

# 情報工学実験II

## テーマ03

### グラフ・ネットワークプログラム

令和5年07月06日

イマム カイリ ルビス

学籍番号：214071

## 目 次

<b>1</b>	<b>概要</b>	<b>3</b>
1.1	グラフ理論とは . . . . .	3
1.2	スタックとは . . . . .	3
1.3	キューとは . . . . .	3
1.3.1	リングバッファによるキュー . . . . .	3
1.4	実行環境 . . . . .	4
<b>2</b>	<b>深さ優先検索と幅優先検索を用いて検索</b>	<b>4</b>
2.1	深さ優先検索 . . . . .	4
2.1.1	深さ優先検索のプログラム . . . . .	4
2.1.2	深さ優先検索のプログラムの動作 . . . . .	9
2.2	幅優先検索 . . . . .	9
2.2.1	幅優先検索のプログラム . . . . .	9

# 1 概要

## 1.1 グラフ理論とは

数学においてグラフ理論とは、グラフを研究する学問であり、グラフはオブジェクト間の対関係をモデル化するために用いられる数学的構造である。グラフを構成するためには、点（節点またはノードとも呼ばれる）と辺（枝またはエッジとも呼ばれる）が必要である [?]. グラフ理論には、辺が方向を持っているかどうかによって分れている

**有向グラフ**：辺の方向が決まっている一方向性のグラフである。

**無向グラフ**：辺が特定の方向を持たず、双方向性を持つグラフである。

本実験では、使用したグラフはすべて無向グラフである。更に、

## 1.2 スタックとは

スタックは、データを一時的に蓄えるためのデータ構造の一つ。データの出し入れは**後入れ先出し**（*LIFO / Last In First Out*）で行われる。すなわち、最後に入れられたデータが最初に取り出される [?].

なお、スタックにデータを入れる操作を**プッシュ**（*push*）と呼び、スタックからデータを取り出す操作を**ポップ**（*pop*）と呼びます [?].

しかし本実験では、実際のスタック機能を模倣するため、ノード数分の大きさを持つ配列を使ってスタックデータ構造を作った。

## 1.3 キューとは

キューは、データを一時的に蓄えるための基本的なデータ構造の一つである。最初に入れられたデータが最初に取り出されるという**先入れ先出し**（*FIFO / First In First Out*）の機構である。 [?]

なお、キューにデータを追加する操作を**エンキュー**（*enqueue*）と呼び、データを取り出す操作を**デキュー**（*dequeue*）と呼ぶ。また、データが取り出される側を**先頭**（*front*）と呼び、データが押し込まれる側を**末尾**（*rear*）と呼ぶ [?].

しかし本実験では、実際のキュー機能を模倣するため、ノード数分の大きさを持つ配列を使ってスタックデータ構造を作った。

### 1.3.1 リングバッファによるキュー

リングバッファとは、配列の末尾が先頭につながっているとみなすデータ構造である [?]. エンキューとデキューを行うと *front* と *rear* の値は変化する。

## 1.4 実行環境

本実験で使用する実行環境：

- プロセッサ：AMD Ryzen 5 5600X
- メモリー：16.0 GB
- OS：Windows 11 Pro
- コンパイラ：gcc

## 2 深さ優先検索と幅優先検索を用いて検索

### 2.1 深さ優先検索

深さ優先探索は、木やグラフのデータ構造を探索するアルゴリズムである。このアルゴリズムは、根（始点）から開始し、バックトラックする前に各辺に沿って可能な限り探索する。

指定した辺に沿ってこれまでに発見されたノードを追跡し、グラフのバックトラックに役立てるために、スタックが必要となる。

#### 2.1.1 深さ優先検索のプログラム

以下は深さ優先検索のプログラムである。

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <ctype.h>
4
5 int countLines(FILE *in) {
6     char c;
7     int count = 0;
8
9     do {
10         c = fgetc(in);
11         if(c == '\n') count++;
12     } while (c != EOF);
13
14     rewind(in);
15
16     return count;
17 }
18
19 void print (int *target, int size) {
20     for (int i = 0; i < size; i++) {
21         printf("[%d] : %d\n", i + 1, *target);
22         target++;
23     }
24 }
25
26 void stackInit(int *stack, int size) {
27     int *p = stack;
28     for (int i = 0; i < size; i++) {
29         *p = -1;
30         p++;
31     }
```

