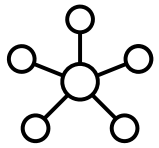

[제03주] 그래프의 사이클 검사

충남대학교 컴퓨터공학과 알고리즘 04 분반
제출일: 2021-10-05
학번: 201701975, 이름: 구건모

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

목차

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제 설명

- 과제에서 해야할 일
- 주요 개념 (PairwiseDisjointSets)

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제 개요

과제에서 해야할 일:

이번 과제의 목표는 지난 과제에서 구현했던 그래프 생성 기능에 Cycle Detection 기능을 추가로 구현하는 것이었습니다.

지난 과제에서 구현했던 그래프 생성 기능을 이용하여 그래프를 생성할 때 edge를 추가하는 과정에서 edge가 하나 추가될 때마다 cycle detection을 하여 cycle이 형성되었는지를 보여주는 것을 구현하는 것이 목표입니다. pairwiseDisjointSets으로 그래프에 추가된 컴포넌트들의 구성상태를 확인하여 추가하려는 edge의 headvertex와 tailvertex가 같은 set에 속한다면 cycle을 형성함을 이미하게 되고 이번 과제에서는 이를 위한 메소드와 클래스(PairwiseDisjointSets)를 구현하는 것이 주요 내용임을 알 수 있습니다.

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제의 주요 이론

과제의 주요 이론 (PairwiseDisjointSets):

pairwise disjointSet란 어떠한 집합도 다른 원소와 겹치는 원소가 존재하지 않는 서로소 관계를 가지는 집합들을 의미합니다.

cycle detection에 사용되는 개념인 pairwiseDisjointSets는 이번 과제의 주요 개념으로, 그래프 컴포넌트들의 연결관계를 tree 형태의 개념으로 구상하고 배열로 해당 정보를 저장하므로서 해당 정보를 이용해 사이클의 존재여부를 확인하는데 이용됩니다.

pairwiseDisjointSets의 기본연산에는 find와 union이 존재하는데 find는 element가 어느 집합에 속하는지를 찾아주고 union은 두 element 가 속한 집합의 합집합을 구하는 연산입니다.

Weighting Rule, Collapsing Rule

weighting rule 과 collapsing rule은 두 rule 모두 tree의 높이를 낮추기 위해 고안된 방법입니다. 연산 과정의 결과로 tree의 높이가 높아지거나 높은 값을 계속 유지하게 된다면 find 수행시 더 많은 loop를 돌게 되므로 비효율을 야기하게 됩니다.

따라서 이를 개선하기 위해 사용된 방식들로 보고서 코드 설명 부분에 따로 정리하였습니다.

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

Class별 설명

- Class PairwiseDisjointSets
- Class AppController

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

PairwiseDisjointSets

```
public class PairwiseDisjointSets {
    private static final int INITIAL_SINGLETON_SET_SIZE = 1;

    private int _numberOfElements;
    private int[] _parentTree;

    public PairwiseDisjointSets(int givenNumberOfElements) {
        this.setNumberOfElements(givenNumberOfElements);
        this.setParentTree(new int[this.numberOfElements()]);
        for (int rootOfSingletonSet = 0; rootOfSingletonSet < this.numberOfElements(); rootOfSingletonSet++) {
            this.setSizeOfSetFor(rootOfSingletonSet, INITIAL_SINGLETON_SET_SIZE);
        }

    }

    public int find(int aMember) {
        int rootCandidate = aMember;
        while (this.parentDoesExist(rootCandidate)) {
            rootCandidate = this.parentOf(rootCandidate);
        }
        int root = rootCandidate;
        // Collapsing Rule 적용
        int child = aMember;
        int parent = this.parentOf(child);
        if (parent >= 0) {
            while (parent != root) {
                this.setParentOf(child, root);
                child = parent;
                parent = this.parentOf(child);
            }
        }
        return root;

    }

    public void union(int aMemberA, int aMemberB) {
        int rootOfSetA = find(aMemberA);
        int rootOfSetB = find(aMemberB);

        int sizeOfSetA = this.sizeOfSetFor(rootOfSetA);
        int sizeOfSetB = this.sizeOfSetFor(rootOfSetB);

        // weighting rule 적용
        if (sizeOfSetA < sizeOfSetB) {
            this.setParentOf(rootOfSetA, rootOfSetB);
            this.setSizeOfSetFor(rootOfSetB, (sizeOfSetA + sizeOfSetB));

        } else {
            this.setParentOf(rootOfSetB, rootOfSetA);
            this.setSizeOfSetFor(rootOfSetA, (sizeOfSetA + sizeOfSetB));

        }

    }
```

