Zweite Runde Theorie

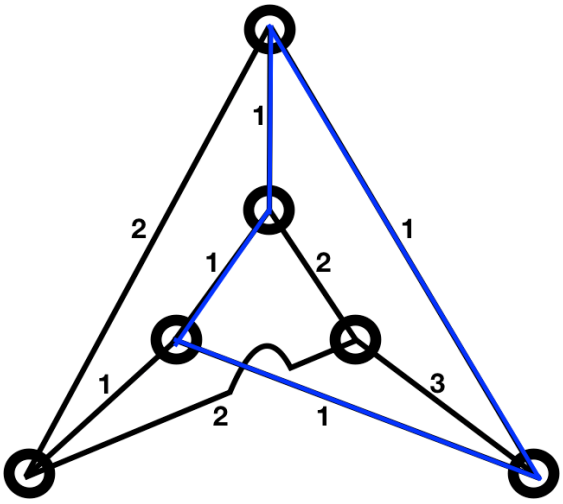
Aufgaben

Swiss Olympiad in Informatics

4. März 2017

Fleptonen

Teilaufgabe 1: iLösen anhand eines Beispiels (10 Points)



Zwei beliebige gegenüberliegende Knoten in dem markierten Kreis sind eine valide Lösung für das Problem (es gibt noch eine weitere Lösung, findest du sie?)

Teilaufgabe 2: Entwickeln eines Algorithmus (50 Points)

Die komplette 50 Punkte Lösung läuft in $O(n \cdot m \log n)$

Wir starten eine leicht modifizierte Version vom Dijkstra Algorithmus an jedem Knoten. Während der Traversierung des Graphen wird nicht nur der kürzeste Pfad gespeichert, aber zusätzlich wird noch der Vorgängerknoten (vom Dijkstra Baum) gespeichert.

Sobald ein bereits besuchter Knoten von einer Kante im Dijkstra Algorithmus mit den gleichen Kosten besucht wird, wissen wir, dass zwei solche Fleptonpfade existieren. Die Lösung wird rekonstruiert, indem die Vorgänger dieser neuen Kante im Dijkstra Baum markiert wird, indem man die gespeicherten Vorgänger zurückläuft, bis man den Startknoten erneut erreicht hat. Nun macht man das gleiche von den Vorgängern der ersten Kante, welche den Knoten besucht hat, bis man an eine Stelle kommt, welche die erste Rückverfolgung schon markiert hat. Dieser Knoten ist der Startpunkt für die Fleptonen, die beiden rückverfolgten Wege sind die Fleptonpfade und der doppelt besuchte Knoten ist der Endpunkt für die Fleptonen.

Dijkstra kann ebenfalls in $O(n \cdot \log(n) + m)$ mithilfe eines Fibonacci-Heaps implementiert werden um eine totale Laufzeit von $O(n^2 \cdot \log(n) + n \cdot m)$, was aber nicht verlangt war.

```cpp
#include <iostream>
#include <queue>

using namespace std;
```

```cpp
struct edge {
  int from, to, price;
};

bool operator>(const edge & lhs, const edge & rhs) {
  if(lhs.price != rhs.price) return lhs.price > rhs.price;
  if(lhs.to != rhs.to) return lhs.to > rhs.to;
  return lhs.from > rhs.from;
}

//Since the priority queue searches for the largest element, but we want the smallest.
bool operator<(const edge & lhs, const edge & rhs) {
  return lhs > rhs;
}

//Wobei 'graph' eine Adjazenzliste des Graphen ist.
pair<int, int> findNodes (const vector<vector<edge> > graph) {
  //start from each vertex
  for(int start = 3; start < graph.size(); ++start) {
    vector<int> visited(graph.size(),-1); //by default -1, elsewise previous number
    vector<int> cost(graph.size(),-1); //by default -1, elsewise cost

    priority_queue<edge> pq;
    pq.push({start,start,0});
    while(!pq.empty()) {
      edge current = pq.top();
      pq.pop();
      if(visited[current.to] == -1) {
        visited[current.to] = current.from;
        cost[current.to] = current.price;
        for(const edge & neighbor : graph[current.to]) {
          pq.push({neighbor.from, neighbor.to, current.price + neighbor.price});
        }
      } else if(current.price == cost[current.to]) {
        //Mark each node in the newly found path with true
        vector<int> mark(graph.size(), false);
        do {
          mark[current.from] = true;
          current.from = visited[current.from];
        } while(current.from != start);
        mark[start] = true;

        //Now track down the first found path until it meets a node, which the newly found path crosses.
        current.from = current.to;
        while(mark[current.from] == false) {
          current.from = visited[current.from];
        }
        return {current.from, current.to};
      }
    }
  }
  return {-1,-1}; //no nodes found
}
```

