알고리즘_과제_11

(Knap-sack_branch_and_bound)

학부: 컴퓨터학부

학번: 20162518

출석번호 : 156번

이름 : 최승서

이번 과제를 통해 Knap-sack 문제를 Depth-First와 Breadth-First, Best-First이 3가지 방법을 통해 풀어 보았다.

먼저 Knap-sack 문제란 배낭에 담을 수 있는 무게의 최댓값(W)이 정해져 있고, 일정 가치(profit)와 무게(WEIGHT)가 있는 짐(ITEM)들을 배낭에 넣을 때, 가치(profit)의 합이 최대가 되도록 짐을 고르는 방법을 찾는 문제이다.

첫번째로 Depth-First 는 깊이 우선 탐색인데, Back Tracking 기법을 사용한다. 우선 깊이 제한에 도달할때까지 목표 노드를 발견하지 못하면 가장 최근의 부모 노드로 돌아와서 다른 자식 노드로 내려가는데 이때 부모 노드로 돌아오는 과정을 Back Tracking 기법이라고 한다.

다음으로는 Breadth-First 이다. Breadth-First 는 너비 우선 탐색인데 먼저 제일 끝까지 내려가는 Depth-First 와 달리 해당 깊이의 노드들을 모두 탐색한 후 다음 깊이로 넘어가는데, 재귀적으로 작성할 수 있는 깊이 우선 탐색과는 달리 방문한 노드를 차례대로 저장한 후 꺼낼 수 있게 해주는 자료구조인 Queue 를 사용해야 한다.

마지막으로 Best-First 는 Breadth-First 를 개량한 알고리즘인데, 사실 Breadth-First 는 Depth-First 와 비교해서 좋은점이 없다. 하지만 각 노드의 한계값을 계산하여 그중 한계값이 가장 좋은 노드를 우선적으로 확장해 나가면 앞선 2 개의 탐색 알고리즘보다 최적값을 보다 빨리 찾을 수 있다.

다음은 손으로 직접 각 3 개의 탐색 방법을 그려 보았다.

〈소스코드〉

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define N 5 //아이템 갯수
#define W 9 //총 무게
int bestset[N+1]= {0, };
int include[N+1]= {0, };
int maxprofit =0;
int numset =0;
int cnt =1;
/**********************아이템 노드 구조체*******************************/
typedef struct{
    int level;
    int profit;
    int weight;
    int bound;
}node;
/**********************queue*****************/
node queue[100];
int head;
int tail;


void enqueue(node n)
{
    queue[tail] = n;
    tail++;
}
node dequeue()
{
    return queue[head++];
}

int empty()
{
    return head == tail;
}
/*****************************************/
int items[N+1][2] = { //아이템값 profit, weight
    {0, 0},
    {20, 2},
    {30, 5},
    {35, 7},
    {12, 3},
    {3, 1},
};

int promising(int i, int profit, int weight){
    int j=0,k=0;
    int totalweight =0;
    int bound;

    if (weight>=W)
        return FALSE;
    else{
        j = i+1;
        bound = profit;
        totalweight = weight;
        while((j<N+1) && (totalweight + items[j][1] <= W)){
            totalweight = totalweight+ items[j][1];
            bound = bound +items[j][0];
            j++;
        }
        k = j;

        if(k<N)
            bound = bound + (W-totalweight)*items[j][0]/items[j][1];
        return bound>maxprofit;

    }
}
void knapsack_DFS(int i, int profit, int weight){
    printf("%d.  %d depth  %d  %d \n", cnt++, i, profit, weight);

    if((weight <= W) && (profit > maxprofit)){ //maxprofit 갱신
```

```
        maxprofit = profit;
        numset = i;
        for(int k =0; k<N; k++) //매번 갱신해줘야됨
            bestset[k] = include[k];
    }
    if(promising(i, profit, weight)){
        //해당 깊이를 포함했을때
        include[i+1]= 1;
        knapsack_DFS(i+1, profit+items[i+1][0], weight + items[i+1][1]);
        //포함 안했을때
        include[i+1] = 0;
        knapsack_DFS(i+1, profit, weight);
    }
}


int bound(node n){
    int j=0, k=0;
    int totalweight =0;
    int result =0;
    if(n.weight>=W)
        return 0;
    else{
        result =  n.profit;
        j = n.level+1;
        totalweight = n.weight;
        while((j<=N) && (totalweight + items[j][1] <= W)){
            totalweight += items[j][1];
            result += items[j][0];
            j++;
        }
        k=j;
        if(k<=N)
            result= result+(W-totalweight)*items[k][0]/items[k][1];
        return result;
    }
}

void knapsack_BFS(){
    node v,u;
    int cnt=1;
    head =0;
    tail=0;

    v.level = 0; v.profit =0; v.weight =0;
    maxprofit =0;
    enqueue(v);
    while(!empty()){
        v = dequeue();
        //printf("%d  %d  %d \n", v.level, v.profit, v.weight);
        u.level=v.level+1;
        //포함했을때
        u.weight= v.weight + items[u.level][1];
        u.profit= v.profit + items[u.level][0];
        if((u.weight<=W) && (u.profit> maxprofit))
            maxprofit = u.profit;
        printf("%d. 포함했음 %d depth  %d  %d  %d \n", cnt++, u.level, u.profit, u.weight, bound(u));
        if(bound(u)>maxprofit){

            enqueue(u);
        }
        //포함안했을때
        u.weight= v.weight;
        u.profit=v.profit;
        printf("%d. 포함안함 %d depth %d  %d  %d\n", cnt++, u.level, u.profit, u.weight, bound(u));
        if (bound(u)>maxprofit){

            enqueue(u);
        }


    }
}

void knapsack_BestFS(){
    node v,u;
    int cnt =1;
    head =0;
    tail=0;
```

