Parameterized Maximum Path Coloring

Michael Lampis

Graduate Center, City University of New York
mlampis@gc.cuny.edu

Abstract. We study the well-known Max Path Coloring problem from a parameterized point of view, focusing on trees and low-treewidth networks. We observe the existence of a variety of reasonable parameters for the problem, such as the maximum degree and treewidth of the network graph, the number of available colors and the number of requests one seeks to satisfy or reject. In an effort to understand the impact of each of these parameters on the problem's complexity we study various parameterized versions of the problem deriving fixed-parameter tractability and hardness results both for undirected and bi-directed graphs.

1 Introduction

The Path Coloring (PC) and Maximum Path Coloring (MaxPC) problems are two well-known and widely studied combinatorial problems with applications in the field of optical networks. In PC we are given a graph representing the optical network and a set of paths on that graph and are asked to find a coloring of the paths such that any two paths which share an edge have distinct colors and the number of colors used is minimized. In the MaxPC problem on the other hand we are given a specified number of colors and must select a maximum cardinality set of paths which can be properly colored with the available colors. If the graph contains cycles we may alternatively be given the endpoints of the communication requests only, with the flexibility to choose the most suitable path for each. Then the problem is often called Routing and Path Coloring (RPC and MaxRPC). Of course, if the underlying graph is a tree the two versions of the problems are equivalent.

PC is unfortunately known to be hard to solve exactly even on very simple topologies and therefore the same holds for MaxPC. As a consequence the vast majority of research on the two problems has focused on coming up with good approximation algorithms for either minimizing the number of colors or maximizing the number of accepted requests. In this paper, however, we investigate the complexity of solving MaxPC on trees and tree-like graphs exactly, from the point of view of parameterized complexity theory. (For an introduction to parameterized complexity theory and the theory of fixed-parameter tractable (FPT) algorithms see [4, 17, 9]).

The main observation we want to exploit is that MaxPC is a problem rich with reasonable parameters. For example in practical situations one may often expect that the network will have moderate maximum degree and it will be a

tree or perhaps "tree-like". Furthermore, technological limitations mean that the number of available colors on each edge is also likely to be moderate. Also, as observed in [1], communications networks are often built with maximum capacity in mind, meaning that typically the available resources should be enough to satisfy all or almost all requests. Interestingly, nothing prevents several of these facts from happening together. This motivates the study of the problem through a parameterized lens: one identifies a parameter (or set of parameters) expected to be small and then attempts to design an FPT algorithm for this particular parameterization or prove that none exists. For example, if one is interested in instances with moderate maximum degree Δ, the goal is to design an algorithm with running time $f(\Delta) \cdot n^c$, for some moderately exponential function f or to prove that no such algorithm exists (but perhaps an algorithm running in time $n^{g(\Delta)}$ is possible).

Of course, the observation that Δ or some other parameters may be small in practice is not new; in fact the traditional complexity of PC has been investigated for bounded degree trees for example. The contribution of this paper is that, in addition to giving new results it puts these known results under the light of parameterized complexity, where $f(\Delta) \cdot n^c$ and $n^{g(\Delta)}$ algorithms are considered completely different cases with only the first called tractable, whereas for traditional complexity both are "polynomial for fixed Δ". This allows us to more systematically enter more parameters into the problem and better assess their impact on complexity. Interestingly, it also leads us to discover some interesting gaps between the complexity of PC and MaxPC which were previously overlooked in the literature.

Previous work PC and MaxPC are very well-studied problems, starting from the 1980s (see [12]). As mentioned, when the network graph contains cycles one may consider either the case where requests are pre-routed or where routing is part of the problem. Furthermore, the communication network can either be assumed to be undirected or bi-directed, where in the second case every request has a direction and two requests with the same color can share an edge if they use it in opposite directions.

PC is known to be hard even in very restricted topologies, from which fact the hardness of MaxPC also follows trivially. Specifically, PC is NP-hard for undirected stars by equivalence to edge coloring in multi-graphs [6], undirected rings (here the problem is equivalent to coloring circular-arc graphs [10]) and bi-directed binary trees [6, 15]. However, it is known to be FPT in undirected trees when parameterized by the maximum degree of the tree Δ [6, 15], and also to be FPT in bi-directed trees when parameterized by the maximum number of requests touching any node [6]. A 4/3-approximation algorithm is known for PC in undirected trees [6] and a 5/3-approximation for bi-directed trees [8].

For MaxPC a 2.22-approximation is known for bi-directed trees [5] and a 1.58-approximation is known for undirected trees [18]. For bi-directed trees it is also known that MaxPC is solvable in polynomial time if both the maximum degree and the number of colors are constant [5], a result which can be extended

to undirected trees in a straightforward manner. Note though that this is an XP, not an FPT algorithm, a fact that we will return to later. For the special case where only one color is available the problem is also known as Maximum Edge Disjoint Paths (MaxEDP), and is known to be NP-hard for bi-directed trees [7] but in P for undirected trees [11].

To the best of our knowledge PC and MaxPC have not been explicitly studied before from a parameterized perspective, though there are results in the literature which can be translated to the fixed-parameter tractability terminology, such as the algorithm for PC on bounded degree undirected trees mentioned above. For the related Call Control problem however there has been an investigation of its complexity when parameterized by the number of rejected requests [1], which is one of the parameters we consider in this paper as well. Note though that the situations for Call Control and MaxPC are quite different for this parameter, as MaxPC is usually hard even when no requests can be rejected (this is the PC problem), while Call Control is easy in that case. Thus, for MaxPC this parameter can only be useful in combination with other parameters that make PC fixed-parameter tractable.

Contributions of this paper In this paper we study several parameterized versions of MaxPC mainly on trees. First, we study parameterizations which do not involve the objective function, that is, the number of requests to be satisfied or rejected. Specifically, for trees the parameters we consider are the maximum degree Δ and the number of available colors W. As mentioned, for undirected trees PC is FPT parameterized by Δ alone while for bi-directed trees PC is FPT parameterized by Δ and W, from the FPT result when parameterized by the maximum number of requests touching a node. However, for MaxPC all that is known is an XP algorithm running in roughly $n^{\Delta W}$ time. From the traditional complexity perspective it is easy to overlook the difference as in both cases we have algorithms polynomial for fixed Δ and W. However, from the parameterized complexity perspective it is natural to ask why no FPT algorithm is known for MaxPC and whether this can be fixed by designing an algorithm that would take at least one or ideally both of the parameters out of the exponent of n. We resolve this question fully by showing that neither parameter can be removed from the exponent of n, under standard complexity assumptions, even if the other is a small constant. This points out the existence of a (previously unknown) gap between the complexity of PC and MaxPC. In particular our results imply that MaxPC is NP-complete even on binary trees (where PC is solvable in polynomial time), which to the best of our knowledge was not known before. They also show that the complexity of MaxPC grows much faster as Δ and W grow than the complexity of PC.

