SOMMAIRE

USAGE
▼ DESCRIPTION
  FONCTIONNALITÉS
  LE FORMAT PAR DÉFAUT
  LECTURE DE LA LIGNE DE COMMANDE
  COMMENT LES GRAPHES SONT-ILS GÉNÉRÉS ?
  APPELS À DES PROGRAMMES EXTERNES
▼ OPTIONS
  ▼ AIDE INTÉGRÉE
    `-help|? [<terme>]` | `-<option>|`
    `<graphe> [<paramètre>]... ?`
    `-list` | `-list-options` | `-list-graphs`
    `-version`
    `-html`
  ▼ REQUÊTES
    `-directed` | `-undirected`
    `-loop` | `-noloop`
    `-fast`
    `-variant <v>`
    `-norm <ℓ> [<paramètre>]`
    ▼ `-xy <option> [<paramètre>]...`
      `-xy load <fichier>`
      `-xy box <w> <h>`
      `-xy noise <r> <p>`
      `-xy seed <k> <p>`
      `-xy permutation`
      `-xy mesh`
      `-xy mesh-x <x>`
      `-xy mesh-y <y>`
      `-xy cycle`
      `-xy unif`
      `-xy circle`
      `-xy disk`
      `-xy hyper <p>`
      `-xy convex` | `-xy convex2`
      `-xy polygon <p>`
      `-xy tree <r> <d>`
      `-xy ratio <r>`
      `-xy surface <s>`
      `-xy round <p>`
      `-xy unique`
      `-xy none`
    `-store`
    `-print-prop`
    `-print-test`
    `-id <id>`
    `-seed <s>`
    `-discard`
    `-dup`
    `-output <fichier>`
    `-visu-as <fichier>` | `-visu`
  ▼ GROUPES DE REQUÊTES
    `-group`
    `-group-n <n>`
    `-ungroup`
    `-forget-ids`
    `-shift-ids <s>`
    `-dup-group`

- `-extract`
- `-extract-all`
- `-filter`
- `-sort[-inv]`
- `-output-group <fichier>`

OPÉRATIONS ET ALGORITHMES

- `-op <mode> [<paramètre>]...`
  - `-op identity (= -op del-v 0)`
  - `-op not`
  - `-op loop`
  - `-op noloop`
  - `-op del-e <p>`
  - `-op del-v <p>`
  - `-op permute`
  - `-op redirect <p>`
  - `-op star <n>`
  - `-op apex <n>`
  - `-op blow-up <r>`
  - `-op subst-e <u> <v>`
  - `-op subst-v <v>`
  - `-op cartesian-prod`
  - `-op tensor-prod`
  - `-op half`
  - `-op transpose`
  - `-op simplify`
  - `-op subdiv <n>`
  - `-op prune <d>`
  - `-op maincc`
  - `-op accessible <u_1> … <u_n> .`
  - `-op cc+`
  - `-op union`
  - `-op union-d (= -op union-s .)`
  - `-op union-s <v_1,1> … <v_1,k> … <v_n,1> … <v_n,k> .`
  - `-op union-de`
  - `-op diff`
  - `-op inter`
  - `-op line-graph`
  - `-op line-graph-root` | `-op iligra`
  - `-op iso`
- `-check [variant <v>] <mode> [<paramètre>]...`
  - `-check info`
  - `-check bfs <s>`
  - `-check bellman <s>`
  - `-check stretch`
  - `-check volm`
  - `-check ball <r>`
  - `-check ncc` | `-check connected`
  - `-check dfs <s>`
  - `-check deg` | `-check edge[s]`
  - `-check degenerate`
  - `-check gcolor`
  - `-check kcolor <k>`
  - `-check kcolorsat <k>`
  - `-check kindepsat <k>`
  - `-check ps1` | `-check ps1b` | `-check ps1c` | `-check ps1x <n> <u_1> <v_1> … <u_n> <v_n>`
  - `-check paths <x> <y>`
  - `-check iso`
  - `-check sub` | `-check span`
  - `-check isub`
  - `-check minor`
  - `-check twdeg`
  - `-check tw`
  - `-check clique`
  - `-check routing [hash <h>] [scenario [nomem] <s>] <schéma> [<paramètre>]...`
    - `-check routing cluster <k>`
    - `-check routing dcr <k>`
    - `-check routing agmnt <k>`
    - `-check routing tzrplg <t>`
    - `-check routing hdlbr <k>`
    - `-check routing bc <k>`

RÉGLAGES GLOBAUX

- `-main-seed <s>`
- `-seed-bits <b>`
- `-width <m>`
- `-header` | `-noheader`
- `-caption <titre>`
- `-label [depth <d>] <b>`
- ▼ `-prop <propriété>`
  - `-prop id`
  - `-prop stack`
  - `-prop groups`
  - `-prop group-size`
  - `-prop order`
  - `-prop size`
  - `-prop cc`
  - `-prop deg[min|max]`
  - `-prop radius`
  - `-prop diameter`
  - `-prop degenerate`
  - `-prop gcolor`
  - `-prop girth`
  - `-prop cut-vertex`
  - `-prop tw`
  - `-prop hyper`
  - `-prop clique`
  - `-prop prop`
- ▼ `-test <test> [<paramètre>]...`
  - `-test true`
  - `-test not`
  - `-test and`
  - `-test or`
  - `-test xor`
  - `-test random <p>`
  - `-test prop <sélecteur>` | `-test <propriété> <sélecteur>`
  - `-test same-id`
  - `-test iso`
  - `-test unique`
  - `-test has-minor` | `-test is-minor`
  - `-test has-sub` | `-test is-sub`
  - `-test has-isub` | `-test is-isub`
  - `-test deg <sélecteur>`
  - `-test forest <sélecteur>`
  - `-test is-forest (= -test forest t)`
  - `-test is-tree (= -test forest 1)`
  - `-test cycle (= -test forest t -test not)`
  - `-test bipartite (= -test gcolor <3)`
  - `-test connected (= -test cc 1)`
  - `-test biconnected`
  - `-test ps1` | `-test ps1b` | `-test ps1c` | `-test ps1x <n> <u_1> <v_1> … <u_n> <v_n>`
  - `-test tw2`
- `-format <type>`
- `-xy-as-default`
- ▼ `-vcolor <mode> [<paramètre>]...`
  - `-vcolor none`
  - `-vcolor deg[r]`
  - `-vcolor degm`
  - `-vcolor randg`
  - `-vcolor kcolor <k>`
  - `-vcolor pal <grad>`
  - `-vcolor list`
- ▼ `-vsize`
  - `-vsize none`
  - `-vsize deg`
- ▼ `-view <option> [<paramètre>]...` | `-view no <option>`
  - `-view vsize <f>`
  - `-view scale <s>|<x>,<y>|auto`
  - `-view pos`
  - `-view grid <p>`
  - `-view zero`
  - `-view border`
- ▼ `-dot <option> [<paramètre>]...`
  - `-dot len <p>`
  - `-dot dir <type>`
  - `-dot filter <f>`
  - `-dot attr <s>`

▼ CONTRÔLE DU FLUX DE COMMANDES

- `-while <étiquette>`
- `-n-times <étiquette> <n>`
- `-quit`

ACCESSOIRES

- `-print <texte>`
- `-pause`
- `-chrono`
- `-chrono-reset`
- `-print-seeds`

GRAPHES

GRAPHES DE BASE

- `grid n_1 … n_k .`
- `ring n c_1 … c_k .`
- `cage n c_1 … c_k .`
- `arboricity n k`
- `rarytree n b z`
- `ringarytree h k r p`
- `rectree h f_1 f_2 … f_d .`
- `kpage n k`
- `cactus n`
- `ktree n k`
- `kpath n k`
- `kstar n k`
- `rig n k p`
- `apollonian n`
- `polygon n`
- `planar n f d`
- `hyperbolic p k h`
- `rlt p q d`
- `mikado n`
- `kneser n k r`
- `gpetersen n r`
- `squashed n k p`
- `antiprism n`
- `rpartite a_1 … a_k .`
- `ggosset p d_1 v_1 … d_k v_k .`
- `schlafli`
- `crown n`
- `half-graph n`
- `sum-graph n`
- `flip n`
- `interval n`
- `circle n`
- `permutation n`
- `prime n`
- `paley n`
- `mycielski k`
- `barbell n_1 n_2 p`
- `chess p q u v`
- `sat n m k`
- `calls_tree <définition> f n_1 … n_k .`
- `kout n k`
- `expander n k`
- `margulis n`
- `parachute n`
- `alkane <type> n`
- `icosahedron`
- `rdodecahedron`
- `deltohedron n | trapezohedron n`
- `tutte`
- `hgraph`
- `rgraph | fish`
- `cricket`
- `moth`
- `bull`
- `antenna`
- `suzuki`
- `harborth`
- `doily`
- `herschel`
- `goldner-harary`
- `triplex`
- `jaws`
- `starfish`
- `fritsch`
- `soifer`
- `poussin`

heawood4
errera
kittell
frucht
treep p
halin p
butterfly d
shuffle d
debruijn d b
kautz d b
linial n t
linialc m t
pancake n
bpancake n
gpstar n d
pstar n
hexagon p q
whexagon p q
hanoi n b
sierpinski n b
banana n k
moser
markstrom
robertson
wiener-araya
zamfirescu
hatzel

clebsch n
gear n
helm n
haar n
turan n r
klein p q
flower_snark n
biggs n
udg n r
gabriel n
rng n
knng n k
mst n
thetagone n p k w
pat p q r
line-graph n k
uno n p q
unok n p q k_p k_q
wpsl n p q | upsl n p q | wpsld n p q | upsld n p q
wdis n p q | udis n p q | wdisd n p q | udisd n p q
ngon p c x
behrend p k
rplg n t
bdrg n_1 d_1 … n_k d_k .
fdrg n_1 d_1 … n_k d_k .
caterpillar n

▼ GRAPHES ORIENTÉS

aqua c_1 … c_n .

collatz n a_0 b_0 … a_{k-1} b_{k-1} .

▼ GRAPHES COMPOSÉS

mesh p q *(= grid p q .)*
hypercube d *(= grid 2 … 2 .)*
path n *(= grid n .)*
cycle n *(= ring n 1 .)*
triangle *(= ring 3 1 .)*
torus p q *(= grid -p -q .)*
null *(= ring 0 .)*
stable n *(= ring n .)* | empty n *(= ring n .)*

clique n *(= ring n . -op not)* | complete n *(= ring n . -op not)*
tournament n *(= ring n . -op not -op half -directed -noloop)*
matching n *(= ring n . -op star -1)*
fan p q *(= grid p . -op apex q)*
windmill n *(= ring n . -op star -1 -op apex 1)*
split n k *(= ring n-k . -op apex -k)*

comb n *(= grid n . -op star -1)* | centipede n *(= grid n . -op star -1)*

sunlet n *(= grid -n . -op star -1)*

bipartite p q *(= rpartite p q .)*

utility *(= rpartite 3 3 .)*

domino *(= grid 2 3 .)*

kite *(= banana 1 3 -op not)*

dart *(= sum-graph 5)*

parapluie n *(= parachute n -op not)*

hourglass *(= barbell 3 3 0)*

cuboctahedron *(= linial 4 2)*

octahedron *(= antiprism 3)*

d-octahedron d *(= ring d . -op star -1 -op not)*

tetrahedron *(= ring 4 . -op not)*

cube *(= crown 4)* | hexahedron *(= crown 4)*

associahedron *(= flip 6)*

johnson n k *(= kneser n k k-2 -op not, si k≥2)* | johnson n k *(= kneser n n-1 0 -op not, si k=1)*

odd n *(= kneser 2n-1 n-1 0)*

biggs-smith *(= biggs 17)*

claw *(= rpartite 1 3 .)*

star n *(= rpartite 1 n .)*

tree n *(= arboricity n 1)*

outerplanar n *(= kpage n 1)*

squaregraph n *(= planar n 4 4)*

random n p *(= ring n . -op not -op del-e 1-p)*

netgraph *(= sierpinski 2 3 -op not)*

sunflower n *(= cage 2n 2 2 .)*

gem *(= grid 4 . -op apex 1)*

egraph *(= grid 3 . -op star -1)*

tgraph *(= banana 1 3)* | fork *(= banana 1 3)*

ygraph *(= banana 3 1)*

cross *(= banana 1 4)*

knight p q *(= chess p q 1 2)*

antelope p q *(= chess p q 3 4)*

camel p q *(= chess p q 1 3)*

giraffe p q *(= chess p q 1 4)*

zebra p q *(= chess p q 2 3)*

king p q *(= udg pq 1 -norm Lmax -xy mesh-x p)*

petersen *(= kneser 5 2 0)*

tietze *(= flower_snark 3)*

mobius-kantor *(= gpetersen 8 3)*

dodecahedron *(= gpetersen 10 2)*

desargues *(= gpetersen 10 3)*

durer *(= gpetersen 6 2)*

prism n *(= gpetersen n 1)*

cylinder p q *(= grid p -q .)*

nauru *(= pstar 4)*

heawood *(= cage 14 5 -5 .)*

franklin *(= cage 12 5 -5 .)*

mcgee *(= cage 24 12 7 -7 .)*

bidiakis *(= cage 12 -4 6 4 .)*

dyck *(= cage 32 5 0 13 -13 .)*

pappus *(= cage 18 5 7 -7 7 -7 5 .)*

tutte-coexter *(= cage 30 -7 9 13 -13 -9 7 .)*

gray *(= cage 54 7 -7 25 -25 13 -13 .)*

chvatal *(= cage 12 3 6 3 6 6 3 6 -3 3 -3 3 3 .)*

grotzsch *(= mycielski 4)*

hajos *(= sierpinski 2 3)*

house *(= grid 5 . -op not)*

wagner *(= ring 8 1 4 .)*

mobius n *(= ring n 1 n/2 .)*

ladder n *(= grid 2 n .)*

diamond *(= grid 2 . -op apex 2)*

gosset *(= ggosset 8 2 3 6 -1 .)*

wheel n *(= ringarytree 1 0 n 2)*

web n r *(= ringarytree r 1 n 2)*

binary h *(= ringarytree h 2 2 0)*

arytree h k r *(= ringarytree h k r 0)*

rbinary n *(= rarytree n 2 0)* | rbinaryz n *(= rarytree n 2 1)*

tw n k *(= ktree n k -op del-e .5)*

pw n k *(= kpath n k -op del-e .5)*

tadpole n p (= barbell −n 1 p) | dragon n p (= barbell −n 1 p)
lollipop n p (= barbell n 1 p)
pan n (= barbell −n 1 1)
banner (= barbell −4 1 1)
paw (= barbell −3 1 1)
theta0 (= barbell −5 −5 −2)
nng n (= knng n 1)
td-delaunay n (= thetagone n 3 3 1)
theta n k (= thetagone n 3 k 6/k)
dtheta n k (= thetagone n 3 ⌊k/2⌋ 6/k)
yao n k (= thetagone n 0 k 2/k)
percolation a b p (= udg a·b 1 −norm L1 −op del-e 1-p −xy mesh a b)
hudg n r (= udg n r −norm hyper −xy hyper r)
point n (= ring n . −xy unif)
star-polygon n (= ring n 1 . −xy disk)
convex-polygon n (= ring n 1 . −xy convex)
regular n d (= fdrg n d .)
cubic n (= fdrg n 3 .)
plrg n t (= bdrg n_1 d_1 … n_k d_k .)
tree_fibo n (= calls_tree "…" f n .)
tree_part n k
tree_binom n k (= calls_tree "…" binom n k .)
syracuse n (= collatz n 1 0 3 1 .)
kakutami_3x+1 n (= collatz n 1 0 6 2 .)
kakutami_5x+1 n (= collatz n 3 0 30 6 3 0 2 0 3 0 30 6 .)
kakutami_7x+1 n (= collatz n 15 0 210 30 15 0 10 0 … 210 30 .)
farkas n (= collatz n 6 0 6 6 6 0 4 0 6 0 … 18 6 .)

▼ GRAPHES EXTERNES

load <fichier>[:<sélecteur>]
load+ <fichier>[:<sélecteur>]
load-str <description>
load-str+ <description>

▼ HISTORIQUE

v1.2, octobre 2007
v1.3, octobre 2008
v1.4, novembre 2008
v1.5, décembre 2008
v1.6, décembre 2009
v1.7, janvier 2010
v1.8, juillet 2010
v1.9, août 2010
v2.0, septembre 2010
v2.1, octobre 2010
v2.2, novembre 2010
v2.3, décembre 2010
v2.4, janvier 2011
v2.5, mars 2011
v2.6, juin 2011
v2.7, octobre 2011
v2.8, novembre 2011
v2.9, février 2013
v3.0, octobre 2013
v3.1, décembre 2013
v3.2, mai 2014
v3.3, juillet 2014
v3.4, février 2015
v3.5, mars 2015
v3.6, juin 2015
v3.7, juillet 2015
v3.8, novembre 2015
v3.9, février 2016
v4.0, mars 2016
v4.1, avril 2016
v4.2, mai 2016
v4.3, juin 2016
v4.4, juillet 2016
v4.5, août 2016
v4.6, septembre 2016
v4.7, janvier 2017
v4.8, mars 2017
v4.9, juin 2017

v5.0, août 2017
v5.1, août 2018
v5.2, juin 2019
v5.3, janvier 2021
v5.4, février 2023
v6.0, avril 2024
v6.1, octobre 2024
v6.2, novembre 2024
v6.3, février 2025

USAGE

```
gengraph [-html] [-help|?|-list|-version]
gengraph -help|? <mot>
gengraph <graphe>|-<option> [<paramètre>]... ?
gengraph [<graphe>|-<option> [<paramètre>]...]...
```

DESCRIPTION

Génère et traite des graphes. Ceux-ci sont conçus à partir du nom d'une classe de graphes le plus souvent adjointe de paramètres (typiquement le nombre de sommets), et peuvent subir diverses opérations, être analysés et être écrits dans des fichiers (ou la sortie standard) dans un format texte simple ou un format graphique.

Le programme inclut un système d'aide très complet qui tire parti du formatage de son manuel.

FONCTIONNALITÉS

Aide intégrée.

Ex :
```
gengraph
gengraph -html -help
gengraph -list-graphs | grep -P ' {6}' | sort
gengraph -op not ?
gengraph -xy unique ?
gengraph tree ?
gengraph ? arbre
```

Génération de graphe dans un format texte simple.

Ex :
```
gengraph tutte
gengraph hypercube 8
```

Génération de graphe dans un format graphique.

Ex :
```
gengraph rdodecahedron -visu
gengraph tree 80 -visu-as g.html
gengraph web 10 3 -visu-as g.html
gengraph gabriel 50 -caption "Gabriel with n=50" -visu
gengraph sierpinski 7 3 -visu
gengraph udg 400 .1 -visu-as g.svg
gengraph arytree 6 3 3 -dot filter circo -visu
gengraph dyck -dot filter circo -visu
gengraph ringarytree 4 2 3 0 -label 1 -visu
gengraph prime 15 -directed -label 1 -visu
gengraph aqua 3 2 1 . -label 1 -dot filter dot -visu
```

Positionnement des sommets.

Ex :
```
gengraph gabriel 2000 -xy seed 1 0.15 -visu
gengraph gabriel 700 -xy seed 1 -0.3 -visu
gengraph udg 400 .1 -xy seed 3 1.5 -visu-as g.svg
gengraph rng 30 -xy box 15 15 -xy round 0 -view grid 16 -visu
```

Coloration des sommets.

Ex :
```
gengraph udg 400 -1 -vsize -vcolor deg -visu-as g.svg
gengraph arboricity 100 2 -vcolor degr -visu
gengraph rplg 300 3 -op maincc -vcolor degr -vcolor pal wz -vsize -visu
```

Exécution d'algorithme sur un graphe.

Ex :
```
gengraph cycle 3 -directed -check bfs 0
gengraph tutte -prop diameter -print-prop
gengraph linial 7 3 -check kcolorsat 3 | glucose -model
```

Application d'opérations en notation polonaise inversée.

Ex :
```
gengraph mesh 7 3 -op not
gengraph mesh 50 50 -op del-e .5 -op maincc -fast -visu
gengraph random 20 .1 -op line-graph
gengraph star 10 -group path 3 -n-times - 5 -op union-s 1 1 1 1 1 . -check
```

Certains algorithmes produisent un résultat au format Dimacs CNF compatible avec des solveurs SAT comme Glucose et MiniSat :

- https://www.labri.fr/perso/lsimon/fr/research/glucose/
- http://minisat.se/

LE FORMAT PAR DÉFAUT

Se reporter à l'option `-format` pour voir les autres formats de sortie possibles.

Le **format par défaut**, dit aussi **format standard**, **format simple** ou encore *format simple de Gen-Graph*, est une liste d'arêtes ou de chemins écrits au format texte.

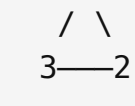
Ce format minimaliste est relativement proche de celui du format DOT de Graphviz. Les sommets sont numérotés consécutivement de 0 à n-1 où n est le nombre de sommets présents dans le graphe. Une arête entre i et j est représentée par `i-j`, un arc de i vers j par `i->j`. Les sommets isolés sont simplement représentés par le numéro du sommet tout seul.

Pour une représentation plus compacte, les arêtes (ou arcs) consécutives d'un chemin du graphe peuvent être regroupées en blocs `i-j-k-…`. Par exemple, les deux arêtes `3-5` et `5-8` peuvent être regroupées en `3-5-8`. Également, les arêtes (ou arcs) d'une étoile peuvent être groupées avec `i-(j k …)`. Par exemple, `3-(5 7 8)` représente les arêtes `3-5`, `3-7` et `3-8`. Il n'est pas possible cependant de combiner chemins et étoiles, comme `3-(5-7-8)` ou `3-(5-(7 8))`. Toutefois, `3-5-(7 …)` est correct, mais pas `3-(5 6)-7` ni même `(3 5)-6`.

Les sommets isolés et les arêtes (ou les blocs d'arêtes) sont séparés par des espaces ou des sauts de ligne. Une boucle sur un sommet i est codée par `i-i`. Les arêtes multiples sont codées par la répétition d'une même arête, comme par exemple `i-j i-j`, ou encore `i-j-i` (même convention pour les arcs `i->j->i`).

Ex :

```
0 1-2-3-1
```



représente un graphe à 4 sommets, composé d'un sommet isolé (0) et d'un cycle à trois sommets (1, 2, 3). À droite est donnée une représentation graphique possible.

Ex :

```
4-2-1-0-3-2-5
```

représente un graphe à 6 sommets composé d'un cycle de longueur 4 et de deux sommets de degré 1 attaché à 2. On aurait pu coder le même graphe avec l'expression `2-(1 3 4 5) 1-0-3`. En voici une représentation graphique possible :

Ex :

```
     1
    / \
  4-2   0
   / \ /
  5   3
```

Il y a également des sucres pour les sommets universels : écrire `i-*`, `i->*` ou `*->i` permet de connecter le sommet i à tous les autres sommets du graphe. `*-i` et `i->*->j` sont invalides.

Plusieurs graphes au sein d'une même description peuvent être identifiés par un entier naturel à placer entre crochets.

Ex : `[17] 0-1 [22] 0->1->2->0`

représente un groupe composé de deux graphes : un chemin à deux sommets (d'identifiant 17) ainsi qu'un cycle orienté à trois sommets (d'identifiant 22). Les identifiants n'ont pas à être uniques (voir `-id`).

Le seul sucre susceptible d'être utilisé lors de l'écriture de graphes au format simple par GenGraph est celui qui concerne les arêtes consécutives d'un chemin (`i-j-k`). Les autres (étoiles et sommets universels) sont reconnus lors de la lecture (voir GRAPHES EXTERNES).

Il peut aussi y avoir des commentaires de fin de ligne, préfixés par `//` (comme en C++). Certaines options activent la génération de tels commentaires dans la sortie.

LECTURE DE LA LIGNE DE COMMANDE

Les lignes de la section USAGE ci-dessus présentent de manière simplifiée comment utiliser le logiciel. Mais, plus généralement :

- chaque paramètre donné à GenGraph sur la ligne de commande est soit un début de commande soit un paramètre de commande ;
- une commande est soit un type de graphe à construire (nom et paramètres), soit une option (commençant par `-`) ;
- ces commandes sont lues et traitées l'une après l'autre ;
- chaque graphe à construire est placé au sommet d'une ***pile de requêtes***, traitée en notation polonaise inversée ;
- les effets des options de la section REQUÊTES ci-dessous s'appliquent au graphe au sommet de la pile, tandis que celles de la section GROUPES DE REQUÊTES s'appliquent au groupe de graphes au sommet de la pile (par défaut toute la pile) ;
- les options de la section RÉGLAGES GLOBAUX plus bas affectent les générations de graphes suivantes, et sont écrasées par une nouvelle occurrence de la même option ;

- si la pile de requêtes n'est pas vide à la fin du programme, les graphes correspondants sont générés et affichés sur la sortie standard, en commençant par le plus profond.

NB : on parle indifféremment de pile de graphes ou de pile de requêtes, les graphes n'étant généralement générés que lorsque la requête est consommée par une opération telle que `-check`, `-store` ou `-output`.

L'affichage final est équivalent à `-output-group` (avec tous les groupes concaténés) s'il y a plusieurs graphes et que le format simple est actif, sinon il ressemble à `@1 -output - -print '' -test true -while 1`, en affichant les graphes dans le sens inverse.

Ex :
```
gengraph -format dot-svg caterpillar 50 -output caterpillar.svg \
  -label 2 path 30 -op permute -output path.svg \
  -format simple stable 100
```

Dans cette commande d'exemple :

- l'option `-format dot-svg` définit le format de sortie à utiliser désormais ;
- `caterpillar 50` empile un graphe chenille à 50 sommets (sans le générer) ;
- `-output caterpillar.svg` génère le graphe, l'enregistre dans `caterpillar.svg` (au format `dot-svg` précédemment spécifié) et le dépile ;
- `-label 2` définit le mode d'étiquetage à utiliser désormais ;
- `path 30` empile un chemin à 30 sommets (sans le générer) ;
- `-op permute` programme la permutation des sommets du graphe `path 30` qui vient d'être empilé ;
- `-output path.svg` génère le graphe, l'enregistre dans `path.svg` (toujours au format `dot-svg`) et le dépile ;
- `-format simple` définit le format de sortie à utiliser désormais ;
- `stable 100` empile un stable à 100 sommets.

Il reste alors dans la pile le stable à 100 sommets. Il est donc généré puis affiché, au format simple comme défini à la fin, et avec l'étiquetage défini par `-label 2` (mais qui, avec le format simple, a le même effet que la valeur par défaut). Notez que ce `stable 100` aurait pu être placé n'importe où ailleurs sur la ligne de commande, sauf entre l'empilement d'un graphe et le dépilement correspondant avec `-output`, sans que cela change le résultat ; le stable serait alors resté au fond de la pile sans être modifié, jusqu'à être généré et affiché à la fin, en utilisant les réglages en vigueur à ce moment-là (`-label 2 -format simple`).

COMMENT LES GRAPHES SONT-ILS GÉNÉRÉS ?

Plusieurs modes de génération peuvent exister pour un même graphe. Ils sont tous paresseux : aucun ne stocke le graphe entier en mémoire (sauf pour certains graphes de taille O(n)) et ne requièrent au pire qu'une structure de données en O(n) (n étant le nombre de sommets).

Le mode par défaut, qui fonctionne pour tous les graphes, est la matrice d'adjacence : le générateur appelle une fonction adj(i,j) qui lui indique si les sommets i et j (entiers entre 0 et n-1) sont adjacents. Le graphe est produit en parcourant la matrice d'adjacence du graphe (seulement le triangle supérieur dans le cas non orienté) et en appelant adj(i,j) pour chaque paire ou couple de sommets. Complexité en temps : $O(n^2)$.

Il y a deux autres modes de génération, qui ne sont disponibles que pour certains graphes, et qui peuvent être activés avec l'option `-fast`. Le premier de ces modes fournit une par une les arêtes du graphe. Le second fournit le voisinage des sommets, ce qui permet de parcourir la liste d'adjacence du graphe. Complexité en temps : O(n+m) (m étant le nombre d'arêtes).

L'option `-check info` permet de savoir quels modes sont disponibles pour un graphe.

APPELS À DES PROGRAMMES EXTERNES

GenGraph est voulu léger et avec un minimum de dépendances.

Pour la visualisation de graphes, l'application fait appel au programme `dot` de Graphviz (https://graph-viz.org/), qui doit donc être installé pour que cette fonction soit mise en œuvre.

Le programme `less` est parfois sollicité pour l'affichage de l'aide en ligne de commande (dite aide intégrée), mais il n'est pas requis. Dans tous les cas, il est possible de rediriger la sortie avec un tube (`|`) dans la commande, vers `less -RS`, `tee` ou autre.

GenGraph ne requiert normalement pas autre chose qu'un environnement POSIX, un compilateur GCC ou Clang et GNU Make pour être compilé et fonctionner.

OPTIONS

AIDE INTÉGRÉE

Le système d'aide utilise l'encodage de caractères UTF-8 et les codes d'échappement ANSI pour la couleur, que le terminal doit donc prendre en charge.

Actuellement, l'ouverture de l'aide cause l'arrêt du programme, c'est-à-dire que les paramètres suivant celui qui a causé l'ouverture de l'aide ne sont pas traités.

`-help|? [<terme>]`
`-<option>|<graphe> [<paramètre>]... ?`

Affiche l'aide intégrée, lue depuis le fichier `MANUAL.txt`. Si `terme` est précisé, alors les options et noms de graphe contenant `terme` sont affichés. La variante `-<option>|<graphe> ?` affiche une aide détaillée sur une option ou un graphe précis.

La première variante tente de solliciter `less` (plus précisément `less -RS`) pour faciliter la visualisation, mais dans tous les cas la sortie brute peut être récupérée en faisant un tube vers une commande telle que `tee`.

Ex :
```
gengraph
gengraph -html -help
gengraph ? arbre
gengraph ktree ?
gengraph ? hedron | tee
gengraph ? planaire | sed 's,\x1B\[[0-9;]*[a-zA-Z],,g' > planaires.txt
```

La dernière commande supprime les codes ANSI de formatage du manuel prévus pour le terminal, et enregistre le résultat dans un fichier.

Attention : Le caractère `?` est un métacaractère pour la plupart des interpréteurs de commandes. Il risque d'être remplacé par des noms de fichiers d'un seul caractère s'il y en a dans le répertoire en cours. Pour éviter cela, il faut échapper le métacaractère en écrivant `\?` ou `'?'`.

`-list`
`-list-options`
`-list-graphs`

Affiche un sommaire de la liste des commandes, c'est-à-dire tous les titres de section contenus dans les sections OPTIONS et GRAPHES du manuel (fichier `MANUAL.txt`). `-list-options` et `list-graphs` permettent de demander seulement l'une ou l'autre section. Tout comme avec `-help`, `less` est sollicité si présent.

Avec `-html`, cette option permet d'avoir le sommaire développé automatiquement.

Ex :
```
gengraph -list-graphs | grep -P ' {6}' | sort
```

Cette commande fournit la liste de tous les graphes disponibles, triés par ordre alphabétique.

`-version`

Affiche des informations sur la version de GenGraph installée, extraites du manuel (fichier `MANUAL.txt`).

`-html`

Utilise l'aide HTML plutôt que l'aide intégrée pour les options `-help|? [<terme>]`, `-list` et les paramètres incorrects. La variable d'environnement `URL_OPENER` peut être définie pour déterminer le programme auquel fournir le chemin vers le fichier.

REQUÊTES

Ces options s'appliquent aux requêtes de graphe au sommet de la pile. La plupart n'affectent que le graphe tout au sommet, mais certaines consomment plusieurs graphes (cas de `-check iso`). Les options affectant un nombre variable de graphes sont présentées dans la section suivante (GROUPES DE REQUÊTES).

`-directed`
`-undirected`

L'option `-directed` passe le graphe en mode orienté. Lors de sa génération, les n^2 arcs possibles seront testés. En format simple ou DOT, un arc apparaît comme `i->j`, au lieu de `i-j` ou `i--j` pour une arête.

L'option `-undirected` permet de revenir à la situation par défaut, qui conduit à tester n(n-1)/2 paires de sommets lors de la génération.

Tous les graphes ne sont pas prévus pour fonctionner avec l'option `-directed` : certaines fonctions d'adjacence supposent i<j et `-undirected`. Avec `-directed`, la plupart des graphes vont apparaître comme orientés symétriques.

Ex :
```
gengraph clique 5 -directed
gengraph cycle 5 -directed
```

`-loop`
`-noloop`

Autorise ou interdit les boucles lors de la génération du graphe en mode normal (non `-fast`). Elles sont par défaut autorisées avec les graphes orientés et interdites avec les graphes non orientés ; l'utilisation de `-[un]directed` écrase la configuration mise en place par `-[no]loop`.

L'existence de ces options vient du fait qu'en mode normal, des comparaisons sont souvent effectuées sans prendre en compte le cas i=j, ouvrant la porte à de nombreuses boucles inattendues si elles n'étaient pas désactivées par défaut.

`-fast`

Programme la génération rapide du graphe en passant par la génération arête par arête plutôt que par la matrice d'adjacence. Dans ce cas, le générateur regardera d'abord si le graphe offre une génération arête par arête. Si ce n'est pas le cas, elle est émulée en utilisant la liste d'adjacence si disponible, et sinon la matrice d'adjacence. Notez que dans ce dernier cas, `-fast` rendra en fait la génération plus lente que la configuration par défaut qui utilise la matrice d'adjacence directement, sans émulation. Dans les autres cas, la génération est généralement plus rapide puisqu'elle passe à O(m) ou O(n+m), au lieu de $O(n^2)$.

L'option `-check info` permet de savoir quels modes de génération sont disponibles pour un graphe. Le temps pris par chacun des deux types de génération peut être mesuré avec `-check info` ou `-chrono`.

Alors que le mode de génération par défaut fournit toujours les arêtes dans l'ordre lexicographique et sans doublon, celui-ci est susceptible de générer des multi-arêtes et d'utiliser un autre ordre. `-op simplify` permet d'obtenir le même résultat qu'avec le mode de génération par défaut.

Ex : `gengraph load fichier -op del-v 0.3 -fast -check ncc`

Cet exemple permet de calculer le nombre de composantes connexes sur un sous-graphe contenu dans un fichier, le tout en temps linéaire. D'autres graphes peuvent supporter une génération rapide si elle est implantée dans l'initialisation de la fonction d'adjacence.

`-variant <v>`

Demande la variante `v` du graphe. Ce paramètre est utilisé par certaines classes de graphes, par exemple `tree_part`. La valeur par défaut est v=0.

