Chapitre 13

Démineur

13.1. Nouveaux thèmes abordés dans ce chapitre

- Récursivité (première approche)
- Frames multiples
- Boutons radio

13.2. Exploration d'un labyrinthe

Pour introduire la notion de récursivité, nous allons, avant de nous occuper du jeu du démineur, voir comment explorer un labyrinthe.

Voici ci-dessous un labyrinthe (les « X » sont des murs), avec une case d'entrée notée « E » et une sortie notée « S ».

```
XXXXX
X X S
X   X
XEX X
X X X
XXXXX
```

Le programme ci-dessous lit un fichier contenant le plan d'un labyrinthe. Il parcourt toutes les cases vides à partir de la position courante en les marquant avec un « . », afin d'éviter de tourner en rond. Il essaie successivement les directions *droite*, *gauche*, *bas* puis *haut*. C'est ce que fait la procédure récursive `thesee` (l'adjectif récursif signifie qu'elle s'appelle elle-même).

labyrinthe.py

http://ow.ly/QBK1n

```
# Exploration récursive d'un labyrinthe

def lire_labyrinthe():
    # met en mémoire la labyrinthe
    # et renvoie les coordonnées du point de départ
    global case, haut, larg
    fichier = open("dedale.txt", "r")
    lignes = fichier.readlines()
    fichier.close()
    haut = len(lignes)
    larg = len(lignes[0])-1
    print(haut,"lignes",larg,"colonnes")
    case = [[0 for col in range(larg)] for row in range(haut)]
```

```
    for i in range(haut):
        for j in range(larg):
            case[i][j]=lignes[i][j]
            if lignes[i][j]=='E':
                i0=i
                j0=j
                case[i0][j0]="E"
    return i0,j0

def imprimer_labyrinthe():
    global case, haut, larg
    for i in range(haut):
        for j in range(larg):
            print(case[i][j],end="")
        print()
    print()

def thesee(i,j):
    global case
    if case[i][j] == "S":
        imprimer_labyrinthe()
    elif case[i][j] == " " or case[i][j] == "E":
            case[i][j] = "."
            thesee(i,j+1)
            thesee(i,j-1)
            thesee(i+1,j)
            thesee(i-1,j)
            case[i][j] = " "

# programme principal

i0, j0 = lire_labyrinthe()
imprimer_labyrinthe()
print("Solution(s)")
thesee(i0, j0)
```

La récursivité permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

La récursivité utilise toujours la pile du programme en cours. On appelle pile une zone mémoire réservée à chaque programme. Son rôle est de stocker les variables locales et les paramètres d'une procédure. Supposons que nous sommes dans une procédure `proc1` dans laquelle nous avons des variables locales. Faisons ensuite appel à une procédure `proc2` ; comme le microprocesseur va commencer à exécuter `proc2` mais qu'ensuite il reviendra continuer l'exécution de `proc1`, il faut bien stocker quelque part les variables de la procédure en cours `proc1` ; c'est le rôle de la pile. Tout ceci est géré de façon transparente pour l'utilisateur. Dans une procédure récursive, toutes les variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si on ne fait pas attention on peut arriver à un *débordement de pile* (*stack overflow* en anglais). Ensuite, les variables sont désempilées.

Dans notre exemple du labyrinthe, les variables seront les coordonnées des cases du labyrinthe. Chaque fois qu'une case sera marquée, elle sera mise dans la pile. Quand la marque sera effacée, la case sera désempilée. C'est l'idée-clé pour bien comprendre la récursivité.

Pour voir en détails ce que fait le programme, mettons des coordonnées aux cases et affichons toutes les cases parcourues, dans l'ordre :

```
 01234
0XXXXX
1X X S      S se trouve aux coordonnées (1 ; 4)
2X   X
3XEX X      E se trouve aux coordonnées (3 ; 1)
4X X X
5XXXXX
```

```
3 1         case de départ ; on marque la case avec un « . »
3 2         on essaie à droite : c'est un mur
3 0         on essaie à gauche : c'est un mur
```

