Shortest Component Path Generation of C2-Style Architecture Using Improved A* Algorithm

Lijun Lun
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: lunlijun@yeah.net

Lin Zhang
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: 1029372257@qq.com

Xin Chi
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: xinc1990@163.com

Hui Xu
Heilongjiang University of Chinese Medicine Library, Harbin, China
Email: xuhui8413@163.com

***Abstract*—There are always more than one shortest paths between two components in software architecture, and in the application of path selection with additional constraints, several optimal or near optimal paths are desired. Traditional A* algorithm has been successfully used in software testing activities such as finding the shortest path, selecting test suites and test suites prioritization. Little work has been specifically targeted towards the shortest component path of software architecture applications. In this paper, we propose an improved A* algorithm, and combine with an example to explain the algorithm solving process. Finally, we implement the A* algorithm and the improved A* algorithm, and the results are compared. It is shown that the shortest component path using improved A* algorithm is completely feasible and effective.**

***Index Terms*—software architecture; C2-style; component interaction graph; shortest component path; improved A* algorithm**

I. INTRODUCTION

Software architecture [1] represents the earliest software design decisions. These design decisions are the most critical to get right and the most difficult to change downstream in the system development cycle. The software architecture is the first design artifact addressing reliability, modifiability, real-time performance, and inter-operability goals and requirements. Software architectures are nowadays used for different purposes [2], including documenting and communicating design decisions and architectural solutions [3], driving analysis techniques (like testing, model and consistency checking, performance analysis [4,5]), for code generation purposes in model-driven engineering, for product line engineering, for risks and cost estimation [6], and many more.

In the above applications, specific to path generation method in the software architecture testing [7-9], the shortest path problem, which is one of key technologies of software architecture system, concerns with finding the shortest path from a specific origin component to a specified destination component in a given software architecture while minimizing the total distance, time or cost associated with the path.

The Dijkstra algorithm [10] is the one of the best algorithm to solve source-source shortest path problem. This algorithm can be used to directed graph, and can be used to solve undirected graph. However, it can be obtained from the source point to every other point of the shortest path, the result is a shortest path tree. The Dijkstra algorithm is blind, it has the advantages of no need of relevant information to the specific problems, only need to connect the relationship between nodes and edges of the search, the solution is global optimal. In fact, there are many the shortest paths from one point to other point. The Dijkstra algorithm can not give all the shortest paths.

A* algorithm [11] is a heuristic algorithm based on the Dijkstra algorithm. In fact, it is a kind of heuristic search. It is different from the blind type of Dijkstra algorithm, and different from the breadth-first search also. A* algorithm adds heuristic factor associated with the problem domain to reduce the search range in the search process, speed up the search. Specific to the shortest path problem, it is in the choice of the next node, in addition to consider local information known, also make the estimate to the distance from current node to end node, as a measure of evaluation of the node in the possibility of the optimal path. So, it introduces the global information, the search is not blind and the search range is reduced greatly.

doi:10.4304/jsw.9.6.1471-1478

Therefore, improve the efficiency of the Dijkstra algorithm.

However, these traditional algorithms have major shortcomings: firstly, they are not suitable for software architecture with components and connectors; secondly, the algorithms search only for the shortest path, but they cannot determine all the shortest component paths; thirdly, they exhibit high computational complexity. Therefore, the shortest path algorithms tend to be too computationally intensive for real-time one-to-one applications in software architecture.

In this paper, an improved A^* algorithm based on the C2-style architecture model to search for the shortest component path in software architecture is proposed. The technique makes full use of their advantages and uses the improved A^* algorithm to do global search in the beginning of 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.

The paper is organized as follows. In Section 2, C2-architecture model is introduced and briefly discussed. The improved A^* algorithm and the particle encoding mechanism to solve the shortest component path problem in the C2-style architecture is presented in Section 3. The results from computer simulation experiments are discussed in Section 4. In Section 5, an overview of related works on path coverage and software architecture testing coverage are given. Section 6 concludes the paper.

II. BACKGROUND

We introduce here some basic concepts that will be used through this work.

A. A^* Algorithm

A^* algorithm is a heuristic algorithm proposed by Hart et al. to find optimum path from starting position to destination. A^* algorithm's main idea is to treat the testing area as a grid collection and generate the optimized path. In the standard A^* algorithm each movement along the optimized path is evaluated by the formula:

$$f(i) = g(i) + h^*(i)$$

Where, f(i) represents the total moving cost from source to target, while g(i) represents the moving cost from the source to current position, and $h^*(i)$ refers to the estimated moving cost from the current grid to target. The open list keeps grids which are still in the evaluating process by the path finding algorithm, and closed list keeps grids which have already been evaluated by the path finding solution. A^* algorithm searches the whole area by maintaining an open list and a closed list to find an optimized path.

