コンピュータネットワーク　レポート

- 学籍番号：2264088
- 氏名：河原畑 宏次
- 所属：情報工学EP

課題1　ダイクストラ法に基づいて最短経路問題を解くプログラムの作成

ソースファイル

ソースファイル：`example_NodePath.txt`

```
9 6
0 1 1
0 2 4
1 2 2
1 3 4
1 4 3
2 3 2
3 4 2
3 5 4
4 5 1
```

ソースファイル：`Dijkstra.c`

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>


// ダイクストラ法で計算した結果を保存する構造体
typedef struct {
    int node;       // ノード番号
    int cost;       // コスト
    int parent;     // 親ノード番号
    int isFixed;    // 確定したノードなら1。そうでないなら0
} NODEINFO;


/* プロトタイプ宣言 */
```

```c
// コストテーブルを作成する関数
void makeCostTable(int nodeNum, int costTable[nodeNum][nodeNum], int pathNum, FIL

// リザルトテーブルを作成する関数
void makeResultTable(int nodeNum, NODEINFO resultTable[nodeNum], int startNode);

// ダイクストラ法を実行する関数
void exeDijkstra(int nodeNum, int costTable[nodeNum][nodeNum], int startNode, int

// 結果（最短経路）を表示する関数
void printResult(NODEINFO resultTable[], int startNode, int dstNode);

// リザルトテーブルを表示する関数（デバッグ用）
void printResultTable(int nodeNum, NODEINFO resultTable[nodeNum]);
// ----- プロトタイプ宣言ここまで -----


int main(void) {
    // ファイルの読み込み
    char fileName[] = "example_NodePath.txt";  // ファイル名
    FILE *fp = NULL;                           // ファイルポインタ
    if ((fp = fopen(fileName, "r")) == NULL) {
        printf("%s : ファイルを開けませんでした\n", fileName);
        exit(-1);
    }


    // パスの数とノードの数を入力
    int pathNum = -1;  // パスの数
    int nodeNum = -1;  // ノードの数
    fscanf(fp, "%d %d", &pathNum, &nodeNum);


    // スタートノードとゴールノードを入力
    int startNode = -1;  // スタートノード
    int endNode = -1;    // ゴールノード
    do {
        printf("スタートノードとゴールノードを入力してください\n");
        printf("A: 0, B: 1, C: 2, ...\n");
        printf("スタート: ");
        scanf("%d", &startNode);
        printf("ゴール: ");
        scanf("%d", &endNode);
        printf("\n");
    } while (startNode < 0 || startNode >= nodeNum || endNode < 0 || endNode >= n

    // コストテーブル（経路表）を作成
    int costTable[nodeNum][nodeNum];  // コストテーブル（経路表）
    makeCostTable(nodeNum, costTable, pathNum, fp);


    // ダイクストラ法の結果を保存するリザルトテーブルを作成
    NODEINFO resultTable[nodeNum];
    makeResultTable(nodeNum, resultTable, startNode);
```

```
    // ダイクストラ法を実行しリザルトテーブルを作成
    exeDijkstra(nodeNum, costTable, startNode, endNode, resultTable);


    // リザルトテーブルの表示（デバッグ）
    // printResultTable(nodeNum, resultTable);


    // 結果（最短経路と総コスト）の表示
    printResult(resultTable, startNode, endNode);
    printf("\n総コスト: %d\n", resultTable[endNode].cost);

    return 0;
}



// コストテーブル（経路表）を作成する関数
void makeCostTable(int nodeNum, int costTable[nodeNum][nodeNum], int pathNum, FIL
    // コストテーブルの初期化
    for (int i = 0; i < nodeNum; i++)
        for (int j = 0; j < nodeNum; j++) costTable[i][j] = -1;

    // コストテーブル（経路表）の作成
    for (int i = 0; i < pathNum; i++) {
        // 1行（出発点 到着点 かかるコスト）ごと読み取り
        int srcNode = -1, dstNode = -1, cost = -1;
        fscanf(fp, "%d %d %d", &srcNode, &dstNode, &cost);

        // 表にコストを記入。出発点と到着点が逆の場合も記入する
        costTable[srcNode][dstNode] = cost;
        costTable[dstNode][srcNode] = cost;
    }
}


// リザルトテーブルを作成する関数
void makeResultTable(int nodeNum, NODEINFO resultTable[nodeNum], int startNode) {
    for (int i = 0; i < nodeNum; i++) {
        resultTable[i].node = i;
        resultTable[i].cost = INT_MAX;  // 最初のコストは無限大
        resultTable[i].isFixed = 0;

