JOURNAL OF Optical Communications and Networking

Recursive first fit: a highly parallel optimal solution to spectrum allocation

GEORGE N. ROUSKAS* AND CHAITANYA BANDIKATLA

Department of Computer Science, North Carolina State University, Raleigh, North Carolina 27695, USA
**Corresponding author: rouskas@ncsu.edu*

Received 8 October 2021; revised 30 December 2021; accepted 31 December 2021; published 8 February 2022

We revisit the classical spectrum allocation (SA) problem, a fundamental subproblem in optical network design, and make three contributions. First, we show how some SA problem instances may be decomposed into smaller instances that may be solved independently without loss of optimality. Second, we prove an optimality property of the well-known first-fit (FF) heuristic. Finally, we leverage this property to develop a recursive and parallel algorithm that applies the FF heuristic to find an optimal solution efficiently. This recursive FF algorithm is highly scalable because of two unique properties: (1) it completely sidesteps the symmetry inherent in SA and hence drastically reduces the solution space compared to typical integer linear programming formulations, and (2) the solution space can be naturally decomposed in non-overlapping subtrees that may be explored in parallel almost independently of each other, resulting in faster than linear speedup. © 2022 Optica Publishing Group

https://doi.org/10.1364/JOCN.445492

1. INTRODUCTION

Spectrum/wavelength allocation (SA/WA) is a problem underlying a range of optical network design problems [1], including routing and WA (RWA) [2–5], routing and SA (RSA) [6,7], traffic grooming [8,9], and network survivability [10]. The SA and WA problems may be tackled along with routing in an integrated manner; various such one-step RWA/RSA approaches have been developed and are discussed, e.g., in [2,6,7]. In this work, we assume that routing and spectrum/wavelength allocation are two separate steps: a path is first selected for each optical connection, and the list of paths is passed to a spectrum/wavelength assignment algorithm that is responsible for allocating spectrum or wavelength resources to each path.

The SA and WA problems are NP-hard [11], and hence, there are no polynomial-time algorithms that can optimally solve general instances of the problems. Consequently, since the early days of optical network research, a wide range of heuristic algorithms have been developed, including first-fit (FF), best-fit, most-used, and least-loaded [12], to select which wavelength or spectrum slots to assign to each traffic demand. These heuristics represent a variety of design choices in terms of algorithmic complexity and the amount of network state information considered. FF, one of the earliest and simplest heuristics that requires no global knowledge, has been shown to perform well across various network topologies and sets of traffic demands [2,13] and is one of the most commonly used algorithms for spectrum/wavelength assignment.

Beyond heuristics, numerous integer linear programming (ILP) formulations for the SA and WA problems have been developed, typically as part of ILPs for the more general RSA and RWA problems. However, these formulations suffer from a serious challenge that relates to spectrum/wavelength *symmetry* [14]. As explained in [15] with reference to the RWA problem, "as the wavelengths are interchangeable, given an optimal solution of the RWA problem or of one of its continuous relaxation, one can derive a large number of equivalent solutions using any permutation of the wavelengths." In other words, in a network with W wavelengths, an ILP solver will have to evaluate all $W!$ distinct optimal solutions, and hence, its running time can be prohibitively, albeit unnecessarily, long. Since blocks of contiguous spectrum slots are also interchangeable (for instance, a request for two slots may be assigned to slots 1 and 2, or 2 and 3, or 3 and 4, and so on), ILP solvers face the same symmetry challenge in tackling the SA problem.

In earlier work [16], we developed an ILP formulation based on maximal independent sets (MIS) for the RWA problem in rings. This formulation does not suffer from symmetry and can be used to obtain optimal solutions to maximum size SONET rings with any number of wavelengths in seconds, an improvement of several orders of magnitude over conventional ILP formulations. However, MIS-based formulations may not be applied to general topology networks of realistic size, as the number of variables increases exponentially with the network size. In this work, we present a solution that overcomes the symmetry challenge in networks of general topology; to the best of our knowledge, such a solution has eluded the research community until now.

As another consideration, handling routing and spectrum/wavelength allocation as separate and sequential steps in network design may lead to suboptimal or even infeasible solutions [2]. Motivated by the observation that many network design problems encompass two tasks, routing and resource allocation, recently we have shown [17] that it is possible to optimally decouple these two aspects and tackle each separately. Accordingly, we developed a recursive algorithm to search the routing space exhaustively yet efficiently. This work complements our earlier results in [17] by developing an optimal recursive algorithm for the SA problem.

The remainder of the paper is organized as follows. In Section 2, we define the SA problem we consider in this work and show how large problem instances may be decomposed optimally into smaller ones. In Section 3, we prove an optimality property of the FF heuristic, and in Section 4, we leverage this property to develop an optimal recursive algorithm for the SA problem. In Section 5, we explain how to use multiple threads to parallelize the execution of the algorithm. We evaluate the algorithm in Section 6, and we conclude the paper in Section 7.

2. SPECTRUM ALLOCATION PROBLEM

We consider an optical network with a topology described by graph $G = (V, A)$, where V is the set of vertices (nodes), and A is the set of arcs (directed fiber links) in the network. Let $N = |V|$ be the number of nodes and $L = |A|$ be the number of directed links; without loss of generality, we assume that if there is a fiber link from some node A to some other node B in the network, then there is a fiber link in the opposite direction, from node B to node A. We are given a set $\mathcal{T} = \{T_i\}$ of traffic requests, and each request is a tuple $T_i = (s_i, d_i, p_i, t_i)$, where

- s_i and d_i are the source and destination nodes, respectively, of the request,
- p_i is the (fixed and pre-determined) physical path between nodes s_i and d_i in the network over which the request must be routed, and
- t_i is the amount of spectrum (e.g., in units of spectrum slots) required to carry the traffic from s_i to d_i.

We consider the following basic definition of the SA problem.

Definition 2.1 (SA). Given a graph $G = (V, A)$ and a set $\mathcal{T} = \{T_i = (s_i, d_i, p_i, t_i)\}$ of traffic requests, assign t_i spectrum slots along the physical path p_i for each request T_i so as to minimize the total amount of spectra used on any link in the network, under three constraints: (1) each request T_i is assigned a block of t_i contiguous spectrum slots (contiguity constraint), (2) each request is assigned the same block of spectrum slots along all links of its path p_i (spectrum continuity constraint), and (3) requests whose paths share a link are assigned non-overlapping spectrum slots (non-overlapping spectrum constraint).

In earlier work [11], we showed that the SA problem is NP-hard even for chain (i.e., single-path) networks with four or more links. When all the spectrum demands are equal, i.e., $t_i = t \forall i$, the SA problem reduces to the WA problem, which can be solved in polynomial time for chain networks but remains NP-hard for rings or networks of a general topology [18]. In the next subsection, we show that under certain conditions, a large SA problem instance may be decomposed into smaller instances that can be solved independently.

A. Exact Decomposition

Consider a request set $\mathcal{T}$ that can be partitioned into, say, two sets $\mathcal{T}_1$ and $\mathcal{T}_2$, such that the paths of requests in $\mathcal{T}_1$ use only links in set $A_1 \subset A$, the paths of requests in $\mathcal{T}_2$ use only links in set $A_2 \subset A$, and the corresponding link sets are disjoint, i.e., $A_1 \cap A_2 = \emptyset$. In this case, it can be seen that allocation of spectrum to requests in $\mathcal{T}_1$ does not affect the allocation of spectrum to requests in $\mathcal{T}_2$, and vice versa. Therefore, the original SA problem on set $\mathcal{T}$ is decomposed exactly into two smaller SA instances on request sets $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, that may be solved independently; the solution to the original problem is simply the maximum of the solutions to the two smaller instances.