Continuing this line of reasoning we observe that the $n^{\Delta W}$ algorithm can be extended in a straightforward way to graphs of treewidth t[1], running in time

[1] The treewidth of a graph is a measure which estimates how "tree-like" the graph is; for more information see [2]

roughly $n^{\Delta W t}$. This poses the new problem of whether at least t can be moved out of the exponent which we again resolve negatively.

One intuitive explanation for the complexity gap between PC and MaxPC is that in the instances of our reductions a large fraction of the requests must be rejected, making the situation very different from PC where all requests must be satisfied. Thus, we are led to add as another parameter the number of rejected requests. In this case the complexity gap (at least partially) closes again: we show that for both undirected and bi-directed trees the MaxPC problem is FPT if one considers as parameters Δ, W and the number of requests which can be rejected. Also we show that for undirected binary trees MaxPC is FPT parameterized by the number of rejected requests.

Finally, we consider the naturally parameterized version of MaxPC (that is, parameterized by the size of the solution) and show that it is FPT on any topology where the naturally parameterized version of MaxEDP is FPT, using a color-coding technique. From this, it immediately follows that this parameterization of MaxPC is FPT for undirected trees and for rings, since in those cases MaxEDP is solvable in polynomial time. For bi-directed trees, where MaxEDP is NP-hard, we show that its naturally parameterized version is FPT, a result which may be of independent interest, thus settling the fixed-parameter tractability of MaxPC in this case as well.

2 Definitions and Preliminaries

In this paper we discuss the Path Coloring problem (PC) and its corresponding maximization problem MaxPC. Our main topic is their restriction to trees. The input we are given in this case consists of an undirected tree $G(V, E)$ and a multi-set of demands $D \subseteq V \times V$, each demand corresponding to the unique path in G that connects its two vertices. We are also given two integers W (the number of colors) and B (the number of demands we seek to satisfy). The question is whether there exist W mutually disjoint subsets $D_1, D_2, \ldots, D_W \subseteq D$ s.t. no set D_i contains two demands that share an edge and $\sum_{i=1}^{W} |D_i| \geq B$.

In other words we are asked if there exists a W-colorable set of at least B paths from the set of the given demands. This problem, where we seek to maximize B is usually called MaxPC, while PC is simply the special case when $B = |D|$. The graph G can either be considered undirected, in which case the ordering of each demand pair is irrelevant, or bi-directed, in which case two satisfied demands with the same color are allowed to use the same edge but only in opposite directions (another way to think of this is as replacing every undirected edge with two parallel arcs of opposite directions). In this paper we will deal with both undirected and bi-directed graphs.

The problems can be generalized to graphs that contain cycles. Here, we will focus on the case where for each demand we are given the path that it must follow on the graph, but also briefly mention how our results can be extended to the case where routing is part of the problem.

We will use Δ to denote the maximum degree of G, n to denote the number of vertices, t to denote the treewidth of G (which is of course 1 if G is a tree) and $T = |D| - B$ to denote the number of demands we are allowed to reject. We will consider various tractable special cases and parameterizations of MaxPC, for example on trees of bounded degree, or in instances with a small number of colors (we assume here that the reader is familiar with the basic definitions of parameterized complexity theory, such as the class FPT). The candidate parameters we are interested in are Δ, t, W, B and T. To keep the presentation short and concise we will use a notation where different parameterizations of MaxPC are denoted by prepending it with the list of variables we consider constant or parameters. For example, the $(p\Delta)-$MaxPC problem is the parameterized version of MaxPC when Δ is our only parameter, while $(pB)-$MaxPC is the parameterized version where B is the parameter (the "naturally" parameterized version of MaxPC). The reason for this notation is that we will consider various combinations of parameters and also cases where some values are parameters and some others are fixed constants. For example $(pW, c\Delta)-$MaxPC is the special case of $(pW)-$MaxPC restricted to bounded degree trees. Observe that this is not the same as the problem $(pW, p\Delta)-$MaxPC since a hypothetical algorithm running in time say $2^W n^\Delta$ is FPT for the first problem but not for the second.

Our aim here is to investigate how different parameters (and combinations of parameters) affect the complexity of the problem. Table 1 contains a summary of some of the already known results on the complexity of PC and MaxPC and the results of this paper. Worthy of note is the contrast between some already known tractable cases of PC ($(p\Delta)$-PC for undirected trees and $(p\Delta, pW)$-PC for bi-directed trees) and the hardness we establish for the corresponding cases of MaxPC. Interestingly, tractability returns if we add T as a parameter to $(pW, p\Delta)$-MaxPC in both bi-directed and undirected trees, which makes intuitive sense since T quantifies the "distance" between a PC and a MaxPC instance. We can also prove that (pT)-MaxPC is FPT on undirected binary trees.

3 Structural Parameterizations

In this section we investigate parameterizations which do not involve the objective function. The candidate parameters will be the maximum degree Δ, the input graph's treewidth t and the number of available colors W. Some fixed-parameter tractability results are known in the case of PC for these cases, but unfortunately for the corresponding cases of MaxPC only XP algorithms are known and as we will show this can probably not be improved.

First, recall that in [5] it was shown that MaxPC can be solved in polynomial time on bi-directed trees if both Δ and W are constant. The basic idea is a bottom-up dynamic programming technique which can be extended in a straightforward way to undirected trees also. Our first observation is that this idea can in fact be extended to graphs of bounded treewidth as well.

