Aufgabe 3: Zauberschule

Team-ID: 00054

Team-Name: K7Bots

Bearbeiter/-innen dieser Aufgabe:
Felix Mölle

7. Oktober 2023

Inhaltsverzeichnis

1 Lösungsidee

Die Lösungsidee besteht darin, den A*-Algorithmus zu verwenden, um den optimalen Pfad von A nach B zu finden.
Der A* Algorithmus ist geeignet da er sehr effizient ist.
Durch die Verwendung der „Luftlinie“ zum Ziel ist er z.B. dem Dijkstra-Algorithmus überlegen
Da z.T. sehr große Ebenen verwendet werden, ist auch das Überprüfen aller möglicher Wege nicht sinnvoll

Dazu wird die gegebene Textdatei zuerst eingelesen und in eine verwendbare Datenstruktur (z.B. 3 Dimensionales Array) überführt

Dabei wird ein Suchbaum erstellt, wobei jeder Knoten einen Zustand repräsentiert, der aus Rons aktueller Position, dem Abstand zum Startpunkt (g), dem kürzest möglichen Abstand zum Ziel (h) und dem Vorgängerknoten besteht.
Der Algorithmus besitzt 2 Listen. Die „openList“ enthält alle zu untersuchenden Knoten und die „closedList“ alle bereits abschließend untersuchten Knoten.
Der Algorithmus bewertet die Knoten basierend auf den Kosten für das Erreichen des aktuellen Knotens und einer geschätzten Kostenfunktion für das Erreichen des Ziels.

Folgendes geschieht solange es noch zu untersuchende Knoten in der „openList“ gibt:

1. Es wird immer der Knoten (x) mit dem geringsten Wert von $f = g + h$ überprüft.
2. Wenn (x) bereits das Ziel ist bricht der Algorithmus ab.
3. (x) wird aus der „openList“ entfernt und als abschließend untersuchter Knoten eingetragen
4. Es werden für alle alle begehbaren Nachbarn (n) von (x) die g und h Werte berechnet
   Dabei gilt $g_n = g_x + dist(x, n); dist(x, n) \in \{1; 3\}$ und $h_n = optimal\_dist(n, target)$
5. Die Nachbarn werden mitsamt ihrer g und h Werte in die „openList“ eingetragen.

Durch die Expansion der Knoten und die Auswahl desjenigen mit der geringsten Gesamtkostenbewertung wird der kürzeste Weg ermittelt. Um die Wände zu berücksichtigen, wird bei der Erstellung des Nachbarknotens geprüft, ob dieser begehbar ist.
Im Generellen wird immer der Übergang in das jeweils andere Stockwerk in Betracht gezogen, wobei diese Wege durch den hohen g Wert tendentiell unattraktiv sind.

2 Umsetzung

Die Umsetzung dieser Aufgabe erfolgte in Rust, einer Programmiersprache, die für ihre Sicherheit, Geschwindigkeit und Nebenläufigkeit bekannt ist. Der Code ist in folgende Module aufgeteilt, um die Lesbarkeit und Wartbarkeit zu verbessern:

- main.rs -> Dieses Modul enthält die main-Funktion, die das Programm startet und die Eingabedaten liest. Es ruft den A*-Algorithmus auf, um den kürzesten Weg zu finden, und gibt die Ergebnisse aus.
- utils.rs -> In diesem Modul werden Hilfsfunktionen und Datenstrukturen definiert, die in verschiedenen Teilen des Programms verwendet werden. Dazu gehören das Einlesen der Eingabedaten, die Konvertierung von Zeichen in Zahlen und Datenstrukturen wie PVector und Node.
- astar.rs -> Hier wird der A*-Algorithmus implementiert, um den optimalen Pfad zu finden. Dieses Modul enthält die Hauptlogik zur Pfadberechnung.

2.1 Datenstrukturen

PVector

Die PVector-Struktur repräsentiert einen 3D-Vektor, der in diesem Kontext Rons Position auf dem Spielfeld darstellt.
Sie speichert die Koordinaten für x, y und das Stockwerk z. Diese Struktur dient als Key in der Nodes Hashpam um eine Node im A*-Algorithmus zu identifizieren.

Node

Die Node-Struktur repräsentiert einen Knoten im Suchbaum des A*-Algorithmus.
Sie enthält Informationen über die Kosten g (bisherige Kosten, um diesen Knoten zu erreichen), die geschätzten Kosten h (geschätzte Kosten, um vom aktuellen Knoten zum Ziel zu gelangen) und einen Verweis auf den Vorgängerknoten (Parent).

2.2 Einlesen der Eingabedaten

Die Eingabedaten werden in `utils.rs` verarbeitet. Die Funktion `read_data()` liest die Dimensionen des Spielfelds, die Positionen von Ron und dem Ziel, sowie die Spielfeldkonfiguration (Wände, Felder usw.) aus einer Datei oder von der Benutzereingabe.