Algorithm 1 uses up-tree structures and *Union-Find* operations [19] to partition the set $\mathcal{T}$ of requests into subsets whose paths use pairwise disjoint sets of links. Up-trees are used to represent sets that are pairwise disjoint; the *Find* operation is used to determine the set to which an element belongs, while the *Union* operation performs the union of two disjoint sets. The algorithm starts by creating singleton sets ℓ_j, each consisting of one network link $l_j \in A$. The algorithm then considers the requests in $\mathcal{T}$ one by one, in arbitrary order. For each request T_i, it performs a *Find* operation on each link l_j of the path p_i of T_i to locate the up-tree to which link l_j belongs; initially, the up-tree is the singleton set ℓ_j. Then, the algorithm forms the *Union* of the up-trees to which the links of T_i belong. As a result, at the end of line 11, the non-empty up-trees represent a partition of the link set A such that each link subset (i.e., up-tree) corresponds to a subset of the request set $\mathcal{T}$ whose paths use only links in that up-tree.

Algorithm 1. Request Partition Algorithm

```
Input:
    $G = (V, A)$: network topology
    $\mathcal{T} = \{T_i = (s_i, d_i, p_i, t_i)\}$: set of traffic requests
Output:
    A partition of $\mathcal{T}$ into subsets that pairwise use disjoint sets of links
1:  {Make a singleton set for each link}
2:  for each link $l_j \in A$ do
3:      $\ell_j = \{l_j\}$;
4:  end for
5:  for each $T_i \in \mathcal{T}$ do
6:      {include all links of the path into the same up-tree, i.e., link subset}
7:      $F_i \leftarrow \emptyset$
8:      for each $l_j \in p_i$ do
9:          $F_i \leftarrow Union(F_i, Find(l_j))$
10:     end for
11: end for
12: for each distinct non-empty subset $F_i$ do
13:     return the set of requests with paths using links in $F_i$
14: end for
```

Each *Union* operation takes $O(1)$ time, while each *Find* operation takes time that is logarithmic in the number of singleton sets [19], which in this case is equal to the number L of links. Therefore, the computational complexity of Algorithm 1 is determined by the **for** loop in lines 5–11 and is $O(KL\log(L))$, where K is the number of requests in $\mathcal{T}$, and L is the number of links in the network.

Without loss of generality, in the remainder of this paper, we assume that the request set $\mathcal{T}$ cannot be further decomposed into smaller independent request sets using Algorithm 1.

3. OPTIMALITY PROPERTY OF THE FF HEURISTIC

Consider the SA problem on graph G and request set $\mathcal{T}=\{T_i, i=1,\ldots,K\}$. Let P be a permutation (i.e., an ordering) of the traffic requests T_i. We say that P is a *partial* (respectively, *complete*) permutation if only a subset of (respectively, all K) requests in $\mathcal{T}$ appear in P. Let $SOL(P)$ denote the solution to the SA problem obtained by the FF heuristic by considering each traffic request in the order implied by the permutation P. If P is a complete permutation, then $SOL(P)$ is a feasible solution to the SA problem, but if P is a partial permutation, then $SOL(P)$ is only a partial solution to the SA problem.

Let OPT denote the objective value of an optimal solution to the SA problem. A lower bound LB on the optimal objective value may be obtained by ignoring any spectrum fragmentation that may result from enforcing the spectrum contiguity and continuity constraints and simply accounting for the fact that each link $l \in A$ must use at least as many spectrum slots as to carry all the traffic demands whose path includes this link:

$$LB = \max_{l\in A}\left\{\sum_{T_i\in\mathcal{T}:l\in p_i} t_i\right\}. \tag{1}$$

Clearly, for any complete permutation P of the traffic requests, we have that

$$LB \le OPT \le SOL(P). \tag{2}$$

We now prove the following optimality property of the FF heuristic with respect to the SA problem.

Lemma 3.1 (FF Optimality Property). There exists a permutation $P^\star_{FF}$ of the traffic requests such that applying the FF heuristic to the requests in the order implied by $P^\star_{FF}$ yields an optimal solution to the SA problem, i.e., $SOL(P^\star_{FF})=OPT$.

Proof. By construction.

Consider an optimal solution to the SA problem with an objective value equal to OPT. Label the slots on each link as $1,2,\ldots,OPT$. By definition, the optimal solution is a feasible solution that satisfies all three constraints of the SA problem in that each request T_i is allocated the same block of t_i contiguous spectrum slots on each link along its path p_i, and no other request whose path shares a link with p_i is allocated slots from the same block. Let also f_i denote the slot with the lowest index within the block of t_i slots allocated to request T_i.

Let $P^\star$ be the complete permutation in which the requests T_i are listed in increasing order of f_i in the optimal solution, with ties broken arbitrarily. Consider the block of t_j contiguous spectrum slots allocated to some request T_j by the optimal solution starting at slot f_j. Let us remove this block of t_j slots from the optimal solution. In the remaining partial solution, it is possible that there exists a block of t_j slots that start at a lower indexed slot $f'_j < f_j$ that are available on all links of path p_j. If so, we can allocate the lower-indexed t_j slots starting with slot f'_j to request T_j without affecting the optimality of the solution.

Based on this observation, we modify the optimal solution by considering the requests one by one, in increasing order of f_i as listed in permutation $P^\star$. For each request T_i, we remove its block of spectrum slots that starts at slot f_j from the solution, and we allocate to it an equal block of slots starting at the lowest possible slot index f'_j in the partial solution, keeping in mind that f'_j may be equal to f_j. This modified solution must not use more than OPT slots on any link, since any modifications involve the allocation of lower-indexed spectrum slots to requests. At the same time, since the starting solution is an optimal one, the modified solution may not use fewer than OPT slots on any link. Hence, the modified solution is an optimal one with an objective value equal to OPT. Importantly, by construction, the modified solution is such that no request may be allocated to a spectrum block that starts at a lower-indexed slot.

Let $P^\star_{FF}$ be the complete permutation in which the requests are relabeled so that they are listed in increasing order of f'_i in the modified solution, and let us apply the FF heuristic to this permutation. The FF heuristic allocates to each request T_i a block of t_i contiguous slots starting at the lowest-indexed slot for which such a block is available on all links of path p_i. Therefore, the FF heuristic will construct an optimal solution that is identical to the modified solution above. ■

Figure 1 illustrates the construction proof we described above on a network with four links and eight requests. Figure 1(a) shows an optimal solution in which the requests are labeled $T_1,\ldots,T_8$, in increasing order of their lowest index slot (we assume that spectrum slot indices increase from bottom to top in the figure). This is an optimal solution since the highest assigned spectrum slot on link 3 is equal to the lower bound on the traffic demands. Clearly, none of the requests T_1, T_2, T_3, T_4, and T_7 may be re-assigned to a block of slots with a smaller starting index. However, as shown in Fig. 1(b), request T_5 may be moved to a lower block that starts just above request T_3; request T_6 may be moved to start just above the block of the newly re-assigned request T_3; and request T_8 may be moved to a new block that starts just above request T_1. The new permutation in increasing order of lowest index slot in Fig. 1(b) is T_1, T_2, T_8, T_3, T_4, T_5, T_6, T_7. The FF heuristic produces the optimal solution of Fig. 1(b) on this sequence of requests. (Note that the sub-order of requests with the same lowest slot index has no impact on the solution produced by the FF heuristic, and hence, we could have used the permutation T_1, T_8, T_2, T_3, T_5, T_4, T_6, T_7 instead of the one above.)

