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

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、データを一時的にた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

本実験で使用される実行環境：

- 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.

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

int countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);

    return count;
}

void print (int *target, int size) {
    for (int i = 0; i < size; i++) {
        printf("[%d] : %d\n", i + 1, *target);
        target++;
    }
}

void stackInit(int *stack, int size) {
    int *p = stack;
    for (int i = 0; i < size; i++) {
        *p = -1;
        p++;
    }
```

```
}

int stackSearchEmpty(int *stack, int size) {
    int *p = stack;
    int offset = 0;
    while (*p >= 0 && offset < size) {
        p++;
        offset++;
    }
    return offset;
}

void stackPush (int *stack, int size, int data) {
    int *p = stack;
    if(stackSearchEmpty(stack, size) < size) {
        int offset = stackSearchEmpty(stack, size);
        p = p + offset;
        *p = data;
    } else {
        printf("Stack Push: Stack full, data %d not stored\n", data);
    }
}

void stackPop(int *stack, int size) {
    int *p = stack;
    if(stackSearchEmpty(stack, size) <= size) {
        int offset = stackSearchEmpty(stack, size) - 1;
        if (offset >= 0) {
            p = p + offset;
            *p = -1;
        } else {
            printf("Stack Pop: Stack Already Empty\n");
        }
    } else {
        printf("Stack Pop: Stack Full\n");
    }
}

int stackTop(int *stack, int size) {
    int *p = stack;
    if(stackSearchEmpty(stack, size) - 1 < size) {
        int offset = stackSearchEmpty(stack, size) - 1;
        if (offset >= 0) {
            p += offset;
        } else {
            printf("Stack Top: Stack is Empty\n");
        }
    } else {
        printf("Stack Top: Stack Full\n");
    }
    return *p;
}

int stackIsEmpty(int *stack) {
    int *p = stack;
    if (*p == -1) return 1;
    else return 0;
}

void storeArray(FILE *in, int *dst) {
    int ch;
    int i = 0;

    while ((ch = fgetc(in)) != EOF) {
        if(isdigit(ch)) {
            *dst = ch - '0';
            dst++;
```

```
        }
    }
}

// row disini mulai dari 1
void printArrayAt(int *data, int size, int row) {
    if (row <= size) {
        row--;
        int *p = data + row*size;
        for (int i = 0; i < size; i++) {
            printf("%d ", p[i]);
        }
        printf("\n");
    } else {
        printf("Print Array: Row exceeded limit\n");
    }
}

// return first index of 1, row disini mulai dari 1
int searchRow(int *data, int size, int row, int *flag) {
    int offset = 0;
    if (row <= size) {
        row--;
        int *p = data + row*size;
        for (int i = 0; i < size; i++) {
            if(*p == 1 && flag[offset] == 0) {
                flag[offset] = 1; //update flag
                break;
            }
            else offset++;
            p++;
        }
        return ++offset;
    } else {
        printf("Search Row: Row exceedede limit");
        return -1;
    }
}

void printResult(int *result, int size) {
    for (int i = 0; i < (size-1) * 2; i+=2) {
        printf("%d: %d -> %d \n",(i+1)/2 + 1 , result[i], result[i+1]);
    }
}

int searchRight(int *result, int size, int target) {
    int last = 2*(size-1)-1;
    int index = last;
    while (result[index] != target && index >= 0) {
        index -= 2;
        // printf("index = %d\n", index);
    }
    return index;
}

int sameLeft(int *result, int size) {
    int current = result[0];
    int count = 1;
    for (int i = 0; i < (size-1) * 2; i += 2) {
        if (result[i] != current) {
            count ++;
            current = result[i];
        }
    }
    return count;
}

```

```c
void countHeight(int *result, int size) {
    int last = 2*(size-1)-1;
    // int index = 2*(size-1)-1;
    int height = 1;
    for (int i = last; i >= 0; i -= 2) {
        int index = i;
        int dummy = 1;
        // printf("%d\n", result[i]);
        while(result[index - 1] != 1) {
            index = searchRight(result, size, result[index - 1]);
            dummy += 1;
        }
        if (dummy > height) height = dummy;
    }
    printf("Height = %d\n", height);
}

void countLeaf(int *result, int size) {
    int last = 2*(size-1)-1;
    int *flag = (int *)malloc(size * sizeof(int));

    for (int i = 0; i <= last; i += 2) {
        flag[result[i] - 1] = 1;
    }

    int count = 0;
    for (int i = 0; i < size; i++) {
        count += flag[i];
    }

