Aufgabe 1

(a)

v	$r[v]$	$p[v]$	$d[v]$	$f[v]$	$a[v]$	$n[v]$
a	1	10	1	20	1	10
b	2	3	2	7	2	3
c	5	7	8	15	2	4
d	9	9	16	19	2	2
e	3	1	3	4	3	1
f	4	2	5	6	3	1
g	6	5	9	12	3	2
h	8	6	13	14	3	1
i	10	8	17	18	3	1
j	7	4	10	11	4	1

(b) $f[v] = d[v] + 2n[v] - 1$
$d[v] = 2r[v] - a[v]$
$f[v] = 2(r[v] + n[v]) - a[v] - 1$

(c) Betrachtet man die Funktion $A(u, v)$, die 1 ist, wenn u ein Ahne von v ist und 0 sonst, so gilt:

Die durchschnittliche Anzahl der Ahnen ist die Summe der Anzahlen der Ahnen über alle Knoten geteilt durch die Anzahl der Knoten:

$$\begin{aligned}&\frac{1}{|V|}\sum_{v\in V} \#\text{Ahnen von v}\\ =&\frac{1}{|V|}\sum_{v\in V}\sum_{u\in V} A(u,v)\end{aligned}$$

Ist u ein Ahne von v, so ist v ein Nachkomme von u.

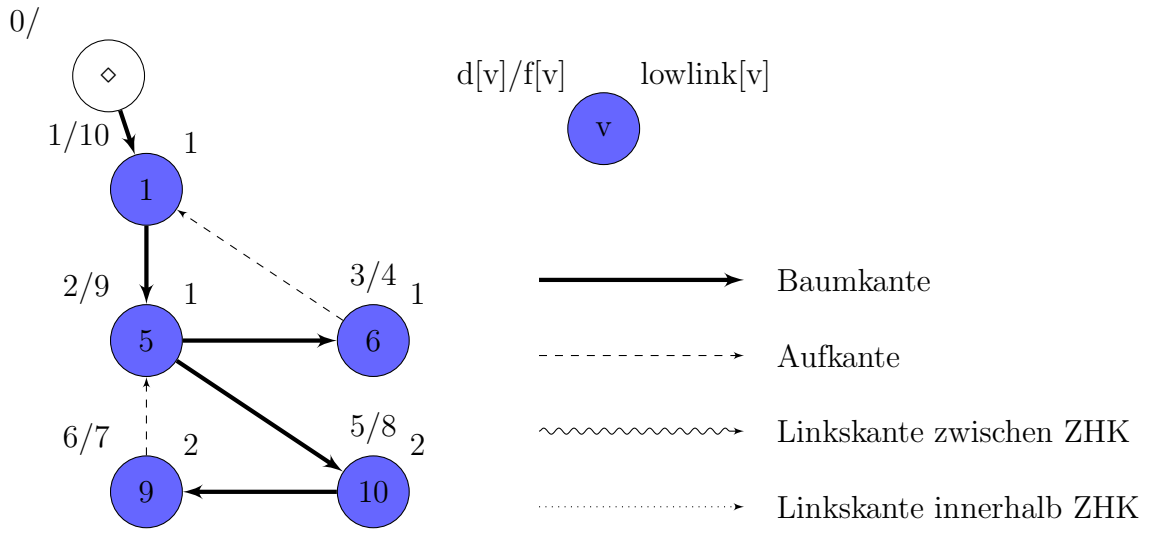
Die durchschnittliche Anzahl der Nachkommen ist die Summe der Anzahlen der Nachkommen über alle Knoten geteilt durch die Anzahl der Knoten:

$$\begin{aligned}&\frac{1}{|V|}\sum_{u\in V} \#\text{Nachkommen von u}\\ =&\frac{1}{|V|}\sum_{u\in V}\sum_{v\in V} A(u,v)\\ =&\frac{1}{|V|}\sum_{v\in V}\sum_{u\in V} A(u,v)\end{aligned}$$

