AN OBJECT-ORIENTED CLUSTER SEARCH ALGORITHM

DMITRY SILIN AND TAD PATZEK

Abstract. In this work we describe two object-oriented cluster search algorithms, which can be applied to a network of an arbitrary structure. First algorithm calculates all connected clusters, whereas the second one finds a path with the minimal number of connections. We estimate the complexity of the algorithm and infer that the number of operations has linear growth with respect to the size of the network.

Introduction

Pore-network modeling approach to study fluid flow in porous media was initiated in pioneering works [5, 6, 7] by Fatt in the fifties. The results obtained by early nineties were summarized in an overview [4] by Entov. More recent results are gathered in [3]. In earlier works, the researchers used structured (*e.g.*, hexagonal or rectangular) networks of capillary tubes with circular cross-sections. Later, more complicated networks with angular pores and throats were introduced for modeling two-phase flow [14]. Recently, the pore-network modeling approach was revisited due to the progress in scanning electron microscopy. With the ability of obtaining pore space images with super-high resolution, it became possible to construct pore networks imitating the structure of the void space of real rock and even reproduce the flow properties [2, 11, 12].

A pore network is a set of nodes and links connecting the nodes. The nodes model the volumetric properties of the pore space, whereas the links define the flow properties. Each link connects two nodes, whereas each node can be connected to several neighboring nodes. The number of neighbors of a node is called the coordination number. The size of a network is characterized by the total number of nodes and links.

In two-phase flow, some links and nodes may be entirely occupied by one of the phases and, therefore, open for the flow of the occupying fluid only. It is important to find connected clusters of nodes open to the flow of a particular fluid phase. A set of nodes makes a connected cluster if between any two nodes there is a chain of links and nodes open to flow. For regular networks, an efficient algorithm of computing all the connected clusters was proposed by Hoshen and Kopelman [9]. This algorithm was designed for studies of molecular clusters in a crystal structure. Therefore, it was assumed that the network is a regular lattice. In pore network flow modeling, the structure of the pore space is usually extremely irregular because the rock skeleton is composed of grains of different sizes and shapes. For irregular networks, a modification of Hoshen–Kopelman algorithm was proposed and implemented in MatLab by Al-Futaisi and Patzek [1]. This modification proved to be very efficient

Key words and phrases. Pore network, cluster, Hoshen-Kopelman algorithm, object-oriented approach.

for various network. Its efficiency was augmented by the smart use of MatLab vectorized operations.

In present work, we describe a new algorithm for calculating the clusters of connected nodes in an arbitrary network. The algorithm is based on the object-oriented approach and it is of linear complexity. Two major versions are presented. The first one computes all the connected clusters of nodes. The other one checks if there is a connection between the inlet and the outlet of the network and computes a minimal path in case such a connection exists. The first algorithm is based on "Depth First" principle, whereas the other one is a variation of "Breadth First" search [10].

The paper is organized as follows. In the Section 1, we briefly introduce the basic concepts of object-oriented programming approach, which are used in explaining the algorithm. In Section 2 we define the node and link classes of objects and define the network. In Section 3, we describe the first algorithm and estimate its complexity in Section 4. In the last section we present the second algorithm, which computes the network inlet-outlet connection. In Appendix, we provide C++ code implementing both algorithms.

1. Object-oriented approach

Object-oriented approach in programming (OOP) is a powerful development tool. For the purposes of the present work, we overview only some aspects of this approach. More information about OOP can be found in numerous publications that appeared during the last two decades, see e.g. [8] and the references therein.

An *object*, by definition, is an entity that packages both data and the procedures that operate on those data [8]. The object data are usually called the properties and the procedures are called the methods or interface. The similarity between the objects is determined by the defining *class*. In a sense, an object is the next step of complication from a *structure*. To derive a class of objects from a structure the latter has to be supplied with an interface. For the procedures affiliated with a given object, the data are treated as global, whereas for the rest of the code the data are usually encapsulated and often cannot be accessed directly. One of the benefits of the OOP approach is the possibility of creating customized tools needed specifically for the problems in question. These tools can be reused or easily extended through the mechanism of *inheritance*, assembled into aggregate objects, etc.

