

Vehicle Routing Problem With Time Windows

— Računarska Inteligencija —

Anja Cvetković i Stefan Jevtić

Matematički fakultet
Beograd, 2024.

Pregled

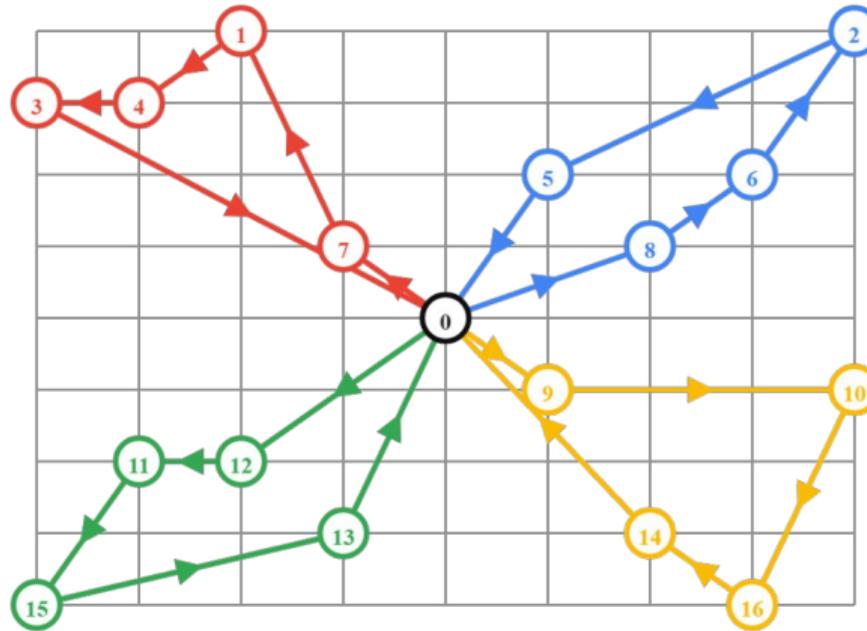
Opis problema

Genetski algoritam

Optimizacija parametara

Opis problema

- Skup gradova V i skup mušterija C
- Ograničenja
- Funkcija cilja



Genetski algoritam

- Pseudokod

Algorithm 1 Genetski algoritam

Create initial population

Evaluate population

while Termination criterion is not met **do**

 Select good individuals for reproduction

 Perform crossover operation on said individuals

 Perform mutation operation on children individuals with probability P_m

 Evaluate new population

end while

- A zpravo...

