Karlsruhe Institute of Technology

AIFB

Teil 3

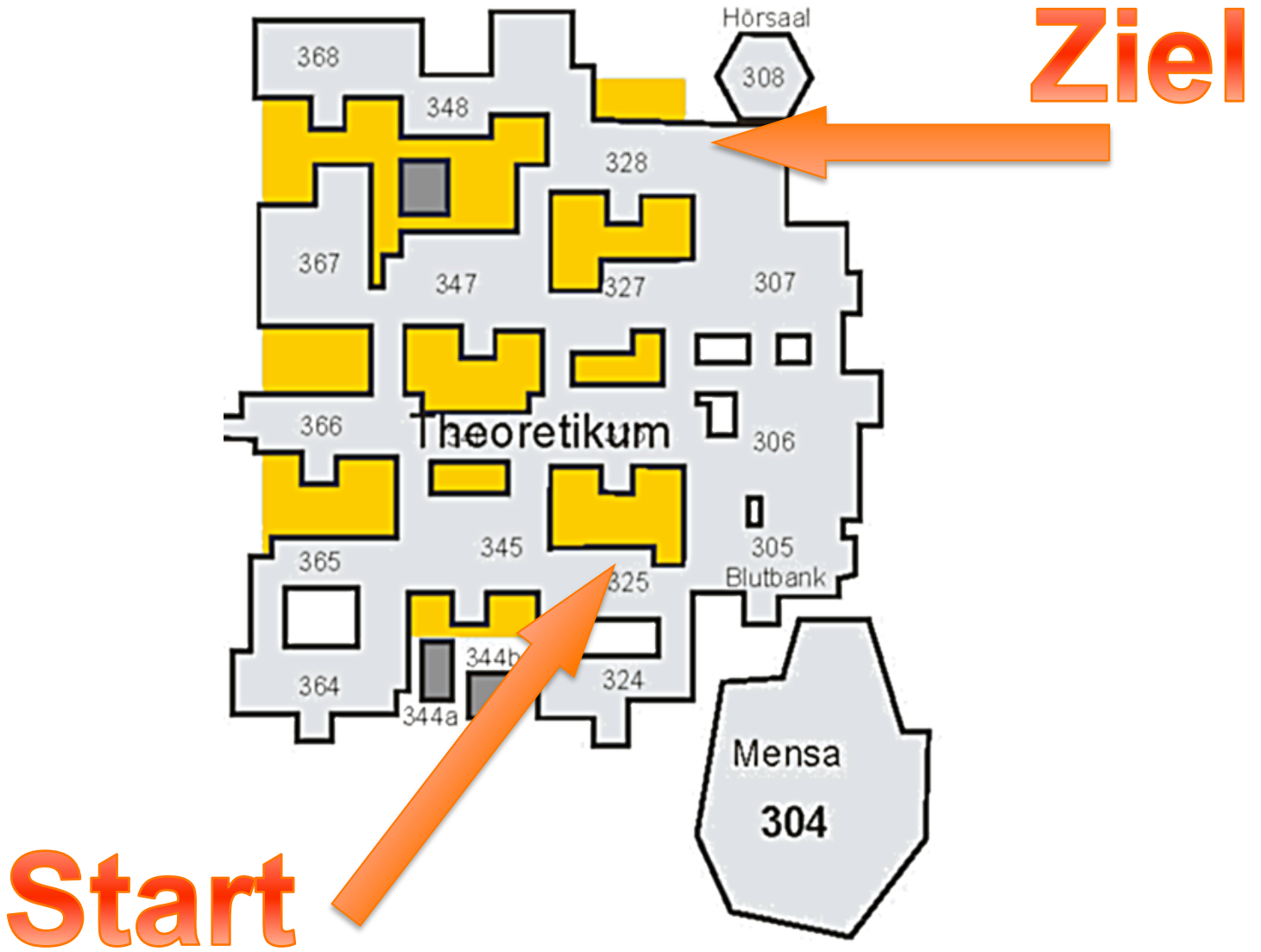
Suchverfahren

Überblick

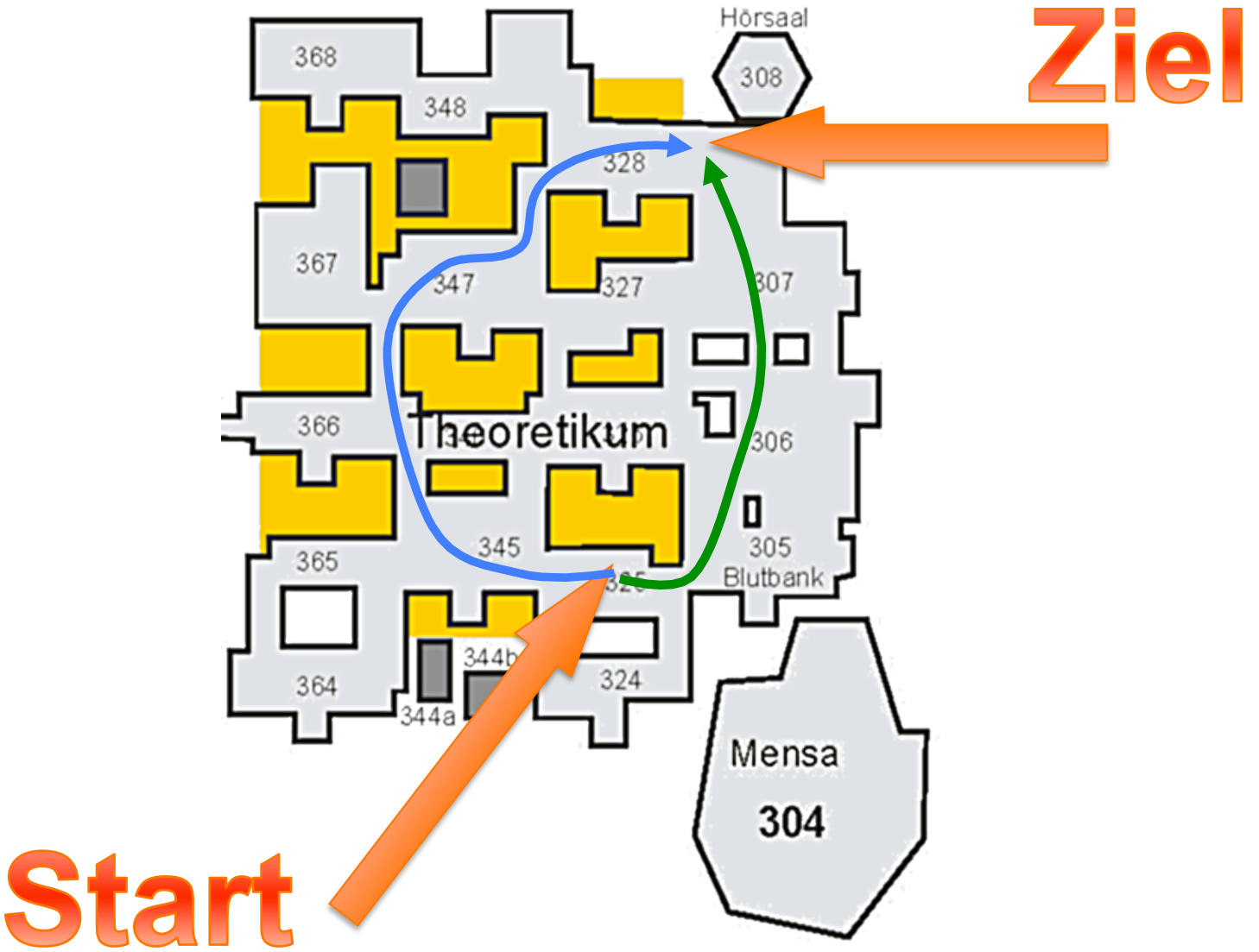
- Suchproblem
- Allgemeines Suchverfahren
- Nicht informierte Suche
- Informierte Suche
- Iterative Suche
- Spiele

Beispiel 1

Beispiel 1


Hörsaal
368
308
348
328
367
347
327
307
366
Theoretikum
306
345
305
365
325
Blutbank
344b
364
344a
324
Mensa
304
Ziel
Start

Beispiel 1


Hörsaal
308
368
348
328
367
347
327
307
Theoretikum
366
306
345
305
Blutbank
365
344b
364
344a
324
Mensa
304
Ziel
Start

SUCHPROBLEME

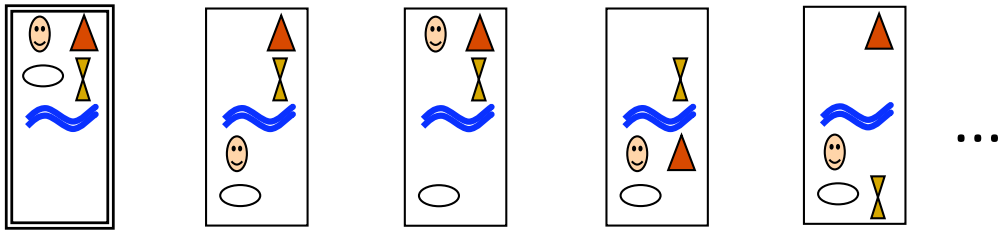
Problemlösen durch Suche

- Problemformulierung = Beschreibung des Suchraums:
  - Zustandsraum *Z*
  - Startzustand *s*
  - Zielzustand *g* und Zieltestfunktion
  - Menge von Aktionen (Operatoren) *A*
  - Zustandsübergangsfunktion $f: Z \times A \mapsto Z$, $f(z,a) = z'$
  - Pfadkostenfunktion
- Lösung
  - Pfad vom Start- zum Zielzustand als Folge von Zuständen bzw. Operatoren
- Performance-Evaluierung durch Addition von
  - Kosten für den Pfad und
  - Kosten für die Suche

Beispiel 2

Bauer, Fuchs, Gans und Getreide

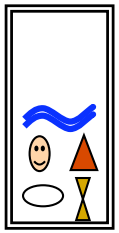
- Zustände:



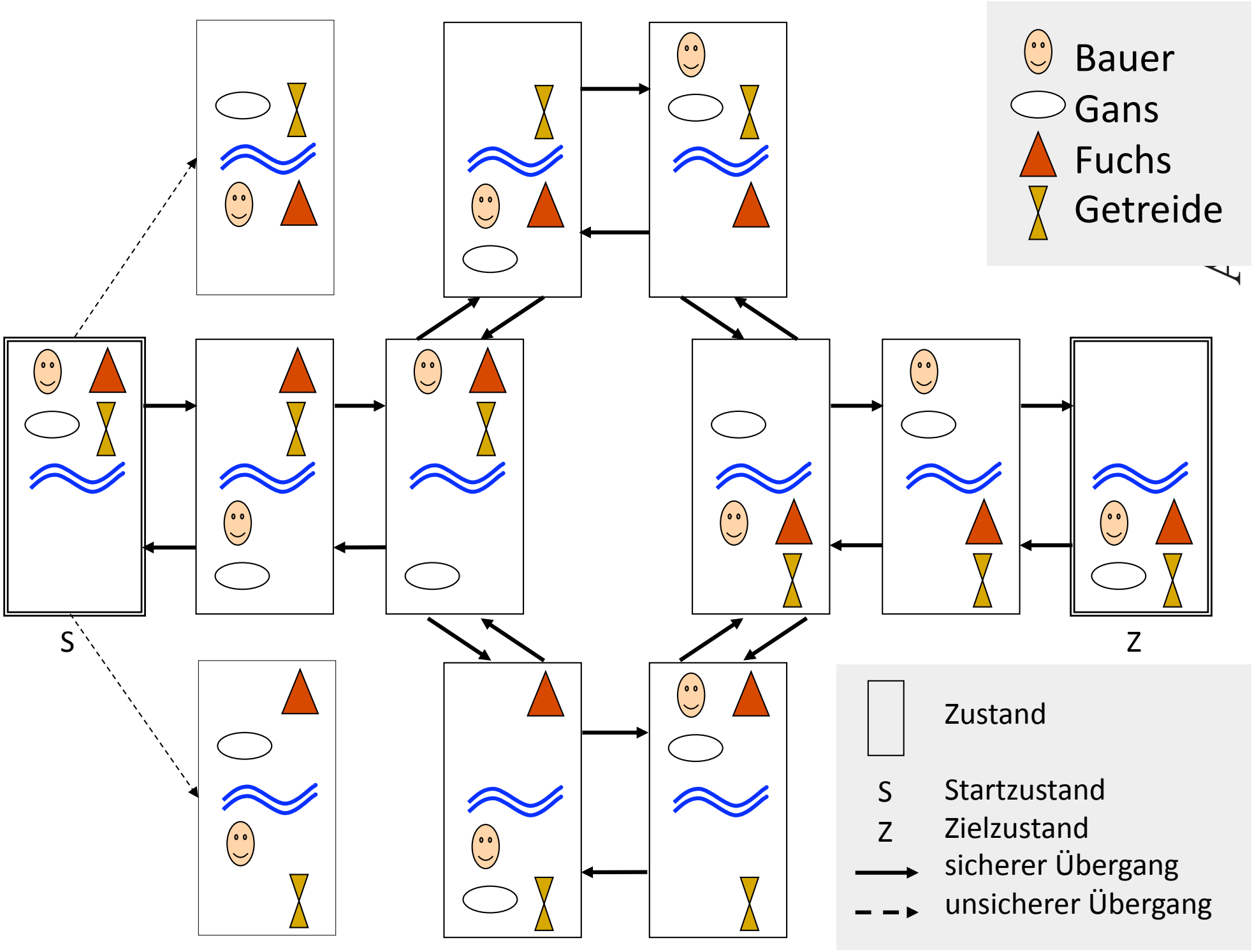
...

- Operatoren:
  Bauer überquert Fluss mit Fuchs, Gans oder Getreide

- Nicht erlaubt Zustände
  - ⇨ Fuchs allein mit Gans (Fuchs frisst Gans)
  - ⇨ Gans allein mit Getreide (Gans frisst Getreide)

- Ziel-Test:
  Zustand  erreicht

- Pfad-Kosten:
  Anzahl der Überquerungen


Bauer
Gans
Fuchs
Getreide
S
Z
Zustand
S Startzustand
Z Zielzustand
sicherer Übergang
unsicherer Übergang

Beispiel 3

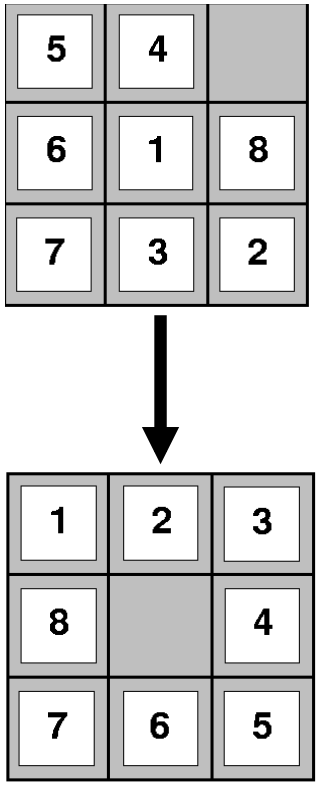
- 8er Puzzle
  - ⇨ Zustände:

    Positionen der Teile
  - ⇨ Operatoren:

    leeres Teil nach {links, rechts, auf, ab} bewegen
  - ⇨ Ziel-Test:

    Zielzustand erreicht?
  - ⇨ Pfad-Kosten:

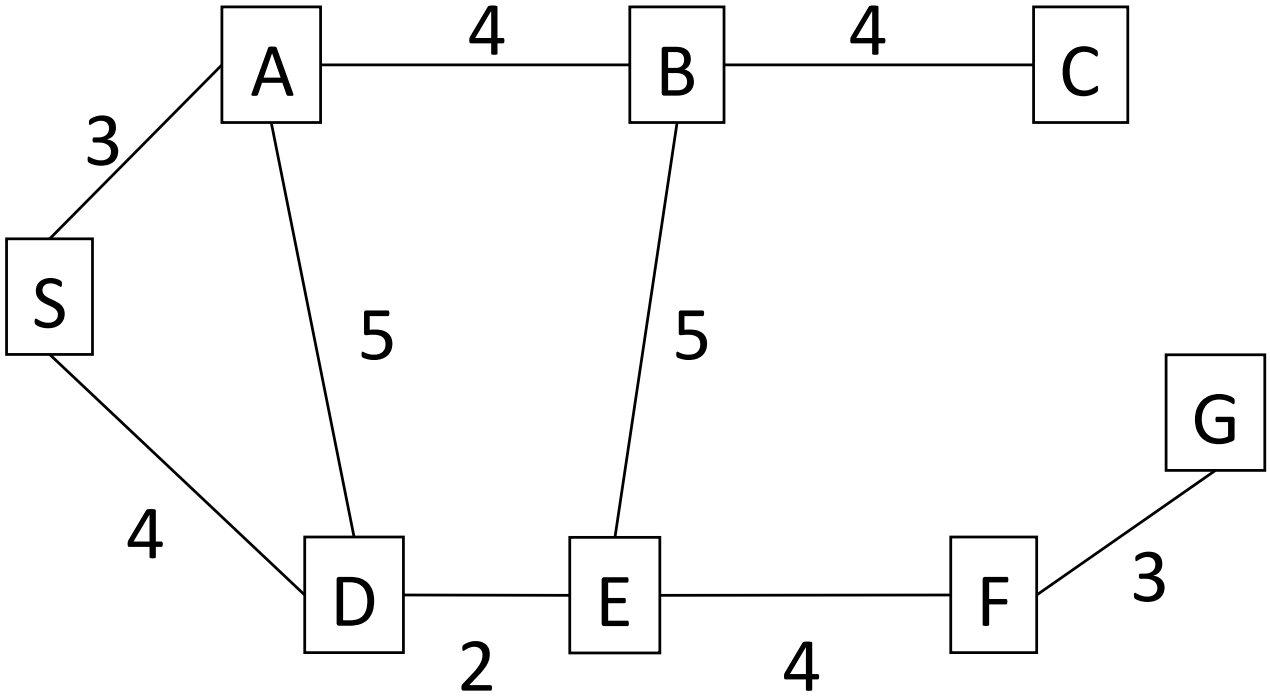
    Anzahl der Bewegungen
- Bestimmung der optimalen Lösung für *n*-Puzzle ist NP-hart



Vorgehensweise

- **Problemformulierung** (Zustände, Operatoren, Ziele, Kostenfunktion)
- (Partieller) Aufbau eines **Suchbaums**:
  - Wurzel stellt Anfangszustand dar,
  - Nachfolgeknoten sind die jeweils direkt erreichbaren Folgezustände
  - zusätzliche Informationen für jeden Knoten: Tiefe bzw. Kosten
  - keine Zyklen (Abbruch, bevor ein solcher entsteht)
- Verwaltung einer Pfadliste, die potentielle Wege vom Start in Richtung Ziel enthält

