YILDIZ TEKNİK ÜNİVERSİTESİ

KİMYA-METALÜRJİ FAKÜLTESİ

MATEMATİK MÜHENDİSLİĞİ BÖLÜMÜ

Tasarım Uygulamaları Çalışması

Çizgi Kuramı Algoritmaları Ve Algoritmaların Uygulamaları

Tez Yöneticisi: Dr. Öğr. Üyesi Mert Bal

Öğrencinin Adı, Soyadı, Numarası:

İbrahim Bayburtlu 19052017

İstanbul 2024

İÇİNDEKİLER Sayfa

ÖNSÖZ

Bu çalışma, çizgi kuramı ve algoritmalarının implementasyonlarına odaklanarak bir tasarım uygulaması sunmayı amaçlamaktadır. Çizgi kuramı, matematiksel bir dal olarak karmaşık sistemlerin analizinde ve modellemesinde yaygın olarak kullanılmaktadır. Bu alandaki temel kavramlar ve algoritmalar, grafiklerin (networklerin) temsil edilmesi, analizi ve üzerinde operasyonlar gerçekleştirilmesi gibi birçok alanda uygulama bulmaktadır.

Bu tez, çizgi kuramının temel prensiplerini ele alarak, bu prensiplerin pratikte nasıl uygulanabileceğini göstermektedir. Ayrıca, çeşitli çizgi kuramı algoritmalarının nasıl tasarlandığı ve uygulandığına dair detaylı bir inceleme sunmaktadır. Okuyucular, bu çalışma sayesinde çizgi kuramı ve algoritmalarının temel kavramlarını anlamakla kalmayacak, aynı zamanda bu kavramları gerçek dünya problemlerine nasıl uygulayabileceklerini de öğreneceklerdir.

Bu çalışmanın hazırlanmasında birçok kişiye teşekkür borçluyum. Öncelikle, danışmanım Mert Bal'a sonsuz teşekkürlerimi sunarım; sabrı, rehberliği ve destekleri olmadan bu çalışma tamamlanamazdı.

Son olarak, aileme ve sevdiklerime, bu süreç boyunca verdikleri moral ve manevi destek için minnettarım. Onların sevgisi ve güveni, bu tezin tamamlanmasında büyük bir motivasyon kaynağı oldu.

Umarım bu çalışma, çizgi kuramı ve algoritmalarıyla ilgilenen herkes için yararlı olur ve bu alanda daha fazla araştırmanın kapılarını aralar.

Saygılarımla,

Özet

Bu tez, çeşitli graf algoritmalarının incelenmesi ve bu algoritmaların günlük hayattaki kullanım alanlarının araştırılması amacıyla hazırlanmıştır. Graf algoritmaları, bilgisayar bilimlerinde ve pratik problemlerin çözümünde yaygın olarak kullanılan önemli araçlardır. Bu çalışmada, Depth First Search (DFS), Breadth First Search (BFS), Topological sort, Minimum Spanning Tree, Prim's Algorithm, Kruskal's Algorithm, Bellman-Ford Algorithm, Dijkstra's Algorithm (Shortest Path Algorithm), A* Search Algorithm, Floyd-Warshall Algorithm, Ford-Fulkerson Algorithm for Maximum Flow ve Greedy Algorithm for Graph Colouring gibi önemli algoritmalar ele alınmıştır.
Bu algoritmaların sırasıyla:

- Nedir?
- Ne işe yarar?
- Hangi sorunu çözer
- Günlük hayat problemlerinde kullanımı
- Örnek Java implementasyonu

gibi sorulara cevaplar verilmiştir.

Sonuçlar bölümü, graf algoritmalarının temel kavramlarını, işleyişlerini, günlük hayattaki kullanım alanlarını ve örnek uygulamalarını açıklayarak okuyuculara geniş bir bakış açısı sunmayı hedeflemektedir.

Abstract

The thesis examines various graph algorithms and investigates their applications in everyday life. Graph algorithms are essential tools widely used in computer science and practical problem-solving. In this study, important algorithms such as Depth First Search (DFS), Breadth First Search (BFS), Topological sort, Minimum Spanning Tree, Prim's Algorithm, Kruskal's Algorithm, Bellman-Ford Algorithm, Dijkstra's Algorithm (Shortest Path Algorithm), A* Search Algorithm, Floyd-Warshall Algorithm, Ford-Fulkerson Algorithm for Maximum Flow, and Greedy Algorithm for Graph Colouring are discussed. These algorithms are addressed in the following order:

- What is it?
- What is its purpose?
- Which problem does it solve?
- Usage in everyday life problems
- Sample Java implementation

The conclusion section aims to provide readers with a comprehensive overview of graph algorithms, covering their fundamental concepts, workings, practical applications in daily life, and example implementations.

Giriş

Çizgi kuramı, matematik ve bilgisayar bilimlerinde çizgilerin ve ağların özelliklerini inceleyen bir dalıdır. Bu alanda, çizgilerin yapısı, çizgiler arasındaki bağlantılar, ağların yapısı ve benzeri konular üzerine çalışılır. Çizgi kuramı, ağların optimizasyonu, iletişim ağlarının tasarımı, sosyal ağ analizi gibi birçok alanda kullanılmaktadır.

Algoritma, bir problemi çözmek veya belirli bir görevi yerine getirmek için adım adım talimatlar dizisi olarak tanımlanabilir. Algoritmalar, bilgisayar bilimleri, mühendislik, matematik ve diğer birçok disiplinde karşımıza çıkar. Algoritmalar, veri işleme, yapay zeka, sıralama, arama ve optimizasyon gibi birçok alanda kullanılır.

Çizgi kuramı ve algoritmaları, gerçek hayatta karşılaşılan birçok karmaşık problemi çözmek için kullanılır. Bu alanlar, ulaşım ve lojistik problemlerinden sosyal ağ analizine, telekomünikasyondan yapay zeka ve makine öğrenime kadar geniş bir yelpazede uygulama alanı bulmuştur.

Örneğin, ulaşım ve lojistik sektöründe, en kısa yol problemi ve maksimum akış problemleri gibi algoritmalar, navigasyon sistemlerinin, kargo taşımacılığının ve ulaşım planlamasının optimize edilmesinde kritik bir rol oynamaktadır. Bu algoritmalar, bir noktadan diğerine en kısa ve en etkili yolu bulmayı sağlar, böylece zaman ve kaynak tasarrufu sağlar.

Sosyal ağ analizi alanında, çizgi kuramı, sosyal medya ağlarının analizinde ve kullanıcı etkileşimlerinin anlaşılmasında kullanılır. Bu, pazarlama stratejilerinin ve sosyal medya kampanyalarının etkili bir şekilde oluşturulması ve yönetilmesi için önemlidir.

