In the Search of Optimal Tree Networks: Hardness and Heuristics

Pavel Martynov
Jane Street
London, United Kingdom

Maxim Buzdalov
Aberystwyth University
Aberystwyth, United Kingdom

Sergey Pankratov
Institute of Science and Technology
Klosterneuburg, Austria

Vitaliy Aksenov
City St George's, University of London, United Kingdom
ITMO University
Saint Petersburg, Russia

Stefan Schmid
Technical University Berlin
Berlin, Germany

Abstract

Traffic in datacenters may follow some pattern: some pairs of servers communicate more frequently than others. Demand-oblivious networks may perform poorly for such workloads, and demand-aware networks optimized for traffic should be used instead. Unfortunately, not all shapes of networks are feasible in real hardware. Practical limitations are usually provided in the form of a topology. For example, a network may be required to be a binary tree, a bounded-degree graph or a Fat tree.

In this work, we consider a topology of a binary tree, one of the most fundamental network topologies. We show that already finding an optimal demand-aware binary tree network is NP-hard. Then, we explore how various optimization techniques, including simple local searches, as well as deterministic mutation and crossover operators, cope with generating efficient tree networks on real-life and synthetic workloads.

CCS Concepts

• **Computing methodologies** → **Discrete space search**; • **Mathematics of computing** → **Graph algorithms**.

Keywords

demand-aware networks, binary trees, NP-hardness, heuristics

ACM Reference Format:
Pavel Martynov, Maxim Buzdalov, Sergey Pankratov, Vitaliy Aksenov, and Stefan Schmid. 2025. In the Search of Optimal Tree Networks: Hardness and Heuristics. In *Genetic and Evolutionary Computation Conference (GECCO '25), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3712256.3726425

1 Introduction

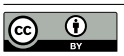
Modern datacenters serve huge amounts of communication traffic which impose stringent performance requirements on the underlying network. Most of these datacenters are designed for uniform (all-to-all) traffic independently of the actual traffic patterns they serve, and typically rely on a fat-tree topology [31].

This paper explores an alternative design which has recently received attention: *demand-aware* networks, that is, networks whose topology is optimized toward the traffic. These are made possible by the recent advances in optical technologies [21, 24, 25, 32] that enabled easy reconfiguration of physical network topologies. See [8] for an algorithmic taxonomy of the field.

In terms of topology efficiency, most existing algorithms provide only approximated solutions [5–7, 22, 40]. The research on the optimal demand-aware network topologies is currently limited. We are aware of only two results providing optimal demand-aware networks. The first one [23] shows that the problem is NP-hard when a demand-aware network should be a graph of degree two, i.e., a line or a cycle. The second one [39] shows that a problem for a binary *search* tree topology can be solved in polynomial time. This topology requires the following property: all nodes in the left subtree have smaller identifiers than the root, while those in the right subtree have larger identifiers. The construction algorithm is polynomial and employs dynamic programming.

Given the limited existing work in this area, we decided to investigate the generic binary tree topology as the next step. This topology is particularly appealing due to three main advantages. First, each node only needs to maintain three connections. Second, the network becomes planar. Finally, routing decisions are much simpler compared to more complex topologies, such as Fat Trees. In modern data centers, these advantages are less relevant because nodes typically support a large number of simultaneous connections and are powerful enough to make routing decisions quickly. However, in some uncommon cases, the binary tree topology can still offer benefits. For example, it can be useful for coordinating unmanned aerial vehicles [48] and for low-cost wireless mesh connections [28], where low-degree nodes and a planar topology are needed. Simple routing is also essential when using very basic FPGAs as routing nodes [45].

By that, we continue the discussion on whether we can construct the best (or a reasonably good) network under some constraints, e.g., bounded-degrees, given the distribution of requests over some time, for example, a day or a month. Since usually the load in the network is periodic, optimizing the network this way could lead to the better network utilization, e.g., reducing the load over the edges, and, thus, could improve the performance of the communication-heavy computations in a datacenter, e.g., all-to-all communications in a learning process of a machine learning model [49] or a MapReduce computation [1].



GECCO '25, July 14–18, 2025, Malaga, Spain

ACM ISBN 979-8-4007-1465-8/2025/07
https://doi.org/10.1145/3712256.3726425

Our contribution. We show that unlike the search tree variant, the problem of finding an optimal binary tree is NP-hard. Then we propose a set of optimization heuristics consisting of various problem-dependent initialization, mutation and crossover operators, some of which are based on the algorithm from [39], and some introduce entirely different ideas, such as the maximum spanning tree construction heuristic. These operators are then employed within a gray-box-like [46] optimization framework. We show that our mutation-based heuristics work well on synthetic and real loads: our generated binary tree networks outperform the binary search tree networks constructed over random permutations of vertices, ranging from 2% to 2.8× of improvement. The studied crossover operators, however, did not result in any additional improvement.

Roadmap. In Section 2, we state the problem and discuss optimization algorithms. In Section 3, we prove that the problem is NP-hard. In Section 4, we propose components that can be used to implement local search with restarts, and show that these can already find good enough solutions. Section 5 deals with our efforts to design crossover operators, all of which appear to be of little use for this problem. We conclude with Section 6.

2 Background

Demand Matrix. The *demand* of a network refers to the pattern of usage and traffic on the network over a certain period of time. It is a characterization of the amount and type of data that is transmitted across the network at different times of the day, week, or month.

The demand is affected by various factors such as the type of applications being used, the time of day, etc. The nature of the demand for a network can vary widely depending on the specific network and its usage patterns [3, 4, 18, 37].

In our chosen theoretical model, *demand*, or *load*, can be defined as a square symmetrical *demand matrix* W of size $n \times n$ where n is the number of hosts in the network. An integer W_{ij} numerically denotes the amount of traffic between nodes i and j of the network — we can think of it as the frequency of communication between these two nodes, or the probability of sending a message. In this paper, we abstract from the actual meaning of these numbers.

Static Optimal Networks. A static optimal network is a network that is designed to provide the best possible performance and efficiency for a particular set of conditions, without considering changes in traffic patterns or usage over time.

Finding a static optimal network amounts to minimizing the sum $\sum_{1 \le i,j \le n} W_{ij} \cdot D_{ij}$ where W is the demand matrix and D is a distance matrix in the constructed network, subject to chosen constraints. We refer to the value of the sum above as the *cost* of the network, which is a common model in the literature:

$$C(D, W) = \sum_{1 \le i,j \le n} W_{ij} \cdot D_{ij}. \tag{1}$$

Fitness evaluation. If an algorithm or an operator does not calculate the cost on its own, we use the classic algorithm for finding lowest common ancestors (LCA) [2], which reduces the problem to range minimum queries [9, 10]. After preprocessing in $O(n \log n)$ time and space, for each non-zero demand entry W_{ij} we compute the distance between nodes i and j in the tree by finding their LCA in time $O(1)$. This takes $O(n \log n + m_D)$ time, where m_D is the number of non-zero W_{ij} entries where $i < j$.

