Enumerating All Cycles in an Undirected Graph

Philipp Sch

9 Sep 2018 CPOL

Finding a fundamental Cycle Set forming a complete basis to enumerate all cycles of a given undirected graph

Download source - 9.7 KB

1. Introduction

Graphs can be used in many different applications from electronic engineering describing electrical circuits to theoretical chemistry describing molecular networks. It can be necessary to enumerate cycles in the graph or to find certain cycles in the graph which meet certain criteria. In this article, I will explain how to in principle enumerate all cycles of a graph but we will see that this number easily grows in size such that it is not possible to loop through all cycles. However, the ability to enumerate all possible cycles allows one to use heuristical methods like Monte Carlo or Evolutionary Algorithms to answer specific questions regarding cycles in graphs (e.g., finding the smallest or largest cycle, or cycles of a specific length) without actually visiting all cycles. Here, I will address undirected unweighted graphs (see Figure 1a for an example) but the algorithm is straightforwardly transferable to weighted graphs.

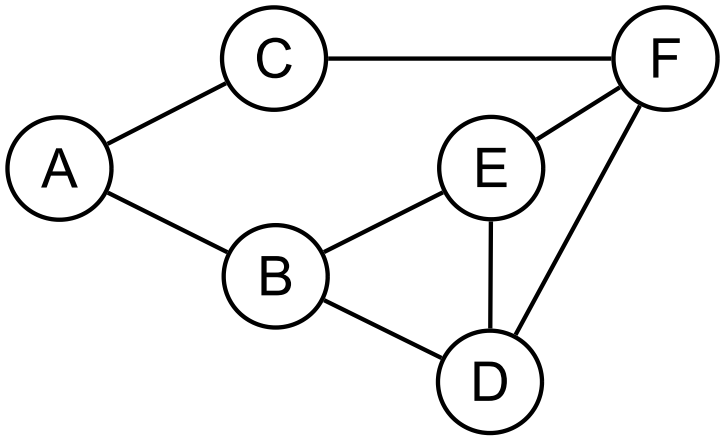
2. Fundamentals

In this section, all tools which are absolutely necessary to understand the following sections will be explained.

a) Representation of a Graph

The first topic is the representation of a given graph (e.g., as shown in Fig. 1a) in the program code. A common and practical approach is the *adjacency matrix (A)*. It consists of NxN elements, where N is the number of nodes in the graph. Each Element A_{ij} equals 1 if the two nodes i and j are connected and zero otherwise. The adjacency matrix for the Graph shown in Fig. 1a is shown in Fig. 1b. As we are dealing with undirected graphs, the adjacency matrix is *symmetrical*, i.e., just the lower or upper half is needed to describe the graph completely because if node A is connected to node B, it automatically follows that B is connected to A. Additionally also, the diagonal elements are neglected which were only needed to indicate that one node is connected with itself. Consequently, this would automatically be a fundamental node of the whole graph because it cannot be divided further.

The code provides a class `HalfAdjacencyMatrix` used to represent a graph. As described, it just stores one half of the matrix and additionally neglects the diagonal elements.
The class can also be used to store a cycle, path or any kind of substructure in the graph. In that case, there might be nodes which do not belong to the substructure and therefore have no edges. The adjacency matrix might also contain two or more disjoint substructures (see below).



(a)

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(b)

Fig. 1: An undirected graph (a) and its adjacency matrix (b).

b) Combining Two Paths / Cycles

To determine a set of fundamental cycles and later enumerate all possible cycles of the graph, it is necessary that two adjacency matrices (which might contain paths, cycles, graphs, etc.), can be merged.

This will be done in the following by applying the logical XOR operator on each edge of the two adjacency matrices. In the following two examples are presented how the XOR-operator can be used to yield merged paths and cycles.

