The Shortest Path Algorithms of Hypergraphs Based on Search Strategies

Xinquan Chen[1,2*]

[1] School of Computer Science & Engineering, Chongqing Three Gorges University, Chongqing, China.
[2] Web Sciences Center, University of Electronic Science & Technologhy of China, Chengdu, China.

* Corresponding author. Tel.: 0086-23-58105157; email: chenxqscut@126.com
Manuscript submitted March 22, 2014; accepted August 23, 2014.

Abstract: In order to search the shortest path from hypergraphs with unweighted hyperedges, a kind of searching algorithm based on the width-first strategy is presented at first. For hypergraphs with weighted hyperedges, a kind of searching algorithm based on the minimum-cost-first strategy to search the shortest path is presented. In order to enhance the reliability of the two algorithms, their correctness is proven in theory. The ideas and outlines of the two algorithms are illuminated by using two examples. Through the simulation of some artificial hypergraphs, the two algorithms are compared in time cost and the average percentage between the number of nodes in branching trees and the number of all nodes. We find that they can get the same average percentage. This result is consistent with the actual instance. In the end, it gives a research expectation to disinter and popularize the two algorithms.

Key words: Hypergraph, minimum-cost-first, shortest path, width-first.

1. Introduction

Due to the expansion of application fields, exploring valuable information from some massive, multidimensional, and multiscale objects attracts the attention of some researchers. In exploring different relationships and interesting characters of complex networks, hypergraph model obtains further research and application. Today, hypergraph [1]-[4], which is an extension of the traditional graph theory, gets more attention in the fields of complex network analysis, Internet searching, data mining, and so on. The shortest path of hypergraph can depict the connection degree of two different nodes in the complex network system from one side. So developing effective and efficient algorithms that search the shortest path from hypergraphs has become an important research task.

As we know, Bellman–Ford–Moore algorithm, which searches the shortest paths from a single source node to all of the other nodes in a weighted graph, is the first shortest path algorithm. Almost at the same time, two famous shortest path algorithms of weighted graphs are proposed by Dijkstra and Floyd respectively. Li ChunMing [5] and Gong Qu *et al*. [6] presented two shortest path algorithms of hypergraphs according to the ideas of searching the shortest paths of graph. This paper presents two more efficient shortest path algorithms of hypergraphs by using two kind of effective search strategies and the methods that constructs the set of associated hyperedges of node, the set of nodes of hyperedge, and the set of associated nodes of node.

The remainder of this paper is organized as follows. Section 2 gives the definition of the shortest path of hypergraphs and some related definitions. Section 3 presents a kind of width-first searching algorithm from

a hypergraph with unweighted hyperedges and a kind of minimum-cost-first searching algorithm from a hypergraph with weighted hyperedges. Section 4 further explains the two algorithms using two examples. Section 5 compares and validates the two algorithms by simulation experiments. Conclusions and future works are presented in Section 6.

2. The Definition of the Shortest Path of Hypergraphs

Suppose a hypergraph H is described by a set of nodes $OS = \{o_1, o_2, \ldots, o_n\}$, a set of weight of nodes $OW = \{ow_1, ow_2, \ldots, ow_n\}$, a set of hyperedges $ES = \{e_1, e_2, \ldots, e_m\}$ that describes the group relationships in the set of nodes OS, and a set of weight of hyperedges $EW = \{ew_1, ew_2, \ldots, ew_m\}$.

Definition 1. Refering to the shortest path concept of graph, in hypergraph H, the length of the shortest hyperedge path from node o_i ($i = 1, 2, \ldots, n$) to another node o_j ($j \neq i$), $SEP(o_i, o_j)$, is defined as:

$$SEP(o_i, o_j) = \min\{(ew_k + \ldots + ew_s)\}, \quad o_i \xrightarrow{e_k} \ldots \xrightarrow{e_s} o_j, k, \ldots, s \in \{1,2,\ldots,m\}. \tag{1}$$

where $(e_k, \ldots, e_s)$ is a sequence of hyperedges from node o_i to node o_j. The shortest hyperedge path from node o_i to node o_j is a sequence of hyperedges with the minimal weight sum of hyperedges. Perhaps there are multiple sequences of hyperedges with the minimal weight sum of hyperedges. If there is no sequence of hyperedges from node o_i to node o_j, then the shortest hyperedge path from node o_i to node o_j does not exist.

Definition 2. Refering to the shortest path concept of graph, in hypergraph H, the length of the shortest node path from node o_i ($i = 1, 2, \ldots, n$) to another node o_j ($j \neq i$), $SNP(o_i, o_j)$, is defined as:

$$SNP(o_i, o_j) = \min\{(ow_i + \ldots + ow_j)\}, \quad o_i \xrightarrow{e_k} \ldots \xrightarrow{e_s} o_j, k, \ldots, s \in \{1,2,\ldots,m\}. \tag{2}$$

Where $(o_i, \ldots, o_j)$ is a sequence of nodes from node o_i to node o_j. The shortest node path from node o_i to node o_j is a sequence of nodes with the minimal weight sum of nodes. Perhaps there are multiple sequences of nodes with the minimal weight sum of nodes. If there is no sequence of nodes from node o_i to node o_j, then the shortest node path from node o_i to node o_j does not exist.

