Manual of NOCAD - **N**etwork based **O**bservability and **C**ontrollability **A**nalysis of **D**ynamical Systems toolbox for MATLAB and OCTAVE

Dániel Leitold, Ágnes Vathy-Fogarassy, János Abonyi

June 3, 2019

Contents

Chapter 1

Introduction

1.1 The NOCAD toolbox

The implemented NOCAD toolbox (**N**etwork based **O**bservability and **C**ontrollability **A**nalysis of **D**ynamical Systems toolbox) is divided into modules that are: network mapping, system characterisation and improvement and robustness. Accordingly to the implemented functions, the toolbox deals with the workflow shown in Figure 1.1.

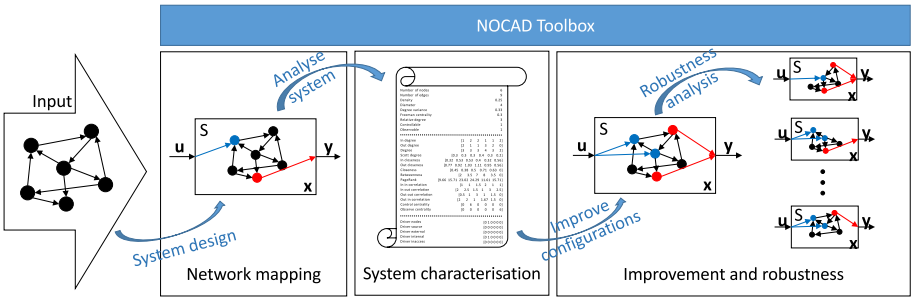


Figure 1.1: Workflow of the utilisation of the NOCAD toolbox. The network mapping module provides two methods to create a dynamical system based on the topology of the state variables. The system characterisation module generates more than 49 measures to analyse, classify and characterise the developed system. The improvement and robustness module offers five algorithms to improve the system with additional inputs (observers) as well as outputs (controllers), and can analyse the robustness of the designed system.

The *network mapping module* creates a dynamical system from a given network structure, i.e. the necessary matrices of the state-space model are generated for the topology in such a way, that the created system is structurally controllable and structurally observable. The determination of the input and output matrices can be achieved by the path finding and signal sharing methods [15], which modify the result of the maximum matching algorithm.

The *system characterisation module* performs the calculation of 49 numerical measures to qualify the dynamical system based on its structure. The implemented measures, on the one hand, are well-known static measures (e.g. the number of nodes and edges, closeness and betweenness centralities), and, on the other hand, measures that characterise the dynamics of the system (e.g. structural controllability, observability, control centrality and relative degree). This module can also be used for the purpose of simple network analysis.

The *improvement and robustness module* integrates two main functions. On the one hand, it enables the input and output configurations of the system to be extended in such a way that the relative degree of the modified system does not exceed the initially defined threshold. For

this purpose, this module implements five methods, namely the set covering-based grassroot and retrofit methods [18], the centrality measures-based method [18], the modified Clustering Large Applications based on Simulated Annealing algorithm (mCLASA), and the Geodesic Distance-based Fuzzy c-Medoid Clustering with Simulated Annealing algorithm (GDFCMSA) [18, 17]. On the other hand, this module allows users to examine the robustness of the extended configurations by removing nodes from the network representation and by checking the structural controllability and structural observability of the damaged system.

In the root directory of the toolbox, the *toolboxExample.m* file contains a simple example that shows how the toolbox works.

1.2 Dynamical system design

State-space models are widely used to represent linear time-invariant 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 $\mathbf{A} \in \Re^{N\times N}$, $\mathbf{B} \in \Re^{N\times M}$, $\mathbf{C} \in \Re^{K\times N}$, $\mathbf{D} \in \Re^{K\times M}$.

1.2.1 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 structurally 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 [13, 22, 30, 10], 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 structurally controllable, if the Kalman's criterion satisfied [13], 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 [10]. 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 structurally observable, if the Kalmans criterion satisfied [13], i.e. the observability matrix has maximum structural rank (Equation (1.6)).

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

1.2.2 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 [19] 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 [19, 21].

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 $\mathbf{B}$ and $\mathbf{C}$ or for vectors $\mathbf{u}$ and $\mathbf{y}$. 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.

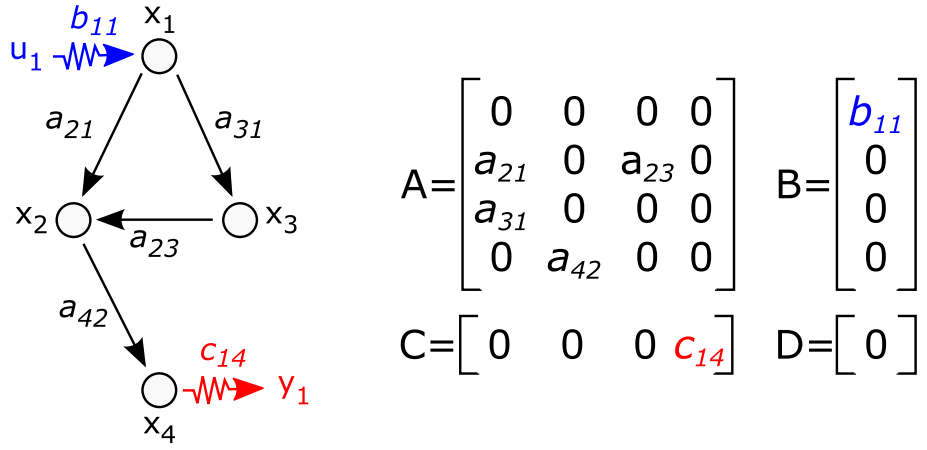


Figure 1.2: Network representation of the dynamical system described by the matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$.

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.

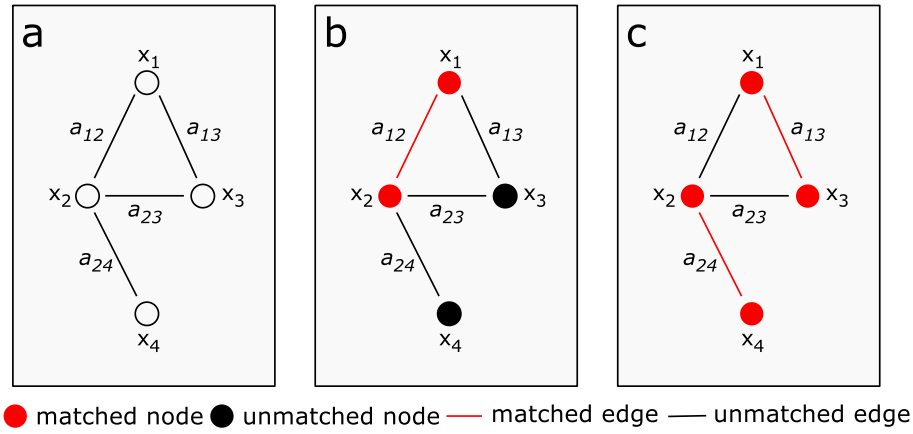


Figure 1.3: (a) An undirected graph. (b) Maximal matching. (c) Maximum matching.