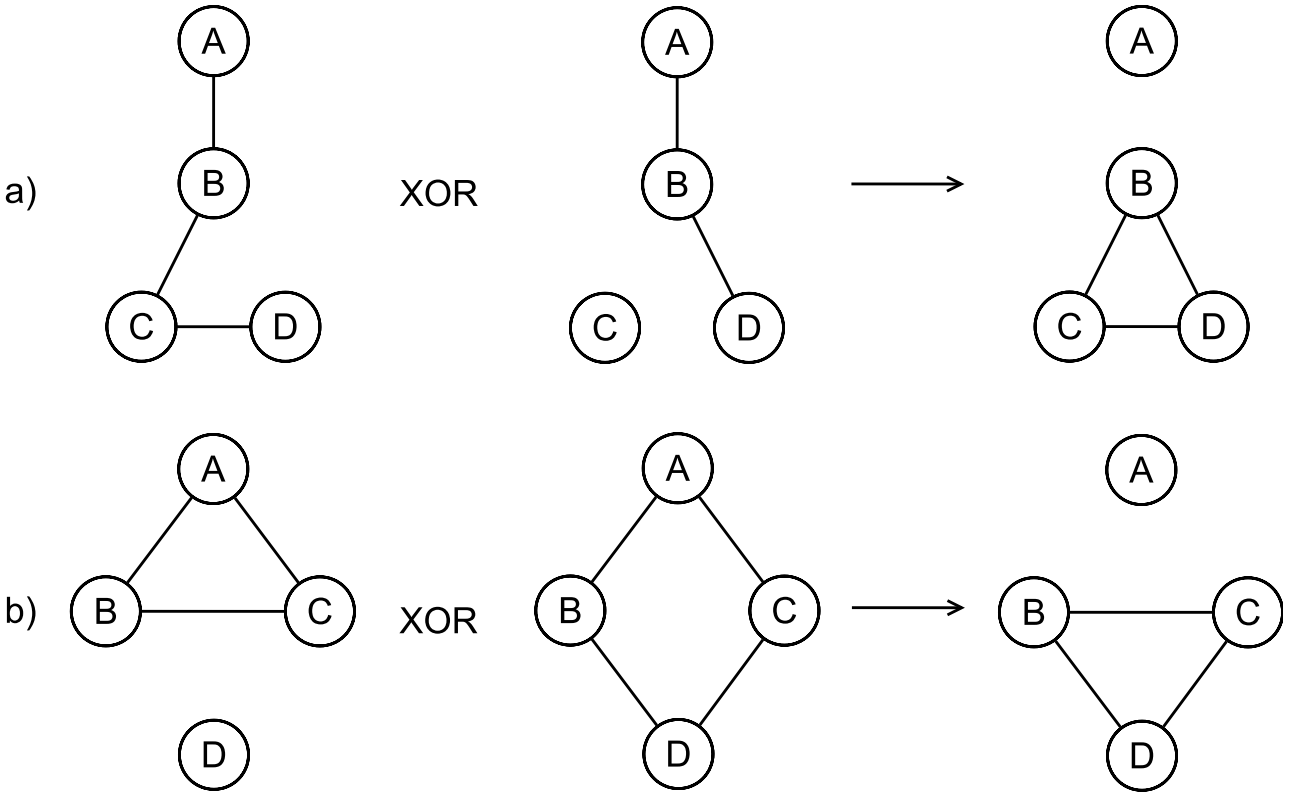


Fig. 2: Illustration of the XOR operator applied to two distinct paths (a) and to two distinct cycles (b) within an arbitrary graph.

In Fig. 2a, the XOR operator is applied to two paths both emerging from the root element in the given graph. The result is a closed cycle B-C-D-B where the root element A was excluded. This scheme will be used to yield a fundamental cycle from two paths of a graphs spanning tree as described in Sec. 3.

Two cycles are combined in Fig. 2b yielding a new cycle. This scheme will be used in Sec. 4 to form new cycles from the cycle base of the graph.

The implementation of the XOR-operator (`operator^`) is straightforward. The function loops over each bit present in the two matrices and applies XOR to each bit (edge), individually. The class additionally provides `operator^=` for convenience.

```
// performs a xor operation on the two matrices and returns a new one.
inline HalfAdjacencyMatrix operator^(const HalfAdjacencyMatrix& rhs) const
{
    if(m_nNodes != rhs.m_nNodes)
        throw std::runtime_error("HalfAdjacencyMatrix::operator^():
                                  The two matrices MUST be of the same size!");
```

```
    // constructor initializes A_{ij} = 0
    HalfAdjacencyMatrix result(m_nNodes);
    for(size_t i = 0; i < m_aBits.size(); ++i)
    {
        // XOR for each bit: If the bit is true for any of the two matrices
        // AND the bits in both matrices are not equal
        // the bit is again true in the result matrix.
        if((m_aBits[i] || rhs.m_aBits[i]) && (m_aBits[i] != rhs.m_aBits[i]))
        {
            result.m_aBits[i] = 1;
            ++result.m_nEdges;
        }
    }
    return result;
}
```