Definition 3. The associated matrix [1, 2], $A = (a_{ik})_{n*m}$, between the set of nodes, $\{o_1, o_2, \ldots, o_n\}$, and the set of hyperedges, $\{e_1, e_2, \ldots, e_m\}$, is defined as:

$$a_{ik} = \begin{cases} 1 & o_i \in e_k \\ 0 & o_i \notin e_k \end{cases}, \quad i = 1,2,\ldots,n; k = 1,2,\ldots,m. \tag{3}$$

Definition 4. The set of hyperedges of node o_i ($i = 1, 2, \ldots, n$) is defined as:

$$AES(o_i) = \{e_k \mid o_i \in e_k, k \in \{1,2,\ldots,m\}\}. \tag{4}$$

Usually, the number of hyperedges of node o_i is regarded as its degree of associated hyperedges, denoted by $|AES(o_i)|$.

Definition 5. The set of nodes of hyperedge e_k ($k = 1, 2, \ldots, m$) is defined as:

$$EOS(e_k) = \{o_i \mid o_i \in e_k, i \in \{1,2,\ldots,n\}\}. \tag{5}$$

Usually, the number of nodes contained in hyperedge e_k is regarded as its degree of associated nodes,

denoted by $|EOS(e_k)|$.

Definition 6. The set of associated nodes of node o_i (i = 1, 2, …, n) is defined as:

$$AOS(o_i) = \{o_j \mid \{o_j, o_i\} \subseteq e_k, j \neq i, j \in \{1,2,\ldots,n\}; k \in \{1,2,\ldots,m\}\}. \quad (6)$$

Usually, the number of nodes shared with node o_i in common hyperedges is regarded as its degree of straight-connected nodes, denoted by $|AOS(o_i)|$.

Definition 7. The cost function $f(o_i)$ of node o_i (i = 1, 2, …, n) in its any expanding tree is defined as:

If o_i is the root of the expanding tree, set $f(o_i) = 0$;

If o_i is not the root of the expanding tree, set

$$f(o_i) = f(\text{parent}(o_i)) + ew(\text{MinEdge}(\text{parent}(o_i), o_i)). \quad (7)$$

where parent(o_i) is the father node of node o_i, MinEdge(parent(o_i), o_i) is the hyperedge with minimum weight in the set of hyperedges that contains node parent(o_i) and node o_i, and ew(MinEdge(parent(o_i), o_i)) is the weight of hyperedge MinEdge(parent(o_i), o_i). Obviously, this is a recursive definition.

3. The Shortest Path Algorithms of Hypergraphs Based on Search Strategies

Because the shortest node path algorithms of hypergraphs is similar to the shortest hyperedge path algorithms of hypergraphs, so we only discuss the latter.

3.1. A Width-First Searching Algorithm of Hypergraphs with Unweighted Hyperedges

The width-first searching algorithm of hypergraphs with unweighted hyperedges is presented below.

Algorithm Name: a width-first searching algorithm of hypergraphs with unweighted hyperedges (Algorithm 1).

Input: a set of nodes, $OS = \{o_1, o_2, \ldots, o_n\}$; a set of hyperedges, $ES = \{e_1, e_2, \ldots, e_m\}$; a associated matrix, $A = (a_{ik})_{n*m}$, between the set of nodes and the set of hyperedges.

Output: the shortest hyperedge path between a pair of nodes and its length.

Procedure:

Step1. According to the matrix A, definition 4, and definition 6, construct two sets AES (o_i) and AOS (o_i) for every node o_i (i = 1, 2, …, n). According to the matrix A and definition 5, construct the set EOS (e_k) for every hyperedge e_k (k = 1, 2, …, m).

The set $AES(o_i)$ of node o_i and the set $EOS(e_k)$ of hyperedge e_k can be constructed easily from the matrix A. The set $AOS(o_i)$ is constructed from two sets $AES(o_i)$ and $EOS(e_k)$. The concrete constructing procedure is described as follows:

```
/* Constructing the set EOS (ek) of hyperedge ek (k = 1, 2, …, m). */
for (k = 1; k ≤ m; k++) {
    The set EOS (ek) is set as the empty set;
    for (i = 1; i ≤ n; i ++)
        if (aik == 1)                    //Node oi belongs to superedge ek.
            EOS (ek) ← EOS (ek) ∪{oi};    /* Nodes oi is inserted into the set EOS (ek). */
}//endfor
/* Constructing the set AES (oi) of node oi (i = 1, 2, …, n). */
for (i = 1; i ≤ n; i ++) {
    The set AES (oi) is set as the empty set;
    for (k = 1; k ≤ m; k++)
        if (aik == 1)                    // Node oi belongs to superedge ek.
```

```
                AES(o_i) ← AES(o_i) ∪{e_k};      /* Superedge e_k is inserted into the set AES (o_i). */
    }//endfor
    /* Constructing the set AOS (o_i) of node o_i (i = 1, 2, …, n). */
    for (i = 1; i ≤ n; i ++) {
        The set AOS (o_i) is set as the empty set;
        for (every element e_k in the set AES (o_i))          /* Superedge e_k belongs to the set AES (o_i). */
            AOS (o_i) ← AOS (o_i) ∪(EOS (e_k) – {o_i});       /* The other elements except node o_i in the set EOS
(e_k) is inserted into the set AOS (o_i). */
    }//endfor
```

