8パズル探索アルゴリズム

目次

プログラム概要

8パズルをBFS,DFS,A*の3つの探索プログラムを使って解くプログラムをPythonで作成した。
プログラム中ではパズルを9桁のint型変数、または要素数9の1次元リストとして扱い、適時変換してパズルの操作やリストの参照をしている。

例： 123
456 → int型変数：123456780　リスト：[1,2,3,4,5,6,7,8,0]
780　※0は空き駒

まず、解くことが可能な8パズル（駒の並べ方によって半分は解くことが不可能）を作成し、3つのアルゴリズムそれぞれで探索を行い、探索終了後に完成形までの経路を調べ、探索にかかった時間を記録する。
これを100回繰り返し行い、探索にかかった時間（実行時間）、アルゴリズムのループ回数（処理回数）、経路の長さ（経路長）を集計して平均を求める。
集計結果とすべての探索経路を、アルゴリズムごとにテキストファイルとして出力。

アルゴリズムの手順

BFS,DFSは第3回講義資料、A*は第4回講義資料と教科書中に書かれている手順をもとにプログラムを作成した。
ソースコードはこちら。
節点を格納するコレクションとして次の3つを使う。

- openList: 未展開の節点の順序を記録するためのリスト（BFSとDFSでの探索のみ）
  `[int型パズル1, int型パズル2, int型パズル3,...]`
- openDict: 未展開の節点を、親節点・節点の深さ・fと紐付けて保存する辞書型変数
  openListとは異なり、情報が付与された節点の集まり （子節点の深さとfはA*での探索のみ）
  `{int型パズル: {"parentInt": 親のint型パズル, "depth": 節点の深さ, f: A*のf(n)},...}`
- closedDict: 展開済みの子節点をopenDictと同様に保存する辞書型変数

openDictとclosedDictに2次元リストを使わなかった理由は、リストでは要素を検索するのに時間がかかってしまうため。

BFS(Breadth-First Search)

> 同じ深さの節点をすべて探索してから次の深さの節点を探索する方法。
> 出典：第3回講義資料

1. 出発節点をリストopenList、辞書openDictに入れる。
2. openDicが空なら探索は失敗、終了。
3. openListの先頭の節点nを取り出し、これをもとにopenDictから情報付きの節点nを取り出し、closedDictに追加する。
4. nが目標節点なら探索成功、終了。
5. nが展開できるなら展開し、得られた子節点のうちopenDictにもclosedDictにもないものを、openList,openDictの**末尾**に追加する。
6. 2へ戻る

DFS(Depth-First search)

> ある節点を探索したとき、次の探索節点はその子節点の1つとし、その節点から深い節点を優先的に調べていく方法。
> 出典：第3回講義資料

5以外はBFSと同じ手順。

5. nが展開できるなら展開し、得られた子節点のうちopenDictにもclosedDictにもないものを、openList,openDictの**先頭**に追加する。

A*

> 評価関数f(n)を使って各節点の評価値を求め、評価値の良いものから優先的に探索する方法。
> ここで、
> f(n) = g(n) + h(n)
> g(n) : 出発節点から節点nに至る最適経路のコストの評価値
> h(n) : 節点nから目標節点に至る最適経路のコストの評価値（ヒューリスティック関数）
> また、
> h = h1 + h2
> h1 : 正しい位置に置かれていない駒の個数
> h2 : 各駒の現在位置と正しい位置との間の距離の総和
> とする。
> 出典：人工知能の基礎

3,5以外はBFSと同じ手順

3. openDictからfの値が最小となる節点nを取り出し、closedDictに追加する。
   （講義資料や教科書ではソーティング済みのリストから先頭の節点を取り出すが、ソーティングは計算コストが大きいため、計算コストが小さいかつ同じ結果が得られるmin()関数を用いた。）
4. nが目標節点なら探索成功、終了。
5. nが展開できるなら展開し、得られた子節点全てに対し以下の手順を行う。
    1. nがopenDictまたはclosedDictに含まれていなければ、nをopenDictに追加する。
    2. nがopenDictに含まれていて、fがnのものより大きければ、openDictのnを上書きする。
    3. nがclosedDictに含まれていて、fがnのものより大きければ、closedDictからnを取り除き、openDictへnを追加する。

