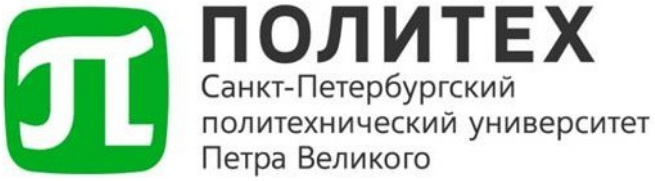
Министерство образования и науки РФ
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии



К У Р С О В А Я Р А Б О Т А

Методы разработки алгоритмов

по дисциплине «**Алгоритмы и структуры данных»**

Выполнил
студент гр. 5130904/30005 Кладковой Максим Дмитриевич

Руководитель Червинский Алексей Петрович

«25» мая 2024

Санкт-Петербург

2024 г

Содержание

Введение.

Тема: Методы разработки алгоритмов

Вариант 3.3.
Задача о рюкзаке

Решение дискретной задачи о рюкзаке методом динамического программирования, методом поиска с возвратом, методом ветвей и границ, методом грубой силы. Проанализировать полученные решения.

Основная часть

Описание алгоритма решения и используемых структур данных

Алгоритм динамического программирования

Алгоритм динамического программирования (DP) для задачи о рюкзаке строит двумерную таблицу, где строки соответствуют предметам, а столбцы - возможным весам рюкзака от 0 до максимального. Ячейка таблицы `iw` содержит максимальную стоимость, которую можно получить, заполнив рюкзак весом `w` предметами до `i`-го включительно.

Используемые структуры данных:

- Двумерный массив `dp` для хранения максимальных стоимостей.
- Одномерные массивы для хранения весов `weights` и стоимостей `values` предметов.

Метод поиска с возвратом (Backtracking)

Метод поиска с возвратом использует рекурсию для перебора всех возможных комбинаций предметов. Для каждой комбинации проверяется, превышает ли суммарный вес рюкзака допустимый предел и если нет, то сравнивается текущая стоимость с наилучшей найденной стоимостью.

Используемые структуры данных:

- Рекурсивный стек для отслеживания текущего состояния.
- Одномерные массивы для хранения весов `weights` и стоимостей `values` предметов.
- Логические массивы `currentSet` и `bestSet` для хранения текущего и лучшего набора предметов.

Метод ветвей и границ (Branch and Bound)

Метод ветвей и границ улучшает поиск с возвратом за счёт отсечения заведомо неоптимальных путей. При этом используется оценка верхней границы стоимости, которая может быть получена из текущего состояния.

Используемые структуры данных:

- Очередь или приоритетная очередь для хранения узлов дерева решений.
- Одномерные массивы для хранения весов `weights` и стоимостей `values` предметов.
- Логические массивы `currentSet` и `bestSet` для хранения текущего и лучшего набора предметов.

Метод грубой силы (Brute Force)

Метод грубой силы перебирает все возможные комбинации предметов и выбирает комбинацию с наибольшей стоимостью, удовлетворяющую ограничениям по весу.

Используемые структуры данных:

- Одномерные массивы для хранения весов `weights` и стоимостей `values` предметов.
- Логические массивы `currentSet` и `bestSet` для хранения текущего и лучшего набора предметов.

Анализ алгоритма

Временная и пространственная сложность

- **Динамическое программирование:**
  - Временная сложность: $O(n \cdot W)$, где n — количество предметов, W — максимальный вес рюкзака.
  - Это объясняется тем, что алгоритм заполняет таблицу размером $n \times W$, где n — количество предметов, а W — максимальная вместимость рюкзака. Для 10 предметов и максимальной вместимости 100 алгоритм завершился за 172 микросекунды, а для 25 предметов — за 77 микросекунд. Наблюдается квадратичный рост времени выполнения от входных параметров, что согласуется с теоретической временной сложностью.
  - Пространственная сложность: $O(n \cdot W)$.
- **Поиск с возвратом:**
  - Временная сложность: В худшем случае $O(2^n)$.
  - Этот метод имеет экспоненциальную временную сложность, так как в худшем случае необходимо перебрать все возможные комбинации предметов. В примере время выполнения для 10 предметов составило 88 микросекунд, а для 25 предметов — 528687 микросекунд, что подтверждает экспоненциальный рост времени выполнения с увеличением количества предметов.
  - Пространственная сложность: $O(n)$ для рекурсивного стека.
- **Ветви и границ:**
  - Временная сложность: Улучшается за счёт отсечения, но в худшем случае $O(2^n)$.
  - Метод ветвей и границ имеет экспоненциальную временную сложность в худшем случае, но на практике оказывается быстрее за счет отсечений. Для 10 предметов алгоритм завершился за 24 микросекунды, а для 25 предметов — за 65 микросекунд, что

показывает значительное улучшение по сравнению с методом поиска с возвратом и подтверждает эффективность отсечений.
- Пространственная сложность: $O(n)$.

