Künstliche Intelligenz

Vorlesung 1, Teil 2: Suchverfahren

Uninformierte Suche

Problemlösen

Die zentralen Fragen mit denen wir uns beschäftigen sind

1. Was ist ein Problem?
2. Was ist eine Lösung?
3. Wie findet man eine Lösung?

Einführung: Suchalgorithmen

- Der einfache Reflexagent kann nur die aktuelle Wahrnehmung auf eine Aktion übertragen.
- Der zielbasierte Agent kann Aktionsfolgen planen, um die Ziele des Agenten erreichen.
- Wir werden nun einen Typ zielorientierter Agenten, den problemlösenden Agent untersuchen. Die hier betrachteten Problemlösungsagenten folgen einem formulate-search-execute Design.
  - Was ist ein Problem und was ist seine Lösung?
  - Wie finde ich eine Lösung?
  - Suchalgorithmen BFS, DFS, DLS, IDS, BiS

Bemerkung

Zuerst werden wir uninformierte Suchalgorithmen untersuchen, d.h. Algorithmen, die keine problemspezifischen Informationen ausnutzen (und eine atomare Repräsentation verwenden).

Ein Problemlösungsagent mit formulate-search-execute Design verfährt nach folgendem Schema:

1. Zunächst wird das Ziel formuliert: Fahre von Berlin nach München
2. Anschließend transformiert der Agent die Zielformulierung in eine adäquate Problemformulierung, bei der die möglichen Zustände und Aktionen festgelegt werden. Sowohl Ziel- als auch Problemformulierung gehören zur formulate-Phase des Agenten.
3. In der search-Phase sucht der Agent eine Sequenz von Aktionen, die angewendet auf den Anfangszustand (Berlin) in den Zielzustand (München) führen. Beispiel: Fahren von Berlin nach Nürnberg und dann von Nürnberg nach München.
4. Schließlich in der execute-Phase wird die gefundene Lösung ausgeführt.

Problemlösungsagenten

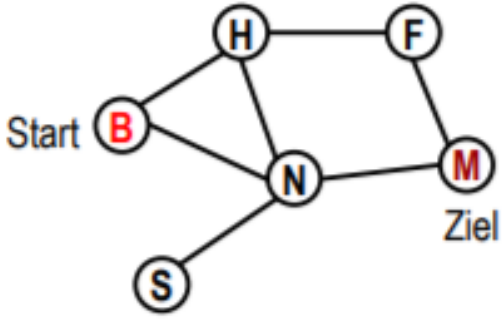
Beispiel für einen Problemlösungsagenten

⇨ **Ziel formulieren** (formulate I):
- → Fahre von Berlin nach München

⇨ **Problem formulieren** (formulate II):
- → Zustände: Städte B, F, H, M, N, S
- → Aktionen: Fahre zwischen Städten

⇨ **Suche (search):**
- → Finde Sequenz von Aktionen zum Ziel
- → z.B.: **B – N - M**

⇨ **Ausführen (execute):**
- → Führe gefundene Lösung aus



B	Berlin (Start)
F	Frankfurt
H	Hannover
M	München (Ziel)
N	Nürnberg
S	Stuttgart

Problemlösungsagenten

Wichtige Entwurfsentscheidungen

- Problemformulierung
  - Wie formuliert man Probleme?
- Suchalgorithmen
  - Wie findet man eine Lösung?

- Die wichtigsten Entwurfsentscheidungen, die wir zu treffen haben sind die Problemformulierung und die Suche nach einer Lösung.
- Beides ist problemabhängig und mehr eine Kunst als eine Wissenschaft.
- Insbesondere von der Problemformulierung hängt die Leistungsfähigkeit eines Problemlösungsagenten ab.
- Bei einer idealen Problemformulierung fällt uns die Lösung gewissermaßen in den Schoß. Bei einer schlechten Problemformulierung wird die Suche nach einer Lösung zu einem komplizierten und aufwändigen Unterfangen.
- Die Wahl oder der Entwurf eines Suchverfahrens hängt nicht nur vom Problem selbst sondern u.a. auch von der Problemformulierung ab.

Problemformulierung

- Problemraum: Menge S von Zuständen
- Problem: Tripel $P = (S_a, S_Z, A)$, bestehend aus
  - einer Menge $S_a \subseteq S$ von Anfangszuständen
  - einer Menge $S_Z \subseteq S$ von Zielzust'anden
  - einer Menge A von Aktionen
- Aktionen: $a \in A, a\colon S' \to S$, mit $S' \subseteq S$
  - Aktionen sind partiell definiert (Anwendungsbedingungen)

Bemerkungen

1. Es kann mehrere Anfangszustände geben. Beispiel: Finde Wege von Berlin und Hannover nach München.
2. Es kann mehrere Zielzustände geben. Beispiel: 8-Dame Problem. Positioniere 8 Damen auf dem Schachbrett, so dass keine Dame eine andere schlagen kann. Für dieses Problem gibt es mehrere Konfigurationen. Jede dieser Konfigurationen ist ein Zielzustand.
3. Eine Aktion muss nicht nur auf einen Zustand sondern kann auf einer Teilmenge von Zuständen definiert sein. Zum Beispiel auf einem Schachbrett den König um eine Position nach links zu schieben, kann von mehreren Feldern ausgeführt werden, sofern die Anwendungsbedingungen gemäß der Regeln des Schachs erfüllt sind.

Problemformulierung

- Lösung: Sequenz $(s_0, \ldots, s_k)$ von Zuständen mit
  - $s_0 \in S_a$ ist ein Anfangszustand
  - $s_k \in S_Z$ ist ein Zielzustand
  - es gibt eine Aktion a_i mit $a_i(s_i) = s_{i+1}$ für alle $i = 0, \ldots, k-1$
- Optimale Lösung: Lösung minimaler Länge
- Problemlösen: Finden einer (optimalen) Lösung
  - Überführung eines Anfangzustandes in einen Zielzustand durch Anwendung einer (minimalen) Folgen von Aktionen

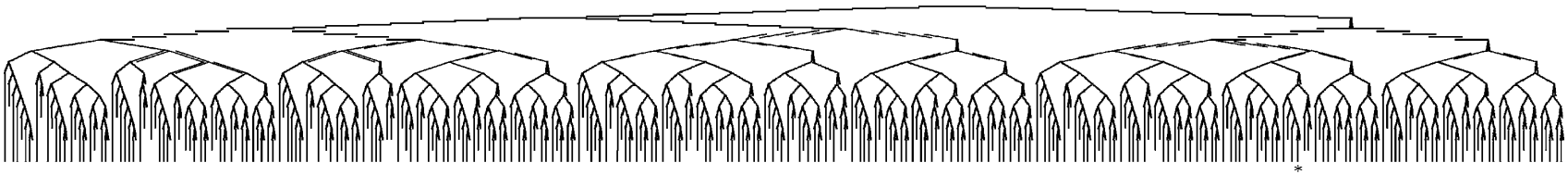
Suchverfahren

Beispiele

- Spiele: Suche nach dem besten Zug
- Logik: Suche nach einer Herleitung einer Aussage
- Agenten: Suche nach der optimalen nächsten Aktion
- Planen: Suche nach einer Folge von Aktionen eines intelligenten Agenten.
- Optimierung: Suche eines Maximums einer mehrstelligen Funktion auf den reellen Zahlen

Wie sucht man nach einer Lösung?

- Bei fast allen Inferenzsystemen stellt die Suche nach einer Lösung, bedingt durch die extrem großen Suchbäume, ein Problem dar.
- Aus dem Startzustand gibt es für den ersten Inferenzschritt viele Möglichkeiten. Für jede dieser Möglichkeiten gibt es im nächsten Schritt wieder viele Möglichkeiten und so weiter.



WIE SUCHT MAN NACH EINER LÖSUNG

$\longrightarrow$ SUCHSTRATEGIE!!!

Wie suchen wir nach einer Lösung

- Dieses machen wir durch Suche im Problemraum.
- Die systematische Suche generiert Suchbäume.
- Wir geben Kriterien an, die uns sagen, wie effizient die systematische Suche ist.
- Unterschiedliche Verfahren unterscheiden sich in der Strategie, mit der ein Suchbaum aufgebaut wird.

Inferenz

Schach

- Verzweigungsfaktor $b = 30$, Tiefe $d = 50$
- $30^{50} = 7.2 \cdot 10^{73}$ Blattknoten
- Anzahlt Inferenzschritte = $\sum_{d=0}^{50} 30^d = \frac{1-30^{51}}{1-30} = 7.4 \cdot 10^{73}$

Beispiel: Schach - Inferenzkomplexität

- 10000 Computer
- je einer Milliarde Inferenzen pro Sekunde
- Parallelisierung ohne Verluste
- Rechenzeit:

$$\frac{7.4 \cdot 10^{73} \text{ Inferenzen}}{10000 \cdot 10^{9} \text{ Inferenzen/sec}} = 7.4 \cdot 10^{60} \text{ sec} = 2.3 \cdot 10^{53} \text{ Jahre},$$

- 10^{43} mal Alter des Universums.

SCHLUSSFOLGERUNG

Suchproblem

Definition

Ein Suchproblem wird definiert durch folgende Größen

- *Zustand: Beschreibung des Zustands der Welt in dem sich ein Suchagent befindet.*
- *Startzustand: der Initialzustand in dem der Agent gestartet wird.*
- *Zielzustand: erreicht der Agent einen Zielzustand, so terminiert er und gibt (falls gewünscht) eine Lösung aus.*
- *Aktionen: Alle erlaubten Aktionen des Agenten.*
- *Lösung: Der Pfad im Suchbaum vom Startzustand zum Zielzustand.*
- *Kostenfunktion: ordnet jeder Aktion einen Kostenwert zu. Wird benötigt, um kostenoptimale Lösungen zufinden.*
- *Zustandsraum: Menge aller Zustände.*

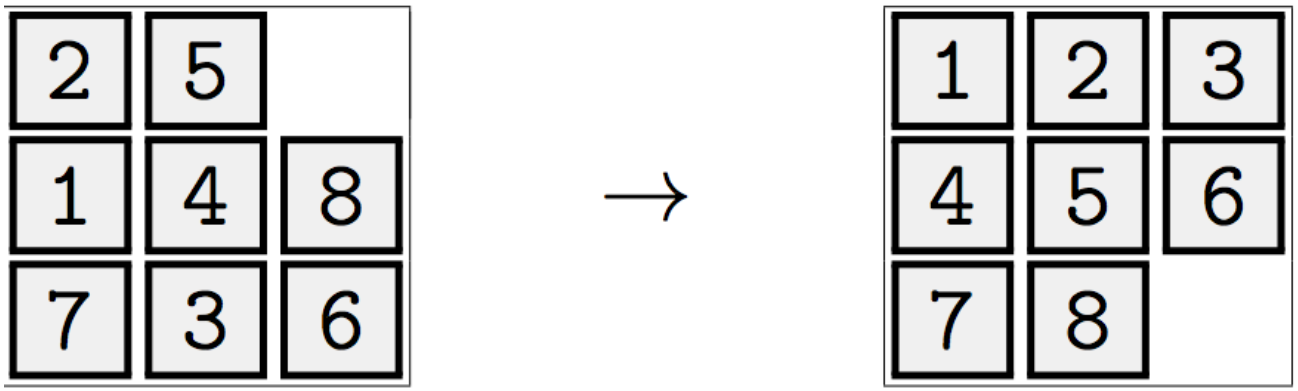
Beispiele

- Schach
  - Startzustand/Anfangssituation: Eine Stellung im Spiel
  - Aktionen/Nachfolgerfunktion: Mögliche Züge
  - Zielsituation/Zielzustand: Wenn man gewonnen hat
  - Gesucht: Zug der zum Gewinn führt
- Deduktionssystem
  - Startzustand/Anfangssituation: Zu beweisende Aussage A, Menge von Axiomen und bewiesenen Sätzen
  - Aktionen/Nachfolgerfunktion: Deduktionsregeln
  - Zielzustand/Zielsituation: Beweis
  - Gesucht: Beweis für A
- Planen
  - Startzustand/Anfangssituation: Formale Beschreibung des interessierenden Bereichs z.B. Fahrplan, Start- und Zielort,
  - Aktionen/Nachfolgerfunktion: Zugverbindungen usw.
  - Gesucht: Plan, der Reise ermöglicht

Uninformierte Suche

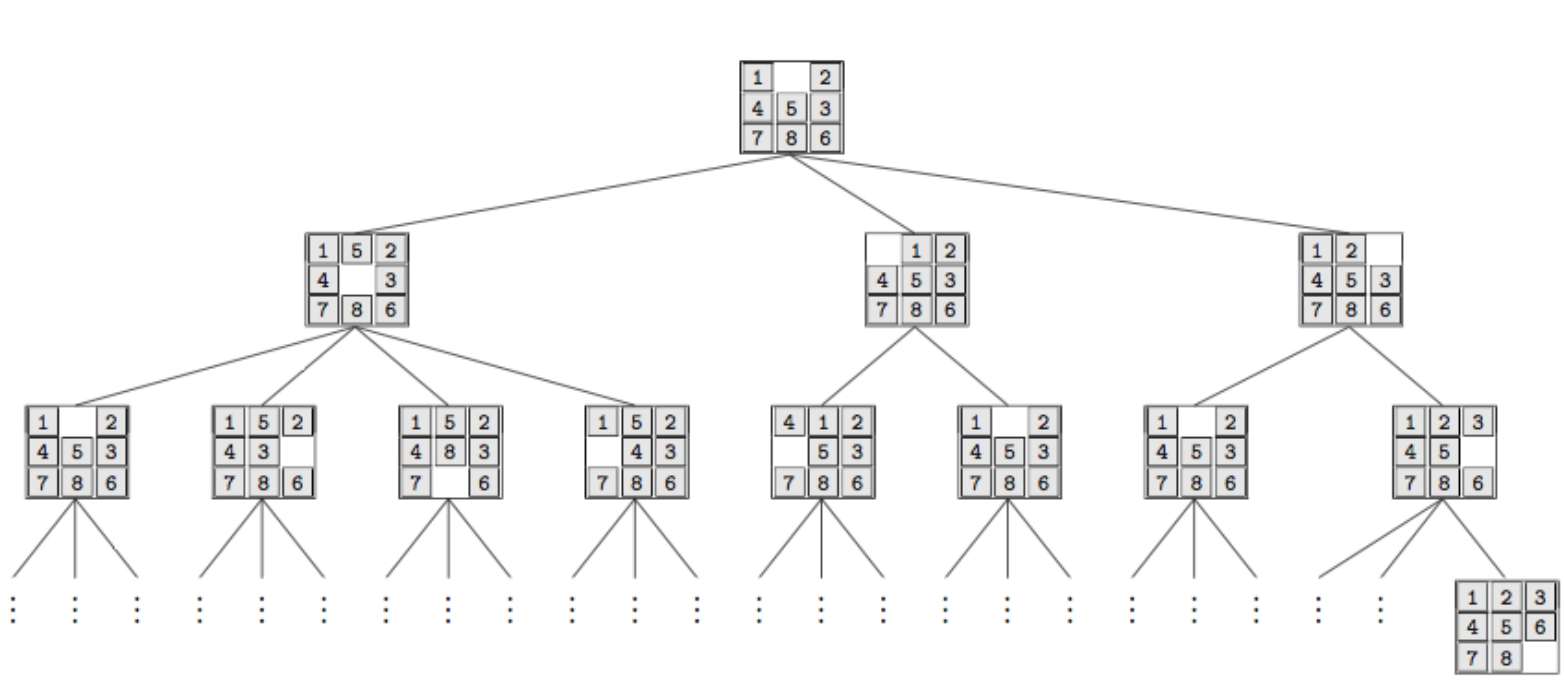
Wie funktioniert uninformierte Suche, d.h. das blinde Durchprobieren aller Möglichkeiten?

Beispiel: 8-Puzzle



Mögliche Start- und Zielzustände des 8-Puzzle.

8-Puzzle: Suchbaum



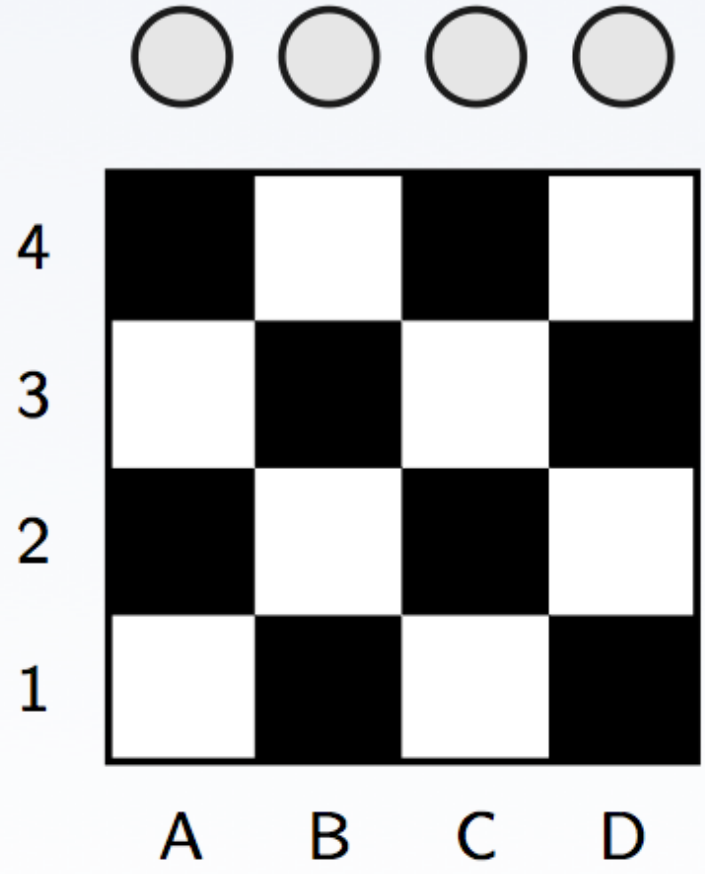
Wie entstehen Suchbäume?

Ausgehend von einem Zustand s führt eine Aktion a_1 in einen neuen Zustand s'. Es gilt also $s' = a_1(s)$. Eine andere Aktion kann in einen anderen Zustand s'' führen, das heißt $s'' = a_2(s)$. Durch die rekursive Anwendung aller möglichen Aktionen auf alle Zustände, beginnend mit dem Startzustand, entsteht der Suchbaum.

8-Puzzle

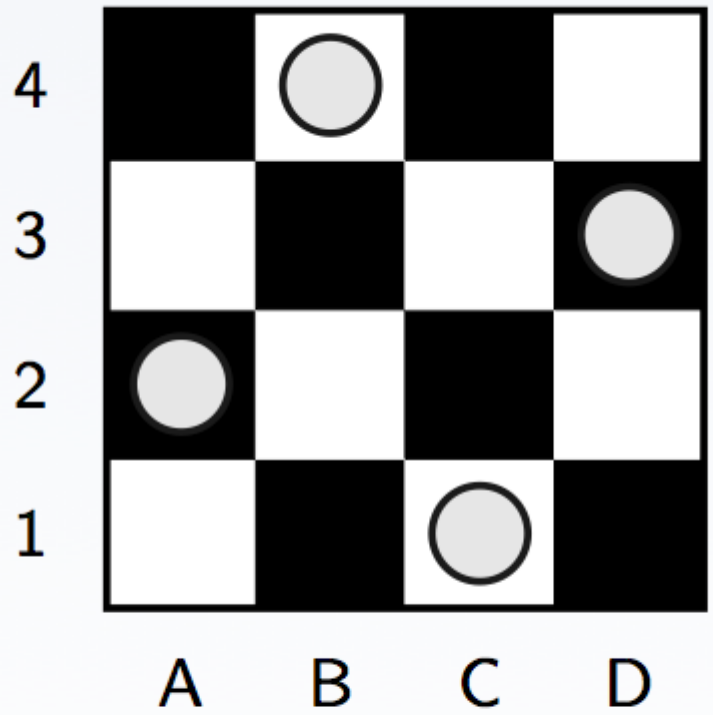
- Zustand: 3×3 Matrix S mit den Werten $1, 2, 3, 4, 5, 6, 7, 8$ (je einmal) und einem leeren Feld.
- Startzustand: Ein beliebiger Zustand.
- Zielzustand: Ein beliebiger Zustand.
- Aktionen: Bewegung des leeren Feldes S_{ij} nach links (falls $j \neq 1$), rechts (falls $j \neq 3$), oben (falls $i \neq 1$), unten (falls $i \neq 3$).
- Kostenfunktion: Die konstante Funktion 1, da alle Aktionen gleich aufwändig sind.
- Zustandsraum: Der Zustandsraum zerfällt in Bereiche, die gegenseitig nicht erreichbar sind. Daher gibt es nicht lösbare 8-Puzzle-Probleme.

Beispiel: n Damen



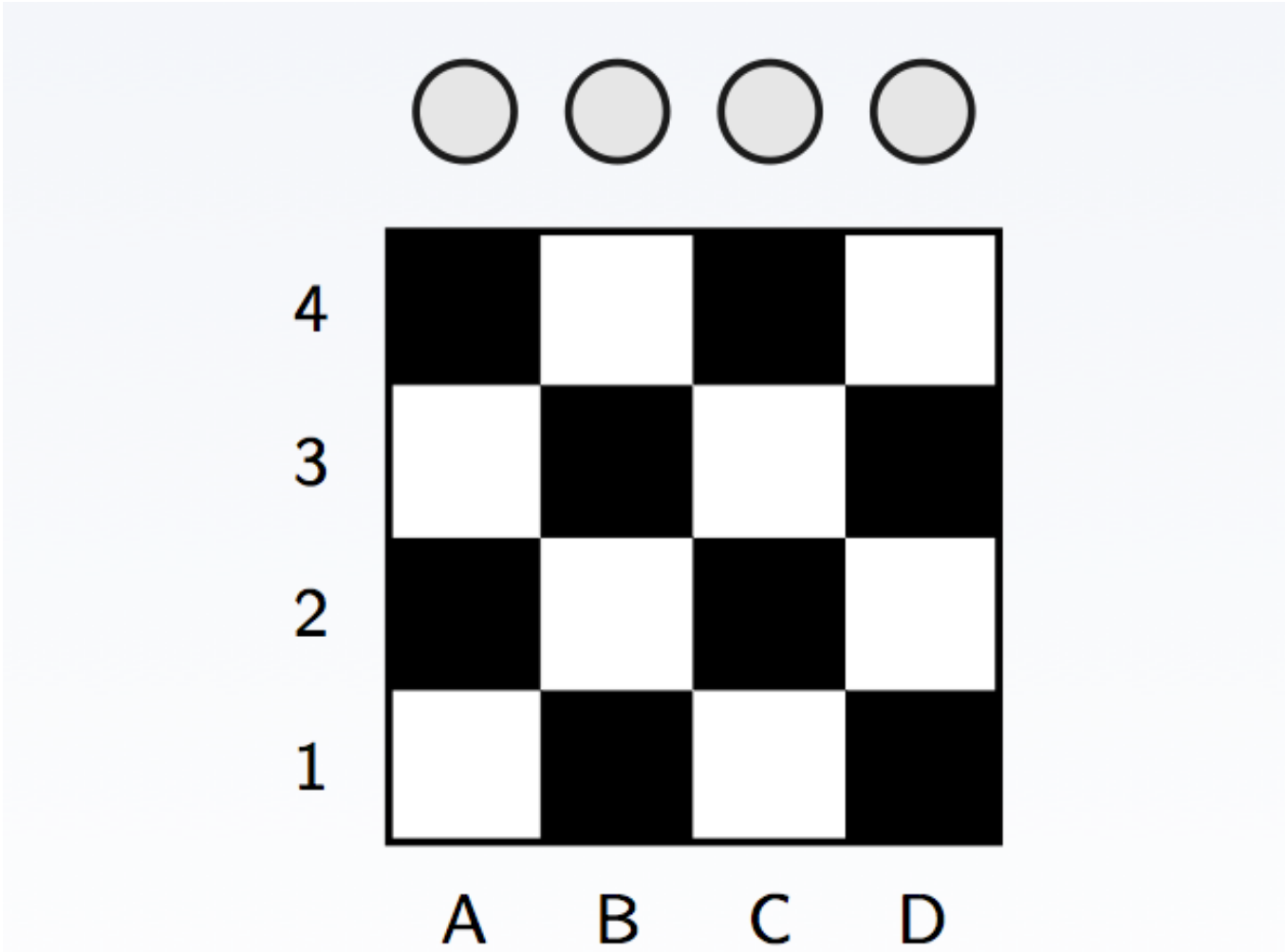
Platziere auf $n \times n$ Schachbrett n Damen,
so dass keine die andere bedroht

Beispiel: n Damen, mögliche Lösung



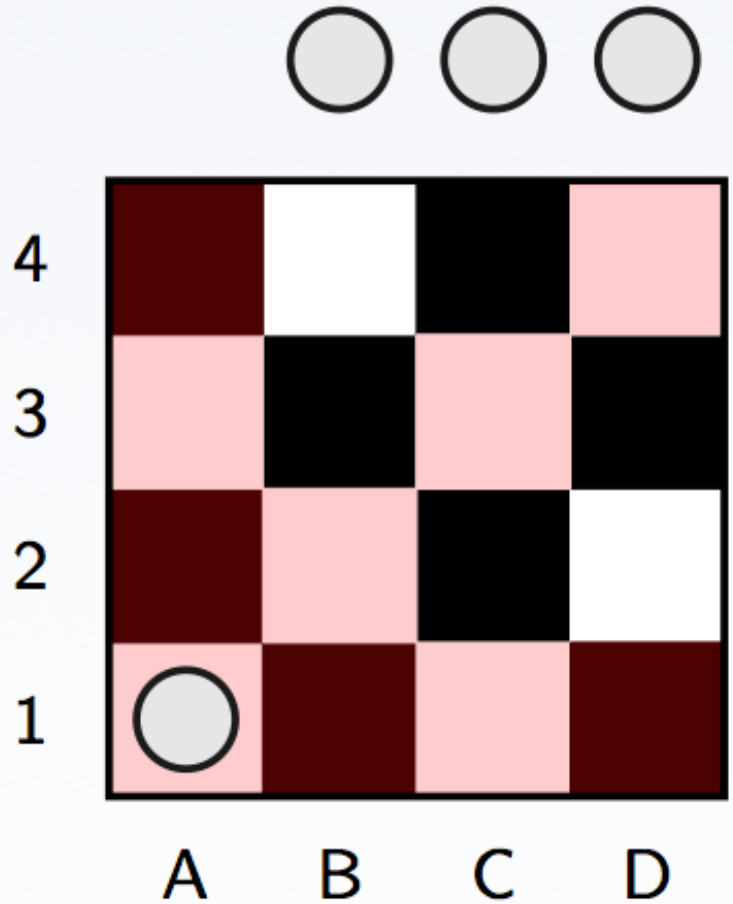
Platziere auf $n \times n$ Schachbrett n Damen,
so dass keine die andere bedroht

Beispiel: n Damen, mögliche Suche



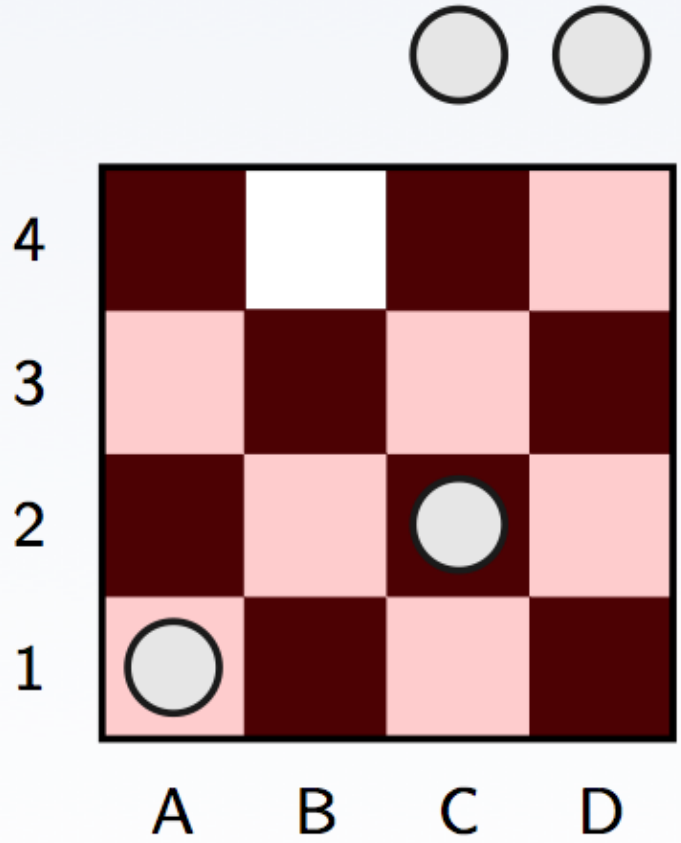
Platziere die Damen zeilenweise nacheinander, und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



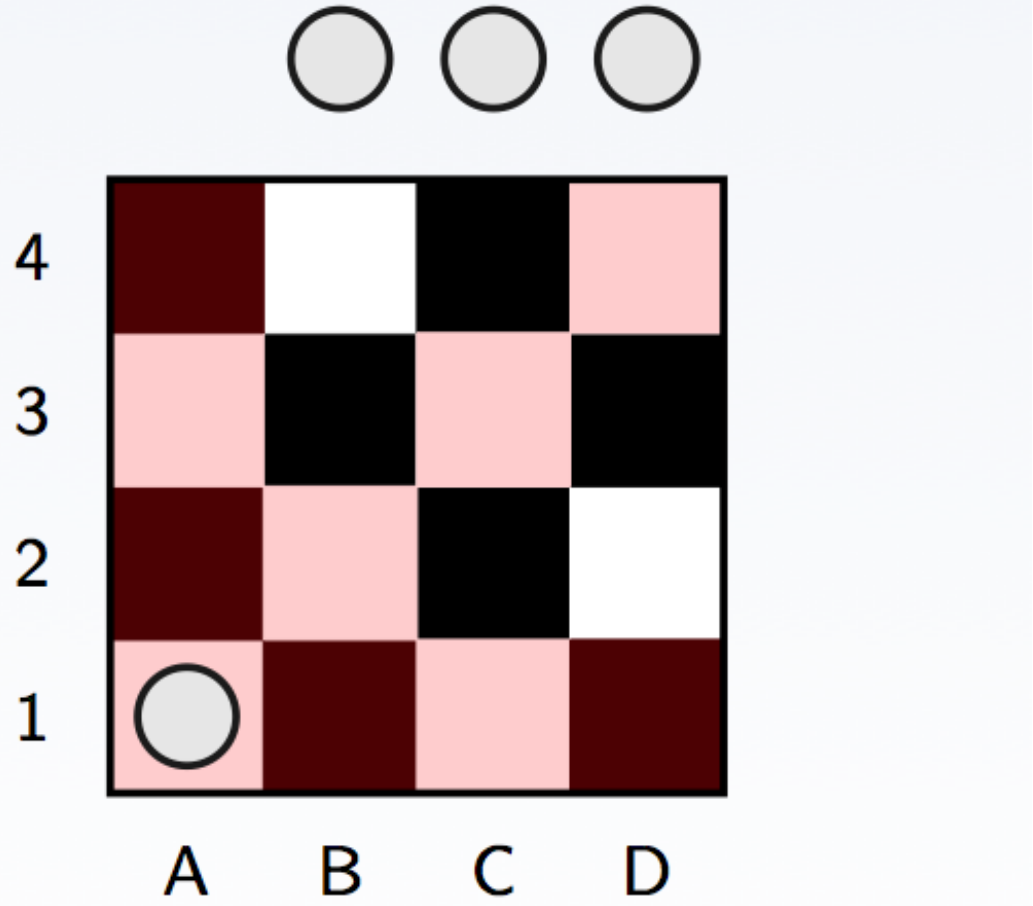
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



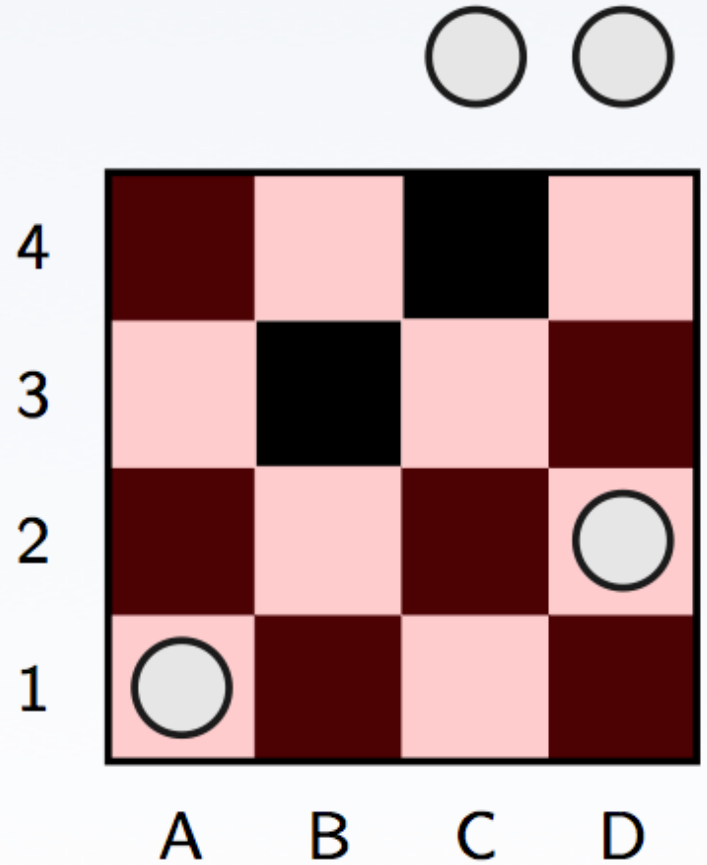
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



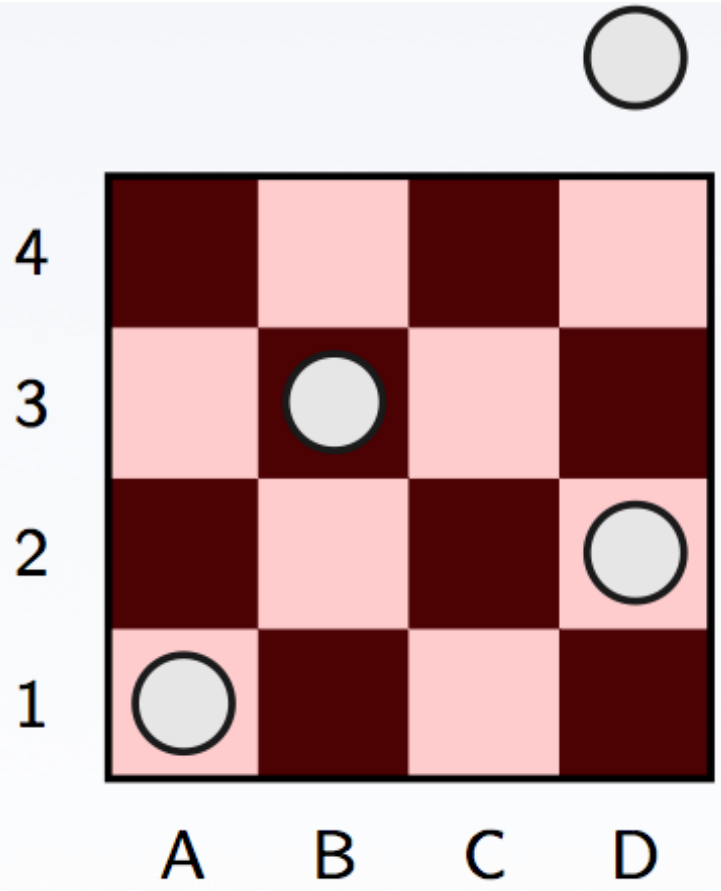
Platziere die Damen zeilenweise nacheinander, und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



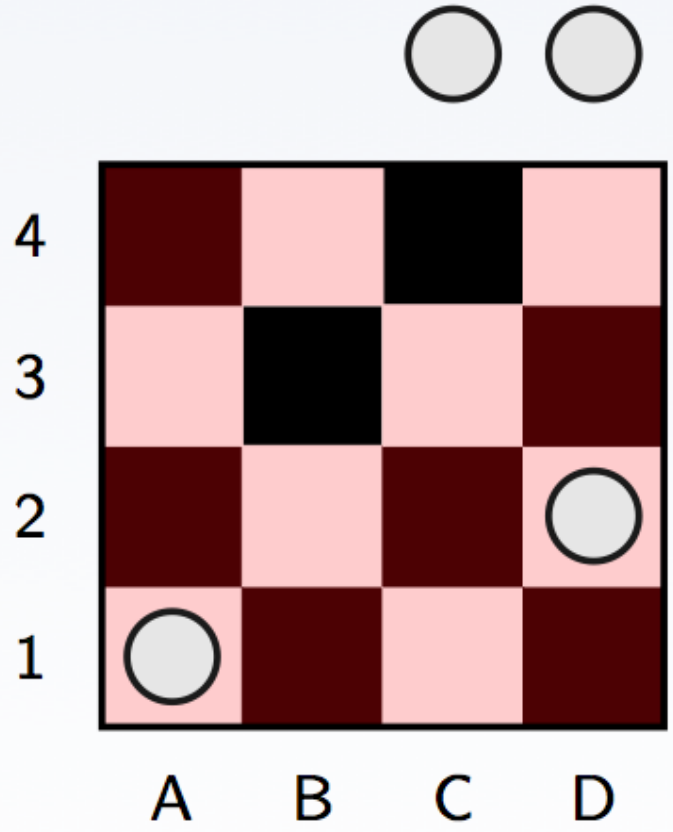
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



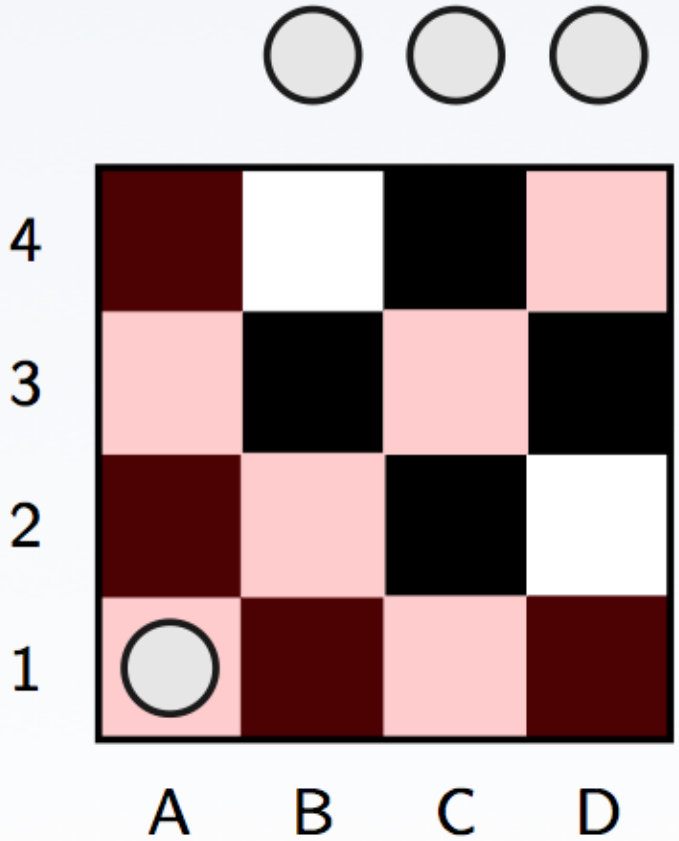
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



