Deuxième Tour Théorique

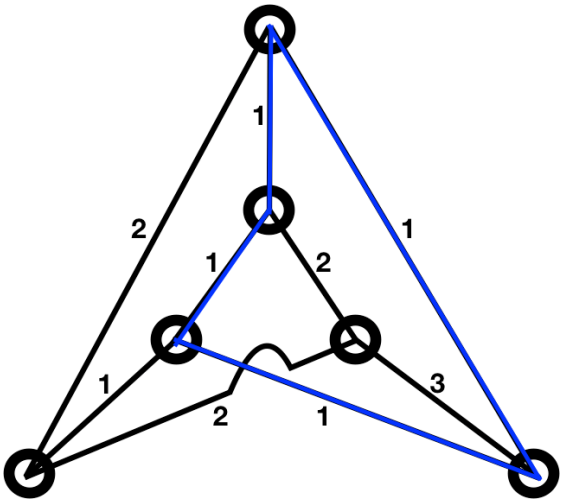
Tâches

Swiss Olympiad in Informatics

4 mars 2017

Fleptons

Sous-tâche 1 : Résous l'exemple (10 Points)



N'importe quelle paire de sommets du cercle est une solution du problème (il y en a une autre, peux-tu la trouver ?)

Sous-tâche 2 : Développe un Algorithme (50 Points)

La solution complète pour 50 points a une complexité temporelle de $O(n \cdot m \log n)$.

On commence l'algorithme de Dijkstra depuis chaque sommet et durant cette traversée du graphe, on stocke non seulement le chemin le plus court (au début, quand aucun chemin n'existe, on assigne une valeur par défaut arbitraire par exemple -1), mais en plus, pour chaque sommet, le sommet précédent dans l'algorithme de Dijkstra.

La première fois qu'un arc visite un sommet dans l'algorithme de Dijkstra qu'il avait déjà visité auparavant avec la même (plus courte) distance, on peut construire la solution. Puisqu'il y a deux chemins menant de x à y et que leur dernier arc est différent, il doit exister un sommet z où les deux chemins se séparent.

En effectuant un retour sur trace (backtracking) des deux chemins (en descendant deux fois le chemin si nécessaire), ce sommet z peut être trouvé. Il n'est certes pas garanti qu'on trouve un tel chemin en partant d'un seul sommet, toutefois, en commençant Dijkstra de chaque point, il est garanti qu'on commence aussi Dijkstra à ce sommet z et quand c'est le cas, il est garanti qu'on trouve deux tels chemins.

Dijkstra peut aussi être implémenté en $O(n \cdot \log(n) + m)$ pour obtenir une complexité en temps plus basse de $O(n^2 \cdot \log(n) + n \cdot m)$.

```
1 #include <iostream>
2 #include <queue>
3
```

```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};
      }
    }
  }
```

```
56   return {-1,-1}; //no nodes found
57 }
```

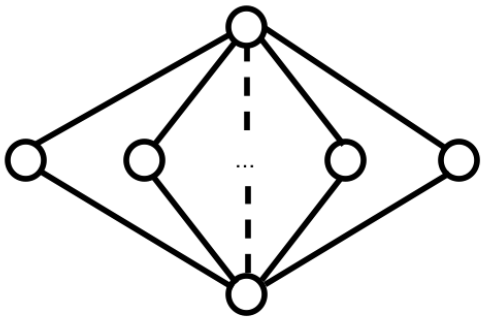
Barème

— Idée correcte / Pseudocode : 30 points
— Complexité temporelle : 10 points.
— Justesse : 5 points.
— Complexité spatiale : 5 Punkte.

Sous-tâche 3 : Configuration pour détecter Plus de Fleptons (40 Points)

Il n'y a qu'une configuration pour $n = 1, 2, 3$, le graphe complet trivial.

Pour la solution avec tous les points on peut utiliser un graphe en forme de diamant, en plaçant un sommet au-dessus et un au-dessous. Au milieu, on place les détecteurs restants en une ligne et on finit par connecter chaque sommet du milieu une fois avec le sommet du dessus et une fois avec celui du dessous. Cela nous amène à $(n-2)\cdot 2 = 2n-4$ arcs, ce qui était nécessaire pour obtenir tous les points.



