Contents

1 Part 1, nauty, pipes, and graph6 2

1.1 Problem statement 2

1.2 Solution: 11 graphs on 4 vertices with no triangles in g6 format 2

1.3 Solution: Read, process, and write graphs in g6-format and avoid K5 $\in \bar{G}$ 3

1.4 Solution: Read, process, and write graphs in g6-format and avoid K6 $\in \bar{G}$ 7

2 Part 2, no programming, just some nauty help 8

2.1 Problem statement 8

2.2 Solution: Draw nicely the single graph obtained above for $k = 12$. 8

2.3 Solution: Draw nicely the two most symmetric graphs among those you obtained above for $k = 11$ (on 12 vertices). 9

3 Debugging 11

4 Source code 14

4.1 Bash scripts for high-level management and piping 14

4.2 Scripts to generate all possible graphs and to filter the results 16

4.3 Utility functions for the low-level details 21

4.4 Scripts to count the number of edges in each graph 29

Listings

1 Output from nauty for 11 graphs on 4 vertices with no triangles. 2

2 Contents of `n4.g6`. 3

3 Results for (3,5,n)-graphs for all possible edges e. 4

4 Output for $k = 11$ (12 vertices). 6

5 Output for $k = 12$ (13 vertices). 6

6 Output for $k = 13$ (14 vertices). 6

7 Results for (3,6,n)-graphs for all possible edges e. 7

8 Confirm that my `countedges.py` script produces the correct results, given nauty input, which is known to be correct. 13

9 The adjacency lists of the two graphs used in debugging. 13

10 Bash script `./run35ne.sh` to create all $(3, 5, n, e)$-graphs. 14

11 Bash script `./run36ne.sh` to create all $(3, 6, n, e)$-graphs. 15

12 `gengraphs.py` : Generate all the graphs with $k + 1$ vertices from the current input graphs with k vertices. 16

13 `filterC3andK5.py` : Remove graphs G with C3 $\in G$ and K5 $\in \bar{G}$. 17

14 `filterC3andK6.py` : Remove graphs G with C3 $\in G$ and K6 $\in \bar{G}$. 19

15 `utils.py` : Utility functions. 21

16 `genperm.py` : For a given k, generate all permutations of 0s and 1s of length k. 27

17 Permutations output for $k = 4$ from `genperm.py`. 28

18 `compare.py` : Compare my output vs. nauty output. 29

19 `countedges.py` : Count number of edges in G. Used for debugging and for printing the table of edges vs. number of vertices. 29

20 `createtable.py` : Produce the table of edges vs. number of vertices. 31

1 Part 1, nauty, pipes, and graph6

1.1 Problem statement

In this part of the assignment, use (nauty) functions which read and write graphs in g6-format[1] of graphs. Use pipes in at least some places.

1. In part 2.1 of the previous assignment you found 11 graphs on 4 vertices. Print g6-format of those among them which have no triangles; put them into the file `n4.g6`.
2. Show that you can read, process, and write graphs in g6-format. Write a program which reads graphs from input file `I = n[k].g6` and makes output file `O = n[k+1].g6`, such that `O` consists exactly of all canonically labeled graphs, which have $(k+1)$ vertices, have no triangles, and no independent sets of order 5.
   - Iterate your program for `(k=4;k<14;k++)`. Which nauty functions and with what options did you use? Make use of some pipes. Include any special script, if any. You may corroborate your results with the contents of table III on page 46 of the paper at position #95 of the list[2] (just 4 tables at tabs88.pdf[3]).
   - Print g6-format of graphs you obtained for $k = 11$, $k = 12$, and $k = 13$.
   - Print commented source code you wrote for this assignment (do not include nauty code or any parts of other libraries, but do include any of your scripts using them).
3. (Optional) Follow the process of item 2., suitably adjusted, for triangle-free graphs but avoiding K6 instead of K5.

1.2 Solution: 11 graphs on 4 vertices with no triangles in g6 format

Listing 1. Output from nauty for 11 graphs on 4 vertices with no triangles. Checking with `showg` shows that this matches the output from HW01. The stdout is directed to the `n4.g6` file.

```
>>  geng 4 -tv > n4.g6

>A geng -td0D3 n=4 e=0-4
>C 1 graphs with 0 edges
>C 1 graphs with 1 edges
>C 2 graphs with 2 edges
>C 2 graphs with 3 edges
>C 1 graphs with 4 edges
>Z 7 graphs generated in 0.00 sec
```

[1] `http://users.cecs.anu.edu.au/~bdm/data/formats.txt`
[2] `https://www.cs.rit.edu/~spr/PUBL/publ.html`
[3] `https://www.cs.rit.edu/~spr/COURSES/CCOMP/tabs88.pdf`

Listing 2. Contents of `n4.g6`.

```
C`
C?
C@
CB
CF
Cr
CR
```

I started with $4 \leq k \leq 14$ as stated by the problem, then I started with a graph of 1 vertex and 0 edges (which is `@` in .g6 format), ran the script from $1 \leq k \leq 14$, and got the correct results with both methods (Listing 3 on page 4).

1.3 Solution: Read, process, and write graphs in g6-format and avoid K5 $\in \bar{G}$

1.3.1 Procedure

Given an input file `n[k].g6` and the value k, construct all possible graphs, label and filter such that only non-isomorphic $(3, 5, k+1, e)$-graphs remain, and write these results to an output file `n[k+1].g6`. Write utility functions that encode/decode the .g6 format as defined at `https://users.cecs.anu.edu.au/~bdm/data/formats.txt`.

The bash script `run35ne.sh` (Listing 10 on page 14) executes this process:

1. Use `genperm.py` to generate all 2^k permutations P of 0s and 1s for connecting the $(k+1)^{th}$ vertex to an existing graph on k vertices.[4]

2. Use `gengraphs.py` to append every permutation p to the binary form of every graph G in the k^{th} input file. Simply append p to the binary list created from the .g6 ASCII format; no need for 0-1 matrices yet.

3. Pipe the output to the nauty function `labelg -gq` to canonically label the graphs.

4. Pipe to `sort` and `uniq` to sort lexicographically and then to remove duplicates, since canonical labeling turns graph isomorphism into a simple string comparison. Now we know that we have unique graphs to filter.

