DETERMINISTIC GREEDY ALGORITHMS FOR OPTIMAL SENSOR PLACEMENT

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Venkata Santosh Chintamaneni

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

March 2012

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Deterministic Greedy Algorithms for Optimal Sensor Placement

By

Venkata Santosh Chintamaneni

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Kendall Nygard
Chair

Dr. Simone Ludwig

Dr. Saeed Salem

Dr. Chao You

Approved:

03/08/2012
Date

Dr. Kendall Nygard
Department Chair

ABSTRACT

A sensor is a device which can be sensitive to any physical stimulus, such as light, heat, or a particular motion, and responds with a respective impulse. A graph is an abstract representation for a set of objects of any kind where some object pairs are connected by links. The interconnected objects are called vertices or nodes.

In this paper, one of the antiquated and classical examples of a Non-Deterministic Polynomial hard (NP-hard) optimization problem in computer science, the "Set Coverage Problem," is discussed. Monitoring spatial constraints in real-world networks with huge data requires the best sensor placement for a given network. For any typical network, it has been proven that a greedy placement algorithm achieves a minimum of 63% optimal reduction in total variance. The focus in this paper is on sensor-optimization problem with the help of two approximation algorithms inherited from traditional greedy approach with supporting experimental results.

ACKNOWLEDGEMENTS

I would take this opportunity to thank my adviser, Dr. Kendall Nygard, who has given me valuable support, encouragement and advice without which this work would not have been completed. I am thankful to members of my committee, Dr. Simone Ludwig, Dr. Saeed Salem, and Dr. Chao You, for their support.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES vii

1. INTRODUCTION 1

2. LITERATURE REVIEW 4

3. PROBLEM FORMULATION 8

3.1. Problem Definition 8

4. PROPOSED SOLUTION 10

4.1. Graph Generation 10

4.1.1. Random Graph Algorithm with Distance Parameter 11

4.1.2. Random Graph Approach with Distance and Cost Parameters 14

4.1.3. Pure Random Network 14

4.1.4. Load Graph 16

4.2. Deterministic Greedy Heuristic 16

4.2.1. Description 16

4.2.2. Explanation of DG Heuristic 18

4.3. Deterministic Greedy Heuristic with Look Ahead 23

4.3.1. Description 23

4.3.2. DG Heuristic with Look-Ahead Node Explanation 26

4.4. AMPL 34

4.4.1. Graphical User Interface (GUI) 34
4.4.2. The AMPL Language Processor 35
4.4.3. AMPL-Plus Solvers 35
4.4.4. AMPL Working Procedure 35
5. EXPERIMENTAL RESULTS 39
5.1. Test Cases 39
5.1.1. Test Case 1 39
5.1.2. Test Case 2 42
5.1.3. Test Case 3 44
5.1.4. Test Case 4 47
5.1.5. Test Case 5 49
5.1.6. Test Case 6 52
5.1.7. Test Case 7 54
5.1.8. Test Case 8 57
5.1.9. Test Case 9 59
5.1.10. Test Case 10 61
5.1.11. Test Case 11 64
5.1.12. Test Case 12 66
5.2. Conclusion and Future Work 69
REFERENCES 70

LIST OF FIGURES

Figure Page

1.1. A Seven-node power system network..... 2

3.1. Visibility matrix of the power system with PMU placed on vertex V_4. 8

4.1. Sample real time network with three substations. 11

4.2. A 3 x 3 matrix representing the above graph. 12

4.3. A network graph generated by Random Graph Approach (4.1.1) representing a network of 15 nodes. 13

4.4. An initial three-node network graph with no connection between the nodes. 15

4.5. The final network graph obtained after implementing the Pure Random Network approach. 15

4.6. A screenshot of the input box allowing a user to enter the network size..... 18

4.7. A screenshot of the input box allowing the user to enter the acceptable distance between any two nodes in the network. 18

4.8. The resulting seven-node graph generated from the user inputs. 19

4.9. The matrix representation of the graph generated from the user input. 19

4.10. A code snippet representing the network coverage. 20

4.11. The matrix representation of the graph after placing sensor a node 3. 21

4.12. The resultant matrix obtained after performing the sequence of operations in Step 4. 22

4.13. The final matrix indicating that the graph is observable. 23

4.14. A sample graph used to help identifying the concept of Look-Ahead nodes. 25

4.15. Screenshots of the input prompts allowing the user to enter the number of nodes and the acceptable distance between two nodes of a network. 27

4.16. A random seven-node graph generated from user input. 27

4.17. Corresponding matrix representation for the generated graph in Figure 4.16. 28

4.18. A code snippet for determining the coverage of a graph. 29

4.19. The matrix representation of a network after applying a sensor on node 2. 30

4.20. A code snippet for applying a sensor on any given node. 31

4.21. Matrix representation of the network after computing all neighboring nodes for node 3. 32

4.22. Matrix representation of the network after applying a sensor on node 3. 32

4.23. Matrix representation of the network after applying sensors on the Maximum-Weight node (Node 2) and the Look-Ahead node (Node 6). 33

4.24. A screenshot depicting the MDI window user interface. 34

4.25. A screenshot representing the procedure to load a model file in AMPL. 37

4.26. A screenshot representing the parameters required to provide a network matrix to AMPL. 38

4.27. A screenshot depicting the optimal solution generated by the CPLEX solver after clicking the solve button. 38

5.1. A Random Five-Node Network. 40

5.2. Graph comparing the total number of sensors for all three approaches (for a five-node graph). 41

5.3. Graph comparing the execution time for all three approaches (for a 25-node graph). 41

5.4. A Random 10-Node Network. 42

5.5. Graph comparing the total number of sensors for all three approaches (for a 10 node graph). 43

5.6. Graph comparing the execution time for all three approaches (for a 10-node graph). 44

5.7. A Random 15-Node Network. 45

5.8. Graph comparing the total number of sensors for all three approaches (for a 15-node graph). 46

5.9. Graph comparing the execution time for all three approaches (for a 15-node graph). 46

5.10. A Random 20-Node Network. 47

5.11. Graph comparing the total number of sensors for all three approaches (for a 20 node graph). 48

5.12. Graph comparing the execution time for all three approaches (20-node graph). 49

5.13. A Random 25-Node Network. 50

5.14. Graph comparing the total number of sensors for all three approaches (25-node graph).... 51

5.15. Graph comparing the execution time for all three approaches (25-node graph). 51

5.16. A Random 30-Node Network. 52

5.17. Graph comparing the total number of sensors for all three approaches (30-node graph).... 53

5.18. Graph comparing the execution time for all three approaches (30-node graph). 54

5.19. A Random 35-Node Network. 55

5.20. Graph comparing the total number of sensors for all three approaches (35-node graph).... 56

5.21. Graph comparing the execution time for all three approaches (35-node graph). 56

5.22. A Random 40-Node Network. 57

5.23. Graph comparing the total number of sensors for all three approaches (40-node graph).... 58

5.24. Graph comparing the execution time for all three approaches (40-node graph). 59

5.25. A Random 45-Node Network. 59

5.26. Graph comparing the total number of sensors for all three approaches (45-node graph).... 60

5.27. Graph comparing the execution time for all three approaches (45-node graph). 61

5.28. A Random 50-Node Network. 62

5.29. Graph comparing the total number of sensors for all three approaches (50-node graph).... 63

5.30. Graph comparing the execution time for all three approaches (50-node graph). 63

5.31. A Random 500-Node Network. 64

5.32. Graph comparing the total number of sensors for all three approaches (500-node graph). 65

5.33. Graph comparing the execution time for all three approaches (500-node graph). 66

5.34. A Random 1000-Node Network. 67

5.35. Graph comparing the total number of sensors for all three approaches (1000-node graph). 68

5.36. Graph comparing the execution time for all three approaches (1000-node graph). 68

1. INTRODUCTION

Modern-day networks include and produce a large amount of data. To store and manage huge data such as graphs or networks involve several complexities, such as time and cost. In this paper, to demonstrate such a complex networking problem, Power System Networks is considered as input graphs. For secure and appropriate operations of any power system, the prime factor is synchronization. Synchronization of a power grid with its subcomponents (substations) is essential and can be accomplished by using state estimators that reside on a control center computer that keeps track of all measurements and information from each substation to the respective power grid. By continual monitoring, state estimators may provide the approximation for electrical quantities, such as voltage, current, and power. The estimators may also detect and correct any gross errors in a power system. To circumvent disadvantages due to analog meters, installing a sensor is an appropriate choice.

The synchronized Phasor Measurement Unit (PMU), developed in the 1980s, is considered to be one of the most important devices in the future of power systems [1]. The PMU is a device that is capable of recording electrical characteristics and disturbances for a long period of time in phasor format. The recent development in PMU technology provides high-speed, precisely synchronized sensor data, which have been found to be useful for dynamic state estimation of the power grid [1]. While PMU measurements currently cover less than 1% of the nodes in the U.S. power grid, the power industry has gained momentum to advance the technology and install more units. However, with limited resources, the installation must be selective. In this paper, each substation in the power system network is treated as a vertex or node. We can safely assume PMU is a sensor in this scenario. By solving the current state of a power system with sufficient, but necessary, knowledge on the measurement set and its

distribution, the power system or graph becomes observable (measurable). PMUs or sensors installed on all nodes or substations/vertexes can provide complete observability. However, installing PMUs on every node/vertex may not be economically feasible. When a PMU is deployed at a substation, that substation is termed “directly observable.” A substation is termed as "calculated" when it does not have a PMU installed but has a connection to another substation that has a PMU installed. A system is said to be completely observable when all substations are either directly observable or calculated. If any substations are not observed, then that system is defined as “unobservable.” For instance, consider a power system network that contains seven nodes/vertices as described in Figure 1.1.

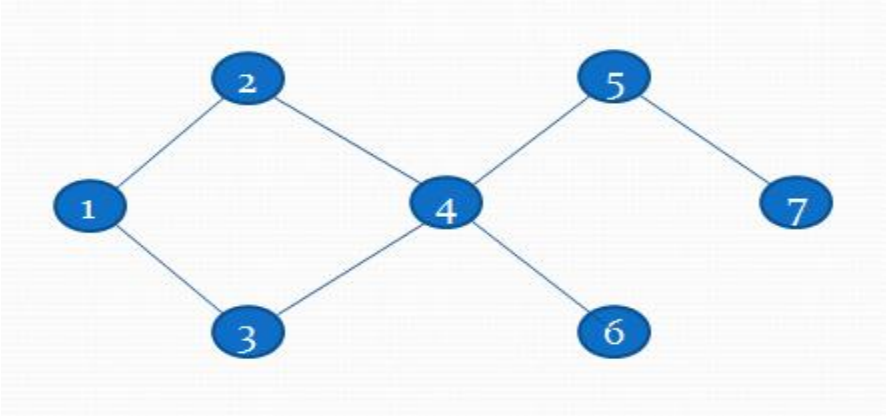

Figure 1.1. A Seven-node power system network.

Assume that PMUs are installed on vertices v_1, v_4, and v_7. We can differentiate directly observable vertices from calculated substations as follows.

- The PMU on v_1 makes vertices v_2 and v_3 visible.
- The PMU on v_4 makes vertices v_2, v_3, v_5, and v_6 visible.
- The PMU on v_7 makes the vertice v_5 visible.

Hence, v_1, v_4, and v_7 are designated as directly observable substations. The remaining substations (v_2, v_3, v_5, and v_6) are designated as calculated substations. A system is termed as completely

observable if and only if all substations are visible. Thus, by choosing an optimum position to install PMUs, we can achieve complete observability along with a reduced deployment cost. The main contributions of this paper are as follows.

- We formulate a graph's theoretical PMU placement problem. This formulation is considerably different and comprehensible than the previous formulations based on the integer linear programming approach.
- The heuristics proposed in this paper are easily comprehensible and extendable to the well-known set-covering problem, vertex-covering problem.
- The execution time of the proposed heuristics is computationally acceptable compared to the traditional integer linear programming technique.
- To standardize our results, we benchmark our proposed heuristics on randomly generated networks with a cost parameter.
- The remainder of this paper is organized as follows. In Chapter 2, we review previous work. The Problem Formulation is detailed in Chapter 3, followed by proposed heuristics in Chapter 4. Simulation results are presented in Chapter 5 and concluding the paper in section VI.

2. LITERATURE REVIEW

The Phasor Measurement Unit (PMU) was first developed and utilized in reference [2] and [3]. The PMU placement problem has its roots embedded in classical graph theoretical optimization problems, such as set covering, vertex covering, and quadratic assignment. In power systems, we are aware of only a few works that have attempted the PMU placement problem.

Considering partially observable systems (with an inadequate number of PMUs), the authors in reference [4] presented an estimation algorithm based on singular value decomposition (SVD) which did not require a complete network system to be observable prior to estimation. In reference [5], the very first placement heuristic based on a spanning tree approach was proposed. However, the high computational complexity of the heuristic prevented usability on large-scale power systems. To improve the execution time of a spanning tree heuristic, a simple modification was proposed to the original spanning tree heuristic. The modification warranted the placement of a PMU on every third substation while walking along the spanning tree of the power system. If the power system was unobservable at the end of the walk, then a second pass was made over the spanning tree to achieve complete observability. However, we argue that the modified spanning tree heuristic may fail to guarantee complete observability when complex power system networks are considered. Our proposed heuristics are scalable and designed to alleviate the issues of loops and back edges.

In reference [6], an integer programming approach to the PMU placement problem was discussed. The execution time of the approach was very high, making it infeasible for large-scale power systems. Our proposed graph theatrical formulation and the proposed heuristics do not rely on the assumptions (like injection nodes).

In reference [7], a matrix-reduction technique to transform a real power system into arrays of busses, branches, and injections was proposed. The transformation is useful to conceive simple placement heuristics. However, the transformation technique itself is very computationally intensive.

In reference [8], a data-driven approach that addresses the three central aspects of the placement problem (measuring the predictive quality of a set of sensor locations regardless of whether sensors were ever placed at these locations, predicting the communication cost involved with these placements, and designing an algorithm with provable quality guarantees that optimizes the NP-hard tradeoff) was presented. Data from a pilot deployment was used in reference [8] to build non-parametric probabilistic models called Gaussian Processes (GPs), both for the spatial phenomena of interest and for the spatial variability of link qualities, which allows us to estimate predictive power and communication cost of un-sensed locations. Using the uncertain models, [8] presented a novel, polynomial-time, data-driven algorithm, pSPIEL, which selects Sensor Placements at Informative and cost-Effective Locations. The approach followed in this paper concentrates on two important properties of this problem: submodularity, formalizing the intuition that adding a node to a small deployment can help more than adding a node to a large deployment, and locality, under which nodes that are far from each other provide almost independent information. All these properties are detailed with supportive results.

In reference [1], an approach that utilizes stochastic models of the signals and measurements, to characterize the observability and corresponding uncertainty of power-system static states (bus voltage magnitudes and phase angles), for any given configuration of PMUs was introduced. Here, we present a new approach to design optimal PMU placement according to estimation uncertainties of the dynamic states. Previous work [9] and [10] was dedicated to the

selection of the best locations to install new PMUs. Several algorithms were developed, primarily with the aim of utilizing a minimum number of PMUs to ensure full network observability. In reference [1], there were two issues related to optimal PMU placement for dynamic state estimation: (a) it was typically assumed that all bus voltages and phase angles are measured directly by PMUs. In reality, one does not have the luxury of having or installing a complete set of PMUs throughout the power grid. It would be useful to know how well a set of available and/or planned PMUs could serve the dynamic state estimator. (b) In previous research, the network topology was the primary focus. In practice, the networks usually have complex topologies where more than one solution for the same minimum number of PMUs will be obtained. In such cases, the planning engineers need to make a choice from among these solutions. Reference [1] focuses on bridging these gaps. A stochastic model that captures dynamic state estimation uncertainties to facilitate the assessment of PMUs installed on a subset of the buses was developed. In reference [1], an optimal PMU placement evaluation algorithm, incorporating uncertainty estimates into topological considerations for the specific network, was designed and presented an approach for the comparison among alternative configurations via quantitative measure of expected uncertainties.

Optimal PMU placement for full observability was studied in reference [9]. An algorithm for finding the minimum number of PMUs required for power-system state estimation was developed, where simulated annealing optimization and graph theory were utilized to formulate and solve the problem. In reference [10], the authors focused on analyzing network observability and PMU placement when using a mixed-measurement set. They developed an optimal placement algorithm for PMUs using integer programming. In reference [11], a strategic PMU placement algorithm was developed to improve the bad data-processing capability of state

estimation by taking advantage of the PMU technology. Furthermore, the PMU placement problem was re-studied, and a generalized integer linear programming formulation was presented in reference [9].

3. PROBLEM FORMULATION

3.1. Problem Definition

For any network, the fundamental task is selecting the best sensor placements. Not only should the sensors be informative, but they should also be able to communicate efficiently [8]. Consider a power system represented as a graph, G (V; E). Let V represent the set of vertices (substations of a power system) for a graph. The set of vertices can be denoted as $\{v_1; v_2; v_3; v_4; \ldots v_n\}$, where $v_i \in V$. Let E represent a set of edges (transmission lines in a power system) for the graph. For n substations, there may be m transmission lines (edges) connecting all vertices/substations. These edges can be represented by $e_{jk} \in E$, where j and k represent the vertices of the edge. A vertex, v_i, is termed visible if there exists a PMU on v_i or on one of v_i's neighboring vertices. Thus, we can write an (n x n) matrix that represents the visibility of a given graph when a certain number of PMUs is placed on the vertices. This matrix is a visibility matrix, denoted by ε. To illustrate the concept of the visibility matrix, consider the power system represented in Figure 1.1. Assume that the power system has a PMU placed on vertex v_4. The visibility matrix of the power system is given in Figure 3.1.

$$\mathcal{E} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{array} \begin{array}{c} \begin{array}{ccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} = 0$$

Figure 3.1. Visibility matrix of the power system with PMU placed on vertex V_4.

Because the example matrix, ε, depicts the visibility of the underlying graph, the elements of ε are Boolean values. From the example power system, we can interpret that vertices v_2, v_3, v_5, and v_6 must be visible when the power system has PMU installed on substation v_4. For that reason, example matrix ε has those elements that are being made visible by the PMU installed on v_4 equal to one. For instance, (a) if we consider v_2, then the elements (v_2; v_4) and (v_4; v_2) are equal to one; (b) If we consider v_7, then the elements (v_7; v_4) and (v_4; v_7) are equal to zero. Moreover, we say that the entire graph is visible if and only if each column (or each row) has at least a single element equal to one. (Because the visibility matrix is always a square bijectional matrix, the condition must either be observed on the rows or the columns, not both.)

Based on the above condition, we can formally state the PMU placement problem as follows. "Find the minimum number of sensor locations that can provide complete observability for the entire system."

Let X be defined as an n x 1 matrix that represents the PMU placement of a graph (power system). An entry, xi ∈ X, is equal to one when there is a PMU on a substation or zero otherwise. Then, the PMU placement problem for an n-vertex power system can be stated as follows.

$$\min \sum_{i=1}^{n} x_i \text{ , where}$$

$$x_i = \begin{cases} 1 \text{ if sensor is installed} \\ 0 \text{ if there is no sensor} \end{cases}$$

It is our belief that the PMU placement problem is closely related to the set-covering problem [8]. The set-covering problem is proven to be NP complete even for the simplest case of identifying a cover for three sets. Because of this similarity, we can safely assume that the PMU is in the NP-class. An optimal solution for a large-scale power system would be inconceivable. Therefore, we must design efficient and effective heuristics.

4. PROPOSED SOLUTION

Before we start detailing our heuristics, one must note that, in a generalized case, there can be no polynomial time algorithm to tackle the PMU placement problem. We successfully determined theoretical bounds for the performance of each proposed heuristic by comparing the optimal solution generated with a tool called AMPL.

When we traverse through a n-vertex graph with a cognizant approach, the traversal order (heuristic solution/optimal solution) is equal to one. Unlike the optimal approach, the heuristic approach should traverse through each vertex to ascertain sufficient knowledge for installing a sensor on any node/vertex. In a generalized case, the traversal order to attain complete visibility varies from one heuristic to another.

The following section details the type of graphs that the heuristics take as input and how such graphs are generated using known techniques, maintaining close resemblance with real-world networks. The generation of graphs as input and attaining a solution for such graphs are integrated into a single, programmable module using Visual Studio 2008's Visual Basic in a machine configured with Windows 7, a 2.3-GHz processor, and 4 GB of Random Access Memory. The following section will render the process of creating a graph which is followed by a working explanation of heuristics.

4.1. Graph Generation

In this paper, we generated our graph/network using four different approaches to gain the advantage of randomness. In each approach, we took some assumptions to achieve a graph that can be close to a real-time network. The following assumptions are common for all four approaches.

- An (i x j) area is considered as the ground (a place to hold our entire graph).

- Each node/substation occupies a certain area in ground.
- No two nodes share the same area.

4.1.1. Random Graph Algorithm with Distance Parameter

In this algorithm, the program asks total number of nodes and acceptable distance as input. The total number of nodes is defined as the number of nodes/substations that our graph shall contain. If an optimal solution is to be found for a network of 200 substations/nodes, then the total number of nodes will be 200.

Acceptable distance can be defined as the minimum distance between two nodes to be qualified as neighboring nodes. Two nodes will be considered neighboring nodes only if the distance between them is less than or equal to a certain distance. Consider a real-time network with three substations: 1, 2, and 3. If substation 1 is located 5 miles from substation 2, if substation 3 is located in the middle of substation 1 and substation 2, and if we can install power lines up to a 3-mile distance, then, in this case, we cannot install a connection between substation 1 and substation 2 (because the distance between the substation 1 and substation 2 is greater than the maximum connection distance of 3 miles). We can install a connection between substation 1 and substation 3, and a connection between substation 3 and substation 2 as shown in Figure 4.1. In this instance, 3 miles is the acceptable distance.

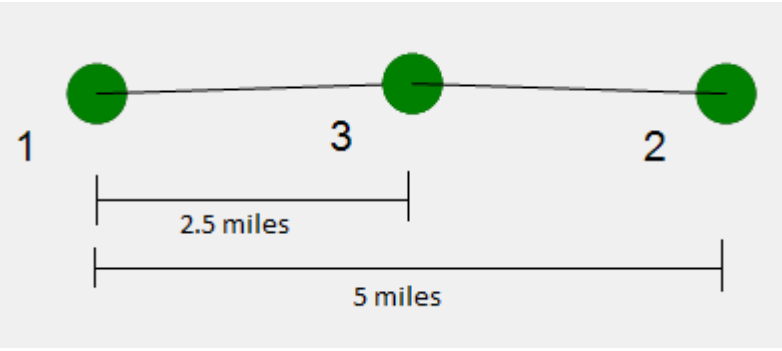

Figure 4.1. Sample real time network with three substations.