        if (i == startNode) {                   // スタートノードの場合
            resultTable[i].parent = startNode;  // 親は自分自身
            resultTable[i].cost = 0;            // スタートノードのコストは0
            resultTable[i].isFixed = 1;         // スタートノードは確定
        } else {
            resultTable[i].parent = -1;
        }
    }
}
```

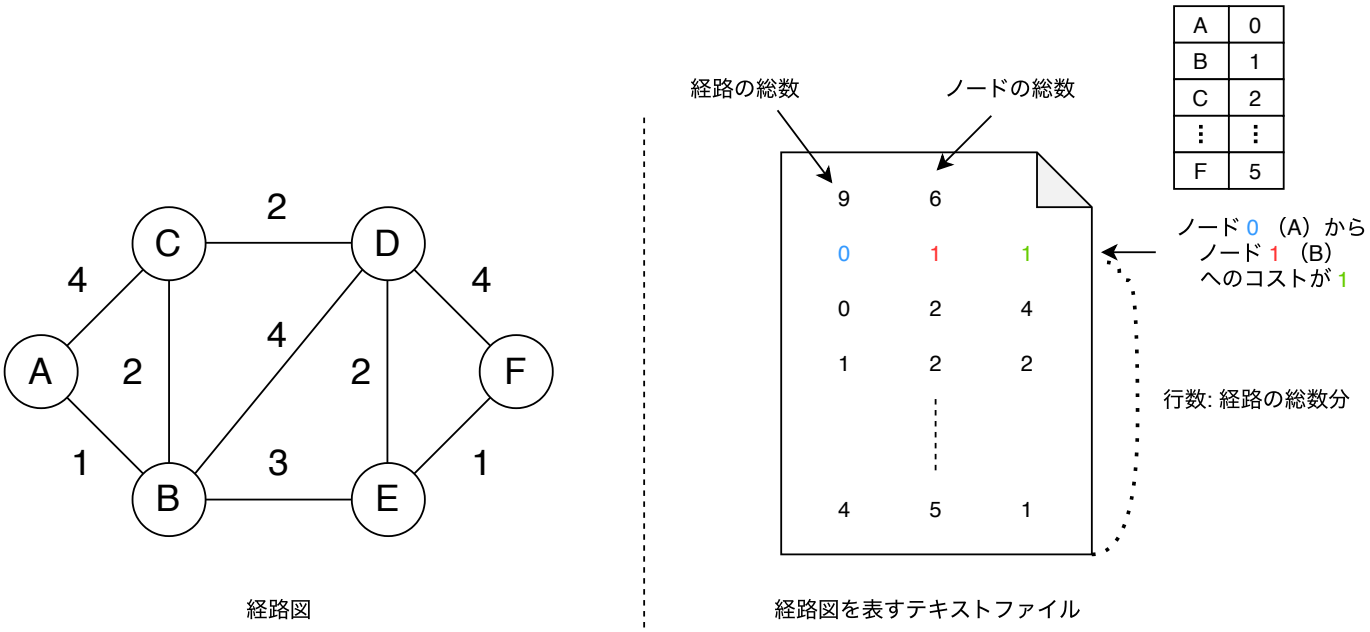
```
// ダイクストラ法を実行する関数
void exeDijkstra(int nodeNum, int costTable[nodeNum][nodeNum], int startNode, int
    int srcNode = startNode;    // 出発ノード
    do {
        int minimumCostNode = -1;   // 確定していない中で最小のコストのノード
        int minimumCost = INT_MAX;  // 最小のコスト

        for (int i = 0; i < nodeNum; i++) {
            // ノードiまでのコスト = ノードsrcからノードiまでのコスト + スタートノードからノ-
            int costToNode_i = costTable[srcNode][i] + resultTable[srcNode].cost;

            // コストテーブルのコストが0より大きく（経路がある）、現在のリザルトコストより小さ
            if (costTable[srcNode][i] > 0 && costToNode_i < resultTable[i].cost)
                resultTable[i].cost = costToNode_i;
                resultTable[i].parent = srcNode;
            }

            // 確定していないノードの中でコストが最小のノードを検索
            if (!resultTable[i].isFixed && resultTable[i].cost < minimumCost) {
                minimumCost = resultTable[i].cost;
                minimumCostNode = i;
            }
        }

        // 確定していないノードの中でコストが最小のノードを確定させて次の出発ノードに選択
        resultTable[minimumCostNode].isFixed = 1;
        srcNode = minimumCostNode;

    } while (!resultTable[endNode].isFixed);  // ゴールノードが確定すると終了
}


// 結果（最短経路）を表示する関数
void printResult(NODEINFO resultTable[], int startNode, int dstNode) {
    // 親ノードがスタートノードと一致しない場合
    if(resultTable[dstNode].parent != startNode) {
        printResult(resultTable, startNode, resultTable[dstNode].parent);
    }
    // 親ノードがスタートノードである場合はそのノードはスタートノードなのでノード名を出力（再帰処
    else {
        printf("%c", 'A' + startNode);
    }
    // 現在のノードから次に訪れるノードを出力
    printf(" -> %c", 'A' + resultTable[dstNode].node);
}


// ダイクストラ法を実行した後のリザルトテーブルを表示する関数（デバッグ用）
void printResultTable(int nodeNum, NODEINFO resultTable[nodeNum]) {
    // リザルトテーブルを表示
    printf("ノード | コスト | 親ノード\n");
    for (int i = 0; i < nodeNum; i++) {
```

