Chapitre 2: Modélisation

INF889B — Algorithmes d'optimisation combinatoire

Alexandre Blondin Massé

Université du Québec à Montréal

Hiver 2020

Plan

1. Vocabulaire de base
2. Algorithmes
3. Expériences
4. Exemple complet

Vocabulaire de base

Plusieurs types de problèmes

- On considère un ensemble d'instances **possibles** $\mathcal{I}$ et
- $\mathcal{J} \subseteq \mathcal{I}$ un sous-ensemble d'instances **réalisables** ou **admissibles**

Décision

- Soit $I \in \mathcal{I}$
- Décider (en répondant *oui* ou *non*) si $I \in \mathcal{J}$

Dénombrement

- Calculer la valeur $|\mathcal{J}|$

Énumération

- Énumérer exactement une fois chaque élément de $\mathcal{J}$

Optimisation

- Soit $f : \mathcal{I} \to \mathbb{R}$ une fonction objectif
- Trouver $I \in \mathcal{I}$ telle que $f(I) \preceq f(I')$ pour tout $I' \in \mathcal{I}$

Question sur les types de problèmes

- Soit G un graphe simple fini et connexe. Est-ce que G admet un cycle hamiltonien?

Question sur les types de problèmes

- Soit G un graphe simple fini et connexe. Est-ce que G admet un cycle hamiltonien?

$\rightarrow$ **décision**

- Une compagnie doit livrer plusieurs colis à des adresses différentes. Elle souhaite minimiser la distance totale parcourue.

Question sur les types de problèmes

- Soit G un graphe simple fini et connexe. Est-ce que G admet un cycle hamiltonien?

$\rightarrow$ **décision**

- Une compagnie doit livrer plusieurs colis à des adresses différentes. Elle souhaite minimiser la distance totale parcourue.

$\rightarrow$ **optimisation**

- Un département souhaite attribuer les cours aux enseignants selon leurs préférences et leurs contraintes, en proposant 10 scénarios différents.

Question sur les types de problèmes

- Soit G un graphe simple fini et connexe. Est-ce que G admet un cycle hamiltonien?

$\rightarrow$ **décision**

- Une compagnie doit livrer plusieurs colis à des adresses différentes. Elle souhaite minimiser la distance totale parcourue.

$\rightarrow$ **optimisation**

- Un département souhaite attribuer les cours aux enseignants selon leurs préférences et leurs contraintes, en proposant 10 scénarios différents.

$\rightarrow$ **optimisation** et **énumération**

- Étant donné une relation d'ordre partiel R sur un ensemble fini E, combien existe-t-il d'extensions linéaires de R sur E?

Question sur les types de problèmes

- Soit G un graphe simple fini et connexe. Est-ce que G admet un cycle hamiltonien?

$\rightarrow$ **décision**

- Une compagnie doit livrer plusieurs colis à des adresses différentes. Elle souhaite minimiser la distance totale parcourue.

$\rightarrow$ **optimisation**

- Un département souhaite attribuer les cours aux enseignants selon leurs préférences et leurs contraintes, en proposant 10 scénarios différents.

$\rightarrow$ **optimisation** et **énumération**

- Étant donné une relation d'ordre partiel R sur un ensemble fini E, combien existe-t-il d'extensions linéaires de R sur E?

$\rightarrow$ **dénombrement**

Vocabulaire de base en optimisation

- **Espace**: ensemble de toutes les instances possibles, réalisables et non réalisables
- **Ensemble des solutions admissibles**: ensemble de toutes les instances respectant un certain nombre de contraintes
- **Solution admissible**: un élément de l'espace des solutions admissibles
- **Fonction objectif** (ou **objective**): la fonction à optimiser, c'est-à-dire à minimiser ou à maximiser
- **Solution optimale**: solution admissible qui maximise ou minimise la fonction objectif
- **Valeur optimale**: valeur atteinte par la fonction objectif lorsqu'on l'évalue en une solution optimale
- **Borne inférieure** ou **supérieure**: valeur dont on sait qu'elle est respectivement plus petite ou plus grande que la valeur optimale

Différents scénarios possibles

Nombre infini de solutions

- L'espace des solutions admissibles doit être **infini**
- Fréquent lorsque l'espace est **continu**