`-norm <ℓ> [<paramètre>]`

Fixe la norme d'un vecteur (x,y) du plan (ou la fonction de distance entre deux points du plan) pour l'adjacence de certains graphes géométriques (dont `udg`, `gabriel`, `rng`, `nng`...). Par défaut, c'est la norme euclidienne (`L2`) qui est utilisée. Les valeurs possibles pour ℓ sont :

- `L1` → |x|+|y|, distance de Manhattan
- `L2` → $\sqrt{x^2+y^2}$, norme euclidienne
- `Lp <p>` → (|x|^p+|y|^p)^(1/p), norme Lp avec `p`>0
- `Lmax` → max{|x|,|y|}
- `Lmin` → min{|x|,|y|}
- `poly <p>` → distance polygonale de paramètre `p`
- `hyper` → distance hyperbolique

Dans certains cas, il s'agit de pseudo-norme ou pseudo-distance. Par exemple, la norme `Lmin` ne vérifie pas l'inégalité triangulaire, la norme `poly <p>` n'est pas symétrique pour `p` impair, la norme `Lp p`

est une norme si `p`≥1, le disque unité n'étant plus convexe si 0<`p`<1, et la norm `hyper` est une distance mais pas une norme, car elle n'est pas invariante par translation.

La norme `poly <p>` est le rayon du cercle inscrit dans le polygone régulier convexe à `p` côtés contenant (x,y), le polygone étant centré en (0,0) et orienté de façon à avoir vertical son côté le plus à droite. Ainsi, `poly 4` correspond à la norme `Lmax`. Une valeur de `p`<3 est interprétée comme `p`=+∞ ce qui correspond à la norme euclidienne.

La norme `hyper`, la distance hyperbolique, n'est définie que pour des points du disque ouvert unité centré en (0,0). C'est une distance, mais pas une norme car elle n'est pas invariante par translation.

`-xy <option> [<paramètre>]...`

Cette option contrôle la façon dont sont générées les coordonnées des sommets. Elle rend le graphe géométrique s'il ne l'était pas. Par défaut, les positions sont tirées aléatoirement uniformément dans le carré $[0, 1]^2$ (`-xy unif`), mais cela peut être changé grâce aux autres options `-xy`. Notez bien que, même si c'est improbable, deux sommets peuvent avoir les mêmes positions (voir l'option `-xy unique`).

Ex : `gengraph path 20 -xy unique -visu`

`-xy load <fichier>`

Charge les positions à partir du fichier `fichier`, ou de l'entrée standard si fichier est `-`. Cela permet de tester les adjacences d'un graphe géométrique à partir de positions prédéterminées. Le format est celui de `-format xy`.

Ex : `gengraph gabriel 10 -xy load fichier.pos`

La structure du fichier texte doit être :

```
n
x_1 y_1
x_2 y_2
…
x_n y_n
```

où n est le nombre de positions. Les positions x_i y_i ne sont pas forcément dans l'intervalle [0, 1]. Notez qu'avec l'option `-format xy`, il est possible d'effectuer la transformation d'un fichier de positions. L'exemple suivant normalise les coordonnées du fichier `g.pos` dans le carré unité pour 10 positions :

Ex : `gengraph empty 10 -xy load g.pos -xy box 1 1 -format xy`

`-xy box <w> <h>`

Effectue un redimensionnement des positions de sorte qu'elles se situent dans le rectangle [0, w] × [0, h]. En prenant w=h=1, les coordonnées seront renormalisées dans le carré $[0, 1]^2$. Cette opération est effectuée juste avant la génération des arêtes, mais après avoir effectué les opérations `-xy noise` et `-xy load` éventuellement programmées.

`-xy noise <r> <p>`

Effectue une perturbation aléatoire sur les positions des sommets. Le déplacement de chaque sommet est effectué dans sa boule de rayon `r` (pour p>0) selon une loi en puissance de paramètre `p`. Prendre p=0.5 pour une perturbation uniforme dans cette boule, p>0.5 pour une concentration des valeurs vers le centre et p<0.5 pour un écartement du centre. Les valeurs <0 de `p` donnent des écartements au-delà du rayon `r`.

Plus précisément, une direction (angle dans $[0, 2\pi[$) est choisie aléatoirement uniformément, puis, selon cette direction, un décalage aléatoire est effectué selon une loi en puissance : si x est uniforme dans [0, 1], le décalage sera d(x)=r·x^p. Après cette opération, il est possible que les points ne soient plus dans le rectangle d'origine, ce qui peut bien sûr être corrigé par `-xy box`.

`-xy seed <k> <p>`

Génère les points à partir (ou autour) de k>0 graines. Les graines sont choisies uniformément dans le carré $[0, 1]^2$ puis centrées par rapport à leur barycentre. Chaque point est alors tiré aléatoirement autour d'une des graines et à une distance variant selon une loi en puissance (voir `-xy noise`) de paramètre `p` et de rayon r ≃ √(ln(k+1)/k). Ce rayon correspond au seuil de connectivité pour un Unit Disk Graph à `k` sommets dans le carré $[0, 1]^2$ (voir `udg n r`). On peut obtenir une distribution uniforme dans un disque avec `-xy seed 1 0.5`, le centre étant alors en (1/2,1/2), alors qu'avec `-xy disk` il est en (0, 0).

Ex :
```
gengraph point 1000 -xy seed 1 1
gengraph point 1000 -xy seed 1 0.5
```

`-xy permutation`

Génère les points correspondant à une permutation π aléatoire uniforme. Le point i aura pour position (i,π(i)).

`-xy mesh`

Place les sommets sur une grille carrée.

`-xy mesh-x <x>`

Place les sommets sur une grille de `x` colonnes.

`-xy mesh-y <y>`

Place les sommets sur une grille de `y` lignes.

`-xy cycle`

Génère les points régulièrement espacés le long d'un cercle de centre (0,0) et de rayon 1. Les points sont ordonnées selon l'angle de leurs coordonnées polaires.

Ex : `gengraph cycle 10 -xy cycle -visu`

`-xy unif`

Génère les points aléatoirement uniformément dans le carré $[0,1]^2$. C'est la distribution par défaut.

`-xy circle`

Génère les points le long d'un cercle de centre (0,0) et de rayon 1, en suivant une loi aléatoire uniforme. Les points sont ordonnées selon l'angle de leurs coordonnées polaires.

`-xy disk`

Génère les points aléatoirement uniformément dans le disque unité de centre (0, 0) triés selon l'angle de leurs coordonnées polaires. Cette distribution permet de générer, par exemple, un polygone *star-shaped* (`star-polygon`). La distribution est similaire à l'option `-xy seed 1 0.5` sauf que les points sont ordonnées.

Ex :
```
gengraph cycle 25 -xy disk -visu
gengraph gabriel 300 -xy disk -xy ratio 2/3 -visu
```

Remarque : les points sont générés avant l'application des options comme `-xy round`, `-xy noise`, ou `-xy unique` qui modifient les coordonnées et qui peuvent donc produire des croisements avec le graphe cycle par exemple.

`-xy hyper <p>`

Génère les points aléatoires selon une loi exponentielle de paramètre `p` dans le disque unité de centre (0, 0), chaque point étant placé à une distance exp(-p·u) du centre où u est uniforme dans [0, 1]. Les points sont triés selon l'angle de leurs coordonnées polaires.

`-xy convex`
`-xy convex2`

Génère les points aléatoirement en position convexe à l'intérieur d'un cercle de rayon 1 et de centre (0, 0), ce qui peut être modifié par `-xy ratio`. Ils sont numérotés consécutivement selon le parcours de l'enveloppe convexe. On les génère comme suit, n étant le nombre points à générer. Inductivement, une fois que n-1 points en position convexe ont été générés, on choisit un angle α aléatoire du cercle de rayon 1 et de centre (0, 0) supposé à l'intérieur du convexe. On détermine ensuite la partie S du segment d'angle α où chacun des points de S=[a,b[forment avec les n-1 points précédant un convexe. Enfin, on choisit aléatoirement un point de S selon la probabilité $\sqrt{|b-a|}$ pour obtenir n points en position convexe. Les angles des trois premiers points sont choisis parmi trois secteurs non adjacents d'angle $\pi/3$ si bien que l'origine est toujours à l'intérieur de l'ensemble convexe.

La variante `-xy convex2` génère également des points aléatoires en position convexe, selon la méthode suivante. On génère n points aléatoires u_i du carré $[0, 1]^2$ puis on calcule les vecteurs différences v_i ≡ u_{i+1}-u_i (mod n). Les vecteurs (dont la somme est nulle) sont ensuite triés par angle croissant, puis les points en position convexe sont obtenus de proche en proche en ajoutant chacun des vecteurs v_i. Cette méthode tend à générer des points proches d'un cercle, chaque angle et chaque longueur entre deux points consécutifs suivant une loi normale.

Ex :
```
gengraph cycle 25 -xy convex -visu
gengraph dtheta 100 6 -xy convex -visu
```

L'ordre des sommets peut être modifié par certaines options (voir la remarque à propos de `-xy disk`).

`-xy polygon <p>`

Génère des points aléatoires uniformément dans un polygone convexe régulier à p≥3 côtés inscrit dans le cercle de centre (0, 0) et de rayon 1 de sorte qu'un des côtés du polygone soit vertical. Les sommets ne sont pas spécifiquement ordonnés. Pour une distribution uniforme dans un disque, soit lorsque p=+∞, utiliser `-xy disk`. L'option pour p=4 est similaire à `-xy unif`, mais fait la distribution dans le carré [-c, +c] × [-c, +c] où $c = \cos(\pi/4) = ½\sqrt{2} \simeq 0{,}707\ldots$ au lieu du carré $[0, 1]^2$.

`-xy tree <r> <d>`

Place les sommets sur un arbre déterminé par le parcours en profondeur d'un graphe.

Cette option absorbe le graphe au sommet de la pile, et s'applique au graphe précédent. Le graphe absorbé est généré en mémoire dès que cette option est rencontrée. Lorsque les positions du graphe précédent sont générées, le graphe absorbé est alors parcouru en profondeur. Les deux graphes doivent avoir exactement le même nombre de sommets. Pour positionner correctement les sommets d'un arbre, il suffit d'utiliser l'option `-dup`.

Le paramètre `r` est la racine de l'arbre, c'est-à-dire le numéro du sommet dont le parcours doit partir. `d` quant à lui est la direction de l'arbre : ordonnées croissantes si `d`=0, abscisses croissantes si `d`=1.

L'algorithme place chaque sommet au milieu de ses descendants sur un axe, et sur l'autre axe (celui déterminé par `d`), il place les sommets selon leur profondeur dans le parcours.

Tous les sommets doivent être accessibles depuis le sommet initial dans le graphe absorbé. Si on a obtenu une k-orientation, elle est normalement dans le mauvais sens : il faut alors utiliser `-op transpose` pour que le parcours en profondeur fonctionne.

Ex :
```
gengraph tree 50 -dup -xy tree 25 1 -label -1 -format tikz
gengraph tree_fibo 10 -op transpose -dup -xy tree 0 0 -view scale .05,-.1 -
```

`-xy ratio <r>`

Modifie les distributions de points faisant intervenir une forme de largeur 1 et de hauteur `r`, comme : `-xy unif`, `-xy circle`, `-xy cycle`, `-xy convex`, `-xy disk`, `-xy seed`. La valeur par défaut est r=1. Le réel r>0 est donc le ratio de la hauteur par la largeur de la forme. Par exemple, pour la distribution par défaut (`-xy unif`), les points seront aléatoires uniformes dans le rectangle [0, 1] × [0, r]. Si la forme est un cercle (`-xy circle` ou `-xy disk`), alors la forme devient une ellipse dont le rayon horizontal est 1 et le vertical `r`. Dans le cas de `-xy seed`, les graines sont alors générées dans le rectangle [0, 1] × [0, r]. `r` peut être une fraction de nombres décimaux, par exemple `2.5/4`.

`-xy surface <s>`

Définit la signature `s` de la surface sur laquelle va être construit le graphe géométrique. La surface peut être orientable ou non, avec ou sans bord. Elle est représentée par un polygone convexe régulier inscrit dans un cercle de rayon 1 et dont les 2|s| côtés sont appariés. Cette option se charge également de générer des points aléatoirement uniformes sur la surface. La signature est un mot `s` sur l'alphabet {h,c,b} de longueur |s|=2g, où g est le genre de la surface, indiquant comment sont appariés les 4g côtés du polygone. Chaque côté est apparié avec le côté +2 (le suivant du suivant) ou le côté -2 selon l'une des trois coutures suivantes :

- `h` (handle) = couture orientée ou anse
- `c` (crosscap) = couture non orientée
- `b` (border) = aucune couture

La caractéristique d'Euler de la surface (ou sa courbure) vaut 2-|s| = 2-2g. Certaines signatures ont des synonymes : par exemple, `-xy surface torus` est synonyme de `-xy surface hh`. Les synonymes sont listés dans les exemples ci-dessous.

Ex :
```
-xy surface bb (ou plane ou square)  → plan réel
-xy surface hb (ou cylinder) ....... → cylindre
```

```
-xy surface cb (ou mobius) ......... → ruban de Möbius
-xy surface hh (ou torus) ........... → tore
-xy surface ch (ou klein) ........... → bouteille de Klein
-xy surface cc (ou projective) ..... → plan projectif
```

Cette option active également `-xy polygon 4` et `-xy ratio 1` pour générer des points aléatoires uniformément sur la surface.

`-xy round <p>`

Arrondit les coordonnées à 10^-p près. Il faut que `p` soit un entier < `DBL_DIG` (la précision du type `double` en C), soit p<15 en général. Donc p=0 arrondi à l'entier le plus proche. Cet opérateur est appliqué après `-xy box`. Il sert aussi à préciser le nombre de décimales à afficher pour l'option `-format xy` (par défaut p=6). Par exemple, la combinaison `-xy box 100 100 -xy round -1` permet d'avoir des coordonnées multiples de 10.

`-xy unique`

Supprime les sommets en double, correspondant aux mêmes positions. Cela peut être utile lorsqu'on utilise `-xy round` par exemple. Cette opération est appliquée après toutes les autres, notamment après `-xy box` et `-xy round`. Ceci est réalisé à l'aide d'un tri des points, l'ordre n'est donc pas préservé.

`-xy none`

Supprime toutes les données géométriques de la requête. Pour une classe de graphes géométriques, les données géométriques par défaut seront néanmoins utilisées. Voir aussi `-view pos`.

`-store`

Stocke le graphe en mémoire. Cela requiert un espace en O(n+m), où n est le nombre de sommets et m le nombre d'arêtes. Le graphe est remplacé dans la pile par un graphe `load` déjà chargé, la plupart de ses attributs étant conservés.

Ex : `gengraph random 50 .05 -store -fast -dup -op maincc -fast`

Cet exemple génère un graphe aléatoire à 50 sommets. Il est stocké, mis en mode `-fast`, puis dupliqué de sorte qu'il en existe deux instances. La seconde instance est absorbée par l'opération `-op maincc` qui extrait la principale composante connexe du graphe. À la fin, se trouvent dans la pile le résultat de cette opération ainsi que le graphe d'origine, qui sont donc affichés comme un groupe sur la sortie standard. De cette manière, la matrice d'adjacence du graphe d'origine n'est parcourue entière-

ment qu'une seule fois : au moment du stockage avec `-store`, les autres parcours (causés par `-dup`, `-op maincc -fast` et l'affichage final) travaillant tous sur la liste d'adjacence.

`-print-prop`

Affiche la valeur définie préalablement avec `-prop` pour le graphe au sommet de la pile, sans le dépiler. Le graphe est préalablement stocké si nécessaire, comme avec `-store`. Le nom de la propriété est affiché, ainsi que l'identifiant du graphe (`-id`) s'il en a un.

Ex :
```
gengraph … @1 -prop size -print-prop -discard -test true -while 1
gengraph … -prop … -sort @1 -print-prop -discard -n-times 1 10
```

Le premier exemple affiche le nombre d'arêtes de chaque graphe. Le second exemple permet d'avoir les dix valeurs maximum pour la propriété, en sondant tous les graphes de la pile.

`-print-test`

Affiche le résultat du test préalablement défini avec `-test`, dans le même style que `-print-prop`.

`-id <id>`

Donne au graphe l'identifiant `id`, qui doit être un entier positif. Plusieurs graphes peuvent avoir le même identifiant, mais alors il est impossible de les distinguer, ou bien le graphe le plus proche du sommet de la pile masque les autres.

`-seed <s>`

Initialiser le générateur d'aléatoire de la requête avec la graine `s`, qui doit être un nombre entier positif sur 64 bits. Cela donne la possibilité de retrouver une suite aléatoire.

Si cette option n'est pas utilisée, le générateur est initialisé avec une graine aléatoire produite par le générateur principal (voir `-main-seed`). Cette graine peut être affichée avec les options `-print-seeds`, `-header` et `-caption`. Voir `-seed-bits` pour configurer la valeur maximale de la graine.

Les sous-requêtes des graphes composés, qui ne sont pas accessibles, obtiennent une graine de la part du générateur de la requête qui les contient. Par exemple, utiliser `-seed` sur un graphe `percolation` change la graine du graphe `udg` sous-jacent.

Le générateur affecté par cette option est utilisé pour générer le graphe et les coordonnées des sommets pour les graphes géométriques. Pour le reste (algorithmes `-check` notamment), le générateur principal est utilisé.

`-discard`

Supprime le graphe au sommet de la pile.

`-dup`

Duplique le graphe au sommet de la pile.

`-output <fichier>`

Écrit le graphe dans `fichier` et le dépile. `fichier` peut être `-` pour désigner la sortie standard.

`-visu-as <fichier>`
`-visu`

Enregistre le graphe dans `fichier` et essaie de l'ouvrir. Le format est forcément déterminé par l'extension comme avec `-format auto`.

La variable d'environnement `FILE_OPENER` peut être définie pour déterminer le programme auquel fournir le chemin vers le fichier une fois qu'il est écrit.

Avec `-visu`, le nom du fichier est déterminé par la variable d'environnement `GENGRAPH_VISU`, ou par défaut `gg-visu.pdf`.

GROUPES DE REQUÊTES

La pile de requêtes peut être divisée en plusieurs groupes de requêtes. Concrètement, ces groupes sont délimités par des séparateurs au sein de la pile, placés soit par une option comme `-group` soit par un groupe de graphes comme `load+`. En l'absence de séparateur, la pile entière constitue un groupe. Les séparateurs sont automatiquement supprimés lorsqu'un graphe du groupe précédent est consommé, avant quoi un groupe de taille 0 peut subsister au sommet de la pile.

Les options de cette section agissent sur le groupe au sommet de la pile (les options des autres sections ignorent les groupes). Si un groupe est consommé, cela signifie que son séparateur est supprimé, et le groupe précédent se trouve alors au sommet de la pile.

`-group`

Place un séparateur au sommet de la pile, créant un groupe de taille nulle.

`-group-n <n>`

Groupe les `n` graphes au sommet de la pile. Ils doivent tous être préalablement dans le même groupe.

`-ungroup`

Supprime le dernier séparateur, de sorte à concaténer le dernier et l'avant-dernier groupe. S'il n'y a qu'un seul groupe, cette option est simplement sans effet.

`-forget-ids`

Supprime les identifiants de tous les graphes de la pile, de sorte qu'ils soient regénérés à partir de zéro lors d'un affichage de groupe (`-output-group`).

`-shift-ids <s>`

Ajoute l'entier `s` à tous les identifiants du groupe. `s` peut être négatif ; les identifiants inférieurs à −s sont éliminés.

`-dup-group`

Crée un groupe formé des mêmes graphes que le groupe au sommet de la pile.

`-extract`

Place au sommet de la pile le premier graphe du groupe pour lequel le test défini avec `-test` est vrai, en commençant l'analyse à partir du sommet de la pile. Si aucun graphe ne correspond, c'est une erreur fatale.

`-extract-all`

Place dans un groupe au sommet de la pile tous les graphes du groupe pour lesquels le test défini avec `-test` est vrai, en préservant l'ordre relatif des graphes dans les deux groupes.

`-filter`

Supprime tous les graphes du groupe pour lesquels le test défini avec `-test` est faux.

`-sort[-inv]`

Trie les graphes du groupe selon la valeur préalablement définie avec `-prop`. Au sommet de la pile se trouvera le graphe ayant la plus grande valeur (`-sort`) ou la plus petite (`-sort-inv`).

`-output-group <fichier>`

Écrit le groupe de graphes dans `fichier` sous la forme d'un groupe, c'est-à-dire précédés de leur identifiant entre crochets. Les identifiants non définis sont remplacés par le premier identifiant libre dans le groupe. Le groupe est consommé.

Cette option ne fonctionne qu'avec les formats texte, c'est-à-dire non visuels. Avec le format simple (`-format`), les graphes pourront être chargés par la suite avec `load` ou `load+`.

OPÉRATIONS ET ALGORITHMES

`-op <mode> [<paramètre>]...`

Ces options placent sur la pile une opération, qui fonctionne comme une requête de graphe, qui a absorbé un ou plusieurs graphes au sommet de la pile. Le nombre de graphes absorbés, s'il n'est pas indiqué explicitement dans la description de l'opération, est de 1. Il peut aussi s'agir du groupe au sommet de la pile (voir GROUPES DE REQUÊTES). Ces graphes sont tout bonnement extraits de la pile pour devenir les paramètres de l'opération.

Le caractère orienté ou non orienté (`-[un]directed`) est déterminé par l'opération. Plusieurs opérations peuvent être enchainées, et le paramètre de la requête qui est encore sur la pile prime.

Si elles ne sont pas elles-mêmes paramétrées par une option `-xy`, certaines opérations recopient les positions des sommets du graphe qu'elles absorbent.

Il y a deux types d'opérations, distinguées par la façon dont la génération fonctionne :

- certaines opérations agissent en mémoire : dans ce cas, elles exécutent `-store` sur chaque graphe absorbé, puis génèrent leur propre graphe, en mémoire ;
- les autres opérations utilisent les modes de génération paresseux de leurs opérandes pour prendre également en charge un ou plusieurs modes de génération paresseux.

En l'absence de précision ci-dessous, l'opération prend en charge la génération paresseuse (au moins la génération par matrice). La plupart des opérations prennent en charge un mode `-fast`, qui va généralement utiliser la génération par liste ou par arêtes des graphes absorbés, éventuellement émulée par la génération par matrice. Pour éviter la lente émulation, il est souvent préférable d'utiliser `-store` sur les opérandes.

`-op identity (= -op del-v 0)`

Reprend le graphe tel quel (identité). Cela peut servir à appliquer au graphe une géométrie avec `-xy`, indépendante de celle du graphe sous-jacent, cette dernière étant utilisée pour déterminer les arêtes si le graphe est géométrique.

`-op not`

Calcule le complémentaire.

`-op loop`

Ajoute une boucle sur tous les sommets qui n'en avaient pas encore. Active `-loop`.

`-op noloop`

Supprime les boucles.

`-op del-e <p>`

Supprime chaque arête avec une probabilité `p`.

`-op del-v <p>`

Supprime chaque sommet avec une probabilité `p`, ou −`p` sommets si `p` est un entier négatif. Les arêtes incidentes aux sommets supprimés sont également supprimées.

`-op permute`

Permute les numéros de sommets. Les numéros de sommets demeurent dans [0, n[où n est le nombre de sommets du graphe.

`-op redirect <p>`

Redirige chaque arête `i-j` ou `i->j`, avec une probabilité `p`, en `i-k` ou `i->k`, où k est choisi aléatoirement parmi les sommets. Cette opération peut créer et supprimer des boucles.

Cette opération charge les graphes en mémoire.

`-op star <n>`

Ajoute des sommets pendants (c'est-à-dire de degré 1) aux sommets du graphe absorbé. Si le graphe est orienté, les nouveaux sommets recevront un arc sortant seulement.

- Si n>0, alors n représente le nombre total de sommets ajoutés, chacun des n sommets étant connecté aléatoirement uniformément à un sommet du graphe absorbé.
- Si n<0, alors |n| sommets sont ajoutés à chacun des sommets du graphe.

Avec `-label 1`, les nouveaux sommets portent le nom de leur voisin précédé de « + ».

`-op apex <n>`

Ajoute |n| sommets universels au graphe absorbé, c'est-à-dire connectés à tous les sommets du graphe. Si n<0, les sommets ajoutés formeront une clique, sinon un stable.

Si le graphe est orienté, les sommets ajoutés recevront des arcs sortants vers les sommets du graphe absorbé.

Avec `-label 1`, les nouveaux sommets sont numérotés à partir de 0 après un signe « + ».

`-op blow-up <r>`

Réalise le *blow-up* de paramètre r. Il faut r≥0. Chaque sommet est remplacé par un stable à r sommets, et chaque arête par un biparti complet $K_{r,r}$.

`-op subst-e <u> <v>`

Remplace toutes les arêtes i-j d'un graphe G par un graphe H, en fusionnant i et j avec respectivement les sommets `u` et `v` de H.

Ex : `gengraph star 5 cycle 4 -op subst-e 0 2 -visu`

Cet exemple génère une étoile dont les 5 branches sont des cycles de 4 sommets.

En orienté, substituer toutes les arêtes par un `path 2` est équivalent à utiliser `-op transpose`, sauf que le graphe G est chargé en mémoire.

`-op subdiv n` est équivalent à `path n+2 -op subst-e 0 n+1`.

Avec `-label 1`, les sommets du graphe G gardent le même nom, et les sommets du graphe H ont le format `e:i` où `e` est le numéro de l'arête substituée et `i` est le nom d'origine du sommet.

`-op subst-v <v>`

Remplace tous les sommets u d'un graphe G par un graphe H, en fusionnant u avec le sommet `v` de H.

Avec `-label 1`, les sommets du graphe G gardent le même nom, et les sommets du graphe H ont le format `u:i` où `i` est le nom d'origine du sommet de G et `i` est le nom d'origine du sommet de H.

`-op cartesian-prod`

Réalise le produit cartésien de deux graphes. Le produit cartésien consiste à créer, pour chaque sommet u du premier graphe et chaque sommet v du second, un sommet nommé (u,v), qui se trouve à la fois dans une copie du premier graphe (sommets de même v) et dans une copie du second (sommets de même u).

`-op tensor-prod`

Réalise le produit tensoriel de deux graphes. Ce produit consiste à créer, pour chaque sommet u du premier graphe et chaque sommet v du second, un sommet (u,v), connecté à (u',v') si le premier graphe contient l'arête `u-u'` et si le second contient l'arête `v-v'`.

`-op half`

Ne garde que le triangle supérieur de la matrice d'adjacence, c'est-à-dire que toutes les adjacences i->j avec i>j sont supprimées. Cela n'a normalement d'effet qu'avec les graphes orientés, aussi cette opération est `-directed` par défaut.

`-op transpose`

Transpose le graphe : retourne tous les arcs dans l'autre sens. Cela n'est bien entendu utile qu'avec les graphes orientés, c'est pourquoi cette opération est `-directed` par défaut.

`-op simplify`

Simplifie le graphe : fusionne les arêtes parallèles (arêtes reliant le même couple de sommets). Aussi, les arêtes sont triées. Cependant, les boucles sont conservées ; utiliser `-op noloop` pour s'en débarrasser.

Bien entendu, cet algorithme n'a pas d'intérêt avec le mode de génération par matrice qui est le mode par défaut (voir `-fast`) puisque celui-ci ne peut pas générer d'arêtes parallèles. Cet algorithme est à combiner avec le mode `-fast`.

Le procédé consiste à parcourir la liste d'adjacence triée du graphe.

`-op subdiv <n>`

Subdivise uniformément le graphe de sorte que chaque arête possède `n` nouveaux sommets. Les variantes suivantes sont possibles (`-variant <v>`), avec m le nombre d'arêtes du graphe initial :

- v=0 (valeur par défaut) : subdivision uniforme, chaque arête étant remplacée par un chemin comprenant `n` sommets internes ;
- v=1 : subdivision comprenant un total de `n` nouveaux sommets répartis aléatoirement uniformément sur les m arêtes du graphe ;
- v=2 : comme v=1 sauf que chaque arête est subdivisée au moins une fois (il faut donc que n≥m).

Si n<0, alors c'est équivalent à mettre |n·m| comme paramètre.

Ex :
```
gengraph clique 4 -op subdiv 30 -visu
gengraph binary 4 -op subdiv -2 -variant 2 -visu
```

`-op subst-e` est une généralisation de la variante 0.

Cette opération charge les graphes en mémoire.

`-op prune <d>`

Supprime récursivement tous les sommets de degré ≤ `d`, de sorte qu'il n'en reste aucun après la suppression.

Ex : `gengraph mesh 10 10 -op del-v .3 -op prune 1 -visu`

Cette opération charge les graphes en mémoire.

`-op maincc`

Ne conserve que la composante connexe principale du graphe, c'est-à-dire celle qui comporte le plus grand nombre de sommets. S'il y en a plusieurs, celles ayant les numéros de sommet les plus grands sont ignorées.

Si le graphe absorbé ne nomme pas lui-même ses sommets (`-label 1`), cette opération leur donne comme nom leur identifiant dans le graphe absorbé.

`-op accessible <u_1> … <u_n> .`

Ne conserve que le sous-graphe accessible depuis au moins un des sommets dont l'identifiant se trouve dans l'ensemble {u_1,...,u_n}. Dans un graphe non orienté, il s'agit des composantes connexes contenant ces sommets.

Le nommage fonctionne comme avec `-op maincc`.

Cette opération charge les graphes en mémoire.

`-op cc+`

Crée un groupe constitué des composantes connexes. Tout comme `load+`, il s'agit d'un groupe de graphes : le graphe absorbé sera remplacé sur la pile par le groupe des graphes correspondant à ses composantes connexes. L'algorithme est exécuté dès la lecture de cette commande : les graphes sont immédiatement stockés en mémoire.

Actuellement, cette opération requiert que le graphe absorbé soit préalablement stocké, par exemple avec `-store`.

`-op union`

Réalise l'union des graphes d'un groupe, c'est-à-dire que les sommets sont fusionnés de sorte que le graphe a autant de sommets que le graphe absorbé qui a le plus de sommets, et il y a une adjacence

entre deux sommets si elle existe dans au moins un des graphes absorbés.

Les noms des sommets (`-label 1`) sont ceux du premier graphe absorbé qui a le plus de sommets.

`-op union-d (= -op union-s .)`

Réalise l'union disjointe des graphes d'un groupe. Le nombre de sommets du graphe final est la somme des nombres de sommets de tous les graphes du groupe.

`-op union-s <v_1,1> … <v_1,k> … <v_n,1> … <v_n,k> .`

Réalise une union semi-disjointe. k est le nombre de graphes du groupe absorbé, et n est le nombre de sommets pour lesquels une union est réalisée. v_i,j est le numéro du sommet pris dans le graphe n° j pour le sommet n° i. Un nombre négatif peut être fourni pour ne pas prendre de sommet dans le graphe. Un même sommet peut apparaître plusieurs fois dans la liste et sera alors dupliqué.

Le « s » dans le nom signifie « selected ».

Ex : `gengraph cycle 10 path 6 -op union-s 0 0 5 5 . -fast -label -1 -visu`

`-op union-de`

Réalise l'union des sommets et l'union disjointe des arêtes des graphes d'un groupe. La seule différence par rapport à `-op union` se trouve avec le mode `-fast`, avec lequel les arêtes sont cumulées au lieu d'être fusionnées. Si deux sommets sont adjacents dans plusieurs graphes, ils seront donc reliés par plusieurs arêtes parallèles.

Ex : `gengraph path 11 -dup -op union-de -fast -op subdiv 4 -visu`

`-op diff`

Fait la différence des graphes d'un groupe. Une arête est présente dans le graphe final si et seulement si elle est présente dans le premier graphe et n'est présente dans aucun des autres graphes du groupe. Le nombre de sommet du graphe final est hérité du premier graphe.

`-op inter`

Réalise l'intersection des graphes d'un groupe. Une arête est présente dans le graphe final si et seulement si elle est présente dans tous les graphes du groupe. Le nombre de sommets du graphe final est celui du graphe du groupe qui a le moins de sommets. Les noms des sommets (`-label 1`) sont ceux du premier graphe.

`-op line-graph`

Calcule le graphe ligne dit aussi graphe adjoint. Les arêtes deviennent les sommets, et sont connectées aux arêtes avec lesquelles elles avaient une extrémité en commun. Dans le graphe adjoint d'un graphe orienté, les arcs devenus sommets ont des arcs sortants vers les arcs sortants de leur sommet destination, c'est-à-dire qu'on obtient la transformation suivante :

```
1->2->3
```

```
(1,2)->(2,3)
```

L'algorithme utilisé est en O(n+m×d), avec n, m et d respectivement le nombre de sommets, le nombre d'arêtes et le degré sortant maximum du graphe à traiter.

Les noms des sommets (`-label 1`) sont composés des identifiants des extrémités des arêtes dans le graphe absorbé.

Voir aussi `line-graph`.

`-op line-graph-root`
`-op iligra`

Calcule le graphe racine (*root graph*) dont le graphe est adjoint (voir `-op line-graph`), s'il y en a un et un seul — sans quoi le graphe `null` sera remis. L'algorithme ne prend en charge que les graphes simples non orientés.

La fonction utilise l'algorithme ILIGRA, en O(n+m) où n est le nombre de sommets du graphe à traiter et m est son nombre d'arêtes.

Attention : L'article sur ILIGRA semble ne pas prendre en compte tous les cas de graphes lignes. Pour un graphe peu dense, il se peut que l'utilisation de cette opération conduise GenGraph à afficher un avertissement comme quoi le résultat est peut-être faux.

Une variante (`-variant 1`) permet d'exécuter la fonction sans vérifier que le graphe a bien un et un seul graphe racine, ce qui permet de baisser sa complexité à O(n). Un graphe sera alors fourni en sortie dans tous les cas (et donc même si le graphe en entrée n'est pas un graphe adjoint).

Le `triangle` est le seul graphe adjoint connexe à avoir plusieurs graphes racines : lui-même et la griffe (`claw`). Les graphes ayant un seul graphe racine ont notamment la propriété de ne pas contenir la griffe comme sous-graphe induit.

Cette opération stocke le graphe opérande pour s'exécuter.

`-op iso`

Cette opération absorbe deux graphes et les charge en mémoire.

Si les deux graphes ne sont pas isomorphes, elle donne le graphe `null`. Sinon, elle donne le premier graphe. L'intérêt par rapport à `-check iso` se trouve avec `-label 1` : les numéros de sommets des deux graphes seront affichés.

`-check [variant <v>] <mode> [<paramètre>]...`

Stocke le graphe sous la forme d'une liste d'adjacence (comme avec `-store`), et lui applique un algorithme. Cette option consomme le graphe ; pour le conserver, il faut utiliser `-dup` au préalable.

Le paramètre `v` permet de contrôler certaines fonctionnalités des algorithmes, la valeur par défaut étant v=0. Par exemple, `-check variant 1 routing cluster -1` calcule une variante du schéma de routage `cluster`.

Certaines options absorbent le graphe au sommet de la pile afin de s'en servir comme paramètre. Il est alors lui aussi stocké, et l'algorithme est appliqué sur le graphe qui se trouvait en dessous.

Certaines options `-check <propriété>` font redondance avec la combinaison `-prop <propriété> -print-prop`, mais affichent davantage d'informations, par exemple le nombre de tests effectués. Aussi, lorsqu'il n'y a pas d'option `-check` de même nom, `-check <propriété>` est un alias pour `-prop <propriété> -print-prop -discard`, afin de conserver certaines options `-check` datant d'avant GenGraph v6.0.

