The Farthest First Algorithm

Amirou Sanoussy

February 2024

1 Introduction

Hardest part of this program for me was interpreting correctly what each component of the parameter did in terms of what data it held and how it held. Following that, I also had trouble with figuring out the logic. However, in the end, my final product for $farfirst.c$ was successful.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vec.h"

int calc_arg_max(double data[], int num_points, int dim, int centers[], int m,
    int max_index = -1;
    double max_distance = -1.0;

    for (int i = 0; i < num_points; i++) {
        double min_distance_to_centers = INFINITY;

        int is_already_center = 0;
        for (int n = 0; n < m; n++) {
            if (i == centers[n]) {
                is_already_center = 1;
                break;
            }
        }
        if (!is_already_center) {
            for (int c = 0; c < m; c++) {
                for (int d = 0; d < dim; d++) {
                    temp_diff[d] = data[i * dim + d] - data[centers[c] * dim + d
                }
                double distance = vec_norm_sq(temp_diff, dim);

                if (distance < min_distance_to_centers) {
                    min_distance_to_centers = distance;
```

```
                }
            }
            if (min_distance_to_centers > max_distance) {
                max_distance = min_distance_to_centers;
                max_index = i;
            }
        }
    }

    return max_index;
}

double calc_cost_sq(double data[], int num_points, int dim, int centers[], int k
    for (int i = 0; i < num_points; i++) {
        distances[i] = INFINITY;
    }

    for (int j = 0; j < num_points; j++) {
        for (int c = 0; c < k; c++) {
            double dist_sq = 0;
            for (int d = 0; d < dim; d++) {
                double diff = data[j * dim + d] - data[centers[c] * dim + d];
                dist_sq += diff * diff;
            }
            if (dist_sq < distances[j]) {
                distances[j] = dist_sq;
            }
        }
    }

    double max_dist_sq = 0;
    for (int i = 0; i < num_points; i++) {
        if (distances[i] > max_dist_sq) {
            max_dist_sq = distances[i];
        }
    }

    return max_dist_sq;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Command usage: %s %s\n", argv[0], "k");
        return 1;
    }
    int k = atoi(argv[1]);
```

```
int num_points, dim;
if (scanf("%*c %d %d", &num_points, &dim) != 2) {
    printf("Error reading file info\n");
    return 1;
}
printf("num_points = %d, dim = %d\n", num_points, dim);

double* data = (double*)malloc(sizeof(double) * num_points * dim);
double* distances = (double*)malloc(sizeof(double) * num_points);
double* temp_diff = (double*)malloc(sizeof(double) * dim);

for (int i = 0; i < num_points; i++) {
    distances[i] = INFINITY;
}

for (int i = 0; i < num_points; i++) {
    if (vec_read_stdin(&data[i * dim], dim) != dim) {
        printf("Error reading file\n");
    }
}

int centers[k];
centers[0] = 0;
for (int m = 1; m < k; m++) {
    centers[m] = calc_arg_max(data, num_points, dim, centers, m, distances,
}

double cost_sq = calc_cost_sq(data, num_points, dim, centers, k, distances);

printf("approximate optimal cost = %f\n", cost_sq);


printf("Approximate optimal centers: ");
for(int i = 0; i < k; i++) {
    printf("%d", centers[i]);
    if (i < k - 1) {
        printf(" ");
    }
}
printf("\n");

free(data);
free(distances);
free(temp_diff);
return 0;
```

}

Coming up with this program, I did a lot of research to find certain methods and keywords to make my program efficient. I also used some of the heap space for the *data*[] array, *distance*[] array that stored the norm squared, and the $temp_diff$[] array that stored the difference of the point and the center points. In retrospect, I think using so much of the heap space may have hindered the efficiency of the program, thus slowing it down a bit and ultimately costing my efficiency. Though it worked!

2 Testing Task

