「プログラミング演習１(Python)

レポート３」

クラス: c 　　　担当教員名: 小林 郁夫

学籍番号: 20K1026 　　氏名: 日比野 将己

**1, 課題：**

　迷路探索機能について、試した各アルゴリズムの説明を行え。さらに、各アルゴリズムを比較し、考察せよ。

* 表紙を付けること。
* 問題４と問題５のプログラムは、棚ⓢくしている部分を付録として最後に掲載せよ。

**2, 課題の目的：**

* 迷路プログラム作成を通して、幅優先探索と深さ優先探索の考え方を完全に理解する。
* また、自身が見返したり他人が見たりしても完璧に理解できるレポートを目指す。
* 迷路やその他の構造がどちらのアルゴリズムに適しているかを考察する。

**3, 本論**

　ここでは、まず幅優先探索と深さ優先探索の２つのアルゴリズムについて

説明していく。

　まず、初めに、幅優先探索について説明する。このアルゴリズムは、探索の開始点から近いノード（要素）から順番に訪問していく方法である。このアルゴリズムを考えるうえでの一番の大きな特徴は、これから訪問するノードを保持しておく領域（ブッファ）に、キューと呼ばれるデータ構造を用いることである。キューは、最初に入ってきたノードが最初に取り出される、デキューという特徴があり、この性質をFirst-In First-outという。これを用いることで、このデータ構造に早く並んだ順番で、処理の順番が回ってくるようになる。

　今回のプログラムでは、このキューをリストを用いて疑似的に再現することで幅優先探索のアルゴリズムを可能にしている。具体的には、

1. リスト（キュー）を定義する。
2. 今いる場所の上下左右が探索可能なら、リストにそのノードを追加する（.append）。
3. インデックス０の要素を取り出し(.pop(0))、探索する。