3. Finding a Fundamental Cycle Set

a) Spanning Trees

The algorithm described here follows the algorithm published by Paton [1]. The central idea is to generate a *spanning tree* from the undirected graph. After the spanning tree is built, we have to look for all edges which are present in the graph but not in the tree. Adding one of the missing edges to the tree will form a cycle which is called *fundamental cycle*. Note that a graph can have many different spanning trees depending on the chosen root node and the way the tree was built. Consequently, each spanning tree constructs its own fundamental cycle set. All fundamental cycles form a cycle basis, i.e., a basis for the *cycle space* of the graph. As the basis is complete, it does not matter which spanning tree was used to generate the cycle basis, each basis is equally suitable to construct all possible cycles of the graph. Two possible spanning trees of the exemplary graph shown in Fig. 1a are shown in Fig. 3 which were built using the depth-first (a) and the breadth-first search (b), respectively.
Note that Paton prefers depth-first search over breadth-first search because using depth-first search each node just differs by one edge from the main branch. This can be utilized to construct the fundamental cycles more efficiently. For simplicity, I use the XOR operator to combine two paths of the spanning tree and thus both, depth-first and breadth-first search are equally efficient.

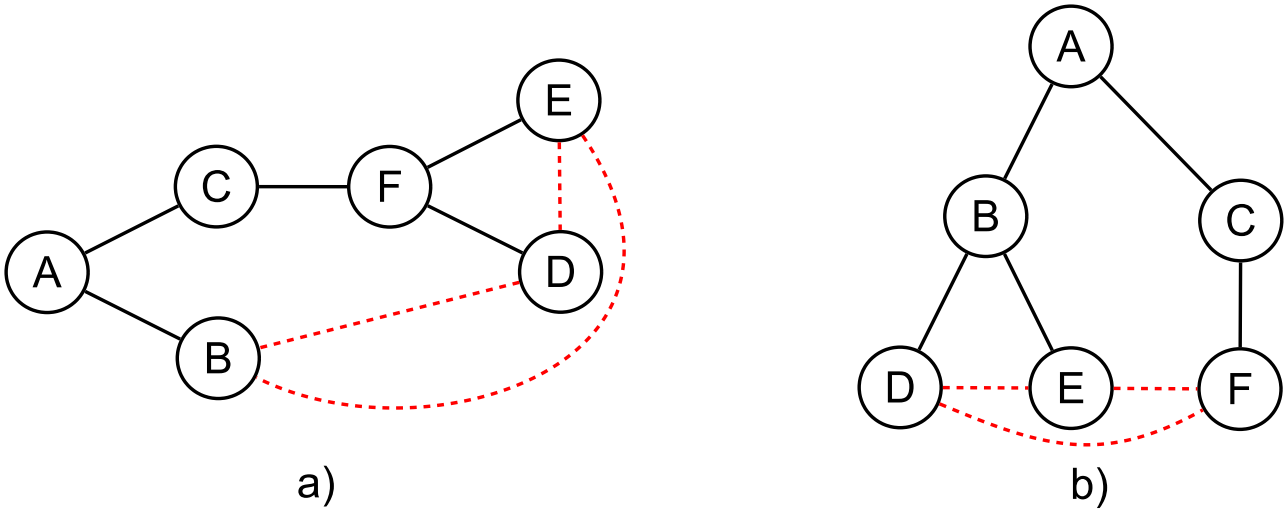


Fig. 3: Generation of a minimal spanning tree of the undirected graph in Fig. 1a. Depth-first search (a) is illustrated vs. breadth-first search (b). All edges which are missing in the tree but present in the graph are shown as red dashed lines.

b) Size of the Cycle Base

As stated in the previous section, the fundamental cycles in the cycle base will vary depending on the chosen spanning tree. However, the number of fundamental cycles is always the same and can be easily calculated:
For any given undirected graph having V nodes and E edges, the number of fundamental cycles N_{FC} is:

$$N_{\mathrm{FC}} = E - V + 1$$

assuming that the graph is fully connected in the beginning [2]. This number is also called "cycle rank" or "circuit rank" [3].

