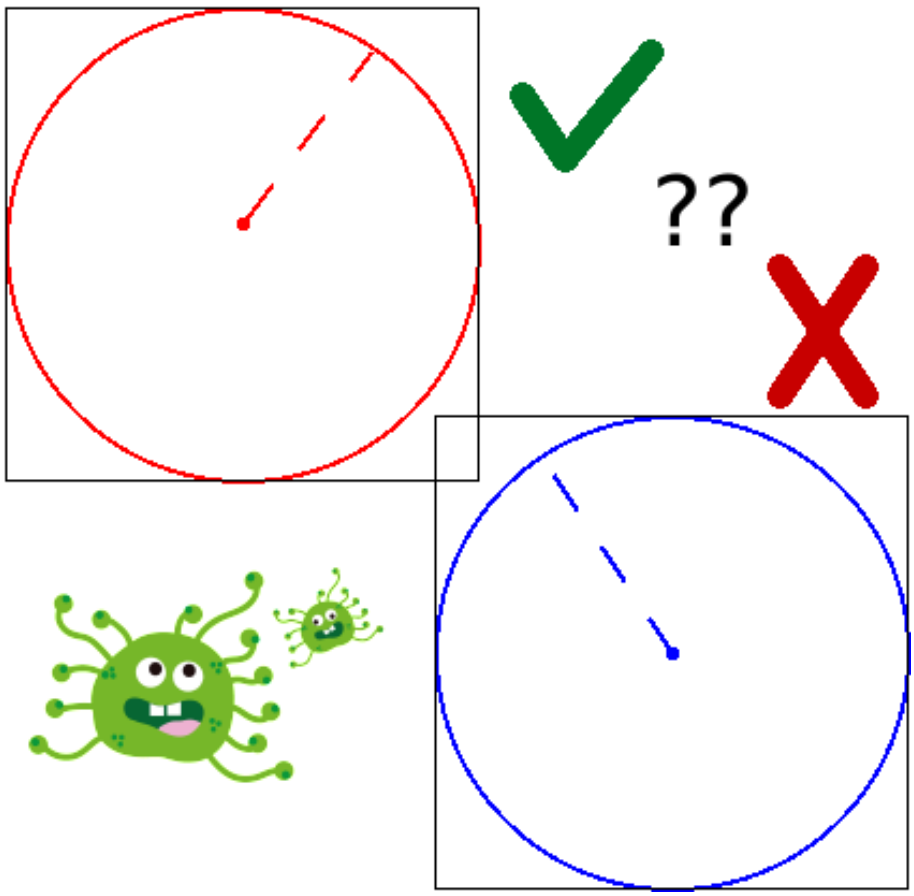
SAÉ 1.02 - Comparaison d'approches algorithmiques

Optimisation d'algorithmes pour une simulation de diffusion ~~de la~~ du covid



Descriptif détaillé de la SAÉ

En quoi consiste cette SAÉ ?

En partant d'un besoin exprimé par un client, il faut réaliser une implémentation, comparer plusieurs approches pour la résolution d'un problème et effectuer des mesures de performance simples. Cette SAÉ permet une première réflexion autour des stratégies algorithmiques pour résoudre un même problème.

Quelles sont les productions de cette SAÉ ?

— Code de l'application.
— Présentation du problème et de la comparaison des différentes approches.

Quelles sont les compétences développées ?

Appréhender et construire des algorithmes :
— AC 1 Analyser un problème avec méthode (découpage en éléments algorithmiques simples, structure de données, ...)
— AC 2 Comparer des algorithmes pour des problèmes classiques (tris simples, recherche, ...)

Projet de 12h, travail **en binôme**.

Contexte

Nous sommes en mars 2020. Le président de la République annonce le premier confinement à cause de la pandémie mondiale due au coronavirus. La raison? Il parait que moins vous vous déplacez, moins il y aura de contamination.

Effectivement, le Washingpost en parle et le prouve par une simulation :
`https://www.washingtonpost.com/graphics/2020/world/corona-simulator/`
Cette simulation vous amuse beaucoup et vous décidez de la redévelopper pour le fun.

Ça tombe bien. On est jeudi soir et vous avez beaucoup de temps à perdre, enfermé chez vous. Coïncidence incroyable, vous avez également tout juste fini de suivre le cours de python à l'IUT de Calais et vous êtes un super boss en la matière. Allez! Ça doit sûrement se coder en quelques minutes ces quelques cercles qui bougent sur un écran!

Après "quelques **trèèèèès longues** minutes", vous avez une première version...

