Kap. 8: Lösen schwerer Probleme

Approximationsalgorithmen
Exakte Verfahren
Heuristische Verfahren

Lösen schwerer Probleme

- Wie kann man ein Problem P angehen, von dem man weiß/vermutet, dass es NP-schwer ist?

1. Exakte Verfahren
   - Algorithmus mit exponentieller Laufzeit, der P löst
   - wird angewendet, falls
     - exakte Lösung wichtig ist,
     - notwendige Rechenkapazität zur Verfügung steht
     - Eingabegrößen hinreichend klein sind
2. Approximationsalgorithmen
   - Algorithmus mit polynomieller Laufzeit, der P mit einem garantierten max. Fehler löst
   - wird angewendet, falls
     - 1. nicht möglich ist
     - Fehlerabschätzung bei der Lösung verlangt wird (z.B. Risikoabschätzung, stat. Analyse der Ergebnisse, etc.)
3. Heuristische Verfahren
   - Algorithmus mit polynomieller Laufzeit, der eine *wahrscheinlich* gute Lösung für P (ohne Gütegarantie!) berechnet

8.1 Approximationsalgorithmen

- Approximationsfaktor:
  - Annahme: jede Lösung hat positive Kosten
  - Ein Algorithmus A hat einen Approximationsfaktor $\rho(n)$, falls für jede Input-Größe n gilt: $\max(C/C^*, C^*/C) \leq \rho(n)$

    C: von A gefundene Lösung; C^*: optimale Lösung

    A heißt $\rho(n)$-Approximations-Algorithmus.

    Falls $\rho(n)$ konstant ist, auch ρ-Approximations-Algorithmus

    Ein 1-Approximations-Algorithmus ist ein exakter Algorithmus.
- Approximationsschema:
  - Algorithmus A für ein Problem P, der neben der Eingabe für P eine Zahl ε als Parameter hat.

    A heißt Approximationsschema, falls für jedes $\varepsilon > 0$ gilt:
    A ist ein $(1+\varepsilon)$-Approximations-Algorithmus.
  - A heißt Polynomzeit-Approximationsschema, falls A für jedes konstante $\varepsilon>0$ einen max. polynomiellen Laufzeitbedarf hat.

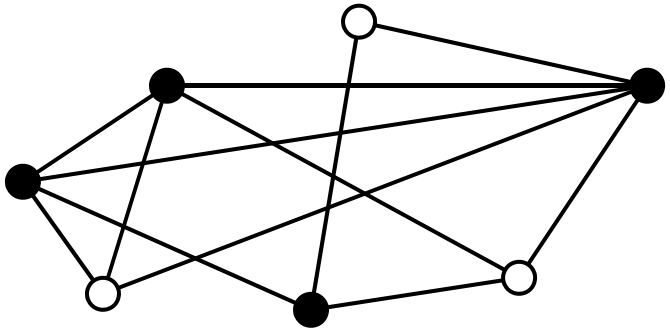
Ein Approximations-Algorithmus für Vertex-Cover

- Problem (Opt-Vertex-Cover):
  - gegeben: ungerichteter Graph G=(V,E)
  - gesucht: minimale Teilmenge $C \subseteq V$ mit:

    $$\forall (v,w) \in E : v \in C \text{ oder } w \in C$$

    Teilmengen mit dieser Eigenschaft werden als **Knotenüberdeckung** bezeichnet.

- Bsp:



schwarze Knoten
sind ein Vertex-Cover

- Opt-Vertex-Cover ist ein NP-schweres Optimierungsproblem.

Vertex-Cover ist NP-schwer

- VERTEX-COVER =

  { <G,k> | Der Graph G=(V,E) besitzt eine Knotenüberdeckung der Größe k }

- **Theorem 34.12:**

  Das k-Threshold-Problem VERTEX-COVER ist NP-vollständig.

  Beweis:

  Teil 1: VERTEX-COVER $\in$ NP

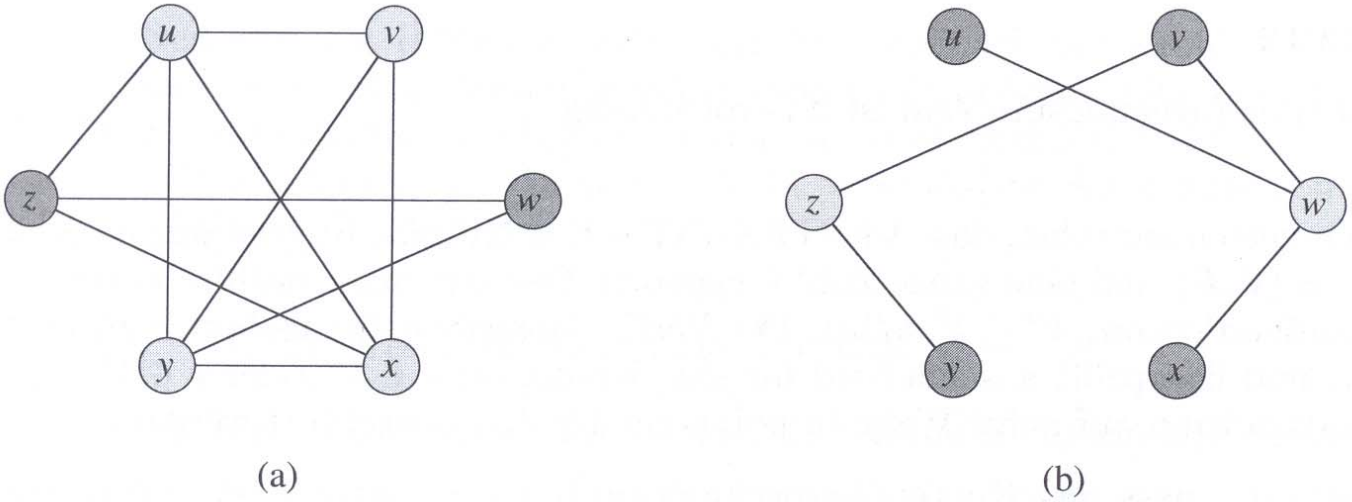
  Zertifikat y: Knotenmenge V‘ (die Knotenüberdeckung)

  - |y| = O(|V|)
  - Es kann in O(|V|+|E|) Zeit geprüft werden, ob |V‘| = k und , eine Knotenüberdeckung ist.

  Teil 2: CLIQUE $\leq_P$ VERTEX-COVER

  Sei $\dot{G}$ = (V,$\dot{E}$) mit $\dot{E}$ = { (u,v) $\in$ V x V | (u,v) $\notin$ E }. $\dot{G}$ ist der Komplement-Graph zu G.

VERTEX-COVER ist NP-schwer



- Sei Γ die Menge der ungerichteten Graphen
- Reduktionsfunktion r(x): Γ x IN → Γ x IN

  Sei <G,k> eine Eingabe des CLIQUE-Problems, dann definieren wir r(<G,k>) = <Ġ, |V| - k>.
- Teil 2.1: r(x) ist in Zeit $O(|V|^2)$ berechenbar:
  - konvertiere die Adjazenzlisten in eine Adjazenzmatrix und
  - invertiere alle nicht-diagonalen Matrixelemente

VERTEX-COVER ist NP-schwer

- Teil 2.2: <G,k> $\in$ CLIQUE => <Ġ, |V| - k> $\in$ VERTEX-COVER

  Sei V‘ eine k-Clique. Wir betrachten die Knotenmenge V‘‘ = V – V‘ in Ġ:

  - |V‘‘| = |V| - k
  - Sei e = (v,w) $\in$ Ė

    => (v,w) $\notin$ E

    => entweder v $\notin$ V‘ oder w $\notin$ V‘, da V‘ eine Clique ist

    => e wird durch V‘‘ in Ġ überdeckt.

- Teil 2.3: <Ġ, |V| - k> $\in$ VERTEX-COVER => <G,k> $\in$ CLIQUE

  Sei V‘ eine Knotenüberdeckung der Größe |V| - k in Ġ. Wir betrachten V’’ = V – V’ in G:

  - |V‘‘| = k
  - Sei e = (v,w) $\in$ V‘‘ x V‘‘

    => e $\notin$ Ė, da sonst V’ keine gültige Knotenüberdeckung wäre

    => e $\in$ E => V‘‘ ist eine Clique.

Algorithmen für Vertex-Cover

```
APPROX-VERTEX-COVER( G )
C = ∅
E‘ = G.E
while E‘ ≠ ∅
  (u,v) = arbitrary e ∈ E‘                          (**)
  C = C ∪ {u, v}
  E‘ = E‘ \ ( {(u,w)|w ∈ u.Adj} ∪ {(w,v)|w ∈ v.Adj)
  return C
```

- Greedy-Strategie: nehme solange Knoten hinzu bis alle Kanten überdeckt sind.
- Offensichtlich ist C am Ende von solve_vertex_cover ein Vertex-Cover.
- Laufzeit: O(|E|) bei geschickter Implementierung der Mengenoperationen

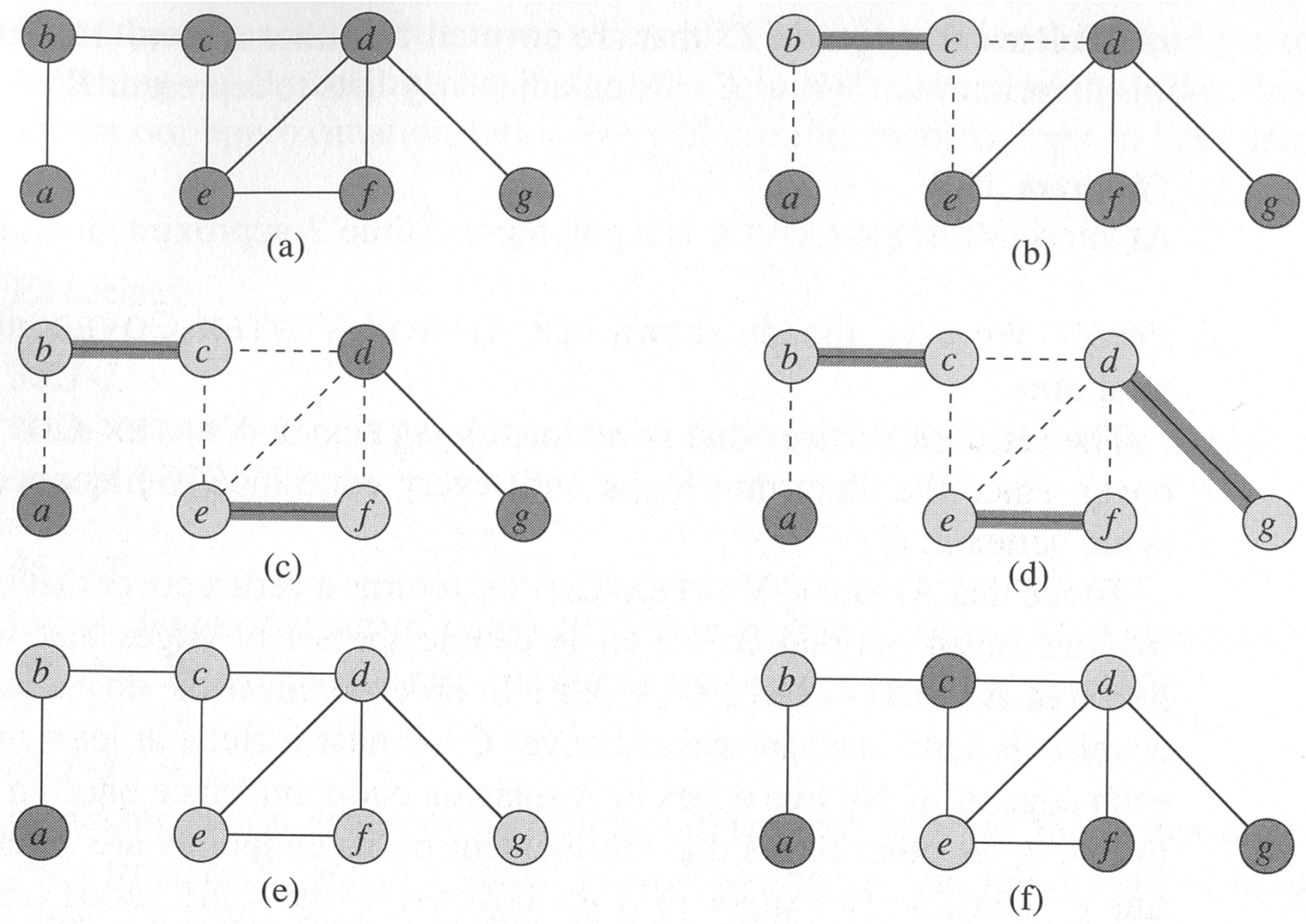
Algorithmen für Vertex-Cover

```
APPROX-VERTEX-COVER-2( G )
C = ∅
V‘ = G.V
E‘ = G.E
while E‘ ≠ ∅
  v = select v ∈ V‘ with maximal node degree in
  G=(V‘,E‘)
  C = C ∪ {v}
  E‘ = E‘ \ {(v,w)|w ∈ v.Adj}
return C
```

- Greedy-Strategie: nehme solange Knoten hinzu bis alle Kanten überdeckt sind.
- Offensichtlich ist C am Ende von solve_vertex_cover ein Vertex-Cover.
- Laufzeit: O(|E|) bei geschickter Implementierung der Mengenoperationen

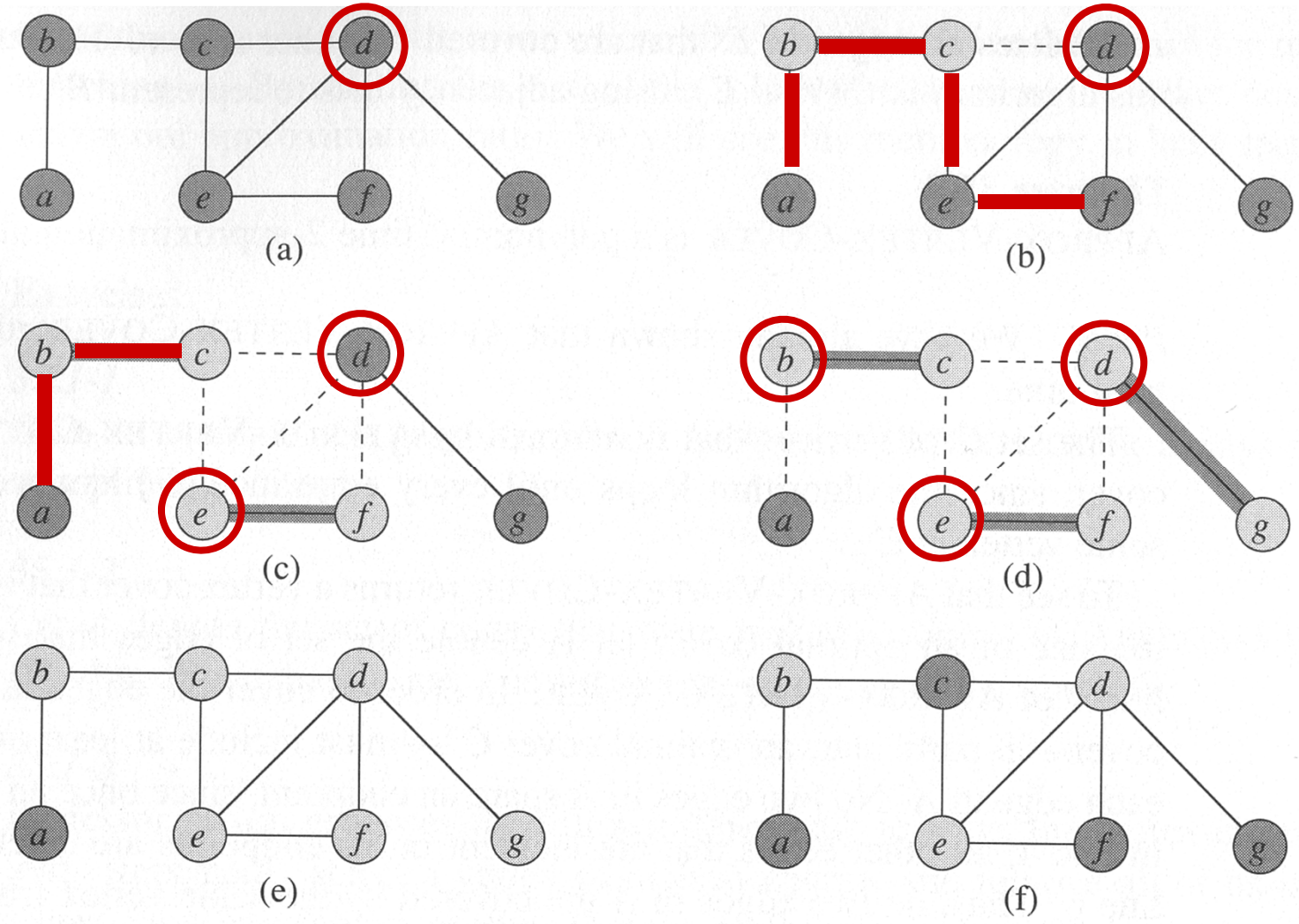
Vertex-Cover (berechnet mit APPROX-VERTEX-COVER())

- Bsp:



Vertex-Cover (berechnet mit APPROX-VERTEX-COVER2())

- Bsp:



Vertex-Cover

- **Theorem 35.1**:

  APPROX-VERTEX-COVER ist eine 2-Approximationsalgorithmus mit polynomieller Laufzeit.

