An Algorithm for Nodes-constrained Shortest Component Path on Software Architecture

Lijun Lun, Xin Chi, Hui Xu

***Abstract*—The vast majority of researches about the shortest component path algorithm in software architecture, nowadays, focus just on the case from the beginning component to the stopping component. When the shortest component path is constrained to the specified number of components, the current technologies will no longer be applied. Based on the research on the present the shortest path algorithm, we propose an algorithm to solve the nodes-constrained shortest component path (π_{NCSCP}) problem in software architecture. The proposed algorithm based on inverse adjacency list of the Component Interaction Graph (CIG) is developed for finding π_{NCSCP} of C2-style architecture. The time complexity of the proposed algorithm is O((m-2)w) (m is the number of nodes-constrained, w is the total number of edges in CIG). Since the inverse adjacency list is used to the graphic storage structure, this algorithm is much simpler and easy to be fulfilled, and reduces the time complexity as well.**

***Index Terms*—software architecture, C2-style, shortest component path, nodes-constrained, inverse adjacency list.**

I. Introduction

THE shortest path problem is a basis and important problem in software architecture [1], which is relatively simple. We often encounter the shortest path problem in software architecture design. Many software architectures can be used as the shortest path, or the shortest path algorithm is used as a sub-problem. So, in software architecture design, it has a very practical significance for solving the shortest path algorithm.

The shortest path problems are divided into four classical algorithms: (1) Algorithms based on graph theory, such as Dijkstra [2], Floyd [3] and its improved algorithms etc., (2) Mathematical planning algorithms based on optimization theory [4], (3) Search algorithms based on traditional artificial intelligence, such as blind search, heuristic A* [5] and its improved algorithms, (4) Search algorithms based on modern computational intelligence, such as artificial neural network, genetic algorithm, immune algorithm and ant colony algorithm [6], [7], [8] etc.

Software architecture has many new characteristics, such as component, connector and so on [9]. All of these characteristics have an impact on interactions among component, connector [10] and finding the shortest component path. In many cases, there are some additional constraints on the shortest path selection, which is often required to find the shortest component path and the shortest component path to meet the additional constraints.

Many previous studies have investigated the shortest path problem. Demetrescu and Italiano presented a new algorithm that achieves nearly-quadratic update bounds for fully dynamic all-pairs shortest paths on graphs with non-negative real edge weights [11]. The techniques are described not only asymptotically efficient, but can yield very fast implementations in many practical scenarios. Chuang and Kung proposed a heuristic procedure to find the fuzzy shortest path length among all possible paths in a network [12]. It is based on the idea that a crisp number is a minimum number if and only if any other number is larger than or equal to it. It owns a firm theoretic base in fuzzy sets theory and can be implemented effectively. Meanwhile, they proposed a method to measure the similarity degree between the fuzzy shortest path length and each fuzzy path lengths. Idwan and Etaiwi studied the improvement of Dijkstra algorithm by using heuristic algorithm [13], and applied it to the shortest path for large graph. Li et al. studied the optimization of the Dijkstra algorithm using the restricted direction method and the binary heap technology, and applied it to the WebGIS [14]. Xu and Ke divided the designated-points shortest paths problem into three categories [15], such as isolated designated-points shortest paths, grouped designated-points shortest paths, and grouped and order-preserving designated-points shortest paths, and established their mathematics models. They presented improved genetic algorithm for the grouped and order-preserving designated-points shortest paths problem. And proposed order-preserving operation to ensure that some particular points are connected based on the determined order, improved the global search capability and convergence. Feng et al. focused on the solving of the shortest paths in the conditions of complicated constraints for network analysis and application [16], geometric algebra is used to develop the network analysis algorithms. A network model and bilateral search algorithms are built based on the multivector representation and multidimensional operators of geometric algebra. The algorithm is proposed to find the shortest paths passing through specified necessary nodes and the least segments.

It should be noted that the traditional ideas on shortest path have difficulty in identifying parts from the framework of software architectures. In particular, there are few researches on the shortest path at the architectural level. Gao et al. proposed an adequate test model and test coverage criteria for component validation [17]. A set of component API-based test coverage criteria is defined based on the test models, and a dynamic test coverage analysis approach is provided. Where minimum-set path coverage from E_i to E_j in G for component C is, if and only if there exists a path set from E_i

Manuscript received November 6, 2015; revised January 14, 2017. This work was supported in part by the Natural Science Foundation of Heilongjiang Province of China under Grant F201036 and by the Scientific Research Foundation of Heilongjiang Provincial Education Department of China under Grant 12541250.

Lijun Lun is with the College of Computer Science and Information Engineering, Harbin Normal University, Harbin, 150080, China, e-mail: lunlijun@yeah.net.

Xin Chi is with the Harbin Normal University, China, e-mail: xinc1990@163.com.

Hui Xu is with the Heilongjiang University of Chinese Medicine,China, e-mail: xuhui8413@163.com.

to E_j which covers all nodes and links reachable from E_i and to E_j and have been traversed by test scripts in component C's test script set. Lun et al. presented an approach of the shortest component path for software architecture [18]. The technique made full use of their advantages and used the improved A* algorithm to do a global search for the beginning of the stage. It improves greatly the efficiency of the convergence of the C2-style architecture, and decreases greatly the computation time of the shortest component path.

This paper focuses on the nodes-constrained shortest component path for C2-style architecture. We propose an algorithm based on the inverse adjacency list of the C2-style architecture model to search the nodes-constrained shortest component path in software architecture. The characteristic of this algorithm is that it is very simple and very easy to be described, fulfilled and understood.

The paper is organized as follows. In Section II, some preliminaries including C2-style architecture model, shortest component path, and nodes-constrained shortest component path are introduced and briefly are discussed. The algorithm based on the inverse adjacency list and the particle encoding mechanism to solve the nodes-constrained shortest component path problem in the C2-style architecture is presented in Section III. And the results from computer simulation experiments are discussed. In Section IV, the conclusion is given.

II. C2-Style Architecture Model

This section first introduces the related concepts of the C2-style architecture, and then gives the definition of the shortest component path and the nodes-constrained shortest component path according to the shortest component path.