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.

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 [5] by Pothen et al. [27]. To generate the block triangular form of the sparse, unsymmetric matrix, the algorithm firstly calculates the maximum matching. Pothen and Fan changed [27] Duff's maximum matching algorithm [4] 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 [11] with $\mathcal{O}(E\sqrt{V})$ and this version was used by Liu et al. in [19], but its performance is case dependent and Pothen's version was faster on denser problem[27].

According to [19], if we determine the unmatched nodes of the network determined by $\mathbf{A}$,

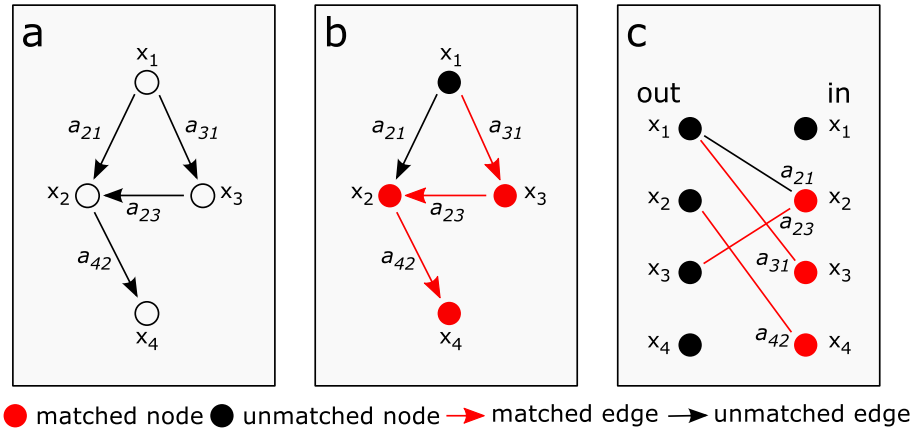


Figure 1.4: (a) A directed graph. (b) Maximum matching. (c) Undirected, bipartite graph based representation.

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.

1.2.3 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 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 [21], 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.

Signal sharing method

Liu et al. claim that after we applied the maximum matching algorithm, we have to check strongly connected components (SCCs) [21]. 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 1.5(a), a simple example can be seen.

In Figure 1.5(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 1.5(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

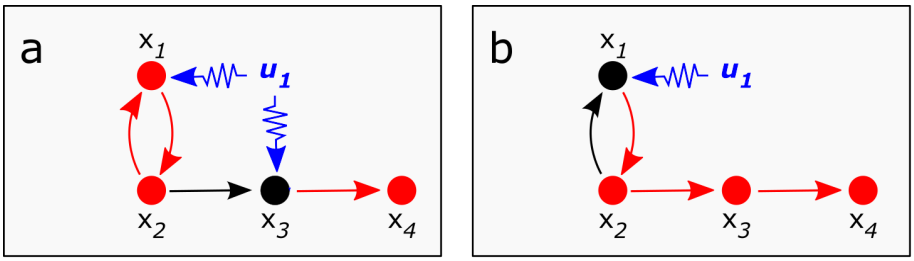


Figure 1.5: Solutions provided by signal sharing (a) and path finding (b) to handle strongly connected components.

is changed to 1. If we generate this **B** matrix and check the Kalman's rank criteria [13] as it was introduced in Equation (1.4), then we get that, the system is controllable.

Path finding method

In Figure 1.5(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 1.5(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 1.5(a) we can choose between x_1 and x_2 according the signal sharing, but in Figure 1.5(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.

1.3 Qualifying the system

The toolbox can qualify the system by several measures. These measures contain centrality measures that qualify the system with one value, node centrality measures that assign a value for each state variable, node clustering measures that create groups from the nodes based on their structural or dynamical properties, and edge centrality measures that assign a value for each edge. Some of these measures demand for both the topology and the input and output configurations, but most of them require only the topology, i.e. these measures can be used for network analysis also. The implemented measures are the following:

1.3.1 Network/system centralities, properties

- **number of nodes**: the number of nodes in a network, or the number of state variables 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 [8]**: 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 [25] of degree [24]**: 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.

1.3.2 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 [29]**: 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 [7]**: 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 [20]**: 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.

1.3.3 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 [12], 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 (1.7).

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1.7}$$

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 (1.7). 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 (1.8).

$$weight(set_j, set_k) = 1 - \frac{\sum_{i \in set_j \cap set_k} |d_{ij} - d_{ik}|}{|set_j \cap set_k| * max(set_j, set_k)} \tag{1.8}$$

In Equation (1.8) 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.

1.3.4 Edge centrality measures

- **edge betweenness [7]**: 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 (1.9).

$$\begin{aligned} endpoint\_similarity(e) =& Jaccard(e_{SC}, e_{EC}) * \\ & * Jaccard(e_{SO}, e_{EO}) \end{aligned} \tag{1.9}$$

In Equation (1.9) 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 (1.10).

$$\begin{aligned} similarity(e, f) =& Jaccard(e_{SC}, f_{SC}) * Jaccard(e_{EO}, f_{EO}) * \\ & * Jaccard(e_{EC}, f_{EC}) * Jaccard(e_{SO}, f_{SO}) \end{aligned} \tag{1.10}$$

In Equation (1.10) 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.

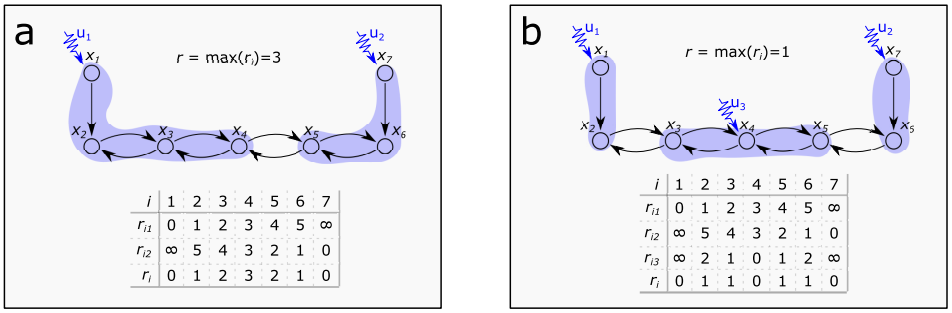
1.4 Improvement of input and output configurations

The modified maximum matching methods results input and output configurations with minimum number of inputs and outputs. As a result, a few driver and sensor nodes are enough to make the system controllable and observable, but the relative degree of the system became very high [17]. This module provide five methodology to extend the set of driver and sensor nodes in order to decrease realtive degree. The set covering-based methods (grassroot and retrofit) and the centrality measures-based methods [16] utilise simple structural heuristic, while the modified Clustering Large Applications based on Simulated Annealing algorithm (mCLASA) and Geodesic Distance-based Fuzzy c-Medoid Clustering with Simulated Annealing algorithm (GDFCMSA) utilise simulated annealing to deal with NP-hardness [17].

