

## ▼ Problem 1.

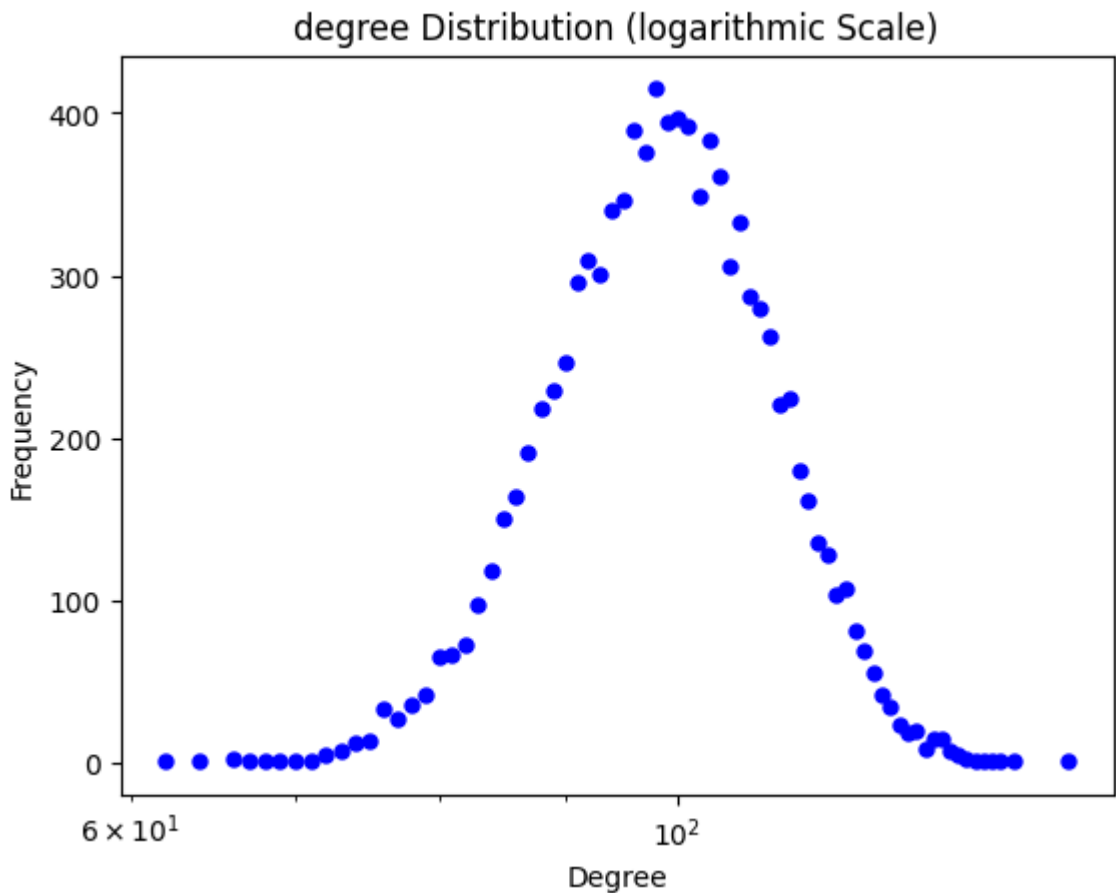
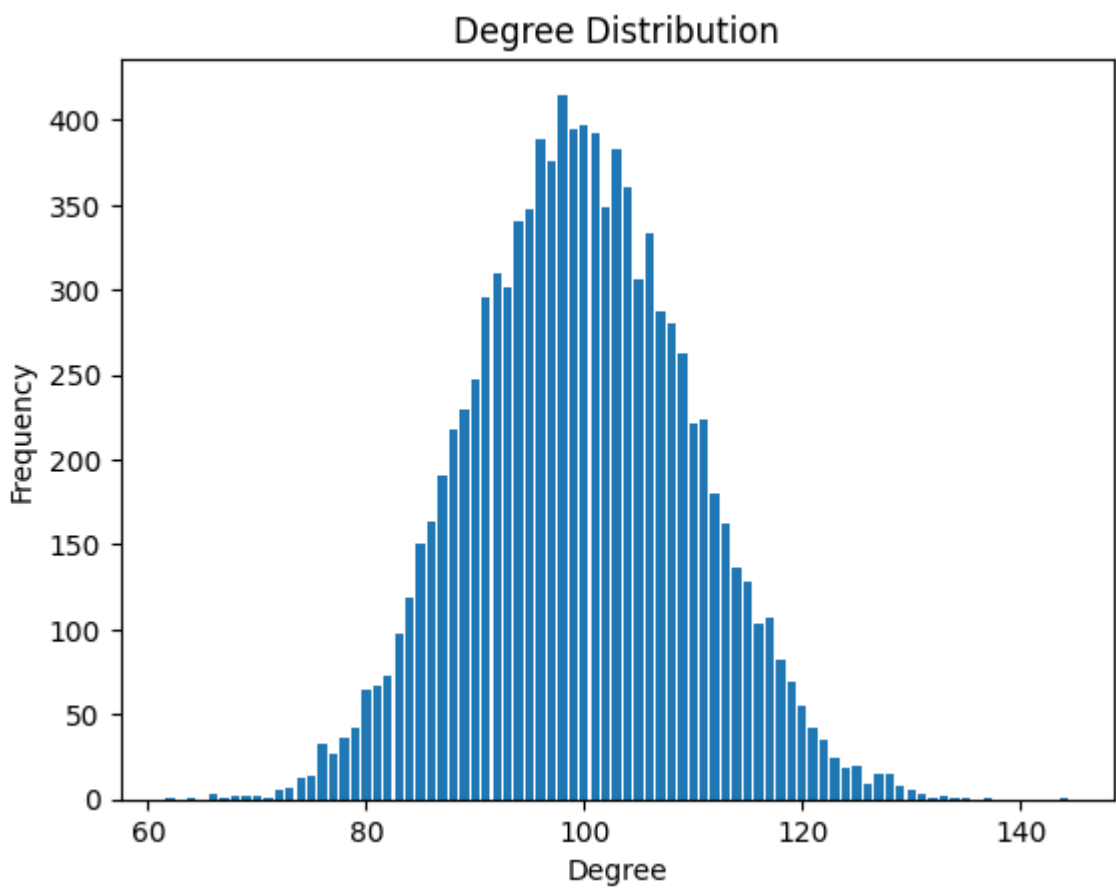
### 1. Erdos-Renyi graph

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import random as rand
4 import multiprocessing
5 from tqdm import tqdm

