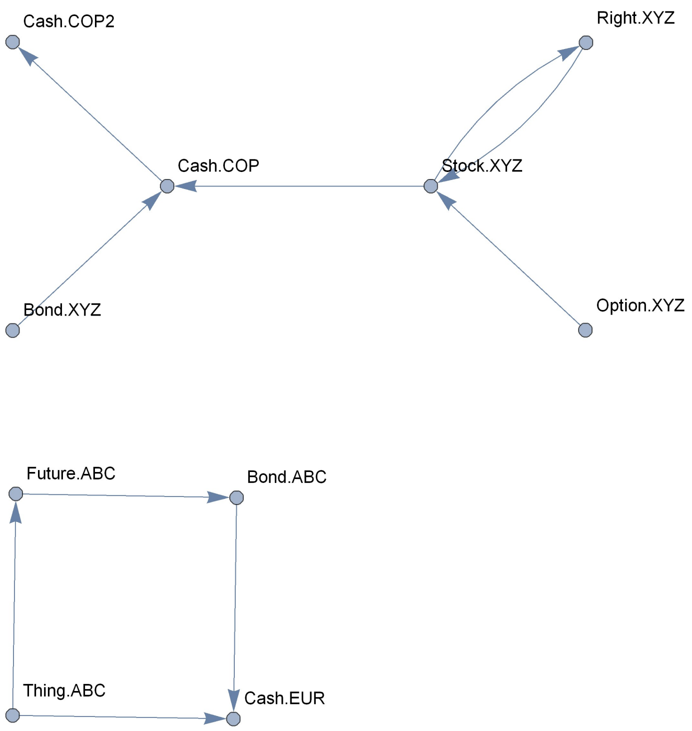
Assets implicated in a lasso

Graphs

Consider a list of assets, which implicate other assets, and the directed graph resulting:

In[ ]:=
```
g = Graph[{"Right.XYZ" ↦ "Stock.XYZ",
"Stock.XYZ" ↦ "Cash.COP",
"Stock.XYZ" ↦ "Right.XYZ",
"Cash.COP" ↦ "Cash.COP2",
"Bond.XYZ" ↦ "Cash.COP",
"Option.XYZ" ↦ "Stock.XYZ",
"Bond.ABC" ↦ "Cash.EUR",
"Future.ABC" ↦ "Bond.ABC",
"Thing.ABC" ↦ "Future.ABC",
"Thing.ABC" ↦ "Cash.EUR"}, VertexLabels → "Name"]
```

Out[ ]=



In[ ]:=
```
VertexList[g]
```

Out[ ]=
```
{Right.XYZ, Stock.XYZ, Cash.COP, Cash.COP2, Bond.XYZ,
 Option.XYZ, Bond.ABC, Cash.EUR, Future.ABC, Thing.ABC}
```

Adjacency matrix

The list of directly directed connections is the adjacency matrix:

In[ ]:= `AdjacencyMatrix[g] // Normal // MatrixForm`

Out[ ]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Determine connections of a vertex to other vertices

In[ ]:= `IncidenceList[g, "Cash.COP", 1]`

Out[ ]= {Stock.XYZ ⟶ Cash.COP, Cash.COP ⟶ Cash.COP2, Bond.XYZ ⟶ Cash.COP}

In[ ]:= `IncidenceList[g, "Cash.COP", 3]`

Out[ ]= {Right.XYZ ⟶ Stock.XYZ, Stock.XYZ ⟶ Cash.COP, Stock.XYZ ⟶ Right.XYZ, Cash.COP ⟶ Cash.COP2, Bond.XYZ ⟶ Cash.COP, Option.XYZ ⟶ Stock.XYZ}

Get a list of vertices that are at a distance of at most *d*

In[ ]:= `AdjacencyList[g, "Cash.COP", 1]`

Out[ ]= {Stock.XYZ, Cash.COP2, Bond.XYZ}

In[ ]:= `AdjacencyList[g, "Cash.COP", Infinity]`

Out[ ]= {Right.XYZ, Stock.XYZ, Cash.COP2, Bond.XYZ, Option.XYZ}

In both **AdjacencyList** and **IncidenceList** the downstream directed vertices and upstream indicated vertices are included equally

Graph distance matrix

Directed distance of each vertex to every other. The rows yield what each vertex influences (0 -> itself, 1 -> direct connection, > 1 = indirect connection, ∞ => no connection). The columns yield which vertices import that vertex. A number of algorithms can calculate this (Dijkstra, Floyd-Warshall, some others - see below sections).