Une autre solution intéressante était le graphe complet, qui satisfait la propriété, parce que tous les somemts sont adjacents entre eux.

Barème :

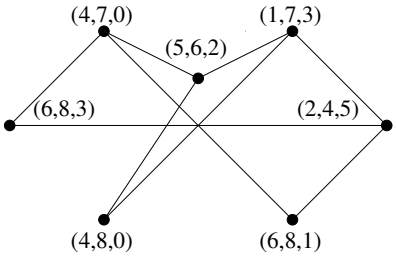
— N'importe quelle solution quadratique : 15 points.
— Solution entre linéaire et quadratique : 20 points.
— N'importe quelle solution linéaire plus coûteuse : 25 points.
— $3 \cdot N + c = 30$ points.
— $2.5 \cdot N + c = 35$ points.
— $2 \cdot N + c = 40$ points.
— $1.5 \cdot N + c = 45$ points.

Si tu as construit une configuration seulement pour un nombre pair ou impair de détecteurs, tu obtiens la moitié des points ci-dessus respectivement.

Teleport

Sous-tâche 1 : Résous un Réseau (10 Points)

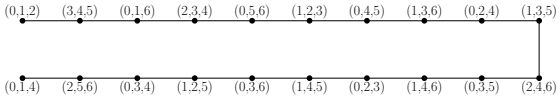
Toutes les paires de téléports disponibles entre lesquels les souris peuvent voyager directement sont montrées dans l'image suivante.



Une des plus courtes distances maximales est $d(F, G) = 3$.

Sous-tâche 2 : Réseau avec Distance Maximum (45 Points)

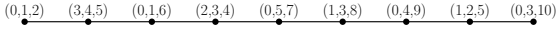
Une solution possible de cette sous-tâche, consistant en 20 téléports, est dépeinte ci-dessous. Les liens montrent toutes les paires entre lesquelles les aliens peuvent voyager directement. C'est aussi la solution maximale, qui a été trouvée par un algorithme bruteforce.



Sous-tâche 3 : Borne pour la Distance Maximale (45 Points)

La première observation est que nous sommes intéressés seulement aux réseaux qui forment une seule "chaîne" de téléports $V_0, \ldots V_n$, où on peut voyager entre tous les V_i et V_{i+1}, mais pas entre deux autres téléports, puisqu'on peut toujours enlever des téléports supplémentaires pour garder uniquement le chemin de longueur désirée.

Beaucoup de vos solutions affirment que le nombre de fréquences différentes est limité par une petite constante (entre 7 et 10). Pourtant, ce n'est pas le cas, puisqu'on peut aussi trouver une chaîne avec 11 fréquences :



L'image de la sous-tâche B montre une solution optimale possible – par un algorithme bruteforce nous avons montré qu'elle consiste de 20 téléports et donc la réponse marquant le maximum de points dans cette sous-tâche est 19.

Nous ne sommes pas au fait d'une preuve mathématique de cette borne. A la place, nous montrons une borne supérieure de la distance maximale un peu plus lâche qui obtient tous les points,

Supposons que les téléports V_0 et V_n sont les plus éloignés dans un réseau et que $d(V_0, V_n) = n$ pour un certain n et n est le plus grand possible.

Appelons les téléports sur le chemin le plus court entre V_0 et V_n séquentiellement comme $V_0, V_1, \ldots, V_{n-1}, V_n$. Dans notre preuve, nous n'aurons pas besoin d'autres téléports et nous pouvons donc les omettre, (Nous pouvons considérer le tel réseau le plus petit. Il consistera des téléports V_i, $0 \leq i \leq n$ uniquement et il sera un chemin.)

Soient $\{x_i, y_i, z_i\}$ trois fréquences distinctes de transmetteurs dans le téléport V_i. Comme V_0 et V_1 sont connectés, nous savons que $\{x_0, y_0, z_0\} \cap \{x_1, y_1, z_1\} = \emptyset$. De plus, comme V_1 n'est pas connecté avec un quelconque V_i, $3 \leq i \leq n$ (sinon nous pourrions raccourcir le chemin le plus court), nous savons que V_i utilise une des fréquences de V_1. Le même argument est valable pour V_0 aussi. Comme les fréquences de V_0 et V_1 sont distinctes, nous savons que chaque V_i, $3 \leq i \leq n$ doit utiliser au moins une fréquence de $\{x_0, y_0, z_0\}$ etune de $\{x_1, y_1, z_1\}$.