Class PairwiseDisjointSets:

pairwiseDisjointSets는 element 개수를 의미하는 _numberOfElements와 Tree정보를 담을 _parentTree 멤버변수가 선언되어 있고 초기 생성시에는 element 개수 만큼의 singleton set을 만든후 초기값으로 사용하기 위해 선언한 INITIAL_SINGLETON_SET_SIZE 값으로 초기화해주게 됩니다

* PairwiseDisjointSets의 메소드

find method

find 메소드는 원소가 속한 집합을 리턴해주게 됩니다. find 메소드는 Collapsing Rule을 적용하여 구현하였습니다. collasping rule은 트리의 높이를 낮게 유지하여 find의 효율을 높히고자 고안된 방식으로, 기존의 find 처럼 루트 노드를 찾은 다음 끝나는 것이 아니라 해당 member의 root를 찾을 때까지 거쳤던 모든 노드들을 root의 자식 노드로 만들어 tree의 높이를 낮추는 방식으로 find가 진행됩니다.
따라서 find 메소드를 수행할 때마다 tree의 높이를 낮추는 방향으로 가게 되므로 높이가 높아져서 생기는 기존 find의 비효율성을 개선하는 효과를 보여줍니다.

union method

union은 union method는 weighting rule을 적용하여 구현하였습니다. weigthing rule을 통해서 tree의 depth 가 깊어지지 않도록 하는 방향을 추구하게 됩니다. tree의 높이가 높아질수록 find 메소드를 수행할 때 효율성이 떨어지게 됩니다. 따라서 이를 해소하기 위한 방안으로 weighting rule을 이용한 것입니다. weighting rule 이란 집합의 원소의 개수가 많으면 상대적으로 tree의 높이가 높을것이라고 가정하여 원소의 개수가 적은 쪽이 많은쪽으로 붙이는 방식입니다. 기존에는 root의 값을 -1으로 지정하여 표현하였지만 weighting rule을 적용하기 위해서는 각 집합의 원소 개수를 알아야하므로 root 값에 -1 대신 -1 * (집합 원소의 개수)로 지정하여 집합의 원소 개수에 대한 정보가 반영되도록 합니다. 여전히 음수값이기 때문에 root를 member 값들과 구분하는데에도 문제가 발생하지 않습니다. 따라서 union 시 해당 값들을 비교하여 결과적으로 원소수가 더 적은 집합이 큰 집합으로 붙게 됩니다.

PairwiseDisjointSets – getter & setter

```
private int numberOfElements() {
    return this._numberOfElements;
}

private void setNumberOfElements(int newNumberOfElements) {
    this._numberOfElements = newNumberOfElements;
}

private int[] parentTree() {
    return this._parentTree;
}

private void setParentTree(int[] newParentTree) {
    this._parentTree = newParentTree;
}

private int parentOf(int aMember) {
    return this.parentTree()[aMember];

}

private void setParentOf(int aChildMember, int newParentMember) {
    this.parentTree()[aChildMember] = newParentMember;
}

private boolean parentDoesExist(int aMember) {
    return (this.parentOf(aMember) >= 0);
}

private int sizeOfSetFor(int aRoot) {
    return (-this.parentOf(aRoot));

}

private void setSizeOfSetFor(int aRoot, int newSize) {
    this.setParentOf(aRoot, -newSize);
}
```

* PairwiseDisjointSets의 getter와 setter

pairwiseDisjointSets 객체의 getter는 멤버변수인 _numberOfElements와 _parentTree 를 반환하며, setter는 각각의 멤버변수에 값을 할당하는데 이 때, 직접 멤버변수를 이용하지 않고 getter로 호출하여 새로운 값을 할당하게 됩니다.

