Programmierkurs Python II

Stefan Thater & Michaela Regneri
Universität des Saarlandes
FR 4.7 Allgemeine Linguistik (Computerlinguistik)

Übersicht

- Topologische Sortierung (einfach)
- Kürzeste Wege finden (ziemlich einfach)
- Größter gemeinsamer Teilgraph
  - Modulares Graph-Produkt (nicht formal)
  - Cliquen (etwas komplexer)

Ein Graph zum Frikadellen-Machen

„X → Y“: X muss erledigt sein, damit Y gemacht werden kann

Zwiebel Schälen → Zwiebeln hacken

Zwiebeln hacken → Zwiebeln in die Schüssel

Hackfleisch auspacken → Hackfleisch in die Schüssel

∅ → Gewürze in die Schüssel

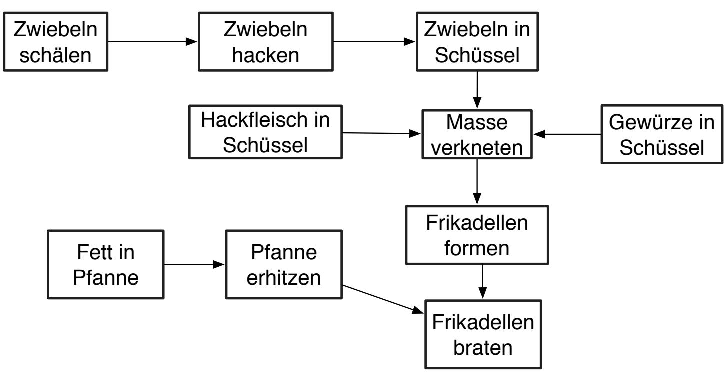
[alles in die Schüssel] → Masse Verkneten

Masse Verkneten → Frikadellen formen

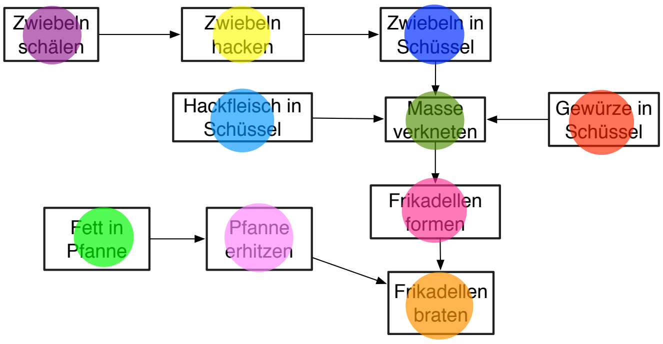
Frikadellen Formen → Frikadellen braten

[...]

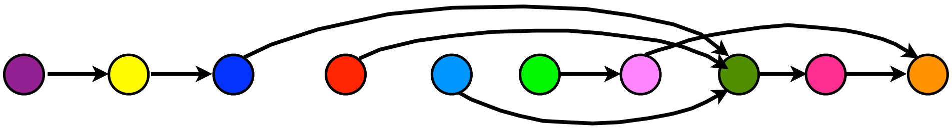
Ein Graph zum Frikadellen-Machen



Topologische Sortierung - graphische Variante

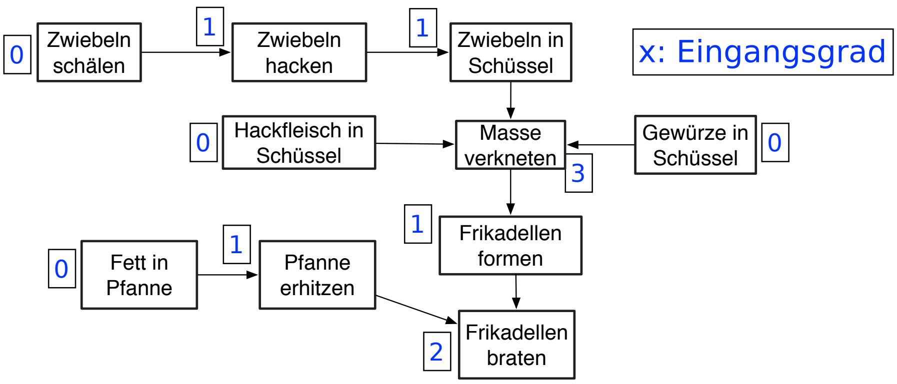


lineare Ordnung, alle Kanten zeigen nach rechts



Topologische Sortierung - Algorithmus (im Bild)

