

Praxis des Programmierens

Graphentheorie

Branch & Bound

Traveling Salesman

Wintersemester 2001/2002

Jens Krinke, Prof. Dr. Snelting

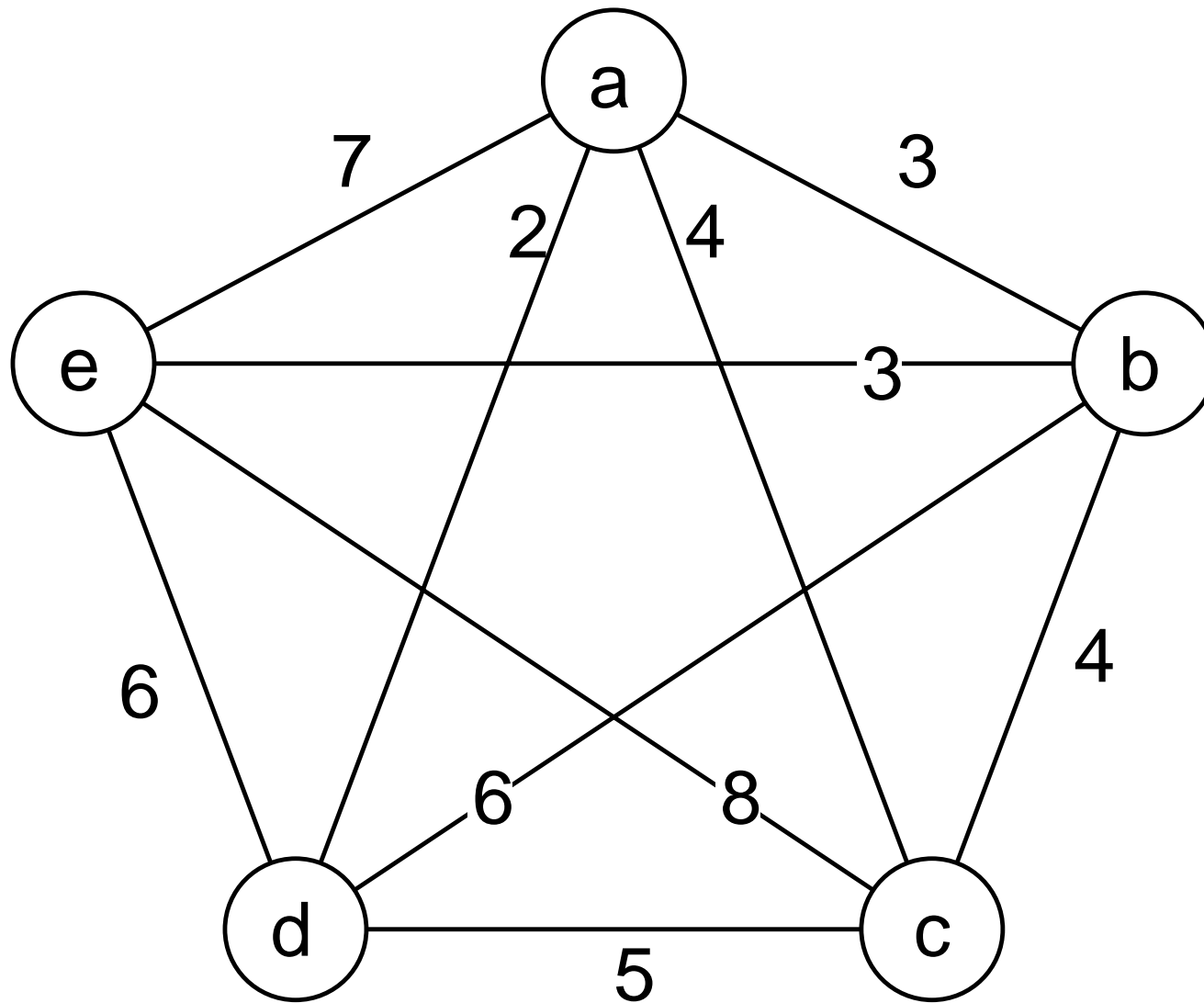
Lehrstuhl Softwaresysteme

Universität Passau



Das Problem des Handlungsreisenden

- Gegeben sei ein ungerichteter Graph
- Zu jeder Kante sei ihre Länge gegeben.
- Gesucht ist die kürzeste geschlossene Tour, die jeden Knoten genau einmal berührt.
- TSP ist **NP-vollständig**,
d.h. alle bekannten Implementierungen laufen in $O(2^{|V|})$.