This FF optimality property helps explain how so many studies of the SA and WA problems have found the FF heuristic to perform quite well in diverse problem instances. However, Lemma 3.1 constructs a permutation $P^\star_{FF}$ on which

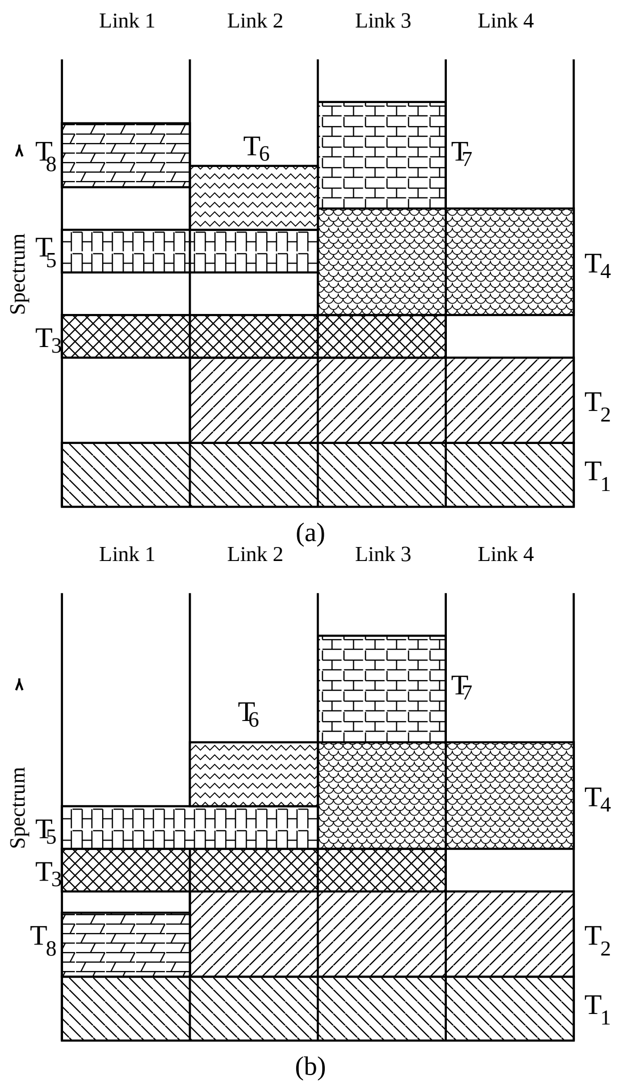

Fig. 1. Proof of the FF optimality property: (a) initial optimal solution and (b) modified optimal solution.

the FF is optimal, but does so by modifying an *unknown* optimal solution, and hence, $P^{\star}_{FF}$ is itself unknown. Nevertheless, the FF optimality property suggests a procedure for finding $P^{\star}_{FF}$: enumerate all permutations of requests, run the FF heuristic on each permutation, and select the one with the smallest objective value. Assuming there is traffic between all node pairs, the size K of the request set is $O(N^2)$, where N is the number of nodes. Therefore, any algorithm that considers all possible permutations of requests to determine the optimal SA must take time that is exponential in the size of the network, $O(N^2!)$.

In the following section, we present a recursive procedure for searching efficiently the space of request permutations to determine this optimal solution.

4. RECURSIVE FIRST FIT

A. Branch-and-Bound Algorithm

We have developed a scalable branch-and-bound recursive FF (RFF) procedure, shown as Algorithm 2, to search the entire

Algorithm 2. Recursive First Fit

```
Input:
    $G = (V, A)$: network topology
    $\mathcal{T} = \{T_i = (s_i, d_i, p_i, t_i\}$: set of traffic requests
    $K = |\mathcal{T}|$: number of traffic requests
    $P_{init}$: initial permutation as discussed in Section 4
    $LB$: the lower bound from expression (1)
    $BestSOL$: best solution so far, initialized to $SOL(P_{init})$
    $BestP$: best permutation so far, initialized to $P_{init}$
Output:
    Best permutation and corresponding SA solution
RFF($P$, $Start$)
$P$: permutation (initial call with $P = P_{init}$)
$Start$: start index of trailing sub-permutation of $P$ that has
not been finalized (initial call with $Start = 1$)
        {Base Case: All $K$ requests finalized in $P$}
        if $Start > K$ then
2:        $S \leftarrow SOL(P)$; {solution obtained by FF on $P$}
          if $S < BestSOL$ then {Update best known solution}
4:          $BestSOL = S$; $BestP = P$;
          end if
6:        return;
        end if
8:      {Main Recursion}
        {Swap $P[Start]$ with all requests that follow it in $P$}
10:     for $k = Start$; $k \le K$; $k{+}{+}$ do
          Swap $P[Start]$ with $P[k]$;
12:     $P_{lead} \leftarrow$ leading permutation $P[1] \cdots P[Start]$;
          {All requests in $P_{lead}$ have been finalized}
14:     $leadSOL \leftarrow SOL(P_{lead})$;
          if  $leadSOL < BestSOL$ then
16:         $newP \leftarrow$ permutation after the swap at Line 11;
$newStart \leftarrow Start + 1$;
18:       RFF($newP$, $newStart$);
        end if
20:     {Restore $P$ and proceed to swap the next request}
        Swap $P[Start]$ with $P[k]$;
22: end for
```

space of request permutations. We start with a complete permutation P_{init} in which the K traffic requests T_i, $i = 1, \ldots, K$ are listed in decreasing order of spectrum demand t_i, and requests with the same demand are listed in decreasing order of path length. We calculate the lower bound LB on the optimal solution OPT using expression (1), and also run the FF heuristic on P_{init} to obtain an initial feasible solution $SOL(P_{init})$, which represents an upper bound on OPT. The algorithm maintains variable $BestSOL$, which indicates the best solution it has found so far; this variable is initialized as $BestSOL = SOL(P_{init})$. Although the recursive procedure will work with any initial complete permutation of requests, our earlier work and other related studies [2,13] indicate that applying the FF heuristic to the requests in the order determined by P_{init} yields better (i.e., lower) solutions that leave a relatively small gap between LB and $SOL(P_{init})$. The RFF procedure then searches the permutation space to find the permutation that yields an optimal solution, as Lemma 3.1 suggests.

Each recursive call takes two arguments: a tentative permutation P and a *Start* index. The recursion builds permutations by maintaining a *Start* index that takes the values $1, 2, \ldots, K$, and divides an input permutation into two parts: a *finalized* leading sub-permutation (prefix) for which the order of requests will not be modified in subsequent recursive calls, and a *tentative* trailing sub-permutation (suffix) for which the order of requests is subject to change and will be finalized by later recursive calls. The *Start* index indicates the start of this trailing sub-permutation. Initially, the ordering of all requests is tentative, and hence, the leading sub-permutation is null and all K requests belong to the trailing sub-permutation. Accordingly, the first call to RFF is with $Start = 1$.

The main recursion is the **for** loop in lines 10–22 of Algorithm 2. Essentially, the **for** loop swaps the first request of the trailing sub-permutation (i.e., the request at index *Start*) with all requests in this trailing sub-permutation, including itself (i.e., requests at index $k = Start, \ldots, K$). After making the swap for one value of k, the procedure updates the permutation (line 16), and increments the *Start* index (line 17) to indicate that the leading sub-permutation of requests whose order has been finalized has increased in size by one. It then makes a recursive call (line 18) to continue swapping requests of the trailing sub-permutation (which has decreased by one). These recursive calls, if allowed to continue without any restriction, will enumerate all possible $K!$ permutations of the K requests.

However, not all permutations will lead to a solution that is better than the currently best known one, *BestSOL*. Therefore, after making a swap and before making a recursive call, in line 14, the algorithm runs the FF heuristic on the leading sub-permutation as it has been expanded after the swap, and compares the result to *BestSOL*. If the result is equal to or higher than *BestSOL*, then it is clear that including more requests to this sub-permutation will produce solutions that are no better than the best one found so far. In other words, continuing further down this subtree of the permutation space is not productive in terms of finding an optimal solution, and this part of the search space can be safely eliminated. Consequently, as shown in lines 15–19, a recursive call is made only if the FF solution on this leading sub-permutation is strictly lower than *BestSOL*.

