Manual of NOCAD - **N**etwork based **O**bservability and **C**ontrolability **A**nalysis of **D**ynamical Systems toolbox

Dániel Leitold, Ágnes Fogarassyné Vathy, János Abonyi

February 12, 2017

Contents

1 The NOCAD toolbox **3**
- 1.1 Introduction . . . 3
- 1.2 Representations of dynamical systems . . . 3
- 1.3 Controllability, observability . . . 4
- 1.4 Maximum matching . . . 4
- 1.5 Types of connections . . . 7
  - 1.5.1 Influence . . . 7
  - 1.5.2 Self-influencing influence . . . 7
  - 1.5.3 Interaction . . . 8
  - 1.5.4 Self-influencing interaction . . . 8
  - 1.5.5 Overview . . . 8
- 1.6 Handling strongly connected components . . . 8
  - 1.6.1 Signal sharing method . . . 9
  - 1.6.2 Path finding method . . . 10

2 Network mapping **11**
- maximumMatching . . . 12
- maximumMatchingPF . . . 13
- maximumMatchingSS . . . 14
- generateMatricesPF . . . 15
- generateMatricesSS . . . 16

3 System characterization **17**
- numNodes . . . 24
- numEdges . . . 25
- density . . . 26
- diameter . . . 27
- degreeVariance . . . 28
- degreeFreeman . . . 29
- isControllable . . . 30
- isObservable . . . 31
- degreeRel . . . 32
- pearsonDir . . . 33
- sumSq . . . 34
- percentLoopSym . . . 35
- getNodes . . . 36
- addEdges . . . 37
- heatmaps . . . 38
- degreeIn . . . 39
- degreeOut . . . 40
- degree . . . 41
- degreeScott . . . 42

closenessCentrality . . . 43
nodeBetweenness . . . 44
pageRank . . . 45
degreeCorrel . . . 46
controlCentrality . . . 47
observeCentrality . . . 48
driverNodes . . . 49
driverType . . . 50
sensorNodes . . . 51
sensorType . . . 52
clusterControl . . . 53
clusterObserve . . . 54
driverSimilarity . . . 55
sensorSimilarity . . . 56
jaccard . . . 57
simWeighting . . . 58
reachC . . . 59
reachO . . . 60
reachability . . . 61
edgeBetweenness . . . 62
endpointSim . . . 63
similarity . . . 64
getConfig . . . 65
matricesToStruct . . . 66

4 System investigation **68**
createDataAndTables . . . 69
genCol . . . 70

Chapter 1

The NOCAD toolbox

1.1 Introduction

The implemented NOCAD toolbox (**N**etwork based **O**bservability and **C**ontrolability **A**nalysis of **D**ynamical Systems) deals with the workflow shown in Figure 1.1.

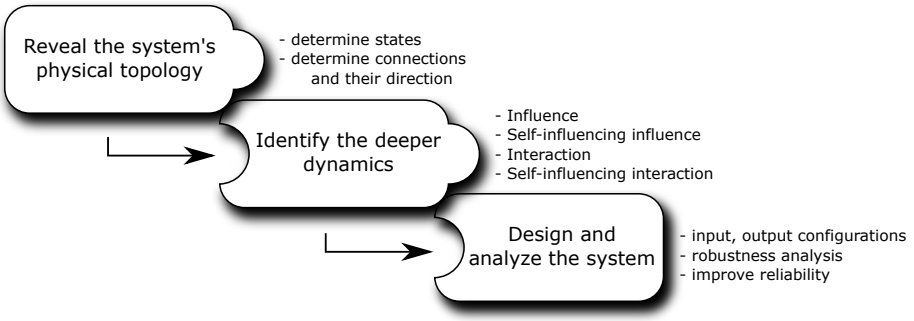


Figure 1.1: Flow chart of the system design and analysis.

Accordingly to the implemented functions, the toolbox is divided into modules that are: network mapping, system characterization, and system investigation. The task of the first, network mapping module is to create a dynamical system from a network, i.e. the necessary matrices of state-space model are generated for a topology, such that the created system is controllable and observable. The second part, system characterization, is the main module, which task is to analyze the system. This module contains numerous measures that are calculated not just to systems, but networks as well, so this module of the toolbox can be useful separately for network analysis as well. It is unequivocal, that the systems' behavior is strongly connected to the network structure and the charging of the system could be analyzed with network based measures. The last, system investigation module deals with the example networks that found in literature and creates the modified networks according to the connection types. It runs the algorithms of module 1 and 2 on the networks and creates comparison tables from the results.

1.2 Representations of dynamical systems

State-space models are widely used to represent time-invariant linear systems mathematically. The state-space model contains two equation, the state equation shown in Equation (1.1) and the output equation presented in Equation (1.2).

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{1.1}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \tag{1.2}$$

In Equation (1.1), $\mathbf{x}$ is a column vector that stands for internal state variables of the dynamical system. Column vector $\mathbf{u}$ represents the inputs, i.e. the external actuator of the system. Matrices $\mathbf{A}$ and $\mathbf{B}$ show how the state variables and inputs influence the state variables, respectively. In Equation (1.2), $\mathbf{y}$ is the vector for the outputs. Matrices $\mathbf{C}$ and $\mathbf{D}$ show how the state variables and inputs influence the output, respectively. We can determine the dimensions of the matrices also. If the number of internal state variables noted by N, the number of inputs noted by M and the number of the outputs noted by K, then $\mathbf{A} \in \Re^{N \times N}$, $\mathbf{B} \in \Re^{N \times M}$, $\mathbf{C} \in \Re^{K \times N}$, $\mathbf{D} \in \Re^{K \times M}$.

1.3 Controllability, observability

As we presented above, the state-space model contains four matrices that can describe a dynamical system. After mapping the topology between the state variables, we have to determine where we have to place inputs and outputs, and it is important, to place them such a way, that the created system will be controllable and observable.

We say that, a system is controllable, if we can drive it from any initial state to any desired final state within finite time with properly selected inputs [10, 15, 23, 6], so we can reach any $\mathbf{x}$ state with a finite number of derivation. To determine the controllability, we have to create the controllability matrix shown in Equation (1.3).

$$\mathcal{C} = [\mathbf{B}, \mathbf{AB}, \ldots, \mathbf{A}^{N-1}\mathbf{B}] \tag{1.3}$$

In Equation (1.3) $\mathbf{A}$ is the state-transition matrix, and $\mathbf{B}$ describes how the inputs influence the state variables, according to Equation (1.1). The system is controllable, if the Kalman's criterion satisfied [10], i.e. the controllability matrix has maximum structural rank, as can be seen in Equation (1.4).

$$rank(\mathcal{C}) = N \tag{1.4}$$

The calculation of the structural rank is easy: if there is a matching or assignment of the columns and rows by non-zero entry, then the matrix has maximal structural rank. Otherwise the rank is equal with the size of maximum matching.

The observability is mathematical dual of controllability. A system is observable if we can determine the system's state at any given time by a finite measurement record of input and output variables [6]. To decide if a system is observable we have to create the observability matrix as it shown in Equation (1.5).

$$\mathcal{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{N-1} \end{bmatrix} \tag{1.5}$$

In Equation (1.5) $\mathbf{A}$ is the state-transition matrix, and $\mathbf{C}$ describes how the outputs observe the state variables, according to Equation (1.1) and Equation (1.2). The system is observable, if the Kalmans criterion satisfied [10], i.e. the observability matrix has maximum structural rank (Equation (1.6)).

$$rank(\mathcal{O}) = N \tag{1.6}$$

1.4 Maximum matching

It is obvious, if we want to create a controllable and observable system from a network, then we have to create suitable $\mathbf{B}$ and $\mathbf{C}$ matrices based on $\mathbf{A}$, that represents the state-transition matrix. In addition, another goal is to make the system controllable and observable with a minimum number of inputs and outputs, and this is what makes this task so challenging. The trivial solution, where all state variables are influenced and observed separately, also determines a controllable and observable system, but it is not cost-efficient because each input and output have a cost in

real life. With brute force technique, we can generate all possible input configurations, but we have to generate $2^N - 1$ configurations, where N is the number of state variables in the system. Each configuration requires the generation of the controllability matrix, which is very resource demanding, to check controllability. Furthermore, for observability, these calculations should be done again. To solve this challenging task, Liu et al. proposed maximum matching algorithm [12] to determine the nodes that get an input, and the nodes, where we have to place an output, to create controllable and observable system. In the following, we will call the state variables with input as *driver* nodes, while the state variable with output will be the *sensor* nodes, as they called in literature as well [12, 14].

Usage of maximum matching is a structural analysis only, it ignores the weights of the edges and care with the structural architecture exclusively. The designed driver and sensor nodes also just structural positions in the system, the methodology does not assign any parameter for matrices **B** and **C** or for vectors **u** and **y**. The network, which is examined, is a topology that described connections between the state variables, or system's components, and represented as an adjacency matrix. An example representation with associated matrices can be seen in Figure 1.2.

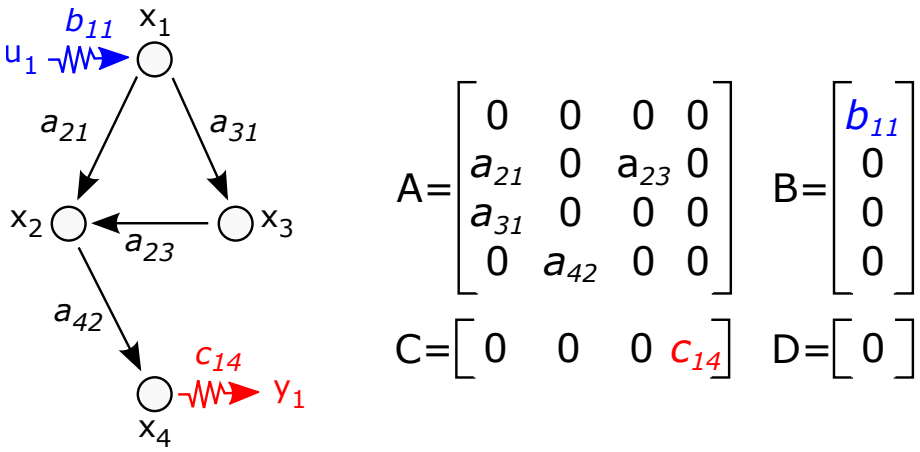


Figure 1.2: The methodology used representation with associated matrices.

The maximum matching algorithm is a combinatorial method and it defines the largest disjoint edge set in a graph. It can be used for undirected and directed graphs as well. In undirected case, two edges are disjoint, if they have no common endpoint. In the following, we introduce the results of the maximum matching through an example.

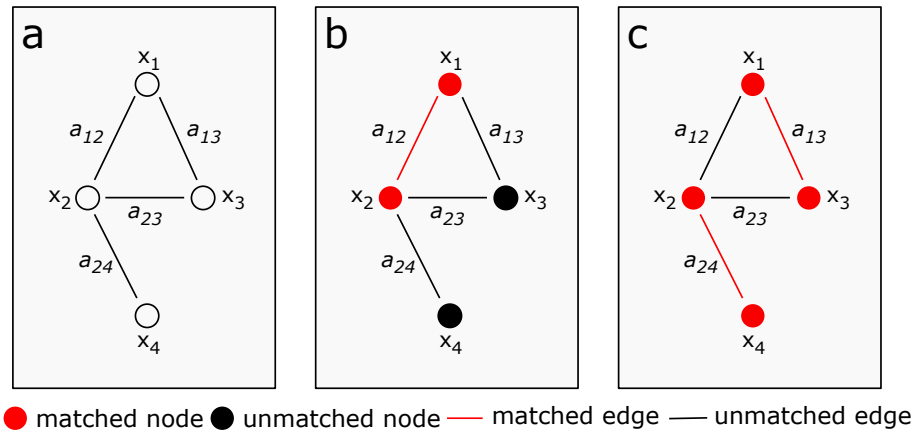


Figure 1.3: (a) An undirected graph. (b) Maximal matching. (c) Maximum matching.

In Figure 1.3(a), we can see a simple, undirected graph, with four nodes and four edges. In Figure 1.3(b) we select edge a_{12} to the disjoint edge set, and nodes x_1 and x_2 will be the matched nodes. The other nodes are the unmatched nodes. In this case, we cannot increase the disjoint edge set, because edges a_{13}, a_{23} and a_{24} have common endpoint with a_{12}. The name of this result is maximal matching. The maximal matching is a disjoint edge set, which cannot be expanded further, i.e. each edge in the graph has at least one intersection where there is an edge from the matching. In Figure 1.3(c) the maximum matching of the graph can be seen. In this case, we select two edges in the matching. As there is no other matching, which has a higher cardinality than 2, we call this maximum matching. So a maximal matching is maximum matching if there is no other maximal matching with higher cardinality. In Figure 1.3(c), we matched all the nodes, so we call it perfect matching, also. The perfect matching is a matching, where all nodes are matched.

In directed case, the disjoint condition is that the two edges can not have common start- or endpoint, but it is possible, that the endpoint of an edge and the start point of another one is the same node. In this case the matched nodes are the endpoints of the matched edges, other nodes are unmatched. A simple example can be seen in Figure 1.4.