5. Pipe to `filterC3andK5.py` to filter out any graphs that have 3-cycles (C3) and/or independent sets of size 5 (K5 $\in \bar{G}$) by brute-force checking all possible combinations (returning early if a bad candidate is found). Use 0-1 matrices here for indexing the edges in C3 and K5.

6. Write the file with $k+1$ vertices to stdout, increment k, and repeat.

[4] To generate, start with base case ($k = 0$) of $P_0 = [[0], [1]]$. Generate P_{k+1} by making two copies of P_k, prepending 0 to one copy, 1 to the other copy, and then joining these two results into $P_{k,new}$ for the next iteration.

1.3.2 Results

Listing 3. Results for (3,5,n)-graphs for all possible edges e. These match the correct results from the 1988 paper. The code runs in under 2 minutes and uses no more than 2 GB of RAM with one CPU core is at 100% while the other three are around 20%.

```
$ time bash run35ne.sh

For graphs with   1  vertices,    1  (3,5)-graphs exist.
For graphs with   2  vertices,    2  (3,5)-graphs exist.
For graphs with   3  vertices,    3  (3,5)-graphs exist.
For graphs with   4  vertices,    7  (3,5)-graphs exist.
For graphs with   5  vertices,   13  (3,5)-graphs exist.
For graphs with   6  vertices,   32  (3,5)-graphs exist.
For graphs with   7  vertices,   71  (3,5)-graphs exist.
For graphs with   8  vertices,  179  (3,5)-graphs exist.
For graphs with   9  vertices,  290  (3,5)-graphs exist.
For graphs with  10  vertices,  313  (3,5)-graphs exist.
For graphs with  11  vertices,  105  (3,5)-graphs exist.
For graphs with  12  vertices,   12  (3,5)-graphs exist.
For graphs with  13  vertices,    1  (3,5)-graphs exist.
For graphs with  14  vertices,    0  (3,5)-graphs exist.

real    1m29.027s
user    1m33.428s
sys     0m0.581s
```

	$n=1$	2	3	4	5	6	7	8	9	10	11	12	13
$e=0$	1	1	1	1									
1		1	1	1	1								
2			1	2	2	1							
3				2	3	3	1						
4				1	4	6	2	1					
5					2	8	7	1					
6					1	7	13	5					
7						4	17	13	1				
8						2	15	27	3				
9						1	10	39	11				
10							4	41	28	1			
11							1	27	59	2			
12							1	15	73	10			
13								6	62	32			
14								2	33	69			
15								1	14	86	1		
16								1	4	65	6		
17									2	32	19		
18										12	31		
19										3	30		
20										1	13	1	
21											4	2	
22											1	5	
23												2	
24												2	
25													
26													1

Table 1. Number of edges e vs. number of vertices n for all $(3, 5, n, e)$-graphs. This matches the results from the paper. Created with Listing 20 on page 31.

1.3.3 .g6 format

The form of .g6 is $[N(n)\ R(x)]$, where $N(n)$ is the number of bytes required to store the number of vertices n and $R(x)$ is a row vector representation of the 0-1 matrix of the graph G. Since for this exercise, n is always ≤ 63, $N(n)$ is always the single byte $n + 63$. Therefore, the first letter of the .g6 output is $@ \rightarrow A \rightarrow B \rightarrow C \rightarrow D \ldots$ indicating $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \ldots$ vertices, respectively. This is a convenient way to check the number of vertices in G.

Listing 4. Output for $k = 11$ (12 vertices). K maps to 12 vertices in .g6, which is correct.

```
K`aAAGUEpRDo
K@AAHWYoYwTO
K`aAIOiDWsCh
K`?CGtDIkwL_
K?CkQMp[cgL@
K?GTa\cUDGrC
KoCIHaO@XDHB
K``@OkcEICoL
KQ`?pMCQ?bcU
Ks_GagjLASko
Ks_HIGZKQSm_
K?_YPMQoPokc
```

Listing 5. Output for $k = 12$ (13 vertices). L maps to 13 vertices in .g6, which is correct.

```
Ls`?XGRQR@B`Kc
```

Listing 6. Output for $k = 13$ (14 vertices). This is empty because no graphs with 14 vertices, no triangles, and no independent sets of size 5 exist!

```
```

1.3.4 Source code

The source code starts in Section 4 on page 14.

1.4 Solution: Read, process, and write graphs in g6-format and avoid K6 $\in \bar{G}$

1.4.1 Procedure

The brute-force procedure is similar to avoiding K5 $\in G$; "just" add another layer to the brute-force, nested loops. However, the problem blows up. The brute-force code ran for over 7 hours and only finished computing for graphs with 12 vertices. In the textbook, there is an algorithm that generates each clique exactly once (instead of $k!$ times for a clique of size k), and I tried it implement it but did not finish.

1.4.2 Results

Listing 7. Results for (3,6,n)-graphs for all possible edges e. These values that were calculated match the correct results from the 1988 paper.

```
$ time bash run36ne.sh

For graphs with   1  vertices,       1  (3,6)-graphs exist.
For graphs with   2  vertices,       2  (3,6)-graphs exist.
For graphs with   3  vertices,       3  (3,6)-graphs exist.
For graphs with   4  vertices,       7  (3,6)-graphs exist.
For graphs with   5  vertices,      14  (3,6)-graphs exist.
For graphs with   6  vertices,      37  (3,6)-graphs exist.
For graphs with   7  vertices,     100  (3,6)-graphs exist.
For graphs with   8  vertices,     356  (3,6)-graphs exist.
For graphs with   9  vertices,    1407  (3,6)-graphs exist.
For graphs with  10  vertices,    6657  (3,6)-graphs exist.
For graphs with  11  vertices,   30395  (3,6)-graphs exist.
For graphs with  12  vertices,  116792  (3,6)-graphs exist.

// process killed here //

real    444m15.676s
user    487m45.096s
sys 1m31.155s
```

1.4.3 Source code

Run with Listing 11 on page 15.

2 Part 2, no programming, just some nauty help

2.1 Problem statement

For each graph below, list generators of its automorphism group and explain why they show up (or not) in your drawing (dreadnaut and `countg --a` may help). Label your graphs suitably.

1. Draw nicely the single graph obtained above for $k = 12$ (on 13 vertices).
2. Draw nicely the two most symmetric graphs among those you obtained above for $k = 11$ (on 12 vertices).