```

32 }
33
34 int stackSearchEmpty(int *stack, int size) {
35     int *p = stack;
36     int offset = 0;
37     while (*p >= 0 && offset < size) {
38         p++;
39         offset++;
40     }
41     return offset;
42 }
43
44 void stackPush (int *stack, int size, int data) {
45     int *p = stack;
46     if(stackSearchEmpty(stack, size) < size) {
47         int offset = stackSearchEmpty(stack, size);
48         p = p + offset;
49         *p = data;
50     } else {
51         printf("Stack Push: Stack full, data %d not stored\n", data);
52     }
53 }
54
55 void stackPop(int *stack, int size) {
56     int *p = stack;
57     if(stackSearchEmpty(stack, size) <= size) {
58         int offset = stackSearchEmpty(stack, size) - 1;
59         if (offset >= 0) {
60             p = p + offset;
61             *p = -1;
62         } else {
63             printf("Stack Pop: Stack Already Empty\n");
64         }
65     } else {
66         printf("Stack Pop: Stack Full\n");
67     }
68 }
69
70 int stackTop(int *stack, int size) {
71     int *p = stack;
72     if(stackSearchEmpty(stack, size) - 1 < size) {
73         int offset = stackSearchEmpty(stack, size) - 1;
74         if (offset >= 0) {
75             p += offset;
76         } else {
77             printf("Stack Top: Stack is Empty\n");
78         }
79     } else {
80         printf("Stack Top: Stack Full\n");
81     }
82     return *p;
83 }
84
85 int stackIsEmpty(int *stack) {
86     int *p = stack;
87     if (*p == -1) return 1;
88     else return 0;
89 }
90
91 void storeArray(FILE *in, int *dst) {
92     int ch;
93     int i = 0;
94
95     while ((ch = fgetc(in)) != EOF) {
96         if(isdigit(ch)) {
97             *dst = ch - '0';
98             dst++;

```

```

99     }
100 }
101 }
102
103 // row disini mulai dari 1
104 void printArrayAt(int *data, int size, int row) {
105     if (row <= size) {
106         row--;
107         int *p = data + row*size;
108         for (int i = 0; i < size; i++) {
109             printf("%d ", p[i]);
110         }
111         printf("\n");
112     } else {
113         printf("Print Array: Row exceeded limit\n");
114     }
115 }
116
117 // return first index of 1, row disini mulai dari 1
118 int searchRow(int *data, int size, int row, int *flag) {
119     int offset = 0;
120     if (row <= size) {
121         row--;
122         int *p = data + row*size;
123         for (int i = 0; i < size; i++) {
124             if(*p == 1 && flag[offset] == 0) {
125                 flag[offset] = 1; //update flag
126                 break;
127             }
128             else offset++;
129             p++;
130         }
131         return ++offset;
132     } else {
133         printf("Search Row: Row exceedede limit");
134         return -1;
135     }
136 }
137
138 void printResult(int *result, int size) {
139     for (int i = 0; i < (size-1) * 2; i+=2) {
140         printf("%d: %d -> %d \n", (i+1)/2 + 1 , result[i], result[i+1]);
141     }
142 }
143
144 int searchRight(int *result, int size, int target) {
145     int last = 2*(size-1)-1;
146     int index = last;
147     while (result[index] != target && index >= 0) {
148         index -= 2;
149         // printf("index = %d\n", index);
150     }
151     return index;
152 }
153
154 int sameLeft(int *result, int size) {
155     int current = result[0];
156     int count = 1;
157     for (int i = 0; i < (size-1) * 2; i += 2) {
158         if (result[i] != current) {
159             count ++;
160             current = result[i];
161         }
162     }
163     return count;
164 }
165

```