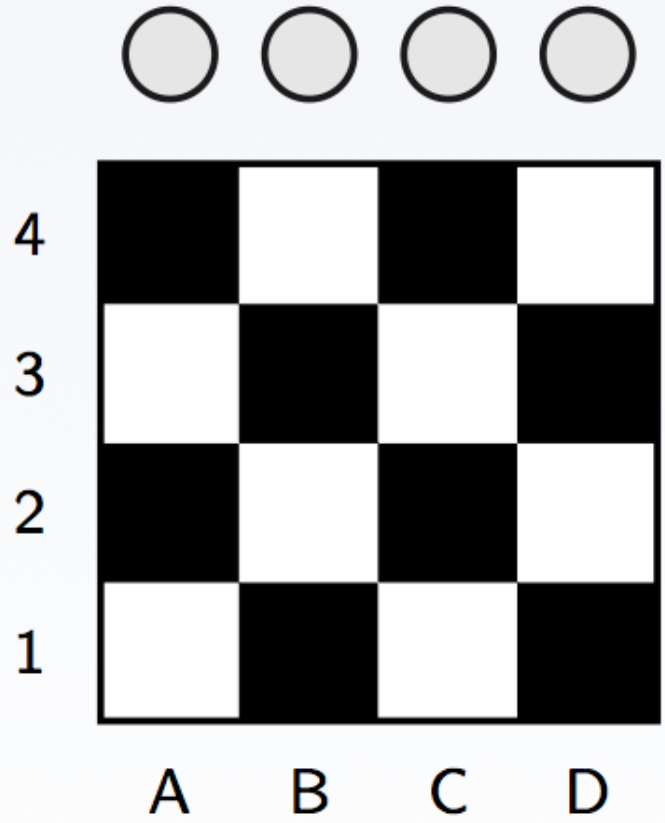
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



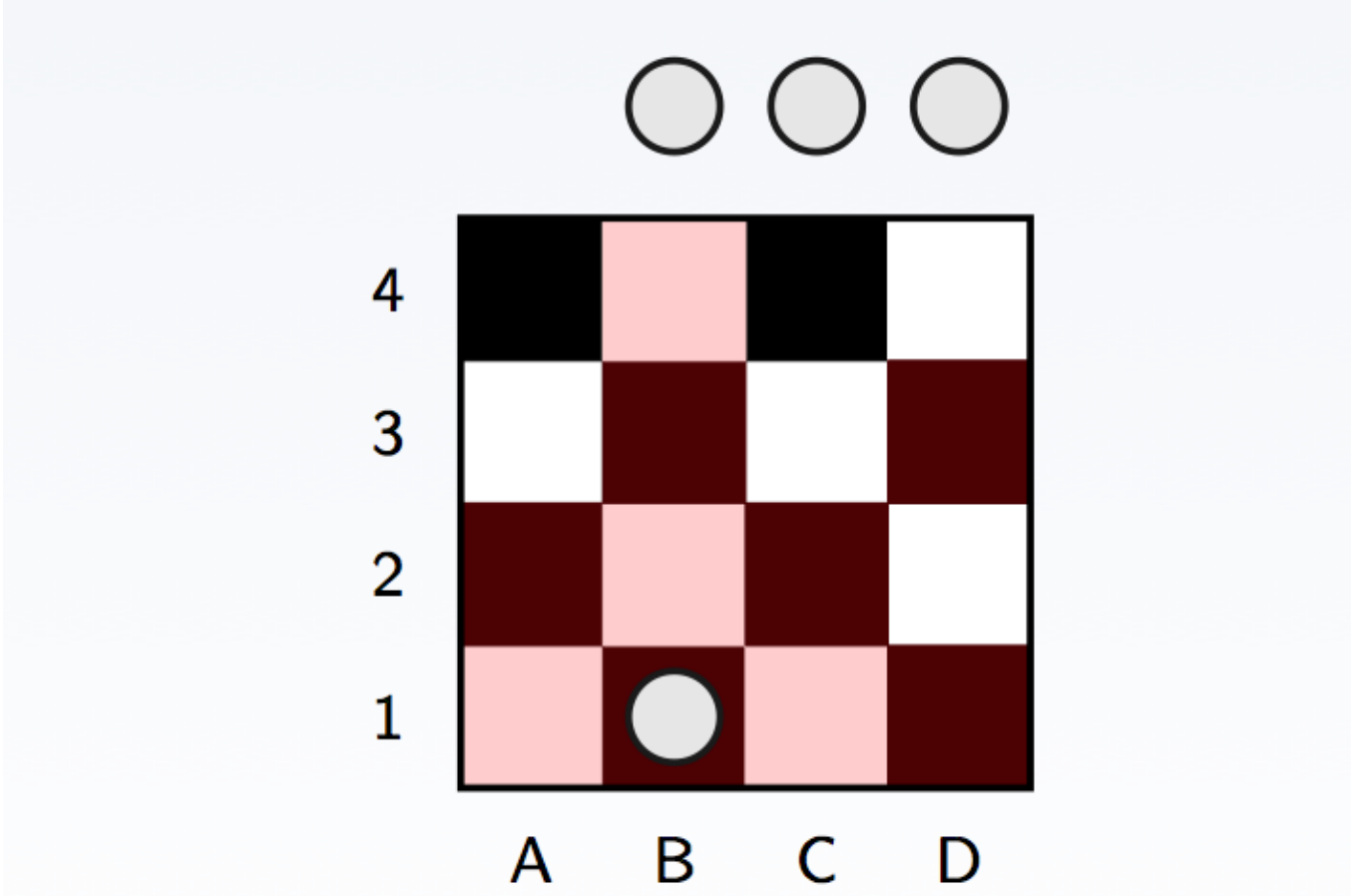
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



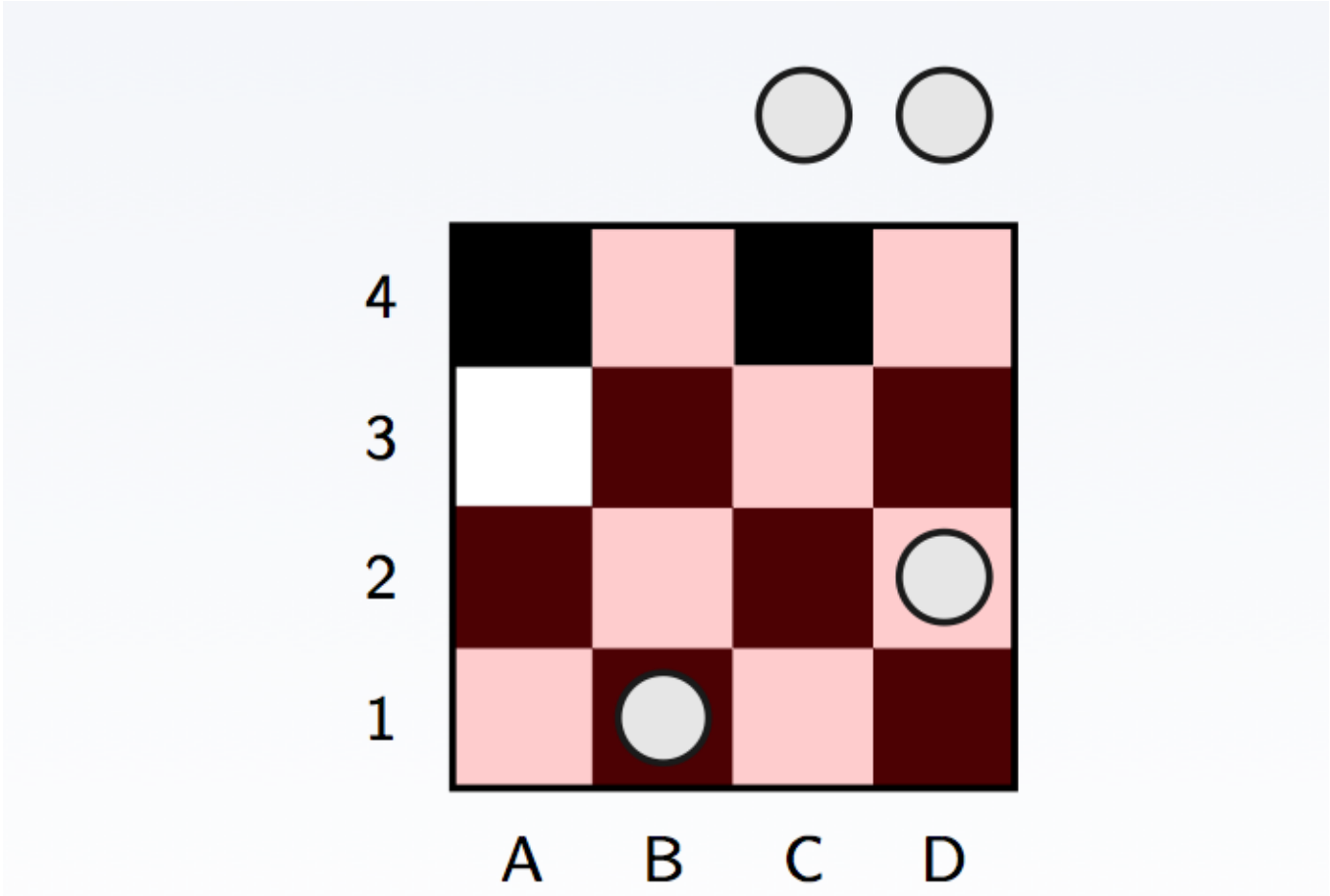
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



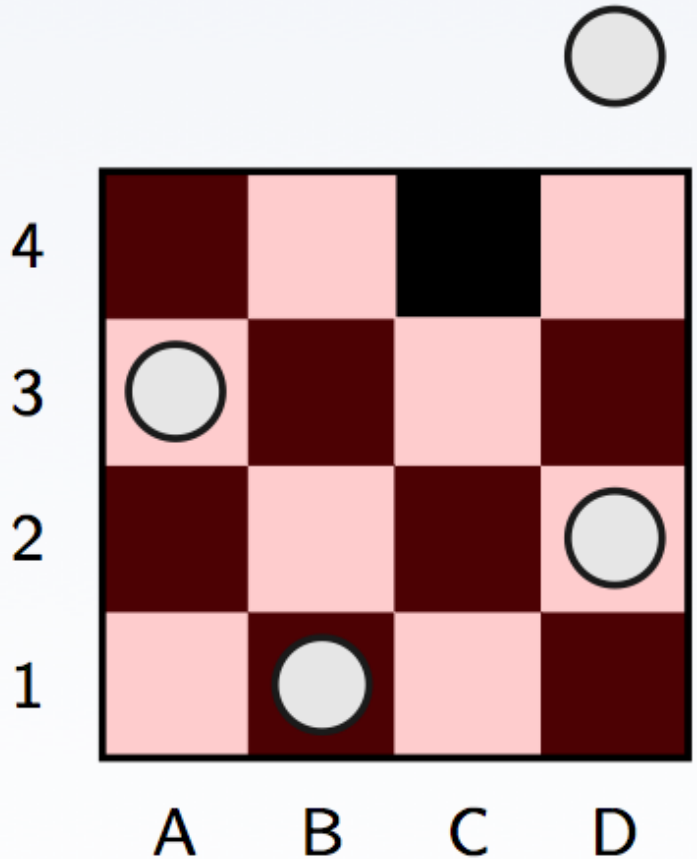
Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



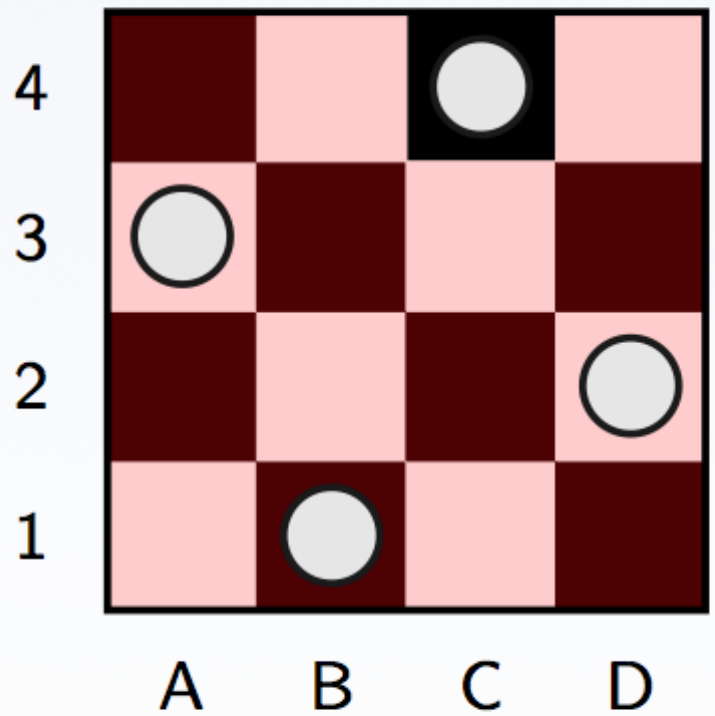
Platziere die Damen zeilenweise nacheinander, und backtracke bei Konflikt.

Beispiel: n Damen, mögliche Suche



Platziere die Damen zeilenweise nacheinander,
und backtracke bei Konflikt.

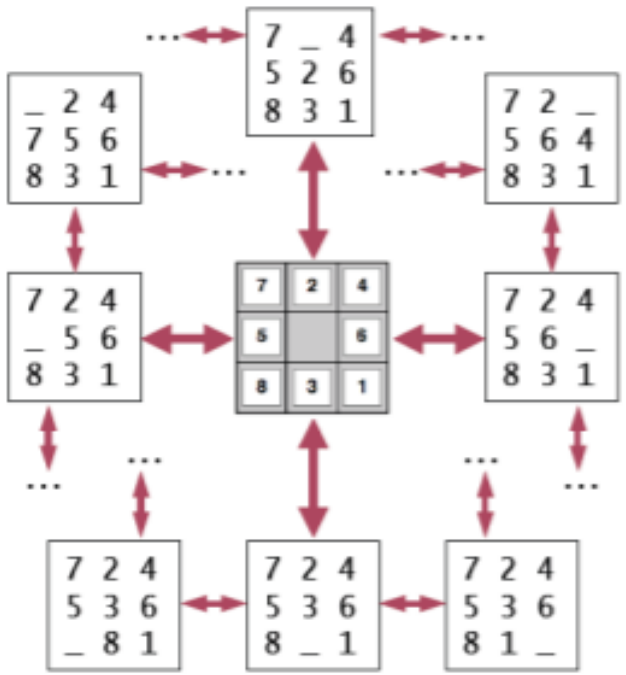
Beispiel: n Damen, mögliche Suche



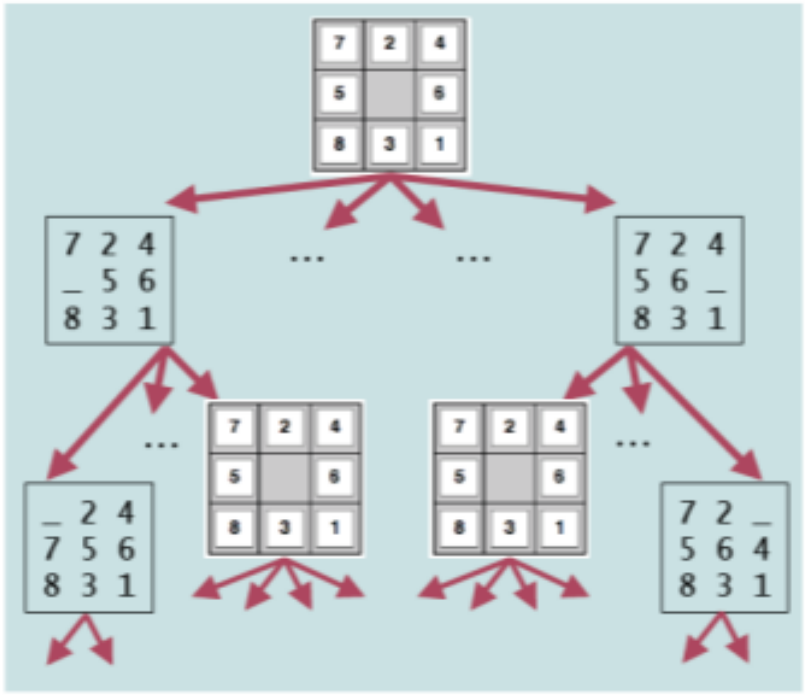
Platziere die Damen zeilenweise nacheinander, und backtracke bei Konflikt.

Suche nach Lösungen

Problemraum als Graph



Suchraum als Baum



Suche: Traversiere Graph als Baum gemäß einer Strategie

- Wir können uns den Problemraum als gerichteten Graphen vorstellen.
- Knoten repräsentieren Zustände und Kanten Aktionen. Hier sehen wir auf der linken Seite einen Ausschnitt des Problemraums repräsentiert als Graph. Die grau eingefärbte Konfiguration in der Mitte der Graphik ist unser Anfangszustand.
- Auf der rechten Seite sehen wir einen Suchbaum. Knoten repräsentieren besuchte Zustände und Kanten angewendete Aktionen. Die Wurzel repräsentiert den Anfangszustand.
- Suche bedeutet, dass wir den Graphen als Suchbaum gemäß einer bestimmten Strategie traversieren.
- Sehr wichtige Bemerkung: Weder der Graph noch der vollständige Suchbaum liegen i.a. explizit vor. Der Suchbaum wird während der Suche schrittweise aufgebaut.

Informelle Beschreibung eines Suchbaum-Algorithmus:

1. Form a one-node tree consisting of the initial state as root.
2. Color root as white.
3. Until there is no white leaf
   a) Select next white leaf as current leaf according to a strategy
   b) If the current leaf represents the goal state, return success.
   c) Color current leaf as grey
   d) Expand current leaf and color its child-nodes as white leafs.
4. Return failure.

Optimierungsproblem!!!

- Intelligente Agenten sollen ihr Leistungsmaß maximieren. Dies kann vereinfacht werden, wenn der Agent ein Ziel annehmen kann und darauf abzielt, es zu erfüllen.

Beispiel:Ein Modell und ein möglicher Lösungsweg

- ein Agent ist in der Stadt Arad
- Der Leistungsmaß ist von vielen Faktoren beeinflußt (Verbesserung des Nutzen, Sehenswürdigkeiten besichtigen, Nachtleben genießen, usw), die Entscheidungen schwer macht
- Nehmen wir nun an, der Agent hat am nächsten Tag eine nicht erstattungsfähige Fahrkarte, um von Bukarest aus zu fliegen
- Zielformulierung - nach Bukarest kommen - vereinfacht das Entscheidungsproblem des Agenten erheblich

Beispiel:Ein Modell und ein möglicher Lösungsweg

- Ein Ziel ist eine Reihe von Zuständen, in denen das Ziel erfüllt ist, aber wir müssen auch andere Zustände und Handlungen akzeptieren, um uns von einem Zustand zum anderen fort zu bewegen
- Problemformulierung
  - Zustände entsprechen Großstädten
  - Aktionen könnten dem Fahren von einer Großstadt in einer andere entsprechen
- Wie finde ich einen Weg nach Bukarest findet, wenn drei Landstraßen von Arad ausgehen, eine nach Sibiu, eine nach Timisoara und eine nach Zerind?
- Wir nehmen an, dass der Agent eine Karte von Rumänien besitzt und kann mögliche Reisen erkunden (suchen), die beste auswählen und dann die Aktionen ausführen.

Problem solving

- Zielformulierung: Was wollen wir erreichen
- Problemformulierung: Wie?
- Aufgabe lösen: Wie finde ich die beste Sequenz von Handlungen, um mein Ziel zu erreichen?
- Lösung angeben: Falls ich die Handlungen weiß, was mache ich dann?

Gut formulierte Probleme

- initialer Zustand: `in(Arad)`
- eine Beschreibung möglicher Handlungen und den entsprechenden, resultierenden Zustände
  - `SUCCESSOR-FN(in(Arad))=[go(Sibiu),in(Sibiu)]`
  - definiert implizit den Zustandsraum (Menge aller erreichbarer Zustände)
  - ein Pfad: ist eine Folge von Zustände verbunden durch Handlungen als Kanten.
- Zieltest: eine Funktion, die bestimmt, ob ein Zustand ein Zielzustand ist oder nicht `in(Bucharest)`
- Pfadkosten: eine Funktion, die numerische Werte jedem Pfad zuordnet (gibt die Effizienz an)

- Eine Lösung ist eine Reihenfolge von Handlungen, die aus einem initialen Zustand zu einem Zielzustand führen
- Eine optimale Lösung ist eine Lösung mit minimalem Kosten.

Probleme

- Toy examples
- Real problems

Real problems

- Route finding problems
- Touring problems: TSP, NP hart
- Product assembly problem: Roboter, Proteindesign aus einer Aminosäuresequenz

Wiederholung Graphentheorie

- Graphen (gerichtet, ungerichtet), Wege, Bäume;
- Erreichbarkeit: Ein Knoten B heißt aus A erreichbar, falls es ein Weg existiert, der A mit B verbindet;
- Erreichbarkeitsbaum: wie können wir diese Bäume finden?

Bestimmung von Erreichbarkeitsbäume

- Die Grundlage sind sogenannte Markierungsalgorithmen
- Um aus dem Knoten A alle erreichbare Knoten zu finden, wir der Graph systematisch durchsucht und alle besuchten Knoten werden markiert.

Tremeaux-Algorithmus der Graphensuche

Gegeben Graph $G = (V, E)$ und Startknoten A

1. Markiere den Startknoten A.
2. Wähle eine Kante (i, j) mit markiertem Knoten i und unmarkiertem Knoten j; wenn es keine solche Kante gibt, beende die Bearbeitung.
3. Markiere j und setze mit Schritt 2 fort.

Ergebnis: Es sind genau die Knoten markiert, die vom Knoten A aus erreichbar sind.

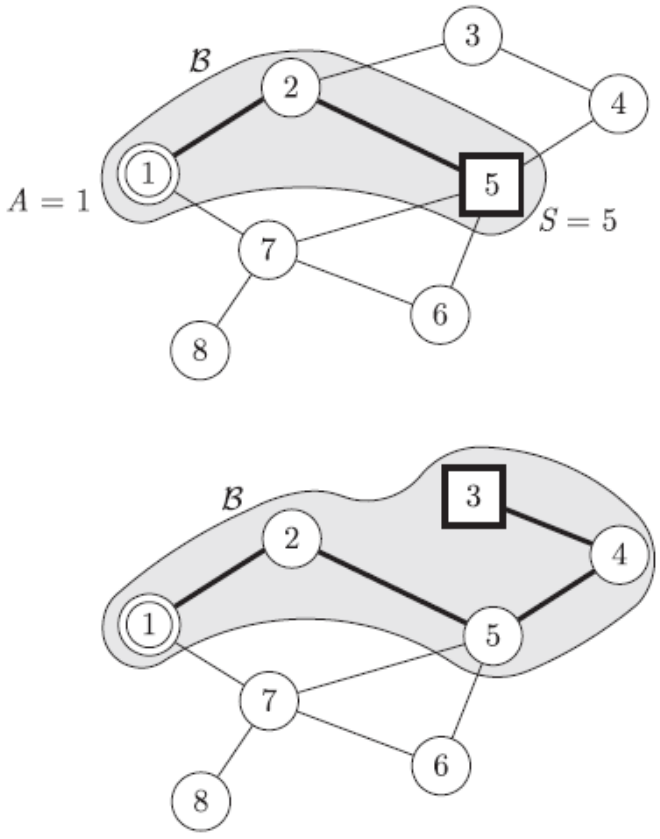
Bestimmung von Erreichbarkeitsbäume

- Sei $\mathcal{M}$ die Menge der markierten Knoten. Die wichtigste Aufgabe der Suchsteuerung besteht in der Auswahl des Suchknotens S aus der Menge $\mathcal{M}$ der zum gegenwärtigen Zeitpunkt markierten Knoten.
- Geradeaussuche, BFS, DFS, etc.

GERADEAUSSUCHE

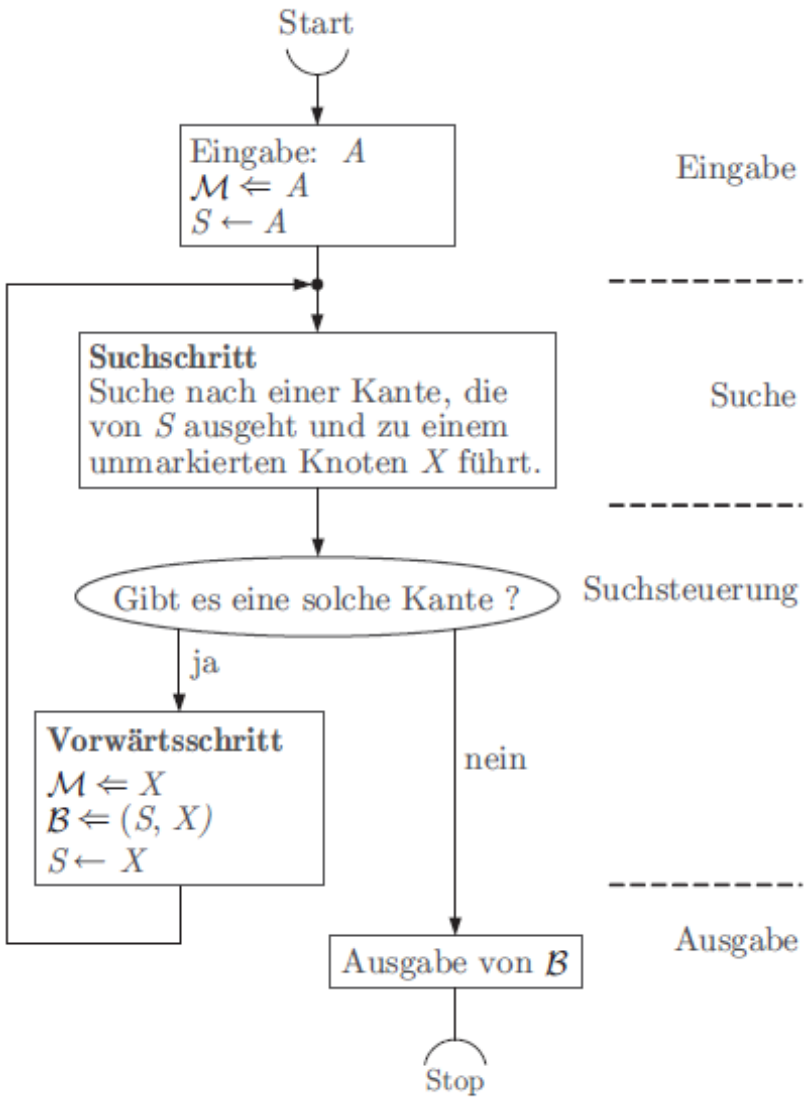
- Am Suchknoten S wird eine Kante gewählt, die zu einem noch nicht markierten Knoten X führt.
- Der Knoten X wird markiert und als neuer Suchknoten verwendet $(S \leftarrow X)$.
- Der Algorithmus endet, wenn es keine Kante gibt, die vom Suchknoten S zu einem noch nicht markierten Knoten führt.
- S Ausgangsknoten der als Ausgangspunkt für den nächsten Suchschritt dient (Suchknoten)
- $\mathcal{M}$ Liste der markierten Knoten
- $\mathcal{B}$ Liste von Kanten, die den bereits gefundenen Teil des Erreichbarkeitsbaumes bilden.

Beispiel Geradeaussuche



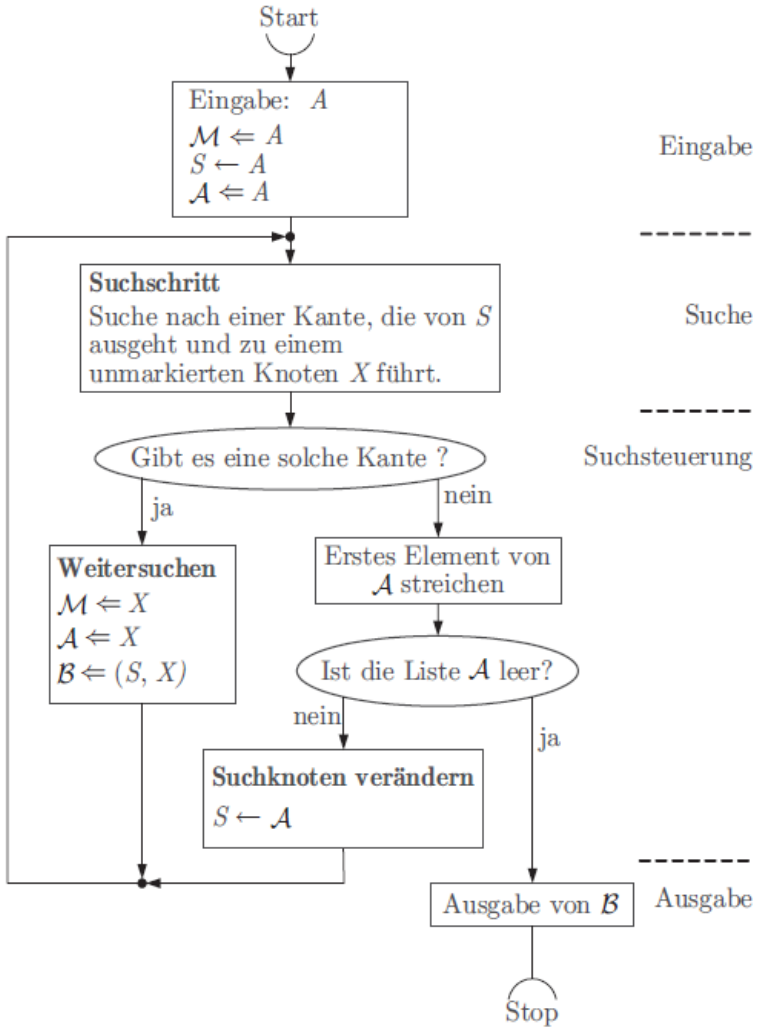
- Auf Grund seiner Einfachheit spielt die Geradeaussuche bei regelbasierten Systemen eine wichtige Rolle.
- Es muss jedoch beachtet werden, dass sie nicht garantiert, dass alle erreichbaren Knoten gefunden werden.
- Die Geradeaussuche ist nicht vollständig, denn sie erzeugt i. Allg. nur einen Teil des Erreichbarkeitsbaumes.

Algorithmus Geradeaussuche

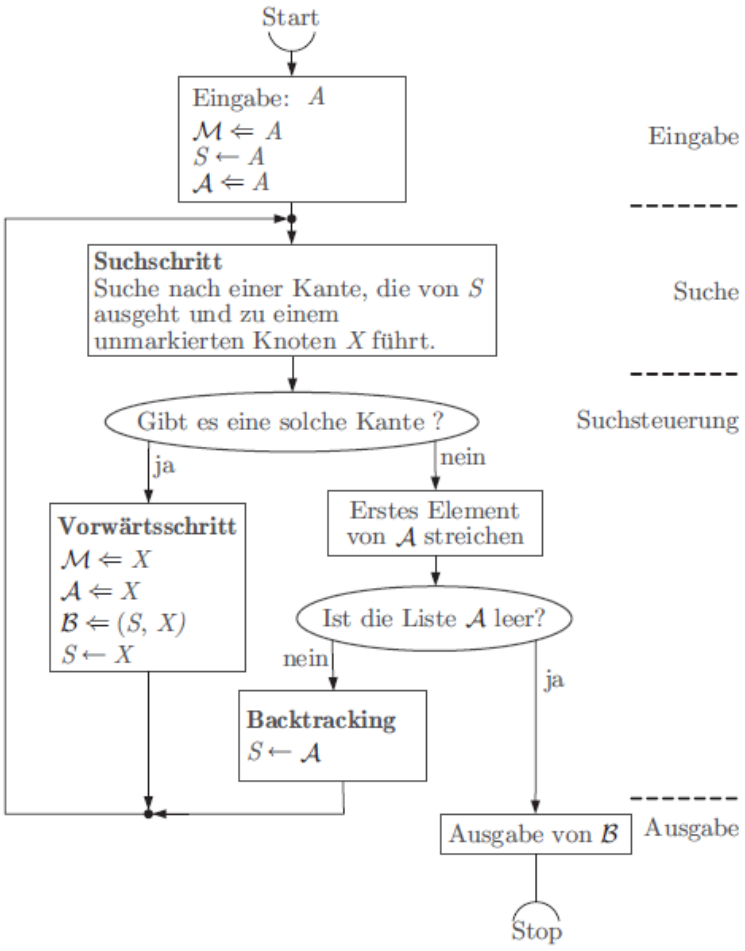


- Charakteristisch für die Geradeaussuche ist, dass die Suchsteuerung lediglich darin besteht, dass jeder neu markierte Knoten sofort als neuer Suchknoten verwendet wird.
- Die Suche ist unwiderruflich, denn diese Veränderung des Suchknotens kann später nicht wieder rückgängig gemacht werden.
- Man spricht deshalb von einer irreversiblen Suche.

Breite-zuerst-Suche



Tiefe-zuerst-Suche

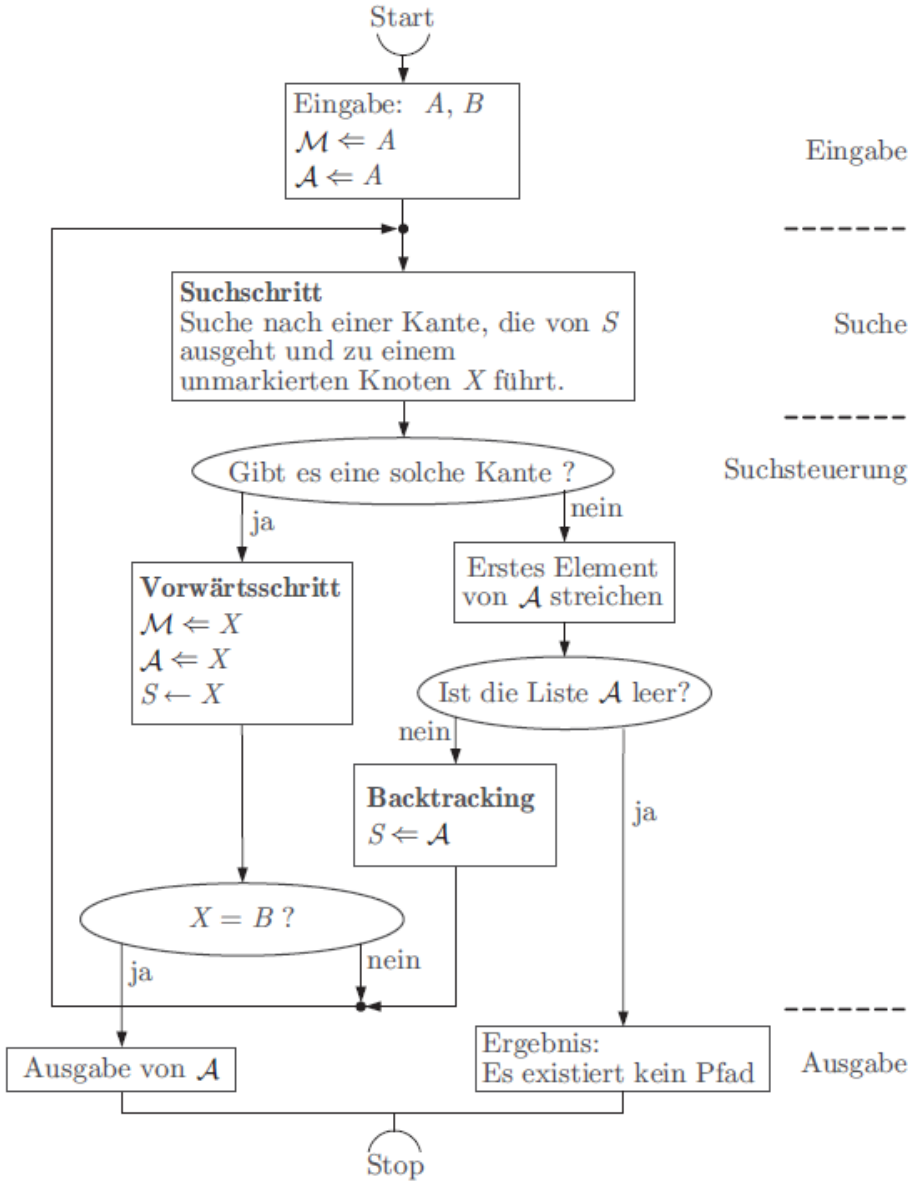


Bestimmung von Pfaden

Tiefe-zuerst-Suche von Pfaden

- Bei vielen Wissensverarbeitungsproblemen wird untersucht, wie eine gegebene in eine gewünschte Situation überführt werden kann.
- Dieser Aufgabe entspricht in der Graphentheorie das Problem, einen Pfad zu bestimmen, der einen Startknoten A mit einem Zielknoten B verbindet.

Tiefe-zuerst-Suche eines Pfades



Optimale Pfade

- Bei vielen Suchproblemen gibt es mehrere Lösungen, denn zwei Knoten A und B sind häufig über mehrere unterschiedliche Pfade $P(A, B)$ miteinander verbunden.
- Die Menge dieser Pfade wird mit $\mathcal{P}(A, B)$ bezeichnet.
- Das Suchproblem wird deshalb häufig so gestellt, dass nicht ein beliebiger, sondern der kürzeste Pfad $P^*(A, B)$ zu bestimmen ist, wobei unter der Länge $|P(A, B)|$ des Pfades die Zahl der Kanten (oder Knoten) des Pfades verstanden wird.

Erweitertes Suchproblem

- In Verallgemeinerung der Länge $|P(A,B)|$ kann eine Kostenfunktion $k_P(P(A,B))$ definiert werden, die sich als Summe der Kosten $k(i,j)$ der im Pfad $P(A,B)$ enthaltenen Kanten (i,j) zusammensetzt:

$$k_P(P(A,B)) = \sum_{(i,j)\in P(A,B)} k(i,j).$$

- Das Suchproblem besteht darin, den Pfad zwischen A und B mit den kleinsten Kosten zu bestimmen und folglich das Optimierungsproblem

$$k_P^*(A,B) = \min_{P(A,B)\in\mathcal{P}(A,B)} k_P(P(A,B))$$

zu lösen.

- Optimaler Pfad $P^*(A,B)$, optimaler Kost $k_P^*(A,B)$.

Dijkstra-Algorithmus

- Für jeden neu gefundenen Knoten X wird die Entfernung

$$g(X) = k_P(P(A, X))$$

  des Knotens vom Pfadanfang A bestimmt.
- Die Liste enthält neben den aktiven Knoten X auch deren Bewertung $g(X)$.
- Sie ist so organisiert, dass der Knoten mit der kleinsten Bewertung $g(X)$ an der ersten Position steht.
- Im Unterschied zur Breite-zuerst- oder Tiefe-zuerst-Suche ist die Liste $\mathcal{A}$ jetzt also nicht mehr als Warteschlange oder Stack organisiert, sondern als sortierte Liste, in der die Elemente X ihrer Bewertung $g(X)$ entsprechend angeordnet sind.
- Der Suchgraph wird an dem Knoten erweitert, der den kürzesten Abstand vom Startknoten A hat.

