ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger
Florian Meier, Ralph Keusch

HS 2017

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 5

Lösungsvorschlag zu Aufgabe 1

Wir modellieren das Problem mit einem Zustandsgraphen und suchen dann einen kürzesten Weg vom Anfangszustand zum Endzustand. Die Knoten des Graphen entsprechen den möglichen Zuständen, und die Kanten den möglichen Übergängen von einem Zustand in den anderen (was jeweils einer Flussüberquerung entspricht). Die Kanten sind ungerichtet, da jeder Übergang in beide Richtungen möglich ist.

Es sei $S = \{M, W, Z, K\}$, wobei M, W, Z und K jeweils den Mann, den Wolf, die Ziege und den Kohl bezeichnen. Jeden Zustand identifizieren wir mit einer Teilmenge X von S, wobei X die Menge der auf der Ausgangsseite Anwesenden beschreibt. Die Anzahl der Zustände ist deshalb gleich der Kardinalität der Potenzmenge von S, d.h. gleich $2^4 = 16$.

Einen Zustand nennen wir *zulässig*, falls weder der Wolf und die Ziege noch die Ziege und der Kohl unbewacht auf einer Seite allein sind. Man sieht sofort, dass genau die 6 Zustände $\{W, Z, K\}, \{M\}, \{W, Z\}, \{M, K\}, \{Z, K\}, \{M, W\}$ nicht zulässig sind. Die verbleibenden 10 Zustände sind in einem bipartiten Graph angeordnet, da der Mann in jedem Schritt die Seite wechselt. Der resultierende Zustandsgraph G ist in Abb. 1 gezeigt.

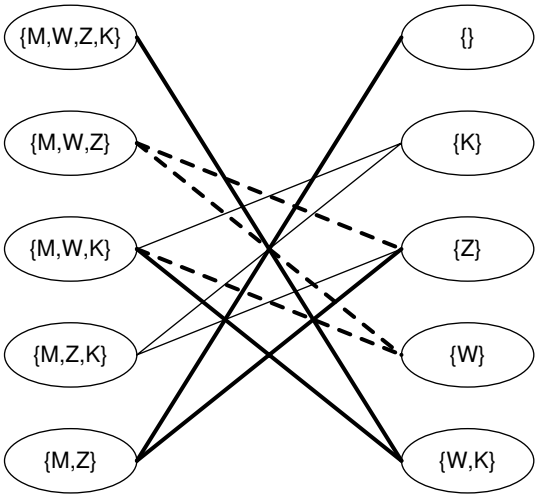

Abbildung 1: Der Zustandsgraph zum Wolf-Ziege-Kohl-Problem. Die fetten (durchgezogenen und gestrichelten) Kanten bilden einen kürzesten Weg vom Anfangs- zum Endzustand

Man sieht leicht, dass ein kürzester Weg in G vom Anfangszustand $\{M, W, Z, K\}$ zum Endzustand $\{\}$ die Länge 7 besitzt. Man sieht auch schön, dass das Problem symmetrisch bezüglich Vertauschung von Wolf und Kohl ist: Wenn die gestrichelten Kanten durch die nicht benutzten Kanten ersetzt werden, ergibt sich wieder ein kürzester Pfad.

Lösungsvorschlag zu Aufgabe 2

(a) Da genau $n - i$ Elemente in A grösser sind als a_i, ist a_i das k-grösste Element aller $2n$ Zahlen genau dann wenn genau $k - (n - i) - 1$ Elemente in B grösser sind wie a_i. Diese Aussage können wir leicht testen: Sei $z := k - (n - i) - 1$. Wenn $z < 0$ oder $z > n$, geben wir 'NEIN' aus. Ansonsten testen wir ob $b_{n-z+1} \geq a_i \geq b_{n-z}$ gilt. Falls Ja geben wir 'JA' aus ansonsten 'NEIN'. Beachte, dass wir $b_0 = -\infty$ und $b_{n+1} = +\infty$ definieren um sicher zu gehen, dass nie auf nicht existierende Elemente im Array zugegriffen wird. Alternativ könnte man die Fälle $z = 0$ und $z = n$ seperat betrachten. Korrektheit folgt aus obiger Aussage und die Laufzeit ist offensichtlich $O(1)$.