In the case of pore network modeling, our main objects are nodes, links and the entire network. The nodes and links are discussed in more detail in the next section. As an object, the network consists of the list of all nodes and the list of all links along with a structure describing the material properties of the fluids and the solid, see Fig. 1. A *list*, in turn, is a generic object containing an array of similar objects. In C++, a list can be implemented based on Standard Template Library (STL) [15]. The main features of a list include the possibility to easily attach or remove an item and to scan the items through the mechanism of *iterators*. Inasmuch as a link and a node are objects of two different classes, the list of all links and the list of all nodes cannot be merged into one single list.

The implementations of the algorithms described below also use the mechanism of *pointers*. A pointer is a logical address of the object in the memory of the computer. The usage of pointers eliminates an unnecessary overhead in calling functions.

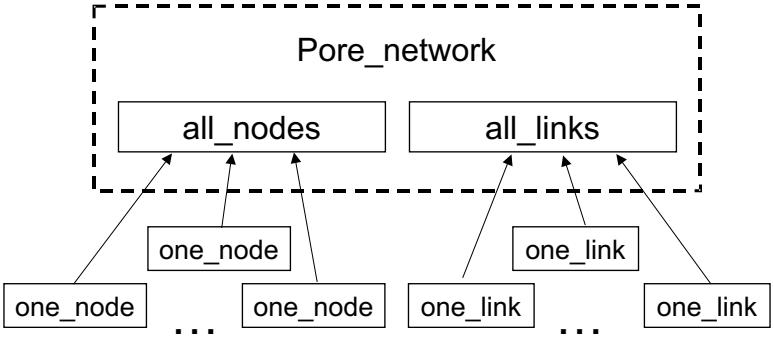

FIGURE 1. The structure of a pore network object

2. Nodes and links

In this section, we define the main objects operated by the algorithm. The algorithm has been implemented in C++ flow simulation code NetSimCPP [13]. NetSimCPP, in turn, is a derivative of MatLab code ANetSim by Tad Patzek implementing the conclusions of [12]. The objects in NetSimCPP were used for much more general tasks than in the present paper, therefore we present them in a reduced form.

Each node object belongs to the class `one_node` described in Appendix. Each node has a unique integer index, which we call `ID`. There are also two integer parameters: an openness flag equal to one if the node is open for a given fluid phase flow or zero otherwise and `cluster_no` (which is equal to the index of the cluster whose element the node is). The last parameter is reused in the second algorithm described in Section 5 for labelling the "already checked" nodes. This algorithm also requires information whether node is at the inlet or at the outlet of the network. Two parameters are reserved for this purpose. Two pointers to a node and to a link are used to store the information about the node and link, from which this node was approached by the algorithm. Each node also includes a list of pointers to the links connecting the node to the neighbors and a list of pointers to these neighbors. Obviously, both lists have the same number of items, which is equal to the *coordination number* of the node. Different nodes may have different coordination numbers, so the length of the lists of links and neighbors is not fixed. A node object also includes a method, which we will call `add_to_cluster`, which is used in computing the clusters, and a method called `next_step` used in calculating the inlet-outlet connection.

The link object properties include an integer *ID*, two node pointers to the nodes connected by the link and an integer flag signifying the openness of the link, see class `one_link` in the Appendix.

As we have already mentioned above, the entire network is represented by two lists: the list of all nodes and the list of all links, see Fig. 1. Such an object is defined by class `pore_network` in the Appendix.

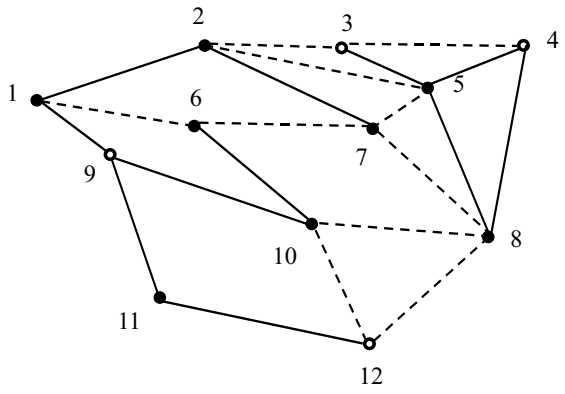

FIGURE 2. Network example

3. The algorithm

We assume that at the beginning the objects are setup, so that each node "knows" all its neighbors and each link "knows" which nodes it connects and the parameters `node_open` and link_open are defined appropriately. Initially, the `cluster_no` is zero for every node.

