CS301 Term Project

Report : Longest Circuit

Group Members:

Defne Çirci

Efe Öztaban

Kayra Bilgin

Zeynep Kılınç

Group Number: 9

Semester: Fall 2020

Instructor: Hüsnü Yenigün

Faculty of Engineering and Natural Sciences

Sabancı Üniversitesi

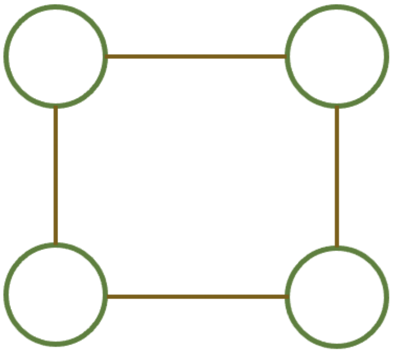
CS301 Term Project

1) Problem Description

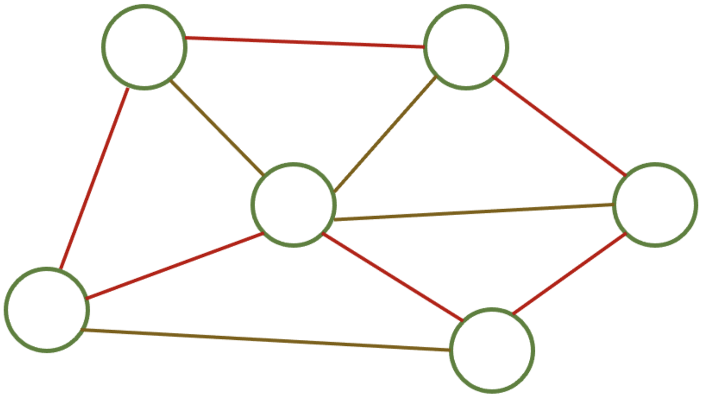
We will try to address the length of a longest simple cycle (LSC) in G where G=(V,E) is a n-node undirected graph. Simple cycle is defined as a path with non-identical vertices v0,v1.. vn where (vn,v0) ∊ E meaning that there is an edge from vn to v0. Among all the simple cycles we seek the one with the highest number of vertices. Hamiltonian Circuit Problem and Traveling Salesman Problem are special cases of this problem which can be used for mapping genomes, electronic circuit design, creating optimized delivery roads for firms and school bus destinations. [1]. There are some groups who have worked on this problem or similar ones. [2,3,4]

Basic Examples:

Simple Cycle



Longest Simple Cycle Consisting of 6 Vertices



Longest Simple Circuit is NP-Complete

In order to show that our problem is an NP-complete problem, we need to follow a theorem which consists of 3 steps:

1. Show that LSC is Nondeterministically Polynomial (NP)
2. Find another problem P which is known to be NP-Complete
3. Show that P can be transformed into LSC in polynomial time

A. Longest Simple Circuit is in NP

For a problem to be NP, it needs to be polynomially verifiable. We will show that a decision of whether a sequence of n vertices, n bigger than a specific number k, is a simple cycle can be made in polynomial time. Our verifier will first iterate through all the vertices in the guess and check if there exists an edge between all consecutive pairs (verifying path) and their uniqueness (simple path). Then it will check if there is an edge between the last and the first vertices (cycle). All can be done in polynomial time which implies that it is in NP.

B. An NP Complete Problem can be reduced to LSC in polynomial time.

Now, we will reduce the Hamiltonian Circuit Problem to LSC. Given a graph G with vertex number of k, example of a Hamiltonian Circuit, we will define an LSC with k number of vertices <G,k>.We can convert this to LCS by adding n-k disconnected vertices. Hamiltonian Cycle in G implies that there is a simple cycle with a length of |V|. If Hamiltonian Cycle cannot be found, then there is no cycle of length |V| or more. Since the reduction to the decision problem of whether there exists a simple cycle with a length k can be done in polynomial time, time it takes for counting and copying, it is a NP-hard problem. In part A, we proved that it is in NP. Thus, it is also NP-Complete. [5]

2) Algorithm Description

As it is proved in the previous part, the longest simple cycle problem is NP-Complete and it cannot be solved in polynomial time. There is an algorithm whose complexity is factorial which finds the exact answer. However, for graphs with a large number of vertices and edges, the running time of the exact algorithm will be significantly long. That exact algorithm is stated in the ratio bound part. That is why a heuristic algorithm which runs faster but does not find the longest simple cycle for every case will be more suitable for this problem. In the project two different heuristic algorithms are used which have the same asymptotic complexity but different ratio bounds and running times. In addition to these, there are three improvement algorithms that will increase the ratio bound of results from these heuristic algorithms without changing the overall time complexity.

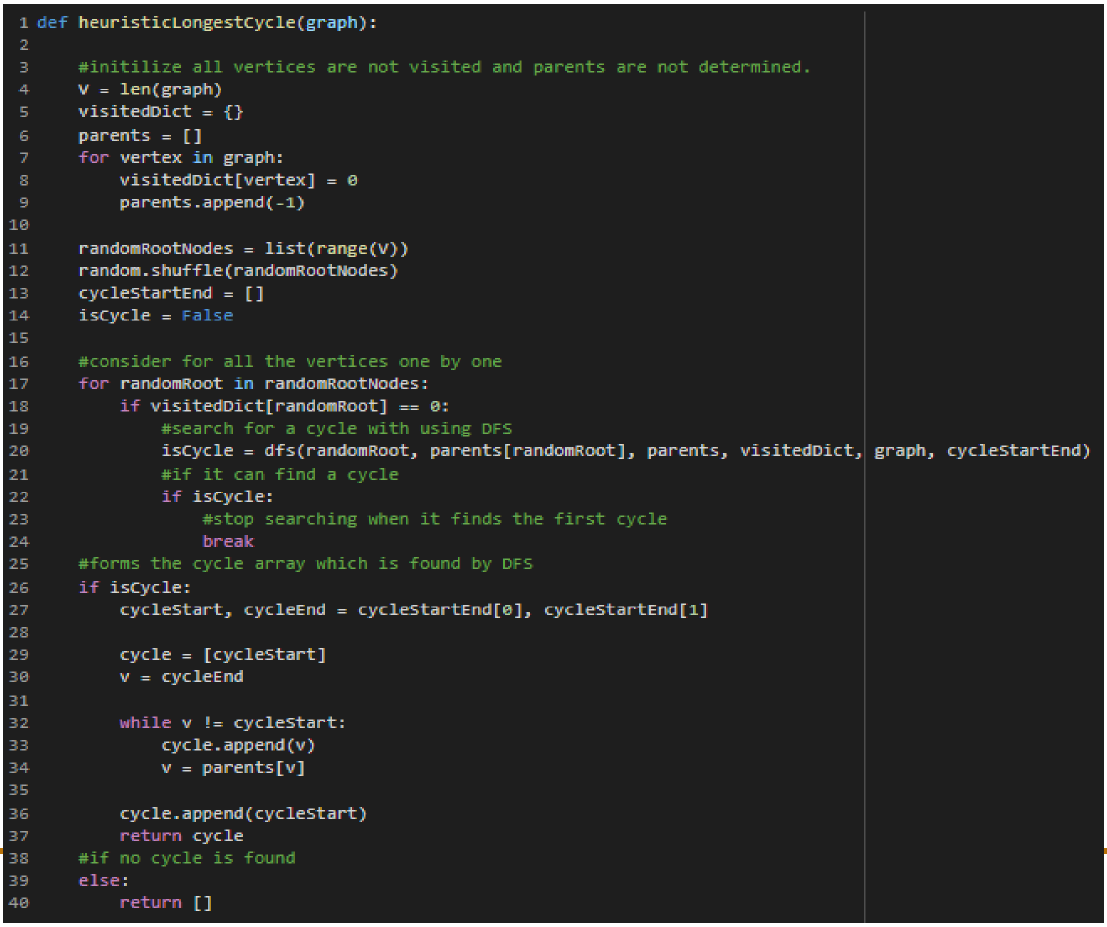
Heuristic Algorithms

A. Heuristic Longest Cycle

Heuristic Longest Cycle Steps:

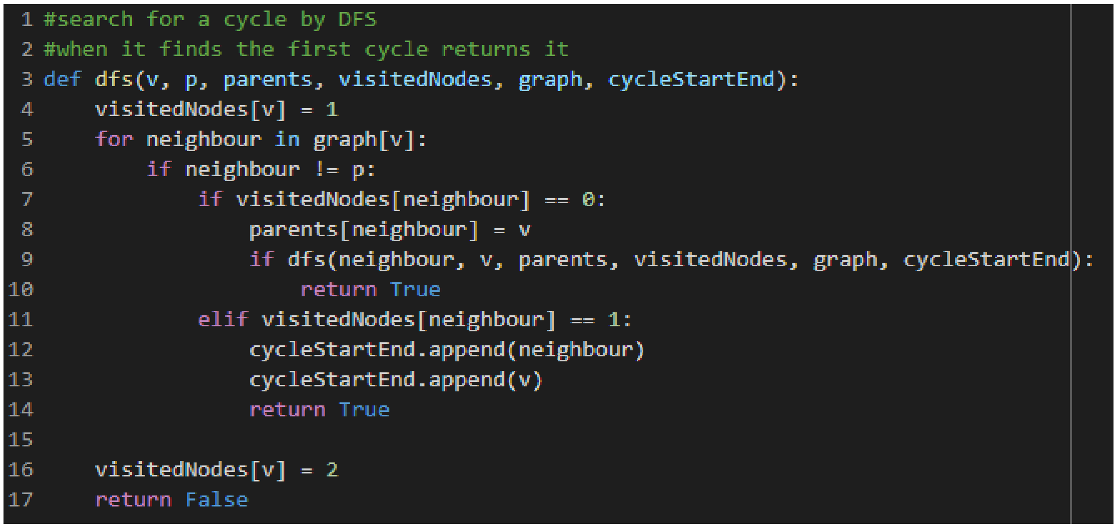
1. Creates and initializes data structures for parent and visited node information
2. Starting from the first vertex searches for a cycle which starts with the first vertex
3. Does the cycle search using DFS
4. If no cycle is found starts searching a cycle which starts with the next vertex
5. When the first cycle is found breaks the loop and returns that cycle

Here is the code implementation for Heuristic Longest Cycle:



```python
def heuristicLongestCycle(graph):

    #initilize all vertices are not visited and parents are not determined.
    V = len(graph)
    visitedDict = {}
    parents = []
    for vertex in graph:
        visitedDict[vertex] = 0
        parents.append(-1)

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

    #consider for all the vertices one by one
    for randomRoot in randomRootNodes:
        if visitedDict[randomRoot] == 0:
            #search for a cycle with using DFS
            isCycle = dfs(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
            #if it can find a cycle
            if isCycle:
                #stop searching when it finds the first cycle
                break
    #forms the cycle array which is found by DFS
    if isCycle:
        cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]

        cycle = [cycleStart]
        v = cycleEnd

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

        cycle.append(cycleStart)
        return cycle
    #if no cycle is found
    else:
        return []
```

Here is the code implementation of DFS used in Heuristic Longest Cycle:



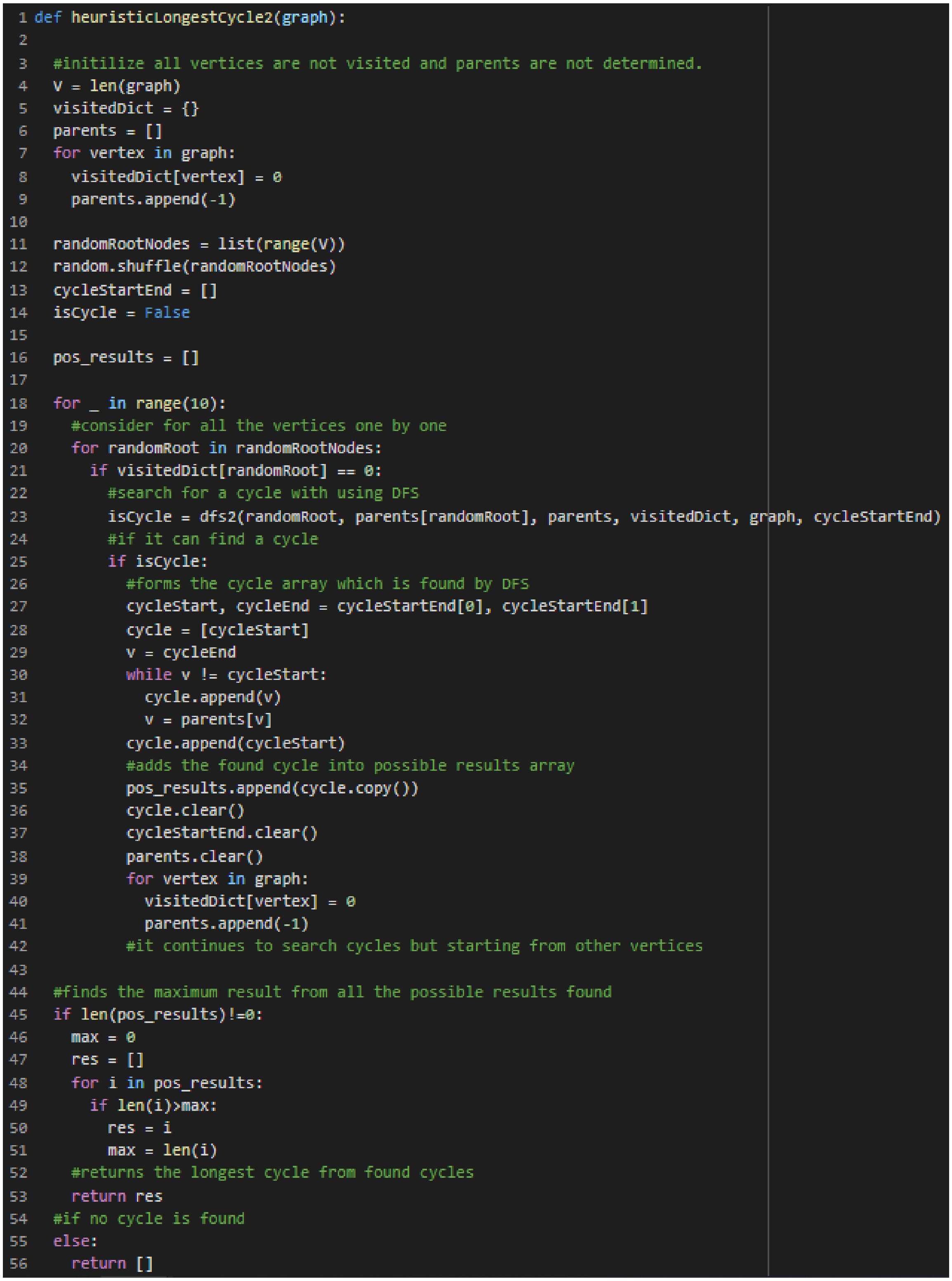
```
#search for a cycle by DFS
#when it finds the first cycle returns it
def dfs(v, p, parents, visitedNodes, graph, cycleStartEnd):
    visitedNodes[v] = 1
    for neighbour in graph[v]:
        if neighbour != p:
            if visitedNodes[neighbour] == 0:
                parents[neighbour] = v
                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
```

B. Heuristic Longest Cycle 2

Heuristic Longest Cycle 2 steps:

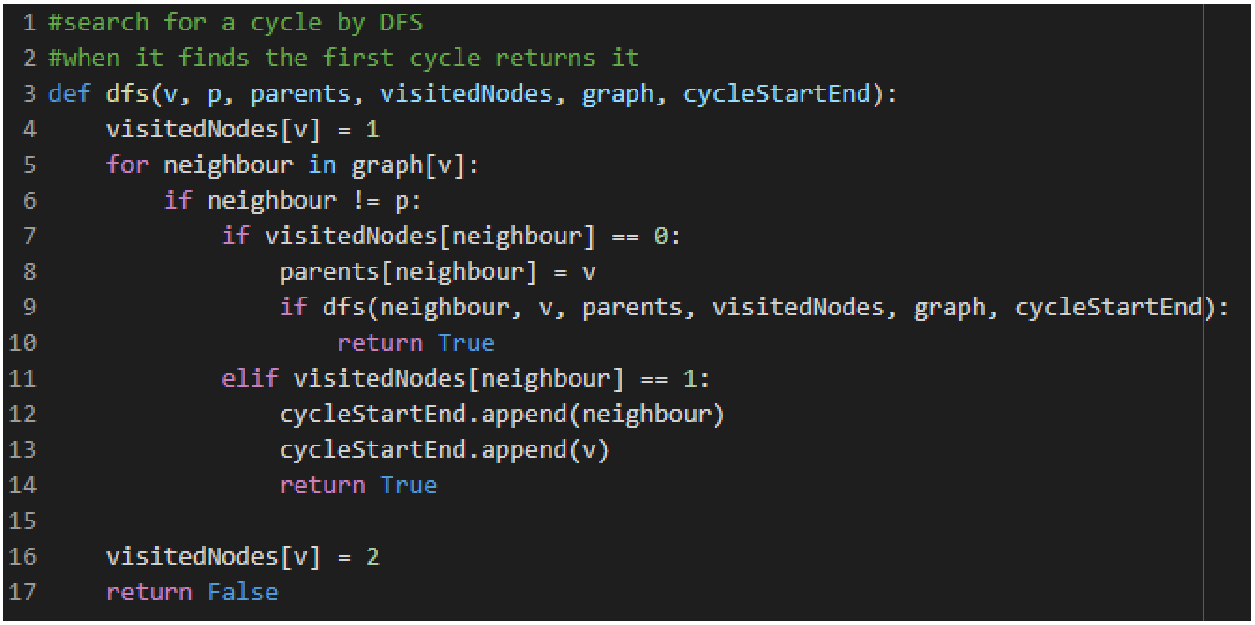
1. Creates and initializes data structures for parent and visited node information
2. Starting from the first vertex searches for a cycle which starts with the first vertex
3. Does the cycle search with using DFS
4. If no cycle is found starts searching a cycle which starts with the next vertex
5. When the first cycle is found pushes the found cycle into possible results array
6. Then continues the cycle search which starts with the next vertex
7. After the cycle search is completed for every vertex takes the longest cycle from the possible results array and returns it

8. Here is the code implementation for Heuristic Longest Cycle 2:



```
def heuristicLongestCycle2(graph):

  #initilize all vertices are not visited and parents are not determined.
  V = len(graph)
  visitedDict = {}
  parents = []
  for vertex in graph:
    visitedDict[vertex] = 0
    parents.append(-1)

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

  pos_results = []

  for _ in range(10):
    #consider for all the vertices one by one
    for randomRoot in randomRootNodes:
      if visitedDict[randomRoot] == 0:
        #search for a cycle with using DFS
        isCycle = dfs2(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
        #if it can find a cycle
        if isCycle:
          #forms the cycle array which is found by DFS
          cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]
          cycle = [cycleStart]
          v = cycleEnd
          while v != cycleStart:
            cycle.append(v)
            v = parents[v]
          cycle.append(cycleStart)
          #adds the found cycle into possible results array
          pos_results.append(cycle.copy())
          cycle.clear()
          cycleStartEnd.clear()
          parents.clear()
          for vertex in graph:
            visitedDict[vertex] = 0
            parents.append(-1)
          #it continues to search cycles but starting from other vertices

  #finds the maximum result from all the possible results found
  if len(pos_results)!=0:
    max = 0
    res = []
    for i in pos_results:
      if len(i)>max:
        res = i
        max = len(i)
    #returns the longest cycle from found cycles
    return res
  #if no cycle is found
  else:
    return []
```

Here is the code implementation of DFS used in Heuristic Longest Cycle 2:



```python
#search for a cycle by DFS
#when it finds the first cycle returns it
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
```

Comparison of the Algorithms

The asymptotic complexity of two heuristic algorithms are the same and shown in the next part. However, their running time in practice is different. The first heuristic algorithm runs faster than the second algorithm, but finds less optimal solutions. Differently, the second heuristic algorithm runs slower than the first algorithm, but finds more optimal solutions. Therefore, the ratio bound of the second algorithm is higher than the first algorithm.