4 1	on se déplace en bas ; on marque la case (4 ; 1)
4 2	on essaie à droite : c'est un mur
4 0	on essaie à gauche : c'est un mur
5 1	on essaie en bas : c'est un mur
3 1	on a tout essayé pour la case (4; 1) ; on remonte et on efface la marque de (4 ; 1), car on arrive sur une case marquée
2 1	la case (3 ; 1) est marquée, donc on reprend où on en était : on monte et on marque la case (2 ; 1)
2 2	on essaie à droite : ça passe ; on marque la case (2 ; 2)
2 3	on essaie à droite : ça passe ; on marque la case (2 ; 3)
2 4	on essaie à droite : c'est un mur
2 2	on essaie à gauche : la case est marquée
3 3	on se déplace en bas ; on marque la case (3 ; 3)
3 4	on essaie on essaie à droite : c'est un mur
3 2	on essaie on essaie à gauche : c'est un mur
4 3	on se déplace en bas ; on marque la case (4 ; 3)
4 4	on essaie à droite : c'est un mur
4 2	on essaie à gauche : c'est un mur
5 3	on essaie en bas : c'est un mur
3 3	on a tout essayé pour la case (4; 3) ; on efface la marque et on retourne à la case du sommet de la pile : (3 ; 3)
2 3	la case (3 ; 3) est marquée, donc on reprend où on en était : on monte, et on efface la marque de (3 ; 3), car on arrive sur une case marquée
1 3	la case (2 ; 3) est marquée, donc on reprend où on en était : on monte. On marque la case (1 ; 3)
1 4 !	on essaie à droite : c'est la sortie !

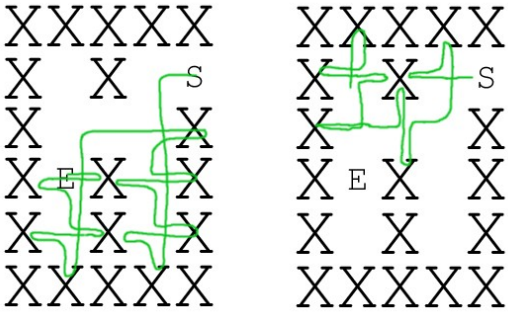
Le programme a trouvé un chemin. Mais cela n'est pas fini... On n'a pas exploré entièrement le labyrinthe. Il y a peut-être d'autres chemins...

```
XXXXX
X X.S
X...X
X.X X
X X X
XXXXX
```

1 2	on essaie à gauche : c'est un mur
2 3	on essaie en bas : la case est marquée
0 3	on essaie en haut : c'est un mur
2 1	on a tout essayé pour la case (1; 3) ; on efface la marque et on retourne à la case du haut de la pile : (2 ; 3). Mais cette case est marquée, donc on reprend où on en était. Or, on a tout essayé pour la case (2; 3) ; on efface la marque et on retourne à la case du haut de la pile : (2 ; 2). Mais cette case est marquée, donc on reprend où on en était : on essaie à gauche : la case est marquée
3 2	on essaie en bas : c'est un mur
1 2	on essaie en haut : c'est un mur
2 0	on a tout essayé pour la case (2; 2) ; on efface la marque et on retourne à la case du sommet de la pile : (2 ; 1). Mais cette case est marquée, donc on reprend où on en était. On essaie à gauche : c'est un mur
3 1	on essaie en bas : la case est marquée
1 1	on se déplace en haut ; on marque cette case
1 2	on essaie à droite : c'est un mur
1 0	on essaie à gauche : c'est un mur
2 1	on essaie en bas : la case est marquée
0 1	on essaie en haut: c'est un mur. on a tout essayé pour la case (1; 1) ; on efface la marque et on retourne à la case du haut de la pile : (2 ; 1). Mais cette case est

marquée, donc on reprend où on en était. Or, on a tout essayé pour la case (2; 1) ; on efface la marque et on retourne à la case du haut de la pile : (3 ; 1). Mais cette case est marquée, donc on reprend où on en était. Or, on a tout essayé pour la case (3; 1).

STOP. La pile est vide. On est donc passé par toutes les cases vides et on a essayé toutes les directions à partir d'elles. Le schéma ci-dessous résume les chemins parcourus, avant et après avoir trouvé la sortie.



Voici un exemple plus complexe avec trois chemins possibles de E à S :

Donnée

dedale2.txt

http://ow.ly/QBK1n

```
XXXXXXXXXXXXXXXXXXXXXXXXX
XEX X   X           XX   X
X X X XXXXX XXXXXX XXXX X
X X X X X X X       XXX S
X     X X X XXX XXX XXX X
XXXXX X X   XXX X XX XX X
X         X     X       X
XXX XX XXXXXX XXXX XXXX X
X    X                X X
XXXXXXXXXXXXXXXXXXXXXXXXX
```