B. 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 [12]; at the same time, the rules of the C2-style are broad enough to render it widely applicable [13].

The C2-style architecture [14] consists of components, connectors, and their constraints. Each component has two connection points, a "top" and a "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 components above and notifications of completed services to components below.

The C2-style architecture control flow is usually represented by a direct graph. We use the Component Interaction Graph (CIG) model [8] to represent interaction relationships between components and connectors.

The following is a formal definition of CIG.

Definition 2.1 Given a C2-style architecture, component interaction graph can be defined as direct graph $CIG = (V, E, V_{start}, V_{end})$, where:

- $V = Comp \cup Conn$ is the set of nodes. Where, $Comp = \{Comp_i.top\_in, Comp_i.top\_out, Comp_i.bottom\_in, Comp_i.bottom\_out\}$ is a finite the set of components. $Conn = \{Conn_i.top\_in, Conn_i.top\_out, Conn_i.bottom\_in, Conn_i.bottom\_out\}$ is a finite the set of connectors. Nodes represent the interface of component and connector, and component interface with a hollow circle, connector interface with a solid circle represents.
- $E \subseteq V \times V$ is a finite set of edges.
- $V_{start} \in \{Comp_i.top\_out \mid Comp_i.bottom\_in = \varnothing \wedge Comp_i.bottom\_out = \varnothing, Comp_i \in Comp\}$ is the initial node, this node transmit messages only.
- $V_{end} \in \{Comp_i.bottom\_in \mid Comp_i.top\_out = \varnothing \wedge Comp_i.top\_in = \varnothing, Comp_i \in Comp\}$ is the terminal node, this node receive messages only.

There are three types of edge in the CIG of a C2-style architecture specification, namely, edge from component to connector, edge from connector to component, and edge from connector to connector, which represents information flows between component and connector.

Definition 2.2 Given a component interaction graph $CIG = (V, E, V_{start}, V_{end})$. The element of E is divided into three edges.

- $e_{Comp\text{-}Conn} = \{e \mid e \in (Comp_i.top\_out, Conn_j.bottom\_in) \vee (Comp_i.bottom\_out, Conn_j.top\_in)\}$ represents edge from component $Comp_i$ to connector $Conn_j$.
- $e_{Conn\text{-}Comp} = \{e \mid e \in (Conn_i.bottom\_out, Comp_j.top\_in) \vee (Conn_i.top\_out, Comp_j.bottom\_in)\}$ represents edge from connector $Conn_i$ to component $Comp_j$.
- $e_{Conn\text{-}Conn} = \{e \mid e \in (Conn_i.top\_out, Conn_j.bottom\_in) \vee (Conn_i.bottom\_out, Conn_j.top\_in)\}$

represents edge from connector $Conn_i$ to connector $Conn_j$.

According to the definition 2.1 and 2.2, it is easy to find that CIG can be constructed in the following four steps.

- The C2-style architecture of each component interface and connector interface, an increase in the corresponding node.
- Add the edge from component to connector to attach the CIG. Obviously, this type of edge belongs to the set of $e_{Comp\text{-}Conn}$.
- Add the edge from connector to component to attach the CIG. Obviously, this type of edge belongs to the set of $e_{Conn\text{-}Comp}$.
- Add the edge from connector to connector to attach the CIG. Obviously, this type of edge belongs to the set of $e_{Conn\text{-}Conn}$.

Consider an example KLAX system [12], it is a video game of the C2-style architectural. The C2-style architecture designed for the KLAX system is shown in Fig. 1.

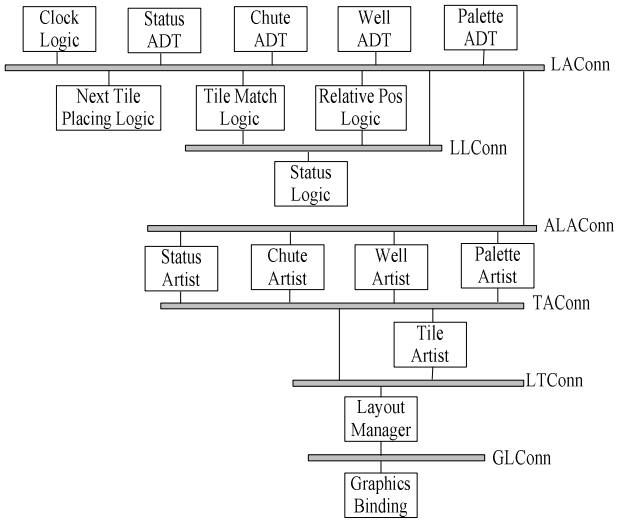


Fig. 1. KLAX architecture in the C2-style

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 [13].

C. Shortest Component Path

We model component interactions using CIG which depicts interaction scenarios among components. Coverage criteria require that a set of entities of the CIG is covered when the test cases are executed.

