Second Round Theoretical

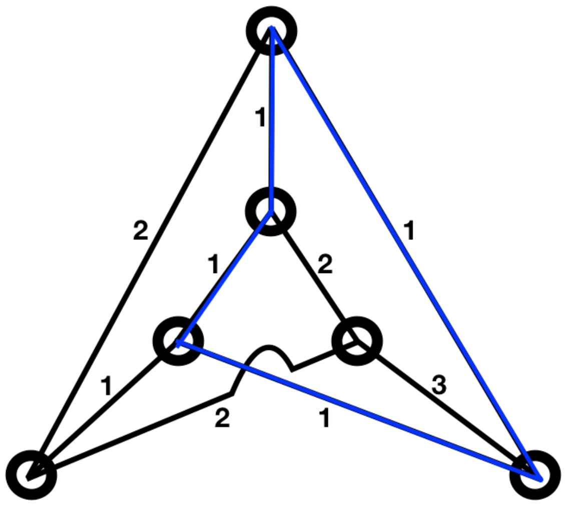
Solutions

Swiss Olympiad in Informatics

March 4, 2017

Fleptons

Subtask 1: Solve the Example (10 Points)



Any two opposite nodes of the marked circle are a valid solution for the problem (there is one other, can you spot it?)

Subtask 2: Develop an Algorithm (50 Points)

The full 60 point solution runs in $O(n \cdot m \log n)$.

We start the Dijkstra algorithm from each of the nodes and during this graph traversal we store not only the shortest path (in the beginning, when no path exists, mark each entry with some arbitrary value, say -1), but additionally on each node, we have to store the predecessor node of the Dijkstra algorithm.

The first time an edge in the Dijkstra algorithm visits a node, which it already visited previously, with the same (shortest) distance, we can construct the solution. Since there are two paths leading from x to y and their final edge is different, there has to be a node z, where the two paths lastly split.

By backtracking from both paths (going down the path twice if need be) this node z can be found. Now it is not guaranteed, that you find such a path starting from only a single node, however when you start Dijkstra from each point, you are guarenteed to also start Dijkstra at this node z and when you do, you are guaranteed to find such two paths.

Dijkstra can also be implemented in $O(n \cdot \log(n) + m)$ to get a faster total runtime of $O(n^2 \cdot \log(n) + n \cdot m)$.

