DAA432C

Lab Test II

Minimum Path connecting three points

Prepared By

Section-B - Group -3

Harsh Gupta (ism2017501)

Pratham Singh (irm2017006)

Indian Institute Of Information Technology

Allahabad

2019

Minimum Path connecting three points

Harsh Gupta† , Pratham Singh*
Indian Institute of Information Technology,Allahabad
Allahabad-211012
Email: †ism2017501@iiita.ac.in, *irm2017006@iiita.ac.in,

Abstract—This Paper introduces an algorithm to find the minimum path which connects all the three given points and outputs the count of all the cells that are connected through this path.

Keywords— Minimum path, divide and conquer algorithm, greedy algorithm .

I. Introduction

Finding shortest path between two points is one of the traditional problem in algorithm designing. But here, we have a twist in the given problem, we have to find the minimum path connecting three points of a matrix instead of two, and moreover we also have to output the minimum number of cells that are connected through this path. So, to achieve our goal we have actually used combination of two algorithm one is divide and conquer and second one is greedy. We have divided the problem in two part, first part is to reach startpoint to midpoint and second part is to reach midpoint to endpoint. For solving each part we have used greedy approach for finding the shortest path between the two points.

The minimum number of cells to travel from one point to another can be calculated by the simple approach given below:

Minimum Cells = |x1 - x2| + |y1 - y2|

Let's assume we need to find it for (2,3) and (5,1).
Minimum cells = |2 - 5| + |3 - 1| = 5

So, the minimum no. of cells in the shortest path connecting (2,3) and (5,1) is 5.

Same thing we will do three points.

II. Proposed Method

Here is the detail approach we have used to achieve our result :

1. First, we will select random three points from matrix between which have to find the shortest path.
2. Then, sort the points from the one with minimum row number at first to one with maximum row number at last.
3. Store the minimum column number and maximum column number among these three points in variable **MinCol** and **MaxCol** respectively.
4. Now we will mark all those cells of the matrix which is going to be included in the path.
5. After that, store row number of the middle cell(from sorted cells) in variable **MidRow** and mark all the cells of this **MidRow** from **MinCol** to **MaxCol**.
6. Now our final step will be to mark all the column number of 1st and 3rd cell till they reach **MidRow**.
7. Here, marking means we will store the required cells coordinate in a set. And this set contains the shortest path to follow.

Algorithm: Minimum_cells()

```
Minimum_Cells(vector<pair<int, int> > v)
{

    int col[3], i, j;

    for i⟵0 to 3 {
        int column_number ⟵ v[i].second;

        // Array to store column number of the given cells
        col[i] ⟵ column_number;
    }

    sort(col, col + 3);

    // Sort cells in ascending order of row number
    sort(v.begin(), v.end());

    // Middle row number
    int MidRow ← v[1].first;

   // Set pair to store required cells
   set<pair<int, int> > s;

   // Range of column number
   int MaxCol ← col[2],
     MinCol ← col[0];

   // Store all cells of mid row within column number range

   for i←MinCol to MaxCol {
        s.insert({ MidRow, i });
   }

   for  i←0 to 3 {
      if (v[i].first == MidRow)
          continue;

      // Final step to store all the column number
      // of 1st and 3rd cell upto MidRow

     for j ← min(v[i].first, MidRow) to max(v[i].first,
                                                   MidRow) {
          s.insert({ j, v[i].second });
      }
   }

   return s.size();
}
```

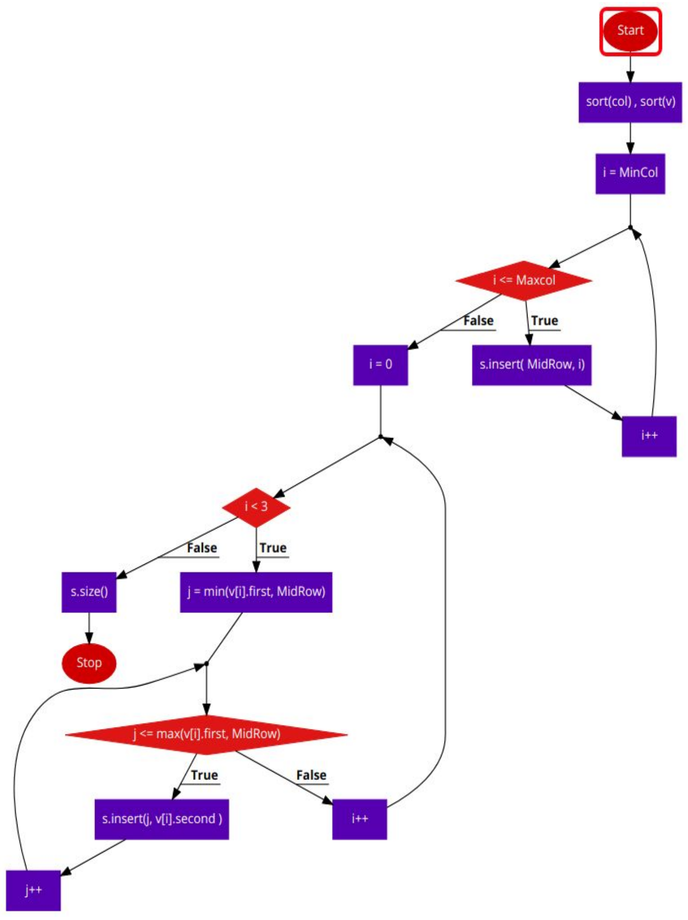
Explanation:

First, we have sorted the cells from the one with minimum row number at first to one with maximum row number at last.

Also, stored the minimum column number and maximum column number among these three cells in variable MinCol and MaxCol respectively.

After that, we have stored row number of the middle cell(from sorted cells) in variable MidRow and marked all the cells of this MidRow from MinCol to MaxCol.

Now in our final step we have marked all the column number of 1st and 3rd cell till they reach MidRow.

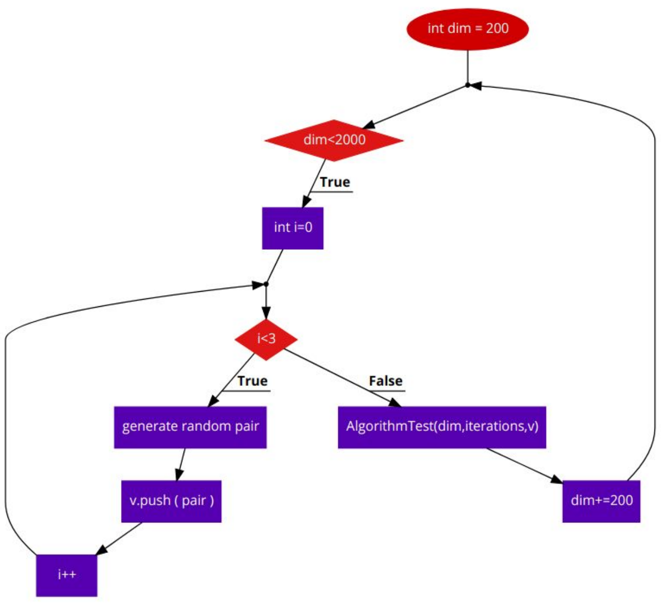


Algorithm: main()

```
main()
{

   FILE* fp;
   fp = fopen("AlgorithmTest.txt", "w+");
   fclose(fp);

    //dim stands for dimension of the matrix.
   for dim ← 200 to 2000{
      vector<pair<int, int> > v;
      for i←0 to 3{
           v.push(pair(random() % dim, random() % dim));
      }
      AlgorithmTest(dim,iterations,v);
   }

   return 0;
}
```

Explanation:

Here, we have performed the algorithm test for different dimensions of the matrix with given no of iterations which helps us to analyse its time complexity and for each case, we have generated random three points. To store the time analysis we have created a log file *AlgorithmTest.txt above.*



Algorithm: AlgorithmTest()

