Programming Report #2

0-1 Knapsack Problem

과목	: 컴퓨터알고리즘
담당교수	: 이상호
학과	: 컴퓨터공학과
학번	: 1829008
이름	: 김민영
이메일	: kmy8759@ewhain.net
제출일자	: 2020년 6월 7일

[Programming Report #2] : 0-1 Knapsack Depth-first Search with branch-and-bound

1) 문제설명

0-1 Knapsack 문제를 (3) Best-First Search with Branch & Bound 방법을 선택하여 파일로 주어지는 각 입력자료에 대해 최대 이윤과 해 벡터를 출력하는 프로그램을 작성한다. 위 방법에 의해 이문제를 해결하려면, Pi/Wi 를 큰 순서로 정렬하는 전처리 과정이 필요한데, 이를 위한 정렬 알고리즘은 내림차순 정렬을 하는 삽입정렬(Insertion Sort)를 사용한다.

*0-1 Knapsack 문제란?

배낭에 담을 수 있는 무게의 최댓값이 정해져있고, 일정 가치와 weight가 있는 짐들을 배낭에 넣을 때, 가치의 합이 최대가 되도록 고르는 방법을 찾는 문제를 Knapsack Problem이라고 한다. 이 중에서도 짐을 쪼갤 수 있는 경우와 쪼갤 수 없는 경우로 나뉘는데 0-1 Knapsack Problem이 바로 쪼갤 수 없는 경우에 속하는 경우이다.

2) 입출력의 예

(1) 입력 자료 형식의 예 : 파일형식(p2data0.txt ~ p2data6.txt)

* p2data0.txt

```
4       // N 개수 ( # of objects)
50 40 10 30     // Pi
10 2 5 5     // Wi
16 // Knapsack capacitiy M
```

(2) 출력 자료 형식의 예

```
<p2data0.txt>
N = 4
pi = 50 40 10 30
wi = 10 2 5 5
pi/wi = 5.0 20.0 2.0 6.0
M = 16

The maximum profit is $90
The solution vector X = (1, 1, 0, 0)   // X=(x1,x2, … xn) 원래의 인덱스 순서
```

3) 문제풀이방법(알고리즘)

전체적인 문제풀이 순서

0. 우선 txt 입력파일을 읽어와 kanpsack을 초기화시킨다.

1. pi/wi를 삽입정렬을 이용하여 내림차순으로 정렬하여 놓는다

2. queue를 초기화 시킨다.

3. Best-First Search with Branch & Bound 방법을 사용하여 해를 구한다.

각 함수설명

read_file 함수 : 파일을 읽어와 Knapsack 의 pi, wi, pi/wi, M 을 저장한다.

Print_file 함수 : 위에서 읽어온 Knapsack 의 pi, wi, pi/wi, M 을 출력한다.

Sort 함수 : Insertion Sort(삽입 정렬)에 해당하는 함수이다.

삽입정렬이란? 모든 요소를 앞에서부터 차례대로 이미 정렬된 배열 부분과 비교 하여, 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘이다. 매 순서마다 해당 원소를 삽입할 수 있는 위치를 찾아 해당 위치에 넣는다.

Queue_init() 함수 : queue 를 생성하는 함수이다.

Is_empty 함수 : 큐가 비어있는지 확인하는 함수이다. 비어있으면, True 비어있지 않으면 False 를 반환한다.

Insert 함수 : Queue 에 Element 를 삽입하는 함수이다. (Enqueue)

Delete_ 함수 : Queue 에서 가장 큰 Element 를 삭제하고 이를 반환하는 함수이다. (Dequeue)

Best_BB 함수 : 우선순위 큐를 이용하여 Best First Search with branch and bound 알고리즘을 수행하는 함수이다.

Bound 함수 : bound 의 값을 계산하여 반환하는 함수이다.

Print_answer 함수 : 정답에 해당하는 Knapsack 의 Max Profit 과, 해벡터를 출력한다.