```
        printf("   %c   |   %2d   |    %c    \n", 'A' + resultTable[i].node,
               resultTable[i].cost, 'A' + resultTable[i].parent);
    }
}
```

プログラムの説明

プログラムの概要

ダイクストラ法に基づいて最短経路問題を解くプログラムを作成した。ただし、グラフが次のように与えられた際、そのグラフをもとに次のようなパラメータを持つデータ形式（txtファイル）でグラフを表現する。



プログラムは主に次の手順で処理を行う。

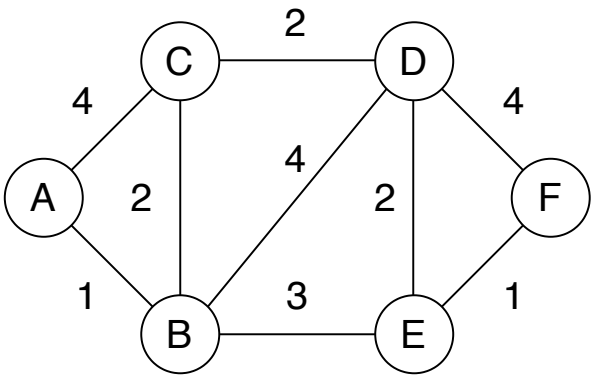
1. 経路データのテキストファイルの読み込み
2. スタートノードとゴールノードのユーザーによる入力受付
3. コストテーブル（経路とそのコストを保存する表）の作成
4. リザルトテーブル（ダイクストラ法の結果を保存する表）の作成
5. ダイクストラ法の実行
6. 結果（最短経路と総コスト）の表示

ここでは、上記の手順3, 4, 5, 6について説明を行う。また説明において、手順1, 2において取得した次のパラメータを用いる。

パラメータ	説明
`pathNum`	経路の数
`nodeNum`	ノードの数
`startNNode`	スタートノード
`endNNode`	ゴールノード

コストテーブルについて

コストテーブルは経路とそのコストをデータファイルから読み込み保存する表のことである。具体的には次の図中の表を二次元配列`int costTable[nodeNum][nodeNum]`で表現する。ただし、表の中で経路が存在しない要素については初期値`-1`を与えた。また、この表の作成は関数`makeCostTable()`が行う。



経路図

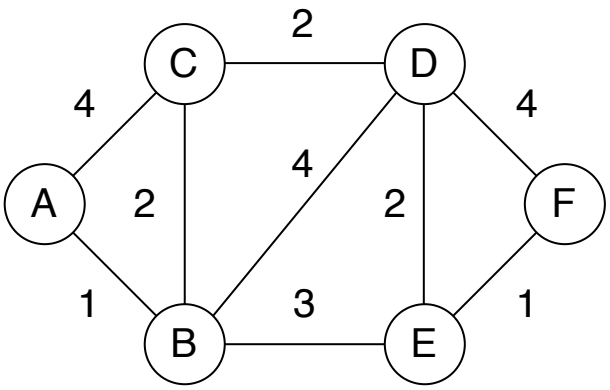
到着ノード

出発ノード	0	1	2	3	4	5
0		1	4			
1	1		2	4	3	
2	4	2		2		
3		4	2		2	4
4		3		2		1
5				4	1	

int costTable [nodeNum][nodeNum]

リザルトテーブルについて

リザルトテーブルは、ダイクストラ法を実行した結果を保存する表のことである。具体的には次の図中の表を構造体配列で表現する。このリザルトテーブルの作成には、関数`makeResultTable()`で行うが、スタートノードについては初期値として、親を自分自身のノード番号、コストを0、確定フラグを1に設定する。下記の表は、スタートノードをA（0）、ゴールノードをF（5）に設定した際にダイクストラ法を実施した場合の途中結果を示す。ただしプログラムにおいて、コストの∞は定数`INT_MAX`で、親ノードの未定は`-1`で表現した。



経路図

ノード	コスト	親ノード	確定フラグ
0	0	0	1
1	1	0	1
2	3	1	0
3	5	1	0
4	4	1	0
5	∞	---	0

NODEINFO resultTable [nodeNum]

```
// ダイクストラ砲で計算した結果を保存する構造体
typedef struct {
    int node;       // ノード番号
    int cost;       // コスト
    int parent;     // 親ノード番号
    int isFixed;    // 確定したノードなら1。そうでないなら0
} NODEINFO;