Undirected			Bi-Directed		
Problem	Result	Comment	Problem	Result	Comment
(cW)-PC, $W = 3$	NP-h	(edge 3-coloring) [6]	(cW)-PC, $W = 1$	NP-h	[6]
$(p\Delta)$-PC	FPT	[6]	$(c\Delta)$-PC, $\Delta = 3$	NP-h	[6]
			$(pW, p\Delta)$-PC	FPT	[6]
$(c\Delta, ct)$-PC	NP-h	(PC on rings) [10]	$(c\Delta, ct)$-PC	NP-h	[10]
$(cW, c\Delta)$-MaxPC	P	[5]	$(cW, c\Delta)$-MaxPC	P	[5]
$(cW, c\Delta, ct)$-MaxPC	P	Theorem 1	$(cW, c\Delta, ct)$-MaxPC	P	Theorem 1
$(pW, c\Delta)$-MaxPC	W[1]-h	Theorem 2	$(pW, c\Delta)$-MaxPC	W[1]-h	Theorem 2
$(cW, p\Delta)$-MaxPC	W[1]-h	Theorem 3	$(cW, p\Delta)$-MaxPC	W[1]-h	Theorem 3
$(cW, c\Delta, pt)$-MaxPC	W[1]-h	Theorem 4	$(cW, c\Delta, pt)$-MaxPC	W[1]-h	Theorem 4
$(pW, p\Delta, pT)$-MaxPC	FPT	Theorem 5	$(pW, p\Delta, pT)$-MaxPC	FPT	Theorem 5
(pT)-MaxPC, $\Delta = 3$	FPT	Theorem 6			
(pB)-MaxPC	FPT	Corollary 1	(pB)-MaxPC	FPT	Corollary 2

Table 1. Summary of results. All results concern trees, except those where the graph's treewidth t is included in the problem description.

Theorem 1. $(cW, c\Delta, ct)$-MaxPC *can be solved in polynomial time for both undirected and bi-directed graphs.*

Theorem 1 essentially applies common dynamic programming techniques associated with treewidth to obtain an XP algorithm. The algorithm is likely to be extremely impractical though, even for small values of the parameters, since the exponent relies on all three. So the natural, and more important question to ask is whether any kind of fixed-parameter tractability result can be obtained.

Ideally, one would like an FPT algorithm running in time $f(W, \Delta, t) \cdot n^c$, that is, an FPT algorithm for $(pW, p\Delta, pt)$-MaxPC. Barring that, it would still be helpful if any one of the three parameters could be moved out of the exponent of n, even by itself. Unfortunately, we resolve this problem in a negative way, showing that even if any two of the parameters are small fixed constants (and are therefore allowed to appear in the exponent of n in an FPT algorithm) it is still impossible to obtain an FPT algorithm for the problem, under standard complexity assumptions. We prove this by using three parameterized reductions.

The reductions presented here will use a slightly more general problem we will call CapMaxPC. In this problem, for each edge $e \in E$ we are given an integer capacity $1 \leq c(e) \leq W$ and have the additional constraint that in a feasible solution at most $c(e)$ satisfied demands may be using e. For parameterizations not involving the objective function this problem is shown FPT-reducible to MaxPC by using a simple trick where limited edge capacity on an edge is simulated by adding an appropriate number of length 1 demands going through the edge.

We will also use another intermediate problem in our reductions, which we will call Disjoint Neighborhoods Packing (DNP). In DNP we are given an undirected graph $G(V, E)$ and are asked to find a maximum cardinality set $V' \subseteq V$ such that $\forall u, v \in V'$ we have $N(u) \cap N(v) = \emptyset$ (we denote by $N(u)$ the set that contains u and all its neighbors, that is, the closed neighborhood of u).

The parameter we consider is the size of V'. This problem is sometimes referred to in the literature as 2-Independent Set, see [14].

Overall our strategy is to start from the well-known W[1]-hard problem Independent Set and present reductions to our problems through the two intermediate problems described above, that is, we aim to prove that IS $\leq_{FPT}$ DNP $\leq_{FPT}$ CapMaxPC $\leq_{FPT}$ MaxPC. The trickiest step in this process will be the second reduction, where we will show three different versions, one for each parameterization of MaxPC we are interested in.

Lemma 1. *For both undirected and bi-directed graphs we have*

- $(pW, c\Delta)$-CapMaxPC $\leq_{FPT}$ $(pW, c\Delta)$-MaxPC ‘
- $(cW, p\Delta)$-CapMaxPC $\leq_{FPT}$ $(cW, p\Delta)$-MaxPC
- $(cW, c\Delta, pt)$-CapMaxPC $\leq_{FPT}$ $(cW, c\Delta, pt)$-MaxPC

Lemma 2. DNP *is W[1]-hard.*

Proof. We present a reduction from the Independent Set problem. Given a graph $G(V, E)$ and assuming without loss of generality that it has no isolated vertices and we are looking for an independent set of size $k > 2$ in G, we will construct an equivalent instance of DNP. First, subdivide every edge of G, that is, replace each $(u, v) \in E$ with a path of length 2. Connect all newly added vertices into a clique. We will argue that the new graph has a packing of k disjoint neighborhoods iff the original graph has an independent set of size k.

If the original graph has an independent set of size k this immediately gives us a packing of the same size on the new graph by selecting the same vertices. The packing is valid since the only way two of the original vertices could have a common neighbor in the new graph is if one of the vertices introduced in the subdivisions is connected to both and that can only happen if an edge was connecting them in the original graph.

If the new graph has a packing of $k > 2$ disjoint neighborhoods given by the set of vertices V', then we can immediately infer that V' cannot include two or more of the vertices introduced in the subdivisions, since they are all connected in a clique. If V' contains one of these new vertices, say the one introduced in the subdivision of (u, v) (call that vertex w) then it cannot contain any vertices in $V \setminus \{u, v\}$ because every original vertex is connected to at least one new vertex and that vertex is connected to w. V' may also contain at most one of $\{u, v\}$, so its total size cannot be more than 2 in this case. We conclude that a packing of $k > 2$ disjoint neighborhoods must consist entirely of vertices found in the original graph. To see that these form an independent set in the original graph, observe that if two were originally connected they would have a common neighbor in the new graph, violating the feasibility of the packing.

□

Theorem 2. $(pW, c\Delta)$-MaxPC *is W[1]-hard for both undirected and bi-directed trees.*

Proof. Given Lemma 1 and Lemma 2 the only thing left to prove is that DNP $\leq_{FPT}$ $(pW, c\Delta)$-CapMaxPC. Given an instance of DNP, that is a graph $G(V, E)$ and a target size for the DNP set k, we construct a CapMaxPC instance as described below. We first show the reduction for undirected trees and then describe how it can be made to work for bi-directed trees as well.