Solutions

```
XXXXXXXXXXXXXXXXXXXXXXXXX
X.X X   X           XX   X
X.X X XXXXX XXXXXX XXXX X
X.X X X X X X       XXX.S
X.....X X X XXX XXX XXX.X
XXXXX.X X...XXX X XX XX.X
X     .....X...   X ......X
XXX XX XXXXXX.XXXX.XXXX X
X    X         ......   X X
XXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXX
X.X X   X  ........XX   X
X.X X XXXXX.XXXXXX.XXXX X
X.X X X X X.X  .... XXX.S
X.....X X X.XXX.XXX XXX.X
XXXXX.X X...XXX.X XX XX.X
X     .....X   ...X ......X
XXX XX XXXXXX.XXXX.XXXX X
X    X         ......   X X
XXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXX
X.X X   X           XX   X
X.X X XXXXX XXXXXX XXXX X
X.X X X X X X       XXX.S
X.....X X X XXX XXX XXX.X
XXXXX.X X   XXX X XX XX.X
X     ..    X     X ......X
XXX XX.XXXXXX XXXX.XXXX X
X    X.............   X X
XXXXXXXXXXXXXXXXXXXXXXXXX
```

Exercice 13.1

Essayez de comprendre le fonctionnement de ce programme en l'exécutant « à la main » sur ce petit labyrinthe :

```
XXXXX
XE  X
X X S
X   X
XXXXX
```

Exercice 13.2

Modifiez le programme Python présenté ci-dessus de telle manière que l'on puisse suivre le cheminement dans le labyrinthe : « droite », « gauche », « bas », « haut ».

Affichez aussi le labyrinthe chaque fois que la procédure `Thesee` est appelée, afin de suivre l'évolution des marques.

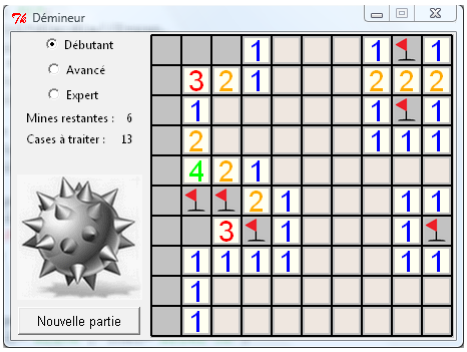
Explorez ensuite le labyrinthe de l'ex. 13.1. et vérifiez que vous avez bien compris le fonctionnement du programme.

13.3. Règles du jeu du démineur

Le *démineur* est un jeu de réflexion. Il faut trouver par déduction l'emplacement de mines dans un quadrillage. En cliquant sur une case, il y a deux possibilités : soit c'est une mine (et on a perdu la partie), soit c'est du terrain déminé. Si la case choisie est bordée par une mine, un chiffre entre 1 et 8 apparaît. Ce chiffre indique le nombre de mines adjacentes à cette case. Par adjacent, on entend qui touche une case soit par un bord, soit par un coin.

Si ce n'est pas une mine et qu'il n'y a aucune mine dans les cases adjacentes, le chiffre 0 n'apparaît pas, mais on découvre toute la zone sans mine délimitée par des mines ou le bord de la grille. C'est cette partie qui est la plus difficile à programmer, et qui nous intéressera particulièrement dans ce chapitre.

Avec le bouton droit de la souris, on peut marquer une case avec un drapeau ou un point d'interrogation. Cela sert d'aide mémoire et n'a pas d'influence sur la fin de la partie.



