MemeticAlgorithm

以下は/nasu2017/share/2018/修士論文/kai-master-thesis.pdfに則って記述している。

メメティックアルゴリズム

古典的なルーティング問題の一つである配送計画問題(Vehicle Routing Problem: VRP)は、顧客数の増加に伴い現実的な時間で最適解を求めることが困難であることが知られている(NP困難)。そこで現在オペレーションズ・リサーチの分野で主流となっているのがメタ戦略による近似解法である。メメティックアルゴリズムはメタ戦略の一つであり、遺伝的アルゴリズム(GeneticAlgorithm: GA)と局所探索法(Local Search: LS)を組合せた手法である。GAは解の大域的探索能力に優れているが、一方で局所探索能力が欠けている。そこでLSを組み合わせることで弱点を克服している。以下は本研究におけるMAの実装に関する説明である。なお、本研究で実装したプログラムはVRPの中でも容量について制約があるCapacitated VRP(CVRP)を対象としている。

遺伝的アルゴリズム

遺伝的アルゴリズム(Genetic Algorithm: GA)は生物進化を模したアルゴリズムで、自然適応の仕組みをコンピュータシステムに取り入れる方法として開発された。GAでは、複数の探索点を集団(population)、各探索点を個体(individual)、個体の評価値を適合度(fitness)と表現する。基本的な枠組みは、集団の中から適合度の高い個体を優先して選択肢、交叉(crossover)、突然変異(mutation)させ、次の世代へ残す個体を淘汰することを繰り返して解を探索する。今世代の個体の中に事前に設定した終了条件を満たす個体が出現した時、解の探索を修了する。集団の生成方法、交叉する個体の選択方法、交叉方法、突然変異確率、次世代の集団の選択方法の設計は解の探索性能に大きく影響する。本プログラムでは交叉方法としてEdge Assembly Crossover(EAX)を使用しており、突然変異は行っていない。また、各親ペアからEAXによる交叉で生成される子個体は親に類似する傾向がある。そこで次世代へ残す個体の選択は、1組の親ペア(親A、親B)によって生成された子個体全てと親Aを比較し、子個体の中で最も適合度が高い解が親Aよりも良ければ集団中の親Aと入れ替える。親と子を家族と見立てその中で最良の個体のみ次世代に残すことから家族内淘汰と呼び、これにより集団中の多様性を維持することが目的である。

Edge Assembly Crossover

EAXはもともと永田らによって巡回セールスマン問題(TSP)の交叉方法として提案されたが、本

研究で参考にしたNagataらの論文でCVRP用に拡張されている。交叉方法には一点交叉や一様交叉等、様々な問題に汎用的に利用できるものがあるが、山村らは問題固有の形質、すなわち形質遺伝性を考慮した交叉設計を行うことが重要だと指摘している。TSPにおける良い形質とは「ルートを構成するエッジまたは連続するエッジと考えるのが自然である」とし、TSPに対する新しい交叉方法としてEAXを提案した。EAXは「良い近似解同士には何らかの類似性が存在する」という、近接最適性の原理(Proximate Optimality Problem: POP)に基づいた交叉方法である。EAXを用いたGAは、TSPに対する最も優れた近似解法の一つとなっている。CVRP用に拡張されたEAXにおいても、既知最良解を求めることができている。詳しい処理についてはkai-master-thesis.pdf、永田らの論文、または本プログラムを参照のこと。

メメティックアルゴリズムの実装

CVRPをMemeticアルゴリズムを用いて解くプログラム

参考文献

参考にしたMAの擬似コード

> Yuichi Nagata and Olli Br¨aysy. Edge assembly-based memetic algorithm for the capaci- tated vehicle routing problem. Networks, Vol. 54, No. 4, pp. 205–215, 2009.

> Yuichi Nagata. Edge Assembly Crossover for the Capacitated Vehicle Routing Problem, pp. 142–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

> 永田裕一, 小林重信, 東条敏. 効果的な局所探索制限によるmemetic algorithmの高速化. 人工知能学会論文誌, Vol. 25, No. 2, pp. 299–310, 2010.

参考にしたEdge Assembly Crossoverが説明されている論文

> 永田裕一, 小林重信. 巡回セールスマン問題に対する交叉: 枝組み立て交叉の提案と評価. 人工知能学会誌, Vol. 14, No. 5, pp. 848–859, 1999.

遺伝的アルゴリズムを実装する上で枠組みとして参考にしたWebページ。OneMax問題を対象に遺伝的アルゴリズムを使用されている。本研究ではこちらの枠組みのみ使用し、内部の実装は全て論文の擬似コードから実装。

> https://qiita.com/Azunyan1111/items/975c67129d99de33dc21

MAのパラメータに関する記述が載っている論文

> Jakub Nalepa and Miroslaw Blocho. Adaptive memetic algorithm for minimizing distance in the vehicle routing problem with time windows. Soft Computing, Vol. 20, No. 6, pp. 2309–2327, 2016.

近接最適性の原理について

> 新妻大地, 安田恵一郎. 近接最適性の原理に基づく多点探索型tabu search. 日本知能情報ファジィ学会 ファジィ システム シンポジウム 講演論文集, Vol. 21, pp. 81–81, 2005.

