Generic Dijkstra for Optical Networks

Ireneusz Szcześniak, Andrzej Jajszczyk, Bożena Woźna-Szcześniak

Abstract—We present the *generic* Dijkstra shortest path algorithm: an efficient algorithm for finding a shortest path in an optical network, both in a wavelength-division multiplexed network, and an elastic optical network (EON). The proposed algorithm is an enabler of real-time softwarized control of large-scale networks, and is not limited to optical networks. The Dijkstra algorithm is a generalization of the breadth-first search, and we generalize the Dijkstra algorithm further to resolve the continuity and contiguity constraints of the frequency slot units required in EONs. Specifically, we generalize the notion of a label, change what we iterate with, and reformulate the edge relaxation so that vertices are revisited, loops avoided, and worse labels discarded. We also used the typical constriction during edge relaxation to take care of the signal modulation constraints. The algorithm can be used with various spectrum allocation policies. We motivate and discuss the algorithm design, and provide our free, reliable, and generic implementation using the Boost Graph Library. We carried out 85000 simulation runs for realistic and random networks (Gabriel graphs) of 75 vertices with about a billion shortest-path searches, and found that the proposed algorithm outperforms considerably three other competing optimal algorithms that are frequently used.

Index Terms—Dijkstra algorithm, shortest path routing, elastic optical network, wavelength-division multiplexing

I. Introduction

Routing of a single connection in an optical network is one of the most important tasks of operating an optical network, and one of many research problems of the optical network design, planning, and operation. In the wavelength-division multiplexed (WDM) network, the problem is called the routing and wavelength assignment (RWA) problem, and in the elastic optical network it is called the routing and spectrum assignment (RSA) problem, or the routing, modulation, and spectrum assignment (RMSA), if we take into account the constraints of the signal modulation. Related problems exist in other optical networks: in the optical transport network (OTN) for the virtual and contiguous concatenation, or in the space division multiplexing (SDM) network for the fiber-core assignment.

In EONs, the optical spectrum (the erbium window) is divided into fine *frequency slot units* (of, e.g., 6.25 GHz width), or just *units*, as opposed to coarse fixed-grid channels (of, e.g., 25 GHz width) of WDM networks. In EONs, *contiguous*

units are concatenated to form a *slot*. Slots are tailored for a specific demand, unlike WDM channels, thus making EONs more spectrum efficient than WDM networks.

Future optical networks should deal with dynamic traffic, where connections are frequently established, and are of short duration as opposed to the quasi-static WDM connections that are characteristic of traditional networks. Furthermore, given the increasing deployment of optical networks, the network densification, the softwarization of the network control, the ever-increasing need for bandwidth and agility, further increased by the content-oriented services, network and service orchestration, and the next generation wireless network requirements, a shortest optical path should be found *fast*. The proposed algorithm enables real-time control of future optical networks.

The RWA, RSA and RMSA problems come in many versions, most notably static (a.k.a. offline) and dynamic (a.k.a. online). The objective of the static version is to route a number of demands along shortest paths in an unloaded network using the least spectrum. The objective of the dynamic version is to route a single demand along a shortest path in a loaded network using the available spectrum. The static version is nondeterministic polynomial time complete (NP-complete), but the dynamic version is not, because it can be solved tractably (though inefficiently) by finding a shortest path in a number of filtered graphs.

Our novel contribution is the algorithm which efficiently solves the dynamic RWA, RSA and RMSA problems. The algorithm is the generalization of the Dijkstra shortest-path algorithm. With simulations, we demonstrate its efficiency in comparison to three other optimal algorithms frequently used in research. The implementation of the algorithm using the Boost Graph Library is available at [2]. We published the first implementation of the algorithm in 2013 [3].

The shortest path Dijkstra algorithm is a premier graph algorithm, amenable to various adaptations due to its simple and clever design. Dijkstra is optimal (i.e., it finds a shortest path) and efficient (no better algorithm has been proposed yet after half a century since it was proposed), and follows the *label-setting* paradigm, as opposed to the label-correcting paradigm [4]. At first look, our generalization seems to discard the label-setting paradigm in favor of the label-correcting paradigm, because we allow for revisiting vertices, which Dijkstra does not do, and which is a hallmark of the label-correcting algorithms. But this is not so; the proposed algorithm is still a label-setting algorithm, only with a generalized notion of a label and reformulated edge relaxation.

The paper is organized as follows. In Section II we review related works, in Section III we define the research problem, in Section IV we describe the proposed algorithm, and in Section V we report on the simulation results. Finally, Section

VI concludes the paper.

II. Related works

We extended our conference paper [1] in a number of significant ways. First, we refined the algorithm description by defining better the relation between labels. Second, we took into account the signal modulation constraints. Third, since the proposed algorithm is not heuristic, we evaluated its performance in comparison with three other optimal algorithms, and not with heuristic algorithms as before.

The proposed algorithm was inspired by the *generic programming* paradigm which is based on mathematical abstraction: generic data structures and algorithms can operate on any algebraic structure, provided it has the required properties, such as operations or ordering relations [5]. We call the proposed algorithm the *generic Dijkstra* algorithm as a tribute to generic programming.

The dynamic RWA, RSA, and RMSA problems are defined in the literature in two different ways. First, they can be defined to optimize the overall network performance expressed in, e.g., the bandwidth blocking probability for a set of demands, bearing similarity to the static version, thus this problem is considered hard [6]. Second, these problems can be defined to optimize a single connection (i.e., to find a shortest path or a path of minimal cost), since in an operational network requests usually arrive sequentially [7]. We concentrate on the second definition of the dynamic RWA, RSA, and RMSA problems, i.e., we optimize a single connection.

The static RWA was proven to be NP-complete nearly three decades ago, and then the static RSA was proven NP-complete too [8]. The status of the single-connection dynamic RWA, RSA, and RMSA problems was unclear: no proof of NP-completeness was proposed, but heuristic algorithms and linear-programming formulations were proposed and reviewed [9], [10], [11].

The dynamic RWA, RSA, and RMSA problems can be solved inefficiently by finding shortest paths in filtered graphs. A filtered graph retains only those edges which can support a given slot. For a given demand and available modulations, we compute set S of slots for which we filter the input graph, and search for a shortest path. From among the shortest paths found, the shortest one is selected. This algorithm, termed *filtered-graphs algorithm*, is of $O(|S| \times |V| \log |V|)$ complexity, where V is the set of vertices of a (sparse, we assume) graph. This computational complexity is the upper bound, and a proof that these problems are tractable.

We argue that the filtered-graphs algorithm is inefficient and we show through simulations on realistic networks that the proposed algorithm is considerably faster than the filtered-graphs algorithm. To the best of our knowledge, we are the first to propose an optimal and efficient algorithm for the dynamic routing problem, as simulated in realistic optical networks.

