ASIAN JOURNAL OF CIVIL ENGINEERING (BUILDING AND HOUSING) VOL. 7, NO. 1 (2006)
PAGES 37-62

# AN EFFICIENT PROGRAM FOR CYCLE BASIS SELECTION AND BANDWIDTH OPTIMIZATION

A. Kaveh* and M. Jahanshahi
Department of Civil Engineering, Iran University of Science and Technology
Narmak, Tehran-16, Iran

## ABSTRACT

The applicability of graph theory for optimizing the sparsity and the bandwidth of cycle adjacency matrices of graphs is shown. Fundamental and subminimal cycle basis selection algorithms are presented in an algorithmic way. It is shown how the pattern of the cycle adjacency matrix changes during different phases of cycle selection and in particular when cycles are ordered. At each stage small pieces of code are presented to illustrate the simplicity of the implementation of the graph theoretical approaches using a computer language such as C++. The use of other languages should not cause much difficulty, although many aspects of an object oriented language such as C++ have been employed extensively throughout. This is intended to demonstrate the efficiency of graph theoretical methods combined with advanced techniques of computer programming

**Keywords:** sparsity, bandwidth optimization, fundamental cycle basis, subminimal cycle basis, ordering, C++

## 1. INTRODUCTION

Graph theory has extended its application in different fields of engineering and science. In civil engineering, graph theory has become an important tool in structural engineering, transportation, construction, and management. In structural analysis, graph theory is employed for the formation of sparse structural matrices, construction of well structured matrices such as those with narrow bandwidth, optimum profile and optimum frontwidth. Graphs are also used extensively in domain decomposition and parallel computing of large-scale problems. These and many others clearly show the extent of applications of graph theory in civil engineering.

Cycle basis of a graph has many applications in system analysis, and in particular structural analysis by the force method. Theoretically the application of the Greedy Algorithm leads to the formation of minimal cycle basis of a graph [1-3]. For generating a subminimal cycle basis of a graph, Kaveh's algorithm is the fastest known approach [4-7].

* E-mail address of the corresponding author: alikaveh@iust.ac.ir

Graph theoretical algorithms for the formation of suboptimal cycle bases are due to Kaveh [8-10].

In this paper, the applicability of graph theory for optimizing the sparsity and the bandwidth of cycle adjacency matrices of graphs is shown. Fundamental and subminimal cycle basis selection algorithms are illustrated in an algorithmic way. It is shown how the pattern of the cycle adjacency matrix changes during different phases of cycle selection and in particular when cycles are ordered. At each stage small pieces of code are presented to show the simplicity of the implementation of the graph theoretical approaches using a computer language such as C++. The use of other languages should not cause much problem, although many aspects of an object oriented language such as C++ have been employed extensively throughout. This is intended to show the efficiency of graph theoretical methods combined with advanced techniques of computer programming. The selected cycle bases can efficiently be employed in the force method of frame analysis as well as the mesh of analysis of hydraulic and electrical networks.

## 2. DATA STRUCTURES

The algorithms presented herein are developed in C++ using data structures that can incorporate standard coding styles such as Standard Template Library (STL) or any other convenient style as appropriate.

All the behaviors of a graph are encapsulated in an object called “TStructure” in which the initial “T” stands for the type. The name “Structure” is intended to have a generic meaning (both for a graph in general and a structure in particular).

Vertices and edges are stored as lists. The container devised for both is the vector data structure that can grow dynamically as new members are added. Cycles are represented as collections of edges belonging to them using the same container.

The following piece of code shows how these ideas take gel in a computer program:

```
class TStructure {
    .
    .
    .
    .
    private:
        vector<int>*   Nodes;
        vector<int>*   Elements;
};
```

In which “Nodes” and “Elements” are containers to store vertices and edges. As it is seen from the code they are part of the “TStructure” to imply that they belong to the graph that they represent.

Generating trees and shortest route trees are simple to implement, but they are easier

using data structures such as queues and stacks. These data structures are described below.

- Queue

Queue is just what the name implies. Items enter at one end and exit from the other. People lining up at a ticket window are a good example for queue. The first to get the ticket is the first to get out.

- Stack

A stack is a last in, first out data structure. The last item stored in a stack is the first to come out just like the stack of books set on top of each other. The first available book is the topmost one.

- Deque

The word deque is an abbreviation for double ended queue. In a deque objects are entered either from the front or back and extracted from either end.

## 3. FUNDAMENTAL CYCLE BASES

For finding a fundamental cycle basis for a graph under consideration, first a spanning tree of the graph is constructed. Such a tree comprising of N(S) – 1 edges is a maximal subgraph which is free from any cycle. Inclusion of any of M(S) – N(S) + 1 edges of the original graph that are not contained in the tree will produce a new cycle. M(S) and N(S) are the numbers of nodes and edges of a graph S, respectively. The cycles obtained in this way are independent because each non-tree edge is exactly contained in one cycle. These cycles form a basis known as a *fundamental cycle basis*.

The algorithm which forms the fundamental cycle basis stores the nodes in a queue as they are discovered and fetch them in that order. Edges encountered in generating the tree are divided into two groups. Edges that expand the tree to newly discovered nodes (tree edges) and those which are used for form cycles (co-tree edges).

In this algorithm, two arrays are defined which play important roles. These are “parent” and “explored”. “Parent” is an array of edge labels and identifies the parent edge of a node through which the node has been reached in generating the tree, and “explored” is an array of node labels and keeps track of the nodes previously explored. These arrays are ordinary arrays i.e. they do not use any specific data structure. With the help of these two arrays it is known when the addition of a new edge produces a new cycle or when it expands the tree.

Every time a co-tree edge is encountered, a new cycle is produced by backtracking the tree from the two end points to the first common node.