Step 2. According to the set AOS (o_i) of node o_i (i = 1, 2, …, n), the width-first searching strategy is used to construct the branching tree of node o_i, whose root is node o_i. The concrete constructing procedure is described as follows:

```
int VisitedNode[1.. n];          /* "VisitedNode[i]" records if node o_i is accessed. */
    int curNode;                 /*   "curNode " records the label of the current accessing node. */
    int VisitedTimes;            /*   "VisitedTimes" records the times of accessed nodes. */
    for (i = 1; i ≤ n; i ++) {   /* Construct the branching tree of node o_i. */
    for (j = 1; j ≤ n; j ++)     /* Initialize the array VisitedNode[1.. n]. */
        VisitedNode[j] ← 0;      /* It means node o_i has not been accessed. */
        VisitedNode[i] ← 1;      /*   Construct the branching tree from node o_i. */
        curNode ← i;             /* Node o_i is the current accessing node. */
        Create an empty queue q;     /* q is the name of the queue. */
        The label i of node o_i enters into queue q;
        Tree t_i ← create_RootT(o_i);        /* Construct a branching tree t_i whose root is node o_i. */
        VisitedTimes ← 0;            /* Initialize the times of accessed nodes. */
        While (queue q is not empty) {
            The head of queue q is moved out and assigned to variable "curNode";
            if (The set AOS(o_curNode) of node o_curNode is not empty)
                if (There exists one or more unaccessed nodes in the set AOS(o_curNode))
                    for (every unaccessed node o_j in the set AOS(o_curNode)) {
                        Node o_j ← create_Node();     /* Create a new node o_j. */
                        Create a branch from node o_curNode to node o_j in t_i;
                        VisitedNode[j] ← 1;           /* Node o_j is labeled as accessed node. */
                        VisitedTimes ← VisitedTimes + 1;
                        The label j of node o_j enters into queue q;
                    }//endfor
        }//endwhile
        if (VisitedTimes < n)
            It means there exists one or more unaccessable nodes from node o_i;
    }//endfor
```

Step 3. The path from the root node o_i to another node in the branch tree t_i (i = 1, 2, …, n) is its shortest path. And the height of the accessed node is the length of its shortest path.

3.2. A minimum-Cost-First Searching Algorithm of Hypergraphs with Weighted Hyperedges

The minimum-cost-first searching algorithm of hypergraphs with unweighted hyperedges is presented below.

Algorithm Name: a minimum-cost-first searching algorithm of hypergraphs with weighted hyperedges (Algorithm 2).

Input: a set of nodes, $OS = \{o_1, o_2, \dots, o_n\}$; a set of hyperedges, $ES = \{e_1, e_2, \dots, e_m\}$; a set of weight of hyperedges, $EW = \{ew_1, ew_2, \dots, ew_m\}$; a associated matrix, $A = (a_{ik})_{n*m}$, between the set of nodes and the set of hyperedges.

Output: the shortest hyperedge path between a pair of nodes and its length.

Procedure:

Step 1. The same as the Step1 of Algorithm 1.

Step 2. If the set $AOS(o_i)$ of node o_i (i = 1, 2, …, n) is not empty, then create an array $AssMEW(o_i)[1 .. |AOS(o_i)|]$ that stores its associated nodes, hyperedges with minimum weight between node o_i and its associated nodes, and the weights of the hyperedges.

For example, suppose node o_j is an associated node of node o_i, then a structure $AssMEW(o_i)[j] = <o_j, \text{MinEdge}(o_i, o_j), ew(\text{MinEdge}(o_i, o_j))>$ can be designed. Where $\text{MinEdge}(o_i, o_j)$ is a hyperedge with minimum weight that contains two associated nodes o_i and o_j, and $ew(\text{MinEdge}(o_i, o_j))$ is the weight of the hyperedge $\text{MinEdge}(o_i, o_j)$.

