# CS301 Project Report

## HAMILTONAIN PATH PROBLEM

Edin Guso 23435
Güneş Başak Özgün 23521
Begüm Arslanhan 24237
Erkut Gürol 23605
Melih Kurtaran 24186

# Contents

## PROBLEM DESCRIPTION

Traversing a (finite) directed or undirected graph by passing through each vertex exactly once results in a Hamiltonian Path.

A formal definition of the problem can be stated as follows;

Let $G = (V, E)$ be a finite graph with vertex set $V$ and edge set $E$. Hamiltonian path exists if and only if a path $P = (V, E')$ exists such that $E' \subseteq E$ and$|E'| = |V| - 1$. $P = (V, E')$ is defined as a path starting from vertex $u$ and ending at vertex $v$. For every vertex $m \in V - \{u, v\}$, the degree of the vertex is 2 and for $u$ and $v$ vertices have degree of 1.

The problem we are considering in this project is given an n-node undirected graph $G = (V, E)$, is there a simple path of edges in G that contains every node in V, and thus contains exactly n−1 edges?

An example undirected graph input can be seen in Figure 1. The algorithm will try to find a Hamiltonian path, one of such paths is shown in Figure 2. The algorithm will return true if such path exists, otherwise false will be returned.

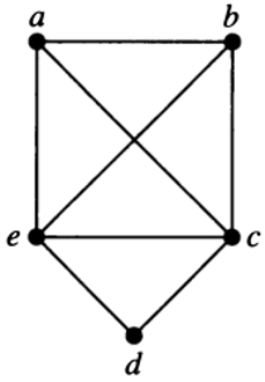



*Figure 1: A possible undirected graph input G= (V, E)*

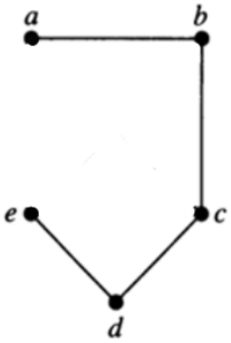



*Figure 2: A Hamiltonian path was found in the input G= (V, E)*

Hamiltonian path problem is NP-complete.

## NP-COMPLETE PROOF

To prove that Hamiltonian Path problem is an NP – Complete problem, following problems should be proven:

1) Hamiltonian Path problem belongs in NP.

2) Hamiltonian Path problem can be reduced to an NP – Hard problem, with the additional note that the viability of reducing a known NP – Hard problem to the Hamiltonian Path problem itself.

### NP Proof

As the first step for the proof of Hamiltonian Path being an NP – Complete problem, proof of the problem for belonging in NP should be conducted. The analysis for the proof of the first section is as follows:

1) Assume that a Hamiltonian Path $G = (V, E)$exists.
2) To prove that every vertex is visited exactly only once, back – track the path by starting from the ending vertex $v_n$ towards the first vertex $v_1$.
3) Since operation back – tracks the path of vertices, its time complexity scales with the number of vertices, which is denoted as $O(V)$.
4) As the operation's complexity is dependent on number of vertices, which scales with the amount of inputs in polynomial scale, Hamiltonian Path problem belongs to the class NP.

Supposedly, a Hamiltonian Path $P = (V, E')$exists. Once the path is back tracked, every vertex in the path can be checked for being visited exactly once in $O(V)$time, which is included in polynomial time operations. Therefore, Hamiltonian Path problem is a problem which belongs to the class NP.

## NP-Hard Proof

To prove Hamiltonian Path problem is NP hard, finding an NP hard problem which can be reduced to Hamiltonian Path is sufficient for completion of the process. For this purpose, the Hamiltonian Cycle problem could be chosen, which is an NP hard problem and can be deterministically reduced to Hamiltonian Path problem.

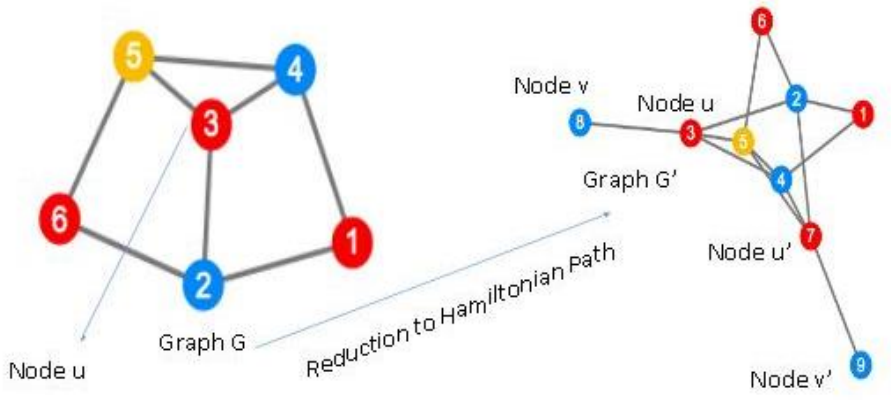



*Figure 3: Graphs that shows reduction from Hamiltonian Cycle to Hamiltonian path problem*

A Hamiltonian cycle is a graph cycle that it visits each node in the graph exactly once except for the starting and ending node which are the same node.

1) Supposing that a graph $G = (V, E)$ containing a Hamiltonian cycle exists, a graph $G' = (V, E')$, which contains a Hamiltonian cycle can be constructed by using the foundry of the same graph.

2) Creating a mirror image of one of the nodes in the graph, along with two arbitrary vertices $v$ and $v'$ which are connected to the chosen vertices and its mirror respectively, one could construct a Hamiltonian path, which starts from either $v$ or $v'$ and ends at $v$ or $v'$.

3) Endpoints are defined as $v$ and $v'$ as these vertices are only connected to the chosen vertex and its mirror image respectively, therefore making it necessary to visit these nodes either at the beginning or at the end of the completion of the tracking of the path.

