Föreläsning 14

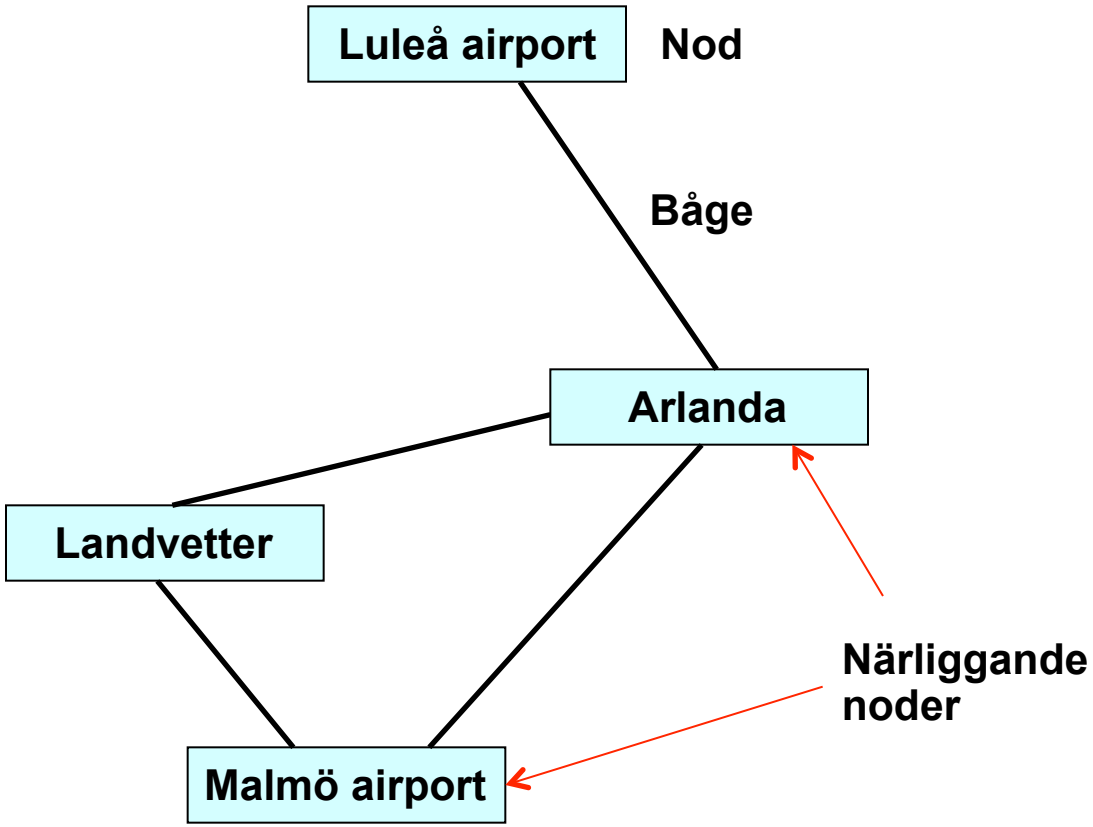
- Graf

JF 19

Graf

I en graf kan en **nod** (**vertex**) referera till ett godtyckligt antal andra noder. En länk till en annan nod kallas för en **båge** (**edge**).
Två noder som sammanbinds med en båge är **närliggande** (**adjacent**).

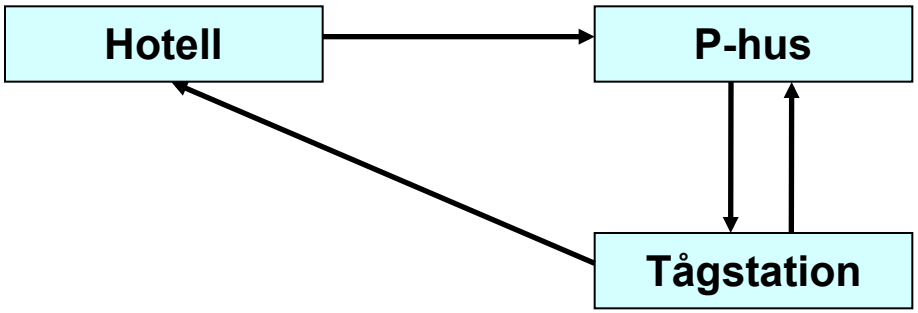
Ett exempel på en graf är flygplatser sammanbundna med flyglinjer.



