Longest Simple Circuit

CS301 Algorithms

Defne Çirci

Efe Öztaban

Kayra Bilgin

Zeynep Kılınç

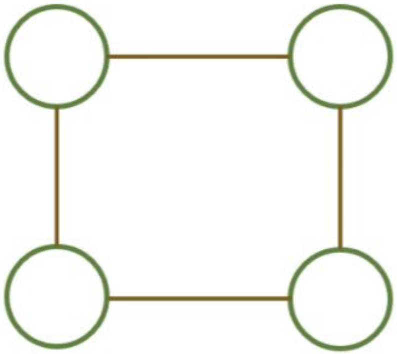
OUTLINE

- Problem Description
- Algorithm Description
- Algorithm Analysis
- Experimental Analysis
- Testing
- Discussion

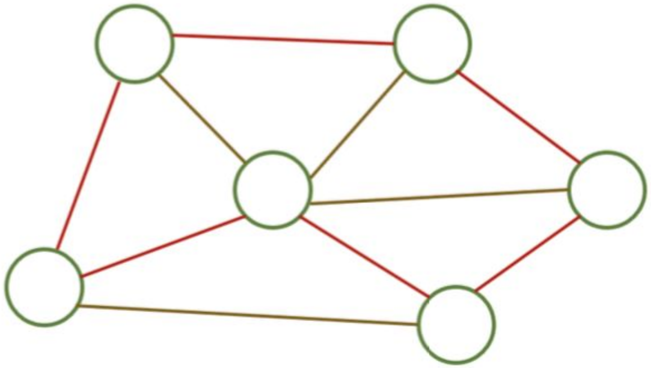
Problem Description

Longest Simple Cycle

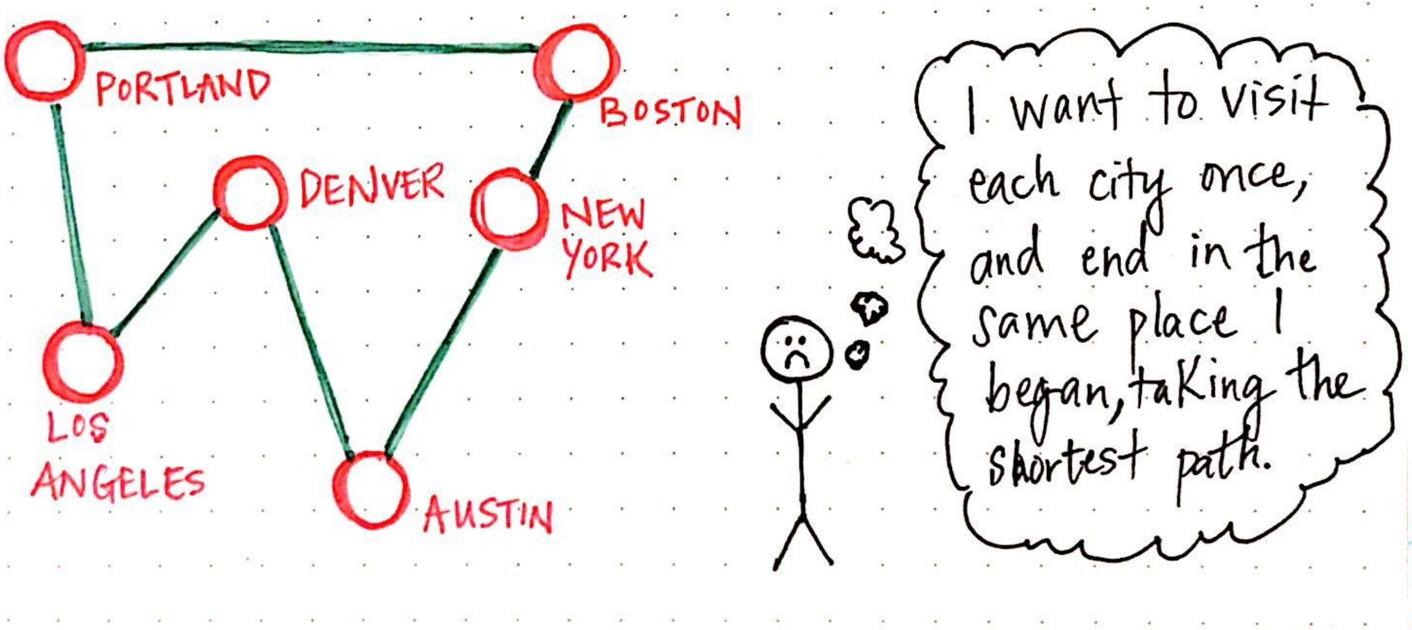
Simple Cycle



Longest Simple Cycle



Traveling Sales"woman" Problem



Longest Simple Cycle is NP-Complete

1. Show that LSC is Nondeterministically Polynomial (NP)

2. Find another problem P which is known to be NP-Complete

$\downarrow$

$\boxed{\text{Hamiltonian Circuit}}$

3. Show that P can be transformed into LSC in polynomial time

Algorithm Description

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

```

```python
    #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 []
```

Heuristic Longest Cycle 2

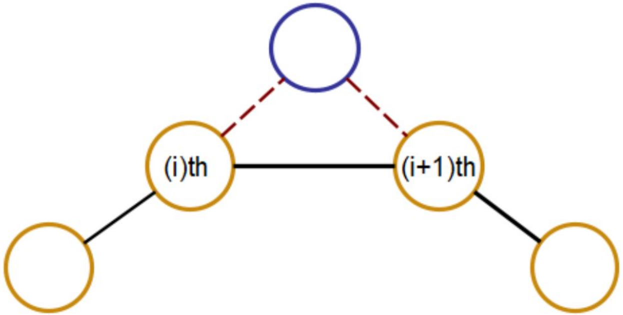
```python
def heuristicLongestCycle2(graph):

  #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 = []
```

```python
  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
```

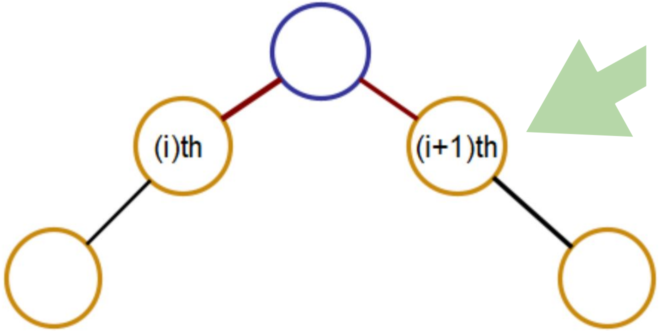
```python
  #finds the maximum result from all the possible results found
  if len(pos_results)!=0:
    max = 0
    res = []
    for i in pos_results:
      if len(i)>max:
        res = i
        max = len(i)
    #returns the longest cycle from found cycles
    return res
  #if no cycle is found
  else: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

