Nolan Donaldson

CSCI406AB – Algorithms

Professor Mehta

9 September 2019

Project 1 Report

1. I implemented the nearest neighbor algorithm by:
    a. Creating a vector containing point objects in the same order as the input file, a map to record the distances between each point and the points' indexes, a set to record which points have been visited, and a vector of point objects to record the path taken by the algorithm.
    b. Selecting the first point in the vector and making that p_0.
    c. Looping through the vector of points until there are no unvisited points.
        i. Within the loop, the algorithm adds the visited points to the vector of point objects that records the path it takes through the points.
        ii. The distances between each point were recorded in the map along with that point's index in the initial point vector.
        iii. When the distance between p_i and every other unvisited point was added to the map, the shortest distance was the point p_i the algorithm moved to next.
    d. When there were no more points to move to, the algorithm returns to point p_0 and prints out the path it took through the points, adding up the distances between each point it visited.

For the exhaustive algorithm, it uses the vector of point objects from the input file, records the start and ending indexes of that vector, and then recursively swaps the beginning point and ending point. It does this n times for every permutation. When the start and ending index variables are the same, it displays the path it took through the set of points for a single permutation. It does all this n! times for every one of its permutations.

2. My nearest neighbor algorithm has a time complexity of O(n^2). This is because it looks at a point p_i and determines the distance between itself and every other unvisited point in the vector of points. It does this once for every n points in the set, so it costs n x n time. It only needs to print the results once, so that has a constant time complexity.

   My permutations algorithm has a time complexity of O(n!). There are n! permutations for a set of n points, and each permutation has a cost of n which it must do n! times.

3.

Trial	Time Taken, Nearest Neighbor Algorithm (microseconds), n = 100	Time Taken, Exhaustive Algorithm (microseconds), n = 10
1	4001	3237438
2	4001	3247703
3	4000	3613492
Avg.	4000.67	3366211

Trial	Time Taken, Nearest Neighbor Algorithm (microseconds), n = 200	Time Taken, Exhaustive Algorithm (microseconds), n = 9

1	18003	308100
2	18002	313070
3	18004	317027
Avg.	18003	312732.33

Trial	Time Taken, Nearest Neighbor Algorithm (microseconds), n = 1000	Time Taken, Exhaustive Algorithm (microseconds), n = 8
1	544059	33007
2	545126	33008
3	514107	33007
Avg.	534430.67	33007.33

Trial	Time Taken, Nearest Neighbor Algorithm (microseconds), n = 50	Time Taken, Exhaustive Algorithm (microseconds), n = 7
1	1008	4000
2	1000	4001
3	1000	4001
Avg.	1002.67	4000.33

4. The theoretical runtimes are consistent with expectations. With a complexity of $O(n^2)$, the expected runtime with an $n = 50$ is roughly 50^2, or 2500. The actual time it took was roughly half that, which makes sense because the $O(n^2)$ is the absolute worst – case scenario. This was equally true for all other n values tested. It's likely the actual time

complexity was slightly lower because the algorithm ignores points that have already been visited, reducing the complexity of the problem with each point visited.

For the exhaustive algorithm, the time values were also sensible. For example, when an n of 10 was used, the runtime was $3.37*10^6$ microseconds. 10! is $3.63*10^6$. The same is true for the other n values tested. 8! is 40320, and the algorithm took 33007.33 microseconds to complete with n = 8.

More important than the exact values is the rate at which the algorithms scaled with increasing n values. Both algorithms scaled at rates similar to their time complexity, especially the exhaustive algorithm. For example, for the nearest neighbors algorithm, an n of 1000 is expected to take roughly (1000^2 / 50^2 = 400) 400 times more time than an n of 50. The results (~500000 microseconds/~1000 microseconds = 500) support this. For the exhaustive algorithm, an n of 7 took roughly 8 times less time than an n of 8 (~33000 microseconds /~4000 microseconds = 8), further giving credence to the time complexities postulated above.