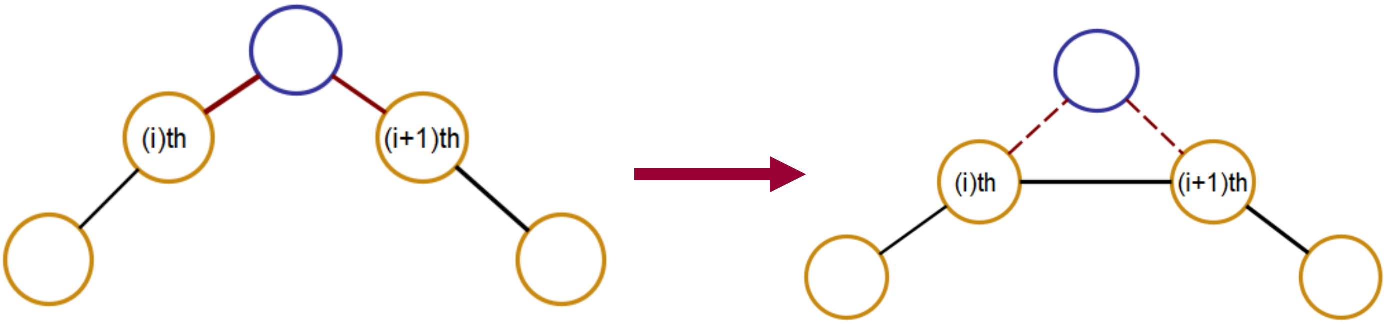
Improvement Algorithms

These algorithms take the graph and the result from the heuristic algorithms as inputs. Then they try to improve the result. In other words, these algorithms try to add new vertices to the previous result and increase the length of the found cycle.

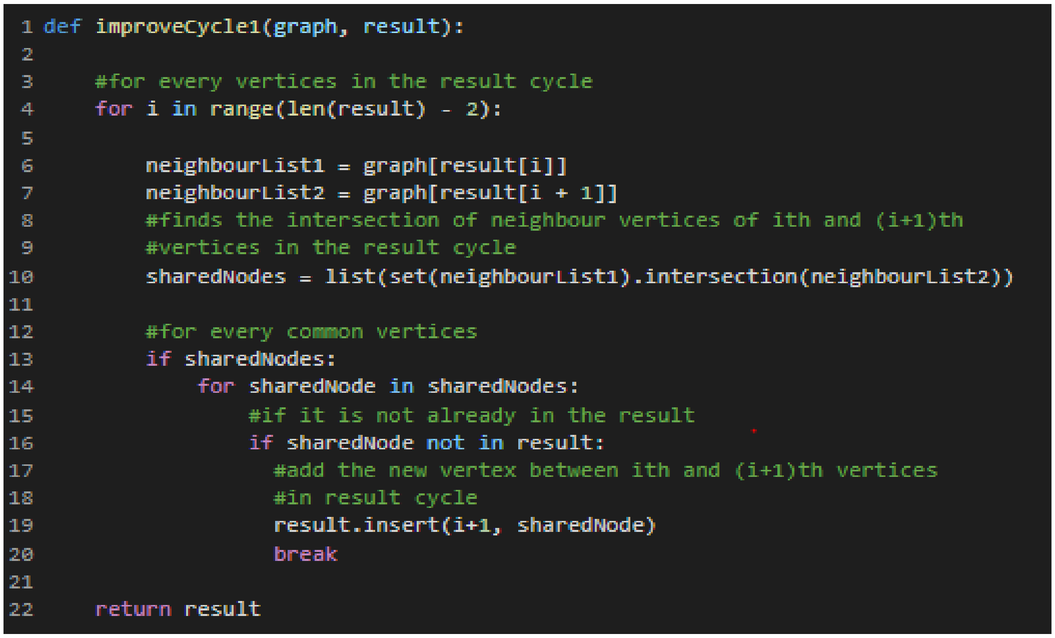
A. Improved Cycle 1

Improved Cycle 1 steps:

1. Takes the result and graph from the heuristic algorithm as inputs
2. Runs for all every vertex in the result
3. Finds the intersections of the nodes which are reachable from (i)th and (i+1)th vertices
4. Adds the first of the common reachable vertex between (i)th and (i+1)th vertices
5. Continues with the next vertex



Here is the code implementation of Improved Cycle 1:

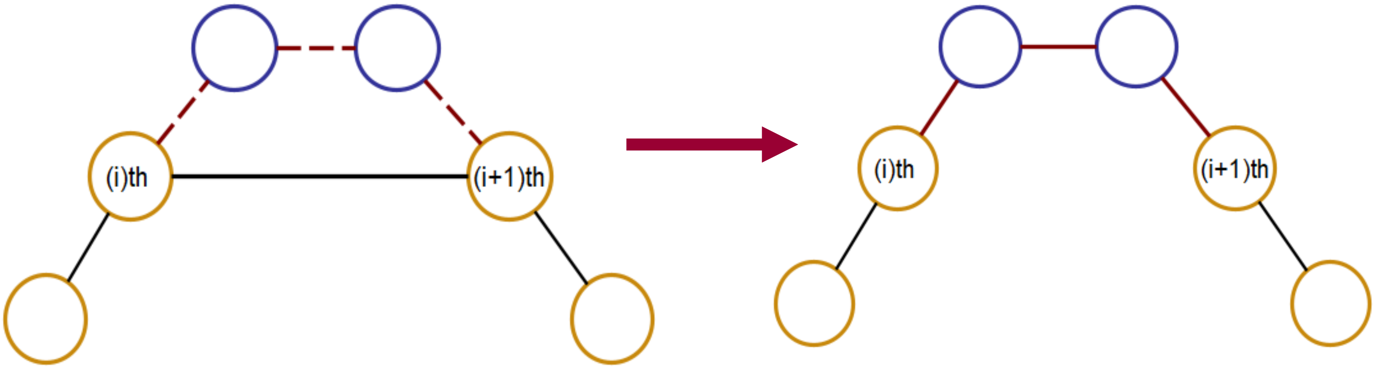


```python
def improveCycle1(graph, result):

    #for every vertices in the result cycle
    for i in range(len(result) - 2):

        neighbourList1 = graph[result[i]]
        neighbourList2 = graph[result[i + 1]]
        #finds the intersection of neighbour vertices of ith and (i+1)th
        #vertices in the result cycle
        sharedNodes = list(set(neighbourList1).intersection(neighbourList2))

        #for every common vertices
        if sharedNodes:
            for sharedNode in sharedNodes:
                #if it is not already in the result
                if sharedNode not in result:
                  #add the new vertex between ith and (i+1)th vertices
                  #in result cycle
                  result.insert(i+1, sharedNode)
                  break

    return result
```

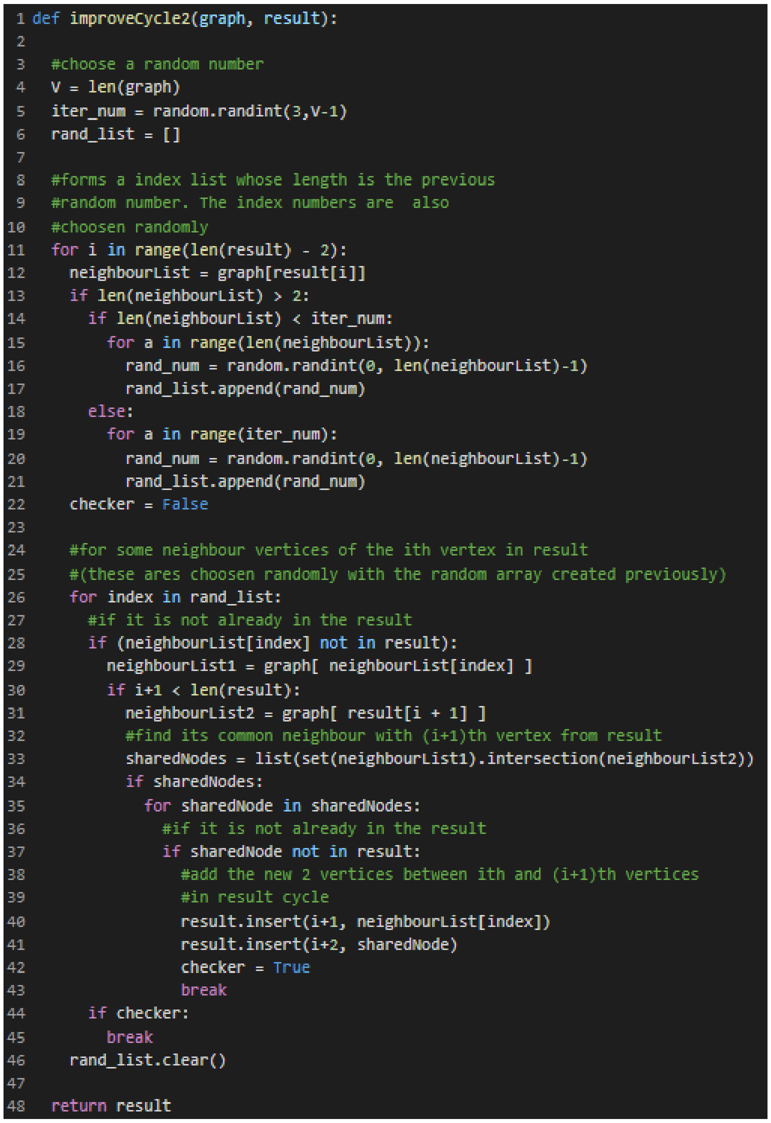
B. Improved Cycle 2

Improved Cycle 2 steps:

1. Takes the result and graph from the heuristic algorithm as inputs
2. Runs for all every vertex in the result
3. Randomly selects some (number is also randomly chosen) vertices which is connected to the (i)th vertex
4. Finds the intersections of the vertices which are reachable from the selected vertices and (i+1)th vertex
5. If it is found, adds the new pair of vertices between (i)th and (i+1)th vertices
6. Continues with the next vertex until the vertex before the last vertex



Here is the code implementation of Improved Cycle 2:

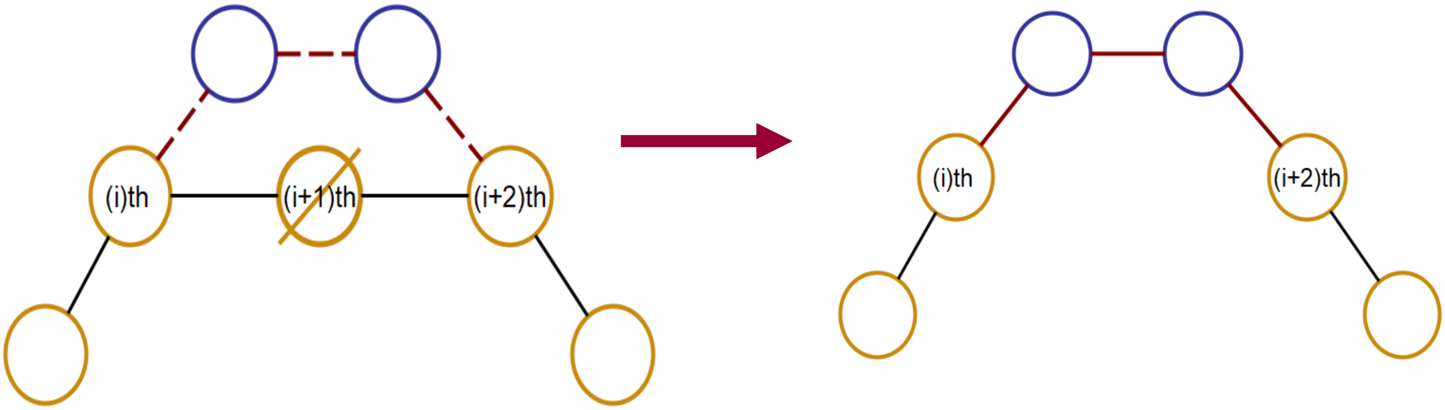


```python
def improveCycle2(graph, result):

  #choose a random number
  V = len(graph)
  iter_num = random.randint(3,V-1)
  rand_list = []

  #forms a index list whose length is the previous
  #random number. The index numbers are  also
  #choosen randomly
  for i in range(len(result) - 2):
    neighbourList = graph[result[i]]
    if len(neighbourList) > 2:
      if len(neighbourList) < iter_num:
        for a in range(len(neighbourList)):
          rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
      else:
        for a in range(iter_num):
          rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
    checker = False

    #for some neighbour vertices of the ith vertex in result
    #(these ares choosen randomly with the random array created previously)
    for index in rand_list:
      #if it is not already in the result
      if (neighbourList[index] not in result):
        neighbourList1 = graph[ neighbourList[index] ]
        if i+1 < len(result):
          neighbourList2 = graph[ result[i + 1] ]
          #find its common neighbour with (i+1)th vertex from result
          sharedNodes = list(set(neighbourList1).intersection(neighbourList2))
          if sharedNodes:
            for sharedNode in sharedNodes:
              #if it is not already in the result
              if sharedNode not in result:
                #add the new 2 vertices between ith and (i+1)th vertices
                #in result cycle
                result.insert(i+1, neighbourList[index])
                result.insert(i+2, sharedNode)
                checker = True
                break
      if checker:
        break
    rand_list.clear()

  return result
```

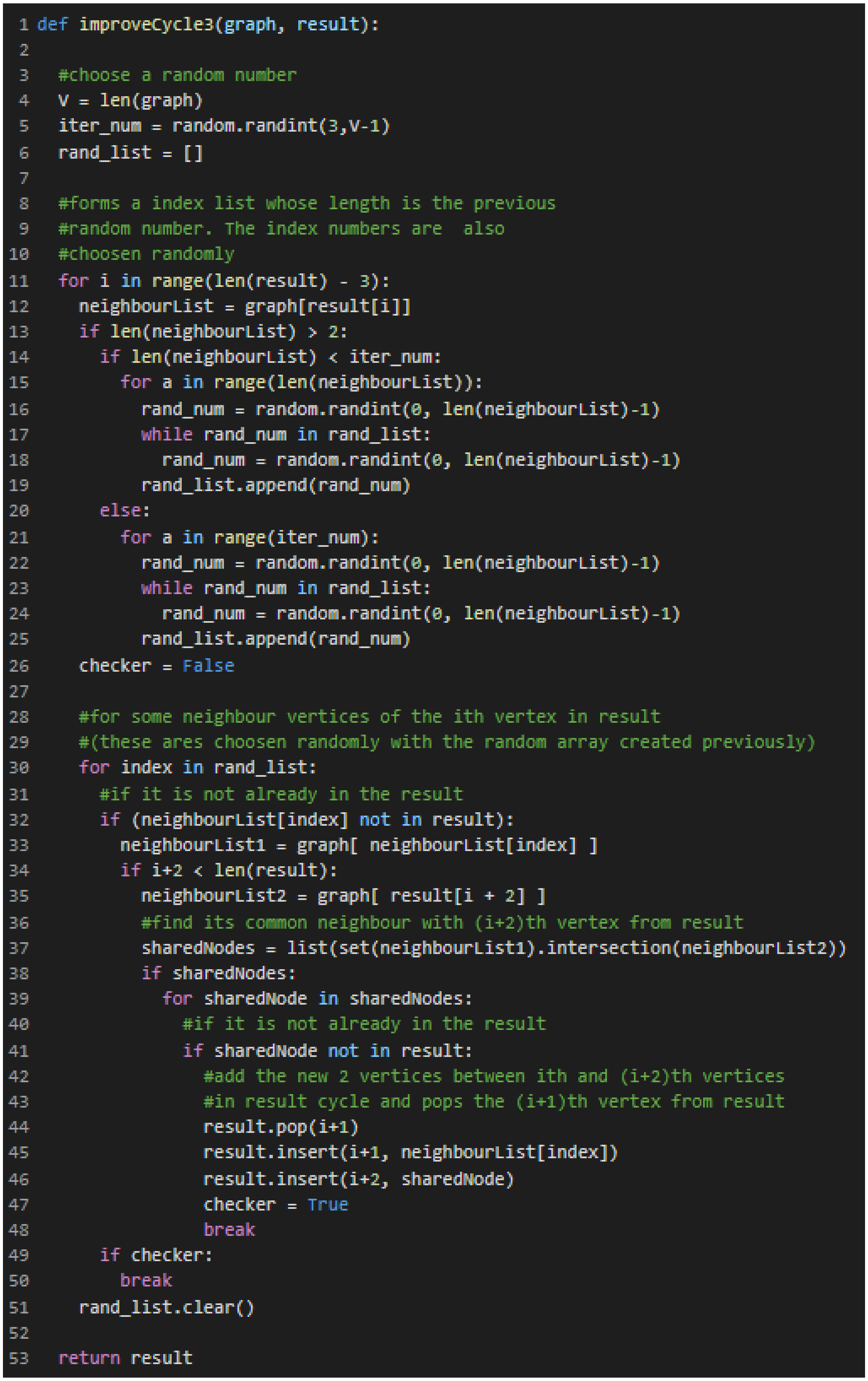
C. Improved Cycle 3

Improved Cycle 3 steps:

1. Takes the result and graph from the heuristic algorithm as inputs
2. Runs for all every vertex in the result
3. Randomly selects some (number is also randomly chosen) vertices which is connected to the (i)th vertex
4. Finds the intersections of the vertices which are reachable from the selected vertices and (i+2)th vertex
5. If it is found, pops the (i+1)th vertex
6. Then adds the new pair of vertices between (i)th and (i+1)th vertices
7. Continues with the next vertex until the vertex before the last vertex



Here is the code implementation of Improved Cycle 3:

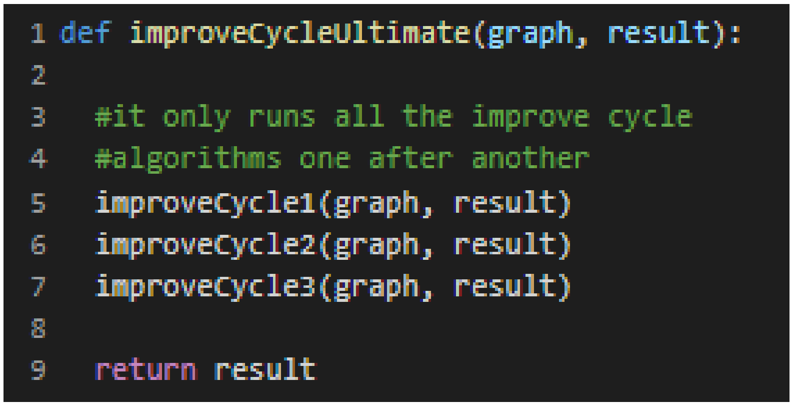


```
def improveCycle3(graph, result):

  #choose a random number
  V = len(graph)
  iter_num = random.randint(3,V-1)
  rand_list = []

  #forms a index list whose length is the previous
  #random number. The index numbers are  also
  #choosen randomly
  for i in range(len(result) - 3):
    neighbourList = graph[result[i]]
    if len(neighbourList) > 2:
      if len(neighbourList) < iter_num:
        for a in range(len(neighbourList)):
          rand_num = random.randint(0, len(neighbourList)-1)
          while rand_num in rand_list:
            rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
      else:
        for a in range(iter_num):
          rand_num = random.randint(0, len(neighbourList)-1)
          while rand_num in rand_list:
            rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
    checker = False

    #for some neighbour vertices of the ith vertex in result
    #(these ares choosen randomly with the random array created previously)
    for index in rand_list:
      #if it is not already in the result
      if (neighbourList[index] not in result):
        neighbourList1 = graph[ neighbourList[index] ]
        if i+2 < len(result):
          neighbourList2 = graph[ result[i + 2] ]
          #find its common neighbour with (i+2)th vertex from result
          sharedNodes = list(set(neighbourList1).intersection(neighbourList2))
          if sharedNodes:
            for sharedNode in sharedNodes:
              #if it is not already in the result
              if sharedNode not in result:
                #add the new 2 vertices between ith and (i+2)th vertices
                #in result cycle and pops the (i+1)th vertex from result
                result.pop(i+1)
                result.insert(i+1, neighbourList[index])
                result.insert(i+2, sharedNode)
                checker = True
                break
      if checker:
        break
    rand_list.clear()

  return result
```

D. Ultimate Improved Cycle

This is not a different improved cycle algorithm. It runs all the improved cycle algorithms (1,2,3) consecutively.

