杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

程序设计实践报告

基于 C 语言的稀疏矩阵快速运算

卓越学院
计算机科学英才班
周靖凯

目录

1 问题描述 **1**

2 稀疏矩阵的表示 **2**

3 稀疏矩阵的基本操作 **3**

3.1 加法 3

3.2 减法 3

3.3 乘法 3

4 性能分析 **4**

4.1 时间复杂度 4

4.2 空间复杂度 4

5 讨论与改进 **4**

5.1 时间复杂度优化 4

5.2 稀疏矩阵的预处理 5

5.3 并行计算 5

6 小结 **6**

A 附录 **7**

A.1 完整测试代码 7

1 问题描述

稀疏矩阵（Sparse Matrix）是一种常见的数学结构，它在各个领域的计算和数据处理中发挥着重要作用。与传统的密集矩阵不同，稀疏矩阵具有大部分元素为零的特点，这意味着它们在内存和计算资源的利用上具有巨大的潜力。稀疏矩阵常出现在诸如图像处理、线性代数、有限元分析、自然语言处理和网络分析等应用中。

本报告旨在介绍一种基于十字链表的稀疏矩阵实现方法，该方法允许创建、储存和执行稀疏矩阵的基本运算，包括加法、减法、乘法和转置等操作。我们将展示如何使用这种方法来处理稀疏矩阵，并提供示例代码和实验结果，以说明其实际应用和性能。

稀疏矩阵的处理对于解决许多实际问题至关重要。通过减少不必要的计算和内存开销，稀疏矩阵的优化方法可以大幅提高计算效率，减少存储需求。本报告将深入探讨如何实现这些优化，并介绍如何应对可能出现的错误情况，以确保稳定的矩阵操作。

在国家的科技创新和工程建设中，十字链表稀疏矩阵运算被广泛应用。例如，在计算机算法的优化中，十字链表稀疏矩阵运算可以用于提高计算机的运算速度和效率，从而更好地支持国家的科技创新和工程建设。此外，十字链表稀疏矩阵运算还可以用于数据挖掘、图像处理、人工智能等领域，为国家的发展做出了重要的贡献。

2 稀疏矩阵的表示

稀疏矩阵是一种特殊的矩阵，其特点是大部分元素为零。与密集矩阵不同，稀疏矩阵在实际应用中可以节省大量的内存空间和计算资源。为了高效地表示和处理稀疏矩阵，我们需要采用合适的数据结构和表示方法。

在本实现中，我们使用了基于十字链表的数据结构来表示稀疏矩阵。这种数据结构以一种紧凑的方式存储非零元素，并使用链表来跟踪每行和每列的非零元素。以下是我们使用的关键数据结构：

稀疏矩阵中的一个非零元素用一个 `struct Position` 表示。它包括元素的行坐标、列坐标和值。

稀疏矩阵中的每个非零元素都由一个 `struct Node` 的节点表示。这个节点包括一个指向 `struct Triple` 的指针，以及指向下一个节点和右边节点的指针。这种结构使得我们可以轻松地遍历矩阵的非零元素。

每一行和每一列都由一个 `struct Line` 结构来表示，其中包括指向头部和尾部节点的指针。这种结构使得我们可以快速访问每行和每列的非零元素。

最终，整个稀疏矩阵由一个 `struct Matrix` 结构表示，包括矩阵的行数、列数以及行和列的链表。这个结构将所有的行和列连接在一起，提供了对整个矩阵的全局访问。

通过使用这种基于十字链表的表示方法，我们可以高效地存储和操作稀疏矩阵，减少内存开销并提高运算效率。在接下来的部分，我们将详细介绍如何使用这些数据结构来创建、操作和执行各种稀疏矩阵运算。

3 稀疏矩阵的基本操作

稀疏矩阵的基本操作包括加法、减法、乘法和转置等。在本节中，我们将详细讨论如何使用基于十字链表的数据结构执行这些操作。

3.1 加法

稀疏矩阵的加法操作是将两个稀疏矩阵相加，得到一个新的稀疏矩阵。我们首先检查两个矩阵的尺寸是否匹配，即它们的行数和列数是否相同。如果不匹配，则加法操作无法执行。

然后，我们遍历两个矩阵的非零元素，将相同位置的元素相加，并将结果存储在新矩阵中。需要注意的是，如果两个元素相加后的结果为零，我们可以选择不将其存储，以节省内存空间。