In[ ]:= `gdm = GraphDistanceMatrix[g];`

In[ ]:= `gdm // MatrixForm[#, TableHeadings → {VertexList[g], VertexList[g]}] &`

Out[ ]//MatrixForm=

	Right.XYZ	Stock.XYZ	Cash.COP	Cash.COP2	Bond.XYZ	Option.XYZ	Bond.A
Right.XYZ	0	1	2	3	∞	∞	∞
Stock.XYZ	1	0	1	2	∞	∞	∞
Cash.COP	∞	∞	0	1	∞	∞	∞
Cash.COP2	∞	∞	∞	0	∞	∞	∞
Bond.XYZ	∞	∞	1	2	0	∞	∞
Option.XYZ	2	1	2	3	∞	0	∞
Bond.ABC	∞	∞	∞	∞	∞	∞	0
Cash.EUR	∞	∞	∞	∞	∞	∞	∞
Future.ABC	∞	∞	∞	∞	∞	∞	1
Thing.ABC	∞	∞	∞	∞	∞	∞	2

The distances of the reverse graph is the transpose of the distance matrix

```
In[ ]:= GraphDistanceMatrix[g // ReverseGraph] //
         MatrixForm[#, TableHeadings → {VertexList[g], VertexList[g]}] &
```

Out[ ]//MatrixForm=

	Right.XYZ	Stock.XYZ	Cash.COP	Cash.COP2	Bond.XYZ	Option.XYZ	Bond.A
Right.XYZ	0	1	∞	∞	∞	2	∞
Stock.XYZ	1	0	∞	∞	∞	1	∞
Cash.COP	2	1	0	∞	1	2	∞
Cash.COP2	3	2	1	0	2	3	∞
Bond.XYZ	∞	∞	∞	∞	0	∞	∞
Option.XYZ	∞	∞	∞	∞	∞	0	∞
Bond.ABC	∞	∞	∞	∞	∞	∞	0
Cash.EUR	∞	∞	∞	∞	∞	∞	1
Future.ABC	∞	∞	∞	∞	∞	∞	∞
Thing.ABC	∞	∞	∞	∞	∞	∞	∞

Cash.COP depends on:

```
In[ ]:= gdm[[VertexIndex[g, "Cash.COP"]]]
Out[ ]= {∞, ∞, 0, 1, ∞, ∞, ∞, ∞, ∞, ∞}
```

Assets which depend on Cash.COP:

```
In[ ]:= gdm[[ ;; , VertexIndex[g, "Cash.COP"]]]
Out[ ]= {2, 1, 0, ∞, 1, 2, ∞, ∞, ∞, ∞}
```

Items influencing Cash.COP

```
In[ ]:= Extract[VertexList[g], Position[
          gdm[[ ;; , VertexIndex[g, "Cash.COP"]]], Except[Infinity], {1}, Heads → False]]
Out[ ]= {Right.XYZ, Stock.XYZ, Cash.COP, Bond.XYZ, Option.XYZ}
```

Items influenced by Cash.COP

```
In[ ]:= Extract[VertexList[g], Position[
          gdm[[VertexIndex[g, "Cash.COP"]]], Except[Infinity], {1}, Heads → False]]
Out[ ]= {Cash.COP, Cash.COP2}
```

```
In[ ]:= Extract[VertexList[g], Position[
          gdm[[VertexIndex[g, "Bond.ABC"]]], Except[Infinity], {1}, Heads → False]]
Out[ ]= {Bond.ABC, Cash.EUR}
```

Specific calculation of graph distances:

```
In[ ]:= GraphDistance[g, "Cash.COP"]
Out[ ]= {∞, ∞, 0, 1, ∞, ∞, ∞, ∞, ∞, ∞}
```

```
In[ ]:= GraphDistance[ReverseGraph@g, "Cash.COP"]
Out[ ]= {2, 1, 0, ∞, 1, 2, ∞, ∞, ∞, ∞}
```

Examples

General documentation and pseudocode for algorithm

https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Super interesting use case...

https://www.boost.org/doc/libs/1_70_0/libs/graph/doc/file_dependency_example.html

Reachability matrix

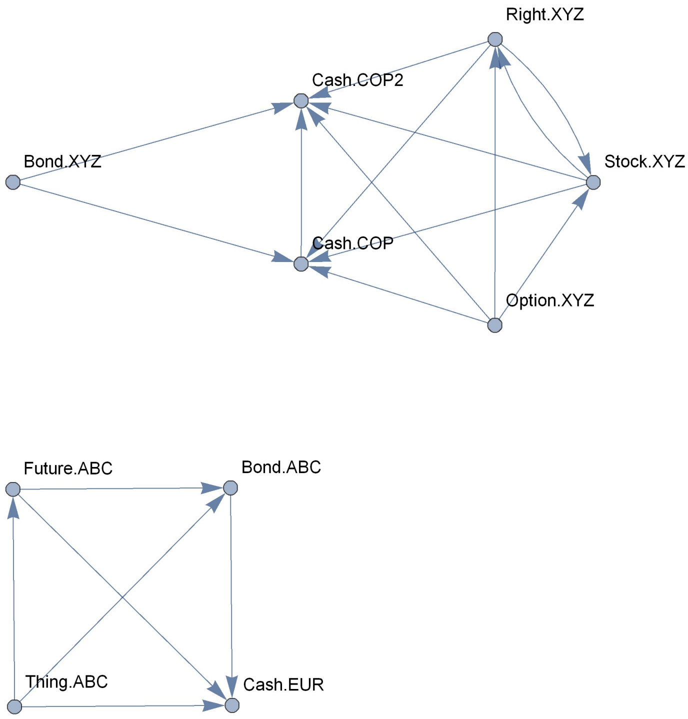
Infer direct connections between each and every vertex via intermediate vertices. Need to compute the transitive closure of the binary relation (being the adjacency matrix).

