Description of Idea
Please note that I will skip over the explanation for verbose=True sections of the code.

We used a modified Genetic Algorithm to solve this problem. A Genetic Algorithm is a type of heuristic search algorithm replicating the process of genes evolving over generations.

Beginning:
To start our process, we first calculated a "good guess" tour, where we use the "nearest unvisited neighbor" heuristic to construct our initial good guess tour. This good guess is used as the starting gene/tour for all tours in the population (note: that a gene here is a tuple where gene[0] is the tour and gene[1] is the cost of that tour). Then, we mutate all tours except for 1, and perform a modified 2-opt on all mutated genes (2 opt will also recalculate the cost).

While-loop:
We initialize a while-loop which ends when the best population cost has not changed for a specified number of generations, or when the input time-limit is reached. Within the while-loop, we first sort the population in the order of ascending cost so that the best gene of the population will always be at the beginning. Then we see if the best cost stays the same or not. After that, we initialize a new population and pass-over the best elitism=4 genes (from the previous generation to the current generation) unchanged so that we cannot change our best genes.

Breeding:
This covers the first for-loop within the while loop. For all non-elite genes, we pick two parents and "breed" them to make a child which replaces each non-elite gene. To pick parent1, there is an 80% chance that we use function tournyWinner: we randomly select 10 genes from our entire population and return the gene with the minimum cost. Otherwise, there is a 20% chance that we select parent1 from tournElite: same as tournyWinner, but we only select from the elite genes. To select parent2, we always use function tournyWinner. The resulting child is "bred" from parent1 and parent2 by selecting 2 random indexes and populating every element between those two indexes in the child tour with elements between those two indexes from the parent1 tour. The rest of the child tour elements are filled with in-order elements from parent2 that are not already in the child.

Mutation:
Our last for-loop will handle mutating and 2-opting each gene that is not elite (like in the beginning, outside the while-loop).

At the end of our while loop, we change the value of the previous best cost tour, set the current population to be a shallow copy of the new population we just made, and increment the generation number.

Pseudo-code

Helper Methods and Classes:

GeneticUtil object with parameters: node coordinate list, number of seconds until timeout

- method **get_pdist** to get distance between two nodes within this tour
- method **get_cost** to get cost of a tour
- method **mutatedGene** (deprecated) mutates tour by randomly switching two nodes
- method **mutatedGeneLoop**: mutates all indexes of a tour at a mutation rate
- method **createGnome** (deprecated) initializes a random tour
- method **onlyMins** creates initial "good guess" tour using nearest-neighbors heuristic
- method **two_opt** performs a modified 2-opt on a tour and recalculates its cost
  - (optimized efficiency by checking if 2-opt swap would reduce cost, then only performs swap if the cost would go down)

Method **breed** that takes in two parent paths (parent1, parent2)

```
child = new tour
randint1 = random integer
randint2 = random integer
while abs(randint1 - randint2) < 3:
        randint2 = random integer

child[randint1:randint2] = parent1[randint1:randint2]
for each item in parent2:
        if item not in child:
                add to child
return child
```

Method **tournyWinner** that takes in a population and tournament size (tSize)

```
randomly choose 10 genes from population
return gene with best cost of selected genes
```

Method **tournyElite** that takes in a population

```
randomly choose 4 genes from elite genes of population
return elite gene with best cost of selected genes
```

Main Method:

```
popSize = 20
elitism = 4
tournySize = 10
mutationRate = 0.015
counterThresh = 100

Method tspGenetic: takes in GeneticUtil object genU, outputFile
        start = genU.onlyMins()
        costOfStart = start.get_cost()
        population = []

        for i in range(popSize):
                population.append([start, costOfStart])
        for i in range(1, popSize):
                mutate population[i] with genU.mutatedGeneLoop(mutationRate)
                two-opt population[i]

        sameCounter = -1
        previousBestCost = minimum cost of current population
        while sameCounter < 10 and time-limit not exceeded:
                sort population by ascending cost

                if previousBestCost == best current population cost
                        sameCounter += 1
                else:
                        sameCounter = 0

                newPopulation = []
                if elitism > 0:
                        newPopulation's elite genes are equal to old population's elite genes

                for i  from elitism to popSize:
                        if-statement (80% chance):
                                parent1 = tournyWinner(population, tournySize)
                        else (20% chance):
                                parent1 = tournyElite(population)\

                        parent2 = tournyWinner(population, tournySize)
                        child = breed(parent1, parent2)
                        newPopulation.append( [child, child.get_cost()] )
```