1.4.1 Relative degree of dynamical systems

We can define relative degree of the system as the "physical closeness" or "direct effect". For nonlinear systems the relative degree is introduced in several paper [3, 18]. For linear systems that are defined by Eq (1.1) and Eq. (1.2), relative degree, r_{ij} of input j and output i is defined as $c_i \mathbf{A}^{r_{ij}-1} b_j \neq 0$, where c_i is row i of $\mathbf{C}$ and b_j is column j of $\mathbf{B}$. By utilizing the network science-based representation, the relative degree r_{ij} can be defined as the shortest path ℓ_{kl} from the driver node k of u_j to sensor node l of output y_i, $r_{ij} = \ell_{kl}$. The relative degree of sensor i is defined as $r_i = \min_j r_{ij}$, while the relative degree of the system is $r = \max_i r_i$. The visual representation of the relative degree can be seen in Figure 1.6.

As can be seen in Figure 1.6, the shaded area represents clusters of state variables that can be achieved by a relative degree that is smaller than r_i. The optimal placement of additional inputs and outputs is NP-hard, thus, heuristic methods were applied to determine additional driver and sensor nodes [16, 17].



i	1	2	3	4	5	6	7
r_{i1}	0	1	2	3	4	5	∞
r_{i2}	∞	5	4	3	2	1	0
r_i	0	1	2	3	2	1	0

i	1	2	3	4	5	6	7
r_{i1}	0	1	2	3	4	5	∞
r_{i2}	∞	5	4	3	2	1	0
r_{i3}	∞	2	1	0	1	2	∞
r_i	0	1	1	0	1	1	0

Figure 1.6: Visual representation of relative degree in the network representation of dynamical systems. (a) The relative degree is three if two driver nodes are placed instead of (b) three driver nodes that result relative degree of one.

1.4.2 Determine additional driver and sensor nodes

In this toolbox five heuristic algorithm were implemented to determine the location of additional inputs and outputs. The first, set covering-based method generates for each state variable the set of state variables that can be controlled or observed in the predefined relative degree. Then a set covering problem is solved on the generated sets. This is the grassroot method. The retrofit method works similarly, but used the driver or sensor nodes generated by path finding or signal sharing as fixed selected set in the set covering problem, thus controllability and observability can be granted. The third method generates the normalised betweenness centrality, Bc, and closeness centrality, Cc, measures for each state variable, then expands the set of drivers or sensors with state variable i, based on the heuristic value: $\max(Cc(i) * Bc(i))$. This step repeat until the relative degree of the system is higher than the predefined one. The fourth and fifth ones are very similar. They utilise simulated annealing. While mCLASA places the additional driver and sensor nodes as the centre of a k-medoid in each iteration, GDFCMSA use geodesic distance based fuzzy c-medoid algorithm to determine the location of new inputs and outputs. The detailed description of the algorithms can be found in [16, 17, 6, 14, 2].

In order to evaluate the result of each appraoch, the toolbox generates the cost of each methods that is calculated as it can be seen in Eq (1.11).

$$cost(\beta) = \beta \max_{i=1}^{K} r_i + (1-\beta)\frac{\sum_{i=1}^{K} r_i}{N} \tag{1.11}$$

1.4.3 Robustness analysis

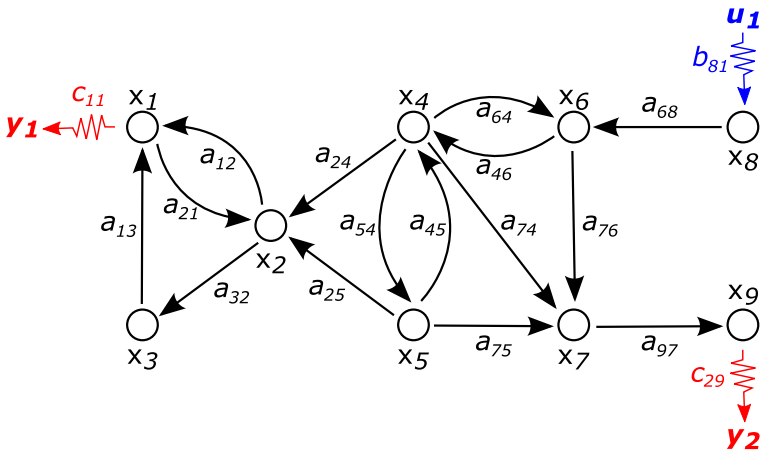
The toolbox provides a simple robustness analysis. The method analyses the structural controllability and observability of the system, if one state variable is removed, i.e. the number of test cases equals with the number of state variable. If a driver/sensor node is removed, then the input/output is removed also. The method counts that how many times the system remain controllable/observable and returns the critical nodes, i.e. those nodes if one them is removed, then controllability/observability propoerty violated.

Copyright of octave-networks-toolbox

Chapter 2

Network mapping

The task of the first module is to create a system based on the adjacency matrix. As we presented in Section 1.2.3, this could be done by path finding and signal sharing methods. An example of the application of the path finding method for the creation of a state-space model from the adjacency matrix (**A**) is shown in Figure 2.1. In this figure, **B** denotes the resulting input matrix, **C** the output matrix and **D** the feedthrough description.



$$A = \begin{bmatrix} 0 & a_{12} & a_{13} & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & a_{24} & a_{25} & 0 & 0 & 0 & 0 \\ 0 & a_{32} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{45} & a_{46} & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{54} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{64} & 0 & 0 & 0 & a_{68} & 0 \\ 0 & 0 & 0 & a_{74} & a_{75} & a_{76} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{97} & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ b_{81} \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} c_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_{29} \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure 2.1: An example network with determined input (blue) and output (red) configurations according to the path finding method.

components

Purpose

Calculates the strongly connected components in a directed network.

Synopsis

`[`**ci**, **sizes**`]=components(`**adj**`)`

Description

The **components** function determines the strongly connected components in a directed graph. The input **adj** is a sparse matrix that is the adjacency matrix of the graph. The output **ci** is the vector that contains the component indices of the nodes, while **sizes** are the vector that contains the sizes of the components.

Algorithm

The **components** function uses Tarjan's method to find components [31]. The algorithm uses the octave-networks-toolbox's **tarjan** function.

See Also

`exist, fileparts, which, addpath, adj2adjL, tarjan, adjL2edgeL, sortrows, hist, unique`

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 [5, 27] 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 [27, 5] to calculate maximum matching and Tarjan's algorithm [31] 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.

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 [27, 5] to calculate maximum matching and Tarjan's algorithm [31] 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.

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 characterisation

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 with their nodes and edges. Most of the functions require only an adjacency matrix while some of them claim other inputs: **B** and/or **C** matrices of the state-space model. The module contains a function, which generates a structure with all the measures calculated by this module based on the matrices of the state-space model.

As the configuration above is not complex enough to demonstrate the functions of the second module, a more complex configuration of the input and output nodes is used. The sample input and output configurations can be seen in Figure 3.1, where the input and the output nodes are denoted by blue and red, respectively.