環境構築

本プログラムはPythonで実装した。Pythonは以下のサイトからダウンロードする。

- https://www.python.org/downloads/

PCはmacを使用して実装しているためmacでの環境構築になるが、ダウンロードしたdmgファイルを実行し
Pythonをインストールする。

Homebrewとpyenvを利用してインストールする方法もある。pyenvではPythonのバージョン切り替え等ができて便利。
以下のサイトを参考にpyenvを導入。なお、本プログラムはPython3系に則った実装である。

- https://qiita.com/ms-rock/items/6e4498a5963f3d9c4a67

実装に使用したmacのOSとPythonのバージョンは以下の通りである。

- macOS：High Sierra
- バージョン：10.13.3
- CPU：2.4GHZ Intel Core i5
- メモリ：8GB 1600 MHz DDR3

```
$ python --version
Python 3.6.1
```

必要なライブラリは全てPythonに搭載されている`pip`コマンドでインストールした。まず`pip`コマンドを最新にアップグレードする。

```
$ pip install --upgrade pip
```

続いて、以下のライブラリをインストールする。

- matplotlib
- networkx
- numpy
- pandas

```
pip install matplotlib networkx numpy pandas
```

本プログラムで用いたそれぞれのライブラリのバージョンは以下の通りである。

```
$ pip freeze
matplotlib==2.0.2
networkx==1.11
numpy==1.13.3
pandas==0.20.2
```

各ライブラリをプログラムで使用する場合、`import`して用いる。以下のような略称で定義することが一般的らしい。

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import networkx as nx
```

ファイル

- main.py : メメティックアルゴリズムを実行させるプログラム
- GeneticAlgorithm.py : 遺伝子情報とその遺伝子の評価値を格納するクラス

主にMAに関するプログラムは上記の2つで、下は実装にあたり補助的に作ったものやテスト用のプログラムである。

- EdgeAssembly.py：EAXの実装テスト用ファイル
- graphPlot.py：./outputフォルダ内にあるcsvファイル内容をプロットする
- CVRPwithMIP：CVRPの最適解を求めるプログラム
- benchmark.py：ベンチマーク問題の顧客情報を元に顧客とデポをプロットするプログラム
- initialize.py：kai-master-thesis.pdfにおける予備実験にも使用したプログラム

    - セービング法によって初期集団生成
    - ペナルティ関数に基いて局所探索(ルート数減少を許容)
    - パラメータαの設定によっては実行不可能解が生成される
    - 実行不可能解だった場合に修正操作[modification()]によって実行可能解へ修正する
    - 修正操作中の局所探索ではペナルティ関数のペナルティ項が減少する解が見つかり次第その解へ更新することを繰り返す(ルート数が減少する更新は禁止)
    - 修正操作をしても実行可能解が見つからない場合もある
    - インクリメントしているパラメータαごとに、実行可能であったか(success or error)、ルート数(route_num)をデータフレームに格納
    - csvファイルに書き出し
- Local.py：局所探索の関数がうまく動作しているかの確認用ファイル
- routePlot.py：main.pyやEdgeAssembly.py等で生成したルートの3次元リストを元に、視覚的にどんなルートだったかを後で確認できるように作ったファイル
- txt2csv.py：Webサイトから取ってきたベンチマークファイルが.txtファイルだったため、本プログラムで使用しやすいような.csvファイルへ加工するプログラム
- txt2csv2.csv：上のプログラムとほぼ同じであるが、実験に使用したベンチマークに対してはこちらを使用した。ベンチマークから.csvファイルを作成し、ヘッダに(x, y, d)を追加。プログラム中でデータフレーム型にそのまま代入できる。
- Golden.py、Taillard.py：txt2csv2.pyとほぼ同じ。実験に使用していないが、Goldenらのベンチマークと、Taillardらのベンチマークを加工するのに微妙な変更点があったため作成した。

実行

MemeticAlgorithmプログラムを実行する

```
$ python main.py
```

交叉EAXのテストプログラムを実行する

```
$ python EdgeAssembly.py
```

予備実験用の前処理プログラムを実行する

```
$ python initialize.py
```

main.pyの内部処理の説明

ライブラリ等

```
import GeneticAlgorithm as ga
import random
from decimal import Decimal
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import sys, time
import copy
import itertools
import math
import csv
```

- numpy、pandas等のライブラリをインストールしておく
- networkxはルートを視覚的に表現するためのグラフをプロットするために使用。matplotlibと組み合わせて使用する。
- itertoolsは2つのリスト内の要素の組合せでループが回せるため非常に便利。
- gaクラスでは各個体のエッジリスト、各個体の(目的関数における)評価値、各個体の距離を格納する。
    - 以下にGeneticAlgorithm.py中のソースコードを示す。

```
class genom:

    genom_list = None
    evaluation = None
    distance = None

    def __init__(self, genom_list, evaluation, distance):
        self.genom_list = genom_list
        self.evaluation = evaluation
        self.distance = distance


