로 알고리즘을 요약하자면, ‘상대가 현재 위치 이후의 어떤 지점에서 더 유리한 결과를 얻는 경우가 존재 한다면, 현재 위치를 만드는 나의 선택은 나에게 무조건 불리한 수 이므로 더 이상 고려할 필요가 없다’ 라는 관점에서 시작한 알고리즘이라는 것이다. 즉, 플레이어에게 불리한 수가 만들어지는 지점부터는 그 이후의 트리는 탐색하지 않고 가지를 쳐버리는(pruning) 것이다. 이때, 어떤 노드에서부터 가지를 칠 것인가를 구체적으로 판단하기 위해서, 현재까지의 탐색의 결과인 나에게 가장 유리한 최댓값과 상대에게 가장 유리한 최댓값을 기억하고 있어야 한다. 트리를 탐색하는 과정에서, 현재의 검색 상황에서 상대의 현재 최대값이 나의 현재 최대값보다 크다면, 그 노드 이하로는 더 이상 고려할 필요가 없어진다.

이 Alpha-Beta Pruning 알고리즘을 의사코드로 표현하면 다음과 같다.

**function** alpha\_beta(node, depth, α, β, maxPlayer)

**if** depth = 0 **or** node is a terminal node

**return** the heuristic value of node

**if** maxPlayer

v := -∞

**for each** child of node

v := max(v, alpha\_beta(child, depth - 1, α, β, FALSE))

α := max(α, v)

**if** β ≤ α

**break** *(\* β cut-off \*)*

**return** v

**else**

v := ∞

**for each** child of node

v := min(v, alpha\_beta(child, depth - 1, α, β, TRUE))

β := min(β, v)

**if** β ≤ α

**break** *(\* α cut-off \*)*

**return** v

의사코드에서 보면, Alpha cut-off 의 경우는 자신이 상대방보다 불리하여 자신이 그 경우를 선택하지 않을 경우를 잘라내는 상황이고, Beta cut-off 의 경우에는 자신이 상대방보다 유리하여 상대방이 그 경우를 선택하지 않을 확률이 높은 경우에 잘라내는 상황이다.

Alpha-Beta Pruning 알고리즘의 장점은, 트리의 탐색 가지를 줄일 수 있다는 것이다. 이러한 경우 탐색은 더 의미 있는 부분 트리에 대해 이루어지므로, 같은 시간 안에 더 깊은 노드까지 탐색할 수 있는 시간적 여유를 가진다. 또한, 노드가 최적에 가까운 순서대로 평가되는 경우, 탐색 깊이가 Alpha-Beta Pruning 알고리즘의 기본이 되는 Mini-Max 알고리즘의 반 이하가 되게 최적화가 가능하다. Alpha-Beta Pruning의 단점은, 상호간의 플레이어가 모두 자신에게 최선이 되는 선택을 한다는 전제로 게임이 진행되는데, 만약 한쪽 플레이어가 최선의 선택이 아닌 다른 선택을 할 경우, 알고리즘이 엉키거나, 그 시점부터 알고리즘을 통한 계산을 다시 해야 하는 경우가 발생할 수 있다.

게임 트리의 마디에서 뻗어 나오는 가지의 수를 b 라 하고, 플레이어 들이 움직일 수 있는 경우의 수를 p 라고 할 때, Alpha-Beta Pruning 알고리즘의 시간 복잡도는 최악의 경우에는 mini-max 알고리즘과 동일한 O()이지만, 노드의 구성이 최선의 순서로 되어있을 경우, 최선의 시간복잡도는 O 가 된다.

5. 구현한 자료구조 및 함수 설명

- C++을 이용하여 프로그램을 작성하였다

- 시작하기에 앞서, 프로그램을 완성하지 못했음을 밝힌다.

* 저장을 위한 자료구조

pair<int, double> p;

- 돌의 번호와 그 돌의 가중치를 저장하기 위한 pair 구조체를 선언하였다.

vector<pair<int, double>> v;

- 앞에서 선언한 pair 구조체를 저장하기 위하여 vector를 사용하였다.

bool stone[1000];

- 깊이 우선 탐색 시, 방문한 돌의 번호 노드와 방문하지 않은 돌의 번호 노드를 구분하기 위한 배열이다. bool 자료 형으로 배열을 생성하여, 방문한 경우 false, 방문하지 않은 경우 true를 저장해두어 쉽게 구분할 수 있게 하였다.

* 평가 함수

double evaluation(int num) {

int tmp = 0;

if (stone[1] == 1) return 0;

if (num == 1) {

for (int i = 1; i <= num\_of\_stones; i++)

if (stone[i] = 1) tmp++;

if (tmp % 2 == 0) return -0.5;

else return 0.5;

}

else if (is\_prime(num)) {

for (int i = 2; (i \* num) <= num\_of\_stones; i++)

if (stone[i\*num] == 1) tmp++;

if (tmp % 2 == 0) return -0.7;

else return 0.7;

}

else {

for (int i = 1; i <= num\_of\_stones; i++)

if ((stone[i] == 1) && is\_prime(i)) tmp++;

if (tmp % 2 == 0) return -0.6;

else return 0.6;

}

}

- 각 상황에 따라 가중치를 부여하는 함수이다.