- Çalışmada kullanılan yöntemler ve bu yöntemlerin seçiliş sebebi açıklanmalıdır.
- Okuyucunun hangi bölümleri okuyacağına karar vermesini kolaylaştırmak için çalışmanın sonraki bölümleri kısaca tanıtılmalıdır.

Graph Algoritmalar

Depth First Search (DFS) Temel Adımlar

Depth First Search (DFS), bir graf (çizge) yapısındaki düğümleri (noktaları) keşfetmek için kullanılan bir graf tarama algoritmasıdır. DFS, bir düğümden başlayarak, derinlemesine ilerler ve mümkün olduğu kadar derine gider. Eğer ilerleme mümkün değilse, geri dönüp daha önce ziyaret edilmemiş bir düğüme yönelir.

DFS, genellikle yığın (stack) veri yapısını kullanarak uygulanır. Yığıtın yardımıyla, bir düğümün tüm komşularını ziyaret ettikten sonra geri dönüp daha önce ziyaret edilmemiş diğer düğümleri keşfetmeye devam eder.

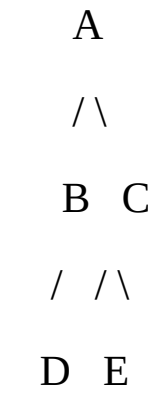
DFS algoritması, grafın yapısını ve özelliklerini analiz etmek, yol bulma, dolaşma, bağlılık kontrolü, topolojik sıralama gibi birçok graf işlemi için kullanılır. Ayrıca, ağların keşfi, yaprak düğüm bulma, bağlantı bileşenleri ve döngüleri tespit etme gibi birçok uygulamada da kullanılır.

DFS algoritmasının temel özellikleri şunlardır:

- **Derinlemesine Tarama :** DFS, bir düğümün tüm dallarını derinlemesine tarar. Yani, bir düğümün tüm komşularını ziyaret ettikten sonra, bu komşuların komşularını ziyaret eder ve bu şekilde devam eder.

- **Yığıt Kullanımı :** DFS algoritması genellikle bir yığıt (stack) veri yapısını kullanarak uygulanır. Bu yığıt, ziyaret edilmemiş düğümleri saklamak ve geri dönüşlerde kullanılmak üzere düğümleri depolamak için kullanılır.

- **Ziyaret Edilen Düğüm İşaretleme :** DFS, bir düğümü ziyaret ettikten sonra, bu düğümü ziyaret edildi olarak işaretler. Bu sayede, aynı düğümün tekrar ziyaret edilmesi engellenir.



Başlangıç düğümü olarak A alırsak:

- A, kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- A'nın komşuları B ve C'dir. Bu düğümler kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- Şimdi kuyruktan B çıkarılır. B'nin komşusu D'dir. D, kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- Aynı şekilde C için, E ve F komşuları kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.

Deep First Search Algoritması Uygulaması

```java
class DeepFirstSearchAlgorithm {
    private final int numVertices;
    private final LinkedList<Integer>[] adjacencyList;

    DeepFirstSearchAlgorithm(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; ++i)
            adjacencyList[i] = new LinkedList<>();
    }

    void addEdge(int source, int destination) {
        adjacencyList[source].add(destination);
    }

    void DFSUtil(int vertex, boolean visited[]) {
        visited[vertex] = true;
        System.out.print(vertex + " ");

        for (int neighbor : adjacencyList[vertex]) {
            if (!visited[neighbor])
                DFSUtil(neighbor, visited);
        }
    }

    void DFS(int startingVertex) {
        boolean visited[] = new boolean[numVertices];

        DFSUtil(startingVertex, visited);
    }
```

```
    public static void main(String args[]) {
        DeepFirstSearchAlgorithm graph = new DeepFirstSearchAlgorithm( numVertices: 4);

        graph.addEdge( source: 0,  destination: 1);
        graph.addEdge( source: 0,  destination: 2);
        graph.addEdge( source: 1,  destination: 2);
        graph.addEdge( source: 2,  destination: 0);
        graph.addEdge( source: 2,  destination: 3);
        graph.addEdge( source: 3,  destination: 3);

        System.out.println("Depth First Traversal starting from vertex 2:");
        graph.DFS( startingVertex: 2);
    }
}
```

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

DepthFirstSearch sınıfı, bir grafı temsil etmek için kullanılır. Sınıf, grafın kenarlarını bir LinkedList dizisi olarak tutar.

addEdge metodu, grafa bir kenar eklemek için kullanılır. Kenarlar, bağlı olduğu düğümün komşuluk listesine eklenir.

DFSUtil metodu, derinlik öncelikli arama algoritmasının ana mantığını uygular. Bu metod, verilen bir başlangıç düğümünden başlayarak grafı derinlik öncelikli olarak dolaşır.

DFS metodu, dolaşımı başlatmak için kullanılır. Bu metod, bir başlangıç düğümünü alır ve derinlik öncelikli arama algoritmasını çağırarak dolaşımı başlatır.

main metodu, bir graf oluşturur, kenarları ekler ve derinlik öncelikli arama algoritmasını çağırarak dolaşımı başlatır. Bu kod, verilen bir grafı derinlik öncelikli arama algoritması kullanarak dolaşarak sonuçları konsola yazdırır.

Breadth First Search (BFS)

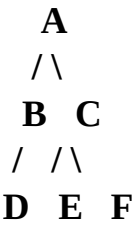
Breadth First Search (BFS), bir graf (çizge) yapısındaki düğümleri (noktaları) seviye keşfetmek için kullanılan bir graf tarama algoritmasıdır. BFS, bir düğümden başlayarak, o düğümün komşularını keşfeder, ardından komşularının komşularını keşfeder ve bu şekilde grafın seviye tüm düğümlerini ziyaret eder.

BFS, genellikle kuyruk (queue) veri yapısını kullanarak uygulanır. Kuyruğun yardımıyla, bir düğümün tüm komşularını ziyaret ettikten sonra sırayla komşularını keşfetmeye devam eder.

BFS algoritması, grafın yapısını ve özelliklerini analiz etmek, en kısa yol bulma, bağlılık kontrolü, genişlik bazında ağları keşfetme gibi birçok graf işlemi için kullanılır. Ayrıca, ağların keşfi, minimum genişlik ilk arama ağaçları oluşturma gibi birçok uygulamada da kullanılır

BFS algoritmasının temel özellikleri şunlardır:

- **Genişlik Bazında Tarama :** BFS, bir düğümün tüm komşularını ziyaret ettikten sonra, bu komşuların komşularını ziyaret eder. Yani, bir düğümden başlayarak, o düğümün seviyesindeki tüm düğümleri ziyaret eder.
- **Kuyruk Kullanımı :** BFS algoritması genellikle bir kuyruk (queue) veri yapısını kullanarak uygulanır. Bu kuyruk, ziyaret edilmemiş düğümleri saklamak ve sırayla düğümleri keşfetmek için kullanılır.
- **Ziyaret Edilen Düğüm İşaretleme :** BFS, bir düğümü ziyaret ettikten sonra, bu düğümü ziyaret edildi olarak işaretler. Bu sayede, aynı düğümün tekrar ziyaret edilmesi engellenir.