또한 외부에서 사용되지 않기 때문에 private으로 선언되었습니다.

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

AppController – 추가된 부분

```
public class AppController {
    private AdjacencyMatrixGraph _graph;
    private PairwiseDisjointSets _pairwiseDisjointSets;

    public AppController() {
        this.setGraph(null);
        this.setPairwiseDisjointSets(null);
    }

    private PairwiseDisjointSets pairwiseDisjointSets() {
        return this._pairwiseDisjointSets;
    }

    private void setPairwiseDisjointSets(PairwiseDisjointSets newPairwiseDisjointSets) {
        this._pairwiseDisjointSets = newPairwiseDisjointSets;
    }


    private boolean addEdgeDoesMakeCycle(Edge anAddedEdge) {
        int tailVertex = anAddedEdge.tailVertex();
        int headVertex = anAddedEdge.headVertex();
        int setForTailVertex = this.pairwiseDisjointSets().find(tailVertex);
        int setForHeadVertex = this.pairwiseDisjointSets().find(headVertex);
        if (setForTailVertex == setForHeadVertex) {
            return true;

        } else {
            this.pairwiseDisjointSets().union(setForTailVertex, setForHeadVertex);
            return false;
        }
    }


    private void initCycleDetection() {
        this.setPairwiseDisjointSets(new PairwiseDisjointSets(this.graph().numberOfVertices()));
    }
```

*** Cycle Detection을 위해 추가된 부분**

멤버변수 _pairwiseDisjointSets가 선언되고, 사용자의 입력이 들어오기 전까지는 element의 수를 알 수 없으므로 AppController의 생성자에도 setter를 이용해 새로 추가된 _pairwiseDisjointSets를 Null로 초기화 해줍니다. 이후에 initCycleDetection method를 통해서 사용자의 입력을 받은 시점에 다시 초기화되어 사용됩니다.

addEdgeDoesExist method는 그래프를 생성하는 과정에서 추가될 edge를 받아 해당 edge가 추가되었을 때 cycle이 형성되는지 여부를 판단하여 return 합니다. edge의 tailVertex와 headVertex에 대해 find 메소드를 수행하여 어느 set에 속하는지를 찾아낸 후 동일한 set에 존재할 경우 사이클이 형성됨을 의미하므로 true를 반환하고, 이외의 경우 두 element가 속한 set에 대해 union을 수행하고 false를 반환합니다. AppController에서 사용자의 입력을 받아 그래프를 만드는 과정을 수행하는 inputAndMakeGraph method에서 사용되어 edge가 추가될 때마다 cycle detection을 수행하는 방식으로 이용되게 됩니다.

AppController - Cycle Detection

* inputAndMakeGraph에 cycle detection

기존에 그래프 정보를 입력받는 기능을 수행했던 inputAndMakeGraph 메서드에 사용자가 edge를 추가할 때마다 그래프에 사이클이 형성되는지 판단하는 부분이 추가되었습니다.