In [12] the authors report that by computing in advance the set of slots that can be assigned to a demand, the complexity added by the contiguity constraint is removed. We, in contrast, show that the time performance of the filtered-graphs algorithm is worse than the time performance of the proposed

algorithm, because we process not a slot, but contiguous units, which can include many slots.

In [13] the authors proposed an algorithm for solving the dynamic RSA problem with the brute-force search strategy of enumerating the paths capable of supporting a demand. The algorithm does not use the edge relaxation to limit the search space. The algorithm stores the complete paths in a priority queue, and does not use the dynamic-programming principle of reusing intermediate results, like Dijkstra does with node labels. For these reasons, we refer to this algorithm as the *brute-force algorithm*.

In [14] the authors proposed an algorithm, which checks whether the consecutive paths provided by the Yen K-shortest path (KSP) algorithm can support a demand. When K is limited to some value (e.g., $K = 10$), we call this algorithm the *limited Yen KSP*, and if there is no limit on K, we call the algorithm the *unlimited Yen KSP*. The limited Yen KSP algorithm is heuristic, while the unlimited Yen KSP is optimal. If there is a path capable of supporting the demand, the unlimited Yen KSP algorithm will eventually find this path, but its K could be very large, possibly a million, even for small graphs.

In [15], the authors proposed a dynamic RWA algorithm, which is based on the algorithm in [16], and in turn based on the Dijkstra-based algorithm proposed in [17]. In [17], the authors introduced the transitive path domination relation to solve the max-min problem, where the network resources are continuous. This relation was adapted in [16] for discrete network resources and then used [15] to model the availability of wavelengths. That adaptation, however, leads to a less efficient search, since it was defined with the $\geq$ relation for the wavelength availability vector, which models the $\supseteq$ set relation. We, in contrast, define the relation of the incomparability of solution labels using the $\supsetneq$ relation for a set of contiguous units, and show that this relation is intransitive.

The authors in [15], and [16] report on the complexity of their algorithms, which may be exponential. This complexity is the result of the way the path cost vectors are defined, compared, and produced when an edge is traversed. In [15] the path cost vector stores the availability information of all wavelengths of a path. The edge traversal produces paths which are non-dominant, but which can have overlapping wavelengths, thus possibly leading to the explosion of the search space (the set of non-dominant paths). We, in contrast, store in a solution label a single sequence of contiguous units (represented by two integer numbers), which is faster to process. During the edge relaxation, we produce a set of incomparable labels (without overlapping units), which limits the search space (the set of incomparable labels).

The algorithm in [15] finds an optimal solution in two phases: in the first phase, the complete set of non-dominated solutions is obtained, and in the second phase an optimal solution is selected. In contrast, our algorithm can make optimization decisions for intermediate solutions by sorting the priority queue elements, taking into account, e.g., the spectrum allocation policy. Thus, our algorithm has a single phase, and produces a single result, which is more efficient.

In [14], the authors also proposed a heuristic algorithm,

termed a modified Dijkstra algorithm, which is the Dijkstra algorithm with the constrained edge relaxation, where a candidate path is rejected if it cannot support a demand.

In [18], the authors proposed a heuristic algorithm, which is a constrained Yen K-shortest path algorithm that drops the path deviations incapable of supporting a demand. The Yen algorithm delegates the shortest path search to the Dijkstra algorithm.

The limited Yen, the modified Dijkstra, and the constrained Yen algorithms, which find a candidate path with the Dijkstra algorithm, fail to find a shortest path meeting the spectrum and modulation constraints, when there is a shorter path unable to meet the spectrum and modulation constraints, because that shorter path diverts the algorithms into a dead end.

III. PROBLEM STATEMENT

Given:

- directed multigraph $G = (V, E)$, where $V = \{v_i\}$ is a set of vertices, and $E = \{e_i\}$ is a set of edges,
- cost function $\mathrm{cost}(e_i)$, which gives non-negative cost (length) of edge e_i,
- available units function $\mathrm{AU}(e_i)$, which gives the set of available units of edge e_i, which do not have to be contiguous,
- s and t are the source and target vertices of the demand,
- a *decision function of monotonically increasing requirements*, which returns true if a candidate solution (the given contiguous units at the given cost) can support the demand, otherwise false,
- the set of all units Ω on every edge.

Find:

- a shortest path (a sequence of edges),
- continuous and contiguous units.

We refer to a set of contiguous units as a CU. We denote a CU with the units starting at a and ending at b inclusive as $[a \ldots b]$. For instance, $[0 \ldots 2]$ denotes units 0, 1 and 2. A set of units can be treated as a set of CUs. For instance, $\{0, 1, 3, 4, 5\}$ and $\{[0 \ldots 1], [3 \ldots 5]\}$ are the same.

Two CUs are *incomparable*, when one is not included in the other. For instance, $[0 \ldots 2]$ is incomparable with $[2 \ldots 3]$. We denote the incomparability of CUs with the $\parallel$ relation. For instance, $[0 \ldots 2] \parallel [2 \ldots 3]$.

We intentionally stated the problem generically by introducing the decision function to consider the RWA, RSA, and RMSA problems at once. The decision function is responsible for accepting or rejecting a candidate solution, and gives a user leeway to define what an acceptable candidate solution is. For RWA, the function should check whether a CU has at least one unit (wavelength), for RSA, whether a CU has at least the number of contiguous units required by the demand, and for RMSA, whether a CU has at least the number of contiguous units required by the demand at a given cost (distance). The decision function could also check other parameters such as the signal quality.

The requirements of the decision function should be monotonically increasing as the cost increases, i.e., a candidate solution rejected at a given cost could not be accepted at a
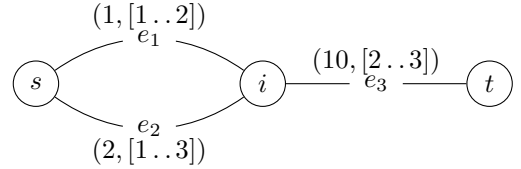

Fig. 1: Example for vertex revisiting and looping.

higher cost. It is a valid assumption, since as the distance grows, the number of required contiguous units can only grow. This assumption allows us to reject a candidate solution due to its insufficient number of the available contiguous units, because as distance grows, that rejected solution would fail to provide the same or greater number of required contiguous units anyway.

The demand bitrate is not a given of the stated problem, because that would narrow the problem statement. If need be, the demand bitrate should be considered by the decision function. In Section V we use the proposed algorithm to solve the RMSA problem, and there, in Algorithm 3, we define a decision function which takes into account the demand bitrate.

IV. PROPOSED ALGORITHM

We generalize and constrain the shortest path Dijkstra algorithm to find a shortest path in EONs for a given demand. The generalization is novel, and the constriction is trivial. The generalization resolves the unit continuity and contiguity constraints, while the constriction takes into account the signal modulation constraints.

In label-setting algorithms, a label is associated with a vertex, and gives information on what cost and how to reach that vertex from the source. In Dijkstra, the label is defined as a pair of a cost and a preceding vertex. Each vertex has at most one label, which we call the *vertex label*. For a given vertex, the vertex label is initially *tentative*, because it can be updated by the edge relaxation, and then becomes *permanent* when the vertex is visited.