  - Beweis Teil 1: APPROX-VERTEX-COVER hat polynomielle Laufzeit
    - siehe vorherige Seite.
  - Beweis Teil 2: $|C| / |C^*| \leq 2$

    C*: minimaler Vertex-Cover,

    C: von APPROX-VERTEX-COVER berechnete Knotenüberdeckung

    Sei A die Menge der Kanten, die in Zeile (**) ausgewählt werden.

    - Kanten in A haben keinen gemeinsamen Knoten

      => zur Überdeckung der Kanten in A benötigt man |A| Knoten

      da $A \subseteq E$, folgt $|C^*| \geq |A|$
    - in jedem Schleifendurchlauf wird A um eine Kante, C um zwei Knoten erweitert, es folgt $|C| = 2\,|A|$

    Insgesamt gilt $|C| = 2\,|A| \leq 2\,|C^*|$.

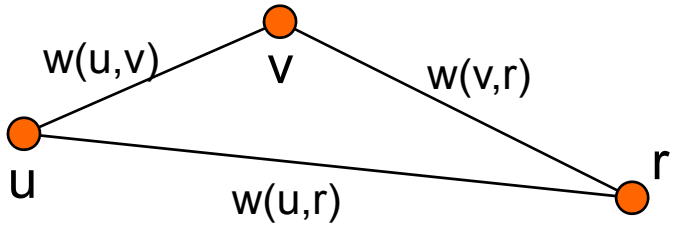
Vertex-Cover

- Kommentare:
  - Wie funktionieren die Beweise?
    - optimale Lösung ist unbekannt, finde zunächst eine untere Schranke (falls minimiert wird)
    - setze die untere Schranke in Beziehung zu der errechneten Lösung.
  - Gibt es Verbesserungsmöglichkeiten von APPROX-VERTEX-COVER?
    - APPROX-VERTEX-COVER hat einen nicht aufgelösten Freiheitsgrad:

      Wahl der Kante, die zu A hinzugenommen wird
    - mögliche Heuristiken zur Verbesserung
      - wähle Kante, dessen inzidente Knoten einen hohen Grad haben
  - Nicht jeder Algorithmus ist ein guter Approximationsalgorithmus
    - APPROX-VERTEX-COVER-2 ist **kein** 2-Approximationsalgorithmus!

Ein Approximations-Algorithmus für Triangle-TSP

- Problem (Triangle-TSP):
  - geg. vollständiger Graph G=(V,E), mit Kantengewichtsfunktion w, w erfüllt die Dreiecksungleichung
  - ges. kürzeste Rundtour durch alle Knoten $v \in V$
- Dreiecksungleichung:
  - für alle Knoten u,v,r gilt: $w(u,v) + w(v,r) \geq w(u,r)$



- Triangle-TSP ist wie TSP NP-schwer.

Ein Algorithmus für Triangle-TSP

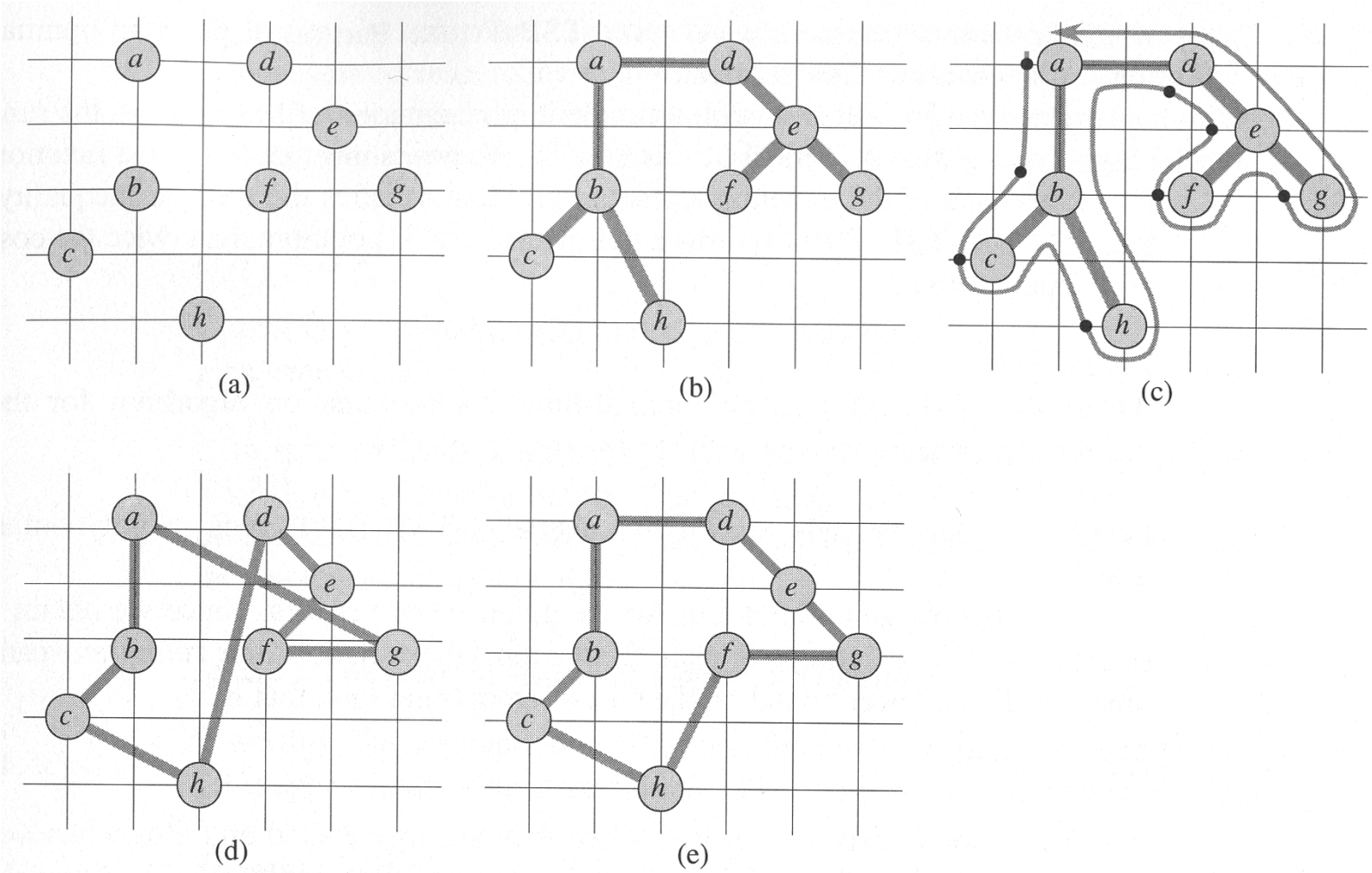
- Schritt 1: berechne minimalen Spannbaum T
- Schritt 2: durchlaufe den Baum und gib alle Knoten beim ersten Besuch aus (sei v.TAdj die Adjazenzliste des MST):
- APPROX-TSP-TOUR(G, c)

```
T = MINIMUM-SPANNING-TREE((V,E), c)
r = arbitrary v∈V
H = PREORDER-TRAVERSAL(T, r, NIL )
return H


PREORDER-TRAVERSAL(T, v , parent)
  output v
  for each w ∈ v.TAdj
    if( w ≠ parent ) PREORDER-TRAVERSAL(T, w, v)
```

Triangle-TSP

- Bsp:



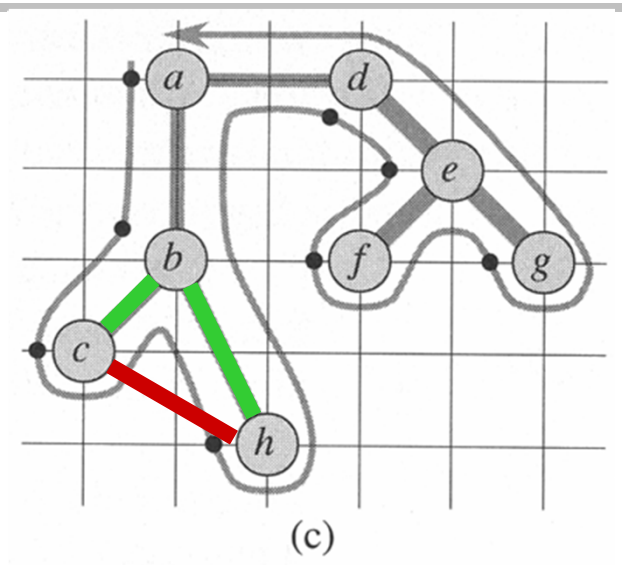
Triangle-TSP

- **Theorem 35.2**

  APPROX-TSP-TOUR ist eine Polynomzeit 2-Approximationsalgorithmus für Triangle-TSP.

  - Beweis Teil 1: APPROX-TSP-TOUR hat polynomielle Laufzeit
    - Minimaler Spannbaum berechnen: $O(N^2 \log N)$
    - preorder-Durchlauf durch T: $O(N)$
    - Gesamtlaufzeit: $O(N^2 \log N)$
  - Beweis Teil 2: $w(H)/w(H^*) \leq 2$
    - H: Lösung von APPROX-TSP-TOUR, H^*: optimale Lösung
    - Das Gewicht eines minimalen Spannbaums T* stellt eine untere Schranke für die Länge einer TSP-Tour H* dar:
      - Durch Löschen einer Kante e aus einer TSP-Tour H entsteht ein Spannbaum T mit w(H) = w(T) + w(e)
      - Für die optimale TSP-Tour H* gilt somit: $w(H^*) \geq w(T^*)$

Triangle-TSP



(c)

C: a–b–c–b–h–b–a–d–e–f–e–g–e–d–a

C‘: a–b–c——h– …

w(C‘) = w(C) - [w(c,b) + w(b,h)] + w(c,h)

≤ w(C)

- Beweis Teil 2 (Fortsetzung.):
  - Sei C der Zyklus, der sich durch einmaliges Umlaufen von T ergibt.
  - C enthält jede Kante e aus T zwei mal, d.h. w(C) = 2 w(T)
  - Löschen eines Knotens w aus Zyklus C verkürzt die Länge des neuen Zyklus C‘ aufgrund der Dreiecksungleichung:
  - H entsteht aus C durch Löschen aller doppelt auftretenden Knoten, also gilt

    $w(H) \leq w(C) = 2\, w(T) \leq 2\, w(H^*)$

Das allgemeine TSP-Problem

- **Theorem 35.3** (Approximierbarkeit von TSP):
  - Falls P $\neq$ NP, gilt für jedes ρ>1, dass für TSP kein Polynomzeit Approximations-Algorithmus mit Approximationsfaktor ρ existiert.
  - Beweis:
    - Idee: nutze ρ-Approx.-Alg. A zur Lösung eines NP-vollständigen Problems (HAM-CYCLE). O.B.d.A. gilt: ρ ist ganzzahlig.
    - Polynomzeitreduktion:

      Sei G=(V,E) Eingabe zu HAM-CYCLE. Konstruiere Eingabe für TSP:

      G‘=(V,E‘), E‘= {(u,v) : u,v $\in$ V },

      $$c(u,v) = \begin{cases} 1 & (u,v) \in E \\ \rho\,|V|+1 & sonst \end{cases}$$

    - G $\in$ HAM-CYCLE => G‘ hat ein Tour der Länge |V|
    - G $\notin$ HAM-CYCLE => alle Touren in G‘ haben Länge > ρ |V|

      => A löst das HAM-CYCLE Problem in polynomieller Zeit.

Kommentare zu Approximationsalgorithmen

- Approximationsalgorithmen ermöglichen die Lösung NP-schwerer Optimierungsprobleme mit einer **Gütegarantie** in polynomieller Zeit.
- Zum Nachweis des **Approximationsfaktors** ist es notwendig, eine Schranke für die optimale Lösung zu finden und diese mit der berechneten Lösung in Beziehung zu setzen.
- Es gibt Probleme, die nachweislich nicht mit einem konstanten Faktor in polynomieller Zeit approximierbar sind (unter der Annahme, dass $P \neq NP$)
- **Nicht-Approximierbarkeit** kann ebenfalls durch Reduktion nachgewiesen werden (NP-Vollständigkeit des k-Threshold-Problems mit Approximationsfaktor)
- Zusätzliche **Randbedingungen** (wie z.B. die Dreiecksungleichung) können das Problem deutlich vereinfachen, so dass sie in Polynomzeit approximierbar oder sogar lösbar werden.

8.2 Exakte Verfahren

- Vorgehen zur exakten Lösung
  - Bei vielen Problemen setzt sich die Lösung aus einer Menge von kleinen Teilentscheidungen zusammen.
  - Bsp:
    - CLIQUE: Gehört ein Knoten zu einer k-CLIQUE oder nicht?
    - TSP: Welchen Knoten besucht man nach Knoten v?
    - SUBSET-SUM: Gehört eine Zahl in die ausgewählte Teilmenge oder nicht?
  - Bei kombinatorischen Problemen ist es in der Regel einfach, die Menge der möglichen Lösungen aufzuzählen.
    - Selektionsprobleme: Bestimmung einer Teilmenge von Objekten
    - Zuordnungs-/Reihenfolgeprobleme: Bestimmung einer Permutation von Objekten
  - Lösungsansatz
    - Einfach: Enumeriere alle möglichen Lösungen und bestimme das Optimum.
    - Besser: Enumeriere alle möglichen Lösungen, die noch besser sein können, als das bisher gefundene Optimum.

