Algorithme_genetique_pour_generer_des_eclairages_modelisation_agr

May 21, 2019

1 Table of Contents

2 Algorithme génétique pour générer des éclairages - texte de modélisation agrég

2.1 Préparation à l'agrégation - ENS de Rennes, 2018-19

- *Date* : 12 mars 2019
- *Auteur* : Lilian Besson
- *Texte*: Annale 2012, "Éclairage graphe" (public2012-D1)

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un texte d'annale de l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en Python 3.

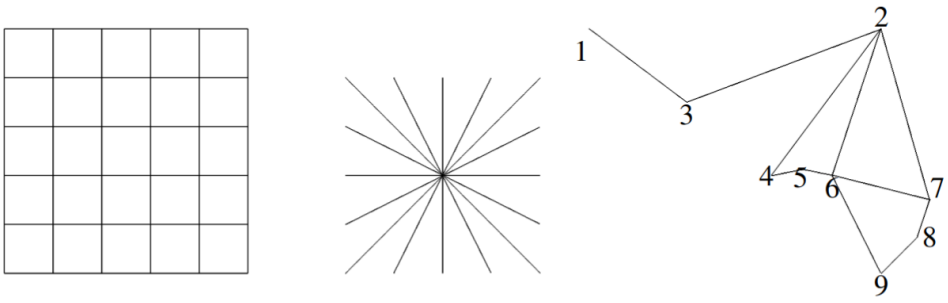

FIG. 1. trois exemples de ville

Graphes de la Figure 1 du texte

2.3 Graphes, éclairages et structures de données

On donne tout de suite un exemple de graphe, en prenant le 3ème exemple de la Figure 1 du texte.

On va définir ce graphe, comme une liste d'arêtes, et plusieurs éclairages.

```
In [13]: graphe1 = [(1,3), (3,2), (2,4), (2,6), (2,7), (4,5), (5,6), (6,7), (6,9), (7,8), (8,9
         graphe1 = [ (u-1, v-1) for (u,v) in graphe1 ]
```

```
In [14]: def nbsommets(graphe):
             n = 0
             for (u, v) in graphe:
                 if u > n or v > n: n = max(u, v)
             return n + 1
```

```
In [15]: nbsommets(graphe1)
```

```
Out[15]: 9
```

Quatre exemples d'éclairages, deux satisfaisant donc l'un trivialement, et les deux autres non satisfaisants :

```
In [16]: eclairage1_sat = [0, 1, 2, 3, 4, 5, 6, 7, 8]  # trivialement valide car on éclaire to
         eclairage2_sat = [1, 2, 3, 5, 6, 8]  # valide mais on éclaire pas tout
```

```
In [17]: eclairage1_nonsat = [2, 4, 8]
         eclairage2_nonsat = [1, 2, 3, 5, 6]
```

2.4 Fonctions nécessaires pour l'algorithme génétique

On va devoir implémenter des fonctions réalisants les tâches suivantes :

- vérifier qu'un ensemble de sommets éclairés est un éclairage valide,
- vérifier qu'un éclairage donné sous forme de tableau de `gauche`, `droite`, `lesdeux` est un éclairage valide,
- compter le nombre de places éclairés pour un éclairage donné sous la forme précédente (c'est la fonction de coût, ou "fitness" de l'algorithme génétique).

Ensuite pour l'initialisation de l'algorithm génétique il nous faudra :

- générer un éclairage aléatoirement,
- faire ça 100 fois pour avoir une population initiale.

Et puis pour chaque étape de l'algorithme génétique, on transformera les 100 individus

- trier une population d'éclairages suivant un critère (= le nombre de lampadaires utilisés),
- faire muter aléatoirement 4 éclairages parmi les 48 meilleurs,
- se faire reproduire les 48 moins bons individus restants par "crossing over", ou mutation croisée.