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

```

```cpp
using namespace std;
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
}
```

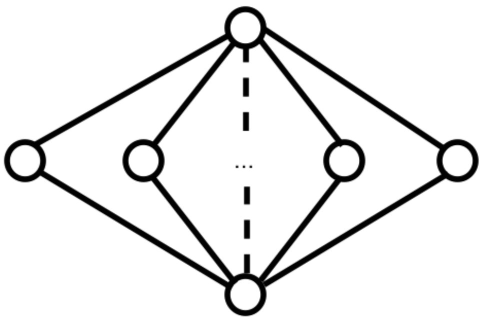
Grading scheme

- Correct Idea / Pseudocode: 30 points
- Runtime: 10 points.
- Correctness: 5 points.
- Memory usage: 5 Punkte.

Subtask 3: Configuration for Detecting more Fleptons (40 Points)

There is only one configuration for $n = 1, 2, 3$, which is the trivial complete graph.

For the full point solution you can use a diamond shape, by putting one node on top and one on the bottom. In the middle you place the remaining detectors in a line and you finally connect each middle node once with the top node and once with the bottom node. This leads to $(n-2) \cdot 2 = 2n - 4$ edges, which was required for the full points.



Another interesting solution was the complete graph, which fullfils the property, because every neighbor is adjacent.

Grading scheme:

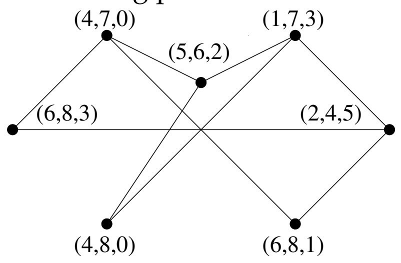
- Any quadratic solution: 15 points.
- Between linear and quadratic solution: 20 points.
- Any slower linear time solution: 25 points.
- $3 \cdot N + c = 30$ points.
- $2.5 \cdot N + c = 35$ points.
- $2 \cdot N + c = 40$ points.
- $1.5 \cdot N + c = 45$ points.

If you only constructed a configuration for an odd/even number of detectors, you get half of the points above respectively.

Teleport

Subtask 1: Solve some Network (10 Points)

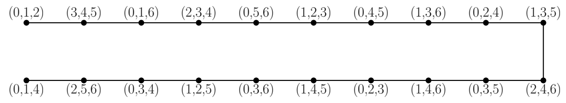
All the available pairs of teleports between which mice can travel directly are shown in the following picture.



One of the possible maximum shortest distances is $d(F, G) = 3$.

Subtask 2: Network with Maximum Distance (45 Points)

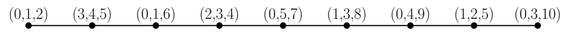
One possible solution of this subtask, consisting of 20 teleports, is depicted below. The links show all of the possible pairs between which aliens can travel directly. This is also the maximum solution, which has been found by a bruteforce algorithm.



Subtask 3: Bound for Maximum Distance (45 Points)

First observation is that we are only interested in networks that form a single "chain" of teleports $V_0, \ldots V_n$, where one can travel between all V_i and V_{i+1}, but not any other pair, since we can always remove leftover teleports to keep only the path of desired length.

Many of your solutions claim that the number of different frequencies is bounded by a small constant (between 7 and 10). However this is not the case, since one can also find a chain that uses 11 frequencies:



The figure from Subtask B shows one possible optimal solution – by a bruteforce algorithm we have shown that it consists of 20 teleports and hence the answer scoring maximum number of points of this subtask is 19.

We are not aware of a mathematical proof of this bound. Instead we show a bit relaxed upper bound on the maximum distance in a network that earns full score.

Let's assume that teleports V_0 and V_n are the most distant in a network and $d(V_0, V_n) = n$ for some n and n is maximal possible. Let's denote the teleports on the shortest path between V_0 and V_n sequentially as $V_0, V_1, \ldots, V_{n-1}, V_n$. In our proof we will not need any other teleports and hence we can omit those. (We can consider the smallest such network. It will consist of teleports V_i, $0 \leq i \leq n$ only and it will be a path.)

Let $\{x_i, y_i, z_i\}$ be the three distinct operating frequencies of transmitters located in teleport V_i. As V_0 and V_1 are connected, we know that $\{x_0, y_0, z_0\} \cap \{x_1, y_1, z_1\} = \emptyset$. Furthermore, since V_1 is not connected with any V_i, $3 \le i \le n$ (otherwise we could shorten the shortest path), we know that V_i uses one of the operating frequencies of V_1. The same argument holds for V_0, too. As the operating frequencies in V_0 and V_1 are distinct, we know that each V_i, $3 \le i \le n$ must use at least one operating frequency of $\{x_0, y_0, z_0\}$ and at least one of $\{x_1, y_1, z_1\}$.

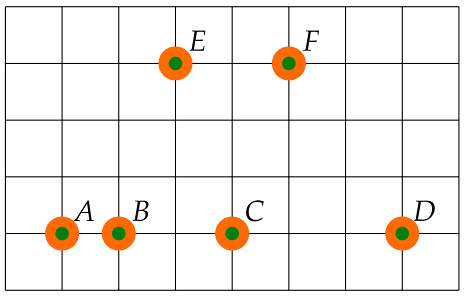
