**Constrains Programming**

**1. Import và khởi tạo mô hình**

from ortools.sat.python import cp\_model

model = cp\_model.CpModel()

* cp\_model được import từ ortools.sat.python, chứa các công cụ cần thiết để xây dựng và giải các bài toán tối ưu hóa ràng buộc.
* model = cp\_model.CpModel() khởi tạo một mô hình CP-SAT.

**2. Khởi tạo các biến quyết định**

**Biến x**

x = []

for i in range(2\*n + 1):

x\_i = []

for j in range(2\*n + 1):

if i == j:

x\_i.append(0)

else:

x\_i.append(model.NewIntVar(0, 1, f'x[{i}][{j}]'))

x.append(x\_i)

* x[i][j] là biến nhị phân chỉ ra có đường đi trực tiếp từ thành phố i đến thành phố j hay không.
* x[i][i] luôn bằng 0 vì không có đường đi từ một thành phố đến chính nó.

**Biến y**

y = [model.NewIntVar(1, 2\*n + 1, f'y[{i}]') for i in range(2\*n + 1)]

* y[i] là thứ tự của thành phố i trong hành trình.

**Biến p**

p = [model.NewIntVar(0, k, f'p[{i}]') for i in range(2\*n + 1)]

* p[i] là số lượng hành khách tại thành phố i.

**3. Ràng buộc**

**Ràng buộc "one way in, one way out"**

for i in range(2\*n + 1):

model.Add(sum(x[i][j] for j in range(2\*n + 1)) == 1)

model.Add(sum(x[j][i] for j in range(2\*n + 1)) == 1)

* Đảm bảo rằng mỗi thành phố có chính xác một đường vào và một đường ra.

**Ràng buộc "start at city 0"**

model.Add(y[0] == 1)

* Đảm bảo rằng hành trình bắt đầu từ thành phố 0.

**Ràng buộc "number of passengers at city 0 equals 0"**

model.Add(p[0] == 0)

* Đảm bảo rằng số lượng hành khách tại thành phố 0 ban đầu là 0.

**Ràng buộc "not subcycles"**

for i in range(2\*n + 1):

for j in range(1, 2\*n + 1):

if i == j:

continue

model.Add(y[i]+1 == y[j]).OnlyEnforceIf(x[i][j])

* Đảm bảo rằng không có chu kỳ con bằng cách sử dụng thứ tự của các thành phố.

**Ràng buộc "picks before drops"**

for i in range(n+1, 2\*n + 1):

model.Add(y[i] > y[i - n])

* Đảm bảo rằng việc đón hành khách xảy ra trước khi trả hành khách.

**Ràng buộc "maximum k passengers"**

for i in range(2\*n + 1):

for j in range(1, 2\*n+1):

if i == j:

continue

# picks

if j <= n:

model.Add(p[j]-p[i]==1).OnlyEnforceIf(x[i][j])

# drops

if j > n:

model.Add(p[i]-p[j]==1).OnlyEnforceIf(x[i][j])

* Ràng buộc số lượng hành khách tối đa k và đảm bảo tính toán chính xác việc đón và trả hành khách.

**4. Mục tiêu**

model.Minimize(sum(sum(x[i][j]\*c[i][j] for j in range(2\*n + 1)) for i in range (2\*n + 1)))

* Mục tiêu là tối thiểu hóa tổng chi phí hành trình dựa trên ma trận chi phí c[i][j].

**5. Giải bài toán**

solver = cp\_model.CpSolver()

t1 = time.time()

stt = solver.Solve(model)

print(f"Time: {time.time()-t1}")

if stt == cp\_model.OPTIMAL:

print(f"Objective value: {solver.ObjectiveValue()}")

sol = {}

for i in range(2\*n + 1):

sol[solver.Value(y[i])] = i

sol = [sol[i+1] for i in range(2\*n + 1)]

print(f"Solution: {sol}")

print(f"Number of blinding constrains: {nOfBlindingConstrains(sol[1:])}")

* Khởi tạo solver và giải bài toán.
* Đo thời gian giải.
* Nếu tìm được lời giải tối ưu, in ra giá trị mục tiêu và thứ tự các thành phố trong hành trình.
* Sử dụng hàm nOfBlindingConstrains (không được định nghĩa trong đoạn mã này) để tính số ràng buộc blinding.