2.4.1 Validité d'un éclairage

```
In [22]: def est_eclairage(graphe, places_eclairees):
             """ Cette fonction est en O(M + L) = O(M) où M est le nombre d'arêtes, et L le no
             """
             n = nbsommets(graphe)
             sont_eclairees = [ False for _ in range(n) ]
             for p in places_eclairees:
                 sont_eclairees[p] = True
             for (u, v) in graphe:
                 if not sont_eclairees[u] and not sont_eclairees[v]:
                     return False
             return True

In [23]: est_eclairage(graphe1, eclairage1_sat)

Out[23]: True

In [24]: est_eclairage(graphe1, eclairage2_sat)

Out[24]: True

In [25]: est_eclairage(graphe1, eclairage1_nonsat)

Out[25]: False

In [26]: est_eclairage(graphe1, eclairage2_nonsat)

Out[26]: False
```

2.4.2 Minimalité d'un éclairage

```
In [35]: def places_moins_une(places_eclairees, place_a_enlever):
             """ En O(L) si L est le nombre de places éclairées."""
             return [place for place in places_eclairees if place != place_a_enlever]

         def est_minimal(graphe, places_eclairees):
             """ Cette fonction est en O(M L) où M est le nombre d'arêtes, et L le nombre de p
```

```
            """
            return est_eclairage(graphe, places_eclairees) and not(all([
                est_eclairage(graphe, places_moins_une(places_eclairees, place_a_enlever))
                for place_a_enlever in places_eclairees
            ]))
```

```
In [36]: est_minimal(graphe1, eclairage1_sat)
```

```
Out[36]: False
```

```
In [37]: est_minimal(graphe1, eclairage2_sat)
```

```
Out[37]: True
```

```
In [38]: est_minimal(graphe1, eclairage1_nonsat)
```

```
Out[38]: False
```

```
In [39]: est_minimal(graphe1, eclairage2_nonsat)
```

```
Out[39]: False
```

2.4.3 Conversion entre liste de valeurs ternaires et éclairage

Si le graphe $G = (V, E)$ est donné par un tableau de ses rues $E = \{(u, v)\}$, on représente une liste de places éclairées V' par un tableau de valeurs ternaires `gauche`, `droite` ou `lesdeux`.

```
In [83]: gauche, droite, lesdeux = "G", "D", "2"
```

```
In [84]: def eclairage_vers_ternaires(graphe, places_eclairees):
             """ O(M)"""
             n = nbsommets(graphe)
             ternaires = []
             sont_eclairees = [ False for _ in range(n) ]
             for p in places_eclairees:
                 sont_eclairees[p] = True
             for (u,v) in graphe:
                 if sont_eclairees[u] and sont_eclairees[v]:
                     ternaires.append(lesdeux)
                 elif sont_eclairees[u]:
                     ternaires.append(gauche)
                 elif sont_eclairees[v]:
                     ternaires.append(droite)
             return ternaires
```

```
In [85]: def ternaires_vers_eclairage(graphe, ternaires):
             """ O(M)"""
             n = nbsommets(graphe)
             places_eclairees = [ False for _ in range(n) ]
             for (u,v), ternaire in zip(graphe, ternaires):
```

```
                if ternaire == gauche or ternaire == lesdeux:
                    places_eclairees[u] = True
                if ternaire == droite or ternaire == lesdeux:
                    places_eclairees[v] = True
            # O(N)
            eclairage = []
            for i, p in enumerate(places_eclairees):
                if p: eclairage.append(i)
            return eclairage
```

```
In [94]: def est_valide_ternaires(graphe, ternaires):
             """ O(M)"""
             return est_eclairage(graphe, ternaires_vers_eclairage(graphe, ternaires))
```

```
In [87]: graphe1
```

```
Out[87]: [(0, 2),
          (2, 1),
          (1, 3),
          (1, 5),
          (1, 6),
          (3, 4),
          (4, 5),
          (5, 6),
          (5, 8),
          (6, 7),
          (7, 8)]
```

```
In [139]: ternaires1_sat = eclairage_vers_ternaires(graphe1, eclairage1_sat)
          ternaires1_sat
```

```
Out[139]: ['2', '2', '2', '2', '2', '2', '2', '2', '2', '2', '2']
```

```
In [89]: print(eclairage1_sat)
         ternaires_vers_eclairage(graphe1, eclairage_vers_ternaires(graphe1, eclairage1_sat))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
Out[89]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
In [141]: ternaires2_sat = eclairage_vers_ternaires(graphe1, eclairage2_sat)
          ternaires2_sat
```

```
Out[141]: ['D', '2', '2', '2', '2', 'G', 'D', '2', '2', 'G', 'D']
```

```
In [95]: print(eclairage2_sat)
         print(eclairage_vers_ternaires(graphe1, eclairage2_sat))
         print(ternaires_vers_eclairage(graphe1, eclairage_vers_ternaires(graphe1, eclairage2_s
         print(est_valide_ternaires(graphe1, eclairage_vers_ternaires(graphe1, eclairage2_sat)
```

```
[1, 2, 3, 5, 6, 8]
['D', '2', '2', '2', '2', 'G', 'D', '2', '2', 'G', 'D']
[1, 2, 3, 5, 6, 8]
True
```