2.2 Solution: Draw nicely the single graph obtained above for $k = 12$.

```
Ls‘?XGRQR@B‘Kc

> < n13.dre & xo

(1 2 3 4)(5 7 8 6)(9 10 12 11)
level 2:  4 orbits; 1 fixed; index 4
(0 1)(2 9)(3 5)(4 10)(6 8)(7 12)
level 1:  1 orbit; 0 fixed; index 13
1 orbit; grpsize=52; 2 gens; 6 nodes; maxlev=3
cpu time = 0.00 seconds
 0:12 (13);
```

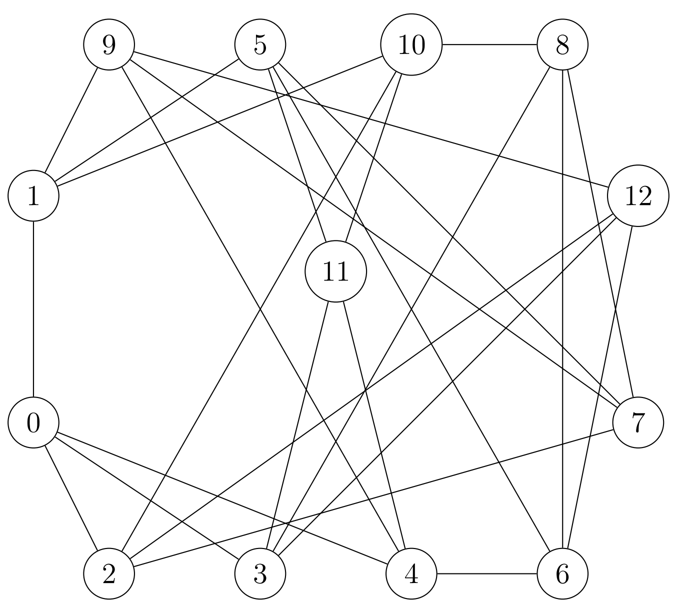

Figure 1. The single graph on 13 vertices with the .g6 format of `Ls`?XGRQR@B`Kc`. All vertices have degree 4. The generators used are `(0 1)(2 9)(3 5)(4 10)(6 8)(7 12)`, and the larger of the pair for each generator is on the top and the smaller is on the bottom. The graph is symmetric about the x axis. However, observe that `11` is not in the cycle of generators, and thus 11 is not part of the x axis symmetry (i.e. if the graph were flipped about the x axis, then 11 would not move).

2.3 Solution: Draw nicely the two most symmetric graphs among those you obtained above for $k = 11$ (on 12 vertices).

Out of the 12 graphs produced, each graph was fed into `dreadnaut`, and the automorphisms were generated with the `xo` command. The graphs with the largest group size (16 and 48, respectively) and the largest number of generators (4 generators for each graph) were chosen. "Graph 3" and "Graph 10" are from the lexicographical ordering of the canonical labeling.

```
K‘aAIOiDWsCh

> < n12.3.dre & xo
(2 3)(4 5)(6 7)(8 9)(10 11)
(2 3)(8 10)(9 11)
level 2:  6 orbits; 8 fixed; index 4
(0 1)(4 6)(5 7)
(0 2)(1 3)(4 8)(5 11)(6 10)(7 9)
level 1:  2 orbits; 0 fixed; index 4
2 orbits; grpsize=16; 4 gens; 9 nodes; maxlev=3
 0:3 (4); 4:11 (8);
```

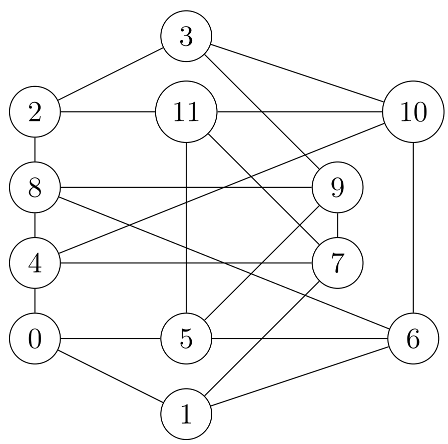

Figure 2. Graph 3. The generators used are `(0 2)(1 3)(4 8)(5 11)(6 10)(7 9)`. The graph is symmetric about the x axis.

```
Ks_GagjLASko

> < n12.10.dre & xo

(1 2)(3 4)(5 6)(7 8)(9 10)
(1 2)(7 9)(8 10)
level 2:  6 orbits; 7 fixed; index 4
(0 1)(2 11)(3 7)(4 10)(5 9)(6 8)
(0 3)(1 7)(2 9)(4 6)(5 11)(8 10)
level 1:  1 orbit; 0 fixed; index 12
1 orbit; grpsize=48; 4 gens; 9 nodes; maxlev=3
 0:11 (12);
```

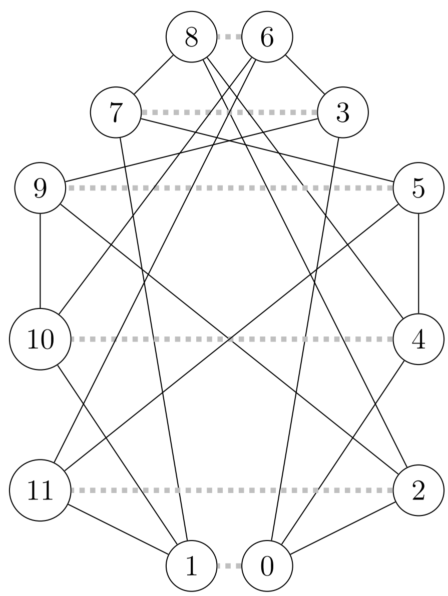

Figure 3. Graph 10. The generators used are `(0 1)(2 11)(3 7)(4 10)(5 9)(6 8)`, and the edges connecting the generating vertices are shown with gray dotted lines. The graph is symmetric about the y axis.

3 Debugging

I was under-counting the number of graphs. Below is my debugging process.

1. Generated all 5-vertex, canonically labeled[5] graphs with `geng 5 -l` and compared to my output from `gengraphs.py`. Looked to see which graphs I was missing and why. My output gave 29 graphs; nauty output gave 34 graphs. When everything is filtered, the final answer should be 13 graphs. The difference between my output and nauty's was exactly K4 with the new vertex either not connected or connected one-by-one. Since K4 has triangles, it is not one of the input graphs to my function, so this makes sense.