A. C2-Style Architecture Representation

We have selected the C2-style architecture as a vehicle for exploring our ideas because it provides a number of useful rules for high-level system composition, demonstrated in numerous applications across several domains [19]; at the same time, the rules of the C2-style are broad enough to render it widely applicable [20].

The C2-style architecture [20] consists of components, connectors, and their constraints. All components and connectors have two interfaces, "top" and "bottom". The top (bottom) of a component can only be attached to the bottom (top) of one connector. It is not possible for components to be attached directly to each other. Each connector always has to act as intermediaries between them. Furthermore, a component cannot be attached to itself. However, connector can be attached together. In this case, each connector considers the other as a component with regard to the publication and forwarding of events. Component communicates by exchanging two types of events: service requests to top of the component and notifications of completed services to bottom of the component.

We define our intermediate representation Component Interaction Graph (CIG) model [21] and discuss how a C2-style architecture can be represented using our notation. CIG is used to depict the interaction relationships between interface of component and interface of connector.

Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph, where V = Comp $\cup$ Conn is the set of nodes, Comp is a finite set of components, each component $Comp_i \in$ Comp has four interfaces, they are top output interface $Comp_i.I_{pt\_o}$, top input interface $Comp_i.I_{pt\_i}$, bottom output interface $Comp_i.I_{pb\_o}$, and bottom input interface $Comp_i.I_{pb\_i}$. Conn is a finite set of connectors, each connector $Conn_j \in$ Conn has four interfaces too, they are top output interface $Conn_j.I_{nt\_o}$, top input interface $Conn_j.I_{nt\_i}$, bottom output interface $Conn_i.I_{nb\_o}$, and bottom input interface $Conn_i.I_{nb\_i}$. E = $e_{Comp-Conn} \cup e_{Conn-Comp} \cup e_{Conn-Conn}$ is a finite set of edges, where $e_{Comp-Conn} = \{e \mid e \in (Comp_i.I_{pt\_o}, Conn_j.I_{nb\_i}) \vee (Comp_i.I_{pb\_o}, Conn_j.I_{nt\_i})\}$ represents the set of edges from top (bottom) output interface of component $Comp_i$ to the bottom (top) input interface of connector $Conn_j$. $e_{Conn-Comp} = \{e \mid e \in (Conn_i.I_{nt\_o}, Comp_j.I_{pb\_i}) \vee (Conn_i.I_{nb\_o}, Comp_j.I_{pt\_i})\}$ represents the set of edges from the top (bottom) output interface of connector $Conn_i$ to the bottom (top) input interface of component $Comp_j$. $e_{Conn-Conn} = \{e \mid e \in (Conn_i.I_{nt\_o}, Conn_j.I_{nb\_i}) \vee (Conn_i.I_{nb\_o}, Conn_j.I_{nt\_i})\}$ represents the set of edges from the top (bottom) output interface of connector $Conn_i$ to the bottom (top) input interface of connector $Conn_j$. $V_{start} \subseteq$ Comp is the set of initial component nodes, these components transmit messages only. That is V_{start} = $\{Comp_i \mid Comp_i.I_{pb\_i} = \emptyset \wedge Comp_i.I_{pb\_o} = \emptyset, Comp_i \in$ Comp$\}$. $V_{end} \subseteq$ Comp is the set of terminal component nodes, these components receive messages only. That is V_{end} = $\{Comp_i \mid Comp_i.I_{pt\_o} = \emptyset \wedge Comp_i.I_{pt\_i} = \emptyset, Comp_i \in$ Comp$\}$.

In C2-style architecture, a component (connector) can interact with the other component (connector) in several ways, i.e., from component to connector, from connector to component, and from connector to connector. The CIG for C2-style architecture should be able to represent these interactions between components and connectors.

In order to construct a representation for the CIG, we carry out static analysis of the C2-style specification. First, we identify all components and connectors and represent the nodes. Then we identify all interaction relationships between components and connectors and represent the edges. If there exists a information flow from component $Comp_i$ to connector $Conn_j$, in such a case, an edge $e \in e_{Comp-Conn}$ is added to connect from the top (bottom) output interface of $Comp_i$ to the bottom (top) input interface of $Conn_j$ of CIG. If there exists a information flow from connector $Conn_i$ to component $Comp_j$, in such a case, an edge $e \in e_{Conn-Comp}$ is added to connect from the top (bottom) output interface of $Conn_i$ to the bottom (top) input interface of $Comp_j$ of CIG. If there exists a information flow from connector $Conn_i$ to connector $Conn_j$, in such a case, an edge $e \in e_{Conn-Conn}$ is added to connect from the top (bottom) output interface of $Conn_i$ to the bottom (top) input interface of $Conn_j$ of CIG.

In order to illustrate our approach in a better way, we used an example KLAX video game application [19]. For this application C2-style architecture has been used. KLAX system includes 16 components and 6 connectors, which is depicted in Fig. 1. Where the rectangle node represents component, such as GraphicsBinding and TileArtist etc. The

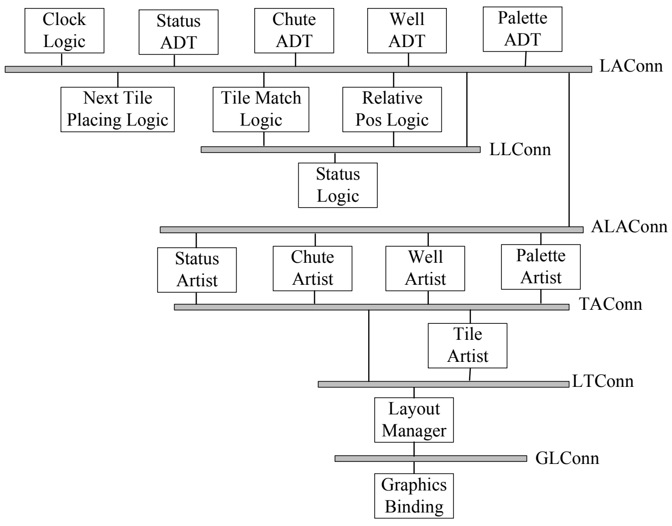

Fig. 1. KLAX Architecture in the C2-Style

long rectangle with shadow node represents connector, such as LAConn and TAConn etc. The edge between component and connector, and between connectors represents that there exists messages transmission between component and connector, such as the edge between GraphicsBinding and GLConn represents that there exists messages transmission between GraphicsBinding and GLConn, and the edge between LTConn and TAConn represents that there exists messages transmission between LTConn and TAConn.

