

# COT5405 - Analysis Of Algorithm

## Assignment 1 - Greedy Algorithms

Submitted By: Karan Asthana



Computer and Information Science and Engineering Department  
Fall 2021  
University of Florida  
October 11, 2021

# 1 Cycle Finding in an Undirected Graph

## 1.1 Algorithm Pseudo Code

The basic idea of the algorithm is to traverse the undirected Graph in a Depth-First Search (DFS) and then keep a track of the nodes/vertices that we've already visited. As soon as a node appears twice in the visited list, it is because of a cycle present in the graph. That is, there are two different ways of reaching a Node  $n$ . We take extra care in the algorithm to pass on the parent of the current node to the next execution of the recursive DFS function (so as to prevent traversing in the backward direction, which would essentially lead to infinite loops or faulty cycle detection).

We start from the initial node and traverse all the edges connected with it recursively. If we, at any time encounter the same node again, we track it and return the cycle according to the traversed nodes in this execution.

*The pseudo code for the algorithm is stated on the next page*

---

**Algorithm 1** FindCycle(G)

---

```
1: function FINDCYCLE(graph)
2:   vSet  $\leftarrow$  graph.vertices(), visited  $\leftarrow$  [], cycle  $\leftarrow$  [], stack  $\leftarrow$  newStack()
3:   for Vertex v in vSet do                                      $\triangleright$  this is to incorporate non-connected graphs
4:     if visited.containsv then continue
5:     end if
6:     stack.clear()
7:     if DFS(v, parent) then                                      $\triangleright$  parent is null or -1 at start
8:       start  $\leftarrow$  stack.top
9:       cycle.append(start)
10:      while stack  $\neq$  Empty do
11:        if stack.top==start then
12:          break
13:        end if
14:        cycle.append(stack.top)
15:      end while
16:      cycle.append(start)
17:      return cycle
18:    end if
19:  end for
20: end function
```

```
1: function DFS(v, p)                                      $\triangleright$  v is the vertex                                      $\triangleright$  p is the parent of vertex v
2:   visited.append(v)
3:   stack.append(v)
4:   for NeighborVertex n in v.neighbors() do
5:     if (not visited.contains(n) AND DFS(n, vertex)) then
6:       return true
7:     end if
8:     if n  $\neq$  parent then                                      $\triangleright$  Adding n to the stack, if the previous node was not n
9:       stack.append(n)
10:      return true
11:    end if
12:  end for
13:  stack.pop()
14:  return false
15: end function
```

---

## 1.2 Proof of Correctness

**Lemma:** If a graph  $G$  contains a cycle. Then during graph traversal, we visit the same vertex twice, it means that there is at least one cycle in the graph.

*Proof. Using Contradiction:*

Let us assume that we have a cyclic graph  $G$  with vertices  $v_1, v_2, \dots, v_n$  and edges  $e_1, e_2, \dots, e_n$ .

Let us assume that the graph  $G$  has 1 cycle from the vertices  $v_j$  to  $v_k$  and during traversal  $v_j$  is visited only once.

Start traversing the graph starting from the parent vertex of  $v_j$ .  
Continue traversing till all the nodes have been traversed.

But on encountering the cycle, the edges again lead to the same vertex  $v_j$  (after the first traversal).

This is a contradiction, since initially we had assumed that every node is visited only once. But in order for a cycle traversal to complete, we have to visit the same vertex more than once. □

**Thus, a graph  $G$  that contains at least a cycle, visits at least one node twice.**

## 1.3 Algorithm Running Time

The worst case time complexity is  $O(V+E)$ , where  $V$  is the number of vertices and  $E$  is the number of Edges.

