Scalable State Space Search with Structural-Bottleneck Heuristics for Declarative IT System Update Automation

Takuya KUWAHARA\(^{(a)}\), Takayuki KURODA\(^{(b)}\), Manabu NAKANOYA\(^{(c)}\), Yutaka YAKUWA\(^{(d)}\), Nonmembers, and Hideyuki SHIMONISHI\(^{(e)}\), Member

SUMMARY As IT systems, including network systems using SDN/NFV technologies, become large-scale and complicated, the cost of system management also increases rapidly. Network operators have to maintain their workflow in constructing and consistently updating such complex systems, and thus these management tasks in generating system update plan are desired to be automated. Declarative system update with state space search is a promising approach to enable this automation, however, the current methods is not enough scalable to practical systems. In this paper, we propose a novel heuristic approach to greatly reduce computation time to solve system update procedure for practical systems. Our heuristics accounts for structural bottleneck of the system update and advance search to resolve bottlenecks of current system states. This paper includes the following contributions: (1) formal definition of a novel heuristic function specialized to system update for A* search algorithm, (2) proofs that our heuristic function is consistent, i.e., A* algorithm with our heuristics returns a correct optimal solution and can omit repeatedly expansion of nodes in search spaces, and (3) results of performance evaluation of our heuristics. We evaluate the proposed algorithm in two cases; upgrading running hypervisor and rolling update of running VMs. The results show that computation time to solve system update plan for a system with 100 VMs does not exceed several minutes, whereas the conventional algorithm is only applicable for a very small system.

key words: orchestration, task planning, system update automation, automated planning, change management, model-based engineering, declarative provisioning, network function virtualization

1. Introduction

Recent development and wide acceptance of SDN [1], [2] and NFV [3], [4] technologies have made network infrastructure greatly flexible, i.e., deployment, modification, scale-in and out, as well as destruction of (virtual) network systems can be done in a software-defined way. Network functions, such as router, firewall, load-balancer, IDS/IPS, and so on, are deployed as a virtual entity on any available physical servers, rather than manually installing physical boxes into the system. Scaling-up the number of such functions, for instance, can be done by instructing orchestrators or NFV platform software, rather than purchasing any new hardware boxes. Researches on automation of NFV resource optimization can be found in many literatures, such as [5], for example. In addition, in plumbing among these functions, there’s no need to physically wire among them, or set up complex VLAN/routing configurations with a lot of constraints, but SDN technology enables ideal plumbing among these functions with arbitrary logical topology, appropriate path selection, and logical separation from other services, regardless of physical network installations. Optimization of co-design of NFV function placement and SDN route optimization can also be found in many literatures, such as [6], for example.

These technologies have made operator’s tasks for deployment and maintenance of the system significantly ease and agile, however, these tasks have to be well-prepared for every situations. Workflows for deployment and maintenance tasks have to be carefully designed so that the order of each step in the workflow ensures no system failure nor effects on running services. This is an another burden for operators to design such workflows in advance of the service operation, as network systems become more complex and demands for network services adapted to allow diverse customization requirement. In addition, such pre-defined workflow can only be applied for well-prepared regular tasks, such as increasing the number of functions within a prepared resource pool, or preparing a new virtual network based with common templates. Let us suppose a case, when a resource pool in some small area has been exhausted and needs to allocate extra resource in different area, there would be a manual configurations to fetch resource in remote area, set up a network path, or extend virtual network span to that area, re-configure service chain topology or job dispatch policy, etc. Someone can imagine another case where any network hardware or service halted unexpectedly and simple switching to stand-by system is not applicable, or there’s any trouble at a single point of failure, the operator is urged to prepare recovery workflow on-demand as quickly as possible. We can also imagine a workflow to patch or upgrade hypervisor, which requires delicate treatment of services running on top of that hypervisor because such services have mutual effects with other services running at other part of the system.

Based on above discussions, we are facing to new frontier of SDN/NFV research towards automation of workflow generation. This will make manual and labor jobs of operation expert fully automated and possibly autonomous self-management of a network infrastructure. In this paper, we discuss automated system update to eliminate human task
of generation of system update plan. To this end, we are proposing to employ Declarative system update [7]–[12] for this issue. Taking this approach, system operators have only to input the desired system state, and planning engines automatically generate system-update plans on behalf of the system operators.

Some declarative system update approaches [7], [8] adopt a “state-space search” as an automated planning method. Although it has been suggested that a state space of a system grows exponentially with increasing number of system components [11], our previous work proposed a kind of a divide-and-conquer technique to solve planning problems concerning updating large-scale systems [8].

However, in some cases, exponential growth of the state space remains an issue concerning planning problems. Accordingly, the notion of global constraints was introduced to planning system updates [12]. This notion makes it possible to impose certain system requirements on the transient and desired states of a system update. For example, when system operators need to update a service that manages critical infrastructure and cannot be stopped even during system update procedures, planning engines accounting for global constraints can generate a system-update plan while keeping the system operating normally.

Unfortunately, if we introduce global constraints to our previous approach, our divide-and-conquer strategy may not sufficiently break an original planning problem down to fine grained sub-problems. In such cases, system update would not be finished in realistic time, even if the size of the system is realistic, for example, consisting of 50 ~ 100 components.

In our preliminary report, we have proposed a basic method for efficient planning of declarative system update [13]. In this paper, we further propose complete method which can omit recomputation of values of heuristic function to be more efficient and present mathematical proof of justification for this optimization. We also present various experimental results to verify the effectiveness of the method.

The proposed method suppresses the increase in time for planning and makes declarative system update a more powerful tool for system management. The proposed planning method is based on the observation that most system updates contain a “bottleneck” component, which takes many steps to update due to dependencies with other components. We found that a system-update plan can be quickly formulated by setting the update of the bottleneck components as a priority. We formalized this finding as a heuristic function for the A* search algorithm, and we designed an efficient planning method tailored for declarative system update.

