

A Einführendes Programmierbeispiel

Package: u02_Labyrinth

Offline Hilfe:

- <https://goalkicker.com/>
- Zeal / Dash

A.1 Labyrinth in Java – Erweitere die vorgegebene Datei Labyrinth.java

- Input analysieren

```
// Angabe Labyrinth I
"#####",
"#  #  #  #",
"## # ## #",
"#  #  #  #",
"## ## #  #",
"#          #",
"## ##### #",
"#          #",
"# ##### #",
"#  #  #  #",
"#  #  #  #",
"#####A####"
```

- TODOs erledigen
- finde **einen** Weg zum Ausgang **A** für alle vier Labyrinth (boolean, ja/nein) `boolean suchen(spalte, zeile,...)`
 - starte bei der angegebenen `spalte, zeile`
 - zur Analyse mit Zwischenschritten (Ausgabe)
 - Tipp: kurze Pause nach jedem Schritt erleichtert das Beobachten
dh. an geeigneter Stelle: `printLabyrinth()` und `sleep()` oder Breakpoint im Debugger setzen.
 - Wichtig: immer das *aktuelle* Feld betrachten – nicht die Nachbarn (die rufen wir ohnehin rekursiv auf).

Grober Ablauf – rekursiv:

 - * einfachster Fall: Ausgang gefunden
 - * leeres Feld
 - markieren
 - rekursive Suche alle vier Nachbarn – wie kombiniert man die vier Ergebnisse?
- finde **alle** Wege für alle drei Labyrinth (Anzahl)
 - **zusätzliche** Methode `int sucheAlle(...)`
 - **keine static** Variable als Zähler (sondern direkt beim Aufruf der Rekursion mitzählen)
 - Tipp: Markierung nach der Suche wieder entfernen oder in der Funktion das Labyrinth kopieren
- schreibe eine Methode, die das Labyrinth aus einer Datei einliest. Tipp: <https://www.dcode.fr/maze-generator>

A.2 Labyrinth in Python

- Programmiere die Labyrinthsuche in Python!
- Was ist schneller (ohne Ausgabe der Zwischenschritte)? Java oder Python. Tipp: Profiler verwenden. Dokumentiere anhand von Screenshots die Ergebnisse: Datei u\dieNummer_\derTitel_Profiler.pdf!
- Erweitere das Python-Programm so, dass man es von der Kommandozeile mit folgender usage starten kann:

```
usage: Labyrinth.py [-h] [-x XSTART] [-y YSTART] [-p] [-t] [-d DELAY] filename
```

```
calculate number of ways through a labyrinth
```

```
positional arguments:
```

```
  filename                file containing the labyrinth to solve
```

```
options:
```

```
  -h, --help              show this help message and exit
```

```
  -x XSTART, --xstart XSTART
                        x-coordinate to start
```

```
  -y YSTART, --ystart YSTART
                        y-coordinate to start
```

```
  -p, --print             print output of every solution
```

```
  -t, --time             print total calculation time (in milliseconds)
```

```
  -d DELAY, --delay DELAY
                        delay after printing a solution (in milliseconds)
```

B Wegsuche – Theorie

B.1 Allgemeiner Ansatz

Vergleiche beiliegenden Foliensatz aus der unverbindlichen Übung *Einführung in Artificial Intelligence*. Ohne Rekursion, stattdessen

- Todo-Liste: Grenze des bekannten Bereichs (**frontier**)
 - am Anfang; der Startpunkt bzw. dessen direkte Umgebung
- Liste durchgehen, jeden Punkt durch seine Nachbarn ersetzen
- Liste von bereits besuchten *Orten* (**explored**, vermeidet Schleifen)

```
def path_search(start, successors, is_goal):
    """
    Find the path from start state to a state
    such that is_goal(state) is true.
    """
    if is_goal(start):
        return [start]
    explored = set() # set of states we have visited
    frontier = [ [start] ] # ordered list of paths we have reached
    while frontier:
        path = pop_first(frontier) # depends on sort order
        s = path[-1] # last element of path: s
        for state in successors(s):
            if state not in explored: # no loop
                explored.add(state)
                # path2 = path + [state, action] # if we need the action, also changes loop
                path2 = path + [state]
                if is_goal(state):
                    return path2
            else:
                frontier.append(path2)
    return []
```

B.2 Varianten

Unterschied in der Implementierung der Liste (**frontier**) bzw. **pop_first()**:

- Tiefensuche
 - sortiert nach *Entfernung/Tiefe* (große zuerst)
 - Stack (bzw. Rekursion)
- Breitensuche
 - sortiert nach *Entfernung/Tiefe* (kleine zuerst)
 - Queue (bzw. Liste): vorne entnehmen, neue Elemente hinten anhängen
 - Alternativ: neue Liste mit allen Nachbarn von allen Positionen in **frontier**
- A*
 - sortiert nach bisheriger *Entfernung* + *Abschätzung(restlicher Weg)*
- Dijkstra: Weg zu allen Knoten
 - sortiert nach *Entfernung*

Hinweise:

- Die aktuelle Liste muss nicht vollständig sortiert werden – es reicht wenn man das kleinste bzw. größte Element an der ersten Stelle hat. Das ist fast in jeder Programmiersprache bereits vordefiniert: Priority-Queue bzw. Heap (Java: `PriorityQueue`, Python: `import heapq`).
- Auch für die Queue gibt es eine Alternative zu einer einfachen Liste: Java – `LinkedList`, Python: `collections.deque`