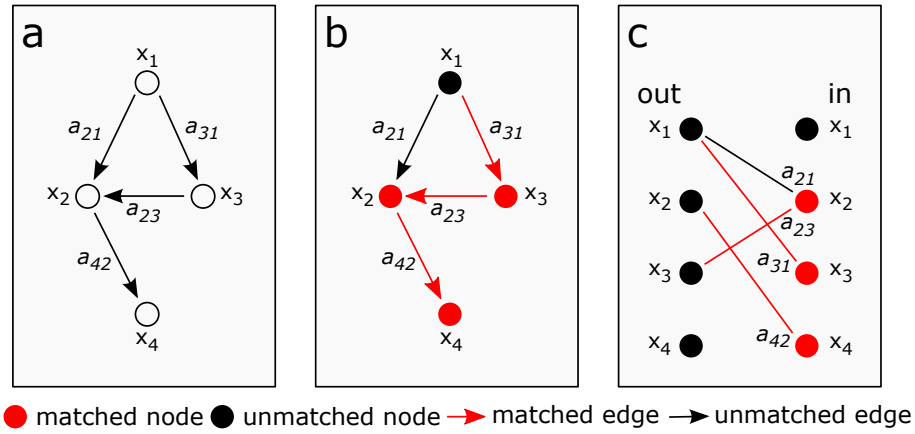


Figure 1.4: (a) A directed graph. (b) Maximum matching. (c) Undirected, bipartite graph based representation.

In Figure 1.4(a) we can see the original network, but, in this case, edges are directed. Figure 1.4(b) shows the maximum matching of the graph with color red. The bipartite representation of the network is shown by Figure 1.4(c). In bipartite representation, we have to create separated node-pair for each node, one for the outgoing edges and one for the ingoing edges. After that, we can add the edges, and apply the undirected version of maximum matching. The matched nodes of the directed graph are the matched nodes of set "in" in the bipartite representation. We can see, the results are the same in Figure 1.4(b) and Figure 1.4(c).

In our work, we used the maximum matching base of Dulmage-Mendelsohn decomposition [3] by Pothen et al. [18]. To generate the block triangular form of the sparse, unsymmetric matrix, the algorithm firstly calculates the maximum matching. Pothen and Fan changed [18] Duff's maximum matching algorithm [2] with computational time $\mathcal{O}(EV)$, where E is the number of the edges and N is the number of nodes in the graph. Although there is an algorithm with better computation time, namely the Hopcroft-Karp algorithm [7] with $\mathcal{O}(E\sqrt{V})$ and this version was used by Liu et al. in [12], but its performance is case dependent and Pothen's version was faster on denser problem[18].

According to [12], if we determine the unmatched nodes of the network determined by $\mathbf{A}$, then we get the sensor nodes, while if we generate the unmatched nodes of the transpose of the network of $\mathbf{A}$, then we get the driver nodes. With these sensor and driver nodes the system will be controllable and observable. In matrix $\mathbf{B}$ each input has a separate column. For driver node i the associated input's column has all zero value, except element i, which is non-zero. Analogously,

in matrix $\mathbf{C}$ each output has a separate row, and for sensor node i in its output's row has only one non-zero at position i.

1.5 Types of connections

The primary aim of our work is to analyze the connection types, that can be well defined in dynamical systems. With interpreting the principle of conservation of mass, energy or momentum in the systems, connection types can be defined as loops and symmetries in the network representations. Another strengthening fact is that dynamical systems also can be described with differential-algebraic equations. The differential parts describe how the state variable changing during the time by its own effects, this way generates loops in the network. The algebraic part mainly describes the effects between two state variables. Again, if we examine more precisely the system, we can see that there are more connections than we presumed. According to this finding, we introduce four connection types that are observable in dynamical systems, then we investigate how these types influence the number of necessary driver and sensor nodes.

1.5.1 Influence

The first type of the connections is the influence. In this case the connection is a simple edge between two state variables and it is equal with the physical topology. An example network and the result of maximum matching presented in Figure 1.5(a).

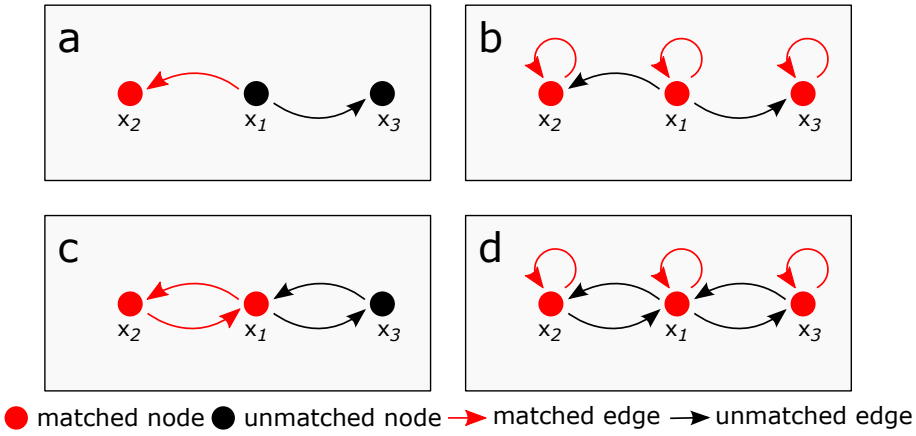


Figure 1.5: The connection types: (a) influence, (b) self-influencing influence, (c) interaction and (d) self-influencing interaction. It can be seen, the result of maximum matching changed by the connection types.

Probably, most of the networks can be found in the literature is based on this type, i.e. the creators of the network think that the states are independent of each other. The network shows the will of the creators of the system, but as we showed before, there can be other effects between the two states because they are usually not independent from each other. In our work, the influence is examined by unchanged adjacency matrix, i.e. we used the original form of datasets as it used in the literature.

1.5.2 Self-influencing influence

The second connection type is the self-influencing influence, where the change of a state variable not only affects other ones, but it changes itself as well. In the network representation the self-influencing influence appears as a loop as it can be seen in Figure 1.5(b), and formally in Equation

(1.7) for the network and in Equation (1.8) for the state-transition matrix.

$$\forall a \in V,\ (a,a) \in E \tag{1.7}$$

$$\mathbf{A}(i,i) = \{1|\ i \in \{1,2,\ldots,N\}\} \tag{1.8}$$

According to the principle of conservation of mass, momentum and energy this connection type is unequivocal and maybe it seems understandable and simple, but these loops can change the result of maximum matching radically, as it can be seen in Figure 1.5(b). In this case the matching does not provide any unmatched nodes, so we have to intervene afterward. In Section 1.6, we present two methods, which can assign driver and sensor nodes for matchings like this.

1.5.3 Interaction

The third connection type is the interaction, where the state variable does not affect itself, but the connections are symmetric between the state variables. In the network presentation, the interaction appears as a symmetric edge-pair, as it can be seen in Figure 1.5(c), and formally in Equation (1.9) for the network and in Equation (1.10) for the state-transition matrix representation.

$$\forall (a,b) \in E \rightarrow (b,a) \in E \tag{1.9}$$

$$\mathbf{A}(i,j) = \{1|\ if\ \mathbf{A}(j,i) = 1\} \tag{1.10}$$

This connection type shows that the systems have dynamics, the effects between two state variables can change their direction. The results generated by maximum matching changed again, as it shown by Figure 1.5(c), but this time, we have an unmatched node. Now, we can see that while for influence two unmatched nodes were generated, for interaction only one was enough. Here we can see that the connection types changed the results.

1.5.4 Self-influencing interaction

The last connection type is the self-influencing interaction, the union of types self-influencing influence and interaction. In the network presentation as it can be seen in Figure 1.5(d), formally in Equation (1.11) for the network and in Equation (1.12) for the state-transition matrix.

$$\forall a \in V,\ (a,a) \in E,\ \wedge \forall (a,b) \in E \rightarrow (b,a) \in E \tag{1.11}$$

$$\mathbf{A}(i,j) = \{1|\ if\ \mathbf{A}(j,i) = 1\ \vee i = j\}\} \tag{1.12}$$

1.5.5 Overview

In this section, the established four connection types will be summarized. To collect the key features of these types, Table 1.1 give a short descriptions about them. In the toolbox, to simulate the different types in the networks, we performed the necessary changes in them as we mentioned above or as it presented by Table 1.1.

As we mentioned, the maximum matching can provide solution without unmatched nodes, and it is more significant in the case of the connection types, as they create strongly connected components in the topology. As we referred above, in the next section we will present two methods to solve this problem.

1.6 Handling strongly connected components

As we introduced at maximum matching, a new problem was raised. The algorithm can provide results, where there are no unmatched nodes, i.e. no driver and/or sensor nodes are determined, thus the created systems are not controllable and/or observable. The task in this part is to produce the matrices of state-space model (Equation (1.1) and (1.2)) from an adjacency matrix

Table 1.1: Connection types and their effects in network.

Connection type	Description	In network	Formalization
Influence	The state variables are independent, just the physical connections are important.	No changing.	Nothing.
Self-influencing influence	The state variables change themselves.	Add loops.	$\forall a \in V,\ (a,a) \in E$
Interaction	Connections between states have no direction.	Change network to symmetric.	$\forall (a,b) \in E \rightarrow (b,a) \in E$
Self-influencing interaction	Each state variables change themselves, and connections between them have no direction.	Add loops and change network to symmetric.	$\forall a \in V,\ (a,a) \in E,\ \wedge$ $\forall (a,b) \in E \rightarrow (b,a) \in E$

that represents the connection between the state variables. The transpose of the adjacency matrix is the state-transition matrix ($\mathbf{A}$), so we need to define the suitable matrices $\mathbf{B}$ and $\mathbf{C}$ to create a well-defined, controllable and observable system. The matrix $\mathbf{D}$ is zero, as it is usual in real systems as well. In the following, we expound two methods that can perform such work. The first one was published by Liu et al. in 2013 [14], while the second approach is proposed by us, to solve the problem. In the following we mainly dealt with the controllability, but since observability is mathematical dual of controllability, every statement is true to the observability as well.

1.6.1 Signal sharing method

Liu et al. claim that after we applied the maximum matching algorithm, we have to check strongly connected components (SCCs) [14]. An SCC is controlled if there is an edge pointing to the SCC from an input, or any state variable, which is not part of the SCC. If there is no such an edge, then we have to place one from an existing input to any of the nodes in the SCC. Thus, every unmatched node has its own input and every uncontrolled SCC is controlled by a shared signal of an existing input. In Figure 1.6(a), a simple example can be seen.

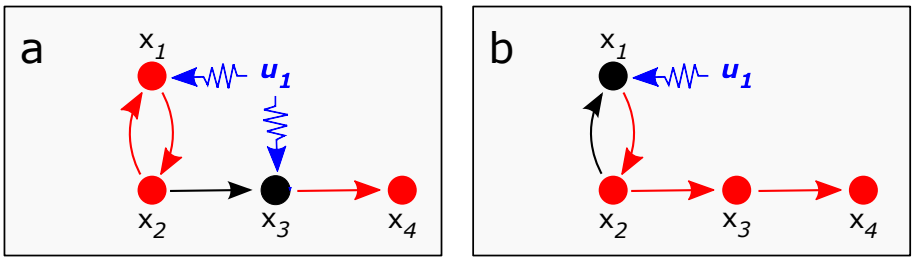


Figure 1.6: Solutions provided by signal sharing (a) and path finding (b) to handle strongly connected components.

In Figure 1.6(a) the unmatched node, i.e. the driver node is x_3. We place an input, u_1, in the system, and with a signal from u_1, we control x_3. After that, we find unmatched SCCs and share input signal on one of the nodes in the SCC, in our case on node x_1 or x_2. In Figure 1.6(a), we chose node x_1.

The design of matrix $\mathbf{B}$ is the following: we have one input and four states, so we create a zero matrix with size $\mathbf{B} \in \Re^{4\times 1}$. Now, we iterating over all inputs. In our case, the first input is u_1 and it influences the nodes x_1 and x_3, so in the first column of $\mathbf{B}$ the first and the third element is changed to 1. If we generate this $\mathbf{B}$ matrix and check the Kalman's rank criteria [10] as it was introduced in Equation (1.4), then we get that, the system is controllable.

1.6.2 Path finding method

In Figure 1.6(a) we can see that, if we change the maximum matching such that, we take out the edge (x_2, x_1) and take in (x_2, x_3), then we get a maximum matching also, and in this case, we can control the whole network with only one input, without sharing input signal as it can be seen in Figure 1.6(b). To reach this result, we invented a new algorithm, which can find this small changes, and create a system, where there is no signal sharing. It is an important virtue because in real life the sharing of an input signal is not always possible.