    def getGenom(self):
        return self.genom_list

    def getEvaluation(self):
        return self.evaluation

    def getDistance(self):
        return self.distance

    def setGenom(self, genom_list):
```

```
        self.genom_list = genom_list

    def setEvaluation(self, evaluation):
        self.evaluation = evaluation

    def setDistance(self, distance):
        self.distance = distance
```

設定するパラメータ

```
# 遺伝子集団の長さ
MAX_GENOM_LIST = 30
# 各両親から生成される子個体の数
MAX_CHILDREN = 10
# 繰り返す世代数
MAX_GENERATION = 100
# 使用できる車両数
m = 8
# 車両の最大積載量
CAPACITY = 160
# セービング値の効果をコントロールする係数
LAMBDA = 1
# N_near()関数で，どこまで近くのノードに局所探索するか
NEAR = 10
# penaltyFunction()で，容量制約違反に課すペナルティの係数
ALPHA = 0.003
```

変更するべきパラメータは以下の通りである。

- MAX_GENOM_LIST：多点探索における集団の数を定義
- MAX_CHILDREN：1組の親ペアから生成する子個体の数を定義
- MAX_GENERATION：繰り返す世代数を定義
- m：出力する解に求めるルート数
- CAPACITY：トラックの容量(制約)
- LAMBDA：セービング法で構築されるルートの形状パラメータ
    - 有効範囲は0≦LAMBDA≦2 とされているが、1.0で問題ないと思われる
- NEAR：局所探索においてエッジをつなぎ替える相手ノードを何番目まで近くのノードから選択するかを定義

以降の説明内に出てくるパラメータにも変更箇所がある。

- S_OP：ルート分割において、どの方法でルートを分割するかを定義
- filename：対象とするcsvファイル名
- saveDirectory：結果を保存するフォルダまでのパス

また、`Local()`、`penaltyFunction()`、`graphPlot()`の引数にも以下の

- f_option：解をどのように評価するか
- reduce_route：ルート数が減る解への更新を許容するか
- option：解をどのように評価するか(f_optionの設定に連動)
- isFirst：現在のグラフを描画し次のグラフを新しいウィンドウで描画するか
- isLast：連続描画における最後の描画か
- title：グラフのタイトル

といった設定するパラメータがあるが、これはプログラムの各所によって使い分けているため詳しくは以降の説明を参照のこと。

これらのパラメータを設定後、

```
$ python main.py
```

により実行する。

メイン部

順に説明します。

本研究で使用したChristofidesらのベンチマークには14個の問題があるが、そのうち6~10,13,14は1~5,11,12と同じ顧客配置である。しかし、6~10,13,14は容量制約に加え期間制約が設定されており、これはDVRPと呼ばれる問題となり本研究では対象としない。よって以下のリストの`skip = []`部分に対象の問題番号を入れておき、ループ部で`continue`によってスキップしている。なお、`Capa = []`と`Vehcle = []`はそれぞれベンチマーク問題に設定されているトラック容量と既知最良解のルート数である。問題番号に対応させてパラメータを変更する。

```
Capa = [160, 140, 200, 200, 200, 160, 140, 200, 200, 200, 200, 200, 200, 200]
Vehicle = [5, 10, 8, 12, 17, 6, 11, 9, 14, 18, 7, 10, 11, 11]
skip = [6, 7, 8, 9, 10, 13, 14]
```

後述するルート分割(`routeSplit()`)において、どの方法でルートを分割するかを設定する。

```
S_OP = 3
```

```
# 0: ランダム
# 1: エッジ数
# 2: 距離
# 3: 需要量
# 4: All
if S_OP == 0:
    method = "ランダム"
elif S_OP == 1:
    method = "エッジ数"
elif S_OP == 2:
    method = "距離"
elif S_OP == 3:
    method = "需要量"
elif S_OP == 4:
    method = "All"
```

ループによってskipリストに入っていない問題番号のベンチマークを処理していく。以降の変数`No`は問題番号に対応する。

```
for No in range(1, 15):
    if No in skip:
        continue
```

各ファイルに対する処理開始

`filename`には現在`vrpnc + str(No)`となっているが、ここには対象とするcsvファイル名を入れればよい。今回扱っているファイル名はvrpnc1~vrpnc14である(うち7つはskip)。`saveDirectory`には今回の実行で得られる結果を保存するフォルダを指定する。

変数`df`にベンチマークの顧客情報を代入する。`len(df.index)`とすることで顧客数が得られ、それを変数`num_shelter`に格納。

またここで、今回の問題におけるトラック容量とルート数をパラメータとして設定する。`m = Vehilce[No-1] + 5`となっているが、これは既知最良界のルート数+5を指定している。

```
filename = "vrpnc" + str(No)
saveDirectory = "./output/MAdemand/plus5/"
# 避難所情報のデータフレームを生成する
# 引数[0]:ファイルパス, [1]:ファイル名
df = createDataFrame("./csv/Christ/", filename)
num_shelter = len(df.index)

CAPACITY = Capa[No-1]
```

```
m = Vehicle[No-1] + 5
```

以下が`createDataFrame()`内の処理である。

```
"""
@INPUT:
    filepath: 読み出すファイルパス
    data_name: 読み出すファイル名
@OUTPUT:
    読み出したデータのデータフレーム
"""

def createDataFrame(filepath, data_name):
    input_path = filepath + data_name + ".csv"
    return pd.read_csv(input_path)
```

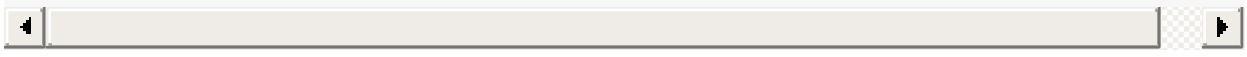
`saveDirectory`に指定したフォルダ内にconfigというファイルを作成しておき、今回の実行ではどのような設定だったかを保存しておくようにしている。保存しておく設定は以下の通りである。なお、プログラム終了時に実行時間等も保存するためにまだcloseしない。

