Examples

Hello World ¶

These examples use the Python 3 interface for the software. After each run a PDF summary is compiled. The content can be specified via the Python script.

```
# Normal printing to the terminal:
print("Hello world")
# Make some headers in the summary:
postChapter("Hello")
postSection("World")
# Load a moleucle from a SMILES string:
mol = smiles("Cn1cnc2c1c(=O)n(c(=O)n2C)C", name="Caffeine")
# Put a visualisation of the molecule in the summary:
mol.print()
```

Graph Loading

Molecules are encoded as labelled graphs. They can be loaded from SMILES strings, and in general any graph can be loaded from a GML specification, or from the SMILES-like format GraphDFS.

```
# Load a graph from a SMILES string (only for molecule graphs):
ethanol1 = smiles("CCO", name="Ethanol1")
# Load a graph from a SMILES-like format, called "GraphDFS", but for general graphs:
ethanol2 = graphDFS("[C]([H])([H])([H])[C]([H])([H])[O][H]", name="Ethanol2")
# The GraphDFS format also supports implicit hydrogens:
ethanol3 = graphDFS("CCO", name="Ethanol3")
# The basic graph format is GML:
ethanol4 = graphGMLString("""graph [
   node [ id 0 label "C" ]
   node [ id 1 label "C" ]
   node [ id 2 label "O" ]
   node [ id 3 label "H" ]
   node [ id 4 label "H" ]
   node [ id 5 label "H" ]
   node [ id 6 label "H" ]
   node [ id 7 label "H" ]
   node [ id 8 label "H" ]
   edge [ source 1 target 0 label "-" ]
   edge [ source 2 target 1 label "-" ]
   edge [ source 3 target 0 label "-" ]
   edge [ source 4 target 0 label "-" ]
   edge [ source 5 target 0 label "-" ]
   edge [ source 6 target 1 label "-" ]
   edge [ source 7 target 1 label "-" ]
   edge [ source 8 target 2 label "-" ]
]""", name="Ethanol4")
# They really are all loading the same graph into different objects:
assert ethanol1.isomorphism(ethanol2) == 1
assert ethanol1.isomorphism(ethanol3) == 1
assert ethanol1.isomorphism(ethanol4) == 1
# and they can be visualised:
ethanol1.print()
# All loaded graphs are added to a list 'inputGraphs':
for g in inputGraphs:
   g.print()
```

Printing Graphs/Molecules

The visualisation of graphs can be “prettified” using special printing options. The changes can make the graphs look like normal molecule visualisations.

```
# Our test graph, representing the molecule caffeine:
g = smiles('Cn1cnc2c1c(=O)n(c(=O)n2C)C')
# ;ake an object to hold our settings:
p = GraphPrinter()
# First try visualising without any prettifications:
p.disableAll()
g.print(p)
# Now make chemical edges look like bonds, and put colour on atoms.
# Also put the "charge" part of vertex labels in superscript:
p.edgesAsBonds = True
p.raiseCharges=True
p.withColour = True
g.print(p)
# We can also "collapse" normal hydrogen atoms into the neighbours,
# and just show a count:
p.collapseHydrogens = True
g.print(p)
# And finally we can make "internal" carbon atoms simple lines:
p.simpleCarbons = True
g.print(p)
# There are also options for adding indices to the vertices,
# and modify the rendering of labels and edges:
p2 = GraphPrinter()
p2.disableAll()
p2.withTexttt = True
p2.thick = True
p2.withIndex = True
# We can actually print two different versions at the same time:
g.print(p2, p)
```