The base case for the recursion is when the order of all K requests in an input permutation P has been finalized. A complete finalized permutation is indicated whenever the input index $Start > K$, and this case is handled in lines 1–7 of the algorithm. Specifically, the algorithm runs the FF heuristic on P, and if the solution is strictly better than the best known solution, the best solution is appropriately updated in line 4, before the call returns.

We emphasize that, in the worst case, the RFF procedure may be forced to generate all, or close to all, possible permutations of requests and hence take exponential time to complete. In the next section, we show how to speed up the exploration of the permutation space by executing the RFF algorithm in parallel using multiple threads.

B. Illustrative Example

To illustrate the operation of the RFF procedure, consider the set of 26 requests $\{A, B, C, \ldots, Z\}$ labeled with letters of the English alphabet. Figure 2 shows part of the tree of recursive calls made, with the root of the entire tree representing the initial call with arguments $P = \{A, B, C, \ldots, Z\}$ and $Start = 1$. The figure is generated by assuming that the **if** condition in line 15 of the algorithm is not checked, and hence, all recursive calls are made to generate all $26! \approx 4.03 \times 10^{26}$ possible permutations of requests. Also, we use red color to indicate the requests in the tentative trailing sub-permutation and green color to indicate the requests in the finalized leading sub-permutation whose order has been set.

In the initial call, all requests are tentative (and colored red), and the **for** loop in lines 10–22 runs 26 times, each time swapping the first request A of P with each of the requests in the set, $A, B, C, \ldots, Z$, as indicated in the figure. The first of these recursive calls is the root of the leftmost subtree and swaps A with itself; at that point, the order of A becomes fixed (indicated in the figure by a change of color from red to green) and does not change for the remaining recursive calls in this leftmost subtree. The **for** loop of the call representing the root of the leftmost subtree ($Start = 2$) runs 25 times, each time

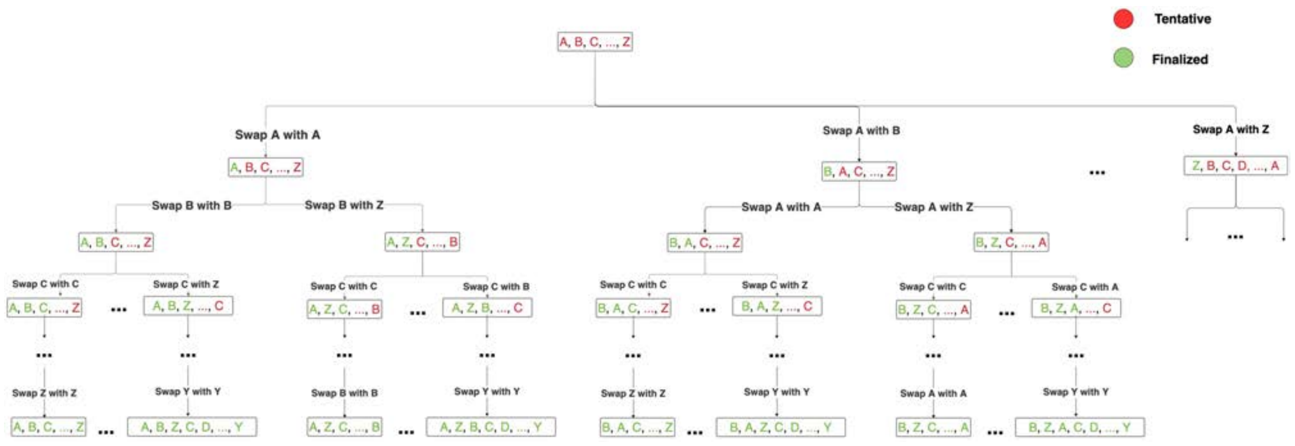

Fig. 2. Partial tree of recursive calls for generating all permutations of the set of requests $\{A, B, C, \ldots, Z\}$ using the RFF recursion. The root of the tree represents the initial call with $P = \{A, B, C, \ldots, Z\}$ and $Start = 1$.

swapping the second request B of the permutation passed to it with each of the requests $B, C, D, \ldots, Z$ in the trailing tentative sub-sequence. This continues recursively until the 25! leaves of this leftmost subtree are reached, each representing one of the 25! possible permutations with request A in the first position of the permutation.

The subtrees of the other 25 children of the root are similar in that each generates the 25! permutations with one of $B, C, \ldots, Z$ in the leftmost position, respectively. For instance, the second of the recursive calls from the root of the whole tree swaps the first request A of P with request B. Subsequent calls in this subtree swap the second request with one of $A, C, D, \ldots, Z$, as before, and so on, until all permutations with B in the leftmost position are generated. Other subtrees of the root are omitted from the figure, except that the figure shows the root of the 26th and final subtree, in which request Z occupies the leftmost position for all permutations generated in that subtree.

C. Implementation Considerations

The RFF algorithm builds a finalized permutation one request at a time. Therefore, when it invokes the FF heuristic in line 14 on the leading sub-permutation P_{lead}, it is not necessary to run the heuristic on the entire sub-permutation. With appropriate bookkeeping (omitted from Algorithm 2 for the sake of clarity and brevity), it is necessary to use FF only to allocate spectrum for just the most recent request added to the leading sub-permutation in line 11 by building upon the solution created by the calling function. Similarly, line 2 of the algorithm does not actually need to run the FF heuristic at all; it can simply reuse the solution of the calling function that finalized the complete input permutation P. This optimization eliminates unnecessary computations and significantly speeds up the running time of the recursion.

Finally, we note that while the RFF algorithm was developed for the basic SA problem of Definition 2.1, it can be adapted to tackle a wide range of SA problem variants that may impose constraints on the spectrum assignment. Constraints on spectrum assignment may be imposed on individual links (e.g., those related to cross talk [20]), sequences of links (e.g., constraints related to attack-aware planning [21], or a combination thereof. To the extent that such constraints disallow certain requests from being allocated consecutive spectrum slots, they reduce the size of the permutation space. Accordingly, the RFF algorithm may be modified to perform appropriate checks while building a permutation incrementally and hence eliminate permutations prematurely if it is so warranted by the constraints.

D. RFF Solution Space, Spectrum Symmetry, and Parallelism

The RFF algorithm searches the request permutation space, but for each permutation, it considers only the spectrum assignment determined by the FF heuristic. Because of the FF optimality property (refer to Lemma 3.1), RFF will identify an optimal solution without the need to consider a different spectrum assignment for any of the various request permutations. Consequently, the RFF algorithm completely sidesteps the symmetry inherent in typical ILP formulations. Such ILP formulations encompass the exponentially large number of equivalent solutions due to symmetry [15], and ILP solvers are forced to explore not only the request permutation space but also the spectrum permutation space.

Commercial ILP solvers employ numerous optimization techniques in addition to the branch-and-bound approach that RFF takes. However, by eliminating symmetry and thus vastly reducing the solution space, our MIS-based formulation of the RWA problem on ring networks [16] outperformed conventional formulations on commercial ILP solvers. RFF accomplishes a similar reduction in the solution space for networks of general topology. While RFF may be further improved by applying optimization techniques used by commercial ILP solvers, we consider such improvements as outside the scope of this work.

A further advantage of the RFF algorithm lies in the fact that the solution space can be naturally decomposed into non-overlapping subtrees that may be explored in parallel almost independently of each other, as we discuss next. Importantly, as we show in Section 6, parallelism leads to faster than linear speedup in the exploration of the solution space.

