Tafelübung 08
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2019

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Übersicht

Backtracking

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Backtracking: Motivation

Beispiel: Schatzsuche

In einem Labyrinth ist ein Schatz versteckt. Da Alice und Bob bereits mit ihrer AuD-Hausaufgabe für diese Woche fertig sind, beschließen sie, ihn zu suchen.

Mögliche Vorgehensweisen

- planlos/zufällig abbiegen und hoffen, den Schatz irgendwann zu finden
- sich aufteilen und parallel an verschiedenen Stellen suchen
  $\leadsto$ *Parallele und Funktionale Programmierung*
- strukturiert alle möglichen Pfade der Reihe nach durchprobieren
- ...

Backtracking: Motivation

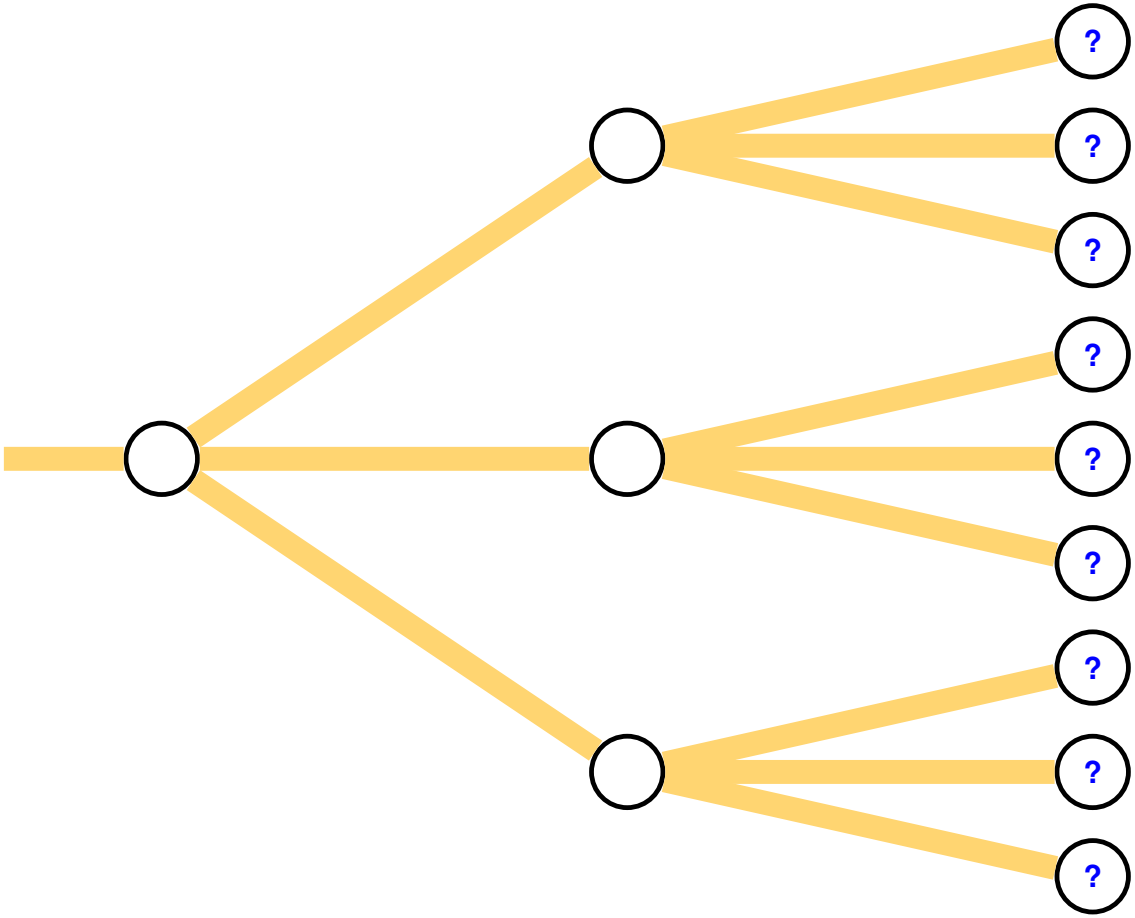
Beispiel: Schatzsuche

In einem Labyrinth ist ein Schatz versteckt. Da Alice und Bob bereits mit ihrer AuD-Hausaufgabe für diese Woche fertig sind, beschließen sie, ihn zu suchen.

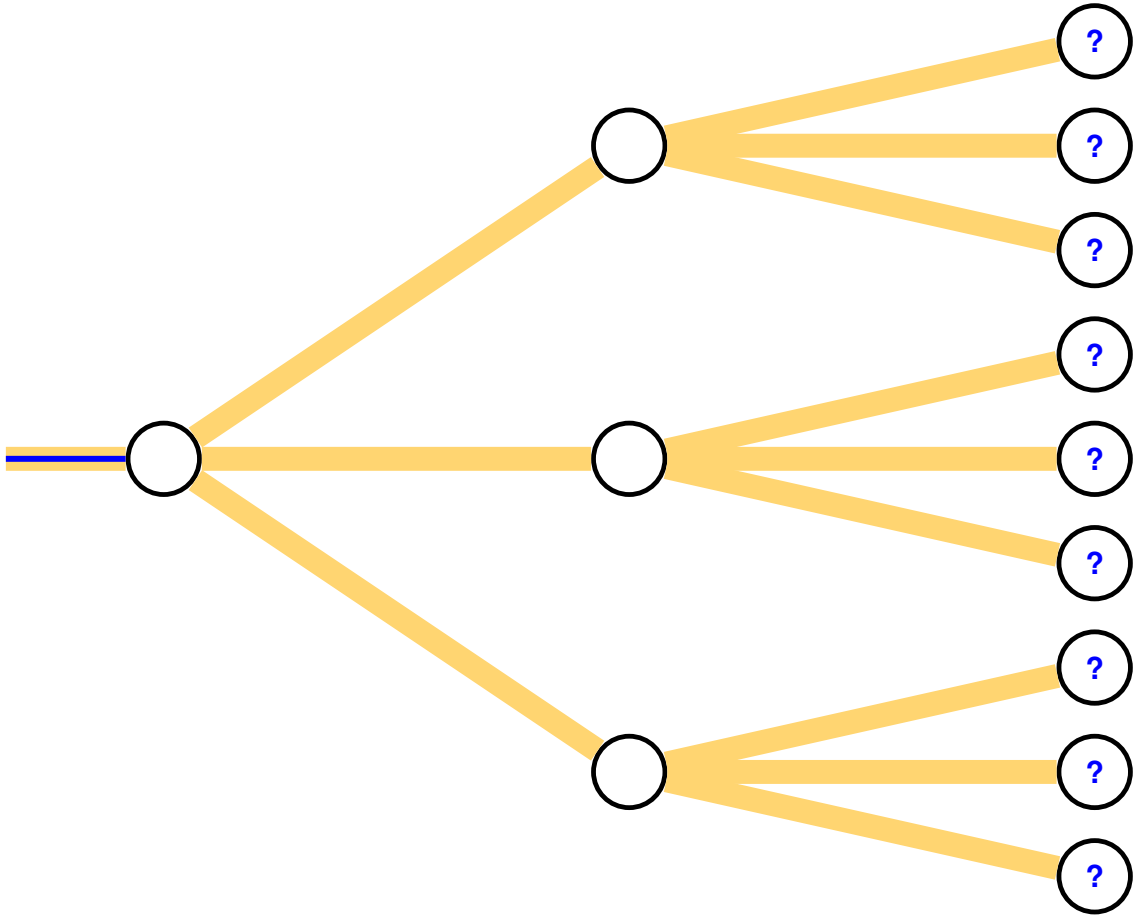
Mögliche Vorgehensweisen

- planlos/zufällig abbiegen und hoffen, den Schatz irgendwann zu finden
- sich aufteilen und parallel an verschiedenen Stellen suchen
  $\leadsto$ *Parallele und Funktionale Programmierung*
- strukturiert alle möglichen Pfade der Reihe nach durchprobieren
- ...

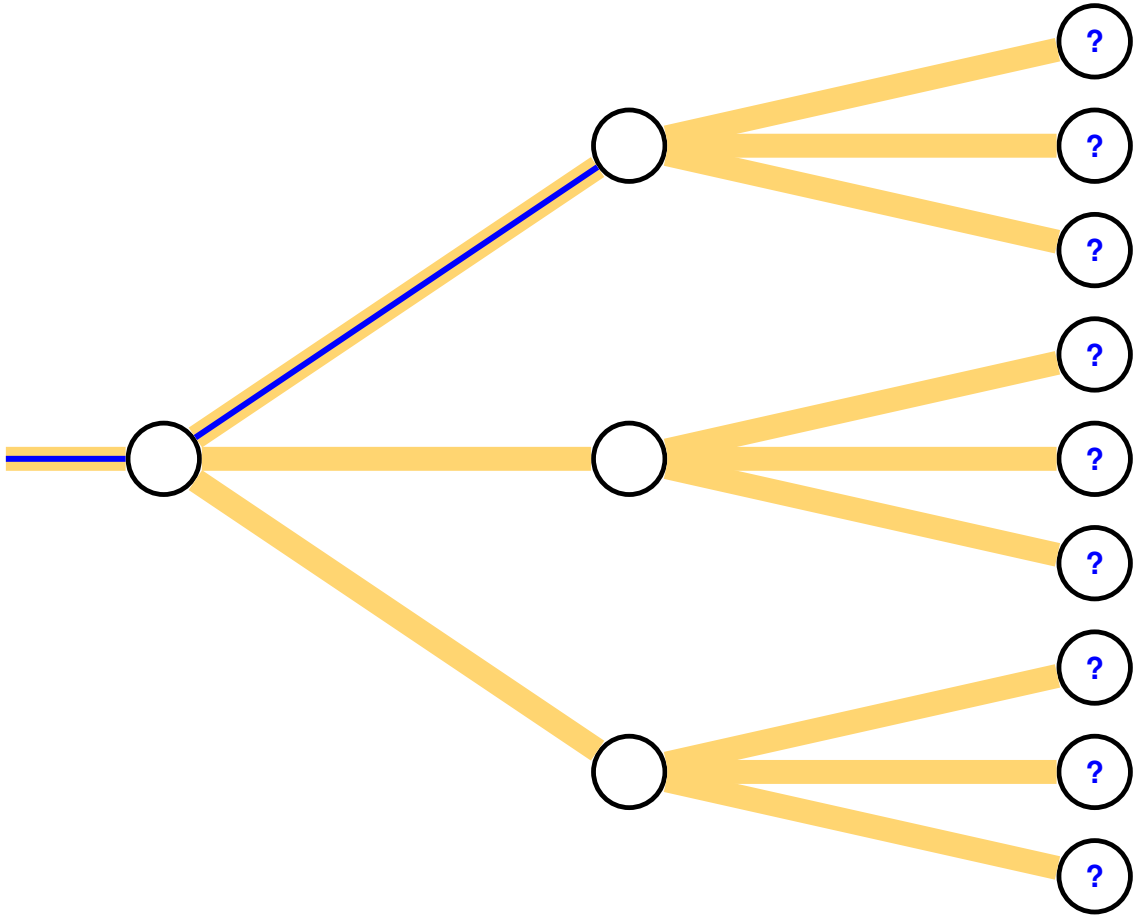
Schatzsuche: Alle möglichen Pfade durchprobieren



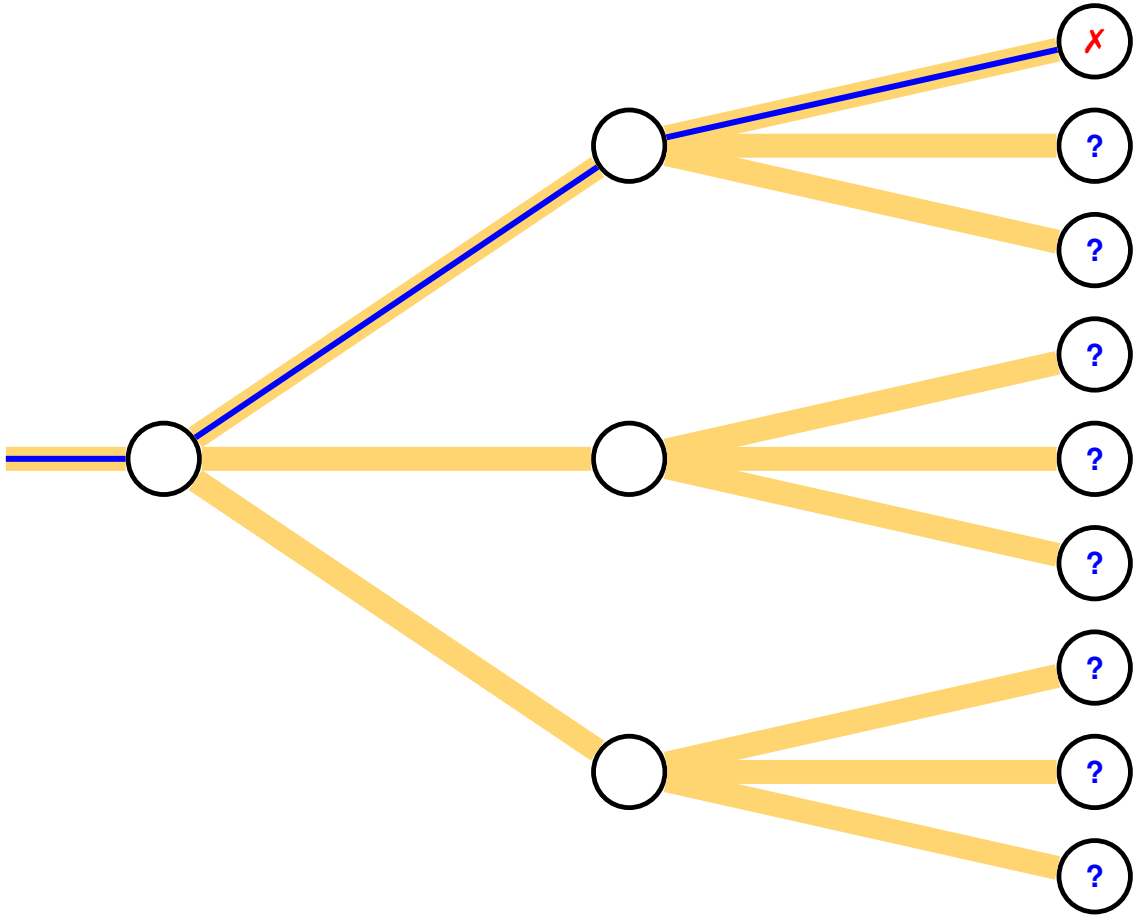
Schatzsuche: Alle möglichen Pfade durchprobieren



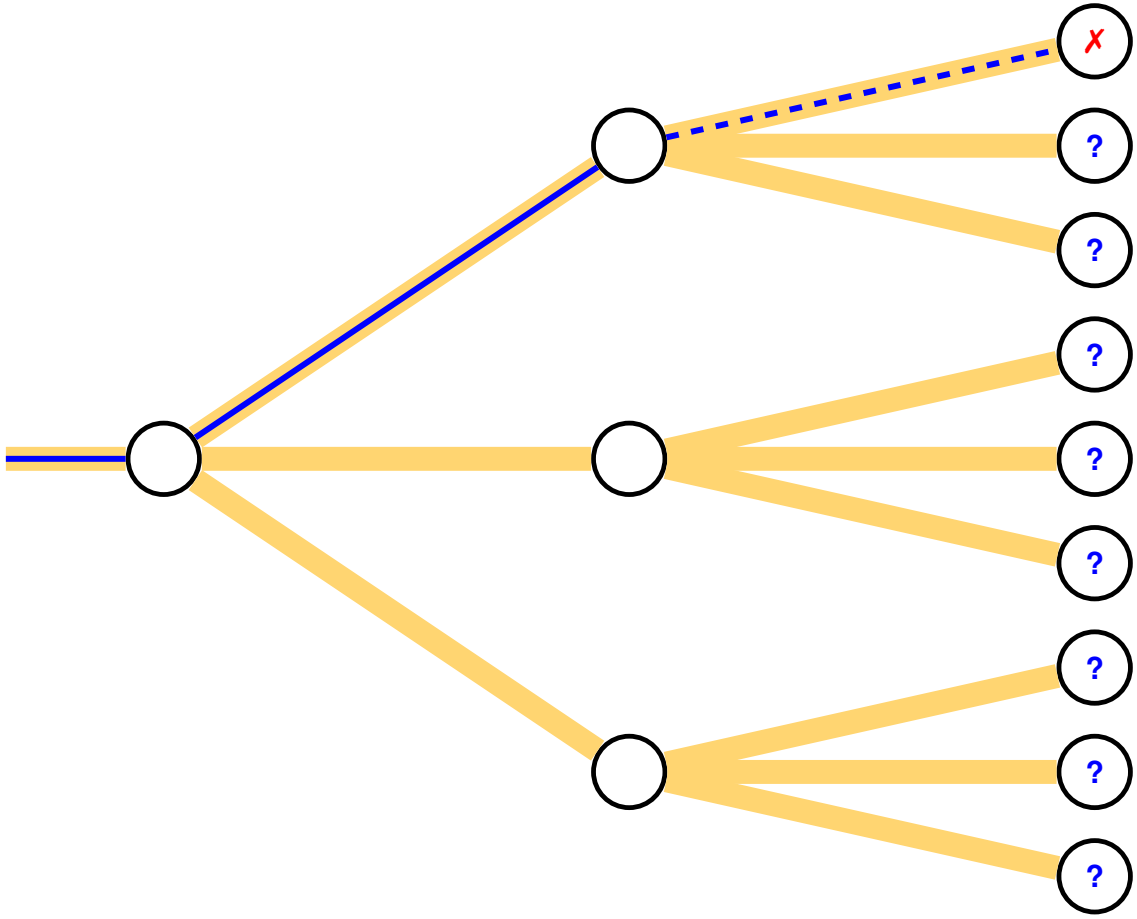
Schatzsuche: Alle möglichen Pfade durchprobieren



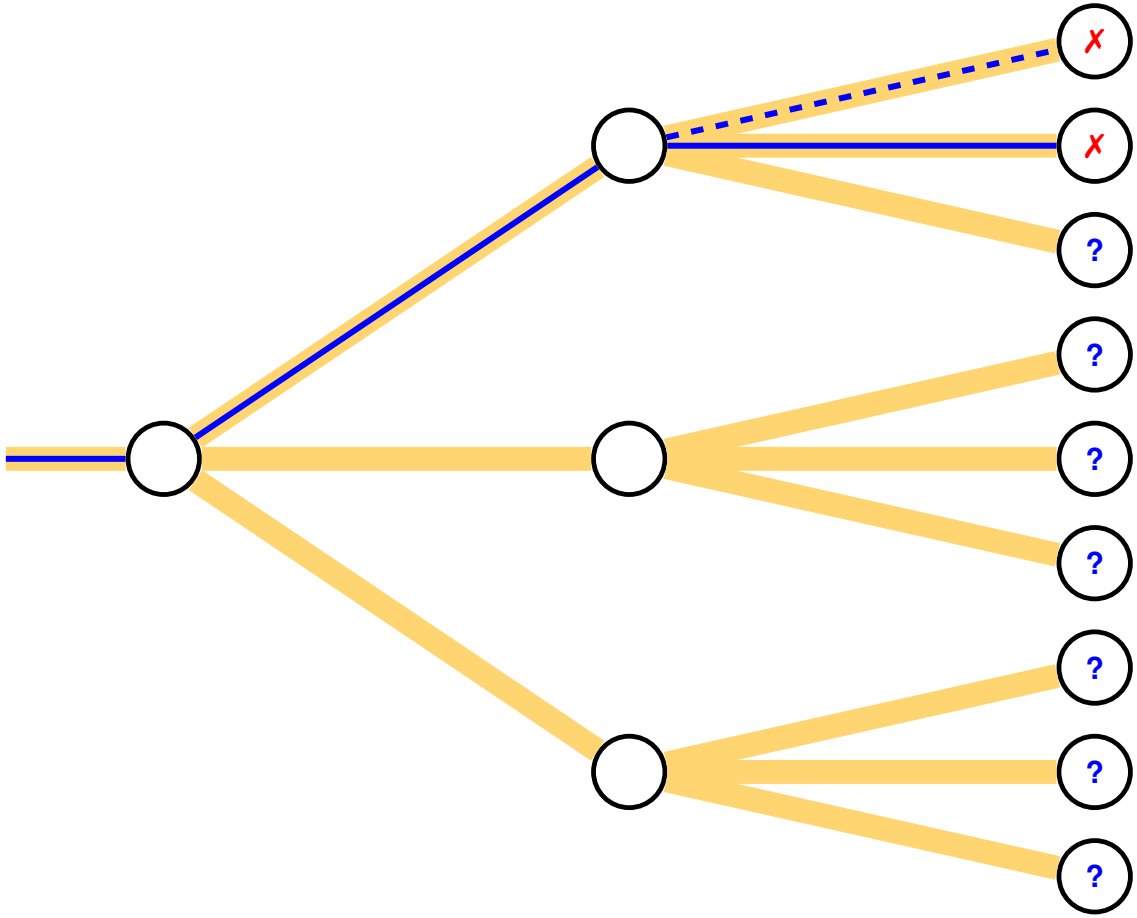
Schatzsuche: Alle möglichen Pfade durchprobieren



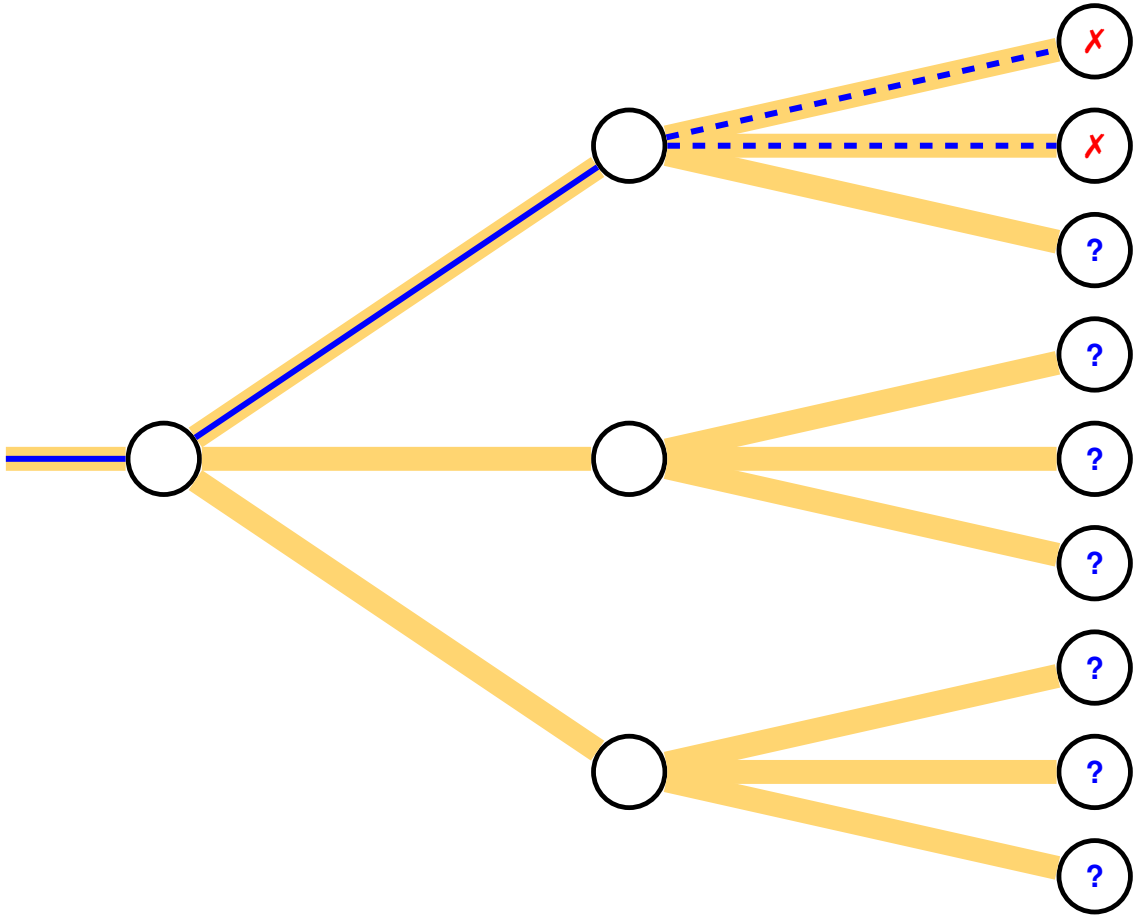
Schatzsuche: Alle möglichen Pfade durchprobieren



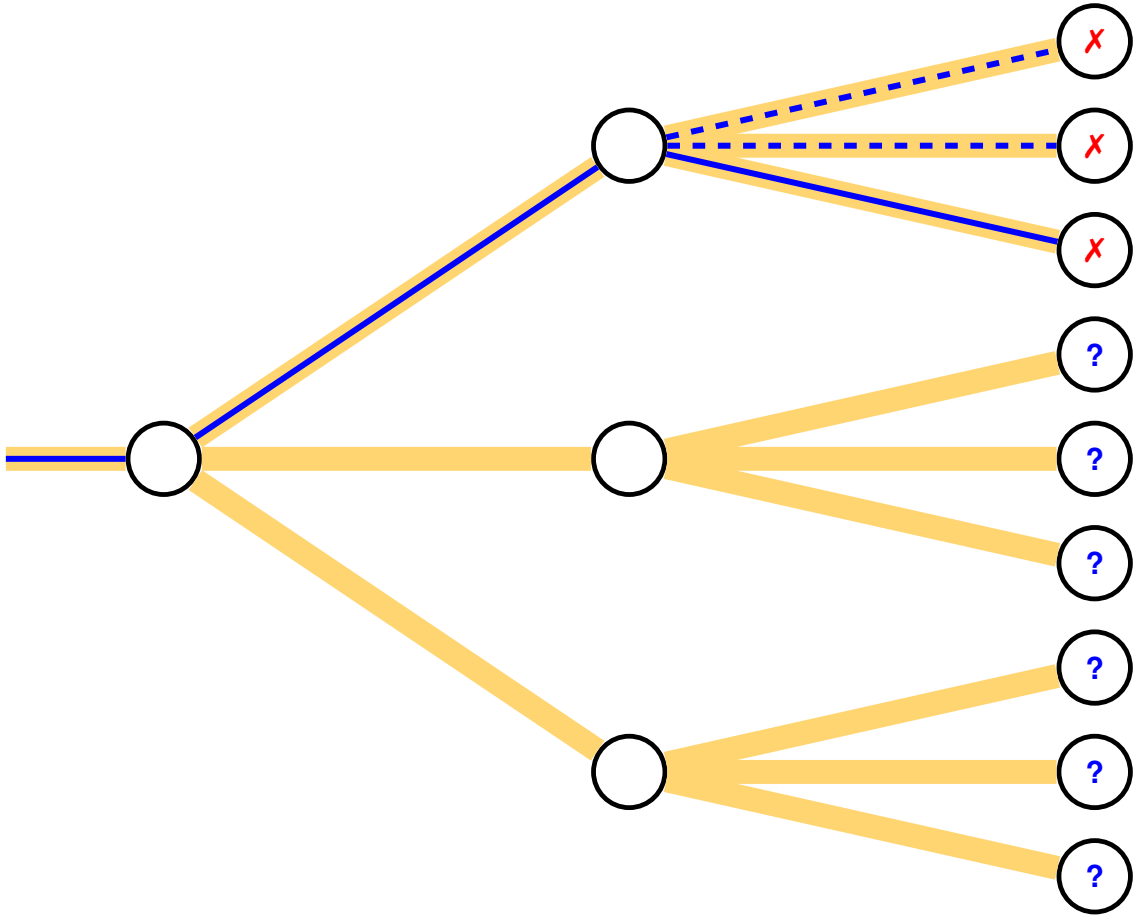
Schatzsuche: Alle möglichen Pfade durchprobieren



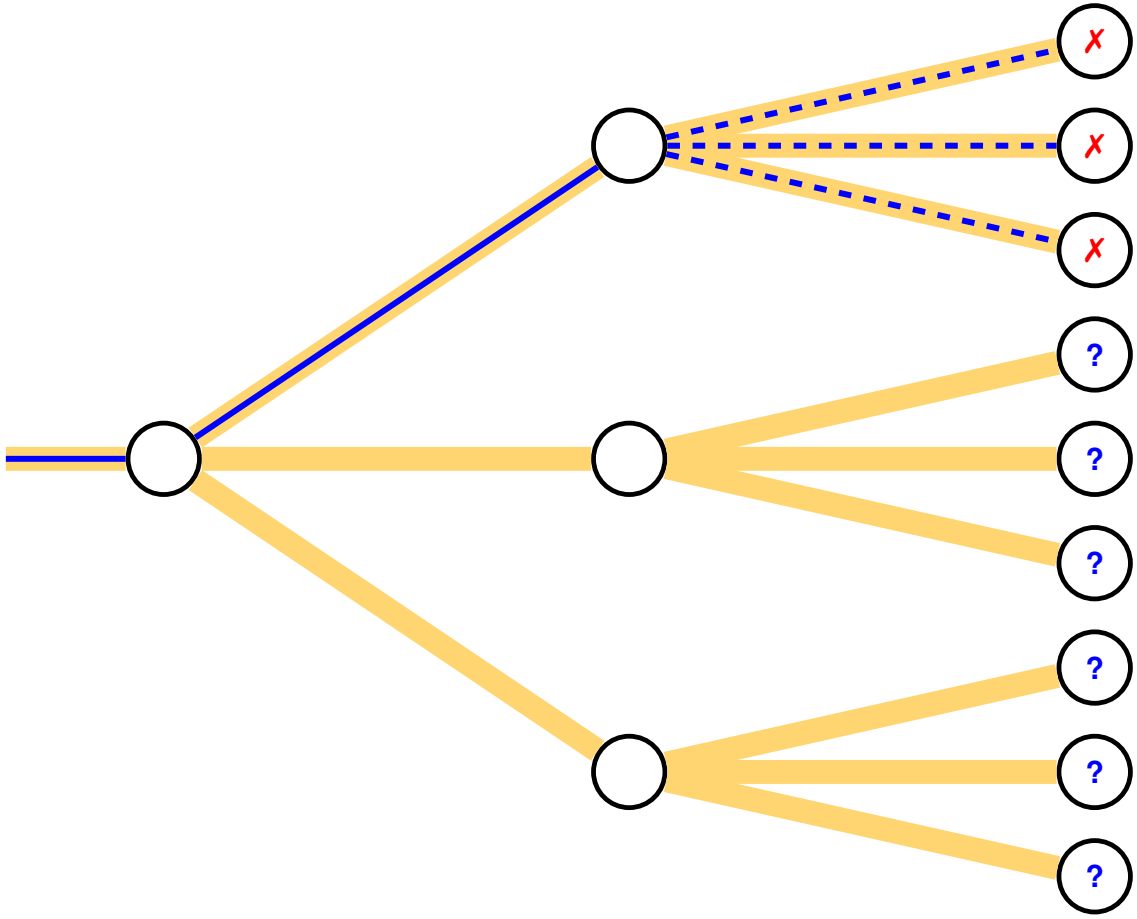
Schatzsuche: Alle möglichen Pfade durchprobieren