Nombre fini non nul de solutions

- Si l'espace ou l'ensemble de solutions admissibles est **fini**
- Évidemment, c'est aussi possible si l'espace est **infini**, qu'il soit **discret** ou **continu**

Aucune solution optimale

- Parce qu'il n'existe aucune solution **réalisable**
- Parce que l'ensemble des valeurs est **non borné**
- Ou il est borné, mais **ouvert** (au sens topologique)

Questions

Dans chacun des exemples suivants, indiquer si l'espace est **discret** ou **continu** et si le nombre de solutions est **nul**, **fini** ou **infini**

1. Soit G un graphe non orienté de n sommets et m arêtes. On souhaite (a) maximiser m sous la contrainte $n < 8$ (b) maximiser n sous la contrainte $m \leq 8$ et (c) maximiser m sous la contrainte $n \geq 8$
2. Soit G un graphe complet. On cherche une sous-arbre induit de G de taille maximale.
3. Soit (x, y) un point du plan cartésien dans la région comprise entre les droites $t \mapsto (2, 1) + t(1, -1)$ et $t \mapsto (-3, -1) + t(2, 1)$. On cherche à (a) minimiser la somme $x + y$ (b) à maximiser la différence $y - x$.
4. Même question, mais (x, y) se trouve dans le disque centré à l'origine de rayon 5.

Algorithmes

Algorithmes

Définition

- **Suite finie** de calculs
- **Retournant** généralement un résultat

Types d'algorithme

- **Exact**: retourne toujours une solution
- **Probabiliste**: retourne une solution avec certaine probabilité
- **Déterministe**: retourne toujours le même résultat, peu importe l'environnement
- **Non déterministe**: peut retourne des résultats différents selon l'environnement
- **Pseudoalgorithme**: algorithme qui ne termine pas toujours

Complexités théoriques

Deux choses qu'on mesure

- **Complexité temporelle**: temps pris par l'algorithme pour se terminer ($\approx$ temps CPU total en théorie)
- **Complexité spatiale**: espace additionnel utilisé par l'algorithme ($\approx$ mémoire nécessaire en théorie)

Différentes analyses possibles

- **Pire cas**: temps pris lorsque l'entrée est « défavorable »
- **Meilleur cas**: temps pris lorsque l'entrée est « favorable »
- **Cas moyen**: temps pris en moyenne (probabiliste)
- **Cas typique**: temps pris par un cas typique (empirique)
- **Analyse amortie**: temps moyen pris par opération (même dans le pire cas).

$\rightarrow$ Les détails dans **INF7341 Structures de données**.

Notation asymptotique

- Soient F l'**ensemble** des fonctions de $\mathbb{N} \to \mathbb{R}_+$.
- Soient $f, g \in F$. On écrit $f \preceq g$ s'il existe $k, C > 0$ tels que

$$f(n) \leq Cg(n),$$

  pour tout $n \geq k$. On dit alors que g **domine** f.
- On définit les cinq **ensembles** suivants:

$$\begin{aligned}
\mathcal{O}(f) &= \{g \in F \mid f \preceq g\} \\
\Omega(f) &= \{g \in F \mid f \in \mathcal{O}(g)\} \\
\Theta(f) &= \{g \in F \mid g \in \mathcal{O}(f) \text{ et } f \in \mathcal{O}(g)\} \\
o(f) &= \left\{g \in F \;\middle|\; \lim_{n \to +\infty} \frac{g(n)}{f(n)} = 0\right\} \\
\omega(f) &= \{g \in F \mid f \in o(g)\}
\end{aligned}$$

Complexités fréquentes

Soit n la taille du paramètre qui nous intéresse.

- **Constante**: $\Theta(1)$
- **Logarithmique**: $\Theta(\log(n))$
- **Linéaire**: $\Theta(n)$
- **n log n**: $\Theta(n \log n)$
- **Quadratique**: $\Theta(n^2)$
- **Cubique**: $\Theta(n^3)$
- **Polynomiale**: $\mathcal{O}(n^k)$ pour un certain entier $k \in \mathbb{N}$
- **Exponentielle**: $\Omega(b_1^n) \cap \mathcal{O}(b_2^n)$ pour certains réels $b_1, b_2 > 1$
- **Quasi-polynomiale**: $\omega(n^k)$ pour tout entier $k \in \mathbb{N}$ et $o(b^n)$ pour tout $b > 1$

Notes: La fonction $\log$ est en base 2, sauf indication contraire.