実行結果

集計表

実行回数 = 100

アルゴリズム	平均実行時間[秒]	平均処理回数[回]	平均経路長
BFS	2.263818	92862.58	22.93
DFS	3.102956	121282.10	47658.85
A*	0.193404	1061.63	23.23

考察

平均実行時間と平均処理回数の大きさはどちらも
　　A* < BFS < DFS
となった。 特にA*と他の2つでは、平均実行時間で10倍以上、平均処理回数で約100倍という大きな差がでた。
しかし平均経路長の大きさは
　　BFS < A* < DFS
となり、BFSがA*より僅かに小さい結果となった。その理由についてこう考える。
BFSは、同じ深さの節点をすべて探索してから次の深さの節点を探索するので、必ず最短の経路を探索できるのに対し、 A*はヒューリスティック関数と節点の深さの和が最小なものを、最短経路である可能性が1番高い節点として選んで展開している。
ヒューリスティック関数

> h = h1 + h2
> 　　h1 : 正しい位置に置かれていない駒の個数
> 　　h2 : 各駒の現在位置と正しい位置との間の距離の総和
> 出典：人工知能の基礎

これは必ずしも最短経路の節点に対して最小の値を返すものではないので、A*の探索経路は最短経路より少し長くなったと考えられる。
8パズルの解を3つのアルゴリズムの内どれかで求めるときは、

- 最短経路を必ず求めたいならBFS
- 少しだけ最短経路より長くても、大量の8パズルを早くときたいならA*

経路一覧

以下に1つのパズル[6,2,7,3,8,1,4,5,0]に対し、3つのアルゴリズムで探索したときの経路を記す。 これから、プログラムは正しく8パズルを解けていることがわかる。

経路：BFS

```
実行時間：3.2437秒  処理回数：131431回  経路長：25
     1: 627381450
     2: 627381405
     3: 627301485
     4: 607321485
     5: 067321485
     6: 367021485
     7: 367201485
     8: 367210485
     9: 360217485
    10: 306217485
    11: 036217485
    12: 236017485
    13: 236107485
    14: 236170485
    15: 236175480
    16: 236175408
    17: 236105478
    18: 236150478
    19: 230156478
    20: 203156478
    21: 023156478
    22: 123056478
    23: 123456078
    24: 123456708
    25: 123456780
```

経路：DFS

```
実行時間：3.2301秒  処理回数：126414回  経路長：52703
     1: 627381450
     2: 627380451
     3: 620387451
     4: 602387451
     5: 062387451
     6: 362087451
     7: 362807451
     8: 302867451
     9: 032867451
    10: 832067451
    11: 832607451
    12: 802637451
```

```
     13: 082637451
     14: 682037451
     15: 682437051
     16: 682437501
     17: 682407531
     18: 602487531
     19: 062487531
     20: 462087531
     21: 462807531
     22: 402867531
     23: 042867531
           ⋮
  52703: 123456780
```

**経路：A*

```
実行時間：0.1166秒 処理回数：1072回 経路長：27
      1: 627381450
      2: 627381405
      3: 627301485
      4: 627310485
      5: 620317485
      6: 602317485
      7: 062317485
      8: 362017485
      9: 362107485
     10: 362170485
     11: 362175480
     12: 362175408
     13: 362105478
     14: 362150478
     15: 360152478
     16: 306152478
     17: 036152478
     18: 136052478
     19: 136452078
     20: 136452708
     21: 136402758
     22: 136420758
     23: 130426758
     24: 103426758
     25: 123406758
     26: 123456708
     27: 123456780
```