4) Similarly, to prove that Hamiltonian Path problem can be reduced to Hamiltonian Cycle problem, additional nodes $v$ and $v'$ are removed. Afterwards, it is observed that one could construct two different Hamiltonian paths which start and end at the previously chosen vertex and its mirror with the order being arbitrarily chosen.

5) Moreover, since it was previously assumed that a graph including the chosen vertex formed a Hamiltonian cycle, it could be stated that the graph including the mirror of the point and excluding the chosen point also forms a Hamiltonian cycle, as the mirror point is connected to the same nodes with the originally chosen point itself.

6) Yet, knowing that a graph $G'$, including the point and its mirror forms a Hamiltonian path with no direct connection between the vertex and its mirror, a cycle including all of the points cannot be constructed.

7) Hence, by removing either the chosen vertex or the mirror vertex point, a graph, which contains all of the originally existing points (as the mirror point contains the same connections with the original point with the exact same properties), that constructs a Hamiltonian cycle can be obtained.

Therefore, it is shown through these steps that both Hamiltonian Cycle problem and Hamiltonian Path problem can be reduced to each other, therefore proving that the Hamiltonian Path problem is a problem belonging to NP Hard class.

## ALGORITHM DESCRIPTION

There is not an exact algorithm that solves Hamiltonian path problem in polynomial time. For instance, for a graph having N vertices, it visits all the permutations of the vertices (N! iterations). In each of those iterations it traverses the permutation to see if adjacent vertices are connected or not, so the complexity is O (N * N!).

However, there are some algorithms that finds if the Hamiltonian path exists or not in polynomial time. They will be an approximation to the exact problem.

In this project, we will cover and analyze the performance of the modified version of Kruskal algorithm.

The algorithm basically does the following:

1. Initially, it takes an edge array and the vertex numbers (n) as inputs and creates an array for parent vertices.

2. Then, it chooses a random edge and checks its vertices to check the condition that the selected vertices have not been visited before and also belong to a different set (i.e. have different parents).

3. If those conditions are met, then the selected vertices are marked as visited and their sets are joined until n/2 edges are selected. We mark the nodes visited as True and we keep them to not visit them again.

4. Then, all the edges that haven't been used yet are considered. In addition, we change all the vertices as unvisited to find the new edges to join to the edges found in third step. So, for all vertices we mark the visited list as False. Then, add the two vertices with the condition that the selected vertices have not been visited before and also belong to a different set.

5. If those conditions are met, then the selected vertices are marked as visited and their sets are joined until n-1 edges are reached.

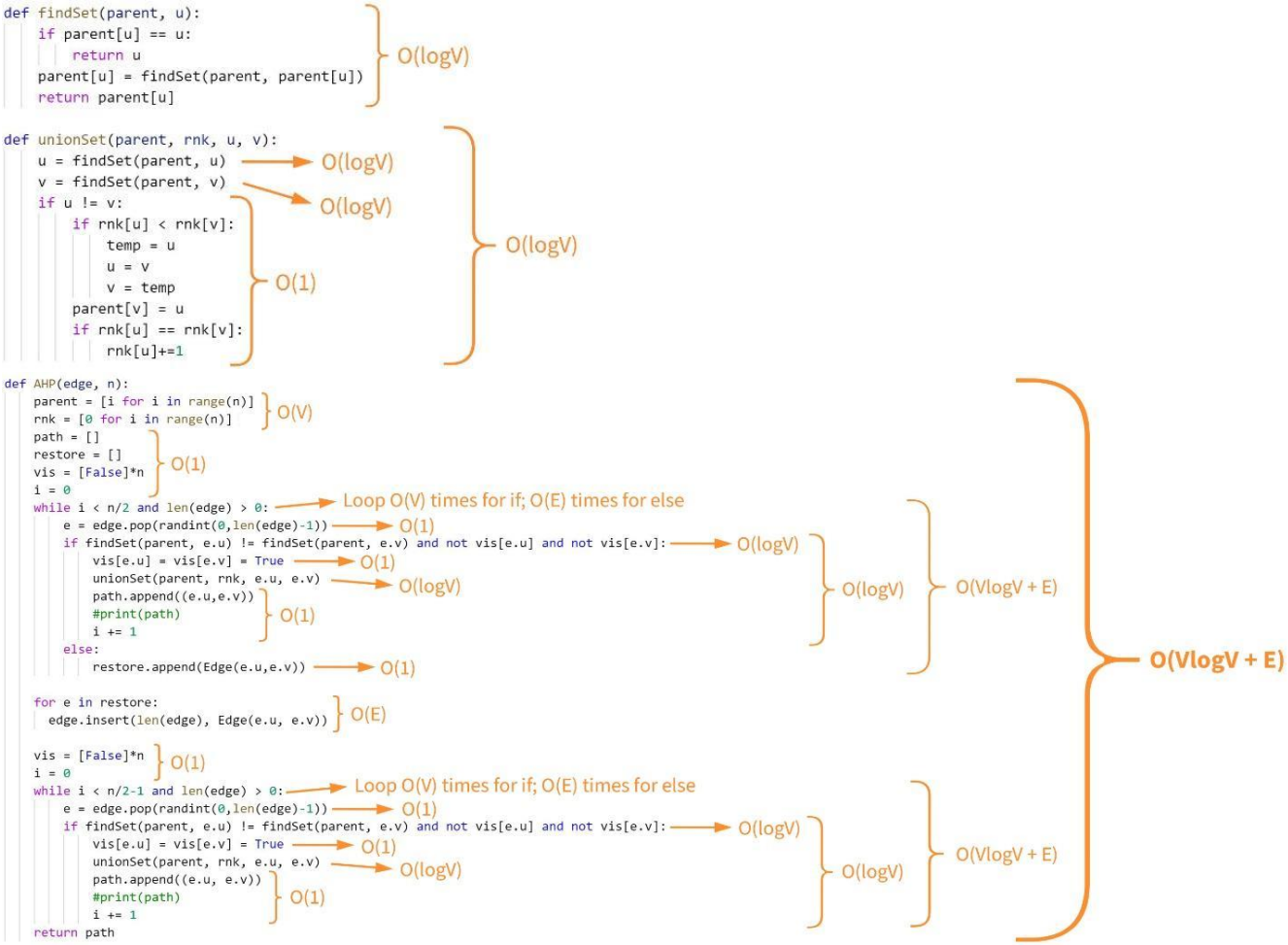
6. If these are all passed, we say that a Hamiltonian path exists, else it doesn't exist.

