ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ ПО КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

КОМПЮТЪРНО И СОФТУЕРНО ИНЖЕНЕРСТВО

Изследване на операциите и приложно програмиране

Курсов проект

„ Реализация на динамично програмиране – задача за натоварване“

Съставил: Ангел Любомиров Стойнов

Факултетен номер: 121222150

Група: 40

Съдържание

Постановка на задачата

Общ капацитет на кораб – Q тона.

За всеки контейнер са известни теглото q_i и стойността c_i**.**

Трябва да се определи по колко броя да се вземат от всеки тип контейнер, така че да не се надвиши общата товароподемност на кораба, а стойността на натоварената стока да е максимална.

Нека общият капацитет на кораба е Q = 10, контейнерите са от 4 различни типа, а теглото и стойността на всеки контейнер са зададени на таб. 1:

Контейнер	Тегло q_i	Стойност c_i
1	2	4
2	8	10
3	3	6
4	4	8

(Таблица 1 – показва примерни стойности) [1]

Дефиниране на задачата като задача от динамичното програмиране

Управлението на стъпка i ще е количеството предмети x_i, които да се вземат на тази стъпка. Задачата има същата постановка, както задачата за разпределение на ресурсите, само функциите $f_i(x)$ се определят от стойността на $c_i x_i$;

Управляемата система S е количеството останал капацитет до запълване на кораба. Рекурентната зависимост е $W_i(S, x_i) = c_i x_i + W_{i+1}(S - q_i x_i)$.

Условната оптимална печалба на всяка стъпка е:

$W_i(S) = \max \{c_i x + W_{i+1}(S - q_i x)\}$,

а условното оптимално управление $x_i(S)$ е тази стойност на x, за която е достигната

максималната стойност на $W_i(S)$ [1].

Реализация

За реализация на задачата е използван програмния език **java** [2] и **apache** библиотека за логване на резултата - **org.apache.logging.log4j** [3].

Етапи на реализация:

1. Изчисляване на оптималната стойност, която може да бъде постигната при товароспособност S чрез Bellman динамично програмиране [4]. Използвана е формулата:
    a. $W_i(S) = max \{c_i x + W_{i+1}(S - q_i x)\}$
2. След намирането на максималната стойност се използва DFS (обхождане в дълбочина) [5], за да се намери всички комбинации на броя контейнери от всеки тип, които дават тази стойност.

```
public static void solve() {  1 usage  Angel Stoynov +1
    int[][] optimalEarningPerCapacity = computeBellmanTables();

    log.info("All optimal solutions:");
    int optimalValue = optimalEarningPerCapacity[1][SHIP_CAPACITY];
    List<Map<Container, Integer>> solutions = findAllOptimalSolutions(optimalValue);

    for (Map<Container, Integer> solution : solutions) {
        printSolution(solution);
    }

    log.info("Number of solutions = {}", solutions.size());
}
```

(Фигура 1, показва публичния метод solve(), който се извиква при изпълнение на програмата.)

Методът solve() (фиг. 1) е публичният основен метод, който допълнително извиква помощни методи - computeBellmanTables(), findAllOptionalSolutions(), printSolution().

```
int[][] optimalEarningPerCapacity = computeBellmanTables();
```

Двумерният масив `optimalEarningPerCapacity` съдържа резултата от помощния метод, математически изразено е $W_i(S)$.

`optimalValue` определя максималната стойност $W_1(Q)$, където Q е товароспособността, репрезентирана от SHIP_CAPACITY = 10.

`Solutions` е списък съдържащ колекция, с ключ даден контейнер и резултат цяло число. Тук се извиква DFS, за да се намерят всички оптимални решения.

```
int optimalValue = optimalEarningPerCapacity[1][SHIP_CAPACITY];
List<Map<Container, Integer>> solutions = findAllOptimalSolutions(optimalValue);
```