Improved Cycle 1

- Finds the intersections
- Adds the first of the common reachable vertex

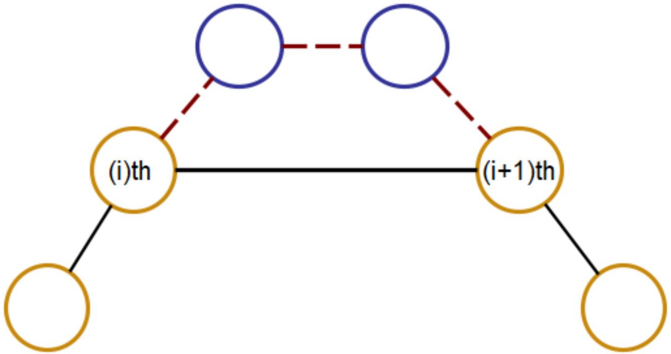



```python
def improveCycle1(graph, result):

    #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
```

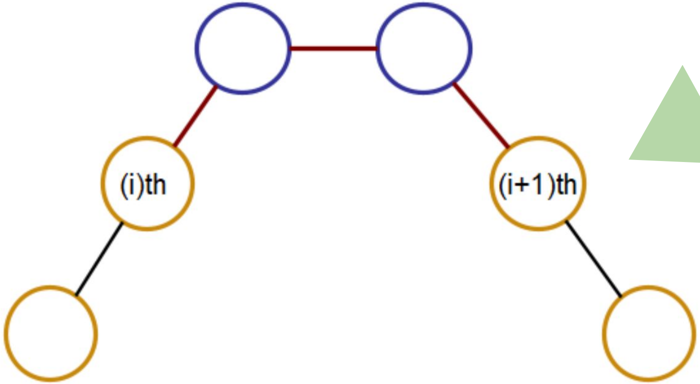


Improved Cycle 2

- Randomly selects vertices
- Finds the intersections of selected vertices and (i+1)th vertex
- Adds the new pair

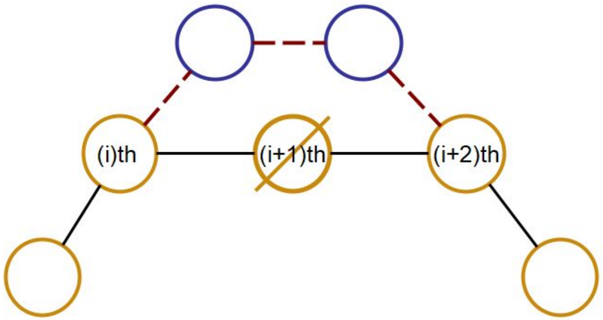



```python
def improveCycle2(graph, result):

  #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
```

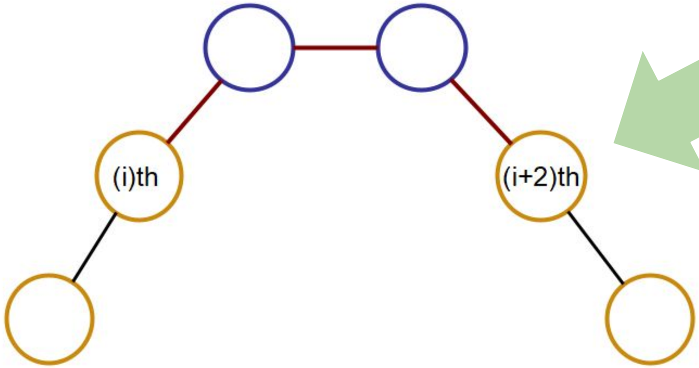


Improved Cycle 3

- Randomly selects vertices
- Finds the intersections of selected vertices and (i+2)th vertex
- Pop (i+1)th vertex
- Adds the new pair



```python
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
```



Ultimate Improved Cycle

- Runs all the improved cycle algorithms

```python
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
```

Algorithm Analysis

Correctness Analysis

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

- Basic Idea:

  - Initialization phase: Before the first iteration of the loop, result is correct because it is empty.

  - Maintenance: After the first iteration, there are two cases:

    1) If no cycle found, result is still empty.
    2) If a cycle is found, the updated result is definitely a simple cycle. Because;
       - every visited vertex is marked and it is impossible to visit a vertex more than once in the algorithm
       - the search is stopped only when when the first and last vertex in the path is the same

    For both cases, result is correct.

  - Termination: The loop is terminated only for two different cases:
    1) If all paths are searched and no cycle is found. Result is still empty for this case.
    2) If a cycle is found, loop is terminated. Result is definitely a simple cycle as shown.

    For both cases, result is correct.

Complexity Analysis

Complexity of DFS:

```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
```

$O(V+E)$

- Complexity of DFS is O(V+E).

Complexity Analysis

Complexity of Heuristic Longest Cycle:

```
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+E)

O(V(V+E))

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

Complexity Analysis

Complexity of 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 []
```

O(V)

O(V+E)

O(V(V+E))

O(V)

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

Complexity Analysis

Complexity 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
```

$O(V)$

$O(V^2)$

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

Complexity Analysis

Complexity 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
```

$O(V^2)$

$O(V)$

$O(V^2)$

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

Complexity Analysis

Complexity 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
```

$O(V^2)$

$O(V^2)$

$O(V)$

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

Complexity Analysis

Complexity of Ultimate Improved Cycle:

```python
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 $O(V^2)$.

