Новосибирский государственный технический университет
Кафедра теоретической и прикладной информатики

Расчетно-графическая работа

Алгоритм сжатия данных

ФПМИ, ПМ-24

Параскун И., Герасименко В.

преподаватель
Сивак Мария Алексеевна

Новосибирск, 2024

Содержание

1 Введение

Данная расчетно-графическая работа представляет собой комплекс проведенных теоретических и прикладных исследований.

1.1 Алгоритм Хаффмана

Первой частью работы стало изучение алгоритма Хаффмана в качестве алгоритма сжатия данных, выявление его преимуществ и недостатков, а также определение оптимальных условий для его применения.

1.2 Практическая реализация

Вторая часть состояла в практической реализации изученного алгоритма. Задача заключалась в том, чтобы предоставить компактное решение реальной проблемы - сжатие данных.

2 Теоретическая часть

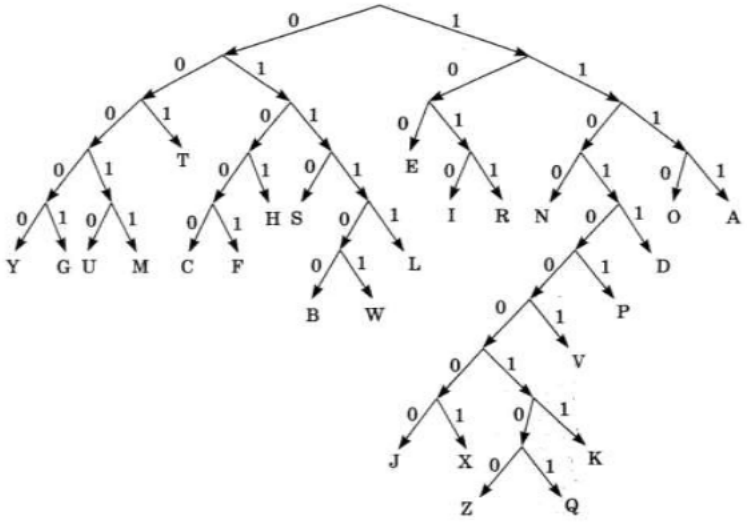
Алгоритм Хаффмана - алгоритм оптимального префиксного кодирования алфавита. Основная используемая структура данных - дерево Хаффмана.

2.1 Дерево Хаффмана

Дерево Хаффмана представляет собой бинарное дерево со свойствами кучи, листья - кодируемые символы. Существует два варианта исполнения - статический и динамический.

2.1.1 Статическое дерево Хаффмана

Статическое дерево Хаффмана представляет собой дерево со свойствами бинарной кучи. В дереве закодирован предопределенный набор символов (например, английский алфавит) на основе исследований о средней частоте их использования.



2.1.2 Динамическое дерево Хаффмана

Динамическое дерево Хаффмана строится по кодируемому файлу во время первого обхода. Коды получаются меньше, чем в статической реализации, однако появляется необходимость в записи дополнительной информации о частоте символов.

3 Описание программы

Программа состоит из вспомогательных модулей, подпрограмм кодирования и декодирования.

3.1 Вспомогательные модули

3.1.1 Очередь по приоритетам

Очередь по приоритетам необходима для построения дерева Хаффмана. Выбранная реализация - бинарная куча, первый элемент которой - узел дерева с минимальной частотой.
Асимтотические оценки:

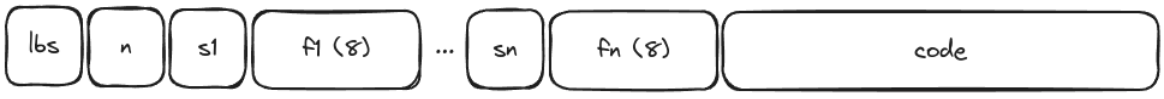
- Вставка - $\mathcal{O}(\log n)$
- Извлечение - $\mathcal{O}(\log n)$
- Изменение - $\mathcal{O}(\log n)$

3.1.2 Дерево Хаффмана

Дерево Хаффмана (1) реализовано по принципу связного списка. Каждое звено, помимо основной информации, хранит свой индекс в очереди, ссылку на левого потомка и ссылку на правого потомка.
Основной модуль - подпрограмма построения дерева (2).

3.2 Структура данных в памяти

Результат выполения программы сохраняется в память следующим образом:



Здесь, единичный квадратик - 1 байт, *lbs* - число бит в последнем байте кодового пространства, n - число уникальных символов. Для каждого уникального символа (s_n) хранится соответсвующая ему частота (f_n).

3.3 Подпрограмма кодирования

В общем виде алгоритм кодирования (3) выглядит следующим образом:

1. Чтение входного файла, составление очереди.
2. Построение дерева Хаффмана.

