CS301 - Term Project

LONGEST CIRCUIT

Pelinsu Çiftçioğlu 25204

Bora Mert Karal 24904

Deniz Küçükahmetler 24879

Mert Malaz 25321

FALL 2020

INSTRUCTOR: HÜSNÜ YENİGÜN

Sabancı Üniversitesi

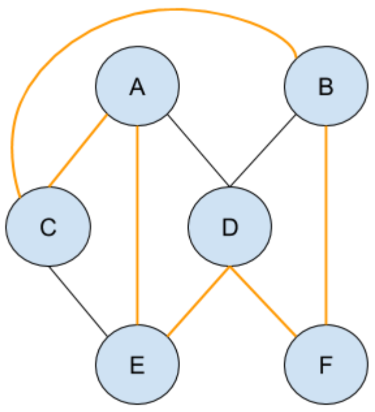
TABLE OF CONTENTS

PROBLEM DESCRIPTION

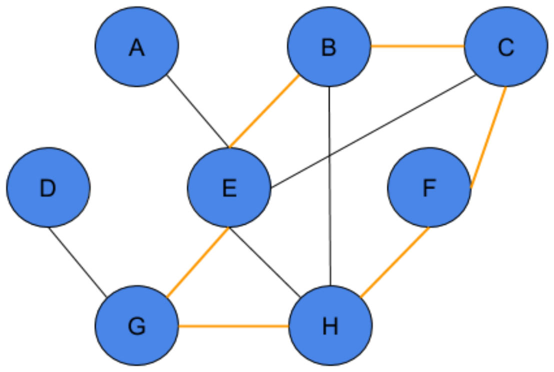
Longest circuit problem indicates the length of the longest simple cycle in a given graph. And the decision version of the same problem checks whether there is a simple cycle in a given graph such that the amount of vertices in the cycle contains at least a requested amount of nodes.

As a more formal definition, given an n-node undirected graph G(V,E) where V is the set of vertices and E is the set of edges it has been checked what is the longest simple cycle that G contains. A simple cycle or a circuit is a path containing at least 3 vertices where the only repeated vertex is the first/last one. And the length of the circuit is simply the number of distinct nodes within.

Following figure illustrates some examples of the Longest Circuits in a given graph:



Longest Circuit: (A, C, B, F, D, E, A)
Length of the Circuit: 6



Longest Circuit: (E, B, C, F, H, G, E)
Length of the Circuit: 6

Note that the length of the longest simple cycle is a sole solution but the cycle is not necessarily unique. Depending on the starting node for the cycle, the resulting cycle might be notated differently but they represent the same cycle. However, there also might be several different cycles that have the maximum length and in such a case it is not important which nodes the cycle contain but rather the number of nodes it has. Additionally, it is also possible to indicate that in dense graphs it is expected to find more cycles that give the same maximum length compared to the sparse graphs since there are more options of edges to form a cycle.

Finding an efficient solution for this problem could be very useful and practical for solving some of the real life problems that people face in real life. For example, one instance of the problem is where it is required to travel all the nodes in the graph and the edges are weighted which corresponds to the famous Travelling Salesman Problem[1]. Assuming there exists a set of cities to be visited and the distances given in between, Travelling Salesman problem is about finding the shortest possible path travelling each and every city exactly ones and going back to the node where it started[2]. Thus, it is possible to say that this is a special case of the Longest Circuit problem where the longest circuit is equal to the number of all the nodes in a given graph and the edges are weighted. And it might be highly important for most of the companies that distribute goods for their services since they would like to find the best route for their operations to minimize their costs in terms of the money they spent on the vehicle expenses.

PROOF OF THE HARDNESS OF THE PROBLEM

For proving the hardness of the Longest Circuit Problem, it will be proven that the problem is NP-Complete. Accordingly, by that way it will be also in the set of NP-Hard since decision version of the NP-Hard Problems are NP-Complete and can be converted to each other[3]. For this purpose, the problem needs to satisfy two conditions as follows: First, it will be shown that the problem is in NP such that the offered solution is verifiable in polynomial time. And second, it will be shown that the problem is as hard as any other problem by a problem that is already NP-Complete and can be reducible to the Longest Circuit problem.

[1] https://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html
[2] https://en.wikipedia.org/wiki/Travelling_salesman_problem
[3] Lecture Slides, CS301-Algorithms-08-NP-Completeness.pdf, p. 5

VERIFIABILITY IN POLYNOMIAL TIME

For checking whether a problem is verifiable in polynomial time so that it is in NP, it is assumed that there is a solution for the problem that is known to be correct and look if that solution can satisfy the formula (or the appropriate structure e.g. graph) in the boundaries of polynomial time complexity. Specifically for the problem Longest Circuit, the decision version of the problem will be used where it formally defined as follows:

Given an n-node undirected graph G(V,E) and a positive integer k, does G contain a simple cycle containing at least k nodes[4]?

For this purpose, it will be checked for a given set of k vertices which is claimed to be a satisfiable solution for the problem, whether they actually form a simple cycle in the concerning graph G. As an obvious solution that comes into mind, a given graph can be traced in the order of the solution set of vertices in linear time. Such as for the graph G(E, V) with n number of nodes within and set of vertices of solution $<v_1, v_2, \dots, v_k>$, following pseudocode can be used for verification:

```
bool checkGraph(input graph **G**, input array of nodes **Vsol**, number of node in the
solution **k**)
{
    for (i = 0, i < k, i++)
    {
        if node Vsol[i+1] is reachable from node Vsol[i] with an edge in E in G;
        // simply continue iterating
        else
            return FALSE;
    }
    return TRUE;
    // if it doesn't break in the loop that means there exist a graph with the given set
}
```

As it can be seen from the algorithm above, there is only one loop that iterates over the graph G and it can simply check in constant time whether the given node in the set and one after it has an edge connecting them by going into the given index in the graph (assuming that the locations of each node is known). Therefore, it is only important how many times the loop

[4] https://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html

will iterate and it depends on k which is the number of indices in the tour and therefore the time complexity is O(k). Thus it is possible to verify the given set of solutions in linear time hence in polynomial time in graph G. Knowing that the Longest Circuit problem is in NP (given that it is polynomially verifiable), it will be also checked whether it is NP-Complete in the following section.

REDUCTION

For performing the reduction, it will be shown that another problem that is known to be NP-Complete can be transformed into the problem Longest Circuit. For this purpose, it has selected the Hamiltonian Circuit problem which asks the question "Is it possible to form a tour in a given graph G by visiting every vertex exactly once?"[5]. Formally, a graph G = (V,E) contains a Hamiltonian Circuit if there exists a simple cycle in an order <v1, v2, … , vn> of the vertices of G, where n is equal to the number of vertices in G, such that $(v_n, v_1) \in E$, $\forall\ 1 \leq i < n$ and $(v_i, v_i+1) \in E$.

For reducing Longest Circuit to Hamiltonian Cycle, again the decision version of the problem will be used where it is formally defined in the above sections. Since Hamiltonian Cycle problem is known to be NP-Hard its decision version is in NP-Complete and reduction from Longest Circuit to Hamiltonian will be a proof for the problem to show that it is NP-Complete and hereby NP-Hard[6].

For a problem that is given to check whether there is an Hamiltonian Cycle in a graph G, Longest Circuit problem can be converted to this problem as follows:

Given an n-node undirected graph G(V,E) and a positive integer k, does G contain a simple cycle containing k = n (= |V|) nodes?

By this way, if the Longest Circuit problem is solved for n = k, it means that the Hamiltonian Cycle problem has been solved for the given graph[7]. And since it only takes a constant statement for the solution as checking whether the found longest cycle has exactly n

[5] Lecture Slides, CS301-Algorithms-08-NP-Completeness.pdf, p. 47
[6] Lecture Slides, CS301-Algorithms-08-NP-Completeness.pdf
[7] https://cseweb.ucsd.edu/classes/fa98/cse101/hw3ans/hw3ans.html

vertices, the reduction can take place in constant time. Therefore it can be easily understood that the Hamiltonian Cycle problem is just a special case for the Longest Circuit problem and Longest Circuit is also in NP-Complete.

ALGORITHM DESCRIPTION

Since the Longest Circuit problem is in NP-Hard, all known exact solutions cannot work in polynomial time but rather work in exponential time. However, heuristic algorithms can give wrong but optimal solutions, to an extent, to NP-Hard problems in polynomial time. The following heuristic algorithm provides an approximate solution to the Longest Circuit problem. It is implemented using a recursive depth-first search algorithm. The steps of the algorithm are as follows:

1. Create two empty lists to hold parent and visit information for each vertex, one list to store all the cycles found.
2. Choose the first vertex as the root.
3. Set the visit and parent lists to default values.
4. Do a depth-first search starting from the root using parent and visit lists. If there is a cycle, then store the start and end of the cycle found as CS (Cycle Start) and CE (Cycle End and proceed. Otherwise, first assign an empty cycle then choose the next vertex as the root and go back to Step 3
5. Create a list called cycle to store the current cycle, append CS as the first element.
6. While CS is not the same as CE, retrace the steps of the CE by looking at its parent and add each step to the cycle list to create the cycle.
7. Append the start of the cycle to the end to complete the cycle.
8. If there are still vertices to iterate over, then go to Step 3. Otherwise, return the longest element of the Cycles list.