The Dijkstra algorithm is label setting in that once a vertex is visited, its label is set for good (the status of the vertex label changes from *tentative* to *permanent*), but before that happens, the vertex label converges to its optimum by *edge relaxation*.

In Dijkstra, when relaxing edge e, a tentative vertex label is updated with a better candidate label. The tentative vertex label, if it exists (i.e., it has been found by some earlier relaxation), is the tentative vertex label of the target vertex of edge e. The candidate label is the label produced for edge e, and tells the cost of reaching the target vertex of edge e. The candidate label is better than the tentative vertex label if it has a lower cost.

A. Observations on the stated problem

The following three observations shaped the generalization.

1) Revisit vertices: In Dijkstra, a vertex is visited only once for a single label. We, however, want to *revisit* a vertex even for a label of a higher cost than the vertex label, because it
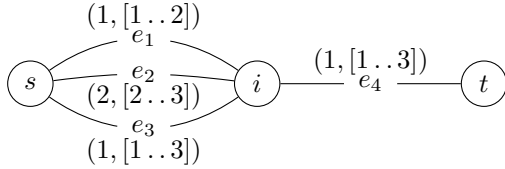
Fig. 2: Example for discarding worse labels.

may eventually yield a shortest path capable of supporting a given demand.

To demonstrate vertex revisiting, we show an example in Fig. 1, where an edge is annotated with the length and the available units, e.g., $(1, [1 . . 2])$ says the edge is of length 1 with units 1 and 2 available. We are searching for a shortest path with two units from vertex s to vertex t.

In the first iteration of Dijkstra, vertex s is visited, vertex i is discovered along edge e_1, and the discovery along edge e_2 is discarded because of a higher cost. In the second iteration, vertex i is visited, and now we know that we can get to vertex i along edge e_1 at cost 1, and with $[1 . . 2]$. The problem is that vertex t cannot be discovered, because the spectrum continuity constraint would be violated: the demand requires two units, vertex i is visited with $[1 . . 2]$, but $\mathrm{AU}(e_3) = [2 . . 3]$. We reach a dead end; the search stops with no solution.

Continuing with the example, and allowing for vertex revisiting, now vertex i is discovered along both parallel edges e_1 and e_2, and none of the discoveries is discarded. Now there are two tentative labels for vertex i. Then vertex i is visited along edge e_1 at cost 1 with $[1 . . 2]$, and then revisited along edge e_2 at cost 2 with $[1 . . 3]$, thus allowing vertex t to be discovered, end eventually visited at cost 12 with $[2 . . 3]$.

2) Avoid loops: In Dijkstra, loops are avoided, because an edge is relaxed only if it yields a candidate label of a lower cost. Since edge costs are non-negative, loops cannot decrease cost, and so they will not be allowed by edge relaxation.

The problem is we will find loops if we revisit vertices at higher costs. For instance, considering the same example in Fig. 1: when we visit vertex i, we rediscover vertex s and later revisit it, thus finding the loop with edges e_1 and e_2.

To avoid loops, we refine when we can revisit a vertex. We still allow a revisit at a higher cost, *but only for a CU not included in the CUs of previous visits.* Therefore, a vertex is visited and possibly revisited always at the lowest cost for a CU not included in the CUs of previous visits.

For example, in Fig. 1, we start the search by visiting vertex s at cost 0, and with the CU of Ω. While visiting vertex i at cost 1, and with $[1 . . 2]$, we rediscover vertex s along edge e_2, but the edge will not be relaxed, because vertex s was already visited with Ω, which includes $[1 . . 2]$. This example would hold even if the loop did not increase the cost.

3) Discard labels: In Dijkstra, when a tentative vertex label is updated by the edge relaxation, the previous value of the tentative label is discarded. In generic Dijkstra, when we relax an edge, we can discard a number of tentative labels.

For instance, in Fig. 2, while visiting vertex s, three edges are relaxed: first edge e_1 is relaxed at cost 1, and with $[1 . . 2]$, next edge e_2 is relaxed at cost 2, and with $[2 . . 3]$, and, finally,

edge e_3 is relaxed at cost 1, and with $[1 . . 3]$. The results for edges e_1, and e_2 are discarded, because the result for edge e_3 is better: at cost 1 it offers $[1 . . 3]$, which includes both $[1 . . 2]$, and $[2 . . 3]$. Thus, vertex i is visited at cost 1, and with $[1 . . 3]$.

B. Changes to the Dijkstra algorithm

Based on the observations, we motivate the changes to the Dijkstra algorithm that make it generic, and applicable to the stated problem.

1) Labels: We define a label as *a tuple of cost, a CU, and a preceding edge*, to keep track of the CU used. For instance, label $(1, [1 . . 2], e_1)$ says that a vertex is reached at cost 1 and with the CU of $[1 . . 2]$ along edge e_1. To allow for multigraphs, we keep a preceding edge, not a preceding vertex, in the label. The cost of label l_i we denote as $\mathrm{cost}(l_i)$, and the contiguous units as $\mathrm{CU}(l_i)$. This label can be a candidate label, a tentative vertex label, or a permanent vertex label.

Label l_i is better than label l_j (or label l_j is worse than label l_i), denoted by $l_i < l_j$, if either:

1) label l_i offers a CU which includes the CU of label l_j at a lower cost than the cost of label l_j, i.e., $\mathrm{cost}(l_i) < \mathrm{cost}(l_j)$ and $\mathrm{CU}(l_i) \supset \mathrm{CU}(l_j)$, or
2) label l_i offers a CU which properly includes the CU of label l_j at a cost that is lower than or equal to the cost of label l_j, i.e., $\mathrm{cost}(l_i) \leq \mathrm{cost}(l_j)$ and $\mathrm{CU}(l_i) \supsetneq \mathrm{CU}(l_j)$.

Our $<$ label relation is a *strict partial order*, since it is irreflexive and transitive [19]. Furthermore, in a strict partial order some pairs can be incomparable. We say that labels l_i and l_j are incomparable, denoted by $l_i \parallel l_j$, when neither $l_i < l_j$ nor $l_j < l_i$ holds. Indeed, our labels can be incomparable.

However, our $<$ order is not a *strict weak order*, because the incomparability of labels is not transitive. For example, while $(0, [1 . . 1]) \parallel (2, [1 . . 2])$ and $(2, [1 . . 2]) \parallel (1, [1 . . 1])$ hold, $(0, [1 . . 1]) \parallel (1, [1 . . 1])$ does not, because $(0, [1 . . 1]) < (1, [1 . . 1])$ holds.

Table I shows the label relations depending on their costs and CUs, where relation $l_i > l_j$ means $l_j < l_i$.

When we rediscover or revisit a vertex, we grow *a set of incomparable labels*, i.e., for any labels l_i and l_j that are different, $l_i \parallel l_j$ is true, or equivalently $l_i < l_j$ is false. The incomparability of labels insures that in the set we do not store a label that is worse than some other label.