1 class Graph:
2     def __init__(self):
3         self.nodes = {}
4         self.edges = {}
5
6     def add_node(self, node):
7         if node not in self.nodes:
8             self.nodes[node] = set()
9
10    def add_edge(self, node1, node2):
11        if node1 in self.nodes and node2 in self.nodes:
12            self.edges[(node1, node2)] = 1
13            self.edges[(node2, node1)] = 1
14            self.nodes[node1].add(node2)
15            self.nodes[node2].add(node1)
16
17    '''
18    The easiest property to study is the degree distribution. A given node v is incident with
19    n - 1 potential edges, and each of them exists with probability p independently of each other.
20    '''
21    '''
22    in Section 3.2 we mentioned
23    that in social networks we observe heavy-tailed degree distributions. This shows one of the
24    main problems of the Erdős-Rényi random graph model for modeling social networks. In other
25    words in the Erdős-Rényi model the nodes are very similar
26    '''
27
28
29    def compute_degree_distribution(self):
30        degree_counts = {}
31        for node in self.nodes:
32            degree = len(self.nodes[node])
33            degree_counts[degree] = degree_counts.get(degree, 0) + 1
34        return degree_counts
35
36    def compute_diameter(self):
37
38    '''
39    def compute_diameter(self):
40        # Use breadth-first search to find the maximum shortest path length
41        max_path_length = 0
42        for node in self.nodes:
43            queue = [(node, 0)]
44            visited = set([node])
45
46            while queue:
47                current_node, path_length = queue.pop(0)
48                max_path_length = max(max_path_length, path_length)
49
50                for neighbor in self.nodes[current_node]:
51                    if neighbor not in visited:
52                        visited.add(neighbor)
53                        queue.append((neighbor, path_length + 1))
54
55        return max_path_length
56    '''
57    def compute_clustering_coefficient(self):
58        if len(self.nodes) == 0:
59            return 0.0 # or return a default value
60
61        pool = multiprocessing.Pool()
62
63        total_clustering_coefficient = 0
64
65        nodes = list(self.nodes.keys())
66
67        # Use multiprocessing.Pool.map to apply the compute_local_clustering_coefficient function to each node
68        local_cc = pool.map(self.compute_local_clustering_coefficient, nodes)
69
70        # Compute the total clustering coefficient
71        total_clustering_coefficient = sum(local_cc)
72
73        return total_clustering_coefficient / len(self.nodes)
74
75    def compute_local_clustering_coefficient(self, node):
76        neighbors = self.nodes[node]
77        num_neighbors = len(neighbors)
78        if num_neighbors < 2:
79            return 0.0
80
81        num_edges = 0
82        for neighbor1 in neighbors:
83            for neighbor2 in neighbors:
84                if neighbor1 != neighbor2 and neighbor1 in self.nodes[neighbor2]:
85                    num_edges += 1
86
87        local_clustering_coefficient = num_edges / (num_neighbors * (num_neighbors - 1))
88        return local_clustering_coefficient
89
90    def plot_degree_distribution(self):
91        degree_dist = self.compute_degree_distribution()
92        degrees = list(degree_dist.keys())
93        frequencies = list(degree_dist.values())
94
95        plt.bar(degrees, frequencies)
96        plt.xlabel("Degree")
97        plt.ylabel("Frequency")
98        plt.title("Degree Distribution")
99        plt.show()
```

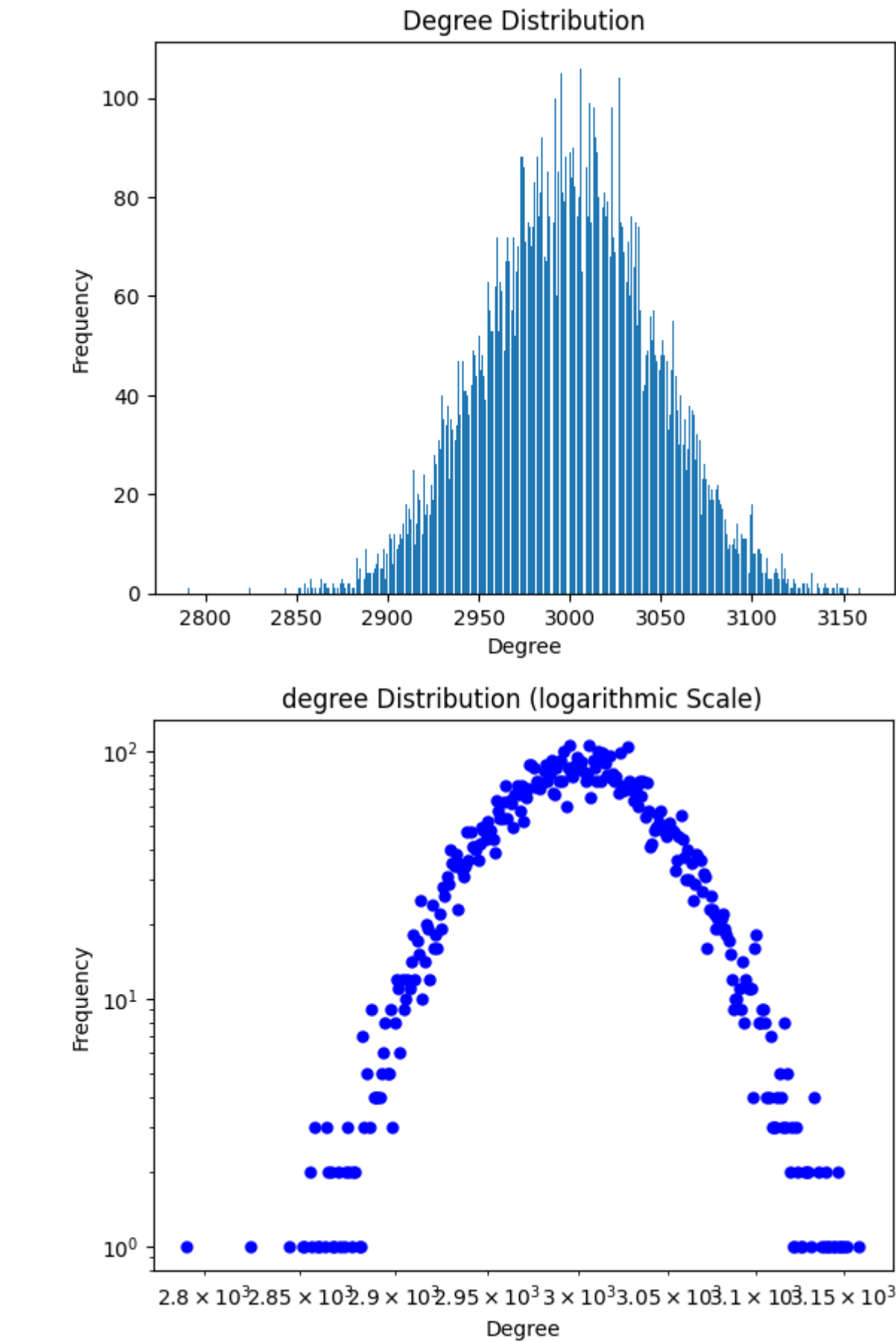
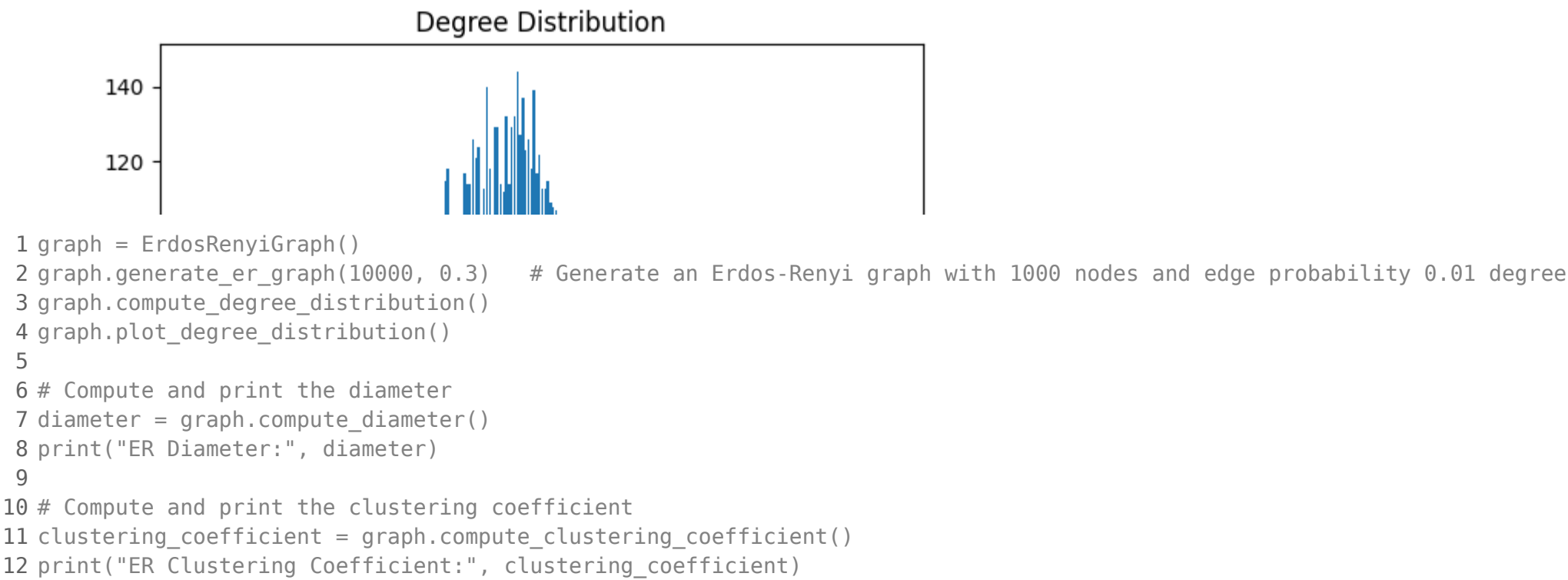
```
100
101     plt.loglog(degrees, frequencies, 'bo', markersize=5)
102     plt.xlabel("Degree")
103     plt.ylabel("Frequency")
104     plt.title("degree Distribution (logarithmic Scale)")
105     plt.show()
106
107     def plot_graph(self):
108         G = nx.Graph()
109         G.add_nodes_from(self.nodes.keys())
110         G.add_edges_from(self.edges.keys())
111
112         pos = nx.spring_layout(G)
113         nx.draw(G, pos, with_labels=True)
114         plt.show()
115
116 class ErdosRenyiGraph(Graph):
117     def generate_er_graph(self, n, p):
118         self.nodes = {}
119         self.edges = {}
120
121         self.add_node(0) # Add the first node
122
123         for i in range(1, n):
124             self.add_node(i) # Add nodes from 1 to n-1
125
126             for j in range(i):
127                 if rand.random() < p: # Add an edge with probability p
128                     self.add_edge(i, j)
129
130 class WattsStrogatzGraph(Graph):
131     def generate_ws_graph(self, n, k, beta):
132         self.nodes = {}
133         self.edges = {}
134
135         for node in range(n):
136             self.add_node(node)
137
138         # Create initial ring lattice
139         for node in self.nodes:
140             for i in range(1, k // 2 + 1):
141                 neighbor = (node + i) % n
142                 self.add_edge(node, neighbor)
143
144         # Rewire edges
145         for node in self.nodes:
146             for i in range(1, k // 2 + 1):
147                 if rand.random() < beta:
148                     neighbor = (node + i) % n
149                     self.rewire_edge(node, neighbor)
150
151     def rewire_edge(self, node, neighbor):
152         # Remove existing edge
153         self.remove_edge(node, neighbor)
154
155         # Select a random node to connect
156         new_neighbor = rand.choice(list(self.nodes.keys()))
157
158         # Make sure the new neighbor is not the same as the original node or an existing neighbor
159         while new_neighbor == node or new_neighbor == neighbor or new_neighbor in self.nodes[node]:
160             new_neighbor = rand.choice(list(self.nodes.keys()))
161
162         # Add the new edge
163         self.add_edge(node, new_neighbor)
164
165     def remove_edge(self, node1, node2):
166         if (node1, node2) in self.edges:
167             del self.edges[(node1, node2)]
168             del self.edges[(node2, node1)]
169             self.nodes[node1].remove(node2)
170             self.nodes[node2].remove(node1)
171
172
173 class BarabasiAlbertGraph(Graph):
174     def generate_ba_graph(self, n, l):
175         self.nodes = {}
176         self.edges = {}
177
178         self.add_node(0) # Add the first node
179
180         for i in range(1, n):
181             self.add_node(i) # Add nodes from 1 to n-1
182
183             # Connect the new node to l existing nodes
184             selected_nodes = self.select_nodes_to_connect(l)
185             for node in selected_nodes:
186                 self.add_edge(i, node)
187
188     def select_nodes_to_connect(self, l):
189         # Calculate the total degree of the graph
190         total_degree = sum([len(neighbors) for neighbors in self.nodes.values()])
191
192         if total_degree == 0:
193             # If total_degree is zero, assign equal probabilities to all nodes
194             probabilities = [1 / len(self.nodes)] * len(self.nodes)
195         else:
196             # Calculate the probability for each node to be selected
197             probabilities = [len(neighbors) / total_degree for neighbors in self.nodes.values()]
198
199         # Select l nodes based on the probabilities
200         selected_nodes = rand.choices(list(self.nodes.keys()), weights=probabilities, k=l)
201         return selected_nodes
202
203
204
205
206
207
208
209
210
211
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
```