PSEUDOCODE

Algorithm 1 Find Longest Cycle (graph)

```
Parents, Visited, Cycles = empty list

for each vertex in the graph do

  for each vertex in the graph do
    Set Visited to false
    Set Parent to -1
  end for

  root = current vertex
  cycleExists = DepthFirstSearch(root, Parents, Visited, graph, CycleInfo)

  if cycleExists then
    cycleStart = CycleInfo.Start
    cycleEnd = CycleInfo.End

    Create cycle as an empty list
    Append CycleStart to cycle

    while CycleEnd != CycleStart do
      Append CycleEnd to cycle
      Update CycleEnd by its parents' value
    end while

    Append CycleStart as the last element to cycle
    Append the cycle to Cycles list
  end if
end for

return the longest element of Cycles
```

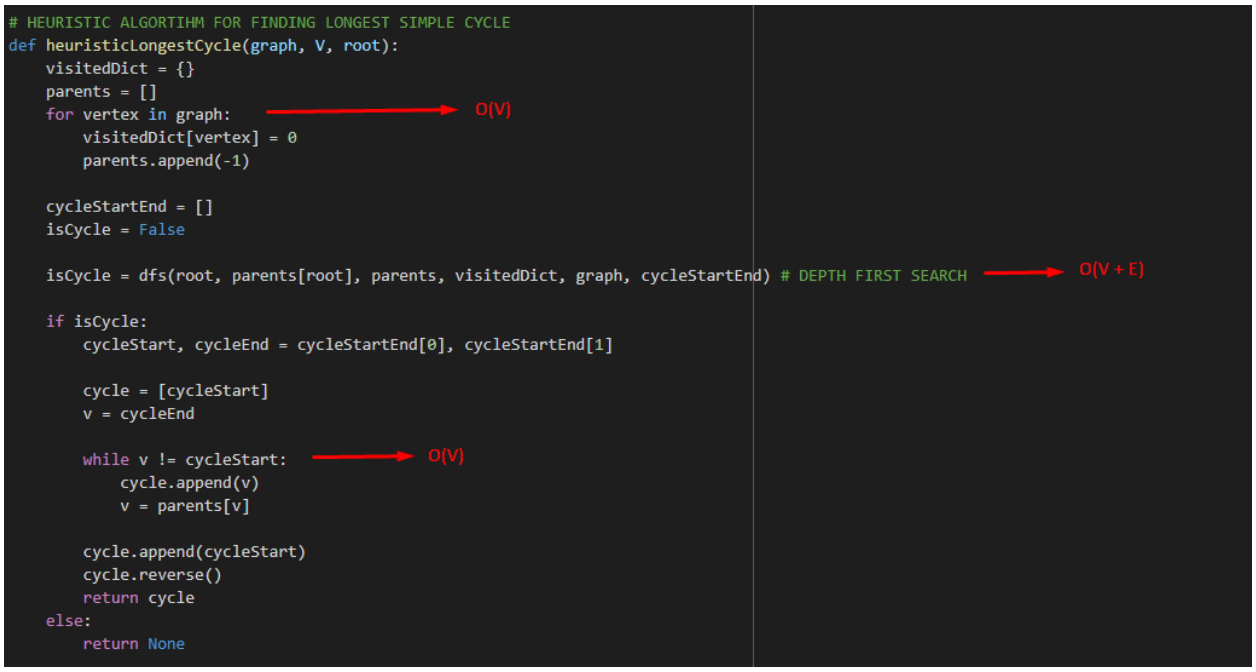
Algorithm 2 DepthFirstSearch (root, Parents, Visited, graph, CycleInfo)

```
Mark the root as visited
for each neighbour of the root do
  if neighbour is not parent of the root then

    if neighbour is not visited then

      Set neighbours parent to the root
      cycleExists = DepthFirstSearch(neighbour, Parents, Visited, graph,
      CycleInfo)
      if cycleExists then
        return true
      end if

    else if neighbour is visited then

      A cycle is found. Append the neighbour and the root to the CycleInfo
      return true

    end if
  end if
end for

Couldn't find a cycle in this path, mark the roots visited information as 2
return false
```

ALGORITHM ANALYSIS

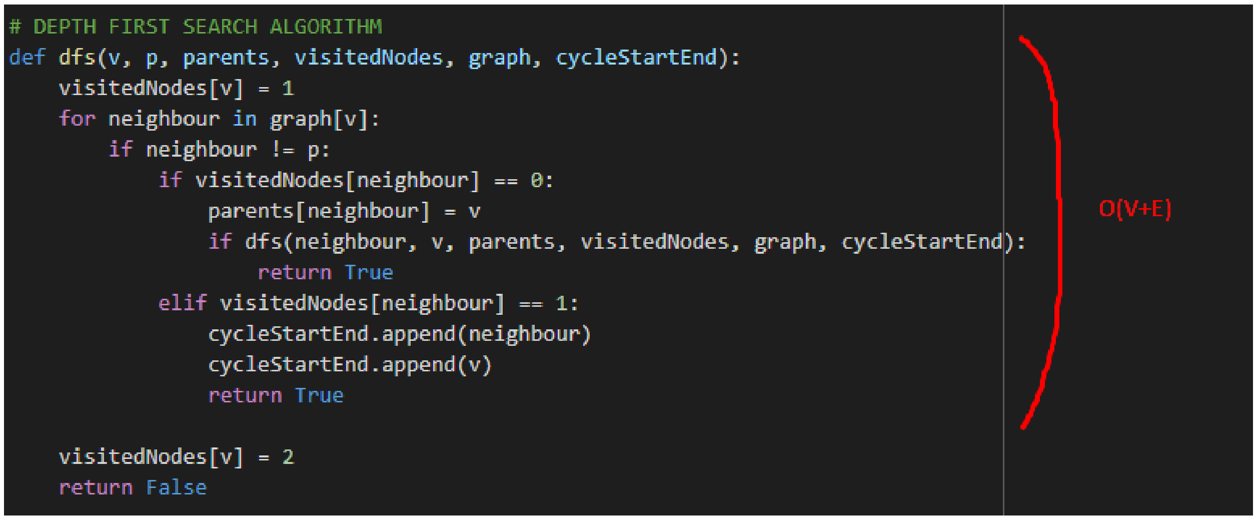
TIME COMPLEXITY

Below, the algorithm implementation can be found.[8]



```
# HEURISTIC ALGORTIHM FOR FINDING LONGEST SIMPLE CYCLE
def heuristicLongestCycle(graph, V, root):
    visitedDict = {}
    parents = []
    for vertex in graph:
        visitedDict[vertex] = 0
        parents.append(-1)

    cycleStartEnd = []
    isCycle = False

    isCycle = dfs(root, parents[root], parents, visitedDict, graph, cycleStartEnd) # DEPTH FIRST SEARCH

    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
```

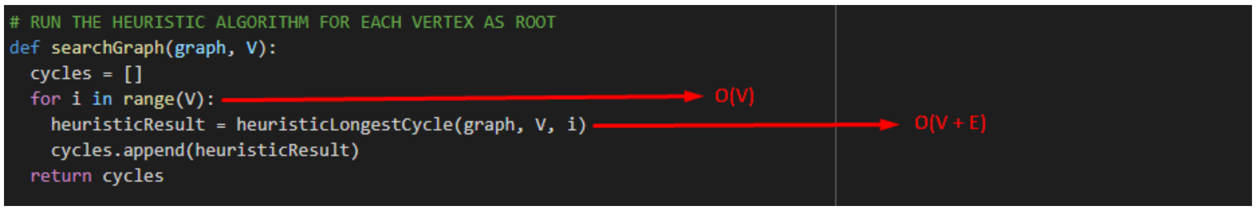
Time complexity: O(V) + O(V + E) + O(V) = O(V + E)



```
# DEPTH FIRST SEARCH ALGORITHM
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
```

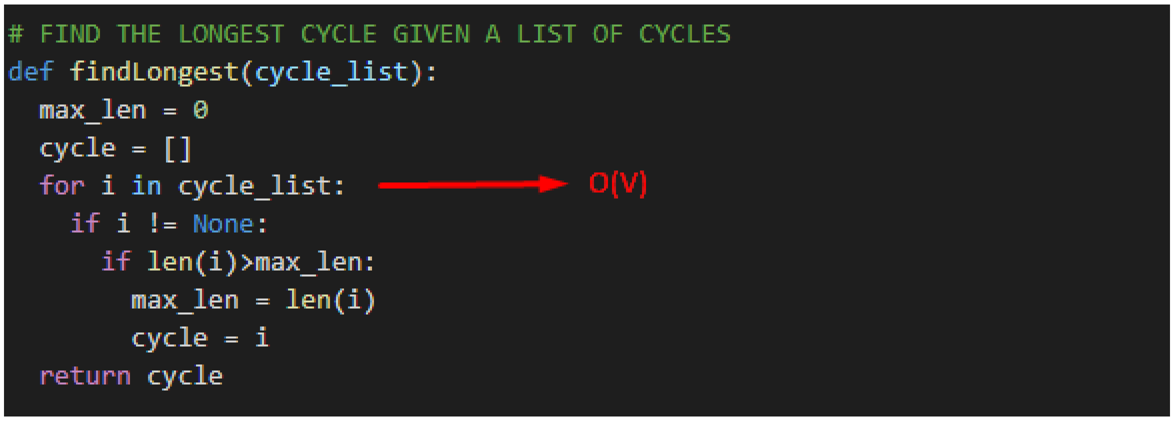
Time complexity: O(V + E)

[8] https://cp-algorithms.com/graph/finding-cycle.html



```
# RUN THE HEURISTIC ALGORITHM FOR EACH VERTEX AS ROOT
def searchGraph(graph, V):
  cycles = []
  for i in range(V):
    heuristicResult = heuristicLongestCycle(graph, V, i)
    cycles.append(heuristicResult)
  return cycles
```

Time complexity: O(V) + O(V + E) = O(V + E)



```
# FIND THE LONGEST CYCLE GIVEN A LIST OF CYCLES
def findLongest(cycle_list):
  max_len = 0
  cycle = []
  for i in cycle_list:
    if i != None:
      if len(i)>max_len:
        max_len = len(i)
        cycle = i
  return cycle
```

Time complexity: O(V)

The algorithm executes depth first search in O(V + E) for each distinct vertex. It finds the longest simple cycle from an array with the maximum length of V. Therefore, the worst case time complexity of the heuristic algorithm is:

$$O(V) + O(V^2 + E*V) = \mathbf{O(V^2 + E*V)}$$

- When E is approximately V: $O(V^2)$
- When E is approximately V^2: $O(V^3)$

This shows that the algorithm is expected to perform better in terms of running time when the input graph is sparse.

SPACE COMPLEXITY

The algorithm is not an inplace algorithm meaning that the memory provided by input, the graph, is not enough to execute the algorithm. The input itself has space complexity of **O(V * E)** as it is a dictionary that holds all the vertices and the corresponding edges for each of them. This complexity tends to translate into **O(V^2)** as the number of edges is close to the number of vertices, and **O(V^3)** when the number of edges is close to the square of the number of vertices. Auxiliary space required to execute the algorithm is as follows:

- Array with parent information of each vertex: **O(V)**
- Array with visited information of each vertex: **O(V)**
- Array to hold the cycle found for the current root: **O(V + 1)**
- Matrix to hold all the cycles found: O(V * (V + 1)) = **O(V^2)**

The depth-first search algorithm uses memory provided by the heuristic algorithm thus it does not require additional memory. Finally, the worst-case space complexity can be computed as follows:

$$O(V * E) + O(V) + O(V) + O(V^2) = \mathbf{O(V^3)}$$

EXPERIMENTAL ANALYSIS

RUNNING TIME EXPERIMENTAL ANALYSIS

This section examines the running time of the algorithm which is implemented using Python3. The algorithm is analysed with a variety of different inputs where the standard deviation, the standard error is given together with the interval of confidence. Running time of the algorithm is plotted and observed for different input parameters.

