Migwel

Geste de savoir

Un Secret Santa Hamiltonien

23 octobre 2020

Table des matières

Il y a deux ans, je célébrais Noël avec 10 amis et nous avions décidé de s'offrir des cadeaux et pour ce faire, nous avons organisé un "Secret Santa". Pour ceux qui ne connaissent pas Secret Santa, l'idée est très simple: avant Noël, un père Noël secret est attribué à chaque invité. Ce père Noël aura la tâche de trouver un cadeau pour cette personne et de lui offrir lors de la fête de Noël. Le jour du réveillon, on choisit une personne au hasard et celle-ci offre son cadeau à la personne qui lui a été attribuée puis cette personne fait de même et ainsi de suite.

Donc, comme je le disais, nous avions décidé d'organiser un Secret Santa mais il s'est passé quelque chose d'horrible : il y avait une sous-boucle dans la distribution! J'ai commencé en donnant mon cadeau à une personne A qui a continué en donnant son cadeau à une personne B qui a continué en donnant son cadeau à… moi. Cela signifiait donc qu'on allait devoir choisir une autre personne au hasard pour continuer la distribution entre les 7 personnes restantes.

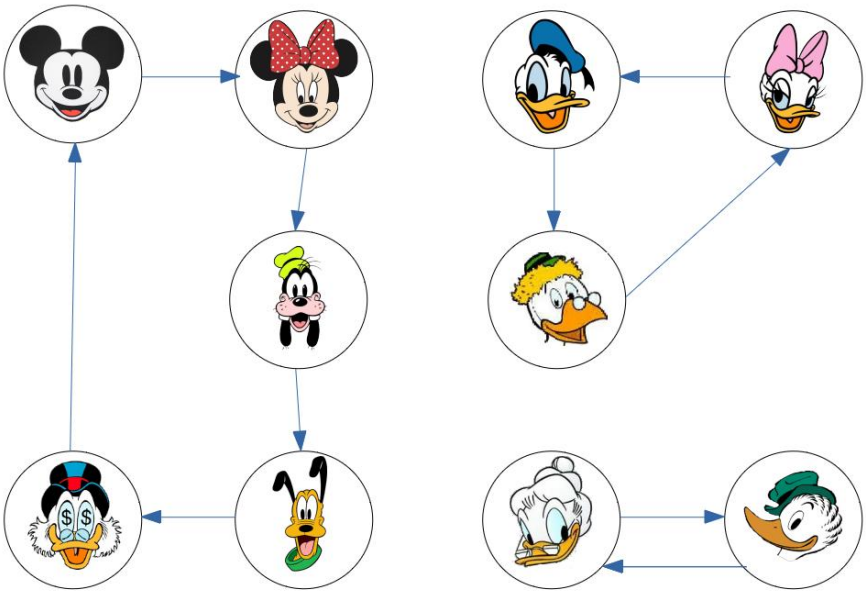

FIGURE 0.1. – Mauvaise distribution

L'année dernière, j'ai décidé de ne pas laisser cette situation se reproduire. Voyons comment écrire une application qui attribue des pères Noëls secrets à l'aide de notre meilleure amie : les mathématiques.

1. Ce que nous disent les maths

1.0.1. La théorie des graphes

Une discipline passionnante des maths est la théorie des graphes. Elle permet de représenter des problèmes complexes sous forme de graphes.

Un graphe, c'est une structure, orientée ou non, composée de sommets et d'arêtes. Un exemple typique est d'utiliser un graphe pour représenter des voyages possibles entre destination. Dans ce cas, les sommets sont des villes et les arêtes représentent s'il est possible de se rendre de façon direct de la ville A à la ville B. Ceci est représenté dans l'illustration ci-dessous.