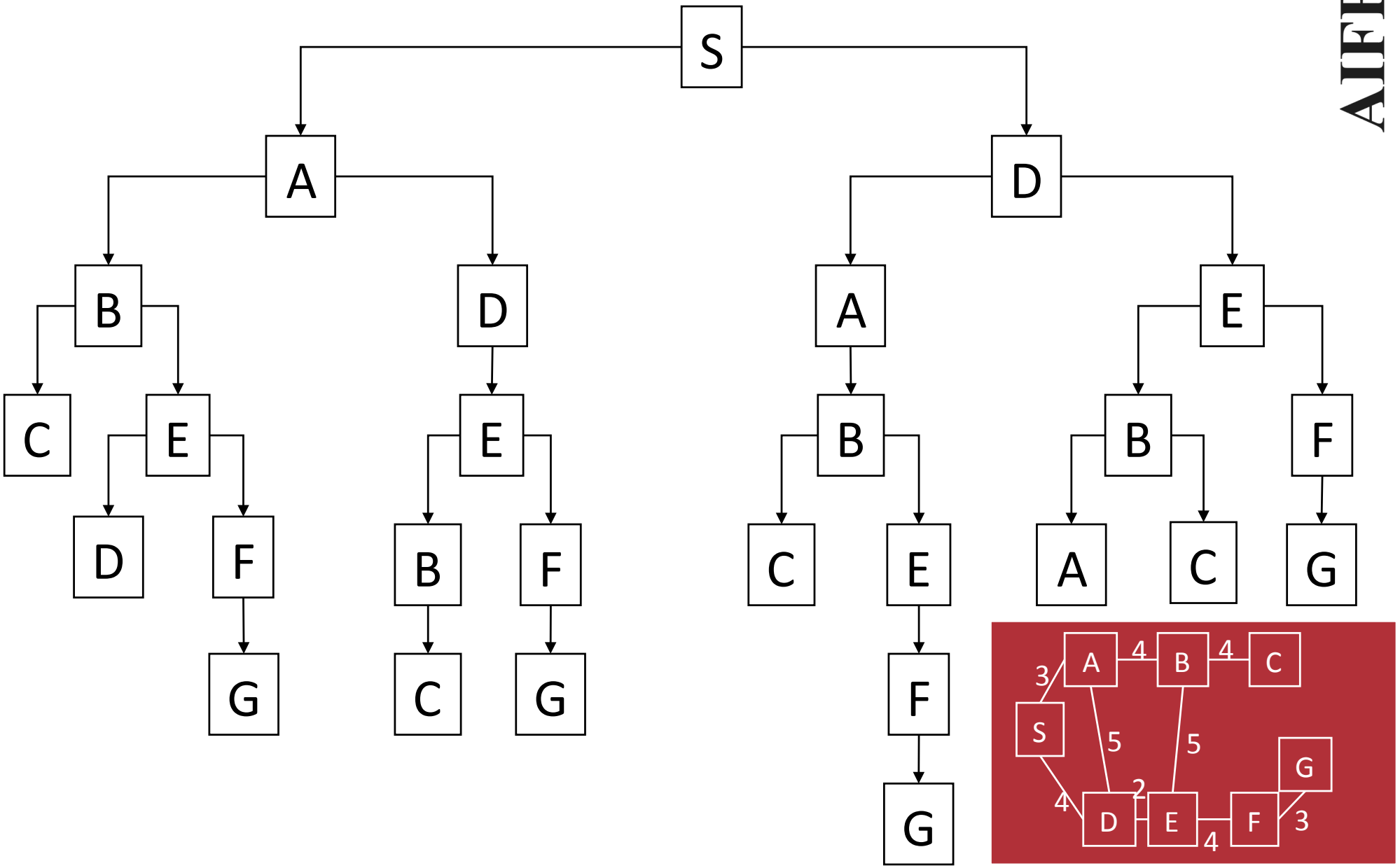
Problemstellung



Gegeben: Gewichteter Graph

Gesucht: Kostenminimaler Pfad von S nach G

Suchbaum



Generischer Algorithmus

- Allgemeine Suche:
  - Initialisiere die Liste der Pfade mit dem Pfad (der Länge 0), der nur aus dem Wurzel-Element besteht
  - Solange die Liste der Pfade nicht leer ist, und das Ziel nicht erreicht wurde:
    - Wähle einen Pfad aus der Liste.
    - Ersetze ihn durch alle Pfade, die aus diesem Pfad durch Erweiterung um einen der Nachfolger entstehen.
    - Pfade, die eine Schleife enthalten, werden gelöscht.
    - Pfade, die nicht weiter führen, werden gelöscht.
- Unterschiede zwischen den Verfahren (im Wesentlichen):
  - Auswahl des zu ersetzenden Pfades (Verwaltung der Pfadliste)

Beurteilung von Suchverfahren

- Vollständigkeit
  - Ein Suchverfahren heißt **vollständig**, wenn es für jedes bearbeitete Problem, für das eine Lösung existiert, eine Lösung findet
- Komplexität:
  - Abhängig von Tiefe *m*, Verzweigungsgrad *b* des Suchbaums
  - Zeit
  - Speicher
- Optimalität
  - Ein Suchverfahren heißt **optimal**, wenn die durch das Verfahren gefundene Lösung von allen möglichen Lösungen für das bearbeitete Problem die geringsten Kosten aufweist

SUCHVERFAHREN IM ÜBERBLICK

Nicht informierte Suche (1)

Nicht optimale Verfahren

- **Breitensuche**: Es wird jeweils der erste Pfad ausgewählt, neue Pfade werden an das Ende der Liste angehängt (Schlange, FIFO-Prinzip)
- **Strahlensuche**: Wie Breitensuche, beschränkt sich jedoch auf eine Auswahl von *w* Knoten in jeder Suchebene (z.B. anhand der bisherigen Kosten)
- **Bidirektionale Breitensuche**: Es wird ein zweiter Suchbaum aufgestellt, der vom Ziel ausgehend rückwärts sucht.

  Eine Lösung ist gefunden, wenn sich beide Suchbäume treffen.
- **Tiefensuche**: Es wird jeweils der erste Pfad ausgewählt, neue Pfade werden an den Kopf der Liste gestellt (Stack, LIFO-Prinzip)
- **Varianten der Tiefensuche**: Tiefensuche mit Begrenzung der Suchtiefe und/oder schrittweiser Erweiterung der Suchtiefe (**Iterative Deepening**)

Nicht-informierte Suche (2)

Optimale Verfahren

- **Britisches Museum:** Der gesamte Suchbaum wird untersucht (mit Tiefen- oder Breitensuche), insbes. kein Abbruch wenn Ziel zum ersten Mal gefunden wird. Alle Pfade zum Ziel werden gefunden.
- **Branch & Bound:** Es wird jeweils der Pfad ausgewählt, der **bisher die geringsten Kosten** verursacht hat.
- **Branch & Bound + Dynamisches Programmierung**: Wie B&B, zusätzliche Überlegungen **vermeiden Suche auf redundanten Teilpfaden.**

Informierte Suche (1)

Es existiert eine Funktion zur Schätzung der Restkosten bis zum Ziel vom aktuellen Knoten aus

Nicht optimale Verfahren: Es wird jeweils der Pfad ausgewählt, der am erfolgversprechendsten aussieht

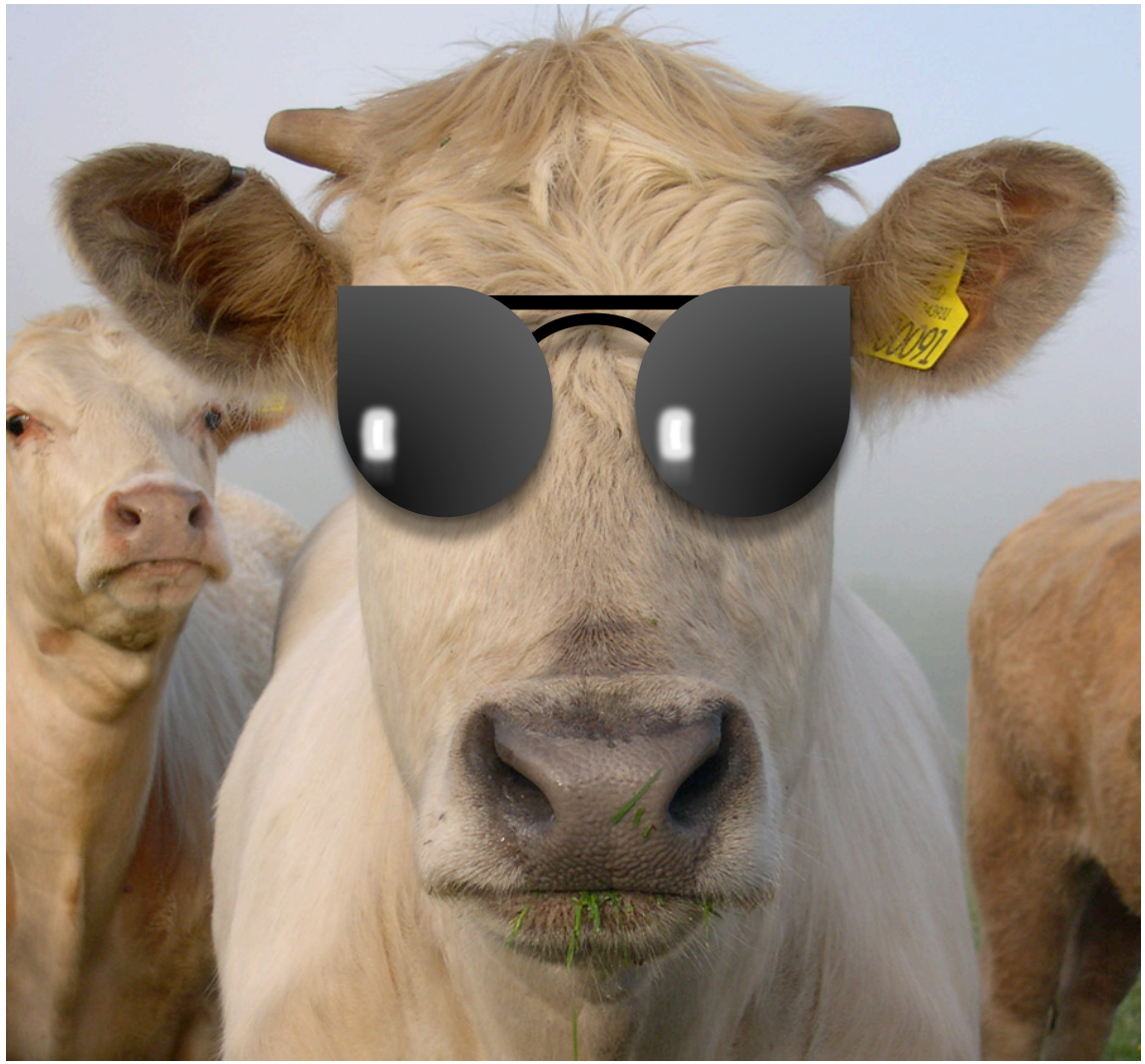
- **Strahlensuche**: Wie Breitensuche, beschränkt sich jedoch auf eine Auswahl von *w* Knoten in jeder Suchebene (anhand der Schätzfunktion)
- **Bergsteigen (Hillclimbing)**: Wie Tiefensuche, die Reihenfolge der zu besuchenden Nachfolgerknoten wird anhand der Schätzfunktion festgelegt (aufsteigende Sortierung)
- **Besten-Suche:** Es wird jeweils der Pfad ausgewählt, der dem Ziel am nächsten zu sein scheint (geringste geschätzte Restkosten)

Informierte Suche (2)

Es existiert eine **optimistische** Funktion zur Schätzung der Restkosten bis zum Ziel vom aktuellen Knoten aus

Optimale Verfahren

- **Branch & Bound + Restkostenabschätzung:** Es wird jeweils der Pfad ausgewählt, bei dem die Summe der bisherigen Kosten und der geschätzten Restkosten minimal ist
- **A*:** Branch & Bound + Dynamische Programmierung + Restkostenabschätzung



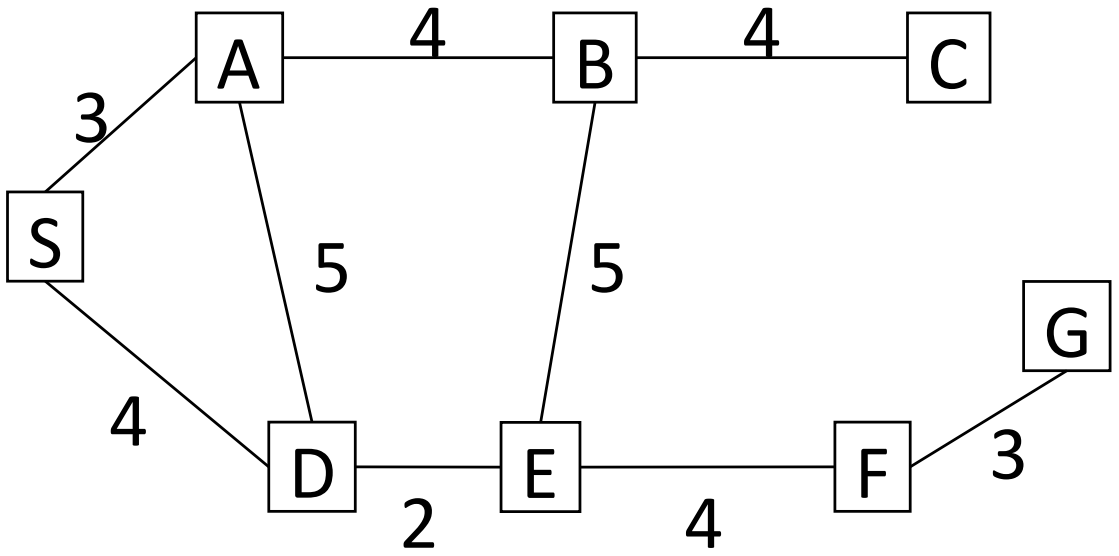
Karlsruhe Institute of Technology

AIFB

„Blinde Kuh“:
NICHT INFORMIERTE SUCHE

Aufgabe

- Wenden Sie die Breitensuche für den gezeigten Graphen an beginnend bei S!
  - ⇨ Geben Sie dazu die entstehende Pfadliste pro Suchschritt an

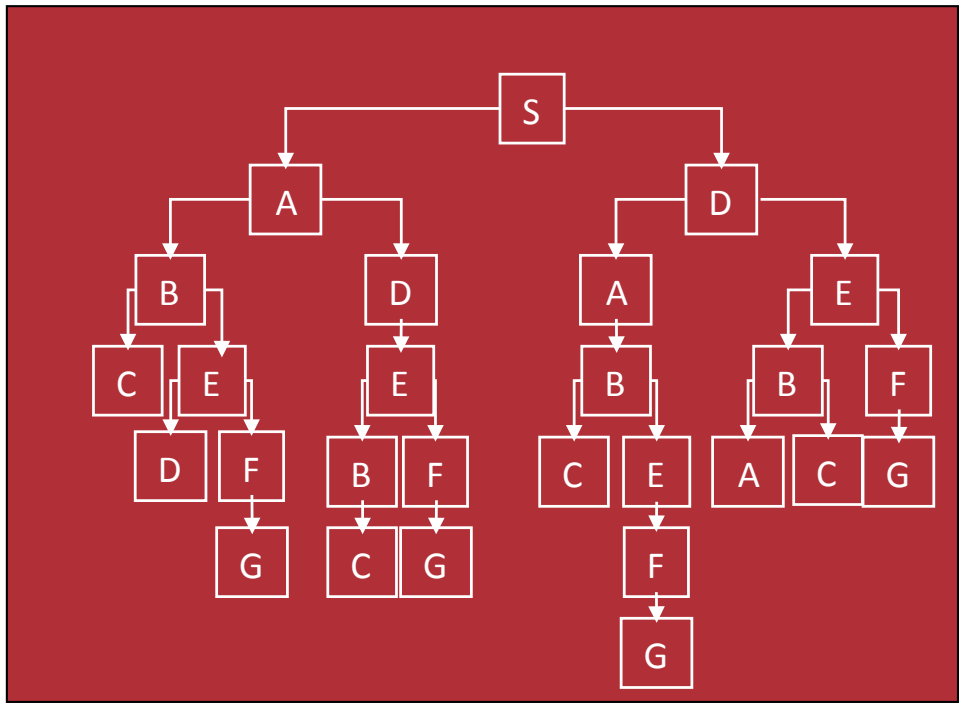


Breitensuche I

Schritt 1

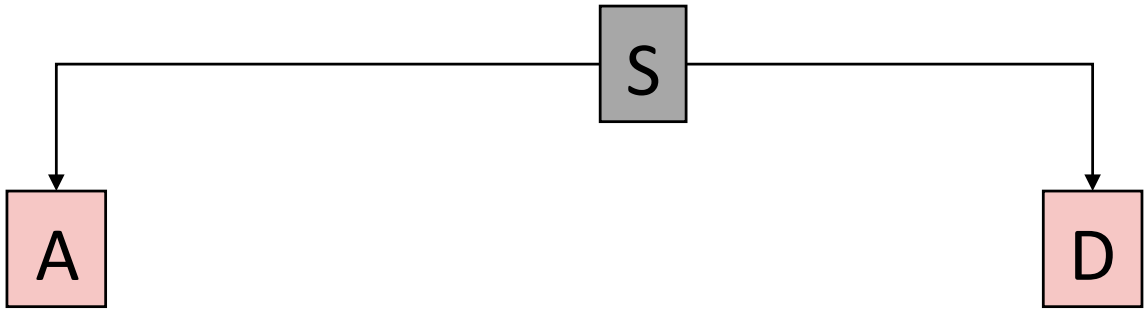
S

Liste:
{S}



Breitensuche II

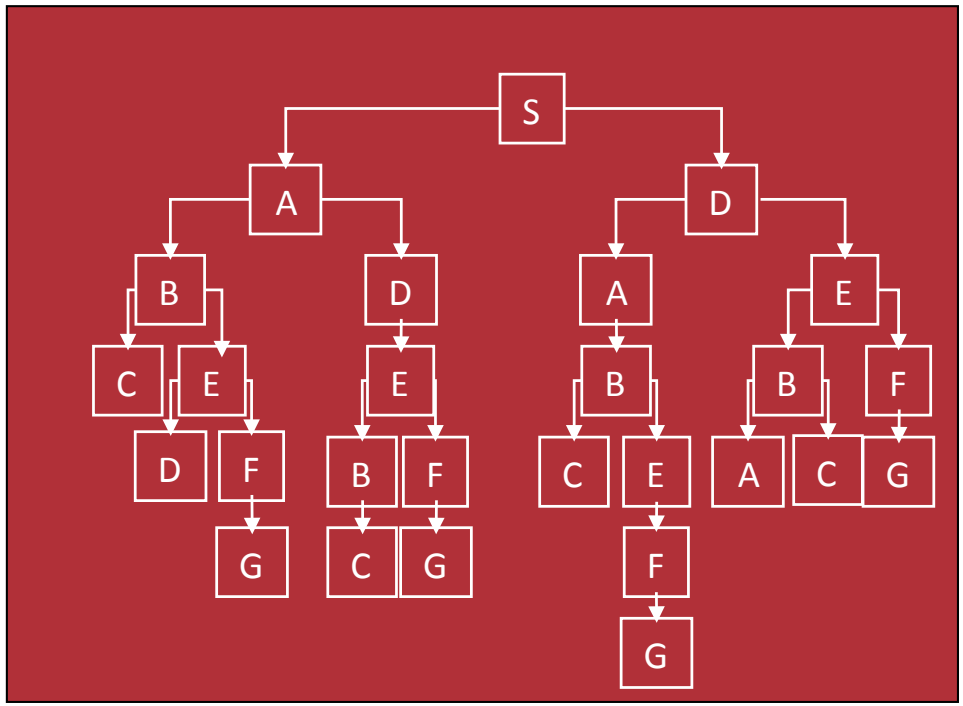
Schritt 2



Liste:

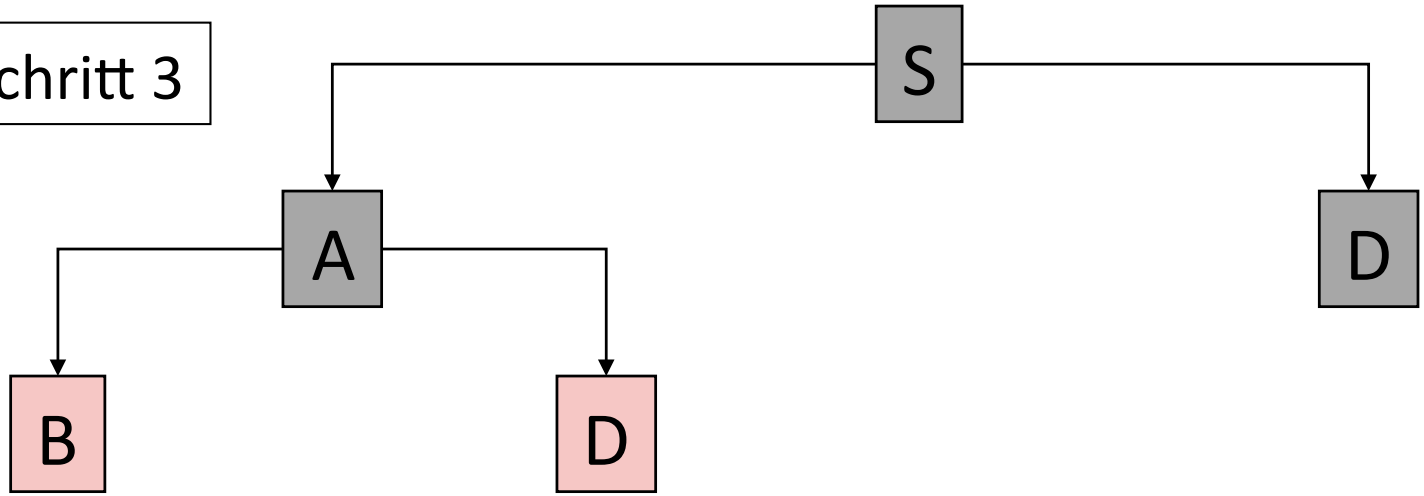
→{S,A}

{S,D}



Breitensuche III

Schritt 3

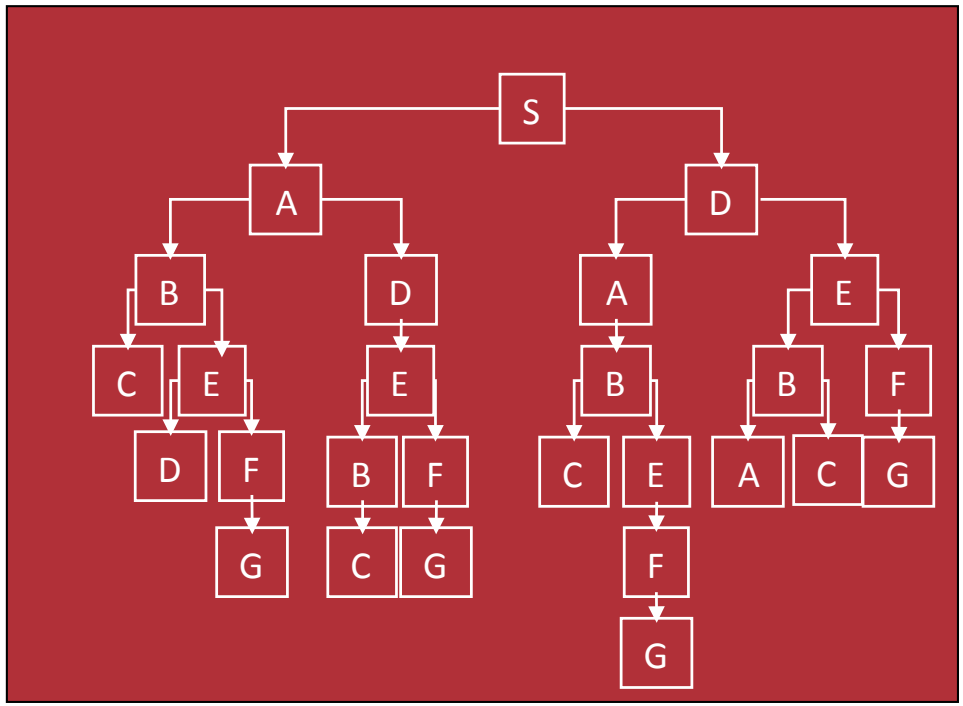


Liste:

→{S,D}

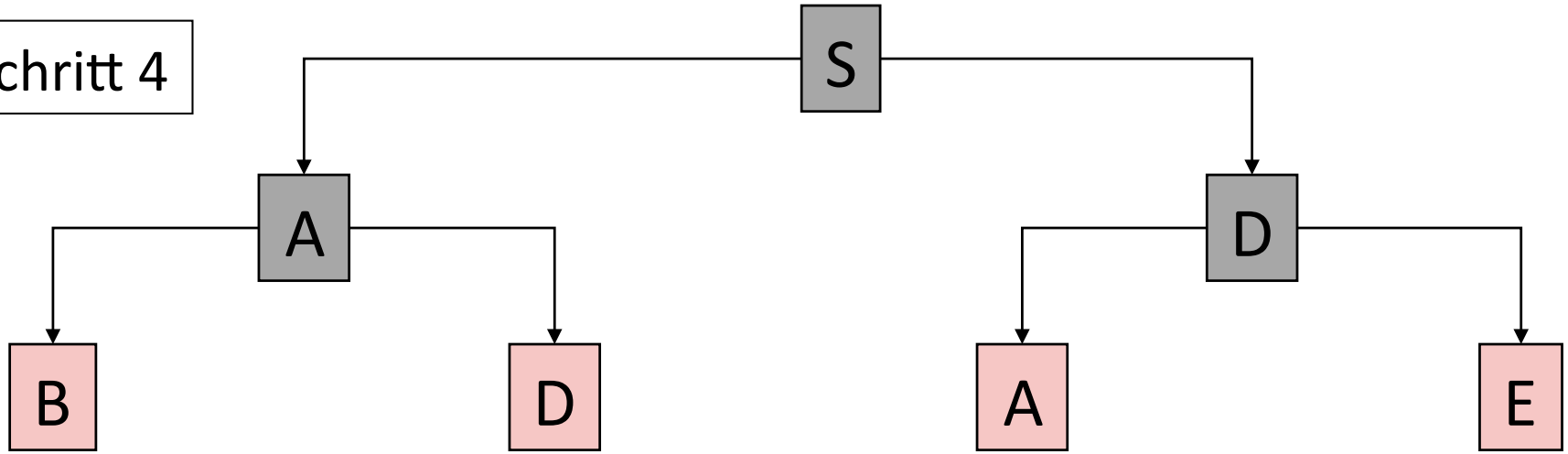
{S,A,B}

{S,A,D}



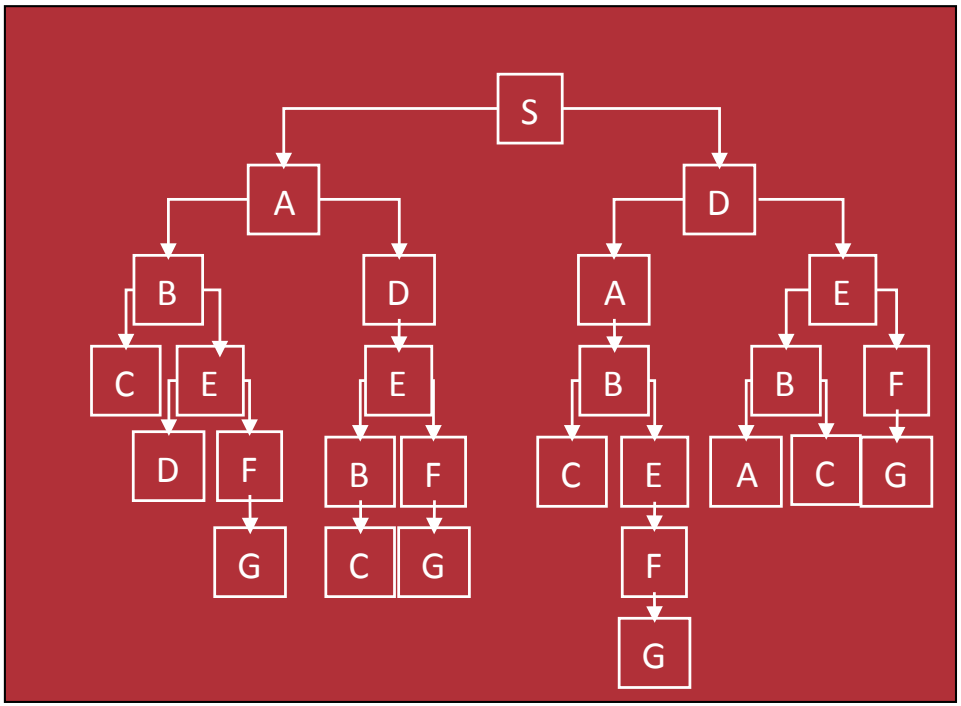
Breitensuche IV

Schritt 4



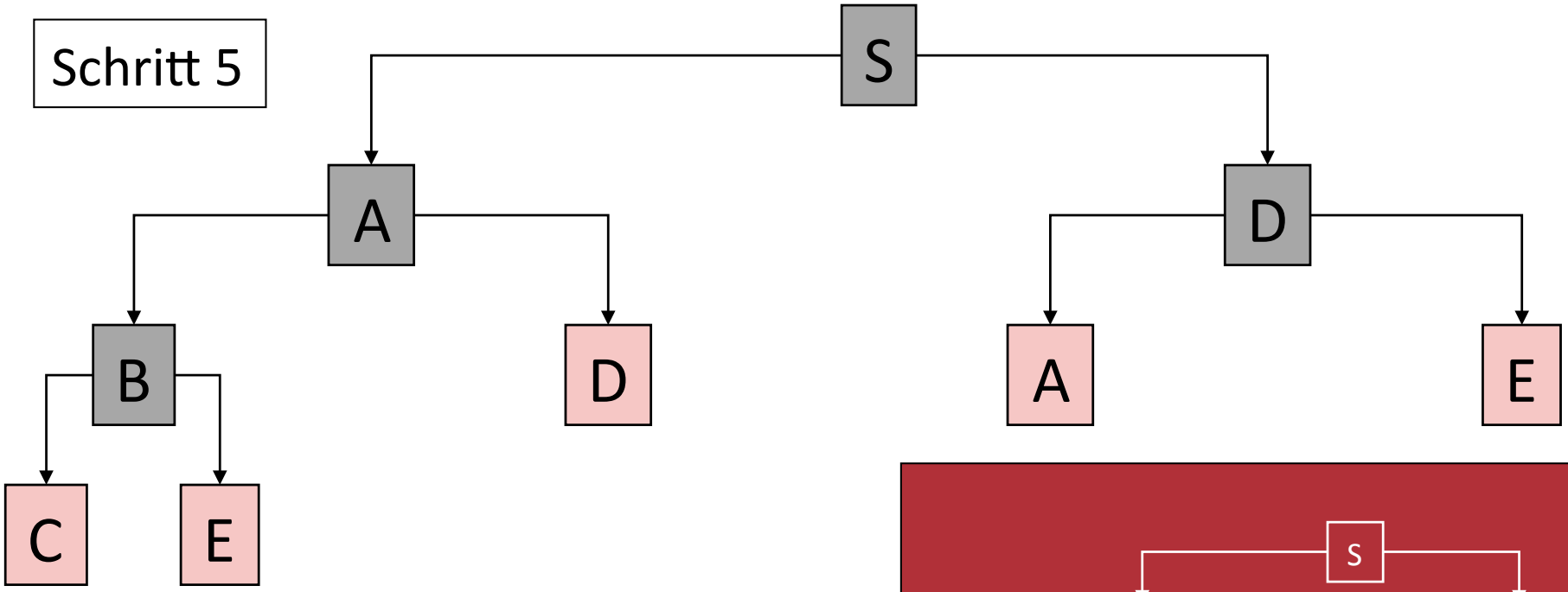
Liste:

{S,A,B}
{S,A,D}
{S,D,A}
{S,D,E}



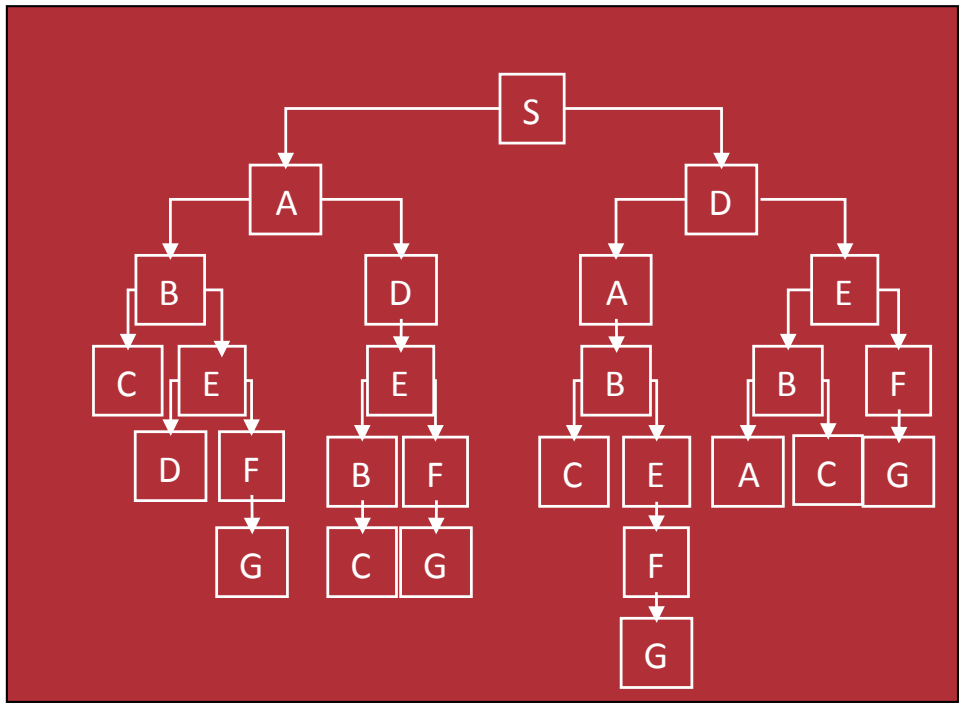
Breitensuche V

Schritt 5



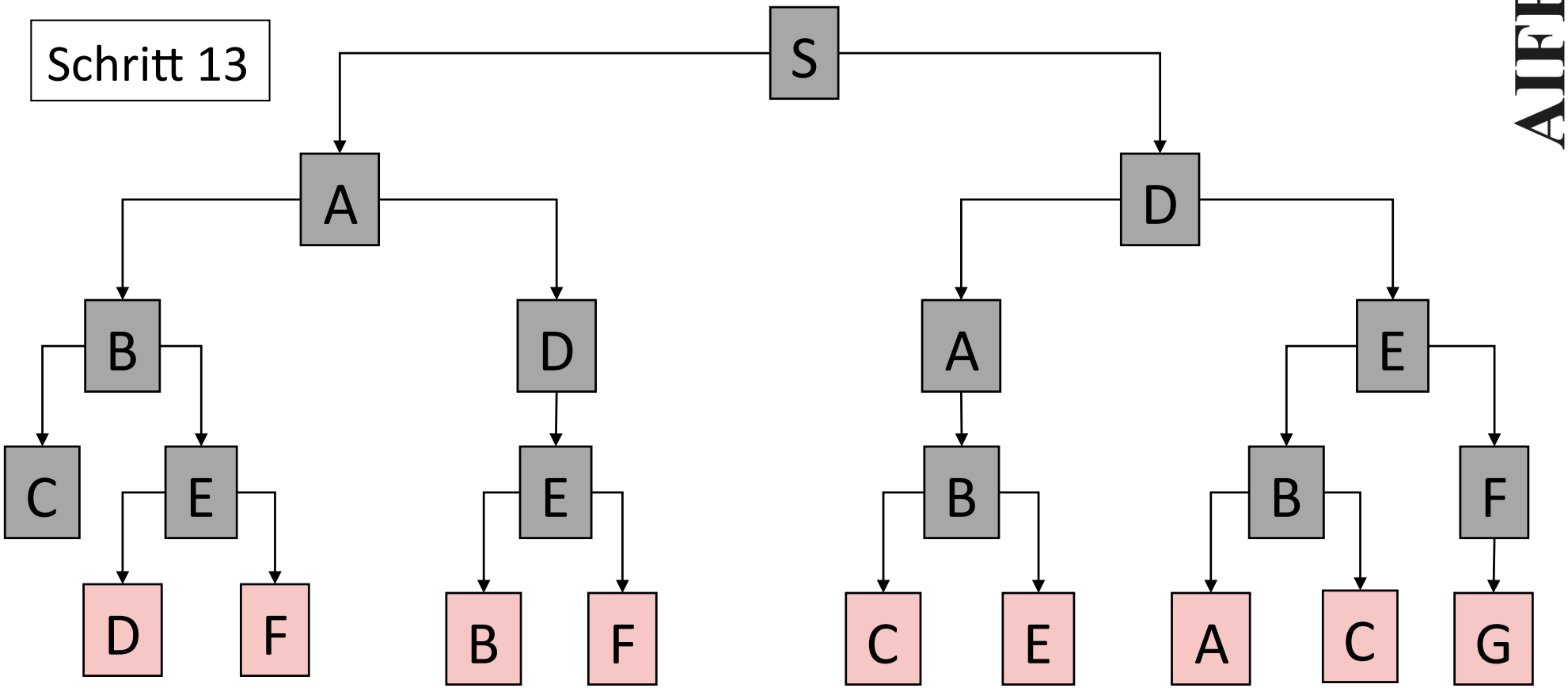
Liste:

→ {S,A,D}
{S,D,A}
{S,D,E}
{S,A,B,C}
{S,A,B,E}



Breitensuche VI

Schritt 13



Liste:

{S,A,B,E,D}

...

{S,D,E,F,G}

Breitensuche: Eigenschaften

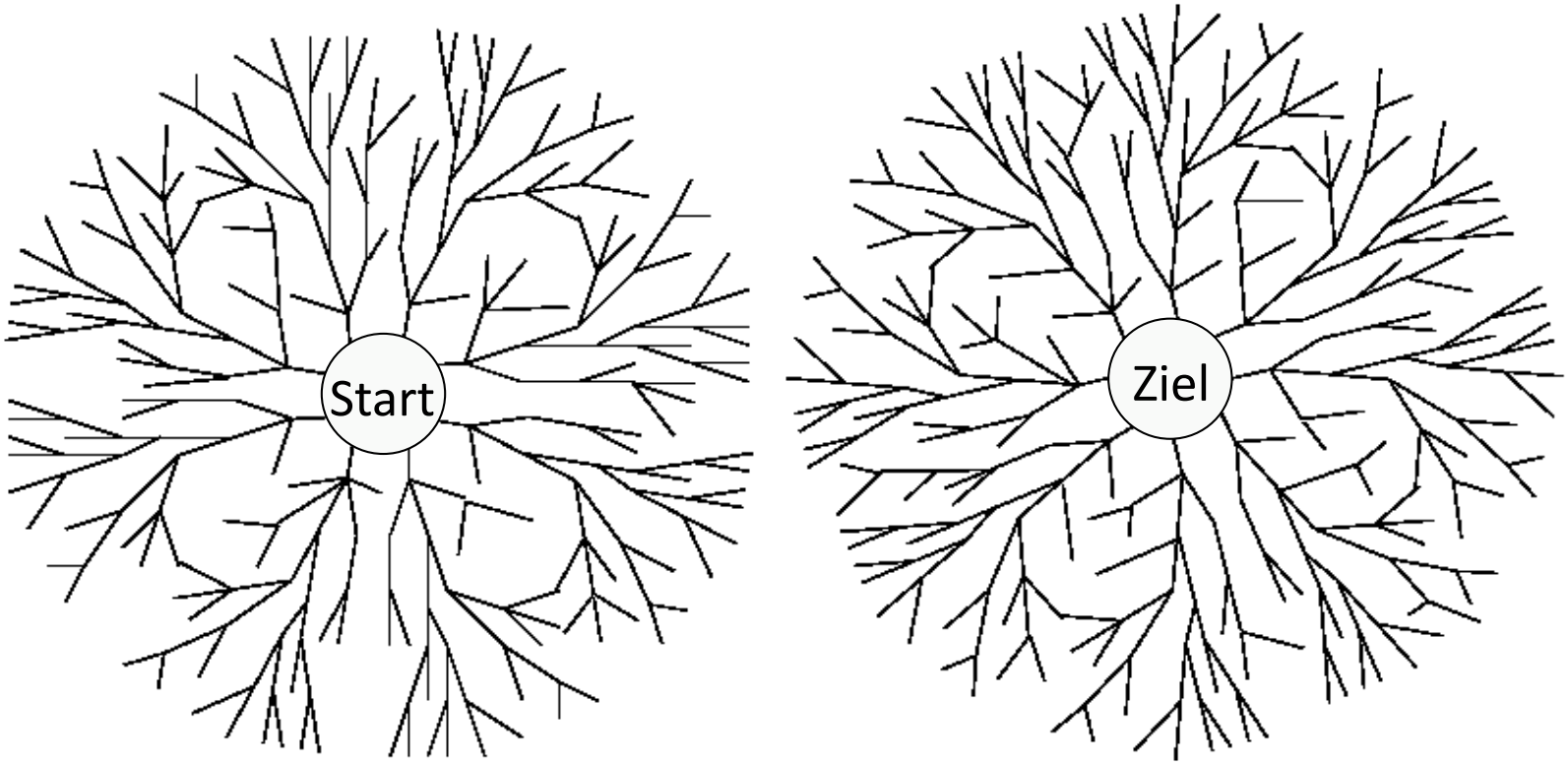
- vollständig, falls *b* endlich ist
- Zeitkomplexität: $1 + b + b^2 + b^3 + \ldots + b^m = O(b^m)$
- Speicherkomplexität ebenfalls $O(b^m)$
  - ⇨ Achtung: Pfadliste darf nicht explizit gespeichert werden
  - ⇨ Stattdessen: invertierter Suchbaum (also Suchbaum mit umgekehrter Kantenrichtung)
- optimal?
  - ⇨ ja, wenn Kosten = 1 pro Schritt
  - ⇨ nein, wenn Kanten unterschiedlich gewichtet sind

b: maximale Anzahl von Verzweigungen pro Knoten
m: maximale Baumtiefe

Strahlensuche

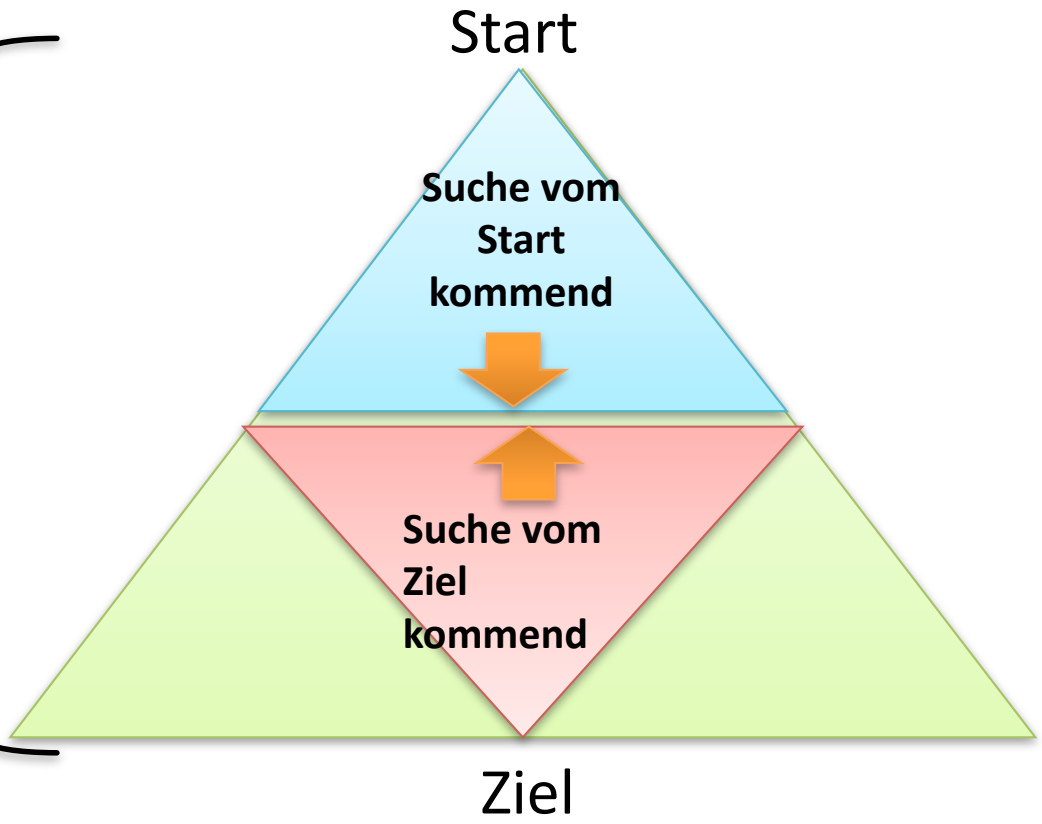
- Eine Variante der Breitensuche:
  - Anstatt alle Pfade in Suchtiefe *i* werden nur die *w* besten (also mit den bisher geringsten Kosten) expandiert
  - Zeit- und Speicheraufwand: $O(m \cdot w)$