According to the construction method of CIG, Fig. 2 shows the corresponding CIG for the example KLAX system of Fig. 1 according to C2-style architecture specification [20]. In order to simplify the representation, the name of the component and the connector are abbreviated. Where nodes represent the interface of the component and the connector, and component interface with a hollow circle, connector interface with a solid circle represents. $GB.I_{pt\_o}$, $SL.I_{pt\_o}$, and $NTPL.I_{pt\_o}$ are initial nodes. $CL.I_{pb\_i}$, $PADT.I_{pb\_i}$ and so on are terminal nodes.

B. Shortest Component Path

Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph, C_i, C_{i+1}, ..., C_k $\in$ V = Comp $\cup$ Conn. A path is a finite sequence of nodes $\pi_P(C_i, C_k)$ = $C_i \rightarrow C_{i+1} \rightarrow \ldots \rightarrow C_k$ in V such that for all i $\leq$ j $\leq$ k - 1, (C_j, C_{j+1}) $\in$ $e_{Conn-Comp} \vee e_{Comp-Conn} \vee e_{Conn-Conn}$. The length of $\pi_P(C_i, C_k)$ is the number of edges from C_i to C_k, called $\omega(\pi_P(C_i, C_k))$ for short.

In order to simplify the path representation, we ignore the interfaces of the components and connectors in the path, so that the same component and the same connector appear only once in the path. For example in Fig. 2, LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT is a π_P from component LayoutManager to component WellADT, and its length is 7. And StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutMnager $\rightarrow$ GLConn is a π_P from component StatusArtist to connector GLConn, and its length is 4.

Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph. If there exists a path $\pi_P(C_i, C_k)$ in CIG, and C_i $\in$ Comp $\wedge$ C_k $\in$ Comp, then the path $\pi_P(C_i, C_k)$ is a component path, called $\pi_{CP}(C_i, C_k)$ for short. The length of $\pi_{CP}(C_i, C_k)$ is the number of edges from C_i to C_k, called $\omega(\pi_{CP}(C_i, C_k))$ for short.

π_{CP} describes the messages transfer between components in C2-style architecture. In fact, a π_{CP} is just a series of pairs of components and its response component and connector sequences. It starts from a message that activates a corresponding component to execute, and ends on a component that does not issue any messages from its own.

Note, the π_{CP} has two forms according to the type of edges, one is all of edges from the beginning of top interface of component and connector to the stopping of bottom interface of component and connector, the other is all of edges from the beginning of bottom interface of component and connector to the stopping of top interface of component and connector.

For example in Fig. 2, there are eight component paths from LayoutManager to WellADT are given as follows, where, the length of the first four component paths is 6, the length of the following four component paths is 7.

- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ WellArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ PaletteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ WellArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ PaletteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT

Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph and there exists a component path $\pi_{CP}(C_i, C_k)$ in CIG. The shortest component path from C_i to C_k is called $\pi_{SCP}(C_i, C_k)$ for short, the length of $\pi_{SCP}(C_i, C_k)$ is called $\omega(\pi_{SCP}(C_i, C_k))$ for short and is defined as:

$$\omega(\pi_{SCP}(C_i, C_k)) = \min_{\pi_{CP}(C_i, C_k)} \omega(\pi_{CP}(C_i, C_k)) \quad (1)$$

where $\min\limits_{\pi_{CP}(C_i, C_k)} \omega(\pi_{CP}(C_i, C_k))$ represents the minimum length of component path from C_i to C_k.

For example in Fig. 2, there are four shortest component paths $\pi_{SCP}(LayoutManager, WellADT)$ given as follows, where, the length of each shortest component path is 6.

- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ WellArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT
- LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ PaletteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ WellADT

It is clear that a shortest component path between components contains other shortest component paths within it.