`-check info`

Affiche quelques caractéristiques du graphe, après avoir effectué un tri puis un parcours de sa liste d'adjacence. Indique :

- l'identification du graphe (noms de graphes et d'opérations impliqués avec graines) et ses fonctionnalités (matrice d'adjacence, arêtes, liste d'adjacence pour `-fast` ; noms pour `-label 1`) ;
- si le graphe est orienté, s'il contient des boucles, des multi-arêtes, etc. ;
- l'occupation mémoire du graphe, le temps de génération ou de chargement et le temps de parcours.

Le graphe lui-même n'est pas affiché.

`-check bfs <s>`

Effectue un parcours en largeur d'abord sur le graphe depuis le sommet `s`. La distribution des distances depuis `s` est affichée, ainsi que l'arborescence (-1 indique que le sommet n'a pas de père). La longueur du plus petit cycle passant par `s` est aussi donnée. Elle vaut -1 s'il n'existe pas.

`-check bellman <s>`

Calcule les plus courts chemins depuis le sommet `s` par l'algorithme de Bellman-Ford. Si le graphe est géométrique, le poids de chaque arête correspond à la distance euclidienne entre ses extrémités, sinon il vaut 1 et le résultat sera similaire à un BFS. Dans le cas géométrique, l'étirement maximum depuis `s` est calculé, ainsi qu'un chemin le réalisant. L'implémentation à l'aide d'une file prend un temps linéaire en pratique.

`-check stretch`

Calcule, comme `-check bellman <s>`, l'étirement d'un graphe géométrique depuis chaque source `s`. On affiche une source atteignant l'étirement maximum, mais aussi une source atteignant l'étirement minimum, ainsi qu'un chemin réalisant ces étirements.

`-check volm`

Calcule la distribution du volume monotone des sommets. Le volume monotone d'un sommet u est le nombre d'arcs du sous-graphe des sommets accessibles depuis u dans le graphe où seuls les arcs u->v avec u<v ont été gardés. Cette mesure dépend de la numérotation des sommets. À utiliser en combinaison avec l'option `-op permute`.

`-check ball <r>`

Calcule la distribution du nombre de sommets des boules fermées de rayon `r` des sommets.

Ex : `gengraph tree 1000 -format no -check ball 3`

`-check ncc`
`-check connected`

Donne le nombre de composantes connexes, leur taille s'il y en a plusieurs, ainsi que le nombre de points d'articulation du graphe. Ces informations sont aussi affichées avec `-check dfs 0`.

`-check dfs <s>`

Effectue un parcours en profondeur d'abord de toutes les composantes connexes du graphe généré depuis le sommet `s`. Complète l'affichage de `-check ncc` en affichant en plus l'arborescence (-1 indique une racine) ainsi que la distribution de profondeur des sommets.

`-check deg`
`-check edge[s]`

Affiche la distribution des degrés et le nombre d'arêtes du graphe.

`-check degenerate`

Donne la dégénérescence du graphe, ainsi que l'ordre d'élimination correspondant des sommets.

`-check gcolor`

Donne une borne supérieure sur le nombre chromatique du graphe en utilisant l'heuristique du degré minimum.

`-check kcolor <k>`

Donne une k-coloration du graphe (et la couleur pour chaque sommet), si c'est possible. Pour cela, une recherche exhaustive de toutes les k-colorations est effectuée. Le temps est raisonnable si k=3 et n<20.

`-check kcolorsat <k>`

Donne une formulation SAT de la k-coloration du graphe. Il s'agit de la formulation multi-valuée classique, un sommet pouvant avoir plusieurs couleurs sans que cela nuise à la validité du résultat. Les contraintes sont décrites au format Dimacs CNF. On peut alors envoyer le résultat à un solveur SAT comme MiniSat ou Glucose. Le graphe n'est pas affiché, et donc `-format no` n'est pas nécessaire.

Ex : `gengraph linial 6 3 -check kcolorsat 3 | glucose -model`

`-check kindepsat <k>`

Donne une formulation SAT d'un ensemble indépendant de taille k du graphe. Les variables i=1 à n indiquent si le sommet numéroté i-1 est dans la solution ou pas. Les contraintes sont décrites au format Dimacs CNF. On peut alors envoyer le résultat à un solveur SAT comme MiniSat ou Glucose. Le graphe n'est pas affiché, et donc `-format no` n'est pas nécessaire.

Pour le problème clique de taille k, il suffit de chercher un ensemble indépendant de taille k pour le complément du graphe. Et pour le problème « vertex cover » de taille k, c'est un ensemble indépendant de taille n-k sur le complémentaire qu'il suffit de chercher.

`-check ps1`
`-check ps1b`
`-check ps1c`
`-check ps1x <n> <u_1> <v_1> … <u_n> <v_n>`

Applique le test ps1 ou l'une de ses variantes (voir `-test ps1` pour plus de détails sur ce test). Affiche aussi le nombre de tests réalisés (nombre de paires de sommets et de chemins testés).

`-check paths <x> <y>`

Liste tous les chemins simples entre les sommets `x` et `y`. N'affiche rien si `x` et `y` ne sont pas connectés. L'ordre est défini suivant le premier plus court chemin dans l'ordre des sommets depuis le sommet `x`.

`-check iso`

Absorbe un graphe H et cherche si le graphe G est isomorphe à H. Si oui, l'isomorphisme de G à H est donné. Le nombre de tests affiché est le nombre de fois où les graphes ont été comparés, la comparaison prenant un temps linéaire en la taille des graphes. Plus les graphes sont symétriques (comme un `cycle` ou un `hypercube`), plus le nombre de tests sera important.

Ex : `gengraph linialc 4 2 linial 4 2 -check iso`

Cet exemple exécute exactement 524 209 tests (ce qui se fait en moins de 0"05), pour un graphe 4-régulier avec 12 sommets. Le nombre de tests monte à 731 767 si on inverse l'ordre des deux graphes.

Tester l'isomorphisme entre deux cycles de 8 sommets étiquetés aléatoirement prend environ 4 000 tests, et entre deux cycles de 12 sommets, 30 millions de tests soit 1" environ. Pour deux arbres à 75 sommets (aléatoires mais isomorphes), moins de 20 tests suffisent. En revanche, le test pour des graphes arêtes et sommets transitifs à 16 sommets, comme `gpetersen 8 3` et `haar 133`, est hors de portée.

`-check sub`
`-check span`

Absorbe un graphe H et cherche si le graphe G est un sous-graphe couvrant de H (donc avec le même nombre de sommets). S'ils ont le même nombre d'arêtes, le test est équivalent à l'isomorphisme (`-check iso`). Le nombre de tests est le nombre total de fois où deux graphes sont comparés. On peut tester si H est hamiltonien en prenant pour G un cycle.

Tester un cycle de longueur 12 dans une grille 3×4 prend jusqu'à environ 32 millions de tests (parfois bien moins), soit au plus 10".

`-check isub`

Absorbe un graphe H et cherche si le graphe G est un sous-graphe induit de H. Contrairement à l'option `-check sub`, G et H n'ont pas forcément le même nombre de sommets.

`-check minor`

Absorbe un graphe H et cherche si le graphe G contient H comme mineur. Les graphes peuvent être non connexes. S'ils ont le même nombre de sommets, le test est équivalent à celui du sous-graphe (`-`

`check sub`). Dans le cas positif, un modèle de H dans G est fourni. L'exemple ci-dessous teste si le graphe H = cube est un mineur de G = $K_{3,7}$. La réponse est non, après 2 729 118 tests en 16" environ.

Ex : `gengraph bipartite 3 7 cube -check minor`

Le principe consiste à contracter des arêtes de G, de toutes les manières possibles, et à tester si H est un sous-graphe du graphe contracté. Le nombre de tests affichés est le nombre de contractions plus le nombre total de tests réalisés par les tests de sous-graphe. Pour H=K_4 il est préférable d'utiliser `-check twdeg` qui donne < 3 ssi le graphe ne contient pas K_4 comme mineur.

`-check twdeg`

Donne une borne supérieure et inférieure sur la largeur arborescente (*treewidth*) du graphe. Pour la borne supérieure, on utilise l'heuristique du sommet de degré minimum que l'on supprime et dont on complète le voisinage par une clique. En cas d'égalité (même degré), on sélectionne le sommet auquel il faut rajouter le moins d'arêtes. La borne inférieure qui est donnée provient de la dégénérescence. La treewidth est exacte si 0, 1 ou 2 est retourné. L'algorithme est en $O(n^2)$.

`-check tw`

Calcule la largeur arborescente (*treewidth*) du graphe en analysant tous les ordres d'éliminations. La complexité est donc en n!. Il ne faut l'utiliser que si le nombre de sommets est < 13.

Ex : `gengraph random 12 .5 -check tw -chrono`

Cet exemple donne 5 en environ 750 millions de tests. Parfois, l'utilisation de `-op permute` peut accélérer le traitement, car partir d'un ordre d'élimination déjà bon permet d'éliminer rapidement beaucoup d'ordres possibles.

`-check clique`

Liste toutes les cliques maximales en utilisant l'algorithme de Bron–Kerbosch.

Une clique est maximale lorsqu'elle n'est pas incluse dans une clique plus grande. `-prop clique` donne directement la taille de la plus grande clique (dite clique maximum).

`-check routing [hash <h>] [scenario [nomem] <s>] <schéma> [<paramètre>]...`

Construit les tables de routage pour le graphe selon le schéma de routage `schéma`, ce schéma pouvant comporter des paramètres spécifiques. La sortie consiste en statistiques sur les tables (taille, temps de calcul) et le graphe. L'option `scenario` permet en plus de tester certains types de routage (`s`) sur le graphe et d'afficher des statistiques sur les longueurs de routes générées (dont l'étirement). L'option

`hash` permet de préciser la fonction de hachage (`h`) appliquée le cas échéant aux sommets souvent utilisés dans les schémas de type « name-independent ». Le graphe doit être connexe et comporter au moins une arête, propriétés qui sont toujours testées.

Ex : `gengraph rplg 200 2.3 -op maincc -op permute -op noloop -check routing scena`

L'option `nomem` après `scenario` permet d'optimiser la mémoire en ne stockant pas les distances calculées pour établir l'étirement (par défaut elles le sont). En contrepartie le temps de calcul est allongé. Cela peut avoir un intérêt si le graphe est très grand (n ≃ 1 000 000 sommets) et qu'un scénario comme `pair -1000` est testé. Les scénarii possibles sont (n=nombre de sommets du graphe) :

- `scenario none` → aucun routage (scénario par défaut)
- `scenario all` → n(n-1) routages possibles
- `scenario edges` → tous les routages entre voisins
- `scenario npairs` → n paires aléatoires de sommets différents
- `scenario one u` → n-1 routages depuis u (choix aléatoire si -1)
- `scenario pair u v` → routage de u à v (choix aléatoire si -1)
- `scenario pair -p` → routage depuis p>0 paires aléatoires
- `scenario until s` → routage jusqu'à l'étirement s ou plus

Les fonctions de hachage h:[0,n[->[0,k[possibles (`shuffle` et `mod` atteignent le nombre de collisions minimum de ⌈n/k⌉) sont :

- `hash prime` → h(x)=((a·x+b)%p)%k où 0<a,b<p sont aléatoires et p=2^31-1 est premier (hachage par défaut)
- `hash mix` → h(x)=mix(a,b,x)%k où a,b sont aléatoires sur 32 bits et mix() est la fonction mélange de Bob Jenkins (2006)
- `hash shuffle` → h(x)=π(x)%k où π(x) est une permutation de [0,n[basée sur deux entiers aléatoires de [0,n[
- `hash mod` → h(x)=(x+a)%k où a est aléatoire dans [0,k[

`-check routing cluster <k>`

Schéma de routage name-independent `cluster` de paramètre `k`. Un sommet de degré maximum est choisi comme « centre », puis k-1 de ses voisins de plus hauts degrés sont choisis pour former un cluster de taille au plus k. Un arbre BFS est enraciné depuis le centre. Chaque sommet possède une boule par rayon croissant qui s'arrête avant de toucher un sommet du cluster. On route de u vers v d'abord dans la boule de u. Sinon on va dans le cluster pour chercher un sommet du cluster responsable du hash de v. Une fois atteint on route selon l'arbre BFS ou selon un plus court chemin si la distance à la destination est ≤ log n / log log n. Les sommets ayant un voisin dans le cluster et qui ne sont pas eux-mêmes dans le cluster possèdent dans leur table tout leur voisinage. Les boules sont optimisées par l'usage d'un voisin par défaut.

Si k=-1, alors k est initialisé à ⌈√n⌉. Si k=-2 il est initialisé à n si bien que le cluster est fixé à tout le voisinage du centre. Dans tous les cas, l'étirement est toujours ≤ 5. Il est même ≤ 3 si k=1. La taille des tables est réduite dans le cas de graphes power-law (comme RPLG).

Il existe plusieurs variantes (`-check variant <v>`) où chacun des bits de `v` mis à 1 est interprété comme suit (pour activer les bits désirés, il faut faire la somme des valeurs entre crochets) :

- bit 0 [1] : le routage est réalisé sans les boules de voisinage, ce qui réduit la taille moyenne mais augmente l'étirement maximum.
- bit 1 [2] : le routage dans le cluster est réalisé dans l'étoile couvrant le cluster, sans les autres arêtes du cluster. Cela réduit la taille des tables sans modifier l'étirement maximum. Si de plus k=1, le routage est alors réalisé via la racine de l'arbre BFS ce qui réduit au minimum la taille moyenne des tables (2 en moyenne).
- bit 2 [4] : les tables des sommets voisins du cluster sont vides. L'étirement devient ≤ 7 au lieu de 5 au maximum, mais la taille des tables est réduite.
- bit 3 [8] : les tables des sommets voisins du cluster ne contiennent que des sommets qui ne sont pas dans le cluster. L'étirement ≤ 5 est préservé, mais généralement la taille maximum des tables est réduite, l'étirement moyen n'augmentant que très légèrement.

`-check routing dcr <k>`

Schéma de routage name-independent `dcr` de paramètre k>0 représentant le nombre de couleurs. C'est une simplification du schéma « agmnt ». L'étirement est toujours ≤ 5 et le nombre d'entrées des tables est en moyenne f(k,n) = 2n/k + (k-1)·(H(k)+1) où H(k) ~ ln(k)+0.577... est le k-ième nombre harmonique. Le principe du schéma est le suivant. Chaque sommet possède une couleur, un entier aléatoire de [0,k[, les sommets landmarks étant ceux de couleur 0. Les boules de voisinage des sommets sont définies par volume comme la plus petite boule contenant au moins chacune des couleurs (ou tous les sommets si une couleur n'est pas représentée), les sommets du dernier niveau étant ordonnés par identifiant croissant. Le routage s→t s'effectue selon un plus court chemin si t est dans la boule de s ou si t est un landmark. Sinon, on route vers le sommet w de la boule de s dont la couleur est égale au hash de t, une valeur aussi dans [0,k[. Puis le routage w→t s'effectue dans l'arbre BFS enraciné dans le plus proche landmark de s ou de t, celui minimisant la distance de w à t.

Si k=-1, alors k est initialisé à sa valeur optimale théorique, celle qui minimise le nombre moyen d'entrées f(k,n), valeur calculée numériquement et qui vaut environ k ≃ √(n/ln(n))/2, ce qui donne environ 2√(n·ln(n·ln(n))) entrées en moyenne. Si k=-2, le nombre de couleurs est initialisé à n. Les valeurs de k>n sont possibles, il s'agit alors d'un routage de plus courts chemins comme pour le cas k=n ou k=1. Il existe une variante (`-check variant 1`) lorsque k>1 qui a pour effet de choisir les landmarks comme les sommets de plus haut degré. Plus précisément, les ⌈n/k⌉ sommets de plus haut degré sont coloriés 0 et les autres coloriés aléatoirement dans [1,k[. Dans cette variante, la borne sur l'étirement est toujours garantie mais plus sur le nombre maximum d'entrées. Cependant, pour certains graphes, l'étirement moyen est amélioré.

`-check routing agmnt <k>`

Schéma de routage name-independent dit « agmnt » du nom de ses auteurs Abraham et al. (2008). C'est la version originale du schéma « dcr » qui diffère par l'algorithme de routage. Comme dans « dcr », le routage s→t s'effectue directement vers t si t est dans la boule de s ou est un landmark. Sinon, vers le sommet w de la boule de s dont la couleur est égale au hash de t. Le routage w→t s'effectue suivant la meilleure des options suivantes : router via un arbre BFS d'un des landmarks ; ou bien router via un arbre couvrant la boule d'un sommet s' contenant à la fois w et un sommet x voisin d'un sommet y contenu dans la boule de t (les boules s' et de t, si elles existent, sont dites « contiguës via l'arête x-y »). L'étirement est toujours ≤ 3 et les tables ont le même nombre d'entrées que « dcr », bien que plus complexes. Le temps de calcul des tables est plus important que pour « dcr ». Toutes les variantes de « dcr » (`-check variant <v>`, k<0) s'appliquent aussi à « agmnt ».

`-check routing tzrplg <t>`

Schéma de routage étiqueté inspiré de celui de Thorup & Zwick (2001) optimisé pour les Random Power-Law Graphs (voir `prlg`) de paramètre réel `t` (power-law exponent) et proposé par Sommer et al. (2012). L'étirement est toujours ≤ 5. Les valeurs de t entre]0,1.5] sont interdites. Le schéma utilise des sommets landmarks où des arbres BFS sont enracinés, ainsi que des boules (de voisinage) définies par rayon croissant qui s'arrête avant de toucher un landmark. Le routage s'effectue alors en priorité via les boules ou alors via le landmark le plus proche de la destination (sans raccourci), information précisée dans l'étiquette de la destination. Les landmarks sont les sommets de plus haut degré. Par défaut, leur nombre vaut :

- si t>1.5, ⌈n^((t-2)/(2t-3))⌉
- si t=0, ⌈√n⌉
- si t<0, |t|

Avec `-check variant 1` et t>1.5, alors les landmarks sont tous les sommets de degré > n^1/(2t-3). Avec `-check variant 2` et t>0, alors les landmarks sont `t` sommets choisis aléatoirement. L'étirement est ≤ 3 si un seul landmark est choisi.

`-check routing hdlbr <k>`

Schéma de routage name-independent HDLBR selon Tang et al. (2013) avec k>0 landmarks qui sont les sommets de plus haut degré. Si k<0, alors k est initialisé à ⌈√n⌉. Chaque sommet qui n'est pas un landmark possède une boule dont le rayon est juste inférieur au plus proche landmark. Chaque sommet possède sa boule inverse (qui peut être vide), définie comme l'ensemble des sommets le contenant dans leur boule. Chaque landmark a une couleur unique. On route directement de u à v si v est dans la boule de u, la boule inverse de u, ou si v est un landmark. Sinon, on route vers le landmark (selon un plus court chemin) dont la couleur vaut le hash de v. De là, on route suivant un plus court chemin vers le plus proche landmark de v, l(v). On utilise ensuite le next-hop de l(v) vers v, un sommet nécessairement contenant v dans sa boule inverse. À chaque étape du routage, si v est dans la boule ou boule inverse du sommet courant, on route directement vers celui-ci. La longueur de route entre u et v

est au plus 2d(u,v)+2r, où r est la distance maximum entre deux landmarks. La valeur de r est bornée par une constante pour k ≃ √n et pour les Random Power-Law Graphs (voir `rplg`).

`-check routing bc <k>`

Schéma de routage étiqueté selon Brady-Cowen (2006). L'étirement est additif ≤ 2k, et donc multiplicativement ≤ 2k+1. En particulier, si k=0, il s'agit d'un routage de plus court chemin. Il est adapté aux Power-Law Random Graphs (voir plrg). Le principe est le suivant : on construit un arbre BFS (=T) enraciné dans un sommet (=r) de plus haut degré. Le cœur (=C) est la boule de rayon k depuis r. On construit une liste (=L) de BFS couvrant G ainsi qu'une forêt (=H) de BFS de G comme suit. Au départ, L={T}, et H est la forêt T\C. Puis, pour chaque arête {u,v} de G\C\T, on vérifie si l'ajout de {u,v} à H crée un cycle ou pas. Si c'est non, on met à jour la forêt H en lui ajoutant {u,v}. Si c'est oui, on calcule un BFS de G de racine u (ou v) qu'on ajoute à L. Une fois le calcul de H terminé, on calcule une forêt BFS couvrante de H qu'on ajoute à L. L'algorithme de routage de u à v consiste simplement à router dans l'arbre de L qui minimise la distance de u à v.

RÉGLAGES GLOBAUX

Ces options ne touchent pas à la pile de requêtes. La plupart servent à paramétrer le comportement des options s'appliquant aux requêtes.

`-main-seed <s>`

Réinitialiser le générateur d'aléatoire principal avec la graine `s`, qui doit être un nombre entier positif sur 64 bits. Cela donne la possibilité de retrouver une suite aléatoire.

Au lancement, GenGraph initialise son générateur avec une graine aléatoire sur 16 bits produite par la technologie `arc4random`. Cette graine peut être affichée avec `-print-seeds`. Voir aussi `-seed-bits` pour générer une graine aléatoire sur davantage de bits.

Pour la graine du générateur utilisé pour le contenu des graphes, voir `-seed`.

`-seed-bits <b>`

Configure le nombre de bits à utiliser pour les graines des générateurs de nombres aléatoires. `b` peut valoir `-` pour le maximum (c'est-à-dire 64). Dès que cette option est utilisée, le générateur d'aléatoire principal est réinitialisé avec une nouvelle graine produite par `arc4random`.

Par défaut, les graines sont sur 16 bits seulement (<65536) afin d'être plus faciles à manipuler.

`-width <m>`

Limite à `m` le nombre d'arêtes et de sommets isolés affichés par ligne. Cette option n'a d'effet que sur les formats simple, DOT et TikZ. Par exemple, `-width 1` affiche une arête (ou un sommet isolé) par ligne. L'option `-width 0` affiche tout sur une seule ligne. La valeur par défaut est 12.

`-header`
`-noheader`

Active ou désactive l'affichage d'un préambule joint au graphe et aux résultats de `-check`. Ce préambule donne certaines informations sur le graphe, sous forme de commentaire à la C++ (`//`). Par défaut aucun préambule n'est affiché. Les informations affichées sont :

- l'heure, la date et la graine du générateur d'aléatoire ;
- la ligne de commande qui a produit la génération du graphe ;
- le nombre de sommets, d'arêtes, les degrés maximum et minimum.

Avec la plupart des formats, le nombre d'arêtes (et les degrés min et max) n'est pas déterminé avant l'affichage du graphe. Pour cette raison, ces nombres ne sont affichés qu'après le graphe.

Pour n'avoir que le préambule, utiliser `-header` avec l'option `-format no`. Voir aussi `-check info`.

`-caption <titre>`

Permet d'ajouter une légende à un graphe avec les formats graphiques. Il est possible d'afficher la graine principale de l'aléatoire avec le code `%SEED%` (voir `-main-seed`).

Ex :
```
gengraph gabriel 30 -caption ex1 -visu
gengraph gabriel 30 -caption "Exemple 2" -visu
gengraph gabriel 30 -caption "graph with seed=%SEED%" -visu
```

`-label [depth <d>] <b>`

Active (`b`≠0) ou désactive (`b`=0, valeur par défaut) l'affichage du nom des sommets. Dans les formats textuels, les numéros de sommet sont remplacés par le nom tel que décrit ci-dessous.

Les valeurs possibles de l'entier `b` sont dans [-3,3]. Le signe de `b` ne détermine que la position des étiquettes dans les formats graphiques : au centre si `b`>0, à côté du sommet sinon.

- Si |`b`|=1, les noms seront déterminés par le graphe. Par exemple, pour un `hypercube`, ce sont des mots binaires. Si le graphe ne donne pas de nom à ses sommets, l'effet est identique à |`b`|=2.
- Si |`b`|=2, les noms seront sous forme d'entiers de [0,n[. Avec les formats textuels, c'est identique à `b`=0.

- Si |`b`|=3, les noms seront les coordonnées des points lorsqu'elles existent (voir `-xy`) ; sinon, l'effet est identique à |`b`|=2.

Ex :
```
gengraph petersen -label 1 -width 1
gengraph petersen -label 1 -format dot | grep label
gengraph petersen -label 1 -dot len 2 -visu
gengraph gabriel 30 -view no pos -label 1 -visu
gengraph gabriel 30 -label -3 -view scale 4 -xy round 2 -visu
```

Le paramètre `d` peut devenir utile lors de l'usage d'opérations : il détermine à quelle profondeur s'applique le type de nom choisi. Cette notion vient du fait que les opérations demandent généralement à leurs graphes opérandes les noms de leurs sommets pour générer leurs propres noms. Si |`b`|=1, cela implique une profondeur infinie. Si |`b`|=2 ou 3, alors par défaut `d`=1.

Ex :
```
gengraph mesh 5 2 path 5 -op union-s 5 0 9 4 . -label depth 2 2 -visu
gengraph hypercube 3 -op star 3 -label depth 2 -2 -visu
```

`-prop <propriété>`

Définit la propriété utilisée pour les options `-print-prop`, `-sort` et `-test prop`. À l'heure actuelle, ces propriétés sont toutes des nombres entiers.

Certaines propriétés requièrent que le graphe soit chargé en mémoire pour être calculées, auquel cas le stockage du graphe est implicitement déclenché, afin qu'il fonctionne comme un graphe `load`.

`-prop id`

Identifiant du graphe (voir `-id`).

Ex :
```
gengraph tutte -id 42 -prop id -print-prop -discard
```

Cet exemple affiche 42.

`-prop stack`

Position du graphe dans la pile (à partir de 1). Pour le graphe au sommet de la pile, il s'agit aussi de la taille de la pile. C'est la propriété par défaut.

`-prop groups`

Nombre de groupes dans la pile.

`-prop group-size`

Taille du groupe au sommet de la pile. Ce groupe peut être de taille 0, sinon c'est celui du graphe au sommet de la pile.

`-prop order`

Ordre du graphe, c'est-à-dire son nombre n de sommets.

`-prop size`

Taille du graphe, c'est-à-dire son nombre m d'arêtes.

`-prop cc`

Nombre de composantes connexes du graphe. Voir aussi `-check ncc`.

`-prop deg[min|max]`

Degré minimum ou maximum du graphe. Voir aussi `-check deg`.

`-prop radius`

Rayon du graphe, ou -1 si le graphe n'est pas connexe. Le rayon est la profondeur du plus petit arbre couvrant le graphe, c'est-à-dire la plus petite distance à laquelle se trouve un sommet de tous les autres.

`-prop diameter`

Diamètre du graphe, ou -1 si le graphe n'est pas connexe. Le diamètre est la profondeur du plus grand arbre couvrant le graphe obtenu par un parcours en largeur, c'est-à-dire la plus grande distance entre deux de ses sommets.

`-prop degenerate`

Dégénérécence du graphe. Voir aussi `-check degenerate`.

`-prop gcolor`

Nombre de couleurs obtenu selon l'heuristique du degré minimum. Voir aussi `-check gcolor`.

`-prop girth`

Maille du graphe, c'est-à-dire taille de son plus petit cycle, ou -1 si le graphe n'a pas de cycle. Elle n'est définie que pour les graphes orientés.

`-prop cut-vertex`

Nombre de points d'articulation du graphe. Un sommet est un point d'articulation si sa suppression augmente le nombre de composantes connexes.

`-prop tw`

Largeur arborescente (*treewidth*) du graphe.

Son calcul, lorsqu'il a lieu, est assez lent. Pour les petites valeurs de `tw`, des alternatives sont possibles (voir `-check tw` et `-test tw2`). Pour savoir si un graphe G a une largeur arborescente de 3, il suffit de vérifier si G contient l'un des 4 mineurs suivants :

```
gengraph clique 5 wagner prism 5 \
  -group hajos cycle 3 -op union \
  load G -test is-minor -filter
```

`-prop hyper`

Hyperbolicité du graphe. Il s'agit de la valeur (entière) maximum, sur tous les quadruplets de sommets {u,v,x,y}, de la différence des deux plus grandes sommes parmi les sommes de distances : uv+xy, ux+vy et uy+vx. La complexité est en $O(n^4)$.

`-prop clique`

Ordre de la plus grande clique du graphe. Ce nombre est trouvé par force brute. Le délai pour le trouver est raisonnable pour les graphes de moins de 100 sommets.

Voir aussi `-check clique`.

`-prop prop`

Propriété actuelle, définie avec `-prop`. Une commande `-prop prop` est sans effet ; mais la commande `-test prop` permet d'utiliser la propriété actuelle.

`-test <test> [<paramètre>]...`

Définit le test à utiliser pour les options `-filter`, `-extract`, `-extract-all`, `-while` et `-print-test`. Chaque test est une fonction booléenne prenant en paramètre une requête de graphe. Il peut être inversé avec `-test not`.

La plupart des tests requièrent que le graphe soit préalablement stocké. Ainsi, si un graphe à tester n'est pas encore stocké, alors il est stocké comme si on lui avait appliqué l'option `-store`.

Certains tests absorbent le graphe ou le groupe de graphes se trouvant au sommet de la pile afin de l'utiliser comme paramètre. La description du test parle alors d'un « graphe absorbé » ou « groupe absorbé ». Chaque graphe absorbé est immédiatement stocké.

Certains tests demandent en paramètre un sélecteur de nombres entiers, notamment `-test prop` et `-test <propriété>`. Par exemple, `-test id 5-8 -filter` conduit à ne garder dans la pile que les graphes dont l'identifiant se situe entre 5 et 8. De manière générale, un sélecteur est soit `t` (pour « true ») pour accepter toutes les valeurs, soit une suite de codes séparés par des `,` (interprétées comme « ou »), respectant l'un de ces formats (avec `x` et `y` entiers) :

- `x` ou `=x` → égal à x
- `<x` → inférieur à x
- `>x` → supérieur à x
- `<=x` → inférieur ou égal à x
- `>=x` → supérieur ou égal à x
- `x-y` → dans l'intervalle [x, y]
- `%x` → divisible par x
- `%x=y` → congru à y modulo x

Ex :
```
gengraph … -test order '5,7-13,>100' -filter
gengraph … -test order 5-10 -filter @1 \
  -prop order -print-prop -discard -test true -while 1
gengraph … -test cc %2=1 -filter
```

Le premier exemple conserve les graphes ayant un ordre n vérifiant soit n=5, soit 7 ≤ n ≤ 13, soit n>100. Le deuxième exemple affiche le nombre de sommets des graphes ayant entre 5 et 10 sommets. Le dernier exemple ne conserve que les graphes ayant un nombre impair de composantes connexes. Notez que les symboles `<` et `>` doivent être échappés (par exemple avec des guillemets comme dans `'>14'`) avec les interpréteurs de commandes les plus courants.

Il est possible de combiner plusieurs tests en notation polonaise inversée, à l'aide de `-test and` par exemple. À cette fin, une (petite) pile de tests est maintenue, ses éléments étant libérés lorsqu'elle déborde.

`-test true`

Toujours vrai (test par défaut). Avec `-while`, cela sert aussi à faire le saut tant qu'il reste des graphes dans la pile.

`-test not`

Contraire du test précédemment défini.

`-test and`

Si les deux tests précédemment définis sont vrais.

`-test or`

Si au moins l'un des deux tests précédemment définis est vrai.

`-test xor`

Si exactement un des deux tests précédemment définis est vrai.

`-test random <p>`

Vrai avec une probabilité `p` (entre 0 et 1).

`-test prop <sélecteur>`
`-test <propriété> <sélecteur>`

Présence, dans `sélecteur`, de la valeur de la propriété.

`-test prop` utilise la propriété qui a été définie précédemment avec `-prop`, en vigueur au moment de la mise en place du test. La deuxième variante permet de tester directement n'importe quelle propriété acceptée également par `-prop`.

Ex :
```
gengraph clique 5 -prop size -print-prop -test prop =10 -print-test -discar
```

Cet exemple affiche 10, puis oui.

`-test same-id`

Égalité entre l'identifiant du graphe et celui d'un graphe du groupe absorbé. Les graphes absorbés ne sont pas stockés ; seul leur identifiant est conservé pour les tests.

Ex :
```
gengraph load+ F1 load+ F2 -test same-id -test not -filter
```

Cet exemple charge les graphes du fichier `F1` et supprime ceux dont l'identifiant se trouve également dans `F2`.

`-test iso`

Existence d'un isomorphisme au graphe absorbé. Ce test est très lent, voir `-check iso`.

Ex : `gengraph cycle 3 complete 3 cube triangle -test iso -test not -filter`

Dans cet exemple, le graphe `triangle` est absorbé par le test. N'est laissé sur la pile que le graphe `cube` car c'est le seul qui n'est pas isomorphe au triangle.

`-test unique`

Existence d'un isomorphisme à un des graphes précédents dans le groupe de graphes. Ce test est très lent, voir `-check iso`.

`-test has-minor`
`-test is-minor`

Présence du graphe absorbé comme mineur du graphe (`has-minor`), ou au contraire, présence du graphe comme mineur du graphe absorbé (`-is-minor`).

Si le graphe absorbé est `clique 4`, il est préférable d'utiliser `-test tw2`.

Voir `-prop tw` pour un exemple.

`-test has-sub`
`-test is-sub`

Si le graphe a le même nombre de sommets que le graphe absorbé, présence de ce dernier comme sous-graphe couvrant du graphe (`has-sub`), ou au contraire, comme sur-graphe couvert (`is-sub`) par le graphe. Un graphe G est couvert par un sous-graphe H s'ils ont le même nombre de sommets et si chaque arête de H se retrouve dans G.

`-test has-isub`
`-test is-isub`

Présence du graphe absorbé comme sous-graphe induit du graphe (`has-isub`), ou au contraire, présence du graphe comme sous-graphe induit du graphe absorbé (`is-isub`).

`-test deg <sélecteur>`

Si tous les degrés du graphe sont compris dans l'ensemble déterminé par `sélecteur`. Ainsi, `-test deg 4-7` est vrai pour tous les graphes avec un degré minimum d'au moins 4 et un degré maximum d'au plus 7.

`-test forest <sélecteur>`

Si le graphe est une forêt dont le nombre d'arbres est dans l'ensemble déterminé par `sélecteur`.

`-test is-forest (= -test forest t)`

Si le graphe est une forêt.

`-test is-tree (= -test forest 1)`

Si le graphe est un arbre.

`-test cycle (= -test forest t -test not)`

Présence d'un cycle dans le graphe.

`-test bipartite (= -test gcolor <3)`

Si le graphe est biparti.

`-test connected (= -test cc 1)`

Connexité du graphe.

`-test biconnected`

2-connexité du graphe. Un graphe G est k-connexe s'il n'y a pas d'ensemble avec <k sommets qui déconnecte G ou laisse G avec 1 sommet. Un graphe est 2-connexe s'il est connexe, ne possède pas de sommet d'articulation et a plus de 2 sommets. Les cliques de taille k+1 sont k-connexes.

`-test ps1`
`-test ps1b`
`-test ps1c`
`-test ps1x <n> <u_1> <v_1> … <u_n> <v_n>`

Test `ps1`, correspondant à la fonction f(G,{}) décrite ci-après.