The following code fragment reveals the details of constructing fundamental cycle basis for a given graph.

```
// Find the node with maximum valency
for (unsigned i = 0; i < numNodes; i++) {
  unsigned valency = nodes[i].GetValency();

  if (valency > maxValency) {
    nodeId          = i;
```

```
    maxValency   = valency;
   }

   parent[i] = numElements;  // No element has an Id equal to numElements

   explored[i] = false;
  }

  queue->Put(nodeId);

  explored[nodeId] = true;

  while (!queue->IsEmpty()) {
   unsigned u = queue->Get();

   TEntityCollection& incidentElements = nodes[u].GetIncidentElements();

   unsigned n = incidentElements.GetItemsInContainer();

   // Order the elements incident to node u according to the valency of
   // opposite node
   for (unsigned i = 0; i < n - 1; i++) {
    unsigned node1  = OppositeNode(u, incidentElements[i], elements);
    unsigned val1   = nodes[node1].GetValency();

    for (unsigned j = i + 1; j < n; j++) {
     unsigned node2 = OppositeNode(u, incidentElements[j], elements);
     unsigned val2  = nodes[node2].GetValency();

     if (val2 > val1) {
      val1 = val2;

      Swap(incidentElements[i], incidentElements[j]);
     }
    }
   }

   for (unsigned i = 0; i < n; i++) {
    elementId = incidentElements[i];

    if (elementId == parent[u])
     continue;

    unsigned v = OppositeNode(u, elementId, elements);
```

```
if (explored[v]) {
 if (!nonTreeElements->HasMember(elementId)) {
  // Find a cycle on the co-tree element connecting nodes u and v
  nonTreeElements->Add(elementId);

  TCycle* cycle = new TCycle();

  // Add the co-tree element
  cycle->Add(elementId);

  // Follow the path from u to root
  nodeId = u;
  while ((elementId = parent[nodeId]) != numElements) {
   nodeId = OppositeNode(nodeId, elementId, elements);

   uNodes->Add(nodeId);

   uParents->PutRight(elementId);
  }

  unsigned rootNode = nodeId;

  // Follow the path from v to root
  nodeId = v;
  while (parent[nodeId] != numElements) {
   elementId = parent[nodeId];

   nodeId = OppositeNode(nodeId, elementId, elements);

   vParents->PutRight(elementId);
  }

  unsigned joinNode;

  nodeId = v;

  // Add elements of second path extending from node v to the common
  // node
  while (!vParents->IsEmpty()) {
   elementId = vParents->GetLeft();

   cycle->Add(elementId);

   nodeId = OppositeNode(nodeId, elementId, elements);
```

```
          if (uNodes->HasMember(nodeId)) {
            joinNode = nodeId;

            break;
          }
        }

        bool addEdge;
        if (joinNode == rootNode)
          addEdge = true;
        else
          addEdge = false;

        // Add elements of first path extending from node u to the common
        // node
        while (!uParents->IsEmpty()) {
          elementId = uParents->GetRight();

          if (addEdge == true)
            cycle->Add(elementId);
          else if (IsIncident(joinNode, elementId, elements))
            addEdge = true;
        }

        Cycles->Add(cycle);
        numCycles++;

        vParents->Flush();

        uNodes->Flush();
      }
    }
    else {
      queue->Put(v);

      explored[v] = true;

      parent[v] = elementId;
    }
  }
}
```

In order to find the cycle produced by the inclusion of the co-tree edge, two paths are

backtracked from the end points of the edge to the root and the nodes visited are stored in two deques, namely “uParents” and “vParents”. The reason for using a deque data structure is that it makes it easy to push the labels of edges at one end when backtracking and popping them from the other when generating the cycles. The next step is to determine the first node at which the two paths meet. The cycle can be formed as a set containing the co-tree and the two paths extending from the common node to the end nodes of the co-tree edge. Finally, it is important to note that the fundamental cycles are stored according to their length. This leads to a banded cycle adjacency matrix when replacing long cycles with shorter ones as the following section describes.

## 4. SUBMINIMAL CYCLE BASES

In generating fundamental cycles, each time only one co-tree edge was allowed to be added. Thus including long cycles was almost inevitable as the tree expanded. If we remove this constraint and allow the inclusion of more than one co-tree edge, we can improve the basis by replacing long cycles with shorter ones. The only problem which remains to be solved is to preserve the independency of cycles.

The exchange algorithm presented here, starts with long cycles obtained in previous section and tries to replace them with the shortest possible cycles. To do this, it takes the co-tree edge on which the original cycle has been formed and goes through finding the shortest path between the end nodes of this edge other than the edge itself. To ensure the independence, the co-tree edge is temporarily removed from the graph. This removal of course does not actually take place for the original graph. It is implied by removing the co-tree edge from the list of edges incident to its end nodes. In this sense, the removal process is temporary, but since the co-tree edge does not exist in the incidence list of nodes anymore, it clearly means that in each step one co-tree edge is removed from the graph. These steps of course necessitate the identification of the co-tree edges of fundamental cycles. To ease the process of identification, co-tree edges are always stored as the first edge of each cycle. The following code fragment is an implementation of these ideas.

```
// Loop over all cycles
for (unsigned i = 0; i < Cycles->GetItemsInContainer(); i++) {
  TCycle* oldCycle = (*Cycles)[i];

  element1 = (*oldCycle)[0];  // Original co-tree element

  unsigned x = elements[element1].GetiNode();
  unsigned y = elements[element1].GetjNode();

  // Remove co-tree element from original graph to preserve independence
  nodes[x].GetIncidentElements().Detach((unsigned long)element1);
  nodes[y].GetIncidentElements().Detach((unsigned long)element1);
```

```
// Find a shorter path from x to y
bool found = false;

unsigned maxLength = 3;  // A cycle with minimum length!

while (maxLength < oldCycle->Length()) {
 for (unsigned i = 0; i < numNodes; i++) {
  parent[i] = numElements;

  distance[i] = 0;
 }

 stack->Push(x);

 while (!stack->IsEmpty()) {
  unsigned u = stack->Pop();

  unsigned vDistance = distance[u] + 1;

  if (vDistance > maxLength - 1) {
   parent[u] = numElements;

   continue;
  }

  TEntityCollection& incidentElements = nodes[u].GetIncidentElements();

  unsigned n = incidentElements.GetItemsInContainer();

  for (unsigned j = 0; j < n; j++) {
   element2 = incidentElements[j];

   if (element2 == parent[u])
    continue;

   unsigned v = OppositeNode(u, element2, elements);

   unsigned& dist = distance[v];
   if (dist == 0 || dist > vDistance) {
    dist = vDistance;

    parent[v] = element2;

    if (v == y) {
```

```
          found = true;

          stack->Flush();

          goto form_cycle;
        }

        stack->Push(v);
      }
    }
  }

  maxLength++;
}

form_cycle:
if (found) {
  delete oldCycle;

  TCycle* newCycle = new TCycle;

  newCycle->Add(element1);  // Add co-tree element

  while ((element2 = parent[y]) != numElements) {
    newCycle->Add(element2);

    y = OppositeNode(y, element2, elements);
  }

  (*Cycles)[i] = newCycle;
 }
}
```

There are some implementation notes for this code that are worthwhile to be mentioned. First, as the distances are important here, a new array namely “distance” has been introduced. To prevent things from getting complicated, this new array also plays the role of “explored” array in the previous code. Second, as it does not matter how one selects the nodes when generating the tree from the end points of the co-tree edge, a stack has been used as the underlying data structure instead of a queue. Finally the algorithm tries to replace the old cycle with a cycle of length three and if it cannot do so it tries with a cycle of length four and this continues until no cycle shorter than the original one can be found. Of course this is not an obligation and one can choose another strategy for selecting a shorter cycle. The method adopted here seems to work well for moderately short cycles encountered in structural applications.

## 5. ORDERING CYCLES FOR BANDWIDTH OPTIMIZATION

Perhaps the final step and the most important one in optimizing the pattern of cycle adjacency matrix is ordering of cycles. Cycles are ordered according to two key properties. First their distances from the root and second their relationship with neighboring cycles (there are of course other important aspects that are not considered here). The first property is easily determined by generating a Shortest Route Tree (SRT) from the root and recording the distances of nodes (see [1] for a simple algorithm). The second is not as straight forward as the first and requires further elaboration because the connectivity of individual cycles is not so clear. In order to solve the connectivity problem, a special graph is associated with the minimal cycle basis obtained in previous section. This graph, namely the associate graph A(B(S) of a generalized cycle basis B(S), is a graph whose nodes are in a one to one correspondence with the elements of B(S), and two nodes are connected if two elements of B(S) have at least one member in common (reference [1]).

