## 1: The First Problem

**(a) Step1:** Randomly prepare the training and test data.

```
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::shuffle(tmp.begin(), tmp.end(), std::default_random_engine(seed));

int testNum = int(tmp.size() / (1 + ratio));
cout << "Test Number: " << testNum << endl;
cout << "Total Number: " << tmp.size() << endl;
int trainNum = tmp.size() - testNum;
cout << "Train Number: " << trainNum << endl;
ofstream outfile;
outfile.open("../test.txt", ios::app);
for (int i = 0; i < testNum; i++) {
    outfile << tmp[i] << endl;
}

ofstream output;
output.open("../train.txt", ios::app);
for (int i = testNum; i < tmp.size(); i++) {
    output << tmp[i] << endl;
}
output.close();
```

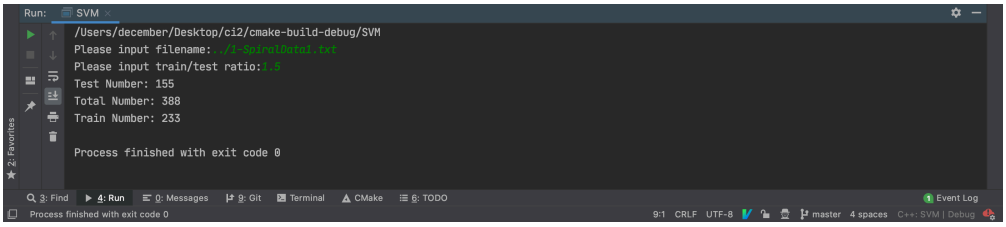


Figure 1: Data1

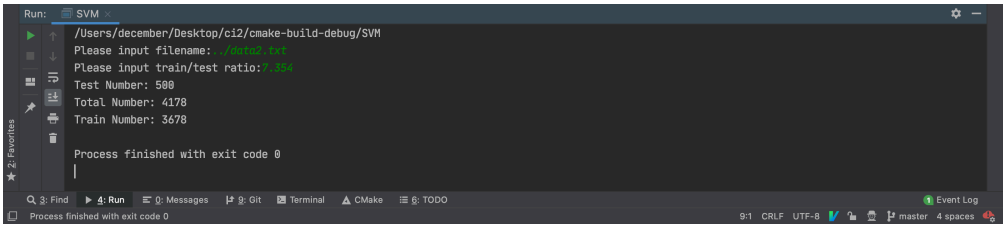


Figure 2: Data2

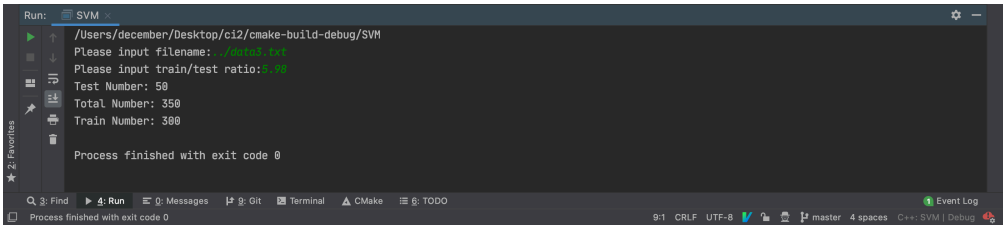


Figure 3: Data3

**(b) Step2:** Adjust SVM parameters and test the best accuracy.

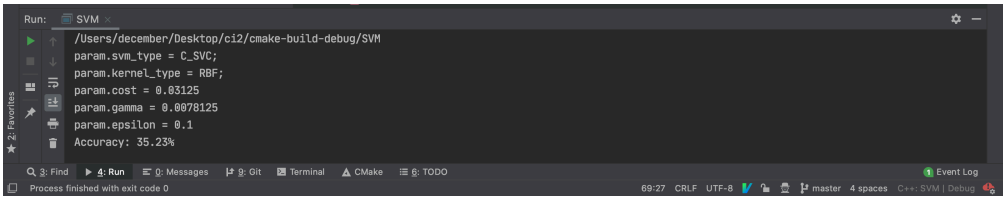


Figure 4: Adjust parameters on data1

For dataset 2, the effect of optimization is still not ideal.


