Genetic algorithms and TSP

Intro

- 72 stations
- 5 lines
- 7 intersections
- 124 kilometers of track
- Open 7am-11.59pm

- How do we visit every station in a single day?

  (with least time spent, by physically stepping outside of the train)

- Classic example of a Traveling Salesman Problem
- An upper bound on # routes is 72!
- Hard to brute force an optimal solution

Stations

- Numeric positive id
- Name in Russian
- Name in English
- Line #
- List of neighbours

```
class Station:
    def __init__(self):
        self.id = -1
        self.name = ""
        self.name_eng = ""
        self.line = -1
        self.neighbours = []
```

```
id=1, line=1, name=Devjatkino,
```

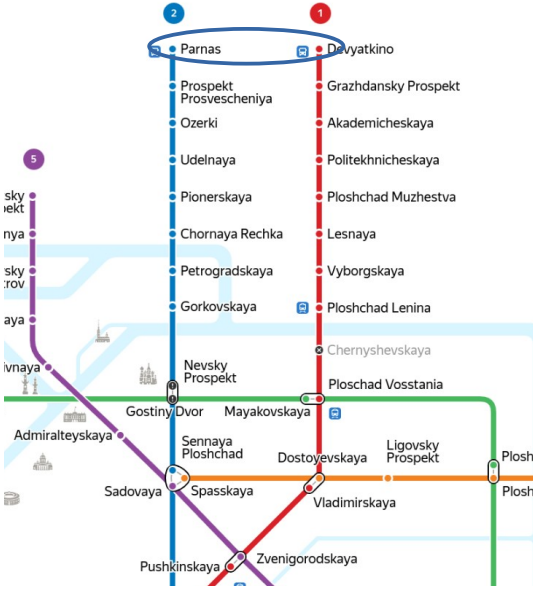
Neighbours

- Relation between two neighbouring stations
- Neighbour Station
- Travel interval
- Travel time
- Travel method (walk/suwbay/taxi)

```
class Neighbour:
    def __init__(self):
        self.station = None
        self.travel_interval = 0
        self.travel_time = 0
        self.travel_method = "subway"
```

End Station Example

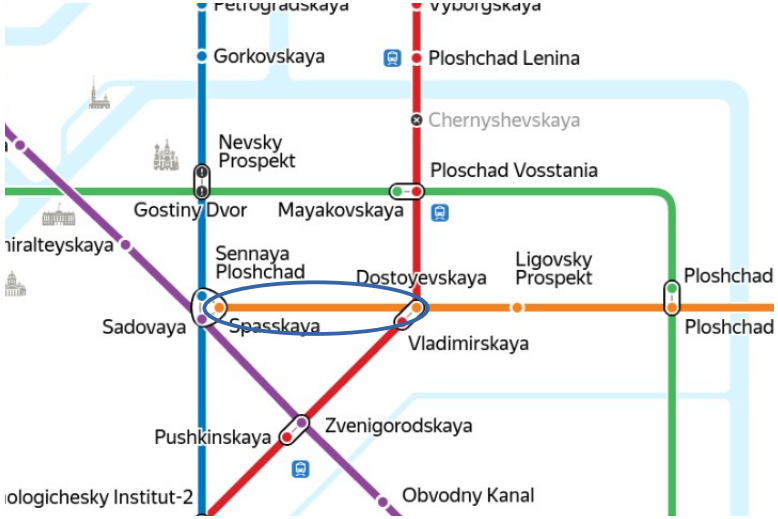

```
id=1, line=1, name=Devjatkino, neighbours:
        interval=3, time=4, method=subway, station ==> id=2, line=1, name=Grazhdanskij prospekt
        interval=3, time=14, method=taxi, station ==> id=20, line=2, name=Parnas
```



