北京邮电大学

计算机学院

动态规划算法设计与分析实验报告

姓名: 吴清柳

学号: 2020211597

班级: 2020211323

指导老师: 邵蓥侠

课程名称: 算法设计与分析

November 12, 2022

Contents

1 实验题目

1.1 0-1 背包问题

1.1.1 题目描述

有 N 件物品和一个容量为 V 的背包. 第 i 件物品的大小是 c_i, 价值是 w_i. 求解将哪些物品装入背包可以让这些物品的大小总和不超过背包容量, 且价值总和最大.

1.1.2 输入格式

输入文件名为 bag.in, 输入共两行.

- 第一行有 2 个整数 V 和 N, 用一个空格隔开, V 代表背包容量, N 代表物品数目.
- 接下来的 N 行每行包括两个整数, 分别表示某件物品的大小和其价值.

1.1.3 输出格式

输出文件名为 bag.out, 输出共一行. 第一行包含 1 个整数表示最大价值.

1.1.4 输入输出样例

bag.in	bag.out
70 3 71 100 69 1 1 2	3

1.1.5 数据范围

$0 < N \leq 2500$, $|V| \leq 2500$, $0 < c_i \leq 2500$, $0 < w_i \leq 100000$.

1.1.6 解法

分别使用基本的动态规划方法求解和应用跳跃点优化的动态规划方法求解.

2 实验过程

2.1 动态规划求解 01 背包问题

2.1.1 算法类别

此处介绍的 01 背包问题解法是一种动态规划方法.

2.1.2 算法思路

2.1.2.1 0-1 背包问题的子问题描述 设所给 01 背包问题的子问题为

$$\max \sum_{k=i}^{n} v_k x_k \tag{1}$$

其限制条件为

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \leq j \\ x_k \in \{0,1\}, i \leq k \leq n \end{cases} \tag{2}$$

2.1.2.2 0-1 背包问题的最优子结构性质 容易证明 0-1 背包问题具有最优子结构性质: 设 0-1 背包问题的子问题1的最优值为 $m(i,j)$, 即 $m(i,j)$ 是背包容量为 j, 可选择物品为 $i,i+1,\ldots,n$ 的时候的最优值.

设 $(y_1,y_2,\ldots,y_n)$ 是 $x_i\tilde{x}_n$ 的一个最优解, 则可以推断, $(y_2,y_3,\ldots,y_n)$ 是一个子问题4的最优解.

$$\max \sum_{i=2}^{n} v_i y_y \tag{3}$$

$$\begin{cases} \sum_{i=1}^{n} w_i y_i \leq C - w_1 y_1 \\ y_i \in \{0,1\}, 1 \leq i \leq n \end{cases} \tag{4}$$

否则, 假设 $(y_2,y_3,\ldots,y_n)$ 不是子问题的最优解, 则存在一组 $(y'_2,y'_3,\ldots,y'_n)$ 是子问题式4的最优解, 则 $(y_1,y'_2,y'_3,\ldots,y'_n)$ 是原问题的最优解, 矛盾. 因此 $(y_2,y_3,\ldots,y_n)$ 是原问题的最优解. 0-1 背包问题的最优子结构得证.

据此, 可得 0-1 背包问题的计算 $m(i,j)$ 的递归式 (状态转移方程) 如下:

$$m(i,j) = \begin{cases} max\{m(i+1,j), m(i+1,j-w_i)+v_i\}, & j \geq w_i \\ m(i+1,j), & 0 \leq j < w_i \end{cases} \tag{5}$$

$$m(n,j) = \begin{cases} v_n, & j \geq w_n \\ 0, & 0 \leq j < w_n \end{cases} \tag{6}$$

2.1.3 关键函数及代码段的描述

采用动态规划的方式实现 0-1 背包问题.

其核心代码实现如下:

```
void BackPack01::backPackDP() {
    maxVal = -1;

    // Initialize
    for (int j = 0; j <= volume; j++) {
        dp[n - 1][j] = (j >= w[n - 1]) ? val[n - 1] : 0;
    }

    int i = n - 2, j = volume;
    for (; i >= 0; i--) {
        j = volume;
        for (; j >= 0; j--) {

            // Compare total value of items in the backpack between put
            // item[i] in and not put it in.
            dp[i][j] = (j >= w[i]) ? std::max(dp[i + 1][j],
                                              dp[i + 1][j - w[i]] + val[i])
                                   : dp[i + 1][j];
        }
    }

    maxVal = dp[0][volume];
}
```

2.1.4 算法时间及空间复杂性分析

2.1.4.1 空间复杂度分析 0-1 背包问题的动态规划实现需要一个大小为 N*C 的数组进行辅助, 所以空间复杂度为 $O(NC)$.