ソースコード

8パズル作成

```
# 解くことが可能な8パズルを作成する関数
def makePuzzle(GOAL_INT: int) -> int:
    goal = cnvIntToList(GOAL_INT)   # 完成形をリスト化
    goal[goal.index(0)] = 9         # 0を9に変えておく
    start = copy.deepcopy(goal)     # 開始形を作成

    # シャッフルして、それが解ける配置なのか調べる、を繰り返す
    while(True):
        random.shuffle(start)       # パズルをシャッフルする
        copyStart = copy.deepcopy(start)
        cnt = 0

        # 空き駒の最短距離を計算
        startRow = start.index(9) // 3    # 9の位置（行・列）を求める
        startCol = start.index(9) % 3
        goalRow = goal.index(9) // 3
        goalCol = goal.index(9) % 3
        zeroDistance = abs(startRow - goalRow)      # 横方向・縦方向の距離を取得し、zeroDistanceに加算
        zeroDistance += abs(startCol - goalCol)

        # startをgoalにするための入れ替え回数の偶奇が、zeroDistanceの偶奇と等しければ解ける->終了
        for i in range(1,10):
            if copyStart == goal:
                if cnt % 2 == zeroDistance:     # 条件を満たした場合
                    start[start.index(9)] = 0       # 9を0に戻す
                    print("start:{0}".format(start))
                    return cnvListToInt(start)      # int化したstartを返す
                continue                        # 条件を満たさない場合、ループの最初に戻る
            targetIndex = copyStart.index(i)
            if targetIndex != i-1:          # iの配置が異なれば入れ替える
                copyStart = swap(copyStart, targetIndex, i-1)
                cnt += 1
```