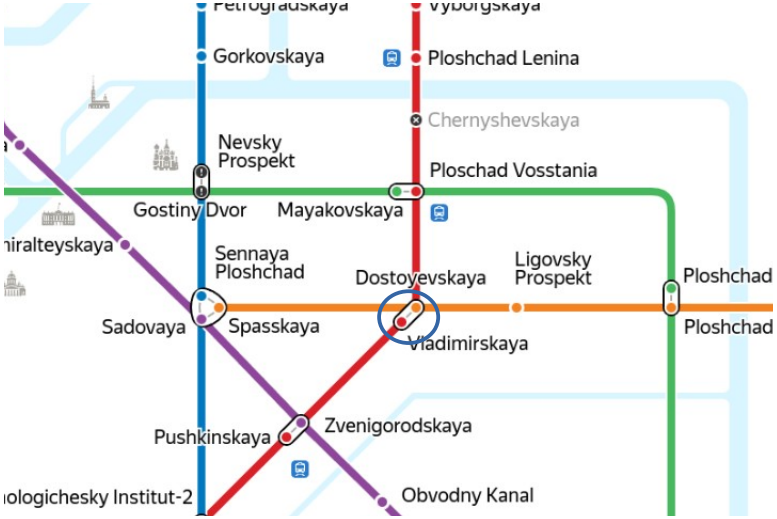

```
id=1, line=1, name=Devjatkino, neighbours:
       interval=3, time=4, method=subway, station ==> id=2, line=1, name=Grazhdanskij prospekt
       interval=3, time=14, method=taxi, station ==> id=20, line=2, name=Parnas
```

Intersection Example


```
id=50, line=4, name=Dostoevskaja, neighbours:
        interval=3, time=2, method=subway, station ==> id=51, line=4, name=Ligovskij Prospekt
        interval=3, time=4, method=subway, station ==> id=49, line=4, name=Spasskaja
        interval=0, time=3, method=walk, station ==> id=11, line=1, name=Vladimirskaja
```




```
id=50, line=4, name=Dostoevskaja, neighbours:
       interval=3, time=2, method=subway, station ==> id=51, line=4, name=Ligovskij Prospekt
       interval=3, time=4, method=subway, station ==> id=49, line=4, name=Spasskaja
       interval=0, time=3, method=walk, station ==> id=11, line=1, name=Vladimirskaja
```




```
id=50, line=4, name=Dostoevskaja, neighbours:
       interval=3, time=2, method=subway, station ==> id=51, line=4, name=Ligovskij Prospekt
       interval=3, time=4, method=subway, station ==> id=49, line=4, name=Spasskaja
       interval=0, time=3, method=walk, station ==> id=11, line=1, name=Vladimirskaja
```

Routes

- Sequence of all 72 stations
- Has a time metric (WTT)
- The time is computed from Dijkstra's algorithm

- Stations are nodes on the graph
- Neighbour relations are edges of the graph
- travel_interval + travel_time are weights

Worst Travel Time Metric (WTT)

- The worst possible amount of time to travel between stations
- Assuming exiting at every station and waiting for a new train
- Soft-upper bound on the full journey → want to minimize

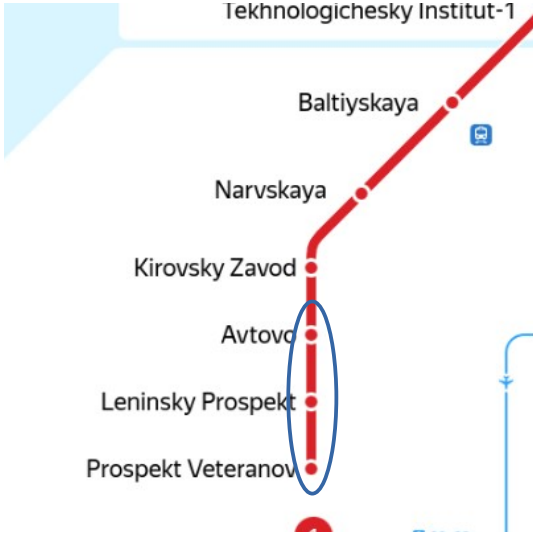
Prospekt Veteranov → Avtovo

- Prospekt Veteranov → Leninsky Prospekt
- travel_interval (3) + travel_time (2) = 5

