

adjacencyMatrix 과제

20223147 주감동

전체 프로그램 구조

- 1. 입력 단계
- 사용자에게 행렬 입력 방식 선택
 - 1: 0/1 난수로 관계 행렬 자동 생성
 - 2: 수동 입력
- mat[5][5]배열에 관계 저장
- 생성된 행렬은 print_matrix()로 출력

전체 프로그램 구조

- 2. 관계 성질 검사 단계
 - `is_reflexive()` → 대각선이 모두 1인지 검사
 - `is_symmetric()` → (i, j) 와 (j, i) 가 항상 같은지 검사
 - `is_transitive()` → $i \rightarrow k$, $k \rightarrow j$ 이면 $i \rightarrow j$ 인지 확인
 - 세 성질이 모두 `true` → 동치 관계 판정

전체 프로그램 구조

- 3. 동치류 계산 단계
- 전체 동치류: `all_equivalence_classes()`
- 같은 행에서 관계가 1인 원소들을 묶어 동치류 생성
- 개별 동치류: `equivalence_classes(x)`
- x번째 행에서 1인 위치를 모아 해당 원소의 동치류 출력
- 화면에
- “동치류 1: ...”
- “1의 동치류: ...”
- 형태로 출력

전체 프로그램 구조

- 4. 폐포(closed relation) 계산 단계
- 반사 폐포reflexive_closure()
- 대각선 원소를 모두 1로 설정
- 대칭 폐포symmetric_closure()
- $\text{in}[i][j] = 1 \rightarrow \text{out}[j][i] = 1$ 추가
- 추이 폐포transitive_closure()
- 워셜 알고리즘 방식으로 $i \rightarrow k \rightarrow j$ 경로를 $i \rightarrow j$ 로 추가
- 각 폐포 적용 후:
 - 새로운 행렬 출력
 - 해당 관계의 동치 여부 및 동치류 재확인

전체 프로그램 구조

- 5. 전체 요약
- 1. 입력 방식 선택 (자동/수동)
- 2. 원본 관계 행렬 출력
- 3. 반사, 대칭, 추이성 검사
- 4. 동치 관계면 동치류 출력
- 5. 성질 부족 시 반사/대칭/추이 폐포 각각 적용
- 6. 폐포 결과를 다시 출력 및 판정

- const int n = 5;
 - → 집합 A의 원소 개수를 5로 고정하고, 5×5 관계 행렬을 사용.
 - 2차원 배열 int mat[n][n]; 에 관계 행렬을 저장한다.
-
- main함수에서 다음과 같이 입력 방식을 선택한다.
 - 1입력: 0과 1을 난수로 사용하여 관계 행렬 자동 생성
 - 2입력: 사용자로부터 5×5 행렬을 직접 입력
 - srand(time(NULL));를 사용하여 실행할 때마다 다른 랜덤 행렬이 생성되도록 설정

```
matrix 자동 생성 1 or 수동 입력 2: 1
```

```
----- origin matrix:
```

```
0 1 0 0 0
```

```
0 0 1 0 0
```

```
0 0 1 0 0
```

```
1 1 1 1 1
```

```
1 0 1 0 1
```

```
동치 관계가 아님.|
```

```
matrix 자동 생성 1 or 수동 입력 2: 2
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
0 0 0 1 1
```

```
0 0 0 1 1
```

```
----- origin matrix:
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
0 0 0 1 1
```

```
0 0 0 1 1
```

- `print_matrix(mat);` 함수를 통해, 현재 관계를 나타내는 5×5 행렬을 화면에 출력한다.

```
----- origin matrix:  
1 1 1 0 0  
1 1 1 0 0  
1 1 1 0 0  
0 0 0 1 1  
0 0 0 1 1
```

- 동치 관계 판별 및 동치류 출력
- `print_equivalence_classes(mat);`함수에서
- `is_reflexive`, `is_symmetric`, `is_transitive`를 순서대로 호출하여
- 관계가 동치 관계인지 확인한다.
- 세 함수가 모두 `true`를 반환하면:
 - "동치 관계."메시지 출력
 - `all_equivalence_classes`와 `equivalence_classes`를 호출하여
 - 전체 동치류 분할, 각 원소별 동치류를 화면에 출력한다.
 - 하나라도 만족하지 않으면 "동치 관계가 아님."을 출력한다.

```
void print_equivalence_classes(int mat[][n]) {
    if (is_reflexive(mat) && is_symmetric(mat) && is_transitive(mat)) {
        cout << "동치 관계." << endl;
        vector<vector<int>> als = all_equivalence_classes(mat);
        for (int i = 0; i < als.size(); i++) {
            cout << "동치류 " << i + 1 << ": ";
            for (auto j: als[i]) {
                cout << j << ' ';
            }
            cout << endl;
        }
        cout << endl;

        vector<int> tmp;
        for (int i = 0; i < n; i++) {
            cout << i + 1 << "의 동치류: ";
            for (auto j: equivalence_classes(i, mat)) {
                cout << j << ' ';
            }
            cout << endl;
        }

    } else {
        cout << "동치 관계가 아님." << endl;
    }
    cout << endl;
}
```

