Aufgabe 5: Hüpfburg

Team-ID: 00428

Team: MingXu

Bearbeiter dieser Aufgabe:
Max Obreiter

1. October 2022

Inhaltsverzeichnis

Lösungsidee....1
Umsetzung....1
Beispiele....2
Quellcode (Rust)....4

Lösungsidee

Zuerst wird für jeden Schritt n geschaut, auf welchen Feldern f(n) Sasha/Mika sein können. Gibt es eine Überschneidung u auf den möglichen Feldern für Sasha und Mika, so können beide nach n Schritten treffen. Gab es für ein Schritt m mit m < n schon dieselbe Konstellation der möglichen Feldern für Sasha/Mika, so können sich diese nie treffen, da die Schritte von m bis n und von n bis n + (n – m) genau gleich sind (Konstellation f(n) führt immer zur selben Konstellation f(n + 1)).
Um danach die Sprungfolgen zu finden, die Sasha/Mika springen müssen, wird dasselbe umgekehrt gemacht; beginnend vom Feld u (Überschneidung) wird geschaut, welche Knoten K zu u führen. Alle Knoten die sowohl in K als auch in f(n – 1) sind, sind als vorletzter Schritt möglich, da es möglich ist, in f(n – 1) Schritten zu diesen Knoten zu kommen. Dies wird bis zu f(0) = Startpunkt wiederholt. Da nur nach einem Weg gefragt ist, wird für jede Überschneidung nur ein Weg gesucht, d. h. ist |K| > 1, so wird nur der erste Knoten aus K als nächstes u verwendet. (Bei der Aufgabe Hüpfburg2 gibt es 2 Überschneidungen – hierfür werden dann 2 Wege ausgegeben.)

Umsetzung

Um f(n) zu speichern, wird ein BitVec von der Bibliothek bit-vec (früher in std) verwendet, welches Vec<bool> bei c++ entspricht. Ein „true“ bedeuted hierbei erreichbar. Hiermit kann man effizient K und f(n – 1) schneiden lassen. Der Graph wird in einem a dimensionalen Array bei a Knoten gespeichert, bei dem jedes Element e ein BitVec ist, dessen Nachfolgerknoten von e „true“ sind, während der Rest „false“ ist. Um f(n + 1) zu bestimmen, werden alle möglichen Nachfolgerknoten von f(n) (für alle Indeces i bei der f(n)[i] = true, Nachfolgerknoten[i]) mit einem BitOr verrechnet. Genauso werden die Sprungfolgen bestimmt, nur diesmal mit den Vorgängerknoten.
Alle f(n) für Sasha und Mika werden in jeweils einer LinkedList rückwärts gespeichert, d. h. der letzte Sprung ist an 0ter Position. Um zu überprüfen, ob eine Konstellation schon gesehen wurde, wird f(n) für Sasha und Mika in einer HashMap gespeichert.

Beispiele

Hüpfburg0:
So kommen Sasha & Mika zum Knoten 10 in 3 Sprünge:
Sasha:
1 -> 18 -> 13 -> 10
Mika:
2 -> 19 -> 20 -> 10

Hüpfburg1:
So kommen Sasha & Mika zum Knoten 4 in 121 Sprünge:
Sasha:
1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8
-> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 4
Mika:
2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 ->
17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 1 -> 2 -> 3 -> 4

Hüpfburg2:
So kommen Sasha & Mika zum Knoten 27 in 8 Sprünge:
Sasha:
1 -> 51 -> 76 -> 59 -> 42 -> 65 -> 54 -> 92 -> 27
Mika:
2 -> 106 -> 136 -> 108 -> 81 -> 12 -> 83 -> 72 -> 27
So kommen Sasha & Mika zum Knoten 51 in 8 Sprünge:
Sasha:
1 -> 51 -> 76 -> 59 -> 110 -> 133 -> 85 -> 8 -> 51
Mika:
2 -> 106 -> 136 -> 108 -> 81 -> 12 -> 114 -> 3 -> 51

Hüpfburg3:
Keine Lösung! (Anzahl der Sprünge bis zu einer bekannten Situation: 11)

Hüpfburg4:
So kommen Sasha & Mika zum Knoten 12 in 16 Sprünge:
Sasha:

1 -> 99 -> 89 -> 79 -> 78 -> 77 -> 76 -> 66 -> 56 -> 55 -> 54 -> 44 -> 43 -> 33 -> 23 -> 13 -> 12
Mika:
2 -> 12 -> 11 -> 100 -> 12 -> 11 -> 100 -> 2 -> 12 -> 11 -> 100 -> 2 -> 12 -> 11 -> 100 -> 2 -> 12