Description de l'application

L'application (fichier `pandemic.py`) consiste en une interface graphique rectangulaire modélisant une zone géographique restreinte dans laquelle se baladent des personnes représentées par des cercles de couleur (voir figure 1). Le code couleur est le suivant :

— noir : personne en bonne santé,
— rouge : personne infectée,
— vert : personne guérie donc immunisée.

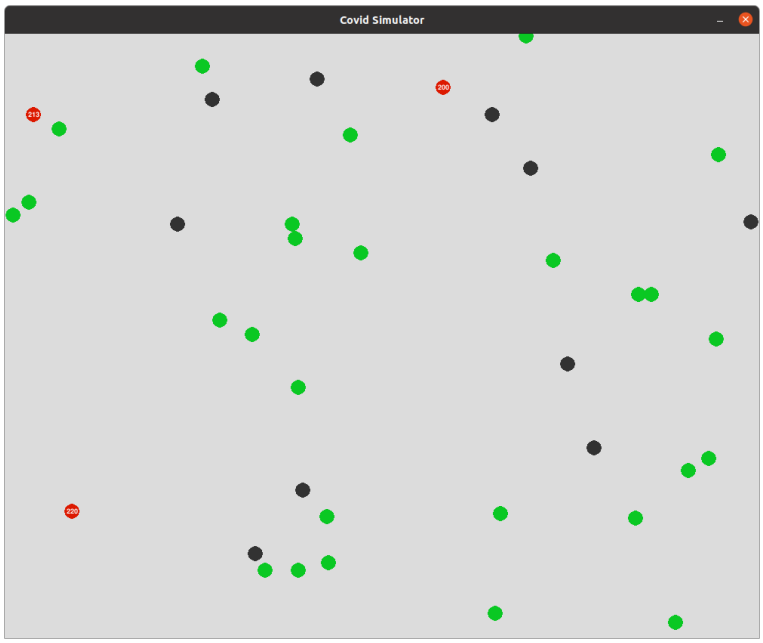


FIGURE 1 – Capture d'écran de la simulation de diffusion du covid.

L'application montre des personnes se déplaçant en ligne droite à vitesse constante. Les personnes en bonne santé sont affichées en noir, les personnes infectées en rouge avec un compteur qui se décrémente jusqu'à guérison. Ces personnes guéries passent alors en vert. Lorsqu'une personne infectée croise une personne en bonne santé, elle la contamine. La personne contaminée passe alors en rouge et le compteur s'affiche. Le programme s'arrête lorsque l'on ferme la fenêtre ou lorsque plus personne n'est infecté.

Les personnes sont décrites par une direction de déplacement, une vitesse (qui peut être nulle) et un état (en bonne santé, infecté, immunisé).

Les paramètres, tels que le nombre de personnes, le taux de personnes pouvant se déplacer, la valeur initiale du compteur avant guérison, etc. se trouvent dans le fichier `constants.py`.

Le fonctionnement général de cette application est très simple et est décrit grâce aux algos 1, 2 et 3.

Algorithm 1: Programme principal

```
1  persons ← créer la liste de personnes
2  fin ← faux
3  while non fin do
4      for toutes les personnes p de persons do
5          Mettre à jour la position de p
6          collisions = Calculer les collisions entre p et persons
7          Traiter collisions
8      if on ferme la fenêtre then
9          fin ← vrai
10     fin ← tester la fin du programme
11 Afficher les statistiques
```

Algorithm 2: Calculer les collisions entre `p` et la liste `persons`

```
1  collisions ← []
2  for toutes les personnes q de persons do
3      if intersection entre les cercles représentants p et q then
4          collisions ← collisions + (p, q)
5  return collisions
```

Lorsque vous exécutez le fichier `pandemic.py`, la simulation semble lente et saccadée. En regardant de plus près, on peut même s'apercevoir que toutes les personnes ne se déplacent pas en même temps.

Algorithm 3: Traiter les collisions

```
1 for toutes les collisions c de collisions do
2     p ← c[0]
3     q ← c[1]
4     if p en bonne santé et q infecté then
5         Passer p en infecté
6     if p en infecté et q en bonne santé then
7         Passer q en infecté
```

Toutefois, lors de la fermeture de l'application, les FPS ("Frame Per Second" ou encore IPS pour "Image Par Seconde") affichées dans le terminal ont une valeur élevées. Cette valeur est artificiellement élevée du fait de la non synchronicité de déplacement. En effet, une image est comptabilisée chaque fois qu'une personne est déplacée. Cette valeur n'a donc pas vraiment de sens. Il faudrait comptabiliser une nouvelle image seulement lorsque toutes les personnes ont été mises à jour.

