Simulated Annealing as Traveling Sales Problem

**Algorithm** simulate annealing is

Initialize parameters: set initial temperature and end temperature

1: *t* set initial temperature

2: n number of cities

3: d distance between all the pair

Update\_T

Generate\_InitialSolution()

4: x some initial candidate solution

5: *Best*  x

6: c 0 cities of the closed tour

7: **repeat**

8: c c + 1

9:

10: y Tweak(Copy(x)

11: **for** *n*-1 times **do**

12: **if** Quality(y)>Quality(x) or if a random number chosen from 0 to 1 < e^Quality(x)-Quality(y) **then**

13: x y

14: Decrease t

15: **if** Quality(x) > Quality(*Best*) **then**

16: *Best* x

17: **until** *Best* is the ideal solution, we have run out of time, or t <= 0

18: **return** *Best*

Start

Set Initial parameters, initial value for temperature(t), number of cities(n), and distance between all the pair(d)

Generate an initial solution x randomly

Generate a candidate solution y randomly based on current solution x and a specified neighborhood structure

y > x?

P = exp((-f(y)-f(x)))

*t*

Generate r in [0,1) randomly

*r* < p?

x = y

No

Stop condition of inner loop is met?

No

Yes

Decrease the temperature *t*

Stop condition of inner loop is met?

Yes

Stop

Output the solution x

Genetic Algorithm as Travel Salesman Algorithm

**Algorithm** genetic algorithm is

Initialize parameters:

1: popsize desired population size

2: g gene (x,y coordinates)

3: c a single route satisfying the conditions above

4: p parents

5: mp Mating pool

6: poprank fitness

7: m mutation

8: e elitism

Create\_population

9: popsize [ ]

10:  **for** I in range(0, popsize):

11: population.append(createRoute(cList))

12: **return** population

Determine\_fitness

13: fitnessResults [ ]

14: **for** i in range(0, len(population)):

15: fitnessRessults[i] **Fitness**(population[i]).routeFitness()

16: **return sorted**

Select\_mating\_pool

17: selectionResults [ ]

18: **for** i in range (0, e):

19: selectionResults.append(popRank[i][0])

20: **for** i in range(0, len(popRank) – e.Size):

21: pick 100\*random.random()

22: selectionResults.append(popRank[i][0])

23: **return** selectionResult

Create\_Mattingpool

Breed\_and\_mutate

24. child [ ]

25. childP1 [ ]

26. childP2 [ ]

27: **for** I in range(startGene, endGene):

28: **childP1.append(parent1[i])**

29: childP2 [item for item in parent2 if item not in childP1]

30: child = childP1 + childP2

31:  **return child**

Next\_Generation

32: **repeat**

33:poprank rankrouts((currentGen)

34: selectionResults selection(poprank, elitesize)

35: mattingpool matingpool(currentGen, selectionResults)

36: children breedPopulation(matingpool, eliteSize)

37: nextGeneration mutatePopulation(children, mutationRate)

38: **return** n**extGeneration**

Start

Initialize parameters: population, popSize, eliteSize, mutationRate, generations

Initialize Population

Determine Fitness

Best popsize > Fitness Route

Reiterate through fitness assessment

No

Yes

Best Popsize

Breed

Select Matting Pool

Stop

Mutate