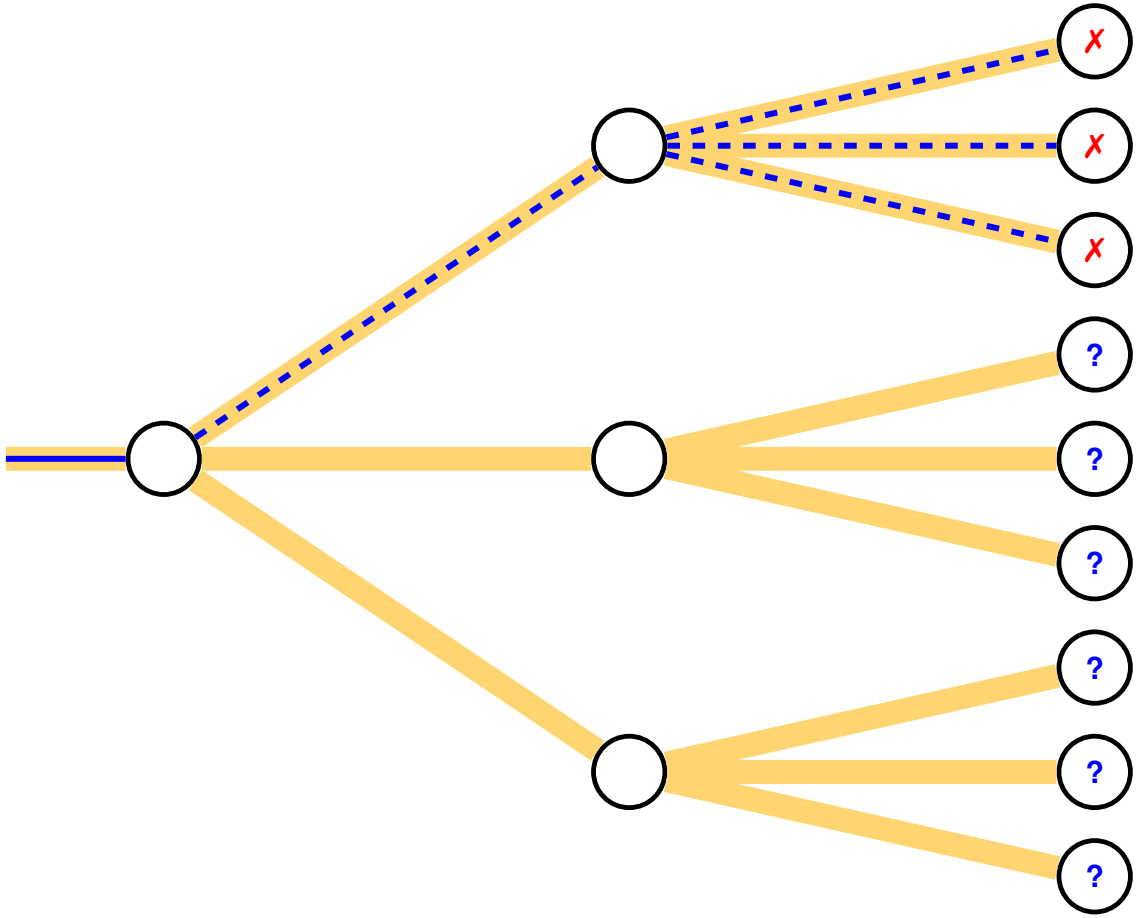
Schatzsuche: Alle möglichen Pfade durchprobieren



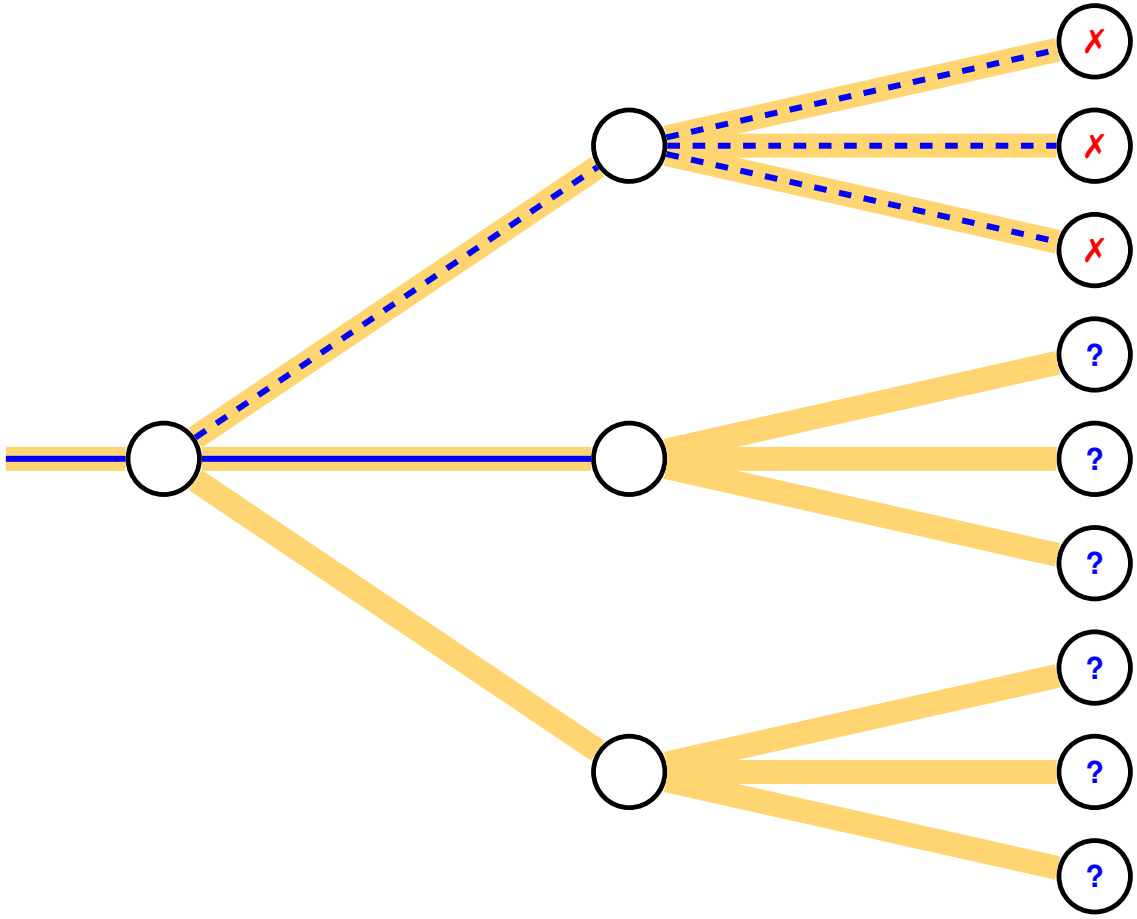
Schatzsuche: Alle möglichen Pfade durchprobieren



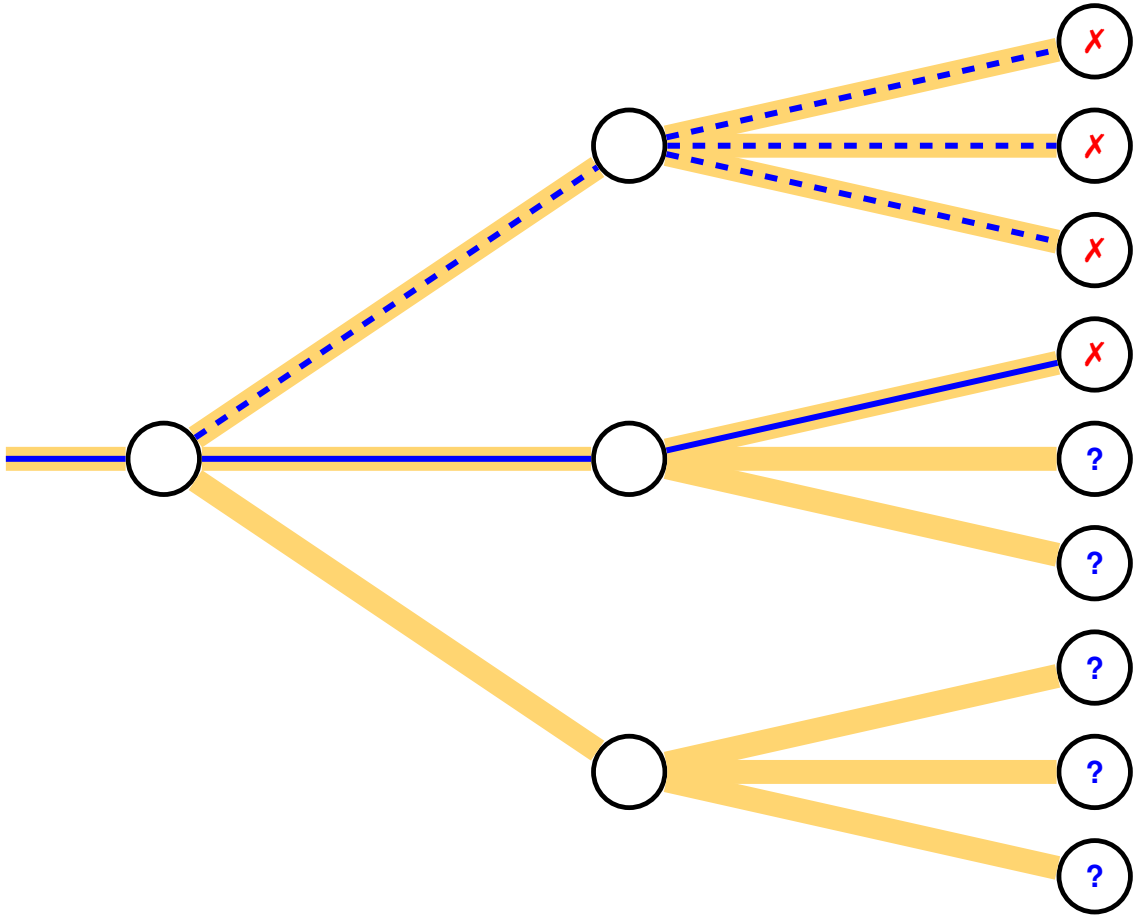
Schatzsuche: Alle möglichen Pfade durchprobieren



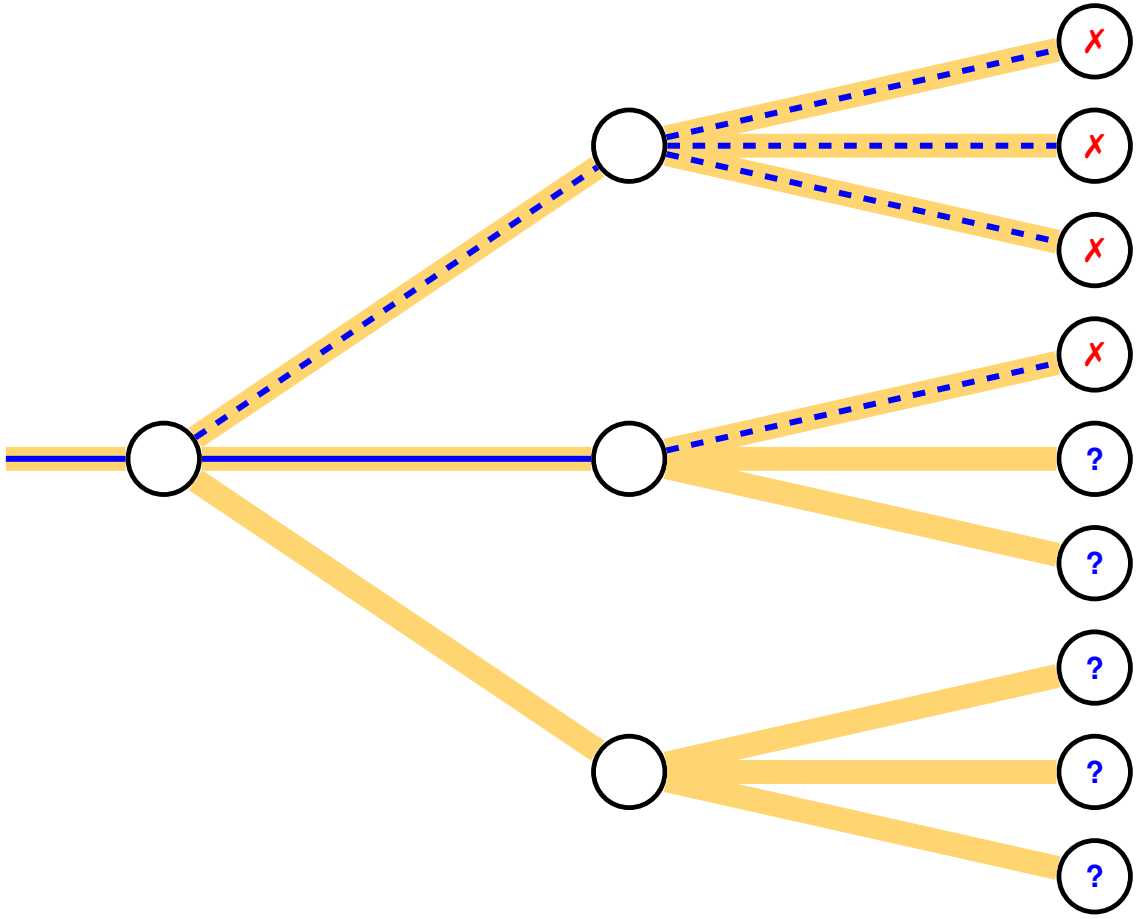
Schatzsuche: Alle möglichen Pfade durchprobieren



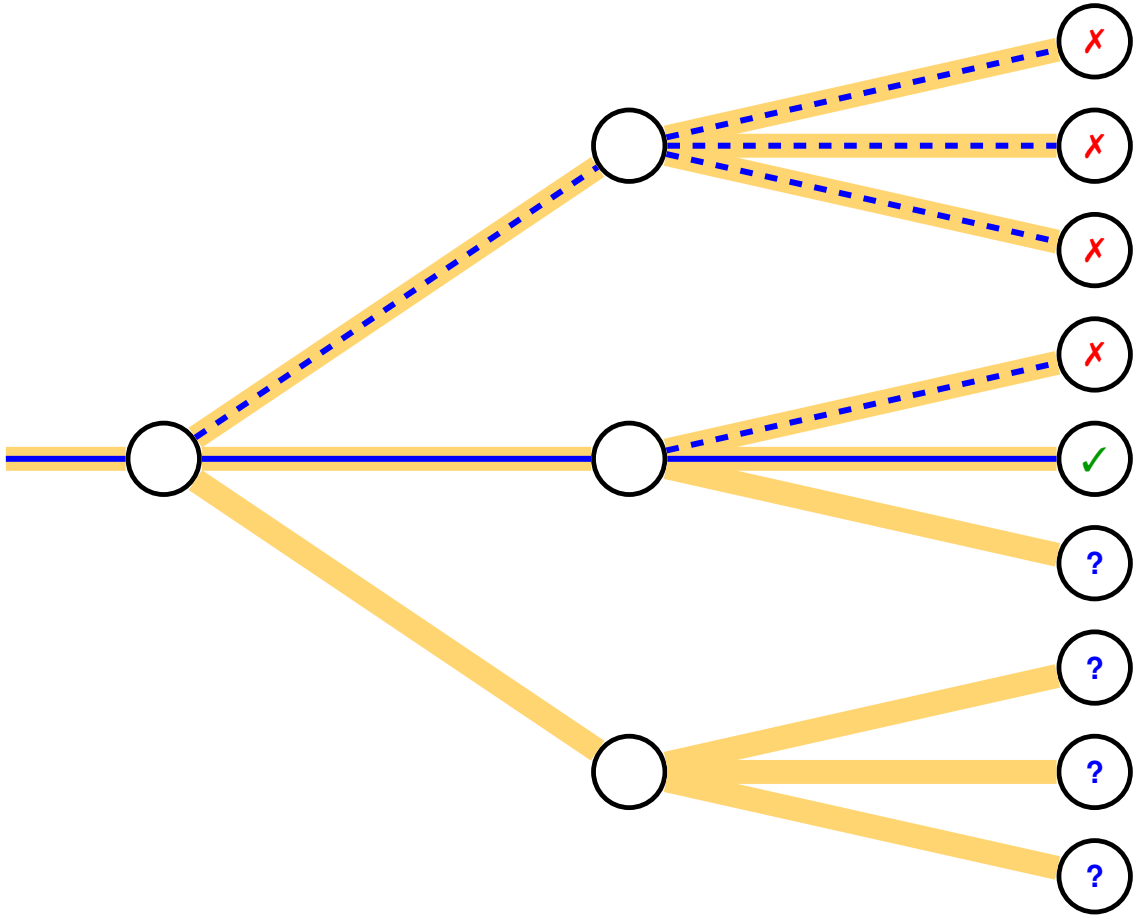
Schatzsuche: Alle möglichen Pfade durchprobieren



Schatzsuche: Alle möglichen Pfade durchprobieren



Schatzsuche: Alle möglichen Pfade durchprobieren



Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

$\rightsquigarrow$ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking: Allgemeines Schema

- Backtracking läuft fast immer nach demselben Schema ab

$\rightsquigarrow$ wenn man sich passende Methoden definiert, kann man für die Implementierung oft dasselbe Grundgerüst verwenden

- erforderliche Methoden sind:
  - `isFinal()` überprüft, ob eine Lösung gefunden wurde
    $\rightsquigarrow$ z.B.: ist ein Sudoku vollständig gefüllt?
  - `getExtensions()` gibt alle möglichen Erweiterungen zurück
    $\rightsquigarrow$ z.B.: alle erlaubte Zahlen für das aktuelle Feld
  - `apply()` verändert den aktuellen Zustand
    $\rightsquigarrow$ z.B.: schreibe die aktuelle Zahl ins aktuelle Feld
  - `revert()` stellt den vorherigen Zustand wieder her
    $\rightsquigarrow$ z.B.: lösche die letzte Zahl aus dem aktuellen Feld

Backtracking: Allgemeines Schema

- Backtracking läuft fast immer nach demselben Schema ab

$\rightsquigarrow$ wenn man sich passende Methoden definiert, kann man für die Implementierung oft dasselbe Grundgerüst verwenden

- erforderliche Methoden sind:
  - `isFinal()` überprüft, ob eine Lösung gefunden wurde

  $\rightsquigarrow$ z.B.: ist ein Sudoku vollständig gefüllt?
  - `getExtensions()` gibt alle möglichen Erweiterungen zurück

  $\rightsquigarrow$ z.B.: alle erlaubte Zahlen für das aktuelle Feld
  - `apply()` verändert den aktuellen Zustand

  $\rightsquigarrow$ z.B.: schreibe die aktuelle Zahl ins aktuelle Feld
  - `revert()` stellt den vorherigen Zustand wieder her

  $\rightsquigarrow$ z.B.: lösche die letzte Zahl aus dem aktuellen Feld

Backtracking: Grundgerüst

Backtracking: Grundgerüst

```
static int[][] backtrack(int[][] state) { // z.B. Sudoku
  if (isFinal(state)) { // z.B. Sudoku komplett gefüllt
    return state;
  } else {
    // z.B. erlaubte Zahlen für aktuelles Feld
    int[] candidates = getExtensions(state);
    for (int i = 0; i < candidates.length; i++) {
      int c = candidates[i];
      state = apply(state, c); // z.B. Zahl setzen
      if (backtrack(state) != null) { // Rekursion
        return state;
      }
      state = revert(state, c); // z.B. Zahl löschen
    }
    return null; // keine Lösung gefunden -> Schritt zurück
  }
}
```

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Motivation

Zusätzliche Anforderungen

Beim Durchsuchen eines Suchraums mittels Backtracking genügt es oft nicht, nur herauszufinden, *ob* eine Lösung existiert. Deswegen werden häufig zwei zusätzliche Anforderungen an den Backtrackingalgorithmus gestellt:

- Finden eines Lösungswegs
- Lösungsweg soll minimal sein

Beispiel: Labyrinth

- **bisher**: Wurde der Schatz gefunden?
- **jetzt**: Wo muss entlanggelaufen werden, um den Schatz zu erreichen, und ist der Weg dorthin minimal?

Motivation

Zusätzliche Anforderungen

Beim Durchsuchen eines Suchraums mittels Backtracking genügt es oft nicht, nur herauszufinden, *ob* eine Lösung existiert. Deswegen werden häufig zwei zusätzliche Anforderungen an den Backtrackingalgorithmus gestellt:

- Finden eines Lösungswegs
- Lösungsweg soll minimal sein

Beispiel: Labyrinth

- **bisher**: Wurde der Schatz gefunden?
- **jetzt**: Wo muss entlanggelaufen werden, um den Schatz zu erreichen, und ist der Weg dorthin minimal?

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

⇝ Backtracking findet garantiert eine Lösung, so sie denn existiert

Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
    - falls eine Möglichkeiten zum Erfolg führt: gut ☺
      - Suche kann abgebrochen werden *(außer man will alle Lösungen finden)*
    - andernfalls:
      - Entscheidung war wohl falsch...
      - Entscheidung „rückgängig machen“

$\rightsquigarrow$ Backtracking findet garantiert eine Lösung, so sie denn existiert

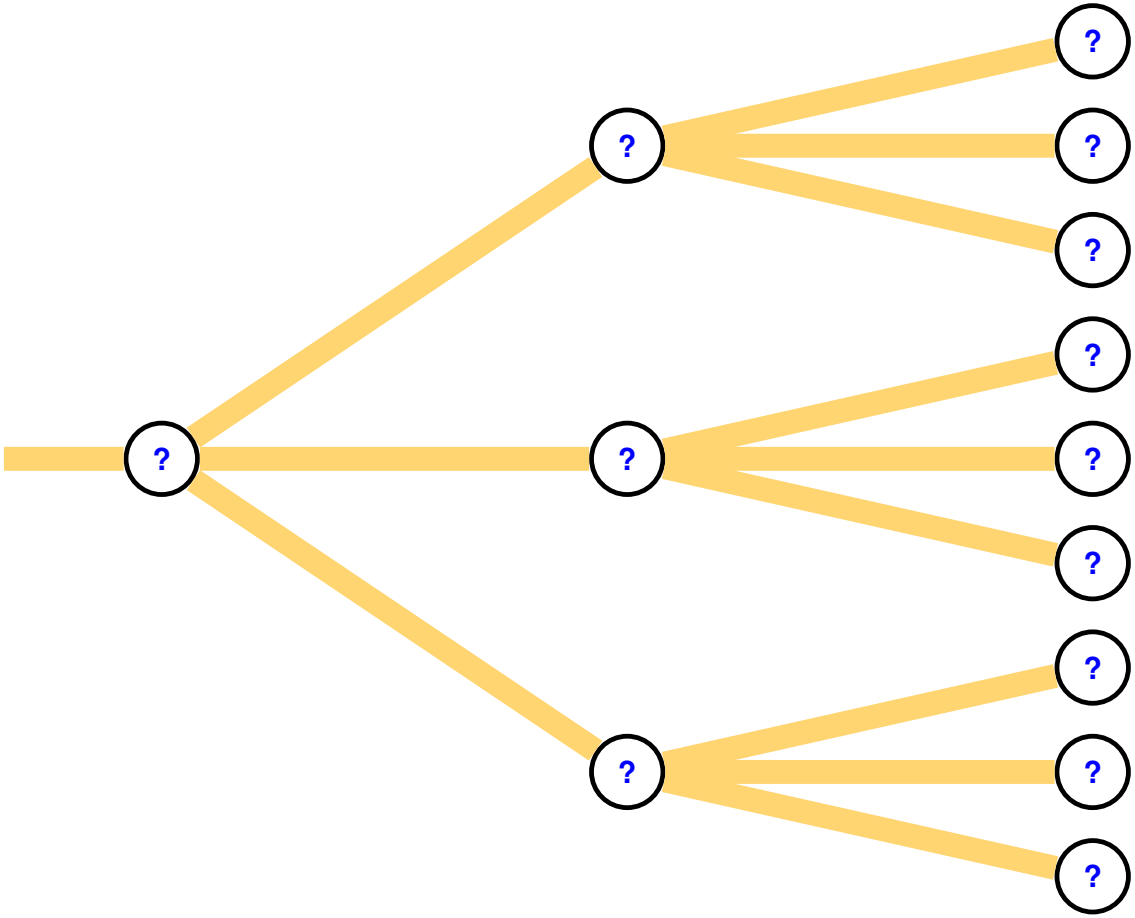
Backtracking: Zusätzliche Anforderungen

Zusätzliche Anforderungen

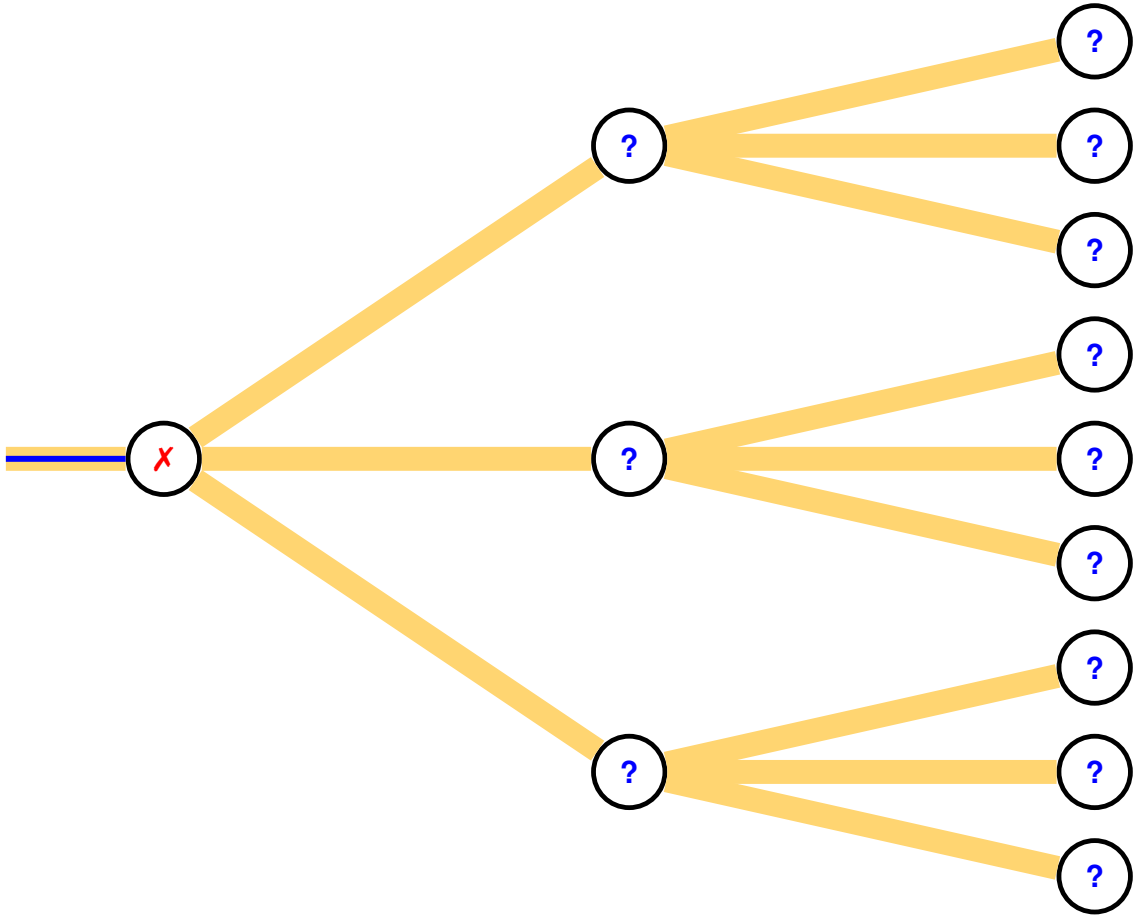
Kann der Backtrackingalgorithmus die zusätzlichen Anforderungen erfüllen?:

- Finden eines Lösungswegs: Zurückgeben einer Reihung, die alle Entscheidungen enthält, die zur Lösung geführt haben
- Lösungsweg soll minimal sein: Nachdem der komplette Suchraum betrachtet wurde, kann der kürzeste unter den gefundenen Lösungswegen zurückgegeben werden