Enumeratoren

- Enumeration von Selektionen
  - Auswahl einer Teilmenge T über eine endliche Grundmenge G kann durch einen Binärstring der Länge |G| beschrieben werden
    - b_i = 0 : i-tes Element der Grundmenge ist nicht in T enthalten
    - b_i = 1 : i-tes Element der Grundmenge ist in T enthalten
  - Mögliche Implementierung (Cormen Kap. 17.1)

    Sei B[0..length[B]-1] ein Bit-Array mit length[B] Elementen

```
BINARY-INCREMENT( B )
  i = 0
  while i < B.length and B[i] == 1 do
    B[i] = 0; i = i+1
  if( i < length[B] ) B[i] = 1
```

  - BINARY-INCREMENT berechnet die nächste Auswahl in einer n- elementigen Menge (aufsteigender Binärwert)

Enumeratoren

- Enumeration von Permutationen
  - Zuordnungen und Reihenfolgen können über Permutationen beschrieben werden
  - Beispiel:
    - TSP: Lösungsraum ist Permutation der Knoten des Graphen
    - QUADR. ASSIGNMENT: Lösungsraum ist die Permutation p der Objekte (p[i] = j bedeutet: Objekt i wird in Slot j platziert)

```
ALL-PERMUTATIONS( P, i )
  // Sei P[1..P.length] ein Array mit Zahlen 1,2,...,P.length
  if( i == 1 ) DO-SOMETHING(P)
  else
    ALL-PERMUTATIONS(P, i-1)
    for j = 1 to i-1
      SWAP( P[i], P[j] )
      ALL-PERMUTATIONS(P, i-1)
      SWAP( P[i], P[j] )
```

Enumeratoren

- Iterative Implementierung
  - Berechne jeweils die nächst größere Permutation in lexikographischer Reihenfolge

    Sei P[1..P.length] ein Array mit Zahlen 1,2,...,P.length

```
PERM-INCREMENT( P )
 t = 1
 while( t < P.length and P[t+1] > P[t] ) t = t+1
 if( t < P.length )
   s = t
   while( s > 1 and P[t+1] < P[s-1] ) s = s-1
   SWAP( P[t+1], P[s] )
 for( i = 1 to ⌊t/2⌋ ) SWAP( P[i], P[t-i+1] )
```

Enumeratoren

- Bsp: BINARY-INCREMENT

```
   i: 9 8 7 6 5 4 3 2 1 0
   B: 0 1 1 0 0 1 1 1 1 1
BINARY-INCREMENT:
   B: 0 1 1 0 1 0 0 0 0 0
BINARY-INCREMENT:
   B: 0 1 1 0 1 0 0 0 0 1
BINARY-INCREMENT:
   B: 0 1 1 0 1 0 0 0 1 0
```

suche die erste Stelle mit einer 0, kehre alle Werte bis zu dieser Stelle um.

- Bsp: PERM-INCREMENT

```
   i: 9 8 7 6 5 4 3 2 1
   P: 6 3 2 7 9 8 5 4 1
              t   s
PERM-INCREMENT:
   P: 6 3 2 8 1 4 5 7 9
PERM-INCREMENT:
   P: 6 3 2 8 1 4 5 9 7
PERM-INCREMENT:
   P: 6 3 2 8 1 4 7 5 9
```

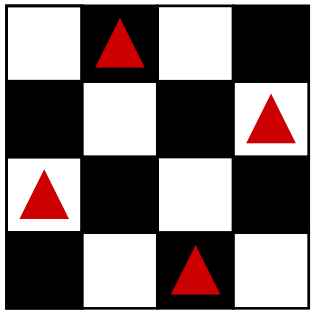
t: letzte Position der aufsteigenden Folge

s: kleinster Wert der aufsteigenden Folge, der größer P[t+1] ist.

Das n-Damen-Problem

- n-Damen-Problem:

  Platziere n Damen auf einem n x n Schachbrett, so dass sie sich gegenseitig nicht schlagen können (d.h. unterschiedliche Spalten, Zeilen und Diagonalen)



Lösung durch Enumeration:

- Variante 1: Enumeration über Schachbrettpositionen

```
N-QUEENS-1( n )
    for( i = 0 to n²-1 ) B[i] = 0
    do
      if( CHECK-N-QUEENS(B) ) output B
      BINARY-INCREMENT(B)
    while( B ≠ [0, 0, …., 0] )
```

n=8: $2^{64} \approx 18.446.744.000.000.000$ Positionen

Das n-Damen-Problem

- Variante 2: Enumeration über Damenpositionen

  Nutze die Eigenschaft, dass genau 8 Damen platziert werden müssen.

```
N-QUEENS-2( P, n, d )
// platziert die d-te Dame auf das Schachbrett Position 0, .., n^2 -1
// P[i] speichert die Position der i-ten Dame, i = 1, …, n
  for( i = 1 to n^2 -1 )
    P[d] = i
    if( d < n ) N-QUEENS-2(P, n, d+1)
    else
      if( CHECK-N-QUEENS(P) ) output P
```

n=8: $64^8 = 2^{48} \approx 281.474.497.000.000$ Positionen

Das n-Damen-Problem

- Variante 3:
  Ordne die Damen, so dass P[i] < P[j] für alle i < j gilt.

```
N-QUEENS-3( P, n, d )
// platziert die d-te Dame auf das Schachbrett Position 0, .., n² -1
// P[i] speichert die Position der i-ten Dame, i = 1, …, n; P[0] = -1
  for( i = P[d-1]+1 to n² -1 )
    P[d] = i
    if( d < n ) N-QUEENS-3(P, n, d+1)
    else
      if( CHECK-N-QUEENS(P) ) output P
```

n=8: $\binom{64}{8}$ = 64*63*...*57 / (8!) = 4.426.165.368

Das n-Damen-Problem

- Variante 4:

  Nutze die Eigenschaft, dass zwei Damen nicht in der gleichen Zeile stehen können

  Sei P[i] nun die Spalte, in der die Dame der i-ten Zeile steht

```
N-QUEENS-4( P, n, d )
// platziert die d-te Dame auf das Schachbrett in Zeile d, Spalte P[d]
for( i = 1 to n )
  P[d] = i
  if( d < n ) N-QUEENS-4(P, n, d+1)
  else
    if( CHECK-N-QUEENS(P) ) output P
```

n=8: $8^8 = 2^{24} = 16.777.216$

Das n-Damen-Problem

- Variante 5:
  Nutze die Eigenschaft, dass zwei Damen weder in der gleichen Zeile, noch in der gleichen Spalte stehen können.

  Da in jeder Spalte genau eine Dame stehen muss, können wir die Platzierung als Permutation der Spalten 1,…,8 interpretieren.

  Sei P[i] nun die Spalte, in der die Dame der i-ten Zeile steht

```
N-QUEENS-5( P, n, d )
// platziert die d-te Dame auf das Schachbrett in Zeile d, Spalte P[d]
  for( i = 1 to n ) P[i] = i
  do
    if( CHECK-N-QUEENS(P) ) output P
    PERM-INCREMENT(P)
  while( P ≠ [1, 2, ..., n] )
```

  n=8: 8! = 40.320

Backtracking

- Schwäche aller bisherigen Lösungsversuche:

  CHECK-N-QUEENS() wird erst aufgerufen, nachdem ALLE Damen platziert wurden.
- Testen und Wiederverwenden von Teillösungen:
  - Zur Auswertung von Lösung werden Rechenschritte über gemeinsame Teillösungen wiederholt ausgeführt.
    - Bsp.: Auswertung beim n-Damen-Problem

      CHECK-N-QUEENS() überprüft für alle Paare von Damen, ob sie sich schlagen können.

      Seien P und P‘ zwei Platzierungen von Damen, mit P[i] = P‘[i] für alle i < d:

      Falls für i<j<d gilt: Dame P[i] schlägt Dame P[j] nicht, so gilt dies für P und für P‘. Der Test muss nur ein mal ausgeführt werden.

      Falls für i<j<d gilt: Dame P[i] schlägt Dame P[j]: so kann weder P noch P‘ zu einer gültigen Lösung erweitert werden.
  - Wiederholte Auswertung von Teillösungen kann vermieden werden, wenn die Lösungen sukzessive aufgebaut werden.

Backtracking

- Betrachte Lösungsraum als ein Baum (Suchbaum):
  - Wurzel: leere Teillösung
  - innere Knoten: Teillösungen
  - Blätter: Lösungen
  - Auf jeder Ebene wird eine Teilentscheidung gefällt und damit die Teillösung des Parent-Knotens erweitert.
  - Die Child-Knoten repräsentieren die alternativen Möglichkeiten respektive der Teilentscheidung
- Backtracking-Algorithmus:
  - systematische Tiefensuche in einer (den Lösungsraum repräsentierenden) Baumstruktur nach der optimalen Lösung
  - Teillösungen, die sich nicht zu vollständigen Lösungen erweitern lassen, müssen nicht weiter betrachtet werden.
  - Die Baumstruktur existiert nur implizit.

Ein Backtracking-Algorithmus für das n-Damen-Problem

- Idee:
  - Platziere die Damen nacheinander auf Positionen, (i, P[i])
  - Vor Platzierung der d-ten Dame:
    - Prüfe, ob die Platzierung (d, P[d]) von den anderen Damen blockiert wird (d.h. eine Dame an (d, P[d]) geschlagen werden kann
    - Falls ja: Verwerfe Platzierung (d, P[d])
    - Falls nein: Versuche rekursiv, die Dame d+1 zu platzieren
  - Nach Platzierung der n-ten Dame:
    - wissen wir, dass die Lösung korrekt ist!

```
CHECK-LAST-QUEEN( P, d )
    // prüft, ob (d, P[d]) von einer Dame i < d blockiert wird
    for i ← 1 to d-1 do
      if( P[i] == P[d] or                  // gleiche Spalte
          P[i] - P[d] == i - d or          // Diagonale nach oben links
          P[i] - P[d] == d - i )           // Diagonale nach oben rechts
          return FALSE
    return TRUE
```

Ein Backtracking-Algorithmus für das n-Damen-Problem

```
N-QUEENS-6( n )
    for( i = 1 to n ) P[i] = 0
    N-QUEENS-BACKTRACK( P, n, 1 )

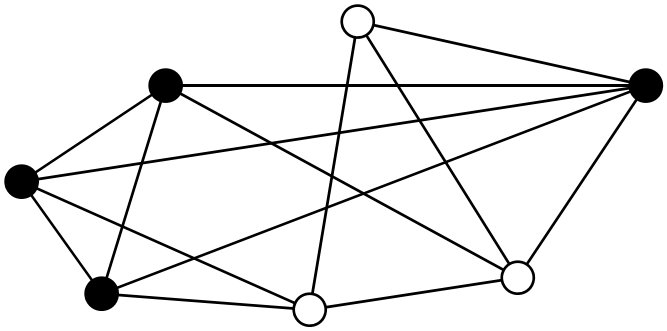
N-QUEENS-BACKTRACK(P, n , d)
    // Damen 1, …, d-1 sind platziert auf Positionen (i, P[i]), so dass sie
    // sich paarweise nicht schlagen können
    if( d == n+1 ) output P
    else
      for( i = 1 to n )          (*)
        P[d] = i
        if( CHECK-LAST-QUEEN( P, d ) )
          N-QUEENS-BACKTRACK( P, n, d+1 )
```

n=8: Backtracking-Baum hat 2057 Knoten, erste Lösung wird nach 114 rekursiven Aufrufen gefunden.

for-Schleife (*) und der CHECK-LAST-QUEEN()-Test lassen sich durch Implementierung einer Datenstruktur zur Speicherung nicht blockierter Spalten und Diagonalen noch vermeiden.

Ein Backtracking-Algorithmus für das OPT-CLIQUE-Problem

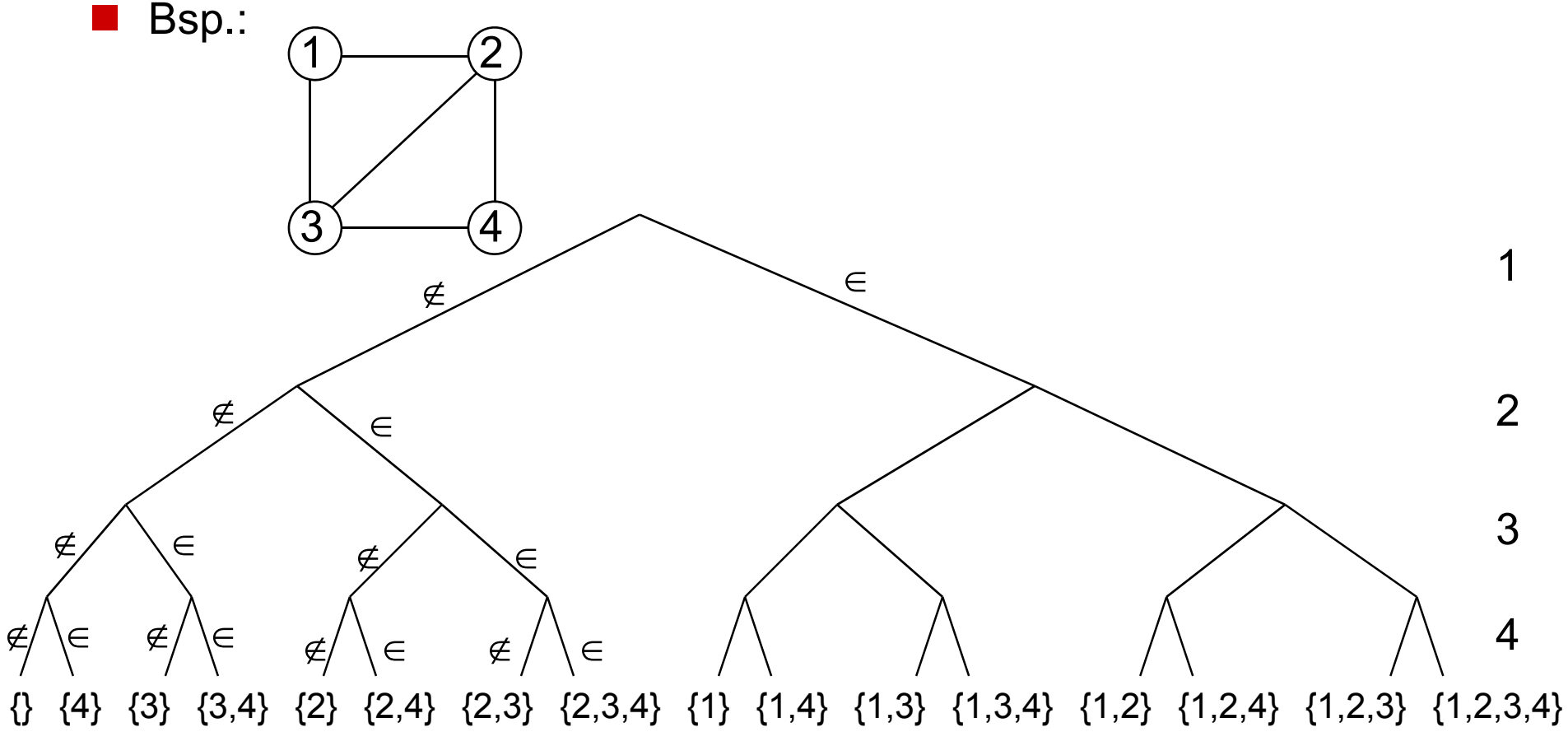
- OPT-CLIQUE-Problem:
  - Gegeben: ein ungerichteter Graph G = (V, E)
  - Gesucht: maximaler vollständiger Teilgraph (Clique) von G



- Idee
  - Baue die Knotenmenge der Clique sukzessive auf
  - für eine Teilmenge C $\subseteq$ V: prüfe, ob alle Knoten paarweise miteinander verbunden sind
  - falls für zwei Teilmengen C und C‘ gilt C $\cap$ C‘ = I $\neq\emptyset$:
    - Ist I bereits eine Clique, müssen die Knoten nicht mehr überprüft werden
    - Ist I bereits keine Clique, so sind C und C‘ ebenfalls keine Cliquen

Backtracking-Algorithmus für OPT-CLIQUE

- Lösungsraum von OPT-CLIQUE als Baumstruktur:
  - Blätter: alle möglichen Teilmengen I der Knotenmenge V
  - Teilentscheidung auf Ebene i des Baumes: $v_i \in I$ oder $v_i \notin I$?
- Bsp.:



Backtracking-Algorithmus für OPT-CLIQUE

```
CLIQUE-BACKTRACK(G, I, d, M )
    if( d == |V|+1 )      // Clique gefunden
      if( |I| > |M| ) M = I
    else
      v_d = select d-th node from G.V
        // Fall 1: Clique I wird um Knoten v_d erweitert
      is_clique = TRUE    // prüfe, ob I ∪ {v_d} eine Clique ist
      for( each u ∈ I )                    (**)
        if( u ∉ Adj[v_d] ) is_clique = FALSE
      if(is_clique) CLIQUE-BACKTRACK(G, I ∪ {v_d}, d+1, M)
      // Fall 2: Clique I wird nicht um Knoten v_d erweitert
      CLIQUE-BACKTRACK( G, I, d+1, M )
```

Aufruf: G: Eingabe-Graph,
I=∅ (bereits in der Teilmenge enthaltene Elemente),
d=1 (Tiefe im Baum)
M=∅ (maximale Clique)

Backtracking-Algorithmus für CLIQUE

- Laufzeit:
  - Sei N = |V|, dann hat der Suchbaum 2^{N+1} -1 Knoten
  - Ein rekursiver Aufruf von CLIQUE-BACKTRACK kann in O(N) Zeit bearbeitet werden

    ((**) wird durch Vergleich sortierter Listen realisiert)
  - Worst-Case Laufzeit: $O(N 2^N)$

    (z.B. falls G ein vollständiger Graph ist)
  - Tatsächliche Laufzeit hängt stark von der Kantendichte des Graphen ab
- Kommentare:
  - Der Algorithmus kann - leicht modifiziert - zum Aufzählen aller Cliquen, bzw. aller maximalen Cliquen verwendet werden
  - Der Algorithmus ist – je nach Fragestellung - bereits sehr effizient in der Praxis
  - besser: Bron-Kerbosch-Algorithmus (Chemieinformatik)

Kommentare zu Enumeratoren und Backtracking

- Je genauer die Randbedingungen an die Lösung für die Enumeration modelliert werden, umso weniger Varianten müssen enumeriert werden: N-QUEENS (für N=8, 0,1 μsec/CHECK-N-QUEENS-Aufruf)

Algorithmus	# Varianten	Zeit
1. Schachbrett	>18.446.744.000.000.000	58,5 Mio y
2. Damen-Positionen	>281.474.497.000.000	892 y
3. Reihenfolge-Unabhängigkeit	4.426.165.368	123 h
4. Zeilen-Eigenschaft	16.777.216	28 min
5. Spalten-Eigenschaft	40.320	4,0 sec
6. Teillösungen (Backtracking)	2.057	0,2 sec

- Randbedingungen, die in der Enumeration modelliert sind, müssen nicht explizit geprüft werden.
- Durch den hierarchischen Aufbau von Lösungen können Teillösungen bereits auf Gültigkeit geprüft werden ➔ Backtracking
- Enumeration eignet sich prinzipiell nur für diskrete (d.h. ganzzahlig modellierbare) Probleme

Branch & Bound

- Wie lässt sich die Laufzeit weiter verkürzen?
  - Reduktion der Anzahl der besuchten Knoten im Suchbaum
  - Pruning: Abschneiden von Teilbäumen, die nicht zu einer optimalen Lösung führen können
    1. alle Lösungen in diesem Teilbaum sind ungültig
    2. alle Lösungen in diesem Teilbaum sind garantiert schlechter als die optimale Lösung
- zu 1.: prüfe Gültigkeit der Lösungen
  - CLIQUE-BACKTRACK betrachtet keine Teilmengen, die keinen vollständigen Teilgraph beschreiben.
- zu 2.: berechne Schranken für die Güte der Lösungen
  - Annahme:
    - wir maximieren die Zielfunktion f, sei C* die optimale Lösung
  - untere Schranke L für C*:
    - eine gültige Lösung C stellt eine untere Schranke für C* dar: $f(C^*) \geq f(C) = L$

Branch & Bound

- obere Schranke U(v):
  - Sei v ein Knoten im Suchbaum
  - Alle Lösungen im Teilbaum unter v haben eine Teillösung I gemeinsam
  - Berechne, wie gut eine Lösung, die I enthält maximal werden kann
- Vorgehen Branch & Bound:
  - Suche eine gültige Lösung C, setze L ← f(C)
  - Führe ein Backtracking durch, für jeden besuchten Knoten v:
    - Falls v eine gültige Lösung C‘ mit f(C‘) > L repräsentiert, setze L ← f(C‘)
    - Berechne obere Schranke U(v)
    - Falls U(v) ≤ L (d.h. alle Lösungen im Teilbaum unter v sind garantiert schlechter als die bisher gefundene beste Lösung)
      - Rückkehr zum Parent-Knoten
    - Falls U(v) > L (d.h. es könnten Lösungen im Teilbaum unter v sein, die besser sind als die bisher gefundene beste Lösung)
      - Erweitere Teillösung, rekursiver Aufruf

Branch Bound

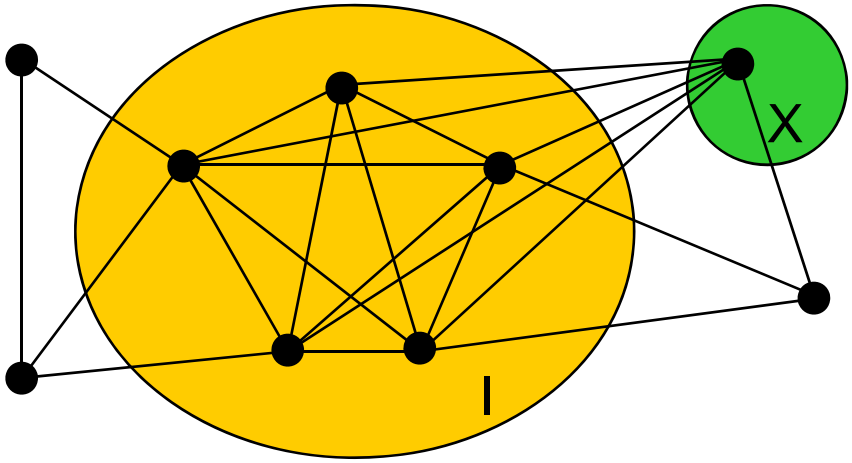
Branch & Bound-Algorithmus für CLIQUE

- untere Schranke L:
  - max. Anzahl der Knoten in allen bisher gefundenen Cliquen
  - Teilmengen sind bereits Cliquen und können zur Verbesserung von L herangezogen werden
- obere Schranke an Knoten v auf Ebene d mit Teillösung I U(d,I):
  - alle Cliquen unter v enthalten die Knoten aus I
  - die Knotenmenge aus I kann maximal um (n-d) Knoten erweitert werden.

    => U(d,I) = |I| + n-d
  - Verbesserung: Bestimme Knotenmenge X (‚extension set') von Knoten, die nicht in I sind, aber zu allen Knoten aus I adjazent sind.

    => U(I,X) = |I| + |X|

Branch & Bound-Algorithmus für CLIQUE



CLIQUE-BB(G, I, X, d, M)

- G: Eingabegraph
- I: Knotenmenge; aktuell betrachtete Teillösung
- X: Knotenmenge; Menge der Knoten, um die die aktuelle Teillösung erweitert werden kann (Extensionsmenge)
- d: Anzahl bereits betrachteter Knoten (Rekursionstiefe)
- M: Knotenmenge; maximale Clique, die gefunden wurde

Erster Aufruf: CLIQUE-BB(G, ∅, V, 0, M)

Branch & Bound-Algorithmus für CLIQUE

```
CLIQUE-BB(G, I, X, d, M )
    if( d == |G.V| +1 ) // Clique gefunden
      if( |I| > |M| ) M = I
    else
      v_d = select d-th node from G.V
        // obere Schranke U = |I| + |X|, untere Schranke ist |M|
      if( |I| + |X| < |M| ) return // BOUND!
        // Fall 1: Knoten v_d ∈ I                    BRANCH!
        // I ∪ {v_d} ist eine Clique, g.d.w. v_d ∈ X
      if( v_d ∈ X ) CLIQUE-BB(G, I∪{v_d}, X∩Adj[v_d], d+1, M)
      // Fall 2: Knoten v_d ∉ I
      CLIQUE-BB( G, I, X\{v_d}, d+1, M )
```

Branch & Bound für das Assignment-Problem

- Assignment-Problem:
  - geg.: Anzahl n von Agenten und Jobs, Kostenmatrix $[c_{ij}]_{1 \le i,j \le n}$ mit c_{ij} = Kosten der Ausführung von Job j durch Agent i, $c_{ij} > 0$
  - ges.: Zuordnung p: {1,…,n} → {1,…,n} der Agenten zu den Jobs, so dass:
    - jeder Job durch einen Agenten bearbeitet wird
    - minimalen Gesamtkosten C(p) entstehen

$$C(p) = \sum_{i=1}^{n} c_{ip(i)}$$

Beispiel:

		Jobs			
	$[c_{ij}]$	1	2	3	4
Agenten	a	11	12	18	40
	b	14	15	13	22
	c	11	17	19	23
	d	17	14	20	28

C( ) = 69

Branch & Bound für das Assignment-Problem

- Beschreibung des Lösungsraums:
  - Abbildung der Agenten auf die Jobs f: {1,…,n} → {1,…,n}
    f(i) = j : Agent i erfüllt Job j
  - Randbedingung: Abbildung f muss bijektiv sein

- Obere Schranke für die optimale Lösung f*:
  - Wähle f(i) = i oder f(i) = n+1-i

$$C(f^*) \leq C(f) = \sum_{i=1}^{n} c_{if(i)}$$

- Untere Schranke für die optimale Lösung:
  - Jeder Job muss bearbeitet werden:

$$l_{\text{jobs}} = \sum_{j=1}^{n} \min_{1 \leq i \leq n} \{c_{ij}\}$$

  - Jeder Agent muss einen Job bearbeiten:

$$l_{\text{agents}} = \sum_{i=1}^{n} \min_{1 \leq j \leq n} \{c_{ij}\}$$

Branch & Bound für das Assignment-Problem

Jobs

Agenten

$[c_{ij}]$	1	2	3	4
a	**11**	12	18	40
b	14	**15**	13	22
c	11	17	**19**	23
d	17	14	20	**28**

$$C(f) = \sum_{i=1}^{n} c_{if(i)} = 73$$

Agenten

$[c_{ij}]$	1	2	3	4
a	**11**	**12**	18	40
b	14	15	**13**	**22**
c	11	17	19	23
d	17	14	20	28

$$l_{\text{jobs}} = \sum_{j=1}^{n} \min_{1 \le i \le n} \{c_{ij}\} = 58$$

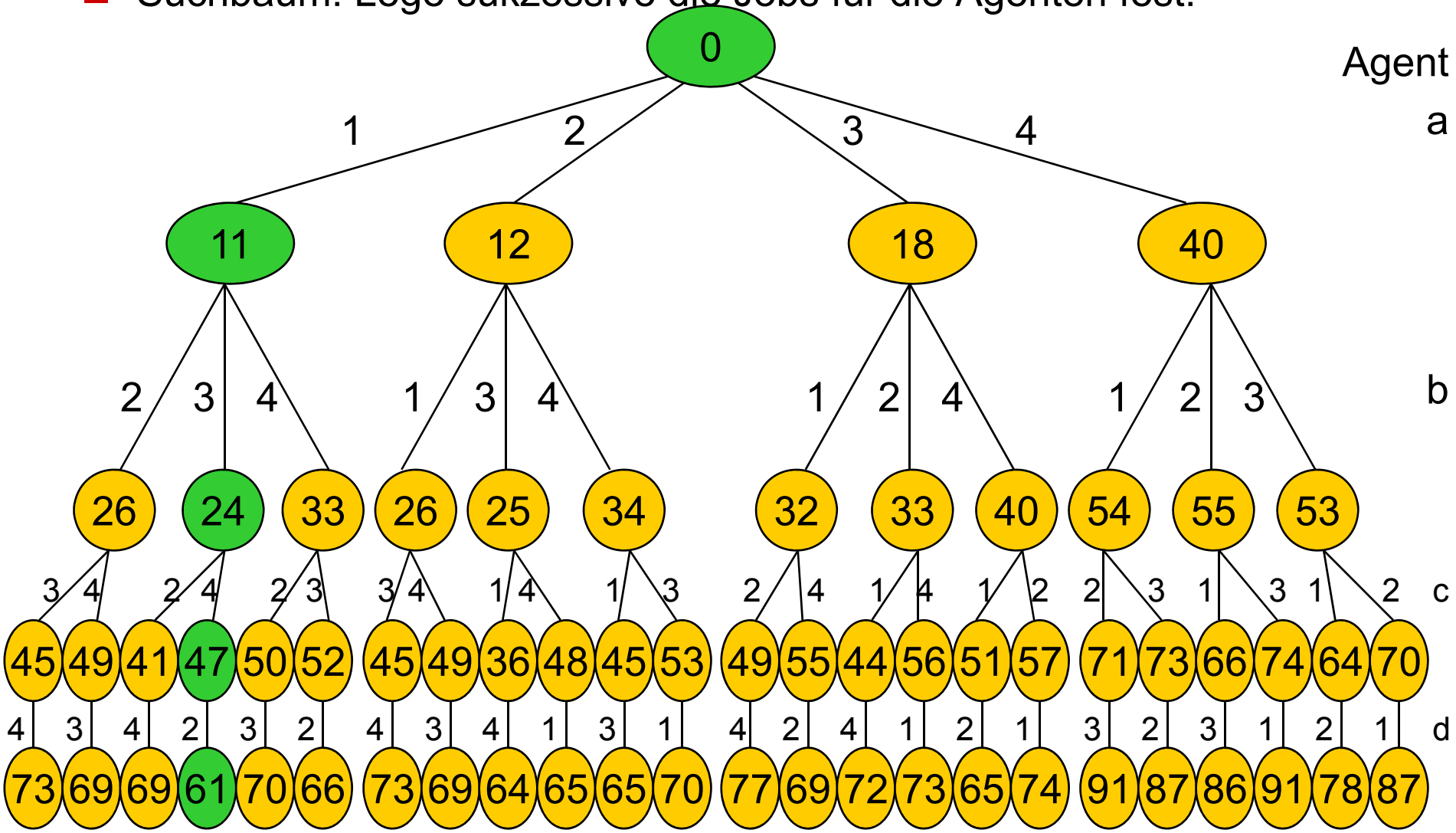
Agenten

$[c_{ij}]$	1	2	3	4
a	**11**	12	18	40
b	14	15	**13**	22
c	**11**	17	19	23
d	17	**14**	20	28

$$l_{\text{agents}} = \sum_{i=1}^{n} \min_{1 \le j \le n} \{c_{ij}\} = 49$$

Branch & Bound für das Assignment-Problem

- Suchbaum: Lege sukzessive die Jobs für die Agenten fest:



Branch & Bound für das Assignment-Problem

- Berechnung der oberen / unteren Schranken im Algorithmus:
  - für einen Knoten v auf Ebene k gilt: Sei f die Zuordnungsfunktion
    - die Zuordnung f der ersten k Agenten zu Jobs ist erfolgt

      => Kosten für die ersten k Agenten können bereits berechnet werden, für die Agenten k+1, …, n werden die Schranken verwendet:

      => obere Schranke: wähle eine beliebige Zuordnung für i > k

      Sei u:{1,…,n}→{1,…,n} eine bijektive Funktion mit u(i) = f(i) für i ≤ k, dann ist $U = \sum_{i=1}^{n} c_{iu(i)}$ eine obere Schranke für C*