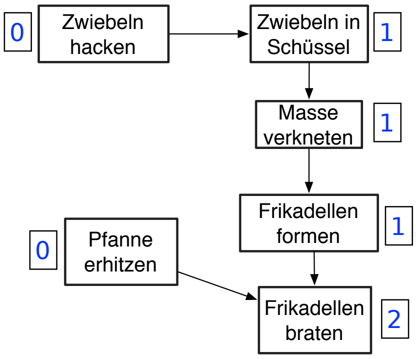
Schritt 1



Topologische Sortierung - Algorithmus (im Bild)

Schritt 2

x: Eingangsgrad



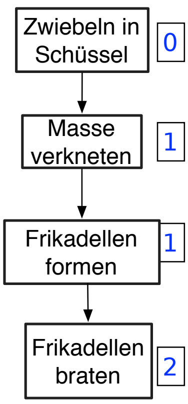
Zwiebeln schälen | Fett in Pfanne | Gewürze in Schüssel | Hackfleisch in Schüssel

Beliebige Ordnung untereinander

Topologische Sortierung - Algorithmus (im Bild)

Schritt 3

x: Eingangsgrad



Zwiebeln schälen | Fett in Pfanne | Gewürze in Schüssel | Hackfleisch in Schüssel | Zwiebeln hacken | Pfanne erhitzen

Topologische Sortierung - Algorithmus (im Bild)

Schritt 6

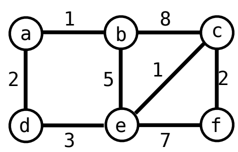
x: Eingangsgrad

Komplexität:
bester Fall: O(n)
worst case: $O(n^2)$

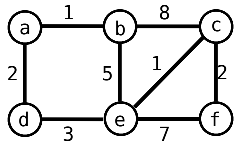
Frikadellen braten | 0

Zwiebeln schälen | Fett in Pfanne | Gewürze in Schüssel | Hackfleisch in Schüssel | Zwiebeln hacken | Pfanne erhitzen | Zwiebeln in Schüssel | Masse verkneten | Frikadellen formen

Kürzeste Wege finden



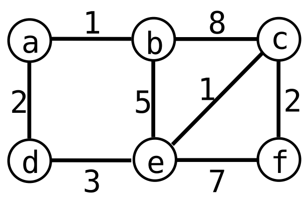
- *„Single source shortest path problem“*
- Aufgabe: Gegeben einen Knoten im Graph, finde die kürzesten Wege zu allen anderen Knoten
- Meistens assoziiert mit Kantengewichten (kürzester Weg = *billigster* Weg)
- Der Weg vom Startknoten zu unerreichbaren Knoten kann unendlich sein (unzusammenhängende ungerichtete / nicht stark zusammenhängende gerichtete Graphen)



Dijkstra-Algorithmus

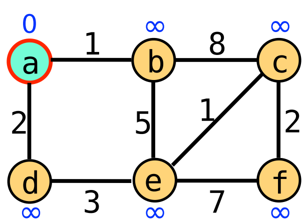
- Initialisierung:
  - Startknoten: Distanz 0, *permanent*, aktiv
  - andere: Distanz ∞, *temporär* (nicht aktiv)
- So lange es Knoten mit temporären Nachbarn gibt...
  - Berechne die Distanz der temp. Nachbarn des aktiven Knoten (Distanz aktiver Knoten + Kantengewicht)
  - wenn berechnete Distanz > notierter Distanz:
    - aktualisiere notierte Distanz;
    - notiere aktiven Knoten als Vorgänger
  - neuer aktiver Knoten: Knoten mit kleinster (Gesamt-)Distanz; markiere den als permanent