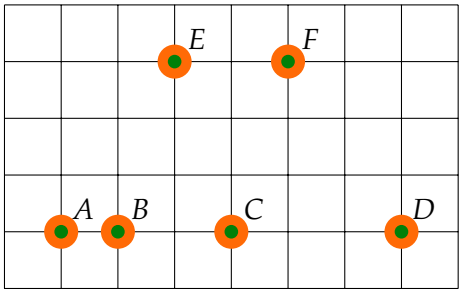
Il y a au total 9 combinaisons de fréquences dont une doit être présentedans chaque V_i, $3 \leq i \leq n$. Il existe donc une combinaisonprésente dans au moins $\frac{n-2}{9}$ téléports. [1] Sans perte de généralité, appelons cette combinaison $\{x_0, x_1\}$ et soient les $T := \frac{n-2}{9}$ téléports appelés séquentiellement comme ils apparaissent dans le chemin $V_{t_1}, V_{t_2}, \ldots V_{t_T}$. Soient les fréquences d'émission dans V_{t_i} $\{x_0, x_1, z_{t_i}\}$. Note que tous les z_{t_i} (pour $1 \leq i \leq T$) doivent être différents, sinon nous pourrions raccourcir le plus court chemin, ce qui serait une contradiction. Nous allons compter combien de fréquences distinctes z_{t_i} nous pourrions avoir dans nos téléports sur notre chemin.

Note que comme V_{t_i} et V_{t_j} partagent des fréquences d'émission, ils ne sont pas connectés et donc dans notre chemin nous devons avoir V_{t_i-1} tel que les fréquences de V_{t_i-1} et V_{t_i} sont distinctes ($1 \leq t \leq T$). Mais qu'arriverait-il si aucun z_{t_j} ($1 \leq j \leq T$, $j \neq i$) n'était une fréquence d'émission de V_{t_i-1} ? Alors les fréquences d'émission de V_{t_i-1} et V_{t_j} (pour $i \neq j$) seraient distinctes et donc V_{t_i-1} et V_{t_j} seraient connectés, causant à nouveau une contradiction. Les fréquences d'émission de V_{t_i-1} doivent donc toutes être des z_{t_j} ($i \neq j$) disponibles. Comme V_{t_i-1} peut avoir au plus 3 telles fréquences avec z_{t_i} nous pouvons avoir au plus 4 telles fréquences, ce qui implique que $\frac{n-2}{9} = T \leq 4$ ce qui nous donne une borne de $n \leq 38$.

1. Le -2 compte pour V_0, V_1 et V_2.

Jardin de Carottes

Sous-tâche 1 : Exemple



Il y a trois états finaux possibles : $\{A, B\}$, $\{E, F\}$ et $\{A, B, C, D, E, F\}$. Les deux premiers sont obtenus en partant de A, B et E, F respectivement. Avec toutes les autres combinaisons, toutes les carrotes sont finalement infectées.

Sous-tâche 2 : Algorithme Nous commençons en transformant le problème en un problème de graphe. Chaque carotte est un sommet dans le graphe non dirigé $G = (V, E)$. Chaque sommet est connecté avec chaque autre sommet et le poids de cet arc est la distance entre les deux carottes.

Nous faisons l'observation suivante : Supposons que les sommets $v_1, v_2, \ldots, v_r$ sont un état final possible et que x est la distance maximale entre les sommets dans cet état final. A quoi le graphe G ressemble-t-il si nous enlevons tous les arcs dont le poids est plus grand que x ? D'un côté, les sommets $v_1, v_2, \ldots, v_r$ ne sont pas connectés au reste du graphe d'une quelconque manière, sinon «Stoffulus »pourrait se répandre le long de cette connection et il ne s'agirait pas d'un état final. D'un autre côté, les sommets $v_1, \ldots, v_r$ sont connectés à chaque autre sommet dans cet ensemble, puisque chaque poids est plus petit que x(car x est par définition le maximum de tous ces poids).

Pour résumer, chaque état final $F = v_1, v_2, \ldots, v_r$ remplit les conditions suivantes :

— Il n'y a pas d'arc de F vers le reste du graphe($\nexists x \in F, y \in V \backslash F$ tels que $x, y \in E'$).