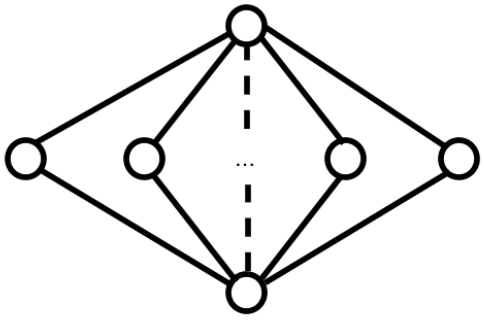
Bewertunsschema:

- Korrekte Idee / Pseudocode: 30 Punkte.
- Laufzeit: 10 Punkte.
- Korrektheit: 5 Punkte.
- Speicherverbrauch: 5 Punkte.

Teilaufgabe 3: Konfiguration um mehr Fleptonen zu messen (40 Points)

Es gibt nur eine Konfiguration für $n = 1, 2, 3$, den trivialen kompletten Graphen.

Um die volle Punktzahl zu erreichen, brauchte man eine Konfiguration von $2 \cdot N + c$ Kanten. Eine solche Konfiguration ist ein Gerüst in Diamantform, indem zwei Knoten mit allen anderen Knoten verbunden wurden. Untereinander verbindet man die restlichen $(n - 2)$ Knoten nicht.
Genau analysiert hat diese Lösung $(n - 2) \cdot 2 = 2n - 4$ Kanten verwendet.



Eine andere interessante Lösung war der komplette Graph, welche die Kriterien erfüllt, da jeder Nachbar zu jedem anderen Nachbar adjazent ist.

Bewertungsschema:

- Eine beliebige quadratische Lösung: 15 Punkte.
- Eine zwischen linear und quadrate Lösung: 20 Punkte.
- Eine langsamere lineare Lösung: 25 Punkte.
- $3 \cdot N + c = 30$ Punkte.
- $2.5 \cdot N + c = 35$ Punkte.
- $2 \cdot N + c = 40$ Punkte.

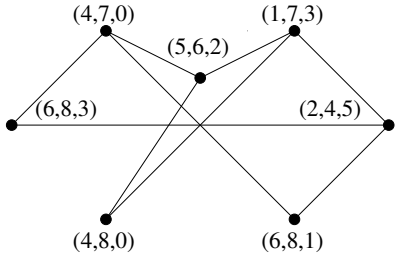
- $1.5 \cdot N + c = 45$ Punkte.

Wenn man nur eine Konstruktion mit geraden/ungeraden Anzahl Detektoren gefunden hat, bekommt man respektive nur 50% von den oberen Punkten.

Teleport

Teilaufgabe 1: Löse ein bestimmtes Netzwerk (10 Punkte)

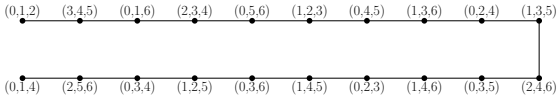
Alle vorhandenen Paare von Teleporter zwischen denen Mäuse sich direkt bewegen können, sind im folgendem Bild angezeigt:



Einer der möglichen maximalen kürzesten Pfade ist $d(F, G) = 3$.

Teilaufgabe 2: Netwerk mit maximaler Distanz (45 Punkte)

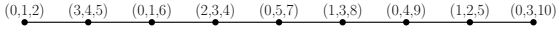
Eine mögliche Lösung für diese Teilaufgabe besteht aus 20 Teleporter, welche unten beschrieben ist. Die Verbindungen zeigen alle mögliche Paare von Teleportern auf, zwischen welchen Aliens direkt reisen können. Dies ist ebenfalls die grösst-mögliche Lösung und sie wurde mit einem Bruteforce-Algorithmus gefunden.



Teilaufgabe 3: Obere Grenze für die Anzahl Detektoren (45 Punkte)

Die erste Observation ist, dass wir nur in Netzwerke interessiert sind, welche aus einer einzelnen Verkettung von Teleportern $V_0, \ldots V_n$ bestehen, wobei man nur zwischen V_i und V_{i+1} reisen kann, um die gewünschte Länge zu erhalten.

Viele eurer Lösungen behaupten, dass die Anzahl von unterschiedlichen Frequenzen durch eine kleine obere Grenze (zwischen 7 und 10) beschränkt ist. Dies ist aber nicht der Fall, da man auch eine Kette mit 11 Frequenzen finden kann:



Das Transportationsnetzwerk von Subtask B zeigt eine mögliche optimale Lösung, welche mithilfe eines Bruteforce gefunden wurde. Die best-mögliche Antwort für diesen Task wäre also 19.