Dijkstra-Algorithmus



Startknoten hier: a

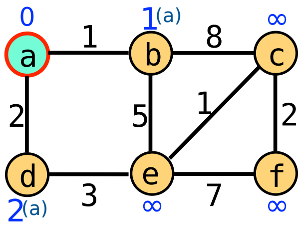
Initialisiere Distanzen:
Startknoten = 0, alle anderen = ∞



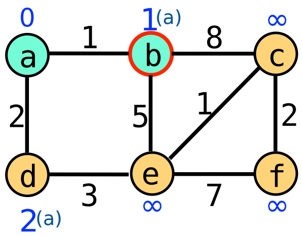
Initialisiere Markierungen:
Startknoten = permanent
alle anderen = temporär

Initialisiere aktiven Knoten:
Startknoten

Dijkstra-Algorithmus

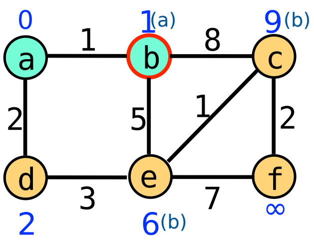


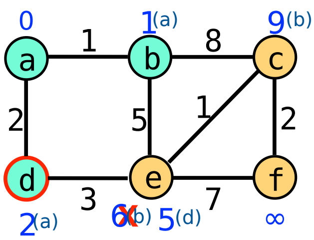
- Berechne alle Nachbar-Distanzen zum aktiven Knoten (nur zu temporären Knoten)
- Wenn die berechnete Distanz kleiner ist als die bisher gefundene, aktualisiere Distanz und Vorgänger

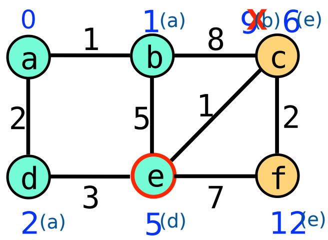


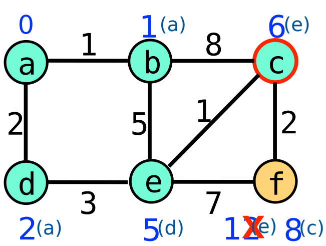
- neuer aktiver Knoten: temporärer Knoten mit kleinster Distanz
- markiere neuen aktiven Knoten als permanent

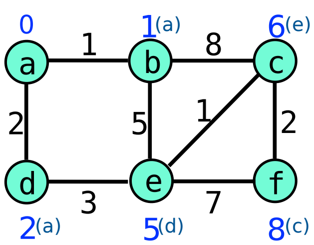
Dijkstra-Algorithmus







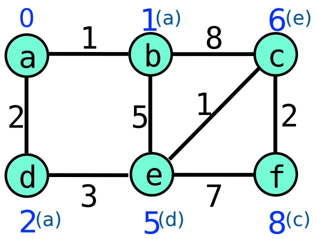


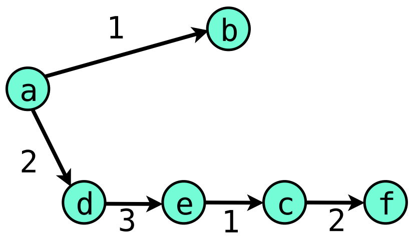


fertig

Dijkstra-Algorithmus - Ergebnis

- jeder Knoten: minimale Distanz, und direkter Vorgänger
- Ableitbarer „Spannbaum“: ein Teilgraph des Graphen, der ein Baum ist und alle Knoten des Graphen enthält
- die enthaltenen Kanten markieren hier die kürzesten Pfade



