EFS ISN3
2ème année - Janvier 2024

Durée totale : *2h*
Documents autorisés : *Synthèse personnelle de 4 pages format A4 (2 feuilles).*

— Le barème est sur 20 points.
— Le sujet est sur 16 pages - il y a 5 exercices.

Exercice 1 : PIX (1 point)

Exercice 2 : APP (Evaluation par les pairs, 1 point)

Exercice 3 : QCM (9 points)

Attention ! Certaines questions admettent plusieurs réponses justes. Il faut toutes les donner. Chaque mauvaise réponse est sanctionnée de **-0.25 point**

3.1 Dictionnaires (2 pts)

Nous considérons un dictionnaire `person` donné ci-dessous.

```
1 person = {
2     "name": "Alice",
3     "age": 25,
4     "city": "Wonderland"
5 }
```

(3.1) Quelles instructions Python permettent d'accéder à l'âge de la personne ? (0.5 pts)

- ☐ `person('age')`
- ☐ `person.get('age', 42)`
- ☐ `person['age']`
- ☐ `person.age`
- ☐ `person[1]`

Correction :
`person['age']` et `person.get(age,42)` (0.25 chaque)

En utilisant le dictionnaire `person_info` ci-dessous, on affiche à l'écran la phrase `Charlie lives in Happyville`.

```
person_info = {
    "name": "Charlie",
    "address": {
        "street": "123 Main St",
        "city": "Happyville"
    }
}
```

(3.2) Lequel de ces codes fait cet affichage ? (0.5 pts)

- ☐ `person_info['city']`
- ☐ `print(f"{person_info['name']} lives in {person_info['city']}")`
- ☐ `person_info['address']['city']`
- ☐ `print(f"{person_info['name']} lives in {person_info['address']['city']}")`
- ☐ `print(person_info['age'])`

Correction :
`print(f"person_info['name'] lives in person_info['address']['city']")`

Nous exécutons le code suivant :

```
d = {
    "a" : 3,
    "e" : 5,
    "k" : 2,
    "m" : 6,
    "x" : 1
}
d["y"] = 1
d["k"] = 3
d["o"] = 8
```

(3.3) Quelle est alors la valeur de `len(d)` ? (0.5 pts)

☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ impossible de dire ☐ un message d'erreur

Correction :
7 car élément avec la clé "k" existe déjà, donc sa valeur va être mise à jour avec l'instruction `d["k"] = 3`

Nous exécutons le code suivant :

```
1  d = {
2      "a" : 3,
3      "e" : 5,
4      "k" : 2,
5      "m" : 6,
6      "x" : 1
7  }
8
9  s = 0
10 for i in range(len(d)):
11     s += d[i]
12
13 print(s)
```

(3.4) Qu'est-ce qui va être affiché lors de l'exécution ? (0.5pts)

☐ 0 ☐ 5 ☐ 17 ☐ impossible de dire ☐ un message d'erreur

Correction :
un message d'erreur

3.2 Propriétés de graphes (3 pts)

Soit $G = (V, E)$ un graphe non orienté.

(3.5) Quelles affirmations suivantes sont vraies ? (1pt)

- ☐ La matrice d'adjacence est toujours symétrique
- ☐ Le degré entrant d'un sommet est égal à son degré sortant
- ☐ Chaque paire de sommets est reliée par une arête dirigée
- ☐ La somme des degrés est toujours paire
- ☐ Le nombre d'arcs est toujours pair
- ☐ Le degré d'un sommet est toujours supérieur au nombre de sommets

Correction :
La matrice d'adjacence est toujours symétrique, La somme des degrés est toujours paire. 0.5 pt chacune

Soit $p =< u_0, u_1, ..., u_k >$ un chemin simple dans un graphe non orienté $G = (V, E)$.

