Abstraktion

Bei dem beschriebenen Problem handelt es sich im Grundlegenden um das Finden eines optimalen Pfades mithilfe eines Wegfindungsalgorithmus; dabei gibt es aber nicht einen Zustand, der nur die Koordinate/derzeitige Position unseres Objektes, das das Labyrinth traversiert festhält, sondern einen, der zwei solcher Koordinaten zugleich betrachtet. Kurzum, es existiert ein Koordinatenpaar für – in dem Fall der Aufgabenstellung – zwei Personen; nennen wir diese folglich Person A und Person B. Wir möchten also schlussendlich nur einen Pfad finden, der universell für beide Labyrinthe und somit beide Personen – A und B – anwendbar ist. Dafür bieten sich mehrere Algorithmen an.

Lösungsweg zum Algorithmus

Vorgedanken

Es stehen mehrere Algorithmen für den genannten Anwendungszweck zur Auswahl – dazu gehören A* und Dijkstra – besonders anbieten tut sich hierbei jedoch der (recht simple) Breadth-First-Suchalgorithmus (BFS), da er:

- vergleichsweise leicht zu implementieren ist,
- sich äußerst gut in das oben genannte Schema mit Zuständen aus Koordinatenpaaren für Person A und B verwenden lässt,
- eine noch reell anwendbare und vertretbare Laufzeit- sowie auch Speichereffizienz im Kontext der Aufgabenstellung aufweist (für die Big O-Notation, siehe Seite 6).

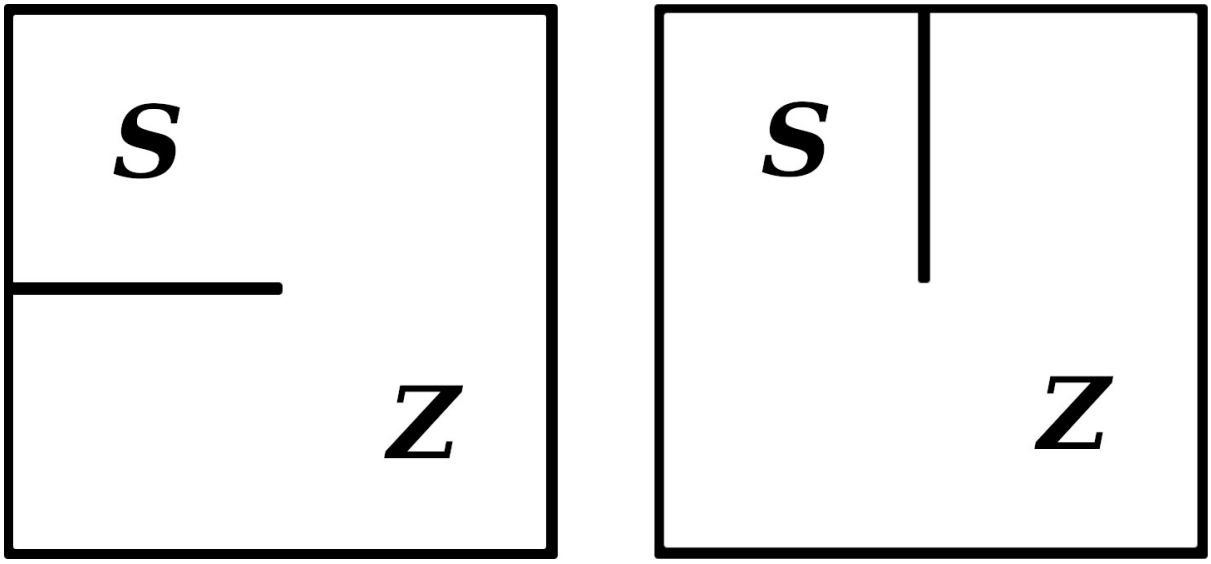
Da wir unser Problem aber lediglich als einen Baum (man könnte argumentieren, es sei ein Spielbaum ähnlich wie bei Engines in Gesellschaftsspielen wie Schach, wo es mehrere mögliche Züge gibt) darstellen können – jeder Knoten ist ein Zustand und jeder Ast zu den weiteren Kindknoten beschreibt einen Zug, also nach oben, rechts, unten oder links – kann man das jetzt beschriebene Verfahren auch problemlos auf Algorithmen wie A* portieren; dabei wäre eine Nutzung der Manhattan-Distanz zum Zielpunkt (n, m) zur Führung des Algorithmus eine sinnvolle Optimierung. Anstelle einer normalen Queue würde dann eine Priority Queue verwendet werden, die stets nach der besten Distanz zum Ziel umsortiert wird

und Schritte, die eher zum Ziel führen als andere, früher untersucht. Ich habe dies ebenfalls (lediglich als kleinen Exkurs) implementiert, stieß jedoch noch – durch das fanatische Verfolgen einer suboptimalen Heuristik[1] – auf eine schlechtere Qualität der Ausgaben (mehr Schritte), auch wenn das Programm etwas schneller lief.

Ich entschied letztendlich: für die gegebenen Aufgaben war A* schlicht und ergreifend nicht notwendig; eine Breadth-First-Suche reichte vollkommen aus, um an das Ziel zu kommen, selbst für größere Labyrinthe von weit über 100 x 100 Feldern – und fand die besten Pfade.

Funktionsweise des Algorithmus

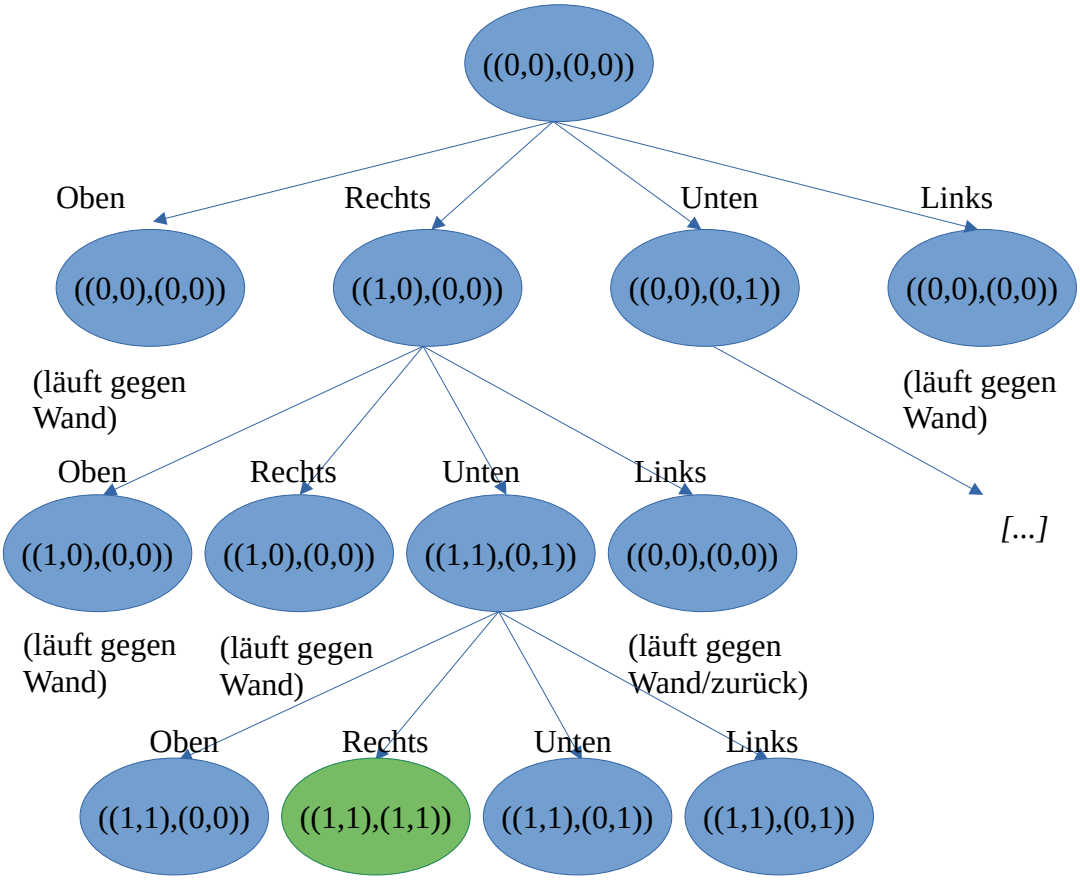
Stellen wir uns diesen Spielbaum visuell dar; gegeben seien zwei sehr kleine Labyrinthe.



Ziel ist es, für beide Labyrinthe ins Ziel zu gelangen und eine so kurz wie mögliche Liste an Anweisungen zu generieren. Beide Personen – A und B – beginnen bei den Koordinaten (0, 0). Zusammen bilden sie also das Koordinatenpaar ((0, 0), (0, 0)), welches hier unseren Zustand verkörpert.

1 Getestet wurden: Reziproke Summe der Manhattan-Distanzen beider Personen in den Labyrinthen zum Ziel, arithmetisches Mittel beider Entfernungen zum Ziel, Summe der Entfernungen zum Ziel.

Zeichnen wir uns nun einen (unvollständigen) Baum auf, der unseren möglichen Spielverlauf darstellt.



Zwar ist der Baum an vielerlei Stellen stark verkürzt, aber wir können trotzdem den wichtigsten Aspekt sehen: Sobald wir an unserer Zielkoordinate angekommen sind (ergo: sobald wir diese mit der BFS im Spielbaum gefunden haben), können wir einfach den genommenen Pfad wieder bis nach oben hin zurückverfolgen und dann daraus den optimalen Weg rekonstruieren. Machen wir das, sehen wir: Wir können vom Punkt (1, 1) über „Rechts“, „Unten“ und noch einmal „Rechts“ zum Startpunkt zurückkehren. Der Vollständigkeit halber drehen wir die Reihenfolge der eben ermittelten Schritte noch einmal um, denn logischerweise müssen wir vom Startpunkt aus zum Ziel kommen. Hier ändert dies nichts, der Schritt ist aber dennoch notwendig. Und damit haben wir unsere Anweisungen für beide Labyrinthe:

Rechts → Unten → Rechts

Dies löst beide Labyrinthe in nur drei Schritten – die Minimalanzahl. Was jetzt noch fehlt, ist eine Funktion, die uns aus einem gegebenen Knoten die nächsten Zustände generiert. Diese nimmt beide Labyrinthe und den derzeitigen Zustand – also das Koordinatenpaar – als Argumente und generiert daraus vier mögliche Folgezustände (in unserer Analogie mit dem Baum also vier Kindknoten; in der eigentlichen Implementierung nutzen wir jedoch in Ermangelung an Notwendigkeit nicht tatsächlich einen mit Referenzen oder gar Pointern verketteten Baum als Datenstruktur, es hilft jedoch zur Visualisierung).