```
f = open(saveDirectory + "config/" + "Problem" + str(No) + "_config.csv", "w")
writer = csv.writer(f, lineterminator="\n")
paramArray = [["ファイル名", filename + ".csv"],
       ["顧客数", num_shelter-1],
       ["繰り返す世代数", MAX_GENERATION],
       ["集団数", MAX_GENOM_LIST],
       ["生成子数", MAX_CHILDREN],
       ["トラック台数(m+3)", m],
       ["トラック容量", CAPACITY],
       ["ルート分割方法", method]]

writer.writerows(paramArray)
paramArray = []
```

遺伝的アルゴリズムにおける世代ごとの評価値や移動コストを後でグラフで出力できるように、データフレームに保存しておく。以下はそのヘッダ部を作成している。

```
result_df = pd.DataFrame(index=[], columns=['世代', 'ペナルティ関数値', '総移動コスト'])
```



避難所情報が入ったデータフレームから各顧客間の移動コスト行列を作成する。`cost`リストは書き換えることがないため、全ての関数から参照できるようにする。

```
# 各避難所間の移動コスト行列を生成する
# 2次元配列costで保持
cost = createCostMatrix(num_shelter)
```

以下は`createCostMatrix()`の説明である。

```
"""
避難所の距離に基づいたコスト行列を返す
@INPUT:
    num_shelter : 避難所数
@OUTPUT:
    arr : コスト行列
"""
def createCostMatrix(num_shelter):
    dis = []
    arr = np.empty((0, num_shelter), float) #小数点以下を加える→float型

    for i in range(num_shelter):
        for j in range(num_shelter):
            x_crd = df.ix[j].x - df.ix[i].x
            y_crd = df.ix[j].y - df.ix[i].y

            dis.append(round(np.sqrt(np.power(x_crd, 2) + np.power(y_crd, 2)), 2))
            if j == num_shelter - 1:
                arr = np.append(arr, np.array([dis]), axis=0)
                dis = []
    np.savetxt("./output/cost.csv", arr, delimiter=',', fmt='%.2f')
    return arr
```

各世代における集団を保管しておくリスト`current_generation_individual_group = []`を作成する。変数`indi_count`は初期解を生成する度にインクリメントする。実行時間を求めるために、ループに入る前の時間を`start`に保存する。`m_time`はMAの最後に行う局所探索(修論の局所探索3の説明参照)にかかった時間を保存するために使用する。ここからwhile文に入り、終了条件を満たすまでループ処理を続ける。

```
# 第一世代の個体集団を生成
current_generation_individual_group = []
indi_count = 0

start = time.time()
```

```
m_time = 0

while(True):
```

セービング法によって初期解を生成する。セービング法は初期会を生成する構築法の一つである。

```
"""
セービング法により初期解を生成
"""
sa_route, distance = savingMethod(num_shelter, cost)
```

以下は`savingMethod()`の説明である。

```
"""
セービング法によりヒューリスティックな解を生成する
@INPUT:
    num_shelter : 避難所数
    cost : 避難所間のコスト行列
@OUTPUT:
    route：構築されたルートの２次元リスト
    drt：移動距離

"""
def savingMethod(num_shelter, cost):
```

個体(解)の表現方法について

本プログラムでは個体をエッジのリストとして表現しているが、2種類の表現方法を用いている。1つは2次元のリストで、全てが各顧客を結ぶエッジを表す。

もう一方は3次元リストであり、こちらは先述した2次元リスト内のエッジ情報を見て顧客を辿ることで得られる閉路毎にまとめたものである。これにより、各顧客同士が同じルート内に含まれる顧客かどうかの判別やルート数が求めやすくなる。以下に例を示す。

```
path = [
    [[14, 0], [25, 14], [13, 25], [41, 13], [40, 41], [19, 40], [42, 19], [44, 42],
    [[18, 0], [4, 18], [47, 4], [0, 47]],
    [[27, 0], [48, 27], [48, 23], [23, 7], [7, 43], [43, 24], [24, 6], [0, 6]],
    [[34, 0], [30, 34], [39, 30], [33, 39], [45, 33], [15, 45], [0, 15]],
```

```
    [[46, 0], [11, 46], [2, 11], [16, 2], [38, 16], [5, 38], [12, 5], [0, 12]],
    [[49, 0], [10, 49], [9, 10], [50, 9], [29, 50], [21, 29], [0, 21]],
    [[32, 1], [1, 22], [0, 22], [0, 20], [35, 20], [36, 35], [36, 3], [3, 28], [31,
    ]
```

プログラム中に用いている変数名に `route` に近いものや `path` に近いものが多数出現するが、本プログラムにおいて、2次元リストによって表現しているものを `route`、3次元リストによって表現しているものを `path` と定義している。関数内の処理によってそれぞれ利用しやすい場合があるために混在している。なお、 `route` → `path`、 `path` → `route` に変換する関数を用意している。

- routeToPath(route)
- pathToRoute(path)

以下は `routeToPath` と `pathToRoute` の説明である。

```
"""
2次元のエッジリストから，各閉路毎にエッジを持つ3次元リストに変換する
@INPUT:
    route: エッジの２次元リスト
@OUTPUT:
    Path: ルート情報を含むエッジの3次元リスト
"""
def routeToPath(pre_route):
```

```
"""
3次元のエッジリストから，2次元リストに変換する
@INPUT:
    path: ルート情報を含むエッジの3次元リスト
@OUTPUT:
    route: エッジの2次元リスト
"""
def pathToRoute(path):
    EdgeList = []
    for edge in path:
        for j in edge:
            EdgeList.append(j)
    return EdgeList
```

`savingMethod` で構築した初期解は `sa_route` に保存されているが、上述した `route`、 `path` のいずれの表現にもなっていない。よって以下の関数で3次元リストの初期解に変換する。ルート数は `len()` で求まる。