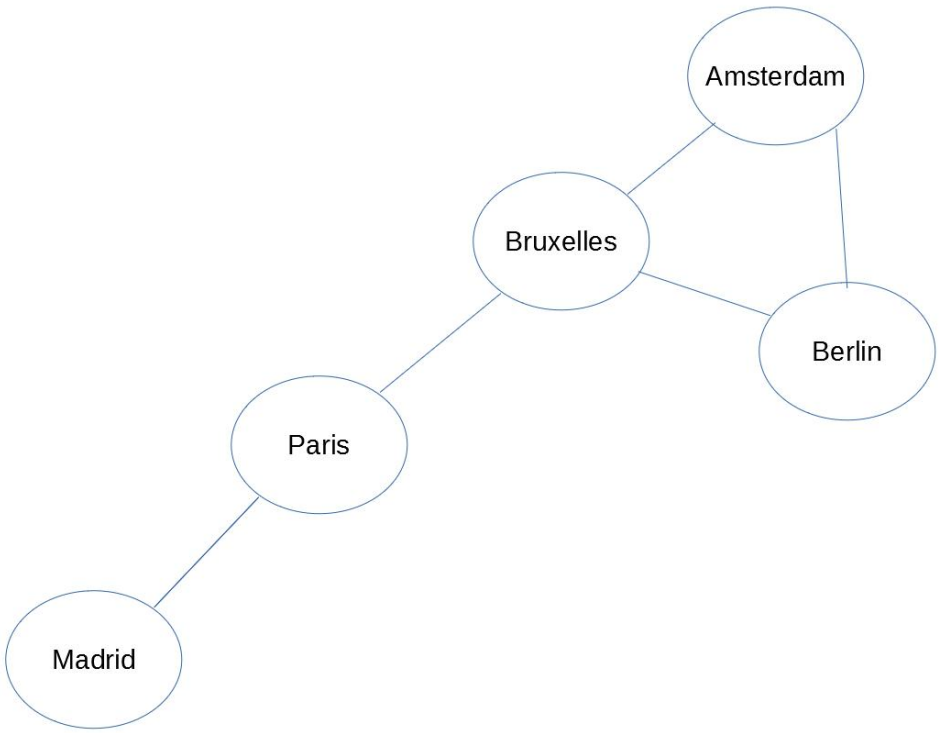

FIGURE 1.2. – Graphe de villes

Dans cet exemple, il est possible de se rendre directement de Paris à Madrid mais pas de Paris à Berlin. Pour ce faire, il faut d'abord transiter par Bruxelles. Il est également possible de donner une valeur aux arêtes. Dans notre exemple, cela pourrait représenter la distance entre les villes. Ensuite, nous pourrions essayer de trouver le chemin le plus court entre deux villes.

Comme mentionné plus haut, une graphe peut être orienté ou non-orienté. L'exemple des villes est non-orienté: une arête existe entre Paris et Bruxelles, cela signifie qu'il est possible de se rendre de Paris à Bruxelles mais également de faire le chemin inverse, de Bruxelles à Paris. On pourrait imaginer rendre ce graphe orienté, ce qui représenterait des routes à sens unique par exemple.

Très bien, maintenant que nous avons un peu discuter de la théorie, essayons de l'appliquer à notre cas. Pour le Secret Santa, nos sommets seront les participants et les arêtes représenteront l'acte de donner un cadeau à quelqu'un. Il s'agira donc d'un graphe orienté car on veut savoir qui est la personne qui donne le cadeau et qui est celle qui le reçoit. Enfin, on ne donnera aucun poids à nos arêtes étant donné qu'on donne toujours un cadeau ou non. On ne donne jamais de demi-cadeau.

1.0.2. Hamilton à la rescousse

Des nombreux mathématiciens se sont intéressés aux graphes et celui qui va nous être utile ici est William Rowan Hamilton. En effet, un graphe est dit hamiltonien si il possède un cycle hamiltonien. Un cycle hamiltonien peut être décomposé en deux parties :

- Cycle : Un chemin est considéré comme un cycle si le sommet duquel le chemin débute est également celui auquel il se termine
- Hamiltonien : un chemin est hamiltonien s'il ne passe qu'une et une seule fois par chaque sommet

La bonne nouvelle, c'est qu'il s'agit exactement de ce que nous voulons ! On veut que notre chemin implique tous les participants et ceux-ci ne doivent donner (et recevoir) qu'un seul cadeau. La mauvaise, c'est qu'il n'existe pas d'algorithme qui permette de trouver ce cycle hamiltonien en un temps polynomial.

