OEIS: Maximal Cliques of Authors from Comments

ADVANCED ALGORITHMS AND GRAPH MINING

Professors:
Andrea Marino
Massimo Nocentini

Student:
Paula Mihalcea

July 2021

Contents

1 Introduction

OEIS is the online encyclopaedia of **integer sequences**. It lists thousands of number sequences in lexicographic order, such as the prime numbers or the Fibonacci sequence, easing the work of countless researchers since 1964, its foundation year.

The OEIS is made of a series of **JSON files**, one for each integer sequence. Given their regular, human-readable format, these files can be easily manipulated in order to further analyze them. Indeed, each page of the OEIS not only lists the integers of the corresponding sequence, but also a series of information such as formulas, references, links and comments.

This work aims to create, step-by-step, a **Python 3** script capable of loading these files and parsing their content in order to build a **graph** where:

- **nodes** represent all unique **authors** that can be found in each comment of every sequence, and
- **edges** link two authors who have **commented the same sequence**.

Three main algorithms are then implemented in order to find:

1. a **maximal clique**;
2. a list of **all maximal cliques**;
3. the **maximum clique**.

The library of choice for creating the graph is **NetworkX**, a fast Python module for the creation, manipulation, and study of the structure of complex networks. Other libraries such as itertools, NumPy, os and random are also used for efficiency purposes, as they provide highly optimized functions. The complete list of packages can be found in section 2.

2 Requirements

Before starting, a series of packages must be installed for the subsequent code to be executable. The simplest way is to use `pip`, a package manager for Python callable from the system terminal.

The commands needed for this operation are listed in the following cell; the Jupyter magic function `%%cmd` (`%%bash` for Unix users) at the beginning allows to use it as a terminal. Make sure to follow the recommended install order, as it helps avoiding errors which can sometimes be generated by different versions of the packages.

Note: the `argparse` package is only needed for the execution of the `mihalcea.py` script, and is not necessary for the current Jupyter notebook.

[1]:
```
%%cmd

pip install numpy
pip install networkx
pip install tqdm
pip install argparse
```

The freshly installed modules can be now used by simply importing them, along with other native Python packages:

[2]:
```
import argparse # Only needed for the mihalcea.py script
import itertools as its
import json
import networkx as nx
import numpy as np
import os
import random
import re
import timeit
import tqdm
import warnings
```

3 Sequence files

Having installed the required packages, we can now proceed with analyzing the files.

The raw OEIS sequence files can be found in `data/sequences`. We can start by writing a function capable of opening one of them using the JSON package available in Python, and use it to load a file's content as a Python dict, then print it:

[3]:
```
def load_json(file_path, print_result=False):
    try:
        with open(file_path, 'r') as file:
            raw_data = json.load(file)
```

```
            if print_result:
                print('File ' + file_path.split('/')[-1] + ' contents:')
                print()
                print(json.dumps(raw_data, indent=True))
                print()
                print(
                    'The \'json\' Python module returns a dictionary, which can be
  confirmed by invoking the \'type\' function on the loaded data: ' + str(
                        type(raw_data)) + '.')
                print('This dictionary\'s keys are: ' + str(raw_data.keys()).
  replace('dict_keys([', '').replace('])',

                        '') + '.')
            return raw_data
        except OSError:
            print('Could not open file: {}, exiting program.'.format(file_path.split('/
  ')[-1]))
```

Note that this function correctly **handles input/output errors**, and can be used to **return a file's content** as a Python **dictionary** even without printing it, by either omitting the `print_result` argument or setting it to `False`.

We can thus view the first JSON file and its keys:

[4]:
```
# Load sample sequence file
print('\n' + 'Printing sample OEIS JSON file...')

file = load_json('data/sequences/A000001.json', print_result=True)
```

```
Printing sample OEIS JSON file...
File A000001.json contents:

{
 "greeting": "Greetings from The On-Line Encyclopedia of Integer Sequences!
http://oeis.org/",
 "query": "id:A000001",
 "count": 1,
 "start": 0,
 "results": [
  {
   "number": 1,
   "id": "M0098 N0035",
   "data": "0,1,1,1,2,1,2,1,5,2,2,1,5,1,2,1,14,1,5,1,5,2,2,1,15,2,2,5,4,1,4,1,51,1,2,1,14
,1,2,2,14,1,6,1,4,2,2,1,52,2,5,1,5,1,15,2,13,2,2,1,13,1,2,4,267,1,4,1,5,1,4,1,50,1,2,3,4,
1,6,1,52,15,2,1,15,1,2,1,12,1,10,1,4,2",
   "name": "Number of groups of order n.",
   "comment": [
    "Also, number of nonisomorphic subgroups of order n in symmetric group S_n. - _Lekraj
Beedassy_, Dec 16 2004",
    "Also, number of nonisomorphic primitives of the combinatorial species Lin[n-1]. -
_Nicolae Boicu_, Apr 29 2011",
```

"The record values are (A046058): 1, 2, 5, 14, 15, 51, 52, 267, 2328, 56092, 10494213, 49487365422, ..., and they appear at positions (A046059): 1, 4, 8, 16, 24, 32, 48, 64, 128, 256, 512, 1024, ... _Robert G. Wilson v_, Oct 12 2012",
"In (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008), a(n) is called the \"group number of n\", denoted by gnu(n), and the first occurrence of k is called the \"minimal order attaining k\", denoted by moa(k) (see A046057). - _Daniel Forgues_, Feb 15 2017",
"It is conjectured in (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008) that the sequence n -> a(n) -> a(a(n)) = a^2(n) -> a(a(a(n))) = a^3(n) -> ... -> consists ultimately of 1s, where a(n), denoted by gnu(n), is called the \"group number of n\". - _Muniru A Asiru_, Nov 19 2017",
"MacHale (2020) shows that there are infinitely many values of n for which there are more groups than rings of that order (cf. A027623). He gives n = 36355 as an example. It would be nice to have enough values of n to create an OEIS entry for them. - _N. J. A. Sloane_, Jan 02 2021"
],
"reference": [
"S. R. Blackburn, P. M. Neumann, and G. Venkataraman, Enumeration of Finite Groups, Cambridge, 2007.",
"L. Comtet, Advanced Combinatorics, Reidel, 1974, p. 302, #35.",
"J. H. Conway et al., The Symmetries of Things, Peters, 2008, p. 209.",
"H. S. M. Coxeter and W. O. J. Moser, Generators and Relations for Discrete Groups, 4th ed., Springer-Verlag, NY, reprinted 1984, p. 134.",
"CRC Standard Mathematical Tables and Formulae, 30th ed. 1996, p. 150.",
"R. L. Graham, D. E. Knuth and O. Patashnik, Concrete Mathematics, A Foundation for Computer Science, Addison-Wesley Publ. Co., Reading, MA, 1989, Section 6.6 'Fibonacci Numbers' pp. 281-283.",
"M. Hall, Jr. and J. K. Senior, The Groups of Order 2^n (n <= 6). Macmillan, NY, 1964.",
"D. Joyner, 'Adventures in Group Theory', Johns Hopkins Press. Pp. 169-172 has table of groups of orders < 26.",
"D. S. Mitrinovic et al., Handbook of Number Theory, Kluwer, Section XIII.24, p. 481.",
"M. F. Newman and E. A. O'Brien, A CAYLEY library for the groups of order dividing 128. Group theory (Singapore, 1987), 437-442, de Gruyter, Berlin-New York, 1989.",
"N. J. A. Sloane, A Handbook of Integer Sequences, Academic Press, 1973 (includes this sequence).",
"N. J. A. Sloane and Simon Plouffe, The Encyclopedia of Integer Sequences, Academic Press, 1995 (includes this sequence)."
],
"link": [
"H.-U. Besche and Ivan Panchenko, Table of n, a(n) for n = 0..2047 [Terms 1 through 2015 copied from Small Groups Library mentioned below. Terms 2016 - 2047 added by Ivan Panchenko, Aug 29 2009. 0 prepended by _Ray Chandler_, Sep 16 2015.]",
"H. A. Bender, A determination of the groups of order p^5, Ann. of Math. (2) 29, pp. 61-72 (1927).",
"Hans Ulrich Besche and Bettina Eick, Construction of finite groups, Journal of Symbolic Computation, Vol. 27, No. 4, Apr 15 1999, pp. 387-404.",
"Hans Ulrich Besche and Bettina Eick, The groups of order at most 1000 except