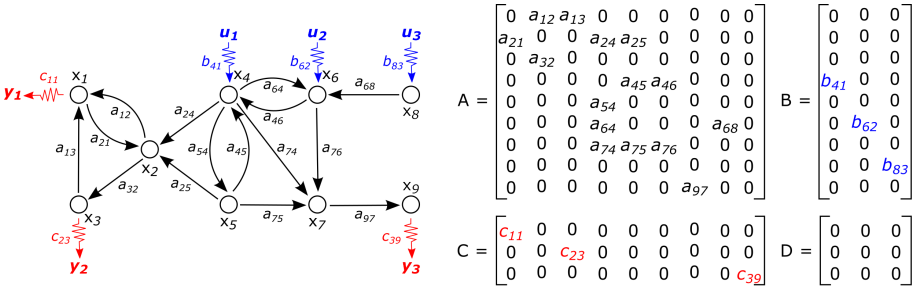
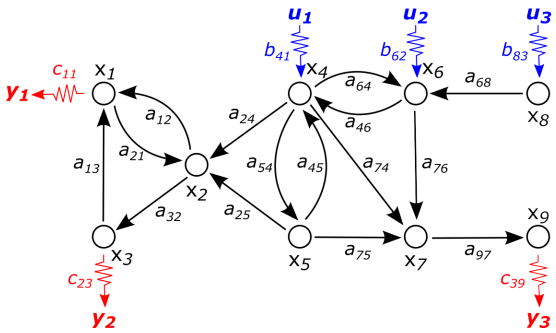


Figure 3.1: The topology and associated configuration of the system for demonstration of the System characterisation module of the NOCAD toolbox.

The system presented in Figure 3.2 consists of 9 state variables, and 15 directed connections between them. The density shows that the number of edges is almost a fifth of the possible maximum, and the diameter of the system (i.e. the longest shortest path in the network that presents its structure) is 4. The degree variance is 2.67, while the Freeman's centrality is 0.43. The relative degree of the system is also 4. The Pearson coefficient shows that the in-in and in-out correlations are assortative in nature, while out-out and out-in correlations are likely to be disassortative. The system is controllable and observable. As no loop is present in the network, the percentage of loops relative to edges is 0%. As there are 6 edges that have symmetric edge pairs and the number of connections is 15, the percentage of the symmetric edge pairs relative to the edges is 40%.

Node centrality measures assigned to the state variables of the system are also presented in Figure 3.4. One of the most important values is the highest degree of the nodes, which belongs to state variable x_4. As Scott's centrality is a normalised degree, the most important node is once again x_4. The closeness of node x_i is calculated as the ratio of the number of nodes reachable from x_i to the sum of their distances from x_i. The higher value indicates the more central position of



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: The example network with system centrality measures.

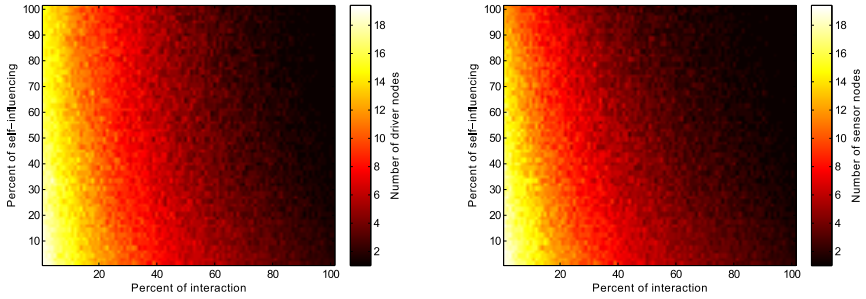
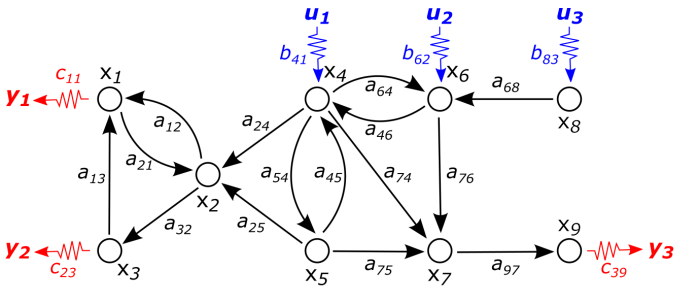


Figure 3.3: Heatmaps generated by the implemented function.

the node, and, once again, node x_4 is the most central element. The betweenness centrality shows how many shortest paths intercept the given node. If a node has a high value, then it is a critical node in the structure. The highest value belongs to nodes x_2 and x_4. The PageRank assigns a percentage value for each node, based on their centrality roles if Markov-chains are modelled. The measure referred to as correlation shows the proportion of the number of edges of neighbours and the number of neighbours. This information is useful when determining the assortativity of the system. The control centrality and observe centrality measures determine how many state variables can be influenced or observed by the nodes.

In Figure 3.5, the first vectors (referred to as driver and sensor nodes) show the driver and sensor nodes as logical vectors. The following four vectors classify these nodes as source, external, internal and inaccessible driver and sensor nodes. These types of nodes are introduced in [28] in detail. In the next section of the figure, the controlling and observing matrices are presented. Generally, these matrices are sparse matrices, as only the columns of drivers and sensors contain nonzero values. In Figure 3.5, we converted them into row vectors for their appropriate visualisation. The values show the number of derivations necessary to influence or observe a state variable in the system. Next, the similarity of the driver and sensor nodes is presented. The similarity of driver nodes x_4 and x_6 is 0.81. In this case, the reason why it is less than 1 is that although they control the same set of nodes, the numbers of derivations that influence them are different. In terms of sensor similarity, sensor nodes x_2 and x_3 observe the same set of nodes and they do this almost simultaneously, so their similarity is 0.91.

In Figure 3.6, $R_{\mathcal{C}}$ and $R_{\mathcal{O}}$ are the simple reachability matrices, respectively. They show which nodes can be controlled or observed by a given node. In $R_{\mathcal{C}}$, the i^{th} column shows which nodes can control node i. From the other viewpoint, elements in row i highlight those nodes which can be controlled by node i. In this example, node x_8 can influence every node, but it does not

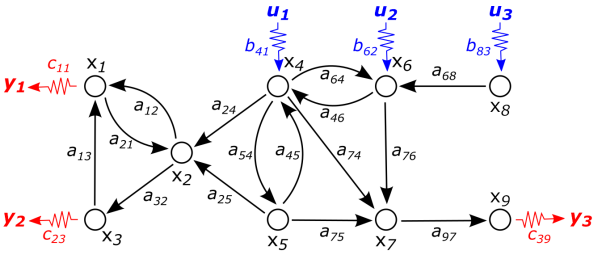


node ID:	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
in degree:	2	3	1	2	1	2	3	0	1
out degree:	1	2	1	4	3	2	1	1	0
degree:	3	5	2	6	4	4	4	1	1
Scott's centrality:	0.19	0.31	0.13	0.38	0.25	0.25	0.25	0.06	0.06
closeness:	4.1	5.6	3.90	7.15	5.95	5.75	4.2	2.91	2.5
betweenness:	1	9	0	9	0	7	4	0	0
PageRank:	25.42	26.46	13.74	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: The example network with node centrality measures.

guarantee structural controllability. The $R_{\mathcal{O}}$ matrix can be interpreted analogously with regard to observability.