<continued on next page>

```
        for newGene in all non-elite newGenes of newPopulation:
            mutate newGene with genU.mutatedGeneLoop(mutationRate)
            two-opt newGene

        perviousBestCost = best cost of old population
        population = newPopulation

    return best population and cost
```

Rationale:

We chose to use a genetic algorithm here as we thought this would be both interesting and challenging to implement. After implementing this algorithm as best we could, without any modifications, we noticed that purely relying on randomness was not good enough to guarantee a solution that was as cost-efficient as we wanted. So, we looked to other sources to help improve upon this base model. First, we implemented "crossover" breeding in our breed function to help preserve sections of previous genes, and we implemented tournyWinner to select the fittest parent from a random mating pool. This helped a little, but then we decided to implement tournyElite to ensure that one parent always came from the elite set of genes. After tinkering with these functions, we found that it was ideal to use tournyWinner for parent1 80% of the time (and use tournyElite the other 20% of the time), while always using tournyWinner for parent 2.

These modifications helped but still did not improve our algorithm enough. Then, we reviewed one of the professor's announcements in which a PowerPoint slide introduced us to the 2-opt function. One can think of this function as a way to check if a tour has intersected itself. If it does, then this function reverses the middle part of the intersection so that the tour no-longer intersects there (and this new, non-intersecting tour is kept if the cost is lower). This addition made our algorithm find very good solutions but did not scale up to more than 100 cities in a reasonable amount of time.

Originally, the 2-opt method was very slow since it did n^2 iterations where each iteration copied the entire tour, reversed a section, then recalculated the new cost of the entire tour to see if the cost went down. When scaling up to 1000 nodes, this meant that each iteration did around 3000 element operations total (which took a long time to run). With our modification, we were able to just check the cost between two edges before the switch, and the two new edges if the switch was used (since 2-opt essentially removes 2 edges and adds 2 edges). If the cost of the new edges is less, then we reverse the edge section and update the tour. This lowers the number of operations per iteration to be ~5 since all we have to do is calculate 4 edge costs and compare. In the end, we are left with a 2-opt'ed tour and we recalculate the cost for this tour.

This change improved the 2-opts speed considerably. Because of this, our genetic algorithm now scaled up to around 1000 nodes and reliably finds a solution that is efficient enough.

Output:

Ten Run Output Cost:
[27750.0, 28044.0, 27750.0, 27750.0, 27603.0, 27750.0, 27750.0, 27750.0, 27750.0, 27750.0]

Ten Runs Output Tours:
for cost 27750.0 (I checked, each output of this cost had same tour):

[1, 2, 6, 10, 11, 12, 15, 19, 18, 22, 23, 21, 29, 28, 26, 25, 27, 24, 16, 20, 17, 14, 13, 9, 7, 3, 4, 8, 5, 1]

for cost 28044.0:

[1, 2, 6, 10, 11, 15, 19, 18, 17, 21, 22, 23, 29, 28, 26, 20, 25, 27, 24, 16, 14, 13, 12, 8, 9, 7, 3, 4, 5, 1]

for cost 27603.0 (only one output of this cost):

[1, 2, 6, 10, 11, 12, 15, 19, 18, 17, 21, 22, 23, 29, 28, 26, 20, 25, 27, 24, 16, 14, 13, 9, 7, 3, 4, 8, 5, 1]

Mean: 27764.7

Standard Deviation: 102.9