NODEINFO resultTable[nodeNum];
```

ダイクストラ法の実行について

用意したコストテーブルやリザルトテーブルを用いてダイクストラ法を実際に行う処理は、関数 `exeDijkstra()` に記述した。この関数では次のようなアルゴリズムに基づいてダイクストラ法を実行する。

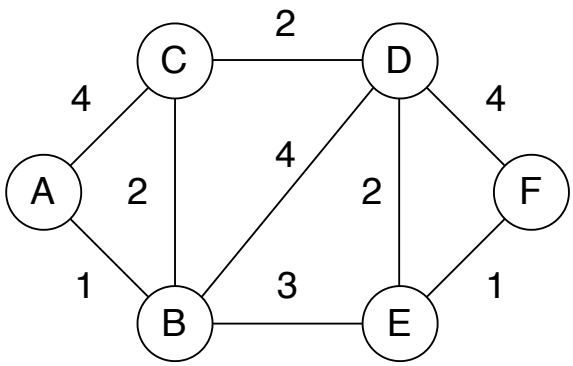
1. スタートノードを出発ノードとして設定する
2. リザルトテーブルを用いて、スタートノードから出発ノードまでのコストを求める
3. コストテーブルを用いて、出発ノードから各ノードまでのコストを求める
4. 2, 3で読み取ったコストの和（スタートノードから出発ノードを通り次のノードに行くまでにかかるコスト）を求める
5. 4で得たコストとがリザルトテーブルのコストより小さければリザルトテーブルのコストを更新する
6. リザルトテーブルの確定していないノードの中でコストが最小のノードを確定させて次の出発ノードに選択する
7. 1～6をゴールノードが確定するまで繰り返す

ここで、２のスタートノードから出発ノードまでのコストに関しては、リザルトテーブルを用いて簡単に計算できる。なぜなら、出発ノードは必ず確定ノードであるため、リザルトテーブルのコストの欄に、スタートノードからの最小コストが記載されているからである。また、５の処理については、コストテーブルのコストが-1（経路が存在しない）の場合は計算を行わないようにしている。

最短経路の表示について

ダイクストラ法を実行した後に、求めた最短経路を表示する処理は、関数`printResult()`に記述した。経路はスタートノードからゴールノードまでの経路を表示したい。しかし、リザルトテーブルからはゴールノードからスタートノードの順（逆順）に経路が判明する。そこで、再起処理を用いて次のようなアルゴリズムを組むことで、スタートノードからゴールノードの順に最短経路を表示した。

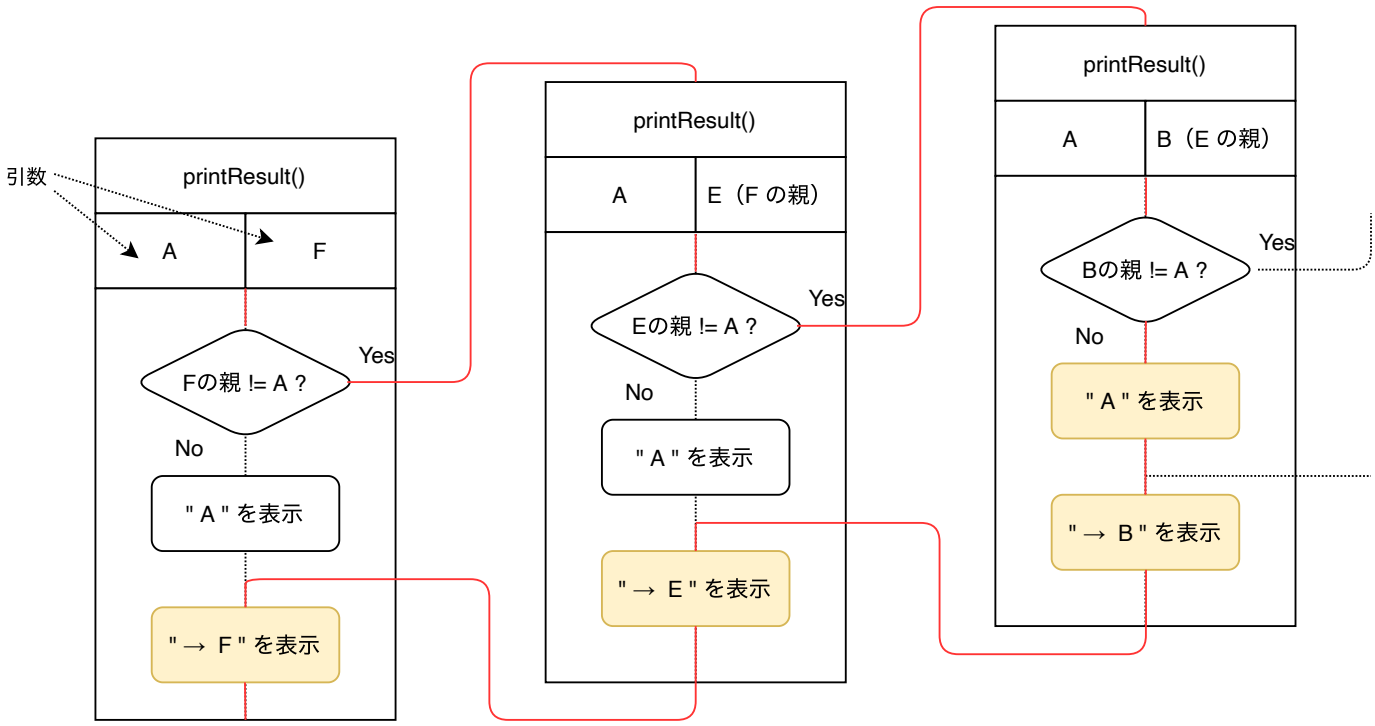
1. 到着ノード（最初の呼び出しではゴールノード）の親ノードがスタートノードと一致するかどうかを確認
2. 一致しない場合、関数は自身を再帰的に呼び出し、到着ノードの親ノードを新たな到着ノードとしてパスをたどる
3. 親ノードが到着ノードと一致する場合、つまり現在のノードがスタートノードである場合、そのノードを表示する
4. 最後に、現在のノードから次に訪れるノードの名前を表示する



経路図

ノード	コスト	親ノード	確定フラグ
0 (A)	0	0 (A)	1
1 (B)	1	0 (A)	1
2 (C)	3	1 (B)	1
3 (D)	5	1 (B)	1
4 (E)	4	1 (B)	1
5 (F)	5	4 (E)	1

NODEINFO resultTable [nodeNum]



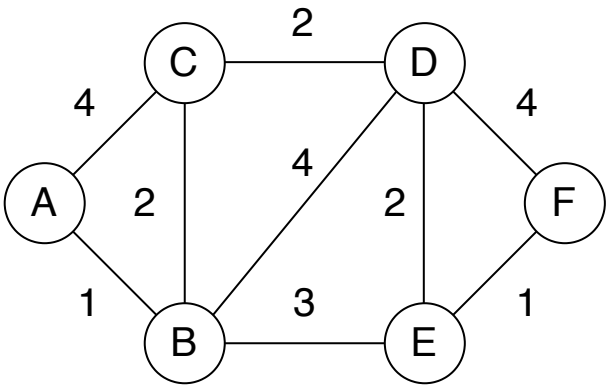
ノードAからノードFの最短経路の表示の例

コンソール

```
A → B → E → F
```

実行結果

- `example_NodePath.txt`を用いて、プログラムを実行した結果



example_NodePath.txt の経路図

- スタートノード: 0（A）, ゴールノード: 5（F）とした場合