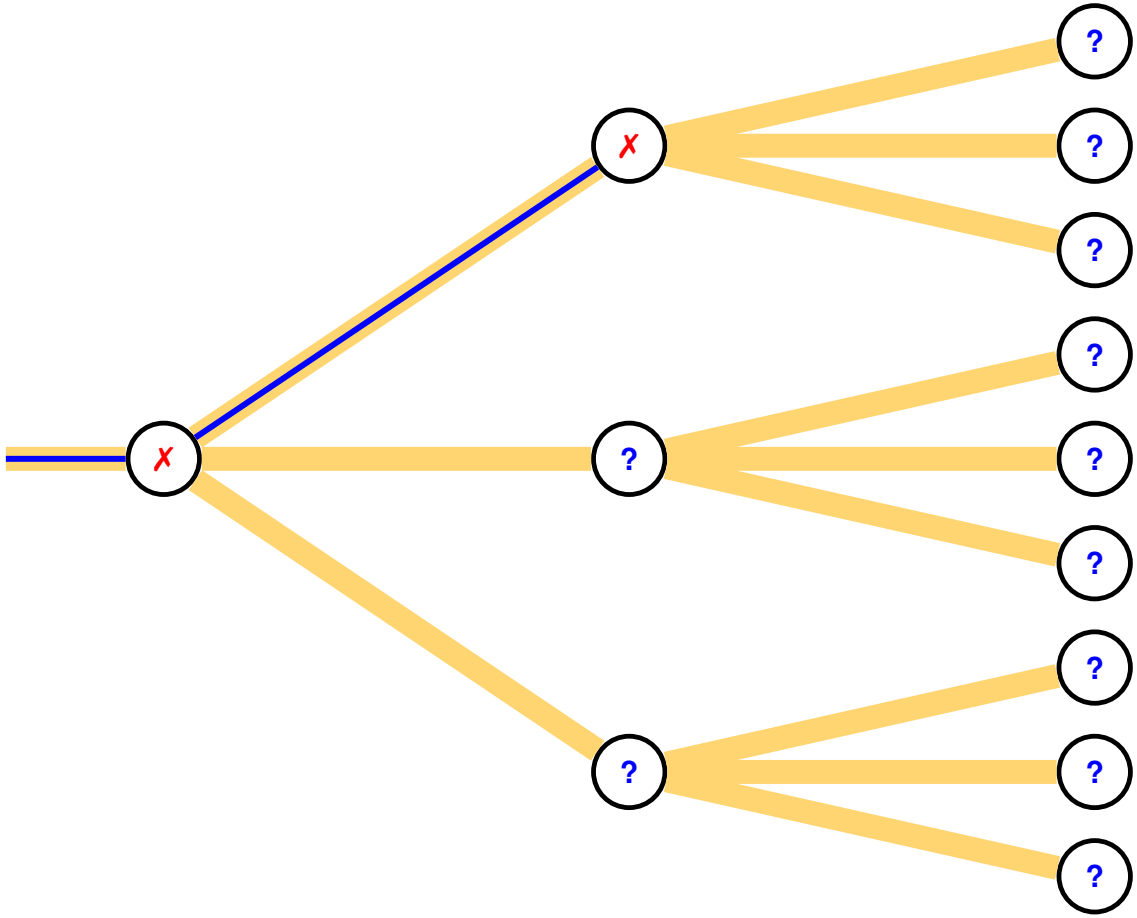
Kürzester Lösungsweg mit Backtracking



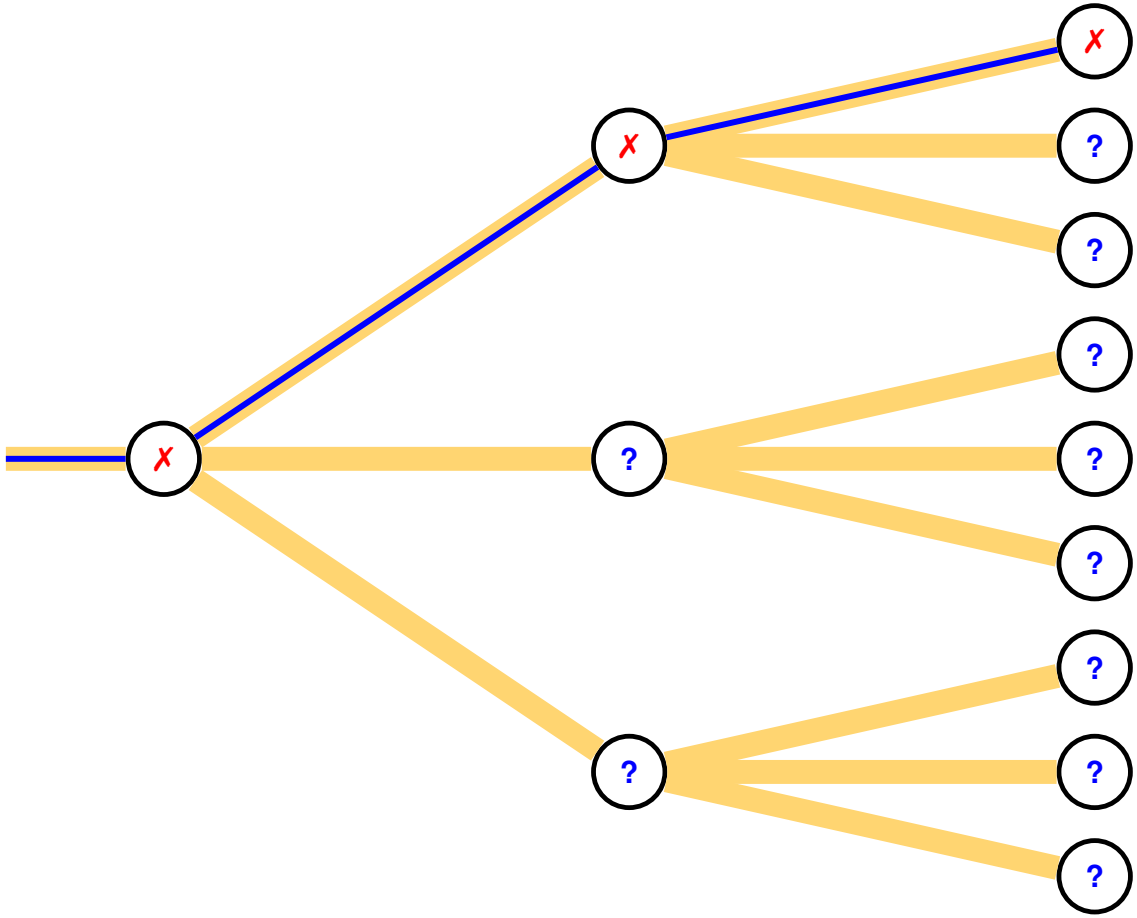
Kürzester Lösungsweg mit Backtracking



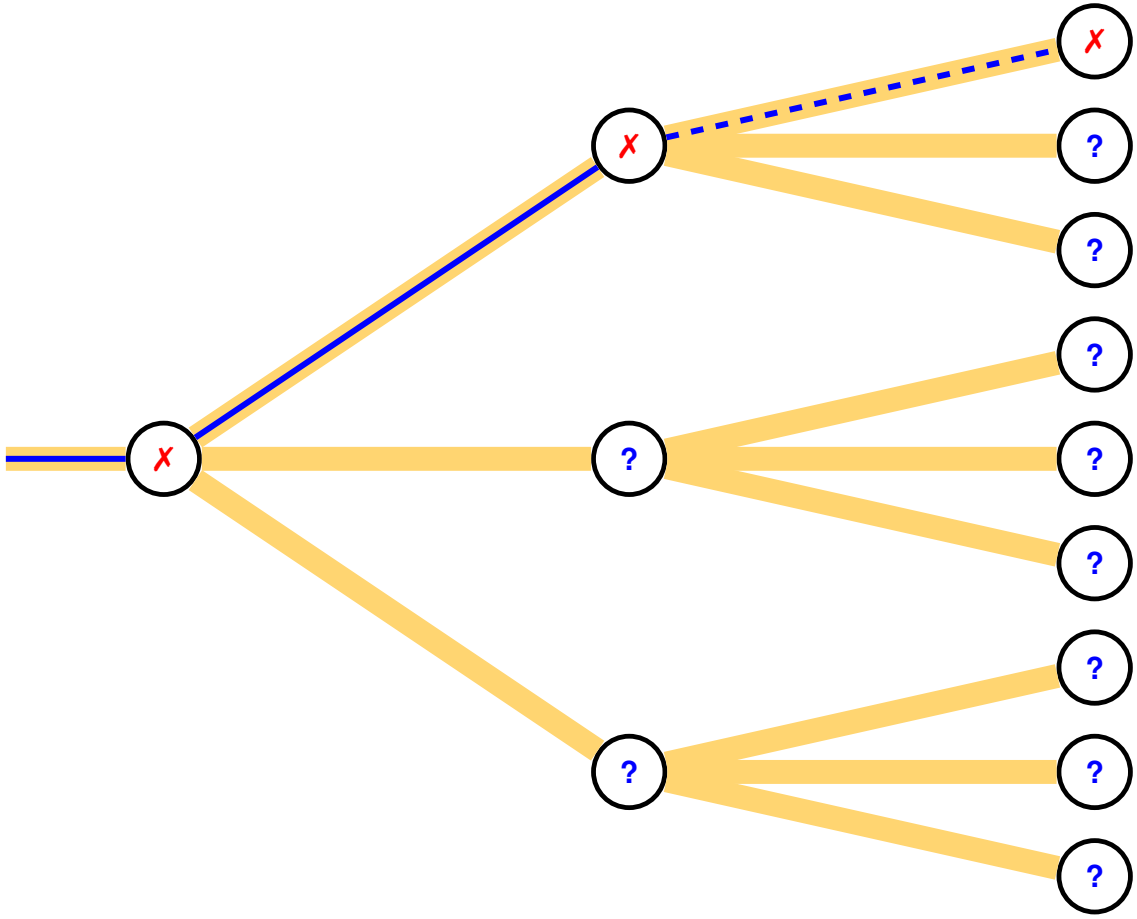
Kürzester Lösungsweg mit Backtracking



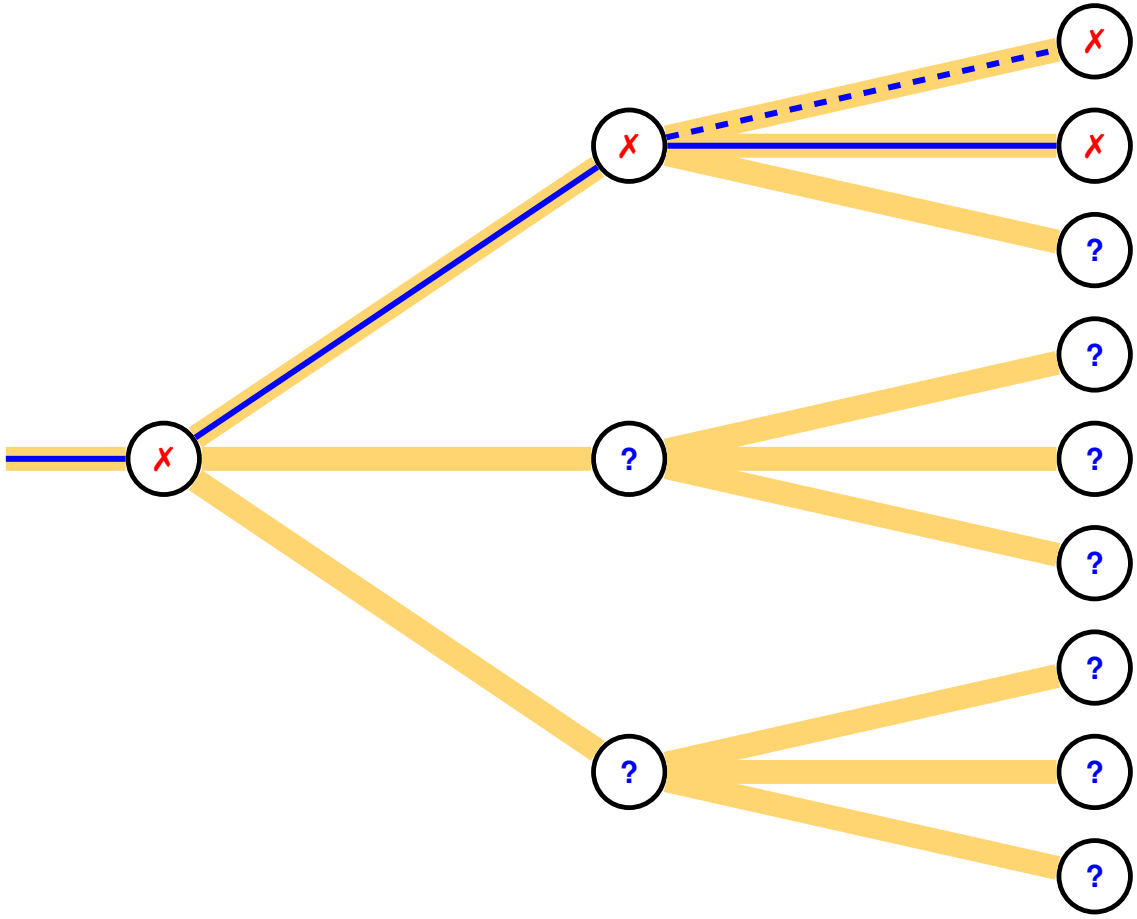
Kürzester Lösungsweg mit Backtracking



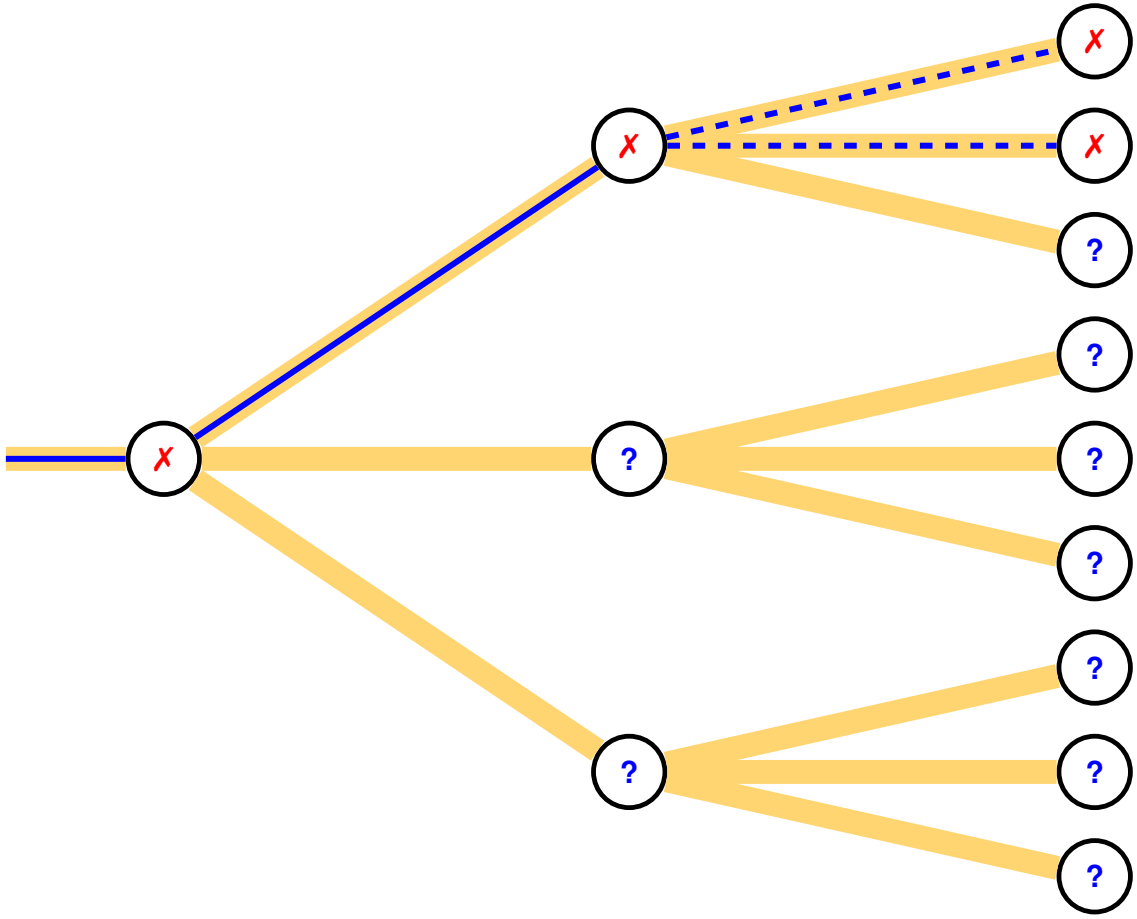
Kürzester Lösungsweg mit Backtracking



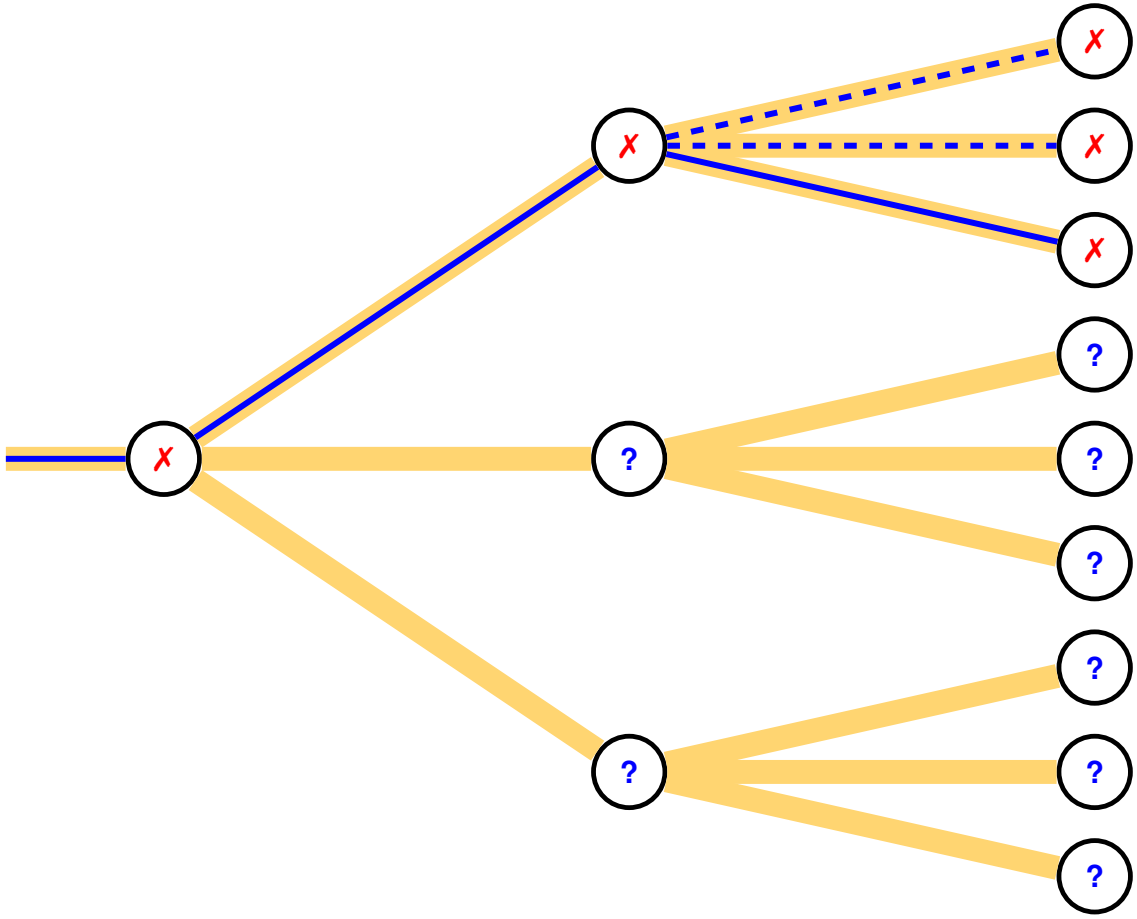
Kürzester Lösungsweg mit Backtracking



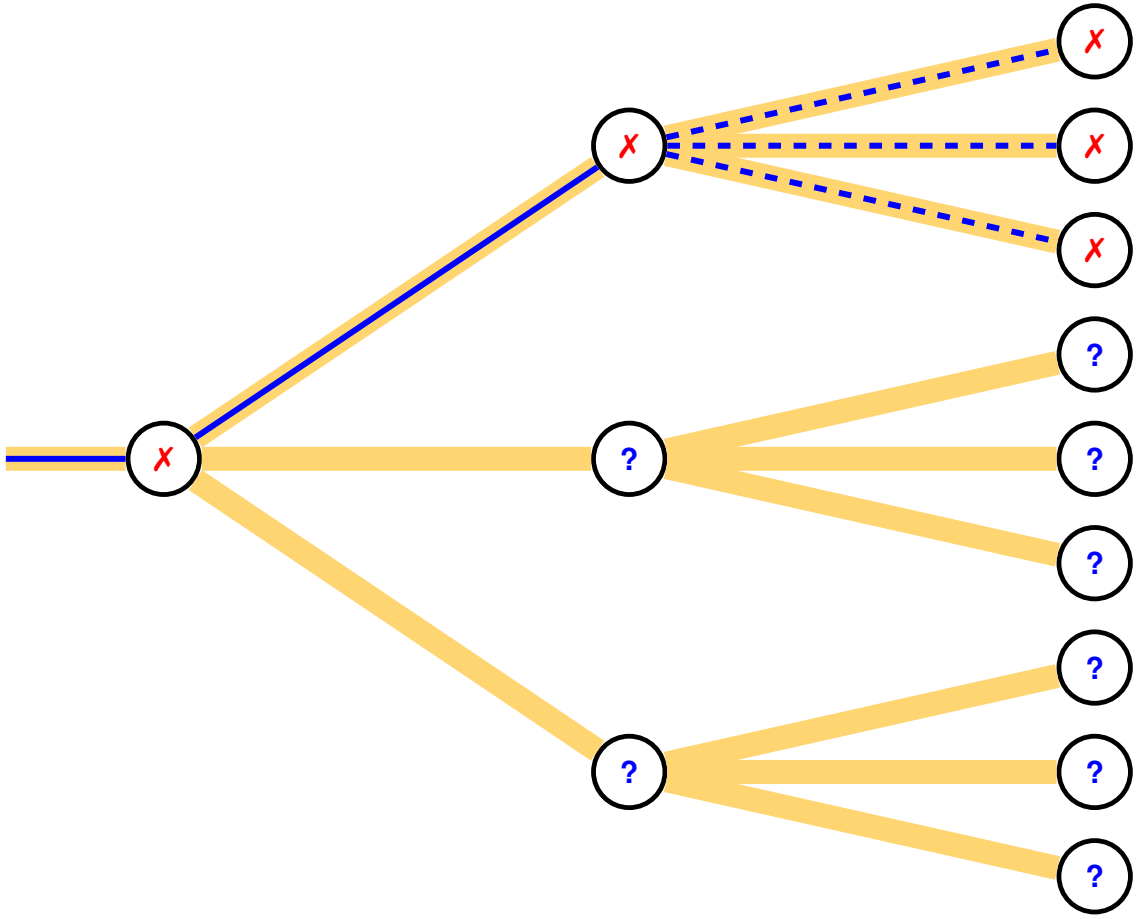
Kürzester Lösungsweg mit Backtracking



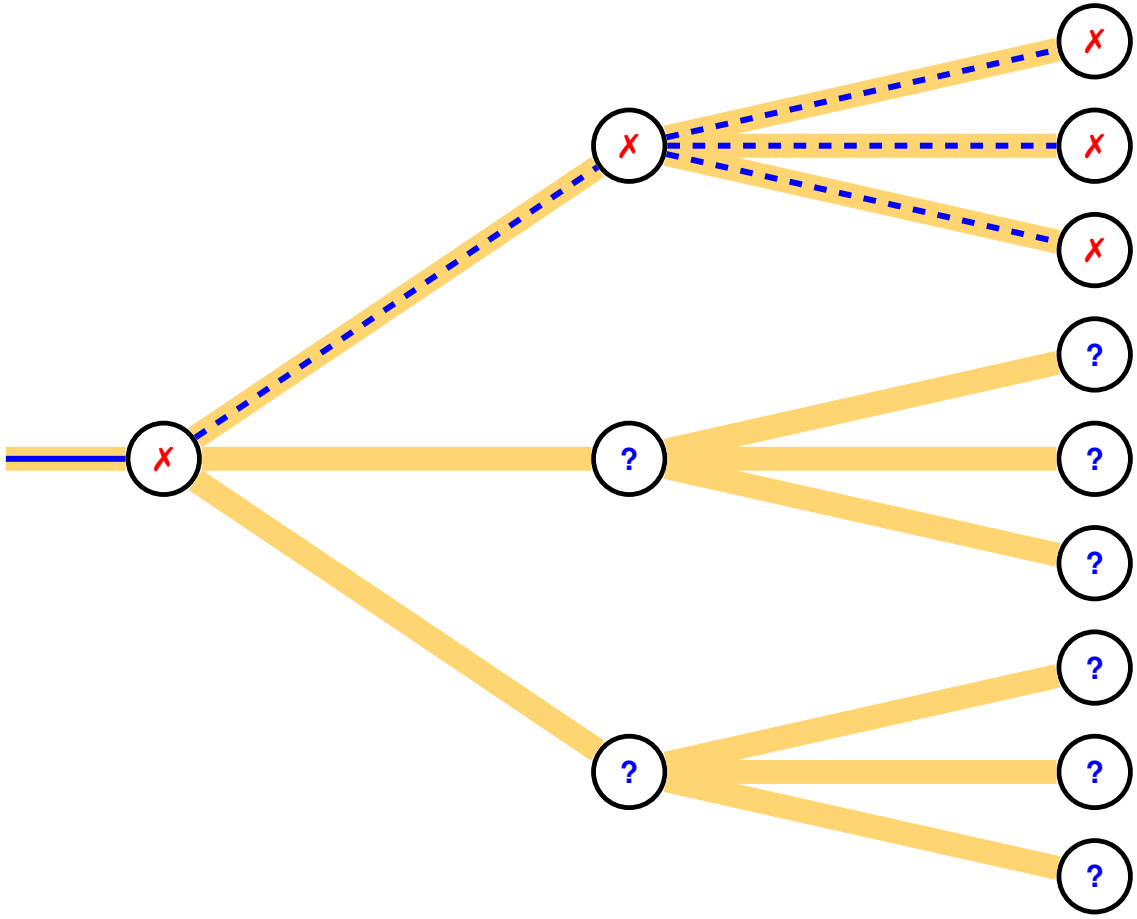
Kürzester Lösungsweg mit Backtracking



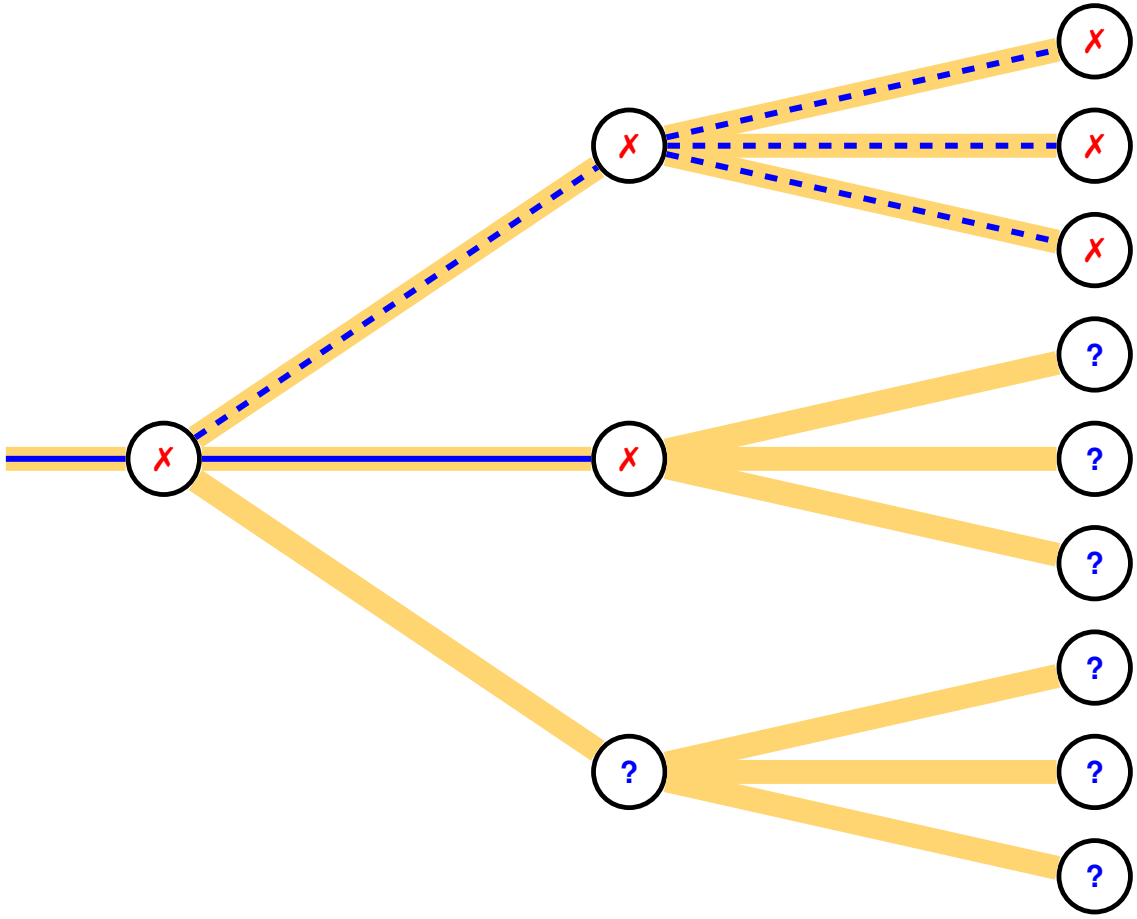
Kürzester Lösungsweg mit Backtracking



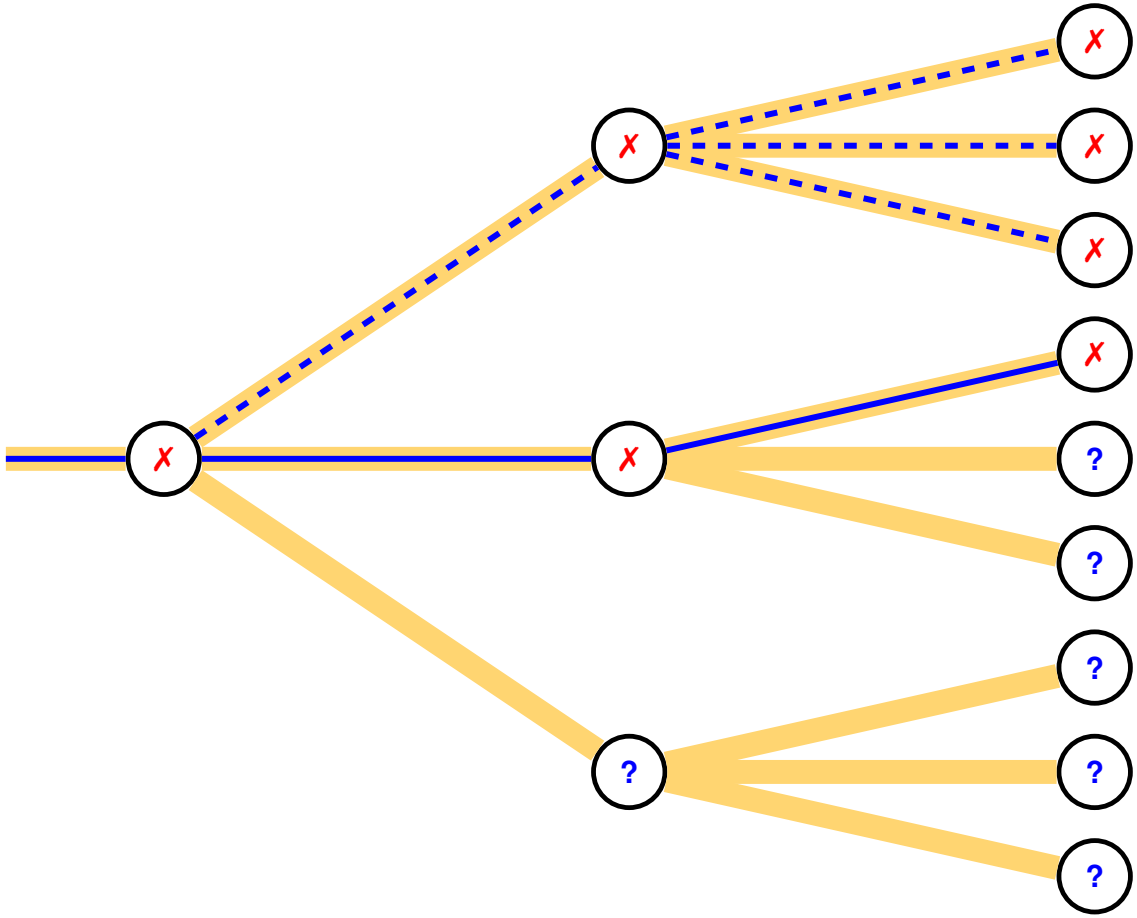
Kürzester Lösungsweg mit Backtracking



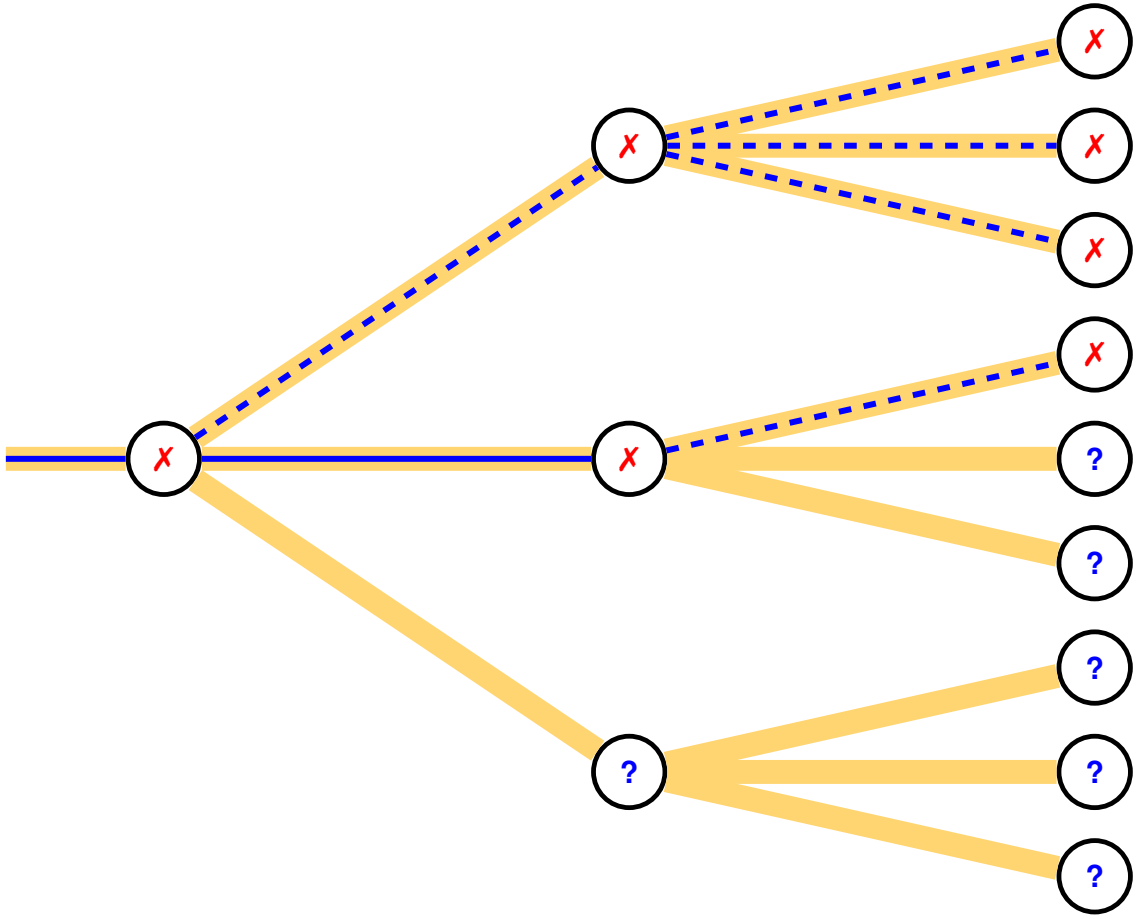
Kürzester Lösungsweg mit Backtracking



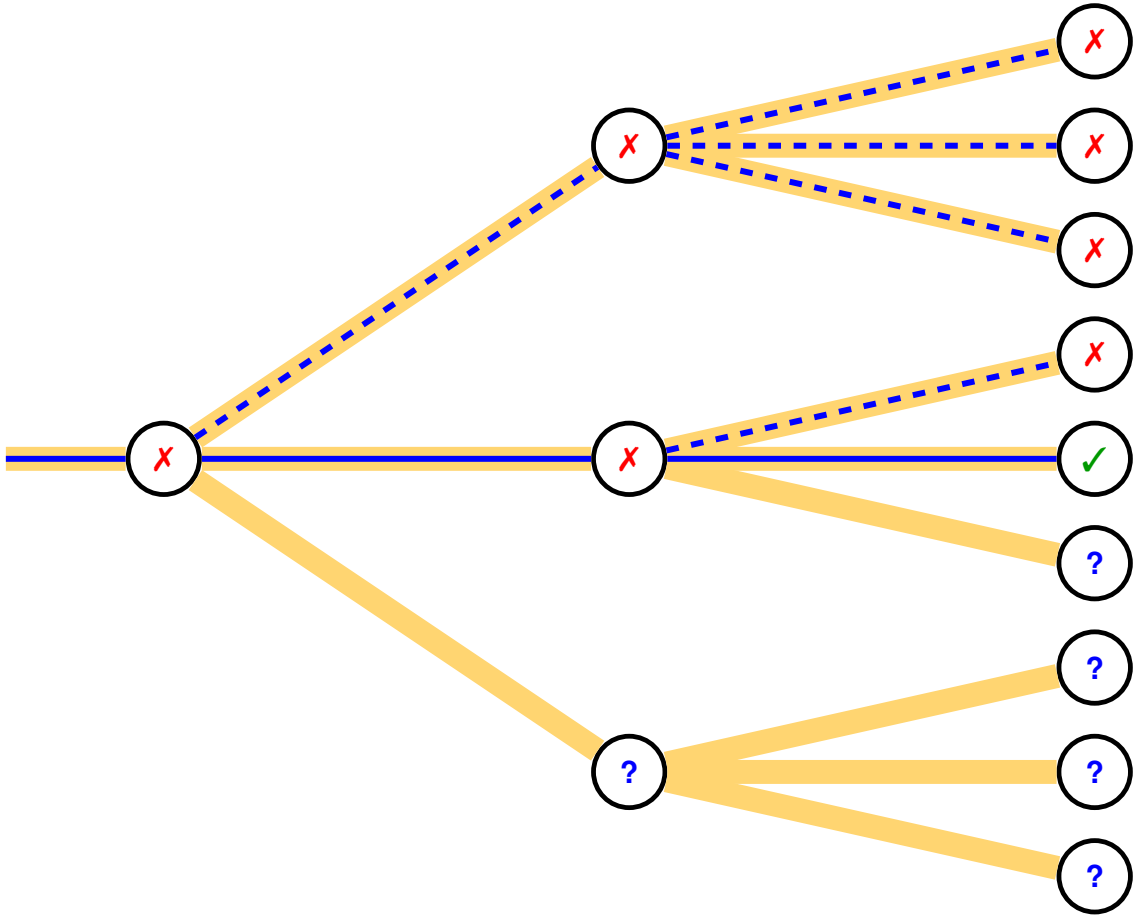
Kürzester Lösungsweg mit Backtracking



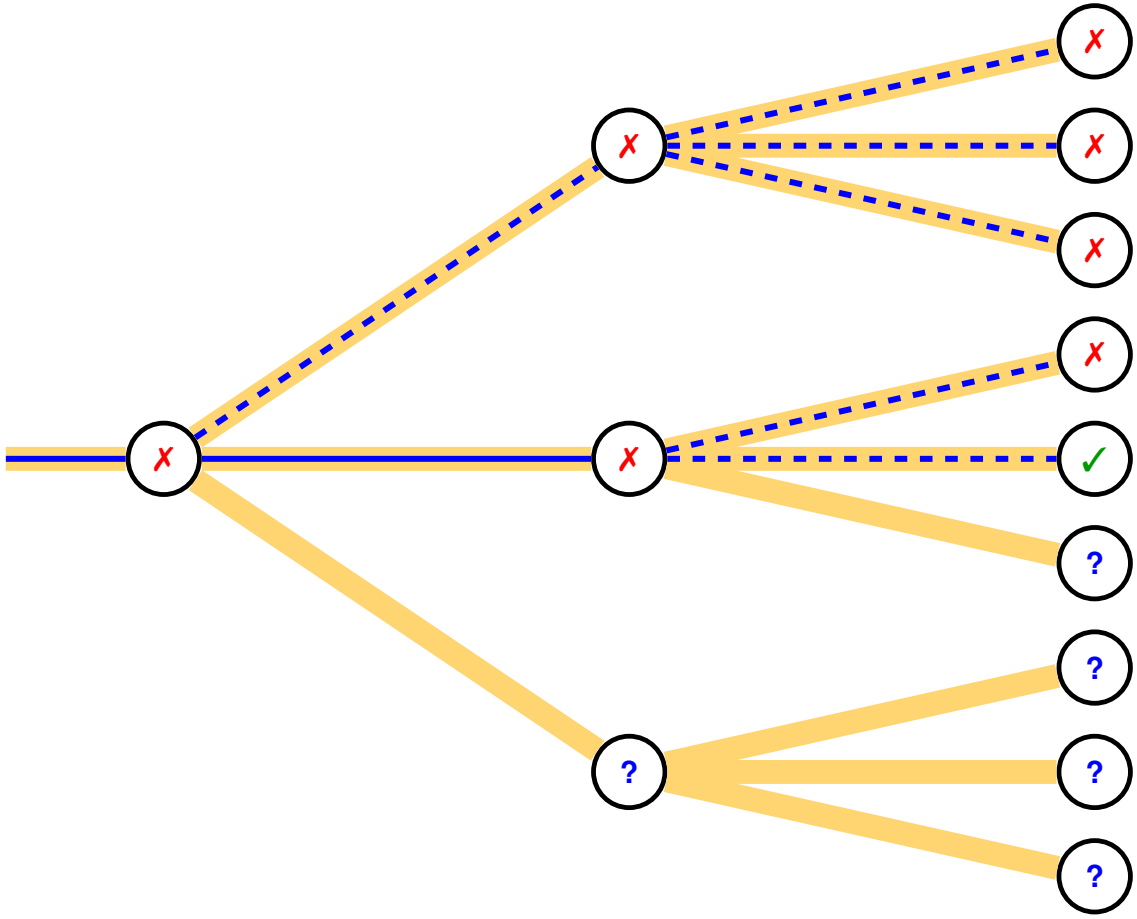
Kürzester Lösungsweg mit Backtracking



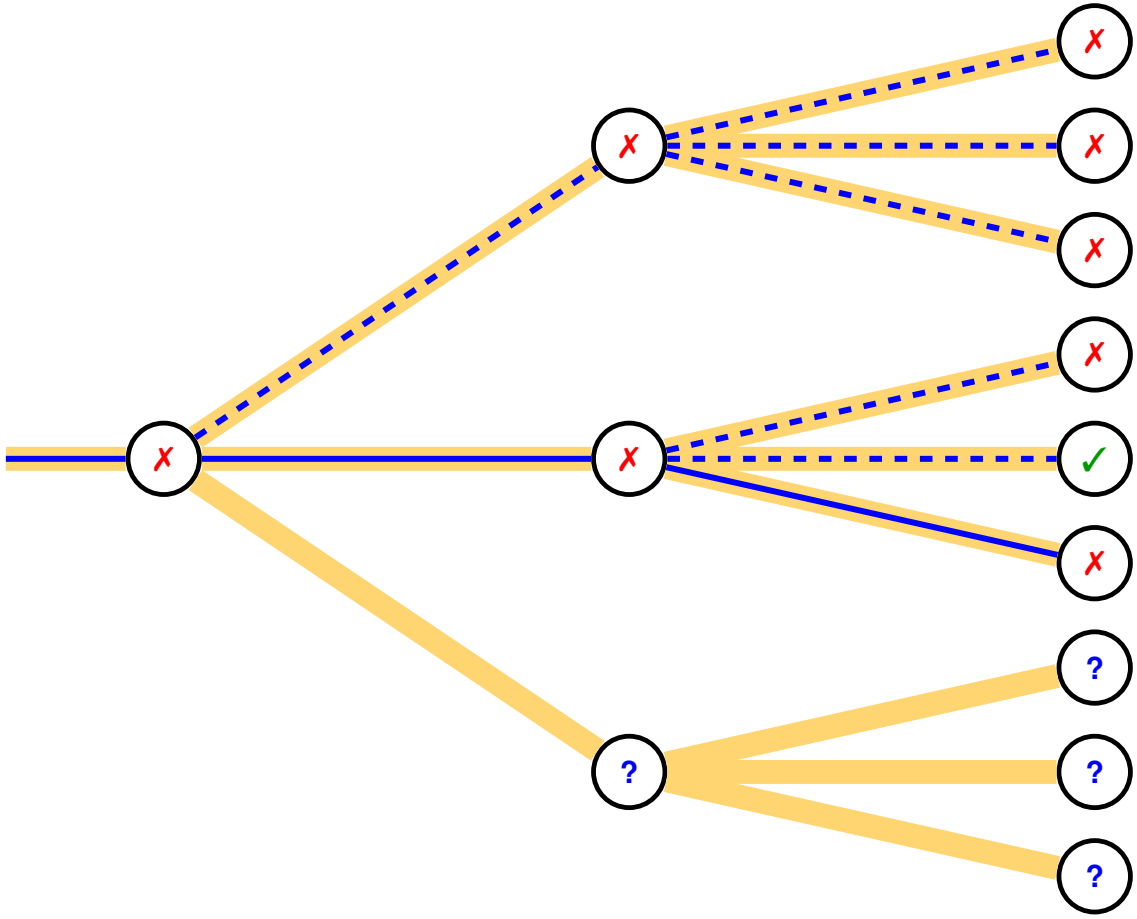
Kürzester Lösungsweg mit Backtracking



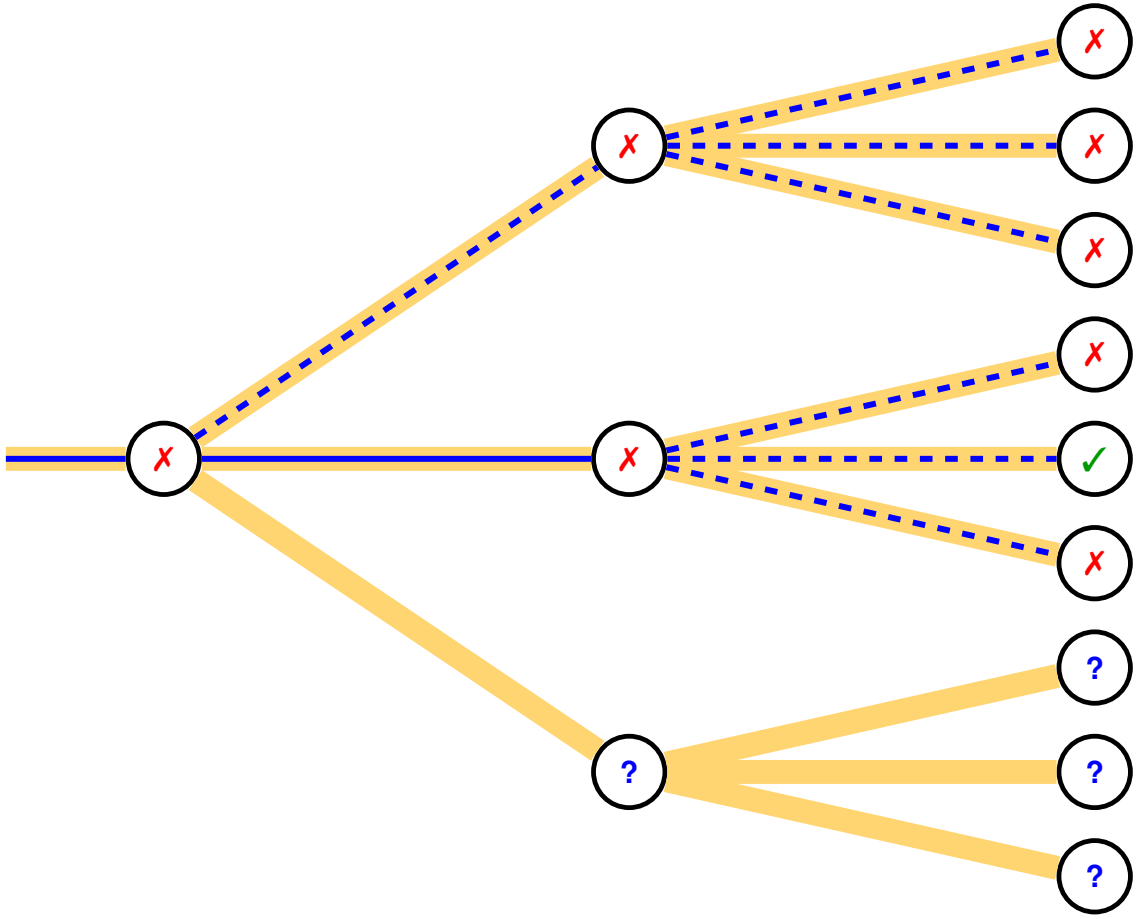
Kürzester Lösungsweg mit Backtracking



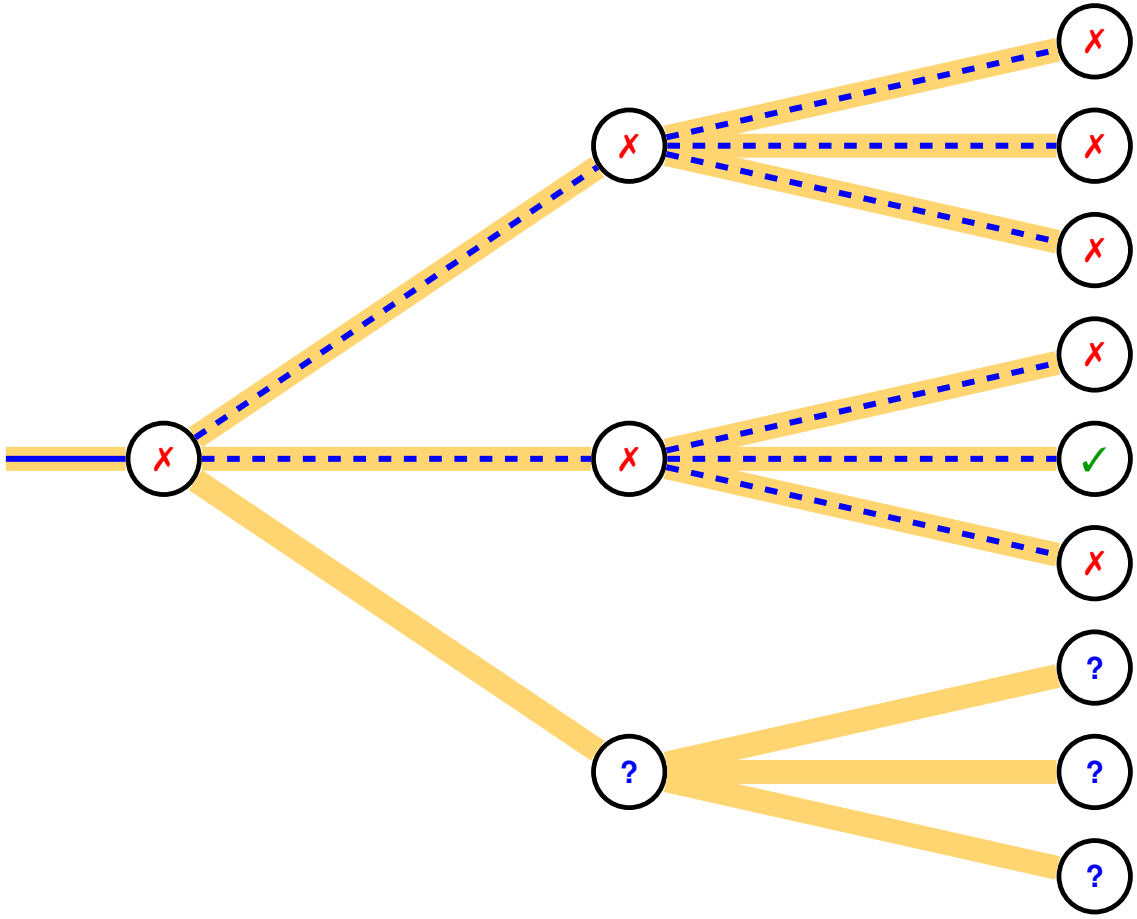
Kürzester Lösungsweg mit Backtracking



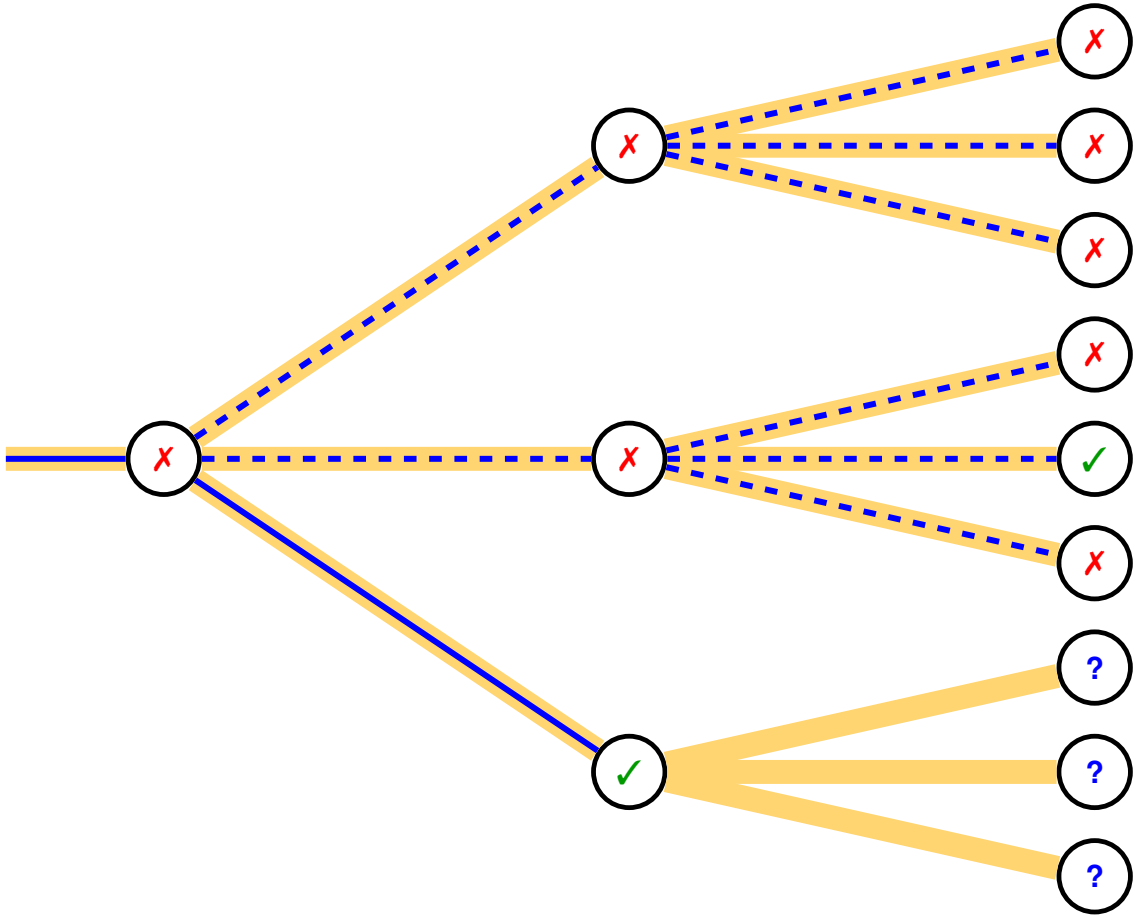
Kürzester Lösungsweg mit Backtracking



Kürzester Lösungsweg mit Backtracking



Kürzester Lösungsweg mit Backtracking



Alternative: Schrittweises Backtracking

- setze $n := 1$
- solange nicht alle Zustände im Suchraum betrachtet wurden:
  - betrachte alle Zustände, die mit einer Rekursionstiefe von maximal n erreichbar sind
    - falls einer der Zustände die Lösung ist: gut ☺
      - Suche kann abgebrochen werden
    - andernfalls:
      - $n := n + 1$ und weiter geht's

$\rightsquigarrow$ schrittweises Backtracking iterativ oft einfacher zu implementieren