추가로, 상태공간트리에 사용되는 노드는 Element를 이용하여 만들어 놓는다. a,b는 상태공간 트리의 좌표이고 value는 상태공간 트리의 값이며, mm은 현재까지의 채워진 용량이다. 또한 해벡터까지 저장해놓는다.

<Key idea> Best-First Search with Branch & Bound 방법

Promising 한지 검사를 할 때에는 bound, totweight를 구하고 넘치지 않았는지를 판단하여 non Promising한지도 체크를 해주면 된다.

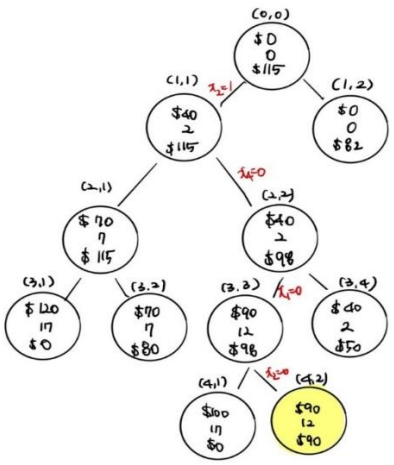
(1) Weight >=M

(2) bound <= maxprofit(current best solution), where $bound = (profit + \sum_{j=i+1}^{n} P_j) + (M - totweight) \times (P_{k+1}/W_{k+1})$, $totweight = weight + \sum_{j=i+1}^{n} W_j$

또한, 이 문제에서 최적의 해답에 빨리 도달하기 위한 전략이 있는데 이는 다음과 같다. 주어진 노드의 모든 자식노드를 검색하확장되지 않은 노드를 살펴보고 그 중에서 가장 좋은 bound를 가진 node를 확장한다. 이때, best bound를 가진 노드를 우선적으로 선택하기 위해서 우선순위큐를 사용한다. 우선순위 큐는 heap을 사용하여 구현하면 된다. 이의 슈도코드는 아래와 같다.

```
void best_b&b (T, best) {
  node u, v;
  initialize (PQ);          /* PQ means the Priority Queue, e.g., a Heap */
  v = root of T;            /* T is a state space tree : implicit */
  insert(PQ, v);
  best = value(v);
  while (!empty(PQ)) {
     delete(PQ, v);
     if (bound(v) is better than best)
       for (each chid u of v in the left-to-right order) {
         if (value(u) is better than best)
            best = value(u);
         if (bound(u) is better than best)
            insert(PQ, u);
     }
   }
}
```

이를 기반으로 앞의 예제에 대한 Pruned state space tree를 그려보면 아래와 같다.