A. Standard Deviation and Standard Error

Standard deviation and standard error is found using “numpy” and “math” libraries according to the formula below:

$$\mathrm{SD} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}} \qquad SE = \frac{\sigma}{\sqrt{n}}$$

```
sd = np.std(times) #sample standard deviation
sm = sd / math.sqrt(N) #standard error
```

B. Running Time

Since the time complexity of the algorithm depends both on the number of vertices and the number of edges, experimental analysis is done by keeping one of the terms constant and changing the other one. A confidence interval is given by the following equation:

[m - t*sm , m + t*sm],

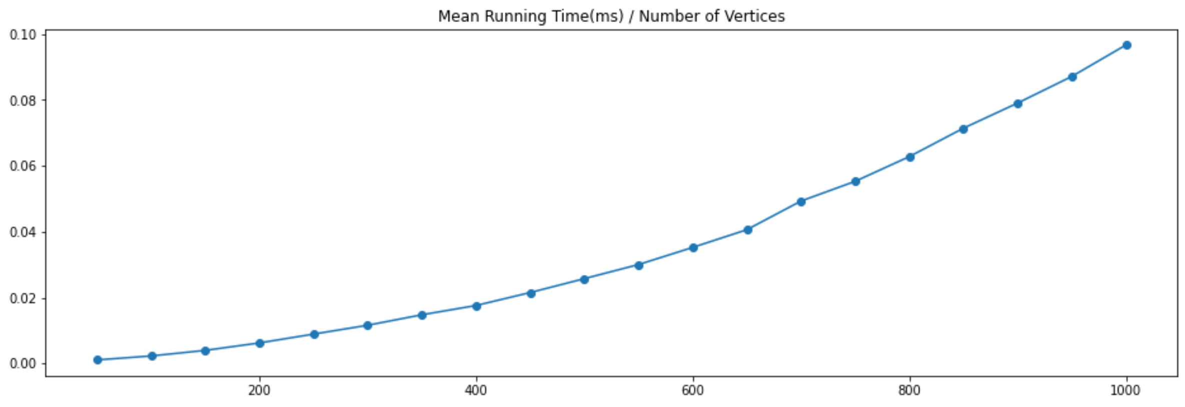
where m is the mean runtime, sm is the standard error and t is the t-value for the given confidence level.

Number of edges is constant (E = 200)

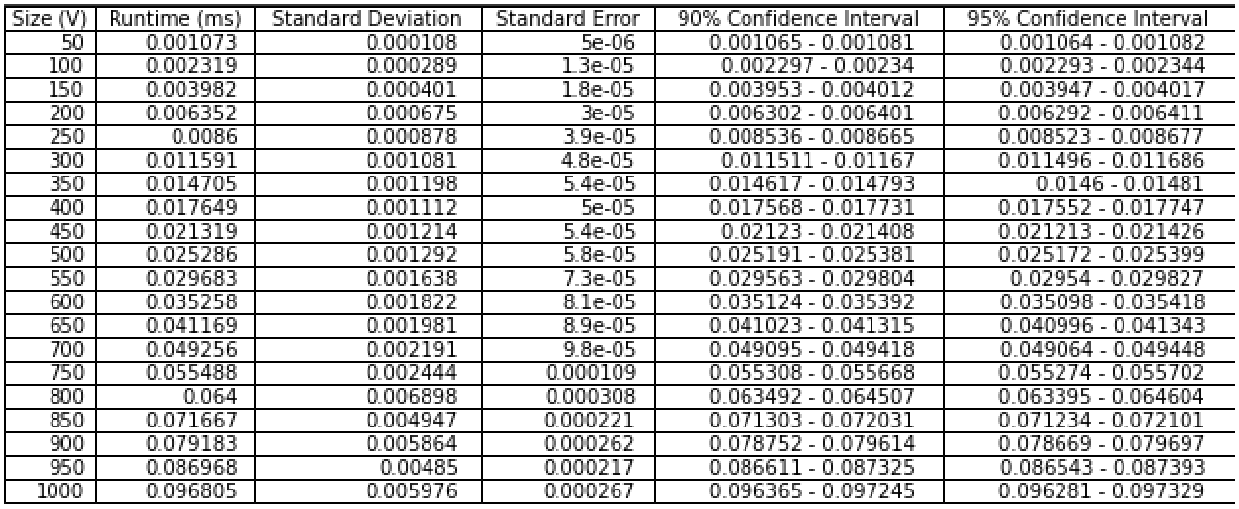
In this part, the number of edges is kept constant but the number of vertices ranges from 50 to 1000 with a step size of 50.

Number of samples per row = 100

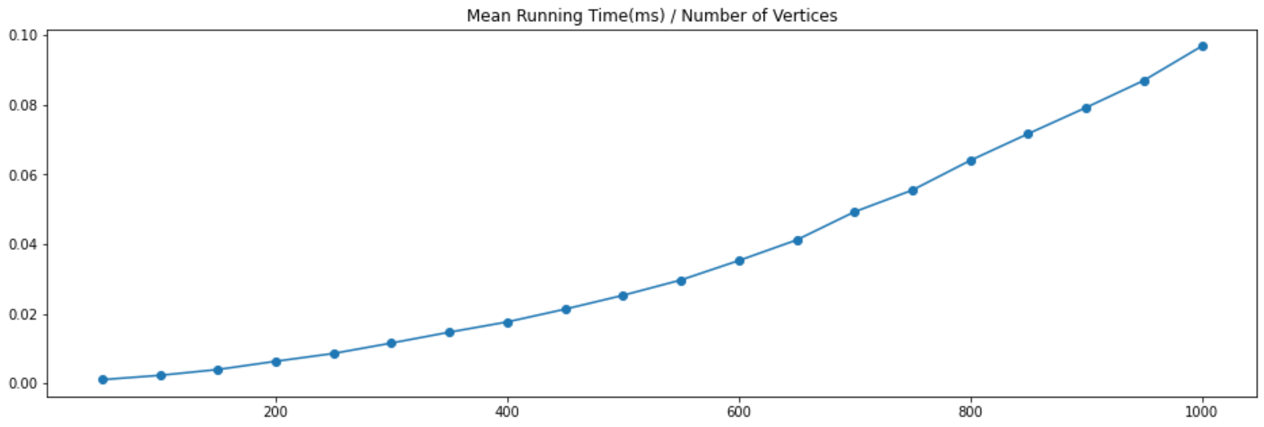
Size (V)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.001087	0.000212	2.1e-05	0.001052 - 0.001122	0.001045 - 0.001128
100	0.002288	0.000232	2.3e-05	0.002249 - 0.002326	0.002242 - 0.002333
150	0.003979	0.000388	3.9e-05	0.003915 - 0.004043	0.003903 - 0.004055
200	0.006244	0.00074	7.4e-05	0.006122 - 0.006366	0.006099 - 0.006389
250	0.008921	0.000878	8.8e-05	0.008777 - 0.009066	0.008749 - 0.009093
300	0.011605	0.000941	9.4e-05	0.011451 - 0.01176	0.011421 - 0.01179
350	0.014778	0.000992	9.9e-05	0.014615 - 0.014941	0.014584 - 0.014973
400	0.017593	0.001032	0.000103	0.017423 - 0.017763	0.017391 - 0.017795
450	0.02152	0.000987	9.9e-05	0.021358 - 0.021682	0.021326 - 0.021713
500	0.025741	0.001216	0.000122	0.025541 - 0.025941	0.025503 - 0.025979
550	0.030013	0.001381	0.000138	0.029786 - 0.03024	0.029742 - 0.030284
600	0.035207	0.002134	0.000213	0.034856 - 0.035558	0.034789 - 0.035626
650	0.040582	0.001764	0.000176	0.040292 - 0.040872	0.040236 - 0.040928
700	0.049274	0.001955	0.000195	0.048952 - 0.049596	0.048891 - 0.049657
750	0.055262	0.002267	0.000227	0.054889 - 0.055635	0.054818 - 0.055706
800	0.062755	0.002647	0.000265	0.062319 - 0.06319	0.062236 - 0.063273
850	0.071377	0.009232	0.000923	0.069858 - 0.072895	0.069567 - 0.073186
900	0.079002	0.002836	0.000284	0.078536 - 0.079469	0.078446 - 0.079558
950	0.087074	0.00261	0.000261	0.086644 - 0.087503	0.086562 - 0.087585
1000	0.096614	0.008226	0.000823	0.09526 - 0.097967	0.095001 - 0.098226



