CS301 – Term Project

Project Report Group Members:

ECEAY ÇELTİK - 25164

BERFU AYDIN - 23520

İZLEM KURT- 24977

İLAYDA KURAN - 25528

Semester: 2020 Fall

Instructor: Husnu Yenigun

Faculty of Engineering and Natural Sciences

Content

1) PROBLEM DESCRIPTION

The longest circuit problem is a kind of essential NP-hard graph issue whose aim is basically finding the maximum length of a simple cycle in a graph. Simple cycle is a kind of connected subgraph of the graph whose all vertices have degree 2 and no repetitions in vertices or edges. Moreover, the longest circuit problem can be defined as finding a simple cycle whose length is maximum. Therefore, our main question is, in the given graph (V,E) and number of vertices denoted as k, from starting point s, if k number of distinct vertices are visited and come back to starting point or not? If the answer is yes, this means this graph should be a hamiltonian cycle. Otherwise, whichever maximum length of cycle it can be got should be our result.

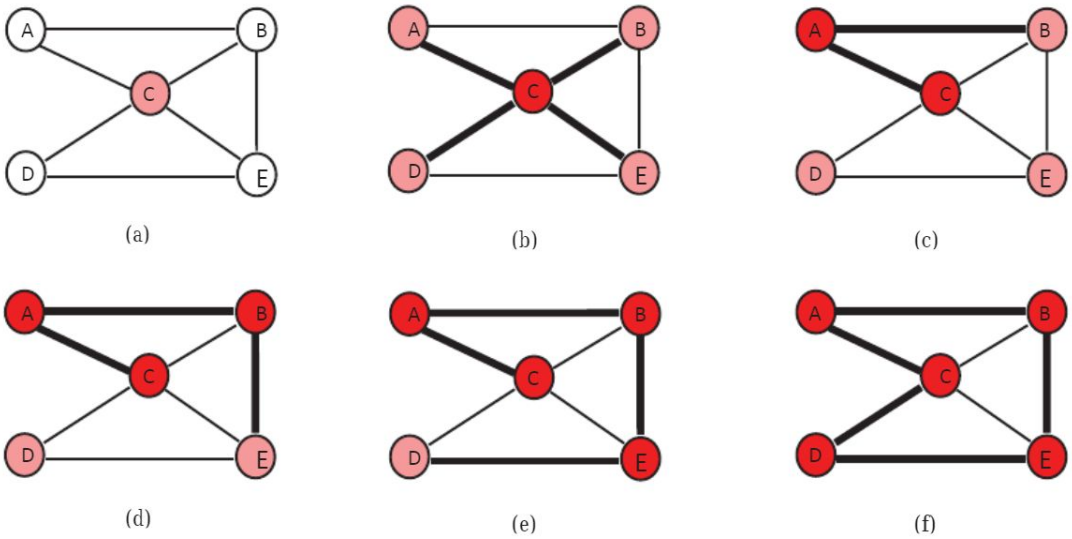
Nowadays, with the increasing amount of data, connected data has become more complicated so that the need for new methods of solving all these connections has increased. All of these have led to the creation of efficient algorithms which figure out some problems such as the longest circuit problem which is explained above. To give an example of the areas where they are used, they become significant in numerous real-world applications especially in complex networks including analysis of social networks or bioinformatics.

4 | Longest Circuit Problem

The Longest Circuit problem will be analyzed in this project. Many exact formulations have been proposed to solve this problem. For example, one of the definitive approaches in solving this problem is Dixon and Goodman's Formulation of the Longest Simple Cycle Problem which is An integer linear programming (**ILP**) mathematical formulation.In addition, some heuristic and approximation algorithms have been proposed, apart from all these exact approaches. Even, in some approximation algorithms, a recursive approach is adopted for disjoint cycle detection procedures.

To elaborate, the main objective of the project is finding a simple cycle of maximum length that contains the maximum number of vertices. While finding this, our aim is to find the longest circuit in the most efficient way.

A n-node undirected graph G(V,E) where V represents set of n vertices V = {v1, v2, . . ., vn} and E represents the set of m edges.



The goal is to find the longest path from u to u with the maximum edge length for all u,v ε V. This problem is an NP-Complete problem, as it is a generalized version of the problem of the Hamiltonian path and can not be solved unless P = NP in polynomial time.

In order to prove that the Longest Circuit is NP complete, the following must be shown.

1. Longest Circuit is in NP.
2. Longest Circuit can be reducible to an NP complete problem in polynomial time.

a) Longest Circuit is in NP

For a given graph G, we can solve the longest circuit by non-deterministically selecting edges from G that are to be included in the cycle.

Consider the Longest Simple Cycle undirected graph G(V,E) with an integer k and E edges. Here, G has a simple cycle.

Let (G,k) be an instance of an undirected graph and certificate of proof y, which is a sequence of at least k vertices, from the graph.

The verifier traverses each vertex one by one. Marking each node in which the verifier traverses and the verifier ensures that there are no duplicate nodes. In the certificate, there is an edge from each vertex to the next one.

Now, the verifier tests if an edge from the last vertex in the certificate back to the first vertex which completes the cycle. If the verifier satisfies the certificate, gets output 1. Otherwise, gets output 0.

To search the number of vertices, this verifier takes linear time. If a simple cycle with at least k vertices exists, then passing the cycle as a certificate causes the verifier outputs 1. Otherwise, no certificate contains a valid cycle with at least k vertices, so for this case, the verifier always gives 0 output. The problem is in NP because the verifier is correct and takes polynomial time.

b) Longest Circuit can be reducible to an NP complete problem in polynomial time.

