Cake Protection Agency

Subtask 1: Latest Starting Time

We can compute the times $(t_1 = T, t_2, \dots, t_{k-1}, t_{\text{cake}})$ at which mouse Gehr arrives at each of the vertices $(v_1 = v_{\text{gehr}}, v_2, \dots, v_{k-1}, v_k = v_{\text{cake}})$ 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 $(v_1 = v_{\text{gehr}}, v_2, \dots, v_{k-1})$ at time $(t_1 = T, t_2, \dots t_{k-1})$ 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 $\max_{k \in \{1,\dots,k-1\}}(t_k - d(v_{\text{cpa}}, v_k))$ 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 $t_k - d(v_{\text{cpa}}, v_k) \geq \max_{k \in \{1,\dots,k-2\}}(t_k - d(v_{\text{cpa}}, v_k))$.) 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);
}
```