Graphen

- mathematische Definition:

$$G = (V, E) \text{ mit } \begin{cases} V & \text{Menge der Knoten} \\ E & \text{Menge der Kanten} \end{cases}$$

- G **gerichtet** $\Leftrightarrow E \subseteq V \times V$
- G **ungerichtet** $\Leftrightarrow E$ Menge von 2-elementigen Teilmengen von 2^E
- Auf den Kanten wird häufig eine **Gewichtsfunktion** $k : E \rightarrow R$ definiert, die jeder Kante ein **Gewicht** zuordnet.

Implementierung von Graphen

- Als Adjazenzmatrix:

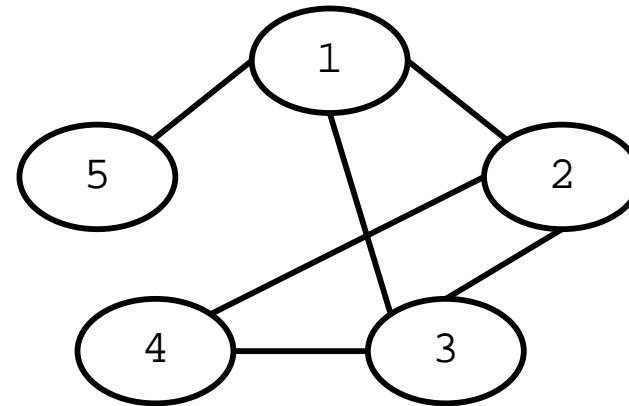
$$A[i, j] = \begin{cases} 1, & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

- Als **Listenstruktur**, der Graph besteht dabei aus einer verketteten Liste aller Knoten
 - **Knotenliste**: In diesem Fall verfügt jeder Knoten über eine Liste aller adjazenter Knoten
 - **Kantenliste**: Hier verfügt jeder Knoten über eine Liste aller inzidenter Kanten

Implementierung von Graphen: Beispiele

Sei $G = (V, E)$ ein Graph mit

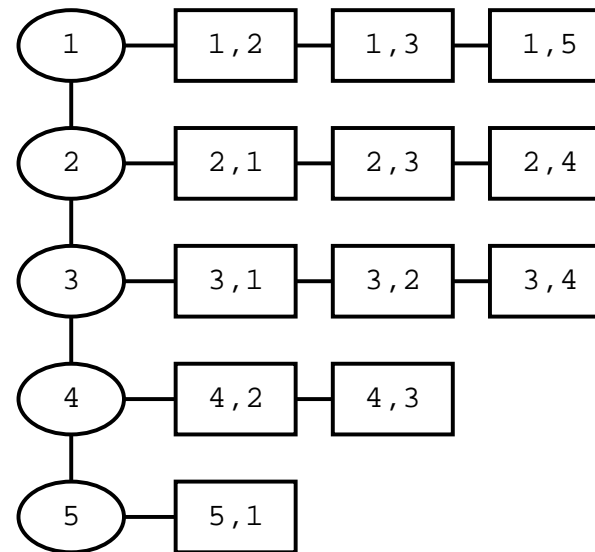
- $V = \{1, 2, 3, 4, 5\}$,
- $E = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$.



Graph als Adjazenzmatrix:

$$A[i, j] = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Der Graph als Kantenliste:



Arbeiten mit Adjazenzmatrizen

- Implementierung als 2-dim Arrays. Vorsicht, erste Array-Position hat in JAVA Index 0!
- **Vorteil:** Zugriff auf eine bestimmte Kante in $O(1)$.
- **Nachteil:** Nicht geeignet zum Traversieren des Graphen.