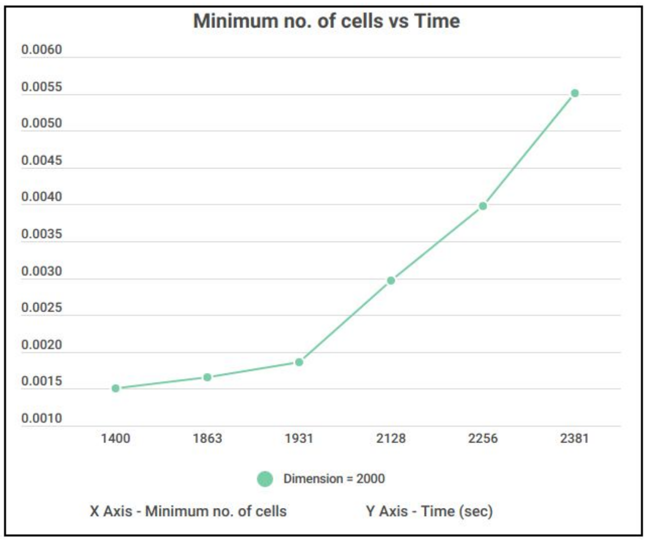
```
AlgorithmTest (dimension, iterations,vector<pair> v){

    latency[iterations];

    for i⟵0 to iterations {
      Time  t0, t1;
      gettimeofday(&t0, 0);
      result ⟵ Minimum_Cells(v);
      gettimeofday(&t1, 0);

      elapsed ⟵ (t1 - t0)
      latency[i] ⟵ elapsed;

       // output the time taken in each iteration.
    }
    // output the minimum cells that are included in the
           shortest path.

    // Analyze Measurements

    sum ⟵ 0.0;
    sumSquared ⟵ 0.0;

    // Statistical analyze
    for i⟵0 to iterations{
      sum ⟵sum +  opmLatency[i];
      sumSquared ⟵ sumSquared + latency[i]$^2$
    }

    mean ⟵ sum / iterations;
    squareMean ⟵ sumSquared / iterations;
    standardDeviation ⟵ sqrt(squareMean - pow(mean, 2.0));

    // output in the console
           “Mean               : mean”
           “Standard Deviation : standardDeviation”
}

// write all the console outputs to the log file.
```

Explanation:
Here, we will run the program for given no of iteration under the fixed dimension and same three coordinates. After the completion, we have taken mean and standard deviation of the latencies to obtain an overview of the time complexity.

III. *Experimental Result*

1. **Minimum no. of cells vs Time:**
   For dimension of matrix = 2000

Minimum Cells	Time (sec)
1400	0.001510s
1863	0.001656s
1931	.001863s
2128	0.002961s
2256	0.003980s

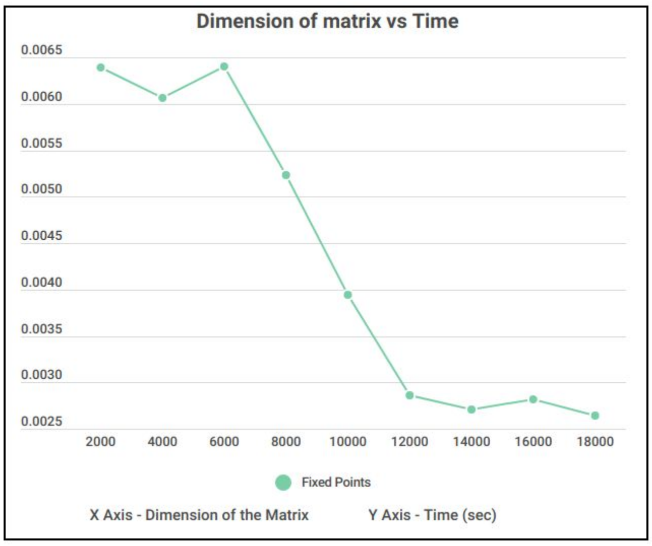


Graph: Minimum no. of cells vs Time(sec)

2. **Dimension of matrix vs Time:**

For fixed points:

Dimension	2000	4000	6000	8000	10000
Time (sec)	.00691	.006061	0.006405	0.005235	0.003941



Graph: Minimum no. of cells vs Time(sec)

IV. Discussion

On theoretical time analysis we find that its worst case time complexity is O(n) where n is the dimension of the matrix.

But from the above experimental result we can say that time complexity actually depends approximately linearly on Minimum no of cells.

If we fix the points, and on changing dimension we observed a strange behaviour that is it decreases with time may be due to internal paging, or cache memory.

From observing with many cases we found that minimum no. of cells doesn't exceeds 2n where n is the dimension. So the space complexity is O(2n).

V. Conclusion

So, here we can conclude that our algorithm have worst case time complexity only O(n) and space complexity O(2n) where n is the dimension of the matrix. Our algorithm have performed very well in the case when dimensions are large because we are considering only those cells which are in between starting and ending point which also decreases space complexity along with time complexity.

VI. References

[1] Author: Wikipedia, online[https://en.wikipedia.org/wiki/Closest_pair_of_points_problem], Available on 21st April 2019.
[2] Author: Egoista, online[https://www.geeksforgeeks.org/minimum-cost-connect-cities/], Available on 21st April 2019.

VII. Appendix

Code link:https://github.com/DreamyPhobic/Daa-Lab-test-2
(Also contains console snapshots, and log file named "AlgorithmTest.txt")