Aufgabe 5: Hüpfburg

Team-ID: 00730

Team-Name: SilverBean

Bearbeiter/-innen dieser Aufgabe:
Philip Gilde

19. November 2022

Inhaltsverzeichnis

1 Zusammenfassung

Die Aufgabe wird mit Hilfe einer Breitensuche gelöst, welche für jeden Spielschritt alle Felder, auf denen Mika und Sasha zu diesem sein können, findet und miteinander vergleicht, bis eine Abbruchbedingung erfüllt ist.

2 Lösungsidee

Das Spielfeld mit n Feldern wird als gerichteter Graf mit den Knoten $V = \{1, 2, \ldots, n\}$ und den Kanten $E \subseteq V \times V$ als Menge von 2-Tupeln der Ein- und Ausgangsknoten der Kanten modelliert, in dem jedes Feld für einen Knoten und jeder Pfeil für eine Kante steht.
Die Spieler starten an den Knoten v_1 und v_2. Die möglichen Positionen beim ersten Spielschritt $t = 0$ sind für Spieler 1 $M_{1,0} = \{v_1\}$ und für Spieler 2 $M_{2,0} = \{v_2\}$. Für jeden weiteren Spielschritt werden von allen Positionen des vorherigen Schrittes alle ausgehenden Kanten zum nächsten Knoten traversiert:

$$M_{s,t+1} = \{w | (v, w) \in E | v \in M_{s,t}\}$$

Dieser Prozess wird nun wiederholt, bis sich die Knoten, auf denen die Spieler zu einem Zeitpunkt sein können, überlappen; formal bis $M_{1,t} \cap M_{2,t} \neq \emptyset$. Wenn die beiden Spieler sich allerdings nie treffen können, würde man so unendlich lange wiederholen. Um das zu verhindern, wird weiterhin eine Menge B aller besuchten Knotenpaare, zu denen die beiden Spielern schon gleichzeitig sein könnten geführt:

$$B_0 = \emptyset$$
$$B_{t+1} = B_t \cup (M_{1,t+1} \times M_{2,t+1})$$

Die Wiederholung wird abgebrochen, wenn in einem Schritt keine weiteren Knotenpaare besucht werden: $B_t = B_{t+1}$. Das hat folgenden Grund: Wenn die Spieler sich in einem Schritt t treffen, müssen sie beide

von zwei verschiedenen Knoten dort hin gekommen sein; zu diesem Paar müssen die Spieler auch durch ein Paar unterschiedlicher Knoten gekommen sein, und so weiter. Wenn in einem Schritt allerdings keine neuen solchen Knotenpaare betreten werden, heißt das, dass keine neuen Pfade entdeckt werden können, die zu einem Knoten führen könnten, diese wurden alle schon in vorherigen Schritten betreten und davon ausgehende Pfade erkundet. Wenn die ausgehenden Pfade nämlich nicht vollständig erkundet wurden, würden durch diese neue Knotenpaare hinzukommen. So terminiert das Verfahren in höchstens $|V|^2 - |V|$ Schritten, da es höchstens so viele ungleiche Knotenpaare gibt.
Um, wenn das Spiel lösbar ist, den Weg zu rekonstruieren, wird weiterhin eine Zuordnung $H_t : B_t \to B_{t-1}$ eingeführt, die für erkundete Knotenpaare die jeweiligen Herkunftsknoten speichert. Es kann auch mehrere Herkunftsknotenpaare geben, allerdings wird für jedes Knotenpaar nur ein Herkunftsknotenpaar benötigt, um den Pfad dahin zurückzuverfolgen. Dabei wird immer der kürzeste Pfad zum Knotenpaar verwendet, in dem, sobald einmal ein Pfad gefunden wurde, dieser in späteren Iterationen nicht verändert wird.

$$H_0(1,2) = (-1,-1)$$
$$H_{t+1}(v_1,v_2) = \begin{cases} H_t(v_1,v_2) & wenn (v_1,v_2) \in B_t \\ (w_1,w_2) | (w_1,v_1),(w_2,v_2) \in E \wedge w_1 \in M_{1,t} \wedge w_2 \in M_{2,t} & wenn (v_1,v_2) \in B_{t+1} \end{cases}$$

Wenn sich die Spieler im Schritt t auf dem Knoten v treffen, kann man über $H_t(v,v)$ die Herkunftsknoten der beiden Spieler bestimmen. Für das Herkunftsknotenpaar kann man genauso die Herkunftsknoten bestimmen und so weiter bis die Herkunftsknoten irgendwann (1,2) sind.