Figure 5: Adjust parameters on data2

```
nSV = 1502, nBSV = 1502
*
optimization finished, #iter = 779
nu = 0.935736
obj = -1558.000000, rho = -1.000000
nSV = 1558, nBSV = 1558
Total nSV = 2309
*
optimization finished, #iter = 751
nu = 0.917532
obj = -1502.000000, rho = 1.000000
nSV = 1502, nBSV = 1502
*
optimization finished, #iter = 751
nu = 0.981699
obj = -1502.000000, rho = 1.000000
nSV = 1502, nBSV = 1502
*
optimization finished, #iter = 779
nu = 0.935736
obj = -1558.000000, rho = -1.000000
nSV = 1558, nBSV = 1558
Total nSV = 2309
*
optimization finished, #iter = 750
nu = 0.916870
obj = -1500.000000, rho = 1.000000
nSV = 1500, nBSV = 1500
*
optimization finished, #iter = 750
nu = 0.981675
obj = -1500.000000, rho = 1.000000
nSV = 1500, nBSV = 1500
*
optimization finished, #iter = 778
nu = 0.935096
obj = -1556.000000, rho = -1.000000
nSV = 1556, nBSV = 1556
Total nSV = 2306
Cross Validation Accuracy = 36.6823%
```

Figure 6: Adjust parameters on data2

```
▲ Desktop/ci2/libsvm ./svm-predict ../test2.txt train2.txt.model ../output.tx
t
Accuracy = 39% (195/500) (classification)
```

Figure 7: Adjust parameters on data2

When optimizing dataset 3, it is found that there is no obvious difference between the linear and rbf kernels, but the linear speed is faster.