```
    v.level = 0; v.profit =0; v.weight =0;
    maxprofit =0;
    enqueue(v);
    while(!empty()){
        v = dequeue();
        if(v.bound > maxprofit){
            u.level = v.level+1;
            //포함했을때
            u.weight= v.weight + items[u.level][1];
            u.profit= v.profit + items[u.level][0];

            if((u.weight<=W) && (u.profit> maxprofit))
                maxprofit = u.profit;

            u.bound = bound(u);

            printf("%d. 포함했음 %d depth  %d  %d  %d \n", cnt++, u.level, u.profit, u.weight, bound(u));
            if(u.bound >maxprofit){

                enqueue(u);
            }
            //포함안했을때
            u.weight= v.weight;
            u.profit=v.profit;
            u.bound = bound(u);
           printf("%d. 포함안함 %d depth  %d  %d  %d \n", cnt++, u.level, u.profit, u.weight, bound(u));
            if (u.bound > maxprofit){

                enqueue(u);
            }
        }

    }
}


int main() {
    printf("\n---------------------------------\n");
    printf("Depth first algorithm\n\n");
    knapsack_DFS(0, 0, 0);
    printf("\n최적이 되는 집합\n");
    for(int i = 1; i<=N; i++)
        printf("bestset[%d] : %d\n", i, bestset[i]);
    printf("\nMax Profit:  %d\n", maxprofit);
    printf("\n---------------------------------\n");
    maxprofit =0;
    printf("Breadth first algorithm\n\n");
    knapsack_BFS();
    printf("\nMax Profit:  %d\n", maxprofit);
    printf("\n---------------------------------\n");
    maxprofit =0;
    printf("Best first algorithm\n\n");
    knapsack_BestFS();
    printf("\nMax Profit:  %d\n", maxprofit);



    return 0;
}
```

〈실행화면〉

5-33) Depth-First

```
------------------------------
Depth first algorithm

1.  0 depth  0  0
2.  1 depth  20  2
3.  2 depth  50  7
4.  3 depth  85  14
5.  3 depth  50  7
6.  4 depth  62  10
7.  4 depth  50  7
8.  5 depth  53  8
9.  5 depth  50  7
10.  2 depth  20  2
11.  3 depth  55  9
12.  3 depth  20  2
13.  1 depth  0  0

최적이 되는 집합
bestset[1] : 1
bestset[2] : 0
bestset[3] : 1
bestset[4] : 0
bestset[5] : 0

Max Profit:  55

------------------------------
```

앞서 손으로 직접 그렸던 그래프와 동일하게 탐색 순서가 출력됨을 알 수 있다.
또한 최적이 되는 집합이 Item1, Item2인것을 알 수 있다.

6-1) Breadth-First

```
--------------------------------
Breadth first algorithm

1. 왼 1 depth  20  2  60
2. 오 1 depth 0  0  50
3. 왼 2 depth  50  7  60
4. 오 2 depth 20  2  55
5. 왼 2 depth  30  5  50
6. 오 2 depth 0  0  43
7. 왼 3 depth  85  14  0
8. 오 3 depth 50  7  58
9. 왼 3 depth  55  9  55
10. 오 3 depth 20  2  35
11. 왼 4 depth  62  10  0
12. 오 4 depth 50  7  53

Max Profit:  55
```

넓이 우선 탐색도 그래프와 동일한 순서로 출력되고, Max profit이 55임을 알 수 있다.

6-4) Best-First

```
------------------------------
Best first algorithm

1. 왼 1 depth  20  2  60
2. 오 1 depth  0  0  50
3. 왼 2 depth  50  7  60
4. 오 2 depth  20  2  55
5. 왼 3 depth  85  14  0
6. 오 3 depth  50  7  58
7. 왼 3 depth  55  9  55
8. 오 3 depth  20  2  35
9. 왼 4 depth  62  10  0
10. 오 4 depth  50  7  53

Max Profit:  55
Program ended with exit code: 0
```

All Output Filter

마지막으로 Best-First도 마찬가지로
그래프와 동일한 순서로 출력됨을 소스코드를 통해 보였다.

위의 그래프와 소스코드 출력 결과를 통해 각 Depth-First와 Breadth-First의 검색 노드 개수는 13임을 확인했고 Best-First는 검색 노드의 개수가 11개로 앞선 2개의 탐색 방법보다 검색 노드의 개수가 적음을 확인했다. 이를 통해 Best-First 방법이 최적값을 가장 빨리 찾는 것을 알 수 있었다.