Topological constraints. It is obvious that the optimal solution in the absence of any constraints is the complete graph (a clique). However, this topology is not scalable. For scalability reasons, it seems natural to narrow down the possible network topology for our statically optimal network. There exist several standard topologies: a line; a binary tree; a tree topology; and Δ-bounded topology, more specifically, 3-bounded topology.

The algorithm from [39] constructs, for a given permutation of vertex indices $\{1, 2, \ldots, n\}$, the "optimal binary search tree". The name "binary search tree" refers to the construction strategy, which results in construction of a tree that satisfies the search tree property on the chosen order of vertices (that is, all vertices in the left subtree are "less" than this vertex, and all vertices in the right subtree are "greater", with regards to the chosen order). The complexity of this algorithm is $O(n^3)$. Note that this approach would produce the optimal solution, if an appropriate order of vertices is supplied as the permutation. This is, however, sufficiently unlikely for realistic input sizes.

Our problem. In this work, we are interested in demand-aware networks with the binary tree topology. Unlike [39], we do not require an additional search property from the binary tree. We show that a problem to find an optimal binary tree topology is NP-hard and then we present optimization algorithms that achieve better results than the optimal binary search tree.

Gray-box optimizers. When dealing with problems that have a well-understood or transparent structure, one may benefit from specialized operators, either specialized construction heuristics (such as using the algorithm from [39]) instead of random generation, efficient deterministic local search operators [15] or crossovers that constructively recombine best parts of different solutions [38, 44]. These approaches are currently collectively known as gray-box optimization. A related concept is operators that re-evaluate individuals quickly, including mutations [11, 12, 14, 16] and crossovers [13, 35].

When such operators are available, search algorithms tend to converge to a format which amounts to running multiple local searches and recombining local optima using higher-order operators, where a prominent example is the design of the Lin-Kernighan-Helsgaun family of optimizers [26, 44], which, among everything else, allows such optimizers to be competitive in terms of wall-clock running time with other specialized algorithms. For this reason, we will also choose this route from the very beginning. We will first study local searches in the form of combining various initializers and mutations, and then turn to exploring possibilities offered by crossovers, an opportunity which should not be missed [20, 27, 42].

3 NP-hardness

To prove that our problem is NP-hard, we consider a decision version of the problem, which we call the *Optimal Binary Tree Problem*, or OBT, and prove that it is NP-complete; the original version of the problem is obviously not easier. Given the symmetric demand matrix W and the required arranged cost C, the problem is to decide whether there exists a binary tree such that its cost, according to (1), does not exceed C.

As OBT is in NP, we need to reduce another NP-complete problem to it. For that, we choose an NP-complete problem called Simple

Optimal Linear Arrangement Problem (OLA) [23]. Given an undirected graph (V, E) with $|V| = n$ vertices, the problem is to decide whether there exists a bijective function $\phi : V \to \{1, \ldots, n\}$ such that the sum of $|\phi(u) - \phi(v)|$ for each edge $uv \in E$ does not exceed a number W. Now, we give the formal definitions of both problems.

Problem 1 (OBT). *Let $W = [w_{ij}]_{n\times n}$ be a symmetric demand matrix. Let C be the required arrangement cost. For a connected undirected graph G, $D_G(i, j)$ denotes the shortest distance between vertices i and j.*

Question: Does there exist a binary tree graph B, such that

$$\sum\nolimits_{i>j} w_{ij} \cdot D_B(i, j) \leq C?$$

Problem 2 (OLA). *Let $G = (V, E)$ be an undirected graph with $|V| = n$. Let W be the required cost of the bijection. Since G is undirected, we assume that edge uv is the same edge as vu.*

Question: Does there exist a bijective function $\phi : V \to \{1, \ldots, n\}$, such that

$$\sum\nolimits_{uv\in E} |\phi(u) - \phi(v)| \leq W?$$

Now we prove that OBT is NP-complete by reducing OLA to it.

Theorem 3.1. *OBT problem is NP-complete.*

Proof. It is clear that OBT lies in NP. To prove the NP-hardness, we provide a reduction of OLA to OBT where OLA is known to be NP-complete [23].

Consider an instance of OLA: a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ and the required bijection cost W. We build an instance of OBT in the following manner (see a left graph on Figure 1b).

We introduce $n + 2$ new vertices $H = \{h_1, h_2, \ldots, h_{n+2}\}$. Sometimes we refer to a vertex h_i as v_{n+i} and it should be clear from the context.

Then, we introduce two new sets of edges. $L = \{(h_i, h_j) : h_i, h_j \in H; |i-j| = 1\}$ is a set of edges for a line on vertices from H, shown in blue on the left of Figure 1b. And $I = \{(v_i, h_j) : v_i \in V, j \in \{2, n+1\}\}$ is a set of edges connecting vertices from V to the second and penultimate nodes of H, shown in red on the left of Figure 1b.

Also, we introduce a demand matrix W for $2 \cdot n + 2$ vertices in $V \cup H$:

$$W_{ij} = \begin{cases} 1, & \text{for } (v_i, v_j) \in E. \\ d_1 = X + 2 \cdot m + 1, & \text{for } (v_i, h_{j-n}) \in I. \\ d_2 = (n^2 + n + 1) \cdot d_1, & \text{for } (h_{i-n}, h_{j-n}) \in L. \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Finally, we calculate the constant C for our OBT problem.

$$C = (n + 1) \cdot d_2 + n \cdot (n + 1) \cdot d_1 + X + 2 \cdot m \tag{3}$$

Now we show the correctness of our reduction.

OLA $\Rightarrow$ *OBT*: Suppose there exists a bijection function ϕ for G with the cost $\leq X$ then there exists a binary tree with the cost at most C.

We construct binary tree B as shown on the right of Figure 1b: vertices of H form a line and vertices of V are connected to the named line in order ϕ, starting from the second vertex of H. In other words, vertex $v_i \in V$ is connected to vertex $h_{1+\phi(i)}$. Now to show that this arrangement costs at most C:

- the sum of costs for edges in $L = (n + 1) \cdot d_2$: each edge traverses distance of 1 — the adjacent vertices in H are mapped to the adjacent vertices;
- the sum of costs for edges in $I = n \cdot (n + 1) \cdot d_1$: for every $v_i \in V$, $D_B(v_i, h_2) + D_B(v_i, h_{n+1}) = n + 1$;
- the sum of costs for edges in $E \leq W + 2 \cdot m$: each edge (v_i, v_j) traverses distance of $|\phi(i) - \phi(j)| + 2$.