The list of nodes is scanned by calling for each node the function `add_to_cluster` with an integer parameter `cluster_count`. This parameter is increased by one if the return value of the function is nonzero.

As a method of the node class, the function `add_to_cluster` can access the properties of the current node or properties of the other nodes through their pointers. The execution of the function `add_to_cluster(cluster_count)` proceeds in the following way (the references to the respective lines of the code in Appendix are provided in parentheses).

Step 1. If `cluster_no` of this node is nonzero or the flag does not equal one, return immediately with return value zero (lines 3–4).

Step 2. Set `cluster_no` equal to `cluster_count` (line 5).

Step 3. Scan all links and neighbor nodes: if the current link is open, then call function `add_to_cluster(cluster_count)` for the respective node (lines 6–14).

Step 4. Return with return value 1 (line 15).

The function `add_to_cluster(cluster_count)` is called recursively for different nodes. Clearly, if the outermost call returns nonzero, it means that a new cluster has been created and the `cluster_count` must be increased by one.

Let us illustrate the application of the algorithm on a simple example. Consider the network shown in Fig. 2. The nodes and links open for the flow are shown as filled circles and solid lines, respectively. The nodes and links which are not open are shown as non-filled circles and dashed lines. Clearly the network has four clusters: 1-2-7, 5-8, 6-10 and 11. Let us track the work of the algorithm.

Step 1. Set `cluster_count` equal to one and call function `add_to_cluster` for the first node. Since node 1 is open, its `cluster_no` property is assigned to `cluster_count`, *i.e.*, to one. Then the lists of links and neighbors are

scanned. First link 1-2 is open, therefore the function `add_to_cluster` is called for node 2. Again the second node is open, therefore, the `cluster_no` property is set equal to one and the list of links and neighbors of node 2 is scanned. Links 2-3 and 2-5 are skipped because the flag is equal to zero. Since both link 2-7 and node 7 are open and the node is still "non-aligned" with any cluster, the `cluster_no` is set equal to the current `cluster_count` and the links and neighbors of node 7 are scanned. Clearly, links 7-5, 7-8 and 7-6 are skipped, and checking link 7-2 results in no action because the `cluster_no` property of node 2 is already nonzero. Thus, the function returns from node 7, which, in turns completes scanning the links and neighbor nodes of node 2. Continued scanning of the links and neighbors of node 1 results in no action because link 1-6 is not open, as well as the neighbor node 9. The first step of the algorithm results in creating the first cluster 1-2-7 and the `cluster_count` is incremented by one.

Step 2. For nodes 2–4, the function `add_to_cluster` immediately returns zero because the `cluster_no` of node 2 is nonzero and nodes 3 and 4 are not open.

Step 3. For node 5, the function `add_to_cluster` assigns the `cluster_no` equal to the current `cluster_count`, *i.e.*, two. Scanning nodes 3 and 4 produces no change because they are not open, nodes 2 and 5 are skipped because the respective links are not open. Therefore, the function `add_to_cluster` is only called for node 8. The latter has no open links and neighbor nodes except node 5 whose `cluster_no` property is already nonzero, therefore no action is performed. Thus, as the result of step 3, the new cluster 5-8 is detected and `cluster_count` is increased by one.

Step 4. Clearly, scanning node 6 results in detecting cluster 6-7 and further increment of `cluster_count`.

Step 5. Nodes 7–8 are skipped because they are already assigned to clusters, *i.e.*, their respective `cluster_no` properties are different from zero. Node 9 is not open, therefore no action is performed. Node 10 is already assigned to a cluster, so no action here as well.

Step 6. Scanning node 11 results in a single-element cluster because the two neighbors 9 and 12 are not open.

Step 7. Finally, node 12 is not open, so no action is performed.

Thus, the algorithm stops after detecting all four clusters.

4. Estimate of complexity

To estimate complexity of the algorithm described above, denote by N_n the total number of nodes and let N_l be the total number of links. We will call the sum $N_n + N_l$ the size of the network.