```
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("ER Clustering Coefficient:", clustering_coefficient)
13
14
```



ER Diameter: 3  
ER Clustering Coefficient: 0.00997833208934889

```
1 graph = ErdosRenyiGraph()
2 graph.generate_er_graph(10000, 0.11) # Generate an Erdos-Renyi graph with 1000 nodes and edge probability 0.01 degree
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("ER Diameter:", diameter)
9
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("ER Clustering Coefficient:", clustering_coefficient)
```

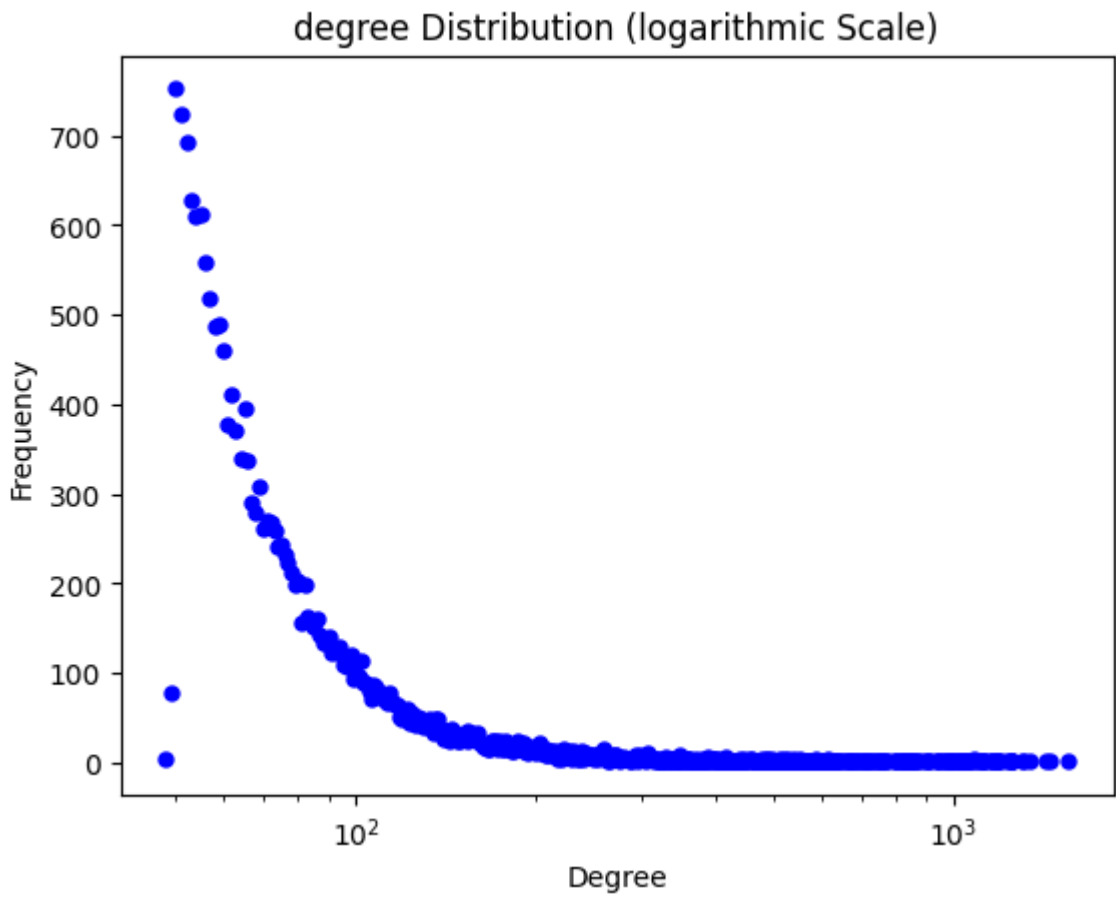
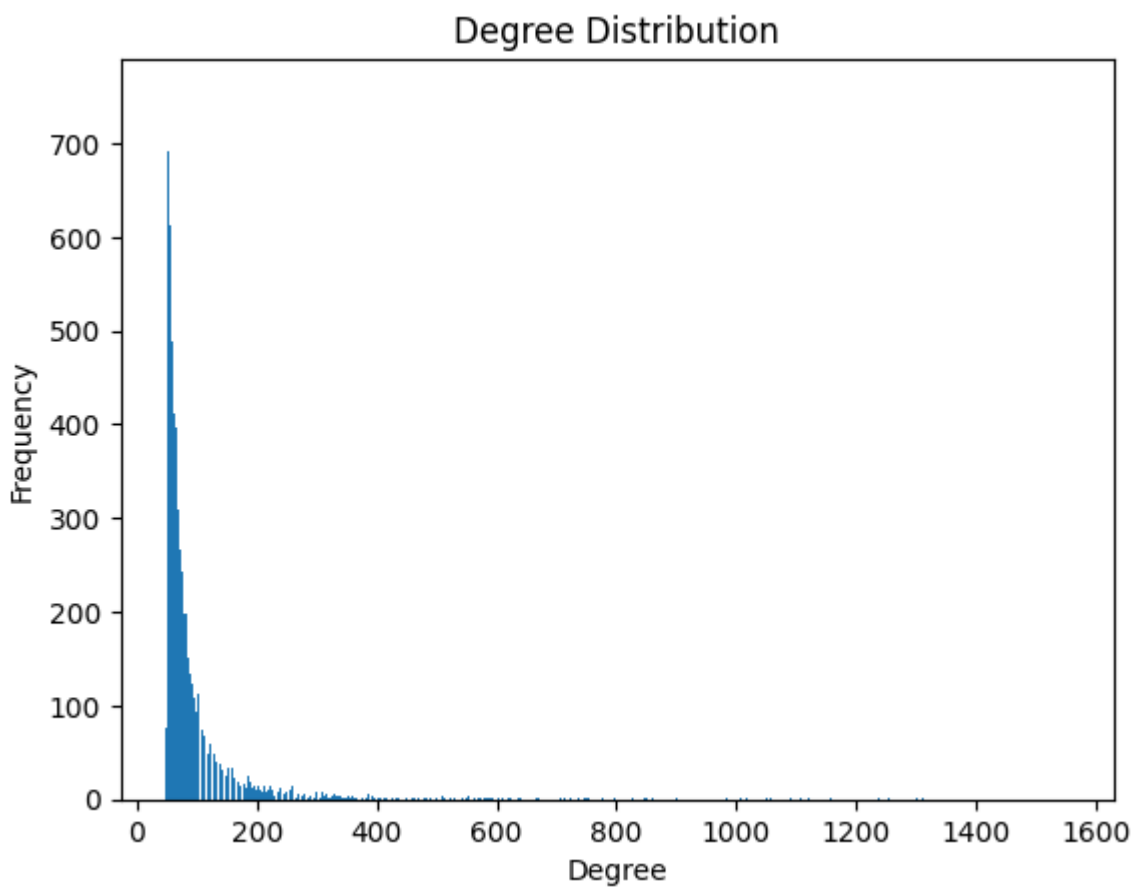


