PWr : Optimization Methods Theory and Applications - Project

Problem

The goal here is to implement an algorithm to help tourists visit a city. Given a list of places to visit and some contraints we want to provide the optimal tour in the city while satisfying the constraints.

We gather a list of places to visit in the city, the overall grade of these places according to websites like Google Maps or Trip Advisor, and the total number of reviews. We also want to categorize these places in 4 categories :

- Attractions
- Cafes
- Restaurants
- Bar

We then collect a list of constraints from the user, like when and where to start the tour, if he wants to eat lunch at a restaurant or go to the bar at the end of the tour.

The main issue with this problem will first be to find a solution that satisfies the constraints.

Modeling

Problem

The problem is a list of places, a matrix of time to travel between each place, and constraints.

Places

Each place contains the following attributes :

- id
- name
- grade and number of reviews
- average time to visit
- open and close hours
- average price
- location

Travel Time Matrix

To simplify the problem here it's only the time to travel between each location on a straight line at average walking speed. But for future implementation this matrix is intended to be fetched from mapping services like Google Maps.

Constraints

Here are a list of constraints the user can set :

- start location
- end location
- start time
- total duration
- breakfast, lunch, dinner, bar
    - window (when does he want to eat)
    - duration (how long does he want to spend eating)

Example

```
start: Main train station
end: not specified
start time: 7 am
total duration: 15 hours (07h - 22h)
breakfast: no
lunch: yes between 12h and 14h for 1h
dinner: no
bar: yes after 20h
```

Solution

A solution is an ordered list of places. Given this list and to get the detailed tour information (when to be at each place) we use the following algorithm :

```
t = constraints.startTime
for (place in tour) {
  // Take the time to travel to the place
  // If it's the first place we consider previous_place to be the start location
constraint
  t += time_to_travel_from_previous_place

  // Check if it's the breakfast
  if (constraints.breakfast && place.type == cafe && !breakfastDone) {
    if (t < place.open) {
      // Wait for the opening of the place
      t = place.open
    }

    // ... Perform some action on this place

    // Spend time on breakfast
    t += constraints.breakfastDuration

    breakfastDone = true

    // Go to next place
    continue
  }
  // Idem for lunch (type = restaurant), dinner (type = restaurant) and bar
  else if // ....

  // Just an attraction
  else {
    if (t < place.open) {
      // Wait for the opening of the place
      t = place.open
    }

    // ... Perform some action on this place

    // Visit the place
    t += place.timeToVisit

    // Go to next place
    continue
  }

}
```

We use a simple list to keep track of the tour rather than keeping more detailed informations (eg. when should we be at a specific place or if it's the place to take lunch) to simplify operations on the tour (ie. crossover and mutation operators can be simple swaps on the list). The main issue with this approach is that the solution encoding allows non-compliant solutions.

A solution is compliant if it meets the following requirements :

- compliant with the user defined constraints
- tour contains no duplicates
- places are open when visiting them

Repair

We can repair a solution if it's not compliant. The repair process is quite the same than the random generation, except that instead of using shuffle list of places from the problem we try to use the places from the solution, and fallback to the problem places. For instance if the solution has a cafe too late and no restaurant the repair solution will include the cafe at the good time and a random restaurant from the problem. This method discards the attractions that are too late. Before performing the repair we remove the duplicates from the solution.

Fitness

1. Get compliancy of the solution
2. Compute penalty and fitness
3. Get weighted sum of penalty and fitness

1. Compliancy

For each constraints (user and others) we check if the solution is compliant, and for time constraints we get the distance with the constraint. For instance, if the constraints is the lunch before 14h but the lunch ends at 14h15 this constraint will have a value of `-15`. It's possible in the configuration to allow bonuses (ie. positive distances for time constraints)

2. Penalty

Boolean constraints get a flat penalty and distances get weighted. We then sum those penalties.

Configuration

It is possible to tweak the settings of the fitness computation in a `.json` file. You can find an example in `config/default.json`.

Generation

We can generate random test cases to test the optimizers against different problem instances. Those test cases are generated using a different executable and saved in a folder to be used as an input for the optimizers. By doing that we can be sure that the optimizers are compared on a same instance of the problem.

The generation process take the following options :

- city size
- number of places
- number of problems to generate

Optimization

Random

To generate a random solution we use a similar approach than the algorithm used to get informations on the tour. Using this approach we can generate solution almost compliant regarding :

- user defined constraints : start location, eating, tour duration
- tour contains no duplicates But some constraints can be broken :
- places are open when visiting them

Simple GA

Due to lack of time only a simple point crossover and a swap mutation were implemented.

Iteration

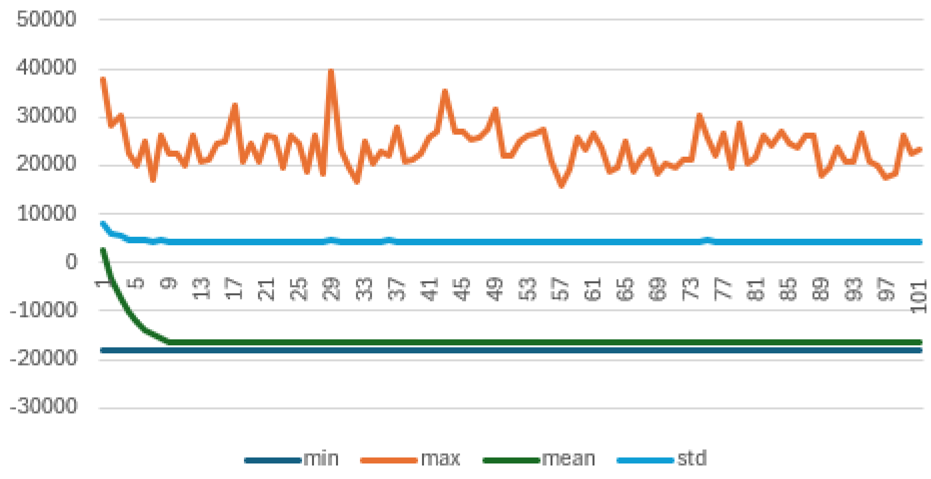
We perform selection and crossover until the population size is reached. Then for each solution we repair it if its not compliant.

Results

Running the GA without the repair function will degrade the solutions in a few generation, resulting to a worse average fitness at the end than at the beginning.

With the repair function the result is a bit better but we still see no improvement across the runs.

Here is an example of a run with the repair function



Here is the best solution found by this run compared to the random strategy with the same number of iterations (`1 000 000`)

Optimizer	Raw Fitness	Penalty	Weighted Sum
Random	`69604`	`-5212`	`43539`
GA	`66007`	`-5329`	`39362`

Random mean fitness for this run : `2683`

Problem parameters :

- City size : 5000
- Number of places : 100

Discussion

First of all, the main issue that I faced during this project is learning C++. For the starting project I choose Python to be able to focus more on how Genetic Algorithms works and not how to implement them. But given the poor performances I got, mainly due to each run taking a lot of time, I decided to implement the main project with C++. This was a challenge for me because allong with designing how to model the problem and generate solutions, I had to learn C++. This was a great experience for me because I learned of lot of things and was able to get faster run speed, but I lost a lot of time on the implementation and sadly I wasn't able to reach all my goals for this project. So if I was given more time to implement this project here are the different things I would try :

- Mutli-objective optimizer
- Linkage learning methods
- More metaeuristic methods

To improve the results of the GA here are some things that could be tested :

- Crossover that don't break too much the solution, for instance only swapping places of the same type
- Same thing for the mutation
- Try some other repair methods