CJ 대한통운 미래기술챌린지 2023

Team: IECS

강세정

장현우

장동현

최세연

Contents

Algorithm

Genetic algorithm은 최적화 및 meta-heuristic 알고리즘의 한 종류로 생물학적 진화의 개념을 모방해 최적에 가까운 해를 단계적으로 찾아준다. 핵심 개념은 generation과 individual이다. 전체 알고리즘은 다음 figure와 같이 이뤄진다. 현재 문제 상황은 배송 가능한 시간, 차량의 용량 등의 다양한 제약 조건이 있기 때문에 search space에서 안정적으로 최적해를 찾아가기 위해 해당 알고리즘을 선택했다.

Genetic Algorithm
Initialization random하게 solution들을 생성한다. 이를 'individual'이라 표현하며, 생성된 모든 solution을 'population'이라 한다. Repeat Fitness evaluation Objective function을 통해 각 individual의 적합도를 평가한다. Selection 적합도를 기반으로 높은 점수를 얻은 유전자에 가중치를 주어 random sampling해 다음 세대에 포함될 후보군을 선정한다. Selection method로는

roulette wheel 방법을 사용했다.

Crossover

selection에서 추출된 individual간의 breeding을 통해 새로운 후보 individual를 생성(offspring generation)한다. Breeding을 통해 적합도를 높이되 individual의 다양성을 유지한다.

Mutation

일정한 확률로 individual의 정보를 변이한다. Mutation 역시 다양성을 유지하고, local minima에 빠지는 것을 방지한다.

Replacement

Offspring을 현재 individual로 설정하고, fitness evaluation 부터의 과정을 반복한다.

Until T(generation 횟수)

Baseline

각 python 파일은 다음과 같은 기능들을 수행한다.

main.py: genetic algorithm을 위한 data 및 parameter들을 지정하는 파일.

pyVRP.py: genetic algorithm의 주요 로직이 담긴 파일.

eval.py: target function 등에서 거리, 시간, 용량, 비용을 계산하기 위해서 보조적인 계산을 진행하는 파일.

utils.py: 이외 편의를 위해 사용한 plotting, data pre-processing, 기타 ga위한 알고리즘 등을 담은 파일.

main.py를 실행하면 전체 코드가 실행되어 차량 배치 결과를 얻을 수 있다.

Main.py

Main 함수에서는 day, time_batch, terminal을 돌면서 genetic algorithm을 돌려 각 시점의 최적 solution을 찾는다. 이때 time_batch는 우리가 계산을 하는 시점을 말하며, 24시간을 plan_time_hour으로 나눠 각각의 time_batch를 설정한다. 예를 들어 plan_time_hour가 2시간이면, 하루에 12번 상차를 하고, time_batch는 하루 중 해당 상차 시간의 index로 0, 1, … 11 중 하나로 선정된다.

모든 vehicle은 특정 하나의 terminal에 소속되어 있다. 각 배치에서는 해당 terminal의 vehicle을 통해 주문을 모두 배송하고, 자신의 terminal에 돌아가는 것을 가정하되 부족분이나 여유분이 있는 terminal의 경우 reallocate_veh을 통해 vehicle의 소속 terminal을 바꾸는 형식으로 vehicle들을 조정한다. vehicle의 재배치 알고리즘은 다음과 같다.

Run_ga의 결과물로 해당 terminal에서 현재 사용중인 vehicle에 대한 정보 fleet_used_now를 얻고 이를 기반으로 각 terminal별로 차량의 하한선 min_car를 계산한다. 모든 terminal은 반드시 자신의 terminal에 min_car 이상 개수의

vehicle을 갖고 있어야 한다.

매 time_batch마다 전체 terminal에 위치한 차들의 정보와 미처리 주문 정보 *unassigned_orders*, *min_car* 등을 reallocate_veh 함수의 input으로 넣어 모든 vehicle 을 재배치한다. 이 과정을 통해 추가적인 차량이 필요한 terminal이 주변 가까운 terminal에서 차를 배정받을 수 있다.