Uns ist kein mathematischer Beweis dieser oberen Grenze bewusst. Stattdessen zeigen wir eine etwas entspanntere obere Schranke, welche immer noch die volle Punktzahl erreicht:

Nehmen wir an die Teleporter V_0 und V_n sind diejenigen mit der grössten Distanz

zwischen einander in dem Transportationsnetzwerk und das $d(V_0, V_n) = n$ für irgend ein n.

Beschreiben wir die Teleporter zwischen V_0 und V_n sequentiell als $V_0, V_1, \ldots, V_{n-1}, V_n$.

In unserem Beweis brauchen wir keine anderen Teleporter und wir können die restlichen weglassen (Betrachten wir das kleinste solche Netzwerk. Es wird nur aus den Teleportern $V_i, 0 \leq i \leq n$ bestehen und es wird ein Pfad sein.)

Sei $\{x_i, y_i, z_i\}$ die drei unterschiedlichen Frequenzen der Transmitter im Teleporter V_i. Da V_0 und V_1 verbunden sind, wissen wir, dass $\{x_0, y_0, z_0\} \cap \{x_1, y_1, z_1\} = \emptyset$.

Desweiteren, da V_1 nicht mit einer der $V_i, 3 \leq i \leq n$ verbunden ist (ansonsten könnten wir den kürzesten Pfad im Netzwerk verkürzen), wissen wir, dass V_i einer der Frequenzen voon V_1 benutzt. Das gleiche Argument gilt auch für V_0.

Da die Frequenzen in V_0 und V_1 unterschiedlich sind, wissen wir, dass jedes $V_i, 3 \leq i \leq n$ mindestens eine Frequenz von $\{x_0, y_0, z_0\}$ und eine Frequenz von $\{x_1, y_1, z_1\}$ verwendet.

Diese gibt es in 9 möglichen Kombinationen von Frequenzen, von denen je einer in $V_i, 3 \leq i \leq n$ vorkommen muss. Also existiert eine Kombination welche in $\frac{n-2}{9}$ Teleporter präsent ist. [1].

Ohne Beschränkung der Allgemeinheit nennen wir diese Kombination $\{x_0, x_1\}$ und bezeichnen die $T := \frac{n-2}{9}$ Teleporter in der sequentiellen Reihenfolge, in der sie im Pfad erscheinen mit $V_{t_1}, V_{t_2}, \ldots V_{t_T}$.

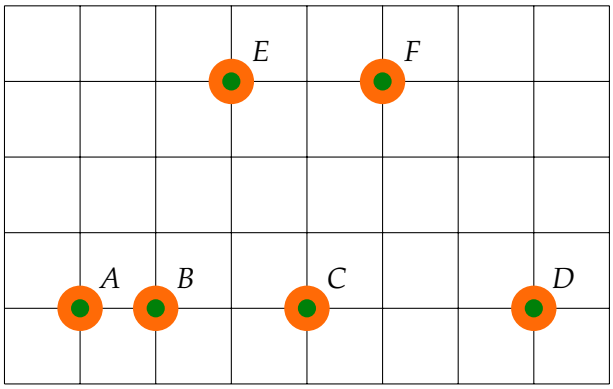
Sei $\{x_0, x_1, z_{z_i}\}$ die Frequenzen von V_{t_i}. Beachte das all diese z_{t_i} (für $1 \leq i \leq T$) unterschiedlich sein müssen, ansonsten könnten wir die Pfade kürzen, was zu einem Widerspruch führen würde. Wir werden die Anzahl unterschiedliche Frequenzen z_{t_i} zählen, welche wir in Teleportern auf dem Pfad haben können.

Beachte, dass wenn V_{t_i} und V_{t_j} mindestens eine Frequenz teilen, die Teleporter nicht verbunden sind und verbundene Teleporter V_{t_i-1} und V_{t_i} unterschiedliche Frequenzen haben (wobei $1 \leq t \leq T$). Aber was würde passieren, wenn keine $z_{t_j} (1 \leq j \leq T, j \neq i)$ Frequenzen von V_{t_i-1} wären? Dann wären die Frequenzen von V_{t_i-1} und V_{t_j} (für $i \neq j$) unterschiedlich und weil V_{t_i-1} und V_{t_j} verbunden wären, würde dass wieder zu einem Widerspruch führen. Also müssen die Frequenzen von V_{t_i-1} alle vorhandenen $z_{t_j} (i \neq j)$ sein. Da V_{t_i-1} höchstens 3 solche Frequenzen mit z_{t_i} haben kann, können wir höchstens 4 solche Frequenzen haben, was $\frac{n-2}{9} = T \leq 4$ impliziert, was uns obere Schranke $n \leq 38$ gibt.