**Tổng kết**

Đoạn mã này xây dựng và giải một bài toán tối ưu hóa có liên quan đến việc định tuyến hành trình với các ràng buộc cụ thể về việc đón và trả hành khách, sử dụng thư viện OR-Tools của Google. Nó bao gồm việc khởi tạo mô hình, định nghĩa các biến quyết định, thêm các ràng buộc và mục tiêu, sau đó giải bài toán và in kết quả.

**Mixed Interger Programing**

**1. Khởi tạo solver**

from ortools.linear\_solver import pywraplp

solver = pywraplp.Solver.CreateSolver("SCIP")

* pywraplp được import từ ortools.linear\_solver, cung cấp các công cụ cần thiết để giải các bài toán tối ưu hóa tuyến tính và tuyến tính số nguyên.
* solver = pywraplp.Solver.CreateSolver("SCIP") khởi tạo một solver sử dụng SCIP, một công cụ mạnh mẽ để giải các bài toán tối ưu hóa.

**2. Khởi tạo các biến quyết định**

**Biến x**

x = []

for i in range(2\*n + 1):

x\_i = []

for j in range(2\*n + 1):

if i == j:

x\_i.append(0)

else:

x\_i.append(solver.IntVar(0, 1, f'x[{i}][{j}]'))

x.append(x\_i)

* x[i][j] là biến nhị phân chỉ ra có đường đi trực tiếp từ thành phố i đến thành phố j hay không.
* x[i][i] luôn bằng 0 vì không có đường đi từ một thành phố đến chính nó.

**Biến y**

y = [solver.IntVar(1, 2\*n + 1, f'y[{i}]') for i in range(2\*n + 1)]

* y[i] là thứ tự của thành phố i trong hành trình.

**Biến p**

p = [solver.IntVar(0, k, f'p[{i}]') for i in range(2\*n + 1)]

* p[i] là số lượng hành khách tại thành phố i.

**3. Ràng buộc**

**Ràng buộc "one way in, one way out"**

for i in range(2\*n + 1):

solver.Add(sum(x[i][j] for j in range(2\*n + 1)) == 1)

solver.Add(sum(x[j][i] for j in range(2\*n + 1)) == 1)

* Đảm bảo rằng mỗi thành phố có chính xác một đường vào và một đường ra.

**Ràng buộc "start at city 0"**

solver.Add(y[0] == 1)

* Đảm bảo rằng hành trình bắt đầu từ thành phố 0.

**Ràng buộc "number of passengers at city 0 equals 0"**

solver.Add(p[0] == 0)

* Đảm bảo rằng số lượng hành khách tại thành phố 0 ban đầu là 0.

**Ràng buộc "not subcycles"**

for i in range(2\*n + 1):

for j in range(1, 2\*n + 1):

if i == j:

continue

solver.Add(y[i] - y[j] + (2\*n + 1)\*x[i][j] <= 2\*n)

* Đảm bảo rằng không có chu kỳ con bằng cách sử dụng thứ tự của các thành phố và một ràng buộc bậc lớn.

**Ràng buộc "picks before drops"**

for i in range(n+1, 2\*n + 1):

solver.Add(y[i] >= (y[i - n] + 1))

* Đảm bảo rằng việc đón hành khách xảy ra trước khi trả hành khách.

**Ràng buộc "maximum k passengers"**

for i in range(2\*n + 1):

for j in range(1, 2\*n+1):

if i == j:

continue

# picks

if j <= n:

solver.Add(p[j] - p[i] >= 1 - (k + 1)\*(1 - x[i][j]))

solver.Add(p[j] - p[i] <= 1 + (k - 1)\*(1 - x[i][j]))

# drops

if j > n:

solver.Add(p[i] - p[j] >= 1 - (k + 1)\*(1 - x[i][j]))

solver.Add(p[i] - p[j] <= 1 + (k - 1)\*(1 - x[i][j]))

* Ràng buộc số lượng hành khách tối đa k và đảm bảo tính toán chính xác việc đón và trả hành khách.

**4. Mục tiêu**

solver.Minimize(sum(sum(x[i][j]\*c[i][j] for j in range(2\*n + 1)) for i in range (2\*n + 1)))

* Mục tiêu là tối thiểu hóa tổng chi phí hành trình dựa trên ma trận chi phí c[i][j].