Beschreibung der Funktion zur Generierung der Folgezustände

Diese eben genannte Funktion lässt sich wie folgt beschreiben:

1. Bei einem gegebenen Zustand, zwei Labyrinthen für jeweils beide Personen und der Zielkoordinate,
2. wiederhole für jede Richtung aus OBEN, RECHTS, UNTEN, LINKS:
    1. Wiederhole für jede Person:
        1. Wenn die Person in eine Wand in ihrem Labyrinth rennen würde, bereits am Ziel ist oder das Gebiet des Labyrinthes verlassen würde: Person bleibt dort stehen, wo sie ist.
        2. Wenn die Person in eine Grube fällt: setze ihre Position auf den Start, also (0, 0), zurück.
        3. Ansonsten: Bewege die Person in die Richtung, aktualisiere die Position (x, y) entsprechend.
    2. Füge die Positionen der individuellen Personen zurück in einen Zustand.
3. Übergebe alle generierten Zustände.

Beschreibung des übrigen Wegfindungsalgorithmus

Nun fehlt nur noch der übrige Algorithmus zur Wegfindung; er ist lediglich eine Implementierung der BFS und funktioniert wie folgt:

1. Initialisiere ein Dictionary, das für jeden Zustand bereithält, durch welche Richtung (in Spielterminologie, durch welchen Zug) man in ebendiesen Zustand gekommen ist und was der Zustand zuvor war,
2. initialisiere zudem eine Queue (bzw. „Schlange“ an „anstehenden“ Zuständen, die abgearbeitet werden sollen),
3. füge den Startzustand (in welchem beide Personen, A und B, am Startpunkt stehen) in diese Queue ein: ((0, 0), (0, 0)).
4. Solange die Queue nicht leer ist:
    1. Entnehme den ersten (!), anstehenden Zustand aus der Queue,
    2. ist der Zustand der Endzustand *((n – 1, m – 1), (n – 1, m – 1))*? Wenn ja, verlasse die Schleife und gehe über zur Rückverfolgung des Wegs;
    3. sonst: generiere alle möglichen Kindknoten unseres Spielbaumes (also für alle möglichen Züge) mit der oben erläuterten Funktion.
        1. Wenn wir diesen Knoten bereits zuvor besucht haben (er also bereits in unserem Dictionary genannt wird), ist dieser zu überspringen und die restlichen Unterpunkte (2 und 3) werden nicht ausgeführt.
        2. Ansonsten wird der neue Zustand der Queue am Ende (!) hinzugefügt.
        3. Zudem wird in unserem Dictionary dieser neue Zustand notiert, mitsamt dem derzeitigen Zustand und der Richtung, die wir für diesen einschlugen.
5. Sobald der Endzustand erreicht wurde (siehe Punkt 4.2):
    1. Initialisiere eine Liste, die später die Anweisungen enthält,
    2. setze den derzeitigen Zustand Z auf den Endzustand.
    3. Solange der Z nicht dem Startzustand entspricht:

    1. Suche den derzeitigen Zustand Z im Dictionary (verfolge ihn somit weiter zurück) und füge immer die eingeschlagene Richtung ganz vorn an der Liste an, damit sie die korrekte Reihenfolge behält.
    2. Setze Z auf den im Dictionary gefundenen, vorherigen Zustand.
6. Wenn der Endzustand nie erreicht wird, gibt es keinen Endzustand im Spielbaum; die Suche konnte ihn mit keinen gültigen Zügen ermitteln; die Labyrinthe sind – zumindest simultan – unlösbar.

Mit diesem Algorithmus lassen sich (lösbare) Paare an Labyrinthen zuverlässig lösen.

Laufzeitanalyse

Die Zeitkomplexität einer simplen Breadth-First-Suche ist im Falle eines Labyrinthes logischerweise

$$O(nm),$$

denn es sind $n \cdot m$ Zustände zu untersuchen. Dies lässt sich auf unseren Anwendungsfall gut übertragen, da wir lediglich zwei Labyrinthe zugleich lösen und somit die Menge an möglichen Zuständen allen Kombinationen aus den Koordinatenpaaren beider Labyrinthe entspricht. Dies lässt sich als

$$O((nm)^2)$$

ausdrücken, was unserer Worst-Case-Zeitkomplexität entspricht.

Details zur Implementierung

Die Implementierung meines oben erklärten Algorithmus wurde wieder in Python vollzogen; unglücklicherweise weist diese Sprache eine notorische Speicherineffizienz auf: Ein primitiver Typ wie ein Integer nutzt gern einmal 28 Byte Speicher – in Sprachen wie C nutzt ein 32-bit-Integer bekanntlich gerade einmal 4 Byte:

```
Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.getsizeof(int(1234))
28
>>>
```

Für unseren Algorithmus ist das etwas unpraktisch, das Problem lässt sich aber einfach vermindern; gegenüber einer klassischen BFS habe ich in meiner Implementierung erst einmal auf eine Datenstruktur verzichtet, die im Auge behält, welche Zustände wir bereits besuchten; warum der zusätzliche Speicheraufwand, wenn unser Dictionary uns diese Aufgabe doch bereits abnimmt?

Zudem habe ich mich umfassend an Bitwise-Operationen bedient, die uns hier einen enormen Speichervorteil verschaffen können. Unsere Labyrinthe sind in den Aufgaben nie über 255 x 255 Kästen groß; uns genügt also pro Achse in den Koordinaten gerade einmal ein `uint_8`. Den gibt es in Python zwar nicht, wir können aber unsere Koordinatenpaare, die 2x 2 Integer enthalten (X- und Y-Koordinaten für jeweils beide Personen, A und B) mithilfe von Bitmanipulationen in nur einen 32-bit-Integer hineinstopfen. In Python ist der dann immer noch nur 28 Byte groß! Wir sparen somit eine Menge. Mithilfe von zwei dafür zurechtgelegten Funktionen können wir dann schnell und transparent konvertieren. Da Bitwise-Operationen in so ziemlich jeder Befehlssatzarchitektur zu Genüge verfügbar und optimiert sind, laufen diese mit extrem hoher Geschwindigkeit, wenn sie mittels eines Interpreters wie PyPy[2] JIT zu Maschinenbefehlen kompiliert werden. Wir büßen kaum Geschwindigkeit ein, sparen aber enorm an Speicher.

2 https://pypy.org/

Relevante Ausschnitte der Implementierung

1. Funktionen zum speicherplatzeffizienten packen und entpacken von verschachtelten Tupeln mittels Manipulationen auf Bit-Ebene:

```python
def enc(t: CoordPair) -> int:
    """
    Kodiert die speicherintensiven Koordinatenpaare mit bitweiser Logik, um
    Arbeitsspeicher zu sparen. Diese Funktion sollte sehr schnell sein, da sie
    nur bitweise Operationen enthält und eine Loop von Hand unrolled ist.
    """

    r = 0b0
    r |= t[0][0]
    r |= t[0][1] << 8
    r |= t[1][0] << 16
    r |= t[1][1] << 24

    return r


def dec(n: int) -> CoordPair:
    """
    Dekodiert die speichereffizienten Integer zurück zu Koordinatenpaaren.
    """

    mask = 0xFF
    a = n & mask
    b = (n >> 8) & mask
    c = (n >> 16) & mask
    d = n >> 24

    return ((a, b), (c, d))
```

2. Funktion zum Generieren der nächsten Zustände aus einem derzeitigen Zustand, den Labyrinthen und dem Endpunkt (gibt einen Python-Generator zurück, der on-the-fly erst die nächsten Kindknoten ermittelt, wenn `next()` auf diesen aufgerufen wird, wie z.B. in einer Schleife):

```python
def generate_next_positions(
    cur_pos: CoordPair, max_boundary: Coord, mazes: MazePair
):
    """
    Passt die Positionen beider Spieler*innen in Hinblick auf jede Richtung an.
    """

    for dir_ in [DIR_UP, DIR_RIGHT, DIR_DOWN, DIR_LEFT]:
        next_pos = [(0, 0), (0, 0)]
        for player in [0, 1]:
            cur_player_pos = cur_pos[player]

            if (
                get_point_in_maze(
                    mazes, player, cur_player_pos[0], cur_player_pos[1]
                )
                & WALL_DIR_MAPPINGS[dir_]  # ein Wenig bitwise-Magie! :)
            ) or cur_player_pos == max_boundary:
                # Hier laufen wir gegen eine Wand oder sind bereits am Ziel -
                # also tun wir stattdessen nichts!
                next_pos[player] = cur_player_pos
                continue

            next_player_pos = (
                clamp(cur_player_pos[0] + dir_[0], 0, max_boundary[0]),
                clamp(cur_player_pos[1] + dir_[1], 0, max_boundary[1]),
            )

            if (
                get_point_in_maze(
                    mazes, player, next_player_pos[0], next_player_pos[1]
                )
                & FLAG_PIT
            ):
                # Hier fallen wir in ein Loch, also gehen wir zurück zu (0, 0)!
                next_player_pos = (0, 0)

            next_pos[player] = next_player_pos

        if tuple(next_pos) == cur_pos:
            # Hatten wir schon, müssen wir nicht in Betracht ziehen.
            continue

        yield (next_pos[0], next_pos[1]), dir_
```

3. Funktion zum Lösen des Labyrinthes mittels BFS – Kommentare und Docstrings entfernt, damit es gut lesbar bleibt; Vollversion befindet sich im Quelltext.

```
def trace_mazes(
    mazes: MazePair,
) -> "tuple[dict[int, Tuple[int, int]], CoordPair, bool]":
    # Zuerst überprüfen wir, ob die Eingabe fehlerfrei ist.
    for idx, maze in enumerate(mazes, 1):
        if len(maze) == 0 or len(maze[0]) == 0:
            raise InvalidMazeException(f"Leeres Labyrinth #{idx}.")
        if any(len(el) != len(maze[0]) for el in maze):
            raise InvalidMazeException(
                f"Ungleich lange Zeilen im Labyrinth #{idx}!"
            )

    # Ob wir das Ziel gefunden haben
    success: bool = False

    start_coords, goal_coords = (0, 0), (
        len(mazes[0][0]) - 1,
        len(mazes[0]) - 1,
    )

    goal_pos: CoordPair = (goal_coords, goal_coords)
    begin_pos: CoordPair = (start_coords, start_coords)

    queue: "deque[int]" = deque([enc(begin_pos)])
    trace: "dict[int, Tuple[int, int]]" = {}

    while len(queue) != 0:
        cur_pos = dec(queue.popleft())

        if cur_pos == goal_pos:
            success = True
            break

        for new_pos, dir_ in generate_next_positions(
            cur_pos, goal_coords, mazes
        ):
            enc_new_pos: int = enc(new_pos)

            if enc_new_pos in trace:
                continue

            queue.append(enc_new_pos)
            trace[enc_new_pos] = (enc(cur_pos), DIR_MAP[dir_])

    return trace, goal_pos, success
```