To prove that Longest Circuit is NP-Complete, it should be made a reduction from Hamiltonian Cycle to Longest Simple Cycle in polynomial time.

Given the graph G, which is the HAM-CYCLE instance, define the LONGEST SIMPLE CYCLE instance (G,k) containing the same graph , and k=|V|, the number of vertices in G. This reduction takes polynomial time as only simple copying and counting is included. Now, if it can be found a Hamiltonian Cycle in the graph, then this cycle

should be a simple cycle which contains |V| vertices, therefore, in case of Longest Simple Cycle, the appropriate decision should be 1. If in case in which it can not be found any Hamiltonian cycle, then graph should not contain any simple cycle with |V| or more vertices, so in this case the appropriate decision LONGEST SIMPLE CYCLE is 0. Thus, the reduction is correct and takes polynomial time, showing that NP-hard is the **LONGEST SIMPLE CYCLE**. Since it is also in NP, it is **NP-complete**.

2) ALGORITHM DESCRIPTION

There is no algorithm found in polynomial time for the longest simple cycle in an undirected graph. However, there are some heuristic algorithms which can solve the longest simple cycle problem in polynomial time, but they are not guaranteeing an optimal solution. In this project, we are going to implement an algorithm to solve this problem.

The algorithm steps:

At the beginning there will be an empty parents stack which keeps the parent information of the vertices and also an empty dictionary which keeps the unvisited and visited information of vertices. We have an empty stack called cycleStartEnd.

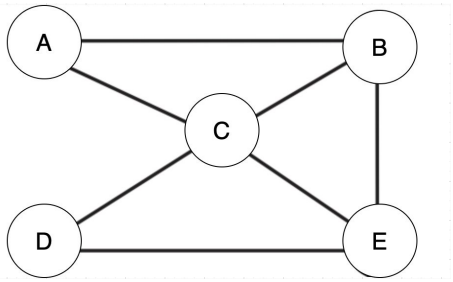
1. First all vertices are unvisited and the algorithm picks a random node as a root node.
2. Applying the DFS algorithm. visitedNodes[v] = 0 means that v is not visited; visitedNodes[v] = 1 means that v is visited and can be in the cycle; visitedNodes[v] = 2 means that v is eliminated.
    a. Create a recursive function that takes the index of node, the predecessor of that vertex, visited array, parent stack, graph itself, cycleStartEnd stack.
    b. Mark the current node v as visited and check for the adjacency nodes of v. If v's neighbor is not the predecessor of v and check if it is visited or not. If it is not visited then make the v's neighbor's parent as v. Call the recursive DFS function.
    c. If the neighbor is visited add the neighbor and node v to the cycleStartEnd stack and cycle is found. Otherwise, make the visited information of that node as 2 and return false.
3. After DFS cycleStartEnd will include the beginning of the cycle and the node before the beginning vertex. If the cycle is found in the DFS algorithm. Make a stack, which includes the beginning vertex, and called cycle in order to add the vertices of that cycle into this stack. Add the vertices into the cycle stack with traversing the parent stack. At the end add the beginning node again.

4. After finding a cycle, apply an improvement operator -to find a longer cycle- which takes a pair of adjacent nodes and looks at if they have an edge to common another node which improves the cycle.

```
def dfs(v, p, parents, visitedNodes, graph, cycleStartEnd):
    visitedNodes[v] = 1
    for neighbour in graph[v]:
        if neighbour != p:
            if visitedNodes[neighbour] == 0:
                parents[neighbour] = v
                if dfs(neighbour, v, parents, visitedNodes, graph, cycleStartEnd):
                    return True
            elif visitedNodes[neighbour] == 1:
                cycleStartEnd.append(neighbour)
                cycleStartEnd.append(v)
                return True

    visitedNodes[v] = 2
    return False
```

```
def heuristicLongestCycle(graph, V):
    visitedDict = {}
    parents = []
    for vertex in graph:
        visitedDict[vertex] = 0
        parents.append(-1)

    randomRootNodes = list(range(V))
    random.shuffle(randomRootNodes)
    cycleStartEnd = []
    isCycle = False

    for randomRoot in randomRootNodes:
        if visitedDict[randomRoot] == 0:
            isCycle = dfs(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
            if isCycle:
                break
    if isCycle:
        cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]

        cycle = [cycleStart]
        v = cycleEnd

        while v != cycleStart:
            cycle.append(v)
            v = parents[v]

        cycle.append(cycleStart)
        cycle.reverse()
        return cycle
    else:
        return None
```

```
def improveCycle(graph, result):
    toBeAdded = {}
    for i in range(len(result) - 2):
        neighbourList1 = graph[result[i]]
        neighbourList2 = graph[result[i + 1]]
        sharedNodes = list(set(neighbourList1).intersection(neighbourList2))
        if sharedNodes:
            for sharedNode in sharedNodes:
                if sharedNode not in result:
                    toBeAdded[sharedNode] = i + 1
                    break

    for node in toBeAdded:
        result.insert(toBeAdded[node], node)

    return result
```