After constructing the associate graph, a nodal ordering is performed on its nodes starting from a node corresponding to the cycle containing the root. The nodal ordering algorithm adopted here, first generates an SRT from this node labeled the root for associate graph and obtains the last node discovered. The path through which this last node is reached in generating the SRT is selected as a special path (called transversal path [1]). Then, the nodes encountered through following the transversal path form the root to the last node, are recorded as representative nodes. The final step is to generate an SRT from each representative node and label the nodes in order of their occurrence.

By constructing an associate graph and ordering the nodes as described above, the connectivity of cycles is taken into account in some sense. Now that the distance numbers of cycles are determined and they are ordered according to their connectivity, it is an easy task to reorder them for pattern optimization. Cycles with small distance numbers have priority over the others and when the distance numbers of cycles are equal, their labels obtained in nodal ordering governs the selection of cycles, i.e. the cycles with smaller labels have the highest priority. The following code fragment is an implementation of these ideas:

```
// Reset 'parent' and 'distance' arrays
for (unsigned I = 0; I < numNodes; I++) {
  parent[I] = numElements;

  distance[I] = 0;
}

// Step1: Find cycle distances from the root
queue->Put(nodeId);

while (!queue->IsEmpty()) {
  unsigned u = queue->Get();

  unsigned vDistance = distance[u];
```

```
 for (unsigned I = 0; I < numCycles; I++) {
  Tcycle& cycle = *(*Cycles)[I];

  if (cycle.HasNode(u, elements))
   cycle.SetDistance(vDistance);
 }

 vDistance++;

 TentityCollection& incidentElements = nodes[u].GetIncidentElements();

 unsigned n = incidentElements.GetItemsInContainer();

 for (unsigned I = 0; I < n; I++) {
  elementId = incidentElements[I];

  if (elementId == parent[u])
   continue;

  unsigned v = OppositeNode(u, elementId, elements);

  if (distance[v] == 0) {
   queue->Put(v);

   parent[v] = elementId;

   distance[v] = vDistance;
  }
 }
}

delete[] distance;

delete[] parent;

// Step2: Construct the associate graph
Tcycles& cycles = *Cycles;

Tstructure* str = new Tstructure;

// Add each node for each cycle and also find the first cycle which contains
// the root
unsigned cycleId = numCycles;
```

```
for (unsigned I = 0; I < numCycles; I++) {
  Tnode node;  // The coordinates are immaterial

  str->AddNode(node);

  if (cycleId == numCycles && cycles[I]->HasNode(nodeId, elements))
    cycleId = I;
}

// Connect the nodes whose underlying cycles have at least one element in common
for (unsigned I = 0; I < numCycles – 1; I++)
  for (unsigned j = I + 1; j < numCycles; j++)
    if (I != j && HaveCommonElement(*cycles[I], *cycles[j]))
      str->AddElement(I, j);

str->SetIncidency();

// Step3: Generate an SRT from the node associated with the cycle containing
// the root and record the distances
Tnodes&    aNodes      = str->GetNodes   ();      // Associate nodes
Telements& aElements= str->GetElements();     // Associate elements

unsigned n1 = str->GetNumNodes ();
unsigned n2 = str->GetNumElements();

parent = new unsigned[n1];

distance = new unsigned[n1];

for (unsigned I = 0; I < n1; I++) {
  parent[I] = n2;

  distance[I] = 0;
}

queue->Put(cycleId);

unsigned u;
while (!queue->IsEmpty()) {
  u = queue->Get();

  unsigned vDistance = distance[u] + 1;

  TentityCollection& incidentElements = aNodes[u].GetIncidentElements();
```

```
 unsigned n = incidentElements.GetItemsInContainer();

 for (unsigned I = 0; I < n; I++) {
  elementId = incidentElements[I];

  if (elementId == parent[u])
   continue;

  unsigned v = OppositeNode(u, elementId, aElements);

  if (distance[v] == 0) {
   queue->Put(v);

   parent[v] = elementId;

   distance[v] = vDistance;
  }
 }
}

// Step4: Find the transversal nodes
unsigned n = distance[u];

unsigned* tNodes = new unsigned[n];

for (unsigned I = 1; I <= n; I++) {
 tNodes[n – I] = u;

 elementId = parent[u];

 u = OppositeNode(u, elementId, aElements);
}

// Step5: Order the nodes of each contour starting with transversals as their
// representative
bool* explored = new bool[n1];

unsigned priority = 0;

cycles[cycleId]->SetPriority(priority++);  // Topmost priority

// Loop over transversal nodes
for (unsigned I = 0; I < n; I++) {
 unsigned Id = tNodes[I];
```

```
  unsigned dist = distance[Id];

  // Generate an SRT from the transversal node in order to find the nodes of
  // corresponding contour using the distances obtained in previous step
  for (unsigned I = 0; I < n1; I++) {
    parent[I] = n2;

    explored[I] = false;
  }

  queue->Put(Id);

  while (!queue->IsEmpty()) {
    u = queue->Get();

    if (distance[u] == dist)
      cycles[u]->SetPriority(priority++);

    TentityCollection& incidentElements = aNodes[u].GetIncidentElements();

    unsigned n = incidentElements.GetItemsInContainer();

    for (unsigned I = 0; I < n; I++) {
      elementId = incidentElements[I];

      if (elementId == parent[u])
        continue;

      unsigned v = OppositeNode(u, elementId, aElements);

      if (!explored[v]) {
        queue->Put(v);

        parent[v] = elementId;

        explored[v] = true;
      }
    }
  }
}

// Ordering the cycles
for (unsigned I = 0; I < numCycles – 1; I++) {
  Tcycle* cycle1 = cycles[I];
```

```
double distance1 = cycle1->Distance();

for (unsigned j = I + 1; j < numCycles; j++) {
  Tcycle* cycle2 = cycles[j];

  double distance2 = cycle2->Distance();

  if (distance2 > distance1)
    continue;
  else if (distance2 < distance1 || cycle2->Priority() < cycle1->Priority()) {
    cycles[I] = cycle2;
    cycles[j] = cycle1;

    cycle1 = cycle2;

    distance1 = distance2;
  }
 }
}
```

The "TStructure" is an object which is designed to resemble the property of a graph. Therefore, when constructing the associate graph, it is not necessary to repeat all sort of operations that are needed to initialize a new graph. Just make an instance of "TStructure" and the associate graph will be ready for use.

In the above code, it should be noted that to improve the performance, pointers to cycles are replaced not the cycles themselves. This of course eliminates the necessity of replacing the individual members of cycles which requires a great deal of time.

## 6. EXAMPLES

After all these details, it would be illuminating to present some examples showing how these algorithms are put together in a working program. A well designed program with a user friendly interface can help to create and manipulate different models quickly and obtain results in fractions of a second which would require great deal of time to implement by hand.