Prinzip des Dijkstra-Algorithmus

- Während es bei der Pfadsuche bisher gleichgültig war, welcher Pfad $P(A, B)$ vom Wurzelknoten A zu einem erreichbaren Knoten B im Erreichbarkeitsbaum $\mathcal{B}$ enthalten war, kommt es jetzt darauf an, die Kanten für den Baum $\mathcal{B}$ so auszuwählen, dass die Pfadkosten minimal sind.
- Aus diesem Grund können Kanten, über die unmarkierte Knoten gefunden werden, nicht mehr ohne weitere Prüfung in den Lösungsbaum $\mathcal{B}$ eingetragen werden.
- Der Lösungsbaum $\mathcal{B}$ enthält jetzt nicht mehr alle markierten Knoten, sondern nur die passiven Knoten und Kanten zwischen passiven Knoten

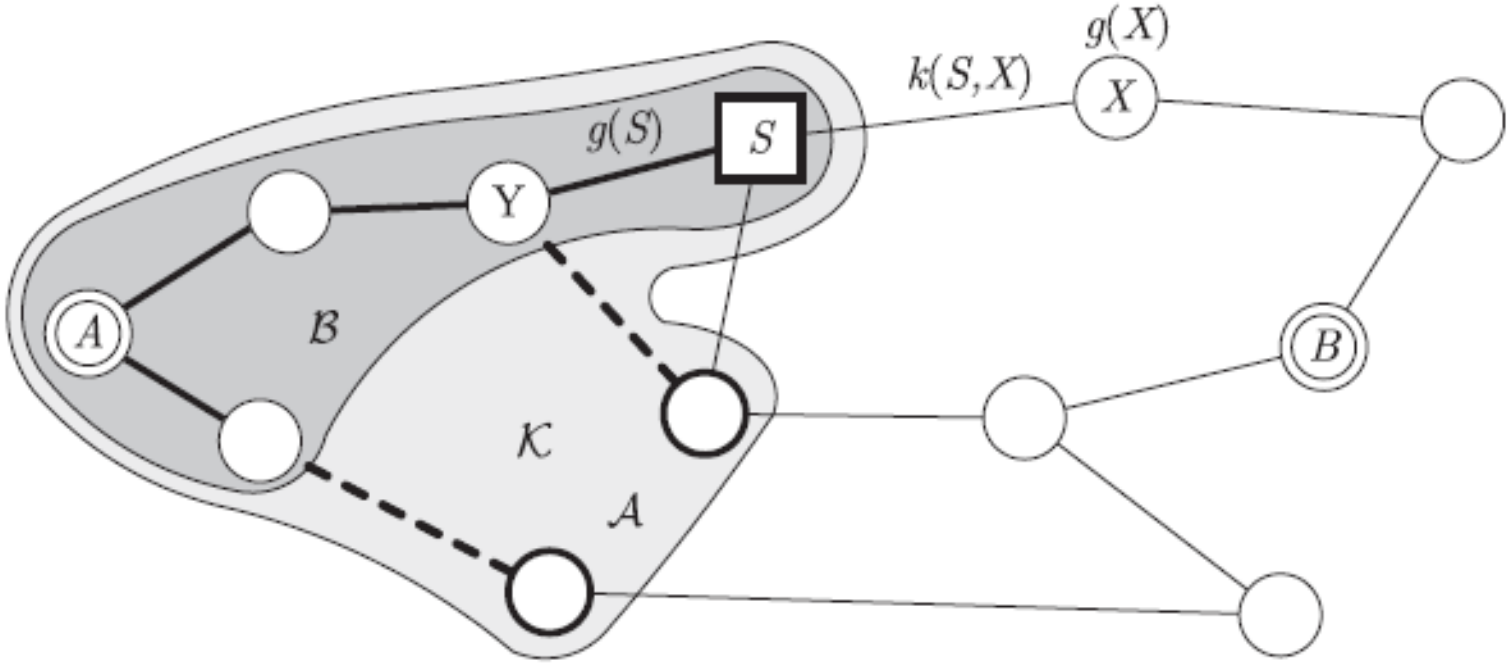
Prinzip des Dijkstra-Algorithmus

- Unmarkierte Knoten: Knoten X, die noch nicht untersucht wurden und für die deshalb nicht bekannt ist, ob es einen Pfad $P(A, X)$ gibt.
- Aktive Knoten: Knoten X, für die ein Pfad $P(A, X)$ und dessen Länge $g(X)$ bekannt sind, aber der Pfad $P(A, X)$ muss nicht der kürzeste Pfad sein.
- Passive Knoten: Knoten X, für die ein kürzester Pfad $P^*(A, X)$ bekannt ist.

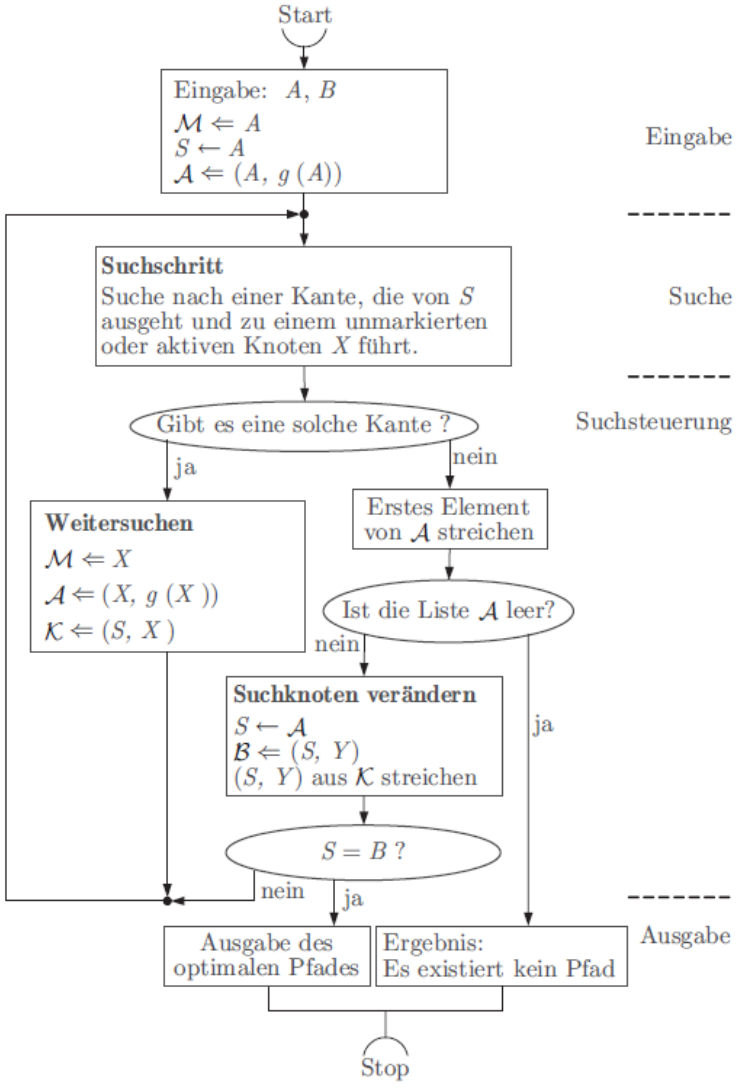
Prinzip des Dijkstra-Algorithmus

- Der Algorithmus speichert die Zwischenergebnisse dementsprechend in drei Listen:
- $\mathcal{B}$ - Baum mit dem Wurzelknoten A, in dem jeder Knoten X mit A über einen kürzestmöglichen Pfad $P^*(A, X)$ verbunden ist (in der Abbildung durch dick gezeichnete Kanten dargestellt). $\mathcal{B}$ enthält alle passiven Knoten.
- $\mathcal{A}$ - Liste der aktiven Knoten. Von diesen Knoten, die in der Abbildung dick gezeichnet in der hellgrauen Fläche liegen, wird die Suche fortgesetzt.
- $\mathcal{K}$ - Liste der Kanten zwischen passiven und aktiven Knoten, die in der Abbildung gestrichelt gezeichnet sind. Für diese Kanten muss noch überprüft werden, ob sie zum Lösungsbaum gehören.

Prinzip des Dijkstra-Algorithmus



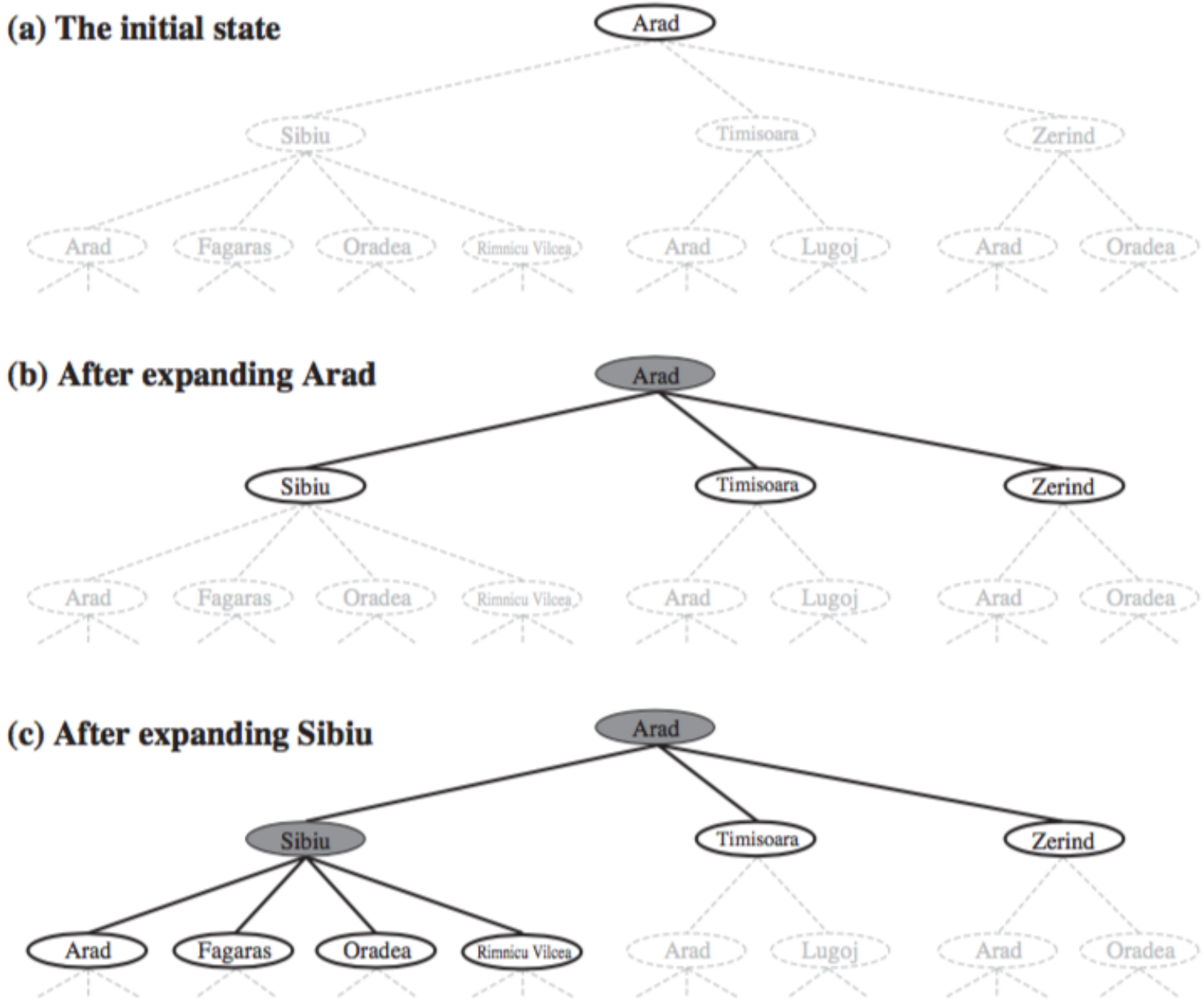
Dijkstra-Algorithmus zur Bestimmung optimaler Pfade



Wie wird eine Lösung gefunden?

- State space search!!!
- Starte in initialen Knoten (root)
- Teste ob der initiale Knoten ein Zielknoten ist
- Falls der aktuelle Zustand kein Zielzustand ist, expandiere den Zustand und erzeuge neue Zustände
- Selektiere einen neuen Zustand mit Hilfe einer Suchstrategie

PROBLEM SOLVING



PROBLEM SOLVING

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

PROBLEM SOLVING

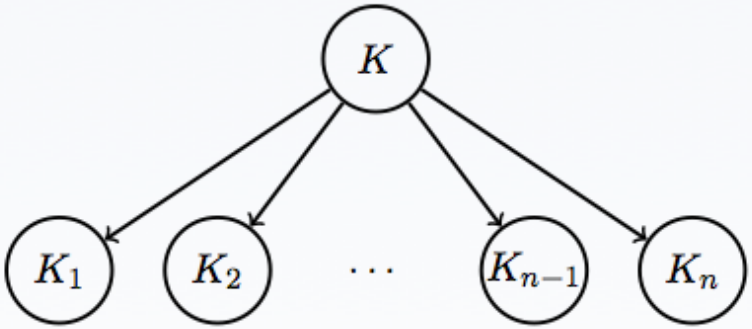
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Suchraum/Suchgraph

- Suche nach Lösung = Suche in einem gerichten Graphen
  $\rightarrow$ Der Graph ist nicht explizit gegeben!
- Suchgraph (Suchraum) gegeben durch:
  - Knoten: Situation, Zustände
  - Kanten: implizit als Nachfolger-Funktion N
  - Anfangssituation
  - Zieltest: Entscheidbarer Test, ob Knoten Zielknoten

Eigenschaften

- Verzweigungrate des Knotens K (branching factor):
  Anzahl der direkten Nachfolger von K, also $|N(K)|$.



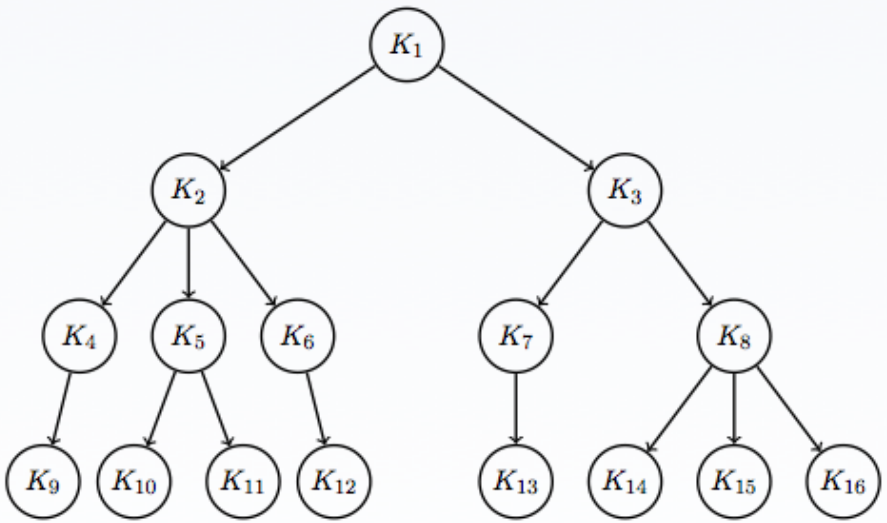
$$N(K) = \{K_1, \ldots, K_n\}$$
$$|N(K)| = n$$

- Mittlere Verzweigunsrate des Suchraumes:
  Durchschnittliche Verzweigungrate aller Knoten.

Eigenschaften

- Größe des Suchraumes ab Knoten K in Tiefe d:
  Anzahl Knoten, die von K aus in d Schritten erreichbar sind
  D.h. $|\overline{N}^d(K)|$, wobei

$$\overline{N}^1(M) = \bigcup\{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$

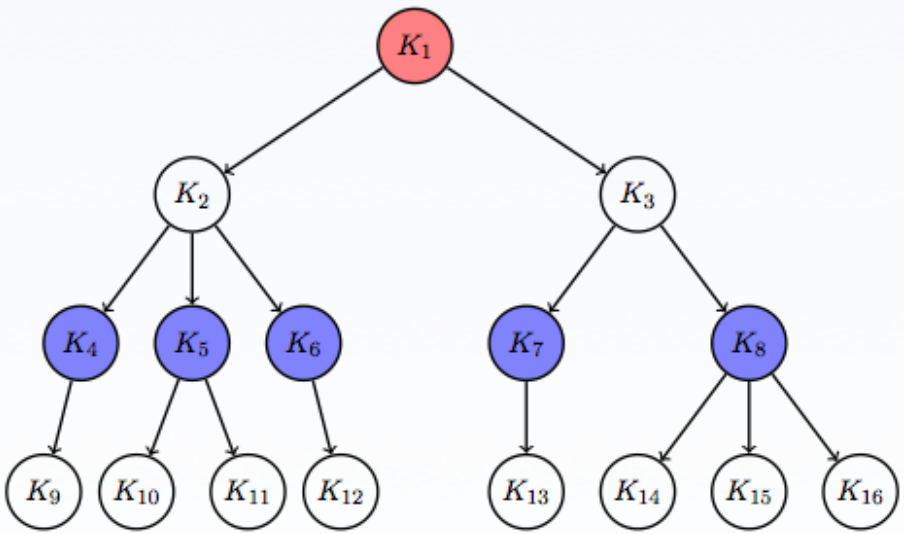


Eigenschaften

- Größe des Suchraumes ab Knoten K in Tiefe d:
  Anzahl Knoten, die von K aus in d Schritten erreichbar sind

  D.h. $|\overline{N}^d(K)|$, wobei

$$\overline{N}^1(M) = \bigcup\{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$



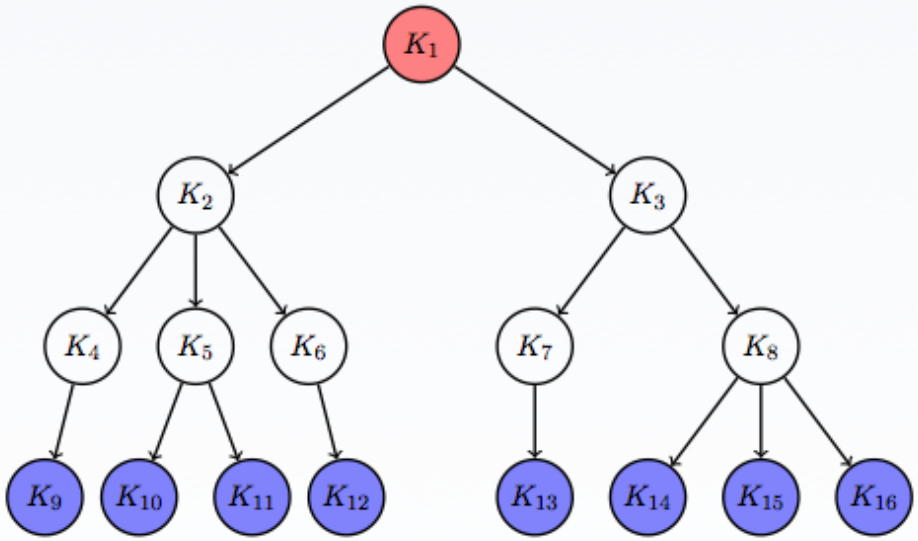
- Größe des Suchraums ab K_1 in Tiefe $2 = 5$

Eigenschaften

- Größe des Suchraumes ab Knoten K in Tiefe d:
  Anzahl Knoten, die von K aus in d Schritten erreichbar sind

  D.h. $|\overline{N}^d(K)|$, wobei

$$\overline{N}^1(M) = \bigcup\{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$



- Größe des Suchraums ab K_1 in Tiefe $2 = 5$
- Größe des Suchraums ab K_1 in Tiefe $3 = 8$

Eigenschaften

Eine Suchstragie ist vollständig, wenn sie einen Zielknoten nach endlichen vielen Schritten findet, falls dieser existiert.

Kombinatorische Explosion

- Üblicherweise: mittlere Verzweigungrate > 1
- $\Rightarrow$ Suche ist expontiell in der Tiefe des Suchraums
- das nennt man: kombinatorische Explosion
- Die meisten Suchprobleme sind NP-hart (NP-vollständig)

Knoten vs. Zustand

Zustandsraum ist vom Suchgraphen verschieden!

- Ein Knoten besteht aus
  - aktueller Zustand
  - eine Verbindung zum Elternknoten
  - eine Handlug, die zum aktuellen Zustand geführt hat
  - Pfadkosten vom root zum Knoten
  - Tiefe (Anzahl der schritte vom root)

Blind Search

- Blind Search = Nicht-informierte Suche
- Nur der Suchgraph ist (implizit) gegeben
- keine anderen Informationen (z.B. Heuristik)

Eingabe:

- Menge der initialen Knoten
- Menge der Zielknoten, bzw. eindeutige Festlegung der Eigenschaften der Zielknoten
- Nachfolgerfunktion N

Ausgabe:

Pfad zum Zielknoten (falls dieser existiert)

Nicht-informierte Suche, allgemein

Algorithmus Nicht-informierte Suche

Datenstrukturen:L = Menge von Knoten, markiert Weg dorthin
Eingabe: Setze L := Menge der initialen Knoten mit leerem Weg
Algorithmus:

1. Wenn L leer ist, dann breche ab.
2. Wähle einen beliebigen Knoten K aus L.
3. Wenn K ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn K kein Zielknoten, dann nehme Menge $N(K)$ der direkten Nachfolger von K und verändere L folgendermaßen:
   $L := (L \cup N(K)) \setminus \{K\}$ (Wege entsprechend anpassen)
   Mache weiter mit Schritt 1.


Nicht-informierte Suche, allgemein

Algorithmus Nicht-informierte Suche

Datenstrukturen:L = Menge von Knoten, markiert Weg dorthin
Eingabe: Setze L := Menge der initialen Knoten mit leerem Weg
Algorithmus:

1. Wenn L leer ist, dann breche ab.
2. Wähle einen beliebigen Knoten K aus L.
3. Wenn K ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn K kein Zielknoten, dann nehme Menge $N(K)$ der direkten Nachfolger von K und verändere L folgendermaßen:
   $L := (L \cup N(K)) \setminus \{K\}$ (Wege entsprechend anpassen)
   Mache weiter mit Schritt 1.

Keine Strategie!, nichtdeterministisch

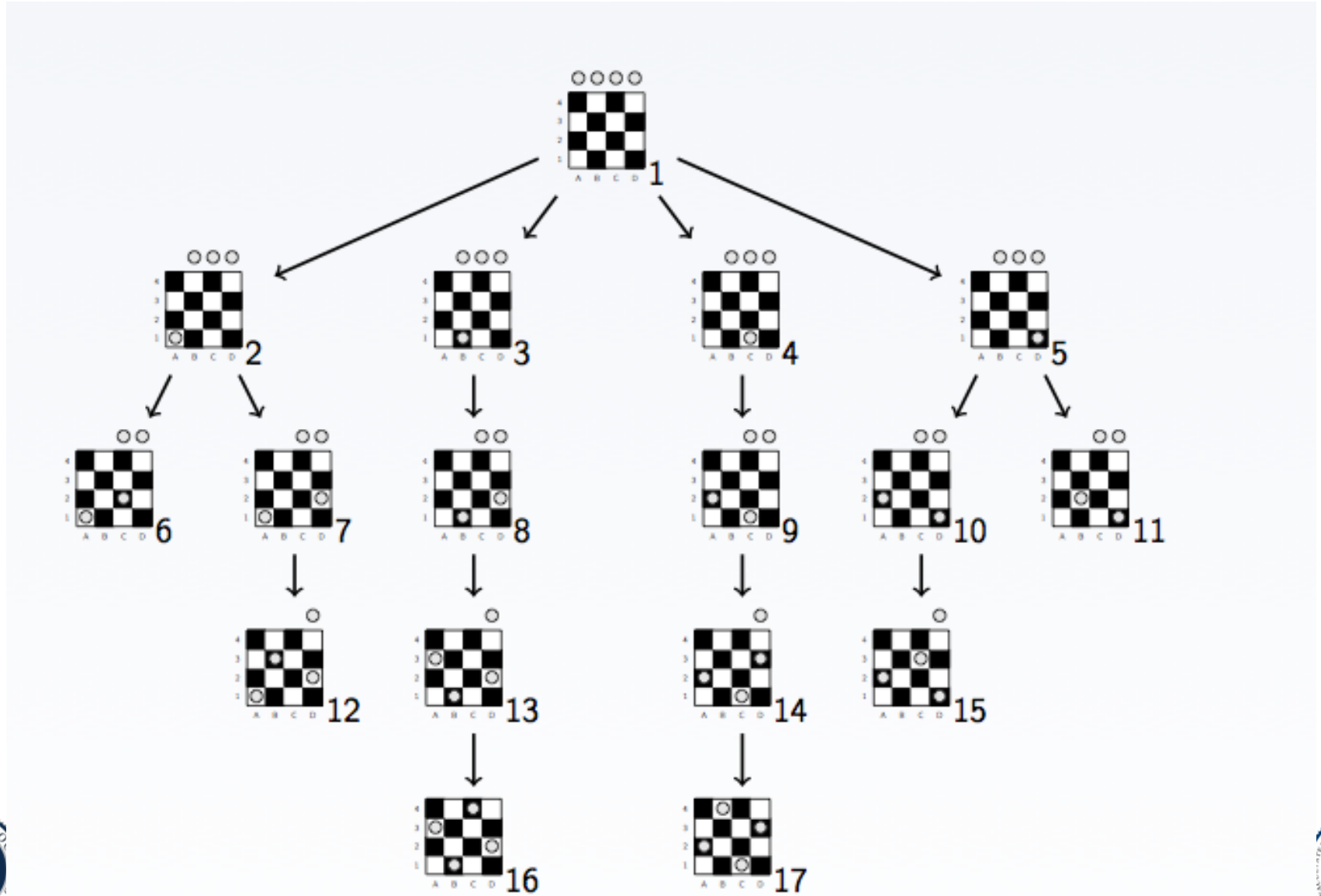

Tiefensuche

Algorithmus Tiefensuche

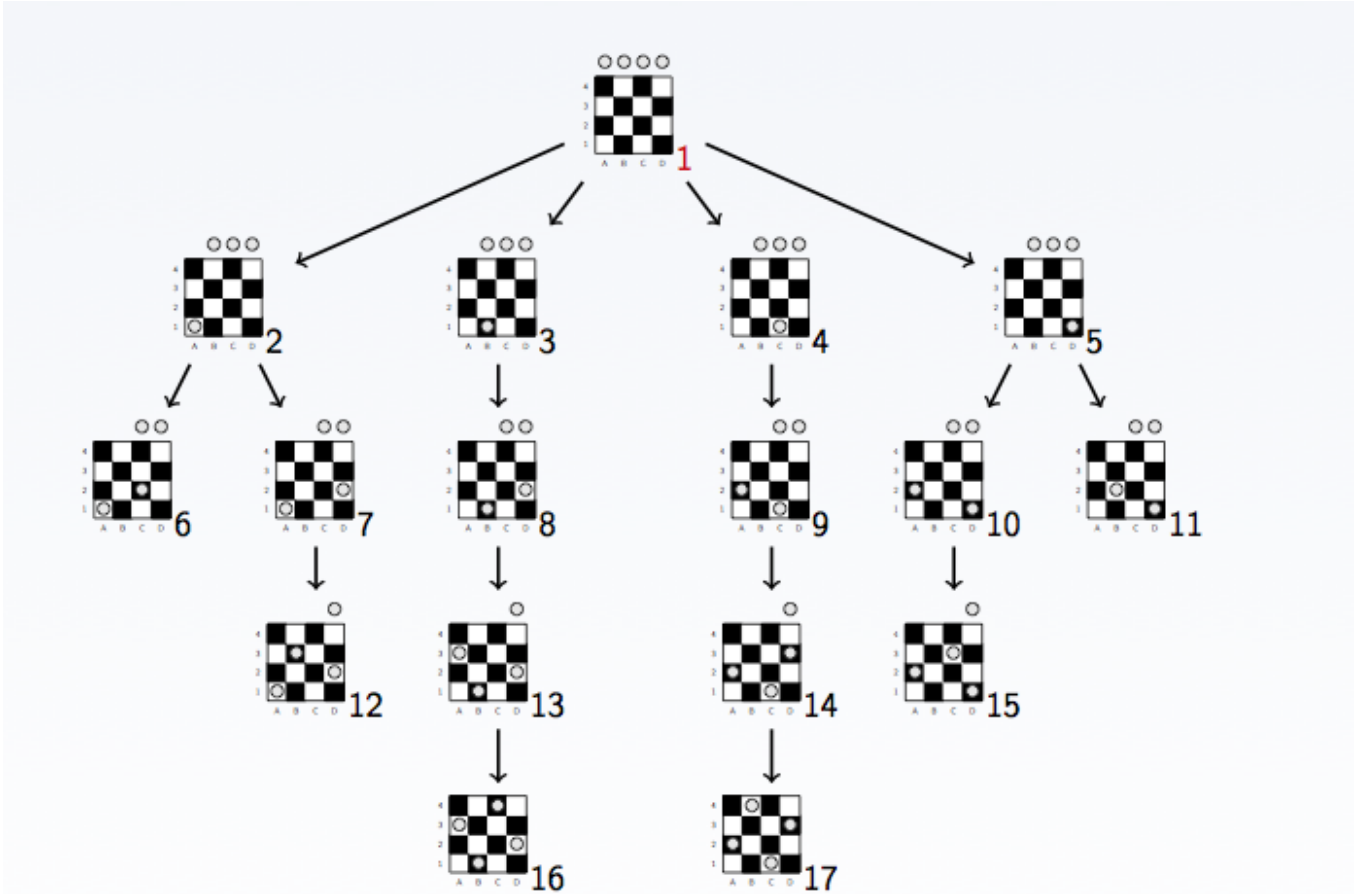
Datenstrukturen: L = Liste (Stack) von Knoten, markiert Weg dorthin
Eingabe: Füge die initialen Knoten in die Liste L ein.
Algorithmus:

1. Wenn L die leere Liste ist, dann breche ab.
2. Wähle ersten Knoten K aus L, sei R die Restliste.
3. Wenn K ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn K kein Zielknoten, dann sei $N(K)$ die (geordnete) Liste der direkten Nachfolger von K, mit dem Weg dorthin markiert
   $L := N(K) + +R$. (wobei $++$ Listen zusammenhängt)
   Mache weiter mit 1.


Beispiel Tiefensuche



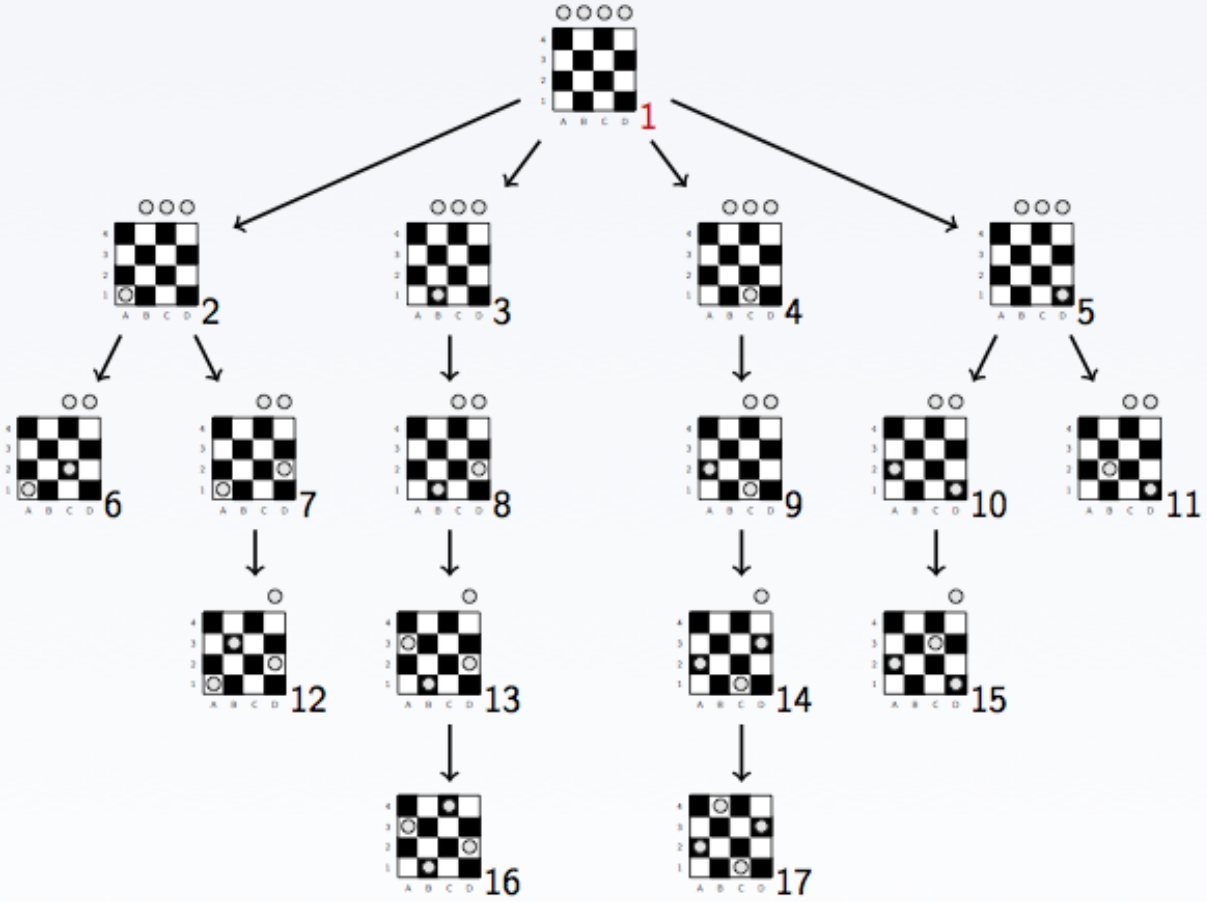
Beispiel Tiefensuche



Am Anfang:

$L := [(1, [])]$

Beispiel Tiefensuche



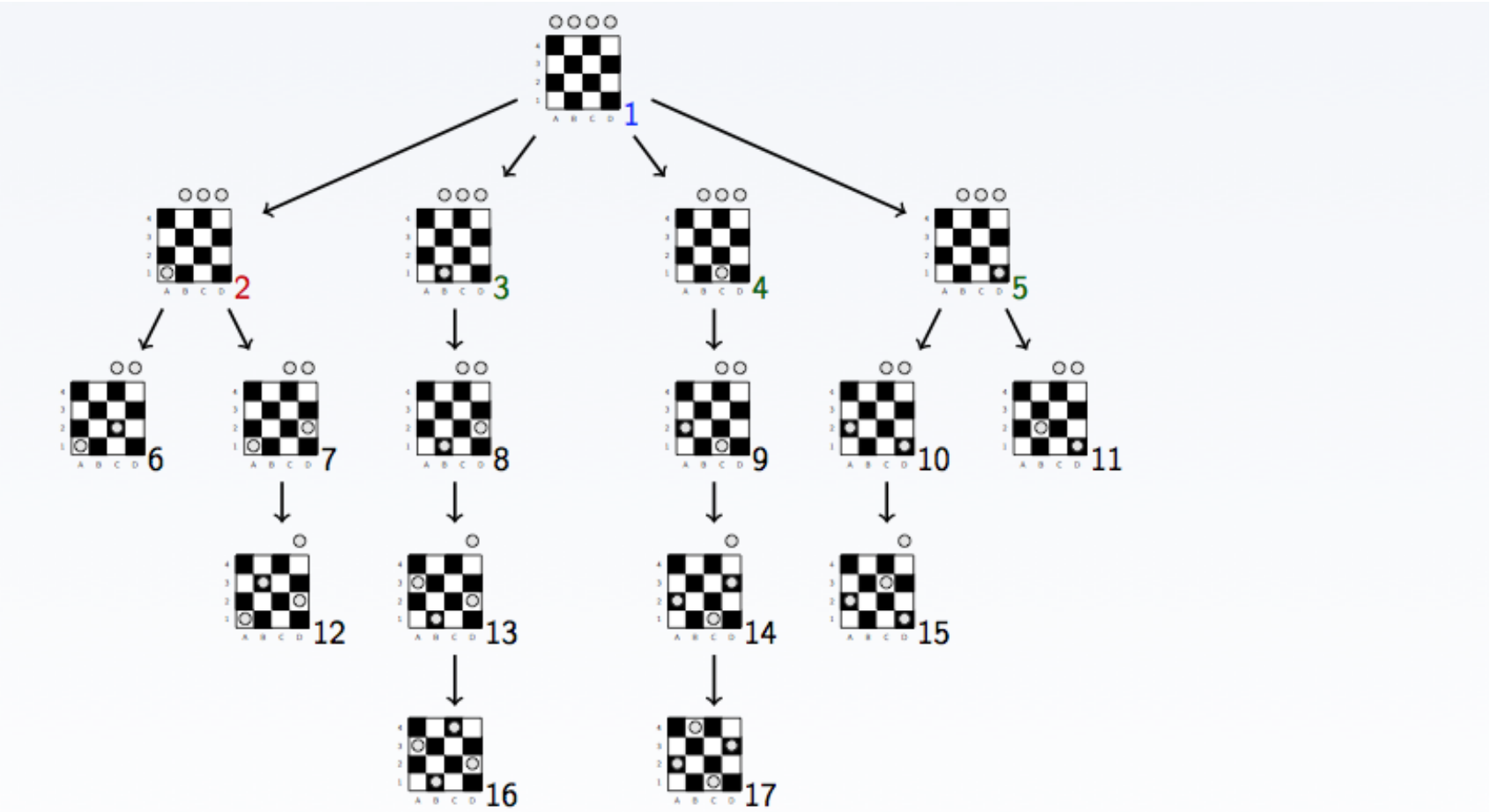
1. Knoten

$K := (1, [])$ $R := []$

$NF(K) = [2, 3, 4, 5]$

$L := [(2, [1]), (3, [1]), (4, [1]), (5, [1])] ++ R = [(2, [1]), (3, [1]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