c) Pseudo-Code

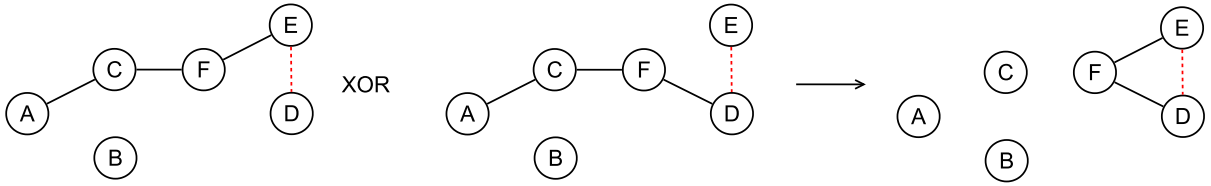
```
V: All Vertices
E: All Edges
pick start Vertex v in V
T = {v}  // Spanning tree
Q = {v}  // Process Queue

while not Q.empty:
    i := Q.top
    for j < |V|
        if (i,j) in E
            if j in T
```

```
            // Fundamental Cycle Found!
            pi := T.pathFromRootToElement(i)
            pj := T.pathFromRootToElement(j)
            cycle := merge pi and pj
        else
            Q.push(j)
            insert j to T and set i as parent of j
```

The above psudo code finds a set of fundamental cycles for the given graph described by V and E.
There are a few things to address here:

- How to pick the start vertex `v`?
  As discussed earlier, the spanning tree will depend on the chosen start vertex `v` and therefore also the fundamental cycle set. However as the cycle bases are all equivalent, it does not matter which vertex is chosen, thus in this implementation, just the first node in the graph is picked.
- How are the two paths, `pi` and `pj` merged?
  The simplest method to merge the two paths is the XOR operator:



  Here, one directly gets rid of all the edges which are present in both paths, i.e., A-C and C-F in the given example. Note that the edge E-D (red dotted line) is not present in the tree and must be added to yield the fundamental cycle.

d) Implementation

The implementation follows a standard depth-search algorithm. As soon as a node is found which was already visited, a cycle of the graph was found. By combining the paths to the current node and the found node with the XOR operator, the cycle represented by an adjacency matrix is obtained and stored in the class for later usage.

```
void Graph::computeFundamentalCycles()
{
    // Lazy evaluation; save the fundamental cycles in the Graph class and
    // just compute it ONCE when its needed
    if(!m_aFundamentalCycles.empty())
        return;

    std::unique_ptr<TreeNode[]> aTree(new TreeNode[m_aNodes.size()]);
    std::stack<size_t> nodeStack;

    // Start arbitrarily with the first Node!
    nodeStack.push(0);

    // Copy the adjacency matrix as it will be necessary to remove edges!
```

```
    HalfAdjacencyMatrix adjMat = m_adjMat;

    // At the beginning, all tree nodes point to itself as parent!
    // The tree is built on the fly
    for(size_t i = 0; i < m_aNodes.size(); ++i)
    {
        aTree[i].parent = &aTree[i];
        aTree[i].index = i;
    }

    // Loop until all nodes are removed from the stack!
    while(nodeStack.size() > 0)
    {
        // Next node index:
        size_t currentNodeIndex = nodeStack.top();
        nodeStack.pop();
        TreeNode& currentTreeNode = aTree[currentNodeIndex];

        // Iterate though all edges connecting this node:
        for(size_t j = 0; j < m_aNodes.size(); ++j)
        {
            // not connected?
            if(!adjMat.isConnected(currentNodeIndex, j))
                continue;

            // Is the foreign node already in the tree?
            // This is the case, if the parent element of the TreeNode does not point to itself!
            if(aTree[j].parent != &aTree[j])
            {
                // Fundamental Cycle found!
                // Get unique paths from both nodes within the spanning tree!
                HalfAdjacencyMatrix pi(m_aNodes.size()), pj(m_aNodes.size());
                unique_tree_path(&aTree[currentNodeIndex], pi);
                unique_tree_path(&aTree[j], pj);

                // also the connection between currentNodeIndex and j has to be inserted
                // to ONE of the two paths (which one does not matter)
                pi.connect(currentNodeIndex, j);

                // combine the two matrices with XOR (^) to obtain the fundamental cycle.
                m_aFundamentalCycles.push_back(pi ^ pj);
            }
            else
            {
                // The foreign node is not contained in the tree yet; add it now!
                aTree[j].parent = &currentTreeNode;
                // add the node to the search stack!
```