というように、プログラムを設計することで、キューを再現し、これを繰り返し行っていくことで、図1のような幅優先探索を可能にする。

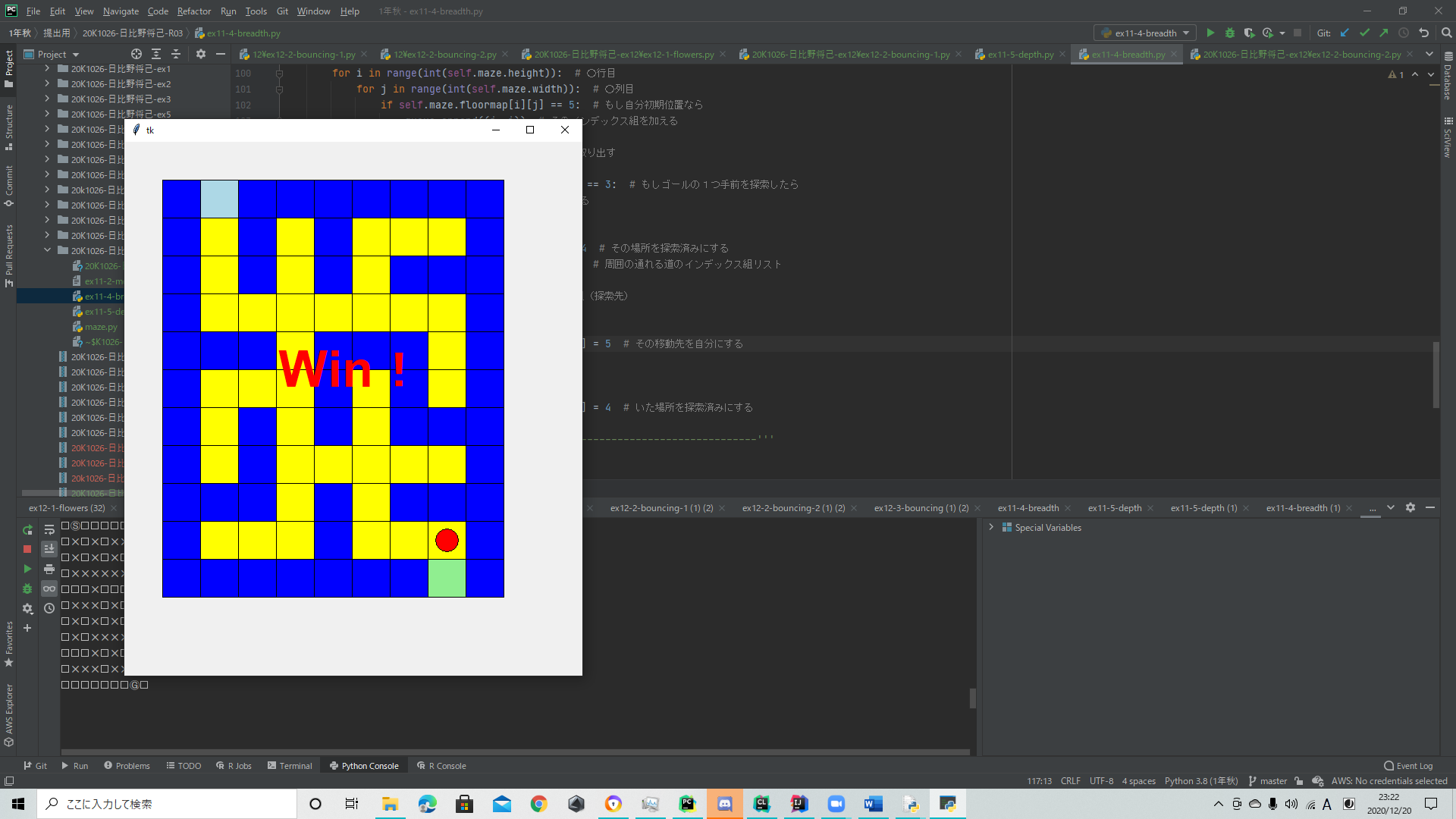


図１, 幅優先探索

　次に、深さ優先探索について説明する。このアルゴリズムは、探索の開始点からリンクをできるだけたどって遠いノードまで訪問し、いける頂点まで行ったら戻ってくるというのを繰り返し行う方法である。このアルゴリズムを考える上での一番の特徴は、これから訪問するノードを保持しておく領域に、幅優先探索のキューとは対照的な、スタックと呼ばれるデータ構造を用いるということである。このスタックに要素を入れることをプッシュするといい、スタックから要素を取り出すことをポップするという。スタックは、最後に積まれた要素が、最初に取り出されるという特徴を持ち、この性質をLast-In Frist-outという。また、一番新しい要素が置かれた場所をスタックトップ、一番古い要素が置かれた場所をスタックの底という。

　今回のプログラムでは、キュー同様、このスタックをリストを用いて疑似的に再現することで深さ優先探索のアルゴリズムを可能にしている。また、迷路においては、プッシュする順序を決めておく（今回は、上→左→右→下）ことが肝心である。具体的には、

1. リスト（スタック）を定義する。
2. 今いる場所の上下左右が探索可能なら、リストに上→左→右→下の順でプッシュする(.append)。
3. スタックトップ（最後のインデックス）から要素をポップし(.pop())、探索する。