Graph Interface

Graph objects have a full interface to access individual vertices and edges. The labels of vertices and edges can be accessed both in their raw string form, and as their chemical counterpart (if they have one).

```
g = graphDFS("[R]{x}C([O-])CC=O")
print("|V| =", g.numVertices)
print("|E| =", g.numEdges)
for v in g.vertices:
   print("v%d: label='%s'" % (v.id, v.stringLabel), end="")
   print("\tas molecule: atomId=%d, charge=%d" % (v.atomId, v.charge), end="")
   print("\tis oxygen?", v.atomId == AtomIds.Oxygen)
   print("\td(v) =", v.degree)
   for e in v.incidentEdges:
      print("\tneighbour:", e.target.id)
for e in g.edges:
   print("(v%d, v%d): label='%s'" % (e.source.id, e.target.id, e.stringLabel), end="")
   try:
      bt = str(e.bondType)
   except LogicError:
      bt = "Invalid"
   print("\tas molecule: bondType=%s" % bt, end="")
   print("\tis double bond?", e.bondType == BondType.Double)
```

Graph Morphisms

Graph objects have methods for finding morphisms with the VF2 algorithms for isomorphism and monomorphism. We can therefore easily detect isomorphic graphs, count automorphisms, and search for substructures.

```
mol1 = smiles("CC(C)CO")
mol2 = smiles("C(CC)CO")
# Check if there is just one isomorphism between the graphs:
isomorphic = mol1.isomorphism(mol2) == 1
print("Isomorphic?", isomorphic)
```

```
print("Isomorphic?", isomorphic)
# Find the number of automorphisms in the graph,
# by explicitly enumerating all of them:
numAutomorphisms = mol1.isomorphism(mol1, maxNumMatches=2**30)
print("|Aut(G)| =", numAutomorphisms)
# Let's count the number of methyl groups:
methyl = smiles("[CH3]")
# The symmetry of the group it self should not be counted,
# so find the size of the automorphism group of methyl.
numAutMethyl = methyl.isomorphism(methyl, maxNumMatches=2**30)
print("|Aut(methyl)|", numAutMethyl)
# Now find the number of methyl matches,
numMono = methyl.monomorphism(mol1, maxNumMatches=2**30)
print("#monomorphisms =", numMono)
# and divide by the symmetries of methyl.
print("#methyl groups =", numMono / numAutMethyl)
```

Rule Loading

Rules must be specified in GML format.

```
# A rule (L <- K -> R) is specified by three graph fragments:
# left, context, and right
destroyVertex = ruleGMLString("""rule [
   left [
      node [ id 1 label "A" ]
   ]
]""")
createVertex = ruleGMLString("""rule [
   right [
      node [ id 1 label "A" ]
   ]
]""")
identity = ruleGMLString("""rule [
   context [
      node [ id 1 label "A" ]
   ]
]""")
# A vertex/edge can change label:
labelChange = ruleGMLString("""rule [
   left [
      node [ id 1 label "A" ]
      edge [ source 1 target 2 label "A" ]
   ]
   # GML can have Python-style line comments too
   context [
      node [ id 2 label "Q" ]
   ]
   right [
      node [ id 1 label "B" ]
      edge [ source 1 target 2 label "B" ]
   ]
]""")
# A chemical rule should probably not destroy and create vertices:
ketoEnol = ruleGMLString("""rule [
   left [
      edge [ source 1 target 4 label "-" ]
      edge [ source 1 target 2 label "-" ]
      edge [ source 2 target 3 label "=" ]
   ]
   context [
      node [ id 1 label "C" ]
      node [ id 2 label "C" ]
      node [ id 3 label "O" ]
      node [ id 4 label "H" ]
   ]
   right [
      edge [ source 1 target 2 label "=" ]
      edge [ source 2 target 3 label "-" ]
      edge [ source 3 target 4 label "-" ]
   ]
]""")
# Rules can be printed, but label changing edges are not visualised in K:
ketoEnol.print()
# Add with custom options, like graphs:
p1 = GraphPrinter()
p2 = GraphPrinter()
p1.disableAll()
p1.withTexttt = True
p1.withIndex = True
p2.setReactionDefault()
for p in inputRules:
   p.print(p1, p2)
# Be careful with printing options and non-existing implicit hydrogens:
p1.disableAll()
p1.edgesAsBonds = True
p2.setReactionDefault()
p2.simpleCarbons = True # !!
ketoEnol.print(p1, p2)
```