2. Implémentation

2.0.1. Représentation du graphe

Maintenant que nous connaissons la théorie, voyons comment nous pouvons implémenter une solution en commençant pas la représentation. Une méthode typique pour représenter un graphe dirigé est d'utiliser un array `tab` à 2 dimensions où la valeur `tab[x][y]` représente si une arête existe depuis x vers y. Dans notre cas, nous utiliserons deux valeurs :

- `POSSIBLE` si une arête peut être tracée de x vers y
- `IMPOSSIBLE` si il n'existe pas d'arêtes de x vers y
  Ceci est représenté dans l'image ci-dessous où une case blanche représente l'état `POSSIBLE` et une case rouge l'état `IMPOSSIBLE`.

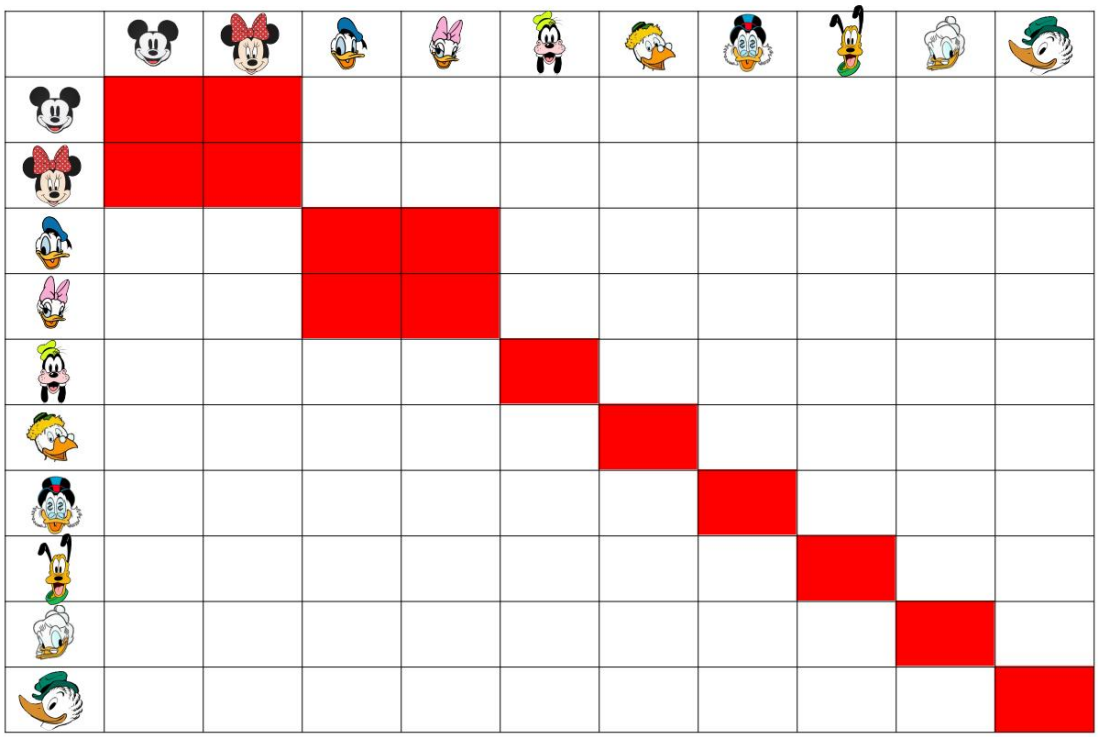

FIGURE 2.3. – Représentation sous forme de tableau

La diagonale est uniquement composée d'états impossible car personne ne peut être son propre Secret Santa. J'ai aussi ajouté une contrainte que les couples ne peuvent pas être le Secret Santa l'un de l'autre afin d'ajouter quelque contraintes (et je trouvais cela plus amusant). On pourrait très bien imaginer plus ou moins de contraintes (par exemple, si Donald était le Secret Santa de Pluto l'année dernière, on pourrait l'interdire cette année).

