과목: 알고리즘

교수: 정민영 교수님

Algorithm [Assignment #3]

- Implementation of 0-1 Knapsack Problem -

홍지훈

이름: 홍지훈

학과: 소프트웨어학부

분반: 나

학번 : 20201777

과제: Backtracking 기법과 Branch-and-bound 기법을 활용하여 0-1 knapsack 문제 해결

주제 및 목표: 0-1 knapsack 문제에서 최적의 solution 을 얻는 알고리즘들을 구현하고 평가함

Backtracking 알고리즘은 되추적알고리즘 이라고도 하는데, 트리에서 깊이 우선 탐색(DFS)을 진행하고 각 노드가 유망한지 검사, 유망하지 않으면 그 노드의 부모 노드로 돌아가서 다른 노드를 검사하는 식으로 되추적하는 알고리즘이다. 기존 깊이 우선 탐색만 사용하는 것보다 훨씬 적은 노드를 방문하여 답을 찾아낼 수 있다.

Branch-and-bound 알고리즘은 분기한정법으로 backtracking 과 유사하지만 각 트리를 방문하는데 특정한 경로로 제한되지 않으며, 최적화 문제에서 주로 사용된다는 차이점이 있다. 이런 분기한정법 중 Knapsack Problem 에서는 Breadth-First Search 알고리즘과 Best-First Search 알고리즘을 사용한다.

Breadth-First Search 알고리즘은 BFS 라 불리는 너비 우선 탐색 알고리즘이다. 뿌리 마디부터 검색을 시작하고, 다음 수준에 있는 모든 마디를 검색하고 다음 수준으로 넘어간다. 이러한 특성 때문에 backtracking 기법에서는 사용하기 어려우며 Queue 를 사용한다.

Best-First Search 알고리즘은 BFS 와 큰 차인 없지만, 최적의 해답에 더 빨리 도달하기 위해 일반 Queue 가 아닌 Priority Queue(우선순위 큐)를 사용했다. 주어진 노드의 모든 자식 노드를 검색하고 유망하면서 확정되지 않은 노드를 살펴보고, 그중 가장 좋은 한계치를 가진 노드를 확장한다.

0-1 Knapsack Problem 은 배낭 문제이다. K kg 만큼 담을 수 있는 가방에 N 개의 물건(각 물건은 가치 p 와 무게 w 를 가지고 있다)을 잘 조율해서 최대한 많은 가치를 담는 문제이다. 0-1 배낭 문제는 일반 배낭 문제와 다르게 물건을 나눌 수 없기 때문에 일반적인 그리디 알고리즘으론 최적해를 구할 수 없으며, 백트래킹이나 bfs 등의 방법을 사용하여 풀어야 한다.**구현**

기본적으로 구현은 C++을 사용하였고, 예시코드와 강의자료의 코드를 참고하였다.

기본단위인 node 의 구조이다.

```
1.  struct node {
2.      int profit; // 물건의 가치
```

```
3.      int weight; // 물건의 무게
4.      float value; // 무게대비 가치 (profit / weight)
5.      int level; // value rank
6.      float bound; // 한계치
7.  };
```

Class Queue: 직접 구현한 일반 Q 이다. 기본적으로 BFS 에서 사용하는 push, pop 만 구현하였다.

Int N: 물건의 개수 (1 <= N <= 20)

Int K: 가방에 담을 수 있는 최대 무게(1 <= K <= 20000)

Node V[]: 각 물건의 정보.

Int maxprofit: 현재까지 구해진 최대 가치, 초기값은 0.

