Experience Track

# Klotski: Efficient and Safe Network Migration of Large Production Datacenters

Yihao Zhao*
Peking University

Xiaoxiang Zhang*
Meta

Hang Zhu
Johns Hopkins University

Ying Zhang
Meta

Zhaodong Wang
Meta

Yuandong Tian
Meta AI

Alex Nikulkov
Meta

Joao Ferreira
Meta

Xuanzhe Liu
Peking University

Xin Jin
Peking University

## ABSTRACT

This paper presents the design, implementation, evaluation, and deployment of Meta's production network migration system. We first introduce the network migration problem for large-scale production datacenter networks (DCNs). A network migration task at Meta touches as many as hundreds of switches and tens of thousands of circuits per datacenter (DC), and involves physical deployment work on site that can last months. We describe real-world migration challenges, covering complex and evolving DCN architectures and operational constraints. We mathematically formalize the problem of generating efficient and safe migration plans, and exploit the inherent symmetry and locality of DCN topologies to prune the search space. We design an ordering-agnostic compact topology representation to eliminate redundant satisfiability checking, and apply the A* algorithm with a domain-specific priority function to find the *optimal* plan. Evaluation results on a range of production migration cases show that Klotski reduces the time to find optimal migration plans by up to 381× compared to prior solutions. We hope by introducing the problem and sharing our deployment experience, this work can provide a useful context for network migration in the real world and inspire future research.

## CCS CONCEPTS

• **Networks** → **Network management**; *Network design principles*.

## KEYWORDS

Datacenter network, network migration

*Yihao Zhao and Xiaoxiang Zhang contributed euqally.


*ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA*

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604818

**ACM Reference Format:**
Yihao Zhao, Xiaoxiang Zhang, Hang Zhu, Ying Zhang, Zhaodong Wang, Yuandong Tian, Alex Nikulkov, Joao Ferreira, Xuanzhe Liu, and Xin Jin. 2023. Klotski: Efficient and Safe Network Migration of Large Production Datacenters. In *ACM SIGCOMM 2023 Conference (SIGCOMM '23), September 10–14, 2022, New York City, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3603269.3604818

## 1 INTRODUCTION

Large web service providers heavily depend on datacenter networks (DCNs) to deliver reliable and smooth experience to users. DCNs are constantly scaling and evolving because of at least three reasons. First, the datacenter (DC) traffic is increasing at a fast pace under new technology trends, exemplified by recent advancements in cloud computing [40, 47], big data [12, 28], and machine learning [10, 51]. Second, switch failures are the norm rather than the exception in large-scale multi-layer DCNs [4, 14]. A failure recovery process usually requires replacing failed switches or updating their firmware or software. Third, there are routine procedures to retire old hardware and onboard new hardware [48, 53].

The scaling and evolution of DCNs rely on network migration, which changes the network by adding, removing, or swapping switches and circuits. Migrating a DCN is time-consuming, especially for large-scale multi-layer production DCNs. Importantly, unlike updating the control plane configurations or data plane flow tables, network migration involves *physical* deployment work—a *manual* process performed by human operators. A migration task at Meta can touch hundreds of switches and tens of thousands of circuits per DC, and can take months based on the scale and complexity (§2).

The objective for network operators is to perform network migrations *efficiently* and *safely*. First, it is important to minimize the migration time, because network migration is often triggered by important operational needs, such as increasing the total capacity, recovering from failures, and onboarding switches with new functionalities. These operational needs are strongly correlated with application demands and user experience. Second, safety must be guaranteed during the migration. That is, the network, when old switches have been removed and new switches are not onboarded, must satisfy dynamic traffic demands during the migration and leave sufficient headroom to absorb traffic bursts from flash crowds.

Efficiency and safety are difficult to achieve under various constraints in production settings. First, because we cannot precisely drain all circuits at the same time, severe congestion can happen in the transient, which is known as traffic funneling. Second, multiple datacenters may be migrated at the same period, and independently migrating each can make them unconnected during some middle steps. Third, there can be different meshing patterns between the old topology and new topology, increasing the complexity of the migration. Forth, multiple generations of DCNs can coexist. Migrating them would need different plans and migrating multiple of them together requires even more careful planning.

Given these constraints, it is challenging to find the optimal plan that minimizes migration time while ensuring safety. A migration plan contains a sequence of actions that drain or undrain traffic from particular switches and circuits. Finding the optimal plan amounts to finding the optimal *ordering* of actions, which is a hard combinatorial optimization problem. For example, changing 100 switches has more than 100! (or $9.3 \times 10^{157}$) plans, while large DCNs have tens of thousands of switches.

We present Klotski, a system for efficient and safe DCN migration. Klotski is able to find the optimal migration plan under safety constraints for large DCNs with $O(10,000)$ switches and $O(100,000)$ circuits in just *a few minutes*.

To address the scalability challenges, Klotski first exploits the inherent *symmetry* of DCN topologies to prune the search space. It divides a DCN topology into multiple symmetry blocks, and different orderings of actions in the same block have *equivalent* search states. We remark that leveraging the topology symmetry has been applied to different contexts of network management [4, 32]. In particular, Janus [4] is a recent work that applies symmetry to planning network changes. For complex DCN topologies, however, one symmetry block contains few switches, which cannot prune the search space effectively. Klotski further considers the switch locality. Based on our observation that neighbor switches can be operated in parallel with little operational cost and little impact on safety, we merge several symmetry blocks into one operation block, in which the switches are operated together.

Klotski applies A* algorithm [18] to efficiently find the optimal migration plan in the pruned search space. A* algorithm is a canonical informed search algorithm that uses a priority function to give preferences to the states that are likely to be close to the target state. The crux of our solution is a domain-specific priority function tailored to network migration. The function is a composition of the existing cost from the original state to the current state and the estimation of the lowest future cost from the current state to the target state. It enables Klotski to quickly find the target, i.e. efficiency, and guarantees that the target is optimal, i.e. optimality.

The dominant part of the search is checking the satisfiability of the safety constraints for the visited states. Satisfiability checking is time-consuming because it examines the traffic demands and port constraints on the DCN topology that can contain as many as $O(10,000)$ switches and $O(100,000)$ circuits. We observe that many states are *equivalent* in terms of constraint satisfiability as it is only concerned with the intermediate network topology. A natural idea is to store the satisfiability of each visited intermediate topology in a table, obviating the need for redundant satisfiability checking. But naively doing so incurs excessive indexing overhead and memory footprint due to the scale of large DCN topologies.

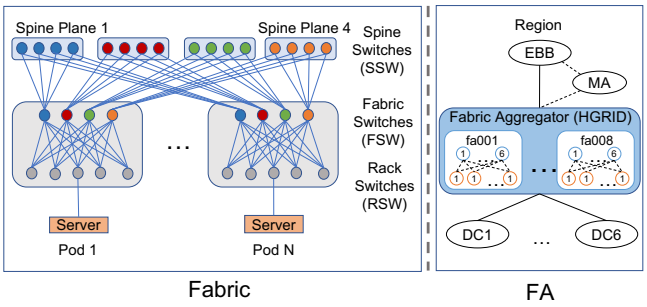



**Figure 1: Datacenter network architecture at Meta.**

Our important intuition is that the ordering of the actions does *not* matter for satisfiability checking, because satisfiability checking only examines the topology after these actions are performed. Based on this intuition, we design a *compact* topology representation that uses a vector to only store the number of finished actions of each type. This *ordering-agnostic* representation enables Klotski to efficiently store intermediate satisfiability checking results and significantly reduce the computation overhead and memory footprint.

From an engineering perspective, we implement an end-to-end pipeline for managing the entire life cycle of network migration. Klotski is productionized as a component of Engineering Design Package at Meta. We develop a Network Product Definition (NPD) format that represents production DCNs as input to Klotski. It also provides an interface to accommodate different cost functions. We evaluate Klotski with a variety of production DCN topologies and network migration cases. Klotski reduces the planning time by up to 381× compared to state-of-the-art DCN migration planners [4, 37].

We have deployed Klotski at Meta since 2020 to generate migration plans for over 100 datacenters within 20 regions. We share our deployment experience from three aspects. First, we share the need to consider special routing configurations and incorporate traffic demand forecasts into the migration process. Failing to do so would result in congestion during the migration. Second, we share operational issues we have to deal with when applying Klotski in production, including failures during operation duration, simultaneous operations, space and power constraints, unexpected traffic surges, and operating expense (OPEX) savings. Finally, there is a growing interest in applying deep learning (DL) to networking in both academia and industry. We started in early 2021 and collaborated with top DL researchers and engineers at Meta for over two years. We share what we have tried, contrast classical methods and DL methods with first-hand experience, and inform the community of the practical obstacles we found when trying to apply DL to a real-world network management problem.

## 2 NETWORK MIGRATION AT META

In this section, we first introduce the DCN architecture at Meta. Then we describe the network migration operations and discuss the practical challenges we face. We lastly summarize network migration types at Meta.