After entering the total number of nodes and the acceptable distance, the algorithm creates a matrix “Val (i , j)” that represents a graph. If the total number of nodes is represented by N, then our algorithm creates a N x N matrix. For the graph shown in Figure 4.1, the matrix will be defined as shown in Figure 4.2. There are three substations in the graph, and a 3 x 3 matrix will be created.

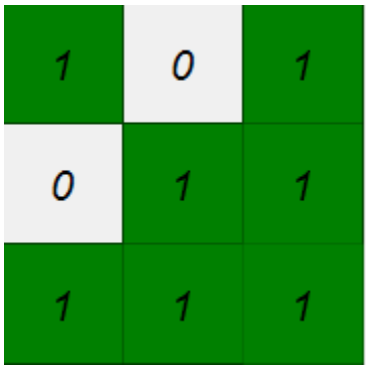

Figure 4.2. A 3 x 3 matrix representing the above graph.

In the above matrix, Val (1, 1) = Val (2, 2) = Val (3, 3) =1 represents that a node is always observable to itself. We do not need a sensor to achieve observability.

Step 1: Get the total number of nodes (N),

Step 2: Generate the matrix,

- Initiate a N x N matrix.
- Assign each value of Val (i, j) to 0, if i <> j.
- By using the random function in the .Net framework, a random point (x-coordinate, y-coordinate) is selected from a (1000, 700) ground. This random point will be unique from all the previous random points (to avoid any overlapping substations). Place a node at every unique random point until we have all N nodes on the ground. After placing nodes on the ground, browse through each node (selected node) to find all neighboring nodes.

- A node is called a neighboring node for a selected node if and only if the distance between them is less than or equal to the acceptable distance. If a node is within the acceptable distance of a selected node, an edge/connection is created between these two nodes, and Val (i, j) is updated to 1. In the above example, node 3 is considered a neighboring node to node 1. Therefore, Val (1, 3) and Val (3, 1) are updated as "1."
- Get neighboring nodes for every node in the graph.

Step 3: Display the graph, and store the matrix in a text file.

- The image in Figure 4.3 shows a random network (including nodes and edges) for 15 nodes in a ground area of (1000p x 700p), 10p (p as in pixels) of node size, and 400 pixels of acceptable distance.

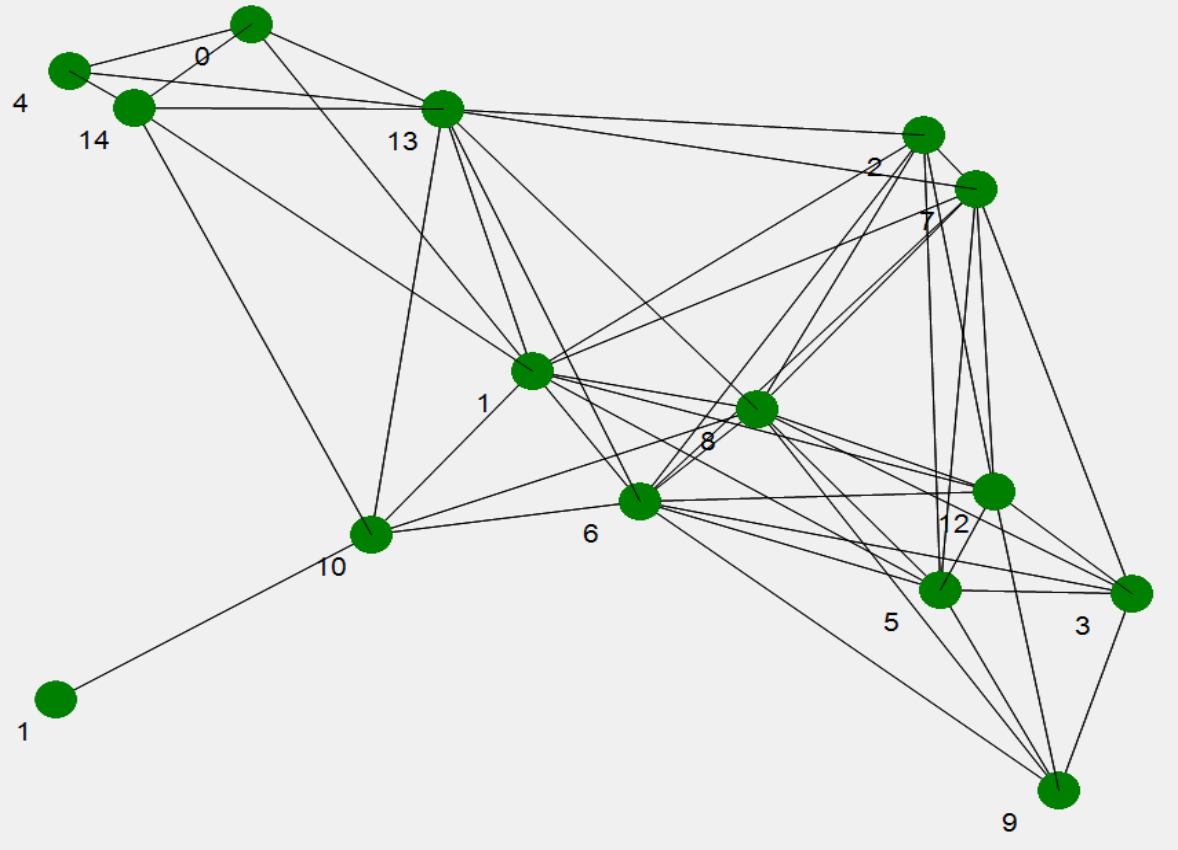

Figure 4.3. A network graph generated by Random Graph Approach (4.1.1) representing a network of 15 nodes.

4.1.2. Random Graph Approach with Distance and Cost Parameters

To make networks close to real-time physical networks, a cost parameter is included in this algorithm. In any real-world networks, there is always a concept of priority. Some locations (in this case nodes/power stations) will always have higher priority when compared to other nodes. There are several reasons for a node to be prioritized. Some of them are as follows:

- Heavy traffic at certain nodes.
- Altitude and latitude of the location, etc.
- This approach unweaves such issues with the help of the random functions available. In this algorithm, we have five different priority levels. Each node in a network is assigned a priority using the random number generator algorithm. The obtained priority is added to a respective node's weight (total number of neighboring nodes). The advantage of adding a priority parameter to a node's weight is detailed in the following section.

4.1.3. Pure Random Network

This approach is a complete random approach that does not involve any dependence parameters such as cost, distance, etc. The idea behind this approach is to develop a network that is completely random. In this approach, the term "random" is significantly different from all other approaches that have been detailed in this document. In this approach, node locations are generated randomly, and also, the connections between nodes are generated randomly. The following steps can brief the process.

For an 'N' node network:

Step 1: Get "N" unique points on the graph using a random number generator algorithm.

Step 2: Use a random number generator to install a connection between two nodes.

- Generate a random number between 0 and 1. A 0 indicates no connection, and 1 indicates a connection between two nodes.
- Update the network matrix .

Consider a three-node network. Each node is placed on the ground using a random number generator as shown in Figure 4.4. A 3 x 3 matrix is created with every element initialized to 0. As defined in Step 2, when node 0 is considered, a random number between 1 and 0 is generated for every connection of node 0. In this case, the connection between node 0 and node1 and also the connection between node 0 and node 2 is 1. The only left-out connection is the connection between node 2 and node 1. In this case, the random number generator produces a value of 0. Due to this value, there is connection between node 2 and node 1. The final graph and matrix is shown in Figure 4.5.

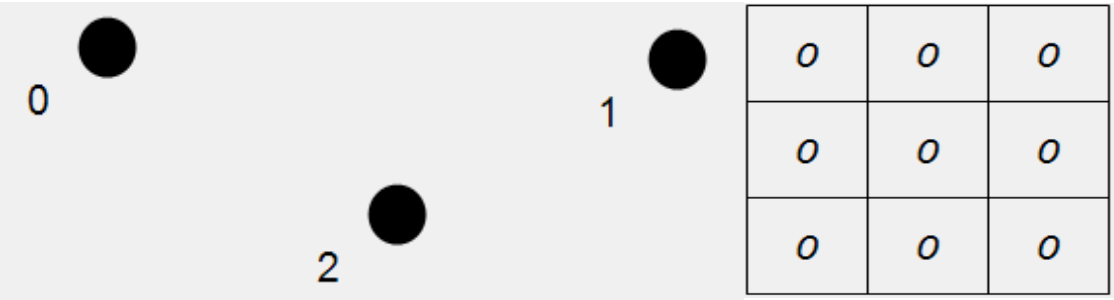

Figure 4.4. An initial three-node network graph with no connection between the nodes.

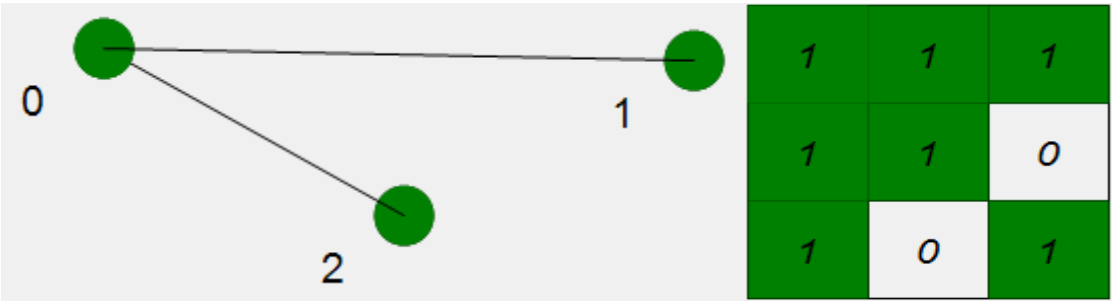

Figure 4.5. The final network graph obtained after implementing the Pure Random Network approach.

4.1.4. Load Graph

This is a standard approach which is generally used when there is a need to determine an optimal solution for a known graph. This method plays a vital role when the ideal approach (AMPL Solutions) is compared with the heuristics proposed in this document. After generating a random graph using any of the above approaches, the same graph should be applied to each algorithm to compare results. With this approach, one can load the same graph for every approach and generate solutions accordingly.

Below, we detail our two proposed heuristics described by pseudo-code followed by analyzing the performance of algorithms by calculating the mean on randomly generated network models and comparing the ideal solution produced by AMPL.

4.2. Deterministic Greedy Heuristic

The Deterministic Greedy (DG) algorithm is the first greedy inherited algorithm among the two proposed solutions in this paper. The DG heuristic is divided into different parts where each part has a specific algorithm developed. The following section details the process of this heuristic in a procedural way.

4.2.1. Description

Input: Randomly generated graph with n vertices.

Output: Set of vertices that can make the entire graph observable/visible

- Assumptions
    - Weight: Total number of neighboring nodes for a given node.
    - Acceptable distance: Any two nodes are considered as neighboring nodes if and only if the distance between these nodes is less than or equal to acceptable distance.

- Vertex overlapping: In order to achieve a network close to real-world networks, the vertex overlapping assumption is implemented to avoid overlap for any two nodes. In a real-world network such as power-system network, no two substations are in an overlapping range.

Step 1: Generate a graph

- Place a node randomly in a given area using any of the random graph-generation techniques discussed above.
- Repeat the above step without any vertex overlapping n times.
- Connect every node with respective neighboring nodes.
- Generate respective network matrix.

Step 2: Check Observability

- Check graph/network for complete observability (O_b). If O_b is achieved, then go to Step 6. Else Continue.

Step 3: Get Maximum-Weight Node

- Get a node that has the maximum weight.

Step 4: If there are more than two nodes with the same weight, store all such sensors in a list called SensorNodesWithSameWeight.

- Get Secondary Weight: Weight of each neighboring node for a given node in the list.
- Select the node for which the secondary weight is the list minimum.

Step 5: Apply a sensor to the graph and repeat Step 2.

Step 6: Exit function.

A detailed explanation of the above DG algorithm is provided in Section 4.2.2 with the help of a seven-node graph.

4.2.2. Explanation of DG Heuristic

The first step is to generate a complete random network using the random graph-generation approach described in Section 4.1.1.

Step 1: Generate graph

- Assign a network size of 7 nodes. The application asks for the size of the network as shown in Figure 4.6.

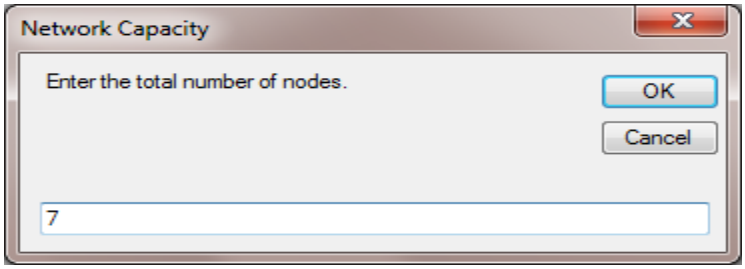

Figure 4.6. A screenshot of the input box allowing a user to enter the network size.

After assingning network size, input the acceptable distance variable which will determine the maximum distance between any two nodes (Figure 4.7).

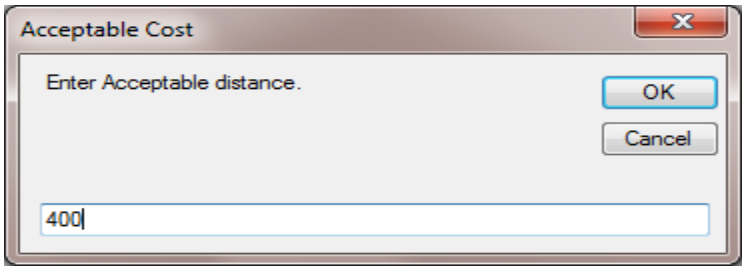

Figure 4.7. A screenshot of the input box allowing the user to enter the acceptable distance between any two nodes in the network.

- A random graph (Figure 4.8) will be generated, displaying the GUI part to the user (front end) and storing the respective network matrix (N_m) both in the application memory and as a text file (for future reference).

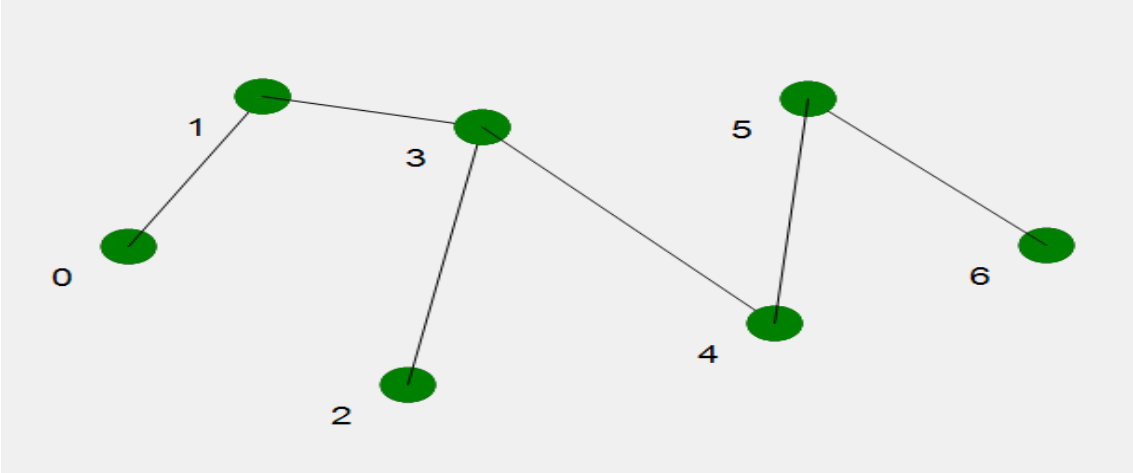

Figure 4.8. The resulting seven-node graph generated from the user inputs.

1	1	0	0	0	0	0
1	1	0	1	0	0	0
0	0	1	1	0	0	0
0	1	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	0	1	1	1
0	0	0	0	0	1	1

Figure 4.9. The matrix representation of the graph generated from the user input.

Step 2: Check Observability

- Before checking for the matrix observability, create a copy of the network matrix (C_m).
- Check for the matrix coverage/observability. (Browse through each column, and check for any -1 value in a column. -1 indicates that the sensor is covered).
- If every column in the matrix has -1 in it, then such a matrix will be considered observable.

- As you can see in Figure 4.9, there is no -1 in any column, and therefore, this matrix (N_m) can be considered an uncovered matrix. (A sample VB code is shown in Figure 4.10.)

```
Returns a boolean value which indicates if a network is covered or not
Public Function IsCovered() As Boolean
    'Browsing throgh each column of the M X N matrix to look for a sensor.
    For j = 0 To N - 1
        Dim SensorCovered As Boolean = False
        For i = 0 To M - 1
            'If any cof the column value is =-1,
            'then the respective node will be considered as covered or observable
            If val(i, j) = -1 Then
                SensorCovered = True
                Exit For
            End If
        Next

        If Not SensorCovered Then
            Return False
        End If
    Next

    Return True
End Function
```

Figure 4.10. A code snippet representing the network coverage.

Step 3: Get Maximum-Weight Node

- Browse through every node, and get a set of nodes that has the maximum weight. This looks similar to the traditional greedy approach. Heuristic 4.2 is called a Deterministic Greedy approach due to the following reasons.
- After browsing through each node, get a complete set of nodes such that the weight of each node in the respective set is equal to the maximum weight. In this seven-node example, it will be just node 3 with a weight of 4 (3 (connections to node 3) + 1 (itself) = 4)
- Place a sensor on this node. (Change all 1's to -1's for node 3 in the network matrix (N_m).) The copy matrix (C_m) will remain unchanged.

1	1	0	0	0	0	0
1	1	0	-1	0	0	0
0	0	1	-1	0	0	0
0	-1	-1	-1	-1	0	0
0	0	0	-1	1	1	0
0	0	0	0	1	1	1
0	0	0	0	0	1	1

Figure 4.11. The matrix representation of the graph after placing sensor a node 3.

- Go to Step 2 to check for the coverage. Because some of the columns in the N_m matrix still do not contain -1, N_m can be considered as uncovered.
- Repeat Step 3 until there is more than one Maximum-Weight node. In the seven-node example, after placing the sensor on node 3, nodes 5 and 6 will have the same weight. (Placing a sensor on either node 5 or node 6 will cover both nodes 5 and 6.) One should note that placing a sensor on node 0 can cover only node 0 because node 1 is covered by node 3.
- If there are two or more possible sensor nodes, calculate their respective secondary weight.

Step 4: Get Secondary Weights

- Getting the secondary weight of a node is nothing but the summation of each neighboring node's weight for the respective node.
- In this case, for node 5, the neighboring nodes are nodes 4 and 6. Because node 4 is already covered, it is excluded from the list. Now, the secondary weight of node 5 is the weight of

node 6 = 2 (i.e., There are only two possible ways to cover node 6.). Similarly for node 6, the secondary weight is the weight of node 5 =3.

- Choose a node that has the minimum secondary weight. The reason behind this approach is that the secondary weight of a node indicates the total number of ways a specific set of nodes will be covered. In this case for node 5, its neighboring node 6 can be covered by two different nodes. For node 6, its neighboring node 5 can be covered by three nodes. Hence, the probability for neighboring nodes of node 5 getting left out is higher than for node 6.
- The resultant matrix is shown in Figure 4.12.

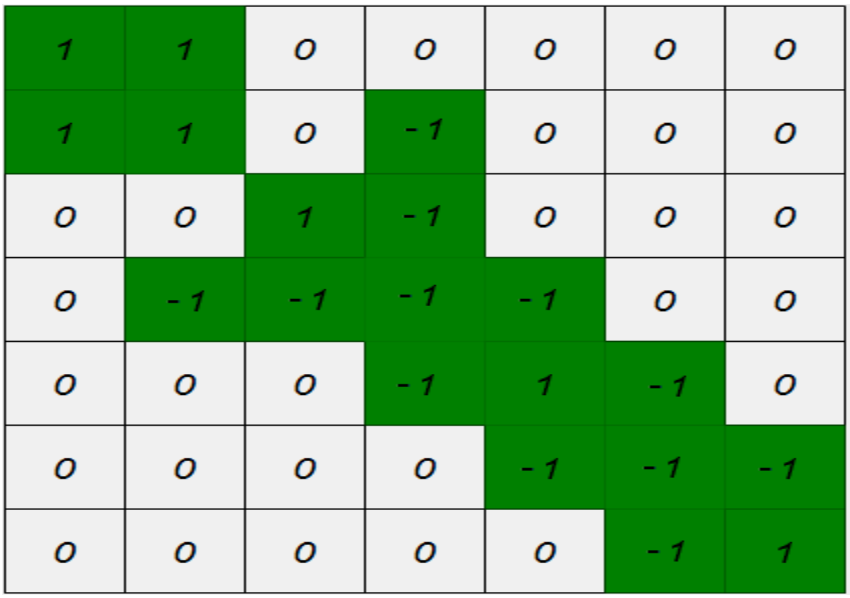

Figure 4.12. The resultant matrix obtained after performing the sequence of operations in Step 4.

- Place a sensor on node 5, and go to Step 2. Because column 0 still does not have a -1, the algorithm proceeds to Step 3.
- Get the maximum node from the matrix. Node 0 has a weight of 1 (because its only neighboring node, node 1, is already covered), and node 1 has a weight of 1 (due to node 0). Now, there are two Maximum-Weight nodes. Go to Step 4.

- Get the secondary weight of node 0. Because there are no uncovered nodes as neighboring nodes for node 0, the secondary weight is 0. For node 1, the secondary weight is 1.
- Choose the node that has the minimum secondary weight. Node 0 is selected; hence, the set is complete, and graph is completely observable.

-1	-1	0	0	0	0	0
-1	1	0	-1	0	0	0
0	0	1	-1	0	0	0
0	-1	-1	-1	-1	0	0
0	0	0	-1	1	-1	0
0	0	0	0	-1	-1	-1
0	0	0	0	0	-1	1

Figure 4.13. The final matrix indicating that the graph is observable.

4.3. Deterministic Greedy Heuristic with Look Ahead

A new concept called Look Ahead is introduced in the second proposed heuristic. In this greedy inherited algorithm, the same idea of the Maximum-Weight node as explained in algorithm 4.2 but with a slightly different approach. Section 4.3.1 provides a pseudo code presentation of algorithm 4.3 followed by a detailed explanation of the algorithm in Section 4.3.1.

4.3.1. Description

Input: Randomly generated graph with n vertices.