Başlangıç düğümü A olarak alalım:

- A, kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- A'nın komşuları B ve C'dir. Bu düğümler kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- Şimdi kuyruktan B çıkarılır. B'nin komşusu D'dir. D, kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.
- Aynı şekilde C için, E ve F komşuları kuyruğa eklenir ve ziyaret edildi olarak işaretlenir.

Bu süreç, kuyrukta düğüm kalmayana kadar devam eder. Sonuç olarak, düğümler A, B, C, D, E ve F sırasıyla ziyaret edilir ve keşfedilir.

Bu şekilde, BFS algoritması, grafın tüm düğümlerini seviye keşfeder ve genellikle en kısa yol bulma, genişlik bazında ağ analizi ve graf teorisi problemlerini çözmek için kullanılır.

Breadth First Search Algoritması Uygulaması

```java
class BreadthFirstSearch {
    private int V;
    private LinkedList<Integer> adj[];

    BreadthFirstSearch(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    void addEdge(int v, int w) {
        adj[v].add(w);
    }


    void BFS(int s) {
        boolean visited[] = new boolean[V];


        LinkedList<Integer> queue = new LinkedList<>();

        visited[s] = true;
        queue.add(s);

        while (queue.size() != 0) {
            s = queue.poll();
            System.out.print(s + " ");

            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
```

```java
    public static void main(String[] args) {
        BreadthFirstSearch g = new BreadthFirstSearch( v: 4);

        g.addEdge( v: 0,  w: 1);
        g.addEdge( v: 0,  w: 2);
        g.addEdge( v: 1,  w: 2);
        g.addEdge( v: 2,  w: 0);
        g.addEdge( v: 2,  w: 3);
        g.addEdge( v: 3,  w: 3);

        System.out.println("Başlangıç düğümü 2 olan BFS:");
        g.BFS( s: 2);
    }
```

```
Başlangıç düğümü 2 olan BFS:
2 0 3 1
```

İlk olarak, BreadthFirstSearch sınıfı oluşturulur ve grafa komşuluk listesini temsil etmek için bir LinkedList dizisi oluşturulur.

Ardından, addEdge metodu, grafa bir kenar eklemek için kullanılır.

BFS metodu, genişlik öncelikli arama algoritmasını uygular. Bu metot, verilen bir başlangıç düğümünden başlayarak grafı dolaşır ve düğümleri genişliğe göre tarar.

Son olarak, main metodu, bir graf oluşturur, kenarları ekler ve genişlik öncelikli arama algoritmasını çağırarak dolaşımı başlatır.

Bu kod, verilen bir grafı genişlik öncelikli arama algoritması kullanarak dolaşarak sonuçları konsola yazdırır.

Topological sort

Topolojik Sıralama (Topological Sort), bir grafın düğümlerini, grafın yönlerine uygun olarak bir sıraya koyma işlemidir. Topolojik sıralama, genellikle yalın bir grafda (yönlü ağaçta) döngü olmadığı durumlarda kullanılır. Bu tür bir sıralama, bir düğümün bağımlılıklarını göz önünde bulundurarak, bağımlılık ilişkilerini dikkate alarak düğümleri sıralar.

Topolojik Sıralamanın Özellikleri :

- **Yönlü Graf Gerekliliği:** Topolojik sıralama, yalnızca yönlü graf (grafda çizgilerin yönleri olan graf) üzerinde tanımlıdır.
- **Döngü Kontrolü :** Döngüler (cycle) içermeyen graf (DAG - Directed Acyclic Graph) için geçerlidir. Döngüler içeren bir graf için topolojik sıralama yapmak mümkün değildir.
- **Bağımlılık Sıralaması :** Topolojik sıralama, bir düğümün bağımlılıklarını göz önünde bulundurarak, bağımlılık ilişkilerini dikkate alarak düğümleri sıralar.

Algoritma Adımları :

- Başlangıçta bağımsız(diğer düğümlerle bağlı olmayan) düğümleri bulun.
- Bu bağımsız düğümleri sırasıyla çıkarın ve sonuç listesine ekleyin.
- Düğümü kaldırtıktan sonra,düğüm komşularını göz önünde bulundurun ve bağlantıları kırın(düğümün bağımlılıklarını kaldırın).
- Yine bağımsız düğümleri bulun ve süreci tekrarlayın.

Bu grafın topolojik sıralaması şu şekildedir: C, D, A, B, E veya D, C, A, B, E olarak olabilir. Döngü içermediği için bu grafın topolojik sıralaması mevcuttur.

Topological Sort Algoritması Uygulaması

```
class TopologicalSort {
    private int V;
    private LinkedList<Integer> adj[];
    TopologicalSort(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }
    void addEdge(int v, int w) {
        adj[v].add(w);
    }
    void topologicalSortUtil(int v, boolean visited[], Stack<Integer> stack) {

        visited[v] = true;
        Integer i;


        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext()) {
            i = it.next();
            if (!visited[i])
                topologicalSortUtil(i, visited, stack);
        }

        stack.push(new Integer(v));
    }
```

```
    void topologicalSort() {
        Stack<Integer> stack = new Stack<Integer>();


        boolean visited[] = new boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, stack);

        while (stack.empty() == false)
            System.out.print(stack.pop() + " ");
    }

    public static void main(String args[]) {
        // Bir örnek graf oluşturulur
        TopologicalSort g = new TopologicalSort( v: 6);
        g.addEdge( v: 5,  w: 2);
        g.addEdge( v: 5,  w: 0);
        g.addEdge( v: 4,  w: 0);
        g.addEdge( v: 4,  w: 1);
        g.addEdge( v: 2,  w: 3);
        g.addEdge( v: 3,  w: 1);

        System.out.println("Topolojik sıralama:");
        g.topologicalSort();
    }
}
```

```
Topolojik sıralama:
5 4 2 3 1 0
```

TopologicalSort sınıfı, bir grafı temsil etmek için kullanılır. Sınıf, grafın kenarlarını bir LinkedList dizisi olarak tutar.

addEdge metodu, grafa bir kenar eklemek için kullanılır. Kenarlar, bağlı olduğu düğümün komşuluk listesine eklenir.

topologicalSortUtil metodu, topolojik sıralamayı bulmak için kullanılır. Bu metot, derinlik öncelikli arama mantığına benzer şekilde, ziyaret edilmemiş düğümleri ziyaret eder ve bunların komşularını keşfeder. Ardından, bir düğümün işlenmesini tamamladığında, onu bir yığın veri yapısına iter.

topologicalSort metodu, topolojik sıralamayı başlatmak için kullanılır. Bu metot, grafı dolaşarak topolojik sıralamayı bulur ve bulunan sırayı bir yığın kullanarak saklar. Daha sonra, yığındaki düğümleri sırayla çıkararak sıralamayı konsola yazdırır.

main metodu, bir graf oluşturur, kenarları ekler ve topolojik sıralamayı bulmak için topologicalSort metodu çağrılır. Bu kod, verilen bir grafın topolojik sıralamasını bulur ve sonuçları konsola yazdırır.