Difficulté d'un problème

Problème polynomial

Problème qu'on peut résoudre en temps polynomial

Problème NP

Problème de **décision** dont on peut vérifier en temps polynomial (algorithme de vérification) si une solution donnée (certificat) est bien valide

- NP $\neq$ *non-polynomial*
- NP $=$ *non-deterministic polynomial*
- **Problème NP-complet**: problème de **décision** les plus difficiles de la classe NP
- **Problème NP-difficile**: problème au moins aussi difficile que les plus difficiles problèmes NP-complets.

Note: Plus de détails dans le cours INF7440 Conception et analyse des algorithmes

Tractabilité

Problème tractable

- Considéré résoluble en temps et en espace « raisonnables »
- Tractable $\approx$ polynomial

Problème intractable

- Considéré trop long à résoudre ou en utilisant une quantité acceptable de mémoire
- Intractable $\approx$ superpolynomial
- Inclut les problèmes **exponentiels**
- Mais il existe des classes entre **polynomial** et **exponentiel**
- Par exemple, un algorithme peut être **quasi-polynomial**

$\rightarrow$ voir développements récents sur le problème d'isomorphisme de graphes, où la solution est dans $\mathcal{O}(2^{c \log n})$ pour une certaine constante $c > 1$.

Description d'un algorithme

En texte continu

- **Avantages**: synthétique, plus simple à écrire, plus court
- **Inconvénients**: souvent vague ou ambigu

Étapes numérotées

- **Avantages**: assez synthétique, plus détaillé que du texte, plus facile de décrire des structures conditionnelles ou les boucles qu'en texte
- **Inconvénients**: un peu plus long, parfois difficile de décrire les structures de contrôle correctement

Pseudocode

- **Avantages**: plus clair (quand bien fait), prépare mieux la personne qui va programmer
- **Inconvénients**: plus long, plus difficile d'évaluer le niveau de détails nécessaire

Exemple de pseudocode

1: **fonction** $\textsc{ComposantesConnexes}(G$: graphe) : ensembles disjoints
2: Soit D une structure de données d'ensembles disjoints
3: **pour** $v \in G.\textsc{Sommets}()$ **faire**
4: $D.\textsc{AjouterSingleton}(v)$
5: **pour** $\{u, v\} \in G.\textsc{Arêtes}()$ **faire**
6: $D.\textsc{Fusionner}(u, v)$
7: **retourner** D

8: **fonction** $\textsc{SontDansMêmeComposante}(D$: ensembles disjoints, u, v : sommets)
9: **retourner** $D.\textsc{Représentant}(u) = D.\textsc{Représentant}(v)$

- Organisation du code à l'aide de **blocs**
- Les structures de **contrôle** sont plus claires
- On peut **appeler** d'autres fonctions
- On peut tout de même mettre des **phrases**

Structures de données

- Selon le contexte, vous pouvez **supposer** que vous disposez de certaines structures de données
- **Exemple**: nombres, tableaux, listes, ensembles, tableaux associatifs, matrices, piles, files d'attente, arbres, graphes, etc.
- Même des structures connues **plus complexes**: ensembles disjoints, arbres équilibrés, arbres d'intervalles, files à priorité, etc.
- Par contre, il est important de décrire les **opérations** disponibles, ainsi que le **coût** de chaque opération

Exemple

- Reprenons la fonction $\textsc{ComposantesConnexes}$ précédente
- Soit n et m le nombre de sommets et d'arêtes

Alors on suppose que

- $\textsc{AjouterSingleton}(v)$ s'effectue en $\mathcal{O}(1)$
- $D.\textsc{Fusionner}(u, v)$ s'effectue en $\mathcal{O}(1)$
- $D.\textsc{Représentant}(v)$ s'effectue en temps amorti $\mathcal{O}(\alpha(n))$, où α est une fonction d'Ackermann inversée, qui croît très, très lentement

Pourquoi?