512 and 768, Journal of Symbolic Computation, Vol. 27, No. 4, Apr 15 1999, pp. 405-413.",
"H. U. Besche, B. Eick and E. A. O'Brien, The groups of order at most 2000, Electron. Res. Announc. Amer. Math. Soc. 7 (2001), 1-4.",
"H. U. Besche, B. Eick and E. A. O'Brien, The Small Groups Library",
"H. U. Besche, B. Eick and E. A. O'Brien, Number of isomorphism types of finite groups of given order",
"H.-U. Besche, B. Eick and E. A. O'Brien, A Millennium Project: Constructing Small Groups, Internat. J. Algebra and Computation, 12 (2002), 623-644.",
"H. Bottomley, Illustration of initial terms",
"J. H. Conway, Heiko Dietrich and E. A. O'Brien, Counting groups: gnus, moas and other exotica, Math. Intell., Vol. 30, No. 2, Spring 2008.",
"Yang-Hui He and Minhyong Kim, Learning Algebraic Structures: Preliminary Investigations, arXiv:1905.02263 [cs.LG], 2019.",
"Otto H\u00f6lder, Die Gruppen der Ordnungen p^3, pq^2, pqr, p^4, Math. Ann. 43 pp. 301-412 (1893).",
"Rodney James, The groups of order p^6 (p an odd prime), Math. Comp. 34 (1980), 613-637.",
"Rodney James and John Cannon, Computation of isomorphism classes of p-groups, Mathematics of Computation 23.105 (1969): 135-140.",
"Desmond MacHale, Are There More Finite Rings than Finite Groups?, Amer. Math. Monthly (2020) Vol. 127, Issue 10, 936-938.",
"G. A. Miller, Determination of all the groups of order 64, Amer. J. Math., 52 (1930), 617-634.",
"Ed Pegg, Jr., Sequence Pictures, Math Games column, Dec 08 2003.",
"Ed Pegg, Jr., Sequence Pictures, Math Games column, Dec 08 2003 [Cached copy, with permission (pdf only)]",
"D. S. Rajan, The equations D^kY=X^n in combinatorial species, Discrete Mathematics 118 (1993) 197-206 North-Holland.",
"E. Rodemich, The groups of order 128, J. Algebra 67 (1980), no. 1, 129-142.",
"Gordon Royle, Combinatorial Catalogues. See subpage \"Generators of small groups\" for explicit generators for most groups of even order < 1000.",
"D. Rusin, Asymptotics [Cached copy of lost web page]",
"Eric Weisstein's World of Mathematics, Finite Group",
"Wikipedia, Finite group",
"M. Wild, The groups of order sixteen made easy, Amer. Math. Monthly, 112 (No. 1, 2005), 20-31.",
"Gang Xiao, <a href=\"http://wims.unice.fr/~wims/wims.cgi?module=tool/algebra/smallgr

```
oup\">SmallGroup</a>",
    "<a href=\"/index/Gre#groups\">Index entries for sequences related to groups</a>",
    "<a href=\"/index/Cor#core\">Index entries for \"core\" sequences</a>"
   ],
   "formula": [
    "From _Mitch Harris_, Oct 25 2006: (Start)",
    "For p, q, r primes:",
    "a(p) = 1, a(p^2) = 2, a(p^3) = 5, a(p^4) = 14, if p = 2, otherwise 15.",
    "a(p^5) = 61 + 2*p + 2*gcd(p-1,3) + gcd(p-1,4), p >= 5, a(2^5)=51, a(3^5)=67.",
    "a(p^e) ~ p^((2/27)e^3 + O(e^(8/3)))",
    "a(pq) = 1 if gcd(p,q-1) = 1, 2 if gcd(p,q-1) = p. (p < q)",
    "a(pq^2) = one of the following:",
    "* 5, p=2, q odd,",
    "* (p+9)/2, q=1 mod p, p odd,",
    "* 5, p=3, q=2,",
    "* 3, q = -1 mod p, p and q odd.",
    "* 4, p=1 mod q, p > 3, p != 1 mod q^2",
    "* 5, p=1 mod q^2",
    "* 2, q != +/-1 mod p and p != 1 mod q,",
    "a(pqr) (p < q < r) = one of the following:",
    "* q==1 mod p r==1 mod p r==1 mod q a(pqr)",
    "* No...No...No...1",
    "* No...No...Yes...2",
    "* No...Yes...No...2",
    "* No...Yes...Yes...4",
    "* Yes...No...No...2",
    "* Yes...No...Yes...3",
    "* Yes...Yes...No...p+2",
    "* Yes...Yes...Yes...p+4 (table from Derek Holt) (End)",
    "a(n) = A000688(n) + A060689(n). - _R. J. Mathar_, Mar 14 2015"
   ],
   "example": [
    "Groups of orders 1 through 10 (C_n = cyclic, D_n = dihedral of order n, Q_8 =
quaternion, S_n = symmetric):",
    "1: C_1",
    "2: C_2",
    "3: C_3",
    "4: C_4, C_2 X C_2",
    "5: C_5",
    "6: C_6, S_3=D_6",
    "7: C_7",
    "8: C_8, C_4 X C_2, C_2 X C_2 X C_2, D_8, Q_8",
    "9: C_9, C_3 X C_3",
    "10: C_10, D_10"
   ],
   "maple": [
    "GroupTheory:-NumGroups(n); # with(GroupTheory); loads this command - _N. J. A.
Sloane_, Dec 28 2017"
   ],
   "mathematica": [
    "FiniteGroupCount[Range[100]] (* _Harvey P. Dale_, Jan 29 2013 *)",
    "a[ n_] := If[ n < 1, 0, FiniteGroupCount @ n]; (* _Michael Somos_, May 28 2014 *)"
```

```
   ],
   "program": [
    "(MAGMA) D:=SmallGroupDatabase(); [ NumberOfSmallGroups(D, n) : n in [1..1000] ]; //
_John Cannon_, Dec 23 2006",
    "(GAP) A000001 := Concatenation([0], List([1..500], n -> NumberSmallGroups(n))); #
_Muniru A Asiru_, Oct 15 2017"
   ],
   "xref": [
    "The main sequences concerned with group theory are A000001 (this one), A000679,
A001034, A001228, A005180, A000019, A000637, A000638, A002106, A005432, A000688, A060689,
A051532.",
    "Cf. A046058, A023675, A023676. A003277 gives n for which A000001(n) = 1, A063756
(partial sums).",
    "A046057 gives first occurrence of each k.",
    "A027623 gives the number of rings of order n."
   ],
   "keyword": "nonn,core,nice,hard",
   "offset": "0,5",
   "author": "_N. J. A. Sloane_",
   "ext": [
    "More terms from _Michael Somos_",
    "Typo in b-file description fixed by _David Applegate_, Sep 05 2009"
   ],
   "references": 156,
   "revision": 191,
   "time": "2021-03-27T03:48:47-04:00",
   "created": "1991-04-30T03:00:00-04:00"
  }
 ]
}
```


```
The 'json' Python module returns a dictionary, which can be confirmed by invoking the
'type' function on the loaded data: <class 'dict'>.
This dictionary's keys are: 'greeting', 'query', 'count', 'start', 'results'.
```

As mentioned before, each sequence file contains additional information, specifically:

- a simple `greeting`;
- a `query`, containing the sequence's ID;
- `count`;
- `start`;
- `results`, which contains a list with another dictionary as its first element.

It can be seen from this file's content that the most relevant information is actually found in the `results` **sub-dictionary**, which can be easily accessed with:

[5]:
```
# Print 'results' section of the sample sequence
print('\n' + 'Printing sample file "results" section...' + '\n')

results = file.get('results')
if results:
```

```
    print(json.dumps(results[0], indent=True))
else:
    print('No "results" section found.')
```

```
Printing sample file "results" section...

{
 "number": 1,
 "id": "M0098 N0035",
 "data": "0,1,1,1,2,1,2,1,5,2,2,1,5,1,2,1,14,1,5,1,5,2,2,1,15,2,2,5,4,1,4,1,51,1,2,1,14,1
,2,2,14,1,6,1,4,2,2,1,52,2,5,1,5,1,15,2,13,2,2,1,13,1,2,4,267,1,4,1,5,1,4,1,50,1,2,3,4,1,
6,1,52,15,2,1,15,1,2,1,12,1,10,1,4,2",
 "name": "Number of groups of order n.",
 "comment": [
  "Also, number of nonisomorphic subgroups of order n in symmetric group S_n. - _Lekraj
Beedassy_, Dec 16 2004",
  "Also, number of nonisomorphic primitives of the combinatorial species Lin[n-1]. -
_Nicolae Boicu_, Apr 29 2011",
  "The record values are (A046058): 1, 2, 5, 14, 15, 51, 52, 267, 2328, 56092, 10494213,
49487365422, ..., and they appear at positions (A046059): 1, 4, 8, 16, 24, 32, 48, 64,
128, 256, 512, 1024, ... _Robert G. Wilson v_, Oct 12 2012",
  "In (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008), a(n) is called the \"group
number of n\", denoted by gnu(n), and the first occurrence of k is called the \"minimal
order attaining k\", denoted by moa(k) (see A046057). - _Daniel Forgues_, Feb 15 2017",
  "It is conjectured in (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008) that the
sequence n -> a(n) -> a(a(n)) = a^2(n) -> a(a(a(n))) = a^3(n) -> ... -> consists
ultimately of 1s, where a(n), denoted by gnu(n), is called the \"group number of n\". -
_Muniru A Asiru_, Nov 19 2017",
  "MacHale (2020) shows that there are infinitely many values of n for which there are
more groups than rings of that order (cf. A027623). He gives n = 36355 as an example. It
would be nice to have enough values of n to create an OEIS entry for them. - _N. J. A.
Sloane_, Jan 02 2021"
 ],
 "reference": [
  "S. R. Blackburn, P. M. Neumann, and G. Venkataraman, Enumeration of Finite Groups,
Cambridge, 2007.",
  "L. Comtet, Advanced Combinatorics, Reidel, 1974, p. 302, #35.",
  "J. H. Conway et al., The Symmetries of Things, Peters, 2008, p. 209.",
  "H. S. M. Coxeter and W. O. J. Moser, Generators and Relations for Discrete Groups, 4th
ed., Springer-Verlag, NY, reprinted 1984, p. 134.",
  "CRC Standard Mathematical Tables and Formulae, 30th ed. 1996, p. 150.",
  "R. L. Graham, D. E. Knuth and O. Patashnik, Concrete Mathematics, A Foundation for
Computer Science, Addison-Wesley Publ. Co., Reading, MA, 1989, Section 6.6 'Fibonacci
Numbers' pp. 281-283.",
  "M. Hall, Jr. and J. K. Senior, The Groups of Order 2^n (n <= 6). Macmillan, NY,
1964.",
  "D. Joyner, 'Adventures in Group Theory', Johns Hopkins Press. Pp. 169-172 has table of
groups of orders < 26.",
  "D. S. Mitrinovic et al., Handbook of Number Theory, Kluwer, Section XIII.24, p. 481.",
  "M. F. Newman and E. A. O'Brien, A CAYLEY library for the groups of order dividing 128.
Group theory (Singapore, 1987), 437-442, de Gruyter, Berlin-New York, 1989.",
```

```
  "N. J. A. Sloane, A Handbook of Integer Sequences, Academic Press, 1973 (includes this
sequence).",
  "N. J. A. Sloane and Simon Plouffe, The Encyclopedia of Integer Sequences, Academic
Press, 1995 (includes this sequence)."
 ],
 "link": [
  "H.-U. Besche and Ivan Panchenko, <a href=\"/A000001/b000001.txt\">Table of n, a(n) for
n = 0..2047</a> [Terms 1 through 2015 copied from Small Groups Library mentioned below.
Terms 2016 - 2047 added by Ivan Panchenko, Aug 29 2009. 0 prepended by _Ray Chandler_,
Sep 16 2015.]",
  "H. A. Bender, <a href=\"http://www.jstor.org/stable/1967981\">A determination of the
groups of order p^5</a>, Ann. of Math. (2) 29, pp. 61-72 (1927).",
  "Hans Ulrich Besche and Bettina Eick, <a
href=\"http://dx.doi.org/10.1006/jsco.1998.0258\">Construction of finite groups</a>,
Journal of Symbolic Computation, Vol. 27, No. 4, Apr 15 1999, pp. 387-404.",
  "Hans Ulrich Besche and Bettina Eick, <a
href=\"http://dx.doi.org/10.1006/jsco.1998.0259\">The groups of order at most 1000 except
512 and 768</a>, Journal of Symbolic Computation, Vol. 27, No. 4, Apr 15 1999, pp.
405-413.",
  "H. U. Besche, B. Eick and E. A. O'Brien, <a
href=\"http://www.ams.org/era/2001-07-01/S1079-6762-01-00087-7/home.html\">The groups of
order at most 2000</a>, Electron. Res. Announc. Amer. Math. Soc. 7 (2001), 1-4.",
  "H. U. Besche, B. Eick and E. A. O'Brien, <a href=\"http://www.icm.tu-
bs.de/ag_algebra/software/small/\">The Small Groups Library</a>",
  "H. U. Besche, B. Eick and E. A. O'Brien, <a href=\"http://www.icm.tu-
bs.de/ag_algebra/software/small/number.html\">Number of isomorphism types of finite
groups of given order</a>",
  "H.-U. Besche, B. Eick and E. A. O'Brien, <a
href=\"http://dx.doi.org/10.1142/S0218196702001115\">A Millennium Project: Constructing
Small Groups</a>, Internat. J. Algebra and Computation, 12 (2002), 623-644.",
  "H. Bottomley, <a href=\"/A000001/a000001.gif\">Illustration of initial terms</a>",
  "J. H. Conway, Heiko Dietrich and E. A. O'Brien, <a
href=\"http://www.math.auckland.ac.nz/~obrien/research/gnu.pdf\">Counting groups: gnus,
moas and other exotica</a>, Math. Intell., Vol. 30, No. 2, Spring 2008.",
  "Yang-Hui He and Minhyong Kim, <a href=\"https://arxiv.org/abs/1905.02263\">Learning
Algebraic Structures: Preliminary Investigations</a>, arXiv:1905.02263 [cs.LG], 2019.",
  "Otto H\u00f6lder, <a href=\"http://dx.doi.org/10.1007/BF01443651\">Die Gruppen der
Ordnungen p^3, pq^2, pqr, p^4</a>, Math. Ann. 43 pp. 301-412 (1893).",
  "Rodney James, <a href=\"http://dx.doi.org/10.1090/S0025-5718-1980-0559207-0\">The
groups of order p^6 (p an odd prime)</a>, Math. Comp. 34 (1980), 613-637.",
  "Rodney James and John Cannon, <a
href=\"http://dx.doi.org/10.1090/S0025-5718-1969-0238953-8\">Computation of isomorphism
classes of p-groups</a>, Mathematics of Computation 23.105 (1969): 135-140.",
  "Desmond MacHale, <a href=\"https://doi.org/10.1080/00029890.2020.1820790\">Are There
More Finite Rings than Finite Groups?</a>, Amer. Math. Monthly (2020) Vol. 127, Issue 10,
936-938.",
  "G. A. Miller, <a href=\"http://www.jstor.org/stable/2370630\">Determination of all the
groups of order 64</a>, Amer. J. Math., 52 (1930), 617-634.",
  "Ed Pegg, Jr., <a href=\"http://www.mathpuzzle.com/MAA/07-Sequence%20Pictures/mathgames
_12_08_03.html\">Sequence Pictures</a>, Math Games column, Dec 08 2003.",
  "Ed Pegg, Jr., <a href=\"/A000043/a000043_2.pdf\">Sequence Pictures</a>, Math Games
column, Dec 08 2003 [Cached copy, with permission (pdf only)]",
```

```
  "D. S. Rajan, <a href=\"http://dx.doi.org/10.1016/0012-365X(93)90061-W\">The equations
D^kY=X^n in combinatorial species</a>, Discrete Mathematics 118 (1993) 197-206 North-
Holland.",
  "E. Rodemich, <a href=\"http://dx.doi.org/10.1016/0021-8693(90)90244-I\">The groups of
order 128</a>, J. Algebra 67 (1980), no. 1, 129-142.",
  "Gordon Royle, <a
href=\"http://staffhome.ecm.uwa.edu.au/~00013890/data.html\">Combinatorial
Catalogues</a>. See subpage \"Generators of small groups\" for explicit generators for
most groups of even order < 1000.",
  "D. Rusin, <a href=\"/A000001/a000001.txt\">Asymptotics</a> [Cached copy of lost web
page]",
  "Eric Weisstein's World of Mathematics, <a
href=\"http://mathworld.wolfram.com/FiniteGroup.html\">Finite Group</a>",
  "Wikipedia, <a href=\"http://en.wikipedia.org/wiki/Finite_group\">Finite group</a>",
  "M. Wild, <a href=\"http://www.jstor.org/stable/30037381\">The groups of order sixteen
made easy</a>, Amer. Math. Monthly, 112 (No. 1, 2005), 20-31.",
  "Gang Xiao, <a href=\"http://wims.unice.fr/~wims/wims.cgi?module=tool/algebra/smallgrou
p\">SmallGroup</a>",
  "<a href=\"/index/Gre#groups\">Index entries for sequences related to groups</a>",
  "<a href=\"/index/Cor#core\">Index entries for \"core\" sequences</a>"
 ],
 "formula": [
  "From _Mitch Harris_, Oct 25 2006: (Start)",
  "For p, q, r primes:",
  "a(p) = 1, a(p^2) = 2, a(p^3) = 5, a(p^4) = 14, if p = 2, otherwise 15.",
  "a(p^5) = 61 + 2*p + 2*gcd(p-1,3) + gcd(p-1,4), p >= 5, a(2^5)=51, a(3^5)=67.",
  "a(p^e) ~ p^((2/27)e^3 + O(e^(8/3)))",
  "a(pq) = 1 if gcd(p,q-1) = 1, 2 if gcd(p,q-1) = p. (p < q)",
  "a(pq^2) = one of the following:",
  "* 5, p=2, q odd,",
  "* (p+9)/2, q=1 mod p, p odd,",
  "* 5, p=3, q=2,",
  "* 3, q = -1 mod p, p and q odd.",
  "* 4, p=1 mod q, p > 3, p != 1 mod q^2",
  "* 5, p=1 mod q^2",
  "* 2, q != +/-1 mod p and p != 1 mod q,",
  "a(pqr) (p < q < r) = one of the following:",
  "* q==1 mod p r==1 mod p r==1 mod q a(pqr)",
  "* No...No...No...1",
  "* No...No...Yes...2",
  "* No...Yes...No...2",
  "* No...Yes...Yes...4",
  "* Yes...No...No...2",
  "* Yes...No...Yes...3",
  "* Yes...Yes...No...p+2",
  "* Yes...Yes...Yes...p+4 (table from Derek Holt) (End)",
  "a(n) = A000688(n) + A060689(n). - _R. J. Mathar_, Mar 14 2015"
 ],
 "example": [
  "Groups of orders 1 through 10 (C_n = cyclic, D_n = dihedral of order n, Q_8 =
quaternion, S_n = symmetric):",
  "1: C_1",
```

```
  "2: C_2",
  "3: C_3",
  "4: C_4, C_2 X C_2",
  "5: C_5",
  "6: C_6, S_3=D_6",
  "7: C_7",
  "8: C_8, C_4 X C_2, C_2 X C_2 X C_2, D_8, Q_8",
  "9: C_9, C_3 X C_3",
  "10: C_10, D_10"
 ],
 "maple": [
  "GroupTheory:-NumGroups(n); # with(GroupTheory); loads this command - _N. J. A.
Sloane_, Dec 28 2017"
 ],
 "mathematica": [
  "FiniteGroupCount[Range[100]] (* _Harvey P. Dale_, Jan 29 2013 *)",
  "a[ n_] := If[ n < 1, 0, FiniteGroupCount @ n]; (* _Michael Somos_, May 28 2014 *)"
 ],
 "program": [
  "(MAGMA) D:=SmallGroupDatabase(); [ NumberOfSmallGroups(D, n) : n in [1..1000] ]; //
_John Cannon_, Dec 23 2006",
  "(GAP) A000001 := Concatenation([0], List([1..500], n -> NumberSmallGroups(n))); #
_Muniru A Asiru_, Oct 15 2017"
 ],
 "xref": [
  "The main sequences concerned with group theory are A000001 (this one), A000679,
A001034, A001228, A005180, A000019, A000637, A000638, A002106, A005432, A000688, A060689,
A051532.",
  "Cf. A046058, A023675, A023676. A003277 gives n for which A000001(n) = 1, A063756
(partial sums).",
  "A046057 gives first occurrence of each k.",
  "A027623 gives the number of rings of order n."
 ],
 "keyword": "nonn,core,nice,hard",
 "offset": "0,5",
 "author": "_N. J. A. Sloane_",
 "ext": [
  "More terms from _Michael Somos_",
  "Typo in b-file description fixed by _David Applegate_, Sep 05 2009"
 ],
 "references": 156,
 "revision": 191,
 "time": "2021-03-27T03:48:47-04:00",
 "created": "1991-04-30T03:00:00-04:00"
}
```


Again, there are many different keys, among which we can find the one which is relevant to this project: the `comment` key containing a list of **comments** with their **authors**:

[6]:
```
# Print 'comment' subsection of the sample sequence
print('\n' + 'Printing sample file "comment" subsection...' + '\n')
```

```
comment_list = results[0].get('comment')
if comment_list:
    print(json.dumps(comment_list, indent=True))
else:
    print('No "comments" subsection found.' + '\n')
```

```
Printing sample file "comment" subsection...

[
 "Also, number of nonisomorphic subgroups of order n in symmetric group S_n. - _Lekraj
Beedassy_, Dec 16 2004",
 "Also, number of nonisomorphic primitives of the combinatorial species Lin[n-1]. -
_Nicolae Boicu_, Apr 29 2011",
 "The record values are (A046058): 1, 2, 5, 14, 15, 51, 52, 267, 2328, 56092, 10494213,
49487365422, ..., and they appear at positions (A046059): 1, 4, 8, 16, 24, 32, 48, 64,
128, 256, 512, 1024, ... _Robert G. Wilson v_, Oct 12 2012",
 "In (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008), a(n) is called the \"group
number of n\", denoted by gnu(n), and the first occurrence of k is called the \"minimal
order attaining k\", denoted by moa(k) (see A046057). - _Daniel Forgues_, Feb 15 2017",
 "It is conjectured in (J. H. Conway, Heiko Dietrich and E. A. O'Brien, 2008) that the
sequence n -> a(n) -> a(a(n)) = a^2(n) -> a(a(a(n))) = a^3(n) -> ... -> consists
ultimately of 1s, where a(n), denoted by gnu(n), is called the \"group number of n\". -
_Muniru A Asiru_, Nov 19 2017",
 "MacHale (2020) shows that there are infinitely many values of n for which there are
more groups than rings of that order (cf. A027623). He gives n = 36355 as an example. It
would be nice to have enough values of n to create an OEIS entry for them. - _N. J. A.
Sloane_, Jan 02 2021"
]
```

4 Author parsing

Now that we know where to find the authors' names, we can proceed with building a function to parse all of them from a given file.

4.1 Regular expressions

The most efficient way of doing this is to use a **regular expression** (also known as *regex*), a set of characters specifying a *search pattern*.

We must first identify the ways in which the names have been written; by analyzing some comments, **six main patterns** have been identified, along with the **four regular expressions** needed to match them:

1. *"_Name Surname_"* `(?<=_)[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=_)`
2. *"[Name Surname]"* and *"[Surnamea, Surnameb]"* `(?<=\[)[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=\])`
3. *"- Name, Surname ("* and *"- Name Surname,"* `(?<=- )[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?= \(|, )`
4. *"(Name Surname,"* `(?<=\()[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=,)`

In spite of their apparent complexity, the meaning of these patterns is quite simple and be easily debugged with tools like Regex101. Each of them matches only strings that:

- begin with certain characters `_`, `[`, `-`, `(`,
  - followed by a capital letter `[A-Z]`,
    - not followed by another capital letter `(?!=[A-Z])`,
  - followed by at least any two characters `{2,}?`,
    - at the condition that none of them belong to a list of forbidden symbols `[^0-9+\(\)\[\]\{\}\\\/_:;""]` (where `^` is as a negation operator),
- end with certain characters `_`, `]`, `(` or `,`, `,`.