Here is the code implementation of Improved Cycle Ultimate:



```
def improveCycleUltimate(graph, result):

  #it only runs all the improve cycle
  #algorithms one after another
  improveCycle1(graph, result)
  improveCycle2(graph, result)
  improveCycle3(graph, result)

  return result
```

3) Algorithm Analysis

Correctness Analysis

The correctness analysis of both heuristic algorithms is made using loop invariant method.

A. Heuristic Longest Cycle

Initialization Phase: Firstly, it is necessary to show that loop invariant holds before the first iteration of the vertex loop. The cycle is defined as an empty array at the initialization part which is at the beginning of the algorithm. For every graph, an empty cycle will be the correct result because an empty cycle exists in every graph.

So the result before the first iteration of the loop is correct because it is empty. Therefore, it holds.

Maintenance: Secondly, it is necessary to show that loop invariant holds after the first iteration of the vertex loop. In the first iteration of the loop, the algorithm searches for a cycle which starts and ends with the first vertex of the input graph. In the searching part every path starting from the first vertex is visited with DFS. After the visits to every vertex, these vertices are marked as visited. So, it is impossible to visit a node more than once. And the search ends only if the last visited vertex is equal to the first visited vertex or if all the paths are visited and no cycle is found. After the first iteration, the result can be updated as a cycle or can stay as an empty cycle. If it stays the same after no cycle is found, because an empty cycle is always a correct result as explained above, it still holds. If the result is updated with the found cycle, it should be simple (every vertex can be visited at most one time) and it should be a cycle (first and last vertex should be the same). Because every visited vertex is marked and it is impossible to visit a vertex more than once in the algorithm, the result is definitely simple. In addition, the search is stopped only when every path is visited and no cycle found or when the first and last vertex in the path is the same. So, the first and the last vertex must be the same according to the algorithm. If the result is updated, the result will also definitely be a cycle. Therefore, it also holds after the first iteration.

Termination: Lastly, it is necessary to show that loop invariant holds after the last iteration of the vertex loop. The loop is terminated only for two different cases. The first case is after all the paths starting from every vertex is searched and no cycle found. In that case the final result is an empty cycle and it is a correct result for every graph. The second case for termination of the loop is finding a cycle. If a cycle is found, the loop is broken and terminated in the algorithm. In the previous part it is proven that if a cycle is found, it is definitely a simple cycle. So, the result for the second case is also correct. Therefore, after the termination of the loop and the whole algorithm, the result is correct and it holds.

B. Heuristic Longest Cycle 2

Correctness analysis of the second heuristic is mostly the same with the previous part. However, for the differences in the algorithm, the related parts are updated.

Initialization Phase: Firstly, it is necessary to show that loop invariant holds before the first iteration of the vertex loop. The possible results array is defined with only an empty array at the initialization part which is at the beginning of the algorithm. For every graph, an empty cycle will be the correct result because an empty cycle exists in every graph. So, the result before the first iteration of the loop, is correct because it is empty. Therefore, it holds.

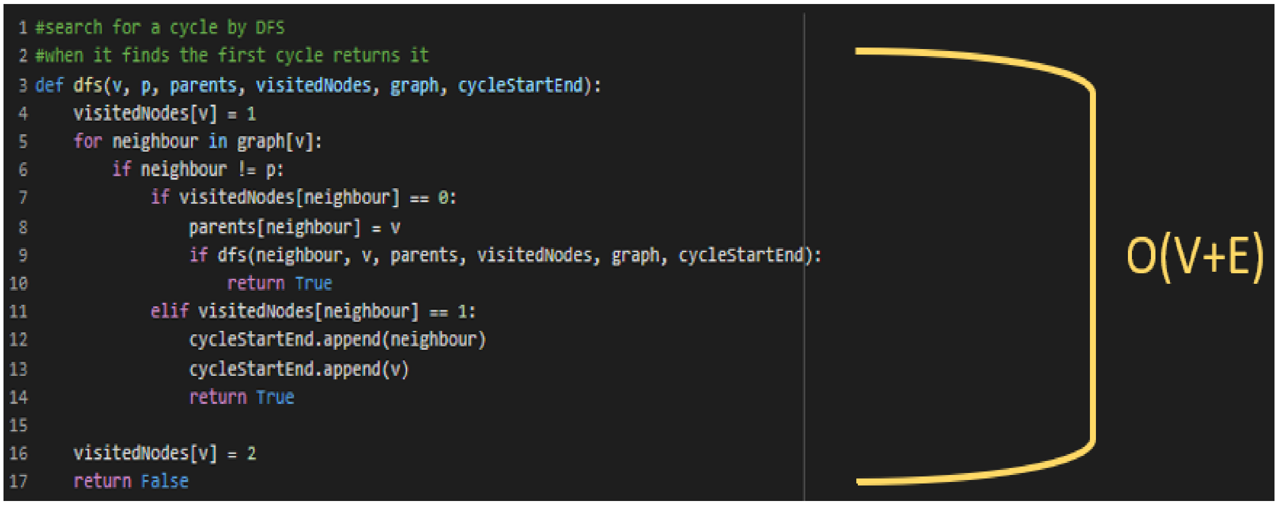
Maintenance: Secondly, it is necessary to show that loop invariant holds after the first iteration of the vertex loop. In the first iteration of the loop, the algorithm searches for a cycle which starts and ends with the first vertex of the input graph. In the searching part every path starting from the first vertex is visited with DFS. After the visits to every vertex, these vertices are marked as visited. So, it is impossible to visit a node more than once. And the search ends only if the last visited vertex is equal to the first visited vertex or if all the paths are visited and no cycle is found. After the first iteration, a found cycle can be added to the possible results array or the possible results array can stay with only an empty cycle. If it stays the same after no cycle is found, because an empty cycle is always a correct result as explained above, it still holds. If the possible results array is updated with the found cycle, it should be simple (every vertex can be visited at most one time) and it should be a cycle (first and last vertex should be the same). Because every visited vertex is marked and it is impossible to visit a vertex more than once in the algorithm, the found cycle is definitely simple. In addition, the search is stopped only when every path is visited and no cycle found or when the first and last vertex in the path is the same. So, the first and the last vertex must be the same according to the algorithm. If the possible results array is updated, the added cycle will also definitely cycle. Therefore, it also holds after the first iteration.

Termination: Lastly, it is necessary to show that loop invariant holds after the last iteration of the vertex loop. The loop is terminated only after all the iterations are done for every vertex. For every iteration, there can be two cases. The first case is after all the paths starting from every vertex is searched and no cycle found. In that case, search will continue for the next iteration with the next vertex. For this situation, visited vertex information is protected. For the next iteration, if it continues to find no cycle until the end, no cycle will be added into the possible results array. So, the final result is an empty cycle and it is the correct result for every graph. If a cycle is found, for next iterations for the first case, it will be considered as the second case. The second case for every iteration of the loop is finding a cycle. If a cycle is found, that iteration ends and a found cycle is added to the possible results array. In the previous part it is proven that if a cycle is found, it is definitely a simple cycle. After all the iterations are done and the loop is terminated, also different cases can happen. Firstly, for all the iterations the first case can occur. In this instance, there will be only an empty cycle the possible results array. Because an empty cycle is a correct answer, this case holds. Secondly, a cycle can be found once or more than once and added to the possible results array. In that case, the longest cycle will be chosen as the result. Because it is proven that all the found cycles are definitely simple cycles, the second case also holds. Therefore, after the termination of the loop and the whole algorithm, the result is correct and it holds.

Complexity Analysis

Worst case asymptotic complexity of the algorithms as follows:

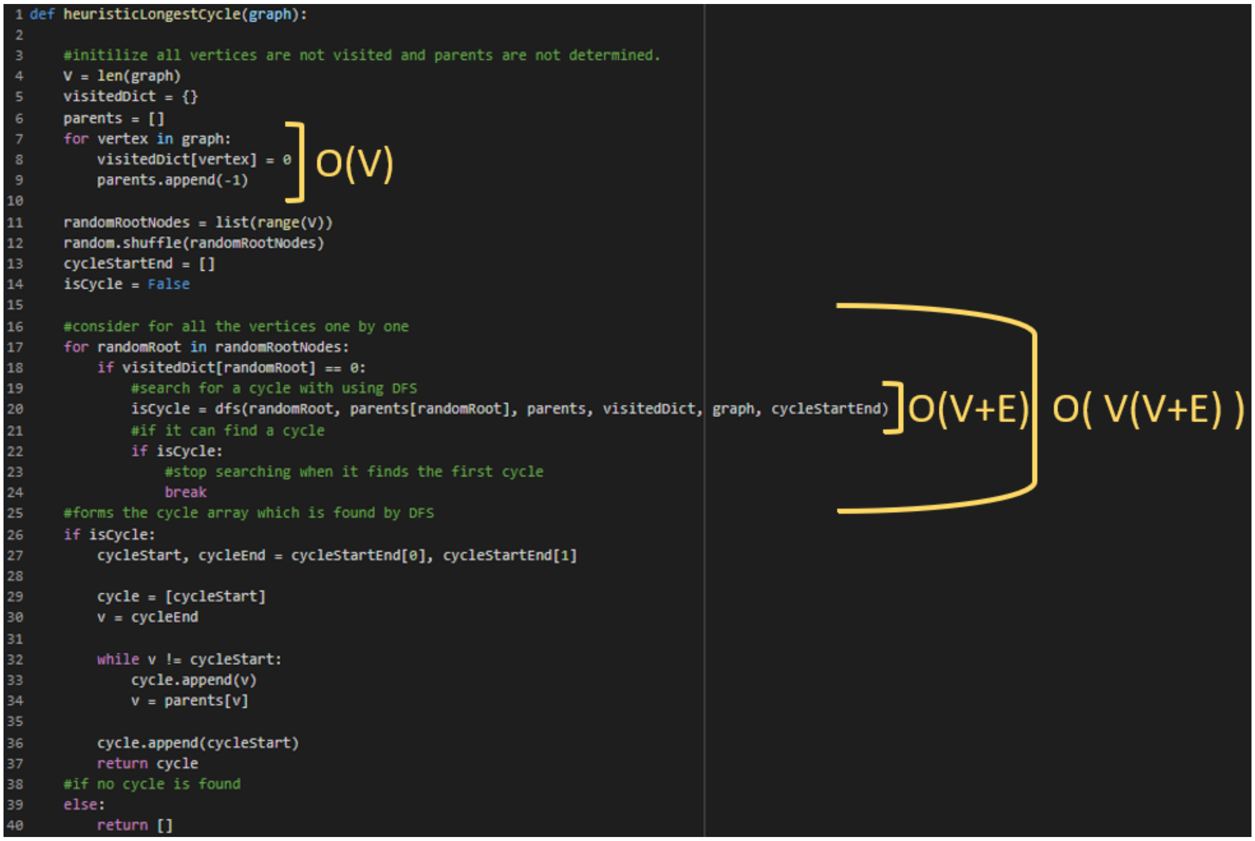
Complexity of DFS which is used in both heuristic Algorithms:



```
#search for a cycle by DFS
#when it finds the first cycle returns it
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
```

Complexity of DFS is O(V+E).

A. Heuristic Longest Cycle

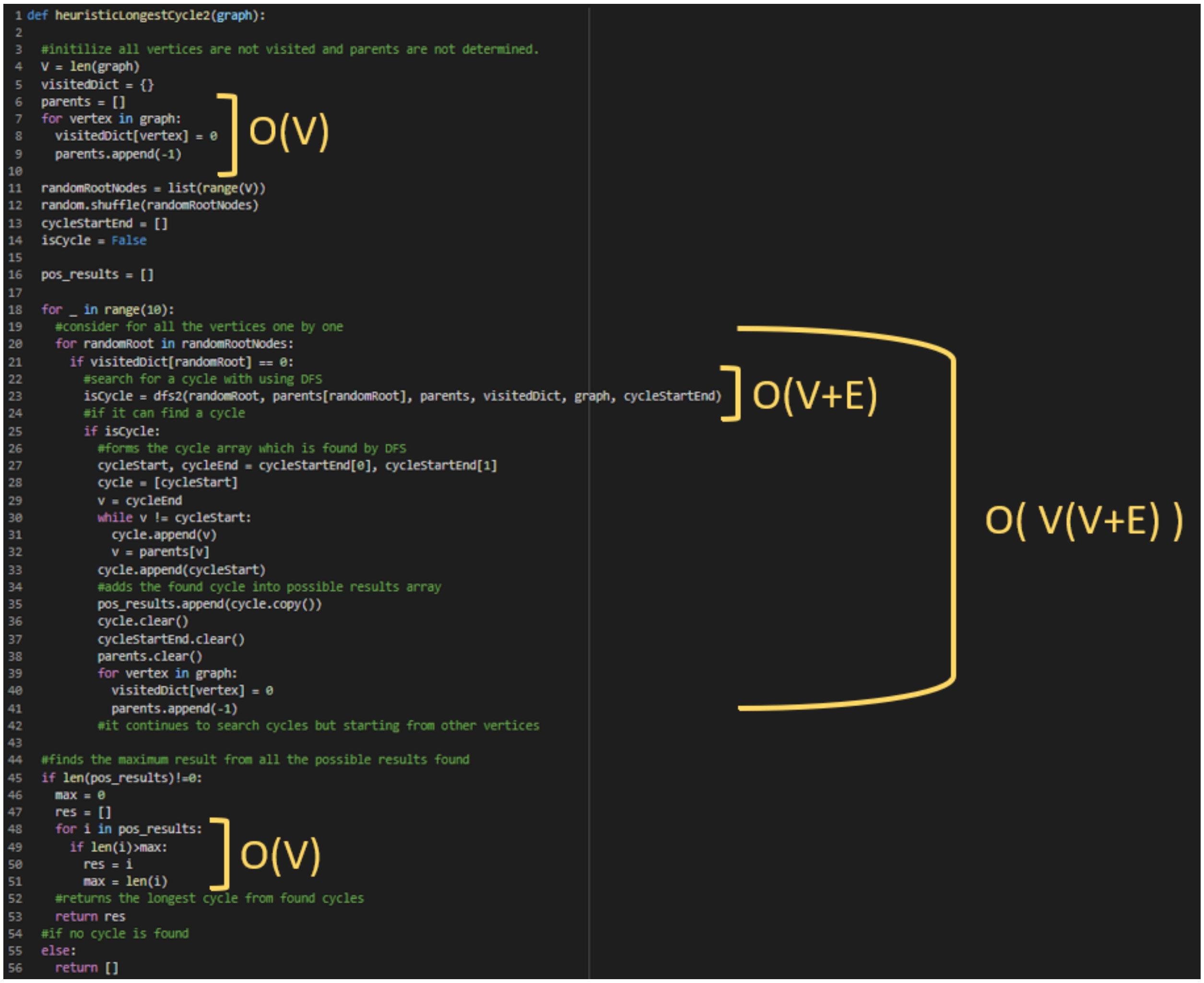


```python
def heuristicLongestCycle(graph):

    #initilize all vertices are not visited and parents are not determined.
    V = len(graph)
    visitedDict = {}
    parents = []
    for vertex in graph:
        visitedDict[vertex] = 0
        parents.append(-1)

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

    #consider for all the vertices one by one
    for randomRoot in randomRootNodes:
        if visitedDict[randomRoot] == 0:
            #search for a cycle with using DFS
            isCycle = dfs(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
            #if it can find a cycle
            if isCycle:
                #stop searching when it finds the first cycle
                break
    #forms the cycle array which is found by DFS
    if isCycle:
        cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]

        cycle = [cycleStart]
        v = cycleEnd

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

        cycle.append(cycleStart)
        return cycle
    #if no cycle is found
    else:
        return []
```

O(V) + O(V(V+E)) = O(V(V+E)).

Complexity of Heuristic Longest Cycle is O(V(V+E)).

B. Heuristic Longest Cycle 2

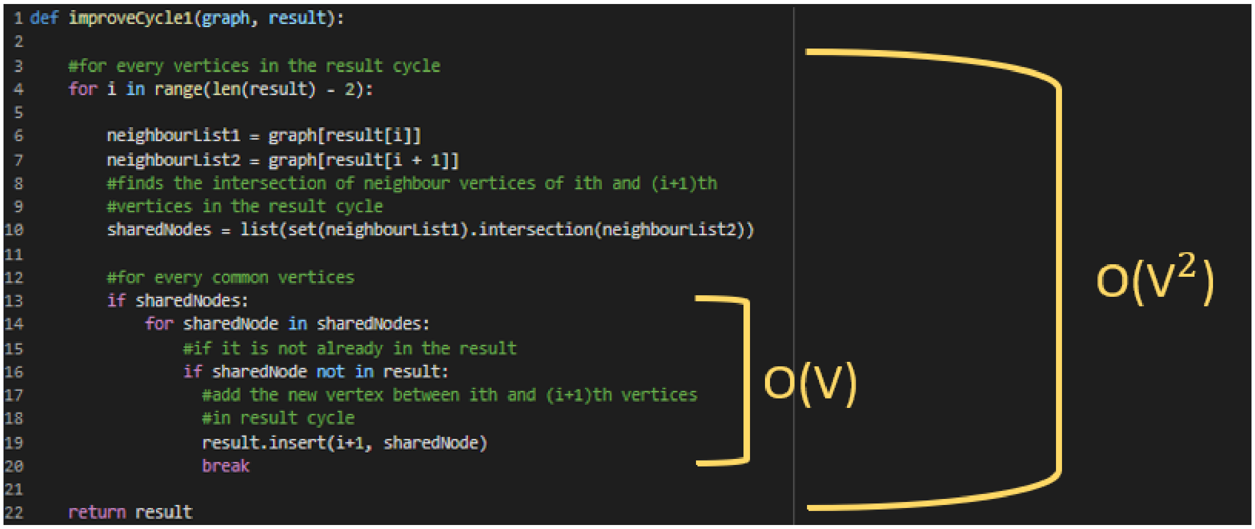


```python
def heuristicLongestCycle2(graph):

  #initilize all vertices are not visited and parents are not determined.
  V = len(graph)
  visitedDict = {}
  parents = []
  for vertex in graph:
    visitedDict[vertex] = 0
    parents.append(-1)

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

  pos_results = []

  for _ in range(10):
    #consider for all the vertices one by one
    for randomRoot in randomRootNodes:
      if visitedDict[randomRoot] == 0:
        #search for a cycle with using DFS
        isCycle = dfs2(randomRoot, parents[randomRoot], parents, visitedDict, graph, cycleStartEnd)
        #if it can find a cycle
        if isCycle:
          #forms the cycle array which is found by DFS
          cycleStart, cycleEnd = cycleStartEnd[0], cycleStartEnd[1]
          cycle = [cycleStart]
          v = cycleEnd
          while v != cycleStart:
            cycle.append(v)
            v = parents[v]
          cycle.append(cycleStart)
          #adds the found cycle into possible results array
          pos_results.append(cycle.copy())
          cycle.clear()
          cycleStartEnd.clear()
          parents.clear()
          for vertex in graph:
            visitedDict[vertex] = 0
            parents.append(-1)
          #it continues to search cycles but starting from other vertices

  #finds the maximum result from all the possible results found
  if len(pos_results)!=0:
    max = 0
    res = []
    for i in pos_results:
      if len(i)>max:
        res = i
        max = len(i)
    #returns the longest cycle from found cycles
    return res
  #if no cycle is found
  else:
    return []
```

O(V) + O(V(V+E)) O(V) = O(V(V+E)).