5. PARALLEL EXECUTION OF THE RFF ALGORITHM

Returning to Fig. 2, which shows the tree of recursive calls for generating all permutations, we observe that sequences of calls that belong to non-overlapping subtrees do not interact with each other and hence may be executed in parallel. In the RFF algorithm, however, recursive calls in non-overlapping subtrees are not completely independent of each other: to eliminate the exploration of permutations that do not lead to better solutions, each call checks and possibly updates the value of variable *BestSOL* as shown in Algorithm 2. Therefore, by locking access to this variable, threads responsible for non-overlapping subtrees may execute in parallel and will interfere with each other only in accessing or updating the value of *BestSOL*.

Let us assume that there are M threads available to be executed in parallel, where the value of M depends on the availability of computing resources such as cores or processors. Note that, since the number of requests K is $O(N^2)$, we expect that $M \ll K$ for nation- or continental-scale networks. One straightforward way to parallelize the execution of RFF is to assign each of the M threads to a different (first-level) subtree of the root, e.g., the M subtrees from left to right in Fig. 2, or in any order since each subtree explores a different part of the permutation space. Upon termination, each of the M threads will have generated all permutations with the request corresponding to that subtree in the first position. At the time a thread terminates, a new thread is spawned and assigned to one of the subtrees of the root that have not been explored yet. This process may continue until either all subtrees have been explored or a time limit has been reached. In the former case, RFF will return the optimal solution; in the latter case, the best solution returned by RFF may or may not be an optimal one.

Clearly, other parallelization options are possible. For instance, one might assign threads to subtrees at the second or third level from the root; doing so would speed up the exploration of the corresponding first-level subtree of the root.

To gain insight on how to best utilize multiple threads to explore the extremely large permutation space, we ran the RFF algorithm in a single thread (i.e., no parallelism) and recorded the times at which the algorithm found better solutions (i.e., the times at which line 4 of Algorithm 2 was executed). Note that a single thread starts at the root node of Fig. 2 and explores the tree in a depth-first search (DFS) manner. Specifically, it follows the leftmost path in the tree until it either reaches the leftmost leaf or eliminates the rest of the path, and then it backtracks to the previous node and starts exploring the leftmost unexplored path from that node in a similar manner.

Figure 3 shows the improvement in the solutions found by the RFF algorithm as a function of how long the algorithm has run, starting from the FF solution it receives as input at time $t = 0$ until we terminate the algorithm after 5 h (note that the time axis is not in linear scale). (Please refer to the next section where we discuss the two network topologies we consider, NSFNET and GEANT2, and the traffic distributions we use to generate random SA problem instances and we explain how we derive the normalized solutions shown in the figure.) We show two problem instances, one for NSFNET and one for the larger GEANT2, for which RFF finds a solution that is better than FF but is higher than the lower bound (hence, the algorithm runs for the full 5 h). It takes less than 5 s (respectively, 45 s) for RFF to find the best solution in the case of NSFNET (respectively, GEANT2); in the remaining time, the algorithm explores solutions that are not better than the best one found in the first few seconds.

The trends in Fig. 3 are very similar to ones we observed for all instances of the corresponding networks and can be explained by two observations. First, the RFF algorithm starts with an initial permutation in which the traffic requests are sorted in decreasing order of spectrum demand. As we mentioned in Section 4, such a permutation has been shown to work well for the FF heuristic, and hence, as expected, the RFF heuristic is able to find good solutions quickly while searching the solution space around this permutation. Second, since a single thread traverses the tree in Fig. 2 in a DFS manner, it spends the 5 h exploring the solution space just around this initial permutation. Consequently, it misses any better solutions that exist in parts of the solution space that are far from the left part of the tree.

These observations indicate that using the whole computational budget in exploring the same part of the permutation space is not effective. A better search approach would be to seek solutions across the entire solution space by (1) dividing the permutation space into multiple non-overlapping parts, and (2) exploring each part for a short amount of time, as Fig. 3 suggests.

To this end, we use multi-threading to equally divide the computational budget among the various parts of the optimization space defined by the first-level subtrees of the root in Fig. 2. Suppose that we have K requests (and hence, K first-level subtrees), M threads that may run in parallel, and a time budget of S time units. In this case, we need $B = \lceil K/M \rceil$ batches of M threads to cover all subtrees of the root. Therefore, we spawn M threads and assign them the M leftmost subtrees of the root. We let these threads run in parallel for S/B time units, at which time we terminate them. We then spawn M new threads to which we assign the next M unexplored leftmost subtrees of the root, let these run in parallel for S/B time units, terminate them, and spawn another set of M threads. We continue in this manner until we have explored each of the subtrees of the root for exactly S/B time units.

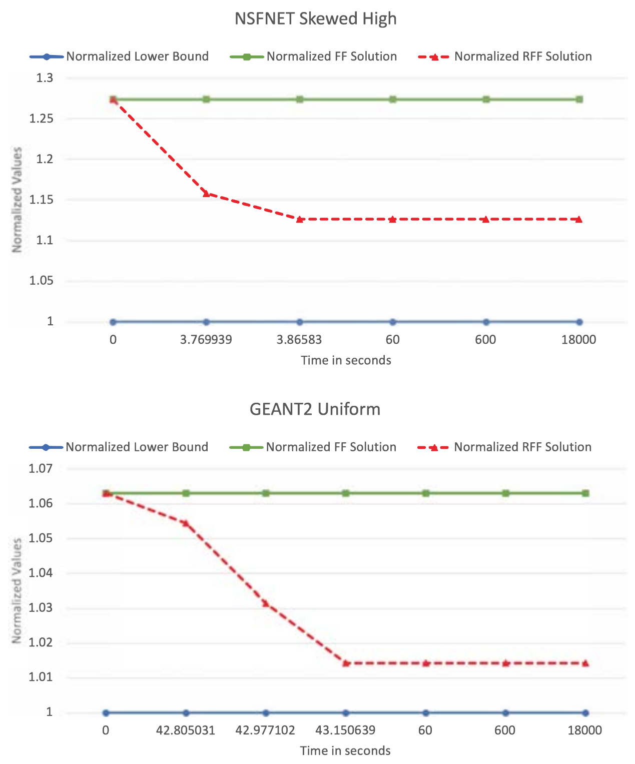

Fig. 3. Improvement of RFF solution versus time, no parallelism (single-thread execution); time axis not in linear scale.

6. NUMERICAL RESULTS

A. Simulation Setup

We now present the results of simulation experiments to evaluate the performance of the RFF algorithm. In our study, we have considered the two network topologies shown in Fig. 4, namely, the 14-node, 21-link NSFNET and the 32-node, 54-link GEANT2 network. We consider shortest-path routing, and we use Dijkstra's algorithm to find the shortest path (with ties broken arbitrarily) between all source–destination pairs. We assume that a traffic request may take one of five possible data rates, namely, 10, 40, 100, 400, or 1000 Gbps. We also assume that each spectrum slot has a width of 12.5 GHz, and we adopt the parameters of [22] to determine the number of spectrum slots required for each traffic request based on its data rate and path length.

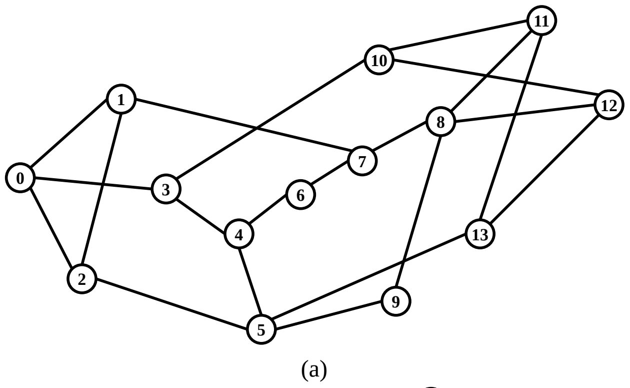
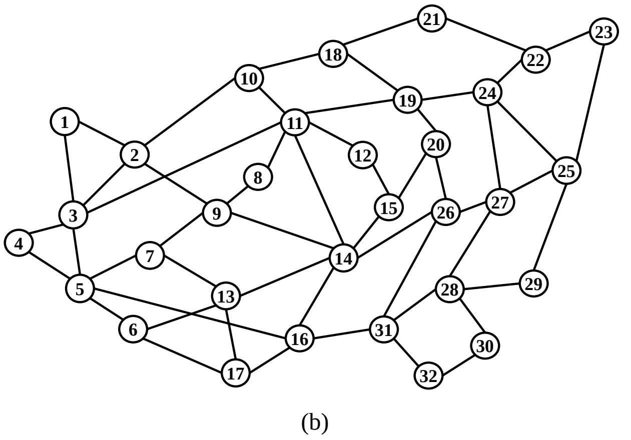

Fig. 4. Network topologies used in our study: (a) NSFNET and (b) GEANT2.