If we sum all these costs we get a value at most C.

OBT $\Rightarrow$ *OLA*: Suppose there exists a binary tree B for matrix W from Formula 2 with the cost of at most C from Formula 3 then there exists OLA with the cost at most W.

(1) Every edge from L has the distance of exactly 1 over B. We prove this by contradiction: if at least one edge in L traverses a distance of at least 2, then, all edges of L traverse the distance of at least $(n+2)$. Thus, the total cost of B is at least $(n+2) \cdot d_2 = (n+1) \cdot d_2 + (n^2+n) \cdot d_1 + W + 2 \cdot m + 1 = C + 1$, which contradicts that the cost is at most C.
(2) As a direct consequence, vertices in H have to form a line segment in B and in the right order.
(3) At most one vertex from V can be adjacent to h_i for $2 \leq i \leq n + 1$. Since B is a binary tree, the maximum degree of h_i in b is 3. By Statement 2, h_i is adjacent to h_{i-1} and h_{i+1}, which means only one vertex from V can be adjacent to h_i in B.
(4) For some v_i, the minimal possible value of $x = D_B(v_i, h_2) + D_B(v_i, h_{n+1})$ is $n + 1$. We have three cases: 1) if v_i is adjacent to h_j $(2 \leq j \leq n + 1)$, then $x = n + 1$; 2) if v_i is adjacent to h_1 or h_{n+2}, then $x = n + 3$; 3) if v_i is not adjacent to any vertex in H, then $x > n + 1$ since B is a tree.
(5) In B every vertex of V is adjacent to one of the vertices in $\{h_2, h_3, \ldots, h_{n+1}\}$. We prove this statement by contradiction: if some vertex v_i is not adjacent to one of vertices from $\{h_2, h_3, \ldots, h_{n+1}\}$, then $x = D_B(v_i, h_2) + D_B(v_i, h_{n+1}) > n + 1$. By Statement 4, all other vertices from V contribute at least $(n + 1) \cdot d_1$ to the cost for edges in I. It follows that the total cost of edges in I is at least $n \cdot (n+1) \cdot d_1 + d_1$. By Statement 1, the total cost of edges in L is $(n+1) \cdot d_2$. Hence, the total cost of B is at least $(n+1) \cdot d_2 + n \cdot (n+1) \cdot d_1 + W + 2 \cdot m + 1 = C + 1$, which contradicts that the cost is at most C.
(6) The arrangement cost of E is

$$\sum\nolimits_{(v_i,v_j)\in E} D_B(v_i, v_j) \leq W + 2 \cdot m.$$

From Statements 1 and 5, the arrangement cost of L and I in B is $(n + 1) \cdot d_2 + n \cdot (n + 1) \cdot d_1$. Subtracting that from C we get the upper bound on the total cost of edges in E.

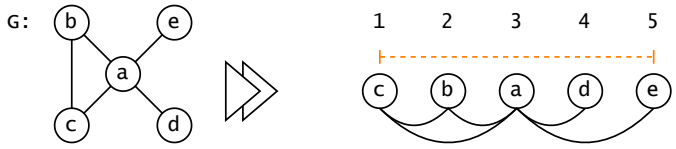
Thus, by Statement 5, every vertex $v_i \in V$ is adjacent to some h_j with $j \in \{2, \ldots, n + 1\}$, and we can define $\phi(i) = j - 1$. ϕ is bijective by Statement 3 and B being a tree. It follows from the above that

$$\sum_{(v_i,v_j)\in E} |\phi(i) - \phi(j)| = \sum_{(v_i,v_j)\in E} (D_B(v_i, v_j) - 2) = \sum_{(v_i,v_j)\in E} D_B(v_i, v_j) - 2 \cdot m.$$

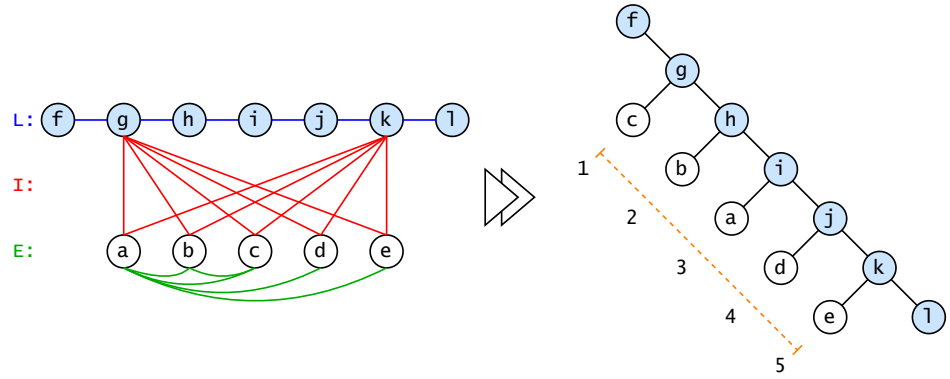
By Statement 6, the cost of ϕ does not exceed W. □

4 Local searches with restarts

In this section, we limit ourselves with optimization algorithms that employ two type of operators: *initializers*, or *zero-arity operators*,



(a) A solution of OLA for graph G and labeling cost $W = 7$. The orange line represents indexing of vertices.



(b) A solution of OBT, constructed from the OLA instance in (1a). Vertices a to e represent V. Vertices f to l represent H and are highlighted with light blue. On the left, edges of E are highlighted with green and have demand 1 each. Edges of I are highlighted with red and have demand $d_1 = 18$. Edges of L are highlighted with blue and have demand $d_2 = 558$. The right graph is a binary tree with cost $C = 3905$ corresponding to the bijection at Figure 1a.

Figure 1: The example of OLA instance and an OBT instance reduced from OLA

that generate new solutions based on the problem instance only, and *mutation operators*, or *unary operators*, that generate new solutions based on one existing solution. Each of these operators may be randomized and, given its inputs, may produce a large number of solutions. We also delegate the responsibility of computing the fitness of the solution to such an operator. More formally, both initializers and mutation operators generate a sequence of pairs $\langle T, F \rangle$, where T is the solution and F is its fitness value, equal to the one computed by (1). Instead of such a pair, an algorithm can return an empty value `null`, which can only happen if either the time limit is exceeded or the search space is explored completely.

An initializer can then be used as an optimization algorithm on its own, similar to random search, with a difference that it can generate something better than just random solutions (for instance, optimal binary search trees over random permutations of vertices). However, we can iteratively improve a solution generated with an initializer by repeatedly applying a mutation operator and rejecting a solution if it becomes worse, which results in a local search algorithm. If the mutation operator can no longer produce more different offspring (i.e. it returns `null`), we may safely restart the optimizer by sampling a new solution with an initializer and beginning anew. We present our operators below.