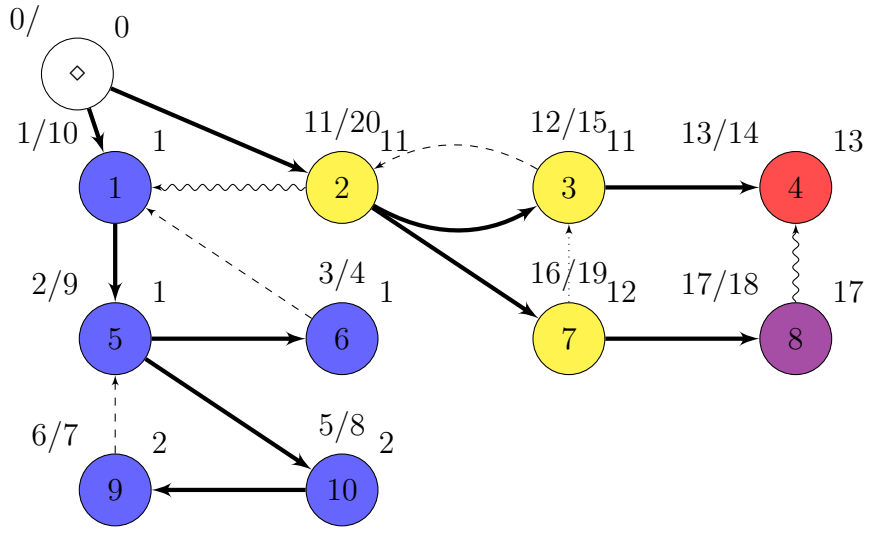
Jedes Ahne-Nachkomme-Paar geht genau einmal in diese Summe ein.

Aufgabe 2

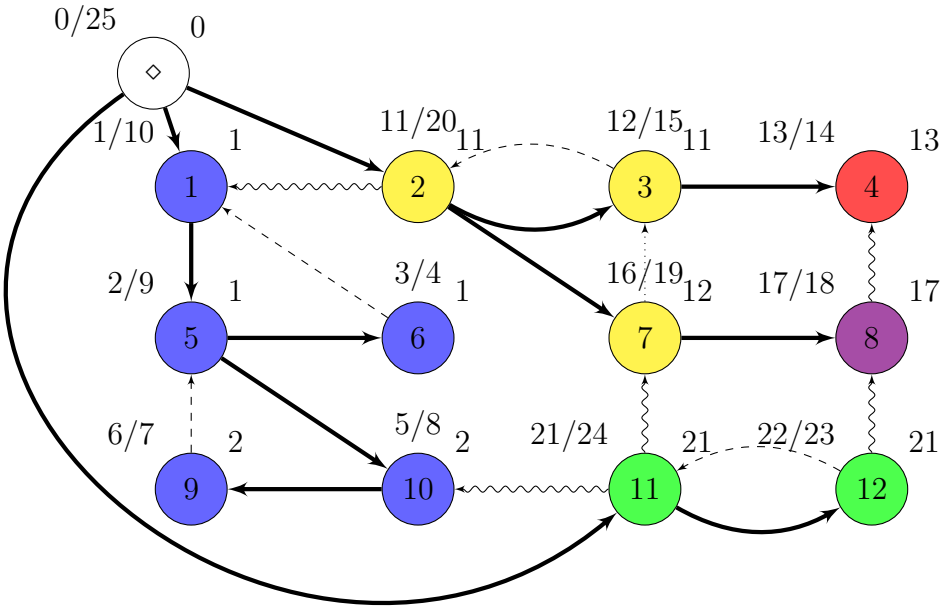
Nachdem alle Nachkommen von 1 besucht wurden hat sich der lowlink Wert 1 von 6 zu 5 propagiert und der lowlink Wert 2 von 9 zu 10. 1 wird als Tor erkannt (lowlink[1]=d[1]) und alle Nachkommen von 1 werden als Komponente 10 markiert.



Die Suche wird an 2 fortgesetzt, 4 und 8 werden als Tore erkannt und als Komponente 14 und 18 markiert, bevor 2 als Tor erkannt und die Knoten 2,3 und 7 als Komponente 20 markiert werden.



Danach wird 11 als Tor erkannt und die Knoten 11 und 12 als Komponente 24 markiert. $\diamond$ bildet schließlich eine Komponente 25, die nicht zum ursprünglichen Graphen gehört.



Aufgabe 3

Im wesentlichen führen wir eine Tiefensuche auf dem transponierten Graphen aus, die von den roten Knoten ausgeht.

```
procedure visit(Graph G, Knoten v, Knoten ρ)
      r[v] ← ρ
      for jede Kante [u, v⟩ in G, die nach v führt do
            if r[u] = ⊥ then
                  visit(u, ρ)

function red_search(Graph G, Menge R)
      initialisere r[] mit ⊥
      for every ρ ∈ R do
            if r[ρ] = ⊥ then
                  visit(G, ρ, ρ)
      return r
```

Für jeden roten Knoten besuchen wir alle Knoten die diesen roten Knoten erreichen können und für die noch kein erreichbarer roter Knoten gefunden wurde. Dabei gilt die Invariante, dass nachdem ein Knoten besucht wurde, alle Knoten die diesen Knoten erreichen können bereits besucht wurden.

Da jeder Knoten höchstens einmal besucht und jede Kante höchstens einmal betrachtet wird ist die Laufzeit in $\mathcal{O}(|V| + |E|)$.