```
1 graph = BarabasiAlbertGraph()
2 graph.generate_ba_graph(10000, 5)    # Generate an
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("BA Diameter:", diameter)
9
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("BA Clustering Coefficient:", clustering_coefficient)
13 print("Connectedness:", graph.compute_connectedness())
14 print("Average Path Length:", graph.compute_average_path_length())
15
```

```
1 graph = BarabasiAlbertGraph()
2 graph.generate_ba_graph(20000, 5)    # Generate an
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("BA Diameter:", diameter)
9
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("BA Clustering Coefficient:", clustering_coefficient)
13 print("Connectedness:", graph.compute_connectedness())
14 print("Average Path Length:", graph.compute_average_path_length())
15
```

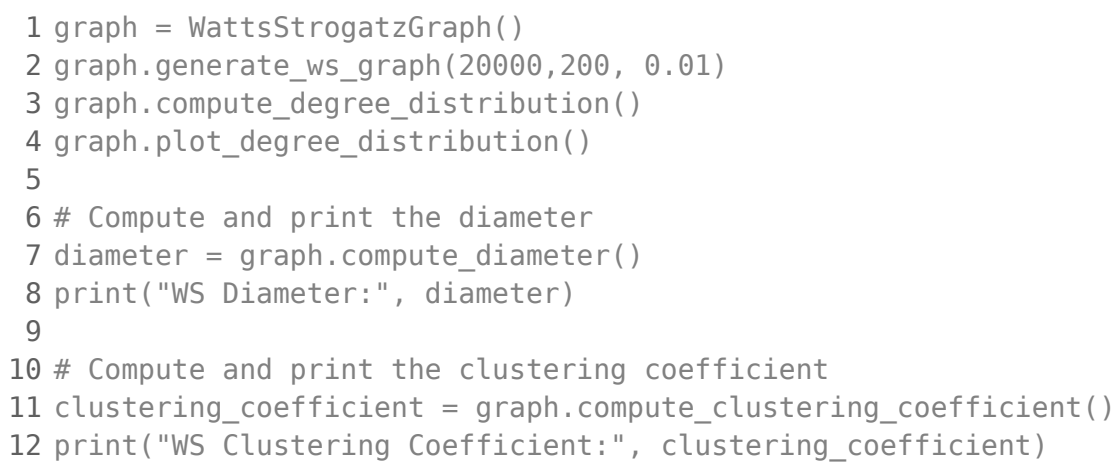
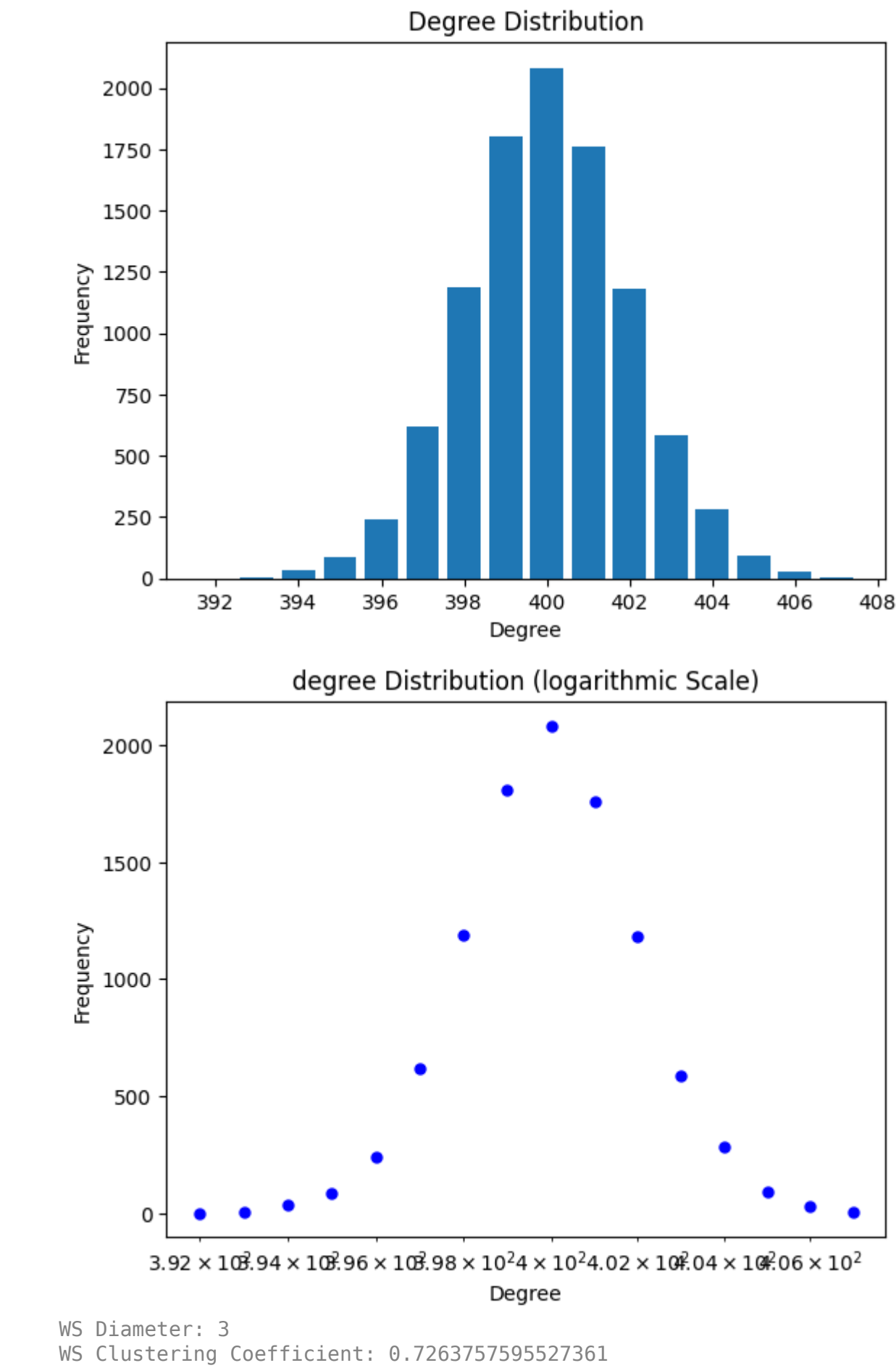
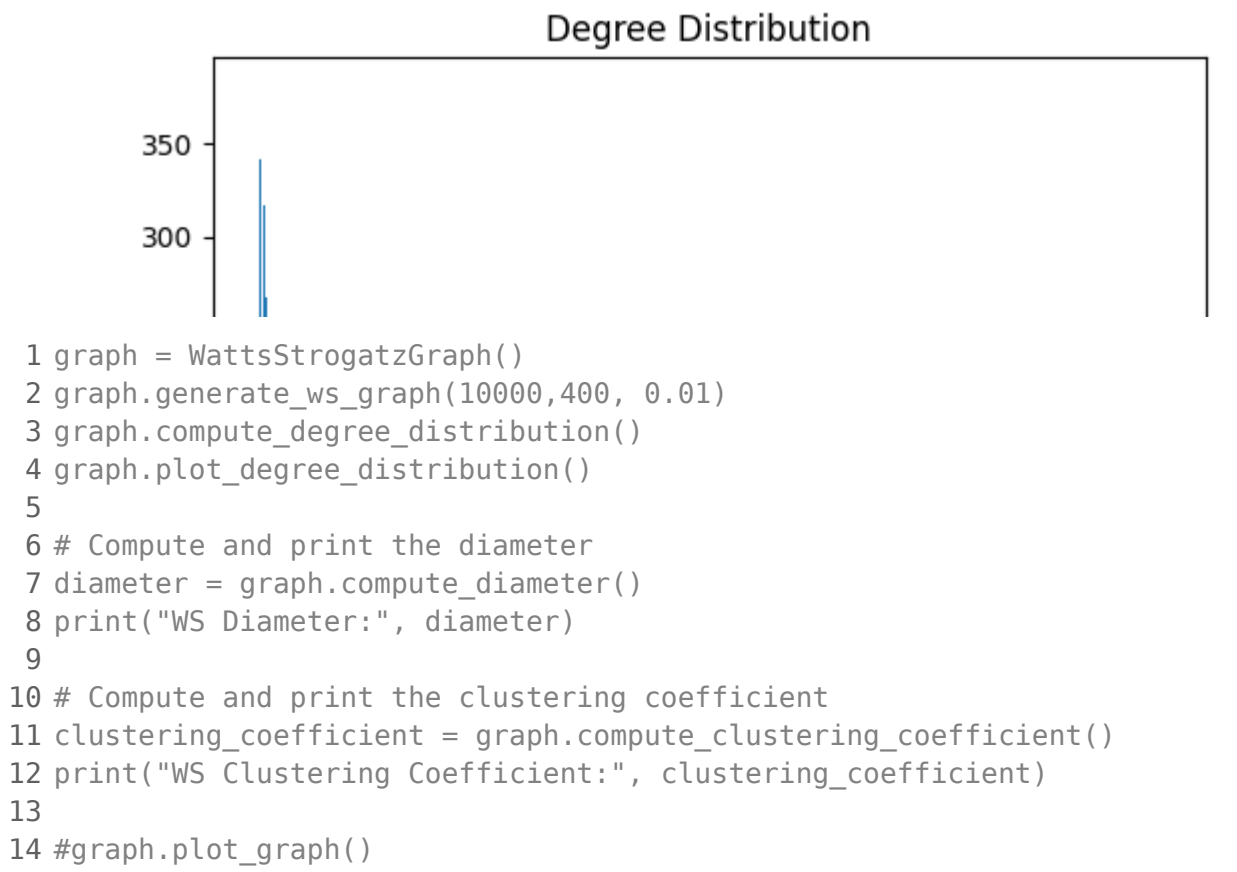
```
1 graph = BarabasiAlbertGraph()
2 graph.generate_ba_graph(20000, 50)    # Generate an
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("BA Diameter:", diameter)
9
```

