ALDA VT12: Grafer

Leon Hennings leonh

Kamyar Sajjadi kamy-saj

16 februari 2012

Innehåll

Koden finns att ladda ner på `http://people.dsv.su.se/~kamy-saj/download/ALDA/`
Detta för att det ska bli lättare för er att göra en peer-review i er favorit editor.

Koden till zip-filen är: **rwa5uy3**

1 Motivering till implemenetation

Vi valde att göra vår graf som en adjacency list då vi i detta fallet inte skulle ha så många bågar mellan noderna i grafen. Vi drog denna slutsats från testkoden och även från att det generellt inte finns så många kopplingar mellan noder i en genomsnittlig graf. Då Kruskals algoritm är mer beroende av antalet bågar än hur stort trädet är så är den mer lämplig för uppgiften. Den är O(E log E) där E är antalet bågar. Det är viktigt att alltid ta hänsyn till hur grafen kommer att användas. Hade vi haft många bågar mellan noderna i grafen hade det varit mer lönsamt ddatt använda sig av prims algoritm. Eftersom vi skulle ha viktade bågar skapade vi en egen klass för dem. För att kruskals algoritm enkelt skulle kunna ta den lägst viktade bågen gjorde vi dessa comparable så de kunde ordnas i en prioritetskö. För sammanslagningen av träd i algoritmen valde vi att använda depth-first search för att kontrollera så två noder inte befinner sig i samma träd. Alternativet hade varit att göra de mindre träden till set och göra merge på dessa då noder i de olika träden ska sammankopplas. Detta kräver flera datastrukturer för de olika träden. Då vi inte skulle hantera särskilt stora grafer kändes depth-first search fullt tillräckligt. Se kommentarer i koden för mer ingående beskrivning av algoritmen.

2 MyGraph

2.1 Privata variabler samt Edge klassen

```
1  public class MyGraph<T extends Comparable<? super T>> implements
       MiniGraph<T>
2  {
3    private HashSet<T> nodes = new HashSet<T>();
4    private ArrayList<Edge<T>> edges= new ArrayList<Edge<T>>();
5
6    /**
7     * The Edge class.
8     * this class is pretty straightforward.
9     */
10   private static class Edge<T> implements Comparable<Edge<T>>
11   {
12     private T node1;
13     private T node2;
14     private int weight;
15
16     /**
17      * Constructor
18      * @param n1 One of the nodes
19      * @param n2 The other node ^^
20      * @param w weight of the edge
21      */
22     public Edge(T n1, T n2, int w){
23       node1 = n1;
24       node2 = n2;
25       weight = w;
26     }
27
28     /**
29      * GetMethod for weight
30      * @return int weight
31      */
```

```
32      public int getW(){
33        return weight;
34      }
35
36      /**
37       * GetMethod for node1
38       * @return T node1
39       */
40      public T getN1(){
41        return node1;
42      }
43
44      /**
45       * GetMethod for node2
46       * @return T node2
47       */
48      public T getN2(){
49        return node2;
50      }
51
52      /**
53       * A basic compareTo method
54       * this method is comparing the weight of the edges
55       * @return int
56       */
57      public int compareTo(Edge other){
58        if(other.weight < weight)
59          return 1;
60        else if (other.weight < weight)
61          return -1;
62        else
63          return 0;
64      }
65
66      /**
67       * toString
68       * @return String
69       */
70      public String toString()
71      {
72        return node1+" "+ node2+" Weight: "+weight;
73      }
74    }
```

2.2 addNode

```
1   /**
2    * Method for adding a node to the graph. Silently ignores any
       duplicates
3    * @param n The node to add to the graph.
4    */
5   public void addNode(T n)
6   {
7     nodes.add(n);
8   }
```