First, let $|V| = n$ and we construct a "backbone", which is simply a path on $n + 2$ vertices. We take $n + 2$ disjoint copies of a path on n vertices and attach one of the endpoints of each to one of the vertices of the backbone so that each backbone vertex now has a path hanging from it. Label the backbone vertices $b_i, 0 \leq i \leq n+1$ and the vertices of the other paths $p_{i,j}, 0 \leq i \leq n+1, 1 \leq j \leq n$, so that the path vertex connected to b_i is called $p_{i,1}$, its other neighbor is $p_{i,2}$ and so on. Finally, for each $1 \leq i, j \leq n$ we add three vertices in the graph $v_{i,j}, u_{i,j}$ and $w_{i,j}$ and the edges $(v_{i,j}, w_{i,j}), (u_{i,j}, w_{i,j})$ and $(w_{i,j}, p_{i,j})$. In other words, we construct a path on three vertices and connect the middle vertex to $p_{i,j}$. This completes the description of the graph, which is a tree of maximum degree 3.

Now let us describe the demands. Suppose that the vertices of the original graph are numbered $\{1, 2, \ldots, n\}$. For each $i \in V$ we consider the closed neighborhood $N(i)$ in increasing order and let $N(i) = \{j_0, j_1, j_2, \ldots, j_{d(i)}\}$, where $d(i)$ is the degree of i. We add a demand from $p_{0,i}$ to $u_{j_0,i}$. Then, for each $l, 0 \leq l < d(i)$ we add a demand from $v_{j_l,i}$ to $u_{j_{l+1},i}$. We also add a demand from $v_{j_{d(i)},i}$ to $p_{n+1,i}$. We add all these demands for each $i \in V$ and call these demands global demands. Finally, for each $1 \leq i, j \leq n$ we add a demand from $v_{i,j}$ to $u_{i,j}$. We call these demands local.

The only thing left is to specify W, which we set to $W = 2k$, and the capacities. We leave all capacities unconstrained except for the edges $(b_i, p_{i,1}), 1 \leq i \leq n$, which have a capacity of 2 and the edges $(u_{i,j}, w_{i,j})$ and $(v_{i,j}, w_{i,j})$ which have a capacity of 1. The construction is now complete.

To give some intuition about this construction, notice the interaction between local and global demands. Each local demand intersects exactly two global demands in edges of capacity 1. Thus, if the local demand is satisfied the global demands are rejected. Furthermore, if exactly one of the global demands is satisfied in a solution we can exchange it with the local demand, therefore this gadget ensures that either both global demands will be taken or both will be rejected in some optimal solution. Observe also that from all the local demands found in a branch attached to the backbone at most one will be rejected, since the edge $(b_i, p_{i,1})$ acts as a bottleneck allowing at most two global demands to go through. The idea will be that if a vertex i is in the neighborhood packing then we will select the global demand starting at $p_{0,i}$ and satisfy one after the other pairs of demands that go into branches that correspond to its neighbors, making these branches unusable for other global demands.

For a more precise argument, suppose that the original graph has a packing V' of size k, we will construct a CapMaxPC solution of size $n^2 + k$. Start with a solution of size n^2 by selecting all the local demands of the instance and nothing else. Now for each $i \in V'$ we will inrease the size of the solution by 1. We do this by satisfying all the global demands associated with i, that is, all demands

touching a vertex $p_{j,i}$ for any j. Each time we perform this improvement step we use two new colors (the two colors are sufficient to color the global demands since they form essentially a path) and remove from the solution all local demands that intersect with these global demands (it is not hard to see that this gives a profit of exactly one demand). Since we are using different colors in each step the only way this process could run into a problem is in an edge where fewer than $2k$ colors can be used. For that to happen we must be trying to satisfy more than two requests going through an edge $(b_j, p_{j,1})$ but that would imply that j is a common neighbor of two vertices of the packing, violating its feasibility.

For the other direction, suppose that a solution of size $n^2 + k$ exists. As mentioned, if in a set of one local and its two intersecting global demands the solution satisfied exactly one of the global demands, we exchange it with the local demand. This means that for each edge $(b_i, p_{i,1})$ we are either satisfying two of the demands crossing it or none and furthermore that if we are satisfying two, one of them is going "left" (that is, its other endpoint is towards b_{i-1}) and the other is going "right" (so its other endpoint is towards b_{i+1}). Therefore, the number of satisfied requests going through each edge (b_i, b_{i+1}) is constant for all i; call this number L. We will establish that $L = k$. Pick an arbitrary satisfied demand which uses a backbone edge and delete it from the solution. This will reduce the size of the solution by one, but it will also allow us to reduce L by one, since by the same arguments used before we can make the number of satisfied demands on each backbone edge the same without affecting the size of the solution[2]. Repeat this process L times and now we have a solution which satisfies only local demands and has size $n^2 + k - L$. Since there are exactly n^2 local demands it must be the case that $k = L$. Now we can conclude that there are k vertices in the branch connected to b_0 whose demands are satisfied and all subsequent global demands associated with them are also satisfied. These give us a neighborhood packing in the original graph because if two of them had a common neighbor the solution would be exceeding some branch's bottleneck capacity of 2.

It is not hard to modify this reduction to also work for bi-directed trees. The only difference in the network is that edges $(b_i, p_{i,1})$ are given a capacity of 1, since they are intended to be traversed twice but in different directions, and that it is now sufficient to have $W = k$ since all the global demands corresponding to a vertex are non-intersecting. Other than that we remain consistent with the ordering that we have implied in our description, that is, every global demand is ordered towards the vertex that lies further to the right (the vertex closer to $p_{n+1,n}$ so to speak). We also make sure that the local demands are directed in such a way that they intersect both global demands with which they share an edge and the rest of the arguments of the reduction go through unchanged.

□

[2] This is implicitly relying on the fact that all global demands must intersect some local demand, which is true if the original graph had no isolated vertices

Theorem 3. $(cW, p\Delta)$-MaxPC *is W[1]-hard for both undirected and bi-directed trees. The result holds even for instances where all the vertices but one have degree bounded by 3.*

Proof. Once again we will describe a reduction from DNP, but now the produced instance will have maximum degree depending on k and constant W. We will reuse some of the ideas of Theorem 2, properly adjusted. Again we will first describe a construction for undirected graphs and then discuss how it can be modified for bi-directed graphs.

