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 [1]．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 [2].

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 [2].

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．[2]

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 [2].

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 [2]．エンキューとデキューを行うと *front* と *rear* の値は変化する．

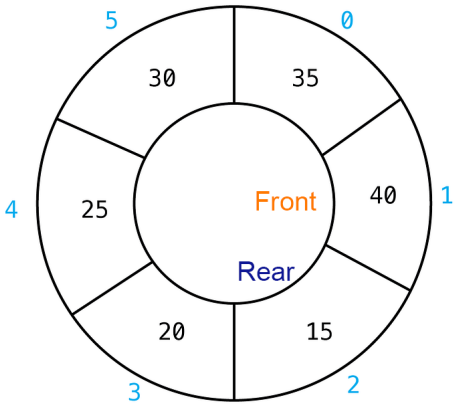

図 1: リングバッファによるキュー

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++;
        }
    }
}

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");
    }
}

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;
                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;
    }
    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 height = 1;
    for (int i = last; i >= 0; i -= 2) {
        int index = i;
        int dummy = 1;
        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;
        }
    }
    printf("Max Child = %d\n", getMax(child, aSize));
    free(child);
}

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

    // printResult(result, size);
    // countHeight(result, size);
    // countLeaf(result, size);
    // countChild(result, size);

    printf("Connected Component = %d\n", count);

    free(stack);
    free(data);
    free(flag);
    free(result);
    fclose(fp);

    return 0;
}
```

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. 現在の点に隣接している最も低い点に訪問する．
4. 現在点から行き場所はない場合，スタックからポップする．
5. 全ての点を訪問されるまで繰り返す．

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点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．

```
#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[i]);
    }
}

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

int queueIsEmpty(int *queue, int qHead, int qTail) {
    if (qHead == (qTail - 1) && queue[qHead] == -1) {
        return 1;
    } else return 0;
}

int queueIsFull(int *queue, int qHead, int qTail) {
    if (qHead == (qTail - 1) && queue[qHead] != -1) {
        return 1;
    } else return 0;
}


void queueIndex(int *index, int size) {
    if(*index >= size) *index = 0;
    else if(*index == -1) *index = size - 1;
}

void queueEnqueue(int *queue, int size, int *qHead, int *qTail, int data) {
    if(!queueIsFull(queue, *qHead, *qTail)) {
        int index = *qTail - 1;
        queueIndex(&index, size);
```

```
        queue[index] = data;
        *qTail = *qTail + 1;
        queueIndex(qTail, size);
    } else {
        printf("Enqueue: Queue is already Full, %d not stored\n", data);
    }
}

void queueDequeue(int *queue, int size, int *qHead, int *qTail) {
    if(!queueIsEmpty(queue, *qHead, *qTail)) {
        if (queue[*qHead] != -1) {
            queue[*qHead] = -1;
            *qHead = *qHead + 1;
            queueIndex(qHead, size);
        }
    } else {
        printf("Dequeue: Queue already empty\n");
    }
}

int queueTop(int *queue, int qHead) {
    return queue[qHead];
}

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

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

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");
    }
}

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;
                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;
    }
    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 height = 1;
    for (int i = last; i >= 0; i -= 2) {
        int index = i;
        int dummy = 1;
        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;
        }
    }
    printf("Max Child = %d\n", getMax(child, aSize));
    free(child);
}

int checkFlag(int *flag, int size) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (flag[i] == 0) break;
        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));
    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;
        r++;
        queueEnqueue(queue, size, &qHead, &qTail, currentNode);
    }