Genetski algoritam

```
population = [Individual(data, capacity, num_of_vehicles, service_time) for _ in range(population_size)]
new_population = deepcopy(population)

best_solutions = []
for i in range(num_generations):
    population.sort(key = lambda x: x.fitness)
    best_solutions.append(population[0])
    new_population[:elitism_size] = population[:elitism_size]

    for j in range(elitism_size, population_size, 2):
        parent1 = selection(selection_params, population[:elitism_size])
        parent2 = selection(selection_params, population)

        while(parent1 == parent2):
            parent2 = selection(selection_params, population)

        crossover(params, parent1, parent2, new_population[j], new_population[j+1])

        new_population[j].solution = deepcopy(mutation(params, new_population[j], mutation_prob))
        new_population[j+1].solution = deepcopy(mutation(params, new_population[j+1], mutation_prob))

        offset = min(i/400, 0.25)
        insertion_based_repair(new_population[j], offset)
        insertion_based_repair(new_population[j+1], offset)

        new_population[j].fitness = new_population[j].calc_fitness()
        new_population[j+1].fitness = new_population[j+1].calc_fitness()

        new_population = check_and_update_num_of_vehicles(j, j+1, new_population)

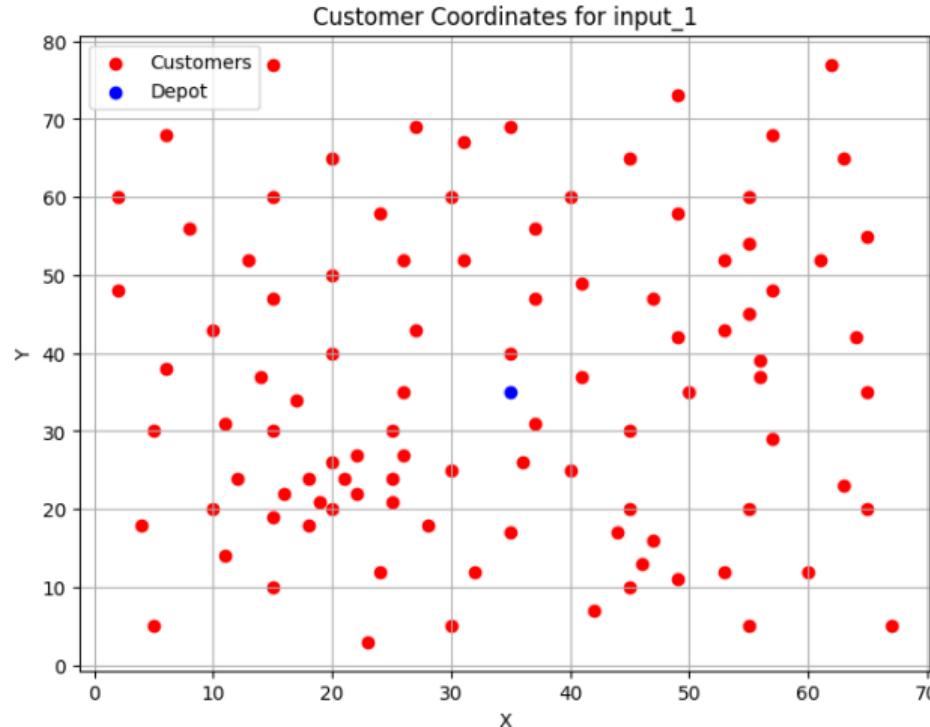
    population = deepcopy(new_population)

print("Number of vehicles after for loop: ", population[0].num_of_vehicles)
return min(population, key = lambda x: x.fitness), best_solutions
```

Parametri

```
POPULATION_SIZE = 300
ELITISM_SIZE = 60
MUTATION_PROB = 0.25
TOURNAMENT_SIZE = 50
NUM_GENERATIONS = 30
CAPACITY = 200
SELECTION = [random_selection, tournament_selection, roulette_selection, rang_selection]
MUTATION = [swap_mutation, invert_mutation, shaking_mutation]
CROSSOVER = [order_crossover, partially_mapped_crossover, best_route_better_adjustment_crossover]
NUM_OF_VEHICLES = 50
SERVICE_TIME = 10
```

Skup podataka

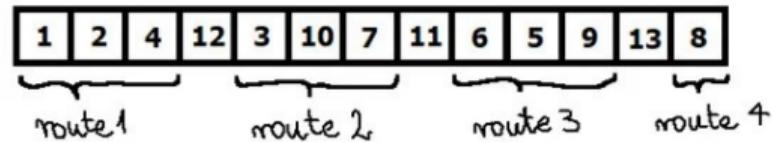


- Marius M. Solomon
- R101

Inicijalizacija hromozoma

Algorithm 2 Inicijalizacija hromozoma

```
1: Create remaining customers list from available customers
2: routes = []
3: last visited = Depot
4: current route = []
5: while Remaining customers list is not empty do
6:   Create list of feasible customers and sort it according to distance from last visited customer
7:   if List of feasible customers is empty then
8:     last visited = Depot
9:     Append current route to routes list
10:    current route = []
11:    continue
12:   end if
13:   if random[0, 1] ≤ randomizing parameter then
14:     Choose a random feasible customer and add it to current route
15:     Remove said customer from feasible customers list
16:     Update last visited customer
17:   else
18:     Choose a nearest customer from feasible customer list and add it to current route
19:     Remove said customer from feasible customers list
20:     Update last visited customer
21:   end if
22: end while
```



- Nearest Neighbour heuristika
- Kako postići diverzifikaciju inicijalne populacije?