Finally, measures of edge centrality are seen in Figure 3.7. The betweenness has the same meaning as in the case of nodes, that is, it yields the number of shortest paths that intercept the edge [7]. From this perspective, the most critical edge is the edge a_{46} with a value of 10. The endpoint similarity shows how similar the influenced and observed sets of the state variables with regard to the endpoints of edges are. This metric has a high value if the edge is part of a cycle or creates a bridge in the network. As no bridges are present in this network, only cycles can be recognised by this measure. The edge similarity shows how similar the roles of edges are, and it allows redundancies, to be located. In the topology presented, nodes x_1, x_2 and x_3, or nodes x_4, x_5, x_6 and x_7 also create parts of the network that possess redundancy.



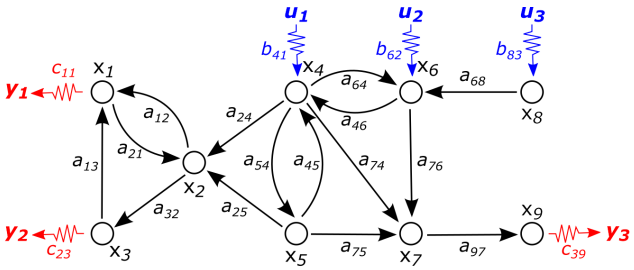
node ID	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
driver nodes:	0	0	0	1	0	1	0	1	0
driver source:	0	0	0	0	0	0	0	1	0
driver external	0	0	0	0	0	0	0	0	0
driver internal	0	0	0	0	0	0	0	0	0
driver inaccessible:	0	0	0	1	0	1	0	0	0
controlling (ID): x_4	3	2	3	1	2	2	2	0	3
x_6	4	3	4	2	3	1	2	0	3
x_8	5	4	5	3	4	2	3	1	4
others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

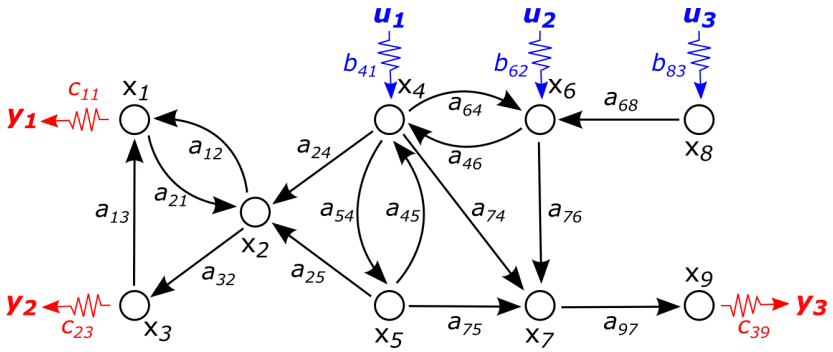
Figure 3.5: First part of measures of node clustering and the representation of the topology of the network.



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 measures of node clustering and the representation of the topology of the network.



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 for the given topology.

numOfNodes

Purpose

Determine the number of the nodes in the given network.

Synopsis

[numNodes]=numOfNodes(**adj**)

Description

The **numOfNodes** function determines the number of the nodes in the network. The function returns with **numNodes**, 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

numOfEdges

Purpose

Determine the number of the edges in the given network.

Synopsis

`[numEdges]=numOfEdges(adj)`

Description

The **numOfEdges** function determines the number of the edges in the network. The function returns with **numEdges**, 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

`numOfNodes, numOfEdges`

allShortestPaths

Purpose

Calculates the length of shortest paths between each node-pair in the given network.

Synopsis

`[`**shortestPaths**`]=allShortestPaths(`**adj**`)`

Description

The **allShortestPaths** function determines the shortest paths in the network described by adjacency matrix **adj**. The adjacency matrix should be directed and could be both weighted and unweighted. The function returns with **shortestPaths** matrix that contains the length of paths. In row i and column j, the length of geodesic path from node i to node j can be found.

Algorithm

The shortest path calculated by the dijkstra algorithm. For calculating the shortest paths, the toolbox uses the octave-networks-toolbox's **simpleDijkstra** function.

See Also

`length, cat, simpleDijkstra`

diam

Purpose

Determine the diameter of the given network.

Synopsis

[diameter]=diam(**adj**)

Description

The **diam** function determines the diameter of the network. The function returns with **diameter**, 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.

See Also

allShortestPaths, 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

`numOfNodes, 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 [8], 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

`numOfNodes, 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 [13] 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

`numOfNodes, 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 [13] 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

`numOfNodes, 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, allShortestPaths, logical, min, max`

pearsonDir

Purpose

The **pearsonDir** function calculates the Pearson correlation [25] 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, 26]. 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, 26]. 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`

getNumNodes

Purpose

The **getNumNodes** 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]=getNumNodes(A)`

Description

The **getNumNodes** 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 **getNumNodes** 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, getNumNodes, 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

`numOfNodes, 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

`numOfNodes, 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** [29]. 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 [29] 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

`numOfNodes, 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:

$$in\_closeness(i) = \frac{|reach(i)|^2}{(N-1) * \sum_{j \neq i} d_{ji}}.$$

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

`allShortestPaths, sum, isnan`

nBetweenness

Purpose

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

Synopsis

`[betweenness]=nBetweenness(adj)`

Description

The **nBetweenness** 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 **nBetweenness** 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 octave-networks-toolbox's **nodeBetweennessFaster** function.

See Also

`exist, fileparts, which, addpath, nodeBetweennessFaster, nchoosek, size`

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

`numOfNodes, 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 [20] 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 [13, 22] 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 [27, 5] to calculate the rank of the controllability matrix.

See Also

`numOfNodes, 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 [27, 5] to calculate the rank of the observability matrix.

See Also

`numOfNodes, 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 [28].

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 [28]. **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, numOfNodes, 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 [28]. **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, numOfNodes, 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. The number of necessary derives equal with the shortest path plus one because one derivation needs to influence the first state.

See Also

`sum, allShortestPaths`

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. The number of necessary derives equal with the shortest path plus one because one derivation needs to observe the first state.

See Also

`sum, allShortestPaths`

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 [12]. 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

`numOfNodes, 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 [12]. 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

`numOfNodes, 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 [12].

Algorithm

The **jaccard** function determine the similarity of two sets according to Jaccard publication [12]. 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`

betweenness

Purpose

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

Synopsis

`[betw]=betweenness(adj)`

Description

The **betweenness** 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 **betweenness** 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 octave-networks-toolbox's **edgeBetweenness** function.

See Also

`exist, fileparts, which, addpath, length, edgeBetweenness, 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, numOfNodes, numOfEdges, 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, numOfEdges, 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" [23]. 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.

