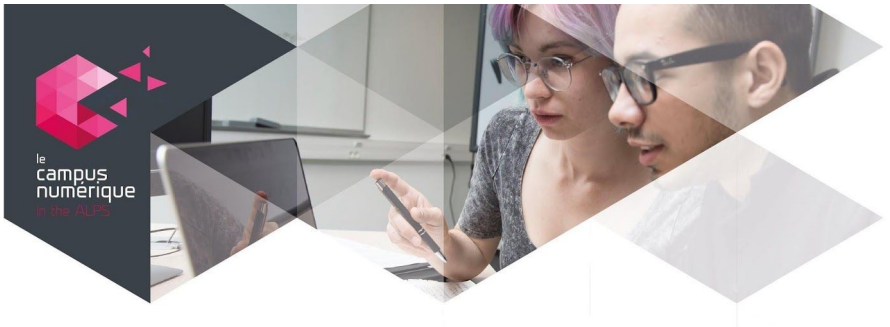


Algorithmique

Labyrinthe

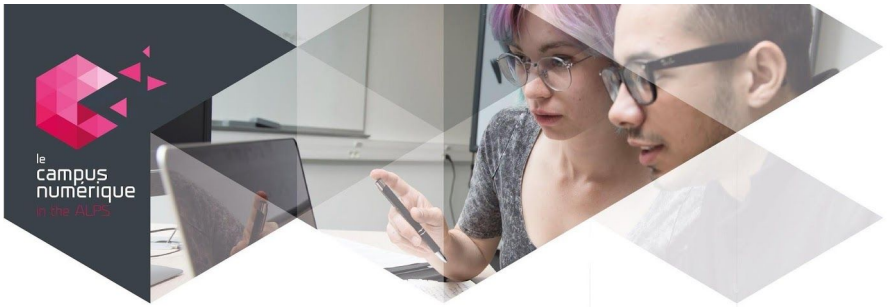
Projet “Algorithmique avancée”

Objectifs pédagogiques

- Découvrir une nouvelle classe de problèmes, qui peuvent être modélisés par des graphes, et dont la résolution revient à parcourir ce graphe.
- Découvrir différents algorithmes de parcours de graphe

Compétences pédagogiques

- Implémenter un modèle de données pour représenter un labyrinthe
- Implémenter un parcours de graphe en profondeur
- Implémenter un parcours de graphe en largeur



<- Le jeu du labyrinthe ->

Rendus

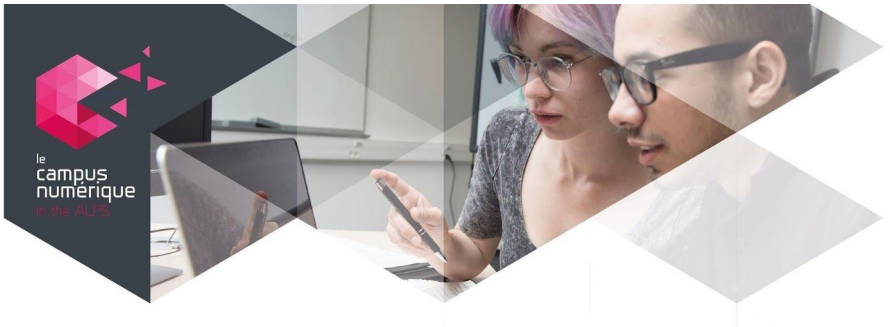
Au fil des étapes, les rendus doivent être “commits” sur un repository github structuré (ex un dossier par étape). Les rendus sont le code source, les réponses aux questions, les pseudo-code, etc.

Le travail doit être réalisé seul sur les étapes de code, en ilôt sur les étapes de réflexion et d’écriture du pseudo-code.

Situation - Problème

Vous êtes nouvellement arrivé·e pour un premier stage en entreprise. Cette dernière est spécialisée dans la réalisation de versions web de jeux anciens, et souhaite rajouter à sa bibliothèque un **jeu du labyrinthe**.