`(?>=)` and `(?=)` indicate that the matched strings should be preceded or followed (respectively) by the character(s) to the right of the = symbol.

Escaping certain characters distinguishes them from a regex special symbol (e.g. `\(\)` matches the string *()*, while `()` is an empty regex group); whitespaces are simply represented by, well, a whitespace ().

By combining these four expressions with the OR character (`|`) we can create the following regular expression to match all patterns at once in Python:

```
(?<=_)[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=_)|(?<=\[)[A-Z](?!=[A-Z])
[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=\])|(?<=- )[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_
:;""]{2,}?(?= \(|, )|(?<=\()[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?(?=,)
```

Completeness

It should be noted that these expressions **do not find all the authors** present in the comments because they are not written consistently across all sequences. One might argue that it would be sufficient finding all patterns used in order to get all the names; while this would be a good, if not really feasible solution (we do not know how many they are), the problem remains because certain patterns also match formulas and other unrelated data, making them unusable for retrieving only names.

The definitive solution would be to either manually get the names, or to allow the matching of extraneous data in order to remove it later from the list of names; this would take too long, though, and goes beyond the purpose of this project.

4.2 Parsing function

The parsing function gets the `dict` **raw data** read by the JSON library in input and returns a **set of all author names** present in the comments of the loaded file (or `None` if there are none).

Basically, after preparing the regex pattern (`re.compile()`), for each `comment` in a non-empty `comment_list` the function gets a list of the authors' names using Python's `re` package for regular expressions, and uses it to update the set of unique authors called `authors` (which contains all names found in the file). The list comprehension in the `update` method is needed to flatten the many lists of lists returned by `re.findall()`.

[7]:
```
def parse_authors_from_comments(raw_data):
    # Regex pattern
    common_pattern = r'[A-Z](?!=[A-Z])[^0-9+\(\)\[\]\{\}\\\/_:;""]{2,}?'
    pattern_list = [('(?<=_)', '(?=_)'), ('(?<=\[)', '(?=\])'), ('(?<=- )', '(?= \(|,
  )'), ('(?<=\()', '(?=,)')]
    pattern = re.compile('|'.join([start + common_pattern + end for start, end in
  pattern_list]))
```

```
    # Comment parsing
    comment_list = raw_data.get('results')[0].get('comment')
    if comment_list:
        authors = set()
        for comment in comment_list:
            authors.update([n for names in re.findall(pattern, comment) for n in names.
split(', ')])
        return authors
    return
```

Some observations:

- the regex pattern is initially split into its **subpatterns** for better readability and to avoid repetitions;
- this pattern has been accurately written so as to **not return empty matches**, normally generated by *capturing groups* (groups of characters between round parentheses) and for which additional `if`s would have been needed, resulting in a more complicated list comprehension;
- **some sequences do not contain comments**, hence the check on `comment_list`;
- a **set** has been chosen for the `authors_set` variable in order to **exclude duplicate names**, since the data needed for the project only concerns the presence or absence of a given author in the comments of a sequence, not all his/her instances. Python's `set` data structure allows to store items in a hash table, without duplicating them.

5 Graph building

We can now proceed by parsing the authors from all OEIS sequences in the `data/sequences` directory and build their graph using the NetworkX library, eventually saving it to disk to avoid loading every time all the JSON files.

Considering that each **node** of the graph should contain the **name of a single author** (without duplicates), we only need to:

1. add each author of each sequence as a node;
2. add edges between all pairs of authors which have commented the same sequence.

By repeating this procedure for every file in the `data/sequences` directory we get a graph of all authors, where people who have commented the same sequence are connected by an edge.

The creation of such a graph is quite simple with the NetworkX library, since we only need to:

- parse each sequence file;
- extract its authors;
- add them as nodes;
- create a list of all possible pairs of authors in each sequence;
- add an edge for each pair.

Since the first two operations have been already implemented in the previous steps (see the `parse_authors_from_comments()` function), the other two are as simple as two lines of code, knowing that **NetworkX does not complain when adding existing nodes or edges**: we do not need to check every time if a given author has already been inserted or if a certain edge already exists, because the library will *not* duplicate them[7]. In fact, we could even skip the `add_nodes_from()` function, since NetworkX

automatically inserts non-existing nodes when adding edges connecting them (which is why it has been commented in the code below).

The best way to compute all author pairs for each sequence is given by the itertools library, which implements efficient looping.

Some notes about the `build_graph_from_directory()` function:

- all it needs as input arguments is the **path** of the directory containing the JSON files and a **boolean flag** to specify if the resulting graph should also be saved to disk (instead of simply returned) - along with a name for the newly created JSON graph file, eventually (otherwise `comments_authors_graph.json` is applied by default);
- it begins with checking the correctness of the JSON files path and creating the necessary variables, among which:
  - a list of all files in the given directory (using `os.listdir()`);
  - an empty NetworkX graph g;
  - a tqdm progress bar, only needed to visualize the overall progress of the parsing process.

[8]:
```
def build_graph_from_directory(dir_path, save=False, filename='comments_authors_graph'):
    # Get file list
    if dir_path[-1] != '/':
        dir_path += '/'
    file_list = [json_file for json_file in os.listdir(dir_path) if json_file.endswith('.
  json')]

    # Prepare variables
    g = nx.Graph()
    progress_bar = tqdm.tqdm(total=len(file_list))

    # Parse all JSON files
    for f in file_list:
        progress_bar.set_description('Parsing file {}'.format(f))
        file_path = dir_path + f
        raw_data = load_json(file_path)

        authors = parse_authors_from_comments(raw_data)
        if authors:
            # g.add_nodes_from(authors)
            g.add_edges_from(list(its.combinations(authors, 2)))
        progress_bar.update(1)

    # Save graph
    if save:
        try:
            with open(dir_path.split('/')[0] + '/' + filename + '.json', 'w') as
  out_file:
                json.dump(nx.readwrite.json_graph.node_link_data(g), out_file)
        except OSError:
            print('Could not save file: {}, exiting program.'.format(filename + '.json'))

    return g
```

Alternatively, assuming that the graph has already been built and saved to disk, it can be loaded from an existing JSON file with the `load_json_graph()` function, which simply takes the JSON graph's path as input:

[9]:
```
def load_json_graph(file_path):
    try:
        with open(file_path, 'r') as file:
            raw_data = json.load(file)
            return nx.readwrite.json_graph.node_link_graph(raw_data)
    except OSError:
        print('Could not open file: {}, exiting program.'.format(file_path.split('/
  ')[-1]))
```

The graph can thus be created by running:

[10]:
```
# Graph creation (either from raw data or existing JSON graph)
build_graph = False  # Set to 'True' in order to build graph from raw data

if build_graph:  # Build graph and save to file
    print('\n' + 'Building graph g, where:')
    print('- nodes represent all unique authors that can be found in each comment of
  every sequence;')
    print('- edges link two authors who have commented the same sequence...')

    g = build_graph_from_directory('data/sequences', save=True)
else:  # Load graph from disk
    print('\n' + 'Loading graph g from "data/comments_authors_graph.json", where:')
    print('- nodes represent all unique authors that can be found in each comment of
  every sequence;')
    print('- edges link two authors who have commented the same sequence.')

    g = load_json_graph('data/comments_authors_graph.json')

print('\n' + 'Graph g has {} nodes and {} edges.'.format(len(g.nodes), len(g.edges)))
```

```
Loading graph g from "data/comments_authors_graph.json", where:
- nodes represent all unique authors that can be found in each comment of every sequence;
- edges link two authors who have commented the same sequence.

Graph g has 2831 nodes and 49096 edges.
```

All variable names are lowercase with words separated by undescores in order to be compliant with the Python Enhancement Proposals 8 (PEP 8) style guide[4].

6 Maximal cliques

As stated in the introduction, our goal for this project is to explore the problem of **finding maximal cliques** in a graph by building three algorithms to find:

1. a maximal clique;
2. a list of all maximal cliques;

3. the maximum clique.

Before proceeding with the actual Python code, we shall provide first some useful definitions and theoretical notions .

6.1 Definitions

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph where $\mathcal{V}$ is the set of all nodes and $\mathcal{E}$ the set of all edges.

Def. *A **clique** of $\mathcal{G}$ is a **complete subgraph**, or a simple undirected graph in which each pair of vertices is connected by an edge[8][9].*

Def. *A **maximal clique** is a clique that cannot be extended by including one more adjacent vertex, meaning it is not a subset of a larger clique.*

Fact. *The **maximum clique** in a graph (i.e. the clique of largest size) is always maximal, while the converse does not hold[10].*

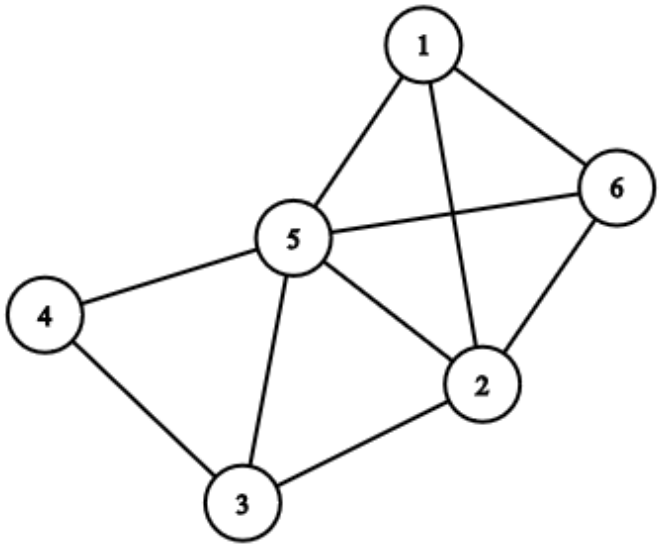


Figure 1: The *maximal cliques* of this graph are $\{1,2,5,6\}$, $\{2,3,5\}$ and $\{3,4,5\}$. Among these, the *maximum clique* is $\{1,2,5,6\}$.

6.2 Greedy algorithm

The easiest way to find **one arbitrary maximal clique** is to run on our graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a **greedy algorithm**, which makes the local optimal choice at each step:

Algorithm 1: GREEDY-MAXIMAL-CLIQUE

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ simple undirected graph
Output: $\mathcal{C} \subset \mathcal{V}$ maximal clique of $\mathcal{G}$

$s \in \mathcal{V}$ starting node
$\mathcal{C} \subset \mathcal{E}, \mathcal{C} = \{s\}$ the set of nodes in the maximal clique

for $v \in \mathcal{V} \setminus \{s\}$ **do**
  if $\exists (v,u) \in \mathcal{E} \quad \forall u \in \mathcal{C}$ **then**
    $\mathcal{C} = \mathcal{C} \cup \{v\}$
  end
end

The algorithm simply keeps adding nodes connected to all of the nodes already present in the current clique $\mathcal{C}$ until no other node is found. $\mathcal{C}$ can be initialised to contain any node s in the graph.