Definition 2.3 Given a component interaction graph CIG = (V, E, V_{start}, V_{end}) for C2-style architecture, C_1, C_2, …, $C_k \in$ Comp $\cup$ Conn. A path from node C_1 to C_k is a sequence of nodes $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k$, such that for i = 1, 2, …, k-1, each $(C_i, C_{i+1}) \in e_{Conn\text{-}Comp} \vee e_{Comp\text{-}Conn} \vee e_{Conn\text{-}Conn}$. If $C_1 \in$ Comp $\wedge$ $C_k \in$Comp, then the path is a component path for the CIG, called CP for short. The length of CP is the number of edges from C_1 to C_k, called $\omega(C_1, C_k)$ for short.

CP describes the messages transfer between components in C2-style architecture. In fact, a 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.

From Fig. 2, there are two component paths from LayoutManager to StatusArtist given below.

(1) LayoutManager → LTConn → TAConn → StatusArtist → ALAConn → LAConn → WellADT

(2) LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → WellADT

Definition 2.4 Given a component interaction graph CIG = (V, E, V_{start}, V_{end}) for C2-style architecture, C_i, C_j. ∈Comp. The shortest component path SCP(C_i, C_j) from C_i to C_j is defined as follow:

$$SCP(C_i, C_j) = \begin{cases} \min\{w(C_i, C_j)\} & \text{if CP exists from } C_i \text{ to } C_j \\ \infty & \text{otherwise} \end{cases}$$

From Fig. 2, there is a shortest component path from LayoutManager to StatusArtist given below.

(1) LayoutManager → LTConn → TAConn → StatusArtist → ALAConn → LAConn → WellADT

The shortest component path have properties as follow:

(1) Subpaths of the shortest component path are shortest component path: If SCP = $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k$ is the shortest component path from C_1 to C_k, then for all $1 \le i \le j \le k$, SCP′ = $C_i \rightarrow C_{i+1} \rightarrow \ldots \rightarrow C_j$ is the shortest component path from C_i to C_j.

(2) For any the shortest component path SCP = $C_i \rightarrow C_{i+1} \rightarrow \ldots \rightarrow C_k$, for all $i \le j \le k$, SCP(C_i, C_k) ≤ SCP(C_i, C_j) + $\omega(C_j, C_k)$.

III. An Algorithm for Shortest Component Path

We present our improved A* algorithm for automatic generation of the shortest component paths for the C2-style architecture, which uses a new evaluation function to evaluate the generated the shortest component path.

A. SCPA Generation Algorithm

To generate SCP, we propose an algorithm called the shortest component path algorithm (SCPA). The input to the SCPA algorithm is the CIG and the output is the corresponding the shortest component path set.

The SCPA algorithm requires two tables, respectively, to save for the lattice to be detected and has been detected. One array saves the nodes of the shortest component path.

- OpenTable — The OpenTable is used to save nodes which have not been searched.
- CloseTable — The CloseTable is used to save nodes which have been searched.
- Array — The Array is used to save nodes in the shortest component path.

The improved A* algorithm use the idea that evaluate the current node in A* algorithm, increase the parent node of the current node in the shortest component path and

evaluate the distance from the node to destination node. The core of the algorithm is: use the sum of the shortest distance from current node to start node, the value from current node to destination node and the value from the parent node of current node to destination node as the possible measure that evaluate the node in the shortest component path. This optimization method is called improved A^* algorithm. Assume the current node is i, then its evaluation function can be defined as:

$$f(i) = g(i) + h^*(i) + h^*(j)$$

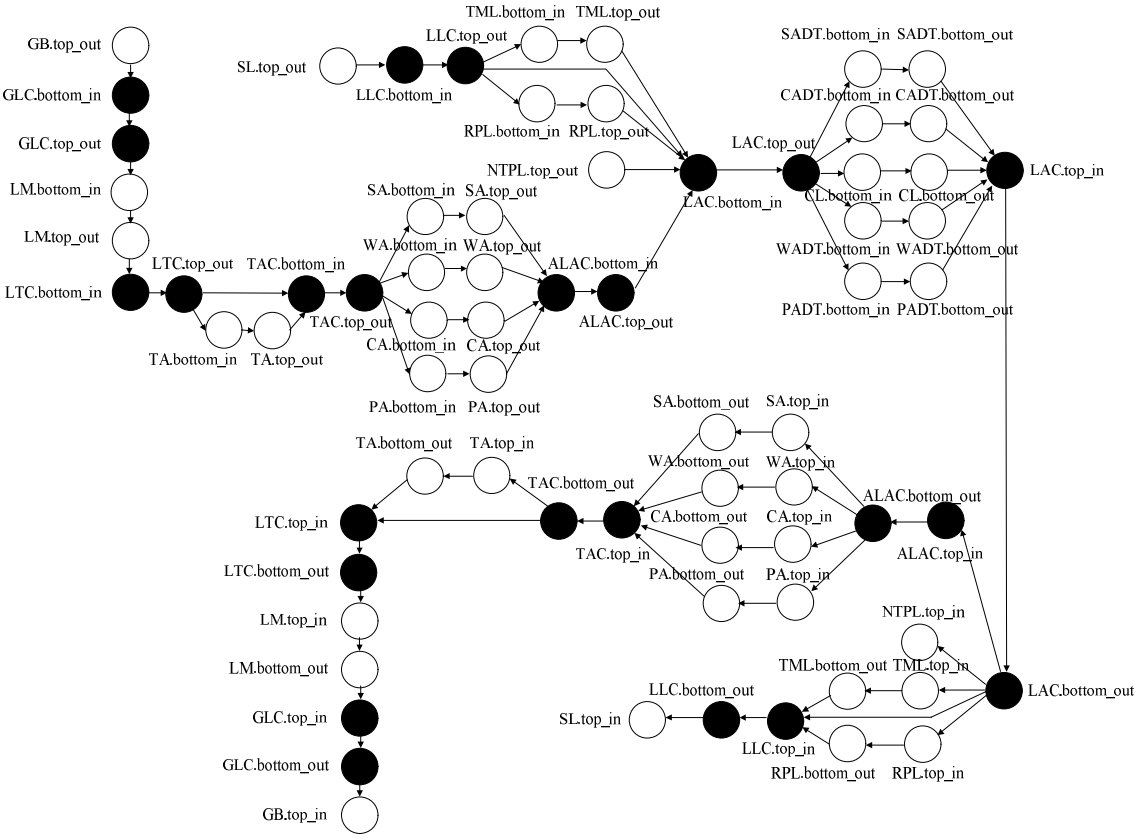


Fig. 2. CIG of KLAX system

Where, the g(i) and $h^*(i)$ is same as the A^* algorithm, g(i) is the shortest distance from current node to start node, $h^*(i)$ is the value of shortest distance from current node to destination node, $h^*(j)$ is the value of shortest distance from the parent node j of current node to the destination node.

In our approach, the basic procedure to perform the shortest component path consists of the following steps:

- Traverse the OpenTable, get the node has minimum value f from OpenTable. If current node is the destination node, save the nodes in Array, output the shortest component path in reverse chronological order, output current shortest component path. If a node and the node in current shortest component path at the same level, replace the node, output the shortest component path.
- If next node of current node to other nodes has way, extend node. If the node is not in OpenTable and CloseTable, update g, h, f, set the parent node to the current node and insert the node into OpenTable. After extend, delete current node in OpenTable, insert CloseTable.

The pseudo code for SCPA is shown as follow.

Algorithm SCPA(CIG, SCPS)

Input: CIG is component interaction graph
Output: SCPS is the shortest component path set
begin
for each C_i in Comp **do**
  for each C_j in Comp **do**
    SCP(C_i, C_j);
end
Procedure SCP(C_i, C_j)
begin
C_i save to CloseTable;
for (; ;)
{ search next node of C_i, and insert into OpenTable;}
while (openTable ! = NULL) // traverse the OpenTable
{ getTableNode(); // get the minimum value f of the node in OpenTable
  if (current.index == end.index) // if current node is the destination node
  { // output current SCP
    i = 0;
    while (current != NULL)
    { Array[i++] = p - > node.index;
      current = current - > parent;} // save the nodes in Array, output SCP in reverse chronological order
      // if a node and the node in current SCP at the same level of node, replace the node, out SCP

```
        return;
      }
      else
      { for (n = 0; n < countOfNodes; n + +)
        { if (current - > next.index == 1) // if there is a path
from current node to other nodes
          {  // extend node
            if (!isInOpenTable(current - > next.index, n)
&&!isInCloseTable(current - > next.index, n))
            { // calculate g, if greater than the original, do
nothing, or set the parent node to the current point, update
g, f
              if (current - > next.index == end.index)h = 1;
              else
              { if (current - > next.index < end.index)
                  h = abs(end_node_id - node_id + 7);
                else h = abs(8 + end_node_id - node_id);
              }
              g = p - > g + 1;
              f = g + h + p - > h;
              node - > node_index = current - > next.index;
              node - > g = g;
              node - > h = h;
              node - > f = f;
              node - > parent = current; // set the parent
node to the current node
              addToTable(&OpenTable, node);
            }
          }
          delete current node in OpenTable, insert
CloseTable;
          addToTable(&CloseTable, current); // insert
current node into CloseTable;
      }
    }
    end Algorithm SCPA
```

B. Illustration of Working of SCPA

We explain the working of SCPA algorithm by using the example form WellADT to GraphicsBinding. First, number the component and connector in the CIG is shown as Tab. I, and assume i = 0, 1, …, 21.