Un·e précédent·e stagiaire, vous a laissé un export JSON contenant la description de plusieurs labyrinthes générés aléatoirement. Ce qu’il vous reste à faire est de créer une procédure permettant de **résoudre le labyrinthe automatiquement.**

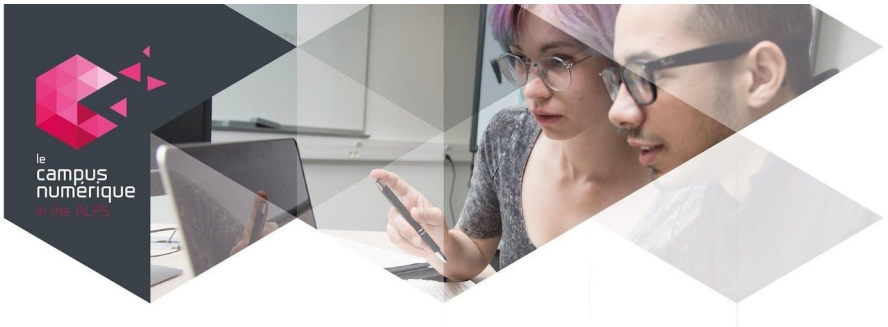


Alternative au labyrinthe

Pour les plus rapides, qui ont terminé toutes les étapes du labyrinthe, vous pourrez adapter votre code pour résoudre d'autres problèmes nécessitant un parcours de graphe. Par exemple la résolution du taquin (https://fr.wikipedia.org/wiki/Taquin). La logique est strictement la même, mais est conceptuellement un poil plus abstraite (et donc un peu plus dure). Demandez au formateur, le support qui était donné précédemment à la place du labyrinthe.

Conseil général

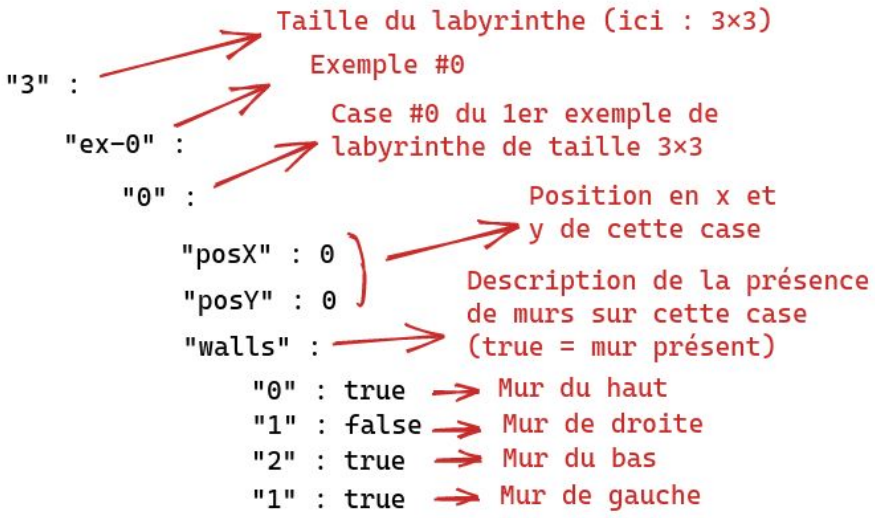
Pensez à régulièrement lâcher la souris et le clavier et à travailler **ordinateur éteint** avec papier et stylo. Il est toujours très important de prendre du recul par rapport à son code. **Une étape fondamentale est la réflexion autour de la manière de structurer vos données**.