In what follows a few examples with a different level of complexity are presented to clarify the ideas developed in previous sections and to show how the algorithms work.

**Example 1:** In Figure 1 a typical graph is shown. This graph has special properties which can confuse algorithms developed for cycle selection easily. Fundamental cycle basis found by the program are shown in Figure 2 (a) through (h).

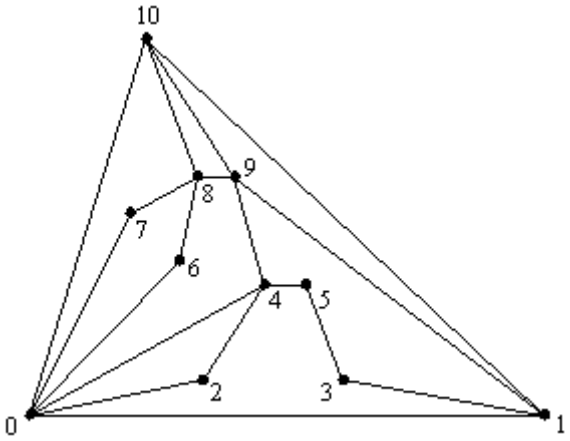
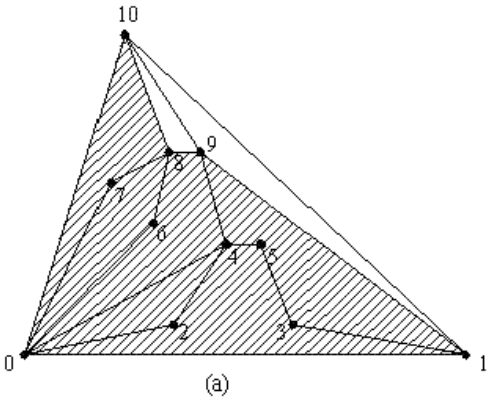
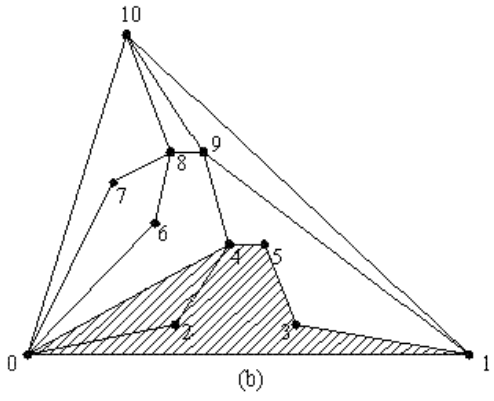


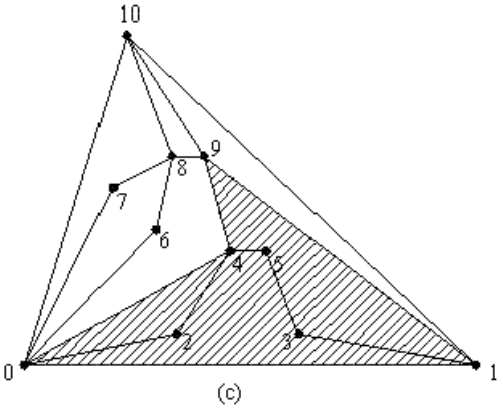

Figure 1. The graph of Example 1



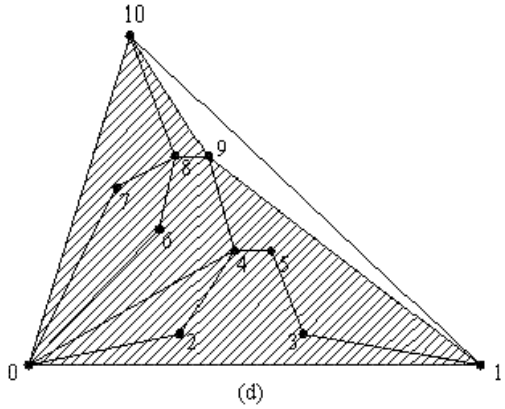

(a)




(b)




(c)




(d)

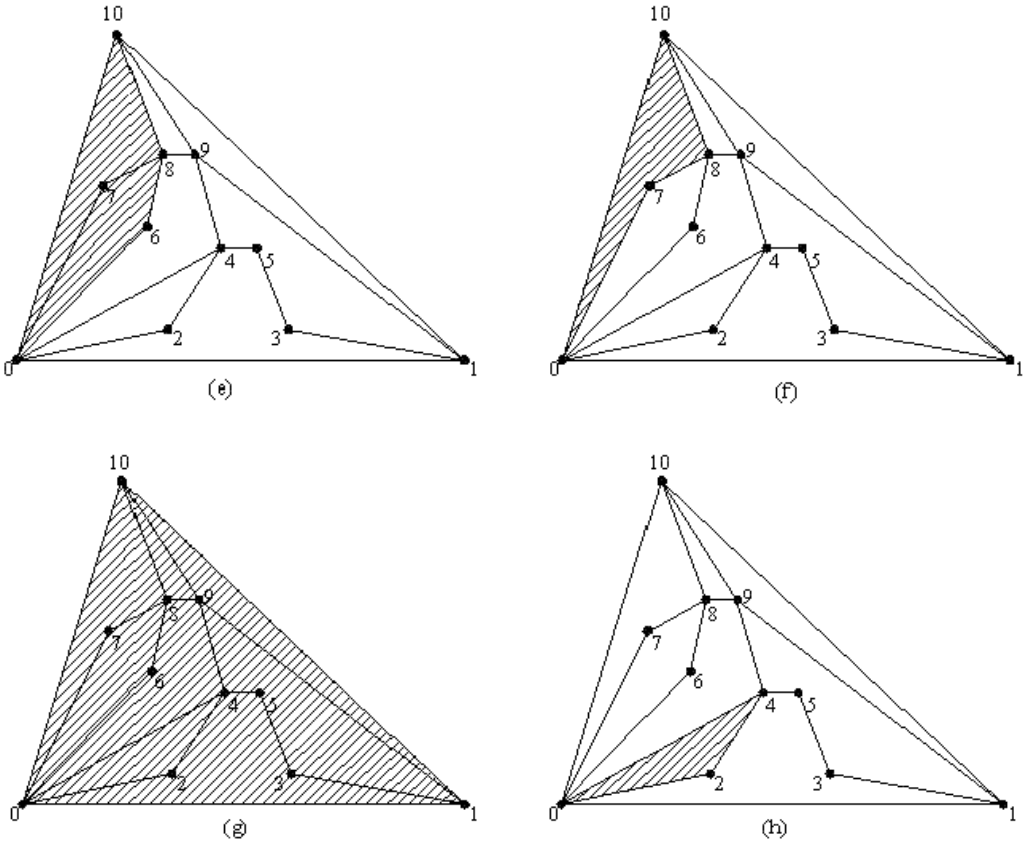
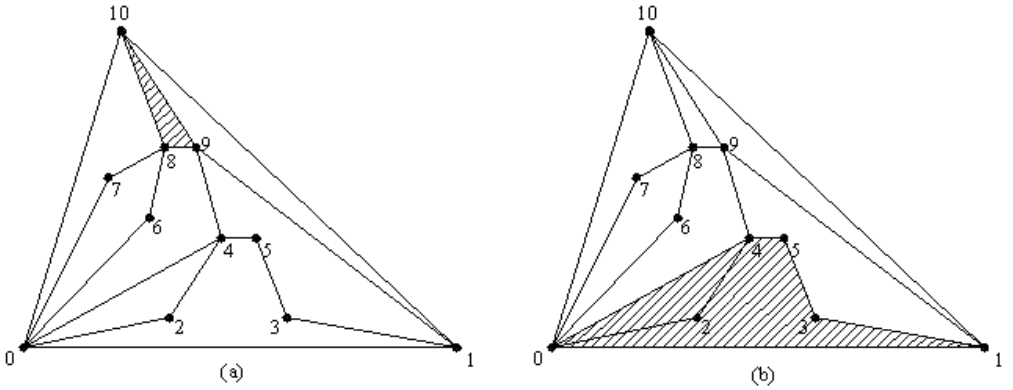