4.1 Initializers

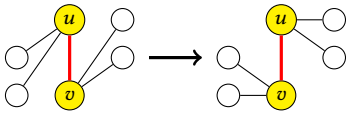
We use three operators as initializers, where two are considered general-purpose initializers suitable for local searches, and one is only treated as a separate optimizer.

Optimal binary search trees, random permutations. To generate a tree, this initializer samples a random permutation over $\{1, 2, \ldots, n\}$ and uses the algorithm presented in [39] to construct using the sampled permutation as the ordering on the vertices. One such invocation requires time $O(n^3)$.

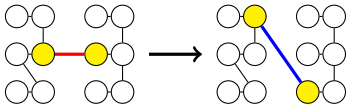
Optimal binary search trees, all permutations. The previous approach can be optimized if the considered permutations are not sampled at random, but rather iterated in the lexicographical order, starting from a random permutation. For two permutations π_1 and π_2, such that π_2 follows π_1 in the lexicographical order, their common prefix is large, which allows to avoid recomputing large parts of data structures used in the algorithm above. This consideration allows to construct optimal trees for all permutations in time $O(n! \cdot n^2)$, starting from a random permutation, as opposed to a more naïve $O(n! \cdot n^3)$ approach. This improvement in performance, noticeable even for partial evaluations, is further assessed in experiments. We use this initializer only on its own.

Maximum spanning tree. The algorithms listed above require time of order $\Omega(n^3)$ to construct even one meaningful approximation, which can be too expensive for large networks. Thus, we need alternative approaches which do not guarantee any kind of optimality, but are still good as a starting point.

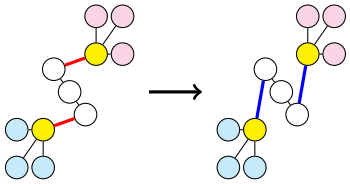
One such approach is to assign some potential value to each potential edge (an unordered pair of vertices) and then to select edges greedily according to this value, while keeping in mind that we want to obtain a tree with a maximum vertex degree of 3: if the next edge connects already connected parts of the tree, or is incident to a vertex with degree 3, it is skipped. Connectivity tests in this case are easily implemented by the Disjoint Set data structure [43], and the overall algorithm resembles Kruskal's algorithm [30] for finding minimum spanning trees, subject to additional degree checks.



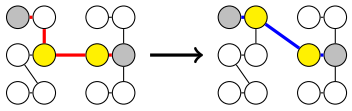
(a) Edge switch



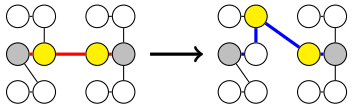
(b) Random edge replacement



(c) The subtree swap operator

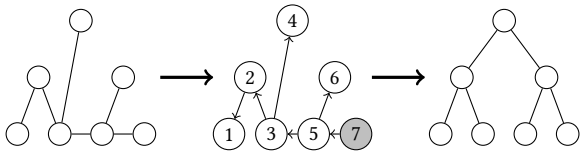


Distance decreases by 1 if the connection moves towards the vertex



Distance increases by 1 if the connection moves away from the vertex

(d) Optimal edge replacement: the main principle



(e) Optimal binary search tree based on random depth-first traversal

Figure 2: Mutation operators employed in this work

Such heuristics consider the demand matrix and treat each entry W_{ij} as the weight of an edge between vertices i and j. In order to minimize the sum of $W_{ij} \cdot D_{ij}$, where D_{ij} is the distance between vertices i and j in the resulting tree, we would aim at decreasing D_{ij} for large W_{ij}. This results in the algorithm for finding the *maximum* spanning tree, with a constraint on the maximum vertex degree. As this problem is NP-hard [34], we only hope for an approximation.

Our algorithm runs in time $O(m_D(\log m_D + \alpha(n)))$. We need $O(m_D \log m_D)$ to sort all the potential edges. Then, we add each edge in disjoint sets with the complexity equal to the inverse Ackermann function $\alpha(n)$, leading to $O(m_D \cdot \alpha(n))$ in total. After this part has concluded, we graph may still not be connected, either because the demand graph is not connected or due to the degree constraints. In this case, we can add arbitrary edges to make a tree, because the particular choice of these edges will not influence the cost of the tree. This part takes at most $O(n)$ time, which does not change the complexity in the real-life case (that is, $m_D \gg n$).

This approach may be able to generate a good starting point quickly. As the demand matrix may contain multiple equal values, we sort the edges once and generate each new maximum spanning tree by first shuffling the edges with equal weights.

4.2 Mutation Operators

We propose four mutation operators: *edge switch*, random and optimal *edge replacement*, *subtree swap* and *optimal BST for random depth-first traversal*.

Edge switch. This operator samples an arbitrary edge (u, v) in the tree. Then, for each vertex i different from v that was adjacent to u, the operator removes an edge (u, i) and adds a new edge (v, i), and similarly, for each vertex j different from u that was adjacent to v, the operator removes an edge (v, j) and adds a new edge (u, j). This way, the endpoints of the edge appear switched. The example execution of this operator is illustrated in Fig. 2a.

Consider two connected components that would appear in the tree if the edge (u, v) was to be removed. Denote as U the component that contains vertex u, and as V the component that contains vertex v. For any pair of vertices $i, j \in U$, the distance between i and j does not change as an effect of this operator, similarly it does not change for any $i, j \in V$. For any $i \in U \setminus \{u\}$ and any $j \in V \setminus \{v\}$, D_{ij} also remains unchanged, and the only changes are: D_{iu} and D_{jv} increase by 1, D_{iv} and D_{ju} decrease by 1.

As a result, this operator can be implemented in time $O(n + \deg(W, u) + \deg(W, v))$ including the recalculation of the cost: the time $O(n)$ is needed to mark the connected components U and V, and then we need to consider only the edges of the demand graph that are adjacent to u and v. This operator is the only one in the paper which requires time less than $O(m_D)$, so we expect it to be able to explore much more solutions compared to other operators.

Since the number of edges to remove is only $n-1$ for a given tree, we can track which edges we have previously tried and give up by returning `null` if all edges have been tried and no improvement has been found, which facilitates restarts.

Edge replacement: random. A mutation which is similarly minor in structure, but more disruptive in the terms of cost changes, is to remove a randomly chosen edge and to connect the two components of the tree with a different edge. The example execution of this operator is illustrated in Fig. 2b. When sampling the replacement edge, we check whether adding this edge violates the degree constraint: as there is always a constant fraction of edges in each component with degree two or less, resampling invalid edges does not affect the running time. After this operator, we evaluate the cost from scratch in $O(n \log n + m_D)$ time. As the number of possible actions is $O(n^3)$, our implementation never gives up sampling.