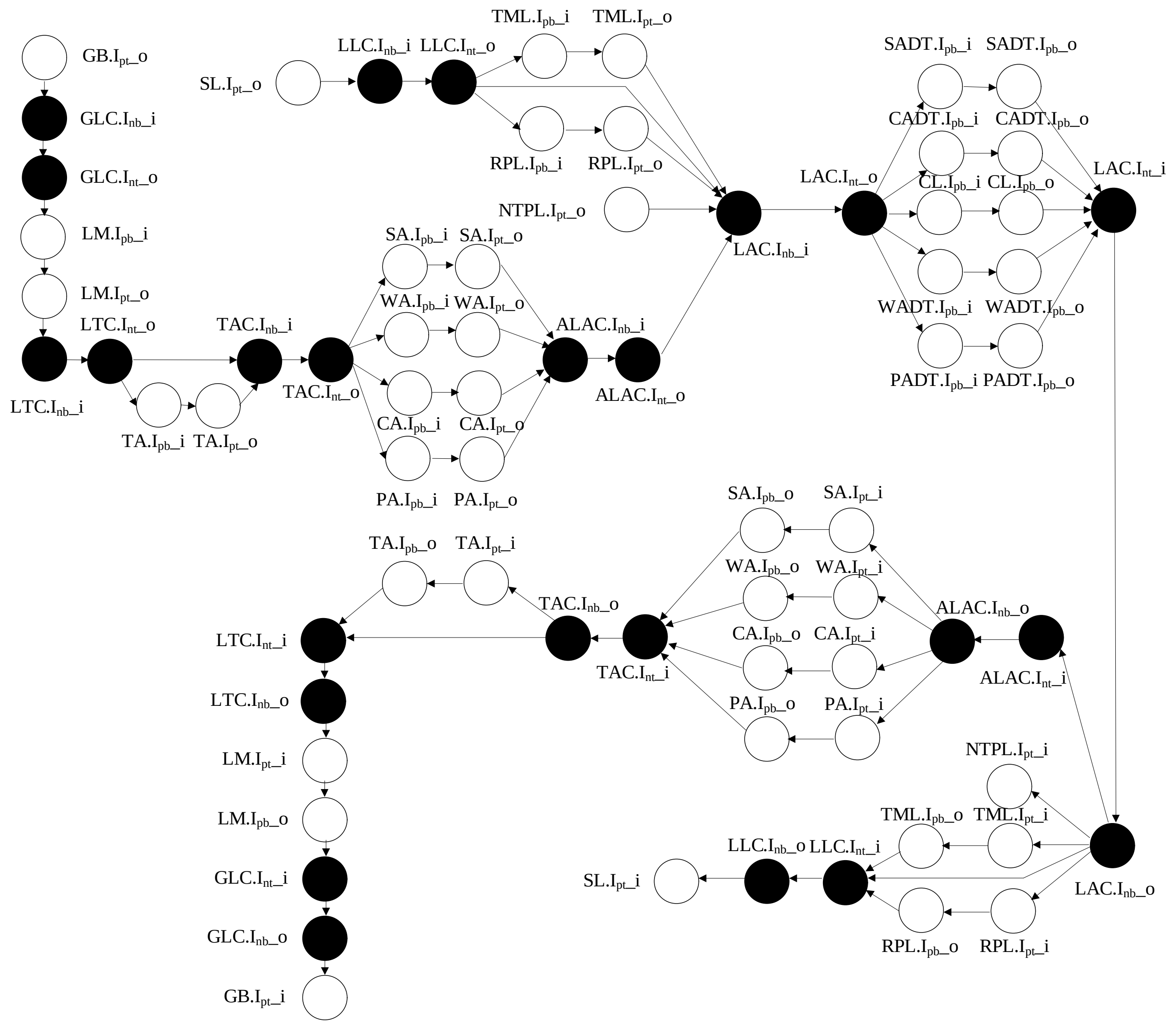

Fig. 2. CIG of KLAX System

Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph and there exists a shortest component path $\pi_{SCP}(C_i, C_k)$ in CIG, a set of nodes $V_{\pi_{SCP}(C_i,C_k)}$ of $\pi_{SCP}(C_i, C_k)$ is $V_{\pi_{SCP}(C_i,C_k)} = \{V_{p_j} \mid V_{p_j} \in$ Comp $\cup$ Conn $\wedge\ V_{p_j} \neq C_i \wedge V_{p_j} \neq C_k$, j = 1, 2, ..., l, $1 < \omega(\pi_{SCP}(C_i, C_k))\}$. If there exists constrained:

$$|V_{\pi_{SCP}(C_i,C_k)}| \leq k \ (k \in N^+) \tag{2}$$

then corresponding shortest component path is called nodes-constrained shortest component path, called $\pi_{NCSCP}(C_i, C_k)$ for short, the length of $\pi_{NCSCP}(C_i, C_k)$ is called $\omega(\pi_{NCSCP}(C_i, C_k))$ for short, where N^+ is a positive integer set.

For example in Fig. 2, there exists four shortest component paths $\pi_{SCP}(LayoutManager, WellADT)$, but if the request that the total number of nodes required to pass through this shortest component path is no more than 6, the shortest component paths $\pi_{NCSCP}(LayoutManager, WellADT)$ are given as follows, where, the number of nodes of the first four shortest component paths is 7, their length is 6, the number of nodes of the following four shortest component paths is 8, their length is 7.

- LayoutManager → LTConn → TAConn → StatusArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TAConn → ChuteArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TAConn → WellArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TAConn → PaletteArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TileArtist → TAConn → ChuteArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TileArtist → TAConn → WellArtist → ALAConn → LAConn → WellADT
- LayoutManager → LTConn → TileArtist → TAConn → PaletteArtist → ALAConn → LAConn → WellADT

We will present the algorithm for solving nodes-constrained shortest component path in the next section.

III. Approach for Finding π_{NCSCP}

It is found that, for any one node, the node is not directly connected to its neighboring nodes, that is, in the CIG of software architecture, for any one node C_i, which has path with a small number of nodes, that is only a few nodes C_j to meet the distance between two nodes $\omega(\pi_P(C_i, C_k)) < \infty$. We propose an algorithm NCSCP to generate the π_{NCSCP} set.

A. Inverse Adjacency List

In NCSCP algorithm, we use the inverse adjacency list as the storage structure, the core operation of the NCSCP algorithm is built on the node based on path.

The inverse adjacency list is a linked list. In the inverse adjacency list, a single linked list of each node of the CIG is established. The inverse adjacency list representation of CIG consists of two parts: the header nodes and single linked list for each node in CIG. For each header node C_i, the single linked list consists of all the nodes and each node exists an edge to C_i.

TABLE I
NUMBER OF COMPONENT AND CONNECTOR

No.	Node	No.	Node
1	GraphicsBinding	12	ClockLogic
2	LayoutManager	13	StatusADT
3	TileArtist	14	ChuteADT
4	StatusArtist	15	WellADT
5	ChuteArtist	16	PaletteADT
6	WellArtist	17	GLConn
7	PaletteArtist	18	LTConn
8	StatusLogic	19	TAConn
9	TileMatchLogic	20	ALAConn
10	NextTilePlacingLogic	21	LLConn
11	RelativePosLogic	22	LAConn

In order to illustrate the correctness of NCSCP algorithm, we choose KLAX system as the application under test. When we choose the beginning component and the stopping component with nodes-constrained, it is necessary to ensure that there exists the shortest component path meets the conditions. At the same time, in order to make the sequence of intermediate results not too long and the influence to visually verify the result is correct, here components and connectors in the CIG are numbered as Table I so as to simplify the representation of inverse adjacency list, and the number of the component GraphicsBinding is 1, the number of the connector LTConn is 18.

Fig. 3 illustrates the inverse adjacency list from the beginning component GraphicsBinding to the stopping component WellADT in Fig. 2, where the number of node represents component and connector. In Fig. 2, component StatusArtist is connected to connector ALAConn, component ChuteArtist is connected to connector ALAConn, component WellArtist is connected to connector ALAConn, and component PaletteArtis is connected to connector ALAConn. Hence, in Fig. 3, there exists $20 \to 4 \to 5 \to 6 \to 7$ where nodes in a single linked list are arranged in no particular order.

B. Data Structure in NCSCP Algorithm

Let there are n nodes C_1, C_2, ..., $C_n \in$ V = Comp $\cup$ Conn, where $C_s \in$ Comp (1 $\leq$ s $\leq$ n) is a beginning node, $C_t \in$ Comp (1 $\leq$ t $\leq$ n) is a stopping node, k is the number of nodes-constrained (1 $\leq$ k $\leq$ n).