      => untere Schranke: wähle minimale Kosten für Agenten i > k:

$$L = \sum_{i=1}^{k} c_{if(i)} + \sum_{i=k+1}^{n} \min_{j \in \{1,\ldots,n \mid f(h) \neq j \ \forall h \leq k\}} c_{ij}$$

$$= \sum_{i=1}^{k} c_{iu(i)} + \sum_{i=k+1}^{n} \min_{j \in \{u(k+1),\ldots,u(n)\}} c_{ij}$$

Branch & Bound für das Assignment-Problem

- ASSIGNMENT-UPPER-BOUND(c, f, n)

```
u = 0
for( i = 1 to n ) u = u + c[i,f[i]]
return u
```

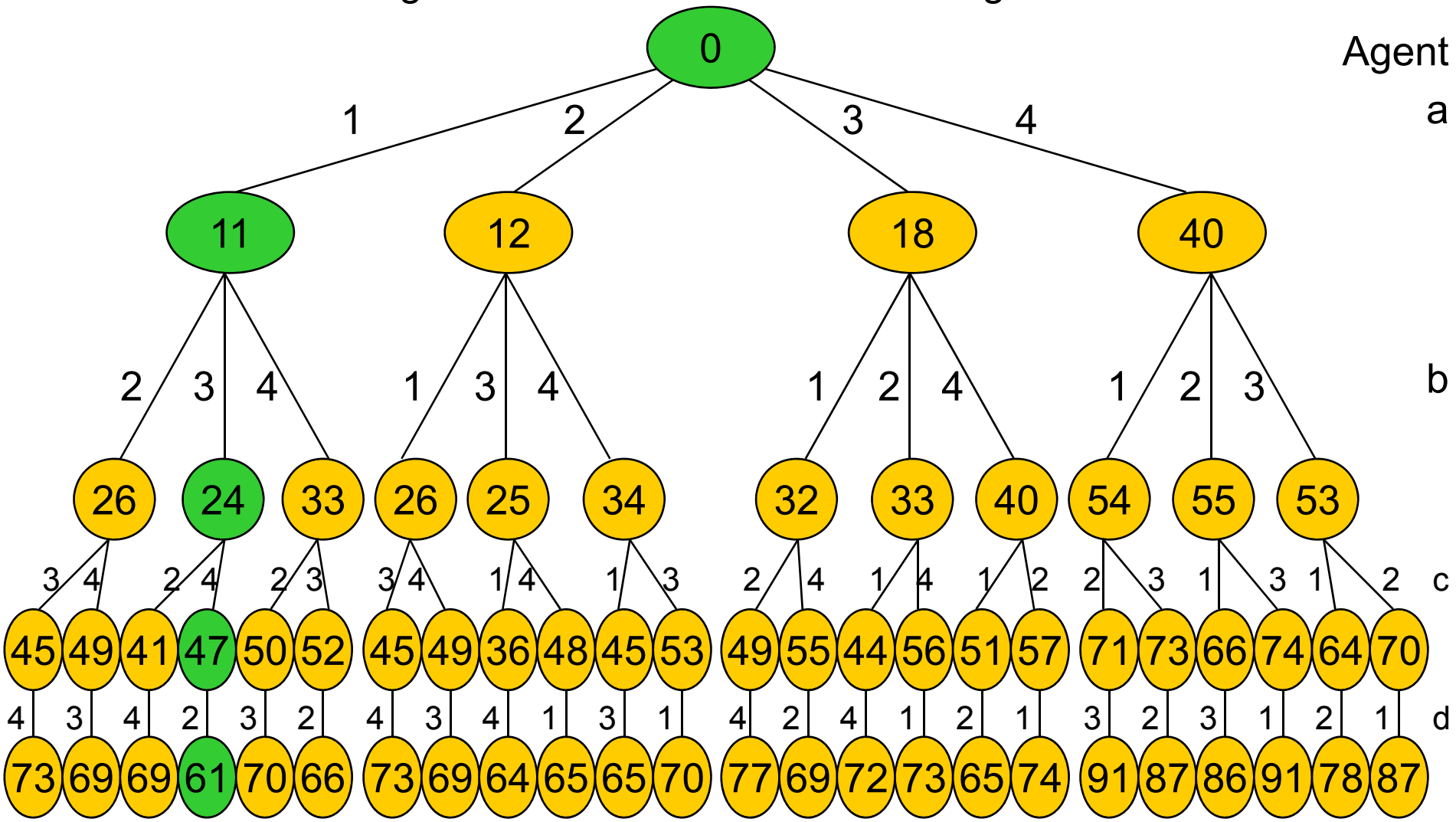
- ASSIGNMENT-LOWER-BOUND(c, f, k, n)

```
l = 0
for( i = 1 to n )
  if( i ≤ k ) l = l + c[i,f[i]]
  else
    m = c[i,f[k+1]]
    for( h = k+2 to n )
      if( c[i,f[h]] < m ) m = c[i,f[h]]
    l = l + m
return l
```

c: Kostenmatrix
n: Anzahl Agenten
f: aktuelle Zuordnung
k: Anzahl bereits fest zugeordneter Agenten

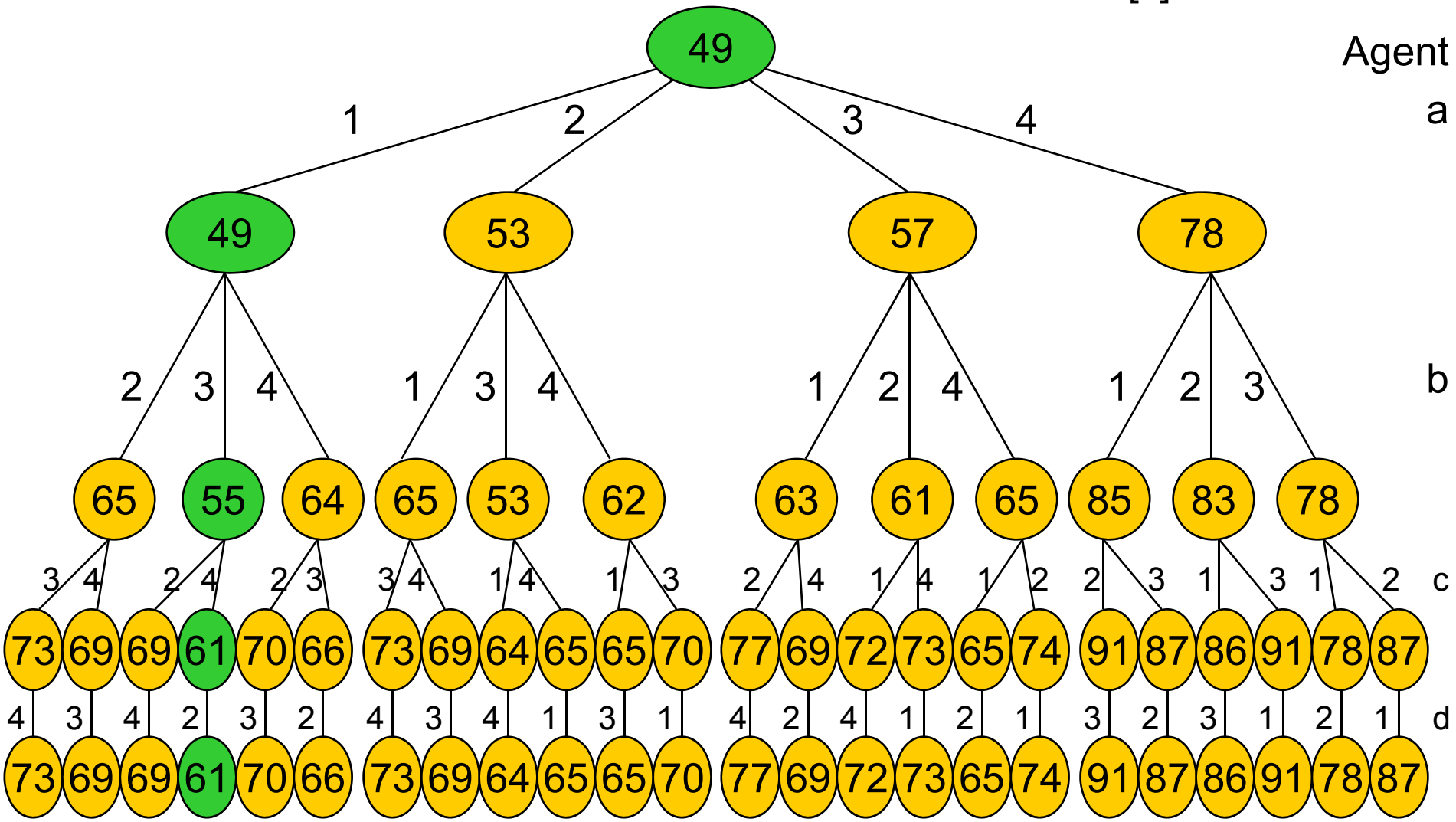
Branch & Bound für das Assignment-Problem

- Suchbaum: Lege sukzessive die Jobs für die Agenten fest:



Branch & Bound für das Assignment-Problem

- Suchbaum: Knoten v enthalten nun untere Schranke lb[v]:



Branch & Bound für das Assignment-Problem

- Allgemeines Schema: Sei U die globale obere Schranke für C*
  - wähle einen Knoten v des Suchbaums:

    v.level : Ebene des Suchbaums

    v.f : Zuordnungsfunktion des Knotens

    v.lb : Untere Schranke der Lösungen unter Knoten v
  - falls v.lb > U verwerfe die Lösung v.f
  - sonst teste alle möglichen Zuordnungen für Agent v.level+1:
    - berechne und aktualisiere neue obere Schranke U
    - erzeuge neue Knoten für die erweiterte Lösung
- Suchstrategien:
  - Depth-First (Tiefensuche): gehe rekursiv von v.level zu allen Nachfolgern auf Ebene v.level+1
  - Breadth-First (Breitensuche): gehe Ebene für Ebene vor
  - **Best-First** (Kombinierte Tiefen- und Breitensuche): wähle den Knoten mit v.lb minimal

Branch & Bound für das Assignment-Problem

- Umsetzung der Best-First-Strategie:
  - Verwende Prioritätswarteschlange Q zur Speicherung der zur Auswertung bereitstehenden Knoten (Suchfront)
    - Prioritätskriterium: Untere Schranke v.lb

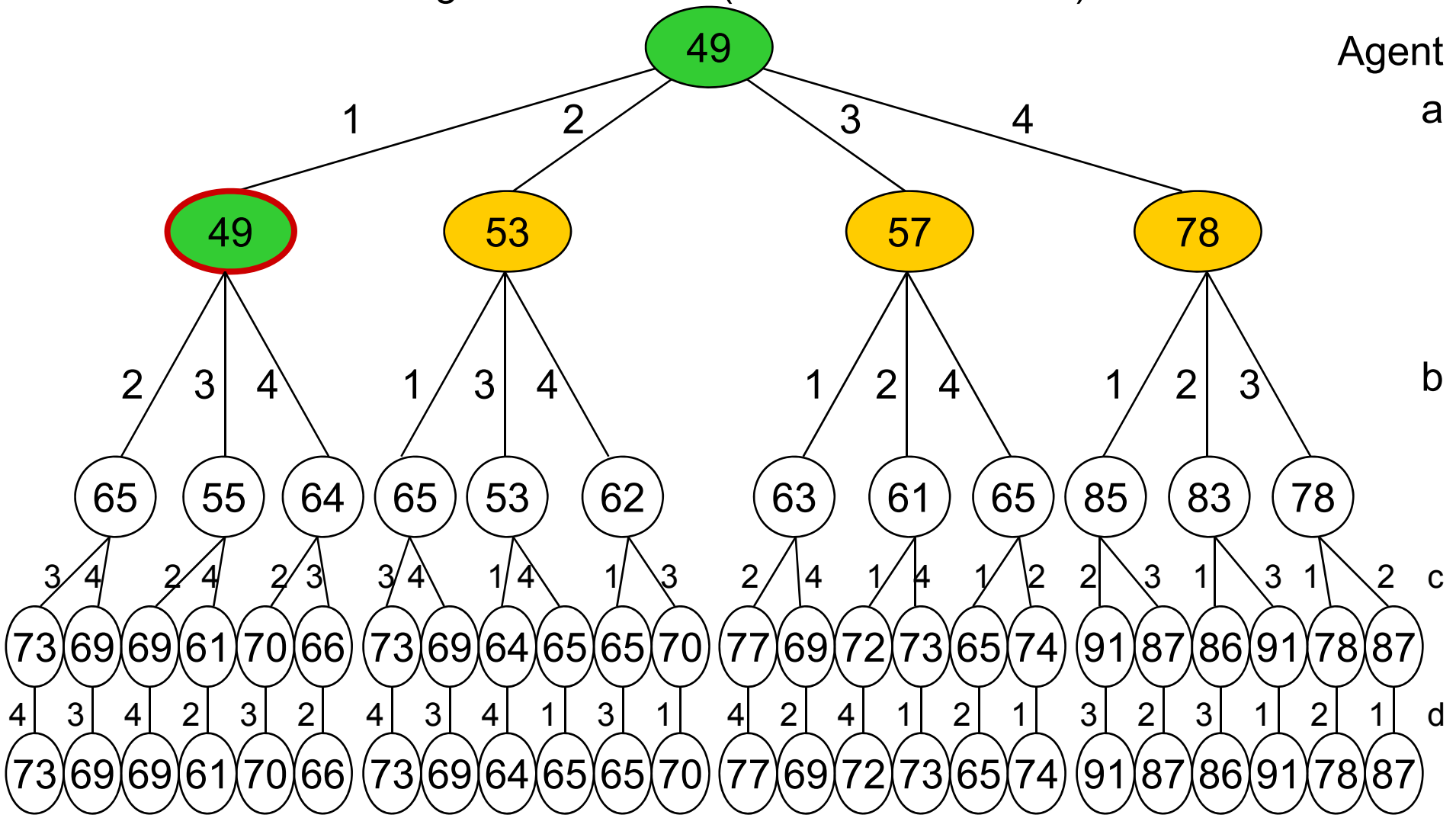
- ```
  ASSIGNMENT(c, n)
  Q = INIT-MIN-HEAP()
  // berechne Wurzel
  for( i = 1 to n ) v.f[i] = i
  v.level = 0
  v.lb = ASSIGNMENT-LOWER-BOUND(c, v.f, v.level, n)
  // berechne initiale obere Schranke U
  U = ASSIGNMENT-UPPER-BOUND(c, v.f, n)
  INSERT-MIN-HEAP(Q, v)
  // Fortsetzung …
  ```
  ```

- Fortsetzung von ASSIGNMENT()