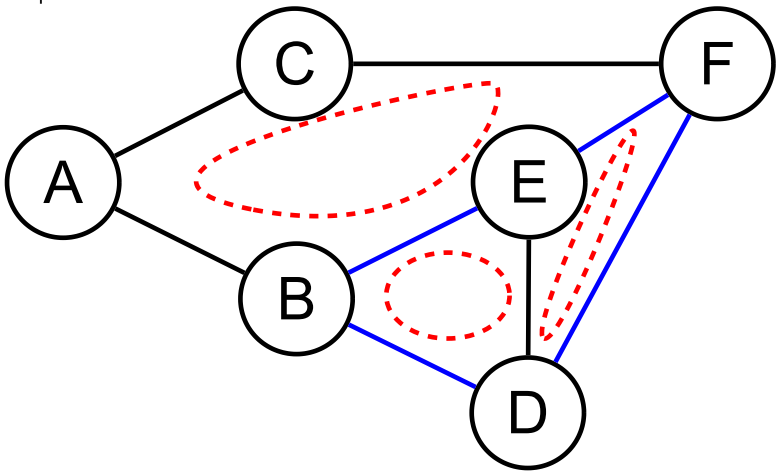
```
                nodeStack.push(j);
            }
            // Either way remove this connection!
            adjMat.disconnect(currentNodeIndex, j);
        }
    }
}
```

4. Iterating Through All Possible Cycles

In this last section, we use the set of fundamental cycles obtained as a basis to generate all possible cycles of the graph. As the set of fundamental cycles is complete, it is guaranteed that *all* possible cycles will be obtained.

a) How to Combine Fundamental Cycles?

To combine two cycles again, the XOR operator can be used. Assume the three fundamental cycles (A-B-E-F-C-A; B-D-E-B; D-E-F-D) illustrated with red dotted lines are found by our algorithm as complete basis:



As an example, combining the two cycles B-D-E-B and D-E-F-D using XOR will erase the edge D-E and yields the circle B-D-F-E-B (blue lines).

However, it is not sufficient to just combine pairs of circles because then the encircling cycle (A-B-D-F-C-A) would not be found which is only obtained if all *three* fundamental cycles are combined, erasing the edges B-E, D-E and E-F. In general, it is necessary to iterate through all possible tuples of fundamental cycles starting with pairs and ending with the N_{FC}-tuple (total number of fundamental cycles).

b) How to Represent a Tuple of Fundamental Cycles?

Straightforwardly, tuples of fundamental cycles can be represented in the code by a `bitstring` of length N_{FC}. For the example graph, the `bitstring` would therefore be of length `3` yielding the following possible combinations of the three fundamental cycles (FCs):

Bitstring	XOR Combination	Cycle
100	FC1	A-B-E-F-C-A
010	FC2	B-D-E-B
001	FC3	D-E-F-D
110	FC1 ^ FC2	A-B-D-E-F-C-A
101	FC1 ^ FC3	A-B-E-D-F-C-A
011	FC2 ^ FC3	B-D-F-E-B
111	FC1 ^ FC2 ^ FC3	A-B-D-F-C-A

Within the representation of `bitstring`s, all possible cycles are enumerated, i.e., visited, if all possible permutations of all `bitstring`s with $2 \leq k \leq N_{\mathrm{FC}}$, where k is the number of 1s in the `string`, are enumerated.
E.g., if a graph has four fundamental cycles, we would have to iterate through all permutations of the bitstrings, `1100`, `1110` and `1111` being `11` iterations in total.

c) Combinatorics

Let's talk about some math at this point to see how this approach scales. Starting with pairs, we have to know how many permutations of 2 ones in a bitstring of N_{FC} are possible. This number is directly given by the binomial coefficient of N_{FC} choose 2". In general, if we want to know how many permutations of k ones in a bitstring of length N_{FC} are possible, this number is given by the binomial coefficient of N_{FC} choose k". To get the total number of combinations of fundamental cycles, the binomial coefficients starting from $k = 2$ to $k = N_{\mathrm{FC}}$ have to be summed up yielding the following equation:

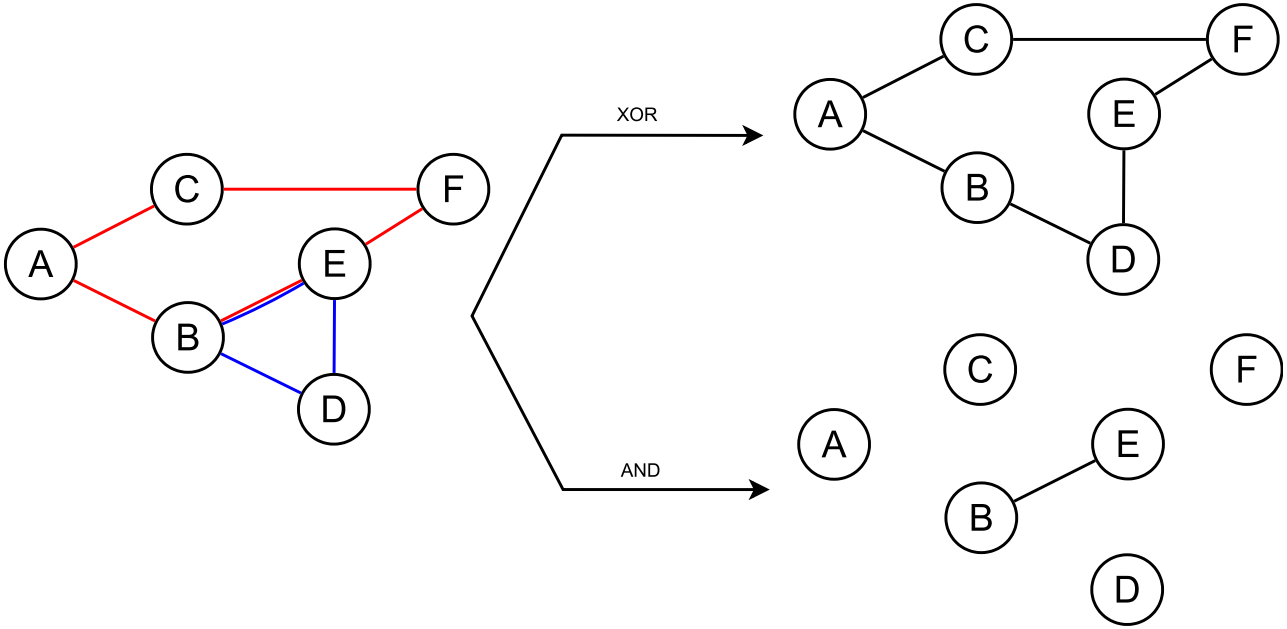
$$\sum_{k=2}^{N=N_{\mathrm{FC}}} \binom{N}{k} = \sum_{k=0}^{N} \binom{N}{k} - \binom{N}{1} - \binom{N}{0} = 2^N - N - 1$$

The code therefore scales exponential with the *number of fundamental cycles* in the graph. Exponential scaling is always a problem because of the vast number of iterations, it is usually not possible to iterate through all combinations as soon as N grows in size. To get an impression of the scaling, we estimate that one iteration needs 10ms to be computed. Then one would need 10 seconds for $N = 10$ but approximately 11 years for $N = 35$. One can easily see that the time needed for one iteration becomes negligible as soon as N becomes large enough yielding an unsolvable problem.

Note that this is only true if one would really want to enumerate each and every possible cycle. However, for most questions, it is sufficient to just be *in principle* able to visit every cycle without doing so, e.g. heuristical algorithms, Monte Carlo or Evolutionary algorithms. In general, it is therefore a good idea to rethink the question, asked to the graph, if an enumeration of all possible cycles of a graph is necessary.

d) Cycle Validation

Now that we know how to combine the different fundamental cycles, there is still one problem left which is related to the XOR operator: Combining two disjoint cycles with an XOR operation will again lead two disjoint cycles. Therefore, each combination must be validated to ensure that one joint cycle is generated.
Let's start with how to check if a pair of fundamental cycles generates one adjoint cycle. This is rather straightforward because we just have to apply the `AND` operator and check if there are edges belonging to both cycles. This check can be integrated into the `XOR` operation directly: If one or more edges are cleaved by the operation, then the two cycles have at least one edge in common and generate a new valid cycle.



For higher tuples, the validation unfortunately is not that simple: Consider merging three cycles, then it is *necessary* that at least two edges are cleaved during the XOR operation. However, this test is not *sufficient* because two of the three cycles could have two edges in common and the third cycle is disjoint. One option would be to keep track of all pairs and check if edges are cleaved between a valid pair and the third cycle but this would result in two major disadvantages:

1. All possible pairs of fundamental cycles have to be computed before triples can be computed. As soon if we have to deal with quadruples, quintuples or higher tuples all "lower" tuples have to be computed before the higher tuples can be evaluated. Thus random accessing any possible bitstring is not possible anymore.
2. Recall that given by the combinatorics this method would require a vast amount of memory to store valid combinations.