4) 소스코드(소스코드.cpp)

```
#include <stdio.h>
#include <stdlib.h>

#define file_len 7 //pdata0.txt~pdata6.txt 의 갯수
#define TRUE 1
#define FALSE 0

int N;
int M;

typedef struct knapsack {
	int index;
	int pi;
	int wi;
	float pi_wi;
	int X;
}knapsack;

typedef struct element {
	int a, b; // (a,b) 좌표
	int value; // 현재까지의 value
	int current_sum; //현재까지의 합
	float bound; //bound 값
	int *X; // 해벡터
}Element;

typedef struct heaptype {
	Element *heap;
	int size;
} HeapType;

knapsack *kanpsack;
HeapType PQ;

void read_file();
void print_file();
void sort();
void queue_init(void);
int is_empty();
void insert(Element node);
Element delete_();
Element bound(int a, int b);
Element best_BB(void);
void print_answer(Element node);

int main(void) {
	read_file(); // 파일을 읽어와 knapsack 저장
	sort(); // 삽입정렬을 이용하여 pi/wi 내림차순으로 정렬
	queue_init(); //queue 초기화
	print_answer(best_BB()); //Best-First Search With Branch&Bound 방법을 사용하여
해를 구하고 이를 출력한다.
	return 0;
}
```

```
void read_file() {
	FILE *fp;
	fp = fopen("C:/Users/KimMinyoung/Downloads/p2data/p2data0.txt", "r"); // txt
파일 읽어오기
	if (fp == NULL) { // 파일을 읽어오지 못하는 경우에 대한 예외처리
		printf("could not read file");
		return;
	}
	// N 읽어오기
	fscanf(fp, "%d", &N);
	kanpsack = (knapsack*)malloc(sizeof(knapsack)*N);
	// index 지정
	for (int i = 0; i < N; i++)
		kanpsack[i].index = i;
	// pi 읽어오기
	for (int i = 0; i < N; i++)
		fscanf(fp, "%d ", &kanpsack[i].pi);
	// wi 읽어오기
	for (int i = 0; i < N; i++)
		fscanf(fp, "%d ", &kanpsack[i].wi);
	// pi/wi 계산
	for (int i = 0; i < N; i++)
		kanpsack[i].pi_wi = (float)kanpsack[i].pi / (float)kanpsack[i].wi;
	// M 읽어오기
	fscanf(fp, "%d", &M);
	printf("< pr2data0.txt> ");
	print_file();
}

// 입력한 file data를 출력하는 함수
void print_file() {
	int i;
	printf("\n");
	printf("N = %d", N);
	printf("\n");
	printf("pi = ");
	for (i = 0; i < N; i++) {
		printf("%d ", kanpsack[i].pi);
	}
	printf("\n");
	printf("wi = ");
	for (i = 0; i < N; i++) {
		printf("%d ", kanpsack[i].wi);
	}
	printf("\n");
	printf("pi/wi = ");
	for (i = 0; i < N; i++) {
		printf("%.1f ", kanpsack[i].pi_wi);
	}
	printf("\n");
	printf("M = %d", M);
	printf("\n");
}

// Insertion Sort 를 이용하여 정렬
void sort() {
	float key_value;
	knapsack key;
	int j;
	for (int i = 1; i < N; i++) {
		key_value = kanpsack[i].pi_wi;
		key = kanpsack[i];
		// 정렬 완료 : i-1 이전
```

```
                    // 정렬 미완료 : i-1 이후
                    // key 값보다 정렬된 배열에 있는 값이 크면 j번째를 j+1번째로 이동
                    for (j = i - 1; j >= 0 && kanpsack[j].pi_wi <= key_value; j--) {
                              kanpsack[j + 1] = kanpsack[j];
                    }
                    kanpsack[j + 1] = key;
          }
}


// queue 초기화
void queue_init() {
          PQ.heap = (Element*)malloc(sizeof(Element)*N);
          PQ.size = 0;
}
// 큐가 비어있는지 확인
// 비어있으면 True, 비어있지 않으면 False 반환
int is_empty() {
          if (PQ.size == 0)
                    return TRUE;
          else
                    return FALSE;
}

// 삽입 (enqueue) 함수
void insert(Element node) {
          Element temp;
          int i = PQ.size;  // queue의 크기
          int parent = (i - 1) / 2; // 부모의 인덱스
          PQ.heap[i] = node;
          PQ.size++; // queue의 크기 증가
          // 해당하는 부분을 찾아서 삽입한다.
          while ((PQ.heap[i].value > PQ.heap[parent].value) && (i > 0)) {
                    temp = PQ.heap[i];
                    PQ.heap[i] = PQ.heap[parent];
