Méthodes locales et pratiques de résolutions de labyrinthes

Comment sortir d'un labyrinthe ?

Paul Mabileau

Plan

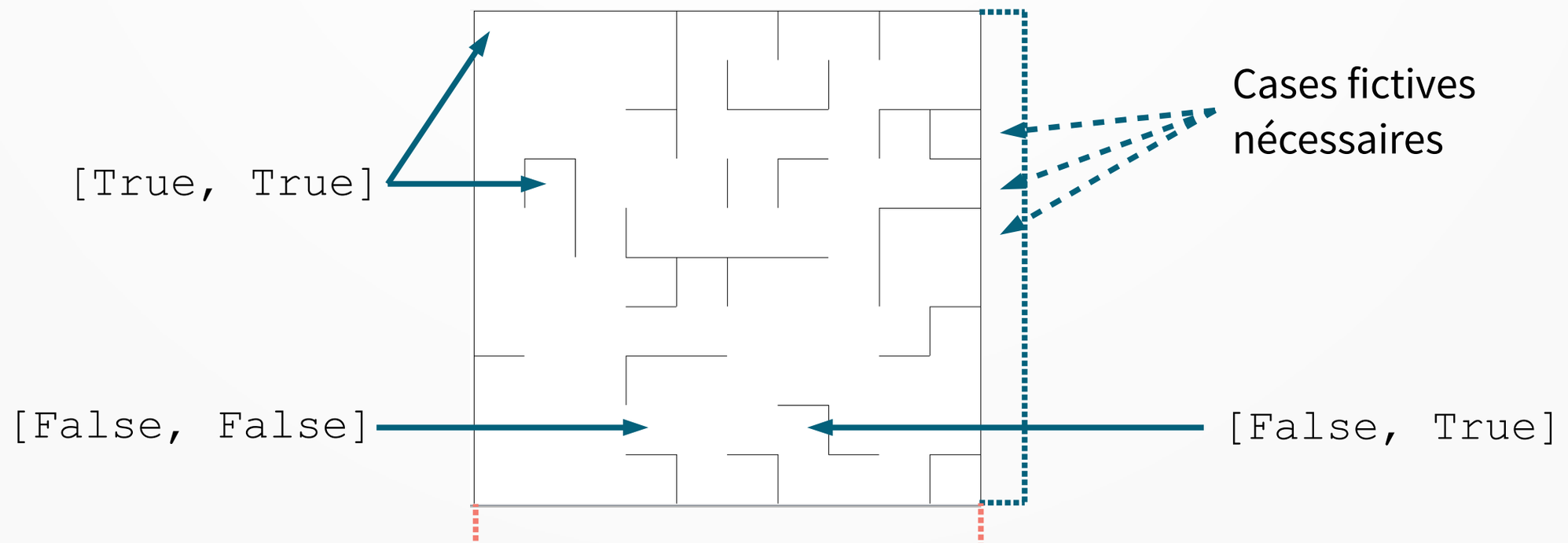
Plan

Introduction et motivations

- L'option informatique de CPGE propose d'étudier en partie la théorie des graphes, ce qui nous a permis de généraliser la notion d'arbre pour modéliser des relations encore plus abstraites. Un exemple d'utilisation sont les labyrinthes.
- Dans ce cadre, on connaît une méthode pour calculer un chemin de plus courte longueur entre deux points d'un labyrinthe, ce qui permet en particulier d'en trouver la sortie. Ceci semble alors simple, mais cela requiert la donnée de tout le labyrinthe, alors : comment ferais-je, moi, pour sortir ?

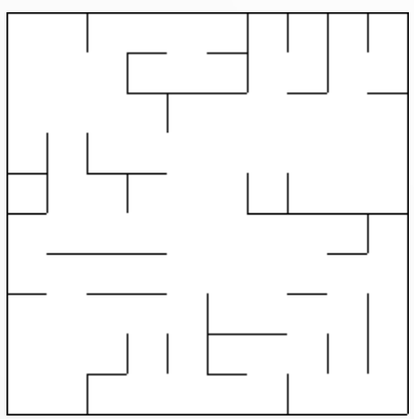
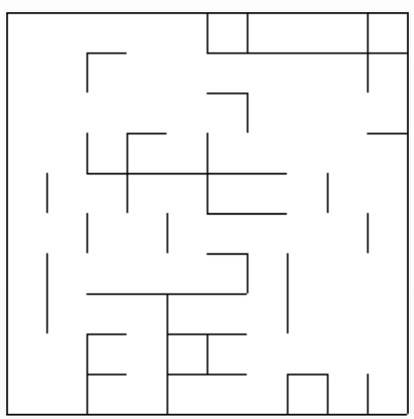
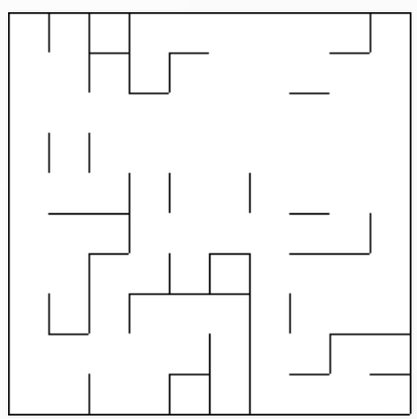
Représentation des données

- On représente un labyrinthe par une matrice où chaque coefficient est un couple de booléens indiquant la présence ou non d'un mur à gauche puis en haut de la case.



I. Générations a. Aléatoire

- En partant d'un labyrinthe vide (enceinte comprise), chaque mur est présent au final avec une certaine probabilité donnée en entrée, et ce, indépendamment des autres.

- On remarque de premier abord que certaines cases ou régions sont isolées. Si la probabilité est trop haute, on risque de ne pas pouvoir sortir.

Théorie de la percolation

- En percolation planaire, on se munit du graphe $(\mathbb{Z}^2, E^2)$

  où E^2 est constitué des paires de voisins $\{(x_1, y_1); (x_2, y_2)\} \subset \mathbb{Z}^2$ tq $(x_1 - x_2)^2 + (y_1 - y_2)^2 = 1$

- Soit $p \in [0,1]$. On prend l'espace probabilisé (Ω, T, P_p) qui porte la famille $(X_e)_{e \in E^2}$ de variables aléatoires discrètes mutuellement indépendantes suivant toutes $\mathcal{B}(p)$. On appelle $\omega \in \Omega$ une configuration et est telle que

$$\forall e \in E^2, \quad X_e(\omega) = \begin{cases} 1 & \text{si } e \text{ est ouverte} \\ 0 & \text{sinon} \end{cases}$$

Chemins et événements

- On appelle chemin une suite $x_0, x_1, \cdots$ de sommets distincts tels que $\forall i \in \mathbb{N}, \{x_i ; x_{i+1}\} \in E^2$. C'est donc auto-évitant.

- Un circuit est un chemin fini $x_0, x_1, \cdots, x_n$ tel que $\{x_n ; x_0\} \in E^2$

- On note l'événement $x_0 \leftrightarrow x_n$ « il existe un chemin entre x_0 et x_n » et $x_0 \leftrightarrow \infty$ « il existe un chemin infini de départ x_0 ».

Probabilités

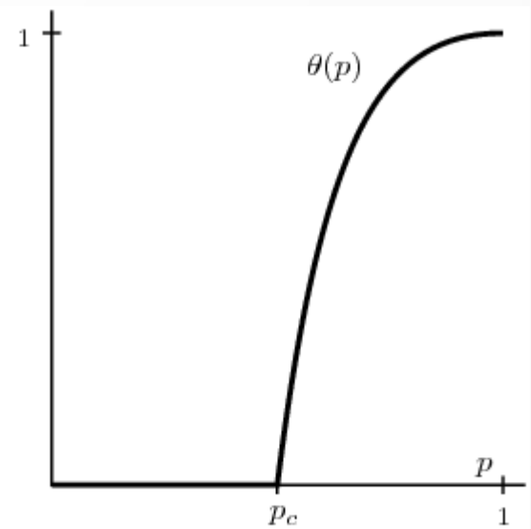
- On pose $\theta(p) := P_p(0 \leftrightarrow \infty)$ la probabilité d'existence d'une composante infinie dans le réseau carré. $\theta(0) = 0$ et $\theta(1) = 1$
- Cette fonction est croissante. On raisonne par couplage : soient $(p_1, p_2) \in [0;1]^2$ telles que $p_1 < p_2$. On se munit des familles indépendantes $(X_e)_{e \in E^2}$ avec $X_e \to \mathcal{B}(p_2)$ et $(Y_e)_{e \in E^2}$ où $Y_e \to \mathcal{B}\left(\frac{p_1}{p_2}\right)$ puis $X'_e = X_e \cdot Y_e \to \mathcal{B}(p_1)$ Donc $X'_e \leq X_e$
- Ainsi, on a les configurations aux niveaux p_1 et p_2 , et si $0 \leftrightarrow \infty$ pour p_1 alors $0 \leftrightarrow \infty$ pour p_2 aussi. D'où

$$\boxed{\theta(p_1) \leq \theta(p_2)}$$

Transition de phase

- La percolation admet un phénomène de transition de phase :

$$\exists\, p_c \in [0;1],\ \forall\ p \in [0;1],\ \begin{cases} p \leq p_c \Rightarrow \theta(p) = 0 \\ p > p_c \Rightarrow \theta(p) > 0 \end{cases}$$



- En effet, on note la probabilité critique comme borne supérieure d'une partie bornée de $\mathbb{R}$: $p_c := \sup\{p \in [0;1] : \theta(p) = 0\}$
- On a donc une limite inférieure pour la percolation : si les arêtes ne sont pas assez souvent ouvertes (hormis jamais), nous ne sommes pas sûr de pouvoir passer. On a surtout :

$$0 < p_c < 1$$

Preuve 1 : $p_c > 0$ et $\theta(0) = 0$

Soient $p \in [0;1]$ et $n \in \mathbb{N}$. On note Ω_n l'ensemble des chemins de $\mathbb{Z}^2$ de longeur n partant de 0.
On a alors l'encadrement suivant :

$$\begin{aligned} 0 \leq \theta(p) = P_p(0 \leftrightarrow \infty) &\leq P_p(\exists\,(x_1,\dots,x_n) \in \Omega_n,\ \forall\ k \in [\![1;n]\!],\ X_{x_k}(\omega) = 1) \\ &\leq \sum_{(x_0,\dots,x_n) \in \Omega_n} P_p(\forall\ k \in [\![1;n]\!],\ X_{x_k}(\omega) = 1) \\ &= |\Omega_n|\ p^n \leq 4 \cdot 3^{n-1} \cdot p^n \end{aligned}$$

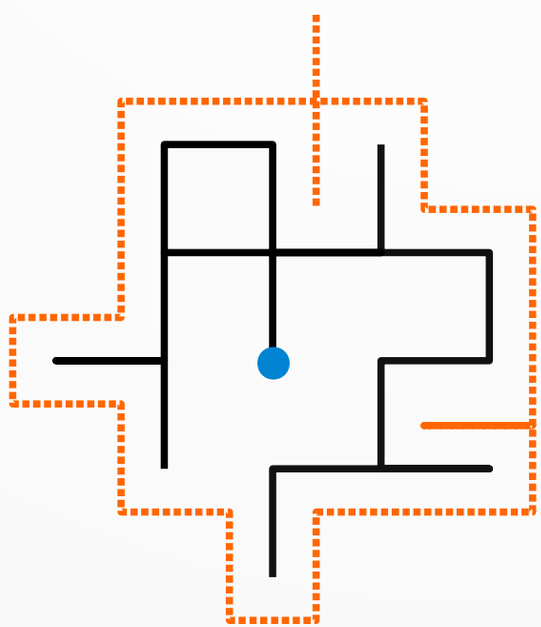
En appliquant ce résultat avec $p < \frac{1}{3}$, le dernier membre de droite tend vers 0 quand $n \to \infty$.

Donc $\forall\ p \in \left[0;\frac{1}{3}\right[$, $\theta(p) = 0$.

D'où $p_c \geq \frac{1}{3} > 0$.

Preuve 2 : $p_c < 1$ et $\theta(1) = 1$

On va s'intéresser ici à la quantité $1 - \theta(p) = 1 - P_p(0 \leftrightarrow \infty) = P_p(|C_0| < \infty)$ où C_0 est la composante connexe du réseau carré contenant 0. On utilise la notion de graphe dual. Le graphe dual de $(\mathbb{Z}^2, E^2)$ est le graphe $(\mathbb{Z}^{2*}, E^{2*})$ avec $\mathbb{Z}^{2*} = \mathbb{Z}^2 + (\frac{1}{2}, \frac{1}{2})$ et E^{2*} est composé de toutes les arêtes e^* associée à e telles que e et e^* se coupent orthogonalement en leur milieu. Dans notre cas, on donne aux arêtes e et e^* le même état d'ouverture. Ainsi, si $|C_0| < \infty$, les arêtes duales sont fermées autour de 0 :



On trouve alors un circuit d'arêtes duales fermées entourant l'origine.

Preuve 2 : $p_c < 1$ et $\theta(1) = 1$

En notant Γ_n l'ensemble des circuits duaux de longeur n entourant 0, on a :

$$
\begin{aligned}
P_p(|C_0| < \infty) &\leq P_p\left(\exists\, (x_0^*, \ldots, x_n^*, x_0^*) \in \Gamma_n,\ \forall\ k \in [\![1;n]\!],\ X_{x_k^*}(\omega) = 0\right) \\
&\leq \sum_{n \geq 4} \sum_{(x_0^*, \ldots, x_n^*, x_0^*) \in \Gamma_n} P_p\left(\forall\ k \in [\![1;n]\!],\ X_{x_k^*}(\omega) = 0\right) \\
&\leq \sum_{n \geq 4} |\Gamma_n| (1-p)^{n+1} \leq \sum_{n \geq 4} 4 \cdot 3^{n-1} n (1-p)^{n+1} \underset{p \to 1}{\to} 0
\end{aligned}
$$

$$
\Rightarrow \qquad \boxed{\lim_{p \to 1} \theta(p) = 1 = \theta(1) \text{ et } p_c < 1}
$$

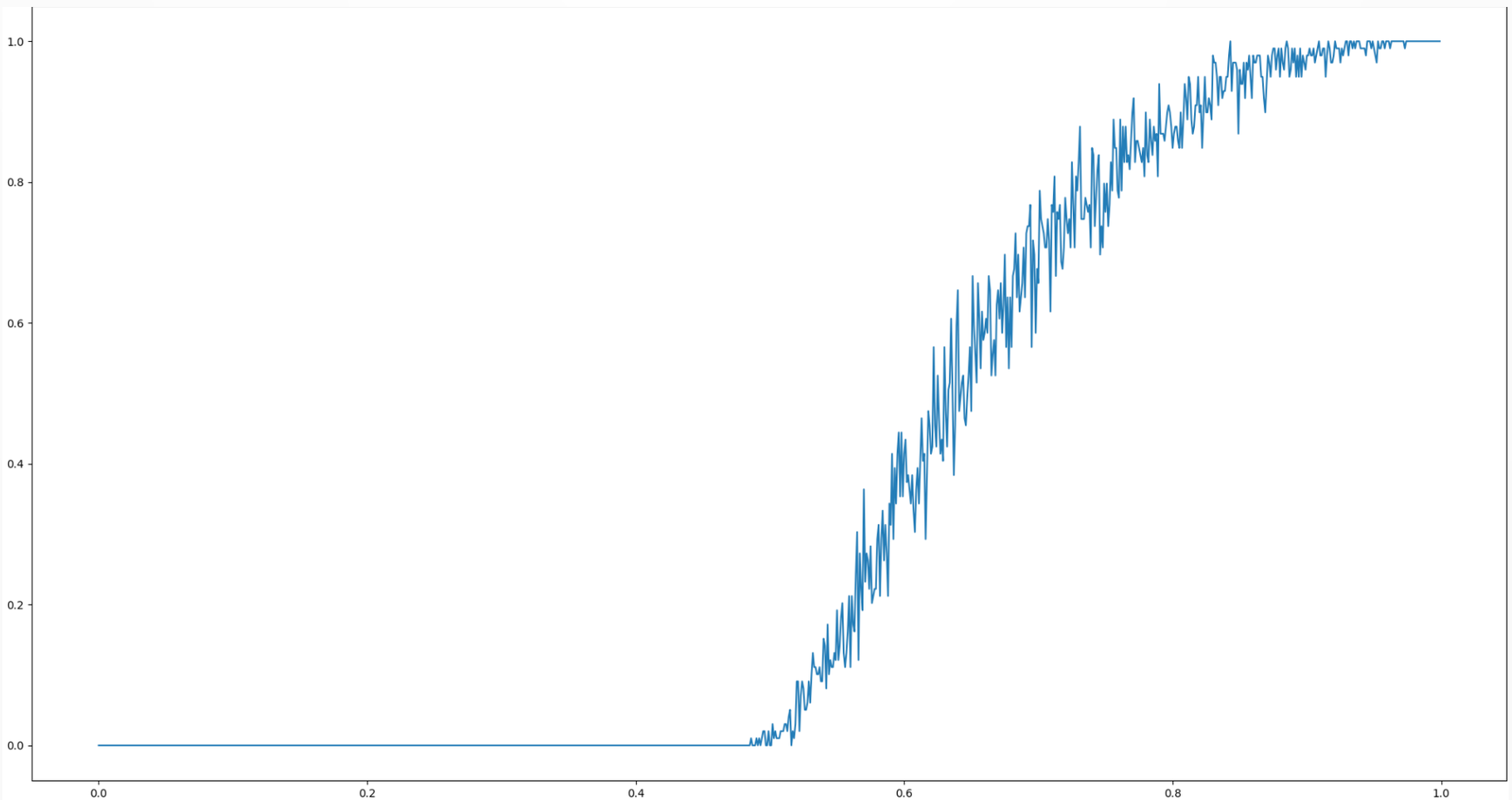
Ainsi

$$
\boxed{\frac{1}{3} \leq p_c < 1}
$$

Probabilité critique en dimension 2

$p_c = \frac{1}{2}$: tests réalisés sur $[\![0\,;100]\!]^2$, avec 100 essais pour un pas de probabilité 0.1

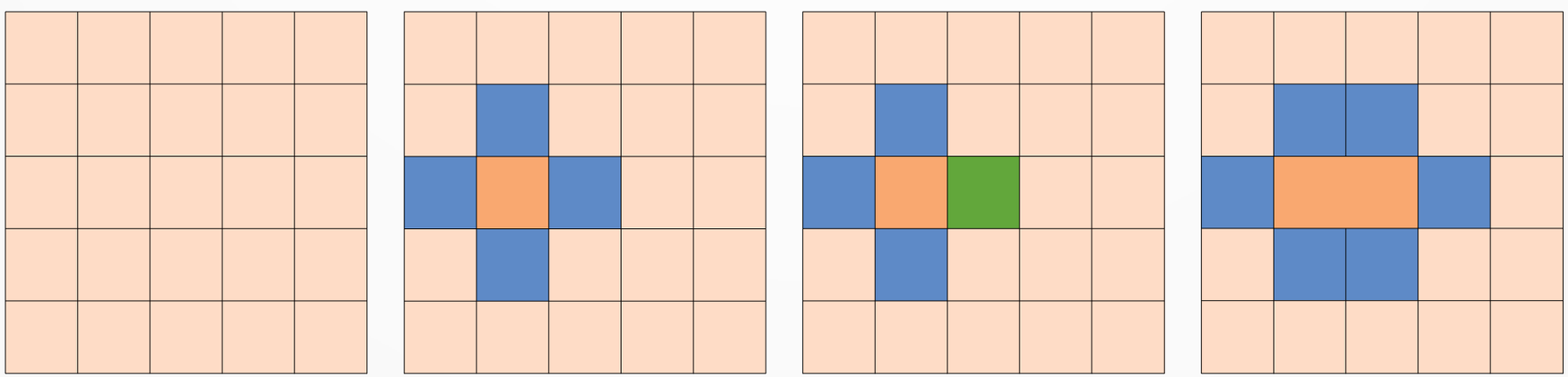
Succès moyen



Probabilité p

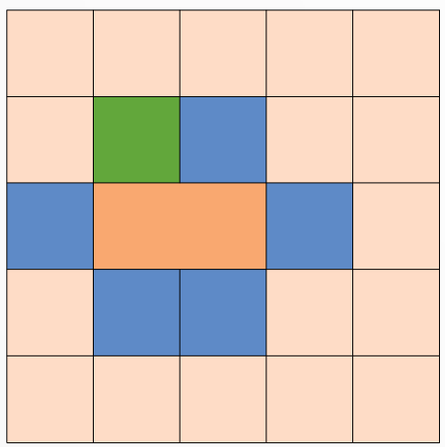
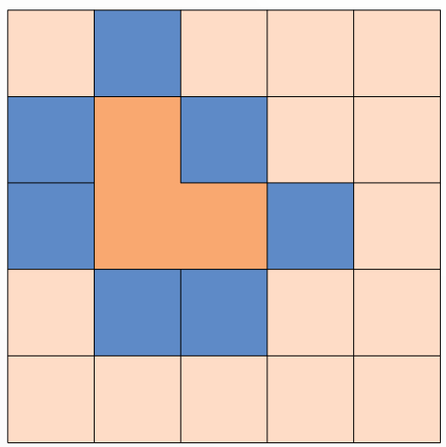
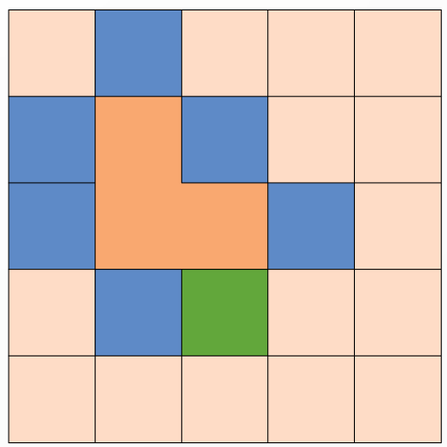
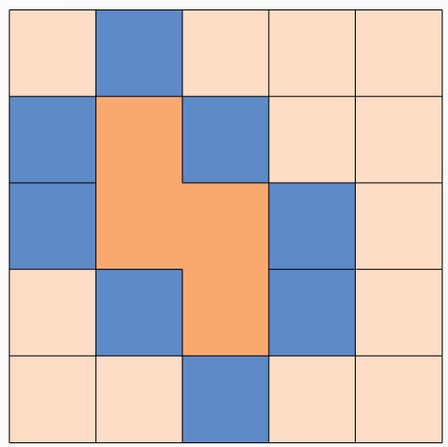
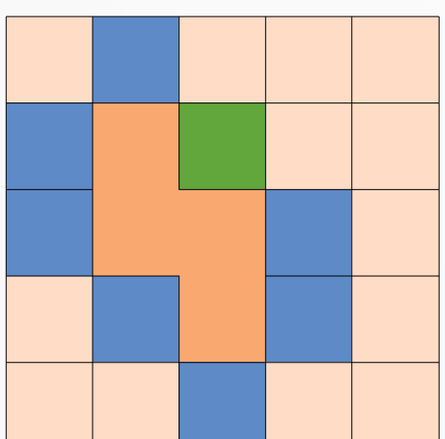
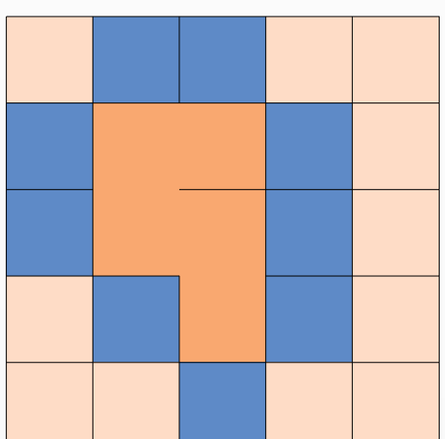
I.b. Génération parfaite

- Un labyrinthe est dit parfait si entre tout couple de cases, il existe un unique chemin auto-évitant. Pour en générer aléatoirement, on utilise une variante de l'algorithme de Prim :



I.b. Génération parfaite

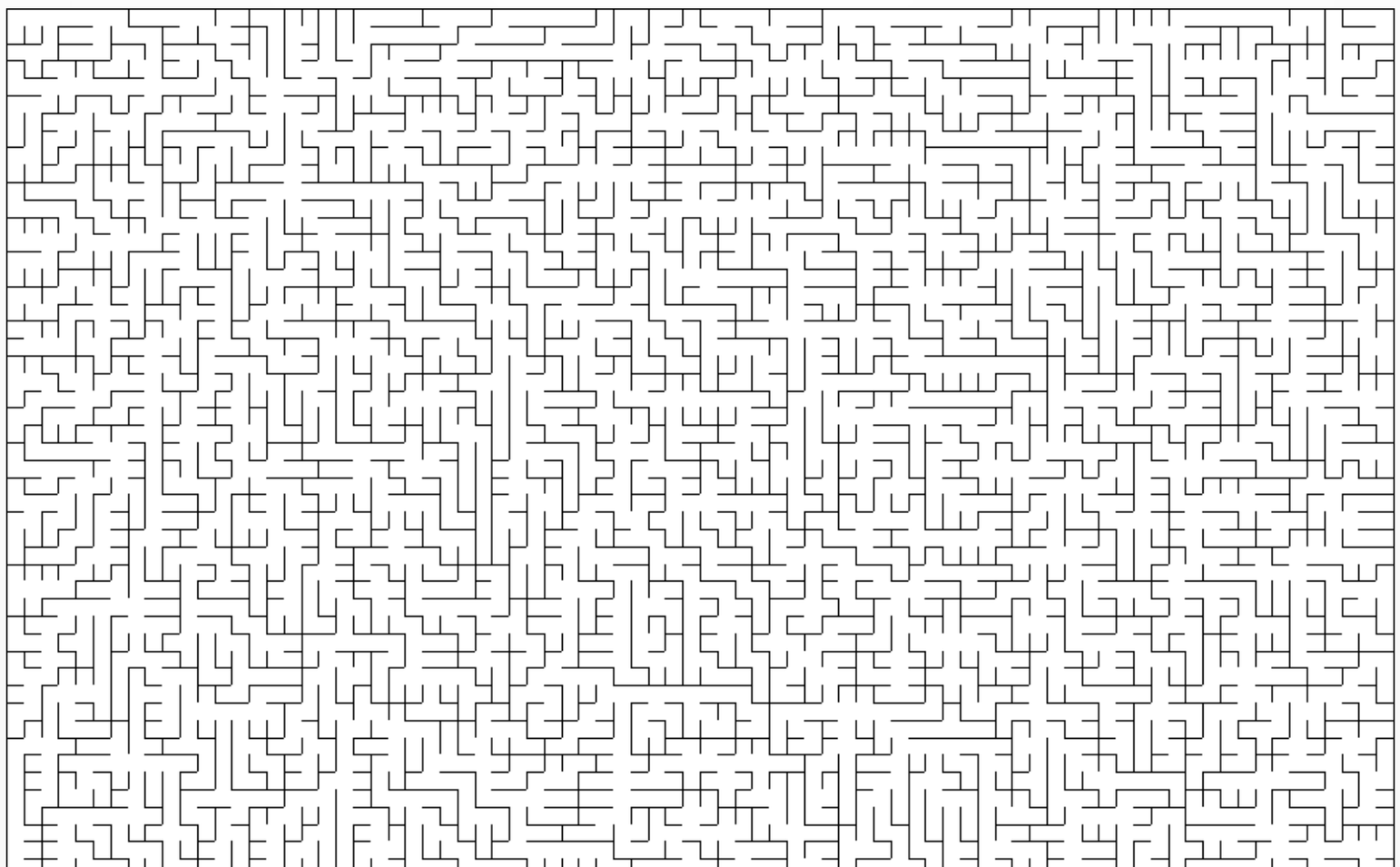
- Puis :

. . . .

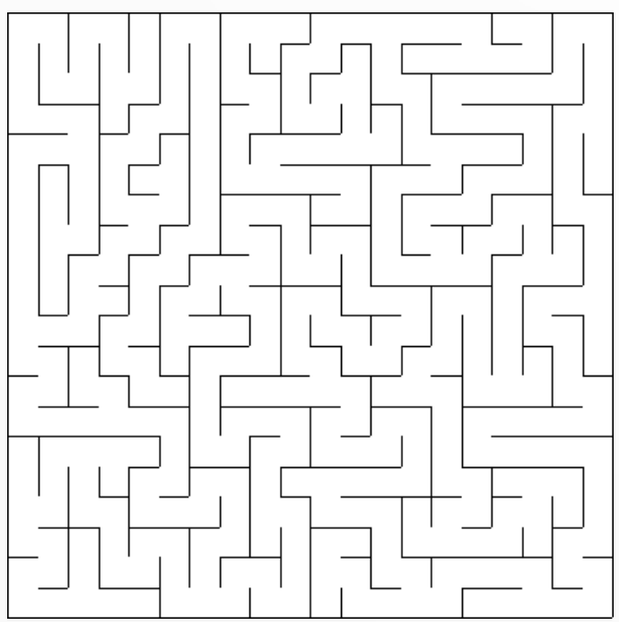
I.b. Génération parfaite

- On obtient par exemple :

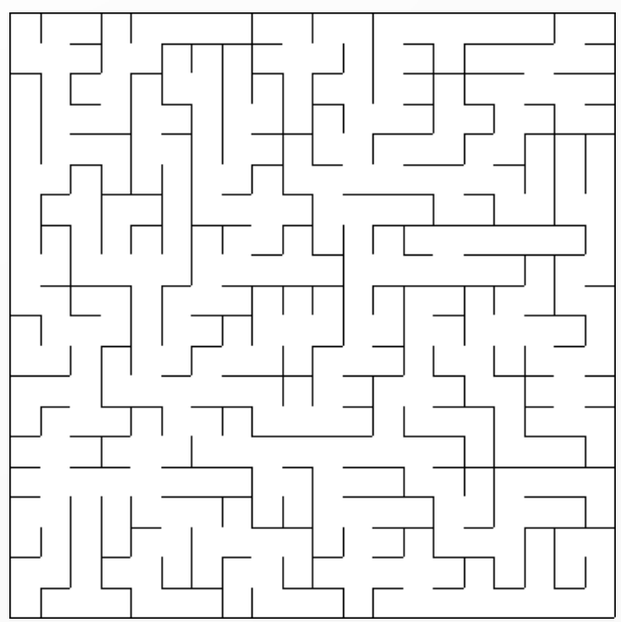


I.b. Génération parfaite

- Cet algorithme a été choisi car, il a tendance à créer plus de « branches » que d'autres, notamment un algorithme récursif de type « depth first » qui réalise les chemins d'abord.



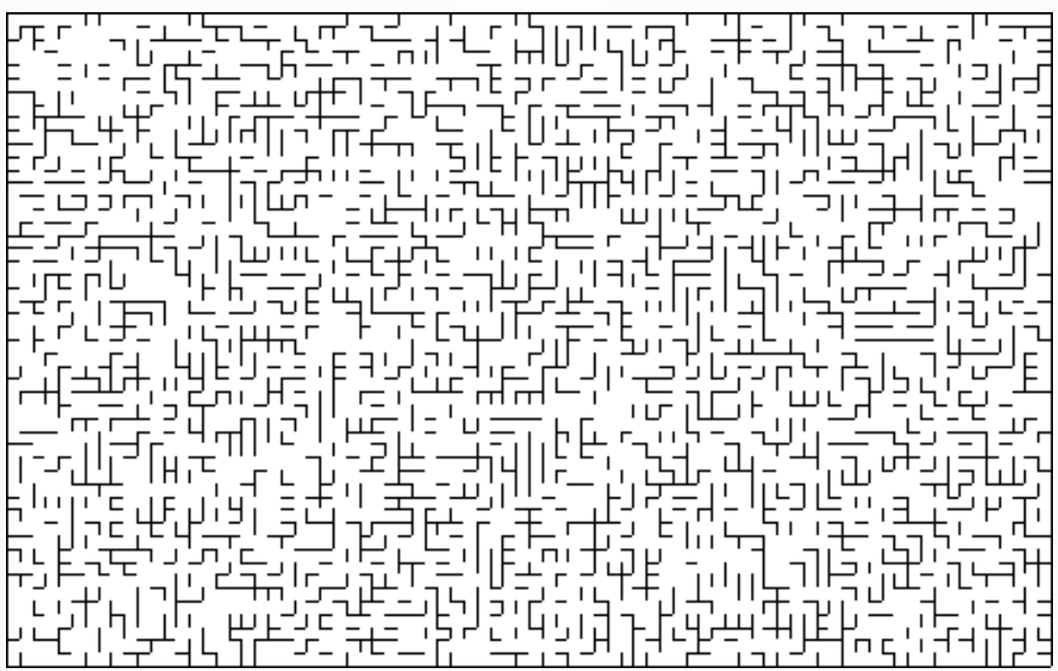
Recursive backtracker



Prim

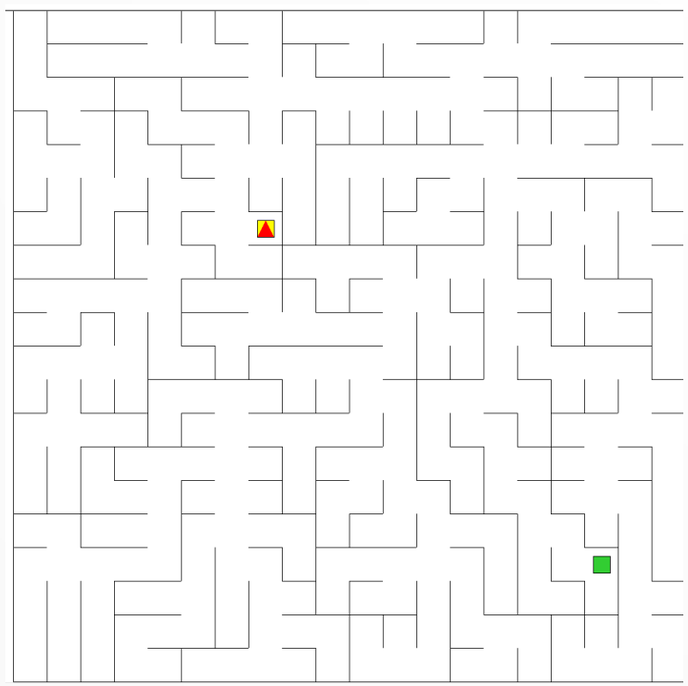
I.c. Génération presque parfaite

- A partir d'un labyrinthe parfait aléatoire, on retire chaque mur avec une probabilité donnée en entrée, et ce, indépendamment des autres. On a donc toujours au moins un chemin entre toute paire de cases. Par exemple :

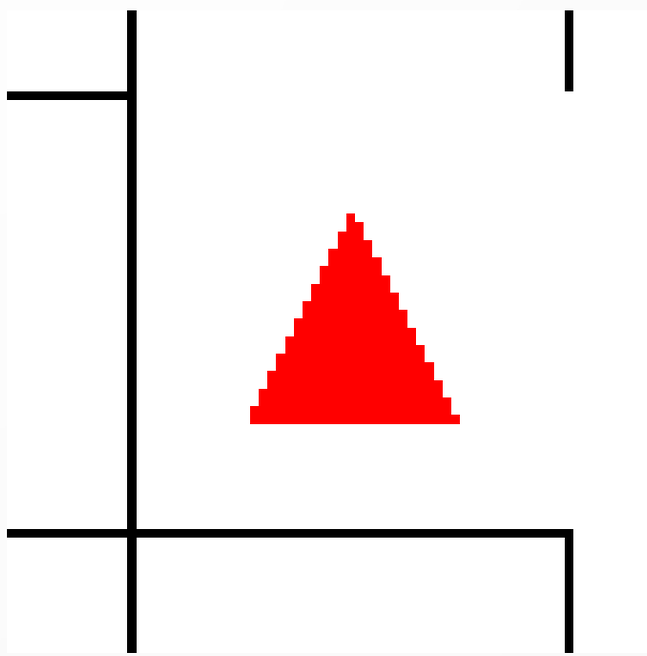


II. Recherche de chemins

- On le fait dans deux contextes distincts :



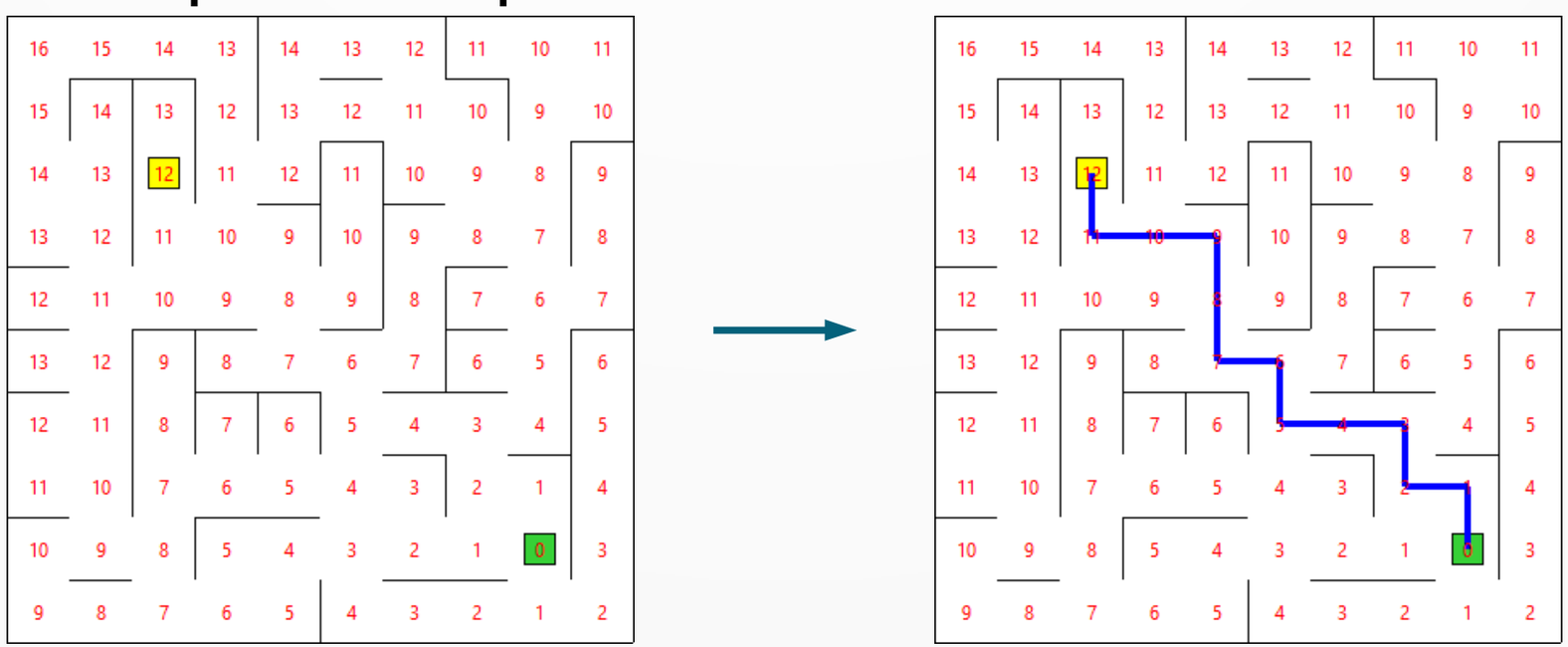
Vision globale



Vision locale

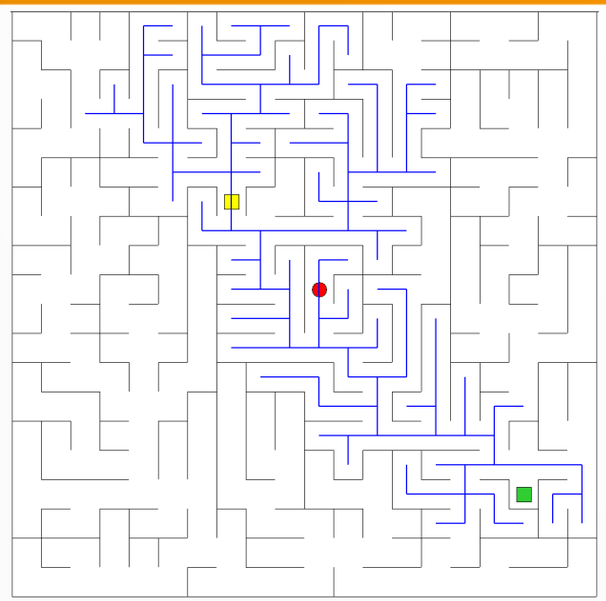
II.a. Avec connaissance globale

- Dans ce cas, on est en mesure de déterminer un plus court chemin entre deux points, ce qui permet en particulier de rejoindre la sortie.
- Une méthode de Dijkstra simplifiée nous donne la carte des distances à partir du point d'arrivée :

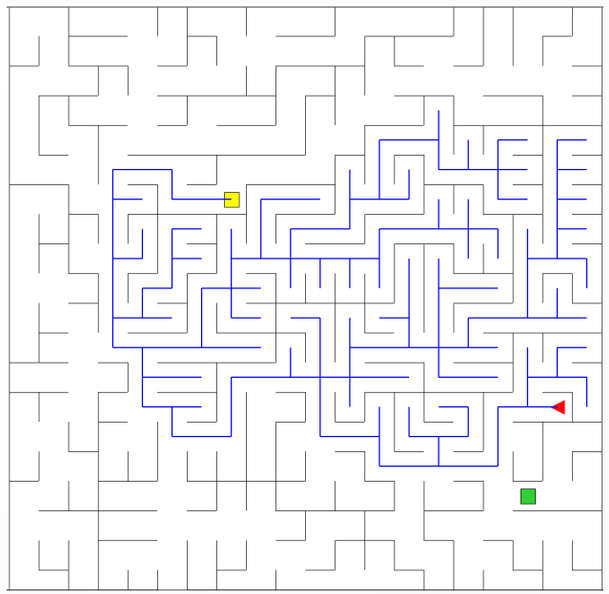


II.b. Sans connaissance globale

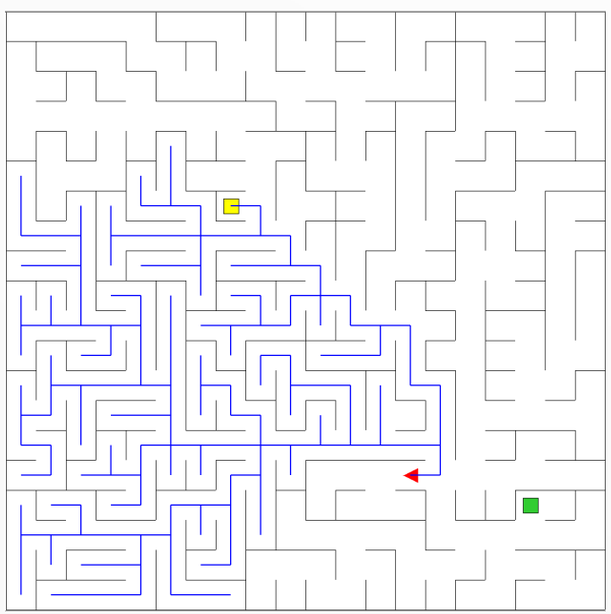
Aléatoire



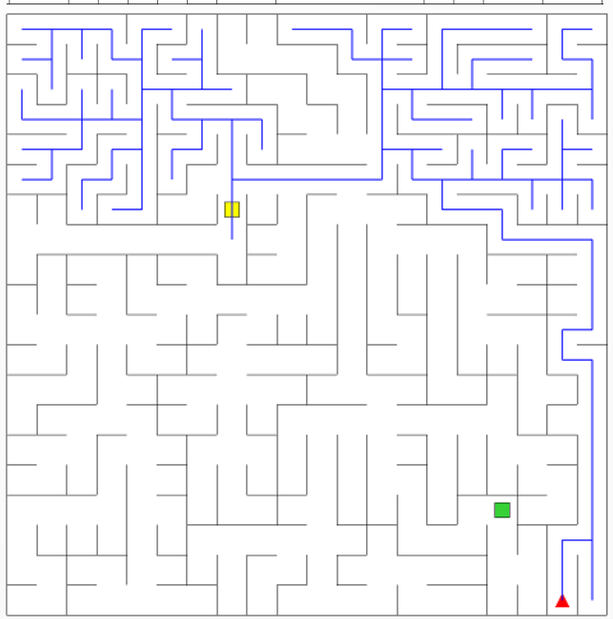
Main gauche



Pledge

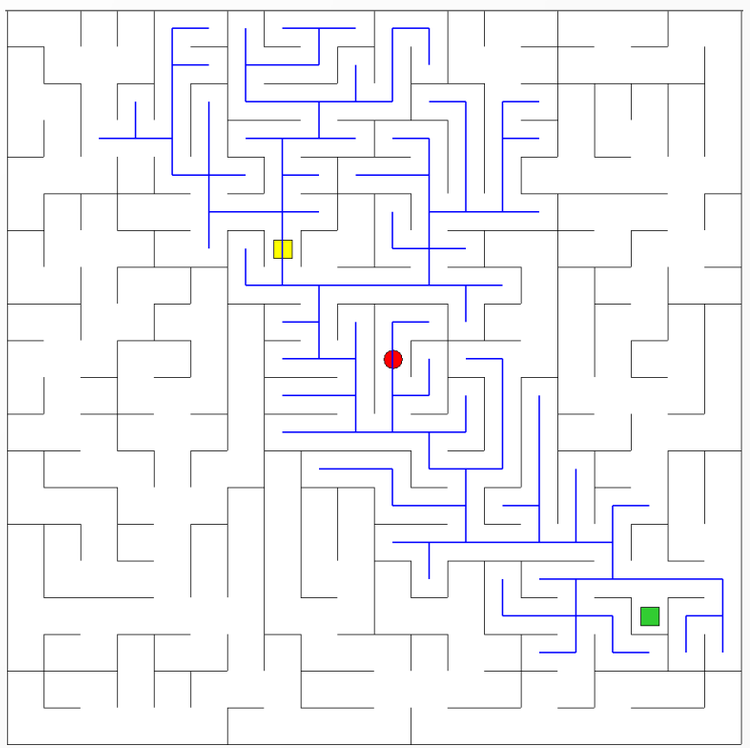


Trémaux



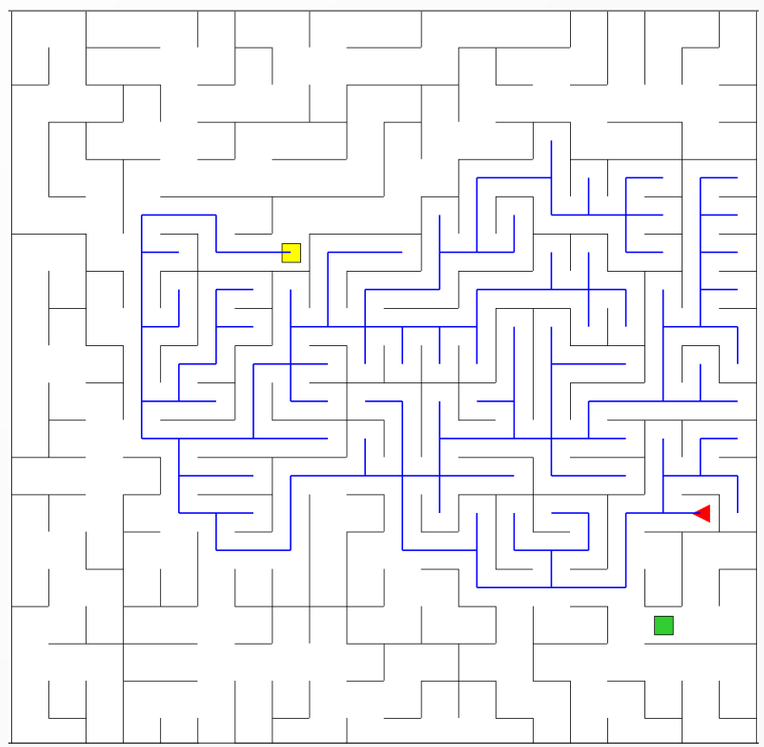
Première idée : l'aléatoire complet

- A chaque étape, on tire aléatoirement et de manière indépendante des choix précédents une direction (Nord, Est, Sud ou Ouest). Si le mouvement est possible, on le réalise, sinon on ne fait rien.
- Désavantage immédiat : très lent et peu convaincant.



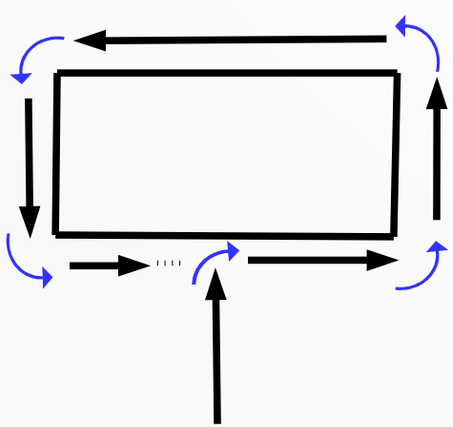
Première méthode : la main gauche

- Il s'agit ici d'apposer une main (toujours la même) sur un mur et de le suivre pour garder la main en contact.
- L'avantage est que c'est une méthode très simple mais qui permet une exploration – partielle – d'un labyrinthe parfait : cela en fait un parcours en profondeur.

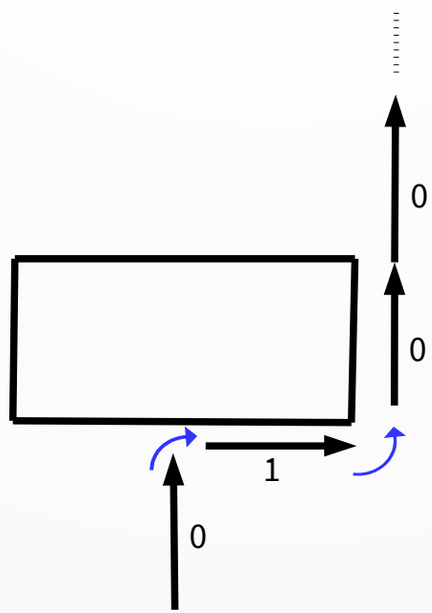


Deuxième méthode : Pledge

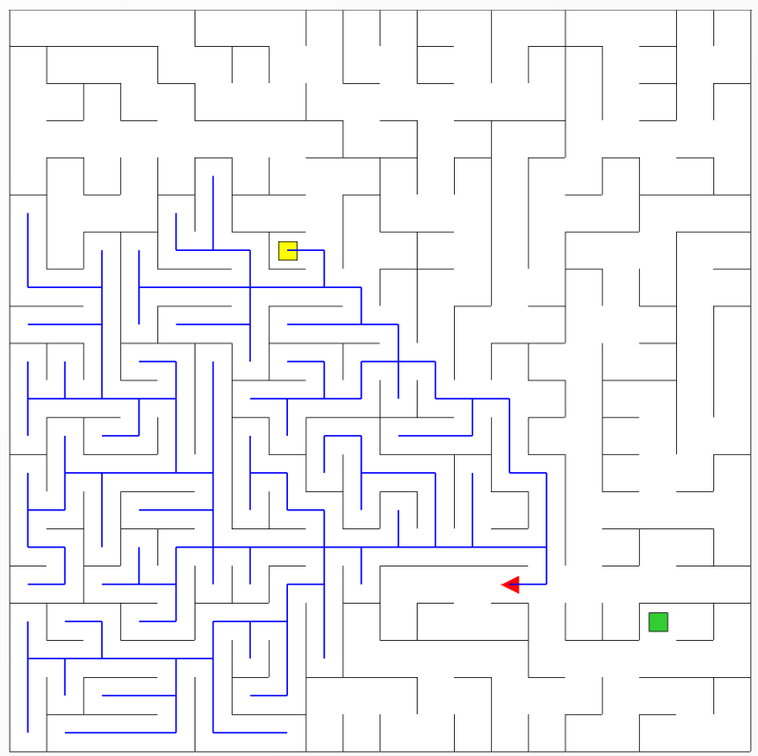
- L'algorithme de Pledge améliore la méthode de la main gauche grâce à un compteur qui permet d'éviter les îlots :



Main gauche

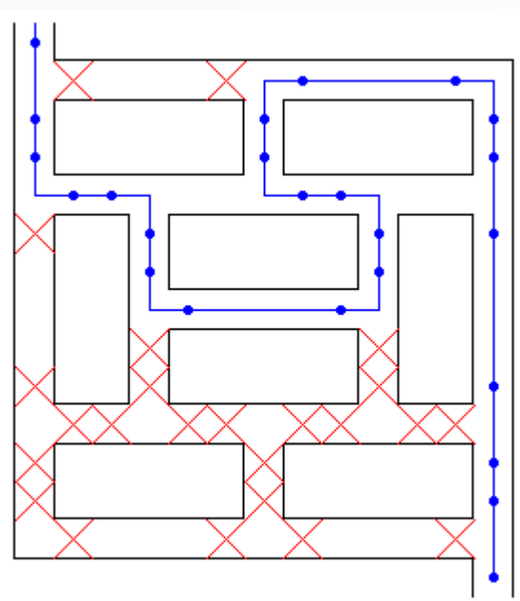


Pledge

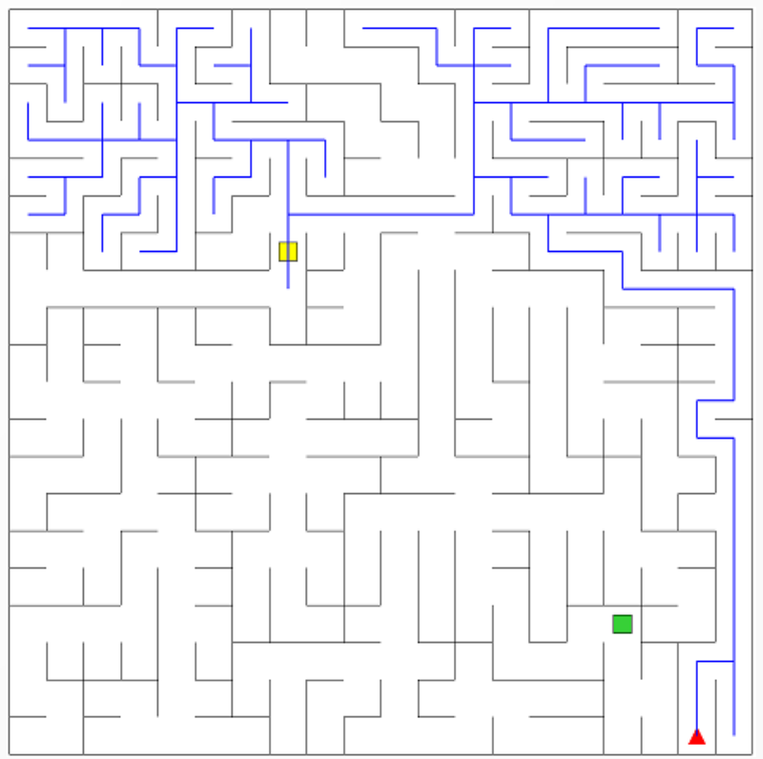


Dernière méthode : Trémaux

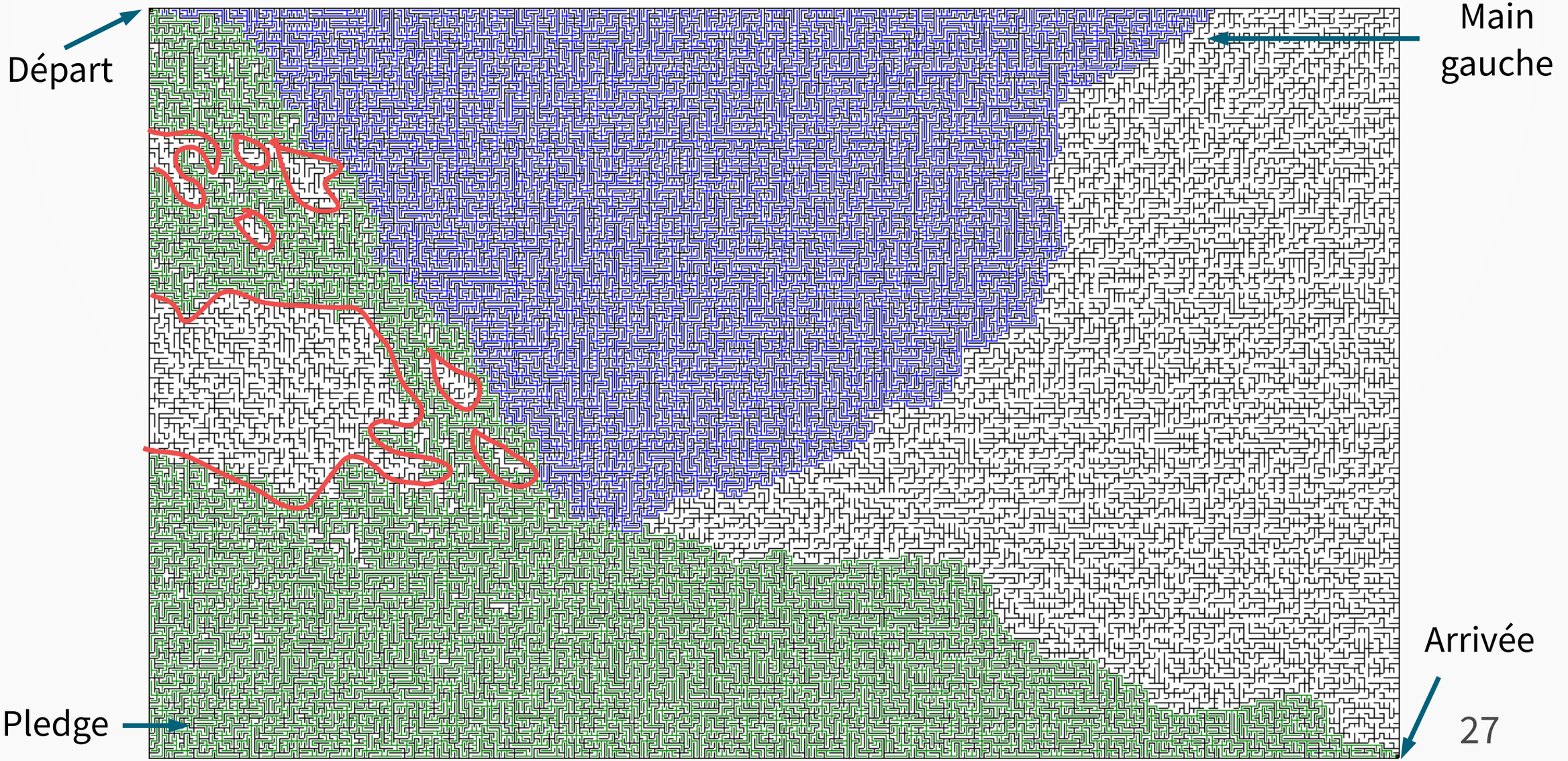
- On utilise ici des marquages au sol qui notent les passages au fur et à mesure de l'exploration.



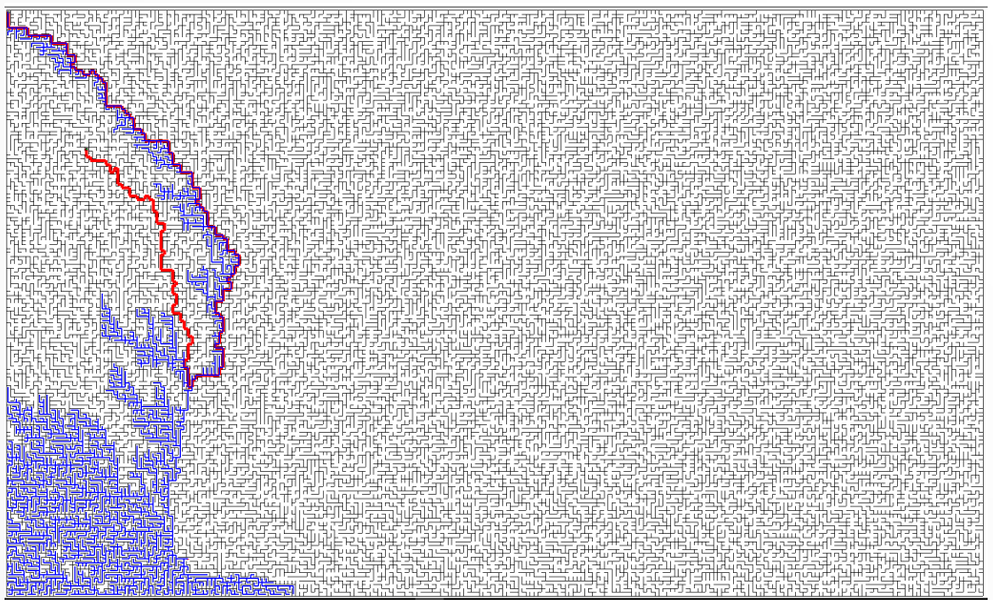
Chaque croisement est muni des directions possibles associées chacune à une marque : 0 (chemin jamais emprunté), 1 (chemin emprunté) et 2 (ne pas y aller).
On marque des deux côtés un chemin quand on l'emprunte.
Croisement non visité → choix arbitraire
Sinon, marque 1 → demi-tour et marque 2
sinon → choix du moins marqué.

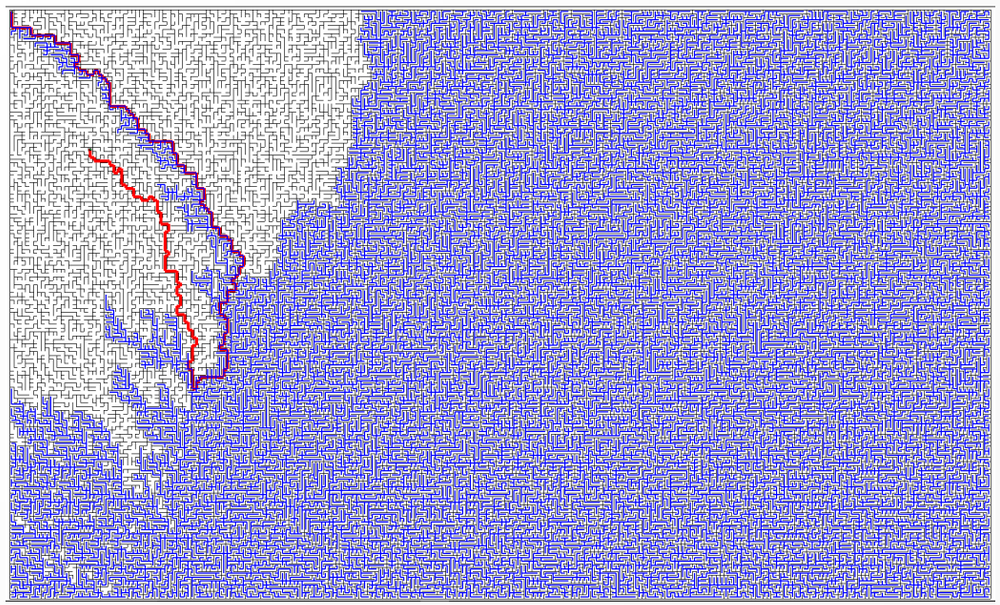


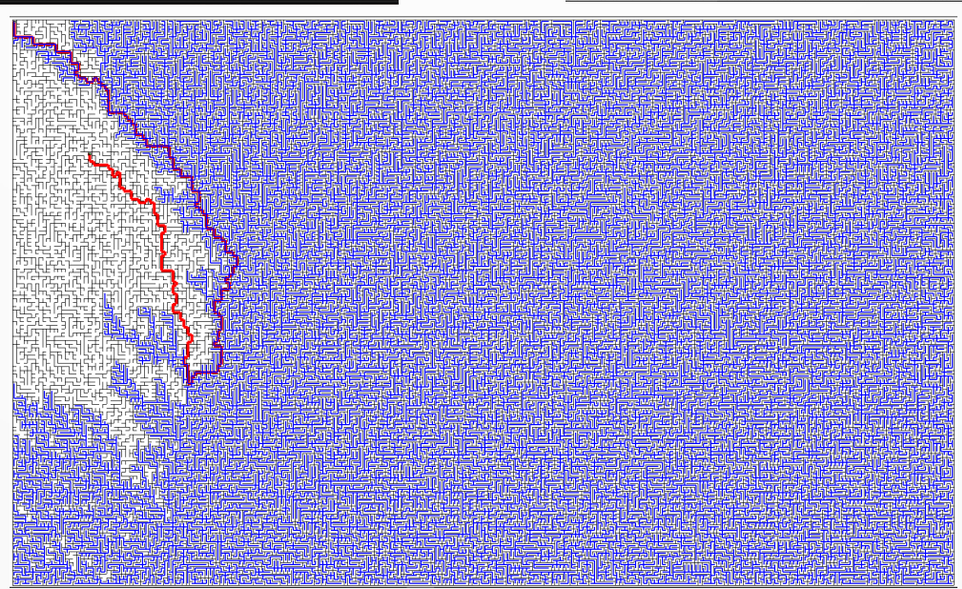
Main gauche *vs.* Pledge : avantages



Main gauche *vs.* Pledge : inconvénients

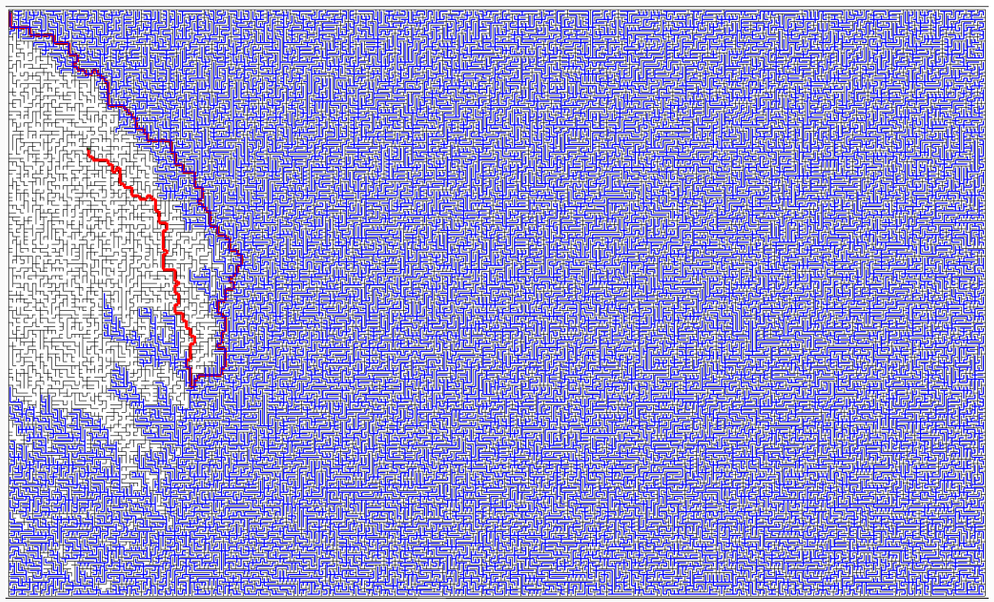


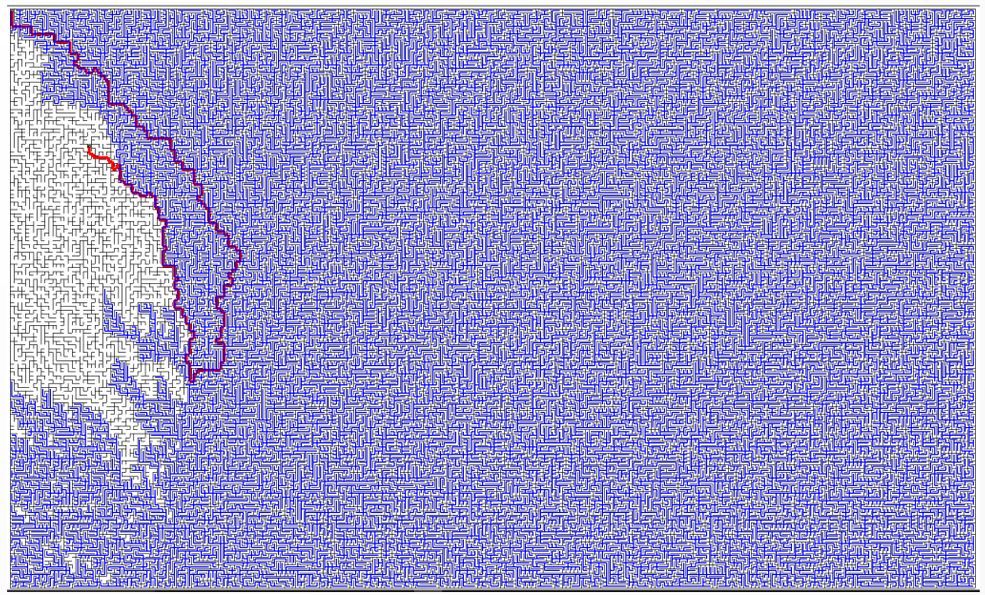


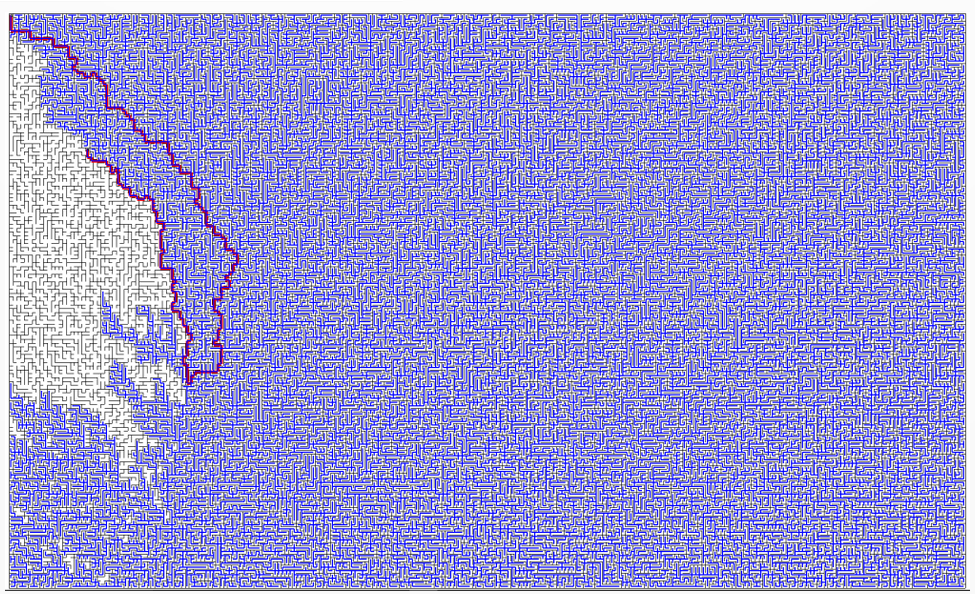


Chemin
plus court

Main gauche *vs.* Pledge : inconvénients

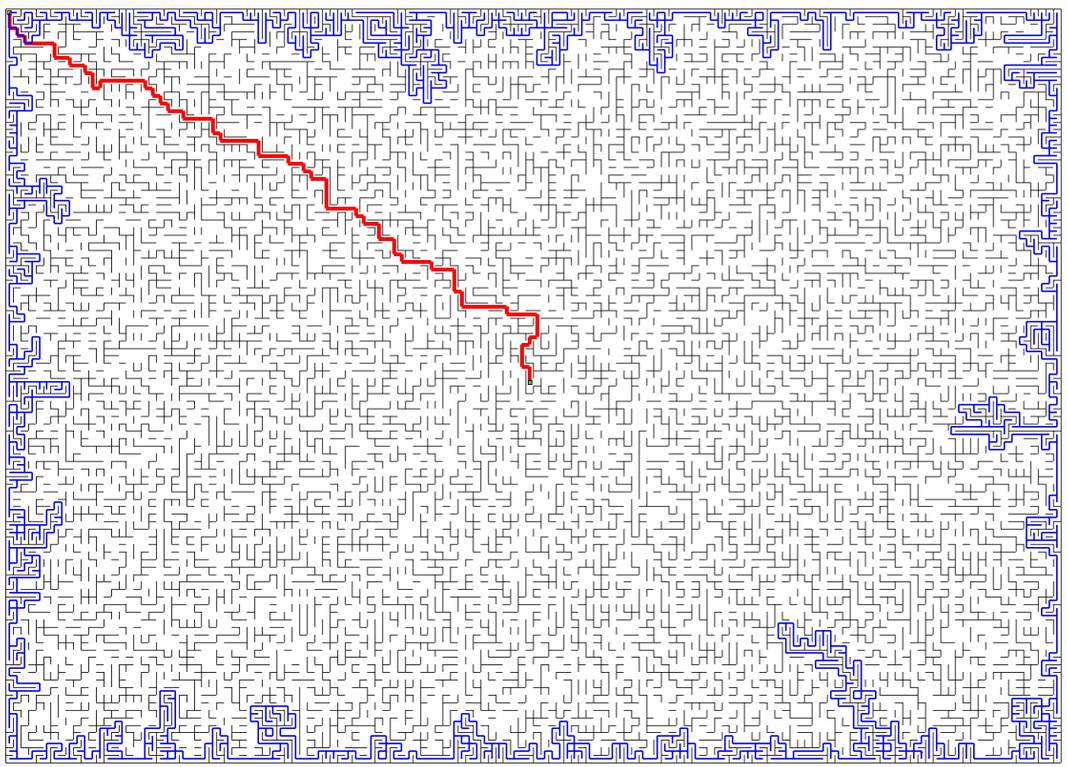


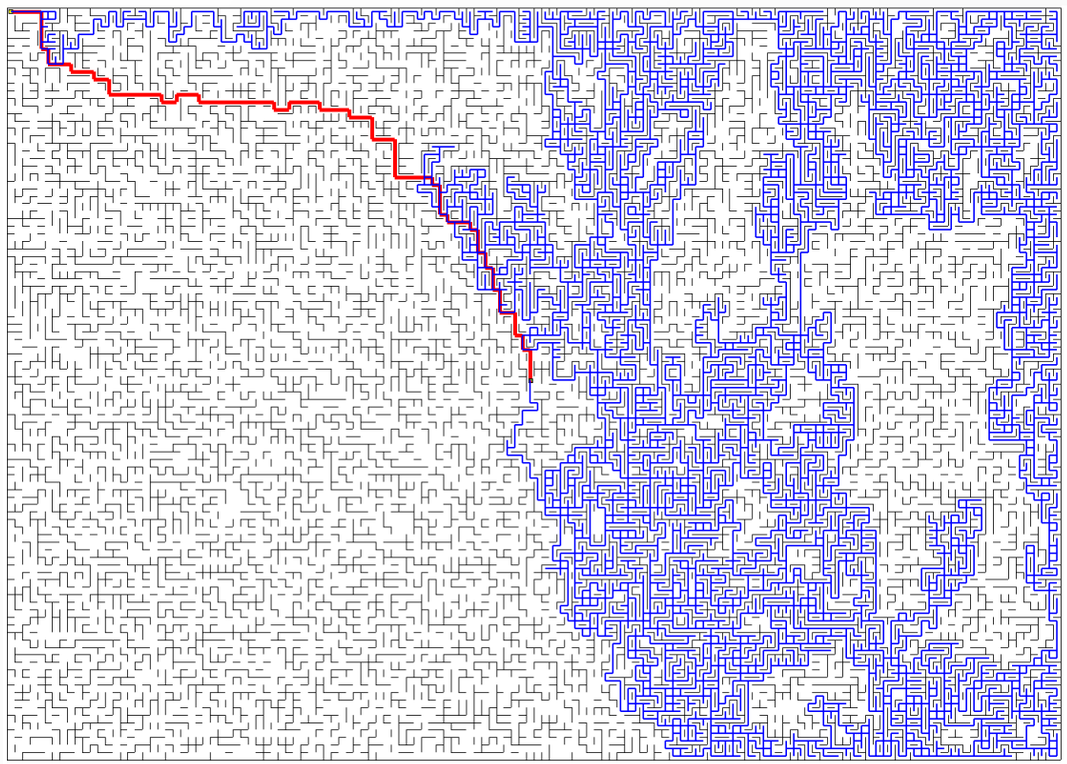




Chemin
plus court

Trémaux : la sortie assurée

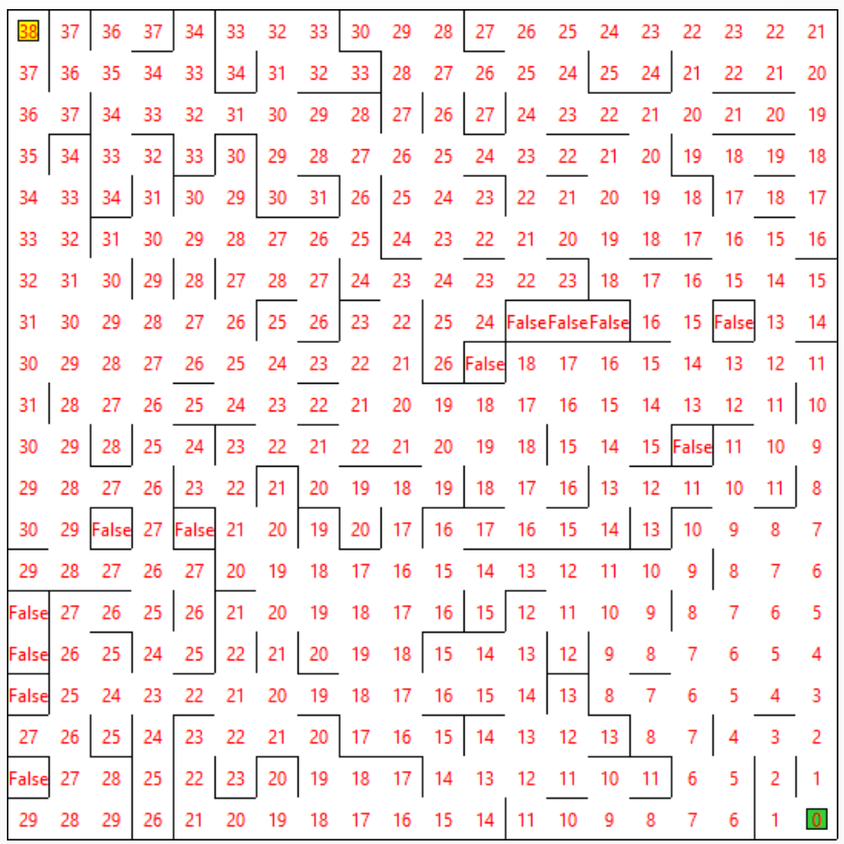




Si l'objectif se trouve au centre d'un imparfait, Pledge et main gauche risquent de tourner en rond, alors que Trémaux permet bien de l'atteindre à chaque fois.

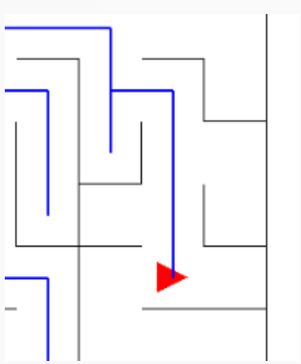
III. Simulations a. Percolation

- A l'aide de la fonction qui génère la carte des distances, on peut simplement en déduire si une case est atteignable depuis une autre :
- On peut alors réaliser de nombreux tests sur des labyrinthes aléatoires et en déduire un succès moyen pour chaque probabilité p.
- On obtient ainsi le graphe approché de θ



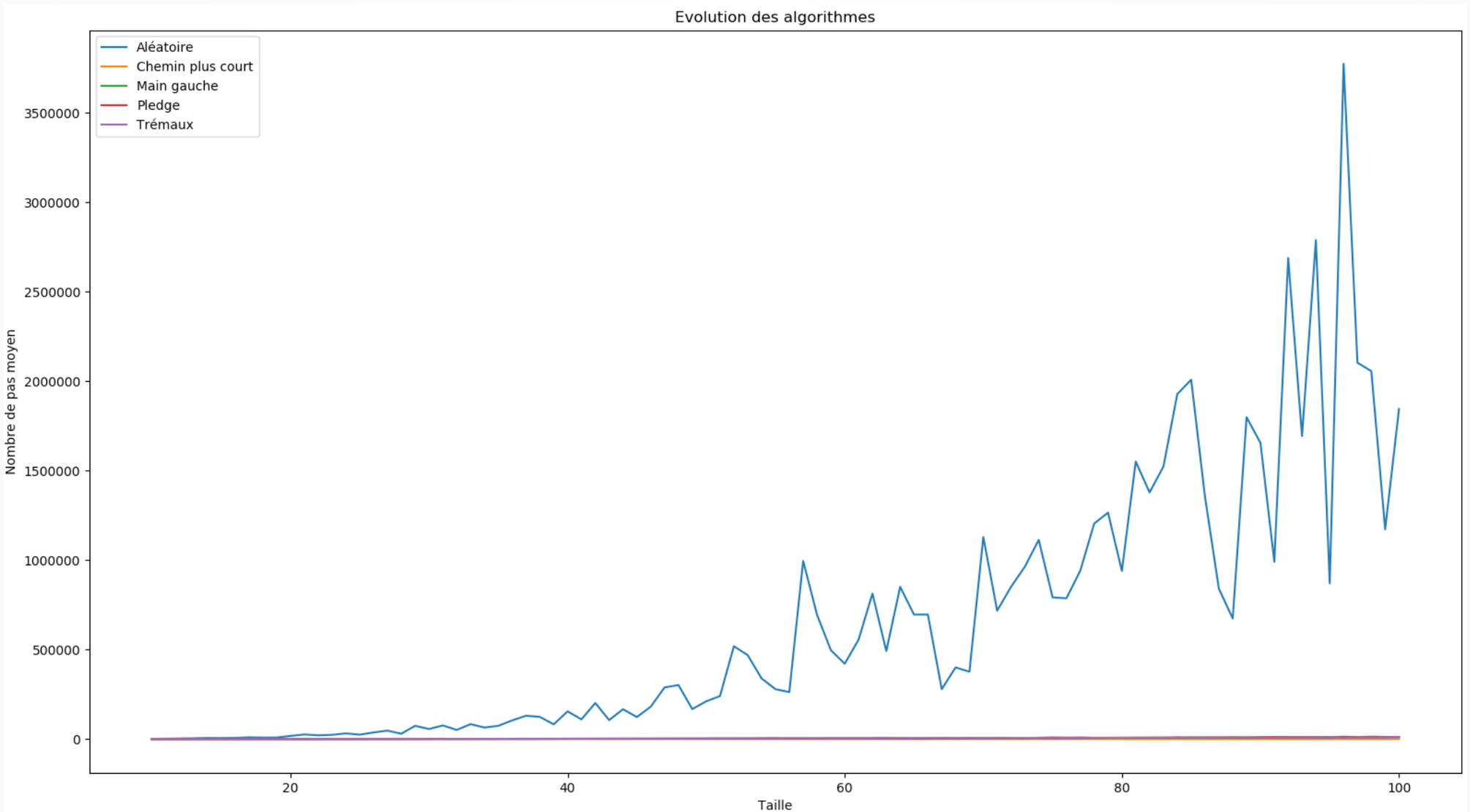
III.b. Méthodes de recherche

- On réalise la mesure du nombre de pas pour chaque méthode sur plusieurs labyrinthes parfaits aléatoires carrés avec les départ et arrivée dans les coins, pour des tailles différentes.

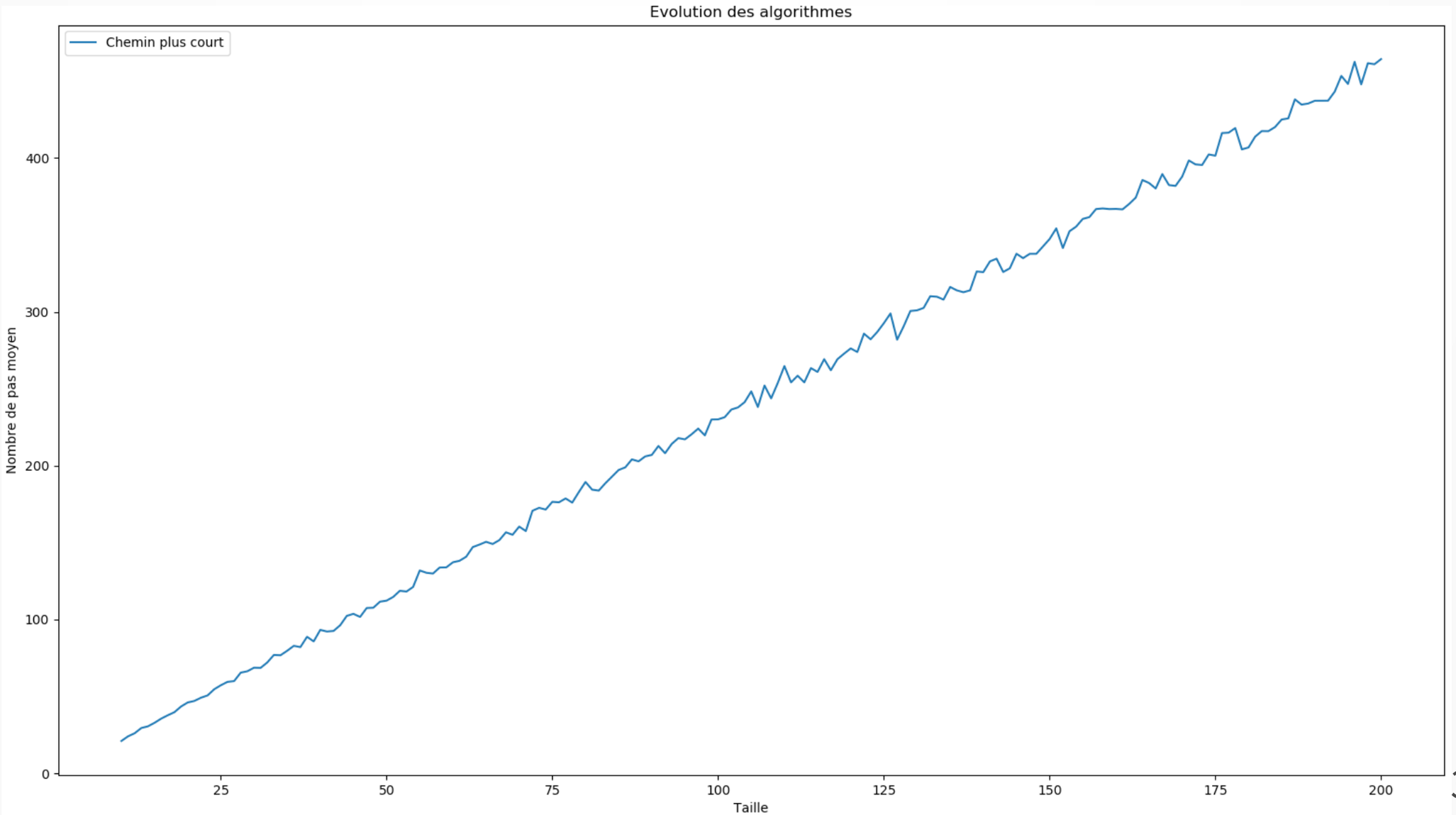


Taille	Méthode	Pas
25x25	Pledge	374
25x25	Trémaux	358
...		

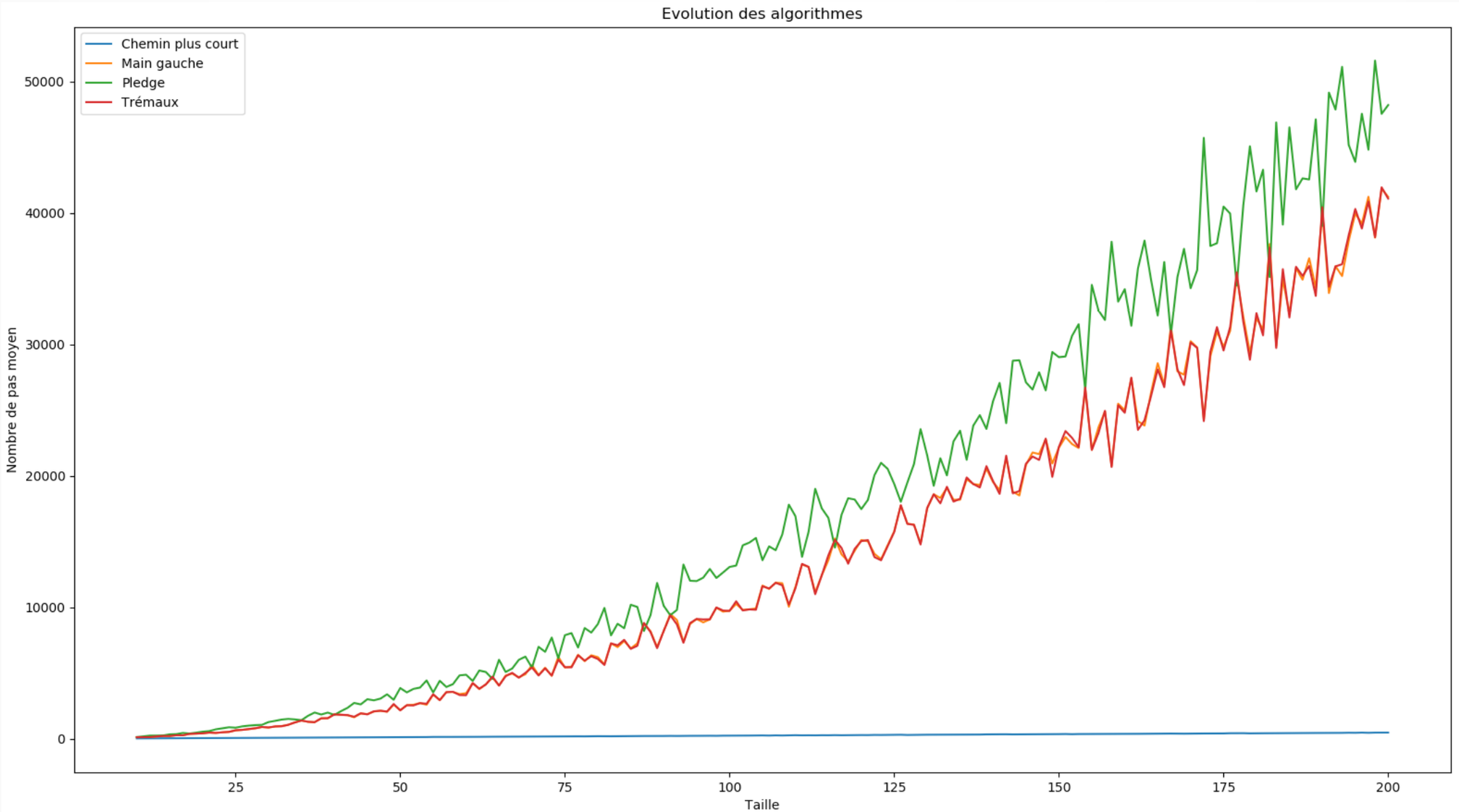
Résultats avec aléatoire : explosif



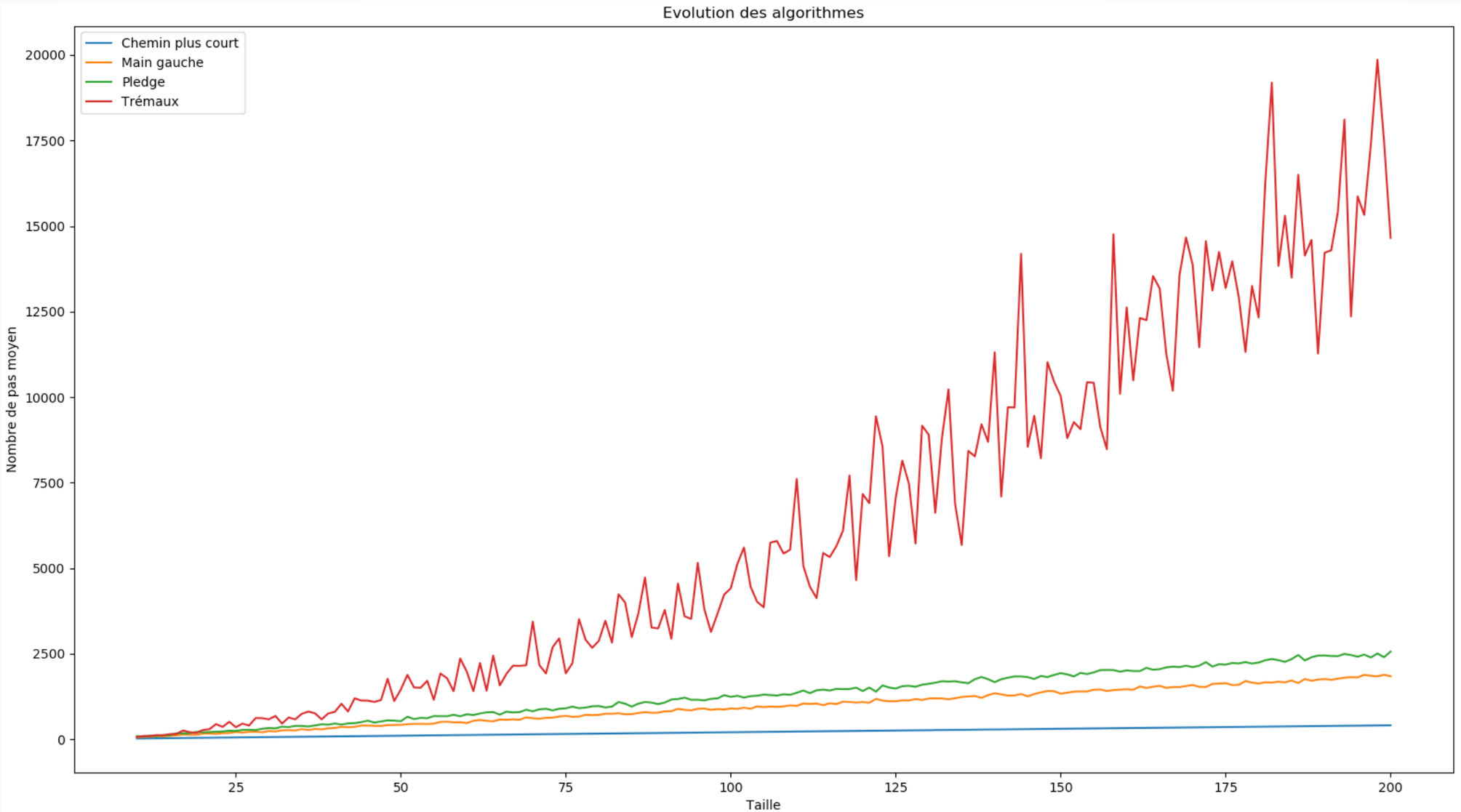
Le chemin le plus court : efficace



Sur parfaits



Sur imparfaits : l'exploration coûte



Estimation des complexités

- En réalisant une régression quadratique, on obtient pour un modèle $N = al^2 + bl + c$ (N nombre de pas moyen, l longueur) :

	a	b	c
Chemin plus court	0	2,330	-1,902
Aléatoire	350,1	-14972	166297
Main gauche	1,016	-3,607	135,9
Pledge	1,173	18,35	-455,6
Trémaux	1,021	-4,626	167,9

Sur parfaits

	a	b	c
Chemin plus court	0	2,031	0,4415
Main gauche	0,0008677	9,284	-40,67
Pledge	0,0005748	12,86	-65,23
Trémaux	0,3707	-11,98	-188,8

Sur imparfaits

Conclusion

- Ainsi, on remarque que la complexité asymptotique des méthodes proposées est au pire $O(l^2)$, où l est la taille du labyrinthe carré utilisé.
- L'algorithme de Pledge, en tant que méthode de la main gauche améliorée, sera à privilégier si l'on cherche à sortir car la sortie est au bord.
- Si l'on veut atteindre un autre objectif en revanche, l'algorithme de Trémaux assure l'exploration sans oubli.

Merci

Annexe : programmes informatiques

Labyrinthes.py :

```
import random
random.seed()
# Ce fichier donne les fonctions de bases des labyrinthes.

def creerLab(l, h): # 'Lab' = Labyrinthe
    lab = [[]] * h
    for i in range(h):
        lab[i] = [False] * l
    return lab

def creerLabAv(l, h):  # 'Av' = Avancé
    lab = [[]] * (h+1)

    lab[0] = [[True, True]]  # Ce premier paquet est pour la première ligne, spéciale.
    for a in range(1, l):
        lab[0].append([False, True])
    lab[0].append([True, False])

    for i in range(1, h):
        lab[i] = [[True, False]]  # Celui-là est pour les lignes communes, qui sont identiques
        for b in range(1, l):
            lab[i].append([False, False])
        lab[i].append([True, False])
```

```
    for c in range(0, l):  # Ce dernier est pour la dernière ligne, spéciale.
        lab[h].append([False, True])
    lab[h].append([False, False])

    return lab

def demanderLabAv(h, l):
    lab = creerLabAv(l, h)

    for i in range(h):
        for j in range(l):
            ans = input("Case ({0}, {1}), à gauche ?\n".format(i, j))

            while ans not in ["y", "n"]:
                print("Entrée non valable")
                ans = input("Case ({0}, {1}), à gauche ?\n".format(i, j))

            lab[i][j][0] = ans == "y"
            ans = input("Case ({0}, {1}), en haut ?\n".format(i, j))

            while ans not in ["y", "n"]:
                print("Entrée non valable")
                ans = input("Case ({0}, {1}), en haut ?\n".format(i, j))

            lab[i][j][1] = ans == "y"

    return lab
```

```python
def voisinsAccessiblesAv(i, j, lab):
    r = []

    if 0 <= i < len(lab) - 1 and 0 <= j < len(lab[0]) - 1:
        if not lab[i][j][1]:
            r.append([i - 1, j])
        if not lab[i][j + 1][0]:
            r.append([i, j + 1])
        if not lab[i + 1][j][1]:
            r.append([i + 1, j])
        if not lab[i][j][0]:
            r.append([i, j - 1])
    else:
        raise Exception(i, "et/ou", j, "ne sont pas des positions de cases possibles du labyrinthe avancé.")

    return r

def insererSiNouveau(x, l1):
    for y in l1:
        if x == y:
            return    # Pour arrêter immédiatement la fonction.

    l1.append(x)      # Et donc cette instruction ne s'effectuera que s'il n'y a pas l'élément x dans l1.

def insererSiNouveaux(l2, l1):
    for a in l2:
        insererSiNouveau(a, l1)

def elementAuHasard(l):
    return l.pop(random.randint(0, len(l) - 1))

def caseAuHasard(lab):
    return random.randint(0, len(lab)), random.randint(0, len(lab[0]))
```

```python
def ajouterDesMursPartout(lab):
    for i in range(0, len(lab) - 1):
        for j in range(0, len(lab[0]) - 1):
            lab[i][j] = [True, True]


def retirerUnMur(pos1, pos2, lab):
    if pos1[0] == pos2[0] - 1:
        lab[pos2[0]][pos2[1]][1] = False
    elif pos1[0] == pos2[0] + 1:
        lab[pos1[0]][pos1[1]][1] = False
    elif pos1[1] == pos2[1] - 1:
        lab[pos2[0]][pos2[1]][0] = False
    elif pos1[1] == pos2[1] + 1:
        lab[pos1[0]][pos1[1]][0] = False


def voisinsPotentiels(i, j, lab):
    if i < 0 or i >= len(lab) - 1 or j < 0 and j >= len(lab) - 1:
        raise Exception(i, "et/ou", j, "ne sont pas des positions de cases possibles du labyrinthe
avancé.")

    l = [[i-1, j], [i, j+1], [i+1, j], [i, j-1]]
    r = []
    for a in range(0, 4):
        if 0 <= l[a][0] < len(lab) - 1 and 0 <= l[a][1] < len(lab[0]) - 1:
            r.append(l[a])

    return r
```

```python
def creerLabParfait(l, h):                          # On utilise ici une méthode appélée "Growing tree algorithm".
      lab = creerLabAv(l, h)
      ajouterDesMursPartout(lab)
      casesAcces = []                                     # 'casesAcces' = cases accessibles
      a = random.randint(0, h - 1)
      b = random.randint(0, l - 1)

      ite = creerLab(l, h)                          # On crée une grille vide,
      ite[a][b] = True                              # qui aura des valeurs vraies pour dire : 'ite' = visitée
      casesAcces.extend(voisinsPotentiels(a, b, lab))

      has = []
      newPot = []
      vis = []

      while casesAcces != []:
            has.clear()
            e = random.randint(0, len(casesAcces) - 1)
            has = list(casesAcces[e])               # 'has' = case au hasard
            ite[has[0]][has[1]] = True

            newPot.clear()
            newPot = voisinsPotentiels(has[0], has[1], lab)      # 'newPot' = Nouveaux Potentiels
            vis.clear()                                 # 'vis' = visités

            for a in newPot:
                  if ite[a[0]][a[1]]:
                        vis.append(a)
                  else:
                        casesAcces.append(a)

            newCasesAcces = []
            for b in casesAcces:                    # Cette partie là est pour éliminer les cases qui auraient pu devenir visitées.
                  if not ite[b[0]][b[1]]:
                        newCasesAcces.append(b)

            casesAcces.clear()
            casesAcces = list(newCasesAcces)

            retirerUnMur(has, vis[random.randint(0, len(vis) - 1)], lab)


      return lab
```

```python
def creerLabParfaitRB(l, h):                    # Pour créer un labyrinthe parfait selon un algorithme de type "depth first".
    lab = creerLabAv(l, h)                      # Cet méthode de création est appelée "Recursive backtracker".
    ajouterDesMursPartout(lab)
    cellulesVisitees = creerLab(l, h)
    pileCases = []
    i, j = random.randint(0, h - 1), random.randint(0, l - 1)
    pileCases.append([i, j])
    cellulesVisitees[i][j] = True

    while pileCases != []:
        celluleEnCours = pileCases.pop()
        voisinsEnCours = [voisin for voisin in voisinsPotentiels(celluleEnCours[0], celluleEnCours[1], lab) if not
                                                                cellulesVisitees[voisin[0]][voisin[1]]]

        if voisinsEnCours != []:
            celluleSuivante = voisinsEnCours[random.randint(0, len(voisinsEnCours) - 1)]
            retirerUnMur(celluleEnCours, celluleSuivante, lab)
            cellulesVisitees[celluleSuivante[0]][celluleSuivante[1]] = True
            pileCases.extend([celluleEnCours, celluleSuivante])

    return lab


def creerLabPresqueParfait(l, h, r = 20):   # 'r' : On retirera les murs avec une probabilité de r%
    lab = creerLabParfait(l, h)

    for i in range(0, len(lab) - 1):
        for j in range(0, len(lab[0]) - 1):
            if lab[i][j][0] and j != 0:
                if random.randint(1, 100) <= r:
                    lab[i][j][0] = False

            if lab[i][j][1] and i != 0:
                if random.randint(1, 100) <= r:
                    lab[i][j][1] = False

    return lab
```

```python
def carteDistances(i, j, lab):
    carte = creerLab(len(lab[0]) - 1, len(lab) - 1)
    vis = list(carte)                # Une liste qui indique quelle case est déjà visitée.

    def voisins(i, j, lab):              # Une fonction qui renvoie les voisins accessibles non visités d'une cas
        l = []                           # aux positions i et j dans un labyrinthe lab.
        for b in voisinsAccessiblesAv(i, j, lab):
            if not vis[b[0]][b[1]]:
                l.append(b)

        return l

    vis[i][j] = True        # On visite la case de départ.
    dis = 0
    lCases = [[i, j]]       # Une liste de cases à visiter.
    newLCases = []          # Une liste qui va uniquement contenir les cases dont on va créer la liste de voisins.

    while lCases != []:
        dis += 1
        for a in lCases:
            for b in voisins(a[0], a[1], lab):
                vis[b[0]][b[1]] = True      # On 'visite' ici.
                carte[b[0]][b[1]] = dis
                newLCases.append(b)

        lCases.clear()
        lCases = list(newLCases)    # On remplace ici lCases par newLCases pour ne travailler que sur les cases
        newLCases.clear()           # intéressantes et donc pour que la condition de la boucle while soit correcte.

    carte[i][j] = 0                     # On applique à la case de départ une distance de 0.
    return carte
```

```python
def cheminPlusCourt(i1, j1, i2, j2, lab):
    def voisin(i, j, lab, dis):       # Une fonction qui renvoie un des voisins accessibles à la distance
= distance en cours - 1.
        for b in voisinsAccessiblesAv(i, j, lab):
            if carte[b[0]][b[1]] == dis - 1:
                return b

    carte = carteDistances(i2, j2, lab)
    chemin = [[i1, j1]]
    i = i1
    j = j1
    dis = carte[i1][j1]

    while i != i2 or j != j2:
        v = voisin(i, j, lab, dis)
        i = v[0]
        j = v[1]
        chemin.append([i, j])
        dis -= 1

    chemin.append([i2, j2])
    return chemin
```

Exploration.py :

```
from Labyrinthes import *
# Ce présent fichier décrit les algorithme de résolution de labyrinthes, créés dans le ficher Labyrinthes
# Les intelligences artificielles :

def prochainePositionIATenirGauche(i, j, orientation, lab):
     if orientation == "N":                                  # Structure générale :
          if not lab[i][j][0]:                               # On essaie de tourner à gauche
               return i, j - 1, "W", False                   # si on peut, on y va;
          elif not lab[i][j][1]:                             # sinon, on essaie d'aller tout droit
               return i - 1, j, "N", False                   # si on peut, on y va;
          else:                                                   # sinon,
               return i, j, "E", True                        # on tourne a droite.

     if orientation == "W":                          # Pareil dans les autres cas d'orientation : juste d'autres coordonnées.
          if not lab[i + 1][j][1]:
               return i + 1, j, "S", False
          elif not lab[i][j][0]:
               return i, j - 1, "W", False
          else:
               return i, j, "N", True

     if orientation == "S":
          if not lab[i][j + 1][0]:
               return i, j + 1, "E", False
          elif not lab[i + 1][j][1]:
               return i + 1, j, "S", False
          else:
               return i, j, "W", True

     if orientation == "E":
          if not lab[i][j][1]:
               return i - 1, j, "N", False
          elif not lab[i][j + 1][0]:
               return i, j + 1, "E", False
          else:
               return i, j, "S", True
```

```python
def prochainePositionIAInvariante(i, j, orientation, lab):
    if orientation == "N":
        if not lab[i][j][1]:
            return i - 1, j, "N", False
        else:
            return i, j, "W", True
    if orientation == "W":
        if not lab[i][j][0]:
            return i, j - 1, "W", False
        else:
            return i, j, "S", True
    if orientation == "S":
        if not lab[i + 1][j][1]:
            return i + 1, j, "S", False
        else:
            return i, j, "E", True
    if orientation == "E":
        if not lab[i][j + 1][0]:
            return i, j + 1, "E", False
        else:
            return i, j, "N", True
```

```python
def prochainePositionIAMouvementAleatoire(i, j, lab):
    NWSE = ("N", "W", "S", "E")
    direction = NWSE[random.randint(0, 3)]

    if direction == "N":
        if not lab[i][j][1]:
            return i - 1, j, False
        else:
            return i, j, True
    if direction == "W":
        if not lab[i][j][0]:
            return i, j - 1, False
        else:
            return i, j, True
    if direction == "S":
        if not lab[i + 1][j][1]:
            return i + 1, j, False
        else:
            return i, j, True
    if direction == "E":
        if not lab[i][j + 1][0]:
            return i, j + 1, False
        else:
            return i, j, True
```

```python
def prochainePositionIAPledge(i, j, orientation, counter, lab):
    if orientation == 'N':
        if counter == 0:                                # Si on est pas encore rentré dans l'algorithme de Pledge,
            if not lab[i][j][1]:                                # S'il n'y a pas de mur en face,
                return i - 1, j, 'N', 0, False                  # on avance.
            else:
                return i, j, 'W', -1, True                      # Sinon, on tourne à gauche.
        else:                                                   # Si on est entré = conpteur non nul,
            if not lab[i][j + 1][0]:                            # S'il n'y a pas de mur à droite,
                return i, j + 1, 'E', counter + 1, False        # on tourne et on y va.
            elif not lab[i][j][1]:                              # Sinon, s'il n'y a pas de mur en face,
                return i - 1, j, 'N', counter, False            # on avance.
            else:
                return i, j, 'W', counter - 1, False            # Sinon, on tourne à gauche.

    if orientation == 'W':                                      # Même idée.
        if counter == 0:
            if not lab[i][j][0]:
                return i, j - 1, 'W', 0, False
            else:
                return i, j, 'S', -1, True
        else:
            if not lab[i][j][1]:
                return i - 1, j, 'N', counter + 1, False
            elif not lab[i][j][0]:
                return i, j - 1, 'W', counter, False
            else:
                return i, j, 'S', counter - 1, False
```

```
if orientation == 'S':                                  # Idem.
    if counter == 0:
        if not lab[i + 1][j][1]:
            return i + 1, j, 'S', 0, False
        else:
            return i, j, 'E', -1, True
    else:
        if not lab[i][j][0]:
            return i, j - 1, 'W', counter + 1, False
        elif not lab[i + 1][j][1]:
            return i + 1, j, 'S', counter, False
        else:
            return i, j, 'E', counter - 1, False

if orientation == 'E':                                  # Idem.
    if counter == 0:
        if not lab[i][j + 1][0]:
            return i, j + 1, 'E', 0, False
        else:
            return i, j, 'N', -1, True
    else:
        if not lab[i + 1][j][1]:
            return i + 1, j, 'S', counter + 1, False
        elif not lab[i][j + 1][0]:
            return i, j + 1, 'E', counter, False
        else:
            return i, j, 'N', counter - 1, False
```

```python
def orientationOpposee(orientation):
    if orientation == "N":
        return "S"
    elif orientation == "E":
        return "W"
    elif orientation == "S":
        return "N"
    else:
        return "E"

def orientationAssociee(i, j, pos):
    if pos[0] == i - 1 and pos[1] == j:
        return "N"
    elif pos[0] == i and pos[1] == j + 1:
        return "E"
    elif pos[0] == i + 1 and pos[1] == j:
        return "S"
    else:
        return "W"

def positionAssociee(i, j, orientation):
    if orientation == "N":
        return i - 1, j
    elif orientation == "E":
        return i, j + 1
    elif orientation == "S":
        return i + 1, j
    else:
        return i, j - 1
```

```python
def grilleMarques(lab):
    marks = []

    for i in range(len(lab) - 1):
        marks.append([])

        for j in range(len(lab[0]) - 1):
            marks[i].append({})

            for voisin in voisinsAccessiblesAv(i, j, lab):
                marks[i][j][orientationAssociee(i, j, voisin)] = 0

    return marks
```

```
def prochainePositionIATremaux(i, j, orientation, marques, marquePrec, lab):
    dictVoisins = marques[i][j].keys()                # On récupère les voisins (voir grilleMarques pour la structure).

    if len(dictVoisins) == 1:                                    # Si on est pas à un croisement,
        if marquePrec <= 1:                                      # on utilise la main gauche
            return prochainePositionIATenirGauche(i, j, orientation, lab) + tuple([marquePrec + 1])
        else:
            return prochainePositionIATenirGauche(i, j, orientation, lab) + tuple([2])# et on augmente la marque en cours
    elif len(dictVoisins) == 2:
        return prochainePositionIATenirGauche(i, j, orientation, lab) + tuple([marquePrec])
    else:
        marques[i][j][orientationOpposee(orientation)] = marquePrec          # Sinon, c'est qu'on vient d'un croisement,
        marksCount = 0                                                       # qu'on est passé par des non-carrefours,

        for orVoisin in dictVoisins:                 # et qu'on vient d'arriver sur un nouveaux carrefour -> on le marque.
            if marques[i][j][orVoisin] >= 1:
                marksCount += 1                      # On compte le nombre de voisins marqué.

        if marksCount <= 1:                                          # Si seulement d'où on vient est marqué,
            for orVoisin in dictVoisins:
                if marques[i][j][orVoisin] == 0:                     # on prend un chemin jamais marqué
                    marques[i][j][orVoisin] = 1                      # et on l'emprunte en marquant ce début.
                    return positionAssociee(i, j, orVoisin) + (orVoisin, False, 1)
        else:
            if marques[i][j][orientationOpposee(orientation)] == 1:    # Sinon, si d'où l'on vient n'a qu'une seule marque,
                marques[i][j][orientationOpposee(orientation)] = 2     # on en rajoute une deuxième et on fait demi-tour.
                return positionAssociee(i, j, orientationOpposee(orientation)) + (orientationOpposee(orientation), False, 2)
            else:
                for orVoisin in dictVoisins:             # Sinon on emprunte une voie restante ayant un minimum de marques,
                    if marques[i][j][orVoisin] == 0:
                        marques[i][j][orVoisin] = 1   # en la marquant
                        return positionAssociee(i, j, orVoisin) + (orVoisin, False, 1)
                else:
                    for orVoisin in dictVoisins:
                        if marques[i][j][orVoisin] == 1:
                            marques[i][j][orVoisin] = 2
                            return positionAssociee(i, j, orVoisin) + (orVoisin, False, 2)       # adéquatement.
```

Animation.py :

```
import tkinter
from Exploration import *
# Ce présent fichier permet d'animer les intelligences artificielles crées dans le fichier Exploration

winWidth = 1500              # Réglages de la fenêtre d'affichage.
winHeight = 900

root = tkinter.Tk()          # Initialisation de la fenêtre graphique.
root.resizable(height = False, width = False)

canvas = tkinter.Canvas(root, width = winWidth, height = winHeight, bg = "white")
canvas.winfo_toplevel().title("Labyrinthes")
canvas.focus_set()
canvas.pack()

# Fonctions utilitaires :

def ajustementsAuto(lab):
    lMax = min((float(canvas["width"]) - 10) / (len(lab[0]) - 1), (float(canvas["height"]) - 10) / (len(lab) - 1))
    return (lMax,
            (float(canvas["height"]) - 10 - (len(lab) - 1) * lMax) / 2 + 6,
            (float(canvas["width"]) - 10 - (len(lab[0]) - 1) * lMax) / 2 + 6)

def tracerLabAvAuto(lab):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)

    for i in range(0, len(lab)):
        for j in range(0, len(lab[0])):
            if lab[i][j][0]:
                canvas.create_line(j * lMax + ajustJ, i * lMax + ajustI, j * lMax + ajustJ, (i + 1) * lMax + ajustI
                                   , fill = "black", width = 1)
            if lab[i][j][1]:
                canvas.create_line(j * lMax + ajustJ, i * lMax + ajustI, (j + 1) * lMax + ajustJ, i * lMax + ajustI
                                   , fill = "black", width = 1)
```

```
def tracerCheminPlusCourt(i1, j1, i2, j2, lab, couleurChemin = "red"):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)
    chemin = cheminPlusCourt(i1, j1, i2, j2, lab)

    for i in range(0, len(chemin) - 1):
        canvas.create_line(chemin[i][1] * lMax + lMax/2 + ajustJ, chemin[i][0] * lMax + lMax/2 + ajustI, chemin[i + 1][1]
            * lMax + lMax/2 + ajustJ, chemin[i + 1][0] * lMax + lMax/2 + ajustI, fill = couleurChemin, width = lMax / 10)


def tracerCarteDistances(i1, j1, lab):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)
    carte = carteDistances(i1, j1, lab)

    for i in range(len(carte)):
        for j in range(len(carte[0])):
            canvas.create_text(j * lMax + lMax/2 + ajustJ, i * lMax + lMax/2 + ajustI, text = str(carte[i][j])
                               , fill = "red")


def tracerMarques(marques, lab):
    canvas.delete("M")
    lMax, ajustI, ajustJ = ajustementsAuto(lab)

    for i in range(len(marques)):
        for j in range(len(marques[0])):
            canvas.create_text(j * lMax + lMax / 2 + ajustJ, i * lMax + lMax / 2 + ajustI, font = ("Fontana", 15)
                               , text = str(marques[i][j]), fill = "black", tag = "M")


def tracerPointsDepartArrivee(i1, j1, i2, j2, lab):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)
    canvas.create_rectangle(j1 * lMax + lMax/4 + ajustJ, i1 * lMax + lMax/4 + ajustI, j1 * lMax + 3*lMax/4 + ajustJ
                            , i1 * lMax + 3*lMax/4 + ajustI, fill = "yellow", width = 1)
    canvas.create_rectangle(j2 * lMax + lMax/4 + ajustJ, i2 * lMax + lMax/4 + ajustI, j2 * lMax + 3*lMax/4 + ajustJ
                            , i2 * lMax + 3*lMax/4 + ajustI, fill = "lime green", width = 1)
```

```
def initialiserPositionIAFleche(i1, j1, lab):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)
    return canvas.create_polygon(j1 * lMax + lMax/2 + ajustJ, i1 * lMax + lMax/4 + ajustI, j1 * lMax + lMax/4 + ajustJ
     , i1 * lMax + 3*lMax/4 + ajustI, j1 * lMax + 3*lMax/4 + ajustJ, i1 * lMax + 3*lMax/4 + ajustI, fill = "red", width = 1)


def deplacerIAFleche(IA, i, j, orientation, lab):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)

    if orientation == "N":            # Mettre la flèche dans le bon sens, pour être plus facilement compréhensible.
        canvas.coords(IA, j * lMax + lMax/2 + ajustJ, i * lMax + lMax/4 + ajustI, j * lMax + lMax/4 + ajustJ
                      , i * lMax + 3*lMax/4 + ajustI, j * lMax + 3*lMax/4 + ajustJ, i * lMax + 3*lMax/4 + ajustI)
    if orientation == "W":
        canvas.coords(IA, j * lMax + lMax/4 + ajustJ, i * lMax + lMax/2 + ajustI, j * lMax + 3*lMax/4 + ajustJ
                      , i * lMax + 3*lMax/4 + ajustI, j * lMax + 3*lMax/4 + ajustJ, i * lMax + lMax/4 + ajustI)
    if orientation == "S":
        canvas.coords(IA, j * lMax + 3*lMax/4 + ajustJ, i * lMax + lMax/4 + ajustI, j * lMax + lMax/4 + ajustJ
                      , i * lMax + lMax/4 + ajustI, j * lMax + lMax/2 + ajustJ, i * lMax + 3*lMax/4 + ajustI)
    if orientation == "E":
        canvas.coords(IA, j * lMax + lMax/4 + ajustJ, i * lMax + lMax/4 + ajustI, j * lMax + lMax/4 + ajustJ
                      , i * lMax + 3*lMax/4 + ajustI, j * lMax + 3*lMax/4 + ajustJ, i * lMax + lMax/2 + ajustI)


def ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleur = "blue"):
    lMax, ajustI, ajustJ = ajustementsAuto(lab)

    if not surPlace:
        cheminParcouru.append([i, j])
        canvas.create_line(cheminParcouru[-1][1] * lMax + lMax/2 + ajustJ, cheminParcouru[-1][0] * lMax + lMax/2 + ajustI
                           , cheminParcouru[-2][1] * lMax + lMax/2 + ajustJ, cheminParcouru[-2][0] * lMax + lMax/2 + ajustI
                           , fill = couleur, width = lMax / 20)
        del cheminParcouru[0]
```

```
def interfaceEdition(h, l, lab = []): # Pour pouvoir créer un labyrinthe avec une interface graphique -> plus rapide.
    if lab == []:
        lab = creerLabAv(l, h)

    lMax, ajustI, ajustJ = ajustementsAuto(lab)

    def switchLeftWall(event):
        for i in range(h + 1):
            for j in range(l + 1):
                if lMax * i + ajustI <= event.y < lMax * (i + 1) + ajustI and lMax * j + ajustJ
                                                    <= event.x < lMax * (j + 1) + ajustJ:
                    lab[i][j][0] = not lab[i][j][0]
                    canvas.delete("all")
                    tracerLabAvAuto(lab)
                    break

    def switchUpWall(event):
        for i in range(h + 1):
            for j in range(l + 1):
                if lMax * i + ajustI <= event.y < lMax * (i + 1) + ajustI and lMax * j + ajustJ
                                                    <= event.x < lMax * (j + 1) + ajustJ:
                    lab[i][j][1] = not lab[i][j][1]
                    canvas.delete("all")
                    tracerLabAvAuto(lab)
                    break

    root.bind("<Button-1>", switchLeftWall)
    root.bind("<Button-3>", switchUpWall)
    tracerLabAvAuto(lab)
    root.mainloop()

    return lab
```

```
def diminuerVitesseAnimation(event):
     global tempsPause
     tempsPause += 10

def terminerAnimation(event):
     global tempsPause
     tempsPause = 0

def switchPause(event):
     global enPause
     enPause = not enPause

def animerIACheminPlusCourt(i1, j1, i2, j2, lab, couleurChemin = "blue"):
     lMax, ajustI, ajustJ = ajustementsAuto(lab)
     chemin = cheminPlusCourt(i1, j1, i2, j2, lab)
     tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
     IA = canvas.create_oval(j1 * lMax + lMax/4 + ajustJ, i1 * lMax + lMax/4 + ajustI, j1 * lMax + 3*lMax/4 + ajustJ
                             , i1 * lMax + 3*lMax/4 + ajustI, fill = "red", width = 1)
     cheminParcouru = [[i1, j1]]
     index = 0

     def Rec():
          nonlocal index
          global tempsPause, enPause
          if not enPause:
               if index == len(chemin) - 1:
                    return
               else:
                    canvas.coords(IA, chemin[index][1] * lMax + lMax/4 + ajustJ, chemin[index][0] * lMax + lMax/4 + ajustI
                                  , chemin[index][1] * lMax + 3*lMax/4 + ajustJ, chemin[index][0] * lMax + 3*lMax/4 + ajustI)
                    ajouterAuChemin(chemin[index][0], chemin[index][1], False, cheminParcouru, lab, couleurChemin)
                    index += 1
                    canvas.after(tempsPause, Rec)
          else:
               canvas.after(100, Rec)

     Rec()
```

```
def animerIATenirGauche(i1, j1, i2, j2, lab, couleurChemin = "blue"):
    orientation = "N"
    tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
    IA = initialiserPositionIAFleche(i1, j1, lab)
    cheminParcouru = [[i1, j1]]
    i, j = i1, j1

    def Rec():
        nonlocal i, j, orientation
        global tempsPause, enPause

        if not enPause:
            if i == i2 and j == j2:
                return
            else:
                i, j, orientation, surPlace = prochainePositionIATenirGauche(i, j, orientation, lab)
                deplacerIAFleche(IA, i, j, orientation, lab)
                ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleurChemin)
                canvas.after(tempsPause, Rec)
        else:
            canvas.after(100, Rec)

    Rec()
```

```
def animerIAInvariante(i1, j1, i2, j2, lab, couleurChemin = "blue"):
    i, j = i1, j1
    orientation = "N"
    tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
    IA = initialiserPositionIAFleche(i1, j1, lab)
    cheminParcouru = [[i1, j1]]

    def Rec():
        nonlocal i, j, orientation
        global tempsPause, enPause

        if not enPause:
            if i == i2 and j == j2:
                return
            else:
                i, j, orientation, surPlace = prochainePositionIAInvariante(i, j, orientation, lab)
                deplacerIAFleche(IA, i, j, orientation, lab)
                ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleurChemin)
                canvas.after(tempsPause, Rec)
        else:
            canvas.after(100, Rec)

    Rec()
```

```
def animerIAMouvementAleatoire(i1, j1, i2, j2, lab, couleurChemin = "blue"):
    i, j = i1, j1
    lMax, ajustI, ajustJ = ajustementsAuto(lab)
    tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
    IA = canvas.create_oval(j1 * lMax + lMax/4 + ajustJ, i1 * lMax + lMax/4 + ajustI
                    , j1 * lMax + 3*lMax/4 + ajustJ, i1 * lMax + 3*lMax/4 + ajustI, fill = "red", width = 1)
    cheminParcouru = [[i1, j1]]

    def Rec():
        nonlocal i, j
        global tempsPause, enPause

        if not enPause:
            if i == i2 and j == j2:
                return
            else:
                i, j, surPlace = prochainePositionIAMouvementAleatoire(i, j, lab)
                canvas.coords(IA, j * lMax + lMax/4 + ajustJ, i * lMax + lMax/4 + ajustI
                              , j * lMax + 3*lMax/4 + ajustJ, i * lMax + 3*lMax/4 + ajustI)
                ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleurChemin)
                canvas.after(tempsPause, Rec)
        else:
            canvas.after(100, Rec)

    Rec()
```

```
def animerIAPledge(i1, j1, i2, j2, lab, couleurChemin = "blue"):
    i, j = i1, j1
    orientation = "N"
    counter = 0
    tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
    IA = initialiserPositionIAFleche(i1, j1, lab)
    cheminParcouru = [[i1, j1]]

    def Rec():
        nonlocal i, j, orientation, counter
        global tempsPause, enPause

        if not enPause:
            if i == i2 and j == j2:
                return
            else:
                i, j, orientation, counter, surPlace = prochainePositionIAPledge(i, j, orientation
                                                                                 , counter, lab)
                deplacerIAFleche(IA, i, j, orientation, lab)
                ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleurChemin)
                canvas.after(tempsPause, Rec)
        else:
            canvas.after(100, Rec)

    Rec()
```

```
def animerIATremaux(i1, j1, i2, j2, lab, couleurChemin = "blue"):
    orientation = "S"
    tracerPointsDepartArrivee(i1, j1, i2, j2, lab)
    IA = initialiserPositionIAFleche(i1, j1, lab)
    i, j = i1, j1
    cheminParcouru = [[i1, j1]]
    marques = grilleMarques(lab)
    marquePrec = 0

    def Rec():
        nonlocal i, j, orientation, marques, marquePrec
        global tempsPause, enPause

        if not enPause:
            if i == i2 and j == j2:
                return
            else:
                i, j, orientation, surPlace, marquePrec = prochainePositionIATremaux(i, j
                                                          , orientation, marques, marquePrec, lab)
                deplacerIAFleche(IA, i, j, orientation, lab)
                ajouterAuChemin(i, j, surPlace, cheminParcouru, lab, couleurChemin)
                canvas.after(tempsPause, Rec)
        else:
            canvas.after(100, Rec)

    Rec()
```

```
enPause = True                  # Pour toutes les IA : une manière d'interrompre ou de reprendre l'animation.
tempsPause = 201                # Pour toutes aussi  : le temps en ms entre chaque mise à jour graphique.

root.bind("<Up>", augmenterVitesseAnimation)
root.bind("<Down>", diminuerVitesseAnimation)
root.bind("<Return>", terminerAnimation)
root.bind("<space>", switchPause)
root.focus_set()

hauteur, largeur = 20, 20
iDepart, jDepart, iArrivee, jArrivee = 0, 0, hauteur - 1, largeur - 1

#print(interfaceEdition(hauteur, largeur))
labyrinthe = creerLabAvAleatoire(largeur, hauteur)
tracerLabAvAuto(labyrinthe)
tracerPointsDepartArrivee(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
#initialiserPositionIAFleche(iDepart, jDepart, labyrinthe)
#tracerCheminPlusCourt(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
tracerCarteDistances(iArrivee, jArrivee, labyrinthe)
#animerIACheminPlusCourt(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
#animerIATenirGauche(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
#animerIAPledge(iDepart, jDepart, iArrivee, jArrivee, labyrinthe, couleurChemin="green")
#animerIAInvariante(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
#animerIAMouvementAleatoire(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)
#animerIATremaux(iDepart, jDepart, iArrivee, jArrivee, labyrinthe)


root.mainloop()
```

Extraction.py :

```
import sqlite3, os, tqdm, pylab, numpy
from scipy import stats
from decimal import getcontext, Decimal
from scipy.optimize import least_squares
from Exploration import *

listeNomsIA = ["Aléatoire", "Chemin plus court", "Main gauche", "Pledge", "Trémaux"]
nbDeci = 5      # Nombre de chiffres utilisés pour l'affichage par matplotlib des résultats d'analyse.

def creerBaseDonneesMethodes(nomFichier = "Résultats.sqlite"):     # Pour créer le fichier sqlite.
    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()

    with open(os.path.dirname(os.path.realpath(__file__)) + "\\Script DB méthodes.sql"
                                , mode = 'r', encoding = "utf-8") as file:
        curseur.executescript(file.read())

    connexion.commit()
    curseur.close()
    connexion.close()
```

```python
def resoudrePar(nomIA, i1, j1, i2, j2, lab):      # Une fonction qui réalise la résolution par un certain algorithme,
    if nomIA == "Chemin plus court":          # et qui renvoie le nombre de pas qui ont été nécessaires pour atteindre l'objectif.
        return len(cheminPlusCourt(i1, j1, i2, j2, lab))
    else:
        steps = 1
        i, j = i1, j1
        orientation = "N"
        surPlace = False

        if nomIA == "Aléatoire":
            while i != i2 or j != j2:
                i, j, surPlace = prochainePositionIAMouvementAleatoire(i, j, lab)
                if not surPlace:
                    steps += 1
            return steps
        elif nomIA == "Main gauche":
            while i != i2 or j != j2:
                i, j, orientation, surPlace = prochainePositionIATenirGauche(i, j, orientation, lab)
                if not surPlace:
                    steps += 1
            return steps

        elif nomIA == "Pledge":
            counter = 0
            while i != i2 or j != j2:
                i, j, orientation, counter, surPlace = prochainePositionIAPledge(i, j, orientation, counter, lab)
                if not surPlace:
                    steps += 1
            return steps

        elif nomIA == "Trémaux":
            marquePrec = 0
            marques = grilleMarques(lab)
            while i != i2 or j != j2:
                i,j,orientation,surPlace, marquePrec = prochainePositionIATremaux(i, j, orientation, marques, marquePrec, lab)
                if not surPlace:
                    steps += 1
            return steps
```

```python
def realiserTests(N, largeur, hauteur, i1,j1,i2,j2, nomFichier="Résultats.sqlite", listeIA=listeNomsIA, labParfait=True, r=20):
    if not os.path.isfile(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier):
        creerBaseDonneesMethodes(nomFichier)                                    # On crée la DB si elle n'existe pas

    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()

    try:                                                # On essaie de récupérer le dernier identifiant généré.
        previousLastLabId = curseur.execute("SELECT labId FROM Données").fetchall()[-1][0]
    except IndexError:                                  # Si on peut, c'est que la table des données n'est pas vide,
        previousLastLabId = 0                           # sinon, on la remplit comme une première fois.

    for iD in range(1, N + 1):
        if labParfait:
            lab = creerLabParfait(largeur, hauteur)
        else:
            lab = creerLabPresqueParfait(largeur, hauteur, r)

        for IA in listeIA:
            steps = resoudrePar(IA, i1, j1, i2, j2, lab)
            curseur.execute(''' INSERT INTO Données (labId, largeurLab, hauteurLab, methodId, i1, j1, i2, j2, stepsNbr)
                                VALUES (?, ?, ?, (SELECT id FROM Algorithmes
                                WHERE nom = ?), ?, ?, ?, ?, ?)''', (iD + previousLastLabId, largeur, hauteur, IA, i1
                                                                    , j1, i2, j2, steps))

    connexion.commit()
    curseur.close()
    connexion.close()


def serieTestsCarres(tailleDepart,tailleArrivee,nbrTests,nomFichier="Résultats.sqlite",listeIA=listeNomsIA,labParfait=True,r=20):
    for taille in tqdm.trange(tailleDepart, tailleArrivee + 1, desc = "Progress :", dynamic_ncols = True):
        realiserTests(nbrTests, taille, taille, 0, 0, taille - 1, taille - 1, nomFichier, listeIA, labParfait, r)
```

```python
def analyseResultats(tailleDepart, tailleArrivee, nomFichier="Résultats.sqlite", listeIA=listeNomsIA, typeAnalyse="affichage"):
    if not os.path.isfile(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier):
        raise FileNotFoundError

    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()
    dictDonnees = {}

    for IA in listeIA:
        dictDonnees[IA] = ([], [])              # : ([taille], [nbrPasMoyen])

        for taille in range(tailleDepart, tailleArrivee + 1):
            tmp = [element[0] for element in curseur.execute("""SELECT stepsNbr FROM Données WHERE largeurLab = ?
                        AND hauteurLab = ? AND methodId = (SELECT id FROM Algorithmes WHERE nom = ?)
                        AND i1 = ? AND j1 = ? AND i2 = ? AND j2 = ?""", (taille, taille, IA, 0, 0, taille - 1, taille - 1))]

            dictDonnees[IA][0].append(taille)
            dictDonnees[IA][1].append(sum(tmp) / len(tmp))
            tmp.clear()

    pylab.close("all")
    pylab.figure("Résultats", figsize = (17, 9))          # Partie présentation grâce à matplotlib (incluant pylab)
    plotsId = []

    for IA in listeIA:                                              # On affiche chaque courbe de données.
        plotsId.append(pylab.plot(dictDonnees[IA][0], dictDonnees[IA][1], label = IA)[0])

    if typeAnalyse in ("linéaire", "lin"):                      # Si l'on souhaite réaliser des régressions linéaires,
        for IA in listeIA:
            slope, intercept, r_value, p_value, std_err = stats.linregress(dictDonnees[IA][0], dictDonnees[IA][1])
            coeffs = (c[0] + '.' + c[1][0:min(nbDeci, len(c[1]))] for c in (str(Decimal(e)).split('.') for e in
                                                                                (slope, intercept)))
            plotsId.append(pylab.plot(dictDonnees[IA][0], slope * numpy.array(dictDonnees[IA][0]) + intercept
                                      , label = "Régression {}; ({}, {})".format(IA, *coeffs))[0])

        pylab.title("Régressions linéaires")
```

```
elif typeAnalyse in ("quadratique", "quad"):            # ou "quadratiques" c'est-à-dire une parabole,
    for IA in listeIA:
        def f(x, t, y):
            return x[0] * t ** 2 + x[1] * t + x[2] - y

        x0 = numpy.array([1, 1, 1])
        t = numpy.array(dictDonnees[IA][0])
        y = numpy.array(dictDonnees[IA][1])
        res = least_squares(f, x0, args = (t, y)) # moindres carrés SciPy pour n'importe quel modèle.
        coeffs=(c[0]+'.'+c[1][0:min(nbDeci,len(c[1]))] for c in (str(Decimal(e)).split('.') for e in res.x))
        plotsId.append(pylab.plot(t,f(res.x,t,0), label="Régression {}; ({},{},{})".format(IA, *coeffs))[0])

    pylab.title("Régressions quadratiques")

elif typeAnalyse in ("affichage", "aff", ""):           # sinon par défaut seulement l'affichage,
    pylab.title("Evolution des algorithmes")            # (gestion du titre ici).
else:
    raise ValueError("'{}' n'est pas un type d'analyse".format(typeAnalyse))

pylab.xlabel("Taille")
pylab.ylabel("Nombre de pas moyen")
pylab.legend(handles = plotsId, loc = "upper left")
figManager = pylab.get_current_fig_manager()
figManager.full_screen_toggle()
pylab.show()

connexion.commit()
curseur.close()
connexion.close()
```

```python
def creerBaseDonneesPercolation(nomFichier = "Résultats.sqlite"):  # Pour créer le fichier sqlite.
    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()

    with open(os.path.dirname(os.path.realpath(__file__)) + "\\Script DB percolation.sql", mode = 'r', encoding="utf-8") as file:
        curseur.executescript(file.read())

    connexion.commit()
    curseur.close()
    connexion.close()


def testsPercolation(taille, nombreEssais, pasProbaP = 1, nomFichier = "Résultats.sqlite"):
    if not os.path.isfile(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier):
        creerBaseDonneesPercolation(nomFichier)

    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()

    for p in tqdm.tqdm(numpy.arange(0, 100 + pasProbaP, pasProbaP)):
        for test in range(nombreEssais):
            curseur.execute("INSERT INTO Données (taille, probaP, succes) VALUES (?, ?, ?)"
            , (taille,float(p),bool(carteDistances(0,0,creerLabAvAleatoire(taille,taille,r=float(p)))[taille-1][taille-1])))

    connexion.commit()
    curseur.close()
    connexion.close()
```

```python
def afficherResultatsPercolation(taille, nomFichier = "Résultats.sqlite"):
    if not os.path.isfile(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier):
        raise FileNotFoundError

    connexion = sqlite3.connect(os.path.dirname(os.path.realpath(__file__)) + "\\" + nomFichier)
    curseur = connexion.cursor()

    donnees = list(curseur.execute("SELECT succes, probaP FROM Données WHERE taille = ? ORDER BY probaP ASC", (taille,)))
    resultats = [[], []]
    tmpProba = donnees[0][1]
    tmpSum = 0
    tmpNum = 0

    for couple in donnees:
        if couple[1] > tmpProba:
            resultats[0].append(tmpSum / tmpNum)
            resultats[1].append(tmpProba / 100)
            tmpProba = couple[1]
            tmpSum = tmpNum = 0
        else:
            tmpSum += int(couple[0])
            tmpNum += 1

    resultats[1].reverse()
    pylab.close("all")
    pylab.figure("Résultats", figsize = (17, 9))
    pylab.plot(resultats[1], resultats[0])
    pylab.xlabel("Probabilité de liaison")
    pylab.ylabel("Succès moyen")
    pylab.title("Simulation de percolation sur [|-{0}, {1}|]²".format(taille, taille))
    figManager = pylab.get_current_fig_manager()
    figManager.full_screen_toggle()
    pylab.show()

    connexion.commit()
    curseur.close()
    connexion.close()
```

Script DB percolation.sql :

```sql
CREATE TABLE Données (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    taille INT,
    probaP REAL,
    succes BOOLEAN
) ;
```

Script DB méthodes.sql :

```sql
CREATE TABLE Données (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    labId INT NOT NULL,
    largeurLab INT,
    hauteurLab INT,
    methodId INT,
    i1 INT,
    j1 INT,
    i2 INT,
    j2 INT,
    stepsNbr INT
) ;

CREATE TABLE Algorithmes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nom TEXT
) ;

INSERT INTO Algorithmes (nom) VALUES ("Aléatoire") ;
INSERT INTO Algorithmes (nom) VALUES ("Chemin plus court") ;
INSERT INTO Algorithmes (nom) VALUES ("Main gauche") ;
INSERT INTO Algorithmes (nom) VALUES ("Pledge") ;
INSERT INTO Algorithmes (nom) VALUES ("Trémaux") ;
```