4. Funktion zum Rekonstruieren des Pfades, nachdem das Dictionary vollständig aufgebaut und der Endpunkt gefunden wurde:

```python
def reconstruct_path_from_trace(
    trace: "dict[int, Tuple[int, int]]", goal_pos: CoordPair
) -> "tuple[str, int]":
    """
    Baut aus den Aufzeichnungen zu den Koordinatenpaaren durch Rückverfolgung
    des Lösungsweges einen optimalen Pfad auf.
    """

    # Nun müssen wir nur noch zurückverfolgen, wie unser Weg war!
    optimal_path: "list[Direction]" = []
    cur_pos = enc(goal_pos)

    # Solange wir nicht wieder zurück zum Anfang gefunden haben ...
    while dec(cur_pos) != ((0, 0), (0, 0)):
        # ... verfolgen wir die Spur, die wir uns zuvor hinterlassen haben.
        cur_pos, dir_ = trace[cur_pos]
        optimal_path.insert(0, DIR_MAP_INV[dir_])

    # Jetzt wandeln wir sie in menschenlesbare Form um.
    return " ".join([HUMAN_READABLE_DIRS[step] for step in optimal_path]), len(
        optimal_path
    )
```

Weitere Eigenschaften und Informationen zur Implementierung

Das Labyrinth wird in meiner Implementierung ebenfalls mittels **Les- und Schreiboperationen auf Bitebene** durchgeführt, um bessere Optimierungen durch JIT-Kompilierer zu ermöglichen, sehr schnelle Vergleiche und eine effiziente Detektion von Wänden bereitzustellen.

Hierbei besteht das Labyrinth selbst aus einer zweidimensionalen Liste voller Integer, die folgende Bedeutung auf Bit-Ebene (ausgehend von einem System mit Little-Endian) haben:

Das erlaubt uns beispielsweise, mittels Operationen wie „`feld & 16`“ innerhalb nur eines Taktzyklus unserer CPU zu ermitteln, ob es sich bei einem Feld um eine Grube handelt. Dies ist sehr schnell. Davon macht die Funktion zum Generieren der nächsten Zustände Gebrauch.

Wie auch meine Implementierung für Aufgabe 1 handelt es sich bei meiner Implementierung um eine weitestgehend UNIX-konforme Befehlszeilenanwendung.

Es gibt wieder ein paar Command-Line-Flags, mit denen man das Programm beeinflussen kann:

Flag	**Funktionalität**
`--time`	Misst die benötigte Zeit zum Lösen des Labyrinthpaars.
`--help`	Gibt Informationen zur korrekten Programmnutzung aus.

Ein Beispiel zur Ausführung des Programms im Terminal wäre:

```
$ pypy ./labyrinthe.py ./Beispiele/labyrinth0.txt -time
```

Und auch hier – weitere Eigenschaften der Implementierung:

- ist umfassend dokumentiert und kommentiert,
- nutzt keinerlei Bibliotheken von Dritten,
- ist sehr streng typisiert; weder Linter-Fehler noch Linter-Warnungen,
- sollte dem PEP8-Standard vollständig entsprechen und konform sein,
- läuft problemlos in verschiedenen Interpretern, darunter CPython und PyPy,
- kann unter Anwendung von JIT-Compilation sehr hohe Geschwindigkeiten und gute Speichereffizienz erreichen,
- folgt den UNIX-Paradigmen und gibt korrekte Exit-Codes an den ausführenden Elternprozess zurück.

Es wird stark empfohlen, das Programm auf einem unixoiden oder unix-ähnlichen Betriebssystem (macOS, Ubuntu, Debian, …) mittels eines JIT-Compilers wie PyPy auszuführen. Genügend RAM sollte bei großen Labyrinthen bereitgestellt werden. Im Notfall bietet sich natürlich das Verwenden des Swaps an.

Ausgaben des Programms zu den Beispieleingaben von der Website des BWINF

Folgend sind die Ausgaben des Programms zu den Beispieleingaben von der Website des BWINF sichtbar. Berechnet wurden sie mit folgendem Setup:

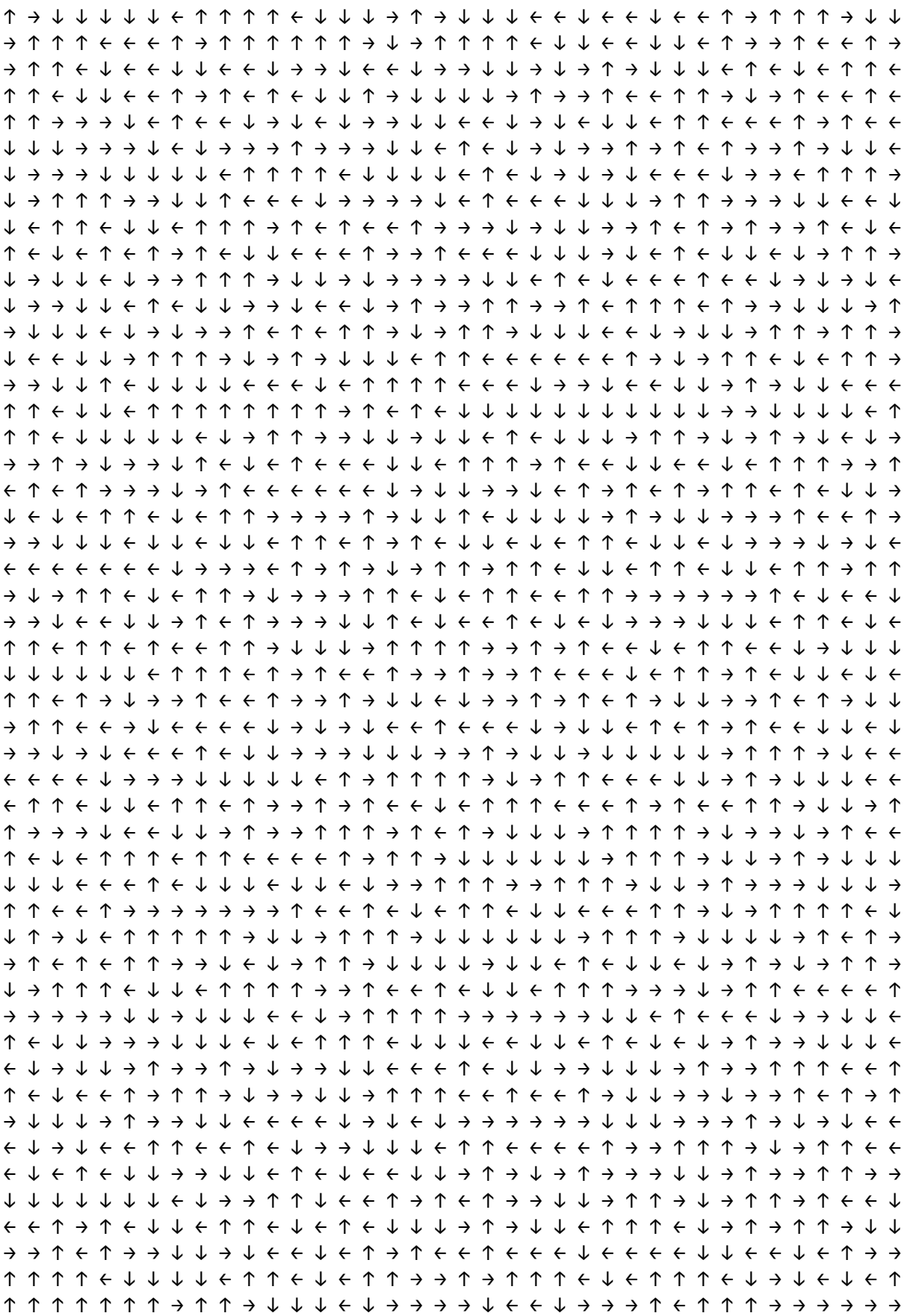
Betriebssystem	Ubuntu 22.04, kernel 6.8.0-57-generic
Interpreter	PyPy 3.11 v7.3.19
CPU	AMD Ryzen 7 7700X 8 Kerne @ 5.4 GHz
Arbeitsspeicher	32 GB DDR5 @ 5600 MT/s CL-36