3.2 减法

稀疏矩阵的减法操作类似于加法，不同之处在于我们需要计算一个矩阵减去另一个矩阵的结果。减法操作也需要检查尺寸匹配，并按照相同的方式遍历非零元素进行减法操作。

3.3 乘法

稀疏矩阵的乘法操作是其中最复杂的操作之一。乘法操作需要检查第一个矩阵的列数是否等于第二个矩阵的行数。然后，我们遍历两个矩阵的非零元素，并执行乘法运算，将结果存储在新矩阵中。

转置

矩阵的转置操作将矩阵的行和列交换，得到一个新的矩阵。在稀疏矩阵中，这可以通过重新排列节点并更新链表的方式来实现。

4 性能分析

4.1 时间复杂度

稀疏矩阵的基本操作的时间复杂度取决于非零元素的数量和矩阵的尺寸。以下是基本操作的时间复杂度分析：

- 假设稀疏矩阵 A 和 B 的非零元素数量分别为 c_1 和 c_2，那么加法操作的时间复杂度为 $O(c_1 + c_2)$。
- 减法操作与加法操作具有相同的时间复杂度，也为 $O(c_1 + c_2)$。
- 假设稀疏矩阵 A 的非零元素数量为 n_1，B 的非零元素数量为 n_2，A 的列数为 m_1，B 的行数为 n_1。乘法操作的最差时间复杂度为 $O(n_1 \cdot m_1 \cdot n_2)$，但是稀疏矩阵的时间复杂度会大大降低。
- 转置操作的时间复杂度取决于稀疏矩阵的非零元素数量 c 和矩阵的尺寸。时间复杂度为 $O(c)$。

4.2 空间复杂度

令稀疏矩阵的行数为 n，列数为 m，非零元素的数量为 c。空间复杂度为 $O(n + m + c)$。

5 讨论与改进

5.1 时间复杂度优化

- 加法和减法：由于这两个操作的时间复杂度与非零元素的数量成正比，可以考虑使用多线程或并行计算来加速操作，特别是在处理大型稀疏矩阵时。并行计算可以有效地利用多核处理器的性能。
- 乘法：乘法操作的时间复杂度较高，因此可以考虑使用稀疏矩阵乘法的优化算法，如 Strassen 算法或分块矩阵乘法。这些算法可以减少计算量，降低时间复杂度。

5.2 稀疏矩阵的预处理

在执行稀疏矩阵操作之前，可以考虑进行一些预处理操作，例如矩阵分解、缩放或填充零元素，以减小非零元素的数量或简化操作。这可以帮助加速后续的操作。

5.3 并行计算

对于大型稀疏矩阵，可以考虑使用分布式计算或 GPU 加速来处理操作。这将允许我们利用大规模计算集群或 GPU 的并行计算能力，以加速操作。

6 小结

稀疏矩阵是在各种科学和工程领域中广泛应用的数据结构，它们能够有效地存储和处理具有大量零元素的数据。在本文中，我们讨论了稀疏矩阵的表示和基本操作，并进行了性能分析。

通过基于十字链表的数据结构，我们成功地实现了稀疏矩阵的创建、输入输出、加法、减法、乘法和转置等操作。这些操作为稀疏矩阵的处理提供了基础，使我们能够在不浪费大量内存的情况下高效地执行各种数学运算。

在性能分析中，我们了解到稀疏矩阵的操作时间复杂度和空间复杂度的特点。我们提出了一些优化策略，包括多线程计算、稀疏矩阵乘法的优化算法、稀疏矩阵表示的改进方法以及预处理策略，以提高程序的性能。

在实际应用中，根据具体问题的需求和数据的特性，我们可以选择适当的稀疏矩阵表示方法和优化策略，以实现高效的数据处理。稀疏矩阵的处理在图计算、线性代数、机器学习等领域都有广泛的应用，因此对其性能的优化具有重要的意义。