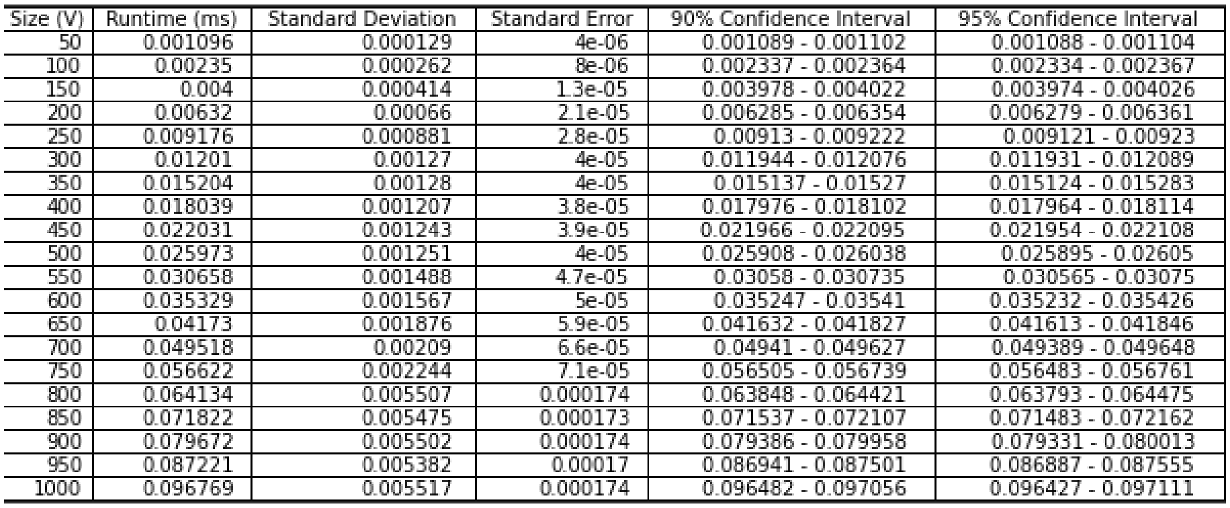
- Number of samples per row = 500



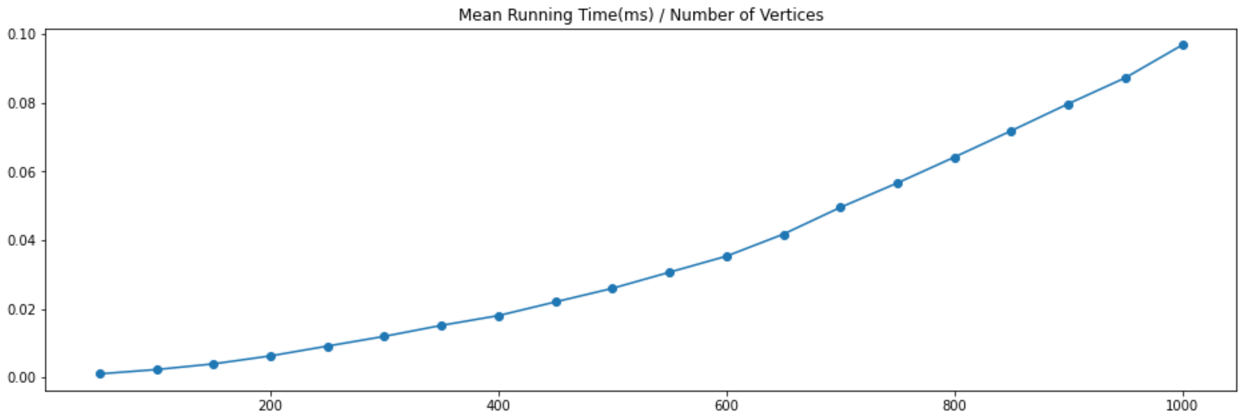
Size (V)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.001073	0.000108	5e-06	0.001065 - 0.001081	0.001064 - 0.001082
100	0.002319	0.000289	1.3e-05	0.002297 - 0.00234	0.002293 - 0.002344
150	0.003982	0.000401	1.8e-05	0.003953 - 0.004012	0.003947 - 0.004017
200	0.006352	0.000675	3e-05	0.006302 - 0.006401	0.006292 - 0.006411
250	0.0086	0.000878	3.9e-05	0.008536 - 0.008665	0.008523 - 0.008677
300	0.011591	0.001081	4.8e-05	0.011511 - 0.01167	0.011496 - 0.011686
350	0.014705	0.001198	5.4e-05	0.014617 - 0.014793	0.0146 - 0.01481
400	0.017649	0.001112	5e-05	0.017568 - 0.017731	0.017552 - 0.017747
450	0.021319	0.001214	5.4e-05	0.02123 - 0.021408	0.021213 - 0.021426
500	0.025286	0.001292	5.8e-05	0.025191 - 0.025381	0.025172 - 0.025399
550	0.029683	0.001638	7.3e-05	0.029563 - 0.029804	0.02954 - 0.029827
600	0.035258	0.001822	8.1e-05	0.035124 - 0.035392	0.035098 - 0.035418
650	0.041169	0.001981	8.9e-05	0.041023 - 0.041315	0.040996 - 0.041343
700	0.049256	0.002191	9.8e-05	0.049095 - 0.049418	0.049064 - 0.049448
750	0.055488	0.002444	0.000109	0.055308 - 0.055668	0.055274 - 0.055702
800	0.064	0.006898	0.000308	0.063492 - 0.064507	0.063395 - 0.064604
850	0.071667	0.004947	0.000221	0.071303 - 0.072031	0.071234 - 0.072101
900	0.079183	0.005864	0.000262	0.078752 - 0.079614	0.078669 - 0.079697
950	0.086968	0.00485	0.000217	0.086611 - 0.087325	0.086543 - 0.087393
1000	0.096805	0.005976	0.000267	0.096365 - 0.097245	0.096281 - 0.097329



- Number of samples per row = 1000



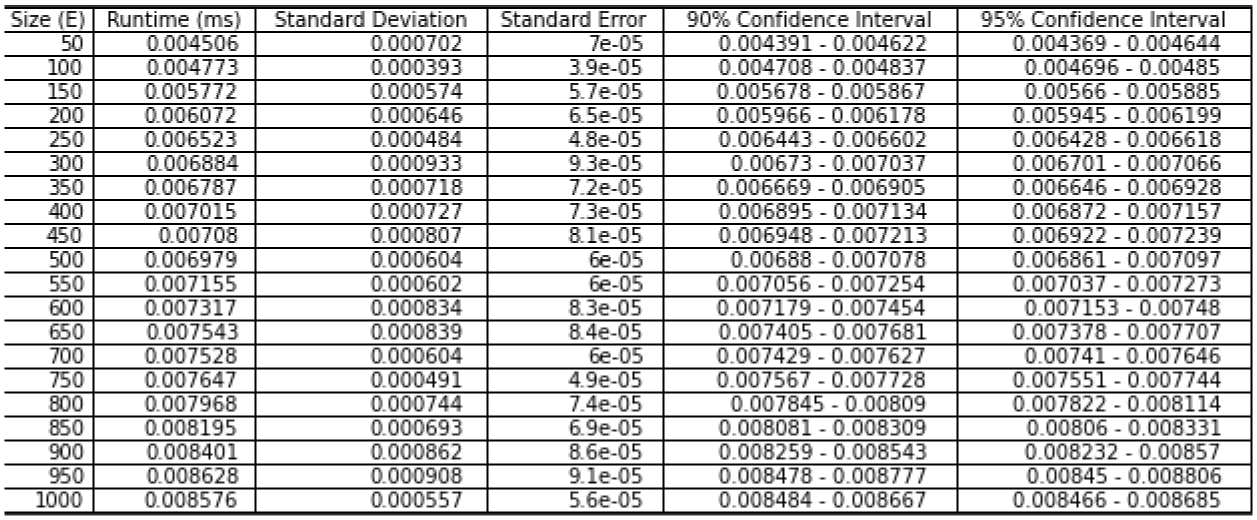
Size (V)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.001096	0.000129	4e-06	0.001089 - 0.001102	0.001088 - 0.001104
100	0.00235	0.000262	8e-06	0.002337 - 0.002364	0.002334 - 0.002367
150	0.004	0.000414	1.3e-05	0.003978 - 0.004022	0.003974 - 0.004026
200	0.00632	0.00066	2.1e-05	0.006285 - 0.006354	0.006279 - 0.006361
250	0.009176	0.000881	2.8e-05	0.00913 - 0.009222	0.009121 - 0.00923
300	0.01201	0.00127	4e-05	0.011944 - 0.012076	0.011931 - 0.012089
350	0.015204	0.00128	4e-05	0.015137 - 0.01527	0.015124 - 0.015283
400	0.018039	0.001207	3.8e-05	0.017976 - 0.018102	0.017964 - 0.018114
450	0.022031	0.001243	3.9e-05	0.021966 - 0.022095	0.021954 - 0.022108
500	0.025973	0.001251	4e-05	0.025908 - 0.026038	0.025895 - 0.02605
550	0.030658	0.001488	4.7e-05	0.03058 - 0.030735	0.030565 - 0.03075
600	0.035329	0.001567	5e-05	0.035247 - 0.03541	0.035232 - 0.035426
650	0.04173	0.001876	5.9e-05	0.041632 - 0.041827	0.041613 - 0.041846
700	0.049518	0.00209	6.6e-05	0.04941 - 0.049627	0.049389 - 0.049648
750	0.056622	0.002244	7.1e-05	0.056505 - 0.056739	0.056483 - 0.056761
800	0.064134	0.005507	0.000174	0.063848 - 0.064421	0.063793 - 0.064475
850	0.071822	0.005475	0.000173	0.071537 - 0.072107	0.071483 - 0.072162
900	0.079672	0.005502	0.000174	0.079386 - 0.079958	0.079331 - 0.080013
950	0.087221	0.005382	0.00017	0.086941 - 0.087501	0.086887 - 0.087555
1000	0.096769	0.005517	0.000174	0.096482 - 0.097056	0.096427 - 0.097111



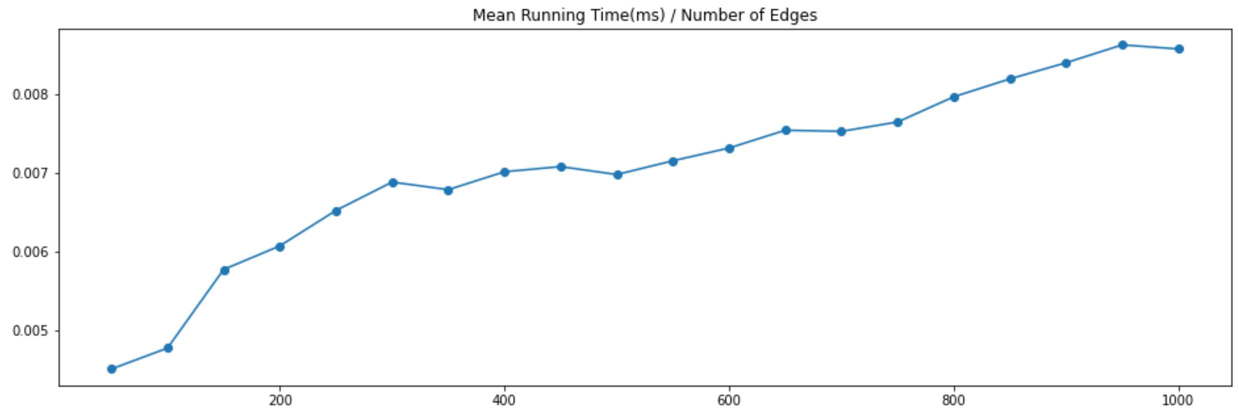
Number of vertices is constant (V = 200)

Different from the previous part, the number of vertices is kept constant but the number of edges ranges from 50 to 1000 with a step size of 50.