Using Mathematica

TransitiveClosureGraph calculates the graph of every vertex connected to every other using known intermediate vertices:

In[ ]:=
```
TransitiveClosureGraph[g, VertexLabels → Automatic]
```

Out[ ]=



The adjacency matrix of this graph is the **ReachablityMatrix**

In[ ]:=
```
AdjacencyMatrix@TransitiveClosureGraph[g] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Manually

Note a little fiddling is required for the diagonal elements

In[ ]:=
```
x = IdentityMatrix[10] +
    Total@Table[MatrixPower[AdjacencyMatrix[g], i] // Normal, {i, 1, 10}];
MatrixForm@
 x
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 6 & 5 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 6 & 5 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 5 & 5 & 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 1 \end{pmatrix}$$

In[ ]:=
```
(x /. n_ /; n > 0 → 1) - IdentityMatrix[10] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Warshall's algorithm

Warshall's algorithm is a modified version of the Floyd-Warshall algorithm. The path length function is modified with Boolean operations, such that min(...) → ... || ... and ... + ... → ... && .... By storing booleans, the storage is smaller and the performance is slightly faster by a constant factor; the complexity remains $O(N^3)$.

Require the adjacency matrix in Boolean form:

In[ ]:=
```
adjBool = Map[(# ≠ 0) &, Normal[AdjacencyMatrix[g]], {2}];
MatrixForm@adjBool
```

Out[ ]//MatrixForm=

False	True	False	False	False	False	False	False	False	False
True	False	True	False	False	False	False	False	False	False
False	False	False	True	False	False	False	False	False	False
False	False	False	False	False	False	False	False	False	False
False	False	True	False	False	False	False	False	False	False
False	True	False	False	False	False	False	False	False	False
False	False	False	False	False	False	False	True	False	False
False	False	False	False	False	False	False	False	False	False
False	False	False	False	False	False	True	False	False	False
False	False	False	False	False	False	False	True	True	False

```
In[]:= MatrixQ[adjBool, BooleanQ]
Out[]= True
```

Use a recursive definition of the Warshall algorithm. In Mathematica the use of "memoization" gives an important improvement to performance but this requires *k* not be "too big" otherwise we'll blow out the memory.

```
In[]:= Clear@pathD;
pathD[adjB_ /; MatrixQ[adjBool, BooleanQ]][i_, j_, 0] :=
  pathD[adjB][i, j, 0] = adjB[[i, j]];
pathD[adjB_][i_, j_, k_] := pathD[adjB][i, j, k] =
   pathD[adjB][i, j, k - 1] ||
    (pathD[adjB][i, k, k - 1] && pathD[adjB][k, j, k - 1]);
```

```
In[]:= warshallSoln =
  Boole /@ MapIndexed[pathD[adjBool][First@#2, Last@#2, 10] &, adjBool, {2}];
```

```
In[]:= warshallSoln // MatrixForm
Out[]//MatrixForm=
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

```
In[]:= warshallSoln
Out[]= {{1, 1, 1, 1, 0, 0, 0, 0, 0, 0}, {1, 1, 1, 1, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 1, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 1, 1, 0, 0, 0, 0, 0, 0}, {1, 1, 1, 1, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 1, 1, 0, 0}, {0, 0, 0, 0, 0, 0, 1, 1, 1, 0}}
```

Suspect that the variation in the diagonal elements indicates self-cycles?

```
In[]:= (warshallSoln - AdjacencyMatrix@TransitiveClosureGraph[g]) // MatrixForm
Out[]//MatrixForm=
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Using a potentially more efficient **Fold** construct - this is closer to a procedural programming approach

In[]:=
```
fSoln = Boole /@
   Fold[Table[#1⟦i, j⟧ || (#1⟦i, #2⟧ && #1⟦#2, j⟧), {j, 1, 10}, {i, 1, 10}] &,
    adjBool, Range[10]];
```

In[]:=
```
MatrixForm@fSoln
```

Out[]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Interestingly the recursive technique is faster than folding. For very large graphs this is unlikely to hold.

In[]:=
```
RepeatedTiming[
 Boole /@ Fold[Table[#1[[i, j]] || (#1[[i, #2]] && #1[[#2, j]]), {j, 1, 10},
      {i, 1, 10}] &, adjBool, Range[10]];]
```

Out[]=
```
{0.0011, Null}
```

In[]:=
```
RepeatedTiming[
 Boole /@ MapIndexed[pathD[adjBool][First@#2, Last@#2, 10] &, adjBool, {2}];]
```

Out[]=
```
{0.00017, Null}
```