Once a node is assigned to a cluster, steps 2–4 of the method `add_to_cluster` (lines 5–15) are never performed on this node again. Therefore, this part of `add_to_cluster` is performed no more than N_n times. For a given node, the "`if`" operation in step 1 can be performed several times. Indeed, the method `add_to_cluster` is called whenever the node is approached from a just incorporated in the same cluster neighbor through the respective open link. Each link connecting two open nodes can be explored no more than twice: when the first node scans its neighbors after being just "signed-up" to a cluster and when the other node scans its links and neighbors. A link connecting an open node to a

closed one is checked only once from the open node. A link connecting two closed nodes is never reached by the algorithm. Therefore, the total number of checks of the links is no more than twice the number of links N_l. Each check of the link may or may not result in a call of function `add_to_cluster` for one of the end-nodes. Thus, the total number of calls of `add_to_cluster` is estimated from above by $N_n + N_l$, which is a linear combination of N_n and N_l.

To summarize, the step 1 of function `add_to_cluster` is called $O(N_n+N_l)$ times, whereas steps 2–4 are called no more than N_n times. In other words, the complexity is linear with respect to the size of the network.

The above-described algorithm detects and calculates the connected clusters in the network but does not provide any information about the sizes of the clusters. The sizes can be easily evaluated by counting the nodes with the same parameter `cluster_no`, which requires an additional scan of the list of the nodes . However, the function `add_to_cluster` can be slightly modified so that the cluster sizes will be computed within the algorithm. Using the *polymorphism* principle, the function `add_to_cluster` can be *overloaded* by creating a different version with a different set of parameters. Namely, consider a function `add_to_cluster` of two integer parameters: `cluster_count` and `cluster_size`. The modified function works in the following way:

Step 1. If `cluster_no` of this node is nonzero or the flag does not equal one, return immediately with return value `cluster_size`.

Step 2. Set `cluster_no` equal to `cluster_count` and increase `cluster_size` by one.

Step 3. Scan all links and neighbor nodes: if the current link is open, then call function `add_to_cluster(cluster_count, cluster_size)` for the respective node assigning the return value to `cluster_size`.

Step 4. Return the value of `cluster_size`.

The return value of the outermost call of function `add_to_cluster` is equal to the size of the cluster created by this call. In particular, if this value equals zero, then no new cluster has been created and no increment of `cluster_count` is needed. Clearly, this modification does not change the linear estimate of the complexity of the algorithm. A code implementation of this modification of the algorithm is straightforward and it is not presented in the Appendix.

5. Inlet-outlet connection

In some situations, as in the flow-simulation code NetSimCPP [13], parts of the whole network are singled out as inlet and outlet. The inlet nodes are the ones through which the fluids enter the network and the outlet nodes are those through which the fluids can leave the network. The flow of a given phase is possible only if there is a chain of nodes and links open to the flow of this phase and spanning from the inlet to the outlet, *e.g.*, the network 10-8-11-4 in Fig. 3. To figure out whether such a chain exists it is unnecessary to compute all the clusters. We describe such an algorithm in this section.

Indeed, the function `add_to_cluster` can be called only from the inlet nodes. This does not prevent recursive calls of this function from the interior nodes. The computations can be stopped as soon as an outlet node has been reached. A modification of the algorithm implementing this approach is described below. The recursive function calls are replaced by maintaining the list of recently checked

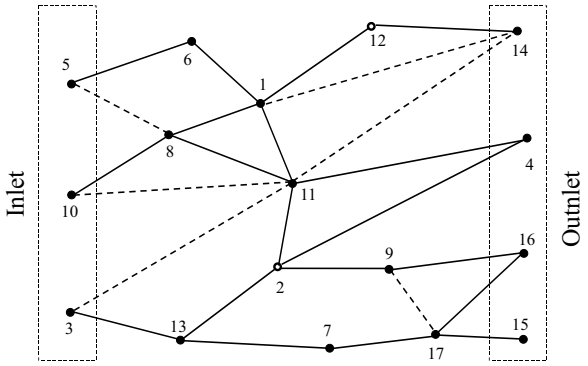

FIGURE 3. Network example: the shortest path between inlet and outlet is 10-8-11-4

nodes. As a byproduct, the minimal number of links connecting an inlet node to an outlet node is computed. As above, this modification can be implemented through overloading the original function `add_to_cluster`.