Rule Morphisms

Rule objects, like graph objects, have methods for finding morphisms with the VF2 algorithms for isomorphism and monomorphism. We can therefore easily detect isomorphic rules, and decide if one rule is at least as specific/general as another.

```
# A rule with no extra context:
small = ruleGMLString("""rule [
   ruleID "Small"
   left [
      node [ id 1 label "H" ]
      node [ id 2 label "O" ]
      edge [ source 1 target 2 label "-" ]
   ]
   right [
      node [ id 1 label "H+" ]
      node [ id 2 label "O-" ]
   ]
]""")
# The same rule, with a bit of context:
large = ruleGMLString("""rule [
   ruleID "Large"
   left [
      node [ id 1 label "H" ]
      node [ id 2 label "O" ]
      edge [ source 1 target 2 label "-" ]
   ]
   context [
      node [ id 3 label "C" ]
      edge [ source 2 target 3 label "-" ]
   ]
   right [
      node [ id 1 label "H+" ]
      node [ id 2 label "O-" ]
   ]
]""")
isomorphic = small.isomorphism(large) == 1
print("Isomorphic?", isomorphic)
atLeastAsGeneral = small.monomorphism(large) == 1
print("At least as general?", atLeastAsGeneral)
```

Formose Grammar

The graph grammar modelling the formose chemistry.

```
formaldehyde = smiles("C=O", name="Formaldehyde")
glycolaldehyde = smiles( "OCC=O", name="Glycolaldehyde")
ketoEnolGML = """rule [
   ruleID "Keto-enol isomerization"
   left [
      edge [ source 1 target 4 label "-" ]
      edge [ source 1 target 2 label "-" ]
      edge [ source 2 target 3 label "=" ]
   ]
   context [
      node [ id 1 label "C" ]
      node [ id 2 label "C" ]
      node [ id 3 label "O" ]
      node [ id 4 label "H" ]
   ]
   right [
      edge [ source 1 target 2 label "=" ]
      edge [ source 2 target 3 label "-" ]
      edge [ source 3 target 4 label "-" ]
   ]
]"""
ketoEnol_F = ruleGMLString(ketoEnolGML)
ketoEnol_B = ruleGMLString(ketoEnolGML, invert=True)
aldolAddGML = """rule [
   ruleID "Aldol Addition"
   left [
      edge [ source 1 target 2 label "=" ]
      edge [ source 2 target 3 label "-" ]
      edge [ source 3 target 4 label "-" ]
      edge [ source 5 target 6 label "=" ]
   ]
   context [
      node [ id 1 label "C" ]
      node [ id 2 label "C" ]
      node [ id 3 label "O" ]
      node [ id 4 label "H" ]
      node [ id 5 label "O" ]
      node [ id 6 label "C" ]
   ]
   right [
      edge [ source 1 target 2 label "-" ]
      edge [ source 2 target 3 label "=" ]
      edge [ source 5 target 6 label "-" ]

      edge [ source 4 target 5 label "-" ]
      edge [ source 6 target 1 label "-" ]
   ]
]"""
aldolAdd_F = ruleGMLString(aldolAddGML)
aldolAdd_B = ruleGMLString(aldolAddGML, invert=True)
```

Including Files

We can include other files (à la C/C++) to seperate functionality.

```
include("../examples/050_formoseGrammar.py")
postSection("Input Graphs")
for a in inputGraphs:
   a.print()
postSection("Input Rules")
for a in inputRules:
   a.print()
```