Aus den Gleichungen lässt sich mithilfe der Abbruchbedingungen ein Algorithmus formulieren, der die Werte von M, B und H im jeweils nächsten Schritt, ausgehend von den Startwerten solange berechnet, bis eine der Abbruchbedingungen erfüllt ist:

$M_1 \leftarrow \{1\}$
$M_2 \leftarrow \{2\}$
$M_{1l} \leftarrow \emptyset$
$M_{2l} \leftarrow \emptyset$
$B \leftarrow \{(1,2)\}$
$B_l \leftarrow \emptyset$
$H \leftarrow (1,2) \mapsto (-1,-1)$
while $M_1 \cap M_2 = \emptyset \wedge B \neq B_l$ **do**
    $B_l, M_{1l}, M_{2l} \leftarrow B, M_1, M_2$
    $M_1 \leftarrow \{w | (v,w) \in E | v \in M_1\}$
    $M_2 \leftarrow \{w | (v,w) \in E | v \in M_2\}$
    $B \leftarrow B \cup M_1 \times M_2$
    $H \leftarrow (v1,v2) \mapsto \begin{cases} H(v_1,v_2) & wenn (v_1,v_2) \in B_l \\ (w_1,w_2) | (w_1,v_1),(w_2,v_2) \in E \wedge w_1 \in M_1 l \wedge w_2 \in M_2 l & wenn (v_1,v_2) \in B \end{cases}$
end while
if $M_1 \cap M_2 \neq \emptyset$ **then**
    $P \leftarrow [(x,x) | x \in M_1 \cap M_2]$
    while $P[-1] \neq (1,2)$ **do**
        $P \leftarrow P + + [H(P[-1])]$
    end while
    return P
else
    return []
end if

3 Umsetzung

Die Lösung der Aufgabe wurde in Python 3.11 implementiert. Dabei wurde der `set`-Datentyp für die endlichen Mengen und der `dict`-Datentyp für die Zuordnung H verwendet. Die Menge der Kanten wird als Menge von `tupel`s dargestellt. Die Knotenpaar-Tupel werden auch mit dem Datentyp `tupel` dargestellt. Das Programm ist eine direkte Transkription des Pseudocodes aus der Lösungsidee.
Wenn das Programm gestartet wurde, muss der Benutzer den Pfad zur Eingabedatei angeben, dann gibt

das Pogramm entweder aus, wie die Spieler hüpfen müssen um sich zu treffen, oder es gibt aus, dass das Spiel nicht lösbar ist.

4 Beispiele

Die Beispiele stammen von der BwInf-Webseite und sind im Verzeichnis `beispiele` zu finden.
huepfburg0.txt:

```
 Treffen möglich
 Pfad 1:
  1 -> 18 -> 13 -> 10
 Pfad 2:
  2 -> 19 -> 20 -> 10
```

huepfburg1.txt:

```
 Treffen möglich
 Pfad 1:
  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
 Pfad 2:
  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
```

huepfburg2.txt:

```
Treffen möglich
Pfad 1:
  1 -> 51 -> 76 -> 59 -> 110 -> 116 -> 112 -> 95 -> 51
Pfad 2:
  2 -> 106 -> 136 -> 108 -> 100 -> 12 -> 114 -> 3 -> 51
```

huepfburg3.txt:

```
Treffer nicht möglich
```

huepfburg4.txt:

```
 Treffen möglich
 Pfad 1:
  1 -> 99 -> 89 -> 88 -> 78 -> 77 -> 76 -> 66 -> 56 -> 55 -> 54 -> 44 ->
   43 -> 33 -> 23 -> 13 -> 12
```

```
 Pfad 2:
  2 -> 12 -> 11 -> 100 -> 12 -> 11 -> 100 -> 2 -> 12 -> 11 -> 100 -> 2
  -> 12 -> 11 -> 100 -> 2 -> 12
```

5 Quellcode

```python
# Findet Pfade, über die die Spieler sich treffen können
def find_paths(edges):
    pos1 = {1}
    pos2 = {2}
    pos1_last = set()
    pos2_last = set()
    pairs = {(1, 2)}
    pairs_last = set()
    origins = {(1, 2): (-1, -1)}
    while (not pos1 & pos2) and (pairs != pairs_last):
        pairs_last, pos1_last, pos2_last = pairs, pos1, pos2
        pos1 = {w for (v, w) in edges if v in pos1}
        pos2 = {w for (v, w) in edges if v in pos2}
        pairs = pairs | {(a, b) for a in pos1 for b in pos2}
        origins = {
            (v1, v2): (w1, w2)
            for (w1, v1) in edges
            for (w2, v2) in edges
            if w1 in pos1_last and w2 in pos2_last
        } | origins
    if pos1 & pos2:
        path = [(a := (pos1 & pos2).pop(), a)]
        while path[-1] != (1, 2):
            path.append(origins[path[-1]])
        return path[::-1]
    return None


# Graph aus Datei einlesen
edges = set()
with open(input("Pfad: ")) as f:
    while line := f.readline():
        edges.add(tuple(map(int, line.split())))

# verbindende Pfade finden
path = find_paths(edges)
if path:
    print("Treffen möglich")
    print("Pfad 1:")
    print("->".join(" {} ".format(x[0]) for x in path))
    print("Pfad 2:")
    print("->".join(" {} ".format(x[1]) for x in path))
else:
    print("Treffen nicht möglich")
```

huepfburg doc.py