A 附录

A.1 完整测试代码

matrix.h

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct Triple
{
    int x, y;
    int val;
};

struct Node
{
    struct Triple position;
    struct Node *down, *right;
};

struct Line
{
    struct Node *head, *tail;
};

struct Matrix
{
    int n, m;
    struct Line *row, *column;
};

```

```
28 int cmp(const void *a, const void *b)
29 {
30     if (((struct Triple *)a)->x != ((struct Triple *)b)->x
       )
31         return (((struct Triple *)a)->x - ((struct Triple
           *)b)->x);
32     else
33         return (((struct Triple *)a)->y - ((struct Triple
           *)b)->y);
34 }
35
36 struct Matrix init_zero_matrix(const int n, const int m)
37 {
38     struct Matrix mat;
39     mat.n = n, mat.m = m;
40     mat.row = (struct Line *)malloc(sizeof(struct Line) *
       n);
41     mat.column = (struct Line *)malloc(sizeof(struct Line)
        * m);
42     for (int i = 0; i < n; i++)
43         mat.row[i].head = mat.row[i].tail = NULL;
44     for (int i = 0; i < m; i++)
45         mat.column[i].head = mat.column[i].tail = NULL;
46     return mat;
47 }
48
49 void destroy_matrix(struct Matrix *mat)
50 {
51     for (int i = 0; i < mat->n; i++)
52         for (struct Node *p = mat->row[i].head, *np; p !=
           NULL; p = np)
```

```
53          {
54              np = p->right;
55              free(p);
56          }
57      free(mat->row);
58      free(mat->column);
59      return;
60  }
61
62  struct Matrix create_matrix(const struct Triple a[], const
        int size, const int n, const int m)
63  {
64      if (n <= 0 || m <= 0)
65      {
66          fprintf(stderr, "The number of rows and columns
                must be greater than 0");
67          exit(EXIT_FAILURE);
68      }
69      struct Triple *b = (struct Triple *)malloc(sizeof(
            struct Triple) * size);
70      for (int i = 0; i < size; i++)
71          b[i].x = a[i].x - 1, b[i].y = a[i].y - 1, b[i].val
                = a[i].val;
72      for (int i = 0; i < size; i++)
73      {
74          if (b[i].x < 0 || b[i].x >= n || b[i].y < 0 || b[i
                ].y >= m)
75          {
76              fprintf(stderr, "Illegal element");
77              free(b);
78              exit(EXIT_FAILURE);
```

```
79          }
80          if (i > 1 && b[i].x == b[i - 1].x && b[i].y == b[i
                - 1].y)
81          {
82              fprintf(stderr, "Duplicate nodes");
83              free(b);
84              exit(EXIT_FAILURE);
85          }
86      }
87      qsort(b, size, sizeof(struct Triple), cmp);
88      struct Matrix mat = init_zero_matrix(n, m);
89      for (int i = 0; i < size; i++)
90      {
91          if (b[i].val == 0)
92              continue;
93          struct Node *p = (struct Node *)malloc(sizeof(
                struct Node));
94          p->position = b[i];
95          p->down = p->right = NULL;
96          if (mat.row[b[i].x].tail == NULL)
97              mat.row[b[i].x].head = mat.row[b[i].x].tail =
                    p;
98          else
99              mat.row[b[i].x].tail->right = p;
100         mat.row[b[i].x].tail = p;
101         if (mat.column[b[i].y].tail == NULL)
102             mat.column[b[i].y].head = mat.column[b[i].y].
                    tail = p;
103         else
104             mat.column[b[i].y].tail->down = p;
105         mat.column[b[i].y].tail = p;
```

```
106     }
107     free(b);
108     return mat;
109 }
110 
111 void print_matrix(const struct Matrix mat)
112 {
113     for (int i = 0; i < mat.n; i++)
114         for (struct Node *p = mat.row[i].head; p != NULL;
               p = p->right)
115             printf("(%d,%d,%d)\n", p->position.x + 1, p->
                   position.y + 1, p->position.val);
116     return;
117 }
118 
119 void print_complete_matrix(const struct Matrix mat)
120 {
121     for (int i = 0; i < mat.n; i++)
122     {
123         if (mat.row[i].head == NULL)
124         {
125             for (int j = 0; j < mat.m; j++)
126             {
127                 printf("0");
128                 if (j + 1 < mat.m)
129                     printf(" ");
130                 else
131                     printf("\n");
132             }
133         }
134         else
```