13.4. Code du programme

demineur.py

http://ow.ly/QBK1n

```
# Démineur

from tkinter import *
import random, winsound, sys

# Dessine la grille de jeu
def grille(nb_col, nb_lignes, dim, origine):
    x1= origine
```

```
    y1= origine
    # Détermine la largeur de la grille
    y2 = y1 + (dim*nb_lignes)
    # Détermine la hauteur de la grille
    x2 = x1 + (dim*nb_col)
    colonne = 0
    while colonne <= nb_col:
        colonne=colonne+1
        # Création de la ligne verticale
        can.create_line(x1,y1,x1,y2,width=2,fill="black")
        # Décalage de la ligne vers la droite
        x1 = x1 + dim
    x1 = origine
    ligne = 0
    while ligne <= nb_lignes:
        ligne=ligne+1
        # Création de la ligne horizontale
        can.create_line(x1,y1,x2,y1,width=2,fill="black")
        # Décalage de la ligne vers le bas
        y1 = y1 + dim

# Initialise le niveau de jeu
def init_niveau():
    global nb_col, nb_lig, nb_mines
    niveau = choix.get()
    # niveau débutant
    if niveau == 1:
        nb_col, nb_lig, nb_mines = 10, 10, 12
    # niveau avancé
    elif niveau == 2:
        nb_col, nb_lig, nb_mines = 15, 15, 30
    # niveau expert
    else:
        nb_col, nb_lig, nb_mines = 20, 20, 50
    # taille du canevas pour chaque niveau
    can.configure(width=(nb_col*dim)+gap, height=(nb_lig*dim)+gap)
    init_jeu()


# Initialisation des paramètres du jeu
def init_jeu():
    global nb_mines_cachees, nb_cases_vues, on_joue
    on_joue = True
    nb_cases_vues = 0
    can.delete(ALL)
    nb_mines_cachees = nb_mines
    affiche_compteurs()
    # Initialisation des 2 tableaux avec des chaines vides
    y = 0
    while y < nb_lig:
        x = 1
        y += 1
        while x <= nb_col:
            tab_m[x,y]= 0 # Initialisation dans le tableau des mines
            tab_j[x,y]= "" # Initialisation dans le tableau de jeu
            can.create_rectangle((x-1)*dim+gap,(y-1)*dim+gap,
                                 x*dim+gap,y*dim+gap,width=0, fill="grey")
            x += 1
    grille(nb_col, nb_lig, dim, gap) # Dessine la grille
    # place les mines aléatoirement dans la grille de jeu
    nb_mines_voisines = 0
    while nb_mines_voisines < nb_mines:
        col = random.randint(1, nb_col)
        lig = random.randint(1, nb_lig)
        if tab_m[col, lig] != 9: # Vérifie si la cellule contient déjà une mine
            tab_m[col, lig] = 9
            nb_mines_voisines = nb_mines_voisines + 1


# calcule le nombre de mines qu'il reste à trouver
def affiche_compteurs():
    decompte_mines.configure(text=str(nb_mines_cachees))
    # Décompte des cases à traiter pour la fonction gagné
    decompte_cases.configure(text=str((nb_col*nb_lig)-nb_cases_vues))
```

```
# affiche le nombre de mines restantes
def affiche_nb_mines(nb_mines_voisines, col, lig):
    global nb_mines_cachees,nb_cases_vues
    # si la case est vide
    if tab_j[col, lig] == "":
        nb_cases_vues = nb_cases_vues + 1
        # S'il y a un drapeau : modification du compteur de mines
        if (tab_j[col, lig] == "d"):
            # Ajout d'une mine
            nb_mines_cachees = nb_mines_cachees + 1
            # le drapeau est considéré comme une case vue
            nb_cases_vues = nb_cases_vues - 1
            affiche_compteurs()
        tab_j[col, lig] = nb_mines_voisines
        # Dessine un carré "ivoire"
        can.create_rectangle((col-1)*dim+gap+3,(pLig-1)*dim+gap+3,
                             col*dim+gap-3,pLig*dim+gap-3,width=0, fill="ivory")
        # Affichage du nombre de mines avec les couleurs corespondantes
        coul = ['blue','orange','red','green','cyan','skyblue','pink']
        can.create_text(col*dim-dim//2+gap, pLig*dim-dim//2+gap,
                        text=str(nb_mines_voisines),
                        fill=coul[nb_mines_voisines-1],font='Arial 22')


# calcule le nombre de mines qui touchent la case
def nb_mines_adj(col, ligne):
    if col > 1:
        min_col = col - 1
    else:
        min_col = 1
    if col < nb_col:
        max_col = col + 1
    else:
        max_col = col
    if ligne > 1:
        min_lig = ligne - 1
    else:
        min_lig = 1
    if ligne < nb_lig:
        max_lig = ligne + 1
    else:
        max_lig = nb_lig
    txtinfo = ""
    nb_mines = 0
    indice_lig = min_lig
    while indice_lig <= max_lig:
        indice_col = min_col
        while indice_col <= max_col:
            if tab_m [indice_col,indice_lig] == 9:
                nb_mines += 1
            indice_col = indice_col + 1
        indice_lig = indice_lig +1
