ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

----- ☙ 🕮 ❧ -----

TỐI ƯU LẬP KẾ HOẠCH

BÁO CÁO MINI PROJECT

ĐỀ TÀI 7: CBUS

Giảng viên hướng dẫn: TS. Bùi Quốc Trung

Sinh viên thực hiện:
- Đào Thành Mạnh 20211014
- Nguyễn Tiến Hoàng 20210379
- Mai Trung Hiếu 20215369
- Nguyễn Gia Khánh 20204661

Hà Nội, Ngày 10 tháng 6 năm 2024

MỤC LỤC

MỤC LỤC **2**

1. Phân công nhiệm vụ **4**

2. Phát biểu bài toán **4**

3. Các thuật toán đề xuất và công cụ sử dụng **5**

3.1. Constraint Programming 5

3.1.1. Import và khởi tạo mô hình 5

3.1.2. Khởi tạo các biến quyết định 5

3.1.3. Ràng buộc 6

3.1.4. Mục tiêu 7

3.1.5. Giải bài toán 7

Tổng kết 7

3.2. Mixed Interger Programming 8

3.2.1. Khởi tạo solver 8

3.2.2. Khởi tạo các biến quyết định 8

3.2.3. Ràng buộc 8

3.2.4. Mục tiêu 10

3.2.5. Giải bài toán 10

Tổng kết 10

3.3. Branch and Bound 10

3.3.1. Khởi tạo dữ liệu 10

3.3.2. Tìm khoảng cách nhỏ nhất d_min (tối ưu hóa tìm kiếm): 11

3.3.3. Hàm check(v) (kiểm tra tính hợp lệ): 11

3.3.4. Hàm đánh giá fitness(j): 11

3.3.5. Hàm đệ quy Try(j) (xây dựng giải pháp): 11

3.4. Greedy 12

3.4.1. Khởi tạo các biến ban đầu: 12

3.4.2. Hàm kiểm tra tính hợp lệ của nút tiếp theo: 12

3.4.3. Hàm giải quyết chính: 13

3.4.4. Đoạn mã chính để gọi hàm solver: 13

Tổng kết 13

3.5. Greedy+ 13
3.5.1. Khởi tạo các hàm và biến 13
3.5.2. Hàm kiểm tra tính hợp lệ của nút 14
3.5.3. Hàm giải quyết chính 14
3.5.4. Hàm tìm giải pháp tốt nhất 15
3.5.5. Đoạn mã chính để chạy giải thuật 16
Tổng kết 16
3.6. GA with Tabu Search 16
3.6.1. Các hàm tiện ích 16
3.6.2. Khởi tạo quần thể ban đầu 17
3.6.3. Toán tử lai ghép (crossover) 17
3.6.4. Toán tử đột biến (mutation) 18
3.6.5. Chọn lọc tự nhiên (tournaments) 18
3.6.6. Thuật toán di truyền (Genetic Algorithm) 19
3.6.7. Chạy thuật toán với nhiều lần khởi tạo khác nhau 20
4. Kết quả thực nghiệm 20
5. Kết luận 20

1. Phân công nhiệm vụ

Thành viên	MSSV	Thuật toán
Đào Thành Mạnh	20211014	Constraint Programming (CP)
Nguyễn Tiến Hoàng	20210379	Quay lui - Nhánh cận, MIP
Mai Trung Hiếu	20215369	Greedy, Greedy+
Nguyễn Gia Khánh	20204661	GA Tabu Search

2. Phát biểu bài toán

Có n hành khách 1, 2, ... , n. Hành khách i muốn di chuyển từ điểm i đến điểm i + n (với i = 1, 2, ..., n). Có một chiếc xe buýt đặt tại điểm 0 và có k chỗ ngồi để vận chuyển hành khách (có nghĩa là tại bất kỳ thời điểm nào, có tối đa k hành khách trên xe buýt). Bạn được cung cấp ma trận khoảng cách c trong đó c(i, j) là khoảng cách di chuyển từ điểm i đến điểm j (với i, j = 0, 1, ..., 2n). Tính toán lộ trình ngắn nhất cho xe buýt, phục vụ n hành khách và quay lại điểm 0.