Minimum Spanning Tree

Minimum Spanning Tree (MST), bir ağırlıklı grafın (weighted graph) tüm düğümlerini içeren ve toplam ağırlığı en küçük olan alt grafı ifade eder. Minimum Spanning Tree, genellikle ağlarda bağlantı maliyetlerini minimize etmek veya ağdaki tüm düğümleri birbirine bağlamak için kullanılır.

Minimum Spanning Tree'nin Özellikleri:

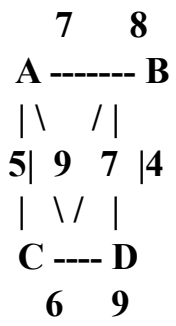
- **Ağırlıklı Graf:** Minimum Spanning Tree, ağırlıklı graf (weighted graph) üzerinde tanımlıdır. Her bir kenarın (çizginin) bir ağırlığı (weight) vardır.
- **Bağlantıları Kaldırma:** Minimum Spanning Tree, grafın tüm düğümlerini bağlamak için gerekli olan kenarları seçerken, döngü oluşturmadan kenarları kaldırır.
- **En Küçük Toplam Ağırlık:** Minimum Spanning Tree, seçilen kenarların toplam ağırlığı en küçük olacak şekilde tasarlanır. Bu, ağdaki toplam bağlantı maliyetini minimize eder.

Algoritma Yöntemleri:

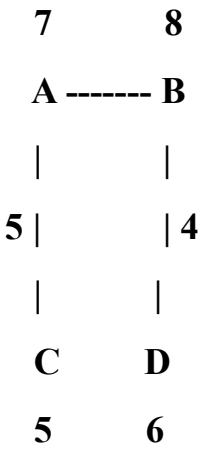
Bir grafın Minimum Spanning Tree'sini bulmak için birçok algoritma kullanılabilir, en yaygın olanları:

- **Kruksal Algoritması:** Kenarları ağırlıklarına göre sıralar ve en küçük ağırlıklı kenarları ekleyerek Minimum Spanning Tree oluşturur. Bu algoritma, greedy (açgözlü) yaklaşıma dayanır.

- **Prim Algoritması:** Başlangıç düğümünden başlayarak, her adımda ağırlığı en küçük kenarı ekler ve bu kenara bağlı yeni düğümleri keşfeder.
- Bu algoritma da greedy yaklaşıma dayanır.



Bu grafın Minimum Spanning Tree'si şu şekildedir:



Toplam ağırlık = 7 + 8 + 4 + 5 + 6 = 30

Minimum Spanning Tree, ABD üçgenini ve CB, BD kenarlarını içerir. Bu durumda toplam ağırlık 30'dur.

Minimum Spanning Tree, telekomünikasyon ağları, su ve elektrik şebekeleri, bilgisayar ağları ve ulaşım sistemleri gibi birçok uygulamada kullanılır. Bu algoritma, ağdaki toplam bağlantı maliyetini minimize etmek ve kaynakları verimli bir şekilde kullanmak için önemlidir.

Minimum Spanning Tree Algoritması Uygulaması

```
public class KruskalsMST {
    static class Edge {
        int src, dest, weight;
        public Edge(int src, int dest, int weight)
        {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }
    static class Subset {
        int parent, rank;

        public Subset(int parent, int rank)
        {
            this.parent = parent;
            this.rank = rank;
        }
    }
```



```
  public static void main(String[] args)
  {
      int V = 4;
      List<Edge> graphEdges = new ArrayList<Edge>(
              List.of(new Edge( src: 0,  dest: 1,  weight: 10), new Edge( src: 0,  dest: 2,  weight: 6),
                      new Edge( src: 0,  dest: 3,  weight: 5), new Edge( src: 1,  dest: 3,  weight: 15),
                      new Edge( src: 2,  dest: 3,  weight: 4)));
      graphEdges.sort(new Comparator<Edge>() {
          @Override public int compare(Edge o1, Edge o2)
          {
              return o1.weight - o2.weight;
          }
      });
      kruskals(V, graphEdges);
  }
```

```java
private static void kruskals(int V, List<Edge> edges)
{
    int j = 0;
    int noOfEdges = 0;
    Subset subsets[] = new Subset[V];
    Edge results[] = new Edge[V];
    for (int i = 0; i < V; i++) {
        subsets[i] = new Subset(i, rank: 0);
    }
    while (noOfEdges < V - 1) {
        Edge nextEdge = edges.get(j);
        int x = findRoot(subsets, nextEdge.src);
        int y = findRoot(subsets, nextEdge.dest);
        if (x != y) {
            results[noOfEdges] = nextEdge;
            union(subsets, x, y);
            noOfEdges++;
        }
        j++;
    }
    System.out.println(
            "Following are the edges of the constructed MST:");
    int minCost = 0;
    for (int i = 0; i < noOfEdges; i++) {
        System.out.println(results[i].src + " -- "
                + results[i].dest + " == "
                + results[i].weight);
        minCost += results[i].weight;
    }
    System.out.println("Total cost of MST: " + minCost);
```

```
Following are the edges of the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Total cost of MST: 19
```

```
    private static void union(Subset[] subsets, int x,
                              int y)
    {
        int rootX = findRoot(subsets, x);
        int rootY = findRoot(subsets, y);

        if (subsets[rootY].rank < subsets[rootX].rank) {
            subsets[rootY].parent = rootX;
        }
        else if (subsets[rootX].rank
                 < subsets[rootY].rank) {
            subsets[rootX].parent = rootY;
        }
        else {
            subsets[rootY].parent = rootX;
            subsets[rootX].rank++;
        }
    }
    private static int findRoot(Subset[] subsets, int i)
    {
        if (subsets[i].parent == i)
            return subsets[i].parent;

        subsets[i].parent
                = findRoot(subsets, subsets[i].parent);
        return subsets[i].parent;
    }
}
```

Edge sınıfı, bir kenarı temsil eder ve başlangıç düğümü (src), hedef düğümü (dest) ve ağırlığı (weight) içerir.