Graf

I en **riktad graf** (**directed graph**) har varje båge en riktning. Bågen startar i en nod och slutar i en nod.

Exempel: Om man ska ta sig från P-hus till Hotell med bil måste man färdas via Tågstation.

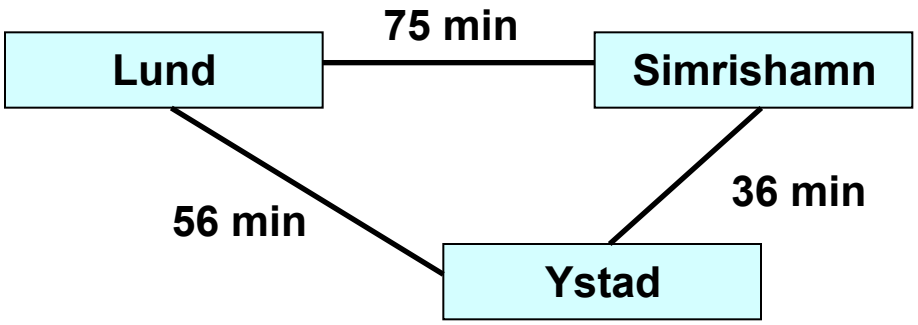


En graf där bågarna ej har riktning är en **oriktad graf** (**undirected graph**).

Graf

I en **viktad graf (weighted graph)** har varje båge en vikt. Vikten kan t.ex. beskriva avståndet mellan noderna (i tid eller sträcka eller …)

Exempel: Om man kör bil från Lund till Simrishamn tar det 75 minuter. Men om man tar vägen via Ystad tar det 92 minuter.



En graf där bågarna ej har någon vikt är en **oviktad graf**.

Graf - implementering

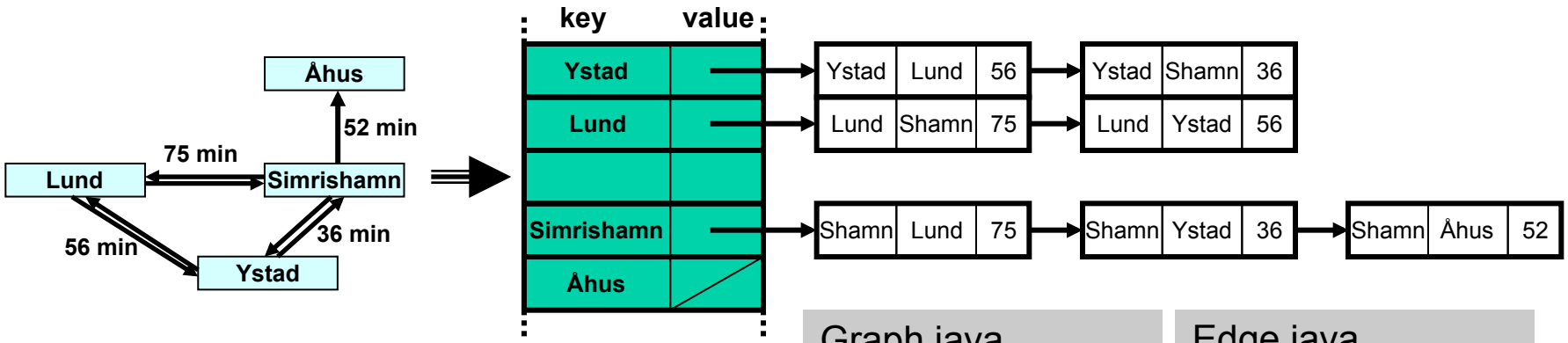
En graf består av en mängd av noder och en mängd av bågar.

Vi kommer implementera en graf som är ***riktad*** och ***viktad***:

Klassen **Edge(from, to, weight)** representerar en båge i grafen.

Klassen **Graph(graph)** representerar en graf med noder och bågar. Noder och bågar lagras i en ***HashMap*** där varje nyckel är en nod och det tillhörande värdet är en lista med bågar som startar i noden:

```
public class Graph<T> {
   private HashMap< T, ArrayList<Edge<T>> > graph = new …
   :
}
```



Graph.java | Edge.java

Graf - implementering

Operationer i klassen **Graph<T>**:

- **addVertex(T vertex) : void**
- **addVertex(T vertex, ArrayList< Edge<T>> adjacentList) : void**
- **addEdge(T from, T to, int weight) : boolean**
- **removeVertex(T vertex) : ArrayList<Edge<T>>**
- **removeEdge(T from, T to) : boolean**
- **getAdjacentList(T vertex) : ArrayList<Edge<T>>**
- **containsVertex(T vertex) : boolean**
- **getEdge(T from, T to) : Edge**

Graf – traversera på djupet

Strategi vid traversering på djupet i en graf är ungefär samma som man använder för att söka genom en labyrint. Testa olika vägar tills hela labyrinten är besökt.