Complexity of Heuristic Longest Cycle 2 is O(V(V+E)).

C. Improved Cycle 1

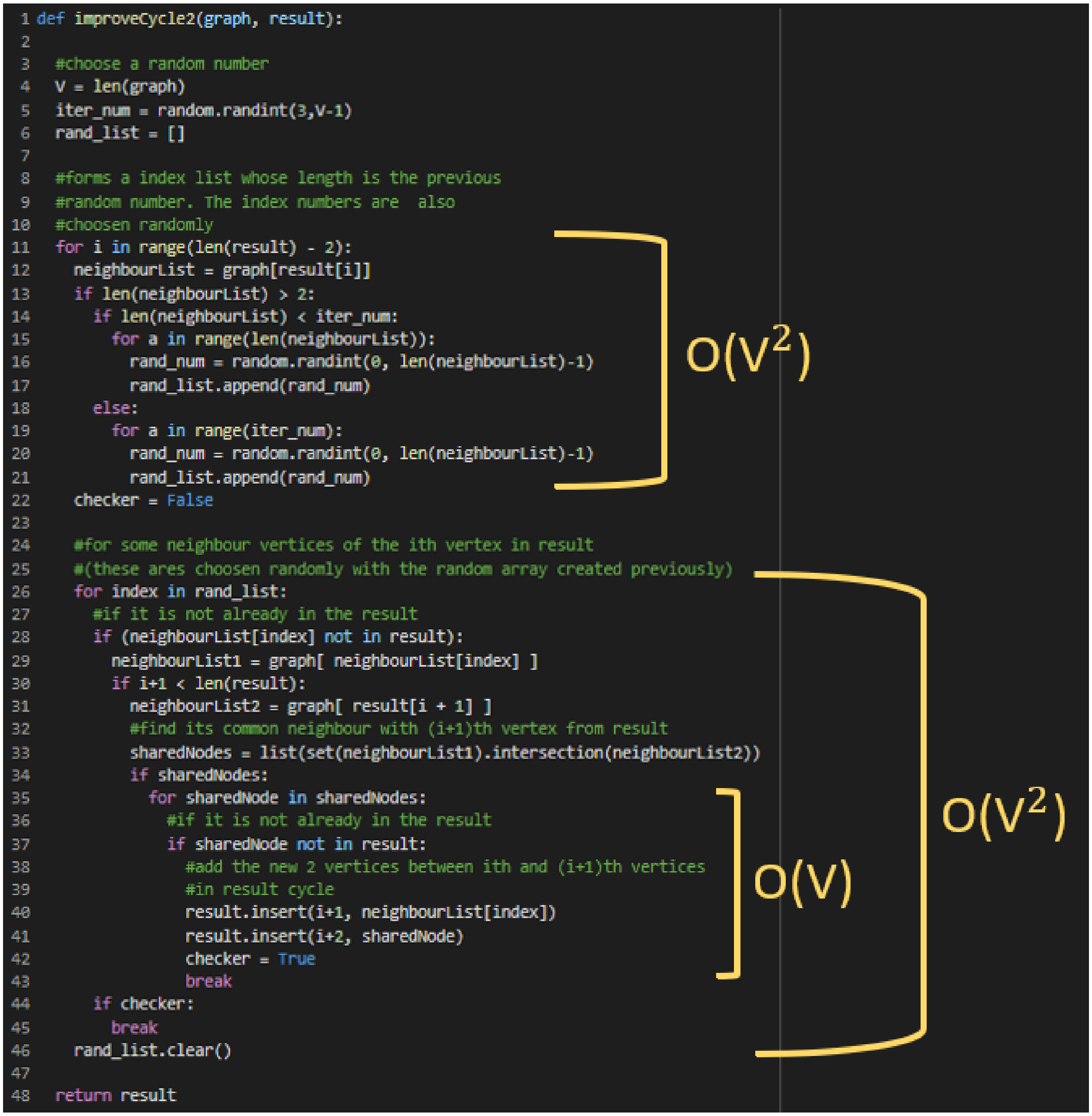


```
def improveCycle1(graph, result):

    #for every vertices in the result cycle
    for i in range(len(result) - 2):

        neighbourList1 = graph[result[i]]
        neighbourList2 = graph[result[i + 1]]
        #finds the intersection of neighbour vertices of ith and (i+1)th
        #vertices in the result cycle
        sharedNodes = list(set(neighbourList1).intersection(neighbourList2))

        #for every common vertices
        if sharedNodes:
            for sharedNode in sharedNodes:
                #if it is not already in the result
                if sharedNode not in result:
                  #add the new vertex between ith and (i+1)th vertices
                  #in result cycle
                  result.insert(i+1, sharedNode)
                  break

    return result
```

$$O(V) * O(V) = O(V^2)$$

Complexity of Improved Cycle 1 is $O(V^2)$.

D. Improved Cycle 2

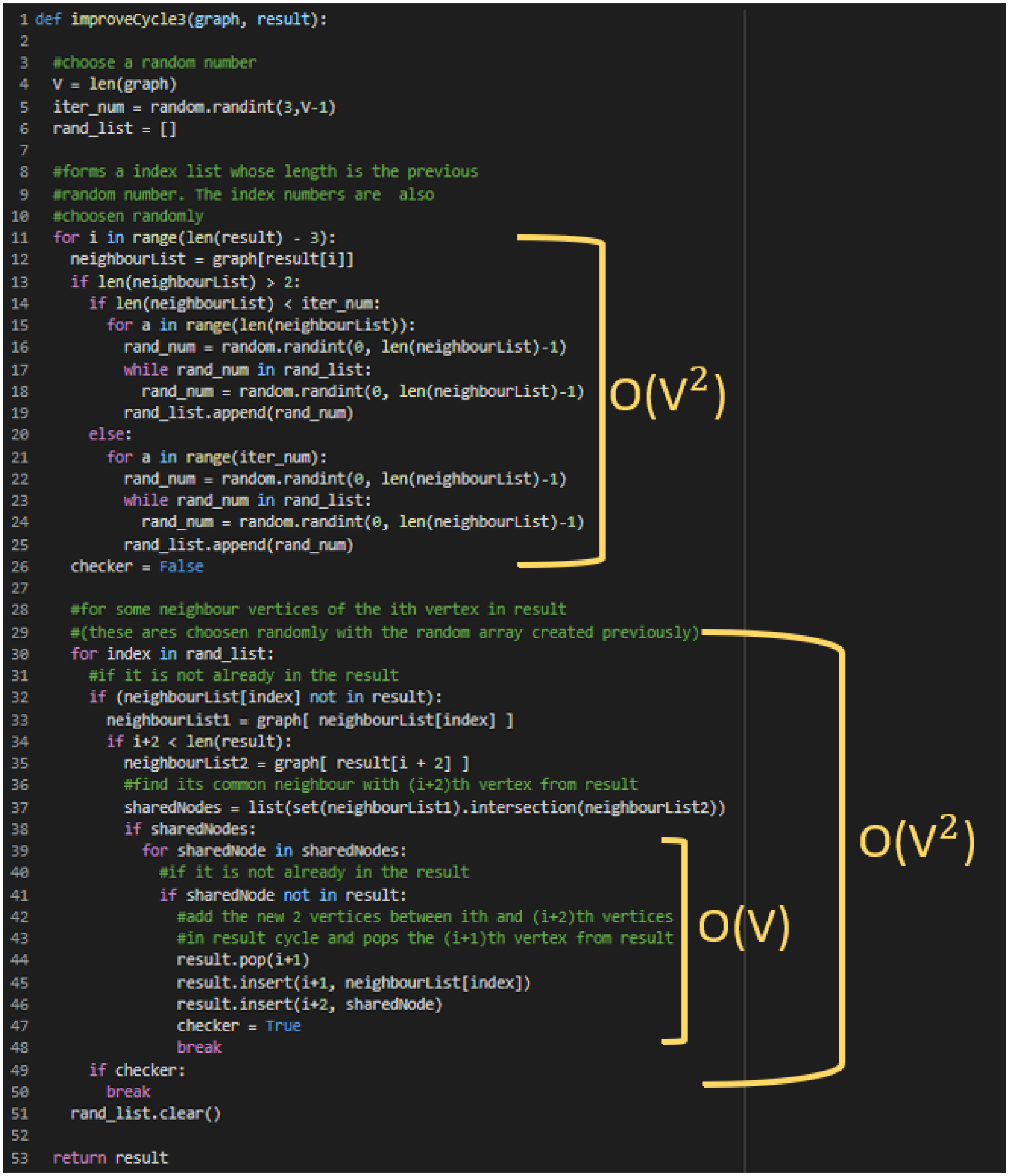


```
def improveCycle2(graph, result):

  #choose a random number
  V = len(graph)
  iter_num = random.randint(3,V-1)
  rand_list = []

  #forms a index list whose length is the previous
  #random number. The index numbers are  also
  #choosen randomly
  for i in range(len(result) - 2):
    neighbourList = graph[result[i]]
    if len(neighbourList) > 2:
      if len(neighbourList) < iter_num:
        for a in range(len(neighbourList)):
          rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
      else:
        for a in range(iter_num):
          rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
    checker = False

    #for some neighbour vertices of the ith vertex in result
    #(these ares choosen randomly with the random array created previously)
    for index in rand_list:
      #if it is not already in the result
      if (neighbourList[index] not in result):
        neighbourList1 = graph[ neighbourList[index] ]
        if i+1 < len(result):
          neighbourList2 = graph[ result[i + 1] ]
          #find its common neighbour with (i+1)th vertex from result
          sharedNodes = list(set(neighbourList1).intersection(neighbourList2))
          if sharedNodes:
            for sharedNode in sharedNodes:
              #if it is not already in the result
              if sharedNode not in result:
                #add the new 2 vertices between ith and (i+1)th vertices
                #in result cycle
                result.insert(i+1, neighbourList[index])
                result.insert(i+2, sharedNode)
                checker = True
                break
      if checker:
        break
    rand_list.clear()

  return result
```

$$O(V^2) + (O(V) * O(V)) = O(V^2)$$

Complexity of Improved Cycle 2 is $O(V^2)$.

E. Improved Cycle 3

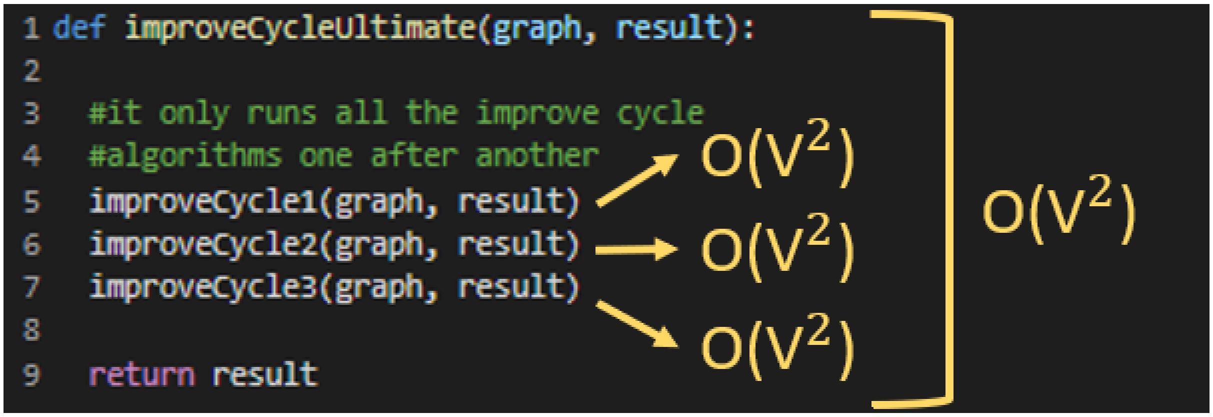


```
def improveCycle3(graph, result):

  #choose a random number
  V = len(graph)
  iter_num = random.randint(3,V-1)
  rand_list = []

  #forms a index list whose length is the previous
  #random number. The index numbers are  also
  #choosen randomly
  for i in range(len(result) - 3):
    neighbourList = graph[result[i]]
    if len(neighbourList) > 2:
      if len(neighbourList) < iter_num:
        for a in range(len(neighbourList)):
          rand_num = random.randint(0, len(neighbourList)-1)
          while rand_num in rand_list:
            rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
      else:
        for a in range(iter_num):
          rand_num = random.randint(0, len(neighbourList)-1)
          while rand_num in rand_list:
            rand_num = random.randint(0, len(neighbourList)-1)
          rand_list.append(rand_num)
    checker = False

    #for some neighbour vertices of the ith vertex in result
    #(these ares choosen randomly with the random array created previously)
    for index in rand_list:
      #if it is not already in the result
      if (neighbourList[index] not in result):
        neighbourList1 = graph[ neighbourList[index] ]
        if i+2 < len(result):
          neighbourList2 = graph[ result[i + 2] ]
          #find its common neighbour with (i+2)th vertex from result
          sharedNodes = list(set(neighbourList1).intersection(neighbourList2))
          if sharedNodes:
            for sharedNode in sharedNodes:
              #if it is not already in the result
              if sharedNode not in result:
                #add the new 2 vertices between ith and (i+2)th vertices
                #in result cycle and pops the (i+1)th vertex from result
                result.pop(i+1)
                result.insert(i+1, neighbourList[index])
                result.insert(i+2, sharedNode)
                checker = True
                break
      if checker:
        break
    rand_list.clear()

  return result
```

$$O(V^2) + (O(V) * O(V)) = O(V^2)$$

Complexity of Improved Cycle 3 is $O(V^2)$.

F. Ultimate Improved Cycle



```
def improveCycleUltimate(graph, result):

  #it only runs all the improve cycle
  #algorithms one after another
  improveCycle1(graph, result)
  improveCycle2(graph, result)
  improveCycle3(graph, result)

  return result
```

$$O(V^2) + O(V^2) + O(V^2) = O(V^2)$$

Complexity of Ultimate Improved Cycle is $O(V^2)$.

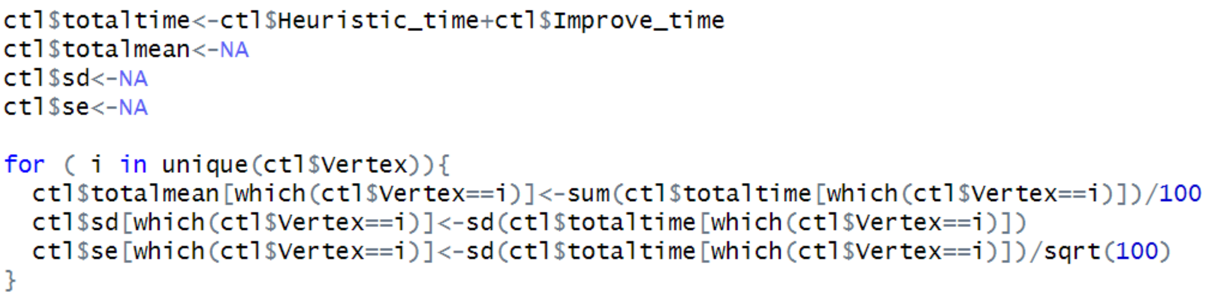
4) Experimental Analysis

Performance Testing

Testing the performance of an algorithm comes with it’s riddles. The test cases to be used is an evident aspect of the quality of the test. However, another there is another major aspect of the test quality, the representativeness of the test instances running times. In order to analyze the running time of the algorithm, for how many times an instance should be tested and how well these tests represent the overall performance should be known.

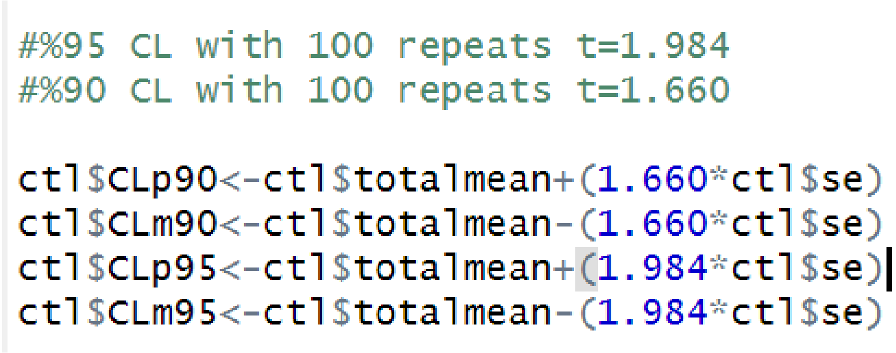
These values can be obtained by central limit theorem;

A. Estimated Standard Error



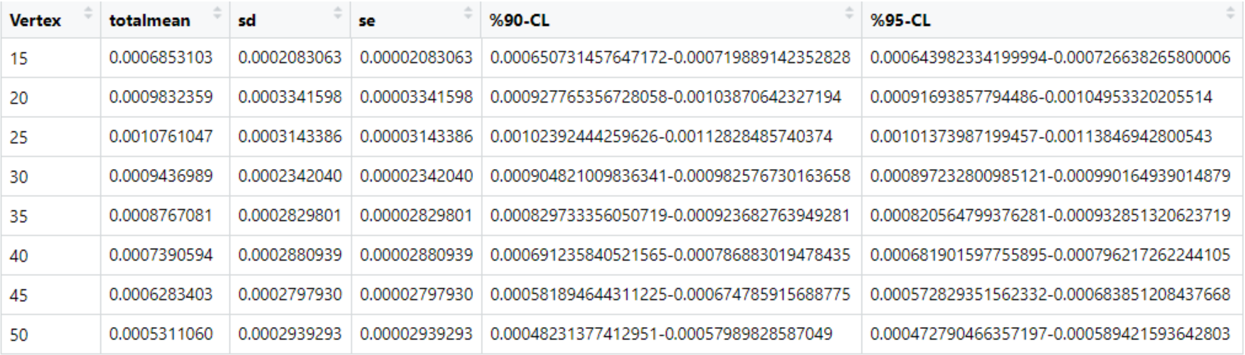
```
ctl$totaltime<-ctl$Heuristic_time+ctl$Improve_time
ctl$totalmean<-NA
ctl$sd<-NA
ctl$se<-NA

for ( i in unique(ctl$Vertex)){
  ctl$totalmean[which(ctl$Vertex==i)]<-sum(ctl$totaltime[which(ctl$Vertex==i)])/100
  ctl$sd[which(ctl$Vertex==i)]<-sd(ctl$totaltime[which(ctl$Vertex==i)])
  ctl$se[which(ctl$Vertex==i)]<-sd(ctl$totaltime[which(ctl$Vertex==i)])/sqrt(100)
}
```

