Московский авиационный институт
(национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студент:	О. А. Мезенин
Преподаватель:	А. А. Кухтичев
Группа:	М8О-306Б-21
Дата:	
Оценка:	
Подпись:	

Москва, 2023

Лабораторная работа №7

Задача: У вас есть рюкзак, вместимостью m, а так же n предметов, у каждого из которых есть вес w_i и стоимость c_i. Необходимо выбрать такое подмножество I из них, чтобы:

- $\sum_{i \in I} m_i \leq m$
- $(\sum_{i \in I} c_i) * |I|$ является максимальной из всех возможных.

$|I|$ – мощность множества I.

1 Описание

Воспользуемся методом динамического программирования. Пусть $dp[i][j][k]$ – максимальная стоимость (стоимость предметов, умноженная на количество предметов) в рюкзаке вместимости k, если можно взять j предметов из первых i. Тогда рекуррентное соотношение будет следующим: $dp[i][j][k] = max(dp[i-1][j][k], (dp[i-1][j-1][k-w[i-1]]/(j-1)+c[i-1])*j)$. Т.е. в дополнение классической задачи о рюкзаке [2] алгоритм будет ещё перебирать количество предметов j.
Такой алгоритм будет иметь временную и пространственную сложность $O(n^2 * m)$.

2 Исходный код

Функция `main` считывает входные данные, вызывает функцию `knapsack` и выводит ответ.

```
int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    size_t n, m;
    std::cin >> n >> m;
    std::vector<uint32_t> w(n), c(n);
    for (size_t i = 0; i < n; ++i) {
        std::cin >> w[i] >> c[i];
    }
    std::vector<int> path;
    uint32_t res = knapsack(n, m, w, c, path);
    std::cout << res << '\n';
    for (int i = path.size()-1; i >= 0; --i) {
        std::cout << path[i] << ' ';
    }
    std::cout << '\n';
    return 0;
}
```

Основной алгоритм описан в функции `knapsack`. Сначала алгоритм выполняется для $j = 1$ (т.е. берём максимум 1 предмет), затем для $j > 1$. Такое разделение сделано из-за деления на ноль, которое появляется при $j = 1$ в рекуррентной формуле. После этого мы ищем максимум по j при $i = n$, $k = m$. Затем ищем путь, начиная с максимума: если $dp[i][j][k] == dp[i-1][j][k]$, значит, мы не брали предмет с индексом $i-1$, иначе необходимо уменьшить k на вес предмета с индексом $i-1$, уменьшить значение j на единицу и добавить предмет с индексом $i-1$ в ответ. Поиск пути закончится, когда значение $dp[i][j][k]$ станет равно нулю.

```
uint32_t knapsack(size_t n, size_t m, std::vector<uint32_t>& w, std::vector<uint32_t>&
     c, std::vector<int>& answerPath) {
    std::vector<std::vector<std::vector<uint32_t>>> dp(n+1, std::vector<std::vector<
        uint32_t>>(n+1, std::vector<uint32_t>(m+1, 0)));
    for (size_t i = 1; i <= n; ++i) {
        for (size_t k = 1; k <= m; ++k) {
            dp[i][1][k] = dp[i-1][1][k];
            if (w[i-1] <= k) {
                uint32_t tmp = dp[i-1][1][k-w[i-1]] + c[i-1];
                if (tmp > dp[i][1][k]) {
                    dp[i][1][k] = tmp;
                }
            }
        }
    }
    for (size_t i = 2; i <= n; ++i) {
```

```
        for (size_t j = 2; j <= i; ++j) {
            for (size_t k = 1; k <= m; ++k) {
                dp[i][j][k] = dp[i-1][j][k];
                if (w[i-1] <= k && dp[i-1][j-1][k-w[i-1]] > 0) {
                    uint32_t tmp = (dp[i-1][j-1][k-w[i-1]] / (j-1) + c[i-1]) * j;
                    if (tmp > dp[i][j][k]) {
                        dp[i][j][k] = tmp;
                    }
                }
            }
        }
    }
    size_t i = n, j = 0, k = m;
    size_t maxValue = 0;
    for (size_t jj = 0; jj <= n; ++jj) {
        if (dp[i][jj][k] > maxValue) {
            j = jj;
            maxValue = dp[i][jj][k];
        }
    }
    while (dp[i][j][k] > 0) {
        if (dp[i][j][k] == dp[i-1][j][k]) {
            --i;
        } else {
            k -= w[i-1];
            --j;
            answerPath.push_back(i--);
        }
    }
    return maxValue;
}
```

3 Консоль

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ make lab7
g++ -std=c++2a -pedantic -Wall -Wextra -Werror main.cpp knapsack.cpp -o lab7.out
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ ./lab7.out
3 6
2 1
5 4
4 2
6
1 3
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ ./lab7.out
5 65
23 22
11 17
25 91
14 18
33 97
396
2 4 5
```

4 Тест производительности

Тесты производительности представляют из себя следующее: алгоритм, основанный на методе динамического программирования, будет сравниваться с алгоритмом полного перебора, т.е. он будет перебирать все подмножества набора из n предметов. Будет три теста: на 10 и 20 предметов с вместимостью рюкзака $m = 100$ и на 10 предметов с вместимостью рюкзака $m = 1000000$.

```
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ ./benchmark.out <test_bm/01.t
DP time: 178us
Primitive enumeration time: 1729us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ ./benchmark.out <test_bm/02.t
DP time: 637us
Primitive enumeration time: 2166362us
aprold@SAI:~/Documents/GitHub/MAI-DA/lab7$ ./benchmark.out <test_bm/03.t
DP time: 105867us
Primitive enumeration time: 2263us
```

Наш алгоритм работает за $O(n^2 * m)$, алгоритм полного перебора – за $O(2^n)$. В общем случае очевидно, что алгоритм, основанный на ДП, работает гораздо быстрее полного перебора. В последнем тесте алгоритм полного перебора побеждает, потому что время его работы не зависит от m.

5 Выводы

Выполнив седьмую лабораторную работу по курсу «Дискретный анализ», узнал про метод динамического программирования и о задачах, где он применяется.
Если говорить о задаче «0-1 рюкзак», то в общем случае метод динамического программирования работает гораздо лучше, чем полный перебор, но всегда можно найти тесты, где полный перебор будет быстрее.

Список литературы

[1] *Задача о рюкзаке*
URL: https://academy.yandex.ru/handbook/algorithms/article/zadacha-o-ryukzake (дата обращения: 24.10.2023).

[2] *Задача о рюкзаке*
URL: https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_рюкзаке (дата обращения: 24.10.2023).