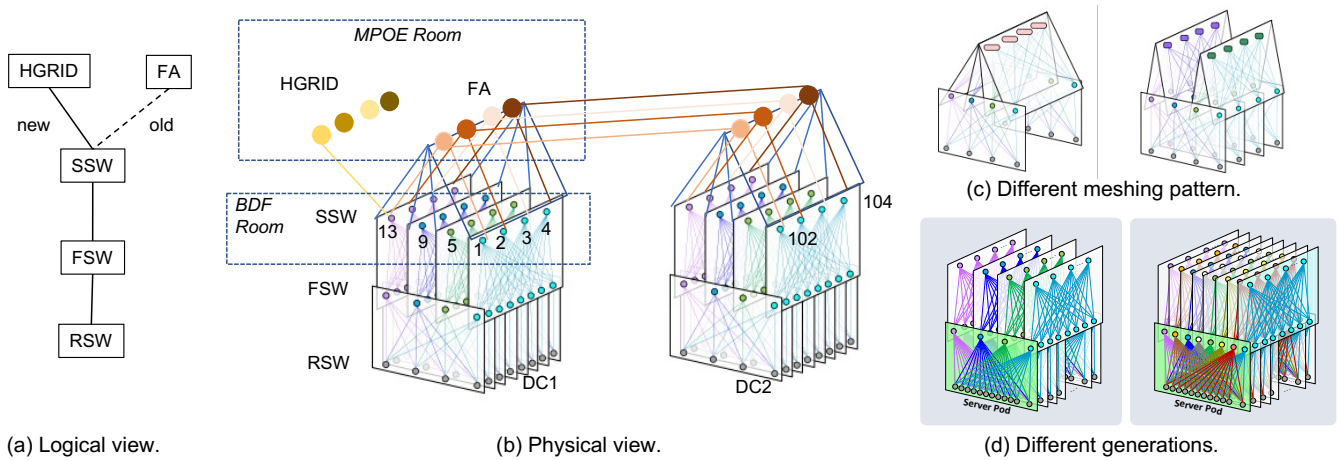



**Figure 2: Motivating examples for network migration.**

## 2.1 DCN Architecture at Meta

DCs at Meta are the most important infrastructure to provide compute and storage capacity to support the applications and services for billions of users. A *region* refers to DC facilities in geographically-closed proximity, e.g., in a campus, which usually consists of six to seven buildings. In the following, we introduce the important switch roles in a bottom-up order in the DCN topology, as shown in Figure 1.

**Datacenter fabric.** A *Fabric* network provides mass connectivity for servers with three layers of switches. A rack of servers is connected to a Rack Switch (**RSW**). RSWs are interconnected by Fabric Switches (**FSWs**) in the upper level, which in turn are connected by Spine Switches (**SSWs**). The smallest unit of deployment in a Fabric is called a *Pod*. One Pod consists of four FSWs and the connected RSWs. Besides, a disjoint end-to-end path within the Fabric serviced by a set of SSWs and FSWs is called a *Plane*. Switches in the same plane are shown in the same color in Figure 1.

**Fabric aggregation layer.** Multiple Fabrics in the same region are connected by a **FA** (Fabric Aggregate) layer. Its primary function is to serve the traffic that flows between buildings in a region (east/west) and traffic exiting or entering a region. Our FA layer has gone through multiple generations, and the latest generation is called **HGRID** (HexaGrid). Different from previous generations of using monolithic expensive switches, the HGRID employs a disaggregated architecture and consists of a set of commodity switches. The sub-switches facing downward to the fabric are grouped together and called **FADU** (Fabric Aggregate Downlink Unit), while the upward sub-switch group is called **FAUU** (Fabric Aggregate Uplink Unit). The disaggregated approach allows us to accommodate larger regions and different traffic patterns while providing the flexibility to adapt to future growth. HGRID provides an ingress and egress point from the region to the backbone. Given its important position, migrating from the previous generations to the latest has been a challenging task.

**Regional aggregation.** Beyond FA, we introduce another layer called **MA** (Metro Aggregation) that provides connectivity to different regions that are close in proximity. We also adopt disaggregation architecture for MA, called **DMAG**.

**Backbone.** At the top layer, the wide-area backbone network interconnects datacenters globally. The datacenter to Backbone connectivity has also evolved due to the scale challenges. **DRs** (Datacenter Routers) are high-end routers sitting at the boundary of datacenter and Backbone. **EB** routers are the border routers from the backbone side to connect to DR. **EBB** (Express Backbone routers) are at the core of the WAN.

## 2.2 Why is it hard?

Network migration refers to the process of changing the network by adding, removing, or swapping switches and circuits. It is a common operation in DCNs and can be triggered for multiple reasons, such as adopting new hardware and technology, decommissioning old equipment, expanding capacity, and introducing new topology/routing. A network migration task in this paper typically touches thousands to tens of thousands of circuits per DC, and thus could result in significant changes to the underlying network topology and capacity. In addition, a migration often involves physical deployment work on site which can last months. Thus, given its potential impact and its lengthy duration, we have to plan the migration carefully and account for all possible intermediate states.

We explain the migration using a simplified DCN topology between two layers. The migration job is to upgrade the FA layer from an old generation to a new generation (HGRID) with more nodes and larger capacity, shown in Figure 2 (a).

**Consider efficiency.** While Figure 2(a) shows the logical view of the migration, we expand its physical connectivity to Figure 2(b). There are four planes of SSWs, shown in four different colors. To simplify, we assume there are four FAs at the top layer. Each FA connects to one SSW in each plane. A naive approach is that we pick one link between SSW and the old FA (e.g., circuit 13), drain the link, physically change the wiring to connect to HGRID, and then undrain the link to recover the traffic. However, the number of circuits that needs to be migrated can be even tens of thousands in one migration. Moving circuits one by one would be inefficient.

**Consider safety.** In the other extreme, we could drain all circuits to the old FA and change wiring all at once. However, this will cause too much capacity to be lost, i.e., *over 10 Tbps of capacity*. The

| Migration | Switches | Circuits | Capacity | Duration |
|---|---|---|---|---|
| HGRID | 320-352 | 13728-26792 | 1.3T-6.3T | 4-9 months |
| SSW Forklift | 144-288 | 14140-40320 | 14T-16T | 3-4 months |
| DMAG | 48-64 | 1648-5576 | 0.2T - 0.5T | 1-2 week(s) |

**Table 1: Migration statistics per DC.**

wiring is conducted in two different physical locations and can be quite complex. Thus, the capacity lost may take weeks. Figuring out the right amount of safe capacity that can be drained in a single step is not trivial, as it relates to the affordable downtime, the predicted traffic demand, and the available capacity in other parts of the network.

**Consider traffic funneling.** Even if we figure out the right number of circuits to drain at once, deciding on which circuits is yet another challenge. Let's assume we drain circuits 1-4 in one step. However, we cannot precisely control all circuits being drained at the same time. When 3 out of the 4 circuits are drained, traffic from FSWs will be all sent to the remaining circuit, causing severe congestion. This is known as the upstream traffic funneling phenomenon. An alternative is to horizontally drain circuits 1, 5, 9, and 13, that is, the first ports in each plane. But similarly, all four circuits cannot be drained at the exact same time. When 3 out of these 4 circuits are drained, the downstream traffic from FA will be flooded to the remaining circuit, causing downstream traffic funneling.

**Consider multiple DCs.** Even if we randomly pick four circuits in each plane to drain, there would still be a problem. Assume DC1 and DC2 are migrated at the same time, and we drain circuits 1,3 in DC1 and circuits 102, 104 in DC2. By looking at DC1 alone, we lose two circuits of capacity. However, DC1's circuits 2 and 4 are effectively lost as well. This is because their connected circuits in DC2 are down. The inter-DC circuits become not usable.

**Consider different meshing patterns.** What's worse, the old and new connectivity patterns can change. Figure 2(c) shows two different ways to interconnect SSW and the aggregation layer. The right one has a smaller capacity per node so it does not have a one-to-one mapping with the downstream planes.

**Consider different generations.** Finally, in production, multiple generations of networks coexist. As shown in Figure 2(d), one DC has four planes while the other DC has upgraded to eight planes. Migrating them would need different plans and migrating them together requires even more careful planning.

## 2.3 Challenges

As illustrated by the previous examples, we summarize the reasons of the migration challenges below.

**Complex and evolving DCN architecture.** The DCN architecture at Meta is more complicated than conventional Clos network topologies [37], as illustrated in §2.1. It contains eight layers and more than nine types of switches. Moreover, the DCNs are evolving gradually. Multiple generations of switches/circuits and routing protocols coexist in a single migration task. The mixture of topology and routing leads to different equivalence of switches (§4.1). Simply applying symmetry rules of [4] does not work. Besides, each region has its own idiosyncrasies, leading to different constraints. For example, some regions have tighter capacity buffers so we should be more careful to bring down switches. A region for hosting storage has a different reliability requirement than a compute region. An intelligent migration plan should take topology and service heterogeneity into consideration.

**Various operational constraints.** Various operational requirements exacerbate the network migration problem further. First, the demand variance is not negligible since one migration task spans from weeks to months (e.g., large migration tasks involving multiple DCs). On the one hand, traffic and services will grow organically over time. On the other hand, additional capacity should be preserved to accommodate unexpected traffic spikes and rerouting due to failures. The second important constraint is the availability of physical port numbers on every switch. The total number of ports on the hardware chassis is a hard constraint for the migration plan. We often need to decommission some circuits first to free up the ports and repurpose them for other connectivity. Such cases require extra steps in the migration plan.

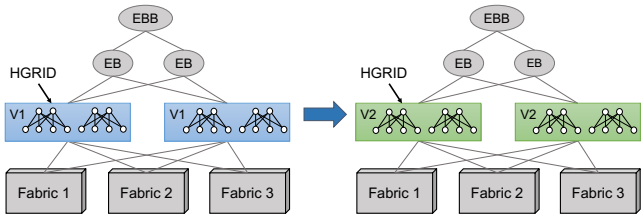
**Large search space.** In a real-world DCN with tens of thousands of switches, one network migration task typically involves tens to hundreds of switches per DC. The size of the search space (i.e., the number of possible migration plans) is the number of permutations of the switches to operate, which is super-exponential to the number of switches to operate. For example, more than 100! (or $9.3 \times 10^{157}$) plans exist for changing 100 switches. It is difficult to solve the problem with off-the-shelf optimization solvers [1, 2].

**Efficiency requirement.** In practice, the network conditions and traffic are time-varying. A network migration plan should be generated efficiently to react to these dynamics. Additionally, operators may need to tune the input constraints for a migration iteratively. A fast generation of migration plans can significantly reduce such trial-and-error process.