Subset sınıfı, bir düğümün bir alt kümesini temsil eder ve kök düğümünü (parent) ve alt kümenin boyutunu (rank) içerir.

main metodu, bir graf oluşturur ve kenarları ağırlıklarına göre sıralar. Ardından, Kruskal algoritmasını çağırarak minimum yayılma ağacını oluşturur.

kruskals metodu, Kruskal algoritmasının ana mantığını uygular. Kenarları ağırlık sırasına göre dolaşır, döngü oluşturmayacak şekilde düğümleri birleştirir ve minimum yayılma ağacını oluşturur.

union metodu, iki alt küme arasında birleştirme işlemi gerçekleştirirken dengeleme stratejisini uygular.

findRoot metodu, bir düğümün kökünü bulurken kök düğümüne kadar ilerler.

Son olarak, oluşturulan minimum yayılma ağacının kenarlarını ve toplam ağırlığını konsola yazdırır.

Bellman-Ford Algoritması

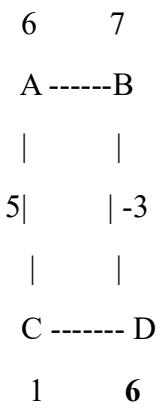
Bellman-Ford Algoritması, tek bir kaynaktan tüm diğer düğümlere olan en kısa yol uzunluklarını hesaplamak için kullanılan bir tek kaynaklı en kısa yol algoritmasıdır. Bu algoritma, negatif ağırlıklı kenarları olan ağırlıklı graf (weighted graph) için de çalışır. Bellman-Ford Algoritması, ağdaki tüm döngüleri tespit eder ve bu döngülerin negatif ağırlık döngüleri olup olmadığını kontrol eder.

Bellman-Ford Algoritmasının Özellikleri:

- **Negatif Ağırlıklı Kenalar :** Bellman-Ford Algoritması, negatif ağırlıklı kenarları olan graf (negatif ağırlıklı kenarlar döngüleri olmamak kaydıyla) için de kullanılabilir.
- **Negatif Ağırlık Döngüsü Tespiti :** Algoritma, negatif ağırlık döngüleri olup olmadığını kontrol eder. Eğer bir döngü negatif ağırlığa sahipse, algoritma bu durumu tespit eder.
- **Tek kaynaklı En Kısa Yol :** Algoritma, tek bir başlangıç düğümünden tüm diğer düğümlere olan en kısa yol uzunluklarını hesaplar**.**

Algoritma Adımları:

- Başlangıçta, tüm düğümlerin en kısa yol uzunluklarını sonsuz (∞) olarak ayarlayın ve başlangıç düğümünün uzunluğunu 0 olarak ayarlayın.
- Tüm kenarları (çizgileri) döngü içinde gözden geçirin.
- Her kenar için,eğer başlangıç düğümünden hedef düğüme olan yol uzunluğu,şu anki yoldan daha kısa ise,yol uzunluğunu güncelleyin.
- Bu adımları toplam düğüm sayısı kadar (n-1) tekrar edin,n grafın düğüm sayısıdır.
- Son olarak,negatif ağırlıklı döngüleri olup olmadığını kontrol edin.



Bu grafın Bellman-Ford Algoritması ile en kısa yolları hesaplanabilir. Başlangıç düğümü olarak A alınırsa:

- **A'dan B'ye: 6**
- **A'dan C'ye: 5**
- **A'dan D'ye: 2**

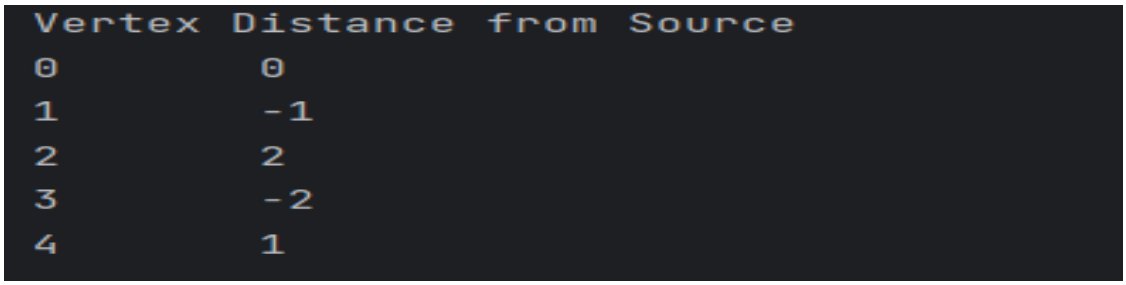
Sonuç olarak,Bellman-Ford Algoritması ile A'dan B'ye olan en kısa yol 6,A'dan C'ye olan en kısa yol 5 ve A'dan D'ye olan en kısa yol 2 olarak bulunur.

Bellman Ford Algoritması Uygulaması

```
class Graph {
    class Edge {
        int src, dest, weight;
        Edge() { src = dest = weight = 0; }
    };
    int V, E;
    Edge edge[];
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i = 0; i < e; ++i)
            edge[i] = new Edge();
    }
```

```
void BellmanFord(Graph graph, int src)
{
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];
    for (int i = 0; i < V; ++i)
        dist[i] = Integer.MAX_VALUE;
    dist[src] = 0;
    for (int i = 1; i < V; ++i) {
        for (int j = 0; j < E; ++j) {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u] != Integer.MAX_VALUE
                    && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
    for (int j = 0; j < E; ++j) {
        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u] != Integer.MAX_VALUE
                && dist[u] + weight < dist[v]) {
            System.out.println(
                    "Graph contains negative weight cycle");
            return;
        }
    }
    printArr(dist, V);
}
```

```
void printArr(int dist[], int V)
{
    System.out.println("Vertex Distance from Source");
    for (int i = 0; i < V; ++i)
        System.out.println(i + "\t\t" + dist[i]);
}
```



Graph sınıfı, bir grafı temsil eder ve içinde bir Edge iç sınıfı bulunur. Bu iç sınıf, bir kenarı (başlangıç düğümü, hedef düğümü ve ağırlığı) temsil eder.

BellmanFord metodu, Bellman-Ford algoritmasının ana mantığını uygular. Algoritma, başlangıç düğümünden diğer düğümlere olan en kısa yol mesafelerini hesaplar. Ancak, grafın her düğümünü bir kez daha gezerek, olası negatif ağırlıklı döngüleri kontrol eder.

printArr metodu, başlangıç düğümünden diğer düğümlere olan en kısa yol mesafelerini konsola yazdırır.

main metodu, bir graf oluşturur, kenarları tanımlar ve Bellman-Ford algoritmasını çağırarak başlangıç düğümünden diğer düğümlere olan en kısa yol mesafelerini hesaplar.

Shortest Path Algorithm (Dijkstra's Algorithm)

Dijkstra's Algorithm, tek bir kaynaktan tüm diğer düğümlere olan en kısa yol uzunluklarını hesaplamak için kullanılan bir tek kaynaklı en kısa yol algoritmasıdır. Bu algoritma, ağırlıklı graf (weighted graph) için tasarlanmıştır ve negatif ağırlıklı kenarları olan graf için uygun değildir. Dijkstra's Algorithm, greedy (açgözlü) yaklaşıma dayanır.