During the algorithm design, we found that, if an unmatched SCC contains a Hamiltonian path, which can continue in an unmatched node, then with the changing of the result of maximum matching a better solution can be given, more specifically, another maximum matching can be generated that more appropriate for this methodology. The root of the problem is that, the maximum matching is not unequivocal, and the algorithm cannot see the difference between the solutions, the only aim of the algorithm is reaching maximum matching. This leads to the following three findings. Firstly, the phenomenon that causes the lack of unmatched nodes is not the uncontrolled SCCs but the uncontrolled SCCs with Hamiltonian cycles. Secondly, as we have to handle cycles, not SCCs, it is not true that we can control an arbitrary SCC with only one driver node. If an SCC cannot be controlled with only one input, then the maximum matching determines at least one unmatched node in the SCC. Thirdly, if we have to control a Hamiltonian cycle in an SCC such that we can also control other nodes in the network, i.e. the path of the controlling continues in an unmatched node, then it is significant which node will be the driver node. For example, in Figure 1.6(a) we can choose between x_1 and x_2 according the signal sharing, but in Figure 1.6(b) we have to choose x_1. Due to these findings, it could be established that the problem appeared with the usage of maximal matching can be addressed to Hamiltonian cycles and Hamiltonian paths, not to strongly connected components.

Chapter 2

Network mapping

The task of the first module is to create a system from an adjacency matrix. As we presented in Section 1.6, this could be done by two methods, path finding and signal sharing. In this module both modified maximum matching algorithm are implemented, and another functions are designed to create the matrices based on the results of maximum matchings. With using the path finding method, the system created from the example network can be seen in Figure 2.1.

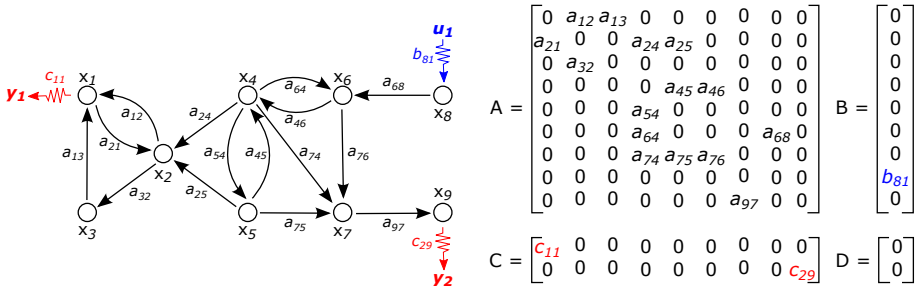


Figure 2.1: Mapped network with determined input (blue) and output (red) configurations.

maximumMatching

Purpose

Generates matched and unmatched nodes, and a vector, which contains the pairing, defined by the maximum matching algorithm.

Synopsis

`[`**matched**, **unmatched**, **matchedBy**`]=maximumMatching(`**adj**`)`

Description

The **maximumMatching** function generates the matched and unmatched nodes of the input network, based on the maximum matching algorithm which generates the maximum size of the disjoint edge set in the graph defined by **adj**. Furthermore, the third vector contains the pairing, where element i of the vector is the node, which is matching the node i. The input network could be adjacent or sparse matrix.

Algorithm

The **maximumMatching** function uses the Dulmage-Mendelson decomposition [3, 18] to calculate maximum matching.

See Also

`dmperm, find`

maximumMatchingPF

Purpose

Generates matched and unmatched nodes, and a vector, which contains the pairing, defined by the maximum matching algorithm. In addition, this function resolves the cycles generated problem, by path finding.

Synopsis

`[`**matched**, **unmatched**, **matchedBy**, **removable**`]=maximumMatchingPF(`**adj**`)`

Description

The **maximumMatchingPF** function generates the matched and unmatched nodes of the input **adj**, based on the maximum matching algorithm which generates the maximum size of the disjoint edge set in the graph defined by the network. Furthermore, the third vector contains the pairing, where element i of the vector is the node, which is matching the node i. This function also changes some pairing of original maximum matching for usage in system theory. The fourth output, **removable**, contains edges, which should be removed from the original network, **adj**, to provide a suitable matching for system's controllability, observability. The input network could be an adjacency matrix or a sparse matrix.

Algorithm

The **maximumMatchingPF** function uses the **maximumMatching**, so the Dulmage-Mendelson decomposition [18, 3] to calculate maximum matching and Tarjan's algorithm [24] for finding strongly connected components. Since the maximum matching algorithm could be used in system theory and the SCCs can cause problems, this function resolve it by finding a path in the SCC such that, if an unmatched node can be matched by the SCC, then they will be connected to each other, so a path will be created through the SCC which continues in the unmatched node. The algorithm uses the matlabBGL toolbox's **components** function.

See Also

`maximumMatching, issparse, components, sparse, hist, max, find, diag, union, unique, setdiff, sum, intersect, isempty`

maximumMatchingSS

Purpose

Generates matched and unmatched nodes, the vector unmatchedSCC which contains one node from each unmatched SCC and a vector, which contain the pairing, defined by the maximum matching algorithm.

Synopsis

`[`**matched**, **unmatched**, **unmatchedSCC**, **matchedBy**`]=...`

`maximumMatchingSS(`**adj**`)`

Description

The **maximumMatchingSS** function generates the matched and unmatched nodes of the input **adj**, based on the maximum matching algorithm which generates the maximum size of disjoint edge set. The third vector contains a node from each unmatched SSC, which need a shared signal from an input or output while the fourth vector contains the pairing, where element i of the vector is the node, which is matching the node i. The input network could be an adjacency matrix or sparse matrix.

Algorithm

The **maximumMatchingSS** function uses the **maximumMatching**, so the Dulmage-Mendelson decomposition [18, 3] to calculate maximum matching and Tarjan's algorithm [24] for finding strongly connected components. The maximum matching algorithm could be used in system theory, but the SSCs can cause problems. This function do not resolve this problem, but it gives some help: in the vector **unmatchedSCC** there is a node, from each unmatched SCC. We have to share signal on these nodes to fully controllability and observability. The algorithm uses the matlabBGL toolbox's **components** function.

See Also

`maximumMatching, issparse, components, sparse, hist, max, find, diag, union, unique, setdiff, sum, intersect, isempty`

generateMatricesPF

Purpose

Create the matrices of state-space model based on **a***dj*, which is the network of the system, such that the whole system will be controllable and observable along with minimum number of inputs and outputs.

Synopsis

`[`**Amatrix**, **Bmatrix**, **Cmatrix**, **Dmatrix**`]=generateMatricesPF(`**adj**`)`

Description

The **generateMatricesPF** function produces matrices **A**, **B**, **C** and **D** from the adjacency matrix of a network created by a real system. With these matrices, the system will be controllable and observable. The input network could be an adjacency matrix or sparse matrix.

Algorithm

The **generateMatricesPF** function uses the **maximumMatchingPF** to calculate the matched and unmatched nodes. The **unmatched** vector gives where we have to place the driver and sensor nodes, so we can create the **B** and **C** matrices. In this case, a signal always affects one node, so the columns of the **B** matrix and the rows of the **C** matrix contain only one nonzero element. Matrix **A** is the transpose of **adj**, matrix **D** is a null matrix with the same number of rows as the number of the sensor nodes and the same number of columns as the number of driver nodes.

See Also

`maximumMatchingPF, full, sparse, length, issparse`

generateMatricesSS

Purpose

Create the matrices of state-space model based on **a***dj*, which is the network of the system, such that the whole system will be controllable and observable along with minimum number of inputs and outputs.

Synopsis

`[`**Amatrix**, **Bmatrix**, **Cmatrix**, **Dmatrix**`]=generateMatricesSS(`**adj**`)`

Description

The **generateMatricesSS** function produces matrices **A**, **B**, **C** and **D** from the adjacency matrix of a network created by a real system. With these matrices, the system will be controllable and observable. The input network could be an adjacency matrix or sparse matrix.

Algorithm

The **generateMatricesSS** function uses the **maximumMatchingSS** to calculate the matched, unmatched and that nodes which are a member of an uncontrolled SCC. The elements of **unmatched** and **unmatchedSCC** give, where we have to place and share the input and output signals, so we can create the **B** and **C** matrices. The elements of **unmatched** is demand separated signal while the nodes in **unmatchedSCC** are satisfied if they get a shared existing signal. In this case, a signal can affect more than one node, so the columns of the **B** matrix and the rows of the **C** matrix can contain more than one nonzero element. Matrix **A** is the transpose of **adj**, matrix **D** is a null matrix with the same number of rows as the number of the sensor nodes and the same number of columns as the number of driver nodes.

See Also

`maximumMatchingSS, full, sparse, length, issparse`

Chapter 3

System characterization

The main goal of the second module is to create a standalone tool, to analyze networks and systems. The implemented functions calculate centrality and clustering measures to characterize the network or the system. Most of the functions require only an adjacency matrix while some of them claim other inputs, such as the **A** matrix and other matrices of the state-space model. There is a function in the module, which task is to generate a structure from the matrices of the state-space model, which contains all the measures calculated by this module. The implemented measures are the following:

- Network/system centralities, properties:
  - **number of nodes**: the number of nodes in a network, or the number of states in a system.
  - **number of edges**: the number of edges in a network, or the number of connections in a system.
  - **density**: gives the proportion of the edges' cardinality to the theoretical maximum.
  - **diameter**: gives the longest geodesic path in the network. This information gives an upper bound for the derivations that are necessary to control or observe any of the states in a system.
  - **variance of degrees**: this measure can show if the degrees are similar, or they have a big difference in the network.
  - **Freeman's centrality with degrees [5]**: similar to the variance, but, while variance measures the distance from the average, this metric measure the distance from the maximum value of degrees.
  - **controllability**: this property can tell if the system is controllable.
  - **observability**: this property can tell if the system is observable.
  - **relative degree**: the simplified interpretation of system order: the maximum of all shortest path from each input to all output.
  - **Pearson correlation [19] of degree [17]**: this measure quantifies the correlation of the degrees. In directed networks, we can distinguish four types: in-in, in-out, out-out, out-in. The toolbox generates all the four types.
  - **heatmaps**: the toolbox can generate two heatmaps for a given topology. The first one belongs to the driver nodes, while the second one belongs to sensor nodes. The heatmap shows that how many driver and sensor nodes are necessary with increasing number of self-influencing and interaction.
- Node centrality measures:

- **degree** (in, out, simple): the module generates the in and the out degree of the nodes, and also their sum as simple degree.
- **Scott's approach of degree centrality [22]**: this measure is a normalized version of the original degree, as the degrees are divided by the theoretical maximum.
- **closeness**: shows how close a node or state is to the others in the network. The in and out closeness is interpretable and will be calculated as well.
- **node betweenness [4]**: the betweenness shows that how many shortest path is going through a given node.
- **PageRank**: this algorithm developed by Google's founders and named after one of them, Larry Page. This measure could show that which state is the most visited in the system if we suppose that random walks, or Markov-chains, are allowed.
- **correlation**: similar to Pearson coefficient, but this correlation gives the proportion of the number of neighbors of node i's neighbors to neighbors of node i. It is also interpretable for in-in, in-out, out-out and for out-in connections.
- **control centrality [13]**: determines how many states can be controlled by a control node.
- **observe centrality**: this measure is analog with control centrality, but, in this case, the measure deals with observability and sensor nodes, not with controllability and driver nodes.

- Node clustering measures:
  - **driver nodes**: the module collects the system's driver nodes in a vector. The toolbox also determines which driver nodes are source drivers, which ones are originated from external and internal dilation, and which ones are inaccessible driver nodes.
  - **sensor nodes**: the module collects the system's sensor nodes in a vector. As at the driver nodes, the toolbox identifies the source, external, internal and inaccessible sensor nodes.
  - **controlling**: the controlling process, i.e. it can determine which driver nodes influence a given state and how many derivations necessary to control it.
  - **observing**: the observing process, i.e. it can determine which sensor nodes observe a given state and how many derivations necessary to observe it.
  - **driver, sensor similarity**: this is a newly developed measure. In this similarity we take each driver node pair, and examine how similar the node sets, which are influenced by the driver nodes. This centrality uses the Jaccard similarity [8], which can give that, how similar two sets. It divides the section of the two sets with the union of them. If both sets are empty, then the Jaccard similarity is zero. The formula can be seen in Equation (3.1).

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

    The similarity in the first step generates the set of nodes, or states, which is influenced by the driver nodes. After this, the actual two driver nodes' sets are compared to each other according to Equation (3.1). After this the function weights the similarity according to that, how far the same state from the driver nodes, i.e. how many derivations needed to affect a state. The weight is computed by Equation (3.2).

$$weight(set_j, set_k) = 1 - \frac{\sum_{i \in set_j \cap set_k} |d_{ij} - d_{ik}|}{|set_j \cap set_k| * max(set_j, set_k)} \tag{3.2}$$

    In Equation (3.2) d_{ij} shows that how many derivations needed, to appear the effect on state i from the driver node j. This weight calculates how similar the distance of the two control node and this measure is normalized with the theoretical maximum.

The sensor similarity is similar to driver similarity, but, in this case, the similarity is calculated to the sensor nodes. In this case, the sets contain the states, which are observable by the given sensor node.