    return nb_mines


# Affiche toutes les cases zéro adjacentes et leur bordure
def vide_plage_zero(col, ligne):
    global nb_mines_cachees, nb_cases_vues
    # si on a déjà la cellule n'est pas vide
    if tab_j[col, ligne] != 0:
        # S'il y a un drapeau on modifie le compteur de mines et de cases traitées
        if (tab_j[col, ligne] == "d"):
            nb_mines_cachees = nb_mines_cachees + 1
            nb_cases_vues = nb_cases_vues - 1
        # Affichage du fond
        can.create_rectangle((col-1)*dim+gap+3, (ligne-1)*dim+gap+3,
                             col*dim+gap-3, ligne*dim+gap-3,
                             width=0, fill="seashell2")
        # Stockage des 0 dans le tableau de jeu
        tab_j[col, ligne] = 0
        nb_cases_vues = nb_cases_vues + 1
        # Vérifie les cases voisines en croix
```

```
        if col > 1:
            nb_mines_voisines = nb_mines_adj(col-1, ligne)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne)
        if col < nb_col:
            nb_mines_voisines = nb_mines_adj(col+1, ligne)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne)
            else:
                affiche_nb_mines(nb_mines_voisines, col+1, ligne)
        if ligne > 1:
            nb_mines_voisines = nb_mines_adj(col, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col, ligne-1)
            else:
                affiche_nb_mines(nb_mines_voisines, col, ligne-1)
        if ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj (col, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col, ligne+1)
        # Vérification des diagonales pour afficher les bords de la plage zéro
        if col > 1 and ligne > 1:
            nb_mines_voisines = nb_mines_adj(col-1, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne-1)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne-1)
        if col > 1 and ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj(col-1, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne+1)
        if col < nb_col and ligne > 1:
            nb_mines_voisines = nb_mines_adj(col+1, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne-1)
            else:
                affiche_nb_mines (nb_mines_voisines, col+1, ligne-1)
        if col < nb_col and ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj(col+1, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col+1, ligne+1)
        affiche_compteurs()


def perdu():
    global on_joue
    on_joue  = False
    # Parcours du tableau pour afficher toutes les mines
    nLig = 0
    while nLig < nb_lig:
        nCol = 1
        nLig = nLig +1
        while nCol <= nb_col:
            # affichage de grille exacte
            if tab_m[nCol, nLig] == 9:
                if tab_j[nCol, nLig] == "?":
                    can.create_image(nCol*dim-dim//2+gap,
                                     nLig*dim-dim//2+gap, image = im_mine)
                elif tab_j[nCol, nLig] == "":
                    can.create_image(nCol*dim-dim//2+gap,
                                     nLig*dim-dim//2+gap, image = im_mine)
            else:
                if tab_j[nCol, nLig] == "d":
                    can.create_image(nCol*dim-dim//2+gap,
                                     nLig*dim-dim//2+gap, image = im_erreur)
            nCol = nCol+1
```

```
    can.create_text((nb_col/2)*dim-15+gap, (nb_lig/2)*dim-5+gap,
                    text='Perdu !', fill='black', font='Arial 50')
    fen.update_idletasks() # Raffraichit la fenêtre avant le bruitage
    winsound.PlaySound('explosion.wav', winsound.SND_FILENAME)


def gagne():
    can.create_text((nb_col/2)*dim-15+gap, (nb_lig/2)*dim-5+gap,
                    text='Bravo !', fill='black', font='Arial 50')
    fen.update_idletasks()
    winsound.PlaySound('gagne.wav', winsound.SND_FILENAME)


# gère le clic gauche de la souris
def pointeurG(event):
    global nb_cases_vues
    # si la partie n'est pas en cours (quand on a perdu), blocage du jeu
    if on_joue :
        nCol = (event.x - gap) // dim +1
        nLig = (event.y - gap) // dim +1
        # si la cellule est vide
        if tab_j[nCol, nLig] == "":
            # Vérifie si on est bien dans le tableau
            if nCol>=1 and nCol<=nb_col and nLig>=1 and nLig<=nb_lig:
                # Vérifie si la cellule contient une mine
                if (tab_m[nCol, nLig] == 9):
                    perdu()
                else:
                    nb_mines_voisines = nb_mines_adj(nCol, nLig )
                    if nb_mines_voisines >= 1:
                        affiche_nb_mines(nb_mines_voisines, nCol, nLig )