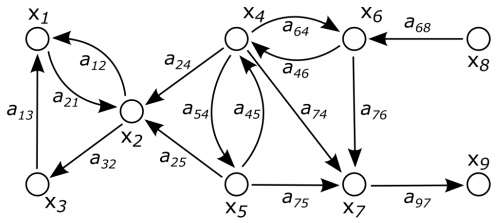
See Also

`struct, find, length, cell, num2str, isControllable, isObservable, numOfNodes, numOfEdges, density, diam, degreeFreeman, degreeVariance, degreeRel, percentLoopSym, degreeIn, degreeOut, degree, degreeScott, closenessCentrality, nBetweenness, pageRank, degreeCorrel, controlCentrality, observeCentrality, driverNodes, driverType, sensorNodes, sensorType, clusterControl, clusterObserve, driverSimilarity, sensorSimilarity, reachC, reachO, betweenness, similarity, endpointSim, pearsonDir, getConfig`

Chapter 4

Improvement and robustness

For the demonstration of the last module, configurations provided by the first module are used again (Figure 2.1). Results provided by this module can be seen in Figure 4.1. In this case, five methods were applied to the system to extend the configuration as follows: the required relative degree was set at 2, while the alpha parameter of the cost function was set at 0.5 [17]. Results show that all the methods determine the same set of driver nodes for the system, that is, they are sufficient to influence state variables x_4 and x_8. The resultant cost is 1.5556, the relative degree is 2 which satisfies the requirements, and the mean of the relative degrees is 1.1111. In this configuration, six different nodes can be identified which can be damaged separately and the system remains controllable. This is expressed by the value of robustness (66.6%). The most important nodes in terms of controllability are x_2, x_4 and x_7. In the case of observability, methods yield different solutions with the exception of the centrality measures-based and mCLASA algorithms which provide the best configuration in this case. Although the cost as well as the maximum and mean of the relative degree were identical in the case of retrofit set covering-based and GDFCMSA methods as well, the robustness analysis of these configurations exhibits a higher degree of vulnerability.



	Set Covering grassroot	Set Covering retrofit	Centrality Measures	mCLASA	GDFCMSA
driver nodes:	[4, 8]	[4, 8]	[4, 8]	[4, 8]	[4, 8]
size:	2	2	2	2	2
cost:	1.5556	1.5556	1.5556	1.5556	1.5556
max:	2	2	2	2	2
mean:	1.1111	1.1111	1.1111	1.1111	1.1111
robustness:	6	6	6	6	6
robustness (%):	0.6667	0.6667	0.6667	0.6667	0.6667
critical nodes:	[2, 4, 7]	[2, 4, 7]	[2, 4, 7]	[2, 4, 7]	[2, 4, 7]
sensor nodes:	[2, 7, 9]	[1, 4, 9]	[1, 7, 9]	[1, 7, 9]	[1, 6, 9]
size:	3	3	3	3	3
cost:	1.4444	1.3889	1.3889	1.3889	1.3889
max:	2	2	2	2	2
mean:	0.8889	0.7778	0.7778	0.7778	0.7778
robustness:	6	6	7	7	6
robustness (%):	0.6667	0.6667	0.7778	0.7778	0.6667
critical nodes:	[1, 2, 6]	[1, 6, 9]	[1, 6]	[1, 6]	[1, 6, 9]

Figure 4.1: Improvement and robustness analysis of the system.

analyseComponents

Purpose

The **analyseComponents** function analyses components and decrease their degree to ensure the required maxEnergy.

Synopsis

`[newNodes]=analyseComponents(`**A**, **actualNodes**, **maxEnergy**`)`

Description

The **analyseComponents** function groups the state variable to the closest one of the driver (or sensor) nodes contained by **actualNodes**. If the maximum of the shortest path is higher in a group then the required one defined by **maxEnergy**, then new driver (or sensor) nodes is selected. If **actualNodes** contains the driver nodes,

then input matrix **A** is the adjacency matrix of the topology, while if **actualNodes** consists from sensor nodes, then input matrix **A** is the state-transition matrix of the system. The result, **newNodes**, contains both the original and the additional driver (or sensor) nodes.

Algorithm

The **analyseComponents** function selects new driver (or sensor) nodes as follow. The closeness, $Cc(i)$, and betweenness, $Bc(i)$, centrality measures generated for all the state varaibles, then the new driver (or sensor) node is the node with highest betweenness and closeness centrality ($\max_i Cc(i) * Bc(i)$).

See Also

`exist`, `fileparts`, `addpath`, `genpath`, `min`, `unique`, `find`, `max`, `measures`, `find`, `union`

evolveOperability

Purpose

The **evolveOperability** function utilises the centrality measures-based and set covering method based algorithms to improve the system through the reduction of relative degree.

Synopsis

[**actCovSet**, **actCovRetSet**, **senCovSet**, **senCovRetSet**, **actNetMesRetSet**, **senNetMesRetSet**]=evolveOperability(**A**, **maxRelDeg**)

Description

The **evolveOperability** function generates driver and sensor node sets for a system described by its adjacency matrix **A** (not state-transition matrix) such that the resulted system have relative degree smaller then **maxRelDeg**. The output variables are the set of driver and sensor nodes generated by three methods, respectively.

Algorithm

The **evolveOperability** function implies the centrality measures-based and the set covering method-based algorithms, and returns with their results. For more information please see the related functions **analyseComponents**, **setCover** and **setCoverExcept**.

See Also

exist, fileparts, addpath, genpath, issparse, full, setCover, generateMatricesPF, find, sum, analyseComponents, setCoverExcept

extendData

Purpose

The **extendData** function appends a new element to the structure data. The new field contains the result of the improvement of the system.

Synopsis

`[data]=extendData(data, targetDegree, alphaPar)`

Description

The **extendData** function takes the structure **data** generated by **matricesToStruct**, and extend it by the result of the improvement of the system. The function grant that the relative degree does not exceed **targetDegree**. The alpha parameter of the mCLASA and GDFCMSA methods can be set up with **alphaPar**. The resulted **data** structure then has the **improved** field that contains the target degree and for each algorithm the resulted set of driver and sensor nodes and the related cardinality of the sets, the value of cost function, the maximum and the mean of the relative degrees.

Algorithm

The **extendData** function call the functions that determine the set of additional driver and sensor nodes for the dynamical system.

See Also

`exist`, `fileparts`, `addpath`, `genpath`, `issparse`, `full`, `allShortestPaths`, `getCost`, `length`, `mCLASArMax`, `GDFCMSArMax`, `struct`

GDFCMSAaddMax

Purpose

The **GDFCMSAaddMax** function determines the additional driver (or sensor) nodes for a system such that the resulted relative degree is minimal. The number of additional nodes is given.

Synopsis

`[res]=GDFCMSAaddMax(`**A**, **addMax**, **Niter**, **Nf**`)`

Description

The **GDFCMSAaddMax** function applies the GDFCMSA algorithm on the system described by **A**. After generating the minimum necessary set of driver (or sensor) nodes, the method assigns **addMax** additional driver (or sensor) nodes to the system to minimise the relative degree. The simulated annealing method has **Niter** iteration to find the best solution, and it is run **Nf** times independently. If the input matrix **A** is the adjacency matrix, then the result set of nodes, **res**, is the driver nodes, while if **A** is the state-transition matrix, then **res** contains the sensor nodes.