- Ça a été démontré mathématiquement par Tarjan en 1984!
- En pratique, $\alpha(n) \leq 5$ pour toute valeur de n pouvant être écrite **physiquement**
- Donc c'est essentiellement comme une fonction **constante**

Style du pseudocode (1/2)

- **Indenter** le code correctement
- Utiliser la **coloration syntaxique**
- Écrire dans la **langue** du document courant
- Préférer la **notation mathématique**: $x \leftarrow 3y^2 - 4$ plutôt que `x = 3 * y ** 2 - 4`
- Déclarer le type de la variable sous la forme x : *type* plutôt que `type x` (comme en C, C++ et Java)
- Les **affectations** peuvent être écrites avec $\leftarrow$ ou $=$

$\rightarrow$ Mais soyez uniformes!

- Préférer les noms de variables **courts**, idéalement une seule lettre

$\rightarrow$ Dans le code, vous pourrez les allonger

- Utiliser une police **uniforme**

$\rightarrow$ Ne pas laisser tel quel en mode mathématique: $Val \leftarrow Val + 1$

$\rightarrow$ Il suffit de protéger avec `\emph` ou `\mathrm`: $\mathit{Val} \leftarrow \mathit{Val} + 1$

Style du pseudocode (2/2)

- Ne pas mettre **trop de détails**
- Utiliser au besoin une **phrase** plutôt qu'une **expression formelle**
- **Factoriser** le code au maximum pour éviter la redondance
- Se limiter à **30 lignes**, sauf cas exceptionnel
- **Diviser** une fonction longue en plus petites fonctions
- Préférer la notation **mathématique** à la notation informatique
- Mais éviter les symboles $\exists$ et $\forall$ (préférer du texte)
- Ne pas utiliser de conventions liées à des **langages spécifiques**, comme les valeurs `null`, `void`, etc.

Exemple

```
1:  fonction Dijkstra(G = (V, E): graphe, w : poids, s, c : sommets) : chemin
2:      Soit Dist un tableau de longueur |V| initialisé à ∞ partout
3:      Soit Parent un tableau de longueur |V| initialisé à s partout
4:      Dist[s] ← 0
5:      M ← ConstruireMonceau(V) avec priorité donnée par Dist
6:      u ← s
7:      tant que M est non vide et u ≠ c faire
8:          u ← M.ExtraireMinimum()
9:          pour v ∈ G.Successeurs(u) faire
10:             si M.Contient(v) alors
11:                 δ ← Dist[u] + w(u, v)
12:                 si δ < Dist[v] alors
13:                     (Dist[v], Parent[v]) ← (δ, u)
14:                     M.ModifierPriorite(v, δ)
15:     si u ≠ c alors
16:         Afficher que le sommet c n'est pas atteignable
17:     sinon
18:         Soit C = [c] un chemin
19:         tant que u ≠ s faire
20:             Ajouter u en tête de la liste C
21:             u ← Parent[u]
22:     retourner C
```

Comment **améliorer** ce pseudocode?

Expériences

Vérification empirique

Expériences

- Dans votre projet de session, vous allez concevoir des **expériences** pour mesurer la performance de façon **empirique**
- Devrait être **diffusée** avec les résultats
- Permet de garantir la **reproductibilité**
- C'est très **long**...

Jeu de données (*benchmark*)

Ensemble d'**instances** du problème de taille de plus en plus grande

- parfois **déjà disponibles**
- parfois il faut les **générer** soi-même, en tenant compte de leur distribution

Mesure et statistiques

Temps de calcul

- Mesure du **temps** d'exécution sur une machine particulière
- Peut **varier** énormément d'une machine à une autre
- Faire des comparaisons **honnêtes**!

Autres statistiques

- Parfois, il est pertinent de mesurer la **mémoire** utilisée
- Ou d'autres **statistiques indépendantes** de la machine:

$\rightarrow$ nombre de solutions visitées

$\rightarrow$ optimums locaux trouvés

$\rightarrow$ vitesse de convergence, etc.

- Le logiciel Gnuplot est particulièrement utile pour automatiser la production de graphiques à partir des statistiques calculées

Exemple complet

Couverture des arêtes par les sommets

Problème

- Soit $G = (V, E)$ un graphe simple (non orienté) de n sommets et m arêtes
- Soit $U \subseteq V$
- On dit que U est une **couverture (des sommets par les arêtes)** (en anglais, *vertex cover*) si pour toute arête $\{u, v\} \in E$, on a $u \in V$ ou $v \in V$
- On souhaite compter le nombre de couvertures de taille **minimale** de G.

Question

- Proposez une **solution naïve** qui prend un temps $\mathcal{O}(2^n)$
- Quelle sa complexité **spatiale**?