Dijkstra's Algorithm'ın Özellikleri:

- Ağırlıklı Graf: Dijkstra's Algorithm, ağırlıklı graf üzerinde tanımlıdır. Her bir kenarın (çizginin) bir ağırlığı (weight) vardır.
- Açgözlü Yaklaşım: Her adımda, şu ana kadar bilinen en kısa yolu kullanarak bir sonraki düğüme gider. Bu açgözlü yaklaşım, en kısa yolu bulmayı garanti eder.
- Tek Kaynaklı En Kısa Yol: Algoritma, tek bir başlangıç düğümünden tüm diğer düğümlere olan en kısa yol uzunluklarını hesaplar.

Algoritma Adımları:

- Başlangıçta, tüm düğümlerin en kısa yol uzunluklarını sonsuz (∞) olarak ayarlayın ve başlangıç düğümünün uzunluğunu 0 olarak ayarlayın.
- Tüm henüz işlenmemiş düğümler arasından en kısa yola sahip düğümü seçin.
- Seçilen düğümü işaretleyin ve bu düğümün komşularını güncelleyin: Eğer şu anki yoldan daha kısa bir yol bulunursa, yol uzunluğunu güncelleyin.
- Tüm düğümler işlenene kadar bu adımları tekrarlayın.

Shortest Path Algorithm Uygulaması

```
class ShortestPath {
    static final int V = 9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }
    void printSolution(int dist[])
    {
        System.out.println(
                "Vertex \t\t Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i + " \t\t " + dist[i]);
    }
```

```java
void dijkstra(int graph[][], int src)
{
    int dist[] = new int[V]; // The output array.
    Boolean sptSet[] = new Boolean[V];
    for (int i = 0; i < V; i++) {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] != 0
                    && dist[u] != Integer.MAX_VALUE
                    && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}
```

```
Vertex          Distance from Source
0               0
1               4
2               12
3               19
4               21
5               11
6               9
7               8
8               14
```

ShortestPath sınıfı, Dijkstra algoritmasını içerir. Bu sınıf, algoritmanın ana mantığını ve sonucunun çıktısını sağlar.
minDistance metodu, henüz işlenmemiş düğümler arasından minimum mesafeye sahip düğümü bulur.
printSolution metodu, başlangıç düğümünden diğer düğümlere olan en kısa yol mesafelerini konsola yazdırır.
dijkstra metodu, Dijkstra algoritmasının ana mantığını uygular. Bu metod, grafı ve başlangıç düğümünü alır, en kısa yol mesafelerini hesaplar ve sonucu konsola yazdırır.
main metodu, bir grafı temsil eden bir matris oluşturur ve Dijkstra algoritmasını çağırarak başlangıç düğümünden diğer düğümlere olan en kısa yol mesafelerini hesaplar.

A* Search Algorithm

A* (A yıldızı) Arama Algoritması, graf tabanlı problemlerde en kısa yol veya en uygun yolun bulunmasında kullanılan bir arama algoritmasıdır. Dijkstra'nın algoritmasına benzerlik gösterse de, A* algoritması hem gerçek maliyetleri hem de tahmini maliyetleri (heuristik değerleri) kullanarak en uygun yolu bulmaya çalışır.

Algoritma, genellikle bir başlangıç düğümü ve bir hedef düğümü arasındaki en uygun yolu bulmak için kullanılır. Her düğüm için iki değer hesaplanır:

- **g(n)**: Başlangıç düğümünden düğüme olan gerçek maliyet.
- **h(n)**: Düğümden hedef düğüme olan tahmini maliyet (heuristik değer).

A* algoritması, f(n) = g(n) + h(n) değerini minimizasyon amacıyla kullanır. Yani, f(n) değeri en küçük olan düğüm seçilir ve bu düğüm ziyaret edilir.

Algoritmanın adımları şunlardır:

Başlangıç düğümünü açık listeye ekleyin.

Açık liste boş olana kadar şu adımları tekrarlayın:

- Açık listeye ekli düğümler arasından f(n) değeri en küçük olan düğümü seçin.
- Seçilen düğümü kapalı listeye taşıyın.
- Seçilen düğümün komşularını kontrol edin:
    - Komşu düğüm kapalı listede ise veya geçerli değilse atlayın.
    - Eğer komşu düğüm açık listede değilse, onu açık liste ekleyin ve f(n), g(n), h(n) değerlerini hesaplayın.
    - Eğer komşu düğüm açık listede ise, daha iyi bir yolu bulduysanız g(n) ve h(n) değerlerini güncelleyin.
- Hedef düğüme ulaştığınızda, en uygun yolu ve bu yolu oluşturan düğümleri elde edin.

A* algoritması, doğru bir heuristik fonksiyonu kullanıldığında, Dijkstra'nın algoritmasından daha verimli sonuçlar verebilir. Ancak, heuristik fonksiyonun doğru ve tutarlı olması önemlidir.

A* Search Algoritması Uygulaması

```
class Cell {
    int parent_i, parent_j;
    double f, g, h;

    Cell() {
        this.parent_i = 0;
        this.parent_j = 0;
        this.f = 0;
        this.g = 0;
        this.h = 0;
    }
}
```

```
public class AStarSearch {

    private static final int ROW = 9;
    private static final int COL = 10;

    public static void main(String[] args) {
        int[][] grid = {
                {1, 0, 1, 1, 1, 1, 0, 1, 1, 1},
                {1, 1, 1, 0, 1, 1, 1, 0, 1, 1},
                {1, 1, 1, 0, 1, 1, 0, 1, 0, 1},
                {0, 0, 1, 0, 1, 0, 0, 0, 0, 1},
                {1, 1, 1, 0, 1, 1, 1, 0, 1, 0},
                {1, 0, 1, 1, 1, 1, 0, 1, 0, 0},
                {1, 0, 0, 0, 0, 1, 0, 0, 0, 1},
                {1, 0, 1, 1, 1, 1, 0, 1, 1, 1},
                {1, 1, 1, 0, 0, 0, 1, 0, 0, 1}
        };

        int[] src = {8, 0};
        int[] dest = {0, 0};

        aStarSearch(grid, src, dest);
    }
```

```java
private static boolean isValid(int row, int col) { return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL); }

private static boolean isUnBlocked(int[][] grid, int row, int col) { return grid[row][col] == 1; }

private static boolean isDestination(int row, int col, int[] dest) { return row == dest[0] && col == dest[1]; }

private static double calculateHValue(int row, int col, int[] dest) {
    return Math.sqrt((row - dest[0]) * (row - dest[0]) + (col - dest[1]) * (col - dest[1]));
}

private static void tracePath(Cell[][] cellDetails, int[] dest) {...}

private static void aStarSearch(int[][] grid, int[] src, int[] dest) {...}
```

```
The destination cell is found
The Path is
-> (8, 0)-> (7, 0)-> (6, 0)-> (5, 0)-> (4, 1)-> (3, 2)-> (2, 1)-> (1, 0)-> (0, 0)
```

isValid: Verilen satır ve sütun değerlerinin ızgara içinde geçerli olup olmadığını kontrol eder.