2. `geng 5 -lt` gave 14, canonically labeled, triangle-free graphs, including the graph with no edges. Deleting that gave a perfect matching to my output (Figure 4 on page 12), so graph generation for 5 vertices worked correctly.

3. Used nauty to generate triangle-free graphs with 6 vertices, then filtered with my script to remove K5 $\in \bar{G}$; this gave 32 graphs, as expected, so my filtering worked correctly.

   Next, used nauty to just canonically label graphs, then tried my filtering script, which still worked (therefore both 3-cycle and indep set of size 5 filtering work correctly). So the problem was in graph generation for graphs with 6 or more vertices.

4. `countedges.py` returned the correct answer for the nauty input (Listing 8 on page 13) so that was correct.

5. Found a graph that should have been in my output but was not; namely, E@hO. Found the "parent" graph of this graph, DAK. Used `compare.py` to do this (Listing 18 on page 29). The adjacency lists for these two graphs are shown in Listing 9 on page 13.

6. The binary form of DAK was correct, but the binary form of what should be E@hO was not correct, which is why E@hO was *not* produced by my script. Below are the details:

   Given a permutation $p = [1, 1, 1, 1, 1]$, below is the binary format[6] of G after appending p. The first five entries in Column 5 should all be 1s, so the two extra 0s at the start of Column 5 are incorrect; we also see that the five 1s are present, but they are shifted over by two places, which is wrong.

```
# of vertices   current columns           new column
-------------   -----------------------   ---------------
     6          1  2    3      4          5
[0,0,0,1,1,0,   0, 0,0, 0,1,0, 0,0,1,1,   0,0,1,1,1,1,1,0]
```

[5] The user must specifically tell nauty to canonically label the output with the `-l` flag; this confused me for a while since I incorrectly assumed that nauty always canonically labels the graphs!

[6] G is exactly the graph DAK in this debugging.

7. Where did these extra 0s and this shift come from? Below is G before appending p (and before updating the number of vertices in the first 6 bits):

```
G = [0,0,0,1,0,1, 0, 0,0, 0,1,0, 0,0,1,1, 0,0]
                                           ^
                                           These two 0s should NOT be here.
```

   G is on 5 vertices so $k = 5$. The correct length G should be $16 = (k)(k-1)/2+6 = (5\cdot 4)/2+6$, but it was incorrectly 18. (For $k = 4$ vertices, the correct length of G is 12, which is a multiple of 6, so the coding error did not appear until $k = 5$.)

   The error was in converting from ASCII to binary when building up G. Because the utility function `ascii2bin(c)` always produces a 6-bit output, then G is always a multiple of 6, and this creates an extraneous number of trailing zeros after G.

   This was fixed in `asciiG_2bin` (which calls `ascii2bin(c)`), because we know that the correct number of bits is exactly equal to the maximum possible number of edges e in G, and we also know that there must be 6 more bits that define the number of vertices in G. So we compute e before-hand, truncate G to length $6 + e$, and return the truncated version of G.

   This solved the under-counting error!

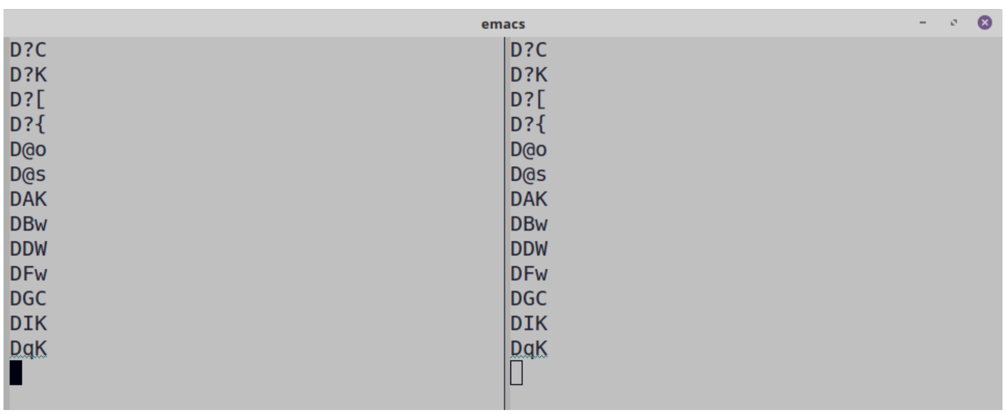

Figure 4. n5 verified. Mine on the left; nauty on the right. This should be all the correct input graphs to generate n6.

Listing 8. Confirm that my `countedges.py` script produces the correct results, given nauty input, which is known to be correct.

```
$ python countedges.py 6 nauty6.g6

edges=0, correct: 0, mine: 0, good
edges=1, correct: 0, mine: 0, good
edges=2, correct: 1, mine: 1, good
edges=3, correct: 3, mine: 3, good
edges=4, correct: 6, mine: 6, good
edges=5, correct: 8, mine: 8, good
edges=6, correct: 7, mine: 7, good
edges=7, correct: 4, mine: 4, good
edges=8, correct: 2, mine: 2, good
edges=9, correct: 1, mine: 1, good
```

Listing 9. The adjacency lists of the two graphs used in debugging. It is a coincidence that both graph are labeled "Graph 7" by nauty, since DAK can be the parent graph to other valid (3,5)-graphs.

```
DAK                  E@hO

!Graph 7.            !Graph 7.
  0 : ;              0 : 4;
  1 : 3;             1 : 5;
  2 : 4;             2 : 3 4;
  3 : 4;             3 : 5;
  4 : ;              4 : ;
                     5 : ;
```

4 Source code

(Comment: I tried to do this assignment in C but gave up after a few days and switched to Python.)