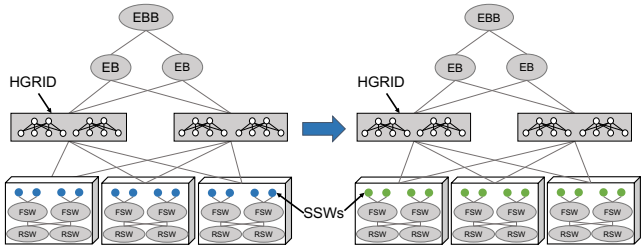
## 2.4 Network Migration Types

As discussed in §2.3, a network migration usually involves switches and circuits on different layers and of different types. In this paper, we share three large-scale migration types that have happened in the past few years and have been supported by Klotski. Table 1 shows the range of the number of switches, circuits, the affected capacity, and the total duration of the migration per DC. Note that the statistics of the table are for per DC and large migration tasks involve multiple DCs. The table shows that these migrations are *large-scale, affecting a huge amount of capacity, and long lasting*.

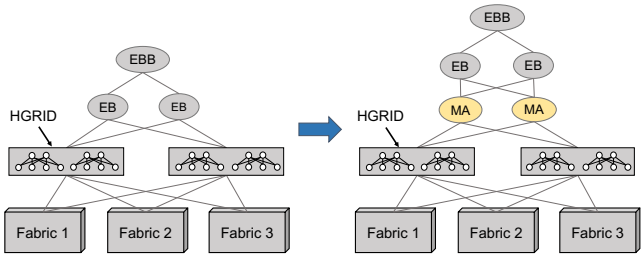
**HGRID V1 → V2 Migration** (Figure 3(a)). It is to replace all switches in the HGRID layer with a new generation of hardware. We need to remove/decommission the old switches first to create space for the new switches in the same location. The purpose of this migration type is to increase the inter-DC capacity for DCs within a region. The challenges of this migration type include three aspects. ($i$) It serves the inter-DC traffic within a region and every step results in a large volume of capacity change. Thus, the planning needs to be done more carefully. ($ii$) Each HGRID connects multiple DCs. Thus, the action space is huge, because there are many choices of which DCs to migrate at every step. ($iii$) Different DCs may have different generations of topologies and connectivity patterns, which further complicates the planning.




(a) HGRID V1 → V2 Migration.




(b) SSW Forklift Migration.




(c) DMAG Migration.

**Figure 3: Three network migration types at Meta.**

**SSW Forklift Migration** (Figure 3(b)). The second migration type has the largest scale. It upgrades all SSW switches in one DC to new-generation hardware to provide more capacity and more advanced routing capability. It is challenging because of its huge action space. Each plane has 36 switches, and there are 4 or 8 planes. The number of choices of the SSW numbers and which SSWs can be $2^{288}$ (or $5 \times 10^{86}$).

**DMAG Migration** (Figure 3(c)). It is to introduce the new regional aggregation, i.e., the DMAG layer, between FAUUs and EBs for interconnecting regions in geographic proximity. This is a significant topology change as the new layer can reduce traffic going into the Backbone. The challenge for DMAG Migration is its complex routing scheme. It interconnects EB and the FA layer, where the backbone uses centralized traffic engineering and has much more dynamic routing. Draining different circuits could cause different impacts, depending on the upstream EB's routing decision.

## 3 PROBLEM FORMULATION

A DCN can be abstracted as a graph where switches and circuits are represented as nodes and edges, respectively. Each switch $s$ has a type $R_s$ introduced in §2.1. Each circuit $c$ connects two switches and has capacity $W_c$. We then describe important definitions and the mathematical problem formulation. Table 2 lists the key notations in the problem formulation.

| | |
|---|---|
| $S_{opt}$ | Switch and circuit set to be operated |
| $R_s$ | Type of switch s |
| $A_s$ | Action type of switch s |
| $W_c$ | Capacity of circuit c |
| $D$ | Demand set containing the source and target switches and the forecasted traffic |
| $T(d, S, C, c)$ | The traffic on circuit c given the demand $d$, the switch set $S$, and the circuit set $C$ |
| $P(d, S, C)$ | A circuit path from the source switch $d_{src}$ to the target switch $d_{tar}$ given the demand $d$, the switch set $S$, and the circuit set $C$ |
| $L$ | Action sequence where $L_i$ is the i-th operated switch |
| $S_i$ | Switch set after action $L_i$ |
| $C_i$ | Circuit set after action $L_i$ |
| $\theta$ | Maximum utilization rate of each circuit |
| $P_s$ | Maximum port number of switch s |

**Table 2: Key notations in the problem formulation.**

**Action**. A network migration task is done by a sequence of actions operated on switches and corresponding circuits, called *action sequence*. Every switch to be operated on has its action type, which is decided by its switch type $R_s$ and the operation type (i.e., drain or undrain). For example, the action type of draining one SSW switch is different from that of draining one FADU switch or undraining one SSW switch. Operating a switch back and forth is meaningless. Thus, a switch can be operated at most once in one migration task.

**Objective**. The objective of the migration planning is to minimize the operational cost, i.e., operational time. We remark that *the operational cost mainly comes from two consecutive actions with different action types*. Different action types mean different switch types or different operation types which cannot be done simultaneously. However, the switches with the same type are usually placed close at Meta. Thus, actions with the same action type can be done by the operators simultaneously with negligible extra operational cost. We formulate the objective as Equation 1.

$$\min_{L} \sum_{i=1}^{|L|-1} \mathbb{1}(A_{L_i} \neq A_{L_{i-1}}) + 1 \tag{1}$$

$$\{L_i | 0 \leq i < |L|\} = S_{opt} \tag{2}$$

$$L_i \neq L_j, \forall 0 \leq i, j < |S_{opt}| \wedge i \neq j \tag{3}$$

$$\exists P(d, S_i, C_i), \forall d \in D, \forall i,$$
$$s.t. (A_{L_i} \neq A_{L_{i-1}} \wedge 1 \leq i < |L|) \vee i = |L| - 1 \tag{4}$$

$$\sum_{d \in D} T(d, S_i, C_i, c)/W_c \leq \theta, \forall c \in C_i, \forall i,$$
$$s.t. (A_{L_i} \neq A_{L_{i-1}} \wedge 1 \leq i < |L|) \vee i = |L| - 1 \tag{5}$$

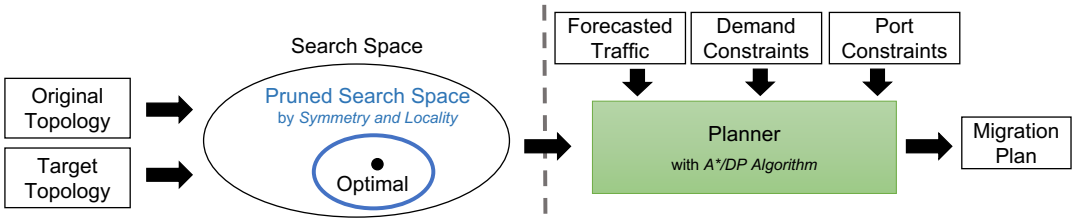



**Figure 4: Overview of Klotski.**

$$\sum_{c:c_{src}=s||c_{tar}=s} 1 \le P_s, \forall s \in S_i, \forall i,$$
$$s.t.(A_{L_i} \neq A_{L_{i-1}} \wedge 1 \le i < |L|) \vee i = |L| - 1 \quad (6)$$

The constraints in Equation 2- 6 are explained below.

- *Availability constraints* (Equation 2- 3). An available action sequence $L$ should contain only switches in $S_{opt}$ and contain each switch in $S_{opt}$ once and only once.
- *Demand constraints* (Equation 4- 5). Multiple traffic demands exist between RSWs or between RSW and EBB. For each demand $d$, at least one path from the source $d_{src}$ to the target $d_{tar}$ should exist. The utilization rate of each circuit, $\sum_{d \in D} T(d, S_i, C_i, c)/W_c$, should be lower than a threshold $\theta$, to survive failures and absorb traffic spikes. The demand constraints should be checked every time the action type changes in the action sequence $L$ and at the end of $L$ as consecutive actions with the same type are operated in parallel.
- *Port constraints* (Equation 6). A switch has a certain number of ports decided by its hardware configuration. The number of switches one switch can connect with should not exceed its port number. The port constraints need to be checked at the same time as the demand constraints.

## 4 KLOTSKI DESIGN

Figure 4 shows an overview of Klotski. Klotski takes the network topologies, forecasted traffic, and constraints as input. Klotski computes the network migration plan in two steps. First, Klotski leverages the inherent symmetry of DCNs and the locality of switches to prune the search space. Then, it computes the optimal migration plan with the lowest operational cost by its DP-based algorithm or A* search algorithm.

### 4.1 Search Space Pruning

Naively, a brute-force approach needs to examine all possible action sequences for a migration task to decide the one with the minimal operational cost. However, it is impractical in production given that one migration task usually involves hundreds of switches and thousands of circuits. For example, there are 100! (or $9.3 \times 10^{157}$) possible sequences for changing 100 switches. It is impossible to go through all of them in a reasonable amount of time. Klotski exploits the inherent symmetry and locality of switches to reduce the search space.

Here we follow the notion of equivalent switches and symmetry blocks in Janus [4]. Briefly, switches that connect to the same hosts and have the same routing table are equivalent, and equivalent switches form a symmetry block. The operation order of equivalent switches does not affect the cost and constraints of migration. However, we find that there is little symmetry in Meta's DCNs. Each symmetry block consists of at most two switches for our three real-world migration types.