2) Iteration: In an iteration, Dijkstra processes a tentative vertex v (i.e., a vertex with a tentative label) of the lowest cost, while generic Dijkstra processes a tentative label of the lowest cost, where the edge of the tentative label has the target vertex v. The labels we iterate over are provided by the edge relaxation.

An iteration corresponds to visiting (or revisiting) vertex v. In Dijkstra, only the status of the label of vertex v changes from tentative to permanent. In generic Dijkstra, we insert the tentative label into the set of permanent (optimal) incomparable labels of vertex v.

3) Relaxation: We reformulate the edge relaxation. In Dijkstra, an edge is relaxed when a candidate label is *better* than the vertex label. In generic Dijkstra, we relax an edge when

TABLE I: Relations between labels l_i and l_j depending on their costs and CUs.

	$\mathrm{CU}(l_i) \supsetneq \mathrm{CU}(l_j)$	$\mathrm{CU}(l_i) = \mathrm{CU}(l_j)$	$\mathrm{CU}(l_i) \subsetneq \mathrm{CU}(l_j)$	$\mathrm{CU}(l_i) \parallel \mathrm{CU}(l_j)$
$\mathrm{cost}(l_i) < \mathrm{cost}(l_j)$	$l_i < l_j$	$l_i < l_j$	$l_i \parallel l_j$	$l_i \parallel l_j$
$\mathrm{cost}(l_i) = \mathrm{cost}(l_j)$	$l_i < l_j$	$l_i = l_j$	$l_i > l_j$	$l_i \parallel l_j$
$\mathrm{cost}(l_i) > \mathrm{cost}(l_j)$	$l_i \parallel l_j$	$l_i > l_j$	$l_i > l_j$	$l_i \parallel l_j$

Algorithm 1 Generic Dijkstra
In: graph G, source vertex s, target vertex t
Out: a pair of a shortest path, and a CU
Here we concentrate on permanent labels l.

$Q_s = \{(0, \Omega, e_\emptyset)\}$
while Q is not empty **do**
 $l = \mathrm{pop}(Q)$
 $e = \mathrm{edge}(l)$
 $v = \mathrm{target}(e)$
 // Add l to the set of permanent labels of vertex v.
 $L_v = L_v \cup \{l\}$
 if $v == t$ **then**
  break the main loop
 for each out edge e' of v in G **do**
  $\mathrm{relax}(e', l)$
return $\mathrm{trace}(L, t)$

Algorithm 2 relax
In: edge e', label l
Here we concentrate on tentative labels l'.

$c = \mathrm{cost}(l)$
$c' = c + \mathrm{cost}(e')$
$C = \mathrm{CU}(l)$
$v' = \mathrm{target}(e')$
for each CU C' in $C \cap \mathrm{AU}(e')$ **do**
 $l' = (c', C', e')$
 if $\mathrm{decide}(l')$ **then**
  if $\nexists l_{v'} \in L_{v'} : l_{v'} < l'$ **then**
   if $\nexists q_{v'} \in Q_{v'} : q_{v'} < l'$ **then**
    // Discard tentative labels $q_{v'}$ such that $l' < q_{v'}$.
    $Q_{v'} = Q_{v'} \setminus \{q_{v'} \in Q_{v'} : l' < q_{v'}\}$
    // Add l' to the set of tentative labels of vertex v'.
    $Q_{v'} = Q_{v'} \cup \{l'\}$

---

there is *no better* vertex label than the candidate label. A small tweak.

This tweak makes no difference when the relation between labels is a strict total order, as in Dijkstra. However, for our label order, the tweak entails we relax an edge not only for better labels, but also for incomparable labels. A big difference.

In Dijkstra, a vertex has a single label, which is either tentative or permanent. In generic Dijkstra, a vertex has a set of tentative labels, and a set of permanent labels. Labels in these sets are incomparable: no label in the two sets is better than some other label, i.e., for any two different labels (tentative or permanent) l_i and l_j of the given vertex, $l_i < l_j$ is false. Our edge relaxation maintains the labels in the two sets incomparable.

As part of the relaxation we discard those tentative labels (of vertex v'), which are worse than the candidate label. The permanent labels are left alone, because they are optimal.

C. Constriction

Typically, a constriction can be introduced in the edge relaxation, where we drop a candidate label if it does not meet some conditions. For instance, to limit the length of a shortest path, we drop a candidate label if its cost is greater than a given value. We use the decision function introduced in the problem statement for the constriction.

D. Algorithm

Algorithm 1 presents the complete algorithm with the typical Dijkstra algorithm structure, where the main loop processes the labels of the priority queue Q sorted in the ascending order of the label cost.

The priority queue Q is a set of Q_v, i.e., $Q = \{Q_v\}$, where Q_v is the set of tentative incomparable labels of vertex v. The solution of the search is maintained in $L = \{L_v\}$, where L_v is the set of permanent (optimal) incomparable labels of vertex v.

To boot the search, we initialize $Q_s = \{(0, \Omega, e_\emptyset)\}$ to make all units available at vertex s at cost 0. The null edge $e_\emptyset$, which is not present in graph G, marks the beginning of a shortest path.

In every iteration of the main loop, we process label l of the lowest cost popped from queue Q, and along edge e we visit vertex v. Function $\mathrm{target}(e)$ gives the target vertex of edge e with the special case of $\mathrm{target}(e_\emptyset) == s$, i.e., the source vertex. If $v == t$, then we found a solution and break the main loop. Otherwise, we try to relax each edge e' leaving vertex v.

Algorithm 2 shows the relaxation of edge e' reached with label l. We relax the edge for a set of incomparable candidate labels l', which we produce for each CU C' in the set obtained by intersecting the CU of label l and the available units of edge e'. The candidate labels l' have the same cost c' and edge e', and differ in the CU C' only. We examine label l' if the decision function $\mathrm{decide}(l')$ permits.

Next, if there is no permanent or tentative label of vertex v' better than l', we relax the edge by first discarding any tentative label of vertex v' which is worse than l', and then adding l' to $Q_{v'}$. Edge relaxation replenishes the queue with tentative labels, and the algorithm keeps iterating until destination vertex t is reached, or the queue is empty.

The required spectrum allocation policy (e.g., first-fit, best-fit) can be taken into account by the priority queue while popping a label. From among the labels of the lowest (equal)

cost, the priority queue should pop the label with the CU preferred by the given spectrum allocation policy. For instance, for the best-fit spectrum allocation policy, the queue should pop the label not only of the lowest cost, but also of the CU with the lowest number of units.

Finally, once we leave the main loop, function $\mathrm{trace}(L, t)$ traces back from node t a shortest path found, if any, based on the vertex labels L, and returns a pair of a path and a CU. The function selects, according to the spectrum allocation policy used, the CU with the required number of units from the CU of the vertex label of node t, which may have more units than required. We do not present the function, since it is the typical Dijkstra path back-tracing.