2.3 connectNodes

```
1   /**
2    * Method for creating an unidirectional edge between two nodes.
       Does
3    * nothing if the cost is negative, the edges are already
       connected, or if
4    * one or more of the nodes doesn't exists.
5    * @param n1 The first node to create an edge between
6    * @param n2 The second node to create an edge between
7    * @param weight The cost for traversing the edge
8    */
9   public void connectNodes(T n1, T n2, int weight)
10  {
11    if(!(weight<0))
12      if(nodes.contains(n1) && nodes.contains(n2))
13        if(!edgeExistsBetween(n1, n2))
14        {
15          Edge<T> e = new Edge<T>(n1, n2, weight);
16          edges.add(e);
17        }
18  }
```

2.4 contains

```
1
2   /**
3    * Method for searching the graph for a certain node. If the node
       is present
4    * in the graph, the method returns true, otherwise, it returns
       false.
5    *
6    * @return boolean true if the graph contains n, otherwise false.
7    */
8   public boolean contains(T n){
9     return nodes.contains(n);
10  }
```

2.5 getNumberOfNodes

```
1   /**
2    * Method for finding the number of nodes in the graph.
3    *
4    * @returns int The number of nodes in the graph.
5    */
6   public int getNumberOfNodes(){
7     return nodes.size();
8   }
```

2.6 edgeExistsBetween

```
1   /**
2    * Checks if there exists and edge between nodes n1 and n2. Used
       for testing
3    * purposes.
4    *
5    * @param n1 The first node that identifies the edge.
6    * @param n2 The second node that identifies the edge.
7    * @return true if and edge exists between n1 and n2, otherwise
       false.
8    */
9   public boolean edgeExistsBetween(T n1, T n2){
10    for(Edge e : edges){
11      if(e.node1 == n1 && e.node2 == n2
12          || e.node1 == n2 && e.node2 == n1)
13        return true;
14    }
15    return false;
16  }
```

2.7 getNumberOfEdges

```
1   /**
2    * Gets the number of edges in the graph. Used for testing
       purposes.
3    *
4    * @return the number of edges in the graph.
5    */
6   public int getNumberOfEdges(){
7     return edges.size();
8   }
```

2.8 getTotalEdgeWeight

```
1   /**
2    * Gets the total weight of all edges. Used for testing purposes.
3    *
4    * @return the total weight of all the edges
5    */
6   public int getTotalEdgeWeight(){
7     int total = 0;
8     for(Edge e : edges){
9       total += e.weight;
10    }
11    return total;
12  }
```

2.9 generateMinimumSpanningTree

```
1
2  /**
3    * Method for calculating a minimum spanning tree for the graph.
4    * This method is generating the minimum spanning tree. We are
       using kruskals
5    * algoritm.
6    *
7    * @return tree Graph A new instance of the Graph class,
       representing a minimal
8    * spanning tree.
9    */
10  public MyGraph<T> generateMinimumSpanningTree(){
11    // grafen som ska returneras
12    MyGraph<T> tree = new MyGraph<T>();
13    // bågarna sorterade efter minst först
14    PriorityQueue<Edge> deck = new PriorityQueue<Edge>(edges);
15
16    while((getNumberOfNodes()-1) > tree.getNumberOfEdges())
17    {
18      Edge<T> tmp = deck.poll();
19
20      // Om ingen av noderna finns med i tree så lägger vi till
          noderna i tree och
21      // kopplar dessa. Notera att
          addAndConnectToMinimumSpanningTree har en
22      // kontroll så att det i inte redan finns en kopling mellan
          noderna. I detta
23      // fall behövs det då en nod som är kopplad till sig själv ej
           ska läggas till.
24      if( (!tree.nodes.contains(tmp.getN1()) || !tree.nodes.
          contains(tmp.getN2()) ))
25      {
26        addAndConnectToMinimumSpanningTree(tmp, tree);
27      }
28
29      // Annars om noderna redan finns med i vårat träd så ska vi
          eventuellt slå
```

```
30        // ihop träden till ett större träd. Även här är det viktigt
              med kontrollen
31        // som ligger i addAndConnectToMinimumSpanningTree detta för
              att inte bågen
32        // ska läggas till om de ligger i samma träd. Om kontrollen
              ej hade varit med
33        // skulle det kunna uppstå cykler.
34        else if(tree.nodes.contains(tmp.getN1()) && tree.nodes.
              contains(tmp.getN2()))
35        {
36          addAndConnectToMinimumSpanningTree(tmp,tree);
37        }
38      }
39      return tree;
40    }
```