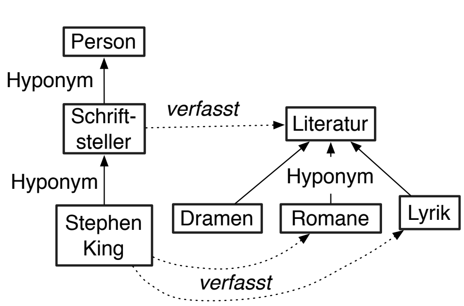


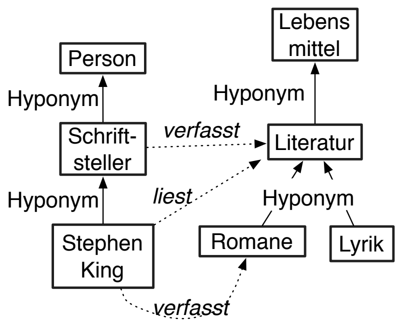
Kürzeste Wege finden

- Komplexität von Dijkstra: *O(Knoten * log(Knoten) + Kanten)*
- SSSP mit neg. gewichteten Kanten
  - Dijkstra erlaubt keine negativen Gewichte
  - SSSP mit negativen Gewichten (aber ohne negative Zyklen): *Bellman-Ford-Algorithmus*
- *All pairs shortest path problem*
  - Aufgabe: berechne für alle Paare von Knoten die kürzesten Pfade zwischen den Knoten
  - *Floyd-Warshall-Algorithmus*

Graph-Ähnlichkeit: Größter gemeinsamer Teilgraph

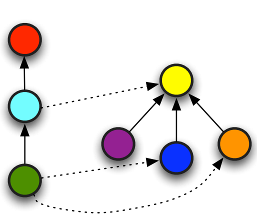
- Problem: Welches ist der maximale Graph, der Teilgraph von zwei Graphen ist?

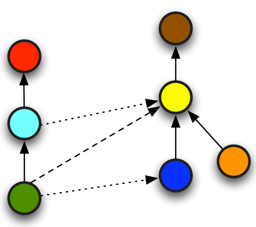




Graph-Ähnlichkeit: Größter gemeinsamer Teilgraph

- Eine Möglichkeit:
  - Berechne das *modulare Produkt* der beiden Graphen (ein neuer Graph, der beide Graphen repräsentiert)
  - finde den größten maximalen vollständigen Teilgraphen (Definition später)





Graph-Ähnlichkeit: Modulares Graph-Produkt (MGP)

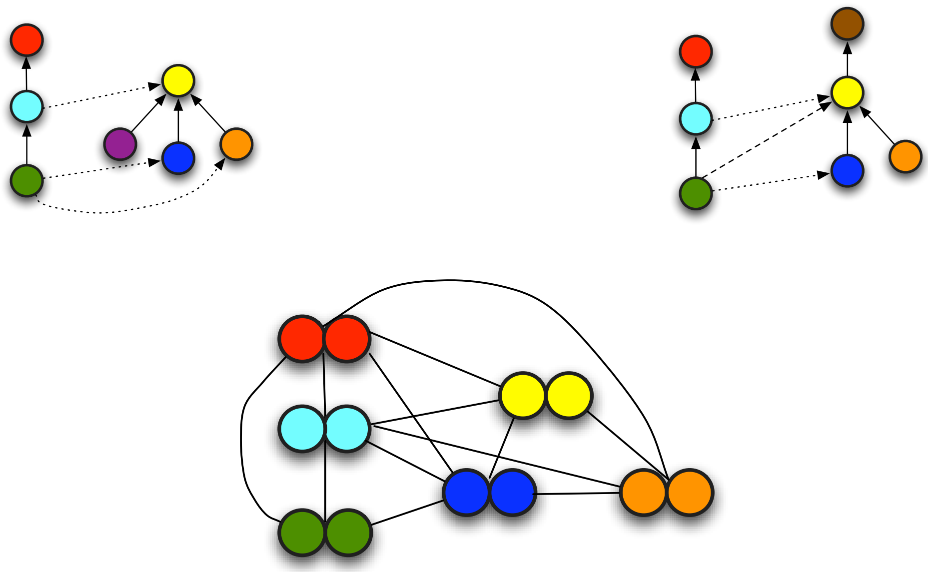
- oft angewendet für „reinen“ Graph isomorphismus (ungelabelte, ungerichtete Kanten, Knoten-Label egal)
- MGP von G=(V,E) und H=(V‘,E‘) ist P=(W,F) so dass
  - W = kartesisches Produkt von E und F
    [(u,v) | u ∈ V, v ∈ W]
  - F = [((u,v),(u‘,v‘))] für alle
    - (u,u‘) ∈ E & (v,v‘) ∈ E‘
    - (u,u‘)∉ E & ∉ (v,v‘) F
  - Knoten sind also Knotenpaare, Kanten repräsentieren „gleiche Verbindung“ in den Ursprungsgraphen