To be general, we stated and solved the problem for a directed multigraph, but the algorithm can be used for undirected multigraphs too. In our simulations we modeled an EON with an undirected graph, and were able to use our algorithm.

V. SIMULATIONS

We compared the memory and time performance of the proposed algorithm applied to the dynamic RMSA problem with three other optimal algorithms: the filtered-graphs algorithm, the brute-force algorithm, and the unlimited Yen KSP algorithm.

Early in our simulation studies, we realized that the unlimited Yen KSP is very time inefficient, which prohibited using it in our large-scale simulations. In one case, Yen produced over three hundred thousand shortest paths in 24 hours which had not met the spectrum continuity and contiguity constraints. For this reason, we were unable to include the unlimited Yen KSP algorithm in the comparison.

We report the simulation results only for the *first-fit* spectrum allocation policy, since the *best-fit* and *random-fit* policies performed worse for all algorithms compared.

To make the comparison unbiased, we implemented all algorithms with a great attention to detail, and an emphasis on time and memory performance. We especially carefully treated the filtered-graphs algorithm, and implemented it using our generic Dijkstra implementation, which is ultra efficient, employing the latest C++17 functionality, such as the extraction of the associative-array elements with the move semantics.

We made sure that the proposed algorithm and implementation were correct with unit tests, assertions, and extra code that validated the optimality and integrity of the results found. We validated, with the filtered-graphs algorithm, not only the final results found, but also the intermediate results. In the production runs, we disabled the assertions and the extra code, so that the time measurements were not disturbed.

In the simulations, we concentrated on the long-haul networks with 75 nodes, since they model the current and future optical transport networks well. However, we also carried out simulations for legacy long-haul networks with 25 nodes, and found that the brute-force algorithm, and the proposed algorithm performed comparably in terms of memory usage (using tens of kB), and execution time (taking milliseconds), while the filtered-graph algorithm was about a hundred times slower, but used only hundreds of bytes.

TABLE II: Statistics of the generated Gabriel networks.

Measured quantity	Min	Average	Max	Variance
number of edges	119	131.53	145	33.84
edge length	1	97.12	499	2712.19
vertex degree	1	3.51	8	1.21
shortest-path hops	1	5.96	20	8.62
shortest-path length	1	510.45	1369	59518.11

We also carried out fragmentary simulations for large, ultra-dense random networks with 100 nodes and 1000 edges. We found that the brute-force algorithm needed very large amounts of memory (more than 96 GB), which we did not have. The proposed algorithm on average was using a few hundred kB, and taking milliseconds. The filtered-graph algorithm was on average a thousand times slower, and used only a few kB.

A. Simulation setting

The simulation setting had three major parts: the network model, the traffic model, and the signal modulation model.

1) Network model: We generated a set of random long-haul graphs with random traffic to obtain reliable statistical results for various populations of interest. Specifically, we used Gabriel graphs, because they have been shown to model the properties of the long-haul transport networks very well [20].

A network model is defined by a network graph, and the number $|\Omega|$ of edge units. We randomly generated 100 Gabriel graphs. Each graph had 75 vertices, which were uniformly distributed over a square area with the typical density of one vertex per 10^4 km^2. In generating Gabriel graphs, the number of edges cannot be directly controlled, as it depends on the location of vertices, and on the candidate edges meeting the conditions of the Gabriel graph. The statistics of the generated graphs are given in Table II.

For $|\Omega|$, we used the three values of 160, 320, and 640. For the conventional band (C-band), 160 units would require the spacing of 25 GHz, 320 units the spacing of 12.5 GHz, and 640 units the spacing of 6.25 GHz.

2) Traffic model: Demands arrive according to the exponential distribution with the rate of λ demands per day. The probability distribution of the demand holding time is also exponential with the mean of δ days. The end nodes of a demand are different and chosen at random. The number of units a demand requests follows the distribution of $(\mathrm{Poisson}(\gamma - 1) + 1)$ with the mean of γ, i.e., the Poisson distribution shifted by one to the right, to ensure that the number of units is greater than zero.

We describe a demand with a requested number of units, and not with a bitrate, to keep the discussion simple, and because the algorithms operate on units, and not on bitrate. If needed, a demand can be described with bitrate, and the required number of units can be obtained using function $n_1(b)$, which should take into account the technical details of the specific modulation and optical hardware used.

To investigate the difference in algorithm performance as γ increases, we used two values of 1 and 10 for γ. Using $\gamma = 1$ approximates the algorithm performance for a traditional WDM network.

We argue that the choice of a traffic model is irrelevant to our study as the traffic only produces the input data (i.e., the state of the graph) for the routing algorithms, and we chose the exponential and Poisson distributions to keep the discussion simple. The question is how the algorithms perform under the given utilization, regardless of how the utilization was obtained, which could have been equally well produced randomly.

We express the mean demand arrival rate λ as a function of the offered load μ as estimated by (1), where α is the mean number of edges of all shortest-paths in a network being simulated. We define the offered load μ as the ratio of the number of units demanded to the number of units in the network. The average number of demanded units is $\lambda\delta\alpha\gamma$, since there are $\lambda\delta$ active connections (assuming every demand has a connection established), and since in an unloaded network a connection takes on average $\alpha\gamma$ units. The number of units in the network is $|E||\Omega|$, and so the offered load $\mu = \lambda\delta\alpha\gamma/(|E||\Omega|)$, from which (1) follows.

$$\lambda(\mu) = \frac{\mu|E||\Omega|}{\delta\alpha\gamma} \qquad (1)$$

We define the *network utilization* as the ratio of the number of units in use to the total number of units on all edges. We cannot directly control the network utilization, but only measure it in response to the offered load μ.

3) Signal modulation model: In EONs, signal modulation can be adapted to the quality of the optical path, which depends on the length of the path and the optical components traversed. If we assume that the quality of the optical path depends mostly on its length, then a modulation can be characterized by the reach, i.e., the maximum length of a path above which the modulation cannot be used, because the signal would suffer unacceptable bit error rate. The reach increases as the spectral efficiency of the modulation decreases.