Figure 2. The selected fundamental cycle basis

The subminimal cycle selection algorithm replaces the fundamental cycles (a) through (h) in Figure 2 with minimal cycles (a) through (h) in Figure 3.



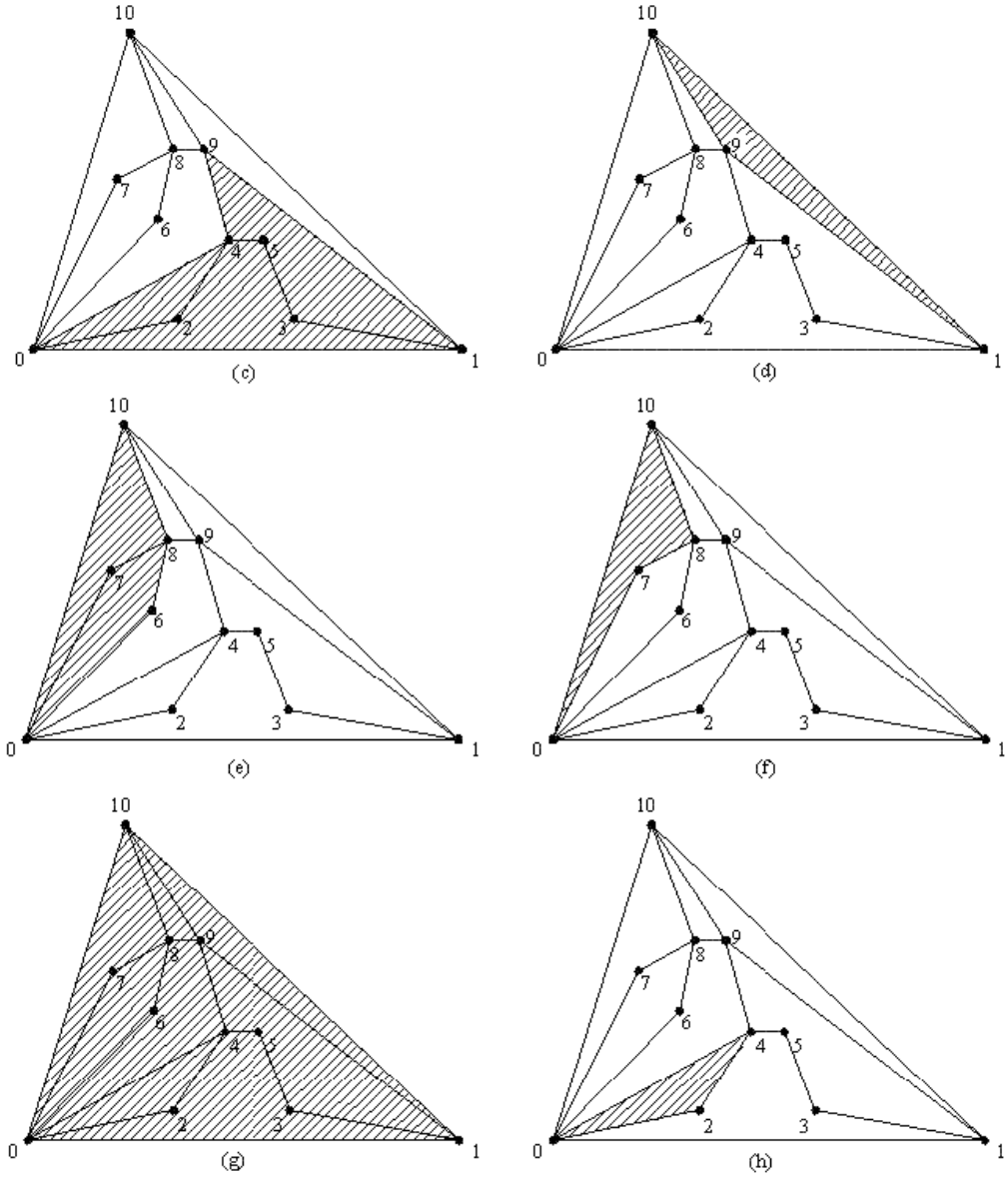



Figure 3. The selected cycle basis by the present algorithm

The pattern of cycle adjacency matrix changes from that of Figure 4 to that shown in Figure 5.

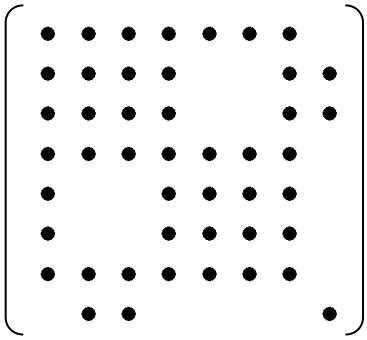


Figure 4. Pattern of the fundamental cycle basis

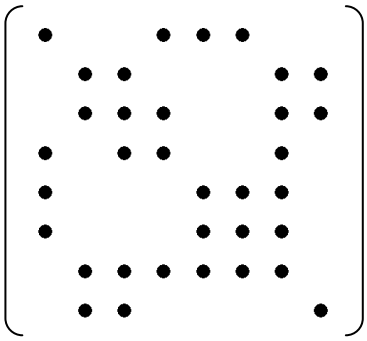


Figure 5. Pattern of the subminimal cycle basis

The number of nonzero elements in cycle adjacency matrix is 46 for fundamental cycle basis which reduces to 34 for minimal cycle basis. After ordering the cycles, the following pattern is obtained (Figure 6):

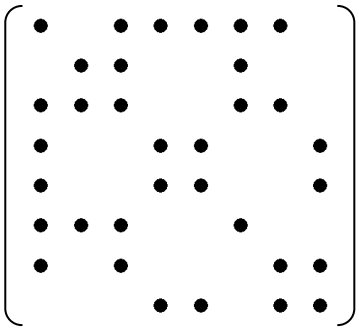


Figure 6. Pattern of the ordered cycle basis

**Example 2**: This example is basically similar to the previous one, Figure 7. It is intended to provide a means to enable to check the results obtained by the program. As before, first the fundamental cycles are selected, and replaced by minimal cycles shown in Figs. 8(a) through (i).