Baseline

```
Terminal_list = 상차지 터미널
Fleet_used_now = 현재 이용중인 vehicle의 list
Demand_df = 주문 당 하차 가능한 시간 list
Unassigned_orders_count = 한 터미널에서 처리되지 못하고 남은 물량
min_car = 각 terminal에서 가져야 하는 최소한의 차의 개수
For day ∈(0,⋯,6):
   For time_batch ∈ Time_batch:
      For terminalID ∈ Terminal_list:
        fleet_used_now <- run_ga(terminal_id, day, group, demand_df)
        min_car <- max(5, len(fleet_used_now))
      if not day ==6 and orders ==4:
        reallocate_veh(min_car, unassigned_orders, veh_table)
```

Run_ga 함수는 input으로 *terminal_id, day, time_batch*를 입력으로 받아 지정된 터미널, 날짜, 시간대에 해당하는 전처리를 수행한 후 genetic algorithm을 시행한다. 함수의 실행 결과로 *fleet_used_now, unassigned_orders*을 반환한다. 또한 mutation_rate, elite(Genetic algorithm에서 *elite* 수만큼 각 세대마다 가장 우수한 상위 개체를 보존하여 breeding을 수행한다.), population_size, generations(generation 횟수)를 parameter로 받는다.
실험을 통해 얻은 최적 parameter 값은 mutation_rate = 0.2, elite = 10, population_size = 100, generations = 100이다.

우선 tmp_df라는 각 time_batch에서 처리해야 할 물량을 계산한다. 미처리 물량과 현재 배치에 새로 들어오는 물량을 합쳐서 산정한다(line 1~5). 이때, 미처리 물량은 크게 unassigned(Unassigned_rows_dict)와 future(future_rows_dixt)로 나뉜다. Unassigned는 해당 terminal에 물량이 지나치게 많아 처리되지 못하고 남은 order를, future는 하차 가능시간이 현재 시점과 지나치게 멀어서 하차 가능시간과 가까워 질때까지 미뤄진 order를 말한다. 현재 time_batch가 새로운 주문이 들어오는 0시, 6시, 12시, 18시에 속한다면 새로운 주문도 포함시킨다(line 2).

마지막 날을 제외하고는 효율적인 동선을 위해 future_rows_dict를 계산하고, tmp_df에서 제외하는 것을 반복한다(line 6~15). 그리고 genetic_algorihtm_vrp 함수에 *index_position, tmp_df, cbm_list, veh_info*를 input으로 넘겨준다.

실제 코드에서 넘겨주는 차량에 대한 정보 veh_info는 전체 차량 중 현재 위치가 terminal_id 인 차량들의 정보를 관리하며 vehicleID, 재활용되는 차량인지(fleet_available_no_fixed_cost), 현재 사용중인 차량인지(fleet_available) 등의

정보를 담고 있다.

Solution을 이용해 각 차량이 배송을 완료하고 복귀하는 시각(return_time)을 계산하여 veh_table에 현재 시점의 time_batch의 차량 정보를 업데이트한다(line 15~16). 이 값은 다음 time_batch에서 사용 가능한 차량을 확인할 때 사용된다. 현재 time_batch에서 처리하지 못한 주문을 파악하여 unassigned_rows로 저장한다. 다음 time_batch에서 다른 주문들과 함께 처리된다(line 18~21, 3, 5).