Rule Composition 1 — Unary Operators

Special rules can be constructed from graphs.

```
include("../examples/050_formoseGrammar.py")
glycolaldehyde.print()
# A graph G can be used to construct special rules:
# (\emptyset <- \emptyset -> G)
bindExp = rcBind(glycolaldehyde)
# (G <- \emptyset -> \emptyset)
unbindExp = rcUnbind(glycolaldehyde)
# (G <- G -> G)
idExp = rcId(glycolaldehyde)
# These are really rule composition expressions that have to be evaluated:
rc = rcEvaluator(inputRules)
# Each expression results in a lists of rules:
bindRules = rc.eval(bindExp)
unbindRules = rc.eval(unbindExp)
idRules = rc.eval(idExp)
postSection("Bind Rules")
for p in bindRules:
   p.print()
postSection("Unbind Rules")
for p in unbindRules:
   p.print()
postSection("Id Rules")
for p in idRules:
   p.print()
```

Rule Composition 2 — Parallel Composition

A pair of rules can be merged to a new rule implementing the parallel transformation.

```
include("../examples/050_formoseGrammar.py")
rc = rcEvaluator(inputRules)
# The special global object 'rcParallel' is used to make a pseudo-operator:
exp = rcId(formaldehyde) *rcParallel*  rcUnbind(glycolaldehyde)
rules = rc.eval(exp)
for p in rules:
   p.print()
```

Rule Composition 3 — Supergraph Composition

A pair of rules can (maybe) be composed using a sueprgraph relation.

```
include("../examples/050_formoseGrammar.py")
rc = rcEvaluator(inputRules)
exp = rcId(formaldehyde) *rcParallel*  rcId(glycolaldehyde)
exp = exp *rcSuper* ketoEnol_F
rules = rc.eval(exp)
for p in rules:
   p.print()
```

Rule Composition 4 — Overall Formose Reaction

A complete pathway can be composed to obtain the overall rules.

```
include("../examples/050_formoseGrammar.py")
rc = rcEvaluator(inputRules)
exp = (
   rcId(glycolaldehyde)
```

```
   rcId(glycolaldehyde)
   *rcSuper* ketoEnol_F
   *rcParallel* rcId(formaldehyde)
   *rcSuper(allowPartial=False)* aldolAdd_F
   *rcSuper* ketoEnol_F
   *rcParallel* rcId(formaldehyde)
   *rcSuper(allowPartial=False)* aldolAdd_F
   *rcSuper* ketoEnol_F
   *rcSuper* ketoEnol_B
   *rcSuper* aldolAdd_B
   *rcSuper* ketoEnol_B
   *rcSuper(allowPartial=False)*
   (rcId(glycolaldehyde) *rcParallel* rcId(glycolaldehyde))
)
rules = rc.eval(exp)
for p in rules:
   p.print()
```

Reaction Networks 1 — Rule Application

Transformation rules (reaction patterns) can be applied to graphs (molecules) to create new graphs (molecules). The transformations (reactions) implicitly form a directed (multi-)hypergraph (chemical reaction network).

```
include("../examples/050_formoseGrammar.py")
# Reaction networks are expaned using a strategy:
strat = (
   # A molecule can be active or passive during evaluation.
   addUniverse(formaldehyde) # passive
   >> addSubset(glycolaldehyde) # active
   # Aach reaction must have a least 1 active educt.
   >> inputRules
)
# We call a reaction network a 'derivation graph'.
dg = dgRuleComp(inputGraphs, strat)
dg.calc()
# They can also be visualised.
dg.print()
```

Reaction Networks 2 — Repetition

A sub-strategy can be repeated.

```
include("../examples/050_formoseGrammar.py")
strat = (
   addUniverse(formaldehyde)
   >> addSubset(glycolaldehyde)
   # Iterate the rule application 4 times.
   >> repeat[4](
      inputRules
   )
)
dg = dgRuleComp(inputGraphs, strat)
dg.calc()
dg.print()
```