Therefore, I will use a very simple approach which might not be the most efficient one: For each k-tuple combination where $k > 2$ a depth search algorithm will be used to check if the merged substructure in the `CycleMatrix` (typedef `HalfAdjacencyMatrix`) is completely connected. This is straightforwardly implemented as just the visited edges have to be counted. If this number is equal to the total number of edges, then the tuple formed one adjoined cycle.

```
bool validateCycleMatrix(const CycleMatrix& m)
{
    // We will use our knowledge on the cycle matrices we are using:
    // We know that all nodes in the matrix which belong to the cycle have exactly 2 edges.
    // when we now start a deep search from any node in the matrix and counting the path length
    // to the starting node this length must be equal to the
```

```
    // total number of edges
    // Again this is exhaustive but it is a very simple approach validating the cycles

    size_t pathLength = 0;
    // Find any edge in the matrix:
    for (size_t i = 0; i < m_aNodes.size(); ++i)
    {
        for (size_t j = 0; j < m_aNodes.size(); ++j)
        {
            if (m.isConnected(i, j))
            {
                // Increment the pathLength and start the recursion
                ++pathLength;
                std::set<size_t> aVisited;
                aVisited.insert(i);
                validateCycleMatrix_recursion(m, pathLength, j, i, aVisited);

                // Version 3:
                //      - From the recursion, the path length will not account
                //          for the last edge connecting the starting node
                //        with the last node from the recursion.
                return pathLength + 1 == m.getNumEdges();
            }
        }
    }
    // When we are here, the matrix does not contain any edges!
    throw std::runtime_error("Graph::validateCycleMatrix():
                Given Cycle Matrix does not contain any edges!");
}
```

The method `validateCycleMatrix` just takes the `CycleMatrix` which is to be validated. Then it looks for the first present edge and starts a depth search (which is related to the same algorithm already used to determine the spanning tree) recursively using `validateCycleMatrix_recursion`. The cycle is valid if the number of edges visited by the depth search equals the number of total edges in the `CycleMatrix`.

```
// i: The node which has to be investigated in the current step
// previousNode: The node which was investigated before node i; necessary to avoid going backwards
// startNode: The node which was investigated first; necessary to determine
// when the recursion can be stopped

void validateCycleMatrix_recursion(const CycleMatrix& m, size_t& pathLength,
         const size_t i, size_t previousNode, std::set<size_t>& aVisited) const
{
    // The path length is also a measure for the recursion steps.
    // If the recursion takes too long, we abort it and throw an error message.
    // If you expect cycles which are longer than 500 edges, you have to increase this number.
    // Also note that there is a limit of maximal recursion levels which cannot be exceeded.
```

```
    // If your cycles exceed that maximum length,
    // you will have to come up with another validation method.
    if (pathLength > 500)
        throw runtime_error
            ("Graph::validateCycleMatrix_recursion(): Maximum recursion level reached.");

    // Find the next connection of the given node, not going back
    for (size_t j = 0; j < m_aNodes.size(); ++j)
    {
        // Are the two elements connected? Ensure that we are not going backwards
        if (m.isConnected(i, j) && j != previousNode)
        {
            // Was this node not visisted before?
            auto ppVisited = aVisited.find(j);
            if (ppVisited != aVisited.end())
            {
                // This node was already visited, therefore we are done here!
                return;
            }

            // This node was not visited yet, increment the path length and
            // insert this node to the visited list:
            ++pathLength;
            aVisited.insert(i);

            // Call the next recursion:
            validateCycleMatrix_recursion(m, pathLength, j, i, aVisited);
            return;
        }
    }
    // When we are here, we have found a dead end!
    throw std::runtime_error("Graph::validateCycleMatrix_recursion(): Found a dead end!");
}
```