2.0.2. Trouver un cycle hamiltonien

Place maintenant à l'algorithme en tant que tel. Comme mentionné plus tôt, il est impossible de trouver un cycle hamiltonien en un temps polynomial. Il existe des algorithmes optimisés qui permettent de limiter le temps nécessaire à sa résolution mais si l'on se limite à un nombre limité d'amis avec qui passer Noël, une simple approche exhaustive fera l'affaire, d'autant plus que notre graphe comporte énormément d'arêtes.

Voici donc notre approche :

1. Choisir un sommet au hasard, l'ajouter au cycle. Il s'agit de notre sommet de départ
2. Pour ce sommet, choisir aléatoirement un autre sommet accessible.
3. Ajouter ce nouveau sommet au cycle :
   — Si ce sommet a déjà été visité, le retirer du cycle et retourner à l'étape 2.

— Si il n'a pas encore été visité et qu'il s'agit du dernier sommet, c'est terminé on a trouvé notre cycle.
— Sinon, répéter l'étape deux à partir du nouveau sommet

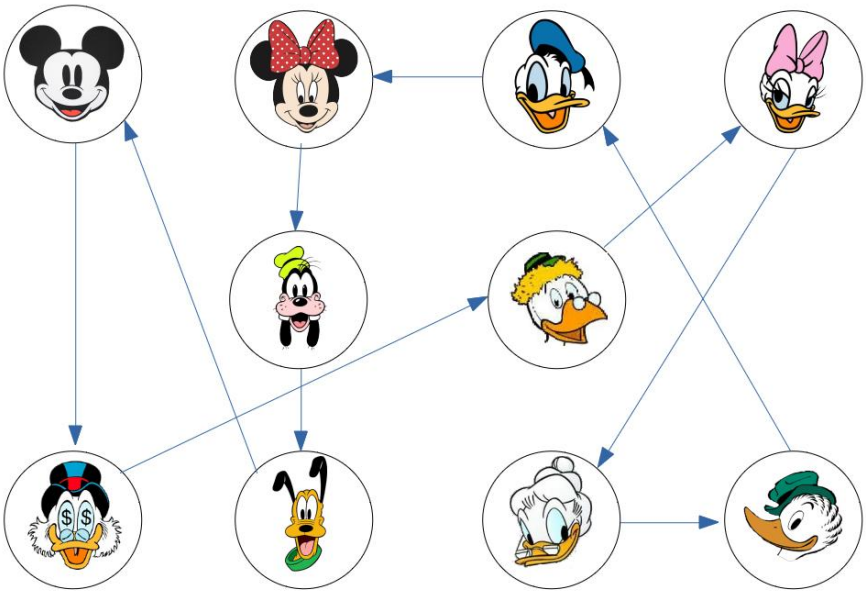

FIGURE 2.4. – Graphe avec cycle hamiltonien

2.0.3. Exemple en Java

Voyons un exemple d'une implémentation en Java.

```java
public class Participant {

    private String name;
    private List<String> exclusionList;
//...
}
```

```java
public class SecretSantaService {