Take k copies of a path on n vertices and label the vertices $S_{i,j}, 1 \leq i \leq k, 1 \leq j \leq n$. Take k more copies and label the vertices $T_{i,j}, 1 \leq i \leq k, 1 \leq j \leq n$. Add a new vertex to the graph, call it C, and connect it to all $S_{i,n}$ and $T_{i,n}$ for $1 \leq i \leq k$. Set the capacities of all edges to 1. Also, for each $i, j, 1 \leq i \leq k, 1 \leq j \leq n$ add a demand from $S(i,j)$ to $T(i,j)$.

Before we go on, let us examine the construction so far. It should be clear that the optimal solution satisfies k paths by selecting k vertices in the S branches and their corresponding vertices in the T branches. The k selected vertices will eventually encode the vertices we will pick for our neighborhood packing. What is of course missing is some machinery to ensure that our selection is indeed a packing in the original graph.

The constraints of a valid packing can be broken down as follows: for each of the $\binom{k}{2}$ pairs of vertices selected for the packing we must make sure that they do not share common neighbors. Thus, our basic tool will be a gadget that takes two of the k choices we have made and checks their compatibility. We will make $\binom{k}{2}$ copies of that gadget, attach them to C and then properly reroute the demands from S to T vertices through these gadgets.

To describe the pairwise consistency gadget, consider the instance constructed in the proof of Theorem 2. We modify it as follows: First, we add local requests gadgets, identical as those used in vertices $p_{i,j}, 1 \leq i, j \leq n$ to the vertices of the paths p_0 and p_{n+1}. We extend all demands which currently had an endpoint in $p_{0,j}$ or $p_{n+1,j}$ for some $j \in \{1, \ldots, n\}$ to the vertices $v_{0,j}$ and $u_{n+1,j}$ respectively, so that they intersect the new local demands. Now we make an exact copy of the branch p_0 and all its connected gadgets (i.e. the vertices $p_{0,j}, u_{0,j}, v_{0,j}, w_{0,j}, 1 \leq j \leq n$). We call the new branch p'_0 (and the new vertices respectively $p'_{0,j}, u'_{0,j}, v'_{0,j}, w'_{0,j}, 1 \leq j \leq n$) and attach it also to b_0. We also make sure to replicate all demands that existed between the branch p_0 and the rest of the graph so that corresponding demands are placed between the branch p'_0 and the rest of the graph. We perform another full copy for the branch p_{n+1} producing the branch p'_{n+1} with identical vertices and demands and attach this to b_{n+1}. Now the whole gadget has $n(n+4)$ local demands overall. We set the capacities of all backbone edges to 4, all edges used by local demands to 1 and all other edges to 2.

To demonstrate the use of this gadget we will connect one such gadget on our initial construction and use it to ensure that in the optimal solution the choices encoded in the paths S_1 and S_2 (i.e. the encoding of the first two choices for the packing) are compatible. Take a gadget as described and connect its b_0 to

C by an edge of capacity 4. Recall that for all $j \in \{1, \ldots, n\}$ there is a demand from $S_{1,j}$ to $T_{1,j}$. Remove these n demands and for all $j \in \{1, \ldots, n\}$ add a demand from $S_{1,j}$ to $u_{0,j}$ and a demand from $v_{n+1,j}$ to $T_{1,j}$ (in other words we are rerouting the $S_1 \to T_1$ demands through the gadget). Do the same for demands from S_2 to T_2, only reroute them through the p'_0 and p'_{n+1} branches.

A solution of size $n(n+4)+k$ can be achieved now iff the selections for active vertices in S_1 and S_2 are compatible, that is, the corresponding vertices of the initial graph have no common neighbors. This follows from the analysis of the properties of our gadget performed in Theorem 2.

It is now possible to complete the construction by adding more of the consistency gadgets so as to make sure that all $\binom{k}{2}$ pairs of choices are compatible. The final graph consists of the $\binom{k}{2}$ gadgets plus the $2k$ paths all attached to a single vertex of degree $\binom{k}{2} + 2k$. The total number of vertices is $O(n^2k^2)$ and a solution of size $\binom{k}{2}n(n+4) + k$ can be achieved iff the original graph has a packing of size k.

Modifying this construction to bi-directed trees is again strightforward, since a direction was implicit in our description. Again the only major difference is that we change edges with capacities 4 and 2 to capacities 2 and 1 respectively. For the last remark of the theorem, notice that the only vertex of high degree is C. All other vertices have degree at most 3, except the b_0 vertices of the gadgets, but even this can easily be fixed since it is not necessary for the reduction to attach p'_0 and p_0 to the same vertex. We can simply subdivide the (b_0, b_1) edge and attach p'_0 there.

□

Theorem 4. $(cW, c\Delta, pt)$-MaxPC *and* $(cW, c\Delta, pt)$-MaxRPC *are W[1]-hard for both undirected and bi-directed graphs.*

As a final note in this section, note that it is known that assuming standard complexity assumptions (specifically the Exponential Time Hypothesis which states that 3-SAT cannot be solved in time $2^{o(n)}$, see [13]) it is not possible to find an independent set of size k on an n-vertex graph in time $n^{o(k)}$. The reductions in Theorems 2 and 4 are linear in the parameter, meaning that assuming the ETH we know there is no $n^{o(W)}$ algorithm for MaxPC even for binary trees and there is no $n^{o(t)}$ algorithm, even when $W = 2, \Delta = 4$. The reduction in Theorem 3 is quadratic in the parameter, meaning that no $n^{o(\sqrt{\Delta})}$ algorithm is possible (see [3]). Putting these results together tells us that no $n^{o(Wt\sqrt{\Delta})}$ algorithm is possible. Contrasting this with the algorithm of Theorem 1 we see that the only small gap left to close here is the complexity as a function of Δ.

4 Parameterizations Involving the Objective Function

In this section we investigate parameterizations of MaxPC where the number of satisfied demands is involved in the parameters. In addition to the parameters of the previous section we consider cases where either one wishes to reject a small

number T of requests or one wishes to satisfy at least a small number of requests B. Note that T cannot possibly lead to tractability results if considered as the only parameter as the case $T = 0$ is exactly the PC case which is known to be NP-hard even for simple graph topologies. Thus, T is considered as a parameter together with W and Δ, a combination known to make PC tractable, but for which MaxPC is still intractable after the results of the previous section.