A n-node undirected graph G(V, E) with node set V and edge set E

3) ALGORITHM ANALYSIS

Worst case asymptotic running time of the algorithm as follows:

```
def dfs(v, p, parents, visitedNodes, graph, cycleStartEnd):
    visitedNodes[v] = 1
    for neighbour in graph[v]:
        if neighbour != p:
            if visitedNodes[neighbour] == 0:
                parents[neighbour] = v
                if dfs(neighbour, v, parents, visitedNodes, graph, cycleStartEnd):
                    return True
            elif visitedNodes[neighbour] == 1:
                cycleStartEnd.append(neighbour)
                cycleStartEnd.append(v)
                return True

    visitedNodes[v] = 2
    return False
```

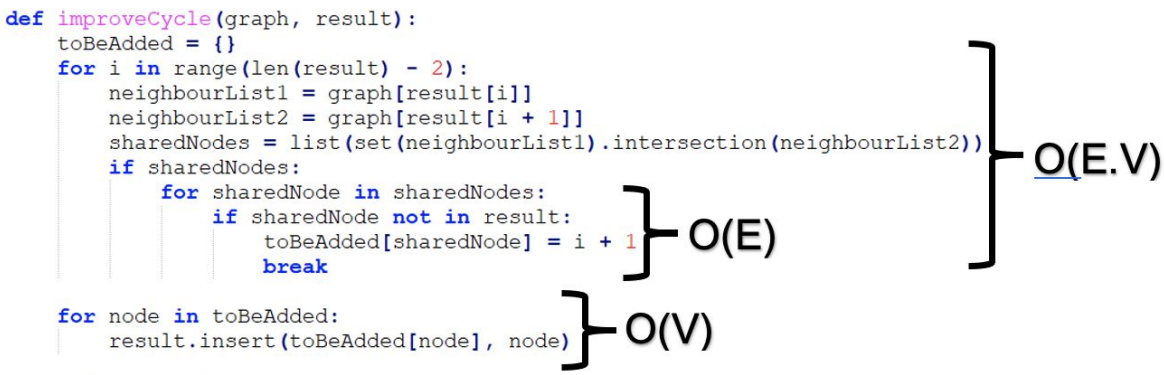
O(E+V)

```
def heuristicLongestCycle(graph, V):
    visitedDict = {}
    parents = []
    for vertex in graph:
        visitedDict[vertex] = 0
        parents.append(-1)

    randomRootNodes = list(range(V))
    random.shuffle(randomRootNodes)
    cycleStartEnd = []
    isCycle = False

    for randomRoot in randomRootNodes:
        if visitedDict[randomRoot] == 0:
            isCycle = dfs(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
            if isCycle:
                break
    if isCycle:
        cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]

        cycle = [cycleStart]
        v = cycleEnd

        while v != cycleStart:
            cycle.append(v)
            v = parents[v]

        cycle.append(cycleStart)
        cycle.reverse()
        return cycle
    else:
        return None
```

O(V)

O(V+E)

$O(V^2+V.E)$



The total time for finding a cycle and improving it is O(V^2 +VE) + O(VE), so it will cost at most O(V^2 +VE). According to the number of edges in the graph, it can be lower.

For the ratio bound:

Practical ratio bound can be found by testing the heuristic algorithm and exact algorithm and taking the average of them, then comparing with the results, performance of them.

Since the exact algorithm has exponential time and takes too long to halt, vertex count of the graphs are not that big.

After testings;

Ratio Bound = AVG[Size(Exact Solution) / Size(Heuristic Solution)] = 1.3

* Note that size represents the number of vertices in the corresponding longest simple cycle solution.

As a result, since the vertex and edge numbers were limited in these cases, ratio bound can be changed and increased when the input size gets bigger.

4) EXPERIMENTAL ANALYSIS

a) Running Time Experimental Analysis

The algorithm running time complexity is experimentally analysed by using standard deviation, standard error, sample mean and confidence level intervals. Functions of excel are used to calculate standard deviation, standard error, sample mean and confidence level intervals.

Running time of the algorithm measured with the following code and different iterations' times for the same number of vertex and edge kept in a stack:

```
def calculation_time(V,E):
    running_times = []
    for i in range (0,100):
        graph1 = createRandomGraph(V, E)
        print(graph1)
        start=datetime.now()
        heuristicResult = heuristicLongestCycle(graph1, V)
        improveresult= improveCycle(graph1, heuristicResult)
        print(50 * "*")
        print(heuristicResult)
        print(improveresult)
        print(datetime.now()-start)
        running_times.append((datetime.now()-start).total_seconds())
    return running_times


def writing_to_excel(running_times):
    workbook = xlsxwriter.Workbook('cs301.xlsx')
    worksheet = workbook.add_worksheet()

    # Start from the first cell.
    # Rows and columns are zero indexed.
    row = 0
    column = 0

    # iterating through content list
    for item in running_times:
        worksheet.write(row, column, item)
        row += 1

    workbook.close()
```

After all running times calculated mean time calculated with total time (add all running times)/N(number of iterations)

Standard deviation formula:

$$s = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \overline{x})^2}{N-1}}$$

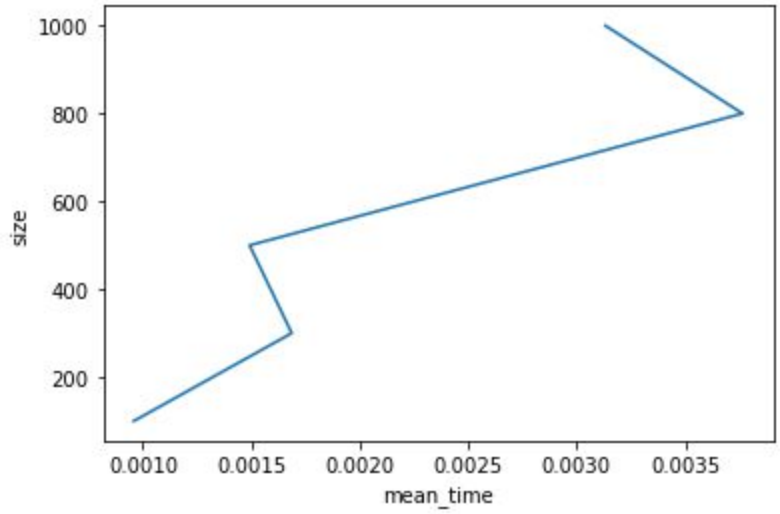
Standard error formula:

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

1) Keep Number of Edges Constant (E = 200)

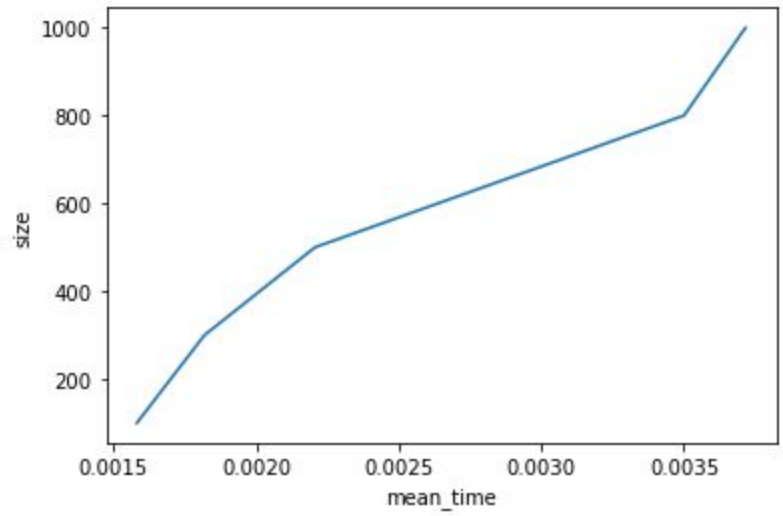
100 Number of Iterations per each Input Size

Size(V)	Mean Time(s)	Standard Deviation	Standard Error	%90 -CL	%95 -CL
100	0,00095777	0,000718289	0,0000718	0,00083962201-0,000118148	0,000816987-0,00109852
300	0,00168572	0,001101685	0,00011	0,00150450901-0,001866931	0,001469793-0,001901646
500	0,00149143	0,000548287	0,0000548	0,0014012447-0,001581615	0,001383967-0,001598892
800	0,00376397	0,004079227	0,000408	0,003092996-0,004434943	0,00296445-0,00456348
1000	0,00313193	0,001648538	0,000165	0,002860769-0,00340309	0,002808822-0,003455037



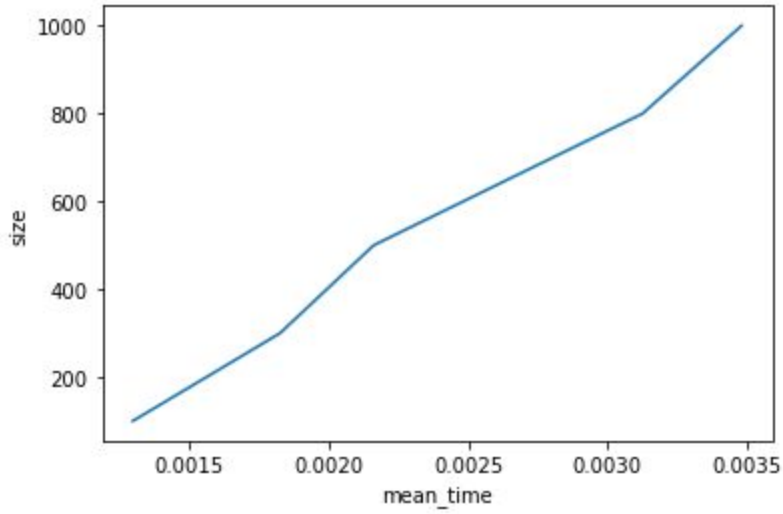
1000 Number of Iterations per each Input Size

Size(V)	Mean Time(s)	Standard Deviation	Standard Error	%90 -CL	%95 -CL
100	0,00158081	0,001526284	0,0000483	0,00150134-0,016602	0,00148609-0,01675
300	0,001817996	0,001800837	0,000057	0,0017242-0,0019117	0,001706-0,0019297
500	0,002205839	0,001535184	0,0000485	0,002125912-0,0022857	0,0021105-0,002301
800	0,003500444	0,002761929	0,0000873	0,003356649-0,0036442	0,00332905-0,0036718
1000	0,003716299	0,003538424	0,000112	0,003532077-0,00390052	0,0034967-0,0039358