```

166 void countHeight(int *result, int size) {
167     int last = 2*(size-1)-1;
168     // int index = 2*(size-1)-1;
169     int height = 1;
170     for (int i = last; i >= 0; i -= 2) {
171         int index = i;
172         int dummy = 1;
173         // printf("%d\n", result[i]);
174         while(result[index - 1] != 1) {
175             index = searchRight(result, size, result[index - 1]);
176             dummy += 1;
177         }
178         if (dummy > height) height = dummy;
179     }
180     printf("Height = %d\n", height);
181 }
182
183 void countLeaf(int *result, int size) {
184     int last = 2*(size-1)-1;
185     int *flag = (int *)malloc(size * sizeof(int));
186
187     for (int i = 0; i <= last; i += 2) {
188         flag[result[i] - 1] = 1;
189     }
190
191     int count = 0;
192     for (int i = 0; i < size; i++) {
193         count += flag[i];
194     }
195
196     printf("Leaf = %d\n", size - count);
197     free(flag);
198 }
199
200
201 int getMax(int *target, int size) {
202     int max = target[0];
203     for (int i = 1; i < size; i++) {
204         if(max < target[i]) max = target[i];
205     }
206     return max;
207 }
208
209 void countChild(int *result, int size) {
210     int current = result[0];
211     int aSize = sameLeft(result, size);
212     int *child = (int *)malloc(aSize * sizeof(int));
213     int *c = child;
214     for (int i = 0; i < (size-1) * 2; i += 2) {
215         if (result[i] == current) {
216             *c = *c + 1;
217         } else {
218             current = result[i];
219             c++;
220             *c = *c + 1;
221         }
222     }
223     // print(child, aSize);
224     printf("Max Child = %d\n", getMax(child, aSize));
225     free(child);
226 }
227
228 int checkFlag(int *flag, int size) {
229     // int *p = flag;
230     int count = 0;
231     for (int i = 0; i < size; i++) {
232         if (flag[i] == 0) break;

```

```

233         // p++;
234         count++;
235     }
236     return count;
237 }
238
239 int main(int argc, char **argv) {
240     char *file;
241     if (argc == 2) {
242         file = argv[1];
243     } else {
244         printf("Set File Name!");
245     }
246
247     const char *dir = "data/";
248     char fileName[50];
249     sprintf(fileName, "%s%s", dir, file);
250
251     FILE *fp = fopen(fileName, "r");
252
253     if (!fp) {
254         perror("fopen");
255         return 1;
256     }
257
258     int size = countLines(fp);
259
260     int *stack = (int *)malloc(size * sizeof(int));
261     stackInit(stack, size);
262
263     int *data = (int *)malloc(size * size * sizeof(int));
264     storeArray(fp, data);
265
266     int *flag = (int *)malloc(size * sizeof(int)); // 1 visited, 0 not yet
267     flag[0] = 1;
268
269     int *result = (int *)malloc(2 * size * sizeof(int));
270     int *r = result;
271
272     int currentNode = 1;
273     stackPush(stack, size, currentNode);
274
275     int count = 1;
276
277     while(!stackIsEmpty(stack)) {
278         do {
279             if (stackIsEmpty(stack)) {
280                 if (checkFlag(flag, size) != size) {
281                     currentNode = checkFlag(flag, size) + 1;
282                     flag[currentNode - 1] = 1;
283                     stackPush(stack, size, currentNode);
284                     count++;
285                 } else {
286                     break;
287                 }
288             }
289             currentNode = searchRow(data, size, stackTop(stack, size), flag);
290             if (currentNode > size) stackPop(stack, size);
291         } while(currentNode > size);
292
293         if (stackIsEmpty(stack)) break;
294         *r = stackTop(stack, size);
295         r++;
296         *r = currentNode;
297         r++;
298         stackPush(stack, size, currentNode);
299     }

```



```

300
301     printResult(result, size);
302     countHeight(result, size);
303     countLeaf(result, size);
304     countChild(result, size);
305
306     printf("Connected Component = %d\n", count);
307
308     free(stack);
309     free(data);
310     free(flag);
311     free(result);
312     fclose(fp);
313 }

```

### 2.1.2 深さ優先検索のプログラムの動作

訪問したすべての点はスタックにプッシュされ、その点から先に行けない場合はスタックトップがポップされる。深さ優先検索のプログラムの主な流れは、以下の通りである：

1. 根をスタックにプッシュする。
2. スタックトップのデータを現在点になるようにピークする。
3. 現在の点に接続している最も低い点に進む。
4. 全ての点を訪れるまで繰り返す。

## 2.2 幅優先検索

幅優先探索は、木やグラフのデータ構造を探索するアルゴリズムである。このアルゴリズムは、根（始点）から開始し、各点に隣接している点を訪問する。

キューは、訪問されたがまだ探索されていない子ノードを追跡するために必要である。

### 2.2.1 幅優先検索のプログラム

以下は深さ優先検索のプログラムである。

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <ctype.h>
4
5  int countLines(FILE *in) {
6      char c;
7      int count = 0;
8
9      do {
10         c = fgetc(in);
11         if(c == '\n') count++;
12     } while (c != EOF);
13
14     rewind(in);
15
16     return count;
17 }