2.4.4 Fonction de coût

On a donc la "fonction de coût" recherchée :

```
In [106]: def nb_places_eclairees(graphe, ternaires):
              """ O(M)"""
              eclairage = ternaires_vers_eclairage(graphe, ternaires)
              return len(eclairage)
```

2.4.5 Génération aléatoire d'un individu

```
In [150]: import random

          def un_ternaire_aleatoire():
              """ O(1)"""
              return random.choice([gauche, droite, lesdeux])

          def un_ternaire_aleatoire_different(valeur):
              """ O(1)"""
              if valeur == gauche:
                  return random.choice([droite, lesdeux])
              elif valeur == droite:
                  return random.choice([gauche, lesdeux])
              else:
                  return random.choice([gauche, droite])

          def un_individu(graphe):
              """ O(M)"""
              return [un_ternaire_aleatoire() for (u,v) in graphe]
```

On peut facilement générer dix individus différents, qui sont tous des éclairages valides, et afficher leur coût :

```
In [108]: for _ in range(10):
              ternaires = un_individu(graphe1)
              assert est_valide_ternaires(graphe1, ternaires)
              cout = nb_places_eclairees(graphe1, ternaires)
              print("L'éclairage", ternaires, "a un coût =", cout)
```

```
L'éclairage ['D', '2', '2', '2', 'D', 'D', '2', 'G', 'D', '2', '2'] a un coût = 8
L'éclairage ['2', 'D', 'G', 'G', 'G', 'D', '2', 'G', 'G', 'D', '2'] a un coût = 7
L'éclairage ['D', 'G', 'G', 'G', 'G', '2', 'G', 'D', '2', '2', '2'] a un coût = 8
L'éclairage ['D', 'G', 'D', '2', 'G', 'D', 'G', 'D', '2', 'G', 'G'] a un coût = 8
```

```
L'éclairage ['D', 'D', '2', 'G', '2', 'G', 'G', 'D', 'D', 'D', '2'] a un coût = 7
L'éclairage ['2', '2', 'D', 'D', 'D', '2', 'G', 'D', 'D', 'D', 'D'] a un coût = 9
L'éclairage ['D', 'G', '2', 'G', '2', '2', 'D', '2', 'G', 'D', '2'] a un coût = 8
L'éclairage ['2', 'D', 'D', '2', 'D', '2', 'G', 'G', 'D', 'G', '2'] a un coût = 9
L'éclairage ['D', 'D', 'D', '2', 'G', '2', 'D', '2', 'G', '2', 'G'] a un coût = 7
L'éclairage ['D', 'D', '2', '2', '2', 'D', '2', '2', 'D', '2', 'D'] a un coût = 8
```

2.4.6 Génération aléatoire d'une population

```
In [110]: def population_initiale(graphe, taille_population):
              return [ un_individu(graphe) for _ in range(taille_population) ]
```