In the rest of this paper, related work is introduced in Sect. 2. In Sect. 3, declarative system update with state-space search is overviewed. In Sect. 4, state models and planning problems are formalized in the similar way to a reported method [8]. In Sect. 5, a heuristic function is formulated and validated, and an efficient algorithm for calculating that heuristic function is devised and validated. In Sect. 6, the results of an experimental evaluation of the performance of the algorithm are presented and discussed, and in Sect. 7, the conclusions of this study are presented.

2. Related Work

A fundamental framework for declarative system update, which accepts a desired state as input and generates workflows that execute system update by automatic planning, has been proposed by El Maghraoui et al. [11].

The planning technique used in [11] is called partial-order planning (POP). In short, a POP engine receives a set of actions, which are elemental operations for system update (and similar to the transitions in our models), and constructs a partial plan.

In contrast to a totally ordered plan specifying a complete order of actions, a partial plan is a partially ordered set of actions and specifies only necessary and sufficient ordering of actions. A partial plan can be more efficiently executed than a totally ordered plan because two or more actions can be executed in parallel when they are not related by their relative orders.

In a comparative study on several planning techniques, the most-suited technique for planning an IT-system update was discussed [14]. It was concluded that hierarchical task network (HTN) algorithms, which decompose a high-level action into several finer-grained actions and finally elementary actions by applying decomposition to the actions hierarchically, are the most suitable.

It was also pointed that a general HTN algorithm is insufficient to solve a planning problem concerning a large-scale IT system. In another work, to tailor an HTN algorithm to IT system planning [15], a hybrid approach to a HTN and state-space search and several optimization were proposed [16].

In contrast to those works, in our previous work [8]–[10], we chose state-space search for making it possible to explicitly deal with system states and easily taking global constraints (cf. [12]) into account. State-space search can be adapted to account for global constraints by restricting a state space to states satisfying given global constraints. This feature is difficult to integrate into methods such as those that handle states of IT systems implicitly (like POP and HTN).

As mentioned in Sect. 1, the size of state spaces grows exponentially with increasing number of components in a system. To address this issue, our previous work [8] adopted a divide-and-conquer strategy, namely, dividing systems into strongly connected components by dependencies. By our divide-and-conquer strategy, the main factor of scalability of a state-space search reduces from the size of a whole system to the maximum size of strongly connected components of a system. However, global constraints make related system components depend on each other, and our divide-and-conquer strategy does not work due to these dependencies.

3. Illustration of Fully Declarative System Update

This section illustrates declarative system update. Figure 1 shows our motivating example, that is a situation in which
In this study, each system component is modeled by state models. State models consist of several transition systems with dependencies on their transitions. Here, a transition system consists of states of a system and transitions between them, and represents behaviors of a system with discrete internal status.

System updates are modeled by the planning problems on state models, which is called state model planning. A state model planning is defined by a state model as well as initial and desired states of each transition system of the state model. Figure 2 shows an example modeled by state-model planning.

Each rectangle in the figure represents a transition system, henceforth called a state element to emphasize that it is an element of a whole state model. Each circle in a state element is a state of it, and each arrow between two states is a transition between them. Each double-lined state is an initial state of each state element, and each filled-with-black state is a desired state of each state element. The goal of system update is for all components to transit from an initial state to a desired state.

In Fig. 2, three state elements are shown. “HV.package” has two states, old and new, which represent an old version and a new version of the package of the hypervisor, respectively. “HV.service” has two states, run and stop, which represent running and stopped states of the hypervisor, respectively. “VM” also has two states, run and stop, which represent running and stopped states of the VM, respectively.

In Fig. 2, some transitions have a label with a name of an element and its state. For example, the label “VM:stop” is attached to the transition from run to stop in HV.service. These labels represent dependencies on transitions. A dependency on a transition represents a requirement to use the transition. That is, dependency “d:s” on a transition from src to dst of state element e means that “e’s state can move from src to dst only if component d is at state s.”

We note that system operators do not have to directly represent a state model of their systems. Representing state model is equally or more time-consuming tasks for system operators. We assume that our declarative system update tool uses high-level models to hide details of states, transitions and dependencies. For example, the component model [9], [10] is proposed, which consists of system components, connections between them, and property of them. In this model, the model encapsulates the details of components and connections, and are converted into a detailed model that corresponds to the state model.

3.2 State Model Planning

The goal of state model planning is to discover the shortest sequence of transitions that transfers states of all components to desired states while keeping all requirements on labels fulfilled. A system-update plan is obtained as a solution of state model planning, and system update can be correctly performed by following the system-update plan. For example, one solution of the problem shown in Fig. 2 is given as the following plan:

1. VM: run → stop
2. HV.service: run → stop
3. HV.package: old → new
4. HV.service: stop → run
5. VM: stop → run

(Here, notation e : a → b denotes a transition from a to b in e.) This solution gives a correct system-update plan directly.

A way to formulate state-model planning as a shortest-path problem on a state graph, which is a transition system integrating all system components, was shown by Kuroda and Gokhale [8]. For example, Fig. 3 shows a state graph of our running example. A system-update plan can be found as

 precise, the “VM” component should be named “VM.service”, but it is omitted for brevity.
paths from the initial state (1) to the desired state (5) over this state graph as the following path: \((1) \rightarrow (2) \rightarrow (3) \rightarrow (7) \rightarrow (6) \rightarrow (5)\).

4. Formal Definitions

In this section, state models and state model planning are formalized.

4.1 Notations

The following useful notations are introduced first.

- Let \(I\) be a set. When tuple \(X = (X_i)_{i \in I}\) is an element of \(\prod_{i \in I} X_i\), \(X[i]\) denotes \(X_i\). That is, \(_{[i]}\) is an \(i\)-th projection map.
- Suppose that \(I = \{i_1, \ldots, i_n\}\). Tuple \(X\) such as \(X[i] = X_{i}\) can also be denoted as \(\{i_1 : X_{i_1}, \ldots, i_n : X_{i_n}\}\).
- Sequence of \(x_1, \ldots, x_n\) arranged in this order is denoted as \((x_1, \ldots, x_n)\).
- When \(s = (x_1, \ldots, x_n)\), length of \(s\) is defined as \(n\) and denoted as \(|s|\).