isUnBlocked: Belirtilen konumun ızgarada engel oluşturmayan bir hücre olup olmadığını kontrol eder.

isDestination: Belirtilen konumun hedef konum olup olmadığını kontrol eder.

calculateHValue: Manhatta mesafesi gibi bir heuristik değer hesaplar. Bu değer, belirli bir konumdan hedef konuma olan tahmini mesafeyi ifade eder.

tracePath: Başlangıçtan hedefe olan yolun izini takip eder ve bu yolu konsola yazdırır.

aStarSearch: A* arama algoritmasını uygular. Başlangıçtan hedefe doğru en kısa yolu bulur ve bu yolu konsola yazdırır.

Floyd-Warshall Algorithm

Floyd-Warshall algoritması, graf tabanlı bir problemin tüm çiftler arası en kısa yolunu bulmak için kullanılan bir dinamik programlama algoritmasıdır. Bu algoritma, negatif ağırlıklara sahip kenarlar içeren grafiklerde de çalışabilir, bu yönüyle Dijkstra ve A* gibi algoritmalardan farklıdır.

Algoritma, tüm düğümler arasındaki en kısa yolların maliyetini hesaplamak için bir matris kullanır. Algoritmanın ana fikri, bir düğümü diğer bir düğme ulaştıran tüm yolları kontrol ederek en kısa yolu bulmaktır.

Floyd-Warshall algoritmasının ana adımları şunlardır:

- İki boyutlu bir dizi (matris) oluşturun ve bu matrisi başlangıç grafik matrisiyle doldurun. Bu matris, iki düğüm arasındaki doğrudan kenarların ağırlıklarını içerir.
- Matris üzerinde üçlü bir döngü kullanarak tüm düğüm çiftleri için en kısa yolları güncelleyin. İlk iki döngü, bir başlangıç ve bir hedef düğümü seçerken, üçüncü döngü ise bu yolları kontrol eder.
- Her iterasyonda, belirli bir düğüm çifti arasında doğrudan bir yol ve aracı düğümler üzerinden geçen bir yol arasından daha kısa olan yolu bulmaya çalışın. Eğer böyle bir yol bulunursa, matrisi güncelleyin.

Algoritma tamamlandığında, matrisin her elemanı, ilgili iki düğüm arasındaki en kısa yolu içerir. Eğer iki düğüm arasında doğrudan bir yol yoksa, ilgili matris elemanı sonsuz değer olacaktır.

Floyd-Warshall algoritması, tüm düğüm çiftleri için en kısa yolları bulmak için idealdir ve özellikle negatif ağırlıklı kenarlara sahip grafiklerde kullanışlıdır. Ancak, büyük grafikler için bellek ve hesaplama maliyeti yüksek olabilir.

Floyd-Warshall Algoritması Uygulaması

```java
import java.lang.*;
class AllPairShortestPath {
    final static int INF = 99999, V = 4;

    void floydWarshall(int dist[][])
    {
        int i, j, k;
        for (k = 0; k < V; k++) {
            for (i = 0; i < V; i++) {

                for (j = 0; j < V; j++) {

                    if (dist[i][k] + dist[k][j]
                            < dist[i][j])
                        dist[i][j]
                                = dist[i][k] + dist[k][j];
                }
            }
        }
        printSolution(dist);
    }
```

```
void printSolution(int dist[][])
{
    System.out.println(
            "The following matrix shows the shortest "
                    + "distances between every pair of vertices");
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (dist[i][j] == INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j] + "   ");
        }
        System.out.println();
    }
}
public static void main(String[] args)
{
    int graph[][] = { { 0, 5, INF, 10 },
            { INF, 0, 3, INF },
            { INF, INF, 0, 1 },
            { INF, INF, INF, 0 } };
    AllPairShortestPath a = new AllPairShortestPath();

    a.floydWarshall(graph);
}
```

```
The following matrix shows the shortest distances between every pair of vertices
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0
```

floydWarshall: Floyd-Warshall algoritmasını uygular. İki düğüm arasındaki en kısa yol mesafesini bulmak için tüm düğümleri dolaşır ve dinamik programlama yaklaşımını kullanır.

printSolution: Sonuç matrisini konsola yazdırır. Matris, her iki düğüm arasındaki en kısa yol mesafelerini içerir. Mesafe INF (sonsuz) ise, iki düğüm arasında bir yol olmadığını gösterir.

Ford-Fulkerson Algorithm for Maximum Flow

Ford-Fulkerson algoritması, bir grafın en büyük akışını (maximum flow) bulmak için kullanılan bir algoritmadır. Bu algoritma, bir grafın kapasiteli kenarları arasında en büyük akışı bulmayı amaçlar ve özellikle akış ağları problemlerinde kullanılır.

Algoritmanın ana fikri, bir başlangıç düğümünden bir hedef düğüme kadar olan yollarda maksimum akışı bulmaktır. Akış ağı, düğümler (kaynak, hedef ve ara düğümler) ve bu düğümleri birbirine bağlayan kapasiteli kenarlar (kanallar) içerir.

Ford-Fulkerson algoritması, genellikle artırıcı yolu bulma (augmenting path) yöntemiyle çalışır. Bu yöntemde:

- Başlangıçta, tüm kenarlarda akış değeri sıfırdır.
- Artırıcı bir yol bulmak için genişlik öncelikli arama (BFS) kullanılır.
- Artırıcı yol bulunduğunda, bu yol üzerindeki kenarların kapasitesi ile mevcut akış değerine göre yeni bir akış değeri belirlenir.
- Bu işlem, artırıcı yol bulunamayana kadar tekrarlanır.

Algoritma tamamlandığında, maksimum akış değeri ve her kenarın üzerinden geçen akış miktarı elde edilir.

Ford-Fulkerson algoritması, teorik olarak etkilidir, ancak uygulamada bazı durumlarda yavaş çalışabilir. Bu nedenle, daha etkili ve hızlı çalışan Edmonds-Karp algoritması gibi varyasyonları da mevcuttur.Bu tür algoritmalar, ağ tasarımı, ulaşım planlaması, telekomünikasyon ve enerji dağıtımı gibi birçok alanda kullanılır.