An implementation procedure of this step is described as follows:

```
for (i = 1; i ≤ n; i ++)
    if (the set AOS (o_i) is not empty)
        for (every node o_j in the set AOS(o_i)) {
            The label j of node o_j is added into AssMEW (o_i)[j];
            Fetch the common associated hyperedges, denoted by ComAssEdge (AES (o_i), AES (o_j)),
from two sets AES (o_i) and AES (o_j);
            for (every hyperedge in the set ComAssEdge(AES (o_i), AES (o_j)))
                AssMEW (o_i) [j] always records the hyperedge with minimum weight and its weight;
        }//endfor
```

Step 3. According to the set $AOS(o_i)$ of node o_i (i = 1, 2, …, n), the minimum-cost-first searching strategy is used to construct the branching tree of node o_i, whose root is node o_i. When constructing the branching tree, the cost function of every node is computed according to Definition 7. The concrete constructing procedure is described as follows:

```
int curNode;                    /*   "curNode " records the label of the current accessing node. */
int CanVisitNode[1.. n];        /* "CanVisitNode[i]" records if node o_i can be accessed.   */
bool ExistExpandNode;           /* "ExistExpandNode " records if there exists one or more expandable nodes
in the branching tree. */
for (i = 1; i ≤ n; i ++) {              /* Construct the branching tree of node o_i. */
    for (j = 1; j ≤ n; j ++) {
        CanVisitNode[j] ← 1;            /* It means node o_j can be accessed. */
        g(o_j) ← +∞;            /* Initialize the preference-selecting function g(o_j). */
    } //endfor
    f(o_i) ← 0;                 /* The cost function f(o_i) of root o_i   is assigned by zero. */
    g(o_i) ← f(o_i);        /* The preference-selecting function g(o_i) of root o_i   is updated by f(o_i). */
    curNode ← i;                /* Node o_i is the current accessing node. */
    CanVisitNode[i] ← 0;            /* Node o_i cannot be accessed after expanded. */
    Tree t_i ← create_RootT(o_i) ;          /* Construct a branching tree t_i whose root is node o_i. */
    do {
        if (the set AOS(o_curNode) is empty, or node o_curNode cannot be expanded) {
            Node o_curNode is labeled as a terminal leaf node;
            CanVisitNode[curNode] ← 0;      /* It means node o_curNode cannot be expanded. */
        }
        else {
            for (every accessable node o_j in the set AOS(o_curNode)) {
                Node o_curNode is regarded as the parent node of node o_j, and the cost function f(o_j) of
node o_j is computed by equation (7);
                if (f(o_j) < g(o_j)) {
                    if (node o_j is already in the branching tree t_i)
                        Delete the old branch reaching node o_j in the branching tree t_i;
```

else
Node o_j ← create_Node(); /* Create a new node o_j. */
Insert a new branch from node $o_{curNode}$ to node o_j in the branching tree t_i;
$g(o_j) \leftarrow f(o_j)$; /* The preference-selecting function $g(o_j)$ of node o_j is updated by $f(o_j)$. */
} //endif
} //endfor
Node $o_{curNode}$ is labeled as a nonleaf node that cannot be expanded; /* Node $o_{curNode}$ has been expanded. */
CanVisitNode[curNode] ← 0; /* Node $o_{curNode}$ cannot be expanded anymore. */
} //endif
ExistExpandNode ← false;
if (There exists one or more accessable nodes in the branching tree t_i) /* There exists one or more nonleaf nodes in the branching tree t_i. */
ExistExpandNode ← true;
if (ExistExpandNode)
The label of the nonleaf node that gets the minimum value of preference-selecting function in the branching tree t_i is assigned to "curNode"; /* Choose the node with minimum value of preference-selecting function from those elements whose value is equal to one in array CanVisitNode[1.. n]. */
} **while** (ExistExpandNode);
if (VisitedTimes < n)
It means there exists one or more unaccessable nodes from node o_i;
} //endfor

Step 4. The path from the root node o_i to another node in the branch tree t_i (i = 1, 2, …, n) is its shortest path. And the value of the cost function of the accessed node is the length of its shortest path.

3.3. Some Notes of Two Algorithms

The minimum-cost-first searching algorithm of hypergraphs with unweighted hyperedges is presented below.

Some notes of Algorithm 1 are listed below:

1) In Step 1, the ordered (in ascending order) adjacency list [3], [4], which stores the set of associated hyperedges of node, the set of nodes of hyperedge, and the set of associated nodes of node, is similar to the adjacency list structure of graph.
2) In Step 2, the rule, "from top to bottom, from left to right", is used to create the branch tree t_i. Here, queue structure can effectively implement this creating order of nodes based on this rule, which reflects the wide-first searching strategy.
3) Time and space complexity analysis of Algorithm 1:

Step 1 needs Time = $O(n \cdot m + \sum_{i=1}^{n} \sum_{e_k \in AES(O_i)} (|EOS(e_k)| \cdot |AOS(O_i)|))$ and Space = $O(n \cdot m + \sum_{i=1}^{n} |AES(o_i)| + \sum_{i=1}^{n} |AOS(o_i)| + \sum_{k=1}^{m} |EOS(e_k)|)$.

Step 2 needs Time = $O(n^2)$ and Space = $O(n^2 + \sum_{i=1}^{n} |AOS(o_i)|)$.

Step 3 needs Time = $O(n^2)$ and Space = $O(n^2)$.

4) This is a kind of searching algorithm based on the width-first strategy. It searches the shortest path from a hypergraph with unweighted hyperedges by constructing the set of associated hyperedges of every node, the set of nodes of every hyperedge, and the set of associated nodes of every node.