5000 Number of Iterations per each Input Size

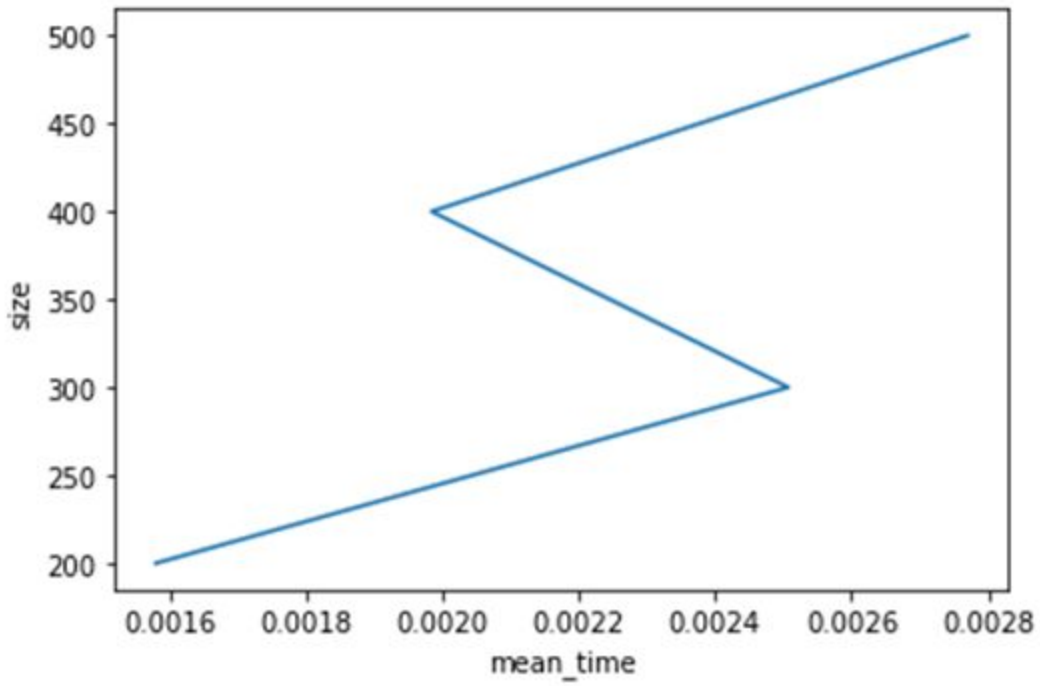
Size(V)	Mean Time(s)	Standard Deviation	Standard Error	%90 -CL	%95 -CL
100	0,001295703	0,001919913	0,0000272	0,00125104-0,0013403	0,001242487-0,0013489
300	0,001824139	0,001888837	0,0000267	0,001780201363-0,001868077	0,00177178407-0,00187649
500	0,002159799	0,002038306	0,0000288	0,002112384-0,00220721	0,002103301-0,0022162
800	0,003125111	0,003108087	0,00004395	0,003052811-0,00319741	0,00303896-0,003211261
1000	0,003480098	0,003193776	0,00004517	0,0034058051-0,00355439	0,0033915726-0,003568623



2) Keep Number of Vertices Constant (V = 200)

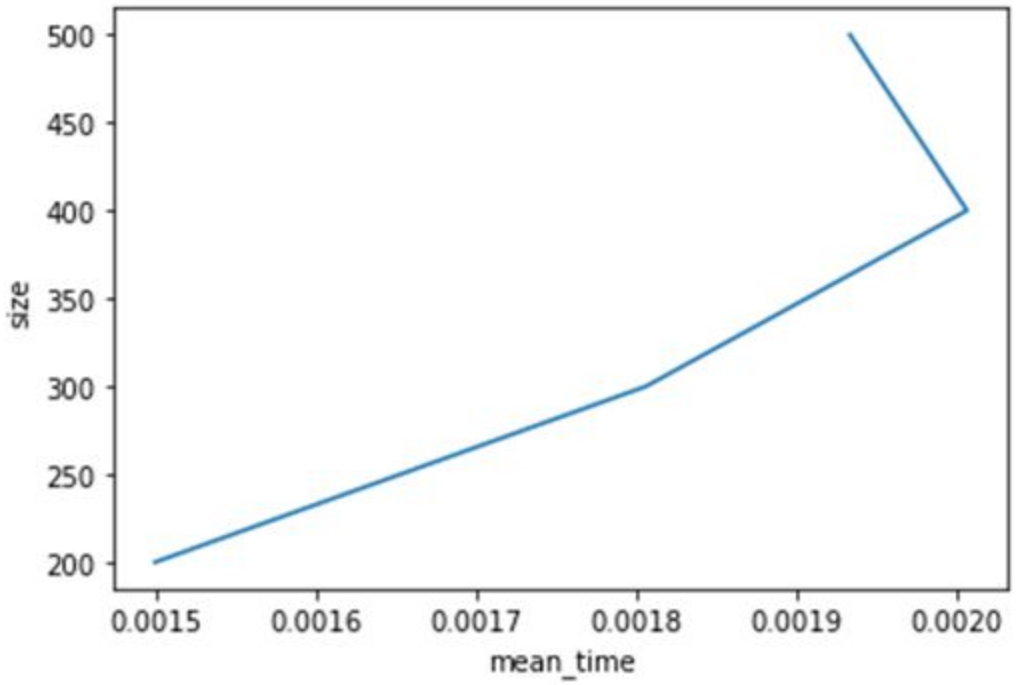
100 Number of Iterations per each Input Size

Size(E)	Mean Time(s)	Standard Devation	Standard Error	%90-CL	%95-CL
200	0,001578	0,001009	0,0001009	0,001313-0,001843	0,001378-0,001778
300	0,002507	0,001942	0,0001942	0,002185-0,002829	0,002122-0,002892
400	0,001983	0,000781	0,0000781	0,001853-0,002113	0,001828-0,002138
500	0,00277	0,001382	0,0001382	0,002541-0,002999	0,002496-0,003044