Experimental Analysis

Performance Testing

- Each measurement → different running time
- Central Limit Theorem
- Sufficiently large data → Finite level of variance → ≅ Mean (M)
- Define;
  - Mean of your set (m)
  - Number of measurements (N)
  - Standard deviation (sd)
  - Standard error (se)
  - Confidence level (CL%)

Vertex	Edge	Iteration	Graph_time	Heuristic_time	Heuristic_Len	Improve_time	Improve_len
15	100	0	0.000734091	0.007646561	11	0.000991821	15
15	100	1	0.000629425	0.007773399	9	0.001401186	15
15	100	2	0.000547886	0.007597923	10	0.000526667	15
15	100	3	0.000448704	0.007563114	11	0.000585794	15
15	100	4	0.000510216	0.007708311	10	0.000699759	15
15	100	5	0.000620127	0.008001089	12	0.000832796	15

- Edge=100, Vertex=[15,20,25,30,35,40,45,50]

```
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)
}
#%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)
```

Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.007883952	0.0005152165	5.152165e-05	0.00779842572561455-0.00796947759438545	0.00778173271191522-0.00798617060808478
20	0.009734683	0.0009128797	9.128797e-05	0.00958314503701153-0.00988622108298848	0.00955356773613908-0.00991579838386093
25	0.011508191	0.0008482979	8.482979e-05	0.011367373183837-0.0116490080761629	0.0113398883328992-0.011676492927 1008
30	0.013416841	0.0011326478	1.132648e-04	0.01322882146132-0.0136048605386 8	0.013192123671 8427-0.0136415583281573
35	0.015417874	0.0012332814	1.233281e-04	0.0152131491753059-0.0156225985846941	0.01517319085945-0.0156625569 0055
40	0.017207470	0.0010366404	1.036640e-04	0.017035387637796-0.017379552242204	0.0170018004896549-0.0174131393903451
45	0.019343567	0.0016949796	1.694980e-04	0.0190622002997594-0.0196249335202406	0.0190072829613751-0.0196798508586249
50	0.021205420	0.0013407710	1.340771e-04	0.0209828524970454-0.0214279884829546	0.0209394115152 88-0.021471429464712

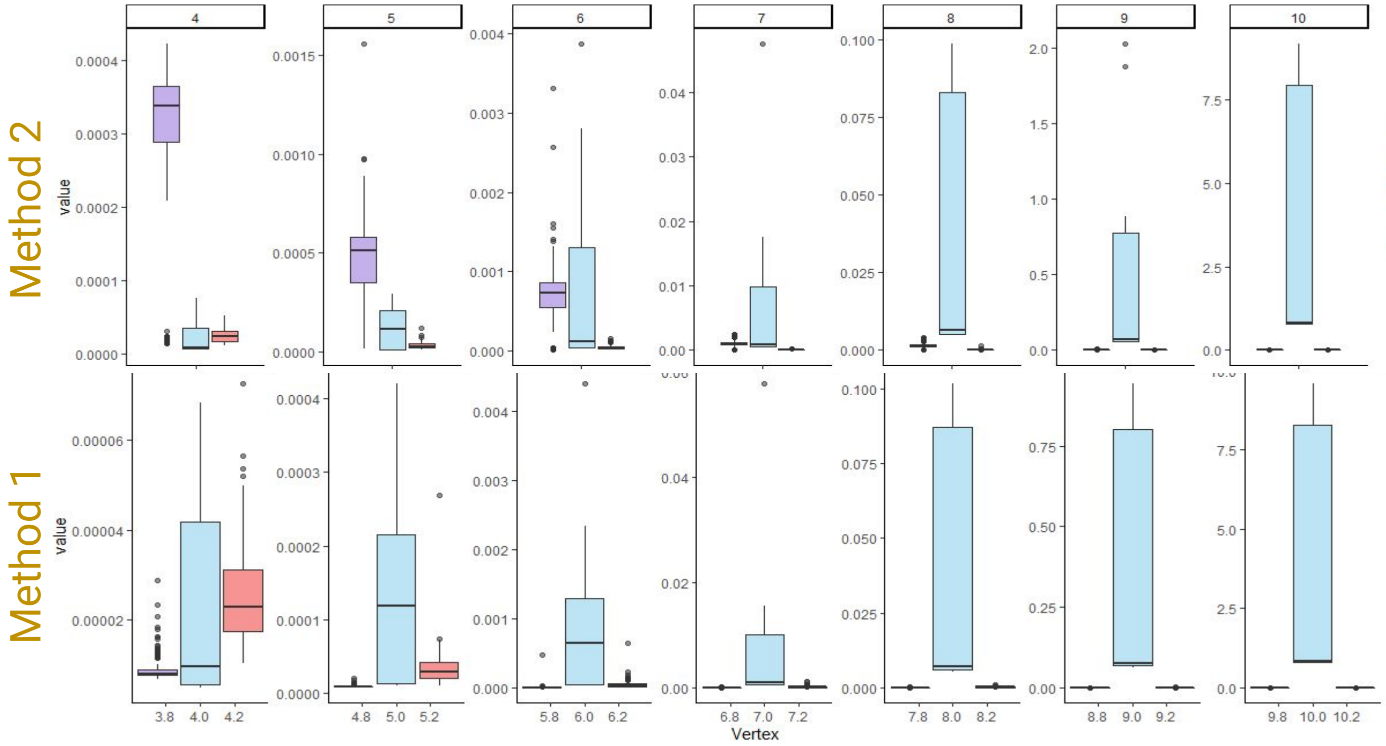
Method 1

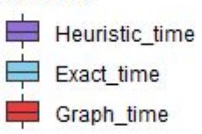
Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.00001983638	0.000002992265	0.0000002992265	0.0000193396640827447-0.0000203330959172553	0.0000192427147109431-0.0000204300452890569
20	0.00002597336	0.000007998601	0.0000007998601	0.0000246455922518945-0.0000273011277481055	0.0000243864375829872-0.0000275602824170128
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	0.000031178185750854-0.00003428671424914 6
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