Projet Labyrinthe

Etape 1 : Construction du labyrinthe

La première étape nécessaire à ce projet est la construction du labyrinthe en lui-même. Un fichier JSON contenant les informations nécessaires à la création du labyrinthe vous est fourni. Il est structuré de la manière qui suit :



Vous avez à votre disposition 144 labyrinthes dans ce fichier JSON : trois exemples de labyrinthe pour différentes tailles, allant de 3 x 3 cases à 25 x 25 cases. Commencez bien par les plus petits, ce qui vous permettra de débugger votre code.

Objectif de cette étape :

1. Servez vous de ce fichier JSON pour créer le fameux labyrinthe. Le rendu final doit ressembler à la figure 1.
2. Passez du temps **hors ordinateur** pour réfléchir à la structure des données la plus adaptée au projet *(j'ai déjà dit ça ?)*. La "qualité" de votre représentation des données déterminera la difficulté des étapes suivantes !

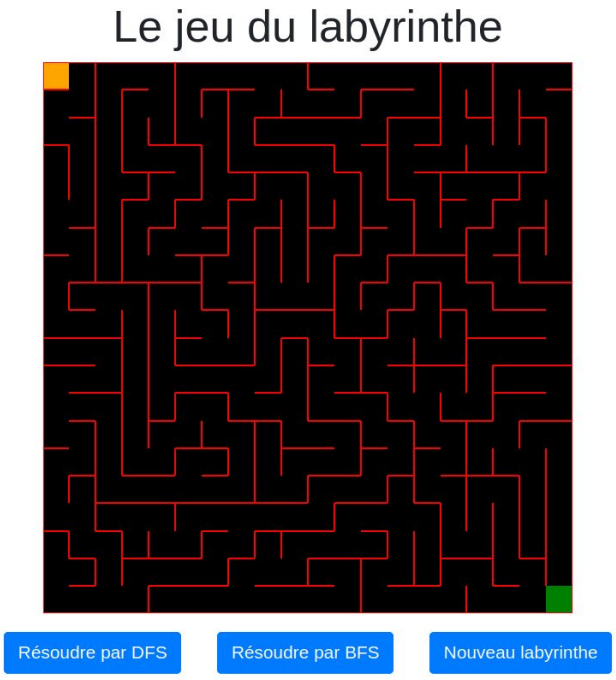


Figure 1. Exemple de labyrinthe généré.
L'entrée est en orange, la sortie en vert, les murs en rouge.

Etape 2 : Résolution du labyrinthe - Intuition :

Objectif de cette étape :
Etablir une première méthode de résolution "à votre sauce".

Etapes :

- Chacun de votre côté, pendant 15 minutes, imaginez vous dans le labyrinthe, et décrivez comment vous pourriez trouver la sortie (en vert) depuis l'entrée (en orange).
- Ecrivez votre algorithme sous forme de pseudo-code (pas de formalisme particulier attendu : https://fr.wikipedia.org/wiki/Pseudo-code)
- Expliquez votre algorithme à votre îlot. Comparez les pour ensuite votre algorithme à celui de votre binôme. Êtes-vous arrivés indépendamment au même algorithme de recherche ?
- **Il n'est pas demandé d'optimisation**. C'est normal que votre algorithme se perde en chemin, et que ce ne soit pas "le plus court". Assurez vous simplement que quel

soit le labyrinthe fourni, vous puissiez trouver à coup sûr la sortie du labyrinthe.

Pistes de réflexion :

- Afin de faciliter la réflexion, imaginons que le labyrinthe soit de taille réduite, par exemple : 6 x 6 cases. Une fois la logique établie sur un petit labyrinthe, elle devrait se généraliser sur un labyrinthe de taille quelconque.