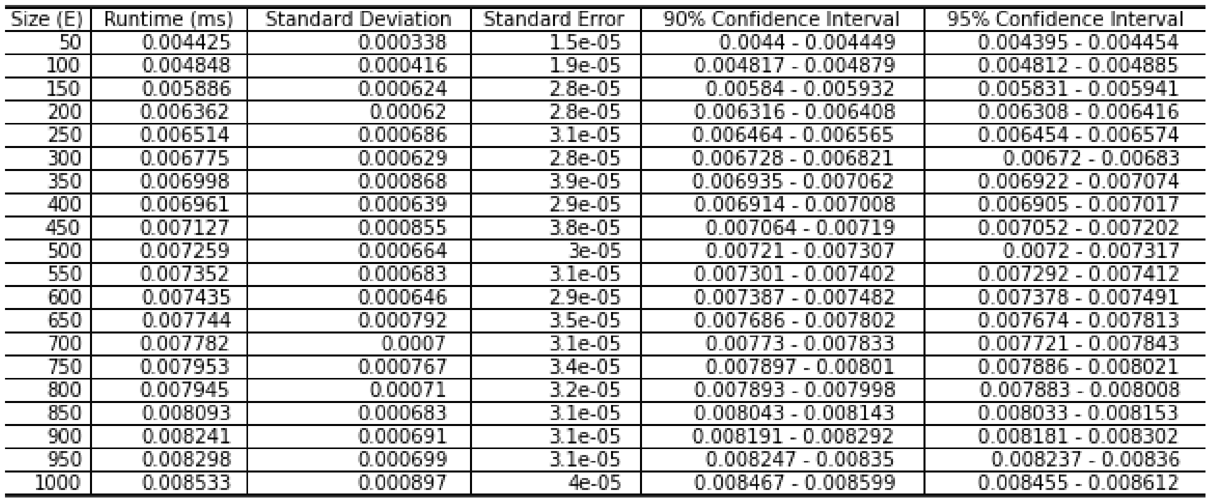
- Number of samples per row = 100



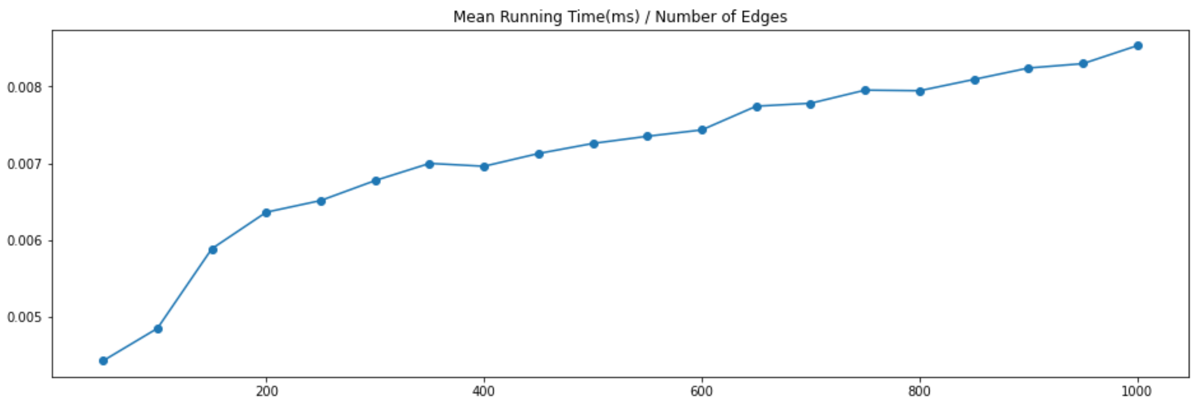
Size (E)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.004506	0.000702	7e-05	0.004391 - 0.004622	0.004369 - 0.004644
100	0.004773	0.000393	3.9e-05	0.004708 - 0.004837	0.004696 - 0.00485
150	0.005772	0.000574	5.7e-05	0.005678 - 0.005867	0.00566 - 0.005885
200	0.006072	0.000646	6.5e-05	0.005966 - 0.006178	0.005945 - 0.006199
250	0.006523	0.000484	4.8e-05	0.006443 - 0.006602	0.006428 - 0.006618
300	0.006884	0.000933	9.3e-05	0.00673 - 0.007037	0.006701 - 0.007066
350	0.006787	0.000718	7.2e-05	0.006669 - 0.006905	0.006646 - 0.006928
400	0.007015	0.000727	7.3e-05	0.006895 - 0.007134	0.006872 - 0.007157
450	0.00708	0.000807	8.1e-05	0.006948 - 0.007213	0.006922 - 0.007239
500	0.006979	0.000604	6e-05	0.00688 - 0.007078	0.006861 - 0.007097
550	0.007155	0.000602	6e-05	0.007056 - 0.007254	0.007037 - 0.007273
600	0.007317	0.000834	8.3e-05	0.007179 - 0.007454	0.007153 - 0.00748
650	0.007543	0.000839	8.4e-05	0.007405 - 0.007681	0.007378 - 0.007707
700	0.007528	0.000604	6e-05	0.007429 - 0.007627	0.00741 - 0.007646
750	0.007647	0.000491	4.9e-05	0.007567 - 0.007728	0.007551 - 0.007744
800	0.007968	0.000744	7.4e-05	0.007845 - 0.00809	0.007822 - 0.008114
850	0.008195	0.000693	6.9e-05	0.008081 - 0.008309	0.00806 - 0.008331
900	0.008401	0.000862	8.6e-05	0.008259 - 0.008543	0.008232 - 0.00857
950	0.008628	0.000908	9.1e-05	0.008478 - 0.008777	0.00845 - 0.008806
1000	0.008576	0.000557	5.6e-05	0.008484 - 0.008667	0.008466 - 0.008685



- Number of samples per row = 500

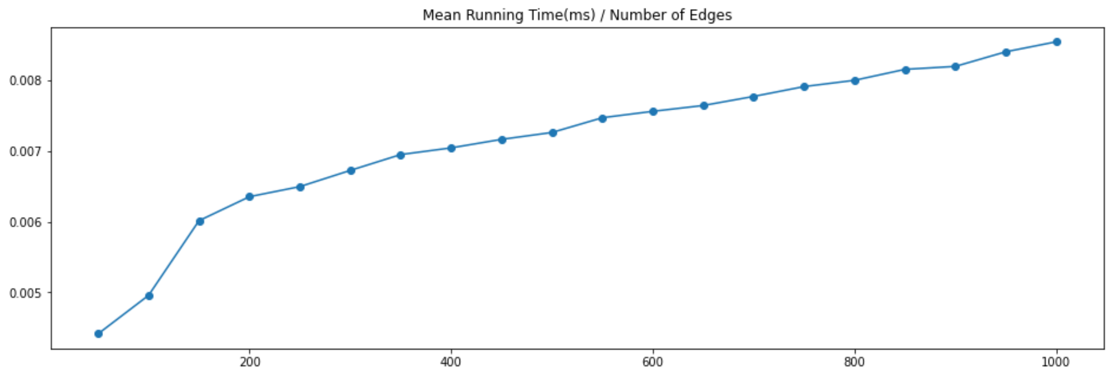


Size (E)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.004425	0.000338	1.5e-05	0.0044 - 0.004449	0.004395 - 0.004454
100	0.004848	0.000416	1.9e-05	0.004817 - 0.004879	0.004812 - 0.004885
150	0.005886	0.000624	2.8e-05	0.00584 - 0.005932	0.005831 - 0.005941
200	0.006362	0.00062	2.8e-05	0.006316 - 0.006408	0.006308 - 0.006416
250	0.006514	0.000686	3.1e-05	0.006464 - 0.006565	0.006454 - 0.006574
300	0.006775	0.000629	2.8e-05	0.006728 - 0.006821	0.00672 - 0.00683
350	0.006998	0.000868	3.9e-05	0.006935 - 0.007062	0.006922 - 0.007074
400	0.006961	0.000639	2.9e-05	0.006914 - 0.007008	0.006905 - 0.007017
450	0.007127	0.000855	3.8e-05	0.007064 - 0.00719	0.007052 - 0.007202
500	0.007259	0.000664	3e-05	0.00721 - 0.007307	0.0072 - 0.007317
550	0.007352	0.000683	3.1e-05	0.007301 - 0.007402	0.007292 - 0.007412
600	0.007435	0.000646	2.9e-05	0.007387 - 0.007482	0.007378 - 0.007491
650	0.007744	0.000792	3.5e-05	0.007686 - 0.007802	0.007674 - 0.007813
700	0.007782	0.0007	3.1e-05	0.00773 - 0.007833	0.007721 - 0.007843
750	0.007953	0.000767	3.4e-05	0.007897 - 0.00801	0.007886 - 0.008021
800	0.007945	0.00071	3.2e-05	0.007893 - 0.007998	0.007883 - 0.008008
850	0.008093	0.000683	3.1e-05	0.008043 - 0.008143	0.008033 - 0.008153
900	0.008241	0.000691	3.1e-05	0.008191 - 0.008292	0.008181 - 0.008302
950	0.008298	0.000699	3.1e-05	0.008247 - 0.00835	0.008237 - 0.00836
1000	0.008533	0.000897	4e-05	0.008467 - 0.008599	0.008455 - 0.008612