Par exemple, une population initiale de taille 5 est :

```
In [129]: pop = population_initiale(graphe1, 5)
          for individu in pop:
              print("L'éclairage", individu, "a un coût =", nb_places_eclairees(graphe1, indivi
```

```
L'éclairage ['D', 'D', 'G', 'G', 'G', 'G', '2', 'G', '2', 'D', 'D'] a un coût = 7
L'éclairage ['D', 'D', '2', 'D', 'G', 'G', 'D', '2', 'G', '2', '2'] a un coût = 7
L'éclairage ['2', 'D', '2', '2', 'G', 'D', 'D', 'G', 'G', '2', 'G'] a un coût = 8
L'éclairage ['D', '2', 'G', 'D', 'G', 'G', 'D', 'D', '2', 'G', 'G'] a un coût = 7
L'éclairage ['D', 'G', 'D', '2', 'G', 'G', 'G', '2', 'G', 'G', 'G'] a un coût = 7
```

```
In [130]: sorted(pop, key=lambda individu: nb_places_eclairees(graphe1, individu))
```

```
Out[130]: [['D', 'D', 'G', 'G', 'G', 'G', '2', 'G', '2', 'D', 'D'],
           ['D', 'D', '2', 'D', 'G', 'G', 'D', '2', 'G', '2', '2'],
           ['D', '2', 'G', 'D', 'G', 'G', 'D', 'D', '2', 'G', 'G'],
           ['D', 'G', 'D', '2', 'G', 'G', 'G', '2', 'G', 'G', 'G'],
           ['2', 'D', '2', '2', 'G', 'D', 'D', 'G', 'G', '2', 'G']]
```

On peut donc facilement trier des

2.4.7 Squelette générique pour l'algorithme génétique

On va écrire une fonction générique. Pour visualiser l'évolution de la population, plutôt que d'afficher une liste de 100 coûts, je préfère afficher un décompte du nombre d'individus ayant un certain coût, en Python cela se fait très facilement avec `collections.Counter` :

```
In [192]: import collections
          collections.Counter([1,1,1,1,1,2,2,2,3])
```

```
Out[192]: Counter({1: 5, 2: 3, 3: 1})
```

```
In [193]: def algorithme_genetique(
              pop_init,
              fct_cout,
```

```
    muter_un,
    croiser_deux,
    taille_pop=100,
    tau_meilleurs=0.48,
    tau_cross=0.48,
    nb_generations=1000,
):
    """ Complexité en O(nb_generations * [
        taille_pop * log(taille_pop) * C_cout
        + taille_pop * C_croisement
        + taille_pop * C_mutation
        ) où :

        - C_cout est le coût de calcul de la fonction d'évaluation fct_cout,
        - C_croisement est le coût de calcul de la fonction de croisement croiser_de
        - C_mutation est le coût de calcul de la fonction de mutation muter_un,
    """
    nb_meilleurs = int(tau_meilleurs * taille_pop)
    nb_enfants = 2 * (int(tau_cross * taille_pop)//2)  # nb paire !
    nb_mutes = taille_pop - nb_meilleurs - nb_enfants
    # première population
    pop = pop_init(taille_pop)
    # nb_generations étapes, toutes identiques
    for generation in range(nb_generations):
        couts = [fct_cout(sol) for sol in pop]
        # bonus: affichage de la liste des couts
        print("La génération numéro", generation, "a les coûts suivants :", collecti
        pop_triees = sorted(pop, key=fct_cout)
        # 1) on prend les 48% meilleurs, laissés tels quels
        meilleurs = pop_triees[:nb_meilleurs]
        # 2) on prend les 48% moins bons, on les croise
        moins_bons = pop_triees[-nb_enfants:]
        enfants = [ ]
        for i in range(len(moins_bons) // 2):
            parent_1 = moins_bons[2*i]
            parent_2 = moins_bons[2*i + 1]
            enfant_1, enfant_2 = croiser_deux(parent_1, parent_2)
            enfants.append(enfant_1)
            enfants.append(enfant_2)
        # 3) on prend les 4% meilleurs, et on les mute un peu
        mutes = [ ]
        for i in range(nb_mutes):
            sain = meilleurs[i]
            un_xmen = muter_un(sain)
            mutes.append(un_xmen)
        # on combine les trois listes en une nouvelle population
        nouvelle_pop = meilleurs + enfants + mutes
        pop = nouvelle_pop
```

```
            # a la fin, on renvoie la meilleure solution
            meilleure_solution = max(pop, key=fct_cout)
            return meilleure_solution
```