Soit P un chemin d'un graphe G tel que G\P est connexe. La fonction f(G,P) est vraie ssi G est vide (en pratique |G|-|P|<3 suffit) ou s'il existe deux sommets x,y de G où y n'est pas dans P tels que pour tout chemin Q entre x et y dans G « compatible » avec P (c'est-à-dire P et Q s'intersectent en exactement un segment) on a les deux conditions suivantes : (1) il n'y a pas d'arête entre les sommets de P\Q et de G\(QuP) ; et (2) pour toute composante connexe C de G\(QuP), f(CuQ,Q) est vraie. Le test est optimisé dans un certain nombre de cas, en particulier : les arbres (toujours vrai), les cliques (vrai ssi n<5).

La variante `ps1b` calcule et affiche de plus un graphe des conflits (affichage modifiable par `-width`), chaque nœud de ce graphe correspondant à un argument (CuQ,Q) évalué à faux par f. La valeur (ou code) d'un nœud est `0` (=lourd ou faux), `1` (=léger ou vrai) ou `-` (indéterminé). Suivant certaines règles,

les valeurs 0 ou 1 sont propagées selon le type des arêtes du graphe des conflits. Résoudre le graphe des conflits revient à trouver une affectation des valeurs 0 ou 1 aux nœuds qui respecte (sans contradiction) toutes les règles.

La fonction f(G,{}) est évaluée à vrai si le graphe des conflits n'a pas de solution, c'est-à-dire si une contradiction a été découverte ou si pour une paire de sommets (x,y) tous ses nœuds sont à 1.

On affiche le code d'un nœud (`0`,`1`,`-`) ainsi que les sommets de sa composante (par ex : `[237]`). Les nœuds du graphe des conflits sont reliés par des arêtes typées. Les voisins v d'un nœud u sont listés avec le type de l'arête, si l'un des 4 cas suivants se produit (il n'y a pas d'arête entre u et v dans les autres cas) :

- `v<` → la composante de v est incluse dans celle de u
- `v>` → la composante de v contient celle de u
- `v=` → les composantes de u et v sont les mêmes
- `v|` → les composantes de u et v sont disjointes

Parmi les règles, on trouve par exemple : si deux nœuds du graphe des conflits u=(CuQ,Q) et v=(C'uQ',Q') sont disjoints, c'est-à-dire C n'intersecte pas C', alors seule une des expressions f(CuQ,Q) ou f(C'uQ',Q') peut être fausse, pas les deux. Dit autrement, les composantes de u et v ne peuvent pas être « lourdes » (=0) toutes les deux en même temps. Et donc, si le code de u est 0, celui de v est 1. Notons que le code de u et v égal à 1 est compatible avec cette règle.

La variante `ps1c` est similaire à `ps1b` sauf que récursivement seul le test `ps1` est appliqué, et pas `ps1b`. Le test `ps1c` est plus long que `ps1` mais plus rapide que `ps1b`. La variante `ps1x` est similaire à `ps1b` sauf que les valeurs v_i sont écrites dans le nœud u_i du graphe des conflits principal (pas ceux générés lors des appels récursifs). Plus précisément, v_1 (0 ou 1) est écrite dans le nœud u_1, puis cette valeur est propagée. Ensuite, v_2 est écrite puis propagée, etc.

Dans tous les cas, si G n'est pas connexe, le résultat n'est pas déterminé.

`-test tw2`

Si la largeur arborescente du graphe est ≤ 2. L'algorithme est en $O(n^2)$. Ce test peut être utilisé pour relever (plus rapidement qu'avec `-test has-minor`) les graphes sans mineur K_4. Voir aussi `-prop tw` et `-check tw`.

`-format <type>`

Spécifie le format de sortie. Les valeurs possibles pour `type` sont :

- `auto` (par défaut) : essaie de déterminer le type en fonction de l'extension de nom de fichier ;
- `simple`, `standard`, `default`, `classic` ou `edges` : format par défaut (voir la section LE FORMAT PAR DÉFAUT), c'est en principe le plus compact ;
- `list` : liste d'adjacence ;

- `matrix` : matrice d'adjacence ;
- `smatrix` : matrice supérieure, diagonale comprise ;
- `vertex <i>` : liste des voisins du sommet `i` ;
- `dot` : format de Graphviz, proche du format simple ;
- `dot-<type>` : format `type` obtenu par génération du DOT puis conversion avec le programme `dot` de Graphviz ;
- `html` : document HTML dynamique utilisant vis.js (cf. https://visjs.org/) ;
- `tikz` : document LaTeX utilisant le paquetage TikZ (requiert un graphe géométrique, voir `-xy`) ;
- `xy` : positions X,Y qui ont été utilisées pour le graphe géométrique ;
- `no` : ne rien afficher, à utiliser en combinaison avec `-header`.

Le choix `auto` conduit à utiliser le format simple pour la sortie standard et les fichiers sans extension. Si une extension est définie (avec par exemple `-output` ou `-visu`), le mappage est le suivant (en ignorant la casse) :

- `.txt`, `.edges` ou `.gg` : format simple ;
- `.list`, `.mat` ou `.smat` : format `list`, `matrix` ou `smatrix` ;
- `.xy` ou `.pos` : format `xy` ;
- `.dot` ou `.gv` : format `dot` ;
- `.htm` ou `.html` : format `html` ;
- `.tex` : format `tikz` ;
- autre : format `dot-<type>`.

Le format HTML requiert le fichier `vis-network.min.js`. Il est cherché dans le répertoire du document ainsi que dans un dépôt officiel sur Internet. Lorsque les sommets ont des positions, le rendu est fixe et similaire à celui obtenu avec DOT. Sinon, les positions sont générées par la bibliothèque, et l'initialisation du document dans le navigateur peut commencer à devenir longue dès qu'il y a plus d'une centaine de sommets.

À propos du format DOT :

- Les options `-dot` permettent de paramétrer la génération faite avec ce format.
- Les positions affichées dans le format DOT (`[pos="…"]`) diffèrent d'un facteur proportionnel à √n par rapport aux positions originales du graphe (qui peuvent être affichées par `-format xy` ou `-label -3`). Ce facteur permet de garder une taille raisonnable pour les sommets car sous DOT les sommets ont une taille fixe minimale.
- Le choix d'un format `dot-<type>` conduit GenGraph à faire un tube vers `dot -T <type> -K <filtre>`. Les valeurs les plus utilisées de `type` sont : `pdf`, `ps`, `fig`, `svg` (formats vectoriels, fortement recommandés) ; et `jpg`, `gif` et `png` (formats matriciels, déconseillés). Utiliser `dot -T.` (ou `man dot`) pour voir tous les formats.

`-xy-as-default`

Sauvegarde les réglages `-xy` du graphe au sommet de la pile afin de les appliquer automatiquement à toute nouvelle requête. Les paramètres prédéfinis de certains graphes restent cependant prioritaires. `-xy none` permet toujours de revenir à un graphe non géométrique.

`-vcolor <mode> [<paramètre>]...`

Ces options permettent de modifier la couleur des sommets. Elles n'ont d'effet qu'avec les formats graphiques : DOT et ses dérivés ainsi que HTML (voir `-format`).

Les attributs par défaut des sommets (couleurs, formes, etc.) peuvent également être modifiés en fournissant des options à Graphviz (voir l'option `-N` de `dot`). Cependant, l'option `-vcolor` permet d'individualiser la couleur d'un sommet, en fonction de son degré par exemple. Ici, le degré est le degré non orienté.

`-vcolor none`

Met fin à l'effet des options `-vcolor` précédentes.

`-vcolor deg[r]`

La couleur dépend du degré du sommet (`deg`) ou du rang du degré du sommet (`degr`). Ainsi, les sommets de plus petit degré obtiennent la première couleur de la palette, les sommets de plus grand degré la dernière couleur de la palette, et les autres sommets une couleur intermédiaire de la palette. Donc une seule couleur est utilisée si le graphe est régulier.

`-vcolor degm`

Effectue une coloration propre (deux sommets voisins ont des couleurs différentes) suivant l'heuristique du degré minimum : récursivement, le sommet de degré minimum obtient la plus petite couleur qui n'est pas utilisée par ses voisins. Cela donne des colorations avec assez peu de couleurs pour les graphes de faible arboricité (`planar`, `tw`, `pw`, `kout`, `expander`) ou de faible degré. Avec cette technique, les graphes bipartis (`tree`, `crown`, `half-graph`) sont coloriés avec deux couleurs. Cette option nécessite un espace et un temps en O(n+m).

`-vcolor randg`

Effectue une coloration propre en utilisant un algorithme glouton sur un ordre aléatoire des sommets : récursivement, le sommet d'indice i obtient la plus petite couleur qui n'est pas utilisée par ses voisins d'indice j<i. Cette option nécessite un espace et un temps en O(n+m).

`-vcolor kcolor <k>`

Effectue une k-coloration propre du graphe, si c'est possible. Si cela n'est pas possible, la première couleur est appliquée à tous les sommets. L'algorithme (exponentiel) est le même que celui utilisé par `-check kcolor`.

`-vcolor pal <grad>`

Permet de fixer la palette de couleurs utilisée par les sommets. Le paramètre `grad` est un mot sur l'alphabet [a-z]. Les caractères en dehors de cet alphabet sont ignorés. Chaque lettre correspond à une couleur de base :

```
a=aquamarine     h=hotpink    o=olive      v=violet
b=blue           i=indigo     p=purple     w=white
c=cyan           j=orange     q=pink       x=gray
d=darkorange     k=khaki      r=red        y=yellow
e=chocolate      l=lavender   s=salmon     z=black
f=forestgreen    m=magenta    t=teal
g=green (lime)   n=navy       u=yellowgreen
```

La palette est calculée selon une interpolation linéaire entre les points définis par le mot `grad`. Par exemple, si `grad` vaut `rb`, la palette sera composée d'un dégradé allant du rouge (`r`) au bleu (`b`). Si `grad` vaut `rgbr`, le dégradé ira du rouge au vert puis au bleu et enfin au rouge. Pour avoir une couleur (de base) unique, disons `w`, sur tous les sommets, poser `grad`=`w`. Par exemple, pour avoir tous les sommets blancs, on peut faire :

Ex : `gengraph gabriel 30 -vcolor deg -vcolor pal w -visu`

La palette par défaut correspond au mot `grad` suivant : `redjykugfocatbhsqvmpinzxlw`. On peut visualiser la palette avec l'option `-vcolor list`.

`-vcolor list`

Produit l'affichage de la palette des couleurs utilisées pour un graphe plutôt que le graphe lui-même. Cela permet en particulier de savoir combien de couleurs ont été utilisées. À défaut d'une option `-vcolor` indiquant comment colorer les sommets, `-vcolor deg` est implicite.

Cette fonctionnalité n'est prise en charge qu'avec le format DOT. La palette est générée en affichant au format DOT un graphe particulier où les sommets (représentés par un rectangle) sont les couleurs utilisées. Le nom des sommets correspond à la lettre de la couleur de base (voir `-vcolor pal`).

Ex1 : `gengraph gabriel 50 -vcolor degm -vcolor list -format dot`

(génère la palette utilisée pour ce graphe de Gabriel)

Ex2 : `gengraph prime 53 -vcolor list -format dot`

(un moyen simple de générer la palette par défaut)

Ex3 : `gengraph clique 100 -vcolor degm -vcolor pal rb -vcolor list -format dot`

(génère un dégradé de 100 couleurs allant du rouge au bleu)

Réutiliser l'option une deuxième fois permet de revenir en mode normal.

`-vsize`

Cette option permet d'individualiser la taille des sommets selon un paramètre, alors que par défaut, tous les sommets ont la même taille. Cela n'a d'effet qu'avec les formats graphiques : DOT et ses dérivés, HTML et TikZ. `-vsize` peut être combiné avec `-vcolor`.

Aux formats DOT et TikZ, lorsqu'il y a une étiquette dans le sommet (mise en place par `-label`), elle impose à celui-ci une taille minimale. On doit alors généralement utiliser `-view vsize` pour voir les différences de taille mises en place par `-vsize`.

`-vsize none`

Met fin à l'effet des options `-vsize` précédentes.

`-vsize deg`

Rend la taille des sommets proportionnelle à leur degré.

`-view <option> [<paramètre>]...`
`-view no <option>`

Ces options permettent de paramétrer le rendu graphique. La plupart ne fonctionnent que lorsque les sommets sont positionnés par GenGraph, c'est-à-dire lorsque le graphe est géométrique. La variante `-view no` permet de désactiver une option, donc soit de revenir à la situation par défaut, soit de désactiver une option qui est par défaut activée.

`-view vsize <f>`

Définit un facteur de grossissement des sommets. Par défaut, f=1.

`-view scale <s>|<x>,<y>|auto`

Spécifie le facteur d'échelle pour les formats DOT et TikZ. Cela permet d'écarter ou rapprocher les sommets les uns des autres. L'argument est soit un nombre `s`, soit deux nombres `x,y`, pour un facteur d'échelle identique ou pas en X et Y. Une valeur négative permet de changer le sens du rendu. Par défaut, aucune échelle n'est appliquée (`s`=1). On peut aussi mettre `auto` qui calcule automatiquement un facteur d'échelle (symétrique en X et Y) qui vaut :

- au format DOT : 1/√n, ou 1/max(ΔX,ΔY) dans le cas d'un graphe géométrique ;
- au format TikZ : √n×(ΔX+ΔY).

Ex : `gengraph gabriel 10 -label -3 -view scale 3,2 -visu`

`-view pos`

Prend en compte les positions des sommets si elles sont disponibles. Cela est activé par défaut. Lorsque c'est désactivé (`-view no pos`), tout graphe est rendu comme s'il n'était pas géométrique.

`-view grid <p>`

Ajoute une grille p × p au graphe généré s'il est géométrique, et au format DOT seulement. C'est particulièrement utile lorsque les coordonnées des points sont entiers. Techniquement, on ajoute à la sortie DOT un sous-graphe représentant la grille où les sommets et les arêtes sont de couleur grise. Si p<0, alors le paramètre est initialisé à 1+⌊√n⌋ ou bien à n si l'option `-xy permutation` est présente, n étant le nombre de sommets du graphe. Pour être visible, la grille générée doit comporter au moins 2 lignes et 2 colonnes.

`-view zero`

Ajoute l'origine (0, 0) au dessin qui est représentée par un cercle rouge. Ne fonctionne que si le graphe est géométrique, et ne s'applique qu'au format DOT.

`-view border`

Ajoute une bordure bleue mettant en valeur la boîte englobante du graphe. Ne fonctionne que s'il est géométrique, et ne s'applique qu'au format DOT.

`-dot <option> [<paramètre>]...`

Cette option permet de contrôler la sortie au format DOT. Elle permet par exemple de modifier le filtre, la longueur des arêtes ou l'échelle du dessin.

`-dot len <p>`

Spécifie la longueur des arêtes pour le format DOT et le filtre `neato`. La valeur par défaut est 1, et une valeur plus grande (comme 2.5 ou 3) allonge les arêtes et permet dans certains cas de mieux visualiser le graphe. C'est parfois nécessaire pour éviter l'intersection des sommets lorsqu'on utilise `-label 1`. On peut obtenir le même genre d'effet avec `-view scale`.

`-dot dir <type>`

Spécifie le type d'orientation et l'affichage d'une arête pour le format DOT. Cela revient à mettre l'attribut `dir=type` pour les arêtes. L'effet est de modifier le rendu d'un arc `A->B`, voire d'une arête `A--B`, présent dans le code DOT. `type` doit être l'un des mots clés suivants :

- `back` : l'arc inverse sera affiché, comme si on avait mis `B->A` (note : en DOT, `A<-B` n'est pas autorisé). C'est très utile en combinaison avec l'option `-directed`. Voir aussi le graphe `-op transpose`.
- `both` : `A<->B` sera affiché.
- `none` : `A--B` sera affiché (cas non orienté par défaut).
- `forward` : `A->B` sera affiché (cas orienté par défaut).

L'exemple ci-après affiche un arbre avec une orientation vers ses fils (au lieu de son père comme par défaut).

Ex : `gengraph tree 15 -label 1 -directed -dot dir back -visu`

`-dot filter <f>`

Spécifie le filtre de Graphviz, c'est-à-dire l'algorithme de dessin utilisé par `dot`. Par défaut, le filtre est `neato`. Les filtres principaux sont : `dot`, `neato`, `twopi`, `circo`, `fdp` et `sfdp`. La commande `dot -K .` permet d'afficher les filtres disponibles.

`-dot attr <s>`

Ajoute un ou plusieurs attributs lors de la génération du graphe au format DOT. La chaîne de caractères `s` est simplement ajoutée au corps du graphe, avant la description des arêtes et des sommets. On peut mettre dans `s` autant d'attributs que l'on veut. Chaque option `-dot attr` écrase la valeur précédente.

Attention : l'ajout d'attributs d'arêtes peut provoquer la création de multi-arêtes.

Ex : `gengraph tree 20 -dot attr '0 [color=red]; 0--1 [color=blue];' -visu`

CONTRÔLE DU FLUX DE COMMANDES

Afin de recommencer l'exécution des commandes à partir d'une certaine position, il est possible de placer des étiquettes dans la liste des commandes. Elles doivent être préfixées avec le symbole `@`, et ne peuvent être que des (petits) entiers strictement positifs. Pour donner une étiquette en paramètre à une commande, il faut donner son numéro (sans le symbole `@`). La même étiquette peut être redéfinie plusieurs fois. Il y a deux étiquettes spéciales, prédéfinies :

- `0` : le début de la liste des commandes ;
- `-` : la commande précédente.

`-while <étiquette>`

Relance l'exécution à partir de l'étiquette fournie tant que le test défini avec `-test` est vrai pour le graphe au sommet de la pile.

Si la pile est vide, le test n'est pas exécuté et est considéré comme faux. `-test true -while étiquette` permet donc de répéter des traitements tant qu'il reste des graphes dans la pile.

`-n-times <étiquette> <n>`

Répète n-1 fois l'exécution à partir de l'étiquette indiquée, de sorte que l'exécution soit faite n fois au total. Si n<2, cette commande n'a pas d'effet.

Ex : `gengraph cycle 10 -n-times - 10 -op union-d -visu`

`-quit`

Termine brutalement le programme, sans traiter les commandes suivantes et en ignorant ce qui se trouve encore dans la pile.

En fin de liste de commandes, cela peut servir à éviter de générer et afficher les graphes encore dans la pile. `-quit` peut également être utile au milieu d'une longue liste de commandes, afin de l'étudier ou de la déboguer.

ACCESSOIRES

`-print <texte>`

Affiche `texte` sur la sortie standard, et revient à la ligne. Le texte est formaté comme le contenu du manuel : les codes de formatage sont transformés en codes ANSI et les lignes sont coupées pour ne pas déborder dans le terminal.

Ex : `gengraph -print 'Voici un cycle à 10 sommets (``cycle 10``).' cycle 10 -visu`

`-pause`

Saute une ligne de texte de l'entrée standard. Cela permet de faire des scripts interactifs.

`-chrono`

Affiche la valeur du chronomètre interne, indiquant le temps écoulé depuis le lancement de l'application ou la dernière commande `-chrono-reset`.

`-chrono-reset`

Remet à zéro le chronomètre interne, dont la valeur peut être affichée avec l'option `-chrono`.

`-print-seeds`

Affiche la graine du générateur d'aléatoire principal, ainsi que celle de la requête au sommet de la pile s'il y en a une.

GRAPHES

Principalement deux séries de classes de graphes sont prises en charge : GRAPHES DE BASE et GRAPHES COMPOSÉS. Ces derniers sont obtenus en paramétrant automatiquement un graphe de base ou orienté.

Une catégorie importante (transversale au découpage de cette section) est celle des graphes géométriques, dans lesquels l'adjacence est déterminée par les coordonnées associées aux sommets, par défaut générées comme avec `-xy unif`.

Les GRAPHES ORIENTÉS quant à eux activent tous l'option `-directed`.

GRAPHES DE BASE

`grid n_1 … n_k .`

Grille à k dimensions de taille n_1 × ⋯ × n_k. L'ensemble des sommets correspond à [0,n_1[× ⋯ × [0,n_k[. Si la taille n_i est négative, alors cette dimension est cyclique. Par exemple, `grid -10 .` donne-

ra un cycle à 10 sommets. L'option `-directed` permet d'obtenir une k-orientation.

`ring n c_1 … c_k .`

Anneaux de cordes à n sommets chacun ayant k cordes de longueur c_1,...,c_k. C'est-à-dire que chaque sommet de numéro u est adjacent aux sommets de numéro u+c_i pour chaque i. Un c_i=1 permet d'avoir un simple cycle, et de faire la différence avec les autres anneaux. Par exemple, `ring 10 5 2 .` place chaque sommet dans deux cycles de taille 2 (c_1=5) et 5 (c_2=2).

Chaque c_i peut être positif ou négatif. L'option `-directed` permet d'obtenir une k-orientation.

`cage n c_1 … c_k .`

Graphe pouvant servir à la construction de graphes n-cage, c'est-à-dire aux plus petits graphes cubiques à n sommets de maille donnée. Ils sont toujours hamiltoniens. Ils peuvent être vus comme des anneaux de cordes irréguliers. Ils sont construits à partir d'un cycle de longueur n découpé en n/k intervalles de k>0 sommets. Le i-ème sommet de chaque intervalle, disons le sommet numéro j du cycle, est adjacent au sommet numéro j+c_i du cycle (modulo n). Chaque c_i peut être positif ou négatif. Il est aussi possible de construire des graphes avec des sommets de degré 4 comme `cage 8 0 2 .` ou `chvatal`, ou avec des sommets de degré 2 comme `cage 4 2 0 .`.

`arboricity n k`

Graphe d'arboricité k à n sommets aléatoire. Ce graphe est composé de l'union de k>0 arbres aléatoires. Il est donc toujours connexe. Chacun des arbres est un arbre plan enraciné aléatoire uniforme dont les sommets sont permutés aléatoirement, sauf le premier arbre dont les sommets sont numérotés selon un parcours en profondeur. Ces graphes possèdent au plus k·(n-1) arêtes, et pour k=1 il s'agit d'un arbre. L'option `-directed` permet d'obtenir une k-orientation.

`rarytree n b z`

Arbre b-aire plan aléatoire uniforme à n nœuds internes. Il faut b≥2. Il possède bn+1+z sommets, z étant un paramètre valant 0 ou 1. La racine est de degré b+z, les autres sommets sont de degré b+1 (soit b fils) ou 1 (=feuille). Les sommets sont numérotés selon un parcours en profondeur modifié : tous les fils du sommet en cours sont numérotés avant l'étape de récursivité. Si n=1, alors le graphe est une étoile à b+z feuilles. Le dessin avec `dot` ne respecte pas le plongement de l'arbre. L'option `-directed` permet d'obtenir une 1-orientation.

`ringarytree h k r p`

Arbre de hauteur h où chaque nœud de profondeur < h a exactement k fils, sauf la racine qui en possède r. Lorsque p>0, un chemin (si p=1) ou un cycle (si p=2) est ajouté entre les sommets de même

profondeur. Notez que `ringarytree h 1 r 0` génère une étoile de degré r où chaque branche est de longueur h. Numérotés selon un parcours en profondeur, le nom des sommets est un mot correspondant au chemin depuis la racine.

`rectree h f_1 f_2 … f_d .`

Arbre récursif de hauteur h où chaque nœud profondeur < h à exactement d fils, le i-ème fils étant la racine d'un arbre similaire de profondeur h-f_i. Il faut f_i>0. Ainsi `rectree h 1 1 .` est un arbre binaire complet de hauteur h. Le graphe `rectree h 1 1 … 1 .` est un arbre complet de hauteur h identique à `ringarytree h d d 0`. L'arbre est composé d'un seul sommet si d=0 ou h<=0. C'est une étoile si h>0 et d>=h. Le nombre de sommets est exponentiel en h dès que d≥2, plus précisément de la forme $\alpha \cdot \rho$^h-1 pour des constantes α>0 et ρ>1 dépendant des f_i. Pour $f_1=f_2=1$, $\alpha=\rho=2$. Pour $f_1=1$ et $f_2=2$, $\alpha\simeq 1.23$ et $\rho\simeq 1.61$. Pour $f_1=1$ et $f_2=3$, $\alpha\simeq 1.34$ et $\rho\simeq 1.47$. Numérotés selon un parcours en profondeur, le nom des sommets est un mot correspondant au chemin depuis la racine.

`kpage n k`

Graphe k-page connexe aléatoire. Un graphe k-page peut être représenté en plaçant les sommets le long d'un cercle, en dessinant les arêtes comme des segments de droites, et en coloriant les arêtes en k>0 couleurs de façon à ce que les arêtes de chaque couleur induisent le dessin d'un graphe planaire-extérieur. La numérotation des sommets est faite le long du cercle. Les graphes 1-page sont les graphes planaires-extérieurs, les 2-pages sont les sous-graphes de graphes planaires hamiltoniens. Les graphes planaires de degré au plus 4 sont 2-pages, les 3-arbres planaires (ou graphes Apolloniens) sont 3-pages, et les cliques avec 2k-1 ou 2k sommets des k-pages. L'option `-directed` permet d'obtenir une 2k-orientation.

Ces graphes sont construits par le processus aléatoire suivant. On génère k graphes planaires-extérieurs aléatoires uniformes connexes à n sommets (plan et enraciné) grâce à une bijection avec les arbres plans enracinés dont tous les sommets, sauf ceux de la dernière branche, sont bicoloriés. On fait ensuite l'union de ces k graphes en choisissant aléatoirement la racine des arbres, sauf celui du premier planaire-extérieur, ce qui correspond à une permutation circulaire des sommets sur la face extérieure.

`cactus n`

Graphe cactus aléatoire à n sommets. Il s'agit d'arbres de cycles, c'est-à-dire de graphes connexes où chaque arête appartient à au plus un cycle. Ce sont aussi les graphes planaires-extérieurs connexes sans cordes. Ils sont générés à partir d'un `outerplanar n` dans lequel les arêtes internes (ou cordes) des composantes biconnexes ont été supprimées. L'option `-directed` permet d'obtenir une 2-orientation.

`ktree n k`

k-arbre aléatoire à n sommets. Il faut n>k≥0. C'est un graphe chordal appelé aussi graphe triangulé (*triangulated*). Il est généré à partir d'un arbre enraciné aléatoire uniforme à n-k nœuds de manière similaire à `tree n-k`. Cela constitue les « sacs » que l'on remplit avec les n sommets comme suit : on met k+1 sommets dans le sac racine connecté en clique, puis, selon un parcours en profondeur de l'arbre, on met un sommet différent pour chacun des autres sacs. Ce sommet est alors connecté à exactement k sommets choisis aléatoirement dans le sac parent qui sont ajoutés à son sac. Lorsque k=1, c'est un arbre, et lorsque k=0, c'est un stable. L'option `-directed` permet d'obtenir une k-orientation.

`kpath n k`

k-chemin aléatoire à n sommets. La construction est similaire à celle utilisée pour `ktree`, sauf que l'arbre est un chemin. Ces graphes sont des graphes d'intervalles particuliers (voir `interval n`). L'option `-directed` permet d'obtenir une k-orientation.

`kstar n k`

k-star aléatoire à n sommets. La construction est similaire à celle utilisée pour `ktree`, sauf que l'arbre est une étoile. Ces graphes, qui sont des graphes scindés (`split`), sont composés d'une clique à k+1 sommets et de n-k-1 sommets indépendants connectés à k sommets aléatoires de la clique. Il est possible d'obtenir le graphe `split n k` si à chaque fois les k sommets de la clique tirés aléatoirement sont toujours les mêmes. L'option `-directed` permet d'obtenir une k-orientation.

`rig n k p`

Graphe d'intersections aléatoires (Uniform Random Intersection Graph). Il possède n sommets, chaque sommet u étant représenté par un sous-ensemble S(u) aléatoire de {1,...,k} tel que chaque élément appartient à S(u) avec probabilité p. Il y a une arête entre u et v ssi S(u) et S(v) s'intersectent. La probabilité d'avoir une arête entre u et v est donc P_e=1-(1-p^2)^k, mais les arêtes ne sont pas indépendantes (Pr(uv|uw)>Pr(uv)). En général, pour ne pas avoir P_e qui tend vers 1, on choisit les paramètres de façon à ce que kp^2<cste. Lorsque k≥n^3, ce modèle est équivalent au modèle des graphes aléatoires d'Erdös-Reny (voir `random n p`). Si p<0, alors p est fixée au seuil théorique de connectivité, à savoir p=√(ln(n)/(nk)) si k>n et p=ln(n)/k sinon.

`apollonian n`

Graphe Apollonien aléatoire uniforme à n≥4 sommets. Les graphes Apolloniens sont les 3-arbres planaires ou encore les graphes planaires maximaux chordaux. Ils sont obtenus en subdivisant récursivement un triangle en trois autres. Ils sont 3-dégénérés, de largeur arborescente 3, et de nombre chromatique 4. La distance moyenne est Θ(log n). Ils sont en bijection avec les arbres ternaires à n-3 nœuds internes. Pour n=5, il s'agit d'un K_5 moins une arête qu'on peut obtenir aussi avec `split 5 3`. L'option `-directed` permet d'obtenir une 3-orientation.

`polygon n`

Triangulation aléatoire uniforme d'un polygone convexe à n≥3 côtés. Ce sont aussi des graphes planaires-extérieurs maximaux aléatoires. Ils sont hamiltoniens, 2-dégénérés, de largeur arborescente 2, et de nombre chromatique 3. Ils sont en bijection avec les arbres binaires (complets) à n-2 nœuds internes ou encore les mots de Dyck de longueur 2(n-2). La numérotation des sommets n'est pas cyclique le long du polygone. Ce graphe n'est pas un graphe géométrique contrairement à ses variantes utilisant `-xy convex` comme dans l'exemple ci-après.

Ex :
```
gengraph polygon 20 -dot filter circo -visu
gengraph td-delaunay 20 -xy convex2 -visu
```

`planar n f d`

Graphe planaire aléatoire composé de n faces internes de longueur f≥3, les sommets internes étant de degré au moins d et ceux de la face externe au moins 2. Ces graphes possèdent entre n+f-1 et n(f-2)+2 sommets, sont 2-connexes, 2-dégénérés, de maille f. Si d>4 alors ils sont d'hyperbolicité O(f). Ils sont construits en ajoutant itérativement les faces par le processus aléatoire suivant. Au départ, il s'agit d'un cycle de longueur f. Pour chaque nouvelle face, on ajoute un sommet u que l'on connecte à un sommet quelconque du cycle C formant le bord de la face extérieure du graphe courant. Puis on ajoute un chemin allant de u à un sommet v de C de façon à respecter : 1) la contrainte des degrés des sommets qui vont devenir internes ; et 2) la contrainte sur la longueur de la nouvelle face créée. Le sommet v est choisit uniformément parmi tous les sommets possibles de C respectant les deux contraintes. Si d<0, alors on fait comme si d=+∞ (aucun sommet ne pouvant alors être interne) et le résultat est un graphe planaire-extérieur hamiltonien, c'est-à-dire 2-connexe. Si f<0, alors chaque face créée est de longueur aléatoire uniforme prise dans [3,|f|] au lieu d'être de longueur exactement |f|. Si f=d=4, il s'agit d'un `squaregraph`. Les valeurs d=0,1,2 sont équivalentes. L'option `-directed` permet d'obtenir une 2-orientation.

`hyperbolic p k h`

Graphe issu du pavage du plan hyperbolique ou euclidien par des polygones réguliers à p≥3 côtés où chaque sommet est de degré k≥2. Le graphe est construit par couches successives de polygones, le paramètre h≥1 représentant le nombre de couches. Lorsque h=1, il s'agit d'un seul polygone, un cycle à p sommets. Dans tous les cas, ces graphes sont planaires avec O((pk)^h) sommets, ils sont 2-connexes et d'arboricité 2 pour p>3. Lorsque p=3, ils sont 3-connexes et d'arboricité 3. L'option `-directed` permet d'obtenir une 3-orientation. Sans être les mêmes graphes, il y a des similarités avec les graphes `planar n f d`. Pour paver le plan hyperbolique, représentable sur le disque de Poincaré, il faut 1/p + 1/k < 1/2. Dans ce cas, le graphe est d'hyperbolicté O(p). Pour paver le plan euclidien, il faut prendre p=k=4 (grille carrée), p=3 et k=6 (grille triangulaire), ou p=6 et k=3 (grille hexagonale). Si k≤3 et p≤5, alors le graphe n'existe que pour certaines valeurs de h≤3. Le cas k=3, p=4, h=2 correspond au `cube`, et k=3, p=5, h=3 est le graphe dodécaèdre (`dodecahedron`). Si k=2, alors h=1 et le graphe est un cycle à p sommets.

`rlt p q d`

Random Lattice Triangulation. Il s'agit d'un graphe planaire aléatoire construit à partir d'une triangulation de points situés sur la grille `p` × `q`, soit `p` colonnes de `q` lignes. Toutes les faces, sauf celle extérieure, sont des triangles. Il possède `pq` sommets et (2`p`-1)(2`q`-1)-`pq` arêtes, dont les 2(`p`+`q`-2) arêtes qui définissent le bord de la grille, les autres étant à l'intérieur. Le paramètre `d` contrôle la longueur maximum des arêtes suivant la norme Lmax (voir `-norm`). Donc seules les paires de points à distance Lmax <= `d` peuvent être connectées. Par exemple, si `d`=1, les arêtes seront soit celles de la grille (verticales ou horizontales) ou diagonales de chaque case. Si `d`<0, alors l'effet est similaire à `d`=+∞, les arêtes entre toutes paires de points étant possibles. Si `d`=0, on obtient un `stable`. Si `p`=1 ou `q`=1 (et `d`≠0), on obtient un chemin.

Ex : `gengraph rlt 14 8 2 -view scale 0.1 -visu`

Il est difficile de générer de telles grilles aléatoirement uniformément. Il faut pour cela utiliser une technique de flips avec une chaîne de Markov dont le mixing time n'est pas connu. Il est cependant bien connu que le milieu de chaque arête e d'une triangulation quelconque T de points de la grille a pour coordonnées (i+1/2,j) ou (i+1/2,j+1/2) où i,j sont des entiers. Et inversement, chaque point « milieu » des cases de la grille est toujours coupé par exactement une arête de T. Il est aussi connu que si l'on parcourt les points milieux de la grille de bas en haut et de gauche à droite, alors il n'y a au plus que deux choix possibles pour l'arête ayant ce milieu. Malheureusement, suivre ce parcours et choisir aléatoirement l'une ou l'autre de ces arêtes ne donne pas une distribution aléatoire uniforme.