Обхождаме всяко решение и принтиране в конзолата.

```
for (Map<Container, Integer> solution : solutions) {
    printSolution(solution);
}
```

```java
private static int[][] computeBellmanTables() {  1 usage  Angel Stoynov +1
    int containerSize = containers.size();
    int[][] optimalEarningPerCapacity = new int[containerSize + 2][SHIP_CAPACITY + 1];

    for (int i = 0; i <= SHIP_CAPACITY; i++) {
        optimalEarningPerCapacity[containerSize + 1][i] = 0;
    }

    for (int i = containerSize; i >= 1; i--) {
        int qi = containers.get(i - 1).weight();
        int ci = containers.get(i - 1).value();

        for (int s = 0; s <= SHIP_CAPACITY; s++) {
            int bestValue = 0;
            int maxCount = s / qi;

            for (int x = 0; x <= maxCount; x++) {
                int newValue = ci * x + optimalEarningPerCapacity[i + 1][s - qi * x];

                if (newValue > bestValue) {
                    bestValue = newValue;
                }
            }

            optimalEarningPerCapacity[i][s] = bestValue;
        }

        log.info("W{}(S): {}", i, Arrays.toString(optimalEarningPerCapacity[i]));
    }

    return optimalEarningPerCapacity;
}
```

(Фигура 2, метод имплементиращ алгоритъм на Белман и връща резултата в табличен вид.)

Двумерният масив `optimalEarningPerCapacity` - ***$W_i(S)$***.

Последният ред $W_{n+1}(S)$ се запълва с 0, което е базовият случай, защото в алгоритъма на Белман винаги има един допълнителен ред.

Обратно обхождане, от i = n към 1:

- За всеки тип контейнер се извлича тегло q_i и стойност c_i.
- За всяка товароспособност S се проверява колко броя от текущия контейнер могат да се вземат.
- Определяме максималния възможен брой x: `int maxCount = s / qi`
- За всяко възможно количество x се изчислява: $c_i x + W_{i+1}(S - q_i x)$
    - $c_i x$ – стойността на избраните контейнери от тип *i*.
    - $W_{i+1}(S - q_i x)$ - най-добрата стойност за останалите типове.
- Избира се максималната стойност:
    - `if (newValue > bestValue) bestValue = newValue,` математически изразено - ***$W_i(S) = max(…)$***

```java
    private static List<Map<Container, Integer>> findAllOptimalSolutions(int optimalValue) {   1 usage   Angel Stoynov
        List<Map<Container, Integer>> results = new ArrayList<>();
        Map<Container, Integer> selection = new LinkedHashMap<>();

        containers.forEach( Container container -> selection.put(container, 0));
        dfs( index: 0, SHIP_CAPACITY, optimalValue, selection, results);

        return results;
    }

```

(Фигура 3, методът намира всички оптимални решения за броя контейнери от всеки тип, които водят до максималната стойност, предварително изчислена от Белман.)

Всеки контейнер получава първоначална стойност 0:

- `{x1 = 0, x2 = 0, x3 = 0, x4 = 0}`

Редът им е запазен чрез `LinkedHashMap`. Извиква се DFS методът, който ще обходи всички възможни комбинации контейнери, но ще добави само тези, които водят до оптимална стойност.

```java
    private static void dfs(int index, int remainingWeight, int remainingValue, Map<Container, Integer> selection, List<Map<Container, Integer>> results) {
        if (remainingWeight < 0 || remainingValue < 0) {
            return;
        }

        if (remainingWeight == 0 && remainingValue == 0) {
            results.add(new LinkedHashMap<>(selection));
            return;
        }

        if (index >= containers.size()) {
            return;
        }

        Container container = containers.get(index);
        int maxCount = remainingWeight / container.weight();

        for (int count = 0; count <= maxCount; count++) {
            selection.put(container, count);

            int updatedValue = remainingValue - count * container.value();
            int updatedWeight = remainingWeight - count * container.weight();

            dfs( index: index + 1, updatedWeight, updatedValue, selection, results);
        }

        selection.put(container, 0);
    }
```

(Фигура 4, методът dfs() търси всички възможни комбинации от контейнери за всеки тип.)