*Proof.* According to the pseudo code mentioned above, in the worst case, we are traversing **all** the vertices once using **all** the edges twice (since in Undirected graphs, every edge can be considered twice during traversal, which is ignored via the parent logic in our algorithm). So, intuitively our time complexity is in the order of the sum of number of vertices and the number of edges

- In the pseudo code, we find all the vertices and start traversals from these nodes (if not visited before) (this is equivalent to a worst-case  $O(V)$  time)
- We then traverse all the edges connected to a vertex  $v$ , using the Depth First logic. That is, we take one vertex and then using one of its edge, go to the next vertex and then, so on. Thereby, we traverse all the edges at max twice, therefore, it is equivalent to  $O(2E)$ .
- Other operations, such as finding max in a cycle are always of a lower order, usually taking constant time.
- Therefore, the effective worst case complexity of the algorithm leads to  $O(V + 2E + V + C)$ , which is equivalent to  $O(V+E)$  □

## 1.4 Implementation

### 1.4.1 Graph Generator

For the implementation of the algorithm, a graph generator library, **jgrapht** (written in Java) was used.

The number of vertices in the order of  $10^6$  were randomly selected.

The number of edges in the order of  $2 * numVertices$  were randomly selected.

The combination of these number of edges and vertices was then input to the graph generator library, which then returned a Graph object with the specified vertices and unweighted edges.

Attached image Figure 1.1, shows a screenshot of the test cases output as seen on the terminal while execution.

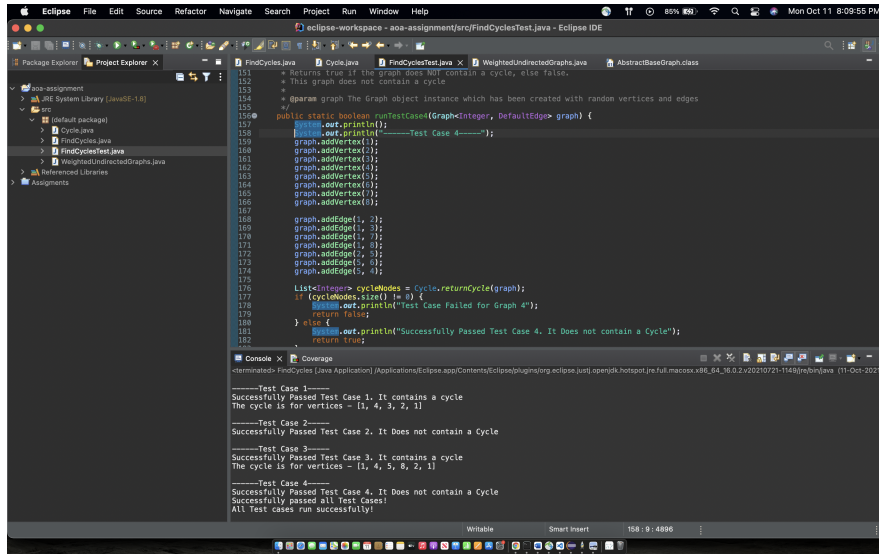


Figure 1.1

Attached Figures 1.2, Figure 1.3, Figure 1.4 and Figure 1.5, shows the test case inputs that were tested by hard-coding their values.

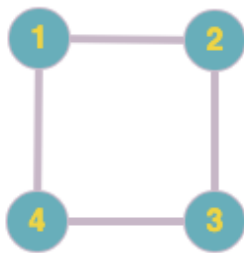


Figure 1.2 (Test Case 1 - Contains Cycle - 1,2,3,4,1)



Figure 1.3 (Test Case 2 - Does not Contain Cycle)

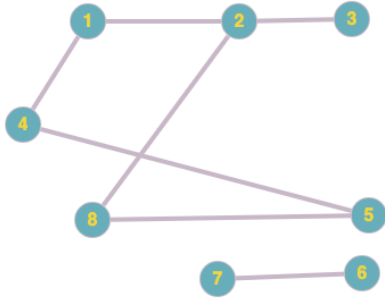


Figure 1.4 (Test Case 3 - Contains Cycle - 1,2,8,5,4,1)

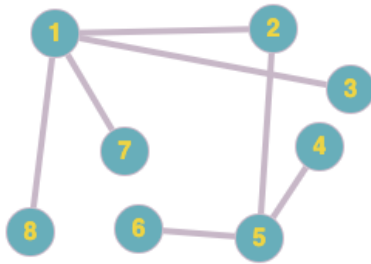


Figure 1.5 (Test Case 4 - Does not Contain Cycle)

### 1.4.3 Test for increasing graph sizes

Using a for loop, the number of vertices were randomly selected, leveraging the random function for a fraction of 1 million nodes in one execution.