As the calculating of the clusters claim the reachability of the network, we also generate and store the reachability matrix for controlling ($R_{\mathcal{C}}$) and observing ($R_{\mathcal{O}}$) among the clustering metrics.

- Edge centrality measures:
  - **edge betweenness** [4]: shows that how many shortest path going through a given edge.
  - **endpoint similarity**: this is the second new measure, and it is developed to characterize edges. It calculates that, how similar the states in the system, which is controlled, or observed by the two endpoints of the given edge. So if the two endpoints control almost the same set of states, and observe the same set of states, then we can assume that the given edge is a bridge in the system or part of a cycle. There could be more bridges, and with endpoint similarity and edge betweenness we can determine, if there are possibly more or fewer bridges in the system. If there is just one, or a small number of bridges, then they are possibly weak points of the system and has high load, thus this measure can be a fundamental part of the reliability analysis. Another perception is that with the elimination of the small valued edges we can get a clustering of the nodes. The formula which determines the similarity shown in Equation (3.3).

$$\begin{aligned} endpoint\_similarity(e) =& Jaccard(e_{SC}, e_{EC}) * \\ & * Jaccard(e_{SO}, e_{EO}) \end{aligned} \tag{3.3}$$

    In Equation (3.3) e_{SC} and e_{EC} means the sets of states, which are controlled by the start point and by endpoint of edge e, respectively. Analogously, e_{SO} and e_{EO} means the sets of states, which are observed by the start and by the endpoint of edge e. To determine the controlled and observed states for a node, we used the introduced reachability $R_{\mathcal{C}}$ and $R_{\mathcal{C}}$. So the nodes that are reachable by node i, create the controlled set of the states in the system for state i, while the nodes, which can reach node i, generate the set of observed states for state i.
  - **edge similarity**: this is the third new measure, and the aim of this centrality is to determine, how similar the edges behavior in the system. This measure generates a value to every node-pair and shows that how similar the edge's function. The formula is shown in Equation (3.4).

$$\begin{aligned} similarity(e, f) =& Jaccard(e_{SC}, f_{SC}) * Jaccard(e_{EO}, f_{EO}) * \\ & * Jaccard(e_{EC}, f_{EC}) * Jaccard(e_{SO}, f_{SO}) \end{aligned} \tag{3.4}$$

    In Equation (3.4) there are four Jaccard similarity parameterized with different sets. The subscript of the edges e and f means that, we generate the state set of the start point (S) or end point (E) of the edge, and the set is the controlled (C) or observed (O) set. If the provided values are close to zero, then the system is separated, the edges are different, there is not enough redundancy. But if the values close to the value 1, then the edges similar to each other, so there are some redundancy in the system, which is advantageous when we dealing with system robustness.

In the following an example can be seen, that present the implemented measures. The original configuration, i.e. the topology of the system can be seen in Figure 3.1, where color blue note the inputs and color red notes the outputs.

In Figure 3.2, there are 9 states and 15 connections in the system. Density shows that the number of edges near the fifth part of the possible maximum, and the diameter, i.e. the longest

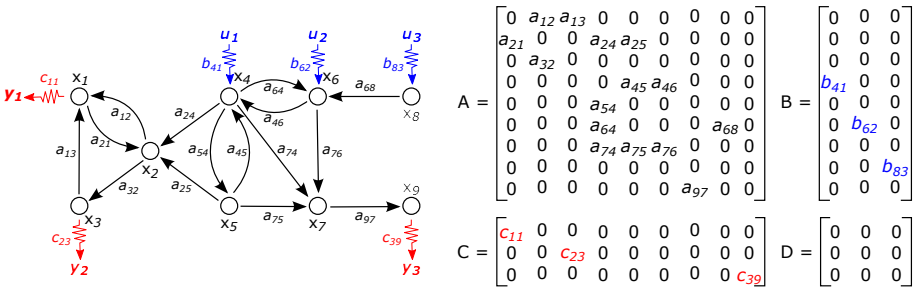
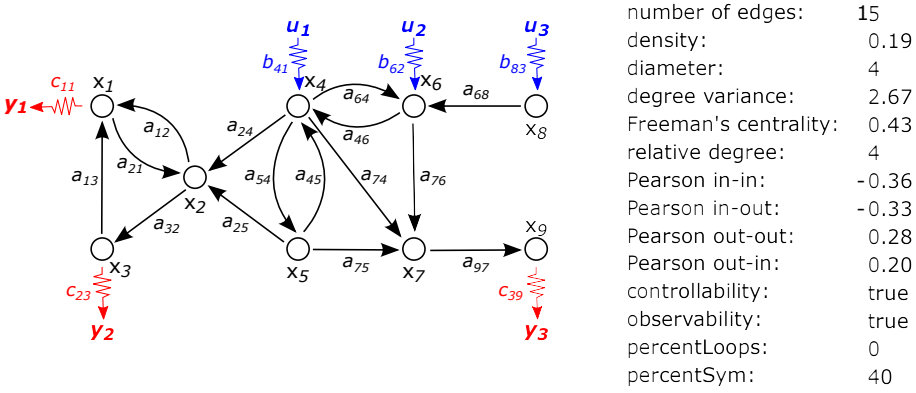


Figure 3.1: Topology and associated configurations of the examined network.

shortest path is 4. The degree variance is 2.67 while the Freeman's centrality is 0.43. The relative degree is also 4 as diameter. The Pearson coefficient shows that the in-in and in-out correlation show assortative nature while out-out and out-in correlation is likely to be disassortative. The system is controllable and observable, and as there is no loop in the network the percentage of loops is 0%, and as there are 6 edges that have symmetric edge pair and the number of connection is 15, the percent of symmetric edge-pairs is 40%. In Figure 3.3 two generated heat maps can be seen, that are not belongs to example system, since such a small example cannot provide clear results.



number of nodes:	9
number of edges:	15
density:	0.19
diameter:	4
degree variance:	2.67
Freeman's centrality:	0.43
relative degree:	4
Pearson in-in:	-0.36
Pearson in-out:	-0.33
Pearson out-out:	0.28
Pearson out-in:	0.20
controllability:	true
observability:	true
percentLoops:	0
percentSym:	40

Figure 3.2: Topology with system measures.

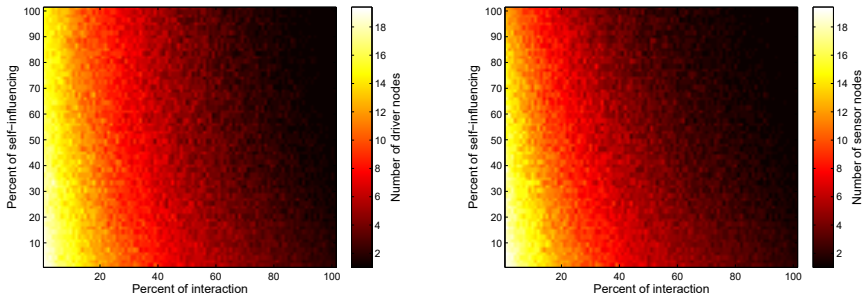
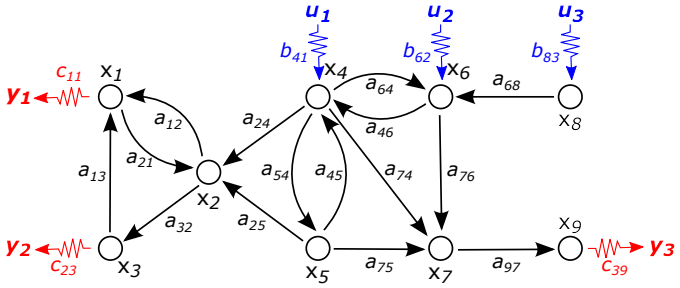


Figure 3.3: Heatmaps generated by the implemented function.

In Figure 3.4 the node centrality measures are presented. One of the most important values is the highest degree, which is belonging to node x_4. Usually, the node with the highest degree is the most central element in a network. The Scott's centrality is a normalized degree, so the most

important node is node x_4 again. The closeness shows which node is the closest to the others in the network. The higher value is the better, and again node x_4 is the central element. Betweenness gives that how many shortest paths going through the given node. If a node has higher value, then it is a critical node in the network. The highest value, 9, is belongs to node x_2 and node x_4. The PageRank gives a percentage for each node, based on how many information visits a given node if Markov-chains are allowed. The highest value belongs to node x_2. The correlation shows the proportion of the number of the neighbors edges and the number of neighbors. This information is raw, and will be useful when we want to determine the system's assortativity. The control centrality and observe centrality measures determine how many states can be influenced or observed by the driver and sensor nodes.



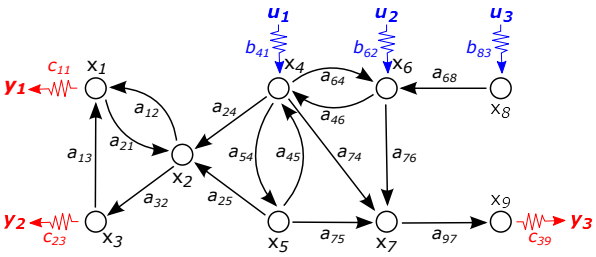
node ID:	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
in degree:	2	3	1	2	1	2	3	0	1
out degree:	1	2	1	4	3	2	1	1	0
degree:	3	5	2	6	4	4	4	1	1
Scott's centrality:	0.19	0.31	0.13	0.38	0.25	0.25	0.25	0.06	0.06
closeness:	4.1	5.6	3.90	7.15	5.95	5.75	4.2	2.91	2.5
betweenness:	1	9	0	9	0	7	4	0	0
PageRank:	25.42	26.46	13.74	6.075	3.78	5.9	7.37	2.49	8.75
correlation in-in:	2	1.67	3	1.5	2	1	1.67	0	3
correlation in-out:	1.5	2.67	2	2.5	4	2.5	3	0	1
correlation out-out:	2	1	1	2	2.33	2.5	0	2	0
correlation out-in:	3	1.5	2	2.25	2.67	2.5	1	2	0
control centrality:	0	0	0	8	0	8	0	9	0
observe centrality:	7	0	7	0	0	0	0	0	6

Figure 3.4: Topology and node centrality measures.

In Figure 3.5 the driver and sensor nodes are indicated by logical vectors. The following four vectors are a kind of resolution of driver and sensor nodes. They are the source, external, internal and inaccessible driver and sensor nodes. In [21] the types are introduced in details. The controlling and observing matrices are sparse while only the drivers and observers columns are nonzero. In Figure 3.5, we converted them to row vectors due to the appropriate visualization. These values contain that how many derivations necessary to influence or observe a state in the system. The similarity of driver nodes x_4 and x_6 is 0.81 instead of 1. Although, they control the same set of nodes, they need different number of derivations to influence the nodes and this is what cause the smaller value. In sensor similarity, sensor nodes x_2 and x_3 observe the same set of nodes and they do it almost exactly the same time, so their similarity is 0.91.

In Figure 3.6, $R_{\mathcal{C}}$ and $R_{\mathcal{O}}$ are the simply reachability matrices. They can say which nodes can be controlled or observed by a given node. In $R_{\mathcal{C}}$ the i^{th} column contains which nodes can control the node i. Another approach is that the elements in row i say which node can be controlled by node i. It is important, that in this example node x_8 can control every node, but it do not guarantee structural controllability. It says that, it is possible to control any of the nodes by node x_8, but it do not say that it is possible within an input configuration. It is thinkable that one driver node can control all the nodes, as in our example, but it is not true in every case. The $R_{\mathcal{O}}$ matrix says the same, but it belongs to the observability, not the controllability.

Finally, the edge centrality measures can be seen in Figure 3.7. The betweenness means the



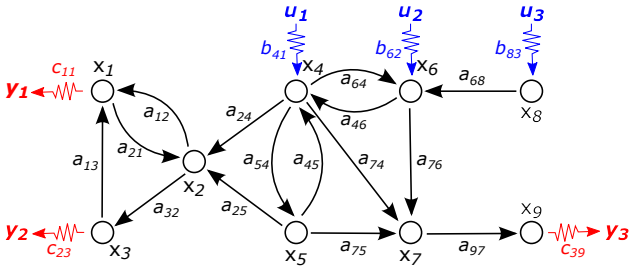
node ID	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
driver nodes:	0	0	0	1	0	1	0	1	0
driver source:	0	0	0	0	0	0	0	1	0
driver external	0	0	0	0	0	0	0	0	0
driver internal	0	0	0	0	0	0	0	0	0
driver inaccesible:	0	0	0	1	0	1	0	0	0
controlling (ID): x_4	3	2	3	1	2	2	2	0	3
x_6	4	3	4	2	3	1	2	0	3
x_8	5	4	5	3	4	2	3	1	4
others	0	0	0	0	0	0	0	0	0

node ID	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
sensor nodes:	1	0	1	0	0	0	0	0	1
sensor source:	0	0	0	0	0	0	0	0	1
sensor external	0	0	0	0	0	0	0	0	0
sensor internal	0	0	0	0	0	0	0	0	0
sensor inaccesible:	1	0	1	0	0	0	0	0	0
observing (ID): x_1	1	2	2	3	3	4	0	5	0
x_3	3	2	1	3	3	4	0	5	0
x_9	0	0	0	3	3	3	2	4	1
others	0	0	0	0	0	0	0	0	0

driver similarity: (ID)	x_4	x_6	x_8
x_4	1	0.81	0.62
x_6	0.81	1	0.71
x_8	0.62	0.71	1

sensor similarity: (ID)	x_1	x_3	x_9
x_1	1	0.91	0.4
x_3	0.91	1	0.4
x_9	0.4	0.4	1

Figure 3.5: First part of node clustering measures with network's topology.