Some notes of Algorithm 2 are listed below:

1) If Step1 uses the improving technique used in Step2, then the procedure of Step1, Constructing the set $AOS(o_i)$ of node o_i (i = 1, 2, …, n), can be replaced by the following procedure.

```
/* Create AssMEW(o_i) that is implemented by the array AssMEW(o_i)[1 .. |AOS(o_i)|]. */
    for (i = 1; i ≤ n; i ++) {
    AssMEW(o_i) is set as the empty set;
    for (every element e_k in the set AES(o_i))        /* Superedge e_k belongs to the set AES(o_i) of node o_i. */
        for (every node o_j in the set EOS(e_k))
            if (node o_j and node o_i are different)
                if (node o_j is not in AssMEW(o_i))
                    /* Node o_j, hyperedge e_k, and the weight of hyperedge e_k are inserted into AssMEW(o_i). */
                    AssMEW(o_i)[j] ← <o_j, e_k, ew(e_k)>;
                else {                    /* Node o_j    is in AssMEW(o_i). */
                    if (the weight of hyperedge e_k is less than ew(MinEdge(o_i, o_j)) of node o_j in AssMEW(o_i))
                        /* Hyperedge e_k and the weight of hyperedge e_k are used to update the latter two items of node o_j in AssMEW(o_i). */
                        AssMEW(o_i)[j] ← <o_j, e_k, ew(e_k)>;
                }
    } //endfor
```

2) In Step 3, "nonleaf nodes" are those nodes that can be expanded in the current branching tree. If the value of f function, which is computed by using equation (7), of every associated node of node $o_{curNode}$ is larger than or equal to the value of g function of the associated node, then "node $o_{curNode}$ cannot be expanded".

3) Time and space complexity analysis of Algorithm 2:

In Step 1, it needs the same time and space with Step 1 of Algorithm 1.

Step 2 needs Time = $O(\sum_{i=1}^{n}\left(|AOS(o_i)| + \sum_{o_j \in AOS(o_i)}(|AES(o_i)| + |AES(o_j)|)\right))$ and Space = $O(n + m + \sum_{i=1}^{n}|AES(o_i)| + \sum_{i=1}^{n}|AOS(o_i)|)$.

Step 3 needs Time = $O(n^2)$ and Space = $O(n^2)$.

Step 4 needs Time = $\theta(n^2) \sim O(n \cdot \sum_{i=1}^{n}|AOS(o_i)|)$ and Space = $O(n^2 + \sum_{i=1}^{n}|AOS(o_i)|)$.

4) This is a kind of searching algorithm based on the minimum-cost-first strategy. It searches the shortest path from a hypergraph with weighted hyperedges by constructing the set of associated hyperedges of every node, the set of nodes of every hyperedge, and the set of associated nodes with minimum hyperedge and its weight of every node.

Table 1. The Time Complexity of Three Algorithms

	Algorithms from [5]	Algorithms from [6]	Algorithms from This paper
Unweighted hyperedges	$O(n \cdot m \cdot (\max_{i=1,2,\ldots,m} \lvert e_i \rvert)^2)$	$O(n^3)$	$O(n \cdot m + \sum_{i=1}^{n}\sum_{e_k \in AES(O_i)}(\lvert EOS(e_k) \rvert \cdot \lvert AOS(O_i) \rvert))$
Weighted hyperedges	$O(n \cdot m \cdot (\max_{i=1,2,\ldots,m} \lvert e_i \rvert)^2 + n^3)$	$O(n^3)$	$O(n \cdot m + \sum_{i=1}^{n}\sum_{e_k \in AES(O_i)}(\lvert EOS(e_k) \rvert \cdot \lvert AOS(O_i) \rvert) + n \cdot \sum_{i=1}^{n}\lvert AOS(o_i) \rvert)$

3.4. Comparison with Two Other Algorithms

Table 1 gives the comparison results the algorithms from this paper with other algorithms [5], [6] in time complexity. In some cases, our algorithms can be more efficient by constructing the set of associated hyperedges of node, the set of nodes of hyperedge, and the set of associated nodes of node.

3.5. The Proof of Correctness of Two Algorithms

Theorem 1. In a hypergraph with unweighted hyperedges, when constructing a branching tree for every node using a kind of searching algorithm based on the width-first strategy, the path from the root to another node in the branching tree is its one shortest path, the depth of every node in the branching tree is the length of the shortest path from the root to the node.

Proof: The conclusion is obvious because of the expanding strategy of nodes in branching trees. In the constructing procedure of branching trees, after expanded a node, its successor node is chosen to expand according to the rule, "from top to bottom, from left to right". The branching trees contain the paths from the root to other nodes.

Theorem 2. In a hypergraph with weighted hyperedges, when constructing a branching tree for every node using a kind of searching algorithm based on the minimum-cost-first strategy, the path from the root to another node in the branching tree is its one shortest path, the value of preference-selecting function of every node in the branching tree is the length of the shortest path from the root to the node.