```
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("BA Clustering Coefficient:", clustering_coefficient)
13
14
```

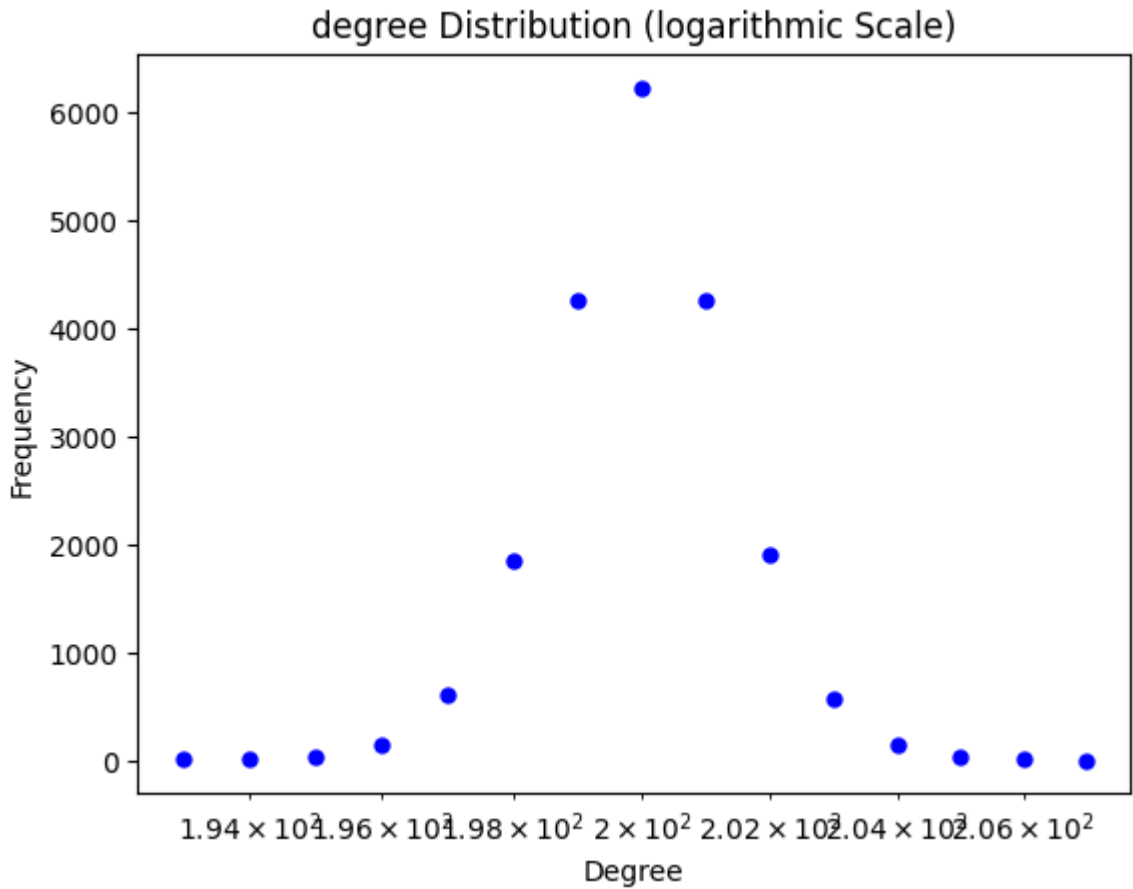
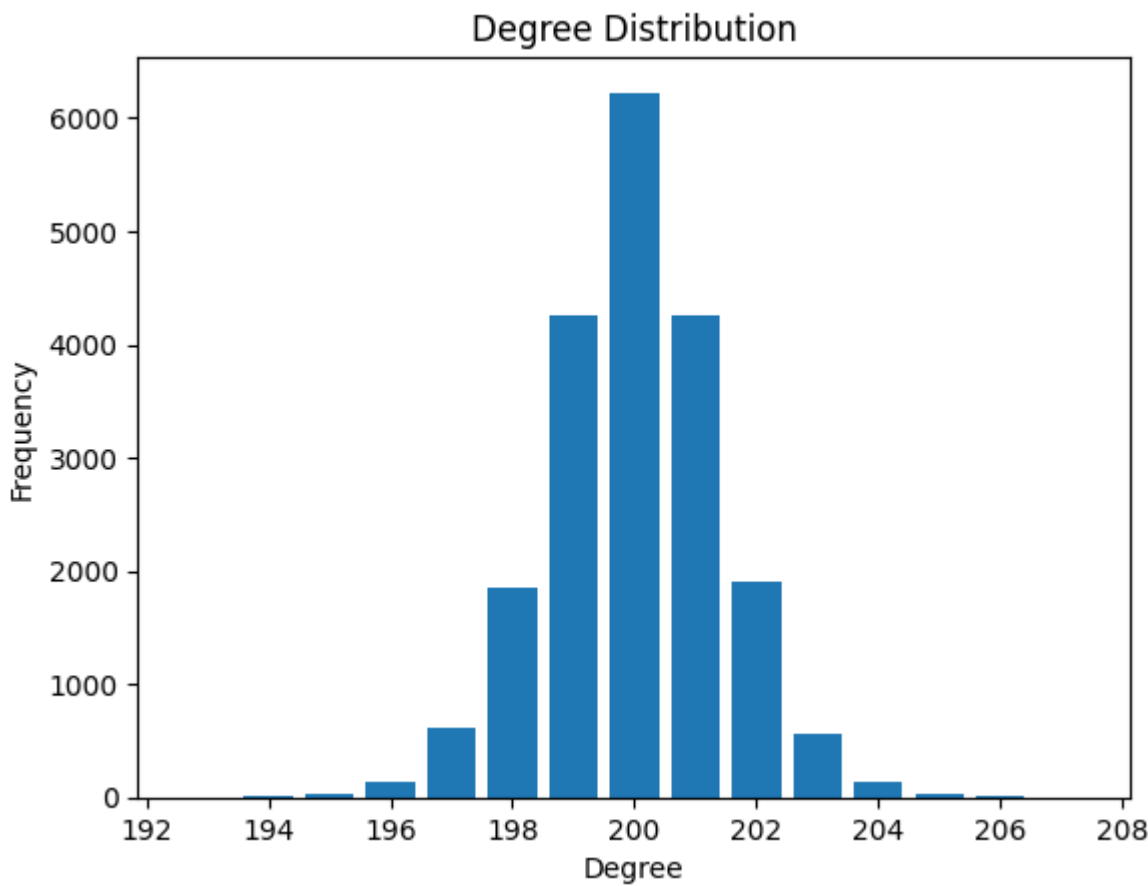


BA Diameter: 3  
BA Clustering Coefficient: 0.019026751954177537

