# Algorithmique avancée : quelques mots sur les algorithmes génétiques (pour le projet)

Marin Bougeret

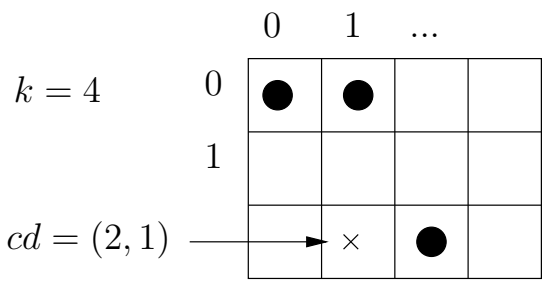
IUT Montpellier

# Problème abordé dans le projet

Problème de la Collecte de Pièces dans une Grille (CPG):

- instance $\mathcal{I} = (p, c_d, k)$:
  - un plateau rect. $p$ où chaque case contient 0 ou 1 pièce
  - une coordonnée de départ dans le plateau $c_d$
  - un nombre de pas $k$ autorisés
- sortie : une solution $s$ est une séquence $s = (c_0, \ldots, c_k)$ de $k+1$ coordonnées telle que
  - $c_0 = c_d$ (on doit partir de $c_d$)
  - $c_{i+1}$ est voisine de $c_i$ (mvt $h$, $b$, $g$ ou $d$ à chaque étape)
- but : maximiser $f(s)$, nombre de pièces ramassées en suivant $s$

- pour $s = ((2,1),(2,2),(2,1),(1,1),(0,1))$
- $f(s) = 2$
- $opt(\mathcal{I}) = 3$



# Problème abordé dans le projet

- CPG est un problème dit d'optimisation (vs "trier un tableau", ou "décider si un graphe est eulérien" (pb de décision))
- un algorithme $A$ est dit exact ssi pour toute instance $\mathcal{I}$, $f(A(\mathcal{I})) = opt(\mathcal{I})$ ($A$ calcule une solution optimale)
- CPG est difficile, au sens où on a peu d'espoir d'avoir un algorithme exact de complexité polynomiale

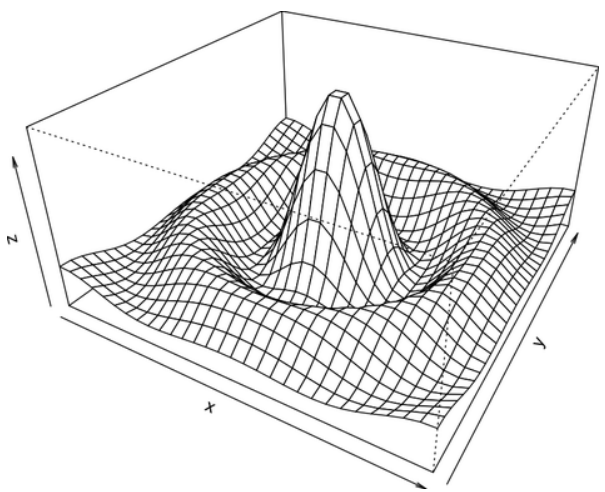
## Approches possibles face à un problème d'optimisation difficile

- résolution exacte, mais en temps exponentiel (brute force, .. )
- résolution approchée (en temps polynomial) :
  - avec garantie de performance (algorithmes d'approximation : prouver par ex que $f(A(\mathcal{I})) \geq \frac{opt(\mathcal{I})}{2}$)
  - sans garantie de performance : heuristiques

# Différents types d'heuristiques

- algorithme ad-hoc (par exemple, algorithme de type glouton qui à chaque étape va vers la pièce restante la plus proche)
- méta-heuristiques (algorithmes qui fonctionnent pour toute une famille de problèmes):
  - abstraire le problème : le voir comme un ensemble de solutions à "explorer"
  - définir maintenant des algorithmes d'exploration

- instance $\mathcal{I} = ..$
- sortie : une sol. $s = (x, y)$
- but : maximiser $f(x, y)$

