ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 7 mars 2011

Résumé

- Algorithmes
  - Traitements asynchrone
  - Simulations
  - Fouille en largeur

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Résumé

Nous avons considéré deux implémentations des files :
LinkedQueue et **CircularArrayQueue**.

Pour l'implémentation à l'aide d'un tableau, l'arithmétique modulo est utilisée afin que les index recommencent au début lorsqu'ils atteignent la fin du tableau.

```
index = ( index + 1 ) % MAX_QUEUE_SIZE
```

L'une des difficultés de cette implémentation c'est distinguer la file vide de la file pleine. Il faut éviter d'écraser les éléments avant lorsque la file est pleine.

Il y a plusieurs solutions dont l'utilisation de valeurs sentinelles, d'un booléen ou encore compter les éléments. Bien entendu, l'implémentation des méthodes dépend du choix.

L'opération **dequeue()** s'appelle parfois **serve()** parce que les files sont souvent utilisées pour les applications client/serveur.

Traitement asynchrone

Les applications de type **producteur/consommateur**, **client/serveur** ou **sender/receiver** nécessitent l'utilisation de files si le traitement des données est asynchrone.

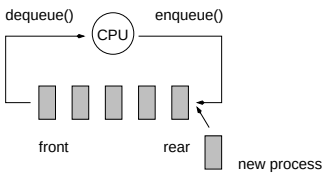
Un traitement asynchrone signifie que le client et le serveur ne sont pas synchronisés, le serveur n'est pas prêt ou capable de recevoir les données au temps et à la vitesse de l'envoi.

Ce traitement nécessite une file :
– Le client insère des données dans la file (enqueue) ;
– Le serveur retire des données de la file (dequeue) au moment opportun.
Une telle file est parfois appelée un **tampon** (*buffer*).

En particulier, les communications entre processus (*inter-process communication*) dans un système d'exploitation fonctionnent de la sorte.

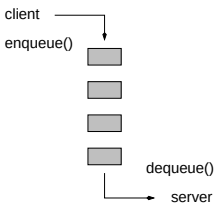
1. gestionnaire d'imprimante (*printer spooler*) ;
2. *buffered i/o* ;
3. l'accès aux disques ;
4. la transmission de messages (paquets) sur un réseau.

Temps partagé



Tous les systèmes d'exploitation modernes sont à temps partagé. L'une des techniques communes pour le partage du temps s'appelle *round-robin*. Le premier processus en file (dequeue) se voit attribuer une tranche de temps après laquelle son exécution est suspendue et le processus est mis à la fin de la file (enqueue), et on passe au processus suivant.

Communications entre processus



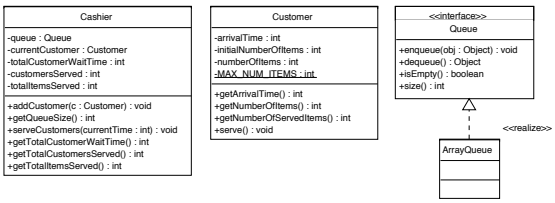

```
while ( true ) {
    while ( ! q.isFull() ) {
        q.enqueue( ... );
    }
}
```

```
while ( true ) {
    while  (! q.empty() ) {
        process( q.dequeue() )
    }
}
```

⇒ *inter-process communication* (IPC), *buffered i/o*, etc.

Applications

– Simulations ;



– Générations de séquences (fouille en largeur, labyrinthe).

Algorithmes à base de file

Algorithme :

1. Insère “”
2. Pour toujours
   (a) s ← retirer
   (b) Insère “s + 0”
   (c) Insère “s + 1”

Que fait l'algorithme ci-haut ?

Il génère **toutes** les séquences possibles de 0s et de 1s, **en ordre croissant de longueur** : 0, 1, 00, 01, 10, 11, 000, 001, . . .