(3.6) Quelles affirmations suivantes sont vraies ? (1 pts)

- ☐ La longueur de p est forcément inférieure à $|V|$
- ☐ $\forall i, j \leqslant k, u_i$ et u_j sont connectés
- ☐ $\forall i < k, \{u_i, u_{i+1}\} \in E$
- ☐ $\forall i, j \leqslant k, u_i \neq u_j$

Correction :
Les 3 premières sont vraies, la 4e est fausse (mais c'est trompeur par erreur). 1-0.25 par réponse fausse

On considère les séquences suivantes : a) 2 3 4 7 b) 3 3 3 3 3 3 c) 2 2 3 3 3 5 d) 2 3 3 3

(3.7) Lesquelles peuvent être les degrés des sommets d'un graphe non-orienté ? (1 pt)

☐ La séquence a) ☐ La séquence b) ☐ La séquence c) ☐ La séquence d)

Correction :
b et c. 0.5 pt chacune

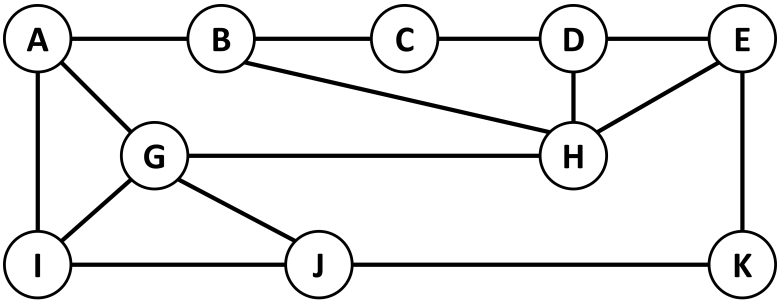
Commentaires :
Le degré est forcément plus petit que le nombre de sommets. Ca élimine "a"). Selon la formule de la somme des degrés, pour un graphe simple non-orienté la somme de degrés est un nombre pair. Ca élimine "d").

3.3 Parcours de graphe : BFS (3 pts)

Nous considérons le graphe $Graph = (V, E)$ représenté ci-dessous, avec

$V = \{A, B, C, D, E, G, H, I, J, K\}$

$E = \{(A, B), (A, G), (A, I), (B, C), (B, H), (C, D), (D, E), (D, H), (E, H), (E, K), (G, H), (G, I), (G, J), (I, J), (J, K)\}$.



Lors d'un parcours en largeur, les sommets sont visités dans un certain ordre.

(3.8) Parmi les séquences ci-dessous, indiquez celle qui peut constituer un parcours en largeur partant du sommet G. (1.5 pts)

- ☐ $GAHJIBCEKD$
- ☐ $GAHJIBCDEK$
- ☐ $GAHIJBDEKC$
- ☐ $GAHIJBCEKD$

Correction :
$GAHIJBDEKC$

Soit T l'arbre des prédécesseurs calculé lors de ce parcours en largeur.

(3.9) Parmi les affirmations suivantes, laquelle est vraie ? (0.5 pts)

- ☐ T permet de calculer le plus court chemin entre toute paire de sommets
- ☐ T permet de calculer le plus court chemin entre G et tous les autres sommets
- ☐ T permet de calculer le plus court chemin entre les feuilles de T
- ☐ T permet de calculer le plus long chemin sur le graphe

Correction :
T permet de calculer le plus court chemin entre G et tous les autres sommets

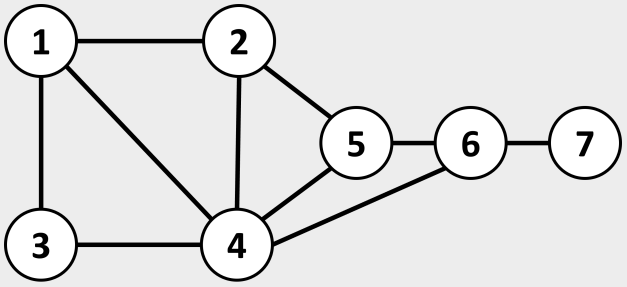
On considère le parcours en largeur (BFS) du graphe non-orienté $G = (V, E)$ à partir du sommet 1.
$V = \{1, 2, 3, 4, 5, 6, 7\}$ et
$E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{5, 6\}, \{6, 7\}\}$.

(3.10) Parmi les arêtes suivantes, lesquelles vont forcément faire partie de l'arbre prédécesseur? (1 pts)

☐ $\{1, 4\}$ ☐ $\{2, 4\}$ ☐ $\{2, 5\}$ ☐ $\{3, 4\}$ ☐ $\{4, 5\}$ ☐ $\{5, 6\}$

Correction :
$\{1, 4\}$

Commentaires :
On peut représenter le graphe G de la façon graphique suivante :



Le sommet 1 est adjacents à $2, 3, 4$. Notons que l'ordre de parcours peut varier.