```
1 graph = BarabasiAlbertGraph()
2 graph.generate_ba_graph(20000, 100)    # Generate an
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("BA Diameter:", diameter)
9
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("BA Clustering Coefficient:", clustering_coefficient)
13
14
```

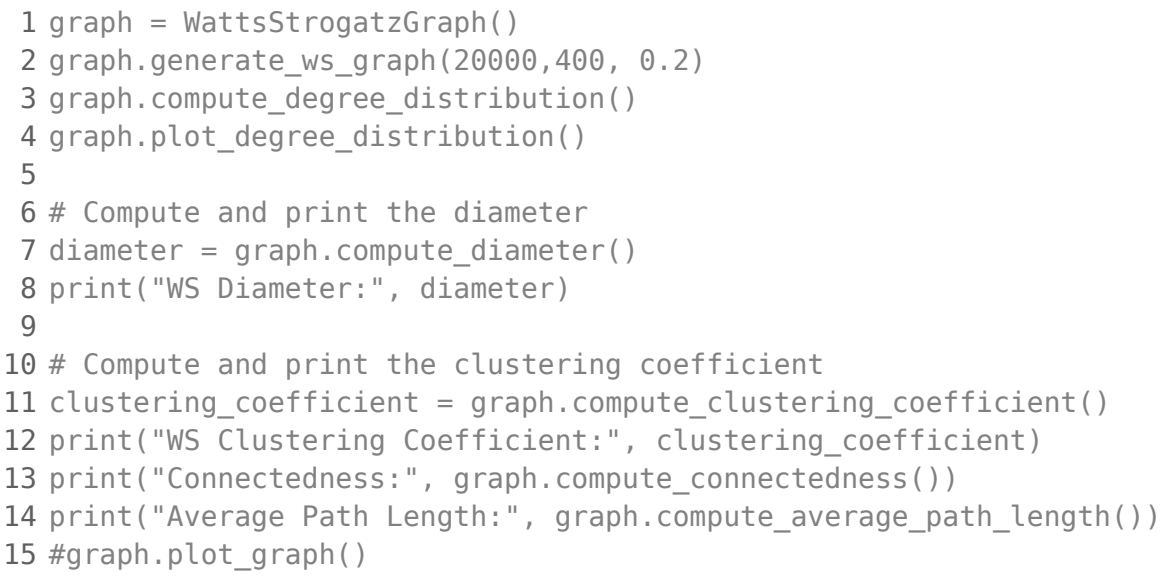
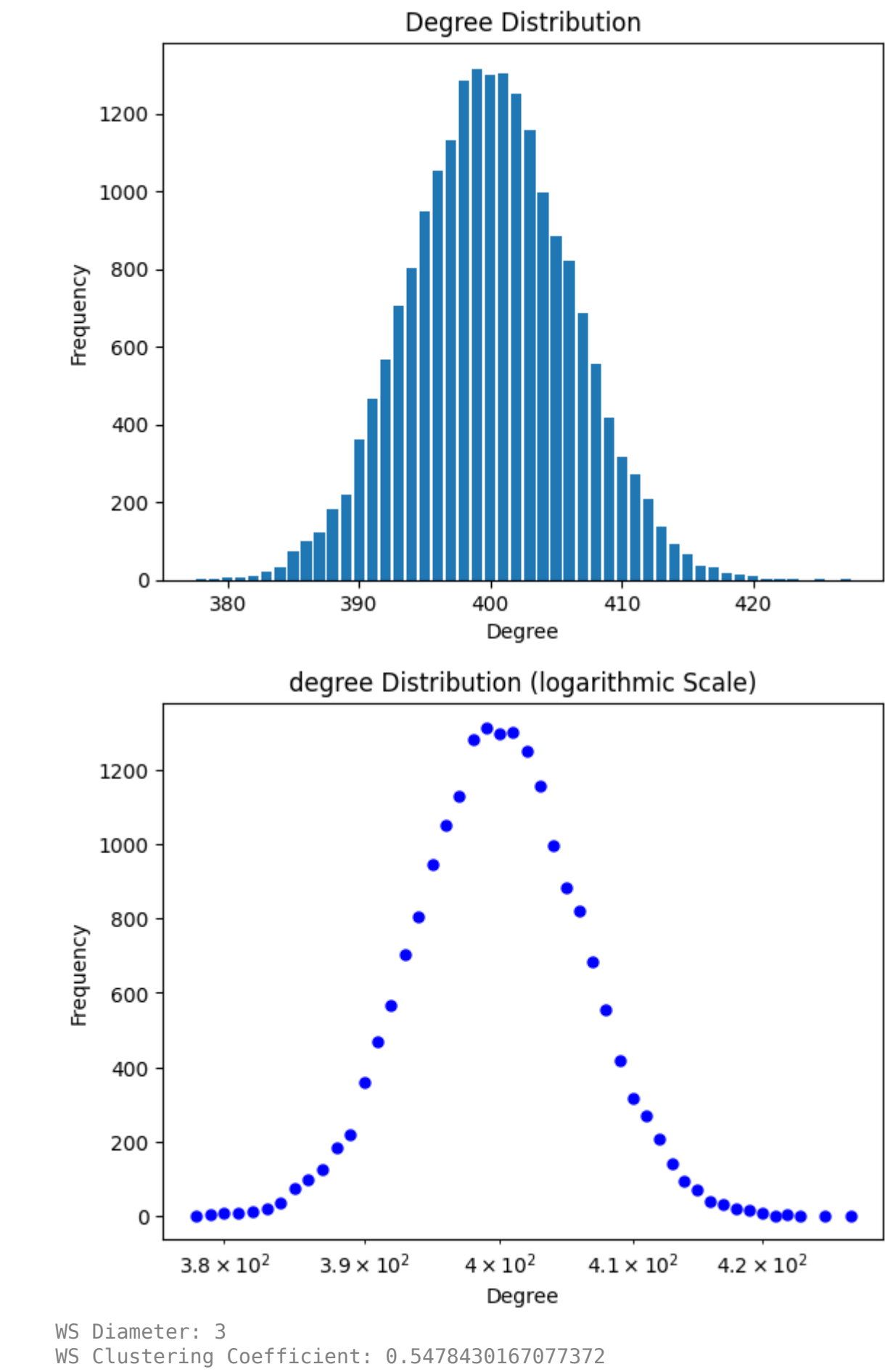
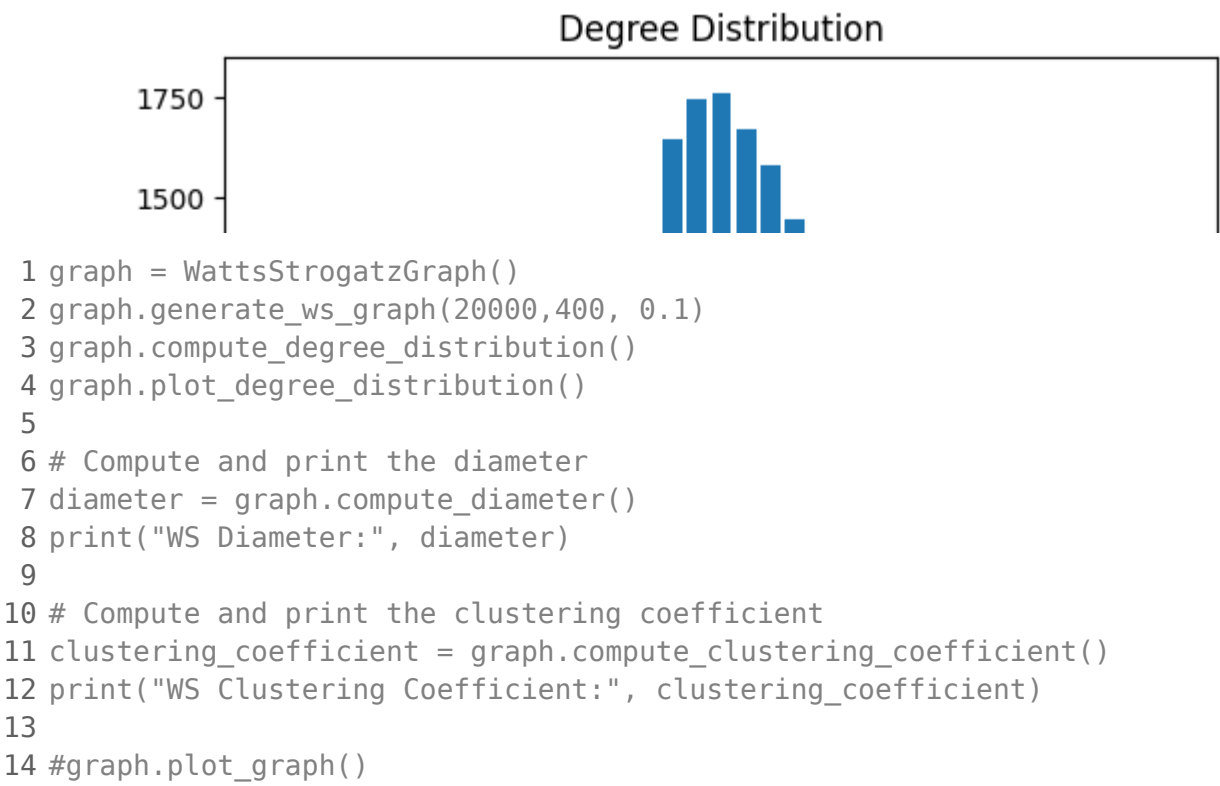


13  
14 #graph.plot\_graph()

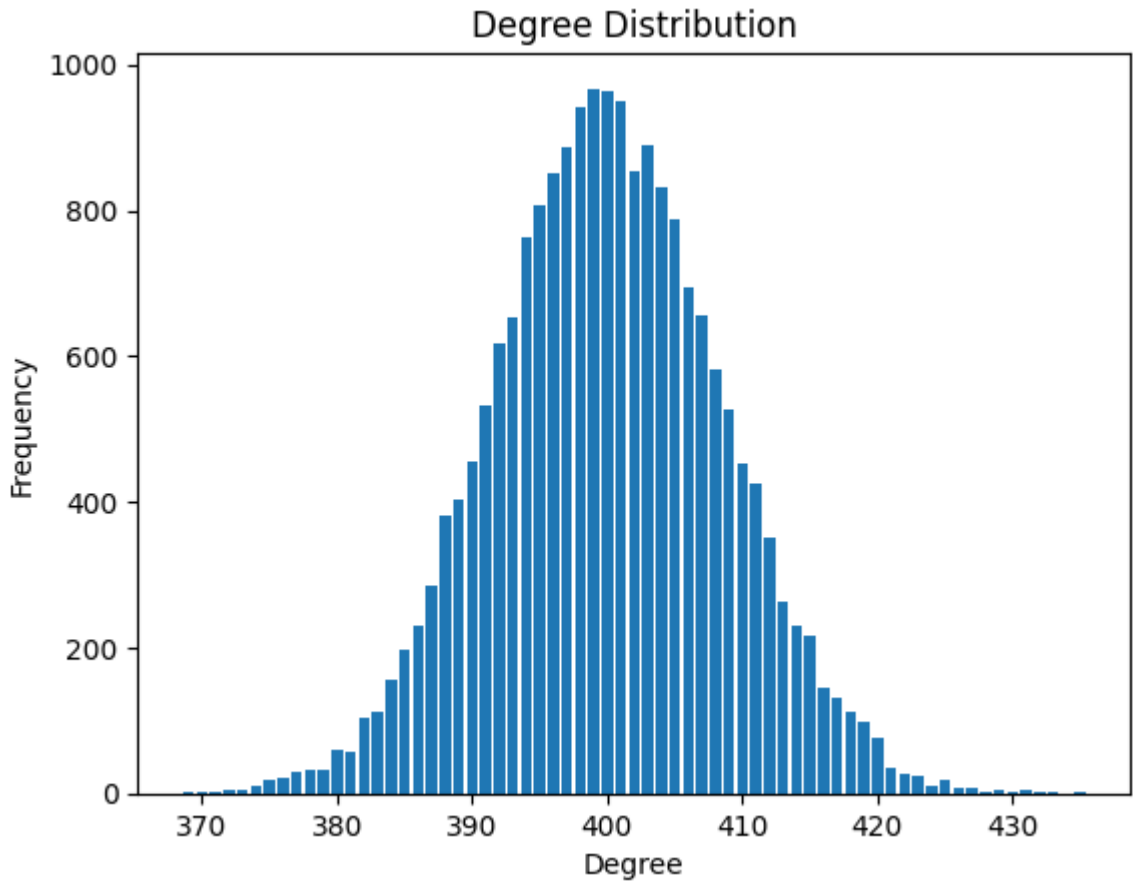


WS Diameter: 4  
WS Clustering Coefficient: 0.7237925804327776