Appendix A: Code

```
#include <iostream>
#include <chrono>
#include <cmath>
#include <iomanip>
#include <map>
#include <set>
#include <vector>
#include <string>
#include <fstream>
#include <random>
#include <algorithm>
```

```cpp
using namespace std;

/* Author: Nolan Donaldson
 * CSCI 406 - Algorithms
 * Project 1: Timing Project
 * For reference, this was done using Clion.
*/

// Rudimentary 2D point class
class Point {
public:
    Point() {
        _X = 0;
        _Y = 0;
    }

    // Calculates absolute value of the distance between two points
    double distCalc(Point p) {
        double dist = sqrt(pow((_X - p.getX()), 2) + pow((_Y - p.getY()), 2));
        return dist;
    }

    bool operator != (Point &p) {
        return !(_X == p._X && _Y == p._Y);
    }

    int getX() {return _X;}
    int getY() {return _Y;}
    void setX(int X) {_X = X;}
    void setY(int Y) {_Y = Y;}

private:
    int _X;
    int _Y;
};

void print(vector<Point> points) {
    double cost = 0;

    // Calculates cost of path taken
    for (int i = 0; i < points.size() - 1; i++) {
        cost += points[i].distCalc(points[i + 1]);
    }

    cout << setprecision(3) << fixed << cost << endl;
    for (int i = 0; i < points.size(); i++) {
        cout << points[i].getX() << " " << points[i].getY() << endl;
    }
    cout << endl;
}

// Reads in properly formatted files
vector<Point> readInputFile() {
     ifstream input;
     int n;

     // Opens the input file, tests if it works, and reads in the information to a
vector.
     input.open("input.txt");
     if (!input.is_open()) {
         cerr << "Cannot read file input.txt";
         exit(0);
     }

     input >> n;
     vector<Point> inPoints;
     Point temp;

     // Reads in contents of input file to array
     while(!input.eof()) {
         input >> n;
         temp.setX(n);
         input >> n;
         temp.setY(n);
         inPoints.push_back(temp);
     }
     return inPoints;
}

// Solves TSP using nearest neighbor algorithm and prints path
void nearestNeighborSol(vector<Point> points) {
    vector<Point> pathTaken;
    // First point in points is p_0
    Point p_0 = points[0];
    map<double, int> dists;
    set<int> pointsTaken;
    pointsTaken.emplace(0);
    pathTaken.push_back(p_0);
    auto startPt = std::chrono::high_resolution_clock::now();
    // While there are still unvisited points

    for (int i = 0; i < points.size(); i++) {
        // Check if a previously visited point is p_i
        for (int j = 0; j < points.size(); j++) {
            // If it isn't, then add p_i to a set of unvisited points
            if (points[i] != points[j] && !pointsTaken.count(j))
dists.emplace(points[i].distCalc(points[j]), j);
        }
        // Add all the visited points to a vector in the order they were visited in
        pointsTaken.emplace(dists.begin()->second);
        pathTaken.push_back(points[dists.begin()->second]);
        dists.clear();
    }
    pathTaken.pop_back();
    pathTaken.push_back(p_0);
    // TODO: Uncomment this code to print results of nearest neighbor algorithm
    //print(pathTaken);
}
```

```cpp
// Solves TSP by trying all permutations to get optimal path
// Algorithm from: https://www.geeksforgeeks.org/write-a-c-program-to-print-all-
permutations-of-a-given-string/
void exhaustiveSol(vector<Point> points,  int startIndex, int endIndex) {
    // Base case
    if (startIndex == endIndex) {
        // Adds first point back to start of the vector to signify the return back to
p_0
        points.push_back(points[0]);
        // TODO: Uncomment this code to print results of exhaustive algorithm
        //print(points);
    }

    // Recursive case
    else {
        for (int i = startIndex; i <= endIndex; i++) {
            swap(points[startIndex], points[i]);
            exhaustiveSol(points, startIndex + 1, endIndex);
            swap(points[startIndex], points[i]);
        }
    }
}

void randGen(vector<Point>& points, int n) {
    std::default_random_engine generator;
    std::uniform_int_distribution<int> distribution(-1000, 1000);

    // Fills points with n random points
    for (int i = 0; i < n; i++) {
        Point temp;
        int randomP = distribution(generator);
        temp.setX(randomP);
        randomP = distribution(generator);
        temp.setY(randomP);
        points.push_back(temp);
    }
}

int main() {
    vector<Point> points;
    int nNeighbor = 1000;
    int nPermutation = 8;
    // TODO: Uncomment this line of code to use the input file's contents for the
algorithms
    //points = readInputFile();

    // TODO: Uncomment this code AND lines 173-176 to use custom values of n with
randomly generated points
    // randGen(points, nNeighbor);
    auto startPt = std::chrono::high_resolution_clock::now();
    cout << "Results using nearest neighbor algorithm: " << endl;
    nearestNeighborSol(points);
    auto stopPt = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stopPt -
startPt);
```

```
    cout << "Time taken by nearest neighbors algorithm: " << duration.count() << " 
microseconds" << endl;
    cout << endl;

    /*
    points.clear();
    randGen(points, nPermutation);
    */

    cout << "Results for exhaustive solution: " << endl;
    startPt = std::chrono::high_resolution_clock::now();
    exhaustiveSol(points, 0, (points.size() - 1));
    stopPt = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::microseconds>(stopPt - 
startPt);
    cout << "Time taken by exhaustive algorithm: " << duration.count() << " 
microseconds" << endl;
    return 0;
}
```