These are in total 9 combinations of operating frequencies one of which must be present in each V_i, $3 \le i \le n$. Hence there exists a combination present it at least $\frac{n-2}{9}$ teleports.[1] Without loss of generality let us denote this combination by $\{x_0, x_1\}$ and let the $T := \frac{n-2}{9}$ teleports be denoted sequentially as they appear in the path by $V_{t_1}, V_{t_2}, \ldots V_{t_T}$. Let the operating frequencies in V_{t_i} be $\{x_0, x_1, z_{t_i}\}$. Note that all z_{t_i} (for $1 \le i \le T$) must be different, otherwise we would be able to shorten our shortest path which would be a contradiction. We will count how many distinct frequencies z_{t_i}s we could have in teleports in our paths.

Note that as V_{t_i} and V_{t_j} share common operating frequencies, they are not connected and hence in our path we must have V_{t_i-1} such that frequencies of V_{t_i-1} and V_{t_i} are distinct ($1 \le t \le T$). But what would happen if no z_{t_j} ($1 \le j \le T$, $j \ne i$) would be an operating frequency of V_{t_i-1}? Then operating frequencies of V_{t_i-1} and V_{t_j} (for $i \ne j$) would be distinct and hence V_{t_i-1} and V_{t_j} would be connected, again yielding a contradiction. Thus operating frequencies of V_{t_i-1} must be all available z_{t_j} ($i \ne j$). As V_{t_i-1} can have at most 3 such frequencies with z_{t_i} we can have at most 4 such frequencies which implies $\frac{n-2}{9} = T \le 4$ which gives us a bound $n \le 38$.

[1]The -2 accounts for V_0, V_1 and V_2.

Carrot Garden

Subtask 1: Example



There are 3 possible final stages: $\{A, B\}$, $\{E, F\}$ and $\{A, B, C, D, E, F\}$. The first two we get when we start A, B and E, F respectively. With all other combinations all carrots get infected eventually.

Subtask 2: Algorithm We start by transforming the problem to a graph problem. Each carrot is a vertex in the undirected Graph $G = (V, E)$. Each vertex is connected with each other vertex and the weight of this edge is the distance between the two carrots.

We make the following observation: Suppose the vertices $v_1, v_2, \ldots, v_r$ are a possible final stage and x is the maximum distance between the vertices in this final stage. How does the graph G look like if we remove all edges whose weight is bigger than x? On one hand the vertices $v_1, v_2, \ldots, v_r$ are not connected to the rest of the graph in any way, otherwise «Stoffulus »could spread along this connection and it wouldn't be a final stage. On the other hand all vertices in $v_1, \ldots, v_r$ are connected to each other vertices in this set, after all each weight is smaller or equal than x(as x is per definiton the maximum of all those weights).

To summarize, each final stage $F = v_1, v_2, \ldots, v_r$ fulfils the follwing conditions:

- There is no edge from F to the rest of the graph($\nexists x \in F, y \in V \backslash F$ such that $x, y \in E'$).
- All vertices in F are mutually connected($\forall x, y \in F$ we have $x, y \in E'$).

Those conditions are no just necessairy, but sufficient. That means, when we have a set of vertices that fulfills those conditions, we know that it is a final stage(if we infect 2 arbitrary carrots in this set, we get thes set as final stage). To find all possible final stages we can simply look for all possible x whether there is a subset that fulfills those conditions.

The algorithm to conduct this procedure efficiently goes as follows: We remove all edges from the graph and sort them by their weight. We then add them back in starting with the smallest and observe the different components. When two edges have the same weight, we insert them at the same time, that is to say we only look at the components after we inserted all edges of the same weight. As soon as a component has $k \cdot (k - 1)$ edges, where k is the number of vertices in that component, he fulfills both conditions and we have found a new final stage. To observe the different components and to keep track of the number of edges in a compoment we use the union-find data structure(see round 2h) where we store in addition the number of edges in each component.

The sorting of the edges takes $O(|E| \cdot \log |E|)$, the initialisation of the union-find takes $O(|V|)$ and the work in each iteration of the for-loop takes constant time(all operation on union-find are $O(1)$). The graph is fully-connected thus we have $|E| = \frac{|V|(|V|-1)}{2}$ and we get our total runtime complexity $O(n + n^2 \cdot \log n^2) = O(n^2 \cdot \log n)$. The memory complexity is $O(n^2)$.