Ces FPS peuvent être une évaluation de la performance de la simulation. Toutefois, pour évaluer la performance, un programme de benchmark est fourni (voir quelques lignes plus bas).

Vous pouvez faire vos propres tests en lançant la simulation plusieurs fois et en faisant varier la vitesse et le taux de personnes pouvant se déplacer. (Ne trainez pas trop non plus !)

Bien que des FPS soient affichées à la fermeture de cette application graphique, ce taux ne peut pas être considéré comme une évaluation correcte de la performance de notre algorithme de simulation. En effet, lors de l'exécution de cette **application graphique**, le programme doit calculer l'image à afficher, et ceci, de nombreuses fois. Ce calcul est très gourmand en ressource et ralentit donc nos calculs. Les FPS affichées ne reflétent donc pas la performance de nos algorithmes car ils comprennent le temps de calcul des images à afficher.

Pour pallier ce problème, en plus de cette application graphique, il vous est fourni un programme benchmark (fichier `bench.py`) permettant d'évaluer réellement les performances de la simulation. Toutefois, comme dit précédemment, les FPS moyens affichés dans cette **première version** n'ont pas vraiment de sens. En effet, ce benchmark simule la génération d'images et, dans cette première version, une image est générée à chaque fois qu'une personne se déplace. Or, pour être cohérent, il faudrait que les images ne soient comptabilisées que lorsque toutes les personnes se sont déplacées.

Le fichier de bench reprend exactement la même chose que la simulation graphique mais sans affichage.

Pour conclure, le programme graphique vous permettra de vérifier le bon fonctionnement de vos algos et le programme benchmark vous permettra d'évaluer les performances.

TODO

Le but du projet

Ce qui nous intéresse ici n'est pas la vitesse de diffusion du virus ni la correspondance avec la vérité mais les performances des algorithmes.

Le but de cette simulation est d'avoir des résultats sur le nombre de personnes en bonne santé, infectées ou guéries **le plus vite possible**. Nous essaierons alors d'avoir une simulation qui s'exécute le plus rapidement possible. L'optimisation de cette simulation passe par la réduction du nombre de calculs que le programme devra faire. Cette simulation repose essentiellement sur des tests d'intersection entre deux personnes.

Ce que vous allez et rendre!

Les différentes étapes vous demanderont de réaliser des optimisations en modifiant l'existant. Toutes ces étapes sont indépendantes. Si vous n'arrivez pas à répondre à l'optimisation demmandée, vous pourrez passer à la suivante sans résoudre la précédente.

Lors de ce projet, vous devrez produire du code commenté, lisible, etc. mais également un compte-rendu dans lequel vous répondrez aux différentes questions. Dans chaque étape, on vous demandera d'optimiser le code du point de vue d'un thème particulier et vous devrez, entre autre, analyser le résultat. Pour chaque étape, vous devrez rendre le code final de cette étape seulement (donc sans les étapes suivantes). N'oubliez donc pas de faire des sauvegardes intermédiaires.

Au final, vous devrez produire :

— **Pour chaque étape, un code source** respectant toutes les normes que vous avez vues en cours (nommage des variables, indentation, commentaires, etc.)

— **Un seul document PDF** contenant les réponses aux questions posées. Il vous sera demandé de faire régulièrement des tests de performances. N'hésitez pas à utiliser des graphiques pour plus de clarté.

Étape 1 : Optimisation de l'affichage

Vous pourrez constater que la simulation est saccadée bien que les FPS soient très élevés. Comme expliqué précédemment, ceci est dû au fait que l'affichage est mis à jour après chaque déplacement d'une personne.

Or, on s'attend à ce que tout le monde se déplace en même temps afin d'avoir une simulation fluide.

(code) - Dans un premier temps, vous allez modifier la simulation pour que toutes les personnes se déplacent en même temps. La mise à jour de l'affichage doit se faire une fois que toutes les personnes se sont déplacées et pas à chaque déplacement d'une personne. Bien que le nombre de FPS va diminuer, il deviendra cohérent et la simulation gagnera en fluidité.

(code) - Faites la même modification dans le fichier de benchmark.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte. Réalisez ensuite un graphique qui montre les performances de votre optimisation en faisant varier le nombre de personnes.
Utilisez bien le programme de benchmark!!! C'est le seul programme qui vous donnera une évaluation correcte de la performance de votre simulation.