2.4.8 Mutations et croisements

On doit encore écrire les deux fonctions clés, `muter_un` et `croiser_deux`.

```
In [180]: def une_mutation(graphe, ternaires):
              position = random.randint(0, len(ternaires) - 1)
              mute = [ t for t in ternaires ]
              mute[position] = un_ternaire_aleatoire_different(mute[position])
              return mute

          def mutation(graphe, ternaires):
              M = len(graphe)
              nb_mutation = random.randint(1, M)
              mute = une_mutation(graphe, ternaires)
              for _ in range(nb_mutation - 1):
                  mute = une_mutation(graphe, mute)
              return mute

In [181]: graphe1
          ternaires1_sat

Out[181]: [(0, 2),
           (2, 1),
           (1, 3),
           (1, 5),
           (1, 6),
           (3, 4),
           (4, 5),
           (5, 6),
           (5, 8),
           (6, 7),
           (7, 8)]

Out[181]: ['2', '2', '2', '2', '2', '2', '2', '2', '2', '2', '2']

In [182]: une_mutation(graphe1, ternaires1_sat)

Out[182]: ['2', '2', '2', '2', '2', '2', 'D', '2', '2', '2', '2']

In [183]: mutation(graphe1, ternaires1_sat)

Out[183]: ['D', '2', 'D', 'D', '2', 'G', 'G', '2', '2', '2', 'D']

In [184]: mutation(graphe1, ternaires1_sat)

Out[184]: ['2', '2', 'G', '2', '2', '2', 'D', '2', '2', 'G', 'G']
```

```
In [185]: mutation(graphe1, ternaires1_sat)

Out[185]: ['2', '2', '2', 'G', '2', 'D', 'D', '2', 'D', '2', '2']

In [186]: def croiser_deux_ternaires(graphe, ternaires_1, ternaires_2):
              M1 = len(ternaires_1) // 2
              M2 = len(ternaires_2) // 2
              enfant_1 = ternaires_1[:M1] + ternaires_2[M2:]
              enfant_2 = ternaires_1[M1:] + ternaires_2[:M2]
              return enfant_1, enfant_2

In [187]: print("Les deux parents suivants :")
          ternaires_1 = mutation(graphe1, ternaires1_sat)
          print(ternaires_1)
          ternaires_2 = mutation(graphe1, ternaires1_sat)
          print(ternaires_2)
          enfant_1, enfant_2 = croiser_deux_ternaires(graphe1, ternaires_1, ternaires_2)
          print("peuvent par exemple donner les deux enfants suivants :")
          print(enfant_1)
          print(enfant_2)

Les deux parents suivants :
['2', 'D', '2', '2', '2', '2', '2', '2', '2', '2', 'G']
['2', '2', '2', 'D', 'G', '2', 'D', 'D', '2', '2', '2']
peuvent par exemple donner les deux enfants suivants :
['2', 'D', '2', '2', '2', '2', 'D', 'D', '2', '2', '2']
['2', '2', '2', '2', '2', 'G', '2', '2', '2', 'D', 'G']
```