## ALGORITHM ANALYSIS

The algorithm we have suggested does not guarantee that it will find the Hamiltonian Path if there exists one. Because, the algorithm does not consider every possible permutation of edges. Instead, an edge is randomly selected for examination. If the selected edge fits the criteria, it is added to the candidate Hamiltonian Path. Every edge is considered at most two times (once in the first loop and once in the second loop) and in random orders. This means that only a tiny subset of all possible permutations is considered.

If the algorithm outputs a path of length less than |V|-1, that means that it has failed to find a Hamiltonian Path. If the length of the output path is |V|-1, it is a valid Hamiltonian Path. Also, it is possible to get different decision results for the same input in two different runs since the edge selection is random.

The running time can be analyzed as shown below:



For experimental analysis, we randomly generated several graphs that has 5, 6, 7, 8, 9, 10, 15, 20 vertices as follows.

# EXPERIMENTAL ANALYSIS

## SUCCESS RATES

Success Rates are determined as if our algorithm outputs a path of length less than |V|-1, that means that it has failed to find a Hamiltonian Path, otherwise it succeeds.

| Number of Vertices | 100 Graphs | 1000 Graphs | 3000 Graphs | 5000 Graphs | 10000 Graphs |
|---|---|---|---|---|---|
| 5 | 74.65 | 71.98 | 73.42 | 73.80 | 72.36 |
| 6 | 60.00 | 62.42 | 62.14 | 61.64 | 62.45 |
| 7 | 83.33 | 74.36 | 76.08 | 76.77 | 77.74 |
| 8 | 69.84 | 61.79 | 62.05 | 60.57 | 63.09 |
| 9 | 67.21 | 77.90 | 76.77 | 75.62 | 76.36 |
| 10 | 60.71 | 59.11 | 61.33 | 61.20 | 61.28 |
| 15 | 75.76 | 71.58 | 75.61 | 73.74 | 75.00 |
| 20 | 61.11 | 54.72 | 57.29 | 59.12 | 57.28 |

*Figure 4: Success rates table of randomly generated graphs with different vertices*

It can be observed that as the number of vertices increase, there will be more connections to necessary to get the path and since our algorithm considers an edge at most two times in random orders, a small subset of possible permutations is realized. So, with the increase in number of vertices, probability of finding the correct combination decreases.

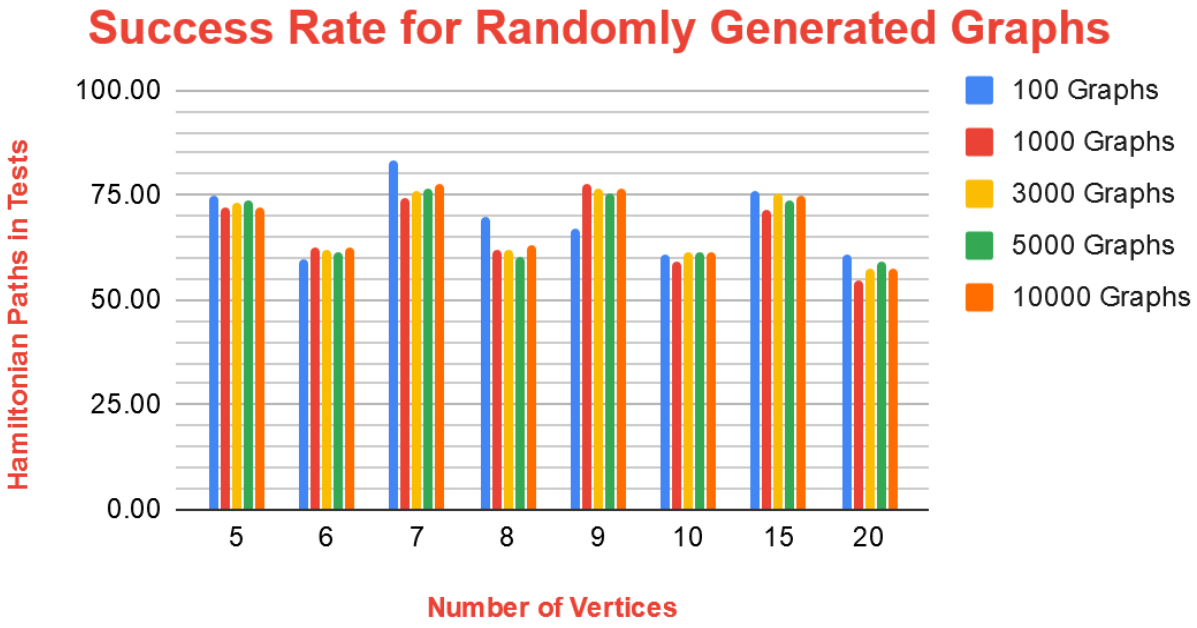



*Figure 5: Success rate graph of randomly generated graphs*

| Number of Vertices | Success Rate |
|---|---|
| 5 | 73.24 |
| 6 | 61.73 |
| 7 | 77.66 |
| 8 | 63.47 |
| 9 | 74.77 |
| 10 | 60.73 |
| 15 | 74.34 |
| 20 | 57.91 |