```
(base) afroamir@matrix:~/cmda3634/CSA04$ gcc -o farfirst farfirst.c vec.c -lm
(base) afroamir@matrix:~/cmda3634/CSA04$ time cat cities120.txt | ./farfirst 8
num_points = 120, dim = 2
approximate optimal cost = 7.587744
Approximate optimal centers: 0 24 23 67 103 88 118 4

real    0m0.005s
user    0m0.002s
sys     0m0.004s
```

As it can be seen above, the test was successful and it ran quite quickly. Though relative to the CSA doc, it was slightly slower, which I learned to be quite taxing on the cost and the time belief. It may have to do with my approach to the problem in terms of logic, which has led me to use many loops in this case.

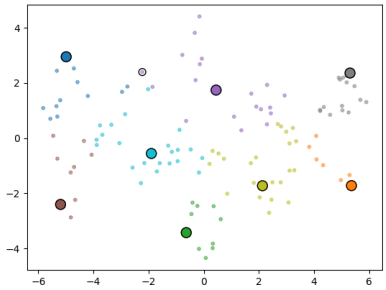

Figure 1: Visualizing farfirst on 120 2-dimensional city locations.

I then followed the testing task following:

```
(base) afroamir@matrix:~/cmda3634/CSA04$ gcc -o farfirst farfirst.c vec.c -lm
(base) afroamir@matrix:~/cmda3634/CSA04$ time cat mnist_test.txt | ./farfirst 16
```

```
num_points = 10000, dim = 784
approximate optimal cost = 8078148.000000
Approximate optimal centers: 0 2462 4086 5494 4999 2337 8254 8061 4294 8266
3841 5141 1377 8570 3817 2151

real 0m6.065s
user 0m6.034s
sys 0m0.050s
```

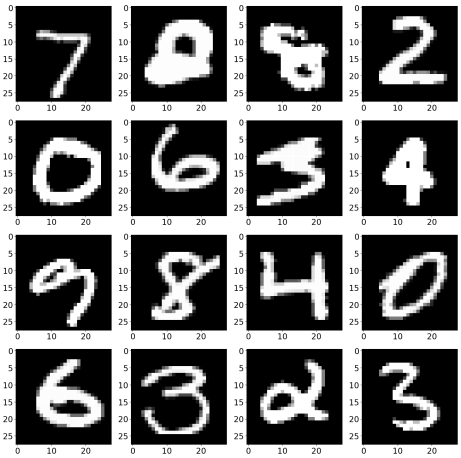

Figure 2: Visualizing farfirst on the MNIST Test Set.

3 Demonstration Tasks

I began by running the following commands which resulted in the following output.

```
(base) afroamir@matrix:~/cmda3634/CSA04$ time cat cities1k.txt  | ./farfirst 16
num_points = 1000, dim = 2
approximate optimal cost = 3.541945
Approximate optimal centers: 0 314 660 40 26 770 562 386 765 846 588 810 191 447 103 410

real 0m0.011s
user 0m0.008s
sys 0m0.003s
```

Which produced the following image

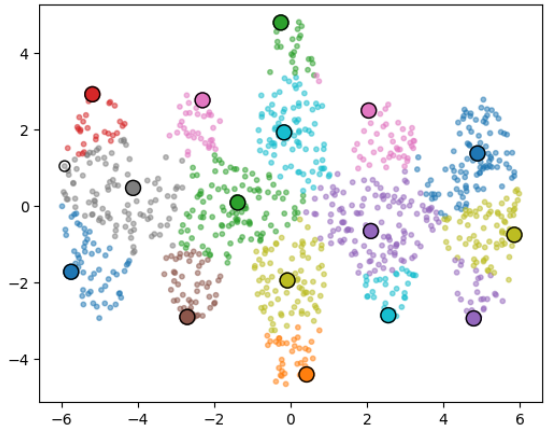

Figure 3: Visualizing farfirst on 1000 2-dimensional city locations 16 centers

I then ran the command again but with k = 64

```
(base) afroamir@matrix:~/cmda3634/CSA04$ time cat cities1k.txt
| ./farfirst 64
num_points = 1000, dim = 2
approximate optimal cost = 0.673040
Approximate optimal centers: 0 314 660 40 26 770 562 386 765 846 588 810 191 447

real    0m0.052s
user    0m0.052s
sys     0m0.000s
```

which produces the following image.

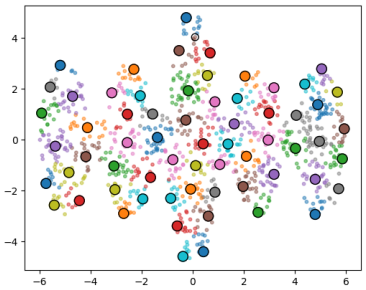

Figure 4: Visualizing farfirst on 1000 2-dimensional city locations 64 centers

4 Further Exploration

When doing my $farfirst_ec.c$ what I did different was that I used $distances[]$ to store the minimum distance of each point to the current set of centers, which I think significantly reduced the number of distance calculations required. So it only updates this array if the distance to the new center is less than the previously stored minimum distance for each point. Another thing I did was that I declared temporary arrays in the functions themselves, such as with $temp_diff[]$. Which I think would help with memory management and speed of the overall program. When updating the $arg_max()$ function for when adding a new center, only calculates the distance from each point to this new center and updates the minimum distance if this new distance is smaller. This was what I was able to figure out from the hint given in the doc. It allowed me to adjust the logic to update distances based on the comparison with the distance to the latest center, rather than recalculating distances to all centers for each point.

