Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Дискретный анализ»

Студент:	Д. С. Пивницкий
Преподаватель:	А. А. Кухтичев
Группа:	М8О-206Б-19
Дата:	01.01.2021
Оценка:	
Подпись:	

Москва, 2021

Лабораторная работа №7

Задача: При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом; оценить время выполнения алгоритма и объем затрачиваемой оперативной памяти. Перед выполнением задания необходимо обосновать применимость метода динамического программирования.

Разработать программу на языке C или C++, реализующую построенный алгоритм.

Формат входных данных: В первой строке заданы $1 \leq n \leq 1001 \leq m \leq 5000$. В последующих n строках через пробел заданы параметры предеметов: w_i и c_i.

Вариант: 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: $\sum_{i \in I} w_i \leq m$ и $\sum_{i \in I}(c_i) \times |I|$ является максимальной из всех возможных. $|I|$ - мощность множества I.

1 Описание

Как описано в [1], динамическое программирование - это метод решения задач, при котором сложная задача разбивается на более простые, решение сложной задачи составляется из решений простых задач.

Этот метод очень похож на «разделяй и властвуй», но динамическое программирование допускает использование метода восходящего анализа, который позволяет изначально решать простые задачи и получать на их базе решение более сложных.

Так же метод запоминает решения подзадач, потому что часто для построения нужно обращаться за оптимальным решением к одним и тем же малым задачам.

Задача о рюкзаке является известной NP-полной задачей, которая при некоторых ограничениях решается за полиномиальное время с помощью метода динамического программирования.

Стадартный вариант задачи описан и доказан в [2]. Для моего варианта задания dp_i, j, k - максимальная стоимость j вещей из первых i, таких, что их суммарный вес не превышает k. То есть алгоритм будет перебирать количество предметов, которые будут в рюкзаке.

Пусть существует оптимальное решение в $dp_i, j, k - w_j - 1$, тогда $dp_i + 1, j + 1, k = max(dp_i, j, k - wj - 1 + c_j + 1, dpi + 1, j, k)$. В рекуррентной формуле рассматривается два варианта: взять вещь $j + 1$ или нет.

Такое решение имеет $n^2 \times m$ состояния, в каждое можно перейти из двух других. Так временная сложность алгоритма $O(n^2 \times m)$.

Хранение всей таблицы состояний слишком дорого по памяти, но необходимо для восстановления ответа. Поэтому будем хранить только dp_i и $dp_i + 1$ и битовые множества предметов, которые оптимальны для решения подзадачи. Пространственная сложность такого подхода $O(n \times m)$.

2 Исходный код

Опишем матрицы $dpPrev$ и $dpCur$ для $dp_j + 1$ и dp_j, матрицы $setCur$ и $setPrev$ для хранения множества предметов. Для достижения пространственной сложности $O(n \times m)$ будем использовать эффективный по памяти $std :: bitset$.