We can devise a **more refined version** which does not test *all* nodes in the graph, but only the neighbors of the nodes in the current clique, thus excluding those vertices which will surely never pass the test (e.g. nodes in different connected components), for better efficiency:

Algorithm 2: GREEDY-MAXIMAL-CLIQUE-NEIGHBORS

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ simple undirected graph
Output: $\mathcal{C} \subset \mathcal{V}$ maximal clique of $\mathcal{G}$

$s \in \mathcal{V}$ starting node
$\mathcal{C} \subset \mathcal{E}, \mathcal{C} = \{s\}$ the set of nodes in the maximal clique

while $\mathcal{N}(\mathcal{C}) \neq \emptyset$ **do**
  $v \in \mathcal{N}(\mathcal{C})$
  $\mathcal{N}(\mathcal{C}) = \mathcal{N}(\mathcal{C}) \setminus \{v\}$
  if $\exists (v, u) \in \mathcal{E} \quad \forall u \in \mathcal{C}$ **then**
    $\mathcal{C} = \mathcal{C} \cup \{v\}$
    $\mathcal{N}(\mathcal{C}) = \mathcal{N}(\mathcal{C}) \cup \mathcal{N}(v)$
  end
end

The Python implementation is almost identical to this pseudocode, with the exception of a `valid` **flag** kept in order to quit the loop as soon as an edge (v, u) does not exist, **to reduce the number of iterations**, and the fact that **only non-trivial maximal cliques are returned** (i.e. only those with more than 2 nodes are considered).

The resulting function `find_one_maximal_clique_greedy()` takes in input a NetworkX graph `g` and, optionally, two boolean flags for:

- choosing the greedy algorithm variant (`naive` or `neighbors`, of which the latter is default), and
- printing the clique found (`print_result`, `False` by default).

The choice to use nested functions has been made to unify the two variants in a single algorithm for finding a maximal clique, since they are simple enough and do not really need to be differentiated (the naive version is more a curiosity than an every-day solution).

This function also checks whether the provided graph is a NetworkX undirected graph, and if it is empty or not (and raises an exception, accordingly).

[11]:
```python
def find_one_maximal_clique_greedy(g, variant='neighbors', print_result=False):
    # Check that g is a NetworkX graph
    if not isinstance(g, nx.classes.graph.Graph):
        raise nx.NetworkXError('The provided graph is not a valid NetworkX undirected
  graph.')

    if g.nodes:
        # Naive variant (all nodes)
        if variant == 'naive':
            # Initialization
            vertices = list(g.nodes)
            s = random.choice(vertices)
            vertices.remove(s)
```

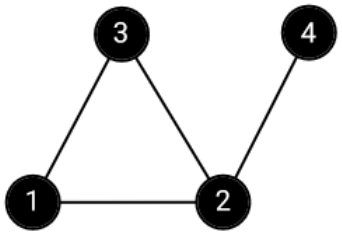
```
            clique = {s}

            # Greedy algorithm
            for v in vertices:
                valid = True
                for u in clique:
                    if not g.has_edge(v, u):
                        valid = False
                        break
                if valid:
                    clique.add(v)
        # Neighbors variant
        else:
            # Wrong argument warning
            if variant != 'neighbors':
                warnings.warn('Invalid algorithm variant (\'{}\'). Using greedy algorithm
restricted to neighbors as default.'.format(variant))

            # Initialization
            s = random.choice(list(g.nodes))
            neighbors = list(g.neighbors(s))
            clique = {s}

            # Greedy algorithm
            while neighbors:
                v = neighbors.pop()
                valid = True

                for u in clique:
                    if not g.has_edge(v, u):
                        valid = False
                        break
                if valid:
                    clique.add(v)
                    neighbors.extend(list(g.neighbors(v)))

        # Result & printing
        if len(clique) > 2:
            if print_result:
                print(clique)
            return clique
        else:
            return
    else:
        raise nx.NetworkXPointlessConcept('The provided graph is empty.')
```

Greedy algorithm results

[12]:
```
# Find and print one maximal clique
find_one_maximal_clique_greedy(g, print_result=True)
```

```
{'Gregory Pat Scandalis', 'Paolo P. Lava', 'M. F. Hasler', 'Franklin T. Adams-Watters',
   'Omar E. Pol', 'Wolfdieter Lang', 'Reinhard Zumkeller', 'Fredrik Johansson'}
```

[12]:
```
{'Franklin T. Adams-Watters',
 'Fredrik Johansson',
 'Gregory Pat Scandalis',
 'M. F. Hasler',
 'Omar E. Pol',
 'Paolo P. Lava',
 'Reinhard Zumkeller',
 'Wolfdieter Lang'}
```



s = 3
v = {1, 2, 4}
C = {3}

begin for loop
loop 1/3: ***v = 1***
does edge (1, 3) exist? -> yes
add 1 to C: C = {1, 3}

loop 2/3: ***v = 2***
does edge (2, 1) exist? -> yes
does edge (2, 3) exist? -> yes
add 2 to C: C = {1, 2, 3}

loop 3/3: ***v = 4***
does edge (4, 1) exist? -> no
does edge (4, 2) exist? -> yes
does edge (4, 3) exist? -> no
end for loop

C = {1, 2, 3} is a maximal clique

Figure 2: Application of the *naive* greedy algorithm to a graph with 4 nodes. The *neighbors* variant would initialise v (the vertices/neighbors list) to $\{1, 2\}$ and skip the last loop.

6.3 Bron-Kerbosch algorithm

In order to find all maximal cliques in our graph we can proceed by implementing the **Bron-Kerbosch algorithm**[1], designed by its Dutch namesakes in 1973 and still widely used nowadays, either in its classic form or in one of its more efficient variants.

6.3.1 Bron-Kerbosch classic

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, three sets of nodes, R, P and X, play an important role in the algorithm:

- R is the set to be **extended or shrunk** by a new node. Nodes that are eligible to extend it, i.e. that are connected to all the other nodes in R, are collected recursively in the remaining two sets;
- P is the set of **candidates**, i.e. of all nodes that will in due time serve as an extension to the present configuration of R;
- X is the set of all nodes that have at an earlier stage already served as an extension of the present configuration of R and are now explicitly **excluded**.

The core of the algorithm consists of a **recursive function** applied to the three sets, which generates all extensions of the given configuration of R that can be made with the nodes in P and that do not contain any of the nodes in X (all extensions of R containing any node in X have already been generated):

Algorithm 3: BRON-KERBOSCH

Input: R, P, $X \subset \mathcal{V}$
Output: all maximal cliques of $\mathcal{G}$

if $P = \emptyset$ *and* $X = \emptyset$ **then**
  R is a maximal clique
end
for $v \in P$ **do**
  BRON-KERBOSCH($R \cup \{v\}$, $P \cap \mathcal{N}(v)$, $X \cap \mathcal{N}(v)$)
  $P = P \setminus \{v\}$
  $X = X \cup \{v\}$
end

The extra labor involved in maintaining the set X is motivated by the fact that a necessary condition for having created a clique is that P be empty, otherwise R could still be extended. This condition, however, is not sufficient, because if now X is non-empty, we know from the definition of X that the present configuration of R has already been contained in another configuration and is therefore not maximal, so we can state that R is a clique only as soon as *both* X *and* P are empty.

If at some stage X contains a node connected to all nodes in P, we can predict that further extensions (further selection of candidates) will never lead to the removal of that particular node from subsequent configurations of X and, therefore, not to a clique.

In order to find all maximal cliques of a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we only need to set:

- $R = \emptyset$;
- $P = \mathcal{V}$;
- $X = \emptyset$.

The Python implementation of this algorithm translates the pseudocode quite literally:

[13]:
```
# Classic Bron-Kerbosch algorithm
def bron_kerbosch(r, p, x):
    if not p and not x:
```

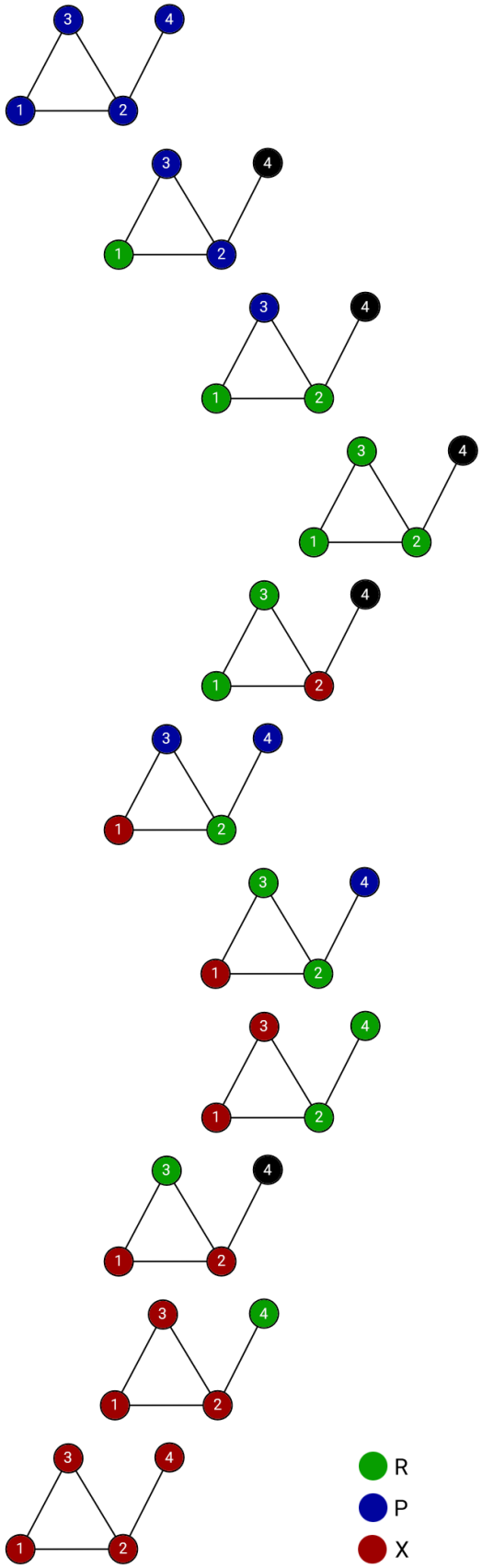


Figure 3: Application of the Bron-Kerbosch algorithm to a graph with 4 nodes.

```
        if len(r) > 2:
            yield r
    for v in {*p}:
        yield from bron_kerbosch(r | {v}, p & {*g.neighbors(v)}, x & {*g.neighbors(v)})
        p = p - {v}
        x.add(v)
```

6.3.2 Bron-Kerbosch with Tomita pivoting

The original Bron-Kerbosch algorithm might require large amounts of memory, as it does not avoid backtracking from useless cases where $P = \emptyset$ and $X = \emptyset$. These unfruitful occurences can be decreased by choosing a **pivot vertex** $u \in P \cup X$ in such a way that maximal cliques must contain either u or a vertex in $P \setminus \mathcal{N}(u)$, or else the clique could be extended by u. In other words, only nodes in $P \setminus \mathcal{N}(u)$ will be candidates in each recursive call to the algorithm. A simple, effective way to choose the pivot is called the **Tomita pivoting**[3]:

Def. *The pivot $u \in P \cup X$ is the node that maximises $|P \cap \mathcal{N}(u)|$, i.e. the node having the most neighbors in P.*

Algorithm 4: BRON-KERBOSCH-TOMITA-PIVOTING

Input: R, P, $X \subset \mathcal{V}$
Output: all maximal cliques of $\mathcal{G}$

if $P = \emptyset$ *and* $X = \emptyset$ **then**
  R is a maximal clique
end
choose pivot $u \in P \cup X$ that maximises $|P \cap \mathcal{N}(u)|$
for $v \in P$ **do**
  BRON-KERBOSCH-TOMITA-PIVOTING($R \cup \{v\}$, $P \cap \mathcal{N}(v)$, $X \cap \mathcal{N}(v)$)
  $P = P \setminus \{v\}$
  $X = X \cup \{v\}$
end

Again, the Python version does not differ much from the pseudocode, except for a `try...except` clause needed to handle empty sets when choosing the pivot:

[14]:
```
# Bron-Kerbosch algorithm with Tomita pivoting
def bron_kerbosch_tomita_pivot(r, p, x):
    if not p and not x:
        if len(r) > 2:
            yield r
    try:
        u = max({(v, len({n for n in g.neighbors(v) if n in p})) for v in p | x},
  key=lambda v: v[1])[0]
        for v in p - {*g.neighbors(u)}:
            yield from bron_kerbosch_tomita_pivot(r | {v}, p & {*g.neighbors(v)}, x &
  {*g.neighbors(v)})
            p = p - {v}
            x.add(v)
    except ValueError:
        pass
```

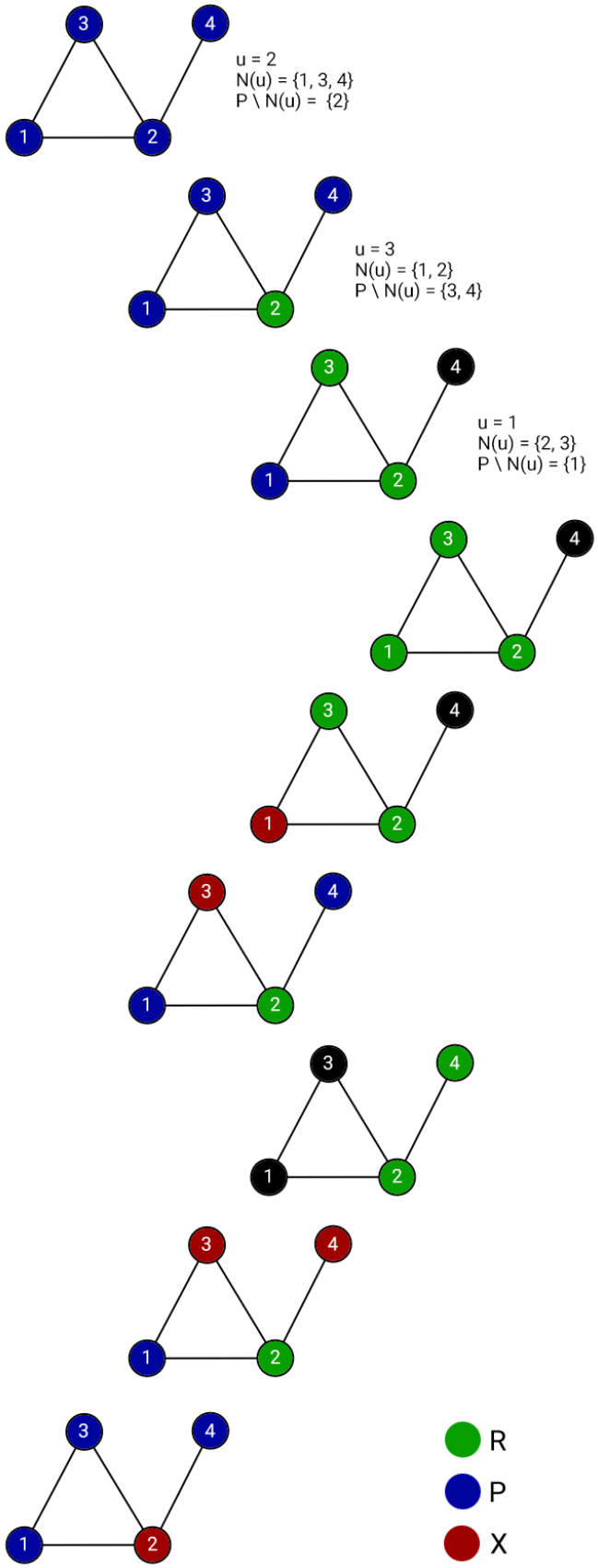


Figure 4: Application of the pivot Bron-Kerbosch algorithm to a graph with 4 nodes.

6.3.3 Bron-Kerbosch with degeneracy ordering

Apart from the pivoting strategy, the **order** in which the vertices of $\mathcal{G}$ are processed by the Bron–Kerbosch algorithm is also very important. Before continuing, let us see the notion of **degeneracy**, which will help to illustrate the next approach.

Def. *The **degeneracy** of an n-vertex graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the smallest number d such that every subgraph of $\mathcal{G}$ contains a vertex of degree at most d.*

Def. *A graph with degeneracy d also has a **degeneracy ordering**, i.e. an ordering of the vertices such that each vertex has d or fewer neighbors that come later in the ordering.*

Degeneracy, along with a degeneracy ordering, can be computed by a simple **greedy strategy** of repeatedly removing a vertex with smallest degree (and its incident edges) from the graph until it is empty:

Algorithm 5: DEGENERACY-ORDERING

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ simple undirected graph
Output: d degeneracy ordering of $\mathcal{G}$

D array s.t. $D[i]$ stores the list of vertices $v \in \mathcal{V}$ of degree i
d array containing the degeneracy ordering

while $D \neq \emptyset$ **do**
  scan D until the first non-empty list $D[i]$ is found
  move a vertex u from $D[i]$ to d
  for $v \in \mathcal{N}(u)$ **do**
    move v from $D[j]$ to $D[j-1]$, where j is the degree of v
  end
  remove u from the graph $\mathcal{G}$
end

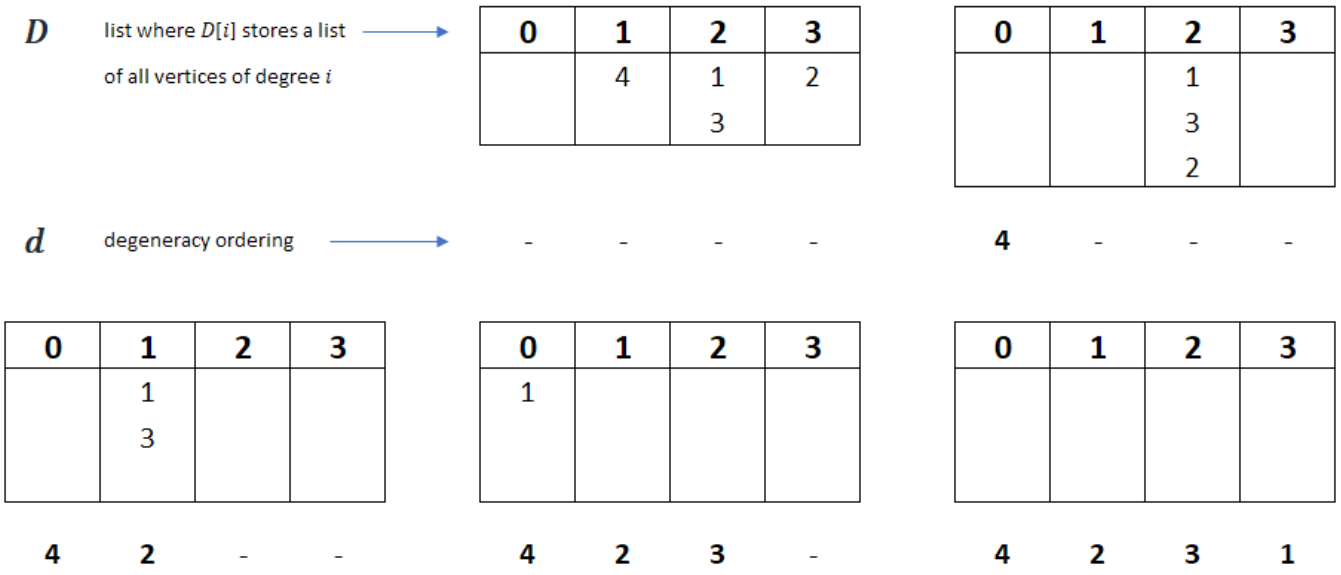


Figure 5: Degeneracy ordering algorithm applied to the graph shown in figures 3 and 4.

Using these facts, Eppstein et al.[2] showed in 2010 that there exists a nearly-optimal algorithm for **enumerating all maximal cliques parametrized by degeneracy**, and that in order to achieve this result a modification of the classic Bron–Kerbosch algorithm was sufficient.

They performed the outer level of recursion of the Bron–Kerbosch algorithm without pivoting, using a

degeneracy ordering to order the sequence of recursive calls, and then switched at inner levels of recursion to the pivoting rule of Tomita et al.[3]:

Algorithm 6: BRON-KERBOSCH-DEGENERACY

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ simple undirected graph
Output: all maximal cliques of $\mathcal{G}$

$R = \emptyset$
$P = \mathcal{V}$
$X = \emptyset$
$d =$ DEGENERACY-ORDERING($\mathcal{G}$)

for $v \in d$ **do**
    BRON-KERBOSCH-TOMITA-PIVOTING($R \cup \{v\}$, $P \cap \mathcal{N}(v)$, $X \cap \mathcal{N}(v)$)
    $P = P \setminus \{v\}$
    $X = X \cup \{v\}$
end

Thanks to this ordering, the sets P passed to each of the recursive calls will have at most d elements in them, minimizing the recursive calls within each of the outer calls, while the set X will consist of all earlier neighbors of v (could be larger than d).