    // printResult(result, size);
    // countHeight(result, size);
    // countLeaf(result, size);
    // countChild(result, size);

    printf("Connected Component = %d\n", count);

    free(queue);
    free(data);
    free(flag);
    free(result);
    fclose(fp);
}
```

2.2.2 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キューの *rear* にエンキューされ，その点から先に行けない場合はキューの *front* からデキューされる．幅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. 根をキューにエンキューする．
2. キューの *front* のデータを現在点になるようにピークする．
3. 現在の点に隣接している全ての点を訪問する．
4. 現在点から行き場所はない場合，キューからデキューする．
5. 全ての点を訪れるまで繰り返す．

2.3　深さ優先検索と幅優先検索の結果

この問題では，検索対象となるデータが複数用意されている．両プログラムを実行した結果，下表のような結果が得られました．

表 1: 深さ優先検索結果

Data	Height	Leaf Count	Max Child
search_100d.dat	99	1	2
search_100s.dat	89	8	2
search_500d.dat	498	2	3
search_500s.dat	483	13	3
search_1000d.dat	999	1	2
search_1000s.dat	982	14	3

表 2: 幅優先検索結果

Data	Height	Leaf Count	Max Child
search_100d.dat	2	94	66
search_100s.dat	4	70	14
search_500d.dat	2	495	345
search_500s.dat	3	453	50
search_1000d.dat	2	993	607
search_1000s.dat	2	946	108

2.4　深さ優先検索と幅優先検索の考察

表（1）と表（2）を見ればわかるように，どちらのアルゴリズムも木を生成するが，その特性は異なる．すべてをまとめるために，この表の特徴を以下に記す．

表 3: 幅優先検索と幅優先検索の特性

特性	深さ優先検索	幅優先検索
高さ	高い	低い
葉の数	少ない	多い
子の数	少ない	多い

結論として，深さ優先検索は木が長くなるが，各点の子数は少なくなる．一方，幅優先検索は木が低くなりますが，各点の子数は多くなります．

3　連結成分数

連結成分とは，点の各対が辺で結ばれている部分グラフのことである．この問題では，与えられたデータがどれだけの連結成分を持つかを数えるというものである．

3.1　連結成分数のプログラム

この問題に対して，新しいプログラムはないので，プログラムのある部分だけに焦点を当てるつもりである．

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 main(int argc, char **argv) {

    ...

    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;
                }
            }

    ...
}
```

以下は幅優先検索のプログラムである．

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

...

int main(int argc, char **argv) {

    ...

    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;
                }
            }
```