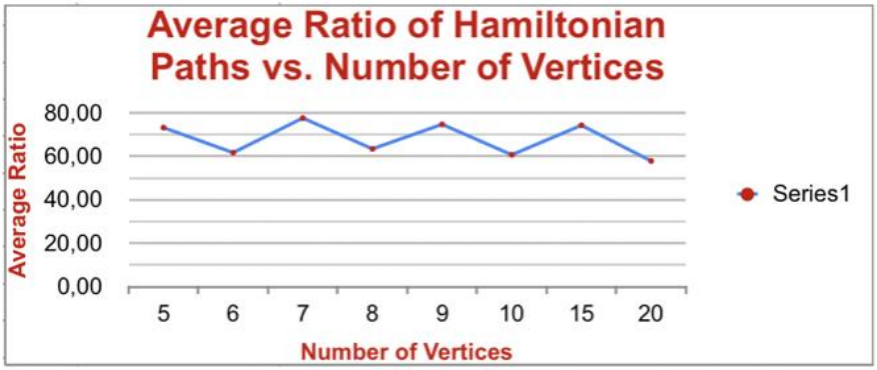



*Figure 6 & 7: Success rate table and graph according to number of vertices*

## RUNNING TIME EXPERIMENTAL MEASUREMENT

Statistics were used to get a better understanding of the experimental running time of the algorithm. We created some functions to interpret the data such as the standard deviation, standard error, mean time and 90% and 95% confidence levels.

```
def calculateSD(data):
    sum = 0
    sD = 0
    for d1 in data:
        sum += d1
    mean = sum / len(data)
    for d2 in data:
        sD += (d2-mean)**2
    sD = math.sqrt(sD/len(data))
    return sD #standart deviation

def calculateStandardError(sD,N):
    return sD / math.sqrt(N)

def getRunningTime(runningTimes):
    sD = calculateSD(runningTimes)
    N = len(runningTimes)
    totalTime = 0
    for r in runningTimes:
        totalTime += r

    m = totalTime / N #sample mean

    tval90 = 1.645 #t-value for 90% confidence
    tval95 = 1.96 #t-value for 95% confidence

    error = calculateStandardError(sD,N)

    upperMean90 = m + tval90*error
    lowerMean90 = m - tval90*error
    upperMean95 = m + tval95*error
    lowerMean95 = m - tval95*error
```

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

$$SE = \frac{\sigma}{\sqrt{n}}$$

$$\bar{x} = \frac{\sum x_i}{n}$$

$$\bar{x} \pm t * SE$$

## 100 RUN PER INPUT SIZE

| Size | Mean Time(s) | Standard Deviation | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 5 | 0.000032496 | 0.000017267 | 0.000001727 | 3.53368e-05 - 2.96560e-05 | 3.58807e-05 - 2.91121e-05 |
| 8 | 0.000054798 | 0.000023685 | 0.000002369 | 5.86943e-05 - 5.09019-05 | 5.94403e-05 - 5.01558e-05 |
| 10 | 0.000084031 | 0.000042455 | 0.000004245 | 9.10144e-05 - 7.70468e-05 | 9.23517e-05 - 7.57095e-05 |
| 15 | 0.000255847 | 0.000114120 | 0.000011412 | 27.46197e-05 - 23.70742e-05 | 27.82144e-05 - 23.34794e-05 |
| 20 | 0.000349996 | 0.000177600 | 0.000017760 | 37.92112e-05 - 32.07809e-05 | 38.48055e-05 - 31.51865e-05 |

*Figure 8: Run for 5, 10, 15, 20 vertices and Mean Times, Standard Deviation, Standard Error, Confidence Levels*