```
スタートノードとゴールノードを入力してください
A: 0, B: 1, C: 2, ...
スタート: 0
ゴール: 5

A -> B -> E -> F
総コスト: 5
```

- スタートノード: 5（F）, ゴールノード: 2（C）とした場合

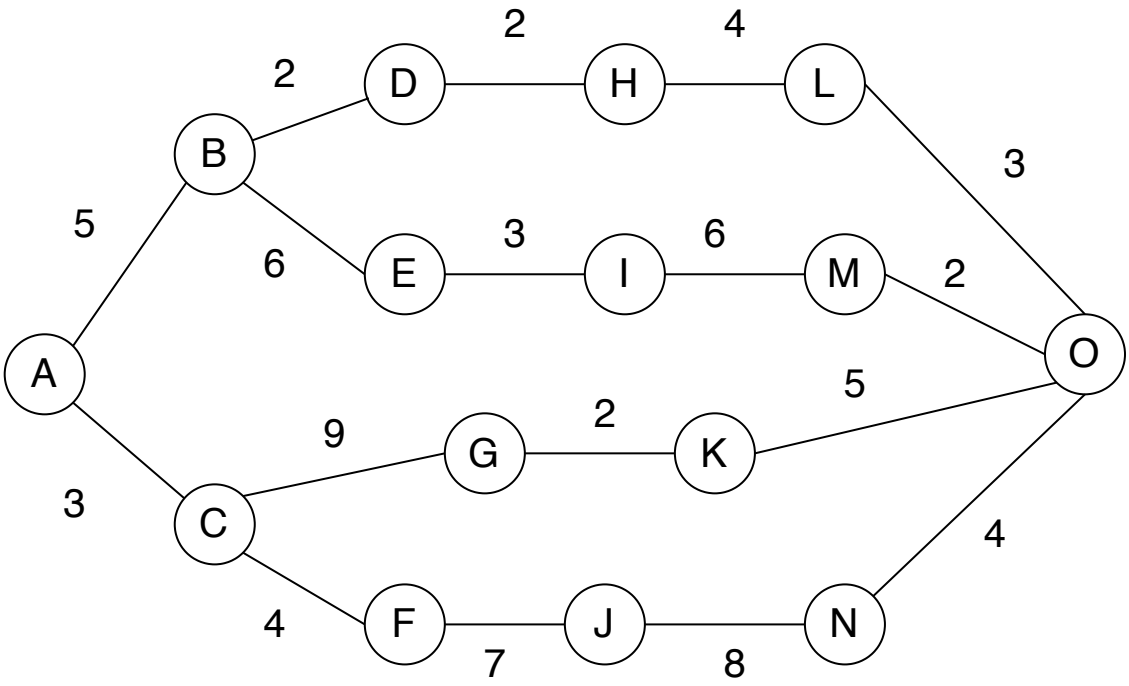
```
スタートノードとゴールノードを入力してください
A: 0, B: 1, C: 2, ...
スタート: 5
ゴール: 2

F -> E -> D -> C
総コスト: 5
```

- `my_NodePath_1.txt`を用いて、プログラムを実行した結果

`my_NodePath_1.txt`

```
17 15
0 1 5
0 2 3
1 3 2
1 4 6
2 5 4
2 6 9
3 7 1
4 8 3
5 9 7
6 10 2
7 11 4
```

```
8 12 6
9 13 8
10 14 5
11 14 3
12 14 2
13 14 4
```



My_NodePath_1.txt の経路図

- スタートノード: 0（A）, ゴールノード: 14（O）とした場合

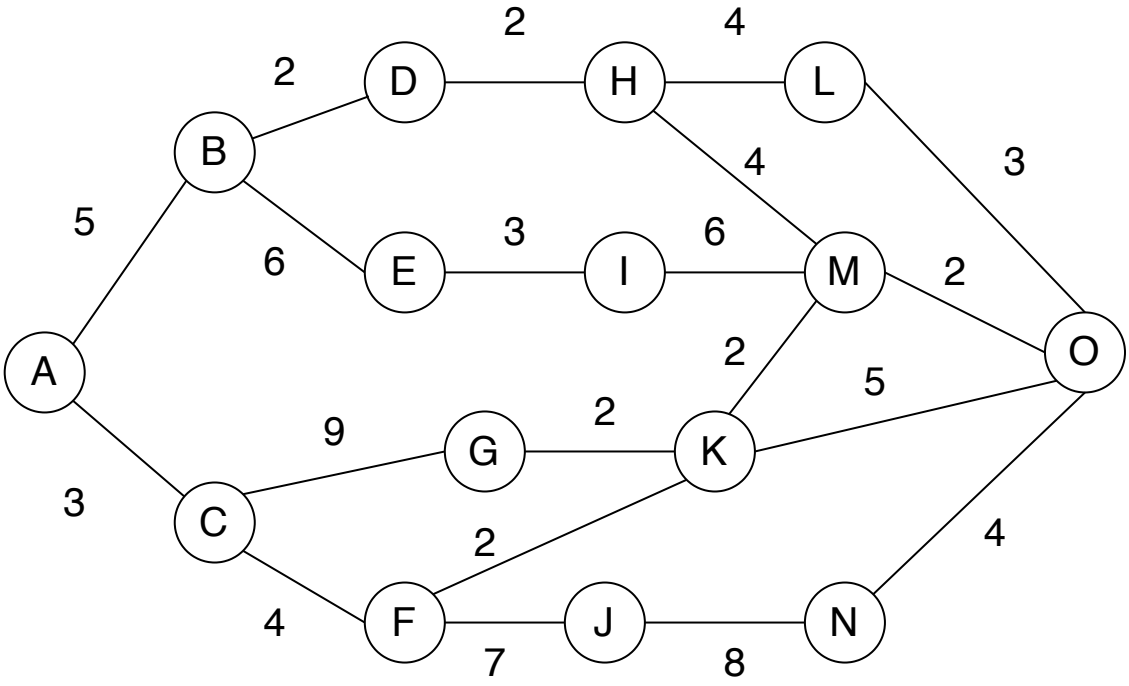
```
スタートノードとゴールノードを入力してください
A: 0, B: 1, C: 2, ...
スタート: 0
ゴール: 14

A -> B -> D -> H -> L -> O
総コスト: 15
```

- **`my_NodePath_2.txt`を用いて、プログラムを実行した結果**

`my_NodePath_2.txt`

```
20 15
0 1 5
0 2 3
1 3 2
1 4 6
2 5 4
```

```
2 6 9
3 7 1
4 8 3
5 9 7
5 10 2
6 10 2
7 11 4
7 12 4
8 12 6
9 13 8
10 12 2
10 14 5
11 14 3
12 14 2
13 14 4
```



My_NodePath_2.txt の経路図

- スタートノード: 0（A）, ゴールノード: 14（O）とした場合