Inicijalizacija hromozoma

```
def generate_feasible_routes(self, routes, remaining_cities, prob) -> [[int]]:
    for route in routes:
        while True:
            if not route:
                feasible_cities = self.get_feasible_cities(remaining_cities, 0, 0, self.capacity)
            else:
                _, current_time, remaining_capacity = self.route_fitness(route)
                feasible_cities = self.get_feasible_cities(remaining_cities, route[-1], current_time, remaining_capacity)

            if not feasible_cities:
                break

            if random.random() < prob:
                city_index = random.choice(feasible_cities)[0]
            else:
                city_index = feasible_cities[0][0]

            remaining_cities.remove(city_index)
            route.append(city_index)

            is_unfeasible, _ = self.is_route_unfeasible(route)
            if is_unfeasible:
                route.pop(-1)
                remaining_cities.append(city_index)
                continue
```

Fitness funkcija

```
def route_fitness(self, route) -> (float, float, float):
    if not route:
        return 0, 0, self.capacity

    fitness = 0
    current_time = 0
    previous_city = 0
    remaining_capacity = self.capacity

    for current_city in route:
        current_city_data = self.data[current_city]
        distance = self.distance_between_cities[previous_city][current_city]

        if round(current_time + distance + self.service_time, 3) > current_city_data["due_time"] + self.tolerance:
            fitness += (current_time + distance + self.service_time - current_city_data["due_time"])*self.time_penalty

        if remaining_capacity < current_city_data["demand"]:
            fitness += (current_city_data["demand"] - remaining_capacity)*self.capacity_penalty

        current_time += distance + self.service_time
        previous_city = current_city
        remaining_capacity -= current_city_data["demand"]

    fitness += current_time + self.distance_between_cities[previous_city][0]
    fitness /= len(route) # fitness normalization
    fitness = round(fitness, 3)

    return fitness, current_time, remaining_capacity
```

- Penalizacija
- Normalizacija

Najbolja kombinacija parametara

```
ga_all_combinations.sort(key = lambda x : x[-1][-1].fitness)
best_selection, best_mutation, best_crossover, best_individual_all_comb, best_solutions_all_comb = ga_all_combinations[0]

print('Best selection:', best_selection)
print('Best mutation:', best_mutation)
print('Best crossover:', best_crossover)
print('Best fitness:', best_solutions_all_comb[-1].fitness)

# for s, m, c, p, sol in ga_all_combinations:
#     print(s, m, c, sol[-1].fitness)
```

```
Best selection: random_selection
Best mutation: swap_mutation
Best crossover: best_route_better_adjustment_crossover
Best fitness: 567.904
```

Nasumična selekcija

```
def random_selection(population):
    return random.choice(population)
```

Best Route Better Adjustment Crossover

```
def best_route_better_adjustment_crossover(parent1, parent2, child1, child2):
    # n/2 best from parent1 into first n/2 of child1
    # the rest elements are from parent2

    def create_child(p1, p2, ch):
        p1_routes = p1.get_routes()
        p1_routes.sort(key = lambda route: p1.route_fitness(route)[0])

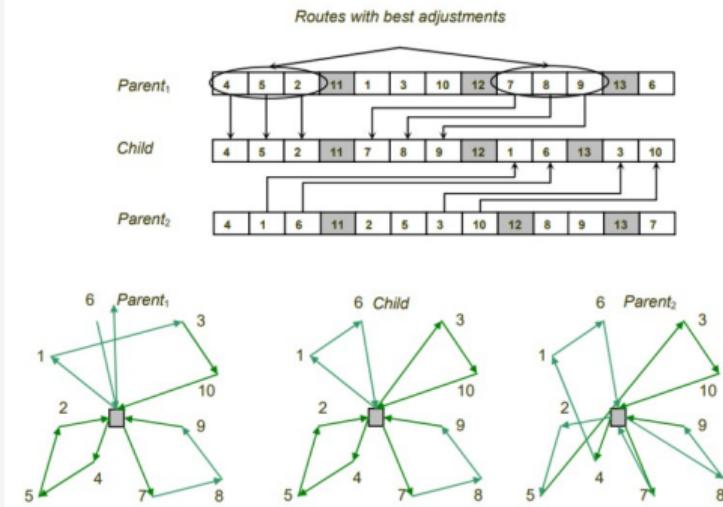
        offspring = []
        route_idx = 0
        while route_idx <= len(p1_routes) / 2:
            offspring.append(p1_routes[route_idx])
            route_idx += 1

        p2_routes = p2.get_routes()

        for route in p2_routes:
            for city in route:
                if city not in offspring:
                    offspring.append(city)

        ch.solution = offspring

    create_child(parent1, parent2, child1)
    create_child(parent2, parent1, child2)
```