```
while( Q ≠ ∅ )
  v = EXTRACT-MIN-HEAP(Q)
  if( v.level < n-1 and v.lb < U )
    for( i = v.level+1 to n )
       // erzeuge neue Lösung w: vertausche in f Positionen v.level und i
      w = v
      w.level = v.level+1
      SWAP( w.f[w.level], w.f[i] )
      lb[w] = ASSIGNMENT-LOWER-BOUND(c, w.f, w.level, n)
       // berechne neue obere Schranke für C*
      U = min( U, ASSIGNMENT-UPPER-BOUND(c, w.f, n) )
      if( w.lb ≤ U ) INSERT-MIN-HEAP(Q, w)
return U
```

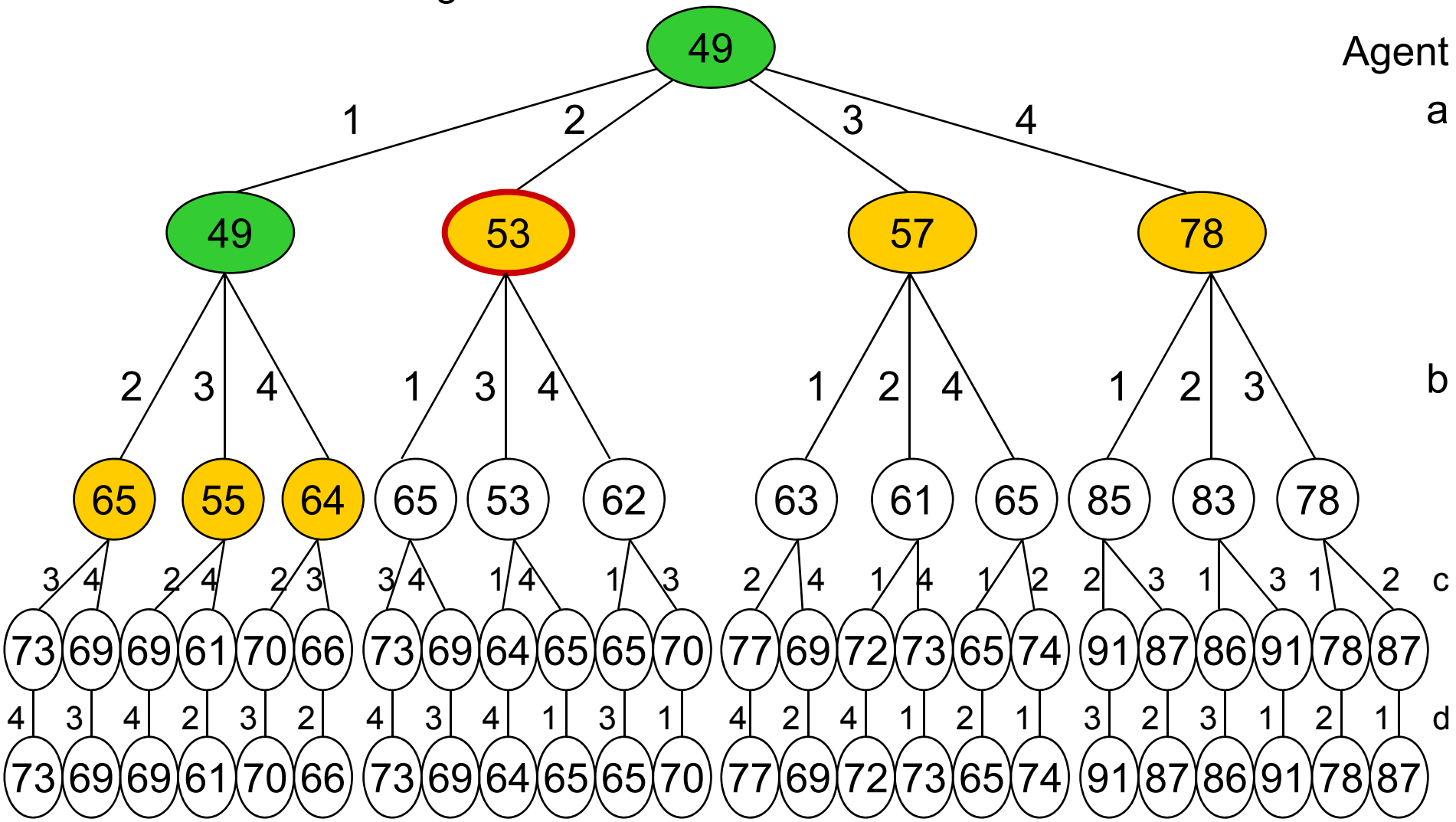
Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 1 (Wurzel des Baums)



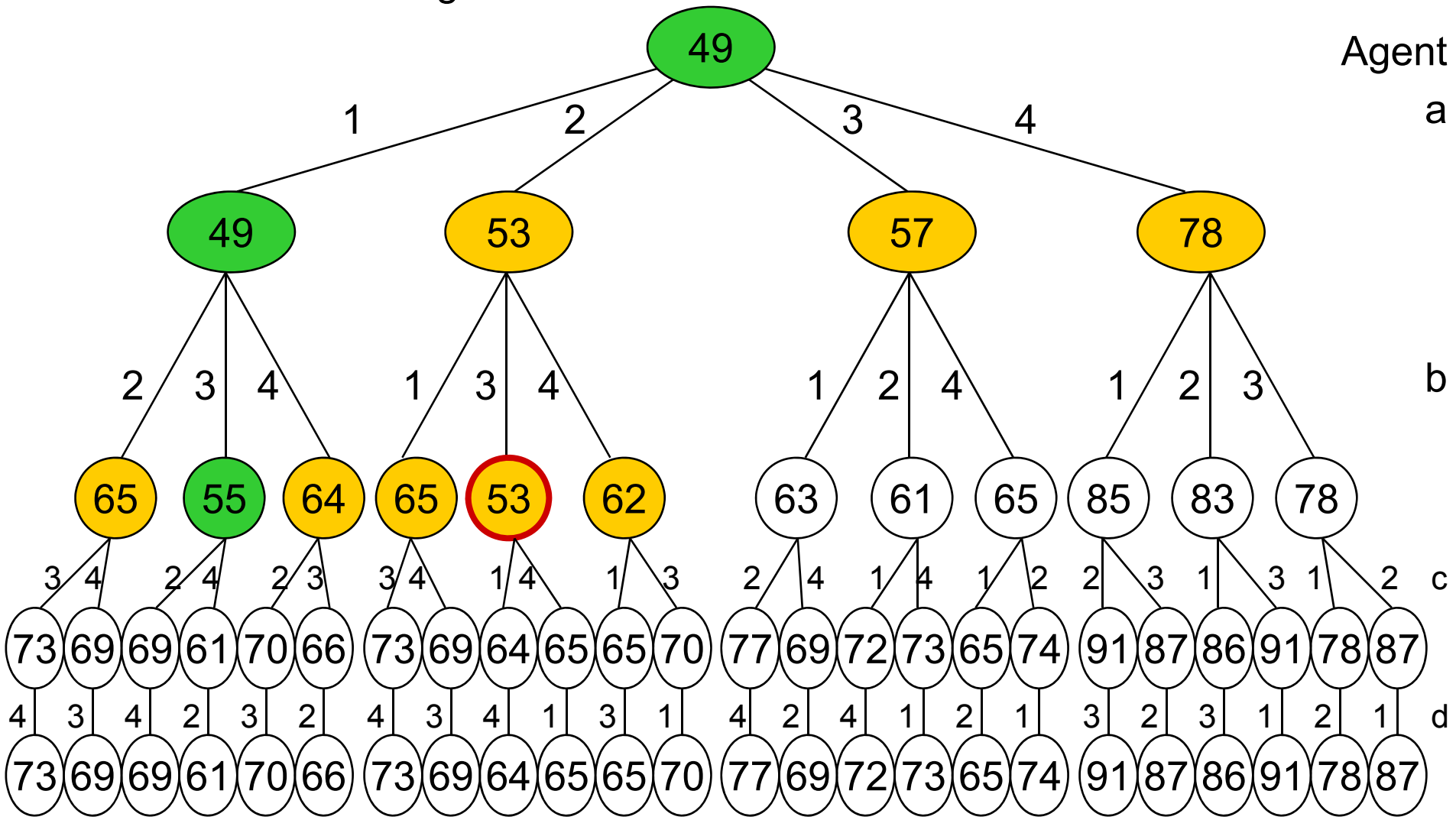
Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 2



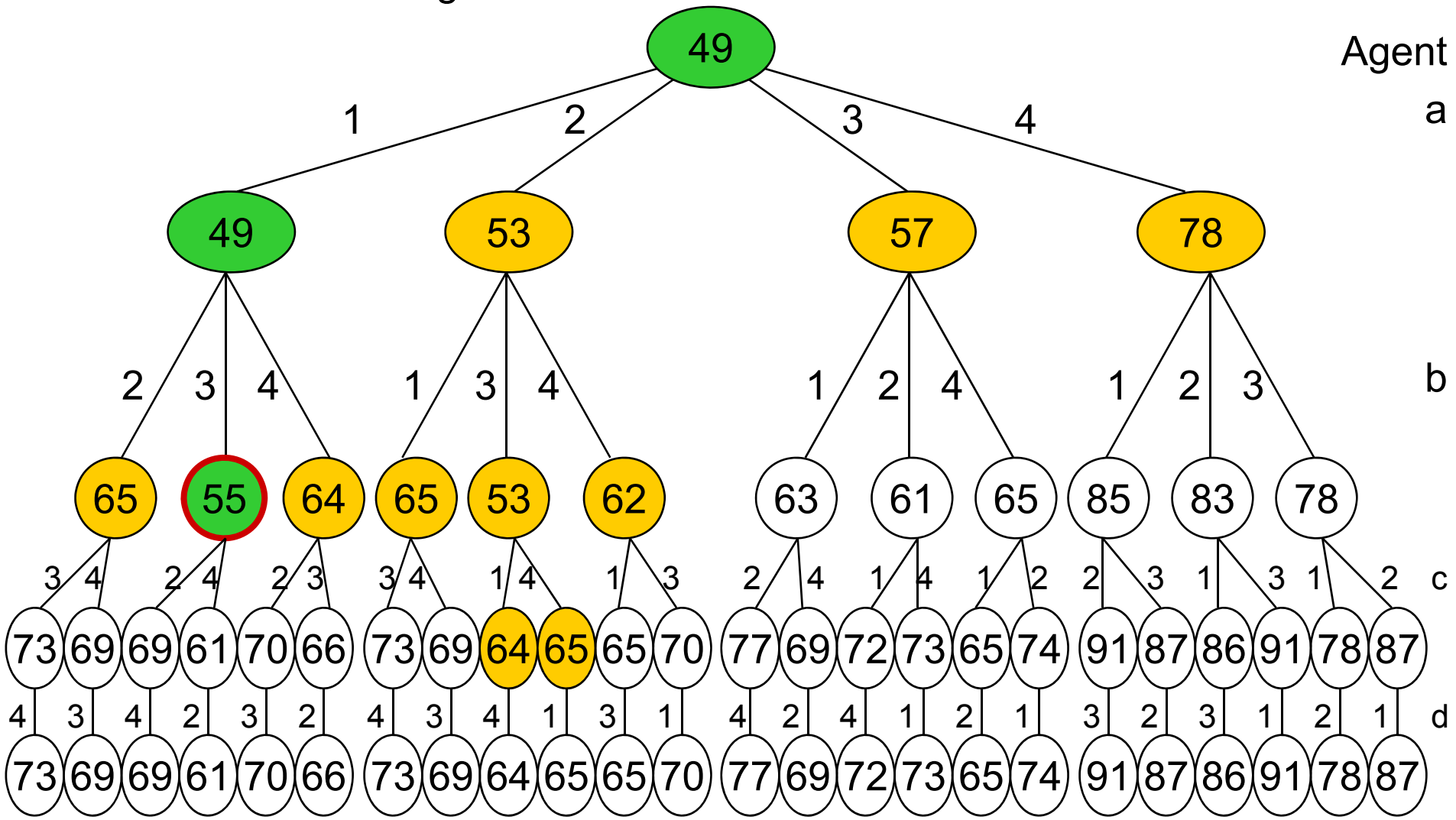
Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 3



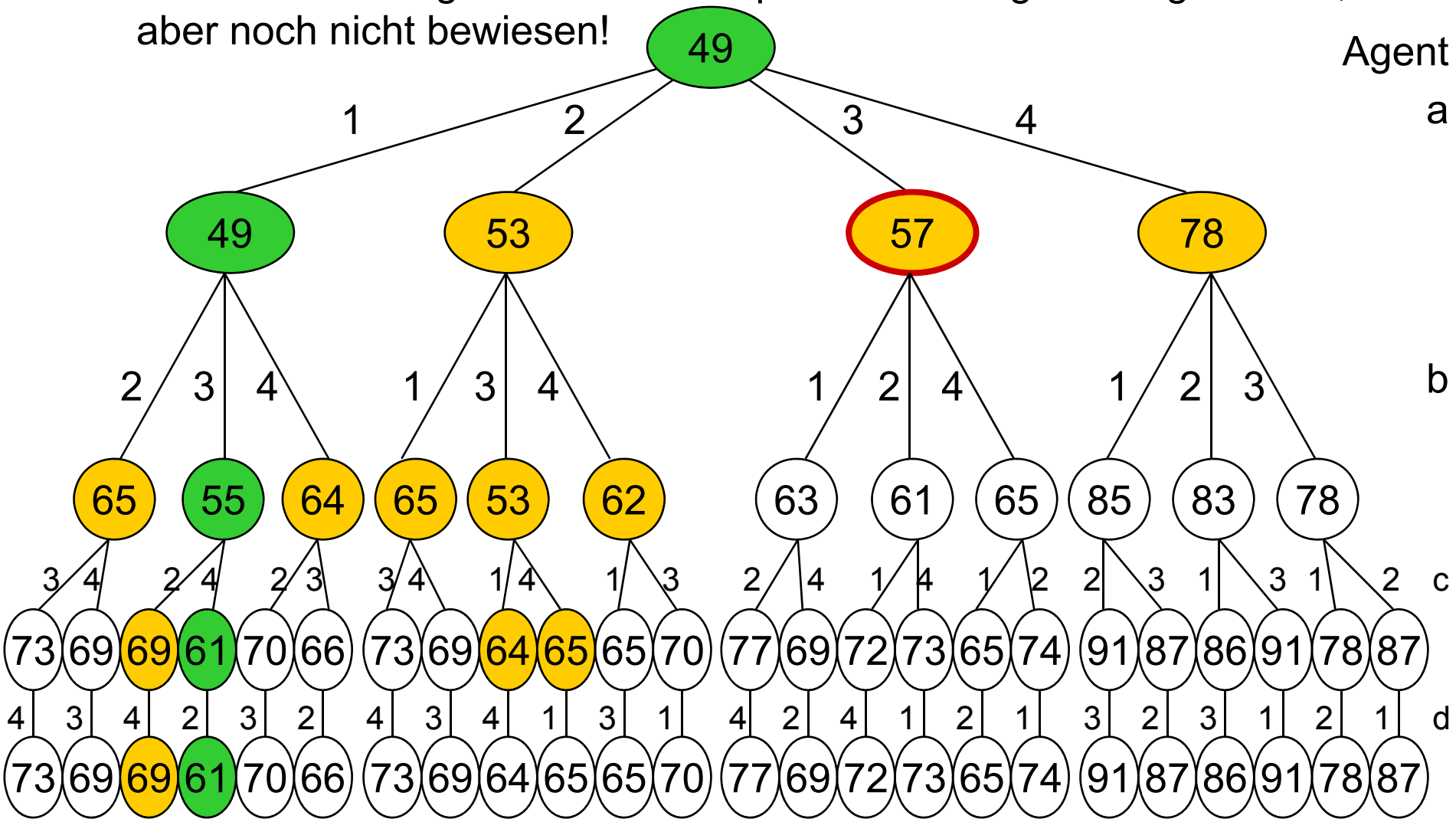
Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 4



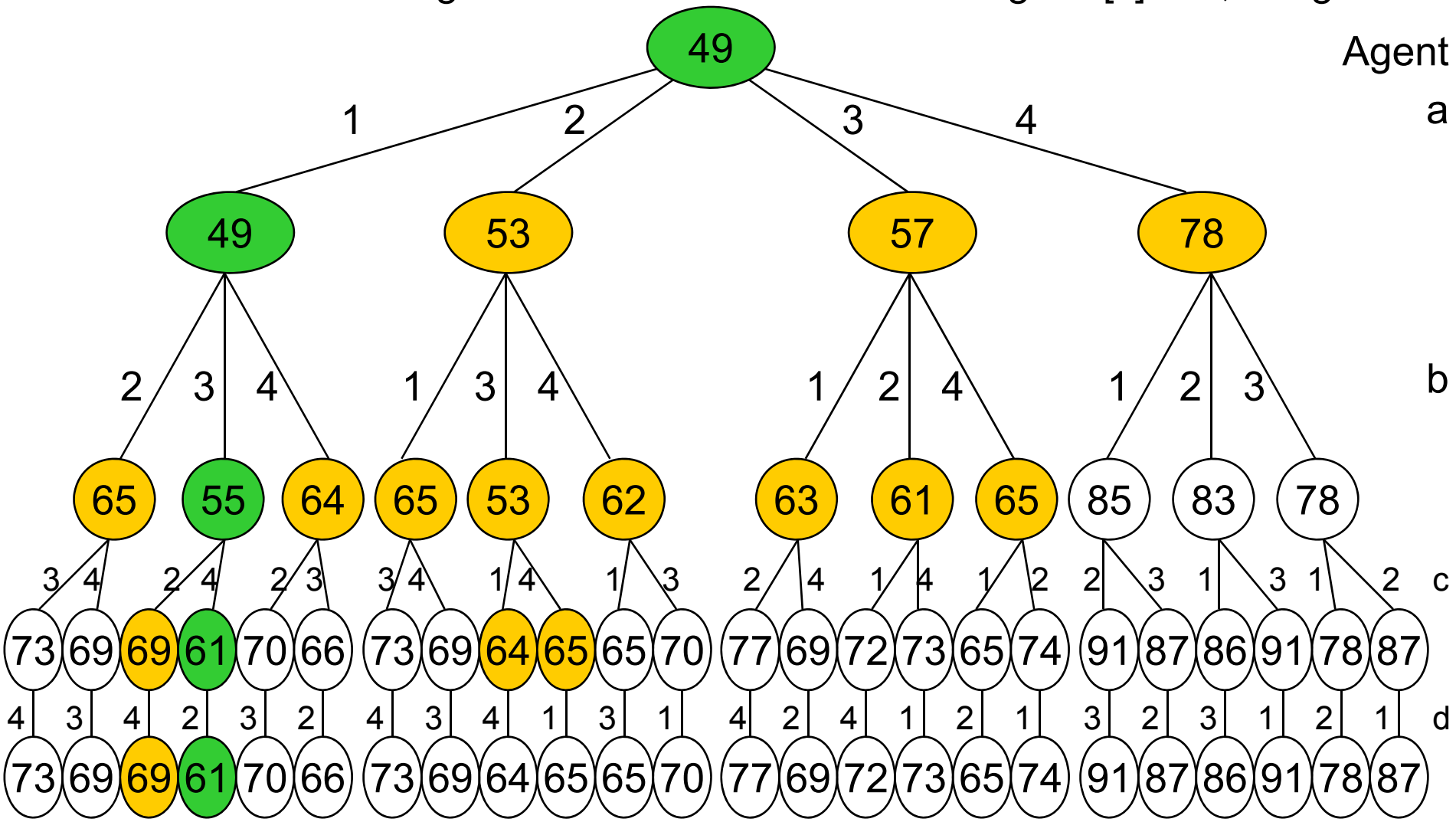
Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 5: optimale Lösung wurde gefunden, aber noch nicht bewiesen!



Branch & Bound für das Assignment-Problem

- Nach Auswertung von Knoten 6: für alle Knoten gilt lb[v] ≥ U, fertig!

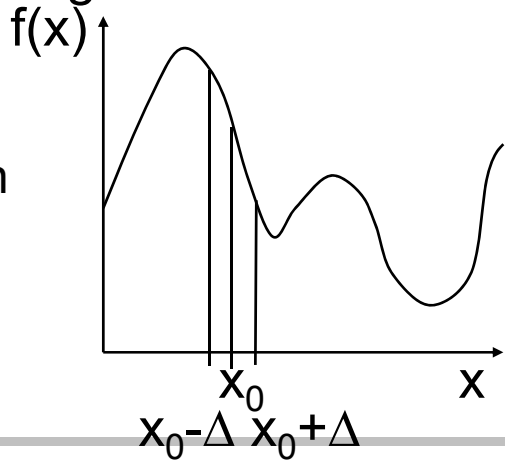


Kommentare zu Branch & Bound

- **Branch & Bound** gehört zu den besten bekannten Strategien für NP-schwere, kombinatorische Optimierungsprobleme.
- Die Qualität der Schranken hat einen großen Einfluss auf die Laufzeit des Algorithmus.
- Das **Finden der oberen Schranke** (bei einem Maximierungsproblem) ist der schwierigste Schritt bei der Entwicklung von Branch&Bound Algorithmen.
- Die **Reihenfolge**, in der Knoten besucht werden, hat einen Einfluss auf die Laufzeit des Algorithmus.
- Gängige **Suchstrategien** sind Depth-First und Best-First:
  - **Depth-First**: durchmustere den Suchbaum rekursiv in die Tiefe
    - Vorteil: speichereffizient Nachteil: potentiell längere Laufzeit
  - **Best-First**: wähle vielversprechendsten Knoten aus der aktuellen Suchfront
    - Vorteil: potenziell kürzere Laufzeit Nachteil: hoher Speicherbedarf
  - Depth-First und Best-First sind zu einer Strategie, die sich je nach Speicher- und Laufzeitverbrauch adaptiert, kombinierbar.

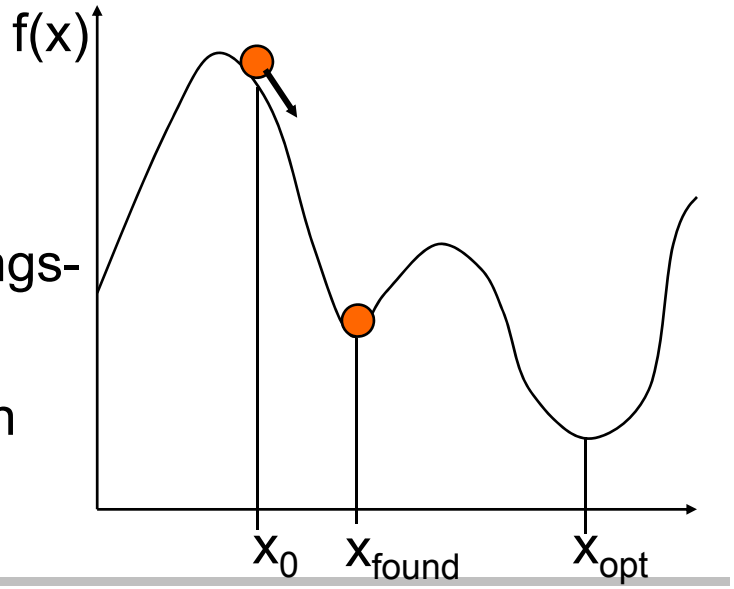
8.3 Heuristische Verfahren

- **Heuristische Verfahren**:
  - Algorithmen zur Lösung von Optimierungsproblemen ohne Gütegarantie
  - Alternative Vorgehensweisen:
    - Generische Optimierungsstrategien, die sich auf viele verschiedene Probleme anwenden lassen
    - Implementierung einer intuitiven Vorgehensweise zur Optimierung des Problems
- **Lokale Suche**
  - Exploration des Suchraums durch schrittweises Verändern einer Lösung (Transition: Übergang zu einer benachbarten Lösung)
  - Annahme: Optimierungsfunktion ist annähernd stetig.
  - Bestandteile:
    - Funktion zur Generierung einer gültigen Lösung
    - Definition von ‚Nachbarschaft' zwischen Lösungen
      - geringfügige Änderung der Lösung
      - möglichst nur geringfügige Änderung des Optimierungsfunktionswertes



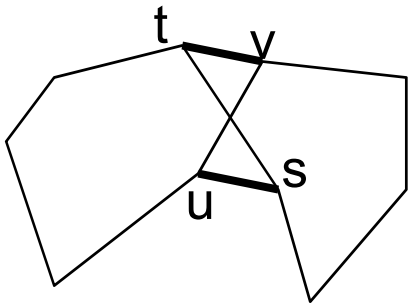
Lokale Suche

- Idee:
  - bestimme eine gültige Lösung
  - wiederhole bis keine Verbesserung mehr erzielt wird:
    - exploriere die Nachbarschaft der Lösung
    - wähle besten Nachbarn
- Eigenschaften der Lokalen Suche:
  - führt von einer gegebenen Lösung zum nächsten lokalen Optimum
  - kann ein lokales Optimum nicht wieder verlassen
- Bedeutung der ‚Anzahl der Nachbarn'
  - gering: kurze Laufzeit eines Optimierungs-schritts
  - hoch: geringe Chance, in einem lokalen Optimum stecken zu bleiben



Lokale Suche für das TSP-Problem

- Nachbarschaft: *two-interchange move*
  - wähle zwei Kanten (u,v), (s,t) der Tour
  - ersetze sie durch Kanten (u,s), (v,t)
  - Anzahl der Nachbarn: n(n-3)/2



- Algorithmus