```
スタートノードとゴールノードを入力してください
A: 0, B: 1, C: 2, ...
スタート: 0
ゴール: 14

A -> C -> F -> K -> M -> O
総コスト: 13
```

考察

今回プログラムの作成にあたり、コードのモジュール化を意識して関数を作成した。その結果、関数の再利用性が高くなり、プログラムの見通しも良くなった。また、ダイクストラ法の計算結果を保存する方法として構造体配列を用いた表を利用したが、その構造体について上手に定義できたと考える。しかし、ダイクストラ法の計算結果を保存する方法については、もう少し効率的な方法があるのではないかとも考えている。例えば、ノードの数が多い場合、構造体配列を用いた表の作成はメモリの無駄遣いになる可能性がある。そのため、今後は、リザルトテーブルの作成方法についてもう少し検討してみたい。　　このことは、コストテーブルの作成についても同様のことが言える。今回は、コストテーブルを二次元配列で表現したが、これも経路数が多い場合、かなりのオーダーでメモリを消費する可能性がある。そのためより大規模な処理も可能にするためには、これら対策としてハッシュマップやポインタと木構造を用いた表の作成など工夫するべきだと考える。

また、ダイクストラ法の計算結果を表示する際に、再帰処理を用いた。この方法は、スタートノードからゴールノードまでの経路を一連の処理で表示できるという利点がある。このほかに、スタックを用いてゴールからスタートまでの経路を保存し、それを逆順に表示する方法（LIFO）も考えた。しかし、再帰処理のプログラムの見通しを良くするという利点とLIFOの無駄なアクセスが多いという欠点（一度読み込んで保存した内容を後ろから再度読み込む必要）から、今回はこの再帰処理の方法を採用した。ただ、再帰処理の深さが深くなるとスタックオーバーフローが発生する可能性があるため、この処理も改善の余地があると考える。特に、今回は経路を表示するための処理をダイクストラ法の処理と独立して作成したが、これらを統合して、ダイクストラ法の実行と平行して最短経路を保存することができれば、より効率的なプログラムになると考えた。

課題2　CRC（Cyclic Redundancy Check）による誤り検出プログラムの作成

ソースファイル

ソースファイル：`CRC.c`

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

/* プロトタイプ宣言 */
// CRCを実装する関数
unsigned short exeCRC(unsigned short data[], int data_length);

// データ系列を表示する関数
void printResult(unsigned short data[], int data_length);

// ----- プロトタイプ宣言ここまで



int main(void){
    // データ系列（オリジナルデータ系列）
    unsigned short data[] = {
        0x1111,
        0x2222,
        0x3333,
        0x4444,
    };
    int data_length = sizeof(data) / sizeof(data[0]);

    // CRCを付加する前のデータ系列を表示
    printf("元データ系列: ");
    printResult(data, data_length);


    // CRC計算する
    unsigned short crc = exeCRC(data, data_length);
    printf("CRC: 0x%04X\n", crc);   // CRCを表示


    // CRCを付加したデータ系列（送信データ系列）
    unsigned short data2[] = {
        0x1111,
        0x2222,
        0x3333,
        0x4444,
        crc,    // CRCを付加
    };
    int data2_length = sizeof(data2) / sizeof(data2[0]);
```

```
    // CRCで符号化されたデータ系列を受信したと仮定してデータ系列の誤りの有無を確認
    unsigned short remainder = exeCRC(data2, data2_length);


    // 実行結果を表示
    printf("CRCを付加したデータ系列: ");
    printResult(data2, data2_length);

    // (受信したデータ/生成多項式)の余りが0なら誤りなし、それ以外なら誤りあり
    if (remainder == 0) {
        printf("誤りなし\n");
    } else {
        printf("誤り検出\n");
    }

    return 0;
}


// CRCを実装する関数
unsigned short exeCRC(unsigned short data[], int data_length) {
    unsigned short crc = 0x0000;  // CRCの初期値
    //  x^16 + x^11 + x^4 + x + 1  ->  (0b 1 0000 1000 0001 0011)  ->  (ox 10813)
    unsigned short poly = 0x0813;  // CRC-16の生成多項式

    for (int i = 0; i < data_length; i++) {
        crc ^= data[i];
        for (int j = 0; j < 16; j++) {
            // 先頭ビットが1の場合は残りの15ビットと生成多項式をXOR
            if (crc & 0x8000) {
                crc = (crc << 1) ^ poly;    // 先頭ビット以外をXORするためにcrcを左シフ
            }
            // 先頭ビットが0の場合はcrcを左シフトしてXORはしない
            else {
                crc = crc << 1;
            }
        }
    }
    return crc;
}


// データ系列を表示する関数
void printResult(unsigned short data[], int data_length) {
    for (int i = 0; i < data_length; i++) {
        printf("0x%04X ", data[i]);
    }
    printf("\n");
}
```