e) Implementation

In the following, all steps necessary to enumerate all cycles of the graph are summarized in one single function which tries to save all cycles in the class; if possible. The code also offers an iterator (`CycleIterator`) which follows an C++ input iterator. Note that this function's purpose is mainly to illustrate how to put all ends described in the previous sections together and it will literally take for ages if the cycle rank of the given graph is large enough.

```
// Exhaustive!!!
void computeAllCycles()
{
    // if the fundamental cycles are not determined yet do it now!
    if(m_aFundamentalCycles.empty())
```

```
        computeFundamentalCycles();

    // all fundamental cycles also are cycles...
    m_aCycles = m_aFundamentalCycles;

    // The bitstring
    std::vector<bool> v(m_aFundamentalCycles.size());
    // Combine each fundamental cycle with any other.
    // attention: not only pairing (M_i ^ M_j) is relevant but also all other tuples
    // (M_i ^ M_j ^ ... ^ M_N)! quite exhausting...
    // we pick r cycles from all fundamental cycles; starting with 2 cycles (pairs)
    for(size_t r = 2; r <= m_aFundamentalCycles.size(); ++r)
    {
        // Fill the bitstring with r times true and N-r times 0.
        std::fill_n(v.begin(), r, 1);
        std::fill_n(v.rbegin(), v.size() - r, 0);

        // The following code in the original source caused an error and is
        // therefore replaced by the line above
        // if (r < 5)
        //    std::fill_n(v.begin() + r + 1, 5 - r - 1, 0);

        // Iterate through all combinations how r elements can be picked from N total cycles
        do
        {
            // Building the cycle matrix based on the current bitstring
            CycleMatrix M(m_aNodes.size());
            size_t nEdges = 0;
            for (size_t i = 0; i < m_aFundamentalCycles.size(); ++i)
                if (v[i])
                {
                    M ^= m_aFundamentalCycles[i];
                    nEdges += m_aFundamentalCycles[i].getNumEdges();
                }

            // as long as pairs are merged the validation is straightforward.
            if(r == 2)
            {
                // When at least one edge was deleted from the adjacency matrix
                // then the two fundamental cycles form one connected cycle
                // as they shared at least one edge.
                if(nEdges > M.getNumEdges())
                    m_aCycles.push_back(M);
            }
            else
            {
                // Here we have combined more than two cycles and the
```

```
                // matrix is validated via depth-first search
                if(validateCycleMatrix(M))
                    m_aCycles.push_back(M);
            }
        }
        // the bitstring is build up with 11...00, therefore prev_permutation
        // has to be used instead of next_permutation.
        while (std::prev_permutation(v.begin(), v.end()));
    }
}
```

5. Outlook

The code can straightforwardly be extended to carry weights for each edge and the use of bitstrings to represent each cycle allows one to directly use a genetic algorithm to find longest paths or shortest paths fulfilling certain constraints without actually visiting all possible cycles.

6. Code

The assigned code contains all described classes and functions. There is also an example code which enumerates all cycles of the graph in Fig. 1a. The function `CreateRandomGraph` generates a random graph with a given connection probability for each edge. The code is tested using VC++ 2017 (on Windows) and GCC 6.4.0 (on Linux). Note that the code uses some C++11 features and therefore must be compiled using -std=c++11 or higher (GCC).

7. History

June 2018 - Version 2

Unfortunately, there was a code error in the original post where a debug code remained in the uploaded version. The following code lines were replaced in the function "`Graph::computeAllCycles()`" and "`Graph::CycleIterator::next()`":

```
if (r < 5)
    std::fill_n(v.begin() + r + 1, 5 - r - 1, 0);

// Was replaced by

std::fill_n(v.rbegin(), v.size() - r, 0);
```

September 2018 - Version 3

I uploaded a patch for an error in the `validateCycleMatrix` method: In line number 666, the line:

```
return pathLength == m.getNumEdges();
```

was replaced by:

```
return pathLength + 1 == m.getNumEdges();
```

This change was necessary as the deep search algorithm used to validate the `CycleMatrix` determines the cycle length but does not account for the last edge closing the cycle which connects the last visited node with the starting node. Thus, the total number of edges in the `CycleMatrix` has to be equal to the path length as obtained by the deep search algorithm plus one.
An additional test with a slightly larger graph than in Fig. 1a is added to test the patch.

The code was changed in both, the article and the download source.

References

- [1] *Paton, Keith (1969), "An Algorithm for Finding a Fundamental Set of Cycles of a Graph", Scientific Applications* 12:514
- [2] *Berge, Claude (2001), "Cyclomatic number", The Theory of Graphs, Courier Dover Publications, pp. 27–30*
- [3] Circuit rank. In *Wikipedia*. Retrieved January 07, 2018, from https://en.wikipedia.org/wiki/Circuit_rank

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Philipp Sch
Germany

No Biography provided

C d Discussions

11 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1158232/Enumerating-All-Cycles-in-an-Undirected-Graph** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Article Copyright 2018 by Philipp Sch
Everything else Copyright © CodeProject, 1999-2019

Web04 2.8.191105.1