2.1.4.2 时间复杂度分析 由于动态规划的本质是上述递推式4, 且实现的核心代码使用了两层循环, 所以可以很方便的计算出时间复杂度为

$$C(n) = O(N) \times O(C) = O(NC) \tag{7}$$

综上, 0-1 背包问题的一般动态规划解法的时间复杂度为 O(NC).

2.2 应用跳跃点方法优化的 0-1 背包问题动态规划解法

2.2.1 算法类别

应用了跳跃点方法优化的 0-1 背包问题解法也是一种动态规划方法.

2.2.2 算法思路

2.2.2.1 概述 由 $m(i,j)$ 的递归式容易证明, 在一般的情况下, 对每一个确定的 $i(1 \le i \le n)$, 函数 $m(i,j)$ 是关于变量 j 的阶梯状单调不减函数. 跳跃点是这一类函数的描述特征. 在一般情况下, 函数 $m(i,j)$ 由其全部跳跃点唯一确定. 如图所示.

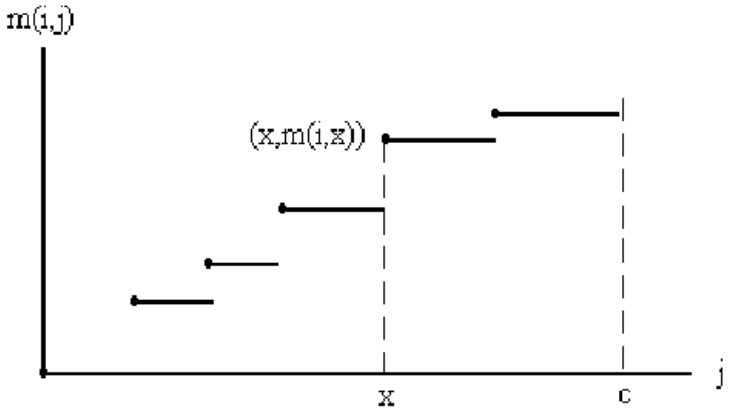

Figure 1: 跳跃点

对每一个确定的 $i,(1 \le i \le n)$, 用一个链表 p[i] 存储函数 $m(i,j)$ 的全部跳跃点. 链表 p[i] 可以计算 $m(i,j)$ 的递归式递归地由表 p[i+1] 计算, 初始的时候 p[n+1]={(0,0)}.

2.2.2.2 算法改进的描述 函数 $m(i,j)$ 是由函数 $m(i+1,j)$ 与函数 $m(i+1,j-w_i)+v_i$ 作 max 运算得到的. 因此, 函数 $m(i,j)$ 的全部跳跃点包含于函数 $m(i+1,j)$ 的跳跃点集 p[i+1] 与函数 $m(i+1,j-w_i)+v_i$ 的跳跃点集 q[i+1] 的并集中.

易知, $(s,t) \in q[i+1]$ 当且仅当 $w_i \le s \le c$ 且 $(s-w_i,t-vi) \in p[i+1]$.

因此, 容易由 p[i+1] 确定跳跃点集 q[i+1] 如下:

$$q[i+1] = p[i+1] \oplus (w_i,v_i) = \{(j+w_i,m(i,j)+v_i) | (j,m(i,j)) \in p[i+1]\} \quad (8)$$

另一方面, 设 (a,b) 和 (c,d) 是 $p[i+1] \cup q[i+1]$ 中的 2 个跳跃点, 则当 $c \ge a$ 且 $d < b$ 的时候, (c,d) 受控于 (a, b), 从而 (c, d) 不是 p[i] 中的跳跃点. 除了受控跳跃点之外, $p[i+1] \cup q[i+1]$ 中的其他跳跃点都是 p[i] 中的跳跃点.

由此可见, 在递归地由表 p[i+1] 计算表 p[i] 的时候, 可以先由 p[i+1] 计算出 q[i+1], 然后合并表 p[i+1] 和表 q[i+1], 并清除其中的受控跳跃点得到表 p[i].

2.2.3 关键函数及代码段的描述

使用跳跃点优化的动态规划方法求解 0-1 背包问题的关键在于根据上述推导过程动态根据上一状态的跳跃点集维护当前状态的跳跃点. 具体代码如下:

对函数均使用类似 C++ 的 Pseudo-Code 描述.

```
void BackPackJumpPoint::JumpPointBackPackDP() {
    int *head = new int[n + 2]; // Track jump point start position.
```

```
    head[n] = 0;
    jp[0][0] = 0; // Store item weight
    jp[0][1] = 0; // Store item value

    // Left points to first jump point of p[i+1], right points to last jump
    // point of p[i+1]. Next is position where next jump point will store.
    int left = 0, right = 0, next = 1;
    head[n - 1] = 1; // Points to the position of first jump point of
        item[n-1].

    for (int i = n - 1; i >= 0; i--) {
        int k = left; // k points to jump points of p[], move k to
                      // evaluate controlled points in p[] and
                      // p[]+(w,v)
        for (int j = left; j <= right; j++) {

            if (jp[j][0] + w[i] > volume) {
                // No enough backpack space to fit item[i] in, exit loop.
                break;
            }

            // Compute new jump point as jp[]+(w,v).
            int x = jp[j][0] + w[i];
            int y = jp[j][1] + val[i];

            // If jp[k][0] < x, then it must be a jump point of current
            // item.
            while (k <= right && jp[k][0] < x) {
                jp[next][0] = jp[k][0];
                jp[next++][1] = jp[k++][1];
            }

            // Clear controlled jump point.
            if (k <= right && jp[k][0] == x) {
                if (y < jp[k][1]) {
                    y = jp[k][1];
                }
                k++;
            }

            if (y > jp[next - 1][1]) {
                jp[next][0] = x;
                jp[next++][1] = y;
            }

            while (k <= right && jp[k][1] <= jp[next - 1][1]) {
                k++;
            }
        }

        // Add remaining jump points.
        while (k <= right) {
            jp[next][0] = jp[k][0];
            jp[next++][1] = jp[k++][1];
        }

        left = right + 1;
        right = next - 1;
```

```
        head[i - 1] = next;
    }


    maxVal = jp[next - 1][1];
}
```

2.2.4 算法时间及空间复杂性分析

2.2.4.1 空间复杂度分析 相比于一般的动态规划解 0-1 背包算法, 应用跳跃点法的动态规划解 0-1 背包算法额外使用数组存储了跳跃点, 这一部分是空间的主要支出, 从跳跃点集 p[i] 的定义可以看出, p[i] 中的跳跃点相应于 $x_i,\ldots,x_n$ 的 0/1 赋值, 因此, p[i] 中的跳跃点个数不超过 2^{n-i+1}. 又由于受控跳跃点会被清除, 因此, 跳跃点集最多使用 $O(min\{N\times 2^N,c\})$ 的额外空间.

由于跳跃点表 p[i] 的计算只依赖于 p[i+1] 的数据, 所以 p[i+2] 及之后的跳跃点表不需要保存, 这样就将跳跃点表的大小从 $O(min\{N\times 2^N,c\})$ 优化到了 $O(min\{2^N,c\})$.

2.2.4.2 时间复杂度分析 跳跃点算法的主要计算量在于计算跳跃点集 $p[i](1\le i\le n)$. 由于 $q[i+1]=p[i+1]\oplus(w_i,v_i)$, 故计算 q[i+1] 需要 $O(|p[i+1]|)$ 的计算时间. 合并 p[i+1] 和 q[i+1] 并清除受控跳跃点也需要 $q[i+1]=p[i+1]\oplus(w_i,v_i)$ 的计算时间. 从跳跃点集 p[i] 的定义可以看出, p[i] 中的跳跃点相应于 $x_i,\ldots,x_n$ 的 0/1 赋值. 因此, p[i] 中的跳跃点个数不超过 2^{n-i+1}. 由此可见, 算法计算跳跃点集 p[i] 所花费的计算时间为

$$O(\sum_{i=2}^{n}|p[i+1]|)=O(\sum_{i=2}^{n}2^{n-i})=O(2^n) \tag{9}$$

从而, 改进后算法的计算时间复杂性为 $O(2^n)$. 当所给物品的重量 $w_i(1\le i\le n)$ 是整数的时候, $|p[i]|\le c+1,(1\le i\le n)$. 在这种情况下, 改进后的算法的计算时间复杂度为 $O(min\{nc,2^n\})$.

3 实验结果

对一般方法和应用跳跃点优化的动态规划解 0-1 背包问题的程序执行环境和运行方式均相同, 包括其测试结果都在此一并介绍.

3.1 程序执行环境

在以下环境下进行过测试.

操作系统 macOS 13.1 22C5044e arm64;

编译器 clang-1400.0.29.102;

构建和测试工具 GNU Make 3.81;

文档编译工具 XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022)

文本编辑器 nvim v0.8.0

3.2 程序运行方式

3.2.1 排序程序的编译, 运行和测试方法

一般方法和应用跳跃点优化的动态规划解 0-1 背包问题都采用 make 进行统一构建和测试, 并使用 diff 命令将结果进行比较, 以确保排序结果正确.

程序 (及文档) 均使用 make 进行构建, 对于排序程序的构建, 在 shell 中执行以下命令:

```
make
```

二进制文件 BackPack01, JumpPointBackPack 将被编译在 build/目录下, 如果要运行默认测试, 可以在 shell 中执行这条命令:

```
make test
```

默认情况下会使用数据生成器 data_gen 生成范围在 0 到 100000 范围内的 500 个数字进行测试. 因为当数据太多的时候, 使用跳跃点方法进行计算的时候会发生溢出.

其中, data_gen 生成的输入存储在 samples/yet_another_sample.in 中, 程序的输出存储在 samples/bag.out 和 samples/jump_bag.out 中.