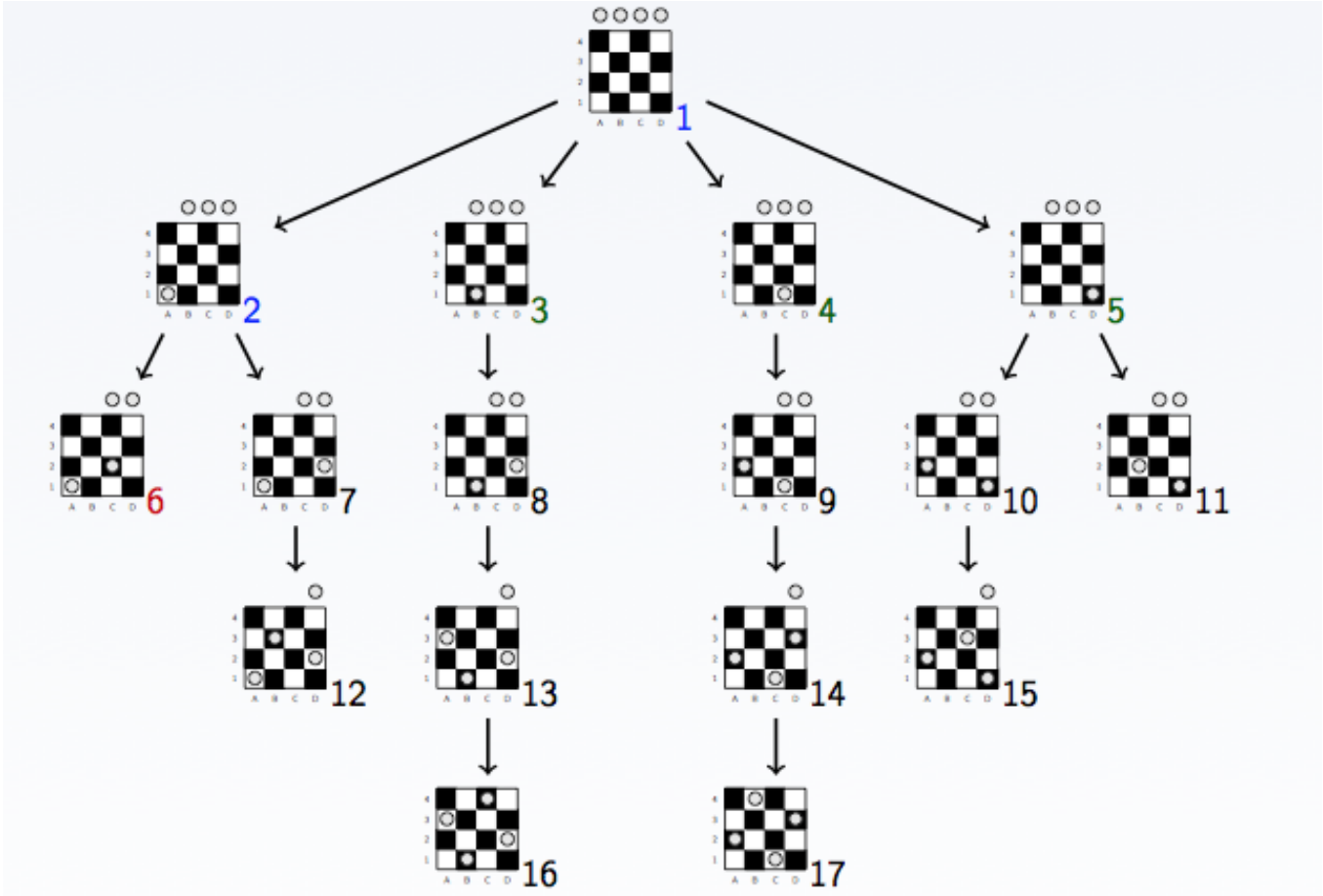
2. Knoten

$K := (2, [1])$ $R := [(3, [1]), (4, [1]), (5, [1])]$

$NF(2) = [6, 7]$

$L := [(6, [1, 2]), (7, [1, 2])]$++$R = [(6, [1, 2]), (7, [1, 2]), (3, 1), (4, 1), (5, 1)]$

Beispiel Tiefensuche



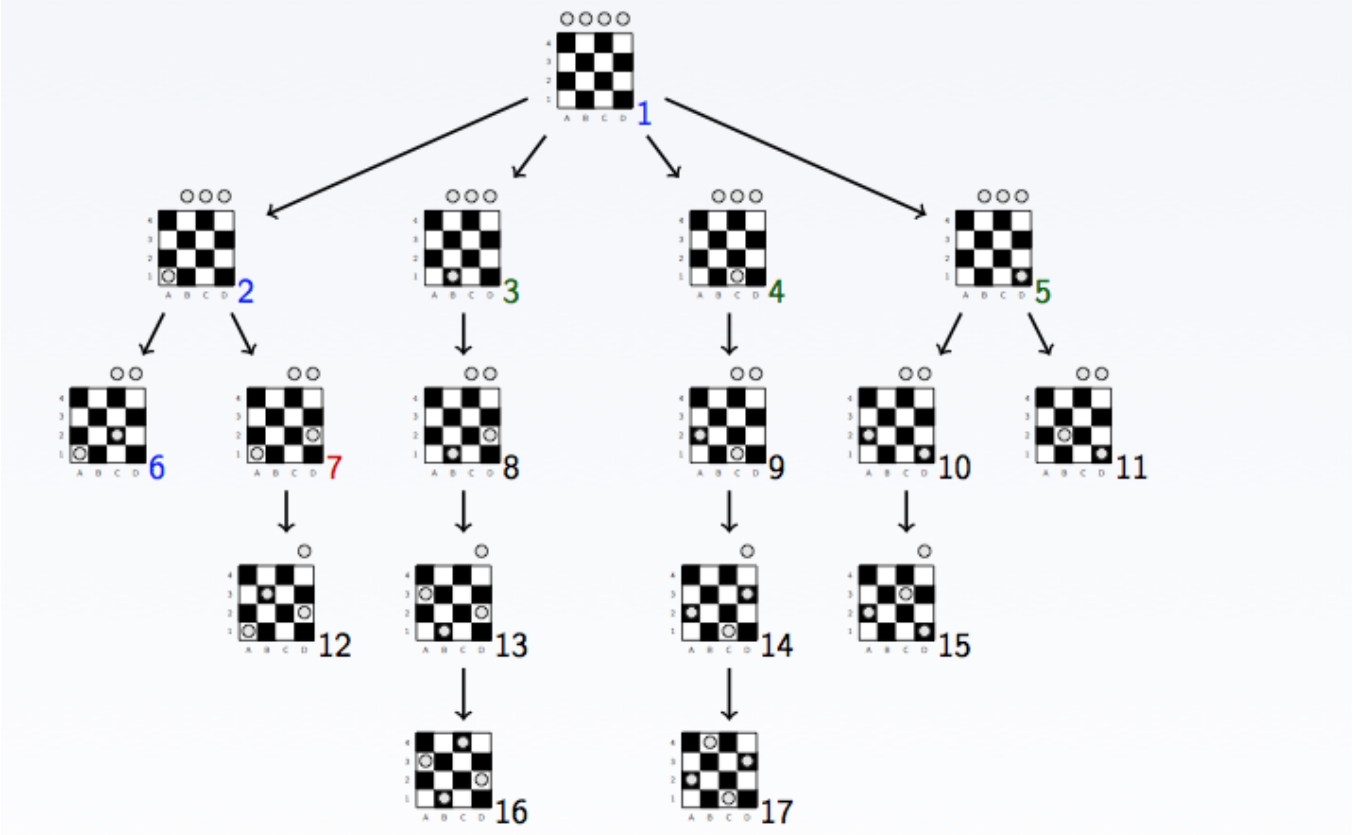
3. Knoten

$K := (6, [1, 2])$ $R := [(7, [1, 2]), (3, [1]), (4, [1]), (5, [1])]$

$NF(6) = []$

$L := []{+}{+}R = [(7, [1, 2]), (3, [1]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



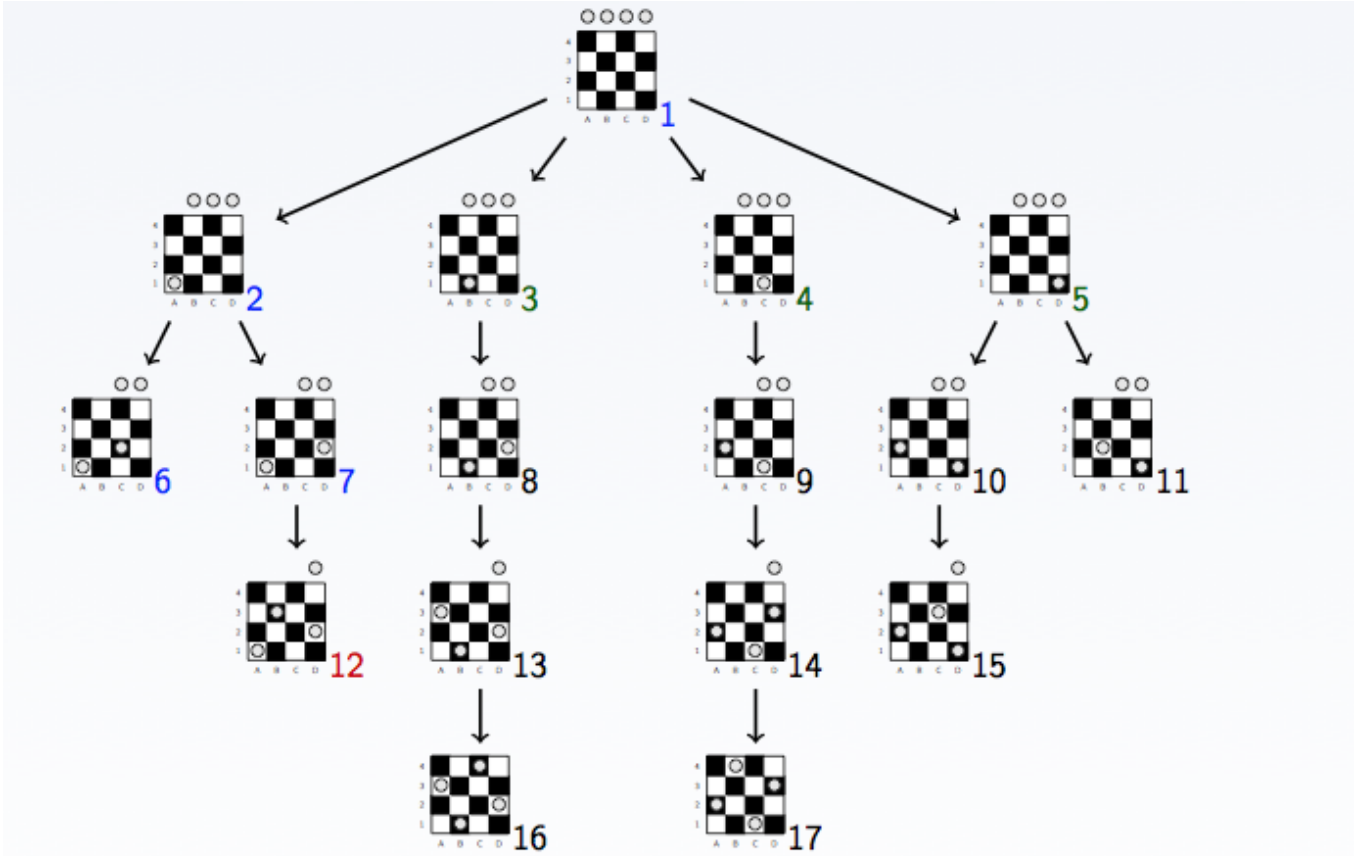
4. Knoten

$K := (7, [1, 2])$ $R := [(3, [1]), (4, [1]), (5, [1])]$

$NF(7) = [12]$

$L := [(12, [1, 2, 7])]$++$R = [(12, [1, 2, 7]), (3, [1]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



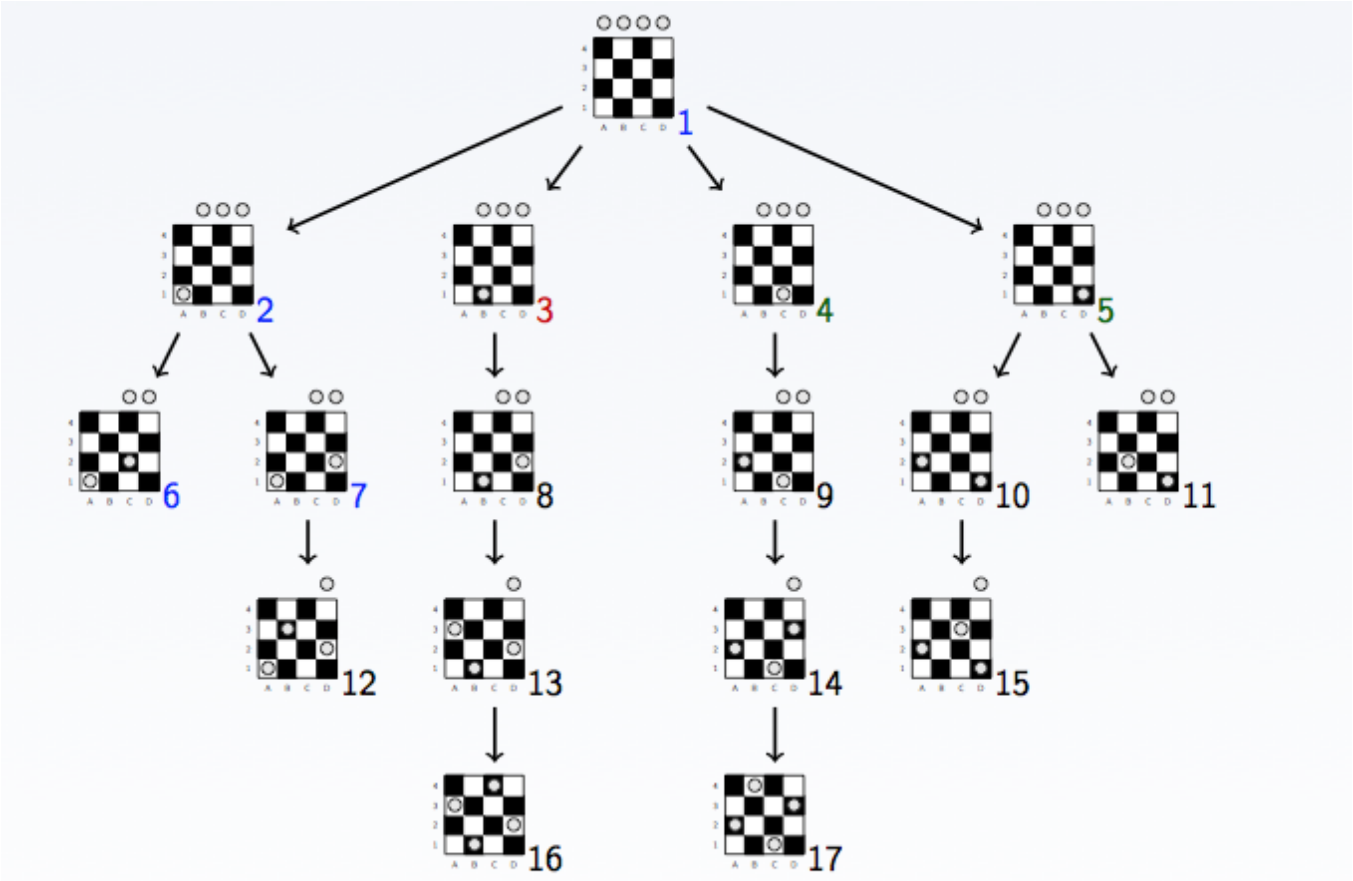
5. Knoten

$K := (12, [1, 2, 7]) \quad R := [(3, [1]), (4, [1]), (5, [1])]$

$NF(12) = []$

$L := []{+}{+}R = [(3, [1]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



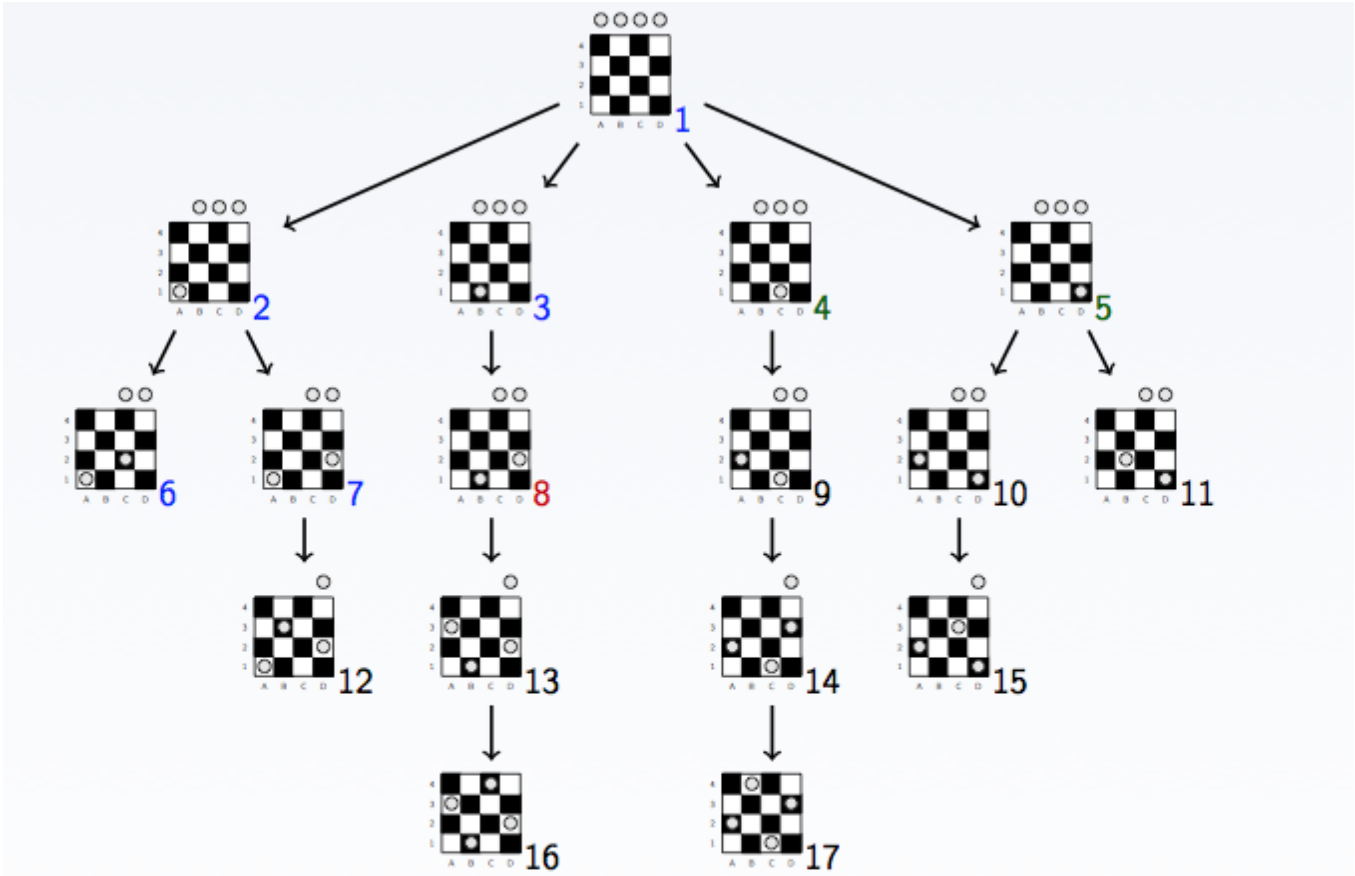
6. Knoten

$K := (3, [1]) \quad R := [(4, [1]), (5, [1])]$

$NF(3) = [8]$

$L := [(8, [1, 3])] ++ R = [(8, [1, 3]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



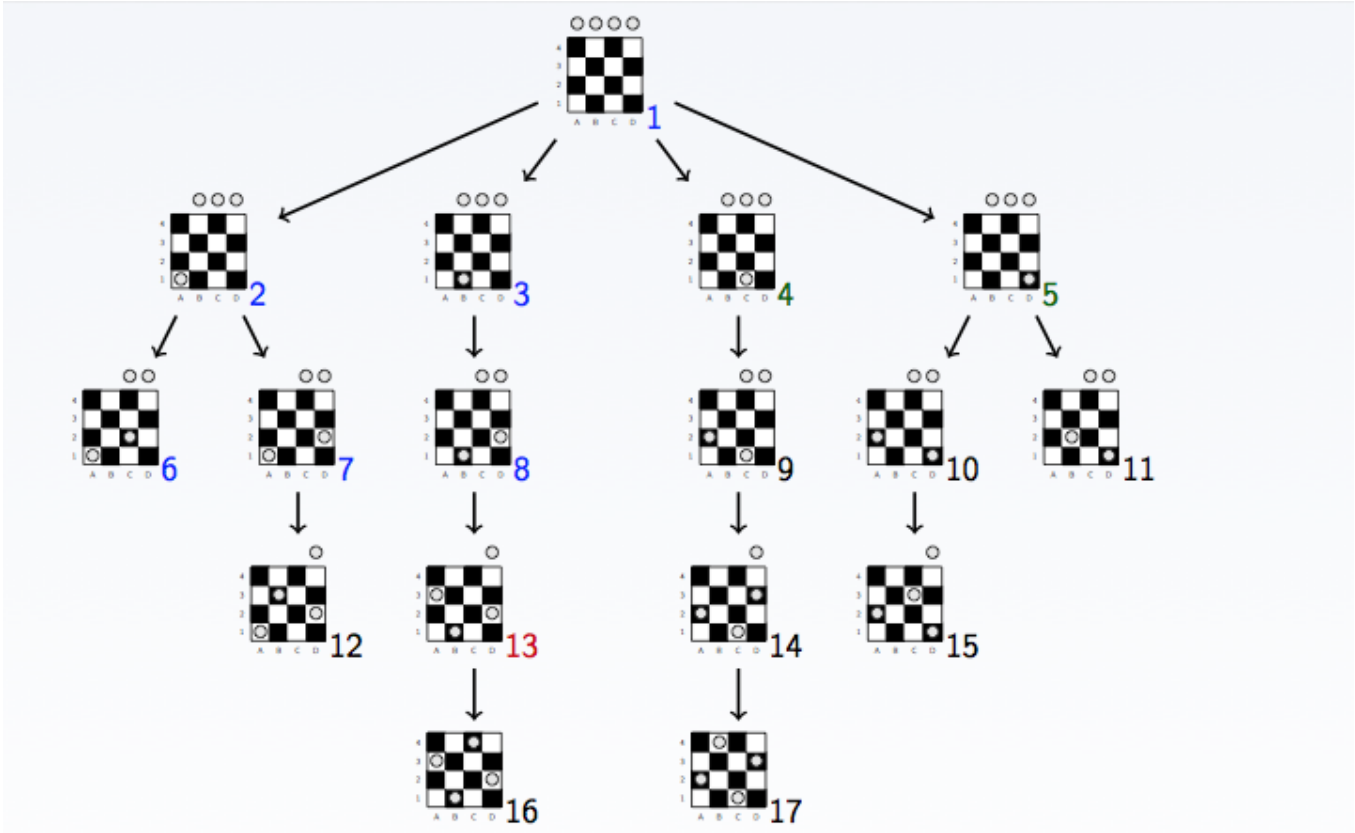
7. Knoten

$K := (8, [1, 3])$ $R := [(4, [1]), (5, [1])]$

$NF(8) = [13]$

$L := [(13, [1, 3, 8])]$++$R = [(13, [1, 3, 8]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



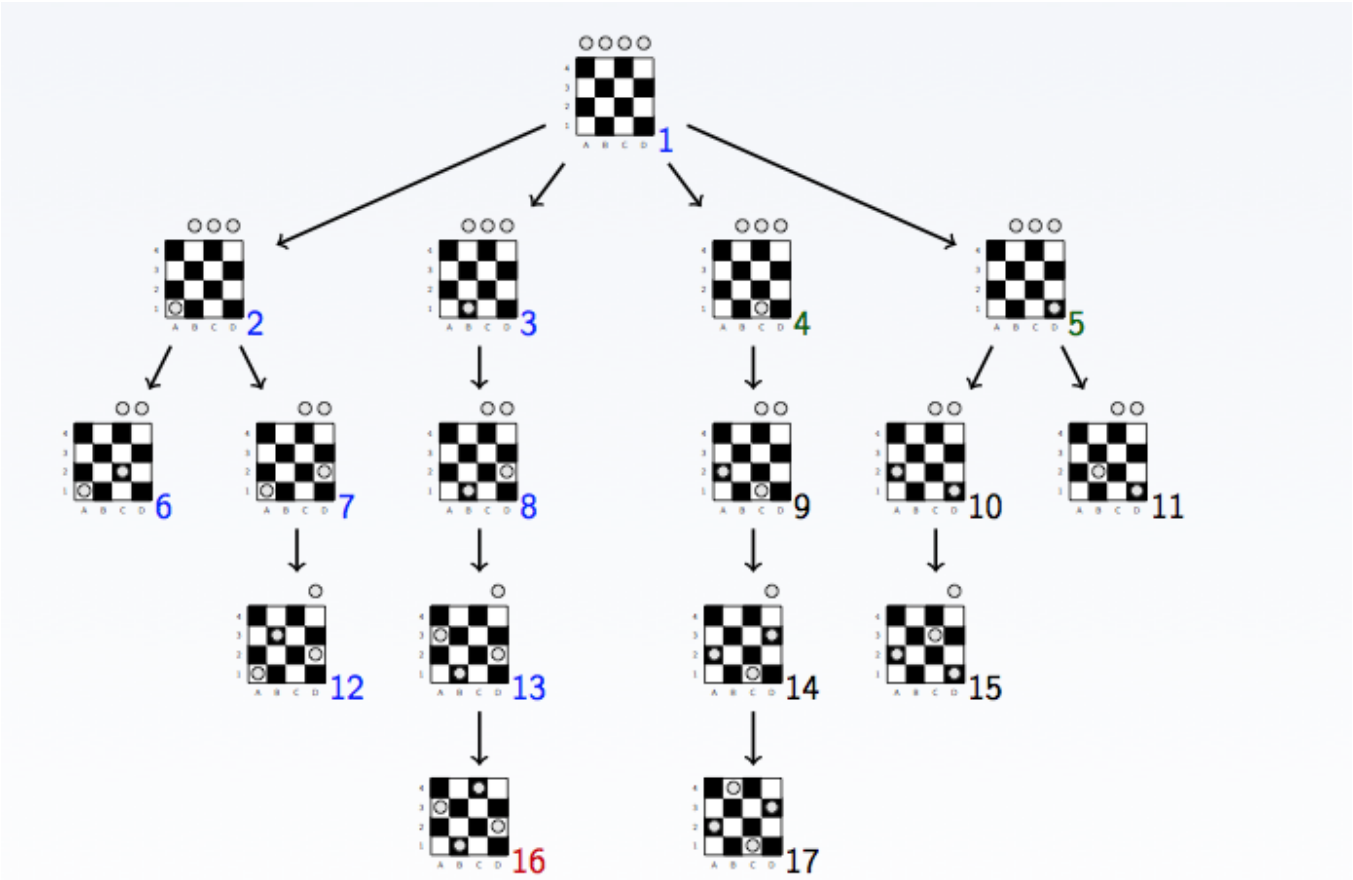
8. Knoten

$K := (13, [1, 3, 8])$ $R := [(4, [1]), (5, [1])]$

$NF(13) = [16]$

$L := [(16, [1, 3, 8, 13])]$++$R = [(16, [1, 3, 8, 13]), (4, [1]), (5, [1])]$

Beispiel Tiefensuche



9. Knoten

$K := (16, [1, 3, 8, 13])$ $R := [(4, [1]), (5, [1])]$

$Ziel(16) ==$ `True` $\Rightarrow$ gebe 1,3,8,13,16 aus

Eigenschaften der Tiefensuche

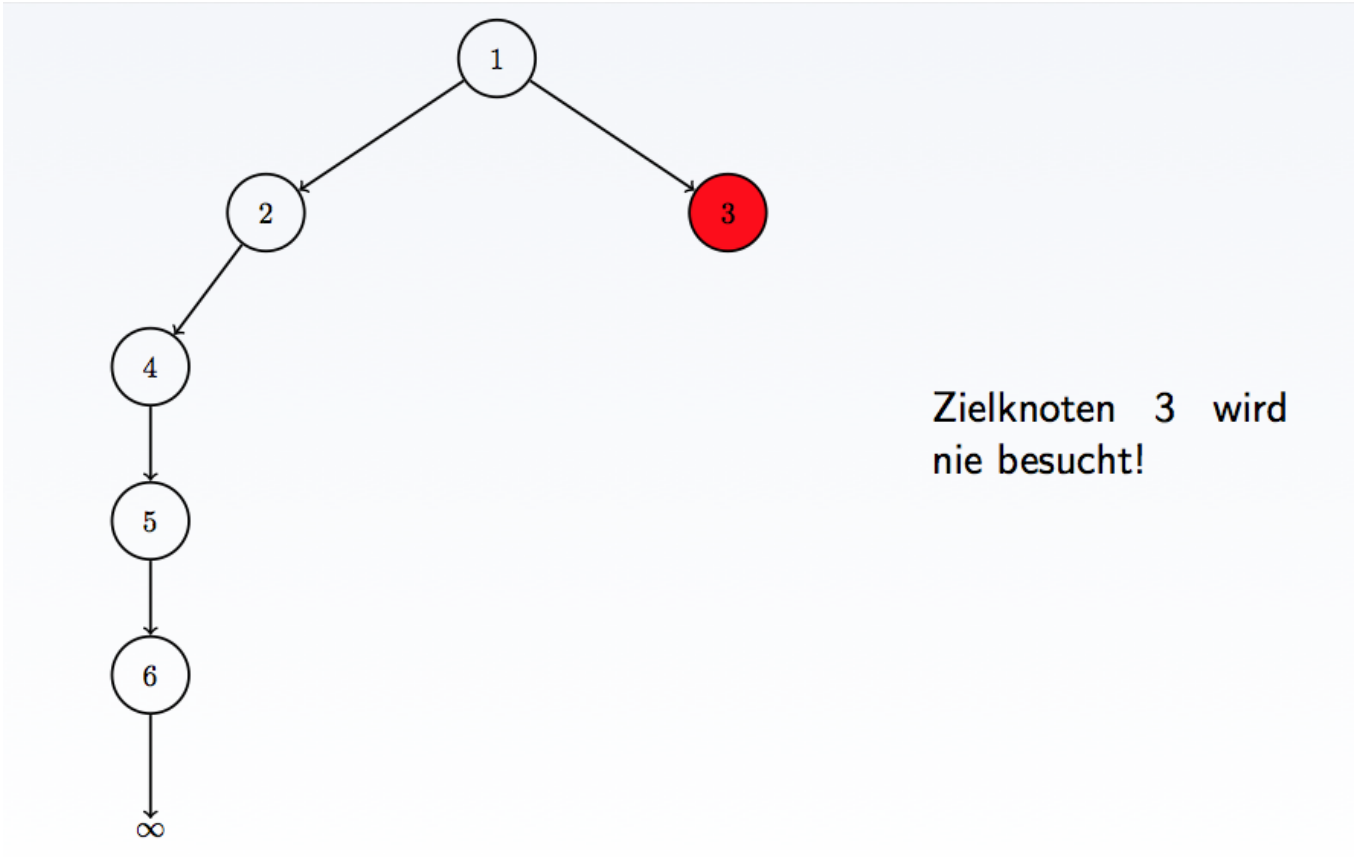
Komplexität (worst-case) bei fester Verzweigungrate $c > 1$:

- Platz: linear in der Tiefe
- Zeit: exponentiell in der Tiefe des Zielknotens

Vollständigkeit:

- Nicht vollständig, wenn der Suchgraph unendlich groß ist

Unvollständigkeit der Tiefensuche



Varianten der Tiefensuche

Tiefensuche mit Tiefenbeschränkung k

- Wenn Tiefe k überschritten, setze $NF(K) = \emptyset$
- Findet Zielknoten, die maximal in Tiefe k liegen

Varianten der Tiefensuche

Tiefensuche mit Sharing

- Merke bereits besuchte Knoten, um Knoten nicht doppelt zu besuchen (bei zyklischen Suchgraphen!)
- Speichern der besuchten Knoten: Hashtabelle
- Platz: Anzahl der besuchten Knoten (wegen Speicher f”ur schon untersuchte Knoten)
- Zeit: $n \times \log(n)$ mit n = Anzahl der untersuchten Knoten.
- Pragmatische Verbesserung: Nur maximal l viele Knoten speichern (damit der Platz beschränkt ist)

Varianten beim Backtracking

Vorgestelltes Verfahren

- Chronologisches Backtracking

Varianten

- Dynamic Backtracking
- Dependency-directed Backtracking

Abkürzungen, wenn sichergestellt ist, dass kein Zielknoten übersehen wird.

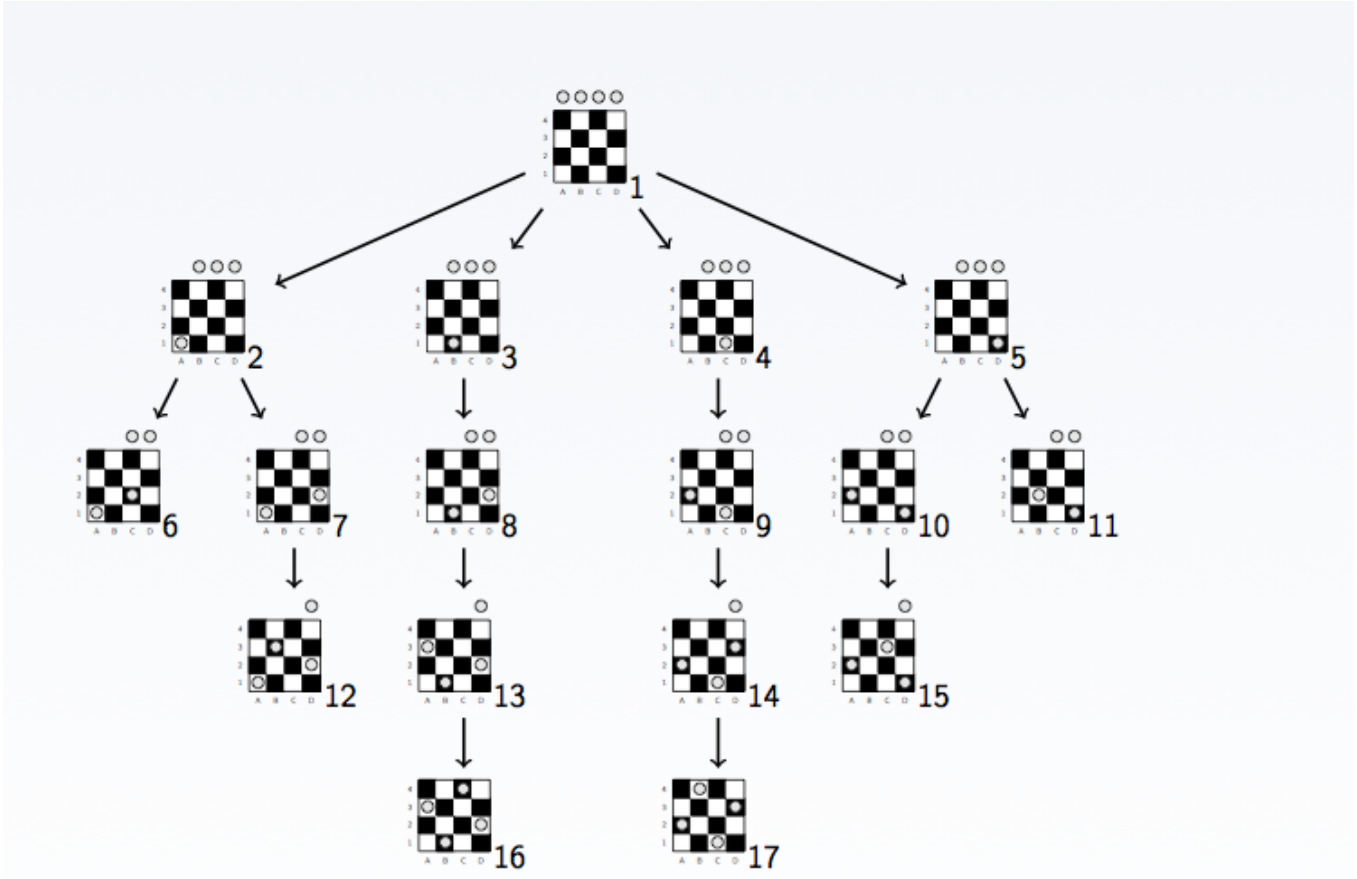
Breitensuche

Algorithmus Breitensuche

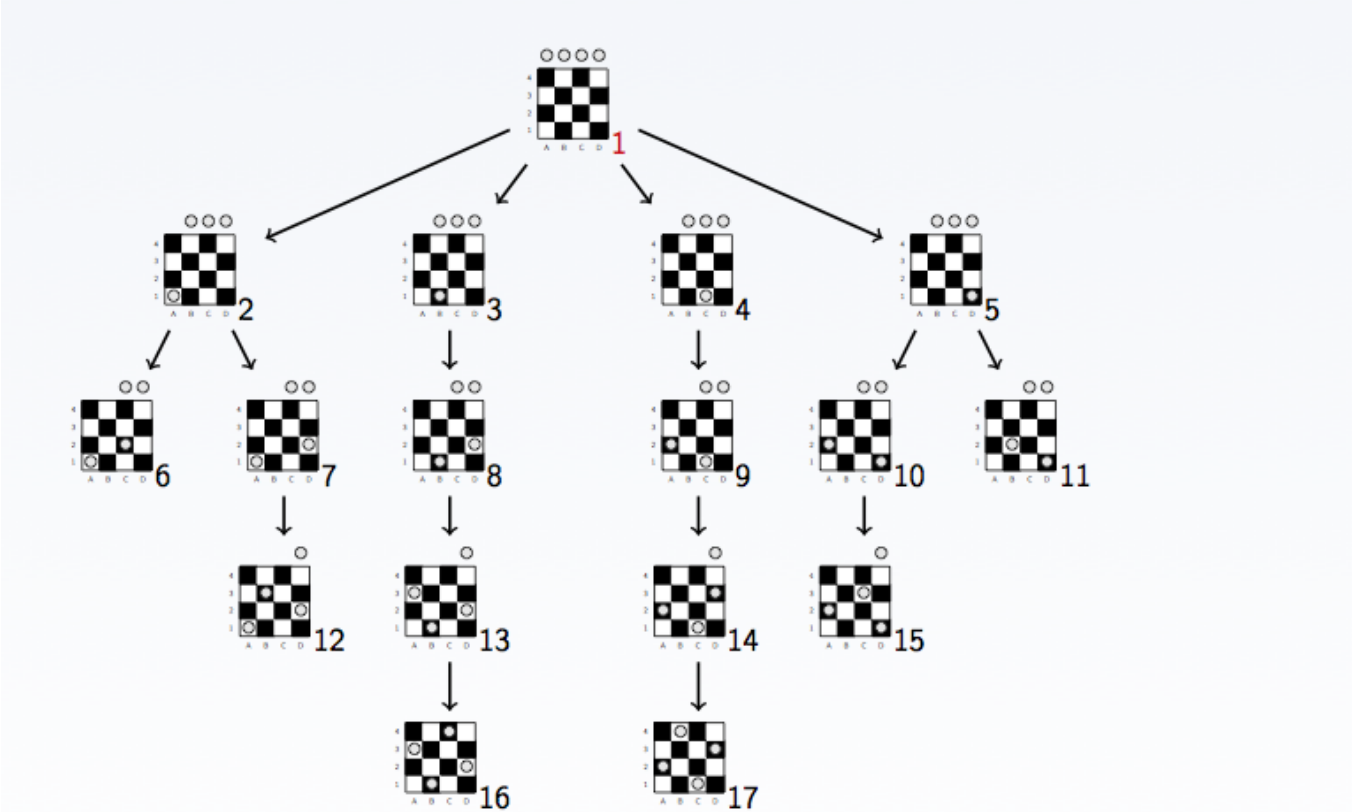
Datenstrukturen: L = Menge von Knoten markiert mit Weg
Eingabe: Füge die initialen Knoten in die Menge L ein.
Algorithmus:

1. Wenn L leer ist, dann breche ab.
2. Wenn L einen Zielknoten K enthält, dann gebe aus: K und Weg dorthin.
3. Sonst, sei $N(L)$ Menge aller direkten Nachfolger der Knoten von L, mit einem Weg dorthin markiert. Mache weiter mit Schritt 1 und $L := N(L)$.