Iteration über alle Kanten des Graphen:

```
for (int i = 0; i < nodes.size() - 1; ++i) {  
    for (int j = i; j < nodes.size() - 1; ++j) {  
        // ok, hier Zugriff auf Kante (i,j)  
        // bzw. auch (j,i)  
    }  
}
```

Arbeiten mit Kantenlisten

- Aufwendigere Implementierung, Zugriff: $O(n)$
- **Vorteil:** leichtes Traversieren des Graphen.
- **Nachteil:** Aufwand, Zugriffskosten

Absuchen aller Knoten des Graphen mit Tiefensuche:

```
void dfs() {
    if (this.isMarked()) {
        return;
    } else {
        // Aktion...
        this.mark();
        for each child do {
            child.dfs();
        }
    }
}
```

Kombination Adjazenzmatrizen + Kantenlisten

- Kombination beider Verfahren, parallele Implementierung von Adjazenzmatrix und Listen.
- Je nach Bedarf Zugriff über Matrix oder Liste, garantiert optimalen Zugriff.
- **Nachteil:** Speicherplatzbedarf.

```
class Graph {  
    boolean[][] adjacencyMatrix =  
        new boolean[MAX_NUM_NODES][MAX_NUM_NODES];  
    List[] edgeLists =  
        new List[MAX_NUM_NODES];  
    ...  
}
```

Branch & Bound

- Algorithmus geeignet zum Lösen von **Optimierungsproblemen**.
- Zerlegt Problem in Teilprobleme, Struktur dieser Teil-Probleme bildet einen **Baum**.
- Jedem **Baumknoten** wird durch eine **Bewertungsfunktion** ein Wert zugeordnet.
- Suche nach Optimum beginnt mit besonders aussichtsreichen Zweigen.
- Zweige, deren Bewertung „schlecht“ ist, werden **abgeschnitten**.
- Mittel zum Aufbau des Suchbaumes: **Rekursion**

Branch & Bound: TSP Pseudocode

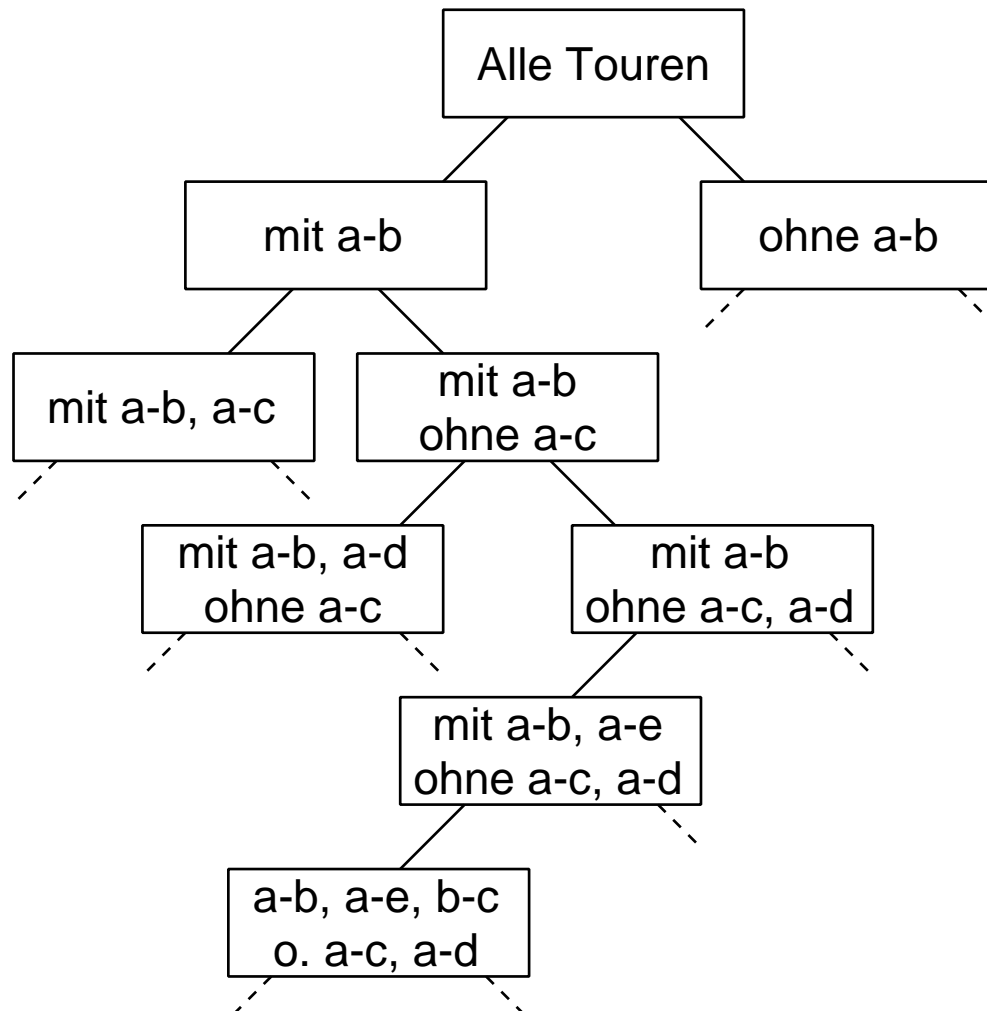
```
proc branchAndBound (int: edgeNum, set of edges: selected) {  
    if (selected ist Lösung) {  
        if (selected ist bessere Lösung)  
            opt := selected;  
    } else {  
        if (Lösung mit neuer Kante edgeNum ist möglich)  
            branchAndBound(edgeNum+1, selected + {edgeNum});  
        if (Lösung ohne neue Kante edgeNum ist möglich)  
            branchAndBound(edgeNum+1, selected);  
    }  
}
```