Algorithm

The **GDFCMSAaddMax** function implements the GDFCMSA method introduced in [17].

See Also

`exist, fileparts, which, addpath, length, allShortestPaths, min, getSensors, getCost, find, sum, repmat, cat, randperm, setdiff, size, cumsum`

GDFCMSArMax

Purpose

The **GDFCMSArMax** function determines the additional driver (or sensor) nodes for a system such that the required relative degree is satisfied with minimal additional nodes. The number of required relative degree is given.

Synopsis

`[res]=GDFCMSArMax(`**A**, **rMax**, **Niter**, **Nf**`)`

Description

The **GDFCMSArMax** function applies the GDFCMSA algorithm on the system described by **A**. After generating the minimum necessary set of driver (or sensor) nodes, the method assigns additional driver (or sensor) nodes to the system until the required relative degree, **rMax**, is ensured. The simulated annealing method has **Niter** iteration to find the best solution, and it is run **Nf** times independently. If the input matrix **A** is the adjacency matrix, then the result set of nodes, **res**, is the driver nodes, while if **A** is the state-transition matrix, then **res** contains the sensor nodes.

Algorithm

The **GDFCMSArMax** function implements the GDFCMSA method introduced in [17].

See Also

`exist, fileparts, which, addpath, length, allShortestPaths, sparse, min, getSensors, getCost, find, sum, repmat, cat, randperm, setdiff, size, cumsum`

getCost

Purpose

The **getCost** function calculates the value of cost function, de maximum and mean of the relative degree for a given system with input (or output) configuration.

Synopsis

`[`**cost**, **maxDeg**, **meanDeg**`]=getCost(`**nodes**, **alpha**, **dists**`)`

Description

The **getCost** function demand for the driver (or sensor) nodes, **nodes**, tha alpha parameter of the cost function, **alpha**, [17], and the distance matrix, **dists**. Based on the distance matrix for each state variable the closest driver (or sensor) node is determined. Based on these distance, relative degree for each driver (or sensor) node is calculated. Then the cost function is calculated, **cost**, and returned with the maximum, **maxDeg**, and the mean, **meanDeg**, of relative degrees.

Algorithm

The **getCost** function calculates the cost as it is presented in [17]. In mathematical form: $\min_S cost(G, S, \beta) = \beta \max_{i=1}^{K} r_i + (1-\beta)\frac{\sum_{i=1}^{K} r_i}{N}$, where S is the set of driver (or sensor) nodes, G notes the topology of the system, and r_i stands for the relative degree of driver (or sensor) node i.

See Also

`min, max, mean`

getSensors

Purpose

The **getSensors** function generates the sensor nodes for the state-transition matrix, or the driver nodes for the adjacency matrix.

Synopsis

`[nodes]=getSensors(A)`

Description

The **getSensors** function contains a part of the **maximumMatchingPF** function, but after generating the driver (or sensor) nodes, it returns with the nodes only instead of the result of matching.

Algorithm

The **getSensors** function generates the driver (or sensor) nodes as it done by **maximumMatchingPF**.

See Also

`exist, fileparts, addpath, dmperm, issparse, components, sparse, hist, max, find, diag, union, unique, setdiff, sum, intersect, isempty`

greedyscp

Purpose

The **greedyscp** function solve greedy set covering problem.

Synopsis

[**solution$_s$etsCell**, **solution$_s$etsLabelsV**]=greedyscp(**setsCell**, **setsLabelsV**)

Description

The **greedyscp** function solve greedy set covering problem. The inputs are the sets, **setsCell** and their labels, .**setsLabelsV**. The results are the selected sets, **solution$_s$etsCell**, and their labels, **solution$_s$etsLabelsV**. The function modified as follows: The disp and display lines are commented except warnings and errors.

Algorithm

The **greedyscp** function implements the method expressed in: F. Gori, G. Folino, M.S.M. Jetten, E. Marchiori, "MTR: Taxonomic annotation of short metagenomic reads using clustering at multiple taxonomic ranks", Bioinformatics 2010., doi = 10.1093/bioinformatics/btq649, [9].

See Also

-

isCont

Purpose

The **isCont** function determines if a topology is controllable with the given set of driver nodes.

Synopsis

`[isContr]=isCont(`**adj**, **drivers**`)`

Description

The **isCont** function create the input matrix based on the driver nodes, **drivers**, then check if the system is controllable with the input configuration. If yes, then **isContr** is true, and false otherwise.

Algorithm

The **isCont** function use the Kalman's rank criterion to determine controllability [13].

See Also

`exist, fileparts, addpath, genpath, zeros, length, isControllable`

isObs

Purpose

The **isObs** function determines if a topology is observable with the given set of sensor nodes.

Synopsis

`[isObserv]=isObs(adj, sensors)`

Description

The **isObs** function create the output matrix based on the sensor nodes, **sensors**, then check if the system is observable with the output configuration. If yes, then **isObserv** is true, and false otherwise.

Algorithm

The **isObs** function use the Kalman's rank criterion to determine observability [13].

See Also

`exist, fileparts, addpath, genpath, zeros, length, isControllable`

mCLASAaddMax

Purpose

The **mCLASAaddMax** function determines the additional driver (or sensor) nodes for a system such that the resulted relative degree is minimal. The number of additional nodes is given.

Synopsis

`[res]=mCLASAaddMax(`**A**, **addMax**, **Niter**, **Nf**`)`

Description

The **mCLASAaddMax** function applies the mCLASA algorithm on the system described by **A**. After generating the minimum necessary set of driver (or sensor) nodes, the method assigns **addMax** additional driver (or sensor) nodes to the system to minimise the relative degree. The simulated annealing method has **Niter** iteration to find the best solution, and it is run **Nf** times independently. If the input matrix **A** is the adjacency matrix, then the result set of nodes, **res**, is the driver nodes, while if **A** is the state-transition matrix, then **res** contains the sensor nodes.

Algorithm

The **mCLASAaddMax** function implements the mCLASA method introduced in [17].

See Also

`exist, fileparts, which, addpath, length, allShortestPaths, min, getSensors, getCost, find, sum, repmat, cat, randperm, setdiff`

mCLASArMax

Purpose

The **mCLASArMax** function determines the additional driver (or sensor) nodes for a system such that the required relative degree is satisfied with minimal additional nodes. The number of required relative degree is given.

Synopsis

`[res]=mCLASArMax(`**A**, **rMax**, **Niter**, **Nf**`)`

Description

The **mCLASArMax** function applies the mCLASA algorithm on the system described by **A**. After generating the minimum necessary set of driver (or sensor) nodes, the method assigns additional driver (or sensor) nodes to the system until the required relative degree, **rMax**, is ensured. The simulated annealing method has **Niter** iteration to find the best solution, and it is run **Nf** times independently. If the input matrix **A** is the adjacency matrix, then the result set of nodes, **res**, is the driver nodes, while if **A** is the state-transition matrix, then **res** contains the sensor nodes.

Algorithm

The **mCLASArMax** function implements the mCLASA method introduced in [17].