- 반사인지 검사
- $A[i][i]$ 는
- 전부 1이어야한다

```
bool is_reflexive(int matrix[][][n])
    for (int i = 0; i < n; i++) {
        if (matrix[i][i] == 0) {
            return false;
        }
    }
    return true;
}
```

- 대칭 검사

- $A[i][j]$ 와 $A[j][i]$ 는
- 같아야한다.

```
bool is_symmetric(int matrix[][]n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            if (matrix[i][j] != matrix[j][i]) {  
                return false;  
            }  
        }  
    }  
    return true;  
}
```

- 추이 검사
- $i \rightarrow k$ $k \rightarrow j$ 라면 $i \rightarrow j$ 여야 한다.

```
bool is_transitive(int matrix[][]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                if (matrix[i][k] == 1 && matrix[k][j] == 1 && matrix[i][j] == 0) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

- 만약 해당 행렬이 동치라면 j와 동치류는 j와 연결된 노드 전체가 된다.

```
vector<int> equivalence_classes(int x, int matrix[][][n]) {
    vector<int> r;
    for (int j = 0; j < n; j++) {
        if (matrix[x][j]) {
            r.push_back(j + 1);
        }
    }
    return r;
}
```

- 따라서
- 동치류로 분할은
- set에 들어있지
- 않은 노드들을
- 넣으면서 만들
- 수 있다.

```

vector<vector<int>> all_equivalence_classes(int matrix[][n]) {
    vector<vector<int>> r;
    set<int> st;
    for (int j = 0; j < n; j++) {
        if (st.count(j)) continue;
        vector<int> tmp;
        for (int k = 0; k < n; k++) {
            if (matrix[j][k]) {
                st.insert(k);
                tmp.push_back(k + 1);
            }
        }
        r.push_back(tmp);
    }
    return r;
}

```

- 반사가 아니라면
- 반사폐포 적용후
- 동치인지
- 동치라면 동치류를 출력한다.
- 대칭, 추이도
- 동일하게 진행한다.

```
int tmp[n][n];
if (!is_reflexive(mat)) {
    reflexive_closure(mat, tmp);
    cout << "----- 반사 폐포 변환 후 matrix:" << endl;
    print_matrix(tmp);
    cout << endl;
    print_equivalence_classes(tmp);
}
if (!is_symmetric(mat)) {
    symmetric_closure(mat, tmp);
    cout << "----- 대칭 폐포 변환 후 matrix:" << endl;
    print_matrix(tmp);
    cout << endl;
    print_equivalence_classes(tmp);
}
if (!is_transitive(mat)) {
    transitive_closure(mat, tmp);
    cout << "----- 추이 폐포 변환 후 matrix:" << endl;
    print_matrix(tmp);
    cout << endl;
    print_equivalence_classes(tmp);
}
```

- 반사폐포는 $i \rightarrow i$ 를 갈 수 있게 바꾼다.

```
void reflexive_closure(int in[][n], int out[][n]) {
    copy_mat(out, in);
    for (int i = 0; i < n; i++) {
        out[i][i] = 1;
    }
}
```

- 대칭폐포는 $i \rightarrow j$ 로 갈 수 있다면 $j \rightarrow i$ 로도 갈 수 있게 바꾼다.

```
void symmetric_closure(int in[][][n], int out[][][n]) {  
    copy_mat(out, in);  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            if (in[i][j] == 1) {  
                out[j][i] = 1;  
            }  
        }  
    }  
}
```

- 추이폐포는 $i \rightarrow k$ 로 갈 수 있고 $k \rightarrow j$ 로도 갈 수 있다면
- $i \rightarrow j$ 로도 갈 수 있게 바꾼다.
- for문 순서가 중요하다. 해당 방법은 플로이드 와샬 알고리즘이다.

```
void transitive_closure(int in[][]n, int out[][]n) {  
    copy_mat(out, in);  
    for (int k = 0; k < n; k++) {  
        for (int i = 0; i < n; i++) {  
            for (int j = 0; j < n; j++) {  
                if (out[i][k] == 1 && out[k][j] == 1) {  
                    out[i][j] = 1;  
                }  
            }  
        }  
    }  
}
```

- 각 폐포를 적용한 뒤 origin 과 동일하게
- print_equivalence_classes를 사용하여
- 동치인지
- 동치라면 동치류를 출력하게 한다.