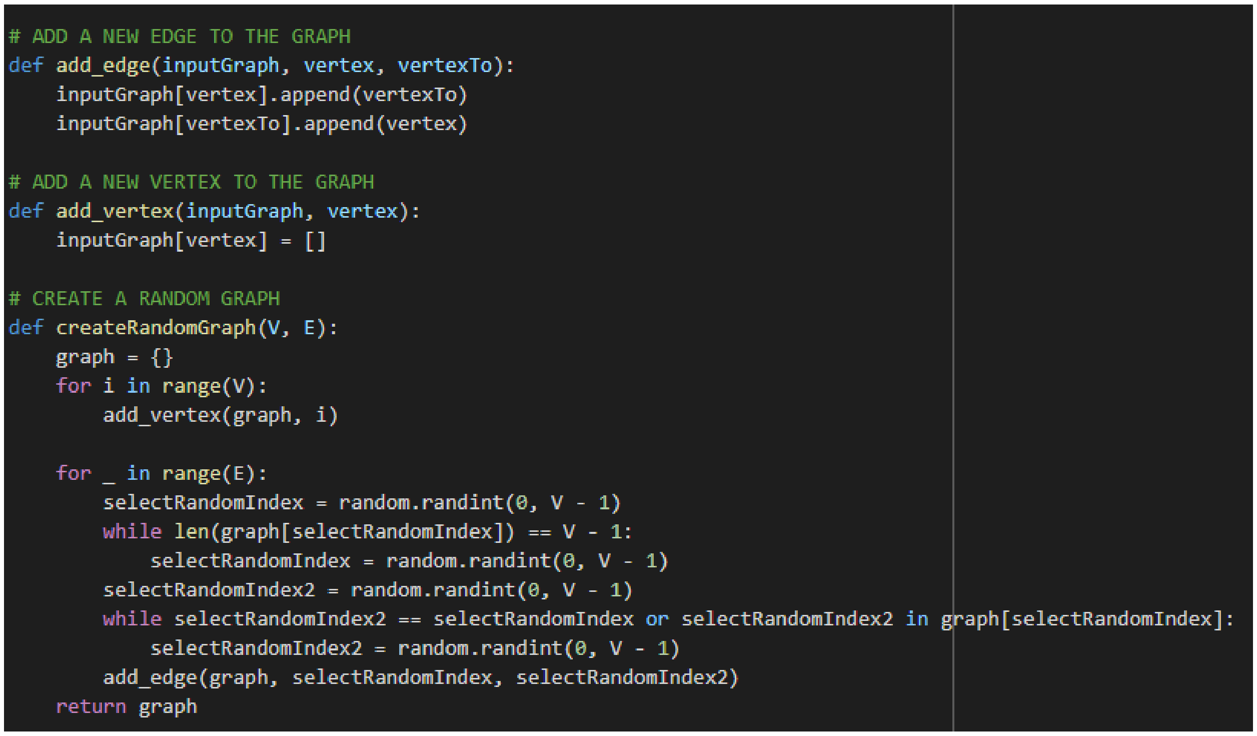
- Number of samples per row = 1000

Size (E)	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
50	0.004412	0.003749	0.000119	0.004217 - 0.004607	0.00418 - 0.004645
100	0.004957	0.000491	1.6e-05	0.004931 - 0.004982	0.004926 - 0.004987
150	0.006013	0.000646	2e-05	0.005979 - 0.006047	0.005973 - 0.006053
200	0.006352	0.000668	2.1e-05	0.006318 - 0.006387	0.006311 - 0.006394
250	0.006495	0.000657	2.1e-05	0.006461 - 0.006529	0.006454 - 0.006536
300	0.006724	0.000682	2.2e-05	0.006689 - 0.00676	0.006682 - 0.006767
350	0.006949	0.000847	2.7e-05	0.006905 - 0.006993	0.006896 - 0.007001
400	0.007043	0.000726	2.3e-05	0.007005 - 0.007081	0.006998 - 0.007088
450	0.007164	0.00066	2.1e-05	0.007129 - 0.007198	0.007123 - 0.007205
500	0.007261	0.000746	2.4e-05	0.007222 - 0.0073	0.007215 - 0.007307
550	0.00747	0.000774	2.4e-05	0.00743 - 0.00751	0.007422 - 0.007518
600	0.007561	0.000787	2.5e-05	0.00752 - 0.007601	0.007512 - 0.007609
650	0.007643	0.000723	2.3e-05	0.007605 - 0.00768	0.007598 - 0.007688
700	0.007772	0.000733	2.3e-05	0.007733 - 0.00781	0.007726 - 0.007817
750	0.007911	0.000739	2.3e-05	0.007872 - 0.007949	0.007865 - 0.007957
800	0.007999	0.000749	2.4e-05	0.00796 - 0.008038	0.007953 - 0.008046
850	0.008154	0.000747	2.4e-05	0.008115 - 0.008193	0.008108 - 0.008201
900	0.008196	0.000715	2.3e-05	0.008159 - 0.008233	0.008152 - 0.008241
950	0.008403	0.000782	2.5e-05	0.008362 - 0.008444	0.008355 - 0.008452
1000	0.008545	0.000809	2.6e-05	0.008502 - 0.008587	0.008494 - 0.008595



CORRECTNESS

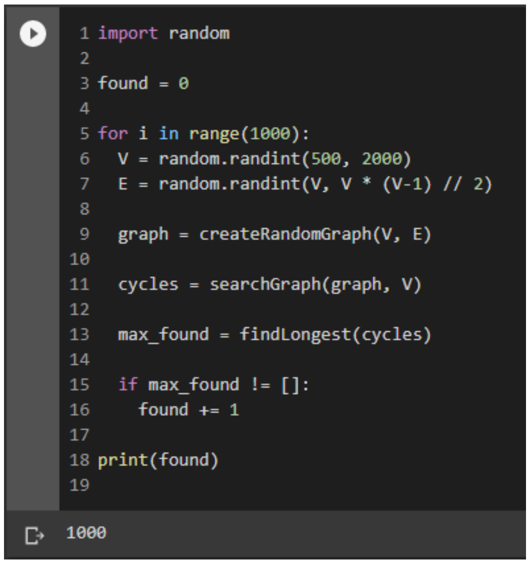
Since the algorithm is a heuristic, it does not always yield the correct answer for the longest simple cycle. However, given the graph has at least one it returns a that is not necessarily the longest. Additionally, in the case of an acyclic graph, it successfully terminates with returning as no detected cycles. In order to experimentally show the mentioned correctness, the algorithm is given 1000 randomly generated graphs and it is asked to find a cycle. Number of vertices is chosen between [500, 2000] and the number of edges is chosen between [V, V * (V - 1) / 2]. Because it is reliable for the random graph to have at least one cycle



```python
# ADD A NEW EDGE TO THE GRAPH
def add_edge(inputGraph, vertex, vertexTo):
    inputGraph[vertex].append(vertexTo)
    inputGraph[vertexTo].append(vertex)

# ADD A NEW VERTEX TO THE GRAPH
def add_vertex(inputGraph, vertex):
    inputGraph[vertex] = []

# CREATE A RANDOM GRAPH
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
```

and have no more than maximum number of edges.

Functions for creating a random graph are given above.



```python
import random

found = 0

for i in range(1000):
  V = random.randint(500, 2000)
  E = random.randint(V, V * (V-1) // 2)

  graph = createRandomGraph(V, E)

  cycles = searchGraph(graph, V)

  max_found = findLongest(cycles)

  if max_found != []:
    found += 1

print(found)
```

```
1000
```

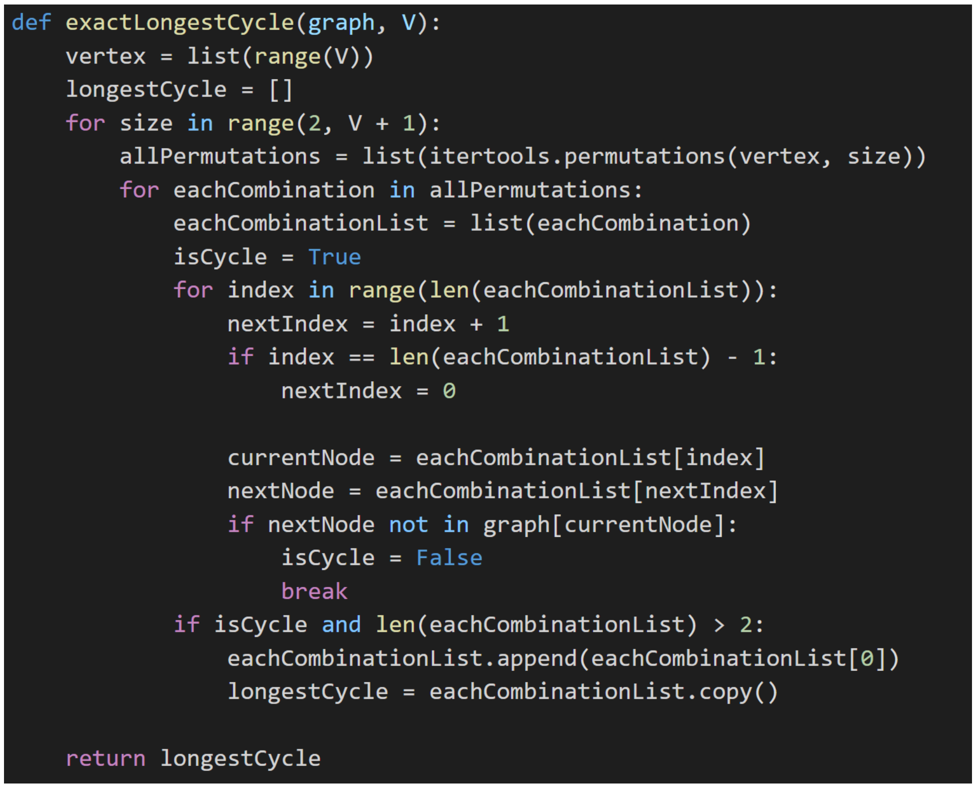
It can be seen that the algorithm yields a result for all 1000 random graph inputs.

RATIO BOUND

Ratio bound is checked by creating random graphs with the given vertex and edge numbers for different samples (N). Both edge and vertex numbers range from 5 to 10. It wasn’t feasible to check the ratio bound with relatively big sized graphs since finding the ratio bound requires us to find the exact length of the longest cycle with brute force besides the cycle found by the heuristic algorithm. Ratio bound is found using the following:

Ratio bound = size(Optimum Longest Cycle) / size(Longest Cycle Found by Heuristic Algorithm)

Brute force algorithm to find the longest circuit:



```python
def exactLongestCycle(graph, V):
    vertex = list(range(V))
    longestCycle = []
    for size in range(2, V + 1):
        allPermutations = list(itertools.permutations(vertex, size))
        for eachCombination in allPermutations:
            eachCombinationList = list(eachCombination)
            isCycle = True
            for index in range(len(eachCombinationList)):
                nextIndex = index + 1
                if index == len(eachCombinationList) - 1:
                    nextIndex = 0

                currentNode = eachCombinationList[index]
                nextNode = eachCombinationList[nextIndex]
                if nextNode not in graph[currentNode]:
                    isCycle = False
                    break
            if isCycle and len(eachCombinationList) > 2:
                eachCombinationList.append(eachCombinationList[0])
                longestCycle = eachCombinationList.copy()

    return longestCycle
```

Brute force algorithm basically tries to find the longest circuit by trying every possible combination of different paths.

Firstly, the vertex number is kept as 10 and the edge number is changed as 5, 6, 7, 8 and 9. N states the number of graphs generated (i.e sample size) with the determined vertex - edge number. It is observed that when the edge number increases keeping the vertex number constant, the ratio bound also increases. This can be interpreted as, the heuristic algorithm works better when the edge/vertex ratio is lower. So, it works better for sparse graphs.

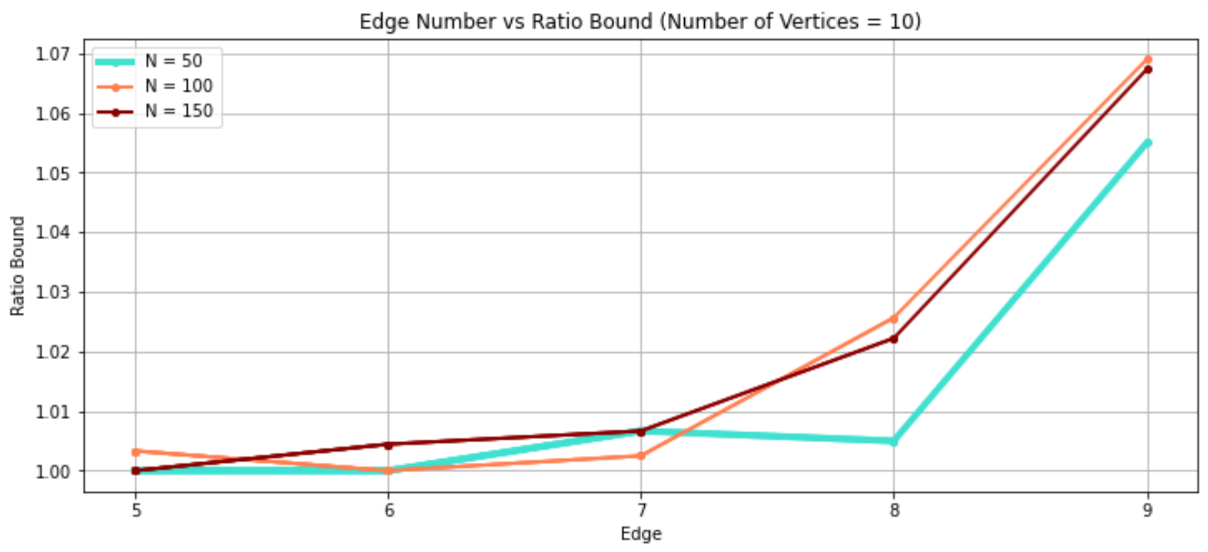


Figure 1

Secondly, the edge number is kept as 10 and the vertex number is changed as 5, 6, 7,8 and 9. Again, denser graphs gave worse ratio bound overall. However, the relation is not that obvious as in Figure 1.