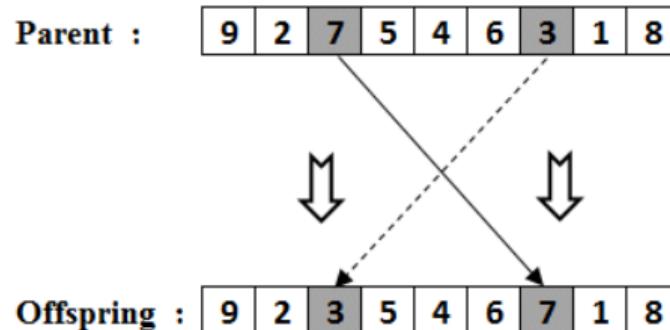


Swap mutacija

```
def swap_mutation(individual, l, r):
    if l == len(individual.solution):
        l -= 1

    if r == len(individual.solution):
        r -= 1

    individual.solution[l], individual.solution[r] = individual.solution[r], individual.solution[l]
    return individual.solution
```



Insertion Based Repair

```
def insertion_based_repair(individual, offset):
    if (individual.is_feasible()):
        return individual

    routes = individual.get_routes()
    cities = range(1, individual.num_of_cities + 1)
    removed_cities_list = []

    # get unvisited cities
    for city in cities:
        if city not in individual.solution:
            removed_cities_list.append(city)

    # get unfeasible routes
    unfeasible_routes = []
    for route in routes:
        is_unfeasible, _ = individual.is_route_unfeasible(route)
        if is_unfeasible:
            unfeasible_routes.append(route)

    # get feasible routes
    feasible_routes = []
    for route in routes:
        if route not in unfeasible_routes:
            feasible_routes.append(route)

    routes = deepcopy(feasible_routes)
```

Insertion Based Repair

```
# remove excess routes
if len(routes) > individual.num_of_vehicles:
    routes.sort(key = lambda route: individual.route_fitness(route)[0])
    while len(routes) > individual.num_of_vehicles:
        route = routes.pop(-1)
        removed_cities_list.extend(route)

# remove unfeasible routes
for i, _ in enumerate(unfeasible_routes):
    unfeasible_routes[i].sort(key = lambda x: (individual.data[x]["ready_time"], individual.data[x]["due_time"]))

# find and eliminate unfeasible cities
while True:
    is_unfeasible, unfeasible_city = individual.is_route_unfeasible(unfeasible_routes[i])
    if not is_unfeasible:
        break

    removed_cities_list.append(unfeasible_city)
    unfeasible_routes[i].remove(unfeasible_city)
    routes.append(unfeasible_routes[i])
```

Insertion Based Repair

```
# first try: insert if possible in existing route
if len(removed_cities_list) > 0:
    for removed_city in removed_cities_list:
        is_inserted = False
        for route_index, _ in enumerate(routes):
            for city_index, _ in enumerate(routes[route_index]):
                route_copy = deepcopy(routes[route_index])
                route_copy.insert(city_index, removed_city)

                is_unfeasible, _ = individual.is_route_unfeasible(route_copy)
                if not is_unfeasible:
                    routes[route_index] = deepcopy(route_copy)
                    is_inserted = True
                    removed_cities_list.remove(removed_city)
                    break

    if is_inserted:
        break
```