$\rightsquigarrow$ die erste Lösung, die gefunden wird, hat garantiert einen minimalen Lösungsweg $\Rightarrow$ es muss nicht der komplette Suchraum betrachtet werden

Alternative: Schrittweises Backtracking

- setze $n := 1$
- solange nicht alle Zustände im Suchraum betrachtet wurden:
  - betrachte alle Zustände, die mit einer Rekursionstiefe von maximal n erreichbar sind
    - falls einer der Zustände die Lösung ist: gut ☺
      - Suche kann abgebrochen werden
    - andernfalls:
      - $n := n + 1$ und weiter geht's

⇝ schrittweises Backtracking iterativ oft einfacher zu implementieren

⇝ die erste Lösung, die gefunden wird, hat garantiert einen minimalen Lösungsweg ⇒ es muss nicht der komplette Suchraum betrachtet werden

Alternative: Schrittweises Backtracking

- setze $n := 1$
- solange nicht alle Zustände im Suchraum betrachtet wurden:
  - betrachte alle Zustände, die mit einer Rekursionstiefe von maximal n erreichbar sind
    - falls einer der Zustände die Lösung ist: gut ☺
      - Suche kann abgebrochen werden
    - andernfalls:
      - $n := n + 1$ und weiter geht's

$\rightsquigarrow$ schrittweises Backtracking iterativ oft einfacher zu implementieren

$\rightsquigarrow$ die erste Lösung, die gefunden wird, hat garantiert einen minimalen Lösungsweg $\Rightarrow$ es muss nicht der komplette Suchraum betrachtet werden

Alternative: Schrittweises Backtracking

- setze $n := 1$
- solange nicht alle Zustände im Suchraum betrachtet wurden:
  - betrachte alle Zustände, die mit einer Rekursionstiefe von maximal n erreichbar sind
    - falls einer der Zustände die Lösung ist: gut ☺
      - Suche kann abgebrochen werden
    - andernfalls:
      - $n := n + 1$ und weiter geht's

$\rightsquigarrow$ schrittweises Backtracking iterativ oft einfacher zu implementieren

$\rightsquigarrow$ die erste Lösung, die gefunden wird, hat garantiert einen minimalen Lösungsweg $\Rightarrow$ es muss nicht der komplette Suchraum betrachtet werden