```

```
3       public enum Status {
4           Possible,
5           Impossible,
6           ;
7       }
8
9       public List<Participant> computeHamiltonCycle(List<Participant>
            participants) {
10          Map<Participant, Map<Participant, Status>> giftsTable =
                buildGiftsTable(participants);
11          ArrayList<Participant> hamiltonCycle = new ArrayList<>();
12          hamiltonCycle.add(participants.get(0));
13          if (!computeHamiltonCycle(giftsTable, hamiltonCycle,
                participants.get(0))) {
14              throw new
                    IllegalArgumentException("Could not compute a proper secret sa
15          }
16
17          return hamiltonCycle;
18      }
19
20      private Map<Participant, Map<Participant, Status>>
            buildGiftsTable(List<Participant> participants) {
21          Map<Participant, Map<Participant, Status>> giftsTable = new
                HashMap<>();
22          for(Participant givingParticipant : participants) {
23              Map<Participant, Status> statusMap = new HashMap<>();
24              for(Participant receivingParticipant : participants) {
25                  if (givingParticipant.getExclusionList() != null &&
                        givingParticipant.getExclusionList().contains(receivingPar
                        {
26                      statusMap.put(receivingParticipant,
                            Status.Impossible);
27                  } else {
28                      statusMap.put(receivingParticipant,
                            Status.Possible);
29                  }
30              }
31              giftsTable.put(givingParticipant, statusMap);
32          }
33          return giftsTable;
34      }
35
36      private boolean computeHamiltonCycle(Map<Participant,
            Map<Participant, Status>> giftsTable, List<Participant>
            hamiltonCycle, Participant givingParticipant) {
37          if(hamiltonCycle.size() == giftsTable.keySet().size() &&
                giftsTable.get(givingParticipant).get(hamiltonCycle.get(0))
                == Status.Possible) {
38              //We are done
```

```
39              return true;
40          }
41
42          List<Participant> possibleReceivingParticipants =
                getPossibleReceivingParticipants(giftsTable.get(givingParticipant)
                hamiltonCycle);
43          while(possibleReceivingParticipants.size() > 0) {
44              int receivingParticipantIdx = getRandomNumberInRange(0,
                    possibleReceivingParticipants.size() - 1);
45              Participant receivingParticipant =
                    possibleReceivingParticipants.get(receivingParticipantIdx);
46              hamiltonCycle.add(receivingParticipant);
47              if (computeHamiltonCycle(giftsTable, hamiltonCycle,
                    receivingParticipant)) {
48                  return true;
49              }
50
51              hamiltonCycle.remove(receivingParticipant);
52
                    possibleReceivingParticipants.remove(receivingParticipantIdx);
53          }
54
55          return false;
56      }
57
58      private List<Participant>
            getPossibleReceivingParticipants(Map<Participant,SecretSantaService.Sta
            receivingParticipants, List<Participant> hamiltonCycle) {
59          return receivingParticipants.entrySet()
60                  .stream()
61                  .filter(e -> e.getValue()
                        !=SecretSantaService.Status.Impossible &&
                        !hamiltonCycle.contains(e.getKey()))
62                  .map(Map.Entry::getKey)
63                  .collect(Collectors.toList());
64      }
65
66      private int getRandomNumberInRange(int min, int max) {
67
68          if(min == max) {
69              return min;
70          }
71
72          if (min > max) {
73              throw new
                    IllegalArgumentException("max must be greater than min");
74          }
75
76          Random r = new Random();
77          return r.nextInt((max - min) + 1) + min;
```

```
78      }
79  }
```

Tout commence par un appel à la méthode `computeHamiltonCycle(List<Participant> participants)` à laquelle on passe une liste de Participants. Chaque Participant est constitué d'un nom et d'une liste d'exclusions (afin d'exclure les couples par exemple).

La méthode `computeHamiltonCycle(List<Participant> participants)` va tout d'abord créer notre graphe (`giftsTable`) sur base des Participants reçus puis va appeler la méthode `computeHamiltonCycle(Map<Participant, Map<Participant, Status> giftsTable, List<Participant> hamiltonCycle, Participant givingParticipant)` en lui fournissant le graphe fraichement créé, une le cycle hamiltonien (initialement uniquement composé du sommet de départ) ainsi que le premier participant duquel le cycle débutera.

Dans cette nouvelle méthode, on vérifie tout d'abord si on n'a pas déjà atteint la fin de notre calcul. Évidemment, pour la première itération, c'est un peu inutile mais comme nous allons avoir des appels récursifs, cela servira de condition d'arrêt.

Ensuite, on choisit une arête au hasard dans les arêtes disponibles ayant un sommet pas encore visité et on la suit afin d'atteindre le prochain sommet. On appelle alors notre méthode à nouveau en utilisant ce nouveau sommet. Si cet appel échoue (retourne `false`), on retire le sommet du cycle et on en choisit un autre aléatoirement.