                        affiche_compteurs()
                    else: # Traitement des cases vides
                        vide_plage_zero(nCol, nLig)
            # Vérification des compteurs
            if ((nb_col*nb_lig) == nb_cases_vues and nb_mines_cachees == 0):
                gagne()


# gère le clic droit de la souris
def pointeurD(event):
    global nb_mines_cachees, nb_cases_vues
    # si la partie n'est pas en cours (quand on a perdu), blocage du jeu
    if on_joue :
        nCol = (event.x - gap)// dim+1
        nLig = (event.y - gap) // dim+1
        # si la cellule est vide
        if tab_j[nCol, nLig]=="":
            # Affiche le drapeau
            can.create_image(nCol*dim-dim//2+gap, nLig*dim-dim//2+gap,
                             image = im_flag)
            tab_j[nCol, nLig]="d"
            nb_cases_vues = nb_cases_vues + 1
            nb_mines_cachees = nb_mines_cachees - 1
        # si la cellule contient  un drapeau
        elif tab_j[nCol, nLig] == "d":
            # Remise à blanc
            can.create_rectangle((nCol-1)*dim+gap+3,(nLig-1)*dim+gap+3,
                                 nCol*dim+gap-3,nLig*dim+gap-3,width=0, fill="grey")
            # Affiche le ?
            can.create_text(nCol*dim-dim//2+gap, nLig*dim-dim//2+gap,
                            text="?", fill='black',font='Arial 20')
            tab_j[nCol, nLig] = "?"
            # le ? n'est pas considéré comme une case traitée
            nb_cases_vues = nb_cases_vues - 1
            # Ajoute une mine car le ? ne désigne pas une mine
            nb_mines_cachees = nb_mines_cachees + 1
        # si la cellule contient un ?
        elif tab_j[nCol, nLig] == "?":
            # Remise à blanc
            can.create_rectangle((nCol-1)*dim+gap+3,(nLig-1)*dim+gap+3,
                                 nCol*dim+gap-3,nLig*dim+gap-3,
                                 width=0, fill="grey")
            # Stocke du vide dans le tableau de jeu
```

```
            tab_j[nCol, nLig] = ""
        affiche_compteurs()
        # Vérification des compteurs
        if ((nb_col*nb_lig) == nb_cases_vues and nb_mines_cachees == 0):
            gagne()


# ---------------------------------------------------------------------------
# Début du programme
# ---------------------------------------------------------------------------

fen=Tk()
fen.title("Démineur")
fen.resizable(width=False, height=False)

# Déclarations des variables lorsqu'on ouvre la fenêtre principale
# niveau débutant par défaut
nb_col, nb_lig, nb_mines = 0,0,0
dim, gap, nb_cases_vues = 30, 3, 0
on_joue = True
# Chargement des images
im_mine = PhotoImage(file = "minej.gif")
im_erreur = PhotoImage(file = "croixj.gif")
im_flag = PhotoImage(file = "drapeauj.gif")
tab_m = {} # tableau des mines
tab_j = {} # tableau des cases modifiées par le joueur

can=Canvas(fen, width=(nb_col*dim)+gap, height=(nb_lig*dim)+gap, bg="grey")
can.bind("<Button-1>", pointeurG)
can.bind("<Button-3>", pointeurD)
can.pack(side=RIGHT)

# Frame à gauche de la grille de jeu pour disposer les boutons radios
f2 = Frame(fen)
# Création de cases à cocher pour le niveau
choix=IntVar()
choix.set(1)
case1=Radiobutton(f2)
case1.configure(text='Débutant', command=init_niveau, variable=choix, value=1)
case1.pack(anchor= NW ,padx=30)
case2=Radiobutton(f2)
case2.configure(text='Avancé', padx=3, command=init_niveau, variable=choix, value=2)
case2.pack(anchor= NW, padx=30)
case3=Radiobutton(f2)
case3.configure(text='Expert', padx=3, command=init_niveau, variable=choix, value=3)
case3.pack(anchor= NW, padx=30)
f2.pack()

# Frame à gauche de la grille de jeu pour les compteurs
f3 = Frame(fen)
# Champ pour l'affichage du décompte des mines
texte_mines = Label (f3, text = "Mines restantes :")
decompte_mines = Label (f3, text = "100")
texte_mines.grid(row=4,column=1,sticky='NW')
decompte_mines.grid(row=4,column=2,sticky='NE')
# Champ pour l'affichage du décompte des cases
texte_cases = Label (f3, text = "Cases à traiter :")
decompte_cases = Label (f3, text = "10")
texte_cases.grid(row=5,column=1,sticky='NW')
decompte_cases.grid(row=5,column=2,sticky='NE')
f3.pack()

# Frame à gauche de la grille de jeu pour disposer les boutons
f1 = Frame(fen)
bou1 = Button(f1, width=14, text="Nouvelle partie", font="Arial 10",
command=init_jeu)
bou1.pack(side=BOTTOM, padx=5, pady=5)
f1.pack(side=BOTTOM)

# Frame à gauche de la grille de jeu pour afficher l'image
f4 = Frame(fen)
photo=PhotoImage(file="mine1.gif")
labl = Label(f4, image=photo)
labl.pack(side=BOTTOM)
```

