SPANNING CACTI FOR STRUCTURALLY CONTROLLABLE NETWORKS

NGO THI TU ANH

NATIONAL UNIVERSITY OF SINGAPORE

2012

SPANNING CACTI FOR STRUCTURALLY CONTROLLABLE NETWORKS

NGO THI TU ANH

(M.Sc., SFU, Russia)

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS

NATIONAL UNIVERSITY OF SINGAPORE

2012

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Ngo Thi Tu Anh
13 August 2012

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Professor Zhang Louxin, who gave me the opportunity to work on this interesting research problem, paid patient guidance to me and gave me much invaluable help and constructive suggestion on it. I would also like to thank him for the financial support during my M.Sc. candidature. It has been my pleasure being his research student.

I would next, like to thank Lou Chang, PhD candidate at Department of Mathematics, NUS, for his insightful discussions, valuable suggestions and great help.

My sincere thanks go to all team members in our bioinformatics group and I have benefited a lot from them.

Finally, I would like to thank my family. Without their continuous encouragement and support, nothing would have been possible for me.

Ngo Thi Tu Anh

August, 2012

Table of Contents

Summary

Finding the minimal spanning cacti for structurally controllable networks is an interesting problem. In this research work, we designed and implemented an algorithm for the problem. The algorithm is based on the Minimum Input Theorem and Lin’s theorem. The Minimum Input Theorem, proved by Liu et al., establishes that the minimum number of inputs needed to control a directed network equals to the number of unmatched nodes from any maximum matching in the network. Lin's theorem states that, a controllable network is spanned by cacti.

Specifically, in order to control the network of a system, we first need to identify the set of nodes that, if driven by different signals, can offer full control over the network. Those nodes are called “driver nodes”. We use the maximum matching algorithm to find the minimum set of driver nodes. After that, the controlled network is built by introducing one input node for each driver node. Moreover, with the help of the maximum matching, we can identify the basic cactus components such as stems and cycles in this controllable network. Finally, the spanning cacti is built by properly connecting those components.

We also address the question how to add extra links into a directed network to make it controllable with only one input. The minimum number of extra links needed to construct an one-input controllable network can also be determined by the maximum matching in the original network.

In this thesis, we discuss the controllability and structural controllability of directed networks. We introduce the problem of minimum spanning cacti for structurally controllable networks, as well as propose an algorithm to find it. We also conduct experiments applying our algorithm on several computer generated and real networks. The algorithm is an analytical tool for studying the network controllability of real systems.

List of Tables

List of Figures

Chapter 1

Introduction to Graphs and Networks

Graphs are mathematical structures that are used to model the pair-wise relations between objects from a certain collection. The study of graphs started in the 18th century, and graph theory is a prime area in discrete mathematics. Graph theory becomes a very useful technique for solving real-world problems. One part of graph theory is the network theory, which studies the networks of real systems. Networks are applied in numerous disciplines. Its applications span from internet to biological systems.

This chapter gives a basic introduction to graphs and networks, the matching problem, as well as cactus graphs.

1.1 Basic Definitions

To formalize our discussion on graph theory, we shall introduce some basic concepts. We start with simple graphs, i.e., undirected, unweighted graphs without multiple edges.

Definition 1.1 *A **graph** or **undirected graph** G = (V, E) is an ordered pair of finite set V and finite set E, where elements of V are called vertices or nodes, and elements of $E \subseteq V \times V$ are called edges or arcs. We call V the vertex set of G, with E being the edge set. The cardinality of V is called the order of G, and the cardinality of E is called the size of G.*

One can label a graph by attaching labels to its vertices. If $(v_1, v_2) \in E$ is an edge of a graph G = (V;E), we say that v_1 and v_2 are adjacent vertices. The edge (v_1, v_2) is also said to be incident with the vertices v_1 and v_2.

Definition 1.2 *A directed edge is an edge such that one vertex incident with it is designated as the head vertex and the other incident vertex is designated as the tail vertex. A directed edge (u, v) is said to be directed from its tail u to its head v. A **directed graph** or digraph G is a graph such that each edge is directed. The in-degree of a vertex $v \in V(G)$ counts the number of edges such that v is the head of these edges. The out-degree of a vertex $v \in V(G)$ is the number of edges such that v is the tail of edges.*

It is important to distinguish a graph G as being undirected or directed. If G is undirected and $(u, v) \in E(G)$ then (u, v) and (v, u) represent the same edge. In case G is a digraph, then (u, v) and (v, u) are different directed edges. Another important type of undirected graph, which will be discussed in this chapter, is bipartite graph.

Definition 1.3 *A graph G is called **bipartite**, if $V(G)$ is partitioned into two disjoint and nonempty subsets A and B such that each edge $(u, v) \in G$ connects a vertex of set A and a vertex of set B. In this case (A, B) is a bipartition of G and G is (A, B)-bipartite.*

Note that, the definitions above concern simple graphs, i.e. multiple edges and self-loops (edges incident to same vertices) are not allowed. In a simple graph, the edges form a set, rather than a multi set, and each edge is a pair of distinct vertices. A general graph can have self-loops, as well as multiple edges.

A ***path*** in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence, possibly excluding the terminate vertex. A path may be infinite even if the graph is finite. But a finite path always has a start vertex and an end vertex. When vertices in a path are not repeated, the path is called ***simple path***. If the starting vertex and the ending vertex of a path are the same, then it is a ***closed path*** or ***cycle***. A cycle whose vertices are all distinct is called ***simple cycle***. For directed graphs, simple paths and simple cycles are called elementary paths and elementary cycles, respectively, and defined as follows.

Definition 1.4 *For a general directed graph, an* ***elementary path*** *is a series of oriented edges* $\{(v_1 \to v_2), (v_2 \to v_3), \dots, (v_{k-1} \to v_k)\}$*, where all vertices* $v_1, v_2, \dots, v_k$ *are distinct. When* v_k *coincides with* v_1*, it is called* ***elementary cycle****.*

Two vertices in a graph are said to be connected if there is a path between them. A graph is said to be **connected** if any two of its vertices are connected. A graph is called a ***weighted graph*** if each of its edges is assigned a number (or weight). Such weights may represent, for example, strengths, lengths, costs or capacities of the edges. In literature, weighted graphs are often referred to as networks. The model of weighted graphs has many applications, in which the core problem is to optimize the total weight of some graph properties. For example, the famous traveling salesman problem can be modeled as a problem of finding a closed path in a weighted graph of cities (weights are the distances between pairs of cities) which contains all vertices while achieving the minimum total weight.

1.2 Cactus Graphs

A well-known graph structure is the cactus graph. It is sometimes called cactus, as well. For undirected graphs, the definition of cactus is the following.

Definition 1.5 *An undirected connected simple graph is called an* ***undirected cactus*** *if any two simple cycles have at most one vertex in common. Equivalently, every edge of such graph belongs to at most one cycle.*

Example of a cactus graph is given in Figure 1.1.

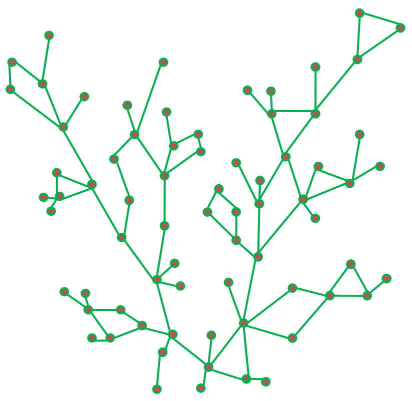


Figure 1.1 Example of an undirected cactus graph

Proposition 1.1 [22] An undirected cactus with n vertices can have at most $\frac{n-1}{2}$ cycles.

Proof Suppose the number of cycles is k. Let n_i is the number of vertices in the i^{th} cycle. Since two cycles have at most one vertex in common, then there are at most $k-1$ vertices that counted 2 times. Thus, we have: $\sum_i^k n_i \leq n+k-1$. But $n_i \geq 3$ because each undirected cycle in simple graph (no multiple edges are allowed) must have at least 3 vertices. Hence, $\sum_i^k n_i \geq 3k$. So, $3k \leq n+k-1$, or $k \leq \frac{n-1}{2}$ ∎

Proposition 1.2 [22] An undirected cactus with n vertices can have at most $\frac{3}{2}(n-1)$ edges.

Proof Since a cycle of m vertices contains m edges, while a path of m vertices contains only $m-1$ edges. Hence, the number of edges in the cactus with n vertices is maximum when the cactus contains a maximum number of cycles. By previous proposition, the maximum number of cycles is $\frac{n-1}{2}$; and this can happen when all cycles are triangles. Thus, the maximum number of edges for a cactus with n vertices is $\frac{3}{2}(n-1)$ ■

Directed cactus graphs have been less studied than undirected cactus graphs. In fact, the definitions of directed cactus graphs vary in literature. For example, in [21], a directed cactus graph is defined as a strongly connected, directed graph in which each edge is contained in at most (and thus, exactly) one directed cycle. This definition looks similar to the undirected cactus definition, except that edges are directed, and the graph is strongly connected, i.e. there is a directed path between any two vertices. Such directed cactus contains only cycles. However, in [2], Lin defined a directed cactus as a strict combination of paths (stems) and cycles (buds). In our work, we consider only directed graphs, and we follow the definition given by Lin. An example of a general directed cactus is shown in Figure 1.2.

Definition 1.6 *A **stem** is an elementary path. The initial (or terminal) vertex of a stem is called the root (or top) of the stem.*

Definition 1.7 *A **bud** is an elementary cycle C plus an additional edge e that ends, but not begins, in a vertex of the cycle. This additional edge e is called the distinguished edge of the bud.*

Definition 1.8 *A **general directed cactus** is a digraph defined recursively as follows. A stem is a cactus. Given a stem S_0 and buds $B_1, B_2, \dots B_l$, then $S_0 \cup B_1 \cup B_2 \cup \dots \cup B_l$ is a cactus if for every i ($1 \leq i \leq l$) the initial vertex of the distinguished edge of B_i is not the top of S_0 and is the only vertex belonging at the same time to B_i and $S_0 \cup B_1 \cup B_2 \cup \dots \cup B_{i-1}$.*

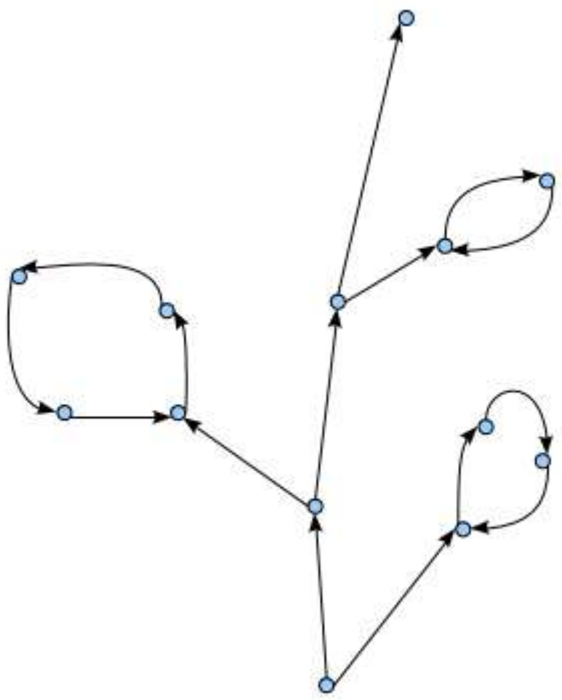


Figure 1.2 Example of a directed cactus graph

Proposition 1.3 The number of edges in a general directed cactus with n vertices is:

$$N_{edges} = n - 1 + N_{buds}$$

where N_{buds} is the number of buds in the cactus.

Proof The above formula can be derived by mathematical induction for N_{buds} as follows:

Let P(N_{buds}) is the statement "$N_{edges} = n - 1 + N_{buds}$ "

- Basic step: N_{buds}= 0, the cactus contains no buds, and then it contains only one stem. Hence, the number of edges is $n - 1$. Therefore, the statement holds for $N_{buds} = 0$.

  For $N_{buds} = 1$, the cactus contains one bud. Suppose the bud contains k nodes, then the stem contains *n*-*k* nodes (with *n*-*k*-*1* edges). The bud has k edges in the cycle and one distinguished edge. Hence, the total number of edges is: *(n*-*k*-*1)*+*k*+*1* = *n*-*1* + N_{buds}. Therefore, the statement holds for $N_{buds} = 1$.

- Inductive step:

  Show that if P(k) holds, then also P(k+1) holds. This can be done as follows. $N_{buds} = k$, the total number of edges is $N_{edges} = n - 1 + k$. We will show that it's correct for $N_{buds} = k + 1$. Suppose the $(k + 1)^{th}$ bud contains p nodes. The cactus without the $(k + 1)^{th}$ bud contains *(n-p)* nodes and *(n-p-1+k)* edges. The $(k + 1)^{th}$ bud introduces p edges in the cycle and one distinguished edge. Hence, the total number of edges of the cactus is:

  $N_{edges} = n - p - 1 + k + p + 1 = n - 1 + (k + 1) = n - 1 + N_{buds}.$■

1.3 Maximum Matching

Matching in graph theory refers to a set of edges that do not share common vertices. Matching has attracted the interest of researchers for more than 20 years. The notion of matching is one of the key points to understand the Minimum Input Theorem, which is presented in the next chapter. This section gives an introduction on matching in graphs, as well as algorithms to find a maximum matching.

Definition 1.9 *For an undirected graph, a* ***matching*** *M is an independent edge set, i.e. a set of edges without common vertices. A vertex is* ***matched*** *if it is incident to an edge in the matching. Otherwise, the vertex is* ***unmatched****. Similarly, an edge of G is* ***matched*** *if it is in M, otherwise, it is* ***unmatched****.*

For a bipartite graph *G(V,E),* and we often denote it as *G (A, B, E),* where *(A, B)* is a partition of *V*. The definition of matching for a bipartite graph is similar to that of an undirected graph.

The generalized definition for **directed** graphs follows.

Definition 1.10 *For a digraph G = (V,E), a* ***matching*** *M is a subset of E that no two edges in M share a common starting vertex or a common ending vertex. A vertex is* ***matched*** *if it is the ending vertex of an edge in the matching M. Otherwise, it is* ***unmatched****. Similarly, an edge of G is* ***matched*** *if it is in M, otherwise, it is* ***unmatched****.*

For all types of graphs (undirected, bipartite or directed), a matching that cannot be extended is called ***maximal***, i.e. upon adding to M any edge not in M, it is no longer a matching. A matching that contains the largest possible number of edges is called ***maximum matching.*** A ***perfect matching*** is the matching that all vertices are matched. A perfect matching is obviously a maximum matching.

We are interested in the maximum matching problem, i.e., given a graph G, find a matching M of maximum size. The bipartite case is simpler in structure; and will be first considered as it helps us to illustrate the basic ideas involved in solving the maximum matching problem. In this section, we present and analyze algorithms for this problem when the underlying graph is bipartite.

The general approach for the matching problem can be considered as an instance of the concept of augmentation. Let's start with concepts of ***alternating paths*** and ***augmenting paths***.

Consider a bipartite graph $G = (V, E)$ and a matching M in it. Let p be a path consists of edges $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ where the vertices are distinct. The path p can simply be represented as $p = [v_1, v_2, \dots, v_k]$.

Definition 1.11 *An* ***alternating path*** *is a path in which the edges belong alternatively to the matching and not to the matching.*

Definition 1.12 *A path $p = [v_1, v_2, \dots, v_k]$ is called* ***augmenting*** *if it is an alternating path and both v_1 and v_k are unmatched.*

Obviously, an augmenting path has an even number of vertices; its starting and ending vertices are unmatched. An example of a bipartite graph and a matching is presented in Figure 1.3.

Let P be the set of edges on an augmenting path $p = [v_1, v_2, \dots, v_k]$ in G with respect to the matching M. The operation $M \oplus P$ denotes the symmetric difference of M and P, i.e. $M \oplus P = (M \backslash P) \cup (P \backslash M)$. The most interesting property of an augmenting path P with respect to a matching M is that if we set $M' = M \oplus P$, then we get a new matching M' and moreover, the size of M' is one unit larger than the size of M. That is, we can form a larger matching M' from M by taking the edges of P not in M and adding them to M' while removing from M the edges that are also in the path p.

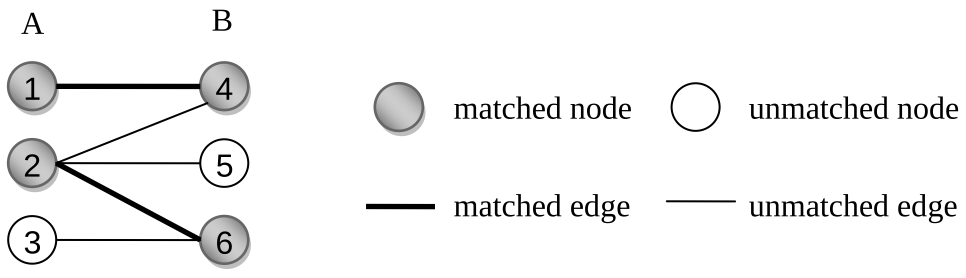


Figure 1.3 Example of a bipartite graph and a matching in it. The path [1,4,2,6,3] is an alternating path. The path [3,6,2,5] is an augmenting path. If we "augment" this path, we get a maximum matching: {(1,4),(2,5),(3,6)}.

The key of most algorithms for finding maximum matching in a graph is based on the following theorem, stated by Berge [6].

Theorem 1.1 A matching M in a graph G is a maximum matching if and only if there is no augmenting path in G with respect to M.

Proof

(⇒) Let P be the set of edges on an augmenting path p with respect to M. Set $M' = M \oplus P$. Then M' is a matching with cardinality greater than M. This contradicts the maximality of M.

(⇐) If M is not maximum, let M^* be a maximum matching, i.e. $|M^*| > |M|$ (note that M^* is a matching with largest size and that M^* may or may not contain M). Let Q = $M \oplus M^*$. Then we have the following.

Q has more edges from M^* than from M (since $|M^*| > |M|$ implies that $|M^* - M| > |M - M^*|$). Each vertex is incident to at most one edge in $M \cap Q$ and at most one edge in $M^* \cap Q$. Thus Q is composed of cycles and paths that alternate between edges from M and M^*.