Autrement dit l'ensemble des chaînes, S, telles que :

$$S \equiv [s \leftarrow 0, 1, s' + 0, s' + 1; s' \in S]$$

```
new queue           []
enqueue(0)          [0]
enqueue(1)          [0,1]
s = 0 = dequeue()   [1]
enqueue(s+0 = 0+0)  [1,00]
enqueue(s+1 = 0+1)  [1,00,01]
s = 1 = dequeue()   [00,01]
enqueue(s+0 = 1+0)  [00,01,10]
enqueue(s+1 = 1+1)  [00,01,10,11]
s = 00 = dequeue()  [01,10,11]
enqueue(s+0 = 00+0) [01,10,11,000]
enqueue(s+1 = 00+1) [01,10,11,000,001]
s = 01 = dequeue()  [10,11,000,001]
enqueue(s+0 = 01+0) [10,11,000,001,010]
enqueue(s+1 = 01+1) [10,11,000,001,010,011]
s = 10 = dequeue()  [11,000,001,010,011]
enqueue(s+0 = 10+0) [11,000,001,010,011,100]
enqueue(s+1 = 10+1) [11,000,001,010,011,100,101]
```

Génération de séquences

L'algorithme peut facilement être généralisé afin de générer toutes les séquences possibles, en ordre croissant de longueur, pour un alphabet quelconque de taille finie.

En particulier, considérons un exemple à partir d'un alphabet de taille 4, $\Sigma = L, R, U, D$.

1. Insère “”
2. Pour toujours
   (a) s ← retirer
   (b) Insère “s + L”
   (c) Insère “s + R”
   (d) Insère “s + U”
   (e) Insère “s + D”

```
nouvelle file       []
enqueue("")         [""]
s = "" = dequeue()  []
enqueue(s+L = L)    [L]
enqueue(s+R = R)    [L,R]
enqueue(s+U = U)    [L,R,U]
enqueue(s+D = D)    [L,R,U,D]
s = L = dequeue()   [R,U,D]
insere(s+L = L+L)   [R,U,D,LL]
insere(s+R = L+R)   [R,U,D,LL,LR]
insere(s+U = L+U)   [R,U,D,LL,LR,LU]
insere(s+D = L+D)   [R,U,D,LL,LR,LU,LD]
s = R = dequeue()   [U,D,LL,LR,LU,LD]
insere(s+L = R+L)   [U,D,LL,LR,LU,LD,RL]
insere(s+R = R+R)   [U,D,LL,LR,LU,LD,RL,RR]
insere(s+U = R+U)   [U,D,LL,LR,LU,LD,RL,RR,RU]
insere(s+D = R+D)   [U,D,LL,LR,LU,LD,RL,RR,RU,RD]
s = U = dequeue()   [D,LL,LR,LU,LD,RL,RR,RU,RD]
insere(s+L = U+L)   [D,LL,LR,LU,LD,RL,RR,RU,RD,UL]
insere(s+R = U+R)   [D,LL,LR,LU,LD,RL,RR,RU,RD,UL,UR]
```

Donnons un sens à ces chaînes

Que sont donc ces Ls, Rs, Us et Ds ?

Nous disons maintenant que chaque symbole de l'alphabet correspond à une direction :

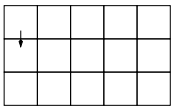
L = *left* ;

R = *right* ;

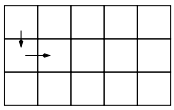
U = *up* ;

D = *down*.

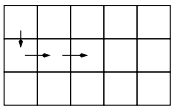
Chaque chaîne correspond à un **chemin** (*path*) dans un espace à 2 dimensions.



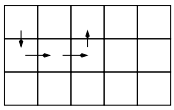
D



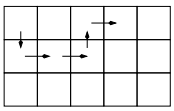
DR



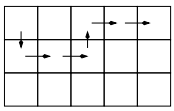
DRR



DRRU



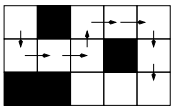
DRRUR



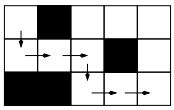
DRRURR

Ajoutons des obstacles

Supposons maintenant que certaines cases soient inaccessibles.