4.1 Bash scripts for high-level management and piping

Listing 10. Bash script `./run35ne.sh` to create all $(3, 5, n, e)$-graphs.

```
# Bash shell script to run the CSCI 761 HW02 assignment for (3,5,n,e)-graphs
# with k vertices where 1 <= k <= 14.
#
# We read the kth file from stdin, run it through the script that adds a vertex
# and constructs all possible connections of that the vertex to the existing
# graph, pipe the output nauty to be canonically labeled, sort with `sort`,
# delete adjacent matching lines `uniq`, and finally write to the n[k+1].g6
# output file with stdout.
#
# For `labelg`, -g sets .g6 format, and -q is quiet mode.
#
# Hannah Miller
# 2019-02-01 (started)
#
for k in {1..14}; do
    # Count the number of lines in the kth input file
    n=($(wc -l < ./35ne/n$k.g6))

    # Pretty-print to the console
    echo "For graphs with " $k " vertices, " $n " (3,5)-graphs exist."

    # Process `n[k].g6` using pipes
    python gengraphs.py $k ./35ne/n$k.g6 \
    | labelg -gq \
    | sort \
    | uniq \
    | python filterC3andK5.py $((k+1)) \
    > ./35ne/n$((k+1)).g6
done
```

Listing 11. Bash script `./run36ne.sh` to create all $(3, 6, n, e)$-graphs.

```
# Bash shell script to run the CSCI 761 HW02 assignment for (3,6,n,e)-graphs
# with k vertices where 1 <= k <= 17.
#
# Very similar to `run35ne.sh`.
#
# For `labelg`, -g sets .g6 format, and -q is quiet mode.
#
# Hannah Miller
#
for k in {1..17}; do
    # Count the number of lines in the kth input file
    n=($(wc -l < ./36ne/n$k.g6))

    # Pretty-print to the console
    echo "For graphs with " $k " vertices, " $n " (3,6)-graphs exist."

    # Process `n[k].g6` using pipes
    python gengraphs.py $k ./36ne/n$k.g6 \
    | labelg -gq \
    | sort \
    | uniq \
    | python filterC3andK6.py $((k+1)) \
    > ./36ne/n$((k+1)).g6
done
```

4.2 Scripts to generate all possible graphs and to filter the results

```
hm@dslmhm ~/Dropbox/RIT/761/hw02 $ python gengraphs.py -h
usage: Get number of vertices k and filename f. [-h] k f

positional arguments:
  k           the number of vertices k in the input graph
  f           the filename f of the input file

optional arguments:
  -h, --help  show this help message and exit
```

Figure 5. Help output for the `argparse` inputs of `gengraphs.py`. This functionality was new to me, and I liked having a command-line, UNIX-esque help for a Python script.

Listing 12. `gengraphs.py` : Generate all the graphs with $k+1$ vertices from the current input graphs with k vertices.

```python
'''See `./run.sh` in this same directory for the run instructions.

The format is described at
https://users.cecs.anu.edu.au/~bdm/data/formats.txt

Hannah Miller
2019-02-03 (started)

'''
# Import modules
import argparse
import copy
import math
import sys  # get access to stdin/stdout

# Custom modules
import genperm  # custom module to generate permutations
import utils  # custom utilities

# Set up the argument parser (not reading from stdin, but rather using this)
# From https://docs.python.org/3/library/argparse.html
parser = argparse.ArgumentParser('Get number of vertices k and filename f.')
parser.add_argument('k', type=int,
                    help='the number of vertices k in the input graph')
parser.add_argument('f', type=str,
                    help='the filename f of the input file')
args = vars(parser.parse_args())

k = args['k']
f = args['f']

```

```
k2,e,verts = utils.compute_useful_values(k)  # WORK - need e or n???

# ---------------------------------------------------------------------------
# Start computing

# Generate the permutations for the (k+1)-th new vertex and all its
# possible connectivities.  There are 2^(k+1) possible connectivities
# (counting isomorphic duplicates, which nauty will remove).
perms = genperm.genperm(k)

# Open the file and loop through each line (i.e. each graph)
the_file = open(f,'r')
for ascii_line in the_file:
    G_parent = utils.asciiG_2bin(e,ascii_line)  # convert graph to binary form

    # Overwrite the first 6 bits of G with the new number of vertices; namely,
    # k+1 vertices
    for i in range(6):
        G_parent[i] = verts[i]

    # Build all possible new graphs
    for p in perms:
        G = copy.deepcopy(G_parent)  # set G to the parent graph
        [G.append(_) for _ in p]  # append the permutation p to the graph G

        # Pad G out to closest upper multiple of 6
        b = 6*int(math.ceil(len(G)/6.0))  # number of bytes b (need 6.0 as float!)
        while len(G) < b:
            G.append(0)

        ascii_form = utils.binG_2ascii(G)  # convert graph to ASCII form
        sys.stdout.write(ascii_form)  # send to stdout to be labeled by nauty
```

Listing 13. `filterC3andK5.py` : Remove graphs G with C3 $\in G$ and K5 $\in \bar{G}$.

```
'''For canonically labeled graphs, filter out graphs with C3 in the
graph and K5 in the complement of the graph (K5 in G-complement is the
same as an independent set of size 5 in G itself).

'''
# Import modules
import argparse
import math
import sys  # get access to stdin/stdout

# Custom modules
import utils  # custom utilities
```

```python

parser = argparse.ArgumentParser('Get the number of vertices k.')
parser.add_argument('k', type=int,
          help='the number of vertices k in the input graph')
args = vars(parser.parse_args())
k = args['k']

k2,e,verts = utils.compute_useful_values(k)

# Process each ASCII line from stdin
for ascii_line in sys.stdin:
    #print("--------------------------------------------------")
    ascii_line = ascii_line.rstrip('\n')  # remove new line
    G = utils.asciiG_2bin(e,ascii_line)  # convert graph to binary form
    M = utils.graph2matrix(G)  # convert to 2D matrix form M

    # Check for cycles of size 3 and independent sets of size 5
    has3cycle = utils.check3cycles(M)
    hasindepset5 = utils.checkindepset5(M)

    # print(ascii_line)
    # print(G)
    # utils.print_matrix(M)

    if has3cycle or hasindepset5:
        # print("\nBAD")
        # print("    it is {} that {} has a 3-cycle".format(has3cycle,ascii_line))
        # print("    it is {} that {} has an indep set of size 5".format(hasindepset5,ascii_line))

        # Do NOT send this one to stdout
        continue
    else:
        # print("\ngood")
        # print("    it is {} that {} has a 3-cycle".format(has3cycle,ascii_line))
        # print("    it is {} that {} has an indep set of size 5".format(hasindepset5,ascii_line))

        # Send the ASCII version to be written to file
        sys.stdout.write(ascii_line + '\n')
```