**5. Giải bài toán**

t1 = time.time()

stt = solver.Solve()

print(f"Time: {time.time()-t1}")

if stt == pywraplp.Solver.OPTIMAL:

print(f"Objective value: {int(solver.Objective().Value())}")

sol = {}

for i in range(2\*n + 1):

sol[y[i].solution\_value()] = i

sol = [sol[i+1] for i in range(2\*n + 1)]

print(f"Solution: {sol}")

print(f"Number of blinding constrains: {nOfBlindingConstrains(sol[1:])}")

* Khởi tạo solver và giải bài toán.
* Đo thời gian giải.
* Nếu tìm được lời giải tối ưu, in ra giá trị mục tiêu và thứ tự các thành phố trong hành trình.
* Sử dụng hàm nOfBlindingConstrains (không được định nghĩa trong đoạn mã này) để tính số ràng buộc blinding.

**Tổng kết**

Đoạn mã này xây dựng và giải một bài toán tối ưu hóa tuyến tính số nguyên hỗn hợp có liên quan đến việc định tuyến hành trình với các ràng buộc cụ thể về việc đón và trả hành khách, sử dụng thư viện OR-Tools của Google. Nó bao gồm việc khởi tạo mô hình, định nghĩa các biến quyết định, thêm các ràng buộc và mục tiêu, sau đó giải bài toán và in kết quả.

**Branch and Bound**

**1. Khởi tạo dữ liệu:**

* n: Số lượng thành phố (bao gồm cả kho).
* k: Sức chứa tối đa của xe về số lượng hành khách.
* c: Ma trận khoảng cách giữa các thành phố (khoảng cách di chuyển giữa từng cặp thành phố).
* x: Mảng kích thước 2n + 1 được khởi tạo với các giá trị 0, dùng để lưu trữ thứ tự viếng thăm các thành phố.
* visited: Mảng Boolean kích thước 2n + 1 được khởi tạo với False cho tất cả các phần tử, theo dõi các thành phố đã được viếng thăm.
* cities: Danh sách chứa tất cả các thành phố được đánh số từ 1 đến 2n (bao gồm cả kho).
* load: Biến lưu trữ số lượng hành khách đang có trên xe, được khởi tạo bằng 0 (bắt đầu từ kho).
* best\_fitness: Biến lưu trữ giá trị (tổng quãng đường) tốt nhất của giải pháp tìm thấy cho đến thời điểm hiện tại, được khởi tạo với giá trị rất lớn (1e9).
* best\_sol: Biến lưu trữ giải pháp (chuỗi viếng thăm) tốt nhất cho đến thời điểm hiện tại, được khởi tạo bằng None.
* d\_min: Biến lưu trữ khoảng cách nhỏ nhất được tìm thấy trong quá trình khởi tạo.

**2. Tìm khoảng cách nhỏ nhất d\_min (tối ưu hóa tìm kiếm):**

* Duyệt qua toàn bộ ma trận khoảng cách c để tìm khoảng cách nhỏ nhất giữa hai thành phố bất kỳ, lưu trữ giá trị này trong d\_min.
* **Mục đích:** Giúp tiết kiệm thời gian tìm kiếm các cạnh tiềm năng trong bước tiếp theo bằng cách chỉ cần kiểm tra các cạnh có khoảng cách nhỏ hơn d\_min.

**3. Hàm check(v) (kiểm tra tính hợp lệ):**

* Kiểm tra xem thành phố v đã được viếng thăm chưa (visited[v]). Nếu đã viếng thăm (True), trả về False.
* Kiểm tra loại thành phố (v):
  + Nếu v > n: Đây là thành phố trả khách (v - n). Kiểm tra xem thành phố đón khách tương ứng (v - n) đã được viếng thăm chưa (visited[v-n]). Nếu đã viếng thăm (True), trả về True (cho phép trả khách). Ngược lại, trả về False (không thể trả khách vì chưa đón).
  + Nếu v <= n: Đây là thành phố đón khách. Kiểm tra xem sức chứa còn trống trên xe (load < k). Nếu còn trống, trả về True (cho phép đón khách). Ngược lại, trả về False (xe đã đầy).