```
LOCAL-SEARCH-TSP( G, c )
    // T stellt eine Tour dar, Generierung z.B. mit Approximation über
    // minimalen Spannbaum, c beschreibt die Kosten der Tour
  opt = CREATE-TSP-TOUR(G, c)
  do
    T = opt
    for( each (u,v), (s,t) ∈ T )
      if( c(opt) > c(T)-c(u,v)-c(s,t)+c(u,s)+c(v,t) )
        opt = TWO-INTERCHANGE-MOVE( T, (u,v), (s,t) )
  while( T ≠ opt )
  return T
```

Simulated Annealing

- Wie kann man das „Stecken-bleiben“ in lokalen Minima vermeiden?
  - Akzeptanz auch von Verschlechterungen während der lokalen Suche
  - Verschlechterungen werden nur mit einer gewissen Wahrscheinlichkeit akzeptiert, die vom Ausmaß der Verschlechterung abhängt.
  - Die Wahrscheinlichkeit zur Annahme von Verschlechterungen sinkt über die Optimierungszeit.
- **Simulated Annealing**
  - Optimierungsschema in Anlehnung an den physikalischen Prozess der langsamen Abkühlung zur Erzeugung niederenergetischer Festkörper (Kristalle)
  - hohe Temperatur: Verschlechterungen werden mit hoher Wahrscheinlichkeit akzeptiert.
  - niedrige Temperatur: Verschlechterungen werden mit geringer Wahrscheinlichkeit akzeptiert.
  - Während der Optimierung verringert sich die Temperatur nach einem festen Abkühlungsschema (*Cooling Schedule*)

Metropolis-Kriterium

- Wahrscheinlichkeitsverteilung der Akzeptanzfunktion

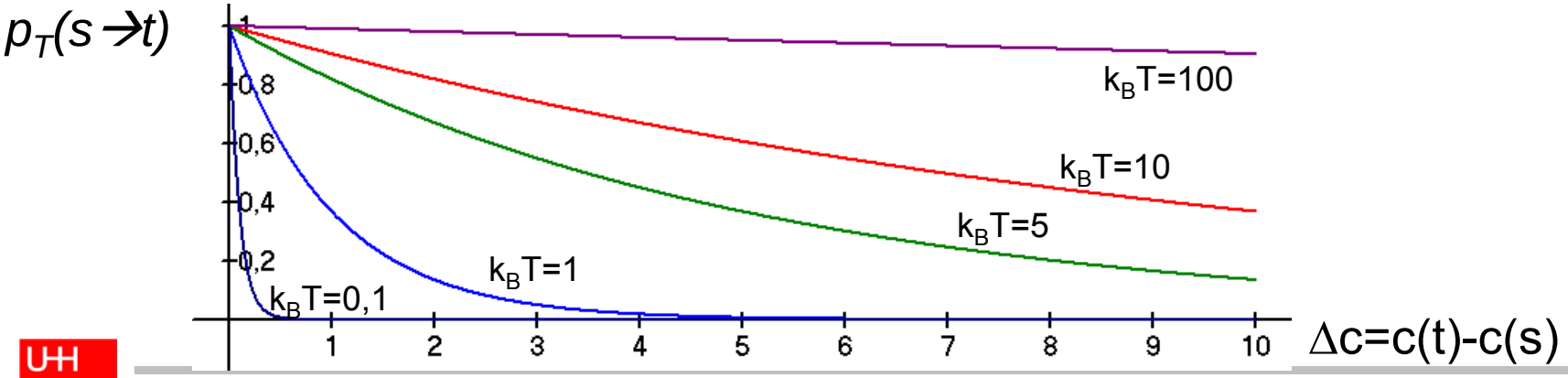
  $p_T(s \rightarrow t)$ (angenommen, wir minimieren)

  - p wird kleiner, je größer die Differenz von c(s) und c(t) ist
  - p wird größer, je höher die Temperatur ist

$$p_T(s \rightarrow t) = e^{-\frac{c(t)-c(s)}{k_B T}}$$

k_B: Boltzmann Konstante
T: Temperatur
c: Optimierungsfunktion

  - Funktion ist gewählt in Anlehnung an die Thermodynamik
  - in der Optimierung, dient k_B als Skalierungsfaktor, da die Optimierungsfunktion typischerweise keine Energie ist



Simulated-Annealing Algorithmus

- Schritte des SA-Algorithmus (auch Metropolis-Algorithmus)

```
// berechne eine initiale, gültige Lösung A
// wähle Anfangstemperatur T und Abkühlungsschema f(T,I)
// Variablen: (T: aktuelle Temperatur, I: Zeitäquivalent)