Therefore, there exists a path with more edges from M^* than from M (all cycles will be of even length and have the same number of edges from $M^*$and M). This path is an augmenting path with respect to M. Hence, there exists an augmenting path with respect to M, which is also a contradiction. ■

In fact, all known matching algorithms for general graphs are based on the idea of finding augmenting paths. Start with any matching M, say the empty matching. Locate an augmenting path p with respect to M, then augment M along p, and replace M by the resulting matching. This procedure is repeated until no more augmenting path exists. By the above theorem, we are guaranteed a maximum matching.

The main challenge for maximum matching is how to find an augmenting path with respect to the current matching M of G. In the case when the graph is bipartite, finding these augmenting paths can be done efficiently, for example, by breadth-first search (BFS). Firstly, a search for augmenting paths must start by constructing alternating paths from the unmatched nodes. Since an augmenting path must have one endpoint in node set A and the other in node set B, without loss of generality, we start the search only from unmatched nodes in set A. For example, choose an unmatched node a_i from A, with BFS we may search for alternating paths by considering all vertices adjacent to a_i. Since a_i is an unmatched node then all its incident edges are unmatched edges. The second step is a straightforward step, from the adjacent nodes of a_i, we look for the matched edges. If there is no matched edges, then all these adjacent nodes are unmatched, hence an augmenting path has been found. Otherwise, we add these matched edges into our alternating path, and we repeat the first step by searching from our new added nodes.

Algorithm for finding a maximum matching in bipartite graphs

Given a current matching M in a bipartite graph $G(A,B,E)$ (initially M is empty), we can speed up the BFS idea above by searching for the next nodes from the set A only, ignoring the nodes in set B. The new idea is the following. We start searching for an augmenting path from an unmatched node a_i in A. Since in the augmenting path, the node adjacent to a_i must be a matched node b_j in the set B; $b_j$must have a "mate" in A by current matching M (we say node x is a mate of node y if the edge (x,y) is matched in the current matching). Let's denote the mate of b_j as a_k. Then, instead of continuing the search from b_j, we look for a_k and continue the search directly from a_k, ignoring the node b_j. To do this efficiently, we construct an auxiliary graph $G_1 = (A, E_1)$, where $(a_i, a_j) \in E_1$ if and only if $a_i$is adjacent to the mate of a_j with respect to current matching M. Now the BFS procedure is performed directly on the auxiliary graph G_1.

Using the above idea, Papadimitriou gives an algorithm for finding a maximum matching in bipartite graphs [7], which is presented in Figure 1.4.

This algorithm terminates when there is no path with unmatched node from the set A. By the construction of the auxiliary graph, this means that there is no augmenting path with respect to current matching. Hence, the current matching is maximum.

For time-complexity of the algorithm, we see that the maximum matching contains at most $min(|A|,|B|)$ edges. Since each augmentation increases the number of matched edges by 1, then the number of stages the algorithm loops is at most $min(|A|,|B|)$ times. Building the auxiliary graph takes $O(|E|)$ time; finding augmenting path in A by breadth-first search takes at most $O(|E|)$ time. Hence, for each stage, finding augmenting path requires $O(|E|)$ time. The augmentation procedure requires at most $O(min(|A|,|B|)$ time. Hence, overall, the algorithm takes $O(min(|A|,|B|)\cdot|E|)$ time.

Algorithm 1.1 BIPARTITE MAXIMUM MATCHING ALGORITHM

Input: A bipartite graph *G = (A,B,E)*
Output: The maximum matching of *G*, represented by the array *mate*.

```
begin
    for all $v \in A \cup B$ mate[v] = 0;        // initialize
 stage: begin
    for all $v \in A$ do unmatched[v] = 0;
    // construct auxiliary graph (A,$E_1$)
    $A := \emptyset$
    for all $[v, u] \in E$ do
        if mate[u] = 0 then unmatched[v] := u
        else
        if $mate[u] \neq v$ then $E_1 := E_1 \cup (v, mate[u])$;
    //initialize the queue Q for breath-first search:
    Q := $\emptyset$
    for all $v \in A$ do
        if mate[v] = 0 then begin
              label[v] := 0;
              Q := Q $\cup$ {v}
        end;
    // start BFS for augmenting path
    while $Q \neq \emptyset$ do begin
        let v be a node in Q;
        remove v from Q;
        if unmatched[v] $\neq$0 then begin // found an augmenting path
              augment(v);
              goto stage;  // restart from beginning
        end
        else
        for all unlabeled v' such that (v,v') $\in E_1$ do begin
              label[v'] := v;
              Q := Q $\cup$ {v'}
        end
    end;// of while
end; // of stage
end;// of algorithm.
//======================//
procedure augment(v)
begin
    if label[v] = 0 then begin
        mate[v] := unmatched[v];
        mate[unmatched[v]] :=v;
    end
    else begin
        unmatched[label[v]] := mate[v];
        mate[v] := unmatched[v];
        mate[unmatched[v]] :=v;
        augment(label[v]);
    end;
end;// of procedure augment()
```

Figure 1.4 Algorithm for maximum matching problem in bipartite graphs

1.4 *Key Types of Networks*

Networks are nothing else but graphs. In general, the notion of graphs often refers to a theoretical model, while networks are often used to represent practical models arising from natural and technological processes. The terms network and graph can be used interchangeably in most contexts. However, in literature networks are often directed and weighted graphs.

The study of networks – network theory, is a part of graph theory which emerges as an important area in computer science. It has numerous applications in statistical physics, computer science, economics, biology, sociology, and operation research. Many types of networks have been intensively studied, such as, flow networks, semantic networks, technological networks, biological networks, social networks, and random networks.

Recently, a branch of network theory that studies complex networks has emerged. ***Complex networks*** are networks with non-trivial topological features, i.e. the features that do not appear in simple graphs such as lattices or random graphs, but often occur in real networks. For example, the structural feature of power-law degree distribution characterizes the scale-free networks, while short path lengths and high clustering are specific features of small-world networks. Some key types of networks such as technological, social and biological networks also display substantial non-trivial features, such as heavy tail degree distribution, high clustering coefficient among vertices, community structure and hierarchical structure. Those networks are examples of complex networks.

Technological networks

Technological networks are networks designed for manipulating technological processes. One of the main purposes of these networks is to distribute information and resources for human activities. Some examples of technological networks are computer networks, power grid networks, high way networks, airport networks. One of the most complex networks is the internet, which is the global network of computers, interconnected by communication channels that allow sharing of resources and information. It serves both as information and telecommunication network. The internet network is the scale-free and has small-world properties.

One type of technological networks that is analyzed in our work is the network of sequential logic electronic circuits [9]. In these circuits, the nodes represent the logic gates and the flip-flops. A directed link from node *u* to node *v* is established if the value at the gate (or the flip-flop) *v* depends on the value at *u*. For example, for the circuit *s402*, the network has 252 nodes and 399 directed edges [see Table 4.3 and Table 4.4].

Biological networks

Biological networks are networks arising from biological systems. Informally, a biological network is a network of biological entities, in which these entities interact or link to each other as a whole system. For examples, species are linked into a food web, or proteins interact to each other to form a protein interaction network (PIN).

Key types of biological networks include PINs, food webs, metabolic networks, gene regulatory networks, etc. In a PIN, nodes are all the proteins of a cell, and edges are the interactions between them. A PIN is an undirected network. It shows the scale-free and small-world properties. Another type of biological networks is the network of metabolic reactions. Chemical compounds inside a living cell are connected by enzymatic reactions that transform one compound into another during the metabolic processes of the cell. These reactions are catalyzed by enzymes, which are also biochemical compounds. Hence, all these compounds form a network of biochemical reactions, which is called a metabolic network. Metabolic networks are also shown to be complex networks with scale-free and small-world features [16].

Furthermore, at a higher level in biology, organisms also form a network, since they depend on each other: they eat or are eaten by others. The network of those bait and prey links is called a food web. In our work, several food webs are analyzed (see Table 4.3 and Table 4.4), including food webs in Little Rock Lake [15], in Grassland [13] and in St. Marks Seagrass [14].

Social networks

A social network is a graph model of social entities and their relationships. Entities in sociology are often called individual actors, which may be persons, groups, organizations, web sites, scholarly publications, etc. The relationships between these entities may be friendship, sexual relationship, email sending–receiving relationship, web links, etc. Social network is a very useful tool for analyzing the structure of social entities as a whole. The study of such structural features often uses social network analysis to find out the common local and global patterns, identify the social entities that are the most influential, examine the network dynamics, analyze the networks' behavior, and evaluate the networks' controllability.

Several social networks are analyzed in our work, including WWW network of political web-blogs [10], trust networks of college students and prison inmates [17], and intra-organizational networks. For example, the network of email sending–receiving relationship between employees in a consulting company in [11] contains 46 nodes and 879 links. The nodes represent persons, while a link from person X to person Y is established if X received emails from Y. In the point of view of network's controllability (which is the main subject of our work), this network in [11] is easily controllable. It can be fully controlled by controlling only 2 nodes (which are called driver nodes) of the network (see Table 4.4 and Figure 4.4(a)).

Chapter 2

Controllability and Structural Controllability

Controllability is one of the fundamental concepts in modern mathematical control theory. This is a qualitative property of control systems and is of particular importance in control theory. Systematic study of controllability started at the beginning of 1960s and the theory of controllability is based on the mathematical description of the dynamical system.

This section includes most of the basic principles about controllability and structural controllability.

2.1 Controllability

Controllability describes the ability to move the system around in its entire configuration space using only certain admissible manipulations. Specifically, a dynamic system is controllable if it can be “driven” from any initial state to any desired final state in finite time with a suitable choice of inputs.

In real life system, most natural and technological systems are organized into networks of components, and the whole systems are governed by some underlying dynamical processes. For example, metabolism is the basic process of any living cell. In this process, cells exchange energy. They grow and build their structures and respond to the environment. Inside a living cell, metabolites are transformed by a series of reactions that are catalyzed by enzymes. These reactions form a scale-free metabolic network. A cell may “control” its metabolism by controlling the concentration of its enzymes.

The controllability of a real system is affected by two independent factors:

1) The system's architecture, represented by the weighted and directed network describing the connections of system's components with each other.
2) The dynamical rules that describe the time-dependent interactions between the components.

Consider the linear dynamic system:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t) \qquad (1)$$

where $x(t) = (x_1(t), \dots, x_N(t))^T$ presents the state of system of N nodes at time t.

$u(t) = (u_1(t), \dots, u_M(t))^T$ is the input vector.

A is the $N \times N$ matrix which describes the system’s wiring and the interaction strength between components (for example: matrix A represents the traffic on individual communication links, or the strength of regulatory interactions in a regulatory network).

B is the $N \times M$ input matrix describing the nodes that are controlled by the outside controllers.

In order to determine the controllability of system (1), R.E Kalman gave an algebraic criterion, which depends only on matrices *A* and *B*, called Kalman Rank Condition [1].

Kalman Rank Condition:

A necessary and sufficient condition for system (1) to be controllable is

$$rank(C) = rank[B, AB, A^2B, \dots, A^{N-1}B] = N$$

C is called Kalman’s controllable matrix of size $N \times NM$.

Example 2.1: Consider a network in Figure 2.1:

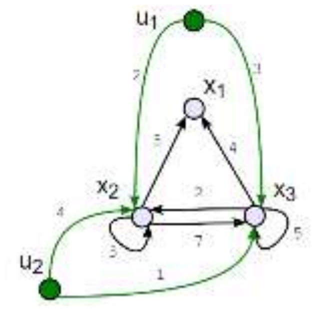


Figure 2.1 An example of a controllable network

The linear dynamics shown above can be written as follows:

$$\dot{x} = \begin{bmatrix} 0 & 3 & 4 \\ 0 & 3 & 2 \\ 0 & 7 & 5 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 2 & 4 \\ 3 & 1 \end{bmatrix} u$$

The controllability matrix for this third-order system is given by:

$$C = [B, AB, A^2B]$$

$$= \begin{bmatrix} 0 & 0 & 18 & 16 & \\ 2 & 4, 12 & 14, & A^2B \\ 3 & 1 & 29 & 33 & \end{bmatrix}$$

Since the first three columns are linearly independent, we conclude that $rank(C) = 3$. Hence there is no need to compute A^2B since it is well known from linear algebra that the row rank of the given matrix is equal to its column rank. Thus, $rank(C) = 3 = N$ implies that the system under consideration is controllable.

2.2 *Structural Controllability*

To test the controllability of a network of arbitrary system using Kalman's rank condition, all the values of matrices A and B are required. However, for most real complex networks, the exact values of all entries of A are often unknown. In addition, checking the rank of C is computationally difficult, especially for large networks. In order to address those difficulties, Lin introduced the concept of structural controllability [2] which will be discussed below.

Now the controlled system (1) is also denoted by a pair (A, B).

Definition 2.1 *Two pairs (A, B) and (A', B'), of the same dimensions, have the* ***same structure*** *if all non-zero entries of one pair correspond to those of the other pair, and vice versa.*

Definition 2.2 *The pair (A, B) of a linear system is called* ***structurally controllable*** *if there exists a controllable pair (A_0, B_0) which has the same structure as (A, B).*

An observation of Lee and Markus [3] is that, the set of all controllable pairs for (1) is open and dense in the space of all pairs (A, B) with standard matrix metric. Hence, if the initial pair (A_0, B_0) is not controllable, then for every $\varepsilon > 0$, there is a controllable pair (A_1, B_1) such that $\|A_1 - A_0\| < \varepsilon$ and $\|B_1 - B_0\| < \varepsilon$.

The concept of structural controllability also has practical meaning, since in reality the non-zero parameters of the system's model may be noisy, yet the model still has the same structure with the underlying system. Furthermore, if a system is structurally controllable then it is controllable (in the usual sense with Kalman's rank condition $rank(C) = N$) for almost all parameter values, except for some pathological cases with zero measure [2]. Those cases happen when some certain accidental constraints are satisfied. In other words, structurally controllability implies controllability in practice.

Thus, the concept of structural controllability helps us to overcome the imperfect measurement of the exact values of the system parameters. Without precise knowledge of those parameters, the controllability of the system can still be evaluated.

Definition 2.3 *A system is said to be* ***strongly structurally controllable*** *if its controllability is independent of the system parameters, as long as they are non-zero.*

Examples of strongly structurally controllable systems are shown in Fig. 2.3 A and C.

The graph of a pair (A, B)

Let *G(A)* denote the directed network of system (1) in which nodes represent the system states, edges are determined by the state matrix *A*. The adjacency matrix of *G(A)* is described by the following way: if the entry $a_{ij} = 0$ then there is no link from node *j* to node *i* in *G(A)*; if $a_{ij} \neq 0$ there is a directed link from node *j* to node *i*. If all non-zero a_{ij} equal to 1, then the matrix A represents the transposition of the adjacency matrix of *G(A)*. By mathematical notation, we have $G(A) = (V_A, E_A)$ where $V_A = \{x_1, \dots, x_N\}$ is the set of state vertices; $E_A = \{(x_j, x_i) | a_{ij} \neq 0\}$ is the set of edges.

The **controlled network**, denoted by the pair *(A,B)*, can be represented by a directed graph $G(A, B) = (V, E)$, where $V = V_A \cup V_B$ is the vertex set; $E = E_A \cup E_B$ is the edge set; $V_B = \{u_1, \dots, u_M\}$ is the set of **input vertices**; $E_B = \{(u_j, x_i) | b_{ij} \neq 0\}$ is the set of **controlled edges**, connecting input vertices to state vertices. The M input vertices are also called the **origins** of the digraph *G(A,B)*. The state vertices that connect to the origins are called **controlled nodes**. Note that, one input can be connected to several controlled nodes.

Example 2.2: Given a pair (A, B) such that:

$A = \begin{bmatrix} 0 & a_{12} & 0 \\ 0 & 0 & a_{23} \\ 0 & 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 \\ 0 \\ b_3 \end{bmatrix}$. where all the entries denoted by zeros are fixed and all the other entries are not fixed. The graph of this pair has a form of a stem (to be defined in next section). This pair is easily seen to be structurally controllable [see Figure 2.2].

Example 2.3: Another basic example

$A = \begin{bmatrix} 0 & a_{12} & 0 \\ 0 & 0 & a_{23} \\ a_{31} & 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 \\ 0 \\ b_3 \end{bmatrix}$.The graph of this pair has a form of a bud (to be defined in next section). This pair is easily seen to be structurally controllable.

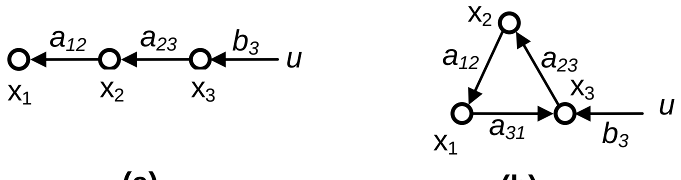


Figure 2.2 Examples of graph of pair *(A,B)*. (a) Graph of pair *(A,B)* in example 2.2. This is a ***stem***. **(b)** Graph of pair *(A,B)* in example 2.3. This is a ***bud***.

Examples of controlled systems that are uncontrollable, structurally controllable, and strongly structurally controllable are shown in Figure 2.3.

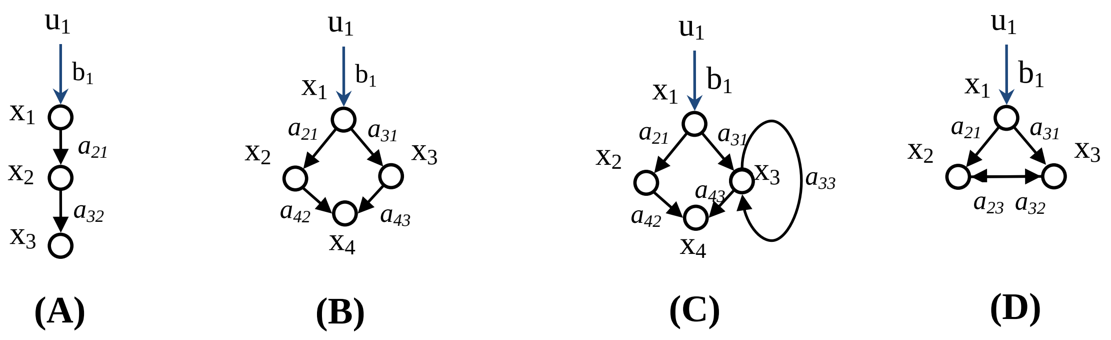


Figure 2.3 Some simple controlled networks. Here, x_1, x_2, x_3, x_4 are state nodes; a_{ij} are values of the connections between system components; u_1 is the input node, with input signal b_1. **(A)** This system is completely controllable, or in other words, *strongly structurally controllable*: its controllability is independent of the actual values of b_1 and a_{ij}, as long as they are non-zero. **(B)** The network is *uncontrollable*. To control it, we need to inject one more input into either node x_2 or x_3. **(C)** This system is *strongly structurally controllable*. Changing the system configuration by adding a self-loop at node x_3 makes the new system controllable, although its original form in **(B)** is not controllable. **(D)** This system is *structurally controllable*. It is uncontrollable when the system parameters a_{ij} accidentally satisfy the constraint $\boldsymbol{a_{32}a_{21}^2 = a_{23}a_{31}^2}$. But after slightly changing parameters off those pathologic cases, the system becomes controllable.