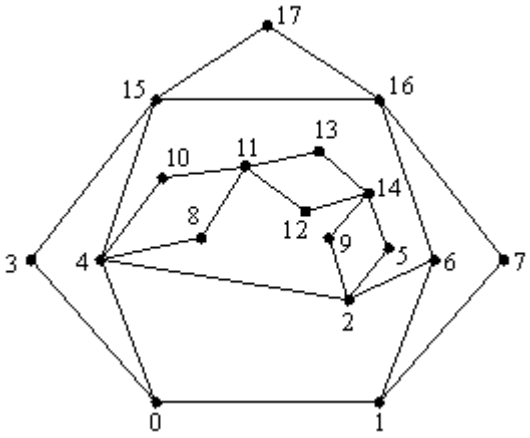
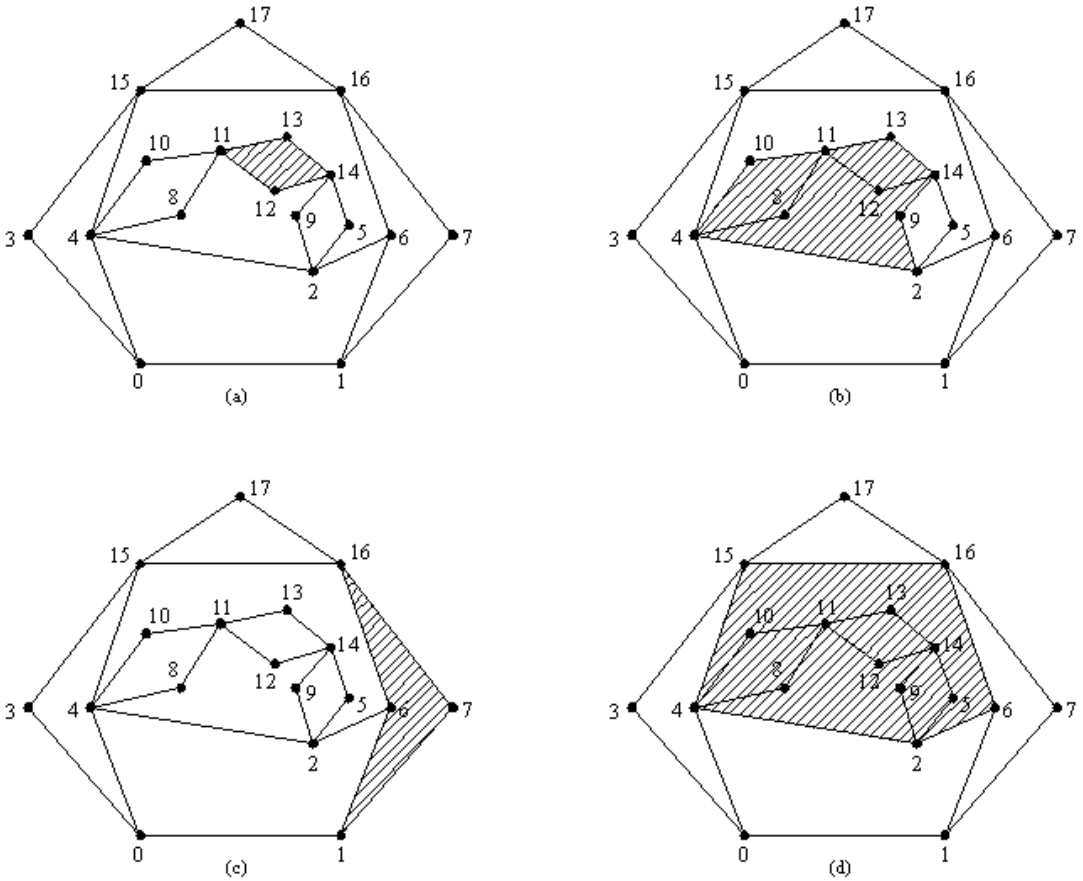



Figure 7. The graph of Example 2



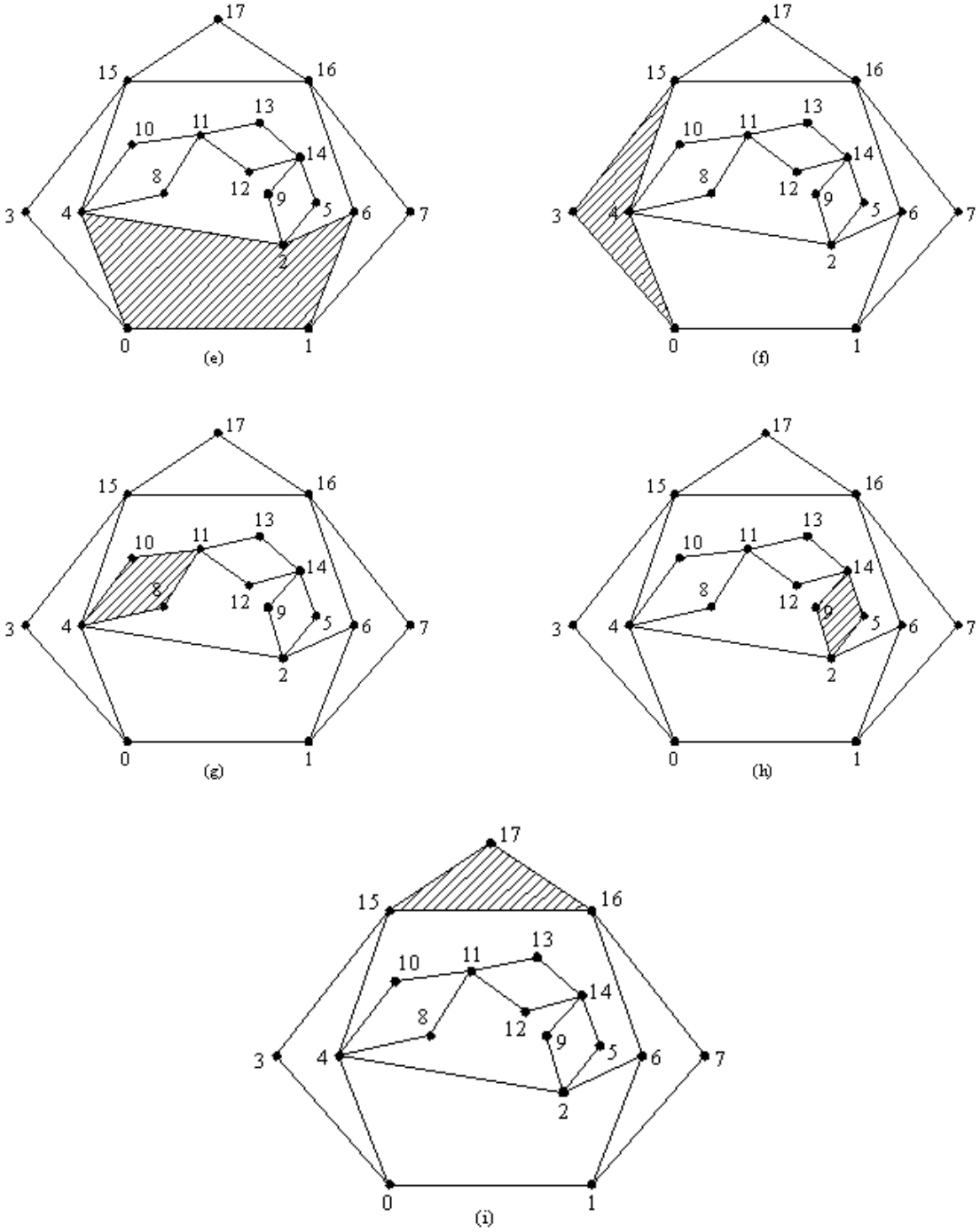



Figure 8. The selected cycle basis by the present algorithm

The pattern of cycle adjacency matrices are shown for two cycle bases in Figs. 9 and 10.