Код: main.cpp

```
#include <bitset>
#include <iostream>
#include <vector>

const size_t MAX_N = 100;

int main() {
int n, m;
std::cin >> n >> m;
std::vector<int> w(n);
std::vector<long long> c(n);
for (int i = 0; i < n; ++i) {
std::cin >> w[i] >> c[i];
}

std::vector< std::vector< long long > > dpPrev(n + 1, std::vector<long long>(m + 1));
std::vector< std::vector< long long > > dpCur(n + 1, std::vector<long long>(m + 1));
std::vector< std::vector< std::bitset<MAX_N> > > setPrev(n + 1, std::vector< std::bitset<MAX_N> >(m + 1));
std::vector< std::vector< std::bitset<MAX_N> > > setCur(n + 1, std::vector< std::bitset<MAX_N> >(m + 1));
long long ans = 0;
std::bitset<MAX_N> res;

for (int j = 1; j < n + 1; ++j) {
for (int k = 1; k < m + 1; ++k) {
dpPrev[j][k] = dpPrev[j - 1][k];
setPrev[j][k] = setPrev[j - 1][k];
if (c[j - 1] > dpPrev[j][k] and k - w[j - 1] == 0) {
dpPrev[j][k] = c[j - 1];
setPrev[j][k] = 0;
setPrev[j][k][j - 1] = 1;
}
if (dpPrev[j][k] > ans) {
ans = dpPrev[j][k];
res = setPrev[j][k];
}
}
}

for (long long i = 2; i < n + 1; ++i) {
for (int j = 1; j < n + 1; ++j) {
```

```
for (int k = 1; k < m + 1; ++k) {
dpCur[j][k] = dpCur[j - 1][k];
setCur[j][k] = setCur[j - 1][k];
if (k - w[j - 1] > 0 and dpPrev[j - 1][k - w[j - 1]] > 0) {
if (i * (c[j - 1] + dpPrev[j - 1][k - w[j - 1]] / (i - 1)) > dpCur[j][k]) {
dpCur[j][k] = i * (c[j - 1] + dpPrev[j - 1][k - w[j - 1]] / (i - 1));
setCur[j][k] = setPrev[j - 1][k - w[j - 1]];
setCur[j][k][j - 1] = 1;
}
}
if (dpCur[j][k] > ans) {
ans = dpCur[j][k];
res = setCur[j][k];
}
}
}
std::swap(dpCur, dpPrev);
std::swap(setCur, setPrev);
}
std::cout << ans << '\n';
for (int i = 0; i < n; ++i) {
if (res[i]) {
std::cout << i + 1 << ' ';
}
}
std::cout << '\n';
}
```

3 Консоль

```
(py37) ~ /DA_labs/lab7$ make
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
(py37) ~ /DA_labs/lab7$ cat tests/1.in
3 6
2 1
5 4
4 2
(py37) ~ /DA_labs/lab7$ ./solution <tests/1.in
6
1 3
(py37) ~ /DA_labs/lab7$ cat tests/2.in
14 41
2 60
6 25
10 56
8 4
7 81
4 40
10 56
7 2
8 32
2 25
6 22
9 5
9 95
8 6
(py37) ~ /DA_labs/lab7$ ./solution <tests/2.in
2674
1 2 3 5 6 10 13
```

4 Тест производительности

Сравним реализованный алгоритм с приближённым алгоритмом, который не всегда даёт верный ответ. Тесты состоят из 10, 50 и 100 вещей.

Моя реализация:

```
(py37) ~ /DA_labs/lab7$ make
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
(py37) ~ /DA_labs/lab7$ make bench
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror benchmark.cpp -o benchmark
(py37) ~ /DA_labs/lab7$ ./benchmark <tests/1.in
Sort 0.74 ms
(py37) ~ /DA_labs/lab7$ ./solution <tests/1.in
DP 0.388 ms
(py37) ~ /DA_labs/lab7$ ./benchmark <tests/2.in
Sort 0.119 ms
(py37) ~ /DA_labs/lab7$ ./solution <tests/2.in
DP 1.584 ms
(py37) ~ /DA_labs/lab7$ ./benchmark <tests/3.in
Sort 0.204 ms
(py37) ~ /DA_labs/lab7$ ./solution <tests/3.in
DP 125.560 ms
```

Видно, что приближённый алгоритм гораздо быстрее динамического программирования, потому что он сортирует предметы по уменьшению веса и возрастанию цены, а количество предметов мало.

5 Выводы

В ходе выполнения лабораторной работы я изучил классические задачи динамического программирования и их методы решения, реализовал алгоритм для своего варианта задания.

Также я познакомился с $std :: bitset$ для уменьшении потребляемой программой памяти, узнал, что $std :: vector$ имеет спецификацию $std :: vector < bool >$, которая тоже эффективна по памяти, как и $std :: bitset$.

Динамическое программирование позволяет разработать точные и относительно быстрые алгоритмы для решения сложных задач, в то время, как переборное решение слишком медленное, а жадный алгоритм не всегда даёт правильный результат.

Например, известную NP-полную задачу о коммивояжере [3] можно решить с помощью динамического программирования по подмножествам за $O(n^2 \times 2^n)$, что гораздо быстрее перебора за $O(n!)$.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Поразрядная сортировка — Вики университета ITMO.* URL: `https://neerc.ifmo.ru/wiki/index.php?title=Цифровая_сортировка` (дата обращения: 01.10.2020).