In [21], the authors experimentally demonstrated that if, for a demand requesting bitrate b in b/s, the most spectrally-efficient modulation available of reach r_M requires $n_M(b)$ units, then a less spectrally-efficient modulation of reach r_m requires $(M + 1 - m)n_M(b)$ units, where $m = M, (M - 1), ..., 1$ is integer and is called the modulation level, and M is the modulation level of the most spectrally-efficient modulation considered. Reach r_m doubles for the next less spectrally-efficient modulation (i.e., m decreases), as given by (2). Therefore, for a path of length $r_M < d \leq r_1$, we need to use modulation level m given by (3), derived from (2) with the assumption that m is integer. For $d \leq r_M$ we use $m = M$, and for $r_1 < d$ we have no modulation available.

$$r_m = r_M 2^{M-m} \qquad (2)$$

$$m(d) = M + 1 - \lceil log_2(2d/r_M) \rceil \qquad (3)$$

However, the assumption that the number of required units is an integer multiple of $n_M(b)$, because m is integer, is too strict. The bit error rate, which is increasing with the increasing path length d, can be lowered by using more units for the overhead of the error correction codes. In the most general

---

Algorithm 3 decide
In: label l'
Out: boolean

// Make sure that l' has the required number of units.
return $n(b, \text{cost}(l')) \leq |\text{CU}(l')|$

---

case, the number of required units should increase by one. For this reason, we allow the number of required units to be any integer from $n_M(b)$ to $Mn_M(b)$ depending on distance $0 \leq d \leq r_1$, as given by (4).

$$n(b,d) = \begin{cases} n_M(b) & \text{if } d \leq r_M \\ \infty & \text{if } r_1 < d \\ \lceil n_M(b)log_2(2d/r_M) \rceil & \text{otherwise} \end{cases} \qquad (4)$$

The decision function in Algorithm 3 uses (4) to check whether candidate label l' is able to support the demand of bitrate b, i.e., whether l' has at least the number of contiguous units required for bitrate b at the cost (distance) of l'.

In our simulations we assumed r_1 equals the length of the longest of all shortest paths multiplied by 1.5, which allows us to consider paths which were far longer than an average shortest path. Using (2), we calculated r_M for (4). We tried to increase the multiple from 1.5 to 2.0, but the brute-force algorithm would cause the simulations to run out of memory. We assumed $M = 4$.

B. Runs and populations

A *simulation run* simulated 100 days of a network in operation. The parameters of a simulation run were: the network graph, the number of units $|\Omega|$, the mean number of demanded units γ, the offered load μ, and the mean connection holding time δ. A simulation run reported the mean network utilization, and, for each of the algorithms evaluated, the mean and maximum memory used, and the mean and maximum times taken by a shortest-path search.

When a demand arrived, we searched for an optical path using all evaluated algorithms. We made sure that either all algorithms found no result, or that all results found were of the same cost and the same number of contiguous units. The result of the proposed algorithm was used to establish a connection.

During a shortest-path search we measured the maximum number of 32 bit words required by the largest data structures of the algorithm evaluated: the sets of tentative and permanent labels of the proposed algorithm, the vertex labels and the priority queue of the filtered-graphs algorithm, and the priority queue of the brute-force algorithm. We tracked separately the words required to store the costs, edges, and units. We stored a cost in one word, an edge in two words, a single unit in one word, and a CU in two words. Using the obtained memory measurements of a shortest-path search, we calculated the mean and maximum memory used by each algorithm throughout a simulation run. With careful implementation and testing, we made sure that the memory measurement had negligible effect on the time measurement.

Time measurement required special attention, because we ran simulations using a supercomputing infrastructure. While we were able to select one specific hardware type for all our simulations, we had little control over how much the hardware was loaded with the processes of other users, which could have severely degraded the performance of our simulations. For this reason we repeated a simulation run after a few hours, and for every time measurement, we took the minimum of the two values obtained, based on which we calculated the mean and maximum time taken by each of the algorithms throughout a simulation run.

We are interested in the results for a statistical population of simulation runs, rather than in the results for a single simulation run only, because a simulation run could have been an outlier with unusual results due to its randomly-generated network and traffic. To estimate a mean result for a population, we carried out the simulation runs which were the population samples, and calculated a *sample mean* of all the mean results reported by the simulation runs. We estimated the pessimistic algorithm performance of a population by a *sample maximum*, which is the maximum of the maximum results reported by the simulation runs.

In a given population there were 100 simulation runs whose parameters differed only with the network model. Hence, we had 100 Gabriel graphs generated, and used for every population. We had 102 populations, because we varied 3 values of the number of units $|\Omega|$ (160, 320, 640), 17 values of the offered load μ (0.05, 0.075, 0.1, 0.125, 0.15, 0.175, 0.2, 0.3, 0.45, 0.55, 0.65, 0.75, 1, 1.25, 1.5, 1.75, and 2), and two values of the mean number of demanded units γ (1, 10). For all populations, the mean connection holding time $\delta = 10$ days was constant. In total we carried out 10200 simulation runs (102 populations $\times$ 100 samples), which then we repeated. For the proposed algorithm and the filtered-graphs algorithms, the sample means credibly estimate the mean results of populations, since their relative standard error is usually around 1%. For the brute-force algorithm, the sample means frequently have the relative standard error of around 20%.

C. Simulation results

Figure 3 shows the sample means and the sample maxima of the time taken and memory used by a shortest path search, regardless of whether the search was successful or not. The results are shown on a logarithmic scale as a function of network utilization. The curves are plotted solid for the proposed algorithm, dashed for the filtered-graphs algorithm, and dotted for the brute-force algorithm. The sample means are plotted thin, and the sample maxima thick. Each curve is drawn using 17 data points for different values of μ. The error bars of the 95% confidence interval, appear only for the sample means of the brute-force algorithm, since their relative standard error was high, frequently around 20%, while for the other sample means the error bars were too small to plot.

Figure 3 has 12 subfigures in four rows and three columns. The first and the second rows show the time results for $\gamma = 1$, and $\gamma = 10$, respectively, and the third and fourth rows show

the memory results for $\gamma = 1$, and $\gamma = 10$, respectively. The first column shows the results for $|\Omega| = 160$, the second for $|\Omega| = 320$, and the third for $|\Omega| = 640$. We use the same scales in the various plots to allow for easy comparison.

The sample mean time results for $\gamma = 1$ show that the proposed algorithm was usually about 10 times faster and at most 20 times faster than two other algorithms, except at very heavy utilization, where most searches ended up with no solution, and where the brute-force algorithm was able to determine this more quickly. As the utilization increased, the mean time of every algorithm decreased, as the solution was less likely to exist. As the number of units increased from 160 to 320, and from 320 to 640 the mean time results increased about twice for the proposed algorithm and the filtered-graphs algorithm, and stayed about the same for the brute-force algorithm.

The sample mean time results for $\gamma = 10$ show that the proposed algorithm *was hundreds of times* faster than two other algorithms, and for the case of 640 units and light utilization, the proposed algorithm was about 500 times faster than the filtered-graphs algorithm.

As to the pessimistic time performance, i.e., the sample maximum time taken, our algorithm and the filtered-graphs algorithm usually took a few seconds, while the brute-force algorithm usually took hundreds of seconds, which makes a gap of *two orders of magnitude*.

The memory results are clear-cut: the filtered-graphs algorithm performed the best (since it used the Dijkstra algorithm), our algorithm performed very well, and the brute-force algorithm performed the worst. While the sample mean memory results of the brute-force algorithm are acceptable (even better than those of the proposed algorithm for heavy utilization), the sample maximum memory results reveal the unacceptable memory performance of the brute-force algorithm: the brute-force algorithm required about *six orders of magnitude* more memory than the proposed and the filtered-graphs algorithms.