```

```

18
19 void print (int *target, int size) {
20     for (int i = 0; i < size; i++) {
21         printf("[%d] : %d\n", i + 1, target[i]);
22     }
23 }
24
25 void queueInit(int *stack, int size) {
26     int *p = stack;
27     for (int i = 0; i < size; i++) {
28         *p = -1;
29         p++;
30     }
31 }
32
33 int queueIsEmpty(int *queue, int qHead, int qTail) {
34     if (qHead == (qTail - 1) && queue[qHead] == -1) {
35         return 1;
36     } else return 0;
37 }
38
39 int queueIsFull(int *queue, int qHead, int qTail) {
40     if (qHead == (qTail - 1) && queue[qHead] != -1) {
41         return 1;
42     } else return 0;
43 }
44
45 void queueIndex(int *index, int size) {
46     if(*index >= size) *index = 0;
47     else if(*index == -1) *index = size - 1;
48 }
49
50 void queueEnqueue(int *queue, int size, int *qHead, int *qTail, int data) {
51     if(!queueIsFull(queue, *qHead, *qTail)) {
52         int index = *qTail - 1;
53         queueIndex(&index, size);
54         queue[index] = data;
55         *qTail = *qTail + 1;
56         queueIndex(qTail, size);
57     } else {
58         printf("Enqueue: Queue is already Full, %d not stored\n", data);
59     }
60 }
61
62 void queueDequeue(int *queue, int size, int *qHead, int *qTail) {
63     if(!queueIsEmpty(queue, *qHead, *qTail)) {
64         if (queue[*qHead] != -1) {
65             queue[*qHead] = -1;
66             *qHead = *qHead + 1;
67             queueIndex(qHead, size);
68         }
69     } else {
70         printf("Dequeue: Queue already empty\n");
71     }
72 }
73
74 int queueTop(int *queue, int qHead) {
75     return queue[qHead];
76 }
77
78 void storeArray(FILE *in, int *dst) {
79     int ch;
80     int i = 0;
81
82     while ((ch = fgetc(in)) != EOF) {
83         if(isdigit(ch)) {
84

```

```

85         *dst = ch - '0';
86         dst++;
87     }
88 }
89 }
90
91 // row disini mulai dari 1
92 void printArrayAt(int *data, int size, int row) {
93     if (row <= size) {
94         row--;
95         int *p = data + row*size;
96         for (int i = 0; i < size; i++) {
97             printf("%d ", p[i]);
98         }
99         printf("\n");
100     } else {
101         printf("Print Array: Row exceeded limit\n");
102     }
103 }
104
105 // return first index of 1, row disini mulai dari 1
106 int searchRow(int *data, int size, int row, int *flag) {
107     int offset = 0;
108     if (row <= size) {
109         row--;
110         int *p = data + row*size;
111         for (int i = 0; i < size; i++) {
112             if(*p == 1 && flag[offset] == 0) {
113                 flag[offset] = 1; //update flag
114                 break;
115             }
116             else offset++;
117             p++;
118         }
119         return ++offset;
120     } else {
121         printf("Search Row: Row exceedede limit");
122         return -1;
123     }
124 }
125
126 void printResult(int *result, int size) {
127     for (int i = 0; i < (size-1) * 2; i+=2) {
128         printf("%d: %d -> %d \n", (i+1)/2 + 1 , result[i], result[i+1]);
129     }
130 }
131
132 int searchRight(int *result, int size, int target) {
133     int last = 2*(size-1)-1;
134     int index = last;
135     while (result[index] != target && index >= 0) {
136         index -= 2;
137         // printf("index = %d\n", index);
138     }
139     return index;
140 }
141
142 int sameLeft(int *result, int size) {
143     int current = result[0];
144     int count = 1;
145     for (int i = 0; i < (size-1) * 2; i += 2) {
146         if (result[i] != current) {
147             count ++;
148             current = result[i];
149         }
150     }
151     return count;