4.2 State Model

System components are modeled as unlabeled transition systems, which called state elements.

**Definition 4.1 (State element)**. A state element \(e\) is defined as a pair \((S, T)\) where \(S\) is called states, and \(T \subseteq S \times S\) is called transitions. \(S\) can be denoted as \(\mathbb{S}(e)\) and \(T\) can be denoted as \(\mathbb{T}(e)\).

State elements of a state model represent behavior of components (i.e., finer-grained parts of systems), and the state model represents the behavior of the whole system that consists of these components. Thus, to formally define a state model, interactions between state elements, called “dependencies”, need to be defined. For this purpose, global states and global transitions of a set of state elements are first introduced.

**Definition 4.2 (Global state and global transition)**. Let \(E\) be a set of state elements.

Global states over \(E\) are defined as a set of all tuples of states over \(E\). Formally, global states \(S_E\) are defined as \(\prod_{e \in E} \mathbb{S}(e)\).

Global transitions over \(E\) are all transitions \(\bigcup_{e \in E} \mathbb{T}(e)\) extended to global states. Formally, let \(e \in E, t \in \mathbb{T}(e)\) and \(\sigma_1, \sigma_2 \in S_E\). \((\sigma_1, t, \sigma_2)\) is a global transition when \(t = (\sigma_1[e], \sigma_2[e])\) holds and \(\sigma_1[e'] = \sigma_2[e']\) holds for each \(e' \in E \setminus \{e\}\).

A set of all global transitions over \(E\) is denoted as \(\mathbb{T}_E\), and \((\sigma_1, t, \sigma_2) \in \mathbb{T}_E\) is denoted as \(L_{\mathbb{T}_E}\).

Intuitively, global states represent states of a whole system, and global transitions represent component-wise transitions of a whole system.

The notion of dependency among state elements can be introduced as follows. As mentioned in Sect. 3.2, dependencies represent restrictions on transitions of state elements, such as “The VM instance can be launched only when the hypervisor is running.”

**Definition 4.3 (Dependency)**. Let \(E\) be a set of state elements.

A dependency over \(E\) is a mapping \(D\) from each transition in \(E\) to a tuple of (non-empty) subsets of states in \(E\):

\[
D : t \mapsto (B_e)_{e \in E} \quad (\text{where } B_e \subseteq \mathbb{S}(e) \land B_e \neq \emptyset)
\]

State element \(e\) is said to be depending to \(e'\) when \(D(t)[e'] \neq \mathbb{S}(e')\) holds for some \(t \in \mathbb{T}(e)\). Dependency from \(e\) to \(e'\) can be denoted as \(e \circ \cdots \cdot e'\). When it is clear from the context, \(D\) can be omitted, and \(e \circ \cdots \cdot e'\) can be simply written.

A dependency maps from each transition to a set of global states that the transition is executable. Formally, A transition \(t\) is said to be executable at a global state \(\sigma\) when \(\sigma[e]\) is in \(D(t)[e]\) for all \(e \in E\).

The following notation is also introduced:

- A set of all dependencies over \(E\) is denoted as \(\mathbb{D}_E\).
- When \(B[e] = \mathbb{S}(e)\) holds, “\(e : \mathbb{S}(e)\)” can be omitted from \([e_1 : B[e_1], e_2 : B[e_2], \ldots, e_n : B[e_n]]\) for simplicity of expression.
- The symbol \(\mathbb{T}\) denotes \([e_1 : \mathbb{S}(e_1), \ldots, e_n : \mathbb{S}(e_n)]\).
- When \(D\) is a dependency and \((s, d)\) is a transition, \(D((s, d))\) can be denoted as \(D(s, d)\).

Then, state models are formally defined as follows. Let \(E\) be a set of state elements and \(D\) be a dependency over \(E\). A state model is defined as pair \(M = (E, D)\).

**Example 4.1.** State elements of the system shown in Fig. 4 are formalized as follows:

- \(HV\text{-package} = (\{(\text{old, new}), (\text{old, new})\})\)
- \(HV\text{-service} = (\{\text{run}_{hv}, \text{stop}_{hv}\})\).
4.4 State Model Planning

State model planning and its solution can now be defined. Formally, state model planning is defined as a planning problem defined for a state graph of a state model from a global initial state to a global desired state.
Therefore, if Dijkstra’s algorithm [18] is used to find a path from an initial state to a desired state, then all over the global states are traversed and the problem will never be solved in a practical time when \( n \) is large (e.g., when \( n > 100 \)). Accordingly, to deal with large systems, a heuristic function tailored for state model planning must be adopted.

5.2 Critical Element Heuristic Function

The proposed heuristic function is based on the following observation. In many cases of system update, a “bottleneck” component occurs in the entire system update. That is, a component has many dependencies to other components directly or indirectly, and resolving its dependencies takes most of the updating time. The element of that bottleneck component is called the critical element. For example, in the system shown in Fig. 2, HV.package is said to be the critical element of the planning problem, because transferring states of all other elements are caused by the update of the HV.package.

The proposed heuristic function can be regarded as one of variants of relaxed planning heuristics [19]. That is, dependencies of a given state model planning are relaxed by extracting dependencies that only relate to the critical element and removing all other dependencies. The length of a solution of the relaxed state model planning under approximates that of the original problem. Thus, this approximate length of a solution can be used as an admissible heuristics of the \( A* \) search algorithm.

The critical element of given state model planning must be identified. The proposed algorithm searches for the critical element by using a brute-force algorithm. Namely, for all elements \( e \), given state model planning is relaxed by \( e \), an approximate length of a solution is computed by solving each relaxed state model planning, and the element that produces maximum approximate length is taken as the critical element.