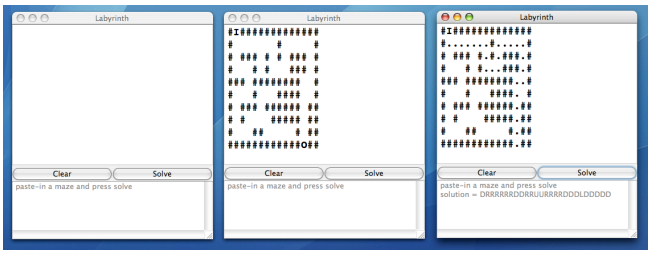
DRRURRDD



DRRDRR

⇒ Quelles modifications sont nécessaires afin que notre algorithme pour la génération de chaînes puisse être utilisée afin de trouver la sortie dans un labyrinthe ?

Labyrinthe



Méthodes auxiliaires

Nous aurons besoin d'une méthode qui vérifie si une solution partielle est valide, **checkPath(String path)**.

Et aussi une méthode qui nous dit si une solution valide atteint son but, **reachesGoal(String path)**.

Structures de données

Une matrice de caractères, i.e. un tableau à 2 dimensions.

```
char[][] maze;
```

Une position inaccessible (un mur) est notée par '#', une case vide par ' ', et une position visitée par '+'.

```
#+######
#+# # #
#++ # #
###   #
##### #
```

checkpath(String path)

```
private boolean checkPath( String path ) {

    boolean[][] visited = new boolean [ MAX_ROW ][ MAX_COL ];

    int row, col;

    row = 0; // entrée en (0,0)
    col = 0;

    int pos=0;

    boolean valid = true;
```

checkpath(String path)

```
    ...
    while ( valid && pos < path.length() ) {
        char direction = path.charAt( pos++ );
        switch ( direction ) {
        case LEFT:
            col--;
            break;
        case RIGHT:
            col++;
            break;
        case UP:
            row--;
            break;
        case DOWN:
            row++;
            break;
        default:
            valid = false;
        }
        ...
```

checkpath(String path)

```
            // après chaque déplacement, nous vérifions que la position courante
            // est valide, i.e. à l'intérieur du labyrinthe, n'est pas un mur,
            // et n'a pas été visité.

            if ( (row >= 0) && (row < MAX_ROW) && (col >= 0) && (col < MAX_COL) )
                if ( visited[ row ][ col ] || grid[ row ][ col ] == WALL )
                    valid = false;
                else
                    visited[ row ][ col ] = true;
            else
                valid = false;

        } // end of while loop

        return valid;
    }
```

(Are we done yet !)

```
private boolean reachesGoal( String path ) {
    int row = 0;
    int col = 0;
    for ( int pos=0; pos < path.length(); pos++ ) {
        char direction = path.charAt( pos );
        switch ( direction ) {
        case LEFT:  col--; break;
        case RIGHT: col++; break;
        case UP:    row--; break;
        case DOWN:  row++; break;
        }
    }
    return grid[ row ][ col ] == OUT;
}
```

Labyrinthe

Nous utiliserons un algorithme à base de file afin de trouver un chemin dans un labyrinthe.

Cet algorithme trouvera toujours le plus court chemin ! Si un tel chemin existe.

Cet algorithme ressemble beaucoup à celui que nous avons utilisé afin de générer des chaînes de caractères, de longueur croissante, pour un alphabet de taille fini.

```
q.enqueue( "" )
while ( true )
  s <- q.dequeue()
  for each char in the alphabet
      q.enqueue( s + char )
```

⇒ La différence principale sera que les éléments seront filtrés avant d'être remis en file — i.e. la file ne contiendra que des chemins valides.

Remarques

L'algorithme présenté s'appelle une fouille en largeur ("breadth-first-search").

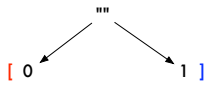
Pourrait-on utiliser une pile ? Discussion.

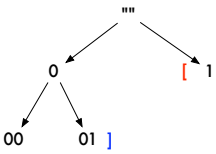
L'algorithme utilisant une pile s'appelle fouille en profondeur ("depth-first-search").