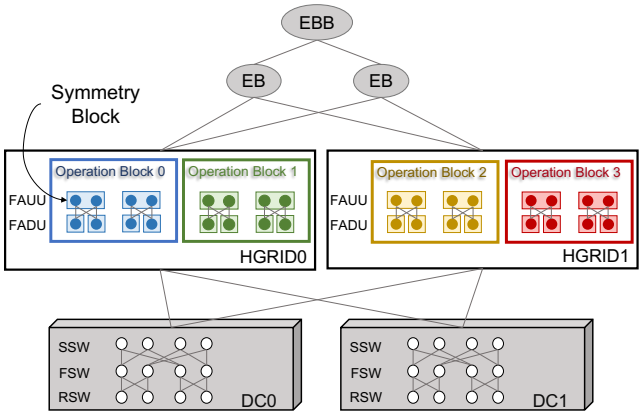



**Figure 5: Example of symmetry blocks and operation blocks in HGRID V1→V2 Migration.**

In practice, we find that the locality of switches influences the operational cost and constraint satisfiability. Specifically, we can operate switches that are close to each other simultaneously with little extra operational cost and little impact on constraints. Thus, we merge symmetry blocks with locality into one operation block, to further prune the search space. Switches in one operation block are operated together. For example, in Figure 5, each FAUU symmetry block contains two switches and each FADU symmetry block contains one switch. We merge two FAUU symmetry blocks and four FADU symmetry blocks into one operation block, and merge six operations on symmetry blocks to one operation on the operation block. The organization policy of operation blocks is based on our operational experience and broadly tested over real-world topologies and migrations. The details are described in §5.

### 4.2 Efficient Satisfiability Checking

Both the DP and A* planners need to check the satisfiability of the practical constraints defined in §3. It is time-consuming as all constraints are checked on a large DCN topology with up to $O(10,000)$ switches and $O(100,000)$ circuits. We propose the notion of *equivalent states* and the design of a *compact topology representation* to speed up the checking.

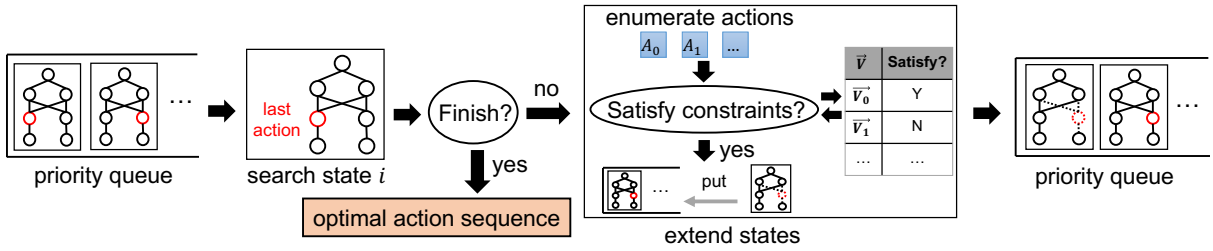



**Figure 6: The search process of A* search planner to find the optimal network migration plan.**

DEFINITION 1. ***Equivalent states.*** *States $n_1$ and $n_2$ are equivalent if they have the same network topology.*

Equivalent states can be derived by going through different action sequences from the original topology. For example, assume there are two action types, 0 and 1. Performing two action sequences $(0, 0, 1, 1)$ and $(0, 1, 0, 1)$ on the same original topology will generate the same topology, since they both do the actions 0 and 1 twice. The satisfiability checking for equivalent states can be done only once because the constraint satisfiability of a state is merely determined by the current network topology, which is the same for equivalent states. Thus, we use a cache table $T_C$ to store the mapping of topologies and the checking results. Every time we need to do the satisfiability checking, we can directly fetch the checking result if the topology is in $T_C$. Otherwise, we examine the constraints and store the checking result in $T_C$ for future reference.

**Compact topology representation.** Due to the large scale of DCN topologies, naively storing the topologies in $T_C$ incurs excessive indexing overhead and memory footprint. Our important insight is that equivalent states with different ordering of actions have the same number of each action type. Thus, we design an ordering-agnostic compact topology representation to represent a topology as $\vec{V} = (v_i)$, where $v_i$ is the number of finished actions with action type $i$. For a state $n$, $\vec{V}$ can be easily obtained by counting the action sequence $L$ of $n$ in linear time, i.e., $O(|L|)$, which is negligible in comparison with the overhead of redundant satisfiability checking.

## 4.3 DP-based Planner

Dynamic programming (DP) is a classical method to solve combinatorial optimization problems [3, 6]. We design a DP-based planner, which is used in an early version of Klotski, to plan our network migration. This planner achieves polynomial time complexity. The key idea is that we can visit each intermediate topology only once with the compact topology representation in a specific order of the representation.

Specifically, the DP state $f(\vec{V}, a)$ stores the minimal cost of changing the original topology to the current topology $\vec{V}$ with the last action type $a$. We track the last action $a$ to calculate the cost for its successor states by comparing it with the next action type. For each state, we enumerate all possible predecessor states, $f(\vec{V^*}, a^*)$, that satisfy Equation 8 to get the minimal cost as shown in Equation 7.

$$f(\vec{V}, a) = \min_{a^*} f(\vec{V^*}, a^*) + \begin{cases} 1 & a \neq a^* \\ 0 & a = a^* \end{cases} \tag{7}$$

$$v_i = \begin{cases} v_i^* & i \neq a \\ v_i^* + 1 & i = a \end{cases}, \forall v_i^* \in \vec{V^*} \tag{8}$$

Note that for a topology represented by $\vec{V} = \{v_i\}$, any predecessor topology represented by $\vec{V^*} = \{v_i^*\}$ has a smaller total number of actions, i.e., $\sum v_i^* < \sum v_i$. As a result, we propagate the states based on Equation 7 in ascending order of the total number of finished actions. We use an auxiliary array $g(\vec{V}, a)$ to store the last action type of the optimal action sequence to state $\vec{V}$. The optimal action sequence can be rebuilt by tracking from the target topology backward to the original topology in $g$. Theorem 1 shows the time complexity of the DP-based planner.

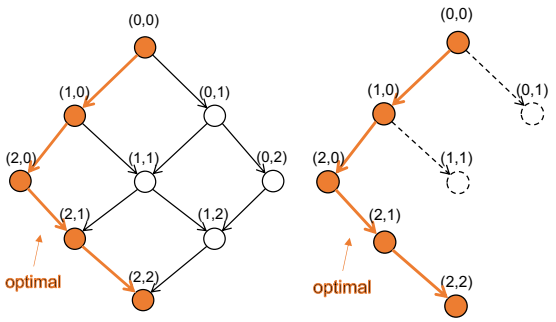
THEOREM 1. *Given that there are $|A|$ action types, $|L|$ actions to do, $|S|$ switches, and $|C|$ circuits, the time complexity of DP-based planner is $O(|A| \cdot (\frac{|L|}{|A|})^{|A|} \cdot (|A| + |S| + |C|))$.*

The proof of Theorem 1 is in Appendix A.1. Briefly, the time complexity of the DP-based planner is the product of the number of DP states and the processing time for each state. The number of DP states is bounded by $|A| \cdot (\frac{|L|}{|A|})^{|A|}$ according to Jensen's Inequality [20]. Each state requires finding the predecessor states and performing constraint satisfiability checking, i.e., $O(|A| + |S| + |C|)$.

## 4.4 A* Search Planner

Considering the scale of network migration tasks in production, it is still time-consuming to run the DP-based planner even after the search space is pruned and compacted. We design an A* search planner based on A* algorithm [18] to further reduce the planning time, and use a domain-specific priority function to guide the search process.

Figure 6 shows the search process of our A* search planner. The search process starts from the original network topology and takes actions step by step until it arrives at the target topology. Each search state can be specified by the operated action sequence from the original state and the action type of the last finished action type. For the current search state $i$, it generates the next state candidates by applying every action type to the current topology. The new topologies that meet the demand constraints and port constraints are regarded as feasible state candidates and will be put into a priority queue. Meanwhile, our A* search planner computes priorities for the newly added state candidates and picks the one with the highest priority in the priority queue as the next search state.




(a) DP Planner. (b) A* Planner.

**Figure 7: Comparison of Klotski planners. Nodes represent states and edges represent actions. Tuples of two numbers are the compact topology representations.**

**Priority design.** In A* algorithm, the priority $f(n)$ represents the estimated cost of a search path that goes through the current state $n$. $f(n)$ is composed of two parts, i.e., $f(n) = g(n) + h(n)$, where $g(n)$ is the existing cost from the original state to the current state $n$ and $h(n)$ is an admissible heuristic function—it is a lower bound on the cost-to-go from the state $n$ to the target state. A* is guaranteed to find the optimal solution if and only if $h(n)$ never overestimates the cost-to-go. In our problem, smaller $f(n)$ represents a higher priority. $g(n)$ is calculated according to the cost function (Equation 1), where $L$ is the operated action sequence of state $n$.

The design of $h(n)$ is important to the performance of A* algorithm. We choose the number of remaining action types as $h(n)$ Such future cost estimation design has two principal benefits. First, it guarantees the *optimality* of the final result. It is obvious that $h(n) \leq h^*(n)$ holds for any state $n$, where $h^*(n)$ represents the actual future cost of $n$, since the number of action type changes in an action sequence is at least the number of action types. Thus, the algorithm is guaranteed to return the optimal result [18]. Second, the computation of $h(n)$ is efficient. $h(n)$ is computed for every state candidate during the search process. Considering the number of state candidates to check, $h(n)$ needs to be computed efficiently. As the total number of actions for each type is given and the number of finished actions for each type can be stored during the search, $h(n)$ can be easily obtained by getting the actions to be done and counting the number of their types.

In practice, we found that many different search states usually have the same value of $f(n)$, especially at the beginning of the search. To differentiate these states, we use the number of finished actions as the secondary priority. Intuitively, the search states with more finished actions should have a higher priority, since these states are usually closer to the target state and have a higher probability to derive the optimal solution. Theorem 2 shows the time complexity of the A* search planner.