— Tous les sommets de F sont mutuellement connectés($\forall x, y \in F$ nous avons $x, y \in E'$).

Ces conditions sont non seulement nécessaires mais suffisantes. Cela signifie que quand nous avons un ensemble de sommets qui remplit ces conditions, nous savons qu'il s'agit d'un état final (si nous infectons 2 carottes arbitraires dans cet ensemble, nous obtenons cet ensemble comme état final). Pour trouver tous les états finaux possibles nous pouvons simplement chercher pour tous les x possibles s'il y a un sous-ensemble qui remplit ces conditions.

L'algorithme pour mener à bien cette procédure efficacement se déroule ainsi : Nous enlevons tous les arcs du graphe et les trions selon leur poids. Nous les ajoutons ensuite à nouveau en partant du plus petit et en observant les différents composants. Quand deux arcs ont le même poids, nous les insérons au même moment, c'est-à-dire que nous

ne regardons les composants qu'après avoir inséré tous les arcs du même poids. Aussitôt qu'un composant a $k \cdot (k-1)$ arcs, où k est le nombre de sommets dans ce composant, il remplit les deux conditions et nous avons trouvé un nouvel état final. Pour observer les différents composants et garder en mémoire le nombre d'arcs dans un composant, nous utilisons la structure de données union-find (cf tour 2h) où nous stockons en plus le nombre d'arcs de chaque composant.

Le tri des arcs prend $O(|E| \cdot \log |E|)$, l'initialisation du union-find prend $O(|V|)$ et le travail à chaque itération de la boucle for prend un temps constant (toutes les opérations sur un union-find sont $O(1)$). Le graphe est totalement connecté donc nous avons $|E| = \frac{|V|(|V|-1)}{2}$ et nous obtenons notre complexité temporelle totale de $O(n + n^2 \cdot \log n^2) = O(n^2 \cdot \log n)$. La complexité spatiale est $O(n^2)$.

```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";
}
```

Barème :

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

Les points ci-dessus sont distribués comme suit :
Idée/Description de l'algorithme : 50%
Pseudocode : 20%
Justesse : 20%
Analyse de la complexité : 10%

Agence de Protection des Gâteaux

Sous-tâche 1 : Dernier Temps de Départ

Nous pouvons calculer les temps $(t_1 = T, t_2, \ldots, t_{k-1}, t_{\text{cake}})$ auxquels la souris Gehr arrive à chaque sommet $(v_1 = v_{\text{gehr}}, v_2, \ldots, v_{k-1}, v_k = v_{\text{cake}})$ en additionnant les poids le long des arcs que la souris Gehr emprunte sur son chemin. L'agent de la CPA peut prévenir le plan maléfique soit s'il atteint un des sommets $(v_1 = v_{\text{gehr}}, v_2, \ldots, v_{k-1})$ au temps $(t_1 = T, t_2, \ldots t_{k-1})$, soit s'il atteint le sommet final au temps $t_k - 1$. Le temps le plus tardif de départ pour l'agent de la CPA est donc le maximum de $\max_{k\in\{1,\ldots,k-1\}}(t_k - d(v_{\text{cpa}}, v_k))$ et $t_k - 1 - d(v_{\text{cpa}}, v_{\text{cake}})$. Nous utilisons l'algorithme de Dijkstra pour calculer la distance de v_{cpa} à tous les autres sommets du graphe. Le temps d'exécution et l'usage de mémoire additionnel sont dominés par l'algorithme de Dijkstra et son donc donnés par $O(|V| \log |V| + |E|)$ et $O(|V|)$ respectivement. [1]

```
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]);
}
```

Sous-tâche 2 : Beaucoup d'Agents

Comme nous avons une réserve illimitée d'agents, la stratégie qui limite le plus Gehr, une fois les agents envoyés, est d'envoyer un agent à chaque sommet aussi vite que possible. La souris Gehr ne peut alors atteindre que les sommets que les agents ne peuvent pas atteindre avant lui (ou possiblement au même moment à l'exception de v_{cake}).

1. Note que l'implémentation de Dijkstra habituellement utilisée en pratique a une plus grande complexité théorique maximale : $O((|V| + |E|) \log |V|)$.