Proof: The conclusion is also obvious because of the expanding strategy of nodes in branching trees. In each node expansion of the branching trees, the nonleaf node with the minimum value of preference-selecting function is chosen to expand. The branching trees also contain the paths from the root to other nodes.

4. Two Examples to Explain the Algorithms

4.1. Example 1

Suppose there is a hypergraph that is described by a set of nodes $\{o_1, o_2, o_3, o_4, o_5\}$ and a set of hyperedges $\{e_1 = \{o_1, o_2, o_3\}, e_2 = \{o_2, o_3\}, e_3 = \{o_4, o_5\}\}$.

The results of example 1 after executed Step 1 of Algorithm 1 are given in Table 2 and Table 3.

Table 2. The Sets *AES* and *AOS* of Nodes in Algorithm 1

o_i	o_1	o_2	o_3	o_4	o_5
$AES(o_i)$	$\{e_1\}$	$\{e_1, e_2\}$	$\{e_1, e_2\}$	$\{e_3\}$	$\{e_3\}$
$AOS(o_i)$	$\{o_2, o_3\}$	$\{o_1, o_3\}$	$\{o_1, o_2\}$	$\{o_5\}$	$\{o_4\}$

Table 3. The Set *EOS* of Hyperedges in Algorithm 1

e_k	e_1	e_2	e_3
$EOS(e_k)$	$\{o_1, o_2, o_3\}$	$\{o_2, o_3\}$	$\{o_4, o_5\}$

The results of example 1 after executed Step 2 of Algorithm 1 are given in Fig. 1.

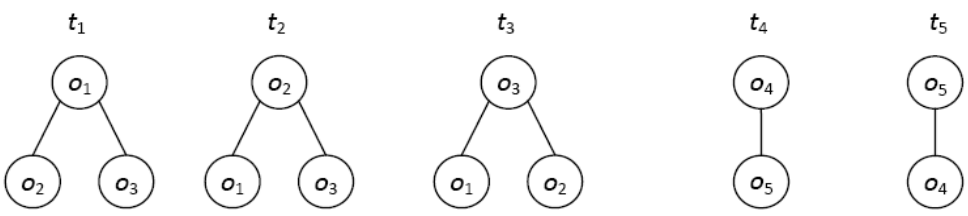

Fig. 1. The branching trees of five nodes in Algorithm 1.

The results of example 1 after executed Step 1 of Algorithm 2 are the same as Algorithm 1 except that the set *AOS* (o_i) in Algorithm 2 contains more information, which is listed in Table 4.

Table 4. The Set AOS with Minimum Hyperedge of Nodes in Algorithm 2

o_i	$AOS(o_i)$
o_1	{<o_2, e_1, 2>, <o_3, e_1, 2>}
o_2	{<o_1, e_1, 2>, <o_3, e_2, 1>}
o_3	{<o_1, e_1, 2>, <o_2, e_2, 1>}
o_4	{<o_5, e_3, 1.5>}
o_5	{<o_4, e_3, 1.5>}

In this simple example, the branching trees of five nodes after executed Step 2 of Algorithm 2 are the same as Algorithm 1, although their searching strategies are different.

4.2. Example 2

Fig. 2 [7] is a hypergraph that is described by a set of nodes {v_1, v_2, v_3, v_4, v_5, v_6, v_7} and a set of hyperedges {e_1 = {v_1, v_2, v_3}, e_2 = {v_2, v_3}, e_3 = {v_3, v_5, v_6}, e_4 = {v_4}. In Algorithm 1 the weights of hyperedges are set as 1. And in Algorithm 2 the weights of hyperedges are set as {3, 2, 3, 1}.

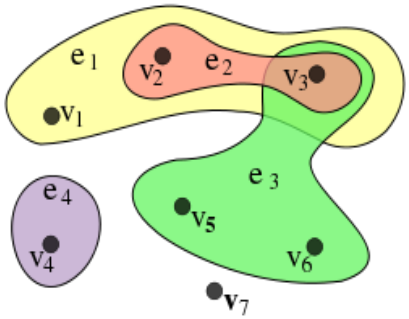

Fig. 2. A hypergraph from Wikipedia [7].

The results of example 2 after executed Step 1 of Algorithm 1 are given in Table 5 and Table 6.

Table 5. The Sets *AES* and *AOS* of Nodes in Algorithm 1

v_i	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$AES(v_i)$	{e_1}	{e_1, e_2}	{e_1, e_2, e_3}	{e_4}	{e_3}	{e_3}	Φ
$AOS(v_i)$	{v_2, v_3}	{v_1, v_3}	{ v_1, v_2, v_5, v_6}	Φ	{v_3, v_6}	{v_3, v_5}	Φ

Table 6. The Set *EOS* of Hyperedges in Algorithm 1

e_k	e_1	e_2	e_3	e_4
$EOS(e_k)$	{v_1, v_2, v_3}	{v_2, v_3}	{v_3, v_5, v_6}	{v_4}

The results of example 2 after executed Step 2 of Algorithm 1 are given in Fig. 3.