```

```

152 }
153
154 void countHeight(int *result, int size) {
155     int last = 2*(size-1)-1;
156     // int index = 2*(size-1)-1;
157     int height = 1;
158     for (int i = last; i >= 0; i -= 2) {
159         int index = i;
160         int dummy = 1;
161         // printf("%d\n", result[i]);
162         while(result[index - 1] != 1) {
163             index = searchRight(result, size, result[index - 1]);
164             dummy += 1;
165         }
166         if (dummy > height) height = dummy;
167     }
168     printf("Height = %d\n", height);
169 }
170
171 void countLeaf(int *result, int size) {
172     int last = 2*(size-1)-1;
173     int *flag = (int *)malloc(size * sizeof(int));
174
175     for (int i = 0; i <= last; i += 2) {
176         flag[result[i] - 1] = 1;
177     }
178
179     int count = 0;
180     for (int i = 0; i < size; i++) {
181         count += flag[i];
182     }
183
184     printf("Leaf = %d\n", size - count);
185     free(flag);
186 }
187
188 int getMax(int *target, int size) {
189     int max = target[0];
190     for (int i = 1; i < size; i++) {
191         if(max < target[i]) max = target[i];
192     }
193     return max;
194 }
195
196 void countChild(int *result, int size) {
197     int current = result[0];
198     int aSize = sameLeft(result, size);
199     int *child = (int *)malloc(aSize * sizeof(int));
200     int *c = child;
201     for (int i = 0; i < (size-1) * 2; i += 2) {
202         if (result[i] == current) {
203             *c = *c + 1;
204         } else {
205             current = result[i];
206             c++;
207             *c = *c + 1;
208         }
209     }
210     // print(child, aSize);
211     printf("Max Child = %d\n", getMax(child, aSize));
212     free(child);
213 }
214
215 int checkFlag(int *flag, int size) {
216     // int *p = flag;
217     int count = 0;
218     for (int i = 0; i < size; i++) {

```

```

219         if (flag[i] == 0) break;
220         // p++;
221         count++;
222     }
223     return count;
224 }
225
226 int main(int argc, char **argv) {
227     char *file;
228     if (argc == 2) {
229         file = argv[1];
230     } else {
231         printf("Set File Name!");
232     }
233
234     const char *dir = "data/";
235     char fileName[50];
236     sprintf(fileName, "%s%s", dir, file);
237
238     FILE *fp = fopen(fileName, "r");
239
240     if (!fp) {
241         perror("fopen");
242         return 1;
243     }
244
245     int size = countLines(fp);
246
247     int *queue = (int *)malloc(size * sizeof(int));
248     int qHead = 0;
249     int qTail = 1;
250     queueInit(queue, size);
251
252     int *data = (int *)malloc(size * size * sizeof(int));
253     storeArray(fp, data);
254
255     int *flag = (int *)malloc(size * sizeof(int)); // 1 visited, 0 not yet
256     flag[0] = 1;
257
258     int *result = (int *)malloc(2 * size * sizeof(int));
259     int *r = result;
260
261     int currentNode = 1;
262     queueEnqueue(queue, size, &qHead, &qTail, currentNode);
263
264     int count = 1;
265
266     while (!queueIsEmpty(queue, qHead, qTail)) {
267         do {
268             if (queueIsEmpty(queue, qHead, qTail)) {
269                 if (checkFlag(flag, size) != size) {
270                     currentNode = checkFlag(flag, size) + 1;
271                     flag[currentNode - 1] = 1;
272                     queueEnqueue(queue, size, &qHead, &qTail, currentNode);
273                     count++;
274                 } else {
275                     break;
276                 }
277             }
278             currentNode = searchRow(data, size, queueTop(queue, qHead), flag);
279             if (currentNode > size) queueDequeue(queue, size, &qHead, &qTail);
280         } while (currentNode > size);
281
282         if (queueIsEmpty(queue, qHead, qTail)) break;
283         *r = queueTop(queue, qHead);
284         r++;
285         *r = currentNode;

```

```

286         r++;
287         queueEnqueue(queue, size, &qHead, &qTail, currentNode);
288     }
289
290     printResult(result, size);
291     countHeight(result, size);
292     countLeaf(result, size);
293     countChild(result, size);
294
295     printf("Connected Component = %d\n", count);
296
297     free(queue);
298     free(data);
299     free(flag);
300     free(result);
301     fclose(fp);
302 }

```