These two algorithms can be implemented in Python as follows:

[15]:
```python
def get_degeneracy_ordering(graph):
    # Check that g is a NetworkX graph
    if not isinstance(graph, nx.classes.graph.Graph):
        raise nx.NetworkXError('The provided graph is not a valid NetworkX undirected
  graph.')

    if graph.nodes:
        g = graph.copy()

        # Create and populate lists of lists
        max_degree = max([d for n, d in g.degree()])
        d = [[] for deg in range(max_degree + 1)]
        for node in g.degree():
            d[node[1]].append(node[0])

        # Degeneracy ordering
        degeneracy_ordering = []
        while d:
            # Get current node u
            u = next(i for i in d if i).pop()
            degeneracy_ordering.append(u)

            # Move neighbors of current node
            for v in {*g.neighbors(u)}:
                v_deg = g.degree(v)
                d[v_deg].remove(v)
                d[v_deg-1].append(v)

            # Remove current node from graph
```

```
            g.remove_node(u)

            # Remove last list of d if empty (ensure termination of while loop)
            if not d[len(d)-1]:
                d.pop()

        return degeneracy_ordering
    else:
        raise nx.NetworkXPointlessConcept('The provided graph is empty.')
```

`get_degeneracy_ordering()` begins by checking whether the provided graph is a NetworkX undirected graph, and if it is empty or not, and raises an exception, accordingly. It then continues by making a copy of the input graph, which is necessary because the progressive removal of nodes would otherwise leave the original graph empty after applying this function to it.

The function does not differ much from its pseudocode, apart from the use of a `while` loop which is terminated by progressively removing any empty list at the tail of `d` (since neighbors of the current node will always be moved to a list of lower degree, and never higher). Given the need for dynamic arrays, the simple Python list has been used instead of more efficient data structures like in the clique methods.

[16]:
```
# Bron-Kerbosch algorithm with Tomita pivoting & degeneracy ordering
def bron_kerbosch_degeneracy(r, p, x):
    for v in get_degeneracy_ordering(g):
        yield from bron_kerbosch_tomita_pivot(r | {v}, p & {*g.neighbors(v)}, x & {*g.
  neighbors(v)})
        p = p - {v}
        x.add(v)
```

6.3.4 Complexity

The **worst-case analysis** for the Bron-Kerbosch algorithm is $O(3^{\frac{n}{3}})$ running time. It is optimal as a function of n, since there are at most $3^{\frac{n}{3}}$ maximal cliques in an n-vertex graph[5]. It also has the nice property that it generates **all and only maximal cliques without duplication**.

Eppstein et al.'s variant instead runs in time $O(dn3^{\frac{d}{3}})$, and the degeneracy d is expected to be low in many real-world applications. The time needed to obtain the degeneracy ordering is irrelevant, as it runs linear to the number of vertices n and edges m of the graph, i.e. $O(n+m)$.

Eppstein et al.'s results originate from the observation that given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with degeneracy d:

- $\mathcal{G}$ has at most $d(n - \frac{d+1}{2})$ edges;
- the maximum clique size can be at most $d+1$, for any larger clique would form a subgraph in which all vertices have degree higher than d;
- if d is a multiple of 3 and $n \geq d+3$, then the largest possible number of maximal cliques is $(n-d)3^{\frac{d}{3}}$.

Real-world graphs are actually quite **sparse**, making the **degeneracy** version of the Bron-Kerbosch algorithm an **excellent choice**.

Wrapper function for the Bron-Kerbosch algorithm

The actual Python implementation of the presented algorithms consist of a single method `find_all_maximal_cliques_bk()` which defines three nested functions, one for each Bron-Kerbosch variant, and which takes in input only a NetworkX graph g and, optionally, two boolean flags for:

- choosing the Bron-Kerbosch variant (`classic`, `tomita` and `degeneracy`, of which the latter is default), and
- printing the cliques found (`print_result`, `False` by default).

The choice to use nested functions has been made to avoid repeating, for every variant, the definition of the sets R, P and X used in all three variants of the algorithm, as well as the checks on the input graph. This way the algorithms work correctly without compromising their legibility with language-specific code, resulting in an almost literal implementation of the pseudocode previously provided.

The complete code for the function to find all maximal cliques is then the following:

[17]:
```
def find_all_maximal_cliques_bk(g, variant='degeneracy', print_result=False):
    # Check that g is a NetworkX graph
    if not isinstance(g, nx.classes.graph.Graph):
        raise nx.NetworkXError('The provided graph is not a valid NetworkX undirected
  graph.')

    # Classic Bron-Kerbosch algorithm
    def bron_kerbosch(r, p, x):
        if not p and not x:
            if len(r) > 2:
                yield r
        for v in {*p}:
            yield from bron_kerbosch(r | {v}, p & {*g.neighbors(v)}, x & {*g.
  neighbors(v)})
            p = p - {v}
            x.add(v)

    # Bron-Kerbosch algorithm with Tomita pivoting
    def bron_kerbosch_tomita_pivot(r, p, x):
        if not p and not x:
            if len(r) > 2:
                yield r
        try:
            u = max({(v, len({n for n in g.neighbors(v) if n in p})) for v in p | x},
  key=lambda v: v[1])[0]
            for v in p - {*g.neighbors(u)}:
                yield from bron_kerbosch_tomita_pivot(r | {v}, p & {*g.neighbors(v)}, x &
  {*g.neighbors(v)})
                p = p - {v}
                x.add(v)
        except ValueError:
            pass

    # Bron-Kerbosch algorithm with Tomita pivoting & degeneracy ordering
    def bron_kerbosch_degeneracy(r, p, x):
        for v in get_degeneracy_ordering(g):
            yield from bron_kerbosch_tomita_pivot(r | {v}, p & {*g.neighbors(v)}, x &
  {*g.neighbors(v)})
            p = p - {v}
            x.add(v)
```

```
    # Main clique function
    if g.nodes:
        # Set initialization
        r = {*()}
        p = {*g.nodes}
        x = {*()}

        # Bron-Kerbosch algorithm
        if variant == 'classic':
            cliques = bron_kerbosch(r, p, x)
        elif variant == 'tomita':
            cliques = bron_kerbosch_tomita_pivot(r, p, x)
        elif variant == 'degeneracy':
            cliques = bron_kerbosch_degeneracy(r, p, x)
        else:
            warnings.warn('Invalid algorithm variant (\'{}\'). Using Bron-Kerbosch with
  degeneracy ordering as default.'.format(variant))
            cliques = bron_kerbosch_degeneracy(r, p, x)

        # Printing
        if print_result:
            print(*cliques, sep='\n')

        return cliques
    else:
        raise nx.NetworkXPointlessConcept('The provided graph is empty.')
```

Note: the **degeneracy ordering function** has been left outside the `find_all_maximal_cliques()` method since it does not require particular checks on the graph (except those for its validity), nor the definition of subsets of nodes, meaning that unlike the Bron-Kerbosch algorithms, it can be used outside this particular application (for different purposes) without stringent requisites.

It should also be noted that sets R, P and X are implemented using Python's efficient `set`, and in particular empty sets are created using set literals `{*()}`[6], which are slightly faster (and more elegant) than the equivalent `set()` constructor, as demonstrated by this code snippet:

[18]:
```
# Efficiency of set() vs. {*()}
print('\n' + 'Efficiency of {*()} vs. set():')

number_set = 100000000
empty_literal_time = (timeit.timeit('{*()}', number=number_set)) / number_set
set_time = (timeit.timeit('set()', number=number_set)) / number_set

print('- empty literal execution time: {} s.'.format(empty_literal_time))
print('- set constructor execution time: {} s.'.format(set_time))
if empty_literal_time < set_time:
    print('Empty literal is faster than set constructor.')
else:
    print('Set constructor is faster than empty literal.')
```

```
Efficiency of {*()} vs. set():
```

```
- empty literal execution time: 6.956522499999948e-08 s.
- set constructor execution time: 7.692990500000008e-08 s.
Empty literal is faster than set constructor.
```

Bron-Kerbosch algorithm results

Let us now find all maximum cliques, and print them exactly once. Since our graph has more than 2000 nodes, this operation could take too long, so we will restrict this search to a **random subgraph of 100 vertices** in order to effectively test our algorithm while still saving time. NetworkX' library functions make the extraction of the subgraph immediate:

[19]:
```python
def sample_random_subgraph(g, n):
    # Check that g is a NetworkX graph
    if not isinstance(g, nx.classes.graph.Graph):
        raise nx.NetworkXError('The provided graph is not a valid NetworkX undirected
  graph.')

    # Check that g is not empty
    if g.nodes:
        return g.subgraph(random.sample(g.nodes, n))
    else:
        raise nx.NetworkXPointlessConcept('The provided graph is empty.')
```

The resulting cliques, calculated using the efficient degeneracy ordering variant of the Bron-Kerbosch algorithm, are:

[20]:
```python
# Find and print all maximal cliques in a random subgraph of 100 nodes
subgraph = sample_random_subgraph(g, 100)
find_all_maximal_cliques_bk(subgraph, variant='degeneracy', print_result=True)
```

```
{'Roman Witula', 'Gary W. Adamson', 'Steve Butler'}
{'Gary W. Adamson', 'Manda Riehl', 'Jean-Luc Baril'}
{'Gary W. Adamson', 'Peter M. Chema', 'Colin Hall'}
{'R. J. Mathar', 'Dmitry Zaitsev', 'Manda Riehl'}
{'Gary W. Adamson', 'John Keith', 'Alexandre Wajnberg'}
{'Gary W. Adamson', 'Ahmed Fares', 'Kem Phillips'}
{'David Christopher', 'Gary W. Adamson', 'David W. Wilson'}
{'Gary W. Adamson', 'Graham H. Hawkes', 'Ph. Leroux'}
{'R. J. Mathar', 'Gary W. Adamson', 'Karl V. Keller'}
{'R. J. Mathar', 'Jonathan Sondow', 'Gary W. Adamson', 'Peter M. Chema'}
{'Berlin', 'Ahmed Fares', 'Gary W. Adamson', 'Manda Riehl'}
{'David W. Wilson', 'R. J. Mathar', 'Frank Ruskey', 'Gary W. Adamson'}
{'R. J. Mathar', 'Gary W. Adamson', 'Manda Riehl', 'Bryan T. Ek'}
{'R. J. Mathar', 'Roman Witula', 'Gary W. Adamson', 'Rogério Serôdio'}
{'R. J. Mathar', 'Alzhekeyev Ascar M', 'Gary W. Adamson', 'R. K. Guy'}
{'Gary W. Adamson', 'Isaac Saffold', 'R. J. Mathar', 'William Entriken'}
{'Richard Stanley', 'Gary W. Adamson', 'Kassie Archer', 'Dimitris Valianatos'}
{'David W. Wilson', 'R. J. Mathar', 'Gary W. Adamson', 'Ahmed Fares', 'Isaac Saffold'}
{'David W. Wilson', 'Richard Stanley', 'Douglas Latimer', 'Gary W. Adamson', 'AA >= n'}
{'David W. Wilson', 'Richard Stanley', 'R. J. Mathar', 'Douglas Latimer', 'Gary W.
Adamson'}
{'Richard Stanley', 'R. J. Mathar', 'Gary W. Adamson', 'Dimitris Valianatos'}
```

```
{'Dimitris Valianatos', 'R. J. Mathar', 'Jonathan Sondow', 'Gary W. Adamson', 'Alexandre
Wajnberg', 'Michael B. Porter'}
{'Douglas Latimer', 'David W. Wilson', 'R. J. Mathar', 'Gary W. Adamson', 'Ahmed Fares',
'Vladimir Letsko'}
{'David W. Wilson', 'R. J. Mathar', 'Gary W. Adamson', 'Ahmed Fares', 'Manda Riehl'}
{'Jonathan Sondow', 'Gary W. Adamson', 'Alexandre Wajnberg', 'For example', 'R. K. Guy'}
{'R. J. Mathar', 'Jonathan Sondow', 'Gary W. Adamson', 'Alexandre Wajnberg', 'R. K. Guy'}
{'David W. Wilson', 'Jonathan Sondow', 'Gary W. Adamson', 'Alexandre Wajnberg', 'For
example'}
{'David W. Wilson', 'Chai Wah Wu', 'R. J. Mathar', 'Jonathan Sondow', 'Gary W. Adamson',
'Alexandre Wajnberg'}
{'David W. Wilson', 'R. J. Mathar', 'Graham H. Hawkes', 'Gary W. Adamson', 'Manda Riehl',
'Michael B. Porter', 'Alexandre Wajnberg'}
{'David W. Wilson', 'R. J. Mathar', 'Jonathan Sondow', 'Gary W. Adamson', 'Alexandre
Wajnberg', 'Michael B. Porter'}
{'R. J. Mathar', 'Roman Witula', 'Gary W. Adamson', 'Manda Riehl'}
{'R. J. Mathar', 'Roman Witula', 'Gary W. Adamson', 'Jonathan Sondow'}
{'Neven Juric', 'Roman Witula', 'Gary W. Adamson', 'Jonathan Sondow', 'Ravi Kumar
Davala', 'Liam Solus', 'Kostas Manes'}
```