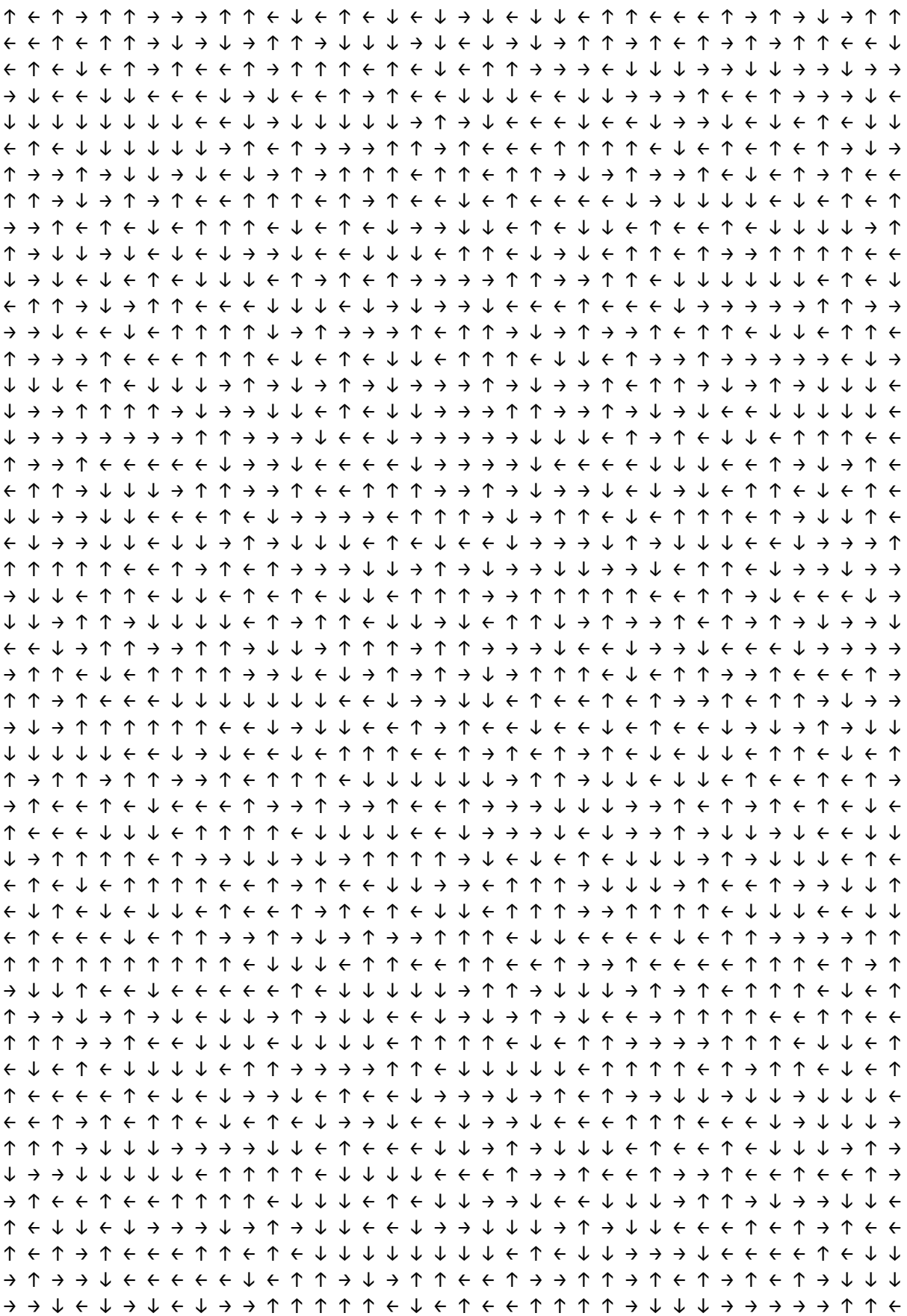
```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe0.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (8 Schritte). Lösung:

↓ ↓ → ↑ ↑ → ↓ ↓

--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe1.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (31 Schritte). Lösung:

→ → → → ↓ ↓ ← ↑ ← ↓ ← ↑ ← ↓ ↓ → ↑ → ↓ → ↑ ↑ ← ← ↑ → → → ↓ ↓ ↓

--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe2.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (65 Schritte). Lösung:

→ → ↓ ↓ ↓ → → → ↑ ↑ → → → ↓ ← ↓ → ↓ → ↓ ← ↓ ← ↑ ← ← ← ← ↑ ← ← ↓ ↓ ← ↓ → ↓ →
→ → → ↓ ↓ → → ↑ ↑ ↑ → ↑ ↑ → → → → → ↓ ← ↓ ← ↓ ↓ → → ↓

--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe3.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (164 Schritte). Lösung:

↓ ↓ ↓ ↓ → ↓ ↓ → ↑ → → → ↓ ↓ ↓ ↓ → → → → ↓ ← ↓ ↓ ↓ ← ↓ ↓ ↓ ← ← ↓ ← ↓ ← ↓ ↓ ↓
↓ → → ↓ ↓ → → ↓ ↓ → ← ↓ ↓ ↓ ← ↓ → ↓ ← ← ↓ ← ↑ ← ↓ ↓ → → ↓ ↓ → → ↓ ← ↓ ↓ → →
```

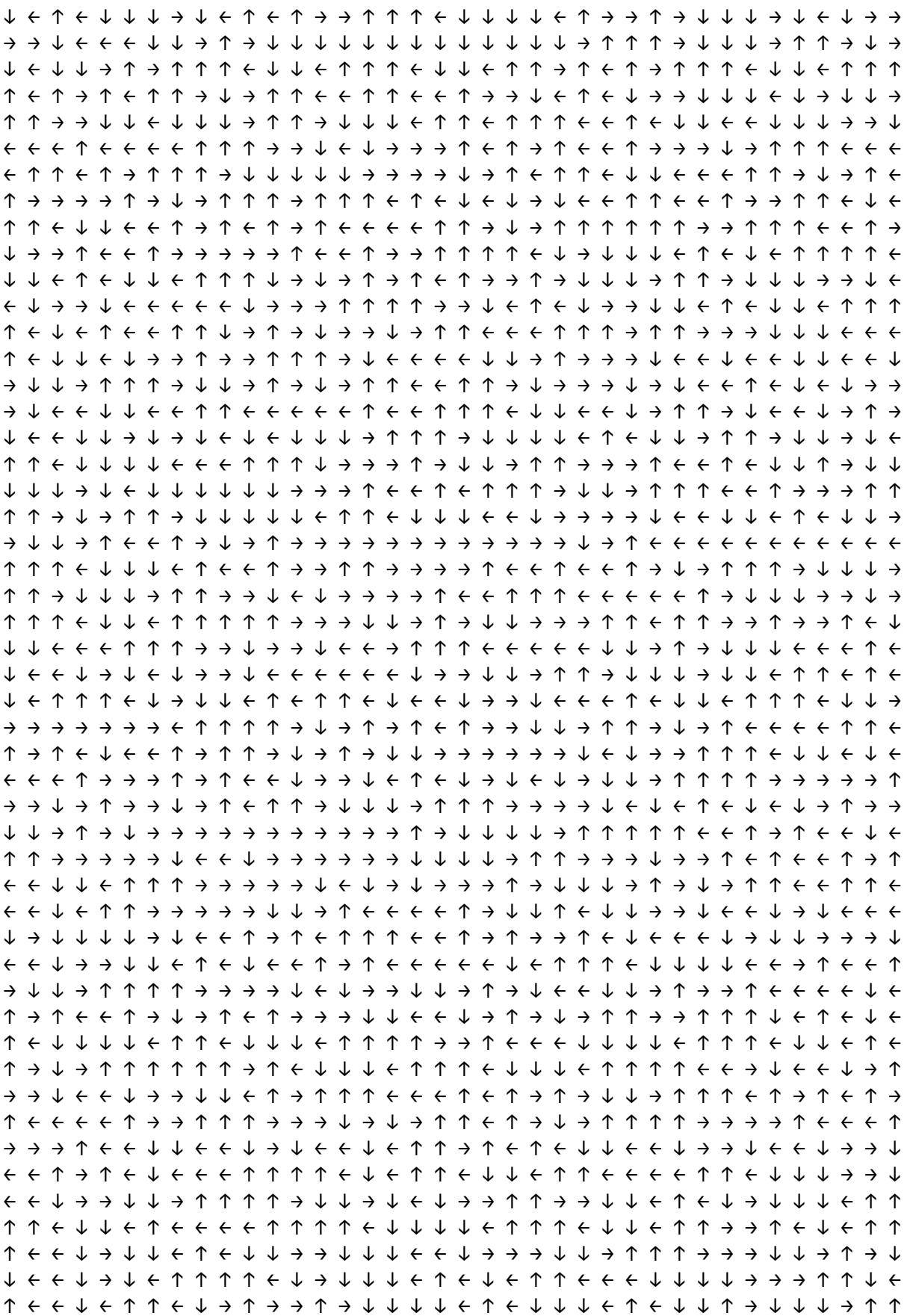
→ → ↓ ↓ → → ↓ → → ↓ → ↓ ↓ ← ← ↓ ↓ ← ↓ ← ↓ ↓ → ↑ → ↓ ↓ ↓ ← ← ↓ ← ↑ ← ↑ ← ↑ ←
↓ ← ↓ → → ↓ ↓ ↓ → → ↑ → → ↑ → ↑ → ↓ ↓ ↓ ↓ ← ↓ ← ← ↑ ↑ → ← ↓ ↓ ↓ ↓ ← ↓ ↓ ↓ →
↑ → ↓ → ↓ → ↓ → → → → →

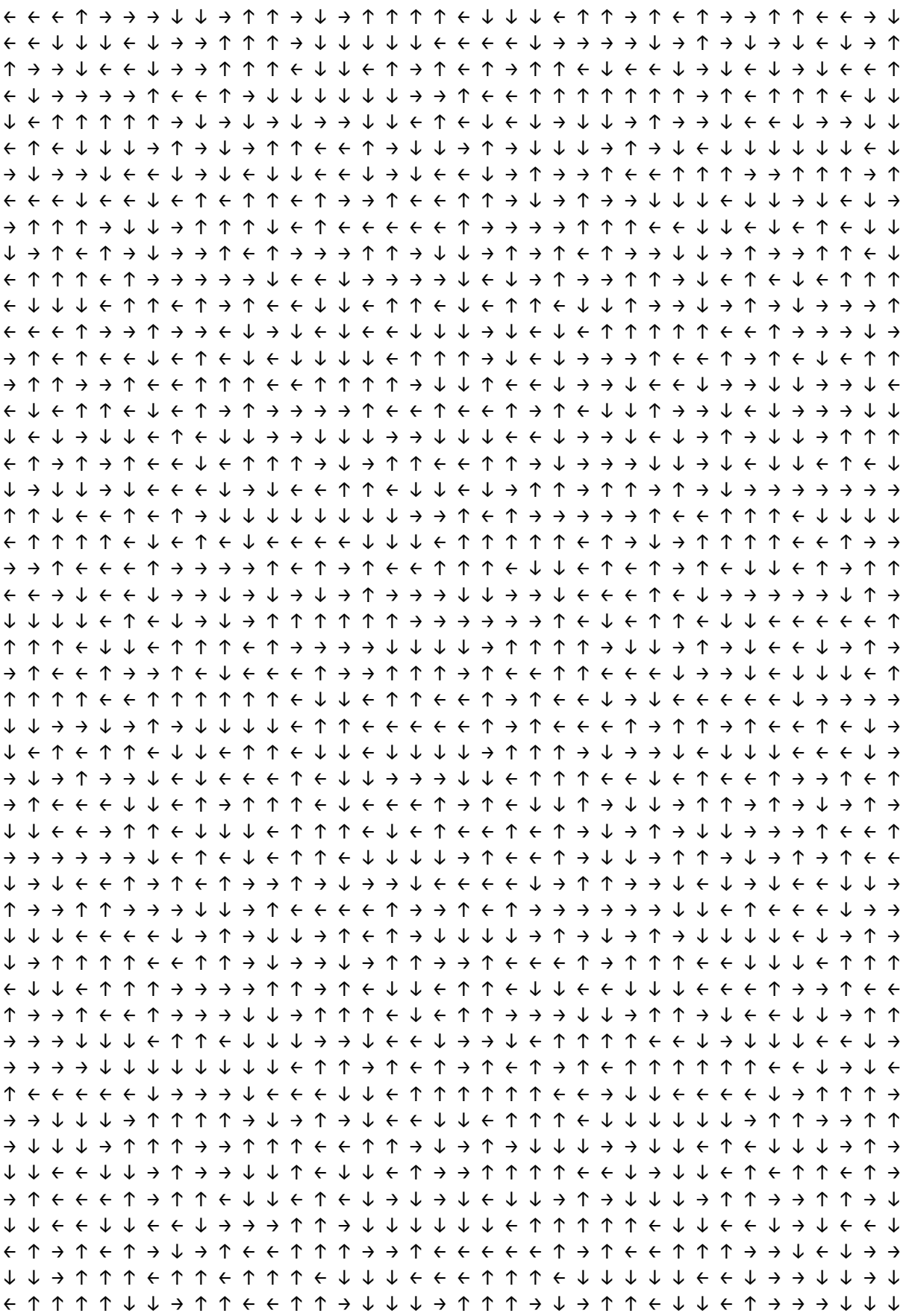
```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe4.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (14384 Schritte). Lösung:
```