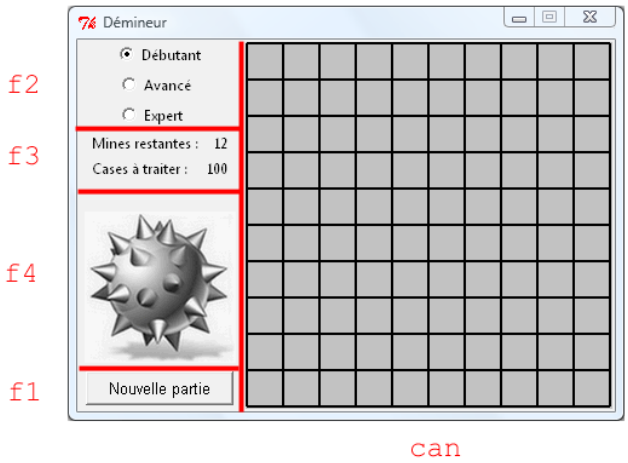
```
f4.pack(side=BOTTOM)

init_niveau()
init_jeu()
fen.mainloop()
```

13.5. Analyse du programme

Comme le programme est long (presque 400 lignes) et suffisamment commenté, nous allons nous concentrer sur les parties contenant des nouveautés.

13.5.1. Frames multiples

La fenêtre est découpée en 4 frames et un canevas (voir la dernière page du programme) :



Elles ont été placées à l'aide de l'instruction pack (voir § 8.4.3).

13.5.2. Boutons radio

Un **bouton radio** est un élément visuel des interfaces graphiques (un widget). Les boutons radio sont toujours utilisés en groupe (donc deux boutons radio au minimum), puisque leur objectif est de permettre à l'utilisateur de choisir une, et une seule, option parmi plusieurs possibles. Graphiquement (voir image ci-dessus, dans la frame `f2`), un bouton radio est représenté par un cercle et est accompagné d'une étiquette, c'est-à-dire un court texte qui décrit le choix qui lui est associé. Si l'utilisateur choisit cette option, un point apparaît à l'intérieur du cercle pour symboliser le choix, sinon le cercle reste vide.

```
# Frame à gauche de la grille de jeu pour disposer les boutons radios
f2 = Frame(fen)
# Création de cases à cocher pour le niveau
choix=IntVar()
choix.set(1)
case1=Radiobutton(f2)
case1.configure(text='Débutant', command=init_niveau, variable=choix, value=1)
case1.pack(anchor= NW ,padx=30)
case2=Radiobutton(f2)
case2.configure(text='Avancé', padx=3, command=init_niveau, variable=choix, value=2)
case2.pack(anchor= NW, padx=30)
case3=Radiobutton(f2)
case3.configure(text='Expert', padx=3, command=init_niveau, variable=choix, value=3)
case3.pack(anchor= NW, padx=30)
f2.pack()
```

Les boutons radio ont été placés dans la frame `f2`.

Ils serviront à donner une valeur à la variable `choix`. On précise que cette valeur sera du type entier :

```
choix=IntVar()
```

Par défaut, la valeur de choix est 1. Cela correspond au niveau « débutant » :

```
choix.set(1)
```

C'est l'instruction :

```
case1=Radiobutton(f2)
```

qui crée le bouton dans la frame `f2`. Il doit ensuite être configuré : l'étiquette sera « Débutant » et quand on « pressera » sur ce bouton, on appellera la procédure `init_niveau` et on mettra la valeur 1 dans la variable `choix`.

```
case1.configure(text='Débutant', command=init_niveau, variable=choix, value=1)
```

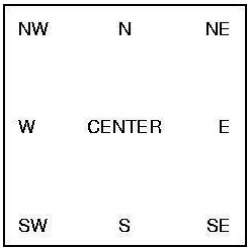
Dans la procédure `init_niveau`, on voit la ligne

```
    niveau = choix.get()
```

qui récupère la valeur dans la variable `choix`.

Le terme de *cavité restante* a été introduit au § 8.4.3

Le bouton doit finalement être placé dans la frame. L'instruction `anchor=NW` indique qu'il sera placé en haut à gauche (au nord-ouest), dans la *cavité restante* . Les valeurs possibles pour `anchor` sont :



L'instruction `padx=30` définit l'espacement avec les autres boutons.

```
case1.pack(anchor= NW,padx=30)
```