---

```

1: for  $i$  in (1,2000) do
2:    $numVertices \leftarrow (random() * 10^6)$ 
3:    $numEdges \leftarrow (random() * 2 * numVertices)$ 
4:    $jgraphT.generateGraph(numVertices, numEdges)$ 
5:   return  $graph$ 
6: end for

```

---

### 1.4.4 Run time vs number of nodes Plot

Attached image Figure 1.6, shows a screenshot of the run time vs number of nodes plotted on a graph.

The x-axis contains the number of nodes (with respect to 1million)

The y-axis contains the time of execution (in nano seconds, with respect to 1second)

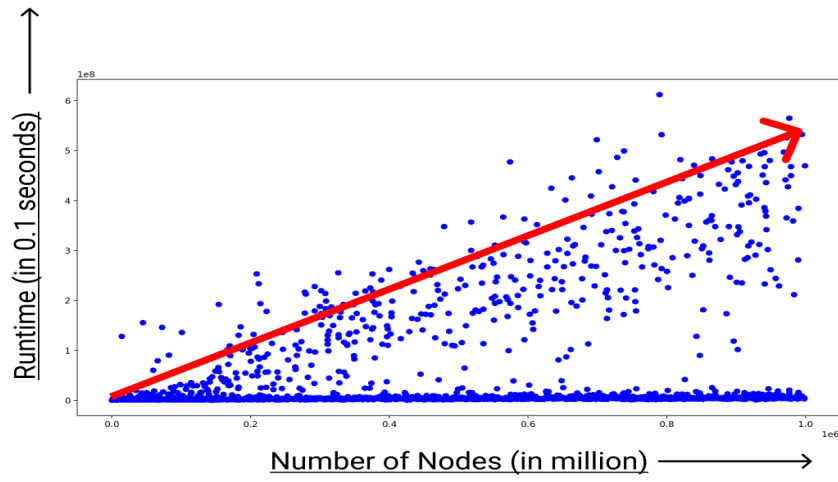


Figure 1.6

## 2 Minimum Spanning Tree for "sparse" graphs

### 2.1 Algorithm PseudoCode

The basic idea of the algorithm is to leverage a modified version of the cycleFinding algorithm (ref: Algorithm 1), which returns the heaviest in the first cycle that is found in the graph.

This process is repeated until there are no cycles in the weighted graph. (which is also

$numCycles \leftarrow (numEdges - numVertices + 1)$  )

*The pseudo code for the algorithm is stated below*

---

**Algorithm 2** FindMST(G)

---

```
1: function FINDMST(graph)
2:   vSet  $\leftarrow$  graph.vertices(), eSet  $\leftarrow$  graph.edges()
3:   numVertices  $\leftarrow$  vSet.size(), numEdges  $\leftarrow$  eSet.size()
4:   for i in (numVertices + 1 - numEdges) do           ▷ this is to remove the extra edges
5:     cycle  $\leftarrow$  FINDCYCLE(graph)
6:     heaviestEdge  $\leftarrow$  cycle.heaviestEdge           ▷ this returns the cycle as well as the heaviest edge
       in it
7:     graph  $\leftarrow$  graph.delete(heaviestEdge)
8:   end for
9:   return graph
10: end function
```

---

### 2.2 Proof of Correctness

**Lemma:** A Minimum Spanning Tree with  $n$  vertices has  $n-1$  edges.

*Proof.* Using Mathematical Induction

**Base Case ( $n=1$ ):**

Let us assume that we have a tree with only 1 node and the number of edges in this tree = 0 (i.e.  $n-1$ ).