Quellcode (Rust)

```rust
pub fn a5(graph: String) {
  let mut huepfburg: Huepfburg = graph.parse().unwrap();

  // solange es keine Überschneidung gibt und es eine neue Konstellation
gibt,
  // wird f(n + 1) berechnet
  while !huepfburg.gleicher_erreichbarer_knoten() {
    // true wenn diese Konstellation schon bekannt ist
    if huepfburg.versuche_merken() {
      // gibt den Text für keine Lösung aus
      huepfburg.keine_loesung();
      return;
    }
    huepfburg.naechster_sprung();
  }

  // die Indeces der Überschneidungen
  let ueberschneidungen = huepfburg.ueberschneidungen();
  for (end_feld, weg_sasha, weg_mika) in
huepfburg.get_sprungfolgen(&ueberschneidungen) {
    println!(
      "So kommen Sasha & Mika zum Knoten {end_feld} in {} Sprünge:",
      huepfburg.spruenge
    );
    println!("Sasha:\n{weg_sasha}");
    println!("Mika:\n{weg_mika}");
  }
}

fn naechster_sprung(&mut self) {
  // get_neue_erreichbare_knoten berechnet f(n + 1) durch f(n)
  self.sasha_erreichbare_knoten_folge

.push_front(self.get_neue_erreichbare_knoten(self.sasha_erreichbare_knote
n()));
  self.mika_erreichbare_knoten_folge

.push_front(self.get_neue_erreichbare_knoten(self.mika_erreichbare_knoten
()));
```

```
  self.spruenge += 1;
}

fn get_neue_erreichbare_knoten(&self, momentane_knoten: &BitVec) ->
BitVec {
  // keine_knoten gibt ein BitVec mit der Länge a mit a = Anzahl_Knoten
zurück,
  // bei dem jedes Element "false" ist
  let mut neue_erreichbare_knoten = self.keine_knoten();
  momentane_knoten
    .iter()
    .enumerate()
    // nur erreichbare Knoten werden angeschaut
    .filter(|(_knoten, erreichbar)| *erreichbar)
    // BitOr von allen Nachfolgerknoten der erreichbaren Knoten
    .for_each(|(knoten, _erreichbar)| {
      neue_erreichbare_knoten.or(&self.nachfolger_knoten[knoten]);
    });
  neue_erreichbare_knoten
}

fn ueberschneidungen(&self) -> Vec<usize> {
  self.sasha_erreichbare_knoten()
    .bit_and(self.mika_erreichbare_knoten())
    .iter()
    // Indeces bestimmen
    .positions(|b| b)
    .collect_vec()
}

fn get_sprungfolgen(&self, treffpunkte: &[usize]) -> Vec<(usize, String,
String)> {
  // get_sprungfolge bestimmt die Sprungfolge für Sasha/Mika ohne 0-index
  let sasha = self.get_sprungfolge(&self.sasha_erreichbare_knoten_folge,
treffpunkte);
  let mika = self.get_sprungfolge(&self.mika_erreichbare_knoten_folge,
treffpunkte);
  treffpunkte
    .iter()
    // +1, da die Knoten intern 0-indexed sind
    .map(|treffpunkt| treffpunkt + 1)
```

```
    // für jeden Treffpunkt werden die entsprechenden Sprungfolgen als
String formattiert
    .map(|treffpunkt| {
      [&sasha, &mika]
        .iter()
        .map(|kind| {
          kind.iter()
            // die Sprungfolge finden für das Kind finden, die beim
Treffpunkt endet
            .find(|s| s[self.spruenge] == treffpunkt)
            .unwrap()
            .iter()
            // mit einem "->" verbinden
            .join(" -> ")
        })
        .collect_tuple()
        // Rückgabe in Form von (Treffpunkt_Knoten, Sasha_Sprungfolge,
Mika_Sprungfolge)
        .map(|(sasha, mika)| (treffpunkt.to_owned(), sasha, mika))
        .unwrap()
    })
    .collect_vec()
}

fn get_sprungfolge(
  &self,
  erreichbare_knoten_folge: &LinkedList<BitVec>,
  treffpunkte: &[usize],
) -> Vec<Vec<usize>> {
  // für jeden Treffpunkt eine Sprungfolge für Sasha/Mika bestimmen
  treffpunkte
    .iter()
    .map(|erreichbarer_knoten| {
      erreichbare_knoten_folge
        .iter()
        .skip(1)
        .fold(
          // Weg als LinkedList
          LinkedList::from([*erreichbarer_knoten]),
          |weg, zuletzt_erreichbare_knoten| {
```

```
            // für jeden Schritt wird ein Knoten zur Sprungfolge/Weg
hinzugefügt hinzugefügt
            self.finde_weg(weg, zuletzt_erreichbare_knoten)
          },
        )
        .into_iter()
        // Knoten sind nicht 1-indexed
        .map(|n| n + 1)
        .collect_vec()
    })
    .collect_vec()
}

fn finde_weg(
  &self,
  mut weg: LinkedList<usize>,
  zuletzt_erreichbare_knoten: &BitVec,
) -> LinkedList<usize> {
  weg.push_front(
    self.vorgaenger_knoten[*weg.front().unwrap() as usize]
      .iter()
      .zip(zuletzt_erreichbare_knoten.iter())
      // die Vorgänger Knoten des frühesten Knotens m in der aktuellen
Sprungfolge
      // werden mit f(m - 1) durch ein BitAnd verrechnet. Dadurch gibt es
mindestens
      // ein Knoten, der bei beiden auf "true" ist (sonst könnte man
nicht zu f(m) kommen).
      // Der Index des ersten Knotens wird hierbei als früherer Knoten
      // in der Sprungfolge verwendet.
      .position(|(v, m)| v && m)
      .unwrap(), // mindestens 1 Weg
  );
  weg
}
```