探索アルゴリズム

```
# 探索アルゴリズム関数。出発節点・目標節点・探索アルゴリズムを引数にして、得られた経路・経路長・処理回数を返す。
@timeDeco
def searchAlgorithm(START_INT: int, GOAL_INT: int, searchType: SearchType):

    # step1 出発節点をOpenListに入れる->要素の検索の計算コストが小さい辞書を用いる
    if searchType == SearchType.BFS or searchType == SearchType.DFS:
        openList = deque([START_INT])       # BFS,DFS用に節点のリストを作成。
    openDic = defaultdict(dict)             # OpenListの代わりの辞書
    closedDic = defaultdict(dict)           # ClosedListの代わりの辞書
    openDic[START_INT]["parentInt"] = -1
```

```
    if searchType == SearchType.ASTAR:      # A*アルゴリズムなら深さとfも保存
        openDic[START_INT]["depth"] = 0
        openDic[START_INT]["f"] = heuristic(START_INT, GOAL_INT)

    cnt = 1     # 処理回数
    while(True):
        # step2 OpenListが空なら探索失敗、終了。
        if not openDic:
            print("{0}\r探索失敗：openListが空になりました。プログラムを中断します。{1}"
                  .format(Color.RED, Color.END))
            exit()

        # step3 openDicから節点を取り出し、ClosedDicに追加
        if searchType == SearchType.BFS or searchType == SearchType.DFS:
            # BFSかDFSなら先頭の節点を取り出す
            puzzleInt: int = openList.popleft()
        elif searchType == SearchType.ASTAR:
            # A*ならfが最も小さい節点を取り出す
            puzzleInt: int = min(openDic.items(),
            key=lambda puzzleDict: puzzleDict[1]["f"])[0]
        closedDic[puzzleInt] = openDic.pop(puzzleInt)

        # step4 取り出した節点が目標節点なら探索は成功、終了。
        if puzzleInt == GOAL_INT:
            searchTypeValue = "<" + searchType.value + ">"
            print("{0}\r ✔ {1}探索成功 {2:7} | ".format(
                Color.GREEN, Color.END, searchTypeValue), end="")
            break

        # step5
        childInts = puzzleExpand(puzzleInt)     # 節点を展開

        # step5-続き（BFS,DFSの場合）
        if searchType == SearchType.BFS or searchType == SearchType.DFS:
            for childInt in childInts:
                # 子節点がopenDic,closedDicになければ
                if childInt not in openDic.keys() and childInt not in closedDic.keys():
                    openDic[childInt]["parentInt"] = puzzleInt  # openDicに子節点と親節点を保存
                    if searchType == SearchType.BFS:    # BFSなら子節点を末尾に追加
                        openList.append(childInt)
                    elif searchType == SearchType.DFS:  # DFSなら子節点を先頭に追加
                        openList.appendleft(childInt)

        # step5-続き（A*の場合）
        elif searchType == SearchType.ASTAR:
            fList = []      # fのリスト
            sortedChildPuzzleInts = []
            for childInt in childInts:
                # 子節点から目標節点に至る最適経路のコストの評価値
```

```
                h = heuristic(childInt, GOAL_INT)
                # 出発節店から子節点に至る最適経路のコストの評価値
                g = closedDic[puzzleInt]["depth"]
                f = h + g
                childDict = {       # この子節点の辞書を作成
                    "parentInt": puzzleInt,
                    "depth": g + 1,
                    "f": f
                }
                if childInt in openDic.keys():          # 子節点と同じ配置の節点がopenDicにあり、
                    if openDic[childInt]["f"] > f:          # 子節点の方がfが小さい場合、
                        openDic[childInt] = childDict       # openDicを上書き
                elif childInt in closedDic.keys():      # 子節点と同じ配置の節点がclosedDicにあり、
                    if closedDic[childInt]["f"] > f:        # 子節点の方がfが小さい場合、
                        del closedDic[childInt]             # closedDicから削除し、
                        openDic[childInt] = childDict       # openDicに追加
                else:
                    openDic[childInt] = childDict       # どちらにもない場合そのままopenDicに追加

        if cnt % 100 == 0:
            print('\r{0}回目の実行'.format(cnt), end="")
        cnt += 1

        # step6 step2へ戻る

    # closedDicを使って出発節点までの経路をたどる
    childInt = GOAL_INT
    puzzleRoute = deque([childInt])     # 経路中の節点をint型で保存するリスト
    startTime = time.time()
    while(True):
        if childInt in closedDic:   # 子節点がclosedDicの中にある場合、親節点を取得
            parentInt = closedDic[childInt]["parentInt"]
            if parentInt == -1:     # 親節点が出発節点なら経路探索成功
                break
            puzzleRoute.appendleft(parentInt)   # 経路リストに親節点を追加
            childInt = parentInt                # 子節点<-親節点
        else:
            print("{0}経路が見つかりませんでした。プログラムを中断します。{1}"
                  .format(Color.RED, Color.END))
            exit()
        if time.time() - startTime > 5:     # 経路探索に5秒以上かかった場合タイムアウトする。
            print("{0}開始時のパズルまでの経路を見つからないためタイムアウトしました。"
            "プログラムを中断します。{1}".format(Color.RED, Color.END))
            exit()
```

```
    return puzzleRoute, len(puzzleRoute), cnt
```

ヒューリスティック関数

```
# A*アルゴリズムのヒューリスティック値を取得する関数
def heuristic(puzzleInt: int, GOAL_INT: int) -> int:
    puzzleList = cnvIntToList(puzzleInt)    # 2節点ともリスト化
    goalList = cnvIntToList(GOAL_INT)

    h1 = 0      # ヒューリスティック値1：正しい位置に置かれていない駒の数
    for i in range(len(puzzleList)):
        if puzzleList[i] == 0:      # 空き駒は非対象
            continue
        if puzzleList[i] != goalList[i]:    # 位置が正しくない場合+1
            h1 += 1

    h2 = 0      # ヒューリスティック値2：各駒の現在位置と正しい位置の間の距離の総和
    for i in range(1, len(puzzleList)):
        puzzleRow = puzzleList.index(i) // 3    # 2節点の行と列を求める
        puzzleCol = puzzleList.index(i) % 3
        goalRow = goalList.index(i) // 3
        goalCol = goalList.index(i) % 3
        h2 += abs(puzzleRow - goalRow)      # 横方向・縦方向の距離を取得し、h2に
加算
        h2 += abs(puzzleCol - goalCol)

    return h1 + h2
```