B. Confidence Level & p-value



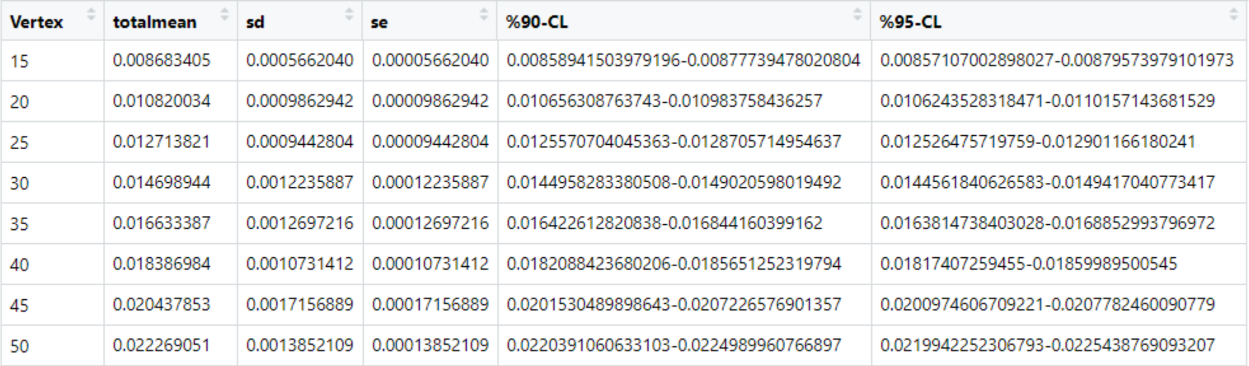
```
#%95 CL with 100 repeats t=1.984
#%90 CL with 100 repeats t=1.660

ctl$CLp90<-ctl$totalmean+(1.660*ctl$se)
ctl$CLm90<-ctl$totalmean-(1.660*ctl$se)
ctl$CLp95<-ctl$totalmean+(1.984*ctl$se)
ctl$CLm95<-ctl$totalmean-(1.984*ctl$se)
```

Heuristic Method1 + Ultimate



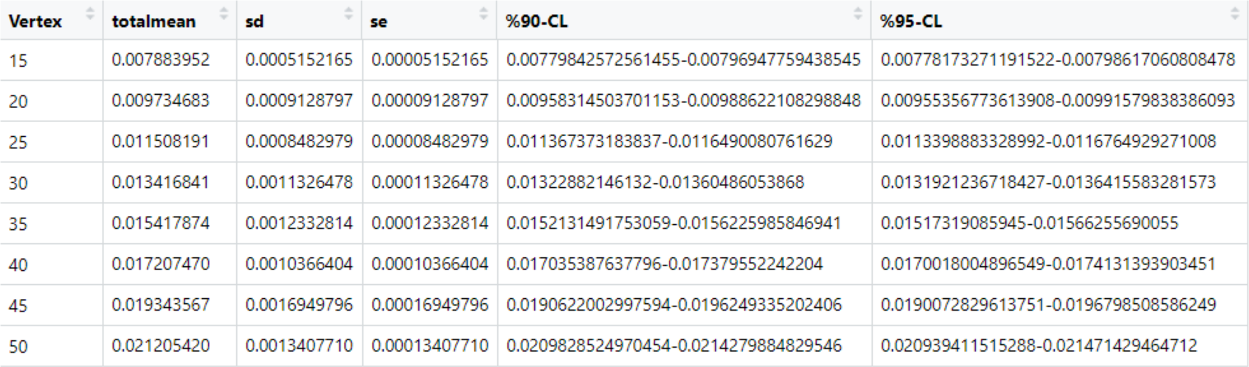
Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.0006853103	0.0002083063	0.00002083063	0.000650731457647172-0.000719889142352828	0.000643982334199994-0.000726638265800006
20	0.0009832359	0.0003341598	0.00003341598	0.000927765356728058-0.00103870642327194	0.00091693857794486-0.00104953320205514
25	0.0010761047	0.0003143386	0.00003143386	0.00102392444259626-0.00112828485740374	0.00101373987199457-0.00113846942800543
30	0.0009436989	0.0002342040	0.00002342040	0.000904821009836341-0.000982576730163658	0.000897232800985121-0.000990164939014879
35	0.0008767081	0.0002829801	0.00002829801	0.000829733356050719-0.000923682763949281	0.000820564799376281-0.000932851320623719
40	0.0007390594	0.0002880939	0.00002880939	0.000691235840521565-0.000786883019478435	0.000681901597755895-0.000796217262244105
45	0.0006283403	0.0002797930	0.00002797930	0.000581894644311225-0.000674785915688775	0.000572829351562332-0.000683851208437668
50	0.0005311060	0.0002939293	0.00002939293	0.00048231377412951-0.00057989828587049	0.000472790466357197-0.000589421593642803

Heuristic Method2 + Ultimate



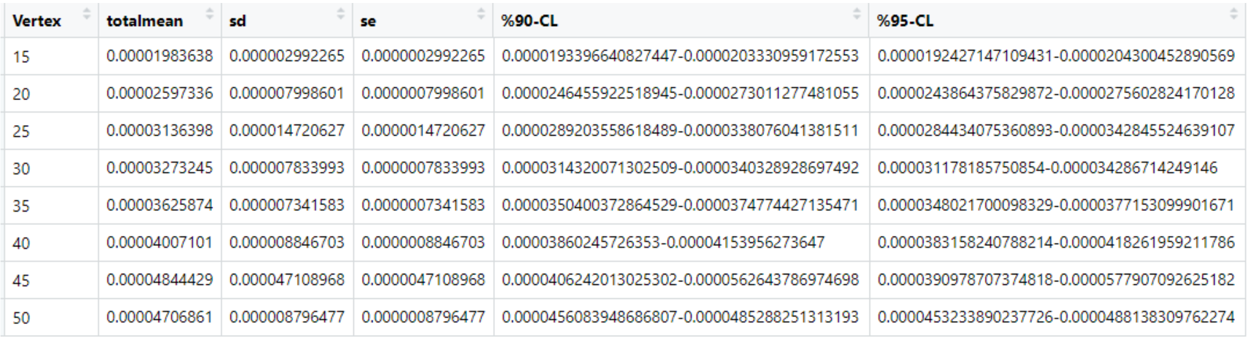
Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.008683405	0.0005662040	0.00005662040	0.00858941503979196-0.00877739478020804	0.00857107002898027-0.00879573979101973
20	0.010820034	0.0009862942	0.00009862942	0.010656308763743-0.010983758436257	0.0106243528318471-0.0110157143681529
25	0.012713821	0.0009442804	0.00009442804	0.0125570704045363-0.0128705714954637	0.012526475719759-0.012901166180241
30	0.014698944	0.0012235887	0.00012235887	0.0144958283380508-0.0149020598019492	0.0144561840626583-0.0149417040773417
35	0.016633387	0.0012697216	0.00012697216	0.016422612820838-0.016844160399162	0.0163814738403028-0.0168852993796972
40	0.018386984	0.0010731412	0.00010731412	0.0182088423680206-0.0185651252319794	0.01817407259455-0.01859989500545
45	0.020437853	0.0017156889	0.00017156889	0.0201530489898643-0.0207226576901357	0.0200974606709221-0.0207782460090779
50	0.022269051	0.0013852109	0.00013852109	0.0220391060633103-0.0224989960766897	0.0219942252306793-0.0225438769093207

Heuristic Method 2 only



Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.007883952	0.0005152165	0.00005152165	0.00779842572561455-0.00796947759438545	0.00778173271191522-0.00798617060808478
20	0.009734683	0.0009128797	0.00009128797	0.00958314503701153-0.00988622108298848	0.00955356773613908-0.00991579838386093
25	0.011508191	0.0008482979	0.00008482979	0.011367373183837-0.0116490080761629	0.0113398883328992-0.0116764929271008
30	0.013416841	0.0011326478	0.00011326478	0.01322882146132-0.01360486053868	0.0131921236718427-0.0136415583281573
35	0.015417874	0.0012332814	0.00012332814	0.0152131491753059-0.0156225985846941	0.01517319085945-0.01566255690055
40	0.017207470	0.0010366404	0.00010366404	0.017035387637796-0.017379552242204	0.0170018004896549-0.0174131393903451
45	0.019343567	0.0016949796	0.00016949796	0.0190622002997594-0.0196249335202406	0.0190072829613751-0.0196798508586249
50	0.021205420	0.0013407710	0.00013407710	0.0209828524970454-0.0214279884829546	0.020939411515288-0.021471429464712

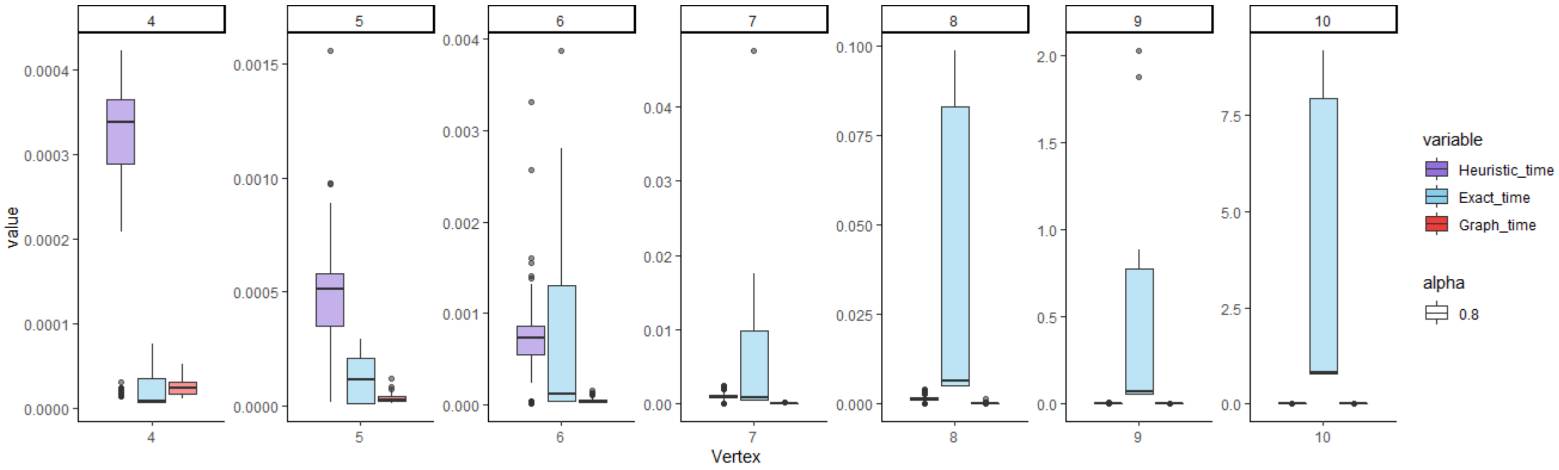
Heuristic Method 1 Only



Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.00001983638	0.000002992265	0.0000002992265	0.0000193396640827447-0.0000203330959172553	0.0000192427147109431-0.0000204300452890569
20	0.00002597336	0.000007998601	0.0000007998601	0.0000246455922518945-0.0000273011277481055	0.0000243864375829872-0.0000275602824170128
25	0.00003136398	0.000014720627	0.0000014720627	0.0000289203558618489-0.0000338076041381511	0.0000284434075360893-0.0000342845524639107
30	0.00003273245	0.000007833993	0.0000007833993	0.0000314320071302509-0.0000340328928697492	0.000031178185750854-0.000034286714249146
35	0.00003625874	0.000007341583	0.0000007341583	0.0000350400372864529-0.0000374774427135471	0.0000348021700098329-0.0000377153099901671
40	0.00004007101	0.000008846703	0.0000008846703	0.00003860245726353-0.00004153956273647	0.0000383158240788214-0.0000418261959211786
45	0.00004844429	0.000047108968	0.0000047108968	0.0000406242013025302-0.0000562643786974698	0.0000390978707374818-0.0000577907092625182
50	0.00004706861	0.000008796477	0.0000008796477	0.0000456083948686807-0.0000485288251313193	0.0000453233890237726-0.0000488138309762274

Running Time Performance

As it has been stated, Heuristic algorithms aim to maximize the speed with a sacrifice of finding the optimal solution. It's already known that theoretically, heuristic algorithms have lower running time complexity then an algorithm which finds the optimal results. In this section this theoretical running time of our algorithm is reflected on the practical running time by various analyses.



As promised by the definition of an heuristic algorithm, the practical running time of the implementation showed significant improvement in speed in comparison to optimal solution, particularly as the number of vertices increases.

In order to analyze the running time of our implementation in practice, different comparison methods are used. These methods aims to compare the running time of the heuristic algorithm with different implementations and improvements according to;

a) Vertex count,

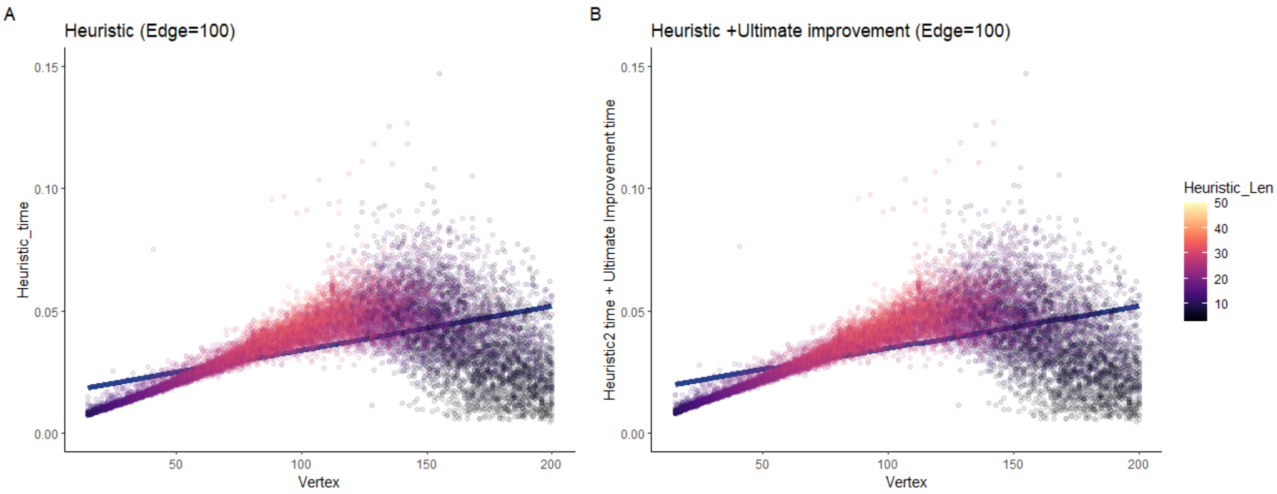
b) Edge count,

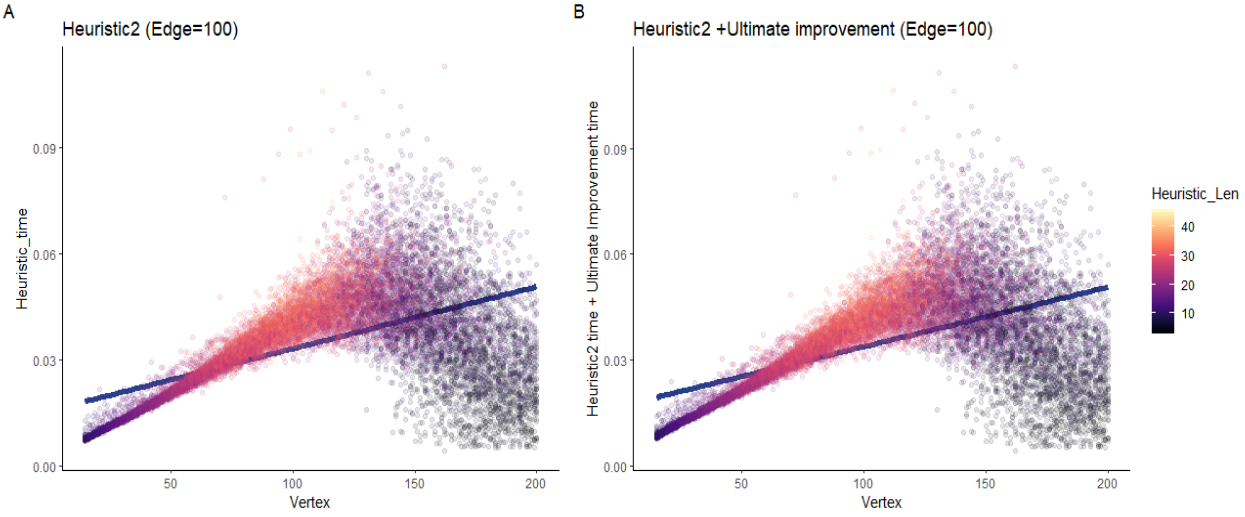
c) Between sparse and dense graphs,

d) Edge and Vertex count,

A.Vertex Change

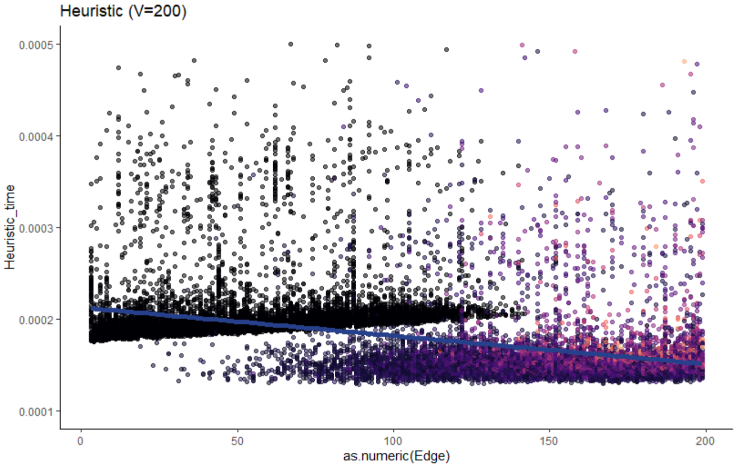
To see the running time variations with respect to vertex count a specific Edge count is chosen. The vertex range is considered to be the minimum edge possible by mathematical definitions to twice the edge count. For each case the results are repeated for 100 times as it's been found that 100 measurements gives approximately the same mean with population mean. The difference between the running time of two methods used (Heuristic1&2) to find the longest circuit can be observed from graphs with the slope of the fitted line. Heuristic method 2 is known to have higher running time but closer result to optimal, as it can be understood from the difference between the brightness profiles.

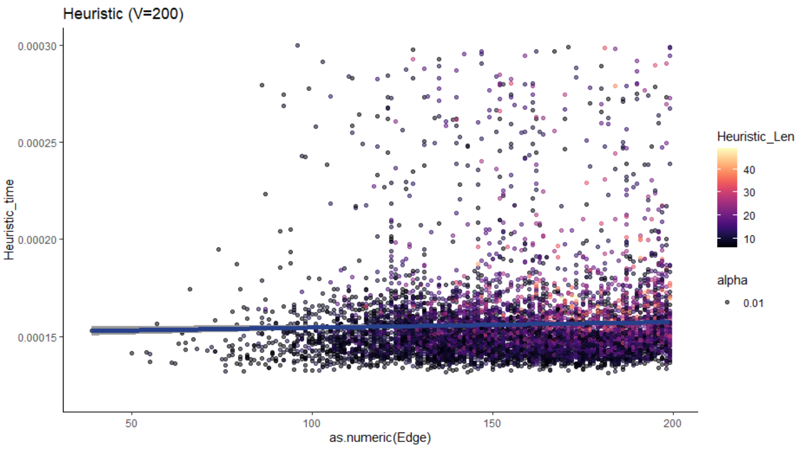


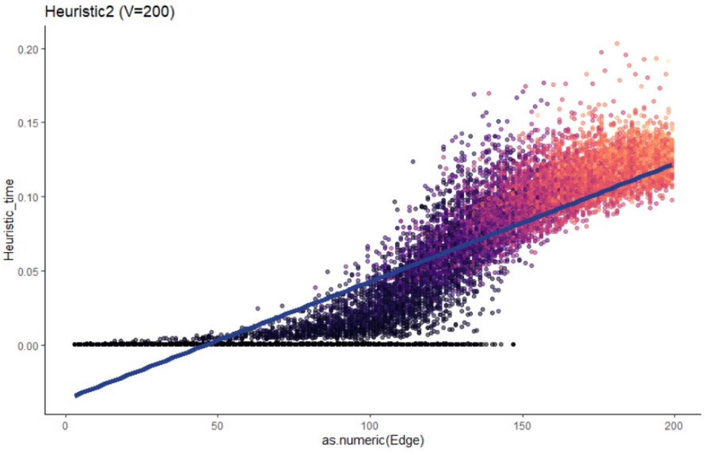


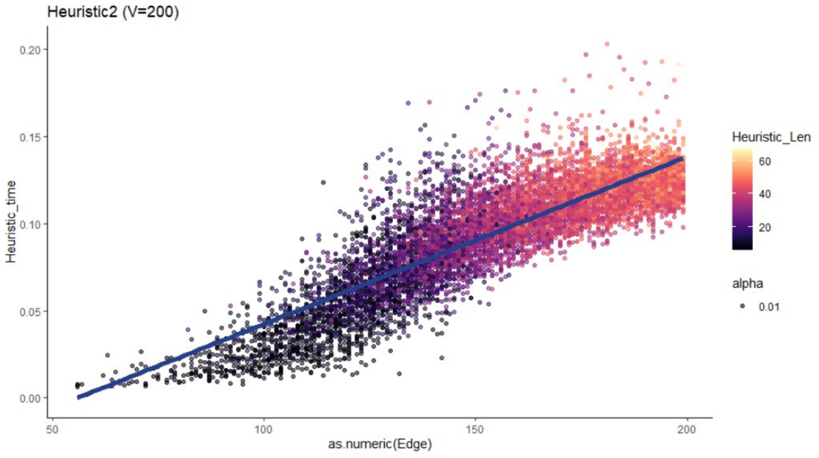
B. Edge Change

To compare the running time performance of our algorithm as the edge number changes, Vertex number kept fixed and the running time of the two methods plotted with respect to vertex number, each measurement's longest cycle length is indicated by color profile.