⚠ N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 2 : Optimisation mathématique

Vous allez maintenant optimiser la simulation d'un point de vue mathématique. Votre simulation passe le principal de son temps à tester des intersections entre des cercles. Cette partie du code doit donc être la plus efficace possible. Ce code se trouve dans la fonction `circleCollision` du fichier `engine.py`. Elle prend le centre de deux cercles en paramètre. Les rayons sont connus (`PERSON_RADIUS`) et présents dans le fichier `constants.py`.

Appelons `c1` et `c2` les deux cercles passés en paramètre. L'idée principale pour vérifier si `c1` intersecte `c2` (ou inversement) est de tester si un point `p` contenu dans `c1` se trouve également dans `c2` (voir figures 2 et 3). Si un tel point peut être trouvé alors les deux cercles s'intersectent.

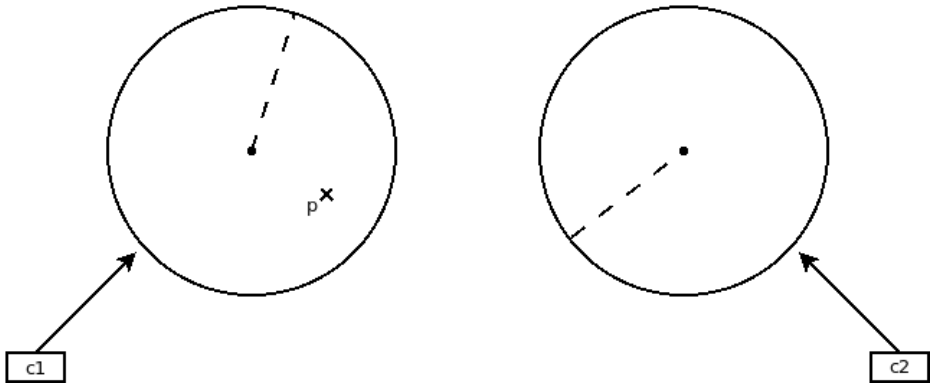


FIGURE 2 – Pas d'intersection entre `c1` et `c2`. Aucun point `p` appartenant à `c1` et `c2` ne peut être trouvé.

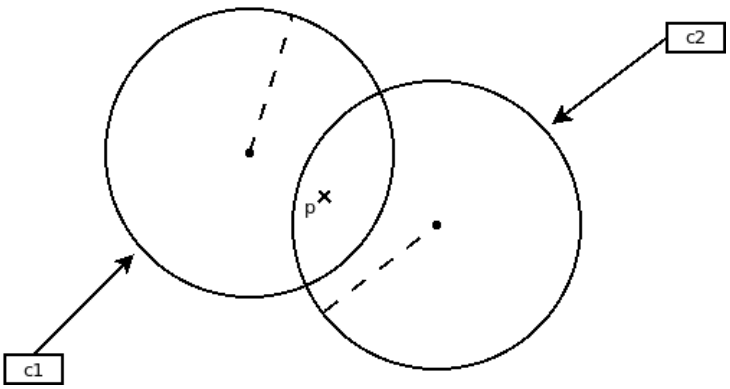


FIGURE 3 – Intersection entre `c1` et `c2`. On a trouvé au moins un point `p` de `c1` appartenant également à `c2`.

Le but de cette fonction (`circleCollision` du fichier `engine.py`) est donc de trouver un point `p` de `c1` appartenant à `c2`. Si un tel point est trouvé alors les cercles s'intersectent, sinon il n'y a pas d'intersection. Il est évident que les points `p` testés ne doivent pas être pris au hasard. Les points `p` testés seront donc pris sur le bord du cercle `c1` à intervalle régulier et

seront assez rapprochés. Ainsi, si un point du bord du cercle appartient également à l'autre cercle alors nous aurons prouvé leur intersection.

Pour vérifier si le point `p` appartient au cercle `c2`, il suffit de calculer la distance entre `p` et le centre du cercle `c2`. Si cette distance est plus petite que le rayon du cercle alors le point appartient bien au cercle.

L'algorithme de test d'intersection est décrit sur l'algo 4.

Algorithm 4: Tester l'intersection de `c1` et `c2`

```
1 # c1 désigne le centre du cercle c1
2 # c2 désigne le centre du cercle c2
3 for angle ← 0 à 2 × π par pas de 0.1 do
4     p_x ← c1_x + cosinus(angle) * rayon de c1
5     p_y ← c1_y + sinus(angle) * rayon de c1
6     if distance(c2,p) < rayon de c2 then
7         return vrai
8 return faux
```

C'est cet algorithme que vous allez devoir optimiser!