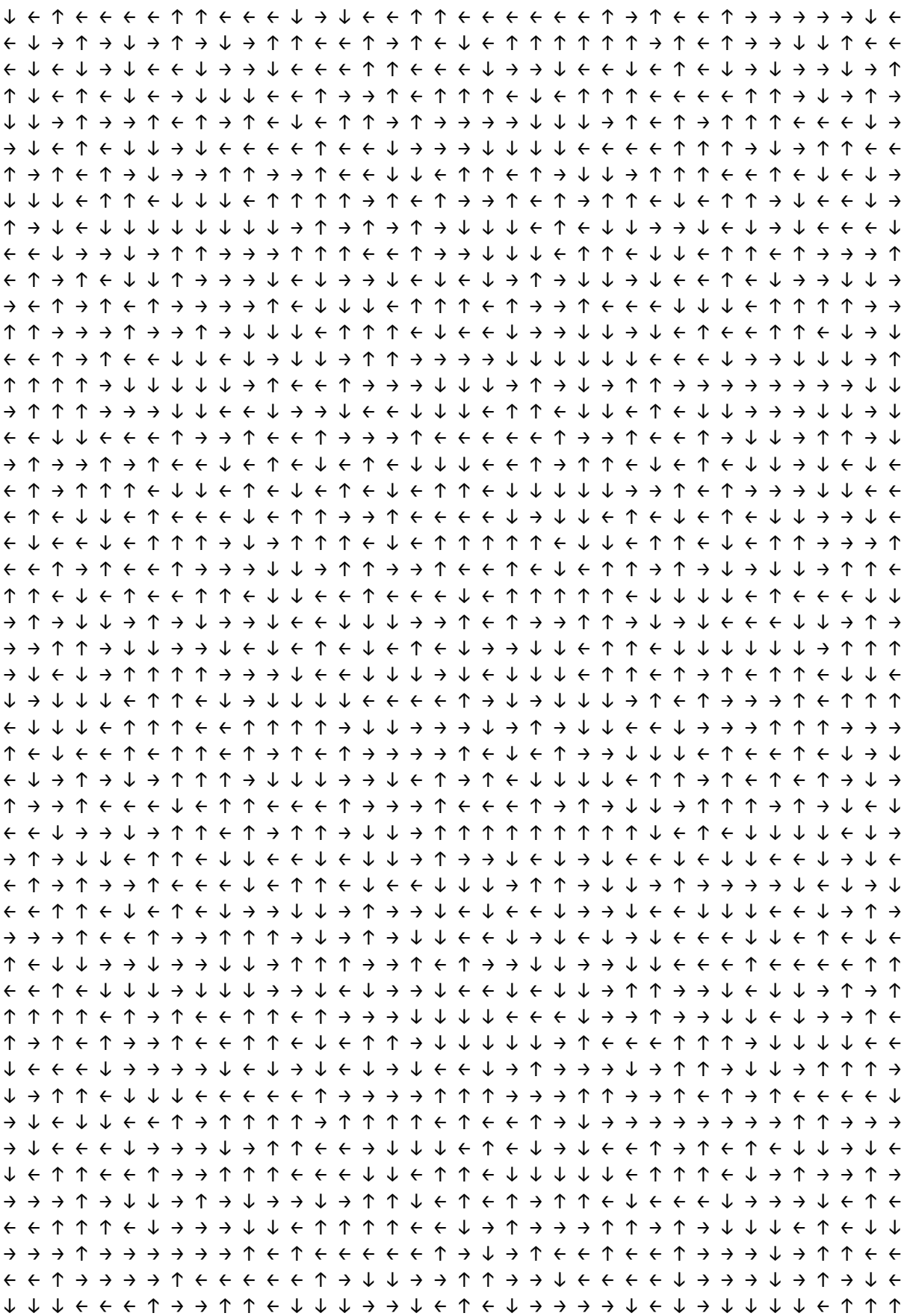
→ → → ↓ ↓ ← ↑ ← ← ↓ → ↓ ← ↓ → ↓ ← ↓ → → ↓ → ↑ ↑ ← ↑ ↑ → ↓ → ↑ → ↓ ↓ ← ↓ ↓ ←
↓ ← ← ← ↑ ← ↓ ↓ ↓ → ↑ → → → ↓ ↓ ↓ → ↑ ↑ → → ↑ ← ← ↑ → → → → ↑ ↑ ↑ ↑ ↑ → ↓
← ↑ ← ↓ ← ← ↓ → ↓ ← ← ↓ → → → → ↓ ← ← ← ↑ → ↑ ↑ ↑ ↑ → → ↑ ← ← ↓ ← ↑ ↑ → → →
→ ↓ ↓ ↓ ↓ → ↑ → → → → → → → ↓ ↓ ← ↑ ← ↑ ↑ ← ← ← ← ↓ ↓ ← ↓ → → ↓ → ↑ ↑ ← ← ↑
→ → → ↓ ↓ → ↑ → ↓ ↓ ↓ ← ← ↓ ← ← ↓ → → → ↑ ← ← ↑ ↑ ↑ → → ↓ ↓ ← ↓ ↓ ↓ → ↑ → →
↑ ← ← ← ↓ ← ↑ ↑ → ↑ → → → → ↓ ↓ ↓ ↓ → ↑ → → ↑ → → ↑ ← ← ← ↓ ← ↑ ↑ → → → → →
→ ↓ ← ↓ ↓ ← ← ↓ ↓ ← ↑ ← ↓ ← ← ↓ ↓ → ↑ → ↓ → ↑ → ↓ ↓ ↓ ↓ ↓ → ↑ → ↑ ← ↑ → → →
→ → ↑ ← ← ← ↑ → ↓ → ↑ ← ↑ → → → → ↓ ← ← ↓ → → ↓ ← ↑ ← ↓ → ↓ ← ↓ → → → ↓ ← ←
↓ ← ← ↑ ↑ ← ↓ ↓ ← ↑ → ↑ ↑ ↑ ← ↓ ↓ ← ↓ ← ← ↓ → ↓ ← ↑ ↑ ← ← ← ← ↑ → → → ↑ ← ←
← ← ↑ ↑ ↑ ↑ ← ↓ ↓ ← ← ↓ → → ↓ ↓ ↓ ← ← ↓ → → → ↓ ← ← ← ← ↑ ↑ ← ↓ ← ↑ ↑ ↑ ↑
↑ ← ↓ ↓ ↓ ← ↑ ← ↓ ← ↑ ↑ ← ↓ → → ↓ ↓ → ↑ ↑ → ↓ ↓ ↓ ← ← ← ↓ → ↓ ← ← ↓ → → ↓ ↓
→ ↑ ↑ → ↑ → ↑ → ↓ → → ↓ ↓ ← ↓ ↓ ← ← ← ↓ ← ↑ ↑ → ↑ ← ← ← ← ← ↑ → → ↑ ↑ ← ↓ ←
↑ ↑ ↑ ↑ ← ↓ ↓ ↓ → → → → ↓ ↓ ↓ ← ↑ ↑ ↑ ↑ ↑ ↑ ← ↓ ↓ ← ↑ ↑ ↑ ← ↓ → → → → ↓ ↓ ↓
← ← → ↑ ← ↑ → → ↑ → → → ↑ ↑ → → → ↑ ← ← ↑ ↑ ↑ ↑ → → ↑ ← ↑ → → ↓ ↓ ↓ ↓ ← ← ↓
→ → → ↑ ↑ → ↓ ↓ ↓ ↓ → ↑ → → ↑ ↑ ← ↓ ← ← ↑ ↑ ↑ ↑ → → → → → → ↓ ← ← ← ↓ ↓ → ↑
→ → → ↑ ↑ → → ↓ ← ↓ → ↑ → → ↑ ← ← ↓ ↓ ← ↑ → ↑ ↑ → ↑ → ↓ → ↑ ↑ → ↓ → ↑ ← ← ↑
↑ ↑ ← ↓ ↓ ↑ → ↓ ↓ → ↑ ↑ → ↓ ↓ ↓ ← ↑ ↑ ↑ ↑ ← ↓ ← ← ↓ ← ↑ ↑ → → → → → → → ↑ ←
← ← ↑ ← ↓ → → ↓ ← ↑ ← ↓ ← ← ↓ ↓ ↓ ↓ → → → ↑ → ↓ ↓ ← ← ↓ ← ↑ ← ↓ ← ↑ ← ↓ ← ↑
← ↓ ↓ ↓ → ↑ → → → → → → ↑ → → ↑ ↑ → → ↑ → ↓ ← ← ↓ ↓ ← ↑ ↑ ← ↓ → ↓ ↓ ↓ ↓ ↓ ↓
↓ ↓ ↓ → ↓ ← ↓ ↓ ↓ → → ↑ ← ↑ ↑ → ↓ → ↑ ↑ ← ← ↑ → → ↑ ← ← ↑ ↑ ↑ ← ← ↑ → → →
↓ → ↑ → ↓ ↓ ↑ ← ↓ ↓ ↓ ↓ ← ↑ ↑ ← ↓ ← ↑ ← ← ↑ ↑ ↑ ↑ → ↓ ↓ ↑ ← ↓ ↓ ↓ → ↑ ↑ →
→ → ↓ → → ↓ ↓ ↓ ↓ → ↑ ↑ → ↑ ← ← ↑ → → → ↓ → ↑ → → → → → ↓ → ↑ ← ↑ ↑ ↑ → ↓ →
↑ → ↓ ↓ → → ↓ ← ← ← ← ↓ → → → ↓ ↓ ↓ → ↓ → ↑ ↑ ← ↑ → ↑ ← ↑ → → ↓ → → ↓ → ↑
↑ → ↓ ↓ ↓ ↓ ← ↑ ← ↓ ↓ ↓ ↓ ← ↑ ↑ ← ↓ ↓ → → ↓ ↓ ← ← ↑ → ↑ ← ↑ ↑ ← ↓ ↓ ↓ ↓ ↓ →
→ → ↑ ↑ ↑ → → → → → ↓ ↓ ↓ ← ↑ → ↑ ↑ ↑ ↑ ← ← ← ← ↑ ← ← ↑ ← ↓ → → → → ← ↑ ↑ ←
← ↑ → → → ↓ → ↑ → ↓ ← ← ← ← ← ↓ → ↑ ↑ → → → ↓ → ↑ → ↓ → ↑ → → → → → → → → →
↓ ← ← ↓ → → ↓ → ↑ ↑ ↑ → ↓ ↓ → → ↓ ↓ ↓ ↓ ← ← ← ↑ ← ↓ ← ← ← ↑ → → ↑ ← ↓ ← ↑ ↑
← ← ← ↓ → → ↓ ← ↓ ↓ ← ↓ ↓ ↓ ← ← ↓ → → ↓ ← ← ← ↑ ↑ ↑ → → ↑ ↑ ← ↑ → ↑ ← ← ↑ ←
↓ ← ↑ ↑ ← ← ↑ → → ↑ → ↓ → ↑ ↑ → → ↑ ← ← ↑ ↑ ← ↓ ← ← ↑ ↑ ↑ ↑ ← ← ↑ ← ↓ ← ← ↓
↓ ↓ ↓ ↓ ↓ → ↑ ↑ ↑ ↑ ← ↑ → → → ↓ ← ← ↓ → → ↓ ↓ → ↑ ↑ → → ↓ ← ↑ ← ← ← ← ↓ → →
↓ ← ↓ ← ↓ → → ↑ → ↓ → → ↓ ← ← ↓ ← ↑ ↑ ← ← ↑ ↑ → ↑ ↑ → ↑ → ↑ ← ↓ ↓ ↓ ↓ ↓ ← ↓
← ↑ ↑ ← ← ← ← ↑ ← ↓ ↓ → → → → → ↓ ← ↓ ↓ ↓ → → ↓ ↓ → ↑ ↑ ← ← ↑ → → → ↓ ↓ → →
↓ ← ← ↓ → ↓ ← ↑ ← ↓ ← ↑ ← ↑ ← ↓ ↓ → ↓ ← ↓ ↓ ↓ → ↑ ↑ → ↓ ↓ ↓ → ↑ → → ↑ ← ↓ ←
↑ ↑ ← ↑ → → → ↓ ↓ → → ↑ ← ↑ ↑ ↑ → → → → ↑ ← ← ↑ ↑ → ↓ → ↑ → → ↓ ← ↓ ↓ ↓ ← ↑
← ← ← ↓ → → ↓ ← ↓ → ↓ → ↓ ↓ ← ↑ ← ↑ ← ↓ ↓ → → ↓ ← ← ← ↑ → ↑ ↑ ← ↓ ← ← ← ← ↓
→ → ↓ ↓ ↓ ↓ ↓ → ↑ → → → → ↑ ↑ ↑ → ↑ ← ↑ → → ↓ ↓ → ↑ ↑ → → ↑ → ↓ → → ↑ ← ↑
← ← ← ↓ ← ← ↑ → → ↑ → ↓ → ↑ ← ← ↑ ↑ ↑ ↑ → → → ↓ → ↑ ↑ ← ← ← ← ↑ ↑ → → → →
→ ↓ ↓ → ↑ ↑ → → → → → → → → → ↓ ← ↓ → → → ↓ ↓ ← ↑ ← ← ← ↑ ↑ ← ↓ → ↓ ← ↓ → ↓
← ← ↑ ↑ ↑ ↑ ← ↓ ↓ ← ↑ ↑ ← ← ↓ → ↓ ← ← ← ↓ → ↓ ↓ → ↑ ↑ → → → ↓ ← ← ↓ → → → →
→ ↓ ← ↓ ↓ ← ↓ → ↑ → → ↑ ← ↑ → → ↓ ↓ ↓ ↓ ↓ → ↑ ↑ → ↑ ← ↓ ← ↑ ↑ ↑ → ↓ → → ↓
→ ↑ ↑ ↑ ↑ ← ← ↑ → → → ↓ → ↑ → → → ↓ → ↑ → ↓ ↓ ↓ ↓ ↓ ← ↓ → ↓ ← ↑ ← ↓ ↓ ↓ →