**Case ( $n=2$ ):**

Let us assume that we have a tree with only 2 nodes and the number of edges in this tree =  $num(0) + 1$  (since it has to connect to only 1 existing node so as to not form any cycle)

Thus,  $n-1 = 1$  edges.

**Case ( $n=n$ ):**

Let's assume that the inductive step is true, i.e. for  $n$  nodes, number of edges =  $n-1$  edges.

**Case ( $n=n+1$ ):**

The number of edges will be  $(n-1) +$  number of edges required to connect to the  $n$  node tree

Thus, the total number of edges will be  $(n-1) + 1 = n$  edges.

Thus, we can say that Case( $n+1$ ) is true.

**Hence, using the Principle of Mathematical Induction, it is proved that for  $n$  vertices, a minimum spanning tree will contain  $(n-1)$  edges.**

In our algorithm's pseudo code, we are reducing the number of edges to  $n-1$  and hence, we can conclude to say that our algorithm is correct.

□

## 2.3 Algorithm Running Time

The worst case time complexity is  $O(V)$ , where  $V$  is the number of vertices.

*Proof.* According to the given problem statement, in the worst case, we have  $(V+8)$  edges, i.e. 9 cycles in the graph. So, to form an MST, we need to obsolete out all the cycles in the graph (by removing their heaviest edge). By (1.3), we know that finding a Cycle takes  $O(V+E)$  worst-case time. Since, we have a maximum of 9 cycles, we will take

$9 * (\text{find\_heaviest\_edge\_in\_a\_cycle}) * (\text{removal\_of\_heaviest\_edge})$   
time to remove all the cycles and thus, form the MST.

- Using a modified version of Algorithm 1, getting the heaviest edge of a cycle from the graph takes  $O(V+E)$  time, where  $E = (V+9)$ , i.e.  $O(V+V+9)$ , i.e.  $O(2V+9)$
- Doing the above operation for all the extra edges (9), we get the overall run-time as  $9 * O(2V+9)$ , i.e.  $O(18V + 81)$ , which is effectively  $O(V)$ .

□

## 2.4 Implementation

### 2.4.1 Graph Generator

For the implementation of the algorithm, a graph generator library, **jgrapht** (written in Java) was used.

A weighted Tree was initially generated for a random number of vertices (it will have exactly 1 lesser edge than the number of vertices)

After generating a weighted tree, it was converted into a graph object and then, random number of edges (between 0 and 9) edges were added to the tree with random weights.

The number of vertices in the order of  $10^6$  were randomly selected.

The number of edges (between  $numVertices - 1$  and  $numVertices + 8$ ) was randomly selected.

### 2.4.2 Test code to validate algorithm correctness

For testing the correctness of the algorithm, random small test cases were written whose result was already known. The results of the algorithm were then compared to the already known results.

For testing the MST, we have used the resultant edges of the MST as a way of testing correctness. (If the edges in the resultant MST are correct, it was an MST)

Attached image Figure 2.1, shows a screenshot of the test cases output as seen on the terminal while execution.