Ainsi, les arêtes $(1, 2), (1, 3), (1, 4)$ vont toujours être inclus dans l'arbre prédécesseur.

Si c'est le sommet 2 qui est choisi ensuite, alors le sommet 5 va être ajouté dans la file et c'est l'arête $(2, 5)$ qui va faire partie de l'arbre prédécesseur.

Cependant, si c'est le sommet 4 qui est traité avant, c'est toujours le sommet 5 qui va être ajouté dans la file mais c'est l'arête $(4, 5)$ qui va faire partie de l'arbre.

Donc, la présence des arêtes $(2, 5)$ et $(4, 5)$ dans l'arbre prédécesseur dépend de l'ordre dans lequel les sommets 2 et 4 apparaissent dans la file.

l'arête $(5, 6)$ peut ne pas être choisie si le parcours commence d'abord par le noeud 4 et que le parcours commencer par visiter le noeud 6

3.4 Appariement (1 pts)

Soit $G = (A \cup B, E)$ avec $|A| = |B|$ et $E \subset A \times B$ un graphe biparti. On calcule un appariement avec l'algorithme de Gale-Shapley sur ce graphe en prenant en compte des préférences de A vers B et des scores de B vers A.

(3.11) Quelles affirmations sont vraies ? (0.5 pts)

- ☐ Le coût total de l'appariement est minimal
- ☐ L'appariement est un couplage parfait
- ☐ L'appariement est stable
- ☐ Les participants sont toujours appariés à leur choix préféré.

Correction :
couplage parfait, appariement stable

Durant l'algorithme de Gale-Shapley, les éléments de A sont proposés aux éléments de B.

(3.12) Que se passe-t-il si un élément de B reçoit plusieurs propositions ? (0.5 pts)

- ☐ Il rejette toutes les propositions
- ☐ Il accepte la première proposition et rejette les suivantes
- ☐ Il garde la proposition qu'il préfère
- ☐ Il accepte toutes les propositions successivement

Correction :
Il accepte la proposition de son choix préféré

Exercice 4 : Réflexions sur les arbres (3 points)

On considère un arbre non-orienté, $G = (V, E)$.

(4.1) Quel est $|E|$, le nombre d'arcs de G ? (0.5pts)

☐ $|V|^2$ ☐ $\frac{|V|*(|V|-1)}{2}$ ☐ $|V| - 1$ ☐ $|V| + 1$

Correction : $|V| - 1$

(4.2) Que vaut $\sum_{u \in V} d(u)$, la somme des degrés de G ? (0.5pts)

☐ $|V| * (|V| - 1)$ ☐ $|E|^2$ ☐ $2|E|$ ☐ $|V||E|$

Correction : $2|E|$

On suppose que dans cet arbre 2 sommets sont de degré 4, 1 sommet est de degré 3, 2 sommets sont de degré 2. Tous les autres sommets ont degré 1.

(4.3) En utilisant les réponses aux deux questions précédentes, calculez x, le nombre de sommets de degré 1. Écrivez votre raisonnement. (2 pts)

Correction :
$|V| = x + 2 + 1 + 2 = x + 5$ (0.25pt)
$\sum d(u) = x * 1 + 2 * 4 + 1 * 3 + 2 * 2 = x + 15$ (0.25pt)
$|E| = |V| - 1 = x + 4$ (0.5pt)
$\sum d(u) = 2|E| \Rightarrow x + 15 = 2x + 8$ (0.5pt)
$\Rightarrow x = 7$ (0.5pt)

Compter les points si raisonnement juste mais formules fausses aux questions précédentes et que l'élève se rend compte qu'il y a un problème.

Si formules fausses et l'élève rend un nombre non entier sans sourciller $=>$ 1pt max.

Accepter un dessin d'arbre qui marche (0.75pt) et un argument convaincant qui dit que c'est plus général (0.75pt).

Exercice 5 : Algorithmique de graphe (6 points)

Soit $G = (V, E)$ un graphe connexe non pondéré.

On définit le diamètre de G comme la longueur du plus long plus court chemin :

$$\varnothing(G) = \max_{u,v \in V} dist(u, v)$$

où $dist(u, v)$ est la longueur d'un plus court chemin de u à v.

Dans cet exercice, on va écrire un algorithme qui calcule le diamètre de G.

On considère que la fonction `BFS(G,u)` existe déjà. Elle calcule un parcours en largeur de G en partant d'un sommet u donné en paramètre et renvoie l'arbre des prédécesseurs sous forme d'un dictionnaire. Sa signature en python est la suivante :

```
def BFS(G,u):
    """
       Calcule un parcours BFS de G en partant de u et renvoie un dictionnaire représentant l'arbre
           des prédécesseurs

       Entrées:
          G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
          u : une clé du dictionnaire G représentant un sommet du graphe
       Renvoie:
          un dictionnaire dont les clés sont les sommets de G et les valeurs les prédécesseurs de
              chaque sommet dans le parcours en partant de u. La valeur de la clé u est None.
    """
```

Puisque G est connexe, tous les sommets de G seront dans l'arbre des prédécesseurs. On peut se servir de cela dans la suite.

Les questions suivantes vous demandent d'écrire des fonctions pour arriver progressivement au calcul du diamètre. **Écrivez les en python mais, si vous n'y arrivez pas, donnez un pseudo-code qui décrit l'algorithme de la fonction, vous aurez une partie des points**.

(5.1) Écrivez la fonction `distance(v,T)` qui prend en paramètre un somment v et l'arbre des prédécesseurs obtenu par un appel à `BFS(G,u)`, où u est un sommet de G, et renvoie $dist(u,v)$ = la longueur du plus court chemin de u à v obtenu en utilisant T (3 pts)

.

```
def distance(v,T):
    """
        Renvoie dist(u,v) où u est le sommet ayant servi à calculer T par un BFS
        Entrées:
            v : un sommet d'un graphe G
            T : un dictionnaire obtenu par l'appel à BFS(G,u)
        Renvoie:
            un entier valant la longueur du plus court chemin de u à v calculé en utilisant T
    """
```

Correction :

```
def distance(v,T):
    """
        Renvoie dist(u,v) où u est le sommet ayant servi à calculer T par un BFS
        Entrées:
            v : un sommet d'un graphe G
            T : un dictionnaire obtenu par l'appel à BFS(G,u)
        Renvoie:
            un entier valant la longueur du plus court chemin de u à v calculé en utilisant T
    """
    if T[v] == None :
        d = 0
    else :
        d = 1 + distance(T[v],T)
    return d
```

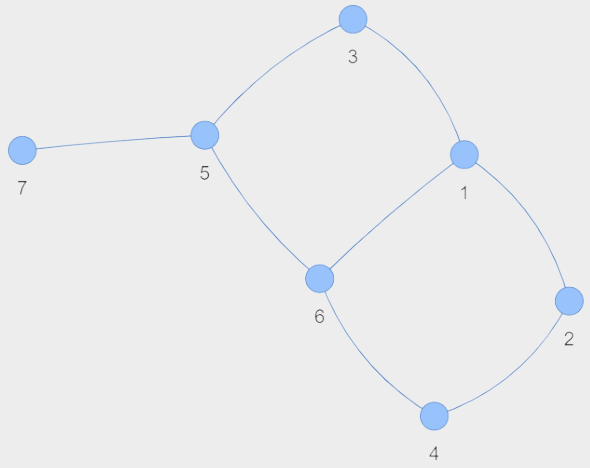
On accepte une version non récursive avec une boucle while.

```
    def distance(v,  T):
    '''Fonction qui calcule itérativement le plus court chemin vers un sommet
       d'arrivee donné à partir de la connaissance des prédécesseurs
    Args:
        dico_prec (dict) : prédécesseurs de chaque sommet
        s (str) : le sommet d'arrivée
    Return:
        (dict) : le chemin vers le sommet d'arrivée
    '''
    d = 0
    while v != None:
        d += 1
        v = T[v]
    return d - 1
```

Si malgré le conseil sur la connexité de G il y a un test sur le fait que v est dans T, on accepte.

* Si calcul du chemin (itératif ou récursif) puis calcul de la longueur de la liste => -0,5pt * algo récursif du chemin avec cas de base qui renvoie 0 et appel récursif distance(u,T[u]) + [u] -2pt (mélange de calcul de nombres + constitution de chemin * -0,5 si le cas de base de la racine est oublié (mais pas sur l'appartenance à T ou pas)

Exemple :
Prenons un exemple du graphe ci-dessous :



donné par :

```
G = {'1': ['2', '3', '6'],
     '2': ['1', '4'],
     '3': ['1', '5'],
     '4': ['2', '6'],
     '5': ['3', '6', '7'],
     '6': ['1', '4', '5'],
     '7': ['5']
    }
```

Soit `T = BFS(G, '2')`, alors : `T = {'1': '2', '2': None, '3': '1', '4': '2', '5': '3', '6': '1', '7': '5'}`

La distance entre '2' et '7' correspond donc au chemin '2' -> '1' -> '3' -> '5' -> '7'. Donc la distance est égale à 4.

Maintenant qu'on a une fonction qui donne la longueur d'un plus court chemin entre deux sommets, on peut calculer *l'excentricité* d'un sommet u, c'est-à-dire la longueur du plus grand plus court chemin partant de u : $excentricite(u) = \max_{v \in V} dist(u, v)$.

(5.2) Écrivez la fonction `excentricite(G,u)` qui prend en paramètre le graphe G et un sommet u et renvoie l'excentricité de u. Utilisez la fonction `distance` de la question précédente. (1.5 pt)

```
def excentricite(G,u):
    """
       Renvoie la longueur maximale d'un plus court chemin partant de u
       Entrées:
          G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
          u : une clé du dictionnaire G représentant un sommet du graphe
       Renvoie:
            un entier valant la longueur du plus long plus court chemin partant de u
    """
```

Correction :

```
def excentricite(G,u):
    """
        Renvoie la longueur maximale d'un plus court chemin partant de u
       Entrées:
          G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
          u : une clé du dictionnaire G représentant un sommet du graphe
```

```
        Renvoie:
             un entier valant la longueur du plus long plus court chemin partant de u
    """
    T = BFS(G,u)
    e = 0
    for v in G:
        d = distance(v,T)
        if d > e:
            e = d
    return e
```

-0.5 appels redondants à distance (pour le calcul du max) : exemple : si if distance(v,T) $>$ e : e $=$ distance(v,T)
-0.5 si T calculé dans la boucle for
-0.5 si pas d'appel du tout à BFS en debut de fonction (cas exclusif du precedent)
On accepte l'utilisation de max
Pas trop strict sur la syntaxe puisqu'on accepte le pseudo code

Vous pouvez maintenant calculer le diamètre de G :
$\varnothing(G) = \max_{u,v \in V} dist(u,v) = \max_{u \in V} excentricite(u)$.

(5.3) Écrivez la fonction `diametre(G)` qui prend en paramètre le graphe G et renvoie son diamètre. Utilisez la fonction `excentricite` de la question précédente. (1.5 pt)

```
def diametre(G):
    """
       Renvoie le diametre d'un graphe G
       Entrées:
          G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
       Renvoie:
            un entier valant le diametre de G
    """
```

Correction :

```
def diametre(G):
    """
       Renvoie le diametre d'un graphe G
       Entrées:
          G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
       Renvoie:
            un entier valant le diametre de G
    """
    d = 0
    for u in G:
        d = max(d,excentricite(G,u))
    return d
```

0.5 pour la boucle for, 0.5 pour l'appel à excentricité, 0.25 pour le max (fonction ou fait maison), 0.25 si tout le reste est bon

(5.4) BONUS : le calcul de l'excentricité tel qu'il est décomposé pourrait être optimisé pour réduire la quantité de calculs à faire. Pourquoi ? Quels sont les calculs redondants ? (1 pt)

Correction :
Pour calculer la distance d'un sommet v à u, on calcule la distance de tous les ancêtres de v. On calcule la distance d'un sommet autant de fois qu'il a de descendants.

Si l'idée y est mais que c'est brouillon, donner le point quand même. Si c'est presque bon, mais un peu faux, aller jusqu'à 0.5

(5.5) BONUS : proposez une solution pour diminuer ces calculs redondants (1 pt pour l'idée, 3 pts pour une implémentation)

Correction (version récursive) :

```
1  def distance(v,T, memoire):
2      """
3          Renvoie dist(u,v) où u est le sommet ayant servi à calculer T par un BFS. Met à jour
               memoire pour chaque calcul intermédiaire de distance
4          Entrées:
5              v : un sommet d'un graphe G
6              T : un dictionnaire obtenu par l'appel à BFS(G,u)
7              memoire : un dictionnaire mémorisant les distances déja connues.
8                  A minima memoire[u]==0
9          Renvoie:
10             un entier valant la longueur du plus court chemin de u à v calculé en utilisant T
11     """
12     if v not in memoire :
13         memoire[v] = 1 + distance(T[v],T, memoire)
14     return memoire[v]
15
16
17 def excentricite(G,u):
18     """
19         Renvoie la longueur maximale d'un plus court chemin partant de u
20        Entrées:
21           G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
22           u : une clé du dictionnaire G représentant un sommet du graphe
23        Renvoie:
24             un entier valant la longueur du plus long plus court chemin partant de u
25     """
26     T = BFS(G,u)
27     e = 0
28     memoire = {u:0}
29     for v in G:
30         d = distance(v,T,memoire)
31         if d > e:
32             e = d
33     return e
```

Correction (version itérative) :

```
1  def distance(v,T, memoire):
2      """
3          Renvoie dist(u,v) où u est le sommet ayant servi à calculer T par un BFS. Met à jour
               memoire pour chaque calcul intermédiaire de distance
4          Entrées:
5              v : un sommet d'un graphe G
6              T : un dictionnaire obtenu par l'appel à BFS(G,u)
7              memoire : un dictionnaire mémorisant les distances déja connues.
8                  A minima memoire[u]==0
9          Renvoie:
10             un entier valant la longueur du plus court chemin de u à v calculé en utilisant T
11     """
```

```
12
13      chem = []
14      cur = v
15
16      while cur not in memoire.keys():
17          chem.insert(0, cur)
18          cur = T[cur]
19
20      d = memoire[cur]
21
22      for cur in chem:
23          d += 1
24          memoire[cur] = d
25
26      return memoire[v]
27
28
29
30  def excentricite(G,u):
31      """
32          Renvoie la longueur maximale d'un plus court chemin partant de u
33         Entrées:
34            G : un dictionnaire représentant un graphe sous forme de listes d'adjacence
35            u : une clé du dictionnaire G représentant un sommet du graphe
36         Renvoie:
37              un entier valant la longueur du plus long plus court chemin partant de u
38      """
39      T = BFS(G,u)
40      e = 0
41      memoire = {u:0}
42      for v in G:
43          d = distance(v,T,memoire)
44          if d > e:
45              e = d
46      return e
```

Une autre manière de calculer efficacement l'excentricité en partant du noeud u serait de faire retourner à la fonction BFS, le dernier noeud traité. L'exeentriciuté dans ce cas n'est autre que le calcul de la distance de ce dernier au noeud u

```
1       def BFS(G, u):
2       a_traiter = [u]
3       dico_prec = {}
4       dico_prec[u] = None
5       deja_traites = []
6
7       while len(a_traiter) > 0:
8           s  = a_traiter.pop(0)
9           deja_traites.append(s)
10
11          if s in G.keys() :
12              for si in G[s]:
13                  if si not in deja_traites and si not in a_traiter:
14                      a_traiter.append(si)
15                      dico_prec[si] = s
16
17      return dico_prec, deja_traites[-1]
18
19
20  def excentricite(G,u):
21      T, last = BFS(G,u)
22      d = distance(last,T)
23      return d
```

encore une manière optimisée de trouver l'excentricité : le chemin le plus long du noeud u aux feuilles de l'arbre T. Methode : 1/trouver les feuilles 2/trouver le chemin le plus loing reliant une feuille à u

```python
def excentricite(G,u):
    T = BFS(G,u)

    feuilles = []

    for s in G:
        if s not in T.values():
            feuilles.append(s)

    d_max = 0
    for f in feuilles:
        d = distance(f, T)
        if d > d_max:
            d_max = d

    return d_max
```