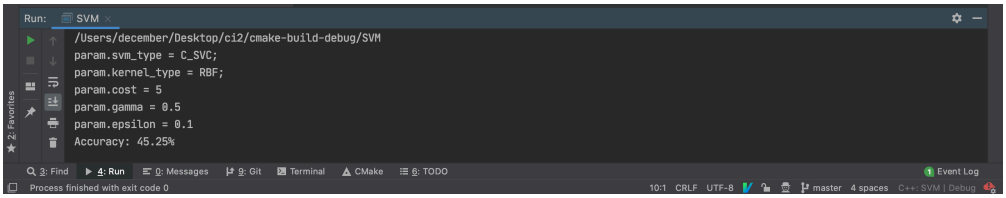


Figure 8: Adjust parameters on data3

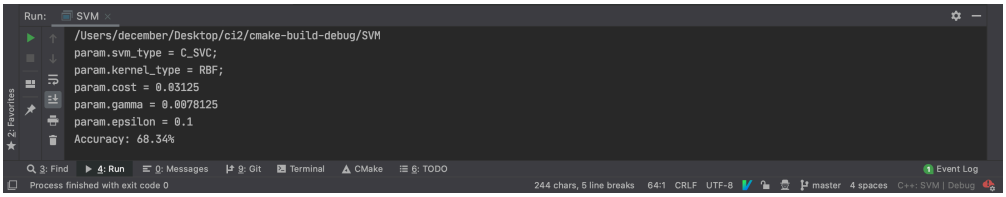


Figure 9: Adjust parameters on data3


Figure 10: Adjust parameters on data3

## 2: The Second Problem

**(a) Step1:** Implement four functions

```
void InitPop(int ***CrntPop, int ***NextPop, int **Fitness, int **BestMember, doub
    int i, j, t, temp;
    srand(Seed);
    *CrntPop = new int *[cPopSize];
    *NextPop = new int *[cPopSize];
    for (i = 0; i < cPopSize; i++) {
        (*CrntPop)[i] = new int[cIndividualLength];
        (*NextPop)[i] = new int[cIndividualLength];
    }
    *Fitness = new int[cPopSize];
    *rFitness = new double[cPopSize];
    *BestMember = new int[cIndividualLength];
    if (Fitness == NULL || BestMember == NULL)exit(1);
    for (i = 0; i < cPopSize; i++) {
        for (j = 0; j < cIndividualLength; j++)
            (*CrntPop)[i][j] = j;
        for (j = 0; j < cIndividualLength; j++) {
            temp = RandInt(cIndividualLength);
            t = (*CrntPop)[i][j];
            (*CrntPop)[i][j] = (*CrntPop)[i][temp];
            (*CrntPop)[i][temp] = t;
        }
    }
}
```

```
int EvaluateFitness(int *Member) {
    //Evaluate the fitness function from distance table
    int p1, p2;
    int TheFitness = 0;
    for (int i = 1; i < cIndividualLength; i++) {
        p1 = Member[i];
        p2 = Member[i - 1];
        TheFitness += lookupTable[p1][p2];
    }
    return (TheFitness);
}

void Crossover(int *P1, int *P2, int *C1, int *C2) {
    int i, Left, Right;
    switch (CrossoverType) {
        case eRandom: // swap random genes
            for (i = 0; i < cIndividualLength; i++) {
```

```
\question{2}{The Second Problem}

\part{a} \one Implement four functions
\begin{lstlisting}
void InitPop(int ***CrntPop, int ***NextPop, int **Fitness, int **BestMember, doub
    int i, j, t, temp;
    srand(Seed);
    *CrntPop = new int *[cPopSize];
    *NextPop = new int *[cPopSize];
    for (i = 0; i < cPopSize; i++) {
        (*CrntPop)[i] = new int[cIndividualLength];
        (*NextPop)[i] = new int[cIndividualLength];
    }
    *Fitness = new int[cPopSize];
    *rFitness = new double[cPopSize];
    *BestMember = new int[cIndividualLength];
    if (Fitness == NULL || BestMember == NULL)exit(1);
    for (i = 0; i < cPopSize; i++) {
        for (j = 0; j < cIndividualLength; j++)
            (*CrntPop)[i][j] = j;
        for (j = 0; j < cIndividualLength; j++) {
            temp = RandInt(cIndividualLength);
            t = (*CrntPop)[i][j];
            (*CrntPop)[i][j] = (*CrntPop)[i][temp];
            (*CrntPop)[i][temp] = t;
        }
```

```
    }
}

int EvaluateFitness(int *Member) {
    //Evaluate the fitness function from distance table
    int p1, p2;
    int TheFitness = 0;
    for (int i = 1; i < cIndividualLength; i++) {
        p1 = Member[i];
        p2 = Member[i - 1];
        TheFitness += lookupTable[p1][p2];
    }
    return (TheFitness);
}

void Crossover(int *P1, int *P2, int *C1, int *C2) {
    int i, Left, Right;
    switch (CrossoverType) {
        case eRandom: // swap random genes
            for (i = 0; i < cIndividualLength; i++) {
                if (RandInt(2)) {
                    C1[i] = P1[i];
                    C2[i] = P2[i];
                } else {
                    C1[i] = P2[i];
                    C2[i] = P1[i];
                }
            }
            break;
        case eUniform: // swap odd/even genes