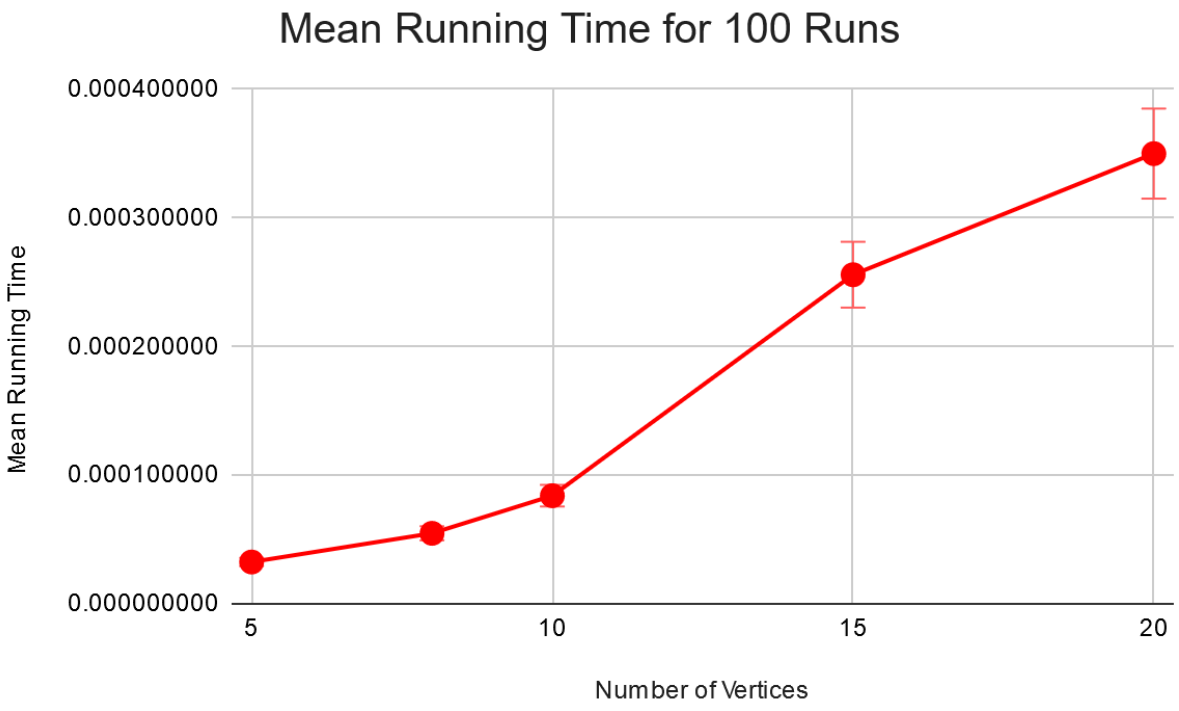



*Figure 9: Mean running time for 100 runs*

## 1000 RUN PER INPUT SIZE

| Size | Mean Time(s) | Standard Deviation | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 5 | 0.000028465 | 0.000015837 | 0.000000501 | 2.9288e-05 - 2.7640e-05 | 2.9446e-05 - 2.7483e-05 |
| 8 | 0.000053675 | 0.000024963 | 0.000000789 | 5.4973e-05 - 5.2376e-05 | 5.5221e-05 - 5.2127e-05 |
| 10 | 0.000089500 | 0.000042856 | 0.000001355 | 9.1729e-05 - 8.7271e-05 | 9.2156e-05 - 8.6844e-05 |
| 15 | 0.000242365 | 0.000133968 | 0.000004236 | 24.9334e-05 - 23.5396e-05 | 25.0668e-05 - 23.4061e-05 |
| 20 | 0.000347586 | 0.000166958 | 0.000005280 | 35.6270e-05 - 33.8900e-05 | 35.7934e-05 - 33.7237e-05 |

*Figure 10: Run for 5, 10, 15, 20 vertices and Mean Times, Standard Deviation, Standard Error, Confidence Levels*

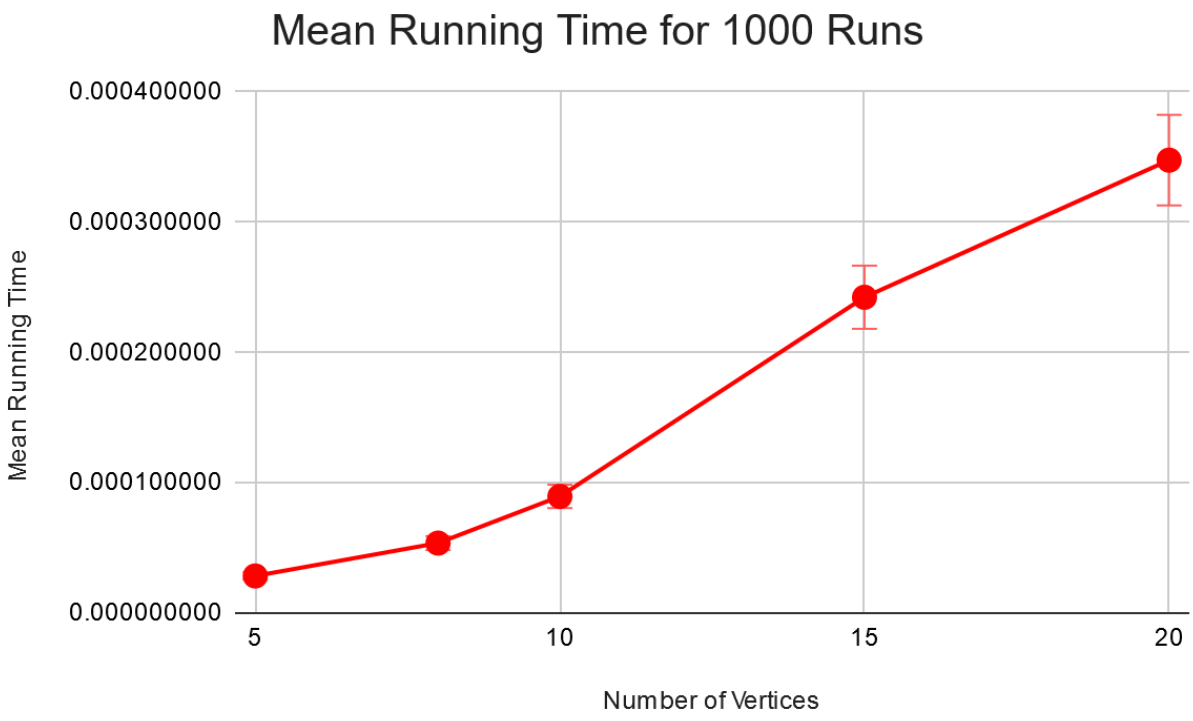



*Figure 11: Mean running time for 1000 runs*

## 3000 RUN PER INPUT SIZE

| Size | Mean Time(s) | Standard Deviation | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 5 | 0.000027366 | 0.000016602 | 0.000000235 | 2.7752e-05 - 2.6979e-05 | 2.7826e-05 - 2.6905e-05 |
| 8 | 0.000053334 | 0.000026294 | 0.000000372 | 5.3946e-05 - 5.2722e-05 | 5.4063e-05 - 5.2605e-05 |
| 10 | 0.000084896 | 0.000041764 | 0.000000591 | 8.5867e-05 - 8.3924e-05 | 8.6053e-05 - 8.3738e-05 |
| 15 | 0.000241553 | 0.000127683 | 0.000001806 | 24.452e-05 - 23.858e-05 | 24.509e-05 - 23.801e-05 |
| 20 | 0.000356408 | 0.000181924 | 0.000002573 | 36.064e-05 - 35.217e-05 | 36.145e-05 - 35.136e-05 |

*Figure 12: Run for 5, 10, 15, 20 vertices and Mean Times, Standard Deviation, Standard Error, Confidence Levels*