At the node level, the algorithm is implemented by method `next_step`, see the Appendix. This function is called by a `pore_network` method, which we call `find_inlet_outlet` first for all open inlet nodes. Then this function is called for all nodes on the list constructed by this function and passed by a pointer as a parameter. If an outlet node is encountered, then the function returns the pointer to this node, otherwise it returns `NULL`. It works in the following way.

Step 1. If `cluster_no` of a node is nonzero or the flag does not equal one, return immediately with return value `NULL` (lines 4–5).

Step 2. Scan all the links and the neighbor nodes. If the link is open, check the respective neighbor node. If it is open and, simultaneously, has not been checked yet and is not at the inlet, pass "this" node pointer and the pointer to the link to the neighbor's properties `who_called` and `how_called`, respectively (lines 15–16). If the neighbor is at the outlet, return its pointer (lines 17–19). Otherwise, label the neighbor as checked (line 22) and add its pointer to the list of "front" nodes (lines 21–22).

Step 3. If no outlet node has been encountered, return `NULL` pointer (line 25).

To check whether the inlet and the outlet of a network are connected, a pore network method, which we call `find_inlet_outlet`, is used. This function is called with the pointers to a list of nodes and a list of links as parameters. These lists are used to output the inlet–outlet chain of nodes and links if such a chain exists. Initially, both lists must be empty. If such a chain does exist, then the return value is the length of the path, otherwise the return value is zero. The method works in the following way.

Step 1. Create two lists of nodes for storing the current list of checked nodes and a list of new checked nodes (lines 5–7).

Step 2. Put all open inlet nodes in the current list (lines 10–16)

Step 3. If the current list is not empty, call `next_step` for each node of this list with the pointer to the new list as the parameter (lines 24–47). If an outlet node is encountered, that is signalled by a nonzero return value of

`next_step`, stop scanning (lines 31–42), otherwise swap the lists and iterate the procedure (lines 43–46).

Step 4. If an inlet-outlet connection has been detected, build the respective spanning chain of nodes and links using information stores in the properties `who_called` and `how_called` in every checked node (lines 55–65).

Step 5. Return the length of the found minimal path equal to the number of links (line 68) or zero if the inlet and outlet are not connected (lines 70–72)

Clearly, every link is checked no more than twice, therefore, the estimate of the function calls remains the same as above: $O(N_n + N_l)$.

Conclusions

In this work, we have described two object-oriented algorithms of cluster computation for a pore network. The first one labels every node of the network by an integer number which is the index of the cluster to which the node belongs. The second algorithm checks whether there is a cluster spanning from the inlet to the outlet of the network and calculates the shortest path through the network.

We have demonstrated that the complexity of both algorithms has a linear estimate with respect to the size of the network.

Acknowledgements

This work was sponsored by the DOE OGRT Partnership Program under Contract DE-ACO3-76FS0098 to the Lawrence Berkeley National Laboratory. Partial support was also provided by gifts from ChevronTexaco and ConocoPhillips to UC Oil, Berkeley.

References

1. A. Al-Futaisi and T. W. Patzek, *Extensiojn of the Hoshen-Koppelamn algorithm to a nonlattice environment*, Physical Review A **6** (2002), no. 3, 197–207.
2. S. Bakke and P. E. Øren, *3-d pore-scale modelling of heterogeneous sandstone reservoir rocks and quantitative analysis of the architecture, geometry and spacial continuity of the pore network. spe 35479*, European 3-D Reservoir Modelling Conference (Stavanger, Norway), SPE, 1996, pp. 35–45.
3. M. J. Blunt, *Flow in porous media - pore-network models and multiphase flow*, Current Opinion in Colloid & Interface science **6** (2001), no. 3, 197–207.
4. V. M. Entov, *The micromechanics of flow in porous media*, Soviet Academy Izvestia. Mechanics of Gas and Fluids (1992), no. 6, 90–102.
5. I. Fatt, *The network nodel of porous media. 1. Capillary pressure characteristics*, Trans. AIME **207** (1956), no. 7, 144–159.
6. ———, *The network nodel of porous media. 2. Dynamic properties of a single size tube network*, Trans. AIME **207** (1956), no. 7, 160–163.
7. ———, *The network nodel of porous media. 3. Dynamic propertries of networks with tube radius distribution*, Trans. AIME **207** (1956), no. 7, 164–181.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns. Elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.
9. J. Hoshen and R. Kopelman, *Percolation and cluster distribution i. Cluster multiple labeling technique and critical concentration algorithm*, Physical Review B **14** (1976), no. 8, 3438–3445.
10. Donald Knuth, *The art of computer programming*, vol. 1, Addison-Wesley Pub. Co., Reading, MA, 1968.
11. P. E. Øren, S. Bakke, and O. J. Arntzen, *Extending predictive capabilities to network models*, SPE Journal (1998), no. December, 324–336.