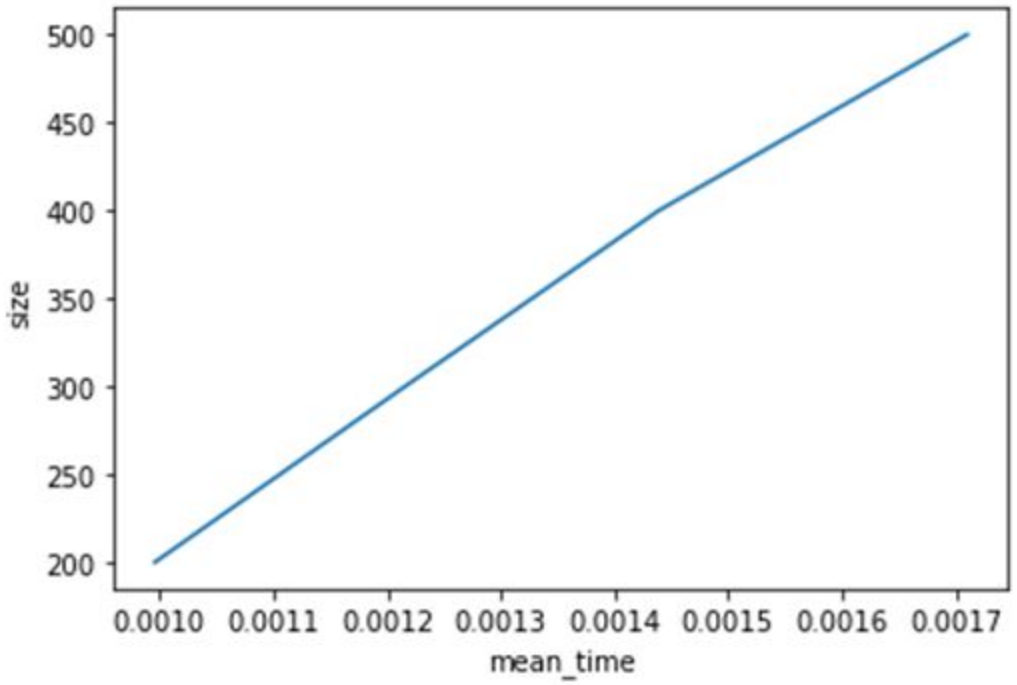
1000 Number of Iterations per each Input Size

Size(E)	Mean Time(s)	Standard Devation	Standard Error	%90-CL	%95-CL
200	0,0015	0,004045	0,000127914	0,001289-0,001711	0,001249-0,001751
300	0,001806	0,0037	0,000117	0,001813-0,002199	0,001776-0,002236
400	0,002006	0,001902	0,00006015	0,001371-0,001569	0,001352-0,001588
500	0,001933	0,002122	0,0000671	0,001623-0,001843	0,001601-0,001865



5000 Number of Iterations per each Input Size

Size(E)	Mean Time(s)	Standard Devation	Standard Error	%90-CL	%95-CL
200	0,000996	0,00207	0,0000293	0,000948-0,0010442	0,000939-0,00105
300	0,001216	0,002369	0,0000335	0,001161-0,001271	0,00115-0,00128
400	0,001439	0,002078	0,0000294	0,00139-0,001487	0,0013814-0,0015
500	0,00171	0,001948	0,0000275	0,0016647-0,001755	0,001656-0,001764



b) Correctness

The heuristic algorithm for the longest simple cycle problem doesn't always provide the longest simple cycle for a given graph. However, it provides a cycle in all of the iterations since it traverses vertices until it finds a cycle in the Graph and improves it to make it longer.

The following algorithm compares the size of the longest cycle which comes from the exact algorithm and the size of the result of the heuristic algorithm, if it matches, the correct solution is found. We tested the algorithm with different vertex and edge sizes (vertices up to 10 because of the exponential exact algorithm) and did several iterations for the same sizes. Then calculate the correctness rate which is 20%. The correctness rate is very high for 5, 6 vertices, when the vertex number increases the correctness rate decreases.

```
correct= []
def correctness(V,E):
  for i in range (0,5):
      graph1 = createRandomGraph(V, E)
      heuristicResult = heuristicLongestCycle(graph1, V)
      improveresult= improveCycle(graph1, heuristicResult)
      exactResult = exactLongestCycle(graph1, V)
      if len(improveresult) == len(exactResult):
          correct.append(1)
      else:
          correct.append(0)

correctness(5,10)
correctness(6,15)
correctness(7,14)
correctness(8,16)
correctness(9,20)
correctness(9,17)


def iterations():
    corresult=0
    failed=0
    for i in correct:
      if i==1:
        corresult+=1
      else:
        failed+=1

    total=corresult+failed
    rate= corresult/total*100
    print(rate)

iterations()
```

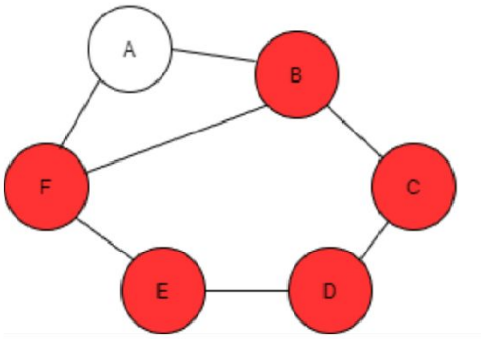
The correctness rate:

```
In [62]: runfile('D:/masa üstü/301proj/igraphs.py', wdir='D:/masa
üstü/301proj')
20.0
```