- Pointer array adjlist[] of length n: It is a inverse adjacency list, where, each element of adjlist[i] (1 $\leq$ i $\leq$ n) points to a list, there is a path from each element of list to C_t, and all of nodes in list that there is path from these nodes to C_i.
- Array dist[]: Each of element of dist[j] (1 $\leq$ j $\leq$ n) saves the current shortest length from C_s to C_j, initialize the direct distance C_s to C_j.
- Array pdist[]: Saves each element value of dist[] in previous cycle, in order to compare whether the corresponding elements of the dist[] is modified in next cycle.
- Two dimensional array path[][]: Each element of path[k][r] (0 $\leq$ k $\leq$ m - 2, 1 $\leq$ r $\leq$ n) represents the number of direct predecessor nodes C_s to C_r of current shortest path passing through most k nodes.

C. NCSCP Algorithm

This subsection presents our NCSCP algorithm in pseudo-code form. The NCSCP algorithm uses the inverse adjacency list as the storage structure, the core operation of the NCSCP algorithm is based on the nodes of the direct path, and the similar algorithm mostly builded the core operation of the algorithm on all nodes.

The input to the NCSCP algorithm is the CIG, beginning component, stopping component, and the number of nodes-constrained and the output is the corresponding to the nodes-constrained shortest component path. The NCSCP algorithm of our proposed technique to generate the nodes-constrained shortest component path is illustrated as follows.

```
1. for (k=1; k $\leq$ m-2; k++) do
2.   for (j=1; j $\leq$ n; j++) do
3.     for each $C_i$ in V do
4.       if dist[i] > pdist[adjlist[i] -> node] + adjlist[i] -> $\omega$ then
5.         dist[i] = pdist[adjlist[i] -> node] + adjlist[i] -> $\omega$;
6.         path[k][i] = adjlist[i] -> node;
7.       end if
8.       if adjlist[i] -> next ! = NULL then
9.         adjlist[i] = adjlist[i] -> next;
10.      else
11.        if pdist[i] = = dist[i] then
12.          path[k][i] = path[k-l][i];
13.        end if
14.      end if
15.    end for
16.    pdist[j] = dist[j];
17.  end for
18.  i = 1;
19. end for
```

20. Output the shortest component path from C_s to C_t, that is to output the every node in the shortest component path.

According to the description above, there are three nested loops in NCSCP algorithm. The outermost loop takes m - 2 times, the middle loop takes n times, and the number of the innermost loops is related to the C_i nodes. If the number of

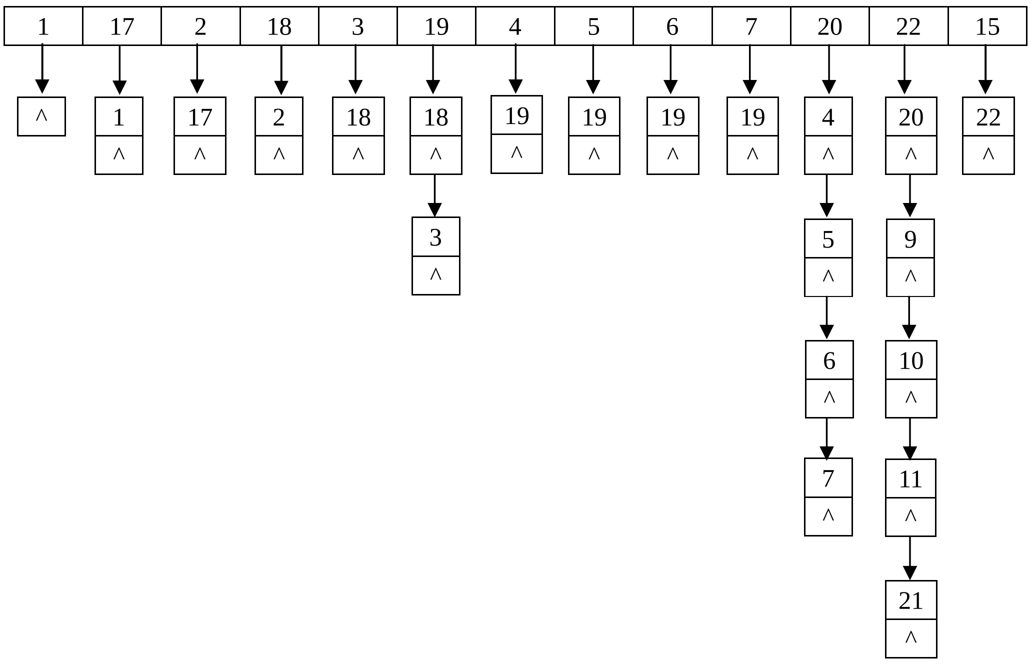

Fig. 3. Inverse Adjacency List Structure of CIG for KLAX System

nodes in the CIG to C_i is e_i, then the loop takes e_i times. Because, $\sum_{i=1}^{n} e_i = w$ (w is the total number of edges in CIG). Thus, the time complexity of NCSCP algorithm is O((m-2)w).

D. *Case Study*

In order to examine the validity of the NCSCP algorithm, we illustrate the working of NCSCP algorithm by using the example KLAX system. Let the beginning component is C_s = 1, stopping component is C_t = 15, and the number of nodes-constrained is m = 9. First, the establishment of CIG's inverse adjacency list is stored as shown in Fig. 3.

The concrete shortest component path solving process is shown in Table II and Table III.