                    PQ.heap[parent] = temp;
                    i = parent;
                    parent = (i - 1) / 2;
          }
}

// 삭제 : dequeue 함수
Element delete_() {
          Element result = PQ.heap[0]; //제일 위에있는 루트노드 반환할 노드
          Element tmp;
          int i = 0, big = 0, cleft = 1, cright = 2;
          PQ.size--; // 삭제하였으므로 queue의 크기 감소
          PQ.heap[0] = PQ.heap[PQ.size];
          while ((cleft < PQ.size) && (cleft < PQ.size)) {
                    // 더 작은 자식노드를 찾는 과정
                    if (PQ.heap[cleft].value > PQ.heap[cright].value) {
                              tmp = PQ.heap[i];
                              PQ.heap[i] = PQ.heap[cleft];
                              PQ.heap[cleft] = tmp;
                              i = cleft;
                    }
                    else {
                              tmp = PQ.heap[i];
                              PQ.heap[i] = PQ.heap[cright];
                              PQ.heap[cright] = tmp;
                              i = cright;
                    }
                    cleft = 2 * i + 1; //왼쪽 자식노드
```

```
                    cright = 2 * i + 2; //오른쪽 자식노드
          }
          if (cright == PQ.size) {
                    tmp = PQ.heap[i];
                    PQ.heap[i] = PQ.heap[cleft];
                    PQ.heap[cleft] = tmp;
                    i = cleft;
                    cleft = 2 * i + 1; //왼쪽 자식노드
                    cright = 2 * i + 2; //오른쪽 자식노드
          }
          return result;
}

//Best First Search With Branch And Bound
Element best_BB(void) {
          int best; // best 값
          Element u, v, rnode;
          int i;
          v = bound(0, 0); //(0,0)을 루트로 하여 먼저 넣는다.
          insert(v);
          best = v.value;
          // Back Tracking 방법을 사용하여 해를 찾는 과정
          while (!is_empty())
          {
                    v = delete_();
                    for (i = 0; i < 2; i++) {
                              u = bound(v.a + 1, 2 * v.b + i);
                              if (u.current_sum <= M) {
                                        // 넘치지 않는 경우 best 값을 업데이트 시켜준다.
                                        if (u.value > best) {
                                                  best = u.value;
                                                  rnode = u;
                                        }
                                        if (u.bound > best) {
                                                  insert(u);
                                        }
                              }
                    }
          }
          rnode.value = best;
          return rnode;
}

//Bound 의 값을 계산하는 함수
Element bound(int a, int b) {
          int enable_capacity;
          Element node;
          float c;
          node.a = a;
          node.b = b;
          node.X = (int*)malloc(sizeof(int)*a);
          // X 에 해당값이 들어가면 1 을 넣고 그렇지 않으면 0 을 넣는다.
          for (int i = 0; i < node.a; i++) {
                    node.X[i] = (b >> (node.a - i - 1) & 1);
          }
          node.value = 0;
          node.current_sum = 0;
          for (int i = 0; i < node.a; i++) {
                    // 들어가 있는 경우
                    if (node.X[i] == 1) {
                              node.value += kanpsack[i].pi;
                              node.current_sum += kanpsack[i].wi;
                    }
```