3. Повторное чтение входного файла, запись кодового пространства в результирующий файл.

4. Запись мета информации в результирующий файл.

Алгоритм построения дерева Хаффмана модифицирует данную ему очередь. В связи с этим, очередь, после построения, должна быть скопирована для дальнейшего занесения в мета информацию.

3.4 Подпрограмма декодирования

Алгоритм декодирования (4):

1. Чтение мета информации из входного файла.

2. Построение дерева Хаффмана.

3. Чтение кодового пространства из входного файла, обход дерева и запись результата в файл.

Пары символ-частота сохраняются в мета блоке в том порядке, в котором они находились во внутреннем массиве очереди, а не по приоритетам. Это позволяет сразу составить требуемую для алгоритма декодирования очередь, без необходимости восстановления свойств бинарной кучи.

4 Результаты тестирования

4.1 Корректность представления

Для проверки соотвествия представления заявленному, приведем простой тестовый файл и результат работы алгоритма сжатия:

```
Входной текст (in.txt):
ABC

Результат (in.txt.huff):
00000000  08 04 41 01 00 00 00 00  00 00 00 42 01 00 00 00  |..A........B....|
00000010  00 00 00 00 43 01 00 00  00 00 00 00 00 0a 01 00  |....C...........|
00000020  00 00 00 00 00 00 39                              |......9|
00000027
```

**четвертый символ - символ переноса строки. Он имеет ASCII код 0a.*

4.2 Кодирование ASCII символов

```
Входной текст (build.gradle):
plugins {
    id 'cpp-application'
}
```

```
application {
    binaries.configureEach {
        def compileTask = compileTask.get()

        compileTask.source.from fileTree(dir: "src/main/c", include: "**/*.c")
        compileTask.compilerArgs = ["-x", "c", "-std=c11", "-g3"]

        def linkTask = linkTask.get().linkerArgs = ["-nodefaultlibs", "-lc"]
    }
}
```

Результат:
357* build.gradle
592* build.gradle.huff

**размер файла в байтах*

4.3 Кодирование не-ASCII символов

Входной текст (in.txt):
ABC

Первый круг:
```
00000000  08 04 41 01 00 00 00 00  00 00 00 42 01 00 00 00  |..A........B....|
00000010  00 00 00 00 43 01 00 00  00 00 00 00 00 0a 01 00  |....C...........|
00000020  00 00 00 00 00 00 39                              |......9|
00000027
```

Второй круг:
```
00000000  06 09 08 01 00 00 00 00  00 00 00 04 01 00 00 00  |................|
00000010  00 00 00 00 41 01 00 00  00 00 00 00 00 0a 01 00  |....A...........|
00000020  00 00 00 00 00 00 00 1c  00 00 00 00 00 00 00 42  |...............B|
00000030  01 00 00 00 00 00 00 00  43 01 00 00 00 00 00 00  |........C.......|
00000040  00 01 04 00 00 00 00 00  00 00 39 01 00 00 00 00  |..........9.....|
00000050  00 00 00 63 96 7f 43 fa  8f ef 3f b4              |...c..C...?.|
0000005c
```

4.4 Кодирование специализированных файлов

Под специализированностью понимается низкая уникальность и высокая частотность.

Входной текст (in.txt):
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG

Результат (in.txt.huff):
00000000  04 08 0a 07 00 00 00 00  00 00 00 47 38 00 00 00  |...........G8...|
00000010  00 00 00 00 46 39 00 00  00 00 00 00 00 44 3b 00  |....F9.......D;.|
00000020  00 00 00 00 00 00 43 3c  00 00 00 00 00 00 00 42  |......C<.......B|
00000030  3d 00 00 00 00 00 00 00  45 3a 00 00 00 00 00 00  |=.......E:......|
00000040  00 41 3e 00 00 00 00 00  00 00 ff ff ff ff ff ff  |.A>.............|
00000050  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000060  ff c6 db 6d b6 db 6d b6  db 6d b6 db 6d b6 db 6d  |...m..m..m..m..m|
00000070  b6 db 6d b6 db 6d b6 db  61 6d b6 db 6d b6 db 6d  |..m..m..am..m..m|
00000080  b6 db 6d b6 db 6d b6 db  6d b6 db 6d b6 db 6d a2  |..m..m..m..m..m.|
00000090  49 24 92 49 24 92 49 24  92 49 24 92 49 24 92 49  |I$.I$.I$.I$.I$.I|
000000a0  24 92 49 24 92 48 1b 6d  b6 db 6d b6 db 6d b6 db  |$.I$.H.m..m..m..|
000000b0  6d b6 db 6d b6 db 6d b6  db 6d b6 db 09 24 92 49  |m..m..m..m...$.I|
000000c0  24 92 49 24 92 49 24 92  49 24 92 49 24 92 49 24  |$.I$.I$.I$.I$.I$|
000000d0  92 48 12 49 24 92 49 24  92 49 24 92 49 24 92 49  |.H.I$.I$.I$.I$.I|
000000e0  24 92 49 24 92 49 24 80                           |$.I$.I$.|
000000e8