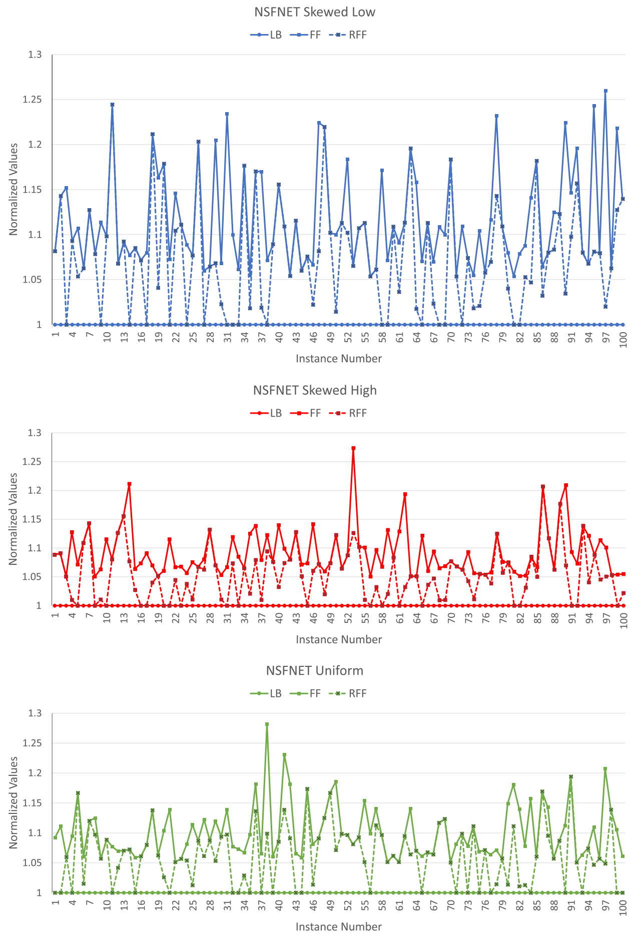

Fig. 5. Normalized solutions to 300 problem instances, NSFNET.

We create random SA problem instances characterized by two parameters: the network topology (i.e., NSFNET or GEANT2) and the traffic distribution. Each SA problem instance consists of one traffic request for each node pair in the corresponding network topology. For a given node pair, we generate a random value for its data rate based on one of three distributions:

(1) *uniform*: each of the five data rates, 10, 40, 100, 400, or 1000 Gbps, is selected with equal probability;
(2) *skewed low*: the rates *above* are selected with probabilities 0.30, 0.25, 0.20, 0.15, and 0.10, respectively; or
(3) *skewed high*: the five rates are selected with probabilities 0.10, 0.15, 0.20, 0.25, and 0.30, respectively.

Once the data rates between all node pair have been generated, we assign spectrum slots to the traffic requests as discussed above.

We run our experiments on Henry2, a Linux HPC cluster operated by NC State University that offers of the order of 1000 compute nodes with well over 10,000 cores [23]. In our experiments, we let the RFF algorithm run until it reaches either the lower bound (in which case, we know for certain it has found an optimal solution) or a 5 h limit on running time; while in the latter case, we are not certain that the algorithm has found an optimal solution, the results we present next indicate that the solution is very close to optimal.

B. RFF Solution Quality

The performance measure we consider is the maximum number of spectrum slots on any network link as obtained by either the FF or RFF algorithm. For meaningful comparisons between problem instances, we normalize the solutions returned by FF or RFF by dividing with the lower bound *LB* for the corresponding instance from expression (1). Clearly, the closer the normalized value is to 1.0, the better the solution.

Figures 5 and 6 present results for the NSFNET and GEANT2 topologies, respectively. Each figure includes three subfigures, one each for demand matrices generated by the skewed low, skewed high, and uniform distributions. Each subfigure plots the normalized FF solution, normalized RFF solution, and normalized lower bound (the last one as a horizontal line at $y = 1.0$), for each of 100 random problem instances generated for the stated parameters (i.e., network topology and traffic demand distribution).

We first note that the FF algorithm produces solutions of good quality that are within 30% (respectively, 12%) of the lower bound for the 300 NSFNET (respectively, GEANT2) problem instances. These results are consistent with earlier research indicating that the FF algorithm performs well. Regarding the RFF algorithm, we observe that it finds better solutions than FF in most instances. Table 1 summarizes the average relative performance of the FF and RFF algorithms in

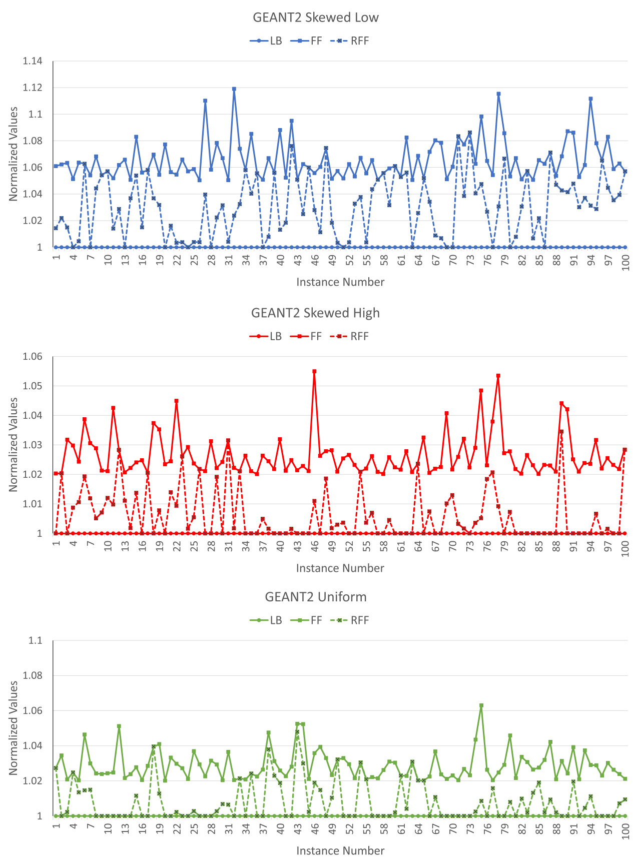

Fig. 6. Normalized solutions to 300 problem instances, GEANT2.

terms of how far (in percentage) their solutions are from the lower bound, the number of instances (out of 100 for each distribution) that the RFF produces better solutions than FF, the number of instances that RFF finds a solution equal to the lower bound (i.e., a guaranteed optimal solution), and the average absolute difference between the FF and RFF solutions, in spectrum slots. For the NSFNET (respectively, GEANT2) network, RFF improves on the FF solution in 47–53 (respectively, 71–79) instances, depending on the traffic distribution, of which it finds a solution equal to the lower bound in 20–26 (respectively, 14–33) instances. Also, although the percentage improvement over the FF solution is lower for the GEANT2 network, the absolute difference is more than twice that for the NSFNET network. In other words, even a small improvement in the larger GEANT2 network results in significantly larger spectrum savings, especially since it applies across many more network links.