```
1 graph = WattsStrogatzGraph()  
2 graph.generate_ws_graph(20000,200, 0.11)  
3 graph.compute_degree_distribution()  
4 graph.plot_degree_distribution()  
5  
6 # Compute and print the diameter  
7 diameter = graph.compute_diameter()  
8 print("WS Diameter:", diameter)  
9  
10 # Compute and print the clustering coefficient  
11 clustering_coefficient = graph.compute_clustering_coefficient()  
12 print("WS Clustering Coefficient:", clustering_coefficient)  
13 #graph.plot_graph()
```







degree Distribution (logarithmic Scale)

```
1 graph = WattsStrogatzGraph()
2 graph.generate_ws_graph(20000,800, 0.2)
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("WS Diameter:", diameter)
9
10 # Compute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("WS Clustering Coefficient:", clustering_coefficient)
13 print("Connectedness:", graph.compute_connectedness())
14 print("Average Path Length:", graph.compute_average_path_length())
15 #graph.plot_graph()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-7f0417b47873> in <cell line: 1>()
----> 1 graph = WattsStrogatzGraph()
      2 graph.generate_ws_graph(20000,800, 0.2)
      3 graph.compute_degree_distribution()
      4 graph.plot_degree_distribution()
      5

NameError: name 'WattsStrogatzGraph' is not defined
```

SEARCH STACK OVERFLOW

```
1 graph = WattsStrogatzGraph()
2 graph.generate_ws_graph(20000,800, 0.5)
3 graph.compute_degree_distribution()
4 graph.plot_degree_distribution()
5
6 # Compute and print the diameter
7 diameter = graph.compute_diameter()
8 print("WS Diameter:", diameter)
9
10 # ompute and print the clustering coefficient
11 clustering_coefficient = graph.compute_clustering_coefficient()
12 print("WS Clustering Coefficient:", clustering_coefficient)
13 print("Connectedness:", graph.compute_connectedness())
14 print("Average Path Length:", graph.compute_average_path_length())
15 #graph.plot_graph()
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-11b5b807c5fe> in <cell line: 1>()
----> 1 graph = WattsStrogatzGraph()
      2 graph.generate_ws_graph(20000,800, 0.5)
      3 graph.compute_degree_distribution()
      4 graph.plot_degree_distribution()
      5

NameError: name 'WattsStrogatzGraph' is not defined
```

SEARCH STACK OVERFLOW

► Problem 2

[ ] ↪ 1 cell hidden

▼ Problem 5

Files:

nodeld.edges : The edges in the ego network for the node 'nodeld'. Edges are undirected. The 'ego' node does not appear, but it is assumed that they follow every node id that appears in this file.

nodeld.circles : The set of circles for the ego node. Each line contains one circle, consisting of a series of node ids. The first entry in each line is the name of the circle.

nodeld.feats : The features for each of the nodes that appears in the edge file.

nodeld.egofeat : The features for the ego user.

nodeld.featsnames : The names of each of the feature dimensions. Features are '1' if the user has this property in their profile, and '0' otherwise. This file has been anonymized for facebook users, since the names of the features would reveal private data.

```
1 import pandas as pd
2 import numpy as np
```

```
3 import codecs
4 import csv
5
1 from google.colab import drive
2 drive.mount('/content/drive')

Mounted at /content/drive
```

## ▼ Data Preprocessing

```
1 import os
2 import numpy as np
3
4 data_dir = '/content/drive/MyDrive/Social Networks Homework_Vano Mazashvili/facebook'
5 node_ids = ['0', '107', '348', '414', '686', '698', '1684', '1912', '3437', '3980']
6
7 def process_feat_files(data_dir, node_id):
8     feat_file_path = os.path.join(data_dir, f'{node_id}.feat')
9     featname_file_path = os.path.join(data_dir, f'{node_id}.featnames')
10
11     with open(feat_file_path, 'r') as feat_file, open(featname_file_path, 'r') as featname_file:
12         featnames = featname_file.read().strip().split('\n')
13         arrays = []
14
15         for line in feat_file:
16             line = line.strip()
17             if line:
18                 feat = list(map(int, line.split()[1:]))
19                 selected_featnames = [featnames[i-1].split()[-1] for i, val in enumerate(feat, start=1) if val == 1]
20
21                 new_array = np.zeros(1283, dtype=int)
22                 for featname in selected_featnames:
23                     new_array[int(featname)] = 1
24
25                 arrays.append(new_array)
26
27         arrays = np.array(arrays)
28         return arrays
29
30
31 edge_files= [f for f in os.listdir(data_dir) if f.endswith('.edges')]
32
33 edge_file_path = os.path.join(data_dir, f'{node_ids}.edges')
34
35 edge_files = [f for f in os.listdir(data_dir) if f.endswith('.edges')]
36
37 edges = []
38 for edge_file in edge_files:
39     with open(os.path.join(data_dir, edge_file), 'r') as file:
40         for line in file:
41             edge = line.strip().split()
42             edge = list(map(int, edge)) # Convert edges to integers
43             edges.append(edge)
44
45 unique_nodes = set()
46 for edge in edges:
47     unique_nodes.add(edge[0])
48     unique_nodes.add(edge[1])
49
50
51
52 features = None
53
54 for node_id in node_ids:
55     result = process_feat_files(data_dir, node_id)
56     if features is None:
57         features = result
58     else:
59         features = np.concatenate((features, result), axis=0)
60
61 circles_files = [f for f in os.listdir(data_dir) if f.endswith('.circles')]
62 l = {}
63 for circles_file in circles_files:
64     with open(os.path.join(data_dir, circles_file), 'r') as file:
65         for line in file:
66             circle = line.strip().split()
67             circle_name = circle[0]
68             circle_nodes = circle[1:]
69
70             if circle_name in l:
71                 l[circle_name].extend(circle_nodes)
72             else:
73                 l[circle_name] = circle_nodes
74
75 # Assign labels to nodes
76 node_ids = set()
77 for node_list in l.values():
78     node_ids.update(node_list)
79
80 max_label_index = len(l) - 1
81
82 label_vectors = []
83 for node_id in range(1, 4168): # Assuming 4167 nodes in total
84     label_vector = [0] * (max_label_index + 1)
85     for label_index, node_list in enumerate(l.values()):
86         if str(node_id) in node_list:
87             label_vector[label_index] = 1
88     label_vectors.append(label_vector)
89
90 labels = []
91 labels = np.sum(label_vectors, axis=1)
92
93 num_nodes = len(unique_nodes)
94 lb = np.zeros((num_nodes, 46), dtype=int)
95
96 for node_id, node_list in enumerate(l.values()):
97     for circle_name, circle_nodes in l.items():
98         circle_index = int(circle_name.replace("circle", "")) - 1
99         if str(node_id + 1) in circle_nodes:
```