```
void print_equivalence_classes(int mat[][n]) {
    if (is_reflexive(mat) && is_symmetric(mat) && is_transitive(mat)) {
        cout << "동치 관계." << endl;
        vector<vector<int>> als = all_equivalence_classes(mat);
        for (int i = 0; i < als.size(); i++) {
            cout << "동치류 " << i + 1 << ":" ;
            for (auto j: als[i]) {
                cout << j << " ";
            }
            cout << endl;
        }
        cout << endl;

        vector<int> tmp;
        for (int i = 0; i < n; i++) {
            cout << i + 1 << "의 동치류: ";
            for (auto j: equivalence_classes(i, mat)) {
                cout << j << ' ';
            }
            cout << endl;
        }

    } else {
        cout << "동치 관계가 아님." << endl;
    }
    cout << endl;
}
```

- 1. 관계 행렬 입력 가능

```
matrix 자동 생성 1 or 수동 입력 2: 2
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
0 0 0 1 1
```

```
0 0 0 1 1
```

- 1. 관계 행렬 입력 가능

```
matrix 자동 생성 1 or 수동 입력 2: 2
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
0 0 0 1 1
```

```
0 0 0 1 1
```

- 2. 동치 관계 판별 기능

```
----- origin matrix:
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
1 1 1 0 0
```

```
0 0 0 1 1
```

```
0 0 0 1 1
```

동치 관계.

- 3. 동치 관계일 경우 동치류 출력 기능

동치 관계.

동치류 1: 1 2 3

동치류 2: 4 5

1의 동치류: 1 2 3

2의 동치류: 1 2 3

3의 동치류: 1 2 3

4의 동치류: 4 5

5의 동치류: 4 5

|

• 4. 폐포 구현 기능

----- 츠미 폐포 변환 후 matrix:

```
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1
```

동치 관계.

동치류 1: 1 2 3 4 5

1의 동치류: 1 2 3 4 5

2의 동치류: 1 2 3 4 5

3의 동치류: 1 2 3 4 5

4의 동치류: 1 2 3 4 5

5의 동치류: 1 2 3 4 5

----- 대칭 폐포 변환 후 matrix:

```
0 0 1 1 0  
0 0 1 0 1  
1 1 0 1 0  
1 0 1 0 1  
0 1 0 1 0
```

동치 관계가 아님.

----- 반사 폐포 변환 후 matrix:

```
1 0 1 1 0  
0 1 1 0 0  
1 0 1 1 0  
1 0 0 1 1  
0 1 0 0 1
```

동치 관계가 아님.

- 5. 추가기능
- 입력한 옵션이 1이라면 c++ rand함수로 랜덤 matrix으로 설정, srand(time)으로 실행할 때마다 랜덤하게 설정.

```

cout << "matrix 자동 생성 1 or 수동 입력 2: ";
int opt;
cin >> opt;
int mat[n][n];
srand(time(NULL));
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (opt == 2) cin >> mat[i][j];
        else mat[i][j] = rand() % 2;
    }
}

```

matrix 자동 생성 1 or 수동 입력 2: 1

----- origin matrix:

0	0	1	1	0
0	0	1	0	0
1	0	0	1	0
1	0	0	0	1
0	1	0	0	0

동치 관계가 아님.

```
matrix 자동 생성 1 or 수동 입력 2: 1
```

```
----- origin matrix:
```

```
1 1 0 1 1  
0 1 1 0 1  
1 1 0 0 1  
0 0 1 1 0  
1 0 1 1 0
```

동치 관계가 아님.

```
----- 반사 폐포 변환 후 matrix:
```

```
1 1 0 1 1  
0 1 1 0 1  
1 1 1 0 1  
0 0 1 1 0  
1 0 1 1 1
```

동치 관계가 아님.

```
----- 대칭 폐포 변환 후 matrix:
```

```
1 1 1 1 1  
1 1 1 0 1 |  
1 1 0 1 1  
1 0 1 1 1  
1 1 1 1 0
```

동치 관계가 아님.

```
----- 주이 폐포 변환 후 matrix:
```

```
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1  
1 1 1 1 1
```

동치 관계.

동치류 1: 1 2 3 4 5

1의 동치류: 1 2 3 4 5

2의 동치류: 1 2 3 4 5

3의 동치류: 1 2 3 4 5

4의 동치류: 1 2 3 4 5

5의 동치류: 1 2 3 4 5

```
matrix 자동 생성 1 or 수동 입력 2: 2
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
0 0 0 1 1
0 0 0 1 1
----- origin matrix:
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
0 0 0 1 1
0 0 0 1 1
```

동치 관계.

동치류 1: 1 2 3

동치류 2: 4 5

1의 동치류: 1 2 3

2의 동치류: 1 2 3

3의 동치류: 1 2 3

4의 동치류: 4 5

5의 동치류: 4 5

github

- <https://github.com/jjthis/adjacencyMatrix/blob/main/main.cpp>