1 Flytta från nod till nod utan att besöka en nod två gånger.

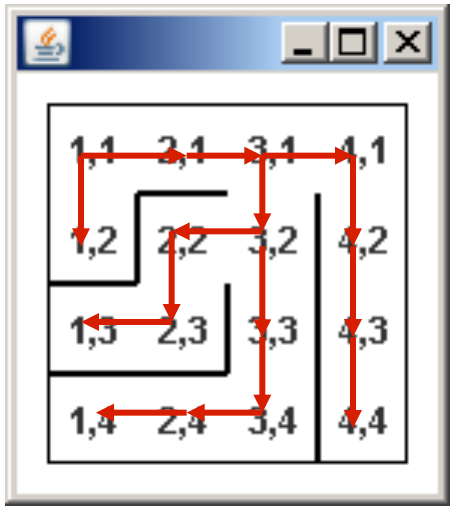
2 Om man besökt alla (nåbara) noder så är traverseringen klar.

3 Om man inte kan flytta vidare till en icke besökt nod så gå tillbaka, nod för nod, tills det finns ett alternativt vägval som leder till en icke besökt nod. Och fortsätt sedan enligt punkt 1.

Exempel:

Traversering med start i 1,1

```
1,1-> 1,2
1,1-> 2,1-> 3,1-> 3,2-> 2,2-> 2,3-> 1,3
                    3,2-> 3,3-> 3,4-> 2,4-> 1,4
              3,1-> 4,1-> 4,2-> 4,3-> 4,4
```



Graf – traversera på djupet

```
public static <T> Iterator<T> traversalDF(Graph<T> graph, T start) {
   LinkedList<Edge<T>> stack = new LinkedList<Edge<T>>();
   LinkedHashSet<T> visited = new LinkedHashSet<T>();
   Edge<T> edge;

   visited.add(start);
   stack.addAll(graph.getAdjacentList(start));
   while( !stack.isEmpty() ) {
      edge = stack.removeLast();
      if(!visited.contains(edge.getTo())) {
         visited.add(edge.getTo());
         stack.addAll(graph.getAdjacentList(edge.getTo()));
      }
   }
   return visited.iterator();
}
```

GraphExample.java GraphSearch.java

Graf – sökning på djupet

Strategi vid sökning på djupet i en graf är ungefär samma som man använder för att hitta i en labyrint. Testa olika vägar tills man kommit till målet.

1 Flytta från nod till nod utan att besöka en nod två gånger.

2 Om man kommer till noden man söker så är sökningen klar.

3 Om man inte kan flytta vidare till en icke besökt nod så gå tillbaka nod för nod tills det finns ett alternativt vägval som leder till en icke besökt nod. Och fortsätt sedan enligt punkt 1.

Exempel:

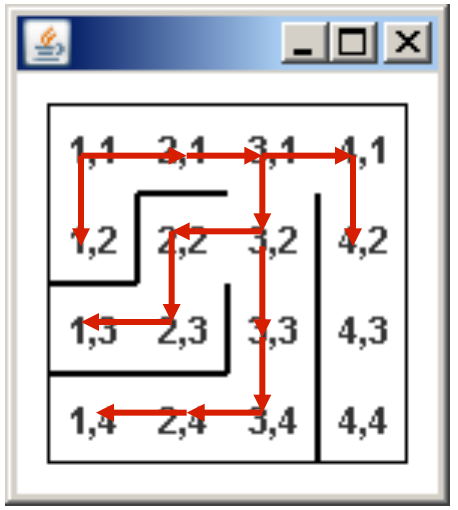
Sökning från 1,1 till 4,2 kan t.ex. bli så här

```
1,1-> 1,2
1,1-> 2,1-> 3,1-> 3,2-> 2,2-> 2,3-> 1,3
                  3,2-> 3,3-> 3,4-> 2,4-> 1,4
            3,1-> 4,1-> 4,2
```

Sökning från 1,1 till 4,2 kan t.ex. bli så här

```
1,1-> 2,1-> 3,1-> 4,1-> 4,2
```