```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]) {
      parent[u] = parent[parent[u]];
      u = parent[u];
    }
  }

  void unite(int v, int u) {
    int rv = find(v);
    int ru = find(u);

    if(rv == ru) {
      num_edges[rv]++;
    } else {
      int e = num_edges[rv] + num_edges[ru];
      int s = size[rv] + size[ru];
      if(rand()%2) {
        parent[ru] = rv;
      } else {
        parent[rv] = ru;
      }
      counted[parent[ru]] = false;
      num_edges[parent[ru]] = e;
      size[parent[ru]] = s;
    }
  }

  int count_new_components() {
    int count = 0;
    for(size_t i = 0; i < parent.size(); i++) {
      if(i == parent[i] && !counted[i]) {
        if(num_edges[i] == size[i]*(size[i]-1)/2) {
          count++;
          counted[i] = true;
        }
      }
    }
  }
};


struct edge {
  int u;
  int v;
  int weight;

  bool operator<(edge const &e) const {
    return weight <  e.weight;
  }
};

int carrots(int n, vector<edge> edges) {
  sort(edges.begin(), edges.end());
  union_find uf(n);
  int result = 0;

  int last_x = -1;
  for(edge& e : edges) {
    uf.unite(e.u, e.v);
    if(last_x != e.weight) {
      last_x = e.weight;
      result += uf.count_new_components();
    }
  }
  cout << result << "\n";
}
```

Grading scheme:
Brute-force $O(n^5)$: 20 points.
Brute-force $O(n^4)$: 40 points.
Solution using MST: $O(n^3)$: 70 Points.
Solution using Union-Find: 90 Points.

The points above are distributed as follow:
Idea/Description of the algorithm: 50%
Pseudocode: 20%
Correctness: 20%
Complexity Analysis: 10%

Cake Protection Agency

Subtask 1: Latest Starting Time

We can compute the times $(t_1 = T, t_2, \ldots, t_{k-1}, t_{\text{cake}})$ at which mouse Gehr arrives at each of the vertices $(v_1 = v_{\text{gehr}}, v_2, \ldots, v_{k-1}, v_k = v_{\text{cake}})$ by summing up the weights along the edges that mouse Gehr travels along on his path. The CPA agent can prevent the evil plot if it either reaches one of the vertices $(v_1 = v_{\text{gehr}}, v_2, \ldots, v_{k-1})$ at time $(t_1 = T, t_2, \ldots t_{k-1})$ or if it reaches the final vertex at time $t_k - 1$. Hence the resulting last possible leaving time for the CPA agent is the maximum of $\max_{k \in \{1,\ldots,k-1\}}(t_k - d(v_{\text{cpa}}, v_k))$ and $t_k - 1 - d(v_{\text{cpa}}, v_{\text{cake}})$. To compute this result, it is actually enough to only consider t_{k-1} and t_k, because we could imagine the agent moving along a shortest path to v_{k-1} simultaneously with mouse Gehr catching him only there. (This establishes that $t_k - d(v_{\text{cpa}}, v_k) \geq \max_{k \in \{1,\ldots,k-2\}}(t_k - d(v_{\text{cpa}}, v_k))$.) The result is then given by $\max(t_{k-1} - d(v_{\text{cpa}}, v_k), t_k - 1 - d(v_{\text{cpa}}, v_{\text{cake}}))$

We use Dijkstra's Algorithm to compute the distance of v_{cpa} to all other nodes in the graph. The running time and the additional memory usage are dominated by Dijkstra's algorithm and are therefore given by $O(|V| \log |V| + |E|)$ and $O(|V|)$ respectively.[1]

```cpp
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]);
}
```

[1]Note that the implementation of Dijkstra usually used in practice has a larger asymptotic running time: $O((|V| + |E|) \log |V|)$.

Subtask 2: Plenty of Agents

As we have an unlimited supply of agents, the course of action that limits mouse Gehr the most, after they have been dispatched, is to send one agent to each vertex, as quickly as possible. Mouse Gehr can then only reach those vertices that the agents are not able to reach before him (or possibly at the same time, with the exception of v_{cake}).

Further note that without loss of generality, mouse Gehr will always travel along a shortest path. To see why this is the case, note that it is always better for mouse Gehr to be at a given vertex earlier rather than later. This immediately excludes the possibility that mouse Gehr might move around in cycles. Assume that mouse Gehr successfully moved from vertex v_{gehr} to vertex w along some (simple) path p that is not a shortest path. Now consider some shortest path q from vertex v_{gehr} to vertex w. As by assumption, $p \neq q$, there are vertices that are in q but not in p. The only reason why mouse Gehr might need to avoid those vertices and take a longer path instead is because the CPA can reach a vertex x among them at least as early as mouse Gehr. However, this immediately implies that the CPA will reach the vertex w strictly earlier than mouse Gehr (the CPA can just move along a shortest path from vertex x to vertex w). Therefore, it will catch him even if $w = v_{\text{cake}}$, which means that mouse Gehr cannot gain anything by moving along a path that is not a shortest path.

We can compute for each vertex v, at which time t_v the agents would need to be dispatched to prevent mouse Gehr from entering this vertex, by subtracting the time it takes the CPA agents to reach the vertex from mouse Gehr's earliest possible arrival time at this vertex (for $v = v_{\text{cake}}$, we additionally subtract 1). The correct result will be t_v for one of the vertices v. (Note that t_v can be ∞ if v is not reachable by mouse Gehr and $-\infty$ if it is reachable by mouse Gehr but not by the CPA agents.)

Observe that if the agents are dispatched at a certain time that allows them to still catch mouse Gehr, they are either able to reach the cake strictly before mouse Gehr, or they will be able to block all vertices surrounding the cake at or before the time that mouse Gehr can reach any of them. (Because if they can catch mouse Gehr earlier than at the surrounding vertices, they could also block the surrounding vertices in time by moving there on shortest paths.)

Let $\Gamma^-(v)$ denote the set of vertices from which v can be reached by following a single edge in E. The final result is $\max(t_{\text{cake}}, \min_{v \in \Gamma^-(v_{\text{cake}})} t_v)$. (The agents can choose to either block the cake or the vertices surrounding the cake, and they will choose the option that allows them to be dispatched later.)