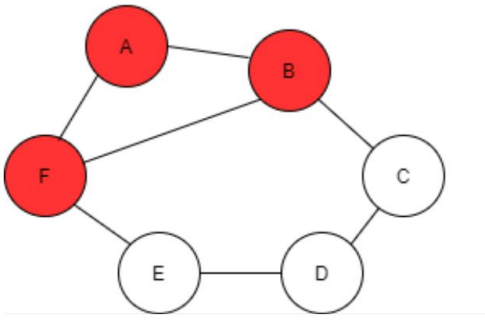
5) TESTING

For testing our heuristic algorithm blackbox testing technique used, we randomly generate graphs with different vertex and edge sizes. The following code is for creating a random graph.

```
def add_edge(inputGraph, vertex, vertexTo):
    inputGraph[vertex].append(vertexTo)
    inputGraph[vertexTo].append(vertex)


def add_vertex(inputGraph, vertex):
    inputGraph[vertex] = []


def createRandomGraph(V, E):
    graph = {}
    for i in range(V):
        add_vertex(graph, i)

    for _ in range(E):
        selectRandomIndex = random.randint(0, V - 1)
        while len(graph[selectRandomIndex]) == V - 1:
            selectRandomIndex = random.randint(0, V - 1)
        selectRandomIndex2 = random.randint(0, V - 1)
        while selectRandomIndex2 == selectRandomIndex or selectRandomIndex2 in graph[selectRandomIndex]:
            selectRandomIndex2 = random.randint(0, V - 1)
        add_edge(graph, selectRandomIndex, selectRandomIndex2)
    return graph
```

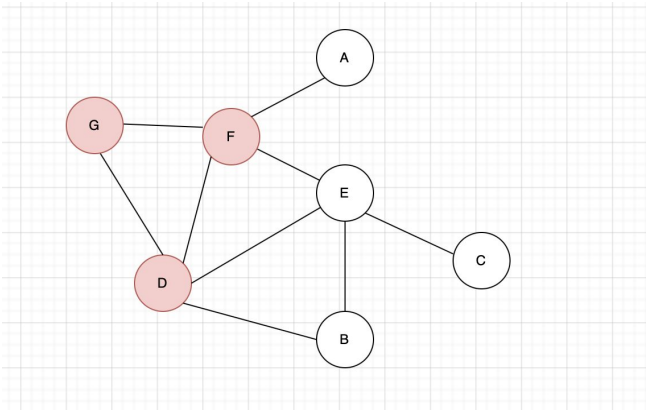


(1)

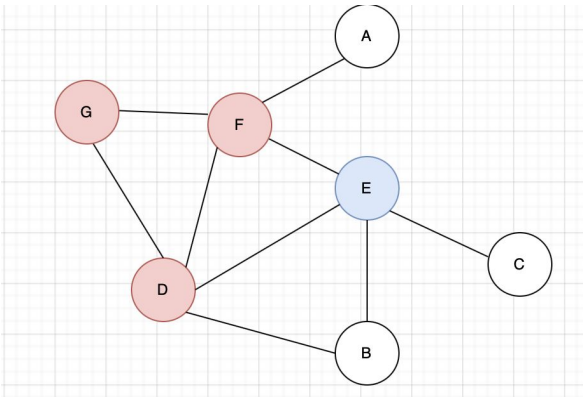


(2)

FAILED: Graph (1) represents the longest simple cycle which has 5 vertices for this graph but our algorithm failed to find it for this instance. It generated the outcome given in graph(2) which has 3 vertices.

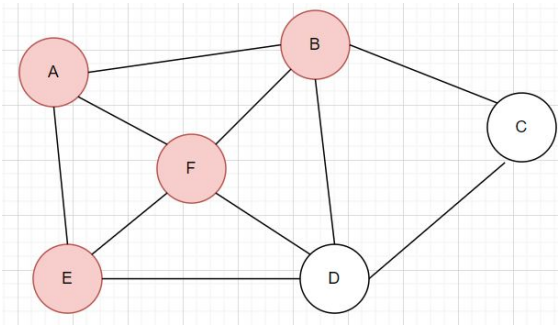


(1)

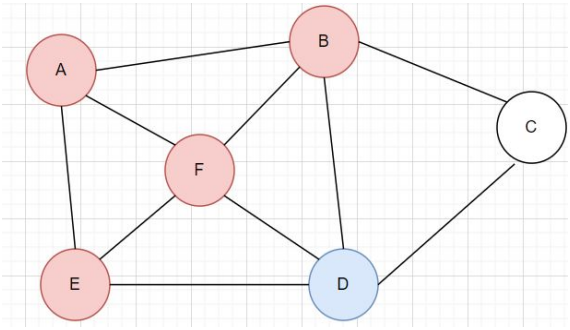


(2)

FAILED: Graph(1) represents the cycle but not the longest one with 3 vertices. In the improveCycle part, we find the vertex E, then add it to the cycle that we had in Graph(1) . But also this graph(2) is not the longest one since we cannot add B.