**4. Hàm đánh giá fitness(j):**

* Tính tổng quãng đường di chuyển của giải pháp hiện tại (x).
* Bắt đầu từ chỉ số j trong mảng x, cộng dồn khoảng cách giữa các thành phố được viếng thăm theo thứ tự.
* c[x[j]][0]: Khoảng cách từ vị trí bắt đầu (gán cho x[0]) đến thành phố đầu tiên được viếng thăm (x[j]).
* Vòng lặp tính toán khoảng cách giữa các cặp thành phố lân cận trong giải pháp (c[x[idx]][x[idx+1]]).

**5. Hàm đệ quy Try(j) (xây dựng giải pháp):**

* Thực hiện tìm kiếm giải pháp bằng thuật toán tham lam cải tiến.
* Tham số j là chỉ số thành phố đang được xây dựng trong giải pháp.
* Duyệt qua danh sách các thành phố cities để kiểm tra tính hợp lệ của từng thành phố v bằng hàm check(v).
* Nếu v là thành phố hợp lệ:
  + Cập nhật thứ tự viếng thăm x[j] = v.
  + Cập nhật trạng thái viếng thăm visited[v] = True.
  + Cập nhật số lượng hành khách trên xe:
    - Nếu v <= n: Tăng load lên 1 (đón khách).
    - Nếu v > n: Giảm load xuống 1 (trả khách).
* Kiểm tra điều kiện dừng

**Thuật toán Greedy**

**Khởi tạo các biến ban đầu:**

solution = [] # danh sách thứ tự các điểm đã đi qua

solution.append(0) # bắt đầu từ điểm 0

distance = 0 # tổng khoảng cách đã đi (chi phí)

visited = [] # mảng đánh dấu các điểm đã được thăm

visited.append(1) # điểm bắt đầu đã được thăm

for i in range(2 \* n):

visited.append(0) # các điểm còn lại chưa được thăm

**Hàm kiểm tra tính hợp lệ của nút tiếp theo:**

def is\_valid(sol, node: int) -> bool:

# kiểm tra nếu nút đã được thăm

if visited[node] == 1:

return False

# tạo bản sao của giải pháp hiện tại và thêm nút mới

next\_sol = sol.copy()

next\_sol.append(node)

# kiểm tra tính hợp lệ

mark = [0] \* (n + 1)

n\_passengers = 0

for city in next\_sol[1:]:

if city <= n: # nếu là khách đón

n\_passengers += 1

mark[city] = 1

else: # nếu là khách trả

n\_passengers -= 1

mark[city - n] = 0

if n\_passengers > k: # kiểm tra số lượng hành khách vượt quá k

return False

if np.sum(mark) - n\_passengers != 0:

return False

return True

Hàm is\_valid kiểm tra xem nút tiếp theo có hợp lệ không dựa trên:

1. Nút đã được thăm chưa (visited[node] == 1).
2. Số lượng hành khách trên xe không vượt quá giới hạn k.
3. Tất cả khách đã đón thì phải được trả về đúng nơi.

**Hàm giải quyết chính:**

def solver(sol):

for \_ in range(1, 2 \* n + 1):

last\_node = sol[-1] # lấy nút cuối cùng trong giải pháp hiện tại

possible\_move = [] # danh sách các di chuyển khả thi (khoảng cách, nút)

for i in range(1, 2 \* n + 1):

if not is\_valid(sol, i):

continue

possible\_move.append((c[last\_node][i], i)) # tính toán khoảng cách và thêm vào danh sách

possible\_move.sort(key=lambda x: x[0]) # sắp xếp theo khoảng cách tăng dần

next\_node = possible\_move[0][1] # chọn nút có khoảng cách ngắn nhất

sol.append(next\_node) # cập nhật giải pháp

visited[next\_node] = 1 # đánh dấu nút đã được thăm

global distance

distance += possible\_move[0][0] # cập nhật tổng khoảng cách

return sol # trả về giải pháp

**Đoạn mã chính để gọi hàm solver:**

t1 = time.time()

abc = solver(solution) # bắt đầu giải quyết

print(f"Time: {time.time() - t1}") # in thời gian thực thi

print("Objective value: ", distance + c[abc[-1]][0]) # in giá trị mục tiêu (khoảng cách tổng cộng)

print("Solution: ", abc) # in giải pháp cuối cùng

print(f"Number of blinding constrains: {nOfBlindingConstrains(abc[1:])}") # in số lượng ràng buộc

**Tổng kết**

Đoạn mã trên là một giải pháp cho bài toán đón và trả khách trên tuyến đường sao cho khoảng cách di chuyển là ngắn nhất, tuân thủ theo các ràng buộc về số lượng hành khách trên xe không vượt quá giới hạn k. Hàm is\_valid kiểm tra tính hợp lệ của các nút tiếp theo, và hàm solver tìm kiếm giải pháp tối ưu thông qua việc lựa chọn nút tiếp theo có khoảng cách ngắn nhất từ nút hiện tại.

**Greedy +**

**Khởi tạo các hàm và biến**

sqrt\_n = np.sqrt(n)

def weight\_j(r, i):

ln = -np.log(r + 1e-5)

if ln < 1e-4:

return 0

return np.power(ln, 1 + 50 \* np.sqrt(i / n))

* sqrt\_n: tính căn bậc hai của n.
* weight\_j(r, i): tính trọng số cho khoảng cách r tại vị trí thứ i trong giải pháp. Hàm này sử dụng logarit tự nhiên của r và nhân với một hệ số phụ thuộc vào i và n.

**Hàm kiểm tra tính hợp lệ của nút**

def valid(sol, city, visited):

if visited[city] == 1:

return False

next\_sol = sol.copy()

next\_sol.append(city)

mark = [0] \* (n + 1)

n\_passengers = 0

for \_city in next\_sol[1:]:

if \_city <= n:

n\_passengers += 1

mark[\_city] = 1

else:

n\_passengers -= 1

mark[\_city - n] = 0

if n\_passengers > k:

return False

if np.sum(mark) - n\_passengers != 0:

return False

return True

Hàm valid kiểm tra tính hợp lệ của việc thêm city vào giải pháp sol:

1. Kiểm tra nếu city đã được thăm.
2. Tạo giải pháp tiếp theo và kiểm tra số lượng hành khách trên xe không vượt quá k.
3. Đảm bảo tất cả các khách đã đón thì phải được trả về đúng nơi.

**Hàm giải quyết chính**

def solve():

solution = []

solution.append(0) # bắt đầu tại thành phố 0

fitness = 0 # giá trị mục tiêu

visited = [0] \* (2 \* n + 1)

visited[0] = 1

for \_ in range(1, 2 \* n + 1):

c\_city = solution[-1] # thành phố hiện tại

weights = [] # phân phối xác suất

possible\_move = []

d\_max = 0

for city in range(1, 2 \* n + 1):

if valid(solution, city, visited) == False:

continue

d\_i = c[c\_city][city]

possible\_move.append([city, d\_i]) # [thành phố, khoảng cách]

possible\_move = sorted(possible\_move, key=lambda x: x[1])

if len(possible\_move) > 20:

possible\_move = possible\_move[:20]

for city in possible\_move:

if d\_max < city[1]:

d\_max = city[1]

if d\_max == 0:

next\_city = possible\_move[0]

else:

for move in possible\_move:

r = move[1] / d\_max

weights.append(weight\_j(r, \_))

if np.sum(weights) == 0:

next\_city = possible\_move[0]

else:

next\_city = random.choices(possible\_move, k=1, weights=weights)[0]

solution.append(next\_city[0])

visited[next\_city[0]] = 1

fitness += next\_city[1]

return solution, fitness

* solution: mảng chứa thứ tự các thành phố đã đi qua, bắt đầu từ thành phố 0.
* fitness: tổng khoảng cách đã đi.
* visited: mảng đánh dấu các thành phố đã được thăm.

**Hàm tìm giải pháp tốt nhất**

def solver():

record = 10e9

best\_sol = None

for seed in range(10):

print("seed: ", seed)

random.seed(seed)

sol, fit = solve()

print(fit + c[sol[-1]][0])

if record > fit:

record = fit

best\_sol = sol.copy()

return best\_sol, record

* solver chạy hàm solve với các giá trị ngẫu nhiên khác nhau để tìm giải pháp tốt nhất.
* record: giữ giá trị của giải pháp tốt nhất (khoảng cách ngắn nhất).
* best\_sol: giữ giải pháp tốt nhất tìm được.

**Đoạn mã chính để chạy giải thuật**

t1 = time.time()

best\_sol, best\_fitness = solver()

print(f"Time: {time.time() - t1}")

print("Objective value: ", best\_fitness + c[best\_sol[-1]][0])

print("Solution: ", best\_sol)

print(f"Number of blinding constrains: {nOfBlindingConstrains(best\_sol[1:])}")

* t1 = time.time(): bắt đầu đếm thời gian thực thi.
* best\_sol, best\_fitness = solver(): tìm giải pháp tốt nhất và giá trị mục tiêu.
* In ra thời gian thực thi, giá trị mục tiêu và giải pháp tìm được.

**Tổng kết**

Thuật toán này sử dụng một chiến lược dựa trên xác suất và trọng số để chọn lựa bước đi tiếp theo, đảm bảo rằng các bước đi tiếp theo không vi phạm các ràng buộc về số lượng hành khách. Giải thuật lặp lại nhiều lần với các hạt giống ngẫu nhiên khác nhau để tìm ra giải pháp tốt nhất có thể.

* **GA with Tabu Search**

**1. Các hàm tiện ích**

**1.1. Hàm tính tổng khoảng cách**

def totalDistance(inv):

summ = np.sum([dis\_mat[int(inv[i])][int(inv[i+1])] for i in range(2\*n-1)])

return summ + dis\_mat[0][int(inv[0])] + dis\_mat[int(inv[-1])][0]

Hàm totalDistance tính tổng khoảng cách của một chuỗi các thành phố. Nó tính tổng khoảng cách giữa từng cặp thành phố liên tiếp trong chuỗi inv, cộng thêm khoảng cách từ điểm bắt đầu (0) tới thành phố đầu tiên và từ thành phố cuối cùng về điểm bắt đầu.

**1.2. Hàm tính số lượng ràng buộc bị vi phạm**

def nOfBlindingConstrains(inv):

count = 0

mark = [0]\*(n+1)

n\_passengers = 0

for i in range(2\*n):

if inv[i] <= n:

n\_passengers += 1

mark[inv[i]] = 1

else:

n\_passengers -= 1

mark[inv[i]-n] = 0

if n\_passengers > k:

count += 1

return count + np.sum(mark)

Hàm nOfBlindingConstrains kiểm tra số lượng ràng buộc bị vi phạm trong một chuỗi các thành phố. Các ràng buộc bao gồm:

* Số lượng hành khách trên xe không được vượt quá k.
* Các hành khách phải được trả về đúng nơi.

Hàm này đếm số lần các ràng buộc này bị vi phạm.

**1.3. Hàm tính giá trị thích nghi (fitness)**

def fitness(inv):

pen = totalDistance(np.arange(1, len(inv) + 1))

distance = totalDistance(inv)

return 5\*pen\*nOfBlindingConstrains(inv)/np.sqrt(distance) + distance

Hàm fitness tính giá trị thích nghi của một chuỗi các thành phố, bao gồm tổng khoảng cách và một phần phạt dựa trên số lượng ràng buộc bị vi phạm.

**2. Khởi tạo quần thể ban đầu**

def initPopulation(cities, size):

population = []

for i in range(size):

invi = list(cities)

random.shuffle(invi)

distance = fitness(invi)

population.append([distance, invi, 0])

return population

Hàm initPopulation khởi tạo một quần thể ngẫu nhiên gồm size cá thể. Mỗi cá thể là một chuỗi các thành phố được xáo trộn ngẫu nhiên và tính giá trị thích nghi.

**3. Toán tử lai ghép (crossover)**

**3.1. Lai ghép 1 điểm cắt**

def singlePointCrossover(father1, father2):

point = random.randint(0, len(father1) - 1)

new\_invi1 = father2[0:point]

for invi in father1:

if invi not in new\_invi1:

new\_invi1.append(invi)

new\_invi2 = father1[0:point]

for invi in father2:

if invi not in new\_invi2:

new\_invi2.append(invi)

return new\_invi1, new\_invi2

Hàm singlePointCrossover thực hiện lai ghép 1 điểm cắt. Nó chọn một điểm cắt ngẫu nhiên và tạo hai cá thể mới bằng cách kết hợp các phần tử từ hai cá thể cha.

**3.2. Lai ghép 2 điểm cắt**

def multiPointCrossover(father1, father2):

points = sorted(random.sample(father1, 2))

point1 = int(points[0]) - 1

point2 = int(points[1]) - 1

new\_invi1 = father2[point1:point2]

c1 = 0

for invi in father1:

if invi not in new\_invi1:

c1 += 1

if c1 <= point1:

new\_invi1.insert(c1-1, invi)

else:

new\_invi1.append(invi)

new\_invi2 = father1[point1:point2]

c1 = 0

for invi in father2:

if invi not in new\_invi2:

c1 += 1

if c1 <= point1:

new\_invi2.insert(c1-1, invi)

else:

new\_invi2.append(invi)

return new\_invi1, new\_invi2

Hàm multiPointCrossover thực hiện lai ghép 2 điểm cắt. Nó chọn hai điểm cắt ngẫu nhiên và tạo hai cá thể mới bằng cách kết hợp các phần tử từ hai cá thể cha.

**4. Toán tử đột biến (mutation)**

def mutation(father, lenCities=51):

new\_invi = father.copy()

for \_ in range(5):

points = random.sample(new\_invi, 2)

temp = new\_invi[int(points[0]) - 1]

new\_invi[int(points[0]) - 1] = new\_invi[int(points[1]) - 1]

new\_invi[int(points[1]) - 1] = temp

return new\_invi

Hàm mutation thực hiện đột biến bằng cách chọn ngẫu nhiên hai thành phố và đổi chỗ chúng. Thao tác này được thực hiện năm lần cho mỗi cá thể.

**5. Chọn lọc tự nhiên (tournaments)**

def tournaments(old\_population, new\_invivduals, pop\_size = 100):

merge = sorted(old\_population + new\_invivduals, key=lambda x: (x[0], -x[2]))

best\_fitness = merge[0][0]

new\_population = merge[:10]

selected = []

for inv in merge[10:]:

if inv[0]/best\_fitness > 0.75 hoặc inv[2] < 50:

selected.append(inv)

new\_population = new\_population + random.choices(selected, k=pop\_size-10)

for inv in new\_population:

inv[2] += 1

return new\_population

Hàm tournaments chọn lọc các cá thể tốt nhất từ quần thể cũ và mới. Nó giữ lại 10 cá thể tốt nhất và chọn ngẫu nhiên các cá thể khác dựa trên giá trị thích nghi và số thế hệ mà cá thể đã tồn tại.

**6. Thuật toán di truyền (Genetic Algorithm)**

def GA(

population,

lenCities = 51,

pop\_size = 100,

num\_of\_generations = 2000,

crossover\_rate = 0.99,

mutation\_rate = 0.2,

crossover\_func=singlePointCrossover

):

for \_ in range(num\_of\_generations):

new\_inviduals = []

n\_i = int(pop\_size / 2)

for \_i in range(n\_i):

father1, father2 = chooseFather(population)

if father1 != None:

if random.random() <= crossover\_rate:

inv1, inv2 = crossover\_func(father1, father2)

if random.random() <= mutation\_rate:

inv1 = mutation(inv1, lenCities)

inv2 = mutation(inv2, lenCities)

else:

inv1, inv2 = father1, father2

else:

return sorted(population)

new\_inviduals.append([fitness(inv1), inv1, 0])

new\_inviduals.append([fitness(inv2), inv2, 0])

population = tournaments(population, new\_inviduals, pop\_size)

population = sorted(population)

best\_invi = population[0]

if best\_invi[2] > 20 \* (n\*\*0.5):

return population

if \_ % 100 == 0:

print(f'Loop: {\_},\tFitness: {best\_invi[0]},\tDistance: {totalDistance(best\_invi[1])},\tBlinding constrains: {nOfBlindingConstrains(best\_invi[1])}')

return population

Hàm GA thực hiện thuật toán di truyền:

* Khởi tạo các cá thể mới bằng cách lai ghép và đột biến các cá thể hiện tại.
* Chọn lọc các cá thể tốt nhất để tạo thành quần thể mới.
* Kiểm tra các điều kiện dừng (số thế hệ hoặc không cải thiện sau một số thế hệ nhất định).
* In ra thông tin về giá trị thích nghi, tổng khoảng cách và số ràng buộc bị vi phạm sau mỗi 100 thế hệ.

**7. Chạy thuật toán với nhiều lần khởi tạo khác nhau**

total\_distance = 0

total\_time = 0

total\_bCons = 0

for n\_seed in range