Method 2

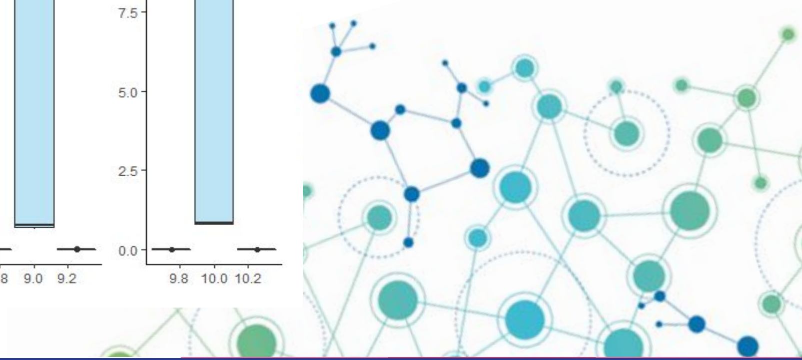
Vertex	totalmean	sd	se	%90-CL	%95-CL
15	0.007883952	0.0005152165	0.00005152165	0.00779842572561455-0.00796947759438545	0.00778173271191522-0.00798617060808478
20	0.009734683	0.0009128797	0.00009128797	0.00958314503701153-0.00988622108298848	0.00955356773613908-0.00991579838386093
25	0.011508191	0.0008482979	0.00008482979	0.011367373183837-0.011649008076162 9	0.0113398883328992-0.0116764929271008
30	0.013416841	0.0011326478	0.00011326478	0.0132288214613 2-0.0136048605386 8	0.0131921236718427-0.0136415583281573
35	0.015417874	0.0012332814	0.00012332814	0.0152131491753059-0.0156225985846941	0.01517319085945-0.0156625569005 5
40	0.017207470	0.0010366404	0.00010366404	0.0170353876377 96-0.0173795522420 4	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