5.3 Definition of Relaxation of State Model Planning and Critical Element Heuristic Function

First, \( e \)-derived dependency is introduced. Intuitively, \( e \)-derived dependency based on dependency \( D \) is dependency \( D \) restricted on a rooted tree whose root is \( e \), nodes are \( E \), and edges are a part of “\( \circ \rightarrow \)” relations.

**Definition 5.1** (\( e \)-derived dependency). Let \( M = (E, D) \) be a state model. A dependency \( D_1 \) is called an \( e \)-derived dependency based on \( D \) when the following conditions hold (1) - (4) for all \( e \in E \):

1. For each transition \( t \) and each state \( e' \), \( D(t)[e'] \subseteq D_1[t][e'] \) holds.
2. \( e \) is not depended from any other \( e \in E \).
3. Dependency \( D_1 \) does not form a cycle.
4. \( \forall e_1, e_2, e_3 \in E \setminus \{ e \} \).

\[
\begin{align*}
e_2 \circ \cdots \circ e_1 & \land e_3 \circ \cdots \circ e_1 \Rightarrow e_2 = e_3
\end{align*}
\]

Example 5.1. Problem \( \mathcal{P} \) shown in Fig. 6 is used in Sect. 5. Dependency \( D_\mathcal{P} \) of \( \mathcal{P} \) is given as:

\[
\begin{align*}
D_\mathcal{P}(a_1, b_1) &= \{ e_2 : \{ b_2, c_2 \}, e_4 : \{ b_4 \} \} \\
D_\mathcal{P}(b_2, a_2) &= \{ e_3 : \{ b_3 \} \} \\
D_\mathcal{P}(a_4, b_4) &= \{ e_2 : \{ a_2 \}, e_3 : \{ b_3 \} \} \\
D_\mathcal{P}(\cdot, \cdot) &= \top \quad \text{(otherwise)}
\end{align*}
\]

The following two \( e_1 \)-derived dependencies are introduced as

1. If all dependencies except for direct dependencies are removed from \( e_1 \), the following \( e_1 \)-derived dependencies \( D_\mathcal{P}^{e_1}_{\text{direct}} \) are obtained:

\[
\begin{align*}
D_\mathcal{P}^{e_1}_{\text{direct}}(a_1, b_1) &= \{ e_2 : \{ b_2, c_2 \}, e_4 : \{ b_4 \} \} \\
D_\mathcal{P}^{e_1}_{\text{direct}}(\cdot, \cdot) &= \top \quad \text{(otherwise)}
\end{align*}
\]

2. If all dependencies except for those along with an \( e_1 \)-rooted tree of dependencies shown in Fig. 7 are removed, the following \( e_1 \)-derived dependencies \( D_\mathcal{P}^{e_1}_{\text{derived}} \) are obtained:

\[
\begin{align*}
D_\mathcal{P}^{e_1}_{\text{derived}}(a_3, b_1) &= \{ e_2 : \{ b_2, c_2 \}, e_4 : \{ b_4 \} \} \\
D_\mathcal{P}^{e_1}_{\text{derived}}(b_2, a_2) &= \{ e_3 : \{ b_3 \} \} \\
D_\mathcal{P}^{e_1}_{\text{derived}}(\cdot, \cdot) &= \top \quad \text{(otherwise)}
\end{align*}
\]

relaxed state model and relaxed state-model planning can thus be defined on the basis of an \( e \)-derived dependency.

**Definition 5.2** (Relaxed state model \( M|\mathcal{P}| \) and relaxed state-model planning \( \mathcal{P}|\mathcal{P}| \)). Let \( M = (E, D) \) be a state model, \( e \in E \), \( D_\mathcal{P} \) be an \( e \)-derived dependency, \( \mathcal{P} = (M, \sigma_0, \sigma_f) \) be a state model planning.

\[
M|\mathcal{P}| = (E, D) \quad \text{is called an} \quad D_\mathcal{P} \text{-relaxed state model of} \quad M \quad \text{and} \quad \mathcal{P}|\mathcal{P}| = (M|\mathcal{P}|, \sigma_0, \sigma_f) \quad \text{an} \quad D_\mathcal{P} \text{-relaxed state model planning of} \quad \mathcal{P}.
\]

Problem \( \mathcal{P}|\mathcal{P}| \) is said to be a relaxed problem of \( \mathcal{P} \) in the sense that a graph \( G(M) \) is a subgraph of \( G(M|\mathcal{P}|) \).

The problem \( \mathcal{P}|\mathcal{P}| \) takes only dependencies derived from \( e \) into account. Accordingly, a critical element and
critical element heuristics can be formally defined as follows.

The following notation is introduced for brevity. Let \( P = (M, \sigma_0, \sigma_f) \) be state model planning and \( \sigma \) be a global state of \( M \). \( P@\sigma \) is written for state-model planning \((M, \sigma, \sigma_f)\).

**Definition 5.3.** Critical-element heuristic function \( \theta_P(\sigma) : S \to \mathbb{N} \) is defined as

\[
\theta_P(\sigma) = \max_{e \in E} L(P|\sigma')
\]

Element \( e \) arg \( L(P|\sigma') \) is called a critical element of \( P \). (Subscript \( P \) is omitted from \( \theta_P \) and \( \theta \) is simply written when it is clear from context.)

The following theorem shows that the function \( \theta \) is consistent. Consistency is important property for heuristic functions in the following two points; (1) a consistent function is also an admissible function, which guarantees A* algorithm calculates a collect shortest path, and (2) values of a consistent function are invariant through A* algorithm. We can optimize A* algorithm using this property.

**Theorem 5.1.** Let \( P \) be state-model planning. Function \( \theta_P \) is consistent, namely,

\[
\theta_P(\sigma) \leq \theta_P(\sigma') + 1
\]

holds for all global states \( \sigma, \sigma' \) such as \( \sigma \to \sigma' \).