Wann ist neue Kante **möglich** bzw. **nicht möglich**?

Kantenliste

- BitSet selected
- Kante unter Betrachtung: $n = \text{edgeNum}$
- Drei Werte: *ja, nein, unbekannt*

1	2	3	4	...	$n - 1$	n	$n + 1$	$n + 2$...
X	-	X	-	...	X	?	?	?	...



Abschneiden von Teilbäumen

- **Suchraum:** 2^E , aufgespannt durch die Rekursion von B&B.
- Wann können Rekursionsäste abgeschnitten werden?
 - Keine Kanten hinzufügen,
die eine Rundreise unmöglich machen.
 - Keine Kanten weglassen,
die in der Rundreise vorhanden sein müssen.
 - Keine Kanten hinzufügen,
die eine zu lange Rundreise erzeugen.
(Länger als die bisher kürzeste gefundene.)

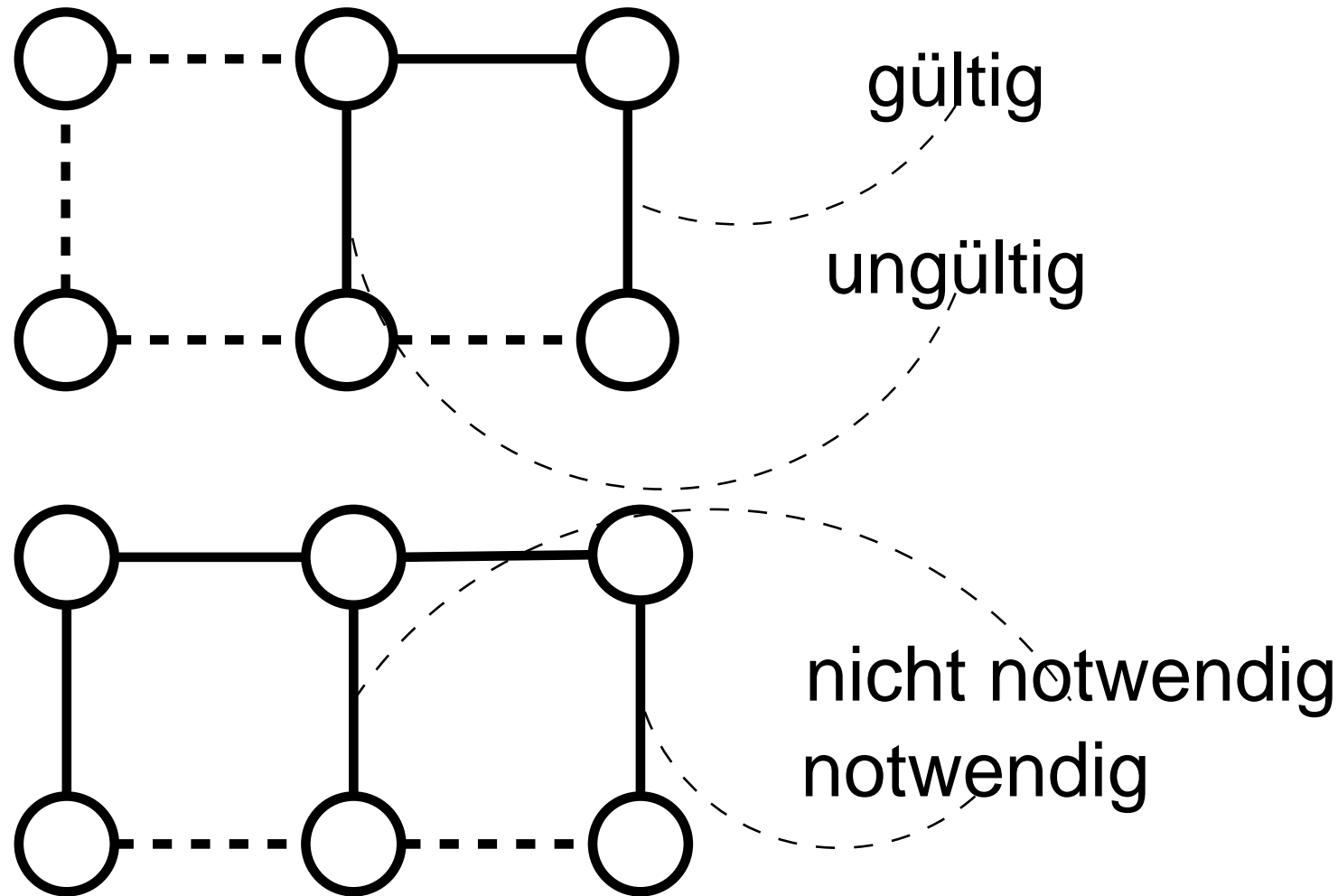
Eine Kante kann hinzugefügt werden,

- wenn sie den Zyklus nicht vorzeitig schliesst.
- wenn sie nicht die dritte hinzugefügte inzidente Kante eines Knoten ist.

Eine Kante kann weggelassen werden,

- wenn sie nicht die letzte unbekannte inzidente Kante eines Knoten ist, der nur eine hinzugefügte inzidente Kante hat.
- wenn sie nicht die vorletzte unbekannte inzidente Kante eines Knoten ist, der noch keine hinzugefügte inzidente Kante hat.

Gestrichelte Linien markieren die bisherige Auswahl.



Greedy-Strategie

Die Kanten werden vorab der Länge nach sortiert.
Kürzere Kanten werden vor längeren Kanten probiert.