```
135         {
136             if (0 < mat.row[i].head->position.y)
137             {
138                 for (int j = 0; j <= mat.row[i].head->
                        position.y - 1; j++)
139                     printf("0 ");
140             }
141             for (struct Node *p = mat.row[i].head; p !=
                    NULL; p = p->right)
142             {
143                 struct Node *np = p->right;
144                 printf("%d", p->position.val);
145                 if (p->position.y == mat.m - 1)
146                     printf("\n");
147                 else
148                     printf(" ");
149                 if (np != NULL)
150                 {
151                     for (int j = p->position.y + 1; j <=
                            np->position.y - 1; j++)
152                         printf("0 ");
153                 }
154                 else
155                 {
156                     for (int j = p->position.y + 1; j <
                            mat.m; j++)
157                     {
158                         printf("0");
159                         if (j + 1 < mat.m)
160                             printf(" ");
161                         else
```

```
                                printf("\n");
                        }
                    }
                }
            }
        }
    return;
}

struct Matrix add(const struct Matrix a, const struct
   Matrix b)
{
    if (a.n != b.n || a.m != b.m)
    {
        fprintf(stderr, "The two matrices being added must
            have the same number of rows and columns");
        exit(EXIT_FAILURE);
    }
    struct Matrix c = init_zero_matrix(a.n, a.m);
    for (int i = 0; i < a.n; i++)
    {
        struct Node *l = a.row[i].head, *r = b.row[i].head
           ;
        while (l != NULL || r != NULL)
        {
            struct Node *p = (struct Node *)malloc(sizeof(
               struct Node));
            p->down = p->right = NULL;
            if (r == NULL || l->position.y < r->position.y
               )
                p->position.x = l->position.x, p->position
```

```
                    .y = l->position.y, p->position.val = l
                    ->position.val, l = l->right;
188             else if (l == NULL || l->position.y > r->
                   position.y)
189                 p->position.x = r->position.x, p->position
                    .y = r->position.y, p->position.val = r
                    ->position.val, r = r->right;
190             else
191                 p->position.x = l->position.x, p->position
                    .y = r->position.y, p->position.val = l
                    ->position.val + r->position.val, l = l
                    ->right, r = r->right;
192             if (p->position.val == 0)
193             {
194                 free(p);
195                 continue;
196             }
197             if (c.row[p->position.x].tail == NULL)
198                 c.row[p->position.x].head = c.row[p->
                    position.x].tail = p;
199             else
200                 c.row[p->position.x].tail->right = p;
201             c.row[p->position.x].tail = p;
202             if (c.column[p->position.y].tail == NULL)
203                 c.column[p->position.y].head = c.column[p
                    ->position.y].tail = p;
204             else
205                 c.column[p->position.y].tail->down = p;
206             c.column[p->position.y].tail = p;
207         }
208     }
```

```
209     return c;
210 }
211
212 struct Matrix minus(const struct Matrix a, const struct
       Matrix b)
213 {
214     if (a.n != b.n || a.m != b.m)
215     {
216         fprintf(stderr, ”The two matrices being minused
               must have the same number of rows and columns”)
               ;
217         exit(EXIT_FAILURE);
218     }
219     struct Matrix c = init_zero_matrix(a.n, a.m);
220     for (int i = 0; i < a.n; i++)
221     {
222         struct Node *l = a.row[i].head, *r = b.row[i].head
               ;
223         while (l != NULL || r != NULL)
224         {
225             struct Node *p = (struct Node *)malloc(sizeof(
                   struct Node));
226             p->down = p->right = NULL;
227             if (r == NULL || l->position.y < r->position.y
                   )
228                 p->position.x = l->position.x, p->position
                       .y = l->position.y, p->position.val = l
                       ->position.val, l = l->right;
229             else if (l == NULL || l->position.y > r->
                   position.y)
230                 p->position.x = r->position.x, p->position
```