如果要使用数据生成器 data_gen 生成指定范围内的均匀分布的随机数进行测试, 可以用这条命令:

```
make test [from=<range start>] [to=<range end>] [amount=<number of random
    numbers>]
# e.g.
# make test from=0 to=12345 amount=50
```

如果要使用自定义文件进行测试, 可以用这条命令:

```
# 一般的0-1背包动态规划解法
build/BackPack01 ${infilename} [${outfilename}]
# 跳跃点优化的0-1背包动态规划解法
build/JumpPointBackPack ${infilename} [${outfilename}]
```

其中, ${infilename} 是按第一章中的输入格式1.1.4组织的输入数据. 默认情况下将同时输出到标准输出和 samples/mergesort.out, 可选自定义输出到 ${outfilename} 路径下. 输出格式为第一章中的输出格式1.1.4.

3.2.2 文档编译

除排序程序使用 make 管理之外, 本实验的课程报告使用 latex 进行编写, 因此同样可以使用 make 进行管理. 如果要编译文档, 运行这条命令:

```
make docs
```

make 将调用 xelatex 进行编译; 为了确保 cross-referencing 的正确工作, make 将执行两遍 xelatex 编译.

编译生成的中间文件及文档都保存在 docs/build 下, docs/report.pdf 和 docs/build/ report.pdf 均是生成的实验报告.

3.3 程序执行示例

此处演示程序从构建到测试的过程.

```
# user input
make clean

# console output
rm -rf build/*
/Library/Developer/CommandLineTools/usr/bin/make -C docs/ clean
rm -rf build/*

# user input
make # Compile backpack programs

# console output
clang++ -c -g -Wall -std=c++17 -O3 01bp/BackPack01.cpp -o build/BackPack01.o
clang++ -Wall -std=c++17 -O3 build/BackPack01.o -o build/BackPack01
clang++ -c -g -Wall -std=c++17 -O3 01bp_optimized/JumpPointBackPack.cpp -o
    build/JumpPointBackPack.o
clang++ -Wall -std=c++17 -O3 build/JumpPointBackPack.o -o
    build/JumpPointBackPack

# user input
make test_default # Run default test with sample given with assignment

# samples/bag.in:
70 3
71 100
69 1
1 2

# console output
build/BackPack01
[BackPack01DP] Time measured: 3.541e-06 seconds.
3
Weight | Value |
    69 | 1 |
     1 | 2 |

build/JumpPointBackPack
[JumpPointBackPack] Time measured: 6.417e-06 seconds.
3

# user input
# test with random number generator
make test # Compile data_gen and generate sample to test

# console output
clang++ -Wall -std=c++17 -O3 data_gen/data_gen.cpp -o build/data_gen
echo "[INFO]: Generate 500 random numbers ranging from 0 to 1000."
```

```
[INFO]: Generate 500 random numbers ranging from 0 to 1000.
build/data_gen 0 1000 500
build/BackPack01 samples/yet_another_sample.in
[BackPack01DP] Time measured: 0.000686916 seconds. #
    一般方法动态规划的求解时间
9712
Weight | Value | # 背包中的物品
    12 | 699 |
     1 | 907 |
    20 | 461 |
     6 | 174 |
    47 | 999 |
     3 | 736 |
     4 | 902 |
    24 | 579 |
     9 | 848 |
    19 | 476 |
    22 | 857 |
    15 | 369 |
     6 | 948 |
    18 | 757 |

build/JumpPointBackPack samples/yet_another_sample.in
[JumpPointBackPack] Time measured: 3.4542e-05 seconds. #
    应用跳跃点优化的动态规划求解时间
9712

# user input
make test from=0 to=1000 amount=100
# It means generate 100 random numbers ranging from 0 to 1000.

# console output
echo "[INFO]: Generate 100 random numbers ranging from 0 to 1000."
[INFO]: Generate 100 random numbers ranging from 0 to 1000.
build/data_gen 0 1000 100
build/BackPack01 samples/yet_another_sample.in
[BackPack01DP] Time measured: 0.000141333 seconds.
5575
Weight | Value |
    62 | 456 |
    80 | 932 |
    86 | 877 |
     3 | 641 |
   109 | 882 |
    76 | 885 |
    80 | 798 |
    15 | 104 |

build/JumpPointBackPack samples/yet_another_sample.in
[JumpPointBackPack] Time measured: 1.7125e-05 seconds.
5575
```

通过上述的例子, 可以看出, 当所给数据的数量级较大的时候, 使用跳跃点优化的动态规划算法要平均比一般方法快出 2 个数量级.

4 实验总结

本次实验花费了 7 小时 45 分钟在 C++ 排序程序代码编写上, 43 分钟在 makefile 用法的学习和编写上, 1 小时 57 分钟在 tex 实验报告的编写上.