Listing 14. `filterC3andK6.py` : Remove graphs G with C3 $\in G$ and K6 $\in \bar{G}$.

```
'''For canonically labeled graphs, filter out graphs with C3 in the
graph and K6 in the complement of the graph (K6 in G-complement is the
same as an independent set of size 6 in G itself).

'''
# Import modules
import argparse
import math
import sys  # get access to stdin/stdout

# Custom modules
import utils  # custom utilities

parser = argparse.ArgumentParser('Get the number of vertices k.')
parser.add_argument('k', type=int,
                    help='the number of vertices k in the input graph')
args = vars(parser.parse_args())
k = args['k']

k2,e,n,verts = utils.compute_useful_values(k)

# Process each ASCII line from stdin
for ascii_line in sys.stdin:
    #print("--------------------------------------------------")
    ascii_line = ascii_line.rstrip('\n')  # remove new line
    G = utils.asciiG_2bin(n,ascii_line)  # convert graph to binary form
    M = utils.graph2matrix(G)  # convert to 2D matrix form M

    # Check for cycles of size 3 (i.e. triangles).  Do triangles first
    # since that is only k^3 of brute-force checking.
    has3cycle = utils.check3cycles(M)
    if has3cycle:
        # Do NOT send this one to stdout
        continue

    # Check for independent sets of size 6 (we have already checked
    # for 3-cycles in the graph generation process)
    hasindepset6 = utils.checkindepset6(M)
    if hasindepset6:
        # Do NOT send this one to stdout
        continue

    # Otherwise, this is OK so send the ASCII version to be written to file
    sys.stdout.write(ascii_line + '\n')

    # # --------------------------------------------------
```

```

    # A,B = utils.build_auxilliary_sets(M)  # build once

    # utils.all_cliques(i,M,A,B,C)
```