THEOREM 2. *Given that there are $|A|$ action types, $|L|$ actions to do, $|S|$ switches, and $|C|$ circuits, the time complexity $T$ of the A* search planner satisfies* $T = \Omega(|L| \cdot (|A| + |S| + |C|)) = O((\frac{|L|}{|A|})^{|A|} \cdot (|A| + |S| + |C|))$.

The proof of Theorem 2 is in Appendix A.2. Briefly, the time complexity of the A* search planner is the product of the number of search states and the processing time for each state. The processing time for each state is also $O(|A| + |S| + |C|)$. Ideally, the planner can directly find the optimal solution by visiting only $|L|$ states. In the worst case, the planner has to search all states, i.e., up to $(\frac{|L|}{|A|})^{|A|}$.

**A* VS DP.** The A* search planner runs faster than the DP-based planner in practice. This is because the A* search planner returns the optimal solution once it visits the target state, and it may visit fewer states than the DP-based planner. We use an example to illustrate the benefit of A* search in Figure 7. In this example, there are two action types and four actions. The A* search planner visits five states and performs four satisfiability checks, while the DP-based planner visits all nine states and performs eight satisfiability checks. However, it is important to note that although A* is typically faster by avoiding the expansion of all nodes, in the worst-case scenario it may still visit every node in the search tree.

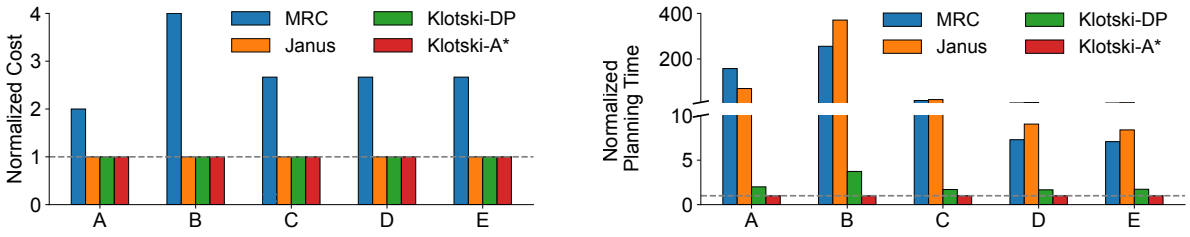
## 5 IMPLEMENTATION

**System-level implementation.** At Meta, we have productionized Klotski by integrating it into EDP-Lite (Engineering Design Package) pipeline, which provides guidance on network topology migrations. For network migration, EDP-Lite pipeline takes Network Product Definition (NPD) format original/target topologies and demand information as inputs. NPD is a generic data structure used at Meta to define high-level properties of network topologies. NPD divides DCNs into six parts and describes them separately for scalability. These six parts are Fabric, HGRID, MA, EB, DR, and BB. In each part, it records the switches based on their roles and positions, and the way these switches are interconnected. Besides, NPD also contains information about migration phases and hardware. After the pipeline receives NPD data, it converts them into topologies and passes the topologies to Klotski. When the planning is done, Klotski returns an ordered list of topology phases. Each phase corresponds to one migration step.

**Organization policy for operation blocks.** We design the organization policy based on the layout and type of switches and circuits. For the HGRID layer, one grid contains multiple near symmetry blocks and is set as one operation block. SSWs are asymmetric to each other. We split SSWs on a plane into several operation blocks, considering the traffic demand. For DMAG migration, we need to drain the circuits between EBs and FADUs, and undrain MAs. To prune action space, we group MAs/circuits by the same type of switches they connect. As one EB connects more switches in practice, we group the MAs/circuits by EBs to release more ports with one action.

**Klotski planner.** As shown in Figure 4, Klotski planner takes the original and target topologies, forecasted traffic, and constraints as input and outputs the migration plan. Following previous work [54], we focus on macro-scale network behavior, e.g., the traffic and capacity of circuits, rather than micro-scale network behavior, e.g., network congestion. We use the equal-cost multi-path (ECMP) routing policy. For efficient satisfiability checking, we utilize an unordered map to store the representation-satisfiability pair, $(\vec{V}, 0/1)$.

**Cost function.** In practice, the total time of operating one operation block is slightly longer than that of operating one switch. The




(a) Optimality of Plan Cost.

(b) Normalized Planning Time.

**Figure 8: Compare Klotski with baselines under various topology sizes.**

| Topology | Switches | Circuits | Actions |
|---|---|---|---|
| $A$ | $\sim 40$ | $\sim 80$ | $\sim 50$ |
| $B$ | $\sim 100$ | $\sim 600$ | $\sim 100$ |
| $C$ | $\sim 600$ | $\sim 8,000$ | $\sim 300$ |
| $D$ | $\sim 1,000$ | $\sim 20,000$ | $\sim 300$ |
| $E$ | $\sim 10,000$ | $\sim 100,000$ | $\sim 700$ |
| $E-DMAG$ | $\sim 10,000$ | $\sim 100,000$ | $\sim 100$ |
| $E-SSW$ | $\sim 10,000$ | $\sim 100,000$ | $\sim 300$ |

**Table 3: Configurations for each topology.**

operational cost is based on action types, operation blocks, and the proficiency of operators. The cost of operating $x$ switches within one operation block can be formulated approximately as a linear function, $f_{cost}(x) = 1 + \alpha(x-1)$, where $\alpha \in [0,1]$. $\alpha$ is set to 0 by default. We extend Klotski to handle the generalized cost function. The main idea is that two adjacent actions with different types are operated serially, while two adjacent actions with the same type can be operated simultaneously with extra cost $\alpha$ for each action. For Klotski-A*, we use Equation 9 to calculate $h(n)$, where $n$ is the current search state, $A$ is the action set, and $N_a$ is the remaining number of actions with type $a$.

$$h(n) = \sum_{a \in A \wedge N_a > 0} 1 + \alpha(N_a - 1) \tag{9}$$

# 6 EVALUATION

We first evaluate the optimality and efficiency of Klotski compared with state-of-the-art planners over different sizes of topologies. We then show the generality of Klotski over different migration types. The results show that Klotski can always find the optimal solution and is up to 381× faster than baselines. We also investigate the design choices of Klotski.

## 6.1 Methodology

**Topology.** We evaluate Klotski on five production network topologies with different scales, i.e., $A$, $B$, $C$, $D$, and $E$. These five topologies are listed in Table 3 in ascending order of topology sizes, with 40–10,000 switches and 80–100,000 circuits. The circuit capacity is in the unit of Tbps. $E$ is comparable to Meta's DCN. These topologies are representative of the migration cases that Klotski handled in over 100 DCs at Meta.

**Traffic.** The traffic demand is forecasted based on historical data collected by Meta's DCNs, reflecting the average traffic requirements in the near future. There are three kinds of source and target pairs, i.e., RSW to EBB, EBB to RSW, and RSW to RSW. The traffic is typically hundreds of Tbps.

**Constraints.** In terms of demand constraints, the maximum utilization rate $\theta$ is set to be 75% by default, according to our practical experience. We also investigate the impact of $\theta$ by changing it from 55% to 95% in §6.4. In the terms of port constraints, the maximum port number $P$ varies according to topologies and real-world configurations.

**Network migrations.** We evaluate Klotski on all three migration types in §2.4. Topologies $A-E$ perform HGRID V1→V2 Migration, and replace all FADUs and FAUUs. The number of actions ranges from 50 to 700. $E-DMAG$ performs DMAG Migration, which drains all circuits between EBs and FADUs, and adds MAs and related circuits. $E-SSW$ performs SSW Forklift Migration, which takes about 300 actions.

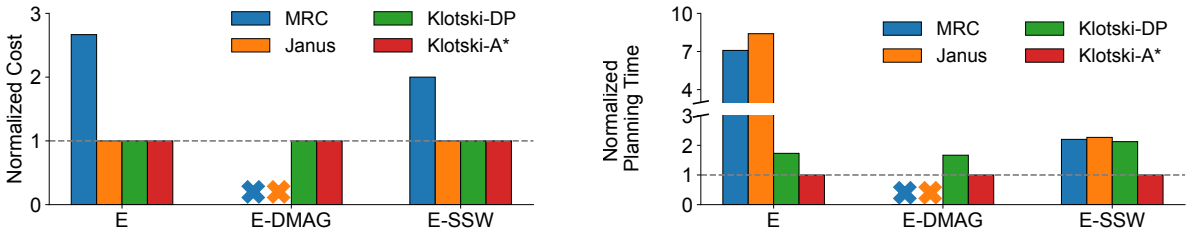
**Baselines.** We compare two versions of Klotski with two state-of-the-art (SOTA) baselines. **Klotski-DP** uses the DP-based planner in Klotski, while **Klotski-A*** uses the A* search planner. Planners that maximize the minimum residual capacity (**MRC**) use a greedy strategy for each step of the migration plan [37]. **Janus** leverages the intrinsic symmetry of DCNs to prune the search space [4]. We define the superblock in Janus as the operation block in Klotski.

**Evaluation metrics.** To compare the optimality and efficiency of planners, we measure the minimum cost each planner can get and the planning time to find the optimal plan. All planners run at most 24 hours as more time for planning does not meet the efficiency requirement in production. We normalize the cost by the optimal cost of every migration task, and normalize the planning time by that of Klotski-A*. While we report normalized time for privacy concerns, Klotski-A* uses less than 4 minutes to generate a plan for the largest topology.