I = 0
while( T > ε )
  B ← arbitrary select B ∈ NEIGHBORHOOD(A)
  if( c(B) ≤ c(A) ) A = B
  else
    r ← random number from (0,1)
    if( r < p_T( A → B ) ) A = B
  I = I+1; T = f(T,I);
return A
```

Simulated Annealing

- Wahl der Systemparameter:
  - **Initiale Temperatur:** wird so gewählt, das möglichst alle Transitionen akzeptiert werden können
  - **Abkühlungsschema:** typisch d Zeitschritte konstant, danach Reduktion um einen Faktor $0.8 \leq r \leq 0.99$, d.h. $f(T, I) = r^{\lfloor I/d \rfloor} T$
- Kommentare
  - SA konvergiert – bei richtiger Wahl von Nachbarschaft, Anfangstemperatur, Abbruchkriterium und Abkühlungsschema – statistisch gegen das globale Optimum.
  - Konvergenz ist nur logarithmisch, SA kann nicht dazu eingesetzt werden, eine optimale Lösung mit Gütegarantie zu erhalten
  - SA ist typischerweise sehr rechenintensiv
  - zur Wahl der Systemparameter sind viele Tests notwendig
  - typischerweise werden deutlich bessere Ergebnisse erzielt als mit einer einfachen lokalen Suche

Genetische Algorithmen (GA): Motivation

- Heuristisches Optimierungsverfahren, inspiriert durch Optimierungsprozesse der Natur:
  - Darstellung von Lösungen (Individuen) durch Vektoren über einen endlichem Alphabet, z.B. Bitstrings (Gene)
  - Erzeugung von neuen Lösungen durch Reproduktion und Rekombination (cross-over) bekannter Lösungen
  - Bewertung und Auswahl von Lösungen durch Fitness-Funktion
  - Zufällige Modifizierung der generierten Lösungen (Mutation)
  - Verwerfen von Lösungen in Abhängigkeit von ihrem Fitnesswert (Selektion)
- Unterscheidende Eigenschaften von GA‘s im Vergleich zu Simulated Annealing:
  - Erzeugung einer Menge von Lösungen (Population)
  - Keine lokal beschränkte Suche (Rekombination)

Beispiel: Genetischer Algorithmus für TSP

- Gesucht: Kürzester Weg über alle Städte 1, 2, ..., n

1. Ausgangspopulation t = 0: $P_0 = \{\alpha_1, \alpha_2, ..., \alpha_k\}$, k = 30 Individuen mit zufälligen Permutationen der Zahlen 1 bis n
2. Berechnung der Fitness *fitness*(α_i) für alle i = 1, 2, ..., k
3. Zufällige Auswahl von k/2 Paaren (β^1_j, β^2_j), j = 1, 2, ..., k/2
   $\rightarrow$ Wahrscheinlichkeitsverteilung P(α_i) korrelliert mit *fitness*(α_i) !
   1. Erzeugung von zwei Nachkommen durch cross over-Rekombination jedes Paars (β^1_j, β^2_j)
      - Wähle zufällig cross over-Punkt $1 \leq c \leq n$
      - Übernehme Reihenfolge der ersten c Städte aus β^1_j und ordne die restlichen Städte in der Reihenfolge an, wie sie in β^2_j vorkommen (und umgekehrt)
   2. Einfügen der Nachkommen in Population P_{t+1}
4. Zufällige Mutation jedes Individuums in P_{t+1}, z.B. durch Vertauschung benachbarter Zahlen
5. Wenn maximale Populationszahl t_{max} erreicht oder Fitness-Zielwert erreicht, dann stop, sonst, gehen zu Schritt 2

Genetische Algorithmen: Schema-Theorem

- Schema: Gemeinsame Eigenschaften mehrerer Individuen, dargestellt durch ein gemeinsames, lokal begrenztes Bitmuster:

**********0110**01001**********

Schema der Länge 11

- Erklärung der Funktionsweise von GA‘s durch das Schema-Theorem:

  Kurze Schemata mit hohem Fitnesswert werden ihr Vorkommen in einer Population im Verlauf der Evolution steigern (building block-Hypothese)

  → „Resistenz“ gegen cross-over Operation

Evolutionsstrategien (ES): Motivation

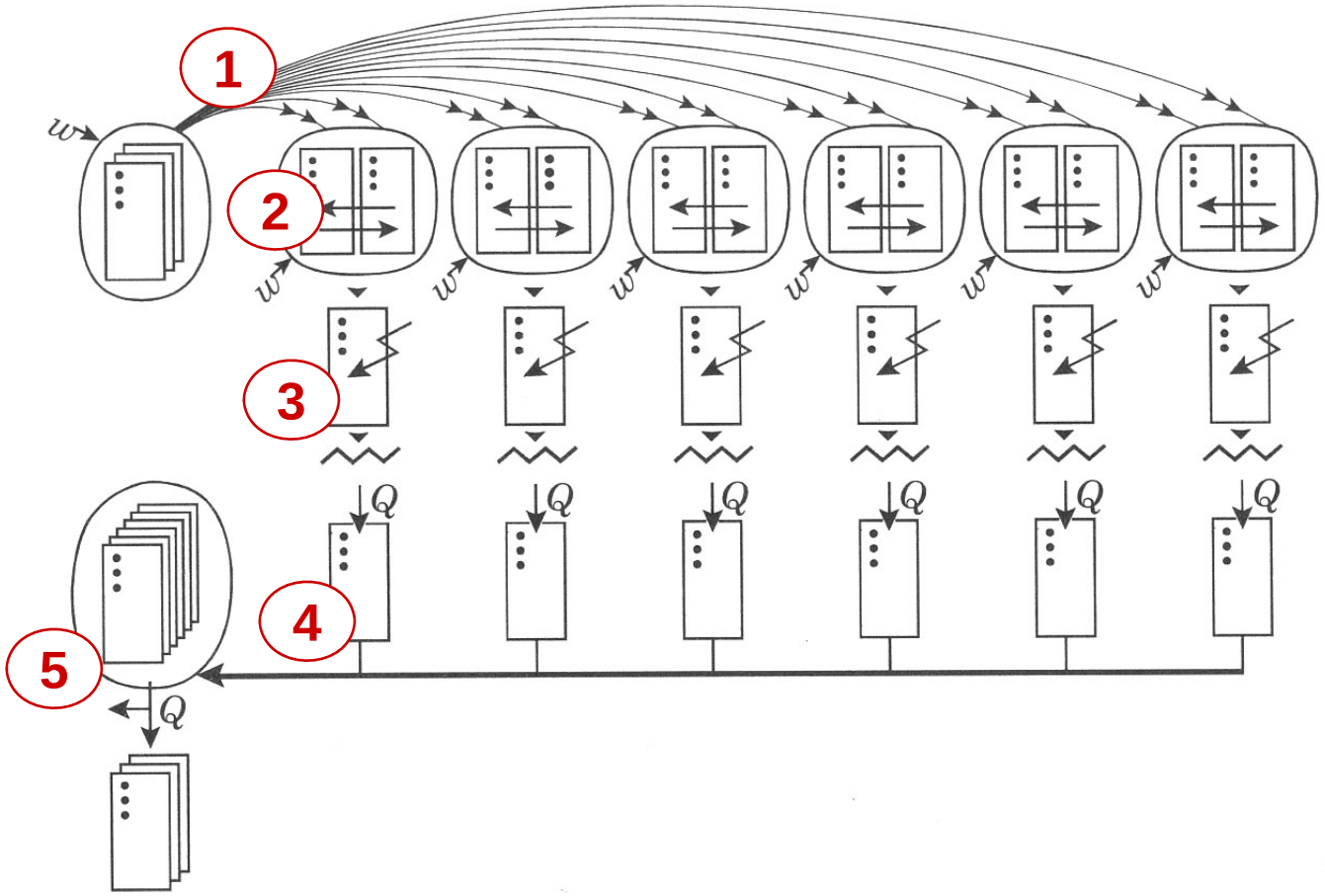
- Vielseitige, heuristische Verfahren zur Lösung von Optimierungsproblemen
- Angelehnt an das Evolutionsprinzip der Natur; ähnliche Konzepte wie bei Genetischen Algorithmen:
  - Darstellung von Lösungen durch Individuen und Populationen
  - Variation der Lösungen z.B. durch Rekombination oder Mutation
  - Auswahl und Überleben der Lösungen nach ihrem Fitnesswert
- Unterschiede:
  - ES basieren auf metrisch skalierbaren Variablenwerten und einem stetigen Lösungsraum (Prinzip der starken Kausalität)
  - Philosophien:
    - ES: Phänotypischer Algorithmus
      (Imitation der Wirkung von genetischen Operationen)
    - GA: Genotypischer Algorithmus
      (Imitation der Funktion von genetischen Operationen)

Varianten von Evolutionsstrategien

- Einheitliche Beschreibung von Evolutionsstrategien durch $(\mu/\rho \dot{+}, \lambda)$-Notation:
  - μ Eltern erzeugen λ Nachkommen
  - Zur Erzeugung eines Nachkommen werden jeweils ρ der μ Eltern ausgewählt und ihre Variablenausprägungen gemischt
  - Die Eltern der neuen Generation werden aus den Nachkommen ((μ , λ)-ES) bzw. aus Eltern und Nachkommen ($(\mu + \lambda)$-ES) selektiert

- Erweiterung: ES mit Mutationsschrittweitenregelung (MSR)
  - Jedes Individuum i speichert neben der Ausprägung x^g_i einer Variable g auch die Schrittweite δ^g_i, mit der diese Variable verändert werden kann

Schema einer (3/2, 6)-gliedrigen Evolutionsstrategie



Ablauf:

(1) Auswahl der Eltern für Nachkommenerzeugung

(2) Erzeugung der Nachkommen durch cross over

(3) Mutation der erzeugten Nachkommen

(4) Realisierung der Nachkommen und Bewertung der Fitness

(5) Auswahl der besten Nachkommen als Eltern für nächste Generation

Eigenschaften von ES und GA

- Die Qualität der gefundenen Lösungen wird u.U. stark durch die Initialisierung der Anfangspopulation und den verwendeten Zufallsgenerator beeinflusst (bedingte Reproduzierbarkeit)

- Richtige Wahl von Parametern erfordert in der Regel viele Tests
  - Darstellung/Codierung der Individuen
  - Populationsgröße
  - Nachkommenzahl
  - Mutationswahrscheinlichkeit und/oder -schrittweite

- GA erfordern eine diskrete Codierung der Lösungen, möglichst nach dem Prinzip der starken Kausalität
  - Problembeispiel: Inversion nur eines führenden Bits bei binärcodierten Dezimalzahlen bedeutet große Veränderung des Zahlenwertes (schwache Kausalität)

Kommentare zu heuristischen Verfahren

- Heuristische Verfahren ermöglichen die Lösung einer Vielzahl verschiedener Optimierungsprobleme mit einem einheitlichen algorithmischen Schema.
- Heuristische Verfahren liefern keine Gütegarantie.
- Algorithmische Schemata gibt es viele:
  - Simulated Annealing
  - Genetische Algorithmen und Evolutionsstrategien
  - Tabu Search, Flooding, Particle Swarm Optimization, Ant Colony Optimization, etc.
- Heuristische Verfahren stellen das letzte Mittel dar: Sie sollten angewendet werden, wenn
  - das Problem schwer zu lösen ist
  - eine Approximation nicht bekannt ist
  - eine exakte Lösung zu zeitintensiv ist
  - eine kombinatorische Modellierung nicht möglich ist.

Rückblick auf die Vorlesung

1. Einführung
- Beschreibung und
- Analyse von Algorithmen

2. Elementare DS
- Beschreibung und
- Analyse von DS

3. Sortieren
- Divide&Conquer
- Heaps
- stochastische Analyse
- Selektion / Median

8. Schwere Probleme
- Approximationsalgorithmen
- Exakte Verfahren (B & B)
- Heuristische Verfahren

Algorithmen und Datenstrukturen

4. Suchen
- balancierte Suchbäume
- Hashing

7. NP-Vollständigkeit
- Komplexität von Problemen
- Zugehörigkeit zu NPC
- Reduktionsbeweise

6. Dyn. Programmierung
- Charakterisierung
- 4-Phasen Entwicklung
- Matrix-Kettenmultiplikation

5. Graphen
- Systematische Suche
- Spannbäume
- Kürzeste Wege

Rückblick auf die Vorlesung

- **Lernziele** (aus dem Modulhandbuch)

  Vermittlung von Problemlösungskompetenz (Konzept und Realisierung) zur Lösung formalisierbarer, schwieriger Probleme

  - Selbstständiges, kreatives **Entwickeln** von Alg. und DS
  - **Korrektheit**sbeweise und **Effizienz**analyse
  - Selbstständiges **Aneignen** neuer Alg. und DS
  - **Übertragung** bekannter Alg. auf neue Probleme
  - **Modifikation** bekannter Alg. auf veränderte Anforderungen
  - **Beurteilung** der Qualität von Alg.
  - Erkennen grundlegender **Beschränkungen** von Alg.
  - Einschätzung von Problemen in Hinblick auf ihre **Komplexität**

Weiterführendes Modul **Algorithmik** (9LP, immer im WiSe)

- **Analysemethoden**:
  - Amortisierte Analyse
- **Fortgeschrittene Datenstrukturen**:
  - Binomial / Fibonacci-Heaps
  - Splay-Trees, Treaps
- **Weiterführende Graphalgorithmen**:
  - All-Pairs und algebraische kürzeste Wege,
  - Netzwerkfluss-Algorithmen,
  - Matching, …
- **Numerische Algorithmen**
  - Matrixoperationen
  - Lineare und Ganzzahlige Programmierung
- **Algorithmische Geometrie**:
  - Schnittprobleme, A&D zu Raumanfragen, Konvexe Hüllen
  - Voronoi-Diagramme und Delaunay-Triangulierung
  - Umschließende Kreise

Das Wichtigste zum Schluss

Was sollten Sie jetzt für‘s (Berufs-)Leben gelernt haben?

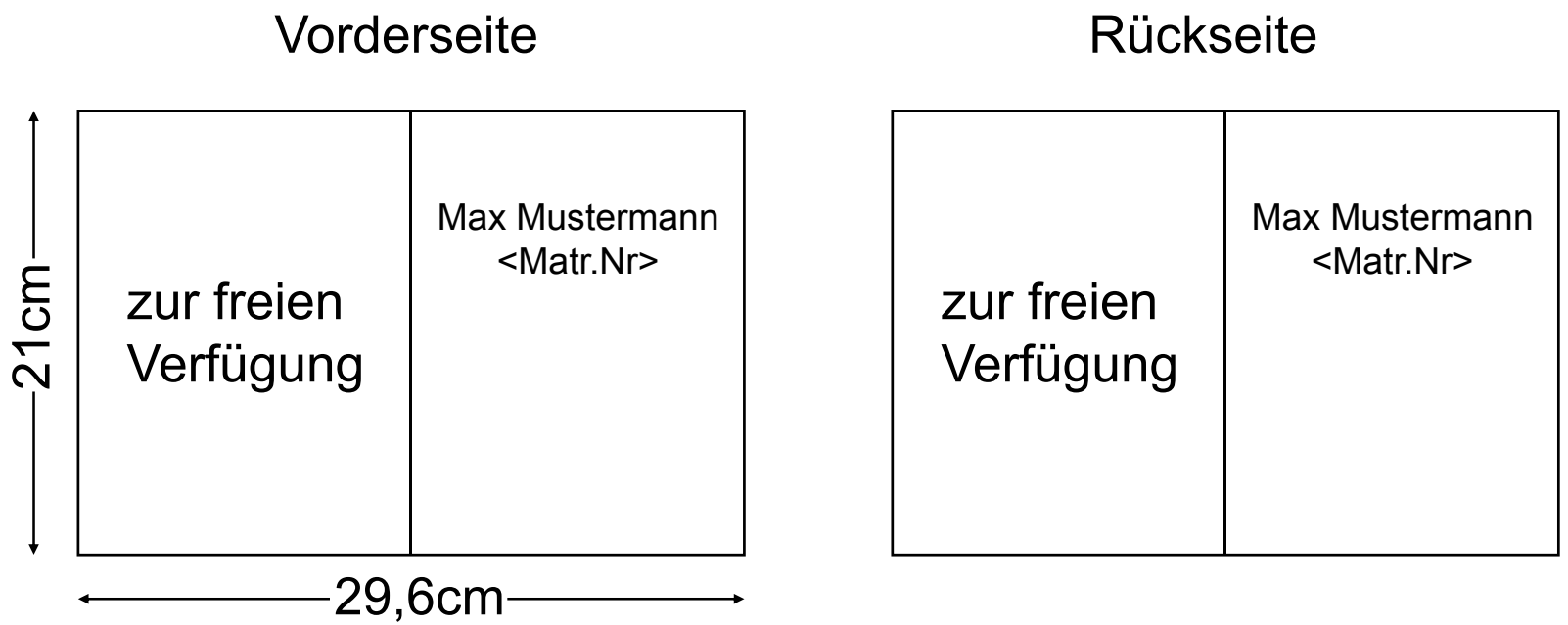
1. Formale Struktur umgangsprachlicher Probleme erkennen
2. Analogien zu bekannten Problemen aus der Informatik erkennen
3. Lösungsstrategien (Algorithmen) für Probleme entwickeln
4. Algorithmen abstrakt beschreiben können
5. Algorithmen bzgl. Ihrer Problem-Adäquatheit / Komplexität beurteilen können (ohne sie zu implementieren)

Tutorium: Montag, 09.2 9:00 – max. 12:00 Hörsaal C, FB Chemie

Beispielaufgaben stehen unter Stine bereits online.

HINWEIS ZUR KLAUSUR

- Erlaubte Hilfsmittel:
  - Kugelschreiber, etc.
  - keine Elektronik, keine Lupen
  - ein Din-A4 Blatt, **handschriftlich** beschrieben (nicht bedruckt, nicht kopiert)



Hinweis: Das Blatt wird zu Beginn der Klausur abgestempelt und mit abgegeben (geht nicht in die Benotung ein).

Viel Erfolg in der Klausur!