La construction simplifiée utilisée ici est la suivante :

Tant qu'il reste au moins un point milieu faire :

1. Choisir uniformément un point milieu M parmi les points milieux restants
2. Déterminer la liste L des arêtes possibles ayant pour milieu M (respectant la planarité et le critère de longueur)
3. Choisir uniformément une arête de L et l'ajouter au graphe

`mikado n`

Graphe Mikado. C'est un graphe planaire aléatoire généré par le processus suivant. On tire au hasard |n| segments de lignes dans le plan, non vides et deux à eux s'intersectant en au plus un point. Si n<0, alors les sommets correspondant à des extrémités de segments sont supprimés, ce qui garantit un degré minimum de deux. Par défaut les 2|n| extrémités sont prises uniformément dans le carré $[0,1]^2$. Les sommets sont les intersections et les extrémités des segments, et les arêtes sont les sous-segments déterminés par deux sommets consécutifs d'un même segment. Il a en moyenne $n^2/8.64$ sommets. Tout graphe planaire (sans sommet isolé) peut être généré par ce procédé, chaque segment étant précisément une arête. Le tirage est réitéré tant qu'il y a deux segments colinéaires s'intersectant. C'est un graphe géométrique dont le dessin correspond aux segments et aux points intersections.

Comme une ligne coupe au plus toutes les autres, le nombre N(n) maximum de sommets du graphe vérifie N(n) = N(n-1) + (n+1) avec N(1)=2, car un segment a deux extrémités qui sont des sommets. Cela fait N(n) = 2+3+⋯+n+(n+1) = (n+1)*(n+2)/2 - 1. En moyenne, le nombre de sommets est binom(n,2)*p(4)/3 = n(n-1)*25/216 = n(n-1)/8.64, où p(4)=(5/6)^2 est la probabilité d'avoir 4 points en position convexe. En effet, étant donné 4 points A,B,C,D en position convexe, exactement une paire de segments sur ces 4 points peut s'intersecter parmi les 3 cas suivants : soit AB intersecte CD, soit AC intersecte BD, soit AD intersecte BC. En position non convexe, aucune intersection n'est possible. En passant, la probabilité d'avoir n points aléatoirement uniforme de $[0,1]^2$ en position convexe vaut p(n) = (binom(2(n-1),n-1)/n!)^2 [Valtr95].

`kneser n k r`

Graphe de Kneser généralisé. Les sommets sont tous les sous-ensembles à k éléments de {1,...,n}, et deux sommets sont adjacents ssi leurs sous-ensembles ont au plus r éléments en commun. Il faut n≥k≥1 et k≥r≥0. Ils ont un lien avec les graphes `johnson` J(n,k) et `odd`. Le graphe de Kneser K(n,k) classique est obtenu avec r=0.

Le graphe K(n,k) est régulier de degré binom(n-k,k). Il est connexe si n>2k, sa clique maximum a ⌊n/k⌋ sommets, et son nombre chromatique vaut n-2k+2 pour tout n≥2k-1>0 (établi par Lovász). K(n,1) est une clique et K(n,n-1) un stable. K(5,2) est le graphe `petersen`.

`gpetersen n r`

Graphe de Petersen généralisé P(n,r), 0≤r<n/2. Ce graphe cubique possède 2n sommets qui sont u_1,...,u_n,v_1,...,v_n. Les arêtes sont, pour tout i : u_i-u_{i+1}, u_i-v_i et v_i-v_{i+r} (indice modulo n). Il peut être dessiné tel que toutes ses arêtes sont de même longueur (*unit distance graph*). Ce graphe est biparti ssi n est pair et r est impair. C'est un graphe de Cayley ssi r^2=1 (modulo n). P(n,r) est hamiltonien ssi r≠2 ou n≠5 (modulo 6). P(n,r) est isomorphe à P(n,(n-2r+3)/2)). P(4,1) est le `cube`, P(5,2) le graphe `petersen`, P(6,2) le graphe `durer`, P(8,3) le graphe `mobius_kantor`, P(10,2) le `dodecahedron`, P(10,3) le graphe `desargues`, P(12,5) le graphe `nauru`, P(n,1) un `prism`.

`squashed n k p`

Squashed Cube aléatoire à n sommets. Il faut 0<k<n et p∈[0,1]. Les sommets sont des mots aléatoires de k lettres sur {0,1,'*'} où p est la probabilité d'obtenir '*'. La probabilité d'obtenir 0 est la même que celle d'obtenir 1, soit (1-p)/2. Deux sommets sont adjacents si la distance de Hamming entre leurs mots vaut 1 avec la convention que la distance à la lettre '*' est nulle. Lorsque que p=0, le graphe généré est un sous-graphe isométrique de l'`hypercube` où certains sommets sont dupliqués en sommets jumeaux non adjacents (ce sont les sommets correspondant au même mot). En particulier, le graphe est biparti et la distance entre deux sommets est donnée par la distance de Hamming entre leurs mots, sauf s'ils ont le même mot. Si p<0, alors p est fixé à l'équiprobabilité de chacune des lettres, soit p=1/3.

antiprism n

Graphe composé de deux cycles à n sommets connectés par 2n triangles. Le prisme est similaire sauf que pour ce dernier les deux cycles sont connectés par des carrés. Il est ainsi planaire, 4-régulier, possède 2n sommets et 4n arêtes. C'est aussi le dual du trapézoèdre n-gonal (`trapezohedron`).

rpartite a_1 … a_k .

Graphe k-parti complet K_{a_1,...,a_k}. Ce graphe possède a_1 + ⋯ + a_k sommets partitionnés en k parts. La i-ème part contient a_i sommets numérotés consécutivement dans l'intervalle [A_i, A_i + a_i [où A_i = a_1 + ⋯ + a_{i-1}. Les sommets i et j sont adjacents ssi i et j appartiennent à des parts différentes.

ggosset p d_1 v_1 … d_k v_k .

Graphe de Gosset généralisé. Les sommets sont tous les vecteurs entiers (et leurs opposés) de dimension d = d_1 + ⋯ + d_k dont les coordonnées comprennent exactement $d_i \geq 1$ fois la valeur v_i. Le nombre de sommets est donc n = 2·∏_i binomial(d-(∑_{j<i}d_i), d_i). Il existe une arête entre les vecteurs u et v ssi le produit scalaire entre u et v vaut l'entier p. Des valeurs intéressantes sont par exemple `ggosset 1 2 -1 2 0 .` ou `ggosset 8 2 3 6 -1 .` (le graphe de Gosset, `gosset`).

schlafli

Graphe de Schläfli. Il s'agit du sous-graphe induit par les voisins d'un quelconque sommet du graphe `gosset`. Il possède 27 sommets, 216 arêtes et est 16-régulier. Il est sans griffe, hamiltonien, de diamètre 2, de maille 3 et de nombre chromatique 9.

crown n

Graphe biparti régulier à 2n sommets où le i-ème sommet de la première partie de taille n est voisin au j-ème sommet de la seconde partie ssi i≠j. Pour n=3, il s'agit du `cycle` à 6 sommets, pour n=4, il s'agit du `cube` (à 8 sommets).

half-graph n

Graphe biparti à 2n sommets où le i-ème sommet de la première partie de taille n est voisin au j-ème sommet de la seconde partie ssi i ≤ j. Pour n=2, il s'agit de `path 4`. Ce graphe ne possède pas de chemin induit à 5 sommets ou plus.

sum-graph n

Graphe connexe à n sommets où i est adjacent à j ssi i+j < n. C'est l'unique graphe connexe où tous les degrés des sommets sont différents, sauf deux. Le degré du sommet i vaut deg(i) = n-i - (i<n/2) ∈ {n-i,n-i-1}. Les deux sommets de même degré sont v et v+1, avec v=⌈n/2⌉ et deg(v)=⌊n/2⌋. Le graphe complémentaire de `sum-graph n`, qui a aussi la propriété des degrés uniques sauf deux, correpond au graphe composé d'un sommet isolé et de `sum-graph n-1`.

`flip n`

Graphe des flips des triangulations d'un polygone convexe à n>2 sommets. Les sommets, qui sont les triangulations, sont en bijection avec des arbres binaires (complets) à m=n-2 nœuds internes qui sont codés par les mots de Dyck de longueur 2m (mots que l'on peut afficher avec `-label 1`). Le nombre de sommets est donc C(m) = binom(2m,m)/(m+1), le nombre de Catalan d'ordre m. Les adjacences peuvent être vues aussi comme des rotations d'arbres. Le diamètre est 2n-10 pour n>12. Le nombre chromatique n'est pas connu. On ne sait pas s'il est constant. Il vaut 3 pour 5 ≤ n ≤ 9, et 4 pour n=10 et 11.

`interval n`

Graphe d'intersection de n intervalles d'entiers aléatoires uniformes pris dans [0,2n[. Des graphes d'intervalles peuvent aussi être générés par `kpath n k`.

`circle n`

Graphe d'intersection de n cordes aléatoires d'un cercle. Il est réalisé par le graphe de chevauchements (intersections sans inclusion) de n intervalles d'entiers aléatoires uniformes pris dans [0,2n[. Les graphes de permutation et les planaires extérieurs sont des exemples de circle graphs.

`permutation n`

Graphe de permutation sur une permutation aléatoire uniforme des entiers de [0,n[.

`prime n`

Graphe à n sommets tel que i est adjacent à j ssi i>1 et j divisible par i. Avec `-directed`, les arcs vont des diviseurs vers les multiples.

`paley n`

Graphe de Paley à n sommets. Deux sommets sont adjacents ssi leur différence est un carré modulo n. Il faut que n soit la puissance d'un nombre premier et que n≡1 (mod 4), mais le graphe est aussi défini pour les autres valeurs. Les premières valeurs possibles pour n sont : 5, 9, 13, 17, 25, 29, 37, 41, 49,... Ces graphes sont hamiltoniens. Si n est simplement premier, alors ils sont de plus auto-complémen-

taires et réguliers. Paley 17 est le plus grand graphe G où ni G ni son complémentaire ne contient K_4, d'où Ramsey(4)=18.

`mycielski k`

Graphe de Mycielski de paramètre (nombre chromatique) k. C'est un graphe sans triangle, k-1 (sommets) connexe, et de nombre chromatique k. Le premier graphe de la série est M2 = K2, puis on trouve M3 = C5 et M4 = `grotzsch`.

`barbell n_1 n_2 p`

Graphe des haltères (Barbell Graph) composé de deux cliques de n_1 et n_2 sommets reliées par un chemin de longueur p. Il possède n_1+n_2+p-1 sommets. Si p=0 (p=-1), le graphe est composé de deux cliques ayant un sommet (une arête) en commun. Plus généralement, si p≤0, le graphe est composé de deux cliques s'intersectant sur 1-p sommets. Si n_1 ou n_2 est négatif, alors on aura un cycle plutôt qu'une clique.

`chess p q u v`

Graphe géométrique composé de pq sommets représentant les cases d'un échiquier p×q (p colonnes de q lignes), deux cases étant connectées s'il existe un déplacement d'une case vers l'autre avec un saut de u cases selon un des axes et v selon l'autre. Le « knight graph » classique est donc un `chess 8 8 1 2` (voir `knight 8 8`), `chess n 2 0 1` correspond à `ladder n`, et `chess p q 0 1` correspond à `mesh p q`.

`sat n m k`

Graphe aléatoire issu de la réduction du problème k-SAT à Vertex Cover. Le calcul d'un Vertex Cover de taille minimum pour ce graphe est donc difficile pour k>2. Soit F une formule de k-SAT avec n>0 variables x_i et m>0 clauses CNF de k>0 termes. Le graphe généré par `sat n m k` possède un Vertex Cover de taille n+(k-1)m si et seulement si F est satisfiable.

Ce graphe est composé d'une union de n arêtes indépendantes et de m cliques à k sommets, plus des arêtes dépendant de F connectant certains sommets des cliques aux n arêtes. Les n arêtes représentent les n variables, une extrémité pour x_i, l'autre pour ¬(x_i). Ces sommets ont des numéros dans [0,2n[, x_i correspond au sommet 2i-2 et ¬(x_i) au sommet 2i-1, i=1…n. Les sommets des cliques ont des numéros consécutifs ≥ 2n et correspondent aux clauses. Le p-ème sommet de la q-ème clique (pour p=1…k et q=1…m) est connecté à l'une des extrémités de la i-ème arête (pour i=1…n) ssi la p-ème variable de la q-ème clause est x_i ou ¬(x_i).

La formule F est construite en choisissant indépendamment et uniformément pour chacune des m clauses et chacun des k termes une des variables parmi x_1,…,x_n,¬(x_1),…,¬(x_n). Ainsi chaque sommet

d'une clique possède exactement un voisin (choisi aléatoirement uniforme) parmi les 2n extrémités d'arêtes.

`calls_tree <définition> f n_1 … n_k .`

Arbre des appels pour la fonction récursive définie par la chaine `définition`, en partant de `f(n_1, …, n_k)`.

La variante `-variant 1` permet d'obtenir des étiquettes plus détaillées (avec `-label 1`), avec nom de fonction et valeur.

Voir `tree_fibo`, `tree_part` et `tree_binom` pour des exemples.

Attention : Les priorités des opérateurs ne sont pas respectées dans l'analyse de la définition de fonction, veuillez mettre des parenthèses.

Ex :
```
gengraph calls_tree "syr(1) = 0; syr(n) [n % 2 = 0] = syr(n / 2);
    syr(n) = syr((3 * n) + 1)" syr 42 . -label 1 -visu
```

Il est possible de spécifier plusieurs gardes pour un même cas de fonction, chacune entre crochets. Il est possible de combiner l'usage de plusieurs fonctions avec des noms différents. Les opérateurs de garde pris en charge sont `= <> < > <= >=`. Les opérateurs de calcul pris en charge sont `+ - * / % ^`. La division utilisée est la division entière. Toutes les valeurs spécifiées ou calculées doivent être des entiers compris dans l'intervalle $]-2^{31}, 2^{31}[$.

`kout n k`

Graphe à n sommets k-dégénéré crée par le processus aléatoire suivant : les sommets sont ajoutés dans l'ordre croissant de leur numéro, i=0,1,...,n-1. Le sommet i est connecté à d voisins qui sont pris aléatoirement uniformément parmi les sommets dont le numéro est < i. La valeur d est choisie aléatoirement uniformément entre 1 et min{i,k}. Il faut 0<k<n. Le graphe est connexe, et pour k=1, il s'agit d'un arbre. L'option `-directed` permet d'obtenir une k-orientation.

Ex :
```
gengraph kout 50 2 -directed -vcolor deg -vcolor pal wbn -vsize -visu
```

`expander n k`

Graphe à n sommets composé de k>0 cycles hamiltoniens aléatoires. Le degré des sommets varie entre 2 et 2k pour n>2. Il possède le cycle 0,1,...,n-1,0 comme cycle hamiltonien, et a la propriété d'expansion à partir de k≥4. Plus précisément, avec grande probabilité, les valeurs propres de la matrice d'adjacence du graphe sont ≤ 2√(2k). On rappelle que la constante d'expansion, isopérimétrique, ou de Cheeger, h(G) d'un graphe d-régulier G est toujours comprise entre $(d-\lambda_2)/2 \le h(G) \le \sqrt{(2d(d-\lambda_2))}$ où λ_2

est la deuxième plus grande valeur propre de la matrice d'adjacence de G. L'option `-directed` permet d'obtenir une k-orientation.

`margulis n`

Graphe de Margulis à n^2 sommets. Il s'agit d'un expandeur avec $\lambda_2 \leq 5\sqrt{2}$ (cf. graphe "expander n k") de degré maximum 8 et de degré minimum 2. Les sommets sont les paires d'entiers (x,y) avec x,y ∈ [0,n[avec les 8 adjacences suivantes : (x+y,y), (x-y,y), (x,y+x), (x,y-x), (x+y+1,y), (x−y+1,y), (x,y+x+1) et (x,y−x+1), toutes ces opérations étant modulo n.

`parachute n`

Graphe Parachute. Il est planaire à n+3 sommets composés du graphe `fan n 2` dont un des deux sommets de degré n possède un sommet pendant. Le graphe classique correspond à n=4. C'est le complémentaire du graphe `parapluie n`.

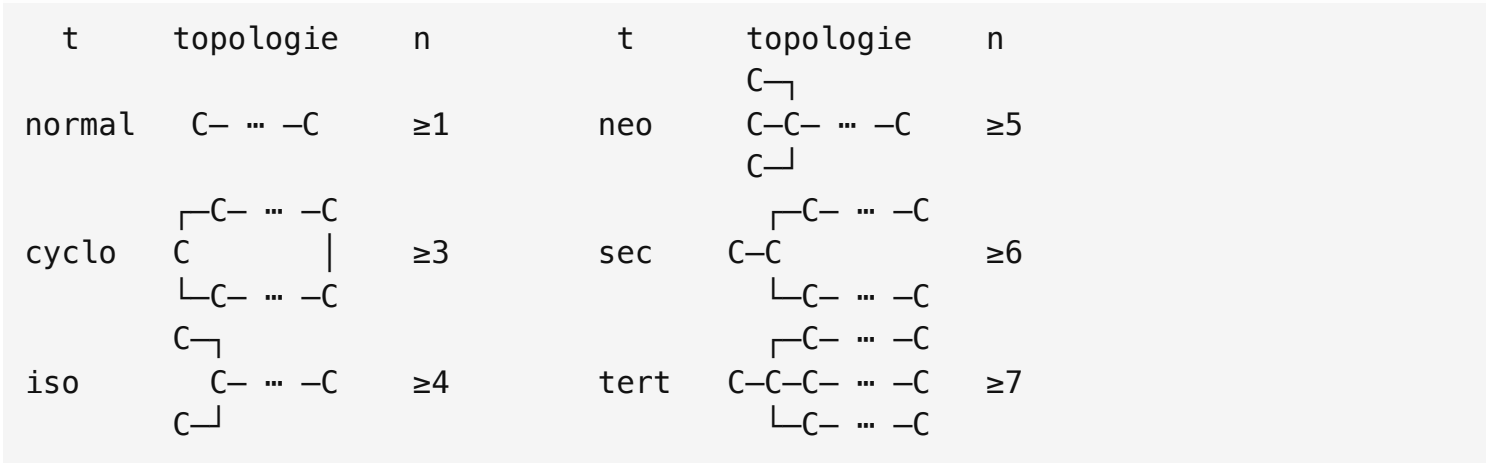
`alkane <type> n`

Graphe planaire dont les sommets sont de degré 1 ou 4 représentant la structure moléculaire d'hydrocarbure alkalin à n atomes de carbone.

Le paramètre `type` (voir ci-dessous les six types possibles) contrôle la topologie des liaisons simples entre atomes de carbone (C), les atomes d'hydrogène (H) étant des sommets pendants de sorte que chaque atome C soit de degré 4.

- Les topologies type≠`normal` ne sont définies qu'à partir d'une certaine valeur de n.
- Les alkalins, de formule C_n H_{2n+2} si type≠`cyclo`, sont des arbres de 3n+2 sommets, alors que les cycloalkalins, de formule C_n H_{2n} si type=`cyclo`, ont 3n sommets et possèdent un cycle.

Chaque type peut être abrégé par ses 2 premières lettres. L'option `-label 1` activée par défaut permet de distinguer les atomes C et H.



Il est possible d'utiliser les alias suivants :

```
n-alkane n ......... (= alkane normal n)
cyclo-alkane n ..... (= alkane cyclo n)
iso-alkane n ....... (= alkane iso n)
neo-alkane n ....... (= alkane neo n)
sec-alkane n ....... (= alkane sec n)
tert-alkane n ...... (= alkane tert n)
t-methane .......... (= alkane t 1)
t-ethane ........... (= alkane t 2)
t-propane .......... (= alkane t 3)
t-butane ........... (= alkane t 4)
t-pentane .......... (= alkane t 5)
t-hexane ........... (= alkane t 6)
t-heptane .......... (= alkane t 7)
t-octane ........... (= alkane t 8)
t-nonane ........... (= alkane t 9)
```

Si t=`normal`, le préfixe peut être omis. Par exemple : `methane` (= `alkane normal 1`). Étant donné la contrainte sur n ci-dessus, certains alias ne donnent aucun résultat.

icosahedron

Icosaèdre : graphe planaire 5-régulier à 12 sommets. Il possède 30 arêtes et 20 faces qui sont des triangles. C'est le dual du dodécaèdre (`dodecahedron`).

rdodecahedron

Rhombic-dodécaèdre : graphe planaire à 14 sommets avec des sommets de degré 3 ou 4. Il possède 21 arêtes et 12 faces qui sont des carrés. C'est le dual du cuboctaèdre (`cuboctahedron`).

deltohedron n
trapezohedron n

Deltoèdre ou trapézoèdre n-gonal : graphe composé de 2n faces en forme de cerf-volant (deltoïdes) décalées symétriquement. C'est un graphe planaire de 2n+2 sommets et 4n arêtes où toutes les faces sont des carrés. C'est aussi le dual de l'antiprisme n-gonal (`antiprism`). Il s'agit d'un `cube` si n=3.

tutte

Graphe de Tutte. C'est un graphe planaire cubique 3-connexe à 46 sommets qui n'est pas hamiltonien.

`hgraph`

Arbre à six sommets dont quatre feuilles en forme de H.

`rgraph`
`fish`

Fish Graph, graphe à six sommets en forme de R ou de poisson. Il est composé d'un cycle à quatre sommets dont un ayant deux sommets pendants.

`cricket`

Cricket Graph, graphe à cinq sommets composé d'un triangle où à l'un des sommets est attaché deux sommets pendant (de degré 1).

`moth`

Moth Graph, graphe à six sommets composé de deux triangles partageant une arête et de deux sommets pendants (degré 1) attachés à un sommet de degré trois.

`bull`

Bull Graph, graphe à cinq sommets auto-complémentaire en forme de A.

`antenna`

Antenna Graph, graphe planaire à six sommets formé du graphe house et d'un sommet pendant attaché à son toit. Plus précisément, il est composé d'un carré, d'un triangle partageant une arête et d'un sommet pendant au sommet de degré 2 du triangle. C'est le complémentaire du graphe formé d'un carré et de deux triangles partageant deux arêtes consécutive du carré.

`suzuki`

Graphe de Suzuki (2010). C'est l'unique graphe 1-planaire à n=11 sommets et ayant le nombre maximum d'arêtes, soit 4n-8 arêtes (ici 36 donc).

`harborth`

Graphe de Harborth. C'est un graphe planaire 4-régulier à 52 sommets qui est distance unitaire, aussi appelé graphe allumette (voir `theta0` et `diamond`). Il peut ainsi être dessiné sans croisement d'arêtes qui ont toutes la même longueur.

doily

Graphe Doily (de Payne). C'est un graphe de 15 sommets qui est un carré généralisé pouvant être représenté par 15 points et 15 lignes, avec 3 points par ligne et 3 lignes par point, et sans triangle.

herschel

Graphe de Herschel. C'est le plus petit graphe planaire 3-connexe qui ne soit pas hamiltonien. Il est biparti, possède 11 sommets et 18 arêtes.

goldner_harary

Graphe de Goldner-Haray. C'est le plus petit graphe planaire maximal qui ne soit pas hamiltonien. Il possède 11 sommets et donc 27 arêtes (voir aussi Herchel). C'est un 3-arbre planaire (voir `apollonian`).

triplex

Graphe cubique de maille 5 à 12 sommets 1-planaire pouvant être dessiné avec seulement deux croisements d'arête. Un des cinq graphes (avec le `petersen`) à être cycliquement-5-connexe (McCuaig 1992).

jaws

Graphe cubique de maille 5 à 20 sommets qui est un *doublecross*, c'est-à-dire dessinable sur le plan avec deux paires d'arêtes se croisant sur la face extérieure. Il est donc 1-planaire. Tout graphe theta-connecté sans `petersen` mais avec Jaws comme mineur est un doublecross.

starfish

Graphe cubique de maille 5 à 20 sommets non planaire, mais peut-être dessiné comme une étoile à cinq branches avec une couronne centrale à 15 sommets formant un circulant avec une corde de longueur 3. Un graphe theta-connecté (cf. Seymour et al. 2015) ssi il ne contient pas de `petersen` comme mineur, si c'est un graphe apex (planaire plus un sommet), un *doublecross* (voir `jaws`) ou un `starfish`.

fritsch

Graphe de Fritsch. Il est planaire maximal à 9 sommets qui peut être vu comme un graphe `hajos` dans un triangle. C'est, avec le graphe `soifer`, le plus petit contre-exemple à la procédure de coloration de Kempe. C'est le plus petit graphe où l'heuristique de degré minimum donne cinq couleurs.

soifer

Graphe de Soifer. Il est planaire maximal à 9 sommets. C'est, avec le graphe `fritsch`, le plus petit contre-exemple à la procédure de coloration de Kempe. C'est le plus petit graphe où l'heuristique de degré minimum donne cinq couleurs.

poussin

Graphe de Poussin. Il est planaire maximal à 15 sommets. C'est un contre-exemple à la procédure de coloration de Kempe.

heawood4

Graphe de Heawood pour la conjecture des 4 couleurs, contre-exemple de la preuve de Kempe. Il est planaire maximal avec 25 sommets, est de nombre chromatique 4, de diamètre 5, de rayon 3 et hamiltonien.

errera

Graphe d'Errera. Il est planaire maximal à 17 sommets. C'est un contre-exemple à la procédure de coloration de Kempe.

kittell

Graphe de Kittell. Il est planaire maximal à 23 sommets. C'est un contre-exemple à la procédure de coloration de Kempe.

frucht

Graphe de Frucht. Il est planaire cubique à 12 sommets. Il n'a pas de symétrie non triviale. C'est un graphe `halin` de nombre chromatique 3, de diamètre 4 et de rayon 3.

treep p

Arbre aléatoire à p>2 feuilles sans sommets internes de degré deux. Il possède entre p+1 et 2p-2 sommets. Ce graphe est à la base de la construction des graphes `halin`.

halin p

Graphe de Halin aléatoire basé sur un arbre à p>2 feuilles. Il possède entre p+1 et 2p-2 sommets. Il est constitué d'un arbre sans sommets de degré deux dont les p feuilles sont connectés par un cycle (de p

arêtes). Ces graphes planaires de degré minimum au moins trois sont aussi arête-minimale 3-connexes, hamiltonien (et le reste après la suppression de n'importe quel sommet), de treewidth exactement 3 (ils contiennent K_4 comme mineur). Ils contiennent toujours au moins trois triangles et sont de nombre chromatique 3 ou 4.

`butterfly d`

Graphe papillon (*butterfly*) de dimension d. Les sommets sont les paires (x,i) où x est un mot binaire de d bits et i un entier de [0,d]. Les sommets peuvent être représentés en d+1 niveaux chacun de 2^d sommets, les arêtes connectant les niveaux consécutifs. Le sommet (x,i) est adjacent à (y,i+1) ssi les bits de x sont identiques à ceux de y sauf pour celui de numéro i+1 (le bit 1 étant le bit de poids le plus faible). Il possède (d+1)·2^d sommets et d·2^(d+1) arêtes, les sommets de niveau 0 et d étant de degré 2 les autres de degré 4.

`shuffle d`

Graphe Shuffle-Exchange de dimension d. Les sommets sont les mots binaires de d lettres. Les sommets w et w' sont voisins si w et w' diffèrent du dernier bit, ou bien si w' peut être obtenu par décalage cyclique à droite ou à gauche de w.

`debruijn d b`

Graphe de De Bruijn de dimension d≥0 et de base b>0. Il a b^d sommets qui sont tous les mots de d lettres sur un alphabet de b lettres. Le sommet (x_1,...,x_d) est voisin des sommets (x_2,...,x_d,*). Ce graphe est hamiltonien, de diamètre d et le degré de chaque sommet est 2b, 2b-1 ou 2b-2. Pour d=3 et b=2, le graphe est planaire.

`kautz d b`

Graphe de Kautz de dimension d>0 et de base b>1. Il a b·(b-1)^(d-1) sommets qui sont tous les mots de d lettres sur un alphabet de b lettres avec la contrainte que deux lettres consécutives doivent être différentes. L'adjacence est celle du graphe de De Bruijn. C'est donc un sous-graphe induit de De Bruijn (`debruijn d b`). Il est hamiltonien, de diamètre d et le degré de chaque sommet est 2b-2 ou 2b-3. Pour d=b=3 le graphe est planaire.

`linial n t`

Neighborhood graph des cycles introduit par Linial. C'est le graphe de voisinage des vues de taille t d'un cycle orienté symétrique à n sommets ayant des identifiants uniques de [0,n[. Il faut n≥t>0 et n≥2. Les sommets sont les t-uplets d'entiers distincts de [0,n[. Le sommet (x_1,...,x_t) est voisin des sommets (x_2,...,x_t,y) où y≠x_1 si n>t et y=x_1 si n=t. Le nombre chromatique de ce graphe est k ssi il existe un algorithme distribué qui en temps t-1 (resp. en temps (t-1)/2 avec t impair) peut colorier en k couleurs

tout cycle orienté (resp. orienté symétrique) à n sommets ayant des identifiants uniques et entiers de [0,n[. C'est un sous-graphe induit de `linialc n t`, et donc du graphe de Kautz (`kautz t n`) et de De Bruijn (`debruijn t`). Le nombre de sommets est n·(n-1)**---**(n-t+1). Certaines propriétés se déduisent du graphe `linialc n t`. Pour n=4 et t=2, il s'agit du cuboctaèdre (`cuboctahedron`).

`linialc m t`

Neighborhood graph des cycles colorés. Il s'agit d'une variante du graphe `linial n t`. La différence est que les sommets du cycle n'ont plus forcément des identités uniques, mais seulement une m-coloration avec m≤n. Il faut m≥t≥0 et m≥2. L'adjacence est identique, mais les sommets sont les t-uplets (x_1,...,x_t) d'entiers de [0,m[tels que x_i≠x_{i+1}. Il s'agit donc d'un sous-graphe induit de `linialc m t`, lui-même sous-graphe induit du graphe de Kautz (`kautz t m`) et donc de De Bruijn (`debruijn t m`). Le nombre de sommets est m·(m-1)^{t-1} et son degré est ≤ 2·(m-1). La taille de la clique maximum est 3 si m>2 et t>1. Le nombre chromatique de ce graphe pour t=3 est 3 pour m=4, 4 pour 5≤m≤24. Pour 25≤m≤70, c'est au moins 4 et au plus 5, la valeur exacte n'étant pas connue. Tout comme `linial 4 2`, pour m=4 et t=2, il s'agit du cuboctaèdre (`cuboctahedron`).

`pancake n`

Graphe « pancake » de dimension n. Il a n! sommets qui sont les permutations de {1,...,n} et (n-1)-régulier. Une permutation, c'est-à-dire un sommet, est voisine de toutes celles obtenues en retournant un de ces préfixes. Plus précisément, les sommets x=(x_1,...,x_n) et y=(y_1,...,y_n) sont adjacents s'il existe un indice k tel que y_i=x_i pour tout i>k et y_i=x_{k-i} sinon. Son diamètre, qui est linéaire en n, n'est pas connu précisément. Les premières valeurs connues, pour n=1...17, sont : 0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19. Donc les diamètres 2,6,12 n'existent pas.

`bpancake n`

Graphe « burn pancake » de dimension n. Il a n!·2^n sommets qui sont les permutations signées de {1,...,n}. Les sommets x=(x_1,...,x_n) et y=(y_1,...,y_n) sont adjacents s'il existe un indice k tel que y_i=x_i pour tout i>k et y_i=-x_{k-i} sinon. Dit autrement la permutation de y doit être obtenue en retournant un préfixe de x et en inversant les signes. Par exemple, le sommet (+2,-1,-5,+4) est voisin du sommet (+5,+1,-2,+4). Comme le graphe `pancake`, c'est un graphe (n-1)-régulier de diamètre linéaire en n.

`gpstar n d`

Graphe « permutation star » généralisé de dimension n. Il a n! sommets qui sont les permutations de {1,...,n}. Deux sommets sont adjacents si leurs permutations diffèrent par d positions. Si d<2, il s'agit d'un `stable`. C'est un graphe régulier.

`pstar n`

Graphe « permutation star » de dimension n. Il a n! sommets qui sont les permutations de {1,...,n}. Deux sommets sont adjacents si une permutation est obtenue en échangeant le premier élément avec un autre. Le graphe est (n-1)-régulier. Le graphe est biparti et de diamètre ⌊3(n-1)/2⌋. C'est un sous-graphe induit d'un `gpstar n 1`.

`hexagon p q`

Grille hexagonale p × q. C'est un planaire composé de p rangées de q hexagones, le tout arrangé comme un nid d'abeilles. Ce graphe peut aussi être vu comme un mur de p rangées de q briques, chaque brique étant représentée par un cycle de longueur 6. Il possède (p+1)·(2p+2)-2 sommets et est de degré maximum 3. Sont dual est le graphe whexagon.

Ex : `gengraph hexagon 20 20 -op del-e 0.2 -op maincc -visu`

`whexagon p q`

Comme le graphe `hexagon p q` sauf que chaque hexagone est remplacé par une roue de taille 6 (chaque hexagone possède un sommet connecté à ses 6 sommets). C'est le dual de l'hexagone. Il possède p·q sommets de plus que l'`hexagon p q`.

`hanoi n b`

[Ce graphe est à revoir, ce n'est plus le graphe Hanoï au-delà de b>3. Voir p. 257 du livre [HKP18] où il y a la description du graphe de Hanoï pour n et b quelconque.]

Graphe de Hanoï généralisé, le graphe de de Hanoï classique est obtenu avec b=3. Le paramètre n correspond au nombre de disques et b au nombre de piquets. La numérotation des sommets (avec `-label 1`) correspond aux mots de n lettres choisies parmi l'alphabet [0,b[, indiquant sur quel piquet est chacun des disques. Par exemple, si n=4, alors « 2010 » indique que le disque 1 est sur le piquet 2, le disque 2 et le disque 4 sur le piquet 0 et le disque 3 sur le piquet 1. Sur un piquet donné, les disques sont ordonnés selon leur taille croissante. Une arête correspond à un mouvement possible. Par exemple « 2010 » est voisin de « 2012 ». Pour b=3, il est 3-coloriable et hamiltonien. Résoudre le problème de la tour de Hanoï revient à chercher un chemin de « 00...0 » à « 22...2 ». Le plus court d'entre eux est de longueur ≤ 2^n - 1 (c'est égal si b=3).

Il est planaire avec b^n sommets et peut être défini de manière récursive comme suit. Le niveau n>0 est obtenu en faisant b copies du niveau n-1 qui sont connectées comme un cycle par une arête, le niveau 0 étant le graphe à un sommet. On obtient le graphe de `sierpinski n b` en contractant ces arêtes-là. Il faut b≥2 et n≥0. Lorsque n=2, on obtient un sorte de fleur, pour n=1 c'est un cycle et pour b=2 il s'agit d'un chemin.

`sierpinski n b`

Graphe de Sierpiński généralisé. Le graphe classique, le triangle Sierpiński qui est planaire, est obtenu avec b=3. Il a ((b-2)·b^n+b)/(b-1) sommets et est défini de manière récursive comme suit. Le niveau n est obtenu en faisant b copies du niveau n-1 qui sont connectés comme un cycle, le niveau 1 étant un cycle de b sommets. Il faut b≥3 et n≥1. À la différence du graphe `hanoi`, les arêtes du cycle sont contractées. Le graphe `hajos` est obtenu avec n=2 et b=3. Pour n=1 il s'agit d'un `cycle`.

`banana n k`

Arbre à n·(k+1)+1 sommets composés de n>0 copies d'étoiles à k branches connectées, par une feuille, à un unique sommet. Si k=0, il s'agit d'un `stable` à k+1 sommets.