12. T. W. Patzek, *Verification of a complete pore network simulator of drainage and imbibition*, SPE Journal **6** (2001), no. 2, 144–156.
13. Dmitry Silin and Tad Patzek, *NetSimCPP: Object-oriented pore network simulator*, Lawrence Berkeley National Laboratory, 2001.
14. A. K. Singhal and W. H. Somerton, *Two-phase flow through a non-circular capillary at low reynolds numbers*, Technology (1970), 197–205.
15. Alexander Stepanov and Meng Lee, *The standard template library*, Hewlett Paccard Laboratories, 1995.

Appendix

Here we present the C++ code implementing the algorithms described above. Defining classes, we skip the constructors and destructors as technical details.

We start with the node object. Its class is described by

```
#include<list>;
using namespace std;
class one_node
{
  private:
    int ID;              // Unique index of this node
    int node_open;       // =1 if open =0 otherwise
    int cluster_no;      // =0 initially
    int inlet;    // =1 for an inlet node =0 otherwise
    int outlet;   // =1 for is an outlet node =0 otherwise
    list<one_node*> neighbor_nodes;
    list<one_link*> links;
    one_node* who_called;
    one_link* how_called;
  public:
// Constructors and destructor
    one_node();
    one_node(const one_node&);
    ~one_node(){;}
// Cluster search methods
    int is_open() const {return node_open;}
    int is_inlet()  const {return inlet;}
    int is_outlet() const {return outlet;}
    int add_to_cluster(int Cluster);
    one_node* next_step(list<one_node*>*);
    one_node* trace_back_node() const {return who_called;}
    one_link* trace_back_link() const {return how_called;}
};
```

The specifier **const** indicates the methods, which do not change the properties of the object one_node. The object one_link is even simpler:

```
class one_link
{
  private:
```

```
        int link_open; // =1 if link_open =0 otherwise
        one_node* end_nodes[2];
        int ID;          // Unique index of this link
    public:
// Constructors and destructor
        one_link();
        one_link(const one_link&);
        ~one_link(){;}
// Cluster search methods
        int is_open() const {return link_open;}
};
```

The methods add_to_cluster are defined according to the algorithms above. For algorithm 1–4 the function have the following form:

```
int one_node::add_to_cluster(int Cluster)
{
   if (!open || cluster_no)
      return 0;
   cluster_no = Cluster;
   node_list::iterator next_neighbor
                        = neighbor_nodes.begin();
   link_list::iterator next_link = links.begin();
   for (unsigned i=0; i<neighbor_nodes.size(); ++i)
   {
      if ( (*next_link)->is_open())
         (*next_neighbor)->add_to_cluster(Cluster);
      ++next_neighbor;
      ++next_link;
   }
   return 1;
}
```

For brevity, we define two new types

```
typedef list<one_node*> node_list;
typedef list<one_link*> link_list;
```

describing lists of pointers to nodes and links, respectively.