節点展開

```
# 節点を展開する関数。int型の節点を引数にして子節点のリストを返す。
def puzzleExpand(puzzleInt: int) -> list:
    puzzleList = cnvIntToList(puzzleInt)    # 節点をリストに変換
    if 0 in puzzleList:
        zeroPosi: int = puzzleList.index(0)+1   # 0(空き駒)を取得
    else:
        print("パズルに空き駒(0)が含まれていません。プログラムを中断します。
\npuzzleList:{}"
              .format(puzzleList))
        exit()

    # 空き駒の上下左右の位置を計算
    up = zeroPosi-3
    left = zeroPosi-1
    right = zeroPosi+1
    down = zeroPosi+3

    # 右または左に動かしたときに、3と4、6と7の間を移動しないようにする
```

```
    if zeroPosi==3 or zeroPosi==6:
        right = 0
    elif zeroPosi==4 or zeroPosi==7:
        left = 0
    aroundPosi: list = [down, right, left, up]

    # パズル(3x3)の範囲外の位置を除外し、changeablePosiに保存
    changeablePosi = []
    for i in aroundPosi:
        if 1<=i<=9:
            changeablePosi.append(i)

    # 空き駒とchangeablePosiを交換し、リストnewPuzzlesに保存
    newPuzzles: list = []
    for i in changeablePosi:
        newPuzzleList = swap(puzzleList, i-1, zeroPosi-1)
        newPuzzleInt = cnvListToInt(newPuzzleList)
        newPuzzles.append(newPuzzleInt)

    return newPuzzles
```