Insertion Based Repair

```
# second try: create new routes
if len(removed_cities_list) > 0 and individual.num_of_vehicles - len(routes) > 0:
    new_routes = [[] for _ in range(individual.num_of_vehicles - len(routes))]
    generated_routes = individual.generate_feasible_routes(new_routes,
                                                               removed_cities_list,
                                                               0.25 - offset)
    routes = routes + generated_routes

individual.solution = individual.create_solution(routes)
```

Optimizacija parametara za najbolju kombinaciju operatora

```
POPULATION_SIZE = list(range(500, 1000, 100))
ELITISM_SIZE = list(range(70, 140, 14))
MUTATION_PROB = 0.25
TOURNAMENT_SIZE = 50
NUM_GENERATIONS = list(range(30, 50, 5))
CAPACITY = 200
SELECTION = globals().get(best_selection)
MUTATION = globals().get(best_mutation)
CROSSOVER = globals().get(best_crossover)
NUM_OF_VEHICLES = 50
SERVICE_TIME = 10
```

Best population size: 700
Best number of generations: 35
Best elitism size: 70
Best fitness: 541.695

Optimizacija parametara za najbolju kombinaciju operatara

```
ga_analysis(best_crossover, best_individual_all_comb)
```

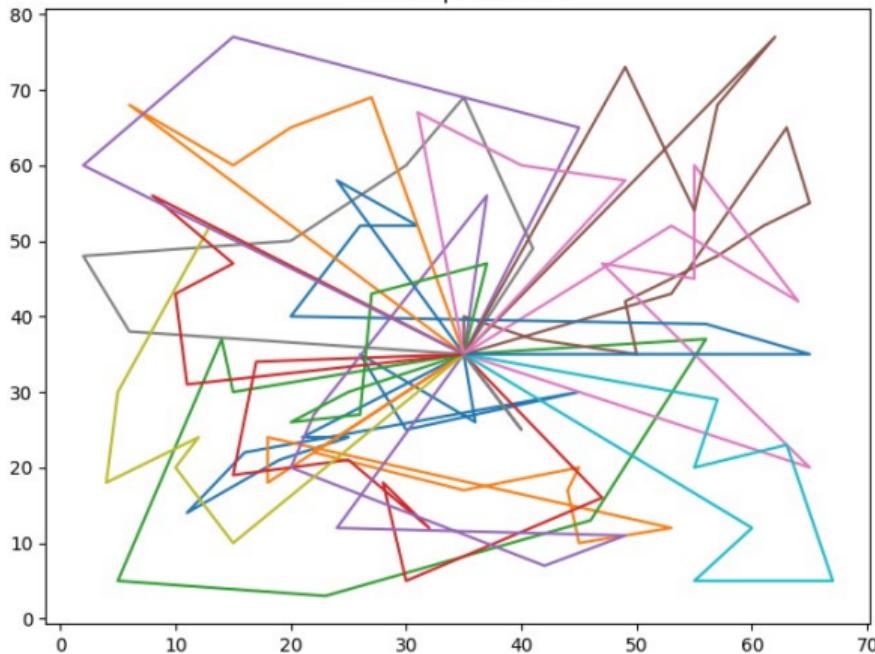
```
best_route_better_adjustment_crossover - is feasible
best_route_better_adjustment_crossover - num of non empty routes: 18
best_route_better_adjustment_crossover - total num of routes: 18
best_route_better_adjustment_crossover - fitness: 567.904
```

```
ga_analysis(best_crossover, best_individual_opt)
```