Notons aussi que sans perte de généralité, la souris Gehre voyagera toujours le long d'un plus court chemin. Pour voir pourquoi c'est le cas, notons qu'il est toujours meilleur pour la souris Gehr d'être à un sommet donné plus tôt que plus tard. Ceci exclut immédiatement la possibilité que Gehr se déplace en cycles. Supposons que Gehr a bougé avec succès du sommet v_{gehr} au sommet w le long d'un (simple) chemin p qui n'est pas un plus court chemin. Maintenant, considérons un chemin le plus court q du sommet v_{gehr} au sommet w. Comme par hypothpse, $p \neq q$, il y a des sommets qui sont dans q mais pas dans p. La seule raison pour laquelle la souris Gehr pourrait avoir à éviter ces sommets et prendre un plus long chemin à la place est parce que la CPA peut atteindre un sommet x parmi eux aussi tôt que la souris Gehr. Cependant, cela implique immédiatement que la CPA atteindra le sommet w strictement plus tôt que la souris Gehr (la CPA peut juste se déplacer le longd'un chemin le plus court du sommet x au sommet w). Ainsi, elle l'attrapera même si $w = v_{\text{cake}}$, ce qui signifie que la souris Gehr ne peut pas gagner quelque chose en bougeant le long d'un chemin qui n'est pas le plus court chemin.

Nous pouvons calculer pour chaque sommet v, à quel moment t_v les agents auraient besoin d'être déployés pour prévenir l'entrée de Gehr sur ce sommet, en soustrayant le temps qu'il prend aux agents de la CPA pour atteindre le sommet du temps d'arrivée le plus rapide de Gehr à ce sommet (pour $v = v_{\text{cake}}$, nous soustrayons en plus 1). Le résultat correct sera t_v pour un des sommets v.

Le problème est maintenant le suivant : Trfouve le plus grand t_v tel que si on enlève tous les sommets w avec $t_w \geq t_v$ du graphe, le gâteau n'est pas atteignable depuis l'emplacement initial de Gehr. Nous pouvons calculer l'atteignabilité dans un graphe donné en $O(|V| + |E|)$, et l'atteignabilité est monotone dans le temps de déploiement de l'agent, donc nous pouvons trier les temps candidats et utiliser la recherche binaire pour trouver le temps correct.

Le temps d'exécution total est $O((|V| + |E|) \log |V|)$. L'usage de mémoire additionnel est encore $O(|V|)$. Le temps d'exécution peut être réduit à $O(|V| \log |V| + |E|)$ en utilisant un algorithme d'atteignabilité $s - t$ incrémentaire au lieu de la recherche binaire. (La recherche devient linéaire, mais le temps total dépensé pour les tests d'atteignabilité passe à $O(|V| + |E|)$.)

```
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;
      }
    }
  }
```

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

Sous-tâche 3 : Un Agent

Nous pouvons à nouveau supposer sans perte de généralité que la souris Gehr se déplace sur un chemin le plus court en direction de v_{cake}. De plus, nous pouvons observer que c'est aussi vrai pour l'agent de la CPA : S'il agent arrive à attraper la souris Gehr en n'étant pas sur un chemin le plus court vers le gâteau, l'agent peut alternativement choisir d'atteindre v_{cake} strictement avant Gehr, tout en étant déployé au même moment. De plus, nous pouvons supposer que si l'agent attrape la souris Gehr quelque part sur le chemin, l'agent se déplace vers le même endroit au même moment précisément. (Sinon, l'agent pourrait alternativement juste se déplacer vers le gâteau au lieu d'attendre que la souris Gehr arrive et, encore une fois, atteindre le gâteau strictement avant Gehr.)

Il y a maintenant deux cas à considérer :

1. L'agent attrape la souris Gehr en se déplaçant vers v_{cake} à un temps strictement plus tôt.
2. L'agent attrape la souris Gehr quelque part sur le chemin, en se déplaçant sur le même sommet au même instant.

Dans le premier cas, le temps T'_{cake} le plus tardif possible pour déployer l'agent est facile à calculer. Le résultat final est soit T'_{cake}, soit $T'_{\text{cake}} + 1$. Dans le deuxième cas, l'agent attrape Gehr avant qu'il n'atteigne le gâteau. Nous allons maintenant avoir besoin de calculer lequel c'est. Il suffit de vérifier si l'agent peut attraper la souris Gehr en se déplaçant vers le même sommet simultanément.