addDoesMakeCycle method를 통해 추가될 edge를 통해 사이클 형성 여부를 반환해주고 그에 따라 사이클이 존재하게 되면 사용자에게 사이클이 만들어졌음을 출력하여 나타내게 됩니다.

```
private void inputAndMakeGraph() {
    AppView.outputLine("> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다: ");
    int numberOfVertices = this.inputNumberOfVertices();
    this.setGraph(new AdjacencyMatrixGraph(numberOfVertices));

    int numberOfEdges = this.inputNumberOfEdges();
    AppView.outputLine("");
    AppView.outputLine("> 이제부터 edge를 주어진 수 만큼 입력합니다.");
    this.initCycleDetection();
    int edgeCount = 0;
    while (edgeCount < numberOfEdges) { /* 현재까지 추가된 edge 개수인 edgeCount가 numberOfEdges가 되기전까지만 수행 */
        Edge edge = this.inputEdge();
        if (this.graph().edgeDoesExist(edge)) {
            AppView.outputLine(
                    "(오류) 입력된 edge (" + edge.tailVertex() + "," + edge.headVertex() + ") 는 그래프에 이미 존재합니다.");
        } else {
            edgeCount++;
            this.graph().addEdge(edge);
            AppView.outputLine("!새로운 edge (" + edge.tailVertex() + "," + edge.headVertex() + ") 가 그래프에 삽입되었습니다.");
            if (this.addEdgeDoesMakeCycle(edge)) {
                AppView.outputLine("![Cycle] 삽입된 edge (" + edge.tailVertex() + "," + edge.headVertex()
                        + ") 는 그래프에 사이클을 만들었습니다.");
            }
        }
    }

}
```

결과분석

- 결과1. 그래프 입력과 Cycle Detection
- 결과2. 오류 검사

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

실행결과 및 분석: 사이클 검사

```
<terminated> _Main_AL03_201701975_구건모 [Java Application] C:₩Users₩gmku
<<< 입력되는 그래프의 사이클 검사를 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 5
? Edge 수를 입력하시오: 6

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 1
!새로운 edge (0,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 2
!새로운 edge (1,2) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 0
!새로운 edge (2,0) 가 그래프에 삽입되었습니다.
![Cycle] 삽입된 edge (2,0) 는 그래프에 사이클을 만들었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 4
? head vertex 수를 입력하시오: 1
!새로운 edge (4,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 3
!새로운 edge (2,3) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 4
3? head vertex 수를 입력하시오:
!새로운 edge (4,3) 가 그래프에 삽입되었습니다.
![Cycle] 삽입된 edge (4,3) 는 그래프에 사이클을 만들었습니다.

> 입력된 그래프는 다음과 같습니다:
[0] -> 1 2
[1] -> 0 2 4
[2] -> 0 1 3
[3] -> 2 4
[4] -> 1 3

<<< 그래프의 입력과 사이클 검사를 종료합니다 >>>
```

*** 그래프 입력과 사이클 검사**

왼쪽의 실행결과를 살펴보면 AppController의 inputAndMakeGraph method가 그래프의 입력을 받고 생성하는 과정을 보실 수 있습니다. 이때 사용자에게 과정이 보이지는 않지만, edge가 생성될 때마다 해당 edge의 headVertex와 tailVertex가 어느 set에 속했는데 find metod로 조사하여 cycle을 형성하는지 판단하게 되고, 같은 set이 아닐경우 union을 수행하게 됩니다.

처음에는 모두 singleton set으로 존재하는 상태입니다.

처음에 사용자가 Edge(0,1)을 추가하게되면 서로 다른 set에 속하므로 0과 1이 속한 set이 union되고 Edge (1,2)를 추가했을 때에도 마찬가지로 1과 2가 다른 set에 속하므로 두 원소가 속한 set들이 union 이 됩니다. 그러면 지금까지 union을 통해 {0,1,2} set이 만들어지게 되는데 이때 Edge(2,0)을 추가하게 되면 0과 2가 같은 set에 이미 속하고 있으므로 해당 edge가 cycle을 만들었다는 메시지가 출력됩니다. 동일한 방법으로 Edge(4,1)을 추가하게되면 동일한 set이 아니므로 union 되어 {0,1,2,4} set이 되고, (2,3)을 추가하면 union 되어 {0,1,2,3,4} set이 만들어집니다. 이때 Edge(4,3)을 추가하게 되면 4와 3이 동일한 set에 속하므로 사이클을 만들었다는 문구가 정상적으로 출력되고 있음을 확인해 볼 수 있습니다.