Running Time Performance



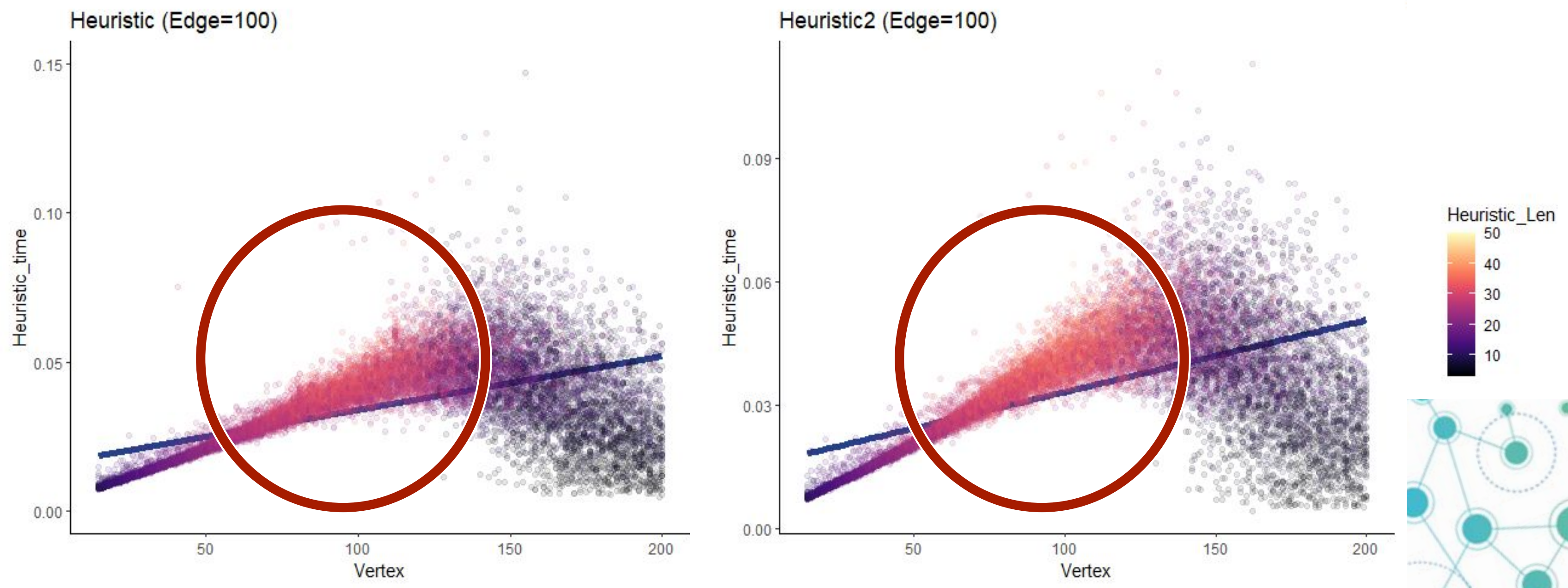


- Vertex count,
- Edge count,
- Edge ∧ Vertex
- Density,

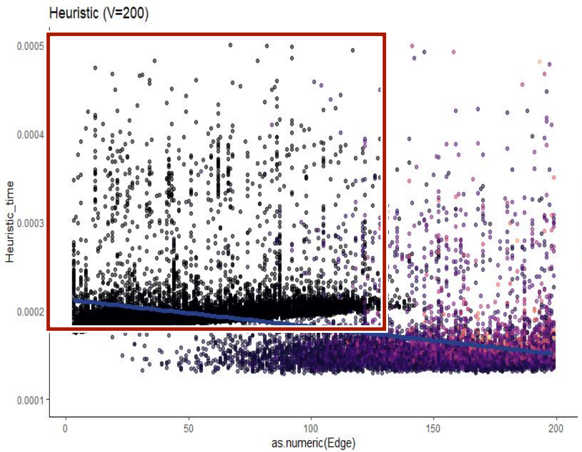


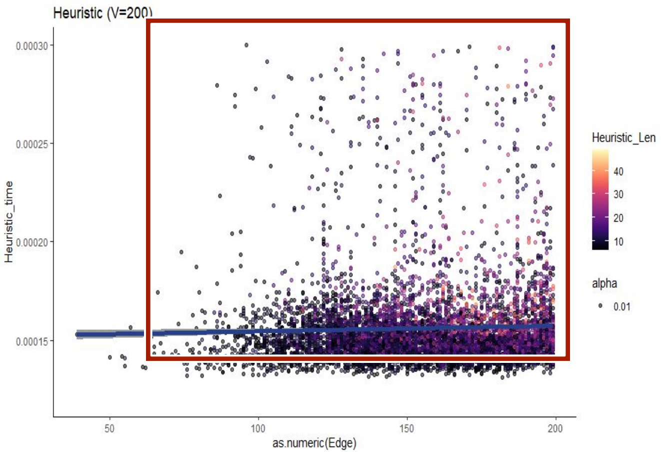
Vertex

- Change in running time when Edge kept constant and Vertex is increasing

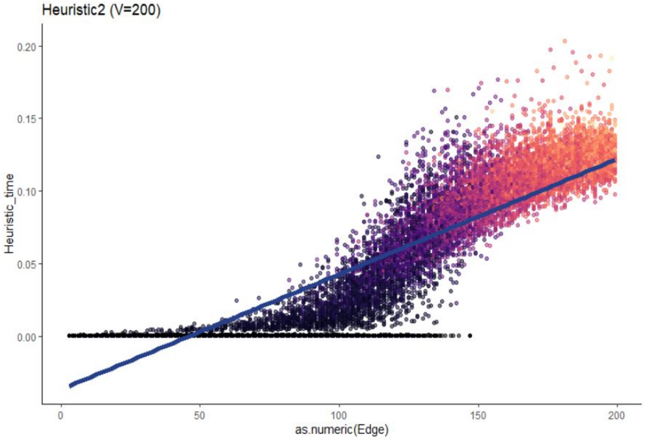


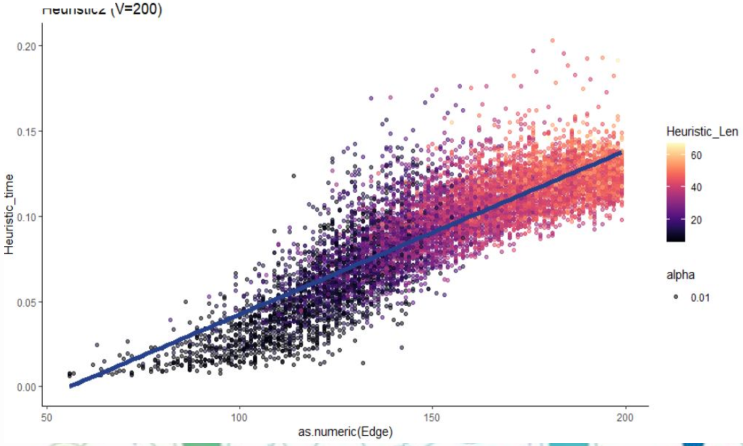
Edge



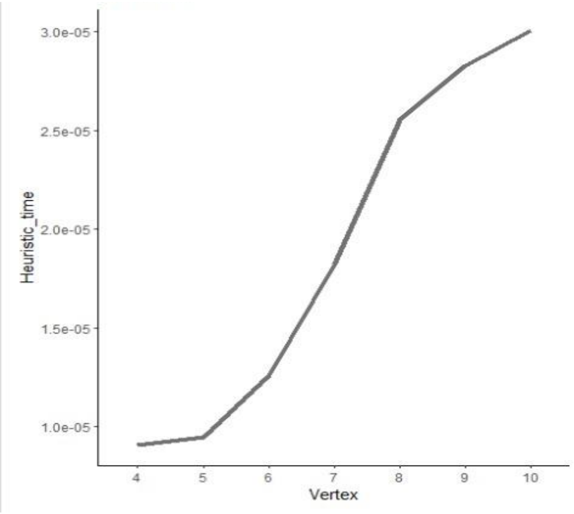


- ❖ Change in running time when Vertex kept constant and Edge is increasing
- ❖ Difference, when cases without a cycle eliminated