The screenshot shows the Eclipse IDE with a Java project. The code in the editor defines a graph with 10 vertices and 11 edges, each with a weight. The edges are: (1,2) weight 2, (2,3) weight 3, (3,4) weight 4, (4,5) weight 5, (5,6) weight 6, (6,7) weight 7, (7,8) weight 8, (8,9) weight 9, (9,10) weight 10, and (1,10) weight 1. The code then uses the jgrapht library to find the MST and compare it with the expected edges: (1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), and (9,10). The console output shows that all test cases passed successfully.

```
graph.addVertex(i);
182
183
184
185 DefaultWeightedEdge e1 = graph.addEdge(1, 2);
186 DefaultWeightedEdge e2 = graph.addEdge(2, 3);
187 DefaultWeightedEdge e3 = graph.addEdge(3, 4);
188 DefaultWeightedEdge e4 = graph.addEdge(4, 5);
189 DefaultWeightedEdge e5 = graph.addEdge(5, 6);
190 DefaultWeightedEdge e6 = graph.addEdge(6, 7);
191 DefaultWeightedEdge e7 = graph.addEdge(7, 8);
192 DefaultWeightedEdge e8 = graph.addEdge(8, 9);
193 DefaultWeightedEdge e9 = graph.addEdge(9, 10);
194 DefaultWeightedEdge e10 = graph.addEdge(1, 10);
195
196 Set<DefaultWeightedEdge> edges = graph.edgeSet();
197
198 int i=0;
199 for (DefaultWeightedEdge e: edges) {
200     graph.setEdgeWeight(e, i++);
201 }
202
203 // Expected Output
204 Graph<Integer, DefaultWeightedEdge> graph2 = graph;
205 graph2.removeEdge(e1); // Because it is the heaviest edge
206 graph2.removeEdge(e10); // Because it is the 2nd most heavy edge
207 graph2.removeEdge(e8); // Because it is the 3rd most heavy edge
208 graph2.removeEdge(e9); // Because it is the 4th most heavy edge
209 Set<DefaultWeightedEdge> expectedEdges = graph2.edgeSet();
210
211 Set<DefaultWeightedEdge> outputEdges = WeightedUndirectedGraphs.performMSTFind(graph, 4, 5);
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
-----Test Case 1-----
Successfully Passed Test Case 1.
The MST has the edges - [(1 : 2), (2 : 3), (3 : 4)]

-----Test Case 2-----
Successfully Passed Test Case 2.
The MST has the edges - [(1 : 2), (2 : 3), (3 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 8)]
Successfully passed both Test Cases!
All Test cases run successfully!
```



Figure 2.1

Attached Figures 2.2 and Figure 2.3, shows the test case inputs that were tested by hard-coding their values.

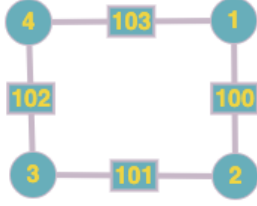


Figure 2.2 (Test Case 1 - Contains MST with edges -  $[(1 : 2), (2 : 3), (3 : 4)]$ )

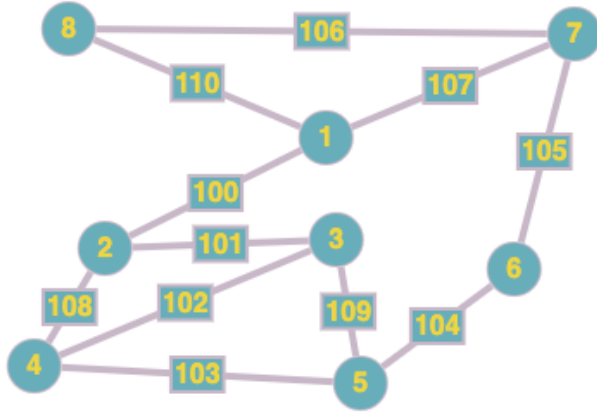


Figure 2.3 (Test Case 2 - Contains MST with edges -  $[(1 : 2), (2 : 3), (3 : 4), (4 : 5), (5 : 6), (6 : 7), (7 : 8)]$ )

#### 2.4.3 Test for increasing graph sizes

Using a for loop, the number of vertices were randomly selected, leveraging the random function for a fraction of 1 million nodes in one execution.

---

```

1: for  $i$  in  $(1, 2000)$  do
2:    $numVertices \leftarrow (random() * 10^6)$ 
3:    $extraEdges \leftarrow (random() * 9)$ 
4:    $graph \leftarrow jgraphT.generateWeightedTree(numVertices)$ 
5:   for  $j$  in  $extraEdges$  do
6:      $graph.addEdge(randomWeight)$ 
7:   end for
8:   return  $graph$ 
9: end for

```

---

#### 2.4.4 Run time vs number of nodes Plot

Attached image Figure 2.6, shows a screenshot of the run time vs number of nodes plotted on a graph.