**Proof.** Let \( e_c, e_c' \) be critical elements of \( P@\sigma, P@\sigma' \) respectively. When no path from \( \sigma' \to \sigma_f \) exists on \( P|\sigma', \theta_P(\sigma') = \infty \) and Theorem 5.1 holds. Therefore, the case that a path from \( \sigma' \) to \( \sigma_f \) exists on \( P|\sigma' \) is considered.

The shortest path \( \pi \) on \( P|\sigma' \) from \( \sigma' \to \sigma_f \) is chosen. Because transition \( t \) such that \( t_{e_c} = (\sigma, t, \sigma') \) exists, path \( \pi' \) exists on \( P|\sigma' \) such that \( t_{e_c}, t_{e_c'}, \ldots, t_{e_c} \). Because \( \theta(\sigma) \) is

\[
\theta(\sigma) \leq |\pi'| = |\pi| + 1
\]

Because \( e_c \) is a critical element of \( P|\sigma', \) the following holds:

\[
L(P|\sigma'@\sigma') \leq L(P|\sigma'@\sigma')
\]

\[
|\pi'| \leq \theta(\sigma')
\]

By (1) and (3), \( \theta(\sigma) \leq \theta(\sigma') + 1 \) holds.

The consistency property makes the proposed algorithm more efficient by eliminating recalculation of the value of the proposed heuristic function. The detail of optimization of A* search algorithm is discussed below.

5.4 Algorithm for Computing \( \theta(\sigma) \)

An algorithm for computing \( L(P|\sigma'@\sigma) \) for calculating \( \theta(\sigma) \) is proposed in the following. We fix e-deriving dependencies for each \( e \) in the rest of this section, and for brevity, \( P|\epsilon \) is abbreviated as \( P|\epsilon \).

We note that the number of states of each state element is assumed to be enough small (e.g., 2~5) to finish the pre-process explained in the following subsection in negligibly short time to whole planning process.

5.4.1 Preprocess

First, the proposed algorithm requires simple e-local paths of each element \( e \). Here, an e-local path is a sequence of transition: \((s_0, d_0), (s_1, d_1), \ldots, (s_k, l_k) \in \mathbb{E}(e)\) such that \( d_{i-1} = s_i \) holds for all \( i \), and a local path is said to be simple when it visits each state at most once. Second, the algorithm requires the length of the shortest e-local paths of each element \( e \). Note that since shortest local paths are simple, these values can easily be obtained after all simple e-local paths of each element \( e \) is obtained.

Before A* search algorithm is executed, the following preprocesses are executed. All simple e-local paths of each state pair \((s, s')\) in each element \( e \) is calculated, and stored in an array \( sp_e \). Then the shortest path of each state pair \((s, s')\) is obtained by choosing the shortest path from \( sp_e[s, s'] \), and its length in an array \( dist_e \).

**Example 5.2.** Let \( P|\epsilon \) be \( P|\epsilon|_{p \epsilon \epsilon} \). After the preprocess is applied to \( P|\epsilon \), the following prerequisite data is obtained. \((sp_e[s, s] = \emptyset) \) and \( dist_e[s, s] = 0 \) are omitted for all \( e \).

- \( e_1 \) (1 = 1 , 3, 4):

\[
sp_{e_1}[s, s'] = \{(s, s')\}, dist_{e_1}[s, s'] = 1
\]

(for all \( s, s' \) s.t. \( s \neq s' \))

- \( e_2 \):

\[
\begin{align*}
sp_{e_2}[b_2, a_2] &= \{(b_2, a_2), (b_2, c_2), (c_2, a_2)\}, \\
sp_{e_2}[a_2, c_2] &= \{(a_2, b_2), (b_2, c_2)\}, \\
sp_{e_2}[c_2, b_2] &= \{(c_2, a_2), (a_2, b_2)\}, \\
sp_{e_2}[s, s'] &= \{(s, s')\} \quad \text{(for all other} \ s, s' \text{ s.t.} \ s \neq s'\}
\end{align*}
\]

\[
\begin{align*}
dist_{e_2}[a_2, c_2] &= dist_{e_2}[c_2, b_2] = 2 \\
dist_{e_2}[s, s'] &= 1 \quad \text{(for all other} \ s, s' \text{ s.t.} \ s \neq s'\)
\end{align*}
\]

5.4.2 Main Algorithm

The value of \( \theta_P(\sigma) \) is the maximum value of \( L(P|\epsilon@\sigma) \) for all states in \( P \). In the proposed algorithm for calculating the value of \( \theta_P(\sigma) \), all of the values of \( L(P|\epsilon@\sigma) \) are calculated by the following algorithm \( calc(P, e, \sigma) \).

Function \( costOn(P, e, R) \) calculates the minimum cost of transferring states of \( e \) and its dependent elements from initial states to desired states under a requirement \( R \) on \( e \). Here, a requirement on an element \( e \) is a sequence of a set of states of \( e \). Under requirement \( \{R_1, \ldots, R_m\} \), \( e \) needs to visit one required state in each \( R_i \) in the order of \( R_1, \ldots, R_m \).
Algorithm 1: \(\text{calc}(\mathcal{P}, e, \sigma)\): Calculate \(L(\mathcal{P}|_{e} @ \sigma)\)

1: for \(e \in E\) do
2: visit[\(e\)] \(\leftarrow\) false
3: end for
4: \(L \leftarrow \text{costOn}(\mathcal{P}|_{e}, e, \sigma, (1))\)
5: for \(e \in E\) do
6: if not visit[\(e\)] then
7: \(L \leftarrow L + \text{dist}(e[\sigma[e]], e[f[e]])\)
8: end if
9: end for
10: return \(L\)

Algorithm 2: \(\text{costOn}(\mathcal{P}, e, \sigma, (R_0, R_1, \ldots, R_m))\)