Implémentation de la solution naïve

- Permet de bien **modéliser** le problème
- Et d'avoir une **référence de base** pour d'éventuelles comparaisons

Considérations techniques

- On choisit un **langage de programmation**
- Préférablement **efficace** et d'assez **bas niveau**
- **Exemples**: C, C++, Rust, Java
- Éviter les langages **moins performants**: Python, Ruby, Javascript, ...
- Ensuite, on propose des **structures** de données épurées
- Éviter les **bibliothèques** génériques
- Puis on **implémente** la solution

Implémentation (1/2)

- Code source complet disponible sur GitLab

Structures de données

```
struct Graph {                     // A simple graph
    unsigned int num_vertices; // Its number of vertices
    unsigned int num_edges;    // Its number of edges
    bool **edges;              // edges[u][v] iff {u,v} is an edge
};

struct Solution {            // A solution is a set of vertices
    unsigned int capacity; // The number of vertices
    bool *included;        // included[u] iff u is in the solution
};

struct Search {                             // A naive search
    const struct Graph *graph;            // The graph to search
    struct Solution *solution;            // The current solution in the search
    unsigned int optimal_size;            // The size of an optimal solution
    unsigned int num_optimal_solutions; // The number of optimal solutions
    bool verbose;                         // If true, displays trace details
};
```

Implémentation (2/2)

- En-tête de fonctions

```
// Operations on graphs
void load_graph(struct Graph *graph);
void print_graph(const struct Graph *graph);

// Operations on solutions
unsigned int solution_size(const struct Solution *solution);
void print_solution(const struct Solution *solution);
bool is_vertex_cover(const struct Graph *graph,
                     const struct Solution *solution);

// Searching
void compute_num_optimal_solutions(struct Search *search,
                                   unsigned int depth);
unsigned int num_optimal_solutions(const struct Graph *graph,
                                   bool verbose);
```

Remarques

- Chargement du graphe sur `stdin`
- Affichage sur `stdout`
- Utilisation d'un argument `verbose` pour affichage de la trace

Représentation d'un graphe

- Format **simple** pour minimiser le travail de chargement
- Soit `n` et `m` sont le nombre de sommets/d'arêtes
- Et `{ui,vi}` les arêtes pour `i = 1,2,...,n`, avec `0 <= ui,vi < n`
- Alors on utilise le **format** suivant:

```
n m
u1 v1
u2 v2
[...]
un vn
```

Exemple:

```
$ cat examples/example-1.graph
6 6
0 1
0 2
[...]
1 5
```

Exemple d'exécution

```
$ bin/vertex-cover < examples/example-1.graph
2

$ bin/vertex-cover verbose < examples/example-1.graph
Loading the graph...
Graph of 6 vertices and 6 edges
  0 -- 1
  0 -- 2
  1 -- 2
  1 -- 3
  1 -- 4
  1 -- 5
Found a better solution: [ 0 1 2 3 4 5 ]
Found a better solution: [ 0 1 2 3 4 ]
Found another currently optimal solution: [ 0 1 2 3 5 ]
Found a better solution: [ 0 1 2 3 ]
Found another currently optimal solution: [ 0 1 2 4 ]
Found another currently optimal solution: [ 0 1 2 5 ]
Found a better solution: [ 0 1 2 ]
Found another currently optimal solution: [ 0 1 3 ]
Found another currently optimal solution: [ 0 1 4 ]
Found another currently optimal solution: [ 0 1 5 ]
Found a better solution: [ 0 1 ]
Found another currently optimal solution: [ 1 2 ]
Naive algorithm...
Number of optimal solutions: 2
```

Évaluation de la performance

- Quelle est l'**efficacité** en pratique de l'algorithme naïf?
- Il suffit de vérifier sur des **petits exemplaires**
- Jusqu'à ce que le temps devienne **trop grand**
- Par exemple, quand ça prend plus de **30 secondes**

Jeu de données

- On peut vérifier sur des **jeux de données** existants
- Ou simplement **générer** des graphes aléatoires
- Plusieurs **modèles**: à *densité* fixée, type *petit monde*, ...

Question préalable

À partir de quelle taille **estimez**-vous que l'algorithme sera trop long?