Graph-Ähnlichkeit: Modulares Graph-Produkt (MGP)

- Abwandlung hier für Knoten mit „Typen“ und ggf. unterschiedliche Kanten
- MGP von G=(V,E) und H=(V‘,E‘) ist P=(W,F) so dass
  - W = [(u,v) | u ∈ V, v ∈ W, *v≈u*]
    (hier: v‘s Knoten-Typ ist kompatibel mit u)
  - F = [{(u,v),(u‘,v‘)}] für alle
    - {u,u‘} ∈ E & {v,v‘} ∈ E‘
    - {u,u‘} ∉ E & {v,v‘} ∉ E‘

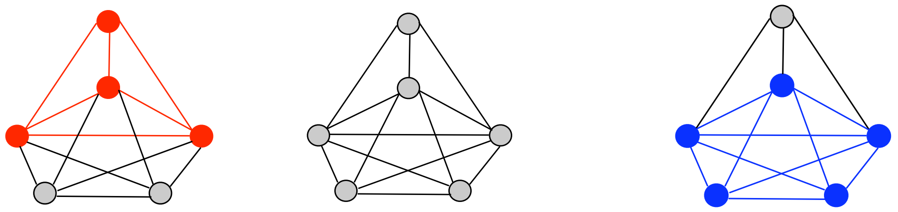
Zur Vereinfachung ignorieren wir hier Kantentypen, weil die Datenstruktur sie so nicht vorsieht.

Graph-Ähnlichkeit: Modulares Graph-Produkt (MGP)

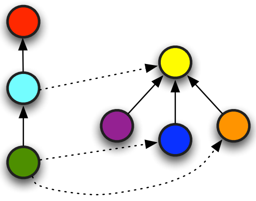


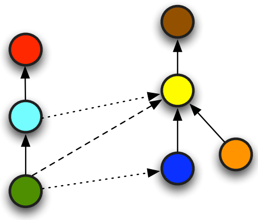
Graph-Ähnlichkeit: Cliquen

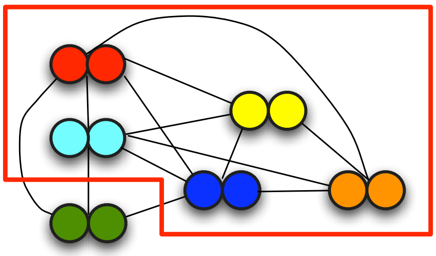
- Ein Graph G heißt *vollständig*, wenn alle seine Knoten paarweise verbunden sind
- Ein maximaler vollständiger Teilgraph (oder eine *Clique*) eines Graphen G ist ein Teilgraph G‘, so dass G‘ vollständig ist und es keinen vollständigen Teilgraphen G‘‘ von G gibt, so dass G‘ ein Teilgraph von G‘‘ ist.