[20]:
```
<generator object find_all_maximal_cliques_bk.<locals>.bron_kerbosch_degeneracy at
0x000002376F418E40>
```

At this point we can **verify** the **efficiency** and **correctness** of the algorithm by running the other two Bron-Kerbosch variants and comparing their results:

[21]:
```
# Efficiency of different Bron-Kerbosch variants
print('\n' + 'Efficiency of different Bron-Kerbosch variants:')

bk_classic_start = timeit.default_timer()
bk_classic_cliques = find_all_maximal_cliques_bk(subgraph, variant='classic')
bk_classic_end = timeit.default_timer()

bk_tomita_start = timeit.default_timer()
bk_tomita_cliques = find_all_maximal_cliques_bk(subgraph, variant='tomita')
bk_tomita_end = timeit.default_timer()

bk_degeneracy_start = timeit.default_timer()
bk_degeneracy_cliques = find_all_maximal_cliques_bk(subgraph, variant='degeneracy')
bk_degeneracy_end = timeit.default_timer()

bk_classic_time = bk_classic_end - bk_classic_start
bk_tomita_time = bk_tomita_end - bk_tomita_start
bk_degeneracy_time = bk_degeneracy_end - bk_degeneracy_start

print('- Bron-Kerbosch classic execution time: {} s.'.format(bk_classic_time))
print('- Bron-Kerbosch with Tomita pivoting execution time: {} s.'.
  format(bk_tomita_time))
print('- Bron-Kerbosch with degeneracy ordering execution time: {} s.'.
  format(bk_degeneracy_time))
if bk_classic_time < bk_tomita_time and bk_classic_time < bk_degeneracy_time:
    print('Bron-Kerbosch classic is faster than the other two variants.')
```

```
elif bk_tomita_time < bk_classic_time and bk_tomita_time < bk_degeneracy_time:
    print('Bron-Kerbosch with Tomita pivoting is faster than the other two variants.')
elif bk_degeneracy_time < bk_classic_time and bk_degeneracy_time < bk_tomita_time:
    print('Bron-Kerbosch with degeneracy ordering is faster than the other two variants.
  ')
```

```
Efficiency of different Bron-Kerbosch variants:
- Bron-Kerbosch classic execution time: 9.559999999986246e-05 s.
- Bron-Kerbosch with Tomita pivoting execution time: 6.86999999999216e-05 s.
- Bron-Kerbosch with degeneracy ordering execution time: 6.690000000020291e-05 s.
Bron-Kerbosch with degeneracy ordering is faster than the other two variants.
```

[22]:
```
# Correctness of different Bron-Kerbosch variants
print('\n' + 'Checking the correctness of different Bron-Kerbosch variants... ', end='')

bk_classic_cliques = list(bk_classic_cliques)
bk_tomita_cliques = list(bk_tomita_cliques)
bk_degeneracy_cliques = list(bk_degeneracy_cliques)

correctness_flag = False

if list(filter(lambda c: c not in bk_classic_cliques, bk_tomita_cliques)) or
  list(filter(lambda c: c not in bk_tomita_cliques, bk_classic_cliques)):
    print('the cliques returned by the classic Bron-Kerbosch algorithm are different from
  those generated by the Tomita pivoting variant.')
elif list(filter(lambda c: c not in bk_classic_cliques, bk_degeneracy_cliques)) or
  list(filter(lambda c: c not in bk_degeneracy_cliques, bk_classic_cliques)):
    print('the cliques returned by the classic Bron-Kerbosch algorithm are different from
  those generated by the degeneracy ordering variant.')
elif list(filter(lambda c: c not in bk_classic_cliques, bk_degeneracy_cliques)) or
  list(filter(lambda c: c not in bk_degeneracy_cliques, bk_classic_cliques)):
    print('the cliques returned by the Tomita pivoting Bron-Kerbosch algorithm are
  different from those generated by the degeneracy ordering variant.')
else:
    correctness_flag = True
    print('the cliques returned by all three algorithms are identical.')

if correctness_flag:
    print('All implemeneted variants are correct.')
else:
    print('There has been an error in the implementation of the Bron-Kerbosch algorithms.
  ')
```

```
Checking the correctness of different Bron-Kerbosch variants... the cliques returned by
all three algorithms are identical.
All implemeneted variants are correct.
```

6.4 Maximum clique

At this point finding the **maximum clique** is as simple as getting the list of all cliques previously calculated and extract from it the **clique with the most elements**.

The Python implementation below is an example of **overloaded function** which allows for greater flexibility in the choice of the input type. Indeed, its main argument `x` can either be:

- a NetworkX undirected graph (in which case the `find_all_maximal_cliques_bk()` function is called in order to compute all cliques first), or
- a list of all maximal cliques, stored as either lists or sets of nodes (which avoids re-computing all cliques if already present in some other variable).

[23]:
```
def find_maximum_clique(x, print_result=False):
    # Check input type
    if isinstance(x, list):
        cliques = x
    else:
        cliques = list(find_all_maximal_cliques_bk(x))

    # Find maximum clique and convert to set (if input is a list of lists)
    maximum_clique = set(cliques[np.argmax(np.array([len(c) for c in cliques]))])

    # Printing
    if print_result:
        print(maximum_clique)

    return maximum_clique
```

Using this function is as simple as writing:

[24]:
```
# Find maximum clique
maximum_clique = find_maximum_clique(bk_degeneracy_cliques, print_result=True)
print('\n' + 'The maximum clique of the random subgraph has length {} and contains nodes:
  \n{}.'.format(len(maximum_clique), maximum_clique), end='')
```

```
{'David W. Wilson', 'R. J. Mathar', 'Graham H. Hawkes', 'Gary W. Adamson', 'Manda Riehl',
'Michael B. Porter', 'Alexandre Wajnberg'}

The maximum clique of the random subgraph has length 7 and contains nodes:
{'David W. Wilson', 'R. J. Mathar', 'Graham H. Hawkes', 'Gary W. Adamson', 'Manda Riehl',
'Michael B. Porter', 'Alexandre Wajnberg'}.
```

7 Conclusions

In this project we built a graph containing all authors extracted from all the comments of 83.218 OEIS JSON sequence files. We learnt about **regular expressions** in order to perform the parsing, and then delved into the variegated world of **cliques** to find one and all maximal cliques of the graph, as well as the largest one.

We studied and correctly implemented **four different algorithms**, three of which as different ways to find all cliques, and introduced less popular (but not less important) concepts such as the **degeneracy** of a graph.

In the end, we not only reached the main goal of the project by parsing the OEIS files and analyzing the graph, but also created a series of **well-readable and efficient Python implementations** of some significant

algorithms, learning along the way about this language's best practices and data structures.

The complete code for this project was originally written as a standalone module in the `mihalcea.py` Python file, which can be executed from the command line with an optional boolean argument, `--build_graph`, in order to either build the graph from scratch or load it from an existing JSON file. All functions described can be imported from said script in order to be used independently in other applications, and are fully documented in the `mihalcea.py` script, too.

8 Testing

This project has been created and successfully tested on the following machine:

- **Motherboard:** MSI MS-B106
- **CPU:** Intel Core i7-6700K @ 4.01 GHz, 8 core
- **GPU:** AMD Radeon RX VEGA64 8GB
- **RAM:** 16 GB DDR4 @ 2133 MHz
- **SSD:** Samsung SSD 850 EVO 500 GB (540/520 MB/s r/w)
- **HDD:** WD Blue 3 TB (180/220 MB/s r/w)
- **OS:** Windows 10 Pro x64 1909
- **IDE:** PyCharm Professional 2021.1
- **Python:** 3.8

9 License

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license. More details are available in the LICENSE file.

References

[1] Joep Kerbosch Coen Bron. "Algorithm 457: finding all cliques of an undirected graph". In: *Communications of the ACM* 16 (9 1973). URL: https://dl.acm.org/doi/10.1145/362342.362367.

[2] Darren Strash David Eppstein Maarten Löffler. "Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time". In: *Algorithms and Computation, ISAAC 2010, Lecture Notes in Computer Science* 6506 (9 2010). URL: https://link.springer.com/chapter/10.1007%2F978-3-642-17517-6_36.

[3] Haruhisa Takahashi Etsuji Tomita Akira Tanaka. "The worst-case time complexity for generating all maximal cliques and computational experiments". In: *Theoretical Computer Science* 363 (1 2006), pp. 28–42. URL: https://www.sciencedirect.com/science/article/pii/S0304397506003586.

[4] Nick Coghlan Guido van Rossum Barry Warsaw. *PEP 8 – Style Guide for Python Code*. URL: https://www.python.org/dev/peps/pep-0008/#function-and-variable-names.

[5] L. Moser J. W. Moon. "On cliques in graphs". In: *Israel Journal of Mathematics* 3 (1965), pp. 23–28. URL: https://link.springer.com/article/10.1007/BF02760024.

[6] Jashua Landau. *PEP 448 – Additional Unpacking Generalizations*. URL: https://www.python.org/dev/peps/pep-0448/.

[7] NetworkX. *Graph - Undirected graphs with self loops*. URL: https://networkx.org/documentation/stable/reference/classes/graph.html.

[8] WolframMathWorld. *Clique*. URL: https://mathworld.wolfram.com/Clique.html.

[9] WolframMathWorld. *Complete Graph*. URL: https://mathworld.wolfram.com/CompleteGraph.html.

[10] WolframMathWorld. *Maximal Clique*. URL: https://mathworld.wolfram.com/MaximalClique.html.