プログラムの説明

プログラムの概要

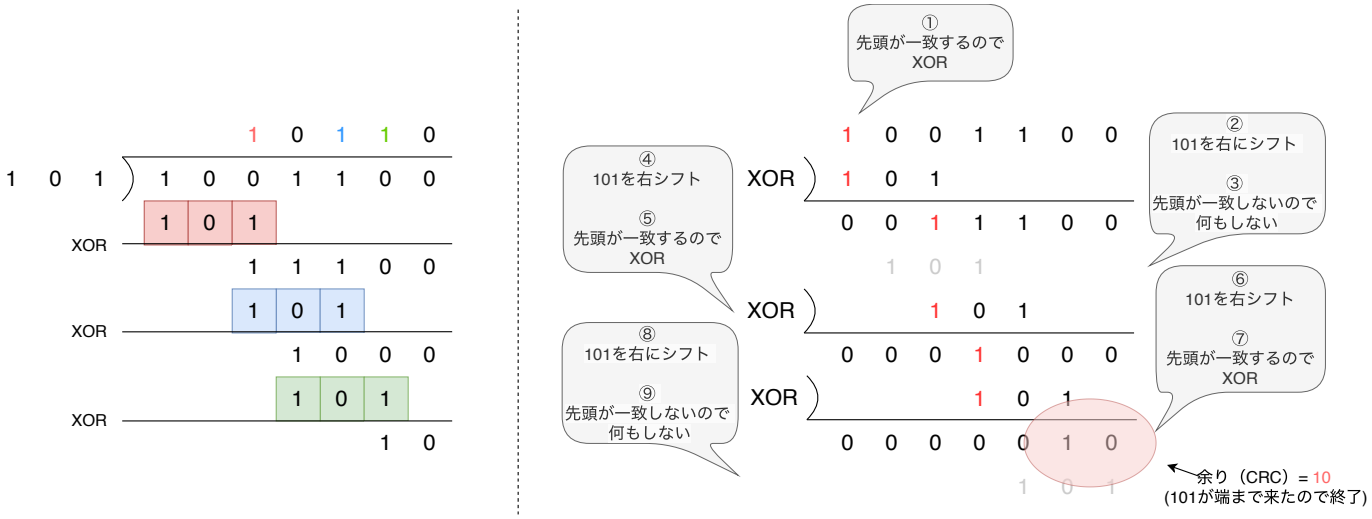
CRC（Cyclic Redundancy Check）による誤り検出プログラムを作成した。CRCは、データ系列に対して生成多項式を用いて誤り検出符号を生成する手法である。このプログラムは、CRCを付加する前のデータ系列を用意し、そのデータ系列に対しCRCを計算し付加した新たなデータ系列を作成する。そして、CRCを付加したデータ系列を受信したと仮定して、データ系列の誤りの有無を確認する。

CRCの計算について

CRCの計算は、関数 `exeCRC()` で行う。CRCの計算自体は、単純にデータ系列に対して生成多項式の除算を行うだけで実現できる（CRC自体は除算の余り）。つまり、関数のアルゴリズムとしては除算のアルゴリズムを考えることが必要である。では、どのようにして除算を行えば良いだろうか。実は次の2つの処理を繰り返すだけで良い。

1. 先頭が一致するビットのXORを取る。一致しない場合は何もしない。
2. ビット列をシフトする

この処理をビット列が端になるまで繰り返すことで、最終的には余りが求まる。簡単な例でためしてみると、次のようになる（ただし実際には元のデータ系列自体が余り（CRC）になる可能性を防ぐために(生成多項式の桁数-1)分だけ元データの右端に0を付加する）。