[1] Die -2 sind um V_0, V_1 und V_2 abzuziehen

Karottengarten

Teilaufgabe 1: Beispiel



Es gibt 3 Mögliche Endstadien: $\{A, B\}$, $\{E, F\}$ und $\{A, B, C, D, E, F\}$. Die ersten beiden erhält man, wenn man A, B bzw. E, F am Anfang infiziert. Bei allen anderen Kombinationen sind am Ende alle Karotten infiziert.

Teilaufgabe 2: Algorithmus Als erstes transformieren wir unser Problem in ein Graphen-Problem. Jede Karotte ist ein Knoten im ungerichteten Graph $G = (V, E)$. Jeder Knoten ist mit jedem anderen Knoten verbunden wobei das Gewicht der Kante die Distanz zwischen den beiden Karotten ist.

Wir machen die folgende Beobachtung: Nehmen wir an die Knoten $v_1, v_2, \ldots, v_r$ sind ein mögliches Endstadium und x ist die maximale Distanz zwischen zwei Knoten in diesem Endstadium. Wie sieht der Graph G aus wenn wir alle Kanten mit einem Gewicht grösser als x entfernen? Zum einen sind die Knoten $v_1, \ldots, v_r$ vom Rest des Graphen abgeschnitten, sonst könnte sich «Stoffulus »weiter ausbreiten und es kein Endstadium. Auf der anderen Seite ist jeder Knoten in $v_1, \ldots, v_r$ mit jedem anderen Knoten in dieser Menge verbunden, schliesslich sind ja alle Gewichte zwischen den Knoten kleiner als x(da x per Definiton das maximum all dieser Gewichte ist). Zusammengefasst erfüllt jedes Endstadium $F = v_1, v_2, \ldots, v_r$ folgende Kriterien: Sei $G' = (V, E')$ ein Teilgraph von G wobei $x = maxw(e)|e \subset F$ und $E' = w(e) \leq x|e \in E$.

- Es gibt keine Kante von F zum rest des Graphen($\nexists x \in F, y \in V \backslash F$ so dass $x, y \in E'$).
- Alle Konten in F sind verbunden($\forall x, y \in F$ haben wir $x, y \in E'$).

Diese Kriterien sind nicht nötig, sondern auch ausreichend. Das heisst wenn wir eine Teilmenge von Konten haben die diese Kriterien erfüllen dann handelt es sich um ein Endstadium(wenn man mit 2 beliebigen Karotten in dieser Teilmenge startet landet man in diesem Endstadium). Um alle Endstadien zu finden können wir also einfach für alle möglichen x schauen, ob es eine oder mehrere Teilmenge gibt, die dieses Kriterium erfüllen.

Der Algorithmus um dies effizient zu machen funktoniert wie folgt: Wir entfernen alle Kanten aus dem Graph und sortieren sie nach ihrem Gewicht. Dann fügen wir die Kanten, beginnend mit dem kleinsten Gewicht, nacheinander wieder ein und beaobachten dabei die verschiedenene Komponenten. Wenn zwei Kanten das selbe Gewicht haben, dann fügen wir diese gleichzeitig ein, das heisst wir schauen erst wieder auf die Komponenten wenn alle Kanten mit dem selben Gewicht eingefügt sind. Sobald ein Komponent $k \cdot (k-1)$ Kanten hat, wobei k die Anzahl Knoten im Komponenten ist, erfüllt er die beiden Kriterien und wir haben einen neuen Endzustand gefunden. Um die verschiedenen Komponenten zu beobachten und über die Anzahl Kanten in einem Komponent buch zu führen, verwenden wir eine Union-Find Datenstruktur(siehe Runde 2h) wobei wir zusätlich für jede Komponente die Anzahl Kanten speichern.

Das sortieren der Kanten benötigt $O(|E| \cdot \log |E|)$, das initialisieren der Union-Find Struktur $O(|V|)$ und die Arbeit in jeder Iteration der For-Schleife ist Konstant(alle Operationen auf Union-Find sind Konstant). Unser Graph ist vollständig, daher gilt $|E| = \frac{|V|(|V|-1)}{2}$ und wir kriegen unsere Laufzeit $O(n + n^2 \cdot \log n^2) = O(n^2 \cdot \log n)$. Die Speicherkomplexität ist entsprechend $O(n^2)$.