1: visit[\(e\)] \(\leftarrow\) true
2: \(\text{spc} \leftarrow \infty\)
3: for \((s_0, s_1, \ldots, s_m) \in \text{trace}((R_0, \ldots, R_m))\) do
4: for \(\pi \in \text{pathThrough}(\sigma[0][e], s_0, s_1, \ldots, s_m, \sigma[f[e]])\) do
5: \(\text{spc}' \leftarrow \infty\)
6: for \(\text{dep} \in \{e' \in e : \pi = e' \sigma\} \) do
7: \(\text{req} \leftarrow \text{pathReqs}(\mathcal{P}, \text{dep}, \pi)\)
8: \(\text{spc}' \leftarrow \text{spc}' + \text{costOn}(\mathcal{P}, \text{dep}, \pi, \text{req})\)
9: end for
10: if \(\text{spc} > \text{spc}'\) then \(\text{spc} \leftarrow \text{spc}'\)
11: end if
12: end for
13: end for
14: return \(\text{spc}\)

The value of \(\text{costOn}(e_4, (\{b_4\}))\) is calculated immediately:
\[\text{costOn}(e_4, (\{b_4\})) = \|((a_4, b_4), (b_4, a_4))\| = 2.\]

In the calculation of the value of \(\text{costOn}(e_2, (\{b_2, c_2\}))\), The cost of paths \(((a_1, b_1), (b_1, c_1), (c_1, a_1))\) and \(((a_1, b_1), (b_1, a_1))\) are compared in line 10, and returns the smaller as the value of \(\text{costOn}(e_2, (\{b_2, c_2\}))\).

\[\text{costOn}(e_2, (\{b_2, c_2\})) = \min[\|((a_1, b_1), (b_1, c_1), (c_1, a_1))\|, \|((a_1, b_1), (b_1, a_1))\|] = \min[3, \|((a_3, b_3), (b_3, a_3))\| + 2] = \min[3, 2 + 2] = 3.\]

Finally, the value of \(\text{costOn}(e_1, (\emptyset))\) is calculated as:
\[\text{costOn}(e_1, (\emptyset)) = \text{costOn}(e_4, (\{b_4\})) + \text{costOn}(e_2, (\{b_2, c_2\})) + \|((a_1, b_1))\| = 2 + 3 + 1 = 6.\]

In the above execution process, each visit[\(e\)] is set to \textit{true} for all \(e\). Thus, \(\text{calc}(\mathcal{P}, e_1, \sigma_0)\) returns 6, which is the return value of \(\text{costOn}(\mathcal{P}|_{e_1} e_1, \sigma_0, (\emptyset))\).

5.5 Optimize A* Search with Critical Element Heuristics

The A* search algorithm was proposed by Hart et al. [17] and is described as \textbf{Algorithm 3}. Here, the detail of backtracking procedure for picking up the found path is omitted for brevity.

\textbf{Algorithm 3}: \(A^*(G = (S, T), s_0, s_f, h)\)’s: Search the shortest path on the graph \(G = (S, T)\) from \(s_0\) to \(s_f\) with the heuristics \(h\).

1: \(\text{cost}[s_0] \leftarrow h(s_0); \text{Open}.\text{add}(s_0)\)
2: while \(\text{Open} \neq \emptyset\) do
3: \(s_{\text{min}} \leftarrow \arg \min\{f(s) : s_{\text{Open}}\}\)
4: if \(s_{\text{min}} = s_f\) then return a shortest path obtained by backtracking.
5: \(\text{Open}.\text{remove}(s_{\text{min}}); \text{Closed}.\text{add}(s_{\text{min}})\)
6: for \(n: \text{all neighbor nodes of } s\) do
7: \(f(n) \leftarrow \text{cost}[s_{\text{min}}] - h(s_{\text{min}}) + h(n) + 1\)
8: \(\text{if } (n \notin \text{Open} \land n \notin \text{Closed})\) then
9: \(\text{if } \forall (n \in \text{Open} \land f(n) < \text{cost}[n])\) then
10: \(\text{cost}[n] \leftarrow f(n); \text{Open}.\text{add}(n)\)
11: end if
12: end while

Additionally, we can adopt the following optimization techniques of A* search algorithm and \(\theta\):

1) According to Theorem 5.1, the value \(f(s)\) in \textbf{Algorithm 3} is not updated. Therefore, the calculation in lines 7 – 9 of \textbf{Algorithm 3} can be omitted when \(n \in \text{Closed}\).
2) A tie breaking technique can be adopted. That is, \(s_{\text{min}}\) can be freely chosen from nodes whose cost is minimum in all nodes in \text{Open} in line 3 of \textbf{Algorithm 3}. The proposed algorithm choose the farthest node to \(s_0\).

Example 5.3. Let \(\mathcal{P}_e|_{\tau} = (M, \sigma_0, \sigma_f)\) be \(\mathcal{P}_e|_{\tau}^\text{devised}\). First, an execution process of \(\text{costOn}(\mathcal{P}_e|_{\tau} e_1, \sigma_0, (\emptyset))\) is shown below. In the following explanation, for simplicity, we omit common arguments \(\mathcal{P}_e|_{\tau}, e_1, \sigma_0, (\emptyset)\) and simply write \(\text{costOn}(e, \pi)\).

First, \(\text{costOn}(e_1, (\emptyset)) = \text{costOn}(e_4, (\{b_4\})) + \text{costOn}(e_2, (\{b_2, c_2\})) + \|((a_1, b_1))\|\)
3) When \( \forall t, e . D^e(t)[e] \subseteq D^f(t)[e] \) holds, we can remove \( e_1 \) from the candidates of critical elements. Additionally, when \( \sigma[e_1] = \sigma_f[e_1] \) and \( \forall t \notin \Xi(e_1) . D^e(t)[e] \subseteq D^f(t)[e] \) hold for all \( e \), we have \( L(P)|\nu^\sigma@\sigma| \leq L(P)|\nu^\sigma|\sigma| \) and have only to calculate the value of \( L(P)|\nu^\sigma|\sigma| \).