Bidirektionale Suche (1)

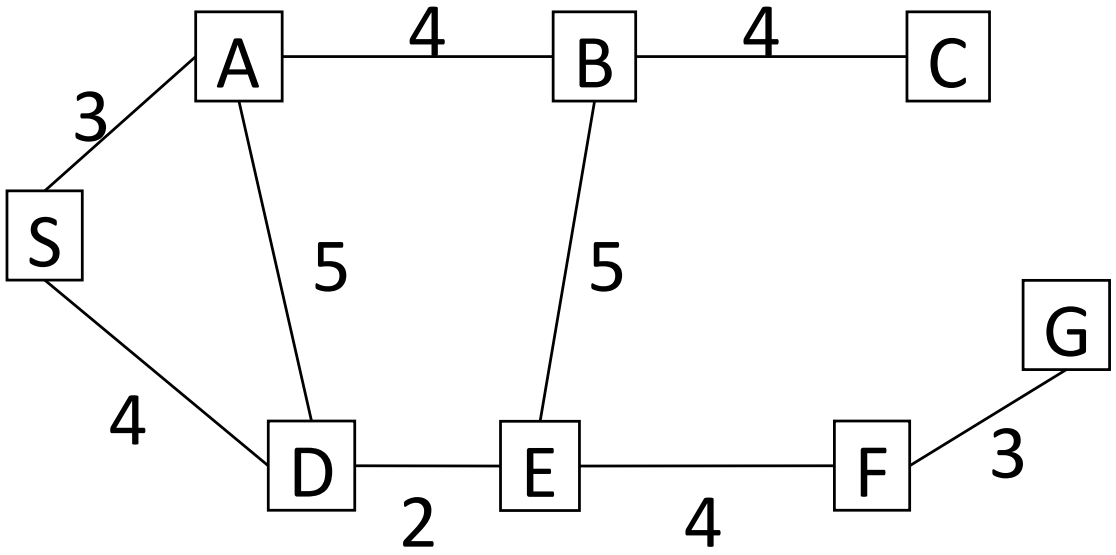


Bidirektionale Suche (2)

Suchbaum mit gewöhnlicher Breitensuche



Bidirektionale Suche: Beispiel



Schritt	Pfadliste von S kommend	Pfadliste von G kommend
1	S	G
2	SA, SD	GF
3	SAB, SAD, SD	GFE
4	SAB, SAD, SDA, SDE	

Bidirektionale Suche: Nutzen

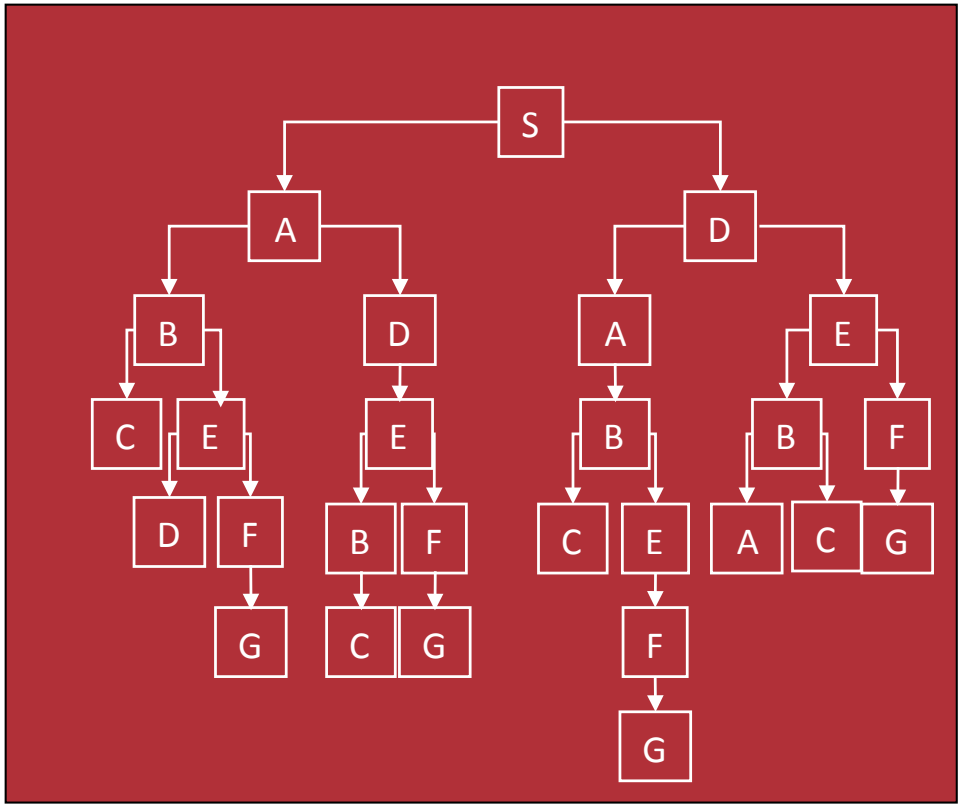
- Annahme: vom Ziel ebenso wie vom Start her suchend hat jeder Knoten ca. *b* Nachbarn
  - ⇨ Wenn Start- und Zielsuche „aufeinander treffen“, sind beide Teile der Suche nur bis Tiefe von *m*/2 vorgedrungen
  - ⇨ also liegt der Gesamtaufwand in $O(b^{m/2})$
- Problem: es muss in jedem Schritt getestet werden, ob die Endknoten der einen Pfadliste in der anderen Pfadliste enthalten sind. Mit geeigneter Datenstruktur kann man den Aufwand hierzu geeignet begrenzen.

- Eingeschränkte Anwendbarkeit der bidirektionalen Suche:
  - ⇨ der Zielknoten muss bekannt sein
  - ⇨ der Vorgänger eines Knotens muss sich vom Ziel her bestimmen lassen

Tiefensuche

Schritt 1

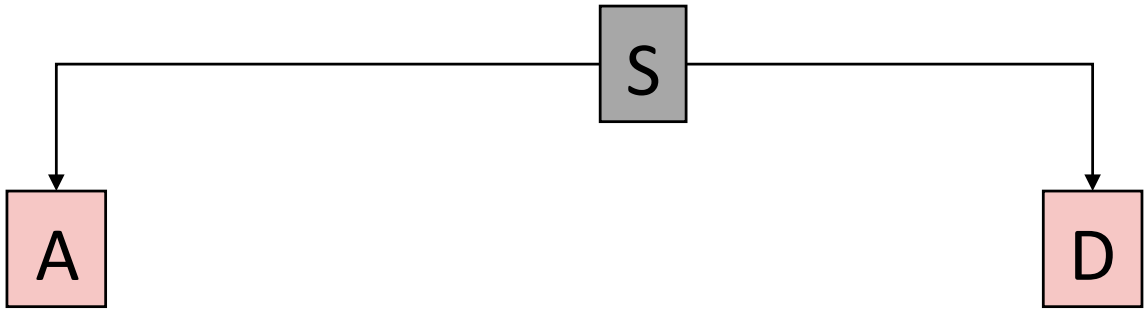
S



Liste:
{S}

Tiefensuche

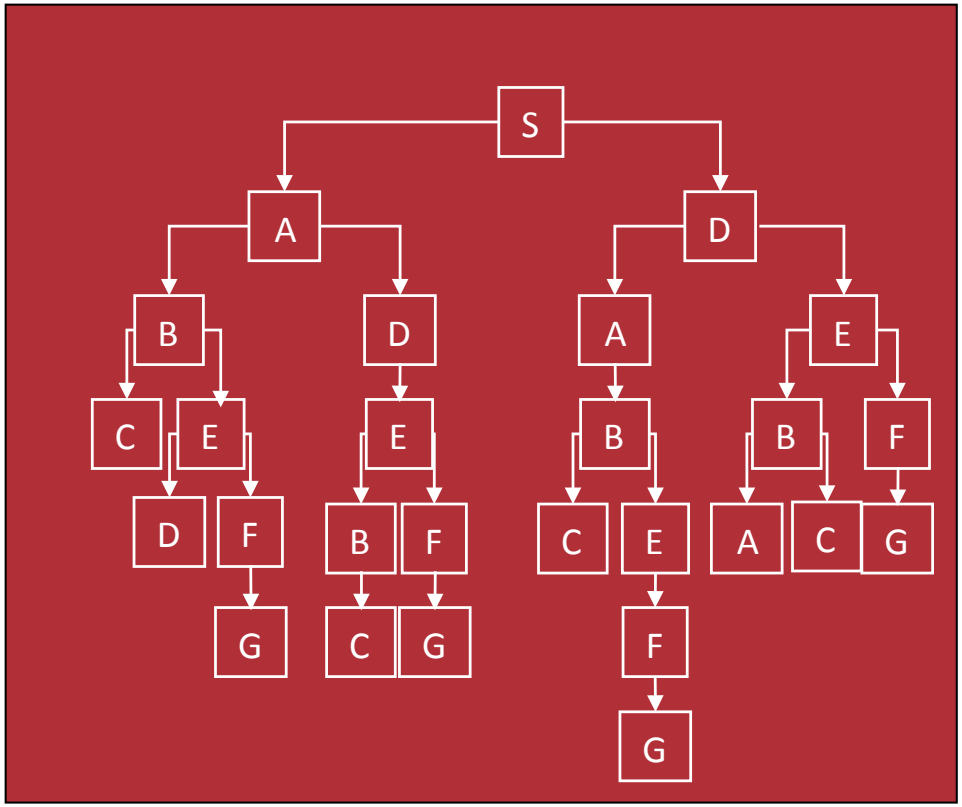
Schritt 2



Liste:

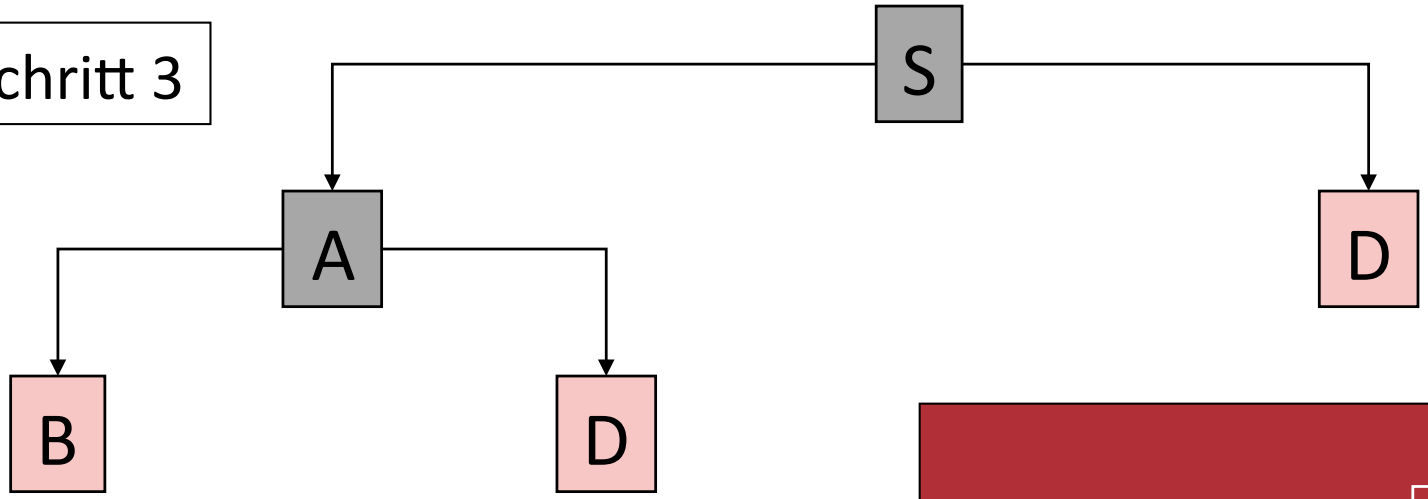
→{S,A}

{S,D}



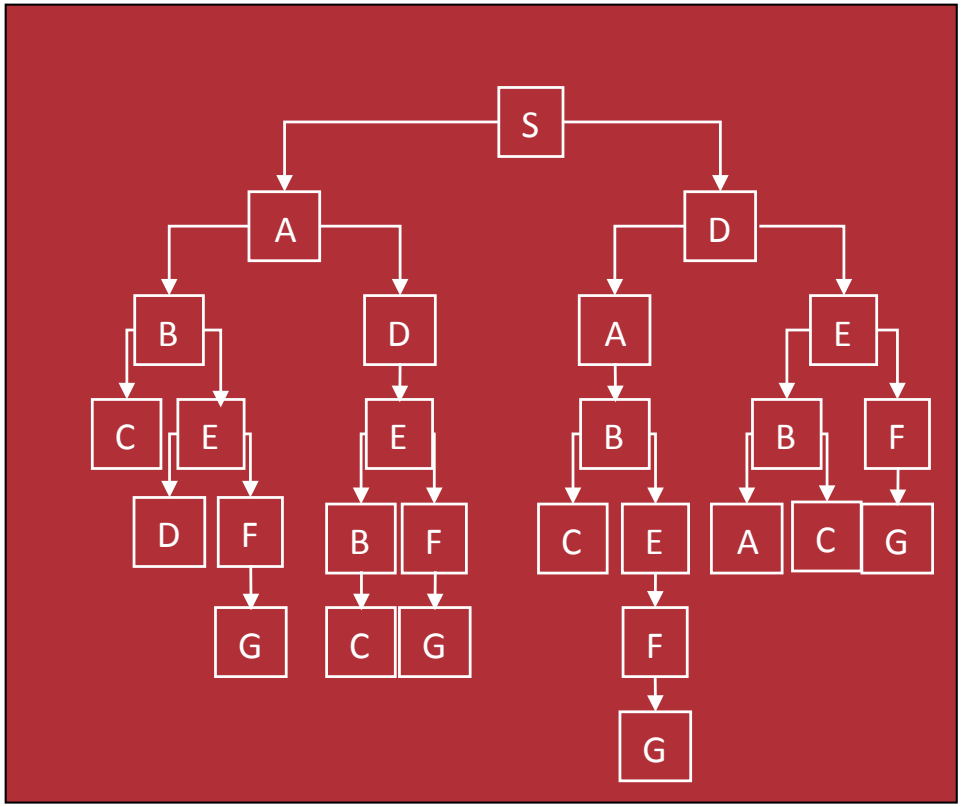
Tiefensuche

Schritt 3



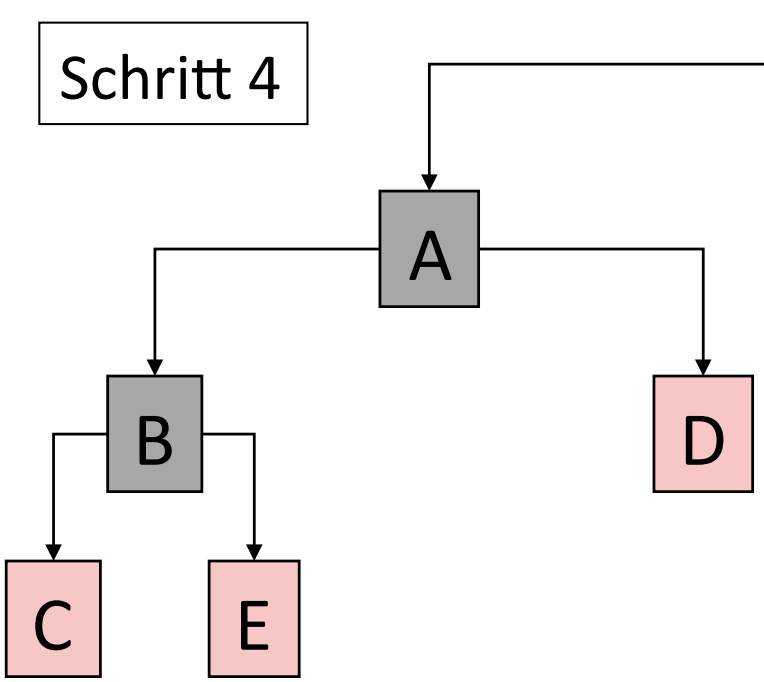
Liste:

{S,A,B}
{S,A,D}
{S,D}



Tiefensuche

Schritt 4



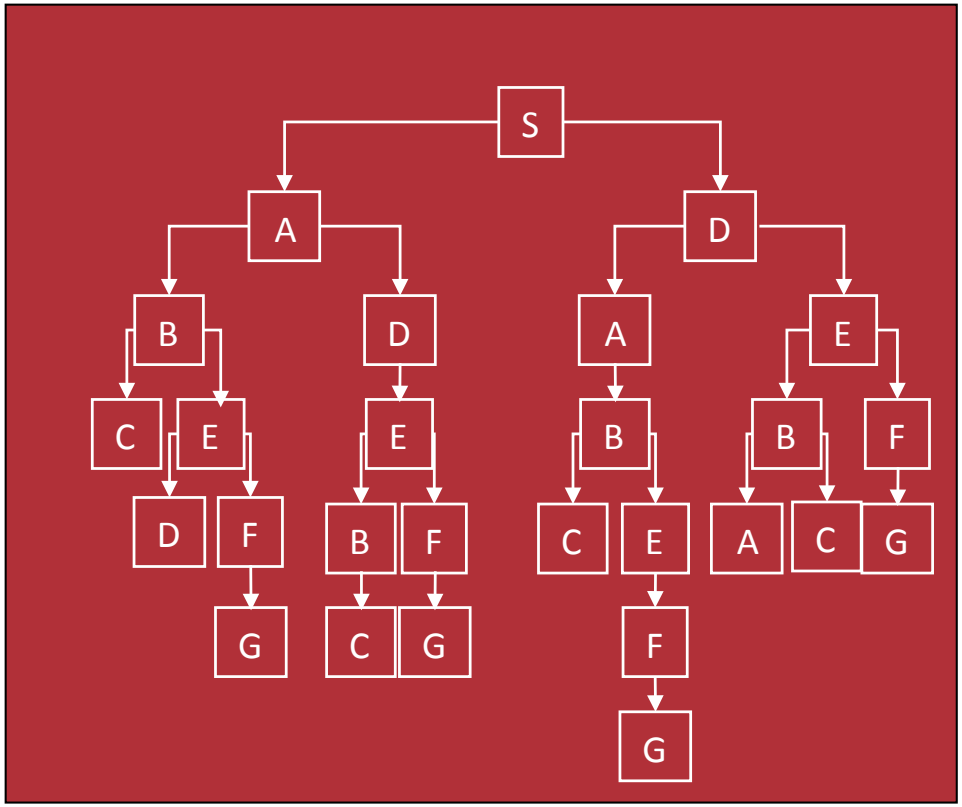
Liste:

{S,A,B,C}

{S,A,B,E}

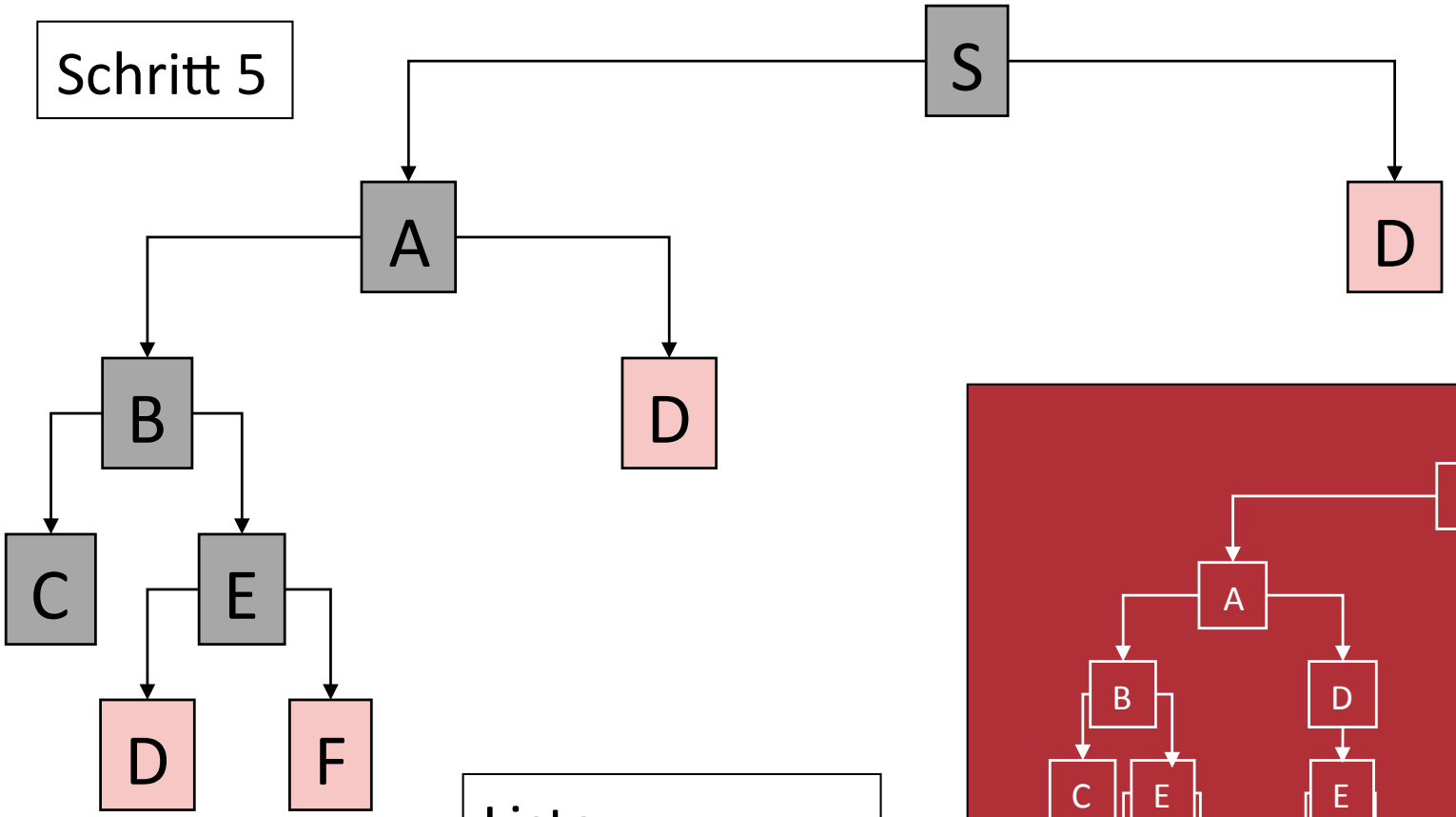
{S,A,D}

{S,D}



Tiefensuche

Schritt 5



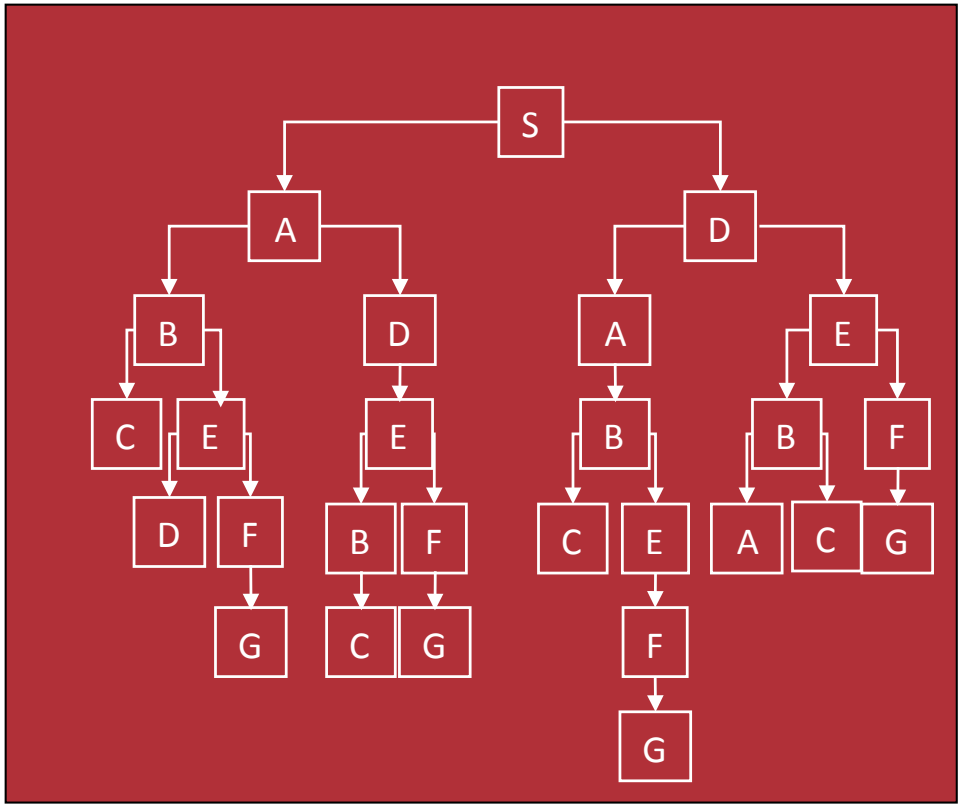
Liste:

→ {S,A,B,E,D}

→ {S,A,B,E,F}

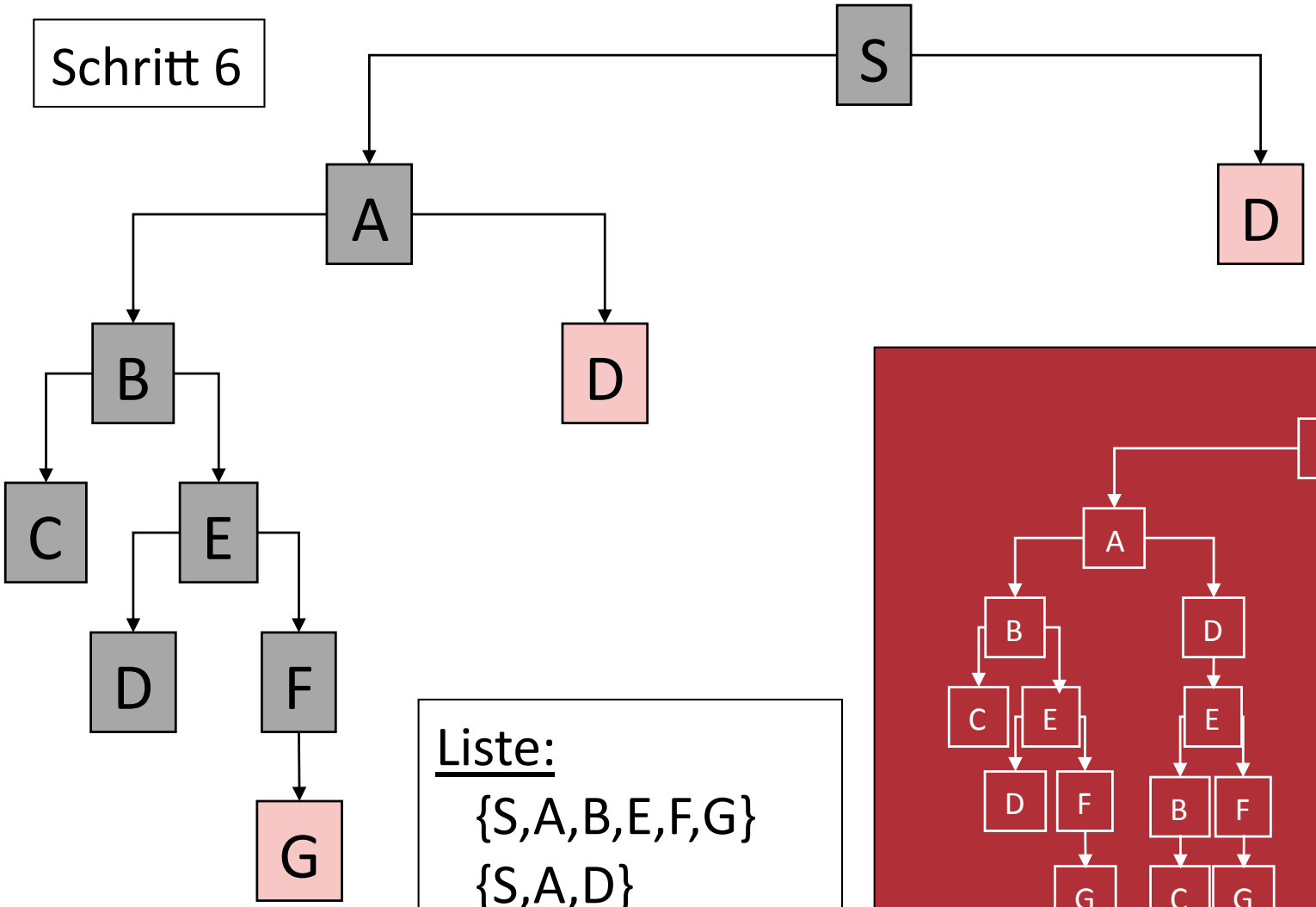
{S,A,D}

{S,D}



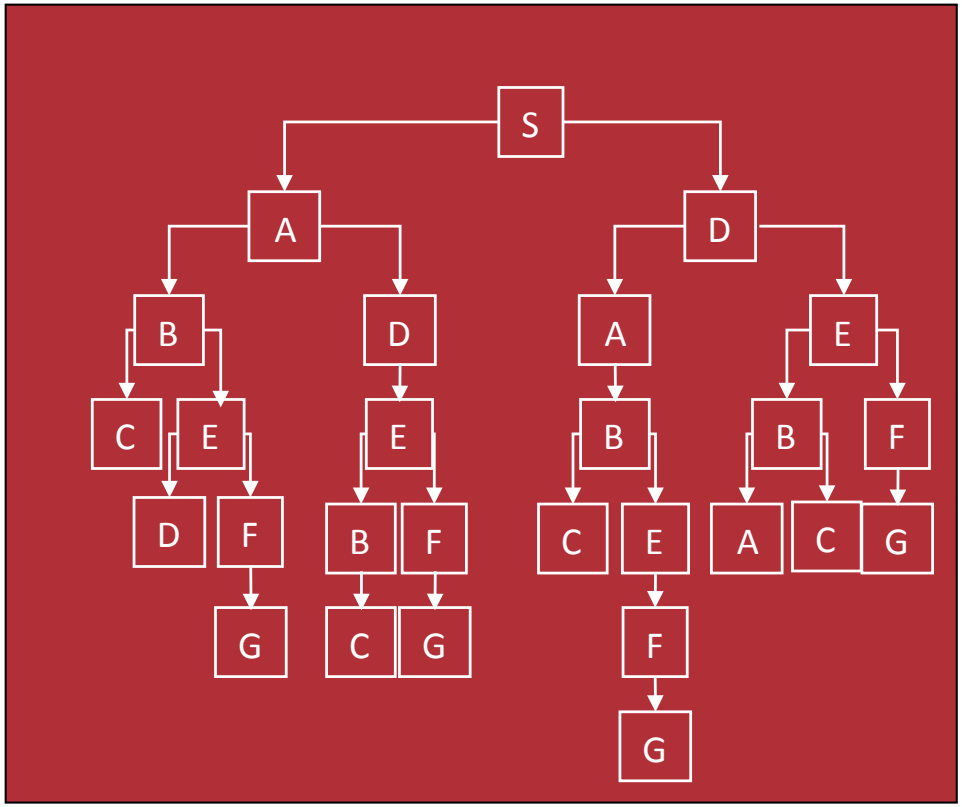
Tiefensuche

Schritt 6



Liste:

{S,A,B,E,F,G}
{S,A,D}
{S,D}



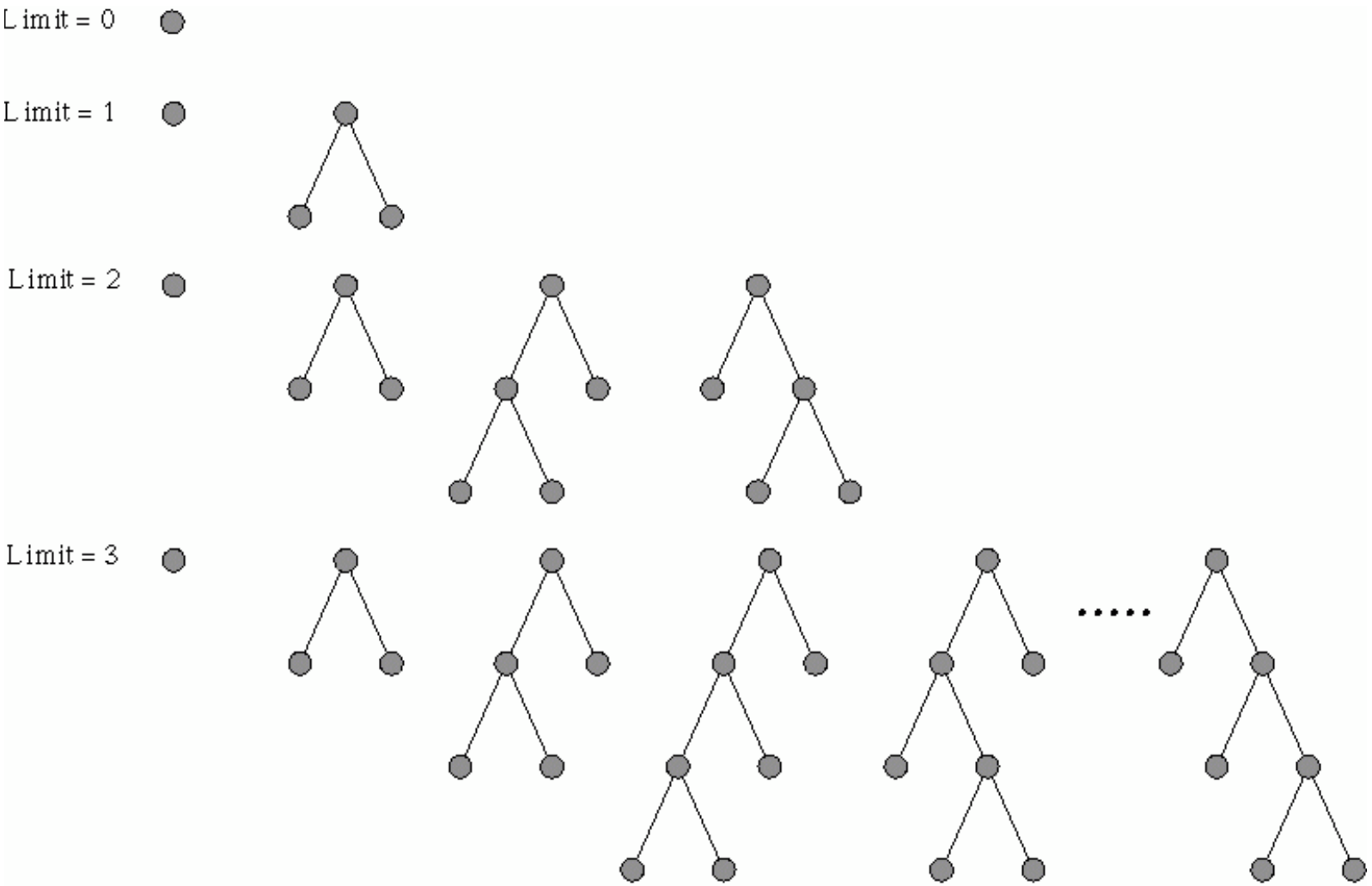
Tiefensuche: Eigenschaften

- vollständig in Zustandsräumen mit endlicher Tiefe
- nicht vollständig in Zustandsräumen mit unendlicher Tiefe
- Zeitkomplexität: $O(b^m)$
- Speicherkomplexität: $O(mb)$
  - ⇨ verkettete Liste bzw. einen Stapel (Stack) nutzen, um sich Pfad bis zur aktuellen Stelle zu merken, ab der die Tiefensuche fortgesetzt wird
- nicht optimal

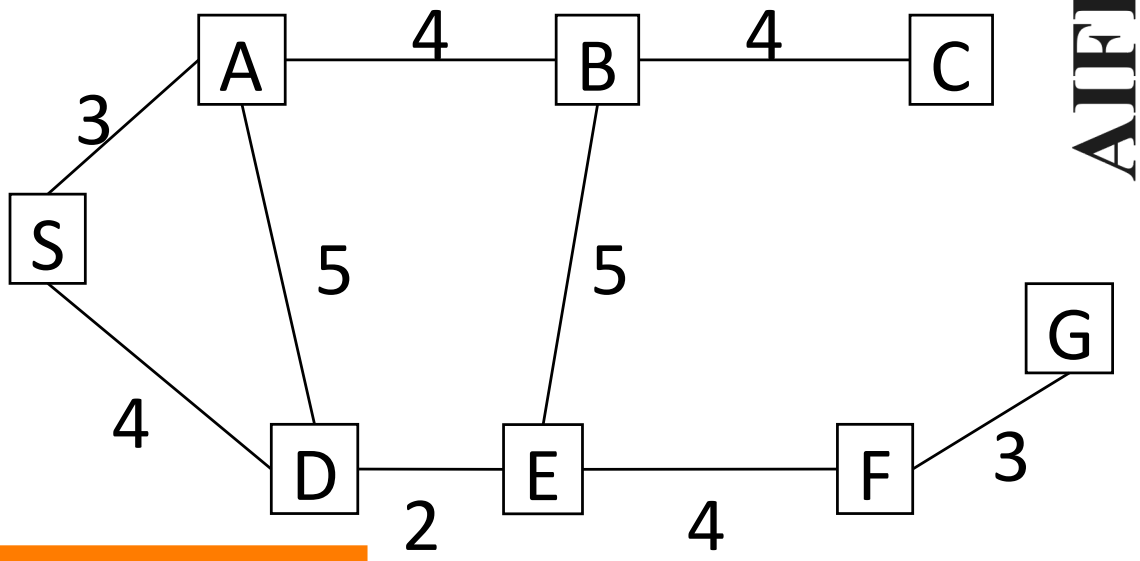
Varianten der Tiefensuche

- Annahme: Jeder Punkt im Zustandsraum ist von jedem anderen maximal *k* Schritte entfernt.
- Tiefen-Begrenzte Suche
  - ⇨ Maximale Tiefe *k* wird festgelegt
  - ⇨ Nachteil: nicht optimal und evtl. nicht vollständig
- Iterative Deepening
  - ⇨ wie Tiefen-begrenzte Suche, wobei die maximale Tiefe nacheinander auf 1, 2, 3, ... gesetzt wird
  - ⇨ Vorteil: vollständig, wesentlich geringerer Speicherverbrauch als Breitensuche
  - ⇨ viele Schritte werden mehrfach ausgeführt (das macht aber meistens nicht viel)

Iterative Deepening

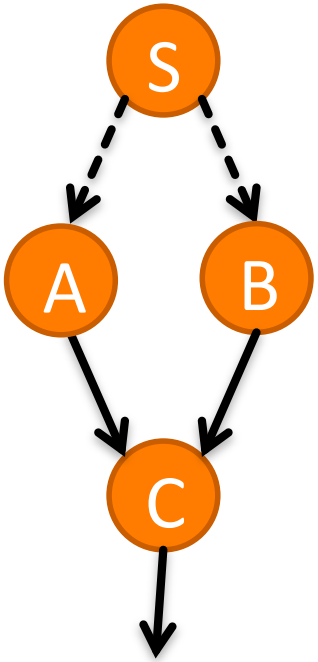


Iterative Deepening: Beispiel



k	Schritt	Pfadliste
1	1	S
1	2	SA, SD
2	1	S
2	2	SA, SD
2	3	SAD, SAB, SDA, SDE
3	1	S
3	2	…