Der Algorithmus ist korrekt weil wir alle Endstadien zählen und nie ein Endstadium zweimal zählen. Wir markieren jedes Endstadium, das wir gezählt haben und es ist deshalb nicht möglich, dass wir eines zwei mal zählen. Jedes Endstadium erfüllt die zwei Bedingungen für ein gewisses x. Da wir alle x(oder zumindest die, bei denen sich der Graph verändert) durchgehen, werden wir jedes Endstadium mindestens einmal zählen.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <algorithm>
using namespace std;


struct union_find {
  vector<int> parent;
  vector<int> size;
  vector<int> num_edges;
  vector<bool> counted;

  union_find(int n) {
    parent.resize(n);
    num_edges.resize(n);
    size.resize(n, 1);
    counted.resize(n);

    for(int i = 0; i < n; i++)
      parent[i] = i;
  }

  int find(int u) {
    while(u != parent[u]) {
```

```
27        parent[u] = parent[parent[u]];
28        u = parent[u];
29      }
30    }
31
32    void unite(int v, int u) {
33      int rv = find(v);
34      int ru = find(u);
35
36      if(rv == ru) {
37        num_edges[rv]++;
38      } else {
39        int e = num_edges[rv] + num_edges[ru];
40        int s = size[rv] + size[ru];
41        if(rand()%2) {
42          parent[ru] = rv;
43        } else {
44          parent[rv] = ru;
45        }
46        counted[parent[ru]] = false;
47        num_edges[parent[ru]] = e;
48        size[parent[ru]] = s;
49      }
50    }
51
52    int count_new_components() {
53      int count = 0;
54      for(size_t i = 0; i < parent.size(); i++) {
55        if(i == parent[i] && !counted[i]) {
56          if(num_edges[i] == size[i]*(size[i]-1)/2) {
57            count++;
58            counted[i] = true;
59          }
60        }
61      }
62    }
63  };
64
65
66  struct edge {
67    int u;
68    int v;
69    int weight;
70
71    bool operator<(edge const &e) const {
72      return weight <  e.weight;
73    }
74  };
75
76  int carrots(int n, vector<edge> edges) {
77    sort(edges.begin(), edges.end());
78    union_find uf(n);
```

```
79   int result = 0;
80
81   int last_x = -1;
82   for(edge& e : edges) {
83     uf.unite(e.u, e.v);
84     if(last_x != e.weight) {
85       last_x = e.weight;
86       result += uf.count_new_components();
87     }
88   }
89   cout << result << "\n";
90 }
```

Bewertungsskala:
Brute-force $O(n^5)$: 20 Punkte.
Brute-force $O(n^4)$: 40 Punkte.
Lösung mit MST $O(n^3)$: 70 Punkte.
Lösung mit Union-Find: 90 Punkte.

Die obigen Punkte werden wie folgt aufgeteilt:
Idee/Beschreibung des Algorithmus: 50%
Pseudocode: 20%
Korrektheit: 20%
Laufzeitanalyse: 10%

Cake Protection Agency

Teilaufgabe 1: Späteste Startzeit

Wir können die Zeiten $(t_1 = T, t_2, ..., t_{k-1}, t_{cake})$, die Maus Gehr hat um die Knoten $(v_1 = v_{gehr}, v_2, ..., v_{k-1}, v_k = v_{cake})$ zu erreichen, berechnen, indem wir die Summe der Kantengewichte berechnen, welche Maus Gehr verwendet um seinen Pfad zu folgen.

Der CPA Agent kann das Attentat verhindern, indem er entweder einer der Knoten $(v_1 = v_{\text{gehr}}, v_2, \ldots, v_{k-1})$ zum Zeitpunkt $(t_1 = T, t_2, ...t_{k-1})$ erreicht oder wenn er den Knoten v_{cake} erreicht zu Zeitpunkt $t_k - 1$ erreicht.

Deswegen ist die letzt-mögliche Zeit um Gehr zu fangen $max_{k\in\{1,...,k-1\}}(t_k - d(v_{\text{cpa}}, v_k))$ und $t_k - 1 - d(v_{\text{cpa}}, v_{\text{cake}})$. Wir verwenden Dijkstra's Algorithmus um die Distanz von v_{cpa} zu allen anderen Knoten im Graphen zu berechnen. Verwendet man einen Max-Heap für Dijkstra erhält man die Laufzeit: $O(|E|\log|V|)$ und mit Fibonacci-Heaps eine Laufzeit von $|\mathcal{V}|\log|\mathcal{V}| + |\mathcal{E}|$. Speicherplatzt ist je nach Implementierung $O(|E|)$ oder $O(|V|)$.