(code) - L'algorithme de test d'intersection de cercles décrit précédemment peut être qualifié de totalement éclaté et loin d'être efficace! Trouvez un meilleur algorithme et remplacez ainsi la fonction toute pourrie!

Rappel : on essaie de supprimer un maximum de calculs et notamment les calculs lourds pour la machine comme les racines carrées. Si vous pouvez, évitez de les utiliser.

(code) - N'oubliez pas de reporter, si nécessaire, les modifications apportés à votre code dans le fichier de benchmark.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte. Réalisez ensuite un graphique qui montre les performances de votre optimisation en faisant varier le nombre de personnes.

⚠N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 3 : Optimisation par approximation

Si vous avez réussi à optimiser l'algorithme de test d'intersection des cercles à l'étape précédente, vous avez dû voir une forte hausse de la performance de la simulation. Toutefois, votre algorithme se base encore sur un calcul de distance. Ce calcul nécessite des opérations mathématiques très lourdes et complexes pour un ordinateur (même si vous avez réussi à ne pas utiliser de racine carrée).

Dans cette étape, nous allons encore optimiser cet algorithme en supprimant définitivement tout calcul complexe de la fonction de test d'intersesction.

Pour ce faire, vous pouvez utiliser les boites englobantes (bounding box en anglais) des cercles. Le calcul d'intersection de boites englobantes est plus efficace que le calcul d'intersection entre les cercles eux-mêmes. Toutefois, il vous faudra garder en tête que nous cherchons à effectuer le moins de calcul possible. Vous devrez donc précalculer les boites englobantes avant leur utilisation et les mettre à jour seulement lorsque ce sera nécessaire.

(code) - Utilisez cette structure pour encore optimiser le calcul des intersections. Remplacez le test d'intersection des cercles par un test d'intersection des boites englobantes des cercles.

(code) - N'oubliez pas de reporter, si nécessaire, les modifications apportés à votre code dans le fichier de benchmark.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte. Réalisez ensuite un graphique qui montre les performances de votre optimisation en faisant varier le nombre de personnes.

(PDF) - Quel est l'inconvénient de cette méthode ? Argumentez et illustrez vos propos.

⚠ N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 4 : Optimisation logique/algorithmique

Si vous avez réussi à utiliser les boites englobantes à l'étape précédente, vous avez dû voir une nouvelle forte hausse de la performance de la simulation. Et bien ce n'est pas fini!!!

Dans l'état actuel de la simulation, lorsqu'une personne est déplacée, un calcul d'intersection est fait avec toutes les autres personnes. Or, seules les personnes infectées peuvent contaminer les personnes en bonne santé. Tenez compte de cette nouvelle remarque pour optimiser le code.

(code) - Optimisez le nombre de tests de collision en ne les effectuant que lorsque c'est nécessaire. On ne va donc faire les calculs que lorsque l'on est sûr qu'ils sont pertinents. Inutile de faire les tests d'intersections lorsque la personne n'est pas infectée.

(code) - N'oubliez pas de reporter, si nécessaire, les modifications apportés à votre code dans le fichier de benchmark.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte. Réalisez ensuite un graphique qui montre les performances de votre optimisation en faisant varier le nombre de personnes.

(PDF) - Quel inconvénient pouvez vous trouver à cette optimisation ? Comment l'expliquez-vous ? (Indice : lancez l'application graphique plusieurs fois et soyez attentif)

⚠ N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Conclusion

(PDF) - Comparez toutes les étapes d'optimisation en un seul graphique et commentez. Cette conclusion devra être faite **en anglais** en comparant les différents algorithmes mis en place, leur type d'optimisation et une quantification de leur impact sur le nombre de calcul réalisable en un temps fini.

Partie optionnelle

1. Maintenant, normalement l'exécution de l'application est devenue bien trop rapide pour que vous puissiez voir tout ce qui se passe durant le déroulement d'une simulation. Vous pouvez modifier l'application graphique afin de limiter la vitesse de simulation. Pour ce faire, vous devrez mesurer le temps utilisé pour faire la mise à jour de toutes les personnes et faire une pause de la bonne durée juste après la mise à jour de l'affichage. Ceci limitera les FPS et vous pourrez ainsi contrôler la vitesse de simulation (comme par exemple à 30 FPS). Attention de bien mesurer le temps de calcul nécessaire au déaplcement de toutes les personnes à chaque tour de boucle car les temps ne sont pas forcément constants entre deux tours.

2. Avez vous d'autres idées ? Expliquez les nous et, si vous avez le temps, codez les !