More specifically, for bi-directed trees PC is known to be hard even when $\Delta = 3$ for unbounded W, or for $W = 1$ for unbounded Δ. Therefore, any parameterization involving T would have to include both Δ and W as parameters if it were to be tractable for such trees. Here we show that $(pW, p\Delta, pT)$-MaxPC is indeed fixed-parameter tractable for bi-directed and also for undirected trees.

Theorem 5. $(pW, p\Delta, pT)$-MaxPC *is FPT for both undirected and bi-directed trees.*

Thus, for the three parameters Δ, W, T the problem is now settled for bi-directed trees: if all three are part of the parameter the problem is FPT, if we drop T the problem is W[1]-hard from the results of the previous section and if we drop any of the other two the problem is NP-hard. For undirected trees it is an interesting question what happens if one drops only W from the list of parameters (the problem $(p\Delta, pT)$-MaxPC). Here we will resolve a special case of this problem by showing that (pT)-MaxPC is FPT when restricted to undirected binary trees.

Theorem 6. (pT)-MaxPC *is FPT on undirected trees of maximum degree 3.*

Proof. (Sketch) The algorithm relies on the fact that PC on undirected trees can be decomposed into PC on stars. We first apply this step and locate good (i.e. locally colorable) and bad stars, pruning away parts of the tree where everything is good. Now, a kernelization-like argument shows that the resulting tree cannot have more than $O(T)$ leaves, otherwise it will be impossible to touch all bad stars by dropping only T requests. By extension, there can be no more than $O(T)$ internal vertices of degree 3 without attached leaves. So, we are left with a graph such that if we remove all leaves most vertices have degree 2 and there is a small number ($O(T)$) of "special" other vertices (degree 3 or 1).

Now, to select the first endpoint of a request to be dropped we simply pick one of the bad leaves. The last crucial ingredient is that for the second endpoint we can either guess the other endpoint among the $O(T)$ "special" vertices, or if the other endpoint is a non-special vertex we can use a provably optimal greedy criterion of picking the endpoint that is furthest away.

We remark that this last step is the only part of the algorithm that crucially relies on $\Delta = 3$, as all previous arguments work generally when Δ is a parameter. □

Let us now move on to consider the "natural" parameterization of MaxPC, that is, the case where the parameter is simply the number of demands B one seeks to satisfy. In this case the hardness results for PC are of course irrelevant.

Here we solve the problem for any topology where the naturally parameterized version of MaxEDP is FPT using a randomized color-coding technique.

Theorem 7. *In any graph topology where* (pB)-MaxEDP *is FPT,* (pB)-MaxPC *is also FPT.*

Corollary 1. (pB)-MaxPC *is FPT on undirected trees and rings.*

For bi-directed trees it is known that MaxEDP is NP-hard ([7]), so we cannot immediately apply Theorem 7. Here we will prove that its naturally parameterized version is FPT, a result which may be of independent interest.

Theorem 8. (pB)-MaxEDP *is FPT on bi-directed trees.*

Corollary 2. (pB)-MaxPC *is FPT on bi-directed trees.*

5 Conclusions and Open Problems

A short way to summarize the results of this paper is the following: it was known that having small (or moderate) Δ and W can help solve PC on trees. We showed that in general this cannot help us much to solve MaxPC, but it does still help if we only want to reject a small (or moderate) number of requests. This short summary captures to a large extent our results for bi-directed trees, while for the undirected case, where it is known that small Δ alone suffices to make PC tractable, we have left the complexity of $(p\Delta, pT)$-MaxPC as an interesting open problem, though settling the special case of $\Delta = 3$ (recall that even this is known to be intractable for the bi-directed case).

Much else could be done in the general direction of this work by experimenting with more parameters for the MaxPC problem and their combinations. In particular, all the structural parameters we considered here have to do with the network only. It would be nice to also explore parameters that have to do with the structure of the demands, for example limiting the maximum number of demands touching a vertex, or the maximum length of a demand. Also, many variations of MaxPC have been proposed in the past (e.g. multi-fiber networks, networks with limited hops where color conversion is allowed) and each is likely to have its own reasonable parameters to be exploited.

References

1. R.S. Anand, T. Erlebach, A. Hall, and S. Stefanakos. Call control with k rejections. *Journal of Computer and System Sciences*, 67(4):707–722, 2003.
2. Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.
3. Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. Linear FPT reductions and computational lower bounds. In László Babai, editor, *STOC*, pages 212–221. ACM, 2004.

4. R.G. Downey and M.R. Fellows. *Parameterized complexity*. Springer New York, 1999.
5. T. Erlebach and K. Jansen. Maximizing the Number of Connections in Optical Tree Networks. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *ISAAC*, volume 1533 of *Lecture Notes in Computer Science*, pages 179–188. Springer, 1998.
6. T. Erlebach and K. Jansen. The complexity of path coloring and call scheduling. *Theoretical Computer Science*, 255(1-2):33–50, 2001.
7. T. Erlebach and K. Jansen. The maximum edge-disjoint paths problem in bidirected trees. *SIAM Journal on Discrete Mathematics*, 14(3):326–355, 2001.
8. T. Erlebach, K. Jansen, C. Kaklamanis, M. Mihail, and P. Persiano. Optimal wavelength routing on directed fiber trees. *Theoretical Computer Science*, 221(1-2):119–137, 1999.
9. J. Flum and M. Grohe. *Parameterized complexity theory*. Springer-Verlag New York Inc, 2006.
10. M.R. Garey, D.S. Johnson, GL Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 1:216, 1980.
11. N. Garg, V.V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.
12. Martin Charles Golumbic and Robert E. Jamison. Edge and vertex intersection of paths in a tree. *Discrete Mathematics*, 55(2):151–159, 1985.
13. Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.
14. MC Kong and Y. Zhao. On computing maximum k-independent sets. *Congressus Numerantium*, pages 47–60, 1993.
15. S.R. Kumar, R. Panigrahy, A. Russell, and R. Sundaram. A note on optical routing on trees. *Information Processing Letters*, 62(6):295–300, 1997.
16. Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. In Mark de Berg and Ulrich Meyer, editors, *ESA (1)*, volume 6346 of *Lecture Notes in Computer Science*, pages 549–560. Springer, 2010.
17. R. Niedermeier. *Invitation to fixed-parameter algorithms*. Oxford University Press, USA, 2006.
18. P.J. Wan and L. Liu. Maximal throughput in wavelength-routed optical networks. *Multichannel Optical Networks: Theory and Practice*, 46:15–26, 1998.