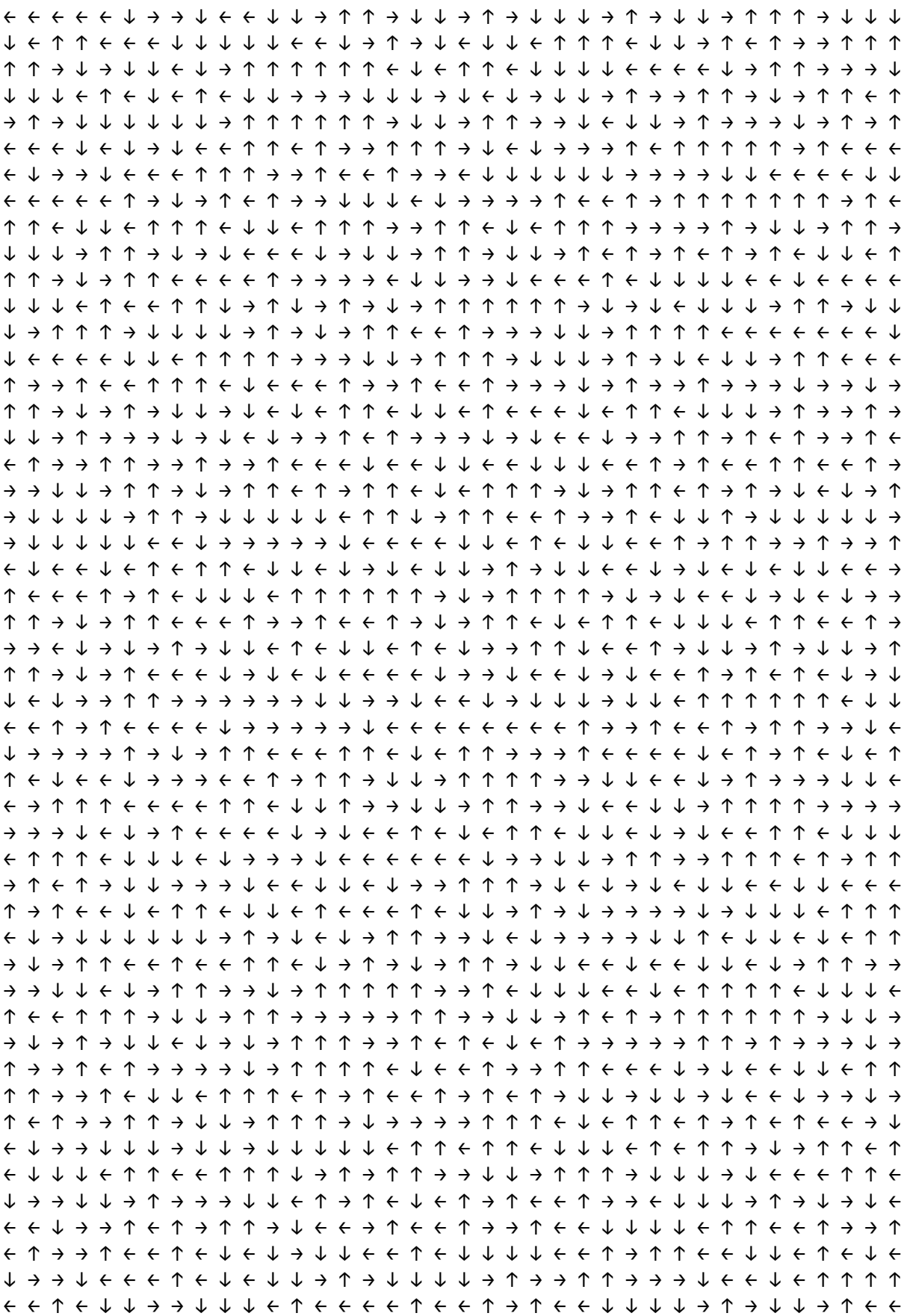


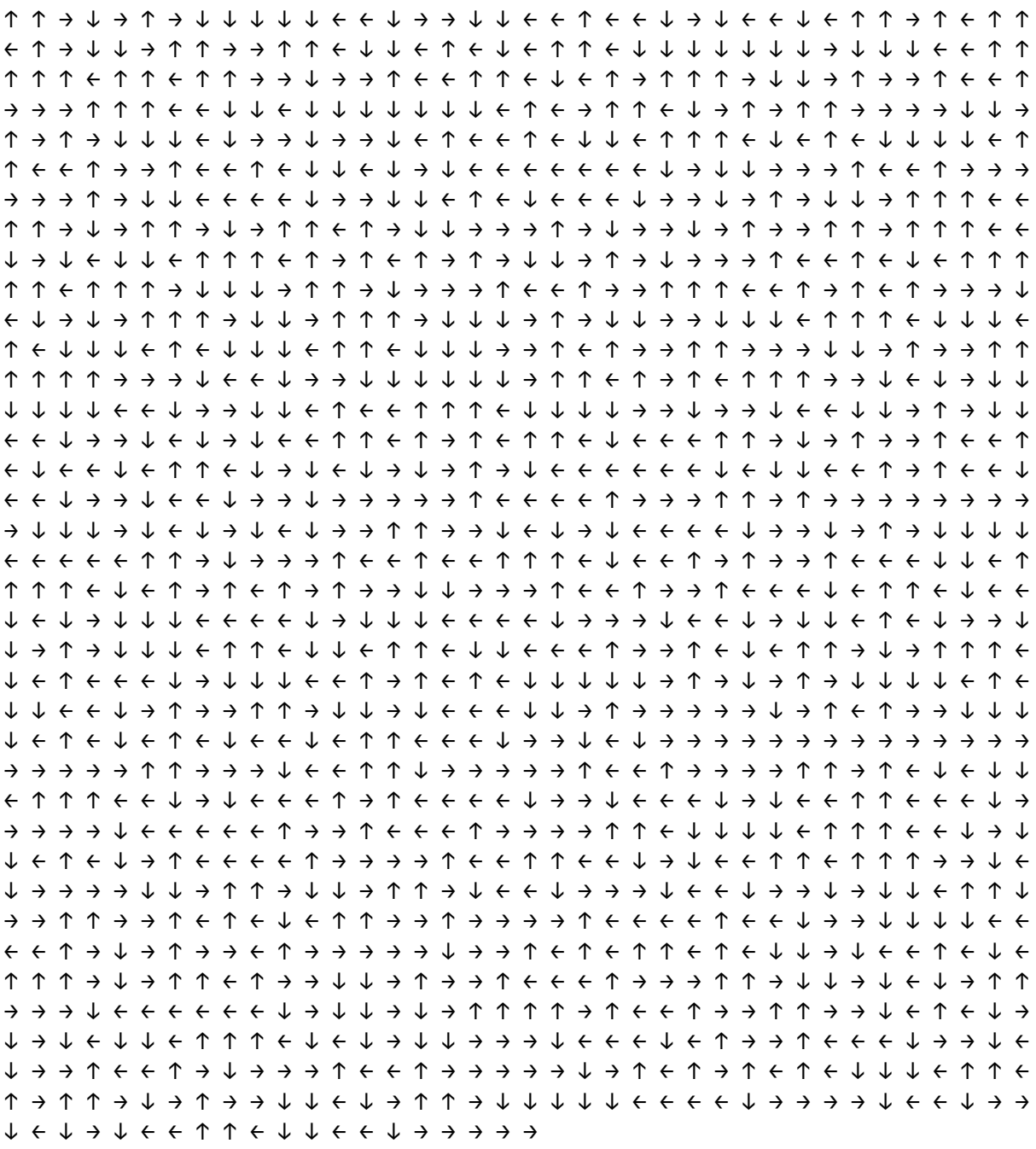




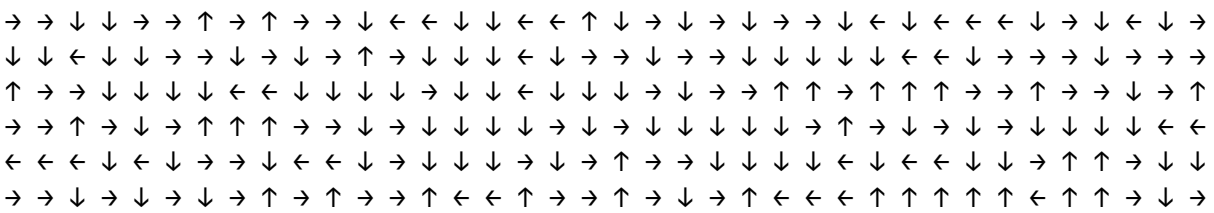


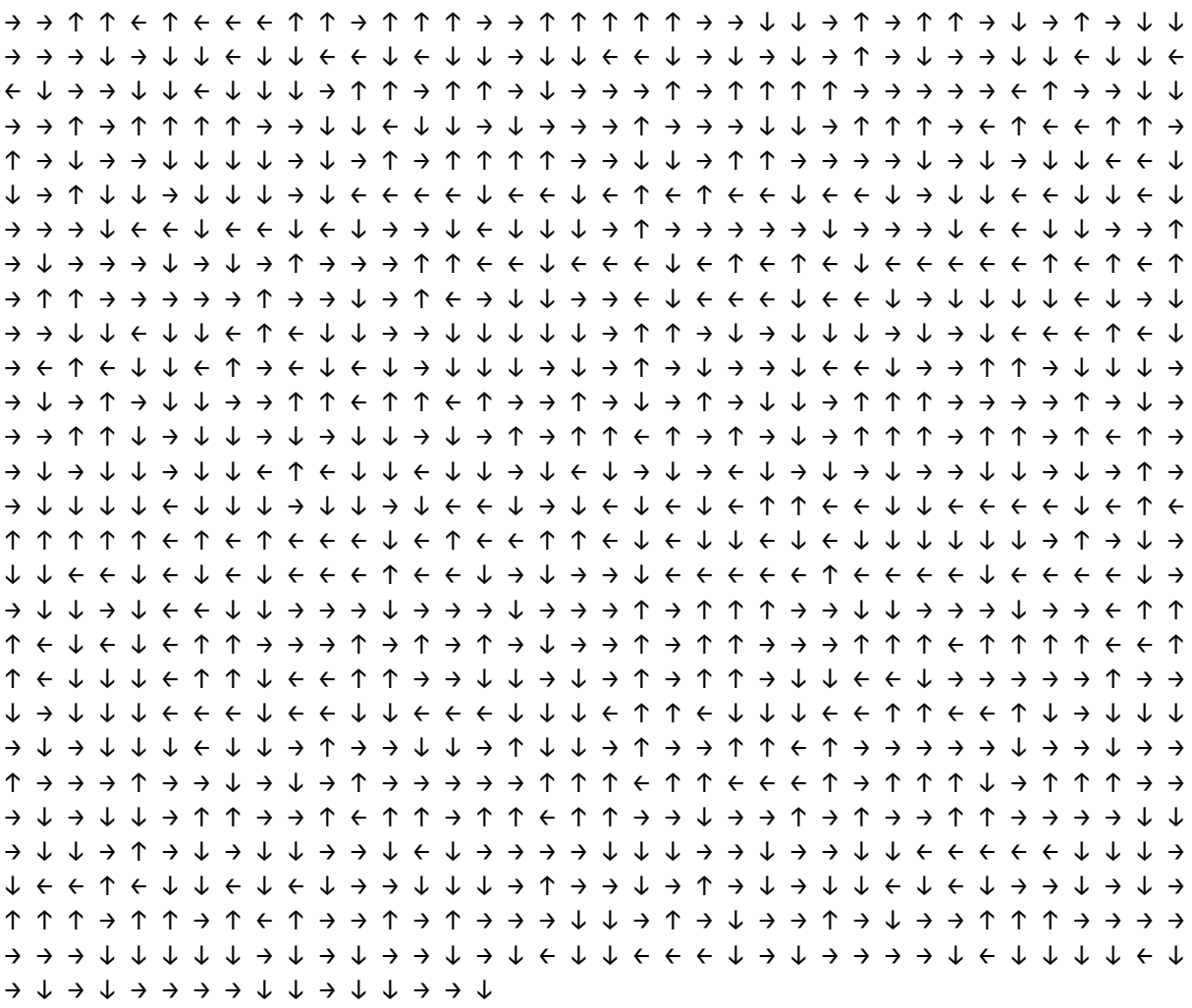




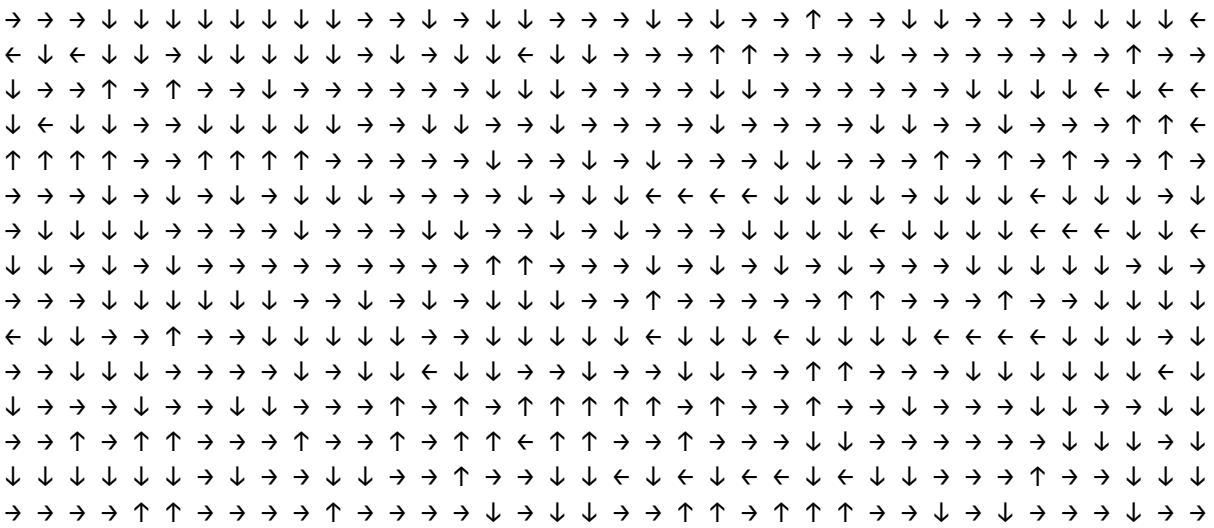