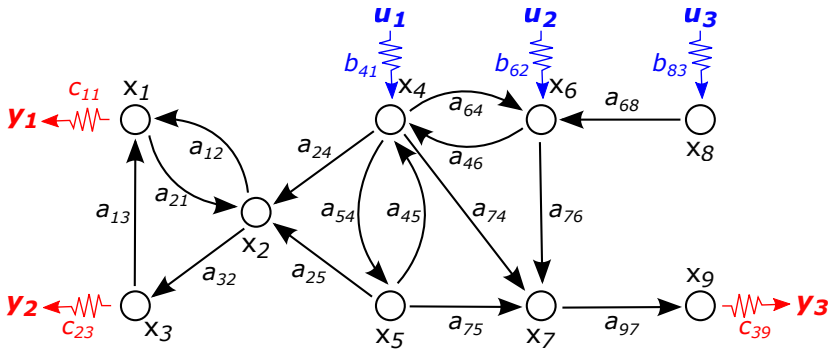


node ID:	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
R_C (ID): x_1	1	1	1	0	0	0	0	0	0
x_2	1	1	1	0	0	0	0	0	0
x_3	1	1	1	0	0	0	0	0	0
x_4	1	1	1	1	1	1	1	0	1
x_5	1	1	1	1	1	1	1	0	1
x_6	1	1	1	1	1	1	1	0	1
x_7	0	0	0	0	0	0	1	0	1
x_8	1	1	1	1	1	1	1	1	1
x_9	0	0	0	0	0	0	0	0	1

node ID:	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
R_O (ID): x_1	1	1	1	1	1	1	0	1	0
x_2	1	1	1	1	1	1	0	1	0
x_3	1	1	1	1	1	1	0	1	0
x_4	0	0	0	1	1	1	0	1	0
x_5	0	0	0	1	1	1	0	1	0
x_6	0	0	0	1	1	1	0	1	0
x_7	0	0	0	1	1	1	1	1	0
x_8	0	0	0	0	0	0	0	1	0
x_9	0	0	0	1	1	1	1	1	1

Figure 3.6: Second part of node clustering measures with network's topology.

same as in node's case, so the number of shortest paths that are going through a given edge [4]. With the value 10, the most critical edge is a_{46}. The endpoint similarity can show that how similar are the influenced and observed sets of states of edge's endpoints. As we mentioned, this metric takes high value, if it is a part of a cycle, or it creates a bridge in the network. As in this network there are no bridges, only cycles can be recognized with this measure. The edge similarity can show how similar are the acting of the edges, and allow to examine redundancy. In the example topology, the nodes x_1, x_2 and x_3, or rather x_4, x_5, x_6 and x_7 create parts of the network that have redundancy, and this is well represented by the edge similarity.



edge ID:		a_{12}	a_{13}	a_{21}	a_{42}	a_{52}	a_{32}	a_{45}	a_{46}	a_{54}	a_{64}	a_{68}	a_{74}	a_{75}	a_{76}	a_{97}
betweenness:		5	2	3	9	3	6	2	10	3	2	8	2	2	4	5
endpoint similarity:		1	1	1	0.21	0.21	1	1	1	1	1	0.22	0.2	0.2	0.2	0.41
edge similarity:	a_{12}	1	1	1	0.21	0.21	1	0.05	0.05	0.05	0.05	0.01	0	0	0	0
(ID)	a_{13}	1	1	1	0.21	0.21	1	0.05	0.05	0.05	0.05	0.01	0	0	0	0
	a_{21}	1	1	1	0.21	0.21	1	0.05	0.05	0.05	0.05	0.01	0	0	0	0
	a_{42}	0.21	0.21	0.21	1	1	0.21	0.21	0.21	0.21	0.21	0.05	0	0	0	0
	a_{52}	0.21	0.21	0.21	1	1	0.21	0.21	0.21	0.21	0.21	0.05	0	0	0	0
	a_{32}	1	1	1	0.21	0.21	1	0.05	0.05	0.05	0.05	0.01	0	0	0	0
	a_{45}	0.05	0.05	0.05	0.21	0.21	0.05	1	1	1	1	0.22	0.2	0.2	0.2	0.02
	a_{46}	0.05	0.05	0.05	0.21	0.21	0.05	1	1	1	1	0.22	0.2	0.2	0.2	0.02
	a_{54}	0.05	0.05	0.05	0.21	0.21	0.05	1	1	1	1	0.22	0.2	0.2	0.2	0.02
	a_{64}	0.05	0.05	0.05	0.21	0.21	0.05	1	1	1	1	0.22	0.2	0.2	0.2	0.02
	a_{68}	0.01	0.01	0.01	0.05	0.05	0.01	0.22	0.22	0.22	0.22	1	0.04	0.04	0.04	0.00
	a_{74}	0	0	0	0	0	0	0.2	0.2	0.2	0.2	0.04	1	1	1	0.08
	a_{75}	0	0	0	0	0	0	0.2	0.2	0.2	0.2	0.04	1	1	1	0.08
	a_{76}	0	0	0	0	0	0	0.2	0.2	0.2	0.2	0.04	1	1	1	0.08
	a_{97}	0	0	0	0	0	0	0.02	0.02	0.02	0.02	0.00	0.08	0.08	0.08	1

Figure 3.7: Calculated edge centrality measures to the topology.

numNodes

Purpose

Determine the number of the nodes in the given network.

Synopsis

[numOfNodes]=numNodes(adj)

Description

The **numNodes** function determines the number of the nodes in the network. The function returns with **numOfNodes**, which contains the number of the nodes. The input matrix could be both directed and undirected graph, and could be both weighted and unweighted.

See Also

`length`

numEdges

Purpose

Determine the number of the edges in the given network.

Synopsis

[numOfEdges]=numEdges(adj)

Description

The **numEdges** function determines the number of the edges in the network. The function returns with **numOfEdges**, which contains the number of the edges. The input matrix could be both directed and undirected graph, and could be both weighted and unweighted.

See Also

nnz

density

Purpose

Determine the density of the given network.

Synopsis

`[dens]=density(adj)`

Description

The **density** function determines the density of the network. The function returns with **dens**, which contains the density of the network. The input matrix could be both directed and undirected graph, and could be both weighted and unweighted.

Algorithm

The density of the network shows the proportion of the existing edges in the network to the all possible edges on the same node set. For undirected simple graph the density is equal with $\frac{2|E|}{|V|(|V|-1)}$, and for directed simple graph with $\frac{|E|}{|V|(|V|-1)}$, where $|E|$ is the number of edges in the graph, and $|V|$ is the number of nodes. Since the systems contain loops, the toolbox uses another way, to calculate the density. In the quotient, the $|V|(|V|-1)$ shows that every node has maximum $|V|-1$ neighbors, but a loop means plus one edge, so this function calculates the density by this formula: $\frac{|E|}{|V|^2}$.

See Also

`numNodes, numEdges`

diameter

Purpose

Determine the diameter of the given network.

Synopsis

`[diam]=diameter(adj)`

Description

The **diameter** function determines the diameter of the network. The function returns with **diam**, which contains the diameter of the network. The input matrix should be directed and could be both weighted and unweighted.

Algorithm

The diameter of the graph is the maximum of the geodesic paths. For calculating the paths, the toolbox uses the matlabBGL toolbox's **all_shortest_paths** function, which uses Johnson's algorithm [9] for calculating the shortest paths.

See Also

`all_shortest_path, max`

degreeVariance

Purpose

The **degreeVariance** function calculates the variance of the nodes' degree for the given network.

Synopsis

`[varianceDegree]=degreeVariance(adj)`

Description

The **degreeVariance** function creates the variance of the nodes' degree in the network. The function returns with **varianceDegree**, which is the calculated measure to the input matrix **adj**. The input matrix shall be directed graph, but could be both weighted and unweighted.

Algorithm

The **degreeVariance** function use the nodes' degree to calculate the measure. The formula is the following: $\frac{\sum_{i=1}^{N}(d(n_i)-d_{mean}(n))^2}{N}$, where N is the number of nodes, $d_{mean}(n)$ is the mean of the degrees and $d(n_i)$ is the degree of node i. The measure is small, if the nodes' degree is almost same, and high, if the degree difference is big between the nodes.

See Also

`numNodes, degree, mean`

degreeFreeman

Purpose

The **degreeFreeman** function generates a measure from the nodes degree to characterize the network.

Synopsis

`[freemanDegree]=degreeFreeman(adj)`

Description

The **degreeFreeman** function creates a measure [5], which can give information about the network. The method uses the nodes' degree to calculate the measure. The function returns with **freemanDegree**, which is the calculated measure to the input matrix **adj**. The input matrix shall be directed graph, but could be both weighted and unweighted.

Algorithm

The **degreeFreeman** function use the nodes' degree to calculate the measure. The formula is the following: $\frac{\sum_{i=1}^{N}(d_{max}(n)-d(n_i))}{(N-1)(N-2)}$, where N is the number of nodes, $d_{max}(n)$ is the maximum of the degrees and $d(n_i)$ is the degree of node i. The measure reaches its maximum, when the network has a star structure.

See Also

`numNodes, degree, max`

isControllable

Purpose

Determine if the given system is controllable.

Synopsis

`[`**controllable**`]=isControllable(`**Amatrix**, **Bmatrix**`)`

Description

The **isControllable** function determines the given system is controllable. The function returns with **controllable**, which is a logical variable, 0 if the system is not controllable, otherwise non-zero. The input matrices, **Amatrix** and **Bmatrix** is the **A** and **B** matrices of the system.

Algorithm

The **isControllable** function uses the Kalman's rank criteria [10] to determine the controllability. A system is controllable, if the rank of the controllability matrix ($\mathcal{C}$) equal with the number of the states (N). The controllability matrix is calculated by: $\mathcal{C} = [\mathbf{B}, \mathbf{AB}, \ldots, \mathbf{A}^{N-1}\mathbf{B}]$.

See Also

`numNodes, sprank`

isObservable

Purpose

Determine if the given system is observable.

Synopsis

`[observable]=isObservable(`**Amatrix**, **Cmatrix**`)`

Description

The **isObservable** function determines the given system is observable. The function returns with **observable**, which is a logical variable, 0 if the system is not observable, otherwise non-zero. The input matrices, **Amatrix** and **Cmatrix** is the **A** and **C** matrices of the system.

Algorithm

The **isObservable** function uses the Kalman's rank criteria [10] to determine the observability. A system is observable, if the rank of the observability matrix ($\mathcal{O}$) equal with the number of the states (N). The observability matrix is calculated by: $\mathcal{O} = [\mathbf{C}^T, (\mathbf{CA})^T, \ldots, (\mathbf{CA}^{N-1})^T]^T$.

See Also

`numNodes, sprank`

degreeRel

Purpose

The **degreeRel** function gives the relative degree of a system configuration.

Synopsis

`[`**relativeDegree**`]=degreeRel(`**Amatrix**, **Bmatrix**, **Cmatrix**`)`

Description

The **degreeRel** function generates the relative degree of the given system. Although the relative degree in its original form requires using Lie derivatives, this relative degree is a simpler perception of relative degree and uses only geodesic paths. The output **relativeDegree** is a value, which is the relative degree of the system. The input matrices are the **A**, **B** and **C** matrices of the system.

Algorithm

The **degreeRel** function generates the simpler perception of relative degree. We calculate the shortest path between all input and output, i.e. from each driver node we generate the shortest path to all reachable sensor nodes. The relative degree is the maximum of these geodesic paths.

See Also

`driverNodes, sensorNodes, issaprse, sparse, all_shortest_paths, max`

pearsonDir

Purpose

The **pearsonDir** function calculates the Pearson correlation [19] on the nodes' degree in a directed network.

Synopsis

`[r]=pearsonDir(`**adj**, **type**`)`

Description

The **pearsonDir** function generates the Pearson's r based on the nodes' degree. With this single value r the degree correlation can be analyze. The input **adj** is the input network, which is directed, and **type** determines the correlation type [1, 20]. The input **type** could be the following: 1: $in - in$, 2: $in - out$, 3: $out - out$, 4: $out - in$. The correlations calculated to a given node. The first direction (in,out) shows which edges determine the actual neighbors. The second direction shows which edges of the neighbors are compared to the examined node's edges. Result **r** is the calculated correlation between the degrees.

Algorithm

The form of **r** can find in many source [1, 20]. In the toolbox the sum of squares based method is used. If the sum of squares is $SS(x)$ or $SS(xy)$, then the form of **r** is the following: $r = \frac{SS(xy)}{\sqrt{SS(x)SS(y)}}$.