2.10 addAndConnectToMinimumSpanningTree

```
1     /**
2      * Internal method for adding nodes and connect them.
3      * Sorry for the long method name.
4      * @param tmp the edge with all info that we need to use.
5      * @param tree the forest we are building.
6      */
7     private void addAndConnectToMinimumSpanningTree(Edge<T> tmp,
          MyGraph<T> tree)
8     {
9       if(!tree.depthFirstSearch(tmp.getN1(), tmp.getN2()))
10      {
11        tree.addNode(tmp.getN1());
12        tree.addNode(tmp.getN2());
13        tree.connectNodes(tmp.getN1(), tmp.getN2(), tmp.getW());
14      }
15    }
```

2.11 depthFirstSearch boolean

```
1     /**
2      * Depth-first search method.
3      * This method will return true if it finds a
4      * path between from-node and to-node
5      * @param from from-node
6      * @param to to-node
7      */
8     private boolean depthFirstSearch(T from, T to)
9     {
10      Set<T> visited = new HashSet<T>();
11      depthFirstSearch(from, visited);
12      return visited.contains(to);
13    }
```

2.12 depthFirstSearch void

```
1   /**
2    * Internal method for depthFirstSearch(T, T).
3    * Building the HashSet with rekursion.
4    * @param T from-node
5    * @param visited the hashset
6    */
7   private void depthFirstSearch(T from, Set<T> visited)
8   {
9     visited.add(from);
10
11    for(Edge<T> e : edges)
12    {
13      T tmp = null;
14
15      // Kollar om någon av noderna är den noden som vi ska utgå
            från.
16      // sedan sätter den destinationsnoden till tmp och gör ett
            rekursivt
17      // anrop.
18      if(e.getN1().equals(from) || e.getN2().equals(from))
19        tmp = e.getN1().equals(from) ? e.getN2(): e.getN1();
20
21      if(!visited.contains(tmp))
22        depthFirstSearch(tmp, visited);
23    }
24  }
```

3 Grafalgoritmer

3.1 Multigrafer

3.1.1 Prims algoritm

Algoritmen fungerar för multigrafer utan modifikationer så länge de är oriktade. För riktade grafer är det irrelevant då ett riktat träd inte kan vara spanning. Anledningen till att det fungerar även för multigrafer är att bågarnas vikt alltid jämförs och det minsta kommer väljas oavsett om det finns flera bågar mellan ett par noder.

3.1.2 Kruskals algoritm

Då Kruskals algoritm endast tar de kortaste bågarna för alla noder som inte redan är ihopkopplade kommer det inte spela någon roll ifall grafen är en multigraf.

3.1.3 Dijkstras algoritm

Denna algoritm kommer att fungera på multigrafer. Dijkstras kommer alltid att hitta den kortaste vägen från X till Y även om det finns fler bågar ut från en nod till en annan. Det man eventuellt ska ta hänsyn till när dijkstras används på multigrafer är att hålla reda på vilken av bågarna som var den kortaste vägen. Så om det finns två bågar ut från A till B så måste vi ta hänsyn till vilken av dessa två bågar som har den lägsta vikten och spara denna i tabellen. Sedan bör de bågar returneras som leder till slutnoden istället som i vanliga fall då noderna som bildar den kortaste vägen returneras.

3.1.4 Djupet först sökning

Djupet först-sökning fungerar för multigrafer. Den kommer se om det finns en koppling mellan X och Y. Man kan tro att samma nod kommer besökas fler gånger om det finns fler bågar ut till samma nod, men eftersom algoritmen markerar noderna när den har besökt dem så kommer vi inte behöva gå till samma nod fler gånger.