2.3 Konvertierung von Zeichen in Zahlen

Die Funktion `char_to_int()` in `utils.rs` konvertiert Zeichen aus der Spielfeldkonfiguration in entsprechende Zahlenwerte. Dies ermöglicht die einfache Identifikation von Wänden, Feldern, Rons Position und dem Ziel auf dem Spielfeld.

Diese Beschreibung gibt einen umfassenden Überblick über die Umsetzung der Aufgabe. Der Code verwendet Rusts Leistungsfähigkeit und Sicherheit, um den kürzesten Weg unter Berücksichtigung der gegebenen Bedingungen in der Zauberschule Bugwarts zu finden.

2.4 A* Algorithmus

Der A*-Algorithmus wird in astar.rs implementiert und ist der Kern des Programms.

Der Algorithmus arbeitet wie folgt:

- Initialisierung der offenen Liste (known_nodes) und der leeren geschlossenen Liste (finished_nodes) als Vec.
- Initialisierung der Map für die Nodes als HashMap<PVector, Node>
- Startknoten wird zur offenen Liste hinzugefügt. Dieser Knoten enthält Informationen über die aktuelle Position von Ron.

- Während die offene Liste nicht leer ist, wird der Knoten mit den geringsten Gesamtkosten (g + h) ausgewählt.
  `let current = *known_nodes.iter().min_by_key(|p| nodes[p].g + nodes[p].h).unwrap();`
- Wenn der ausgewählte Knoten das Ziel ist, wird der Pfad über die parents zurückverfolgt und die Kosten sowie der gefundene Pfad werden ausgegeben.
- Andernfalls werden die Nachbarknoten des ausgewählten Knotens generiert und deren Kosten berechnet. Wenn ein Nachbarnode bereits in der geschlossenen Liste ist und die Kosten geringer sind, wird er übersprungen. Andernfalls wird er zur offenen Liste hinzugefügt.
- Schritte 4 bis 6 werden wiederholt, bis das Ziel erreicht ist oder festgestellt wird, dass kein Pfad existiert.

3 Beispiele

3.1 Zauberschule 0

```
cost: 8; path: [
PVector { x: 5, y: 9, z: 0 },
PVector { x: 5, y: 9, z: 1 },
PVector { x: 6, y: 9, z: 1 },
PVector { x: 7, y: 9, z: 1 },
PVector { x: 7, y: 9, z: 0 }]
```

3.2 Zauberschule 1

```
cost: 4; path: [
PVector { x: 13, y: 7, z: 0 },
PVector { x: 12, y: 7, z: 0 },
PVector { x: 11, y: 7, z: 0 },
PVector { x: 11, y: 6, z: 0 },
PVector { x: 11, y: 5, z: 0 }]
```

3.3 Zauberschule 2

```
cost: 14; path: [
PVector { x: 21, y: 3, z: 0 },
PVector { x: 22, y: 3, z: 0 },
...
PVector { x: 26, y: 5, z: 0 },
PVector { x: 27, y: 5, z: 0 }]
```

3.4 Zauberschule 3

```
cost: 28; path: [
PVector { x: 3, y: 27, z: 0 },
...
PVector { x: 21, y: 25, z: 0 }]
```

3.5 Zauberschule 4

```
cost: 84; path: [
PVector { x: 73, y: 67, z: 0 },
...
PVector { x: 31, y: 55, z: 0 }]
```

3.6 Zauberschule 5

```
cost: 124; path: [
PVector { x: 13, y: 167, z: 0 },
PVector { x: 13, y: 167, z: 1 },
...
PVector { x: 72, y: 155, z: 0 },
PVector { x: 71, y: 155, z: 0 }]
```

3.7 My_Zauberschule 6

```
cost: 364; path: [
PVector { x: 1, y: 1, z: 0 },
PVector { x: 2, y: 1, z: 0 },
...
PVector { x: 79, y: 198, z: 0 },
PVector { x: 79, y: 199, z: 0 }]
```

4 Quellcode

4.1 main.rs: -> Hauptprogramm

```rust
use crate::utils::read_data;
use std::env;

mod astar;
mod utils;

fn main() {
    let args: Vec<String> = env::args().collect();
    let mut fixed_path = String::new();

    if args.len() >= 2 {
        fixed_path = args[1].clone();
    }
    let (size_x, size_y, floors, start, target) = read_data(fixed_path);
    //println!("start: {:?}; target: {:?}", start, target);

    let (cost, path) = astar::astar(floors, start, target, size_x, size_y);
    println!("cost: {:?}; path: {:?}", cost, path)
}
```