Output: Set of vertices that can make the entire graph observable/visible

The assumptions and terminology used are as follows:

- Weight: The weight of a given node can be defined as the total number of nodes that can be covered when a sensor is installed on a particular node. Consider a seven-node graph as shown in Figure 4.7. The weight of Node 0 can be defined as follows:
- Weight (Node 0) ➔ Total Number of Neighboring nodes (Node 0) + 1 (itself)

  ➔2 (Node 1 and Node 2) + 1

  ➔3
- Acceptable Distance: Any two nodes are considered as neighboring nodes if and only if the distance between these nodes is less than or equal to the acceptable distance.
- Node overlapping: No two nodes should overlap in a given graph. This assumption is to compensate the fact that, in real-time networks, no two substations are installed in the same region.
- Network Matrix: A matrix that represents the generated graph and every state of the corresponding graph before and after installing sensors.
- Look-Ahead Node: The concept of a Look-Ahead node is used in Algorithm 4.3 to minimize the ambiguity to select a Maximum-Weight node. The Look-Ahead node for a specific Node A can be defined as the next Maximum-Weight node after placing a sensor on Node A. In other words, the Look-Ahead node for Node A can be defined as a node that can cover the most uncovered nodes. Figure 4.14 gives a detailed explanation of a Look-Ahead node.

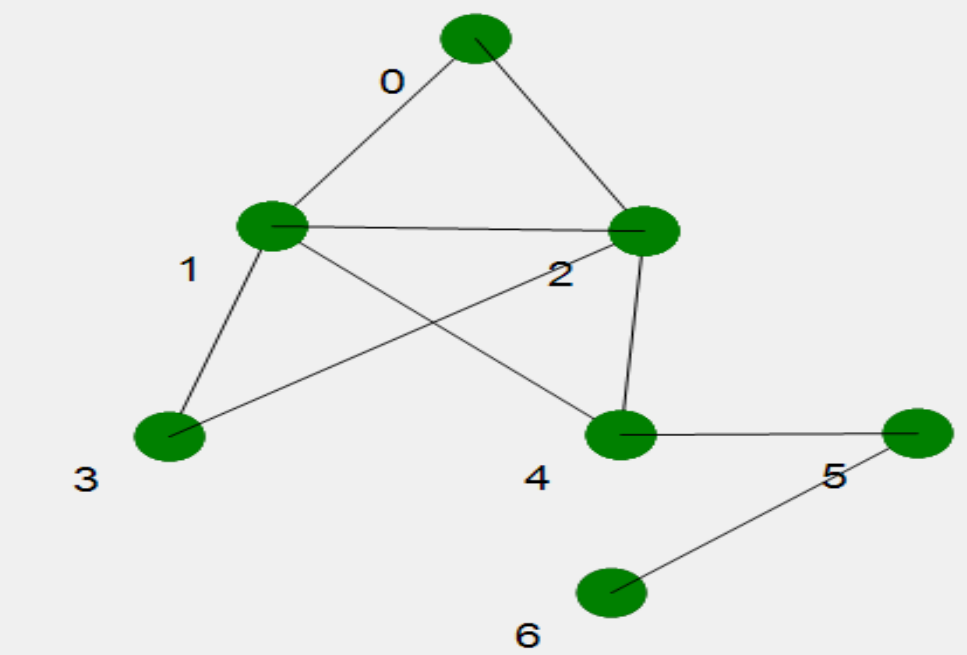

Figure 4.14. A sample graph used to help identifying the concept of Look-Ahead nodes.

In Figure 4.14, Node 2s and 1 have the same weight of 5 (4 neighboring nodes + themselves). For Node 1, the Look-Ahead node is a node that has the maximum weight among all the uncovered, non-neighboring nodes of Node 1 (Nodes 4, 5, and 6). Node 5 has a maximum weight of 3 (2 neighboring nodes + itself), so the Look-Ahead node for Node 1 is Node 5.

- Secondary Weight: The weight of a Look-Ahead node. In the example shown in Figure 4.14, the secondary weight of Node 1 can be defined as follows:
    - Secondary Weight (Node 1) ➔ Weight (Look-Ahead node (Node 1))
      ➔Weight (Node 5)
      ➔3

Step 1: Generate a random graph.

- Generate a random graph using any of the random-graph generation techniques discussed in Section 4.1.

Step 2: Check for coverage of the graph.

- Check for the observability of a given graph. If the graph is completely covered, then go to Step 5; otherwise, go to Step 3.

Step 3: Get the Maximum-Weight node and Maximum-Weight Look-Up node.

- The most Maximum-Weight node is the node that has the maximum weight.
- The Look-Up node is the node that has the maximum weight for all neighboring nodes of the Maximum-Weight node.
- The decision to pick a Maximum-Weight node hugely depends on the Look-Up node.

Step 4: Apply sensors.

- Apply sensors to both nodes (Maximum-Weight node and Look-Up node).

Step 5: Get the removable sensors.

- Check for the coverage percentage of each node, and remove any sensors that are installed unnecessarily.

4.3.2. DG Heuristic with Look-Ahead Node Explanation

- Generate Graph
    - Generate a seven-node graph using one of the random-graph generation techniques discussed in Section 4.1. For Algorithm 4.3, use the approach described in Section 4.1.1.
    - Enter the size of the network as 7 (total number of nodes) as shown in the Figure 4.15.
    - Enter maximum distance allowed between two nodes as 400 (Acceptable Distance) as shown in Figure 4.15.

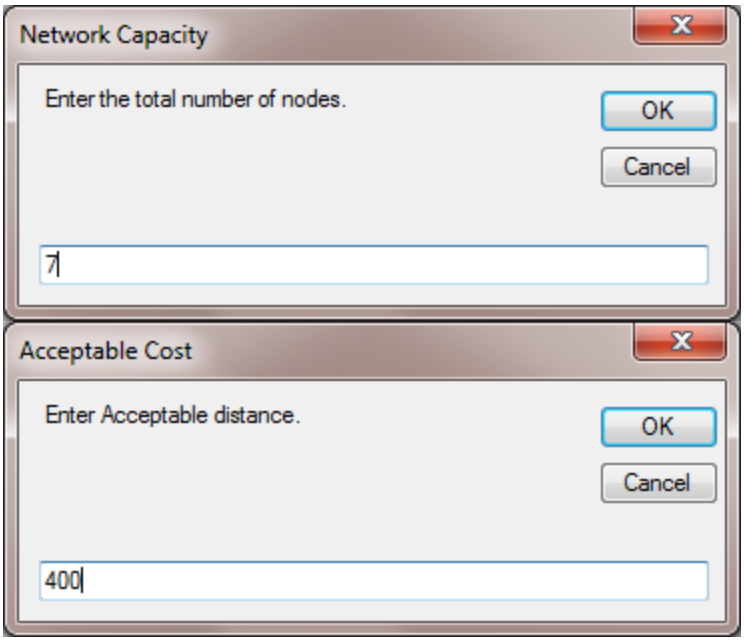

Figure 4.15. Screenshots of the input prompts allowing the user to enter the number of nodes and the acceptable distance between two nodes of a network.

- A random graph will be generated with seven nodes displaying a GUI to the user as shown in Figure 4.16 and also storing a two-dimensional matrix at the backend as shown in Figure 4.17 which can be represented by N_m.

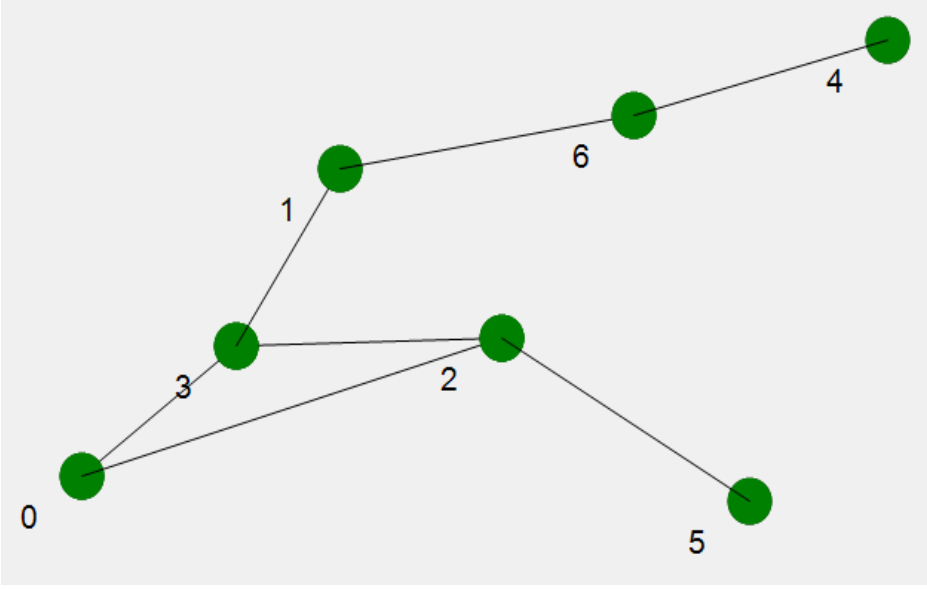

Figure 4.16. A random seven-node graph generated from user input.

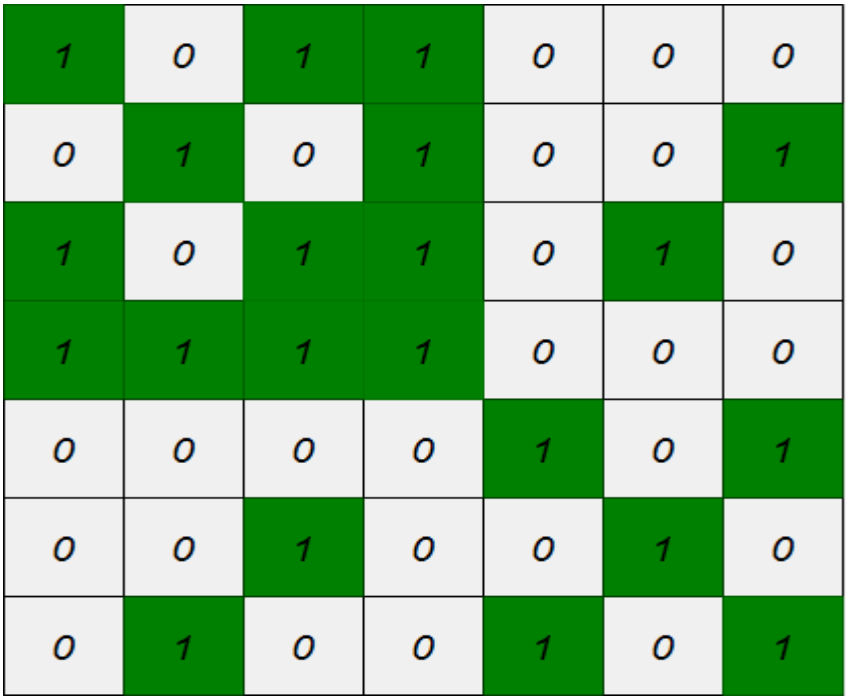

Figure 4.17. Corresponding matrix representation for the generated graph in Figure 4.16.

- Check for complete observability.
    - Although the graph generation for Algorithm 4.3 is similar to Algorithm 4.2, the procedure followed to determine the graph's observability is different.
    - In algorithm 4.3, a graph is said to be covered if and only if every value of its network matrix (N_m) is zero.
    - The algorithm will browse through every column of the matrix looking for a value of one. If there is any such value, the respective matrix is considered to be uncovered.
    - From Figure 4.17, it is evident that the considered seven-node graph is clearly an uncovered graph.
    - A sample VB code (using two iterative For-loops) to determine the coverage of a matrix is given in Figure 4.18.

```
For j = 0 To N - 1
    Dim SensorCovered As Boolean = False
    For i = 0 To M - 1
        If val(i, j) = 1 Then
            Return False
        End If
    Next
Next
Return True
```

Figure 4.18. A code snippet for determining the coverage of a graph.

- Get Maximum-Weight node and Look-Ahead node.
    - Getting the Maximum-Weight node in algorithm 4.3 starts with a similar approach as in algorithm 4.2. The node that has the maximum weight is selected similar to all traditional greedy algorithms. If there are two or more nodes qualified to be the most Maximum-Weight node, then algorithm 4.3 employs a different approach to determine the appropriate sensor location.
    - Unlike algorithm 4.2, algorithm 4.3 tries to use the Look-Ahead node concept to determine which node is best suited to place a sensor in a given graph.
    - Consider the seven-node network shown in Figure 4.16. As described in Step 2, algorithm 4.3 checks the observability of a given graph. Initially, the network matrix (N_m) will not be have any sensors installed. Therefore, it is safe to say that the graph is not covered. In other words, there are still some columns in the network matrix (N_m) that have ones in them, so the matrix (N_m) is still uncovered.
    - To get the Maximum-Weight node for the given matrix (N_m), get a node that has the maximum weight. In this case, Nodes 2 and 3 are the most suitable candidates for the Maximum-Weight node. For Node 2, the weight is Node 0 + Node 3 + Node 5 +

itself = 4 (because Nodes 0, 3, and 5 are neighboring nodes for Node 2). Similarly for Node 3, the weight will be Node 0 + Node 1 + Node 2 + itself = 4.

- To determine the Maximum-Weight node in Nodes 2 and 3, Algorithm 4.3 employs the Look-Ahead node technique.
- **Look-Ahead node technique:** In this technique, the network matrix (N_m) in the current instance is copied to a new matrix (C_m).
- For Node 2, get the nodes that are covered when a sensor is placed on Node 2 (neighboring nodes of Node 2) and also get the nodes that are not covered by Node 2 when a sensor is placed.
- To get nodes that will not be covered by Node 2, browse through each value in Node 2's row, and get all the nodes that have a 0 value. In this case, Nodes 1, 6, and 4 are the non-neighboring nodes for Node 2. After obtaining the non-neighboring nodes for Node 2, apply a sensor on Node 2. The resultant matrix (C_m) is shown in Figure 4.19.

0	0	0	0	0	0	0
0	1	0	0	0	0	1
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	1	0	1
0	0	0	0	0	0	0
0	1	0	0	1	0	1

Figure 4.19. The matrix representation of a network after applying a sensor on node 2.

- Applying a sensor on a particular node is nothing but making every value in a neighboring node's column to 0, including the respective node's column. A sample code to apply a sensor on a given matrix is shown in Figure 4.20.

```
For i = 0 To N - 1
    If LocalVal(node, i) = 1 Then
        For k = 0 To M - 1
            LocalVal(k, i) = 0
        Next
    End If
Next
```

Figure 4.20. A code snippet for applying a sensor on any given node.

- Now to determine the Look-Ahead node, select a node that has the maximum weight from Node 2's non-neighboring nodes: Nodes 1, 4, and 6.
- From Figure 4.19, the weights of non-neighboring nodes (Node 1 (2), Node 4 (2), and Node 6 (3)) are determined. Node 6 is the suitable candidate for the Look-Ahead Node because of its maximum weight of 3. For Node 2, the Look-Ahead node is Node 6 with a secondary weight of 3, so the total weight for Node 2 is Node 2's weight + Node 2's secondary weight (Look-Ahead node's weight) = 4 + 3 = 7.
- Similarly for Node 3, get all non-neighboring nodes. Note the matrix (N_m) still remains unchanged.
- Create and save a copy of the network matrix (N_m) as C_m. Apply a sensor on Node 3 using C_m after obtaining all the non-neighboring nodes of Node 3 (Nodes 4, 5, and 6). The resultant matrix (C_m) is shown in Figure 4.22.
- The weights of all neighboring nodes (Node 4(2), Node 5(1), and Node 6(2)) are calculated from the matrix (C_m). Node 6 is an ideal candidate for the Look-Ahead node with a maximum weight of 2.

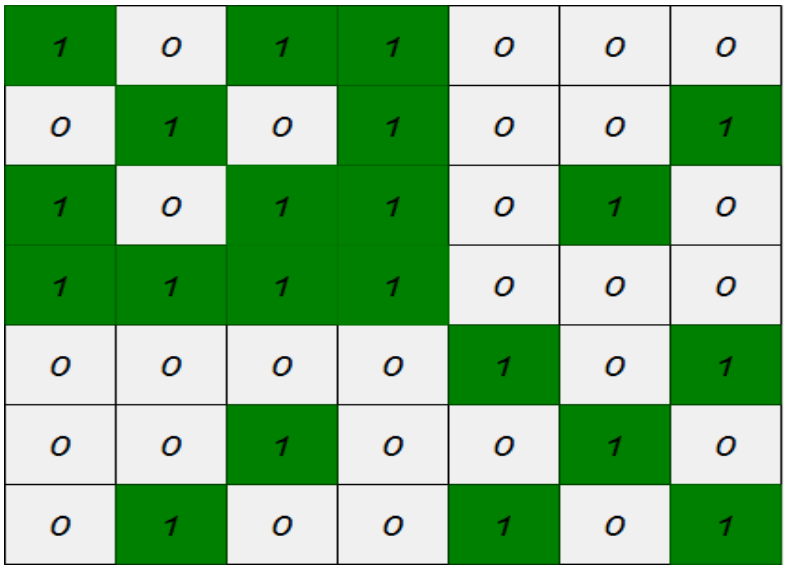

Figure 4.21. Matrix representation of the network after computing all neighboring nodes for node 3.

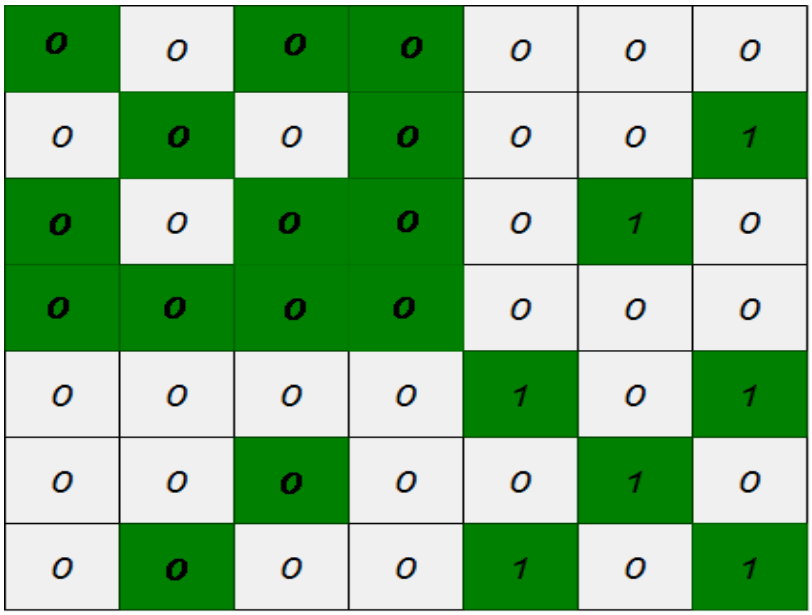

Figure 4.22. Matrix representation of the network after applying a sensor on node 3.

- The total weight for Node 3 is Node 3's weight + Node 3's secondary weight = 4+2 = 6. Therefore, the ideal candidate for the most Maximum-Weight node is the node that has the maximum weight (so that the most probable node along with the Look-

Ahead node) covers the maximum number of nodes possible. In this case, Node 2, with a total weight of 7, is the most Maximum-Weight node.

- Apply sensors on both the Maximum-Weight node (Node 2) and the Look-Ahead node (Node 6). Update the network matrix (N_m).

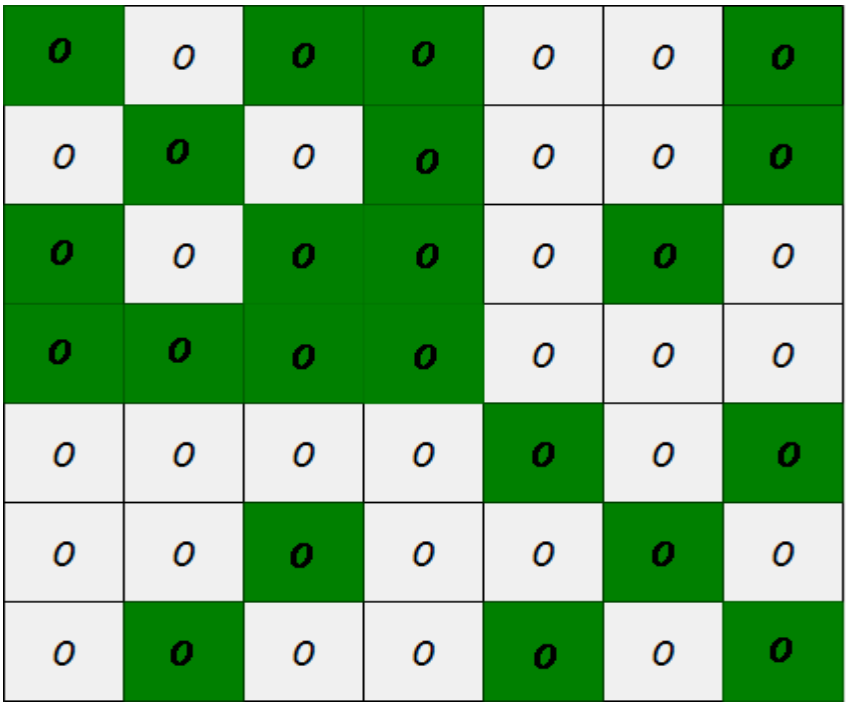

Figure 4.23. Matrix representation of the network after applying sensors on the Maximum-Weight node (Node 2) and the Look-Ahead node (Node 6).

- Check for coverage of the matrix. Because every cell in every column in a given matrix (N_m) is zero, it is safe to assume that the resultant matrix is completely observable.
- Remove unnecessary sensors. The final step after covering a graph includes the validation of every sensor installed on a given graph.
    - Browse through every sensor in a sensor set for a given network.
    - Uninstall the sensor, and check for the matrix coverage.
    - If the matrix is still covered, then remove the sensor.

- If not, end of the sensor list; then, Go to Step 1; otherwise, exit.

The following section includes all the experimental results for Algorithms 4.2 and 4.3 benchmarked with the optimal solutions obtained by using industry a standard tool called AMPL. Before going further into the experimental results, a brief introduction about AMPL is documented in the following section.

4.4. AMPL

Developed at Bell Laboratories, "AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems both in discrete or continuous variables". The AMPL tool lets the user utilize common notations and familiar concepts to formulate optimization models and examine solutions while the computer manages communication with an appropriate solver. In this paper, we used AMPL Plus, a subversion of AMPL that includes a GUI, a language processor, and a solver.