The results of example 2 after executed Step1 of Algorithm 2 are the same as Algorithm 1 except that the set *AOS* (o_i) in Algorithm 2 contains more information, which is listed in Table 7.

In this simple example, the branching trees of seven nodes after executed Step2 of Algorithm 2 are the same as Algorithm 1, although their searching strategies are different.

If the set of hyperedges in this hypergraph is revised as {e_1 = {v_1, v_2, v_3}, e_2 = {v_2, v_3}, e_3 = {v_3, v_5, v_6}, e_4 = {v_1, v_3}}, and the set of weights of hyperedges in Algorithm 2 is revised as {3, 2, 3, 0.5}, then the branching tree t_2 of node v_2 after Step 2 in Algorithm 2 is different from Algorithm 1.

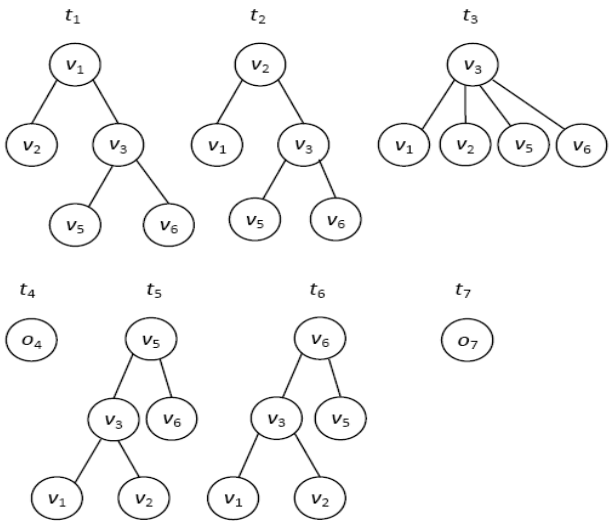

Fig. 3. The branching trees of seven nodes in Algorithm 1.

Table 7. The Set *AOS* with Minimum Hyperedge of Nodes in Algorithm 2

v_i	$AOS(v_i)$
v_1	{<v_2, e_1, 3>, <v_3, e_1, 3>}
v_2	{<v_1, e_1, 3>, <v_3, e_2, 2>}
v_3	{<v_1, e_1, 3>, <v_2, e_2, 2>, <v_5, e_3, 3>, <v_6, e_3, 3>}
v_4	Φ
v_5	{<v_3, e_3, 3>, <v_6, e_3, 3>}
v_6	{<v_3, e_3, 3>, <v_5, e_3, 3>}
v_7	Φ

5. Simulation Experiments

5.1. Experimental Design

Our experiments are finished in a personal computer (Intel(R) Pentium(R) Dual CPU T4500 2.3GHz, 2G Memory). Experimental programs are developed using Visual C++6.0 under Windows XP.

To verify the correctness and the validity of Algorithm 1 and Algorithm 2, there will be some experiments of a kind of artificial hypergraph in the next subsection.

The artificial hypergraph, which was used in [8], has n nodes and m hyperedges. The matrix $A = (a_{ik})_{n*m}$ is designed by producing $n*m$ random real numbers in the region [0, 1] such that 1 is assigned to the elment a_{ik} if the $i*k$-th random real number is larger than threshold parameter Δ, otherwise 0 is assigned to the elment a_{ik}. The set $EW = \{ew_1, ew_2, \ldots, ew_m\}$ is designed by producing m random positive real numbers.

In subsection 5.2, Algorithm 2 will be compared with Algorithm 1 in time cost (measured by second) and property of branching trees by using several hypergraphs with different n and m.

Performance of algorithms is measured by time cost (label: *ST*). A result of algorithms is measured by the average percentage between the number of nodes in branching trees and the number of all nodes (label: Per).

$$Per = \frac{1}{n}\sum_{i=1}^{n}\frac{|tree(O_i)|}{n} \tag{8}$$

where $|tree\ (O_i)|$ (i = 1, 2, ..., n) is the number of nodes in the branching tree t_i whose root is node O_i. The average percentage can measure the connectivity of a hypergraph. The larger the index is, the better the connectivity of the hypergraph has.

5.2. Experimental Results

Table 8 and Table 9 list some comparative experimental results of Algorithm 1 and Algorithm 2 by processing the artificial hypergraphs with several pairs of parameters (n, m). Label *ST* means spend time.