2.3 Lin's Theorem

Lin's approach to the controllability problem (1) was graph theoretic and required a lengthy series of definitions to derive the condition for structural controllability which is a purely algebraic statement.

For the controlled system denoted by the pair *(A,B),* we consider its graph representation. That is, the directed graph *G(A,B)*. For this digraph, the definition of a stem is slightly different from Definition 1.6, while definitions of buds and cacti remain the same. The new definition of a stem is the following.

Definition 2.4 *A **stem** in G(A,B) is an elementary path originating from an input vertex. The initial (or terminal) vertex of a stem is called the root (or top) of the stem.*

This new definition differs from Definition 1.6 by the presence of the input vertex of a controlled network. The definition of a **bud** is the same as in definition 1.7. From now, we use the word stem to denote the stem in the controlled network *G(A,B)*, unless otherwise specified. Examples of a stem and a bud are shown in **Figure 2.2**.

Definition 2.5 *A **cactus** in G(A,B) is a digraph defined recursively as follows. A stem is a cactus. Given a stem S_0 and buds $B_1, B_2, \dots B_l$, then $S_0 \cup B_1 \cup B_2 \cup \dots \cup B_l$ is a* *cactus if for every i* $(1 \leq i \leq l)$ *the initial vertex of the distinguished edge of* B_i*is not the top of* S_0 *and is the only vertex belonging at the same time to* B_i *and* $S_0 \cup B_1 \cup B_2 \cup \ldots \cup B_{i-1}$. *A set of vertex-disjoint cacti is called a* ***cacti****.*

Note that, by this definition, a cactus in *G(A,B)* is always originated from an input vertex. In our work, we use the word cactus to denote ***cactus in G(A,B)***, unless otherwise specified.

Definition 2.6 *A state vertex* x_i *in the digraph G(A,B) is called* ***inaccessible*** *if and only if there is no directed path reaching* x_i *from any of the input vertices (origins).*

Definition 2.7 *The digraph G(A,B) contains a* ***dilation*** *if and only if there is a subset* $S \subset V_A$ *(where* V_A *is the set of states vertices) such that* $|T(S)| < |S|$. *Here, the neighborhood set T(S) of S is defined to be the set of all vertices* v_j *such that there exists an oriented edge from* v_j *to a vertex in S, i.e* $T(S) = \{v_j | (v_j \rightarrow v_i) \in E(G), v_i \in S\}$. *The origins are not allowed to belong to S but may belong to T(S). |S| or |T(S)| is the cardinality of set S or T(S), respectively;* V_A *is the set of states vertices,* $E(G)$ *is set of edges.*

Propositions 2.1 [5]

1. An isolated node with self-edge is not a dilation. But all isolated nodes are inaccessible.
2. Cactus (or cacti) is the minimal structure which contains neither *inaccessible* nodes nor *dilations*. That is, removing an arbitrary edge will cause either inaccessibility or dilation.

Theorem 2.1 (Lin's Theorem on Structural Controllability [2]) The following three statements are equivalent:

1. A linear control system *(A,B)* is structurally controllable.
2. i) The digraph *G(A,B)* contains no inaccessible nodes.
   ii) The digraph *G(A,B)* contains no dilation.
3. The digraph *G(A,B)* is spanned by cacti.

The proof of this theorem is clearly presented in Lin's paper [2].

According to [5], some explanations are given as the following. A system is structurally uncontrollable if its digraph *G(A,B)* contains either inaccessible nodes or dilations. In the case when inaccessible nodes are present, these nodes cannot be controlled by input vertices since the input vertices cannot access to them. For example, a node that has in-degree zero (meaning that no other nodes point to it) cannot get any control signal from input vertices. Another naive example is the isolated nodes. In the case when dilations are present, the system is also uncontrollable. Each dilation forms a sub-graph in which there are more nodes to be pointed to or "controlled" by fewer other nodes. In other words, there are more "subordinates" (nodes under control) than "superiors" (nodes that control other nodes).

Consequently, to get full control of the network, dilations and inaccessible nodes must be removed. This means that, each node of the controlled network must have its own "superior". If a node has no superior, then it is inaccessible, we lose control of it. If it shares superior with other nodes then dilation is formed. In both cases the system is uncontrollable.

2.4 Minimum Input Theorem.

Recently, Liu et al. established a theorem called "Minimum Input Theorem" [5], which provides a simple way to determine the minimal set of nodes (called driver nodes) needed to fully control a dynamic network *G(A).* In this section, the Minimum Input Theorem will be given and proved. we first start with a basic definition of controlled nodes and driver nodes.

Definition 2.8 *In the controlled network G(A,B), the state vertices that are directly connected to input vertices (or origins) are called* ***controlled nodes****. Those controlled nodes which do not share input vertices are called* ***driver nodes****.*

Examples of controlled nodes and driver nodes are shown in Figure 3.5 (B) and (C).

Note that, driver nodes are subset of controlled nodes. They are both from the digraph *G(A),* although they are directly connected to input vertices, which belong to *G(A,B)* only. Hence, to find the minimum number of inputs that are sufficient for fully controlling the systems, we can alternatively find the minimum set of driver nodes.

Minimum Input Theorem

We show the relationship between the maximum matching in the digraph *G(A)* of a system and the minimum number of driver nodes needed to fully control it. This relationship is the content of the Minimum Input Theorem below.

Consider the directed network *G(A*) with *N* nodes, and denote the size of a maximum matching *M* in it by |*M*|.

Theorem 2.2 (Minimum Input Theorem [5]) The minimum number of inputs (N_I) or equivalently the minimum number of driver nodes (N_D) needed to fully control a network *G(A)* is one if there is a perfect matching in *G(A).* (In this case, any single node can be chosen as the driver node.) Otherwise, it equals to the number of unmatched nodes with respective to any maximum matching. (In this case, the driver nodes are just the unmatched nodes.)

$$N_I = N_D = max\, \{N - |M|, 1\}$$

Proof

***Case 1**. There is a perfect matching, i.e.* $|M| = N$

Since there is a perfect matching in the digraph *G(A),* all state vertices are matched. This means that every state vertex must belong to one of the cycles. By introducing one input vertex, connecting it to all cycles to form buds, and modifying any bud to a stem (by deleting the edge that point to the common vertex of the cycle and its distinguished edge), we construct a cactus that spans the controlled network *G(A,B).* By Lin's theorem, the system *(A,B)* is structurally controllable, with one input vertex. Hence, $N_D = 1$.

***Case 2**: There is no perfect matching, i.e.* $|M| < N$

Since there is no perfect matching in *G(A),* there are some nodes that are unmatched. The number of unmatched nodes is $N - |M|$.

Consider an unmatched node *x.* There is no edge in *M* pointing to *x*. The unmatched node *x* must be the starting node of an elementary path in *M*. Now, we can connect one input vertex to each of unmatched nodes, and form *N*-|*M*| stems. This requires *N*-|*M*| input vertices.

Exclude all the nodes that belong to elementary paths in *M*, any of the other nodes of *G(A)* must belong to one of cycles in *M*. For any cycle *R* in *M*, if there is an edge $e(p,q)$ in *G(A)* that connects node p of any stem to node *q* of *R*, then $\{e\} U\ R$ forms a bud. If there is no such edge *e*, we can introduce new edge *e'* (belongs to *G(A,B)*) from any of the above *N*-|*M*| input vertices to any node of *R*. In this case $\{e'\}\ U\ R$ also forms a bud. Thus, forming buds does not require extra inputs.

Now we have a cacti (a cacti is a set of vertex-disjoint cacti) from those stems and buds which spans the controlled network *G(A,B).*The construction requires *N*-|*M*| inputs. According to Lin's theorem, the system is structurally controllable.

Furthermore, we will show that, N_D is the ***lower bound*** of the number of inputs needed, that is, less than N_D number of inputs could not fully control the network. Indeed, let *M* be the maximum matching in *G(A)*, |*M*| is the size of *M*, $N_D = max\{N - |M|, 1\}$. The case $N_D = 1$ is trivial, since we cannot control the network with less than 1 input. Let us consider the case when $N_D = N - |M| > 1$.

Suppose that *G(A)* is fully controlled by $N_I < N_D$ number of inputs, i.e., there exists a structurally controllable network G(A,B) with $N_I < N_D$ number of inputs. By Lin's theorem, there exists a cacti *C* that spans *G(A,B)* and *C* contains $N_I < N_D$ number of vertex-disjoint cacti. We will construct a new matching *M'* in *G(A)* from the cacti *C*, with the size larger than that of M, as follows.

For each cactus *c* of *C*, its stem originates from the input u_c. Note that, a cactus has one and only one stem. Let x_c be the node in the stem that u_c connects to. Let M_c be the set of remaining edges of *c* after removing all the distinguished edges and the edges from u_c. We have that, M_c contains an elementary path started from x_c, and all cycles of *c*. Thus, M_c can also be considered as a matching in *G(A),* in which the node

x_c is unmatched, and all other nodes of M_c are matched.

Let $M' = \bigcup_{c \epsilon C} M_c$, then M' is also a matching in *G(A)*, which contains N_I number of unmatched nodes. Since *C* spans *G(A,B)*, every node of *G(A)* appears in *M'*. The number of matched nodes in M' is obviously $N - N_I$, hence the size of matching M' is also $|M'| = N - N_I > N - N_D = |M|$. This means that, the matching M is not the maximum matching, a contradiction.

Hence, $N_I = N_D = max\,\{N - |M|, 1\}$ is the minimum number of inputs needed to fully control the network *G(A)* ■

Chapter 3 .

Spanning Cacti for Structurally Controllable Networks

We first introduce the problem of minimum spanning cacti for structurally controllable network and propose an algorithm to find it. Then, we introduce the one-input controllable networks, as well as a simple way to reconstruct them. For both problems, our approaches are based on Lin's theorem and the Minimum Input Theorem presented in Chapter 1.

3.1 Minimum Spanning Cacti Problem

First we define the minimum spanning cacti as the following:

Definition 3.1 *A **minimum spanning cacti** on G(A,B) is a spanning cacti containing the smallest number of inputs*.

The minimum spanning cacti problem is formulated in the following.

Problem *Given a directed network G(A) of a system, build the structurally controllable network G(A,B) with the minimum number of inputs, and span it by cacti. In other words, given a directed network G(A), find a minimum spanning cacti for the structurally controllable network G(A,B).*

In order to solve this problem, we first apply the Minimum Input Theorem, which gives us the minimum number of inputs needed to fully control the network *G(A).* This is a simple and straightforward approach, in which the set of input nodes is determined by the maximum matching. From this result, we can build the controlled network *G(A,B)* and find the spanning cacti for *G(A,B)*. Thus, for a linear dynamic system which is represented by *G(A)*, we can build a structurally controllable network *G(A,B)* with minimum number of inputs, as well as its minimum spanning cacti.

The minimum spanning cacti problem can be considered as a direct application of Lin’s Theorem and Minimum Input Theorem. Moreover, the solution of this problem would be an analytical tool for studying the controllability of complex networks of natural and technological systems [4].

3.2 Algorithm for Finding Minimum Spanning Cacti

The general pipeline for solving the above problem is described as the following:

1. From the given network G(A), find the minimum set of driver nodes using maximum matching;

2. Build the controlled network G(A,B);

3. Find the spanning cacti on G(A,B).

We describe the first step separately, and then the two last steps are described together. The overall algorithm – algorithm MSC (which stands for minimum spanning cacti) is presented in the last section.

3.2.1 Identifying the Minimum Set of Driver Nodes

According to the Minimum Input Theorem, to find the minimum set of driver nodes, we need to find a set of unmatched nodes of the maximum matching in the directed network *G(A)*. Since *G(A)* is directed, we need to represent it by the bipartite representation in order to apply the maximum matching algorithm we described previously.

The bipartite representation of a directed graph is defined in the following way:

Definition 3.2 *Given a directed graph G(A) = (V,E), where* $V = \{v_1, v_2, \dots, v_N\}$*,its* ***bipartite representation*** *is a graph:*

$$BP(A) = (V^+ \cup V^-, \Gamma)$$

where V^{+} *is the set of N plus nodes* $V^{+} = \{v_1^{+}, v_2^{+}, \dots, v_N^{+}\}$

V^{-}*is the set of N minus nodes* $V^{-} = \{v_1^{-}, v_2^{-}, \dots, v_N^{-}\}$

Γ *is the set of edges* $\Gamma = \{(v_i^{+}, v_j^{-}) \mid (v_i, v_j) \in E\}$.

Example of the bipartite graph-representation is illustrated in Figure 3.1.

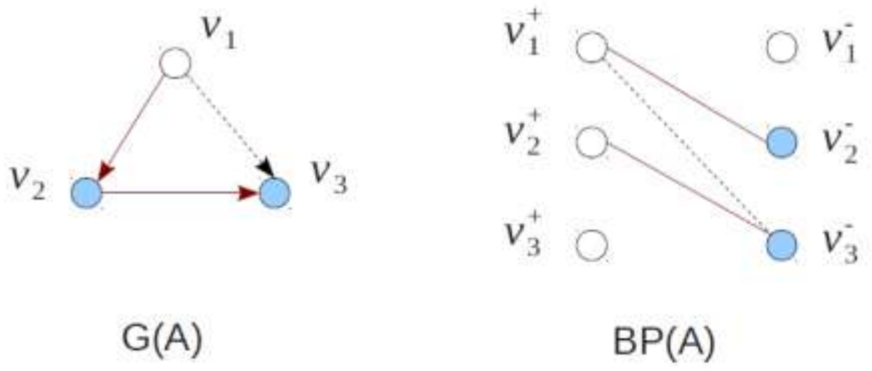


Figure 3.1 The bipartite representation of a directed graph and a matching in it. A vertex in directed graph *G(A)* is represented as a pair of a plus node and a minus node in bipartite graph *BP(A);* an oriented edge is represented as an undirected edge from the corresponding plus node to the corresponding minus node. The matching is shown in red; matched (or unmatched) nodes are shown in blue (or white), respectively.

Note that, each edge of the bipartite representation of a directed graph always contains a plus node and a minus node. Let *BP(A)* be the bipartite representation of a directed graph *G(A)*. For each edge (v_i^{+}, v_j^{-}) of *BP(A)*, the corresponding directed edge in *G(A)* is $(v_i\ , v_j\ )$; and conversely, for each directed edge (v_i, v_j) of *G(A)*, the corresponding edge in *BP(A)* is (v_i^{+}, v_j^{-}).

Let M be a matching in *BP(A)*. We construct the set M' of edges in G(A) corresponding to M as follows. If the edge (v_i^{+}, v_j^{-}) belongs to the matching M, we include the corresponding edge $(v_i\ , v_j\ )$into M'. Thus, $M' = \{(v_i, v_j) \mid (v_i^{+}, v_j^{-}) \in M\}$. Note that.$|M'| = |M|$

Proposition 3.1 M' is a matching in *G(A)*. M' is called the corresponding matching in *G(A)* of M.

Proof Firstly, M' is a subset of E.

Secondly, since M is a matching in *BP(A),* for each of its edge $(v_i^+, v_j^-) \in M$, there is no other edge of M that is adjacent to v_i^+ . Hence, the corresponding edge (v_i, v_j)in M'does not share the starting vertex v_i with any other edge of M'. Similarly, there is no other edge of M that is adjacent to v_j^-; hence, the edge (v_i, v_j) of M' does not share the ending vertex v_j with any other edge of M', as well. So, M' is the subset of E in which no pair of edges shares the same starting or ending vertex. In other words, M' is a matching in *G(A)*■

Conversely, given a matching $M' = \{(v_i, v_j)\}$ in *G(A),* we can construct its corresponding matching M in *BP(A)* as the following: $M = \{(v_i^+, v_j^-)|(v_i, v_j) \in M'\}$. Note that $|M| = |M'|$.Similarly, it is easy to show that M is also a matching in *BP(A). M* is called the corresponding matching in *BP(A)* of M'.

Now we will show that the corresponding matching of a maximum matching is also maximum.

Proposition 3.2 If a matching M in *BP(A)* is a maximum matching, then its corresponding matching M'in *G(A)* is maximum.

Proof.

Since M' is the corresponding matching in *G(A)* of M, we have$|M'| = |M|$.

Suppose that M' is not the maximum matching in *G(A),* that means there is a matchingM_1 in *G(A)* such that$|M_1| > |M'|$. Let M_1' is the corresponding matching in *BP(A)* of M_g. We have $|M_1'| = |M_1|$. Since M is the maximum matching in *BP(A),* we have $|M| \geq |M_1'|$. Here we have a contradiction: $|M'| = |M| \geq |M_1'| = |M_1| > |M'|$.

Thus, M' is also the maximum matching in *G(A)* ■

Return to our problem of finding minimum driver node set. Given a maximum matching M in the directed graph *G(A),* the set of driver nodes can be identified as follows. If M is a perfect matching, i.e., all nodes in *G(A)* are matched, then we can choose any node to be the driver node. In our approach, we choose the node with the highest degree as the driver node. If M is not a perfect matching, then we identify the set of unmatched nodes. These unmatched nodes are chosen as driver nodes.

Steps for finding minimum set of driver nodes N_D in directed network *G(A)* can be summarized as the following (Figure 3.2).