Nous calculons la distance de chaque sommet au gâteau en exécutant l'algorithme de Dijkstra une fois, en partant du gâteau (en reculant le long des arcs dirigés).

Supposons que $E' \subseteq E$ dénote tous les arcs qui sont sur un chemin le plus court vers le gâteau. (Il s'agit précisément des arcs qui connectent deux sommets dont la différence en distance au gâteau correspond précisément au poids de l'arc.)

Nous pouvons ensuite voir la situation comme un jeu : L'état de jeu consiste en deux sommets v_g et v_a dénotant le sommet actuel de la souris Gehr et de l'agent respectivement. Note que l'agent et la souris Gehr peuvent être "out-of-sync", i.e. les deux sommets de l'agent et de la souris Gehr peuvent être à des distances différentes du gateau. Nous autoriserons l'agent à bouger si l'agent est actuellement à une distance du gâteau qui est plus grande que la distance de la souris Gehr au gâteau, sinon la souris Gehr bouge. (Ceci, en particulier, signifie que la souris Gehr bouge en premier si l'agent et la souris Gehr sont actuellement à la même distance du gâteau. Cela correspond à la contrainte que Gehr bouge en premier si lui et l'agent doivent prendre une décision simultanlée.) Gehr et l'agent ne bougent tous deux que le long d'arcs dans E'.

Une position est gagnante pour l'agent si et seulement si l'agent peut attraper la souris

Gehr en partant de cette position en se déplaçant vers le même carré, à moins que Gehr atteigne la gâteau en premier.

On peut mettre en place la récurrence suivante décrivant les positions gagnantes pour l'agent :

— Si la souris Gehr atteint le gâteau, l'agent perd.
— Si l'agent est au moins aussi proche du gâteau que la souris Gehr, la position actuelle (v_g, v_a) est gagnante pour l'agent si et seulement si toutes les positions (v_g, v'_a) que la souris Gehr peut atteindre en se déplaçant le long d'un arc de E' sont des positions gagnantes pour l'agent.
— Si la souris Gehr est plus proche du gâteau que l'agent, la position actuelle (v_g, v_a) est gagnante pour l'agent si et seulement si l'agent peut atteindre une position gagnante (v'_g, v_a) en bougeant le long d'un arc de E'.

Plus formellement : Soit $d(v, v_{\text{cake}})$ la distance du sommet v au gâteau. Nous appelons $\text{win}(v_g, v_a) = 1$ le fait que depuis la position initiale (v_g, v_a), l'agent peut attraper Gehr en se déplaçant sur le même carré au même moment.

Soit $\Gamma'^{+}(v) \subseteq V$ l'ensemble de tous les sommets atteignables du sommet v en suivant un seul arc de E'. [2]

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

Puisqu'il n'y a pas de cycles dans le graphe $G' = (V, E')$, nous pouvons évaluer la récurrence ci-dessus en utilisant la programmation dynamique. Il est plus faciler d'encoder directement la récurrence comme une fonction récursive et d'appliquer la mémoisation.

Le résultat final est

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

I.e. si les positions initiales de la souris Gehr et de l'agent sont une position gagnante pour l'agent dans le jeu que nous avons défini, l'agent peut être déployé à un temps tel que l'agent atteindrait le gâteau au même moment que la souris Gehr. Sinon, l'agent doit

2. Une note sur la notation : Nous appelons $a \wedge b$ l'opération booléenne "and" et $a \vee b$ l'opération booléenne "or". $\bigwedge$ et $\bigvee$ sont à $\wedge$ et à $\vee$ ce que $\sum$ est à $+$. Nous avons $\bigwedge_{x \in \{\}} f(x) = 1$ et $\bigvee_{x \in \{\}} f(x) = 0$.

partir une unité de temps plus tôt pour atteindre le gâteau strictement avant la souris Gehr.

Le temps d'exécution total de cette solution est dominé par l'algorithme de programmation dynamique, dont le temps peut être borné par $O(|V|^3)$. L'usage additionnel de mémoire est dominé par le tableau DP et est donc au plus $O(|V|^2)$.

```
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);
```

```
46   return T + d_gehr[v_cake] - d_cpa[v_cake] - 1 + win(v_gehr, v_cpa);
47 }
```