CS 325: Project 3

Traveling Salesman Problem

June 9, 2017

Group 24

Wei Gao

Tyler Inberg

Benjamin Fields

I. **Algorithm 1:** 2-Opt Neighbor Search **(Algorithm used in main program)**

Description: There are two parts for this algorithm. The first part is to get the nearest neighbour tour. It constructs a tour by starting with a list of all the cities and their coordinates. The starting city would be the city at the origin. Then, out of the remaining cities, the city nearest in distance to the starting city will be moved in the solution array. Repeating the process, until all cities are moved into the solution array. The second part will be 2-opt algorithm. After generate a tour with the nearest neighbor algorithm, a series of exchanges will be performed to improve the tour. The method for this algorithm is to swap the endpoints of two edges of a tour and see if the resulting tour has a lower distance. The algorithm will examine every two cities taking O(n^2) time to see if they can form a shorter distance.

Discussion: This combination of two algorithm may be much closer to the optimal solution. However, it may take a long time to running the larger number of data.

Pseudocode:

```
Nearest_neighbour(int i, int num, struct city tour)
        mindis = int_max
        if  (i == 0)
                return tour[0]
        if i == size-1
                return tour[i];
        else
                for (j=i; j < size; j++)
                        if mindis >= city_distance(tour[i-1], tour[j])
                                mindis = city_distance(tour[i-1], tour[j]);
                                index = j;
                temp_city = tour[i];
                tour[i] = tour[index];
                tour[index] = tem_city;
                return tour[i]//which is the next nearest city


algorithm_2_opt(tour_nn)
        initiate tour_1
```

```
initiate new_distance;
initiate best_distance = tour_distance(tour_nn, num);
for (int i =0; i<size; i++)
    Swap(tour_nn, tour_1);
    new_distance = tour_distance(tour_1, num);
    if (new_distance < best_distance)
            best_distance = new_distance;
            for(int k=0; k<num; k++)
                    tour_nn[k] = tour_1[k];
```

Performance:

Test Case 1: Number of city: 76

Tour: 132480

Time: 0.02 seconds

Test Case 2: Number of city: 280

Tour: 2923

Time: 1.16 seconds

Test Case 3: Number of city: 15112

Tour: 1765584

Time: 36 hours

II. **Algorithm 2:** Nearest Neighbor

Description: With the use of greedy algorithms in a lot of problems, it made sense to search for an algorithm that was fast but didn't always provide the best result, especially if it could reduce the amount of time it took to solve the algorithm. While the 2-opt integration in the above example works well for providing a more optimal solution, focusing in on the nearest neighbor algorithm itself provides speed as it is the greedy algorithm for problems like the Traveling Salesman Problem. Below is another example of a nearest neighbor algorithm put together by USC Professor: Shang-Hua, Teng in the Department of Computer Science and Mathematics.

Discussion: While we knew that the nearest neighbor algorithm would provide a faster solution, we didn't realize just how much faster it actually was. By taking out the 2-opt portion of our code, we were able to cut the time to solve the third example from 36 hours down to an astonishing 6.1 seconds all while still falling under the 1.25 threshold when compared to the optimal solution. It was also able to solve the competition cases with much greater speed helping test case 6 and 7 to fall under the 3 minute threshold. With that came the tradeoff of less optimal solutions however. Each result calculated by the nearest neighbor algorithm was worse than when the 2-opt portion was included in the code.

Pseudocode:

```
Algorithm NN (P=[p1 p2 p3 ... pn]
    for (int i = 1, i<=n, i++)
       for(int j = 1, j<=n, j++)
         compute d[i,j] = ||pi-pj||
    for(int k = 1, k<=n, k++)
        dist[k] = MAX_INT
        for(int l = 1, l<=n, l++)
          if (k!=j) and d[k,l] < dist[k]
              dist[k] = d[k,l]
              NN[k] = l
    return NN, dist
```

cited from USC Professor: Shang-Hua, Teng, Department of Computer Science and Mathematics

retrieved from www.cs.bu.edu/~steng/teaching/Spring2004/lectures/lecture3.ppt

Performance:

Test Case 1:

Tour: 150393

Time: 0 seconds (Unmeasurable)

Test Case 2:

Tour: 3214

Time: 0 seconds (Unmeasurable)

Test Case 3:

Tour: 1949781

Time: 6.1 seconds

III. **Algorithm 3:** Parallel Genetic Algorithm

Description:

In computer science and operations research, a genetic algorithm is a metaheuristic inspired by natural selection that belongs to the larger class of evolutionary algorithms. Genetic algorithms are commonly used in optimization problems or search problems and use bio inspired functions such as mutation, crossover, and selection.

In the case of the Traveling salesman problem the objective is to find a tour, or in the terms of a genetic algorithm, a chromosome, that has the shortest cost. This is where the selection process comes in. In order for the chromosomes the perform cross over two chromosomes must first be selected. The selection process is complete per a fitness function that defines how fit a chromosome is. The fitness for this problem is the distance of the tour. If two chromosomes/tours have the best fitness then they are selected to perform cross over.