Input: Directed graph *G(A)*;
Output: Minimum set of driver nodes *D*
Step1: Build the bipartite graph *BP(A)* from the directed network *G(A)*, using *Definition 3.2;*
Step2: Find the maximum matching from bipartite graph *BP(A)*, using Algorithm 1.1, and build its corresponding maximum matching *M* in *G(A);*
Step3: Identify the minimum set of driver nodes *D* from the maximum matching *M*.

Figure 3.2 Steps for finding minimum set of driver nodes.

3.2.2 Building the Controlled Network and Finding the Spanning Cacti

i) Build the controlled network

With the set of driver nodes D, we can easily build the controlled network by just introducing $N_D = |D|$ input vertices: $\{u_1, u_2, \dots, u_{N_D}\}$ and connecting them to driver nodes one by one. However, this initial controlled network, denoted as $G_0(A,B)$, may be still uncontrollable, due to the fact cycles in this network may be inaccessible from the inputs (see Figure 3.5 (B) for illustration). It will become controllable if more control connections are added from inputs to these cycles. If we introduce all possible new connections from inputs to all cycles, the controlled network will definitely be structurally controllable, but many of these connections may be redundant. Thus, we just build the initial controlled network $G_0(A,B)$ first, then we modify it to be structurally controllable by properly adding new connections. We identify these proper connections while constructing the spanning cacti based on $G_0(A,B)$.

Build the initial controlled network

An initial controlled network $G_0(A,B)$ is built from the set of driver nodes *D* and the network $G(A) = (V_A,E_A)$ as follows. First, a set V_B of N_D number of inputs are introduced. Then, for each input, we connect it to a driver node, if the driver node is not connected by any input. These new edges are assigned to the set of initial controlled edges E_B. The network $G_0(A,B)$ is the combination of *G(A)* with V_B and E_B.

The proof of Minimum Input Theorem presented in the Chapter 2 gives a way to construct stems and buds from the maximum matching. That is, each of the unmatched nodes initiates an elementary path; each elementary path forms a stem.

Beside, all the matched nodes excluding nodes in elementary paths are clustered into cycles of matched nodes; those cycles can further be modified into buds.

ii) Finding the spanning cacti

Now we move on to build the spanning cacti, at the same time we add more controlled edges into *G0(A,B)* to get the structurally controllable network. To do that, we first find all stems and buds. An example of building a spanning cacti is illustrated in Figure 3.5.

Finding stems

A stem is an elementary path that originates from an input node. Thus, the root of a stem is an input node; the node that the root points to is a driver node. A driver node is an unmatched node with respect to the maximum matching *M* in the directed network *G(A).* Given the set of driver nodes, from each of the driver nodes we can find an elementary path. After that, we add the corresponding input node into the path. Since there are N_D driver nodes (and N_D input nodes as well), there will be N_D stems in total.

Note that, stems belong to the network *G(A,B),* but the paths from driver nodes, including driver nodes themselves, belong to the network *G(A).* Only the input nodes do not belong to *G(A).* The algorithm for finding all stems is presented in Figure 3.3.

Algorithm 3.1: FINDING STEMS FROM MATCHING

Input: Maximum matching represented by array *mate[]*; set of driver node *D*;
Output: List of N_D stems $STEM_1$, $STEM_2$, …, $STEMN_D$

```
begin
    for each $n_j$ in D do begin
        $STEM_j$ = {};      // empty
        x = $n_j$;
        while (mate[x] != 0) do begin // x has its mate
            $STEM_j = STEM_j \cup (x, mate[x])$ ;
            x = mate[x];
        end;
        $STEM_j = (u_j, n_j) \cup STEM_j$;
    end;
end;
```

Figure 3.3 The algorithm for finding stems

Finding Cycles

A bud is an elementary cycle with a distinguished edge. Here we describe a procedure to find the elementary cycles. The procedure for finding a distinguished edge for each cycle is described in the next section of finding the spanning cacti. As we know previously, all the matched nodes excluding those nodes on stems are clustered into cycles. And those cycles can be easily found from the matching as well. Let *W* be the set of matched nodes excluding all the nodes on stems:

$$W = \{matched\ nodes\} \setminus \{nodes\ on\ stems\}$$

The procedure for finding cycles of matched nodes in *G(A)* is the following (Figure 3.4):

Algorithm 3.2:FINDING CYCLES OF MATCHED NODES

Input: set of nodes W; maximum matching represented by array *mate[]*
Output: set of cycles C;

```
begin
    for each v in W do visited[v] = false;   // initialize
    for each v in W do
        if (not visited[v]) then begin
            visited[v] = true;   x = v;
            aCycle = {};         // empty
            repeat
                aCycle = aCycle ∪ (x, mate[x]);
                x = mate[x];
                visited[x] = true;
            until x = v;         // meet the first node – form a cycle
            C = C ∪ aCycle;
        end;
end;
```

Figure 3.4 The algorithm for find cycles of matched nodes

Finding the spanning cacti

From the maximum matching in the directed network *G(A)*, we can find stems and cycles. To build the spanning cacti, we just need to connect cycles to stems properly. In other words, we need to find a distinguished edge for each cycle.

Case 1 There are only stems, no cycles

Since a stem is also a cactus. Hence, the spanning cacti is the combination of all stems. The initial controlled network $G_0(A,B)$ is structurally controllable, there is no any new edge to be introduced.

Case 2 There are both stems and cycles

We need to connect each cycle *c* to one of the stems. To do that, we can use one of the available connections of *G(A)* or introduce a new connection to *c*.

Distance from a stem *s* to the cycle *c* is defined as the length of the shortest path from any node of *s* to any node of *c*. If that distance is 1, then we say the stem *s* directly connects to cycle *c*. The edge *e* that connects *s* to *c* is called the direct connection edge.

- There is a direct connection edge *e* from one of the stems to the cycle *c*. In this case, we simply connect *e* to *c*. The edge *e* becomes the distinguished edge for *c*, and ($c \cup \{e\}$) becomes a bud. Note that, there may be many such direct connections; we can choose any of them. A procedure for building cacti using available connections is presented in Figure 3.6 phase 1.

- There is no direct connection edge from any stem to the cycle *c*. In this case, we need to introduce new edge *e* from one of the input nodes to the cycle *c*. The issue here is which input node to choose.

  From the original graph *G(A,B)*, we identify all the connected components. If the cycle *c* belongs to the same connected component with some stems s_1, $s_2, \ldots, s_k$, then among them we identify the stem s_j that the distance from s_j to a node *x* of *c* is minimum. We connect the origin u_j (input node) of s_j to node *x* of *c*. The edge $e = (u_j,x)$ is the distinguished edge for *c*.

  If the cycle *c* is isolated from any stem, then we can choose any input node *u*, and connect *u* to any node *x* of the cycle *c*. The edge $e = (u,x)$ is the distinguished edge for *c*. However, to avoid building a cactus of large size, we suitably choose the input node *u* that currently rooted in a cactus of least size.

  A procedure for building cacti using new connections is presented in Figure 3.7 phase 2.

Case 3 There are only cycles, no stems

In this case, we know that the structurally controllable network *G(A,B)* will contain only one input node; and its spanning cacti contains one stem. Hence, we just need to modify one of the cycles into a stem, and thus, reduce this case into case 2 above. We can choose any cycle *c*, and delete any edge $e = (x,v)$ of *c*. The cycle *c* now becomes an elementary path which starts from node *v*. The node *v* becomes the only driver node of *G(A)*. The only stem in *G(A,B)* is formed by introducing an input node *u*, and connecting *u* to the driver node *v*.

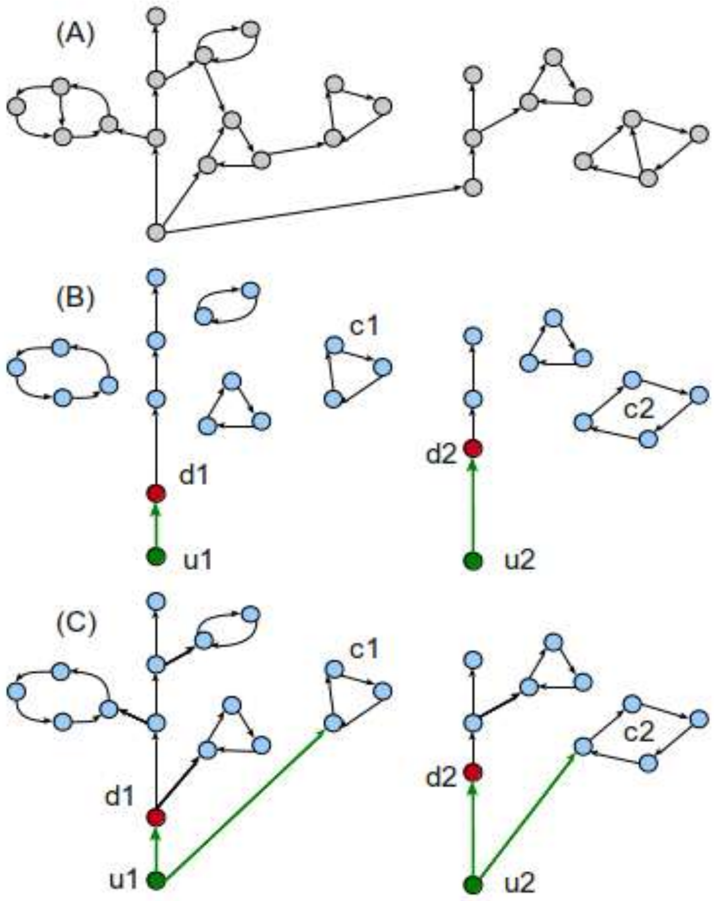


Figure 3.5 Example of building the spanning cacti for a directed network. (**A**) An example network *G(A)*. (**B**) Stems and cycles in the initial controlled network $G_0(A,B)$. Matched nodes are in light blue, driver nodes d_1 and d_2 are in red; input nodes u_1 and u_2 are in green. Applying the maximum matching algorithm, we find the set of driver nodes $\{d_1, d_2\}$. Then 2 input nodes u_1 and u_2 are introduced. After that, 2 stems and 6 cycles are constructed by algorithms 3.1 and 3.2. Note that, this *initial controlled network is not controllable*, because cycles are isolated from input nodes (hence, they are inaccessible). (**C**) Building cacti by connecting cycles and stems. Edges that shown in green are new edges introduced for *G(A,B)*. In the original network *G(A)*, the cycle c_1 is in the same connected component with both two driver nodes. Then for building the cactus, it is connected to input node u_1, since it is nearer. The isolated cycle c_2 is connected to input node u_2, since currently the cactus of u_2 is smaller in size.

For all cases, when a distinguished edge is found, if it previously does not exist in $G_0(A,B)$, it is included into the initial controlled network $G_0(A,B)$. After the spanning cacti is found, the network $G_0(A,B)$ becomes structurally controllable.

Note that, both case 1 and case 3 are sub-cases of the case 2. Hence, for building the spanning cacti, we can use the two procedures described in case 2 (Figure 3.6 and Figure 3.7) as follows. In phase 1, we build the initial spanning cacti using available connections. And then, in phase 2, we complete building the spanning cacti by introducing new connections.

Algorithm 3.3: Building cacti using available connections

```
Input: set of cycles C, set of stems ST, the di-graph G(A)
Output: initial cacti representing by array CACTI;
begin
    for j = 1 .. $N_D$ do CACTIj := STEMj;
    Find connected components in G(A);
    Calculate distance between any cycles in C to any stems in ST;
        for each c in C do begin
        for each s in ST do
        if distance(c,s) = 1 then begin
                Let e be the distinguished edge connects s to c;
                CACTIs := CACTIs ∪ c ∪ e;
        end;
    end;
end;
```

Figure 3.6 Building the initial cacti using available connections (phase 1). Note that finding connected components and calculating distances between cycles and stems can be done using breath-first search, with time-complexity $\boldsymbol{O}(|\boldsymbol{C}| \cdot |\boldsymbol{V_A}|)$, where $|\boldsymbol{V_A}|$ is the number of nodes in *G(A).*

Let *C'* be the set of remaining cycles after phase 1.

Algorithm 3.4: Building cacti using new connections.

```
Input: Initial cacti, set of remaining cycles C' after phase 1, initial controlled network $G_0$(A,B);
Output: Minimum spanning cacti, controlled network G(A,B).
begin
   G(A,B) := G0(A,B);
   for eachc in C' do begin
   ifc is isolated then begin   //c is an isolated cycle
           Let s be the cactus currently has least number of nodes;
           Let v be any node in c;
           CACTIs := CACTIs ∪ c ∪ ($u_s$,v); // $u_s$ is root of cactus s;
           Add the edge ($u_s$,v) into G(A,B);
   end;
   else begin// c is in the same connected component with some cacti
           Let s be the cacti such that distance(c,s) is minimum;
           Let v be any node in c;
           CACTIs := CACTIs ∪ c ∪ ($u_s$,v);
           Add the edge ($u_s$,v) into G(A,B);
   end;
   end;
end;
```

Figure 3.7 Connecting remaining cycles to cacti using new connections (phase 2). This procedure can be done in *O(C').* In the results, the array CACTI stores the minimum spanning cacti for controlled network *G(A,B).*

Note that, in the second phase described above, we build the spanning cacti, and at the same time, build the controllable network *G(A,B).*

3.2.3 Algorithm MSC and Its Correctness

Algorithm for minimum spanning cacti (MSC)

Using all the procedures described in previous sections, we develop a simple algorithm for solving minimum spanning cacti problem. The algorithm MSC is summarized as the following (Figure 3.8):

```
Algorithm MSC
    Input: A directed network G(A) = (V_A,E_A) of the system under control;
    Output: Minimum spanning cacti for controlled network G(A,B);
    begin
        Step1: Build the bipartite graph BP(A) from the directed network G(A), using
               Definition 3.2;
        Step2: Find maximum matching from bipartite graph BP(A), using Algorithm 1.1,
               and build its corresponding maximum matching M in G(A);
        Step3: Identify the minimum set of driver nodes D from the maximum matching M;
        Step4: Building the initial controlled network G_0(A,B);
        Step5: Finding stems and cycles from the matching M, using Algorithm 3.1 and
               Algorithm 3.2;
        Step6: Building cacti using available connections of G(A) by Algorithm 3.3;
        Step7: Completing building spanning cacti and controllable network G(A,B), using
               Algorithm 3.4;
    end;
```

Figure 3.8 Algorithm MSC for solving minimum spanning cacti problem

The correctness and time-complexity of MSC algorithm is given in the following theorem:

Theorem 3.1 *The MSC algorithm correctly solves the minimum spanning cacti problem. It runs in* $O(|V_A| \cdot |E_A|)$ *time.*

Proof <u>*The correctness*</u>

We will show that the network *G(A,B)* built by MSC is structurally controllable, and it contains the smallest possible number of input nodes. Alternatively, we will show that MSC correctly builds a spanning cacti for *G(A,B)*, and the spanning cacti is minimum.

Consider the given network *G(A)* = (V_A, E_A). Denote the resulting network *G(A,B)* as *G(A,B) = (V,E).* We have $V = (V_A \cup V_B)$, $E = E_A \cup E_B$, where V_B and E_B are built by MSC, that is, V_B is the set of input nodes, and E_B is the set of newly introduced edges by MSC. Denote the resulting cacti by C.

The cacti C spans G(A,B)

We will show that any node of *G(A,B)* is included in the cacti *C*, and any edge of *C* is an edge of *G(A,B)*. Indeed, consider any node x of *G(A,B)*. The node x is either an input node, or a node of *G(A)*. If x is an input node, then it is the root of a stem, according to step 5 and Algorithm 3.1 (finding stems). If *x* is a node of *G(A)*, $x \in V_A$, then it is either a matched node or an unmatched node with respect to the matching M. Hence, *x* must belong to either an elementary path or a cycle, according to step 5. In other words, *x* is in the cacti *C*.

On the other hand, consider any edge e of the cacti *C*. The edge e must belong to a stem, a cycle, or it is a distinguished edge. If *e* belongs to a stem or a cycle of the cacti, it must come from the matching *M* in *G(A)*, according to Algorithms 3.1 and 3.2 (finding stems and cycles). If e is a distinguished edge then either it is an available connection found by step 6 (Algorithm 3.3) or it is newly introduced by step 7 (Algorithm 3.4). In the former, e comes from E_A, while in the later, it is included in E_B. Thus, for all cases, *e* is an edge of *G(A,B)*.

The cacti C is the minimum spanning cacti

We have previously shown that the Algorithm 1.1 correctly finds a maximum matching for the bipartite graph *BP(A)* – the bipartite representation of *G(A)*. Hence, step 1 and step 2 correctly find a maximum matching *M* for the directed network *G(A)*.

Since the matching *M* contains the largest possible number of matched nodes, the number of unmatched nodes is minimum. In other words, the number of driver nodes is a minimum (by step 3 and the Minimum Input Theorem).

According to step 4, for each driver node, we introduce one input node. From each input node we build a stem and from each stem, we build a cactus. Hence, the cacti *C* contains the smallest possible number of cacti. *C* is a minimum spanning cacti. Therefore, the MSC algorithm correctly find the minimum spanning cacti for *G(A,B)*. Since *G(A,B)* is spanned by cacti *C*, according to Lin's theorem, it is structurally controllable. Thus, MSC algorithm correctly solves the minimum spanning cacti problem.

<u>*Time-complexity*</u>

For time-complexity, we observe that, the running time of MSC is dominated by the step 2 (Algorithm 1.1). Running time for all other steps is small compared to that of finding the maximum matching. The algorithm in step 2 takes $O(|V_A| \cdot |E_A|)$ time, thus, overall MSC takes $O(|V_A| \cdot |E_A|)$ time.■

Note that, we can improve the running time for MSC algorithm by applying the maximum matching algorithm that uses max-flow approach. That algorithm takes only $O(|V_A|^{1/2} \cdot |E_A|)$ time. Hence, we can improve MSC to run in $O(|V_A|^{1/2} \cdot |E_A|)$ time.

3.3 One-Input Controllable Networks with Minimal Modification

In chapter 2, we presented the work of both Lin (1974) and Liu et al. (2011) on structural controllability. Lin's theorem gives the sufficient and necessary conditions of structural controllability for linear dynamic systems. The conditions are described by the cactus representation of the systems' networks. Liu et al give a qualitative measure on the difficulty of controlling a network via the number of inputs needed to fully control it. In these works, given a fixed network *G(A*) of a underlying system, the controllability of the controlled network *G(A,B)* is evaluated. A particular interest is that, given *G(A),* one wants to build *G(A,B)* with a minimum number of inputs such that *G(A,B)* is structurally controllable.