(b) Sei y das k-grösste Element von A und B zusammen. Wir nehmen zunächst an, dass $y \in A$. Sei $1 \leq i \leq n$ beliebig. Wie zuvor setzen wir $z = k - (n - i) - 1$ und unterscheiden folgende Fälle:

- Falls $z < 0$, so folgt $n - i = k - 1 - z > k - 1$. Dann sind mindestens $k - 1$ Elemente von A grösser als a_i, was wiederum $y \in [a_i, \ldots, a_n]$ impliziert.
- Falls $z > n$, so folgt $(n - i) + n = k - 1 - z + n < k - 1$. Dann sind insgesamt weniger als $k - 1$ Elemente grösser als a_i, und wir haben $y \in [a_1, \ldots, a_{i-1}]$.
- Falls $0 \leq z \leq n$ und $a_i < b_{n-z+1}$, so sind mindestens z Elemente in B grösser als a_i. Deshalb sind insgesamt mindestens $n - i + z = k - 1$ Elemente grösser als a_i und es folgt $y \in [a_i, \ldots, a_n]$.
- Gilt schliesslich $0 \leq z \leq n$ und $a_i > b_{n-z+1}$, so sind höchstens $z - 1$ Elemente in B grösser als a_i. Deshalb sind insgesamt höchstens $z - 1 + n - i = k - 2$ Elemente grösser als a_i, also haben wir $y \in [a_1, \ldots, a_{i-1}]$.

Mithilfe dieser Eigenschaft können wir die Binäre Suche modifizieren, so dass A nach y durchsucht wird (Siehe Pseudocode). Anschliessend kann B mit demselben Algorithmus durchsucht werden. In einem der beiden Arrays ist die k-grösste Zahl enthalten, also finden wir sie entweder bei ersten oder zweiten Aufruf der modifizierten Binären Suche.

Die Korrektheit folgt aus obigen Überlegungen.

Der Algorithmus wird einmal für A und einmal für B ausgeführt, was eine Gesamtlaufzeit von $O(\log n)$ ergibt (die Laufzeitanalyse ist analog zur Analyse der Binären Suche).

(c) Wir testen für jedes $a \in A$, ob es ein $b \in B$ gibt mit $a + b = x$. Dafür rufen wir jeweils BINÄRESUCHE$(B, 1, n, x - a)$ auf. Offensichtlich werden wir so das gesuchte Paar finden, falls es existiert. Da die BINÄRESUCHE n mal aufgerufen wird, ergibt sich eine Laufzeit von $O(n \log n)$.

(d) Sei zunächst $\ell = 1$ und $k = n$. Wir betrachten jeweils das Paar (a_ℓ, b_k). Falls $a_\ell + b_k = x$ so haben wir das gesuchte Paar gefunden. Falls $a_\ell + b_k > x$ so verringern wir k um 1 und falls $a_\ell + b_k < x$ erhöhen wir ℓ um 1. Anschliessend iterieren wir diese Prozedur für die neuen Werte von ℓ und k so lange bis wir entweder ein Paar gefunden haben oder bis $\ell > n$ oder $k < 1$.

Korrektheitsbeweis: Der Algorithmus terminiert: Solange das Paar noch nicht gefunden wurde, wird in jedem Schritt der while-Schleife entweder k um 1 verringert oder ℓ um 1 erhöht. Falls keine a, b mit $a + b = x$ existieren, dann ist klar, dass der Algorithmus 'das gesuchte Paar existiert nicht' ausgibt. Zu zeigen bleibt, dass wir tatsächlich ein Paar ℓ', k' mit $a_{\ell'} + b_{k'} = x$ finden, falls es solche Paare gibt. Wir beoabchten, dass in jedem Schritt der while-Schleife entweder ℓ um 1 erhöht oder k um 1 verringert wird. Betrachte den ersten Zeitpunkt, wo wir auf ein k oder ein ℓ treffen, das in einem solchen gewünschten Paar k', ℓ' enthalten ist. Zu einem solchen Zeit haben wir also $\ell = \ell'$ oder $k = k'$. Nehmen wir an dass $\ell = \ell'$. Es folgt, dass zu diesem Zeitpunkt $k \geq k'$ gilt. Für $k > k'$ gilt $a_\ell + b_{k'} > a_\ell + b_k = x$. Deshalb wird k solange verringert, bis $k = k'$ gilt und somit das gesuchte Paar gefunden ist. Der Fall $k = k'$ funktioniert analog.