TABLE II
SHORTEST COMPONENT PATH SEARCH PROCESS ON DIST[J]

K	Dist[j]												
	1	17	2	18	3	19	4	5	6	7	20	22	15
0	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
2	0	1	2	3	∞	∞	∞	∞	∞	∞	∞	∞	∞
3	0	1	2	3	4	4	∞	∞	∞	∞	∞	∞	∞
4	0	1	2	3	4	4	5	5	5	5	∞	∞	∞
5	0	1	2	3	4	4	5	5	5	5	6	∞	∞
6	0	1	2	3	4	4	5	5	5	5	6	7	∞
7	0	1	2	3	4	4	5	5	5	5	6	7	8

It is can be seen in Table II and Table III, the length of shortest component path from GraphicsBinding to WellADT is 8, this moment path[7][13] = 22, forwards search path[6][12] = 20, forwards search again path[5][11] = 4, 5, 6, 7 to represent the shortest path length is same as 4, 5, 6, 7. Forwards search again path[4][6] = 19, forwards search

TABLE III
SHORTEST COMPONENT PATH SEARCH PROCESS ON PATH[K][J]

K	Path[k][j]												
	1	17	2	18	3	19	4	5	6	7	20	22	15
0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	17	1	1	1	1	1	1	1	1	1	1
2	1	1	17	2	1	1	1	1	1	1	1	1	1
3	1	1	17	2	18	18	1	1	1	1	1	1	1
4	1	1	17	2	18	18	19	19	19	19	1	1	1
5	1	1	17	2	18	18	19	19	19	19	4,5,6,7	1	1
6	1	1	17	2	18	18	19	19	19	19	4,5,6,7	20	1
7	1	1	17	2	18	18	19	19	19	19	4,5,6,7	20	22

again path[3][4] = 18, forwards search again path[2][3] = 2, forwards search again path[1][2] = 17, forwards search again path[0][1] = 1. Thus, the shortest component path set is:

$1 \to 17 \to 2 \to 18 \to 19 \to 7 \to 20 \to 22 \to 15$

$1 \to 17 \to 2 \to 18 \to 19 \to 6 \to 20 \to 22 \to 15$

$1 \to 17 \to 2 \to 18 \to 19 \to 5 \to 20 \to 22 \to 15$

$1 \to 17 \to 2 \to 18 \to 19 \to 4 \to 20 \to 22 \to 15$

Finally, the numbers are replaced with the component name and the connector name, we obtain following the set of shortest component paths:

GraphicsBinding → GLConn → LayoutManager → LT-Conn → TAConn → StatusArtist → ALAConn → LAConn → WellADT

GraphicsBinding → GLConn → LayoutManager → LT-Conn → TAConn → ChuteArtist → ALAConn → LAConn → WellADT

GraphicsBinding → GLConn → LayoutManager → LT-Conn → TAConn → WellArtist → ALAConn → LAConn → WellADT

GraphicsBinding → GLConn → LayoutManager → LT-Conn → TAConn → PaletteArtist → ALAConn → LAConn → WellADT

TABLE IV
COMPARISON OF STATISTICAL RESULTS BETWEEN DIJKSTRA ALGORITHM AND OUR ALGORITHM

Instance of π_{NCSCP}	Passing through nodes	Dijkstra algorithm	Our algorithm	Percentage reduction
$\pi_{NCSCP}(GraphicsBinding, TileArtist)$	LayoutManager	6	4	33.33%
	LayoutManager, GLConn	5	4	20.00%
	LayoutManager, GLConn, LTConn	4	4	0.00%
$\pi_{NCSCP}(GraphicsBinding, ClockLogic)$	StatusArtist	31	10	67.74%
	StatusArtist, TileArtist	19	9	52.63%
	StatusArtist, TileArtist, LTConn	16	9	43.75%
	StatusArtist, TileArtist, LTConn, LAConn	14	9	35.71%
$\pi_{NCSCP}(WellADT, StatusLogic)$	RelativePosLogic	6	5	16.67%
	RelativePosLogic, LLConn	5	3	40.00%
	RelativePosLogic, LLConn, LAConn	4	3	25.00%
$\pi_{NCSCP}(WellADT, LayoutManager)$	ChuteArtist	17	8	52.94%
	ChuteArtist, TileArtist	12	7	41.67%
	ChuteArtist, TileArtist, ALAConn	10	7	30.00%
	ChuteArtist, TileArtist, ALAConn, LTConn	9	7	22.22%
$\pi_{NCSCP}(LayoutManager, ChuteADT)$	ChuteArtist	17	8	52.94%
	ChuteArtist, TileArtist	12	7	41.67%
	ChuteArtist, TileArtist, ALAConn	10	7	30.00%
	ChuteArtist, TileArtist, ALAConn, LTConn	9	7	22.22%
$\pi_{NCSCP}(PaletteArtist, GraphicsBinding)$	LayoutManager	12	7	41.67%
	LayoutManager, TileArtist	9	6	33.33%
	LayoutManager, TileArtist, LTConn	8	6	25.00%
	LayoutManager, TileArtist, LTConn, GLConn	7	6	14.29%
$\pi_{NCSCP}(PaletteADT, GraphicsBinding)$	StatusArtist	31	10	67.74%
	StatusArtist, TileArtist	19	9	52.63%
	StatusArtist, TileArtist, LTConn	16	9	43.75%
	StatusArtist, TileArtist, LTConn, LAConn	14	9	35.71%

IV. DISCUSSION

Because, it is difficult to effectively monitor the time efficiency and the space efficiency of the NCSCP algorithm. We choose some beginning component and stopping component for KLAX system, and the method is verified by experiment. Table IV gives part of the statistical results obtained by the Dijkstra algorithm and the NCSCP algorithm. In Table IV, the first column represents the instance of the shortest component path between components. Because, the statistics the number of shortest component paths passing through nodes are very complicated, so, the second column represents the passing through specified nodes, we choose the number of the passing through specified nodes as 1, 2, 3, and 4. The third column represents the number of nodes in the search for Dijkstra algorithm on passing through specified nodes. The fourth column represents the number of nodes in the search for our algorithm on passing through specified nodes. The fifth column represents the percentage reduction of every shortest component path passing through specified nodes for the Dijkstra algorithm and our algorithm.

