백트래킹 과제

https://m.blog.naver.com/bumsou10/220718783252

https://melomance.github.io/2011/12/24/[AG][BT] 0-1 Knapsack Problem/

OS: Window 10

Programming Language: Java 17

Compiler: javac

IDE: Intellij

CPU: Intel(R) Core(TM) i5-8265U CPU

Memory: 삼성 전자 ddr4 8.0GB * 2개

```
package Algorithm_class;

import java.util.Arrays;

public class KnapsackDFS {

    // Global variables and constants
    static final int n = 12;
    static final int W = 100;
    static int[] w = {0,5,8,15,20,25,30,35,40,45,50,55,60}; // w[0] is meaningless
    static int[] p = {0,10,20,30,35,50,60,80,90,95,100,110,120}; // p[0] is meaningless
    static String[] include = new String[n + 1];
    static String[] bestset = new String[n + 1];
    static int prmNodeCnt = 0;
    static int nprmNodeCnt = 0;
    static int numbest = 0;
    static int maxprofit = 0;

    public static void main(String[] args) {
        long start = System.currentTimeMillis();

        knapsack(0, 0, 0);

        long end = System.currentTimeMillis();

        for (int j = 1; j <= numbest; j++) {
            System.out.print("w" + j + ": " + bestset[j] + " ");
        }

        System.out.println("\nThe answer: $" + maxprofit);
        System.out.println("The number of promising nodes: " + prmNodeCnt);
        System.out.println("The number of non-promising nodes: " + nprmNodeCnt);
    }

    public static void knapsack(int i, int profit, int weight) {
        if (weight <= W && profit > maxprofit) {
            maxprofit = profit;
            numbest = i;
            System.arraycopy(include, 0, bestset, 0, n + 1);
        }

        if (promising(i, profit, weight)) {
            prmNodeCnt++; // Count the number of promising nodes
            include[i + 1] = "YES"; // Include w[i+1]
            knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
            include[i + 1] = "NO";  // Do not include w[i+1]
            knapsack(i + 1, profit, weight);
        } else {
            nprmNodeCnt++; // Count the number of non-promising nodes
        }
    }

    public static boolean promising(int i, int profit, int weight) {
        int j, k;
        int totweight;
        float bound;

        if (weight >= W) {
            return false;
        } else {
            j = i + 1;
            bound = (float) profit;
            totweight = weight;

            while (j <= n && totweight + w[j] <= W) {
                totweight = totweight + w[j];
                bound = bound + p[j];
                j++;
            }

            k = j; // Use k for consistency with formula in text.
            if (k <= n) // Grab fraction of kth item.
                bound = bound + (W - totweight) * ((float) p[k] / w[k]);

            return (bound > maxprofit);
        }
    }
}
```

knapsack Method

If the current Backpack state is more efficient than the stored backpack state (currentWeight <= maxWeight && currentProfit > maxProfit). We update the status of the backpack(arraycopy)

Next, check if the current state is promising (call promising method)
and then Divide the brand(using DFS) between using and not using the current item

Promising Method

The 'promising method' checks that nodes are promising.
If the current weight is bigger than the upper limit weight, Return 'false' Because it violates the condition

Fill in the remaining space to calculate the maximum benefit you can get from continuing to navigate this node. (bound)

This(bound) is the maximum benefit you will get from continuing to navigate this node. if 'bound' is smaller than 'maxProfit', you don't have to search for sub-nodes(It's Becase This Node Search is Not promising). so I use 'return bound > maxProfit'

So this way is Divide the brand and check if the state is promising. if not promising is checked we use backtracking

예제