C. Density Change

The performance of the algorithm may vary significantly with respect to its density. The term density represents the ratio between the edge number of a graph and the maximum edge number the graph could have, as given by the Equation.

$$D_U(V,E) = \frac{|E|}{\mathrm{Max}_U(V)} = \frac{|E|}{\frac{|V|\cdot(|V|-1)}{2}} = \frac{2\cdot|E|}{|V|\cdot(|V|-1)}$$

In order to test our code’s performance for different graph densities, we designed test cases. In the designing of these cases by assigning specific densities, we categorized our graphs in five groups. These are specified with extreme sparse which has density around 1/10, sparse with density 1/4, medium graphs with density of 1/2, dense with 3/4 density and extreme sparse group with density of 9/10. For the following test cases (number of vertices) number of edges to ensure the mentioned densities are calculated as:

test_cases=[10, 50, 100, 200]

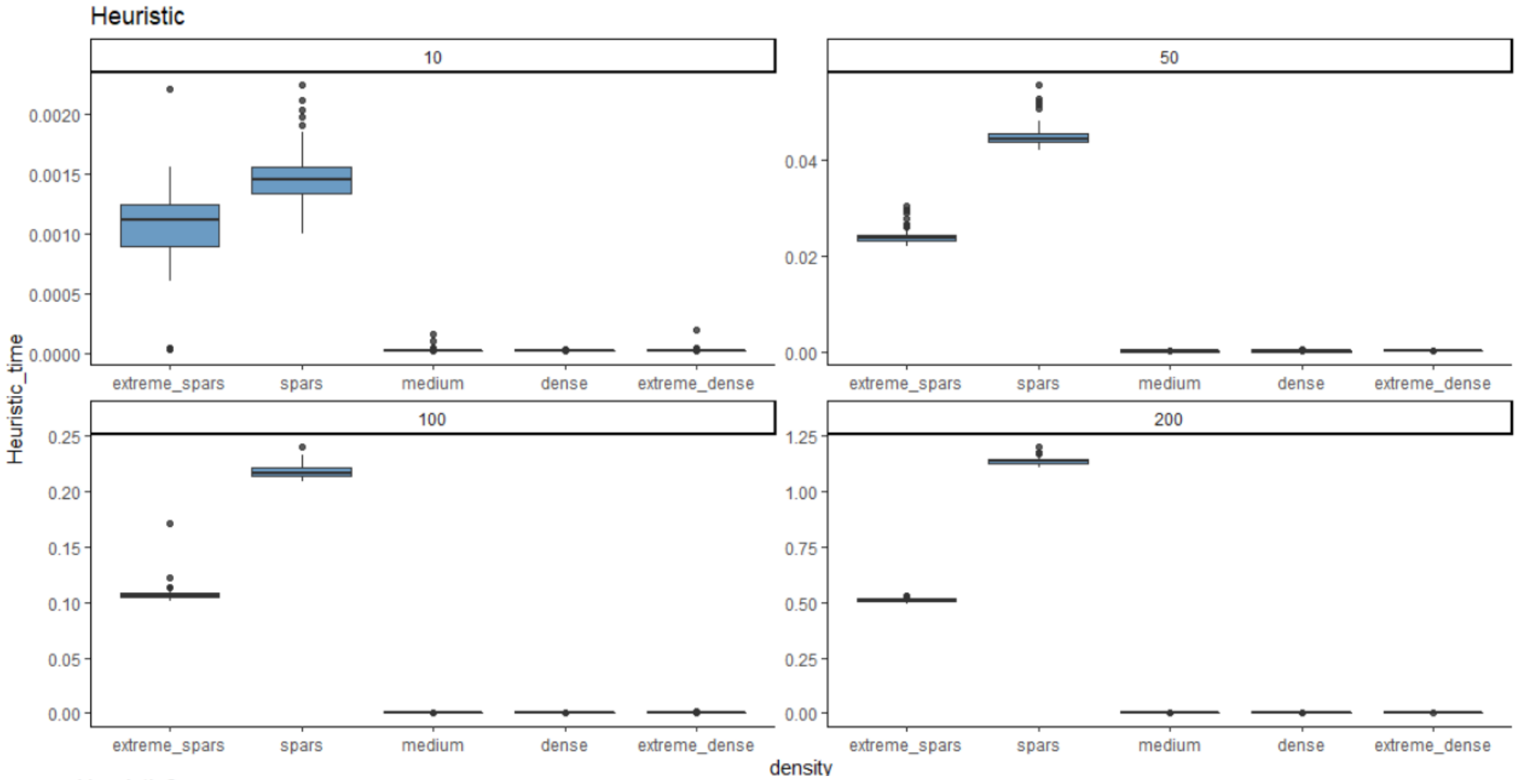
extreme_sparse=[9 ,123 ,495, 1990]

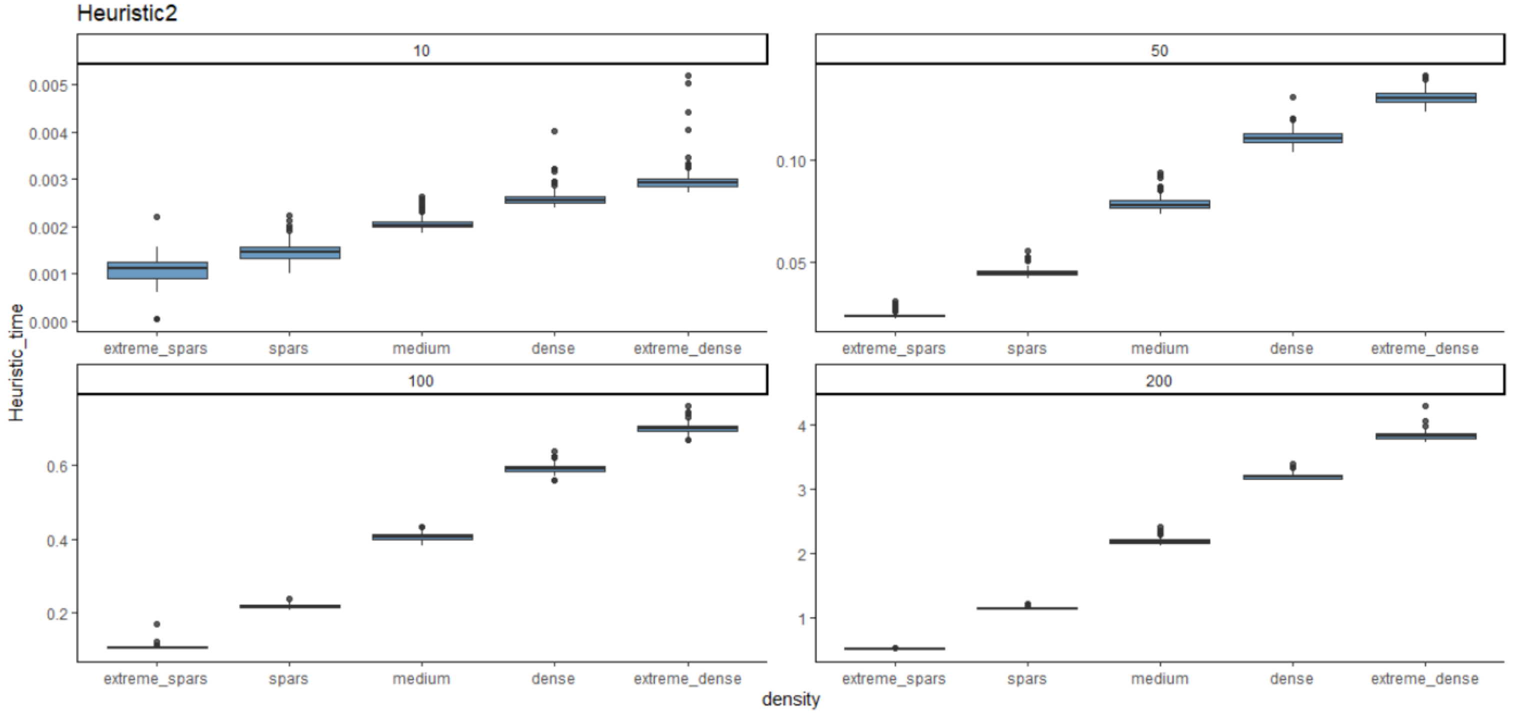
sparse=[12, 307, 1238, 4975]

medium=[23, 613, 2475, 9950]

dense=[34, 919, 3712, 14925]

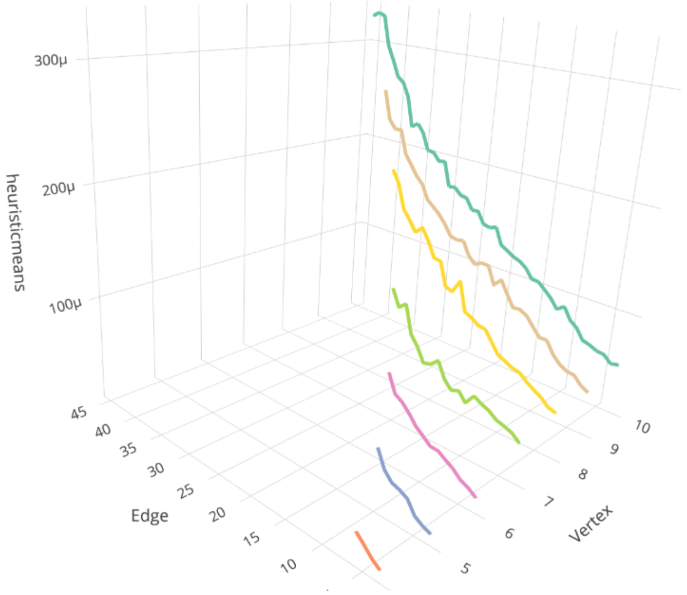
extreme_dense=[41, 1103, 4455, 17910]

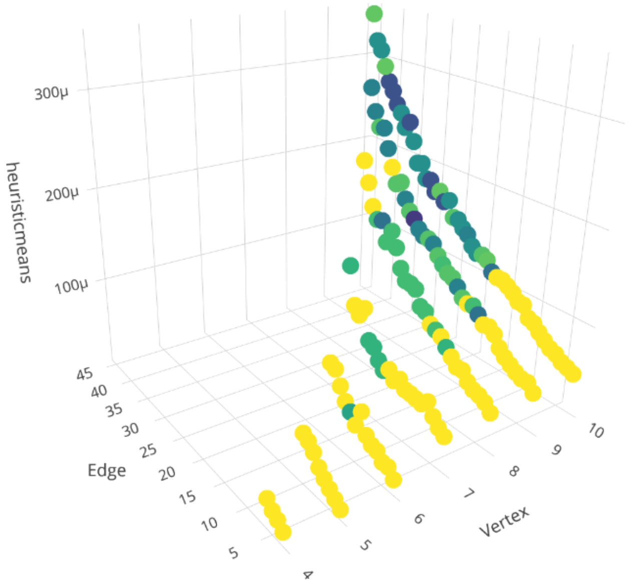

Heuristic
10
50
100
200
Heuristic_time
0.0000
0.0005
0.0010
0.0015
0.0020
0.00
0.02
0.04
0.00
0.05
0.10
0.15
0.20
0.25
0.00
0.25
0.50
0.75
1.00
1.25
extreme_spars
spars
medium
dense
extreme_dense
density


Heuristic2
10
50
100
200
Heuristic_time
0.000
0.001
0.002
0.003
0.004
0.005
0.05
0.10
0.2
0.4
0.6
1
2
3
4
extreme_spars
spars
medium
dense
extreme_dense
density

D. Edge and Vertex count

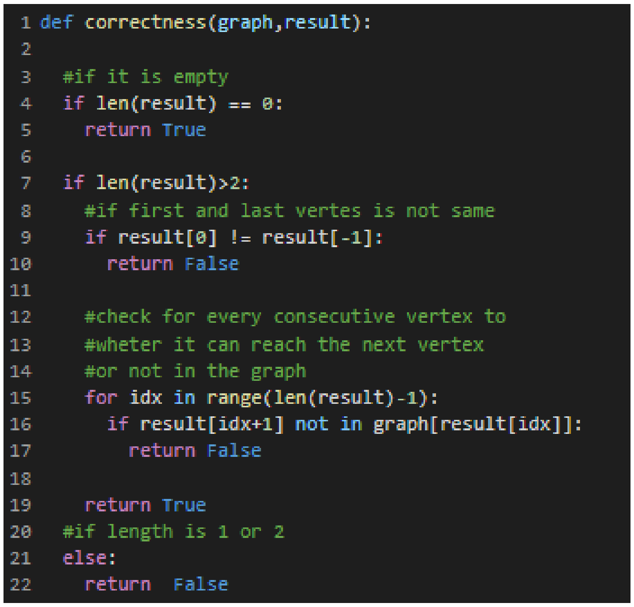
Since the input of the Longest cycle problem is a graph and the running time of the algorithm does depend on multiple factors in input such as vertices and edges and their relation, the running time could not be compared on a 2d dimension for both vertice and the edge at the same time. Thus, to see the relation between both running time, vertex number and edge number 3d plots are used. The data created by making 100 measurements for all possible edges of the first ten vertices. The mean of these 100 measurements are used to plot the line graph. The ratio bound calculated by dividing the cycle length of our algorithm to the length of the cycle which is generated by a brute force algorithm, used for plotting the point plot, the points are colored by the mean ratio bound of the 100 measurements of their cases, yellow being 1.





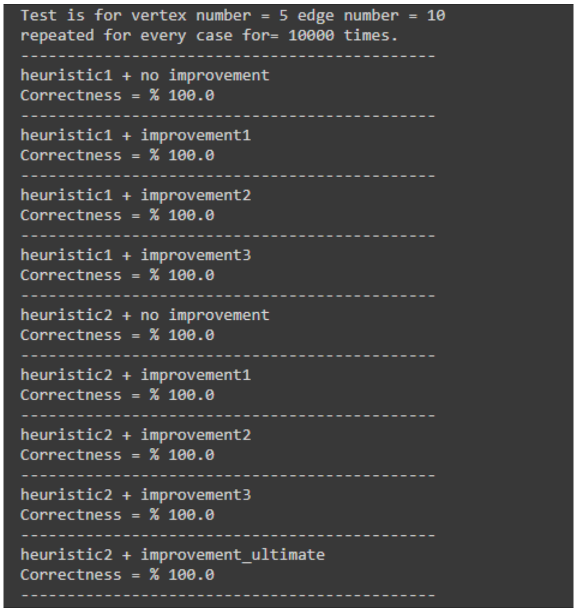
Correctness

This code below is used to calculate the correctness of the results. This code is checking whether the result from the heuristic algorithm is really a cycle or not. If it is a cycle, it also checks if the result cycle really exists in the graph or not. If both are correct, it returns True.



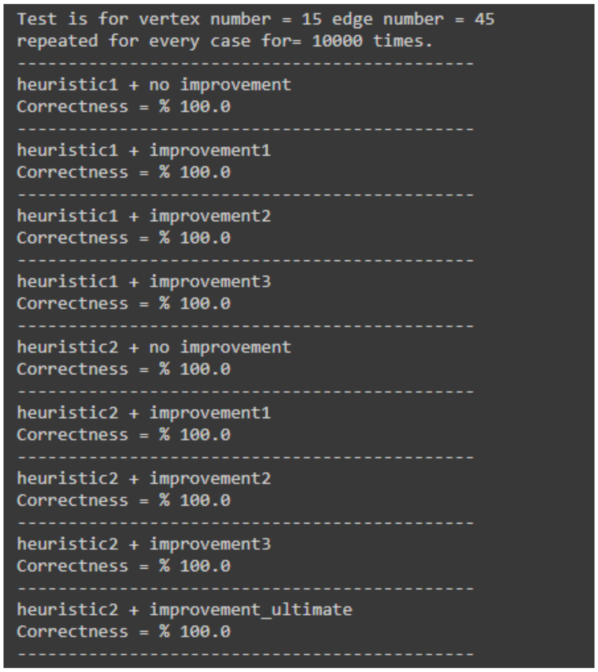
```python
def correctness(graph,result):

  #if it is empty
  if len(result) == 0:
    return True

  if len(result)>2:
    #if first and last vertes is not same
    if result[0] != result[-1]:
      return False

    #check for every consecutive vertex to
    #wheter it can reach the next vertex
    #or not in the graph
    for idx in range(len(result)-1):
      if result[idx+1] not in graph[result[idx]]:
        return False

    return True
  #if length is 1 or 2
  else:
    return  False
```

All algorithms are run for 10000 times with random graphs (with 5 vertices and 10 edges). Correctness of all algorithms are given below:



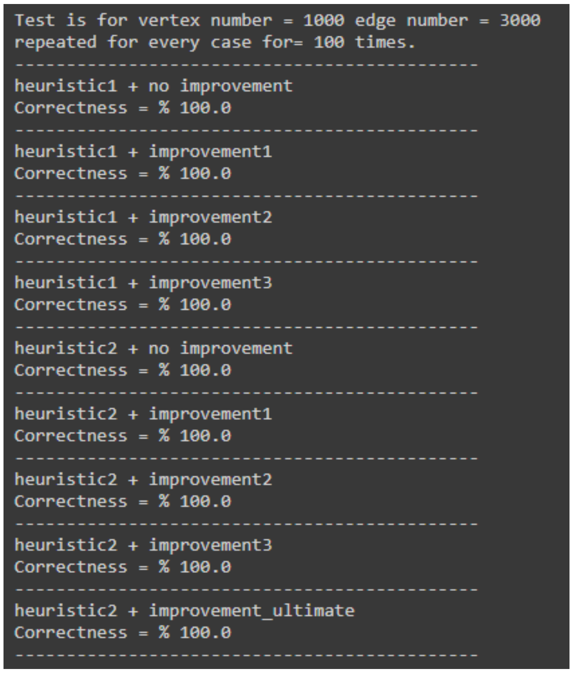
```
Test is for vertex number = 5 edge number = 10
repeated for every case for= 10000 times.
---------------------------------------------
heuristic1 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement_ultimate
Correctness = % 100.0
---------------------------------------------
```