As the structural controllability suggests, adding more links into the network never weakens its controllability. In fact, the Minimum Input Theorem gives an upper bound on the minimum number of inputs needed to control a network with missing links [5]. With more links added, the number of inputs needed may be fewer, i.e. the network becomes easier to control.

Moreover, in practice, the configurations of some human designed systems may change over time, such as computer networks, telecommunication networks and other technological networks. Thus, the network *G(A)* in reality may be changed. If we are allowed to modify the network *G(A)* by adding more links, one question that can be asked is**: *what is the minimum number of extra links needed for G(A) so that one can control it with only one input?*** As it turns out, this problem is equivalent to the problem of adding more edges into a directed graph to get a perfect matching. Furthermore, it can be also answered by directly applying the Minimum Input Theorem.

We call a network that can be controlled by one input as an one-input controllable network. This section presents the one-input controllable networks, their properties, and how to construct them.

3.3.1 One-Input Controllable Networks

We introduce the notions of one-input and n-input controllability to describe the level of difficulty in controlling directed networks.

Definition 3.3 *A network G(A) is called one-input controllable if the minimum number of input needed to fully control it is one. Otherwise, it is called n-input controllable.*

Examples of one-input and n-input controllable networks are shown in Figure 3.9.

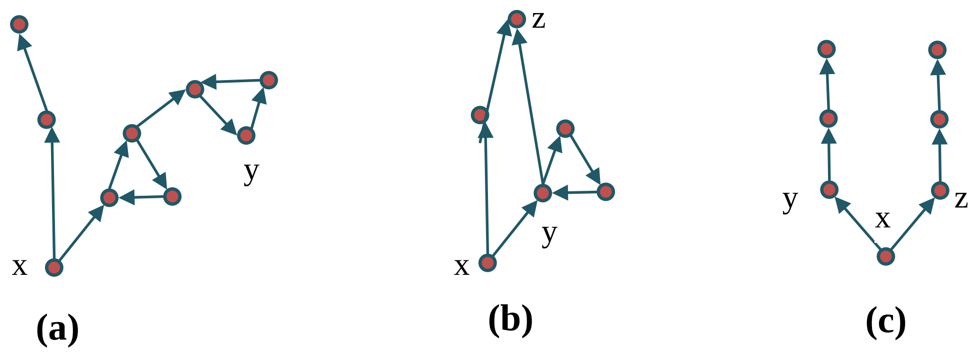


Figure 3.9 Examples of one-input and n-input controllable networks. (a) Network is one-input controllable, since we can control it by introducing one input and connecting this input to x and y. (**b**) Network is also one-input controllable; we can control it by one input connecting to node x. This network also can be spanned by a general directed cactus graph. The cactus can be formed by deleting the edge from y to z. (**c**) This is a 2-input controllable network. We cannot control all three nodes x, y, z by one input. To control this network, we need at least two inputs connecting to x and either to y or to z.

Note that, an one-input controllable network is covered by a cactus. A general directed cactus graph is one-input controllable. Some properties of one-input controllable networks are expressed in following propositions.

Proposition 3.3 A directed network *G(A)* is one-input controllable if and only if there is a cactus that spans *G(A,B).*

Proof. If *G(A)* is one-input controllable, then it can be controlled by one input. Thus, the controlled network *G(A,B)* is controllable with one input. By Lin's theorem, *G(A,B)* is spanned by a cactus.

On the other hand, if there is a cactus that spans *G(A,B)*, then by Lin's theorem, *G(A,B)* is controllable. Since a cactus in *G(A,B)* originates from an input, this means that *G(A)* can be controlled by one input, hence, it is one-input controllable ■

Let N denote the number of node in the network *G(A).*

Proposition 3.4 A directed network *G(A)* is one-input controllable if and only if the maximum matching in *G(A)* has the size of at least N-1.

Proof Let M be a maximum matching in *G(A)*, its size is denoted as |M|.

Suppose *G(A)* is an one-input controllable network, i.e., it can be controlled by only one input. By Minimum Input Theorem, the minimum number of inputs needed to fully control *G(A)* is $N_I = max\{N - |M|, 1\}$. Hence, $max\{N - |M|, 1\} = 1$. In other words, either M is a perfect matching or the number of unmatched node in M is 1. Thus, the maximum matching in *G(A)* must have the size at least N-1.

On the other hand, suppose *M* has the size of at least *N-1*. Then by Minimum Input Theorem, the minimum number of inputs needed to control *G(A)* is $N_I = max\{N - |M|, 1\} = 1$. Thus, *G(A)* is an one-input controllable network ■

3.3.2 Constructing One-Input Controllable Networks with Minimal Modification

Come back to our question of how to modify the network *G(A)* into an one-input controllable network. In general, this question is meaningful for human-designed networks, since one may want to build the system which is simple to control or easy to manipulate. The modification here is made by adding more links into the original networks without adding nodes. The question can be addressed in the following theorem.

Consider a network *G(A)* with *N* nodes, and a maximum matching *M* in *G(A)* with the size of $|M|$.

Theorem 3.2 The minimum number of extra links N_E needed to modify *G(A)* into an one-input controllable network is zero if there is a perfect matching in *G(A)*; otherwise, it equals $N - |M| - 1$, where |M| is the size of a maximum matching in *G(A).*

$$N_E = max\{N - |M| - 1, 0\}$$

Proof

By the proposition 4.2, if there is a perfect matching in *G(A)*, or a matching with size at least *N -1*, then *G(A)* is an one-input controllable, and vice versa. In that case, no extra link is needed.

Consider the case when the maximum matching *M* in *G(A)* has the size less than *N-1.*

By Liu et al., the minimum number of inputs needed to fully control the network *G(A)* is: $N_I = max\{N - |M|, 1\} = N - |M|$. By Lin's theorem, the controlled network *G(A,B)* can be controllable with N_I inputs; and *G(A,B)* can be spanned by N_I vertex disjoint cactuses, in which each input connects to a driver node on a stem.

We can introduce $N_E = N_I - 1$ extra links as follows. For each two stems s_1 and s_2, connect the top node of s_1 to the driver node of s_2. After $N_I - 1$ such modifications, only one stem is left. So, the new network now can be controlled with only one input.

We will show that, less than $N_I - 1$ extra links cannot modify *G(A)* into one-input controllable networks. Indeed, suppose *G'(A)* is the one-input controllable network after adding $N_E < N_I - 1$ extra links into *G(A).* Let M_0 be a maximum matching in *G'(A),* we have $|M_0| \geq N - 1$, by proposition 4.2. Let M_1 be a matching after removing such N_E links of G'(A). We have M_1 is a matching in *G(A)* with the size:

$$|M_1| \geq |M_0| - N_E > |M_0| - N_I + 1 \geq N - 1 + N_I + 1 = N - (N - |M|) = |M|$$

Hence, M_1 is the matching with a lager size than *M*, a contradiction with the fact that *M* is a maximum matching in *G(A)* ■

Constructing One-Input Controllable Networks

The above proof gives a way to construct the one-input controllable network from the original network *G(A).* Steps for such construction are summarized as the following.

1. Find a maximum matching *M* of *G(A)* using bipartite representation.
2. Find stems and cycles from the matching M.
3. Merge all stems into one, using $N_S - 1$ extra links, each link connects a top node of a stem to the root node of another stem, where N_S is the original number of stems.
4. The resulting network after adding $N_S - 1$ links is one-input controllable.

Chapter 4

Experimental Results

In the previous chapter, we presented the MSC algorithm for solving the minimum spanning cacti problem. In this chapter, we will present our experimental results. The main purpose of our experiments is to use the MSC algorithm for finding the minimum spanning cacti, and exploring the properties of the structural controllability of several computer-generated networks and real networks.

For each of these networks, we first ran MSC to build the structurally controllable network *G(A,B)* and to find the spanning cacti. After that, we analyzed the result by calculating some statistics on the minimum spanning cacti, e.g. the number of stems and buds in the cacti, number of the driver nodes, the ratio of driver nodes over the total number of nodes. We also constructed the one-input controllable networks from the original networks. Finally, we visualized the cacti, the original networks and the one-input controllable networks by Cytoscape [8] – a network visualization software.

4.1 Random Network

In our experiments, two types of random networks are generated, including Erdos–Renyi random network model.

1. Type 1: Uniformly generated n nodes and m edges. The network is chosen uniformly at random from collection of all networks that have n nodes and m edges.

2. Type 2: Erdos–Renyi probabilistic random network. The network is generated by randomly connecting n nodes. Each edge has a probability p to be included, independently from the other edges.

Given a number of nodes n and a number of edges m, the random network of type 1 is generated by the following procedure (Figure 4.1):

Algorithm 4.1 Random network - type 1

Input: Number of nodes n, number of edges $m < n(n-1)$;
Output: random network of type 1;

```
begin
    if $m > n(n-1)$ then $m = n(n-1)$;
    Node_set = {1..n};    // initialize
    Edge_set = {};
    for i = 1..m do begin
        repeat
            u = random number in [1..n];
            v = random number in [1..n];
            if $(u \neq v)$ and $(u, v)$ is not in Edge_set then begin
                Edge_set = Edge_set + $(u, v)$;
                break;
            end;
        until false;
    end;
end;
```

Figure 4.1 Procedure for generating a random network of type 1

Given a number of nodes n and a probability p, the random network of type 2 is generated by following procedure (Figure 4.2):

Algorithm 4.2 Random network – type 2

Input: number of nodes n, probability p;
Output: random network of type 2

```
begin
        Node_set = {1..n};          // initialize
        Edge_set = {};
        for u = 1..n do
        for v = 1..n do
        if $(u \neq v)$ and $(u, v)$ is not in Edge_set then begin
                r = random number in [0..1];
                if $(r \leq p)$ then
                        Edge_set = Edge_set + $(u, v)$;
        end;
end;
```

Figure 4.2 Procedure for generating a random network of type 2

Note that, all these networks are directed networks. Furthermore, there is no self-loop and no repeated edges are generated. For each type of networks, we performed the experiment for 100 times, and then averaged the result.

Results

We note that, the number of driver nodes, N_D, always equals the number of cactuses, since each cactus is built from an input node. This is explained as a consequence of the Minimum Input Theorem. Furthermore, the number of stems is exactly the same as the number of cacti in the spanning cacti, which can be derived by definition of cacti.

Table 4.1 and Table 4.2 show the results by running MSC for type 1 and type 2 random networks, respectively. As can be seen, the number of stems or equivalently, the number of driver nodes N_D decreases as the number of network edges increases. When the network is sparse, it requires more input nodes to become structurally controllable. On the other hand, when the number of edges is large in comparison with the number of nodes (for example $N_E = 5N, 10N, 20N, \ldots,$ or $p = 0.05, \ldots$).

The number of driver nodes is very small and in most cases, is only 1. This suggests that the dense networks are easier to control. Thus, the result presented here is consistent with Liu et al result [4].

Table 4.1 Result of running MSC on random network of type 1. For each network of type 1, we show the number of nodes (N); the number of edges ($\boldsymbol{N_E}$); the average number of stems (<$\boldsymbol{N_{stems}}$>); the average number of buds ($< \boldsymbol{N_{buds}} >$) and the average ratio of the number of driver nodes over the total number of nodes ($< \boldsymbol{N_D/N} >$).

N	N_E	**<N_{stems}>**	**<N_{buds}>**	**<N_D/N>**
50	2*N	9	2	0.18
	5*N	1	4	0.02
	10*N	1	2	0.02
	20*N	1	2.8	0.02
100	2*N	29	1	0.29
	5*N	1	5	0.01
	10*N	1	1.1	0.01
	20*N	1	1.6	0.01
200	2*N	41.2	0.4	0.206
	5*N	1.4	3.4	0.007
	10*N	1	4.6	0.005
	20*N	1	3.1	0.005
500	2*N	108.5	0.6	0.217
	5*N	5.5	3.6	0.011
	10*N	1	3.3	0.002
	20*N	1	4.5	0.002

Furthermore, for random networks of type 2, when the probability p increases, the number of buds also increases. This may suggest that, the maximum matching found for random networks of this type tends to form more cycles of matched nodes than stems. These cycles become buds of the spanning cacti.

Currently in this work, we limited the number of nodes for random network by 500. For further analysis, we may need to check for larger networks as well. Nevertheless, we believe that the results will have the same trend.

Table 4.2 Results of running MSC on random network of type 2. For each network of type 2, we show the number of nodes (N); the probability (p) an edge to be included into the network; the average number of edges (<$\boldsymbol{N_E}$>); the average number of stems (<N_{stems}>); the average number of buds (<N_{buds}>) and the average ratio of the number of driver nodes over the total number of nodes (<N_D/N>)

N	p	<$\boldsymbol{N_E}$>	<N_{stems}>	<N_{buds}>	<N_D/N>
50	0.01	64	16	1	0.32
	0.02	87	13	3	0.26
	0.05	157	5	3	0.1
	0.1	282	1	4	0.02
100	0.01	212.4	20.9	0	0.209
	0.02	298	9	1	0.09
	0.05	567	1	2	0.01
	0.1	1061.6	1	4.2	0.01
200	0.01	788.6	5.6	2	0.028
	0.02	1211.6	1.5	4	0.008
	0.05	2375.7	1	4.6	0.005
	0.1	4384.3	1	5.6	0.005
500	0.01	4997.5	1	4.5	0.002
	0.02	7527.7	1	5.2	0.002
	0.05	14986.2	1	7.4	0.002
	0.1	27500.6	1	15.4	0.002

4.2 *Real Networks*

We ran MSC on several real networks, including technological networks (electronic circuits), social networks (World Wide Web, trust, intra-organizational networks) and biological networks (food webs, and metabolic networks).

The datasets are the subset of those that were analyzed in Liu et al [4], except for two networks *Enron* [18] and *Macaque brain* [19], marked with (*). Our datasets are downloaded from *http://hal.elte.hu/~enys/data.htm* [20]. The features of these datasets are summarized in the Table 4.3.

Table 4.3 Features of the real networks analyzed in this work. For each network, we show its type, name, number of nodes (N), edges (N_E), meaning of each directed edge, and the data source references.

Type	Name	N	N_E	Meaning of an edge $X \to Y$	Reference
Electronic circuits	s208	122	189	Value at Y depends on value at X	[9]
	s420	252	399		[9]
WWW	Political blogs	1224	19025	Y has link to X	[10]
Intra-organizational	Consulting	46	879	X received emails from Y	[11]
	Freemans-2	34	830	Y knows X	[12]
	Manufacturing	77	2228	X received emails from Y	[11]
	Enron (*)	156	1669	X received emails from Y	[18]
Food web	Little Rock	183	2494	X prays on Y	[15]
	Grassland	88	137		[13]
	Seagrass	49	226		[14]
Metabolic	E.coli	2275	5763	X controls Y	[16]
	S.cerevisiae	1511	3833		[16]
	C.elegans	1173	2864		[16]
Trust	College student	32	96	Y trusts in X	[17]
	Prison inmate	67	182		[17]
Other	Macaque brain	45	463	Area X is connected to area Y	[19]

Table 4.4 shows the result of running MSC on these networks. We noted that our results are consistent with results presented in Liu et al [4]. That is, we get the same results for the ratio N_D/N of the number of driver nodes over total number of nodes.

In addition to calculating that ratio for each network, we find the number of stems and buds of the spanning cacti. We also construct the spanning cactus for the one-input controllable network.

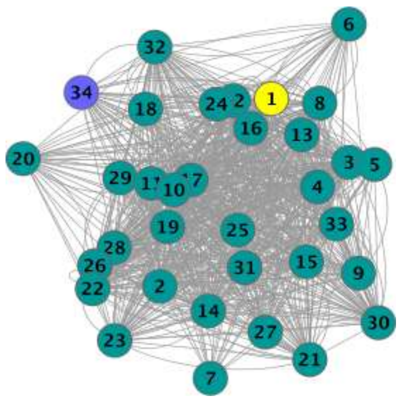
Table 4.4 Results of running MSC on real networks. For each network, we show its name; the number of nodes (N) and edges (N_E), the ratio of the number of driver nodes over the total number of nodes (N_D/N), the number of stems (N_{stems}) and buds (N_{buds}). The networks marked with (*) are not included in Liu et al. dataset.

Name	**N**	N_E	N_D/N	N_{stems}	N_{buds}
s208	122	189	0.2377	29	2
s420	252	399	0.2341	59	4
Political blogs	1224	19025	0.3562	436	35
Consulting	46	879	0.0435	2	3
Freemans-2	34	830	0.0294	1	6
Manufacturing	77	2228	0.0130	1	5
Enron (*)	156	1669	0.0321	5	15
Little Rock	183	2494	0.5410	99	5
Grassland	88	137	0.5227	46	0
Seagrass	49	266	0.2653	13	0
C.Elegans	1173	2864	0.3017	354	25
S.Cerevisiae	1511	3833	0.3289	497	31
E.Coli	2275	5763	0.3824	870	54
College student	32	96	0.1875	6	2
Prison inmate	67	182	0.1343	9	9
Macaque brain (*)	45	463	0.0222	1	5

Among those networks, the two food webs (*Little Rock* [15] and *Grassland* [13]) are the most difficult to control. To fully control each of the two networks, it is necessary to control more than 50% of the nodes. This contradicts our general intuition that the food webs are easy to control, since we expect that if we control some resources, we can control all the food webs. The controllability here is a property of the network itself, describing its ability to move around the state-space. So, manipulating some resources of the food webs may break the food webs, but a majority of resources are needed for the evolution of the food webs over time. On the other hand, the result shows that the intra-organizational networks are much easier to control. This dispels the expectation that these networks are resistant to control because they are quite dense: in the networks, each individual connects to many others, thus the controllability is believed to have no role. Yet the structural controllability suggests that in principle, a few individuals can fully control the whole network.

After running MSC, we used Cytoscape [8] to visualize the spanning cacti. Some cacti are shown in the following figures. The input nodes are shown in red; the driver nodes are in yellow; nodes of stems are in blue; nodes of buds are in green. The controlled edges are shown in red, edges of stems are in blue, edges of buds are in green, and distinguished edges are in orange.