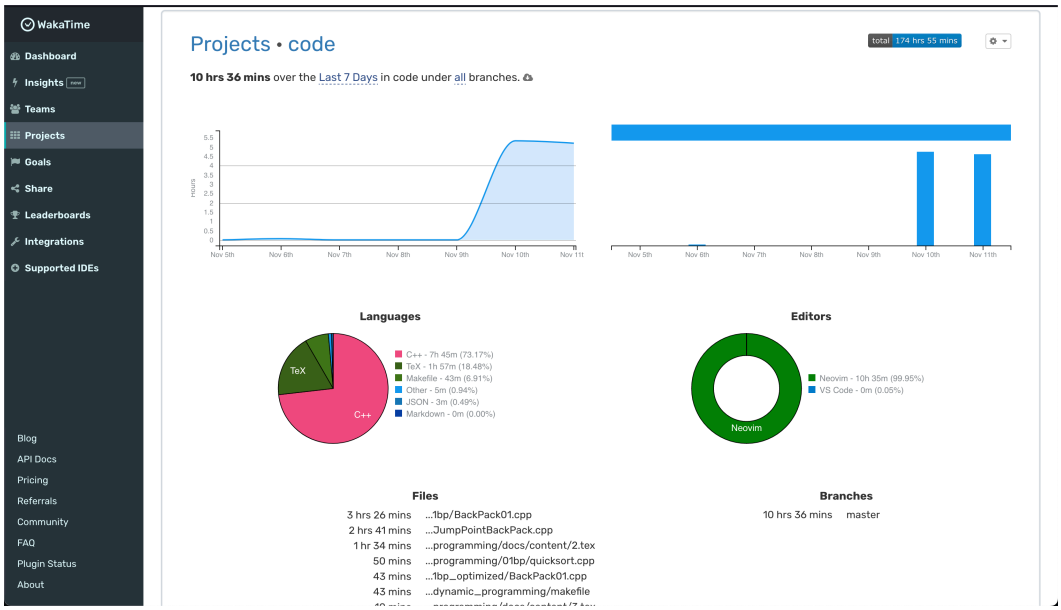

Figure 2: 编写时间统计

通过这次实验, 我动手实现了 0-1 背包问题的动态规划和跳跃点优化算法. 动手实践了各算法的时间复杂度和空间复杂度分析. 此外, 为了综合评价各算法的正确性, 以及衡量各算法的性能, 还动手实现了数据生成器, 用来方便的生成指定数量的均匀分布在给定范围的随机排序的数据作为排序程序的输入.

在构建和管理项目的工具方面, 学习并使用了 make 工具, 方便的实现了对程序的构建和测试; 此外, 对于文档, 也使用了 make 工具作为管理, 很大程度上方便了实验报告修改后的再次编译. 在实验报告编写方面, 练习了 latex 的使用, 对 latex 语法和使用更加熟悉.

但是, 由于时间有限, 没有实现更多的优化内容, 比如使用滚动数组压缩空间, 使用滚动数组优化跳跃点方法避免输入数据过多的时候占用空间指数级上升.

总的来说, 这次实验在动态规划算法之内及之外都学到了许多知识, 收获颇丰.