- 1번돌을 가지고 가지 않은 경우, 0을 반환한다.

- 마지막에 1을 가지고 간 경우, 가지고 올 수 있는 남은 돌을 세서 홀수면 0.5, 짝수면 -0.5를 반환한다.

- 마지막에 소수를 가지고 간 경우, 소수의 배수의 개수를 세어서 홀수면 0.7, 짝수면 -0.7을 반환한다.

- 마지막에 소수가 아닌 돌을 가지고 간 경우, 소수의 개수를 세어서 홀수면 0.6, 짝수면 -0.6을 반환한다

- evaluation 함수를 통해 문제에서 주어진 상황에 따른 평가 점수를 적용한다.

- evaluation 함수를 MAX player 기준으로 설정하였고, MIN player의 차례의 경우 양수, 음수를 반대로 적용한다.

* 재귀호출을 통해 트리 탐색을 구현한 함수

double DFS(int pickstone, bool player) {

step++; //방문한 노드의 갯수를 계산한다

double val;

double tmp, cur = 0;

if (pickstone == 0) {

//게임이 하나도 진행이 안되었을때 - MAX\_player의 첫선택 결정

for (int i = 1; i < ((double)num\_of\_stones /2.0); i+=2) {

stone[i] = 0;

tmp = DFS(i, MAX\_PLAYER);

stone[i] = 1;

if (tmp > cur) {

finalstone = i;

cur = tmp;

}

}

return cur;

}

//------------------------------------------------------

if (player == MAX\_PLAYER) { //MAX\_PLAYER일때

int cnt = 0;

for (int i = pickstone \* 2; i <= num\_of\_stones; i+=pickstone) {

if (stone[i]) {

stone[i] = 0;

tmp = DFS(i, MIN\_PLAYER);

stone[i] = 1;

if (tmp < cur) {

finalstone = i;

cur = tmp;

}

cnt++;

}

if (!cnt) return 1.0;

else {

val = evaluation(finalstone);

return val;

}

}//MAX\_PLAYER일 때, 돌을 한번씩 뽑으며 재귀호출을 통해 트리를 방문한다

}//이중 최소값을 골라 본인의 선택을 한다

else { //MIN\_PLAYER일떄

int cnt = 0;

memset(insu, 0, 100);

make\_insu(pickstone);

for (int i = 1; i <= insu\_num; i++) {

if (stone[insu[i]]) {

stone[insu[i]] = 0;

tmp = DFS(insu[i], MAX\_PLAYER);

stone[insu[i]] = 1;

if (tmp > cur) {

finalstone = i;

cur = tmp;

}

cnt++;

}

}

if (!cnt)

return -1.0;

else {

val = 0 - evaluation(finalstone);

return val;

}

}//MIN\_PLAYER일 때, 돌을 한번씩 뽑으며 재귀호출을 통해 트리를 방문한다

}//이중 최댓값을 골라 본인의 선택을 결정한다

- 설계한 DFS 함수는 재귀 호출을 바탕으로 트리를 생성하여 각 노드를 방문하는 방식으로 설계된 함수이다.

- 게임이 진행되지 않았을 때 MAX player의 최초의 선택, 게임이 진행되었을 때 각 플레이어의 차례일 때 를 구분하여 계산한다.

- 재귀적으로 자식노드들을 탐색한 이후에 본인에게 가장 유리한 선택을 하게 된다.

6. 전체 소스코드

#include <iostream>

#include <algorithm>

#include <cstring>

#include <utility>

#include <vector>

#include <cmath>

#include <cstdio>

#define MAX\_PLAYER 1

#define MIN\_PLAYER 0

using namespace std;

pair<int, double> p; //돌 & 가중치

vector<pair<int, double>> v;

bool stone[1000]; //1 갈수있음 0 못감

int num\_of\_stones; //돌의 갯수

int played;

int max\_depth = 0;

int step = 0;

int game\_played, finalstone;

int insu[100];

int insu\_num;

void make\_insu(int num) {

insu\_num = 0;

for (int i = 1; i < num; i++)

if (num % i == 0)

insu[++insu\_num] = i;

return;

}//주어진 수의 인수를 구하는 함수

bool is\_prime(int num) {

if (num == 1) return false;

for (int i = 2; i <= sqrt(num); i++) {

if (num % i == 0)

return false;

}

return true;

} //주어진 수가 소수인지 판별하는 함수

bool multiple(int num) {

int tmp = 0;

for (int i = num; i <= num\_of\_stones; i += i)

if (stone[i] == 1) tmp++;

return (tmp % 2);