## 6.2 Scalability

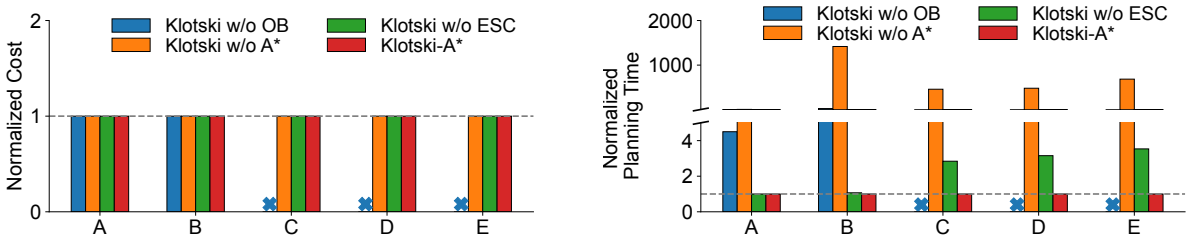
We evaluate Klotski over topologies $A-E$ with up to 10,000 switches under HGRID V1 → V2 Migration.

**Optimality.** Figure 8(a) shows the normalized minimum cost each planner can get. MRC uses a greedy strategy and cannot find the optimal solution for all evaluated topologies. The other three planners can always find optimal solutions.




(a) Optimality of Plan Cost. (b) Normalized Planning Time.

**Figure 9: Compare Klotski with baselines under various migration types. A cross indicates that the planner cannot plan for the migration task.**




(a) Optimality of Plan Cost. (b) Normalized Planning Time.

**Figure 10: Impact of Klotski design choices. A cross indicates that no available solution is found within 24 hours.**

**Efficiency.** Figure 8(b) demonstrates the planning time of each planner normalized by that of Klotski-A*. The absolute planning time increases from $A$ to $E$. The planning times of MRC and Janus are $7.1 - 262.6\times$ and $8.4 - 380.7\times$ slower than Klotski-A*, due to three reasons. First, these two planners need to preprocess all available action combinations, which is time-consuming. Second, there is little symmetry in the complex migration tasks we study that Janus can leverage. Third, Janus traverses the entire search space, while Klotski-A* returns once it visits the target state [18]. Klotski-A* is $1.7 - 3.8\times$ faster than Klotski-DP. Intuitively, Klotski-A* is intelligently guided by the priority and gets to the target topology without visiting all intermediate topologies. However, Klotski-DP has to visit all intermediate topologies to calculate the DP function correctly. We remark that reducing the planning time from hours to minutes is important in practice for two reasons. First, before the migration, operators need to adjust the migration configurations interactively based on the planning solutions for each migration. Second, the traffic demand varies from time to time and we need to re-run the planner with the updated demand during the migration.

## 6.3 Generality

To demonstrate the generality of Klotski over different migration types, we compare all planners over $E$, $E - DMAG$, and $E - SSW$. These three migration tasks are only different in migration types. The evaluation results are shown in Figure 9. Klotski-A* is up to $7.1\times$ faster than MRC, $8.4\times$ faster than Janus, and $2.1\times$ faster than Klotski-DP. Note that MRC and Janus cannot handle migration types that change the topology (E-DMAG), while Klotski is capable of planning all types of migrations, showing the generality of Klotski.

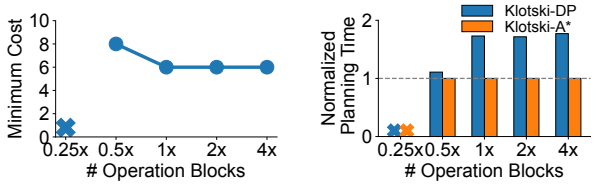
## 6.4 Analysis of Klotski

**Impact of Klotski design.** We compare Klotski-A* with three variants, Klotski-A* without considering operation blocks (Klotski w/o OB), Klotski-A* without A* algorithm (Klotski w/o A*), and Klotski-A* without efficient satisfiability checking (Klotski w/o ESC), as shown in Figure 10.

Klotski w/o OB fails to find available solutions for large topologies, $C - E$, and takes $4.4 - 26.7\times$ longer time on small topologies. These results show that search space pruning with symmetry and locality is essential for efficiency. Compared with Klotski w/o A*, the speedup of Klotski-A* ranges from $7\times$ to $1456.5\times$. The priority function of A* algorithm guides Klotski-A* to visit the states that have high probabilities to form the optimal solution first. Besides, Klotski-A* only explores part of the search space, while Klotski w/o A* needs to explore the whole search space. Klotski-A* runs $1.1 - 3.5\times$ faster than Klotski w/o ESC, showing the benefit of reducing the duplicated satisfiability checking, especially on large topologies. For small topologies, almost no equivalent states are checked repeatedly, leading to small speedups.

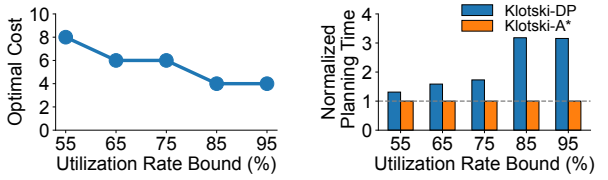
**Impact of operation blocks.** We evaluate Klotski with different organization policies for operation blocks. By merging or splitting the default operation blocks, we get five settings in Figure 11. The factors mean the number of operation blocks compared with the default setting described in §5.

The minimum cost is negatively related to the number of operation blocks. For large operation blocks, lots of switches/circuits are operated together, which makes the constraints harder to meet. As a result, no available solution exists in $0.25 \times E$. In contrast, small operation blocks mean finer-grained actions and may reduce the



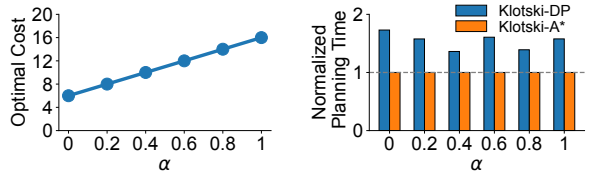

(a) Minimum Cost. (b) Normalized Planning Time.

**Figure 11: Impact of operation blocks. A cross indicates that the migration task has no available action sequence.**




(a) Optimal Cost. (b) Normalized Planning Time.

**Figure 12: Impact of utilization rate bound.**




(a) Optimal Cost. (b) Normalized Planning Time.

**Figure 13: Impact of the cost function.**

minimum cost. Besides, more operation blocks increase the exact time to minimum cost. Thus, the organization policy brings a balance between optimality and efficiency. Klotski-A* is always faster than Klotski-DP and the speedups are $1.1 - 1.8\times$. This is because A* algorithm visits fewer states than DP algorithm. Additionally, efficient satisfiability checking ensures that Klotski-A* performs no worse than Klotski-DP even in extreme settings.

**Impact of demand constraints.** We also study the impact of utilization rate bound for demand constraints as shown in Figure 12. Intuitively, a lower utilization rate bound means more strict constraints, and fewer switches/circuits can be drained together. Therefore, the optimal cost has a negative relationship with the utilization rate bound. Under higher utilization rate bound, i.e., looser constraints, Klotski-A* visits only a few states and achieves up to $3.2\times$ speedup over Klotski-DP.

**Impact of the cost function.** To investigate the impact of the cost function, we vary the $\alpha$ from 0 to 1 and compare Klotski-A* with Klotski-DP on topology $E$, as shown in Figure 13. As expected, the optimal cost increases when $\alpha$ gets larger and both planners can find the optimal solutions. Klotski-A* has shorter planning time than Klotski-DP for all tasks.

# 7 DEPLOYMENT EXPERIENCE

Klotski has been deployed at Meta since 2020 to generate DCN migration plans for over 100 datacenters within 20 regions. Klotski serves tens of migration tasks every day. We are actively exploring its extensions to other networks such as Backbone and PoPs. In this section, we share our deployment experience.

## 7.1 Routing Configurations and Traffic Forecasts

In some production scenarios, network operators need to create special routing configurations during the migration process to utilize the special intermediate topology. For example, during the HGRID V1 → V2 Migration, we can no longer rely on pure ECMP to split the traffic among all switches. This is because an HGRID V1 aggregated switch has a different capacity with HGRID V2: they contain a different number of sub-switches and each sub-switch has a different forwarding capability. The relative number between the two types of switches is also changing as the migration progresses. In one outage, we experienced high packet loss even when draining a single link in V1. This is because another box had been upgraded to V2. The downstream systems performed ECMP to both HGRIDs. The old generation could not provide sufficient capacity even with the minimum unit of capacity loss. To handle this challenge, network operators created temporary routing configurations to balance the traffic between HGRID V1 and V2, and updated them according to the migration plan. We are currently extending Klotski to incorporate more flexible routing configurations into the problem formulation.

Another experience is that we need to incorporate traffic demand forecasts into the migration process. We overlooked the demand increase initially as it is usually only needed for long-term planning. However, during deployment, we found some migrations can easily take more than one month. The traffic in a region or for a service is possibly changing drastically during this period. For example, if the traffic demand increases by ten percent after the first two migration steps, then the rest migration steps would fail to satisfy demand constraints. Therefore, we run the forecast after each migration step. Consequently, we also re-run the migration planning with the updated demand and update the migration plan.

## 7.2 Incorporating Operational Constraints during Migration

Besides the standard factors such as traffic demand and port number, there are other operational constraints that we have to handle when using Klotski in production.

**Failures during operation duration.** Klotski only generates the logical action plan, which needs to be translated to actual configuration changes and to be pushed to switches. The configuration and push pipeline may experience failure due to complex dependency on other systems. For example, an undrain step may be unsuccessful if the network management system experiences an outage. If performing another drain before the recovery of the previous undrain, it could result in capacity insufficiency. Thus, we add extra audits and safety checks to Klotski's plans during operation.

**Simultaneous operations.** While Klotski is used to handle large-scale migration, there is other routine maintenance that is not controlled by Klotski, such as firmware upgrade and device rebuild.

These changes do not require physical changes to the network but they could also cause downtime to the switches. Therefore, it is necessary for Klotski to generate plans according to the real-time topology changes, and as a result, it requires Klotski to generate results efficiently.