Edge replacement: optimal. We may employ the structure of our cost function to reconnect the components optimally after an edge is removed. To do this, we consider moving the connection

point of the new edge in one of the components to one of the adjacent vertices (see Fig. 2d). If we split for a moment the vertices of the affected component (the left-hand-side component in Fig. 2d) into those which have their distance to the connection decreased, and those where this distance is increased, we may note that the cost is decreased by the total demand between the vertices from the first group and the entire second component (the right-hand-side component in the figure), and increased by the total demand between the vertices from the second group and the second component.

We can easily maintain this demand change while each connection point traverses its respective component by moving along the edges, as in depth-first search, with a relatively easy preprocessing in $O(m_D + n)$ that amounts to computing the total demand in all subtrees assuming an arbitrary vertex to be chosen as a root. As all the changes require only the demand values, but not the particular distances, we can perform component traversals independently of each other. As a result, in $O(m_D + n)$ time we are able to find the edge that connects the two components remaining after removing any given edge, such that the resulting tree has the minimum possible cost. There are only $n - 1$ possible actions, so we remember which edges have been tested without an improvement and give up if no improvement is possible.

Subtree swap. We also employ a less local mutation operator, subtree swap, which bears the resemblance with crossover and mutation operators used in tree-based genetic programming [29].

In our case, we sample two different vertices v_1 and v_2 that are not adjacent to each other, which will be the root vertices of the subtrees to be swapped. Then, we search for the vertices u_1 and u_2, such that u_1 is adjacent to v_1, u_2 is adjacent to v_2, and the path from v_1 to v_2 in the tree passes through u_1 and u_2. This can be done straightforwardly with a single depth-first search call in time $O(n)$. After that, we remove edges (v_1, u_1) and (v_2, u_2) and add edges (v_1, u_2) and (v_2, u_1). This is illustrated in Fig. 2c. Due to the complexity of determining the exact changes, we compute the cost from scratch in $O(n \log n + m_D)$ time.

For a small number of vertices in the tree, i.e., $n \le 10^3$, we track which pairs of vertices have been tried without an improvement, so that the operator can sample these pairs without replacement and give up when all pairs have been tried.

Optimal BST for random depth-first traversal. Our last operator takes an existing solution, chooses an arbitrary vertex with degree at most 2, and performs a random depth-first traversal of the tree, which then imposes an ordering on vertices. It then runs the algorithm from [39] to construct an optimal binary search tree for this ordering (see Fig. 2e for an illustration). Note that the original solution can also be represented as a binary search tree for this ordering by construction, as a result, the operator never constructs a solution which is worse than the original one.

Unlike the previous operators, this operator is capable of deep structural changes while offering some quality guarantees, so we expect very good results from using this operator. This comes at a price, however, as its running time is $O(n^3)$. For very big graphs we can only afford few invocations, so it is possible that using cheaper operators, on their own or in addition to this operator, can be overall better.

4.3 Experiments

Based on the heuristics outlined in Sections 4.1–4.2, we test the following local search algorithms:

- "MST": repeated calls to the maximum spanning tree heuristic, with edges of equal weights randomly shuffled between the calls.
- "BST/rand": the construction of optimal binary search trees using randomly generated permutations of vertices.
- "BST/next": the construction of optimal binary search trees using lexicographical enumeration of permutations of vertices, starting with a random one.
- algorithms denoted with the notation "$A + B$", where A denotes the initializer used (either MST or BST), and B is the mutation operator. For the basic mutation operators, the names are "switch" for the edge switch mutation, "subtree" for the subtree swap mutation, "replaceR" for the random edge replacement mutation, "replaceO" for the optimal edge replacement mutation, "bstMut" for the BST mutation for a random depth-first traversal.

As B, we also use "random" for a compound operator mutation that on each call chooses one of "switch", "subtree" or "replaceO" operators, subject to being able to sample new individuals, and "replaceOB" that applies "replaceO" while possible and "bstMut" otherwise. As "bstMut" is very expensive compared to other operators, we do not use it as a part of the "random" operator, and in "replaceOB" we resort to "bstMut" only when absolutely necessary.

We thus have 14 algorithms of the form "$A + B$", examples being "MST+random" or "BST+subtree", in addition to three initializer-only algorithms. To investigate their performance, we use two groups of tests, where each test is essentially a demand graph: the synthetic test and the real-world tests.

The synthetic test is adapted from [4] where they test the properties of dynamically adjusting demand-aware networks. They are characterized by the set of possible vertex pairs between which a message is assumed to be passed, and the temporal locality parameter α: the previous request pair is chosen again with probability α while a random pair is chosen otherwise. In the context of this work, the temporal behavior is not considered, so the demand matrix element W_{ij} is equal to the number of messages passed between i and j in the test. In this work, we use tests generated in the same manner with the same number of vertices 1023 and the temporal locality parameter $\alpha = 0.5$, as our preliminary experiments showed no difference in relative performance of considered algorithms for $\alpha \in [0; 0.9]$.

The following six real-world tests, adapted from their respective sources similarly to the above, are used:

- The small and the large test from [37], which we refer to as "Facebook" ($n = 100$ vertices, $m_D = 2990$ edges) and "FacebookBig" ($n = 10000$, $m_D = 151677$).
- Two tests from [24] referred to as "ProjecToR" ($n = 121$, $m_D = 2322$) and "Microsoft" ($n = 100$, $m_D = 1431$).
- "HPC", a test from [18] ($n = 544$, $m_D = 1620$).
- "pFabric", a test from [3] ($n = 100$, $m_D = 4506$).

To run the experiments, we used a computer with an Intel® Core™ i7-8700 CPU clocked at 3.20 GHz, with 12 cores and 64 gigabytes of memory available. The computations were performed in

Table 1: Local searches: median costs (fitness values) and numbers of queries for all tests. The e-notation is used to save space. In each column, grey are the best entry and entries with statistical significance $p \geq 0.05$ in comparison with the best entry, and yellow is the best entry for algorithms using initializers only (first three algorithms).