Table 8. Comparative Experimental Results of Two Algorithms (Threshold Parameter Δ = 0.95 in the Artificial Hypergraphs)

Pairs of parameters (n, m)		Algorithm 1		Algorithm 2	
n	***m***	***ST***	***Per*(%)**	***ST***	***Per*(%)**
100	5	0	**1.98**	0	**1.98**
100	10	0	**4.46**	0	**4.46**
100	20	0	**32.94**	0	**32.94**
100	30	0	**65.8**	0	**65.8**
100	40	0	**74.1**	0	**74.1**
100	50	0	**82.92**	0	**82.92**
100	60	0	**90.3**	0	**90.3**
100	70	0	**96.07**	0	**96.07**
100	80	0	**99.01**	0	**99.01**
100	90	0	**100**	0	**100**
200	5	0	**1.73**	0	**1.73**
200	10	0	**14.75**	0	**14.75**
200	20	0	**39.25**	0	**39.25**
200	30	0	**64.1**	0	**64.1**
200	40	0	**73.18**	0	**73.18**
200	50	0	**84.68**	0	**84.68**
200	60	0	**90.28**	0	**90.28**
200	70	0	**95.08**	0	**95.08**
200	80	0	**98.02**	0	**98.02**
200	90	0	**100**	1	**100**

Table 9. Comparative Experimental Results of Two Algorithms (Threshold Parameter Δ = 0.99 in the Artificial Hypergraphs)

Pairs of parameters (n, m)		Algorithm 1		Algorithm 2	
n	***m***	***ST***	***Per* (%)**	***ST***	***Per* (%)**
100	5	0	**1.06**	0	**1.06**
100	10	0	**1.12**	0	**1.12**
100	20	0	**1.28**	0	**1.28**
100	30	0	**1.58**	0	**1.58**
100	40	0	**2.4**	0	**2.4**
100	50	0	**2.13**	0	**2.13**
100	60	0	**2.73**	0	**2.73**
100	70	0	**2.29**	0	**2.29**
100	80	0	**7.74**	0	**7.74**
100	90	0	**9.48**	0	**9.48**
100	800	0	**98.02**	0	**98.02**
100	1000	0	**97.03**	0	**97.03**
100	2000	0	**100**	0	**100**
200	5	0	**0.59**	0	**0.59**
200	10	0	**0.64**	0	**0.64**
200	20	0	**1.67**	0	**1.67**
200	30	0	**1.68**	0	**1.68**
200	40	0	**3.33**	0	**3.33**
200	50	0	**2.06**	0	**2.06**
200	60	0	**10.33**	0	**10.33**
200	70	0	**10.34**	0	**10.34**
200	80	0	**9.96**	0	**9.96**
200	90	0	**25.97**	0	**25.97**
200	500	0	**94.6**	0	**94.6**
200	800	0	**100**	0	**100**
200	810	0	**99.50**	0	**99.50**

5.3. Analysis and Conclusions of Experimental Results

Because parameters n and m in the experiments cannot be set too large due to memory constraints, we see that the time cost of Algorithm 1 and Algorithm 2 almost have no difference. From Table 8 and Table 9, we observe that the average percentage (computed by equation (8)) of Algorithm 1 is the same as Algorithm 2, which is consistent with the actual instance.

Actually, the value of each $|tree(O_i)|$ (i = 1, 2, …, n) of any hypergraph with unweighted hyperedges is equal to the corresponding hypergraph with weighted hyperedges. So we can obtain the same average percentage.

6. Conclusions

In the environment of complex Internet and Internet of Things, there is a wide variety of diverse relations among massive objects. The shortest path of a hypergraph model can characterize the associated relationship between different nodes on some degree. To search the shortest path from hypergraphs with unweighted hyperedges, this paper presents a kind of searching algorithm based on the width-first strategy. To search the shortest path from hypergraphs with weighted hyperedges, it presents a kind of searching algorithm based on the minimum-cost-first strategy. In order to enhance the reliability of the two algorithms, their correctness is proved in theory.

The next work is to do more experimental comparison and to study the relationship between the parameters and the experimental results.

Acknowledgment

This work was supported by a project (grant No. cstc2014jcyjA40035) from Chongqing Cutting-edge and Applied Foundation Research Program of China and a project (grant No. 12RC01) from Chongqing Three Gorges University of China.

References

[1] Wang, Z. P., & Wang, Z. T. (2008). *Supernetwork Theory and Its Application*. Beijing: The Science Press.

[2] Voloshin, V. I. (2009). *Introduction to Graph and Hypergraph Theory*. New York: Nova Science Publishers.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein. C. (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts: The MIT Press.

[4] Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures* (3th ed.). Massachusetts: Morgan Kaufmann Publishers.

[5] Li, C. M. (1994). The algorithms of the hypergraph's shortest path. *Journal of Inner Mongolia Polytechnic University*, *13(1)*, 27–32.

[6] Qu, G., & Ji, C. (2005). Shortest path algorithm for hypergraphs. *Journal of Chongqing University (Natural Science Edition)*, *28(11)*, 106–109.

[7] Introduction of Hypergraph. Retrieved from Wikipedia website: http://en.wikipedia.org/wiki/Hypergraph

[8] Chen, X. Q. (2014). Computational model of association activity measure and its algorithmic implementation. *Journal of Software*, *9(5)*, 1135–1140.

Xinquan Chen received the PhD degree from South China University of Technology in China. He is now an associate professor of Chongqing Three Gorges University in China and a postdoctoral researcher of University of Electronic Science & Technologhy of China. His research interests include data mining, machine learning, clustering algorithms, and optimization methods.