Beispiel Breitensuche



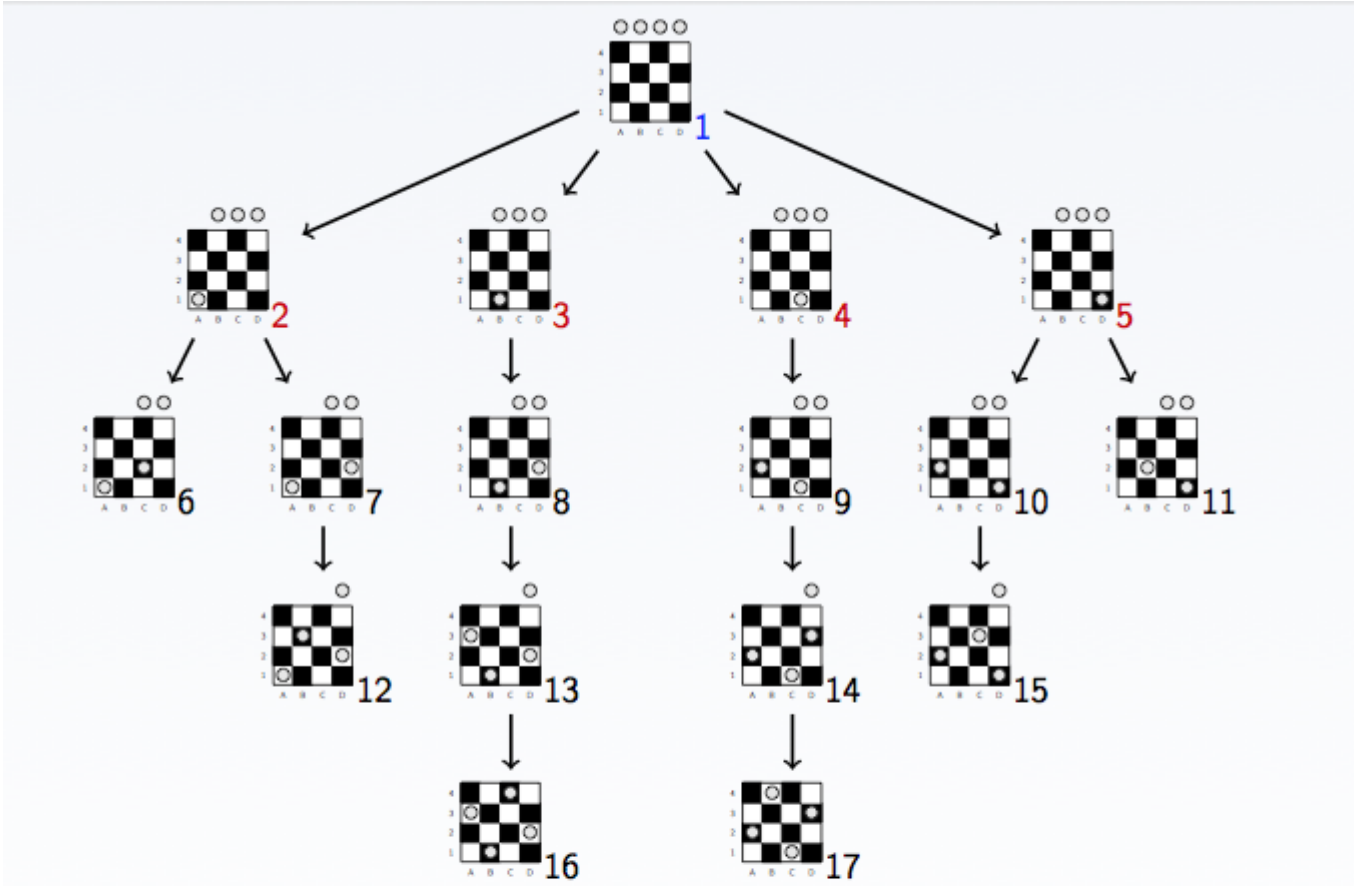
Beispiel Breitensuche



Am Anfang:

$L := \{(1, [])\}$

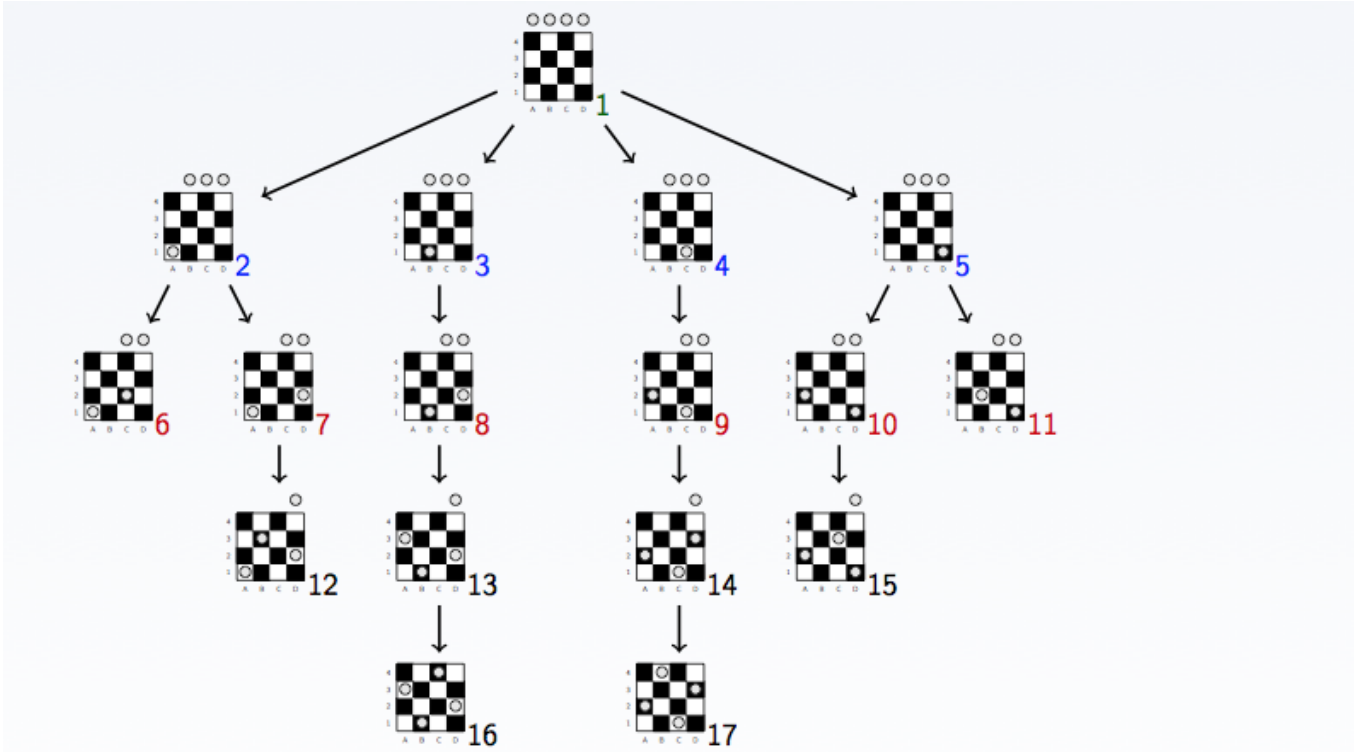
Beispiel Breitensuche



1. Iteration

$L := \{(1, [])\}$
$NF(L) = \{2, 3, 4, 5\}$
$L := \{(2, [1]), (3, [1]), (4, [1]), (5, [1])\}$

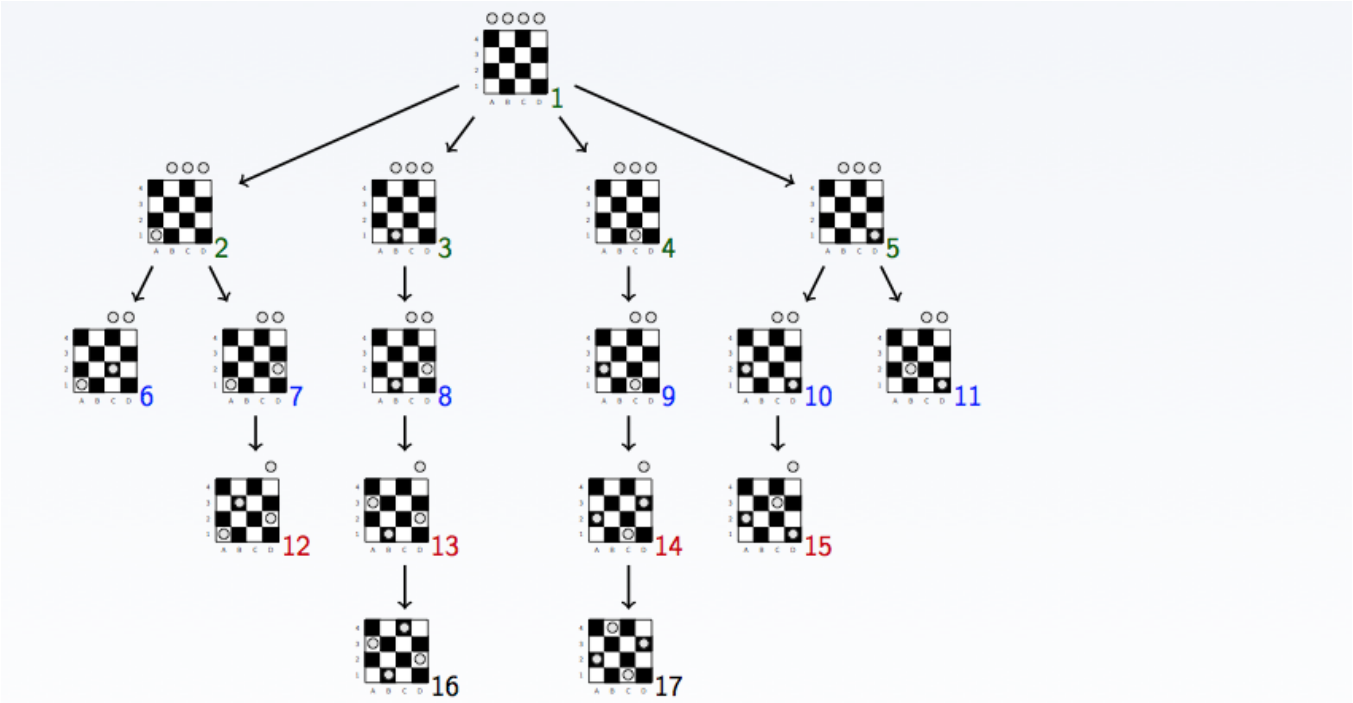
Beispiel Breitensuche



2. Iteration

$L := \{(2, [1]), (3, [1]), (4, [1]), (5, [1])\}$
$NF(L) = NF(2) \cup NF(2) \cup NF(4) \cup NF(5) = \{6, 7, 8, 9, 10, 11\}$
$L := \{(6, [1, 2]), (7, [1, 2]), (8, [1, 3]), (9, [1, 4]), (10, [1, 5]), (11, [1, 5])\}$

Beispiel Breitensuche



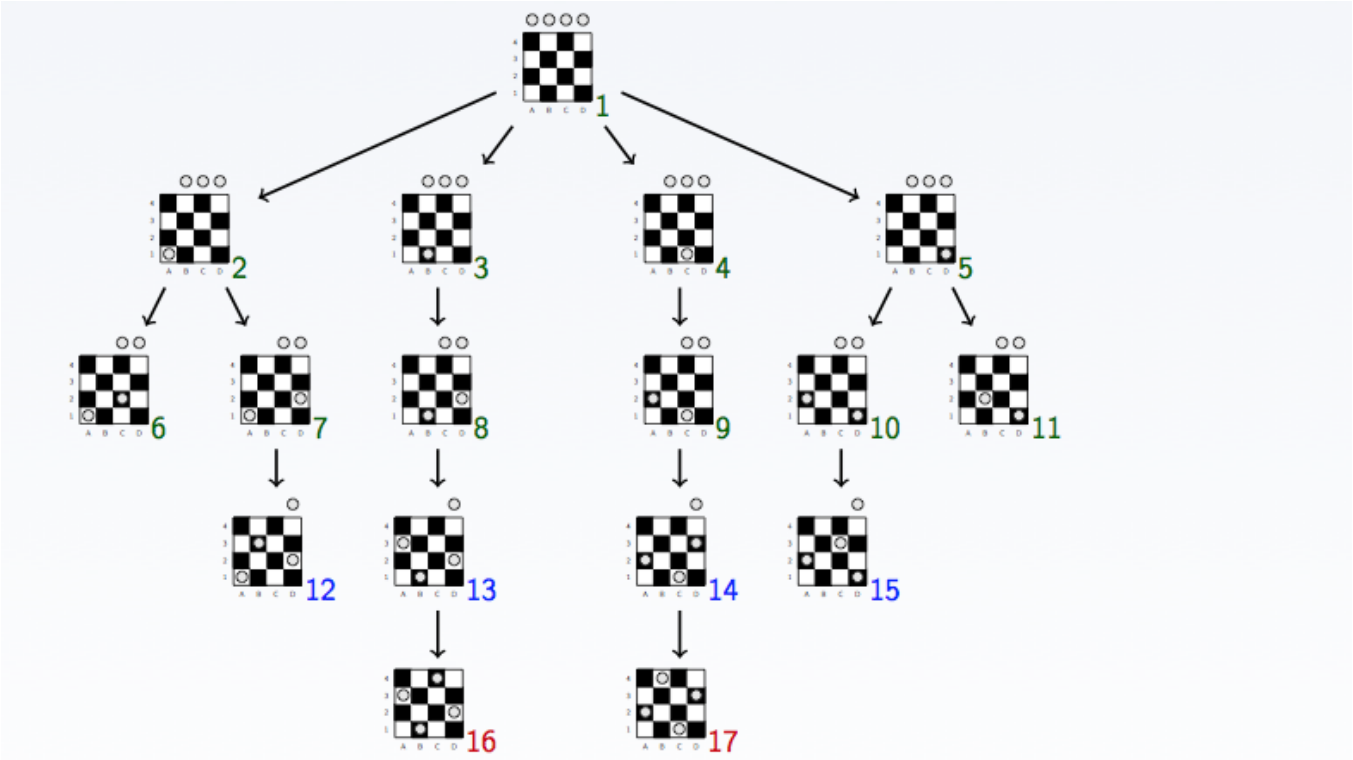
3. Iteration

$L := \{(6, [1, 2]), (7, [1, 2]), (8, [1, 3]), (9, [1, 4]), (10, [1, 5]), (11, [1, 5])\}$

$NF(L) = NF(6) \cup NF(7) \cup NF(8) \cup NF(8) \cup NF(9) \cup NF(10) \cup NF(11) = \{12, 13, 14, 15\}$

$L := \{(12, [1, 2, 7]), (13, [1, 3, 8]), (14, [1, 4, 9]), (15, [1, 5, 10])\}$

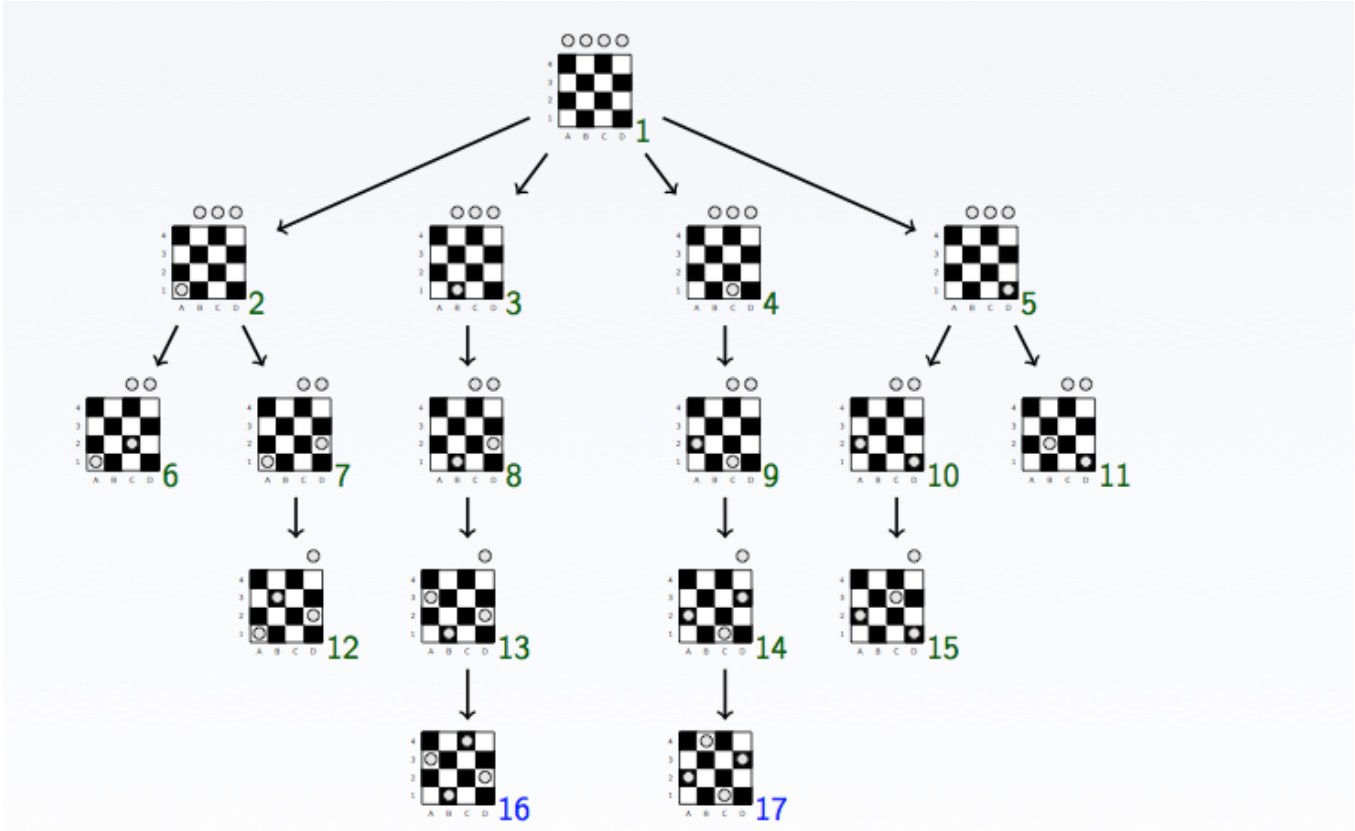
Beispiel Breitensuche



4. Iteration

$L := \{(12, [1, 2, 7]), (13, [1, 3, 8]), (14, [1, 4, 9]), (15, [1, 5, 10])\}$
$NF(L) = NF(12) \cup NF(13) \cup NF(14) \cup NF(15) = \{16, 17\}$
$L := \{(16, [1, 3, 8, 13]), (17, [1, 4, 9, 14])\}$

Beispiel Breitensuche



5. Iteration

$L := \{(16, [1, 3, 8, 13]), (17, [1, 4, 9, 14])\}$

L enhält Zielknoten $\Rightarrow$ gebe 1,3,8,13,16 aus

Eigenschaften

Komplexität (worst-case) bei fester Verzweigungrate $c > 1$

- **Platz**: Anzahl der Knoten in Tiefe d, d.h. $O(c^d) =$ exponentiell in der Tiefe d!
- **Zeit**: $\underbrace{\text{Anzahl der Knoten in Tiefe } d}_{n} + \underbrace{(n \log n)}_{\text{Mengenbildung}} =$

  $O(c^d(1 + d * \log\ c))$

Vollständigkeit

- Die Breitensuche ist vollständig!
  (bei endlicher Verzweigungrate)

Fazit: Tiefen- und Breitensuche

	Tiefensuche	Breitensuche
Zeit	$O(c^d)$	$O(c^d(1 + d \log c))$
Platz	$O(d)$	$O(c^d)$
Vollständig	nein	ja

Iteratives Vertiefen

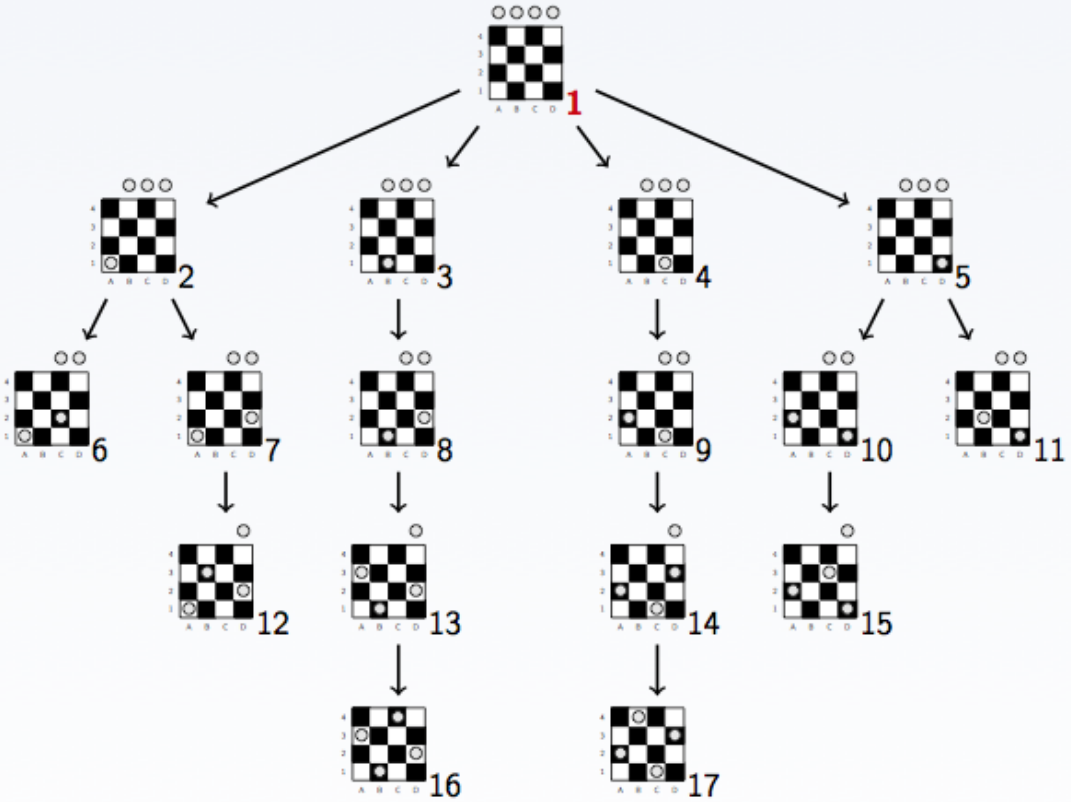
- iterative deepening
- Kompromiss a la *Vollständige Tiefensuche*

Pseudo-Code

:

1. $k := 0$;
2. Tiefensuche mit Tiefenschranke k;
3. **wenn** Ziel gefunden **dann**
   breche ab, und gebe Ziel mit Weg aus
   sonst
4. $k := k + 1$;
5. gehe zu 2