`moser`

Graphe « Moser spindle » découvert par les frères Moser. C'est un « unit distance graph » du plan (deux points sont adjacents s'ils sont à distance exactement 1) de nombre chromatique 4. Il est planaire et possède 7 sommets. On ne connaît pas d'unit distance graph avec un nombre chromatique supérieur. C'est aussi le complémentaire du graphe K_{3,3} dont une arête a été subdivisée.

`markstrom`

Graphe de Markström. Il est cubique planaire à 24 sommets. Il n'a pas de cycle de longueur 4 et 8.

`robertson`

Graphe de Robertson. C'est le plus petit graphe 4-régulier de maille 5. Il a 19 sommets, est 3-coloriable et de diamètre 3.

`wiener_araya`

Graphe découvert en 2009 par Wiener & Araya. C'est le plus petit graphe hypo-hamiltonien planaire connu, c'est-à-dire qu'il n'a pas de cycle hamiltonien mais la suppression de n'importe quel sommet le rend hamiltonien. Il possède 42 sommets, 67 arêtes, et est de diamètre 7.

`zamfirescu`

Graphe de Zamfirescu à 48 sommets découvert en 2007. Il est planaire et hypo-hamiltonien. C'est le second plus petit (voir `wiener_araya`). Il possède 76 arêtes et a un diamètre de 7.

`hatzel`

Graphe de Hatzel. Il est planaire, de diamètre 8, possède 57 sommets et 88 arêtes, et est hypo-hamiltonien (voir `wiener_araya`). C'était le plus petit planaire hypo-hamiltonien connu avant le graphe de

Zamfirescu.

`clebsch n`

Graphe de Clebsch d'ordre n. Il est construit à partir d'un `hypercube` de dimension n en ajoutant une arête entre chaque paire de sommets opposés, c'est-à-dire à distance n. Le graphe classique de Clebsch est réalisé pour n=4 dont le diamètre est deux.

`gear n`

Graphe planaire à 2n+1 sommets composé d'une roue à n rayons (`wheel n`) et de n sommets chacun connecté à deux sommets voisins consécutifs du bord de la roue. Il est construit à partir du graphe `cage 2n 2 0 .` auquel on ajoute un sommet central. Pour n=3, c'est le complémentaire de `helm 3`.

`helm n`

Graphe planaire à 2n+1 sommets composé d'une roue à n≥3 rayons (`wheel n`) et de n sommets pendants connectés au bord de la roue. Pour n=3, c'est le complémentaire de `gear 3`.

`haar n`

Graphe de Haar H(n) pour l'entier n>0. C'est un graphe biparti régulier possédant 2k sommets où k=1+⌊log₂(n)⌋ est le nombre de bits dans l'écriture binaire de n. Ses sommets sont les u_i et v_i pour i=0,...,k-1. Le sommet u_i est adjacent à v_{i+j mod k} ssi le bit j de n vaut 1 (j=0,...,k-1). Si n est impair, H(n) est connexe et de maille 4 ou 6. La valeur maximale de n est 2^32-1 = 4 294 967 295 correspondant à un graphe de 64 sommets. On retrouve respectivement les graphes de Franklin (n=37), de Heawood (n=69) et de Möbius-Kantor (n=133). On a aussi que H(2^n-1) est le biparti K_{n,n}, H(2^n) est « matching n+1 », H(2^n+1) est un cycle à n+1 sommets, H(2^n+3) est le `mobius n+1` si n est pair et le `prism n+1` si n est impair, et H(3·2^n-1) est le `crown n+2`.

`turan n r`

Graphe de Turán à n sommets et r parts. Il s'agit d'un graphe r-parti complet de n sommets avec r-(n mod r) parts de ⌊n/r⌋ sommets et (n mod r) parts de ⌈n/r⌉ sommets. Il faut n≥r>0. On peut aussi le définir comme le graphe ayant une arête entre i et j ssi |i-j| mod r ≠ 0. C'est la définition utilisée pour générer ce graphe. Il est régulier lorsque r divise n. Il possède ⌊(r-1)n²/(2r)⌋ arêtes et est de nombre chromatique r. C'est le graphe sans clique de taille r+1 ayant le plus grand nombre d'arêtes. Lorsque n=r, il s'agit d'une `clique`. Lorsque n=r+1, il s'agit d'une clique moins une arête. Lorsque n=2r, il s'agit du « cocktail party graph » et pour n=8 et r=4 le graphe est 1-planar. Lorsque n=3r, il s'agit du graphe de Moon-Moser, le graphe possédant le plus grand nombre de cliques maximales (soit 3^(n/3)). Lorsque n=6 et r=3, c'est l'octaèdre (`octahedron`).

`klein p q`

Maillage quadrangulaire de genre un, incluant le tore, la bouteille de Klein et le plan projectif. C'est un graphe essentiellement 4-régulier composé d'une grille |p|×|q| augmentée d'arêtes connectant les bords opposés. La connexion est torique ou en twist suivant le signe de chaque dimension (>0 pour torique et <0 pour twist). Pour la bouteille de Klein, il faut p·q<0, et pour le plan projectif il faut p<0 et q<0. Tous les sommets sont de degré 4 sauf pour le plan projectif avec une dimension impaire ou que |p|=|q|=2 (K_4). Il y a dans ce cas 4 sommets de degré 3. Le nombre chromatique est 4 si |p|≠1 et |q|≠1 (cas d'un cycle sinon) et qu'il y a une dimension <0 impaire, ou que |p|=|q|=2 (K_4). Sinon il est < 4. Les plus petits exemples avec un nombre chromatique 4, à part K_4, sont `klein 3 -3` et `klein -3 -3` (qui de plus est sans K_3). Par défaut, les sommets sont dessinés selon une grille |p|×|q|.

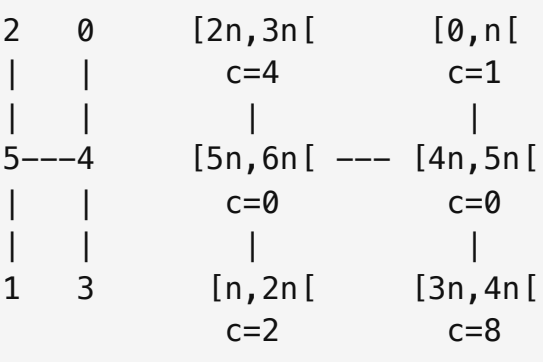
Ex : `gengraph klein -3 -4 -label -1 -xy noise .3 .7 -dot len 1 -visu`

`flower_snark n`

Graphe cubique à 4n sommets construit de la manière suivante : 1) on part de n étoiles disjointes à 3 feuilles, la i-ème ayant pour feuilles les sommets notés u_i,v_i,w_i, i=1…n ; 2) pour chaque x∈{u,v,w}, x_1-⋯-x_n induit un chemin ; et enfin 3) sont adjacents : u_0-u_n, v_0-w_n et w_0-v_n. Pour n>1, ces graphes sont non planaires, non hamiltoniens, 3-coloriables et de maille au plus 6. Pour n=1, il s'agit d'un $K_{1,3}$ (`claw`).

`biggs n`

Graphe de Biggs-Smith généralisé. Il est cubique sauf pour n=1,2,4,8,16 où son degré maximum est 3. Il possède 6n sommets partitionnés en six blocs de n sommets de numéros consécutifs. Les blocs sont connectés comme le graphe H selon un produit cartésien : les sommets homologues dans chaque bloc (de même modulo n) sont adjacents. Chaque bloc (0…5) induit un anneau de cordes de longueur c=0,1,2,4,8 réparties comme ceci :



Donc deux sommets de deux blocs différents sont adjacents si leurs blocs sont adjacents dans H et s'ils ont le même modulo n (sommets homologues). Et deux sommets dans un même bloc sont adjacents si leur différence modulo n vaut c. Sinon, ils ne sont pas adjacents.

`udg n r`

Graphe géométrique aléatoire (random geometric graph) sur n points du carré $[0, 1]^2$ (distribution par défaut). Deux sommets sont adjacents si leurs points sont à distance ≤ r. Il s'agit de la distance selon la norme L2 (par défaut), mais cela peut être changé par l'option `-norm`. Le graphe devient connexe avec grande probabilité lorsque r=rc ~ √(ln(n)/n). Si r<0, alors le rayon est initialisé à rc. Un UDG (unit disk graph) est normalement un graphe d'intersection de disques fermés de rayon 1.

`gabriel n`

Graphe de Gabriel. Graphe géométrique défini à partir d'un ensemble de n points du carré $[0, 1]^2$ (distribution par défaut). Les points i et j sont adjacents ssi le plus petit disque (voir `-norm`) passant par i et j ne contient aucun autre point. Ce graphe est connexe et planaire. C'est un sous-graphe du graphe de Delaunay. Son étirement est non borné.

`rng n`

Graphe du proche voisinage (Relative Neighborhood Graph). Graphe géométrique défini à partir d'un ensemble de n points du carré $[0, 1]^2$. Les points i et j sont adjacents ssi il n'existe aucun point k tel que $\max\{d(k,i),d(k,j)\} < d(i,j)$ où d est la distance (L2 par défaut, voir `-norm`). Dit autrement, la « lune » définie par i et j doit être vide. Ce graphe est planaire et connexe. C'est un sous-graphe du graphe `gabriel`.

`knng n k`

Graphe des k plus proches voisins (k-Nearest Neighbor Graph). Graphe géométrique défini à partir d'un ensemble de n points du carré $[0, 1]^2$ (distribution par défaut). Chaque point i est connecté aux k plus proches autres points (par défaut selon la norme L2, voir `-norm`). Si la norme est L2, le degré des sommets est ≤ 6k. Il peut comporter des croisements d'arêtes dès que k>1. Cependant, chaque arête ne peut être coupée que $O(k^2)$ fois. Plus précisément, c'est un t-planaire pour $t \leq 78k^2-6k$ (cf. [DMW19]). À partir de k=3 apparaît une composante connexe de taille linéaire.

`mst n`

Graphe aléatoire géométrique définissant un arbre couvrant de poids minimum du graphe complet euclidien sur n points tirés aléatoirement uniformément dans le carré $[0, 1]^2$ (distribution par défaut). Par défaut la distance est la norme L2 (voir `-norm`).

Ex : `gengraph mst 2000 -xy seed 1 .3 -visu`

`thetagone n p k w`

Graphe géométrique défini à partir d'un ensemble de n points du plan (par défaut uniformément dans le carré $[0,1]^2$). En général le graphe est planaire et connexe avec des faces internes de longueur au plus p (pour k diviseur de p et w=1). On peut interpréter les paramètres comme suit : $p \geq 3$ est le nombre de côtés d'un polygone régulier, $k \geq 1$ le nombre d'axes (ou de directions), et $w \in [0,1]$ le cône de visibilité. Toute valeur de $p<3$ est interprétée comme une valeur infinie, et le polygone régulier correspondant interprété comme un cercle. L'adjacence entre une paire de sommets est déterminée en temps O(kn).

Plus formellement, pour tous points u et v, et tout entier i, on note P_i(u,v) le plus petit p-gone (polygone convexe régulier à p côtés) passant par u et v dont u est un sommet, et dont le vecteur allant de u vers son centre forme un angle de $i \cdot 2\pi/k$ avec l'axe des abscisses, intersecté avec un cône de sommet u et d'angle $w \cdot (p-2) \cdot \pi/p$ ($w \cdot \pi$ si p est infini) et dont la bissectrice passe par le centre du p-gone. Alors, u est voisin de v s'il un existe au moins un entier $i \in [0,k[$ tel que l'intérieur de P_i(u,v) est vide. La distance entre u et le centre du p-gone définit alors une distance (non symétrique) de u à v.

Si w=1 (visibilité maximale), P_i est précisément un p-gone. Si w=0 (visibilité minimale), P_i se réduit à l'axe d'angle $i \cdot 2\pi/k$ pour un entier i. Si w=.5, P_i est un cône formant un angle égale à 50 % de l'angle défini par deux côtés consécutifs du p-gone, ce dernier angle valant $(p-2)\pi/p$. Si $w=2p/((p-2)k)$ (ou simplement 2/k si p est infini) alors la visibilité correspond à un cône d'angle $2\pi/k$, l'angle entre deux axes. Comme il faut $w \leq 1$, cela implique que $k \geq 2p/(p-2)$ ($k \geq 2$ si p infini). On retrouve le Theta_k-Graph pour chaque $k \geq 6$ en prenant p=3 et w=6/k, le demi-Theta-Graph pour tout $k \geq 3$ en prenant p=3 et w=3/k, le Yao_k-Graph pour chaque $k \geq 2$ en prenant p=0 (infini) et w=2/k, et la triangulation de Delaunay (`td-delaunay`) si p=0 (infini), k très grand et w=1. En fait, ce n'est pas tout à fait le graphe Yao_k, pour cela il faudrait que u soit le centre du polygone (c'est-à-dire du cercle).

`pat p q r`

Graphe possédant pqr sommets, issu d'un jeu à un joueur proposé par Pat Morin (Barbade, mars 2016). Le jeu se déroule sur une grille p×q et comprend r coups. Un coup est un ensemble de positions de la grille strictement croissantes (coordonnées en x et en y strictement croissantes). De plus, si la position (x,y) est jouée alors toutes les positions situées sur la même ligne mais avec une abscisse au moins x ou sur la même colonne mais avec une ordonnées au moins y sont interdites pour tous les coups suivants. Le score est le nombre total de positions jouées en r coups. Il s'agit de trouver le score maximum. Lorsque r=1, le score maximum vaut min(p,q). Lorsque p=q=n et r=2, alors le score maximum vaut $\lfloor 4n/3 \rfloor$. La question est ouverte lorsque r>2, c'est au moins n^1.516 pour r=n où la constante vaut log_9(28).

Les sommets du graphes sont les positions dans les r grilles p×q et deux sommets sont adjacents les positions sont en conflits. Le score du jeu est alors un ensemble indépendant du graphe. Si r=1, le graphe est une grille p×q. Ce graphe est géométrique : un dessin de ce graphe (sous forme de grilles) est proposé.

Ex : `gengraph pat 4 4 4 -check kindepsat 8 | glucose -model`

`line-graph n k`

Graphe ligne aléatoire à n sommets et de paramètre k>0 entier. Plus k est petit, plus le graphe est dense, le nombre d'arêtes étant proportionnel à $(n/k)^2$. Si k=1, il s'agit d'une clique avec boucles et doubles arêtes. Ces graphes sont obtenus en choisissant, pour chaque sommet, deux couleurs de [0,k[. Deux sommets sont adjacents ssi ils possèdent la même couleur. Il contient le graphe `uno n k k` comme sous-graphe. Ces graphes sont sans griffe (c'est-à-dire sans `claw` induit). Tout graphe ligne est claw-free, et les graphes lignes connexes avec un nombre pair de sommets possèdent toujours un couplage parfait.

Si un sommet a deux fois la même couleur, alors il doit avoir une boucle, c'est pourquoi `-loop` est activé par défaut pour ce graphe. Si deux sommets ont la même paire de couleurs, alors ils sont reliés par deux arêtes, ce que l'on ne peut voir qu'avec `-fast`. Ces cas de sommets peuvent empêcher les graphes d'être inversés avec `-op line-graph-root`.

On rappelle qu'un graphe G est le graphe ligne d'un graphe H si les sommets de G correspondent aux arêtes de H et où deux sommets de G sont adjacents ssi les arêtes correspondantes dans H sont incidentes. On parle parfois de graphe adjoint.

Il existe une variante : `-variant 1`, calculant le graphe ligne d'un graphe orienté, option qui devrait être combinée avec `-directed`. Voir `-op line-graph` pour la définition du graphe ligne d'un graphe orienté.

`uno n p q`

Graphe ligne aléatoire à n sommets issu d'un graphe biparti de parts de taille p>0 et q>0. Plus précisément, les sommets sont des paires (i,j) d'entiers aléatoires de [0,p[× [0,q[, pas nécessairement distinctes. Les sommets (i,j) et (i',j') sont adjacents ssi i=i' ou j=j'. C'est un sous-graphe induit du produit cartésien de deux cliques, K_p × K_q. Ce graphe est géométrique, les sommets étant des points de la grille p×q. Les sommets représentent aussi des cartes du jeu de UNO et les arêtes indiquent si un carte peut être jouée consécutivement à une autre. Le graphe `uno n k k` est un sous-graphe de `line-graph n k`.

`unok n p q k_p k_q`

Graphe `uno n p q` particulier où les n points correspondant aux sommets sont pris uniformément parmi les ensembles de n points distincts de [0,p[× [0,q[ayant au plus k_p sommets par ligne et k_q par colonne. Il faut n ≤ min{p·k_p,q·k_q} et p, q, k_p, k_q>0. Si k_p<0, alors on fait comme si k_p=p, de même pour k_q=q si k_q<0. Contrairement à `uno`, deux sommets ont toujours des coordonnées distinctes. Le graphe résultant est de degré au plus k_p+k_q-2, et est de path-width (et aussi de tree-width) au plus celle du produit de `clique` K_{k_p} × K_{k_q} soit environ k_p·k_q/2. Le temps de génération des n points est en O(npq) contre O(n) pour uno, mais une optimisation (algorithme par rejets) fait qu'il est

très souvent en O(n+p+q), dans les cas peu dense par exemple. Si k_p ou k_q=1, le graphe est une union de cliques, et si k_p=k_q=2 et n=2p=2q, c'est une union de cycles.

Ex : `gengraph unok 200 100 100 3 2 -visu`

wpsl n p q
upsl n p q
wpsld n p q
upsld n p q

Weighted/Uniform Planar Stochastic Lattice. Graphe planaire aléatoire connexe dont les sommets correspondent à certains points d'une grille p×q (p colonne et q lignes) et les arêtes à des lignes horizontales ou verticales. Il est généré selon un processus en n≥0 étapes décrit dans un paragraphe ci-après. Il faut p,q≥2. Il comprend au plus 3n+1 faces internes rectangulaires, appelées blocs, qui forment une partition des cases de la grille p×q. C'est un graphe 2-dégénéré qui possède au plus min{5n+4,p·q} sommets dont 4 sont de degré 2, les autres étant de degré 3 ou 4 selon une répartition moyenne 80 %–20 %. La variante `wpsld` (ou `upsld`) représente le graphe dual qui est 4-dégénéré et connexe. Les sommets (au plus 3n+1) correspondent aux blocs (positionnés en leurs centres), deux blocs étant adjacents s'ils ont un bord en commun. Pour le dual, les coordonnées des centres des blocs sont doublées pour être entières, et le dessin n'est pas forcément planaire. L'option `-directed` permet d'obtenir une 2-orientation et une 4-orientation pour le dual.

Ex :
```
gengraph wpsl 100 500 400 -seed 0 -view scale auto -visu
gengraph wpsld 100 500 400 -seed 0 -view scale auto -visu
gengraph wpsl 2500 70000 70000 -view scale auto -visu
gengraph wpsl 2 10 10 -view scale auto -xy grid 10 -visu
```

Le graphe est construit en n étapes. Au départ il y a un seul bloc contenant toutes les cases d'une grille p×q. Le bord de ce bloc correspond aux 4 coins de la grille et forme un cycle de longueur 4. À chacune des n étapes, on sélectionne un bloc B parmi ceux déjà construits selon une probabilité proportionnelle de sa surface (pour `wpsl`) ou uniformément parmi tous les blocs (pour `upsl`). Ici la surface d'un bloc est le nombre de cases — et non de points — de la grille qu'il contient. Puis B est découpé selon une croix dont le centre est un point de la grille interne de B (pas sur un bord) choisi aléatoirement uniformément. (La variante consistant à découper un bloc en deux correspond au graphe `wdis` et ses variantes.) Le bloc n'est pas découpé s'il ne contient pas de points de la grille. Une autre façon de concevoir le processus pour `wpsl` est de choisir n points de la grille p×q aléatoirement et uniformément. Puis, depuis chaque point et dans un ordre quelconque, faire pousser une croix jusqu'à atteindre un bord ou une croix précédente, les points se trouvant sur le passage d'une croix étant supprimés. La création du graphe prend un temps O(n log n) en moyenne, indépendant des dimensions p et q qui peuvent donc être relativement grandes. Ensuite, le test d'adjacence, lié à sa k-orientation, est constant.

`wdis n p q`
`udis n p q`
`wdisd n p q`
`udisd n p q`

Rectangular Dissection. Graphe planaire aléatoire connexe dont les sommets correspondent à certains points d'une grille p×q (p colonne et q lignes) et les arêtes à des lignes horizontales ou verticales. Il est généré selon un processus en n≥0 étapes similaires à `wpsl` (et ses variantes). Les variantes `wdisd` et `udisd` correspondent au graphe dual. À chaque étape du processus, on sélectionne aléatoirement un bloc, soit proportionnellement à sa surface (`wdis`) soit uniformément (`udis`), que l'on coupe en deux sous-blocs. Le sens de la découpe (verticalement ou horizontalement) est soit aléatoire uniforme (`udis`) soit selon une probabilité proportionnelle à la longueur des côtés (`wdis`), préférant découper le plus grand côté. Il possède au plus min{2n+4,p·q} sommets : 4 de degré 2, presque tous les autres de degré 3 sauf quelques-uns de degré 4. Le dual possède n+1 sommets. Il partage un grand nombre de propriétés communes avec `wpsl`, en particulier l'orientation. Tous les graphes `wpsl` peuvent être générés par un `wdis`.

`ngon p c x`

Triangulation particulière d'un polygone régulier. Plusieurs types de triangulations sont produites suivant la valeur des paramètres. Elles ont p, 3p ou 4p sommets. Il y a trois cas selon la valeur de x.

- Le premier cas est x≥0. Si x&4=0, alors la triangulation a 3p sommets (avec p>0) et est composée d'un triangle équilatéral central. Si x&4≠0, alors la triangulation a 4p sommets et est composée d'un carré central avec une diagonale. La triangulation est symétrique dans chacun des 3 ou 4 croissants délimités par chacune des arêtes du polygone central. Si AB est l'une de ces arêtes (A avant B dans le sens direct), alors on note C le point de l'arc de cercle de A à B à distance c de A. On doit avoir c∈[0,p/2]. Les deux bits de poids faible de x définissent comment sont construites les triangulations de l'arc AC et CB. Tous les points de AC sont connectés à A si x&1≠0 et à C sinon. Et, tous les points de BC sont connectés à B si x&2≠0 et à C sinon. Si x&8≠0 alors la triangulation est asymétrique. On remplace c par p-c pour un arc AB sur deux.

- Si x=-1, alors il s'agit d'une autre triangulation. Elle a p sommets et est symétrique par rapport à un axe horizontal comprenant trois `fan` : un depuis le point 0 vers tous ceux de [c,n-c], un depuis c vers tous ceux de [0,c], et enfin un depuis n-c vers tous ceux de [n-c,n].

- Si x=-2, alors il s'agit de la triangulation récursive à 3p sommets. Le paramètre c n'a pas de rôle. Chacun des trois arcs est coupé en deux récursivement. Si p n'est pas une puissance de deux, alors le graphe peut ne pas être une triangulation complète, mais le graphe reste cependant planaire.

La triangulation qui expérimentalement minimise le stretch maximum (voir `-check stretch`) est obtenue avec `ngon p αp 3` où α = 231/512 ≃ 45 %. Le stretch maximum est environ 1.455 réalisé entre les

sommets u=p+30 % et v=3p-20 %.

behrend p k

Graphe régulier de p·k sommets possédant un très grand nombre de cycles de longueur k arête-disjoints où p,k ≥ 2. Si p est premier, il en possède exactement p·c! = p^{2-o(1)} où c ~ log(p)/loglog(p). Son degré est 2c! si k>2 ou c! si k=2. Le graphe est défini que p soit premier ou pas. Il est construit à partir de k stables S_0,...,S_{k-1} de chacun p sommets. Chacun des cycles de longueur k contient exactement un élément de chaque S_j qui est le sommet d'indice i+j·x (mod p) dans S_j avec i∈[0,p[et x∈X où X⊂[0,p[est un ensemble où k entiers quelconques ne sont jamais en progression arithmétique. On construit X comme l'ensemble de tous les entiers < p/(k-1) s'écrivant sur c chiffres distincts pris dans [0,c[en base ck+1 avec c maximum. Donc |X|=c!. Lorsque p est petit, le graphe peut ne pas être connexe.

Par exemple, pour p=421 et k=3 on obtient c=3 et X = { 012, 021, 102, 120, 201, 210 } (nombres écrits en base ck+1=10). On vérifie qu'on a bien p > (k-1)·max{X} = 420. Ce graphe et donc 12-régulier possède p·k = 1263 sommets et p·c! = 5052 triangles arête-disjoints car 421 est premier. La table ci-dessous donne en fonction de k et du degré souhaité la plus petite valeur de p=p(k) possible. Si p est plus petit que p(k), alors le degré sera moindre. Lorsque k=2, le degré est c! au lieu de 2·c!.

	2·2!	2·3!	2·4!	2·5!
degré	4	12	48	240
p(2)	6	106	2,359	62,811
p(3)	15	421	13,885	549,921
p(4)	28	1,054	46,003	2,419,831

Ces graphes sont utilisés en « property testing » pour montrer qu'il est difficile de déterminer si un graphe dense possède ou pas un cycle de longueur k.

rplg n t

Random Power-Law Graph. Graphe aléatoire à n sommets où les degrés des sommets suivent une loi de puissance d'exposant t>1 (typiquement un réel t∈]2,3[). L'espérance du degré du sommet i=0...n-1 est w_i = (n/(i+1))^(1/(t-1)). La probabilité d'avoir l'arête i-j est min{w_i·w_j/S,1} avec S=∑_k w_k. La valeur communément observée pour le réseau Internet étant t=2.1.

bdrg n_1 d_1 … n_k d_k .

Bounded Degree Random Graph. Graphe aléatoire dont la distribution des degrés des sommets est fixée par les paires (n_i,d_i) signifiant qu'il y a n_i sommets de degré au plus d_i. Ainsi `bdrg n 3 .` génère un graphe sous-cubique aléatoire à n sommets, si n est pair. Les sommets sont dupliqués selon leur

distribution de degré puis un couplage aléatoire détermine les arêtes. Les boucles et les arêtes multiples sont supprimées. Il suit que le degré des sommets ne dépasse pas d_i. Il peut cependant être inférieur. Le nombre de sommets est $n=\sum_i n_i$ et le nombre d'arêtes au plus $m = ½\sum_i (n_i \cdot d_i)$. Si cette somme n'est pas entière, alors le degré d'un des sommets ayant $d_i>0$ est diminué d'un. (C'est un sommet avec $d_i>0$ avec le plus grand i qui est choisi.)

`fdrg n_1 d_1 … n_k d_k .`

Fixed Degree Random Graph. Graphe aléatoire asymptotiquement uniforme dont les degrés des sommets sont fixés par les paires (n_i,d_i) signifiant qu'il y a n_i sommets de degré d_i. La suite des degrés doit être graphique, à savoir qu'il existe au moins un graphe simple ayant ces degrés (sinon une erreur est affichée). Ainsi `fdrg n 3 .` génère un graphe cubique aléatoire asymptotiquement uniforme, à condition que n soit pair. Il est possible d'obtenir des graphes non connexes, comme avec `fdrg 3 2 1 0 .` composé d'un triangle et d'un sommet isolé. La complexité est en moyenne $O(m\Delta+\Delta^4)$ où $m=\sum n_i d_i$ et $\Delta=\max\{d_i\}$, et pour être asymptotiquement uniforme, il faut $\Delta=o(m^{1/4})$ ou $\Delta=o(\sqrt{n})$ pour les graphes réguliers (tous les d_i égaux ou k=1).

`caterpillar n`

Graphe chenille à `n` sommets. Il s'agit d'un arbre dont les sommets internes (de degré > 1) induisent un chemin.

Pour le générer uniformément, on génère un bit aléatoire pour chaque sommet. Un bit à 1 signifie que le sommet est le prochain sommet interne, au centre d'une étoile ; un bit à 0 signifie que le sommet est sur l'étoile du sommet interne précédent. Le premier sommet a forcément son bit à 1 ; le deuxième et le dernier sommets ont forcément leur bit à 0. Chaque sous-liste de `n`-3 bits fournissant le même graphe que sa forme retournée, les sous-listes lexicographiquement supérieures à leur forme retournée sont rejetées (afin d'éviter que ces graphes soient générés plus souvent que ceux des sous-listes qui sont leur propre forme retournée).

Par exemple, la suite de bits ci-dessous correspond aux adjacences suivantes pour un `caterpillar 10` (seuls les bits soulignés sont aléatoires) :

```
1001110010

0-1 0-2 0-3-4-5-6 5-7 5-8-9
```

Il y a $2^{(n-3)}$ listes de bits possibles, et $2^{\lceil (n-3)/2 \rceil}$ sous-listes qui restent identiques si on les retourne, donc au total $(2^{n-3}-2^{\lceil (n-3)/2 \rceil})/2+2^{\lceil (n-3)/2 \rceil} = 2^{n-4}+2^{\lfloor n/2-2 \rfloor}$ graphes possibles pour `n`>3.

L'option `-directed` permet d'obtenir une 1-orientation.

GRAPHES ORIENTÉS

`aqua c_1 … c_n .`

Graphe orienté dont les sommets sont les suites de n entiers positifs dont la somme fait c_1 et dont le i-ème élément est au plus c_i. Ils représentent les façons de répartir une quantité c_1 de liquide dans n récipients de capacité c_1 ... c_n. Il y a un arc u->v s'il existe i et j tels que v est le résultat du versement du récipient c_i vers le récipient c_j. Le graphe est isomorphe au graphe où les c_i=0 ont été supprimés, les c_i ont été triés par ordre décroissant et où c_1 a été remplacé par min{c_1,c_2+...+c_n}. Par exemple, `aqua 4 1 0 2 .` est isomorphe à `aqua 3 2 1 .`. Le nombre de sommets ne peut pas dépasser binom{n+c_1}{n}. Le graphe peut être connexe mais non fortement connexe comme `aqua 2 2 .`.

Ex : `gengraph aqua 3 2 1 . -label 1 -dot filter dot -visu`

`collatz n a_0 b_0 … a_{k-1} b_{k-1} .`

Graphe de Collatz généralisé. Il est basé sur la relation C : $x \mapsto (a_i \cdot x + b_i)/k$, définie pour tout entiers x>0, où $i = x\%k$ et où a_i,b_i sont entiers (pas forcément positifs). Il faut $a_i \cdot i + b_i \equiv 0 \pmod k$ pour tout i pour que C(x) soit entier, sinon $\lfloor C(x) \rfloor$ est considérée et C^{-1} n'est plus forcément injective. Les entiers générés forment les sommets du graphe, les arcs étant les relations $x \rightarrow C(x)$.

Le graphe pour le problème « 3x+1 », défini par la relation $x \mapsto x/2$ si $x \equiv 0 \pmod 2$ et $x \mapsto x/2$ si $x \equiv 1 \pmod 2$, est donc le graphe `collatz n 1 0 3 1 .`.

Si n>0, la boule de volume n est générée en itérant la relation inverse depuis x=1, soit C^{-1}: $x \mapsto (k \cdot x - b_i)/a_i$ pour chaque i tel que a_i divise $k \cdot x - b_i$. Il est important que C(x) soit entier pour que C^{-1} soit injective. Si n<0, la relation est itérée depuis chaque entier $x \in [1, |n|]$. Dans le cas n>0, le graphe est connexe et comprend au plus n sommets, alors que pour n<0, il peut ne pas être connexe et contenir plus de |n| sommets. Il s'agit de deux sous-graphes (n>0 et n<0) induit du même graphe infini. Dans tous les cas c'est un graphe orienté avec au plus un successeur (arc sortant) et k prédécesseurs (arcs entrant), et peut contenir des cycles dont des boucles et des arcs symétriques.

Ex : `gengraph collatz -65 1 0 5 -1 5 1 3 1 . -label 1 -visu`

La fameuse conjecture de Collatz pour le problème « 3x+1 » affirme que le graphe `collatz n 1 0 3 1 .` est connexe quel que soit n<0 (voir aussi `syracuse n`). Pour de grandes familles de coefficients a_i,b_i, il est conjecturé que le graphe possède un nombre constant de composantes connexes, comme par exemple 3 pour `collatz n 1 0 5 1 .`. Savoir si le graphe généralisé de Collatz est connexe est indécidable, même si tous les b_i=0 [Conway'72].

La génération du graphe proprement dite est basée sur une file initialisée aux valeurs 1..|n| pour lesquelles la relation C est successivement appliquée (si n<0). Il s'agit donc d'une sorte de parcours en largeur du graphe depuis n sources en parallèle. La construction est limitée arbitrairement à n^2 sommets car, suivant les coefficients a_i,b_i, ce parcours peut ne pas converger pour certains $x \in [1, |n|]$. Attention :

les valeurs C(x) négatives sont exclues du graphe, ce qui peut aussi exclure les valeurs dépassant 2^{31} et donc déconnecter le graphe. Si n>0, on initialise la file avec seulement la valeur 1 et on itère les k relations inverses, si elles s'appliquent, jusqu'à produire exactement n sommets. Il s'agit donc d'un simple parcours en largeur depuis 1. Bien que le nombre d'arcs soit linéaire, la génération de tous les successeurs prend un temps quadratique en le nombre de sommets final du graphe si n<0, et $O(kn^2)$ si n>0.

GRAPHES COMPOSÉS

Les graphes de cette section n'ont pas leur propre fonction d'adjacence : leur invocation construit une requête pour un autre graphe, avec les paramètres indiqués entre parenthèses.

`mesh p q (= grid p q .)`

Grille 2D de pq sommets qui sont [0,p[×[0,q[.

`hypercube d (= grid 2 … 2 .)`

Hypercube de dimension d.

`path n (= grid n .)`

Chemin à n sommets.

`cycle n (= ring n 1 .)`

Cycle à n sommets C_{n}.

`triangle (= ring 3 1 .)`

Graphe complet K_3 (`complete`), `cycle` C_3.

`torus p q (= grid -p -q .)`

Tore à pq sommets qui sont [0,p[×[0,q[.

`null (= ring 0 .)`

Graphe nul : aucun sommet.

stable n *(= ring n .)*
empty n *(= ring n .)*

Stable, graphe vide à n sommets.

clique n *(= ring n . -op not)*
complete n *(= ring n . -op not)*

Graphe complet à n sommets K_{n}, c'est-à-dire dont tous les sommets sont universels, connectés deux à deux.

tournament n *(= ring n . -op not -op half -directed -noloop)*

Tournoi : graphe complet orienté à n sommets.

matching n *(= ring n . -op star -1)*

Couplage de taille `n` : graphe composé de `n` arêtes indépendantes, c'est-à-dire de `n` copies de K_2. L'option `-directed` permet d'obtenir une 1-orientation.

fan p q *(= grid p . -op apex q)*

Graphe de p+q sommets : un chemin de p sommets, tous connectés à chacun des q autres sommets. Le graphe classique « fan n » correspond à `fan n 1`.

windmill n *(= ring n . -op star -1 -op apex 1)*

Graphe composé de `n` graphes `triangle` ayant un sommet commun.

split n k *(= ring n-k . -op apex -k)*

Graphe scindé (dit aussi séparé, fendu ou *split*) complet à n sommets et de clique maximum k. Il s'agit d'un graphe à n sommets composé d'une `clique` à k sommets et d'un ensemble indépendant de n-k sommets connectés chacun à tous ceux de la clique. C'est un graphe triangulé (chordal) et un cas particulier de `kstar n k`. On peut montrer que presque tous les graphes triangulés à n sommets sont des graphes scindés (Bender et al. 1985), par forcément complets. Si k≥n-1, alors il s'agit d'une clique, et si k=n-2, il s'agit d'une clique moins une arête. Si k=1 il s'agit d'une étoile à n-1 branches.

comb n *(= grid n . -op star -1)*
centipede n *(= grid n . -op star -1)*

Arbre de 2n sommets en forme de peigne, composé d'un chemin à n sommets avec un sommet pendant à chacun d'eux. On peut l'obtenir en supprimant une arête d'un `sunlet n`.