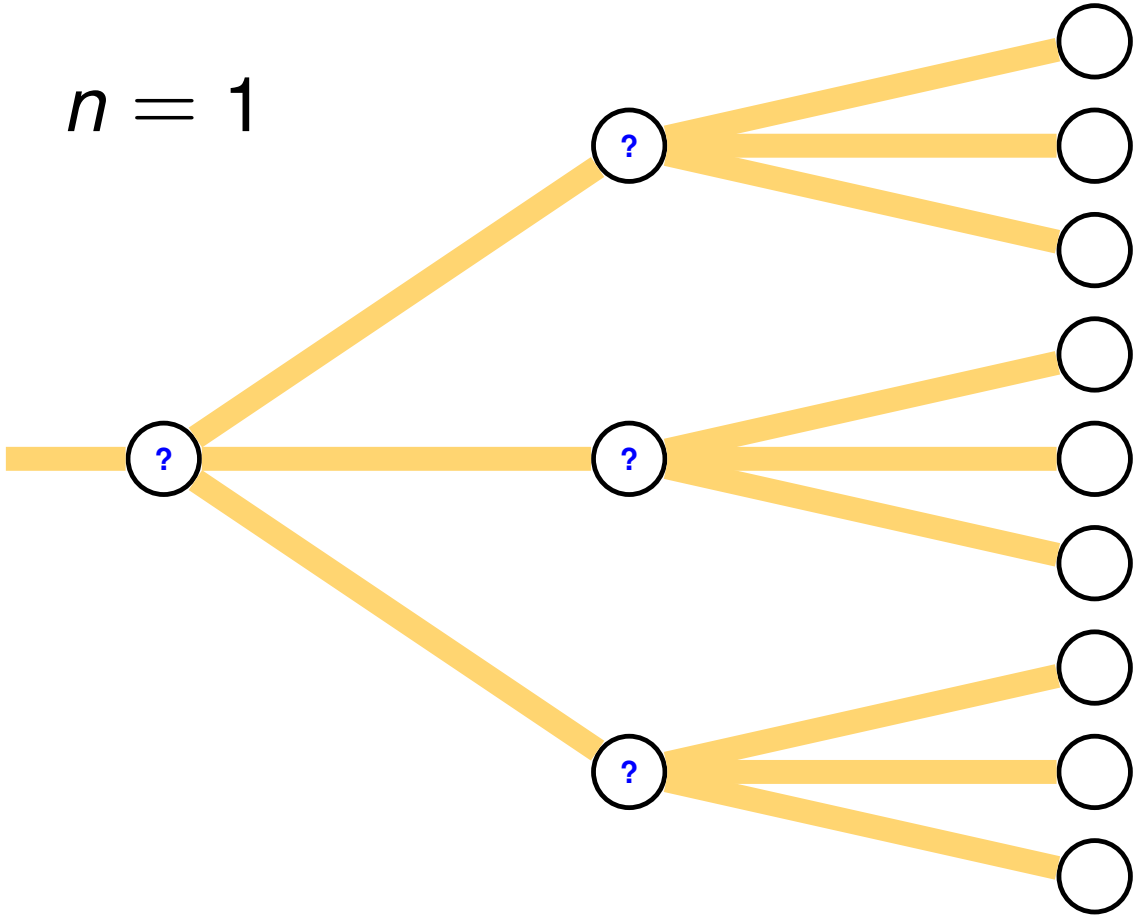
Alternative: Schrittweises Backtracking

- setze $n := 1$
- solange nicht alle Zustände im Suchraum betrachtet wurden:
  - betrachte alle Zustände, die mit einer Rekursionstiefe von maximal n erreichbar sind
    - falls einer der Zustände die Lösung ist: gut ☺
      - Suche kann abgebrochen werden
    - andernfalls:
      - $n := n + 1$ und weiter geht's

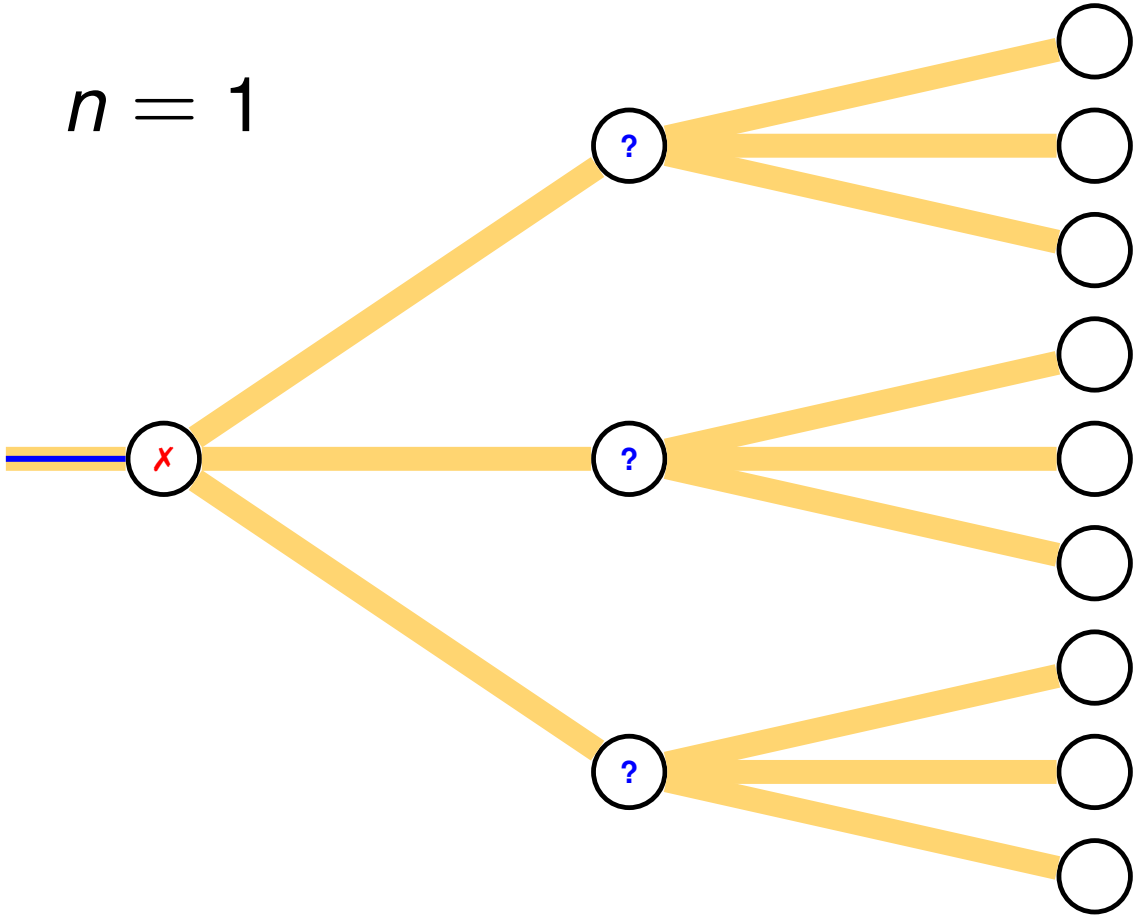
$\rightsquigarrow$ schrittweises Backtracking iterativ oft einfacher zu implementieren

$\rightsquigarrow$ die erste Lösung, die gefunden wird, hat garantiert einen minimalen Lösungsweg $\Rightarrow$ es muss nicht der komplette Suchraum betrachtet werden

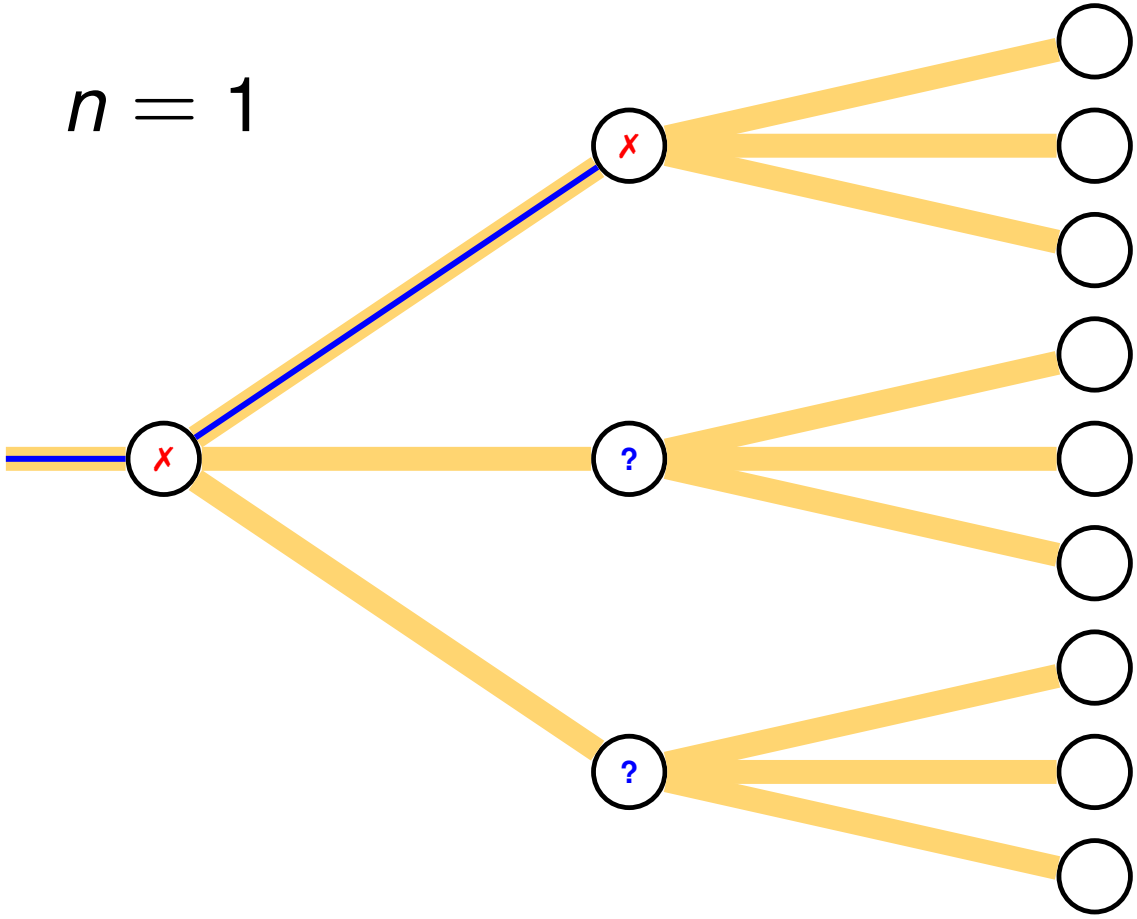
Kürzester Lösungsweg mit schrittweisem Backtracking



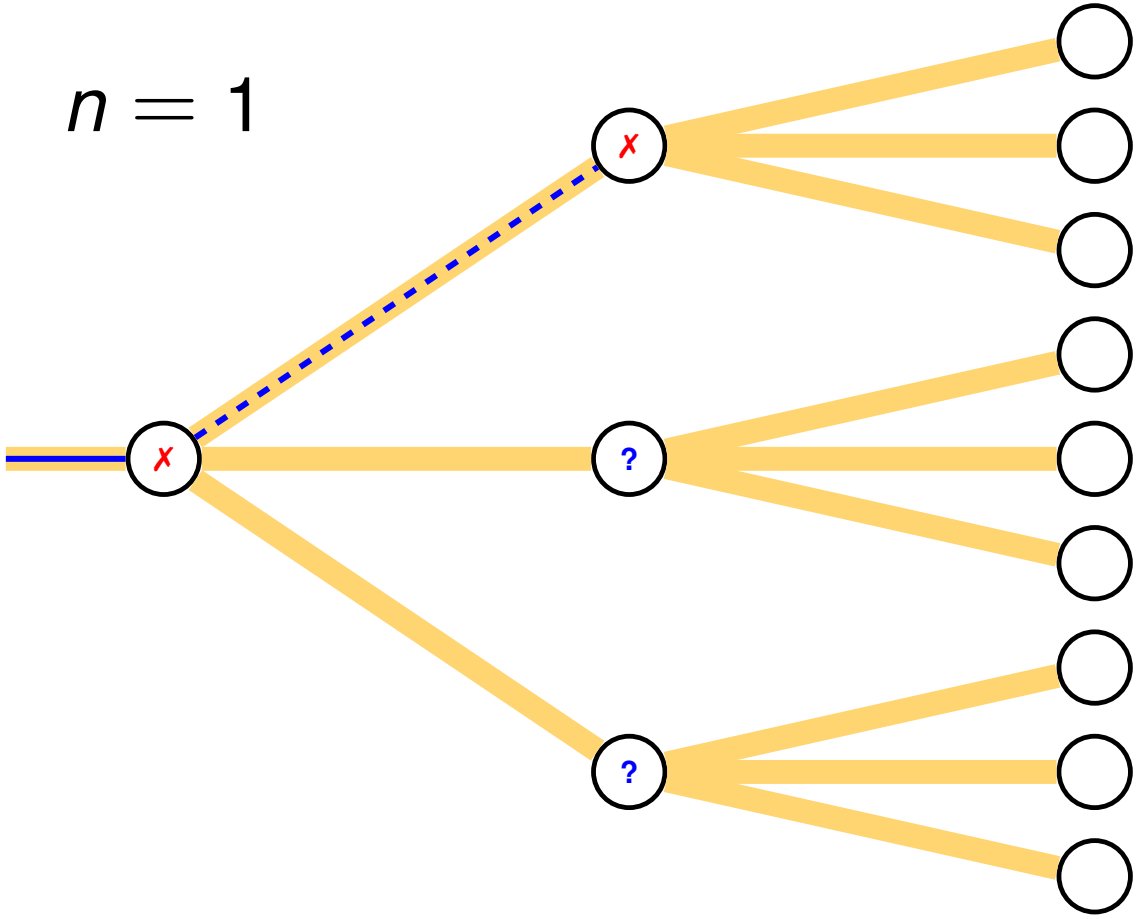
Kürzester Lösungsweg mit schrittweisem Backtracking



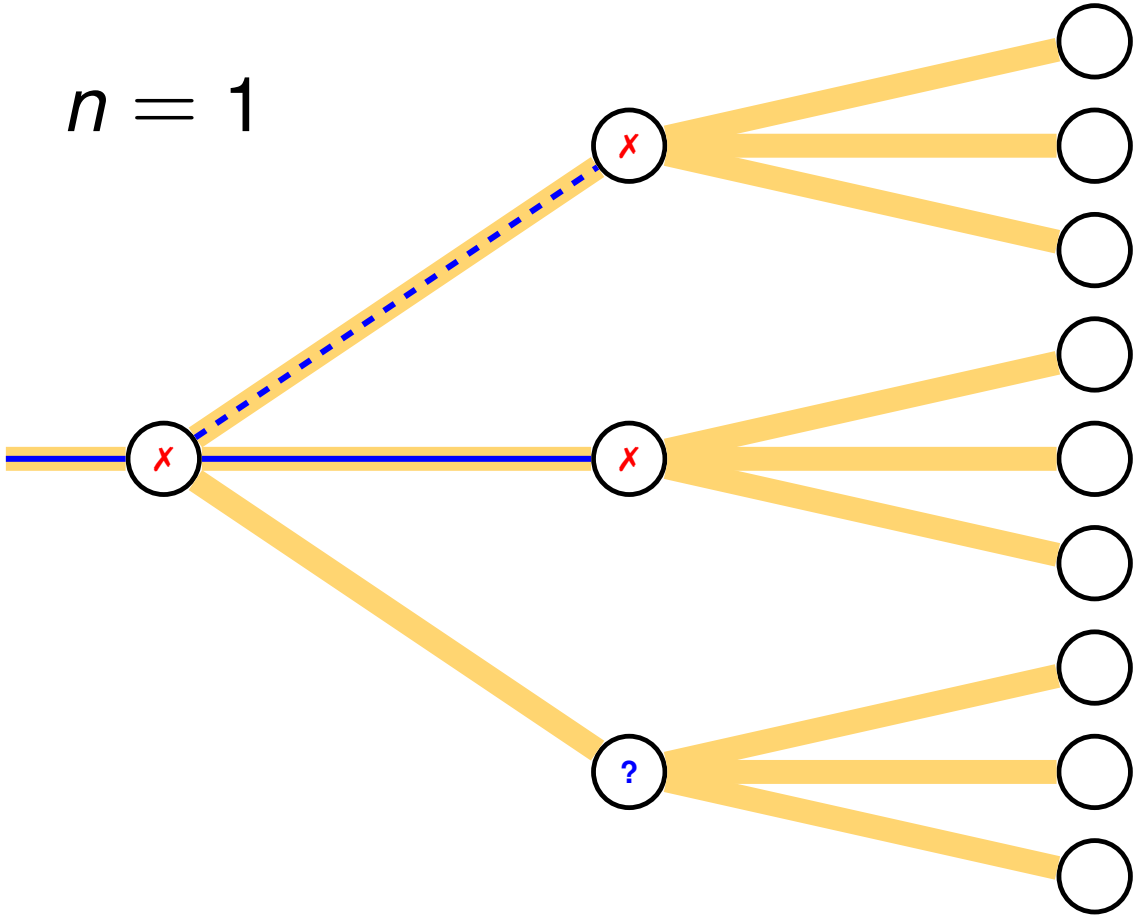
Kürzester Lösungsweg mit schrittweisem Backtracking



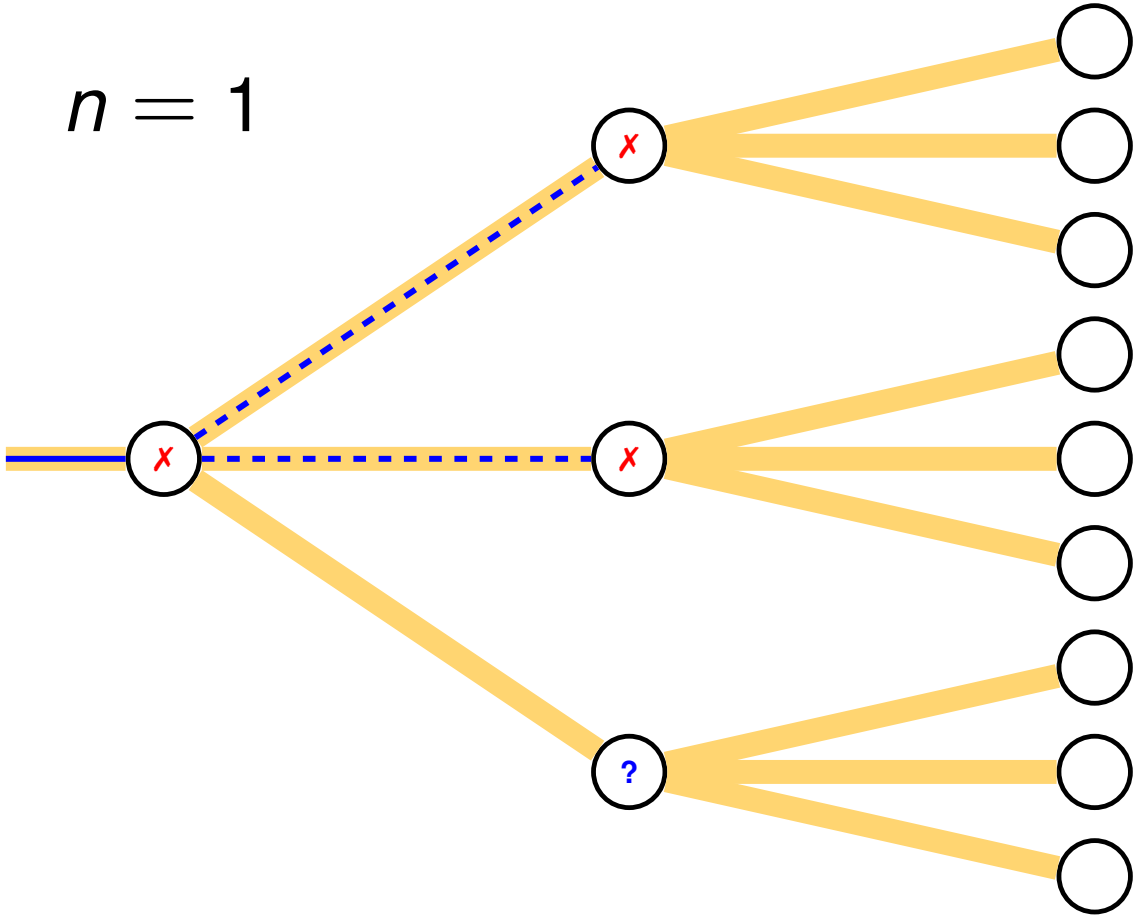
Kürzester Lösungsweg mit schrittweisem Backtracking



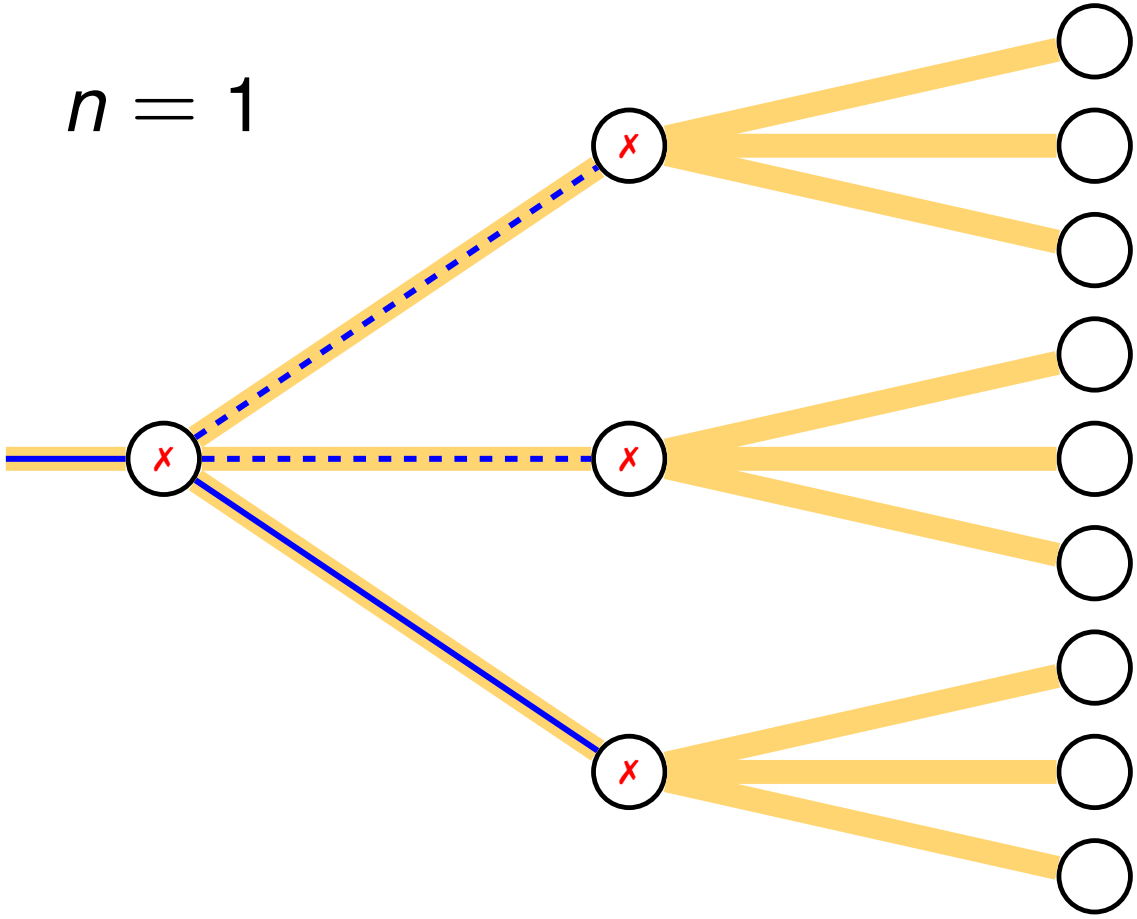
Kürzester Lösungsweg mit schrittweisem Backtracking



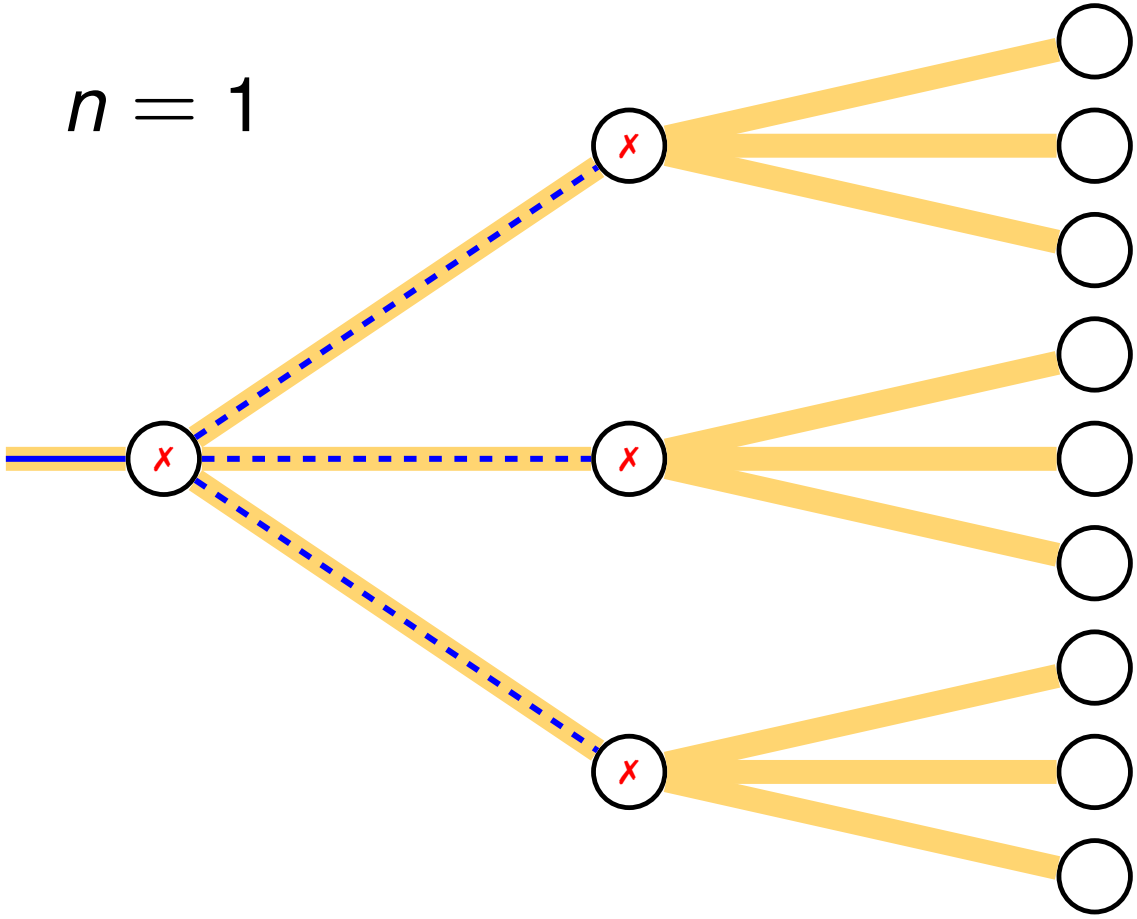
Kürzester Lösungsweg mit schrittweisem Backtracking



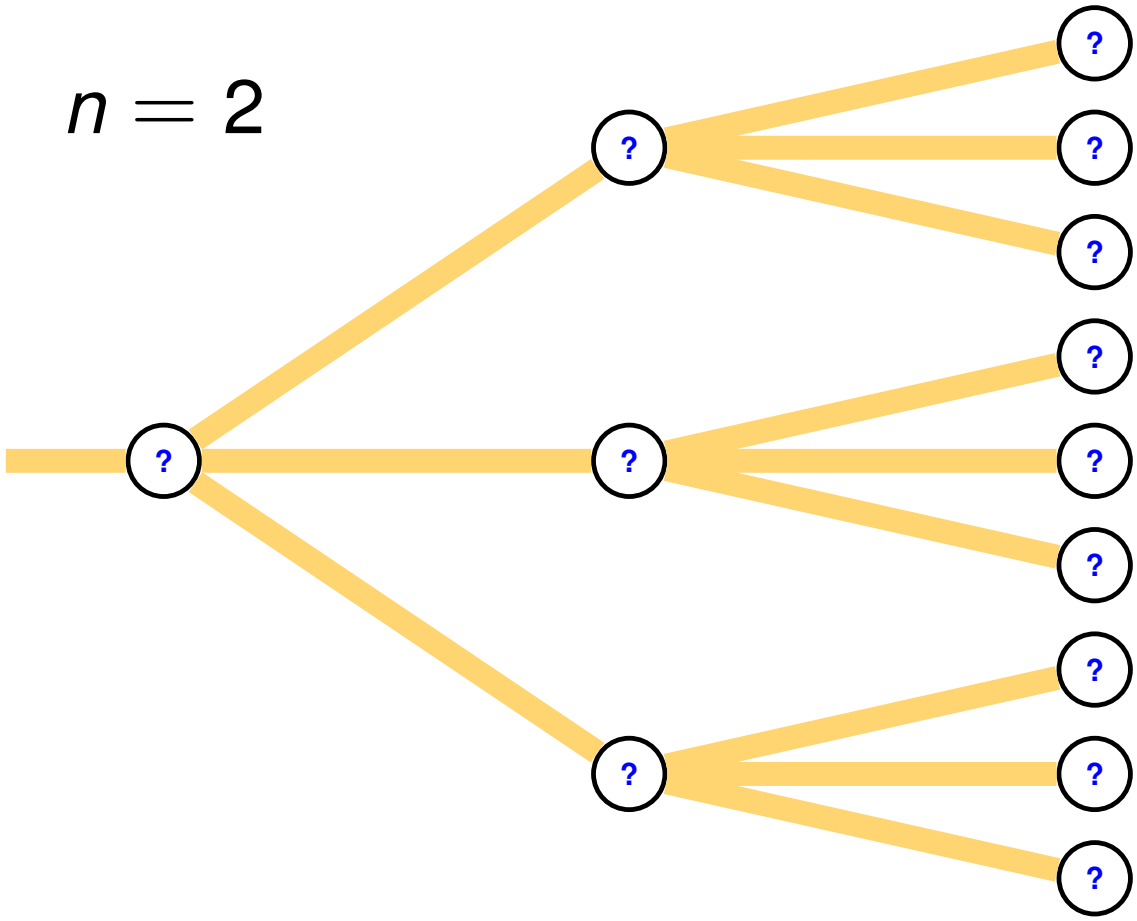
Kürzester Lösungsweg mit schrittweisem Backtracking



