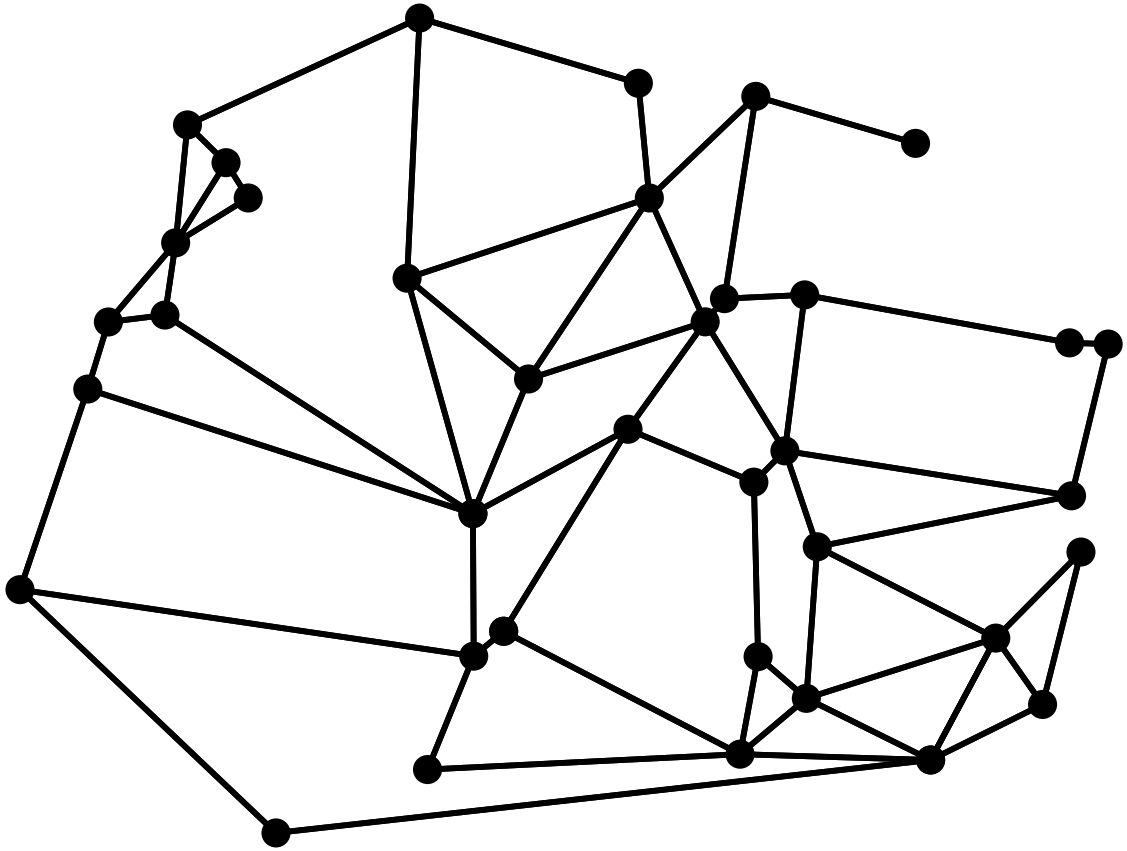
Graphen



Graphen

Definition

Ein ungerichteter Graph ist ein Paar (V, E), wobei V die Menge der Knoten und $E \subseteq \binom{V}{2}$ die Menge der Kanten ist.

Definition

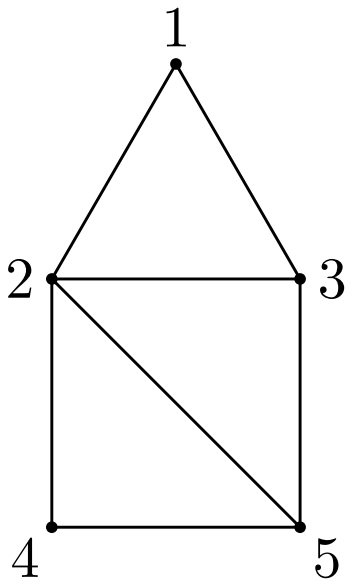
Ein gerichteter Graph ist ein Paar (V, E), wobei V die Menge der Knoten und $E \subseteq V \times V$ die Menge der Kanten ist.

Oft betrachten wir Graphen mit Knoten- oder Kantengewichten.

Dann gibt es zusätzlich Funktionen $V \to \mathbf{R}$ oder $E \to \mathbf{R}$.

Darstellung von Graphen

Adjazenzmatrix

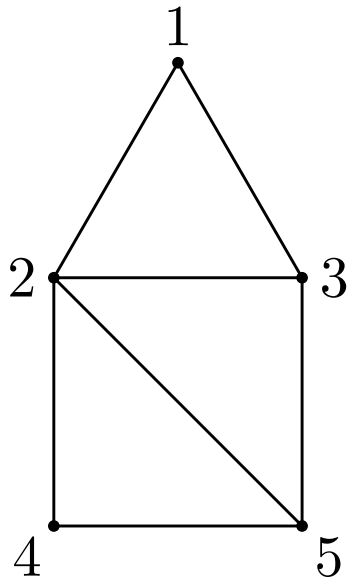


$$\begin{pmatrix} \cdot & 1 & 1 & 0 & 0 \\ \cdot & \cdot & 1 & 1 & 1 \\ \cdot & \cdot & \cdot & 0 & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Speicherbedarf: $\Theta(|V|^2)$

Für gerichtete Graphen wird die ganze Matrix verwendet.