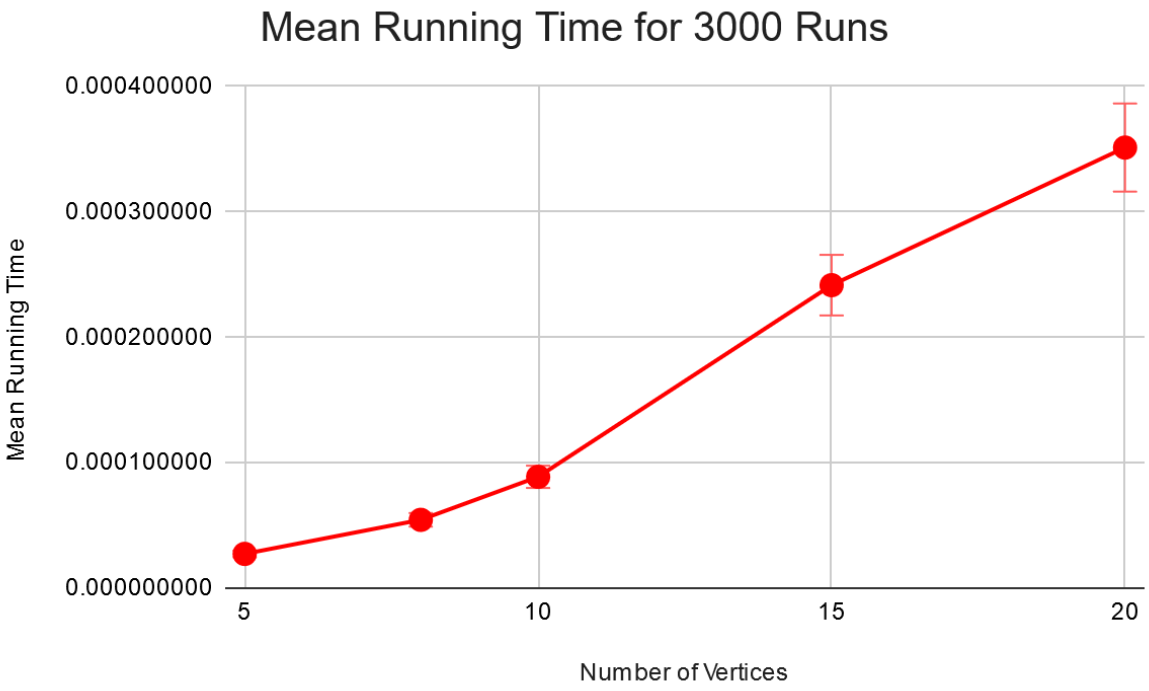



*Figure 13: Mean running time for 3000 runs*

## 5000 RUN PER INPUT SIZE

| Size | Mean Time(s) | Standard Deviation | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 5 | 0.000027294 | 0.000013469 | 0.000000246 | 2.7698e-05 - 2.6889e-05 | 2.7776e-05 - 2.6812e-05 |
| 8 | 0.000054436 | 0.000026454 | 0.000000483 | 5.5230e-05 - 5.3641e-05 | 5.5382e-05 - 5.3489e-05 |
| 10 | 0.000088715 | 0.000049758 | 0.000000908 | 9.0209e-05 - 8.7220e-05 | 9.0495e-05 - 8.6934e-05 |
| 15 | 0.000241525 | 0.000130856 | 0.000002389 | 24.5455e-05 - 23.7595e-05 | 24.6208e-05 - 23.6842e-05 |
| 20 | 0.000351227 | 0.000175799 | 0.000003210 | 35.6506e-05 - 34.5947e-05 | 35.7518e-05 - 34.4936e-05 |

*Figure 14: Run for 5, 10, 15, 20 vertices and Mean Times, Standard Deviation, Standard Error, Confidence Levels*