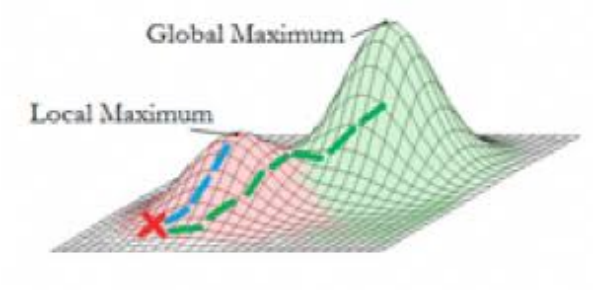


# Différents types d'heuristiques

- algorithme ad-hoc (par exemple, algorithme de type glouton qui à chaque étape va vers la pièce restante la plus proche)
- méta-heuristiques (algorithmes qui fonctionnent pour toute une famille de problèmes):
  - abstraire le problème : le voir comme un ensemble de solutions à "explorer"
  - définir maintenant des algorithmes d'exploration

## Hill Climbing

- étant donné une notion de voisinage (qui dépend du pb)
- partir d'un point, puis tant que possible prendre meilleur voisin

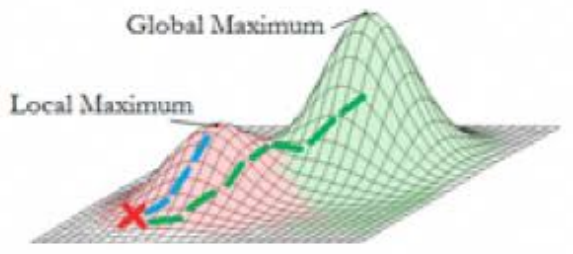


# Différents types d'heuristiques

- algorithme ad-hoc (par exemple, algorithme de type glouton qui à chaque étape va vers la pièce restante la plus proche)
- méta-heuristiques (algorithmes qui fonctionnent pour toute une famille de problèmes):
  - abstraire le problème : le voir comme un ensemble de solutions à "explorer"
  - définir maintenant des algorithmes d'exploration

## Hill Climbing

- ici : en partant de la croix, suit la trajectoire bleue
- pb : peut rester bloqué dans un maximum local (qui n'est pas forcement un maximum global)

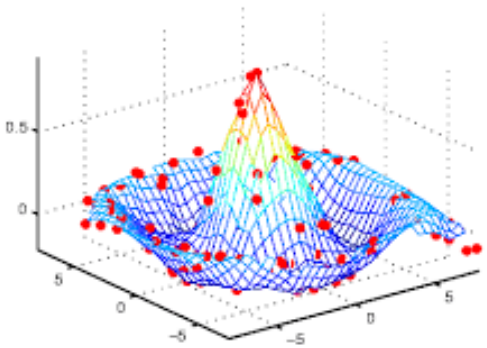


# Différents types d'heuristiques

- algorithme ad-hoc (par exemple, algorithme de type glouton qui à chaque étape va vers la pièce restante la plus proche)
- méta-heuristiques (algorithmes qui fonctionnent pour toute une famille de problèmes):
  - abstraire le problème : le voir comme un ensemble de solutions à "explorer"
  - définir maintenant des algorithmes d'exploration

## Algorithmes génétiques

- étant donnés des mécanismes de croisement, mutation etc. (dépendent du pb)

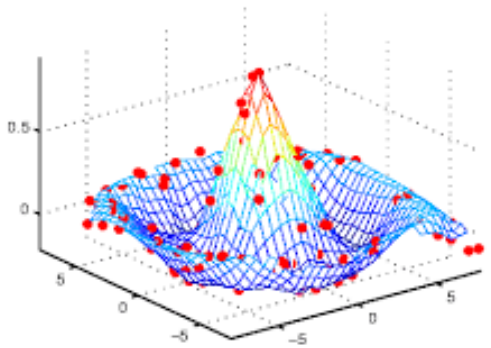


# Différents types d'heuristiques

- algorithme ad-hoc (par exemple, algorithme de type glouton qui à chaque étape va vers la pièce restante la plus proche)
- méta-heuristiques (algorithmes qui fonctionnent pour toute une famille de problèmes):
  - abstraire le problème : le voir comme un ensemble de solutions à "explorer"
  - définir maintenant des algorithmes d'exploration

## Algorithmes génétiques

- partir d'une population initiale, la faire évoluer "nbgen" fois, et garder la meilleure solution de la dernière génération



# Algorithmes génétiques

## Observation

La nature résout des problèmes d'optimisation!

## Exemple

Considérons une espèce animale fixée.

- on définit une fonction de "fitness" d'un individu $fit(i)$ qui mesure à quel point $i$ est adapté à son environnement ($fit(i)$ grand $\Leftrightarrow$ $i$ est bien adapté à son environnement)
- la "nature" cherche donc à maximiser la fitness (en découvrant comment ajuster différents paramètres : longueur jambes, écartement des yeux, ...) génération après génération

# Algorithmes génétiques

## Observation

La nature résout des problèmes d'optimisation!