TABLE I.
NUMBER OF COMPONENT AND CONNECTOR

Number	Node	Number	Node
0	ClockLogic	11	WellArtist
1	StatusADT	12	PaletteArtist
2	ChuteADT	13	TileArtist
3	WellADT	14	LayoutManager
4	PaletteADT	15	GraphicsBinding
5	NextTilePlacingLogic	16	LAConn
6	TileMatchLogic	17	LLConn
7	RelativePosLogic	18	ALAConn
8	StatusLogic	19	TAConn
9	StatusArtist	20	LTConn
10	ChuteArtist	21	GLConn

After the SCPA algorithm execution, CloseTable is the node set of the shortest component path start from 3. f(i) is the value of current node, it is the sum of g(i) of the shortest distance from current node i to start node. $h^*(i)$ is the value from i to destination node. $h^*(j)$ is the value from the parent node j of i to the destination node. F(i) is the parrent node of current node.

Step 1: Insert 3 into CloseTable, insert 16 which is the next node of 3 into OpenTable, then OpenTable = {16}. Because there is only one node in OpenTable, then insert 16 into CloseTable. At this time, CloseTable = {3, 16}, OpenTable = ∅, F(16) = 3.

Step 2: Insert 5,6,7,17,18 which are the next node of 16 into OpenTable, then OpenTable = {5,6,7,17,18}. $h^*(16) = 7$, $g(5) = 2$, $h^*(5) = 17$, $f(5) = 2+17+7 = 26$, $g(6) = 2$, $h^*(6) = 16$, $f(6) = 2+16+7 = 25$; $g(7) = 2$, $h^*(7) = 15$, $f(7) = 2+15+7 = 24$; $g(17) = 2$, $h^*(17) = 6$, $f(17) = 2+6+7 = 15$; $g(18) = 2$, $h^*(18) = 5$, $f(18) = 2+5+7 = 14$. We see 18 has the minimum value f, then insert 18 into CloseTable. At this time, CloseTable = {3,16,18}, OpenTable = {5,6,7,17}, F(18)=16.

Step 3: Insert 9,10,11,12 which are the next node of 18 into OpenTable, then OpenTable = {5,6,7,17,9,10,11,12}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(17) = 15$; $g(9) = 3$, $h^*(9) =13$, $f(9) = 3+13+5 = 21$; $g(10) = 3$, $h^*(10) = 12$, $f(10) = 3+12+5 = 20$; $g(11) = 3$, $h^*(11) = 11$, $f(11) = 3+11+5 = 19$; $g(12) = 3$, $h^*(12) = 10$, $f(12) = 3+10+5 = 18$. We see 17 has the minimum value f, then insert 17 into CloseTable. At this time, CloseTable = {3,16,18,17}, OpenTable = {5,6,7,9,10,11,12}, F(17) = 16.

Step 4: Insert 8 which is the next node of 17 into OpenTable, then OpenTable = {5,6,7,9,10,11,12,8}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(9) = 21$, $f(10) = 20$, $f(11) = 19$, $f(12) = 18$; $g(8) = 3$, $h^*(8) = 14$, $f(8) = 3+14+6 = 23$. We see 12 has the minimum value f, then insert 12 into CloseTable. At this time, CloseTable = {3,16,18,17,12}, OpenTable = {5,6,7,9,10,11,8}, F(12)=18.

Step 5: Insert 19 which is the next node of 12 into OpenTable, then OpenTable = {5,6,7,9,10,11,8,19}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(9) = 21$, $f(10) = 20$, $f(11) = 19$, $f(8) = 23$; $g(19) = 4$, $h^*(19) = 4$, $f(19) = 4+4+10 = 18$. We see 19 has the minimum value f, then insert 19 into CloseTable. At this time, CloseTable = {3,16,18,17,12, 19}, OpenTable = {5,6,7,9,10,11,8}, F(19)=12.

Step 6: Insert 13,20 which are the next node of 19 into OpenTable, then OpenTable = {5,6,7,9,10,11,8,13,20}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(9) = 21$, $f(10) = 20$, $f(11) = 19$, $f(8) = 23$; $g(13) = 5$, $h^*(13) = 9$, $f(13) = 5+9+4 = 18$; $g(20) = 5$, $h^*(20) = 3$, $f(20) = 5+3+4 = 12$. We see 20 has the minimum value f, then insert 20 into CloseTable. At this time, CloseTable = {3,16,18,17,12,19,20}, OpenTable = {5,6,7,9,10,11,8,13}, F(20) = 19.