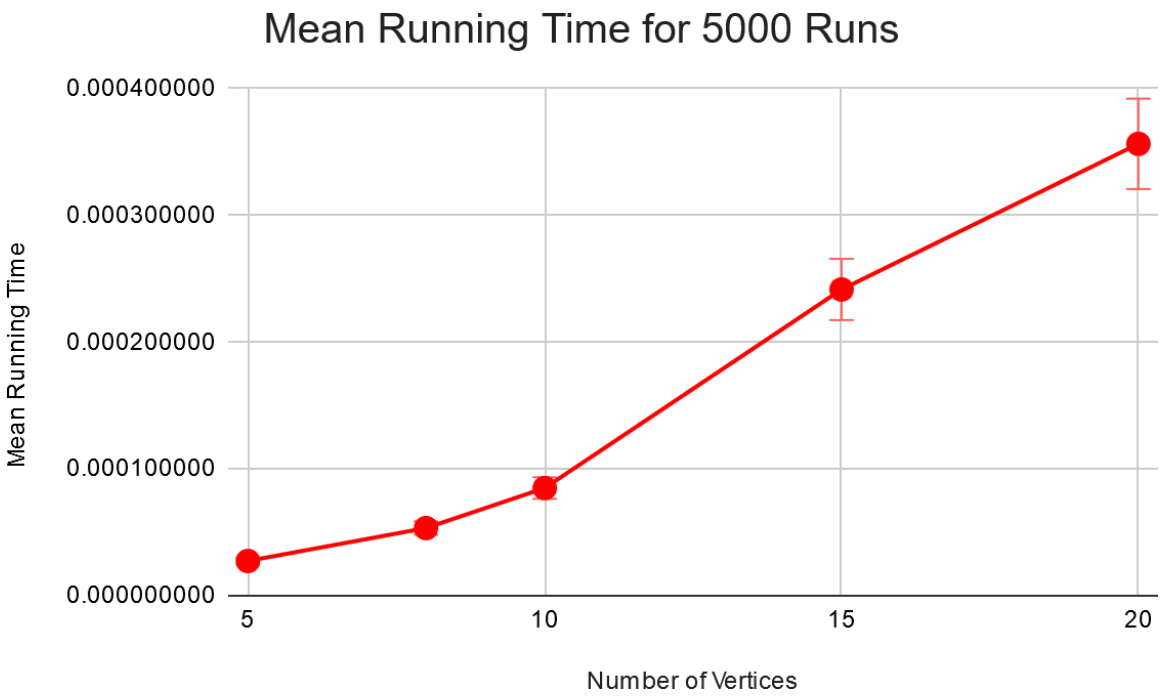



*Figure 15: Mean running time for 5000 runs*

## 10000 RUN PER INPUT SIZE

| Size | Mean Time(s) | Standard Deviation | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 5 | 0.000027646 | 0.000017236 | 0.000000172 | 2.7929e-05 - 2.7362e-05 | 2.7984e-05 - 2.7308e-05 |
| 8 | 0.000055044 | 0.000039898 | 0.000000399 | 5.5700e-05 - 5.4387e-05 | 5.5825e-05 - 5.4261e-05 |
| 10 | 0.000085520 | 0.000043361 | 0.000000434 | 8.6233e-05 - 8.4806e-05 | 8.6369e-05 - 8.4670e-05 |
| 15 | 0.000240504 | 0.000129679 | 0.000001297 | 24.263e-05 - 23.837e-05 | 24.3045e-05 - 23.7961e-05 |
| 20 | 0.000347514 | 0.000173038 | 0.000001730 | 35.036e-05 - 34.466e-05 | 35.090e-05 - 34.412e-05 |

*Figure16: Run for 5, 10, 15, 20 vertices and Mean Times, Standard Deviation, Standard Error, Confidence Levels*

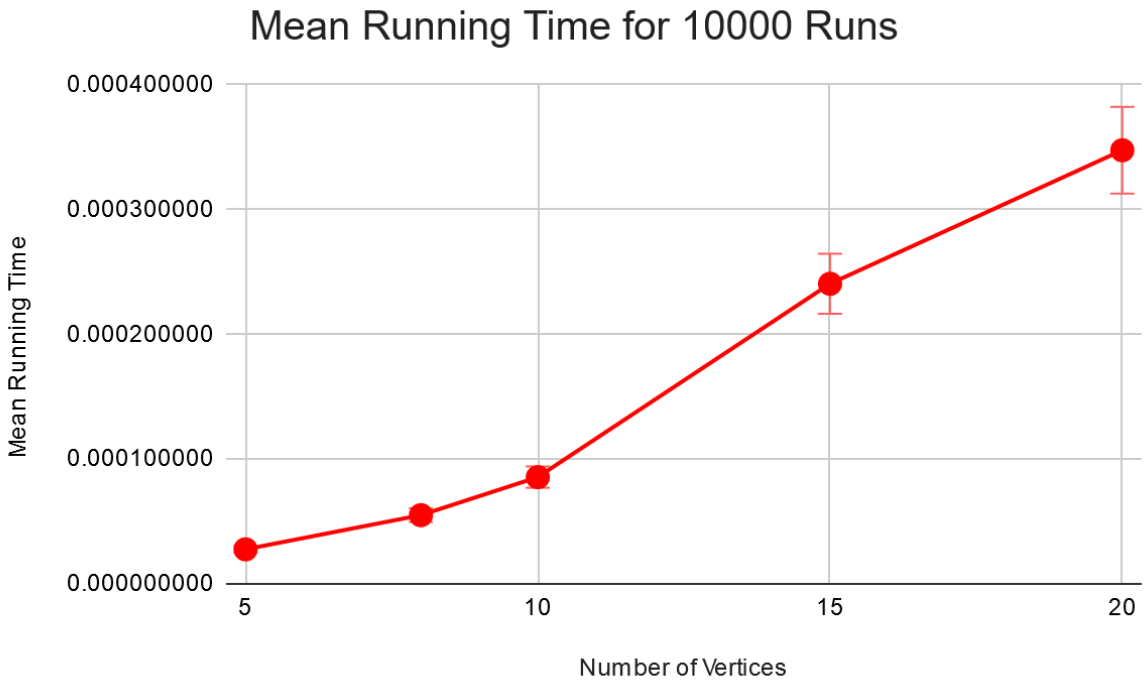



*Figure 17: Mean running time for 10000 runs*

## TESTING

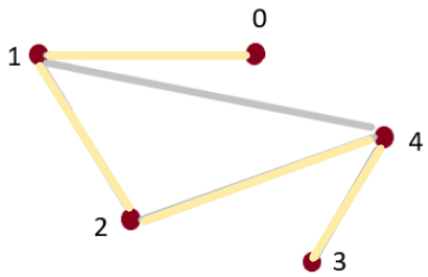
For testing our algorithm with different inputs, we randomly created graphs with different vertex numbers as can be seen below.

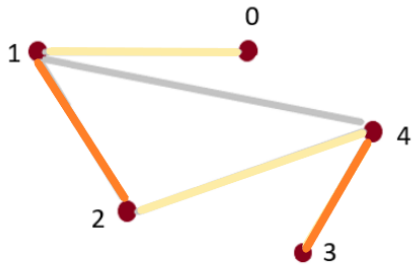
We created a test file for the edge numbers between 1 and all edges.

```
//generates n graphs with V vertices each and writes them to file f
void generateGraph(int V, int n, std::string f) {
    //open file
    std::ofstream output;
    output.open(f);
    if (output.fail()) {
        std::cout << "File cannot open!";
        return;
    }
    //generate all the possible edges
    std::vector<edge> graph;
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            edge e(i, j);
            graph.push_back(e);
        }
    }
    int totalNumEdges = graph.size();

    //create n different graphs
    for (int i = 0; i < n; i++) {
        //select at least V many at most all edges
        int numEdges = rand() % (totalNumEdges - V + 1) + V;

        //select edges randomly
        std::vector<int> edgeIndexes;
        for (int j = 0; j < numEdges; j++) {
            int index = rand() % totalNumEdges;
            if (indexExists(edgeIndexes, index))
                j--;
            else
                edgeIndexes.push_back(index);
        }

        //create the random graph
        std::vector<edge> randomGraph;
        for (int j = 0; j < numEdges; j++)
            randomGraph.push_back(graph[edgeIndexes[j]]);

        //write the graph into the txt file
        for (int j = 0; j < numEdges; j++)
            output << randomGraph[j].V1 << " " << randomGraph[j].V2 << " ";
        output << "\n";

        //clear the vectors
        edgeIndexes.clear();
        randomGraph.clear();
    }
    output.close(); //close file
}
```