Reaction Networks 3 — Application Constraints

We may want to impose constraints on which reactions are accepted. E.g., in formose the molecules should not have too many carbon atoms.

```
include("../examples/050_formoseGrammar.py")
strat = (
   addUniverse(formaldehyde)
   >> addSubset(glycolaldehyde)
   # Constrain the reactions:
   # No molecules with more than 20 atom can be created.
   >> rightPredicate[
      lambda derivation: all(g.numVertices <= 20 for g in derivation.right)
   ](
      # Iterate until nothing new is found.
      repeat(
         inputRules
      )
   )
)
dg = dgRuleComp(inputGraphs, strat)
dg.calc()
dg.print()
```

Advanced Printing

Reaction networks can become large, and often it is necessary to hide parts of the network, or in general change the appearance.

```
include("../examples/212_dgPredicate.py")
# Create a printer with default options:
p = DGPrinter()
# Hide "large" molecules: those with > 4 Cs:
p.pushVertexVisible(lambda m, dg: m.vLabelCount("C") <= 4)
# Hide the reactions with the large molceules as well:
def dRefEval(dRef):
   der = dRef.derivation
   if any(m.vLabelCount("C") > 4 for m in der.left): return False
   if any(m.vLabelCount("C") > 4 for m in der.right): return False
   return True
p.pushEdgeVisible(dRefEval)
# Add the number of Cs to the molecule labels:
p.pushVertexLabel(lambda m, dg: "\\#C=" + str(m.vLabelCount("C")))
# Highlight the molecules with 4 Cs:
p.pushVertexColour(lambda m, dg: "blue" if m.vLabelCount("C") == 4 else "")
# Print the network with the customised printer.
dg.print(p)
```

Double Pushout Printing

Each reaction/derivation can be visualised as a DPO diagram.

```
include("../examples/212_dgPredicate.py")
for dRef in dg.derivations:
   dRef.print()
```

Stereospecific Aconitase

Modelling of the reaction performed by the aconitase enzyme in the citric acid cycle: citrate to D-isocitrate. The rule implements the stereo-specificity of the reaction.

```
water = smiles("O", "H_2O")
cit = smiles("C(C(=O)O)C(CC(=O)O)(C(=O)O)O", name="Cit")
d_icit = smiles("C([C@@H]([C@H](C(=O)O)O)C(=O)O)C(=O)O", name="D-ICit")

aconitase = ruleGMLString("""rule [
   ruleID "Aconitase"
   left [
      # the dehydrated water
      edge [ source 1 target 100 label "-" ]
      edge [ source 2 target 102 label "-" ]
```

```
      # the hydrated water
      edge [ source 200 target 202 label "-" ]
   ]
   context [
      node [ id 1 label "C" ]
      edge [ source 1 target 2 label "-" ] # goes from - to = to -
      node [ id 2 label "C" ]
      # the dehydrated water
      node [ id 100 label "O" ]
      edge [ source 100 target 101 label "-" ]
      node [ id 101 label "H" ]
      node [ id 102 label "H" ]
      # the hydrated water
      node [ id 200 label "O" ]
      edge [ source 200 target 201 label "-" ]
      node [ id 201 label "H" ]
      node [ id 202 label "H" ]
      # dehydrated C neighbours
      node [ id 1000 label "C" ]
      edge [ source 1 target 1000 label "-" ]
      node [ id 1010 label "O" ]
      edge [ source 1000 target 1010 label "-" ]
      node [ id 1001 label "C" ]
      edge [ source 1 target 1001 label "-" ]
      # hydrated C neighbours
      node [ id 2000 label "C" ]
      edge [ source 2 target 2000 label "-" ]
      node [ id 2001 label "H" ]
      edge [ source 2 target 2001 label "-" ]
   ]
   right [
      # The '!' in the end changes it from TetrahedralSym to
      # TetrahedralFixed
      node [ id 1 stereo "tetrahedral[1000, 1001, 202, 2]!" ]
      node [ id 2 stereo "tetrahedral[200, 1, 2000, 2001]!" ]
      # the dehydrated water
      edge [ source 100 target 102 label "-" ]
      # the hydrated water
      edge [ source 1 target 202 label "-" ]
      edge [ source 2 target 200 label "-" ]
   ]
]""")

dg = dgRuleComp(
   inputGraphs,
   addSubset(cit, water) >> aconitase,
   labelSettings=LabelSettings(
      LabelType.Term,
      LabelRelation.Specialisation,
      LabelRelation.Specialisation)
)
dg.calc()
for e in dg.edges:
   p = GraphPrinter()
   p.withColour = True
   e.print(p, matchColour="Maroon")
```