    printf("Leaf = %d\n", size - count);
    free(flag);
}


int getMax(int *target, int size) {
    int max = target[0];
    for (int i = 1; i < size; i++) {
        if(max < target[i]) max = target[i];
    }
    return max;
}

void countChild(int *result, int size) {
    int current = result[0];
    int aSize = sameLeft(result, size);
    int *child = (int *)malloc(aSize * sizeof(int));
    int *c = child;
    for (int i = 0; i < (size-1) * 2; i += 2) {
        if (result[i] == current) {
            *c = *c + 1;
        } else {
            current = result[i];
            c++;
            *c = *c + 1;
        }
    }
    // print(child, aSize);
    printf("Max Child = %d\n", getMax(child, aSize));
    free(child);
}

int checkFlag(int *flag, int size) {
    // int *p = flag;
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (flag[i] == 0) break;
```

```
        // p++;
        count++;
    }
    return count;
}

int main(int argc, char **argv) {
    char *file;
    if (argc == 2) {
        file = argv[1];
    } else {
        printf("Set File Name!");
    }

    const char *dir = "data/";
    char fileName[50];
    sprintf(fileName, "%s%s", dir, file);

    FILE *fp = fopen(fileName, "r");

    if (!fp) {
        perror("fopen");
        return 1;
    }

    int size = countLines(fp);

    int *stack = (int *)malloc(size * sizeof(int));
    stackInit(stack, size);

    int *data = (int *)malloc(size * size * sizeof(int));
    storeArray(fp, data);

    int *flag = (int *)malloc(size * sizeof(int)); // 1 visited, 0 not yet
    flag[0] = 1;

    int *result = (int *)malloc(2 * size * sizeof(int));
    int *r = result;

    int currentNode = 1;
    stackPush(stack, size, currentNode);

    int count = 1;

    while(!stackIsEmpty(stack)) {
        do {
            if (stackIsEmpty(stack)) {
                if(checkFlag(flag, size) != size) {
                    currentNode = checkFlag(flag, size) + 1;
                    flag[currentNode - 1] = 1;
                    stackPush(stack, size, currentNode);
                    count++;
                } else {
                    break;
                }
            }
            currentNode = searchRow(data, size, stackTop(stack, size), flag);
            if (currentNode > size) stackPop(stack, size);
        } while(currentNode > size);

        if (stackIsEmpty(stack))break;
        *r = stackTop(stack, size);
        r++;
        *r = currentNode;
        r++;
        stackPush(stack, size, currentNode);
    }
```

```
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
46 void queueIndex(int *index, int size) {
47     if(*index >= size) *index = 0;
48     else if(*index == -1) *index = size - 1;
49 }
50
51 void queueEnqueue(int *queue, int size, int *qHead, int *qTail, int data) {
52     if(!queueIsFull(queue, *qHead, *qTail)) {
53         int index = *qTail - 1;
54         queueIndex(&index, size);
55         queue[index] = data;
56         *qTail = *qTail + 1;
57         queueIndex(qTail, size);
58     } else {
59         printf("Enqueue: Queue is already Full, %d not stored\n", data);
60     }
61 }
62
63 void queueDequeue(int *queue, int size, int *qHead, int *qTail) {
64     if(!queueIsEmpty(queue, *qHead, *qTail)) {
65         if (queue[*qHead] != -1) {
66             queue[*qHead] = -1;
67             *qHead = *qHead + 1;
68             queueIndex(qHead, size);
69         }
70     } else {
71         printf("Dequeue: Queue already empty\n");
72     }
73 }
74
75 int queueTop(int *queue, int qHead) {
76     return queue[qHead];
77 }
78
79 void storeArray(FILE *in, int *dst) {
80     int ch;
81     int i = 0;
82
83     while ((ch = fgetc(in)) != EOF) {
84         if(isdigit(ch)) {
```

```
            *dst = ch - '0';
            dst++;
        }
    }
}

// row disini mulai dari 1
void printArrayAt(int *data, int size, int row) {
    if (row <= size) {
        row--;
        int *p = data + row*size;
        for (int i = 0; i < size; i++) {
            printf("%d ", p[i]);
        }
        printf("\n");
    } else {
        printf("Print Array: Row exceeded limit\n");
    }
}