            for (i = 0; i < cIndividualLength; i++) {
                if (i % 2) {
                    C1[i] = P1[i];
                    C2[i] = P2[i];
                } else {
                    C1[i] = P2[i];
                    C2[i] = P1[i];
                }
            }
            break;
        case eOnePoint:  // perform 1 point x-over
            Left = RandInt(cIndividualLength);
            if (cDebug) {
                printf("Cut points: 0 <= %d <= %d\n", Left, cIndividualLength - 1);
            }
            for (i = 0; i <= Left; i++) {
                C1[i] = P1[i];
                C2[i] = P2[i];
```

```
            }
            for (i = Left + 1; i < cIndividualLength; i++) {
                C1[i] = P2[i];
                C2[i] = P1[i];
            }
            break;
        case eTwoPoint:  // perform 2 point x-over
            Left = RandInt(cIndividualLength - 1);
            Right = Left + 1 + RandInt(cIndividualLength - Left - 1);
            if (cDebug) {
                printf("Cut points: 0 <= %d < %d <= %d\n", Left, Right, cIndividual
            }
            for (i = 0; i <= Left; i++) {
                C1[i] = P1[i];
                C2[i] = P2[i];
            }
            for (i = Left + 1; i <= Right; i++) {
                C1[i] = P2[i];
                C2[i] = P1[i];
            }
            for (i = Right + 1; i < cIndividualLength; i++) {
                C1[i] = P1[i];
                C2[i] = P2[i];
            }
            break;
        default:
            printf("Invalid crossover?\n");
            exit(1);
    }

    for (i = 0; i < cIndividualLength; i++)
        co[i] = 0;
    co0.clear();
    co2.clear();
    for (i = 0; i < cIndividualLength; i++) {
        co[C1[i]] += 1;
        if (co[C1[i]] == 2)
            co2.push_back(i);
    }
    for (i = 0; i < cIndividualLength; i++) {
        if (co[i] == 0)
            co0.push_back(i);
    }
    int s0 = co0.size();
    for (i = 0; i < s0; i++) {
        C1[co2[0]] = co0[0];
        co2.erase(co2.begin());
        co0.erase(co0.begin());
```

```
    }
    for (int i = 0; i < cIndividualLength; i++) {
        for (int j = i + 1; j < cIndividualLength; j++) {
            if (C1[i] == C1[j]) {
                cout << "C1" << endl;
                system("pause");
            }
        }
    }

    for (i = 0; i < cIndividualLength; i++)
        co[i] = 0;
    co0.clear();
    co2.clear();
    for (i = 0; i < cIndividualLength; i++) {
        co[C2[i]]++;
        if (co[C2[i]] == 2)
            co2.push_back(i);
    }
    for (i = 0; i < cIndividualLength; i++) {
        if (co[i] == 0)
            co0.push_back(i);
    }
    int s2 = co2.size();
    for (i = 0; i < s2; i++) {
        C2[co2[0]] = co0[0];
        co2.erase(co2.begin());
        co0.erase(co0.begin());
    }

    //Remove duplicate cities
    for (int i = 0; i < cIndividualLength; i++) {
        for (int j = i + 1; j < cIndividualLength; j++) {
            if (C2[i] == C2[j]) {
                cout << "C2" << endl;
                system("pause");
            }
        }
    }
}

void Mutate(int *Member) {
    // We swap two  randomly selected cities in the member
    int num = (int) (cIndividualLength / 50);
    for (int i = 0; i < num; i++) {
        int Pick = RandInt(cIndividualLength);
        int Pick1 = RandInt(cIndividualLength);
        int t = Member[Pick];
```

```
        Member[Pick] = Member[Pick1];
        Member[Pick1] = t;
    }
}
```