C. Parallel Exploration of the RFF Solution Space

Let us now explore how parallelism helps speed up the exploration of the solution space. Recall that the RFF solution space consists of the $O(K!)$ permutations represented by the leaves of the RFF tree in Fig. 2. We consider two metrics that provide insight into the number of permutations that the algorithm is able to explore within a certain amount of time.

(1) **Number of leaf nodes of the RFF tree the algorithm visits.** This metric represents the number of times line 2 of the RFF Algorithm 2 is executed, i.e., the number of times the algorithm evaluates a complete permutation of the K requests.
(2) **Number of branches of the RFF tree that the algorithm trims.** This metric represents the number of times the algorithm *skips* the recursive call in line 18 of the RFF algorithm after making a determination in line 14 that continuing with the current partial permutation will not lead to a better solution.

Note that each leaf node visited represents a complete permutation that the algorithm evaluates *directly*. On the other hand, each branch trimmed corresponds to multiple complete permutations with the same prefix (i.e., the same leading partial permutation) that the algorithm evaluates *indirectly* and does not consider, after determining in line 18 that they cannot lead to a better solution. The number of such complete permutations for each branch depends on the level of the tree where the recursive call backtracks.

Tables 2 and 3 present these two metrics as a function of both time and number of threads employed for one problem instance for the NSFNET and GEANT2 topologies, respectively. For the results shown in these tables, we started a number m of threads ($m = 4, 8, 12, 16, 20, 24, 32$) at time $t = 0$ and let all threads run until either the lower bound was reached or 5 h elapsed, whichever occurred first. Each of the threads explores a different subtree of the root (refer to Fig. 2) and hence a different part of the permutation space; we assigned the m threads to the leftmost m subtrees of the root. Results very similar to the ones shown in the two tables have been obtained for all problem instances we explored.

Let us first consider Table 2, which shows results for an NSFNET problem instance with a lower bound equal to 49 and an initial solution obtained by the FF heuristic equal to

Table 1. Relative Performance of FF and RFF Algorithms

		FF	RFF			
	Traffic	**% from LB**	**% from LB**	**# Instances < FF**	**# Instances = LB**	**Avg. Diff. (Slots)**
NSFNET	Skewed high	9.28%	5.46%	53	20	3.78
	Skewed low	11.73%	6.55%	52	26	2.65
	Uniform	10.12%	6.01%	47	23	3.08
GEANT2	Skewed high	2.66%	1.22%	79	14	8.44
	Skewed low	6.58%	3.54%	77	30	7.76
	Uniform	2.88%	1.37%	71	33	6.47

Table 2. Exploration of the Permutation Space, NSFNET[a]

# Threads	Solution < FF	Metric	1 h	2 h	3 h	4 h	5 h
4	54, thread #4, at 4 s	# leaves visited	34,296	66,272	93,306	131,155	163,585
		# branches trimmed	216,663	418,252	588,868	827,575	1,032,258
8	54, thread #7 at 4.3 s	# leaves visited	69,273	134,123	188,173	263,795	328,588
		# branches trimmed	434,245	838,736	1,176,169	1,648,114	2,052,756
12	54, thread #4 at 4.6 s	# leaves visited	127,182	246,496	345,935	485,053	604,269
		# branches trimmed	628,374	1,214,951	1,703,769	2,388,076	2,974,743
16	54, thread #15 at 4.2 s	# leaves visited	209,913	406843	570967	800663	997538
		# branches trimmed	929,582	1,798,143	2,521,971	3,535,185	4,403,676
20	56, 54, thread #1, #18 at 4, 4.1 s, respectively	# leaves visited	275,740	5,59,773	843,719	1,127,720	1,411,716
		# branches trimmed	1,136,325	2,302,512	3,468,185	4,634,389	5,800,467
24	54, thread #1 at 4.4 s	# leaves visited	339,936	688,602	1,037,170	1,385,642	1,734,152
		# branches trimmed	1,334,599	2,698,611	4,062,554	5,426,799	6,790,888
32	54, thread #20 at 4.3 s	# leaves visited	469,795	930,734	1,391,734	1,852,580	2,313,480
		# branches trimmed	2,000,565	3,956,799	5,913,386	7,868,762	9,824,610

[a]Problem instance with LB = 49 and FF solution = 58.

Table 3. Exploration of the Permutation Space, GEANT2[a]

# Threads	Solution < FF	Metric	1 h	2 h	3 h	4 h	5 h
4	N/A	# leaves visited	29,876	60,526	91,167	121,817	152,466
		# branches trimmed	51,533	104,193	156,846	209,507	262,157
8	N/A	# leaves visited	33,007	68,333	103,657	138,984	174,322
		# branches trimmed	241,861	499,835	757,688	1,015,629	1,273,617
12	N/A	# leaves visited	32,797	68,037	103,282	138,510	173,755
		# branches trimmed	348,867	722,884	1,096,902	1,470,812	1,844,861
16	N/A	# leaves visited	27,625	58,646	89,632	120,635	151,622
		# branches trimmed	426,081	902,905	1,379,765	1,856,722	2,33,3593
20	227, thread #18 at 33 s	# leaves visited	7831	17,797	27,765	37,733	47,679
		# branches trimmed	836,549	1,780,946	2,725,429	3,669,986	4,612,521
24	227, thread #18 at 42 s	# leaves visited	7147	15,622	24,078	32,541	41,009
		# branches trimmed	927,868	1,894,965	2,862,343	3,829,659	4,797,085
32	226, thread #26 at 41 s	# leaves visited	8241	17,536	26,835	36,136	45,436
		# branches trimmed	1,298,205	2,703,580	4,108,992	5,514,505	6,919,965

[a]Problem instance with LB = 220 and FF solution = 232.

58. As indicated in the second column of the table, RFF was able to find a better solution equal to 54 within the first 5 s; for the remaining almost 5 h, the search of the permutation space did not yield a better solution. Note that solutions with the same objective value are found by different threads, depending on which thread was able to reach a solution first; since each thread explores a different subtree, solutions obtained by different threads correspond to different permutations even though they have the same objective value.

From Table 2, we first observe that, as expected, for a specific number m of threads employed, both the number of leaf nodes visited and the number of branches trimmed increase almost linearly with time. Now consider how these numbers change for a given amount of running time as the number of threads increases. As we can see, both the number of leaves visited and the number of branches trimmed increases almost linearly as the number of threads increases from 4 to 8 and then to 12. However, the increase is faster than linear as we employ 16 or more threads. As an example, consider the full 5 h running time. With only 4 threads, RFF visits ~160K leaves and trims ~1M branches; however, with 32 threads, the algorithm visits ~2.3M leaves and trims ~9.8M branches. In other words, an $8\times$ increase in number of threads results in a $14\times$ increase in leaves visited and a $9.5\times$ increase in branches trimmed. These trends can be explained as follows. The RFF algorithm visits a leaf F when the permutation it represents has a solution close to the best solution available, and hence, the branch where F resides cannot be trimmed earlier. Note that other leaf nodes in the vicinity of F correspond to permutations with the same (long) prefix as the permutation of F. If these permutations also represent good solutions, the algorithm will visit them quickly, with just a small amount of backtracking (i.e., without having to return all the way to the original call and start the exploration from the top of the subtree). We also note from the second column of Table 2 that multiple solutions with the same value exist in different parts of the tree. With multiple threads running in parallel, the RFF algorithm is able to find clusters of good solutions with the same long prefix in various parts of the tree and visit them, resulting in the faster than linear exploration of leaf nodes. At the same time, with multiple threads, the RFF algorithm will also explore in parallel parts of the permutation space that contain solutions far from optimality. Because of good solutions found in other parts of the RFF tree, the algorithm is able to trim branches that do not lead to promising solutions early on, also resulting in a faster than linear increase in the number of branches trimmed; we will elaborate on this phenomenon shortly.