`sunlet n (= grid -n . -op star -1)`

Cycle à n sommets avec un sommet pendant à chacun d'eux. Un `sunlet 3` est parfois appelé `net-graph`.

`bipartite p q (= rpartite p q .)`

Graphe biparti complet K_{p,q}.

`utility (= rpartite 3 3 .)`

Graphe biparti complet $K_{3,3}$ qui doit son nom au problème de la connexion planaire de trois maisons à trois stations (eau, gaz, électricité). C'est aussi le graphe `haar 7`.

`domino (= grid 2 3 .)`

Graphe planaire à 6 sommets composé de deux carrés partageant une arête.

`kite (= banana 1 3 -op not)`

Kite Graph, graphe à 5 sommets composé de deux triangles partageant une arête et d'un sommet pendant (degré 1) attaché à un sommet de degré deux. C'est aussi un `parachute 2`.

`dart (= sum-graph 5)`

Dart Graph, graphe à 5 sommets composé de deux triangles partageant une arête et d'un sommet pendant (degré 1) attaché à un sommet de degré trois. Il peut être obtenu aussi à partir du graphe `moth` en supprimant un sommet pendant.

`parapluie n (= parachute n -op not)`

Graphe planaire à n+3 sommets, complémentaire du parachute. Le graphe parapluie classique correspond à n=4.

`hourglass (= barbell 3 3 0)`

Graphe sablier (*hourglass*). Il a 5 sommets et est composé de deux triangles partageant un sommet. Il est semblable à `butterfly 1`, mais ce dernier correspond en fait à `cycle 4`.

`cuboctahedron` *(= `linial 4 2`)*

Cuboctaèdre : graphe planaire 4-régulier à 12 sommets. Il possède 24 arêtes et 14 faces qui sont des triangles ou des carrés. C'est le dual du rhombic-dodécaèdre (`rdodecahedron`).

`octahedron` *(= `antiprism 3`)*

Octaèdre : graphe 4-régulier planaire à 6 sommets ayant 8 faces triangulaires. Il s'agit de deux pyramides dont la base à 4 sommets est commune. C'est aussi le graphe `johnson 4 2`.

`d-octahedron d` *(= `ring d . -op star -1 -op not`)*

Octaèdre de dimension d : obtenu à partir d'un octaèdre de dimension d-1 auquel on ajoute deux sommets universels, l'octaèdre de dimension 1 étant composé d'un `stable` de deux sommets. L'`octahedron` classique est obtenu avec d=3, pour d=2 il s'agit d'un carré.

`tetrahedron` *(= `ring 4 . -op not`)*

Tétraèdre : pyramide composée de 4 faces triangulaires. C'est aussi une `clique` à 4 sommets.

`cube` *(= `crown 4`)*
`hexahedron` *(= `crown 4`)*

Hypercube de dimension 3 (`hypercube 3`), graphe planaire cubique à 8 sommets où toutes les faces sont des rectangles. C'est aussi un hexaèdre (6 faces carrées) ou encore le graphe `haar 11`.

`associahedron` *(= `flip 6`)*

Associaèdre (3D) : graphe planaire cubique à 14 sommets composé de 3 faces carrées et 6 faces pentagonales.

`johnson n k` *(= `kneser n k k-2 -op not, si k≥2`)*
`johnson n k` *(= `kneser n n-1 0 -op not, si k=1`)*

Graphe de Johnson J(n,k). Les sommets sont tous les sous-ensembles à k éléments de {1,...,n}. Deux sommets sont adjacents ssi leurs sous-ensembles ont k-1 éléments en commun. Il faut n≥k≥1. La distance entre deux sommets est donnée par la distance de Hamming entre les ensembles correspondant. Ils sont réguliers de degré k(n-k), de diamètre min{k,n-k}, de sommet-connectivité k(n-k). Ils sont aussi distance-réguliers. J(n,1) est K_n, J(n,2) est le complément du graphe de Kneser K(n,2) et aussi le graphe ligne de K_n. En fait, tout sous-graphe induit de J(n,2) est un graphe ligne. J(4,2) est l'octaèdre, J(5,2) le complément du graphe `petersen`.

`odd n (= kneser 2n-1 n-1 0)`

Odd graph de paramètre n>0. Il s'agit du graphe de Kneser K(2n-1,n-1) dont les sommets correspondent aux binom(2n-1,n-1) sous-ensembles à n-1 éléments de {1,...,2n-1} et sont adjacents s'ils ont au plus n-1 éléments en commun. Le plus petit cycle impair est toujours de longueur 2n-1. Il est n-régulier et son diamètre est n-1. Il est distance-transitif (automorphisme entre paires de sommets à même distance) et donc distance-régulier. Si n>3, il est hamiltonien et de maille 6. Il n'est jamais un graphe de Cayley si n>2. Pour n=2, il s'agit de K_3 et pour n=3 du graphe `petersen` (et donc non hamiltonien).

`biggs-smith (= biggs 17)`

Graphe de Biggs-Smith. Il est cubique, possède 102 sommets, et est distance-régulier. Il n'existe que 13 graphes cubiques distance-réguliers. De nombre chromatique 3, son diamètre est 7, comme son rayon, alors que sa maille est 9.

`claw (= rpartite 1 3 .)`

Graphe biparti complet $K_{1,3}$ dit griffe. C'est aussi une étoile à 3 branches (`star 3`).

`star n (= rpartite 1 n .)`

Étoile à `n` branches : arbre à `n` feuilles et de hauteur 1.

Voir aussi `-op star`.

`tree n (= arboricity n 1)`

Arbre plan enraciné aléatoire uniforme à n sommets. Les sommets sont numérotés selon un parcours en profondeur depuis la racine et le long de la face extérieure.

`outerplanar n (= kpage n 1)`

Graphe planaire-extérieur aléatoire connexe à n sommets (plan et enraciné). Ils sont en bijection avec les arbres plans enracinés dont tous les sommets, sauf ceux de la dernière branche, sont bicoloriés. Les sommets sont numérotés le long de la face extérieure. C'est aussi une numérotation selon un parcours en profondeur depuis la racine de l'arbre bicolorié. Il est aussi possible de générer des graphes planaires-extérieurs aléatoires hamiltoniens, donc 2-connexes, avec `planar n f -1` ou `polygon n`. L'option `-directed` permet d'obtenir une 2-orientation.

`squaregraph n (= planar n 4 4)`

Squaregraph aléatoire à n faces. Ce sont des graphes planaires 2-connexes dont toutes les faces (sauf l'extérieure) sont des carrés. De plus, les sommets des faces internes sont de degré au moins 4. Ce sont des sous-graphes de quadrangulations et donc des 2-pages. Ce sont éqalement des graphes médians : pour chaque triplet de sommets {u,v,w}, il existe un seul sommet, le médian, appartenant à un plus court chemin u-v, de v-w et de u-w. L'option `-directed` permet d'obtenir une 2-orientation.

`random n p (= ring n . -op not -op del-e 1-p)`

Graphe aléatoire à n sommets et dont la probabilité d'avoir une arête entre chaque paire de sommets `i-j` est p.

`netgraph (= sierpinski 2 3 -op not)`

Graphe à 6 sommets composé d'un triangle avec un sommet pendant à chacun d'eux. C'est le complémentaire du graphe `hajos`. On peut aussi le générer en utilisant `fdrg 3 3 3 1 .`.

`sunflower n (= cage 2n 2 2 .)`

Tournesol à n pétales. C'est un graphe planaire-extérieur à 2n sommets composé d'un `cycle` de longueur n≥3 où chaque arête partage le côté d'un `triangle`. C'est le graphe `gear n` sans le sommet central. Pour n=3, c'est le graphe `hajos`.

`gem (= grid 4 . -op apex 1)`

Graphe à 5 sommets composé d'un chemin et d'un sommet universel.

`egraph (= grid 3 . -op star -1)`

Arbre à 6 sommets et 3 feuilles en forme de E.

`tgraph (= banana 1 3)`
`fork (= banana 1 3)`

Fork Graph, arbre en forme de T à 5 sommets dont 3 feuilles.

`ygraph (= banana 3 1)`

Arbre à 7 sommets composé d'une étoile à trois branches.

`cross (= banana 1 4)`

Cross Graph, arbre à six sommets en forme de croix chrétienne.

`knight p q (= chess p q 1 2)`

Graphe des déplacements du chevalier dans un échiquier p×q.

`antelope p q (= chess p q 3 4)`

Graphe des déplacements d'une antilope dans un échiquier p×q, une antilope étant une pièce hypothétique se déplaçant de 3 cases selon un axe et de 4 selon l'autre.

`camel p q (= chess p q 1 3)`

Graphe des déplacements d'un chameau dans un échiquier p×q, un chameau étant une pièce hypothétique se déplaçant de 1 case selon un axe et 3 de selon l'autre.

`giraffe p q (= chess p q 1 4)`

Graphe des déplacements d'une girafe dans un échiquier p×q, une girafe étant une pièce hypothétique se déplaçant de 1 case selon un axe et de 4 selon l'autre.

`zebra p q (= chess p q 2 3)`

Graphe des déplacements d'un zèbre dans un échiquier p×q, un zèbre étant une pièce hypothétique se déplaçant de 2 cases selon un axe et de 3 selon l'autre.

`king p q (= udg pq 1 -norm Lmax -xy mesh-x p)`

Graphe des déplacements d'un roi dans un échiquier p×q. Il s'agit d'une grille p×q avec les diagonales.

`petersen (= kneser 5 2 0)`

Graphe de Petersen. C'est un graphe de Kneser particulier. Il est cubique et possède 10 sommets. Il n'est pas hamiltonien et c'est le plus petit graphe dont le nombre de croisements (crossing number) est 2. C'est le complément du graphe ligne de K_5. Voir aussi `gpetersen`.

`tietze (= flower_snark 3)`

Graphe de Tietze. Il est cubique avec 12 sommets. Il possède un chemin hamiltonien, mais pas de cycle. Il peut être plongé sur un ruban de Möbius, a un diamètre et une maille de 3. Il peut être obtenu à partir du graphe `petersen` en appliquant une opération Y-Delta.

`mobius-kantor` *(= gpetersen 8 3)*

Graphe de Möbius-Kantor. Graphe cubique à 16 sommets de genre 1. Il est hamiltonien, de diamètre 4 et de maille 6. C'est aussi le graphe `haar 133`.

`dodecahedron` *(= gpetersen 10 2)*

Dodécaèdre : graphe planaire cubique à 20 sommets. Il possède 30 arêtes et 12 faces qui sont des pentagones. C'est le dual de l'icosaèdre (`icosahedron`).

`desargues` *(= gpetersen 10 3)*

Graphe de Desargues. Il est cubique à 20 sommets. Il est hamiltonien, de diamètre 5 et de maille 6.

`durer` *(= gpetersen 6 2)*

Graphe de Dürer. Graphe cubique planaire à 12 sommets de diamètre 4 et de maille 3. Il peut être vu comme un cube avec deux sommets opposés tronqués (remplacés par un cycle de longueur 3).

`prism n` *(= gpetersen n 1)*

Prisme, c'est-à-dire le produit cartésien d'un cycle à n sommets et d'un chemin à deux sommets. Pour n=3, c'est un graphe `halin` et aussi le complémentaire d'un `cycle 6`, et pour n=4 il s'agit du `cube`.

`cylinder p q` *(= grid p -q .)*

Produit cartésien d'un `path p` et d'un `cycle q`. Cela généralise le prisme (`prism n` = `cylinder n 3`). Un `cube` est un `cylinder 2 4`.

`nauru` *(= pstar 4)*

Graphe de Nauru. C'est un graphe cubique à 24 sommets. Il s'agit d'un graphe « permutation star » de dimension 4. C'est aussi un graphe `gpetersen 12 5`.

`heawood` *(= cage 14 5 -5 .)*

Graphe de Heawood. C'est un graphe cubique biparti à 14 sommets, de maille 6 et de diamètre 3. C'est le graphe d'incidence du plan projectif d'ordre 2 (plan de Fano). Il est 1-planar. C'est le plus petit graphe dont le nombre de croisements (crossing number) est 3. C'est aussi le graphe `haar 69`.

`franklin` *(= cage 12 5 -5 .)*

Graphe de Franklin. C'est un graphe cubique à 12 sommets, de maille 4 et de diamètre 3. C'est aussi le graphe `haar` 37.

`mcgee` *(= cage 24 12 7 -7 .)*

Graphe de McGee. C'est un graphe cubique à 24 sommets, de maille 7 et de diamètre 4.

`bidiakis` *(= cage 12 -4 6 4 .)*

Graphe ou cube de Bidiakis. C'est un graphe planaire cubique à 12 sommets. Il est hamiltonien et son nombre chromatique est 3. On peut le représenter comme un cube où deux faces opposées comportent une arête supplémentaire perpendiculaire joignant deux bords opposés. On peut aussi le représenter comme un cycle avec 3 colonnes et 3 lignes parallèles joignant des sommets opposés (comme une raquette de tennis).

`dyck` *(= cage 32 5 0 13 -13 .)*

Graphe de Dyck. C'est un graphe cubique 3-connexe biparti à 32 sommets. C'est le seul graphe cubique à 32 sommets à être symétrique, c'est-à-dire qui est à la fois arête et sommet transitif. Il est aussi torique, c'est-à-dire de genre 1.

`pappus` *(= cage 18 5 7 -7 7 -7 5 .)*

Graphe de Pappus. C'est un graphe cubique à 18 sommets, de maille 6 et de diamètre 4.

`tutte-coexter` *(= cage 30 -7 9 13 -13 -9 7 .)*

Graphe de Tutte-Coexter appelé aussi 8-cage de Tutte. C'est un graphe cubique à 30 sommets, de maille 8 et de diamètre 4. C'est un graphe de Levi mais surtout un graphe de Moore, c'est-à-dire un graphe d-régulier de diamètre k dont le nombre de sommets est 1+d·S(d,k) (si d impair) ou 2·S(d,k) (si d pair) avec S(d,k)=∑_{i=0}^{k-1} (d-1)^i.

`gray` *(= cage 54 7 -7 25 -25 13 -13 .)*

Graphe de Gray. C'est un graphe cubique à 54 sommets qui peut être vu comme le graphe d'incidence entre les sommets d'une grille 3×3×3 et les 27 lignes droites de la grille. Il est hamiltonien, de diamètre 6, de maille 8, et de genre 7. Il est arête-transitif et régulier sans être sommet-transitif.

`chvatal` *(= cage 12 3 6 3 6 6 3 6 -3 3 -3 3 3 .)*

Graphe de Chvátal (1970). C'est le plus petit graphe régulier sans triangle de nombre chromatique 4. Il est 4-régulier, possède 12 sommets, est non planaire, hamiltonien et de diamètre 2.

grotzsch *(= mycielski 4)*

Graphe de Grötzsch. C'est le plus petit graphe sans triangle de nombre chromatique 4 (sans être régulier contrairement au graphe de Chvátal). Il possède 11 sommets et 20 arêtes. Comme le graphe de Chvátal, il est non planaire de diamètre 2, de maille 4 et hamiltonien. C'est le graphe de Mycielskian du cycle à 5 sommets.

hajos *(= sierpinski 2 3)*

Graphe de Hajós. Il est composé de trois triangles deux à deux partageant un sommet distinct. On peut le dessiner comme un triangle dans un triangle plus grand. Il est planaire et possède 6 sommets. C'est un graphe `sierpinski` ou encore le complémentaire d'un `sunlet 3`, le complémentaire du `netgraph`, un `sunflower 3` ou encore `cage 6 2 0 .`.

house *(= grid 5 . -op not)*

Graphe planaire à 5 sommets en forme de maison. C'est le complémentaire d'un chemin à 5 sommets.

wagner *(= ring 8 1 4 .)*

Graphe de Wagner appelé aussi graphe W_8, un cycle à 8 sommets où les sommets antipodaux sont adjacents. C'est un graphe cubique à 8 sommets qui n'est pas planaire mais sans K_5. C'est aussi une échelle de Möbius (`mobius`).

mobius n *(= ring n 1 n/2 .)*

Échelle de Möbius, graphe cubique à n sommets obtenu à partir d'un `cycle n` sommets dont les sommets opposés sont adjacents. Lorsque n est pair, il s'agit d'un ruban de Möbius, c'est-à-dire d'une échelle dont le premier et dernier barreau sont recollés en sens opposé. Pour n≤5, il s'agit d'une clique à n sommets. Il est donc cubique sauf pour n=1,2,3,5. Lorsque n≥5, le graphe n'est plus planaire, et pour n=8, il s'agit du graphe `wagner`.

ladder n *(= grid 2 n .)*

Graphe échelle à n barreaux, soit une grille à 2 × n sommets.

diamond *(= grid 2 . -op apex 2)*

Clique à quatre sommets moins une arête. C'est un graphe allumette, c'est-à-dire planaire et distance unitaire.

`gosset` *(= `ggosset 8 2 3 6 -1 .`)*

Graphe de Gosset. Il est 27-régulier avec 56 sommets et 756 arêtes, de diamètre, de rayon et de maille 3. Il est 27-arête-connexe, 27-sommet-connexe et hamiltonien. C'est localement un graphe `schlafli`, c'est-à-dire que pour tout sommet le sous-graphe induit par ses voisins est isomorphe au graphe de Schläfli, qui est lui-même localement un graphe `clebsch`.

`wheel n` *(= `ringarytree 1 0 n 2`)*

Roue à n rayons. Graphe planaire à n+1 sommets composé d'un `cycle n` sommets et d'un sommet universel, donc connecté à tous les autres.

`web n r` *(= `ringarytree r 1 n 2`)*

Graphe planaire à 1+n·r sommets composé d'une étoile à n branches de longueur r, les sommets de même niveau étant connectés par un cycle. Il généralise `wheel n` (r=1).

`binary h` *(= `ringarytree h 2 2 0`)*

Arbre binaire complet de hauteur h. Il possède 2^(h+1)-1 sommets et la racine est de degré deux.

`arytree h k r` *(= `ringarytree h k r 0`)*

Arbre complet de hauteur h où chaque nœud interne a exactement k fils, la racine étant de degré r.

`rbinary n` *(= `rarytree n 2 0`)*
`rbinaryz n` *(= `rarytree n 2 1`)*

Arbre binaire plan aléatoire uniforme à n nœuds internes. Il possède 2n-1 sommets (2n pour la variante `rbinaryz`) numérotés selon un parcours en profondeur modifié : tous les fils du sommet courant sont numérotés avant l'étape de récursivité. La racine est de degré 2 (`rbinary`) ou 1 (`rbinaryz`). Le dessin avec `dot` ne respecte pas le plongement de l'arbre. L'option `-directed` permet d'obtenir une 1-orientation.

`tw n k` *(= `ktree n k -op del-e .5`)*

Graphe de largeur arborescente au plus k aléatoire à n sommets. Il s'agit d'un k-arbre partiel aléatoire dont la probabilité d'avoir une arête est 1/2. L'option `-directed` permet d'obtenir une k-orientation.

`pw n k` *(= `kpath n k -op del-e .5`)*

Graphe de pathwidth au plus k, aléatoire et avec n sommets.

`tadpole n p` *(= barbell -n 1 p)*
`dragon n p` *(= barbell -n 1 p)*

Graphe à n+p sommets composé d'un cycle à n sommets relié à un chemin à p sommets.

`lollipop n p` *(= barbell n 1 p)*

Graphe « tapette à mouches » (Lollipop Graph) composé d'une clique à n sommets reliée à un chemin de longueur p. Il a n+p sommets.

`pan n` *(= barbell -n 1 1)*

Graphe à n+1 sommets composé d'un `cycle n` et d'un seul sommet pendant.

`banner` *(= barbell -4 1 1)*

Graphe à 5 sommets composé d'un carré et d'un sommet pendant.

`paw` *(= barbell -3 1 1)*

Graphe à 4 sommets composé d'un `triangle` et d'un sommet pendant.

`theta0` *(= barbell -5 -5 -2)*

Graphe Theta_0. C'est un graphe à 7 sommets série-parallèle obtenu à partir d'un `cycle 6` et en connectant deux sommets antipodaux par un `path 3`. C'est un graphe allumette, c'est-à-dire planaire et distance unitaire.

`nng n` *(= knng n 1)*

Graphe du plus proche voisin (Nearest Neighbor Graph). Graphe géométrique défini à partir d'un ensemble de n points du carré $[0, 1]^2$ (distribution par défaut). Le point i est connecté au plus proche autre point (par défaut selon la norme `L2`, voir `-norm`). Ce graphe est une forêt couvrante du graphe `rng` de degré au plus 6 (si la norme est `L2`).

`td-delaunay n` *(= thetagone n 3 3 1)*

Triangulation de Delaunay utilisant la distance triangulaire (TD=Triangular Distance). Ce n'est malheureusement pas toujours une triangulation, les sommets du bord pouvant être de degré un. Il s'agit d'un

graphe planaire défini à partir d'un ensemble de n points aléatoires du carré $[0, 1]^2$ (distribution par défaut). Ce graphe a un étirement de 2 par rapport à la distance euclidienne entre deux sommets du graphe. Ce graphe, introduit par Chew en 1986, est le même que le graphe « demi-theta_6 », qui est un « theta-graph » utilisant 3 des 6 cônes. La dissymétrie qui peut apparaître entre les bords droit et gauche du dessin est liée au fait que chaque sommet n'a qu'une seule bissectrice de cône dirigée vers la droite, alors qu'il y en a deux obliques vers la gauche.

`theta n k` *(= `thetagone n 3 k 6/k`)*

Theta-graphe à k>0 secteurs réguliers défini à partir d'un ensemble de n points du carré $[0, 1]^2$. Les sommets u et v sont adjacents si le projeté de v sur la bissectrice de son secteur est le sommet le plus proche de u. Ce graphe n'est pas planaire en général (sauf pour k<3), mais c'est un spanner du graphe complet euclidien si k≥6.

`dtheta n k` *(= `thetagone n 3 [k/2] 6/k`)*

Demi-Theta-graphe à k≥2 secteurs réguliers défini à partir d'un ensemble de n points du carré $[0, 1]^2$ (distribution par défaut). La définition est similaire au Theta-graphe excepté que seul 1 secteur sur 2 est considéré. Il faut k pair. Pour k=2, il s'agit d'un arbre, pour k=4, le graphe est de faible tree-width pas toujours connexe. Pour k=6, ce graphe coïncide avec le graphe `td-delaunay`.

Ex :
```
gengraph dtheta 500 6 -visu
gengraph dtheta 500 4 -view no pos -visu
gengraph dtheta 500 2 -view no pos -visu
```

`yao n k` *(= `thetagone n 0 k 2/k`)*

Graphe de Yao à k>0 secteurs réguliers défini à partir d'un ensemble de n points du carré $[0, 1]^2$ (distribution par défaut). Les sommets u et v sont adjacents si v est le sommet le plus proche de u (selon la distance euclidienne) de son secteur. Ce graphe n'est pas planaire en général, mais c'est un spanner du graphe complet euclidien. Le résultat est valide seulement si k≥2. En fait, ce n'est pas tout à fait le graphe de Yao (voir `thetagone`).

`percolation a b p` *(= `udg a·b 1 -norm L1 -op del-e 1-p -xy mesh a b`)*

Grille de percolation à coordonnées entières (i,j) de [0,a[× [0,b[où p représente la probabilité d'existence de chaque arête. La différence avec le graphe `mesh a b -op del-e 1-p` est qu'ici le graphe est géométrique, donc dessiné selon une grille avec `dot`.

`hudg n r` *(= `udg n r -norm hyper -xy hyper r`)*

Graphe géométrique aléatoire hyperbolique sur n points du disque unité. Deux sommets sont adjacents si leurs points sont à distance hyperbolique ≤ r. [À FINIR]

`point n (= ring n . -xy unif)`

Graphe géométrique composé de n points du plan, sans aucune arête. Ce graphe permet de visualiser la distribution des points, par défaut uniforme sur $[0, 1]^2$ (`-xy unif`). Avec une option `-xy`, il est équivalent à `stable n`.

Ex :
```
gengraph point 500 -xy seed 3 2.1 -visu
gengraph point 1000 -xy seed 3 -0.1 -visu
gengraph point 1000 -xy disk -visu
```

`star-polygon n (= ring n 1 . -xy disk)`

Polygone « star-shaped » aléatoire à n côtés contenu dans le disque unité (voir `-xy ratio`).

`convex-polygon n (= ring n 1 . -xy convex)`

Polygone convexe aléatoire à n côtés contenu dans le disque unité (voir `-xy ratio`). Voir aussi le graphe `polygon n`.

`regular n d (= fdrg n d .)`

Graphe d-régulier aléatoire à n sommets asymptotiquement uniforme. Il faut que n·d soit pair. L'algorithme est de complexité $O(n \cdot d^2)$ et pour être asymptotiquement uniforme, il faut $d=o(\sqrt{n})$. On obtient nécessairement un `matching` si d=1, un `stable` si d=0, un `cycle` si d=2 et n<6.

`cubic n (= fdrg n 3 .)`

Graphe cubique aléatoire à n sommets asymptotiquement uniforme. Il faut que n soit pair.

`plrg n t (= bdrg n_1 d_1 … n_k d_k .)`

Power-Law Random Graph (ou scale-free graph). Graphe aléatoire à n sommets dont la distribution des degrés suit une loi en puissance d'exposant t>0 (typiquement un réel t∈]2,3[), la probabilité qu'un sommet soit de degré i>0 étant proportionnelle à 1/i^t. Plus précisément, la distribution est la suivante :

- d_1=1, n_1=⌊exp(α)⌋ + n-s(α)
- d_i=i, n_i=⌊exp(α)/i^t⌋ pour 2≤i≤p(α)

où a est un réel minimisant |n-s(α)| avec p(α)=⌊exp(α/t)⌋ et s(α)=∑_{i=1}^{p(α)} ⌊exp(α)/i^t⌋. Ce sont les mêmes graphes que ceux générés par Brady-Cowen'06 ou ceux étudiés par Lu'01.

`tree_fibo n (= calls_tree "…" f n .)`

Arbre des appels pour la fonction récursive calculant le (n+1)-ième nombre f(n) de la suite de Fibonacci pour tout n≥0. Il est défini par la récurrence f(n) = n si n<2, f(n) = f(n-1) + f(n-2). L'arbre est orienté et possède 2f(n+1)-1 ≃ 1.789*1.618^n sommets. Le nombre de feuilles vaut précisément f(n), ce qui donne un moyen de le calculer en comptant les feuilles avec `-undirected -check deg`.

Ex :
```
gengraph tree_fibo 10 -label 1 -dot filter dot -visu
gengraph tree_fibo 10 -undirected -check deg
```

Une invocation `tree_fibo $n` est équivalente à l'invocation suivante de `calls_tree` :

```
calls_tree "f(0) = 0; f(1) = 1; f(n) = f(n - 1) + f(n - 2)" f $n .
```

`tree_part n k`

Arbre des appels pour la fonction récursive calculant le nombre p(n,k) de partitions de l'entier n>0 en 0<k≤n parts. Il est défini par la récurrence p(n,k) = 1 si k=1, p(n,k) = p(n-1,k-1) si n<2k, et p(n,k) = p(n-1,k-1) + p(n-k,k) sinon. L'arbre est orienté et le nombre de feuilles vaut précisément p(n,k), ce qui donne un moyen de le calculer en comptant les feuilles avec `-undirected -check deg`.

Ex :
```
gengraph tree_part 22 6 -label 1 -dot filter dot -visu
gengraph tree_part 22 6 -undirected -check deg
```

Il existe deux variantes (`-variant 1` et `-variant 2`), calculant également p(n,k), avec des récurrences légèrement différentes.

- Pour `-variant 1` elle est : p(n,k) = 0 si n<k, p(n,k) = 1 si k=1 ou k=n, et p(n,k) = p(n-1,k-1) + p(n-k,k) sinon. Dans ce cas l'arbre est binaire complet (chaque sommet a 0 ou 2 fils), mais le nombre de feuilles n'est plus p(n,k), il est plus grand. De plus il a plus de sommets que la variante par défaut (`-variant 0`).
- Pour `-variant 2` elle est : p(n,k) = 1 si k=1 ou k=n, p(n,k) = p(n-1,k-1) si n<2k, et p(n,k) = p(n-1,k-1) + p(n-k,k) sinon. Elle est très proche de `-variant 0` avec un nombre de feuilles égal à p(n,k) et un arbre qui est binaire mais pas complet. Cette variante possède moins de sommets ayant 1 fils que `-variant 0`, et donc encore moins de sommets que `-variant 1`.

Les invocations des trois variantes `tree_part $n $k` sont équivalentes aux invocations suivantes de `calls_tree` :

```
calls_tree "p(n, 1) = 1; p(n, k) [n < 2 * k] = p(n - 1, k - 1);
  p(n, k) = p(n - 1, k - 1) + p(n - k, k)" p $n $k .
calls_tree "p(n, k) [n < k] = 0; p(n, 1) = 1; p(n, n) = 1;
  p(n, k) = p(n - 1, k - 1) + p(n - k, k)" p $n $k .
calls_tree "p(n, 1) = 1; p(n, n) = 1; p(n, k) [n < 2 * k] = p(n - 1, k - 1);
  p(n, k) = p(n - 1, k - 1) + p(n - k, k)" p $n $k .
```

`tree_binom n k` *(= `calls_tree "…" binom n k .`)*

Arbre des appels pour la fonction récursive calculant le coefficient binomial binom(n,k) pour tous entiers n≥1 et 0≤k≤n. Il est défini par la récurrence binom(n,k) = binom(n-1,k-1) + binom(n-1,k) si 0<k<n, et binom(n,k) = 1 si k=0 ou k=n. L'arbre est un arbre binaire orienté et possède binom(n,k) = n!/k!/(n-k)! feuilles, ce qui donne un moyen de calculer ce nombre en comptant les feuilles avec `-undirected -check deg`.

Ex :
```
gengraph tree_binom 5 2 -label 1 -dot filter dot -visu
gengraph tree_binom 5 2 -undirected -format no -check deg
```

Une invocation `tree_binom $n $k` est équivalente à l'invocation suivante de `calls_tree` :

```
calls_tree "binom(n, 0) = 1; binom(n, n) = 1;
  binom(n, k) = binom(n - 1, k - 1) + binom(n - 1, k)" binom $n $k .
```

`syracuse n` *(= `collatz n 1 0 3 1 .`)*

Graphe orienté issu du problème « 3x+1 » de Collatz. D'après la conjecture, il s'agit d'un graphe connexe. Les arcs sont définis par la relation x↦x/2 si x est pair, et x↦(3x+1)/2 sinon. La boule de volume n est générée depuis 1 si n>0 en itérant la relation inverse, et dans ce cas le graphe a précisément n sommets. Si n<0, la relation est itérée pour tous les entiers de [1,|n|] dans la limite de n^2 sommets.

Ex :
```
gengraph syracuse 18 -label 1 -visu
gengraph syracuse -18 -label 1 -visu
```

Le rayon de la boule de volume n (excluant les valeurs multiples de 3 qui n'ont toujours qu'un seul prédécesseur) est expérimentalement < ln(n)/ln(1.36) pour tout n<10^18 et conjecturée en ln(n)/ln(4/3). La hauteur maximum atteinte à partir d'un entier de [1,n] est expérimentalement < n^2, avec seulement 7 exceptions pour n<10^18 (et dans ces cas là, la hauteur est < $9n^2$). Pour n=27, on obtient une hauteur record de 4,616 ≃ $7n^2$, le second record pour n<10^18. Selon les conjectures, les trajectoires de hauteur maximum pour n ont une forme générale qui consiste à une montée ≃ 8·ln(n) étapes pour atteindre le maximum, puis une descente ≃ 24·ln(n) étapes, et les trajectoires extrêmes ont ≃ 42·ln(n) étapes.

kakutami_3x+1 n *(= collatz n 1 0 6 2 .)*

Variante de `syracuse` « non compressée » qui est définie par la relation x↦x/2 si x est pair et x↦3x+1 sinon. Comme avec le graphe de `collatz`, il est possible d'avoir n<0.

kakutami_5x+1 n *(= collatz n 3 0 30 6 3 0 2 0 3 0 30 6 .)*

Graphe orienté issu du problème « 5x+1 » de Kakutami (cf. `syracuse`) défini par la relation x↦x/2 si x est pair, x↦x/3 si x est divisible par 3 mais pas par 2, et x↦5x+1 sinon. D'après la conjecture de Kakutami, il s'agit d'un graphe connexe. On peut se ramener à un graphe de Collatz en considérant 2×3=6 paires de coefficients. Comme avec le graphe `collatz`, il est possible d'avoir n<0.

Ex : `gengraph kakutami_5x+1 91 -undirected -visu`

kakutami_7x+1 n *(= collatz n 15 0 210 30 15 0 10 0 … 210 30 .)*

Graphe orienté issu du problème « 7x+1 » de Kakutami (cf. `syracuse`) défini par la relation x↦x/2 si x est pair, x↦x/3 si x est divisible par 3 mais pas par 2, x↦x/5 si x est divisible par 5 mais ni par 2 ni par 3, et x↦7x+1 sinon. D'après la conjecture de Kakutami, il s'agit d'un graphe connexe. On peut se ramener à un graphe de Collatz en considérant 2×3×5=30 paires de coefficients. Comme avec le graphe `collatz`, il est possible d'avoir n<0.

farkas n *(= collatz n 6 0 6 6 6 0 4 0 6 0 … 18 6 .)*

Graphe orienté issu de la relation introduite par Farkas en 2005 qui a prouvé qu'elle définissait un arbre de racine 1. Proche du problème « 3x+1 » elle est définie par x↦x/2 si x est pair, x↦x/3 si x est divisible par 3 mais pas par 2, (3x+1)/2 si x≡3 (mod 4) et (3x+1)/2 si x≡1 (mod 4). On peut se ramener à un graphe de Collatz en considérant 12 paires de coefficients. Comme avec le graphe `collatz`, il est possible d'avoir n<0.

Ex : `gengraph farkas -34 -label 1 -visu`

GRAPHES EXTERNES

Ces graphes sont chargés à partir de données externes.

load <fichier>[:<sélecteur>]

Graphe au format simple contenu dans le fichier `fichier`, ou de l'entrée standard si `fichier` vaut `-`. Si le fichier contient un groupe de graphes, alors seul le premier est chargé, et son identifiant n'est pas

conservé. Si un sélecteur est spécifié (voir `-test` pour le format) et que le fichier contient un groupe de graphes, le graphe chargé est le premier du fichier qui fait partie de la sélection.