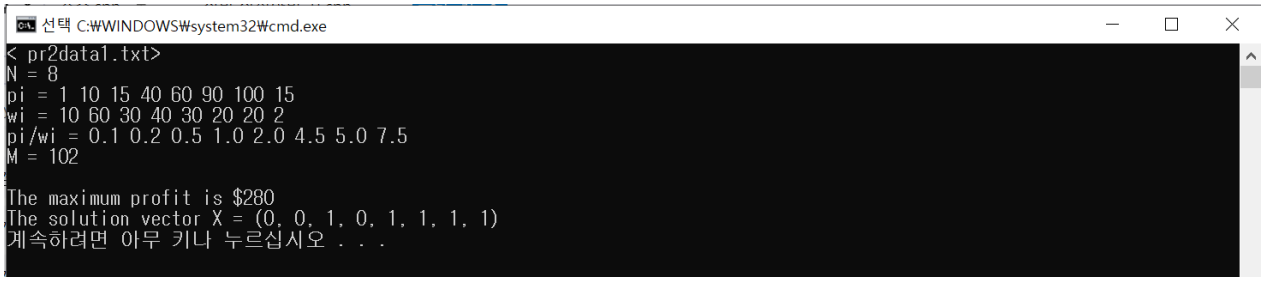
```
        }
        int i = node.a;
        node.bound = node.value;
        enable_capacity = M - node.current_sum;
        // bound 계산
        while ((enable_capacity > 0) && (i < N)) {
                if (enable_capacity >= kanpsack[i].wi) {
                        enable_capacity -= kanpsack[i].wi;
                        node.bound += kanpsack[i].pi;
                        i++;
                }
                else {
                        c = (float)enable_capacity / (float)kanpsack[i].wi;
                        enable_capacity = 0; // 딱 맞게 채워넣는다.
                        node.bound += (float)kanpsack[i].pi * c;
                }
        }
        return node;
}


// 정답을 출력하는 함수
void print_answer(Element node) {
        for (int i = 0; i < N; i++) {
                if (node.X[i] == 1) {
                        kanpsack[kanpsack[i].index].X = 1;
                }
                else {
                        kanpsack[kanpsack[i].index].X = 0;
                }
        }
        // Maximum profit 출력
        printf("\n");
        printf("The maximum profit is $%d", node.value);
        //해벡터 출력
        printf("\nThe solution vector X = (");
        for (int i = 0; i < N - 1; i++) {
                printf("%d, ", kanpsack[i].X);
        }
        printf("%d)\n", kanpsack[N - 1].X);
}
```