Algorithm 1 FIND-K-LARGEST-ELEMENT (A, B, ℓ, r, k)

Eingabe: A,B sortierte Arrays, ℓ, r und k
Ausgabe: k-grösstes Element x, falls $x \in A$

```
b_0 ← −∞; b_{n+1} ← ∞;
if ℓ = r then
  Teste mit Alg. aus Teilaufgabe (a), ob a_ℓ das k-grösste Element ist;
  if a_ℓ ist k-grösstes Element then
    return a_ℓ
  else
    return „k-grösstes Element nicht in A“
  end if
else
  m ← ⌈(ℓ+r)/2⌉;
  z ← k − 1 − (n − m);
  if z < 0 or (z ≤ n and a_m < b_{n−z+1}) then
    return FIND-K-LARGEST-ELEMENT(A, B, m, r, k)
  else
    return FIND-K-LARGEST-ELEMENT(A, B, ℓ, m − 1, k)
  end if
end if
```

Algorithm 2 FIND-PAIR (A, B, x)

Eingabe: A,B sortierte Arrays, x
Ausgabe: $a \in A$, $b \in B$ mit $a + b = x$, falls so ein Paar existiert

```
while ℓ ≤ n and k ≥ 1 do
  ℓ ← 1; k ← n;
  if a_ℓ + b_k > x then
    k ← k − 1;
  else if a_ℓ + b_k < x then
    ℓ ← ℓ + 1;
  else
    return (ℓ, k)
  end if
end while
return „das gesuchte Paar existiert nicht“
```

Laufzeitanalyse: Die while-Schleife wird maximal $2n - 1$ mal ausgeführt. Eine Iteration hat konstanten Zeitaufwand. Somit folgt, dass die Laufzeit $O(n)$ beträgt.

Lösungsvorschlag zu Aufgabe 3

(a) Wir beweisen die Aussage mittels Induktion.

Induktionsannahme: Für $2 \leq n \leq |V|$ gibt es in jedem Turnier einen gerichteten Pfad über n Knoten.

Für $n = 2$ können wir eine beliebige gerichtete Kante betrachten.

Wir nehmen nun an, dass die Behauptung für n erfüllt ist und zeigen, dass sie dann auch für $n + 1 \leq |V|$ erfüllt sein muss. Sei $P := v_i, \ldots, v_n$ ein Pfad über n Knoten. Weil $n + 1 \leq |V|$ gibt es einen Knoten v der nicht auf dem Pfad liegt. Wir betrachten nun drei verschiedene Fälle:

(a) $(v_n, v) \in A$. Durch anhängen von v an den bestehenden Pfad erhalten wir einen Pfad über n Knoten.

(b) $(v, v_1) \in A$. $v, v_1, \ldots, v_n$ ist ein Pfad über n Knoten.

(c) $(v, v_n) \in A$ und $(v_1, v) \in A$. Es muss ein $1 \leq i < n$ geben für welches gilt, dass $(v_i, v) \in A$ und $v, v_{i+1} \in A$. Folglich ist der Pfad $v_1, \ldots, v_i, v, v_{i+1}, \ldots, v_n$ ein Pfad über n Knoten.

Offensichtlich ist immer einer dieser drei Fälle erfüllt.