Stereoisomers of Tartaric Acid

Generation of stereoisomers of tartaric acid, starting from a model without stereo-information and fixating each tetrahedral embedding.

```
smiles("C(C(C(=O)O)O)(C(=O)O)O", name="Tartaric acid")
smiles("[C@@H]([C@H](C(=O)O)O)(C(=O)O)O", name="L-tartaric acid")
smiles("[C@H]([C@@H](C(=O)O)O)(C(=O)O)O", name="D-tartaric acid")
smiles("[C@@H]([C@@H](C(=O)O)O)(C(=O)O)O", name="Meso-tartaric acid")
change = ruleGMLString("""rule [
   ruleID "Change"
   left [
      node [ id 0 stereo "tetrahedral" ]
   ]
   context [
      node [ id 0 label "*" ]
      node [ id 1 label "*" ]
      node [ id 2 label "*" ]
      node [ id 3 label "*" ]
      node [ id 4 label "*" ]
      edge [ source 0 target 1 label "-" ]
      edge [ source 0 target 2 label "-" ]
      edge [ source 0 target 3 label "-" ]
      edge [ source 0 target 4 label "-" ]
   ]
   right [
      node [ id 0 stereo "tetrahedral[1, 2, 3, 4]!" ]
   ]
]""")

dg = dgRuleComp(
   inputGraphs,
   addSubset(inputGraphs) >> repeat(change),
   labelSettings=LabelSettings(
      LabelType.Term,
      LabelRelation.Specialisation,
      LabelRelation.Specialisation)
)
dg.calc()

p = GraphPrinter()
p.setMolDefault()
p.withPrettyStereo = True
change.print(p)
p = DGPrinter()
p.withRuleName = True
p.withRuleId = False
dg.print(p)
```

Non-trivial Stereoisomers

Generation of stereoisomers in a non-trivial molecule.

```
g = smiles("N[C@](O)([C@](S)(P)(O))([C@](S)(P)(O))")
change = ruleGMLString("""rule [
   ruleID "Change"
   left [
      node [ id 0 stereo "tetrahedral" ]
   ]
   context [
      node [ id 0 label "*" ]
      node [ id 1 label "*" ]
      node [ id 2 label "*" ]
      node [ id 3 label "*" ]
      node [ id 4 label "*" ]
      edge [ source 0 target 1 label "-" ]
      edge [ source 0 target 2 label "-" ]
      edge [ source 0 target 3 label "-" ]
      edge [ source 0 target 4 label "-" ]
   ]
   right [
      node [ id 0 stereo "tetrahedral[1, 2, 3, 4]!" ]
   ]
]""")

dg = dgRuleComp(
   inputGraphs,
   addSubset(inputGraphs) >> repeat(change),
   labelSettings=LabelSettings(
```

```
      LabelType.Term,
      LabelRelation.Specialisation,
      LabelRelation.Specialisation)
)
dg.calc()

p = GraphPrinter()
p.setMolDefault()
p.withPrettyStereo = True
change.print(p)
p = DGPrinter()
p.withRuleName = True
p.withRuleId = False
dg.print(p)
```