3.1.5 Bredden först sökning

På samma sätt som djupet först sökning kommer bredden först sökning fungera för multigrafer. Eftersom vilka noder som besökts och deras lägsta vikt sparas undan kommer de fallen där noder har flera bågar inte spela någon roll mer än ur effektivitets perspektiv.

3.1.6 Topologisk sortering

Kommer ej att fungera på grafen om den inte är riktad. Om multigrafen är riktad så kommer denna algoritm att fungera. Algoritmen kollar på den nod som är minimal, dvs den nod som ej har några bågar inåt. Algoritmen tar ej hänsyn till hur många bågar som går utåt från den minimala noden. Därav kommer topologisk sortering att fungera på multigrafer.

3.2 Negativ vikt

3.2.1 Prims algoritm

För grafer med negativa vikter fungerar denna algoritm. Det påverkar inte algoritmen att bågarna har negativa vikter. Algoritmen kommer ta den minsta bågen till nästa nod, men endast om den ej har besökt den noden. Detta gör att det ej spelar någon roll om det är ett negativt tal eller positivt, så länge det är det minsta talet av alla de som finns att välja på. Tabellen som byggs upp i algoritmen består av de lägsta noderna som sammansätter grafen till ett träd, det vill säga att det inte påverkar om det är negativa tal.

3.2.2 Kruskals algoritm

Även denna algoritm fungerar med negativt viktade grafer. Kruskals algoritm bygger trädet med hjälp av de bågar som har lägst värde. Detta innebär att det inte spelar någon roll om bågarna har negativa värden, så länge det är de bågar med lägst vikt som används för att bygga trädet.

3.2.3 Dijkstras algoritm

Dijkstras algoritm kommer ej att fungera på denna sortens grafer. Algoritmen går ut på att man räknar kostnaden hittills från nod A till B. Det innebär att man, ifall grafen är riktad, kan gå tillbaka till en redan besökt nod och på så vis använda den negativa vikten för att minska totalvikten ett oändligt antal gånger för att försöka få fram den kortaste vägen. En lösning för detta problem skulle vara att alltid se till så att samtliga bågar i grafen blir positiva. Detta genom att addera lägsta vikten med ett tills den blir positiv och sedan addera lika mycket till de andra vikterna, så att alla blir positiva men differensen är bevarad sinsemellan. Det viktiga här är att bågarnas vikt i förhållande till varandra ej ändras.

3.2.4 Djupet först sökning

Djupet först-sökning tar inte hänsyn till bågarnas vikt i sin sökning. Negativ vikt har därför ingen verkan på dess funktionalitet. Den kommer endast att hitta en väg mellan noderna, om noderna är negativa eller inte har ingen betydelse.

3.2.5 Bredden först sökning

Den följer alla bågar tills rätt nod hittas utan hänsyn till bågarnas vikt och fungerar därför för negativa vikter av samma anledning som djupet-först-sökning.

3.2.6 Topologisk sortering

Topologisk sortering kommer att fungera med negativt viktade grafer. Sorteringen tar ej hänsyn till vikterna på bågarna. Den kommer endast att leta efter den lägsta noden. Lägsta i topologisk sortering innebär den noden som ej har några bågar in mot sig.

4 Poolfrågor

2. Frågan måste behandla hur grafer kan implementeras.

4.1 Fråga 1

Förklara skillnaden på adjacency list och adjacency matrix, ge exempel på när de är att föredra. För högre betyg ska du även förklara vad som krävs för att båda varianterna ska kunna hantera viktade kanter.

4.2 Fråga 2

Givet en fullständig graf (alla noder har bågar till alla andra noder) vilket sätt är mest lönsamt att implementera denna på och vilken algoritm är bäst lämpad för att bestämma det minimalt spännande trädet i grafen? För högre betyg ska du motivera ditt val av implemenetation och algoritm på ett sätt som visar att du förstått dem båda.