Graph-Ähnlichkeit: größte Clique







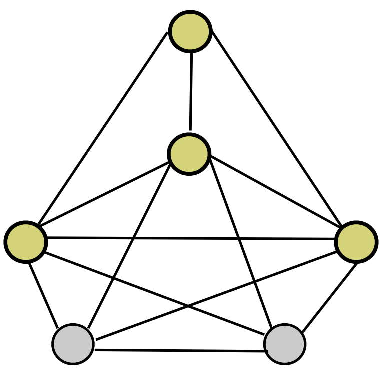
Cliquen finden - Prinzip

- beginne bei einem minimalen vollständigen Teilgraphen (= ein Knoten)
- mache eine Cliquen-Knotenmenge (CS) damit auf
- füge so lange Knoten zu CS, bis es keine Knoten mehr gibt, die mit CS zusammen eine noch unbekannte Clique ergeben
- Speichere CS als Clique und suche die nächste (mit einem anderen Startknoten)

Cliquen finden - Grundlagen des Algorithmus

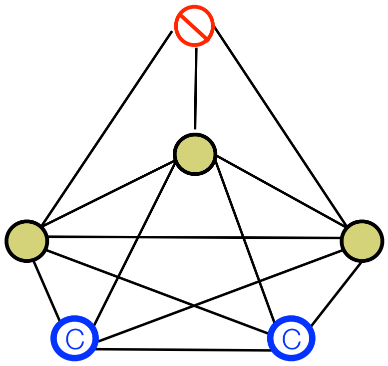
- eine aktuelle Cliquen-Menge CS (inital: leer)
- *Kandidaten*, Knoten, die mit allen Elementen aus CS verbunden sind (initial: alle Knoten)
- *Visited*, Knoten, die schon betrachtet wurden und für das aktuelle CS gültige Erweiterungen sind (= die zu schon gesehenen Cliquen führen) (initial: leer)
- ein aktueller aktiver Kandidat
- im aktuellen Schritt ignorierte Knoten

Cliquen finden - Erfolgsbedingung (finden einer Clique)



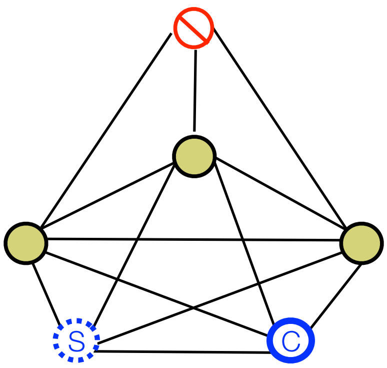
- keine Kandidaten mehr
- keine Elemente in *Visited*
  - sonst gäbe es Knoten, die die aktuelle Cliquen-Menge zu einer Clique erweitern können
  - der vollständige Teilgraph wäre nicht maximal

Cliquen finden - Algorithmus



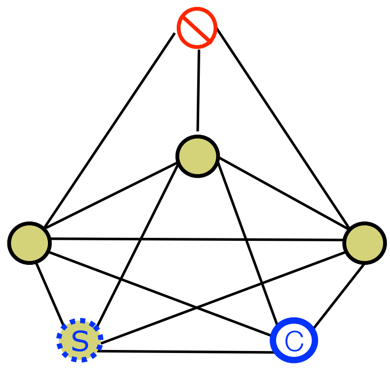
- in diesem Schritt: ein Knoten in *Visited* (= eine Clique schon gefunden)
- 3 Knoten in der Cliquen-Menge
- 2 Kandidaten übrig

Cliquen finden - Algorithmus



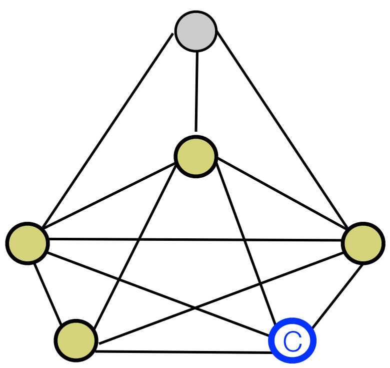
1. Wähle einen aktiven Kandidaten, entferne ihn aus der Kandidaten-Menge