Finding Pathways 1 — A Specific Pathway

A Pathway is an integer hyper-flow: each reaction is assigned a non-negative interger, specifying the number of times the reaction is used. Virtual input and output reactions are added to each molecule.

```
include("../examples/212_dgPredicate.py")
# Use the derivation graph 'dg' already created:
flow = dgFlow(dg)
# Specify which molecules can be fed into the network:
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)
# Specify which molecules that can remain in the network:
flow.addSink(glycolaldehyde)
# Specify restrictions on the amount of input/output molecules:
flow.addConstraint(inFlow(formaldehyde) == 2)
flow.addConstraint(inFlow(glycolaldehyde) == 1)
flow.addConstraint(outFlow(glycolaldehyde) == 2)
# Specify the minimization criteria:
#  number of unique reactions used
flow.objectiveFunction = isEdgeUsed
# Find a solution:
flow.calc()
# Show solution information in the terminal:
flow.solutions.list()
# Print solutions:
flow.solutions.print()
```

Finding Pathways 2 — Extra Constraints

We can add many kinds of constraints. They do not need to be related to input/ouput.

```
include("../examples/212_dgPredicate.py")
# Use the derivation graph 'dg' already created:
flow = dgFlow(dg)
# Specify which molecules can be fed into the network:
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)
# Specify which molecules that can remain in the network:
flow.addSink(glycolaldehyde)
# Specify restrictions on the amount of input/output molecules:
flow.addConstraint(inFlow(formaldehyde) == 2)
flow.addConstraint(inFlow(glycolaldehyde) == 1)
flow.addConstraint(outFlow(glycolaldehyde) == 2)
# Disable too large molecules:
for m in dg.vertexGraphs:
   if m.vLabelCount("C") > 4:
      flow.addConstraint(vertex(m) == 0)
# Disable "strange" misleading input/output flows:
flow.allowIOReverse = False
# Specify the minimization criteria:
#  number of unique reactions used
flow.objectiveFunction = isEdgeUsed
# Find a solution:
flow.calc()
# Show solution information in the terminal:
flow.solutions.list()
# Print solutions:
flow.solutions.print()
```

Finding Pathways 3 — Multiple Solutions

It is often interesting to look for alternate solutions, possibly with a sub-optimal objective value.

```
include("../examples/212_dgPredicate.py")
# Use the derivation graph 'dg' already created:
flow = dgFlow(dg)
# Specify which molecules can be fed into the network:
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)
# Specify which molecules that can remain in the network:
flow.addSink(glycolaldehyde)
# Specify restrictions on the amount of input/output molecules:
flow.addConstraint(inFlow(formaldehyde) == 2)
flow.addConstraint(inFlow(glycolaldehyde) == 1)
flow.addConstraint(outFlow(glycolaldehyde) == 2)
# Disable "strange" misleading input/output flows:
flow.allowIOReverse = False
# Specify the minimization criteria:
#  number of reactions
flow.objectiveFunction = edge
# Enable solution enumeration:
#  at most 10 solutions, any quality
flow.setSolverEnumerateBy(maxNumSolutions=10)
# Find solutions:
flow.calc()
# Show solution information in the terminal:
flow.solutions.list()
# Print solutions:
flow.solutions.print()
```

Finding Autocatalytic Cycles

Some pathways have a specific higher-order structure, e.g., autocatalysis.

```
include("../examples/212_dgPredicate.py")
# Use the derivation graph 'dg' already created:
flow = dgFlow(dg)
# Specify which molecules can be fed into the network:
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)
# Specify which molecules that can remain in the network:
flow.addSink(glycolaldehyde)
# Enable constraints for autocatalysis:
flow.overallAutocatalysis.enable()
# Specify the minimization criteria:
#  number of unique reactions used
flow.objectiveFunction = isEdgeUsed
# Find a solution:
flow.calc()
# Show solution information in the terminal:
flow.solutions.list()
# Print solutions:
flow.solutions.print()
```