Let us now turn our attention to Table 3, which shows the results of similar experiments on a GEANT2 problem instance with a lower bound equal to 220 and an initial FF solution of 232. We first observe that, for the same number of threads and amount of time, the RFF algorithm visits fewer leaf nodes, and trims fewer branches, in the GEANT2 case than for NSFNET. This result can be explained by noting that since GEANT2 is a larger network than NSFNET, the number of requests and their path lengths are larger for GEANT2 than for NSFNET. Hence, each recursive call for the GEANT2 network takes longer to execute than a call for the NSFNET because (1) the **for** loop in line 10 of the RFF Algorithm 2 has a larger number of requests to consider, and (2) line 14 that updates the solution by applying FF has to check a larger number of links, on average, along the path of each request.

We again observe that, for a given number of threads, the number of leaf nodes visited and the number of branches trimmed increase almost linearly with time. But this instance helps demonstrate an important advantage of using multiple threads to explore the permutation space. Specifically, when using up to 16 threads, the RFF algorithm cannot find any solution that is better than the initial FF solution; apparently, there are no solutions in the parts of the 16 leftmost subtrees of the RFF tree that the threads are able to explore within 5 h. However, with 20 threads running in parallel, thread #18 finds a better solution within the first minute. Since this solution (227) is much closer to the lower bound (220), the other threads, including the 16 exploring the leftmost subtrees, may now eliminate permutations much earlier without the need to traverse a path down to a complete permutation at a leaf node. As a result, we observe that, for a given amount of computation time, the number of leaf nodes visited drops sharply as we go from 16 to 20 threads, whereas the number of branches trimmed almost doubles. A similar phenomenon is observed as we increase the number of threads from 24 to 32, and thread #26 finds an even better solution (226). This solution allows other threads to trim more branches even earlier, and hence, there is another significant increase in the number of branches trimmed within a specific amount of time.

As a general observation, when multiple threads explore different parts of the optimization space concurrently, the RFF algorithm is able to identify better solutions faster; in turn, a better solution identified by *any* thread allows *all* threads to terminate a recursive call earlier (i.e., higher in their part of the tree) and hence trim more branches. In the GEANT2 problem instance shown in Table 3, within the 5 h time limit, 4 threads trim ∼262K branches, whereas 32 threads trim ∼6.9M branches, a 26-fold increase. Moreover, each branch trimmed includes multiple permutations that are effectively "explored" and determined to contain solutions worse than the current best one. Consequently, parallelism allows for a faster than linear increase in the amount of optimization space that is explored (as a function of the number of threads employed) within a given computational budget.

7. CONCLUDING REMARKS

We have developed RFF, an algorithm that applies the FF heuristic recursively to solve optimally the SA problem. The algorithm is highly parallel and produces solutions that are close to the lower bound, generally within just 1 min for the NSFNET and GEANT2 topologies we considered in this work. Our group is currently working to extend this work in three directions: (1) develop effective methods to explore the solution space in parallel, including randomization techniques, (2) adapt RFF to problems that impose additional constraints on SA, and (3) integrate RFF with the algorithm in [17] so as to solve large RSA problems efficiently.

Funding. National Science Foundation; Directorate for Computer and Information Science and Engineering (CNS-1907142).

REFERENCES

1. J. M. Simmons, *Optical Network Design and Planning* (Springer, 2008).
2. J. Simmons and G. N. Rouskas, "Routing and wavelength (spectrum) allocation," in *Springer Handbook of Optical Networks*, B. Mukherjee, I. Tomkos, M. Tornatore, P. Winzer, and Y. Zhao, eds. (Springer, 2020).
3. G. N. Rouskas, "Routing and wavelength assignment in optical WDM networks," in *Wiley Encyclopedia of Telecommunications*, J. Proakis, ed. (Wiley, 2001).
4. R. Dutta and G. N. Rouskas, "A survey of virtual topology design algorithms for wavelength routed optical networks," Opt. Netw. **1**, 73–89 (2000).
5. B. Jaumard, C. Meyer, and B. Thiongane, "Comparison of ILP formulations for the RWA problem," Opt. Switch. Netw. **4**, 157–172 (2007).
6. M. Klinkowski, P. Lechowicz, and K. Walkowiak, "Survey of resource allocation schemes and algorithms in spectrally-spatially flexible optical networking," Opt. Switch. Netw. **27**, 58–78 (2018).
7. S. Talebi, F. Alam, I. Katib, M. Khamis, R. Khalifah, and G. N. Rouskas, "Spectrum management techniques for elastic optical networks: a survey," Opt. Switch. Netw. **13**, 34–48 (2014).
8. R. Dutta and G. N. Rouskas, "Traffic grooming in WDM networks: past and future," IEEE Netw. **16**, 46–56 (2002).
9. B. Chen, G. N. Rouskas, and R. Dutta, "Clustering methods for hierarchical traffic grooming in large-scale mesh WDM networks," J. Opt. Commun. Netw. **2**, 502–514 (2010).
10. D. Zhou and S. Subramaniam, "Survivability in optical networks," IEEE Netw. **14**, 16–23 (2000).
11. S. Talebi, E. Bampis, G. Lucarelli, I. Katib, and G. N. Rouskas, "Spectrum assignment in optical networks: a multiprocessor scheduling perspective," J. Opt. Commun. Netw. **6**, 754–763 (2014).
12. H. Zang, J. P. Jue, and B. Mukherjee, "A review of routing and wavelength assignment approaches for wavelength-routed optical WDM networks," Opt. Netw. **1**, 47–60 (2000).
13. Y. Zhu, G. N. Rouskas, and H. G. Perros, "A comparison of allocation policies in wavelength routing networks," Photon. Netw. Commun. **2**, 267–293 (2000).
14. R. Ramaswami and K. Sivarajan, "Routing and wavelength assignment in all-optical networks," IEEE/ACM Trans. Netw. **3**, 489–500 (1995).
15. B. Jaumard, C. Meyer, and B. Thiongane, "ILP formulations for the routing and wavelength assignment problem: symmetric systems," in *Handbook of Optimization in Telecommunications* (Springer, 2006), pp. 637–677.
16. E. Yetginer, Z. Liu, and G. N. Rouskas, "Fast exact ILP decompositions for ring RWA," J. Opt. Commun. Netw. **3**, 577–586 (2011).
17. M. Fayez, I. Katib, G. N. Rouskas, T. F. Gharib, and H. Faheem, "A scalable solution to network design problems: decomposition with exhaustive routing search," in *IEEE GLOBECOM*, December 2020.

18. S. Huang, R. Dutta, and G. N. Rouskas, “Traffic grooming in path, star, and tree networks: complexity, bounds, and algorithms,” IEEE J. Sel. Areas Commun. **24**, 66–82 (2006).
19. M. T. Goodrich and R. Tamassia, *Data Structures & Algorithms in Java* (Wiley, 2010).
20. F. Tang, G. Shen, and G. N. Rouskas, “Crosstalk-aware shared backup path protection in multi-core fiber elastic optical networks,” J. Lightwave Technol. **39**, 3025–3036 (2021).
21. N. Skorin-Kapov, J. Chen, and L. Wosinska, “A new approach to optical networks security: attack-aware routing and wavelength assignment,” IEEE/ACM Trans. Netw. **18**, 750–760 (2010).
22. M. Jinno, B. Kozicki, H. Takara, A. Watanabe, Y. Sone, T. Tanaka, and A. Hirano, “Distance-adaptive spectrum resource allocation in spectrum-sliced elastic optical path network,” IEEE Commun. Mag. **48**(8), 138–145 (2010).
23. “NC State Henry2 Linux cluster,” https://projects.ncsu.edu/hpc/About/ComputeResources.php.