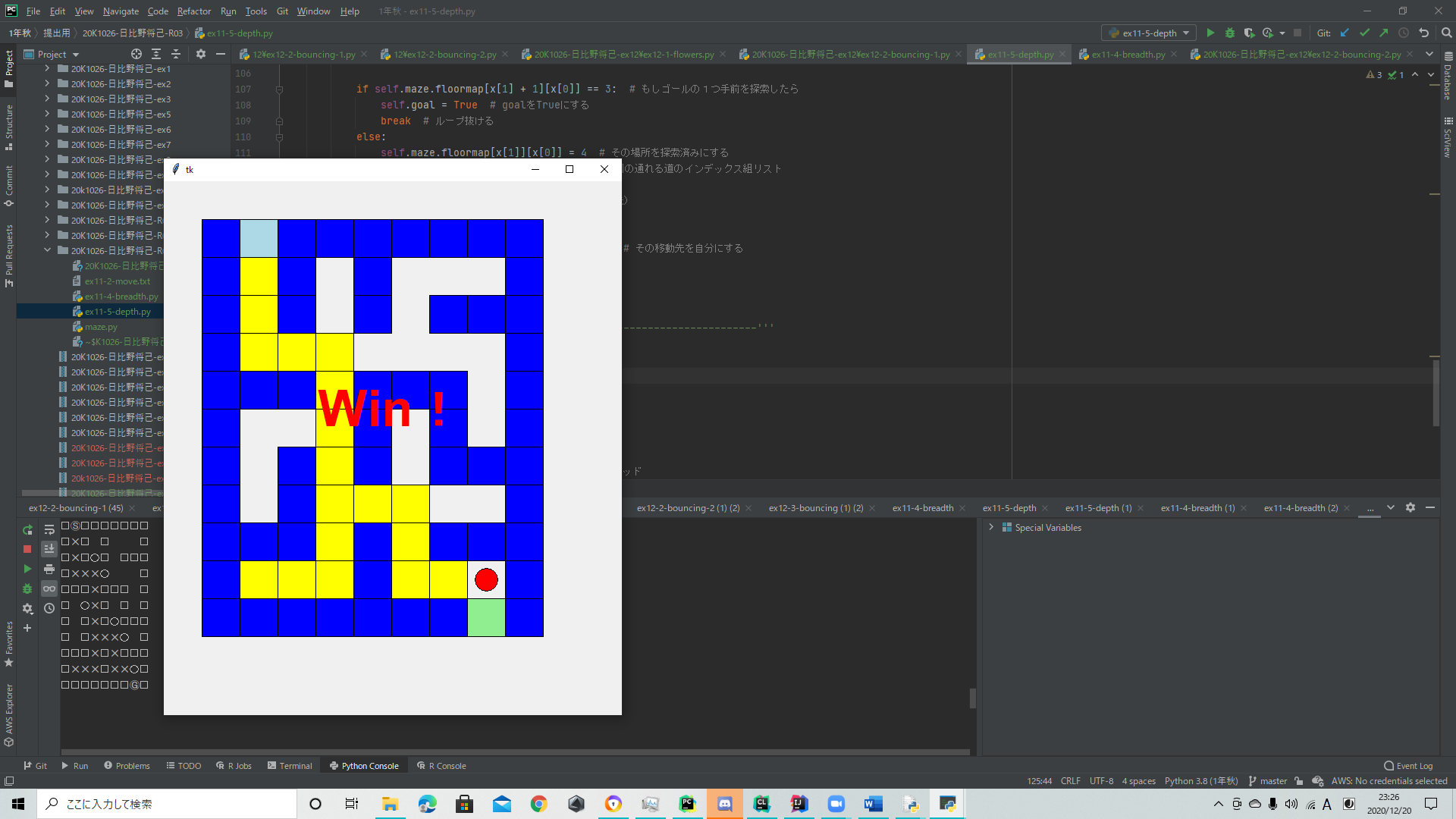
というように、プログラムを設計することで、スタックを再現し、これを繰り返し行っていくことで、図２のような深さ優先探索を可能にする。

図２, 深さ優先探索

**４, 考察：**

　今回、幅優先探索と深さ優先探索のアルゴリズムの特徴をプログラムと交えながら論じ、比較してきた。この比較から、アルゴリズムの優劣を考えるとすると、私は、幅優先探索の方が迷路探索に適していると考える。

確かに、深さ優先探索だと、探索する道が少ないため、探索順序次第では、とても早く導けるかもしれない。しかし、深い道が多数あったり、探索順序が悪かったりすると非常に時間がかかってしまうと考えられる。

　それに対し、幅優先探索は、複数の道を同時に探索するため、無駄な行き止まりの道の探索を最小限に抑えられ、また、スタート地点に近い順に探索していくため、ゴールまでの最適解（最短道）を導けるのではないかと考えられる。