For example, the number of nodes in the search for Dijkstra algorithm from GraphicsBinding to TileArtist passing through LayoutManager is 6, the number of nodes in the search for our algorithm from GraphicsBinding to TileArtist passing through LayoutManager is 4, the percentage reduction is 33.33%. The number of nodes in the search for Dijkstra algorithm from GraphicsBinding to TileArtist passing through LayoutManager and GLConn is 5, the number of nodes in the search for our algorithm from GraphicsBinding to TileArtist passing through LayoutManager and GLConnis 4, the percentage reduction is 20.00%. The number of nodes in the search for Dijkstra algorithm from GraphicsBinding to TileArtist passing through LayoutManager, GLConn, and LTConn is 4, the number of nodes in the search for our algorithm from GraphicsBinding to TileArtist passing through LayoutManager, GLConn, and LTConn is 4, the percentage reduction is 0%. The number of nodes in the search for Dijkstra algorithm from WellADT to LayoutManager passing through ChuteArtist is 17, the number of nodes in the search for our algorithm from WellADT to LayoutManager passing through ChuteArtist is 8, the percentage reduction is 52.94%. The number of nodes in the search for Dijkstra algorithm from WellADT to LayoutManager passing through ChuteArtist and TileArtist is 12, the number of nodes in the search for our algorithm from WellADT to LayoutManager passing through ChuteArtist and TileArtist is 7, the percentage reduction is 41.67%. The number of nodes in the search for Dijkstra algorithm from WellADT to LayoutManager passing through ChuteArtist, TileArtist, ALAConn, and LTConn is 9, the number of nodes in the search for our algorithm from WellADT to LayoutManager passing through ChuteArtist, TileArtist, ALAConn, and LTConn is 7, the percentage reduction is 22.22%.

From Table IV, we can see that, the number of nodes in the search for our algorithm on passing through specified nodes listed in the fourth column is smaller than that the number of nodes in the search for Dijkstra algorithm on passing through specified nodes in the third column. These comparison results

TABLE V
STATISTICS RESULTS OF NUMBER OF SHORTEST COMPONENT PATHS AND ITS LENGTH BETWEEN DIJKSTRA ALGORITHM AND OUR ALGORITHM

Instance of π_{NCSCP}	Passing through nodes	Dijkstra algorithm		Our algorithm	
		#NCSCP	$\omega(\pi_{NCSCP})$	#NCSCP	$\omega(\pi_{NCSCP})$
$\pi_{NCSCP}(GraphicsBinding, TileArtist)$	LayoutManager	1	4	1	4
	LayoutManager, GLConn	1	4	1	4
	LayoutManager, GLConn, LTConn	1	4	1	4
$\pi_{NCSCP}(GraphicsBinding, ClockLogic)$	TAConn	4	8	4	8
	StatusArtist, TileArtist	1	8	1	8
	StatusArtist, TileArtist, LTConn	1	8	1	8
	StatusArtist, TileArtist, LTConn, LAConn	1	8	1	8
$\pi_{NCSCP}(WellADT, StatusLogic)$	LLConn	1	3	1	3
	RelativePosLogic, LLConn	1	4	1	4
	RelativePosLogic, LLConn, LAConn	1	4	1	4
$\pi_{NCSCP}(WellADT, LayoutManager)$	ALAConn	4	6	4	6
	ChuteArtist, TileArtist	1	7	1	7
	ChuteArtist, TileArtist, ALAConn	1	7	1	7
	ChuteArtist, TileArtist, ALAConn, LTConn	1	7	1	7
$\pi_{NCSCP}(LayoutManager, ChuteADT)$	LTConn	4	6	4	6
	ChuteArtist, TileArtist	1	7	1	7
	ChuteArtist, TileArtist, ALAConn	1	7	1	7
	ChuteArtist, TileArtist, ALAConn, LTConn	1	7	1	7
$\pi_{NCSCP}(PaletteArtist, GraphicsBinding)$	LTConn	1	5	1	5
	LayoutManager, TileArtist	1	6	1	6
	LayoutManager, TileArtist, LTConn	1	6	1	6
	LayoutManager, TileArtist, LTConn, GLConn	1	6	1	6
$\pi_{NCSCP}(PaletteADT, GraphicsBinding)$	LTConn	4	8	4	8
	StatusArtist, TileArtist	1	9	1	9
	StatusArtist, TileArtist, LTConn	1	9	1	9
	StatusArtist, TileArtist, LTConn, LAConn	1	9	1	9

mean that our algorithm is superior to the Dijkstra algorithm. Meanwhile, as we can see, with the increase of number of shortest component path passing through specified nodes, the number of nodes in the search for Dijkstra algorithm and our algorithm is decreasing, and the number of nodes in the search our algorithm has small variations. If there exists only shortest component path from beginning component to stopping component, the efficiency of our algorithm will not decrease.

One of the interesting comment for evaluating our algorithm is that, for beginning (stopping) component on the same level and stopping (beginning) component on the same level, the number of nodes in the search passing through specified nodes for Dijkstra algorithm are same as the number of nodes in the search passing through specified nodes for our algorithm. For example in Table IV, for two shortest component paths $\pi_{NCSCP}(GraphicsBinding, ClockLogic)$ and $\pi_{NCSCP}(PaletteADT, GraphicsBinding)$ in Fig. 2, although the direction of the two shortest component paths are different, but the beginning component and stopping component are the same level, so, the number of nodes in the search passing through specified nodes as 1, 2, 3, and 4 for Dijkstra algorithm are same as the number of nodes in the search passing through specified nodes as 1, 2, 3, and 4 for our algorithm.

Table V gives part of the statistical results obtained by the Dijkstra algorithm and the NCSCP algorithm. In Table V, the first column represents the instance of the shortest component path between components. The second column represents the passing through specified nodes, we choose the number of the passing through specified nodes as 1, 2, 3, and 4. The third column #NCSCP represents the number of the shortest component paths passing through specified nodes for Dijkstra algorithm. The fourth column $\omega(\pi_{NCSCP})$ represents the length of the shortest component path passing through specified nodes for Dijkstra algorithm. The fifth column #NCSCP represents the number of the shortest component paths passing through specified nodes for our algorithm. The sixth column $\omega(\pi_{NCSCP})$ represents the length of the shortest component path passing through specified nodes for our algorithm.

In general, it is easy to make sense that, after passing through specified nodes are introduced, the length of the shortest component path is increased, the number of the shortest component path is decreased, and with the increase of number of passing through specified components and connectors, the length of shortest component path is increased, the number of the shortest component path is decreased. The number of the shortest component paths and its length for our algorithm are same as the number of the shortest component paths and its length for our algorithm, the results show that the accuracy of the NCSCP algorithm in computing the shortest component path is 100%.