See Also

`find, sum, sqrt, sumSq`

sumSq

Purpose

The **sumSq** function calculates the sum of square of two set.

Synopsis

`[sS]=sumSq(x,y)`

Description

The **sumSq** function calculates the sum of squares of the input **x** and **y**. The returned **sS** is the result.

Algorithm

The sum of squares is interpreted by the following: $SS(x) = \sum x^2 \frac{(\sum x)^2}{n}$ or $SS(xy) = \sum xy \frac{(\sum x)(\sum y)}{n}$. The function **sumSq** except two input, so to calculate the first case both input should be the same.

See Also

`iscolumn, length, sum`

percentLoopSym

Purpose

The **percentLoopSym** function determines the level of presence of loops and symmetric edge-pairs in a given network.

Synopsis

`[`**percentLoop**, **percentSym**`]=percentLoopSym(`**adj**`)`

Description

The **percentLoopSym** function determines the percentage of loops compared to all nodes, and the percentage of symmetric edge pairs compared to all edges.

Algorithm

The percentage of loops are based on the proportion of loops ($a_{ii} \neq 0$) and the number of nodes ($|V|$). The percentage of symmetric edge-pairs is based on the proportion of the number of edges that has symmetric pair ($a_{ij} \neq 0 \vee a_{ji} \neq 0$) and the number of edges ($|E|$)

See Also

`length, size, find, diag, triu, sum`

getNodes

Purpose

The **getNodes** function returns the number of driver and sensor nodes necessary for system described by input matrix **A** to be controllable and observable.

Synopsis

`[nDriver, nSensor]=getNodes(A)`

Description

The **getNodes** function determines the number of necessary driver and sensor nodes for system described by input matrix **A**. The output **nDriver** and **nSensor** are the necessary driver and sensor nodes, that are used in the generation of heatmaps.

Algorithm

The **getNodes** function us the path finding method to generate the driver nodes, and sensor nodes. Then it is determined the number of generated driver and sensor nodes.

See Also

`fileparts, which, addpath, fullfile, maximumMatchingPF, length`

addEdges

Purpose

The **addEdges** function changes the input **A** such that new loops and symmetries are added to the original topology.

Synopsis

`[A]=addEdges(`**A**, **freeLoop**, **freeSym**, **numLoop**, **numSym**`)`

Description

The **addEdges** function get a system described by **A**. Inputs **freeLoop** gives the nodes with no loop, while **freeSym** is an edge list with the edges, that are not appearing in the network determined by **A**, but its symmetric pair is part of the network. **numLoop** and **numSym** is the number of the loops and the symmetric edge parts that user want to add the network. If **numLoop** or **numSym** greater than **freeLoop** or **freeSym**, then all free edges are added to network. The output is the changed **A** matrix.

Algorithm

The **addEdges** function firstly determine the number of free loops and symmetries. If the number of required loops and/or symmetries are higher than available, then numLoop and/or numSym is reduced to the number of available loops and/or symmetries. Then the function generate the required edges randomly from the available ones. Then the edges are added to the **A**.

See Also

`length, randperm, sparse`

heatmaps

Purpose

The **heatmaps** function generates two heat maps for showing how the necessary driver and sensor nodes depends on the number of self-influencing and interactions.

Synopsis

`[]=heatmaps(`**A**, **iteration**`)`

Description

The **heatmaps** function generates two heat maps for system described by **A**. The function calculates the number of driver and sensor nodes for each self-influencing - interaction pair **iteration** times, and counts the mean of these results. The function shows the two figures generated by the produced data.

Algorithm

The **heatmaps** function firstly determine the number of free loops and symmetries in the system. Calculates the number of necessary driver and sensor nodes, and the number of edges should be added in each cycle. Then in the cycle, for each calculated additional edges added to the network randomly **iteration** times, and calculate the mean of the necessary driver and sensor nodes. After generating the data, the function shows the two heat maps.

See Also

`fileparts, which, addpath, fullfile, percentLoopSym, find, diag, getNodes, round, zeros, iscolumn, ceil, fprintf, disp ,datestr, addEdges, mean, unique, accumarray, sparse, any, figure, imagesc, title, set, xlabel, ylabel, cbar, colormap,`

degreeIn

Purpose

Determine the in-degree for each node in a network.

Synopsis

`[inDegree]=degreeIn(adj)`

Description

The **degreeIn** function determines the in-degree for each node, i.e. the number of edges, which point to the given node, based on the input matrix **adj**, where $\mathbf{adj}_{ij}$ shows the edge, which going from node i to node j. The function returns with **inDegree** vector, where the i^{th} element is the in-degree of node i. The input matrix shall be directed graph, but could be both weighted and unweighted. The loops are removed before the calculation, so they are excluded from the in-degree.

See Also

`numNodes, sum`

degreeOut

Purpose

Determine the out-degree for each node in a network.

Synopsis

`[outDegree]=degreeOut(adj)`

Description

The **degreeOut** function determines the out-degree for each node, i.e. the number of edges, which point from the given node, based on the input matrix **adj**, where $\mathbf{adj}_{ij}$ shows the edge, which going from node i to node j. The function returns with **outDegree** vector, where the i^{th} element is the out-degree of node i. The input matrix shall be directed graph, but could be both weighted and unweighted. The loops are removed before the calculation, so they are excluded from the out-degree.

See Also

`numNodes, sum`

degree

Purpose

Determine the degree for each node in a network.

Synopsis

`[degree]=degree(adj)`

Description

The **degree** function determines the degree for each node, i.e. the sum of the number of edges, which point from and point to the given node, based on the input matrix **adj**, where $\mathbf{adj}_{ij}$ shows the edge, which going from node i to node j. The function returns with **degree** vector, where the i^{th} element is the degree of node i. The input matrix shall be directed graph, but could be both weighted and unweighted. The loops are removed before the calculation, so they are excluded from the degree.

Algorithm

The **degree** function sums the results of the functions **degreeIn** and **degreeOut**.

See Also

`degreeIn, degreeOut`

degreeScott

Purpose

Normalize the nodes' degree in the given network.

Synopsis

`[scottDegree]=degreeScott(adj)`

Description

The **degreeScott** function normalizes the degree for each node in the input matrix **adj** [22]. The normalization is needed because the degree is not informative in every case, but if we can determine how many neighbors has a node, compared to the theoretical maximum, then we can get a more informative measure. The function returns with **scottDegree** vector, where the i^{th} element is the normalized degree of node i. The input matrix shall be directed graph, but could be both weighted and unweighted.

Algorithm

The **degreeScott** function normalizes the degrees in a network. In [22] Scott normalizes the degree with $N-1$, where N is the number of the nodes. It is a good choice for undirected graphs, but in the system case the network is directed, so the function divides the degree with $2(N-1)$, to normalize the degree.

See Also

`numNodes, degree, zeros`

closenessCentrality

Purpose

The **closenessCentrality** function calculates the closeness, the in and out closeness for a given directed graph.

Synopsis

`[`**closeness**, **inCloseness**, **outCloseness**`]=closenessCentrality(`**adj**`)`

Description

The **closenessCentrality** function generates the closeness centrality for directed graphs. The input of the function is a directed graph, which could be weighted and unweighted. The function returns with **closeness**, **inCloseness**, **outCloseness** vectors, where the i^{th} element of the vector is the closeness, in closeness or out closeness of node i.

Algorithm

The **closenessCentrality** function generates the closeness for every node in a directed graph. For undirected graph, the closeness gives that how close a node in the graph for the others. In directed case, we distinguish in and out closeness, according to that, which nodes are reachable from a given node, and which of them can reach it. The in closeness calculated by the following formula: $in\_closeness(i) = |reach(i)|^2 / \sum_{j \in V} d_{ij}$. In the formula, $reach(i)$ gives the nodes, which can reach node i. In the quotient, there is the sum of the distance from each node to node i. If a node cannot reach one, the distance is 0. For out closeness, the formula is the same, but $reach(i)$ gives the nodes, which are reachable from node i. The closeness is the sum of the in and out closeness.

See Also

`issparse, all_shortest_path, sparse, sum, isnan`

nodeBetweenness

Purpose

The **nodeBetweenness** function calculates the betweenness of the nodes' in the given network.

Synopsis

`[betweenness]=nodeBetweenness(adj)`

Description

The **nodeBetweenness** function creates the node betweenness in the network. The function returns with **betweenness** vector, where the i^{th} element is the betweenness measure of node i. The input matrix, **adj**, shall be directed graph, but could be both weighted or unweighted.

Algorithm

The **nodeBetweenness** function calculates the betweenness measure for every node in the given network. If we note the number of geodesic paths between node i and j with σ_{ij}, and we say that $\sigma_{ij}(n_k)$ is number of geodesic path between node i and j such that, node k is included in the path, $k \neq i$ and $k \neq j$, then the betweenness measure for node k is $\sum_{i \neq j \neq k} \frac{\sigma_{ij}(n_k)}{\sigma_{ij}}$. For calculating the betweenness, the toolbox uses the matlabBGL toolbox's **betweenness_centrality** function.

See Also

`betweenness_centrality, sparse`

pageRank

Purpose

The **pageRank** function generates the page rank values to all nodes in the given network.

Synopsis

`[pageR]=pageRank(adj)`

Description

The **pageRank** function creates the page rank values to every node in the network. The function returns with **pageR** vector, where the i^{th} element is the page rank of node i. The input matrix, **adj**, shall be directed graph and the weights are important as well.

Algorithm

The **pageRank** function calculates the page rank values to every nodes. After the calculation, the function change the values to percentage, so it determines how possible a state get a control. To calculate the values the function uses the following form: $\mathbf{x} = \alpha \mathbf{adj}^T \mathbf{D}^{-1} \mathbf{x} + \beta \mathbf{1}$, where $\mathbf{x}$ is the vector which contains the PageRank values, $\mathbf{adj}^T$ is the transpose of matrix **adj**, the matrix $\mathbf{D}$ is a diagonal matrix, where $\mathbf{D}_{ii} = max(degree_{out}(i), 1)$, and $\mathbf{1}$ is a column vector where every value is 1. The parameter α is 0.85, and this give the possibility of random walks in the system. Parameter β is not so important, it is a free given value to every state. In the function $\beta = 1$.

See Also

`numNodes, degreeOut, sum, diag, \, ones, isempty`

degreeCorrel

Purpose

The **degreeCorrel** function calculates the degree correlation for each node in the network.

Synopsis

[in_in, in_out, out_out, out_in]=degreeCorrel(adj)

Description

The **degreeCorrel** function generates the degree correlation of each node, i.e. the average degree of the neighbors. Four different degree correlations can be distinguished, according to the four output variable. Each of them is a vector with length of number of states and contains the degree correlation. In each vector element i shows the degree correlation of node i, according to the given type of correlation. The output variables form is X_Y, where X and Y can be *in* or *out*. X means the direction between the node and the neighbor while Y means the counted edges at the neighbors.

Algorithm

The **degreeCorrel** function calculate the degree correlation (more detail in chapter 7 of [1]). The four type of the degree correlation originated from the direction of the edges. While examining a node, then we can calculate the correlation with its in-neighbors, i.e. the neighbors where an edge is pointing from the neighbor to the node or out-neighbor. Furthermore, the edge direction could be in and out during degree counting at the neighbors. For node i the in_out degree correlation is: $\frac{|neighbors(i)\_out\_edges|}{|in\_neighbors(i)|}$.

See Also

`degreeIn, degreeOut`

controlCentrality

Purpose

Define control centrality measure for driver nodes.

Synopsis

`[CC]=controlCentrality(Amatrix,Bmatrix)`

Description

The **controlCentrality** function calculates the control centrality measures [13] based on matrices **A** and **B**. The function returns with **CC** vector, which has as many elements as many states are in the system. The i^{th} value in **CC** is the control centrality value of state i, if it is a driver node. If node i is not a driver node, then its control centrality value is zero.

Algorithm

The **controlCentrality** function uses a modified controllability matrix [10, 15] to define the control centrality measures. The modified controllability matrix arises from the matrix **A** and the $\mathbf{b}_i$, where $\mathbf{b}_i$ is a N-length column-vector with one non-zero entry on the i^{th} position, and its rank gives the control centrality for driver node i. The modified controllability matrix formal definition: $\mathcal{C}_i = [\mathbf{b}_i, \mathbf{A}\mathbf{b}_i, \ldots, \mathbf{A}^{N-1}\mathbf{b}_i]$. The toolbox uses the Dulmage-Mendelson decomposition [18, 3] to calculate the rank of the controllability matrix.

See Also

`numNodes, zeros, find, sum, sprank`

observeCentrality

Purpose

Define observe centrality measure for sensor nodes.

Synopsis

`[OC]=observeCentrality(Amatrix,Cmatrix)`

Description

The **observeCentrality** function calculates the observe centrality measures based on matrices **A** and **C**, analogous to the control centrality. The function returns with **OC** vector, which has as many elements as many states are in the system. The i^{th} value in **OC** is the observe centrality value of state i, if it is a sensor node. If node i is not a sensor node, then its observe centrality value is zero.