// return first index of 1, row disini mulai dari 1
int searchRow(int *data, int size, int row, int *flag) {
    int offset = 0;
    if (row <= size) {
        row--;
        int *p = data + row*size;
        for (int i = 0; i < size; i++) {
            if(*p == 1 && flag[offset] == 0) {
                flag[offset] = 1; //update flag
                break;
            }
            else offset++;
            p++;
        }
        return ++offset;
    } else {
        printf("Search Row: Row exceedede limit");
        return -1;
    }
}

void printResult(int *result, int size) {
    for (int i = 0; i < (size-1) * 2; i+=2) {
        printf("%d: %d -> %d \n",(i+1)/2 + 1 , result[i], result[i+1]);
    }
}

int searchRight(int *result, int size, int target) {
    int last = 2*(size-1)-1;
    int index = last;
    while (result[index] != target && index >= 0) {
        index -= 2;
        // printf("index = %d\n", index);
    }
    return index;
}

int sameLeft(int *result, int size) {
    int current = result[0];
    int count = 1;
    for (int i = 0; i < (size-1) * 2; i += 2) {
        if (result[i] != current) {
            count ++;
            current = result[i];
        }
    }
    return count;
```

```
}

void countHeight(int *result, int size) {
    int last = 2*(size-1)-1;
    // int index = 2*(size-1)-1;
    int height = 1;
    for (int i = last; i >= 0; i -= 2) {
        int index = i;
        int dummy = 1;
        // printf("%d\n", result[i]);
        while(result[index - 1] != 1) {
            index = searchRight(result, size, result[index - 1]);
            dummy += 1;
        }
        if (dummy > height) height = dummy;
    }
    printf("Height = %d\n", height);
}

void countLeaf(int *result, int size) {
    int last = 2*(size-1)-1;
    int *flag = (int *)malloc(size * sizeof(int));

    for (int i = 0; i <= last; i += 2) {
        flag[result[i] - 1] = 1;
    }

    int count = 0;
    for (int i = 0; i < size; i++) {
        count += flag[i];
    }

    printf("Leaf = %d\n", size - count);
    free(flag);
}

int getMax(int *target, int size) {
    int max = target[0];
    for (int i = 1; i < size; i++) {
        if(max < target[i]) max = target[i];
    }
    return max;
}

void countChild(int *result, int size) {
    int current = result[0];
    int aSize = sameLeft(result, size);
    int *child = (int *)malloc(aSize * sizeof(int));
    int *c = child;
    for (int i = 0; i < (size-1) * 2; i += 2) {
        if (result[i] == current) {
            *c = *c + 1;
        } else {
            current = result[i];
            c++;
            *c = *c + 1;
        }
    }
    // print(child, aSize);
    printf("Max Child = %d\n", getMax(child, aSize));
    free(child);
}

int checkFlag(int *flag, int size) {
    // int *p = flag;
    int count = 0;
    for (int i = 0; i < size; i++) {
```

```
        if (flag[i] == 0) break;
        // p++;
        count++;
    }
    return count;
}

int main(int argc, char **argv) {
    char *file;
    if (argc == 2) {
        file = argv[1];
    } else {
        printf("Set File Name!");
    }

    const char *dir = "data/";
    char fileName[50];
    sprintf(fileName, "%s%s", dir, file);

    FILE *fp = fopen(fileName, "r");

    if (!fp) {
        perror("fopen");
        return 1;
    }

    int size = countLines(fp);

    int *queue = (int *)malloc(size * sizeof(int));
    int qHead = 0;
    int qTail = 1;
    queueInit(queue, size);

    int *data = (int *)malloc(size * size * sizeof(int));
    storeArray(fp, data);

    int *flag = (int *)malloc(size * sizeof(int)); // 1 visited, 0 not yet
    flag[0] = 1;

    int *result = (int *)malloc(2 * size * sizeof(int));
    int *r = result;

    int currentNode = 1;
    queueEnqueue(queue, size, &qHead, &qTail, currentNode);

    int count = 1;

    while (!queueIsEmpty(queue, qHead, qTail)) {
        do {
            if (queueIsEmpty(queue, qHead, qTail)) {
                if (checkFlag(flag, size) != size) {
                    currentNode = checkFlag(flag, size) + 1;
                    flag[currentNode - 1] = 1;
                    queueEnqueue(queue, size, &qHead, &qTail, currentNode);
                    count++;
                } else {
                    break;
                }
            }
            currentNode = searchRow(data, size, queueTop(queue, qHead), flag);
            if (currentNode > size) queueDequeue(queue, size, &qHead, &qTail);
        } while (currentNode > size);

        if (queueIsEmpty(queue, qHead, qTail)) break;
        *r = queueTop(queue, qHead);
        r++;
        *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 }
```