Obere Schranke

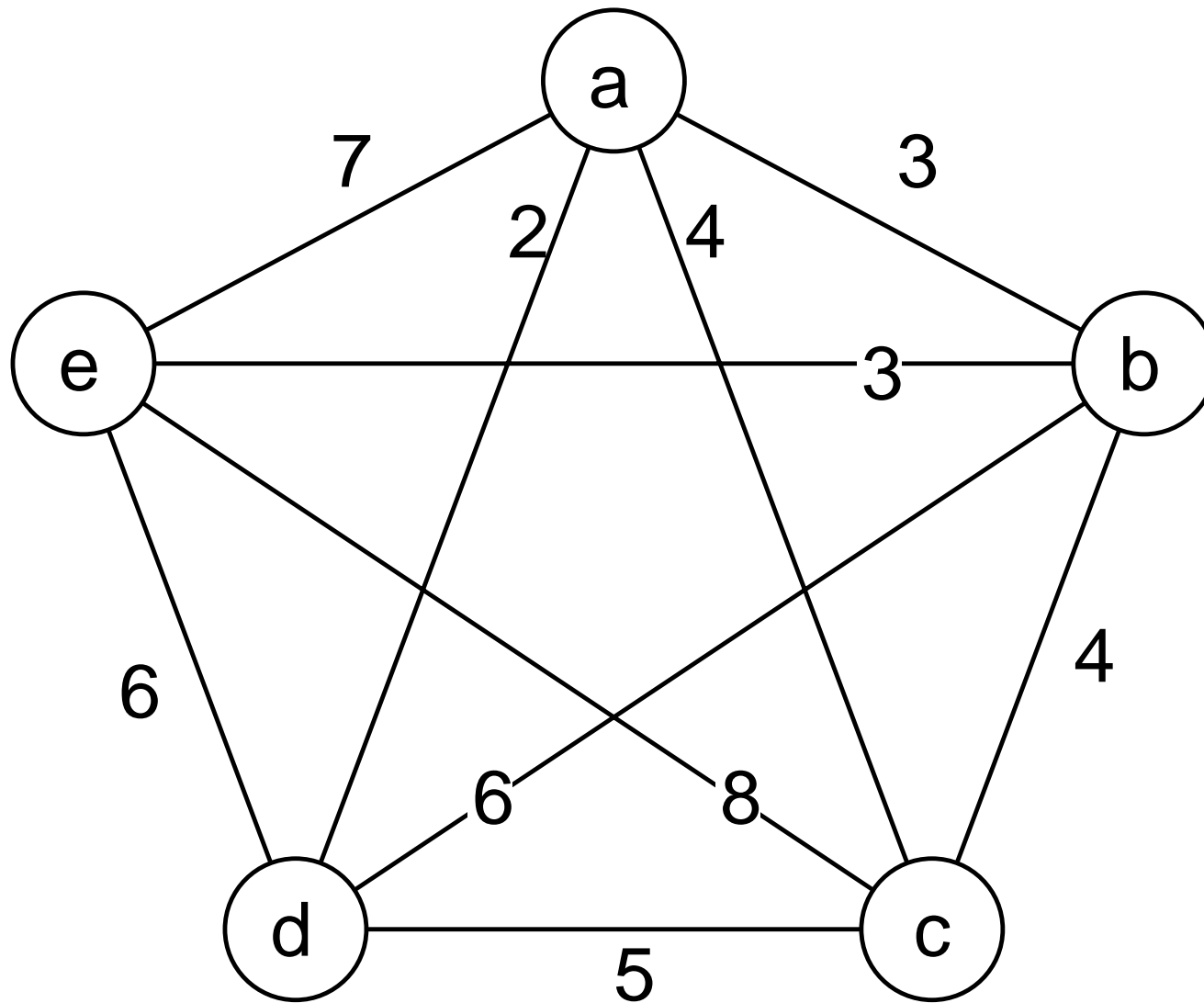
Die Gesamtlänge der Kanten in `selected`
plus die Länge der Kante `edgeNum`
darf nicht länger sein als die beste bisherige Lösung.

Untere Schranke

Ausgehend von den Kanten in `selected` wird eine *untere Schranke* der Länge möglicher Rundreisen bestimmt.

- Zu jedem Knoten bestimmt man die Länge der beiden kürzesten inzidenten Kanten, die in `selected` noch unbekannt sind.
- Hat ein Knoten eine inzidente Kante in `selected` gesetzt, bestimmt man die Länge dieser Kante und der kürzesten inzidenten Kante, die in `selected` noch unbekannt ist.
- Hat ein Knoten zwei inzidente Kanten in `selected` gesetzt, bestimmt man Länge dieser beiden Kanten.

$$\text{Lower Bound} := \frac{\sum \text{aller berechneter Längen}}{2}$$



Erweiterter Algorithmus

```
proc branchAndBound (int: edgeNum, set of edges: selected) {
  if (selected ist Lösung) {
    if (selected ist bessere Lösung)
      opt := selected;
  } else {
    if (Lösung mit neuer Kante edgeNum ist möglich)
      bound1 = lowerBound(edgeNum, selected + {edgeNum})
    else
      bound1 = INFINITY
    if (Lösung ohne neue Kante edgeNum ist möglich)
      bound2 = lowerBound(edgeNum, selected)
    else
      bound2 = INFINITY
    if (bound1 < bound2)
      if (bound1 < Länge von opt)
        branchAndBound(edgeNum + 1, selected + {edgeNum});
      if (bound2 < Länge von opt)
        branchAndBound(edgeNum + 1, selected);
    else
      andersrum
  }
}
```