	run_ga
	Input : *terminal_id, day, group, demand_df* Output: *fleet_used_now, len(unassigned_idx)* Parameters: population_size , mutation_rate, elite, generations Plan_time_hour: 몇 시간 단위로 차를 재출발할 것인지에 대한 변수 Time_batch: 하루 중 몇 번째 batch의 time인지에 대한 변수 Tmp_df: 현재 time_batch, terminal에서 처리해야 하는 주문 Index_positions: 현재 tmp_df의 주문들의 도착지 Cbm_list: 현재 tmp_df의 주문들의 cbm(무게) Veh_list: 현재 tmp_df에 해당 terminal에 위치한 vehicle list Solution: 각 주문이 어떤 차량에 어떤 주문들과 묶어서 이동 되었는 지에 대한 정보와 unassigned_order에 대한 정보를 담고 있는 변수
1	if time_batch의 시각이 새로운 order가 들어오는 시각과 동일:
2	tmp_df <- order_table에서 현재 date, group, terminalid를 만족하는 주문 list
3	tmp_df <- unassigned_rows_dict ∪ future_rows_dict ∪ tmp_df
4	else:
5	tmp_df <- unassigned_rows_dict ∪ future_rows_dict
6	if day != 6:
7	future_rows <- tmp_df 중 6시간 이내에 하차 불가능한 주문
8	if future_rows not empty:
9	future_rows_dict[terminalID] <- future_rows
10	else:
11	future_rows_dict[terminalID] <- ∅
12	else: 마지막 날에는 모든 주문을 처리해야 하므로 future_rows를 만들지 않음
13	future_rows_dict[terminalID] <- ∅
14	tmp_df <- tmp_df – future_rows_dict[terminalID]
15	solution, fleet_used_now <- genetic_algorithm_vrp(index_positions, tmp_df, cbm_list, veh_info)
16	return_time <- 각 vehicle의 return time을 계산
17	update_veh_table (veh_table, return_time)
18	if unassigned_rows not empty:
19	unassigned_rows_dict[terminalID] <- unassigned_rows
20	else:
21	unassigned_rows_dict[terminalID] <- ∅
22	return fleet_used_now, 미처리 물량의 개수
23	

pyVRP.py

Genetic_algorithm_vrp 함수는 genetic algorithm을 사용하여 차량 경로 문제(Vehicle Routing Problem, VRP)을 해결하는 작업을 수행한다. 우선 주어진 고객

의 위치 정보를 바탕으로 초기 해(차량 경로 리스트, initial population)를 생성한다(line 1). Population의 각 individual은 출발지, 사용한 차량, 각 차량의 고객 방문 경로를 포함한다. 각 individual에 target function 및 fitness_function을 적용해 적합도를 계산한다(line 3~4). target function에서는 차량별 고정 비용과 가변 비용을 계산하고 이를 기준으로 total cost를 계산하여 각 경로의 fitness를 평가한다.

초기 해 중에서 total cost가 가장 낮은 해를 elite 개체로 선택한다(line 5~7). 초기 해 구성이 끝나면 지정된 generations 만큼 자손 생성(offspring generation)과 목표 함수 계산(fitness_function), 최적화를 반복한다(line 8~15). 자손 생성 과정에서는 Breeding과 swap, insertion이 포함된 Mutation을 적용하여 다양성을 확보한다. 자손 중 가장 cost가 적은 자손과 기존 엘리트 개체를 비교하여 새로운 자손 중에서 현재 엘리트 개체보다 우수한 해가 발견되면 엘리트 개체를 갱신한다(line 13~15). Fleet_used_now를 통해 현재 세대에서 최종적으로 사용한 차량을 return한다(line 16~18).

	Genetic_algorithm_vrp
	Input: *parameters, generations* Output: *solution, fleet_used_now* Fleet_used_now: **현재 이용중인 vehicle 의 list**
1	Population: **여러 개의 individual 로 이뤄져 있음. 한 세대를 의미**
2	Population <- initial_population()
3	cost, population <- target_function(population), initial_population 의 각 individual 의 cost 계산
4	fitness <- fitness_function(cost)
5	elite_ind <- elite_distance(), 가장 비용이 낮은 individual(elite)의 subroute 의 총 이동 거리 합 float
6	elite_cst <- elite 의 cost
7	solution <- elite individual
8	**Repeat generations times**
9	offspring <- breeding(cost, population, elite). elite & fitness 기반으로 선정한 부모를 룰렛 휠 방식으로 자연선택. 추가로 교차 연산자를 사용하여 부모 개체에서 자손을 생성 및 반환
10	offspring <- mutation(offspring). swap 과 insertion 돌연변이 연산자를 적용하여 변형된 자손 개체 생성 및 반환
11	cost, population <- target_function(population), 각 individual 의 cost 계산
12	fitness <- fitness_function(cost)
13	elite_child <- elite_distance(), 가장 비용이 낮은 individual(elite)의 subroute 의 총 이동 거리 합 float
14	If elite_ind > elite_child :
15	Elite_ind, solution, elite_cst← 자손 중 elite 로 대체
16	For 차량 index ∈ 사용된 차량 index 의 list:
17	fleet_used_now[차량 index] = 1
18	return solution, fleet_used_now