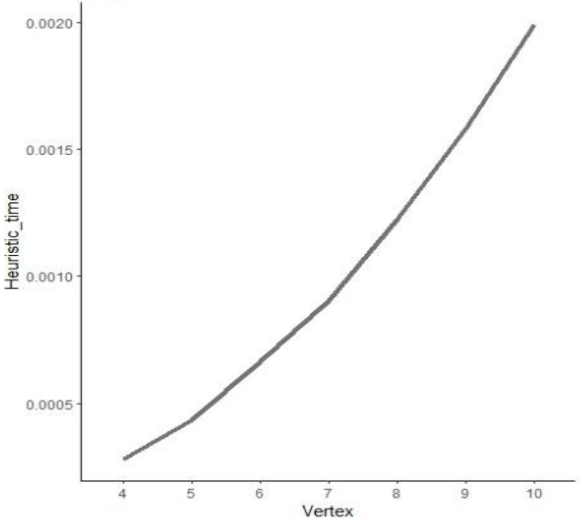




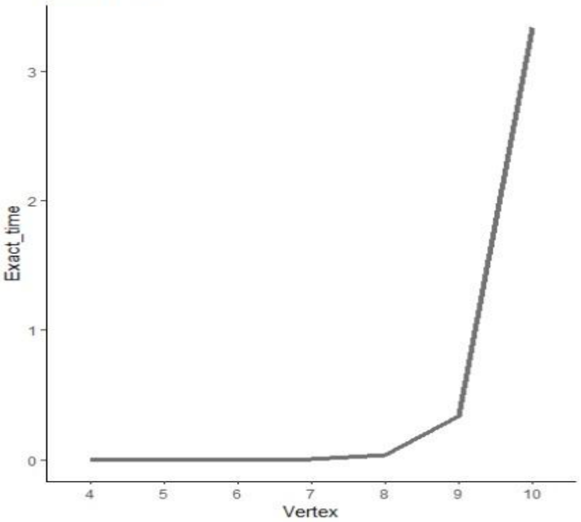
Method 1



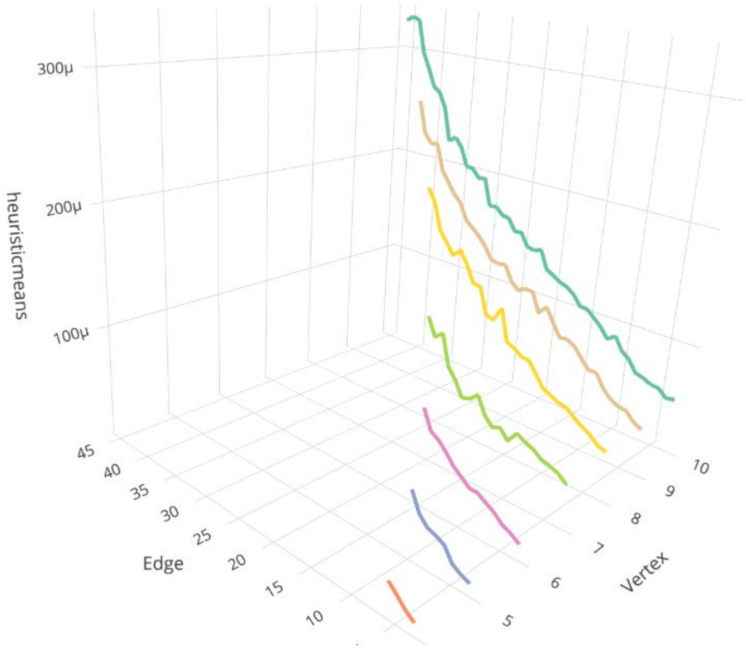
Method 2

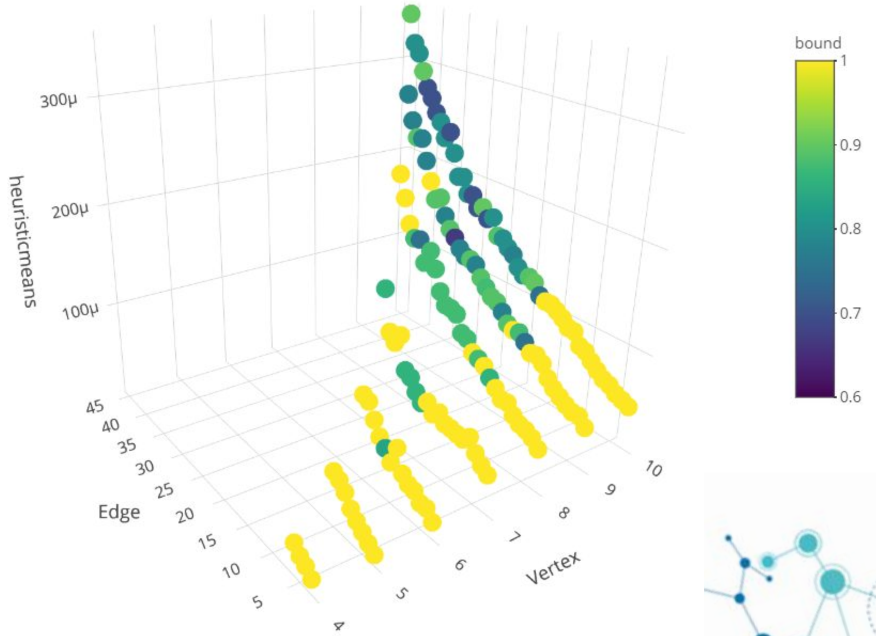


Brute Force



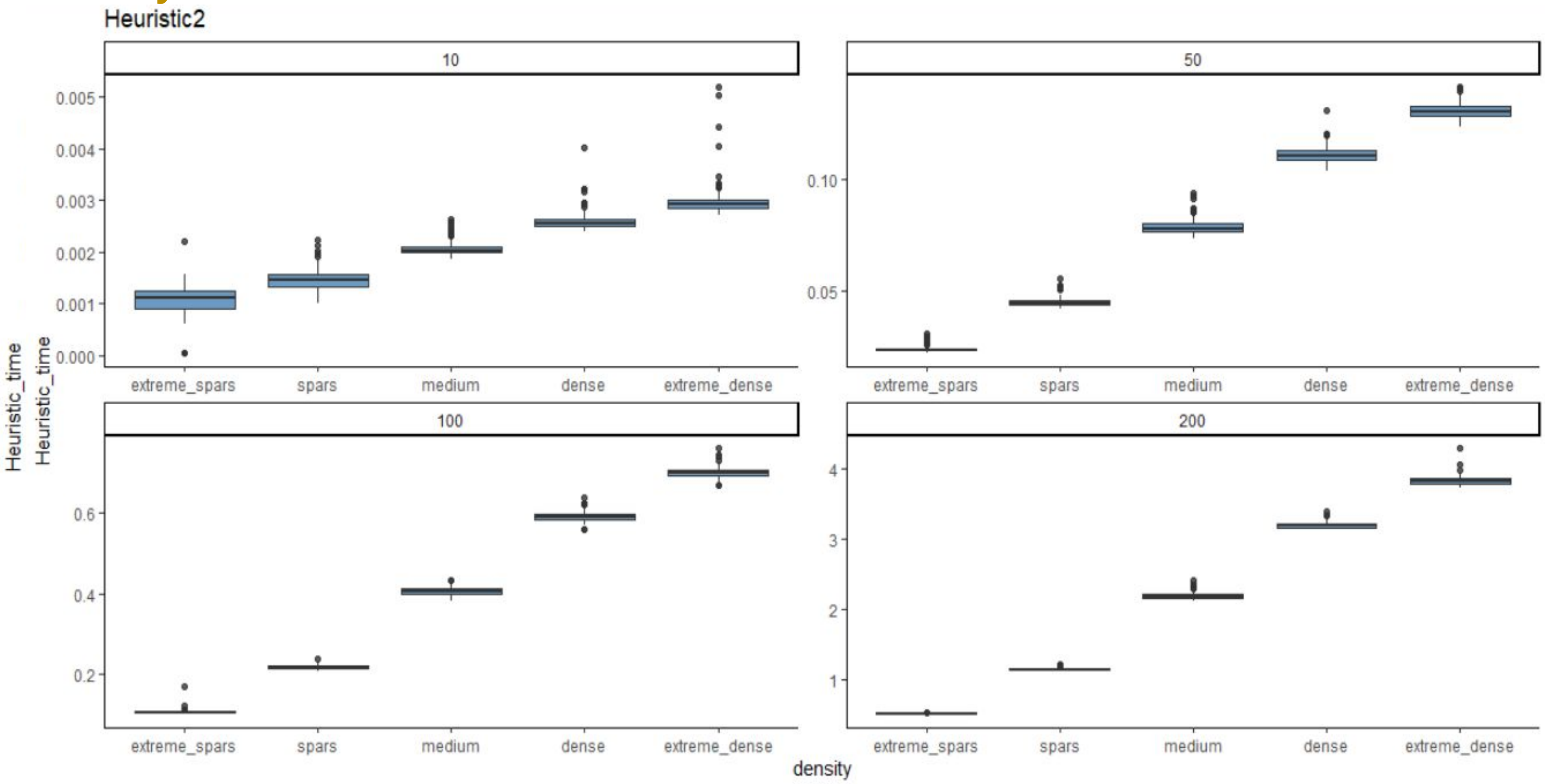
Edge ∧ Vertex





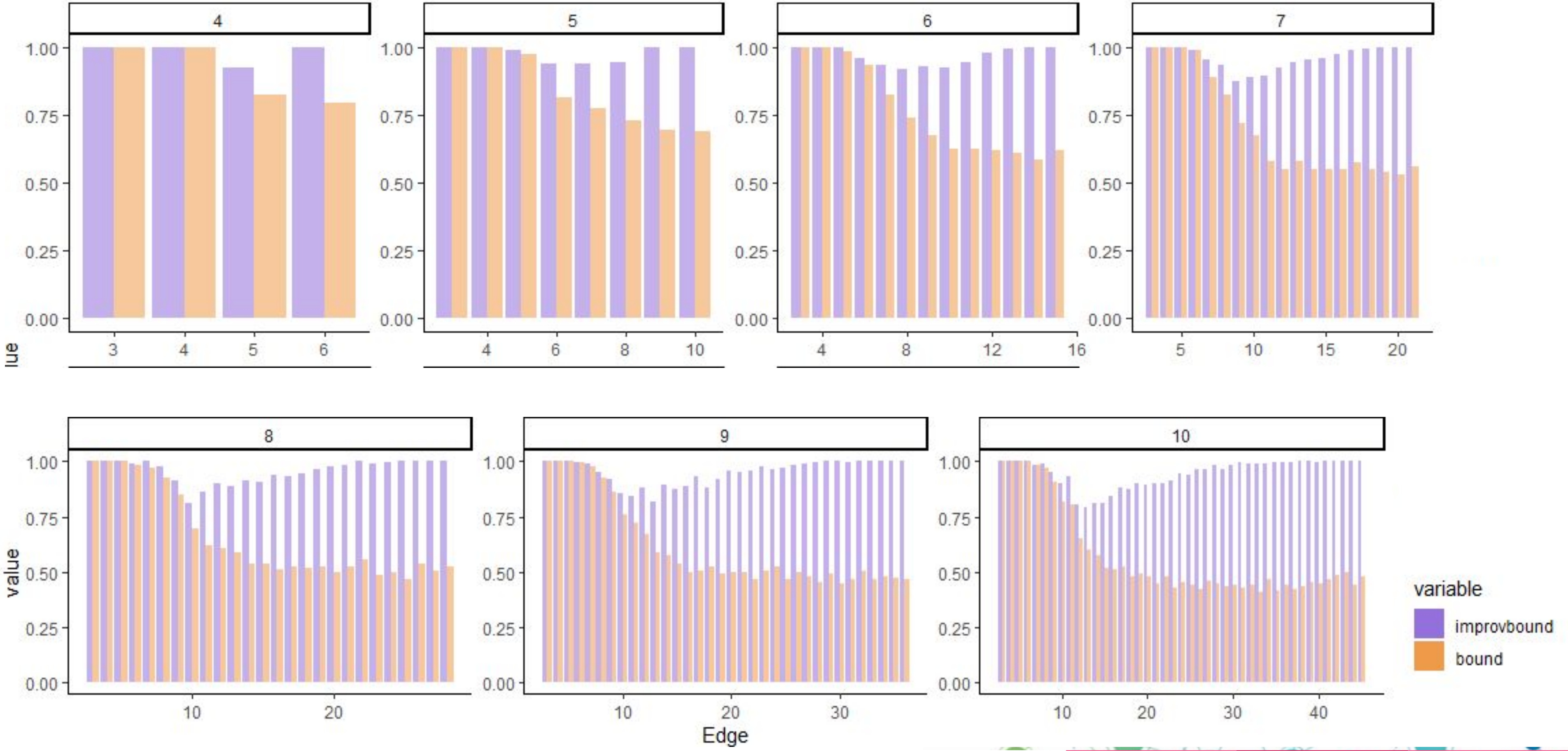
[3d html]

Density


Heuristic2
10
50
100
200
Heuristic_time
density