Verringerung des Suchaufwands



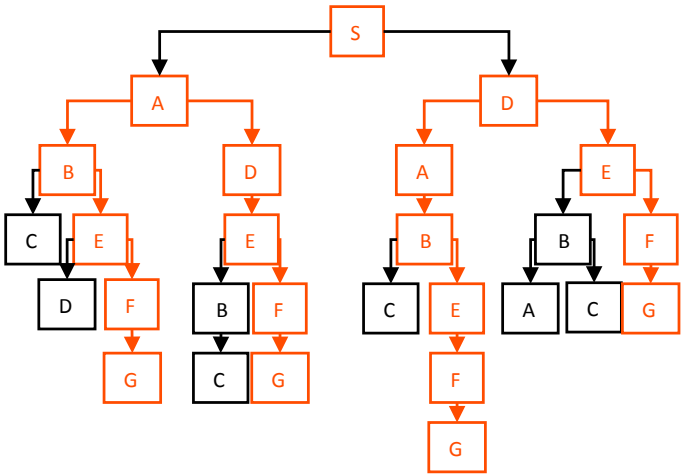
Vermeidung redundanter Suche durch Markierung

- Situation: Siehe nebenstehender Graph
- Die bisherigen Suchverfahren würden den Knoten C zweimal besuchen – einmal von A kommend und einmal von B kommend.
- Wenn man **nicht am optimalen Pfad interessiert** ist, kann man C beim ersten Besuch **markieren.**
- **Markierte Knoten werden beim nächsten Besuch nicht weiter verfolgt.**

Britische Museums-Methode

- Erzeugung aller Pfade und Vergleich der Kosten der erfolgreichen Pfade (Lösungsmenge)
  - ⇨ Kann mit Tiefen- oder Breitensuche geschehen:

    Die Suche stoppt nicht beim ersten Erreichen des Ziels, sondern macht einfach weiter bis der ganze Suchbaum durchforstet ist
  - ⇨ Optimaler Pfad wird offensichtlich (auch) gefunden
- **Aufwand**: Wenn jeder Knoten *b* Nachfolger hat und der Baum die Tiefe *m*, dann gibt es b^m verschiedene Pfade (die nicht unbedingt alle zum Ziel führen)



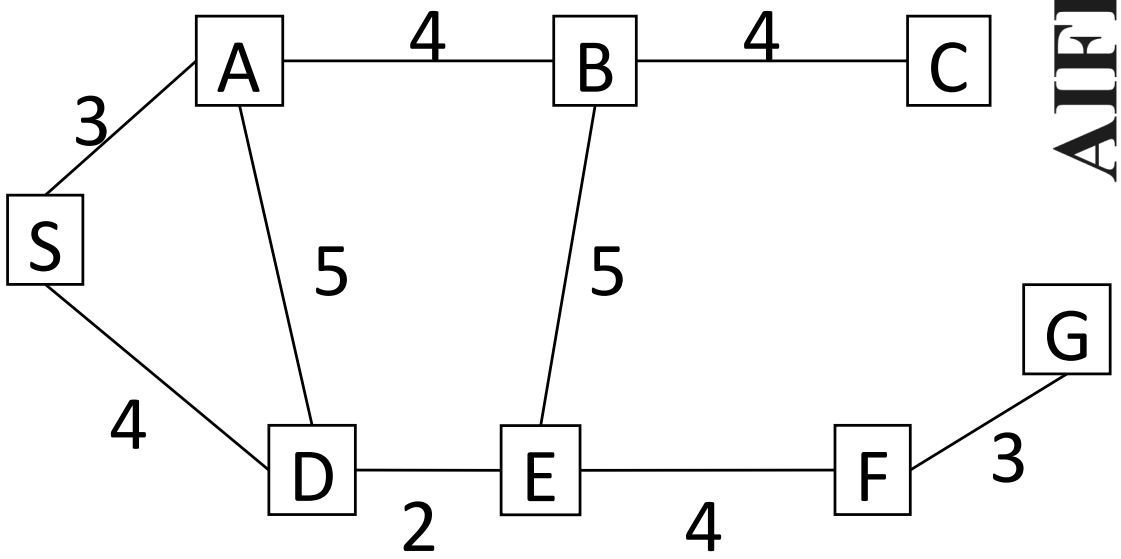
10 Pfade,
davon 4 erfolgreich

Branch & Bound

Ansatz :

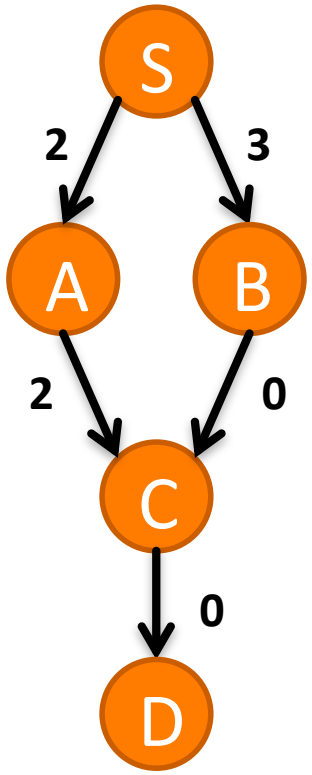
- Es wird jeweils der Pfad erweitert, der bisher die geringsten Kosten verursacht hat
- Ist ein erfolgreicher (d.h. vollständiger) Pfad Start → Ziel gefunden, dann
  - ⇨ erweitere alle weiteren Pfade
  - ⇨ und zwar solange, bis der kürzeste partielle Pfad mit höheren Kosten verbunden ist als der vollständige Pfad

Branch & Bound: Beispiel



Schritt	Pfadliste
1	S/0
2	SA/3, SD/4
3	SAD/8, SAB/7, SD/4
4	SAD/8, SAB/7, SDA/9, SDE/6
5	SAD/8, SAB/7, SDA/9, SDEB/11, SDEF/10
6	SABC/11, SABE/12, SDA/9, SDEB/11, SDEF/10

Branch & Bound + Dynamische Programmierung (1)

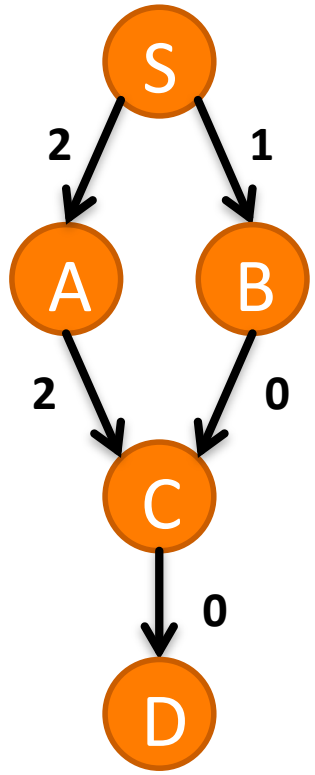


Wieder: Vermeidung redundanter Suche

- Situation siehe Graph:
  - ⇨ Branch & Bound hat zuerst teuren Weg von S nach C entdeckt (nämlich SAC)
  - ⇨ Weil SBC billiger ist, wird C zunächst **nicht** expandiert

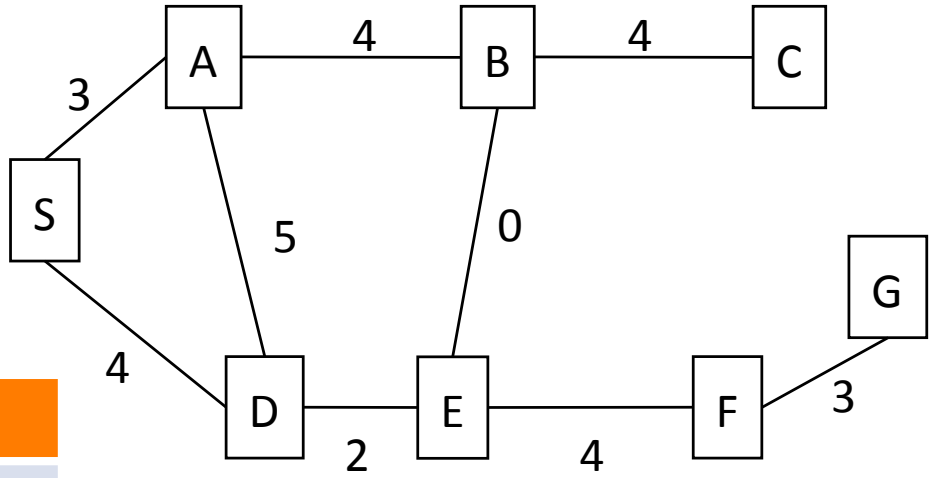
    Das ist immer so bei teuren und billigen Pfaden zu einem Knoten X!
- Idee: Sobald B&B einen billigeren Weg von S nach X entdeckt, kann es den teureren aus der Pfadliste streichen
  - ⇨ Der teure Pfad ist garantiert noch in der Pfadliste, weil X noch nicht expandiert wurde

Branch & Bound + Dynamische Programmierung (2)



- Alternative Situation siehe Graph:
  - ⇨ Branch & Bound entdeckt nun zuerst den billigen Pfad von S nach C (nämlich SBC)
  - ⇨ B&B expandiert als nächstes C und ersetzt SBC durch SBCD in der Pfadliste
- Wenn SAC entsteht, möchte man es aus der Pfadliste streichen
  - ⇨ Dazu kann man wieder das Markierungsverfahren von zuvor verwenden
  - ⇨ C wurde dann bereits markiert, als SBC entdeckt wurde

B&B + dyn. Prog.: Beispiel



Schritt	Pfadliste (Pfad/Bisherige Kosten)
1	S/0
2	SA/3, SD/4
3	SAB/7, SD/4
4	SDE/6, SAB/7
5	SDEB/6, SDEF/10
6	SDEBC/10, SDEF/10
7	SDEFG/13

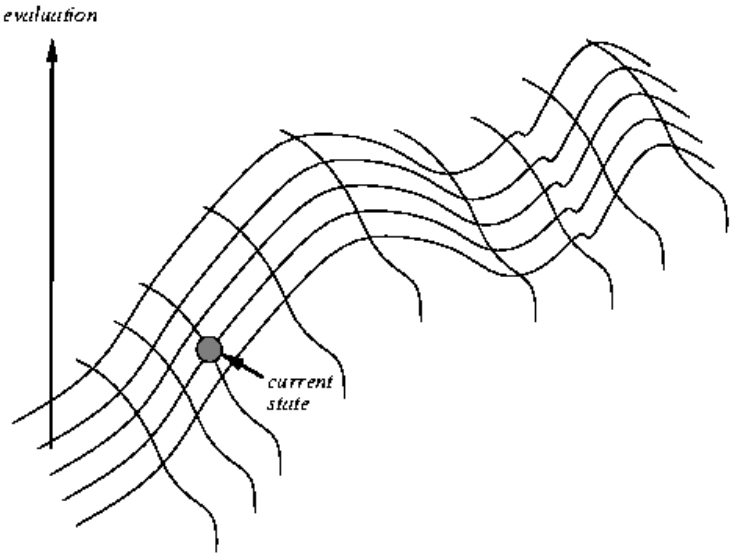
Karlsruhe Institute of Technology

AIFB

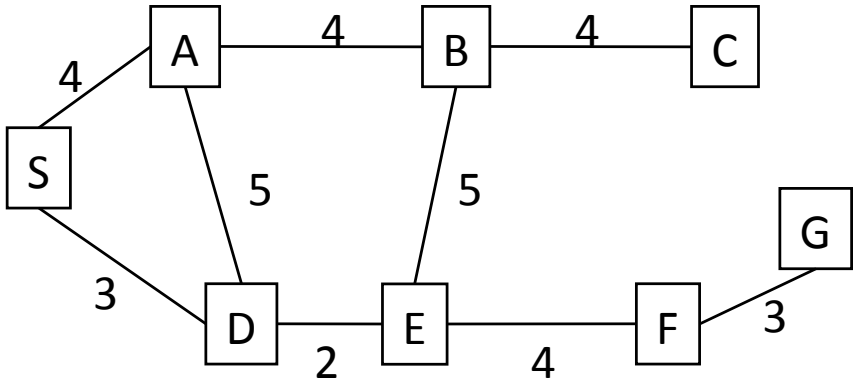
„Topfschlagen“: INFORMIERTE SUCHE

Hill-Climbing (Bergsteigen)

- Spezielle Tiefensuche: es wird vom aktuellen Knoten aus der (noch nicht besuchte) Knoten mit den geringsten geschätzten Restkosten verfolgt
- Bei kontinuierlichem Suchraum: Suchrichtung entlang des Gradienten der Suchfunktion



Aufgabe



Abschätzfunktion: kürzeste Anzahl Kanten vom aktuell betrachteten Pfadende bis zum Ziel G

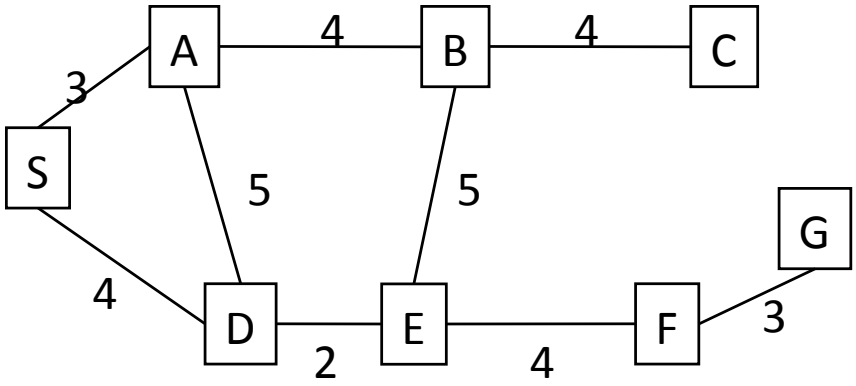
Schritt	Pfadliste (Pfad/Restkosten)
1	S/0
2	SA/4, SD/3
3	SA/4, SDA/4, SDE/2
4	SA/4, SDA/4, SDEB/3, SDEF/1
5	SA/4, SDA/4, SDEB/3, SDEFG/0

Branch & Bound + Restkostenabschätzung

Ansatz:

- Erweitere immer den Pfad aus der Pfadliste, dessen „**geschätzte Gesamtkosten**“ (d.h. die Summe aus den bisher entstandenen Kosten und den geschätzten, noch benötigten Restkosten) minimal sind.
- Forderung: die Restkostenabschätzung muss stets optimistisch sein (also: geschätzte Restkosten $\leq$ tatsächliche Restkosten).
- Zur Garantie der Optimalität:
  - ⇨ Das Verfahren endet erst, wenn das Ziel erreicht wurde **und** kein Pfad mit geringeren geschätzten Gesamtkosten mehr existiert

B&B mit Restkosten: Beispiel



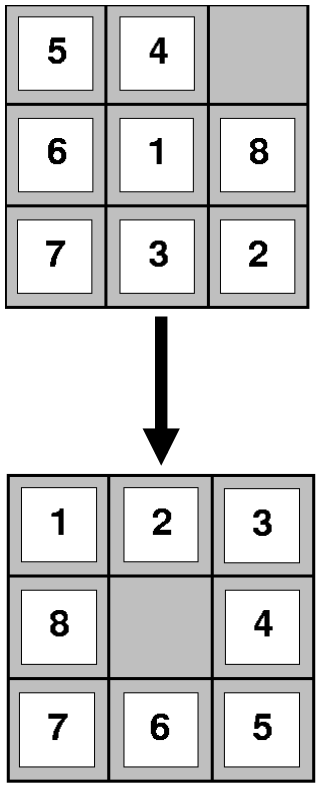
Abschätzfunktion: kürzeste Anzahl Kanten vom aktuell betrachteten Pfadende bis zum Ziel G

Schritt	Pfadliste (Pfad/bisherige Kosten/Restkosten)
1	S/0/4
2	SA/3/4, SD/4/3
3	SAD/8/3, SAB/7/3, SD/4/3
4	SAD/8/3, SAB/7/3, SDA/9/4, SDE/6/2
5	SAD/8/3, SAB/7/3, SDA/9/4, SDEF/10/1, SDEB/11/3

...

Beispiele für Restkostenfktn.

- Welche Restkostenfunktion sollte man wählen, um nach der kürzesten Strassenverbindung zwischen Städten zu suchen?
  - ⇨ Den euklidischen Abstand der Städte (also die Länge der Verbindungsgeraden
- Welche Restostenabschätzung(en) bietet sich für das 8-Puzzle-Problem von vorhin an?
  - ⇨ Bedenken Sie, dass die Abschätzungen optimistisch sein müssen
  - ⇨ Anzahl nicht korrekt platzierter Teile
  - ⇨ Manhattan Distanz: Summe der Entfernungen (in Anzahl der Quadrate) aller Teile von ihrem Zielstandort



A*

A* = Branch & Bound + Dynamische Programmierung + Restkostenabschätzung

- Offensichtlich: Die zusätzliche Restkostenabschätzung hat keinen negativen Einfluss auf das ursprüngliche „B&B + Dynamische Programmierung“
- A* ist eines der wichtigste Suchverfahren
  - Z.B. für Routenberechnungsprogramme
  - Fahrplanauskunft usw.

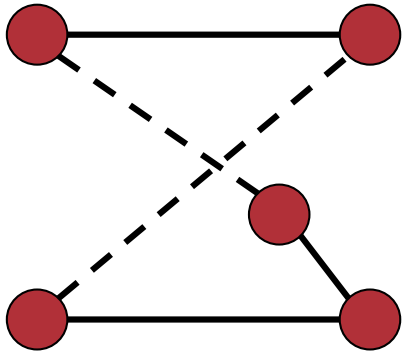
ITERATIVE SUCHE

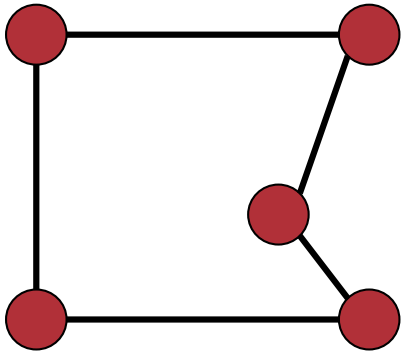
Iterative Verbesserung

- Algorithmen, die mit einer beliebigen Lösung beginnen und diese schrittweise verbessern
  - ⇨ wenn nur die Lösung interessiert, nicht jedoch der Pfad
  - ⇨ wenn die Effizienz des Problemlösungsverfahrens wichtiger ist als die Optimalität
- Hill-Climbing / Gradientenabstiegs-Verfahren
- Simulated Annealing
- Genetische Algorithmen