```
                    .y = r->position.y, p->position.val = -
                    r->position.val, r = r->right;
231             else
232                 p->position.x = l->position.x, p->position
                    .y = r->position.y, p->position.val = l
                    ->position.val - r->position.val, l = l
                    ->right, r = r->right;
233             if (p->position.val == 0)
234             {
235                 free(p);
236                 continue;
237             }
238             if (c.row[p->position.x].tail == NULL)
239                 c.row[p->position.x].head = c.row[p->
                    position.x].tail = p;
240             else
241                 c.row[p->position.x].tail->right = p;
242             c.row[p->position.x].tail = p;
243             if (c.column[p->position.y].tail == NULL)
244                 c.column[p->position.y].head = c.column[p
                    ->position.y].tail = p;
245             else
246                 c.column[p->position.y].tail->down = p;
247             c.column[p->position.y].tail = p;
248         }
249     }
250     return c;
251 }
252
253 struct Matrix multiply(const struct Matrix a, const struct
    Matrix b)
```

```
254 {
255     if (a.m != b.n)
256     {
257         fprintf(stderr, ”The number of columns in the
                first matrix is not equal to the number of rows
                 in the second matrix”);
258         exit(EXIT_FAILURE);
259     }
260     struct Matrix c = init_zero_matrix(a.n, b.m);
261     for (int i = 0; i < a.n; i++)
262         for (struct Node *pa = a.row[i].head; pa != NULL;
                pa = pa->right)
263         {
264             int k = pa->position.y;
265             for (struct Node *pb = b.row[k].head; pb !=
                   NULL; pb = pb->right)
266             {
267                 int j = pb->position.y;
268                 if (c.column[j].tail != NULL && c.column[j
                       ].tail->position.x == i)
269                     c.column[j].tail->position.val += pa->
                           position.val * pb->position.val;
270                 else
271                 {
272                     struct Node *pc = (struct Node *)
                           malloc(sizeof(struct Node));
273                     pc->down = pc->right = NULL;
274                     pc->position.x = i, pc->position.y = j
                           , pc->position.val = pa->position.
                           val * pb->position.val;
275                     if (pc->position.val == 0)
```

```
276                     {
277                         free(pc);
278                         continue;
279                     }
280                     if (c.column[pc->position.y].tail ==
                           NULL)
281                         c.column[pc->position.y].head = c.
                               column[pc->position.y].tail =
                               pc;
282                     else
283                         c.column[pc->position.y].tail->
                               down = pc;
284                     c.column[pc->position.y].tail = pc;
285                 }
286             }
287         }
288     for (int j = 0; j < b.m; j++)
289     {
290         while (c.column[j].head != NULL && c.column[j].
               head->position.val == 0)
291         {
292             struct Node *np = c.column[j].head->down;
293             free(c.column[j].head);
294             c.column[j].head = np;
295         }
296         for (struct Node *p = c.column[j].head; p != NULL;
               p = p->down)
297         {
298             for (struct Node *np = p->down; np != NULL &&
                   np->position.val == 0; np = p->down)
299             {
```

```
300                 p->down = np->down;
301                 free(np);
302             }
303         }
304         for (struct Node *p = c.column[j].head; p != NULL;
                p = p->down)
305         {
306             if (c.row[p->position.x].tail == NULL)
307                 c.row[p->position.x].head = c.row[p->
                       position.x].tail = p;
308             else
309                 c.row[p->position.x].tail->right = p;
310             c.row[p->position.x].tail = p;
311         }
312     }
313     return c;
314 }
315
316 struct Matrix transpose(const struct Matrix a)
317 {
318     struct Matrix t = init_zero_matrix(a.m, a.n);
319     for (int i = 0; i < a.n; i++)
320         for (struct Node *p = a.row[i].head; p != NULL; p
              = p->right)
321         {
322             struct Node *q = (struct Node *)malloc(sizeof(
                   struct Node));
323             q->down = q->right = NULL;
324             q->position.x = p->position.y, q->position.y =
                    p->position.x, q->position.val = p->
                   position.val;
```

```
325             if (t.row[q->position.x].tail == NULL)
326                 t.row[q->position.x].head = t.row[q->
                       position.x].tail = q;
327             else
328                 t.row[q->position.x].tail->right = q;
329             t.row[q->position.x].tail = q;
330             if (t.column[q->position.y].tail == NULL)
331                 t.column[q->position.y].head = t.column[q
                       ->position.y].tail = q;
332             else
333                 t.column[q->position.y].tail->down = q;
334             t.column[q->position.y].tail = q;
335         }
336     return t;
337 }
```