2.5 Génération d'éclairage par algorithme génétique

On assemble le tout :

```
In [194]: def eclairage_genetique(graphe,
              taille_pop=100,
              tau_meilleurs=0.48,
              tau_cross=0.48,
              nb_generations=50,
          ):
              """ Complexité en O(nb_generations * [
                  taille_pop * log(taille_pop) * O(M)
                  + taille_pop * O(M)
                  + taille_pop * O(M)
                  )  = O (nb_generations * taille_pop * log(taille_pop) * M) où :

                  - M est le nombre d'arêtes dans le graphe.

                  Donc si nb_generations et taille_pop sont constantes, cette fonction est en
              """
```

```
    # on définit les quatre fonctions, pour ce graphe
    def pop_init(taille_pop):
        return population_initiale(graphe, taille_pop)
    def fct_cout(individu):
        return nb_places_eclairees(graphe, individu)
    def muter_un(individu):
        return mutation(graphe, individu)
    def croiser_deux(parent_1, parent_2):
        return croiser_deux_ternaires(graphe, parent_1, parent_2)
    # on appelle la fonction générique
    return algorithme_genetique(
        pop_init,
        fct_cout,
        muter_un,
        croiser_deux,
        taille_pop=taille_pop,
        tau_meilleurs=tau_meilleurs,
        tau_cross=tau_cross,
        nb_generations=nb_generations,
    )
```