4.4.1. Graphical User Interface (GUI)

The AMPL-Plus GUI provides a Windows interface to the user. The main display window is called the Multiple Document Interface (MDI). The MDI window consists of a Title Bar (top), Menu Bar (top), Tool Bar (top), Status Bar (bottom), and three standard windows (Command, Solver Status, and Model) as shown in Figure 4.24.

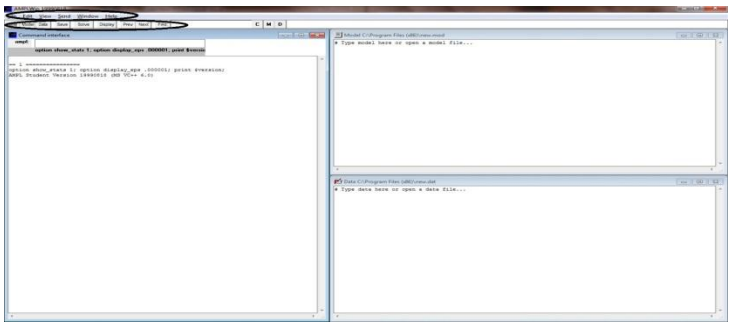

Figure 4.24. A screenshot depicting the MDI window user interface.

4.4.2. The AMPL Language Processor

The first and very basic version of AMPL is a command-line system. In AMPL Plus, selecting an item from the pull-down menus or selecting an icon from the toolbar "generates" commands that are understood by the AMPL language processor.

4.4.3. AMPL-Plus Solvers

The solvers take the output generated by the AMPL language processor and do the work of solving the optimization problem for you. In this paper, we used a solver called CPLEX.

AMPL separates logical statements and modeling data values into two different files called a model file and a data file. Consider the simple optimization problem described below:

Maximize $3x_1 + 2x_2$

Subject to

$2x_1 + x_2 <= 100;$

$x_1 + x_2 <= 80;$

$x_1 <= 40;$

$x_2 >= 0;$

In this example, both the logical statements and modeling data values are applied together as input. If the number of variables has increased from 2 to 99, then it will be hard to represent in a single file. For large-scale optimization problems, the model logic statements described above are separated into two individual files.

4.4.4. AMPL Working Procedure

This section details how the AMPL-Plus tool works on a given network. Consider a seven-node network as shown in Figure 4.16. To solve a placement problem using the AMPL tool, we used a model file to solve the set-coverage problem in general.

The AMPL edition considered in this paper is limited to 300 variables and a total of 300 objectives and constraints and also by the resources available to the computer that is running AMPL. In practice, memory is most often the limiting resource. The amount of memory used by AMPL necessarily depends on the numbers of variables and constraints, and on the number and complexity of the terms in the constraints or in the case of linear programs, on the number of nonzero coefficients in the constraints. Memory use also depends on the size and complexity of the model's sets and parameters. AMPL's memory use for a linear program can be estimated crudely as 1000000 + 260 (m + n) + 50 nz bytes, where m is the number of constraints, n is the number of variables, and nz the number of nonzeroes. Allowing for memory occupied by DOS, this suggests that the following sizes of problems might be accommodated on a PC:

m = 1200, n = 4800, nz = 14400 on a 4MB machine,

m = 3600, n = 14400, nz = 43200 on an 8MB machine,

m = 8000, n = 24000, nz = 96000 on a 16MB machine.

Memory required for a solver must be added, if the solver is to be run while AMPL remains active. Especially difficult problems may cause solver to encounter its own resource limitations. Solvers that accept binary and integer variables, in particular, can consume impractical amounts of time and disk space as well as memory, even for problems of apparently modest size. Nonlinear problems can also be very time-consuming compared to linear programs in the same numbers of variables and constraints.

As a practical matter, there is no substitute for experimentation to determine the resource requirements of a particular problem. The best approach is usually to try out a model on progressively larger collections of data, so that one can record the trend in resource requirements as problem size increases.

Step 1: Load the model file in AMPL. The logical statements inside the model file are shown in Figure 4.25.

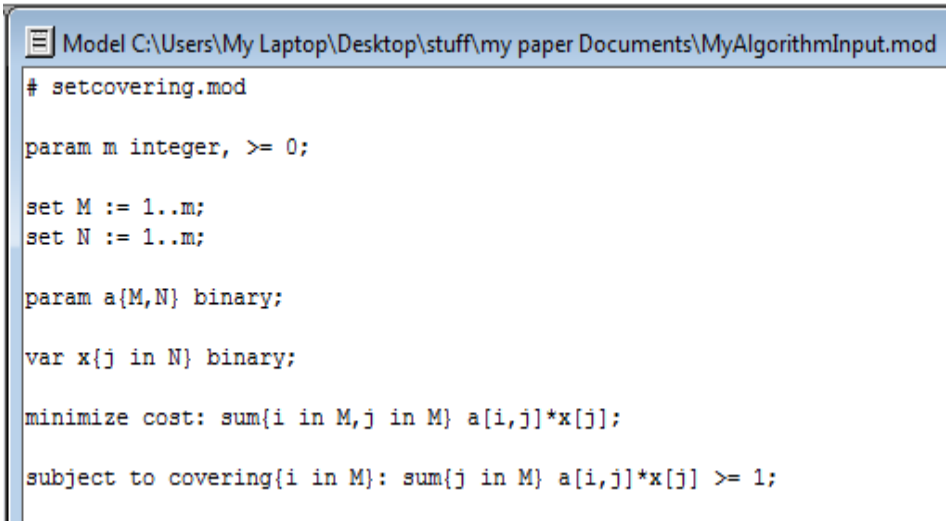

Figure 4.25. A screenshot representing the procedure to load a model file in AMPL.

Step 2: Load the data file.

- In the important special case of linear programming, the largest part of the algorithm's form is the representation of the constraint coefficient matrix. Typically this is a very sparse matrix whose rows and columns number in the hundreds or thousands, and whose nonzero elements appear in intricate patterns. Each element in a matrix represents a connection between two nodes. Therefore AMPL will fail to solve a graph, if there are more than 300 connections between nodes.
- After loading the model file, provide the network matrix (N_m) as shown in Figure 4.26. A data file is generated resembling the considered network. If a seven-node network is generated using any of the random-graph generation techniques discussed in Section 4.1, then a 7 x 7 matrix is also generated, indicating the node connections. If Node-1 is connected to Node-2, then a bit value of 1 will be stored in the (1, 2) and (2, 1) locations of the matrix

to indicate a connection between node-1 and node-2. The generated matrix will have 14 connections and there by 7 variables (< 300 Variable limit). Therefore AMPL can sufficiently solve the generated graph.

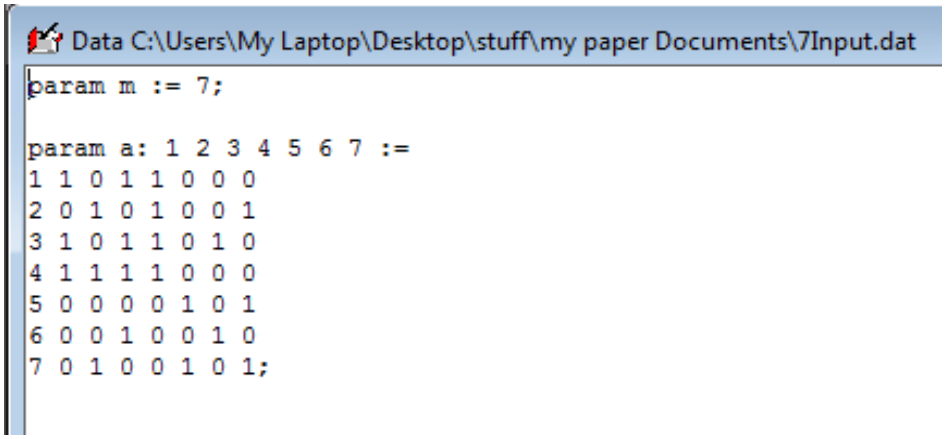

Figure 4.26. A screenshot representing the parameters required to provide a network matrix to AMPL.

Step 3: Click the Solve button in the tool bar. By instantiating the solve button, AMPL will start the CPLEX solver and will try to get an optimum solution (Figure 4.27).

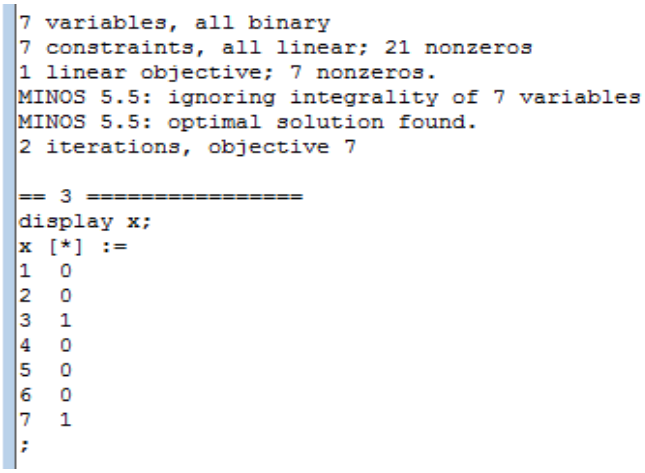

Figure 4.27. A screenshot depicting the optimal solution generated by the CPLEX solver after clicking the solve button.

5. EXPERIMENTAL RESULTS

This chapter demonstrates the effectiveness of our proposed algorithms (Algorithms 4.2 and 4.3) in terms of the minimized total cost (to allocate sensors) and in scalability using various test cases and by calculating the mean for every test case.

5.1. Test Cases

In the following set of test cases for each random network generated, the number of sensors required to attain complete observability is determined by applying both the proposed algorithms and AMPL. After generating a solution for an n-node network with a specific set of connections, a new pair of connection sets is generated for each n-node network, and a mean is calculated for the total number of sensors and the execution time.

5.1.1. Test Case 1

In this test case, a five-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.1. Save the resultant network matrix as a text file (input for Algorithms 4.2 and 4.3) and a data file (input for AMPL-Plus).

Create a network matrix (N_m) from the saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 2 {0, 1}

Execution Time (in seconds) = 0.021

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 2 {0, 2}

Execution Time (in seconds) = 0.022

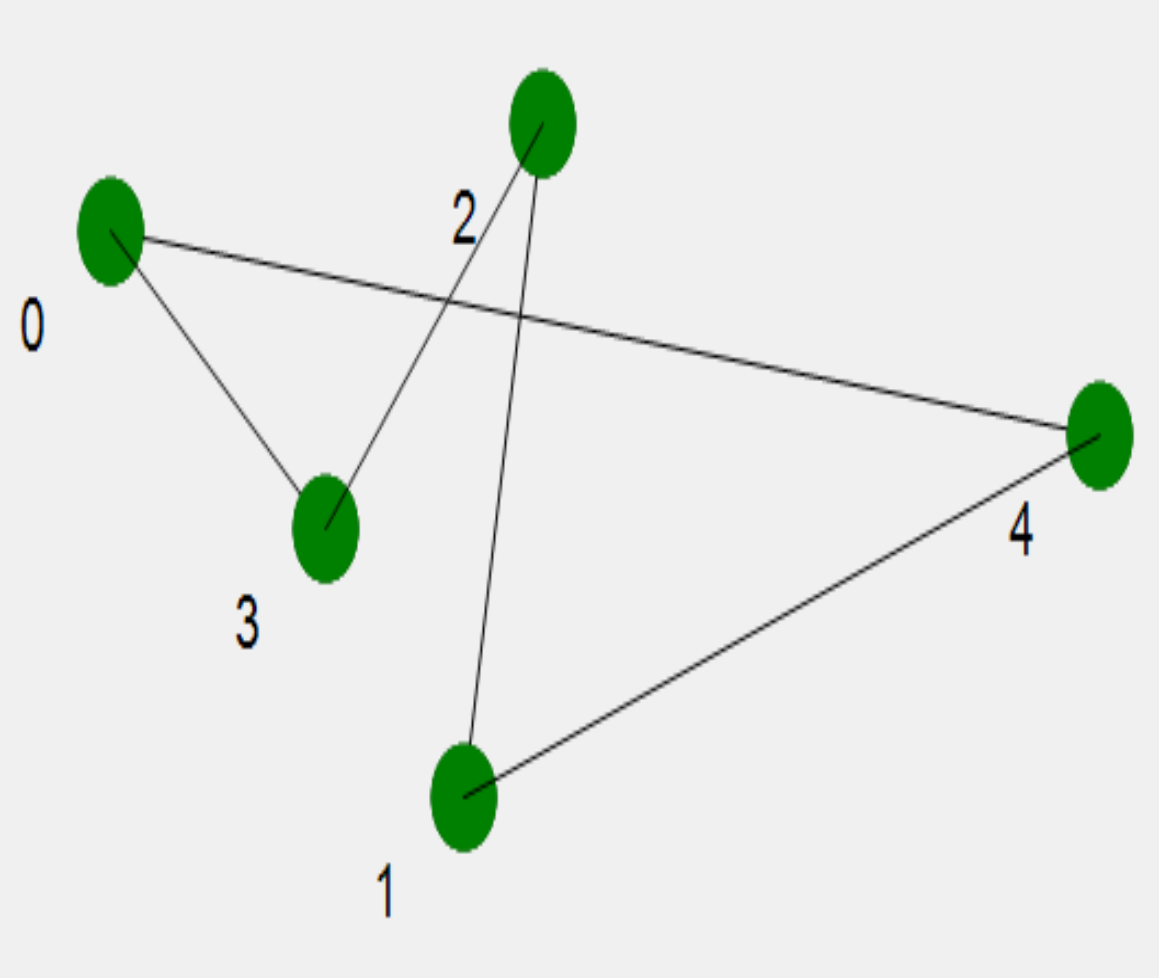

Figure 5.1. A Random Five-Node Network.

Apply the same network matrix (N_m) as input for AMPL using the saved data file

Total Number of Sensors (T_s) = 2 {0, 1}

Execution Time (in seconds) = 0.0

Generating 50 possible replications of the network topology for five nodes, the mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for a number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times of Algorithms 4.2 and 4.3 are generated as shown in Figures 5.2 and 5.3.

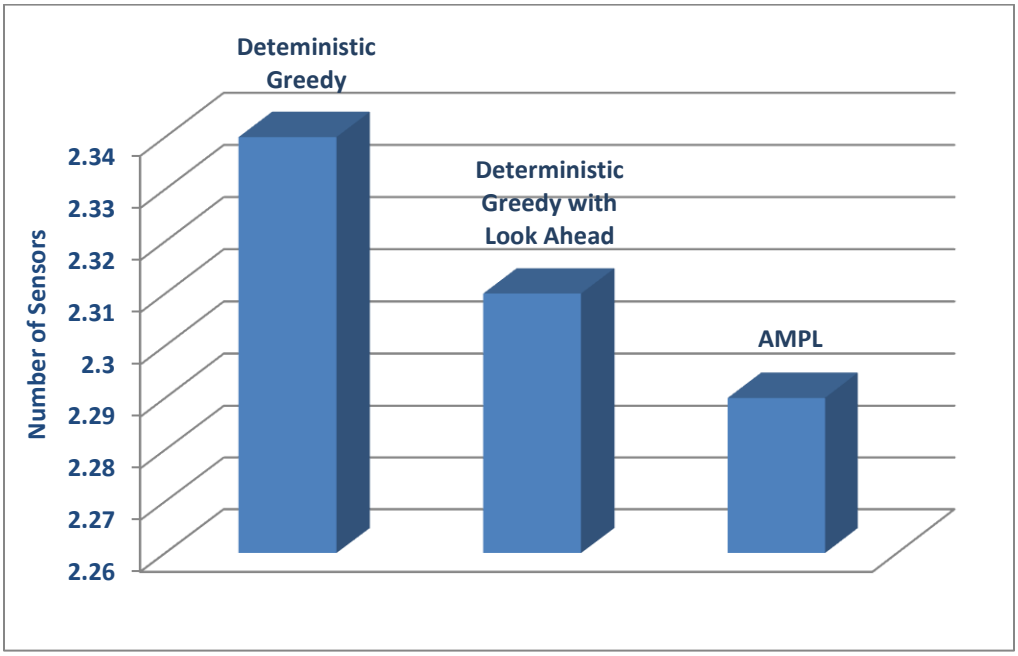

Figure 5.2. Graph comparing the total number of sensors for all three approaches (for a five-node graph).

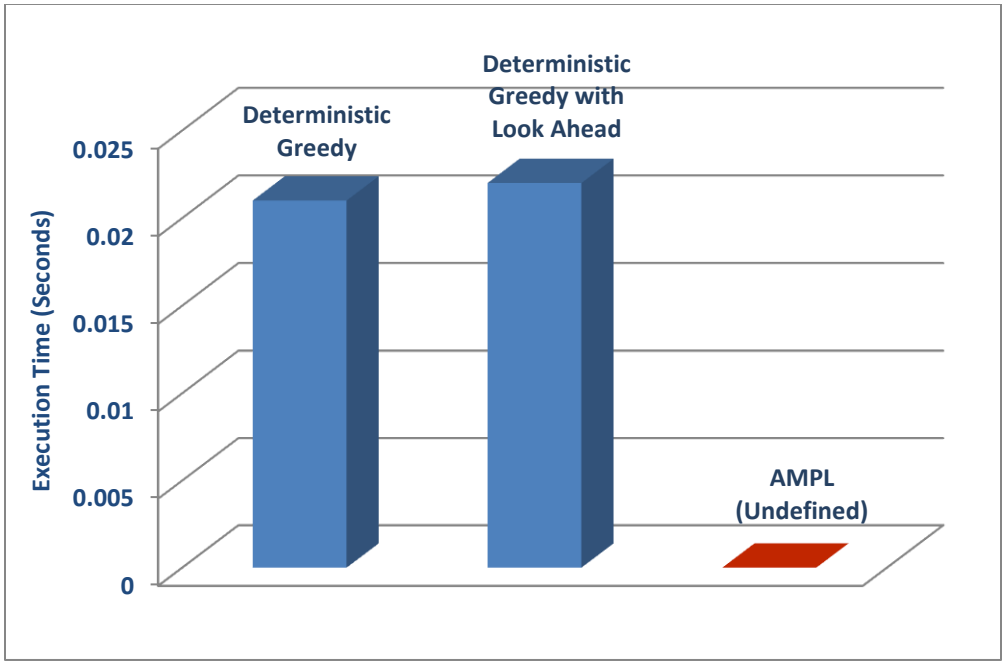

Figure 5.3. Graph comparing the execution time for all three approaches (for a 25-node graph).

5.1.2. Test Case 2

In this test case, a 10-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.4 The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

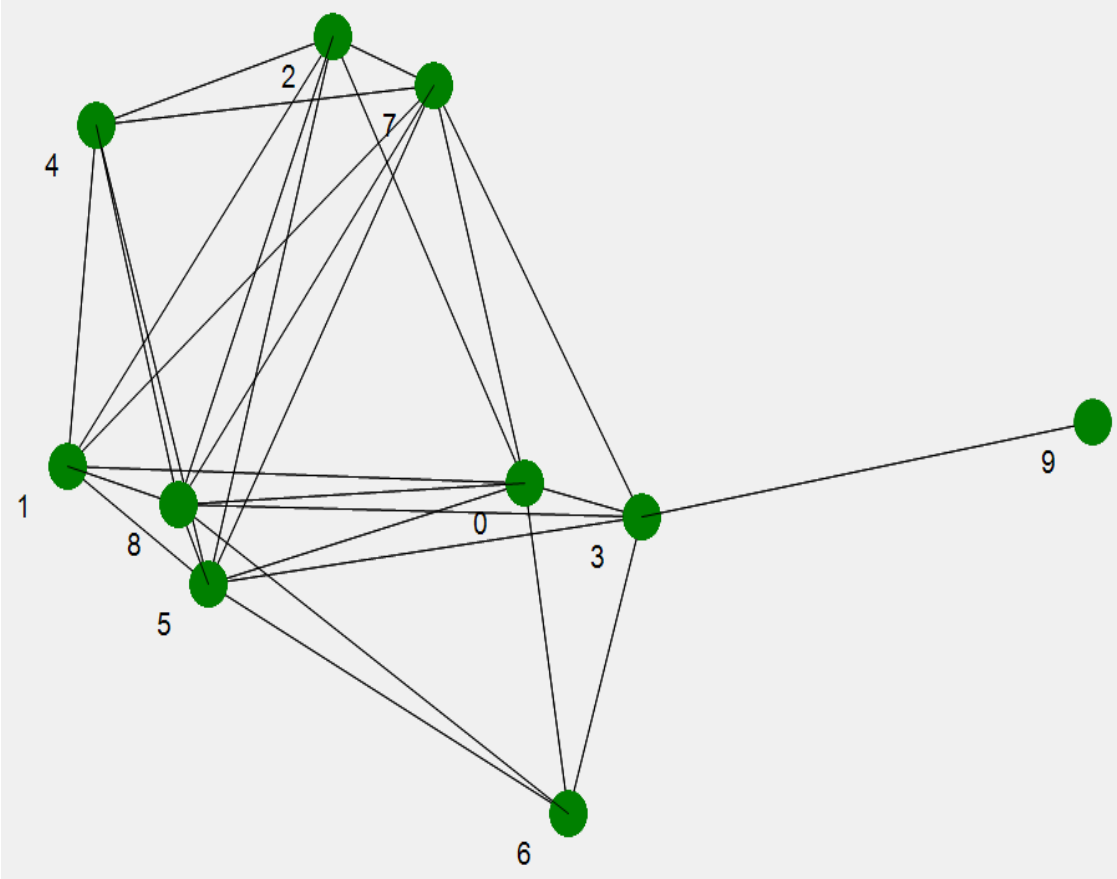

Figure 5.4. A Random 10-Node Network.

Create a network matrix (N_m) from a saved text file and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 2 {5, 9}

Execution Time (in seconds) = 0.023

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 2 {5, 9}

Execution Time (in seconds) = 0.023

Apply the same network matrix (N_m) as input for AMPL using the saved data file

Total Number of Sensors (T_s) = 2 {5, 9}

Execution Time (in seconds) = 0.0

Generating 50 possible replications of the network topology with 10 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for the number of sensors and execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.5 and 5.6.