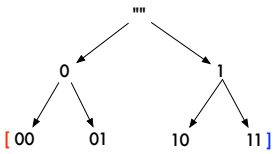
Pourquoi appelle-t-on ces algorithmes "fouille en largeur" et "fouille en profondeur" ?

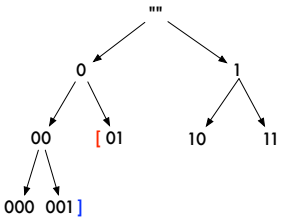
Il y a plusieurs déclinaisons de cet algorithme, l'une d'elle, la recherche en faisceau, utilise une structure de donnée de taille fixe.

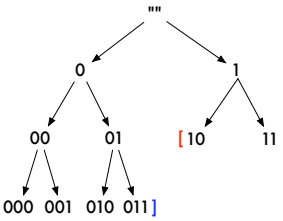

[""]

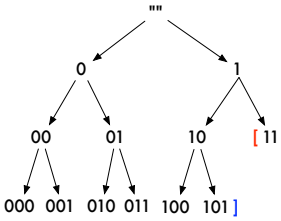

""
[0
1]

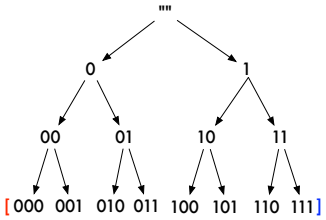

""
0
[1
00
01]

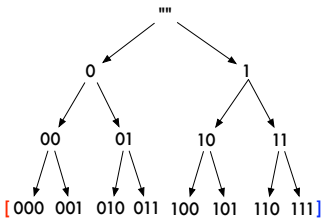

""
0
1
[00
01
10
11]


""
0
1
00
[01
10
11
000
001]


""
0
1
00
01
[10
11
000
001
010
011]



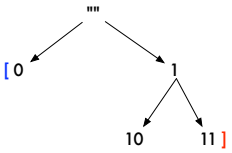


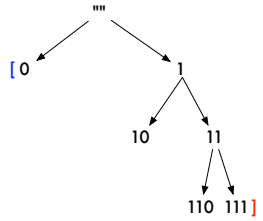


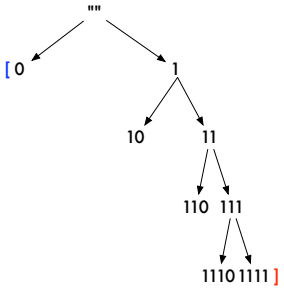
[""]

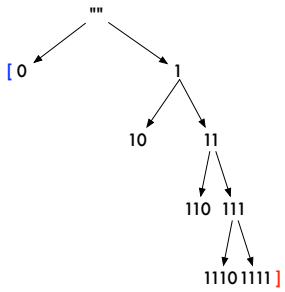
L'algorithme utilisant une **file** s'appelle aussi ≪**fouille en largeur**≫ (**breadth-first search**).

L'**arbre de recherche** est construit niveau par niveau. Toutes les séquences d'un même niveau (donc toutes les séquences de même longueur) sont traitées avant de procéder avec le niveau suivant.







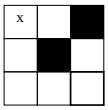


L'algorithme utilisant une **pile** s'appelle aussi une ≪**fouille en profondeur**≫ (**depth-first search**).

L'**abre de recherche** est construit branche par branche. Une séquence est sélectionnée et étendue à répétition jusqu'à ce qu'aucune extension ne soit valide. L'algorithme revient alors en arrière, d'où le surnom d'algorithme de retour-arrière (**backtracking**).

```
#I##############
#    ####      #
## # #    ### #
# ##   ####   #
#    # ## # ###
## ### ## #  ##
#  ###     ####
## ########## ##
#            ##
#############O##
```

```
#I##############
#++++####.....#
## #+#....###.#
#.##+..####...#
#.+++#.##.#.###
##+###.##.#..##
#.+###.....####
##+##########.##
#.++++++++++##
#############O##
```


x