See Also

`exist, fileparts, which, addpath, length, allShortestPaths, min, getSensors, getCost, find, sum, repmat, cat, randperm, setdiff`

measures

Purpose

The **measures** function calculates the normalised closeness and betweenness centrality measures for each node.

Synopsis

`[`**betw**, **outCloseness**`]=measures(`**A**`)`

Description

The **measures** function generates the closeness centrality measures and the betweenness centrality measures, then returns them. The input matrix is the adjacency matrix. The transposed input changes the value of closeness but does not change the value of betweenness centrality.

Algorithm

The **measures** function utilises the **closenessCentrality** function to calculate the closeness, and it uses **nBetweenness** and **allShortestPaths** to calculate betweenness centrality.

See Also

`exist, fileparts, addpath, genpath, closenessCentrality, nBetweenness, allShortestPaths, sum`

robust

Purpose

The **robust** function examines the robustness of the improved systems created by the five methods.

Synopsis

`[data]=robust(data)`

Description

The **robust** function analyse the robustness of the improved systems. The results are appended to the structure **data**.

Algorithm

The **robust** function iterate over each state variable in the system, and in each iteration the state variable is clear from the network. Then the controllability (and observability) is analysed. The function summarised that how many times the system remain controllable (and observable), and a rate is also calculated in per cent. The critical nodes, i.e. the nodes that cannot be removed to grant controllability (and observability).

See Also

`length, zeros, cell, isCont, cat, isequal, union, intersect, isObs, struct`

setCover

Purpose

The **setCover** function improves the system by the set cover problem for a system and a required relative degree. This is the grassroot set covering method.

Synopsis

`[`**nodes**, **sets**`]=setCover(`**adj**, **maxEnergy**`)`

Description

The **setCover** function determine the driver nodes for the input adjacency matrix, **adj**, or the sensor nodes, if the input **adj** is the state-transition matrix. The required relative degree is **maxEnergy**. The function returns the selected driver (or sensor) nodes, **nodes**, and the set of state variables belongs to these drivers (or sensors), **sets**. With this methods the structural controllability (or observability) is not granted.

Algorithm

The **setCover** function generates the sets of reachable state variables for each driver (or sensor) within the required relative degree. Then the set covering problem is solved. The method utilises the greedy set covering [9].

See Also

`greedyscp, logical, length`

setCoverExcept

Purpose

The **setCoverExcept** function improves the system by the set cover problem for a system, an initially defined driver (or sensor) node set and a required relative degree. This is the retrofit set covering method.

Synopsis

`[`**nodes**, **sets**`]=setCoverExcept(`**adj**, **maxEnergy**, **selected**`)`

Description

The **setCoverExcept** function expands the driver nodes for the input adjacency matrix, **adj** and driver node set, **selected**, or the sensor nodes, if the input **adj** is the state-transition matrix and **selected** contains sensor nodes. The required relative degree is **maxEnergy**. The function returns the selected driver (or sensor) nodes, **nodes**, and the set of state variables belongs to these drivers (or sensors), **sets**.

Algorithm

The **setCoverExcept** function generate the sets of reachable state variables for each driver (or sensor) within the required relative degree. Then the set covering problem is solved, where the nodes of **selected** fixed in the result. The method utilises the greedy set covering [9].

See Also

`find, sum, greedyscp, logical, union`

Bibliography

[1] Albert-László Barabási et al. *Network science*. Cambridge university press, 2016.

[2] Shu-Chuan Chu, John F Roddick, and Jeng-Shyang Pan. *A comparative study and extension to K-medoids algorithms*. Contemporary Development Company, 2001.

[3] Prodromos Daoutidis and Costas Kravaris. Structural evaluation of control configurations for multivariable nonlinear processes. *Chemical engineering science*, 47(5):1091–1107, 1992.

[4] Iain S Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software (TOMS)*, 7(3):315–330, 1981.

[5] Andrew L Dulmage and Nathan S Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[6] Balazs Feil and Janos Abonyi. Geodesic distance based fuzzy clustering. In *Soft Computing in Industrial Applications*, pages 50–59. Springer, 2007.

[7] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[8] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[9] Fabio Gori, Gianluigi Folino, Mike SM Jetten, and Elena Marchiori. Mtr: taxonomic annotation of short metagenomic reads using clustering at multiple taxonomic ranks. *Bioinformatics*, 27(2):196–203, 2010.

[10] Katalin M Hangos and Ian T Cameron. *Process modelling and model analysis*. Academic Press,, 2001.

[11] John E Hopcroft and Richard M Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[12] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull Soc Vaudoise Sci Nat*, 37:241–272, 1901.

[13] Rudolf Emil Kalman. Mathematical description of linear dynamical systems. *Journal of the Society for Industrial and Applied Mathematics, Series A: Control*, 1(2):152–192, 1963.

[14] András Király, Ágnes Vathy-Fogarassy, and János Abonyi. Geodesic distance based fuzzy c-medoid clustering–searching for central points in graphs and high dimensional data. *Fuzzy Sets and Systems*, 286:157–172, 2016.

[15] Dániel Leitold, Ágnes Vathy-Fogarassy, and János Abonyi. Controllability and observability in complex networks–the effect of connection types. *Scientific reports*, 7(1):151, 2017.

[16] Daniel Leitold, Agnes Vathy-Fogarassy, and Janos Abonyi. Design-oriented structural controllability and observability analysis of heat exchanger networks. *Chemical Engineering Transactions*, 70:595–600, 2018.

[17] Daniel Leitold, Agnes Vathy-Fogarassy, and Janos Abonyi. Network distance-based simulated annealing and fuzzy clustering for sensor placement ensuring observability and minimal relative degree. *Sensors*, 18(9):3096, 2018.

[18] Daniel Leitold, Agnes Vathy-Fogarassy, and Janos Abonyi. Evaluation of the complexity, controllability and observability of heat exchanger networks based on structural analysis of network representations. *Energies*, 12(3):513, 2019.

[19] Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. Controllability of complex networks. *nature*, 473(7346):167, 2011.

[20] Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. Control centrality and hierarchical structure in complex networks. *Plos one*, 7(9):e44459, 2012.

[21] Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. Observability of complex systems. *Proceedings of the National Academy of Sciences*, 110(7):2460–2465, 2013.

[22] David G Luenberger. *Introduction to Dynamic Systems, Theory, Models, and Applications*. John Wiley & Sons, New York, 1979.

[23] Lev Muchnik. Complex networks package for matlab (version 1.6), 2013.

[24] Mark EJ Newman. Assortative mixing in networks. *Physical review letters*, 89(20):208701, 2002.

[25] K Pearson. VII. note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, jan 1895.

[26] Márton Pósfai, Yang-Yu Liu, Jean-Jacques Slotine, and Albert-László Barabási. Effect of correlations on network controllability. *Scientific reports*, 3:1067, 2013.

[27] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software (TOMS)*, 16(4):303–324, 1990.

[28] Justin Ruths and Derek Ruths. Control profiles of complex networks. *Science*, 343(6177):1373–1376, 2014.

[29] John Scott. Social network analysis. *Sociology*, 22(1):109–127, 1988.

[30] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*. Pearson, 1991.

[31] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.