Step 7: Insert 14 which is the next node of 20 into OpenTable, then OpenTable = {5,6,7,9,10,11,8,13,14}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(9) = 21$, $f(10) = 20$, $f(11) = 19$, $f(8) = 23$, $f(13) = 18$; $g(14) = 6$, $h^*(14) = 8$, $f(14) = 6+8+3 = 17$. We see 14 has the minimum value f, then insert 14 into CloseTable. At this time, CloseTable = {3,16,18,17,12,19,20,14}, OpenTable = {5,6,7,9,10,11,8, 13}, F(14)=20.

Step 8: Insert 21 which is the next node of 14 into OpenTable, then OpenTable = {5,6,7,9,10,11,8,13,21}. $f(5) = 26$, $f(6) = 25$, $f(7) = 24$, $f(9) = 21$, $f(10) = 20$, $f(11) = 19$, $f(8) = 23$, $f(13) = 18$; $g(21) = 7$, $h^*(21) = 2$, $f(21) = 7+2+8 = 17$. We see 21 has the minimum value f, then

insert 21 into CloseTable. At this time, CloseTable = {3,16,18,17,12,19,20,14,21}, OpenTable = {5,6,7,9,10,11, 8,13}, F(21) = 14.

Step 9: Insert 15 which is the next node of 21 into OpenTable, then OpenTable = {5,6,7,9,10,11,8,13,15}. f(5) = 26, f(6) = 25, f(7) = 24, f(9) = 21, f(10) = 20, f(11) = 19, f(8) = 23, f(13) = 18; g(15) = 8, $h^*(15) = 1$, f(15) = 8+1+2 = 11. We see 15 has the minimum value f, then insert 15 into CloseTable. At this time, CloseTable = {3,16,18,17,12,19,20,14,21,15}, OpenTable = {5,6,7,9,10, 11,8,13}, F(15) = 21.

Because the 15 is the destination node, then use the backtracking method to find its parent node 21 and so on. In the end, find the start node 3, then get the shortest component path from the start node to the destination node {3,16,18,12,19,20,14,21,15}.

Because 12 and 9,10,11 are at the same level, so use them replace 12, we obtain other shortest component paths {3,16,18,11,19,20,14,21,15}, {3,16,18,10,19,20,14, 21,15}, and {3,16,18,9,19,20,14,21,15}. Replace the number with the component name and the connection name. We obtain the shortest component path set from WellADT to GraphicsBinding is as follow:

(1) WellADT→LAConn→ALAConn→PaletteArtist→ TAConn→LTConn→LayoutManager→GLConn→Grap hicsBinding

(2) WellADT→LAConn→ALAConn→WellArtist→ TAConn→LTConn→LayoutManager→GLConn→Grap hicsBinding

(3) WellADT→LAConn→ALAConn→ChuteArtist→ TAConn→LTConn→LayoutManager→GLConn→Grap hicsBinding

(4) WellADT→LAConn→ALAConn→StatusArtist→ TAConn→LTConn→LayoutManager→GLConn→Grap hicsBinding

Ⅳ. Experimental Results and Analysis

To better understand our approach, we have performed an experimental study by applying the shortest component path for KLAX system.

A. Experimental Results

We have implemented a prototype tool that generates the shortest component path generation automatically by our approach. We have implemented our tool using Java. The tool uses C2-ADL specification as input. Then it analyzes the names of all components, connectors and interfaces, interfaces types, the connection relationship between components and connectors and so on. These are stored in corresponding data structure respectively. Then according to the shortest component path generation method, it can generate the shortest component path set. In addition, the tool also provides the help documents about the details of the system functions, the operation and some open source code in an html format and so on

Table II gives percentage improvement of every component for the A^* algorithm and the improved A^* algorithm.

TABLE II.
A^* ALGORITHM VS IMPROVED A^* ALGORITHM FOR KLAX SYSTEM

Component	A^* Algorithm	Our Algorithm	Percentage Improvement
ClockLogic	60	56	6.67%
StatusADT	60	56	6.67%
ChuteADT	60	56	6.67%
WellADT	60	56	6.67%
PaletteADT	60	56	6.67%
NextTilePlacingLogic	15	15	0
TileMatchLogic	18	18	0
RelativePosLogic	18	18	0
StatusLogic	26	26	0
StatusArtist	34	33	2.94%
ChuteArtist	34	33	2.94%
WellArtist	34	33	2.94%
PaletteArtist	34	33	2.94%
TileArtist	50	50	0
LayoutManager	57	57	0
GraphicsBinding	77	77	0