As the number of units increased from 160 to 320, and from 320 to 640 the mean memory results increased almost twice for our algorithm, for the filtered-graphs algorithm the results expectedly stayed the same, and for the brute-force algorithm increased about 10%. The memory used by our algorithm and the brute-force algorithm were smaller for $\gamma = 10$ in comparison with $\gamma = 1$, because the spectrum was fragmented less.

Figure 4 shows the stacked plots of the maximum number of required memory words by our algorithm (Fig. 4a), the filtered-graphs algorithm (Fig. 4b), and the brute-force algorithm (Fig. 4c). In each of the plots there are 17×3 data points, because for the 17 values of the offered load μ we report the maximum number of required costs, edges, and units. The results are at the order of 10^4 for our algorithm, 10^2 for the filtered-graphs algorithm, and 10^9 for the brute-force algorithm. Clearly, the filtered-graphs algorithm performs best (requiring at most 1.2 kB), followed by our algorithm (requiring at most 160 kB), and then followed by the brute-force algorithm (requiring at most 10 GB). For our algorithm, the memory was used in 20% for costs, in 40% for edges, and 40% for units, since a label has one cost (one word), one edge

(a) time taken for $\gamma = 1$, and 160 units

(b) time taken for $\gamma = 1$, and 320 units

(c) time taken for $\gamma = 1$, and 640 units

(d) time taken for $\gamma = 10$, and 160 units

(e) time taken for $\gamma = 10$, and 320 units

(f) time taken for $\gamma = 10$, and 640 units

(g) memory used for $\gamma = 1$, and 160 units

(h) memory used for $\gamma = 1$, and 320 units

(i) memory used for $\gamma = 1$, and 640 units

(j) memory used for $\gamma = 10$, and 160 units

(k) memory used for $\gamma = 10$, and 320 units

(l) memory used for $\gamma = 10$, and 640 units

——— proposed algorithm, mean - - - filtered-graphs algorithm, mean ········ brute-force algorithm, mean

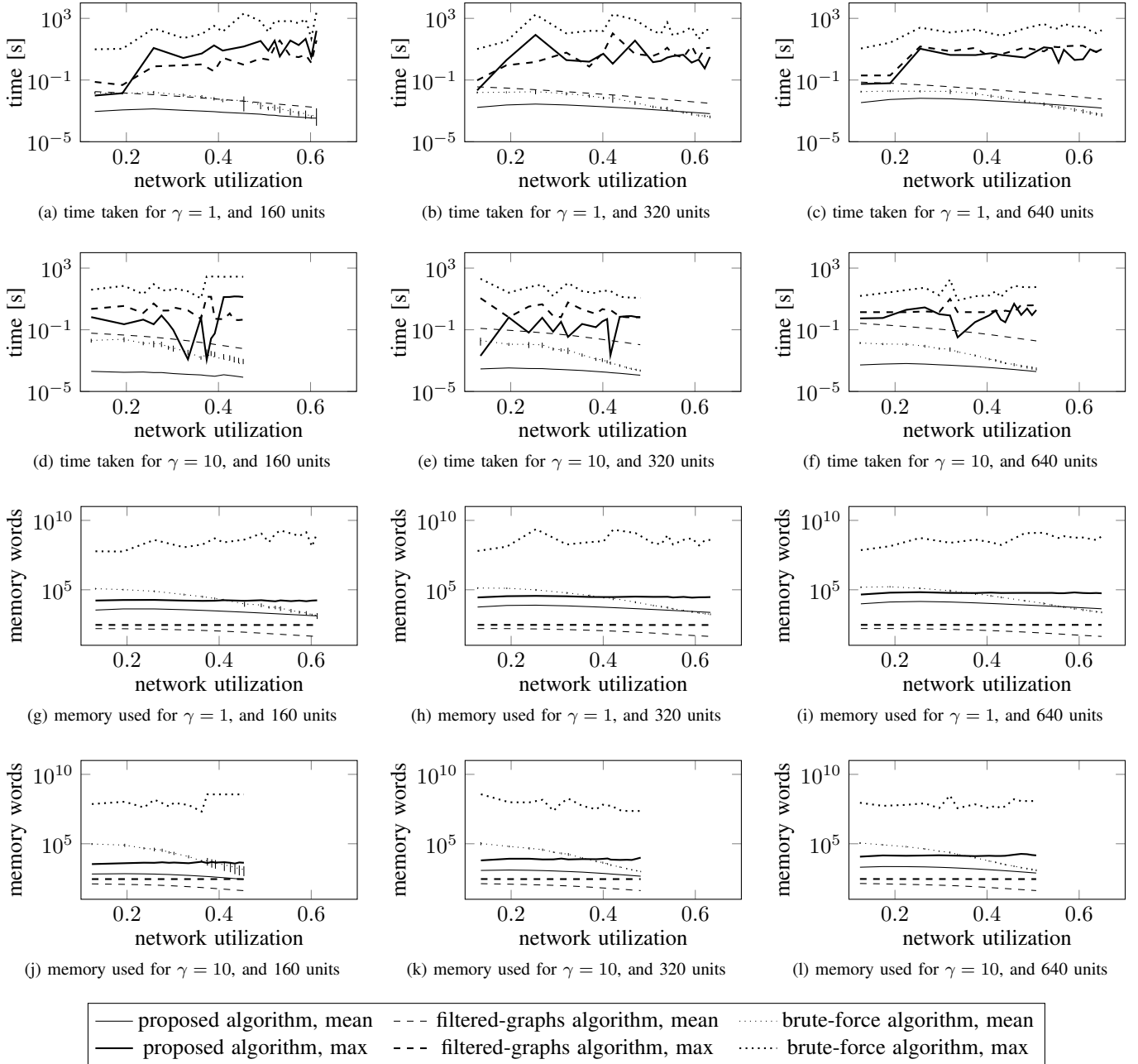——— proposed algorithm, max - - - filtered-graphs algorithm, max ······ brute-force algorithm, max

Fig. 3: Simulation results: the sample means and maxima of the time taken and memory used by the evaluated algorithms.

TABLE III: Summary of algorithm comparison.

Algorithm	Pros	Cons
generic Dijkstra	fastest	complex
filtered-graphs	ultra-low memory usage	slow
brute-force	good for small networks	fails for large networks

(two words), and one CU (two words). For the filtered-graphs algorithm, the memory was used mainly to store edges, and costs, since a label has one cost and one edge. The brute-force algorithm used most of its memory to store the units of the paths in the priority queue.

Table III summarizes the algorithm comparison.

VI. CONCLUSION

We proposed a novel generalization of the Dijkstra shortest path algorithm for finding a shortest path in the wavelength-division multiplexed networks and the elastic optical networks.

Our extensive simulation studies show that the proposed algorithm has a small memory footprint and is considerably (even hundreds of times) faster than other routing algorithms that are frequently utilized.

We provide no proof of correctness or complexity analysis. However, with robust simulations, we corroborated the correctness and efficiency of the proposed algorithm.