- **Грубая сила:**
  - Временная сложность: $O(2^n)$.
  - Метод грубой силы также имеет экспоненциальную временную сложность, так как перебирает все возможные комбинации предметов. Для 10 предметов алгоритм завершился за 514 микросекунд, а для 25 предметов — за 15832597 микросекунд, что также подтверждает экспоненциальный рост времени выполнения.
  - Пространственная сложность: $O(n)$.

Эффективность алгоритмов

- Алгоритм динамического программирования наиболее эффективен для задач с умеренными значениями W.
- Метод поиска с возвратом подходит для небольших значений n из-за его экспоненциальной временной сложности.
- Метод ветвей и границ эффективнее, чем поиск с возвратом, за счёт отсечения неэффективных путей.
- Метод грубой силы наименее эффективен и используется только для проверки и сравнений.

Описание спецификации программы

Детальные требования

1. Программа должна решать задачу о рюкзаке четырьмя методами: динамическим программированием, поиском с возвратом, методом ветвей и границ и методом грубой силы.
2. Входные данные:
    - Количество предметов n.
    - Веса предметов.
    - Стоимости предметов.
    - Максимальный вес рюкзака W.
3. Выходные данные:
    - Максимальная стоимость, которую можно уместить в рюкзак, для каждого метода.
4. Программа должна иметь возможность принимать входные данные из файла и выводить результаты в файл.
5. Программа должна обеспечивать контроль ошибок, включая проверку на корректность входных данных.

Описание программы

Структура программы

Программа состоит из следующих модулей:

1. Модуль чтения/записи данных.
2. Модуль реализации каждого из четырёх алгоритмов.
3. Модуль анализа и сравнения результатов.

Форматы входных данных:

- Первая строка: количество предметов `n`.
- Вторая строка: максимальный вес рюкзака `W`.
- Третья строка: веса предметов, разделённые пробелами.
- Четвёртая строка: стоимости предметов, разделённые пробелами.

Форматы выходных данных:

- Одна строка для каждого метода, содержащая максимальную стоимость, которую можно уместить в рюкзак.

Приложение 1: Текст программы

KnapsackProblem.hpp

```
1:  class Knapsack
2:  {
3:  private:
4:    struct Item
5:    {
6:      int weight;
7:      int value;
8:    };
9:    int n;
10:   int W;
11:   Item* items;
12:   void printSelectedItems(int** dp);
13:   void printSelectedItems(bool* selectedItems);
14:   void knapsackBacktrackingUtil(int i, int currentWeight, int currentValue, int& maxValue, bool* bestSet, bool* currentSet);
15:   void knapsackBranchAndBoundUtil(int i, int currentWeight, int currentValue, int& maxValue, bool* bestSet, bool* currentSet);
16: public:
17:   Knapsack(int numItems, int maxWeight);
18:   ~Knapsack();
19:   void printItems();
20:   void knapsackDP();
21:   void knapsackBacktracking();
22:   void knapsackBranchAndBound();
23:   void knapsackBruteForce();
24: };
```