```
(base) afroamir@matrix:~/cmda3634/CSA04$ gcc -o farfirst_ec farfirst_ec.c vec.c -lm
(base) afroamir@matrix:~/cmda3634/CSA04$ time cat mnist_test.txt | ./farfirst_ec 64
num_points = 10000, dim = 784
Approximate optimal centers: 0 2462 4086 5494 4999 2337 8254 8061 4294 8266 3841 5141 1377 8

real 0m3.264s
user 0m3.204s
sys 0m0.081s
```

The Following is my code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vec.h"

 int calc_arg_max(double data[], int num_points, int dim, int centers[], int m,
    int max_index = -1;
    double max_distance = -1.0;

    for (int i = 0; i < num_points; i++) {
        double min_distance_to_centers = INFINITY;

        int is_already_center = 0;
        for (int n = 0; n < m; n++) {
            if (i == centers[n]) {
                is_already_center = 1;
                break;
            }
        }
```

```
        if (!is_already_center) {
            for (int c = 0; c < m; c++) {
                for (int d = 0; d < dim; d++) {
                    temp_diff[d] = data[i * dim + d] - data[centers[c] * dim + d
                }
                double distance = vec_norm_sq(temp_diff, dim);

                if (distance < min_distance_to_centers) {
                    min_distance_to_centers = distance;
                }
            }
            if (min_distance_to_centers > max_distance) {
                max_distance = min_distance_to_centers;
                max_index = i;
            }
        }
    }

    return max_index;
}

double calc_cost_sq(double data[], int num_points, int dim, int centers[], int k
    for (int i = 0; i < num_points; i++) {
        distances[i] = INFINITY;
    }

    for (int j = 0; j < num_points; j++) {
        for (int c = 0; c < k; c++) {
            double dist_sq = 0;
            for (int d = 0; d < dim; d++) {
                double diff = data[j * dim + d] - data[centers[c] * dim + d];
                dist_sq += diff * diff;
            }
            if (dist_sq < distances[j]) {
                distances[j] = dist_sq;
            }
        }
    }

    double max_dist_sq = 0;
    for (int i = 0; i < num_points; i++) {
        if (distances[i] > max_dist_sq) {
            max_dist_sq = distances[i];
        }
    }
```

```
    return max_dist_sq;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Command usage: %s %s\n", argv[0], "k");
        return 1;
    }
    int k = atoi(argv[1]);

    int num_points, dim;
    if (scanf("%*c %d %d", &num_points, &dim) != 2) {
        printf("Error reading file info\n");
        return 1;
    }
    printf("num_points = %d, dim = %d\n", num_points, dim);

    double* data = (double*)malloc(sizeof(double) * num_points * dim);
    double* distances = (double*)malloc(sizeof(double) * num_points);
    double* temp_diff = (double*)malloc(sizeof(double) * dim);

    for (int i = 0; i < num_points; i++) {
        distances[i] = INFINITY;
    }

    for (int i = 0; i < num_points; i++) {
        if (vec_read_stdin(&data[i * dim], dim) != dim) {
            printf("Error reading file\n");
        }
    }

    int centers[k];
    centers[0] = 0;
    for (int m = 1; m < k; m++) {
        centers[m] = calc_arg_max(data, num_points, dim, centers, m, distances,
    }

    double cost_sq = calc_cost_sq(data, num_points, dim, centers, k, distances);

    printf("approximate optimal cost = %f\n", cost_sq);


    printf("Approximate optimal centers: ");
    for(int i = 0; i < k; i++) {
        printf("%d", centers[i]);
        if (i < k - 1) {
```

```
            printf(" ");
        }
    }
    printf("\n");

    free(data);
    free(distances);
    free(temp_diff);
    return 0;
}
```