4.3 Utility functions for the low-level details

Listing 15. `utils.py` : Utility functions.

```python
'''Utility functions for HW02.

'''
import math

# ------------------------------------------------------------------------------
# Convert an ASCII character `c` to binary using .g6 rules.
#
def ascii2bin(c):
  # Initialize
  k = 5  # counter k; since we index from 0, `k` starts at 5 (not 6)
  answer = [0 for i in range(6)]   # .g6 always uses 6 bits

  # Do the conversion this goes LSB to MSB
  num = ord(c) - 63       # ASCII as number subtract decimal value of 63
  while (num > 0):
      R = num % 2         # the remainder R is a coefficient of either 0 or 1
      answer[k] = R       # update the answer with this step's coefficient
      num = (num - R) / 2 # divide to prepare for the next step
      k = k-1             # decrement counter to populate from LSB to MSB

  return answer


# ------------------------------------------------------------------------------
# Convert binary to ASCII using .g6 rules.
#
def bin2ascii(b):
  k = 5  # counter k
  answer = 0  # string that will hold the computation

  # Do the conversion this goes MSB to LSB
  for i in range(6):
    answer = answer + b[i] * (2**k)
    k = k - 1  # decrement

  answer = answer + 63  # add 63 as defined by .g6
  return chr(answer)  # return as ASCII


# ------------------------------------------------------------------------------
# Given a matrix M, print it.  Used for debugging.
#
def print_matrix(M):
```

```
  k = len(M)  # number of vertices = number of rows (and columns) of M

  for i in range(k):
      print(M[i])  # the row of M


# -----------------------------------------------------------------------------
# Convert a one-line, binary form of a graph G to its 0-1 matrix form M.
#
def graph2matrix(G):
  k = bin2ascii(G[:6])  # first six bits are number of vertices in G
  k = ord(k)-63  # convert to .g6 int

  M = [[0 for i in range(k)] for j in range(k)]  # initialize
  idx = 0  # index into G

  for c in range(k):
    r = 0
    while r < c:
      M[r][c] = G[idx+6]  # must offset indexing into G by 6... very important!
      M[c][r] = G[idx+6]  # fill the lower triangle for use in `all_cliques`
      idx += 1  # increment
      r += 1  # increment

  return M


# -----------------------------------------------------------------------------
# Given matrix form M, check for 3-cycles (triangles) in G.
#
# Check if all the vertices are connected; always have the first index be larger
# than the second to enforce only looking at the upper triangle of M (i.e. i2 >
# i1 or i0)
#
def check3cycles(M):
  k = len(M)  # number of vertices = number of rows (and columns) of M
  has3cycle = False  # initialize

#  print_matrix(M)

  for i2 in range(k-2):
    for i1 in range(i2+1,k-1):

      if M[i2][i1]:  # i2 & i1 are connected so this could be part of a triangle
        for i0 in range(i1+1,k):

          if M[i2][i0] and M[i1][i0]:
            has3cycle = True  # update
```

```

  return has3cycle


# ------------------------------------------------------------------------------
# Given the matrix form M, check for independent sets of size 5 in G.
#
# This is very similar to the `check3cycles` function above.
#
def checkindepset5(M):
  k = len(M)  # number of vertices = number of rows (or columns) of M
  hasindepset5 = False  # initialize

  for i4 in range(k-4):
    for i3 in range(i4+1,k-3):

      if not M[i4][i3]:  # every non-edge ending with i3
        for i2 in range(i3+1,k-2):

          if not M[i4][i2] and not M[i3][i2]:  # every non-edge ending with i2
            for i1 in range(i2+1,k-1):

              if (not M[i4][i1] and  # every non-edge ending with i1
                  not M[i3][i1] and not M[i2][i1]):
                for i0 in range(i1+1,k):

                  if (not M[i4][i0] and  # every non-edge ending with i0
                      not M[i3][i0] and not M[i2][i0] and not M[i1][i0]):
                    hasindepset5 = True

                    # print("sum is {}".format(M[i4][i3] + \
                    #     M[i4][i2] + M[i3][i2] + \
                    #     M[i4][i1] + M[i3][i1] + M[i2][i1] + \
                    #     M[i1][i0] + M[i4][i0] + M[i3][i0] + M[i2][i0]))
                    # print(i4,i3,i2,i1,i0)

  return hasindepset5


# ------------------------------------------------------------------------------
# Given the matrix form M, check for independent sets of size 6 in G.
#
def checkindepset6(M):
  k = len(M)  # number of vertices = number of rows (or columns) of M
  hasindepset6 = False  # initialize

  for i5 in range(k-5):
    for i4 in range(i5+1,k-4):
```

```

      if not M[i5][i4]:
        for i3 in range(i4+1,k-3):

          if not M[i5][i3] and not M[i4][i3]:
            for i2 in range(i3+1,k-2):

              if not M[i5][i2] and not M[i4][i2] and not M[i3][i2]:
                for i1 in range(i2+1,k-1):

                  if (not M[i5][i1] and not M[i4][i1] and not M[i3][i1] and
                      not M[i2][i1]):
                    for i0 in range(i1+1,k):

                      if (not M[i5][i0] and not M[i4][i0] and not M[i3][i0] and
                          not M[i2][i0] and not M[i1][i0]):
                        hasindepset6 = True
                        return hasindepset6  # exit early

  return hasindepset6


# ------------------------------------------------------------------------------
# Given the matrix form M, for all vertices v, build auxilliary sets A
# (adjacency list of v) and B (vertices numbered greater than v).
#
def build_auxilliary_sets(M):
  k = len(M)  # number of vertices = number of rows (and columns) of M

  # Initialize
  A = [[] for _ in range(k)]
  B = [[] for _ in range(k)]

  # Build A and B over all vertices v
  for v in range(k):

    for j in range(k):
      if M[v][j] == 1:
        A[v].append(j)

    B[v] = range(v+1,k)  # vertices with labels greater than v

  # Convert both A and B to a list of Python sets so we can do intersections
  # more easily later
  A = [set(_) for _ in A]
  B = [set(_) for _ in B]

  return A,B
```

```
# # -----------------------------------------------------------------------------
# # Given the matrix form M, auxilliary sets A and B from `build_auxilliary_sets`,
# # and set of choices C, use the algorithm from page 113 of the book to find all
# # the cliques exactly once.
# #
# # The algorithm will end early if it finds a clique of size 3 or of size WORK.
# #
# def all_cliques(i,M,A,B,C):
#   print("---------------------------------------------------")
#   k = len(M)  # number of vertices = number of rows (and columns) of M

#   if i == 0:
#     V = set(0:k)  # 0:k is all the vertices V in M
#     N[i] = V
#     C[i] = V

#     return set([])



#   else:
#     N[i] = A[i-1].intersection(N[i-1])

#     # If N[i] is empty, then A[i-1] is a maximal clique

#     AnB = A[i-1].intersection(B[i-1])
#     AnBnC = AnB.intersection(C[i-1])
#     C[i] = AnBnC


#     for x in C[i]:
#       xi = x
#       all_cliques(i,M,A,B,C)

#     X = set([0:(i-1)])
#     return X




# -----------------------------------------------------------------------------
# Given number of input vertices k, compute some useful values.
#
def compute_useful_values(k):
  k2 = k+1  # `k2` is the total number of vertices when we are done
  e = (k)*(k-1)/2  # max number of edges e for k vertices
```

```python

  # Calculate binary form of N(n), where n = k2 = k+1 (and k is the input number
  # of vertices)
  verts = ascii2bin(chr(63 + k2))

  return k2,e,verts


# ---------------------------------------------------------------------------
# Convert an ASCII form of a graph G to its binary form.
# Must remove new line `\n`, or else `\n` will get converted into a (fake)
# character, which will throw off the indexing by +6 in the ASCII to binary
# conversion.  Convert each character in the line to its .g6 definition.
#
# `e` is the max number of edges in G, and `ascii_line` is the .g6 form of G.
#
def asciiG_2bin(e,ascii_line):
  # Build up G
  G = []  # initialize
  for c in ascii_line.rstrip('\n'):  # `c` is the current character
    char_as_bin = ascii2bin(c)  # convert ASCII -> binary
    [G.append(cc) for cc in char_as_bin]  # put the binary results into G

  # Because `ascii2bin(c)` always produces a 6-bit output, then G is always a
  # multiple of 6, and there may be extraneous number of trailing zeros after G
  # (this was a subtle but major error!).  However, we know that the correct
  # number of bits is exactly equal to the maximum possible number of edges e in
  # G, and we also know that there must be 6 more bits that define the number of
  # vertices in G.  So we compute that value (namely, `6+e`), truncate G to that
  # length, and return it.
  return G[:(6+e)]


# ---------------------------------------------------------------------------
# Convert a binary form of a graph G to its ASCII form.
#
def binG_2ascii(G):
  ascii_form = []  # initialize
  jj = 0  # index into the ASCII form

  for i in range(0,len(G),6):  # use stride length of 6
    ascii_form.append(bin2ascii(G[i:i+6]))  # convert binary -> ASCII

  ascii_form = ''.join(ascii_form)  # join the list of characters
  ascii_form = ascii_form + '\n'  # need a new line so stdout plays nicely

  return ascii_form
```

Listing 16. `genperm.py` : For a given k, generate all permutations of 0s and 1s of length k.

```python
'''Given a value k, generate all permutations of 0s and 1s of length k.  Lots of
copying, looping, and general inefficiency in this function, but it generates up
to k=17 in ~3 seconds (checked with `time python genperm.py`), so it is fine.

'''
import copy

def genperm(k):
    perm = [[0], [1]]  # the base case (k = 0)

    # Create the permutations
    for i in range(1,k):
        # Copy.  (Note: Need to use `copy.deepcopy` to create a true copy of the
        # lists, or else modifying any of the perm, perm0, and perm1 lists will
        # modify the others, which is not what we want!)
        perm0 = copy.deepcopy(perm)  # 0s will be appended to this list
        perm1 = copy.deepcopy(perm)  # 1s will be appended to this list

        # Append 0s and 1s, respectively; use `_` as a throwaway variable
        [perm0[_].append(0) for _ in range(len(perm0))]
        [perm1[_].append(1) for _ in range(len(perm1))]

        # Join the results to use in the next layer of the generation
        perm = perm0 + perm1

    perm.sort()  # sort into lexicographic order

    # # Sanity check of the final result
    # print("k={:02d} : it is {} that expected & actual lengths are equal".format(
    #     k, 2**k == len(perm)))

    # # Write to file
    # with open('./perms/{:02d}.txt'.format(k), 'w') as f:
    #     for p in perm:
    #         f.write("%s\n" % p)

    return perm

### Test
# genperm(17)

### Used to write all permutations to a file
# for k in range(4,18):
#     print("k = {}".format(k))
#     genperm(k)
```