The cross over step is a critical step in Genetic Algorithms. In the case of the TSP, and if probability for cross over is met, the cross over step involves passing a proportion of one parents' traits or cities to an offspring and after which a second parent then passes the rest of the cities. However, the TSP genetic algorithm has a constraint that each city must appear only once, which can be broken in the cross over step. This is avoided using cycle cross over, which involves first assigning the first parents cities to the offspring. The second parent then cycles through each city to determine if it is missing in the offspring. If the offspring does not have the city, then the city is added. This avoids the problem of having duplicate cities in a chromosome/tour.

After cross over there is then a chance for mutation based on a given mutation percentage. Mutation in the TSP problem is the swapping of two cities' position. This still maintains the correctness of the tour as a swap cannot produce duplicates. Once cross over and mutation are complete, then the new population has been created and the cycle repeats for a given number of iterations until no more improvements are made. Additionally, this form of implementation lends itself to parallel computing. Using the OpenMP framework I threaded the program to apply the genetic algorithm on sub populations from the global population. From each local population, the best tour is selected. The best tours from each thread are then compared to get the global best tour.

Discussion:

There has been a fair amount of research on using genetic and evolutionary algorithms for optimization problems, often with promising results. After seeing that research has covered the Genetic Algorithm implementation of the TSA, it was selected as the third choice. Also having some experience with parallel programming, I decided to thread the program using OpenMP. My implementation is a rather naïve genetic algorithm in that it applies a very simple selection, cross over and mutation.

The selection is done by sorting the tours by fitness or distance and then iteratively crossing the top two tours. The cross over is done by selecting a discrete number of cities from each parent from a random starting position and assigning them to the offspring. The rest of the cities are then assigned from the next parent iteratively if the city does not already exist. This is called ordered cross over. The process then repeats until no improvements are seen for a discrete number of iterations. This implementation does not use further optimizations to determine if the algorithm is stuck, or has crossed paths, and because of this no optimal solutions are obtained.

After testing the implementation on each test set, it was verified that the algorithm generated correct solutions per the tsp-verifier.py script, but did not produce optimal answers. With the design of the Genetic Algorithm the answer is highly dependent on the selection and cross over as well as the number in the population. The performance of the program is also dependent on the number in the population and the number of iterations before convergence. For each test case, as the number of cities got larger, it become necessary to lower the size of the population to obtain better performance.

Pseudocode:

The pseudocode for this implementation can be seen below for the Parallel Genetic Algorithm as well as the sub procedures Cross over and Mutate:

Parallel GA TSP (cross over percentage, mutation percentage):

{

Create Answer array to store answer from each thread

Use max number of threads available with OpenMP on system:

{

Create random population of chromosome/tours based on cities

Initialize the iteration counter to check how many iterations have passed

Initialize best tour for tracking

While (numIterations < stopping condition)

Sort population

Cross over

Mutate

Select best population if better than current best

Assign the best seen tour to best array based on the current thread ID

}

Select the best answer from all threads

}

Cross Over (Chromosome1, Chromosome2**):**

{

get a random number between 0-100 to see if cross over happens.

select a random starting position

copy portion of parent1 to offspring1 and copy portion of parent2 to offspring 2

for each position in offspring1

check if empty

if empty assign first position of parent2 that is not in tour

for each position in offspring2

check if empty

if empty assign first position of parent1 that is not in tour

replace old parents with new offspring

}

Mutate(Chromosome**):**

{

Get a random number between 0-100 to see if mutation occurs

Select two random positions

Swap positions

}

Performance:

In running the testcases, the implementation of the parallel algorithm produced correct tours per the tsp-verifier.py script, but failed to find very optimal answers. The implementation needs further optimizations to obtain better answers.

Test Case 1:

Number of Threads: 8

Population Size: 2000

Tour: 443,928

Time: 80.0111 Seconds

Test Case 2:

Number of Threads: 8

Population Size: 500

Tour: 30,962

Time: 19.7064 seconds

Test Case 3:

Number of Threads: 8

Population Size: 6

Tour: 133,586,120

Time: 34.6202 seconds

IV. Final Project Solutions:

tsp_example_1.txt:

Time: 0.02 seconds

Length: 132480

Relation to Optimal Solution: 132480/108159 = 1.225

tsp_example_2.txt:

Time: 1.16 seconds

Length: 2923

Relation to Optimal Solution: 2923/2579 = 1.133

tsp_example_3.txt:

Time: 36 hours

Length: 1765584

Relation to Optimal Solution: 1765584/1573084 = 1.122

(Less Optimal Case) Time: 6.78 seconds

(Less Optimal Case) Length: 1949781

Relation to Optimal Solution: 1949781/1573084 = 1.239

V. Competition Solutions:

Test Case 1:

Time: 0 seconds (smaller than < .01)

Length: 5639

Test Case 2:

Time: 0.03 seconds

Length: 8332

Test Case 3:

Time: 0.49 seconds

Length: 14282

Test Case 4:

Time: 3.78 seconds

Length: 18890

Test Case 5:

Time: 30.63 seconds

Length: 26192

Test Case 6:

Time: 247.06 seconds

Length: 36648

(Less Optimal Case) Time: 0.11 seconds

(Less Optimal Case) Length: 40155

Test Case 7:

Time: 3898.21 seconds

Length: 57669

(Less Optimal Case) Time: 0.72 seconds

(Less Optimal Case) Length: 63220