Enfin, une fois que notre chemin est de longueur égale au nombre de participants, on vérifie que nous pouvons bien "fermer" le cycle en vérifiant que notre dernier sommet a le droit de donner un cadeau au sommet initial. Si c'est le cas, notre algorithme est terminé et nous avons trouvé notre cycle hamiltonien.

2.0.4. Créer un webservice

Afin de partager ce service avec le monde afin d'éviter à tous d'avoir des Secret Santa non-optimaux, on peut facilement utiliser Spring Boot et créer un Controller appelant notre service.

```
1  @Controller
2  @RequestMapping("/")
3  public class SecretSantaController {
4
5      private final SecretSantaService secretSantaService;
6
7      public SecretSantaController(SecretSantaService
           secretSantaService) {
8          this.secretSantaService = secretSantaService;
9      }
10
11     @RequestMapping(method= RequestMethod.GET)
12     public SecretSantaResponse getSecretSanta(SecretSantaRequest
           request) {
```

```
13          var hamiltonCycle =
                secretSantaService.computeHamiltonCycle(request.getParticipants())
14          List<SecretSanta> secretSantas =
                buildSecretSantas(hamiltonCycle);
15          return new SecretSantaResponse(secretSantas);
16      }
17
18      private List<SecretSanta> buildSecretSantas(List<Participant>
            hamiltonCycle) {
19          List<SecretSanta> secretSantas = new ArrayList<>();
20          Participant previousParticipant =
                hamiltonCycle.get(hamiltonCycle.size() - 1);
21          for (Participant participant : hamiltonCycle) {
22              secretSantas.add(new SecretSanta(previousParticipant,
                    participant));
23              previousParticipant = participant;
24          }
25          return secretSantas;
26      }
27  }
```

Un exemple de ceci est disponible à l'adresse https://secretsanta.migwel.dev/ .

2.0.5. Test

Afin de vérifier que tout ceci fonctionne correctement, on peut écrire un test en utilisant JUnit.

```
1       @Test
2       void testGetValidSecretSanta() throws IOException {
3           InputStream inStream =
                this.getClass().getClassLoader().getResourceAsStream("validsecretsa
4           SecretSantaRequest request = new
                ObjectMapper().readValue(inStream,
                SecretSantaRequest.class);
5           SecretSantaResponse response =
                controller.getSecretSanta(request);
6           List<SecretSanta> secretSantas =
                response.getSecretSantas();
7           assertEquals(request.getParticipants().size(),
                secretSantas.size());
8
9           Set<Participant> givingParticipant = new HashSet<>();
10          Set<Participant> receivingParticipant = new HashSet<>();
11          Participant previousReceiver =
                secretSantas.get(secretSantas.size() -
                1).getReceiver();
12          for(SecretSanta secretSanta : secretSantas) {
```

```
13              assertEquals(previousReceiver,
                    secretSanta.getSecretSanta());
14
                    assertFalse(givingParticipant.contains(secretSanta.getSecretSan
15
                    assertFalse(receivingParticipant.contains(secretSanta.getRecei
16              givingParticipant.add(secretSanta.getSecretSanta());
17              receivingParticipant.add(secretSanta.getReceiver());
18              previousReceiver = secretSanta.getReceiver();
19          }
20      }
```

Aussi, il est important de tester que si une requête est invalide (par exemple, si un des participants ne veut donner de cadeau à personne), le service doit échouer.

```
1       @Test
2       void testGetInvalidSecretSanta() throws IOException {
3           InputStream inStream =
                this.getClass().getClassLoader().getResourceAsStream("invalidsecret
4           SecretSantaRequest request = new
                ObjectMapper().readValue(inStream,
                SecretSantaRequest.class);
5           assertThrows(IllegalArgumentException.class, () ->
                controller.getSecretSanta(request));
6       }
```

Nous voici donc prêts à aborder les fêtes de fin d'année comme il se doit, sans avoir à craindre les horribles sous-boucles de Noël. Si vous souhaitez faire tourner votre propre instance, les sources sont disponibles sur Github .