KnapsackProblem.cpp

```
25: #include "KnapsackProblem.h"
26: #include <iostream>
27: #include <cstdlib>
28: #include <ctime>
29:
30: Knapsack::Knapsack(int numItems, int maxWeight)
31: {
32:   n = numItems;
33:   W = maxWeight;
34:   items = new Item[n];
35:   srand(time(0));
36:   for (int i = 0; i < n; ++i) {
37:     items[i].weight = rand() % 100 + 1;
38:     items[i].value = rand() % 100 + 1;
39:   }
40: }
41:
```

```
46:   Knapsack::~Knapsack()
47:   {
48:     delete[] items;
49:   }
50:
51:   void Knapsack::printItems()
52:   {
53:     std::cout << "Items (weight, value):\n";
54:     for (int i = 0; i < n; ++i) {
55:       if (i % 10 == 0 && i > 1)
56:       {
57:         std::cout << std::endl;
58:       }
59:       std::cout << "(" << items[i].weight << ", " << items[i].value << ") ";
60:     }
61:     std::cout << std::endl << std::endl;
62:   }
63:
64:   void Knapsack::knapsackDP()
65:   {
66:     int** dp = new int* [n + 1];
67:     for (int i = 0; i <= n; ++i) {
68:       dp[i] = new int[W + 1];
69:     }
70:
71:     for (int i = 0; i <= n; ++i) {
72:       for (int w = 0; w <= W; ++w) {
73:         if (i == 0 || w == 0) {
74:           dp[i][w] = 0;
75:         }
76:         else if (items[i - 1].weight <= w) {
77:           dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - items[i -
    1].weight] + items[i - 1].value);
78:         }
79:         else {
80:           dp[i][w] = dp[i - 1][w];
81:         }
82:       }
83:     }
84:
85:     std::cout << "Dynamic Programming: Maximum value is " << dp[n][W] << "\
    n";
86:     printSelectedItems(dp);
87:
88:     for (int i = 0; i <= n; ++i) {
89:       delete[] dp[i];
90:     }
91:     delete[] dp;
92:   }
93:
94:   void Knapsack::knapsackBacktracking()
95:   {
96:     int maxValue = 0;
```

```
 97:     bool* bestSet = new bool[n];
 98:     bool* currentSet = new bool[n];
 99:
100:     knapsackBacktrackingUtil(0, 0, 0, maxValue, bestSet, currentSet);
101:
102:     std::cout << "Backtracking: Maximum value is " << maxValue << "\n";
103:     printSelectedItems(bestSet);
104:
105:     delete[] bestSet;
106:     delete[] currentSet;
107:   }
108:
109:   void Knapsack::knapsackBacktrackingUtil(int i, int currentWeight, int
    currentValue, int& maxValue, bool* bestSet, bool* currentSet)
110:   {
111:     if (i == n) {
112:       if (currentWeight <= W && currentValue > maxValue) {
113:         maxValue = currentValue;
114:         for (int j = 0; j < n; ++j) {
115:           bestSet[j] = currentSet[j];
116:         }
117:       }
118:       return;
119:     }
120:     currentSet[i] = true;
121:     knapsackBacktrackingUtil(i + 1, currentWeight + items[i].weight,
    currentValue + items[i].value, maxValue, bestSet, currentSet);
122:
123:     currentSet[i] = false;
124:     knapsackBacktrackingUtil(i + 1, currentWeight, currentValue, maxValue,
    bestSet, currentSet);
125:   }
126:
127:   void Knapsack::knapsackBranchAndBound()
128:   {
129:     int maxValue = 0;
130:     bool* bestSet = new bool[n];
131:     bool* currentSet = new bool[n];
132:
133:     knapsackBranchAndBoundUtil(0, 0, 0, maxValue, bestSet, currentSet);
134:
135:     std::cout << "Branch and Bound: Maximum value is " << maxValue << "\n";
136:     printSelectedItems(bestSet);
137:
138:     delete[] bestSet;
139:     delete[] currentSet;
140:   }
141:
142:   void Knapsack::knapsackBranchAndBoundUtil(int i, int currentWeight, int
    currentValue, int& maxValue, bool* bestSet, bool* currentSet)
143:   {
144:     if (i == n) {
145:       if (currentWeight <= W && currentValue > maxValue) {
```

```
146:           maxValue = currentValue;
147:           for (int j = 0; j < n; ++j) {
148:             bestSet[j] = currentSet[j];
149:           }
150:         }
151:         return;
152:       }
153:
154:       int bound = currentValue;
155:       for (int j = i; j < n; ++j) {
156:         bound += items[j].value;
157:       }
158:
159:       if (bound <= maxValue) {
160:         return;
161:       }
162:
163:       if (currentWeight + items[i].weight <= W) {
164:         currentSet[i] = true;
165:         knapsackBranchAndBoundUtil(i + 1, currentWeight + items[i].weight,
    currentValue + items[i].value, maxValue, bestSet, currentSet);
166:       }
167:
168:       currentSet[i] = false;
169:       knapsackBranchAndBoundUtil(i + 1, currentWeight, currentValue, maxValue,
    bestSet, currentSet);
170:     }
171:
172:     void Knapsack::knapsackBruteForce()
173:     {
174:       int maxValue = 0;
175:       bool* bestSet = new bool[n];
176:
177:       int totalSubsets = 1 << n;
178:       for (int subset = 0; subset < totalSubsets; ++subset) {
179:         int weightSum = 0;
180:         int valueSum = 0;
181:         bool* currentSet = new bool[n];
182:
183:         for (int i = 0; i < n; ++i) {
184:           if (subset & (1 << i)) {
185:             weightSum += items[i].weight;
186:             valueSum += items[i].value;
187:             currentSet[i] = true;
188:           }
189:           else {
190:             currentSet[i] = false;
191:           }
192:         }
193:
194:         if (weightSum <= W && valueSum > maxValue) {
195:           maxValue = valueSum;
196:           for (int i = 0; i < n; ++i) {
```

```
197:             bestSet[i] = currentSet[i];
198:           }
199:         }
200:
201:         delete[] currentSet;
202:       }
203:
204:       std::cout << "Brute Force: Maximum value is " << maxValue << "\n";
205:       printSelectedItems(bestSet);
206:
207:       delete[] bestSet;
208:     }
209:
210:     void Knapsack::printSelectedItems(int** dp)
211:     {
212:       int w = W;
213:       std::cout << "Selected items: \n";
214:       for (int i = n; i > 0 && w > 0; --i) {
215:         if (dp[i][w] != dp[i - 1][w]) {
216:           std::cout << "(" << items[i - 1].weight << ", " << items[i - 1].value
      << ")";
217:           w -= items[i - 1].weight;
218:         }
219:       }
220:       std::cout << std::endl;
221:     }
222:
223:     void Knapsack::printSelectedItems(bool* selectedItems)
224:     {
225:       std::cout << "Selected items: \n";
226:       for (int i = 0; i < n; ++i) {
227:         if (selectedItems[i]) {
228:           std::cout << "(" << items[i].weight << ", " << items[i].value << ")";
229:         }
230:       }
231:       std::cout << std::endl;
232:     }
```