5 算法源代码

5.1 0-1 背包问题动态规划算法

```
#include <algorithm>
#include <assert.h>
#include <chrono>
#include <cstring>
#include <fstream>
```

```
#include <iomanip>
#include <iostream>
#include <queue>
#include <sstream>
#include <unordered_map>
#include <utility>
#include <vector>

class BackPack01 {
  public:
    BackPack01(std::istream &is, bool verbose = false) {
        // Initialize array that dp uses.
        for (int i = 0; i < MAXN; i++) {
            dp[i] = new int[MAXN];
        }

        // Read data from is.
        is >> volume >> n;

        for (int j = 0; j <= volume; j++) {
            dp[n][j] = 0;
        }
        for (int i = 0; i < n; i++) {
            is >> w[i] >> val[i];
        }

        // Set whether track items in backpack at every step.
        trackBackPackStatus = verbose;
    }

    BackPack01(std::fstream &fs, bool verbose = false) {
        // Initialize array that dp uses.
        for (int i = 0; i < MAXN; i++) {
            dp[i] = new int[MAXN];
        }

        // Read data from is.
        fs >> volume >> n;

        for (int j = 0; j <= volume; j++) {
            dp[n][j] = 0;
        }

        for (int i = 0; i < n; i++) {
            fs >> w[i] >> val[i];
        }
        // Set whether track items in backpack at every step.
        trackBackPackStatus = verbose;
    }

    ~BackPack01() {
        delete[] w;
        delete[] val;

        for (int i = 0; i < MAXN; i++) {
            delete[] dp[i];
        }
        delete[] dp;
```

```
    }

    std::string constructorDebugString() {
        std::stringstream ss;
        ss << "Volume: " << volume << ", Items: " << n << std::endl;
        for (int i = 0; i < n; i++) {
            ss << std::setw(3) << w[i] << " | " << std::setw(3) << val[i]
               << std::endl;
        }
        return ss.str();
    }

    void backPackDP(); // DP to calculate best maxVal.
    void backTrace(); // Back track to collect items in the backpack.

    int getMaxVal() const { return maxVal; }

    std::string itemsInMaxValBackPackToString() {
        std::stringstream ss;
        ss << std::setw(6) << "Weight"
           << " | " << std::setw(5) << "Value"
           << " |" << std::endl;
        for (auto &a : itemsInMaxValBackPack) {
            ss << std::setw(6) << w[a] << " | " << std::setw(5) << val[a]
               << " |" << std::endl;
        }
        return ss.str();
    }

    friend std::ostream &operator<<(std::ostream &os, BackPack01 &bp);

  private:
    bool trackBackPackStatus = false; // Track items in backpack at every
        state.
    int maxVal; // Max total value of items in backpack.
    int volume; // Volume of backpack.
    int n; // Number of items.
    std::vector<int> itemsInMaxValBackPack;

    static const int MAXN = 100000;

    int *w = new int[MAXN]; // Weight of items
    int *val = new int[MAXN]; // Value of items

    int **dp = new int *[MAXN]; // dp[MAXN][MAXN], dp[item][volume].
};

void BackPack01::backTrace() {
    int i = 0, j = volume;
    std::queue<int> q;
    while (i < n) {
        int curr = dp[i][j];
        if (dp[i + 1][j] != curr) {
            q.push(i); // TODO: To verify.
            j -= w[i];
        }
        i++;
    }
```

```
    while (!q.empty()) {
        itemsInMaxValBackPack.push_back(q.front());
        q.pop();
    }
}

std::ostream &operator<<(std::ostream &os, BackPack01 &bp) {
    os << bp.getMaxVal();
    if (bp.trackBackPackStatus) {
        os << std::endl;
        os << bp.itemsInMaxValBackPackToString();
    }
    return os;
}

void BackPack01::backPackDP() {
    maxVal = -1;

    // Initialize
    for (int j = 0; j <= volume; j++) {
        dp[n - 1][j] = (j >= w[n - 1]) ? val[n - 1] : 0;
    }

    int i = n - 2, j = volume;
    for (; i >= 0; i--) {
        j = volume;
        for (; j >= 0; j--) {

            // Compare total value of items in the backpack between put
            // item[i] in and not put it in.
            dp[i][j] = (j >= w[i]) ? std::max(dp[i + 1][j],
                                              dp[i + 1][j - w[i]] + val[i])
                                   : dp[i + 1][j];
        }
    }

    maxVal = dp[0][volume];
}

int main(int argc, char **argv) {
    // std::ios_base::sync_with_stdio(false);
    std::fstream fs;
    // Will attempt to open given file if received more then one
    // parameters.
    if (argc > 1) {
        try {
            fs.open(argv[1], std::ios_base::in);
        } catch (std::system_error &e) {
            std::cerr << e.code().message() << std::endl;
            exit(-1);
        }
    } else {
        // Open default input file
        try {
            char filename[] = "samples/bag.in";
            fs.open(filename, std::ios_base::in);
        } catch (std::system_error &e) {
            std::cerr << e.code().message() << std::endl;
```

```
        }
    }

    assert(fs.is_open() == true);

    void *p = new char[sizeof(BackPack01)];
    bool verbose = true;
    BackPack01 *backPack01 = new (p) BackPack01(fs, verbose);
    fs.close();

    auto begin = std::chrono::high_resolution_clock::now();

    backPack01->backPackDP();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
    std::cout.precision(6);
    std::cout << "[BackPack01DP] Time measured: " << elapsed.count() * 1e-9
              << " seconds.\n";

    char output_file_name[] = "samples/bag.out";
    fs.open(output_file_name, std::ios_base::out);
    // Print sorted numbers to output_file_name
    fs << *backPack01 << std::endl;
    fs.close();

    if (verbose) {
        backPack01->backTrace();
    }
    std::cout << *backPack01 << std::endl;

    backPack01->~BackPack01();
    delete[] (char *)p;
    return 0;
}
```