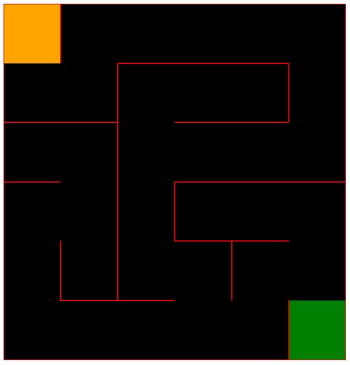
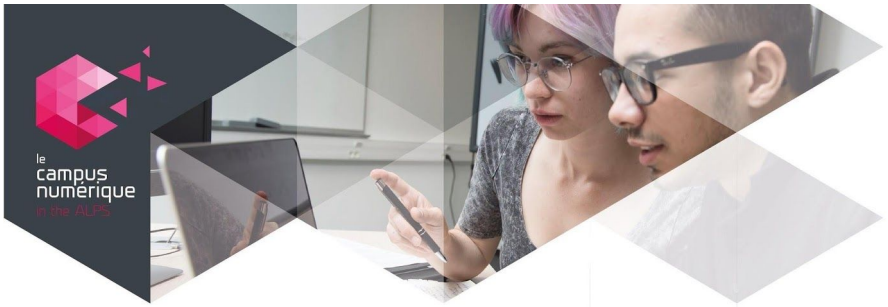


Figure 2 : Exemple de labyrinthe de taille 6 x 6

- Vous pouvez emporter un sac de riz avec vous dans le labyrinthe.

Figure 3 : Un indice de taille !



Etape 3 : Résolution du labyrinthe - Parcours DFS :

Objectif de cette étape :
Implémenter une méthode de recherche de chemin.

Etapes :

1. Comparez votre algo “intuition” avec le parcours dit **Depth First Search (DFS) version itérative** (parcours en profondeur d’abord). Avez-vous eu une idée similaire?

   Version itérative de DFS

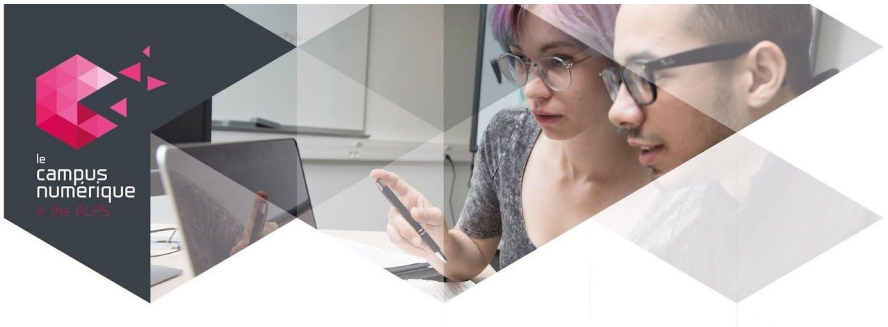
```
DFS-iterative (G, s):
    → let S be stack
    → insert s in the stack
    → mark s as visited.

    while ( S is not empty):
        → Pop a vertex “v” from stack to visit next
      for all neighbours w of v in Graph G:
          if w is not visited :
                  → insert w in the stack
                  → mark w as visited
```

2. Jouer cet algorithme en utilisant des post-its.
3. Quelques ressources pour comprendre :
   - https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/
   - https://brilliant.org/wiki/depth-first-search-dfs/
   - https://www.youtube.com/watch?v=qt6oMBKRxEU
4. Il est plusieurs fois question de graphes dans ces explications. Quel est le rapport avec votre projet ? Êtes-vous capable de représenter le labyrinthe sous forme de graphe ?

Attendus de cette étape :

- Comprendre l’algorithme de parcours DFS.
- Dessiner un exemple de labyrinthe et sa représentation sous forme de graphe.



5. Codez cet algorithmes de résolution DFS
    - Langage à utiliser -> JS
    - Affichez à chaque étape les coordonnées de la case courante.
    - Lorsque vous trouvez la sortie, arrêtez le programme et retournez le résultat.
    - Affichez le nombre d'étapes qui ont été nécessaires avant de trouver la sortie.
6. Essayez de rendre cet algorithme récursif. Vous êtes peut-être déjà tombés sur la solution, si tel est le cas, essayez quand même de la retrouver par vous même ! C'est un très bon exercice. Et codez donc cet algorithme également.

Conseil et notes :

- Utilisez votre débugger comme si votre vie en dépendait.
- Si vous avez peur de la récursivité, c'est normal. Allez voir cette ressource :
    - Récursivité : https://www.freecodecamp.org/news/how-recursion-works-explained-with-flowcharts-and-a-video-de61f40cb7f9/
    - Call stack (pile d'appel) : https://www.shmoop.com/computer-science/recursion/call-stack.html
    - Et n'hésitez pas à demander une remédiation sur ce thème aux formateurs.

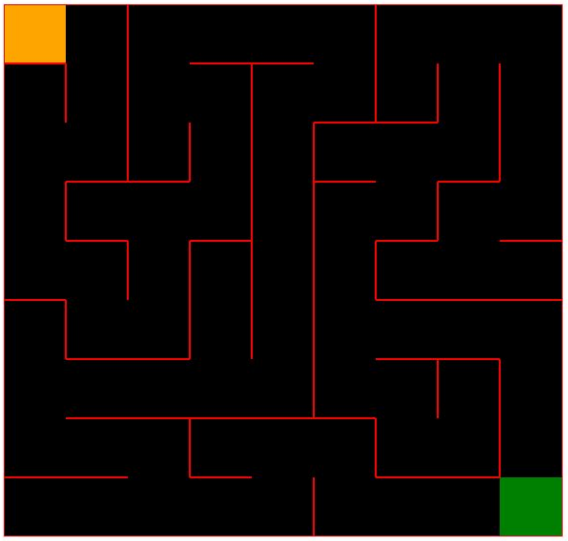
Etape 4 : Résolution du labyrinthe - Parcours DFS (V2) :

Objectif :

- Extraire l'information de chemin

Etapes :

- Lorsque vous trouvez la sortie, l'algorithme ne doit plus seulement vous indiquer qu'il a trouvé la sortie, mais il doit également retourner le chemin depuis l'entrée du labyrinthe jusqu'à la sortie. Vous choisirez un format pertinent pour ce retour.
- L'algorithme vous retourne également la longueur du chemin à parcourir.
- Pour ceux qui le veulent : afficher une représentation graphique dynamique de cet algorithme. Voir GIF#1.



GIF#1 : Exemple d'animation possible sur DFS.
En noir les zones pas encore explorées. En Bleu : la case courante évaluée. En Violet les zones explorées. En gris, les impasses identifiées. En vert, la sortie. En rouge, les murs du labyrinthe.

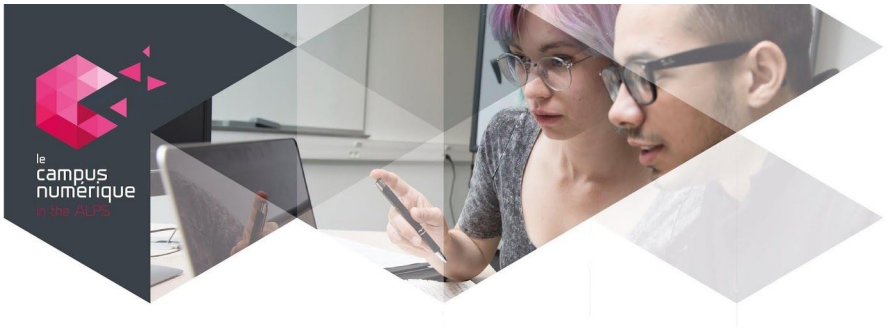
Etape 5 : Résolution du labyrinthe par une autre méthode - Parcours BFS :

Objectif :

- Implémenter d'un autre algorithme de recherche : **Breadth First Search** (BFS) → parcours en largeur d'abord.

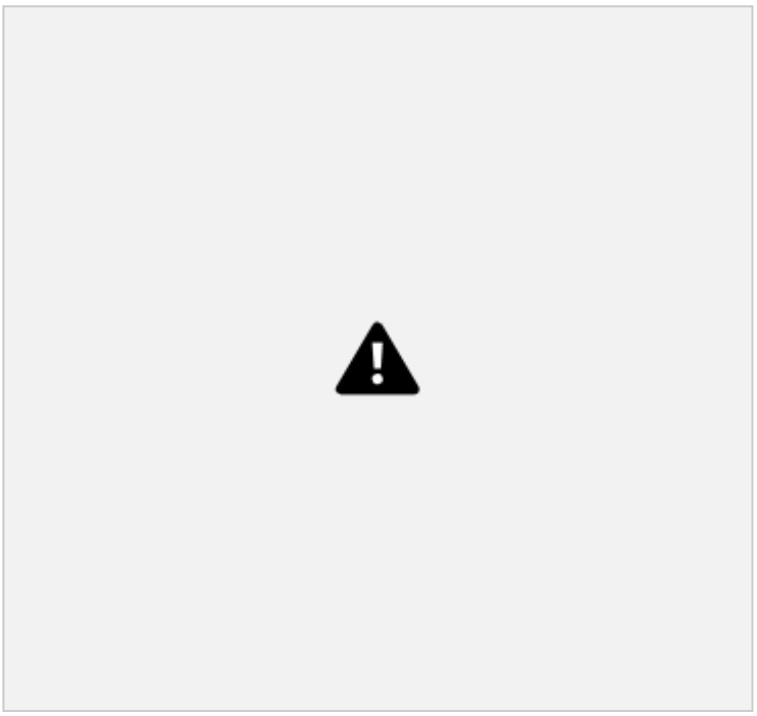
Etapes :

- Il existe de nombreuses façons de parcourir des graphes. Une autre manière adaptée, dans le cas du labyrinthe est la recherche BFS
  Ressource : https://www.youtube.com/watch?v=ec0IJsiIuWk
- Essayez de comprendre seuls et en îlot le pseudo code suivant :




```
BFS (G, s)
      → let Q be queue.
      → add s to the queue.
      → mark s as visited.
      while ( Q is not empty )
            → get next element waiting in the queue, called v, and
              remove it from the queue
            → if v is winning, return v
            for all neighbours w of v in Graph G
                if w is not visited
                      → add w to the queue
                      → mark w as visited.
      → return false
```

- Implémentez l'algorithme BFS, en suivant les mêmes étapes que pour le DFS.
- Essayez de modifier votre programme de manière à faire tomber quelques murs aléatoires dans le labyrinthe. Assurez-vous que vos algos de résolution fonctionnent encore.
- Comparez les résultats du DFS et BFS dans ce cas.
    a. DFS et BFS trouvent-ils toujours le même chemin ?
    b. Quelles sont les différences de résultat ? pourquoi ?



GIF#2 : Exemple de résolution par BFS.
En noir les zones pas encore explorées. En Bleu : la case courante évaluée. En Violet les zones déjà explorées. En orange, la frontière courante. En vert, la sortie. En rouge, les murs du labyrinthe.

Etape 6 : Pour aller plus loin :

Objectif:
Aller creuser les concepts développés jusqu'à maintenant.

Option #1 :

- Fabriquer un générateur aléatoire de labyrinthe
  - Les données que vous avez reçu pour générer vos labyrinthes ont été obtenues par implémentation d'un **"Recursive randomized depth-first search"**. Vous pouvez choisir votre méthode.
  - Analyse : quels sont les cas qui ne sont pas couverts par cette méthode ? Est-ce réellement un labyrinthe le plus général possible. Vous pouvez-vous aider d'une représentation en graphe pour aboutir à la bonne solution.
  - **Ressource** : https://en.wikipedia.org/wiki/Maze_generation_algorithm

Option #2 :

- Appliquer votre solution à la résolution de taquin.