The x-axis contains the number of nodes (with respect to 1million)

The y-axis contains the time of execution (in nano seconds, with respect to 1second)

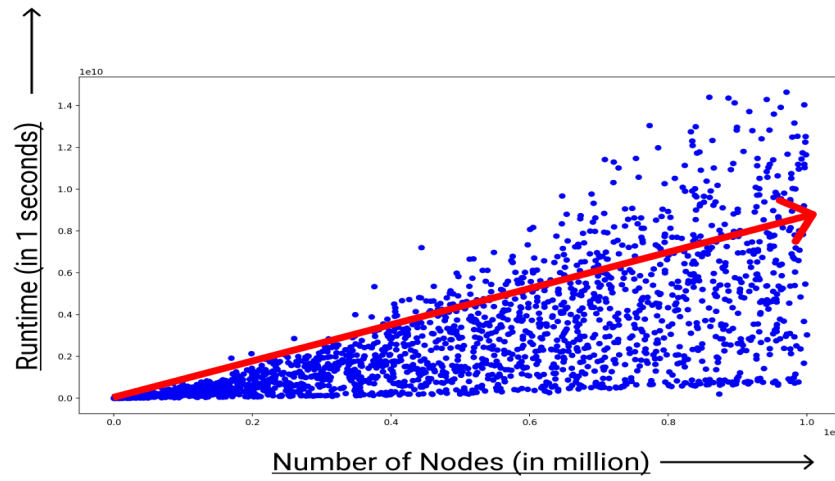


Figure 2.6 - time VS num(nodes) plot

## 3 Solution Specifics

### 3.1 Installation and Setup

#### Installations

- Install Java SDK 1.8
- Download jar file for the Java Library, jgrapht
- Download an IDE, Eclipse for running the code
- Add the jar file for jgrapht-core in the eclipse project
- Install Python (for graph plotting of run-time vs number of nodes)

#### Assignment Question 1

- For running the Assignment Question 1, create a Run configuration that runs the code starting from the Java file, FindCycles.java
- This file internally runs a loop for a large number of times (currently set to 100) and finds out a cycle in a randomly generated graph.
- If there is a cycle present in the graph, the vertices of the graph are printed on the console output.
- If there is no cycle present in the graph, a message is printed for the same.

#### Assignment Question 2

- For running the Assignment Question 2, create a Run configuration that runs the code starting from the Java file, "FindMST.java" (earlier named as WeightedUndirectedGraph.java)
- This file internally runs a loop for a large number of times (currently set to 100) and finds out the MST in a randomly generated graph.
- The randomly generated graph is first created by creating a random weighted tree and then manually adding up to 9 more weighted edges.
- The corresponding edges contained in the MST are then printed on the console output.

### 3.2 Testing

- Separate files named FindCyclesTest.java and MSTTest.java have been created for specifically testing the correctness of the algorithms.
- For testing of the first question of the Assignment, the initial block of lines in the main functions of the "**FindCycles.java**" and the "**FindMST.java**" files have to be uncommented presently.
- The file "**FindCyclesTest.java**" contained 4 hard-coded different test cases, 2 of which had no cycles in the graph. While 2 of them contained cycles.
- The runTestCases function was run, which returned a boolean value for passing or failure of test cases. Along with relevant console messages.
- These test cases have been explained above with images.
- The file "**MSTTest.java**" contained 2 hard-coded different test cases.

- Test Case 1 contained a graph with 4 nodes and 5 edges with varying weights. The output returned was compared to be the exact 3 edges which had the least weights and formed a connected graph.
- Test Case 2 contained a graph with 8 nodes and 11 edges with varying weights. The output returned was compared to be the exact 7 edges which had the least weights and formed a connected graph.
- The runTestCases function was run, which returned a boolean value for passing or failure of test cases. Along with relevant console messages.
- These test cases have been explained above with images.

### **Graph Plot of run-time**

- To map the run time as a function of the number of nodes, the run time for every execution was noted and a CSV was created which mapped the number of nodes, number of edges and their run-time (in nanoseconds).
- Using Python, and the matplotlib library, a graph was generated between the number of nodes and the runtime of each execution.