B. Experimental Results Analysis

- From Table II, we have: For component ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT, the improved A^* algorithm is more efficient than A^* algorithm for improving 6.67%. For component StatusArtist, ChuteArtist, WellArtist, and PaletteArtist, the improved A^* algorithm is more efficient than A^* algorithm for improving 2.94%. Compared with improved A^* algorithm and A^* algorithm, the extend node reduced, search efficiency improved.
- The efficiency of some components increase, some doesn't. Through the experiment, we can draw the conclusion: If the current node to destination node has multiple shortest component paths, the efficiency will increase. If the current node to destination node has only one, the efficiency will not increase.
- When search in the algorithm, the improved evaluation function made the search direction in the A^* algorithm faster towards the destination node, greatly reduce the number of traversal node in the algorithm, and improved the search speed. Of course, if need, we can increase the parent node of the parent node of the parent node of current node. In the same way, evaluate the distance from the node to destination node.
- Not increasing more evaluation node, the efficiency of the algorithm is higher. Though increasing more evaluation node, the traversal node in the algorithm will be less, but first, it will increase the storage capacity, second it will increase the evaluation number, thus increase the burden of algorithm.

Ⅵ. Related Work

Path coverage is a kind of important standard that investigates the sufficiency of software testing [15,16]. The goal of software architecture testing is to choose a certain effect and low cost of coverage criteria, or

according to certain criteria, selection of a finite subset of all logical path to test, using the minimum test case, finding the error of software architecture. There are a few path generation method of the software architecture.

Zhenyi and Offutt proposed a technology of generating test cases [17] in view of architecture description language Wright, according to Interface Connectivity Graph (ICG) and Behavior Graph (BP), and developed testing criteria for generating software architecture level tests from software architecture descriptions.

Stafford et al. introduced software architecture dependence analysis technique [18,19], called chaining, to support software architecture development such as debugging and testing. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing chain of dependencies that can be followed during analysis.

Gao et al. focused on component test coverage issues, and proposed test models (CFAGs and D-CFAGs) [20] to represent a component's API-based function access patterns in static and dynamic views. A set of component API-based test criteria is defined based on the test models, and a dynamic test coverage analysis approach is provided.

Hashim et al. presented Connector-based Integration Testing for Component-based Systems (CITECB) with an architectural test coverage criteria [21], and describe the test models used that are based on probabilistic deterministic finite automata which are used to represent gate usage profiles at run-time and test execution. It also provides a measuring mechanism of how well the existing test suite are covering the component interactions and provides a test suite coverage monitoring mechanism to reveal the test elements that are not yet covered by the test suites. The model extraction technique used to generate the CITECB test models is a simple and less time consuming process. In addition to that, these test models are able to closely represent the component interactions as they are extracted directly from the system.

The shortest path problem [22] is one of the most important optimization problems in such fields as computer sciences and artificial intelligence.

Valverde and Solé [23] stated that the software architecture references as software graph. Consider the measurement experiment in a software graph: choose a pair of nodes (v_i, v_j) and then trace a path from v_i to v_j while traversing the minimum number of edges. Count how many edges have traversed in such path and will obtain the length of the shortest path between the two end-nodes: $d_{min}(i, j)$. According to the shortest path between the two end-nodes, we get is the mean average path length (or characteristic path length) for the software graph.

Ⅶ. Conclusions and Future Work

This paper presents an approach of the shortest component path for C2-style architecture. First, it describes software architecture through C2-style, then represents software architecture through CIG, and abstracted the behavior of interactive between components and connectors. Formalized the shortest component path, generated the shortest component path set that covered the architecture according to improved A^* algorithm. This technology could establish an abstract model to describe the characteristics of dynamic architecture, it covered all the testing component nodes and reduced scale of testing coverage set, so that test the architecture effectively.

As for the future work, the application of the approach needs to study at the implementation level. It is also planned to investigate other testing criteria and testing criteria adequacy and the approach to generate test cases which satisfy the testing criteria without necessarily simulating the execution process of all possible test paths.

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. Part of this work is supported by the Scientific Research Foundation of Heilongjiang Provincial Education Department of China under Grant No. 12541250.

References

[1] D. E. Perry, A. L. Wolf, “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Softw.Eng. Notes*, vol. 17, pp. 40-52, October 1992.

[2] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang. “What Industry Needs from Architectural Languages: A Survey”, *IEEE Trans. on Software Engi.*, vol. 39, pp. 869-891, June 2013.

[3] H. Muccini, A. Bertolino, P. Inverardi, “Using Software Architecture for Code Testing”, *IEEE Trans. Software Engi.*, vol. 29, pp. 160-171, March 2003.

[4] P. Zhang, H. Muccini, B. Li, “A Classification and Comparison of Model Checking Software Architecture Techniques”, *J. Systems and Software*, vol. 83, pp. 723-744, May 2010.

[5] P. Pelliccione, P. Inverardi, H. Muccini, “Charmy: A Framework for Designing and Verifying Architectural Specifications”, *IEEE Trans. Software Engi.*, vol. 35, pp. 325-346, May-June 2009.