4.2 astar.rs -> Die eigentliche Implementierung des a*-Algorithmus:

```rust
use crate::utils::{Node, PVector};
use std::collections::HashMap;

pub fn astar(
  floors: [Vec<Vec<u8>>; 2], start: PVector, target: PVector, size_x: u32, size_y: u32
) -> (u32, Vec<PVector>) {
  let mut nodes: HashMap<PVector, Node> = HashMap::new();

  let mut known_nodes: Vec<PVector> = vec![];
  let mut finished_nodes: Vec<PVector> = vec![];

  known_nodes.push(start);
  nodes.insert(start, Node::new(0, 0, PVector::zero()));

  while !known_nodes.is_empty() {
      let current = *known_nodes
          .iter()
          .min_by_key(|p| nodes[p].g + nodes[p].h)
          .unwrap();

      if current == target {
          return finalize(start, target, &nodes);
      }

      known_nodes.retain(|&p| p != current);
      finished_nodes.push(current);

      let neighbors = get_neighbors(current, &floors, size_x, size_y);
      check_neighbors(&neighbors, current, &mut nodes, &mut known_nodes, &finished_nodes, target);
  }
```

```
31
    (0, vec![])
33 }

35 fn check_neighbors(
    neighbors: &HashMap<PVector, u32>, current: PVector,
37  nodes: &mut HashMap<PVector, Node>, known_nodes: &mut Vec<PVector>,
    finished_nodes: &[PVector], target: PVector,
39 ) {
    for neighbor in neighbors.keys() {
41      let dist = *neighbors.get(neighbor).unwrap();

43      if finished_nodes.contains(neighbor) && nodes[neighbor].g <= nodes[&current].g + dist {
            continue;
45      }

47      let g = nodes[&current].g + dist;
        let h = neighbor.dist(target);
49
        if !known_nodes.contains(neighbor) {
51          known_nodes.push(*neighbor);
        } else if g >= nodes[&neighbor].g {
53          continue;
        }
55
        nodes.insert(*neighbor, Node::new(g, h, current));
57  }
}
59
fn get_neighbors(current: PVector, floors: &[Vec<Vec<u8>>; 2], size_x: u32, size_y: u32)
61  -> HashMap<PVector, u32> {
    let mut neighbors: HashMap<PVector, u32> = HashMap::new();
63
    if current.x > 0 {
65    neighbors.insert(PVector::new(current.x - 1, current.y, current.z), 1);}
    if current.x < size_x - 1 {
67    neighbors.insert(PVector::new(current.x + 1, current.y, current.z), 1);}
    if current.y > 0 {
69    neighbors.insert(PVector::new(current.x, current.y - 1, current.z), 1);}
    if current.y < size_y - 1 {
71    neighbors.insert(PVector::new(current.x, current.y + 1, current.z), 1);}
    if current.z == 1 {
73    neighbors.insert(PVector::new(current.x, current.y, current.z - 1), 3);}
    if current.z == 0 {
75    neighbors.insert(PVector::new(current.x, current.y, current.z + 1), 3);}

77  return neighbors
        .iter()
79      .filter(|n| is_walkable(n.0, floors))
        .map(|(k, v)| (*k, *v))
81      .collect();
}
83
fn is_walkable(current: &PVector, floors: &[Vec<Vec<u8>>; 2]) -> bool {
85  floors[current.z as usize][current.y as usize][current.x as usize] != 0
}
87
```

```
fn finalize(
    start: PVector, target: PVector, nodes: &HashMap<PVector, Node>,
) -> (u32, Vec<PVector>) {
    let mut path: Vec<PVector> = vec![];
    let mut current = target;
    while current != start {
        path.push(current);
        current = nodes[&current].parent;
    }
    path.push(start);
    path.reverse();
    (nodes[&target].g, path)
}
```

4.3 utils.rs -> FileReading und struct Definitionen

```
use std::collections::HashMap;
use std::io::{BufRead, BufReader};

#[derive(Debug, Clone, Copy, Hash, Eq, PartialEq)]
pub struct PVector {
    pub x: u32,
    pub y: u32,
    pub z: u32,
}

impl PVector {
    pub fn new(x: u32, y: u32, z: u32) -> Self {
        PVector { x, y, z }
    }

    pub fn zero() -> Self {
        PVector { x: 0, y: 0, z: 0 }
    }

    pub fn dist(&self, pos: PVector) -> u32 {
        self.x.abs_diff(pos.x) + self.y.abs_diff(pos.y) + self.z.abs_diff(pos.z) * 3
    }
}

#[derive(Debug)]
pub struct Node {
    pub g: u32,
    pub h: u32,
    pub parent: PVector,
}

impl Node {
    pub fn new(g: u32, h: u32, parent: PVector) -> Self {
        Node { g, h, parent }
    }
}

pub fn read_data(fixed_path: String) -> (u32, u32, [Vec<Vec<u8>>; 2], PVector, PVector) {

    //Stark gekuerzt
```

```
    let mut input = String::new();
    //Entweder fixed_path oder Nutzereingabe als Dateipfad verwenden
    let mut lines = //Datei lesen und Zeilen speichern

    let size_y = //Erste Zahl in erster Zeile
    let size_x = //Zweite Zahl in erster Zeile

    let mut start: PVector = PVector::zero();
    let mut target: PVector = PVector::zero();

    let mut floors: [Vec<Vec<u8>>; 2] = [Vec::new(), Vec::new()];

    //Zeilen einlesen und in das Vec-Array speichern

    (size_x, size_y, floors, start, target)
}

fn char_to_int(ch: char) -> u8 {
  //mapping von '#' auf 0; '.' auf 1; 'A' auf 2 und 'B' auf 3
  //da arbeiten auf int einfacher als auch char
}
```