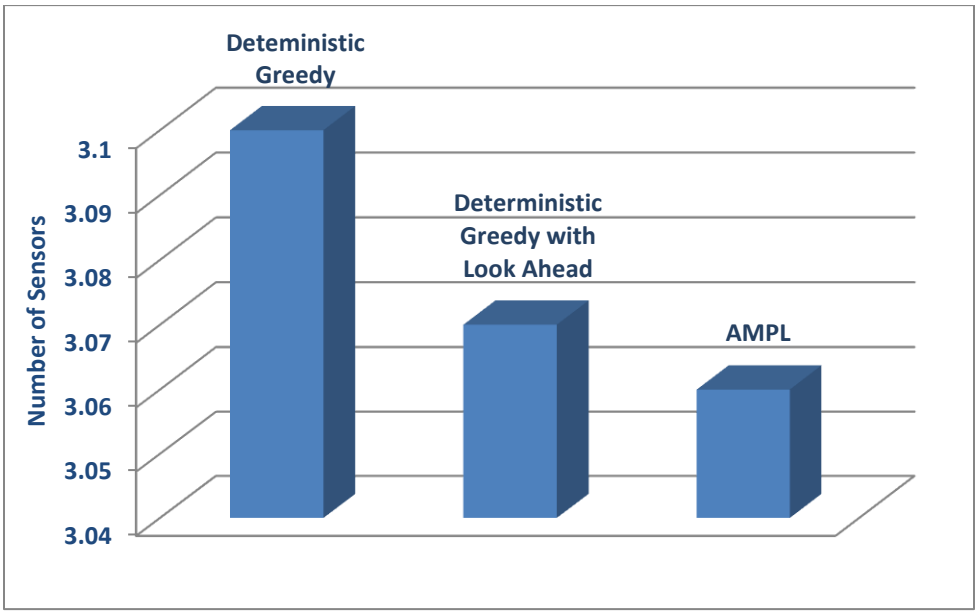

Figure 5.5. Graph comparing the total number of sensors for all three approaches (for a 10 node graph).

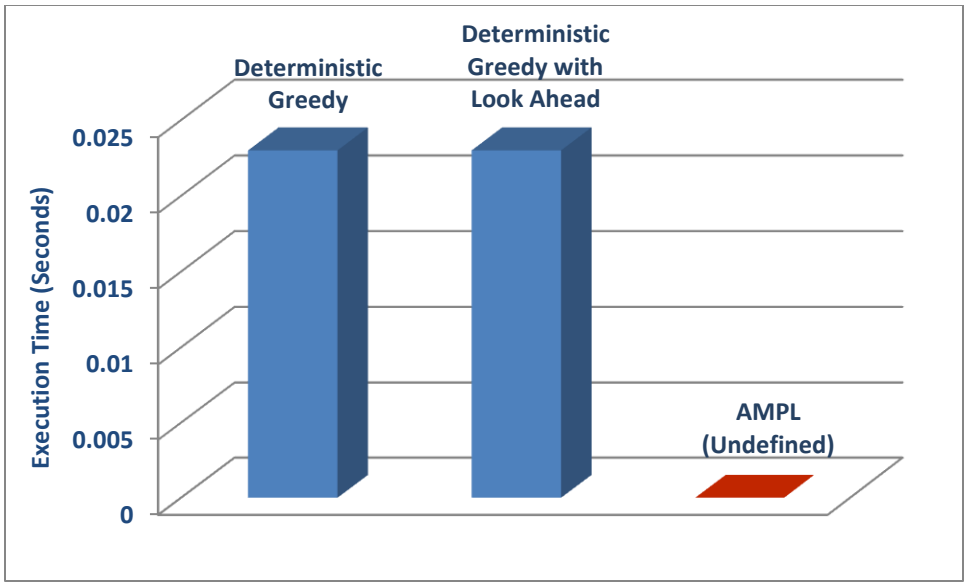

Figure 5.6. Graph comparing the execution time for all three approaches (for a 10-node graph).

5.1.3. Test Case 3

In this test case, a 15-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.7. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

Create a network matrix (N_m) from a saved text file and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 4 {3, 11, 7, 10}

Execution Time (in seconds) = 0.027

Apply the same network matrix (N_m) as input for Algorithm 4.3

Total Number of Sensors (T_s) = 4 {3, 11, 0, 14}

Execution Time (in seconds) = 0.031

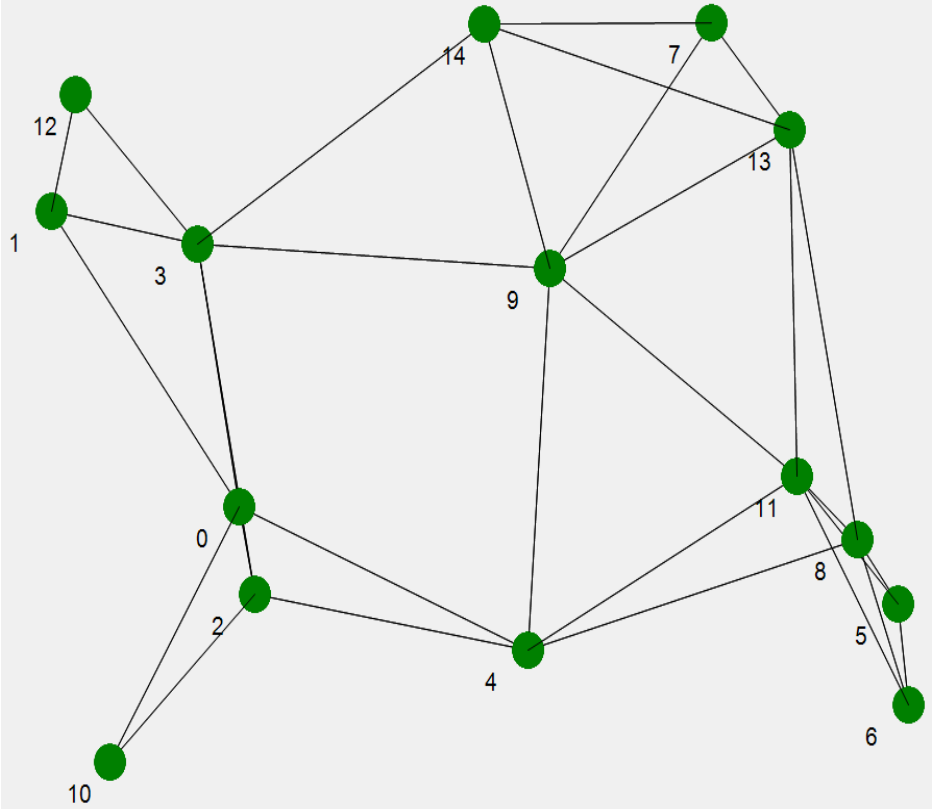

Figure 5.7. A Random 15-Node Network.

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 4 {11, 3, 14, 10}

Execution Time (in seconds) = 0.0

Note: In the above graph, there is a connection between nodes 3 and 2 which is not visible due to the connection between nodes 3 and 0.

Generating 50 possible replications of the network topology for 15 nodes, the mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing mean values for the number of sensors and execution time of AMPL with the corresponding mean values for the total number of sensors and execution times of Algorithms 4.2 and 4.3 are generated as shown in Figures 5.8 and 5.9.

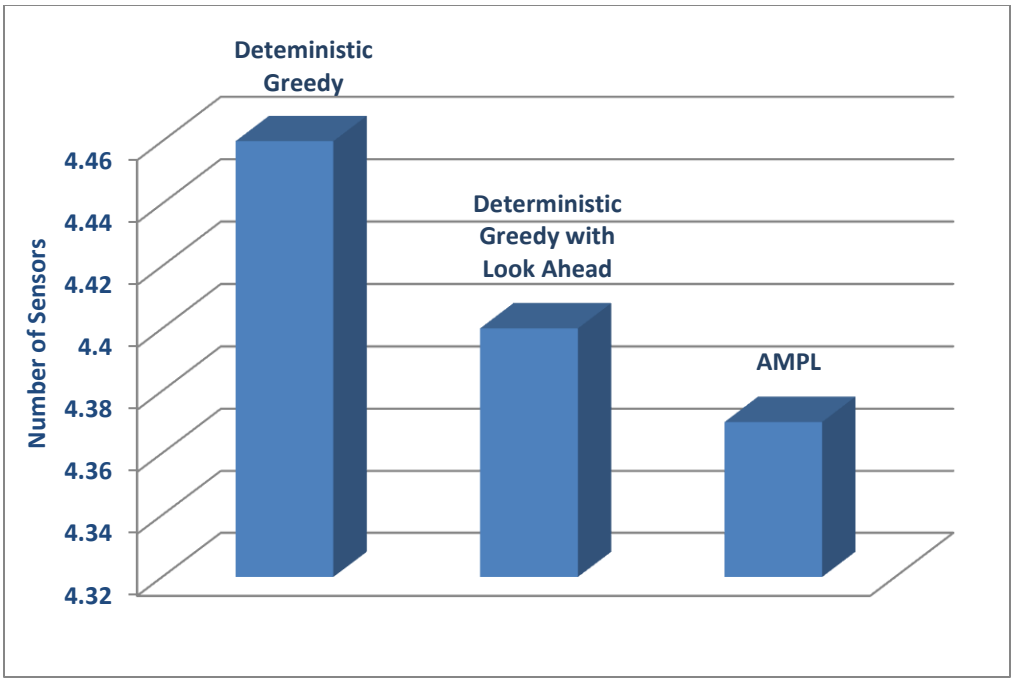

Figure 5.8. Graph comparing the total number of sensors for all three approaches (for a 15-node graph).

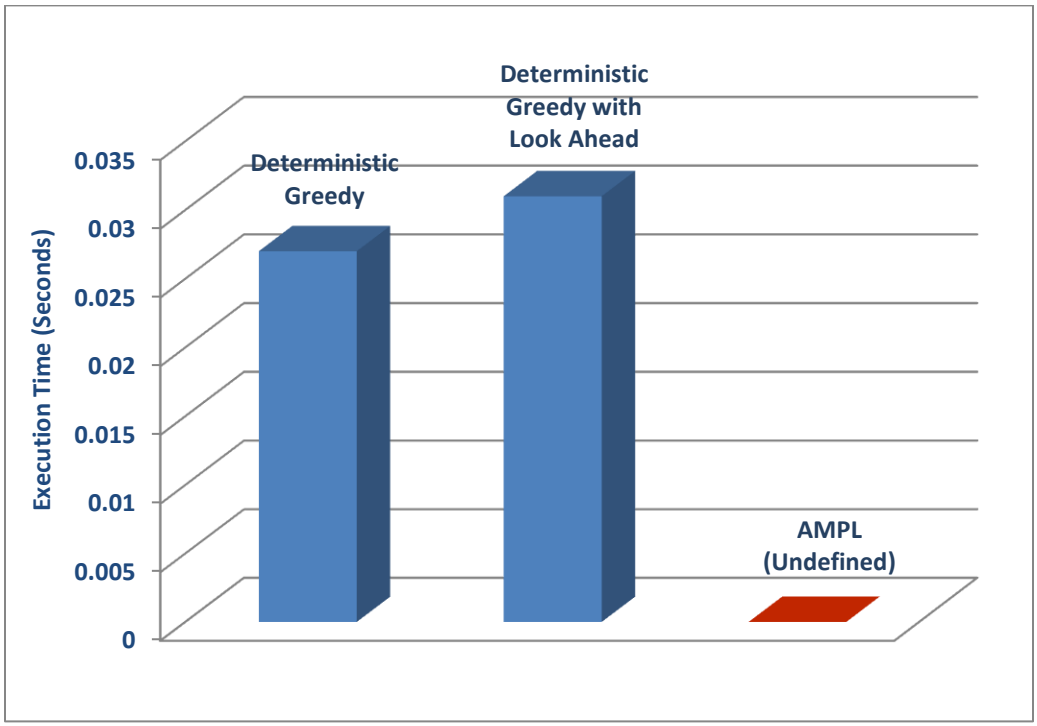

Figure 5.9. Graph comparing the execution time for all three approaches (for a 15-node graph).

5.1.4. Test Case 4

In this test case, a 20-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.10. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

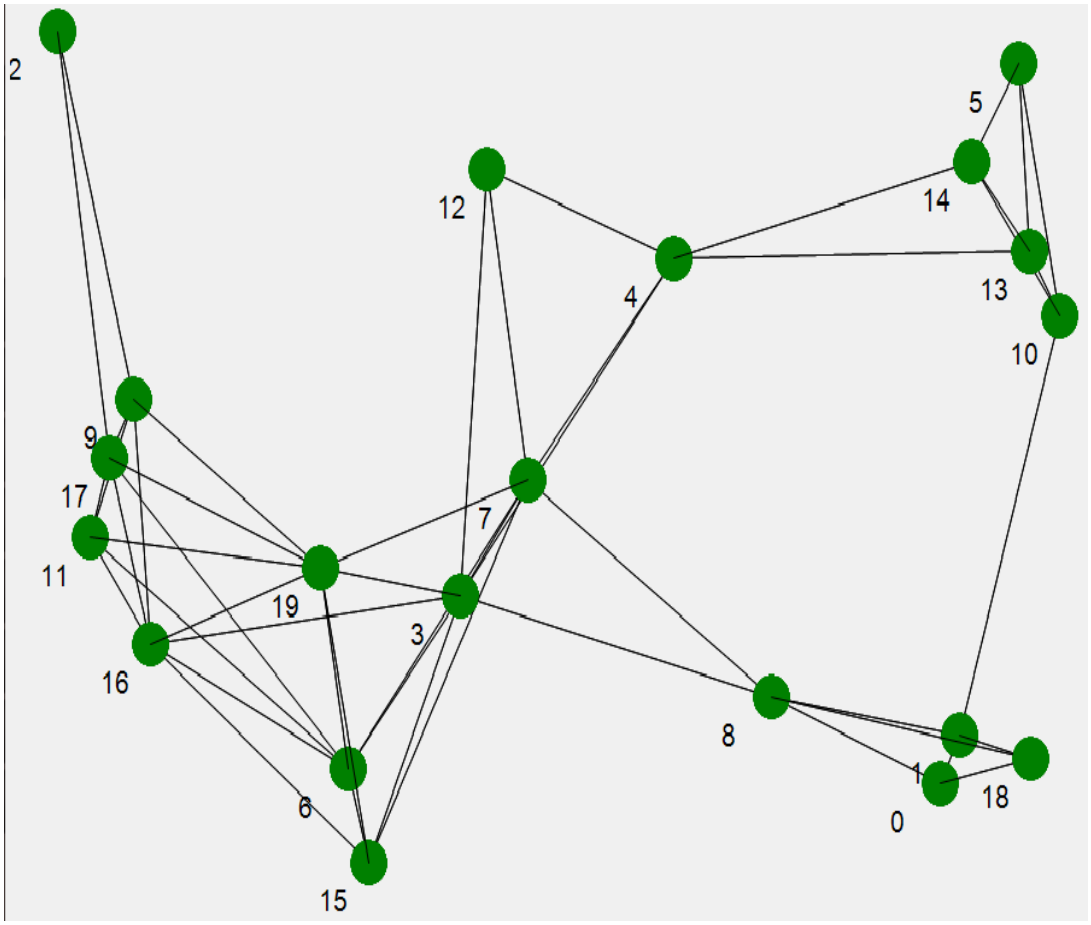

Figure 5.10. A Random 20-Node Network.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 4 {3, 10, 17, 0}

Execution Time (in seconds) = 0.031

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 4 {3, 10, 9, 18}

Execution Time (in seconds) = 0.031

Apply the same network matrix (N_m) as input for AMPL using the the saved data file.

Total Number of Sensors (T_s) = 4

Execution Time (in seconds) = 0.0

Generating 50 possible replications of the network topology of 20 nodes, a mean for total number of sensors for all possible network combinations is calculated. Graphs comparing mean values of Number of sensors and Execution time of AMPL with corresponding mean values of total number of sensors and Execution times of Algorithm 4.2 and Algorithm 4.3 are generated as shown in the Figure 5.11 and 5.12.

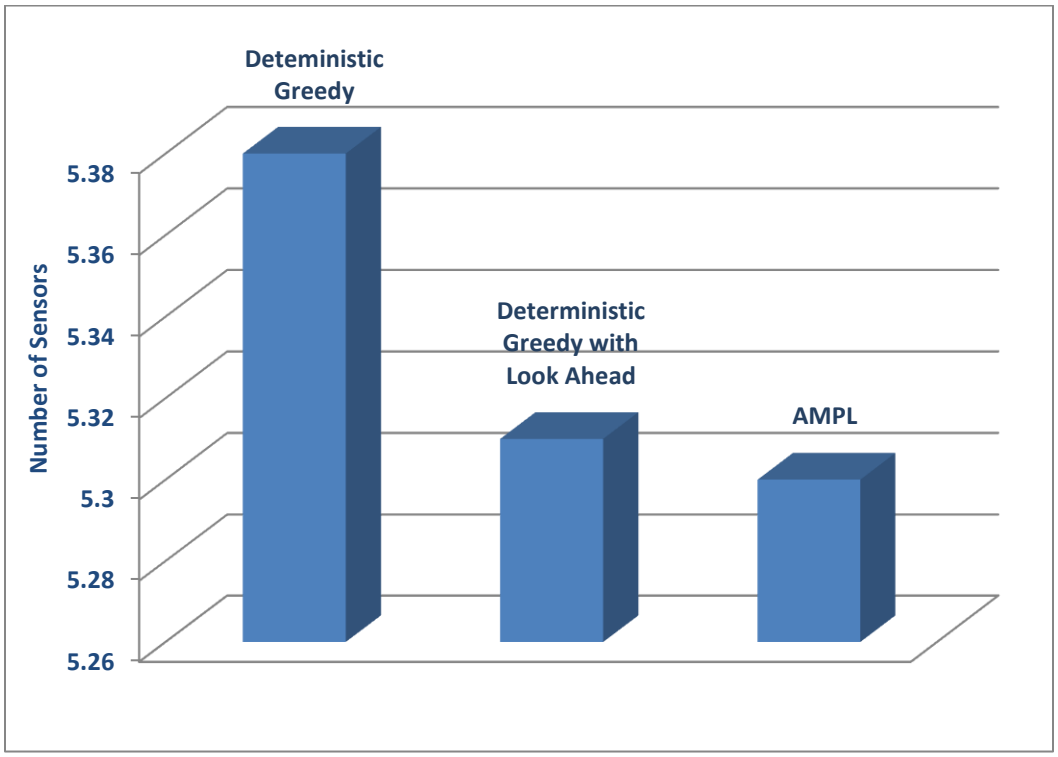

Figure 5.11. Graph comparing the total number of sensors for all three approaches (for a 20 node graph).

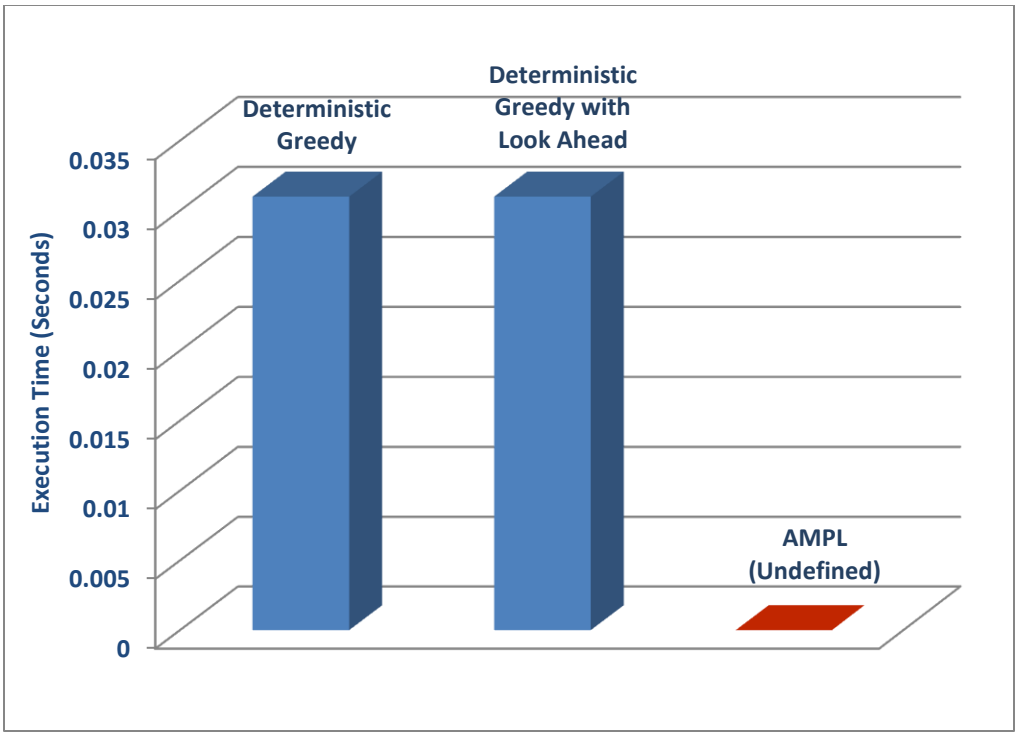

Figure 5.12. Graph comparing the execution time for all three approaches (20-node graph).

5.1.5. Test Case 5

In this test case, a 25-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.13. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 6 {24, 12, 16, 3, 6, 21}

Execution Time (in seconds) = 0.033

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 6 {24, 12, 10, 23, 3, 6}

Execution Time (in seconds) = 0.035

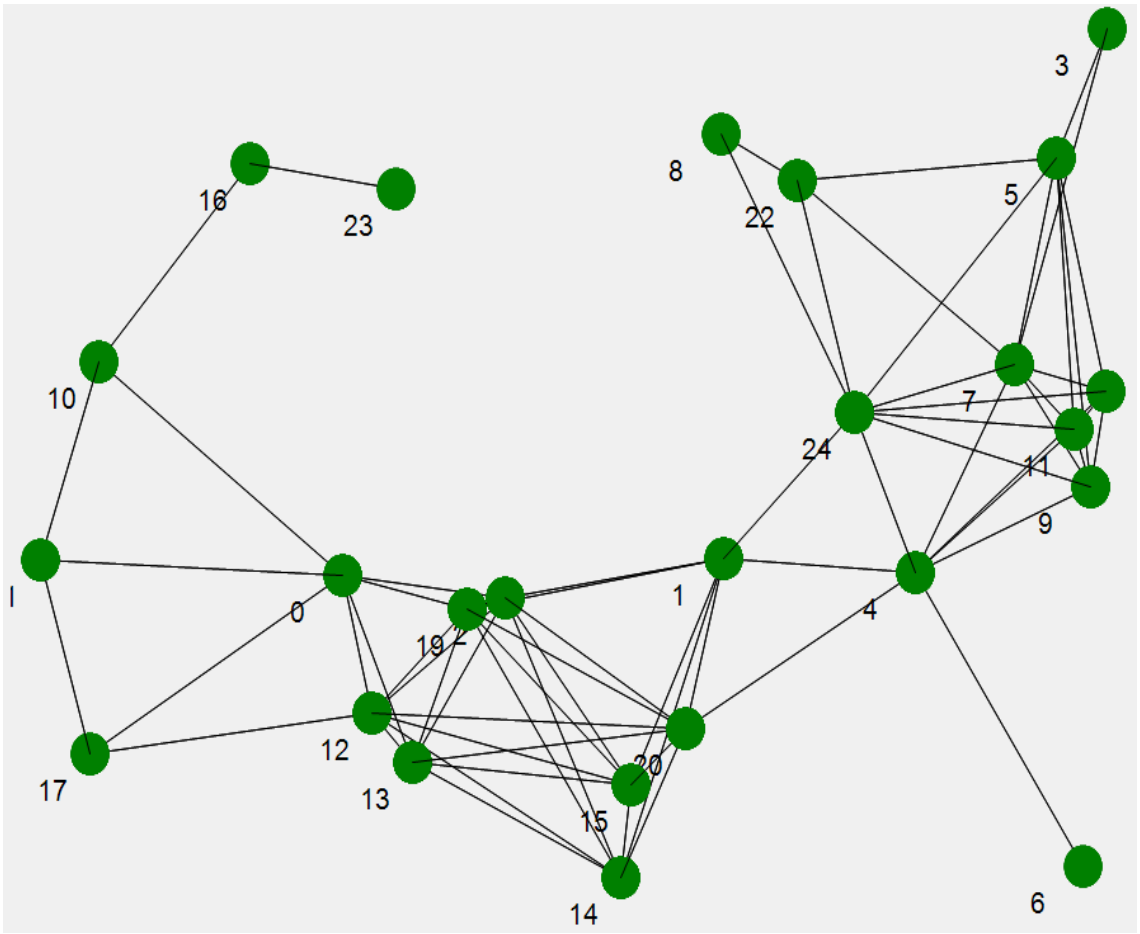

Figure 5.13. A Random 25-Node Network.