[6] E. R. Poort, H. Vliet, “Architecting as a Risk- and Cost Management Discipline”, *in: Proc. of IEEE/IFIP Ninth Working Conference Software Architecture (WICSA2011)*, Boulder, Colorado, USA, pp. 2-11, June 2011.

[7] L. J. Lun, H. Xu, “An Approach to Software Architecture Testing”, *in: Proc. of 9th International Conference for Young Computer Scientists (ICYCS2008)*, Zhangjiajie, China, pp. 1070-1075, November 2008.

[8] L. J. Lun, X. Chi, X. M. Ding, “Edge Coverage Analysis for Software Architecture”, *Journal of Software*, vol. 7, pp. 1121-1128, May 2012.

[9] L. J. Lun, X. Chi, X. M. Ding, “C2-Style Architecture Testing and Metrics Using Dependency Analysis”, *Journal of Software*, vol. 8, pp. 276-285, February 2012.

[10] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs”, *Numer. Math.*, vol. 1, pp. 269-271, December 1959.

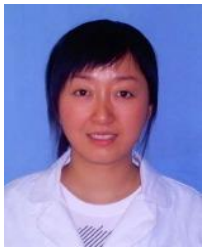
[11] C. Zeng, Q. Zhang, X. P. Wei, "GA-based Global Path Planning for Mobile Robot Employing A* Algorithm", *Journal of Computers*, vol. 7, pp. 470-474, February 2012.
[12] N. T. Richard, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, "A Component- and Message-Based Architecture Style for GUI Software", *IEEE Trans. on Software Engi.*, vol. 22, pp. 390-406, June 1996.
[13] H. Muccini, M. Dias, D. J. Richardson, "Systematic Testing of Software Architectures in the C2 style", *in: Proc. Conference Fundamental Approaches to Software Engineering (FASE2004)*, Barcelona, Spain, LNCS 2984, pp. 295-309, March 2004.
[14] The C2 Architectural Style, http://www.ics.uci.edu/pub/arch/c2.html.
[15] M. Marré, A. Bertolino, "Automatic Generation of Path Covers based on the Control Flow Analysis of Computer Programs", *IEEE Trans. on Software Engi.*, vol. 20, pp. 885-899, December 1994.
[16] G. Ye, X. J. Li, D. Yu, Z. W. Li, J. Yin, "The Design and Implementation of Workflow Engine for Spacecraft Automatic Testing", *Journal of Computers*, vol. 6, pp. 1145-1151, June 2011.
[17] J. Zhenyi, J. Offutt, "Deriving Tests from Software Architectures", *in: Proc. of 12th IEEE International Symposium on Software Reliability Engineering*, Washington, DC, USA, pp. 308-313, November 2001.
[18] J. A. Stafford, D. J. Richardson, A. L. Wolf, "Architecture-Level Dependence Analysis for Software Systems", *Int. J. Softw. Eng. Know.*, vol. 11, pp. 431-451, April 2001.
[19] J. A. Stafford, A. L. Wolf, M. Caporuscio, "The Application of Dependence Analysis to Software Architecture Descriptions", *LNCS*, vol. 2804, pp. 52-62, September 2003.
[20] J. Gao, R. Espinoza, J. He, "Testing Coverage Analysis for Software Component Validation", *in: Proc. Annual International Computer Software and Applications Conference (COMPSAC2005)*, Edinburgh, Scotland, UK, pp. 463-470, July 2005.
[21] N. L. Hashim, S. Ramakrishnan, H. W. Schmidt, "Architectural Test Coverage for Component-based Integration Testing", *in: Proc. International Conference on Quality Software (QSIC2007)*, Portland, Oregon, USA, pp. 262-267, October 2007.
[22] Y. Gu, "Research on Optimization of Relief Supplies Distribution Aimed to Minimize Disaster Losses", *Journal of Computers*, vol. 6, pp. 603-609, March 2011.
[23] S. Valverde, R. V. Solé, "Hierarchical Small Worlds in Software Architecture", *Dynamics of Continuous Discrete and Impulsive Systems: Series B; Applications and Algorithms*, 2007.

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.

Currently, he is a professor, and teaches and conducts research in the areas of software architecture, software testing, and software metrics, etc.

Lin Zhang was born in Mudanjiang, Heilongjiang Province, China, in 1989. She received her B.S. degree in Computer Science and Technology from Harbin Normal University of Computer Science and Technology, China, in 2012. Her research topics are mainly in software architecture and path planning.

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, China, in 2013. Her research topics are mainly in software architecture testing and software metrics.



Hui Xu was born in Harbin, Heilongjiang Province, China, in 1984. She received her Master degree in Computer Science and Technology from Harbin Normal University, China, in 2009. Now she is studying her PhD degree in Harbin Engineering University of Computer Science and Technology, China. Her research topics are mainly in social network and social computing.