extreme_spars
spars
medium
dense
extreme_dense
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

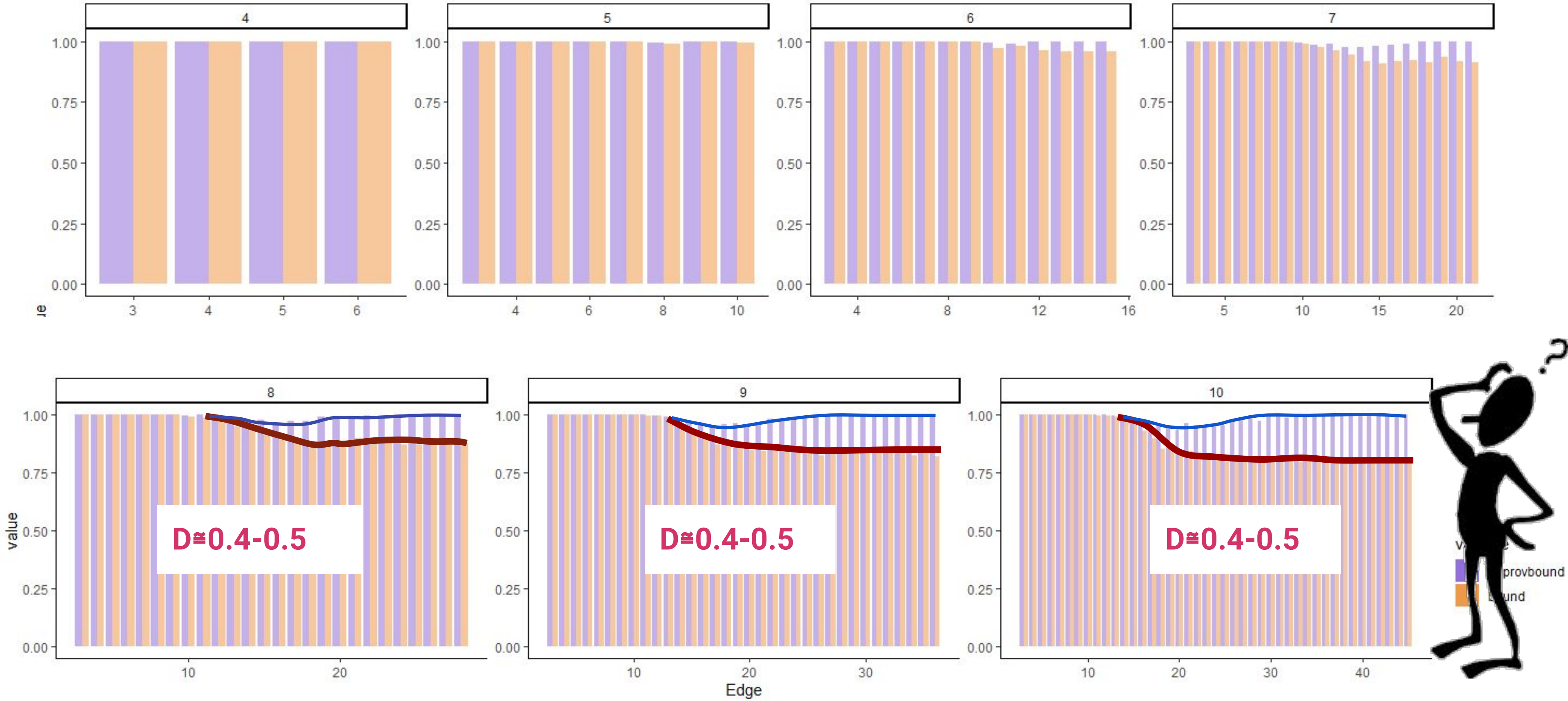
Ratio Bound

Method 1



Ratio Bound

Method 2



Correctness

```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
```

```
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
------------------------------------------------
```