5) 수행결과(캡쳐화면)

(1) p2data0.txt

-> max profit : 90

(2) p2data1.txt

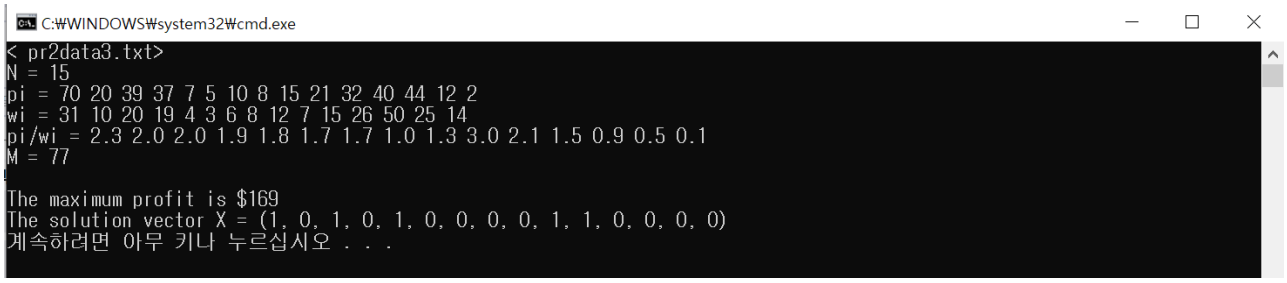


-> max profit : 280

(3) p2data2.txt

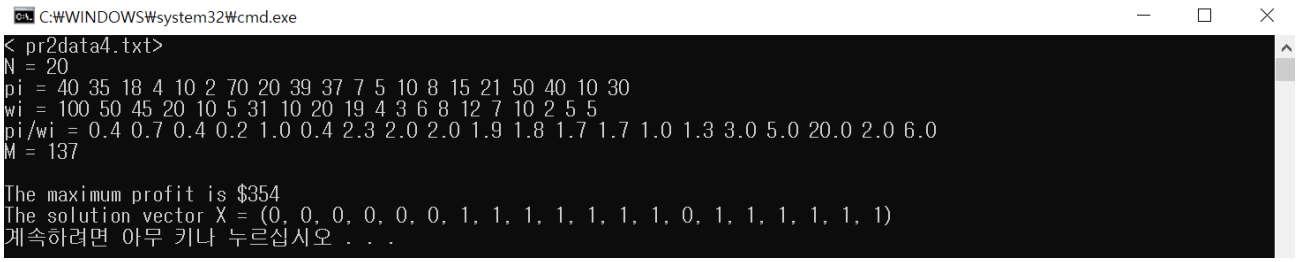
-> max profit : 142

(4) p2data3.txt



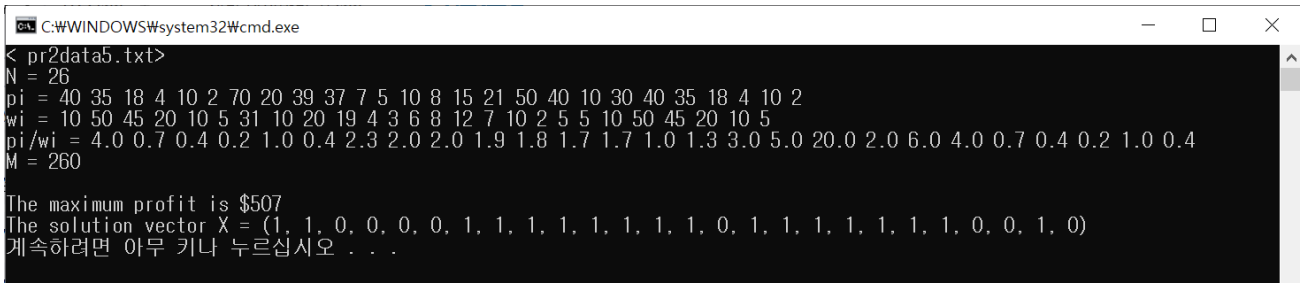
-> max profit : 169

(5) p2data4.txt



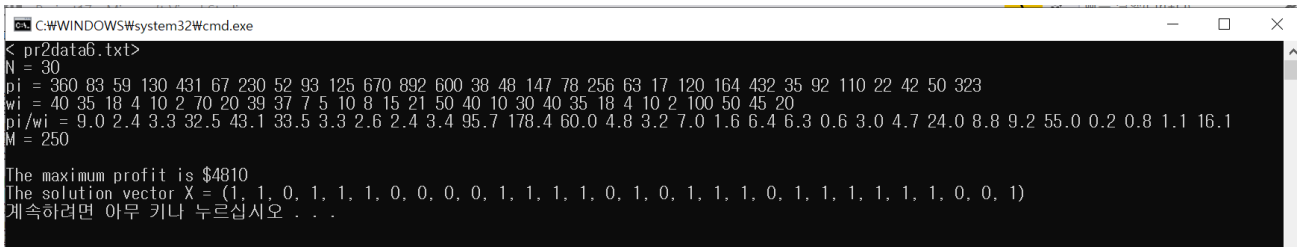
-> max profit : 354

(6) p2data5.txt



-> max profit : 507

(7) p2data6.txt

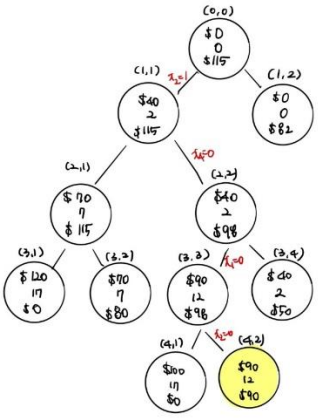


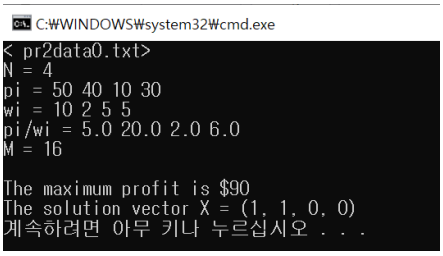
-> max profit : 4810

6) 결과분석 및 토의

모든 경우의 수를 다 따져서 Pruned state space tree를 모두 생성하면, 굉장히 비효율적인데, Best-First Search with Branch & Bound 방법을 통해 필요한 노드만 생성하여 효율적으로 해를 구할 수 있었다. 즉, 큐에서 빼고나서 non-promising 일 수도 있는 경우 더 이상 생성할 필요 없이 효과적으로 prunning을 한다.

위에서 실행한 결과화면을 손으로 풀이해본결과, 맞았음을 확인할 수 있었다.





나는 3번 방법을 이용하여 문제를 풀었는데, 1번 – 2번 방법에 대해서도 살펴볼 필요가 있을 것 같다. 1번 방번은 "Depth-Frist Search" with branch – and – bound prunning 이다. 이는 최적화 문제에 대해 정확히 백트래킹과 일치한다. 2번 방법은 breadth-first search with branch-and-bound pruning 이다. 이는 FIFO B&B라고 불린다. 1번 방법은 Stack을 이용하는 방식이고 2-3번 방식은 Queue를 이용하는 방식이다. 이 중에서 3번방법이 가장 빠르게 최적해에 도달할 수 있다.