　以上の考察から私は、迷路探索のおける探索アルゴリズムは、幅優先探索の方が適していると考える。

**５,　参考文献：**

* 講義資料
* Pythonによるプログラミング

**６, 付録：**

今回、２つの探索プログラムを作成するにあたって、実際に探索処理を行っているコード箇所をそれぞれ添付する。

（課題４-幅優先探索）

# 幅優先探索

def is\_valid(self, x, y): # 道か判定するメソッド p222

if self.maze.floormap[y][x] == 0 or self.maze.floormap[y][x] == 3: # 道かゴールなら

return True # Trueを返す

def neighbors(self, i, j): # 周りのリスト作成メソッド p223

x = [(i, j - 1), (i - 1, j), (i + 1, j), (i, j + 1)] # 上、左、右、下

value = [v for v in x if self.is\_valid(v[0], v[1])] # もしis\_validがTrueならvを与える

return value # 周囲の通れる道のインデックス組リスト

def breadth\_search(self): # 幅優先探索メソッド p252～255（教科書を参考に死ぬ気でググるしかない）

queue = [] # 移動先リスト

for i in range(int(self.maze.height)): # 〇行目

for j in range(int(self.maze.width)): # 〇列目

if self.maze.floormap[i][j] == 5: # もし自分初期位置なら

queue.append((j, i)) # そのインデックス組を加える

while True:

x = queue.pop(0) # queueから最初の組を取り出す

if self.maze.floormap[x[1] + 1][x[0]] == 3: # もしゴールの１つ手前を探索したら

self.goal = True # goalをTrueにする

break # ループ抜ける

else:

self.maze.floormap[x[1]][x[0]] = 4 # その場所を探索済みにする

next = self.neighbors(x[0], x[1]) # 周囲の通れる道のインデックス組リスト

for i in next: # 1つのインデックス組（探索先）

queue.append(i) # queueに追加

print(queue)

self.maze.floormap[i[1]][i[0]] = 5 # その移動先を自分にする

self.redraw() # 再描写

tk.update() # 再描写

time.sleep(0.5) # 0.5秒停止

self.maze.floormap[i[1]][i[0]] = 4 # いた場所を探索済みにする

（課題５-深さ優先探索）

# 深さ優先探索

def is\_valid(self, x, y): # 道か判定するメソッド p222

if self.maze.floormap[y][x] == 0 or self.maze.floormap[y][x] == 3: # 道かゴールなら

return True # Trueを返す

def neighbors(self, i, j): # 周りのリスト作成メソッド p223

x = [(i, j - 1), (i - 1, j), (i + 1, j), (i, j + 1)] # 上、左、右、下

value = [v for v in x if self.is\_valid(v[0], v[1])] # もしis\_validがTrueならvを与える

return value # 周囲の通れる道のインデックス組リスト

def depth\_search(self): # 深さ優先探索メソッド p252～255（教科書を参考に死ぬ気でググるしかない）

stack = [] # 移動先リスト

for i in range(int(self.maze.height)): # 〇行目

for j in range(int(self.maze.width)): # 〇列目

if self.maze.floormap[i][j] == 5: # もし自分初期位置なら

stack.append((j, i)) # そのインデックス組を加える

while True:

x = stack.pop() # stackから最後の組を取り出す

if self.maze.floormap[x[1] + 1][x[0]] == 3: # もしゴールの１つ手前を探索したら

self.goal = True # goalをTrueにする

break # ループ抜ける

else:

self.maze.floormap[x[1]][x[0]] = 4 # その場所を探索済みにする

next = self.neighbors(x[0], x[1]) # 周囲の通れる道のインデックス組リスト

for i in next: # 1つのインデックス組（探索先）

stack.append(i) # stackに追加

print(stack)

self.maze.floormap[i[1]][i[0]] = 5 # その移動先を自分にする

self.redraw() # 再描写

tk.update() # 再描写

time.sleep(0.5) # 0.5秒停止