A Omitted proofs

A.1 Proof of Theorem 1

Proof. (Sketch) The algorithm is based on bottom-up dynamic programming on the tree decomposition of G. Recall that the vertices of a (non-leaf) bag of the decomposition form a separator of G. Root the tree decomposition on some arbitrary bag. The key observation now is that in any feasible solution, for any given bag B, we can only have at most $O(tW\Delta)$ satisfied demands touching the vertices of B, because no edge can have more than W satisfied demands going through it, no vertex has more than Δ edges touching it (or 2Δ for bi-directed graphs) and all bags have at most $t+1$ vertices. This means that we can enumerate all possible sets of satisfied demands touching a bag in polynomial time (about $|D|^{O(tW\Delta)}$).

We now follow the standard treewidth techniques of calculating bottom-up for each possible local solution in a bag what is the maximum number of satisfied demands we can get for the graph induced by the vertices in the bag and those below it in the tree decomposition.

□

A.2 Proof of Lemma 1

Proof. For each edge (or arc) (u, v) whose capacity is $c < W$ we add $W - c$ demands from u to v. Let A be the total number of new demands added this way. After doing this we set the capacity of each edge to W and we now have an instance of MaxPC. It is not hard to see that the original CapMaxPC instance has a coloring satisfying B demands iff the new MaxPC instance has a coloring satisfying $A + B$ demands, because there must exist some optimal solution to the new instance which uses all the newly added demands.

□

A.3 Proof of Theorem 4

Proof. The proof is a modification of the proof of Theorem 2. Informally, the only edges where we needed $W = k$ in that reduction, that is, the only edges which were meant to be traversed by k satisfied paths are those of the backbone, while in the branches numbered 1 through n we allowed up to only two satisfied demands on each edge. So the question is how to fix the backbone and first and last branch to use a constant number of colors also, using a construction of treewidth k.

To do this we replace the backbone with a $(n + 2) \times k$ grid, with its vertices numbered $b_{i,j}, 0 \leq i \leq n + 1, 1 \leq j \leq k$. The edges $(b_{i,j}, b_{i,j+1})$ have capacity 2, while the edges $(b_{i,j}, b_{i+1,j})$ have capacity 1. The branches $p_i, 1 \leq i \leq n$ are identical as in the reduction of Theorem 2 and are connected to the vertices $b_{i,k}$. For the branch p_0 we simply make k copies of it (including the demands) and connect each to a vertex $b_{0,j}, 1 \leq j \leq k$. Similarly, for the branch p_{n+1} we make

k copies and connect them to $b_{n+1,j}, 1 \leq j \leq k$. We set the capacities of the edges inside these copied branches to 1.

Now the construction is complete and even though there are many ways to route the demands, it does not make a difference if we allow every possible routing (i.e. solve MaxRPC) or specify that demands starting at the j-th copy of the branch p_0 must stay on the same level of the grid (for MaxPC). In both cases the arguments of Theorem 2 go through, since it is not hard to establish that for any i the total number of satisfied requests going through the edges $(b_{i,j}, b_{i+1,j})$ is the same as the total number of satisfied requests going through the edges $(b_{i,j}, b_{i-1,j})$ (the quantity L in Theorem 2). The modification for bi-directed graphs is also straightforward.

Finally, observe that this graph has $W = 2, \Delta = 4$ while the treewidth (in fact, the pathwidth) of the graph can be upperbounded by $k + 2$.

□

A.4 Proof of Theorem 5

Proof. (Sketch) Recall that in [6] it was shown that PC is FPT on bi-directed trees when parameterized by the maximum number of demands touching any vertex. The algorithm uses a standard treewidth-like approach of enumerating for each vertex all possible local solutions. From this result it follows (as also observed in [6]) that $(pW, p\Delta)$-PC is FPT on bi-directed trees, since in any YES instance there can be only at most $2\Delta W$ properly colored requests touching a given vertex.

Here, we extend these ideas to the MaxPC problem with few rejections. The first observation is that in a YES instance no vertex can be touched by more than $2\Delta W + T + 1$ demands. In such a case any set of rejected requests would still leave this vertex with too many demands to be colored properly, so if there is a vertex touched by so many demands we can immediately reject.

Otherwise, we have established that all vertices are touched by at most $O(\Delta W + T)$ demands. We now again use a bottom-up dynamic programming technique, since for every vertex it is possible to enumerate all sets of satisfied demands and their colorings in FPT time.

□

A.5 Proof of Theorem 6

Proof. We will show a kernelization-like argument, relying on an edge-splitting trick. As has been observed in previous work ([6]) PC on undirected trees can easily be reduced to PC on stars as follows: take any edge connecting two non-leaf vertices of the tree (u, v). Remove that edge and add two new vertices u', v' and edges $(u, u'), (v, v')$. For every demand from a vertex w to a vertex x that was using the edge (u, v) add a demand from w to u' and a demand from x to v' (we assume that w is the vertex of the demand that lies on u's side of the cut). Now we have two new independent PC instances. If either of them is a NO

instance, then the original instance is a NO instance. If both of them are YES instances, then the original instance is also a YES instance, since we can combine the solutions by reintroducing the edge (u, v) and an appropriate permutation of colors in the solution of one sub-instance. Repeating this process will lead us to the base case of a star, which is the simplest tree where this trick cannot be applied. We call this trick where an edge is split to produce two instances edge slicing.

The fixed-parameter tractability of $(p\Delta)$-PC follows immediately from the above trick and the observation that PC and MaxPC are FPT when parameterized by the size of the graph (see [6] or the results of [16] on graph coloring for graphs of bounded neighborhood diversity). In a bounded degree graph the stars one will find as base cases of the above algorithm have a bounded number of leaves, so it is possible to brute force the problem efficiently.

Let us now extend the edge slicing trick to the MaxPC case. The first step of our algorithm is to slice all edges connecting non-leaf nodes as described previously, producing a number of instances on stars, one for each internal vertex. For each of these star instances we attempt to solve PC with the given number of colors using the known FPT algorithm. Of course, if the answer is YES to all then all the demands in the original instance can be satisfied and we are done. So some star instances must be NO instances.