Beispiel: Iterative Tiefensuche

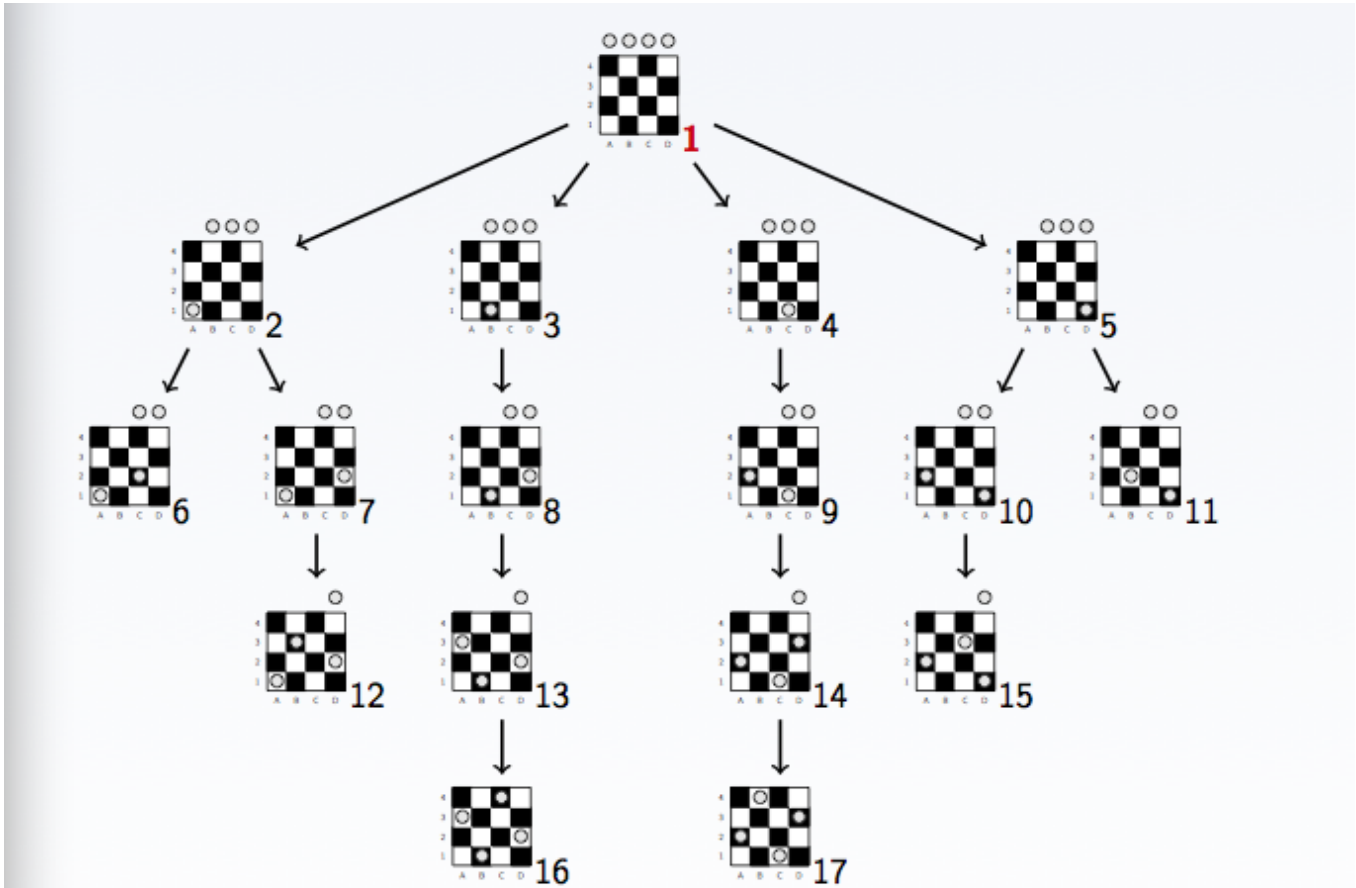


Reihenfolge der Knotenbesuche

$k = 1$

1

Beispiel: Iterative Tiefensuche

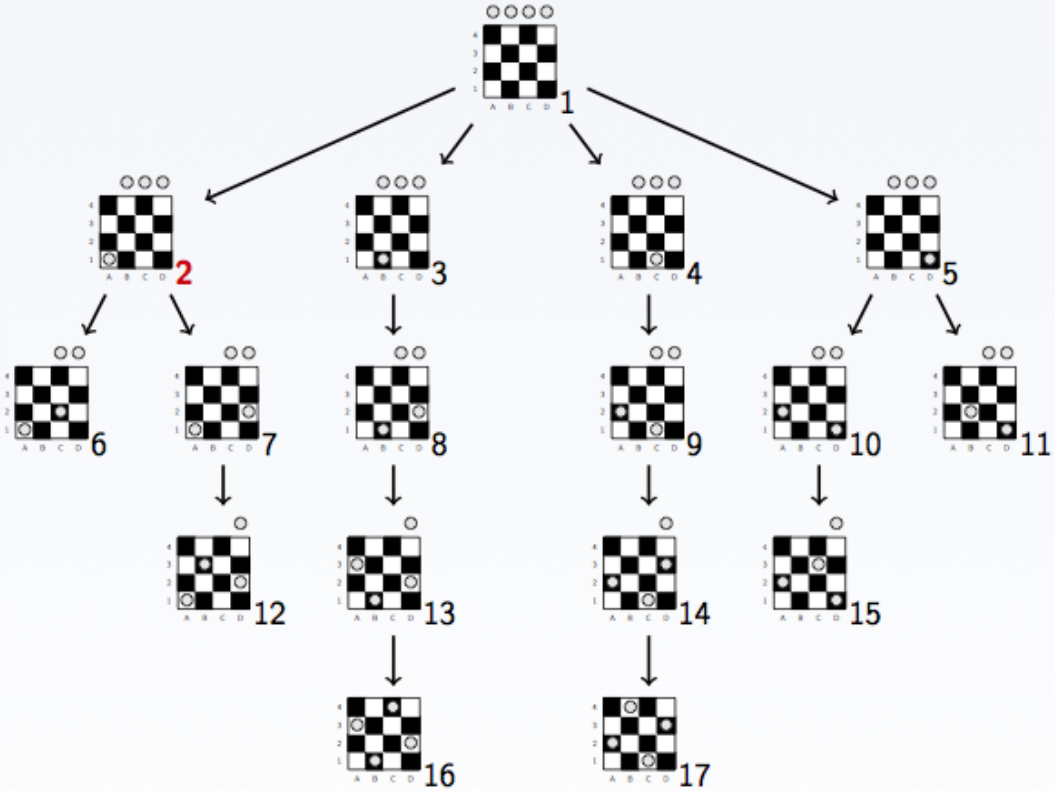


Reihenfolge der Knotenbesuche

$k = 2$

1,2,3,4,5

Beispiel: Iterative Tiefensuche

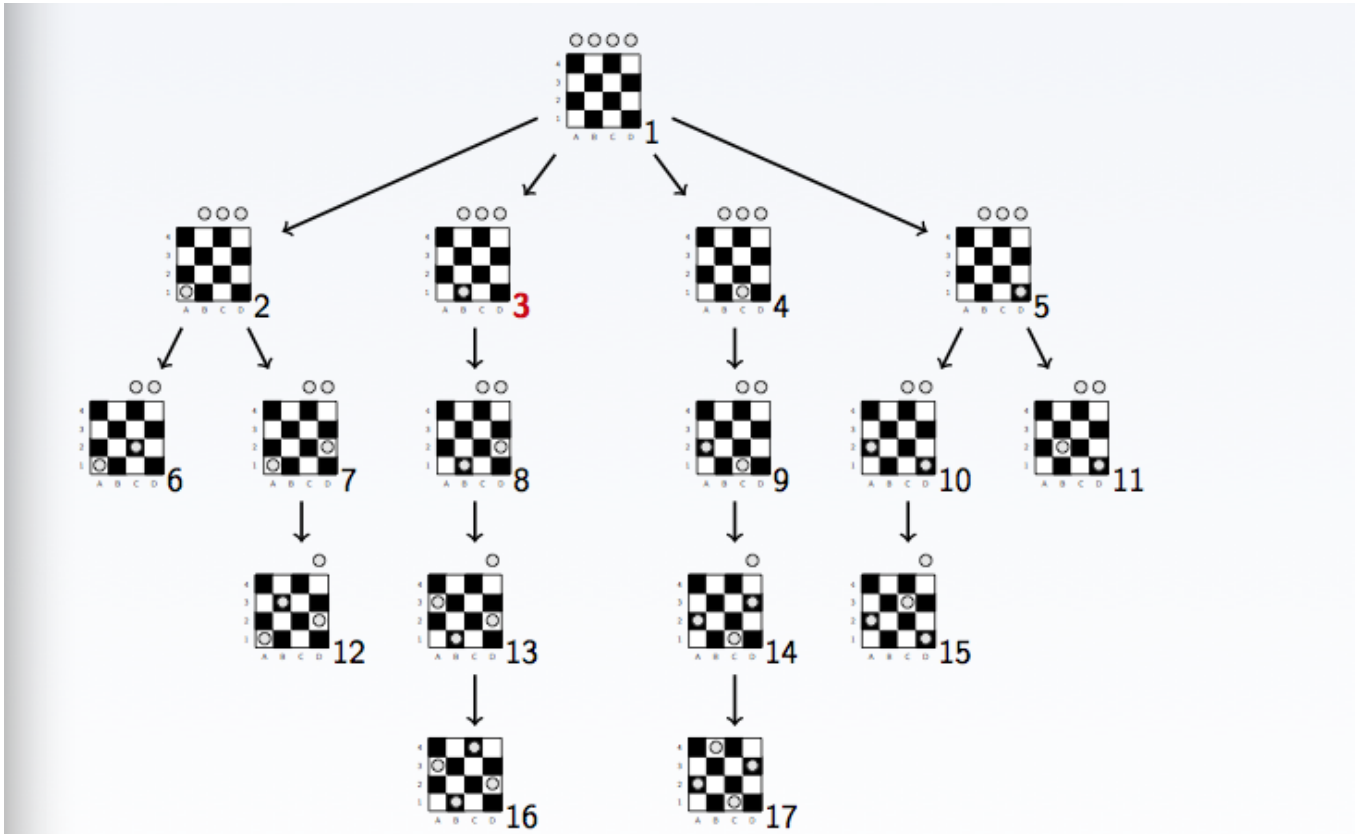


Reihenfolge der Knotenbesuche

$k = 2$

1,2,3,4,5

Beispiel: Iterative Tiefensuche

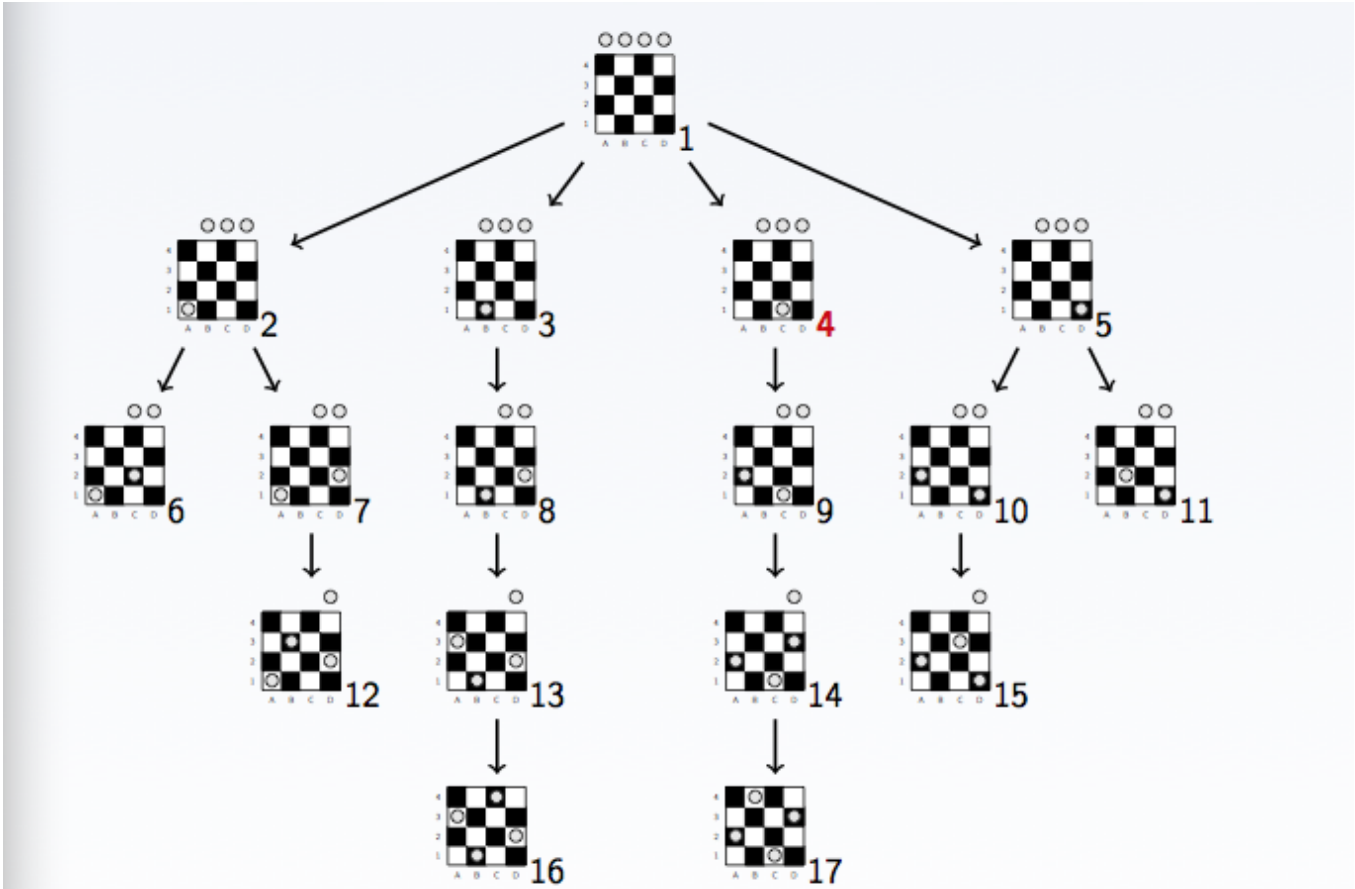


Reihenfolge der Knotenbesuche

$k = 2$

1,2,3,4,5

Beispiel: Iterative Tiefensuche

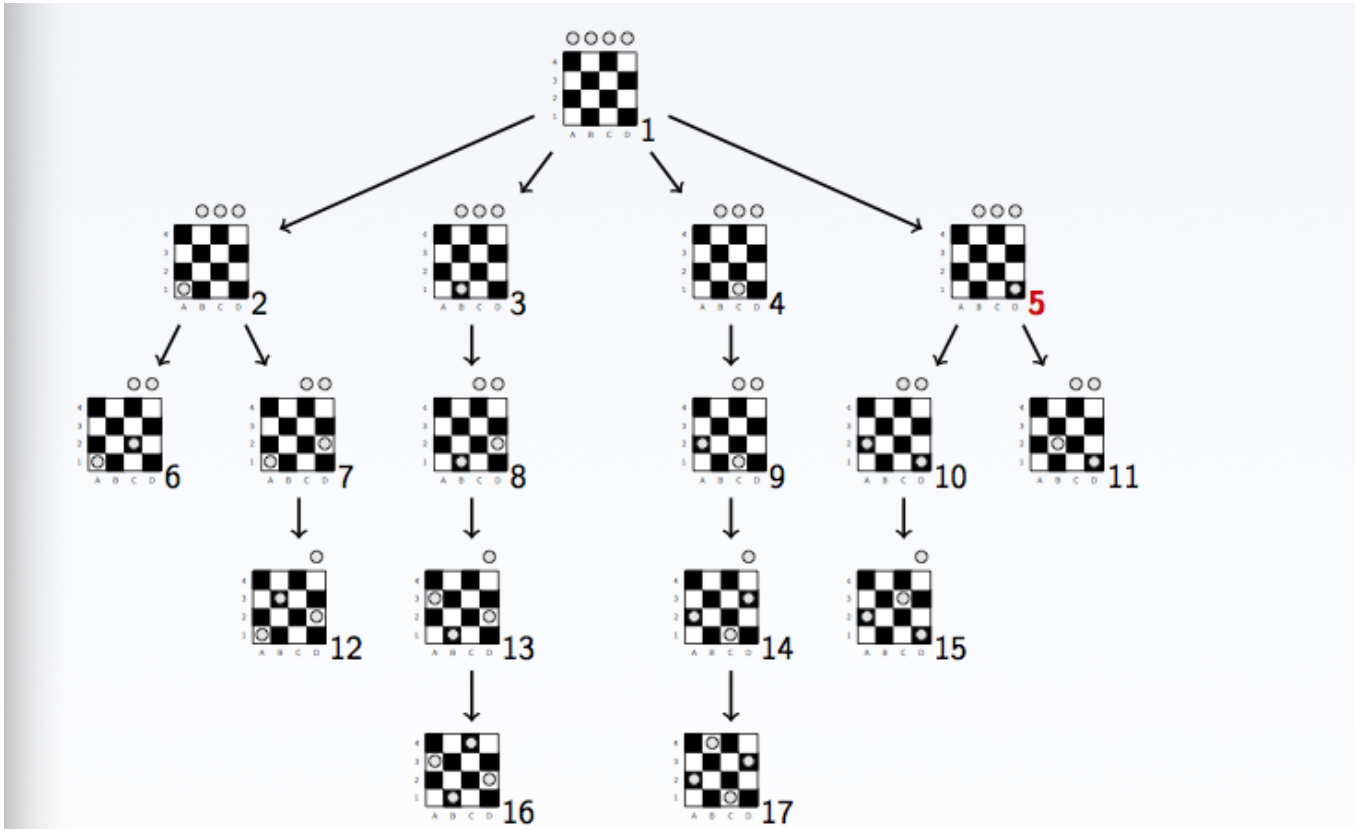


Reihenfolge der Knotenbesuche

$k = 2$

1,2,3,4,5

Beispiel: Iterative Tiefensuche

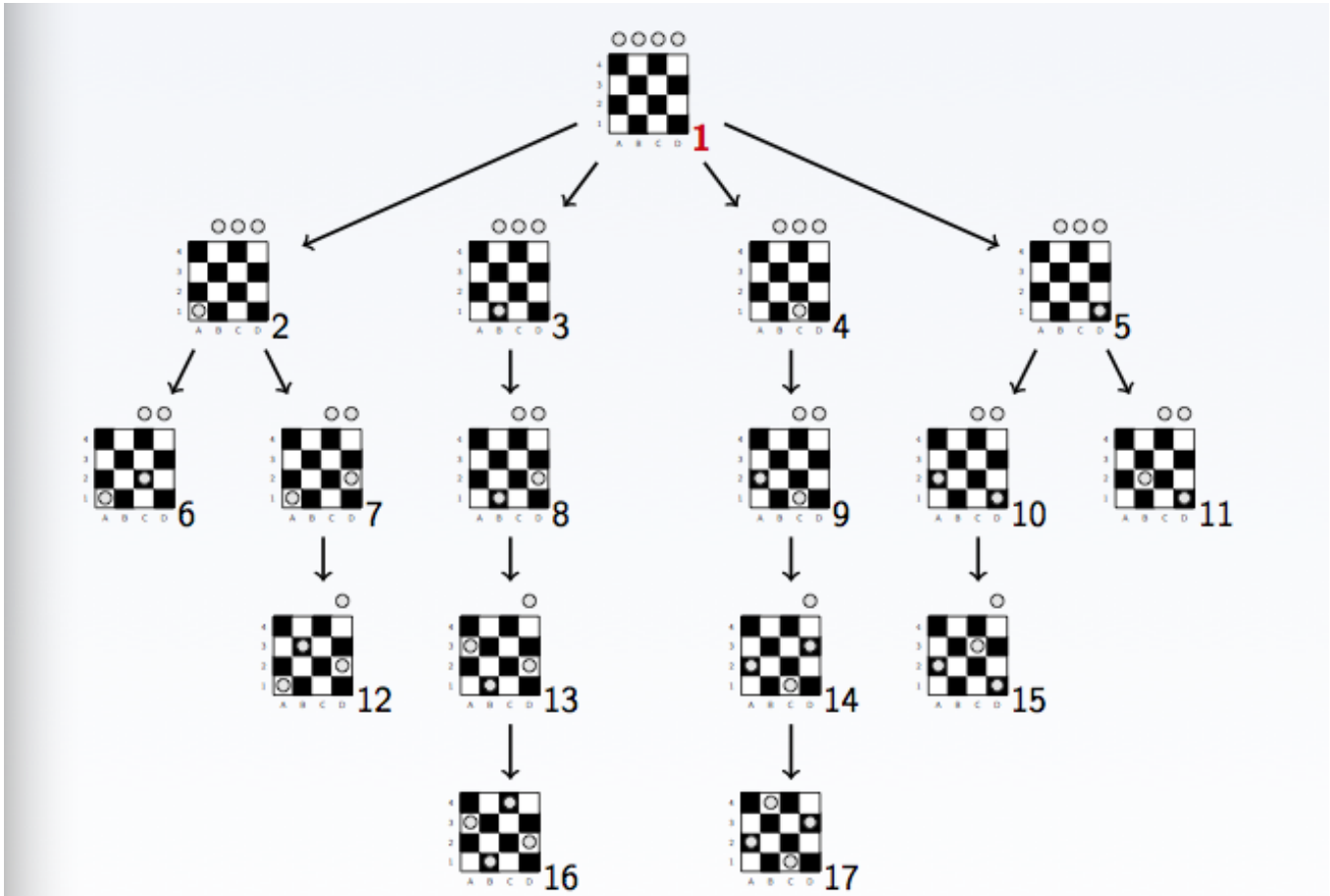


Reihenfolge der Knotenbesuche

$k = 2$

1,2,3,4,5

Beispiel: Iterative Tiefensuche

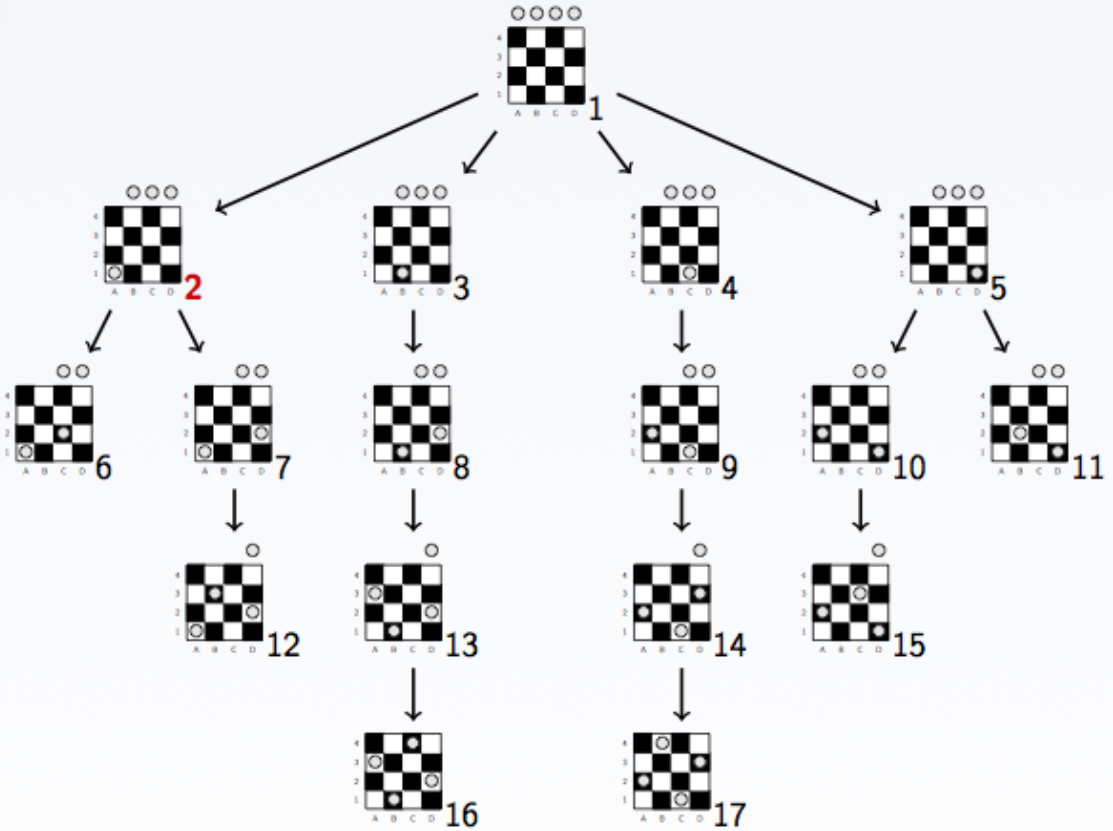


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

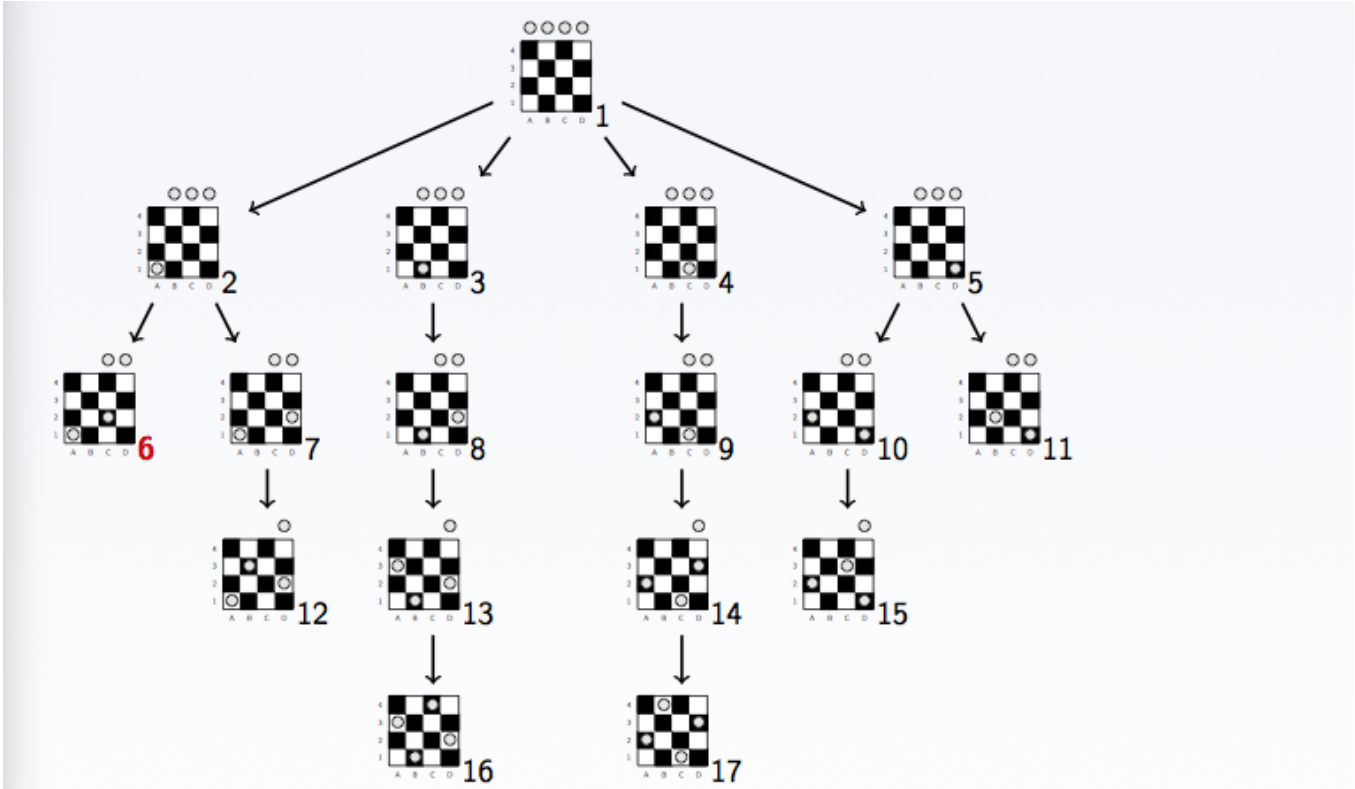


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

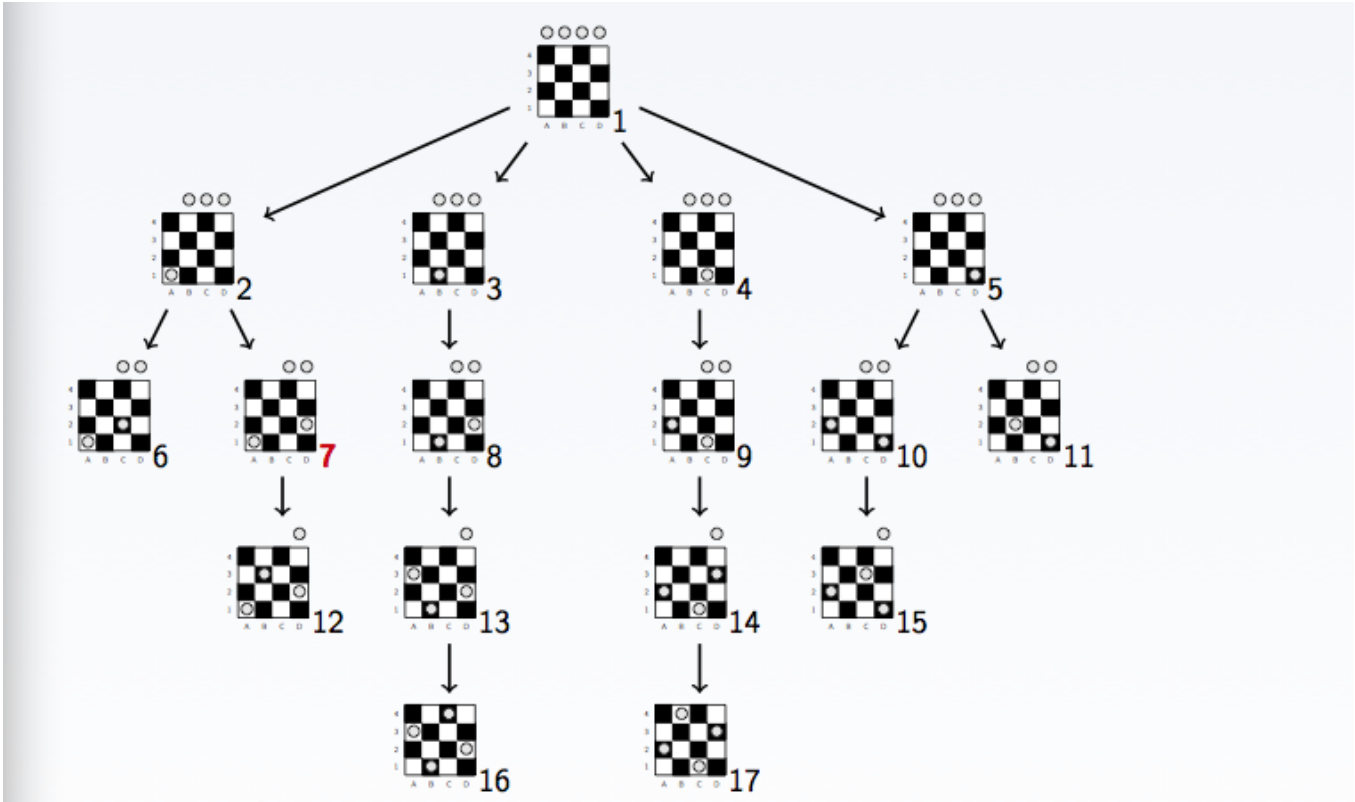


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

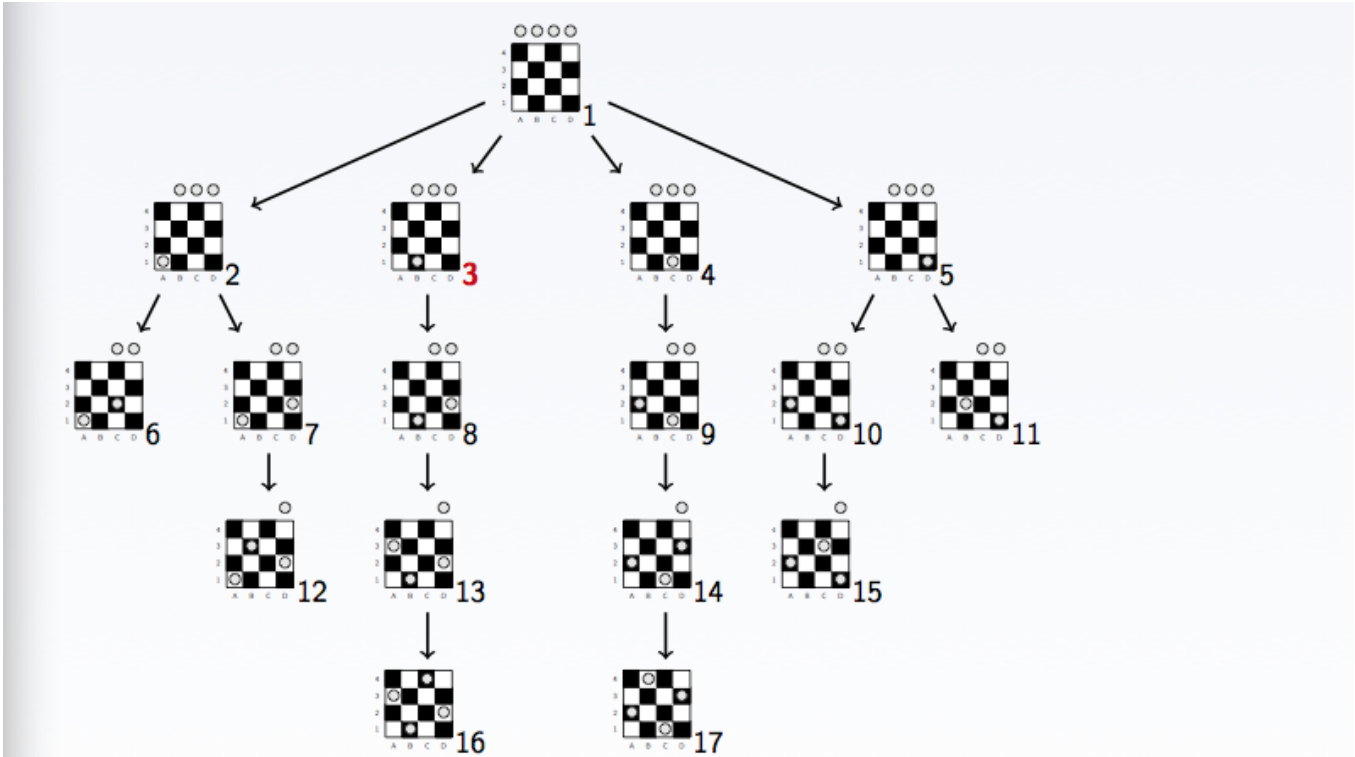


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

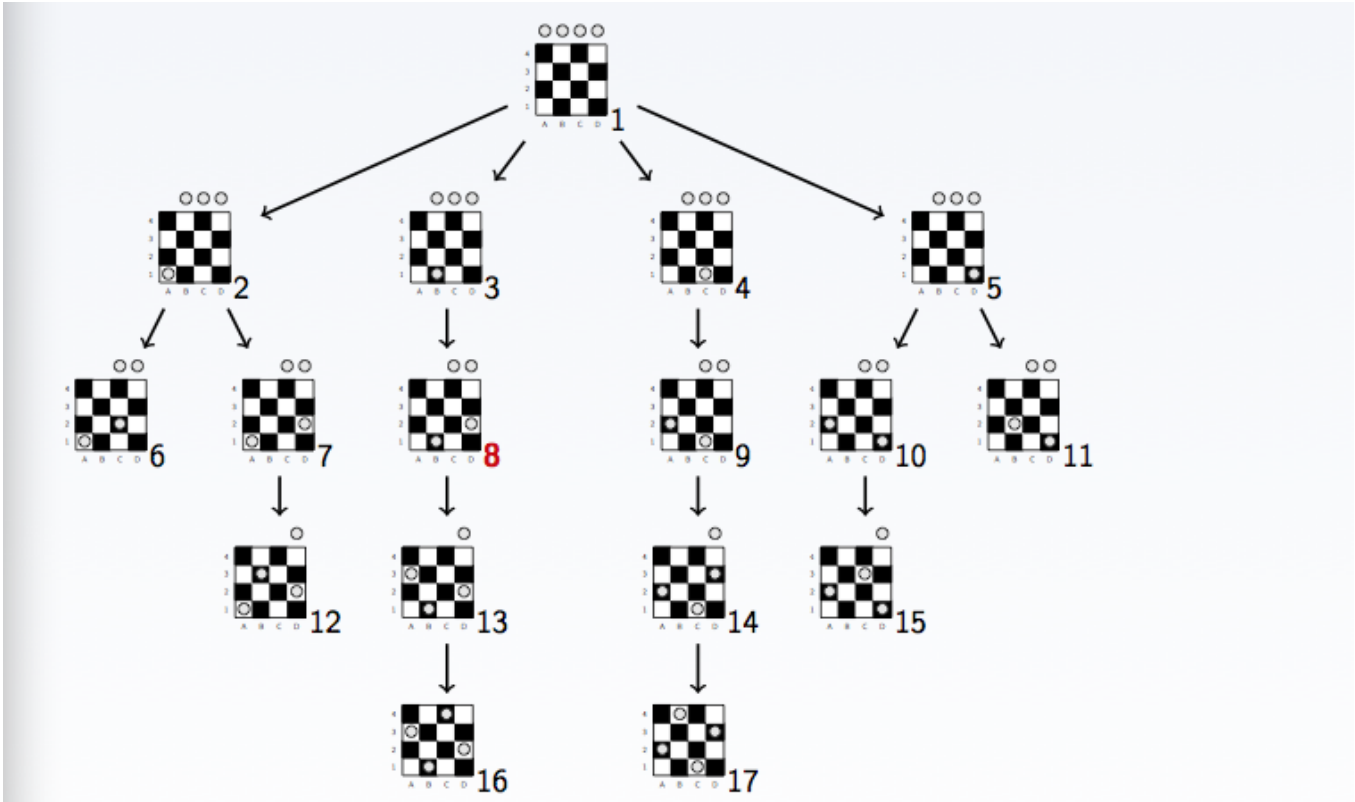


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

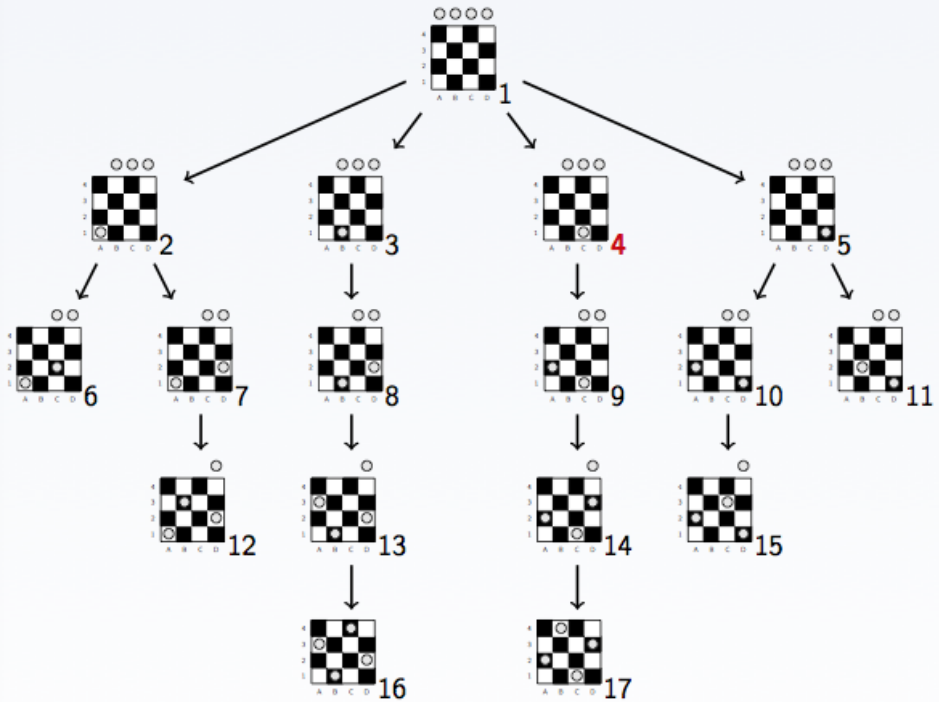


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

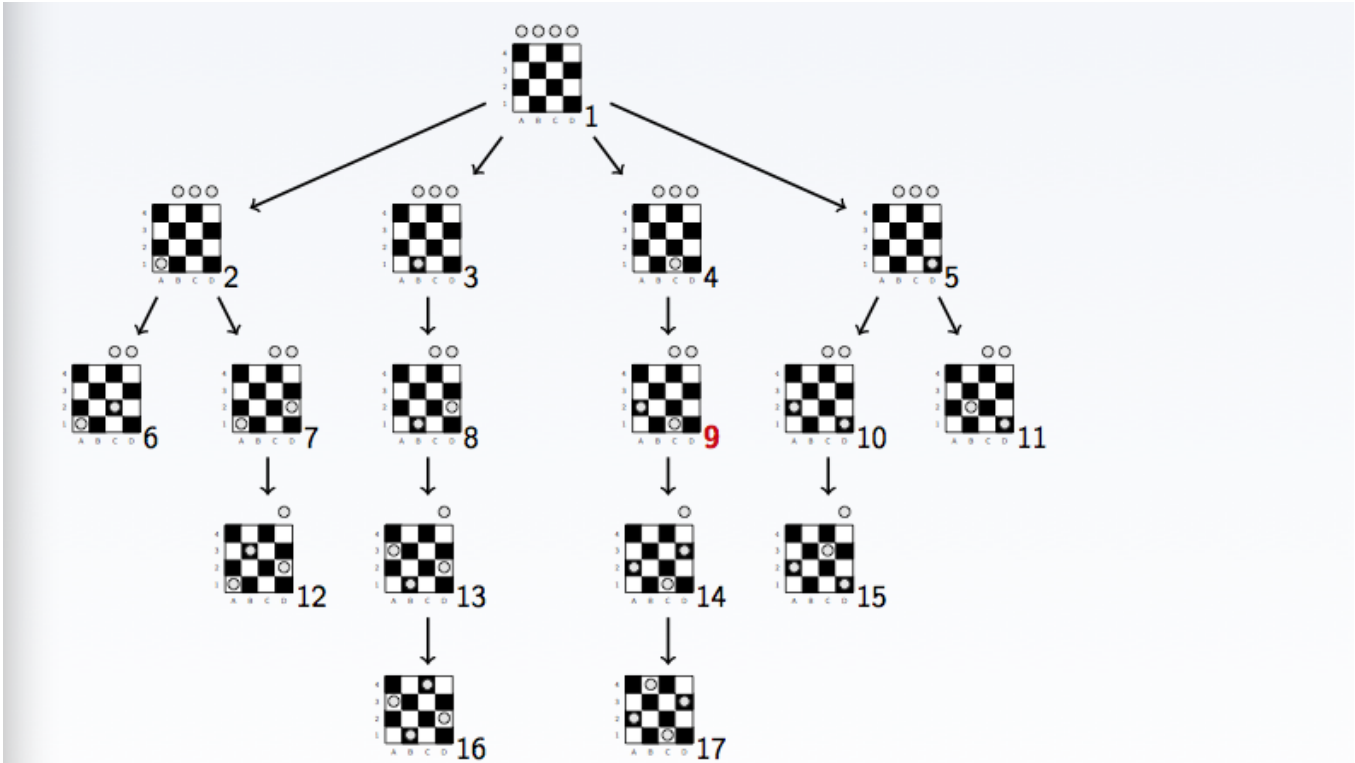


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

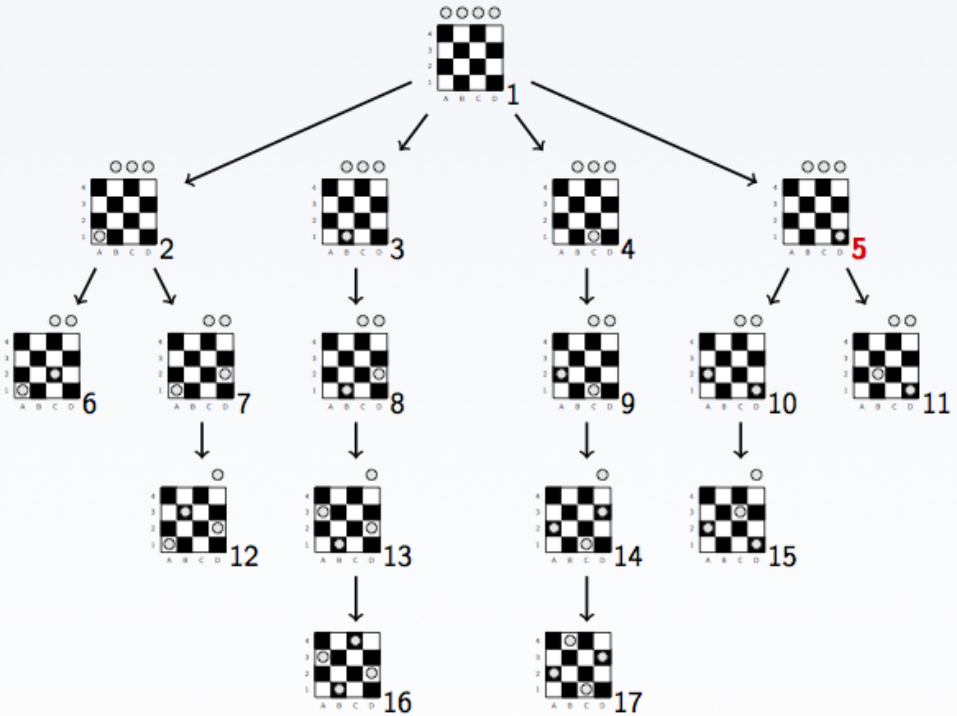


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

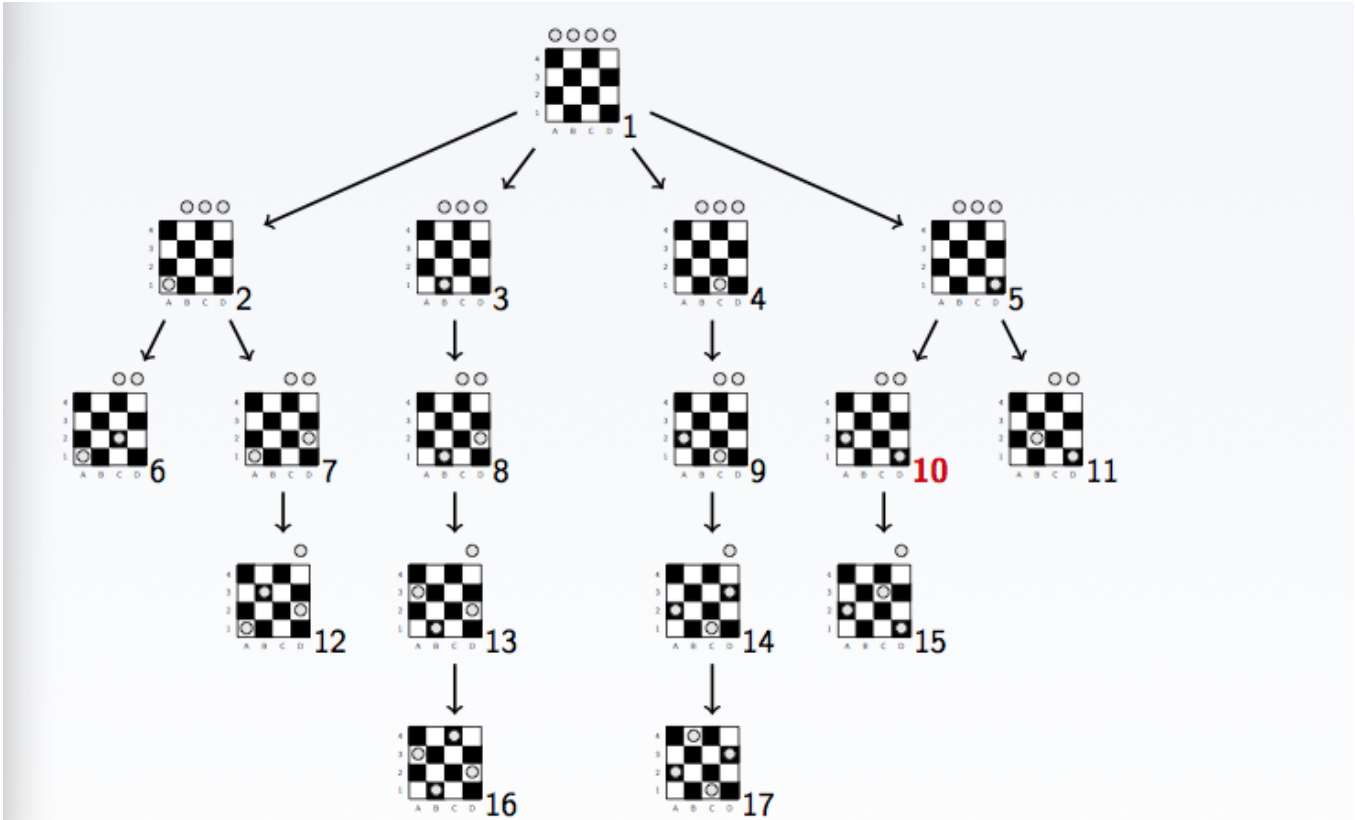


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

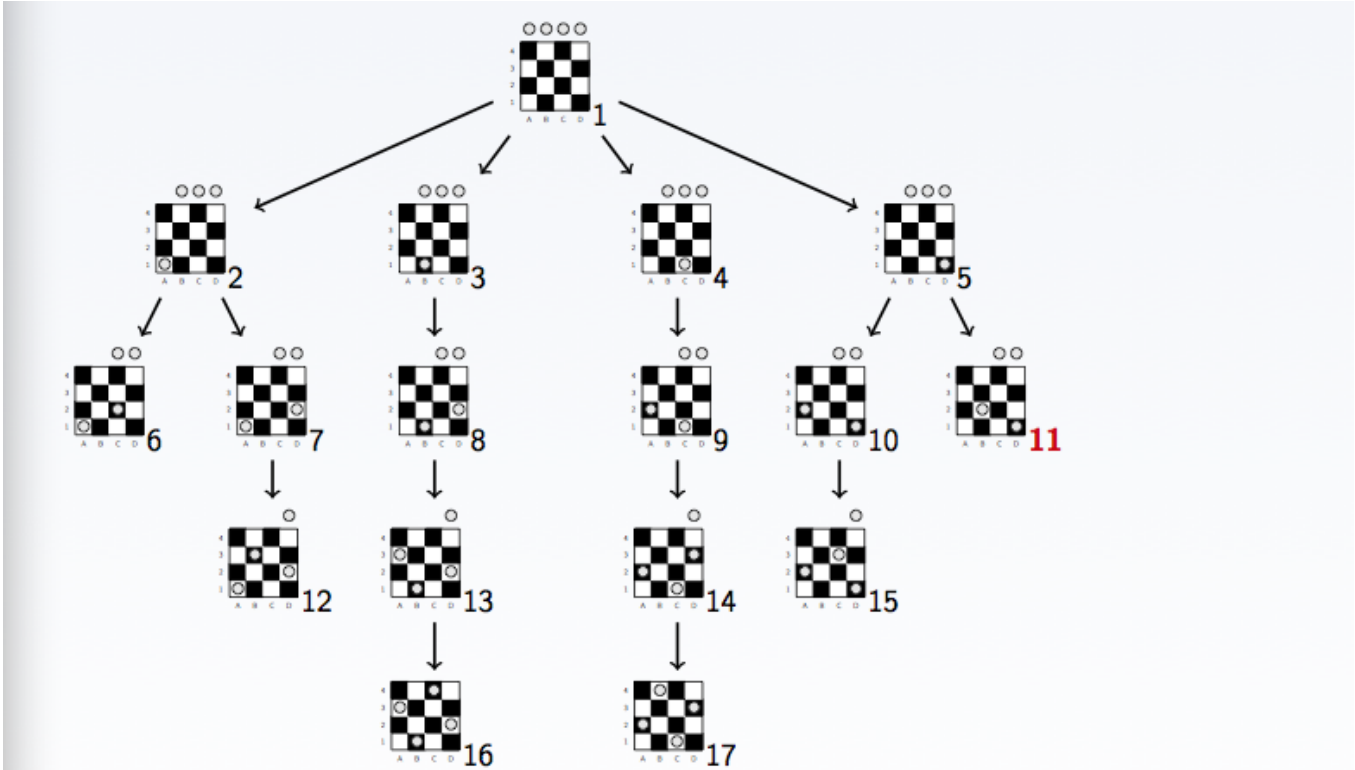


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

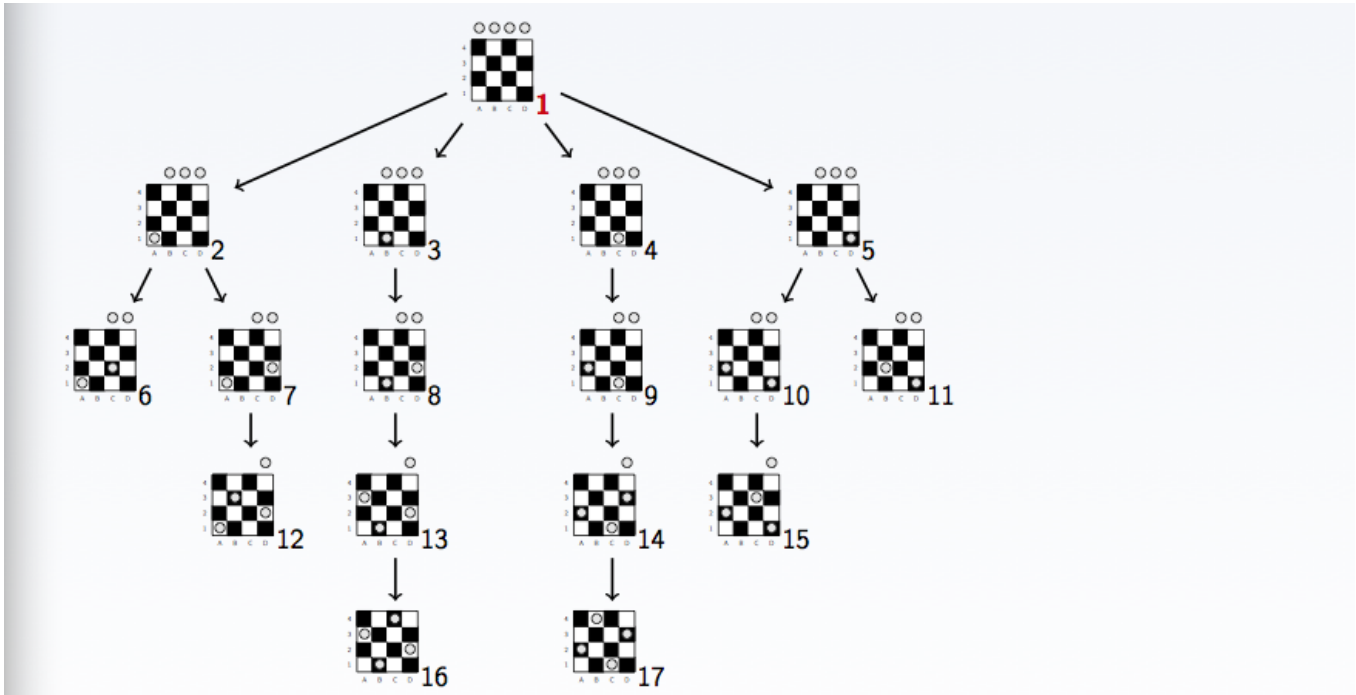


Reihenfolge der Knotenbesuche

$k = 3$

1,2,6,7,3,8,4,9,5,10,11

Beispiel: Iterative Tiefensuche

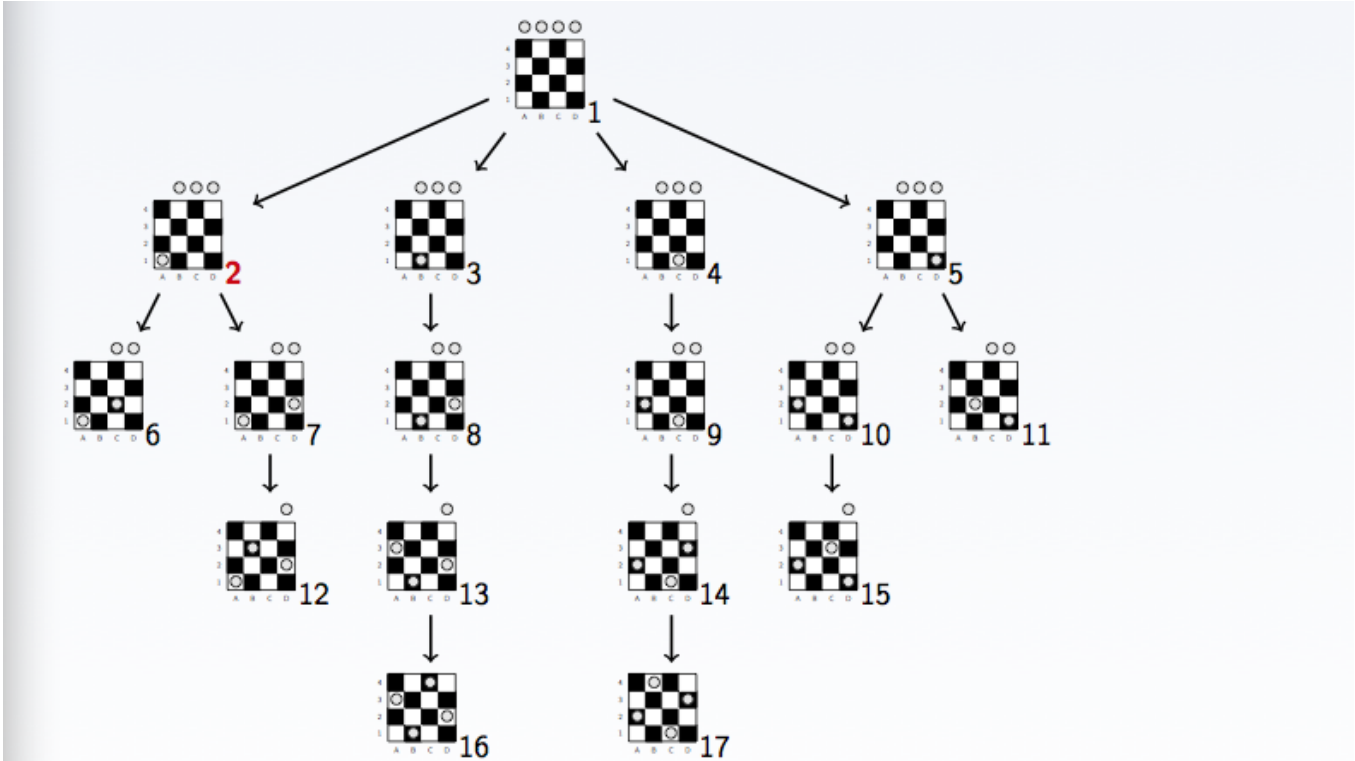


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

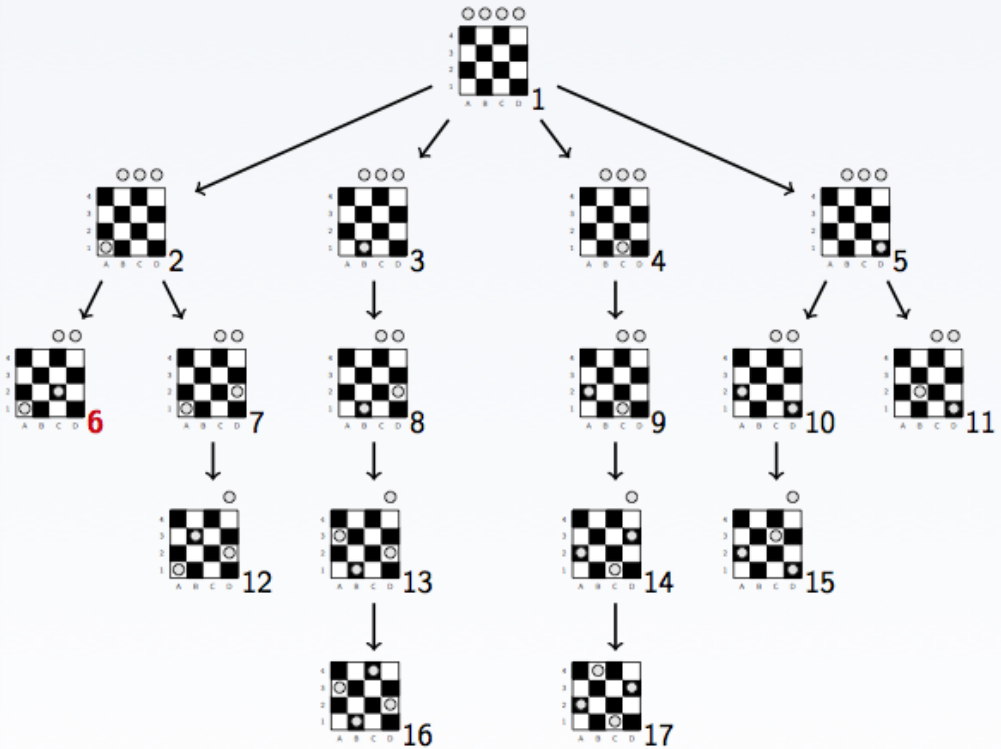


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

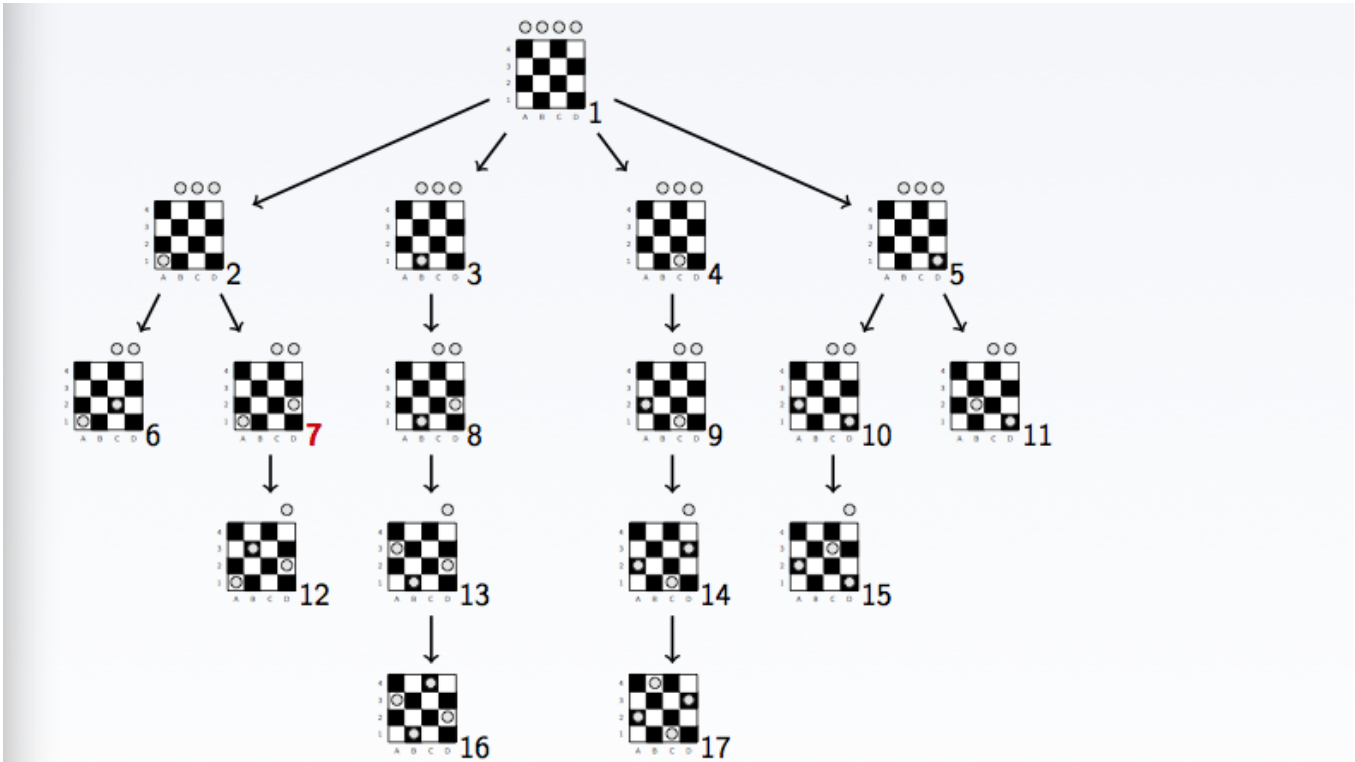


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

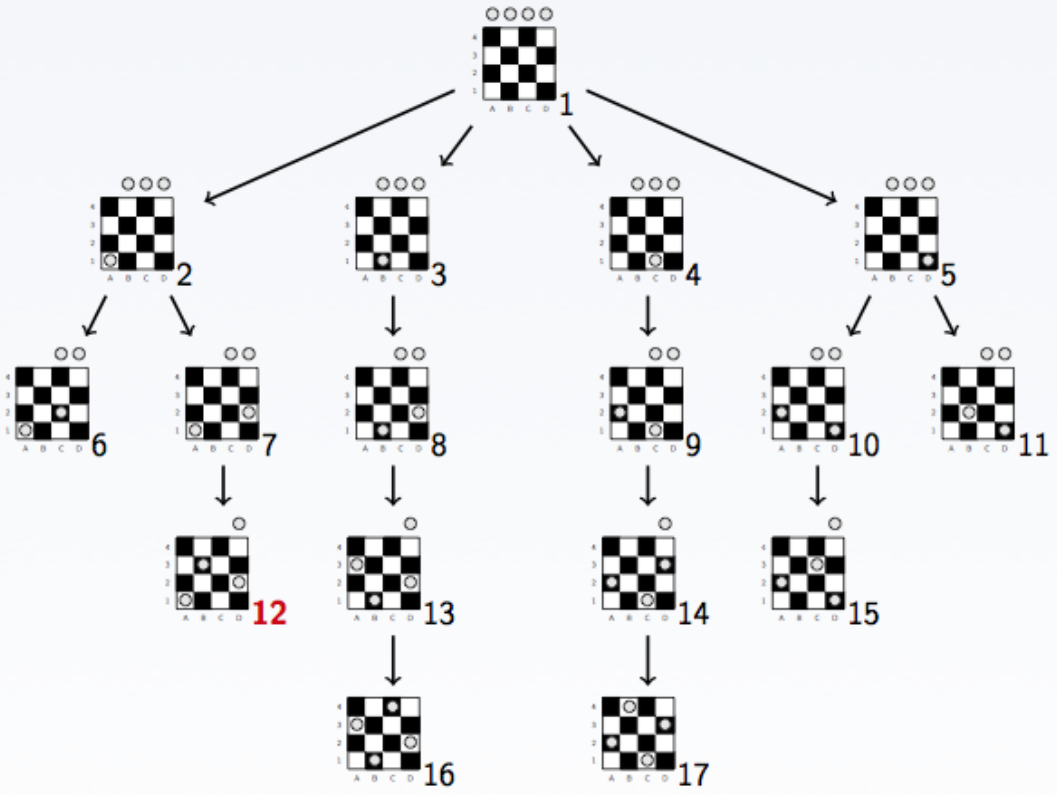


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

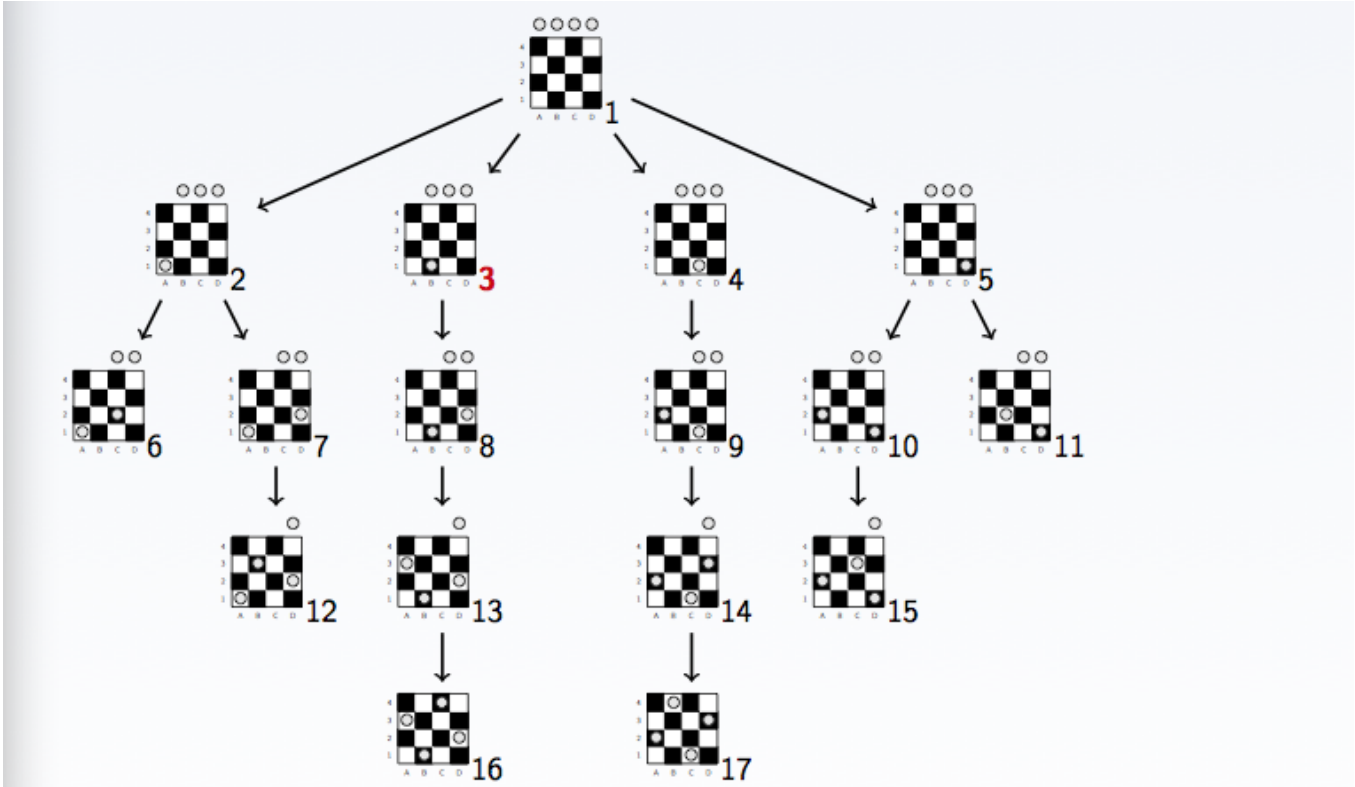


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

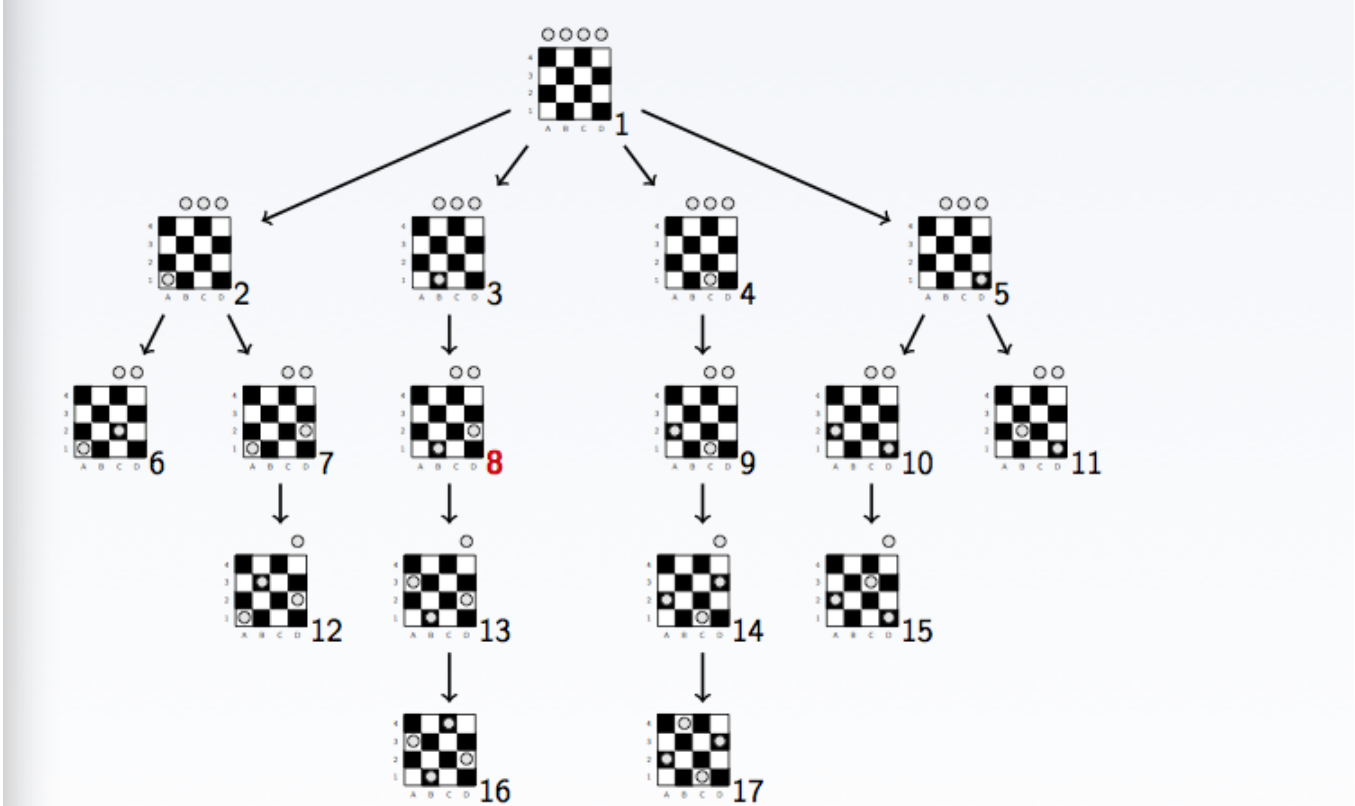