Testing

- We used the black box test to test the heuristic algorithm.
- We generated random graphs for testing and experimental analysis using the algorithm.
- We ran random tests for different vertex and edge numbers for the second heuristic algorithm and ultimate improvement algorithm.

```
#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
```

Black Box

```
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
---------------------------------------------
```

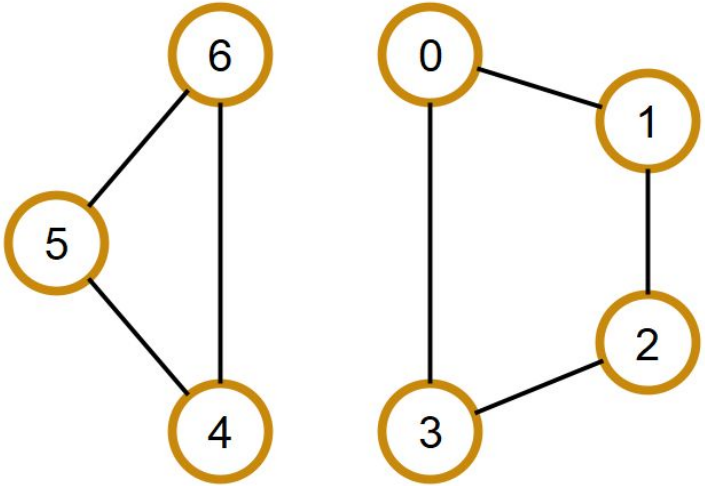
```
Test is for vertex number = 300 edge number = 5000
repeated for every case for= 100 times.
---------------------------------------------
heuristic2 + ultimate improvement
Correctness = % 100.0
---------------------------------------------
```

```
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
---------------------------------------------
```

Black Box

Bipartite graph with cycles

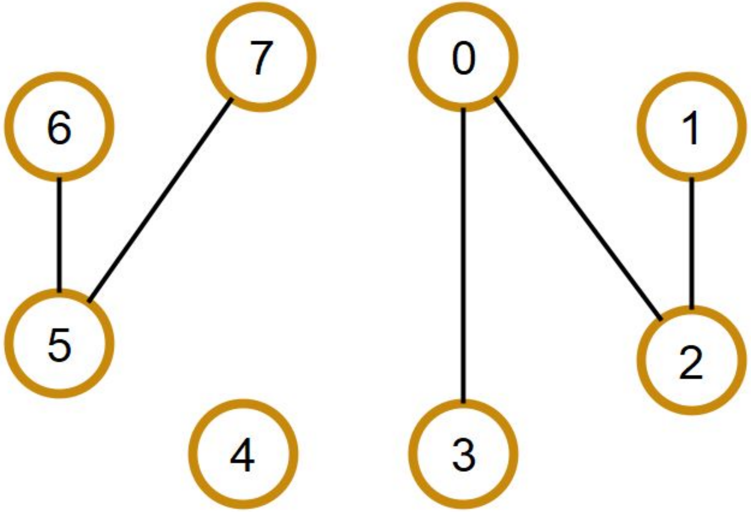


SUCCESS:

```
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
```

Black Box

Bipartite graph without a cycles

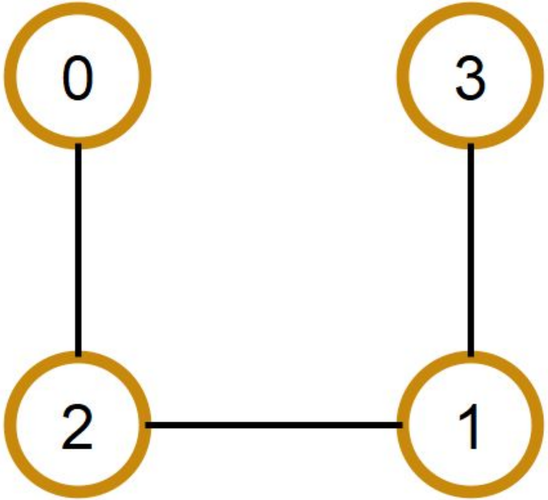


SUCCESS:

```
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.
```

Black Box

Graph with no cycles

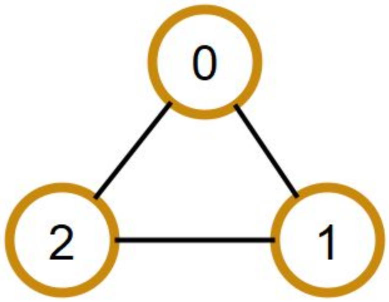


SUCCESS:

```
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.
```

Black Box

Minimum case for a cyclic graph

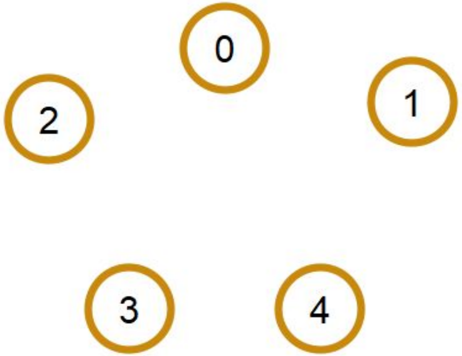


SUCCESS:

```
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
```

Black Box

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.
```

Discussion

- ❖ Longest Simple Circuit has proven to be a NP-Complete problem.
- ❖ It is better to use an heuristic algorithm which has polynomial complexity for NP_hard problems. However, these heuristic algorithms cannot find the best solution for every case.
- ❖ 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 their running time in practice is different.
- ❖ 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 ratio bound of the both heuristic algorithms.