(a)



(b)

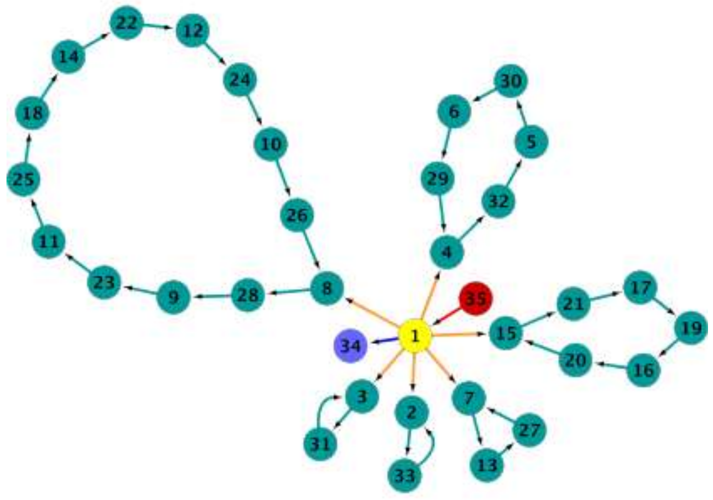
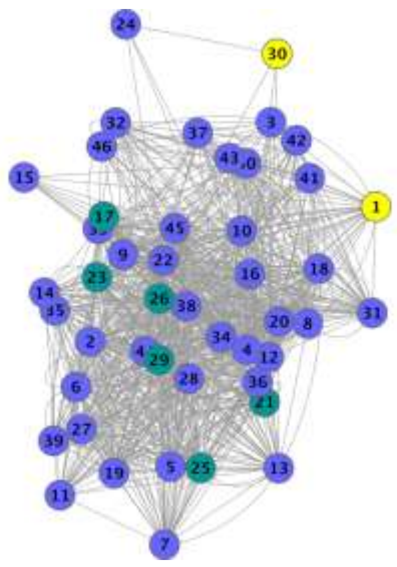
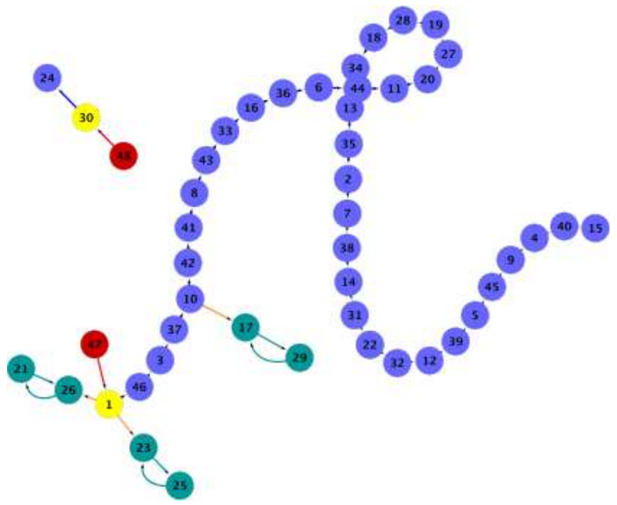


Figure 4.3 Visualizations of the intra-organizational network Freemans-2. (a) The original network (directed). **(b)** The minimum spanning cacti for the controllable network. The input node is *35;* it was introduced to control the network and it is the root of the cactus. Note that, this original network is an example of ***one-input controllable*** networks.

(a)



(b)



(c)

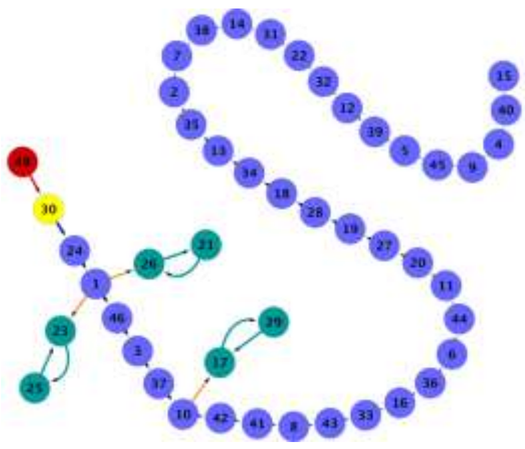
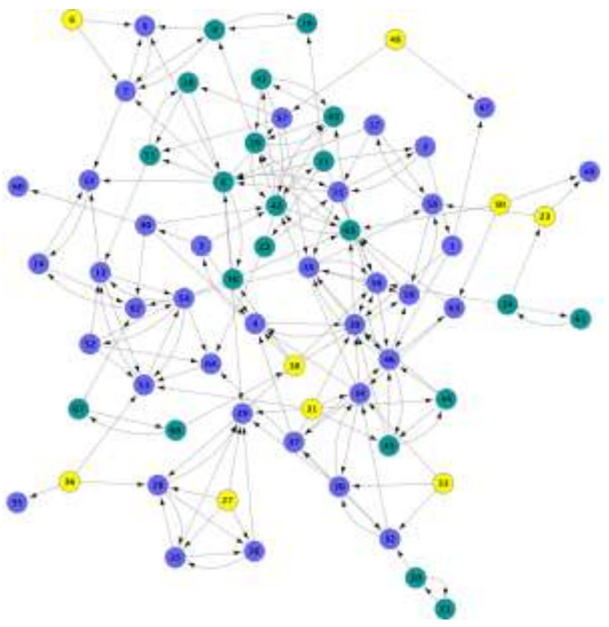
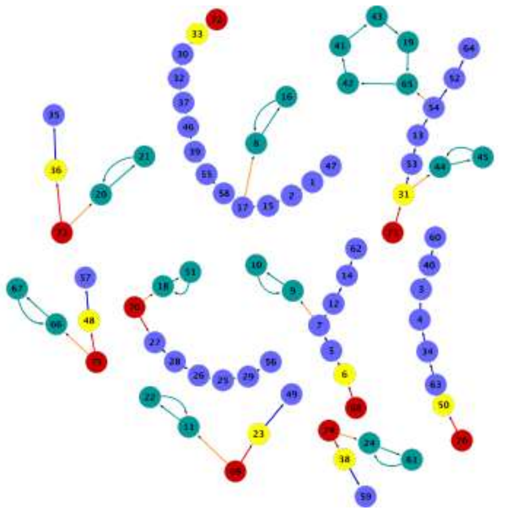


Figure 4.4 Visualizations of the intra-organizational network Consulting. (**a**) The original directed network with 2 driver nodes shown in yellow. (**b**) The spanning cacti of the controllable network. (**c**) Spanning cactus of the one-input controllable network.

(a)



(b)



(c)

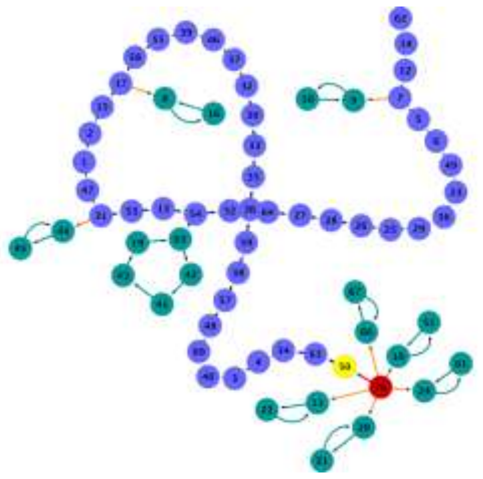
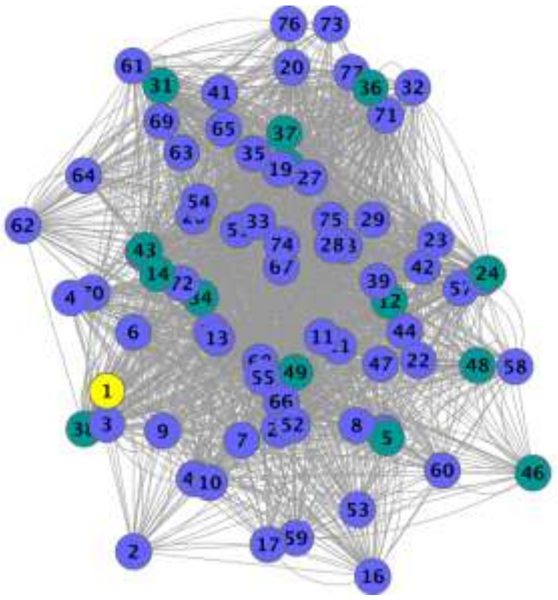


Figure 4.5 Visualizations of the trust network Prison inmate. (**a**) The original network with 9 driver nodes shown in yellow. (**b**) The spanning cacti of the controllable network. The inputs are shown in red. (**c**) Spanning cactus of the one-input controllable network.

(a)



(b)

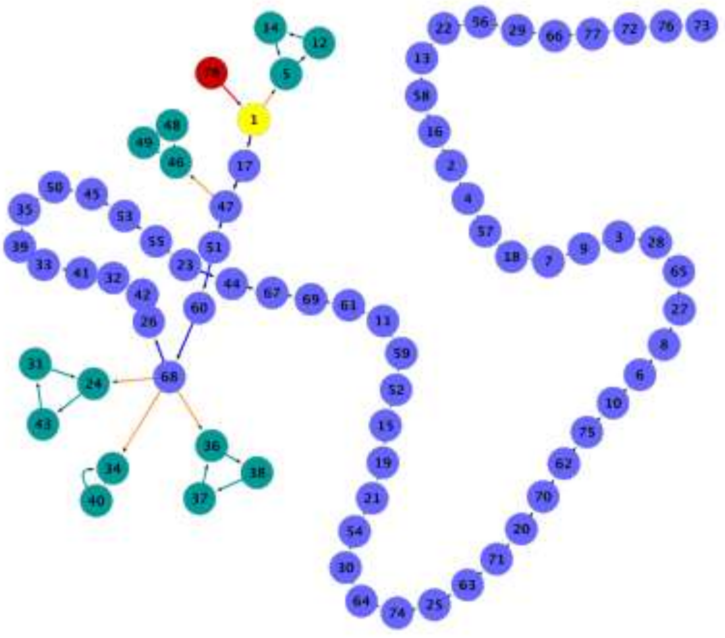
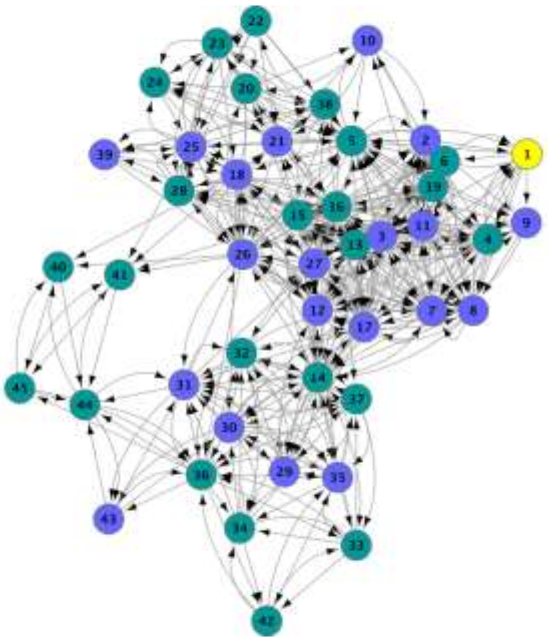


Figure 4.6 Visualizations of the intra-organizational network Manufacturing. (a) The original network with one driver node shown in yellow. **(b)** The minimum spanning cacti for the controllable network. The input node is *78* controls the network and it is the root of the cactus. The original network is also an ***one-input controllable network***.

(a)



(b)

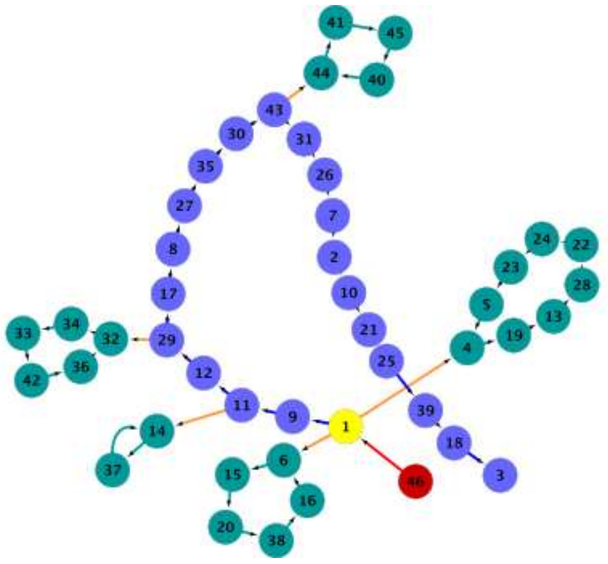


Figure 4.7 Visualizations of the network Macaque brain. (a) The original network with one driver node shown in yellow. **(b)** The minimum spanning cacti for the controllable network. The input node *46* was introduced to control the network and it is the root of the cactus. The original network is also an ***one-input controllable network***.

References

[1] R. E. Kalman, *Mathematical description of linear dynamical systems*. J. Soc. Indus. Appl. Math. Ser. A 1, 152-192(1963).

[2] C. T. Lin, *Structural controllability*. IEEE Trans. Automat. Contr. 19, 201-208 (1974).

[3] E. B. Lee & L. Markus, *Foundations of Optimal Control Theory*. (New York: Wiley, 1967).

[4] Y. Y. Liu, J. J. Slotine & A. L. Barabasi, *Controllability of complex networks*. Nature, 473, 167-173 (2011)

[5] Y. Y. Liu, J. J. Slotine & A. L. Barabasi, *Controllability of complex networks – Supplementary information*. Nature, (2011)

[6] C. Berge. *Two Theorems in Graph Theory*. Proc. Nat. Acad. Sci. U.S.A., 43, 449-844(1957).

[7] C. H. Papadimitriou & K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. (Dover Publications, 1998).

[8] P. Shannon et al. *Cytoscape: a software environment for integrated models of bio-molecular interaction networks*. Genome Research,13 (11):2498-504(2003).

[9] R. Milo et al., *Network Motifs: Simple Building Blocks of Complex Networks.* Science, 298, 824-827 (2002).

[10] L. A. Adamic & N. Glance, The political blogosphere and the 2004 U.S. election, Proceedings of the WWW-2005 Workshop on the Weblogging Ecosystem (2005).

[11] R. Cross & A. Parker, *The Hidden Power of Social Networks*. (Harvard Business School Press,Boston, MA, 2004)

[12] S. Freeman and L. Freeman, Social Science Research Reports 46. University of California, Irvine, CA.(1979)

[13] J. A. Dunne, R. J. Williams, & N. D. Martinez, *Food-web structure and network theory: The role of connectance and size*. PNAS 99, 12917-12922 (2002).

[14] R. R. Christian & J. J. Luczkovich, *Organizing and understanding a winter's seagrass foodweb network through effective trophic levels*. Ecological Modelling 117, 99-124 (1999).

[15] N. Martinez, *Artifacts or attributes? Effects of resolution on the Little Rock Lake food web*. Ecological Monographs 61, 367-392 (1991).

[16] H. Jeong et al., *The large-scale organization of metabolic networks*. Nature 407, 651-654 (2000).

[17] R Miloet al., *Superfamilies of designed and evolved networks.* Science, 303, 1538-42 (2004).

[18] J. Leskovec, K. Lang, A. Dasgupta, & M. Mahoney, *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*. Internet Mathematics, 6(1), 29-123 (2009).

[19] L. Négyessy, T. Nepusz, L. Kocsis and F. Bazsó, *Prediction of the main cortical areas and connections involved in the tactile function of the visual cortex by network analysis*. Eur. J. Neurosci. 23(7):1919-1930 (2006).

[20] Network Dataset http://hal.elte.hu/~enys/data.htm (accessed 24 July 2012).

[21] Anna Palbom. *Complexity of the directed spanning cactus problem*. Discrete Applied Mathematics 146(1): 81-91 (2005).