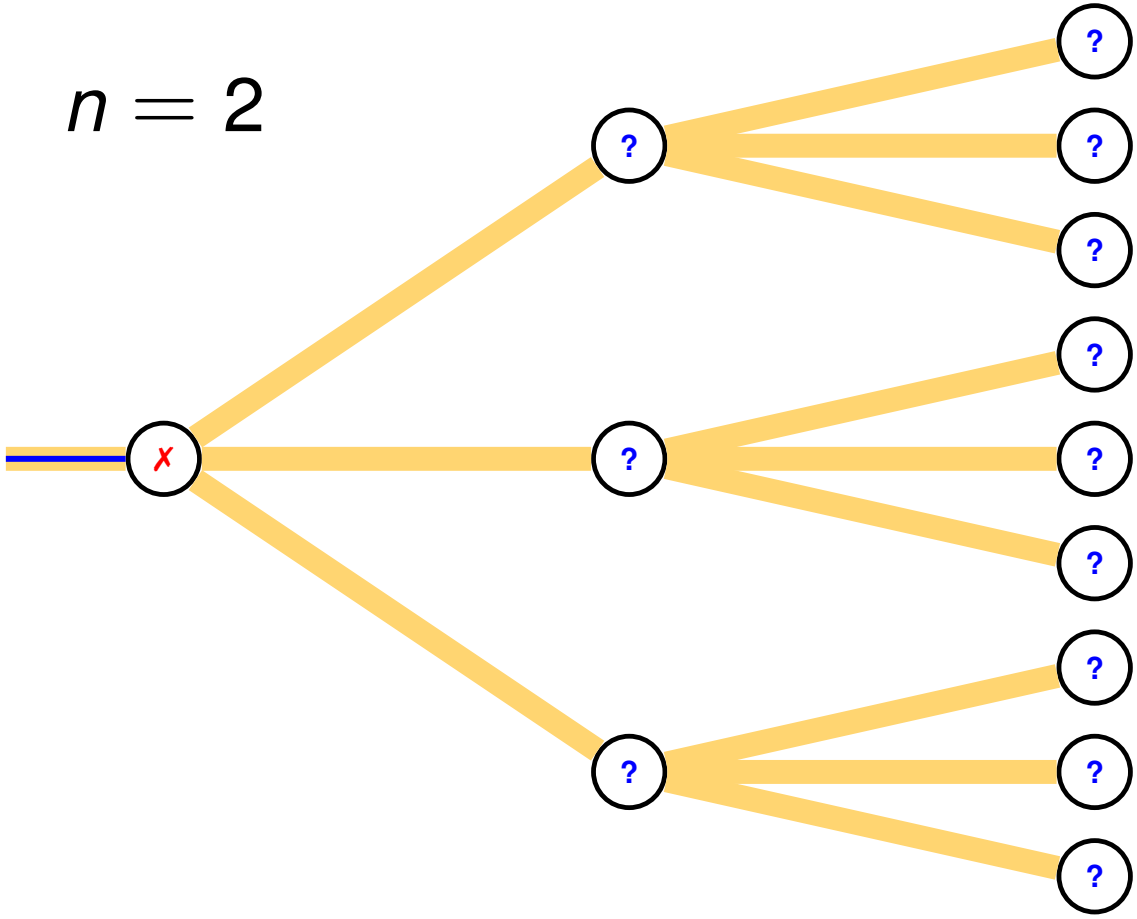
Kürzester Lösungsweg mit schrittweisem Backtracking



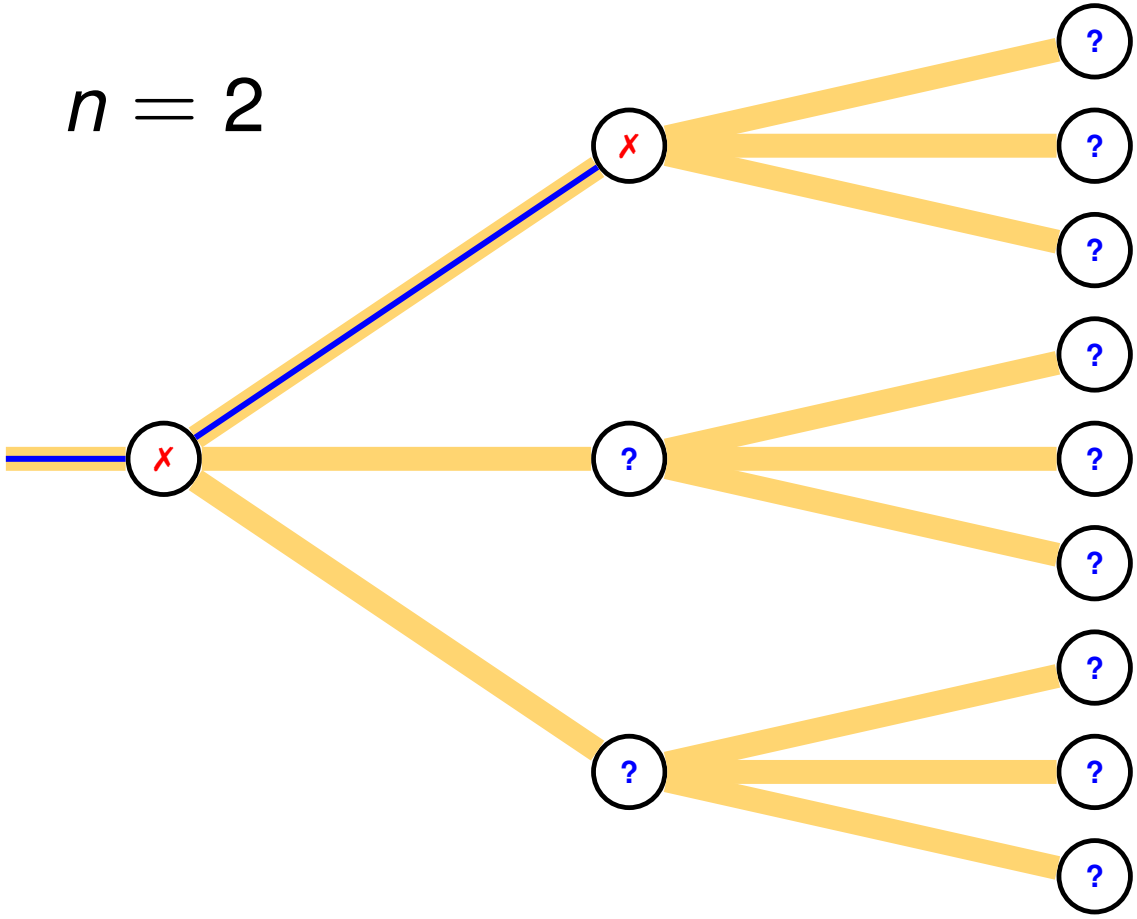
Kürzester Lösungsweg mit schrittweisem Backtracking



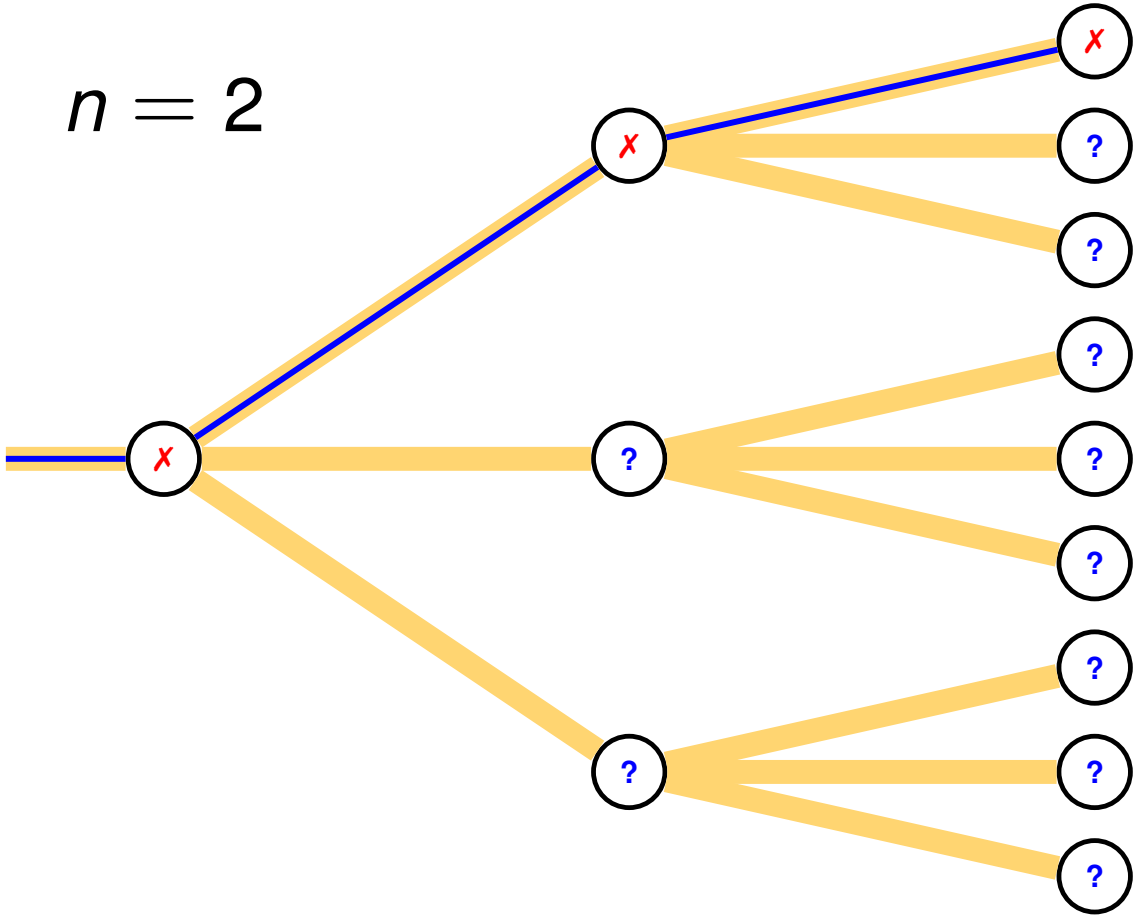
Kürzester Lösungsweg mit schrittweisem Backtracking



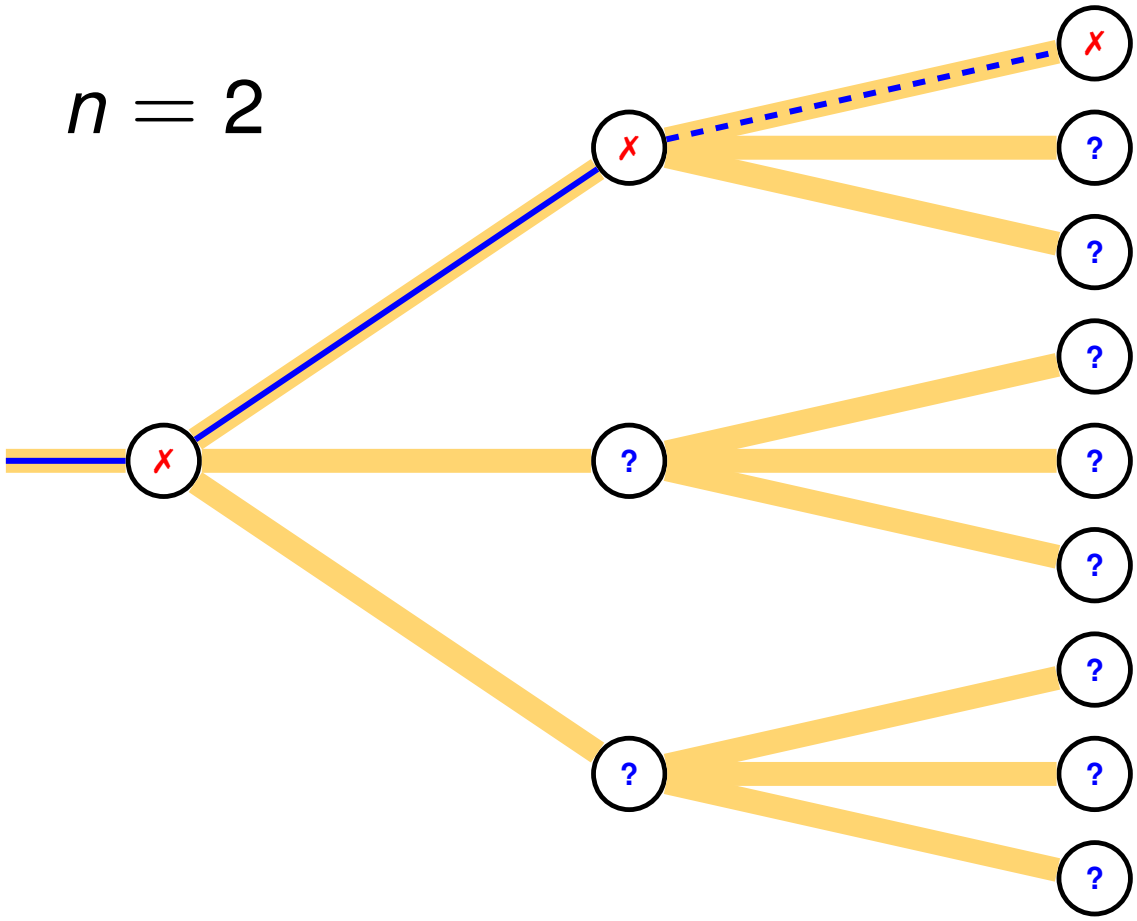
Kürzester Lösungsweg mit schrittweisem Backtracking



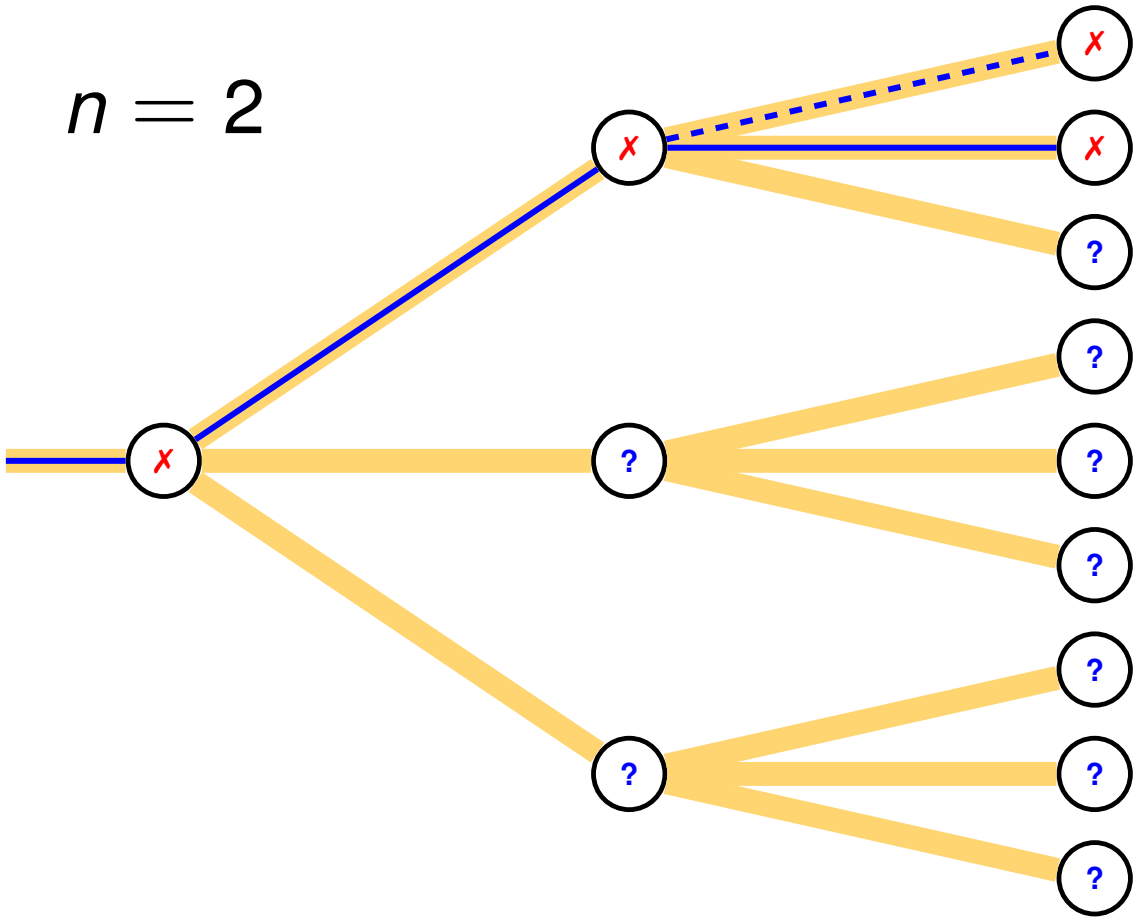
Kürzester Lösungsweg mit schrittweisem Backtracking



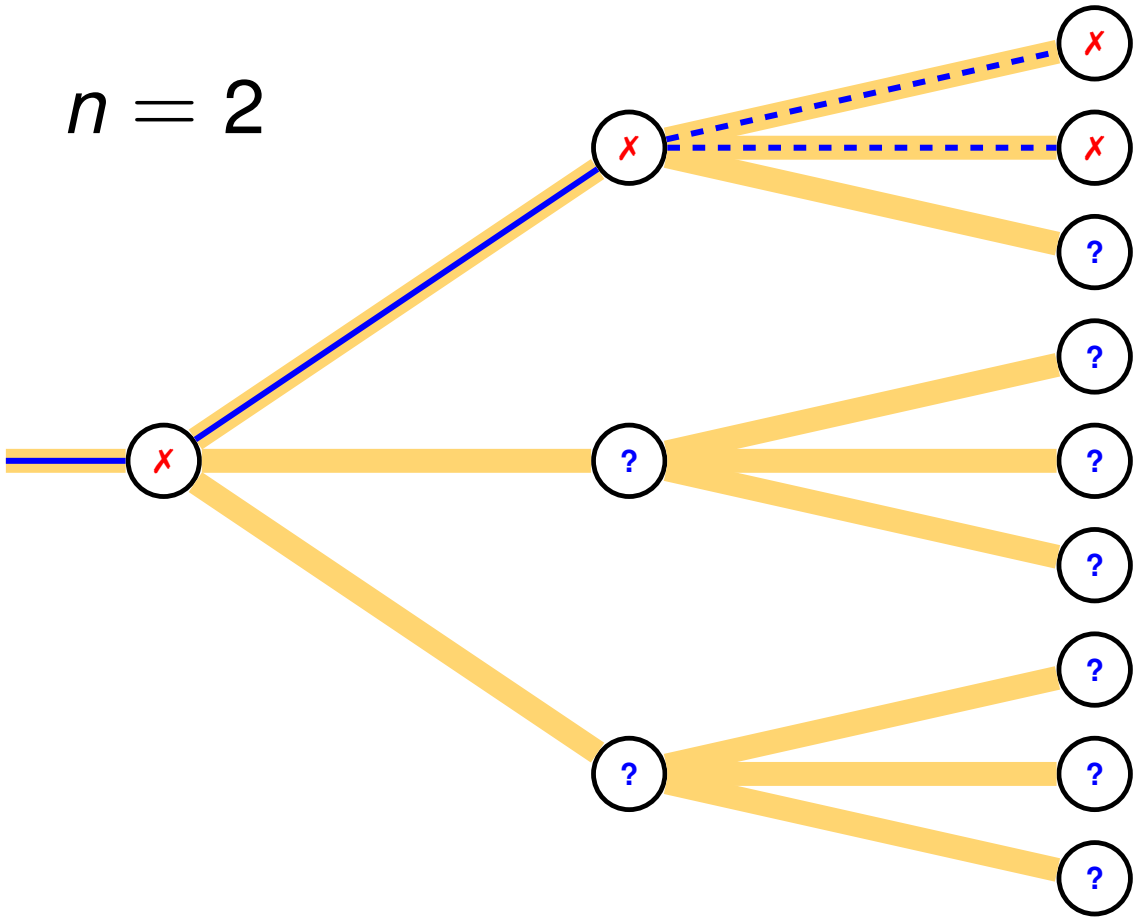
Kürzester Lösungsweg mit schrittweisem Backtracking



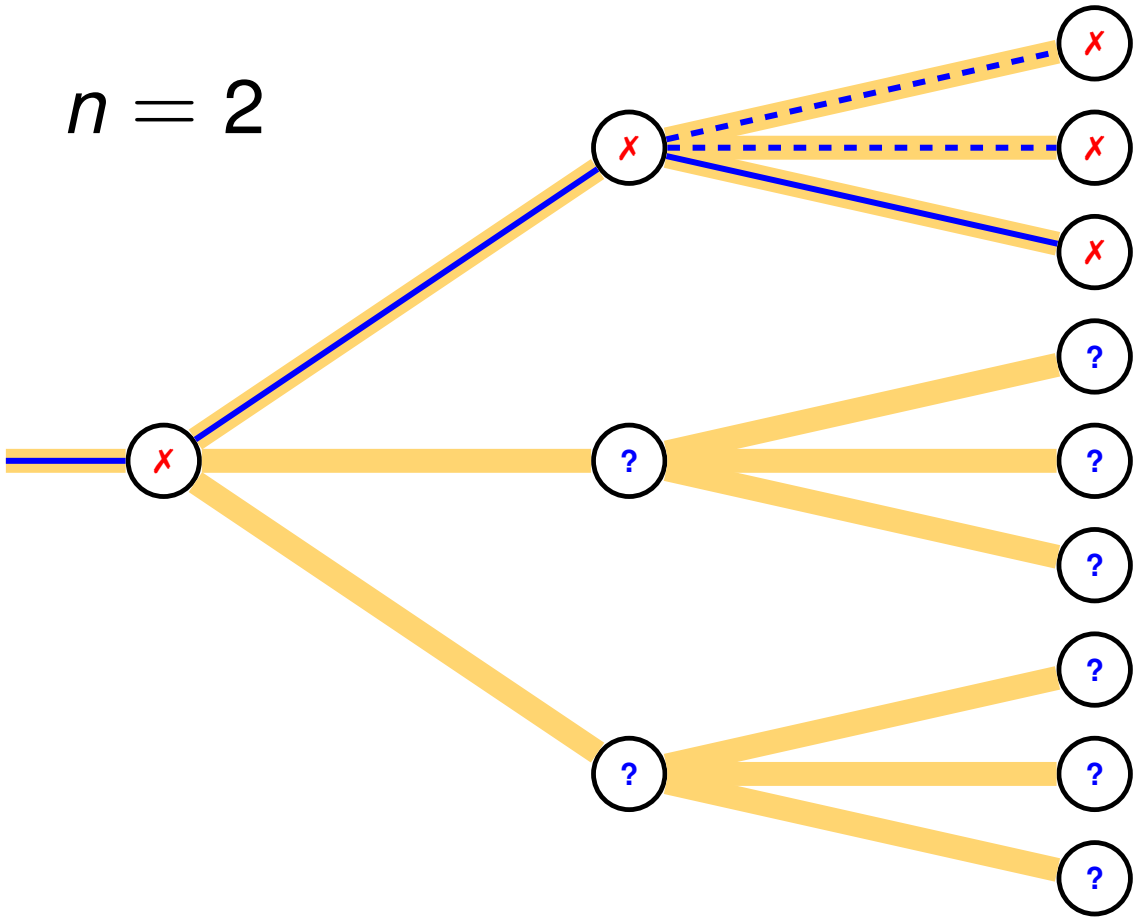
Kürzester Lösungsweg mit schrittweisem Backtracking



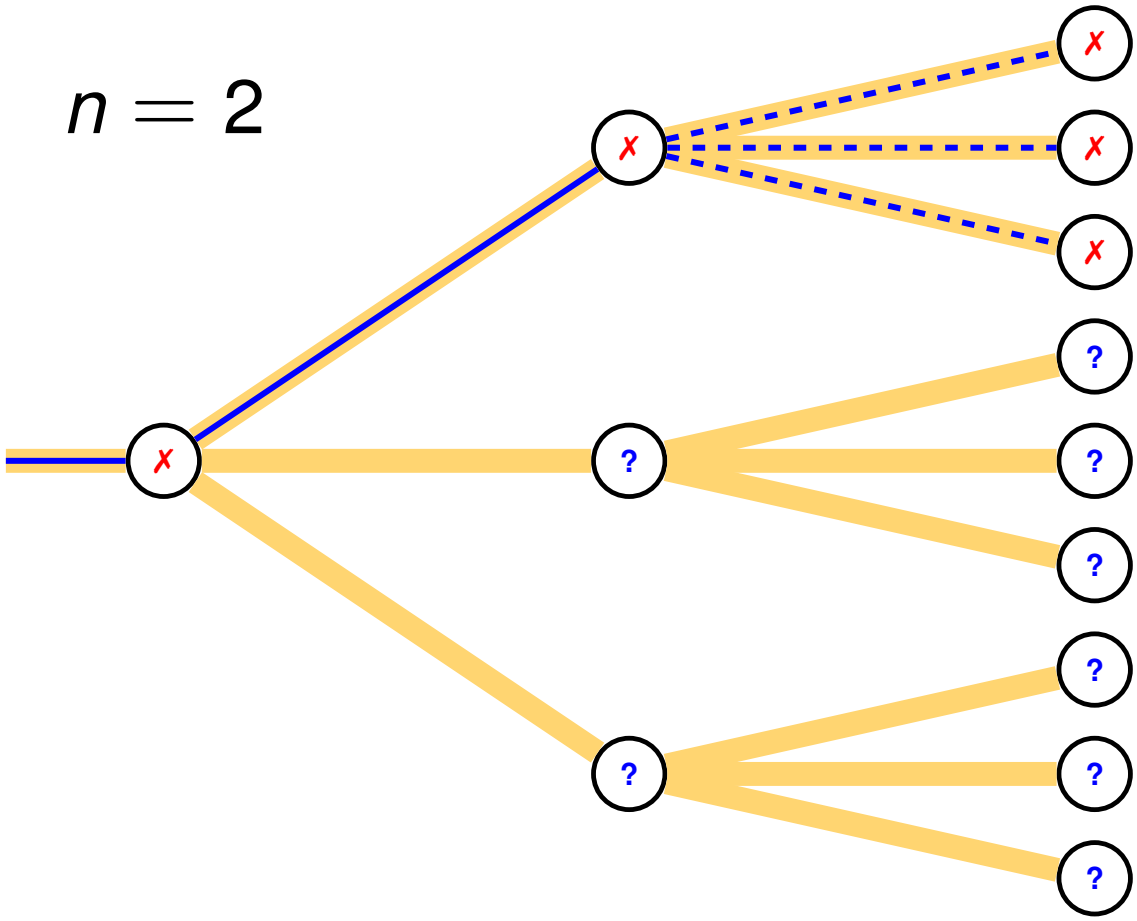
Kürzester Lösungsweg mit schrittweisem Backtracking



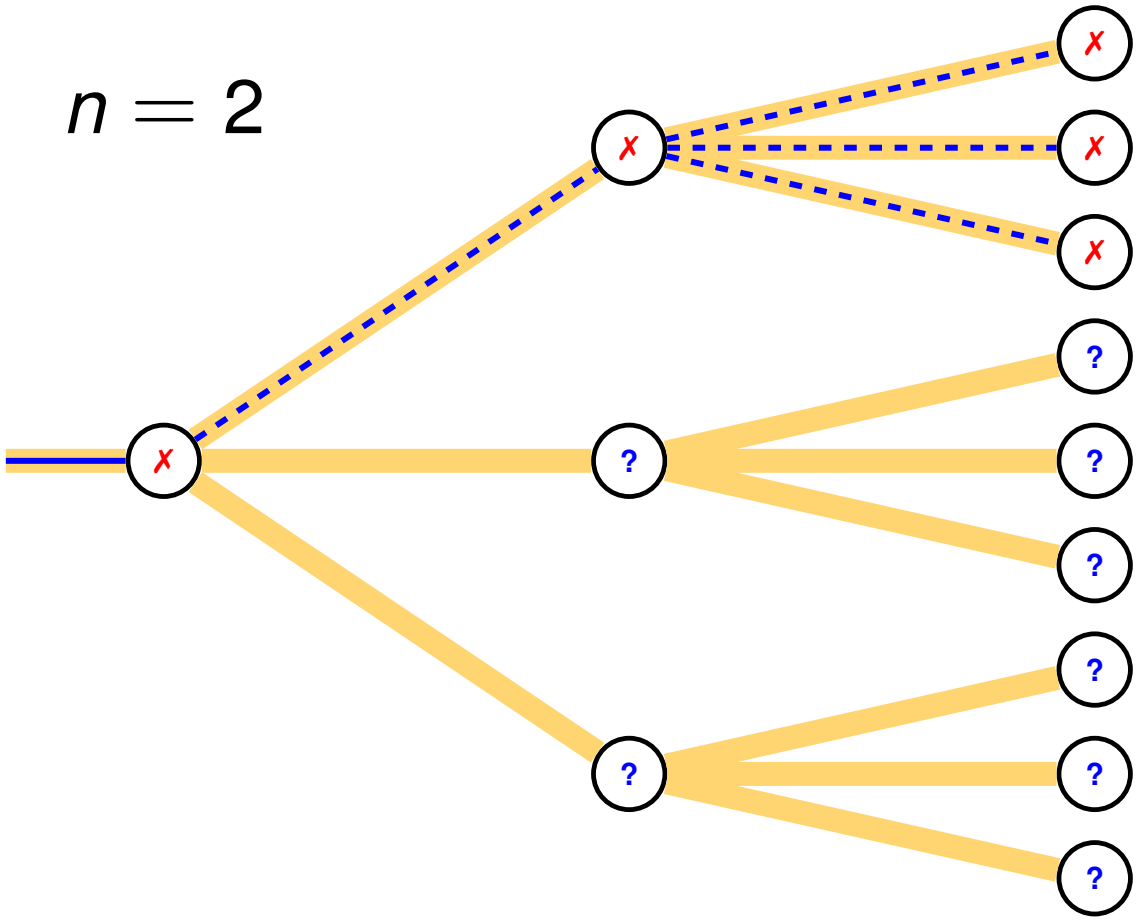
Kürzester Lösungsweg mit schrittweisem Backtracking