All algorithms are run for 10000 times with random graphs (with 15 vertices and 45 edges). Correctness of all algorithms are given below:



```
Test is for vertex number = 15 edge number = 45
repeated for every case for= 10000 times.
---------------------------------------------
heuristic1 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement_ultimate
Correctness = % 100.0
---------------------------------------------
```

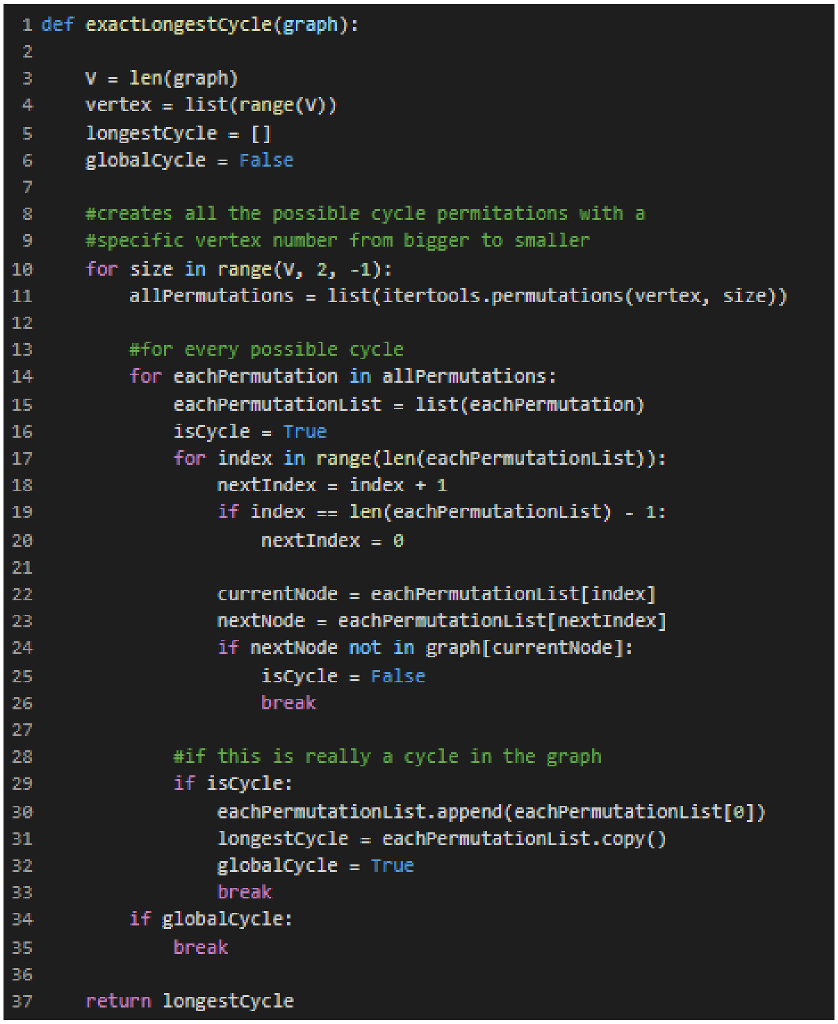
All algorithms are run for 100 times with random graphs (with 1000 vertices and 3000 edges). Correctness of all algorithms are given below:



```
Test is for vertex number = 1000 edge number = 3000
repeated for every case for= 100 times.
---------------------------------------------
heuristic1 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic1 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + no improvement
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement1
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement2
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement3
Correctness = % 100.0
---------------------------------------------
heuristic2 + improvement_ultimate
Correctness = % 100.0
---------------------------------------------
```

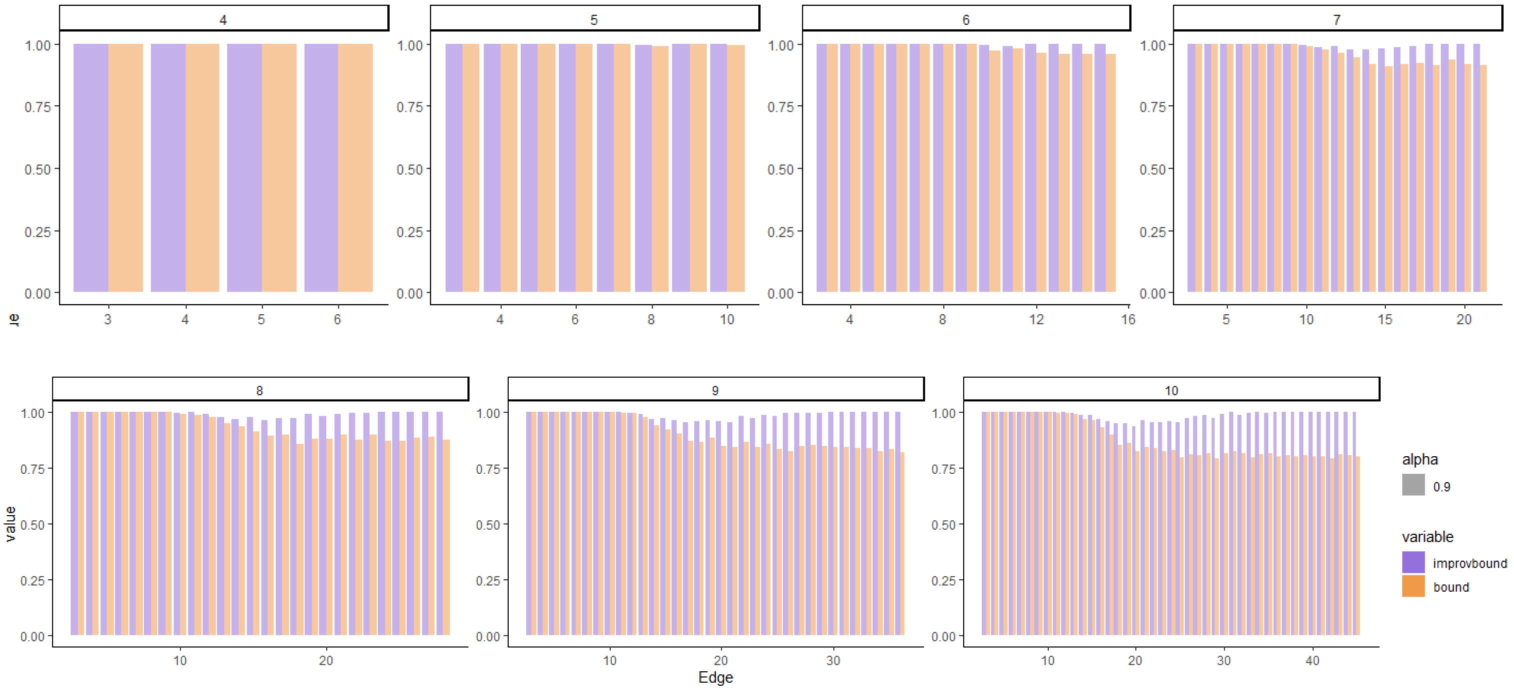
Ratio Bound

The heuristic algorithms which are used in the project are not approximation algorithms. Therefore, there is not an exact value for ratio bound. The ratio bound is calculated experimentally. The code below is the brute force algorithm which runs in logarithmic complexity. This algorithm finds the exact longest circle. This is used for calculating the ratio bound of the heuristic algorithms. The practical ratio bound is calculated by the ratio of the result from brute force exact algorithm and result from the heuristic algorithms.

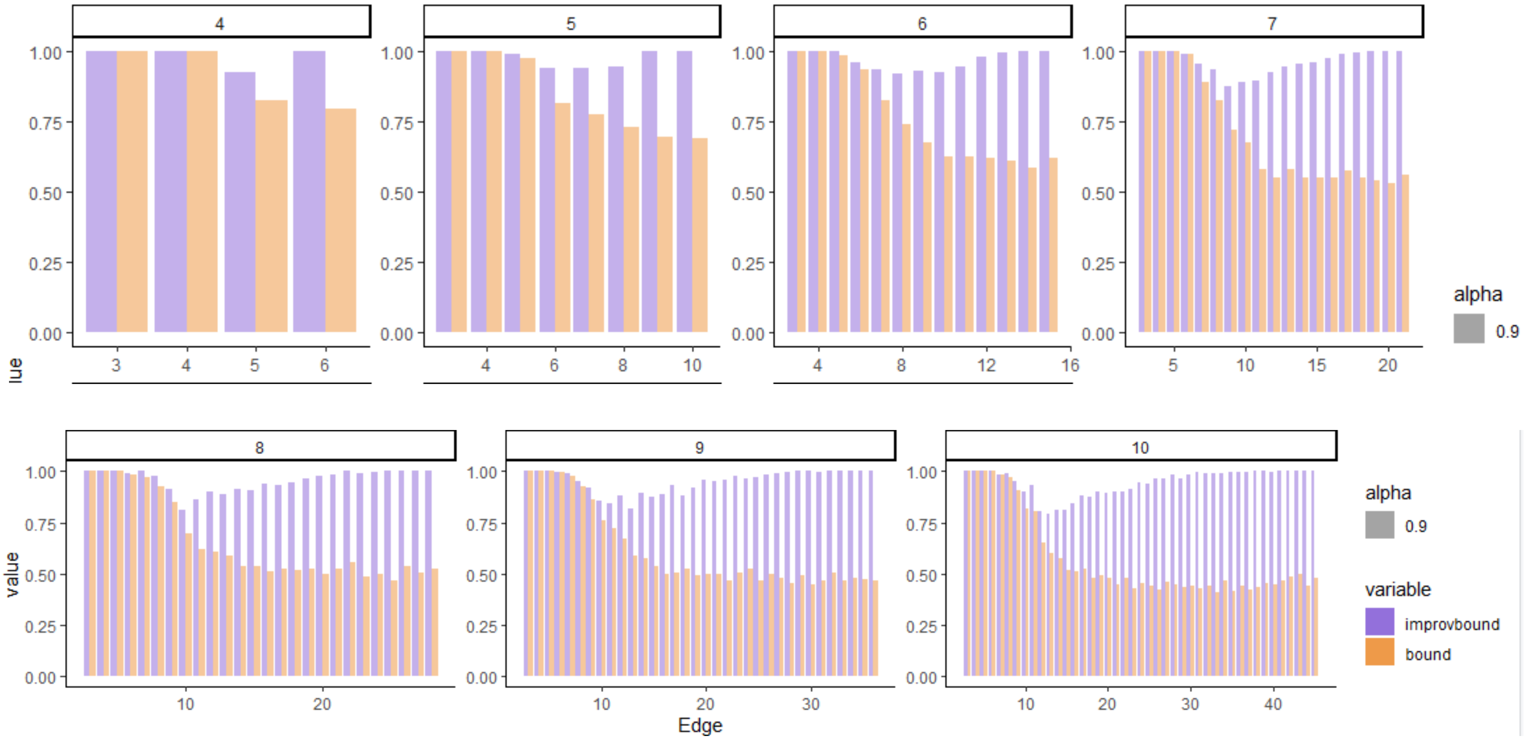


```python
def exactLongestCycle(graph):

    V = len(graph)
    vertex = list(range(V))
    longestCycle = []
    globalCycle = False

    #creates all the possible cycle permitations with a
    #specific vertex number from bigger to smaller
    for size in range(V, 2, -1):
        allPermutations = list(itertools.permutations(vertex, size))

        #for every possible cycle
        for eachPermutation in allPermutations:
            eachPermutationList = list(eachPermutation)
            isCycle = True
            for index in range(len(eachPermutationList)):
                nextIndex = index + 1
                if index == len(eachPermutationList) - 1:
                    nextIndex = 0

                currentNode = eachPermutationList[index]
                nextNode = eachPermutationList[nextIndex]
                if nextNode not in graph[currentNode]:
                    isCycle = False
                    break

            #if this is really a cycle in the graph
            if isCycle:
                eachPermutationList.append(eachPermutationList[0])
                longestCycle = eachPermutationList.copy()
                globalCycle = True
                break
        if globalCycle:
            break

    return longestCycle
```

Heuristic2



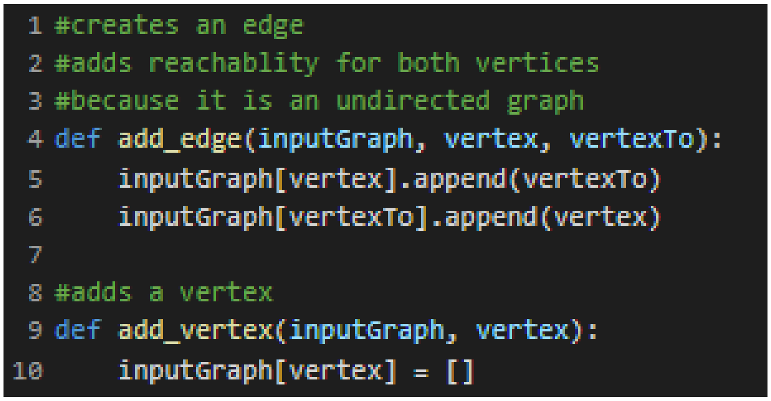
Heuristic 1



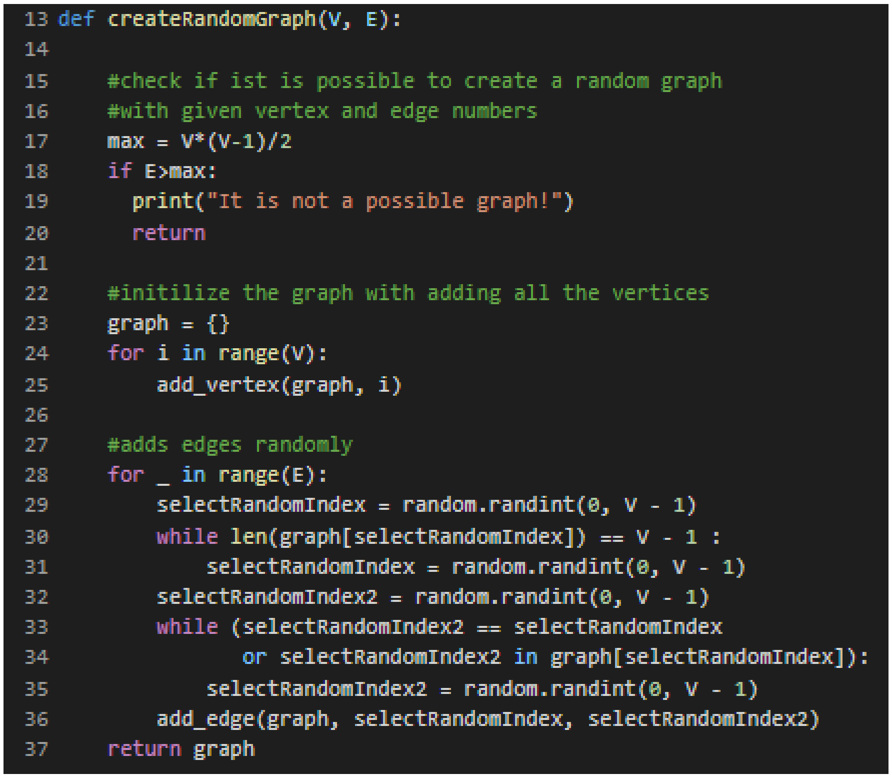
5) Testing

Black Box

Blackbox testing is used for testing the heuristic algorithm. The algorithm below is used to create random graphs for testing and experimental analysis. After random graphs are created, code in the correctness part is used for tests.

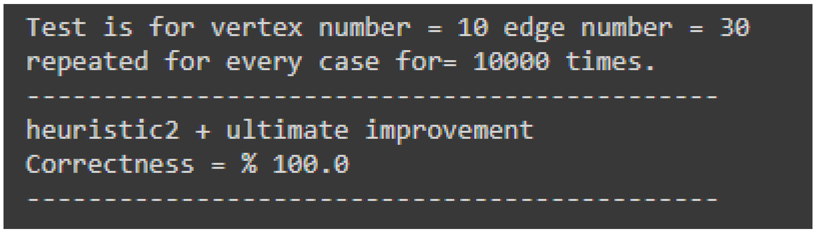


```
#creates an edge
#adds reachablity for both vertices
#because it is an undirected graph
def add_edge(inputGraph, vertex, vertexTo):
    inputGraph[vertex].append(vertexTo)
    inputGraph[vertexTo].append(vertex)

#adds a vertex
def add_vertex(inputGraph, vertex):
    inputGraph[vertex] = []
```

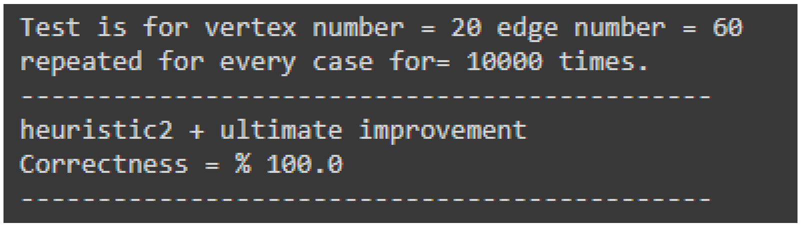


```
def createRandomGraph(V, E):

    #check if ist is possible to create a random graph
    #with given vertex and edge numbers
    max = V*(V-1)/2
    if E>max:
      print("It is not a possible graph!")
      return

    #initilize the graph with adding all the vertices
    graph = {}
    for i in range(V):
        add_vertex(graph, i)

    #adds edges randomly
    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
```

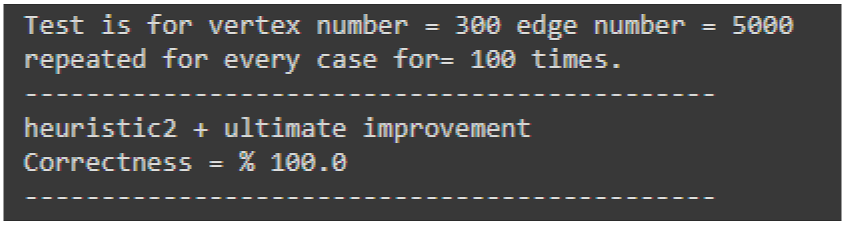
Firstly, random tests are conducted for different vertex and edge numbers for the second heuristic algorithm and ultimate improvement algorithm. Correctness tests of other algorithms are also made in the correctness part. Results are given below:



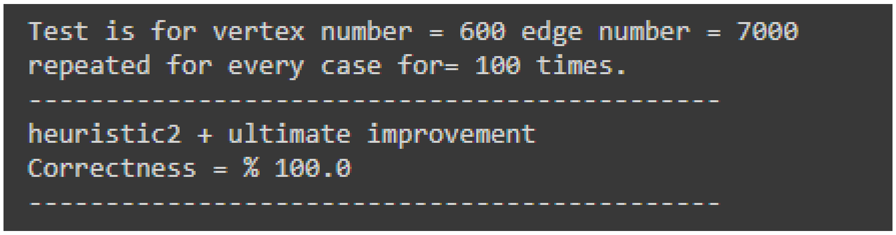
```
Test is for vertex number = 10 edge number = 30
repeated for every case for= 10000 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```



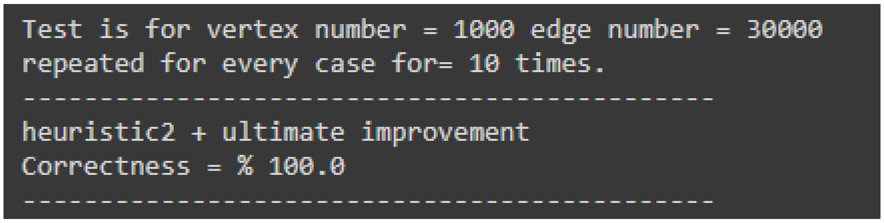
```
Test is for vertex number = 20 edge number = 60
repeated for every case for= 10000 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```



```
Test is for vertex number = 300 edge number = 5000
repeated for every case for= 100 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```

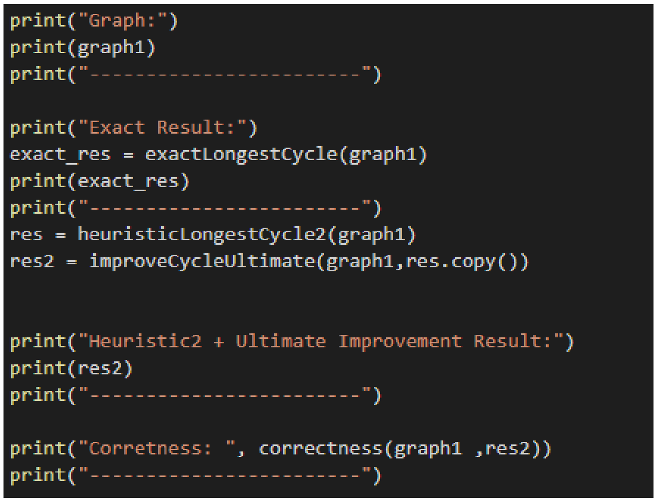