4) A memoization technique can be adopted. Because \texttt{pathReqs} is often called many times with the same arguments, the result of each \texttt{pathReqs} call is stored and reused.

5) Let \( P = (M, \sigma_0, \sigma_f) \) be a state model planning. When we regard a state model \( M \) as a graph with nodes of state elements and edges of dependencies, the graph of \( M \) is sometimes divided into some connected components \( C_1, C_2, \ldots, C_k \) and we can construct an another state model planning \( P_1 \) by restricting \( P \) over state elements of each \( C_i \). Then, we can use more (or equal) accurate heuristics than \( \vartheta \). That is, we adopt sum of values of \( \vartheta_{P_i}(\sigma) \) for each \( P_i \) instead of the value of \( \vartheta_P(\sigma) \).

Finally, the optimized \textit{Algorithm 3} provides a system-update plan, \( P = (M, \sigma_0, \sigma_f) \), as the return value of \( \text{A}^*(M, \sigma_0, \sigma_f, \vartheta_{P_i}) \).

6. Performance Evaluation

We applied the proposed algorithm to solve system update plan of SDN/NFV systems. We start with very basic examples to safely upgrade running hypervisor without risking VMs, or virtual network functions, running on top of the hypervisor. And we also tested the algorithm for a system that needs rolling update to avoid failure incidents and zero downtime update for high availability.

We implemented our heuristic function \( \vartheta \) with \( D^f \) derived in Scala and evaluated performance of our planning algorithm that uses critical element heuristics. Experiments were conducted on a machine with Intel(R) Xeon(R) E5-1620 0 (3.60GHz, 32GB memory) with timeout of 1 hour. We use scala runtime options “-J-Xms16G -J-Xmx16G” in performing all experiments.

6.1 Upgrade of Running Hypervisor

As we mentioned in the prior sections, the problem shown in Fig.1, which is to upgrade a hypervisor, is a difficult problem for declarative system update because this induces search in state space of exponential size. Our first experiment is performed on the problem shown in Fig.1, which includes a hypervisor and \( n \) VMs.

Figure 8 depicts the state model planning \( \text{HV-VM}(n) \), which is a state model representing the system of Fig.1. The state model planning \( \text{HV-VM}(n) \) is similar to our motivating example shown in Fig.2 but has \( n(> 1) \) VMs. The current state of \( \text{HV-VM}(n) \) is that the hypervisor of the old version package is hosting \( n \) VMs and the desired state is that the hypervisor is hosting \( n \) VMs similarly but the package of the hypervisor is updated.

6.2 Rolling Update

As we mentioned in Sect. 1, in system updates, operators are often urged to design workflows so that the system update raise no fatal incidents of running services. If a system consists of two more VMs installed the same function, operators can adopt \textit{rolling update strategy} to update these VMs with no downtime. Namely, in rolling update strategy, operators will not update all VMs simultaneously, but only \textit{subset} of VMs at a time.

Figure 9 shows a load balanced cluster, which is an example of system that we can update each VMs by rolling update. \( VM(1), VM(2), \ldots, \) and \( VM(n) \) are application servers where the same application is installed. The load balancer distributes attached VMs and, in most cases, all VMs are necessarily needed to keep a service in operation. Therefore, if the enough number of VMs is in operation, the service on Fig.9 keeps alive even in system update.

Figure 10 shows the state model planning \( \text{Rolling}(n) \), which represents system update in Fig.9 with \( n \) VMs. The load balancer is not appeared in \( \text{Rolling}(n) \) because the state of the load balancer doesn’t affect the update and apparently keeps in the “running” state. Here, we note that Fig.10 doesn’t take the condition for keeping service running into account. In order to obtain rolling update plans, we need to add a kind of global condition to the planning problem \( \text{Rolling}(n) \) and solve the problem with the additional condition. We add the condition that “at least \( n(> 0) \) VMs are running” to the problem \( \text{Rolling}(n) \). Our planner can easily deal with such an additional conditions by excluding global states violating the condition.
6.3 Zero Downtime Updates of ToR Switches in a Data Center

We consider another example such that uses standby components for update. As we mentioned in Sect. 1, declarative system update tools use extra resources and standby components when system update cannot be completed in main systems. Let us suppose a case of duplicated ToR switches in a data center. Important network appliances, including ToR switches, are often duplicated for high availability. Even when main switches fall down, standby switches activate as a substitute for the main switch and keep servers mounted on the rack accessible. Additionally, even in firmware update of ToR servers with requiring system reboot, the update can be completed with no downtime by detouring traffic to standby switches.

Figure 11 shows the latter case. There are $n$ racks with eight servers and two ToR switches. In each rack, one ToR switches is an active switch, and other is a standby switch. Now, the administrator plans firmware update of active switches with no downtime.

Figure 12 shows a part of state model planning of Fig.11, which represents just one pair of active-standby switch and contains state elements that show a firmware of active switch (SW[i].main.firmware), services of active and standby switch (SW[i].main and SW[i].sub), routing (SW[i].routeBy), connection to core switch (SW[i].connection), and connection to each server (SW[i,1].connection, ..., SW[i,8].connection). The state model planning UpdToR($n$) is defined by $n$ switch pairs SW[1],SW[2]...,SW[n]. Similarly to Rolling($n$), we need to add global condition that “all switches keep connection among connecting components”. That is, all state elements of “.connection” keep in the state “on”.

6.4 Results

The line graph shows the results of experiments to solve HV-VM($n$) by Dijkstra algorithm and our method. Fig. 19 shows the results of experiments using Rolling($n$) with the condition that “at least 1 VMs are running”. The horizontal line shows $n$ and the vertical one shows elapsed time in seconds to find a solution of each problem. The line of “no heuristic” shows results of Dijkstra algorithm and the line of “our method” shows results of A* algorithm with critical-element heuristic function.