또한 cycle detection 과정에서 호출되는 union, find method는 각각 weighting rule과 collapsiong rule을 적용하여 구현하였으므로 과정이 진행됨에 따라 내부적으로 tree의 높이가 낮아지는 방향성을 가지게 되어, find 시 tree 높이가 높아서 발생할 수 있는 비효율을 낮추게 됩니다.

실행결과 및 분석: 오류검사

```
_Main_AL03_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p
<<< 입력되는 그래프의 사이클 검사를 시작합니다 >>>
> 입력할 그래프의 vertex 수와 edge 수를 먼저 입력해야 합니다:
? Vertex 수를 입력하시오: 5
? Edge 수를 입력하시오: 6

> 이제부터 edge를 주어진 수 만큼 입력합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: -1
? head vertex 수를 입력하시오: 5
[오류] 존재하지 않는 tail vertex 입니다: -1
[오류] 존재하지 않는 head vertex 입니다: 5
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: #0
(오류) tail vertex 수 입력에 오류가 있습니다: #0
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: xxx
(오류) head vertex 수 입력에 오류가 있습니다: xxx
? head vertex 수를 입력하시오: 1
!새로운 edge (0,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 1
? head vertex 수를 입력하시오: 0
(오류) 입력된 edge (1,0) 는 그래프에 이미 존재합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 2
[오류] 두 vertex 번호가 동일합니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 2
? head vertex 수를 입력하시오: 1
!새로운 edge (2,1) 가 그래프에 삽입되었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오: 0
? head vertex 수를 입력하시오: 2
!새로운 edge (0,2) 가 그래프에 삽입되었습니다.
![Cycle] 삽입된 edge (0,2) 는 그래프에 사이클를 만들었습니다.
- 입력할 edge의 두 vertex를 차례로 입력해야 합니다:
? tail vertex 를 입력하시오:
```

*** 비정상상황에 대한 오류 메시지 출력 및 예외 처리**

vertex 추가시 존재여부 판단

좌측의 예외처리 결과 이미지를 보시면 사용자가 vertex 의 개수를 5개 선언한 경우이므로 여기서 vertex는 0부터 4까지의 vertex가 존재합니다. tailVertex값이 -1이 들어간 경우 음수값을 가지는 vertex는 존재하지 않으므로 존재하지 않는 tailVertex 라고 출력됩니다. headVertex로 5를 추가하려고 하는 경우 또한 사용자가 초기에 선언한 vertex의 개수를 넘어가는 영역이므로 존재하지 않는 vertex 이고, 오류 메시지를 정상적으로 출력하는 것을 볼 수 있습니다.

type 오류 예외처리

또한 입력한 vertex 값 자체가 int type이 아닌 경우는 입력오류이므로 try-catch 구문을 통해 예외처리되며 에러메시지를 출력하는 것을 보실 수 있습니다.

적절하지 않은 Edge 추가를 방지

Edge의 tail,head vertex 값이 동일한 경우도 추가되지 않도록 되어 있으며 이미 그래프에 존재하는 경우도 오류로 인식하여 오류 메시지를 출력하게 됩니다.

사용자의 정상적인 입력을 통해서만 그래프가 형성되며 형성된 그래프에 대해서 매 edge가 추가될 때마다 cycle detection이 수행되고 있음을 확인해 볼 수 있습니다.

과제 질문에 대한 답변

- 과제에서 해결해야할 질문들

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제 안에 질문에 대한 답변- 생각할 점

*** MVC 모델은 잘 지켜졌는지?**

전반적으로 MVC 모델이 잘 지켜지고 있다고 생각합니다. 이번 과제에서는 cycle-detection을 위해 pairwiseDisjointSets 클래스를 구현하였는데 해당 클래스에 정의된 union과 find를 Controller의 역할을 수행하는 AppController가 이용하여 실질적인 기능을 사용하고 결과를 출력할 때 또한 AppController에서 View에 해당하는 AppView의 메서드를 이용하여 출력하게 됩니다. model과 view가 서로의 기능을 알지 못하며 서로 의존성을 가지지 않고 controller가 두 model, view를 매개하는 기능을 수행하고 있으므로 MVC 모델이 잘 지켜지고 있다고 생각합니다.