- Leninsky Prospekt → Avtovo
- travel_interval (3) + travel_time (3) = 6

- Prospekt Veteranov → Avtovo
- 5 + 6 = 11

- WTT: 11 minutes



Obvodny Kanal → Vladimirskaya

- Obvodny Kanal → Zvenigorodskaya
- travel_interval (3) + travel_time (3) = 6

- Zvenigorodskaya → Pushkinskaya
- travel_interval (0, walking) + travel_time (3) = 3

- Pushkinskaya → Vladimirskaya
- travel_interval (3) + travel_time (2) = 5

- WTT = 6 + 3 + 5 = 14 minutes

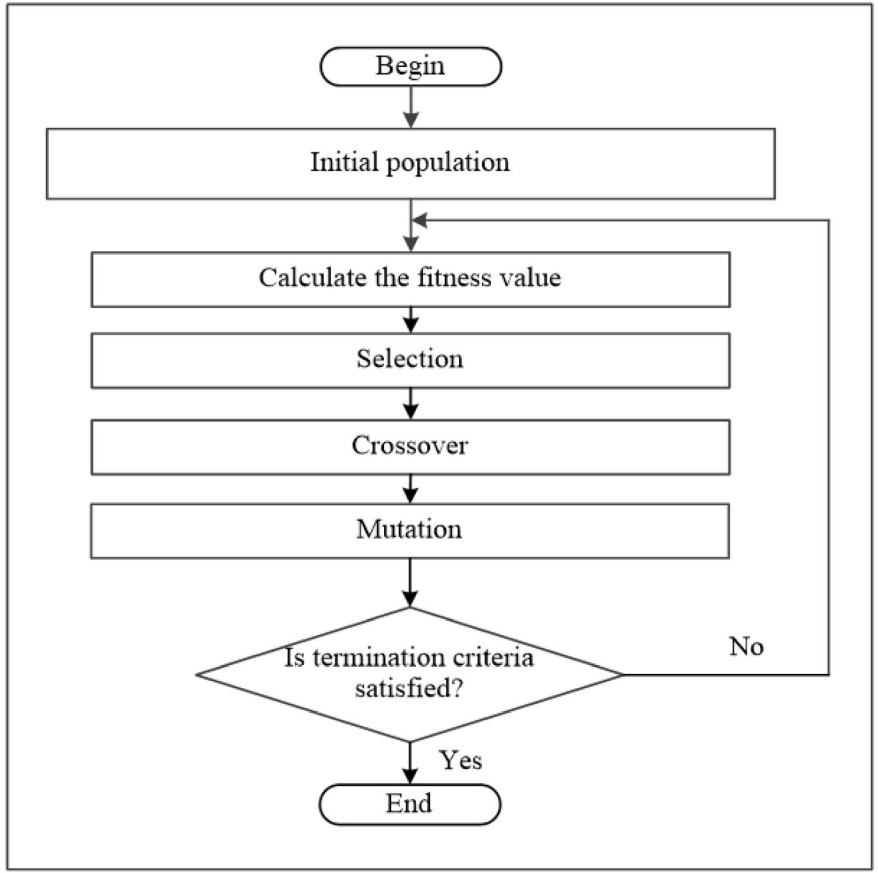
Recap

- Routes are sequences of 72 stations (can be random)
- Each route has a length, defined by a WTT metric
- Goal is to find a route which minimizes WTT
- Brute-forcing is out of the question

- Genetic algorithms!