Sökresultatet beror på i vilken ordning bågarna lagras



Graf – sökning på djupet

```
public static <T> ArrayList<Edge<T>> depthFirstSearch(Graph<T> graph, T from, T to)
{
   LinkedList<Edge<T>> stack = new LinkedList<Edge<T>>();
   HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
   boolean arrived = from.equals(to);
   Edge<T> edge;

   visited.put(from,null);
   stack.addAll(graph.getAdjacentList(from));
   while( !stack.isEmpty() && !arrived ) {
      edge = stack.removeLast();
      if(!visited.containsKey(edge.getTo())) {
         visited.put(edge.getTo(),edge);
         stack.addAll(graph.getAdjacentList(edge.getTo()));
         arrived = to.equals(edge.getTo());
      }
   }
   return getPath(from, to, visited);
}
```

GraphExample.java GraphSearch.java

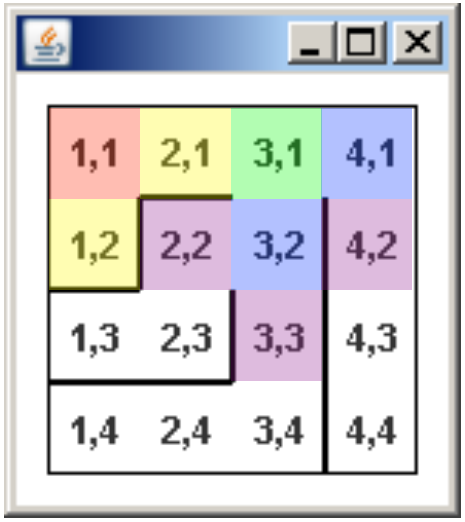
Graf – traversera på bredden

Strategi vid traversering på bredden är att besöka alla närliggande noder. Och därefter besöka deras närliggande noder osv. På det viset breder sökningen ut sig från startnoden.

Exempel:

Traversera med start i 1,1:

1,1-> 1,2 och 2,1	Startnod
1,2 stopp	1 nod bort
2,1-> 3,1	
3,1-> 3,2 och 4,1	2 noder bort
3,2-> 2,2 och 3,3	3 noder bort
4,1-> 4,2	
2,2-> 2,3	4 noder bort
3,3-> 3,4	
4,2-> 4,3 osv	



Graf – traversera på bredden

```
public static <T> Iterator<T> traversalBF(Graph<T> graph, T start) {
   LinkedList<Edge<T>> queue = new LinkedList<Edge<T>>();
   LinkedHashSet<T> visited = new LinkedHashSet<T>();
   Edge<T> edge;

   visited.add(start);
   queue.addAll(graph.getAdjacentList(start));
   while( !queue.isEmpty() ) {
      edge = queue.removeFirst();
      if(!visited.contains(edge.getTo())) {
         visited.add(edge.getTo());
         queue.addAll(graph.getAdjacentList(edge.getTo()));
      }
   }
   return visited.iterator();
}
```

GraphExample.java

GraphSearch.java

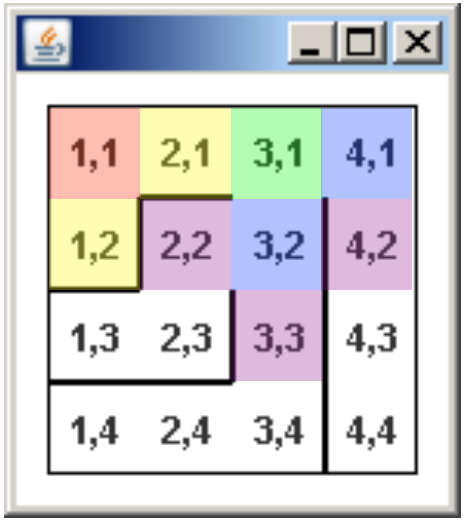
Graf – sökning på bredden

Strategi vid sökning på bredden är att utforska alla närliggande noder. Och därefter utforska deras närligande noder osv. På det viset breder sökningen ut sig från startnoden.