Darstellung von Graphen



Adjazenzliste

1	2, 3
2	1, 3, 4, 5
3	1, 2, 5
4	2, 5
5	2, 3, 4

Speicherbedarf: $\Theta(|V| + |E|)$.

In $O(n^2)$ Schritten kann zwischen beiden Darstellungen konvertiert werden.

Darstellung von Graphen

Java

```
public class Graph {
 int n; //number of nodes
 int m; //number of edges
 Set<Node> nodes;
 Map<Node, List<Node, Edge>> neighbors;
```

Wir wählen die Darstellung durch eine Adjazenzliste.

Java

```
class Node implements Comparable⟨Node⟩ {
   String name;
   double weight;
   static int uniq = 0;
   int f;
   public Node() {f = uniq; name = ":" + uniq++; weight = 0.0; }
   public Node(double w) {this(); weight = w; }
   public String toString() {return name; }
   public int hashCode() {return f; }
   public boolean equals(Object o) {
      if (this ≡ o) return true;
      if (!(o instanceof Node)) return false;
      return f ≡ ((Node)o).f;
   }
   public int compareTo(Node n) {
      if(weight ≡ n.weight) return f − n.f;
      else if(weight < n.weight) return − 1;
      else return 1;
   }
}
```

Java

```
class Edge implements Comparable⟨Edge⟩ {
   Node s, t;
   double weight;
   static int uniq = 0;
   int f;
   public Edge(Node i, Node j) {f = uniq++; s = i; t = j; weight = 0;}
   public Edge(Node i, Node j, double w) {this(i, j); weight = w;}
   public double weight() {return weight;}
   public void setweight(double d) {weight = d;}
   public int hashCode() {return f;}
   public boolean equals(Object o) {
      if (this ≡ o) return true;
      if (!(o instanceof Edge)) return false;
      return f ≡ ((Edge)o).f;
   }
   public int compareTo(Edge e) {
      if(weight ≡ e.weight) return f − e.f;
      else if(weight < e.weight) return − 1;
      else return 1;
   }
}
```

Java

```
public Node addnode() {
    Node newnode = new Node();
    nodes.insert(newnode);
    neighbors.insert(newnode, new List<Node, Edge>());
    //n++;
    return newnode;
}
```

Java

```
public void addedge(Node u, Node v, Double w) {
  neighbors.find(u).insert(v, new Edge(u, v, w));
  m++;
}
```

Java

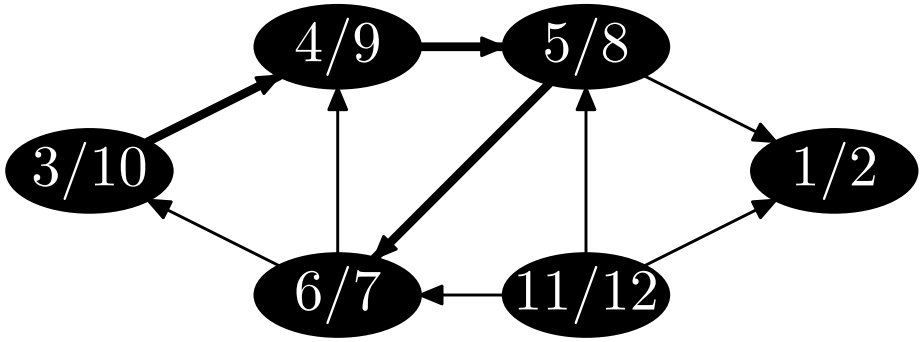
```
public boolean isadjacent(Node u, Node v) {
  return neighbors.find(u).iselement(v);
}
```

Tiefensuche

Tiefensuche ist ein sehr mächtiges Verfahren, das iterativ alle Knoten eines gerichteten oder ungerichteten Graphen besucht.

- Sie startet bei einem gegebenen Knoten und färbt die Knoten mit den Farben weiß, grau und schwarz.
- Sie berechnet einen gerichteten Tiefensuchwald, der bei einem ungerichteten Graph ein Baum ist.
- Sie ordnet jedem Knoten eine Anfangs- und eine Endzeit zu.
- Alle Zeiten sind verschieden.
- Die Kanten des Graphen werden als Baum-, Vorwärts-, Rückwärts- oder Querkanten klassifiziert.

Tiefensuche – Beispiel



- Die Kanten des Tiefensuchwaldes sind dick dargestellt.
- Ein Knoten ist anfangs weiß.
- Ein Knoten ist grau, während er aktiv ist.
- Danach wird er schwarz.

Java

```
public void DFS(Map⟨Node, Integer⟩ d, Map⟨Node, Integer⟩ f,
    Map⟨Node, Node⟩ p) {
  SimpleIterator⟨Node⟩ it;
  Map⟨Node, Integer⟩ color = new Hashtable⟨Node, Integer⟩();
  for(it = nodeiterator(); it.more(); it.step())
    color.insert(it.key(), WHITE);
  int time = 0;
  for(it = nodeiterator(); it.more(); it.step())
    if(color.find(it.key()) ≡ WHITE)
      time = DFS(it.key(), time, color, d, f, p);
}
```

Java

```
public int DFS(Node u, int t,
    Map<Node, Integer> c, Map<Node, Integer> d,
    Map<Node, Integer> f, Map<Node, Node> p) {
  d.insert(u, ++t);
  c.insert(u, GRAY);
  Iterator<Node, Edge> adj;
  for(adj = neighbors.find(u).iterator(); adj.more(); adj.step())
    if(c.find(adj.key()) ≡ WHITE) {
      p.insert(adj.key(), u);
      t = DFS(adj.key(), t, c, d, f, p);
    }
  f.insert(u, ++t);
  c.insert(u, BLACK);
  return t;
}
```