	Synthetic		Facebook		FacebookBig		HPC		Microsoft		ProjecToR		pFabric	
Algorithm	Cost	Queries	Cost	Queries	Cost	Queries	Cost	Queries	Cost	Queries	Cost	Queries	Cost	Queries
MST	3.09e5	2.93e7	4.20e4	5.11e7	1.32e7	5.37e5	2.83e6	3.74e7	1.96e5	9.18e7	2.15e6	5.87e7	3.94e5	3.34e7
BST/rand	1.23e6	5.44e3	3.78e4	5.85e6	9.54e6	4.00e0	5.48e6	5.02e4	1.90e5	5.36e6	2.04e6	3.04e6	3.53e5	5.57e6
BST/next	1.26e6	8.29e5	3.88e4	6.86e7	9.56e6	7.19e3	5.65e6	2.77e6	2.12e5	7.01e7	2.21e6	4.89e7	3.55e5	6.67e7
MST+switch	3.31e5	4.33e8	4.20e4	3.64e9	1.38e7	3.38e7	2.90e6	9.09e8	1.90e5	4.14e9	2.09e6	3.63e9	3.98e5	3.49e9
MST+subtree	1.89e5	8.11e7	3.64e4	1.32e8	9.82e6	1.83e6	2.13e6	1.18e8	1.76e5	2.71e8	1.88e6	1.70e8	3.54e5	9.07e7
MST+replaceR	1.17e5	6.20e7	3.71e4	1.30e8	1.35e7	1.68e6	2.36e6	9.72e7	1.84e5	2.59e8	1.96e6	1.71e8	3.56e5	9.60e7
MST+replaceO	1.13e5	5.12e7	3.60e4	1.37e8	1.12e7	2.01e6	2.21e6	9.36e7	1.77e5	2.57e8	1.89e6	1.71e8	3.52e5	9.99e7
MST+bstMut	1.10e5	5.18e3	3.55e4	5.64e6	9.35e6	5.00e0	1.93e6	3.94e4	1.76e5	5.08e6	1.88e6	2.97e6	3.49e5	5.66e6
MST+replaceOB	1.10e5	8.54e4	3.55e4	5.83e6	9.17e6	5.87e5	1.94e6	2.19e5	1.75e5	5.46e6	1.87e6	3.00e6	3.48e5	5.59e6
MST+random	1.14e5	8.03e7	3.49e4	1.56e8	9.20e6	2.76e6	2.00e6	1.22e8	1.74e5	3.11e8	1.86e6	2.07e8	3.45e5	1.15e8
BST+switch	1.09e6	2.01e8	3.67e4	2.30e9	9.42e6	4.72e5	5.29e6	2.08e8	1.88e5	1.40e9	2.01e6	1.11e9	3.49e5	2.55e9
BST+subtree	3.12e5	7.57e7	3.59e4	1.29e8	9.38e6	1.47e6	2.85e6	1.12e8	1.77e5	2.65e8	1.89e6	1.70e8	3.48e5	9.13e7
BST+replaceR	2.80e5	6.41e7	3.78e4	1.30e8	9.46e6	1.41e6	3.66e6	1.00e8	1.97e5	2.67e8	2.09e6	1.66e8	3.55e5	9.29e7
BST+replaceO	2.46e5	3.70e7	3.63e4	1.35e8	9.39e6	5.27e5	3.24e6	7.78e7	1.81e5	2.42e8	1.95e6	1.60e8	3.52e5	1.08e8
BST+bstMut	1.10e5	4.77e3	3.56e4	5.74e6	9.51e6	4.00e0	1.92e6	4.01e4	1.76e5	5.19e6	1.88e6	3.10e6	3.48e5	5.56e6
BST+replaceOB	1.10e5	2.45e5	3.55e4	5.87e6	9.35e6	4.81e5	1.93e6	4.50e5	1.76e5	5.56e6	1.88e6	3.08e6	3.48e5	5.51e6
BST+random	1.18e5	8.32e7	3.50e4	1.57e8	9.25e6	2.27e6	2.14e6	1.21e8	1.74e5	3.20e8	1.86e6	2.10e8	3.45e5	1.13e8
Improvement	64.41%		7.47%		3.91%		31.78%		8.45%		8.63%		2.16%	

only five parallel runs, each allocated two cores using the `taskset` utility, to reduce core sharing effects. All algorithms were implemented in Java and ran using the OpenJDK VM version 17.0.13. The code is available in a GitHub repository[1].

For each algorithm and each test we conducted 10 independent runs with a time limit of two hours (7200 seconds). Note that all mentioned algorithms are capable of running for indefinite amount of time, except for "BST/next" which may terminate after testing all permutations of vertices, but none of the tests were sufficiently small for this to happen.

The results are presented in Table 1. We show median costs and number of generated solutions (queries) for each combination of a test and an algorithm. The medians are chosen instead of means, because the distributions of randomized search heuristic outcomes are typically far from being normal [17]. For all comparisons we use an R [36] implementation of the Wilcoxon rank sum [33, 47] non-parametric test. Using it, we now answer some of the research questions.

Does the local search improve the results? By comparing results of "MST"-based operators with "MST", and "BST"-based operators with "BST/rand", in most cases we answer positively at the significance level of $p < 9.1 \cdot 10^{-5}$. There are exceptions though. "MST+switch" is worse in the case of FacebookBig, HPC and the synthetic test, "MST+replaceR" is worse in the case of FacebookBig, "BST+replaceR" is worse in the case of Microsoft and pFabric, all at $p = 5.41 \cdot 10^{-6}$. There are also four less significant comparisons of both signs involving the same mutation operators. This indicates that "switch" and "replaceR" operators may be slow to produce improvements, while initializers produce better solutions quicker by just resampling, with some parallels in theoretical research [19].

Is edge switch faster than other mutations? In most cases, using edge switch resulted in more queries than using other mutations at the significance level of $p = 5.41 \cdot 10^{-6}$, which follows from the ability of edge switch to recompute fitness faster. The only exception is "BST+switch" on FacebookBig, where it is quite common for edge switch to produce less queries. However, this can be easily explained by edge switch getting into the local optimum faster than other mutations, which forces the slow BST initializer to work more often.

BST: random or lexicographical enumeration? The number of queries is always bigger for "BST/next" than for "BST/rand", but the quality is always worse. This is at $p = 5.41 \cdot 10^{-6}$ for all tests except FacebookBig, which shows the same trends, but with a less confident $p < 0.0015$. So, with bigger computational budgets, using "BST/rand" seems preferrable.

Edge replacement: Is optimal better than random? Yes, "BST+replaceO" is better than "BST+replaceR", and "MST+replaceO" is better than "MST+replaceR", at $p < 9.1 \cdot 10^{-5}$.

What to use? For the mutation operators, the obvious choices are the compound operators "random" and "replaceOB", where either of these is always a winner at $p < 0.05$. This indicates that just using a single mutation operator is not enough for the best result, and different neighborhood structures of different mutation operators may help each other to get out of local optima. For the initializers, MST and BST are typically close, but in our experiments MST wins slightly more often.

What are the improvements from using local search? The improvements compared to sampling best initializers for the same computational budget are given in the last row of Table 1.

[1] https://github.com/mbuzdalov/tree-for-network/releases/tag/v1/

5 Crossovers

We also considered using crossover operators in the setup common to gray-box optimization, i.e. to recombine local optima. To that end, we employ the local search algorithm, but also maintain the best solution and, once the restarted local search produces yet another local optimum, we run a crossover operator with the best solution and this local optimum.