```
struct Edge {
  int to;
  int weight;
};
struct Node {
  vector<Edge> out;
};
int sub1(const vector<Node> &G, int v_cpa, int v_gehr, int v_cake,
         const vector<int> &p, int T) {
  int t_i = T, t_cake = T;
  for (int i = 1; i < (int)p.size(); i++) {
    t = t_cake;
    for (Edge &e : G[p[i - 1]]) {
      if (e.to == p[i]) {
        t_cake += e.weight;
        break;
      }
    }
  }
  vector<int> distances = dijkstra(G, v_cpa);
  return max(t - distances[p.size() - 1], t_cake - 1 - distances[v_cake]);
}
```

Teilaufgabe 2: Viele Agenten

Da wir eine unlimitierte Anzahl Agenten haben, wird Maus Gehr am meisten gehindert, wenn ein CPA-Agent zu jedem Knoten geschickt wird.

Maus Gehr kann nur Knoten erreichen, die er vor den Agenten erreichen kann (oder zur gleichen Zeit, wenn er den Kuchen v_{cake} erreicht).

Beachte zusätzlich, dass Maus Gehr immer einen kürzesten Weg verwendet. Um einzusehen, dass dies der Fall ist, merke, dass es für Stoffl von Vorteil ist, früher an einem

Knoten zu sein (statt später an diesem Knoten einzutreffen). Das eliminiert direkt die Möglichkeit, dass Maus Stoffl sich in Kreisen bewegt. Nimm ausserdem an, dass Maus Gehr erfolgreich vom Knoten v_{gehr} zum Knoten w über einen Pfad p läuft, welcher kein kürzester Pfad ist. Betrachte jetzt einen kürzesten Weg q von v_{gehr} zum Knoten w. Per Annahme gilt $p \neq q$, also gibt es Knoten die nicht sowohl in p als auch in q sind. Der einzige Grund warum Maus Gehr vielleicht diese Knoten umgehen und einen längeren Pfad bevorzugen würde, wäre falls die CPA einen Knoten x in q vor Maus Gehr erreichen könnten.

Dies impliziert aber direkt, dass die CPA auch Knoten w vor Maus Gehr erreichen kann. Deswegen werden sie ihn auch dann fangen, auch dann wenn $w = v_{\text{cake}}$, was bedeutet, dass Maus Gehr nichts gewinnt, wenn er einen längeren Pfad wählen würde.

Wir können für den Knoten v den Zeitpunkt t_v (der späteste Zeitpunkt die CPA einen Agenten losschicken müssten um Stoffl zu fangen) berechnen, indem wir die kürzeste Zeit die Gehr hat um Knoten v zu erreichen von der kürzesten Zeit die die CPA Agenten haben Knoten v zu erreichen, subtrahieren (für v_{cake} subtrahieren wir zusätzlich 1). Beachte zusätzlich, dass $t_v = \infty$, wenn v nicht von Maus Gehr erreichbar ist und $-\infty$, falls Maus Gehr den Kuchen erreichen kann, die CPA Agenten aber nicht.

Die CPA Agenten müssen nun folgendes Problem lösen: Finde die späteste Zeit T, sodass wenn wir alle Knoten w mit $t_w \geq T$ vom Graphen entfernen, der Kuchen nicht von Maus Gehr's Ursprungsposition erreichbar ist. Für T kommen nur die Zeiten von t_v in Frage. Mit einer Tiefen- oder Breitensuche können wir gegeben einen Teilgraphen, diese Teilbedingung in $O(|V| + |E|)$ Zeit berechnen. Da es einen ganz bestimmten Zeitpunkt gibt, ab dem Gehr den Kuchen nicht mehr erreichen kann, können wir mithilfe einer binären Suche über t_v die geeignete Zeit finden. Die Laufzeit dieses Algorithmus wäre dann $O((|V| + |E|) \cdot \log |V|)$. Der zusätzlichen Speicherverbrauch ist $O(|V|)$.
Eine andere Lösung, wäre die adjazenten Kanten der aufsteigend sortierten Knoten in eine Union Find Struktur hinzuzufügen, bis t_{gehr} einen Pfad zu t_{cake} hat. Die Laufzeit davon wäre $O(|V| \log |V| + |E| \log^* V)$.

Sei $\Gamma^-(v)$ die Nachbarknoten von v, welche eine Kante haben, die auf v zeigt. Das Endresultat ist $max(t_{\text{cake}}, \min_{v \in \Gamma^-(v_{\text{cake}})}[t_v])$ (Entweder blockieren sie den Kuchen und die Knoten um den Kuchen oder sie wählen die Option, die ihnen erlaubt später zu reagieren).