Приложение 2. Протоколы отладки

```
    ----- Knapsack Problem -----

    --- Solving the problem for 10 items ---

    Items (weight, value):
    (45, 9) (59, 93) (70, 25) (20, 40) (85, 87) (69, 58) (40, 51) (51, 32) (61, 23) (32, 1)

    Dynamic Programming: Maximum value is 51
    Selected items:
    (40, 51)
    The time: 172 microseconds

    Backtracking: Maximum value is 51
    Selected items:
    (40, 51)
    The time: 88 microseconds

    Branch and Bound: Maximum value is 51
    Selected items:
    (40, 51)
    The time: 24 microseconds

    Brute Force: Maximum value is 51
    Selected items:
    (40, 51)
    The time: 514 microseconds

    --- Solving the problem for 25 items ---

    Items (weight, value):
    (45, 9) (59, 93) (70, 25) (20, 40) (85, 87) (69, 58) (40, 51) (51, 32) (61, 23) (32, 1)
    (27, 69) (83, 80) (43, 26) (1, 94) (2, 36) (61, 46) (96, 20) (91, 18) (96, 10) (57, 80)
    (96, 26) (90, 35) (28, 40) (18, 88) (62, 2)

    Dynamic Programming: Maximum value is 287
    Selected items:
    (18, 88)(2, 36)(1, 94)(27, 69)
    The time: 77 microseconds

    Backtracking: Maximum value is 287
    Selected items:
    (27, 69)(1, 94)(2, 36)(18, 88)
    The time: 528687 microseconds

    Branch and Bound: Maximum value is 287
    Selected items:
    (27, 69)(1, 94)(2, 36)(18, 88)
    The time: 65 microseconds
```

```
281:
282:    Brute Force: Maximum value is 287
283:    Selected items:
284:    (27, 69)(1, 94)(2, 36)(18, 88)
285:    The time: 15832597 microseconds
```

Заключение

В процессе работы были изучены и реализованы четыре алгоритма решения задачи о рюкзаке: метод динамического программирования, метод поиска с возвратом, метод ветвей и границ и метод грубой силы. Проведён анализ их временной и пространственной сложности. В результате практической части работы разработана программа, реализующая данные методы и позволяющая сравнивать их эффективность на различных наборах данных.

Список литературы

1. [Кормен, Т. Х. и др. "Алгоритмы: построение и анализ". Москва: Вильямс, 2005.]
2. [Horowitz, E., Sahni, S. "Fundamentals of Computer Algorithms". Computer Science Press, 1978.]
3. [Knuth, D. "The Art of Computer Programming". Addison-Wesley, 1997.]