Les sommets doivent être numérotés par des entiers positifs. L'option `-directed` est activée si `fichier` contient au moins un arc, auquel cas l'option `-undirected` n'aura pas d'effet.

Le temps et l'espace nécessaires au chargement des graphes sont linéaires. Toutefois, le mode de génération par défaut parcourt la matrice d'adjacence du graphe, en $O(n^2)$. L'option `-fast` permet de se limiter à un parcours de la liste d'adjacence, en O(n+m).

Pour charger un graphe au format DOT, on peut utiliser le script `dot2gg.awk` en amont (dans le dossier `tools` du dépôt), comme dans l'exemple suivant :

Ex : `nop fichier.dot | awk -f dot2gg.awk | gengraph load -`

Le filtre `nop` de Graphviz, qui est recommandé mais pas forcément nécessaire, permet de normaliser le format DOT initial. Il transforme par exemple les expressions du type `a--{b;c;}` en `a--b;a--c;`.

Notez que la suite d'options `load <fichier> -fast -format dot-<type>` permet de convertir `fichier` au format `type` souhaité.

load-str <description>

Graphe défini par sa description au format simple (qui doit bien sûr être échappée avec des guillemets sur la ligne de commande), comme avec `load`.

Ex : `gengraph load-str '0-1-(2 3 4) 0-5-4-4-6 7 8'`

load+ <fichier>[:<sélecteur>]

Graphes de `fichier`, éventuellement filtrés par `sélecteur`. Tous les graphes du fichier sont chargés en mémoire et placés dans un groupe. Voir `load` pour plus d'informations. Contrairement à ce qui se produit avec `load`, les identifiants des graphes sont repris. Les graphes sont chargés en mémoire dès la lecture de la commande `load+` (ce qui n'est pas le cas avec `load`).

load-str+ <description>

Comme `load+`, mais à partir d'une description fournie en paramètre comme avec `load-str`.

HISTORIQUE

v1.2, octobre 2007

- Première version disponible.

v1.3, octobre 2008

- Options : `-shift`, `-width`.
- Correction d'un bug pour les graphes de `permutation`.
- Accélération du test d'ajacence pour les arbres, de O(n) à O(1), grâce à la représentation implicite.
- Nouveaux graphes : `outerplanar`, `sat`.

v1.4, novembre 2008

- Nouveaux formats de sortie : `matrix`, `smatrix`, `list`.
- Nouveau graphe : `kout`.
- Correction d'un bug dans l'option `-width`.
- Correction d'un bug dans la combinaison `-format -shift -delv`.

v1.5, décembre 2008

- Correction d'un bug dans `tree` lorsque n=1.

v1.6, décembre 2009

- Nouveaux graphes : `rpartite`, `bipartite`.

v1.7, janvier 2010

- Nouveaux graphes : `icosa`, `dodeca`, `rdodeca`, `cubocta`, `geo`, `wheel`, `cage`, `headwood`, `pappus`, `mc-gee`, `levi`, `butterfly`, `hexagon`, `whexagon`, `arytree`, `binary`, `ktree`, `tw`, `kpath`, `pw`, `arboricity`, `wagner`, `mobius`, `tutte-coexter`, `paley`.
- Nouveau format de sortie : `-format dot`.
- Nouvelles options : `-header`, `-h`, `-redirect`, `-dotpdf`.
- Correction d'un bug dans `kout`, et dans `tree` lorsque n=0.
- `tree` devient un cas particulier d'`arboricity`.
- Aide intégrée pour les paramètres des graphes.

v1.8, juillet 2010

- Nouveaux graphes : `chvatal`, `grotzsch`, `debruijn`, `kautz`, `gpstar`, `pstar`, `pancake`, `nauru`, `star`, `udg`, `gpetersen`, `mobius-kantor`, `desargues`, `durer`, `prism`, `franklin`, `gabriel`, `thetagone`, `td-`

`delaunay`, `yao`, `theta`, `dtheta`.
- Suppression du graphe `geo` (remplacé par `udg`).
- Nouvelles options : `-pos`, `-norm`, `-label`, `-dotfilter`.
- Nouvelle famille d'options : `-xy file/noise/scale/seed`.
- Définition plus compacte de `dodeca` (non explicite).
- Utilisation du générateur `random()` plutôt que `rand()`, pour garantir une meilleure génération de valeurs aléatoires.
- Correction d'un bug dans `-format standard` qui provoquait une erreur.
- Correction d'un bug dans kneser pour k=0, n=0 ou k>n/2.
- Nouveaux formats : `-format dot<type>`, `-format xy`.
- Suppression de `-dotpdf` (qui est maintenant : `-format dotpdf`).
- Labeling pour : `gpetersen`, `gpstar`, `pstar`, `pancake`, `interval`, `permutation`.

v1.9, août 2010

- Renommage de `-h` en `-list`.
- Renommage de `-xy file` en `-xy load`.
- Centrage des positions sur le barycentre des graines (`-xy seed`).
- Nouvelles options : `-star`, `-visu`, `-xy round`.
- Les graphes peuvent être stockés en mémoire, sous la forme d'une liste d'adjacence grâce à l'option `-check`.
- Généralisation de `-delv <p>` avec p<0.
- Nouveaux graphes : `caterpillar`, `hajos`, `hanoi`, `sierpinski`, `sunlet`, `load`.
- Labeling pour : `hanoi`, `sierpinski`.
- Aide sur toutes les options (nécessitant au moins un paramètre) et non plus seulement pour les graphes.
- Nouvelle famille d'options : `-vcolor deg/degr/pal`.
- Correction d'un bug pour l'aide dans le cas de commande préfixe (ex : `pal` & `paley`).

v2.0, septembre 2010

- Nouvelles options : `-vcolor degm/list/randg`, `-xy unique/permutation`, `-check bfs`, `-algo iso/sub`.
- L'option `-xy round <p>` admet des valeurs négatives pour `p`.
- Le graphe `load <fichier>` et l'option `-xy load <fichier>` prennent en charge la lecture à partir de l'entrée standard (en donnant `-` comme nom de fichier), la lecture de famille de graphes, et les commentaires.
- Les formats `list`/`matrix`/`smatrix` utilisent un espace linéaire O(n+m) contre O(n^2) auparavant.
- Les sommets sur le bord (graphes géométriques) ne sont plus coupés (boîtes englobantes plus grandes).

- Nouveaux graphes : `kpage`, `outerplanar n` *(= kpage n 1)*, `rng`, `nng`, `fritsch`, `soifer`, `gray`, `hajos` (qui avait été défini mais pas implémenté !), `crown`, `moser`, `tietze`, `flower_snark`, `markstrom`, `clebsch`, `robertson`, `kittell`, `rarytree`, `rbinary`, `poussin`, `errera`.
- Les graphes de `gabriel` (ainsi que `rng` et `nng`) dépendent maintenant de `-norm`.
- `wheel n` a maintenant n+1 sommets, et non plus n.
- Aide intégrée améliorée avec `?`.

  Ex : `gengraph tutte ? ; gengraph -visu ?`

- Les options `-help` et `?` permettent la recherche d'un mot clef.

  Ex : `gengraph -help planaire ; gengraph ? arbre`

- Description plus compacte de `tutte` (et des graphes à partir d'un tableau).
- Correction d'un bug pour `rpartite` (qui ne marchait pas).

v2.1, octobre 2010

- Nouvelles options : `-check degenerate/gcolor/edge/dfs/ps1/paths/paths2/minor/isub`, `-filter minor/sub/iso/vertex/edge/degenerate/ps1`, `-filter degmax/degmin/deg/gcolor/component/radius/girth/diameter`, `-filter cut-vertex/bi-connected/isub/all/minor-inv/isub-inv/sub-inv`.
- Renommage de `-algo iso/sub` en `-check iso/isub`.
- Extension de `-label <b>` à b=2 qui force l'affichage des noms sous forme d'entiers, même avec `-permute`.
- Correction d'un bug pour `house` (qui ne marchait pas).
- Nouveau graphe : `windmill`.

v2.2, novembre 2010

- Gestion des graphes orientés : lecture d'un fichier de graphe (ou d'une famille avec arcs et arêtes).
- Nouvelles options : `-[un]directed`, `-[no]loop`, `-check twdeg/tw`, `-filter tw/id/hyperbol/rename`.
- Permet l'affichage de la valeur (`p`) dans l'option `-filter`.
- Nouveau graphe : `aqua`.
- Correction du graphe `tutte-coexter` et suppression du graphe `levi` (qui en fait était le graphe de Tutte-Coexter).
- Généralisation du graphe `load` à `load :<id> <famille>`.

v2.3, décembre 2010

- Nouvelles options : `-check ps1bis/edge`, `-filter ps1bis/tw2`, `-filter minus/minus-id/unique/connected/bipartite/forest`, `-check ps1ter`.
- Remplacement de `LONG_MAX` par `RAND_MAX` (=2^31-1) dans les expressions faisant intervenir `random()` (qui est de type `long` mais qui est toujours dans [0,2^31[, même si `sizeof(long)`>4). Il y avait un bug pour les architectures avec `sizeof(long)`>4.
- Nouveau graphe : `cylinder`.
- Suppression de la variante `load :<id>` au profit de la forme plus générale `<fichier>:<inter-valle>` valable pour `load`, `-filter`, etc.

v2.4, janvier 2011

- Correction d'un bug dans `-filter minus-id`.
- Correction d'un bug dans `rpartite` (incorrect à partir de r>5 parts).
- Correction d'un bug dans `whexagon` (nombre de sommets incorrect).
- Nouvelles options : `-check ps1x/girth`, `-filter ps1c/ps1x`.
- Renommages : `ps1bis` → `ps1b`, `ps1ter` → `ps1c`.
- Nouveau graphe : `mycielski`.
- Le graphe `grotzsch` est maintenant défini à partir du graphe `mycielski` (la définition précédente était fausse).
- Bug détecté : `td-delaunay 500 -check gcolor -format no -seed 7 | grep '>6'` qui donne jusqu'à 7 couleurs ; le nombre de couleurs affichées dans `-check gcolor` est erroné.

v2.5, mars 2011

- Nouveaux graphes : `line-graph`, `claw`.

v2.6, juin 2011

- Amélioration du test `-filter ps1` : détection de cliques et d'arbres.

v2.7, octobre 2011

- Nouvelle option : `-check bellman` (pour les géométriques seulement).
- Ajout des champs `xpos,ypos` à la structure `graph`.
- Nouveaux graphes : `linial`, `linialc`, `cube`, `diamond`, `theta0`.

v2.8, novembre 2011

- Nouveaux graphes : `ggosset`, `gosset`, `rplg`, `wiener-araya`, `headwood4`.
- Correction d'un bug pour `-xy seed <k> <n>` lorsque k=1.
- Nouvelles options : `-check maincc`, `-maincc` (non documentée).

v2.9, février 2013

- Nouveau graphe : `frucht`, `halin`.
- Correction d'un bug pour `-check gcolor` qui ne renvoyait pas le nombre correct de couleurs, et qui de plus n'utilisait pas l'heuristique du degré minimum.
- Correction d'un bug affectant `permutation -label 1`.

v3.0, octobre 2013

- Nouveaux graphes : `rig`, `barbell`, `lollipop`.
- Généralisation de l'option `-filter forest`.
- Nouvelles options : `-apex`, `-filter isforest`, `-filter istree`, `-filter cycle`.
- Correction d'un bug dans `-filter vertex`.
- Amélioration de l'aide lors d'erreurs de paramètre comme : `-filter <F> vertex` au lieu de `-filter <F> vertex <n>`.
- Amélioration de l'option `-header`.

v3.1, décembre 2013

- Nouveau graphe : `bpancake`.
- Légère modification des labels des sommets des graphes `pancake`, `gpstar` et `pstar`.
- Nouvelles options : `-xy grid`, `-xy vsize`.
- Modification de la taille des sommets pour `dot` permettant de tenir compte de `-xy scale`.

v3.2, mai 2014

- Amélioration du test de `-check ps1b` (ajout de règle et réduction du nombre d'indéterminées dans graphes des conflits).

v3.3, juillet 2014

- Modification importante du code pour `-check ps1`.
- Modification des graphes `linial` et `linialc`.
- Nouvelles options : `-check kcolor`, `-vcolor kcolor`, `-len`, `-check kcolorsat`.

v3.4, février 2015

- Documentation et mise au point de l'option `-maincc`.
- Correction d'un bug lors de la combinaison de `load <fichier>` et de `-vcolor pal <grad>`.
- Correction d'un bug dans la fonction `SortInt()` qui affectait `-vcolor deg`.
- Correction d'un bug avec l'option `-label 1` pour certains graphes (`outerplanar`...).

- Création du script `dot2gen.awk` pour convertir le format DOT en format standard (voir `load`).
- Nouvelles options : `-fast`, `-caption`.
- Introduction du groupement d'arêtes/arcs `i-(j k …)` dans le format standard (il reste un bug si ce format est lu depuis l'entrée standard).

v3.5, mars 2015

- Correction d'un bug pour le dégradé de couleurs avec `-vcolor pal <grad>`.
- Nouvelles options : `-check ncc/connected`, `-check routing scenario […] cluster`.
- Nouvelle variante de graphe : `loadc <fichier>`.
- Nouveaux graphes : `octahedron`, `turan`, `hexahedron`, `tetrahedron`, `deltohedron=trapezohedron`, `antiprism`, `flip`, `associahedron`, `shuffle`.
- Changements de nom (ajout du suffixe `hedron`) pour : `isocahedron`, `dodecahedron`, `cuboctahedron`, `rdodecahedron`.

v3.6, juin 2015

- Nouvelles options : `-version`, `-variant`, `-check info`, `-xy mesh`, `-check routing tzrplg`, `-check routing dcr`.
- Nouveaux graphes : `percolation`, `hgraph`, `cricket`, `moth`, `bull`, `expander`.
- Correction de bug dans `-help ?`, dans `-check girth` (qui donnait -1 pour un cycle de taille n), dans `butterfly` (mauvais nombre de paramètres).
- Vérification que le nombre de sommets n'est pas négatif pour plusieurs graphes dont `tree`, `kout`, etc., ce qui pouvait provoquer des erreurs.
- Vérification du format d'entrée pour `load[c]`.
- L'option `-check maincc` n'affiche plus le graphe initial.
- Description du format des familles de graphes dans l'aide.
- Affiche plus de messages d'erreurs : lecture d'un graphe au mauvais format, mauvaise combinaison d'options…

v3.7, juillet 2015

- Nouvelle implémentation des mots de Dyck, qui sont maintenant uniformes. En conséquence les graphes aléatoires `tree`, `arboricity`, `rarytree`, `rbinary`, `outerplanar`, `kpage`… sont générés de manière uniforme.
- Nouveaux graphes : `treep` (et simplification de `halin`), `ygraph`, `ringarytree` (et simplification de `arytree`, `binary`, `wheel`), `netgraph`, `egraph`, `rgraph=fish`, généralisation de `barbell`, `tadpole=dragon`, `pan`, `banner`, `paw`, `theta0` (redéfinition), `fan`, `gem`, `chess`, `knight`, `antelope`, `camel`, `giraffe`, `zebra`, `utility`, `zamfirescu`, `hatzel`, `web`, `bdrg`, `fdrg`, `regular`, `cubic`, `plrg`.
- Nouvelles options : `-check radius`, `-check diameter`.

- Correction d'un bug si combinaison d'options `-check ncc -format list` (mauvaise gestion de la variable `CHECK`).
- Introduction de caractères UTF-8 mathématiques dans l'aide.

v3.8, novembre 2015

- Nouveaux graphes : `ladder`, `matching`, `d-octahedron`, `johnson`.
- Correction d'un bug pour le graphe `kpage` (introduit à la v3.7).

v3.9, février 2016

- Renommage de l'option `-xy scale` en `-xy box`.
- Correction d'un bug dans `rbinary` (mauvais alias/définition).
- Correction d'un bug (introduit à la v3.6) dans la fonction `max(double,double)` au lieu de `fmax()` qui affectait certains graphes et options géométriques (`rng`, `-xy grid`, `percolation`...).
- Correction d'un bug concernant `rarytree` lorsque b>2.
- Correction d'un bug dans les statistiques pour `-check routing`.
- Généralisation du graphe `rarytree` (augmentation du degré de la racine).
- Nouveaux graphes : `herschel`, `goldner-harary`, `rbinaryz`, `point`, `empty`, `apollonian`, `dyck`, `cross`, `tgraph`, `bidiakis`, `gear`, `centipede`, `harborth`, `sunflower`, `planar`, `squaregraph`, `polygon`.
- Modification de certains graphes (`cage`, `ring`, `gabriel`, `nng`, `rng`) afin que le test d'adjacence ne dépende plus de `N` mais de `PARAM[0]` par exemple.
- Introduction du format `%SEED` pour l'option `-caption`.

v4.0, mars 2016

- Nouveaux graphes : `pat`, `star-polygon`, `convex-polygon`.
- Nouvelles options : `-check kindepsat`, `-xy circle`, `-xy polar`, `-xy unif`, `-xy convex`, `-xy ratio`, `-xy zero`.
- Aide permettant les préfixes comme dans `-check routing cluster`.

v4.1, avril 2016

- Nouvelles options : `-xy convex2`, `-check routing hdlbr/bc`.
- Nouveaux graphes : `kstar`, `split`, `cactus`, `squashed`.
- Suppression de `-check paths2` qui donnait toujours comme `-check paths`.
- Option `-check bellman` apparaît dans l'aide.

v4.2, mai 2016

- Nouvelles options : `-check routing agmnt`, `-format vertex`, `-check routing hash mix`, `-xy unif`, `-xy polygon`.
- Amélioration de l'option `-check info`.
- Prise en compte de `-xy ratio` dans `-xy unif` et `-xy seed`.
- Affichage de l'écart type dans les stats affichées par `-check routing`.
- Affichage des erreurs sur `stderr` plutôt que `stdout`.
- Nouveau graphe : `suzuki`.

v4.3, juin 2016

- Nouveaux graphes : `triplex`, `jaws`, `starfish`.
- Suppression d'un bug (Abort trap: 6) pour `-format dot<type>`.
- Modification de l'initialisation du générateur aléatoire qui pouvait faire comme `-seed 0` sur certains systèmes où `clock()` est toujours nulle.
- Ajout de variantes pour `-check routing cluster`.
- Nouvelles options : `-xy cycle`, `-xy disk`, `-check stretch`, `-xy surface`.
- Modification de l'option `-norm` (`-norm 2` → `-norm L2`, etc.).
- Renommage de `-xy polar` en `-xy disk`.

v4.4, juillet 2016

- Nouvelle option : `-check simplify`.
- Nouveaux graphes : `schlafli`, `doily`, `fork` *(= tgraph)*.
- Suppression d'un bug pour `gosset` (mauvais paramétrage).

v4.5, août 2016

- Modification du terminateur de séquence pour `bdrg` et `fdrg` : `-1` devient `.`.
- Refonte du prototype des fonctions d'adjacence : `adj(i,j)` → `adj(Q)`.
- Désactivation des options `-apex` et `-star` (et donc de `caterpillar`).
- Finalisation de l'implémentation de `fdrg` (et donc de `regular` et `cubic`).
- Redéfinition des graphes : `netgraph`, `egraph`, `sunlet`, `cross`, `tgraph`, `ygraph`.
- Nouveaux graphes : `comb`, `alkane` (et ses nombreuses variantes), `banana`, `rlt`, `circle`.
- L'option `-fast` est effective pour `fdrg` et `bdrg`.
- Suppression d'un bug affectant `-xy mesh`.
- Correction du graphe `interval` qui renvoyait en fait un `circle`.
- Génération d'une aide HTML à partir du source : `gengraph.html`.
- Nouvelle option : `-norm poly <p>`.

v4.6, septembre 2016

- Nouvelle option : `-check routing scenario until <s>`.
- Nouveaux graphes : `uno`, `unok`, `behrend`.

v4.7, janvier 2017

- Dans `-check bfs/dfs`, détection d'une source invalide.
- Dans `-check dfs`, calcul et affichage de la hauteur des sommets.
- Ajout d'un paramètre pour `unok`.

v4.8, mars 2017

- Calcul dans `-check stretch` du stretch max. minimum.
- Correction d'un bug dans `-xy convex` qui pouvait produire des ensembles non convexes et des points en dehors du cercle de rayon 1.
- Nouveau graphe : `ngon`.

v4.9, juin 2017

- Correction d'un bug pour l'option `-xy box` et amélioration du dessin de la grille pour `-xy grid`.
- Généralisation de `-check bellman` aux graphes non valués.
- Renommage des options : `-dotfilter` → `-dot filter`, `-len` → `-dot len`, `-filter hyperbol` → `-filter hyper`.
- Nouvelles options : `-norm hyper`, `-xy hyper`, `-dot scale`, `-label <b>` avec b=3 et b<0, `-check volm`.
- Nouveaux graphes : `hudg`, `hyperbolic`.

v5.0, août 2017

- Nouveaux graphes : `helm`, `haar`.
- Nouveau format HTML utilisant vis.js (voir `-format`).

v5.1, août 2018

- Redéfinition (plus simple) du graphe `turan`.
- Redéfinition de `bidiakis` comme graphe `cage`.
- Redéfinition de `centipede` comme graphe `comb` (et sans `-star`).
- Simplification de `-check bellman` avec optimisation pour utilisations multiples comme dans `-check stretch`.
- Amélioration de détails dans l'aide HTML (blocs `!!!` et `Ex :`).
- Changement des paramètres (maintenant avec un `.` final) pour : `grid`, `aqua`, `rpartite`, `cage`, `ring`, `ggosset`.

- Orientation corrigée avec `-directed` pour les graphes (et leurs composés) utilisant une représentation implicite : `kout`, `ktree`, `kpath`, `kpage`, `hyperbolic`, `cactus`, `planar`, `rarytree`, `apollonian`, `expander`, `polygon`.
- Orientation supportée pour `cage` et `ring` (possibilité de cordes<0).
- Nouveaux graphes : `margulis`, `collatz`, `syracuse`, `mst`, `farkas`, `kakutami_[3|5|7]x+1` (3 graphes), `[w|u][psl|dis][d]` (8 graphes).
- Redéfinition de l'option `-loop` avec suppression de `-noloop`.
- Inversion des lignes/colonnes dans `uno` et `unok`.
- Redéfinition du `cuboctahedron` à partir de `linial`.
- Aide sur le graphe `chvatal` qui était absente.
- `-format xy` : affichage de coordonnées entières si elles le sont.
- Bug corrigé pour `-label <b>` avec b<0 : affiche exclusif centré ou pas.
- Bug corrigé pour `-xy zero` et `-xy bord` apparu à la v4.9.
- Définitions des graphes oubliés `mobius-kantor` et `dragon`.
- Utilisation d'une version uniforme pour remplacer `random()%k`.
- Nouvelle option : `-dot scale auto`.

v5.2, juin 2019

- Nouveaux graphes : `dart`, `kite`, `domino`, `hourglass`, `antenna`, `parachute`, `parapluie`, `klein`.
- Graphes `alkane` : nouvelles abréviations, option `-label 1` activée par défaut, correction d'un bug pour le graphe `methane`.
- Chargement conditionnel des scripts JS dans le format HTML.
- Nouvelles options : `-visuh`, `-check subdiv`.
- Correction d'un bug pour `unok`, introduit à la version v5.1.
- Correction d'un bug pour `-check stretch` (Bellman-Ford avec appels multiples).
- Correction d'un bug dans `nb_edges()` introduit à la version v5.1, et qui affectait par exemple `-check edges`.
- Correction bug dans la détection de séquence graphique qui pouvait affecter `fdrg` et ses composés `regular` et `cubic`.

v5.3, janvier 2021

- Nouveaux graphes : `knng`, `rectree`, `odd`, `biggs`, `biggs-smith`.
- Redéfinition de `nng` à partir de `knng`.
- Renommage de `headwood` et `headwood4` en `heawood` et `heawood4`.
- Nouvelles options : `-check prune`, `-dot dir`.

v5.4, février 2023

- Nouveaux graphes : `tree_fibo`, `tree_part`, `tree_binom`.

- Nouvelles options : `-dot attr`, `-check ball`.
- Génération rapide (`-fast`) pour tous les graphes basés sur une k-orientation et unification pour ceux basés sur `load` (soit au total +15 graphes).
- Correction d'un bug pour `arboricity n k`, introduit à la version v5.1, qui fixait k=1.
- Ajout du synonyme `-check span` pour `-check sub`.
- Ajout de l'aide pour `-check isub`.

v6.0, avril 2024

Cette version est le fruit du travail conséquent d'un étudiant en stage (Sylvain Chiron). Elle apporte une innovation majeure : le traitement des graphes en notation polonaise inversée. Une seule invocation de GenGraph permet ainsi de générer tous les graphes que l'on souhaite et d'enchainer des traitements dessus. De plus, l'étudiant a souhaité travailler sur le caractère convivial du projet, aussi bien pour les utilisateurs que pour les développeurs. Cela se manifeste par la création d'un dépôt du code source (lien en haut du manuel) découpé en de nombreux fichiers, des procédures automatisées pour la compilation, le débogage et l'installation, du travail sur l'interactivité du programme et l'amélioration du format de l'aide. En outre, quelques nouveaux algorithmes de graphes sont disponibles.

Notez que cette nouvelle version apporte des changements syntaxiques importants, notamment : les options qui prenaient un fichier en paramètre n'en prennent plus, l'ancienne option `-filter` est découpée en plusieurs options et l'ordre des commandes devient très déterminant.

La liste complète des changements est donnée dans les catégories ci-dessous.

Graphes :

- L'application peut désormais générer et gérer plusieurs graphes à la fois : voir LECTURE DE LA LIGNE DE COMMANDE.
- Un nouveau graphe `caterpillar` remplace l'ancien algorithme de graphe chenille (désactivé depuis la v4.5), qui n'était pas uniforme.
- Correction du graphe `parapluie` qui n'acceptait aucun paramètre.
- Ajout du graphe `triangle` et du synonyme `complete (= clique)`.
- Ajout de variantes `-fast` pour les graphes `ring`, `grid` et `rpartite`.

Algorithmes :

- Nouvelles options : `-extract`, `-extract-all`, `-print-prop`, `-print-test`, `-prop`, `-sort[-inv]`, `-test`, `-while`.
- L'option `-filter` est modifiée pour utiliser et affecter la nouvelle pile, et reposer sur les options génériques `-test` et `-prop`. Elle ne permet plus d'afficher les valeurs, ce qui est maintenant le rôle de l'option `-print-prop`.
- `-variant <v>` n'affecte plus que les graphes et les opérations. Pour les algorithmes, il faut utiliser l'écriture `-check variant <v>`.

- Les options `-check` qui prenaient un fichier de graphe en paramètre (`-check iso/sub/isub/minor`) tirent maintenant ce dernier de la pile.
- `-check radius/diameter/girth` deviennent des synonymes de la combinaison `-prop ... -print-prop -output -` équivalente.
- Nouveaux algorithmes : `-check clique`, `-prop clique`.
- Les « intervalles » de `-test` sont maintenant appelés « sélecteurs ». Les nouveaux codes `<=x`, `>=x`, `%x` et `%x=y` ont été ajoutés.

Opérations :

- L'option `-op` permet de placer des opérations sur la pile de requêtes.
- `-op simplify`, `-op maincc`, `-op subdiv` et `-op prune` supplantent les options de `-check` du même nom pour plus de fonctionnalités.
- Nouvelles opérations : `-op accessible`, `-op apex`, `-op cc+`, `-op del-e`, `-op del-v`, `-op half`, `-op line-graph`, `-op line-graph-root`, `-op maincc`, `-op not`, `-op not-wl`, `-op star`, `-op transpose`, `-op union`, `-op union-d` et `-op union-de`.

Options diverses :

- Nouvelles options : `-chrono`, `-chrono-reset`, `-discard`, `-dup`, `-dup-group`, `-forget-ids`, `-gen[c]`, `-group`, `-group-n`, `-id`, `-n-times`, `-pause`, `-print`, `-quit`, `-shift-ids`, `-ungroup`, `-xy none`, `-xy-as-default`.
- Suppression de l'option `-shift` (rendue obsolète par `-op union-d` et `-output-union-d`).
- Les options `-xy vsize/grid/zero` sont maintenant des options de `-view`, et `-pos` est remplacé par `-view pos`.

Formats :

- Nouvelles options : `-format auto`, `-output`, `-output-group`, `-output-union-d`, `-view border`, `-view no`, `-visu-as`.
- `load` lit maintenant correctement tous les graphes sur l'entrée standard. De plus, le nouveau graphe `load-str` permet de charger un graphe depuis sa description au format simple donnée directement en paramètre.

  Ex :
  ```
  echo '0-1-(2 3) 4-5' | gengraph load -
  gengraph load-str '0-1-(2 3) 4-5'
  ```

- Des groupes de graphes peuvent être directement chargés avec `load+` et `load-str+`.
- Option `-format dot<type>` renommée en `-format dot-<type>`.
- Le format standard est maintenant plutôt appelé format simple (voir LE FORMAT PAR DÉFAUT). Il a de nouveaux alias : `simple`, `default`, `classic` et `edges`.
- De nouveaux sucres sont pris en charge pour la lecture de graphes au format simple (`i-*`, `i->*` et `*->i`).
- Les options `-visu` et `-visuh` sont supplantées par `-format auto`, `-output`, `-visu` et `-visu-as`.

- `-vcolor`, `-vsize`, `-view vsize` et `-caption` fonctionnent maintenant avec le format HTML, sauf `-vcolor list`.

Dépôt :

- Création d'un dépôt Git du projet.
- Découpage du code en de nombreux fichiers.
- Adoption explicite d'une licence de logiciel libre : CeCILL-C.
- Ajout de lisez-moi en anglais et en français expliquant notamment comment installer le logiciel.

Compilation et installation :

- Ajout d'un `Makefile` pour compiler le projet et l'aide HTML.
- Création de `tools/dependencies.c` pour ajouter la dépendance libbsd à la compilation si utilisation de glibc < 2.36.
- Ajout de la cible `debug` pour faciliter le débogage.
- Ajout de cibles `merge` et `split` (ainsi que `split-all` et `split-debug`) dans le `Makefile`, et de `tools/gengraph_c-split.c` et `tools/gengraph_c-compile.c` pour pouvoir travailler avec un fichier combiné `gengraph.c` comme avant.
- Ajout de cibles `install` et `uninstall` dans le `Makefile` permettant d'installer et de désinstaller GenGraph.

Aide et interactivité :

- Nouvelles options : `-html`, `-list-options`, `-list-graphs`.
- L'aide intégrée et l'aide HTML utilisent un noyau commun en langage C pour l'analyse et le traitement.
- Ajout de formatage dans l'aide (notamment des couleurs dans l'aide intégrée) et les messages d'erreur.
- Ajout d'un sommaire et de liens dans l'aide HTML.
- L'aide intégrée ne fait plus appel aux programmes `sed`, `grep`, `awk`, `sort` et `more` ; elle utilise le programme `less` si disponible.
- Ajout de la vérification du format des nombres dans les paramètres.
- Ajout de règles dans l'aide HTML pour un meilleur rendu à l'impression (vers PDF ou papier).
- La complétion pour Bash est disponible.

v6.1, octobre 2024

- Nouveaux graphes : `half-graph`, `mikado`, `random-wl`, `clique-wl`, `tournament`.
- Nouvelle variante pour `tree_part` : `-variant 2`.
- Chaque requête a maintenant son propre générateur de nombres aléatoires, de sorte à rendre la génération aléatoire plus stable et contrôlable.
- L'option `-seed` devient `-main-seed`. Ajout des options `-seed`, `-print-seeds` et `-seed-bits`.

- Nouvelles opérations : `-op loop`, `-op noloop`, `-op permute`, `-op redirect`, `-op blow-up`, `-op union-s`.
- Suppression des options `-not`, `-loop`, `-dele`, `-delv`, `-permute` et `-redirect`, remplacées par les nouvelles opérations. Notez que le nouveau `-op permute` risque de ralentir sensiblement la génération et que le nouveau `-op redirect` requiert que le graphe soit chargé en mémoire.
- Les boucles sont désormais conservées par défaut même pour les graphes non orientés. Utiliser `-op noloop` pour les supprimer.
- Suppression de `-genc` et `loadc`, qui désormais n'apportent plus rien par rapport au mode `-fast`.
- L'option `-gen` est renommée en `-store`.
- Les options `-check` n'affichent plus le graphe.
- Nouveau format : `-format latex-tkzgraph`.
- Les formats `matrix`, `smatrix`, `list` et `vertex` ne requièrent plus que le graphe soit stocké en mémoire.
- Correction du calcul de `-norm hyper` et de `gabriel`.
- Nouvelle option `-norm Lp p`.

v6.2, novembre 2024

- Le mode `-fast` ne stocke plus le graphe en mémoire.
- Suppression d'`-output-union-d`, qui ne présente plus d'intérêt par rapport à `-op union-d -fast`.
- `-check info` affiche les fonctionnalités offertes par le graphe.
- Ajout de la génération par liste d'adjacence (`-fast`) pour `-op union-s`, `crown`, `half-graph`, `prime`, `cage`.
- Correction d'un bug important : les algorithmes `-check` ne trouvaient plus les données géométriques des graphes (depuis la v6.0).
- Correction d'un bug : plantage lors de la combinaison de `-fast` et d'une option `-vcolor` (depuis la v6.1).
- Correction d'un bug : le positionnement des sommets de `rlt` était erroné (depuis la v6.0).
- Nouveaux graphes : `null (= empty 0)`, `calls_tree`, `king p q`.
- Nouvelle opération : `-op iso`.
- Suppression du cas non planaire pour `gabriel` (points co-cycliques et diamétralement opposés créant des cliques).
- Restauration de `-[no]loop` et suppression de `-op not-wl` ainsi que des deux graphes basés dessus.
- Ajout de la possibilité de spécifier une profondeur à `-label`.
- Remplacement de `-format latex-tkzgraph` par `-format tikz`, et de `-dot scale` par `-view scale`.
- Redéfinition de `-xy mesh`, `tree_fibo`, `tree_binom`, `diamond`, `gem`, `fan`, `split`, `egraph`, `comb`, `sunlet`.

- Nouvelles options : `-xy mesh-x`, `-xy mesh-y`, `-xy tree`.

v6.3, février 2025

- Nouveau graphe : `sum-graph`.
- Redéfinition des graphes : `dart`, `half-graph`.
- Changements mineurs dans `mikado` et `kneser`.
- Correction d'un bug pour `johnson` et pour `octahedron`.
- Nouvelles opérations : `-op cartesian-prod`, `-op tensor-prod`, `-op subst-e`, `-op subst-v`, `-op diff`, `-op inter`.
- Remplacement de `-vsize` par `-vsize deg`. Ajout des options `-vcolor none` et `-vsize none`.
- Ajout du mode `-fast` et d'une variante pour `line-graph`.