Методът е рекурсивен, следователно има няколко базови случаи, които трябва да бъдат спазени:

- Оставащото тегло и стойност да не е по-малко или равно на 0.
- Типовете не трябват да бъдат повече от количеството контейнери.
- Ако теглото и количеството е равно на 0, тогава този резултат е оптимален и се записва.

```java
Container container = containers.get(index);

int maxCount = remainingWeight / container.weight();
```

Взима текущия контейнер и изчисляваме максималното възможно количество.

```
for (int count = 0; count <= maxCount; count++) {
    selection.put(container, count);

    int updatedValue = remainingValue - count * container.value();
    int updatedWeight = remainingWeight - count * container.weight();

    dfs(index + 1, updatedWeight, updatedValue, selection, results);
}

selection.put(container, 0);
```

Изчислява се новата стойност и новото тегло спрямо броя и вида контейнер. После методът рекурсивно се извиква, като индекса се инкрементира с 1 и се подават обновените стойности, и тегло. След края на цикъла се връща назад, след като са пробвани всички count стойности за този контейнер, трябва да се зададе на 0, за да не повлияе на други клонове на DFS дървото.

```
private static void printSolution(Map<Container, Integer> solution) {  1 usage   Angel Stoynov +1
    StringBuilder sb = new StringBuilder();
    int index = 1;

    for (Map.Entry<Container, Integer> entry : solution.entrySet()) {
        sb.append("x").append(index).append("* = ").append(entry.getValue()).append(" ");
        index++;
    }

    log.info(sb.toString());
}
```

(Фигура 5, метод за правилна визуализация на получените стойности)

```
[INFO] 16:09:53 ContainerResolver - W4(S): [0, 0, 0, 0, 8, 8, 8, 8, 16, 16, 16]
[INFO] 16:09:53 ContainerResolver - W3(S): [0, 0, 0, 6, 8, 8, 12, 14, 16, 18, 20]
[INFO] 16:09:53 ContainerResolver - W2(S): [0, 0, 0, 6, 8, 8, 12, 14, 16, 18, 20]
[INFO] 16:09:53 ContainerResolver - W1(S): [0, 0, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[INFO] 16:09:53 ContainerResolver - All optimal solutions:
[INFO] 16:11:47 ContainerResolver - x1* = 0 x2* = 0 x3* = 2 x4* = 1
[INFO] 16:11:47 ContainerResolver - x1* = 1 x2* = 0 x3* = 0 x4* = 2
[INFO] 16:11:47 ContainerResolver - x1* = 2 x2* = 0 x3* = 2 x4* = 0
[INFO] 16:11:47 ContainerResolver - x1* = 3 x2* = 0 x3* = 0 x4* = 1
[INFO] 16:11:47 ContainerResolver - x1* = 5 x2* = 0 x3* = 0 x4* = 0
[INFO] 16:11:47 ContainerResolver - Number of solutions = 5
```

(Фигура 6, получени резултати)

За $\boldsymbol{W_1(S)}$ е напълно достатъчно да се покаже само 20, но това не променя по никакъв начин крайния резултат. Отдолу са изброени решенията, както и техният общ брой.

```
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 20, UpdatedWeight: 10
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 20, UpdatedWeight: 10
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 20, UpdatedWeight: 10
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 20, UpdatedWeight: 10
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 12, UpdatedWeight: 6
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 4, UpdatedWeight: 2
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 14, UpdatedWeight: 7
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 14, UpdatedWeight: 7
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 6, UpdatedWeight: 3
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 8, UpdatedWeight: 4
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 8, UpdatedWeight: 4
[DEBUG] 17:58:17 ContainerResolver - UpdatedValue: 0, UpdatedWeight: 0
```

```
[DEBUG] 17:59:24 ContainerResolver - Add new solution:
{Container[weight=2, value=4]=0, Container[weight=8, value=10]=0, Container[weight=3, value=6]=2, Container[weight=4, value=8]=1}
```

(Фигура 7, дебъг логове)

Поради множеството рекурсии, които се изпълняват, дебъг логовете изглеждат по този начин (фиг. 7), за това не са добавени в крайния резултат. Фигура 7, показва обновеното тегло и стойност за постигането само на едно оптимално решение. Спрямо данните от таблица 1, има още четири решения, което ще изпълни терминала с огромно количество логове.

Цитирани източници

[1] Д. Г. Д. Б. С. Н. Румен Трифонов, Ръководство по изследване на операциите и приложно програмиране, София: Авангард Прима, 2013.

[2] „Java,“ Oracle, [Онлайн]. Available: https://www.java.com/en/download/help/whatis_java.html.

[3] apache, „Apache log4j,“ apache, 29 07 2012. [Онлайн]. Available: https://logging.apache.org/log4j/2.x/index.html.

[4] B. R., Dynamic Programming, Princeton : Princeton University Press, 1957.

[5] R. W. K. Sedgewick, Algorithms, Addison-Wesley, 4th Edition..