## Exemple

Considérons une espèce animale fixée.

- Les outils pour calculer la génération suivante sont
  - croisement (production d'un individu qui "hérite" de caractéristique de ses deux parents)
  - mutation
  - sélection (les individus de grande fitness ont plus de chance d'être sélectionnés)

# Algorithmes génétiques

## Idée

Appliquer ce schéma d'optimisation à son problème (CPG, ..)

## Ingrédients nécessaires

On suppose qu'on dispose d'une notion d'individu, qui sait

- calculer sa solution associée (un individu représente une solution)
- se croiser avec un autre individu pour calculer un fils
- se muter
- évaluer sa fitness

# Algorithmes génétiques

## Idée

Appliquer ce schéma d'optimisation à son problème (CPG, ..)

## (Méta)-algorithme génétique

```
solution AlgoGen(I){ //I instance
  nbgen = 100; //nb de generations
  pop = creer population initiale
  repeter (nbgen) fois{
    pop = calculerNext(pop); //à partir de
        selections, croisements et mutations
  }
  best = selectionner individu dans pop de
      meilleure fitness
  sol = calculer solution associee à best
}
```

# Algorithmes génétiques

## Idée

Appliquer ce schéma d'optimisation à son problème (CPG, ..)

## Conclusion

Pour appliquer le méta algorithme génétique à son problème, il faut donc

1. définir la notion d'individu, ainsi que
   - calcul de sa solution associée
   - croisement avec un autre individu pour calculer un fils
   - mutation
   - évaluation de la fitness d'un individu
2. définir un algorithme pour calculerNext (à partir de sélections, croisements et mutations)

# Algorithmes génétiques : application 1

Problème de maximisation d'une fonction

- instance: $\mathcal{I} = (f, a, b)$ ($f$ de $[a, b] \subseteq \mathbb{N} \to \mathbb{R}$)
- sortie : une solution est un entier $s$ dans $[a, b]$
- but : maximiser $f(s)$

## 1) Définition de la notion d'individu

Par exemple, on définit un individu comme un entier $s \in [a, b]$

- calculer sa solution associée : rien à faire
- croiser $s_1$ avec $s_2$ : on définit $s_{fils} = \lfloor \frac{s_1+s_2}{2} \rfloor$
- se muter : on définit $s_{mut} = s + 1$ ou $s_{mut} = s - 1$ (en tirant au hasard) (sauf si $s = a$ ou $b$ alors faire $+1$ ou $-1$ seulement)
- evaluer sa fitness : on définit $fit(s) = f(s)$

# Algorithmes génétiques : application 1

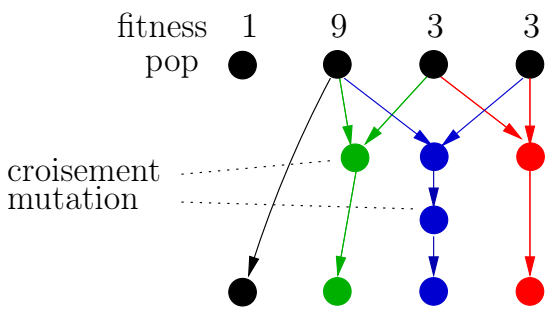
## 2) Définition de calculerNext(pop)

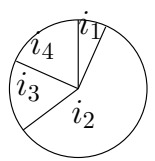
- but : étant donné la génération courante "pop", calculer une génération suivante
- outils :
  - sélection (selon la fitness)
  - croisements
  - mutations
- plusieurs schémas classiques existent!
- détaillons un schéma possible (utilisé dans le projet)

# Algorithmes génétiques : application 1

## 2) Définition de calculerNext(pop) : v1 de calculerNext du projet

- pour tout $i \in pop$, soit $p_i = \frac{fit(i)}{S}$ (avec $S = \sum_i fit(i)$, et avec $fit(i) > 0$)
- on définit une méthode "individu selection(pop)" qui sélectionne au hasard un individu de pop, de telle sorte que proba(i selectionné) $= p_i$ (méthode dit de la roulette)