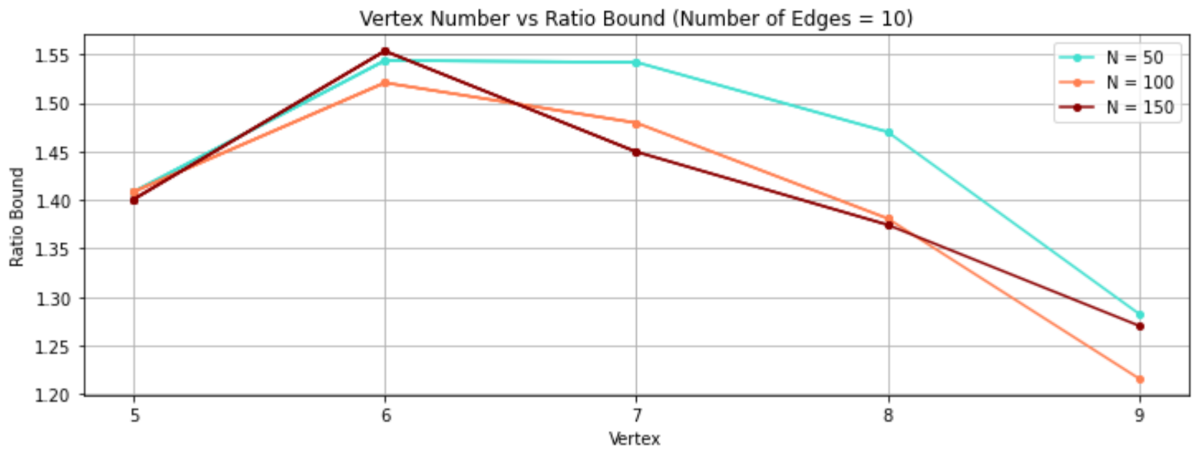
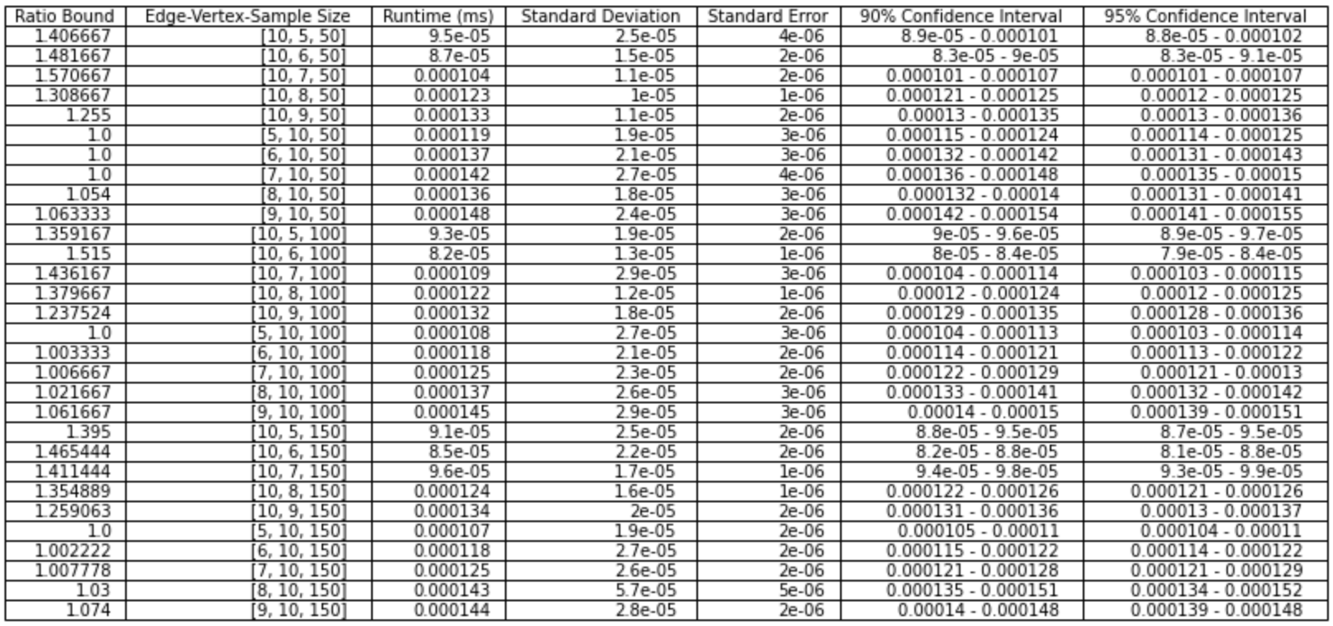


Figure 2

Statistical tests were performed to investigate further.

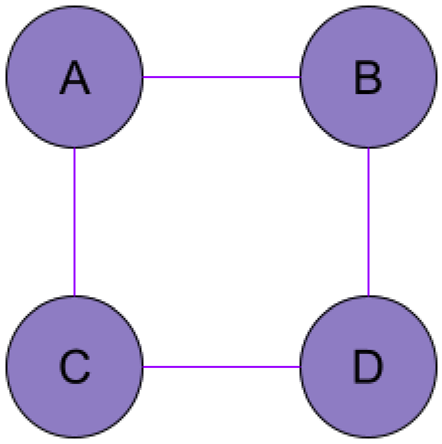


Ratio Bound	Edge-Vertex-Sample Size	Runtime (ms)	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
1.406667	[10, 5, 50]	9.5e-05	2.5e-05	4e-06	8.9e-05 - 0.000101	8.8e-05 - 0.000102
1.481667	[10, 6, 50]	8.7e-05	1.5e-05	2e-06	8.3e-05 - 9e-05	8.3e-05 - 9.1e-05
1.570667	[10, 7, 50]	0.000104	1.1e-05	2e-06	0.000101 - 0.000107	0.000101 - 0.000107
1.308667	[10, 8, 50]	0.000123	1e-05	1e-06	0.000121 - 0.000125	0.00012 - 0.000125
1.255	[10, 9, 50]	0.000133	1.1e-05	2e-06	0.00013 - 0.000135	0.00013 - 0.000136
1.0	[5, 10, 50]	0.000119	1.9e-05	3e-06	0.000115 - 0.000124	0.000114 - 0.000125
1.0	[6, 10, 50]	0.000137	2.1e-05	3e-06	0.000132 - 0.000142	0.000131 - 0.000143
1.0	[7, 10, 50]	0.000142	2.7e-05	4e-06	0.000136 - 0.000148	0.000135 - 0.00015
1.054	[8, 10, 50]	0.000136	1.8e-05	3e-06	0.000132 - 0.00014	0.000131 - 0.000141
1.063333	[9, 10, 50]	0.000148	2.4e-05	3e-06	0.000142 - 0.000154	0.000141 - 0.000155
1.359167	[10, 5, 100]	9.3e-05	1.9e-05	2e-06	9e-05 - 9.6e-05	8.9e-05 - 9.7e-05
1.515	[10, 6, 100]	8.2e-05	1.3e-05	1e-06	8e-05 - 8.4e-05	7.9e-05 - 8.4e-05
1.436167	[10, 7, 100]	0.000109	2.9e-05	3e-06	0.000104 - 0.000114	0.000103 - 0.000115
1.379667	[10, 8, 100]	0.000122	1.2e-05	1e-06	0.00012 - 0.000124	0.00012 - 0.000125
1.237524	[10, 9, 100]	0.000132	1.8e-05	2e-06	0.000129 - 0.000135	0.000128 - 0.000136
1.0	[5, 10, 100]	0.000108	2.7e-05	3e-06	0.000104 - 0.000113	0.000103 - 0.000114
1.003333	[6, 10, 100]	0.000118	2.1e-05	2e-06	0.000114 - 0.000121	0.000113 - 0.000122
1.006667	[7, 10, 100]	0.000125	2.3e-05	2e-06	0.000122 - 0.000129	0.000121 - 0.00013
1.021667	[8, 10, 100]	0.000137	2.6e-05	3e-06	0.000133 - 0.000141	0.000132 - 0.000142
1.061667	[9, 10, 100]	0.000145	2.9e-05	3e-06	0.00014 - 0.00015	0.000139 - 0.000151
1.395	[10, 5, 150]	9.1e-05	2.5e-05	2e-06	8.8e-05 - 9.5e-05	8.7e-05 - 9.5e-05
1.465444	[10, 6, 150]	8.5e-05	2.2e-05	2e-06	8.2e-05 - 8.8e-05	8.1e-05 - 8.8e-05
1.411444	[10, 7, 150]	9.6e-05	1.7e-05	1e-06	9.4e-05 - 9.8e-05	9.3e-05 - 9.9e-05
1.354889	[10, 8, 150]	0.000124	1.6e-05	1e-06	0.000122 - 0.000126	0.000121 - 0.000126
1.259063	[10, 9, 150]	0.000134	2e-05	2e-06	0.000131 - 0.000136	0.00013 - 0.000137
1.0	[5, 10, 150]	0.000107	1.9e-05	2e-06	0.000105 - 0.00011	0.000104 - 0.00011
1.002222	[6, 10, 150]	0.000118	2.7e-05	2e-06	0.000115 - 0.000122	0.000114 - 0.000122
1.007778	[7, 10, 150]	0.000125	2.6e-05	2e-06	0.000121 - 0.000128	0.000121 - 0.000129
1.03	[8, 10, 150]	0.000143	5.7e-05	5e-06	0.000135 - 0.000151	0.000134 - 0.000152
1.074	[9, 10, 150]	0.000144	2.8e-05	2e-06	0.00014 - 0.000148	0.000139 - 0.000148

TESTING

Black box testing is applied to test the suggested heuristic algorithm. Generated random graphs were tested by comparing the results of the heuristic algorithm versus the brute force algorithm implemented. Some of the results obtained are shown below. The path found by the heuristic algorithm is indicated by purple vertices connected with the purple edges for each graph given. In the case the heuristic algorithm produces a wrong result, the longest circuit found by brute force is given with the orange edges and vertices.