5.2 应用跳跃表优化的 0-1 背包问题动态规划算法

```
#include <algorithm>
#include <assert.h>
#include <chrono>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <queue>
#include <sstream>
#include <unordered_map>
#include <utility>
#include <vector>

class Pack {
  public:
    Pack(int weight = 0, int val = 0) : weight(weight), val(val) {}
    int weight;
    int val;
```

```
    friend bool operator<(const Pack &ba, const Pack &bb) {
        return (ba.weight != bb.weight) ? ba.weight > bb.weight
                                        : ba.val < bb.val;
    }
};

class BackPackJumpPoint {
  public:
    BackPackJumpPoint(std::istream &is, bool verbose = false) {
        for (int i = 0; i < MAXN * 1000; i++) {
            jp[i] = new int[2];
        }

        // Read data from is.
        is >> volume >> n;
        for (int i = 0; i < n; i++) {
            is >> w[i] >> val[i];
        }

        // Set whether track items in backpack at every step.
        traceBackPackStatus = verbose;
    }

    BackPackJumpPoint(std::fstream &fs, bool verbose = false) {
        for (int i = 0; i < MAXN * 1000; i++) {
            jp[i] = new int[2];
        }

        // Read data from is.
        fs >> volume >> n;
        for (int i = 0; i < n; i++) {
            fs >> w[i] >> val[i];
        }
        // Set whether track items in backpack at every step.
        traceBackPackStatus = verbose;
    }

    ~BackPackJumpPoint() {
        for (int i = 0; i <= n; i++) {
            delete[] jp[i];
        }
        delete[] jp;

        delete[] w;
        delete[] val;
    }

    std::string constructorDebugString() {
        std::stringstream ss;
        ss << "Volume: " << volume << ", Items: " << n << std::endl;
        for (int i = 0; i < n; i++) {
            ss << std::setw(3) << w[i] << " | " << std::setw(3) << val[i]
               << std::endl;
        }
        return ss.str();
    }

    void JumpPointBackPackDP(); // BackPack DP using jump point optimization.
```

```
    int getMaxVal() const { return maxVal; }

    std::string itemsInMaxValBackPackToString() {
        std::stringstream ss;
        ss << std::setw(4) << "Item"
           << " | " << std::setw(6) << "Weight"
           << " | " << std::setw(5) << "Value"
           << " |" << std::endl;

        for (auto &a : itemsInMaxValBackPack) {
            ss << std::setw(4) << a << " | " << std::setw(6) << w[a] << " | "
               << std::setw(5) << val[a] << " |" << std::endl;
        }
        return ss.str();
    }

    friend std::ostream &operator<<(std::ostream &os, BackPackJumpPoint &bp);

  private:
    bool traceBackPackStatus = false; // Track items in backpack at every
        state.
    int maxVal; // Max total value of items in backpack.
    int volume; // Volume of backpack.
    int n; // Number of items.

    static const int MAXN = 100000;
    int *w = new int[MAXN]; // Weight of items
    int *val = new int[MAXN]; // Value of items
    int **jp = new int *[MAXN * 1000]; // Jump points

    std::vector<int> itemsInMaxValBackPack;
};

std::ostream &operator<<(std::ostream &os, BackPackJumpPoint &bp) {
    os << bp.maxVal;
    return os;
}

void BackPackJumpPoint::JumpPointBackPackDP() {
    int *head = new int[n + 2]; // Track jump point start position.
    head[n] = 0;
    jp[0][0] = 0; // Store item weight
    jp[0][1] = 0; // Store item value

    // Left points to first jump point of p[i+1], right points to last jump
    // point of p[i+1]. Next is position where next jump point will store.
    int left = 0, right = 0, next = 1;
    head[n - 1] = 1; // Points to the position of first jump point of
        item[n-1].

    for (int i = n - 1; i >= 0; i--) {
        int k = left; // k points to jump points of p[], move k to
                      // evaluate controlled points in p[] and
                      // p[]+(w,v)
        for (int j = left; j <= right; j++) {

            if (jp[j][0] + w[i] > volume) {
                // No enough backpack space to fit item[i] in, exit loop.
```

```
                break;
            }

            // Compute new jump point as jp[]+(w,v).
            int x = jp[j][0] + w[i];
            int y = jp[j][1] + val[i];

            // If jp[k][0] < x, then it must be a jump point of current
            // item.
            while (k <= right && jp[k][0] < x) {
                jp[next][0] = jp[k][0];
                jp[next++][1] = jp[k++][1];
            }

            // Clear controlled jump point.
            if (k <= right && jp[k][0] == x) {
                if (y < jp[k][1]) {
                    y = jp[k][1];
                }
                k++;
            }

            if (y > jp[next - 1][1]) {
                jp[next][0] = x;
                jp[next++][1] = y;
            }

            while (k <= right && jp[k][1] <= jp[next - 1][1]) {
                k++;
            }
        }

        // Add remaining jump points.
        while (k <= right) {
            jp[next][0] = jp[k][0];
            jp[next++][1] = jp[k++][1];
        }

        left = right + 1;
        right = next - 1;

        head[i - 1] = next;
    }

    if (traceBackPackStatus) {
        std::stringstream ss;
        for (int i = n; i > 0; i--) {
            ss << std::endl;
            ss << "| " << std::setw(4) << "id"
               << " | " << std::setw(4) << "head"
               << " |" << std::endl;
            ss << "| " << std::setw(4) << i << " | " << std::setw(4) <<
                head[i]
               << " |" << std::endl;
            // Begin and end of item[i-1].
            int begin = head[i];
            int end = head[i - 1];
```

```
            ss << "jump points:\n";
            for (int j = begin; j < end; j++) {
                ss << "| " << std::setw(4) << jp[j][0] << " | " <<
                    std::setw(4)
                   << jp[j][1] << " |" << std::endl;
            }
        }
        std::cout << ss.str();
    }

    maxVal = jp[next - 1][1];
}

int main(int argc, char **argv) {
    // std::ios_base::sync_with_stdio(false);
    std::fstream fs;
    // Will attempt to open given file if received more then one
    // parameters.
    if (argc > 1) {
        try {
            fs.open(argv[1], std::ios_base::in);
        } catch (std::system_error &e) {
            std::cerr << e.code().message() << std::endl;
            exit(-1);
        }
    } else {
        // Open default input file
        try {
            char filename[] = "samples/bag.in";
            fs.open(filename, std::ios_base::in);
        } catch (std::system_error &e) {
            std::cerr << e.code().message() << std::endl;
        }
    }
    assert(fs.is_open() == true);

    void *pJ = new char[sizeof(BackPackJumpPoint)];
    bool verbose = false;
    BackPackJumpPoint *backPackJumpPoint =
        new (pJ) BackPackJumpPoint(fs, verbose);
    fs.close();

    if (verbose) {
        std::cout << backPackJumpPoint->constructorDebugString();
    }

    auto begin = std::chrono::high_resolution_clock::now();

    backPackJumpPoint->JumpPointBackPackDP();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
    std::cout.precision(6);
    std::cout << "[JumpPointBackPack] Time measured: " << elapsed.count() *
        1e-9
              << " seconds.\n";
```

```
    char output_file_name[] = "samples/jump_bag.out";
    fs.open(output_file_name, std::ios_base::out);
    // Print sorted numbers to output_file_name
    fs << *backPackJumpPoint << std::endl;
    std::cout << *backPackJumpPoint << std::endl;
    fs.close();

    delete[] (char *)pJ;
    return 0;
}
```