Cliquen finden - Algorithmus



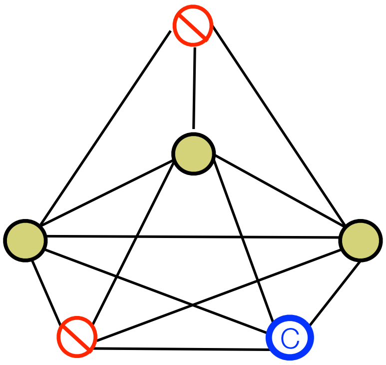
1. Wähle einen aktiven Kandidaten, entferne ihn aus der Kandidaten-Menge
2. *füge ihn zur Cliquen-Menge hinzu*

Cliquen finden - Algorithmus

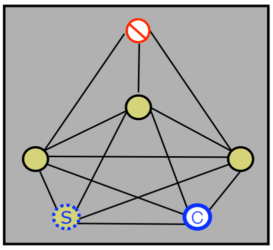


1. Wähle einen aktiven Kandidaten, entferne ihn aus der Kandidaten-Menge
2. füge ihn zur Cliquen-Menge hinzu
3. *gehe in die Rekursion mit neuen Mengen:*
   - Entferne alle Knoten von Kandidaten u. Visited, die nicht mit dem letzten aktiven Kandidaten verbunden sind)
   - beginne damit bei 1

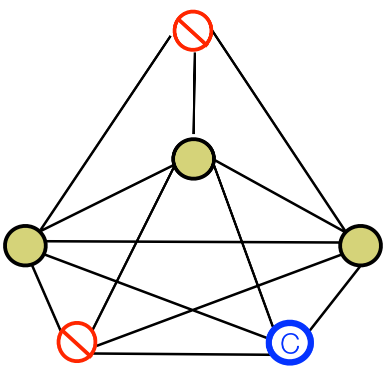
Cliquen finden - Algorithmus



5.Wenn Rekursion fertig:
- alte Mengen (*Visited* und Kandidaten) wieder zurück holen
- den alten aktiven Kandidaten zu *Visited* hinzufügen



Cliquen finden - Algorithmus

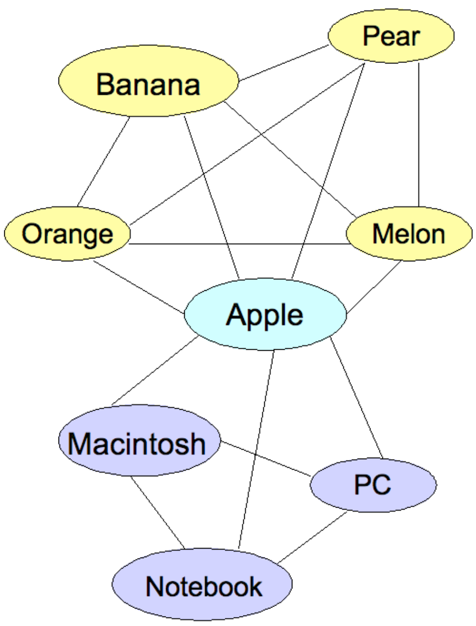


- Abbruchs-Bedingungen:
  - es gibt keine Kandidaten mehr
  - es gibt einen Knoten in *Visited*, der mit allen übrigen Kandidaten verbunden ist

Cliquen finden

- der Basis-Algorithmus ist sehr ineffizient
- Verbesserung des Standard-Falls: *Bron-Kerbosch-Algorithmus*
  - Prinzip: optimiere die Auswahl des nächsten aktiven Kandidaten, so dass möglichst früh abgebrochen wird
  - Ziel: möglichst früh ein Knoten in *Visited*, der zu allen Kandidaten adjazent ist
  - Für jeden Knoten in *Visited* wird die Anzahl der *nicht* Adjazenten Kandidaten gespeichert
  - Der nächste Kandidat hat immer eine Verbindung zu dem Knoten in Visited mit kleinsten Zähler

Mehr Cliquen



Zusammenfassung

- Heute: verschiedene Graph-Algorithmen
- Topologische Soritierung: „Plan-Management“
- Single Source Shortest Path: Kürzeste Wege
- Größter gemeinsamer Teilgraph
  - Modulares Graph-Produkt
  - Cliquen