420 in.txt
232 in.txt.huff
```

5 Заключение

5.1 Условия применения алгоритма

Результаты тестирования показывают, что **динамическая** реализация алгоритма на произвольном наборе ASCII символов менее эффективна, чем на специализированных наборах. Таким образом, чистое применение алгоритма Хаффмана целесообразно только для решения узконаправленных задач. В остальных же случаях, рекомендуется использовать его в комбинации с другими алгоритмами (что и делается в большинстве распространенных алгоритмов сжатия).

5.2 Недостатки реализации

1. Размер блока с частотой статический, что увеличивает пространство мета информации на диске.

```c
#ifndef HTREE_H
#define HTREE_H

#include <stdint.h>

#include "pqueue.h"

struct hnode {
  uint8_t sym;

  size_t frq;
  size_t idx;

  uint64_t code[2];
  size_t len;

  struct hnode *lft, *rgt;
};

struct hnode *ht_new(struct pqueue*);
void ht_del(struct hnode*);

int ht_cmp(void*, void*);
void ht_incr(void*);
void ht_repl(void*, size_t);

#endif // HTREE
```

Listing 1: Интерфейс дерева Хаффмана

```
struct hnode *ht_new(struct pqueue *pq) {
  if (pq->size == 0) return NULL;
  if (pq->size == 1) {
    struct hnode *root = malloc(sizeof(struct hnode));
    root->sym = 255;
    root->lft = pq_extr(pq);
    extend(root->lft, 0);
    return root;
  }

  while (pq->size != 1) {
    struct hnode *a = pq_extr(pq);
    struct hnode *b = pq_extr(pq);
    struct hnode *r = malloc(sizeof(struct hnode));

    r->sym = 255;
    r->frq = a->frq + b->frq;
    r->lft = a;
    r->rgt = b;

    extend(a, 0);
    extend(b, 1);
    pq_ins(pq, r);
  }

  return pq_extr(pq);
}
```

Listing 2: Подпрограмма построения дерева Хаффмана

```
int encode(FILE *src, FILE *dst) {
  struct meta *meta = meta_new();

  for (uint8_t c = getc(src); !feof(src); c = getc(src))
    if (meta->ss[c].sym == 255) {
      meta->ss[c].sym = c;
      meta->ss[c].frq = 1;
      pq_ins(meta->pq, &meta->ss[c]);
    } else
      pq_incr(meta->pq, meta->ss[c].idx);

  meta->pqs = meta->pq->size;
  meta->cpy = malloc(sizeof(struct hnode) * meta->pqs);

  for (int i = 0; i < meta->pqs; ++i)
    meta->cpy[i] = meta->pq->heap[i];

  struct hnode *root = ht_new(meta->pq);

  fseek(src, 0, SEEK_SET);
  fseek(dst, meta_size(meta), SEEK_SET);

  uint8_t cache = 0;

  for (uint8_t c = getc(src); !feof(src); c = getc(src))
    for (int i = 0; i < meta->ss[c].len; ++i) {
      uint64_t blk = i < 64 ? meta->ss[c].code[0] : meta->ss[c].code[1];
      size_t pos = i < 64 ? i : i - 64;

      if (blk & (1UL << (63 - pos)))
        cache = cache | (0b00000001 << (7 - meta->lbs));

      if (++(meta->lbs) == 8) {
        putc(cache, dst);
        meta->lbs = 0;
        cache = 0;
      }
    }

  if (meta->lbs != 0) putc(cache, dst);
  else meta->lbs = 8;

  fseek(dst, 0, SEEK_SET);
  meta_put(dst, meta);
```

Listing 3: Подпрограмма кодирования

```
int decode(FILE *src, FILE *dst) {
  struct meta *meta = meta_new();

  if (meta_get(src, meta) != 0) return -1;

  struct hnode *root = ht_new(meta->pq);
  struct hnode *next = root;

  uint8_t b = getc(src);

  while (!feof(src)) {
    uint8_t n = getc(src);

    if (ferror(src)) return -1;

    for (int j = 0; j < (feof(src) ? meta->lbs : 8); ++j) {
      if (b & (1 << (7 - j))) next = next->rgt;
      else next = next->lft;

      if (next == NULL) return -1;
      if (next->sym != 255) {
        putc(next->sym, dst);
        next = root;
      }
    }

    b = n;
  }

  ht_del(root);
  meta_del(meta);

  return 0;
}
```

Listing 4: Подпрограмма декодирования