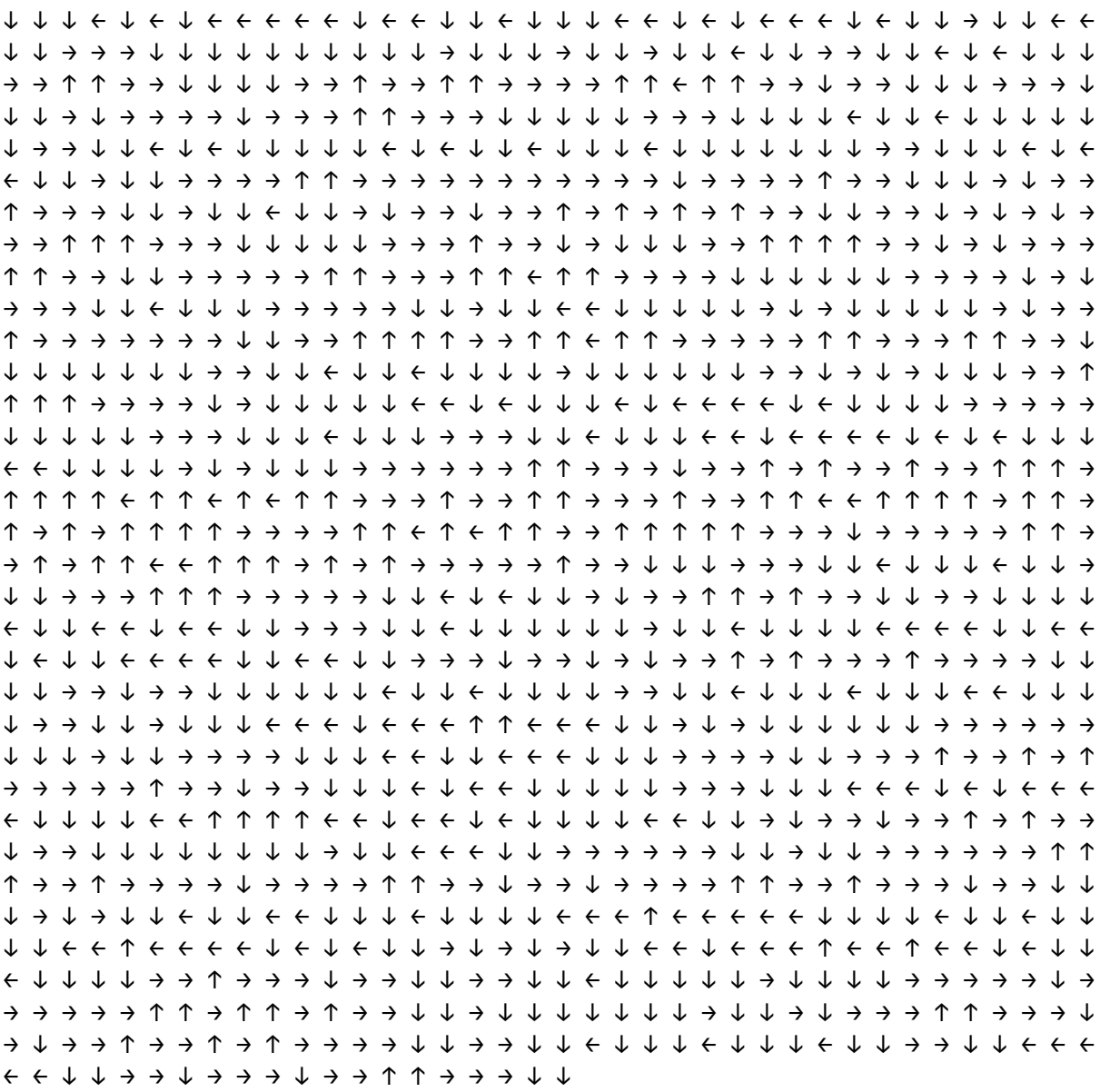
```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe5.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (1308 Schritte). Lösung:
```