**(b) Step2:** Provide $lookup^{nxn}$ table

```
    //initiate the lookupTable
    for (int i = 0; i < cIndividualLength; i++) {
        for (int j = 0; j < cIndividualLength; j++) {
            int x = longitude[i] - longitude[j];
            int y = latitude[i] - latitude[j];
            double w = weightTable[cityType[i] - 1][cityType[j] - 1];
            lookupTable[i][j] = sqrt(x * x + y * y) * w;
        }
    }
```

**(c) Step3:** Roulette Wheel selection

```
    double WorstFitness = -1;
    long sumRFitness = 0;
    for (i = 0; i < cPopSize; i++) {
        if (WorstFitness < Fitness[i])
            WorstFitness = Fitness[i];
        rFitness[i] = Fitness[i];
    }
    for (i = 0; i < cPopSize; i++) {
        rFitness[i] -= WorstFitness;
        sumRFitness += rFitness[i];
    }
    for (i = 0; i < cPopSize; i++) {
        rFitness[i] /= sumRFitness;
    }
```

**(d) Step4:** Experiment and Report

Through the experiment, cCrossoverRate = 0.95, cMutationRate = 0.01 The experimental result is better.

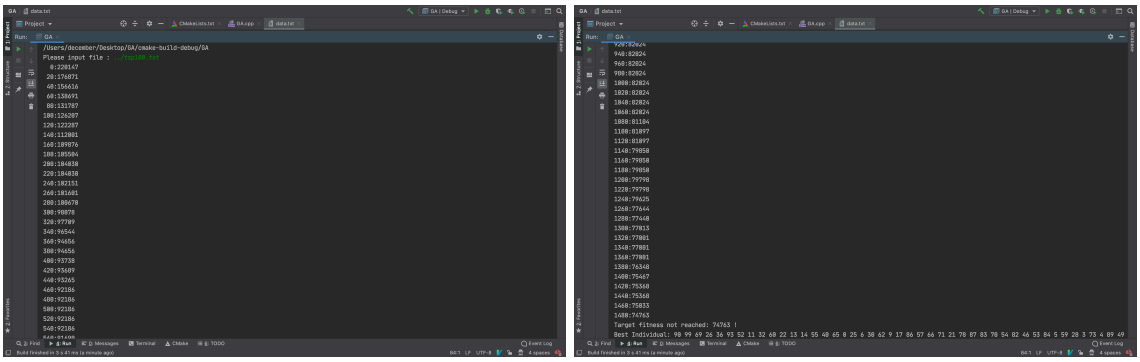

Figure 11: tsp100

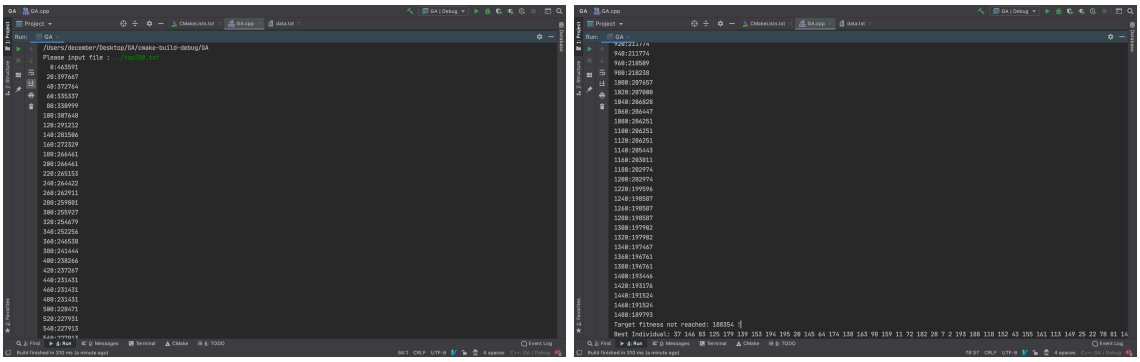

Figure 12: tsp200

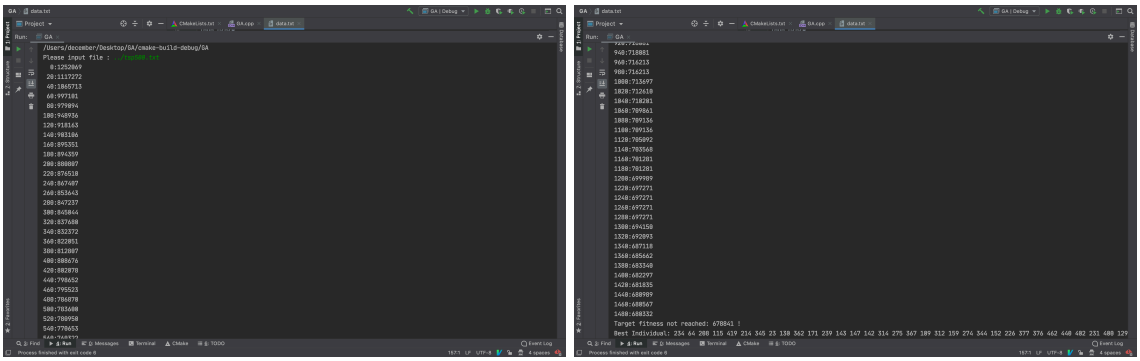

Figure 13: tsp500

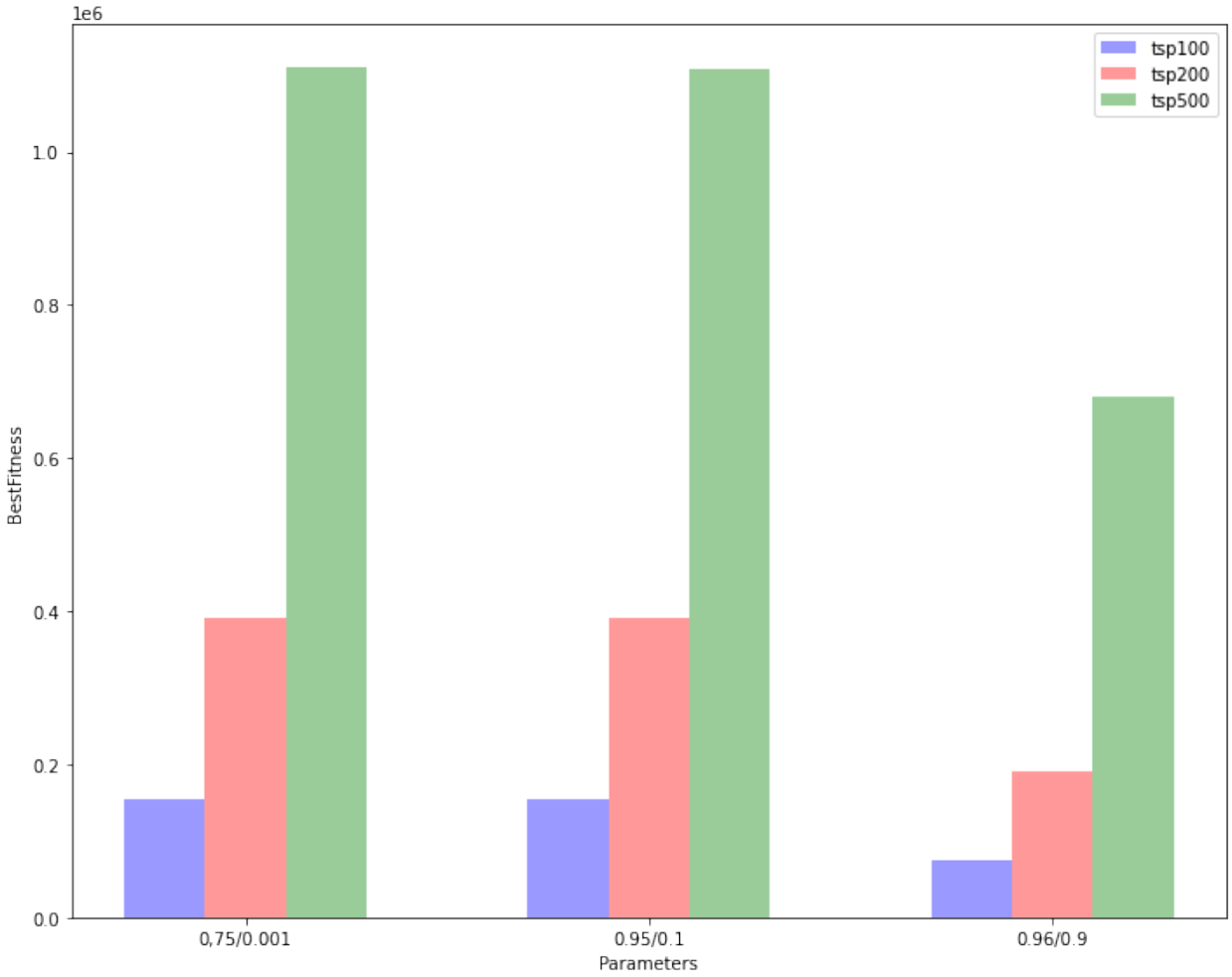


Figure 14: Different Parameters