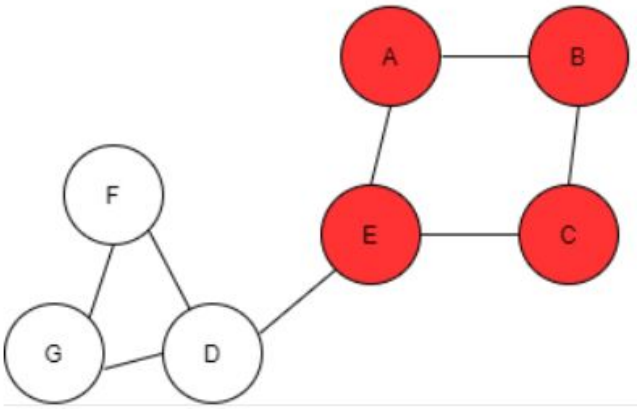


(1)

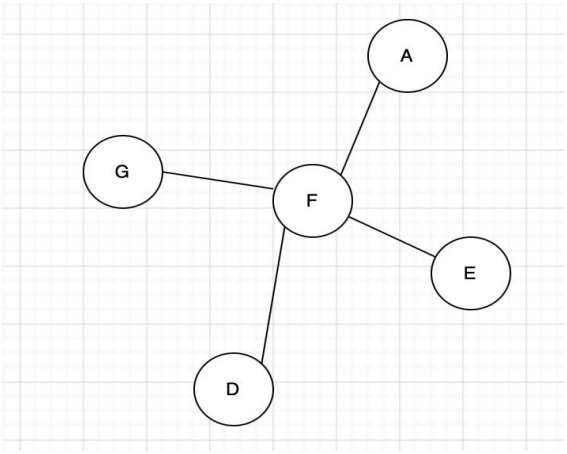


(2)

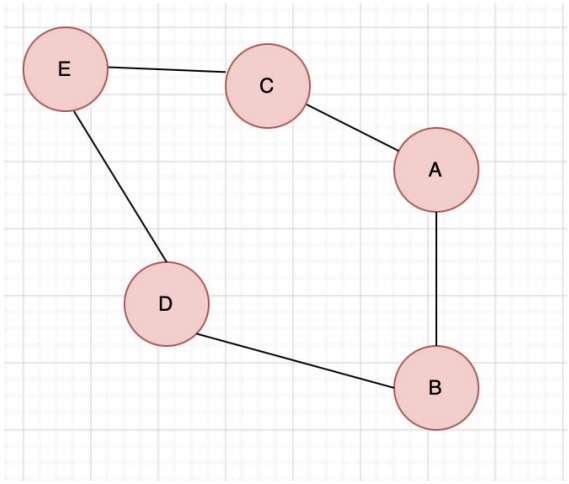
FAILED: Graph(1) represents a simple cycle and finds improvement and adds vertex B to the graph . Graph(2) represents a simple cycle with 5 vertices. Our algorithm finds improvement and adds vertex D to the graph. Although our algorithm finds improvement, graph(2) is not the longest cycle in this case.



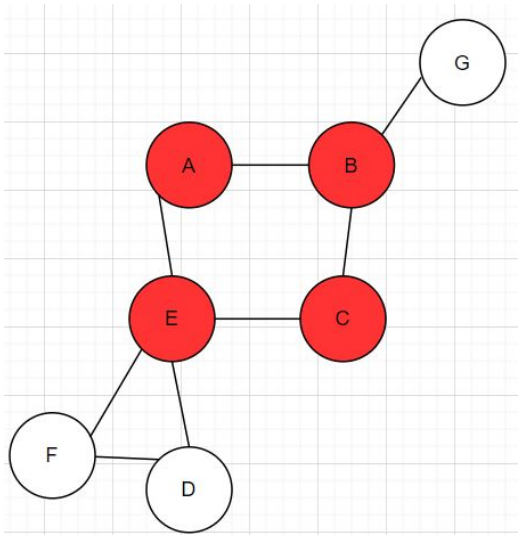
SUCCEEDED: The algorithm found the longest cycle with randomly picking one of the vertices and looked at its neighbours, because of the initial vertex it found correctly.



FAILED: In this graph we have no cycles so that we have no longest cycle as well. As a result, it will fail. Algorithm returns “None”.



SUCCEEDED: Since the only cycle is that algorithm successfully found the longest cycle.



SUCCEEDED: Algorithm finds longest cycle with 4 vertices

6) DISCUSSION

As a result, it is clearly indicated that The Longest Circuit is NP-Complete which can be reduced from the Hamiltonian Cycle in polynomial time. However, it can not be shown that the algorithm figures out the problem in polynomial time. There can be distinct heuristic algorithms which can be found so that it can solve the problem in linear time without ensuring the optimum solution.

The heuristic algorithm for the longest circuit gives a cycle for all graphs which have a cycle in it. However, it does not guarantee to give the longest cycle. The experimental analysis showed that the running time of the longest cycle heuristic algorithm and improvement algorithm is O(V^2 +VE) totally with keeping one input constant (edge or vertex) varying the other and vice versa. The correctness of the heuristic algorithm is determined by comparing results of the exact algorithm and the improved heuristic algorithm by testing with graphs at most 10 vertices. The correctness rate of the algorithm decreases as vertex and the edge number of the graph increases. The heuristic algorithm has produced a cycle and enlarged it with improvement operations for all the tested randomized graphs. A practical ratio bound calculated by testing and comparing an exact and a heuristic algorithm with small input sizes. Ratio bound is AVG(Size(Exact Solution) / Size(Heuristic Solution)). This ratio will not be the same for all inputs it can be changed and increased when the input size gets bigger.

REFERENCES

1. Chalupa, D. (2017, May 24). *Computational Methods for Finding Long Simple*. Retrieved From https://core.ac.uk/download/pdf/151161971.pdf

2. Gupta, N. (2014, July). *A Heuristic Algorithm for Longest Simple Cycle Problem*. Retrieved From https://www.researchgate.net/publication/278009819_A_Heuristic_Algorithm_for_Longest_Simple_Cycle_Problem