```
int sub2(const vector<Node> &G, int v_cpa, int v_gehr, int v_cake, int T) {
  vector<int> d_cpa = dijkstra(G, v_cpa);
  vector<int> d_gehr = dijkstra(G, v_gehr);
  vector<int> t(G.size()), gamma;
  for (int i = 0; i < (int)G.size(); i++) {
    t[v] = T + d_gehr[i] - d_cpa[i] - (i == v_cake);
    for (Edge &e : G[i]) {
      if (e.to == v_cake) {
        gamma.push_back(i);
        break;
      }
```

```
12      }
13    }
14    int worst = INFINITY;
15    for (int v : gamma) {
16      worst = min(worst, t[v]);
17    }
18    return max(t[v_cake], worst);
19  }
```

(In dieser Implementation, können die Vektoren t und gamma ausgelassem werden, indem die notwendigen Berechnungen 'inline' hätten geschehen können. Diese Implementierung wurde gewählt um ein nahe Korrespondez zwischen der Lösung im Text zu haben.)

Teilaufgabe 3: Ein einzelner Agent

Wir können wieder ohne Beschränkung der Allgemeinheit annehmen, dass Maus Gehr immer den kürzesten Weg zwischen v_{cake} wählt. Zusätzlich können wir observieren, dass dies auch der Fall ist für den CPA Agenten: Falls der Agent es schafft Maus Gehr zu fangen, während er nicht auf einem kürzesten Pfad ist, hätte der Agent v_{cake} strikt vor Maus Gehr erreichen können, wenn sie gleichzeitig starten. Zusätzlich können wir annehmen, dass wenn der Agent Maus Gehr irgendwo auf dem Pfad fängt, dann bewegt sich der Agent auf das gleiche Feld zur gleichen Zeit (Ansonsten könnte sich der Agent direkt zum Kuchen bewegen, anstelle zu warten, bis Maus Gehr zu diesem Knoten kommt).

Es gibt zwei Fälle zu beachten:

1. Den Fall wo der Agent Maus Gehr fängt, indem er sich zu v_{cake} strikt früher begibt.
2. Der Agent fängt Maus Gehr irgendwo auf dem Pfad, indem er gleichzeitig an einen Knoten wie Maus Gehr landet.

In Fall 1 ist die späteste mögliche Zeit T'_{cake} um den Agenten loszuschicken, einfach zu berechnen. In Fall 2 fängt der Agent Maus Gehr bevor er den Kuchen erreicht.

Das Endresultat ist entweder T'_{cake} oder $T'_{\text{cake}} + 1$.

Jetzt geht es darum festzustellen, welcher der Fälle eintritt. Es reicht zu überprüfen ob der Agent Maus Gehr fangen kann, in dem sich beide gleichzeitig in einen Knoten bewegen. Wir berechnen die Distanz zu jedem Knoten vom Kuchen aus in dem wir Dijkstra vom Kuchen aus starten (und uns rückwärts auf den gerichteten Kanten bewegen).

Seien $E' \subseteq E$ alle Kanten auf dem kürzesten Pfad zum Kuchen. (Dies sind genau diejenigen Kanten, welche zwei Knoten verbinden, deren Differenz zum Kuchen genau dem Gewicht der Kante entspricht.)

Dann können wir die Situation als Spiel ansehen: Vor jedem Zug sind die beiden Knoten v_g und v_a bekannt, den momentanen Positionen von Maus Gehr und dem Agenten. Beachte, dass dieses Spiel diskretisiert ist, d.h. die Distanzen des Agenten und Maus Gehr zum Kuchen unterschiedlich sein dürfen. Wir lassen denjenigen Spieler

den nächsten Zug durchführen, der sich weiter vom Kuchen entfernt befindet. Sind beide gleich weit entfernt, ist Maus Gehr am Zug. (Dies entspricht der Bedingung, dass sich Maus Gehr zuerst entscheiden muss, wenn er und der Agent gleichzeitig an einer Kreuzung stehen.) Sowohl Maus Gehr als auch der Agent dürfen sich nur entlang von Kanten in E' bewegen.

Eine Position ist genau dann gewinnend für den Agenten, wenn er Maus Gehr von seiner Position aus fangen kann (in dem sich beide in den gleichen Knoten bewegen), ausser Maus Gehr kann sich zum Kuchen flüchten.

Dadurch können wir folgende Rekursion aufstellen, welche die Gewinnpositionen für den Agenten beschreiben:

- Falls Maus Gehr den Kuchen erreicht, hat der Agent verloren.
- Falls der Agent mindestens so nahe am Kuchen ist wie Maus Gehr, dann ist die Position (v_g, v_a) genau dann gewinnend, wenn alle Positionen (v_g, v'_a), welche Maus Gehr durch eine Kante aus E' erreichen kann, für den Agenten gewinnend sind.
- Falls Maus Gehr näher am Kuche ist wie der Agent, ist die Position (v_g, v_a) genau dann gewinnend, wenn der Agent eine gewinnende Position (v'_g, v_a) durch eine Kante aus E' erreichen kann.

Formal: Sei $d(v, v_{\text{cake}})$ die Distanz von Knoten v zum Kuchen. Wir schreiben $\text{win}(v_g, v_a) = 1$ genau dann, wenn der Agent von der Position (v_g, v_a) aus Maus Gehr vor dem Kuchen fangen kann.

Seien $\Gamma'^{+}(v) \subseteq V$ alle Knoten, welche von Knoten v aus durch direkt durch eine Kante aus E' erreichbar sind.[1]

$$\text{win}(v_g, v_a) = 0, \text{ für } v_g = v_{\text{cake}},$$
$$\text{win}(v_g, v_a) = 1, \text{ für } v_g \neq v_{\text{cake}} \text{ and } v_g = v_a,$$
$$\text{win}(v_g, v_a) = \bigwedge_{v'_g \in \Gamma'^{+}(v_g)} \text{win}(v'_g, v_a), \text{ für } v_g \neq v_{\text{cake}}, v_g \neq v_a \text{ und } d(v_a, v_{\text{cake}}) \leq d(v_g, v_{\text{cake}}),$$
$$\text{win}(v_g, v_a) = \bigvee_{v'_a \in \Gamma'^{+}(v_a)} \text{win}(v_g, v'_a), \text{ für } v_g \neq v_{\text{cake}}, v_g \neq v_a \text{ und } d(v_g, v_{\text{cake}}) < d(v_a, v_{\text{cake}}).$$

Da es keine Zyklen im Graph $G' = (V, E')$ gibt können wir die obere Rekursion mittels dynamischer Programmierung auswerten. Es ist am einfachsten, die obige Funktion rekursiv zu implementieren und Memoisierung anzuwenden.

Das endgültige Resultat ist:

[1] Noch mehr Notation: Wir bezeichnen mit $a \wedge b$ das logische "und" und mit $a \vee b$ das logische "oder". $\bigwedge$ und $\bigvee$ verhalten sich zu $\wedge$ und $\vee$ wie $\sum$ zu $+$. Bei leeren Mengen definieren wir $\bigwedge_{x \in \{\}} f(x) = 1$ und $\bigvee_{x \in \{\}} f(x) = 0$.

$$T' = T + d(v_{\text{gehr}}, v_{\text{cake}}) - d(v_{\text{cpa}}, v_{\text{cake}}) - 1 + \text{win}(v_{\text{gehr}}, v_{\text{cpa}}).$$

d.h. falls die Ausgagsposition von Maus Gehr und dem Agenten für den Agenten gewinnend sind, kann der Agent dann loslaufen, dass er gleichzeitig mit Maus Gehr am Kuchen ankommen würde. Sonst muss der Agent eine Zeiteinheit früher losgehen um den Kuchen strikt vor Maus Gehr zu erreichen.

Die Laufzeit dieser Lösung wird durch den Teil mit der dynamische Programmierung dominiert, wessen Laufzeit durch $O(|V|(|V| + |E|))$ gebunden werden kann. Der zusätzliche Speicherverbrauch wird durch die DP-Tabelle dominiert und ist $O(|V|^2)$.