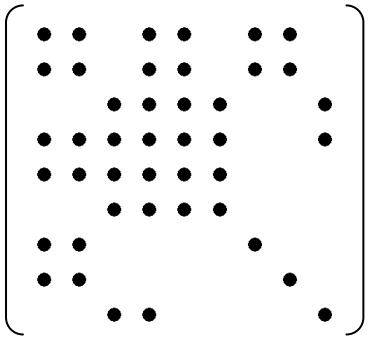


Figure 9. Pattern of the fundamental cycle basis

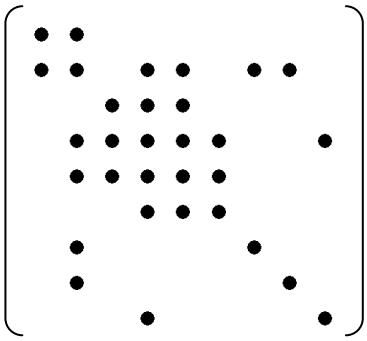


Figure 10. Pattern of the subminimal cycle basis

The number of nonzero elements reduces from 43 to 31. After ordering the cycles, the pattern takes the following form (Figure 11):

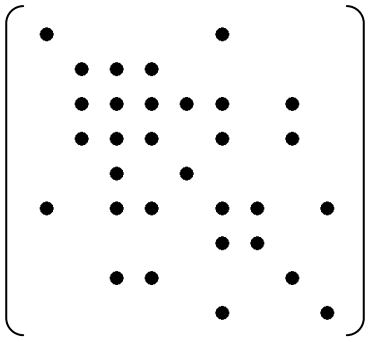


Figure 11. Pattern of the ordered cycle basis

**Example 3**: In previous examples the effect of ordering was not so clear as the dimension of the problems considered was in fact small. Now we consider a problem whose dimension does not permit to do the calculations easily by hand. This is a good example of the applicability of a program developed specially for the purpose of handling large-scale problems.

In the following a graph consisting of 4 bays in x and y and 10 stories in z direction is considered, Figure 12. It is comprised of 275 vertices and 690 edges. It is obvious that finding the fundamental cycles or minimal ones and then calculating the cycle adjacency matrix could be a formidable task. However, by using a suitable tool the time and labor devoted to this task can be saved considerably.

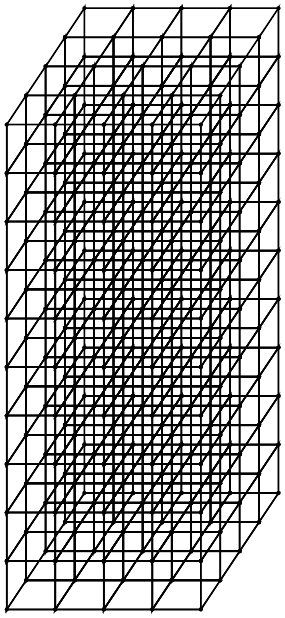


Figure 12. A ten storey space frame

After constructing the fundamental cycle basis, the pattern of cycle adjacency matrix takes a scattered form shown in Figure 13. Replacing the cycles with a subminimal cycle basis, the pattern improves to some extent as shown in Figure 14. Finally by ordering the subminimal cycle basis, an optimal pattern is obtained. This final pattern is shown in Figure 15. All these operations take no longer than a few second to accomplish.