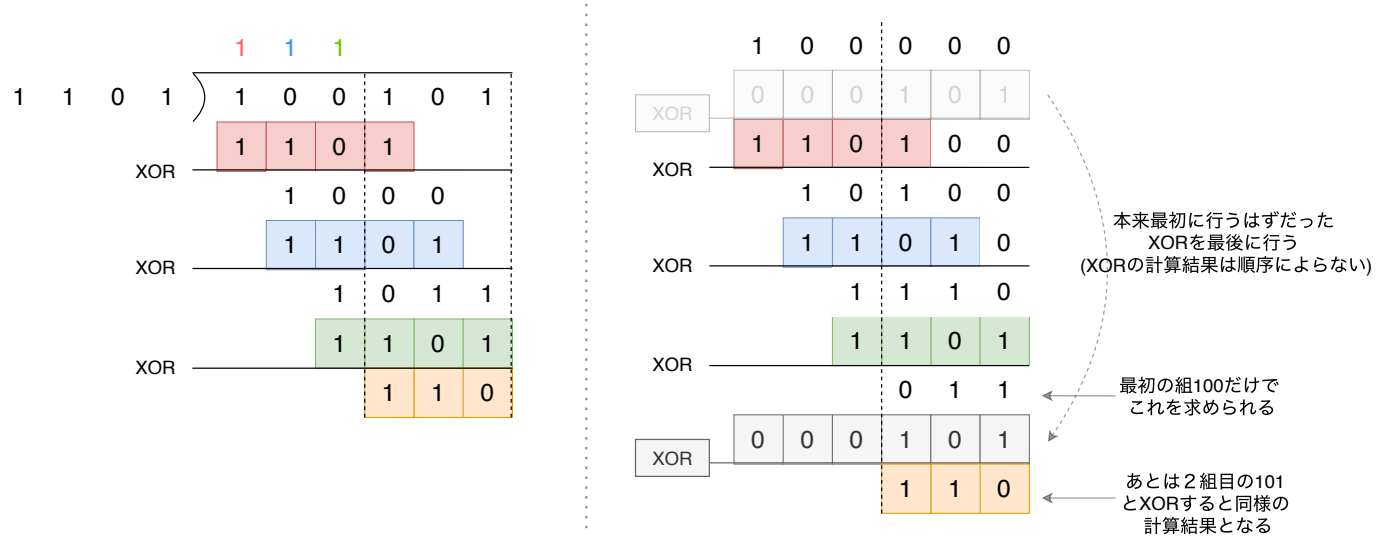


1001100 ÷ 101 の例

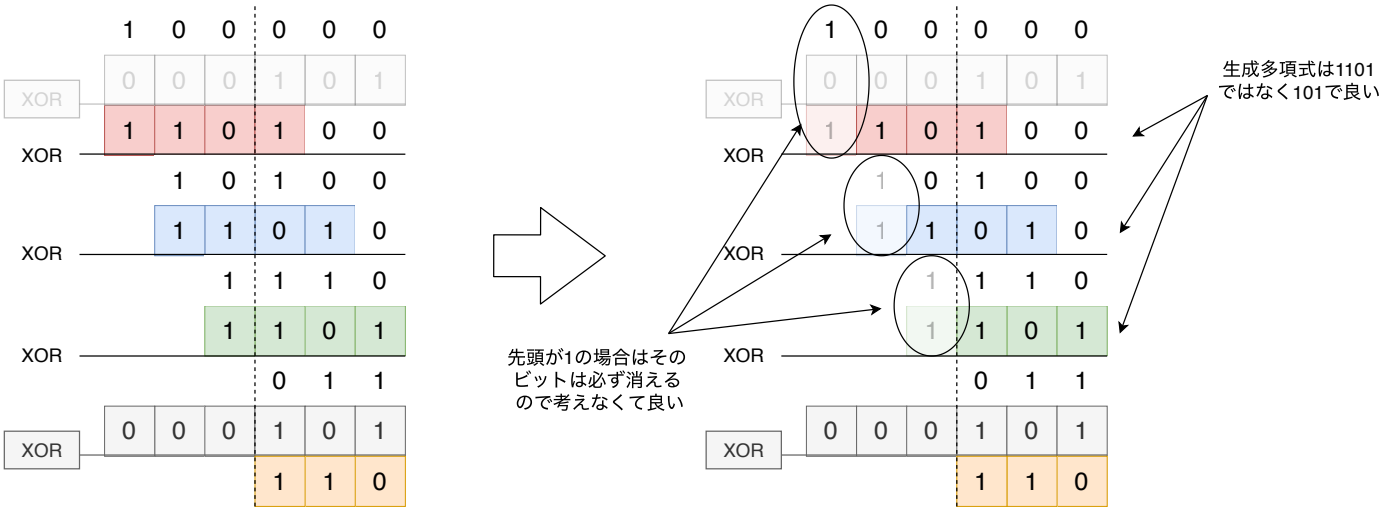
ただしこの方法には問題がある。それは、このアルゴリズム通りに計算しようとすると、割られるデータ系列全体を保存する変数が必要となることである。実際には、例で示した7bit程度ではなく10000bitのような大きなデータ系列を扱う必要がある。そうなるとメモリの大きさを考えるとこの方法では実装不可能である。

そこで実際のCRCの計算では、（生成多項式の桁数-1）ビットだけ保存する変数を用意し、その変数の大きさごとに元のデータ系列を分割することで計算を行う。この方法を用いることで、メモリの使用量を抑

えつつ、CRCの計算を行うことができる。アルゴリズムとしては次の図ようになる。



これにより（生成多項式の桁数-1）ごとに元のデータ系列を分割しても、余り（CRC）を求めることができるようになった。ただし実際のプログラムでは、除算を行う際に先頭のビットの1は必ず消えるためXORの計算に含める必要がない。よって割られるデータ系列の先頭ビットが1の場合は、その先頭ビットを除くビット列でXORを計算するように工夫した。



データ系列の誤り検出について

データ系列を受信した側では、そのデータ系列のCRCを作成した側と同じ生成多項式を用いて除算を計算する。そしてもし、その除算の余りが0であれば、データ系列に誤りはないと判断する。一方で、余りが0でなければ、データ系列に誤りがあると判断する。

プログラムでは、元のデータ系列に対してCRCを計算して付加した新たなデータ系列を用意する。そしてその新たなデータ系列に対してこの処理を行う。ただし、関数 `exeCRC()` での説明でも述べたように、この関数は除算を行う関数でもある。よってこの関数を再利用することで、データ系列の誤り検出を行うことができる。

実行結果

- データ系列 `0x1111 0x2222 0x3333 0x4444` に対して実行した結果

```
元データ系列: 0x1111 0x2222 0x3333 0x4444
CRC: 0x3EE9
CRCを付加したデータ系列: 0x1111 0x2222 0x3333 0x4444 0x3EE9
誤りなし
```

- データ系列 `0x1111 0x2222 0x3333 0x4444` に対してCRCを付加したデータ系列は `0x1111 0x2222 0x3333 0x4444 0x0F0F` であるが、送信データを `0x1111 0x2202 0x3333 0x4444 0x0F0F` と変更した場合の結果

```
元データ系列: 0x1111 0x2222 0x3333 0x4444
CRC: 0x3EE9
CRCを付加したデータ系列: 0x1111 0x2202 0x3333 0x4444 0x3EE9
誤り検出
```

考察

工夫した点として、プログラムの説明でも述べたように、CRCの計算アルゴリズムにおいて、メモリの使用量を抑えるために（生成多項式の桁数-1）ビットごとに元のデータ系列を分割して計算を行うことができるようにした。この工夫により、大きなデータ系列に対してもCRCの計算を行うことができるようになった。また、XORの計算で必要のない先頭ビットの計算を省略することで、計算量やリソースの使用量を抑えることができた。

改善点として、今回のプログラムではデータ系列をコード内で直接定義しているが、実際のデータ系列はファイルから読み込むことが多い。そのため、ファイルからデータ系列を読み込む処理を追加することでより良いプログラムとなると考えた。また、講義で生成多項式は(x+1)を因数にもつものを用いることが多いと学んだ。これは、生成多項式が(x+1)を因数に持つことで、受信側は除算より単純なパリティ検査をすることで誤り検出を行うことができるためである。そのため、生成多項式を(x+1)を因数に持つものに変更することで、より効率的なプログラムとなると考えた。

参考文献

- 巡回冗長検査(CRC, Cyclic Redundancy Check)の原理と実装 - ろぐれこーど. https://dlrecord.hatenablog.com/entry/2020/11/08/151017. (2023/2/16 参照)
- CRC-32 | PPT. https://www.slideshare.net/7shi/crc32. (2023/2/16 参照)