```
const vector<Node> &G;
int v_cpa, int v_gehr, int v_cake;
int T;

vector<int> d_gehr, d_cpa, d_cake;
bool inEPrime(int from, const Edge &e) {
  return d_cake[from] + e.weight == d_cake[e.to];
}

vector<vector<int>> memo;
int win(int v_g, int v_a) {
  if (v_g == v_cake) {
    return 0;
  }
  if (v_g == v_a) {
    return 1;
  }
  if (memo[v_g][v_a] != -1) {
    return memo[v_g][v_a];
  }
  bool result;
  if (d_cake[v_a] <= d_cake[v_g]) {
    // Gehr's turn
    result = true;
    for (Edge &e : G[v_g]) {
      if (inEPrime(v_g, e)) {
        result = result && win(e.to, v_a);
      }
    }
  } else {
    // Agent's turn
    result = false;
    for (Edge &e : G[v_a]) {
      if (inEPrime(v_a, e)) {
        result = result || win(v_g, e.to);
      }
    }
  }
```

```
  memo[v_g][v_a] = result;
  return result;
}
int sub3() {
  d_cpa = dijkstra(G, v_cpa);
  d_gehr = dijkstra(G, v_gehr);
  d_cake = dijkstra(G, v_cake);
  return T + d_gehr[v_cake] - d_cpa[v_cake] - 1 + win(v_gehr, v_cpa);
}
```