[22] Arindam Pal, Efficient algorithms for generating all minimum cuts and the cactus representation of a graph, A Project Report, Indian Institute of Science, Bangalore. (2002)

Appendix

Code Listing

File main.cpp

```
/**
   @file       main.cpp
   @author     Ngo Thi Tu Anh, ngotuanh@gmail.com
   @date       July, 2012

   @section    Description
   This program implements the MSC algorithm to find the spanning cacti for a given
directed network. It first reads in a network, transforms it into bipartite
representation, finds the maximum matching in bipartite representation. It then finds
the basic stems and cycles from the maximum matching, and properly connects them
together to form the spanning cacti. It also finds the spanning cactus for the one-
input controllable network. It outputs the cacti, controllable networks, as well as
the node and edge attribute files for cytoscape visualization.

*/
#include <iostream>
#include <sstream>
#include <vector>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <list>
#include <iomanip>
#include <set>
#include <queue>
//my header files:
#include "di_graph.h"
#include "bipartite_graph.h"

using namespace std;

class Cactus{
    public:
    int ControlNode;
    vector<int> Stem;
    vector<vector<int> > Cycles;
    vector<int> CycleControl;
    vector<int> CycleRoot;
    // Constructors
    Cactus(){}
    Cactus(int cNode, vector<int> aStem){
        ControlNode = cNode;
        Stem.assign(aStem.begin(), aStem.end());
    }
    // Utils
    void PrintCactus(){
        cout <<"    Control Node = u"<<ControlNode;
        cout <<"    Stem : ";
        for (int j = 0; j<Stem.size(); ++j)
            cout <<Stem[j]<<" ";
        cout <<endl;
        cout <<"    Number of cycles "<<Cycles.size()<<endl;
        for (int j = 0; j<Cycles.size(); ++j){
            cout <<"        Cycle "<<j+1<<" (connected from node "<<CycleControl[j]<<"
of the stem to node "<<CycleRoot[j]<<" of cycle) : ";
            for (vector<int>::iterator v = Cycles[j].begin(); v != Cycles[j].end();
++v)
                cout <<*v<<" ";
            cout <<endl;
        }
    }//
};

vector<Cactus> Cacti;

// vars for matching algorithm:
vector<int> mate;
vector<int> exposed;
vector<int> label;

vector<vector<int> > Stems;
vector<vector<int> > Cycles;
vector<bool> AddedCycle;

vector<int> MarkCycle;

vector<int> DrNodes;
vector<vector<int> > DistFromDrNode;

void InitEverything(){
    Cacti.clear();
    Stems.clear();
    Cycles.clear();
    AddedCycle.clear();
    MarkCycle.clear();
    DrNodes.clear();
    DistFromDrNode.clear();
}

/**Matching:*/
void InitMaxBPMatch(Bipartite_Graph BG){
    mate.resize(BG.N*2+2,0);
    exposed.resize(BG.N*2+2,0);
    label.resize(BG.N*2+2,0);
}

//recursive retrieve augmenting path and flip
void augment(int v){
    //cout <<" aug "<<v<<endl;
    if (label[v] ==0){
        mate[v] = exposed[v];
        mate[exposed[v]] = v;
    }
    else {
        exposed[label[v]] = mate[v];
        mate[v] = exposed[v];
        mate[exposed[v]] = v;
        augment(label[v]);
    }
}// augment procedure

void MaxBipartiteMatching(Bipartite_Graph BG){
    set<pair<int, int> > A; // set of two different Plus nodes u,v that there is a
Minus node x,
                            // such that: (v, x) is an edge and mate[x] is u
    //init:
    vector<int>::iterator it;
    for (it = BG.Plus.begin();  it != BG.Plus.end(); ++it)
        mate[*it] = 0;
    for (it = BG.Minus.begin(); it != BG.Minus.end(); ++it)
        mate[*it] = 0;

    list<pair<int,int> >::iterator ip;
    while (true){   // loop to find all augmenting paths
        for (vector<int>::iterator v = BG.Plus.begin(); v != BG.Plus.end(); ++v)
             exposed[*v] = 0;
        A.clear();
        // for each edge:
        for (ip = BG.EdgeList.begin(); ip != BG.EdgeList.end(); ++ip){
            int v = ip->first;
            int u = ip->second;
            if (mate[u]==0)
                exposed[v] = u;
            else {
                if (mate[u] != v)
                    A.insert(pair<int,int>(v, mate[u]));
            }
        }// for each edge of BG

        queue<int> Q; //queue for breadth-first-search
        while (!Q.empty()) Q.pop();

        for (vector<int>::iterator v = BG.Plus.begin(); v != BG.Plus.end(); ++v){
            if (mate[*v] == 0){
                Q.push(*v);
                label[*v] = 0;
            };
            label[*v]=0;
        }

        //BFS for find augmenting path
        int found = false;
        int xxx = 0;

        // Check queue:
        while (!Q.empty() ){
            int v = Q.front();
            Q.pop();
            xxx++;
            if (exposed[v] != 0){   // found new augmenting path!
                augment(v);
                found = true;
                break;
            }// found new
            else{
            for (vector<int>::iterator u = BG.Plus.begin(); u != BG.Plus.end(); ++u)
                if ( (label[*u]==0)and(A.find ( pair<int,int>(v,*u) ) != A.end() ) ){
                    label[*u] = v;
                    Q.push(*u);
                }
            }

            if (found) break;// break the BFS
        }// while queue is not empty

        // if not found any augmenting path, then exit:
        if (!found) break;
    }// end of main loop

//--------- result:-----------
    //cout <<endl<<"Matching result: "<<endl;
        int cnt = 0;
        set<int> Remain;
        Remain.insert(BG.Plus.begin(), BG.Plus.end());
        for (int i = 0; i<mate.size(); ++i){
            if ((mate[i]!= 0) and (mate[i]>BG.N)){
                //cout <<setw(4)<<i<<" -- "<<setw(4)<<mate[i] - BG.N<<endl;
                cnt++;
                Remain.erase(mate[i]-BG.N);
            }
        }// for i in mate[]
        for (set<int>::iterator it = Remain.begin(); it != Remain.end(); ++it){
            //cout <<*it<<" ";
            DrNodes.push_back(*it);
        }

// CASE PERFECT MATCHING: no driver node, then choose random node as driver node:
        if (DrNodes.size() <1){
            DrNodes.push_back(BG.Plus[0]);
            //cout<<" Number of driver nodes after modifying: "<<DrNodes.size()<<endl;
        }
}// end of matching

//Procedures for finding cacti:/
void Find_Stems_Cycles(Di_Graph G){
    //cout <<endl<<"Finding stems and cycles... "<<endl;
    vector<bool> Marked;
    Marked.assign(G.N+1, false);
    MarkCycle.assign(G.N+1, -1);

    // Find stems:
    int x;
    Stems.clear();
    Stems.resize(DrNodes.size());
    for (int i = 0; i<DrNodes.size(); ++i){
        x = DrNodes[i];
        while(!Marked[x]){
            Stems[i].push_back(x);
            Marked[x] = true;
            if (mate[x]!=0) x= mate[x] - G.N;
            else break;
        }
    }
    // print stems:
    /**
    for (int i = 0; i<Stems.size(); ++i){
        cout <<"Stem number "<<i+1<<": ";
        for (int j = 0; j<Stems[i].size(); ++j)
            cout <<Stems[i][j]<<" ";
        cout <<endl;
    }
    //*/
    // Find Cycles:
    Cycles.clear();
    vector<int> ACycle;
    int cycCnt = 0;
    for (int u = 1; u <= G.N; ++u)
        if (!Marked[u]){
            //new cycle:
            cycCnt++;
            ACycle.clear();
            int v  = u;
            while (true){
                Marked[v] = true;
                MarkCycle[v] = cycCnt -1;
                ACycle.push_back(v);
                v = mate[v] - G.N;
                if (v==u) break;
            }// end of new cycle;
            Cycles.push_back(ACycle);
        }
    // print Cycles:
    /**
    cout <<endl<<"Number of cycles: "<<Cycles.size()<<endl;
    for (int i = 0; i < Cycles.size(); ++i){
        cout <<"  Cycle "<<i+1<<" : ";
        for (int j = 0; j < Cycles[i].size(); ++j)
            cout <<Cycles[i][j]<<" ";
        cout <<endl;
    }
    //*/
}

void BuildInitialCacti(Di_Graph G){
    // Add each stem to a cactus:
    Cacti.clear();
    for (int i = 0; i < Stems.size(); ++i){
        Cactus ACactus(G.N+1 + i, Stems[i]);
        Cacti.push_back(ACactus);
    }

    // Add distance-1 cycles to a cactus.
    AddedCycle.assign(Cycles.size(),false);
    for (int i = 0; i<Cacti.size(); ++i){// for each cactus i
        for (int j = 0; j< Cacti[i].Stem.size() - 1; ++j){// for each stem node
            int x = Cacti[i].Stem[j];
            for (list<int>::iterator v = G.AdjList[x-1].begin(); v != G.AdjList[x-
1].end(); ++v){//for each node v on adjcent list of x;
                if ((MarkCycle[*v] != -1) and (!AddedCycle[MarkCycle[*v]])){
                    Cacti[i].Cycles.push_back(Cycles[MarkCycle[*v]]);
                    Cacti[i].CycleControl.push_back(x);
                    Cacti[i].CycleRoot.push_back(*v);
                    AddedCycle[MarkCycle[*v]] = true;
                }
            }// for each node v on adjcent list of x;
        } // for each node x on the stem
    } // for each cactus

    // Result:
    /***
    cout <<endl<<"  CACTI "<<endl;
    cout <<"Number of cacti(flural of cactus) in the Cacti "<<Cacti.size()<<endl;
    for (int i = 0; i < Cacti.size(); ++i){
        cout <<"Cactus "<<i+1<<" :"<<endl;
        Cacti[i].PrintCactus();
        cout <<endl;
    }
    */
}

vector<int> DistanceFromNode(Di_Graph G, int v){    //BFS
    vector<int> Dist;
    Dist.assign(G.N+1, -1);
    vector<bool> Marked;
    Marked.assign(G.N+1, false);
    queue<int> Q;
    Dist[v] = 0;
    Marked[v] = true;
    Q.push(v);
    while (!Q.empty()){
        int u = Q.front();
        Q.pop();
        for (list<int>::iterator x = G.UAdjList[u-1].begin(); x != G.UAdjList[u-
1].end(); ++x)
            if (!Marked[*x]){
                Dist[*x] = Dist[u] + 1;
                Marked[*x] = true;
                Q.push(*x);
            }
    }
    return Dist;
}


void CalcDistanceFromDrNode(Di_Graph G){
    DistFromDrNode.resize(DrNodes.size());
    vector<int> ADist;
    for (int i = 0; i<DrNodes.size(); ++i){
        int v = DrNodes[i];
        ADist = DistanceFromNode(G, v);
        DistFromDrNode[i] = ADist;
    }
}// end calculation Distance from Driver Nodes;

//complete building cacti:
void AddCylcesToCacti(Di_Graph G){
    for (int i = 0; i<Cycles.size(); ++i){
        if (!AddedCycle[i]){
            // find nearest driver node u:
            int minDist = G.N+1;
            int minU=-1, minRoot = Cycles[i][0];
            for (int u = 0; u < DrNodes.size(); ++u){
                for (vector<int>::iterator v = Cycles[i].begin(); v !=
Cycles[i].end(); ++v){
                    if (DistFromDrNode[u][*v] >-1)
                    if (minDist > DistFromDrNode[u][*v]) {
                        minDist = DistFromDrNode[u][*v];
                        minU = u;
                        minRoot = *v;
                    }
                }
            }

            if (minU = -1) {// find smallest cacti so far:
                int minNumOfCircles = G.N+1;
                minU = 0;
                for (int u = 0; u < DrNodes.size(); ++u){
                    if (minNumOfCircles > Cacti[u].Cycles.size()){
                        minNumOfCircles = Cacti[u].Cycles.size();
                        minU = u;
                    }
                }
            }
            // Add this cycle to the cactus of the nearest driver node just found:
            Cacti[minU].Cycles.push_back(Cycles[i]);
            Cacti[minU].CycleControl.push_back(Cacti[minU].ControlNode);
            Cacti[minU].CycleRoot.push_back(minRoot);
        }// still not added
    }// for each Cycle
}// end procedure

//Merge two cactuses:
Cactus MergingTwoCactuses(Cactus C1, Cactus C2){
    Cactus C;
    //merging stem:
    C.Stem.clear();
    for (vector<int>::iterator it = C2.Stem.begin(); it != C2.Stem.end(); ++it )
        C.Stem.push_back(*it);
    for (vector<int>::iterator it = C1.Stem.begin(); it != C1.Stem.end(); ++it )
        C.Stem.push_back(*it);
    //merging cycles:
    C.Cycles.clear();
    C.CycleControl.clear();
    C.CycleControl.clear();
    for (int i = 0; i < C2.Cycles.size(); ++i){
        C.Cycles.push_back(C2.Cycles[i]);
        C.CycleControl.push_back (C2.CycleControl[i]);
        C.CycleRoot.push_back(C2.CycleRoot[i]);
    };
    for (int i = 0; i < C1.Cycles.size(); ++i){
        C.Cycles.push_back(C1.Cycles[i]);
        C.CycleControl.push_back (C1.CycleControl[i]);
        C.CycleRoot.push_back(C1.CycleRoot[i]);
    };
    //merging control node:
    C.ControlNode = C2.ControlNode;

    //modify cyclecontrol node:
    for (int i = 0; i<C.CycleControl.size(); ++i){
        if (C.CycleControl[i] == C1.ControlNode)
       C.CycleControl[i] = C.ControlNode;
    }

    return C;
}

//Find cactus for one-input controllable network:
void FindCactusForOneInputControllable (vector<Cactus> *newCacti){
    newCacti->clear();
    Cactus C2 = Cacti[Cacti.size()-1];
    Cactus C = C2;
    for (int i = Cacti.size()-2; i >=0; --i){
        Cactus C1 = Cacti[i];
        C = MergingTwoCactuses(C1,C2);
        C2 = C;
    }
    newCacti->push_back(C2);
}

// print statistics of the cacti
void PrintStatistics(Di_Graph G){
    cout <<"=====Input:====="<<endl;
    cout <<"    N = "<<G.N<<endl;
    cout <<"    E = "<<G.E<<endl;
    cout <<"=====Statistic of minimum cacti====="<<endl;
    cout <<"    Number of stems "<<Cacti.size()<<endl;
    int nCycles = 0;
    for (int i = 0; i <Cacti.size(); ++i){
        nCycles +=Cacti[i].Cycles.size();
    }
    cout <<"    Number of buds "<<nCycles<<endl;
    double averDegree = 0;
    for (int i = 0; i <Cacti.size(); ++i){
        averDegree += G.getDegree(Cacti[i].Stem[0]);
    }
    averDegree = averDegree / Cacti.size();
    cout <<"    Average degree of driver nodes = "<<averDegree<<endl;
    cout <<"    average degree of the input network = "<<G.E / G.N<<endl;
}

// Print cacti into SIF format for Cytoscape:
int Cacti2SIF(vector<Cactus> Cacti, string fn){
    ofstream MyFile(fn.c_str());
    if (!MyFile.is_open()){
        cerr <<"Cannot open this file "<<fn<<endl;
        return -1;
    }
    cout <<"Write cactus graph into .SIF file: "<<fn<<endl;
    for (int i = 0; i <Cacti.size(); ++i){
        Cactus C = Cacti[i];
        //Print stem:
            MyFile<<C.ControlNode<<" controls "<<C.Stem[0]<<endl;
        for (int j = 0; j < C.Stem.size()-1; ++j)
            MyFile << C.Stem[j]<<" stem "<<C.Stem[j+1] <<endl;
        // Print cycles:
        for (int c = 0; c < C.Cycles.size(); ++c){
            // stem to cycle edge:
            MyFile <<C.CycleControl[c]<<" stem_to_cycle "<<C.CycleRoot[c]<<endl;
            //cycle:
            for (int j = 0; j <C.Cycles[c].size()-1; ++j)
                MyFile << C.Cycles[c][j]<<" cycle "<<C.Cycles[c][j+1]<<endl;
            MyFile<<C.Cycles[c][C.Cycles[c].size()-1] <<" cycle
"<<C.Cycles[c][0]<<endl;
        }// all cycle
    }// all cacti

    MyFile.close();
    cout <<"    Done!"<<endl;
}

int Cacti2NOA(vector<Cactus> Cacti, string fn){  // node attributes
    ofstream MyFile(fn.c_str());
    if (!MyFile.is_open()){
        cerr <<"Cannot open this file "<<fn<<endl;
        return -1;
    }
    cout <<"Write cactus graph node attribute file: "<<fn<<endl;
    MyFile <<"NodeType (class = java.lang.String)"<<endl;
    for (int i = 0; i <Cacti.size(); ++i){
        Cactus C = Cacti[i];
        //Print stem:
        MyFile<<C.ControlNode<<" =  CONTROL_NODE"<<endl;
        MyFile << C.Stem[0]<<" = driver_node "<<endl;
        for (int j = 1; j < C.Stem.size(); ++j)
            MyFile << C.Stem[j]<<" = stem_node "<<endl;
        // Print cycles:
        for (int c = 0; c < C.Cycles.size(); ++c){
            //cycle:
            for (int j = 0; j <C.Cycles[c].size(); ++j)
                MyFile << C.Cycles[c][j]<<" = cycle_node "<<endl;
        }// all cycle
    }// all cacti

    MyFile.close();
    cout <<"    Done!"<<endl;
}

void PrintRes(int N, int E, int Nbuds, int Nstems, int times, double averDegreeDr,
double averDegreeNW){
    string fn;
    stringstream SS;
    SS <<N<<"_"<<E<<".txt";
    SS>>fn;

    ofstream MyFile(fn.c_str());
    if (!MyFile.is_open()){
        cerr <<"Cannot open this file "<<fn<<endl;
        return;
    }

    MyFile<<"number of repeating times "<<times<<endl;
    MyFile<<"total number of stems "<<Nstems<<endl;
    MyFile<<"total number of buds " <<Nbuds<<endl;
    MyFile<<"average degree of driver nodes " <<(averDegreeDr+0.0)/(times+0.0)<<endl;
    MyFile<<"average degree of all nodes" <<(averDegreeNW+0.0)/(times+0.0)<<endl;

    MyFile.close();

}

void PrintRes(int N, double P, int Nbuds, int Nstems, int times, double averDegreeDr,
double averDegreeNW){
    string fn;
    stringstream SS;
    SS <<N<<"_"<<P<<".txt";
    SS>>fn;

    ofstream MyFile(fn.c_str());
    if (!MyFile.is_open()){
        cerr <<"Cannot open this file "<<fn<<endl;
        return;
    }

    MyFile<<"number of repeating times "<<times<<endl;
    MyFile<<"total number of stems "<<Nstems<<endl;
    MyFile<<"total number of buds " <<Nbuds<<endl;
    MyFile<<"average degree of driver nodes " <<(averDegreeDr+0.0)/(times+0.0)<<endl;
    MyFile<<"average degree of all nodes" <<(averDegreeNW+0.0)/(times+0.0)<<endl;

    MyFile.close();

}


int main()//int argc, char  *argv[])
{
/***For running on specific networks:*/
string pwd = "data/";
vector <string> fileNames;
fileNames.push_back(pwd+ "brain_macaque.ncol");
fileNames.push_back(pwd+ "org_Enron_core.ncol");
fileNames.push_back(pwd+ "food_Seagrass.ncol");
fileNames.push_back(pwd+ "food_LittleRock.ncol");
fileNames.push_back(pwd+ "food_Grassland.ncol");
fileNames.push_back(pwd+ "circuit_s208a.ncol");
fileNames.push_back(pwd+ "circuit_s420a.ncol");
fileNames.push_back(pwd+ "trust_CollegeStudents_rd.ncol");
fileNames.push_back(pwd+ "trust_PrisonInmates_rd.ncol");
fileNames.push_back(pwd+ "org_Consulting.ncol");
fileNames.push_back(pwd+ "org_Freemans_2.ncol");
fileNames.push_back(pwd+ "org_Freemans_1.ncol"); // different data
fileNames.push_back(pwd+ "org_Manufacturing.ncol");
fileNames.push_back(pwd+ "metabolic_SCerevisiae.ncol");
fileNames.push_back(pwd+ "metabolic_CElegans.ncol");
fileNames.push_back(pwd+ "metabolic_EColi.ncol");
fileNames.push_back(pwd+ "regulatory_TRN_Yeast_2.ncol");
fileNames.push_back(pwd+ "www_Polblogs_rd.ncol");

//Run for each input file:
for (vector<string>::iterator fn = fileNames.begin(); fn != fileNames.end(); ++fn){
    cerr <<endl<<(int)(fn - fileNames.begin() ) + 1 <<" Running for "<<*fn<<endl;
    InitEverything();
    Di_Graph aG;
    aG.ReadEdgeFile(*fn,"0"); // note: vertices are from 1 instead of 0;
    aG.ExpandUndirected();
    Bipartite_Graph myBG(aG);

    cerr <<"=====MATCHING...====="<<endl;
    InitMaxBPMatch(myBG);
    MaxBipartiteMatching(myBG);
    cerr <<"=====MATCHING... DONE!====="<<endl<<endl;

    cerr <<"=====STEM and CYCLE FINDING...====="<<endl;
    Find_Stems_Cycles(aG);
    cerr <<"=====STEM and CYCLE FINDING... DONE====="<<endl<<endl;

    cerr <<"=====Initial Cacti BUILDING...====="<<endl;
    BuildInitialCacti(aG);
    cerr <<"=====Initial Cacti BUILDING... DONE====="<<endl<<endl;

    cerr <<"=====Distance from Driver Nodes: CALCULATING...====="<<endl;
    CalcDistanceFromDrNode(aG);
    cerr <<"=====Distance from DrNode CALCULATING...DONE====="<<endl<<endl;

    cerr <<"=====ADDING Cycles to Cacti...====="<<endl;
    AddCylcesToCacti(aG);
    cerr <<"=====ADDING Cycles to Cacti...DONE====="<<endl<<endl;

//PRINT OUT RESULT:
    cout <<endl<<"=====RESULT: the CACTI ====="<<endl;
    cout <<"Number of cacti(flural of cactus) in the Cacti "<<Cacti.size()<<endl;
    for (int i = 0; i < Cacti.size(); ++i){
        cout <<"Cactus "<<i+1<<" :"<<endl;
        Cacti[i].PrintCactus();
        cout <<endl;
    };
//Find spanning cactus for one-input controllable
    vector<Cactus> myCacti;
    FindCactusForOneInputControllable(&myCacti);

// Write cytoscape attribute files:
    Cacti2SIF(Cacti, *fn+".sif");     Cacti2NOA(Cacti, *fn+".noa");
Cacti2SIF(myCacti, *fn+".one-input.sif");     Cacti2NOA(myCacti, *fn+".one-
input.noa");
}
 return 0;
//***/

/***For random networks*/

int  NN[5] = {10,50,100,200,500};
int  EE[4] = {2,5,10,20};
double PP[4] = {0.01,0.02,0.05,0.1};

for (int ii = 0; ii<5; ++ii)
for (int jj = 0; jj<4; ++jj)
{//begin
    cout <<" ii = "<<ii<<" jj= "<<jj<<endl;
    int Nbuds = 0, Nstems = 0, times = 100;
    double AllaverDegree = 0, AllaverDegreeNW = 0;

    for (int i = 0; i<times; ++i){
        InitEverything();
        Di_Graph newG;
        //aG.RandomGenerator1(NN[ii],NN[ii]*EE[jj]);    //type 1
        newG.RandomGenerator(NN[ii],PP[jj]);                 //type 2

        //=====Core procedure:
        newG.ExpandUndirected();      // Clone 1 un-directed graph NEEDED!
        //aG.PrintUndirectedGraph();
        Bipartite_Graph myNewBG(newG);
        //cout <<" print bi-graph: "<<endl;
        //myBG.PrintGraph();

        cerr <<"=====MATCHING...====="<<endl;
        InitMaxBPMatch(myNewBG);
        MaxBipartiteMatching(myNewBG);
        cerr <<"=====MATCHING... DONE!====="<<endl<<endl;
        cerr <<"=====STEM and CYCLE FINDING...====="<<endl;
        Find_Stems_Cycles(newG);
        cerr <<"=====STEM and CYCLE FINDING... DONE====="<<endl<<endl;
        cerr <<"=====Initial Cacti BUILDING...====="<<endl;
        BuildInitialCacti(newG);
        cerr <<"=====Initial Cacti BUILDING... DONE====="<<endl<<endl;
        cerr <<"=====Distance from Driver Nodes: CALCULATING...====="<<endl;
        CalcDistanceFromDrNode(newG);

        cerr <<"=====Distance from DrNode CALCULATING...DONE====="<<endl<<endl;
        cerr <<"=====ADDING Cycles to Cacti...====="<<endl;
        AddCylcesToCacti(newG);
        cerr <<"=====ADDING Cycles to Cacti...DONE====="<<endl<<endl;
        //=====End of core procedures

        //PRINT OUT:
        //PrintStatistics(aG);
        Nstems += Cacti.size();
        int nCycles = 0;
        for (int x = 0; x <Cacti.size(); ++x){
            nCycles +=Cacti[x].Cycles.size();
        }

        Nbuds += nCycles;
        double averDegree = 0;
        for (int x = 0; x <Cacti.size(); ++x){
            averDegree += newG.getDegree(Cacti[x].Stem[0])+0.0;
        }
        averDegree = (averDegree+0.0) / (Cacti.size()+0.0);
        AllaverDegree += averDegree;

        AllaverDegreeNW += (newG.E+0.0)/(newG.N+0.0);
    }// for repeating time;

    //PrintRes(NN[ii],NN[ii]*EE[jj],Nbuds, Nstems, times, AllaverDegree,
AllaverDegreeNW); // type 1
    PrintRes(NN[ii],PP[jj],Nbuds, Nstems, times, AllaverDegree, AllaverDegreeNW);
//type 2

}//end; of NN EE
return 0;
//***/

}// end of main
```