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 6

Execution Time (in seconds) = 0.0

Generating 50 possible replications of the network topology with 25 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing mean values of the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.14 and 5.15.

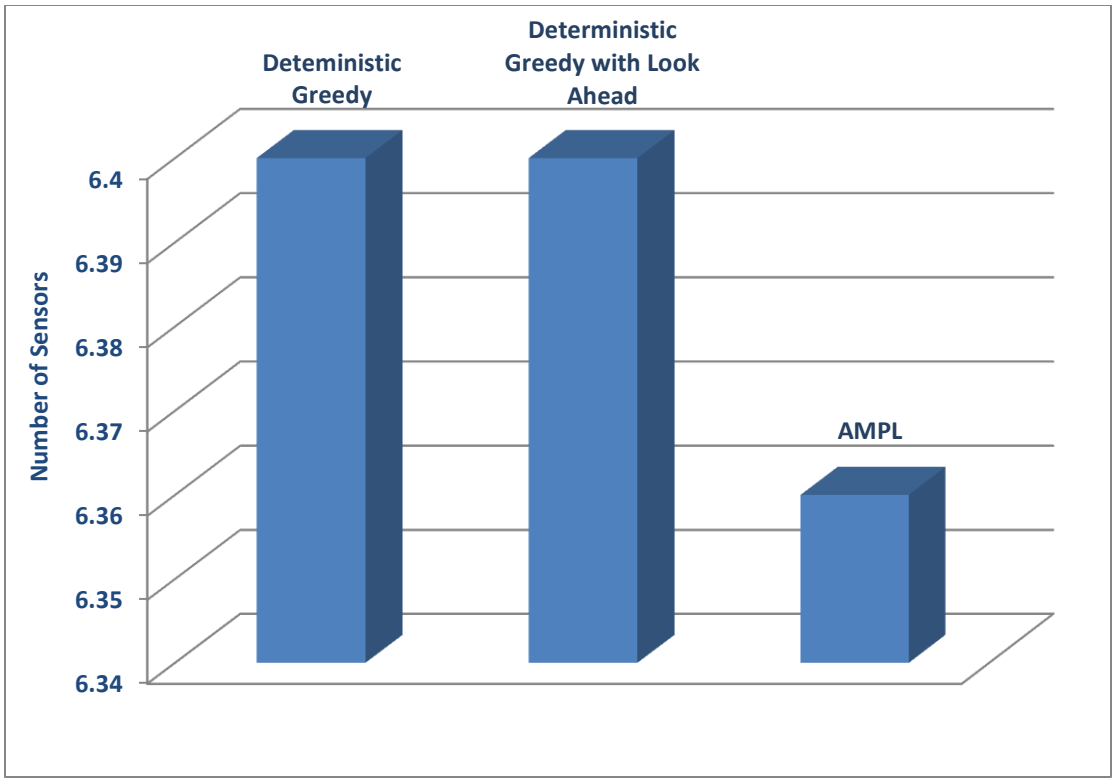

Figure 5.14. Graph comparing the total number of sensors for all three approaches (25-node graph).

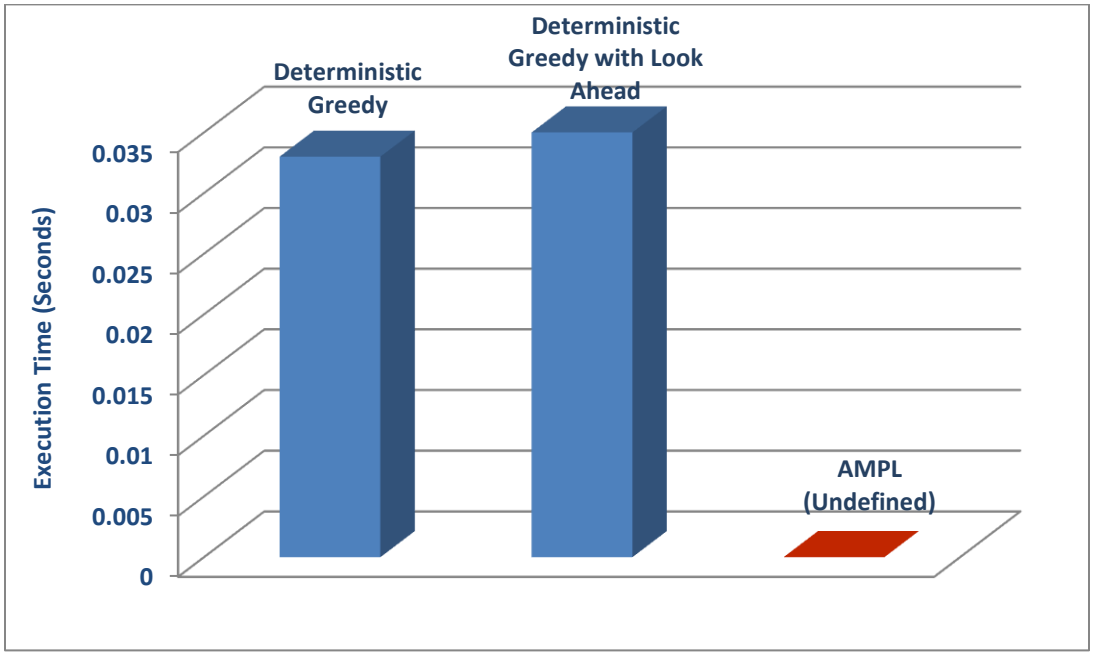

Figure 5.15. Graph comparing the execution time for all three approaches (25-node graph).

5.1.6. Test Case 6

In this test case, a 30-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.16. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

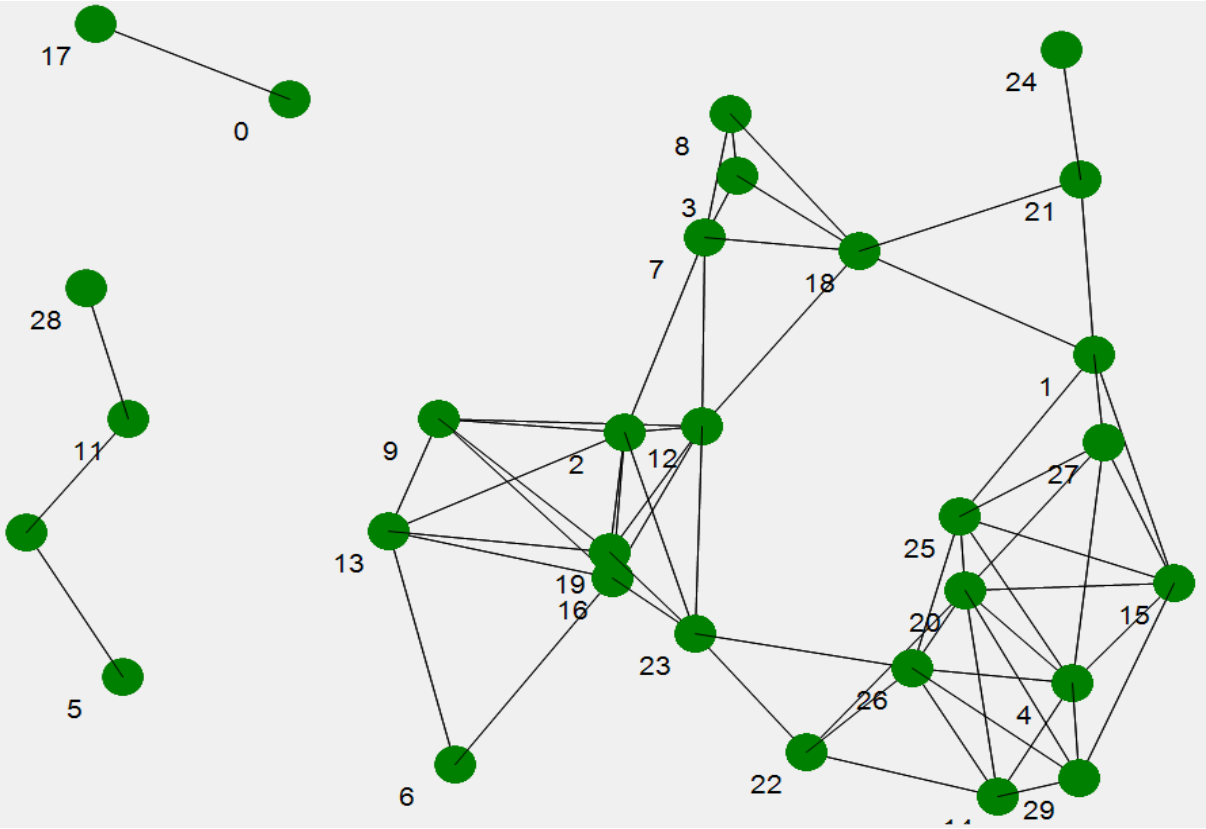

Figure 5.16. A Random 30-Node Network.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 7 {20, 16, 18, 10, 0, 24, 28}

Execution Time (in seconds) = 0.033

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 7 {20, 16, 18, 10, 17, 11, 24}

Execution Time (in seconds) = 0.039

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 7

Execution Time (in seconds) = 0.0

Generating 50 possible replications for the network topology of 30 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values of the number of sensors and execution time of AMPL with the corresponding mean values for the total number of sensors and execution times of Algorithms 4.2 and 4.3 are generated as shown in Figures 5.17 and 5.18.

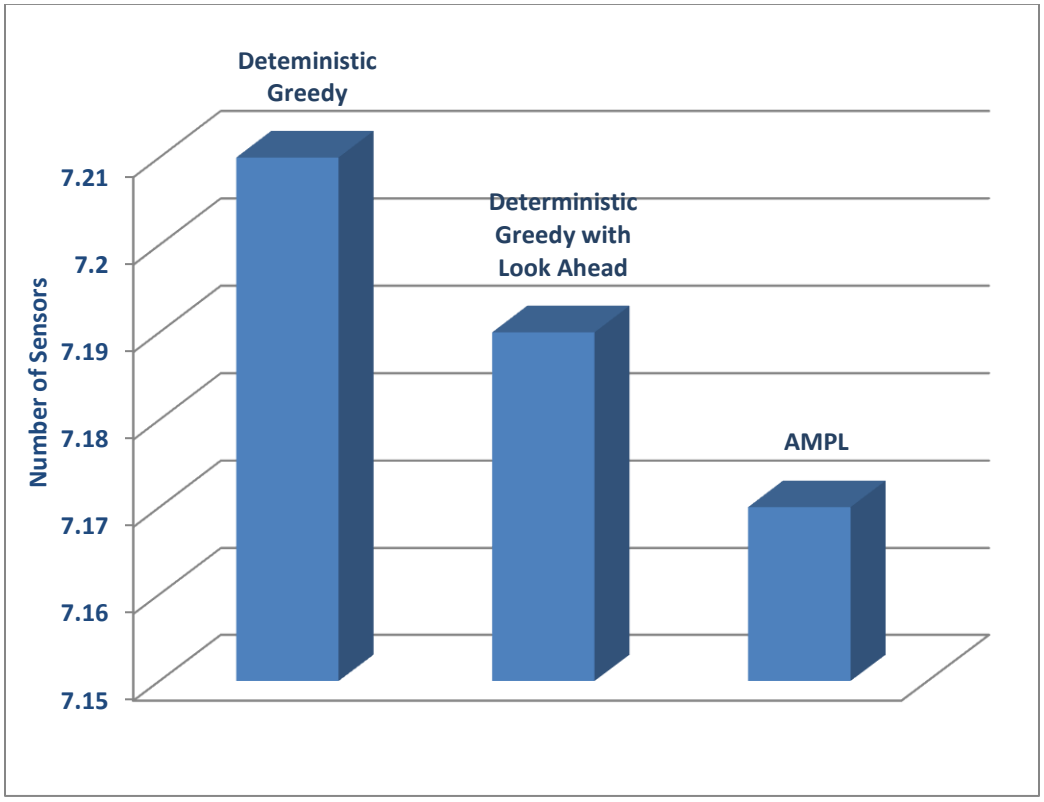

Figure 5.17. Graph comparing the total number of sensors for all three approaches (30-node graph).

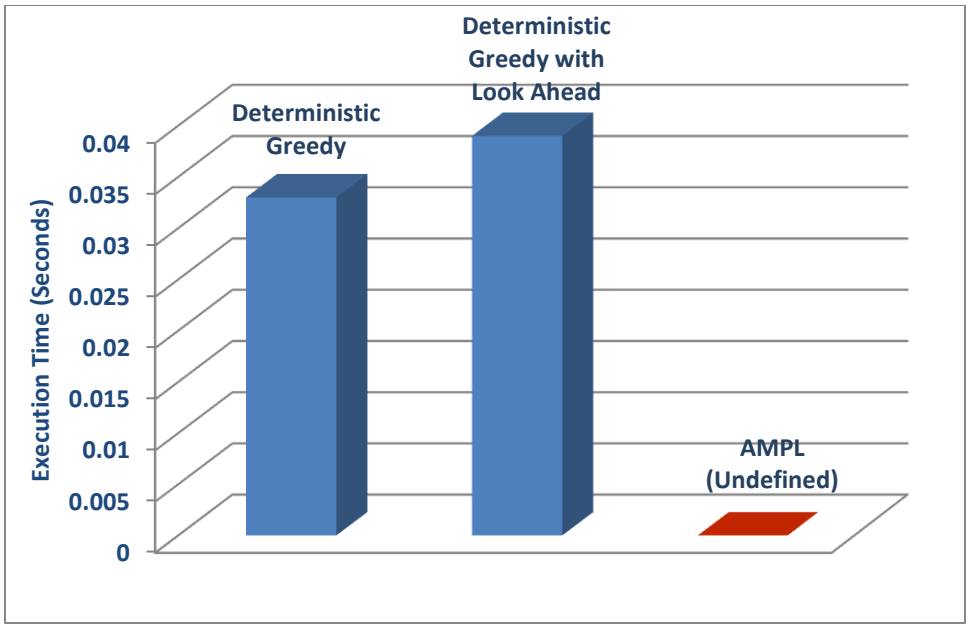

Figure 5.18. Graph comparing the execution time for all three approaches (30-node graph).

5.1.7. Test Case 7

In this test case, a 35-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.19. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 7 {15, 24, 29, 0, 4, 12, 34}

Execution Time (in seconds) = 0.037

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 7 {15, 24, 29, 0, 4, 34, 12}

Execution Time (in seconds) = 0.041

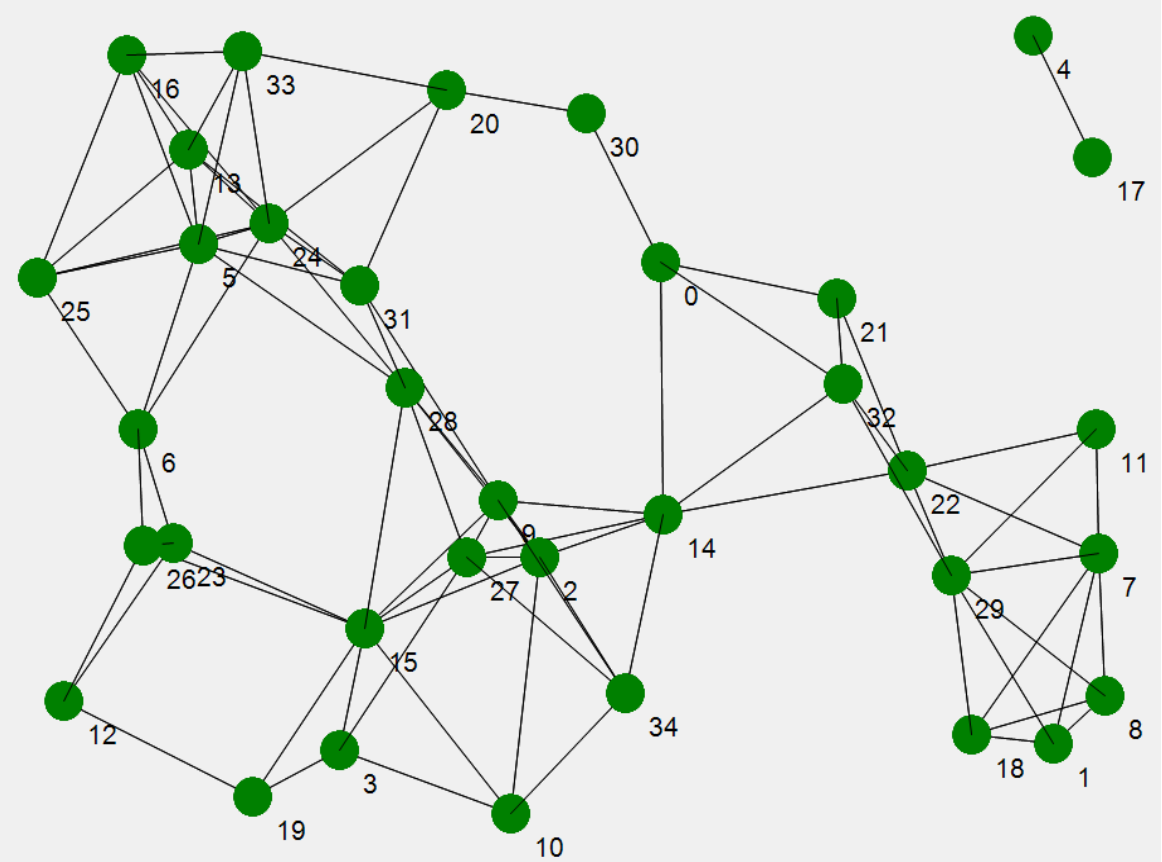

Figure 5.19. A Random 35-Node Network.

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 7

Execution Time (in seconds) = 0.0

Generating 50 possible replications for the network topology of 35 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values of the number of sensors and execution time of AMPL with the corresponding mean values for the total number of sensors and execution times of Algorithms 4.2 and 4.3 are generated as shown in Figures 5.20 and 5.21.

Note: Execution time of AMPL cannot be determined because of the indefinite time interval between any two loads.

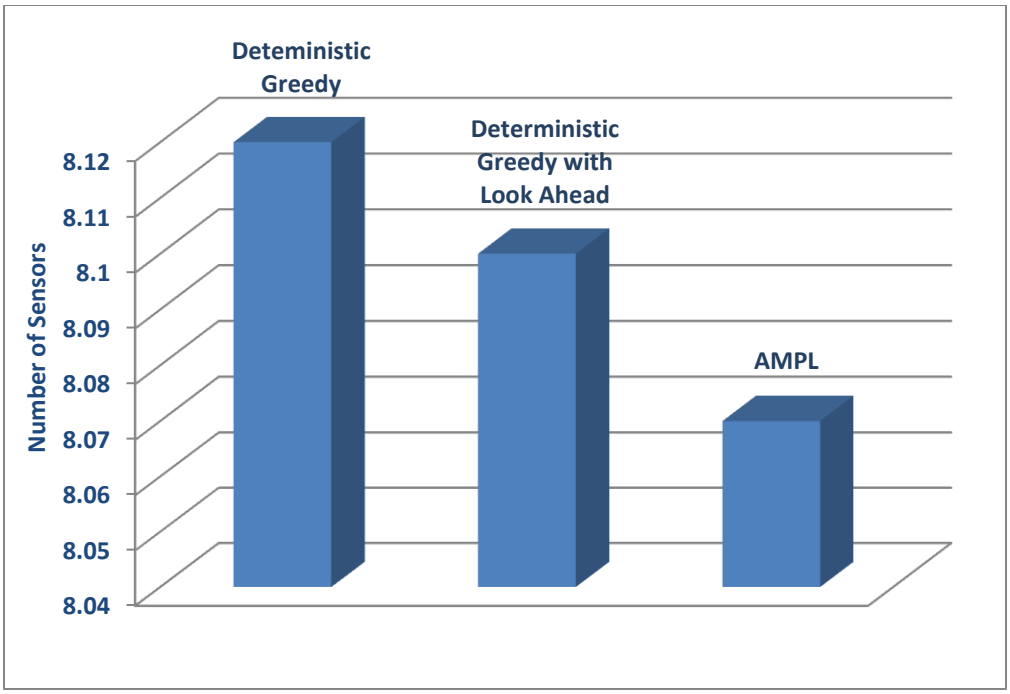

Figure 5.20. Graph comparing the total number of sensors for all three approaches (35-node graph).