```
100         lb[node_id][circle_index] = 1
101
102
103 #print(len(labels))
104 #print(max(labels))
105 #print(labels)
106 #print(features)
107 #print(features[345][2])
108 #print(len(features))
109 print(len(edges))
110 print(len(features))
111 print(l)
112 print(labels)
113 print(len(labels))
114 num_unique_nodes = len(unique_nodes)
115 print(f"Number of unique nodes: {num_unique_nodes}")

170174
4167
{'circle0': ['71', '215', '54', '61', '298', '229', '81', '253', '193', '97', '264', '29', '132', '110', '163', '259', '183', '334', '245', '222', '475', '373', '461', '39', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36', '37', '38', '39', '40', '41', '42', '43', '44', '45', '46', '47', '48', '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '70', '71', '72', '73', '74', '75', '76', '77', '78', '79', '80', '81', '82', '83', '84', '85', '86', '87', '88', '89', '90', '91', '92', '93', '94', '95', '96', '97', '98', '99']}
[1 1 1 ... 0 0 0]
4167
Number of unique nodes: 3959
```

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torch_geometric.data import Data
5 from torch_geometric.nn import GCNConv
6 from torch.nn import Linear
7
8 # Convert the edges list to a PyTorch Geometric data object
9 edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()
10
11 # Convert the features and labels to PyTorch tensors
12 x = torch.tensor(features, dtype=torch.float)
13 y = torch.tensor(labels, dtype=torch.long)
14
15
16 # Create a PyTorch Geometric data object
17 data = Data(x=x, edge_index=edge_index, y=y)
18
19 # Define the GNN model
20 class GNNModel(nn.Module):
21     def __init__(self, input_dim, hidden_dim, output_dim):
22         super(GNNModel, self).__init__()
23         self.conv1 = GCNConv(input_dim, hidden_dim)
24         self.conv2 = GCNConv(hidden_dim, hidden_dim)
25         self.conv3 = GCNConv(hidden_dim, hidden_dim)
26         self.conv4 = GCNConv(hidden_dim, hidden_dim)
27         self.conv5 = GCNConv(hidden_dim, hidden_dim)
28         self.out = Linear(hidden_dim, output_dim)
29
30
31     def forward(self, x, edge_index):
32
33         # First Message Passing Layer (Transformation)
34         x = self.conv1(x, edge_index)
35         x = x.relu()
36         x = F.dropout(x, p=0.5, training=self.training)
37
38         # Second Message Passing Layer
39         x = self.conv2(x, edge_index)
40         x = x.relu()
41         x = F.dropout(x, p=0.5, training=self.training)
42
43         x = self.conv3(x, edge_index)
44         x = x.relu()
45         x = F.dropout(x, p=0.5, training=self.training)
46         x = self.conv4(x, edge_index)
47         x = x.relu()
48         x = F.dropout(x, p=0.5, training=self.training)
49         x = self.conv5(x, edge_index)
50         x = x.relu()
51         x = F.dropout(x, p=0.5, training=self.training)
52         # Output layer
53         x = F.softmax(self.out(x), dim=1)
54         return x
55
56 # Set the dimensions for the GNN model
57 input_dim = x.shape[1]
58 hidden_dim = 128
59 output_dim = len(l)
60
61 # Create an instance of the GNN model
62 model = GNNModel(input_dim, hidden_dim, output_dim)
63
64 # Define the loss function and optimizer
65 criterion = nn.CrossEntropyLoss()
66 optimizer = optim.Adam(model.parameters(), lr=0.001)
67
68 # Train the GNN model
69 def train(model, data, optimizer, criterion, num_epochs):
70     model.train()
71
72     for epoch in range(num_epochs):
73         optimizer.zero_grad()
74         out = model(data.x, data.edge_index)
75         loss = criterion(out[data.train_mask], data.y[data.train_mask])
76         loss.backward()
77         optimizer.step()
78
79         # Perform evaluation on the validation set
80         model.eval()
81         with torch.no_grad():
82             logits = model(data.x, data.edge_index)
83             pred = logits.argmax(dim=1)
84             correct = pred[data.val_mask] == data.y[data.val_mask]
85             val_acc = int(correct.sum()) / int(data.val_mask.sum())
86
87         print(f'Epoch: {epoch+1}, Loss: {loss.item()}, Val Acc: {val_acc}')
88
89 # Split the data into training, validation, and testing masks
```

```
90 num_nodes = len(data.y)
91 train_mask = torch.zeros(num_nodes, dtype=torch.bool)
92 val_mask = torch.zeros(num_nodes, dtype=torch.bool)
93 test_mask = torch.zeros(num_nodes, dtype=torch.bool)
94
95 # Assuming 70% training, 15% validation, and 15% testing split
96 train_mask[:int(num_nodes*0.7)] = True
97 val_mask[int(num_nodes*0.7):int(num_nodes*0.85)] = True
98 test_mask[int(num_nodes*0.85):] = True
99
100 data.train_mask = train_mask
101 data.val_mask = val_mask
102 data.test_mask = test_mask
103
104 # Train the model
105 num_epochs = 100
106 train(model, data, optimizer, criterion, num_epochs)
107
108 # Evaluate the model on the testing set
109 model.eval()
110 with torch.no_grad():
111     logits = model(data.x, data.edge_index)
112     pred = logits.argmax(dim=1)
113     correct = pred[data.test_mask] == data.y[data.test_mask]
114     test_acc = int(correct.sum()) / int(data.test_mask.sum())
115
116 print(f'Test Acc: {test_acc}')
117
Epoch: 44, Loss: 3.3399364948272705, Val Acc: 0.8128
Epoch: 45, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 46, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 47, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 48, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 49, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 50, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 51, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 52, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 53, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 54, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 55, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 56, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 57, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 58, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 59, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 60, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 61, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 62, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 63, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 64, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 65, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 66, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 67, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 68, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 69, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 70, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 71, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 72, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 73, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 74, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 75, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 76, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 77, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 78, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 79, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 80, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 81, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 82, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 83, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 84, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 85, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 86, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 87, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 88, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 89, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 90, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 91, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 92, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 93, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 94, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 95, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 96, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 97, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 98, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 99, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 100, Loss: 3.339937210083008, Val Acc: 0.8128
Test Acc: 0.13258785942492013

1 from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
2
3 # Evaluate the model on the testing set
4 model.eval()
5 with torch.no_grad():
6     logits = model(data.x, data.edge_index)
7     pred = logits.argmax(dim=1)
8     pred_test = pred[data.test_mask].cpu().numpy()
9     labels_test = data.y[data.test_mask].cpu().numpy()
10
11 accuracy = accuracy_score(labels_test, pred_test)
12 precision = precision_score(labels_test, pred_test, average='macro')
13 recall = recall_score(labels_test, pred_test, average='macro')
14 f1 = f1_score(labels_test, pred_test, average='macro')
15
16 print(f'Test Accuracy: {accuracy:.4f}')
17 print(f'Precision: {precision:.4f}')
18 print(f'Recall: {recall:.4f}')
19 print(f'F1-score: {f1:.4f}')
20

📄 Test Accuracy: 0.1326
Precision: 0.0221
Recall: 0.1667
F1-score: 0.0390
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no pr
_warn_prf(average, modifier, msg_start, len(result))
```

So, our feature array will have length of 1283 and will be the same for all nodes to avoid jagged matrix

code given below creates feature vectors with the length of 1283 for each node

Obtaining Labels from the .circles files