Listing 17. Permutations output for $k = 4$ from `genperm.py`. This is correct and can easily be checked by inspection.

```
[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 1, 0]
[0, 0, 1, 1]
[0, 1, 0, 0]
[0, 1, 0, 1]
[0, 1, 1, 0]
[0, 1, 1, 1]
[1, 0, 0, 0]
[1, 0, 0, 1]
[1, 0, 1, 0]
[1, 0, 1, 1]
[1, 1, 0, 0]
[1, 1, 0, 1]
[1, 1, 1, 0]
[1, 1, 1, 1]
```

4.4 Scripts to count the number of edges in each graph

Listing 18. `compare.py` : Compare my output vs. nauty output.

```
'''Compare my output vs. nauty output.

To run:
    python compare.py

'''
# Enter my output and nauty's output as Python `sets`

mine = set(['EAIW', 'EAN_', 'EC\o', 'E?d_', 'E?D_', 'E?dg', 'E?Dg', 'E?F_',
            'E?Fg', 'E@?G', 'E?GW', 'E@GW', 'E@hW', 'EIGW', 'E_lo', 'E?lo',
            'E?Lo', 'E?No', 'E?NO', 'E?^o', 'E?~o', 'E?\o', 'E?So',
])

nauty = set([
    'E???', 'E??G', 'E??W', 'E??w', 'E?@w', 'E?Bw', 'E?D_', 'E?Dg', 'E?F_',
    'E?Fg', 'E?GW', 'E?Lo', 'E?NO', 'E?No', 'E?So', 'E?\o', 'E?^o', 'E?d_',
    'E?dg', 'E?lo', 'E?~o', 'E@?G', 'E@GW', 'E@hO', 'E@hW', 'EAIW', 'EAN_',
    'ECXo', 'EC\o', 'EIGW', 'ES\o', 'E_GW', 'E_lo', 'E`?G', 'E`GW', 'E`dg',
    'EoSo', 'Es\o',
])

# Perform some set operations
both = mine.union(nauty)
justme = mine.difference(nauty)
justnauty = nauty.difference(mine)

# Print the results
print("\n\nboth")
print(both)

print("\n\njust me")
print(justme)

print("\n\njust nauty")
print(justnauty)
```

Listing 19. `countedges.py` : Count number of edges in G. Used for debugging and for printing the table of edges vs. number of vertices.

```
'''Count number of edges in G.  Used for debugging and for printing the table of
edges vs. number of vertices.

Run with this:
    python countedges.py 6 n6.g6
```

```

'''
# ------------------------------------------------------------------------------
# Set up

# Import modules
import argparse
import math
import sys  # get access to stdin/stdout

# Custom modules
import genperm  # custom module to generate permutations
import utils  # custom utilities

# Set up the argument parser (not reading from stdin, but rather using this)
# From https://docs.python.org/3/library/argparse.html
parser = argparse.ArgumentParser('Get number of vertices k and filename f.')
parser.add_argument('k', type=int,
                    help='the number of vertices k in the input graph')
parser.add_argument('f', type=str,
                    help='the filename f of the input file')
args = vars(parser.parse_args())

k = args['k']
f = args['f']

k2,e,verts = utils.compute_useful_values(k)

# ------------------------------------------------------------------------------
# Start computing

# # Set up correct answers for checking (starting at 0 edges) on (3,5)-graphs
# if k == 5:
#     correct_num_of_graphs = [0,1,2,3,4,2,1]
# elif k == 6:
#     correct_num_of_graphs = [0,0,1,3,6,8,7,4,2,1]  # graphs with 6 vertices
# else:
#     print("unsupported number of vertices")

num_of_edges = []  # initialize

# Open the file and loop through each line (i.e. each graph)
the_file = open(f,'r')
for ascii_line in the_file:
    G = utils.asciiG_2bin(e,ascii_line)  # convert graph to binary form
    G = G[6:]  # remove the first 6 bits (the number of vertices) from G
    num_of_edges.append(sum(G))  # count number of edges

```

```
for i in range(len(correct_num_of_graphs)):
    my_num_graphs = num_of_edges.count(i)

    if my_num_graphs == correct_num_of_graphs[i]:
        status = 'good'
    else:
        status = ' BAD'

    print("edges={}, correct: {}, mine: {}, {}".format(
        i, correct_num_of_graphs[i], my_num_graphs, status))
```

Listing 20. `createtable.py` : Produce the table of edges vs. number of vertices.

```
'''Given a directory d, number of vertices n, and number of edges e, read in
each file and produce the table of edges vs. number of vertices.

To run:
    Update the inputs into the script.
    python createtable.py

'''
# Define inputs
d = '/home/hm/Dropbox/RIT/761/hw02/35ne/'
n = 14  # number of vertices n
e = 26+1  # number of edges e (add +1 to include 0 edges)

# ------------------------------------------------------------------------------

# Import modules
import os
import pandas as pd

# Custom modules
import utils

# Initialize list of lists to hold the results
A = [['' for _ in range(n)] for __ in range(e)]

# Loop through the files
for k in range(1,n):
    k2,e,verts = utils.compute_useful_values(k)

    num_of_edges = []  # initialize
    f = os.path.join(d, "n" + str(k) + ".g6")
    the_file = open(f,'r')

    for ascii_line in the_file:
```

```
        #print(ascii_line.rstrip('\n'))
        G = utils.asciiG_2bin(e,ascii_line)  # convert graph to binary form
        G = G[6:]  # remove the first 6 bits (the number of vertices) from G

        # Count number of edges on k vertices by summing 1s in the current graph
        num_of_edges.append(sum(G))

    # Find distribution of the number of edges on k vertices (the looping over
    # `count` every time is inefficient, but the performance is fine for the
    # purposes of this script)
    if len(num_of_edges) > 0:  # list is not empty
        for ee in range(max(num_of_edges)+1):  # loop through all edges ee
            v = num_of_edges.count(ee)  # count how many graphs have ee edges
            if v != 0:
                A[ee][k] = v

# ------------------------------------------------------------------------------
# Output as LaTeX table

df = pd.DataFrame(A)  # convert to Pandas dataframe
df = df.drop(columns=[0], axis=1)  # remove the first column (0 vertices)

print(df.to_latex())
```