Data Input

```
1.  void Input() {
2.      cin >> N >> K;
3.      node temp[25];
4.      for(int i = 0; i < N; i++)  {
5.          node tempnode;
6.          cin >> tempnode.weight >> tempnode.profit;
7.          tempnode.value = (float)tempnode.profit / tempnode.weight;
8.          insert(temp, tempnode, i);
9.      }
10.     // heapsort
11.     for(int i = 0; i < N; i++) {
12.         V[i+1] = remove(temp, N-i);
13.         V[i+1].level = i+1;
14.     }
15. }
16.
```

입력데이터를 받는 과정에서도 간단하지 않았다. 기본적으로 knapsack problem 을 풀기 위해서는 무게당 가치가 큰 순으로 정렬하는 게 기본이다. 그러기 때문에 weight 와 profit 을 받으면서 계산하기 쉽게 value(profit / weight)도 같이 구해줬다. 이를 priority Queue 에 넣어주고, 모든 입력이 끝나게 되면 heap sort 과정을 거쳐서 입력을 받을 때 넣어주었던 priority Queue 에서 모든 노드를 빼서, 순서대로 V[i]에 넣어주게 된다.

Backtracking 구현

```
1.  bool promising(int i, int profit, int weight)
2.  void BTknapsack(int i, int profit, int weight)
3.
```

기본적으로 backtracking 에서 사용한 두 개의 함수이다.

Backtracking 은 재귀 함수를 사용하므로 각각 함수의 멤버변수로 현재까지 구해진 weight 와 profit 을 가져온다. BTknapsack 에서는 현재까지 구해진 profit 이 maxprofit 보다 크면 maxprofit 에 profit 을 대입시켜준다.

그리고 promising 을 호출하는데, promising 에서는 현재 weight 와 profit 이 어디까지 구할 수 있는지 한계점을 찾는다. 여기서 최대로 구할 수 있는 가치가 maxprofit 을 넘으면 다시 BTkanpsack 으로 돌아와서 i 번째 profit 이 있을 때와 없을 때를 각각 재귀 함수로 다시 호출한다. 이런 식으로 한계점이 닿는 한 모든 노드를 방문하여 최대 profit 을 찾게 된다.

Bound 구현

Branch-and-bound 기법을 사용하기 위해서는 bound 함수를 구현해야 했다.

Bound 란 해당 노드에서 얼마나 더 많은 가치를 가질 수 있는지 구하는 함수이다. 본 코드에서는 교재를 참고하여 totweight 가 K 보다 작을 때까지, profit 을 더해주고, 남은 공간은 profit/weight 를 곱하는 형태로 한계점을 구하였다. 이런 식으로 한계점을 구하게 되면 이 한계점을 사용하여 해당 노드의 가치를 구하고, 이후 노드를 탐색할 건지 안 할건지에 대한 여부를 결정 내리게 된다.

Breadth-First Search 구현

Breadth-First Search 알고리즘은 일반 Queue 를 사용한다. 명세에서는 Breadth-First Search 와 Best-First Search 를 각각 구현하라고 명시되어있기 때문에 Breadth-First Search 를 사용하기 위해서는 Priority Queue 가 아닌 일반 Queue 또한 구현해야 했다. 2-1 자료구조 시간에 사용했던 Queue 구조를 참고하여 필요한 기능만 있는 class Queue 를 만들었다.

Queue 구조에선 size 를 측정하는 함수를 따로 만들지 않아 함수 내에서 따로 측정해줬는데, 이 Qsize 변수를 사용하여 while 문을 돌려주었다. while 문에서는 각각 Q 에 있는 node 를 하나씩 뺀 후 node 의 데이터와 node 의 다음 데이터를 더해서 u node 를 만든다. 이 u node 는 bound 를 측정하는 기준이 되고, 해당 노드의 한계점이 maxprofit 을 넘을 가능성이 보이게 되면 Q 에 해당 node 를 추가하여 다음 while 문이 돌아갈 수 있도록 만들어준다.

또한 backtracking 기법과 유사하게 다음 node 를 넣지 않은 상태 또한 bound 를 측정하여, 동일하게 한계점이 maxprofit 을 넘을 가능성이 보이면 Q node 에 추가해준다.