```
    ...
}
```

3.2 連結成分数のプログラムの動作

検索問題でも同じプログラムを利用する．このデータには複数の連結要素があるので，スタックやキューが空になるが，この処理はまだ終わっていない．なので，未訪問の点を追加しなければならない．両アルゴリズムの主な流れは，以下の通りである：

1. データ構造が空になるまでプログラムを実行する
2. すべての点を訪問したかどうかをチェックする．
3. そうでない場合は，最下位の未訪問点をデータ構造に追加する．
4. すべての点を訪問するまで繰り返す

言い換えれば，連結成分数を数えるには，この処理が何回繰り返されたかを数えればいい．

3.3 連結成分数のプログラムの実行結果

これらのプログラムを実行した結果，以下のような結果が得られた．

- 深さ優先検索： 6つ連結成分
- 幅優先検索： 6つ連結成分

3.4 連結成分数の結果の考察

表からわかるように，どのようなアルゴリズムで連結成分をチェックしても結果は同じである．

4 最大クリーク問題

クリークとは部分グラフのことで，すべての点が部分グラフの他のどの点とも辺でつながっているのことである．言い換えれば，最大クリークはより大きなクリークの部分グラフではないクリークである．

4.1 最大クリークのプログラム

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++) {
        if(*target != 0) printf("%d ", i + 1);
        target++;
    }
    printf("\n");
}

// 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");
    }
}

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

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

void initP(int *target, int size) {
    for(int i = 1; i <= size; i++) {
        target[i - 1] = 1;
    }
}

void init(int *target, int size) {
    for (int i = 1; i <= size; i++) {
        target[i-1] = 0;
    }
}

void unionData(int *target, int *condition, int *data, int size, int row) {
    if (row <= size) {
        row--;
        int *t = target;
        int *c_1 = condition;
        int *c_2 = data + row * size;

        for (int i = 0; i < size; i++) {
```

```
            if (c_1[i] || c_2[i]) t[i] = 1;
            else t[i] = 0;
        }
    } else {
        printf("Union Data: Row exceeded limit\n");
    }
}

void intersectionData(int *target, int *condition, int *data, int size, int row)
    {
    if (row <= size) {
        row--;
        int *t = target;
        int *c_1 = condition;
        int *c_2 = data + row * size;

        for (int i = 0; i < size; i++) {
            if (c_1[i] && c_2[i]) t[i] = 1;
            else t[i] = 0;
        }
    } else {
        printf("Intersection Data: Row exceeded limit\n");
    }
}

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

void BronKerbosch(int *R, int *P, int *X, int *data, int size) {
    int *r = R;
    int *p = P;
    int *x = X;
    int *d = data;

    if(isEmpty(p, size) && isEmpty(x, size)) {
        printf("Result\n");
        print(r, size);
        printf("\n");
    } else {
        for (int i = 0; i < size; i++) {
            if (P[i] == 1) {
                int *v = (int *)malloc(size * sizeof(int));
                int *arg1 = (int *)malloc(size * sizeof(int));
                int *arg2 = (int *)malloc(size * sizeof(int));
                int *arg3 = (int *)malloc(size * sizeof(int));
                init(v, size);
                init(arg1, size);
                init(arg2, size);
                init(arg3, size);

                v[i] = 1;
                unionData(arg1, r, v, size, 1);
                intersectionData(arg2, p, d, size, i+1);
                intersectionData(arg3, x, d, size, i+1);

                BronKerbosch(arg1, arg2, arg3, d, size);

                P[i] = 0;
                unionData(x, x, v, size, 1);

                free(v);
                free(arg1);
                free(arg2);
```

```
                free(arg3);
            }
        }
    }
}

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

    int *R = (int *)malloc(size * sizeof(int));
    int *P = (int *)malloc(size * sizeof(int));
    int *X = (int *)malloc(size * sizeof(int));
    init(R, size);
    initP(P, size);
    init(X, size);

    BronKerbosch(R, P, X, data, size);

    free(R);
    free(P);
    free(X);
    free(data);
    fclose(fp);
}
```

4.2 最大クリークのプログラムの動作

この問題のプログラムは，実際にはピボットなしのブロンケルボッシュアルゴリズムの実装です．しかし，このコードを作るために修正した点がいくつかある．このアルゴリズムは集合の共通部分と和集合を利用するので，すべてのデータを実際の数値として格納する代わりに，各データが配列の添え字として扱った．つまり，ある配列の値 0 の場合，その値のデータは存在しない，1 の場合はその値のデータは存在する．

ブロンケルボッシュアルゴリズムは以下の通りである．

```
algorithm BronKerbosch1(R, P, X) is
   if P and X are both empty then
       report R as a maximal clique
   for each vertex v in P do
       BronKerbosch1(R ⋃ v, P ⋂ N(v), X ⋂ N(v))
       P := P \ v
       X := X ⋃ v
```

4.3 最大クリークの結果

この問題では，いくつかのデータを用意した．このプログラムをすべてのデータに対して実行結果は次のようにである．

Data	Number of Maximal Clique	Execution Time [s]
clique_30.dat	182	0.009
clique_50.dat	966	0.026
clique_70.dat	4276	0.123
clique_100.dat	18138	1.358

4.4 最大クリークの考察

点数が多ければ多いほど，最大クリークも多くなり，その数を数えるのに時間がかかる．

そのため，各反復においてピボットを選択することで，このアルゴリズムを改善することができる．ピボットは，アルゴリズムが行う再帰的呼び出しの回数を最小化するために選択され，ピボットを行わないバージョンのアルゴリズムと比較して，実行時間の節約は大きくなる [3].

5 発表者の感想

Ntar ini mah

参考文献

[1] https://en.wikipedia.org/wiki/Graph_theory（参照 2023-07-25）

[2] 柴田望洋，C 言語で学ぶアルゴリズムとでーた構造，SB Creative，2017.

[3] Cazals, F. & Karande, C., ”A note on the problem of reporting maximal cliques”, Theoretical Computer Science, 2008