initial_population 함수는 genetic_algorithm을 위한 초기 population에서 각 individual의 subroute를 생성하는 역할을 한다. Population 내의 각각의

individual은 해당 단계에서 처리하는 모든 주문들을 하나의 차량이 처리하는 subroute들로 구분되어 있다. Genetic algorithm의 특성을 살려 population 내의 다양성을 위해 하나의 subroute에 들어가야 할 주문의 수, 주문의 종류, 차의 종류를 random하게 선정한다(line 15~17). main constraint는 각 차량에 배정한 주문의 CBM의 합이 차량의 maxCBM을 넘지 말아야 한다는 것이다(line 18~19).

현재 주문량이 너무 많아 한번에 처리할 수 없는 경우엔 계속해서 constraint를 만족하지 못하기 때문에 첫 번째 generation에서 total_demand_unassigned에 이번 time_batch에서 처리하지 않을 주문을 append하고, 이후엔 generation에서 해당 주문을 고려하지 않는다(line 4~7). 만약 현재 terminal에 vehicle이 없을 경우엔 전체 주문을 total_demand_unassigned로 넘긴 후 population을 generate하지 않는다.

만약 첫 번째 generation에서 feasible한 individual가 존재했음에도, 두 번째 generation부터 feasible한 solution이 잘 생성되지 않는 경우 이외의 random한 offpsring을 새로 generate하지 않고, 처음의 individual을 그대로 사용한다(line 2~6, 8~9, 26~27).

예를 들어 차량의 CBM이 27, 55이고 order cbm이 13, 14, 22, 33인 경우를 생각해보자. First generation에서 우연히 각 차에 (13, 14), (22, 33) 주문이 배정되었다 하더라도 이외의 모든 generation에선 (13, 22), (14, 33) … 차량의 max CBM을 넘는 경우가 생긴다. 이런 상황을 방지하기 위해서 우리는 첫번째 individual을 고정적으로 사용하는 것이다(line 26, 27).

	Initial_population
	Input: *clients_temp, depots, vehicles, fleet_available, order_CBM* Output: *population* clients_temp: 주문 index list depots: 주문의 하차지 list vehicles: 가능한 차량 list order_CBM: 모든 주문의 CBM을 담은 list
1	Repeat *generation* times
2	Flag_first_is_best<-False
3	While len(clients_tmp)>0:
4	Repeat_count+ =1
5	If repeat_count >=1000:
6	If 현재 turn이 첫 번째 generation:
7	Total_demand_unassigned.append(clients_temp.pop())
8	Else:
9	Flag_first_is_best <- True
10	If not vehicles:
11	If 현재 turn이 첫 번째 generation:
12	Total_demand_unassigned.extend(clients_temp)
13	Else:
14	Flag_first_is_best <- True
15	e<- random.sample(vehicles)
16	d<- random.sample(depots)
17	c<- random.sample(clients_temp)

```
18      if sum(order_CBM) > capacity(e):
19          continue
20      clients_temp <- clients_temp의 주문 중 c에 포함된 주문 제외하여 list update
21      routes_vehicles.append(e)
22      routes_depot.append(d)
23      for idx in c:
24          tmp.append(int(idx))
25      routes.append(tmp)
26      if flag_first_is_best:
27          population.append(first_individual)
28      else:
29          population.append([routes_depot, routes, routes_vehicles,
    total_demand_unassigned])
```

utils.py

reallocate_veh 함수는 한 time_batch가 끝날때마다 실행되는 함수로 vehicle 부족분이 있는 terminal에 다른 terminal의 vehicle 여유분을 보내는 총 terminal의 보유 vehicle_list 재배치를 진행한다. 해당 time_batch 시점에서 각 terminal에 미처리 주문이 있을 경우 가장 가까운 terminal부터 가져올 수 있는 vehicle이 있는지 확인한다(line 1~6).