```
path = savingRoute(sa_route) # 3次元解
print("セービング法のルート数:{}".format(len(path)))
```

以下は`savingRoute()`の説明である。

```
"""
セービング法で構築されたルートに,
デポを出発してデポに帰るようにエッジを辿る
3次元配列を生成する.
@INPUT:
    route:セービング法で構築されたルート(0は表示されていない)
@OUTPUT:
    Path:ルートを辿るエッジの3次元配列(0も表示)
"""
def savingRoute(route):
```

セービング法で得られた初期解のルート数を、パラメータ`Vehicle[]`で設定したルート数に固定するためにルートを分割する。ルートを分割する方法はプログラムの最初に設定した`S_OP`によって決定する。

```
path = routeSplit(path) # ルート数をmに固定
print("分割後ルート数:{}".format(len(path)))
```

以下は`routeSplit`の説明である。

```
"""
経路を分割することでルート数を増やす関数
セービング法によって生成された初期解のうち,
もっとも顧客数の多いルートを選び真ん中で分割する
@INPUT:
    path：解の3次元リスト
    (m): 固定したいルート数
@OUTPUT:
    path
"""
def routeSplit(path):
```

本研究では初期解と交叉によって得られた子解に対してはランダムに選んだ1つのノードに対してのみ局所探索する(修論参照)。よって顧客をランダムに並べたリストを作る。

random.choice()で1つ選んでも良いが、1つ目のノードの局所探索で改善解が見つからなかった場合次のノードを対象に探索するため顧客をランダムに並べる方法を取っている。

なお、局所探索によって得られた解中にデポを含まない部分巡回路等が出来てしまった場合(Falseが返ってくる)に備えて、元の`path`情報を`prePath`にコピーしておき、`path == False`だった場合に`prePath`で上書きする。`False`が返って来ず、`prePath`と内容が変わっていた場合局所探索が成功したとして、ループから抜ける。

```
# 局所探索用のランダムな並びを生成
random_order = [i for i in range(1, num_shelter)]
random.shuffle(random_order)

"""
局所探索
"""
print("")
# first改善山登り法による局所探索法によって解を改善
for n, i in enumerate(random_order):
    prePath = copy.deepcopy(path)
    local_route = Local(i, path, f_option=1, reduce_route=0)
    path = routeToPath(local_route)
    if path == False:
        path = copy.deepcopy(prePath)
    if path != prePath:
        print("局所探索成功")
        break
```

以下は`Local()`の説明である。

```
"""
近傍操作関数
@INPUT:
    v: 近傍操作対象ノード
    path: 解の3次元リスト
    f_option: 解をどのように評価するか
        2: 容量超過
        1: ペナルティ関数による評価
        0: 総移動コストのみの評価
    reduce_route: ルート数が減る解への更新を許容するか
        1: 許容する
        0: 許容しない
"""
def Local(v, path, f_option, reduce_route):
```

また、本研究では局所探索で対象とするノードとエッジをつなぎ替える相手は地理的に近い10ノードから近い順に選択するとしている。そこで`Local()`内で呼び出している、対象のノード

に対して地理的に近い10ノードを返す関数`N_near()`を実装している。

以下は`N_near()`の説明である。

```
"""
あるノードから近いノード集合を返す
引数nodeで与えたノードから，近くにあるノードをnear番目まで選んだ集合
@INPUT:
    node:ノード
    near:近くのノードをいくつ探すか
@OUTPUT:
    near_cost:近くのノード集合
"""
def N_near(node, near):
    near_cost = np.empty((0, 2), int)
    for i, c in enumerate(cost[int(node)][:]):
        if i == 0:
            continue
        near_cost = np.append(near_cost, np.array([[i, c]]), axis=0)
    near_cost = near_cost[near_cost[:, 1].argsort()] # nodeに近い順にソート
    # print(near_cost)

    # print(near_cost[1:near+1, 0])
    return near_cost[1:near+1, 0]
```

ここまでの処理で、ルート分割も踏まえて設定したルート数の解になっているはずであるが、もしそうでなかった場合に備えてルート数が異なる場合はgaクラスに個体を保存せずにそれ以降の処理を`continue`する。

```
if len(path) != m:
    print("ルート数が異なる")
    continue
```

設定したルート数になっていた初期解の、移動距離と評価値を保存する。引数`option`に与える数字によって、返ってくる値が変化する。

```
ditance = penaltyFunction(route, option=0)
evaluation = penaltyFunction(route, option=1)
```

関数`penaltyFunction()`は以下のようになっており、`option`によって返ってくる`F_p`の内容が変わる。

```
"""
ペナルティ関数による評価を行う
@INPUT:
    route: 解の２次元リスト
    option: どのように評価するか
        2: ペナルティ項のみ
        1: ペナルティ関数による評価
        0: 総移動コストのみの評価
@OUTPUT:
    F_p: 関数による評価値
"""
def penaltyFunction(route, option):
```

---

gaクラスに解の情報を保存する。`print()`で内容を確認。

```
# GAクラスに解，評価値，距離を保存
current_generation_individual_group.append(ga.genom(route, evaluation, distance))

print("個体【{}】:{}".format(indi_count, current_generation_individual_group[indi_cou
print("評価{}".format(current_generation_individual_group[indi_count].getEvaluation(
print("距離{}".format(current_generation_individual_group[indi_count].getDistance())
```

いくつ初期解が生成されたかを表す`indi_count`をインクリメントする。`sys.stdout.write()`によってターミナルに表示すると、次の文章が前の文章の上に上書きされるので結果が流れていかずいくつ生成されたかが見やすくて便利。パラメータ`MAX_GENOM_LIST`と同数の初期解が生成されたら終了し、これが初期集団となる。

```
indi_count += 1
sys.stdout.write("\r%d個初期解生成" % indi_count)
sys.stdout.flush()
time.sleep(0.01)
if indi_count == MAX_GENOM_LIST:
    print("")
    break
```

終了条件用の変数を初期化。

```
monotonous = current_generation_individual_group[0].getDistance()
```

```
mono_count = 0
```

初期集団のそれぞれの解の評価値をリスト`fits`に保存しリスト化し、最小値のインデックスを求める(評価値は小さいほど良い)。最小値のインデックス`min_idx`によって、集団中の最良個体の評価値、移動距離を取り出し、データフレームの第1世代として保存。以降、世代を経る度に`result_df = result_df.append(series, ignore_index = True)`によってデータフレームを更新していくことになる。

```
# 今世代の個体適用度を配列化する
fits = [f.getEvaluation() for f in current_generation_individual_group]
# print("fits:{}".format(fits))
min_idx = fits.index(min(fits)) # 最小値を求める
# print("min_idx:{}".format(min_idx))
min_ = current_generation_individual_group[int(min_idx)].getDistance()
min_Eval = current_generation_individual_group[int(min_idx)].getEvaluation()
print("====第{}世代====".format(int(1)))
print("最も優れた個体の評価値:{}".format(min_Eval))
series = pd.Series([int(1), min_Eval, min_], index=result_df.columns)
result_df = result_df.append(series, ignore_index = True)
```

ここまでが第1世代の処理となりこの後は各世代における処理である。

世代始まり

各世代の最初の処理。世代番号は`count_`に対応する。世代の初めに顧客番号をランダムに並べ替える。

```
for count_ in range(2, MAX_GENERATION + 1):
    #集団中の個体をランダムな順列に並べる
    order = setRondomOrder()
```

以下は`setRondomOrder()`の説明である。

```
"""
集団中の個体をランダムな順序に並べるための個体番号順列を生成する
@INPUT:
    None
@OUTPUT:
    ランダムに並べた遺伝子集団サイズ分の順列
"""
```

```
def setRondomOrder():
    #random_order_list = []
    # 集団サイズ分の順列リストを作り，シャッフルする
    random_order = [i for i in range(MAX_GENOM_LIST)]
    random.shuffle(random_order)
    return(random_order)
```

ランダムに並んだ集団の最初から順に2つずつを親A、親Bとして選ぶ。末端では折り返し、全ての個体が親Aおよび親Bとして選ばれることになる。また、交叉後の子個体を格納するリストを準備する。

```
for i in range(MAX_GENOM_LIST):
    # 集団中の全ての個体が丁度一度ずつ親P_Aとして選択される
    if i < MAX_GENOM_LIST -1:
        P_A = current_generation_individual_group[order[i]]
        P_B = current_generation_individual_group[order[i+1]]
    else:
        P_A = current_generation_individual_group[order[i]]
        P_B = current_generation_individual_group[order[0]]

    c = [] # 子解を代入するリスト
    c_cost = [] # 子の移動距離を代入するリスト
    c_eval = [] # 子のペナルティ関数値を代入するリスト
```

交叉EAXの前準備をする。論文におけるEAXの処理ステップの1~3までを実行し、AB-cycleを変数`ABc`に保存する。

```
"""
交叉：EAX
"""
ABc = preEAX(P_A, P_B)
```

以下は`preEAX()`の説明である。

```
"""
EAXのステップ1と2を処理する関数
ステップ1: G_AB集合を生成する
ステップ2: AB-cycleによる閉路を生成する
@INPUT:
    P_A : 親AのgenomClass
    P_B : 親BのgenomClass
    ※それぞれ.getGenom()でエッジ情報を取得
```

```
@OUTPUT:
    C: AB-cycleのリスト
"""
def preEAX(P_A, P_B):
```

EAX後に行う局所探索用の顧客をランダムに並べたリストの準備と、EAXを処理する。引数として`P_A`、`P_B`、`ABc`を渡して子解を得る。

```
# AB_cycleが構築できた場合，子解を生成する
if ABc != False:
    # EAXのステップ3~5
    for j in range(MAX_CHILDREN):
        # 局所探索用のランダムな並びを生成
        random_order = [k for k in range(1, num_shelter)]
        random.shuffle(random_order)

        child = edgeAssemblyCrossover(P_A, P_B, ABc)
```

以下は`edgeAssemblyCrossover()`の説明である。

```
"""
交叉EAXによって子個体を生成する。
@INPUT:
    P_A : 親AのgenomClass
    P_B : 親BのgenomClass
    ※それぞれ.getGenom()でエッジ情報を取得
@OUTPUT:
    child：交叉によって得られた子個体
"""
def edgeAssemblyCrossover(P_A, P_B, ABc):
```

なお、`edgeAssemblyCrossover()`によって得られた子個体はデポを含まない部分巡回路(`intermediate`)を生成する可能性がある。その場合、`edgeAssemblyCrossover()`内部で`EAXstep5()`を呼び出し、ルートのマージ操作を処理する。

以下は`EAXstep5()`の説明である。

```
"""
EAXのステップ5を処理する
中間個体に部分巡回路が含まれる場合に，部分巡回路をランダムな順番で選択し，
m個のルートのどれかに結合することでm個のルートからなる子を得る．
@INPUT:
    intermediate：中間個体のエッジ集合
```

```
    @OUTPUT:
        child：子個体
    """
    def EAXstep5(intermediate):
```

子解を3次元リスト`path`へ変換する。ペナルティ関数が減少する解を局所探索`Local()`によって探索し、見つかり次第更新する。いくつの顧客に対して局所探索するかを変更できるように`local_count`変数を用意しているが、今回は1つの顧客に対してのみ局所探索を適用している。

```
    path = routeToPath(child)
    """
    EAX後の局所探索
    f_option:
        0→距離のみの評価
        1→ペナルティ関数による評価
        2→ペナルティ項が0以下になるように
    """
    local_count = 0
    for n, eax in enumerate(random_order):
        prePath = copy.deepcopy(path)
        local_route = Local(eax, path, f_option=1, reduce_route=0)
        path = routeToPath(local_route)
        if path == False:
            path = copy.deepcopy(prePath)
        if path != prePath:
            local_count += 1
            if local_count >= 1:
                # print("局所探索成功")
                break
```

EAXで得られた子解の２次元リスト、移動距離、評価値をそれぞれ前もって準備していたリストに保存する。

```
    c.append(pathToRoute(path))
    # c.append(route)
    c_cost.append(penaltyFunction(c[j], option=0))
    c_eval.append(penaltyFunction(c[j], option=1))
```

生成された子個体の中で最も評価値の低い(適合度が高いことを意味する)個体が、親Aよりも優れていた場合、集団中の親Aのインデックスを指定して入れ替える(家族内淘汰)。また、個体の

各情報もgaクラスに保存する。

```
# 現在の親ペアから生成された子の中で一番コストの低い個体が親Aよりも
# 環境に適合している場合，親Aのインデックスを指定して入れ替える
if min(c_eval) < P_A.getEvaluation():
    min_idx = c_eval.index(min(c_eval))
    # print("子の評価値:{}".format(c_eval))
    # print("min_idx:{}".format(min_idx))
    # print("子の中で最小のコスト:{}".format(c_eval[min_idx]))
    current_generation_individual_group[order[i]].setGenom(c[min_idx])
    current_generation_individual_group[order[i]].setEvaluation(c_eval[min_idx])
    current_generation_individual_group[order[i]].setDistance(c_cost[min_idx])
```

集団中の全ての個体を親とした交叉と家族内淘汰が終了したため、今世代の結果を確認する。今世代の集団の評価値をリスト化し、最小値のインデックスを`min_idx`とする。この個体が今世代の最良個体となるため、インデックス`min_idx`を指定して移動距離と評価値を取り出し、変数`min_`、`min_Eval`に保存する。

本プログラムでは終了条件として各世代の最良個体の評価値が10世代に渡って不変だった場合としているため、`min_Eval`が`monotonous`と等しかった場合`mono_count`をインクリメントする。変化があった(更新された)場合、`mono_count = 0`とし、`monotonous = min_Eval`として更新する。

```
# 今世代の個体適用度を配列化する
fits = [f.getEvaluation() for f in current_generation_individual_group]
# print("fits:{}".format(fits))
min_idx = fits.index(min(fits)) # 最小値を求める
# print("min_idx:{}".format(min_idx))
min_ = current_generation_individual_group[int(min_idx)].getDistance()
min_Eval = current_generation_individual_group[int(min_idx)].getEvaluation()

if monotonous == min_Eval:
    mono_count += 1
else:
    mono_count = 0
    monotonous = min_Eval
```

今世代の結果を使ってデータフレームを更新する。また結果として今世代の最良個体の評価値を`print()`で確認。

10世代以上評価値に変化がなかった場合、MAのメインループを抜け最終処理へ、終了条件が満たされなければ次の世代`count_ + 1`へ。

```
# データフレームの行を世代によって更新
series = pd.Series([int(count_), min_Eval, min_], index=result_df.columns)
result_df = result_df.append(series, ignore_index = True)

print("====第{}世代====".format(int(count_)))
print("最も優れた個体の評価値:{}".format(min_Eval))

# 10世代以上適合度が変わらなかった場合
if mono_count > 10:
    break
```

世代終わり

対象ベンチマークファイルに対するMAの後処理

プログラムのループ中の実行時間を`elapsed_time`へ、最終世代の最良個体のインデックスを`min_idx`へ保存する。`min_idx`を指定し、最良個体のgaクラス情報を変数`Best`へ保存する。

```
elapsed_time = time.time() - start
min_idx = fits.index(min(fits))
# 最良個体の情報
Best = current_generation_individual_group[min_idx]
print("計算時間:{}".format(elapsed_time))
print(result_df)
print("最も優れた個体:{}".format(Best.getGenom()))
print("最も優れた個体の移動距離:{}".format(Best.getDistance()))
print("最も優れた個体の評価値:{}".format(Best.getEvaluation()))
```

最良個体の3次元リストを`BestPath`とする。`BestPath`の移動距離、評価値、ルート数の各情報と、計算時間をcsvファイルに書き込む。

```
Bestpath = copy.deepcopy(routeToPath(Best.getGenom()))
print(Bestpath)
print("ルート数:{}".format(len(Bestpath)))

paramArray = [["移動距離", Best.getDistance()],
       ["評価値", Best.getEvaluation()],
       ["ルート数", len(Bestpath)],
       ["計算時間", round(elapsed_time, 2)]]
writer.writerows(paramArray)
paramArray = []
```

本プログラムではMAによる探索が終わって得られた最良個体に対して、最後に局所探索3(修論参照)を適用する。この局所探索は初期会や交叉後の子個体に対して適用した局所探索と比較して以下の点で異なる。

- 全ての顧客に対して適用する
- 移動距離に基いて探索する(移動距離を短くする目的)
- 局所探索を適用する顧客とエッジをつなぎ替える相手は、同じルート内の顧客から選ぶ
- 対象の顧客の近傍中で最も移動距離が短くなる解で更新する

また最後の局所探索にかかった時間を`end_time`として保存する。

```
m_start = time.time()
for n, i in enumerate(random_order):
    prePath = copy.deepcopy(Bestpath)
    local_route = Neighborhoods(i, Bestpath, f_option=0, reduce_route=0, first=0)
    Bestpath = routeToPath(local_route)
    if Bestpath == False:
        Bestpath = copy.deepcopy(prePath)
    # graphPlot(local_route, isFirst=0, isLast=0, title="local search")
end_time = time.time() - m_start
```

ここまでで今回の実行で最良となる解が生成されている。`checkCapacity()`では各ルート内の需要量がトラック容量を違反していないかを確認している。違反しているルートがあった場合`r`にはそのルートの3次元リストにおけるインデックスが、`excess`には制約違反量のリストが返ってくる。

局所探索後の解の２次元リスト、移動距離、評価値をそれぞれ`Bestroute`、`distance`、`evaluation`に保存し、もしも制約違反があった場合は`over`にペナルティ項の値が入る。

```
Bestroute = pathToRoute(Bestpath)
r, excess = checkCapacity(Bestpath)
print("ルート毎の需要:{}".format(excess))
print("局所探索後")
distance = penaltyFunction(Bestroute, 0)
evaluation = penaltyFunction(Bestroute, 1)
over = penaltyFunction(Bestroute, 2)

print("距離:{}, 評価値:{}, 超過:{}".format(distance, evaluation, over))
```

以下は`checkCapacity()`の説明である。modification()中に用いるとあるが、EAX中や最終的な解等に違反がないかを確認するために各場所で用いている。

```
"""
modification()中に用いる
入力された3次元リストを元に，トラックの容量超過をしているルートの
インデックスとルート毎の総需要量のリストを返す
@INPUT:
    path：解の3次元リスト
@OUTPUT:
    r：制約違反ルートのインデックスが入ったリスト
    excess：ルート毎の需要量のリスト
        なお、excess内の総和が0を超える場合現在のルート数では実行可能解が
        作れないことを表すため、その時はFalseを返す
"""
def checkCapacity(path):
```

それぞれの結果をcsvファイルに書き込む。

```
writer.writerow(["修正時間", round(end_time, 2)])
writer.writerow(["ルート毎の需要", " "])
writer.writerow(excess)
writer.writerow(["局所探索後距離", distance])
writer.writerow(["評価値", evaluation])
writer.writerow(["超過", over])

writer.writerow(["最優個体", " "])
writer.writerow(Best.getGenom())
writer.writerow(["最優ルート", " "])
writer.writerows(Bestpath)

result_df.to_csv(saveDirectory + "Problem" + str(No) + "_result.csv", index=False)
```

MAの探索によって得られた解とその後局所探索によって修正を加えた解のルートを視覚的に確認するためにグラフとしてプロットする。

```
graphPlot(current_generation_individual_group[min_idx].getGenom(), \
          isFirst=1, isLast=0, title="Problem" + str(No) + " Last Generation")
graphPlot(Bestroute, isFirst=1, isLast=0, title="Problem" + str(No) + " Best Route"
f.close()
```

以下は`graphPlot()`の説明である。

```
"""
グラフをプロットする
@INPUT:
    edgeList：解の２次元リスト
    isFirst：
        0：0を指定した場合特に影響なし
        1：現在のグラフをプロットし、さらに新しいウィンドウで次のグラフをプロットできる
          (いくつかのグラフを並べて表示したい時に使う)
    isLast：
        0：isFirstが0の時、isLastも0にすることで連続プロットができる
          同じウィンドウ上で毎回上書きしながらグラフを更新する
          (局所探索のような次々とエッジ情報が切り替わる処理の様子を観察
          したい時に用いる)
        1：グラフを表示してプログラムが一時停止する
          グラフのウィンドウを閉じるとプログラムの処理が続行される
    title：グラフに付けたいタイトルを指定する

@OUTPUT：
    グラフ描画

なお、処理中にplotDepot()が呼び出され、デポの色や大きさのみを変更可能
"""
def graphPlot(edgeList, isFirst, isLast, title):
```

次のファイルへ

ここまでで1つのファイルに対する処理が終わったため、次のファイル`No + 1`の処理へ移る。

おわりに

本プログラムにおける局所探索は、参考にした論文等に比べると非常に時間がかかるようである。参考にした論文で使用されている言語がC++なのに対し本プログラムがPythonであることや、内部処理に冗長な部分があることが原因であると考えられるので、そのあたりを直すことで処理速度を改善したい。特に局所探索は同じような処理をそれぞれの近傍に対して毎回記述しているので関数化するのが望ましい(当時は時間の関係上そこまでできなかった)。ペナルティ関数における解の評価方法や局所探索の近傍や適用方法、他のベンチマークを用いることで様々な実験ができると思われる。