V. CONCLUSION

This paper proposes an approach to solve the nodes-constrained shortest component path problem in software architecture. Firstly, it describes software architecture through

C2-style, then represents software architecture through component interaction graph CIG, and abstracted the behavior of interaction between components and connectors. Secondly, formalized the shortest component path and nodes-constrained shortest component path, and generated the nodes-constrained shortest component path set according to the NCSCP algorithm. The NCSCP algorithm is proposed based on the research of the shortest component path problem, and it is not affected by the storage structure. By using the inverse adjacency list, the nodes-constrained shortest component path can be realized more easily. How to improve the algorithm, in order to improve the efficiency of the algorithm in the case of larger number of components and connectors and large number of intermediate nodes, it needs further research and exploration.

Acknowledgment

The authors are grateful to the anonymous referees for their detailed comments and insightful suggestions, which helped in refining and improving the presentation of the paper.

References

[1] M. Shahin, P. Liang, and M. A. Babar, "A Systematic Review of Software Architecture Visualization Techniques," *Journal of Systems and Software*, vol. 94, pp. 161-185, 2014.

[2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[3] R. W. Floyd, "Algorithm 97, Shortest Path," *Communications of the ACM*, vol. 5, no. 6, pp. 345, 1962.

[4] L. D. Pugliese and F. Guerriero, "Dynamic Programming Approaches to Solve the Shortest Path Problem with Forbidden Paths," *Optimization Methods & Software*, vol. 28, no. 2, pp. 221-255, 2013.

[5] A. V. Goldberg and C. Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory," in *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, Vancouver, Canada, 2005, pp. 156-165.

[6] S. X. Yang and M. Meng, "An Efficient Neural Network Approach to Dynamic Robot Motion Planning," *Neuralworks*, vol. 13, no. 2, pp. 143-148, 2000.

[7] M. Gen, R. W. Cheng, and D. W. Wang, "Genetic Algorithms for Solving Shortest Path Problems," in *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indian apolis, 1997, pp. 401-406.

[8] W. Wu and Q. Q. Ruan, "A Gene-Constrained Genetic Algorithm for Solving Shortest Path Problem," in *Proceedings of the 7th International Conference on Signal Processing*, Beijing, 2004, pp. 2510-2513.

[9] L. Bass, P. C. Ciements, and R. Kazman, "Software Architecture in Practice," Aonton, MA: Addison-Wesiey, 1998.

[10] L. J. Lun and X. Chi, "Component Dependency Path Coverage Criteria for C2-Style Architecture Testing," *IAENG International Journal of Computer Science*, vol. 42, no. 4, pp. 368-377, 2015.

[11] C. Demetrescu and G. F. Italiano, "A New Approach to Dynamic All Pairs Shortest Paths," *Journal of the ACM*, vol. 51, no. 6, pp. 968-992, 2004.

[12] T. N. Chuang and J. Y. Kung, "The Fuzzy Shortest Path Length and the Corresponding Shortest Path in a Network," *Computers & Operations Research*, vol. 32, no. 6, pp. 1409-1428, 2005.

[13] S. Idwan and W. Etaiwi, "Dijkstra Algorithm Heuristic Approach for Large Graph," *Journal of Applied Sciences*, vol. 11, no. 12, pp. 2255-2259, 2011.

[14] X. Li, X. D. Hu, and W. J. Lee, "On the Union of Intermediate Nodes of Shortest Paths," *Journal of Combinatorial Optimization*, vol. 26, no. 1, pp. 82-85, 2013.

[15] Q. Z. Xu and X. Z. Ke, "Models and Genetic Algorithm for Designated-Points Shortest Path Problem," *Systems Engineering and Electronics*, vol. 31, no. 2, pp. 459-462, 2009.

[16] L. Y. Feng, L. W. Yuan, W. Luo, R. C. Li, and Z. Y. Yu, "Geometric Algebra-Based Algorithm for Solving Nodes Constrained Shortest Path," *ACTA Electronic SINICA*, vol. 42, no. 5, pp. 846-851, 2014.

[17] J. Gao, R. Espinoza, and J. He, "Testing Coverage Analysis for Software Component Validation," in *Proceedings of Annual International Computer Software and Applications Conference*, Edinburgh, UK, 2005, pp. 463-470.

[18] L. J. Lun, L. Zhang, X. Chi, and H. Xu, "Shortest Component Path Generation of C2-Style Architecture Using Improved A* Algorithm," *Journal of Software*, vol. 9, no. 6, pp. 1471-1478, 2014.

[19] N. T. Richard, N. Medvidovic, K. M. Anderson, E. J. Whitehead, and J. E. Robbins, "A Component- and Message-based Architecture Style for GUI Software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390-406, 1996.

[20] M. Muccini, M. Dias, and D. J. Richardson, "Systematic Testing of Software Architectures in the C2 Style," in *Lecture Notes in Computer Science*, vol. 2984, 2004, pp. 295-309.

[21] L. J. Lun, X. Chi, and X. M. Ding, "Edge Coverage Analysis for Software Architecture," *Journal of Software*, vol. 7, no. 5, pp. 1121-1128, 2012.

Lijun Lun was born in Harbin, Heilongjiang Province, China, in 1963. He received his B.S. degree and Master degree in Computer Science and Technology from Harbin Institute Technology of Computer Science and Technology, China, in 1986 and 2000 respectively.

He is currently a professor in computer science and information engineering at Harbin Normal University of Harbin. He has published more than 60 papers in international and Chinese scientific journals. Currently, he teaches and conducts research in the areas of software architecture, software testing and software metrics, etc.

Xin Chi was born in Harbin, Heilongjiang Province, China, in 1990. She received her B.S. degree in Computer Science and Technology from Harbin Normal University of Computer Science and Information Engineering, China, in 2013.

She has published more than 10 papers in international and Chinese scientific journals. Currently, her research interest includes software architecture testing and software metrics, etc.

Hui Xu was born in Harbin, Heilongjiang Province, China, in 1984. She is currently a Ph.D. candidate in computer science and technology at Harbin Engineering University, Harbin. She received her B.S. degree in Information and Computer Engineering from Northeast Forestry University, and Master degree in Computer Science and Technology from Harbin Normal University, China, in 2006 and 2009 respectively.

She has published more than 10 papers in international and Chinese scientific journals. Currently, her research interest includes social computing and complex networks, etc.