## SOME TEST RESULT PATHS

Sample 1: Input graph with 5 vertices and 5 edges:



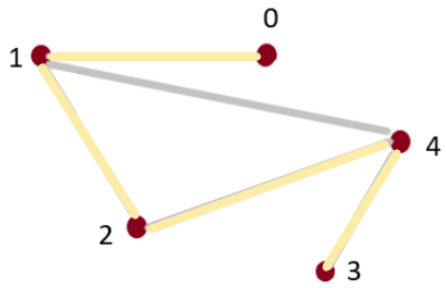



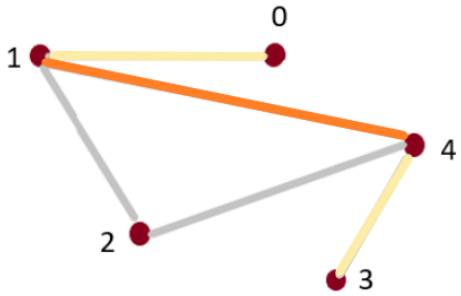

Exact Algorithm: `Path Found`

Approximation: `Path Found`

`This is a successful case.`

Sample 2: Input graph with 5 vertices and 5 edges:



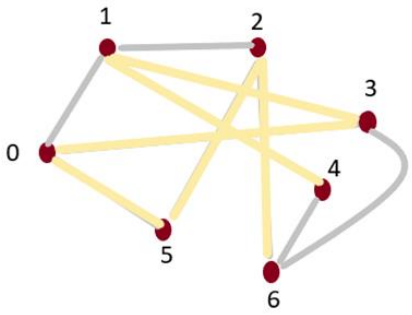



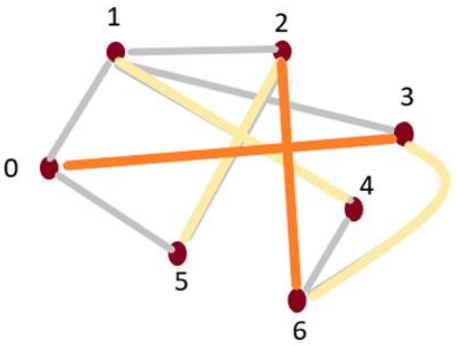

Exact Algorithm: `Path Found`

Approximation: `No Path`

`This is a failed case.`

Sample 3: Input graph with 7 vertices and 10 edges:



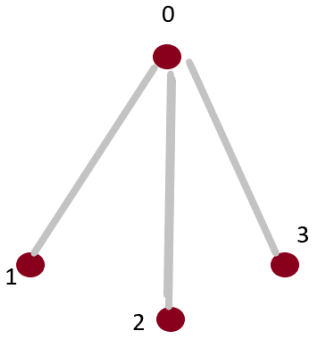



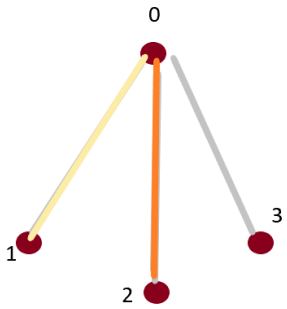

Exact Algorithm: `Path Found`

Approximation: `No Path`

`This is a failed case.`

Sample 3: Input graph with 4 vertices and 3 edges:




Exact Algorithm: `No Path`

Approximation: `No Path`

`This is a successful case.`

## DISCUSSION

We have described the Hamiltonian Path problem and have shown that it is an NP-Hard Problem reduced from the Hamiltonian Cycle problem. Therefore, there does not exist a polynomial time algorithm that gives an exact result.

We used a modified version of Kruskal algorithm which is a randomized greedy algorithm that gives an approximate solution to the Hamiltonian Path problem. We have analyzed the complexity and the correctness of the algorithm. Also, we have done an experimental analysis. As we expected, our algorithm does not always give the correct answer since it picks edges randomly and considers each at most twice.

Experimental analysis showed us an interesting behavior in the success rates. Our algorithm gives relatively better results when the number of vertices is an odd number. The reason is, our algorithm works in two steps as explained previously. If a graph has odd number of vertices, its Hamiltonian Path length will be even. Thus, it will have to find equal number of edges in both steps. However, if the number of vertices is even, and the Hamiltonian Path length is odd, then the number of edges found in the two steps will have to be different. This results in a lower probability of finding the correct permutations with this greedy and randomized approach. The success rate decreases among odd numbered vertices and even numbered vertices separately but both groups 'success rate decrease as the number of vertices increase. Our algorithm can be enhanced in terms of success rate by incorporating a strategic edge selection and giving less work to randomization.

Running time shows a linear trend when we analyzed the mean running times for each run types and our analysis was $O(VlogV + E)$. This is acceptable since the graph sizes in terms of vertex number are very small thus trend line of the data may be interpreted as linear. Further trials with large vertex number can show more accurate experimental running time information.

# REFERENCES

Approximation Code:

https://github.com/francoMG/Approximation-of-Hamiltonian-Path/blob/master/AHP.py

Exact Code:

https://github.com/samarth-p/Euler-and-Hamiltonian-Path

NP-Complete Proof:

http://www.csc.kth.se/utbildning/kth/kurser/DD2354/algokomp10/Ovningar/Exercise6_Sol.pdf