Algorithm

The **observeCentrality** function uses a modified observability matrix to define the observe centrality measures. The modified observability matrix arises from the matrix **A** and the $\mathbf{c}_i$, where $\mathbf{c}_i$ is a N-length row-vector with one non-zero entry on the i^{th} position, and its rank gives the observe centrality for sensor node i. The modified observability matrix formal definition: $\mathcal{O}_i = [\mathbf{C}_i^T, (\mathbf{C}_i\mathbf{A})^T, \ldots, (\mathbf{C}_i\mathbf{A}^{N-1})^T]^T$. The toolbox uses the Dulmage-Mendelson decomposition [18, 3] to calculate the rank of the observability matrix.

See Also

`numNodes, zeros, find, sum, sprank`

driverNodes

Purpose

The **driverNodes** function determines which states are driver nodes in a system.

Synopsis

`[drivers]=driverNodes(Bmatrix)`

Description

The **driverNodes** function generates the output row vector **drivers**, which length is N, where N is the number of states. The i^{th} element is 1, if the state i is a driver node, otherwise 0. The input **Bmatrix** is the **B** matrix of the state-space equation.

Algorithm

The **driverNodes** function determines the driver nodes from **Bmatrix**. If an input signal affects a state, then it is a driver node. The structure of **Bmatrix** make possible to easily count out the control nodes. If the i^{th} row of the **Bmatrix** contain at least one non-zero, then it is a driver node.

See Also

`sum`

driverType

Purpose

The **driverType** function determines the type of a driver node according to [21].

Synopsis

[source, external, internal, inaccessible]=`driverType(`**Amatrix, Bmatrix**`)`

Description

The **driverType** function generates four output. **source** is the driver node, which has no input edges, so they necessary to control because they cannot get any signal from any other node. **external** is the driver nodes, which are controlled by a separated input because they are part of an external dilation [21]. **internal** is contain the driver nodes which belong to internal dilations. **inaccessible** is the driver nodes, which have input edge, and are not internal or external dilation, but no signal can reach them. Each output variable is a logical vector with length of number of states in the system. If element i is 1, then the driver node i is belong to the given type, 0 otherwise.

Algorithm

To generate **source**, we have to take the intersection of the driver nodes and the states with zero in-edge. **external** is analogous to **source**, but, in this case, we care with the nodes with no out-edge. The **internal** could be count by checking the following: if a set of nodes contains more nodes than its set of parents, then it is an internal dilation. The remaining driver nodes belong to **inaccessible**.

See Also

`driverNodes, numNodes, sum, &, find, issparse, sparse, zeros, isequal, length`

sensorNodes

Purpose

The **sensorNodes** function determines which states are sensor nodes in a system.

Synopsis

`[observers]=sensorNodes(Cmatrix)`

Description

The **sensorNodes** function generates the output row vector **observers**, which length is N, where N is the number of states. The i^{th} element is 1, if the state i is a sensor node, otherwise 0. The input **Cmatrix** is the **C** matrix of the state-space equation.

Algorithm

The **sensorNodes** function determines the sensor nodes from **Cmatrix**. The output signals are appearing on the sensor nodes. The structure of **Cmatrix** make possible to easily count out the sensor nodes. If the i^{th} column of the **Cmatrix** contain at least one non-zero, then it is a sensor node.

See Also

`sum`

sensorType

Purpose

The **sensorType** function determines the type of a sensor node analogously to function driverType.

Synopsis

[**source**, **external**, **internal**, **inaccessible**]=`sensorType(`**Amatrix**, **Cmatrix**)

Description

The **sensorType** function generates four output. **source** is the sensor node, which has no output edges, so they necessary to observe because they cannot send any signal to any other node. **external** is the sensor nodes, which are observed by a separated output because they are part of an external dilation [21]. **internal** is contain the sensor nodes which belong to internal dilations. **inaccessible** is the sensor nodes, which have input edge, and not part of internal or external dilation, but no signal can get from them. Each output variable is a logical vector with length of number of states in the system. If element i is 1, then the sensor node i is belong to the given type, 0 otherwise.

Algorithm

To generate **source**, we have to take the intersection of the sensor nodes and the states with zero out-edge. **external** is analogous to **source**, but, in this case, we care with the nodes with no in-edge. The **internal** could be count by checking the following: if a set of nodes contains more nodes than its set of parents, then it is an internal dilation. The remaining sensor nodes belong to **inaccessible**.

See Also

`driverNodes, numNodes, sum, &, find, issparse, sparse, zeros, isequal, length, driverType`

clusterControl

Purpose

The **clusterControl** function creates a matrix, which contains the control flow, i.e. that, how many derives need to control a state by a driver node.

Synopsis

`[`**controlling**`]=clusterControl(`**Amatrix**, **Bmatrix**`)`

Description

The **clusterControl** function generates the controlling of the system. The input **Amatrix** and **Bmatrix** are the **A** and **B** matrices of the system. The output **controlling** is a matrix, where the i^{th} row and j^{th} column contain the number of derives, which need to influence state i by driver node j. If the element **controlling**(i, j) is zero, then state j is not a driver node, or it does not influence state i.

Algorithm

The **clusterControl** function determines how many derivations need to influence the states in a system by the driver nodes. It uses the matlabBGL toolbox's function to find all shortest path. The number of necessary derives equal with the shortest path plus one because one derivation needs to influence the first state.

See Also

`sum, all_shortest_paths, sparse, issparse`

clusterObserve

Purpose

The **clusterObserve** function creates a matrix, which contains the observe flow, i.e. that, how many derives need to observe a state by a sensor node.

Synopsis

`[`**observing**`]=clusterObserve(`**Amatrix**, **Cmatrix**`)`

Description

The **clusterObserve** function generates the observing of the system. The input **Amatrix** and **Cmatrix** are the **A** and **C** matrices of the system. The output **observing** is a matrix, where the i^{th} row and j^{th} column contain the number of derives, which is necessary to observe the state i by sensor node j. If the element **observing**(i, j) is zero, then state j is not a sensor node, or it does not influence state i.

Algorithm

The **clusterObserve** function determines how many derivations need to observe the states in a system by the sensor nodes. It uses the matlabBGL toolbox's function to find all shortest path. The number of necessary derives equal with the shortest path plus one because one derivation needs to observe the first state.

See Also

`sum, all_shortest_paths, sparse, issparse`

driverSimilarity

Purpose

The **driverSimilarity** function determines the similarity of the driver nodes.

Synopsis

`[drivSimilarity]=driverSimilarity(`**Amatrix**, **Bmatrix**`)`

Description

The **driverSimilarity** function determines the similarity of the control nodes in a system. It uses the state-space model's **A** and **B** matrices. The output matrix, **drivSimilarity**, is an $N \times N$ matrix, where N is the number of states in the system, and the i. row and j. column shows the similarity of state i and j. If at least one of them is not a driver node, then the similarity is 0.

Algorithm

The **driverSimilarity** function determines the driver nodes' similarity according to Jaccard's similarity [8]. The function not just determines the similarity of the controlled nodes, but weights it according to **simWeighting**. It calculates how similar the sets that are controlled by the two driver nodes, then weights it according to **simWeighting** function.

See Also

`numNodes, zeros, find, controlNodes, clusterControl, jaccard, simWeighting, issparse, sparse`

sensorSimilarity

Purpose

The **sensorSimilarity** function determines the similarity of the sensor nodes.

Synopsis

`[senSimilarity]=sensorSimilarity(`**Amatrix**, **Cmatrix**`)`

Description

The **sensorSimilarity** function determines the similarity of the sensor nodes in a system. It uses the state-space model's **A** and **C** matrices. The output matrix, **senSimilarity**, is an $N \times N$ matrix, where N is the number of states in the system, and the i. row and j. column shows the similarity of state i and j. If at least one of them is not a sensor node, then the similarity is 0.

Algorithm

The **sensorSimilarity** function determines the sensor nodes' similarity according to Jaccard's similarity [8]. The function not just determines the similarity of the observed nodes, but weights it according to **simWeighting**. It calculates how similar the sets that are observed by the two control nodes, then weights it according to **simWeighting** function.

See Also

`numNodes, zeros, find, sensorNodes, clusterObserve, jaccard, simWeighting, issparse, sparse`

jaccard

Purpose

The **jaccard** function generates the similarity of two sets.

Synopsis

`[`**jaccard_similarity**`]=jaccard(`**set1**, **set2**`)`

Description

The **jaccard** function creates the similarity of two sets. The sets are given in binary vector format, so the vectors' i^{th} element is 1 if the element i is in the set, otherwise 0. The returned **jaccard_similarity** is a number, which is 1 if the two sets are the same, and 0 if there is no same element in the two sets, and it could be between this two value according to the two sets. The similarity was published by Jaccard in 1901 [8].

Algorithm

The **jaccard** function determine the similarity of two sets according to Jaccard publication [8]. The algorithm returns with the value calculated by the cardinality of the section of the two sets divided by the cardinality of the union of them. If both sets are empty, then the similarity is zero. With formula: $similarity(set_1, set_2) = \frac{|set_1 \cap set_2|}{|set_1 \cup set_2|}$.

See Also

`|, &, sum`

simWeighting

Purpose

The **simWeighting** function calculates a weight of two sets similarity, where the elements' distance are determinable.

Synopsis

`[weight]=simWeighting(set1, set2)`

Description

The **simWeighting** function creates a value, **weight**, which can weights the similarity of two sets. The input **set1** and **set2** are row vectors, where the i^{th} value shows that how far the element i. If this value is 0, then the element i is not contained by the set.

Algorithm

The **simWeighting** function determine a value to characterize the similarity, if the distance of the elements are determinable. To generate the value, the function uses the following formula: $weight(set_1, set_2) = 1 - \frac{\sum_{i \in set_1 \cap set_2} |d_{i1} - d_{i2}|}{|set_1 \cap set_2| * max(set_1, set_2)}$, i.e. divide the sum of absolute difference of the common elements with the worst case, and subtract it from 1. If the similarity is high, and the distances are almost the same, then the output is close to 1, if the distance difference is high the output close to 0.

See Also

`find, &, abs, length`

reachC

Purpose

The **reachC** function determines which nodes can control a given one.

Synopsis

`[Rc]=reachC(Amatrix)`

Description

The **reachC** function generates the output **Rc** matrix which is a simple reachability of the given state-transition matrix, **Amatrix**. The i^{th} column of the output matrix is a binary vector, where the j^{th} element is 1 if node j can control node i, 0 otherwise. Another approach is that the i^{th} row of the matrix is a binary vector, where element j is 1 if node i can control j, 0 otherwise. If node i can control all the nodes (the associated row contains only 1), that does not mean that node i can control all the states. It means that there is a control configuration, where any of the nodes can be controlled by node i.

Algorithm

The **reachC** function generate a simple reachability on the transpose of input **Amatrix**.

See Also

`reachability`

reachO

Purpose

The **reachO** function determines which nodes can observe a given one.

Synopsis

`[observable]=reachO(Amatrix)`

Description

The **reachO** function generates the output **observable** matrix which is a simple reachability of the given state-transition matrix, **Amatrix**. The i^{th} column of the output matrix is a binary vector, where the j^{th} element is 1 if node j can observe node i, 0 otherwise. Another approach is that the i^{th} row of the matrix is a binary vector, where element j is 1 if node i can observe j, 0 otherwise. If node i can observe all the nodes (the associated row contains only 1), that does not mean that node i can observe all the states. It means that there is an observing configuration, where any of the nodes can be observed by node i.

Algorithm

The **reachO** function generate a simple reachability on the input **Amatrix**.

See Also

`reachability`

reachability

Purpose

The **reachability** function determines the reachability of the network.

Synopsis

`[reach]=reachability(adj)`

Description

The **reachability** function generates the output **reach** matrix which is a simple reachability of the given network described by **adj**. The input matrix could be weighted and unweighted, but should be directed. The i^{th} column of the output matrix is a logical vector, where the j^{th} element is 1 if node j can reach node i, 0 otherwise. Another approach is that the i^{th} row of the matrix is a binary vector, where element j is 1 if node i can reach node j, 0 otherwise.

Algorithm

The **reachability** function generate a simple reachability on the input network **adj**.

See Also

`|, eye, size, zeros, prod`

edgeBetweenness

Purpose

The **edgeBetweenness** function calculates the betweenness of the edges in the given network.

Synopsis

`[betweenness]=edgeBetweenness(adj)`

Description

The **edgeBetweenness** function creates the edges' betweenness in the network. The function returns with **betweenness** sparse matrix, where the value of i^{th} row and j^{th}column shows the betweenness measure of edge, which going from node i to node j. The input matrix, **adj**, shall be directed graph, but could be both weighted or unweighted.

Algorithm

The **edgeBetweenness** function calculates the betweenness measure for every edge in the given network. If we note the number of geodesic paths between node i and j with σ_{ij}, and we say that $\sigma_{ij|e}$ is number of geodesic path between node i and j such that, edge e is included in the path, then the betweenness measure for edge e is $\sum_{i,j \in V} \frac{\sigma_{ij|e}}{\sigma_{ij}}$. For calculating the betweenness, the toolbox uses the matlabBGL toolbox's **betweenness_centrality** function.