その他

```
import copy
import time
import random
from collections import deque
from collections import defaultdict
from enum import Enum
import enum
import os

# ターミナルに出力する際に、文字に色を付けるためのクラス
class Color:
    RED = '\033[31m'
    GREEN = '\033[32m'
    END = '\033[0m'
    ACCENT = '\033[01m'

# 関数searchAlgorithmの処理時間を計測し、戻り値に処理時間を追加する関数
def timeDeco(func):
    def wrapper(*args, **kwargs):
        startTime = time.time()     # 開始時間
        puzzleRoute, routeLength, searchCnt = func(*args, *kwargs)  #
searchAlgorithmの戻り値を各変数に代入
        searchTime = time.time() - startTime                        # 処理時
間
        print("実行時間：{0:9.6f}秒 | 処理回数：{1:6}回 | 経路長：{2}"
              .format(searchTime, searchCnt, len(puzzleRoute)))
        return {"puzzleRoute":puzzleRoute, "routeLength":routeLength,
                "searchCnt":searchCnt, "searchTime":searchTime}
```

```
    return wrapper

# 探索アルゴリズム名のenum
@enum.unique
class SearchType(Enum):
    BFS = "BFS"
    DFS = "DFS"
    ASTAR = "ASTAR"
    ALL = "ALL"

# 長さ9のパズルリストを9桁のint型に変換する関数
def cnvListToInt(li: list) -> int:
    num = 0
    for i in li:
        num = num * 10 + i
    return num

# int型の9桁のパズルをリストに変換する関数
def cnvIntToList(num: int) -> list:
    li = []
    for i in range(9):
        li.insert(0, num % 10)
        num = num // 10
    return li

# リストの2要素を入れ替えたリストを返す関数
def swap(tmp: list, index1: int, index2: int) -> list:
    li = copy.deepcopy(tmp)
    li[index1], li[index2] = li[index2], li[index1]
    return li

# 1探索ごとに集計したresultsを各要素ごとに集計する関数
def getTotalData(results: list) -> dict:
    runCnt = len(results)
    puzzleRoutes = []   # 経路リスト
    routeLengths = []   # 経路リストの長さ
    searchTimes = []    # 実行時間
    searchCnts = []     # 処理回数
    for result in results:
        puzzleRoutes.append(result["puzzleRoute"])
        routeLengths.append(result["routeLength"])
        searchTimes.append(result["searchTime"])
        searchCnts.append(result["searchCnt"])

    totalData = {"runCnt": runCnt, "puzzleRoutes": puzzleRoutes,
"routeLengths": routeLengths,
                 "searchTimes": searchTimes, "searchCnts": searchCnts}
    return totalData

# printとファイル出力用に、集計結果の文字列を返す関数
def getMeanStr(searchType: SearchType, totalData: dict) -> str:
    runCnt = totalData["runCnt"]
    meanRouteLength = sum(totalData["routeLengths"])/runCnt
    meanSearchTimes = sum(totalData["searchTimes"])/runCnt
```

```
    meanSearchCnts = sum(totalData["searchCnts"])/runCnt

    meanStr = ("探索アルゴリズム：{0:5} | 実行回数：{1}回 | 平均実行時間：{2:9.6f}
秒 | "
               "平均処理回数：{3:8}回 | 平均経路長：{4}").format(
        searchType.value, runCnt, meanSearchTimes, meanSearchCnts,
meanRouteLength)
    return meanStr

# 集計結果と経路をテキストファイルに出力する関数
def outputResult(path: str, totalData: dict, meanStr: str, searchType:
SearchType) -> None:
    with open(path, mode='w') as f:
        f.write('探索アルゴリズム：{}\n'.format(searchType.name))
        f.write(meanStr + '\n')
        for i in range(totalData["runCnt"]):
            f.write("\n-------------------------------------------------
\n\n")
            f.write("経路{0} 実行時間：{1:.4f}秒 処理回数：{2}回 経路長：{3}\n"
                    .format(i+1, totalData["searchTimes"][i],
totalData["searchCnts"][i],
                            totalData["routeLengths"][i]))
            puzzleRoute = totalData["puzzleRoutes"][i]
            for j in range(len(puzzleRoute)):
                puzzleInt = puzzleRoute[j]
                f.write("{0:>6}: {1:09}\n".format(j+1, puzzleInt))

# main関数。探索アルゴリズム（複数可）・完成形・探索回数・出力先のパスを引数にわたす。
def main(*searchTypes: SearchType, GOAL_INT: int = 123804765,
         runCnt: int = 10, outputDirectoryPath: str = '8パズル探索結果/'):

    start = time.time()
    results = defaultdict(list)     # 探索結果
    for i in range(runCnt):
        print("探索{} ".format(i+1), end="")
        START_INT = makePuzzle(GOAL_INT)    # パズル作成
        if SearchType.ALL in searchTypes:   # SearchType.ALLが指定された場合、
            searchTypes = (SearchType.BFS, SearchType.DFS,
SearchType.ASTAR)    # タプルsearchTypesを上書き
        for searchType in searchTypes:      # 指定されたアルゴリズムで実行し、
resultsに追加
            results[searchType].append((searchAlgorithm(START_INT,
GOAL_INT, searchType)))

    # 指定パスのフォルダが存在しない場合は作成
    if not os.path.exists(outputDirectoryPath):
        os.mkdir(outputDirectoryPath)

    print("{0}集計結果{1}".format(Color.ACCENT, Color.END))

    for searchType, result in results.items():
        totalData = getTotalData(result)        # 集計
        meanStr = getMeanStr(searchType, totalData)     # 集計結果の文字列
        print(meanStr)
```

```
        outputFilePath = outputDirectoryPath + searchType.value + '.txt'
# アルゴリズムごとにファイル名を変更
        outputResult(outputFilePath, totalData, meanStr, searchType)    #
集計結果と経路を出力

    print("総時間：{:.6f}秒".format(time.time() - start))
    print("{0}All Complete!{1}".format(Color.GREEN, Color.END))

main(SearchType.ALL, GOAL_INT=123456780, runCnt=100)
```