Meaningfully crossing over graphs, even trees, is notoriously difficult [41], especially if constraints are to be satisfied. We tested three crossover operators:

- *Random edge subsampling*. Edges of both parents are combined, and a new solution is constructed by sampling a random edge, checking whether it is possible to add it to the solution while avoiding cycles and vertices of degree greater than 3, and adding it if allowed. It is not always possible to construct a tree in this way, so in the case of failure, we retry for 10 times, and if still failing, we add random allowed edges to the last solution.
- *Greedy edge subsampling*. Similar to the maximum spanning tree initializer, we sort all edges of the demand matrix in the decreasing order, breaking ties at random, and then traverse these edges twice. On the first pass, we only try adding edges that are present in either of the parent. On the second pass, if needed, we consider also edges that are not present in the parents. Finally, if the solution is still not a tree, we add random allowed edges.
- *Partition crossover* (PX). Following the ideas from [38, 44], we take each edge that exists in both solutions to the result and try to find, for each of the remaining connected components, which solution to take all edges from. Unlike in [38, 44], we don't have a polynomial algorithm for finding the optimal recombination quickly, so we perform branch-and-bound instead, which is still often feasible. The underlying optimization problem is NP-hard even in the simple case when the common edges are all connected.

We performed a single run of the resulting algorithm, with a time limit of 600 seconds, for a selection of instances and base local search combinations. We recorded the best individuals produced by crossover operators, and also how the crossover offsping fare compared to their parents: better ($<$), equal to one of the parents ($=$), stricly between the parents ($\in$), or worse than both parents ($>$). For the partition crossover, we also recorded the average number of connected components (CC). Table 2 presents the results.

A quick conclusion from Table 2 is that these crossover operators do not help much. Edge subsampling crossovers are rarely capable of improving over the parents, and if they are, they still do not produce the best result of the run and seem to be able to improve bad solutions only. While it is also clear that the greedy crossover produces better results than the random crossover, neither of these is really usable in the described framework.

The partition crossover is more capable than edge subsampling crossovers, but in most of the cases the number of connected components was 1 or 2, and the chance of producing an improvement was small. For the synthetic test, the number of components was much higher, so we could not finish the single crossover run within the time limit, but what we obtained looks promising.

Table 2: Crossovers: costs and other statistics

Base algorithm	Xover	Best cost		Xover vs parents			
		All	Xover	$<$	$=$	$\in$	$>$
Synthetic							
MST+switch	Random	3.37e5	4.80e5	0	0	10	8775
MST+replaceO	Random	1.15e5	1.96e6	0	0	0	355
BST+switch	Random	1.10e6	2.47e6	0	0	0	377
BST+replaceO	Random	2.63e5	2.27e6	0	0	0	161
MST+switch	Greedy	3.42e5	3.43e5	4	1	4004	4166
MST+replaceO	Greedy	1.14e5	3.58e5	0	0	0	366
BST+switch	Greedy	3.63e5	3.67e5	6	0	364	5
BST+replaceO	Greedy	2.46e5	3.74e5	0	0	0	153
MST+switch	PX	3.91e5	3.91e5	1	0	CC: 107	
MST+replaceO	PX	1.20e5	1.21e5	1	2	CC: 16.3	
BST+switch	PX	1.10e6	–	0	359	CC: 1	
BST+replaceO	PX	2.61e5	2.69e5	3	167	CC: 3.5	
Facebook							
MST+switch	Random	42590	43090	0	13	9.6e4	2.2e5
MST+replaceO	Random	36093	44054	0	0	0	1.9e4
BST+switch	Random	36804	43682	0	0	0	2.4e5
BST+replaceO	Random	36446	46787	0	0	0	2.5e4
MST+switch	Greedy	41311	41476	5	14	1.4e5	1.7e5
MST+replaceO	Greedy	36134	41217	0	0	0	2.0e4
BST+switch	Greedy	36928	40428	0	0	0	2.3e5
BST+replaceO	Greedy	36477	44566	0	0	0	2.5e4
MST+switch	PX	39245	39245	68	3.3e5	CC: 1.4	
MST+replaceO	PX	35992	35992	6	2.0e4	CC: 1.0	
BST+switch	PX	36789	36789	19	2.5e5	CC: 1.0	
BST+replaceO	PX	36566	–	0	2.5e4	CC: 1.0	
Microsoft							
MST+switch	Random	1.90e5	1.91e5	1	0	179	6.3e5
MST+replaceO	Random	1.78e5	1.88e5	0	0	0	3.9e4
BST+switch	Random	1.89e5	2.09e5	0	0	1	3.4e5
BST+replaceO	Random	1.82e5	2.05e5	0	0	0	3.0e4
MST+switch	Greedy	1.90e5	1.91e5	4	53	8.5e4	4.5e5
MST+replaceO	Greedy	1.77e5	1.85e5	0	0	0	3.8e4
BST+switch	Greedy	1.88e5	1.89e5	2	9	5.8e4	2.7e5
BST+replaceO	Greedy	1.81e5	1.88e5	0	1	3.0e3	2.7e4
MST+switch	PX	1.86e5	1.86e5	104	6.6e5	CC: 1.4	
MST+replaceO	PX	1.77e5	1.77e5	7	4.0e5	CC: 1.1	
BST+switch	PX	1.88e5	1.88e5	34	3.4e5	CC: 1.0	
BST+replaceO	PX	1.82e5	–	0	3.2e4	CC: 1.0	

6 Conclusion

We showed that the construction of an optimal demand-aware binary tree network is NP-hard, and that gray-box local search optimizers are capable of producing significant improvement over existing algorithms on synthetic and real-world workloads.

The future work may include investigation of other (less local) mutation and crossover operators, including a closer inspection of possibilities offered by partition crossover. Finally, we would like to generalize our algorithms for k-ary trees and more complicated networks with bounded degrees.

Acknowledgments. Research was supported by the German Research Foundation (DFG), grant 470029389 (FlexNets).

References

[1] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and T. N. Vijaykumar. 2013. MapReduce with communication overlap (MaRCO). *J. Parallel and Distrib. Comput.* 73, 5 (2013), 608–620.

[2] Alfred Aho, John Hopcroft, and Jeffrey Ullman. 1973. On finding lowest common ancestors in trees. In *Proceedings of 5th ACM Symposium on Theory of Computing*. 253–265.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[4] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. 2020. On the complexity of traffic traces and implications. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (2020), 1–29.

[5] Chen Avin, Alexandr Hercules, Andreas Loukas, and Stefan Schmid. 2018. rDAN: Toward robust demand-aware network designs. *Inform. Process. Lett.* 133 (2018), 5–9.