Genetic Algorithms

- Simulate a real world evolution
- Takes a list of random Routes
- Somehow combine these Routes together
- Produces a new list of “improved” Routes (smaller WTT)
- Repeat N times until WTT is “good enough”


Begin
Initial population
Calculate the fitness value
Selection
Crossover
Mutation
Is termination criteria satisfied?
No
Yes
End

Initial Population

- Pick a fixed number (ie. 1000) of random Routes
- Will probably have horrible WTT (days to ride through)

```
population = [ rm.get_fixed_length_route() for i in range(population_size) ]
```

Fitness Value

- Genetic algorithms try to maximize some value
- We are trying to minimize WTT
- Equivalently, we can be maximizing 1/WTT
- 1/WTT is our fitness

- Smaller WTT ↔ Greater fitness ↔ Better Route

Selection

- Which Routes should contribute to the next generation
- Only the best Routes should contribute!
- One Route can be selected multiple times
- Different methods: tournament, random, truncation, proportion
- Good methods have selection pressure (kind of elitism, where only “good” Routes are selected for crossover)

- Our choice is tournament selection

Tournament Selection

- Sample the current population to get N Routes
- Pick the Route with the highest fitness in the sample
- Add it to the pool
- Repeat until desired pool size is achieved

```python
def tournament_selection(population : list, pool_size: int) -> list:
    mating_pool = []
    for i in range(pool_size):
        tournament = random.sample(population, tournament_size)

        best_chromosome = tournament[0]
        for chromosome in tournament:
            if chromosome.get_fitness() > best_chromosome.get_fitness():
                best_chromosome = chromosome

        mating_pool.append(best_chromosome)

    return mating_pool
```

Crossover aka breeding 😳

- Picking parents, combining their best parts, and producing offspring for the new population
- Important that we only keep the offspring Routes that are valid (ie: have all stations at least once)
- Using special crossover methods for TSP that get around the issue of invalid Routes
- Different methods: ordered, cycle

- Our choice is ordered crossover

Ordered Crossover

- Picks 2 random parents from the pool
- Creates 2 empty offspring Routes
- Defines a cutoff section in the parent Routes
- Adds the cutoff section two respective offspring
- Resolves missing sections in the offspring
- Adds offspring to the new population
- Repeat until desired population count is achieved

Ordered Crossover Example

2.1.2. Order Crossover Operator The order crossover (OX) was proposed by Davis [26]. It builds offspring by choosing a subtour of a parent and preserving the relative order of bits of the other parent. Consider, for example, the two parents tours are as follows (with randomly two cut points marked by "|"):

$$P_1 = (3\ \ 4\ \ 8 \mid 2\ \ 7\ \ 1 \mid 6\ \ 5),$$
$$P_2 = (4\ \ 2\ \ 5 \mid 1\ \ 6\ \ 8 \mid 3\ \ 7). \tag{6}$$

The offspring are produced in the following way. First, the bits are copied down between the cuts with similar way into the offspring, which gives

$$O_1 = (\times\ \ \times\ \ \times \mid 2\ \ 7\ \ 1 \mid \times\ \ \times),$$
$$O_2 = (\times\ \ \times\ \ \times \mid 1\ \ 6\ \ 8 \mid \times\ \ \times). \tag{7}$$

After this, starting from the second cut point of one parent, the bits from the other parent are copied in the same order omitting existing bits. The sequence of the bits in the second parent from the second cut point is "3 → 7 → 4 → 2 → 5 → 1 → 6 → 8." After removal of bits 2, 7, and 1, which are already in the first offspring, the new sequence is "3 → 4 → 5 → 6 → 8." This sequence is placed in the first offspring starting from the second cut point:

$$O_1 = (5\ \ 6\ \ 8 \mid 2\ \ 7\ \ 1 \mid 3\ \ 4). \tag{8}$$

Analogously, we complete second offspring as well:

$$O_2 = (4\ \ 2\ \ 7 \mid 1\ \ 6\ \ 8 \mid 5\ \ 3). \tag{9}$$

```python
def ordered_crossover(mating_pool : list, offspring_len=1000) -> list: # aka (OX)
    offspring = []

    for i in range(0, offspring_len, 2):
        parent_1 = random.choice(mating_pool)
        parent_2 = random.choice(mating_pool)

        route_len = len(parent_1.route_stops)

        offspring_1 = Route(RouteManager.INSTANCE)
        offspring_2 = Route(RouteManager.INSTANCE)

        offspring_1.route_stops = [ None for s in parent_1.route_stops ]
        offspring_2.route_stops = [ None for s in parent_1.route_stops ]

        cutoff_1 = random.randrange(route_len + 1)
        cutoff_2 = random.randrange(route_len + 1)

        cut_start = min(cutoff_1, cutoff_2)
        cut_end = max(cutoff_1, cutoff_2)

        for i in range(cut_start, cut_end):
            offspring_1.route_stops[i] = parent_1.route_stops[i]
            offspring_2.route_stops[i] = parent_2.route_stops[i]

        j_1 = cut_end % route_len
        j_2 = cut_end % route_len

        for i in range(route_len):
            i_conv = (cut_end + i) % route_len

            route_stop_1 = parent_1.route_stops[i_conv]
            route_stop_2 = parent_2.route_stops[i_conv]

            if not offspring_2.has_station(route_stop_1):
                offspring_2.route_stops[j_2] = route_stop_1
                j_2 = (j_2 + 1) % route_len

            if not offspring_1.has_station(route_stop_2):
                offspring_1.route_stops[j_1] = route_stop_2
                j_1 = (j_1 + 1) % route_len

        offspring_1.build_spf()
        offspring_2.build_spf()

        offspring.append(offspring_1)
        offspring.append(offspring_2)

    return offspring
```

Mutation

- Helps in avoiding local optima (stagnation)
- Using the simple inversion mutation
- Pick a random part of each Route in the new population and reverse it

```
def simple_inversion_mutation(offspring: list) -> list: # aka SIM
    mutated_offspring = []

    for o in offspring:
        route_len = len(o.route_stops)

        cutoff_1 = random.randrange(route_len + 1)
        cutoff_2 = random.randrange(route_len + 1)

        cut_start = min(cutoff_1, cutoff_2)
        cut_end = max(cutoff_1, cutoff_2)

        mutated_o = Route(RouteManager.INSTANCE)

        for i in range(cut_start):
            mutated_o.route_stops.append(o.route_stops[i])

        for i in range(cut_end - cut_start):
            mutated_o.route_stops.append(o.route_stops[cut_end - i - 1])

        for i in range(cut_end, route_len):
            mutated_o.route_stops.append(o.route_stops[i])

        mutated_o.build_spf()

        mutated_offspring.append(mutated_o)

    return mutated_offspring
```

- (1 2 3 4 5) → (1 | 2 3 4 | 5) → (1 4 3 2 5)

Combining Everything

- Start with a random population of Routes
- Select good ones for crossover
- Produce offspring Routes by combining random parent Routes
- Mutate the offspring to avoid stagnation
- Set new population to be the new offspring (ie: the new generation)
- Repeat until the result is “good enough”

“Good enough”

- Simulate this process for a large number of generations
- Globally, we keep track of each Route in every generation with lowest WTT
- Once we go through enough generations, we’ll have a good Route with minimal WTT
- Might have to re-run the simulation multiple times to avoid local optimas

Demo

The Final Route

- No taxi
- Enter the subway once, without exiting
- WTT = 522 minutes
- Start at 7am
- Finish at 3:42pm

- (with taxi it is possible to get WTT < 500 minutes)

The Soft-Lower Bound

- Assuming all travel_intervals are 0
- ie: we do not miss any trains
- WTT: 342 minutes

- Expected to take from 5 hours 42 minutes to 8 hours 42 minutes
- We did it in 6.5 hours

Future?

- Other cities (inspired by Moscow Subway Marathon)
- Tracking opening/closing hours of each station
- Tracking train intervals based on the time of day (peak hours, etc)
- More taxi routes (with approximated traffic data)
- Simplify the graph: going to the end of the line can be a single node with combined weight of all nodes. But this will not work with taxi
- Winnipeg bus routes