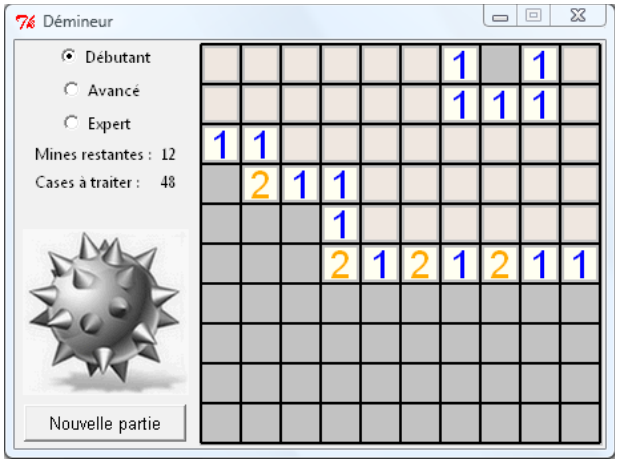
13.5.3. Récursivité

La procédure récursive s'appelle ici `vide_plage_zero`. Quand on la regarde de plus près, elle voit qu'elle ressemble beaucoup, dans l'idée du moins, à la procédure `thesee` qui nous a permis d'explorer un labyrinthe (voir § 13.2). La grande différence, c'est qu'il faut aussi regarder les cases voisines en diagonale.

```
# Affiche toutes les cases zéro adjacentes et leur bordure
def vide_plage_zero(col, ligne):
    global nb_mines_cachees, nb_cases_vues
    # si on a déjà la cellule n'est pas vide
    if tab_j[col, ligne] != 0:
        # S'il y a un drapeau on modifie le compteur de mines et de cases traitées
        if (tab_j[col, ligne] == "d"):
            nb_mines_cachees = nb_mines_cachees + 1
            nb_cases_vues = nb_cases_vues - 1
        # Affichage du fond
        can.create_rectangle((col-1)*dim+gap+3, (ligne-1)*dim+gap+3,
                             col*dim+gap-3, ligne*dim+gap-3,
                             width=0, fill="seashell2")
        # Stockage des 0 dans le tableau de jeu
```

```
        tab_j[col, ligne] = 0
        nb_cases_vues = nb_cases_vues + 1
        # Vérifie les cases voisines en croix
        if col > 1:
            nb_mines_voisines = nb_mines_adj(col-1, ligne)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne)
        if col < nb_col:
            nb_mines_voisines = nb_mines_adj(col+1, ligne)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne)
            else:
                affiche_nb_mines(nb_mines_voisines, col+1, ligne)
        if ligne > 1:
            nb_mines_voisines = nb_mines_adj(col, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col, ligne-1)
            else:
                affiche_nb_mines(nb_mines_voisines, col, ligne-1)
        if ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj (col, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col, ligne+1)
        # Vérification des diagonales pour afficher les bords de la plage zéro
        if col > 1 and ligne > 1:
            nb_mines_voisines = nb_mines_adj(col-1, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne-1)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne-1)
        if col > 1 and ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj(col-1, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col-1, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col-1, ligne+1)
        if col < nb_col and ligne > 1:
            nb_mines_voisines = nb_mines_adj(col+1, ligne-1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne-1)
            else:
                affiche_nb_mines (nb_mines_voisines, col+1, ligne-1)
        if col < nb_col and ligne < nb_lig:
            nb_mines_voisines = nb_mines_adj(col+1, ligne+1)
            if nb_mines_voisines == 0:
                vide_plage_zero(col+1, ligne+1)
            else:
                affiche_nb_mines(nb_mines_voisines, col+1, ligne+1)
        affiche_compteurs()
```

Elle permet ici de déterminer la zone sûre : si une case sans mine n'a aucun voisin avec une mine, on peut automatiquement découvrir les cases voisines, etc. La partie complexe du démineur est la propagation des cases sûres. C'est ceci qui permet d'obtenir la configuration ci-contre juste en cliquant sur le coin supérieur gauche de la grille.



On découvre toutes les cases qui ne contiennent rien (ni mine, ni entier strictement supérieur à 0), en s'arrêtant aux cases qui contiennent un entier strictement supérieur à 0.

Le calcul de la zone sûre se fait de la

manière suivante :

Pour une case donnée (sans mine et non déjà découverte),

1. on compte le nombre de mines parmi les cases adjacentes : on appelle le résultat *n*
2. on met à jour cette case
3. on teste :
    - si *n*>0, on s'arrête,
    - si *n*=0, on recommence l'opération récursivement avec toutes les cases adjacentes.

Exercice 13.3

Ajoutez une cinquième frame, où sera affiché le temps qui passe. Si le joueur dépasse une certaine limite, il aura perdu !

13.6. Ce que vous avez appris dans ce chapitre

- Vous avez vu pour la première fois en détails une procédure récursive. Elles sont assez délicates à utiliser, mais peuvent être très élégantes. Il faut savoir qui toute procédure récursive peut aussi s'écrire sous forme itérative (et vice-versa). Il faut aussi faire attention au fait que les procédures récursives peuvent provoquer des débordements de mémoire et qu'elles ne sont pas toujours recommandées. Mais elles existent et simplifient parfois la vie (comme dans les deux exemples que nous avons vus), à condition de les maîtriser.
- Les boutons radio sont pratiques pour choisir une valeur parmi plusieurs possibles.
- On peut découper une fenêtre en plusieurs frames.