} //주어진 수의 배수 갯수를 계산하는 함수

bool count\_prime(int num) {

int tmp = 0;

for (int i = 1; i <= num\_of\_stones; i++) {

if ((stone[i] == 0) && (is\_prime(i)))

tmp++;

}

return (tmp % 2);

} //현재 남아있는 소수 번호의 돌을 계산하는 함수

double evaluation(int num) {

int tmp = 0;

if (stone[1] == 1) return 0;

//1이 남아있는 경우

if (num == 1) { // 마지막으로 1번돌을 가져갔을 경우

for (int i = 1; i <= num\_of\_stones; i++)

if (stone[i] = 1) tmp++;

if (tmp % 2 == 0) return -0.5;

else return 0.5;

}

else if (is\_prime(num)) { // 소수의 돌을 가져갔을 경우

for (int i = 2; (i \* num) <= num\_of\_stones; i++)

if (stone[i\*num] == 1) tmp++;

if (tmp % 2 == 0) return -0.7;

else return 0.7;

}

else { //위의 상황외의 돌을 가져갔을 경우

for (int i = 1; i <= num\_of\_stones; i++)

if ((stone[i] == 1) && is\_prime(i)) tmp++;

if (tmp % 2 == 0) return -0.6;

else return 0.6;

}

} //가중치를 포함하는 평가 함수

//재귀호출을 통해 깊이 우선 탐색을 실행하는 함수

double DFS(int pickstone, bool player) {

step++; //방문한 노드의 갯수를 계산한다

double val;

double tmp, cur = 0;

if (pickstone == 0) { //게임이 하나도 진행이 안되었을때 - MAX\_player의 첫선택 결정

for (int i = 1; i < ((double)num\_of\_stones /2.0); i+=2) {

stone[i] = 0;

tmp = DFS(i, MAX\_PLAYER);

stone[i] = 1;

if (tmp > cur) {

finalstone = i;

cur = tmp;

}

}

return cur;

}

if (player == MAX\_PLAYER) { //MAX\_PLAYER일때

int cnt = 0;

for (int i = pickstone \* 2; i <= num\_of\_stones; i+=pickstone) {

if (stone[i]) {

stone[i] = 0;

tmp = DFS(i, MIN\_PLAYER);

stone[i] = 1;

if (tmp < cur) {

finalstone = i;

cur = tmp;

}

cnt++;

}

if (!cnt) return 1.0;

else {

val = evaluation(finalstone);

return val;

}

}//MAX\_PLAYER일 때, 돌을 한번씩 뽑으며 재귀호출을 통해 트리를 방문한다

}//이중 최소값을 골라 본인의 선택을 한다

else { //MIN\_PLAYER일떄

int cnt = 0;

memset(insu, 0, 100);

make\_insu(pickstone);

for (int i = 1; i <= insu\_num; i++) {

if (stone[insu[i]]) {

stone[insu[i]] = 0;

tmp = DFS(insu[i], MAX\_PLAYER);

stone[insu[i]] = 1;

if (tmp > cur) {

finalstone = i;

cur = tmp;

}

cnt++;

}

}

if (!cnt)

return -1.0;

else {

val = 0 - evaluation(finalstone);

return val;

}

}//MIN\_PLAYER일 때, 돌을 한번씩 뽑으며 재귀호출을 통해 트리를 방문한다

}//이중 최댓값을 골라 본인의 선택을 결정한다

int main(void) {

char tmp[10];

int stonetmp;

int value = 0;

double res = 0, ret;

scanf("%s", tmp);

fflush(stdin);

if (strcmp(tmp, "play"))

exit(0);

cin >> num\_of\_stones;

for (int i = 1; i <= num\_of\_stones; i++)

stone[i] = 1;

cin >> game\_played; //게임이 얼마나 진행되었는지를 입력받음

if (game\_played) { //게임이 한번이라도 진행되었을때

for (int i = 1; i <= game\_played; i++) {

cin >> stonetmp;

stone[stonetmp] = 0;

} //진행 정도를 입력받음

for (int i = 1; i <= num\_of\_stones; i++) {

if (stone[i] && (((!(game\_played % 2) == MAX\_PLAYER) && (stonetmp % i == 0)) || (((!game\_played % 2) == MIN\_PLAYER) && (i % stonetmp == 0)))) {

stone[i] = 0;

ret = DFS( i , !(game\_played % 2));

stone[i] = 1;

if ((!(game\_played % 2) == MAX\_PLAYER && (ret < res)) || (!(game\_played % 2) == MIN\_PLAYER && (ret > res))) {

res = ret;

finalstone = i;

}

}

}

}

else { //게임이 한번도 진행 안되었을때

res = DFS(0, MAX\_PLAYER);

} //MAX\_PLAYER의 초기 선택을 결정한다

cout << "Best Move : " << finalstone << '\n';

cout << "Calculated Value : " << res << '\n';

cout << "Number of Visited Nodes : " << step << '\n';

cout << "Max depth : " << max\_depth << '\n';

return 0;

}