**Space and power constraints.** The old and new hardware generations often share the same space and power. In some cases, there are additional space and power available to support transient state but that could be limited. We consider such constraints when generating intermediate states in Klotski.

**Unexpected traffic surge.** As shown in §2, these operations can take days to weeks so we need to accommodate unexpected service behavior changes. For example, in one incident, warm storage decided to change its backup placement strategy during a network migration. That caused days of traffic spikes. Meanwhile, Klotski's migration reduced the capacity and worsens the scenario. Thus, service activity should be considered if it could result in large changes in traffic patterns. We re-run Klotski to adapt the migration plan to the changes.

**Traffic funneling.** As discussed in §2.2, traffic funneling happens during migration due to asynchronous operations. To handle this phenomenon, Klotski increases the utilization of related circuits while planning. Meanwhile, we are designing better affinity rules.

**OPEX savings.** Finally, physical migration requires sending workforce to the site to perform manual work. Different sequences of steps could have different costs in terms of human efficiency. Indeed, we are adding a cost model to Klotski which can optimize for OPEX spending.

## 7.3 Classical Methods vs. DL Methods

Deep learning (DL) has been increasingly applied to planning problems [7, 13, 23, 54]. We worked with top DL researchers and engineers at Meta for more than two years. We explored various DL methods with cutting-edge technologies.

For the planning process, we explored three directions. First, we employed reinforcement learning (RL) [30, 43] including DQN [30], PPO [35], and A2C [29], to solve this decision-making problem. To embed the DCN topology, we leveraged graph neural network (GNN) [34, 45], e.g., GCN [22] and GAT [42]. Additionally, we explored domain-specific techniques, e.g., link-node transformation [54] and curiosity-driven reward [36, 52], to assist the training process. Second, we tried transfer learning [44, 46] by training the GNN representation module with labeled migration tasks and transferring the pre-trained module to predict the next action. Third, we also leveraged the pre-trained GNN to score candidate actions and guide the A* search process.

However, we met the following practical obstacles, so that Klotski stayed with classical methods in our deployment. (*i*) *Scalability*. Although DL can solve small-scale migrations, it is difficult to generalize the DL model for large-scale migrations in production. Most DL models we have tried cannot even find feasible solutions for large-scale migrations. (*ii*) *Efficiency*. Training a DL model takes hours even on small topologies while Klotski only takes minutes on large topologies. For economic efficiency, DL methods require expensive GPUs while Klotski just uses cheap CPUs. (*iii*) *Reliability*. As randomness exists commonly in DL, it can not always get the optimal solution. On the contrary, Klotski with classical methods guarantees the optimality. Most of the above obstacles remain open problems in the DL field. Yet, the exploration has not been hindered and we will report back to the community in the future.

# 8 RELATED WORK

**Network topology expansions.** Many efforts have been proposed to support DCN topology expansions. Iteratively-designed DCN structures [15–17, 24, 25] support coarse-grained expansions. Fine-grained incremental expanders [5, 11, 38, 41] have high complexity. Optimization-based methods [8, 9, 50] are weak in scalability or require additional input. FatClique [49] is designed for lower lifecycle management complexity including expansions. Klotski is a general approach for a wide range of network migration types.

**Network migration planners.** Prior work [19, 21, 26, 33] has explored planning network migrations for only specific switch configurations. Janus [4] exploits symmetry of DCN topologies to plan network migrations, which is the closest work to Klotski. Janus assumes the symmetry is not changed during migration which does not hold for our DMAG migration. Klotski does not have this assumption and can handle more general migrations. Moreover, there is little symmetry for complex DCN topologies and migration tasks at Meta, making Janus inefficient. Klotski further considers DCN locality to speed up the planning.

**Practical network management systems.** Network management is a common challenge for large enterprises. Some works [27, 31, 39, 53] propose abstractions for network lifecycle management. Govindan et al. [14] focus on network availability during network migrations and conclude some design principles at Google. Klotski focuses on planning network migrations, which is essential in network management.

# 9 CONCLUSION

In this paper, we present Klotski, a production system for efficient and safe DCN migration at Meta. Klotski formulates the migration planning with domain-specific constraints for safety. Besides, Klotski leverages the inherent symmetry and locality of DCNs to prune the search space, and the power of informed search to find the optimal plans for efficiency. The evaluation results demonstrate that Klotski realizes efficient and safe DCN migration for production network topologies. Klotski has been deployed at Meta since 2020 to generate migration plans for over 100 datacenters within 20 regions.

*This work does not raise any ethical issues.*

**Acknowledgments.** We sincerely thank the anonymous reviewers for their valuable feedback on this paper. We acknowledge Haowen Liu for his help with the experiments. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant number 62172008 and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin, Xuanzhe Liu and Ying Zhang are the corresponding authors. Yihao Zhao, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

# REFERENCES


[1] 2023. CPLEX optimizer. https://www.ibm.com/analytics/cplex-optimizer.
[2] 2023. Gurobi solver. https://www.gurobi.com/.
[3] Hassan AbouEisha, Talha Amin, Igor Chikalov, Shahid Hussain, and Mikhail Moshkov. 2019. *Extensions of dynamic programming for combinatorial optimization and data mining*. Springer.
[4] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk based planning of network changes in evolving data centers. In *ACM SOSP*.
[5] Cristóbal Camarero, Carmen Martínez, and Ramón Beivide. 2017. Random folded Clos topologies for datacenter networks. In *IEEE HPCA*.
[6] Robert L Carraway, Thomas L Morin, and Herbert Moskowitz. 1989. Generalized dynamic programming for stochastic combinatorial optimization. In *Operations Research*.
[7] Xinyun Chen and Yuandong Tian. 2019. Learning to perform local rewriting for combinatorial optimization. In *NIPS*.
[8] Andrew R Curtis, Tommy Carpenter, Mustafa Elsheikh, Alejandro López-Ortiz, and Srinivasan Keshav. 2012. Rewire: An optimization-based framework for unstructured data center network design. In *IEEE INFOCOM*.
[9] Andrew R Curtis, Srinivasan Keshav, and Alejandro Lopez-Ortiz. 2010. LEGUP: Using heterogeneity to reduce the cost of data center network upgrades. In *ACM CoNEXT*.
[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[11] Michael Dinitz, Michael Schapira, and Asaf Valadarsky. 2017. Explicit expanding expanders. In *Algorithmica*.
[12] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON tiles: Fast analytics on semi-structured data. In *ACM MOD*.
[13] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *NIPS*.
[14] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *ACM SIGCOMM*.
[15] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: A high performance, server-centric network architecture for modular data centers. In *ACM SIGCOMM*.
[16] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. Dcell: A scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM*.
[17] Deke Guo, Tao Chen, Dan Li, Mo Li, Yunhao Liu, and Guihai Chen. 2012. Expandable and cost-effective network structures for data centers using dual-port servers. In *IEEE TC*.
[18] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. In *IEEE G-SSC*.
[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*.
[20] Johan Ludwig William Valdemar Jensen. 1906. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. In *Acta mathematica*.
[21] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*.
[22] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[23] Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, learn to solve routing problems!. In *ICLR*.
[24] Dan Li, Chuanxiong Guo, Haitao Wu, Kun Tan, Yongguang Zhang, and Songwu Lu. 2009. FiConn: Using backup port for server interconnection in data centers. In *IEEE INFOCOM*.
[25] Zhenhua Li, Zhiyang Guo, and Yuanyuan Yang. 2016. BCCC: An expandable network for data centers. In *IEEE/ACM TON*.
[26] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating data center networks with zero loss. In *ACM SIGCOMM*.
[27] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *ACM SIGCOMM SelfDN*.
[28] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *ACM MOD*.
[29] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *ICML*.
[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* (2015).
[31] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. 2020. Experiences with modeling network topologies at multiple levels of abstraction. In *USENIX NSDI*.
[32] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *POPL*.
[33] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *ACM SIGCOMM Computer Communication Review*.
[34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE TNNLS* (2008).
[35] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
[36] Pier Giuseppe Sessa, Maryam Kamgarpour, and Andreas Krause. 2022. Efficient model-based multi-agent reinforcement learning via optimistic equilibrium computation. In *ICML*.
[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Holzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM*.
[38] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. 2012. Jellyfish: Networking data centers randomly. In *USENIX NSDI*.
[39] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down network management at facebook scale. In *ACM SIGCOMM*.
[40] EJ Truesdell, Jason Brent Smith, Sarah Mathew, Gloria Ashiya Katuka, Amanda Griffith, Tom McKlin, Brian Magerko, Jason Freeman, and Kristy Elizabeth Boyer. 2021. Supporting computational music remixing with a co-creative learning companion. In *ICCC*.
[41] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. 2016. Xpander: Towards optimal-performance datacenters. In *ACM CoNEXT*.
[42] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *STAT* (2017).
[43] Tongzhou Wang, Simon S Du, Antonio Torralba, Phillip Isola, Amy Zhang, and Yuandong Tian. 2022. Denoised mdps: Learning world models better than the world itself. In *ICML*.
[44] Xiao Wang, Haoqi Fan, Yuandong Tian, Daisuke Kihara, and Xinlei Chen. 2022. On the importance of asymmetry for siamese representation learning. In *IEEE CVPR*.
[45] Xiyuan Wang and Muhan Zhang. 2022. How powerful are spectral graph neural networks. In *ICML*.
[46] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* (2016).
[47] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *ACM ESEC/FSE*.
[48] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiunlin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. 2021. A social network under social distancing: risk-driven backbone management during COVID-19 and beyond. In *USENIX NSDI*.
[49] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. 2019. Understanding lifecycle management complexity of datacenter topologies. In *USENIX NSDI*.
[50] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. 2019. Minimal rewiring: Efficient live expansion for clos data center networks. In *USENIX NSDI*.
[51] Yihao Zhao, Ruihai Wu, and Hao Dong. 2020. Unpaired image-to-image translation using adversarial consistency loss. In *ECCV*.
[52] Lulu Zheng, Jiarui Chen, Jianhao Wang, Jiamin He, Yujing Hu, Yingfeng Chen, Changjie Fan, Yang Gao, and Chongjie Zhang. 2021. Episodic multi-agent reinforcement learning with curiosity-driven exploration. In *NeurIPS*.
[53] Yang Zhou, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. 2022. Evolvable network telemetry at Facebook. In *USENIX NSDI*.
[54] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. 2021. Network planning with deep reinforcement learning. In *ACM SIGCOMM*.