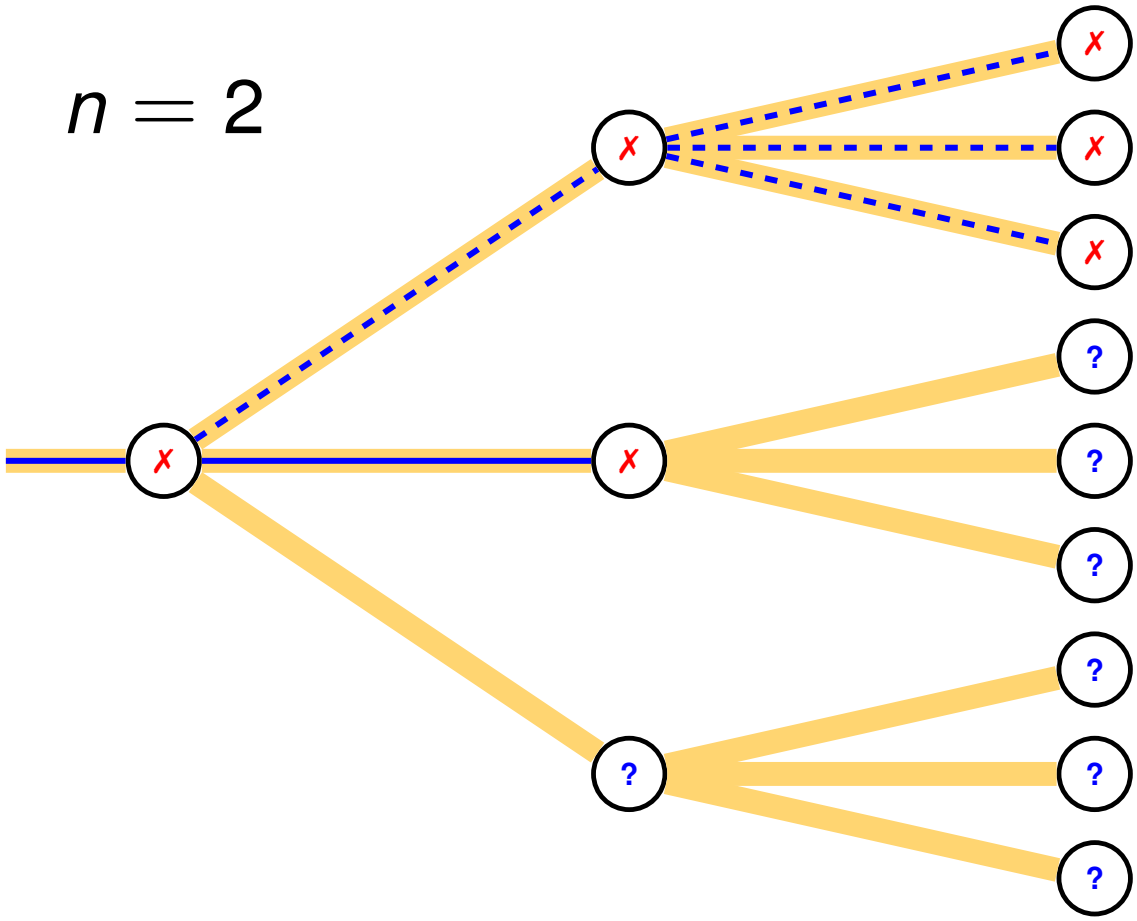
Kürzester Lösungsweg mit schrittweisem Backtracking



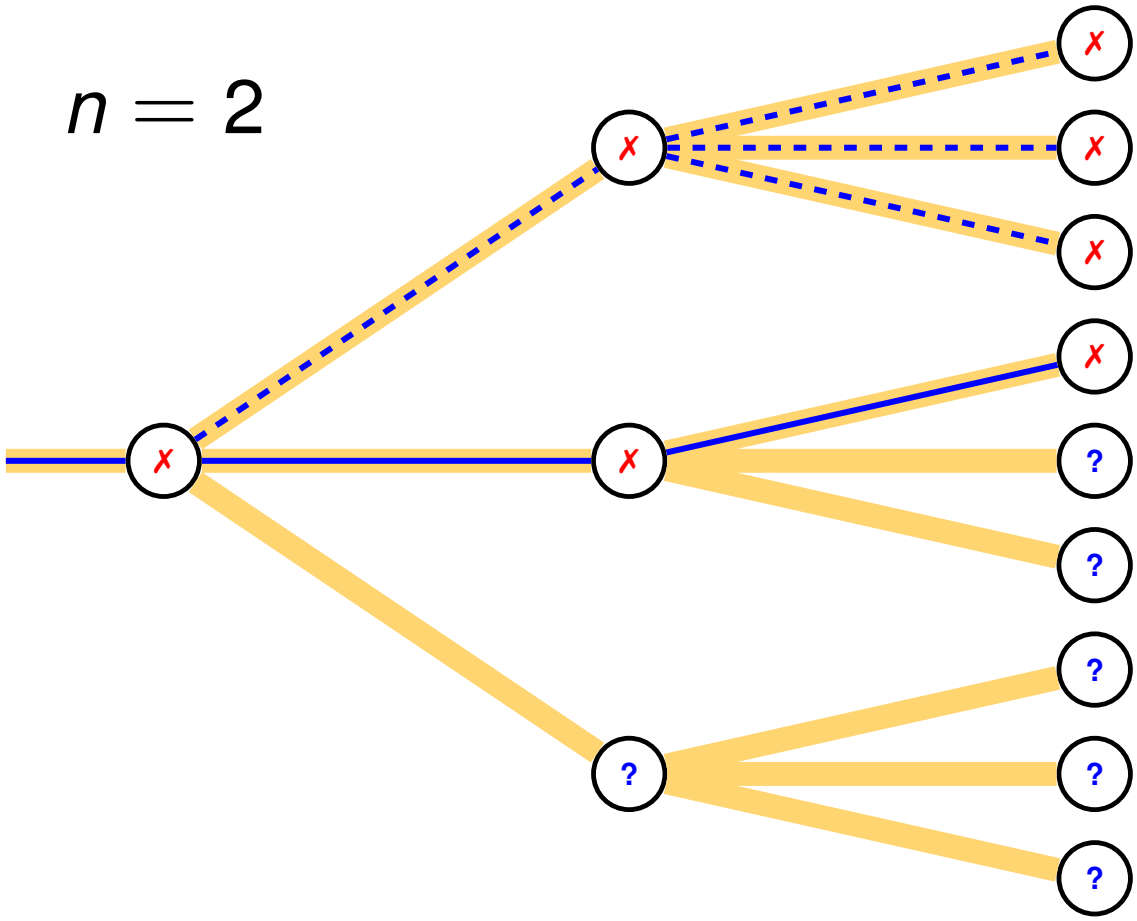
Kürzester Lösungsweg mit schrittweisem Backtracking



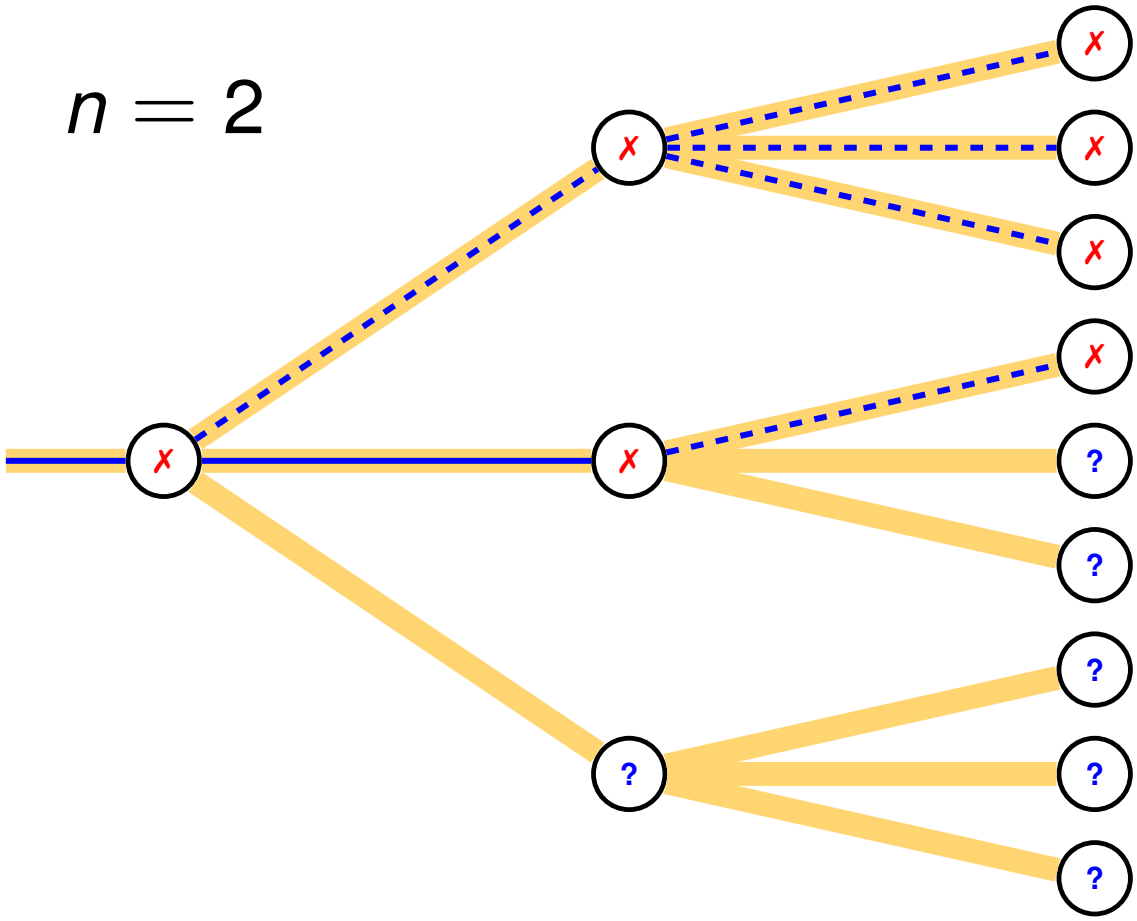
Kürzester Lösungsweg mit schrittweisem Backtracking



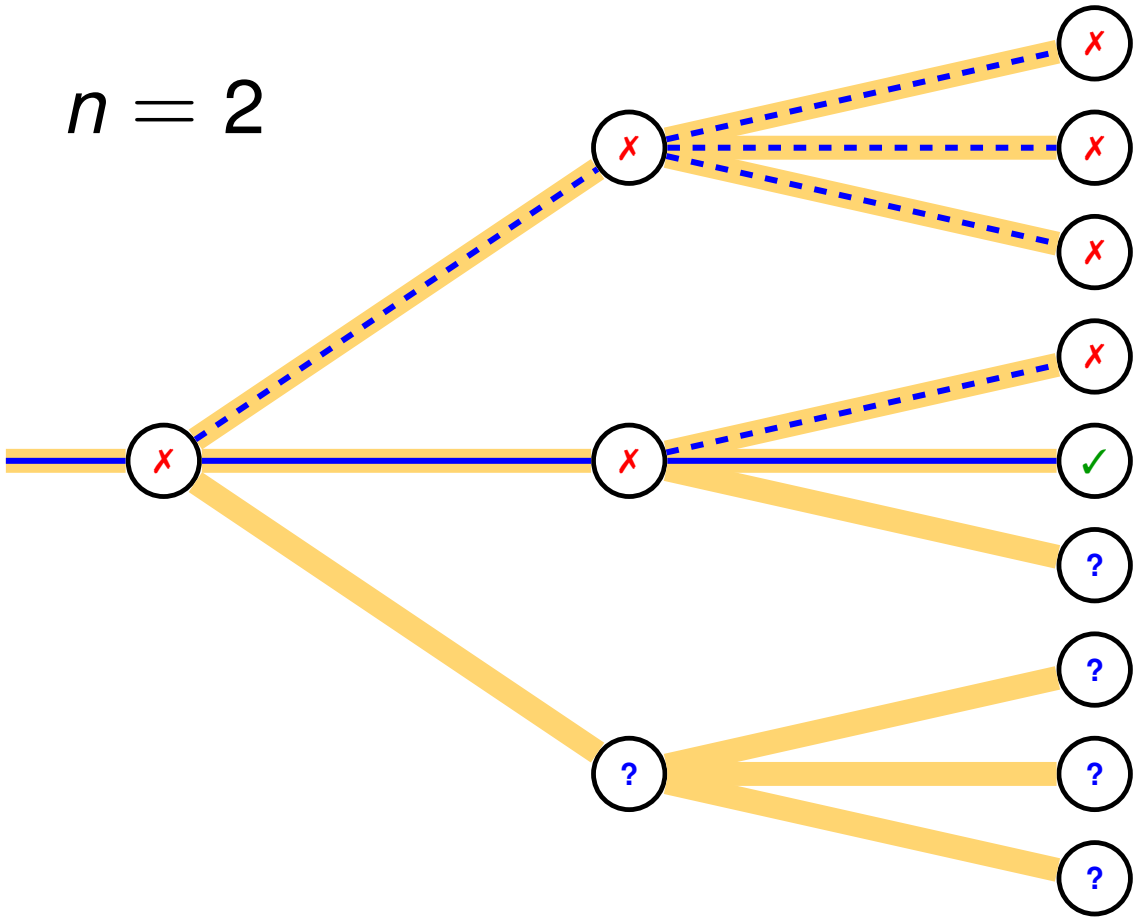
Kürzester Lösungsweg mit schrittweisem Backtracking



Kürzester Lösungsweg mit schrittweisem Backtracking



Kürzester Lösungsweg mit schrittweisem Backtracking



Exceptions

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Motivation

Beispiel ohne Exceptions...

```
public static int fib(int n) {
  if (n <= 0) {
    return -1; // ungültiger Parameter
  }
  if (n <= 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Nicht schön...

Aufrufer muss wissen, dass `fib()` im Fehlerfall `-1` zurückgibt; wird der Fehler beim Aufrufer nicht behandelt, wird mit falschem Ergebniswert weitergerechnet ☹

Motivation

Beispiel ohne Exceptions...

```
public static int fib(int n) {
  if (n <= 0) {
    return -1; // ungültiger Parameter
  }
  if (n <= 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Nicht schön...

Aufrufer muss wissen, dass `fib()` im Fehlerfall `-1` zurückgibt; wird der Fehler beim Aufrufer nicht behandelt, wird mit falschem Ergebniswert weitergerechnet ☹

Exceptions

- Exception $\equiv$ Ausnahme
  - signalisiert das Eintreten eines unerwarteten Ereignisses
  - häufig: ein Fehler bei der Ausführung des Programms
- erlaubt gezielte Ausnahmebehandlung:
  - geeignete Reaktion auf eine solche Ausnahmesituation
- keine Ausnahmebehandlung $\rightsquigarrow$ Abbruch des Programms
- man sagt:
  - eine Exception wird geworfen (*throw*), wenn sie ausgelöst wird
  - eine Exception wird gefangen (*catch*), wenn sie behandelt wird
- Exceptions führen zu neuen möglichen Kontrollflüssen
  - Sprung vom Punkt der Auslösung zum Punkt der Behandlung

Exceptions

- Exception $\equiv$ Ausnahme
  - signalisiert das Eintreten eines unerwarteten Ereignisses
  - häufig: ein Fehler bei der Ausführung des Programms
- erlaubt gezielte Ausnahmebehandlung:
  - geeignete Reaktion auf eine solche Ausnahmesituation
- keine Ausnahmebehandlung $\rightsquigarrow$ Abbruch des Programms
- man sagt:
  - eine Exception wird geworfen (*throw*), wenn sie ausgelöst wird
  - eine Exception wird gefangen (*catch*), wenn sie behandelt wird
- Exceptions führen zu neuen möglichen Kontrollflüssen
  - Sprung vom Punkt der Auslösung zum Punkt der Behandlung

Exceptions

- Exception $\equiv$ Ausnahme
  - signalisiert das Eintreten eines unerwarteten Ereignisses
  - häufig: ein Fehler bei der Ausführung des Programms
- erlaubt gezielte Ausnahmebehandlung:
  - geeignete Reaktion auf eine solche Ausnahmesituation
- keine Ausnahmebehandlung $\rightsquigarrow$ Abbruch des Programms
- man sagt:
  - eine Exception wird geworfen (*throw*), wenn sie ausgelöst wird
  - eine Exception wird gefangen (*catch*), wenn sie behandelt wird
- Exceptions führen zu neuen möglichen Kontrollflüssen
  - Sprung vom Punkt der Auslösung zum Punkt der Behandlung

Exceptions

- Exception $\equiv$ Ausnahme
  - signalisiert das Eintreten eines unerwarteten Ereignisses
  - häufig: ein Fehler bei der Ausführung des Programms
- erlaubt gezielte Ausnahmebehandlung:
  - geeignete Reaktion auf eine solche Ausnahmesituation
- keine Ausnahmebehandlung $\rightsquigarrow$ Abbruch des Programms
- man sagt:
  - eine Exception wird geworfen (*throw*), wenn sie ausgelöst wird
  - eine Exception wird gefangen (*catch*), wenn sie behandelt wird
- Exceptions führen zu neuen möglichen Kontrollflüssen
  - Sprung vom Punkt der Auslösung zum Punkt der Behandlung

Errors

- Error $\equiv$ *Fehler*
  - problematischer Zustand, den man kaum beheben kann
  - häufig: Fehler, der von der Laufzeitumgebung signalisiert wird
    - kein Platz mehr auf dem Stack vorhanden
    - kein Speicher mehr vorhanden
    - ...
  - in der Regel keine sinnvolle Ausnahmebehandlung möglich

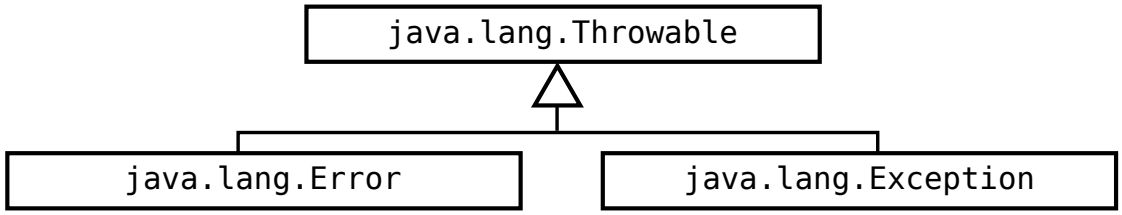
  $\leadsto$ Errors führen in der Regel zum Abbruch des Programms

Errors

- Error $\equiv$ *Fehler*
  - problematischer Zustand, den man kaum beheben kann
  - häufig: Fehler, der von der Laufzeitumgebung signalisiert wird
    - kein Platz mehr auf dem Stack vorhanden
    - kein Speicher mehr vorhanden
    - ...
  - in der Regel keine sinnvolle Ausnahmebehandlung möglich

  $\rightsquigarrow$ Errors führen in der Regel zum Abbruch des Programms

Klassen für Exceptions und Errors in Java



- `Throwable`:
  - Oberklasse für alles, was „geworfen“ werden kann
  - `Exception` und Unterklassen davon:
    - Ausnahmen, die i.A. behandelt werden können
  - `Error` und Unterklassen davon:
    - Ausnahmen, die i.A. nicht „sinnvoll“ behandelt werden können

Checked/Unchecked Exceptions

- Java unterscheidet *Checked* und *Unchecked Exceptions*
  - *Checked Exception*:
    - diese Exceptions *müssen* behandelt werden
    - falls diese nicht behandelt werden $\rightsquigarrow$ Compiler-Fehler

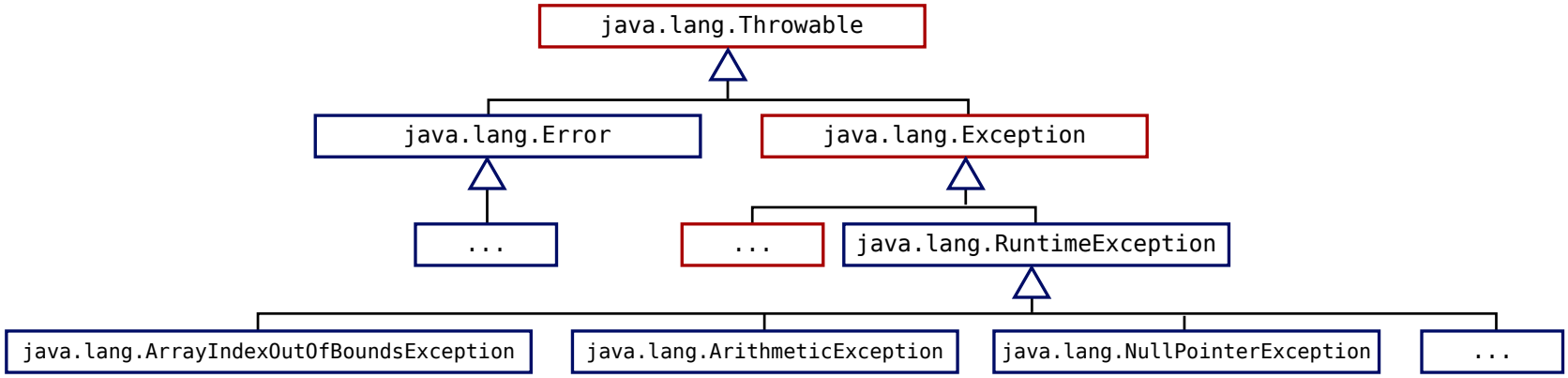
  $\rightsquigarrow$ Programmierer *muss* sich mit Ausnahmebehandlung beschäftigen

  $\rightsquigarrow$ Ausnahmebehandlung an *jeder* potentiellen Stelle einer Ausnahme

  - *Unchecked Exception*:
    - Unterklassen von `java.lang.{RuntimeException,Error}`
    - diese Exceptions *können* behandelt werden
    - Ausnahmebehandlung liegt in der Verantwortung des Programmierers

  $\rightsquigarrow$ Ausnahmebehandlung kann weggelassen werden

Checked/Unchecked Exceptions



Unchecked Exception / *Checked Exception*

„Standard-Exceptions“ in Java

- Exceptions in Java:
  - auch ohne explizites Werfen durch den Anwendungsprogrammierer
    - u.a. durch Laufzeitsystem und Klassen der Java-API

$\rightsquigarrow$ Anwendungsprogrammierer sollte diese Exceptions kennen!

- im Folgenden: Übersicht über ein paar „Standard-Exceptions“ des Java-APIs

NullPointerException

NullPointerException

- Versuch, eine `null`-Referenz zu dereferenzieren
- vom Laufzeitsystem generiert
- *Unchecked Exception*

Beispiel

```
String[] arr = null;
System.out.println(arr[2]); // NullPointerException
```

ArrayIndexOutOfBoundsException

ArrayIndexOutOfBoundsException

- Versuch, auf ein nicht vorhandenes Array-Element zuzugreifen
- vom Laufzeitsystem generiert
- *Unchecked Exception*

Beispiel

```
String[] arr = new String[2];
System.out.println(arr[2]); // ArrayIndexOutOfBoundsException
```

ArithmeticException

ArithmeticException

- bei einem arithmetischen Fehler (Integer-Division durch 0, ...)
- vom Laufzeitsystem und Klassen der Java-API generiert
- *Unchecked Exception*

Beispiel

```
int i = 1/0; // ArithmeticException
```

ClassCastException

ClassCastException

- Versuch einer Typ-Konvertierung mit inkompatiblen Typen
- vom Laufzeitsystem generiert
- *Unchecked Exception*

Beispiel

```
Object i = Integer.valueOf(7);
String s = (String) i; // ClassCastException
```

NumberFormatException

NumberFormatException

- Versuch, einen `String` mit falschem Format in eine Zahl zu konvertieren
- von Klassen der Java-API generiert
- *Unchecked Exception*

Beispiel

```
String s = "Hello";
int i = Integer.parseInt(s); // NumberFormatException
```

IllegalArgumentException

IllegalArgumentException

- signalisiert, dass (mindestens) ein Argument einen ungültigen Wert hat
- von Klassen der Java-API generiert
- *Unchecked Exception*

Beispiel

```
// fiktive Klasse 'Percentage'
Percentage p = new Percentage(102); // IllegalArgumentException
```

„Standard-Errors“ in Java

- Errors in Java:
  - meist ohne explizites Werfen durch den Anwendungsprogrammierer
    - i.d.R. durch das Laufzeitsystem ausgelöst
    - in vielen Fällen Hinweis auf möglichen Implementierungsfehler

$\rightsquigarrow$ Anwendungsprogrammierer sollte auch typische Errors kennen!

- im Folgenden: Übersicht über ein paar „Standard-Errors“ des Java-APIs

StackOverflowError

StackOverflowError

- auf dem Stack ist kein Platz mehr für neue Variablen/Methodenaufrufe/...
  $\rightsquigarrow$ oft ein Indiz für Endlosrekursion
- vom Laufzeitsystem generiert
- *Unchecked* (wie alle `Errors`)

Beispiel

```
public int sum(int n) {
  return n + sum(n-1);
}

// ------------------------------------------------------------

System.out.println(sum(5)); // StackOverflowError
```

OutOfMemoryError

`OutOfMemoryError`

- auf dem Heap ist kein Platz mehr für neue Objekte
  $\rightsquigarrow$ oft ein Indiz für „zu große“ Datenstrukturen
- vom Laufzeitsystem generiert
- *Unchecked* (wie alle `Errors`)

Beispiel

```
String[] arr = new String[2000000000]; // OutOfMemoryError
```

AssertionError

AssertionError

- eine festgelegte `Assertion` schlägt fehl (*mehr Details gleich*)
  $\rightsquigarrow$ eine Bedingung, die eigentlich immer erfüllt sein sollte, ist es nicht
  $\rightsquigarrow$ tritt beispielsweise oft in (fehlschlagenden) Tests auf
- *Unchecked* (wie alle `Errors`)

Beispiel

```
int i = 0;
int j = -1;
assert (i < j); // AssertionError
```

Werfen von Exceptions

Werfen von Exceptions

Zum Werfen einer Exception wird ein neues Objekt der entsprechenden Exception-Klasse erzeugt und mittels `throw` geworfen.
Die meisten Exception-Klassen haben auch einen Konstruktor, dem man eine „Nachricht“ als `String` übergeben kann, die den Fehler genauer beschreibt.

Beispiel

```
// ohne "Nachricht"
if (irgendwasPasstNicht) {
  throw new Exception();
}

// mit "Nachricht"
if (irgendwasPasstNicht) {
  throw new Exception("irgendwas passt nicht");
}
```

Werfen von Exceptions

Werfen von Exceptions

Zum Werfen einer Exception wird ein neues Objekt der entsprechenden Exception-Klasse erzeugt und mittels `throw` geworfen.
Die meisten Exception-Klassen haben auch einen Konstruktor, dem man eine „Nachricht“ als `String` übergeben kann, die den Fehler genauer beschreibt.

Beispiel

```
// ohne "Nachricht"
if (irgendwasPasstNicht) {
  throw new Exception();
}

// mit "Nachricht"
if (irgendwasPasstNicht) {
  throw new Exception("irgendwas passt nicht");
}
```

Im Beispiel von vorhin

Beispiel mit Exceptions...

```
public static int fib(int n) {
  if (n <= 0) {
    throw new IllegalArgumentException("n muss > 0 sein");
  }
  if (n <= 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Schöner als vorher...

Der Aufrufer *kann*[a] jetzt einfacher auf die Ausnahmesituation reagieren. Wenn er es nicht tut, wird das Programm abgebrochen, statt mit fehlerhaften Werten weiter zu rechnen.

[a] zur Erinnerung: `IllegalArgumentException` ist eine *Unchecked Exception*

Im Beispiel von vorhin

Beispiel mit Exceptions...

```
public static int fib(int n) {
  if (n <= 0) {
    throw new IllegalArgumentException("n muss > 0 sein");
  }
  if (n <= 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Schöner als vorher...

Der Aufrufer *kann*[a] jetzt einfacher auf die Ausnahmesituation reagieren. Wenn er es nicht tut, wird das Programm abgebrochen, statt mit fehlerhaften Werten weiter zu rechnen.

[a] zur Erinnerung: `IllegalArgumentException` ist eine *Unchecked Exception*

Konstruktoren und Methoden von Exceptions

Typische Konstruktoren

Die meisten Exception-Klassen haben mehrere Konstruktoren:

- Standardkonstruktor (ohne Parameter)
- mit Nachricht (`String message`)
- mit Grund (`Throwable cause`)
  - `Throwable`, das für diese `Exception` „verantwortlich“ ist
- mit Nachricht und Grund
- ...