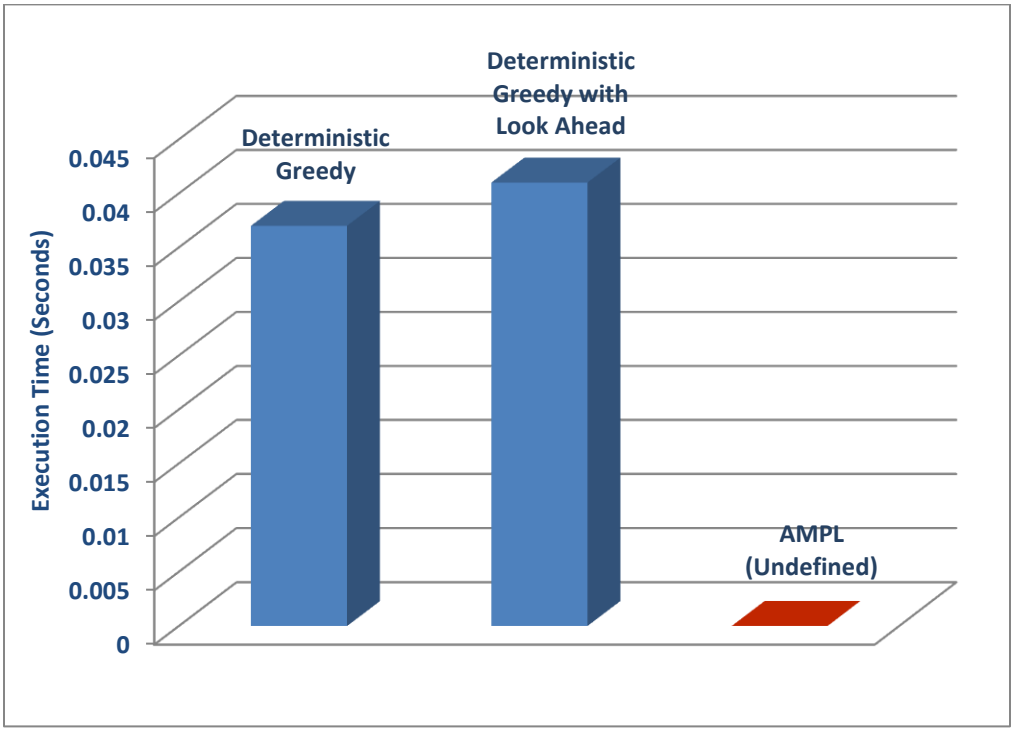

Figure 5.21. Graph comparing the execution time for all three approaches (35-node graph).

5.1.8. Test Case 8

In this test case, a 40-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.22. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

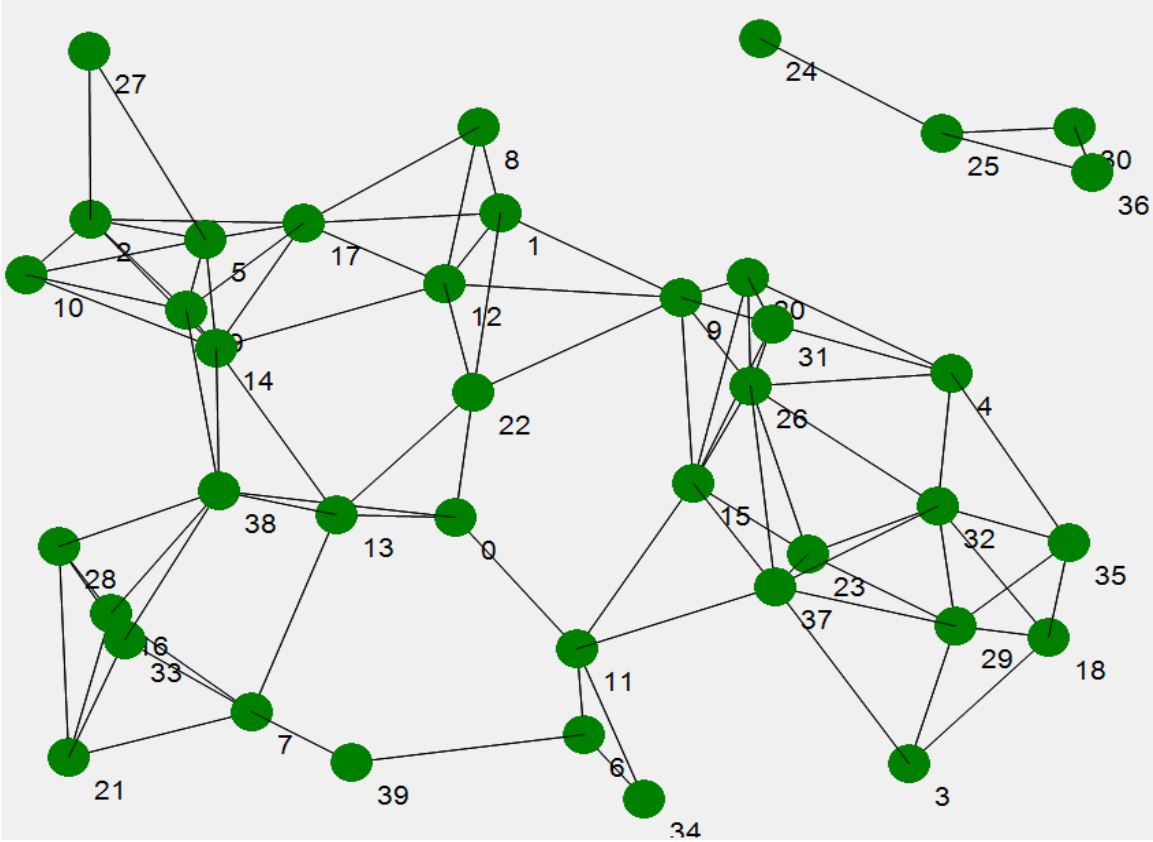

Figure 5.22. A Random 40-Node Network.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 9 {14, 26, 7, 25, 11, 29, 1, 27, 28}

Execution Time (in seconds) = 0.043

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 8 {26, 7, 29, 11, 25, 1, 38, 2}

Execution Time (in seconds) = 0.052

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 8

Execution Time (in seconds) = 0.0

Generating 50 possible replications for the network topology of 40 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.23 and 5.24.

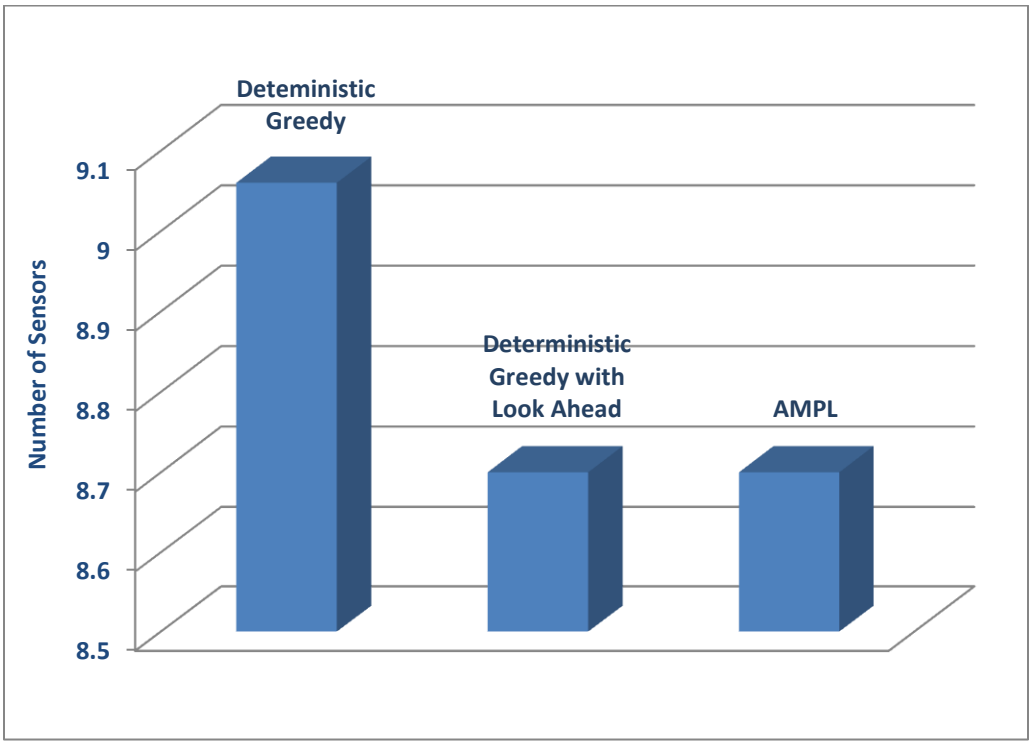

Figure 5.23. Graph comparing the total number of sensors for all three approaches (40-node graph).

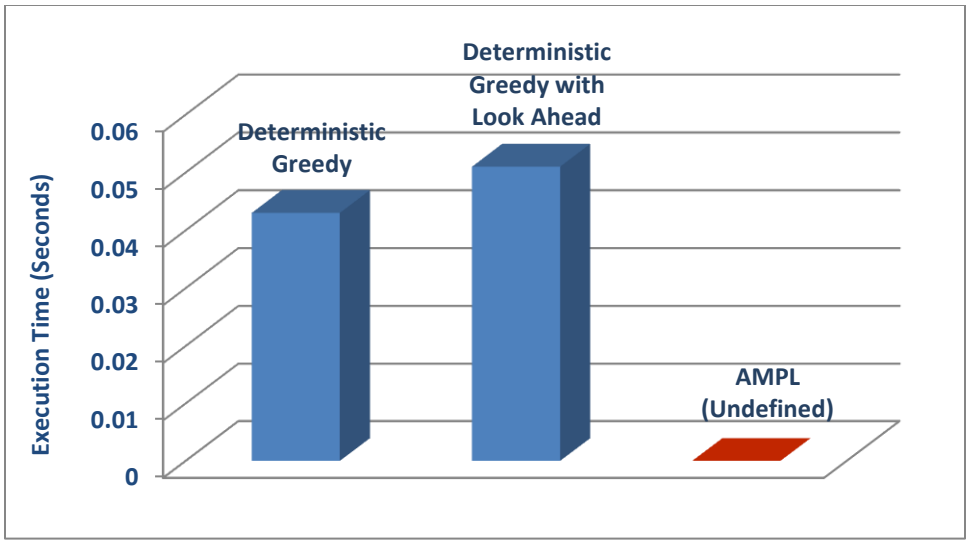

Figure 5.24. Graph comparing the execution time for all three approaches (40-node graph).

5.1.9. Test Case 9

In this test case, a 45-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.25. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

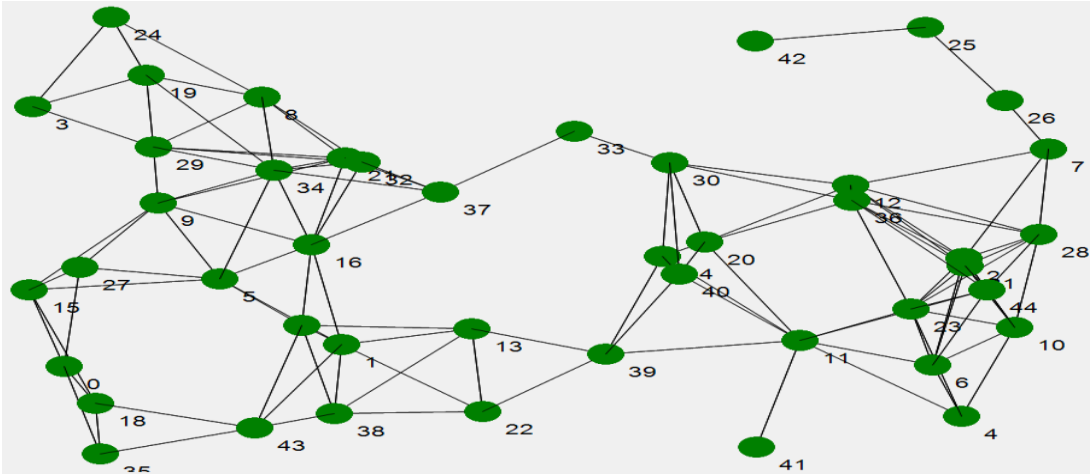

Figure 5.25. A Random 45-Node Network.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 8 {11, 34, 2, 1, 0, 25, 30, 3}

Execution Time (in seconds) = 0.046

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 8 {2, 34, 11, 1, 0, 25, 3, 33}

Execution Time (in seconds) = 0.0586

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = 8

Execution Time (in seconds) = 0.0

Generating 50 possible replications for the network topology of 45 nodes, a mean for the total number of sensors for all possible network combinations is calculated.

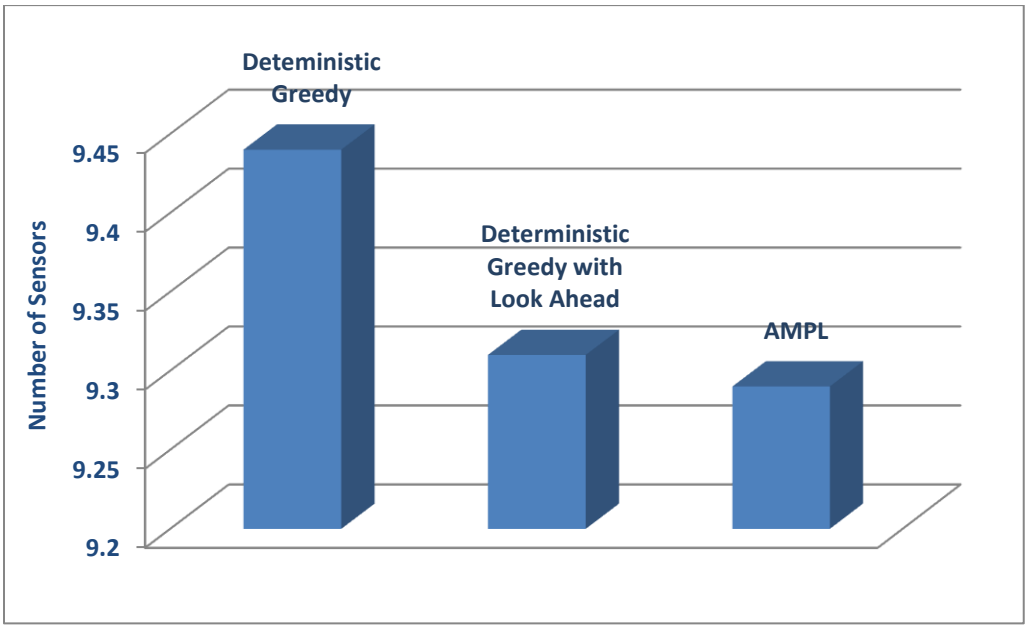

Figure 5.26. Graph comparing the total number of sensors for all three approaches (45-node graph).

Graphs comparing mean values for the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in the Figures 5.26 and 5.27.

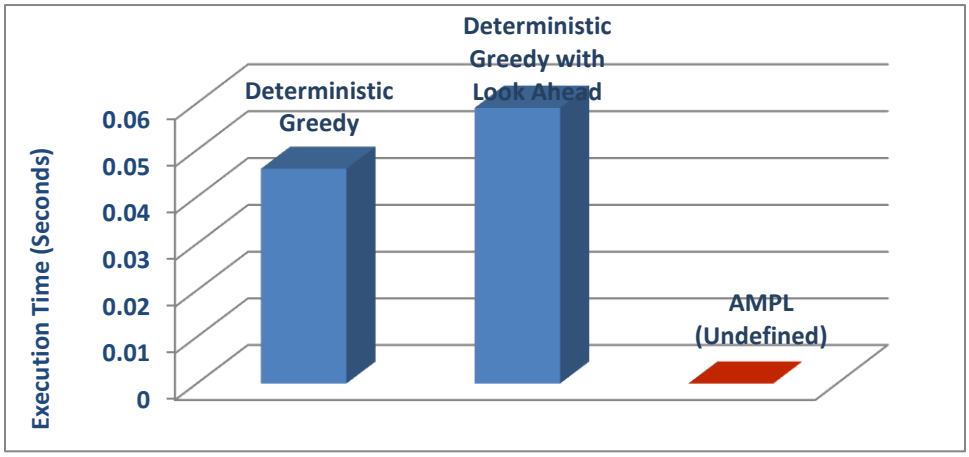

Figure 5.27. Graph comparing the execution time for all three approaches (45-node graph).

5.1.10. Test Case 10

In this test case, a 50-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.28. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

Create a network matrix (N_m) from a saved text file, and apply the N_m matrix as input for Algorithm 4.2.

Total Number of Sensors (T_s) = 11 {29, 22, 18, 15, 3, 31, 10, 4, 8, 21, 30}

Execution Time (in seconds) = 0.049

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 11{29, 22, 18, 15, 3, 48, 0, 35, 8, 30, 21}

Execution Time (in seconds) = 0.059

Apply the same network matrix (N_m) as input for AMPL using the saved data file

Total Number of Sensors (T_s) = 11

Execution Time (in seconds) = 0.0

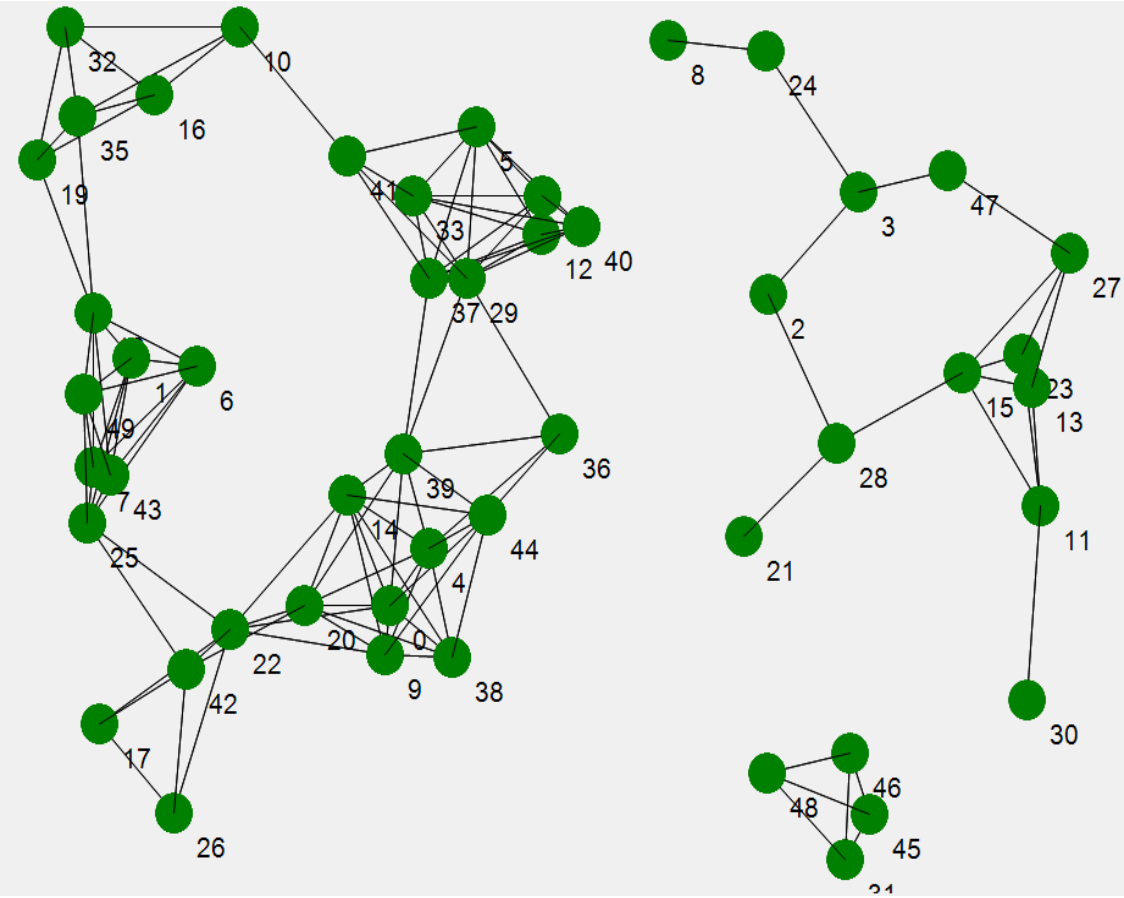

Figure 5.28. A Random 50-Node Network.

Generating 50 possible replications for the network topology of 50 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.29 and 5.30.

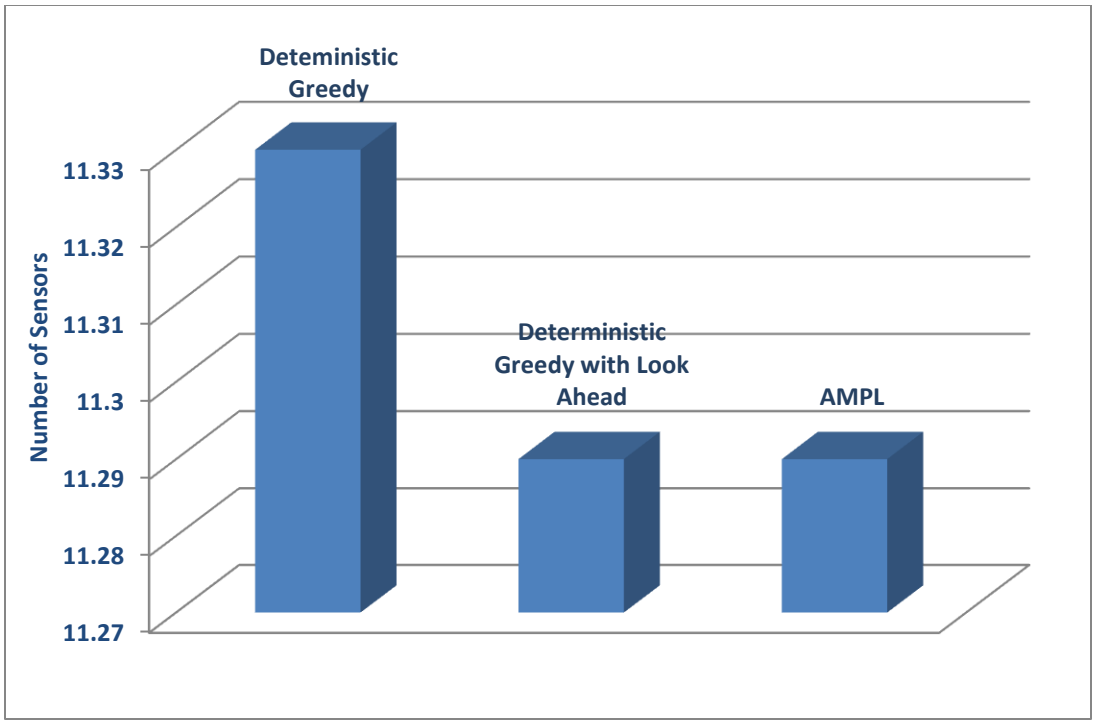

Figure 5.29. Graph comparing the total number of sensors for all three approaches (50-node graph).

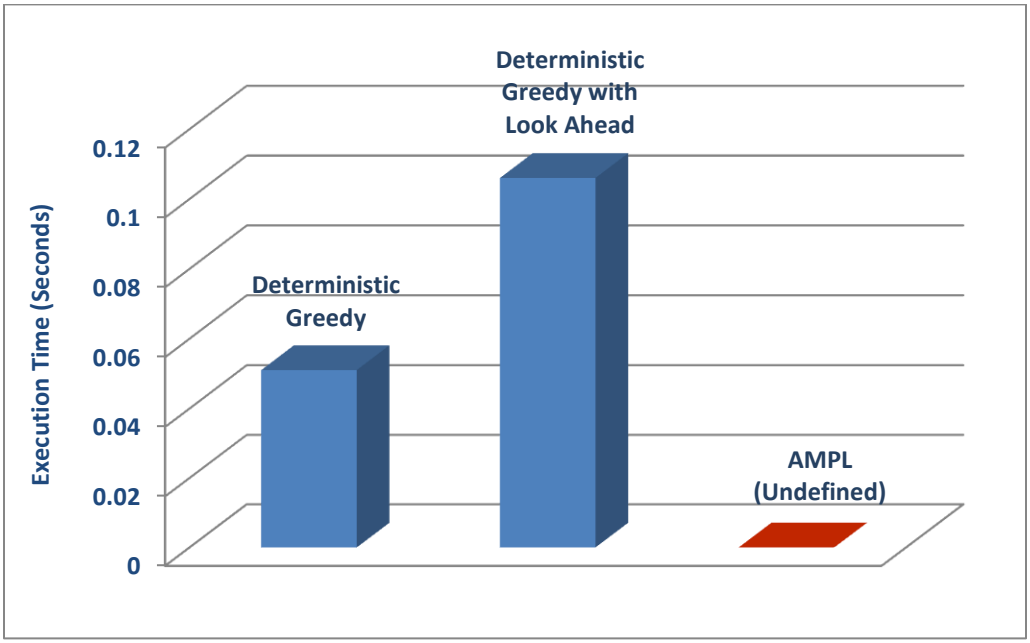

Figure 5.30. Graph comparing the execution time for all three approaches (50-node graph).

5.1.11. Test Case 11

In this test case, a 500-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.31. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus.

Because of the variable and constraint limitations for AMPL, we cannot obtain a solution for the 500-node graph from AMPL. Algorithms 4.2 and 4.3 obtained solutions successfully.

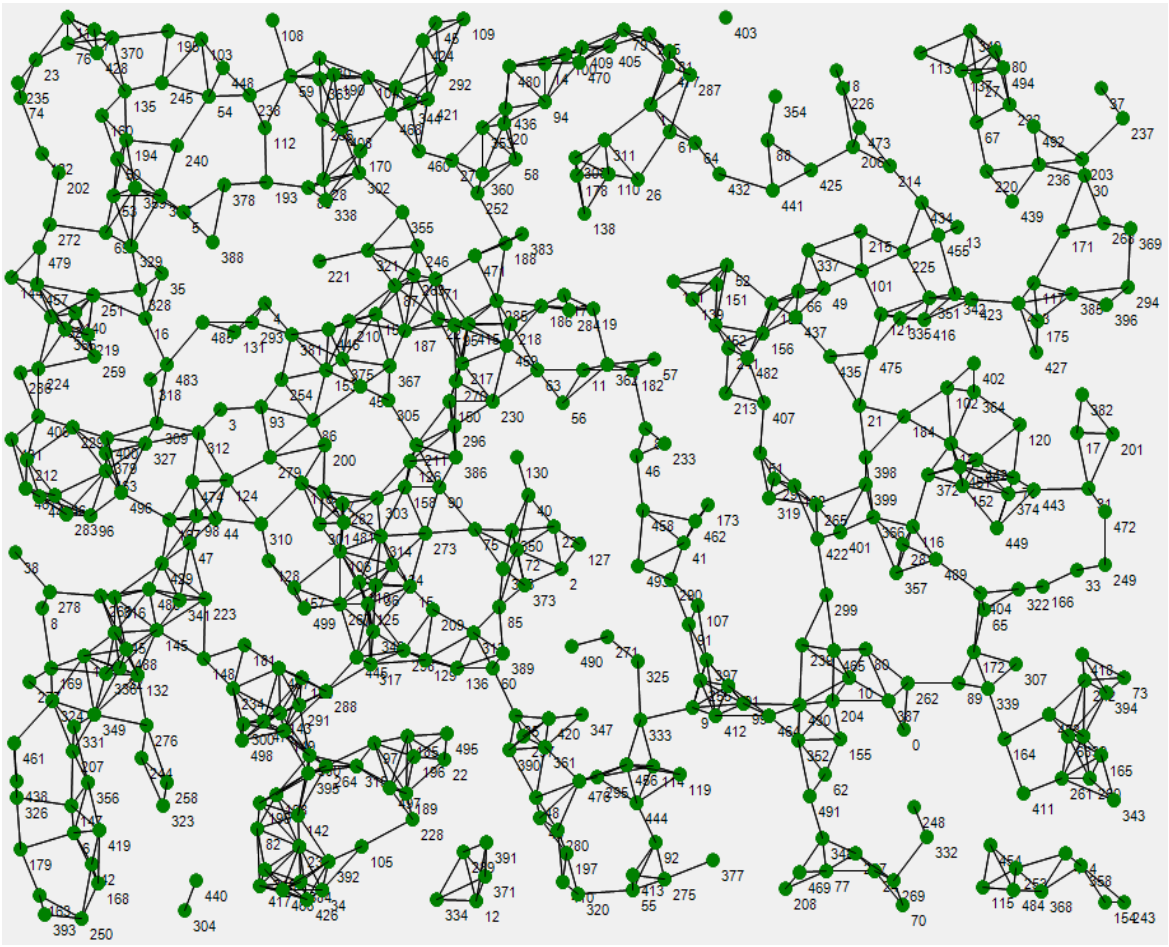

Figure 5.31. A Random 500-Node Network.

Applying the 500 node graph on Algorithm 4.2:

Total Number of Sensors (T_s) = 110

Execution Time (in seconds) = 7.97 Secs

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 110

Execution Time (in seconds) = 11 Secs

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = Undefined

Execution Time (in seconds) = Undefined

Generating 50 possible replications for the network topology of 500 nodes, a mean for the total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.32 and 5.33.

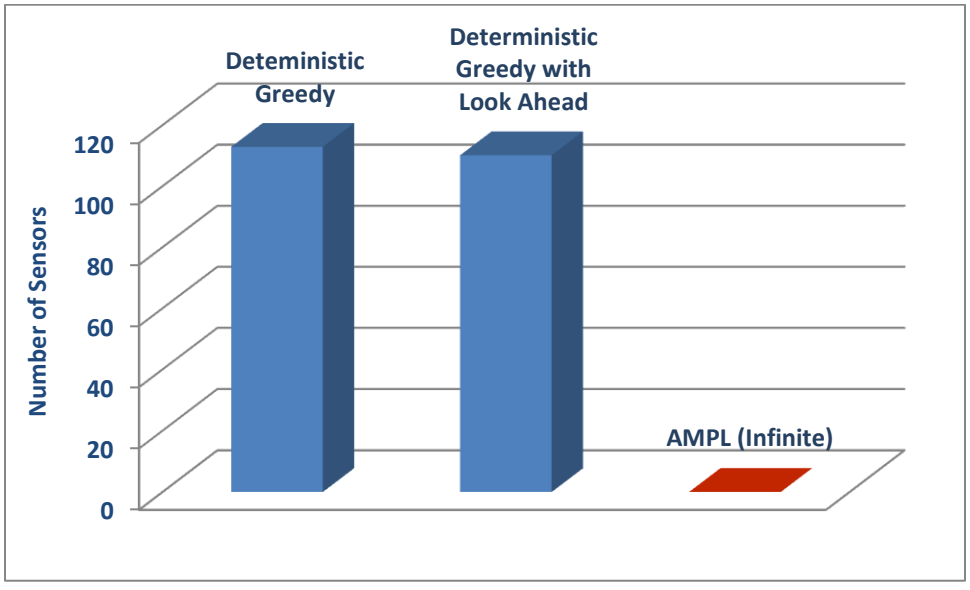

Figure 5.32. Graph comparing the total number of sensors for all three approaches (500-node graph).

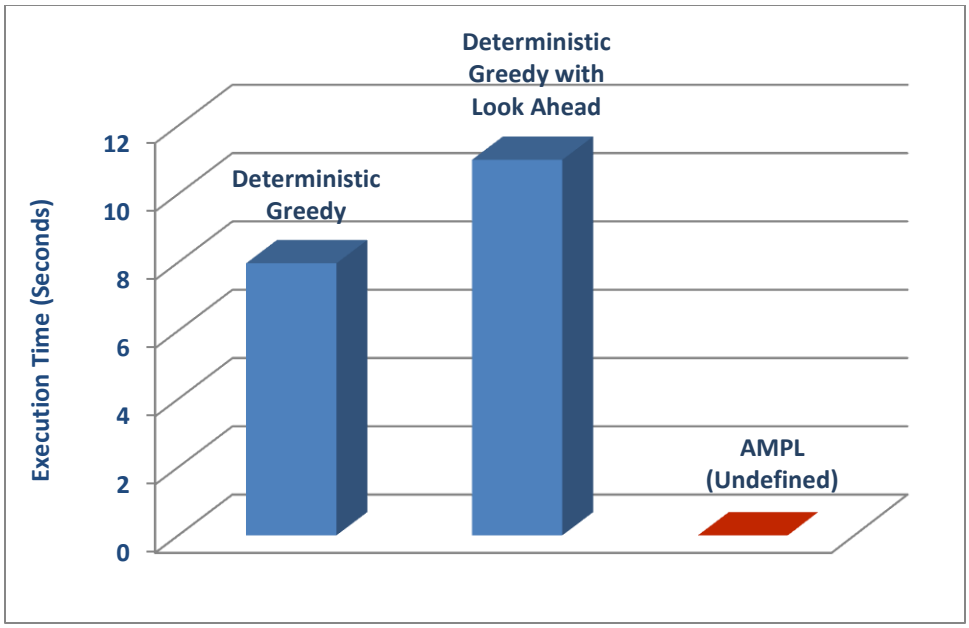

Figure 5.33. Graph comparing the execution time for all three approaches (500-node graph).

5.1.12. Test Case 12

In this test case, a 1000-node graph is generated randomly using the random-graph generation technique discussed in Section 4.1.1. The resultant graph is shown in Figure 5.34. The resultant network matrix is saved as a text file and a data file (file format accepted by AMPL as input) to apply input for Algorithms 4.2 and 4.3, and AMPL-Plus. Because of the variable and constraint limitations of AMPL, we cannot obtain a solution for the 1000-node graph from AMPL. Conversely, Algorithms 4.2 and 4.3 obtained solutions successfully.

Applying the 1000 node graph on Algorithm 4.2:

Total Number of Sensors (T_s) = 188

Execution Time (in seconds) = 15 Secs

Apply the same network matrix (N_m) as input for Algorithm 4.3.

Total Number of Sensors (T_s) = 183

Execution Time (in seconds) = 37 Secs

Apply the same network matrix (N_m) as input for AMPL using the saved data file.

Total Number of Sensors (T_s) = Undefined

Execution Time (in seconds) = Undefined

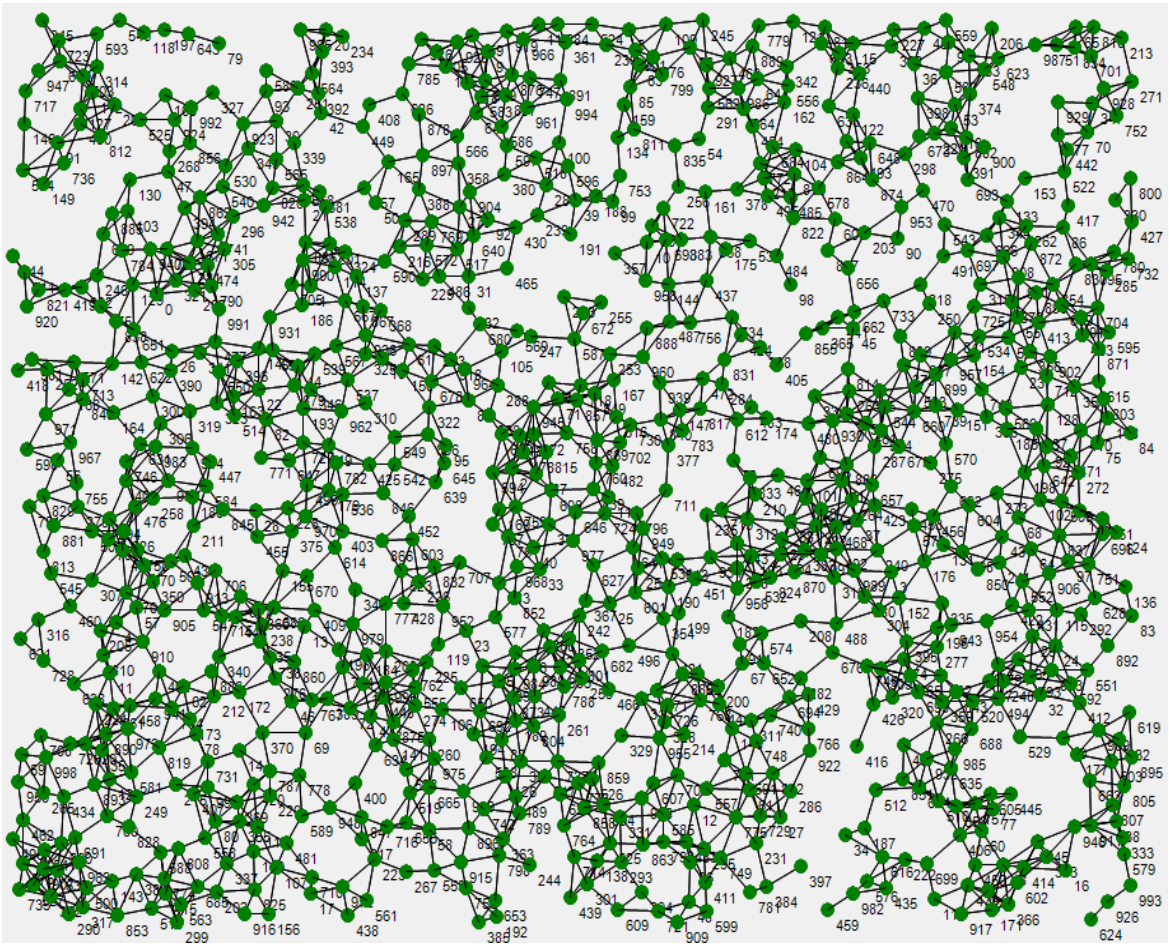

Figure 5.34. A Random 1000-Node Network.

Generating 50 possible replications for the network topology of 1000 nodes, a mean for total number of sensors for all possible network combinations is calculated. Graphs comparing the mean values for the number of sensors and the execution time of AMPL with the corresponding mean values for the total number of sensors and execution times for Algorithms 4.2 and 4.3 are generated as shown in Figures 5.35 and 5.36.

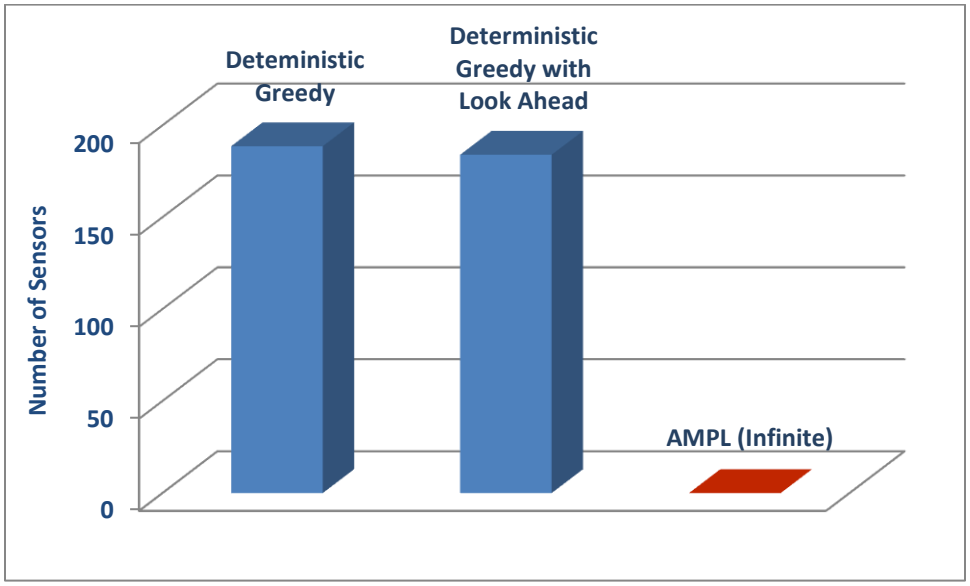

Figure 5.35. Graph comparing the total number of sensors for all three approaches (1000-node graph).

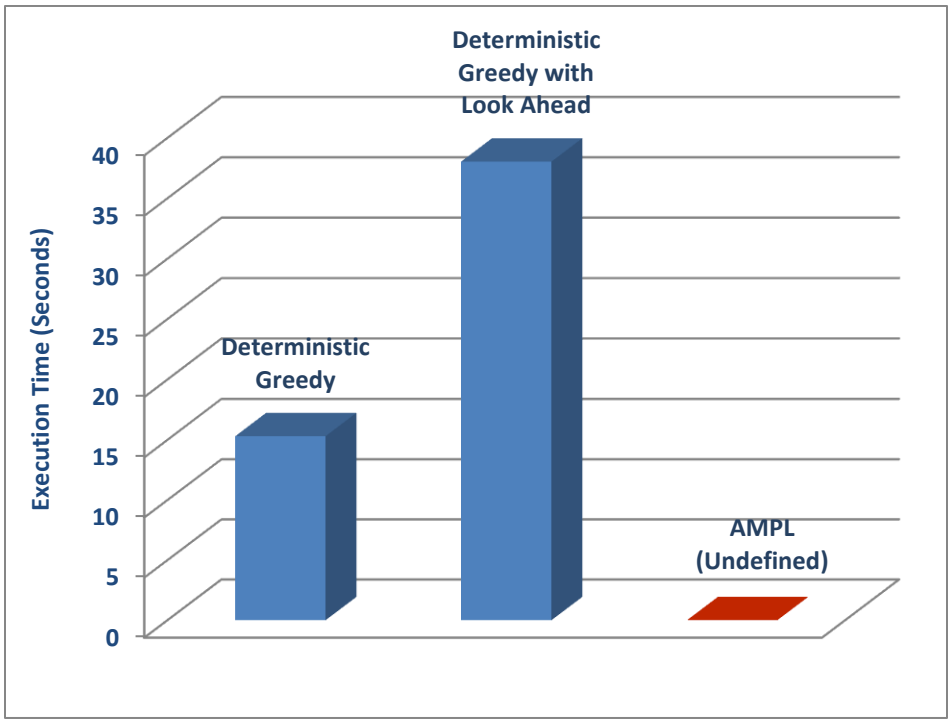

Figure 5.36. Graph comparing the execution time for all three approaches (1000-node graph).

5.2. Conclusion and Future Work

Deriving from the results of each test case discussed in Section 5.1, one can safely conclude that each algorithm has its own advantage over AMPL in terms of scalability. When the scalability limits of AMPL are pushed beyond 70 nodes, the AMPL application will fail to generate any kind of solution, whereas the scalability parameter for Algorithms 4.2 and 4.3 has been successfully tested up to 2000 node networks.

Comparing the results of Algorithms 4.2 and 4.3, it is evident that Algorithm 4.2 has a clear advantage with execution time over Algorithm 4.3, whereas Algorithm 4.3 has a huge advantage in terms of the total number of sensors. Another advantage of Algorithms 4.2 and 4.3 over AMPL is prioritization. One can successfully prioritize a network as discussed in Section 4.1.2. For any real-time network, there will always be a lot of constraints. There are some specific scenarios where a compromise between the optimal solution and the ideal solution should be maintained. The ideal solution can be obtained by placing sensors on every high-prioritized node which may then increase the sensor count drastically for some scenarios. AMPL will not let the user designate any prioritization for any given network, whereas when utilizing the graph-generation technique discussed in Section 4.1.2 for Algorithms 4.2 and 4.3, a compromise between the optimal solution and the ideal solution is achieved while still maintaining the concept of prioritization.

The Deterministic Greedy Algorithm discussed in Section 4.3 can be extended to use the Look-Ahead node to further examine the matrix before deciding on any node to place a sensor. This process may include a slight increase in efficiency (minimizing cost), but a significant amount of time delay is expected.

REFERENCES

[1] A. Abur, B. Xu.Observability Analysis and Measurement Placement for Systems with PMUs, IEEE Power Systems Conference and Exposition (2004).

[2] Reynaldo Francisco Nuqui. State Estimation and Voltage Security Monitoring Using Synchronized Phasor Measurements. Virginia Polytechnic Institute and State University, PhD Thesis (2001).

[3] J. R. Altman. A Practical Comprehensive Approach to PMU Placement for Full Observability, Virginia Polytechnic Institute and State University, Master's Thesis (2007).

[4] Andreas Krause, Carlos Guestrin, Anupam Gupta, and Jon Kleinberg. Near-Optimal Sensor Placements: Maximizing Information while Minimizing Communication Cost, CPSWeek (2006).

[5] Jinghe Zhang, Greg Welch, Gary Bishop, and Zhenyu Huang. Optimal PMU Placement Evaluation for Power System Dynamic State Estimation, Innovative Smart Grid Technologies Conference Europe (2010).

[6] T. Baldwin, L. Mili, J. Boisen, M.B., and R. Adapa. Power system observability with minimal phasor measurement placement. Power Systems, IEEE Transactions (1993).

[7] B. Xu and A. Abur. Observability analysis and measurement placement for systems with PMUs, Power Systems Conference and Exposition (2004).

[8] A. Phadke, J. Thorp, and M. Adamiak. A new measurement technique for tracking voltage phasors, local system frequency, and rate of change of frequency. Power Apparatus and Systems, IEEE Transactions (1983).

[9] A. G. Phadke, J. S. Thorp, and K. J. Karimi. State estimation with phasor measurements, Power Systems. Power Apparatus and Systems, IEEE Transactions (1986).

[10] C. Madtharad, S. Premrudeepreechacharn, and N. R. Watson. Power system state estimation using singular value decomposition. Electric Power Systems Research (2003).

[11] J. Chen and A. Abur. Placement of PMUs to enable bad data detection in state estimation. Power Systems, IEEE Transactions (2006).

[12] B. Gou. Generalized integer linear programming formulation for optimal PMU placement. Power Systems, IEEE Transactions (2008).