*** 캡슐화는 잘 되었는지?**

이번 과제의 PairwiseDisjointSets 클래스에서는 AppController에서 직접 사용할 union , find method와 생성자를 제외하고는 전부 private으로 선언되어 있어 직접 접근을 할 수 없도록 은닉화 되어있습니다.

뿐만 아니라 기존의 Class들을 살펴보면 해당 Class 에서만 사용하는 멤버변수를 private으로 선언하여 변수를 은닉하고 데이터를 조회,변경하는 등 외부에서 어떠한 이유로 접근할 경우 public으로 선언된 getter/setter를 통해서만 변수에 접근할 수 있도록 하여 변수값을 직접 건드리지 못하도록 하였으므로 캡슐화가 잘 이루어지고 있다고 생각하였습니다.

***Adjacency Matrix로 그래프를 표현하는 방법**

Adjacency Matrix는 tailVertex와 headVertex를 각각 Matrix의 index 로 하여 해당 위치에 연결관계를 나타냅니다. 실습에서는 초기에 Matrix 값을 모두 0으로 초기화 해준 뒤, 사용자가 edge를 추가함에 따라 Matrix에서 추가한 Edge의 tailVertex와 headVertex의 위치에 해당하는 Matrix 값을 1로 설정하여 연결관계를 표현하는 식으로 그래프를 표현하게 됩니다.

*** 그래프 보여주는 방법**

현재 그래프를 출력해주는 showGraph 메서드는 구현 종속적 입니다. 따라서 Matrix가 아닌 list로 자료구조가 바뀔경우 다시 구현해줘야 한다는 등의 문제가 있습니다. 따라서 Iterator를 도입하여 구현 종속적이지 않도록 바꿔줄 필요성이 있다고 생각합니다.

*** PairwiseDisjointSet의 cycle 검사 방식과 union-find 알고리을 이용한 구현방법:**

pairwiseDisjointSets는 그래프 컴포넌트들의 연결관계를 배열로 저장하고 find 메소드를 이용하여 cycle이 형성되었는지 확인합니다. 초기에는 element의 개수만큼의 singleton set으로 초기화 된 후 edge 가 추가될 때마다 find를 통해 union을 수행하거나 사이클이 형되었는지 여부를 판단합니다. 이 과정들은 union과 find를 각각 weighting rule과 collasping rule을 적용하여 tree 높이가 낮아지는 방향성을 추구하도록 구현되었습니다.

과제를 통해 느낀점

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

과제를 통해 느낀점

pairwiseDisjointSets를 이용하여 cycle의 형성 유무를 판단하는 코드를 구현하는 과정 중에 union과 find에 대해 각각 weighting rule을 적용한 union과 collasping rule을 적용한 find를 구현해보면서 weighting rule은 두 set을 union 할 때에 어떻게 tree를 연결하냐에 따라 높이가 달라지게 되고 tree의 높이가 find 과정에 비효율을 야기하므로 root 를 의미하는 배열값에 해당 set이 가진 원소수를 반영하여 비교하고 find 과정을 수행한 후에 find 도중 거쳤던 모든 노드들을 root의 child로 지정하여 지나온 경로의 높이를 낮춰주는 방식들이나 배열을 통해 set의 root와 member의 포함관계를 표현하고 다루는 방식들이 굉장히 새로웠던 것 같습니다. 기존에 배웠던 자료구조들을 어떻게 활용하냐에 따라 굉장히 무궁무진하게 알고리즘을 다룰 수 있는 것 같다고 느꼈던 실습이었습니다.

감사합니다!

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com

감사합니다

CNU, Computer Science & Engineering
StuNo. 201701975, stuName. Kugunmo
kgm9179@naver.com