[6] Chen Avin, Kaushik Mondal, and Stefan Schmid. 2020. Demand-aware network designs of bounded degree. *Distributed Computing* 33, 3-4 (2020), 311–325.

[7] Chen Avin, Kaushik Mondal, and Stefan Schmid. 2022. Demand-aware network design with minimal congestion and route lengths. *IEEE/ACM Transactions on Networking* 30, 4 (2022), 1838–1848.

[8] Chen Avin and Stefan Schmid. 2019. Toward demand-aware networking: A theory for self-adjusting networks. *ACM SIGCOMM Computer Communication Review* 48, 5 (2019), 31–40.

[9] Michael Bender and Martín Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*. Number 1776 in Lecture Notes in Computer Science. Springer, 88–94.

[10] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms* 57, 2 (2005), 75–94.

[11] Anton Bouter, Tanja Alderliesten, and Peter A.N. Bosman. 2021. Achieving Highly Scalable Evolutionary Real-Valued Optimization by Exploiting Partial Evaluations. *Evolutionary Computation* 29, 1 (2021), 129–155.

[12] Anton Bouter, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. 2017. Exploiting linkage information in real-valued optimization with the real-valued gene-pool optimal mixing evolutionary algorithm. In *Proceedings of Genetic and Evolutionary Computation Conference*. 705–712.

[13] Maxim Buzdalov. 2023. Improving Time and Memory Efficiency of Genetic Algorithms by Storing Populations as Minimum Spanning Trees of Patches. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*. 1873–1881.

[14] Maxim Buzdalov and Benjamin Doerr. 2017. Runtime analysis of the $(1 + (\lambda, \lambda))$ genetic algorithm on random satisfiable 3-CNF formulas. In *Proceedings of Genetic and Evolutionary Computation Conference*. 1343–1350.

[15] Francisco Chicano, Darrell Whitley, and Andrew M. Sutton. 2014. Efficient identification of improving moves in a ball for pseudo-Boolean problems. In *Proceedings of Genetic and Evolutionary Computation Conference*. 437–444.

[16] Kalyanmoy Deb and Christie Myburgh. 2017. A population-based fast algorithm for a billion-dimensional resource allocation problem with integer variables. *European Journal of Operational Research* 261, 2 (2017), 460–474.

[17] Joaquin Derrac, Salvador Garcia, Daniel Molina, and Francisco Herrera. 2011. A Practical Tutorial on the Use of Nonparametric Statistical Tests as a Methodology for Comparing Evolutionary and Swarm Intelligence Algorithms. *Swarm and Evolutionary Computation* 1, 1 (2011), 3–18.

[18] Department of Energy, US 2016. *Characterization of the DOE Mini-apps*. Department of Energy, US. https://crd.lbl.gov/divisions/amcr/computer-science-amcr/cag/research/past-research/characterization-of-doe-mini-apps-draft/

[19] Benjamin Doerr and Carola Doerr. 2016. The Impact of Random Initialization on the Runtime of Randomized Search Heuristics. *Algorithmica* 75, 3 (2016), 529–553.

[20] Benjamin Doerr, Daniel Johannsen, Timo Kötzing, Per Kristian Lehre, Markus Wagner, and Carola Winzen. 2011. Faster black-box algorithms through higher arity operators. In *Proceedings of Foundations of Genetic Algorithms*. 163–172.

[21] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*. 339–350.

[22] Klaus-Tycho Foerster, Manya Ghobadi, and Stefan Schmid. 2018. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. 89–96.

[23] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. 1976. Some Simplified NP-Complete Graph Problems. *Theor. Comput. Sci.* 1, 3 (1976), 237–267.

[24] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. 2016. ProjecToR: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 216–229.

[25] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. 2014. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 319–330.

[26] Keld Helsgaun. 2000. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *European Journal of Operational Research* 126, 1 (2000), 106–130.

[27] Thomas Jansen and Ingo Wegener. 2002. The analysis of evolutionary algorithms—A proof that crossover really can help. *Algorithmica* 34 (2002), 47–66.

[28] Eun-Seok Kim and Celia A Glass. 2015. Perfect periodic scheduling for binary tree routing in wireless networks. *European Journal of Operational Research* 247, 2 (2015), 389–400.

[29] John R. Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA.

[30] J. B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.* 7, 1 (1956), 48–50.

[31] Charles E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* 100, 10 (1985), 892–901.

[32] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. 2014. Circuit Switching Under the Radar with REACToR. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 1–15.

[33] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18, 1 (1947), 50–60.

[34] Christos H. Papadimitriou and Umesh V. Vazirani. 1984. On two geometric problems related to the traveling salesman problem. *Journal of Algorithms* 5 (1984), 231–246.

[35] Erik Pitzer and Michael Affenzeller. 2021. Cheating Like The Neighbors: Logarithmic Complexity For Fitness Evaluation In Genetic Algorithms. In *Proceedings of IEEE Congress on Evolutionary Computation*. 1431–1438.

[36] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 123–137.

[38] Danilo Sipoli Sanches, Darrell Whitley, and Renato Tinós. 2017. Improving an exact solver for the traveling salesman problem using partition crossover. In *Proceedings of Genetic and Evolutionary Computation Conference*. 337–344.

[39] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. 2016. SplayNet: Towards Locally Self-Adjusting Networks. *IEEE/ACM Transactions on Networking* 24, 3 (2016), 1421–1433. doi:10.1109/TNET.2015.2410313

[40] Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, and Yueping Zhang. 2010. Proteus: a topology malleable data center network. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.

[41] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127.

[42] Dirk Sudholt. 2017. How Crossover Speeds up Building Block Assembly in Genetic Algorithms. *Evolutionary Computation* 25, 2 (2017), 237–274.

[43] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31 (1984), 245–281.

[44] Renato Tinós, Darrell Whitley, and Gabriela Ochoa. 2020. A New Generalized Partition Crossover for the Traveling Salesman Problem: Tunneling between Local Optima. *Evolutionary Computation* 28, 2 (2020), 255–288.

[45] Kizheppatt Vipin. 2019. AsyncBTree: Revisiting Binary Tree Topology for Efficient FPGA-Based NoC Implementation. *International Journal of Reconfigurable Computing* 2019, 1 (2019), 7239858.

[46] Darrell Whitley, Francisco Chicano, Gabriela Ochoa, Andrew M. Sutton, and Renato Tinós. 2019. Next generation genetic algorithms. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*. 1113–1136.

[47] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[48] Daifeng Zhang and Haibin Duan. 2019. Switching topology approach for UAV formation based on binary-tree network. *Journal of the Franklin Institute* 356, 2 (2019), 835–859.

[49] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. 2017. ZipML: Training linear models with end-to-end low precision, and a little bit of deep learning. In *International Conference on Machine Learning*. PMLR, 4035–4043.