Ford Furkerson Algoritması Uygulaması

```
class MaxFlow {
    static final int V = 6;
    boolean bfs(int rGraph[][], int s, int t, int parent[])
    {
        boolean visited[] = new boolean[V];
        for (int i = 0; i < V; ++i)
            visited[i] = false;
        LinkedList<Integer> queue
                = new LinkedList<Integer>();
        queue.add(s);
        visited[s] = true;
        parent[s] = -1;
        while (queue.size() != 0) {
            int u = queue.poll();

            for (int v = 0; v < V; v++) {
                if (visited[v] == false
                        && rGraph[u][v] > 0) {
                    if (v == t) {
                        parent[v] = u;
                        return true;
                    }
                    queue.add(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }
        return false;
    }
```

```
    int fordFulkerson(int graph[][], int s, int t)
    {
        int u, v;
        int rGraph[][] = new int[V][V];
        for (u = 0; u < V; u++)
            for (v = 0; v < V; v++)
                rGraph[u][v] = graph[u][v];
        int parent[] = new int[V];
        int max_flow = 0;
        while (bfs(rGraph, s, t, parent)) {
            int path_flow = Integer.MAX_VALUE;
            for (v = t; v != s; v = parent[v]) {
                u = parent[v];
                path_flow
                        = Math.min(path_flow, rGraph[u][v]);
            }
            for (v = t; v != s; v = parent[v]) {
                u = parent[v];
                rGraph[u][v] -= path_flow;
                rGraph[v][u] += path_flow;
            }
            max_flow += path_flow;
        }
        return max_flow;
    }
```

```
    public static void main(String[] args)
            throws java.lang.Exception
    {
        int graph[][] = new int[][] {
                { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },
                { 0, 4, 0, 0, 14, 0 }, { 0, 0, 9, 0, 0, 20 },
                { 0, 0, 0, 7, 0, 4 }, { 0, 0, 0, 0, 0, 0 }
        };
        MaxFlow m = new MaxFlow();
        System.out.println("The maximum possible flow is "
                + m.fordFulkerson(graph,  s: 0,  t: 5));
    }
}
```

Result : The maximum possible flow is 23

bfs: Genişlik öncelikli arama (Breadth-First Search - BFS) algoritmasını kullanarak artan yolu bulur. Bu yöntem, artan yolu bulmak için kullanılır. Başlangıç düğümünden bitiş düğümüne kadar bir yol bulunabilirse true döner.

fordFulkerson: Ford-Fulkerson algoritmasını uygular. Bu algoritma, artan yolu bulur ve akışı artırarak maksimum akışı bulur.

main: Programın ana bölümüdür. Başlangıç ve bitiş düğümleri arasındaki maksimum akışı hesaplar ve konsola yazdırır.

Greedy Graph Colouring Algoritması

Graf boyama problemi, bir grafın düğümlerini, komşu düğümler arasında aynı rengi paylaşmayacak şekilde minimum sayıda renkle boyamayı amaçlar. Greedy algoritması, bu problemi çözmek için basit ve sezgisel bir yaklaşım sunar.

Greedy algoritması, her adımda mevcut düğüme bir renk atar ve bu renk ile komşu düğümlerin aynı rengi paylaşıp paylaşmadığını kontrol eder. Eğer komşu düğümler aynı rengi paylaşıyorsa, mevcut düğüme farklı bir renk atanır.

- Graf boyama için Greedy algoritmasının basit bir uygulaması şu adımları takip eder:
- Düğümleri başlangıçta renksiz olarak işaretle.
- Düğümleri bir sırayla ziyaret edin.
- Her düğüm için, mevcut renk kümesinde mevcut komşu düğümlerin renklerini kontrol edin.
- Eğer komşu düğümlerden hiçbiri mevcut düğümün rengini kullanmıyorsa, mevcut düğüme en küçük kullanılmayan rengi atayın.

- Eğer tüm renkler kullanılıyorsa, yeni bir renk oluşturun ve mevcut düğüme bu yeni rengi atayın.

Bu yaklaşım, genellikle iyi sonuçlar verir ve hızlı bir şekilde çalışır. Ancak, Greedy algoritması her zaman en az sayıda rengi kullanarak bir grafı boyayamayabilir. Özellikle karmaşık yapıya sahip grafiklerde veya büyük grafiklerde, Greedy algoritması optimal bir çözüm üretmeyebilir.

Bu nedenle, bazen daha karmaşık ve optimizasyon odaklı algoritmalara ihtiyaç duyulabilir. Ancak, basitlik ve hız avantajları nedeniyle Greedy algoritması graf boyama problemi için popüler bir yaklaşımdır.

Greedy Graph Colouring Algoritması Uygulaması

```
class GreedyGraphColouring
{
    private int V;
    private LinkedList<Integer> adj[];
    GreedyGraphColouring(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        adj[w].add(v);
    }
    void greedyColoring()
    {
        int result[] = new int[V];

        Arrays.fill(result, val: -1);

        result[0] = 0;
        boolean available[] = new boolean[V];

        Arrays.fill(available, val: true);

        for (int u = 1; u < V; u++)
        {
            Iterator<Integer> it = adj[u].iterator() ;
```

```
        while (it.hasNext())
        {
            int i = it.next();
            if (result[i] != -1)
                available[result[i]] = false;
        }
        int cr;
        for (cr = 0; cr < V; cr++){
            if (available[cr])
                break;
        }
        result[u] = cr;
        Arrays.fill(available,  val: true);
    }
    for (int u = 0; u < V; u++)
        System.out.println("Vertex " + u + " ---> Color "
                + result[u]);
}
```

```
Coloring of graph 1
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 1

Coloring of graph 2
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 3
```

GreedyGraphColouring: Açgözlü graf renklendirme işlemini gerçekleştiren sınıftır. Grafi temsil etmek için bir LinkedList dizisi kullanır.

addEdge: İki düğüm arasında bir kenar eklemek için kullanılır.

greedyColoring: Açgözlü renklendirme algoritmasını uygular. Her düğüm için komşu düğümlerin renklerini kontrol eder ve kullanılabilir bir renk bulmaya çalışır. Son olarak, her düğümün rengini belirler ve konsola yazdırır.

main: Programın ana bölümüdür. İki farklı graf oluşturur ve her biri için açgözlü renklendirme algoritmasını çağırır. Oluşturulan grafın renklendirilmiş sürümünü konsola yazdırır.

Kaynaklar

- GeeksForGeeks ~ Graph Data Structure And Algorithms
- Graph theory MIT ~ zerocoolhac
- Graphs MIT OpenCourseWare
- https://www.javatpoint.com/graph-algorithms
- https://www.youtube.com/watch?v=DgXR2OWQnLc

Ekler

Özgeçmiş

Ad,Soyad	İbrahim Bayburtlu
Doğum Tarihi	01.06.2001
Doğum Yeri	Amasya
Lise	Rami Atatürk Anadolu Lisesi
Çalışma Geçmişi	Softtech(04/2024 - Devam ediyor)
	IBTech(7 ay)
	Teknosa(9 ay)