```
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;
      }
    }
  }
  int worst = INFINITY;
  for (int v : gamma) {
    worst = min(worst, t[v]);
  }
  return max(t[v_cake], worst);
}
```

(In this implementation, the vectors t and gamma could be elided by performing the necessary computations inline. This implementation was chosen to keep a close correspondence to the solution text.)

Subtask 3: One Agent

We can again assume without loss of generality that mouse Gehr moves on a shortest path towards v_{cake}. Additionally, we can observe that this is true also for the CPA agent: If the agent manages to catch mouse Gehr while not on a shortest path towards the cake, the agent can alternatively choose to reach v_{cake} strictly before mouse Gehr, while being dispatched at the same time. Furthermore, we can assume that if the agent catches mouse Gehr somewhere on the way, the agent moves onto the same field at precisely the same time. (Otherwise, the agent could alternatively just move to the cake instead of waiting until mouse Gehr arrives and, again, reach the cake strictly before mouse Gehr.)

There are now two cases to consider:

1. The agent catches mouse Gehr by moving to v_{cake} at a strictly earlier time.
2. The agent catches mouse Gehr somewhere on the way, by moving to the same vertex at the same time.

For case 1, the latest possible time T'_{cake} to dispatch the agent is easy to compute. The final result is either T'_{cake} or $T'_{\text{cake}} + 1$. For case 2, the agent catches mouse Gehr before he reaches the cake. We will now need to compute which one it is. It suffices to check whether the agent can catch mouse Gehr by moving onto the same vertex simultaneously.

We compute the distance of each vertex to the cake by running Dijkstra's algorithm once, starting at the cake (moving backwards along directed edges).

Let $E' \subseteq E$ denote all edges that are on a shortest path to the cake. (Those are precisely the edges that connect two vertices whose difference in distance to the cake corresponds precisely to the weight of the edge.)

We can then view the situation as a game: The game state consists of the two vertices v_g and v_a denoting the current vertex of mouse Gehr and the agent respectively. Note that the agent and mouse Gehr might be "out-of-sync", i.e. the two vertices of the agent and mouse Gehr might be at different distances from the cake. We will allow the agent to move if the agent is currently at a distance from the cake that is larger than the distance of mouse Gehr to the cake, otherwise mouse Gehr moves. (This in particular means that mouse Gehr moves first if both the agent and mouse Gehr are currently at the same distance to the cake. This corresponds to the constraint that mouse Gehr moves first if he and the agent need to make a simultaneous decision.) Both mouse Gehr and the agent only move along edges in E'.

A position is a winning position for the agent if and only if the agent can catch mouse Gehr starting from this position by moving onto the same square, unless mouse Gehr reaches the cake first.

We can set up the following recurrence describing the winning positions for the agent:

- If mouse Gehr reaches the cake, the agent loses.
- If the agent is at least as close to the cake as mouse Gehr, the current position (v_g, v_a) is a winning position for the agent if and only if all positions (v_g, v'_a) that mouse Gehr can reach by moving along an edge from E' are winning positions for the agent.
- If mouse Gehr is closer to the cake than the agent, the current position (v_g, v_a) is a winning position for the agent if and only if the agent can reach a winning position (v'_g, v_a) by moving along an edge from E'.

More formally: Let $d(v, v_{\text{cake}})$ be the distance from vertex v to the cake. We denote by $\text{win}(v_g, v_a) = 1$ the fact that from the initial position (v_g, v_a), the agent can catch mouse Gehr by moving to the same square at the same time.

Let $\Gamma'^{+}(v) \subseteq V$ denote the set of all vertices reachable from vertex v by following a single edge from E'.[2]

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

Because there are no cycles in the graph $G' = (V, E')$, we can evaluate the above recurrence using dynamic programming. It is easiest to directly encode the recurrece as a recursive function and to apply memoization.

The final result is

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

[2] A note on notation: We denote by $a \wedge b$ the boolean operation "and" and by $a \vee b$ the boolean operation "or". $\bigwedge$ and $\bigvee$ are to $\wedge$ and $\vee$ as $\sum$ is to $+$. We have $\bigwedge_{x \in \{\}} f(x) = 1$ and $\bigvee_{x \in \{\}} f(x) = 0$.

I.e. if the initial positions of mouse Gehr and the agent are a winning position for the agent in the game we have defined, the agent can be dispatched at a time such that the agent would reach the cake at the same time as mouse Gehr. Otherwise, the agent needs to leave one time unit earlier in order to reach the cake strictly before mouse Gehr.

The total running time of this solution is dominated by the dynamic programming algorithm, whose running time can be bounded by $\mathcal{O}(|V|(|V|+|E|))$. The total additional memory usage is dominated by the DP table and is therefore at most $\mathcal{O}(|V|^2)$.

```
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);
}
```