See Also

`betweenness_centrality, sparse`

endpointSim

Purpose

The **endpointSim** function calculates a measure for determine the similarity of the endpoints of an edge in the given network.

Synopsis

`[similarityEndp]=endpointSim(adj)`

Description

The **endpointSim** function creates a sparse matrix with the endpoint similarities. The returned **similarityEndp** contains the similarities, where the i^{th} row and j^{th} column contains the similarity of the endpoints of edge that going from node i to node j. If there is no edge between a node-pair, then the similarity is 0. The input matrix, **adj**, shall be directed graph, but could be both weighted or unweighted.

Algorithm

The **endpointSim** function uses Jaccard's similarity to generate similarity for the edges. To get the similarity, we have to multiply the Jaccard similarity of the controlled and observed sets of the endpoints of the edges. With formula: $endpointSim(e) = Jaccard(e_{SC}, e_{EC}) * Jaccard(e_{SO}, e_{EO})$. The subscript of e means that, we generate the state set of the start point (S) or end point (E) of the edge, and the set is the controlled (C) or observed (O) set. The controlled set is the nodes, which are reachable from the node, and the observed set is containing those nodes, which can reach the given node.

See Also

`reachability, find, numNodes, numEdges, sparse, zeros, jaccard`

similarity

Purpose

The **similarity** function calculates a measure for each edge-pair in the given network.

Synopsis

`[`**similarityEdge**, **edges**`]=similarity(`**adj**`)`

Description

The **similarity** function creates the similarity measure for each edge-pair in the network. The function returns with **similarityEdge** matrix, which size is $|E| \times |E|$, where the i^{th} row and j^{th} column contain the similarity of edge i and j. The sequence of the edges are gives by the vector **edges**, where $edges(i, 1)$ the startpoint, and $edges(i, 2)$ the endpoint of edge i. The input matrix, **adj**, shall be directed graph, but could be both weighted or unweighted.

Algorithm

The **similarity** function uses Jaccard's similarity to generate similarity for edge-pairs. To get the similarity, we have to multiply the Jaccard similarity of the controlled and observed sets of the endpoints of the edges. With formula: $similarity(e, f) = Jaccard(e_{SC}, f_{SC}) * Jaccard(e_{EO}, f_{EO}) * Jaccard(e_{EC}, f_{EC}) * Jaccard(e_{SO}, f_{SO})$. The subscript of e and f means that, we generate the state set of the start point (S) or end point (E) of the edge, and the set is the controlled (C) or observed (O) set. The controlled set is the nodes, which are reachable from the node, and the observed set is containing those nodes, which can reach the given node.

See Also

`reachability, jaccard, numEdges, find, zeros, issparse, sparse`

getConfig

Purpose

The **getConfig** function produces a structure, which contains what measures will be generated by **matricesToStruct**.

Synopsis

`[mesConfig]=getConfig()`

Description

The **getConfig** function creates a structure, which associate a value $[0, 1]$ for each measure contained by this module. The resulted **mesConfig** is an input parameter of funciton **matricesToStruct**.

Algorithm

The function is only the declaration of the resulted structure. With the edition of the file, the generated configuration can be edited.

See Also

`struct`

matricesToStruct

Purpose

The **matricesToStruct** function creates a structure of metrics from the input system.

Synopsis

`[data]=matricesToStruct(`**Amatrix**, **Bmatrix**, **Cmatrix**, **Dmatrix**, **mesConfig**`)`

Description

The **matricesToStruct** function generate a structure from the measures of the input system. The system described with the input matrices **Amatrix**, **Bmatrix**, **Cmatrix** and **Dmatrix**. The parameter **mesConfig** is a structure generated by **getConfig** function, and it contains the information to which measures will be generated. The output **data** is a structure, which contain the calculated measures of the system. The **data** structure is the following:

- data
  - system
    - describe
      - A
      - B
      - C
      - D
      - effectGraph
      - Graph
    - measure
      - controllable
      - observable
      - numOfNodes
      - numOfEdges
      - density
      - diameter
      - degreeFreeman
      - degreeVariance
      - degreeRelative
      - rInIn
      - rInOut
      - rOutOut
      - rOutIn
      - percentLoops
      - percentSym
  - node
    - centrality
      - degreeIn
      - degreeOut
      - degree
      - degreeScott
      - inCloseness
      - outCloseness
      - closeness
      - betweenness
      - pageRank
      - in_inCorrel
      - in_outCorrel
      - out_outCorrel
      - out_inCorrel
      - control
      - observe
    - cluster
      - driverNodes
      - driverSource
      - driverExternal
      - driverInternal
      - driverInaccess
      - sensorNodes
      - sensorSource
      - sensorExternal
      - sensorInternal
      - sensorInaccess
      - controlling
      - observing
      - driverSimilarity
      - sensorSimilarity
      - Rc
      - Ro

  - edge
    * edges
    * betweenness
    * similarity
    * endpointSim

The data **Graph** is another structure in **data – system – describe**. This structure is describe the effect graph of the system, and this structure is compatible with "Complex Networks Package for MatLab" [16]. The structure is the following:

- Graph
  - Type
  - Signature
  - FileName
  - Data
  - Index
    * Names
    * Values
    * Exist
    * Properties

Another important variable is the **data – system – describe – effectGraph**. This is a binary matrix, which contains how the states influence each other. It describes a network which is unweighted. So the matrix is the transpose of the system's **A** matrix, and an element is 1, only if in **A** matrix the according element is non-zero.

Algorithm

The **matricesToStruct** function uses the functions developed in this toolbox, to analyzes and creates a structure from a system. The function runs all implemented function listed below.

See Also

`struct, find, length, cell, num2str, isControllable, isObservable, numNodes, numEdges, density, diameter, degreeFreeman, degreeVariance, degreeRel, percentLoopSym, degreeIn, degreeOut, degree, degreeScott, closenessCentrality, nodeBetweenness, pageRank, degreeCorrel, controlCentrality, observeCentrality, driverNodes, driverType, sensorNodes, sensorType, clusterControl, clusterObserve, driverSimilarity, sensorSimilarity, reachC, reachO, edgeBetweenness, similarity, endpointSim, pearsonDir, getConfig`

Chapter 4

System investigation

The system investigation module is the last module of the NOCAD toolbox, which task is to loading the example networks, that are found in literature, to modifying the networks according to the connection types, to calling the functions implemented in module 1 and 2 and to creating comparison tables based on the generated structure of module 2. The network loader and the structure generator functions have been developed to be easily expandable with new entries. This is true to the generator of comparison table as well. It is important because the computational time can be very high depending on the system's size. After the generation, the module collects and produces some measures to describe the networks and creates a comparison tables from these metrics for all introduced connection type. The fields of the comparison tables can be seen in Table 4.1.

Table 4.1: Fields of a comparison table.

General network measures	
networks name	diameter
number of nodes	relative degree
number of edges	mean of degree
density	
Driver nodes' measures	
number of driver nodes	number of internal driver nodes
number of source driver nodes	number of inaccessible driver nodes
number of external driver nodes	percentage of driver nodes
Sensor nodes' measures	
number of sensor nodes	number of internal sensor nodes
number of source sensor nodes	number of inaccessible sensor nodes
number of external sensor nodes	percentage of sensor nodes
Pearson correlation measures	
Pearson correlation in-in	Pearson correlation out-out
Pearson correlation in-out	Pearson correlation out-in
Structural phenomena	
percentage of loops	percentage of symmetries

createDataAndTables

Purpose

The **createDataAndTables** script generates the four connection types to each network found in the 'networks' directory into and creates the data structure into 'data' directory. Comparison tables also generated into 'tables' directory.

Synopsis

```
createDataAndTables
```

Description

The **createDataAndTables** script load the networks from 'networks' directory, creates the four systems according to connection types, then creates data structures into directory 'data'. Comparison tables also created into directory 'tables'.

Algorithm

The **genTable** script uses functions included in the previous two modules.

See Also

`dir, length, exist, mkdir, getConfig, genCol, fileparts, fprintf, strcat, load, eval, clear, generateMatricesPF, matricesToStruct, speye, save, xlswrite,`

genCol

Purpose

The **genCol** function creates a column of comparison table from the data structures generated by **matricesToStruct** for a system.

Synopsis

`[col]=genCol(`**netw**, **name**`)`

Description

The **genCol** function generates a column of comparison a table. Before the calling of this function, the **matricesToStruct** function should be called for the data structure, **netw**, necessary for **genCol**. **name** attribute is a string, that contain the name of the system. The comparison tables contain the measures can be seen in Table 4.1. This function generates a column only, i.e. it generates the measures for only one system. If the input variable **netw** is not a structure, then the returned column contains the label of the rows.

Algorithm

The **genCol** function uses the data generated by **matricesToStruct** to calculate the measures for a comparison table.

See Also

`strcat, cell, eval, sprintf, load, sum, mean`

Bibliography

[1] Barabási A.-L., *Network Science*. Cambridge University Press, 2015.

[2] Duff, I. S. *On Algorithms for Obtaining a Maximum Transversal*. ACM Transactions on Mathematical Software, Volume 7 Issue 3, Pages 315-330 1981.

[3] Dulmage, A. L. & Mendelsohn, N. S., *Coverings of bipartite graphs*. Canad. J. Math. 10: 517534., 1958.

[4] Freeman, L. C. *A set of measures of centrality based upon betweenness*. Sociometry, Vol. 40, No. 1, pp. 35-41 1977.

[5] Freeman, L. C. *Centrality in social networks conceptual clarification*. Social Networks, Volume 1, Issue 3, Pages 215239, 1979.

[6] Hangos, K.M. and Cameron, I.T. *Process Modelling and Model Analysis*. Academic Press, London, pp. 1-543. 2001.

[7] Hopcroft, J. E. and Karp, R. M. *An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs*. SIAM J. Comput., 2(4), 225231. 1973.

[8] Jaccard, P. *Distribution de la flore alpine dans le bassin des Dranses et dans quelques rgions voisines*. Bulletin de la Socit Vaudoise des Sciences Naturelles 37, 241-272, 1901.

[9] Johnson, D. B. *Efficient Algorithms for Shortest Paths in Sparse Networks*. Journal of the ACM (JACM), Volume 24 Issue 1, Pages 1-13 , 1977.

[10] Kalman, R. E., *Mathematical description of linear dynamical systems*. J. Soc. Indus. Appl. Math. Ser. A 1, 152192, 1963.

[11] Liu, X., Pequito, S., Kar S., Mo, Y., Sinopoli, B., and Aguiar, A. P. *Minimum Robust Sensor Placement for Large Scale Linear Time-Invariant Systems: A Structured Systems Approach*. Proc. of NecSys13 - 4th IFAC Workshop on Distributed Estimation and Control in Networked Systems, 2013.

[12] Liu Y.-Y, Slotine J.-J. and Barabási A.-L., *Controllability of complex networks*. Nature 473, 167173, 2011.

[13] Liu Y.-Y, Slotine J.-J. and Barabási A.-L., *Control Centrality and Hierarchical Structure in Complex Networks*. PLoS ONE 7(9): e44459 , 2012.

[14] Liu Y.-Y, Slotine J.-J. and Barabási A.-L., *Observability of complex systems*. PNAS 2013 110 (7) 2460 - 2465 , 2013.

[15] Luenberger, D. G., *Introduction to Dynamic Systems: Theory, Models, & Applications*. Wiley, 1979.

[16] Muchnik, L. , *Complex Networks Package for MatLab (Version 1.6)*. `http://www.levmuchnik.net/Content/Networks/ComplexNetworksPackage.html`, 2013.

[17] Newman, M. *Assortative mixing in networks*. Phys. Rev. Lett., 89, 2002.

[18] Pothen, A. and Fan C.-J., *Computing the Block Triangular Form of a Sparse Matrix*. ACM Transactions on Mathematical Software Vol 16, No. 4 pp. 303-324., 1990.

[19] Pearson, K. *Notes on Regression and Inheritance in the Case of Two Parents*. Proceedings of the Royal Society of London, 58: 240-242. 1895.

[20] Pósfai, M., Liu, Y.-Y., Slotine, J.-J. and Barabási, A.-L. *Effect of correlations on network controllability*. Scientific Reports 3, Article number: 1067 1895.

[21] Ruths, J. and Ruths, J., *Control Profiles of Complex Networks*. Science, 343 (6177), 1373-1376., 2014.

[22] Scott, J. *Social Network Analysis: A Handbook*. Sage, London, 2nd ed., 2000.

[23] Slotine, J.-J. and Li, W., *Applied Nonlinear Control*. Prentice-Hall, 1991.

[24] Tarjan, R.E., *Depth first search and linear graph algorithms*. SIAM Journal on Computing 1(2), 146160., 1972.

[25] Young, N. E. *Greedy set-cover algorithms*. In M.-Y. Kao (Ed.), Encyclopedia of algorithms (pp. 379381)., 2008.