Đối với phần lập trình, đầu vào và đầu ra có định dạng như sau:

- Input:
  - Dòng 1 chứa n và k ($1 \le n \le 1000$, $1 \le k \le 50$).
  - Dòng i + 1 (với i = 1, 2, ..., 2n + 1) chứa dòng thứ i - 1 của ma trận c (các hàng và cột được đánh số từ 0, 1, 2, ..., 2n).
- Output:
  - Dòng 1: ghi giá trị n.
  - Dòng 2: ghi chuỗi các điểm (đón và trả) của hành khách (các điểm được ngăn cách bằng một ký tự SPACE).

Ví dụ:

• Input

```
53
0 5 8 11 12 8 3 3 7 5 5
5 0 3 5 7 5 3 4 2 2 2
8 3 0 7 8 8 5 7 1 6 5
11 5 7 0 1 5 9 8 6 5 6
```

12 7 8 1 0 6 10 10 7 7 7
8 5 8 5 6 0 8 5 7 3 4
3 3 5 9 10 8 0 3 4 5 4
3 4 7 8 10 5 3 0 6 2 2
7 2 1 6 7 7 4 6 0 5 4
5 2 6 5 7 3 5 2 5 0 1
5 2 5 6 7 4 4 2 4 1 0

- **Output**

5
1 2 6 7 5 10 3 4 8 9

3. Các thuật toán đề xuất và công cụ sử dụng

3.1. Constraint Programming

3.1.1. 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.

3.1.2. 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`.

3.1.3. 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.

3.1.4. 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]`.

3.1.5. 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.

3.2. Mixed Interger Programming

Xử lý tập dữ liệu lấy từ kaggle bằng Visual Studio Code sau đó lưu dữ liệu đã xử lý vào các sheets của file excel gốc **Data.xlsx**

3.2.1. 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.

3.2.2. 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`.

3.2.3. 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.

3.2.4. 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]`.

3.2.5. 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.

3.3. Branch and Bound

3.3.1. 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.

3.3.2. 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`.

3.3.3. 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).

3.3.4. 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]]`).

3.3.5. 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

3.4. Greedy

3.4.1. 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
```

3.4.2. 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.

3.4.3. 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
```

3.4.4. 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.

3.5. Greedy+

3.5.1. 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`.

3.5.2. 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.

3.5.3. 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.

3.5.4. 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.

3.5.5. 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.

3.6. GA with Tabu Search

3.6.1. Các hàm tiện ích

3.6.1.1. 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.

3.6.1.2. 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.

3.6.1.3. 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.

3.6.2. 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.

3.6.3. Toán tử lai ghép (crossover)

3.6.3.1. 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.

3.6.3.2. 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.

3.6.4. 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.

3.6.5. 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.

3.6.6. 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.

3.6.7. Chạy thuật toán với nhiều lần khởi tạo khác nhau

4. Kết quả thực nghiệm

Dưới đây là thông tin bộ dữ liệu sử dụng:

TEST	N	K
1	5	3
2	10	6
3	100	40
4	500	40
5	1000	40

Bảng dưới đây cho thấy kết quả thực nghiệm đạt được của các thuật toán. Từ quá trình thực nghiệm nhóm rút ra những nhận xét sau:

1. Các phương pháp giải chính xác (quay lui, CP, MIP):
   - Được sử dụng cho các test case có bộ dữ liệu nhỏ.
   - Không phù hợp với các test case lớn hơn do độ phức tạp tính toán và tài nguyên bộ nhớ.
   - OR-Tools không thể giải quyết được các bài toán lớn do hạn chế về tài nguyên.
2. Thuật toán tham lam:
   - Nhanh chóng nhưng không đảm bảo tối ưu tuyệt đối.

Test	Branch and Bound		CP		MIP		Greedy		Greedy+		GA with Tabu Search	
	f	t	f	t	f	t	f	t	f	t	f	t
1	37	0,091	37	0,034	37	0,298	49	0,001	50	0,004	54	1,747
2			38	1,894			41	0,001	40	0,016	92	6,399
3							144	0,151	143	1,357		
4							6552	12,147	6367	208,124		
5							12176	151,913				

5. Kết luận