Génération d'un graphe aléatoire

Script Python qui **génère** un graphe aléatoire:

```
# Retrieve arguments
num_vertices = int(sys.argv[1])
density = float(sys.argv[2])

# Retrieve number of edges from density
num_edges = int(density * num_vertices * (num_vertices - 1) / 2)

# Generate the edges
edges = set()
while len(edges) < num_edges:
    u = random.randint(0, num_vertices - 2)
    v = random.randint(u + 1, num_vertices - 1)
    edges.add((u, v))

# Print the graph in compact format
print('%d %d' % (num_vertices, num_edges))
for (u, v) in sorted(edges):
    print('%d %d' % (u, v))
```

Génération d'un jeu de données

Script shell qui **génère** plusieurs graphes aléatoires:

```
#!/bin/bash

bin_dir=bin
data_dir=data
num_instances=5
densities=(0.1 0.2 0.5 0.8 0.9)
num_vertices=(19 20 21 22 23 24 25 26 27 28 29 30)

mkdir -p $data_dir

for i in $(seq $num_instances); do
  for density in ${densities[@]}; do
    for num_vertex in ${num_vertices[@]}; do
      filename="$(printf 'example-%.2d-%s-%.2d.graph'\
                          "$num_vertex" "$density" "$i")"
      echo "Generating graph (n = $num_vertex, d = $density)\
            in $filename"
      "$bin_dir/generate-graph" "$num_vertex" "$density"\
            > "$data_dir/$filename"
    done
  done
done
```

Chronométrage du temps de calcul

- On peut simplement utiliser la commande `time`
- En combinaison avec `timeout` pour limiter le temps de calcul

```
#!/bin/bash

echo 'num_vertex,density,time'
for graph_filename in data/*; do
    n="$(cut -d'-' -f 2 <<< "$graph_filename")"
    d="$(cut -d'-' -f 3 <<< "$graph_filename")"
    t="$(/usr/bin/time 2>&1 -f "%U"\
        timeout 1m bin/vertex-cover quiet < "$graph_filename")"
    printf "%d,%s,%s\n" $n "$d" "$t"
done
```

Génération du graphique des temps de calcul

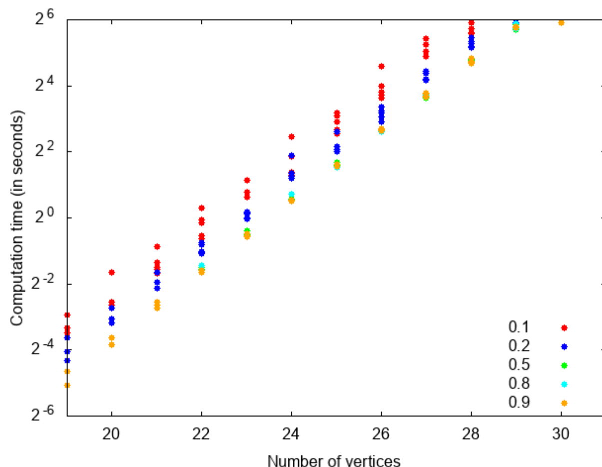
- Finalement, on **visualise** les temps de calcul
- Le logiciel Gnuplot permet d'**automatiser** cela

```
#!/usr/bin/gnuplot

# General settings
set terminal png
set datafile separator ","
set logscale y 2
set format y "2^{%L}"
set xlabel "Number of vertices"
set ylabel "Computation time (in seconds)"
set xrange [19:31]
set key bottom

# Plot times
plot "examples/times.csv" using ($2 == 0.1 ? $1 : 1/0):3\
                          ps 1.0 pt 7 lc "red" title "0.1" with points,\
     "examples/times.csv" using ($2 == 0.2 ? $1 : 1/0):3\
                          ps 1.0 pt 7 lc "blue" title "0.2" with points,\
     "examples/times.csv" using ($2 == 0.5 ? $1 : 1/0):3\
                          ps 1.0 pt 7 lc "green" title "0.5" with points,\
     "examples/times.csv" using ($2 == 0.8 ? $1 : 1/0):3\
                          ps 1.0 pt 7 lc "cyan" title "0.8" with points,\
     "examples/times.csv" using ($2 == 0.9 ? $1 : 1/0):3\
                          ps 1.0 pt 7 lc "orange" title "0.9" with points
```

Résultat



- Que peut-on **observer**?