Method `next_step` is slightly more complicated:

```
one_node* one_node::next_step( node_list* front_nodes)
{
   if (!open)
      return NULL;
   node_list::iterator next_neighbor
                        = neighbor_nodes.begin();
   link_list::iterator next_link = links.begin();
   for (unsigned i=0; i<neighbor_nodes.size(); ++i)
```

```
    {
10      if ( (*next_link)->is_open())
           if ((*next_neighbor)->open
                    && !(*next_neighbor)->cluster_no
                    && !(*next_neighbor)->inlet)
           {
15            (*next_neighbor)->who_called = this;
              (*next_neighbor)->how_called = *next_link;
              if ((*next_neighbor)->outlet)
              // Done if at the outlet
                 return *next_neighbor;
20            // Label the neighbor as 'visited'
              (*next_neighbor)->cluster_no = 1;
              front_nodes->push_back(*next_neighbor);
           }
        ++next_neighbor;
25      ++next_link;
    }
    return NULL;
}
```

The entire network consists of a list of nodes and a list of links. Note that the network as an object "does not know" which link connects which nodes: all this information is distributed among individual nodes and links. Thus, network object has the following structure:

```
#include<list>;
using namespace std;
class pore_network
{
5   private:
       node_list all_nodes;
       link_list all_links;
    public:
// Constructors and destructors
10     pore_network(){;}
       pore_network(const pore_network&);
       ~pore_network(){;}
// Cluster search methods
       int find_all_clusters();
15     int find_inlet_outlet(node_list*, link_list*);
};
```

To compute all the clusters means to assign the property cluster_no to the number of the cluster for each open node. This result is achieved by calling the following method:

```
int pore_network::find_all_clusters()
```

```
{
   node_list::iterator next_node = all_nodes.begin();
   int cluster_count = 1;
   for (unsigned i=0; i<all_nodes.size(); ++i)
      cluster_count +=
        (*next_node++)->add_to_cluster(cluster_count);
   return --cluster_count;
}
```

To find out whether there is a cluster of open nodes and links spanning from the inlet to the outlet and to find a shortest chain the following method can be used:

```
int pore_network::find_inlet_outlet(node_list* node_path,
                                    link_list* link_path)
{
// Create two lists of node pointers
    node_list front_nodes[2];
    node_list* curr_front = &front_nodes[0];
    node_list* next_front = &front_nodes[1];
    node_list* temp_front;

// Put pointers to all inlet nodes in front_nodes[0];
    node_list::iterator next_node = all_nodes.begin();
    for (unsigned i=0; i<all_nodes.size(); ++i)
    {
        if( (*next_node)->is_inlet()
                        && (*next_node)->is_open() )
            curr_front->push_back(*next_node);
        ++next_node;
    }

    int chain_length = 0;
    int inlet_outlet_connected = 0;
    node_list::iterator curr_front_node;
    one_node* the_node;
// Find the next from nodes
    while ( !curr_front->empty() )
    {
       ++chain_length;
        curr_front_node = curr_front->begin();
        for (unsigned i=0; i<curr_front->size(); ++i)
        {
            the_node =
              (*curr_front_node++)->next_step(next_front);
// Reminder: if an outlet node is encountered,
//add_to_cluster returns the pointer to this node
```

```
            if (the_node)
            {
// Stop iterations if an outlet node has been found
               inlet_outlet_connected = 1;
               break;
            }
         }
// Swap the front node lists
         if (inlet_outlet_connected)
           break;
         curr_front->clear();
         temp_front = curr_front;
         curr_front = next_front;
         next_front = temp_front;
      }
// Build the chains of nodes and links
// spanning through the network
      one_link* next_link;

      if (inlet_outlet_connected)
      {
// Find the spanning chain of nodes and links
         node_path->push_front(the_node);
         next_link = the_node->trace_back_link();
         the_node = the_node->trace_back_node();

         while (the_node)
         {
            link_path->push_front(next_link);
            next_link = the_node->trace_back_link();
            node_path->push_front(the_node);
            the_node = the_node->trace_back_node();
         }
// If there is an inlet-outlet connection
// then the shortest number of links is chain_length
         return chain_length;
      }
       else
// No connection
          return 0;
}
```

This method returns the length of a shortest chain if there is a spanning cluster and zero otherwise.

Lawrence Berkeley National Laboratory 1 Cyclotron Road, MS 90-1116 Berkeley, CA 94720
E-mail address: DSilin@lbl.gov

Lawrence Berkeley National Laboratory 1 Cyclotron Road, MS 90-1116 Berkeley, CA 94720 and Department of Civil and Environmental Engineering, University of California, 437 Davis Hall Berkeley, CA 94720
E-mail address: TWPatzek@lbl.gov