```
Test is for vertex number = 600 edge number = 7000
repeated for every case for= 100 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```



```
Test is for vertex number = 1000 edge number = 30000
repeated for every case for= 10 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```

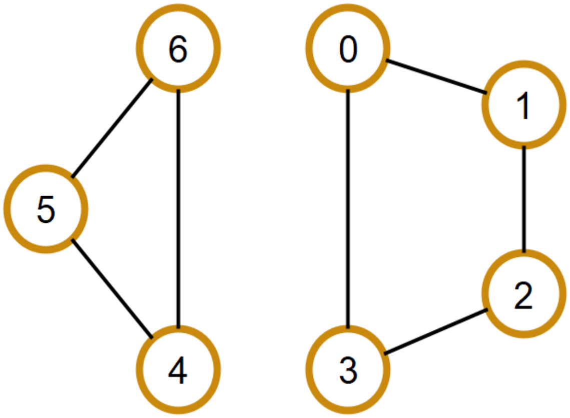
Also some corner cases are tested. Test code:



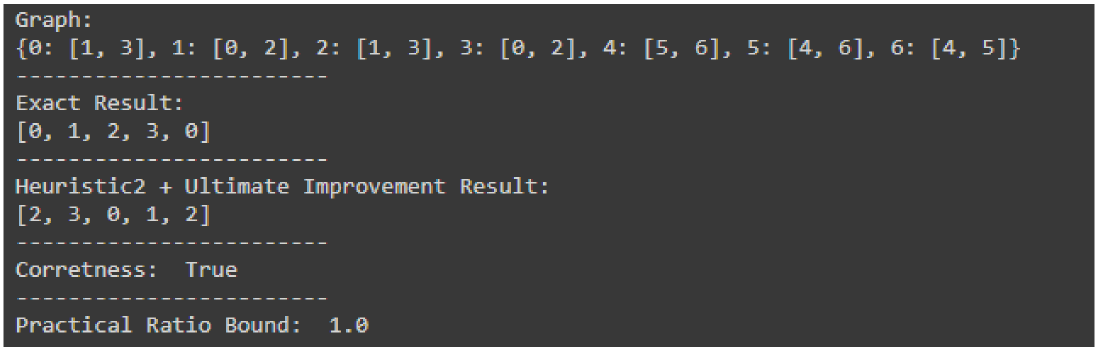
```
print("Graph:")
print(graph1)
print("------------------------")

print("Exact Result:")
exact_res = exactLongestCycle(graph1)
print(exact_res)
print("------------------------")
res = heuristicLongestCycle2(graph1)
res2 = improveCycleUltimate(graph1,res.copy())


print("Heuristic2 + Ultimate Improvement Result:")
print(res2)
print("------------------------")

print("Corretness: ", correctness(graph1 ,res2))
print("------------------------")
```

These cases and test results are given below:

1. Bipartite graph with cycles

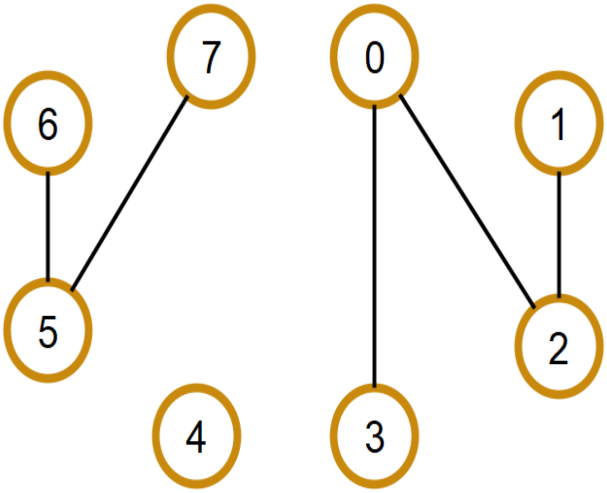


SUCCESS:

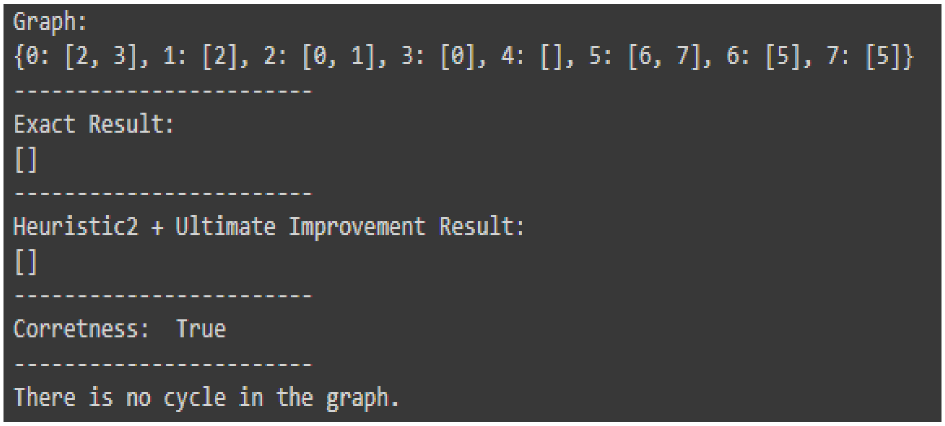


```
Graph:
{0: [1, 3], 1: [0, 2], 2: [1, 3], 3: [0, 2], 4: [5, 6], 5: [4, 6], 6: [4, 5]}
------------------------
Exact Result:
[0, 1, 2, 3, 0]
------------------------
Heuristic2 + Ultimate Improvement Result:
[2, 3, 0, 1, 2]
------------------------
Corretness:  True
------------------------
Practical Ratio Bound:  1.0
```

2. Bipartite graph without a cycle

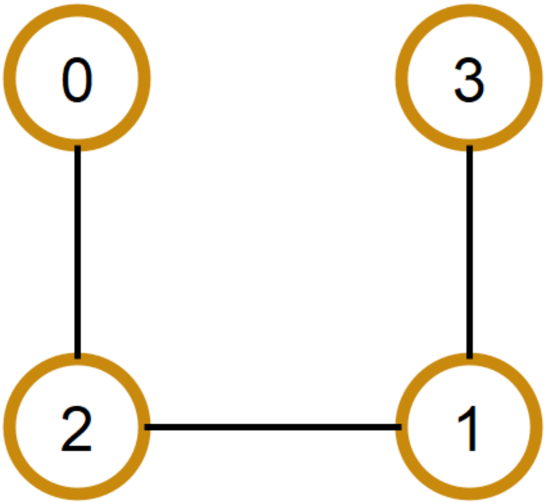


SUCCESS:

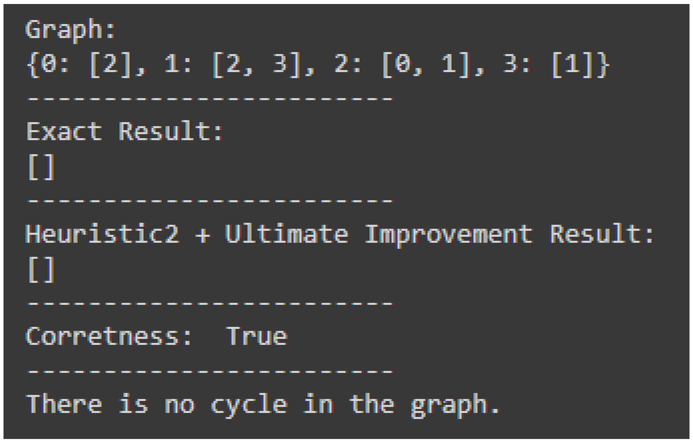


```
Graph:
{0: [2, 3], 1: [2], 2: [0, 1], 3: [0], 4: [], 5: [6, 7], 6: [5], 7: [5]}
------------------------
Exact Result:
[]
------------------------
Heuristic2 + Ultimate Improvement Result:
[]
------------------------
Corretness:  True
------------------------
There is no cycle in the graph.
```

3. Graph with no cycles

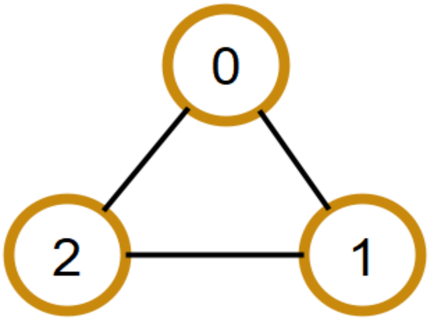


SUCCESS:

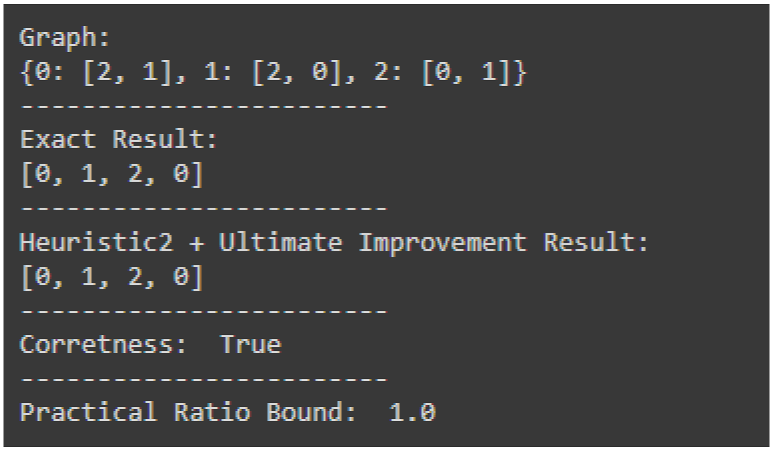


```
Graph:
{0: [2], 1: [2, 3], 2: [0, 1], 3: [1]}
------------------------
Exact Result:
[]
------------------------
Heuristic2 + Ultimate Improvement Result:
[]
------------------------
Corretness:  True
------------------------
There is no cycle in the graph.
```

4. Minimum case for a cyclic graph

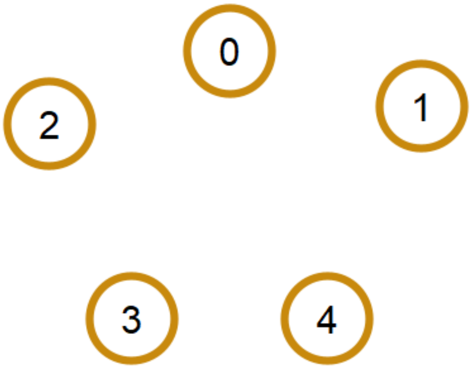


SUCCESS:

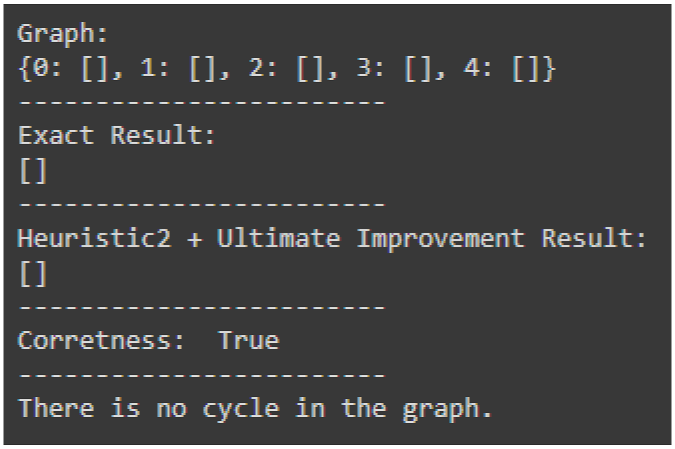


```
Graph:
{0: [2, 1], 1: [2, 0], 2: [0, 1]}
------------------------
Exact Result:
[0, 1, 2, 0]
------------------------
Heuristic2 + Ultimate Improvement Result:
[0, 1, 2, 0]
------------------------
Corretness:  True
------------------------
Practical Ratio Bound:  1.0
```

5. Graph with no edges



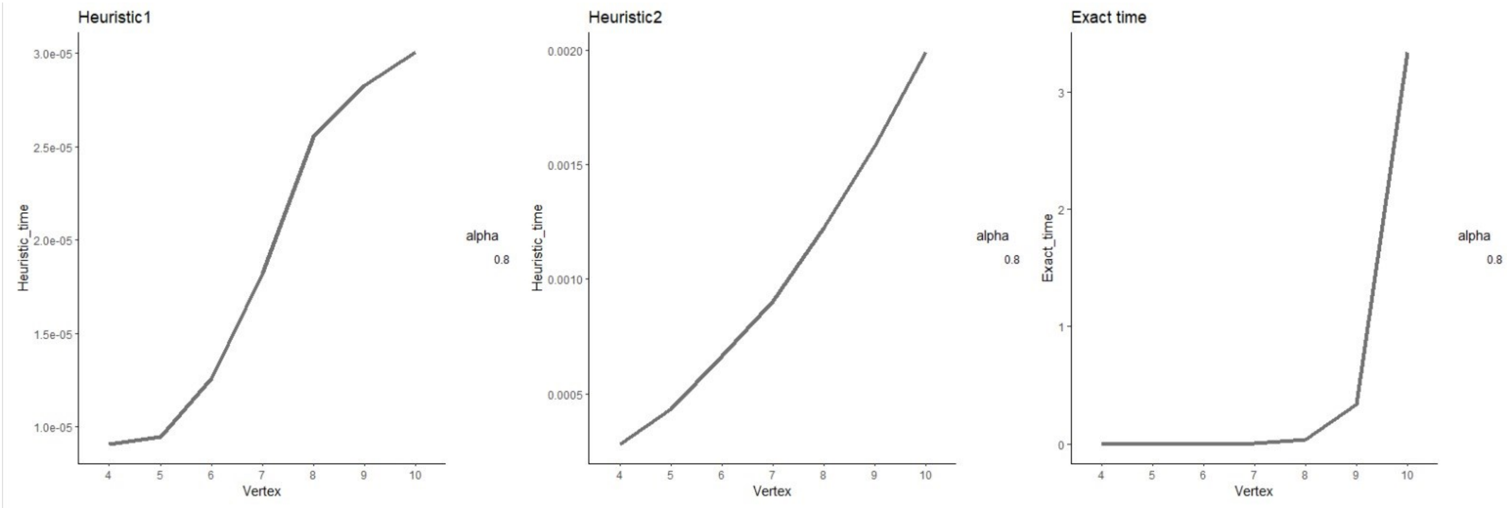
SUCCESS:



```
Graph:
{0: [], 1: [], 2: [], 3: [], 4: []}
------------------------
Exact Result:
[]
------------------------
Heuristic2 + Ultimate Improvement Result:
[]
------------------------
Corretness:  True
------------------------
There is no cycle in the graph.
```

6) Discussion

Heuristic approaches have been widely used to solve the NP_hard problems, since sacrificing the optimality results in great profit in time. The longest circuit problem is a well known NP-hard problem and the heuristic algorithm implemented in this project. In the first chapter of the project, it is shown that the longest circuit problem is a NP-hard problem. Therefore, it is impossible (if P is not equal NP) to solve this problem in polynomial time. It can also be seen from the algorithm in the project which finds the exact solution for the longest circuit problem. The exact solution algorithm has logarithmic asymptotic complexity. It is very hard to reach solutions with that algorithm for the graphs with many vertices and edges. Because of that it is better to use an heuristic algorithm which has polynomial complexity. However, these heuristic algorithms cannot find the best solution for every case. That's why it is important to increase the ratio bound of these heuristic algorithms without increasing the time complexity. In this project, two different heuristic algorithms are proposed to solve the longest cycle problem. These two heuristic algorithms have the same time complexity which is O(V^2+VE). But, they run at different times in practice. For most cases the first heuristic algorithm runs faster than the second algorithm. First one runs for its worst case running time when the edge number of the input graph is small. In addition both heuristic algorithms run at similar times for these cases. However, as edge number increases, the running time of the first algorithm is much faster than the second algorithm. Second heuristic algorithm is slower but better in terms of the bound ratio. Especially for the bigger graphs, the ratio bound of the second algorithm is much closer to the 1. These running time performances and ratio bound comparisons can be understood with the plots in the fourth chapter of the project. Additionally, the running time comparison for both heuristic algorithms and exact solution algorithm can be seen in the graph below:



Apart from the heuristic algorithms, there are additional three improvement algorithms. These algorithms take the results from heuristic algorithms and try to improve. These algorithms do not change the overall time complexity, but increase the ration bound of both heuristic algorithms. These improvements in the ratio bounds can be seen in the ration bound part in chapter four. Finally, black box testing is applied to all the algorithms for both random graphs and some corner cases. After the tests, it is shown that all the algorithms find the correct result even if the ratio bound is not equal to 1. So, the correctness of all algorithms is hundred percent.

7) Acknowledgement

We would like to thank our instructor Husnu Yenigun and our teaching assistant Furkan Reha Tutaş for their support. We have learned a lot from their experiences.

Task Distribution Table

Defne Çirci-24939 · Worked on Np-Completeness proof of the problem · Worked on implementing the heuristic and improvements algorithms. · Worked on complexity analysis of the algorithms · Worked on black box tests for the algorithm · Worked on writing the report
Efe Öztaban-25202 · Worked on implementing the heuristic and improvements algorithms. · Worked on correctness analysis of the algorithms · Worked on complexity analysis of the algorithms · Worked on the ratio bounds of algorithms · Worked on writing the report
Kayra Bilgin-25117 · Worked on implementing the heuristic and improvements algorithms. · Worked on the correctness algorithm and correctness tests · Worked on the descriptions for all the algorithms · Worked on black box tests for the algorithm · Worked on writing the report
Zeynep Kılınç-25159 · Worked on the confidence level calculations · Worked on the running time performance tests · Worked on the ratio bounds of algorithms · Worked on plots of tests · Worked on writing the report

References:

1-https://www.statisticshowto.com/hamiltonian-cycle/#:~:text=Applications%20of%20Hamiltonian%20cycles%20and%20Graphs&text=It%20has%20real%20applications%20in,mapping%20genomes%2C%20and%20operations%20research

2-
https://www.researchgate.net/publication/278009819_A_Heuristic_Algorithm_for_Longest_Simple_Cycle_Problem

3-
https://www.researchgate.net/publication/220224169_The_Longest_Path_Problem_has_a_Polynomial_Solution_on_Interval_Graphs

4-https://core.ac.uk/download/pdf/151161971.pdf

5-
https://cseweb.ucsd.edu/classes/fa98/cse101/hw3ans/hw3ans.html#:~:text=If%20there%20is%20no%20Hamiltonian,SIMPLE%20CYCLE%20is%20NP%2Dhard