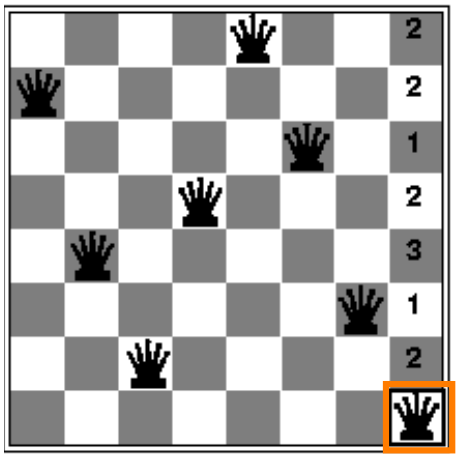
Beispiele

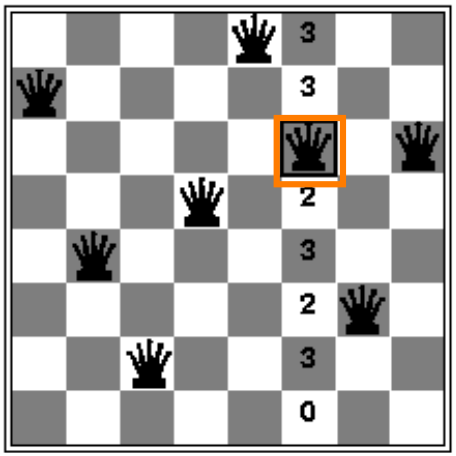
- TSP

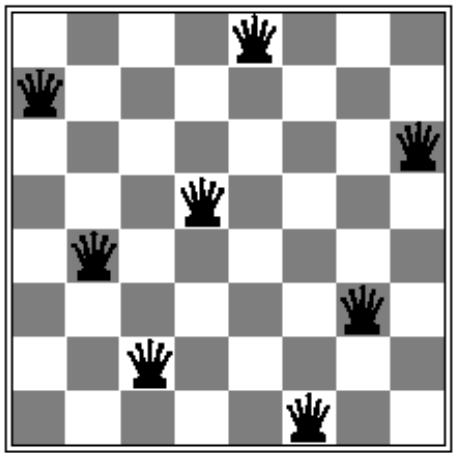




- 8-Damen-Problem







Karlsruhe Institute of Technology

AIFB

SPIELE

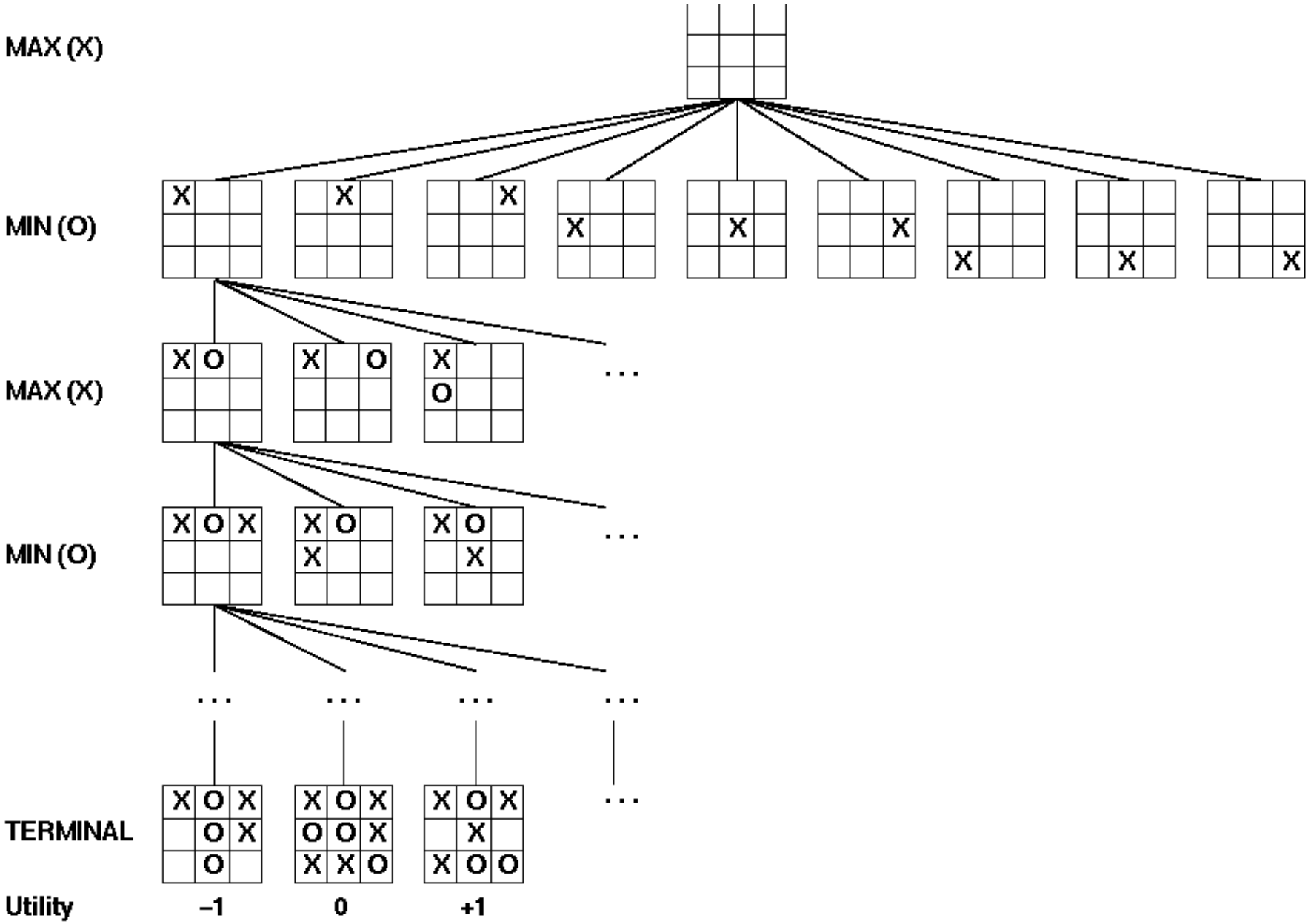
Spielstrategien

- Spiele mit zwei Spielern:
  - ohne Zufallskomponente, z.B. Schach, Dame
  - mit Zufallskomponente, z.B. Backgammon, Monopoly
- Optimale Strategie:
  - Minimax-Algorithmus
  - Aufwand i.d.R. zu hoch
- Begrenzung der Suchtiefe
  - $\alpha\beta$-Pruning

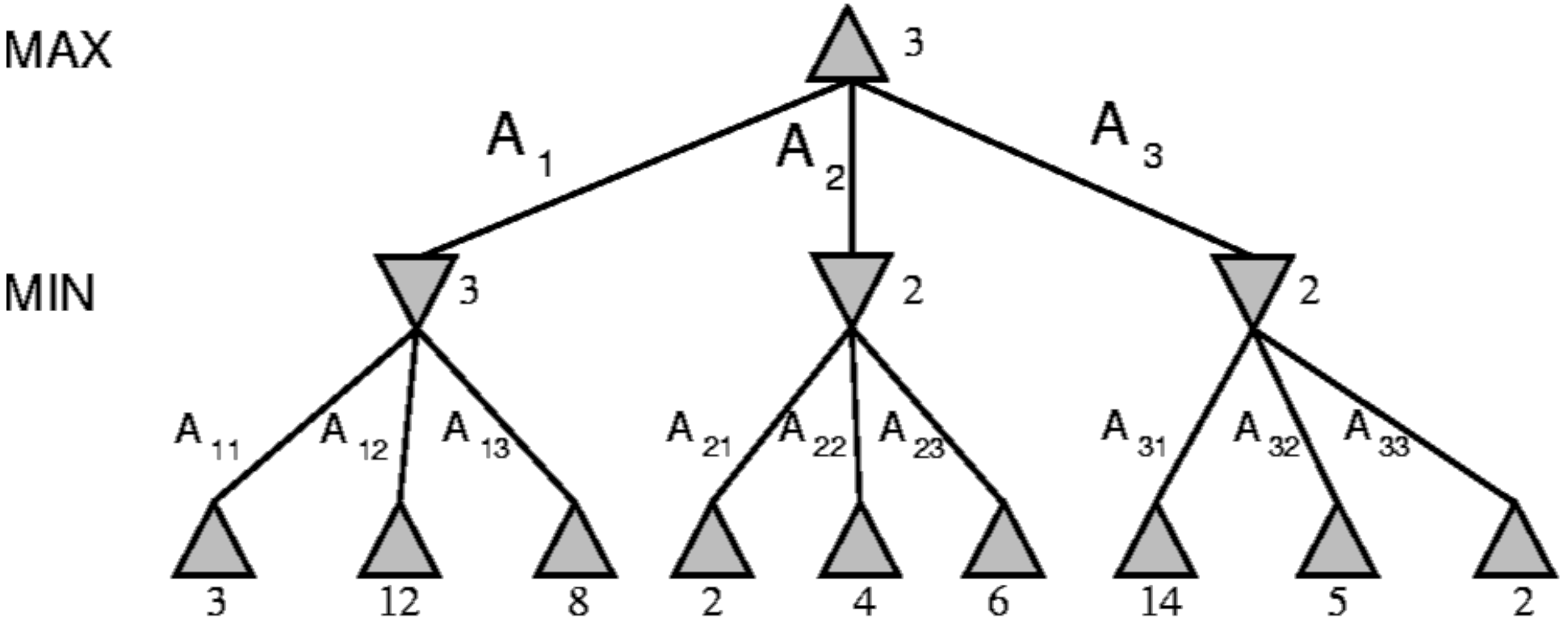
Minimax

- **Idee**: Maximierung des Gewinns von Spieler A unter der Berücksichtigung, dass Spieler B versucht, den Gewinn von A zu minimieren.
- **Algorithmus**:
  - Erzeugung des gesamten Suchbaums (bis zu Terminal-Zuständen)
  - Berechnung des Nützlichkeitswertes für jeden Terminal-Zustand
  - Für jede Ebene in dem Baum: Berechnung der Nützlichkeitswerte der Knoten unter Berücksichtigung der jeweiligen Nachfolgeknoten wobei
    - MAX am Zug: Maximum der Nachfolger
    - MIN am Zug: Minimum der Nachfolger
  - An der Wurzel wird der Zug gewählt, der für MAX den höchsten Gewinn bringt.

Suchbaum bei TicTacToe



Berechnung der Knotenwerte



Resourcenbegrenzung

- **Problem**: Vollständiger Baum ist zu groß ($b \approx 35$, $m \approx 100$ für "normale" Spiele bei 10^{40} legalen, verschiedenen Positionen), um in endlicher Zeit völlig durchsucht zu werden
- **Lösung**:
  - ⇨ Tiefenbegrenzung
  - ⇨ Bewertung einzelner Zustände/Positionen im Hinblick auf eine Gewinnchance
  - ⇨ Für Schach: Materialwert für einzelne Figuren und lineare, gewichtete Summe von Merkmalen,

    Z.B.

$$v(s) = \omega_1 f_1(s) + \omega_2 f_2(s) + \ldots + \omega_n f_n(s)$$

$$\text{z.B. mit } \omega_1 = 9 \text{ und}$$

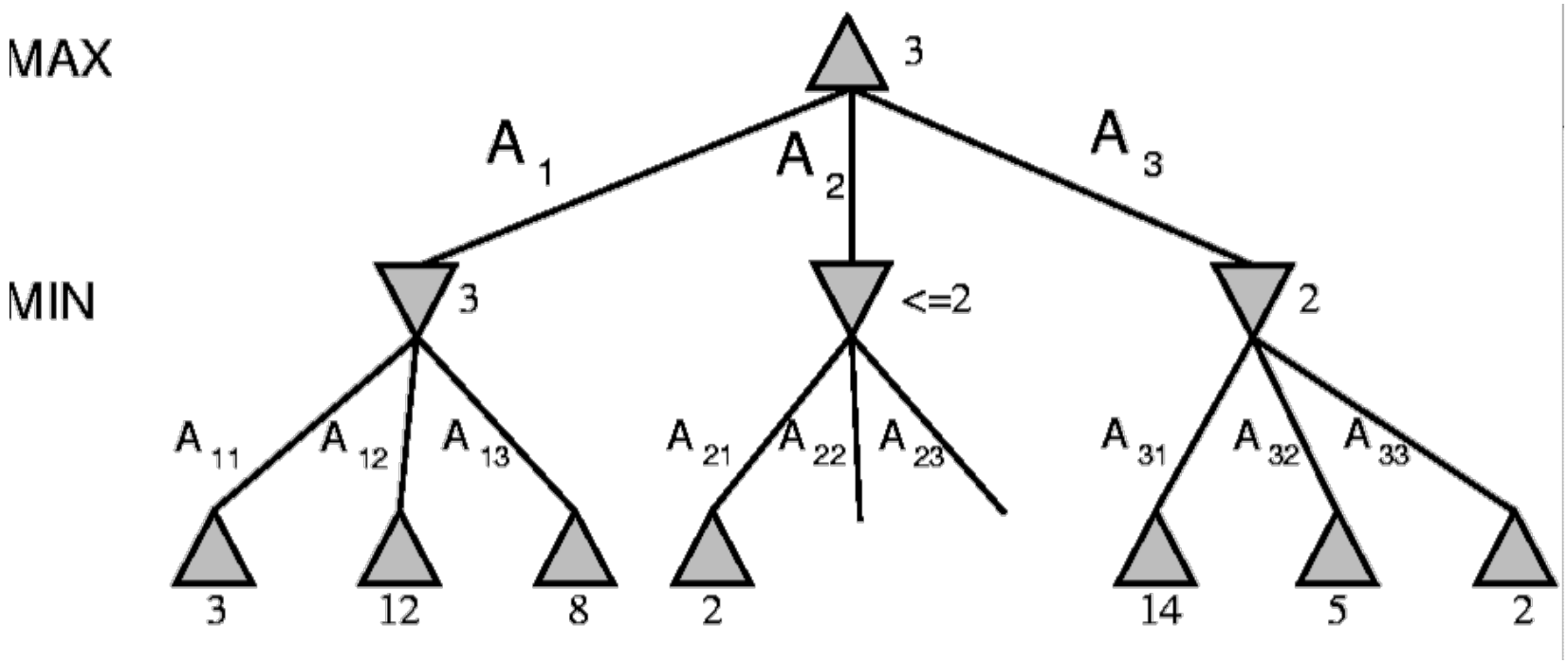
$$f_1(s) = (\text{Anzahl weißer Königinnen} - \text{Anzahl schwarzer Königinnen})$$

MinimaxCutoff

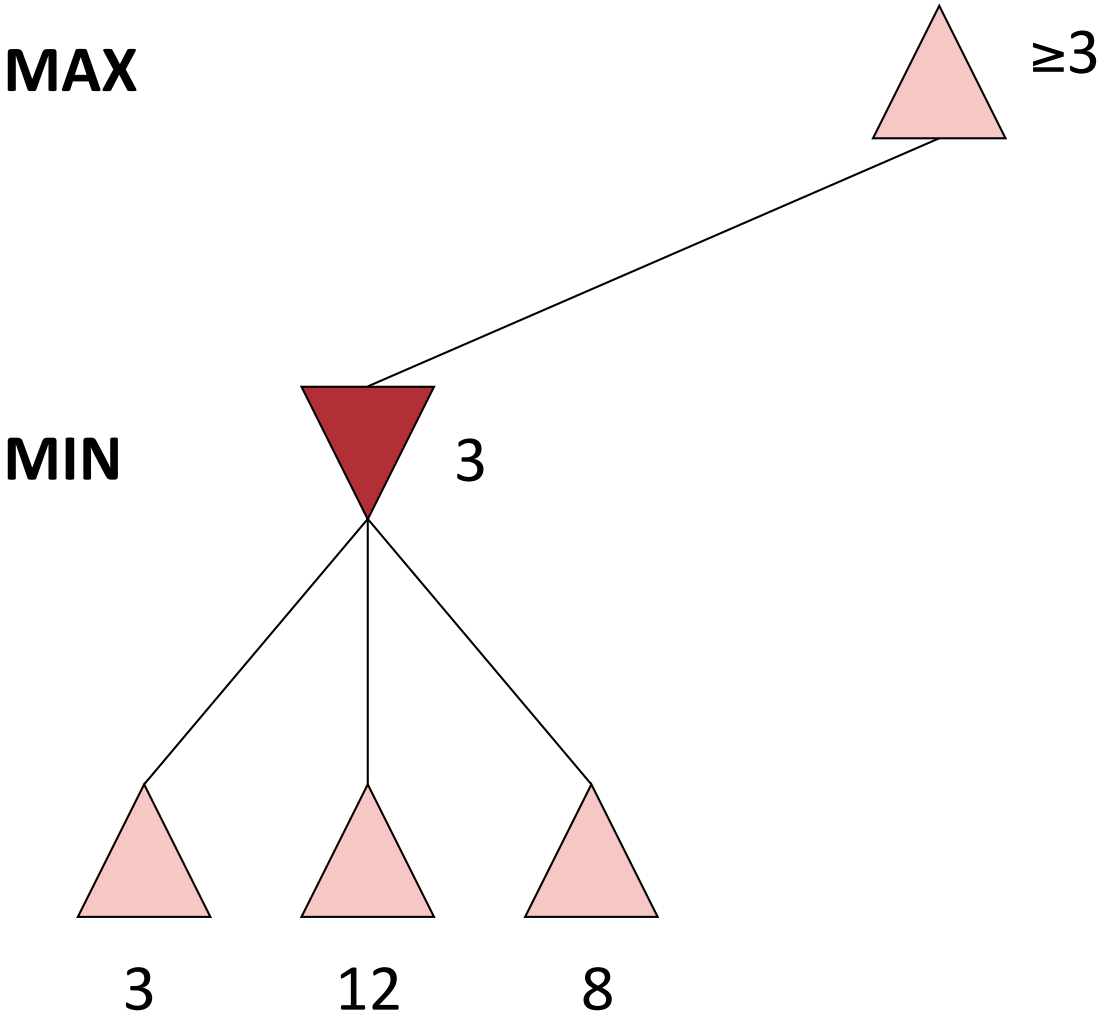
- Minimax-Suche
  - + maximale Tiefe m
  - + Bewertungsfunktion
- Beispiel: Schach
  - ⇨ *m* = 4: Anfänger-Level
  - ⇨ *m* = 8: Meister-Level, typischer PC
  - ⇨ *m* = 12: Kasparov, Deep Blue (RS/6000, 32 Knoten mit je 8 Schach-Prozessoren)

$\alpha\beta$ -Pruning

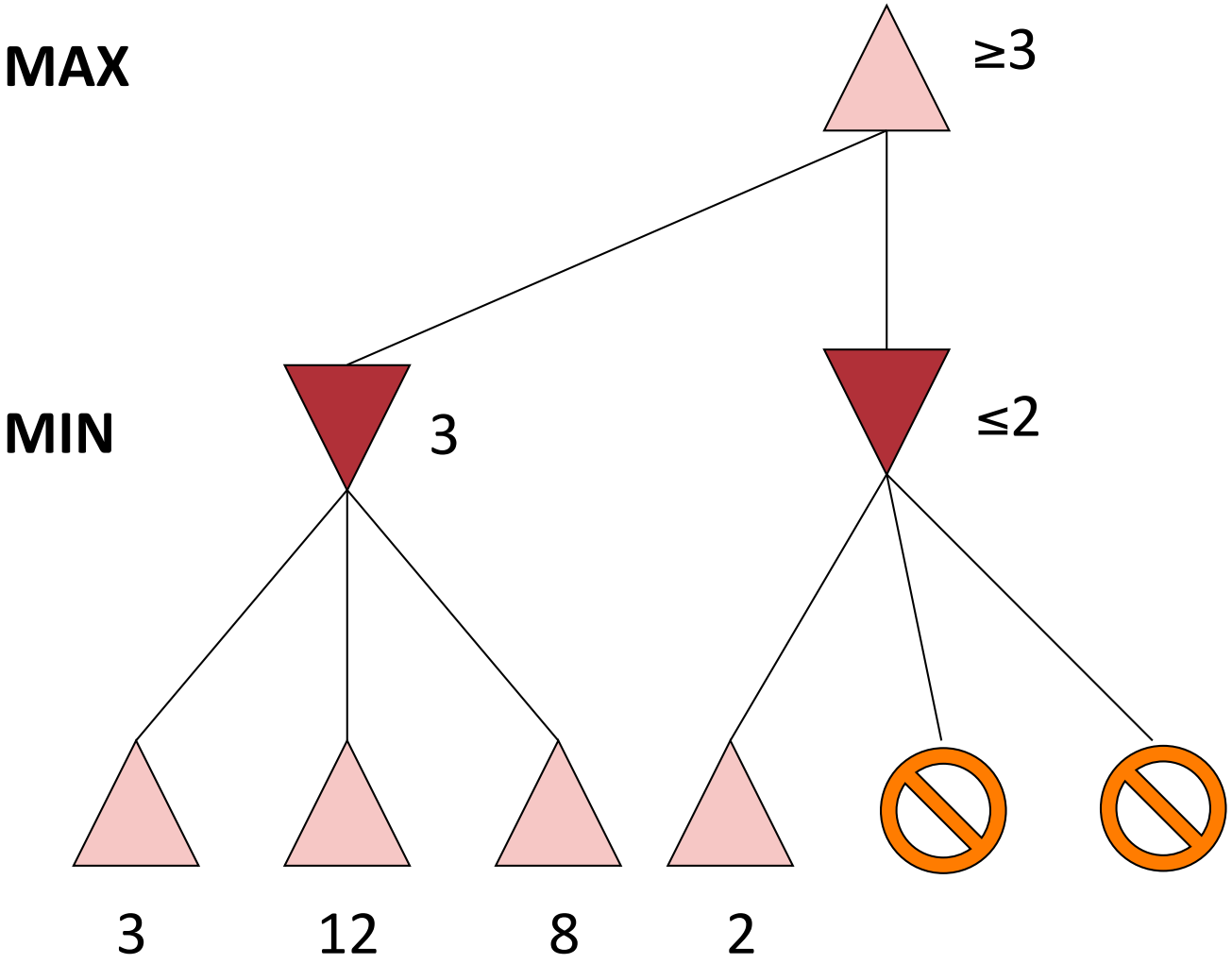
- Verwerfen von Ästen des Suchbaums, die Ergebnis nicht beeinflussen
- “Glückliche” Ordnung reduziert Zeit um bis zur Hälfte bzw. ermöglicht doppelte Tiefe