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

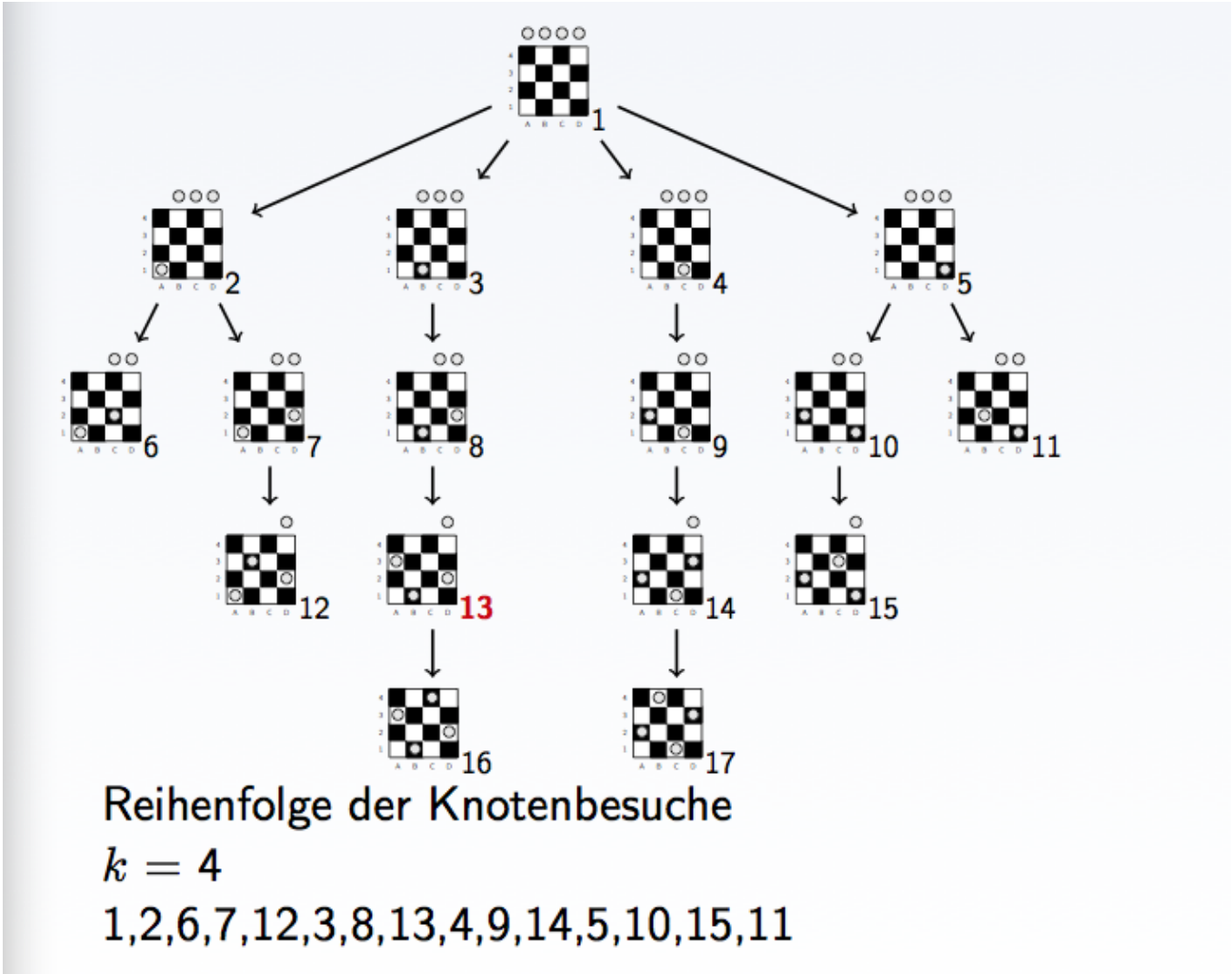


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

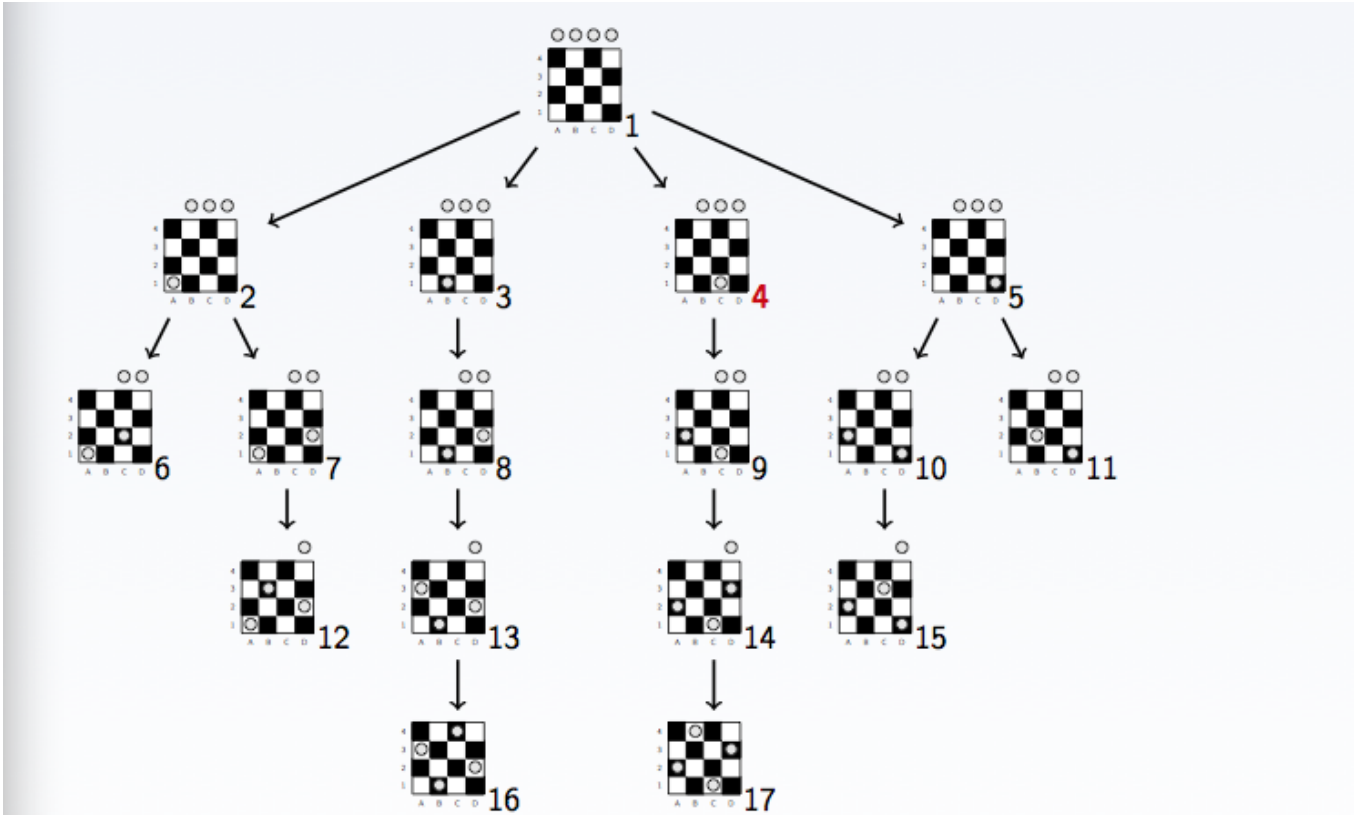


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

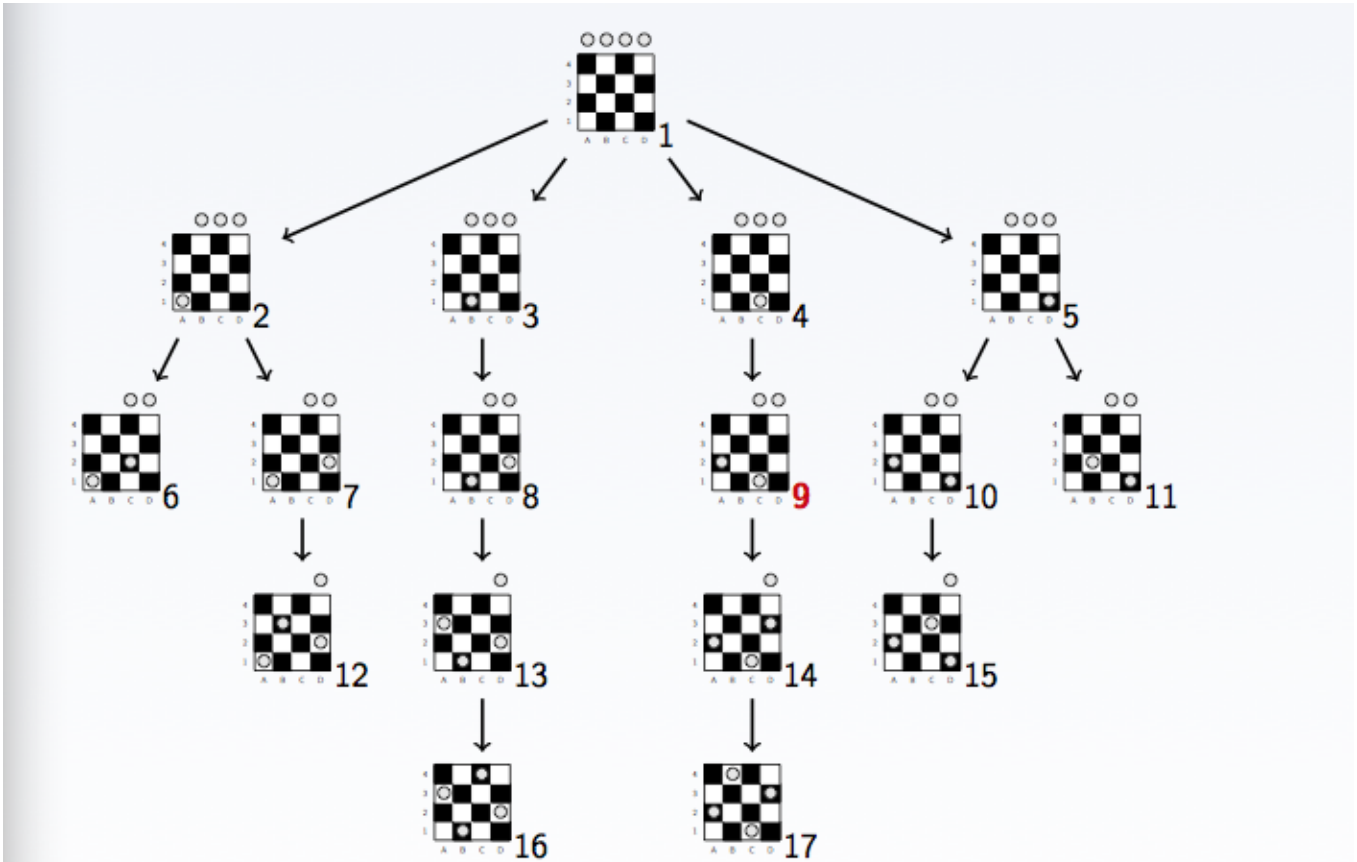


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

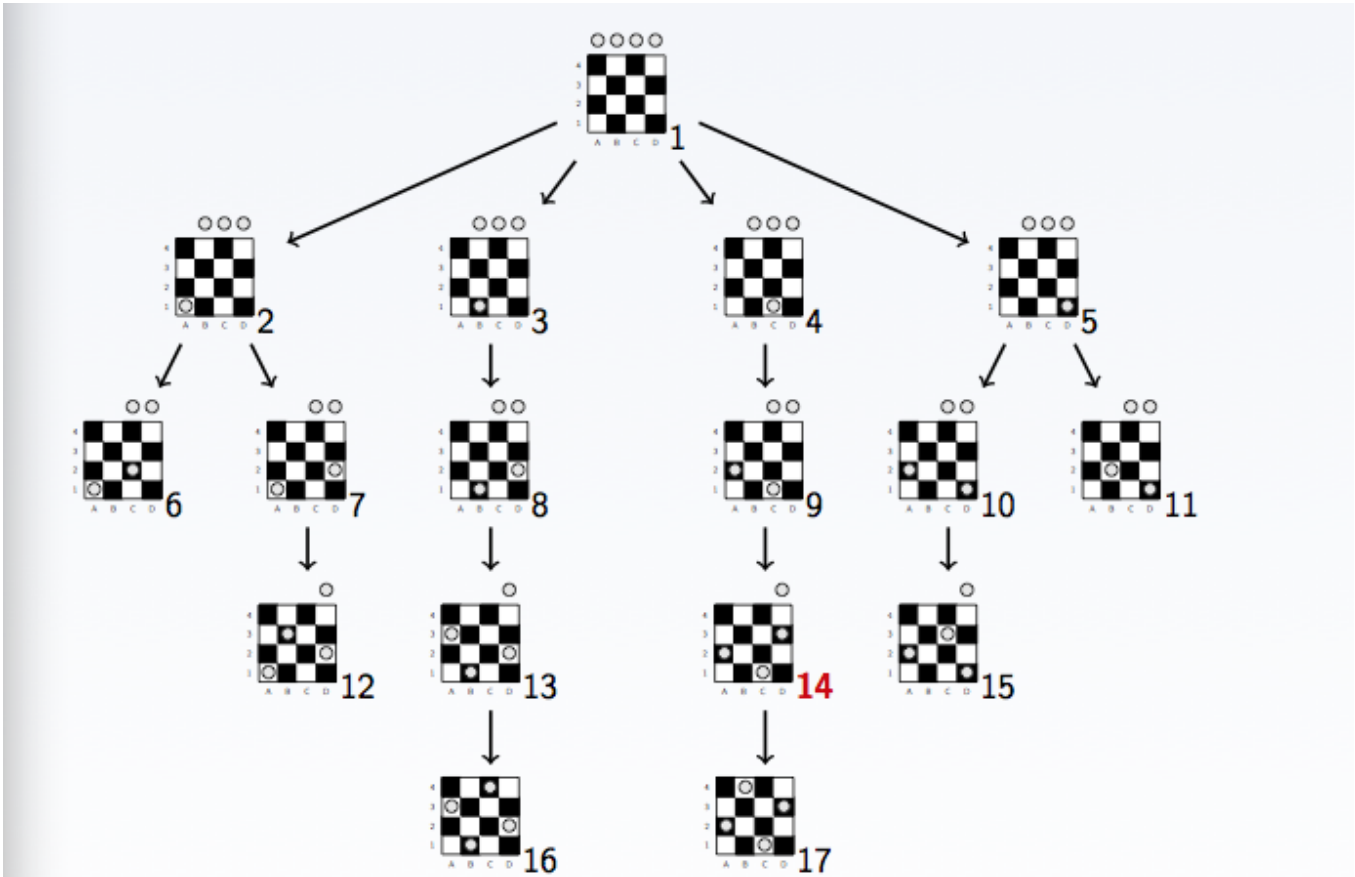


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

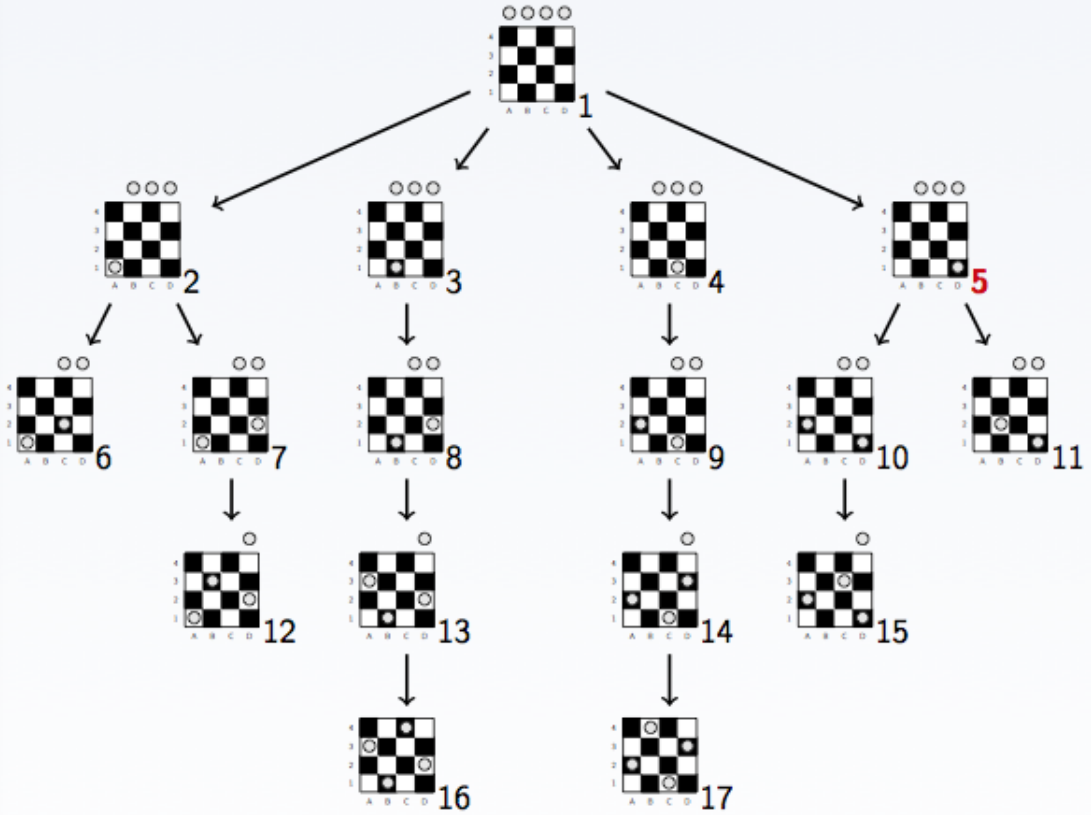


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

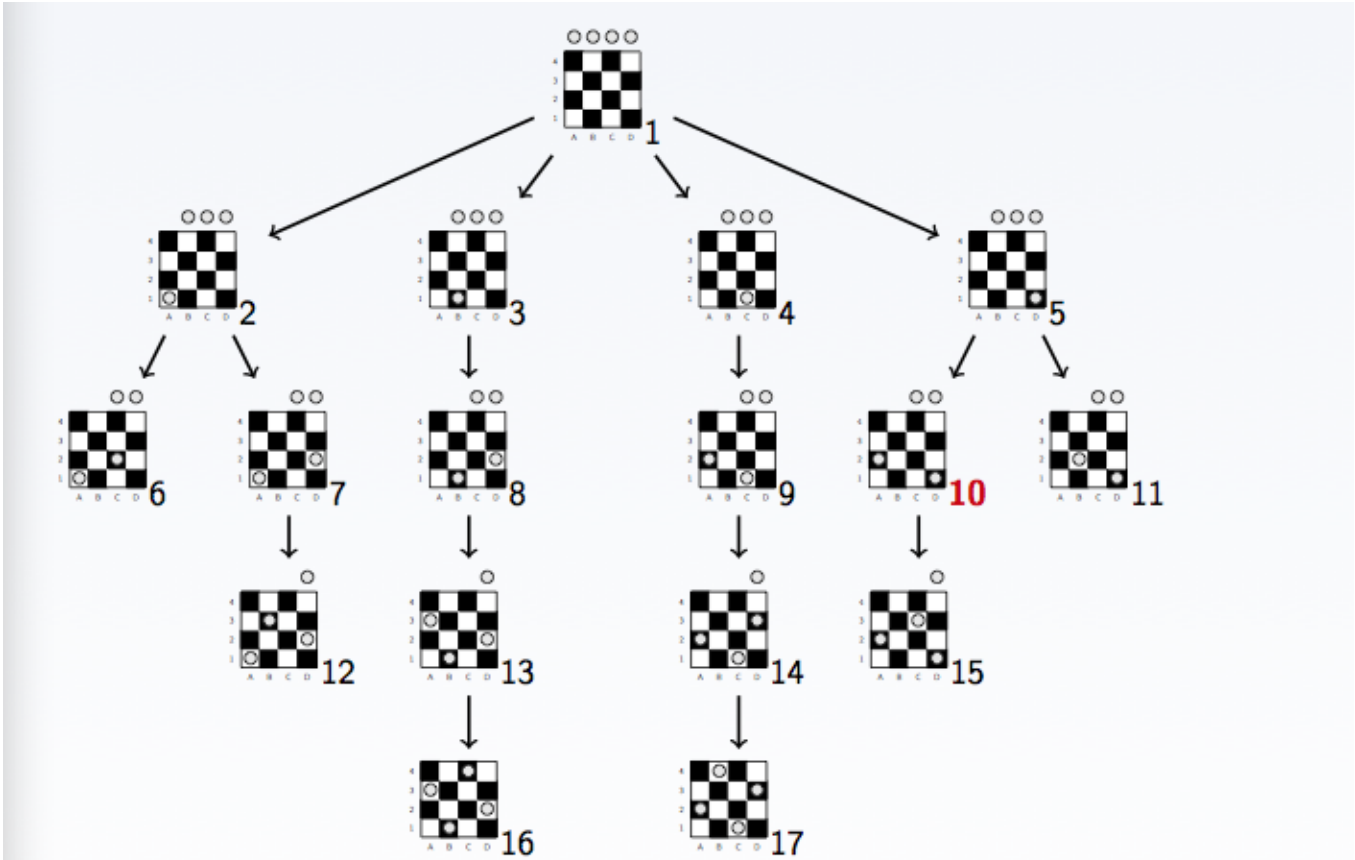


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

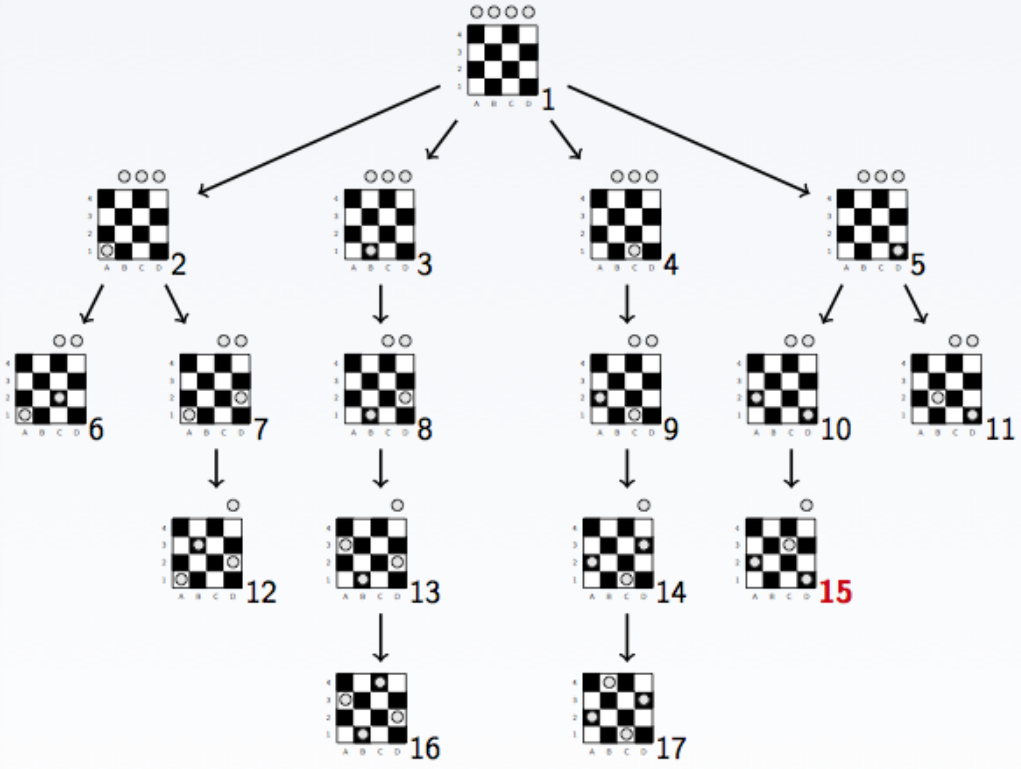


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

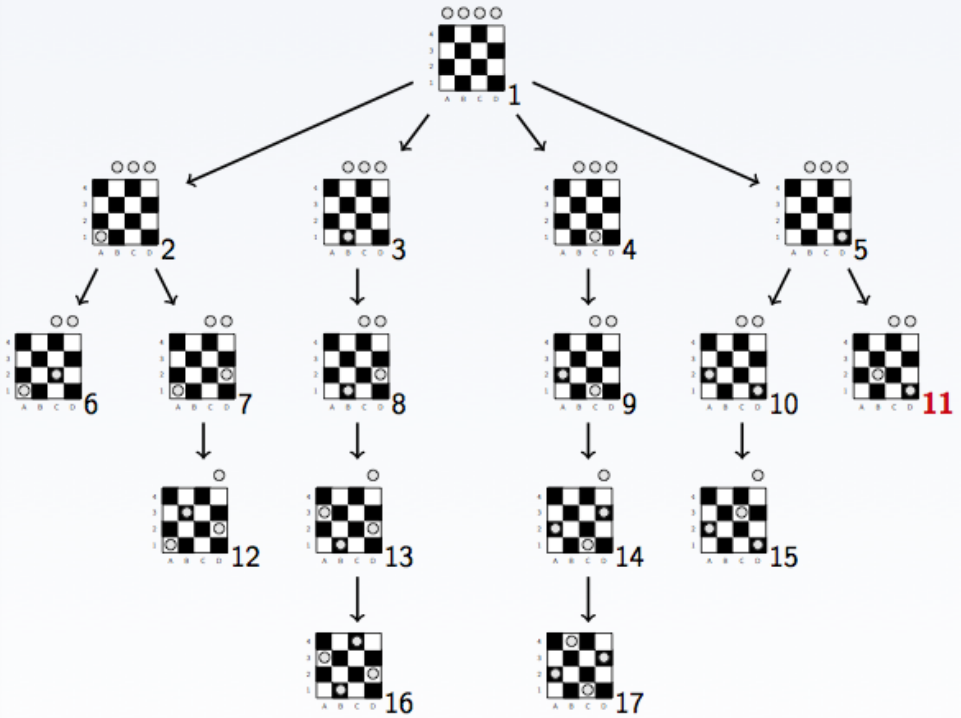


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

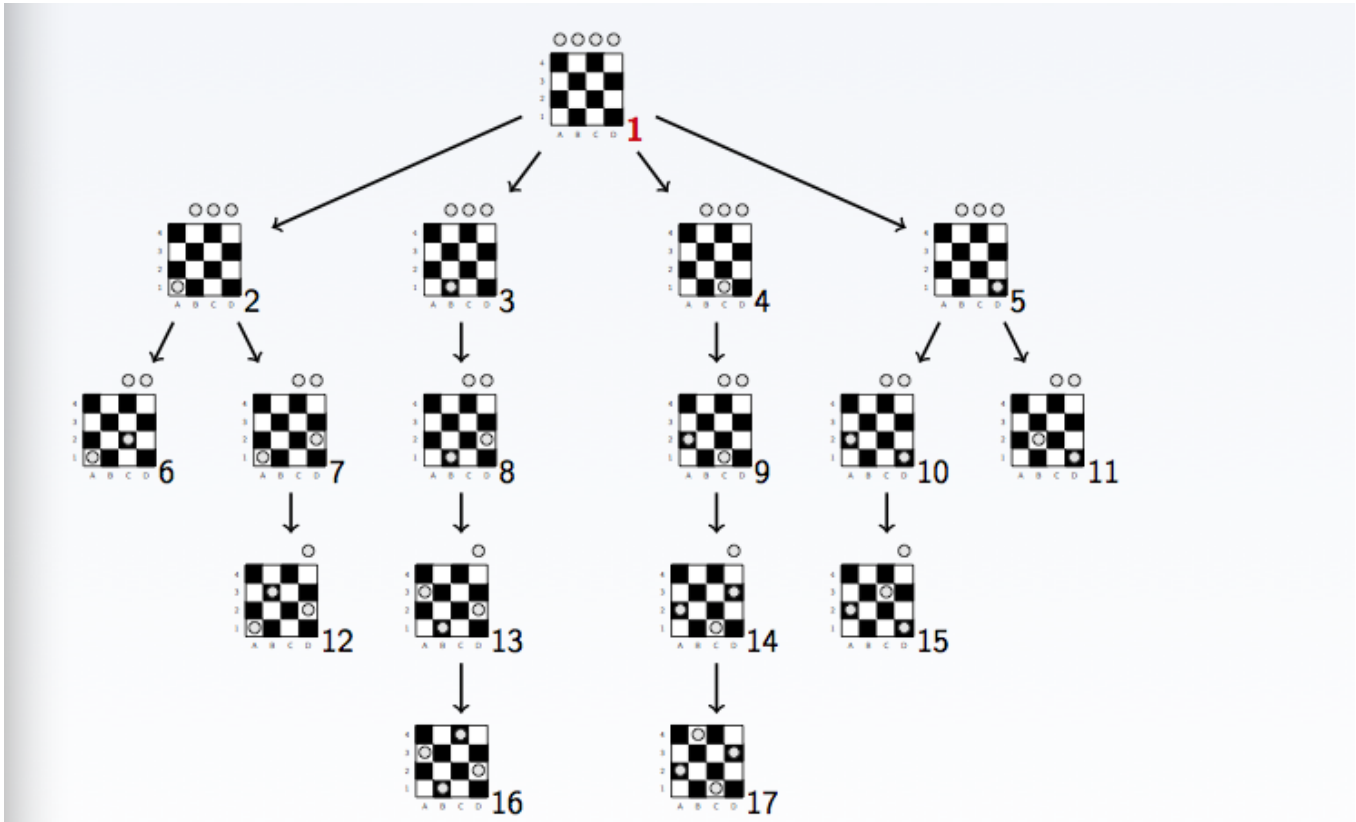


Reihenfolge der Knotenbesuche

$k = 4$

1,2,6,7,12,3,8,13,4,9,14,5,10,15,11

Beispiel: Iterative Tiefensuche

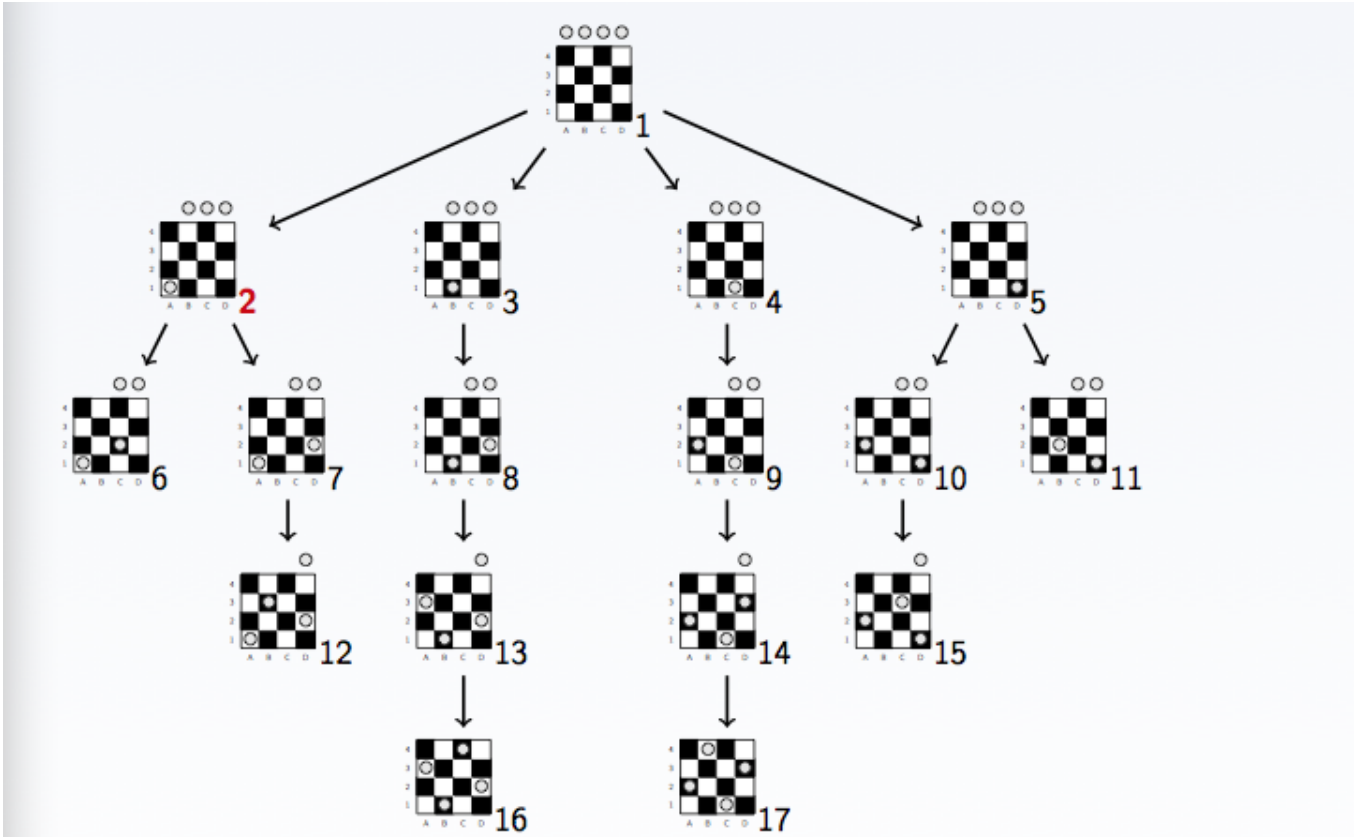


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

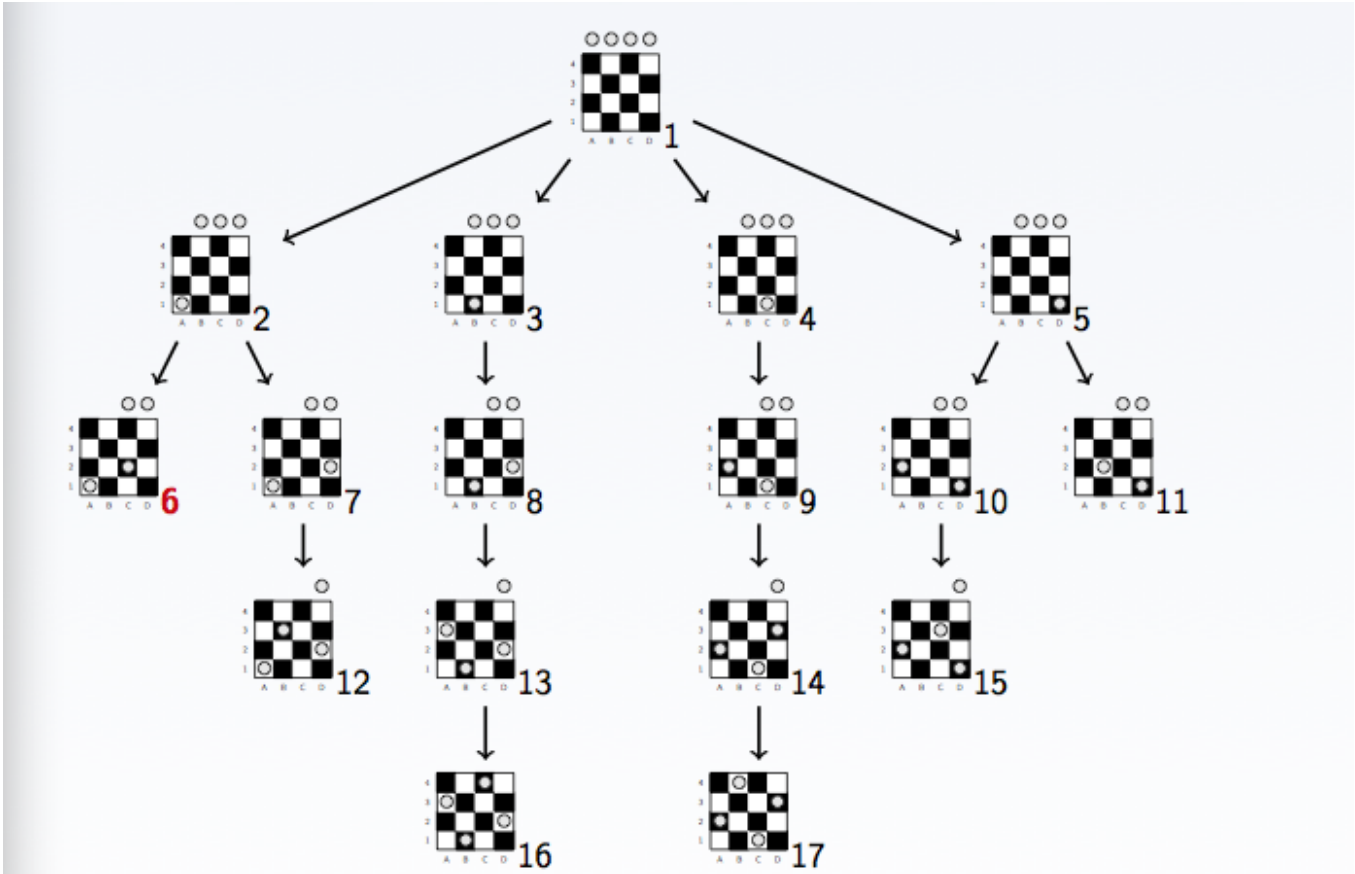


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

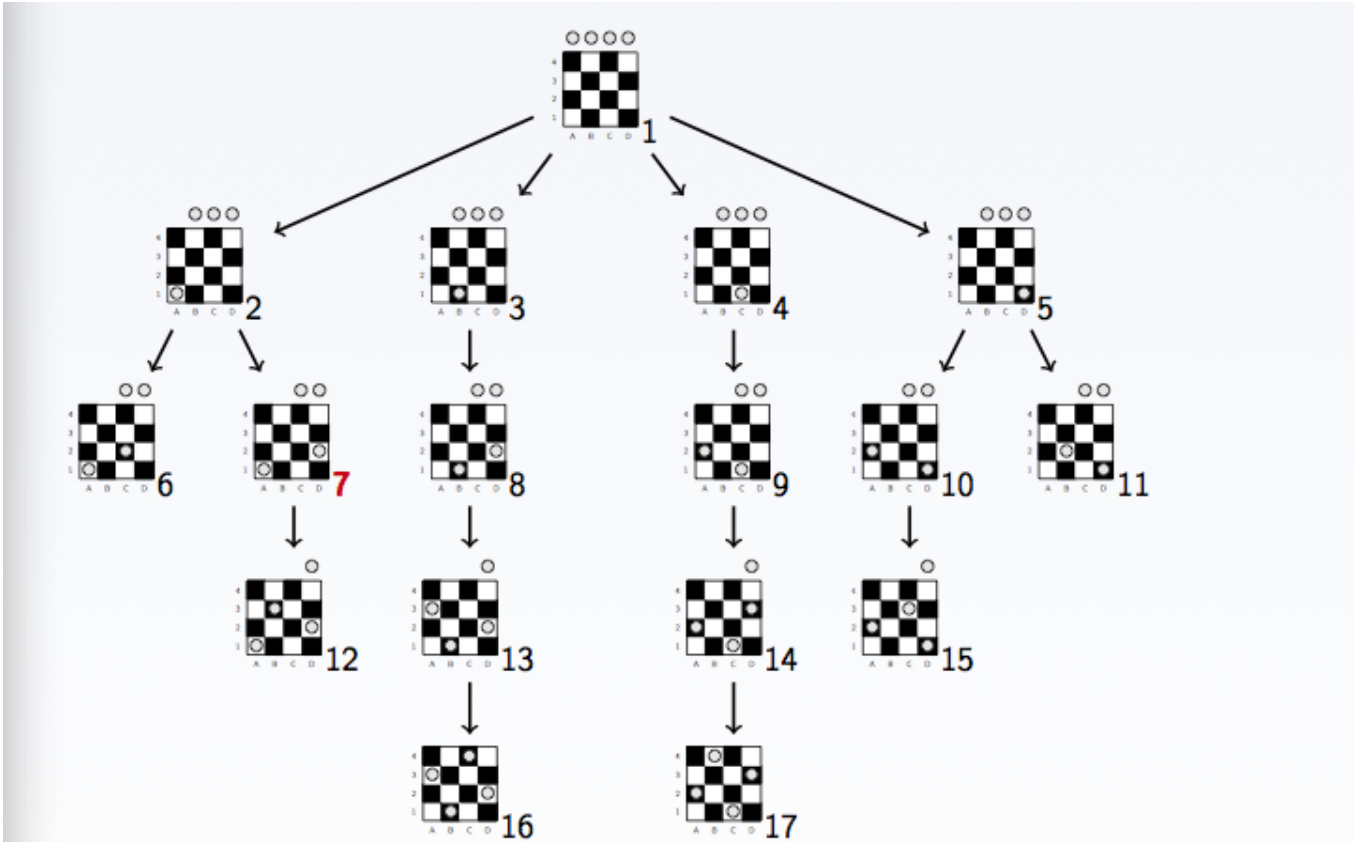


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

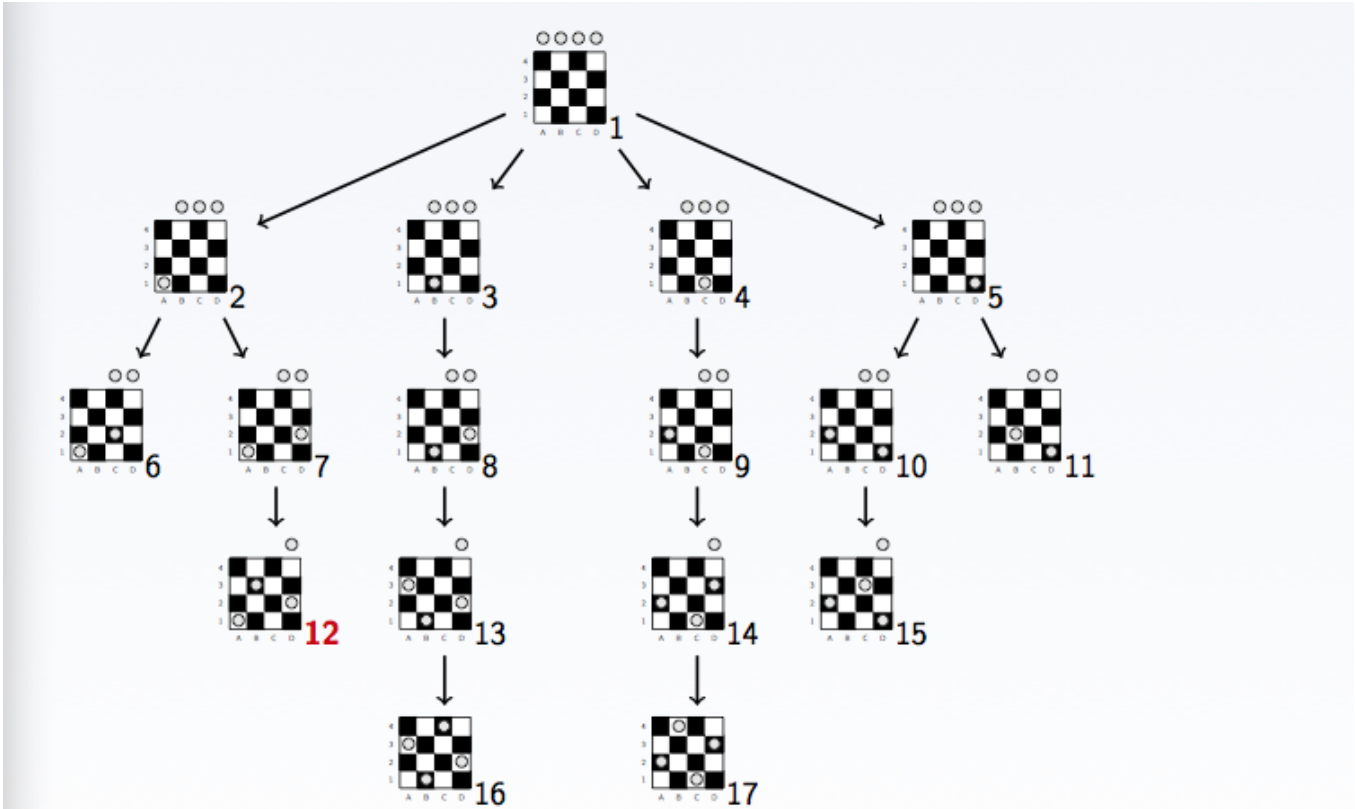


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

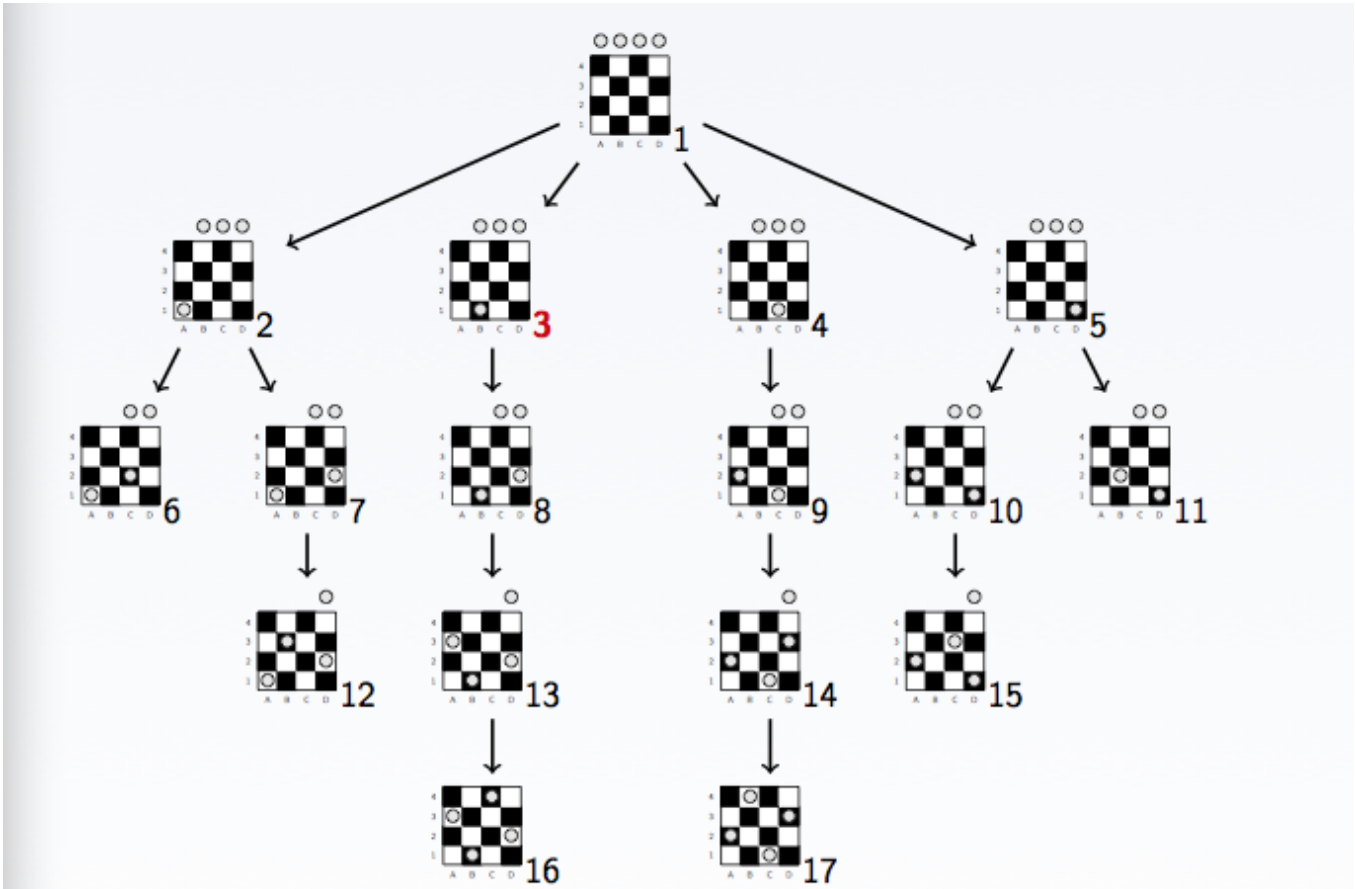


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

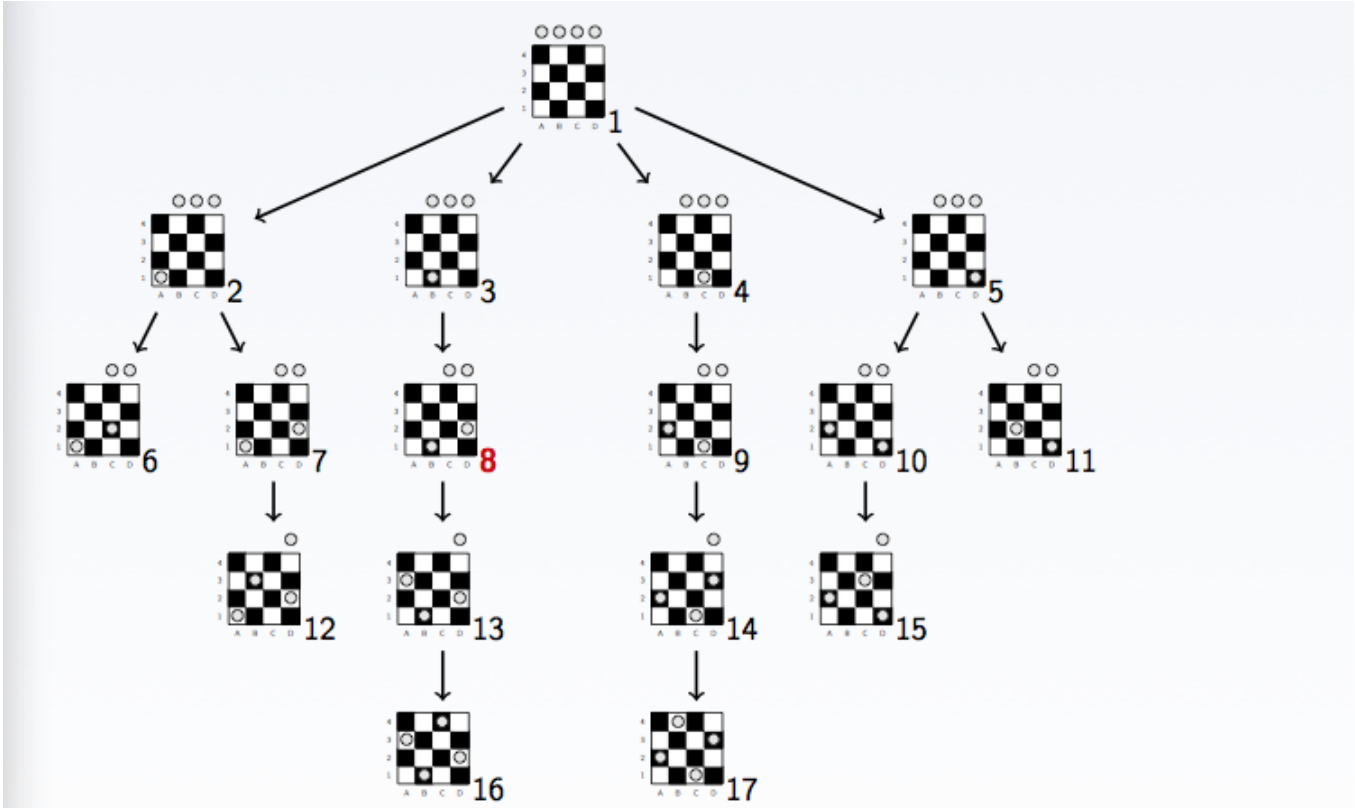


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

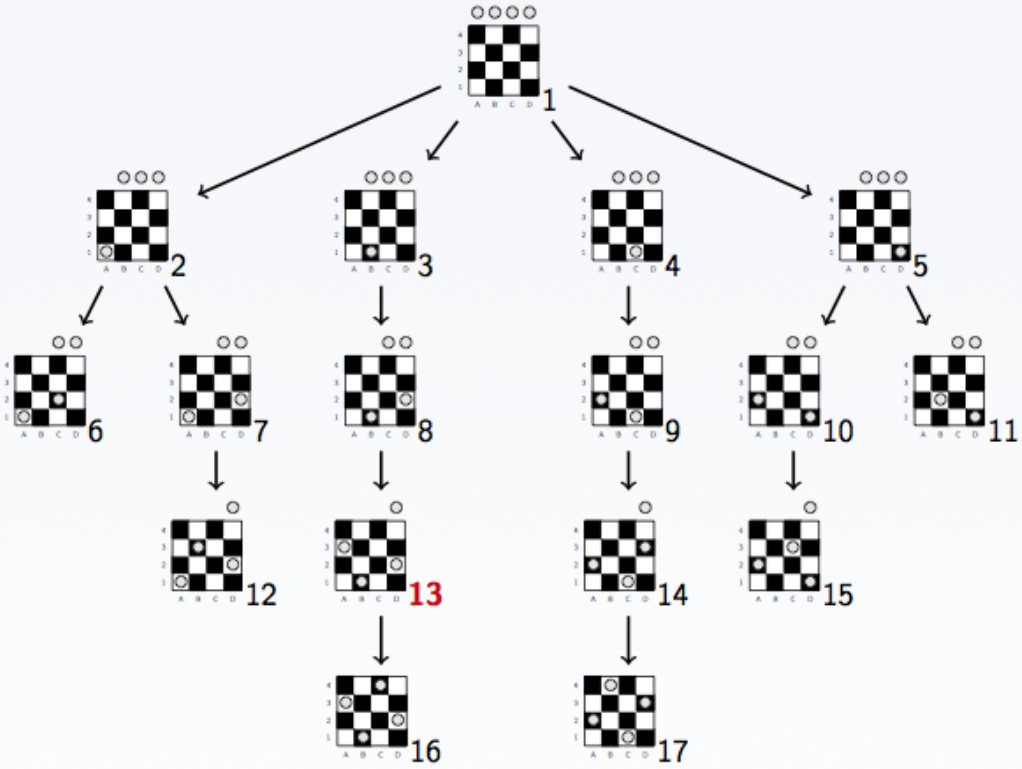


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

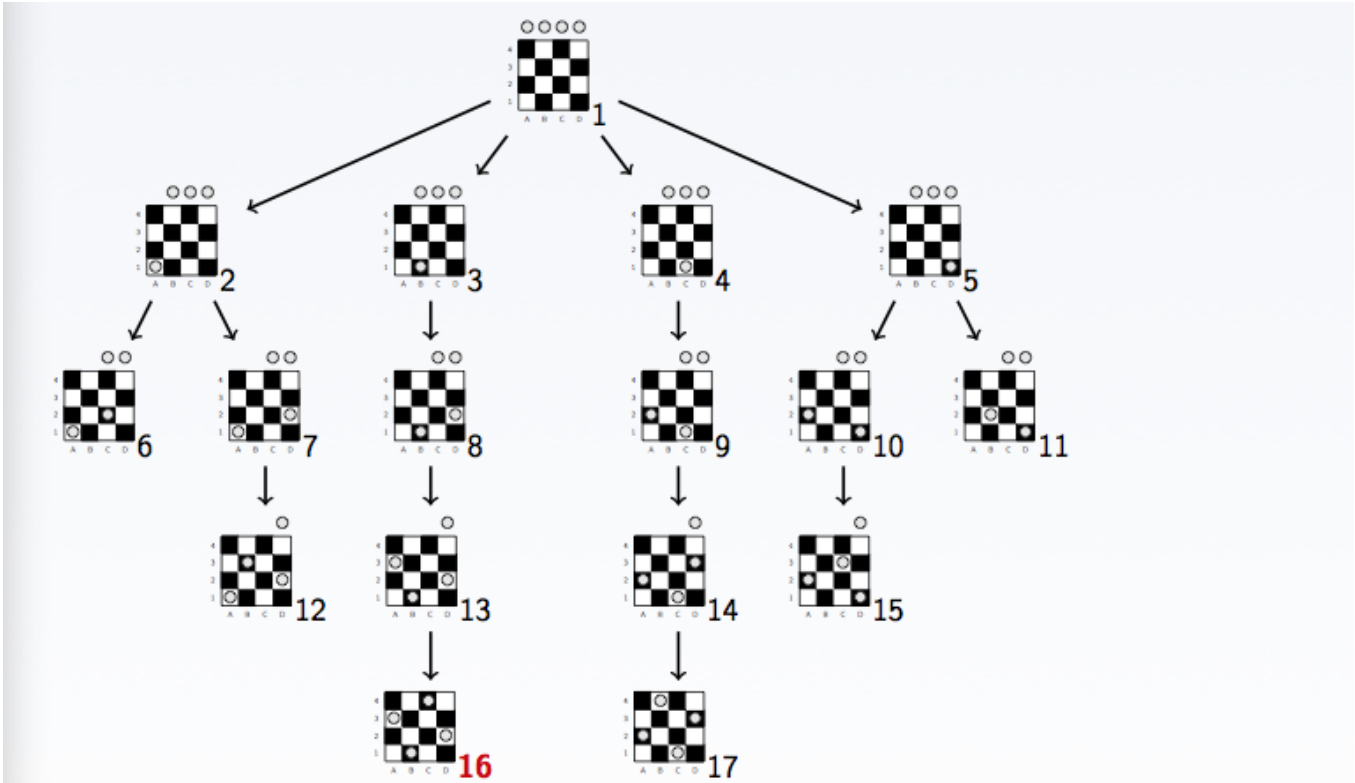


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche

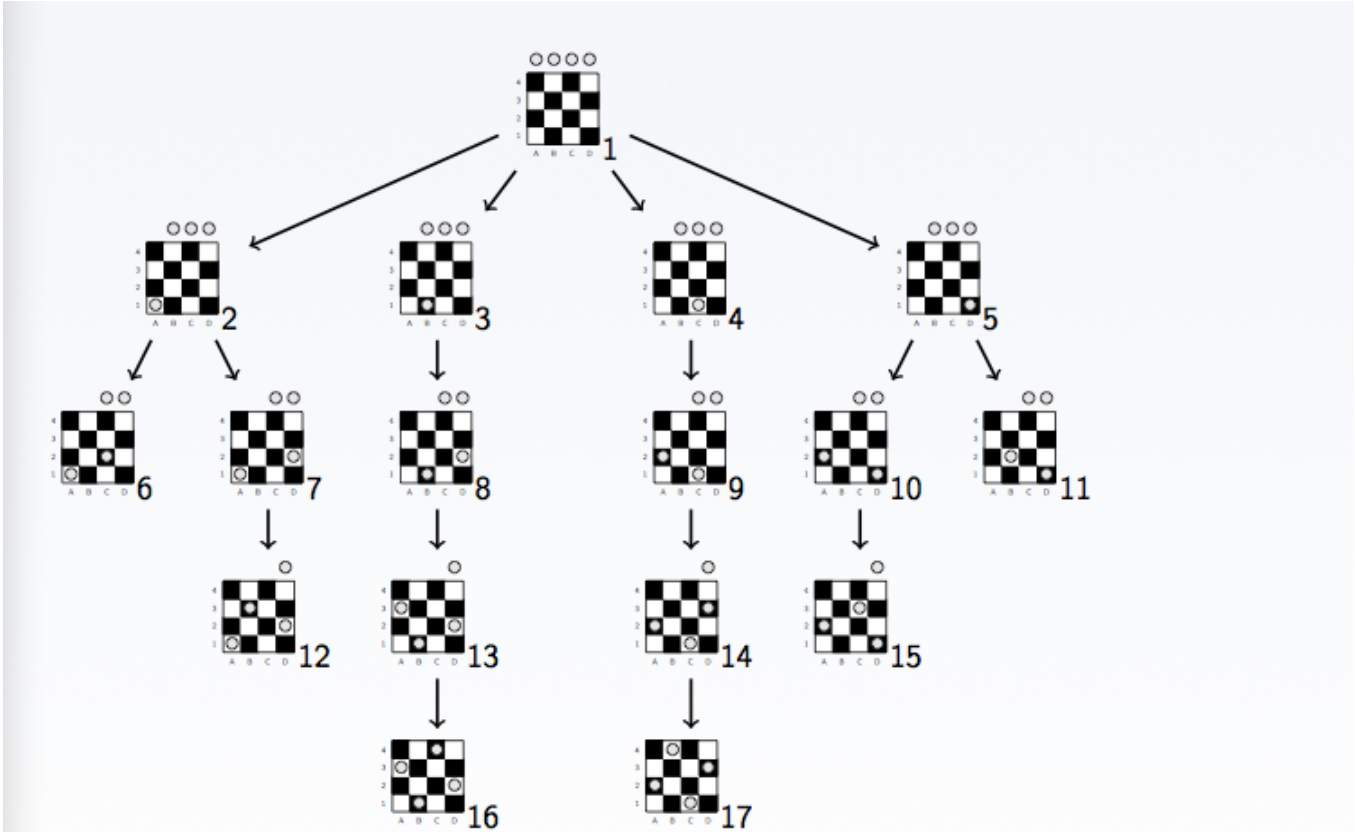


Reihenfolge der Knotenbesuche

$k = 5$

1,2,6,7,12,3,8,13,16 $\Rightarrow$ Ziel

Beispiel: Iterative Tiefensuche



Reihenfolge der Knotenbesuche

41 besuchte Knoten!

Eigenschaften

Komplexität (worst-case)

bei mittlerer Verzweigungsrate $c > 1$

- Platz: Linear in der Tiefe
- Zeit: ? (gleich)

Vollständigkeit

- Die iterative Tiefensuche ist vollständig (bei endlicher Verzweigungrate)

Zeitbedarf iteratives Vertiefen

Näherung: $\sum_{i=1}^{n} a^i \approx \frac{a^{n+1}}{a-1}$

Tiefensuche mit Tiefenbeschränkung k

- Alle Knoten besuchen: $\sum_{i=1}^{k} c^i$
- Im Mittel besuchte Knoten (Zielknoten in Tiefe k):

  $0.5 * (\sum_{i=1}^{k} c^i) \approx 0.5 * (\frac{c^{k+1}}{c-1})$