Appendices are supporting material that has not been peer-reviewed.

# A APPENDIX

## A.1 DP-based Algorithm

Algorithm 1 represents the pseudocode of our DP algorithm. The DP algorithm requires the same inputs as the search algorithm, i.e., the original network topology $G^*$, the target network topology $T^*$, and constraints $C^*$. We initialize the actions to do at first. Then we set all the elements of the DP array $f(\vec{V}, a)$ to be infinite, except for $f(\vec{0}, 0)$, which is 0 (Lines 1-4). Then we go through all DP states in ascending order of the total number of actions, $\sum v_i$ (Line 6). We get the predecessor topology $\vec{V^*}$ with the current topology $\vec{V}$ and its last action $a$ (Lines 7-8). Then we enumerate the last action $a^*$ of topology $\vec{V^*}$ and compare $a^*$ with $a$ to update $f(\vec{V}, a)$ according to Equation 7 (Lines 9-13). The auxiliary array $g(\vec{V}, a)$ is updated correspondingly (Line 14). After all the DP states have been visited, the optimal action sequence can be rebuilt by tracking from the target topology backward to the original topology (Lines 15-16).

**Algorithm 1** DP algorithm of Klotski

**Input:** The original network topology $G^*$; the target network topology $T^*$; constraints $C^*$.
**Main routine:**

1: **// Initialize**
2: $A, \vec{V^*} \leftarrow GetActions(G^*, T^*)$
3: $f(\vec{V}, k) \leftarrow \infty, \forall \vec{V}, k$
4: $f(\vec{0}, 0) \leftarrow 0$
5: **// Enumerate states and update**
6: **for** $\vec{V}$ in ascending order of $\sum v_i$ from $\vec{0}$ to $\vec{V^*}$ **do**
7: **for** $a$ in $A$ AND $v_a > 0$ **do**
8: $\vec{V^*} \leftarrow GetV(\vec{V}, a)$
9: **if** $IsAvailable(\vec{V}, C^*) = True$ **then**
10: **for** $a^*$ in $A$ **do**
11: $\delta \leftarrow ComputeCost(a, a^*)$
12: **if** $f(\vec{V^*}, a^*) + \delta < f(\vec{V}, a)$ **then**
13: $f(\vec{V}, a) \leftarrow f(\vec{V^*}, a^*) + \delta$
14: $g(\vec{V}, a) \leftarrow a^*$
15: **// Get the action sequence**
16: $L \leftarrow GetAnswer(f, g, A, G^*, T^*)$
17: **return** $L$

**Subroutines:**
• $GetActions(G, T)$: Generate the set of action types $A$ and actions to do $\vec{V^*}$ given the original topology $G$ and the target topology $T$.
• $GetV(\vec{V}, a)$: Given the topology represented by $\vec{V}$ and the last action $a$, return the vector $\vec{V^*}$ which represents the predecessor topology.
• $ComputeCost(a^*, a)$: Compute the extra cost of action $a$ given the last action $a^*$.
• $IsAvailable(\vec{V}, C^*)$: Check whether the topology represented by $\vec{V}$ satisfies the demand and port constraints $C^*$.
• $GetAnswer(f, g, A, G, T)$: Get the optimal action sequence backwards from the target topology $T$ to the original topology $G$ based on the action set $A$, DP function $f$ and auxiliary array $g$.

**Time complexity analysis.**

Proof of Theorem 1. Given that the target topology is represented by $\vec{V^*} = (v_i^*)$, the number of action types is $|A|$, and the total number of actions to do is $|L|$, the DP function has $|A| \times \prod_i v_i^*$ states in total, where $\sum_i v_i^* = |L|$. According to Jensen's Inequality [20], the number of states $|A| \times \prod_i v_i^*$ satisfies $|A| \times \prod_i v_i^* \leq |A| \cdot (\frac{|L|}{|A|})^{|A|}$.

For each state, $|A|$ predecessor states are visited to decide the final value and one constraint satisfiability checking is performed. As we need to traverse all switches and circuits of the DCN, the time complexity of one constraint satisfiability checking is $\Theta(|S| + |C|)$, where $|S|$ is the number of switches and $|C|$ is the number of circuits. Thus, the time complexity of DP algorithm is $O(|A| \cdot (\frac{|L|}{|A|})^{|A|} \cdot (|A| + |S| + |C|))$. Note that the time complexity is polynomial to $|L|$ because $|A|$, $|S|$, and $|C|$ are constant for each migration task. □

**Algorithm 2** A* algorithm of Klotski

**Input:** The original network topology $G^*$; the target network topology $T^*$; constraints $C^*$.
**Main routine:**

1: **// Initialize**
2: $A, A' \leftarrow GetActions(G^*, T^*)$
3: $Q.put((ComputeH(A'), 0), (G^*, [\,]))$
4: $T_C \leftarrow \emptyset$
5: **while** not $Q.empty()$ **do**
6: **// Get the state with highest priority**
7: $(prio0, prio1), (G, L) \leftarrow Q.top()$
8: **// Check whether the migration is finished**
9: **if** $G = T^*$ **then**
10: **return** $L$
11: **// Enumerate actions and process each search branch**
12: **for** $a$ in $A$ **do**
13: $b \leftarrow GetBlock(A' - L, a)$
14: $G' \leftarrow UpdateTopo(G, a, b)$
15: $L' \leftarrow L + \{b\}$
16: $\vec{V} \leftarrow CompressG(L')$
17: **if** $IsAvailable(\vec{V}, T_C, G', C^*)$ **then**
18: $g \leftarrow ComputeG(L')$
19: $h \leftarrow ComputeH(A' - L')$
20: $Q.put((g + h, prio1 + 1), (G', L'))$

**Subroutines:**
• $GetActions(G, T)$: Generate the set of action types $A$ and actions to do $A'$ given the original topology $G$ and the target topology $T$.
• $GetBlock(L, a)$: Return the first operation block with action type $a$ in $L$.
• $CompressG(L)$: Return the compact topology representation by the finished action sequence $L$.
• $IsAvailable(\vec{V}, T_C, G, C^*)$: Check satisfiability for network topology $G$, represented by $\vec{V}$ with cache table $T_C$ and constraints $C^*$.
• $UpdateTopo(G, a, b)$: Operate operation block $b$ with action type $a$ on topology $G$, and return the updated topology.
• $ComputeG(L)$: Compute the cost of action sequence $L$.
• $ComputeH(A')$: Compute the number of action types in $A'$ as the estimated cost $h(n)$.

## A.2 A* Search Algorithm

Algorithm 2 shows the pseudocode of the A* algorithm. Given the original network topology $G^*$, the target network topology $T^*$, and constraints $C^*$, we initialize the actions to do at the granularity of the operation block, the priority queue $Q$, and the cache table $T_C$

storing the satisfiability checking results for A* search (Lines 1-4). Each element in $Q$ is composed of two pairs, i.e., the priority pair and the state pair. The priority pair contains the cost $f(n)$ and the number of finished actions. The state pair contains the current network topology and the finished action sequence. For each step, the element with the highest priority (i.e., the element on the top) is popped out of the priority queue (Lines 6-7). Then we check if the current topology is the target topology. If it is, the migration is regarded as finished and the corresponding action sequence is returned as the final result (Lines 8-10). If not, we enumerate all the action types and apply each action type to the current topology (Lines 11-15). If the new topology meets all the demand constraints and port constraints (Lines 16-17), we compute its priority, generate the new state, and put the new element into the priority queue $Q$ (Lines 18-20).

**Time complexity analysis.**

PROOF OF THEMOREM 2. Assume that the target topology is represented by $\vec{V^*} = (v_i^*)$, the number of action types is $|A|$, and the total number of actions to do is $|L|$. The time complexity of the A* algorithm $T$ is the product of the time complexity for each state and the number of visited states.

The time complexity for each state is composed of finding successor states and checking the satisfiability of constraints. One state has $|A|$ successor states. Besides, efficient satisfiability checking guarantees that one state is checked for constraint satisfiability at most once. For one satisfiability checking, we need to examine all the switches and circuits, and the time complexity is $\Theta(|S| + |C|)$, where $|S|$ and $|C|$ are the numbers of switches and circuits, respectively. Thus, the time complexity for each state is $\Theta(|A| + |S| + |C|)$.

The number of visited states varies according to the migration task. In the worst case, the A* algorithm has to visit all states to find the optimal solution. According to the Proof A.1, there are up to $(\frac{|L|}{|A|})^{|A|}$ states. Thus, the upper bound of the time complexity $T$ is $O((\frac{|L|}{|A|})^{|A|} \cdot (|A| + |S| + |C|))$. In the best case, the A* algorithm can always visit the optimal action for each state first, i.e., $|L|$ states are visited in all. Thus, the lower bound of the time complexity $T$ is $\Omega(|L| \cdot (|A| + |S| + |C|))$.

In all, the total time complexity $T$ satisfies $T = \Omega(|L| \cdot (|A| + |S| + |C|)) = O((\frac{|L|}{|A|})^{|A|} \cdot (|A| + |S| + |C|))$. □