We presented the algorithm in the setting of optical networks, but we believe that the novel ideas we introduced

(a) Proposed algorithm.

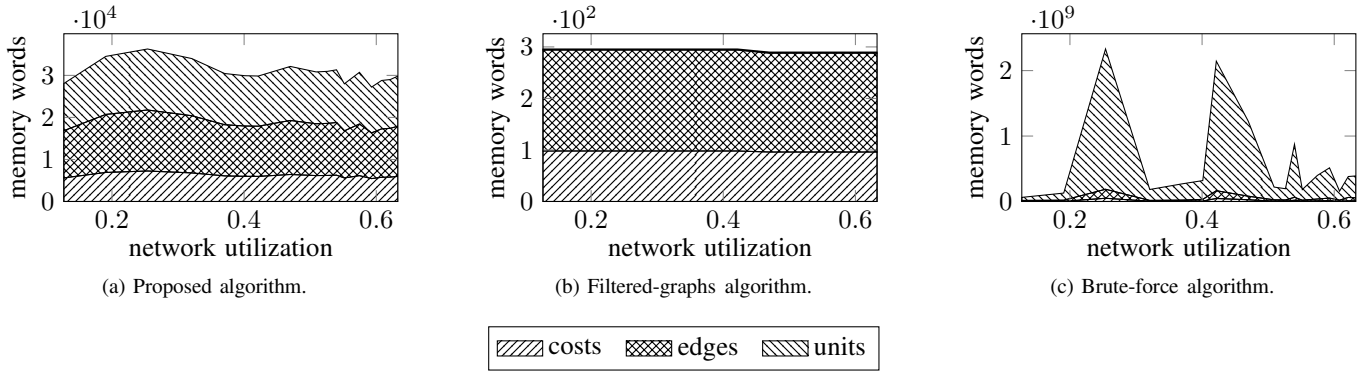(b) Filtered-graphs algorithm.

(c) Brute-force algorithm.

Fig. 4: Simulation results: the maximum number of required words by the evaluated algorithms.

can be applied in the routing in the multilayer and wireless networks to make their control quicker. For instance, the algorithm could be used to solve efficiently the contiguous frequency and time resource allocation in the wireless orthogonal frequency-multiplexed wireless networks.

There are a number of directions for future work. First, the algorithm could be adapted for parallel execution, making it even faster. Second, the algorithm could be turned into a distributed algorithm, very much like a distance-vector algorithm. Next, the algorithm could be extended further to be applicable to multilayer networks or space-division multiplexed networks. Finally, the algorithm could be extended to take into account signal regeneration, spectrum conversion, or inverse multiplexing.

References

[1] I. Szcześniak and B. Woźna-Szcześniak, "Adapted and constrained Dijkstra for elastic optical networks," in *2016 International Conference on Optical Network Design and Modeling (ONDM)*, May 2016, pp. 1–6.

[2] I. Szcześniak. (2018, October) The implementation of the generic Dijkstra. [Online]. Available: http://www.irkos.org/gd

[3] ——. (2013, August) The SDI software. [Online]. Available: http://www.irkos.org/sdi

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[5] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*. Addison-Wesley Professional, 2014.

[6] H. Zang, J. P. Jue, and B. Mukherjee, "A review of routing and wavelength assignment approaches for wavelength-routed optical WDM networks," *Optical Networks Magazine*, vol. 1, pp. 47–60, 2000.

[7] S. Talebi, F. Alam, I. Katib, M. Khamis, R. Salama, and G. N. Rouskas, "Spectrum management techniques for elastic optical networks: a survey," *Optical Switching and Networking*, vol. 13, pp. 34 – 48, 2014.

[8] Y. Wang, X. Cao, and Y. Pan, "A study of the routing and spectrum allocation in spectrum-sliced elastic optical path networks," in *2011 Proceedings IEEE INFOCOM*, April 2011, pp. 1503–1511.

[9] B. C. Chatterjee, N. Sarma, and E. Oki, "Routing and spectrum allocation in elastic optical networks: A tutorial," *IEEE Communications Surveys Tutorials*, vol. 17, no. 3, pp. 1776–1800, third quarter 2015.

[10] F. S. Abkenar and A. G. Rahbar, "Study and analysis of routing and spectrum allocation (RSA) and routing, modulation and spectrum allocation (RMSA) algorithms in elastic optical networks (EONs)," *Optical Switching and Networking*, vol. 23, pp. 5 – 39, 2017.

[11] I. Olszewski, "Improved dynamic routing algorithms in elastic optical networks," *Photonic Network Communications*, vol. 34, no. 3, pp. 323–333, Dec 2017.

[12] L. Velasco, M. Klinkowski, M. Ruiz, and J. Comellas, "Modeling the routing and spectrum allocation problem for flexgrid optical networks," *Photonic Network Communications*, vol. 24, pp. 177–186, December 2012.

[13] X. Wang, K. Kuang, S. Wang, S. Xu, H. Liu, and G. N. Liu, "Dynamic routing and spectrum allocation in elastic optical networks with mixed line rates," *Journal of Optical Communications and Networking*, vol. 6, no. 12, pp. 1115–1127, December 2014.

[14] X. Wan, N. Hua, and X. Zheng, "Dynamic routing and spectrum assignment in spectrum-flexible transparent optical networks," *Journal of Optical Communications and Networking*, vol. 4, no. 8, pp. 603–613, August 2012.

[15] K. Christodoulopoulos, P. Kokkinos, and E. M. Varvarigos, "Indirect and direct multicost algorithms for online impairment-aware RWA," *IEEE/ACM Transactions on Networking*, vol. 19, no. 6, pp. 1759–1772, Dec 2011.

[16] E. M. Varvarigos, V. Sourlas, and K. Christodoulopoulos, "Routing and scheduling connections in networks that support advance reservations," *Computer Networks*, vol. 52, no. 15, pp. 2988 – 3006, 2008.

[17] F. Gutierrez, E. Varvarigos, and S. Vassiliadis, "Multi-cost routing in max-min fair share networks," in *Proc. 38th Annual Allerton Conference on Communications, Control, and Computing*, Monticello, USA, October 2000, pp. 1294–1304.

[18] L. Velasco, A. Castro, M. Ruiz, and G. Junyent, "Solving routing and spectrum allocation related optimization problems: from off-line to in-operation flexgrid network planning," *Journal of Lightwave Technology*, vol. 32, no. 16, pp. 2780–2795, Aug 2014.

[19] A. A. Stepanov and P. McJones, *Elements of programming*. Addison-Wesley Professional, 2009.

[20] E. Cetinkaya, M. Alenazi, Y. Cheng, A. Peck, and J. Sterbenz, "On the fitness of geographic graph generators for modelling physical level topologies," in *5th ICUMT*, Sept 2013, pp. 38–45.

[21] 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 Communications Magazine*, vol. 48, no. 8, pp. 138–145, 2010.