Anforderungen

- Knoten sind durchnummeriert: $1 \dots k$
- Textschnittstelle:
add $i j l$
delete $i j$
tsp
quit
- Ausgabe der Rundreise:
 $(1 n_2 l_1) (n_2 n_3 l_2) \dots (n_k 1 l_k)$
total length = l , m calls

```
$ java TSP
```

```
prompt> add 1 2 3
```

```
prompt> add 1 3 4
```

```
prompt> add 1 5 7
```

```
prompt> add 3 1 1
```

```
Error! multiple add
```

```
prompt> add 1 4 2
```

```
prompt> tsp
```

```
Error! no roundtrip found
```

```
total length = 0, 3 calls
```

```
prompt> add 2 3 4
prompt> add 4 2 6
prompt> add 2 5 3
prompt> add 3 4 5
prompt> add 3 5 8
prompt> add 4 5 6
prompt> tsp
(1 4 2) (4 5 6) (5 2 3) (2 3 4) (3 1 4)
total length = 19, 19 calls
prompt> del 1 3
prompt> tsp
(1 4 2) (4 3 5) (3 5 8) (5 2 3) (2 1 3)
total length = 21, 16 calls
prompt> quit
$ _
```

Hinweise

- Benutzen Sie eine 50×50 Adjazenzmatrix.
- Stellen Sie zuerst sicher, dass Ihr Graph korrekt ist.
- Bereiten Sie Ihren Graphen beim Aufruf von `tsp` vor:
 - Eine sortierte Kantenliste.
 - Für Knoten: Listen der inzidenten Kanten.
- Stellen Sie die korrekte Berechnung von Rundreisen sicher, bevor Sie `TowerBound` implementieren.
- Implementieren Sie Funktionen für Teilaufgaben.

```
class TSPEdge implements Comparable {  
    ...  
    public TSPEdge(node1, node2, length);  
    public int getNode1();  
    public int getNode2();  
    public int getLength();  
    ...  
}
```

```
class GraphException extends Exception { ... }
```

```
class TSPGraph {  
    void addEdge(int node1, int node2, int dist) throws GraphException;  
    void deleteEdge(int node1, int node2) throws GraphException;  
    void calcTSP();  
    List getTour();  
    int getTourLength();  
    int getTspCallCount();  
}
```

Wichtig:

Keep it simple!

Java-Bibliothek

- Comparable
- Collections.sort
- List
- ArrayList
- LinkedList
- BitSet

Bewertung

1. Das Programm berechnet keine korrekten Rundreisen.
2. Das Programm berechnet korrekte Rundreisen.
3. Ihr Programm implementiert und benutzt LowerBound.
4. Ihr Programm hat eine Aufrufzahl
in der *Größenordnung* der Referenzlösung.

Lösungen mit einer besseren Aufrufzahl werden prämiert.

Testfallbeschreibung

- Testen Sie Ihr Programm ausführlich.
- Kann Ihr Programm Problemgraphen behandeln?
- Reichen Sie eine Testfallbeschreibung mit ein:
Datei `TestTSP.java` enthält nur Kommentar.

```
/*
```

```
Testfallbeschreibung
```

```
-----
```

Der erste Testfall stammt aus der Aufgabenstellung:

```
add 1 2 3
```

```
add 1 3 4
```

```
add 1 5 7
```

```
add 3 1 1
```

```
: Error! multiple add
```

```
add 1 4 2
```

```
tsp
```

```
: Error! no roundtrip found
```

```
: total length = 0, 3 calls
```

```
add 2 3 4
```

```
add 4 2 6
```

```
⋮
```

Der dritte Testfall prüft nacheinander die folgenden Fälle:

1. Leerer Graph
2. Nur eine Kante
3. Zwei Kanten
4. Dritte Kante schliesst Rundreise

```
tsp
: Error! no roundtrip found
: total length = 0, 1 calls
add 1 2 1
```

```
tsp
: Error! no roundtrip found
: total length = 0, x calls
add 2 3 2
```

```
tsp
: Error! no roundtrip found
: total length = 0, x calls
add 3 1 3
```

```
⋮
```

Termine

- Bis Mittwoch, 9. Januar 2002, 12:00 Uhr:
Erste Lösung einreichen (empfohlen)
- Bis Mittwoch, 23. Januar 2002, 12:00 Uhr:
Erste Lösung einreichen (fest)