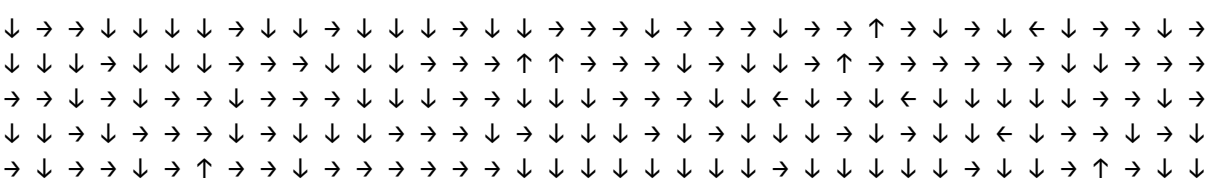


```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe6.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (1844 Schritte). Lösung:
```





```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe7.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Labyrinthpaar nicht lösbar.

--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe8.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (472 Schritte). Lösung:
```



→ ↓ → ↓ ↓ → ↑ → ↓ → → ↓ → ↑ ↑ → ↑ ↑ ← ↓ ↓ ← ↑ ↑ ↑ ← ↑ → ↑ → ↑ → ↓ → ↓ ↓ → →
→ → ↓ ↓ ↓ ↓ → ↓ → → ↓ → ↓ → → ↓ → ↓ → → → ↑ → → → ↓ → → → ↓ → ↓ ↓ ↓ ↓ ↓ → ↓
← ↓ ↓ → → ↓ → → ↓ ↓ ← ← ↓ ↓ → → → → ↑ ↑ → → ↑ ↑ → → → → ↓ → ↓ → ↑ → → → → →
↑ → → ↓ → → → ↓ ↓ ↓ ↓ → ↓ ↓ ↓ ↓ → ↓ ← ← ← ← ← ↓ ↓ ← ← ← ↓ → ↓ ← ← ↓ ↓ ← ↓ ↓
↓ → ↓ ↓ ↓ ↓ → → ↓ → → ↑ ↑ → → → ↑ → ↓ → ↓ ↓ ↓ → → → ↓ → ↑ → → ↓ ↓ → ↓ ↓ ↓ ↓
→ ↓ → → → ↓ → ↓ → → → ↓ ↓ ↓ ↓ ← ← ↑ ← ← ↓ ↓ ↓ ← ← ↓ → ↓ → → ↓ ↓ ← ← ↑ ↓ ← ←
↓ ← ← ← ↓ → ↓ → ↓ ↓ → ↓ → → ↑ → → → → → → → → → → ↓ → ↓ ↓ ↓ → ↓ → → → → ↓ ← ←
↓ → → ↓ ↓ ↓ ↓ ↓ ↓ → ↑ → ↓ ↓ → ↓

```
--- --- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe9.txt --- ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (1012 Schritte). Lösung:
```

↓ → ↓ → → ↓ → → ↑ → → ↓ ↓ ↓ → ↑ → ↑ → → ↓ ↓ ↓ ↓ → → ↑ ↑ → → ↓ → ↓ → ↓ ↓ → ↓
↓ ← ↓ ← ↓ ← ↓ ↓ → ↓ ↓ → ↓ → → → ↑ → ↓ ↓ ↓ ↓ ↓ ← ↓ ← ↓ → ↓ ↓ → ↓ → ↑ → → ↑
→ → ↑ ↑ → ↑ → ↑ ↑ → ↓ → ↓ ↓ → ↓ ↓ → → ↑ → → ↓ ↓ → → ↓ ↓ → → ↓ ← ↑ ↓ ← ← ← ↓
→ ↑ → ↓ → ↓ ← ← ← ↓ → → → ↓ ← ↓ ← ← ↓ → ↓ ← ↓ ← ← ↑ ↑ ← ← ↓ ← ↓ ↓ → ↓ ↓ → ↓
← ↓ ↓ ↓ ↓ → ↓ ↓ → → ↓ ↓ ← ← ↓ ← ↓ ← ↓ ← ← ↑ ← ↓ ↓ ← ↑ ← ↑ ← ← ↓ → ↓ ← ↓ ↓ →
↓ → → ↓ ← ← ↓ ← ↓ → ↓ → ↓ → → ↓ → → ↓ → → ↑ → → → → ↑ → → ↑ ↑ → ← ↑ ← ↑ → ↓
→ → → → ↓ → → → → ↑ ↑ → → ↓ ← ↓ ← ↓ → → ↑ → ↑ ↓ → ↓ ↓ ↓ → ↓ ↓ → ↓ ← ← ← ↓ ←
↓ ← ↓ ↓ ↓ ↓ → ↓ ↓ ← ← ↑ ← ← ↓ ↓ ↓ ↓ ↓ ↓ ↓ ← ↓ ↓ ← ← ↓ ← ↓ → → → → → ↓ → ↓
→ ↓ ← ↑ ↓ → ↑ → ↓ ↓ ← ↓ ← ← ↓ ← ↓ ← ↓ ← ← ← ↓ → ↓ → ↓ → → ↑ → → → ↓ ↓ ↓ → →
↓ → ↓ ↓ ↓ → ↓ → → ↑ ↑ → ↑ → ↑ → ↑ → ↑ ↑ ↑ → → ↑ → → ↓ → ↑ → ← ↓ ← ← ↓ → ↑ →
↑ → → ↑ → ↓ → ↑ ← ↑ ← ← ↑ → ↓ ↓ → → ↓ → → ↓ ↓ → ↑ ↑ → ↑ ↑ → ↑ → ↓ → ↑ → → ↓
→ ↓ ↓ → ↓ → ↓ → → ↓ → → ↓ ← ← ↓ → → ← ↑ ↑ ↑ ← ← ↑ ↓ → → → ↓ ↓ ← ← ← ← ↓ ↓ ↓
↓ → ↓ ← ↓ ← ↓ ↓ → ↓ ← ← ↓ ↓ ↓ → → → ↓ → → → → ↑ ↑ ↑ → → ↓ → → → ↑ ↑ ← ↑ ← ↑
→ ↑ → → ↓ → → → → ↓ → → ↓ ↓ ↓ → ↑ ↑ ↑ → → ↑ ↑ ↑ ← ↑ ↑ ↑ ← ↑ → → → ↑ → ↓ → →
↑ → ↓ ↓ → ↓ ← ↓ → → ↑ ↑ ↑ → → ↑ → → ↑ ↑ ↑ ↑ ← ↑ ← ↑ → ↑ → ↑ ↑ → ↑ → ↓ → →
↓ ↓ → ↓ ↓ → → ↓ → ← ↓ → → → → → ← ↓ ↓ ↓ ← ← ↓ ↓ ← ↓ → ↓ → → ↓ → ↑ → ↓ → → ↓
← ↓ → ↓ → ↑ → → ↓ → ↓ ↓ ↓ → ↑ ← ↓ ↓ ↓ ← ↓ ← ↑ ↑ ← ↓ ↓ ↓ ← ↓ ← ↓ ← ↓ → ↓ → →
→ → ← ↓ → ↓ → ↑ → ↓ → ↓ ↓ → ↑ ↑ → ↑ → ↓ → ↓ → ↓ ← ← ↓ ↓ → ↓ → ↑ → ↓ → → ↓ →
↓ ↓ ↓ → ↑ ↑ ↓ → ↓ ↓ ↓ ↓ ← ← ↓ → ↓ → ↓ → → ↑ → ↑ ↑ ↑ ← ↑ ↑ → → ↓ → ↑ → ↓ → ↑
↓ → ↓ ↓ → → → ↓ ↓ ← ← ↓ ↓ ← ↓ → → ↓ → ↓ ← ↓ ↓ ↓ ← ↓ ↓ ↓ ↓ ← ↓ ← ← ↑ ↑ ← ↓ ↓
← ← ← ↑ ← ↓ ← ↓ ↓ ← ← ↓ ↓ ← ↓ → ↓ → ↓ → ↑ → → → → → ↑ → → ↓ ← → ↓ → ↓ ↓ ↓ ↓
↓ ↓ ← ↓ → ↓ → → ↑ → → ↓ → ↓ ↓ → ↑ → ↑ ↑ → ↓ → → ↑ ← ↑ → → ↓ → ↑ → → → → ↑ ↑
↑ ← ↑ ↑ → ↑ → ↓ → → ↓ ↓ → ↑ → → → → ↓ ↓ ↓ ↓ → ↑ ↑ ↑ ↑ ↑ → ↓ ↓ ↓ ← ↓ ← ↓ ←
↓ ↓ → → → ↓ → ↓ → ↑ → ↑ → ↓ ↓ → ↓ ↓ ↓ ↓ → ↓ → → ↓ → → ↓ ← ↓ ↓ → ↑ → → → ↓ →
→ → ↓ ← ← ↓ → ← ↑ ↑ ← ↑ ↑ ↑ ← ↓ ← ← ↓ ↓ ↓ → ↓ → ↓ ↓ ← ↓ ↓ ↓ → ↓ ← ↓ → ↓ ↓ ↓
↓ ↓ ↓ ↓ → ↓ → ↑ ↑ → ↓ ↓ → ↓ ← ↓ ↓ → → ↑ ↑ ↑ → ↓ → ↓ → ↑ → ↓ → ↓ → ↑ → ↓ → ↑
↑ ↑ → ↑ → ↓ → → ↓ ↓ ↓ ↓ ↓ → → → ↑ → → ↓ ↓ ↓ ↓ ↓

```
--- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe-custom-0.txt ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (3 Schritte). Lösung:
```

→ ↓ →

```
--- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe-custom-1.txt ---
```

```
Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (10 Schritte). Lösung:

↓ ↓ ↓ → ↑ → → ↓ ↓ ↓

--- AUSGABE BEI DATEI ./A2_Simultane Labyrinthe/labyrinthe-custom-2.txt ---

Labyrinthe werden gelöst - dies kann, abhängig von deren Größe, dauern und
viel Arbeitsspeicher beanspruchen.

Optimaler Pfad konnte ermittelt werden (7 Schritte). Lösung:

→ ↓ ↓ ↓ → → →
```