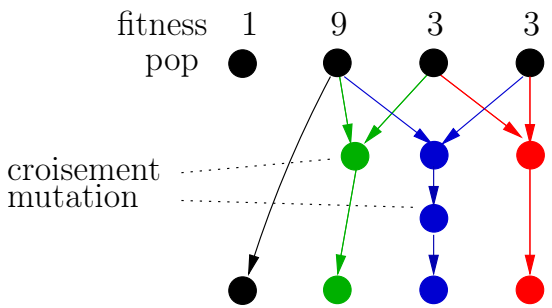



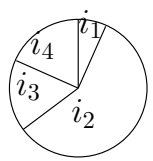

roulette de selection

## 2) Définition de calculerNext(pop) : v1 de calculerNext du projet

Puis:

- commencer par ajouter un individu de fitness maximum
- répéter $|pop| - 1$ fois:
  - sélection de deux parents à la roulette
  - calcul du croisement
  - calcul de la mutation (avec faible proba)




roulette de selection

# Algorithmes génétiques : application 1

**Idée**

Appliquer ce schéma d'optimisation à son problème (CPG, ..)

**Conclusion**

Pour appliquer le méta algorithme génétique à son problème, il faut donc

1. FAIT! définir la notion d'individu, ainsi que
   - calcul de sa solution associée
   - croisement avec un autre individu pour calculer un fils
   - mutation
   - évaluation de la fitness d'un individu
2. FAIT! définir un algorithme pour calculerNext (à partir de sélections, croisements et mutations)

# Algorithmes génétiques : application 1

**Idée**

Appliquer ce schéma d'optimisation à son problème (CPG, ..)

**Conclusion**

Pour appliquer le méta algorithme génétique à son problème, il faut donc

1. FAIT! définir la notion d'individu, ainsi que
   - calcul de sa solution associée
   - croisement avec un autre individu pour calculer un fils
   - mutation
   - évaluation de la fitness d'un individu
2. FAIT! définir un algorithme pour calculerNext (à partir de sélections, croisements et mutations)

Démo via l'application de Fabien Moutarde (attention : son implémentation des points 1 et 2 peut être différente, mais peu importe!)

# Algorithmes génétiques : application 2

Problème CPG

- instance $\mathcal{I} = (p, c_d, k)$
- sortie : une solution $s$ est une séquence $s = (c_0, \ldots, c_k)$ de $k+1$ coordonnées tq ..
- but : maximiser $f(s)$, nombre de pièces ramassées en suivant $s$

## 1) Définition de la notion d'individu

Par exemple, on définit un individu comme une séquence de $k$ pas (qui reste dans le plateau), où un pas $\in \{h, b, g, d\}$.

- calcul de la solution associée $s(i)$ : pour l'ex du slide 2, avec $i = (d, g, h, h)$, on obtient $s(i) = s$
- croisement de $i_1$ avec $i_2$ : mélange des chaînes + normalisation (cf td)
- mutation : échange de deux caractères + normalisation (cf td)
- évaluation de la fitness : on définit $fit(i_1) = 1 + 10 * f(s(i))$

# Algorithmes génétiques : application 2

Problème CPG

- instance $\mathcal{I} = (p, c_d, k)$
- sortie : une solution $s$ est une séquence $s = (c_0, \ldots, c_k)$ de $k+1$ coordonnées tq ..
- but : maximiser $f(s)$, nombre de pièces ramassées en suivant $s$

## 2) Définition de calculerNext(pop) : v1 de calculerNext du projet

Commet avant!

# Conclusion

## Bilan du projet

Résumé du travail demandé :

- coder plusieurs types d'individus
- pour chaque type d'individu, coder plusieurs mut/croisements
- coder plusieurs algorithmes de "calculNext"

But final : trouver la combinaison donnant les meilleurs résultats possibles! (essayer de battre le glouton..)

# Conclusion

## Remarques sur les algorithmes génétiques

Difficulté pour avoir un bon algorithme génétique : ajuster les nombreux degrés de libertés :

- pour la population initiale : partir d'individus "random" ? (mais plusieurs façons de les génerer..), d'individus calculés avec un autre algorithme ?
- pour calculNext :
  - pour la selection, d'autres méthodes que la roulette existent (et ne pas être trop élitiste car de mauvais individus peuvent s'avérer intéressants plus tard..)
  - on peut séparer croisements/mutations, ajuster la probabilité de mutation,..

# Conclusion

## Remarques sur les algorithmes génétiques

Difficulté pour avoir un bon algorithme génétique : ajuster les nombreux degrés de libertés :

- pour les individus :
  - de nombreux codages sont possibles
  - même pour un codage, on peut envisager plusieurs notions de mutation/croisement
- pour les paramètres de l'algo ($|pop|$ et $nbgen$), on a envie que les deux soient grands, mais pas possible (trop long)..