The line graph shows that the computing time of state space search using Dijkstra algorithm drastically increases and exceeds an hour with $n = 24$ on one hand. On the other hand, our method can calculate a plan even with $n = 100$ in 5 minutes.

Figure 13 shows a plan for HV-VM(3) and the corresponding workflow. The plan correctly avoid accidental system stop by stopping the underlying hypervisor.

The results of experiments using Rolling($n$) shows us striking difference of efficiency between two algorithm.

The line graph shows the computing time of state
The problem. We note that because of the HV-VM package because it follows the global initial states. That is, our heuristics estimate the precise value of shortest paths of these inputs by calculating the value of global initial states. In the above experiments, our heuristic function can estimate the corresponding workflow. From beginning step 1 to finishing step 10, VM(1) keeps in operation. Therefore, while performing this workflow, the load balancer can distribute workload to at least one of VM(1) and VM(2) and the system of Fig. 9 keeps alive even during the update.

The line graph Fig. 20 also shows that notable difference between Dijkstra algorithm and our heuristic search. The computing time of state space search using Dijkstra algorithm exceeds an hour with $n = 10$ whereas our method can calculate a plan even with $n = 100$ in 10 minutes.

Figure 14 shows a plan for Rolling(2) and the corresponding workflow. From beginning step 1 to finishing step 5, VM(2) keeps in operation and from beginning step 6 to finishing step 10, VM(1) keeps in operation. Therefore, while performing this workflow, the load balancer can distribute workload to at least one of VM(1) and VM(2) and the system of Fig. 9 keeps alive even during the update.

The line graph Fig. 20 also shows that notable difference between Dijkstra algorithm and our heuristic search. The computing time of state space search using Dijkstra algorithm exceeds an hour with $n = 8$ on one hand. On the other hand, our method can calculate a plan even with $n = 60$.

Figure 15 shows a plan for UpdToR(1) and the corresponding workflow. From beginning step 1 to finishing step 7, core switch and servers in the rack 1 keep connected. Therefore, while performing this workflow, the rack 1 in Fig. 11 keeps in operation.

### 6.5 Evaluation of Estimated Path Length by $\tilde{\theta}$

In the above experiments, our heuristic function can estimate precise value of shortest paths of these inputs by calculating its value of global initial states. That is, our heuristics estimate length of solution of HV-VM(n) as $2n + 3$, Rolling(n) as $5n$, and UpdToR(n) as $7n$ at global initial states of each problem. We note that because Rolling(n) (or UpdToR(n)) can be separated into sub-problems $P_i$ corresponding to each VM(i) (or SW[i]), as we mentioned in Sect. 5.5, we use $\sum_i \tilde{\theta}(\sigma_i)$ instead of $\tilde{\theta}_{\text{Rolling}(n)}(\sigma)$ (or $\tilde{\theta}_{\text{UpdToR}(n)}(\sigma)$).

Figure 16 illustrates computation of $\tilde{\theta}$ by showing the value of $L(HV-VM(2)|D^{\text{derived}}_\text{e})$ and e-rooted trees of $D^{\text{derived}}_\text{e}$ for each element e in HV-VM(2). The critical element of HV-VM(2) is HV.package because it follows the maximum value of $L(HV-VM(2)|D^{\text{derived}}_\text{e})$.

In contrast, we introduce an example problem $P_e$ that $\tilde{\theta}_{P_e}(\sigma_0) \neq L(P_e)$. Any relaxed $P_e$ have strictly shorter solution than the original $P_e$. Because any derived dependencies cannot depend on both $e_4, b_4$ and $e_4, c_4$, solutions of any relaxed problems don’t pass through $e_4, b_4$ and
We took upgrading running hypervisor and rolling update in models are finely enough grained, the proposed algorithm generates system-update plans in sufficiently practical time. We took upgrading running hypervisor and rolling update of running VMs as examples of system update, and showed that our proposed algorithm delivered the update plan within several minutes for a system with 100 VMs, whereas the conventional algorithm is only applicable for very small systems.

Acknowledgments

This work was partly supported by the Ministry of Internal Affairs and Communications, Japan.

References


Takuya Kuwahara received his master's degree of information science and technology from Graduate School of Information Science and Technology, The University of Tokyo in 2015 and has been engaged in research on formal methods for program verification. He joined NEC Corporation in 2015. Now he is working on researches for automation technology for ICT system design and operation.

Takayuki Kuroda received M.E. and Ph.D. degrees from the Graduate School of Information Science, Tohoku University, Sendai, Japan in 2006 and 2009. He joined NEC Corporation in 2009 and has been engaged in research on model-based system management for Cloud applications and Software-Defined networks. As a visiting scholar in the Electrical Engineering and Computer Science department at the Vanderbilt University at Nashville, he studied declarative workflow generation for ICT system update. Now he is working on researched for automation technologies for system design, optimization and operation.

Manabu Nakanoya received his B. Engineering degree from Keio University in 2006. He joined NEC Corporation in 2006 and has been engaged in system integration of governmental ICT system. He now works in NEC's System Platform Research Laboratories and is working on researches for automation technologies for ICT system design, optimization, and operation.

Yutaka Yakuwa received his M. Engineering degree from Waseda University in 2009. He joined NEC Corporation in 2009 and has been undertaking research on formal methods. He now works in NEC's System Platform Research Laboratories and is engaged in research on software-based automation technologies for ICT system operation and management.

Hideyuki Shimonishi received M.E. and Ph.D. degrees from the Graduate School of Engineering Science, Osaka University, Osaka, Japan, in 1996 and 2002. He joined NEC Corporation in 1996 and has been engaged in research on traffic management in high-speed networks, switch and router architectures, and traffic control protocols. As a visiting scholar in the Computer Science Department at the University of California at Los Angeles, he studied next-generation transport protocols. Since then, he engaged in researches on networking technologies including SDN, OpenFlow and NFV for carrier, data center and enterprise networks from their early stages. Now he is working on researches for IoT system platform and automation technologies for system design, optimization, and operation.