5.3 数据生成器 data_gen

```
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>
#include <string>

using namespace std;
const int default_amount_of_random_numbers = 50;

class Parser {
  public:
    Parser(int argc, char **argv);
    const int get_range_from() const { return range_from; }
    const int get_range_to() const { return range_to; }
    const int get_amount_of_random_numbers() const {
        return amount_of_random_numbers;
    }

  private:
    int parse_unit_arg(char *single_arg);
    void parse_arg_count_4();
    void parse_arg_count_3();
    void parse_no_arg();
    int argc;
    char **argv;
    int range_from;
    int range_to;
    int amount_of_random_numbers;
};
int Parser::parse_unit_arg(char *single_arg) {
    int x = 0;
    string arg = single_arg;
    try {
        size_t pos;
        x = stoi(arg, &pos);
        if (pos < arg.size()) {
            cerr << "Trailing characters after number: " << arg << "\n";
        }
    } catch (invalid_argument const &e) {
        cerr << "Invalid number: " << arg << "\n";
    } catch (out_of_range const &e) {
        cerr << "Number out of range: " << arg << "\n";
```

```
    }
    return x;
}

/**
 * Generate an amount of argv[3] random numbers distributed
 * from argv[1] to argv[2].
 */
void Parser::parse_arg_count_4() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
    amount_of_random_numbers = parse_unit_arg(argv[3]);
}

void Parser::parse_arg_count_3() {
    range_from = parse_unit_arg(argv[1]);
    range_to = parse_unit_arg(argv[2]);
}

Parser::Parser(int argc, char **argv) : argc(argc), argv(argv) {
    if (argc == 4) {
        parse_arg_count_4();
    } else if (argc == 3) {
        parse_arg_count_3();
        amount_of_random_numbers = default_amount_of_random_numbers;
    } else {
        range_from = 0;
        range_to = 1000;
        amount_of_random_numbers = default_amount_of_random_numbers;
    }
}

int main(int argc, char **argv) {
    Parser parser(argc, argv);
    int range_from = parser.get_range_from();
    int range_to = parser.get_range_to();
    int amount_of_random_numbers = parser.get_amount_of_random_numbers();

    random_device rand_dev;
    mt19937 generator(rand_dev());
    uniform_int_distribution<int> distr(range_from, range_to);

    fstream fs;
    string filename = "samples/yet_another_sample.in";
    fs.open(filename, ios_base::out);

    if (fs.is_open()) {
        fs << distr(generator) << " " << amount_of_random_numbers <<
            std::endl;
        for (int i = 0; i < amount_of_random_numbers; i++) {
            fs << distr(generator) << " " << distr(generator) << std::endl;
        }
    }
    fs.close();
    return 0;
}
```

6 附录

代码仓库: dynamic_programming