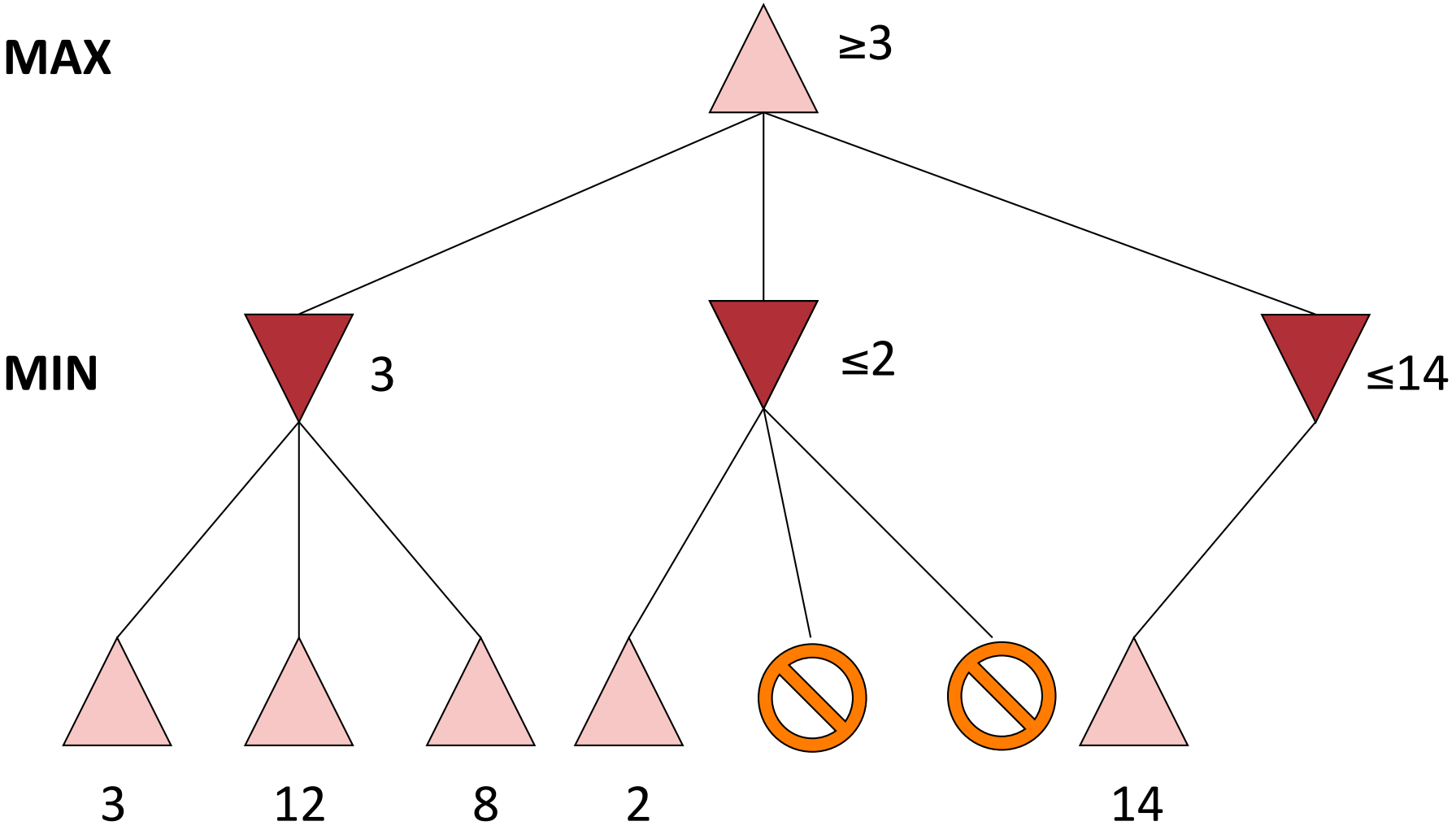
$\alpha\beta$ -Pruning: Beispiel



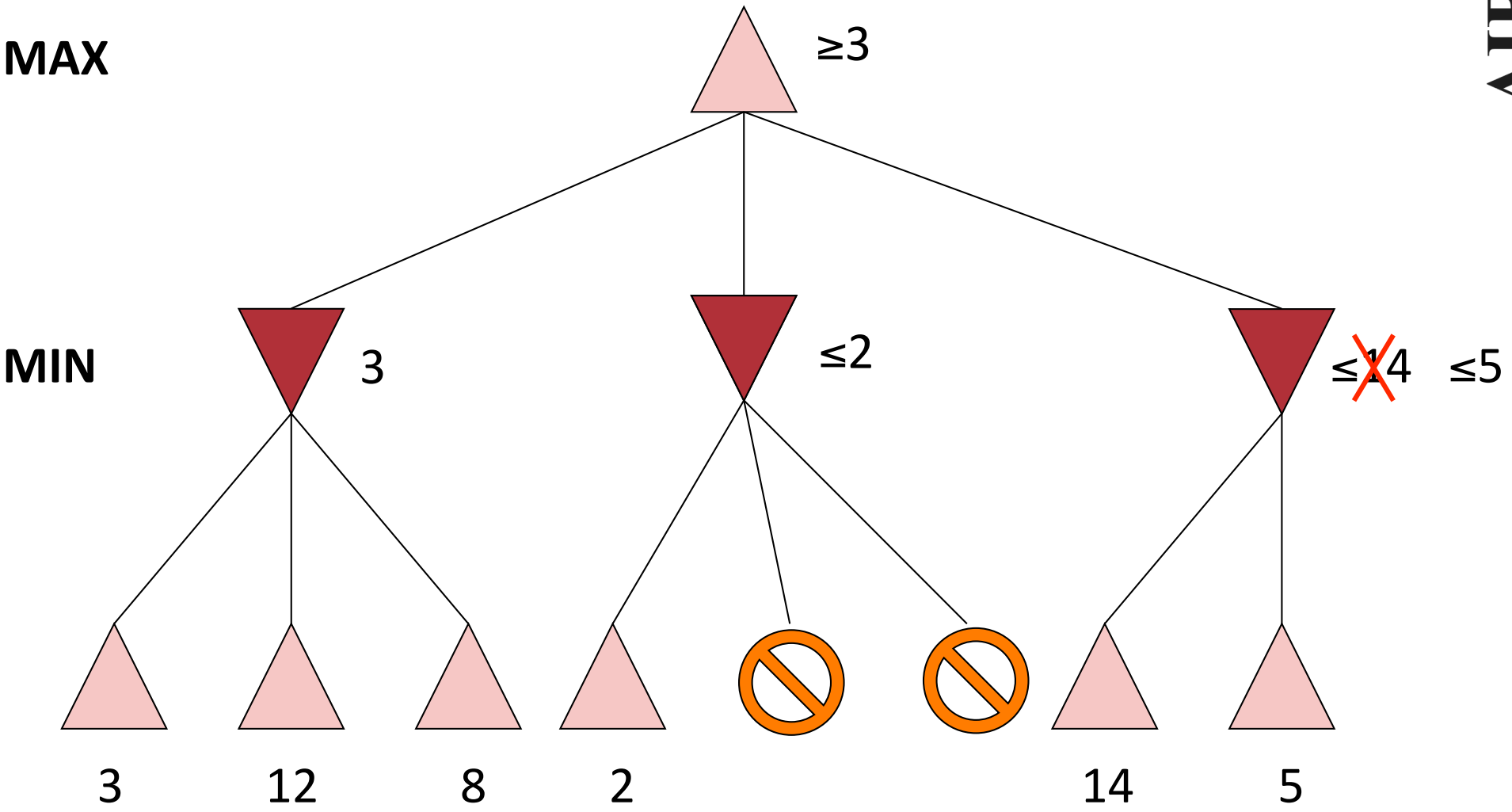
$\alpha\beta$ -Pruning: Beispiel



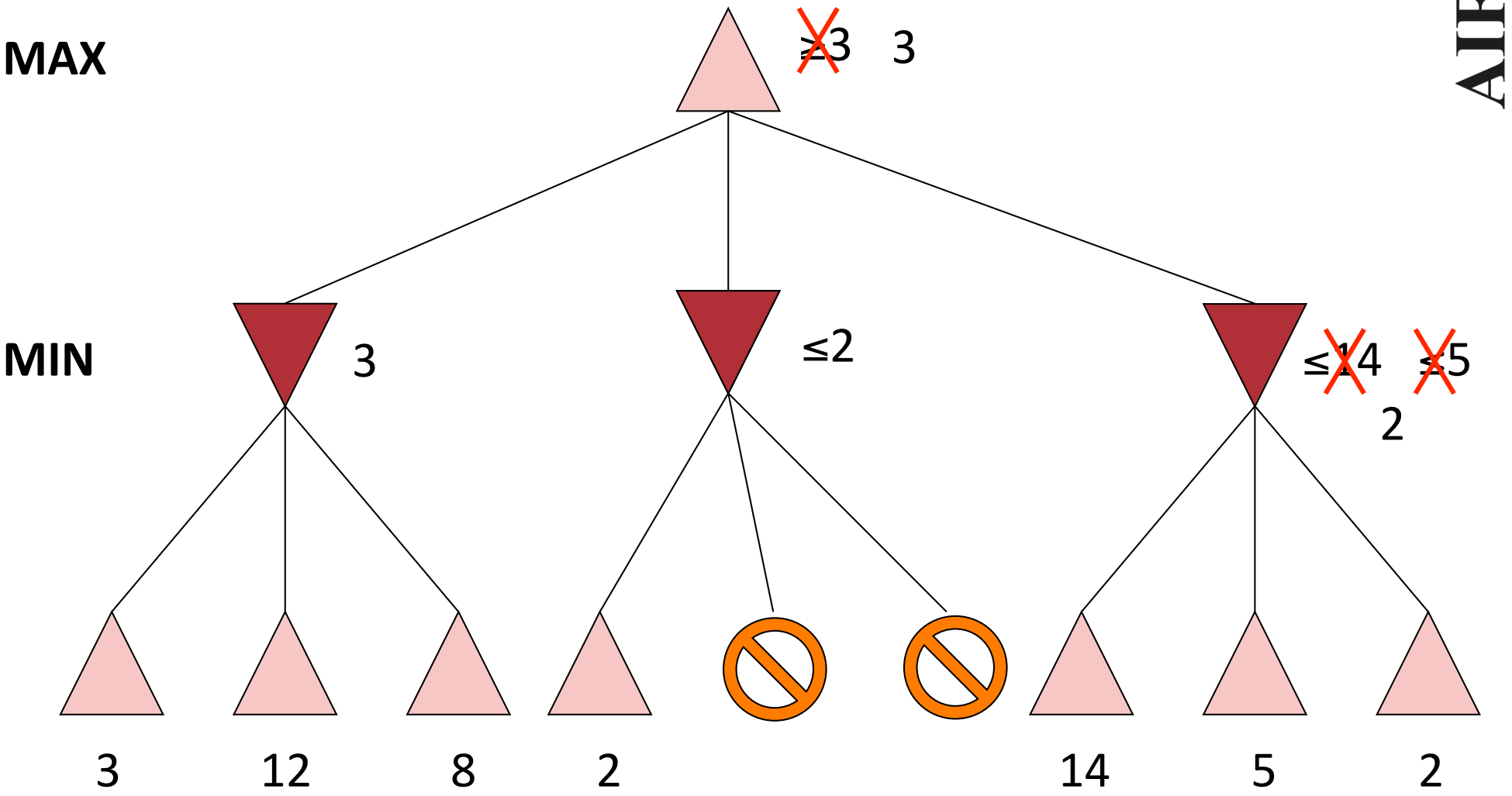
$\alpha\beta$ -Pruning: Beispiel



$\alpha\beta$ -Pruning: Beispiel



$\alpha\beta$ -Pruning: Beispiel



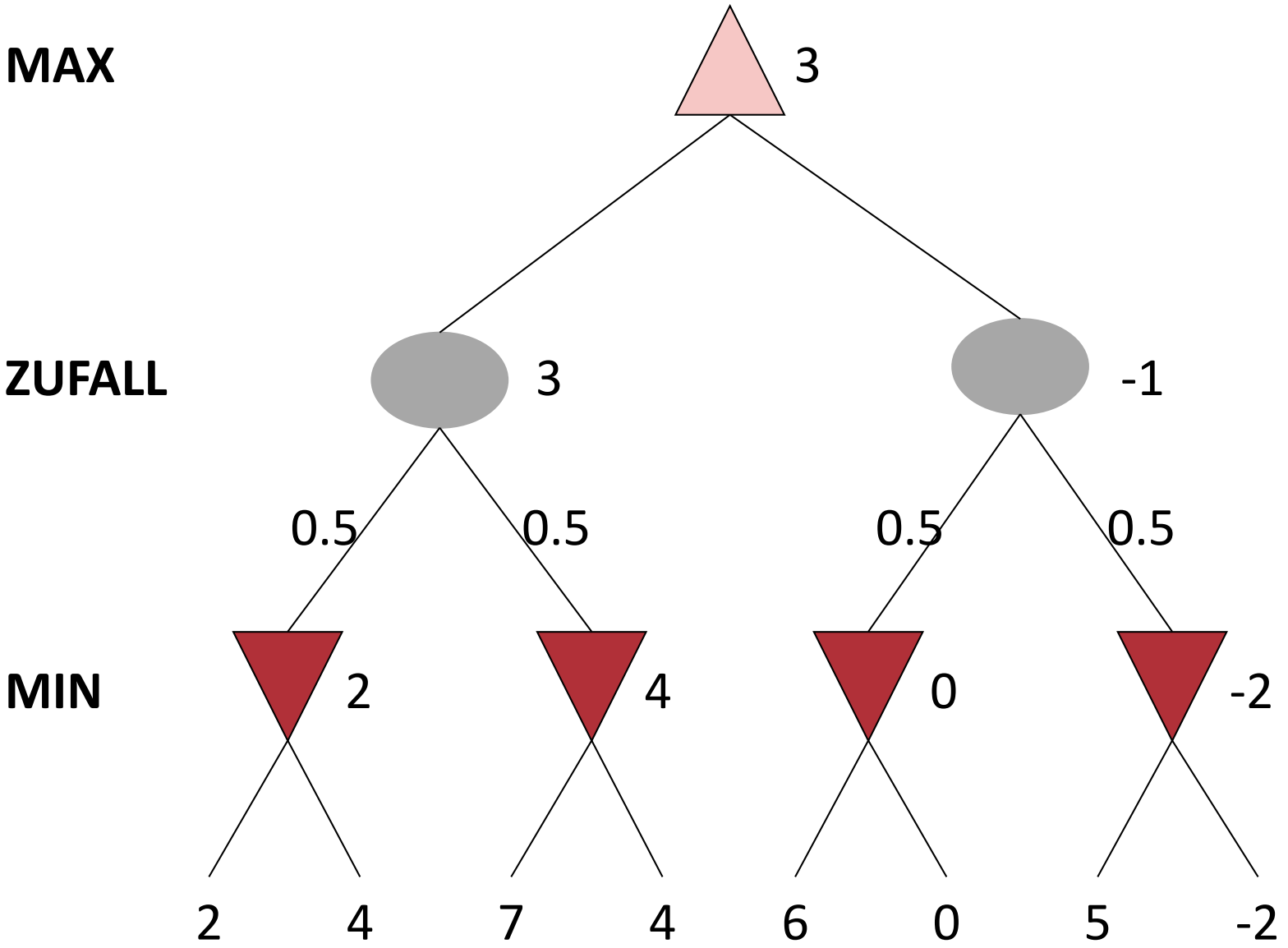
Spiele mit Zufallselement

- Zufallselemente werden durch zusätzliche Knoten einbezogen
- Maximierung von Erwartungswerten

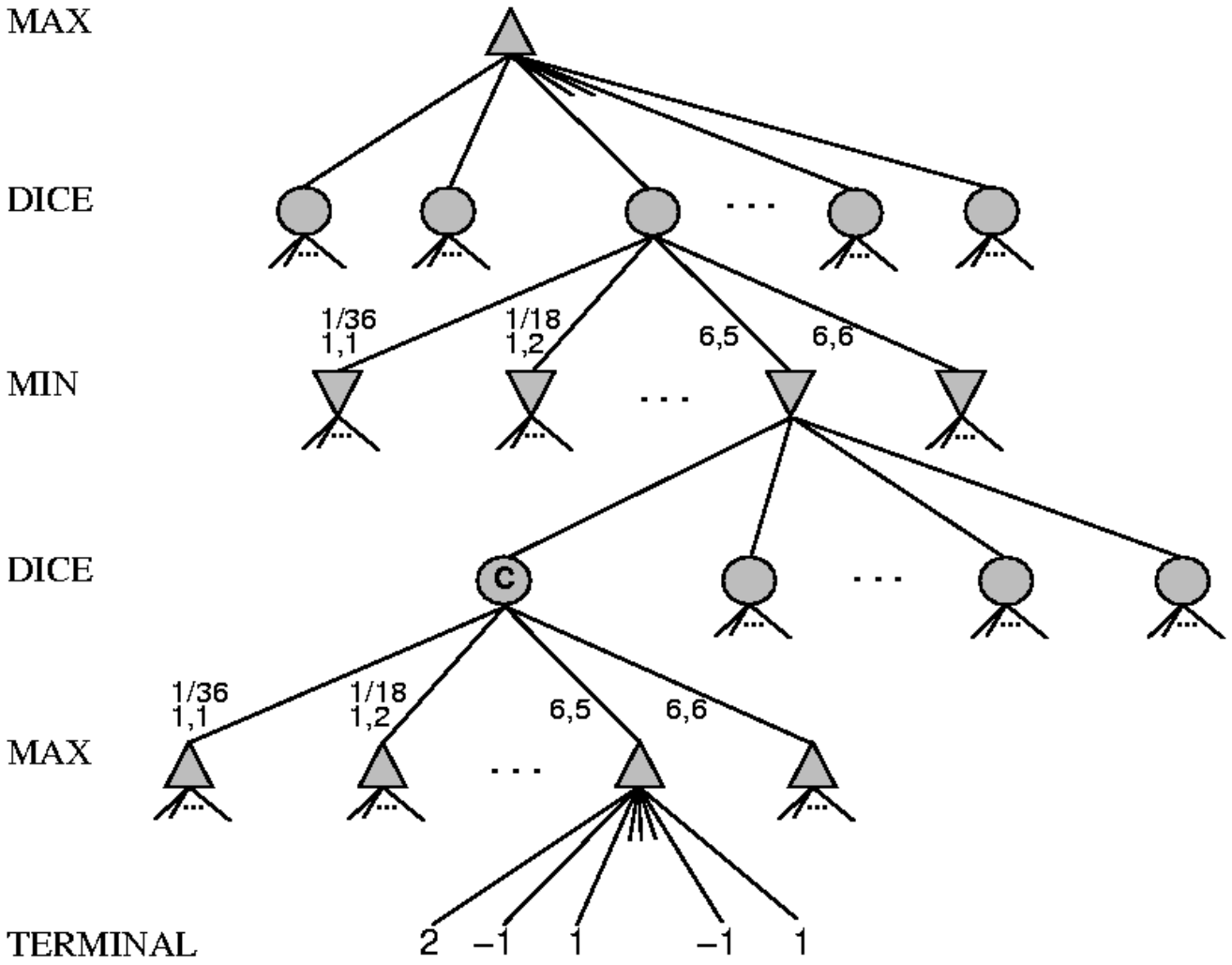
$$expectimax(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (v(s))$$

- $\alpha\beta$-Pruning hier nicht möglich, weil in die Erwartungswertberechnung eines Knotens die Werte aller Nachfolgerknoten eingehen und gebraucht werden

Aufbau Suchbaum



Suchbaum für Spiel mit 2 Würfeln



Mensch vs. Computer

- **Dame**:
  - 1994 wurde Marion Tinsley (40 Jahre Weltmeister) durch einen Computer geschlagen
  - 2007: Team um Jonathan Schaeffer hat das Spiel komplett durchgerechnet und **gelöst**. Rechnerverbund hat 39 Billionen Stellungen bewertet. Perfektes Damespiel liefert immer Remie.
- **Schach**:
  „Schachtürke“ (ab 1769) war getürkt...
  1997 wurde Gary Kasparov durch
  Deep Blue (RS/6000) geschlagen
  - 200 Mio Positions-Evaluierungen /sec
  - Im Einzelfall Tiefensuche bis 40 Züge
  - 2002 spielte Vladimir Kramnik 4:4 gegen Deep Fritz (8 Xeon-Prozessoren)
- **Reversi**: Computer i.A. zu gut (für Meister)
- **GO**: Computer i.A. zu schlecht (für Meister)
- **Backgammon**: auf Weltmeisterniveau