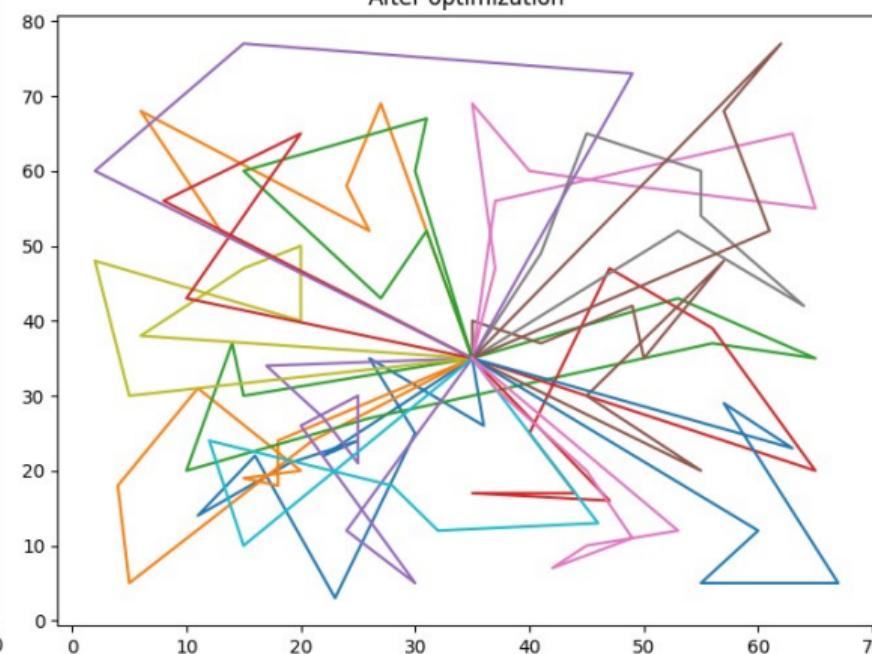
```
best_route_better_adjustment_crossover - is feasible
best_route_better_adjustment_crossover - num of non empty routes: 17
best_route_better_adjustment_crossover - total num of routes: 17
best_route_better_adjustment_crossover - fitness: 541.695
```

Optimizacija parametara za najbolju kombinaciju operatora

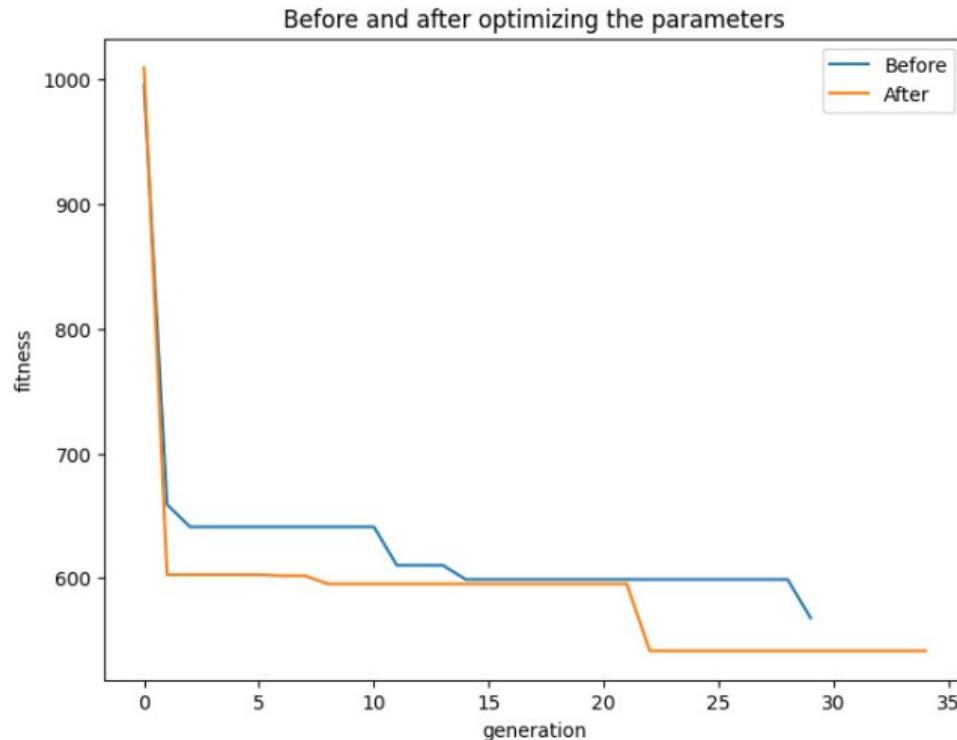
Best Solution Routes best_route_better_adjustment_crossover
Before optimization



Best Solution Routes best_route_better_adjustment_crossover
After optimization



Optimizacija parametara za najbolju kombinaciju operatora



Hvala na pažnji!