File di_graph.h

```
/**
   @file        di_grahp.h
   @author      Ngo Thi Tu Anh, ngotuanh@gmail.com
   @date        July, 2012

   @section     Description
   Header file for the Di_Graph class.

*/
#ifndef DI_GRAPH_H
#define DI_GRAPH_H

#include <list>
#include <vector>
#include <set>
#include <string>

using namespace std;

class Di_Graph
{
    public:
        int N;
        // nodes start from 1;
        vector <list<int> > AdjList;// Adjacent list of a node u is AdjList[u-1];
        vector <list<int> > UAdjList;// Adjacent list of a node u is UAdjList[u-1];
        set<pair<int, int> > EdgeSet;
        int E;

        //constructors
        Di_Graph();
        Di_Graph(string);

        // Utility Functions:
        int getDegree(int anode);
        void ReadFile(string); // Read from file
        void ReadEdgeFile(string fn);    // read from list of edge file
        void ReadEdgeFile(string fn, string type);    // node start from 0
        void PrintGraph();
        void ExpandUndirected();
        void PrintUndirectedGraph();
        void RandomGenerator(int, int);
        void RandomGenerator1(int N, int E);
        void RandomGenerator(int, double);
        void WriteFile(string);

    protected:
    private:
        list<int>       LineParserList(string);
        vector<int>     LineParserVector(string);
        void            CreateFromEdgeSet();
};

#endif // DI_GRAPH_H
```

File di_graph.cpp

```
/**
   @file       di_graph.cpp
   @author     Ngo Thi Tu Anh, ngotuanh@gmail.com
   @date       July, 2012

   @section    Description
   Implementation of Di_Graph class.

*/

#include "di_graph.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <time.h>
#include <set>

using namespace std;

//Contructors:
Di_Graph::Di_Graph(){// empty
}

/* Constructor: Init graph from file*/
Di_Graph::Di_Graph(string fn){
    this->ReadFile(fn);
}

/*  Utility Functions: */
int Di_Graph::getDegree(int anode){
    return AdjList[anode-1].size();
}

void Di_Graph::ReadFile(string fn){
    int n=1,e=0;
    vector <list<int> > AdjList;
    string  st;
    ifstream MyFile (fn.c_str());
    list<int> AList;
    if (MyFile.is_open())
        {   getline(MyFile,st);
            n = atoi(st.c_str());
            for (int i = 0; i<n; ++i){ getline(MyFile, st);
                AList = LineParserList(st);
                AdjList.push_back(AList);
                e += AList.size();
            }
        }
    this->N = n;
    this->AdjList = AdjList;
    this->E = e;
    this->ExpandUndirected();
}

void Di_Graph::ReadEdgeFile(string fn){
    string  st;
    ifstream MyFile (fn.c_str());
    if (!MyFile){
        std::cerr << "Error: File "<<fn<<" could not be opened! "<<endl;
        exit(1);
    }
    vector<int> APair;
    while (getline(MyFile,st)){
        APair = LineParserVector(st);
        EdgeSet.insert(pair<int,int>(APair[0], APair[1]));
    }// done reading file
    cerr << " Read "<<EdgeSet.size()<<" edges from file "<<fn<<endl;

    CreateFromEdgeSet();
}
```

```
void Di_Graph::ReadEdgeFile(string fn, string type){
    string  st;
    ifstream MyFile (fn.c_str());
    if (!MyFile){
        std::cerr << "Error: File "<<fn<<" could not be opened! "<<endl;
        exit(1);
    }
    vector<int> APair;
    while (getline(MyFile,st)){
        APair = LineParserVector(st);
        EdgeSet.insert(pair<int,int>(APair[0] +1, APair[1] +1));
    }// done reading file
    cerr << " Read "<<EdgeSet.size()<<" edges from file "<<fn<<endl;

    CreateFromEdgeSet();
}

void Di_Graph::WriteFile(string fn){
    ofstream MyFile(fn.c_str());
    if (MyFile.is_open()){
        MyFile <<N<<endl;
        for (int i = 0; i<N; ++i){
        if (AdjList[i].size() >0){
        for (list<int>::iterator j = AdjList[i].begin(); j != AdjList[i].end(); ++j)
            MyFile << *j<<" ";
        }
        else MyFile <<0;
            MyFile <<endl;
        }
    }
    MyFile.close();
}

void Di_Graph::ExpandUndirected(){
        UAdjList.clear();
        UAdjList.resize(N);
        for (int i = 0; i<AdjList.size(); ++i){
        for (list<int>::iterator v = AdjList[i].begin(); v != AdjList[i].end(); ++v)
            if (*v != 0){
                UAdjList[i].push_back(*v);
                UAdjList[*v-1].push_back(i+1);
            }
            UAdjList[i].unique();
        }
}// expand to undirected graph

void Di_Graph::PrintGraph(){
    cout <<" Directed Graph:"<<endl;
    cout <<" Number of nodes: "<<N<<endl;
    cout <<" Number of edges: "<<E<<endl;
    cout <<" Adjcent List:"<<endl;
    int i, j;
    for (i = 0; i< AdjList.size(); ++i){
        cout <<"list "<<i+1<<": ";
        for (list<int>::iterator v = AdjList[i].begin(); v != AdjList[i].end(); ++v)
            cout <<*v<<" ";
         cout <<endl;
    }
}

void Di_Graph::PrintUndirectedGraph(){
        cout <<" Un-Directed Graph:"<<endl;
        cout <<" Number of nodes: "<<N<<endl;
        cout <<" Adjcent List:"<<endl;
        int i, j;
        for (i = 0; i< UAdjList.size(); ++i){
            cout <<"list "<<i+1<<": ";
            for (list<int>::iterator v = UAdjList[i].begin(); v != UAdjList[i].end();
++v)         cout <<*v<<" ";
             cout <<endl;
        }
}
```

```
// Randomly generating graph with number of nodes and average outdegree
void Di_Graph::RandomGenerator(int NodeCnt, int AverOutDegree){
    this->N = NodeCnt;
    if (AverOutDegree >=N) AverOutDegree = N-1;
    this->E = NodeCnt * AverOutDegree;
    int cnt = 0;
    this->AdjList.resize(NodeCnt);
    int u, v;
    set <pair<int, int> > EdgeSet;
    srand(time(NULL));

    while (cnt < this->E){
        u = rand()%N;
        v = rand()%N;
        if (u != v)
        if (EdgeSet.find(pair<int,int>(u,v)) == EdgeSet.end()){
            ++cnt;
            AdjList[u].push_back(v+1);
            EdgeSet.insert(pair<int,int>(u,v));
        }
    }// while
    this->EdgeSet = EdgeSet;
    this->ExpandUndirected();
}


void Di_Graph::RandomGenerator1(int N, int E){
    if (E > N*(N-1)) {
        cerr<<"Can not generate more than "<<N*(N-1)<<" edges"<<endl;
        cerr<<"Set number of edges = "<< N*(N-1)<<endl;
        E = N*(N-1);
    }

    this->N = N;
    this->E = E;
    this->AdjList.clear();
    this->AdjList.resize(N);

    int u, v;
    set <pair<int, int> > EdgeSet;
    srand(time(NULL));

   for(int i = 0; i<E; ++i){
        u = rand()%N;
        v = rand()%N;
        if (u != v)
        if (EdgeSet.find(pair<int,int>(u,v)) == EdgeSet.end()){
            AdjList[u].push_back(v+1);
            EdgeSet.insert(pair<int,int>(u,v));
        }
    }
    this->EdgeSet = EdgeSet;
    this->ExpandUndirected();
}

//Generator type 2:
void Di_Graph::RandomGenerator(int NodeCnt, double p){
    this->N = NodeCnt;
    this->AdjList.resize(NodeCnt);

    int Ecnt = 0;
    srand(time(NULL));

    for (int u = 0; u<N; ++u)
    for (int v = 0; v<N; ++v)
        if (u != v)
            if (rand()%100 < p*100){// there is an directed edge from u to v:
               AdjList[u].push_back(v+1);
               Ecnt++;
            }
    this->E = Ecnt;
    this->ExpandUndirected();
}
```

```
// Private utility:
list<int> Di_Graph::LineParserList(string st){
    //1 3 6 7
    istringstream MyISS(st);
    int node;
    list <int> AVec;
    while(MyISS >> node){
        AVec.push_back(node);
    }
    return AVec;
}
// Private utility:
vector<int> Di_Graph::LineParserVector(string st){
    //1 3 6 7
    istringstream MyISS(st);
    int node;
    vector <int> AVec;
    while(MyISS >> node){
        AVec.push_back(node);
    }
    return AVec;
}

void Di_Graph::CreateFromEdgeSet(){
    set<int> NodeSet;
    //get nodes:
    for (set<pair<int, int> >::iterator p = EdgeSet.begin(); p != EdgeSet.end(); ++p){
        NodeSet.insert((*p).first);
        NodeSet.insert((*p).second);
    }
    // create adjlist:
    AdjList.clear();
    AdjList.resize(NodeSet.size());
    for (set<pair<int, int> >::iterator p = EdgeSet.begin(); p != EdgeSet.end(); ++p){
        int u = (*p).first - 1;
        int v = (*p).second;
        AdjList[u].push_back(v);
    }
    this->N = NodeSet.size();
    this->E = EdgeSet.size();
    this->ExpandUndirected();

    cerr << "#Edges "<< E<<endl;
}
```

File bipartite_graph.h

```
/**
   @file        bipartite_graph.h
   @author      Ngo Thi Tu Anh, ngotuanh@gmail.com
   @date        July, 2012

   @section     Description
   Header file for Bipartite_Graph class.

*/

#ifndef BIPARTITE_GRAPH_H
#define BIPARTITE_GRAPH_H

#include "di_graph.h"
#include <list>
#include <vector>

const int PLUS  = +1;
const int MINUS = -1;

class Bipartite_Graph{
    public:
        int N;
        vector<int> Plus;
        vector<int> Minus;
        vector<vector<int> > AdjList;
        list<pair<int, int> > EdgeList;
        // constructors:
        Bipartite_Graph();
        Bipartite_Graph(Di_Graph);
        // Utility Functions:
        void PrintAdjList(vector<vector<int> > , int );
        void PrintGraph();

    protected:
    private:
};

#endif // BIPARTITE_GRAPH_H
```

File bipartite_graph.cpp

```
/**
   @file        bipartite_graph.cpp
   @author      Ngo Thi Tu Anh, ngotuanh@gmail.com
   @date        July, 2012

   @section     Description
   Implementation of Bipartite_Graph class.

*/

#include "bipartite_graph.h"
#include <iostream>
#include <iomanip>

// Constructors:
Bipartite_Graph::Bipartite_Graph(){    //ctor
}

Bipartite_Graph::Bipartite_Graph(Di_Graph g){
    N = g.N;
    for (int i = 1; i<=N; ++i){
        Plus.push_back(i);
        Minus.push_back(i + N);
    }
    AdjList.resize(N+N+2);
    // create Edges from adjcent list
    int i;
    list<int>::iterator j;
    int u,v;
    for (i = 0; i<g.AdjList.size(); ++i){
        u = i+1;
        for (j = g.AdjList[i].begin(); j!= g.AdjList[i].end(); ++j){
            if (*j != 0){
                v = g.N + (*j);
                this->AdjList[u].push_back(v);
                this->AdjList[v].push_back(u);
                this->EdgeList.push_back (pair<int,int>(u,v));
                this->EdgeList.push_back (pair<int,int>(v,u));
            }
        }
    }
}// construct from a directed graph

// Utility Functions
void Bipartite_Graph::PrintAdjList (vector<vector<int> > L, int side){
        int i;
        vector<int>::iterator j;
        int st=1, ed=N;
        if (side == MINUS){
            st = N+1; ed = 2*N;
        }
        for (i = st; i<=ed; ++i){
            cout <<"adj of "<<i<<": ";
            for (j = L[i].begin(); j != L[i].end(); ++j){
                    cout <<*j<<" ";
            }
            cout <<endl;
        }
}

void Bipartite_Graph::PrintGraph(){
    cout <<" Bipartite Graph:"<<endl;
    cout <<" Plus nodes: ";
    for (vector<int>::iterator it=Plus.begin();it!=Plus.end();++it) cout <<*it<<" ";
    cout <<endl;
    cout <<" Adjcent List of Plus Nodes: "<<endl;
    PrintAdjList(AdjList, PLUS);
    cout <<" Minus nodes: ";
    for (vector<int>::iterator it=Minus.begin();it!=Minus.end();++it) cout <<*it<<" ";
    cout <<endl;
    cout <<" Adjcent List of Minus Nodes: "<<endl;
    PrintAdjList(AdjList, MINUS);
    cout <<endl<<"List of edges: "<<EdgeList.size()<<" edges"<<endl;
    list<pair<int,int> >::iterator it;
    for (it = EdgeList.begin(); it != EdgeList.end(); ++it){
        cout <<setw(4)<<it->first<<" -- "<<setw(4)<<it->second<<endl;
    }
}
```