예를 들어 terminalA가 terminalB에서 차를 가져오려면 terminalB의 미처리 물량이 없어야 하고(line 6), 여유분의 차량이 있어야 한다(line 9). 여유분의 차량은 현재 terminal의 차량 대수와 min_car의 차이를 통해 구한다(line 8). 만약 여유분의 차량이 terminalA에서 필요로 하는 차량보다 많다면 terminalA로 보낼 차량을 고르기 위해서 random sampling한다(line11). 만약 terminalA에서 필요로 하는 차량이 terminalB의 가능한 차량보다 더 많다면 terminalB의 모든 차량을 terminalA로 보낸다(line 14). 이렇게 다른 terminal로 이동하는 차량의 정보는 전부 global 변수인 Veh_table를 업데이트하여 관리한다.

Get_total_dict 함수는 veh_table을 input으로 받아 차량 재분배 등 모든 상황을 고려해 현재 시점에서 각 terminal에 위치한 사용할 수 있는 차량 list (*fleet_available),* 한번 출고해 fixed cost 없이 재사용할 수 있는 차량 list(*fleet_available_no_fixed_cost*)를 return한다.

	Reallocate_veh
	Input: *min_car, veh_table, asc_dist_dict, unassigned_orders, terminals, day, group, moved_df* Output: *None* Asc_dist_dict: terminal 별로 다른 teminal까지 걸리는 시간, 거리에 대한 정보를 담은 dictionary
1	For terminal ∈ terminals
2	If unassigned_orders[terminal]:
3	Car_taken = 0
4	For dist, time, arrival_terminal ∈ asc_dist_dict[terminal]:
5	Total_dict <- get_total_dict(veh_table)
6	If not unassigned_orders[arrival_terminal]:
7	Available_cars <-min_car로 정의된 하한선보다 많아 여유분이 있는 경우 count
8	If available_cars:
9	If available_cars>=unassigned_orders[terminal] – car_taken:
10	Cur_car_taken <- unassigned_orders[terminal] – car_taken
11	Cur_car_taken이 terminalB에 있는 여유분보다 작으면 random sampling하여 terminalA로 데려올 차를 선택한다.
12	Veh_table update.
13	else:
14	Cur_car_taken <- available_cars
15	Veh_table update
	Get_total_dict
	Input: *veh_table* Output: *total_dict*
1	For center ∈ veh_table[‘StartCenter’].unique()
2	Center_data <- veh_table[‘CurrentCenter’]==center
3	Total_dict[center] <- [fleet_available, fleet_available_no_fixed_cost]
4	Return total_dict

Appendix

구현환경

Root directory에 eval.py, main.py, pyVRP.py, report.py, utils.py 파일과 requirement.txt 파일이 필요하다.
dataset(od_matrix.csv, orders_table.csv, Terminals.csv, veh_table.csv) 또한 동일한 root directory에 위치해야 한다.

다음 코드를 터미널에 입력하여 실행할 수 있다.

```
$ pip install -r requirements.txt
$ python main.py
```

코드를 실행하면 Pivot_table_filled.csv 파일과 distance_matrix.csv 파일이 생성되며, 결과 파일은 ‘./결과’ 폴더에 생성된다. Main.py에서 FOLDER_PATH 변수를 설정하여 결과를 출력할 경로를 설정할 수 있다.

모듈 버전 정보

Python version 3.9.

기타 구현 함수 설명

PyVRP.py

Target_function
주어진 개체마다 아래의 순서를 반복하며 비용을 계산한다. 비용은 차량 이동에 따른 비용과 제약을 만족하지 못했을 때의 페널티 비용이 더해진 값이다.

Input : *population, distance_matrix, parameters, velocity, fixed_cost, variable_cost, capacity, penalty_value, time_window, route, real_distance_matrix, fleet_available, fleet_available_no_fixed_cost*
Output : *Individual 개체로 이루어진 Population과 각 개체의 비용(cost) list*

1. 개체의 각 서브루트에 대해 거리(evaluate_distance), 시간(evaluate_time), 용량(evaluate_capacity) 평가한다.
2. 시간 제약, 용량 제약에 따른 페널티 추가한다.
3. 각 경로 비용 계산 : 경로에서 사용한 차량이 과거에 이미 사용된 경우 해당 경로의 고정 비용을 0으로 설정한다.
4. 경로 비용과 페널티를 더한 값을 비용 list에 추가한다.

Eval.py

Evaluate_distance
특정 subroute의 이동 거리를 계산한다.

Input : *distance_matrix, depot, subroute, parameters*
Output : *경로의 누적 이동 거리 list*

동작:

1. evaluate_subroute(subroute, parameters) 함수를 호출하여 서브루트(subroute)가 방문하는 도시들을 특정한다.
2. subroute_i와 subroute_j를 생성한다. 이들은 각각 출발 지점을 포함하는 서브루트와 종착 지점을 포함하는 서브루트를 나타낸다.
3. 각 지점 쌍의 tuple로 구성된 list인 subroute_ij를 생성한다. 이는 출발 지점과 종착 지점을 연결한다.
4. distance_matrix에서 subroute_ij에 해당하는 거리 값을 추출하여 누적합을 계산한다. 이로써 각 지점 간의 이동 거리를 누적하여 나타낸다.
5. 이동 거리의 시작에 0.0을 추가하여 출발 지점부터의 거리를 포함한다.
6. 계산된 누적 이동 거리 리스트를 반환한다.

Evaluate_time
경로의 각 지점에서의 대기 시간(wait), 이동 시간(time), 그리고 해당 일자(day_num)를 계산한다. 시간 윈도우 내에서의 이동 및 대기를 고려한 경로를 찾을 수 있게 해준다.

Input : *distance_matrix, parameters, depot, subroute, velocity*
Output : *wait, time, day_num*
동작 :

1. evaluate_subroute(subroute, parameters) 함수를 호출하여 subroute가 방문하는 도착지 특정한다.
2. time window 및 하차시간을 parameters에서 불러와 tw_early, tw_late, tw_st 리스트를 만든다.
3. 출발 지점(depot)을 포함하는 subroute를 생성한다.
4. 서브루트를 이용하여 시간 윈도우 및 시간 관련 매개변수를 각 하차지에 매핑한다.
5. 대기 시간(wait)과 이동 시간(time)을 초기화한 후, 각 지점에서의 시간 계산을 수행한다.
   - 이동 거리에 따른 시간 증가 (time[i])
   - 시간 제약에 따른 대기 시간 계산 (wait[i])
   - 시간 제약으로 인한 하차 일자(day_num) 결정
   - 시간에 하차시간(tw_st) 추가
   - 다음 지점의 시작 시간에 현재 시간을 복사
6. 초기 지점의 대기 시간과 시간에 각 0을 추가하여 최종적으로 대기 시간 list, 이동 시간list, 그리고 해당 일자를 반환한다.

evaluate_capacity
해당 경로의 각 지점에서의 적재량(capacity)을 계산한다. 용량 제약을 고려한 경로를 찾을 수 있게 해준다.

Input : *parameters, depot, subroute*
Output : *capacity*
동작 :
1. parameters 에서 각 지점의 수요 정보를 가져와 demand list로 저장한다.
2. 출발 지점과 subroute, 종착 지점을 포함하는 subroute_ list를 생성한다.
3. demand[subroute_]를 누적합으로 계산하여 적재량을 나타내는 capacity list를 생성한다.
4. 계산된 적재량(capacity) 리스트를 반환한다.

evaluate_cost
경로의 비용(cost)을 계산한다.

Input : *dist, wait, parameters, depot, subroute, fixed_cost, variable_cost, time_window*
Output : *cost*
동작 :
1. parameters 에서 각 지점의 대기 비용을 가져와 tw_wc로 저장한다.
2. evaluate_subroute(subroute, parameters) 함수를 호출하여 subroute가 방문하는 도착지 특정한다.
3. 출발 지점과 종착 지점을 포함하는 subroute_ 리스트를 생성한다.
4. cost 리스트를 subroute_와 동일한 길이의 리스트로 초기화한다.
5. dist, wait, 및 tw_wc를 이용하여 비용을 계산한다. 비용에는 고정비용과 가변비용, 대기비용이 포함된다. 고정비용은 fixed_cost 값에 따라 상황에 맞게 고려된다.
6. 계산된 비용(cost) 리스트를 반환한다.

Reference

Baker, B. et al, 2003, A genetic Algorithm for vehicle routing problem., Comptuers & Operations Research 30 (2003) 787–800.