Wichtige Methoden

- `getMessage()` liefert die Nachricht, die bei der Erzeugung angegeben wurde
- `getCause()` liefert den Grund für diese Exception
- `printStackTrace()` gibt den Stacktrace aus, siehe später
- ...

Konstruktoren und Methoden von Exceptions

Typische Konstruktoren

Die meisten Exception-Klassen haben mehrere Konstruktoren:

- Standardkonstruktor (ohne Parameter)
- mit Nachricht (`String message`)
- mit Grund (`Throwable cause`)
  - `Throwable`, das für diese `Exception` „verantwortlich“ ist
- mit Nachricht und Grund
- ...

Wichtige Methoden

- `getMessage()` liefert die Nachricht, die bei der Erzeugung angegeben wurde
- `getCause()` liefert den Grund für diese Exception
- `printStackTrace()` gibt den Stacktrace aus, siehe später
- ...

Fangen von Exceptions

- drei Schlüsselwörter für das Fangen von Exceptions
- markieren jeweils Beginn eines Blocks

`try` Code, der eine Exception werfen kann

`catch` Code für die Behandlung eines bestimmten Exception-Typs; mehrere Blöcke für unterschiedliche Exceptions möglich

`finally` Code, der unabhängig vom Auftreten einer Exception ausgeführt wird

Beispiel für die Syntax

```
try {
  // Code, der potentiell eine Exception auslöst
} catch (ExceptionA exA) { /* falls ExceptionA aufgetreten ist */
} catch (ExceptionB exB) { /* falls ExceptionB aufgetreten ist */
} finally { /* wird immer ausgeführt */
}
```

Fangen von Exceptions: Beispiel

Beispiel

```
public static void main(String[] args) {
  int argument;

  try {
    argument = Integer.parseInt(args[0]);
    System.out.println("Das Argument war: " + argument);
  } catch (ArrayIndexOutOfBoundsException ex) {
    System.err.println("Keine Argumente übergeben: " + ex.getMessage());
  } catch (NumberFormatException ex) {
    System.err.println("Ungültiges Argument: " + ex.getMessage());
  } finally {
    System.out.println("Tschüß.");
  }
}
```

Fangen von Exceptions: Beispiel

Verschiedene Ausführungen

```
$> java Beispiel 5
Das Argument war: 5
Tschüß.

$> java Beispiel
Keine Argumente übergeben: 0
Tschüß.

$> java Beispiel abc
Ungültiges Argument: For input string: "abc"
Tschüß.
```

Delegieren von Exceptions

- eine Exception muss nicht in der werfenden Methode gefangen werden

$\rightsquigarrow$ dem Aufrufer der Methode soll die Ausnahme signalisiert werden

- wenn kein (passender) `catch`-Block vorhanden ist:
  - Exception wird „automatisch“ an Aufrufer delegiert
  - allerdings: *Checked Exceptions müssen* behandelt werden

  $\rightsquigarrow$ Checked Exceptions müssen explizit delegiert werden
    - dazu Angabe dieser Exceptions mittels `throws` bei Methodendeklaration

Hinweis

Auch *Unchecked Exceptions* können hinter `throws` aufgeführt werden. Dies hat aber nur informativen Charakter und ist für die Übersetzbarkeit des Programms nicht relevant.

Delegieren von Exceptions

- eine Exception muss nicht in der werfenden Methode gefangen werden

$\rightsquigarrow$ dem Aufrufer der Methode soll die Ausnahme signalisiert werden

- wenn kein (passender) `catch`-Block vorhanden ist:
  - Exception wird „automatisch“ an Aufrufer delegiert
  - allerdings: *Checked Exceptions müssen* behandelt werden

  $\rightsquigarrow$ Checked Exceptions müssen explizit delegiert werden
    - dazu Angabe dieser Exceptions mittels `throws` bei Methodendeklaration

Hinweis

Auch *Unchecked Exceptions* können hinter `throws` aufgeführt werden. Dies hat aber nur informativen Charakter und ist für die Übersetzbarkeit des Programms nicht relevant.

Delegieren von Exceptions: Beispiel

Beispiel

```
class MyCheckedException extends Exception {
  /* Checked Exception, da Unterklasse von Exception */
}

class Foo {
  // ...
  public Foo(int a) throws MyCheckedException {
    if (a < 0) {
        throw new MyCheckedException();
    }
    // ...
  }
}
```

Fangen *und* Delegieren

- man kann ein und dieselbe Exception fangen *und* delegieren
  - dasselbe `Exception`-Objekt wird dazu mittels `throw` erneut geworfen

⇝ erlaubt Ausnahmebehandlung an mehreren Stellen

Beispiel

```
public void bar() {
  try {
    foo(); // wirft manchmal eine RuntimeException
  } catch (RuntimeException re) {
     // ... (Ausnahmebehandlung in bar())
    throw re;
  }
}

public void bazz() {
  try {
    bar();
  } catch (RuntimeException re) {
    /* ... */
  }
}
```

Stacktrace

- ein Stacktrace zeigt die Methodenschachteln auf dem Stack

$\rightsquigarrow$ hilfreich für die Rückverfolgung einer Ausnahme

  - In welcher Methode wurde Ausnahme geworfen?
  - Von wo aus wurde diese Methode aufgerufen?

- falls eine Exception in Java nicht gefangen wird:
  - Abbruch des Programms
  - Ausgabe des Stacktrace

Beispiel für einen Stacktrace

```
Exception in thread "main" java.lang.NullPointerException
        at Buch.getTitle(Buch.java:16)
        at Autor.getBookTitle(Autor.java:20)
        at Main.main(Main.java:13)
```

Stacktrace einer Exception ausgeben

- der Stacktrace einer Exception kann auch „manuell“ ausgegeben werden
  - Methode `printStackTrace()` der Klasse `Throwable`

⇝ kann beim Debuggen eines Programms hilfreich sein

Beispiel

```
try {
  // ...
} catch (Exception ex) {
  ex.printStackTrace();
}
```

Branch Coverage

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

JUnit

- JUnit $\equiv$ Framework zum Testen von Java-Programmen
  - genauer: *Unit Testing* (Testen von einzelnen Software-Komponenten)
- Tests können automatisiert ausgeführt werden

$\rightsquigarrow$ Tests häufig starten

$\rightsquigarrow$ neue Fehler fallen schnell auf

- IDEs (z.B. Eclipse) bieten häufig besondere Unterstützung (siehe unten)

JUnit: Beispiel (I)

Folgender Code soll getestet werden:

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

- zum Testen könnte man theoretisch einfach eine `main`-Methode schreiben
  - Nachteil: man müsste den Test auf Korrektheit selbst implementieren

⇝ deshalb: JUnit-Testfall

JUnit: Beispiel (I)

Folgender Code soll getestet werden:

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

- zum Testen könnte man theoretisch einfach eine `main`-Methode schreiben
  - Nachteil: man müsste den Test auf Korrektheit selbst implementieren

$\rightsquigarrow$ deshalb: JUnit-Testfall

JUnit: Beispiel (II)

Erster JUnit-Testfall

```
import static org.junit.Assert.*; // erforderliche Imports
import org.junit.Test; // werden von Eclipse automatisch erzeugt

public class RechteckTest { // Test für Klasse Rechteck

  @Test // Hinweis an JUnit: diese Methode ist ein Test
  public void testeUmfang_0x0() {
    // Methode ist immer "public void" (kein static!)

    int umfang = Rechteck.umfang(0, 0);
      // Methodenaufruf: Methode umfang der Klasse Rechteck
      // mit Parameter laenge = 0 und breite = 0

    assertEquals("Umfang ist nicht 0", 0, umfang);
      // prüft, ob erwarteter Wert in Variable umfang steht
  }
}
```

JUnit: @Test

@Test

- `@Test` ist eine Annotation
  - kennzeichnet die nachfolgende Methode als JUnit-Test
- kann (optional) Parameter erhalten:
  - `@Test(timeout=42)`
    $\Rightarrow$ ist der Test nach 42 ms noch nicht fertig, bricht er ab (*Timeout*)
  - `@Test(expected=NullPointerException.class)`
    $\Rightarrow$ damit der Test erfolgreich ist, muss obige Exception geworfen werden
    (mehr zu Exceptions in ein paar Wochen...)

JUnit: Asserts (I)

assertEquals

- Methode zum Prüfen, ob zwei Variablen denselben Wert haben
- hat drei Parameter:
  1. Fehlermeldung (als `String`)
     - wird angezeigt, wenn Überprüfung fehlschlägt
     - dieser Parameter ist optional
  2. `expected`-Wert
     - (nahezu) beliebiger Typ
     - Wert, der *eigentlich* erwartet wird
  3. `actual`-Wert
     - selber Typ wie `expected`
     - tatsächlicher Wert, der mit `expected` verglichen wird

JUnit: Asserts (II)

Im Beispiel...

```
int umfang = Rechteck.umfang(0, 0);
assertEquals("Umfang ist nicht 0", 0, umfang);
```

- Umfang eines 0×0-Rechtecks sollte 0 sein $\rightsquigarrow$ 2. Parameter ist 0
- tatsächlicher Wert ist in `umfang` gespeichert $\rightsquigarrow$ 3. Parameter ist `umfang`
- falls `umfang` $\neq 0$ ist, wird „Umfang ist nicht 0“ ausgegeben $\rightsquigarrow$ 1. Parameter

Weitere Assert-Methoden

- `assertNotNull` zur Prüfung, ob ein `String`/Array/... den Wert `null` hat
- `assertArrayEquals` zur Prüfung, ob zwei Arrays denselben Inhalt haben
- `assertTrue`/`assertFalse` zur Prüfung, ob ein `boolean` den Wert `true`/`false` hat
- Details siehe `http://junit.org/apidocs/org/junit/Assert.html`

JUnit: Asserts (II)

Im Beispiel...

```
int umfang = Rechteck.umfang(0, 0);
assertEquals("Umfang ist nicht 0", 0, umfang);
```

- Umfang eines 0×0-Rechtecks sollte 0 sein $\rightsquigarrow$ 2. Parameter ist 0
- tatsächlicher Wert ist in `umfang` gespeichert $\rightsquigarrow$ 3. Parameter ist `umfang`
- falls `umfang` $\neq 0$ ist, wird „Umfang ist nicht 0“ ausgegeben $\rightsquigarrow$ 1. Parameter

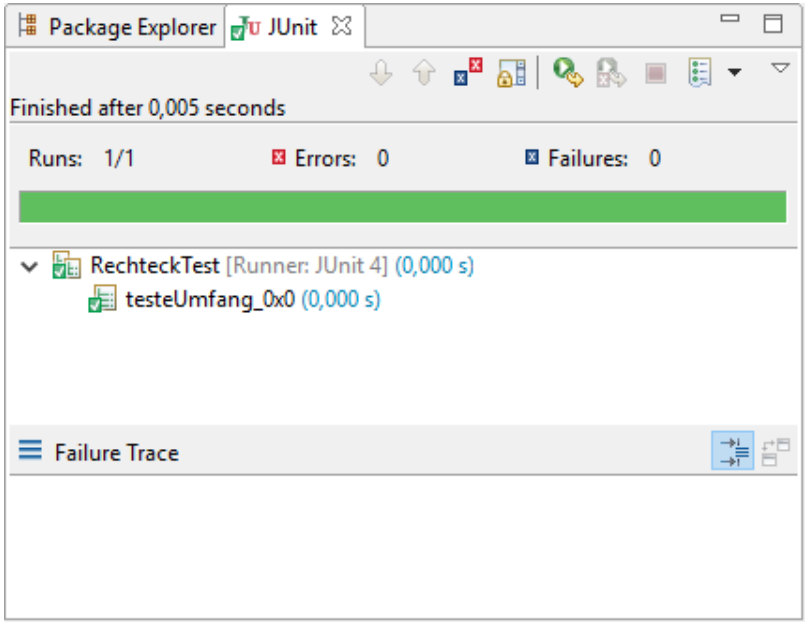
Weitere Assert-Methoden

- `assertNotNull` zur Prüfung, ob ein `String`/Array/... den Wert `null` hat
- `assertArrayEquals` zur Prüfung, ob zwei Arrays denselben Inhalt haben
- `assertTrue`/`assertFalse` zur Prüfung, ob ein `boolean` den Wert `true`/`false` hat
- Details siehe `http://junit.org/apidocs/org/junit/Assert.html`

Einschub: JUnit in Eclipse (I)

- Testklasse auswählen und auf „Run“ klicken
  - ggf. „`Run as JUnit Test`“ (oder ähnliches) auswählen

⇝ JUnit-Ansicht öffnet sich:



Einschub: JUnit in Eclipse (II)

- Mögliche Symbole:
  -  Test hat keine Fehler gefunden und ist erfolgreich durchgelaufen
  -  Test ist fehlgeschlagen, vermutlich ein Assert nicht erfolgreich
  -  Ausführung des Tests unterbrochen (Exception oder Timeout)

JUnit: Beispiel (III)

- heißt das jetzt, dass unser Beispielcode von vorhin korrekt funktioniert?
  - nein!
  - das war nur ein Test mit nur einem möglichen Eingabepaar
  - um alle Fehler zu finden, müsste man alle möglichen Eingaben testen
    - normalerweise ist das nicht möglich!
    - aber: weitere Tests schaden nicht

Weiterer Testfall

```
@Test
public void testeUmfang_5x4() { // neuer Test mit neuem Namen
  int umfang = Rechteck.umfang(5, 4);
  assertEquals("Umfang ist nicht 18", 18, umfang);
}
```

JUnit: Beispiel (III)

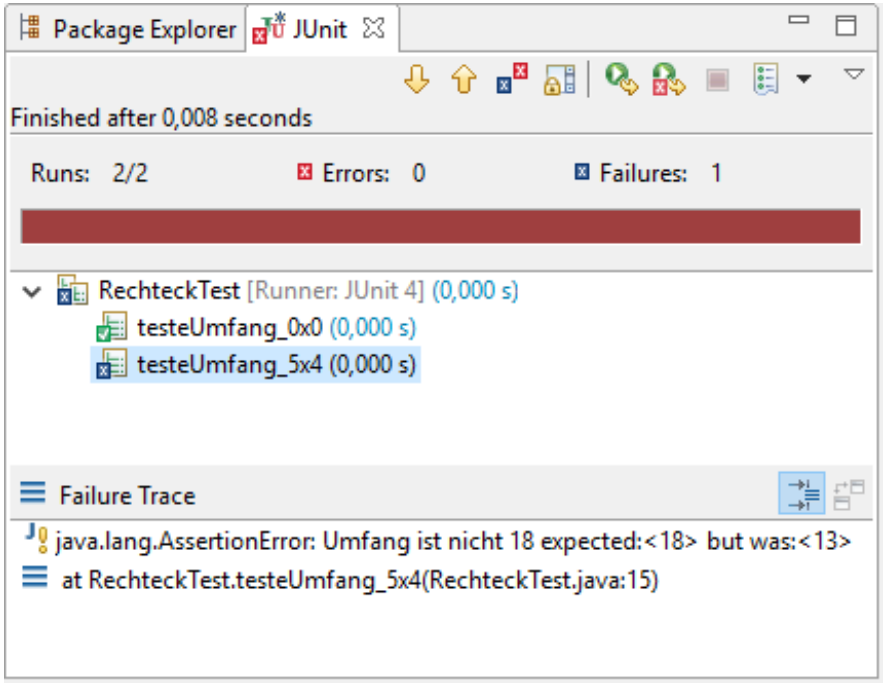
- heißt das jetzt, dass unser Beispielcode von vorhin korrekt funktioniert?
  - nein!
  - das war nur ein Test mit nur einem möglichen Eingabepaar
  - um alle Fehler zu finden, müsste man alle möglichen Eingaben testen
    - normalerweise ist das nicht möglich!
    - aber: weitere Tests schaden nicht

Weiterer Testfall

```
@Test
public void testeUmfang_5x4() { // neuer Test mit neuem Namen
  int umfang = Rechteck.umfang(5, 4);
  assertEquals("Umfang ist nicht 18", 18, umfang);
}
```

JUnit: Beispiel (IV)

- das passiert beim Ausführen:



JUnit: Beispiel (V)

- der neue Test ist fehlgeschlagen!
- im „Failure Trace“ kann man nachlesen, was passiert ist:
  - `java.lang.AssertionError` ist die Fehlerart (vorerst egal)
  - `Umfang ist nicht 18 expected <18> but was <13>`
    - unsere Fehlermeldung
    - zeigt auch den tatsächlichen Wert der Variable (hier: `13`)
  - `at RechteckTest.testeUmfang_5x4(RechteckTest.java:15)`
    - Zeile, in der der Test fehlgeschlagen ist
    - auf diesen Eintrag kann man drauf klicken
      - Eclipse springt dann automatisch an die Fehlerstelle

  ⇝ erleichtert Fehlerlokalisierung und -behebung

JUnit: Beispiel (V)

- der neue Test ist fehlgeschlagen!
- im „Failure Trace“ kann man nachlesen, was passiert ist:
  - `java.lang.AssertionError` ist die Fehlerart (vorerst egal)
  - `Umfang ist nicht 18 expected <18> but was <13>`
    - unsere Fehlermeldung
    - zeigt auch den tatsächlichen Wert der Variable (hier: 13)
  - `at RechteckTest.testeUmfang_5x4(RechteckTest.java:15)`
    - Zeile, in der der Test fehlgeschlagen ist
    - auf diesen Eintrag kann man drauf klicken
      - Eclipse springt dann automatisch an die Fehlerstelle

  ⇝ erleichtert Fehlerlokalisierung und -behebung

JUnit: Beispiel (V)

- der neue Test ist fehlgeschlagen!
- im „Failure Trace“ kann man nachlesen, was passiert ist:
  - `java.lang.AssertionError` ist die Fehlerart (vorerst egal)
  - `Umfang ist nicht 18 expected <18> but was <13>`
    - unsere Fehlermeldung
    - zeigt auch den tatsächlichen Wert der Variable (hier: 13)
  - `at RechteckTest.testeUmfang_5x4(RechteckTest.java:15)`
    - Zeile, in der der Test fehlgeschlagen ist
    - auf diesen Eintrag kann man drauf klicken
      - Eclipse springt dann automatisch an die Fehlerstelle

  ⇝ erleichtert Fehlerlokalisierung und -behebung

JUnit: Beispiel (V)

- der neue Test ist fehlgeschlagen!
- im „Failure Trace“ kann man nachlesen, was passiert ist:
  - `java.lang.AssertionError` ist die Fehlerart (vorerst egal)
  - `Umfang ist nicht 18 expected <18> but was <13>`
    - unsere Fehlermeldung
    - zeigt auch den tatsächlichen Wert der Variable (hier: 13)
  - `at RechteckTest.testeUmfang_5x4(RechteckTest.java:15)`
    - Zeile, in der der Test fehlgeschlagen ist
    - auf diesen Eintrag kann man drauf klicken
      - Eclipse springt dann automatisch an die Fehlerstelle

  $\rightsquigarrow$ erleichtert Fehlerlokalisierung und -behebung

JUnit: Beispiel (VI)

Zurück zum Code

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

Warum kommt bei 5 + 4 * 2 als Ergebnis 13 heraus?

- Punkt vor Strich: 5 + (4 * 2) = 5 + 8 = 13
- wir wollen eigentlich: (5 + 4) * 2 = 9 * 2 = 18
- also: Code muss angepasst und Fehler behoben werden

JUnit: Beispiel (VI)

Zurück zum Code

```java
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return laenge + breite * 2; // ob das wohl so stimmt?
  }
}
```

Warum kommt bei `5 + 4 * 2` als Ergebnis `13` heraus?

- Punkt vor Strich: `5 + (4 * 2) = 5 + 8 = 13`
- wir wollen eigentlich: `(5 + 4) * 2 = 9 * 2 = 18`
- also: Code muss angepasst und Fehler behoben werden

JUnit: Beispiel (VII)

Angepasster Code

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return (laenge+breite) * 2; // sollte jetzt so stimmen
  }
}
```

Tipp für die Übungsaufgaben

- die öffentlichen Testcases testen *nicht* immer die gesamte Funktionalität ab
  - z.B. könnte nur testeUmfang_0x0() enthalten sein
- eigene Tests schreiben kostet oft nicht viel Zeit
- Fehler lassen sich damit oft schon vor Abgabeende finden

JUnit: Beispiel (VII)

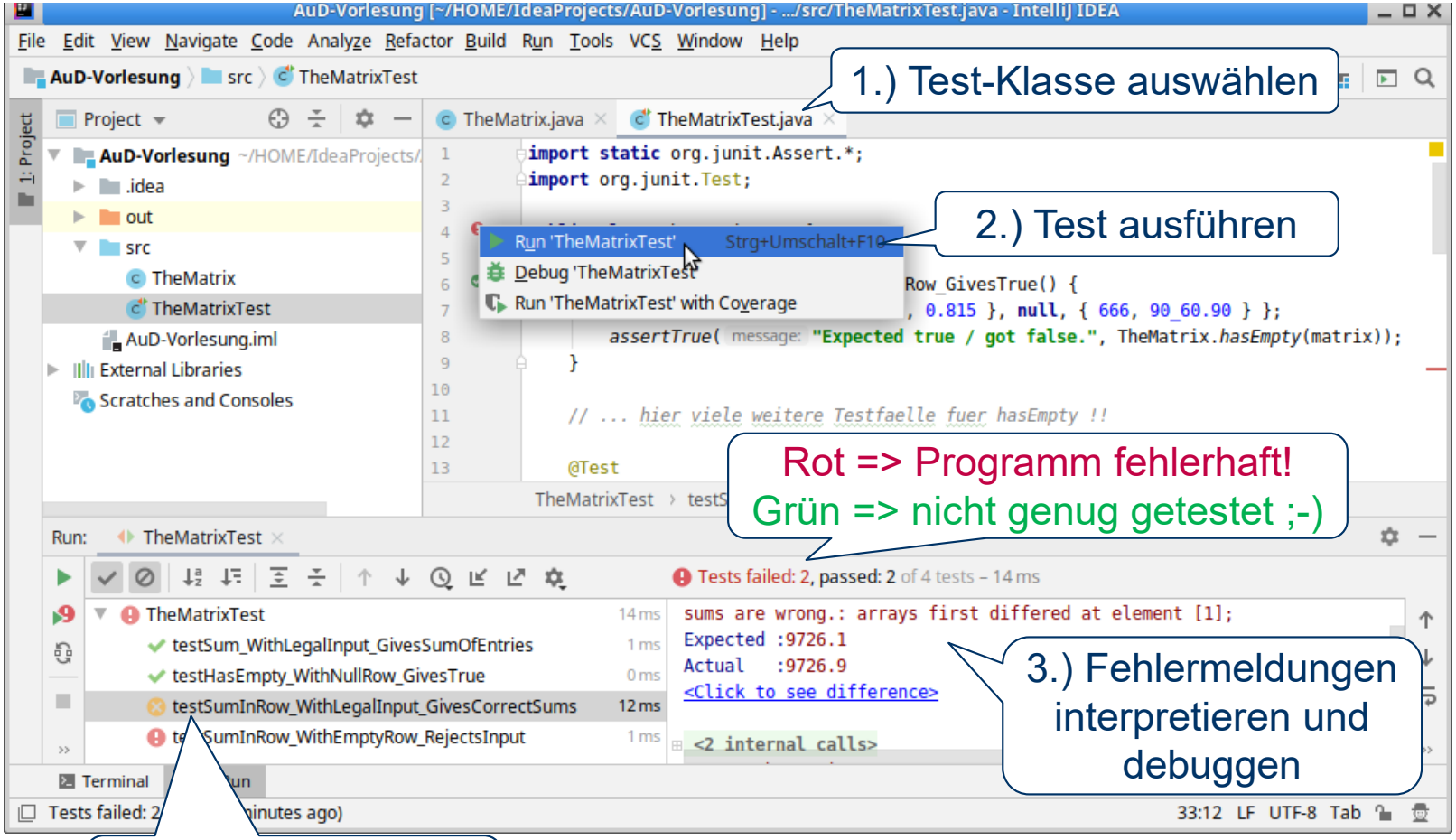
Angepasster Code

```
public class Rechteck {
  /* Berechnet den Umfang eines Rechtecks */
  public static int umfang(int laenge, int breite) {
    return (laenge+breite) * 2; // sollte jetzt so stimmen
  }
}
```

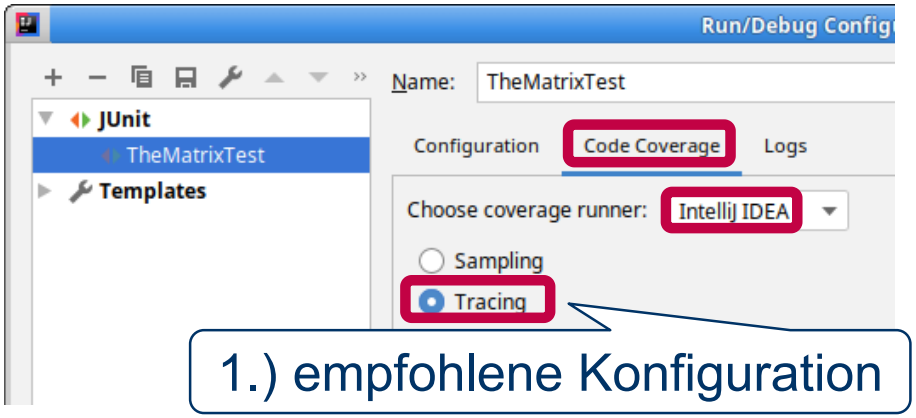
Tipp für die Übungsaufgaben

- die öffentlichen Testcases testen *nicht* immer die gesamte Funktionalität ab
  - z.B. könnte nur testeUmfang_0x0() enthalten sein
- eigene Tests schreiben kostet oft nicht viel Zeit
- Fehler lassen sich damit oft schon vor Abgabeende finden

Testfall erzeugen in Eclipse

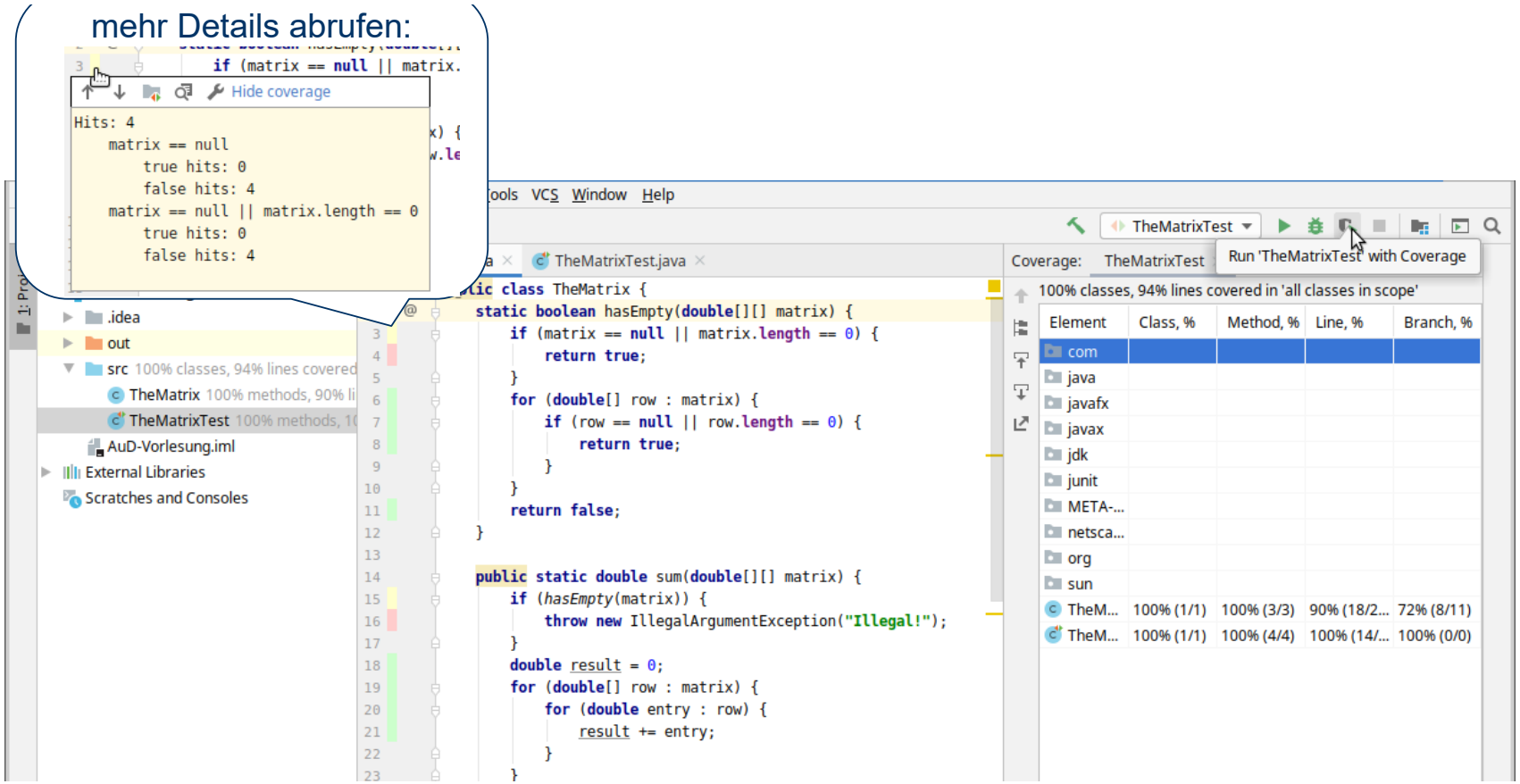


Testfall erzeugen in Eclipse



Testfall erzeugen in Eclipse

Testfall erzeugen in Eclipse



Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT