Leçon d'informatique : Graphes – parcours en largeur et en profondeur

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

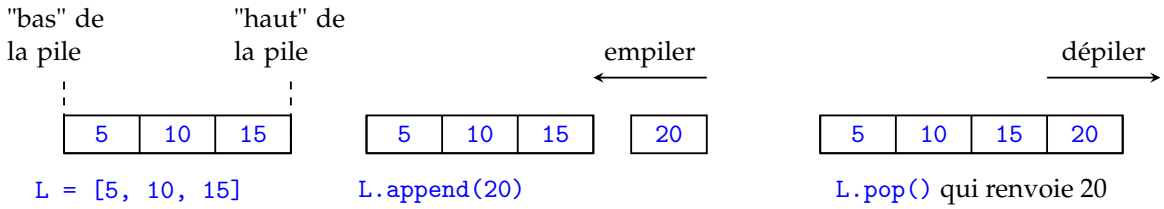
Notions	Commentaires
Parcours d'un graphe.	On introduit à cette occasion les piles et les files ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que `collections.deque`. Détection de la présence de cycles ou de la connexité d'un graphe non orienté.

I Piles, files et *deque*

I.A Pile

On a déjà rencontré des piles dans l'étude de la récursivité. On rappelle qu'une pile[1] est une structure de données basée sur le principe « dernier arrivé, premier sorti » (LIFO en anglais pour *last in, first out*). Un exemple de pile simple est une pile d'assiettes à nettoyer : la dernière assiette posée sur la pile sera la première que l'on va nettoyer.

Implémentation d'une pile par une liste :



Si on implémente une pile à l'aide d'une liste, on a deux opérations importantes :

- **empiler** qui consiste à rajouter un élément en haut de la pile. Ici, le haut de la pile correspond à l'élément le plus à droite. On pourra donc empiler avec une méthode `append` ou une concaténation.

```
def empiler1(L, a):
    # empile a au bout de la liste
    L.append(a)
L1 = [5, 10, 15]
empiler1(L1, 20)
print(L1)
```

```
def empiler2(L, a):
    # empile a au bout de la liste
    return L + [a]
L1= [5, 10, 15]
L1 = empiler2(L1 , 20)
print(L1)
```

1. *stack* en anglais

- dépiler qui consiste à enlever le dernier élément. La méthode `pop()` [a] peut faire ce travail. On doit tout de même faire attention aux cas d'une liste vide.

```
def depiler1(L):
    # depiler et renvoyer le dernier
        element
    return L.pop()
L1 = [5, 10, 15, 20]
a = depiler1(L1)
print(a) # renvoie 20
```

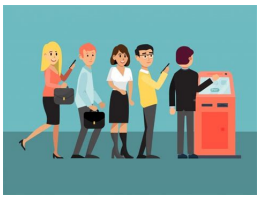
```
def depiler2(L):
    if len(L)!=0:
        return L.pop()
L2 = []
a = depiler2(L2)
print(a) # renvoie none
```

---

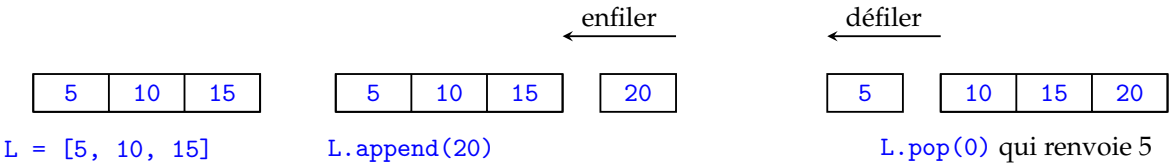
a. ou `pop(-1)` car c'est le dernier terme.

I.B File

Une file est une structure de donnée basée sur le principe : « premier arrivé, premier sorti » [2]. Une image simple est une file d'attente à la caisse d'un supermarché.



Implémentation des files par des listes :
On rappelle qu'une file est une structure de donnée basée sur le principe : « premier arrivé, premier sorti » [a].



Si on implémente une file à l'aide d'une liste, on a deux opérations importantes :

- **enfiler** qui consiste à rajouter un élément dans la file. Ici, la fin de la file correspond à l'élément le plus à droite. On pourra donc enfiler avec une méthode `append` ou une concaténation par la droite.

```
def enfiler1(L, a):
    # empile a au bout de la liste
    L.append(a)
L1 = [5, 10, 15]
enfiler1(L1, 20)
print(L1)
```

```
def enfiler2(L, a):
    # empile a au bout de la liste
    return L + [a]
L1 = [5, 10, 15]
L1 = enfiler2(L1 , 20)
print(L1)
```

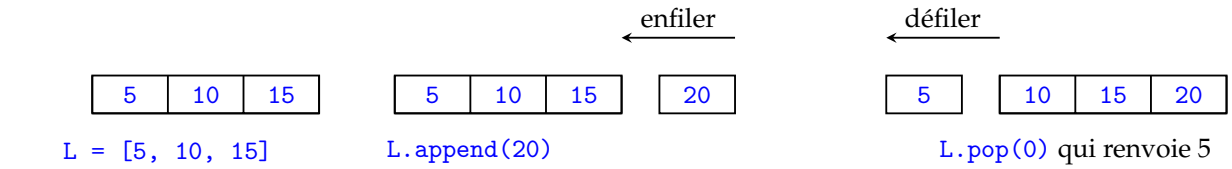
- défiler qui consiste à l'élément en début de file. On peut utiliser la méthode `pop(0)` peut faire ce travail. On doit tout de même faire attention aux cas d'une liste ou un *slicing*. vide.

---

2. *FIFO : first in, first out.*

```python
def defiler1(L):
    # depiler et renvoyer le dernier element
    return L.pop(0)
```

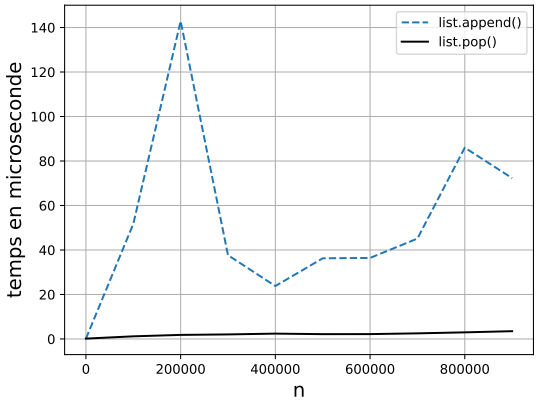
```python
L1 = [5, 10, 15, 20]
a = defiler1(L1)
print(a) # renvoie 15
```

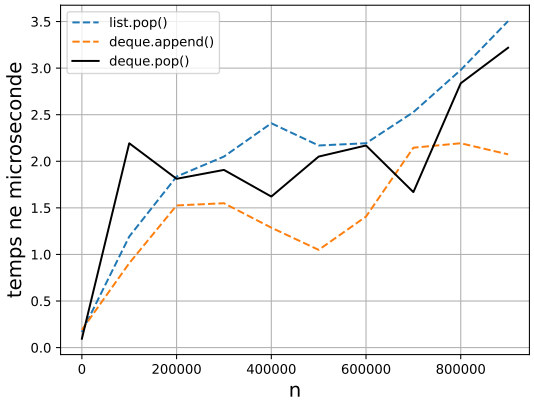


a. *FIFO : first in, first out.*

I.C *deque*

Limite de l'utilisation des listes : Pour des petites listes, l'accès à un de ces éléments se fait en coût constant, c'est-à-dire avec une complexité $O(1)$. Pour une taille n qui devient grande, le coût d'ajout d'un élément peut augmenter grandement (voir courbes ci-dessous). Il peut être utile d'utiliser, pour manipuler des files et des piles, d'utiliser un autre objet. On propose les *deque* du module `collections`.





Manipulation des `collections.deque` : Un *deque*, contraction de *double-ended-queue*, est une structure de donnée qui regroupe les propriété de la file et de la pile. On peut alors ajouter ou retirer des élements par la gauche ou par la droite. La synthaxe est donnée ci-dessous.

```python
from collections import deque

D1 = deque() # creation d'un deque vide
D2 = deque([5, 10, 15]) # deque à trois éléments
D2.append(20) # empiler à droite
D2.pop() # depiler
D2.appendleft(2) # ajout par la gauche
D2.popleft() # defiler par la gauche
```

II Parcours d'un graphe

On revient sur les graphes que l'on va parcourir, c'est-à-dire que l'on souhaite passer par chacun des sommets de ce graphe. Pour simplifier la compréhension, on prend le graphe de la figure 1. On donne pour cette exemple la liste d'adjacence.

```python
L1 = [[1,2,3], [0, 4, 5], [0], [0, 6], [1], [1, 7, 8], [3], [5], [5], []]
```

Nous allons voir deux types de parcours :

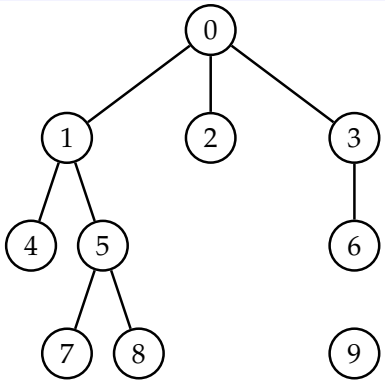


FIGURE 1

- le **parcours en profondeur** ou **DFS** (*Depth-first search*) qui utile une **pile** de visite ;
- le **parcours en largeur** ou **BFS** (*breadth-first search*) qui utile une **file** de visite.

II.A Parcours en profondeur

On souhaite parcourir un graphe depuis un sommet appelé `depart` jusqu'à un autre sommet appelé `arrivee`. On note n le nombre de sommet du graphe. On cherche l'existence d'un chemin entre `depart` et `arrivee`. On renverra `True` s'il existe un chemin entre ces deux sommets et `False` dans le cas contraire. Au fur et à mesure du parcours, on utilise deux listes :

- `dejavu` qui est une liste de taille n initialement composée du booléen `False`. C'est une liste qui va marquer les sommets déjà visités.
- `a_visiter` dans laquelle on stockera les indices des sommets à visiter. Au début, elle ne contient que `depart`.

Parcours en profondeur : Tant que la liste `a_visiter` n'est pas vide :

- on en sort l'élément le plus à droite de `a_visiter` :
  - si c'est l'arrivée, la fonction renvoie `True`. On a trouvé un chemin joignant `depart` et `arrivee`.
  - sinon, si ce sommet n'a pas encore été exploré, on le marque `True` dans la liste `dejavu` et on l'ajoute à la droite de la liste `a_visiter` tous les voisins de ce sommet.
- si la liste `a_visiter` est vide, on renvoie `False` : il n'y a pas de chemin reliant `depart` et `arrivee`.

La figure 2 permet de comprendre l'appellation **parcours en profondeur**[a].

a. *depth first search* en anglais.

Remarque : La liste `a_visiter` a une structure de pile. Dans l'exemple de la figure 1, on peut écrire ce qui se passe à la main, lorsqu'on part de 0 et qu'on souhaite arriver à 8.

Itération	sommet visité	`a_visiter`
0		`[0]`
1	`0`	`[1, 2, 3]`
2	`3`	`[1, 2, 6]`
3	`6`	`[1, 2]`
4	`2`	`[1]`
5	`1`	`[4, 5]`
6	`5`	`[4, 7, 8]`
7	`8`	`[4, 7]`

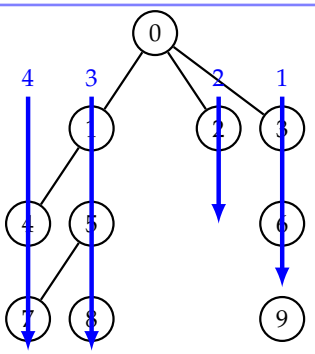


FIGURE 2 – Représentation d'un parcours en profondeur.

Exercice 1 : Écrire un algorithme itératif qui prend en argument la liste d'adjacence d'un graphe non pondéré et deux sommet `depart` et `arrivee`. Cet algorithme renverra `True` ou `False` suivant qu'il existe ou non un chemin entre ces deux sommets obtenu par un parcours en profondeur du graphe.
La réponse est donnée en figure 3.

```
1  def test_chemin_profondeur(GL, depart, arrivee):
2      n = len(GL)
3      dejavu = [False for _ in range(n)]
4      a_visiter = [depart]
5      while len(a_visiter)!=0:
6          x = a_visiter.pop()      # on depile a_visiter
7          if x == arrivee:
8              return True
9          dejavu[x] = True
10         for j in range(len(GL[x])):                # on explore les voisins de x
11             if dejavu[GL[x][j]] == False:
12                 a_visiter.append(GL[x][j])          # on empile les voisins de x
13     return False
14
15 GL1 = [[1,2,3], [0, 4, 5], [0], [0, 6], [1], [1, 7, 8], [3], [5], [5], []]
16 print(test_chemin_profondeur(GL1, 0, 9)) # renvoie False
17 print(test_chemin_profondeur(GL1, 0, 4)) # renvoie True
```

FIGURE 3

Remarque : On voit que cette algorithme donne une méthode assez simple pour visiter l'intégralité des pages d'un site internet en récupérant à chaque étape l'ensemble des liens hypertexte que contient la page visitée.

Remarque sur la connexité : On Peut remarquer que, si un graphe est connexe et non orienté, il existe toujours un chemin joignant deux sommets. Cet algorithme peut donc servir à tester la connexité d'un graphe non orienté.

Coût du parcours : Lors d'un parcours, chaque sommet entre au plus une fois dans la liste des sommets `a_visiter`, et n'en sort donc aussi qu'au plus une fois. Le coût total des manipulations de la liste `a_visiter` est un $O(n)$, avec n le nombre de sommets.
Chaque liste d'adjacence est parcourue au plus une fois donc le temps total consacré à scruter les listes de voisinage est un $O(p)$ avec p le nombre d'arêtes. Dans ce cas, le coût total d'un parcours est un $O(n+p)$.

II.B Parcours en largeur

Parcours en largeur : La liste `a_visiter` était une pile. À chaque étape, on extrayait l'élément le plus à droite (ligne 6 du programme en figure 3). On utilise à présent une file en extrayant à chaque étape l'élément le plus à gauche. Dans l'exemple de la figure 1, on peut écrire ce qui se passe à la main, lorsqu'on part de 0 et qu'on souhaite arriver à 8.

Itération	sommet visité	a_visiter
0		[0]
1	0	[1, 2, 3]
2	1	[2, 3, 4, 5]
3	2	[3, 4, 5]
4	3	[4, 5, 6]
5	4	[5, 6]
6	5	[6, 7, 8]
7	6	[7, 8]
8	7	[8]
9	8	[]

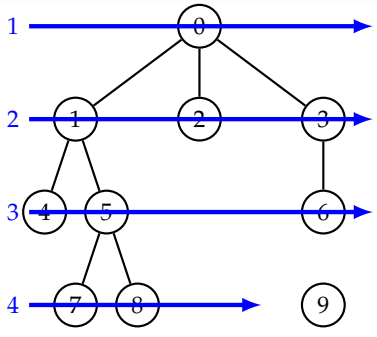


FIGURE 4 – Représentation d'un parcours en largeur.

Exercice 2 : Écrire un algorithme itératif qui prend en argument la liste d'adjacence d'un graphe non pondéré et deux sommet `depart` et `arrivee`. Cet algorithme renverra `True` ou `False` suivant qu'il existe ou non un chemin entre ces deux sommets obtenu par un parcours en largeur du graphe.
La réponse est donnée en figure 5.

```python
def test_chemin_largeur(GL, depart, arrivee):
    n = len(GL)
    dejavu = [False for _ in range(n)]
    a_visiter = [depart]
    while len(a_visiter) != 0:
        x = a_visiter.pop(0)                                  #  on defile a_visiter
        if x==arrivee:
            return True
        dejavu[x] = True
        for j in range(len(GL[x])):                  # on explore les voisins de x
            if dejavu[GL[x][j]]==False:
                a_visiter.append(GL[x][j])           # on enfile les voisins de x
    return False

GL1 = [[1,2,3], [0, 4, 5], [0], [0, 6], [1], [1, 7, 8], [3], [5], [5], []]
print(test_chemin_largeur(GL1, 0, 9)) # renvoie False
print(test_chemin_largeur(GL1, 0, 4)) # renvoie True
```

FIGURE 5

III Exercices

Pour les exemples, on utilisera les 2 graphes ci-dessous :

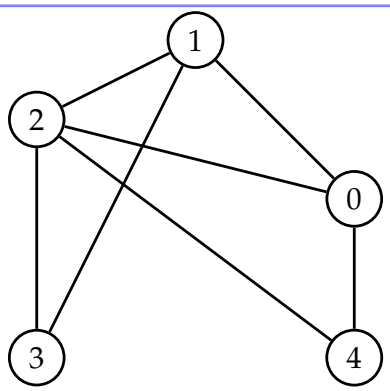


FIGURE 6 – Graphe non pondéré.

```python
L1 = [[1, 2, 4],
      [0, 2, 3],
      [0, 1, 3, 4],
      [1, 2],
      [0, 2],
      ]
```

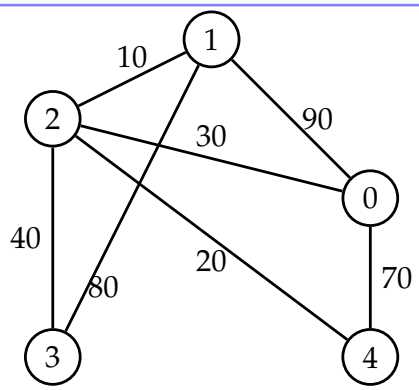


FIGURE 7 – Graphe pondéré.

```python
L2 = [[ [1,90], [2, 30], [4, 70]],
      [[0,90], [2, 10], [3, 80]],
      [[0, 30], [1, 10], [3, 40], [4, 20]],
      [[1, 80], [2, 40]],
      [[0, 70], [2, 20]],
      ]
```

Exercice 3 : Est-ce que les algorithmes proposés ci-dessus fonctionnent pour un graphe orienté ? Si non, quelles modifications sont à faire ?

Exercice 4 : On considère à présent un graphe pondéré représenté par une liste d'adjacacence. Comment faut-il modifier les algorithmes précédents ?

Exercice 5 : On prend le graphe de la figure 6. On fait un parcours depuis le sommet `0` jusqu'au sommet `3`. Remplir le tableau suivant pour un parcours en profondeur puis en largeur :

Parcours en profondeur

Itération	sommet visité	`a_visiter`
0	-	`[0]`
1		
2		
3		
4		
5		
6		

Parcours en largeur

Itération	sommet visité	`a_visiter`
0	-	`[0]`
1		
2		
3		
4		
5		
6		

Réponse : voir figure 8 en page 7.

Exercice 6 : Écrire une fonction `itineraire` qui, pour un graphe non pondéré, renvoie l'itinéraire entre un `depart` et une `arrivee` (sous la forme d'une liste) en suivant un parcours en profondeur ? S'il n'y a pas de chemin entre les deux sommets, on renverra la liste vide. Par exemple, la fonction doit renvoyer `[0, 4, 2, 3]` pour le chemin allant de `0` à `3`.
Réponse : voir figure 9 en page 8.

Exercice 7 : On prend maintenant un graphe pondéré. Écrire une fonction `itineraire2` qui renvoie la distance parcourue en suivant un parcours en profondeur en un `depart` et une `arrivee`.
Par exemple, la fonction doit renvoyer `130` pour le chemin allant de `0` à `3`.
Réponse : voir figure 10 en page 8.

IV Réponses aux exercices

Parcours en profondeur

Itération	sommet visité	`a_visiter`
0		`[0]`
1	0	`[1, 2, 4]`
2	4	`[1, 2, 2]`
3	2	`[1, 2, 1, 3]`
4	3	`[1, 2, 1]`
5		
6		

Parcours en largeur

Itération	sommet visité	`a_visiter`
0		`[0]`
1	0	`[1, 2, 4]`
2	1	`[2, 4, 2, 3]`
3	2	`[4, 2, 3, 3, 4]`
4	4	`[2, 3, 3, 4]`
5	2	`[3, 3, 4, 3]`
6	3	`[3, 4, 3]`

FIGURE 8

```python
def itineraire(GL, depart, arrivee):
    n = len(GL)
    dejavu = [False for _ in range(n)]
    a_visiter = [depart]
    itineraire = []
    while len(a_visiter)!=0:
        x = a_visiter.pop()
        itineraire.append(x)
        if x == arrivee:
            return itineraire
        dejavu[x] = True
        for j in range(len(GL[x])):
            if dejavu[GL[x][j]] == False:
                a_visiter.append(GL[x][j])
    return itineraire

print(itineraire(L1, 0, 3))
```

FIGURE 9

```python
def itineraire2(GL, depart, arrivee):
    n = len(GL)
    dejavu = [False for _ in range(n)]
    a_visiter = [depart]
    itineraire = []
    while len(a_visiter)!=0:
        x = a_visiter.pop()
        itineraire.append(x)
        if x == arrivee:
            return itineraire
        dejavu[x] = True
        for j in range(len(GL[x])):
            if dejavu[GL[x][j][0]] == False:                    #ATTENTION au [0]
                a_visiter.append(GL[x][j][0])                   #ATTENTION au [0]
    return itineraire
def distance(GL, depart, arrivee):
    d = 0
    L = itineraire2(GL, depart, arrivee)
    for i in range(len(L)-1):
        d+= GL[i][i+1][1]                          #ATTENTION au [1] pour distance
    return d

print(distance(L2, 0, 3))
```

FIGURE 10

V Application à la percolation

Définition : La percolation désigne le passage d'un fluide à travers un milieu poreux. On peut penser à l'écoulement de l'eau à travers les différentes strates pour alimenter les nappes phréatiques.

Application au percolateur des machines à café : Pour la préparation d'un expresso, certaines machines ont un percolateur où l'eau se fraye un chemin entre les particules de café (voir figure 11).

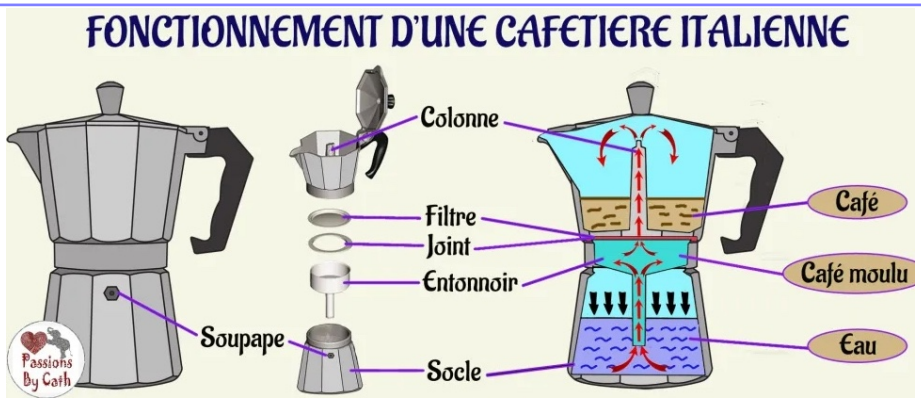


FIGURE 11 – Fonctionnement d'un percolateur de machine à café.

Modélisation par un graphe : On peut modéliser la situation en deux dimensions comme sur la figure 12. Les zones grisées sont les grains de café et les zones blanches sont les interstices dans lesquels l'eau peut passer. Cela revient au graphe de la figure 13. Si l'écoulement de l'eau vient du eau (équivalent du sommet 1), elle va s'écouler selon un parcours de graphe (figure 14 à 16).

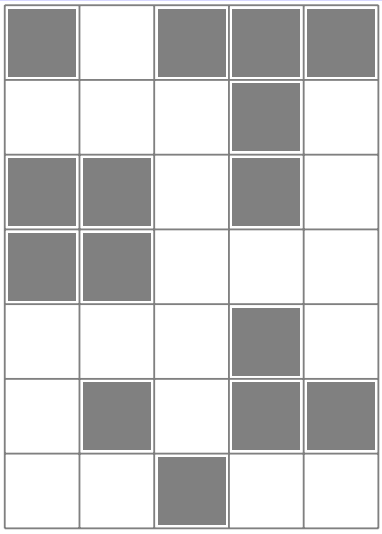


FIGURE 12

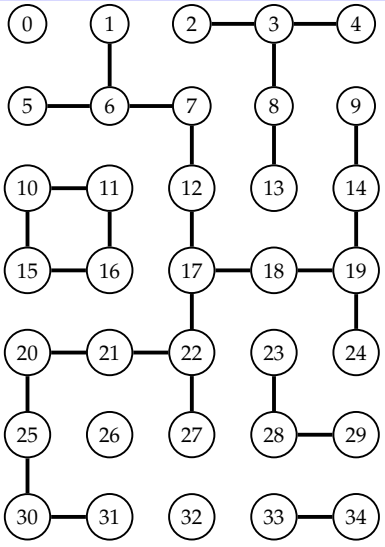


FIGURE 13

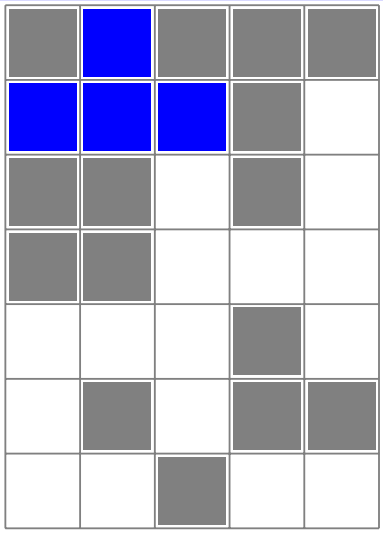


FIGURE 14

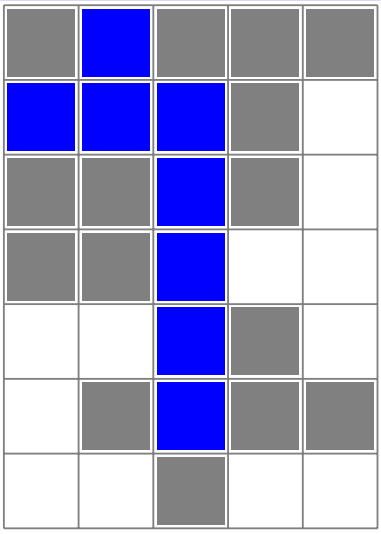


FIGURE 15

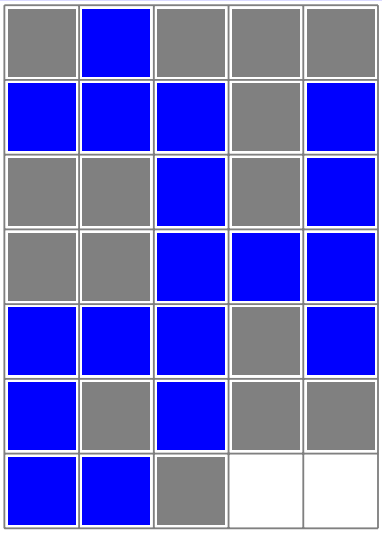


FIGURE 16