Exempel:

Sökning från 1,1 till 4,2, Utforskade noder:

1,1-> 1,2 och 2,1	Startnod
1,2 stopp	1 nod bort
2,1-> 3,1	
3,1-> 3,2 och 4,1	2 noder bort
3,2-> 2,2 och 3,3	3 noder bort
4,1-> 4,2	
2,2-> 2,3	4 noder bort
3,3-> 3,4	
4,2-> 4,3, **framme**	



Graf – sökning på bredden

```
public static <T> ArrayList<Edge<T>> breadthFirstSearch(Graph<T> graph, T from, T to) {
   LinkedList<Edge<T>> queue = new LinkedList<Edge<T>>();
   HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
   boolean arrived = from.equals(to);
   Edge<T> edge;

   visited.put(from,null);
   queue.addAll(graph.getAdjacentList(from));
   while( !queue.isEmpty() && !arrived ) {
      edge = queue.removeFirst();
      if(!visited.containsKey(edge.getTo())) {
         visited.put(edge.getTo(),edge);
         queue.addAll(graph.getAdjacentList(edge.getTo()));
         arrived = to.equals(edge.getTo());
      }
   }
   return getPath(from, to, visited);
}
```

GraphExample.java GraphSearch.java

Minimum Spanning Tree

Den lägsta kostnaden att sammanbinda noderna i en graf beskrivs av ett ***minimalt uppspännande träd*** (***Minimal Spanning Tree***). En graf kan innehålla flera likvärdiga lösningar.

Algoritm (Input: Grafen *graph* och en *node* i grafen)

Skapa en graf, *mst*
Skapa en minheap, *heap*
Lägg till *node* i *mst*
Alla bågar som går från *node* placeras i *heap*
Så länge det finns fler bågar i *heap*
    Ta båge ur *heap* (det är bågen med lägst vikt) och lagra i *edge*
    Lagra noden som *edge* leder till i *node*
    Om *node* ej finns i *mst*
        Lägg till *node* i *mst*
        Lägg till *edge* i *mst*
        Alla bågar som går från *node* placeras i *heap*
returnera *mst*

Minimum Spanning Tree

Den lägsta kostnaden att sammanbinda noderna i en graf beskrivs av ett ***minimalt uppspännande träd*** (***Minimal Spanning Tree***). En graf kan innehålla flera likvärdiga lösningar.

```
public static <T> Graph<T> minimumSpanningTree(Graph<T> graph, T from) {
   Graph<T> minmumSpanningTree = new Graph<T>();
   ArrayHeap<Edge<T>> heap = new arrayHeap<Edge<T>>(10);
   ArrayList<Edge<T>> adjacentList = graph.getAdjacentList(from);
   Edge<T> edge;

   minmumSpanningTree.addVertex(from);
   for(Edge<T> e : adjacentList)
      heap.insert(e);
   while( heap.size()>0 ) {
      edge = heap.delete();
      if(!minmumSpanningTree.containsVertex(edge.getTo())) {
         minmumSpanningTree.addVertex(edge.getTo());
         minmumSpanningTree.addEdge(edge.getFrom(), edge.getTo(), edge.getWeight());
         adjacentList = graph.getAdjacentList(edge.getTo());
         for(Edge<T> e : adjacentList)
            heap.insert(e);
      }
   }
   return minmumSpanningTree;
}
```

GraphSearch.java

Dijkstra – optimerad sökning på bredden

```
public static <T> ArrayList<Edge<T>> dijkstraSearch(Graph<T> graph, T from, T to) {
   Candidate<T> candidate;
   PriorityQueue<Candidate<T>> queue = new PriorityQueue<Candidate<T>>();
   HashMap<T, Edge<T>> visited = new HashMap<T, Edge<T>>();
   boolean arrived = from.equals(to);
   ArrayList<Edge<T>> adjacentList = graph.getAdjacentList(from);

   visited.put(from,null);
   for(Edge<T> edge : adjacentList)
      queue.add(new Candidate<T>(edge,edge.getWeight()));
   while( !queue.isEmpty() && !arrived ) {
      candidate = queue.remove();
      if(!visited.containsKey(candidate.edge.getTo())) {
         visited.put(candidate.edge.getTo(),candidate.edge);
         adjacentList = graph.getAdjacentList(candidate.edge.getTo());
         for(Edge<T> edge : adjacentList)
            queue.add(new Candidate<T>(edge,candidate.getTotalWeight() + edge.getWeight()));
         arrived = to.equals(candidate.edge.getTo());
      }
   }
   return getPath(from, to, visited);
}
```

Candidate<T> implements Comparable<Candidate<T>>
- edge : Edge<T> - totalWeight : int
+Candidate(Edge<T>,int) +compareTo(Candidate<T>)

GraphSearch.java