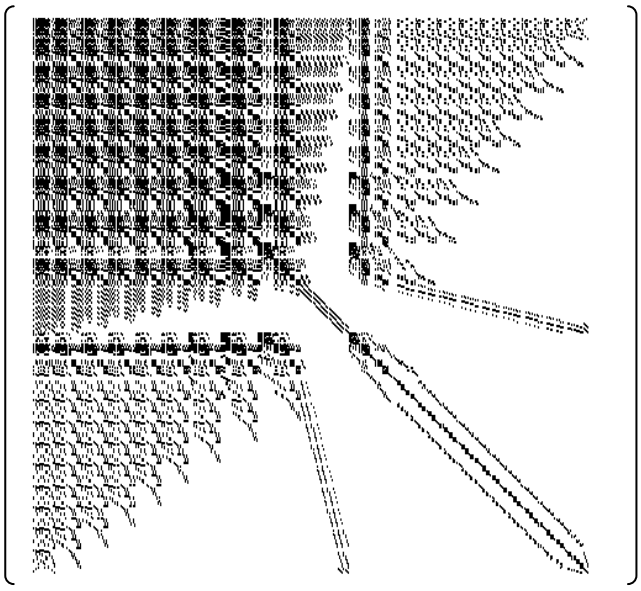


Figure 13. Pattern of the fundamental cycle basis adjacency matrix

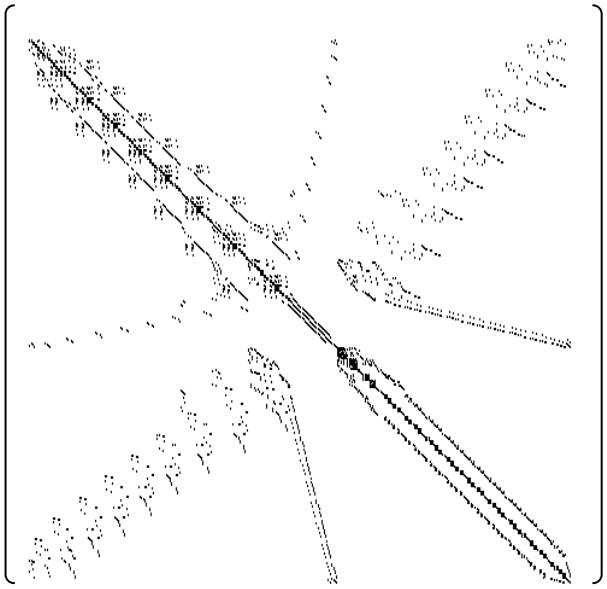


Figure 14. Pattern of the subminimal cycle basis adjacency matrix

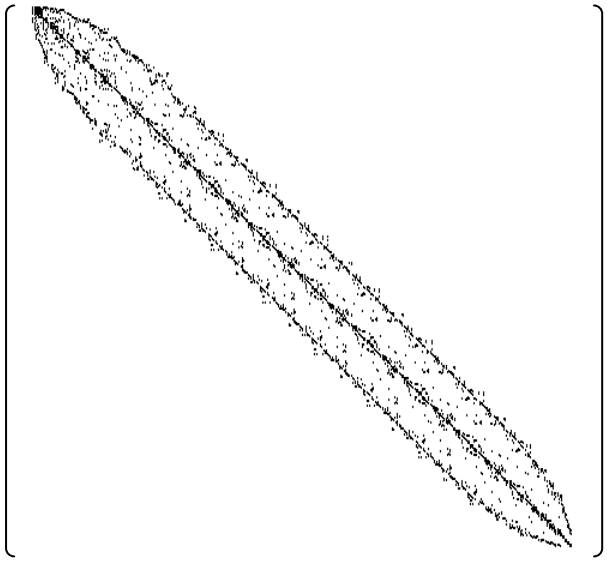


Figure 15. Pattern of the fundamental cycle basis adjacency matrix after bandwidth optimization

## 6. CONCLUDING REMARKS

In order to present some time measures for the algorithms, in Figure 16 a unit block consisting of 8 vertices and 12 edges is studied. This is the building block for the graphs used as our measure in computing the time elapses. In other words, sample graphs are composed of equal number of units in x, y and z directions. For each graph, number of unit

blocks, vertices, and total number of edges are shown in Table 1. Also shown, are the time elapses for computing fundamental and minimal cycles for each case.

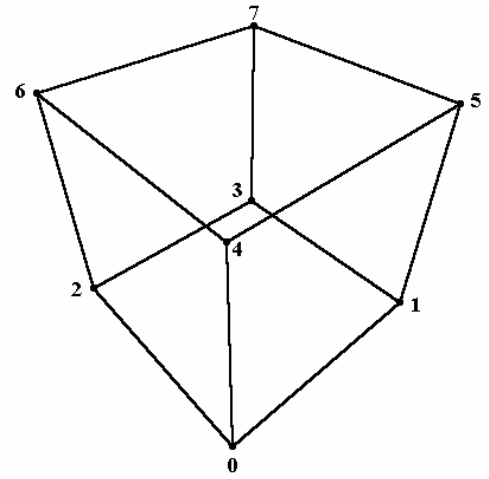



Figure 16. The structural unit used in expansion

Table 1. Elapsed time for sample graphs

| Graphs | Blocks | N(S) | M(S) | Number of Cycles | Elapsed Time (sec.) | |
|---|---|---|---|---|---|---|
| | | | | | Fundamental Cycles | Subminimal Cycles |
| 1 | 5 | 216 | 540 | 325 | 0.054 | 0.025 |
| 2 | 8 | 729 | 1944 | 1216 | 0.755 | 0.299 |
| 3 | 10 | 1331 | 3630 | 2300 | 2.77 | 1.335 |
| 4 | 13 | 2744 | 7644 | 4901 | 15.159 | 7.635 |
| 5 | 15 | 4096 | 11,520 | 7425 | 42.17 | 10.604 |
| 6 | 18 | 6859 | 19,494 | 12,636 | 174.59 | 39.72 |

Figure 17 shows the variation of elapsed time with respect to the number of cycles. It is seen that as the number of cycles is increased, the rate of change of time for fundamental cycles is more pronounced compared to minimal cycles. This clearly demonstrates the amount of work necessary for finding the fundamental cycles for a given graph. However, the two steps of finding the fundamental and minimal cycles can be combined together to reduce the computational effort. However, this has the drawback that the fundamental cycles no more exist to provide a measure for minimal cycle selection algorithm in order to differentiate between good cycles and bad ones. In other words, subminimal cycle selection algorithm

would search blindly in the large space of cycles with the hope to find better cycles.

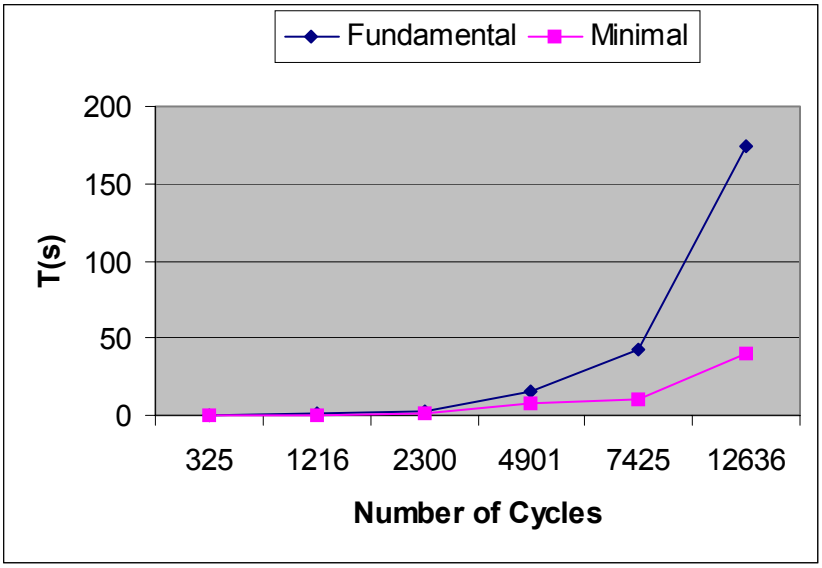



Figure 17. The variation of elapsed time with respect to the number of cycles

## REFERENCES

1. Kaveh, A., *Structural Mechanics: Graph and Matrix Methods*. RSP (John Wiley), Exeter, UK, 3rd edition, 2004.
2. Horton, J.D., A polynomial time algorithm to find the shortest cycle basis of a graph. *SIAM Journal of Computing*, **16**(1987) 358-366.
3. Kaveh, A. and Roosta, G.R., Revised Greedy Algorithm for the formation of a minimal cycle basis of a graph. *Communications in Applied Numerical Methods*, **10**(1994) 523-530.
4. Kaveh, A., Improved cycle bases for the flexibility analysis of structures. *Computer Methods in Applied Mechanics and Engineering,* **9**(1976)267-272.
5. Kaveh, A. Multiple use of a shortest route tree for ordering, *Communications in Numerical Methods in Engineering*, **2**(1986)213-215.
6. Kaveh, A. An efficient program for generating subminimal cycle bases for the flexibility analysis of structures, *Communications in Numerical Methods in Engineering*, **2**(1986)339-344.
7. Kaveh, A. Subminimal cycle bases for the force method of structural analysis, *Communications in Numerical Methods in Engineering*, **3**(1987) 277-280.
8. Kaveh, A., *Optimal Structural Analysis*, Wiley (RSP), 2nd edition, UK, 2006.
9. Kaveh, A., Suboptimal cycle bases of graphs for the flexibility analysis of skeletal structures, *Computer Methods in Applied Mechanics and Engineering*, **71**(1988)259-271.
10. Kaveh, A. Suboptimal cycle bases of a graph for mesh analysis of networks, *International Journal of NETWORKS*, **19**(1989)273-279.