Best-First Search 구현

Best-First Search 알고리즘은 Breadth-First Search 알고리즘에서 일반 Queue 가 Priority Queue 로 바뀐 정도의 차이점을 가진다. 그러므로 기존 bound 함수를 그대로 사용하며, 일반 Queue 가 아닌 Priority Queue 를 사용하기 위해 node 배열을 만들어 주었다.

원래는 Priority Queue 또한 class 로 구현하여 사용하려 했지만, Input 을 만드는 과정에서 이미 Priority Queue 를 배열 기준으로 구현해버렸고, 이를 다시 class 로 구현하기 번거로운 작업이었기 때문에 부득이하게 배열로 구현하게 되었다. 이 부분에서 공간복잡도가 조금 늘어나는 부분이 아쉬웠지만, 명세의 문제에서는 그렇게 많은 데이터를 사용하지 않기 때문에 (1 <= N <= 20) 성능에 크게 영향을 주지 않는다고 판단하였다.

나머지 부분은 거의 동일하며 여기서 차이점이라 하면 Priority Queue 에서 data 를 Pop 시킨 후 바로 bound 를 구한다. 구한 bound 를 다른 과정 없이 바로 maxrprofit 과 비교하여 한계점이 maxprofit 보다 높으면 비로소 Breadth-First Search 의 과정을 거쳐서 똑같이 구현하게 된다.

실험 결과 및 분석

Test case

```
4 7
6 13
4 8
3 6
5 12
Output: 14

```

해당 input 에대한 결과값으로 3 개의 방법 모두 알맞은 답을 도출해냈다.

각 방법에 따른 소요시간

```
Backtacking
real    0m0.068s
user    0m0.015s
sys     0m0.000s

Breadth
real    0m0.039s
user    0m0.000s
sys     0m0.015s

Best
real    0m0.039s
user    0m0.000s
sys     0m0.015s

```

각 방법에 따라 소요 시간은 크게 차이가 안 났지만, Branch-and-Bound 방식을 사용한 2 개의 방법은 DFS 를 사용한 Backtracking 방식보다 속도가 많이 빠른 것을 알 수 있다. Breadth-First

알고리즘보다 Best-First 알고리즘이 이론적으로 빠르지만, 해당 테스트케이스가 크지 않기 때문에 큰 차이를 가지지 못한 것으로 예상된다. 하지만 DFS 와 BFS 중 무엇이 더 좋다고 말할 수 없다.

또한 Best-FIrst 방식에서는 동적인 class 를 만들지 않고, 배열기준의 priority queue 를 만들었기 때문에 공간복잡도 상에서 원래보다 더 많은 공간을 차지할 수 있다.

결론

0-1 knapsack Problem 을 해결할 때는 P/W 순으로 정렬하는 것이 거의 필수적이다. 일반 knapsack problem 에서는 정렬한 후 단순한 greedy Algorithm 으로도 구현할 수 있지만, 0-1 knapsack problem 은 다른 방식을 사용해 주어야 한다.
Backtracking 방식은 DFS 를 사용한다. DFS 방식을 일반적으로 BFS 방식보다 느린 것으로 알려져 있고 Backtracking 방식으로 0-1 knapsack problem 을 해결할 순 있지만, 한계로 인해 이후 거론될 BFS 방식보다 일반적으로 효율이 떨어지는 게 사실이다.
Breadth-First Search 알고리즘에서는 bound 를 사용하여 유망성을 확인한다. Breadth-First Search 에서는 일반 Queue 를 사용하여 구하고 Best-First Search 알고리즘은 priority Queue 를 사용하여 구한다. 일반적으로 Best-First Search 알고리즘이 빠르지만 작은 값에서는 큰 차이가 없다.
Backtracking 방식과 Breadth-First Search 알고리즘, Best-First Search 알고리즘 모두 0-1 knapsack Problem 을 해결할 수 있으며, 일반적으로 Backtracking 방식보다 BFS 방식이 빠르지만, 절대적인 것은 아니다.