Iteratives Vertiefen bis Tiefe k, im Mittel, Zielknoten in Tiefe k:

$k-1$ Tiefensuchen für Tiefe $1, \ldots, k-1$

+Tiefensuche für Tiefe k (Hälfte der Knoten)

$$= \sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1})$$

Zeitbedarf iteratives Vertiefen

Näherung: $\sum_{i=1}^{n} a^i \approx \frac{a^{n+1}}{a-1}$

$$\sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) = \frac{\sum_{i=1}^{k-1} c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1})$$

$$= \frac{(\sum_{i=1}^{k} c^i) - c^k}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx \frac{1}{c-1}((\frac{c^{k+1}}{c-1}) - c^1) + 0.5 * (\frac{c^{k+1}}{c-1})$$

$$= (\frac{c^{k+1}}{(c-1)^2}) - \frac{c}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx (\frac{c^{k+1}}{(c-1)^2}) + 0.5 * (\frac{c^{k+1}}{c-1})$$

Zeitbedarf iteratives Vertiefen

Faktor $\frac{\text{Iteratives Vertiefen}}{\text{Tiefensuche bis } k}$:

$$\frac{\frac{c^{k+1}}{(c-1)^2} + 0.5 * (\frac{c^{k+1}}{c-1})}{0.5 * (\frac{c^{k+1}}{c-1})} = \frac{\frac{c^{k+1}}{(c-1)^2}}{0.5 * (\frac{c^{k+1}}{c-1})} + 1 = \frac{2}{c-1} + 1$$

Tabelle der ca.-Werte des Faktors $d = \frac{2}{c-1} + 1$ ist

c	2	3	4	5	...	10
d	3	2	1,67	1,5	...	1,22

Bemerkungen zur iterativen Tiefensuche

Allgemeine Idee dabei:

- Spare Platz, opfere Zeit (speichern vs. neu berechnen)
- Gegensätzlich zum dynamischen Programmieren:
  Dort: Opfere Platz für schnellere Zeit.

Varianten

Nutze Platz:

- Speichere Knoten, die bereits expandiert wurden, und betrachte sie nicht neu
  $\Rightarrow$ keine wiederholten Expansionen
- Speichere einen Teil des letzten Suchbaums (z.B. den linken Teil), damit er beim nächsten mal nicht betrachtet werden muss.
- Kombiniere Breitensuche mit Tiefensuche: erst in die Breite, ab dort dann Tiefensuche

Rückwärtssuche

Idee

- Suche nicht vom Start das Ziel, sondern umgekehrt
- Statt Nachfolgerfunktion benutze Vorgängerfunktion

Lohnswert?

- Lohnt, wenn die Verzweigungsrate der Vorgängerfunktion kleiner ist, als die der Nachfolgerfunktion
- Problematisch: Finde algorithmische Beschreibung der Vorgängerfunktion

Bidirektionale Suche

- Suche vorwärts und rückwärts
- Erfordert Vergleich der momentan expandierten Knoten (Schnittmenge)
- Vorteil z.B. bei Breitensuche: Platz: statt c^d nur $2 * (c^{d/2})$ $\approx$ Doppelt so tief suchen in gleichem Platz (wenn #Eingangsknoten $\approx$ #Ausgangsknoten)
- Nachteil: Schnittbildung (Zeitintensiv), und Vorgänger- und Nachfolgerfunktion nötig.