(b) Wir verwenden das induktive Argument aus a) um einen Algorithmus zu entwerfen welcher einen bestehenden Pfad in jeder Runde um einen beliebigen Knoten erweitert. Die Korrektheit des Algorithmus folgt aus a).

Ist der Fall 1. oder 2. erfüllt so kann der Pfad mit konstant vielen Operationen erweitert werden. Im dritten fall muss über alle möglichen $i \leq n$ iteriert werden, was in $O(n)$ Operationen erledigt wird. Folglich betragen die Kosten für das hinzufügen eines Knotens v zu einem Pfad der Länge n immer $O(n)$ Operationen . Daraus folgt ein Gesamtaufwand von $O(\sum_{i=2}^{|V|} i) = P(|V|^2)$.

(c) Wir verwenden den selben Ansatzwie in b). Nur wollen wir diesmal nicht alle $1 \leq i \leq n-1$ überprüfen. Stattdessen führen wir jedesmal ExtPath aus, welches einen bestehenden Pfad $v_1, \ldots, v_n$ mittels binärer Suche in $O(\log n)$ um einen beliebigen Knoten $v \notin \{v_1, \ldots, v_n\}$ erweitert.

Es genüg Korrektheit und Laufzeit $O(\log n)$ für ExtPath$(n, v, v_1, \ldots, v_{n-1}, A)$ zu zeigen.

Korrektheit: Falls $a_{v,v_1} = 1$ oder $a_{v_n,v} = 1$ gilt, ist die Korrektheit des zurückgegebenen Pfades offensichtlich. Wir betrachten also die *While*-Schlaufe. Zu beginn jeder Iteration werden ℓ und r so gesetzt, dass die Invariante $(v_\ell, v) \in A$ und $(v, v_r) \in A$ gilt. In jeder Iteration verringert sich $r - \ell$ und deshalb muss irgendwann $\ell = r - 1$ gelte. Zu diesem Zeitpunkt wird die Schlaufe verlassen und die Invariante garantiert die Korrektheit des zurückgegebenen Pfades.

Laufzeit: Weil wir in jeder Iteration $r - \ell$ um die Hälfte verringern wird die *While*-schlaufe nach spätestens $O(\log n)$ Iterationen verlassen. Wir könenn folglich jeden Pfad über n Knoten in $O(\log n)$ Operationen erweitern und haben eine Gesamtlaufzeit von $O(|V| \log |V|)$.

Algorithm 3 EXTPATH$(n, v_1, \ldots, v_n, v, A)$

```
if a_{v,v_1} = 1 then
  return v, v_1, ..., v_n
else if a_{v_n,v} = 1 then
  return v_1, ..., v_n, v
else
  ℓ := 1
  r := n
  while ℓ < r − 1 do
    m := ⌈ℓ + (r − ℓ)/2⌉
    if a_{v,v_m} = 1 then
      r := m
    else
      ℓ := m
    end if
  end while
  return v_1, ..., v_ℓ, v, v_r, ..., v_n
end if
```

(d) Wir beweisen die Aussage mittels Widerspruch. Aus der Vorlesung wissen wir, dass jeder Vergleichsbasierte Algorithmus im schlimmsten Fall $\Omega(n \log n)$ Vergleiche benötigt um eine Sequenz $x_1, \ldots, x_n$ von n Zahlen zu sortieren. Wir nehmen nun an, dass es einen Algorithmus gibt, welcher in jedem Turnier einen Pfad über $|V|$ Knoten findet und dazu nur auf $O(|V|)$ Elemente von A zugreifen muss.

Für eine beliebige Sequenz von Zahlen $x_1, \ldots, x_n$ können wir nun ein Turnier mit n Knoten konstruieren in welchem für $1 \leq i \neq j \leq n$ gilt das $(v_i, v_j) \in A$ genau dann wenn $x_i < x_j$. Wir beobachten nun, dass das Auslesen eines Eintrages der Adjazenzmatrix A genau einem Vergleich von zwei Zahlen entspricht. Unser Algorithmus könnte also mit nur $O(n)$ Vergleichen unsere Sequenz sortieren, was aber ausgeschlossen ist.