Note that sometimes there are more than one circuit that gives the longest path. Since the problem deals with the length of the longest circuit, one of the circuits that gives the maximum number of vertices is shown in the graphs below.

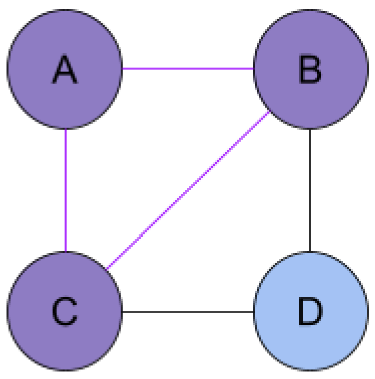


(Z)

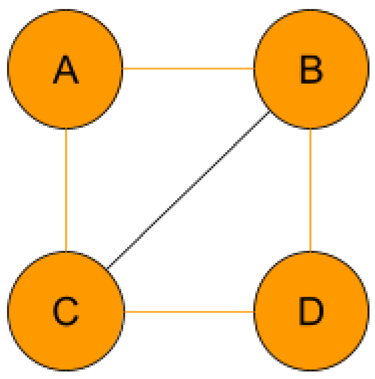
Longest circuit length: 4

Longest circuit length found by the heuristic algorithm: 4

SUCCEEDED: Heuristic algorithm found the longest circuit in the given graph.



(X)

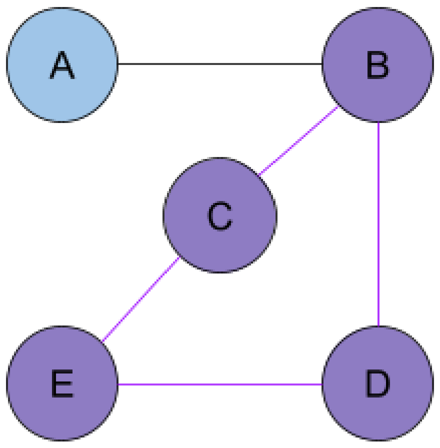


(Y)

Longest circuit length: 4

Longest circuit length found by the heuristic algorithm: 3

FAILED: The heuristic algorithm failed to find the longest circuit (given in the graph X) when there is one more edge added between vertices “b” and “c” to the previously given graph (Z). The heuristic approach gave the length as 3. However, it can be easily seen that the longest circuit has the length 4.

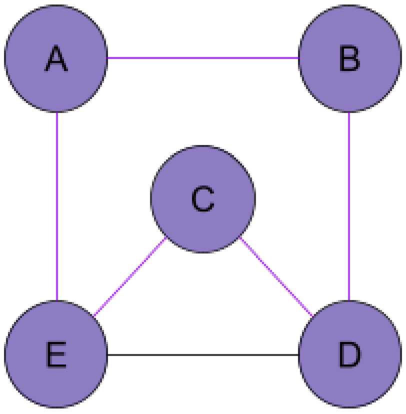


(M)

Longest circuit length: 4

Longest circuit length found by the heuristic algorithm: 4

SUCCEEDED: Heuristic algorithm found the length of the longest circuit in graph M.

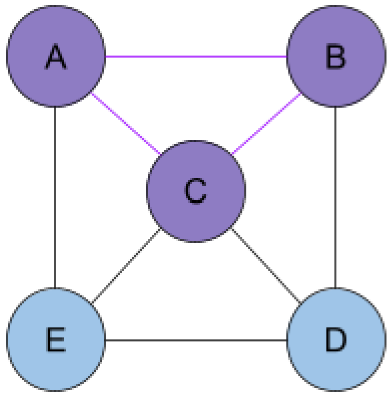


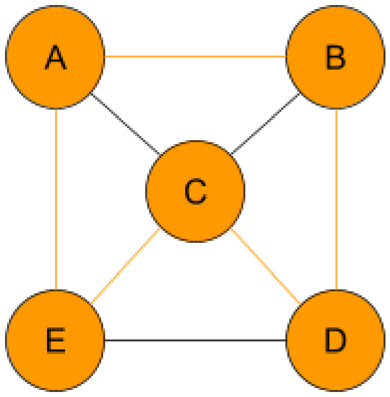
(N)

Longest circuit length: 5

Longest circuit length found by the heuristic algorithm: 5

SUCCEEDED: Heuristic algorithm found the length of the longest circuit correctly in graph N.



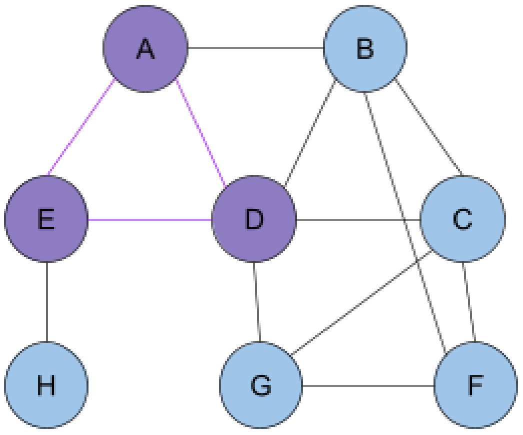


(K) (L)

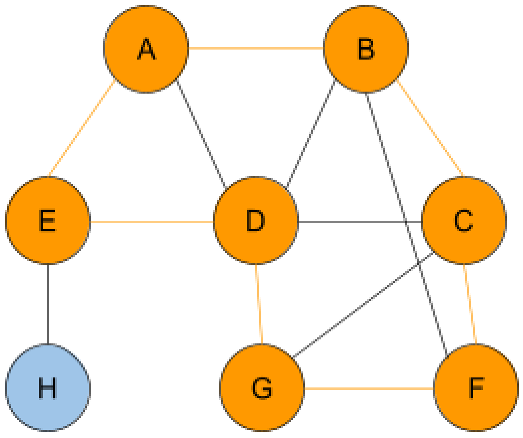
Longest circuit length: 5

Longest circuit length found by the heuristic algorithm: 3

FAILED: The heuristic algorithm failed to find the length of the longest circuit in graph K. The circuit with the optimal length is given in the graph L. The structure of the graph K is similar to the graphs N and M. The difference is, the edge density is increased in graph K. It can be observed that the heuristic algorithm failed when the graph became denser.



(P)



(R)

Longest circuit length: 7

Longest circuit length found by the heuristic algorithm: 3

FAILED: The heuristic algorithm found the length as 3 whereas the brute force algorithm found it as 7.

CONCLUSION & DISCUSSION

In conclusion, the Longest Circuit problem is shown to be NP-Complete as a subset of NP-Hard problems with initially explaining why it is in NP and later how the transformation takes place in order to perform reduction for proving it belongs to NP-Complete. Therefore, the former condition shown to be satisfiable with the problem's verifiability in polynomial time and the latter condition shown by using Hamiltonian Cycle which is known to be NP-Complete and hereby NP-Hard by using the decision version of both problems. Therefore a heuristic algorithm is needed to come up with an optimal solution in polynomial time. The heuristic solution given in this paper runs on a time complexity of $O(V^3)$ and space complexity of $O(V^3)$ using a recursive depth-first search algorithm.

In the experimental analysis running time and the correctness of the heuristic algorithm is examined. Several experimental setups are prepared and it has been observed that when the number of edges kept constant, the algorithm indeed follows a polynomial time complexity. On the other hand, when the number of vertices is constant, the algorithm after some point where the number of edges dominate the number of vertices starts to follow a linear time complexity. These experiments are done for multiple different numbers of samples and the results are analysed for their 95% and 90% confidence intervals. In the correctness part, the algorithm is given a thousand random graphs where all of them have at least a cycle. The algorithm was able to produce a meaningful result for all of them. This observation can also be justified by looking at the algorithm itself where it does not yield a result if there are no cycles. Therefore, the algorithm was said to be experimentally correct. The ratio bound was found for different sized randomly built graphs with different sample sizes. When there are 10 vertices given with edges in the range 5-9, there is an obvious increasing trend in the ratio bound as the number of edges increases (Figure 1) for 50, 100 and 150 randomly generated graphs. As the number of edges increases, the graph becomes denser. The outcome of the ratio bound was expected to be found higher for relatively dense graphs which is confirmed in the experimental analysis for the ratio bound. The reason is, the heuristic algorithm finds the longest cycle in a greedy way for each vertex. Meaning that, after finding a cycle path started from a vertex, the algorithm doesn’t check a longer cycle from that vertex. Since sparse graphs reduce the number of possible cycles, the probability of finding the longest cycle in sparse graphs is higher. This implies that the ratio bound increases (exceeds 1 more and more) when the graph is relatively more sparse. This observation is less obvious when the vertex number is kept as 10 and edge number varied from 5 to 9. By keeping the edge number constant while increasing the vertex number makes the graph less dense. So, the expectation would be observing a lower ratio bound (i. e. closer to 1) as vertex number increases. However, with the same number of edges, graphs with 6 vertices gave a higher ratio bound compared to the graphs with 5 vertices. The possible reasons are, limitations of possible cycles with 5 and 6 edges. Also, the edge number is too far from the maximum number of edges for 10 vertices in the graph when the edge number is given between 5 to 9 which makes it harder to come up with the conclusion by checking only those edge numbers. It wasn’t possible to investigate for more edges because of the running time issue of brute force algorithm. Nevertheless, the overall trend shows a decreasing manner of the ratio bound in Figure 2 which satisfies the expectation. Therefore, it can be concluded that sparse graphs give ratio bound values closer to 1 which means the heuristic algorithm works better for them.

The testing is conducted using black-box testing. It is seen that the increase in the edge number for the graphs with the same vertex number reduced the accuracy of the result found by the heuristic algorithm. The issue while checking the ratio bound and conducting black box testing was the limitation of graph sizes. Since the brute force algorithm should be used for both of them to compare the optimal longest cycle with the result of the heuristic algorithm, it takes too long to come up with the result since the brute force finds the result in exponential time. That’s why, the maximum number of edges/vertices taken into account while generating the random graph was limited by 10.

REFERENCES

- Hüsnü Yenigün, Fall 2020, Lecture Slides, CS301-Algorithms-08-NP-Completeness.pdf
- https://en.wikipedia.org/wiki/Travelling_salesman_problem#:~:text=The%20travelling%20salesman%20problem%20
- An Annotated List of Selected NP-complete Problems, https://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html
- University of California San Diego, https://cseweb.ucsd.edu/classes/fa98/cse101/hw3ans/hw3ans.html
- CP-AlgorithmsPage Authors, https://cp-algorithms.com/graph/finding-cycle.html