Et on donne un exemple :

```
In [205]: eclairage_genetique(graphe1)

La génération numéro 0 a les coûts suivants : Counter({9: 36, 8: 34, 7: 26, 6: 4})
La génération numéro 1 a les coûts suivants : Counter({8: 34, 7: 32, 9: 29, 6: 5})
La génération numéro 2 a les coûts suivants : Counter({7: 35, 9: 30, 8: 28, 6: 7})
La génération numéro 3 a les coûts suivants : Counter({7: 38, 9: 29, 8: 23, 6: 10})
La génération numéro 4 a les coûts suivants : Counter({7: 45, 9: 28, 8: 17, 6: 10})
La génération numéro 5 a les coûts suivants : Counter({7: 48, 9: 27, 8: 13, 6: 12})
La génération numéro 6 a les coûts suivants : Counter({7: 45, 9: 24, 8: 18, 6: 13})
La génération numéro 7 a les coûts suivants : Counter({7: 41, 9: 23, 8: 22, 6: 14})
La génération numéro 8 a les coûts suivants : Counter({7: 39, 8: 25, 9: 22, 6: 14})
La génération numéro 9 a les coûts suivants : Counter({7: 40, 9: 26, 8: 19, 6: 15})
La génération numéro 10 a les coûts suivants : Counter({7: 39, 9: 23, 8: 21, 6: 17})
La génération numéro 11 a les coûts suivants : Counter({7: 40, 9: 26, 6: 20, 8: 14})
La génération numéro 12 a les coûts suivants : Counter({7: 41, 6: 23, 9: 22, 8: 14})
La génération numéro 13 a les coûts suivants : Counter({7: 33, 6: 26, 8: 20, 9: 20, 5: 1})
La génération numéro 14 a les coûts suivants : Counter({7: 30, 6: 28, 9: 23, 8: 17, 5: 2})
La génération numéro 15 a les coûts suivants : Counter({7: 30, 6: 29, 9: 26, 8: 13, 5: 2})
La génération numéro 16 a les coûts suivants : Counter({6: 31, 8: 25, 7: 24, 9: 18, 5: 2})
La génération numéro 17 a les coûts suivants : Counter({6: 33, 7: 24, 9: 24, 8: 17, 5: 2})
La génération numéro 18 a les coûts suivants : Counter({6: 34, 8: 22, 9: 21, 7: 20, 5: 3})
La génération numéro 19 a les coûts suivants : Counter({6: 35, 9: 22, 7: 21, 8: 19, 5: 3})
La génération numéro 20 a les coûts suivants : Counter({6: 37, 7: 21, 9: 21, 8: 18, 5: 3})
La génération numéro 21 a les coûts suivants : Counter({6: 39, 9: 23, 8: 20, 7: 15, 5: 3})
La génération numéro 22 a les coûts suivants : Counter({6: 41, 9: 24, 8: 19, 7: 13, 5: 3})
```

```
La génération numéro 23 a les coûts suivants : Counter({6: 43, 9: 23, 8: 16, 7: 14, 5: 4})
La génération numéro 24 a les coûts suivants : Counter({6: 43, 8: 19, 9: 19, 7: 15, 5: 4})
La génération numéro 25 a les coûts suivants : Counter({6: 46, 9: 19, 8: 16, 7: 15, 5: 4})
La génération numéro 26 a les coûts suivants : Counter({6: 44, 8: 25, 9: 15, 7: 12, 5: 4})
La génération numéro 27 a les coûts suivants : Counter({6: 47, 9: 22, 8: 15, 7: 12, 5: 4})
La génération numéro 28 a les coûts suivants : Counter({6: 44, 9: 21, 8: 19, 7: 11, 5: 5})
La génération numéro 29 a les coûts suivants : Counter({6: 45, 9: 22, 8: 15, 7: 13, 5: 5})
La génération numéro 30 a les coûts suivants : Counter({6: 46, 9: 23, 8: 14, 7: 12, 5: 5})
La génération numéro 31 a les coûts suivants : Counter({6: 44, 9: 21, 8: 18, 7: 12, 5: 5})
La génération numéro 32 a les coûts suivants : Counter({6: 46, 9: 23, 8: 18, 7: 8, 5: 5})
La génération numéro 33 a les coûts suivants : Counter({6: 44, 8: 24, 9: 20, 7: 7, 5: 5})
La génération numéro 34 a les coûts suivants : Counter({6: 43, 9: 29, 8: 12, 7: 11, 5: 5})
La génération numéro 35 a les coûts suivants : Counter({6: 45, 9: 23, 8: 18, 7: 9, 5: 5})
La génération numéro 36 a les coûts suivants : Counter({6: 47, 9: 26, 8: 15, 7: 7, 5: 5})
La génération numéro 37 a les coûts suivants : Counter({6: 44, 9: 22, 8: 20, 7: 9, 5: 5})
La génération numéro 38 a les coûts suivants : Counter({6: 44, 8: 22, 9: 20, 7: 9, 5: 5})
La génération numéro 39 a les coûts suivants : Counter({6: 45, 9: 21, 8: 21, 7: 8, 5: 5})
La génération numéro 40 a les coûts suivants : Counter({6: 47, 9: 25, 8: 13, 7: 10, 5: 5})
La génération numéro 41 a les coûts suivants : Counter({6: 45, 9: 26, 8: 17, 7: 7, 5: 5})
La génération numéro 42 a les coûts suivants : Counter({6: 45, 9: 25, 8: 14, 7: 9, 5: 7})
La génération numéro 43 a les coûts suivants : Counter({6: 44, 9: 21, 8: 16, 7: 11, 5: 8})
La génération numéro 44 a les coûts suivants : Counter({6: 42, 9: 22, 8: 18, 7: 10, 5: 8})
La génération numéro 45 a les coûts suivants : Counter({6: 44, 9: 23, 8: 16, 7: 9, 5: 8})
La génération numéro 46 a les coûts suivants : Counter({6: 43, 8: 21, 9: 18, 7: 10, 5: 8})
La génération numéro 47 a les coûts suivants : Counter({6: 41, 8: 22, 9: 19, 7: 10, 5: 8})
La génération numéro 48 a les coûts suivants : Counter({6: 43, 9: 25, 8: 15, 7: 9, 5: 8})
La génération numéro 49 a les coûts suivants : Counter({6: 42, 9: 21, 8: 17, 7: 12, 5: 8})
```

```
Out[205]: ['2', '2', '2', 'G', 'G', 'G', 'G', '2', 'D', 'D', 'G']
```

On a trouvé un éclairage valide avec seulement 5 places éclairées, en partant d'une population qui avait des coûts entre 7 et 9.

2.6 Conclusion

Si vous êtes curieux, je vous laisse travailler davantage sur ça.