Root the tree on some arbitrary vertex. We will say that an internal vertex u of the tree is good if the corresponding star instance where u is the center of the star is a YES instance and also the same holds for all of u's descendants in the tree. We will say that it is bad if the corresponding star instance is a NO instance. Observe that if u is a good vertex and v is its parent in the tree then slicing the edge (u, v) and keeping only the instance that contains v will not alter the answer to our MaxPC instance. In particular, it is possible to satisfy all but T requests in the original if and only if it is possible to do so in the instance where u's subtree is pruned. The reason is that any solution that drops at most T requests in the pruned instance can be extended to cover u's subtree since all the demands there are W-colorable (i.e. we know that even if we drop none of the demands that use the edge (u, v) we can still satisfy all demands from u and below).

Proceeding in this way, we are left with a graph where every internal vertex is not good. For the sake of analysis, remove all the leaves from the tree and call the resulting graph G'. Now, we are left with a tree where every leaf used to be a bad internal vertex. In a solution to the problem we must pick a set of rejected paths that touches all bad vertices, because we know that locally it is impossible to color all the paths touching a bad vertex. In particular all the leaves of G' must be touched by some rejected path. But no rejected path could be touching more than two leaves of G', meaning that if we have a YES instance then G' has at most $2T$ leaves, thus at most $2T - 1$ vertices of degree 3 (the average degree of a tree is less than two).

Let v be a vertex of degree two in G' and let e_1, e_2 be its two incident edges. If the star centered on v in G can become W-colorable by removing only requests

that use both e_1 and e_2, then we say that v is easy. (In other words, v is easy if it can be locally fixed without dropping any requests that terminated on v or its attached leaf). If a vertex of degree two in G' is not easy we say it is hard. Observe that since at least one dropped request must terminate at each leaf of G' and at each hard degree two vertex, we now know that the total of the number of leaves and hard vertices is at most $2T$ (otherwise we immediately reject).

We will say that a topo-edge of G' is an inclusion-maximal connected set of easy vertices of degree two in G'. Because G' has at most $4T$ hard vertices and vertices with degree other than 2 it must have at most $4T$ topo-edges. Pick a leaf u of G'. We know that if a solution exists, it must reject some request touching u. So, we must drop a request whose first endpoint is either u or one of its (at most two) attached leaves in G. Now we must guess where the other endpoint is.

First, we need to find the vertex v of G' that will be closest to the other endpoint of the dropped request (so the second endpoint of the dropped request will be either v or one of its, at most 2, attached leaves in G). v can either be one of the at most $4T$ hard vertices or vertices that have degree other than two in G', or it can belong in one of the topo-edges of G'. The main claim now is that if v belongs in a topo-edge it suffices to check only the unique vertex of the topo-edge that is furthest away from u. Assuming this, it should be clear that there are at most $O(T)$ choices for a request to be dropped having one endpoint close to u. For each choice we delete a request, decrease T by one and repeat, leading to a $T^{O(T)}$ algorithm.

The only thing left is to prove the claim that it is optimal to select the vertex of a topo-edge that is the furthest away from u. So, suppose that v, v' belong in the same topo-edge, v' is further from u than v and an optimal solution rejects a request from u (or one of its attached leaves) to v (or its attached leaf) and satisfies a request from u to v', giving it say color c_1. If the dropped request terminates at v it uses a subset of the edges of the satisfied request so we can exchange them. If it terminates at v's attached leaf in G (call it w) then we may have a problem trying to color it with c_1 and rejecting the longer request only if c_1 is used on a request terminating at w. However, because v' is further from u the only way that c_1 could be used at this edge is for a request whose other endpoint is v. This length-one request can always be recolored if there is another free color on its edge. This must be the case because v is easy, so the total number of requests originally using this edge must be $\leq W$ and in addition we are looking at a solution which has rejected a request using this edge.

□

A.6 Proof of Theorem 7

Proof. First, observe that we can safely assume that $W < B$, otherwise we can just select any set of B demands and give them different colors. Now randomly color all the demands using W colors, that is, assign each demand one of the colors independently and with equal probability.

This separates the set of demands D into W disjoint sets. For each of these sets of demands we solve the MaxEDP problem. The union of the solutions of the W instances of MaxEDP gives us a solution to MaxPC, by using a different color for each.

Now, let us analyze the running time and probability of success of this algorithm, that is, the probability that if the instance does have a solution of size at least B the algorithm will return such a set. The first step is a randomized coloring procedure. Fix a solution of size B, assuming that it exists. There exist $W!$ proper colorings of these B paths and the probability that our algorithm assigns colors to these B paths according to a specific one of these proper colorings is at least $\frac{1}{W^B}$, so in total the probability of properly coloring the B paths of the fixed solution is at least $\frac{W!}{W^B}$.

If the randomized coloring step was successful then the algorithm solves MaxEDP on the produced instances. By definition, if MaxEDP is solved correctly the solution produced will satisfy in each color class at least as many demands as the fixed solution does, meaning that its size is at least B.

For the running time, if the FPT algorithm for (pB)-MaxEDP takes time $O(f(B)n^c)$, then the second phase takes $O(Wf(B)n^c)$. Repeating the second phase $O(\frac{W^B}{W!})$ times gives a constant probability of success. Taking into account that $W < B$ the running time is clearly FPT.

□

A.7 Proof of Corollary 1

Proof. For trees MaxEDP is solvable in polynomial time by [11]. For rings it can again be solved in polynomial time easily by solving at most $|D|$ instances of the same problem on a path: one simply tries out all of the demands crossing an edge one after the other (or rejecting all of them).

□

A.8 Proof of Theorem 8

Proof. Consider an edge (u, v) in the tree, such that u and v are both non-leaf vertices. Removing this edge would result in the creation of two trees. In case that for one of these trees there exist no demands with both endpoints inside it, we say that the cut (u, v) is unbalanced.

We will now apply a set of reduction rules. The main rule is the following: suppose there exists an unbalanced cut in the tree, that is, there exists an edge (u, v) such that no demands lie wholly within v's side of the edge (u, v). Then, for every demand with one endpoint on u's side and one on v's side, we change its endpoint on v's side to be v. Then we delete all the vertices on v's side except v (which is now a leaf).

It is clear that this rule can be applied in polynomial time, so let us argue about its correctness. If the original instance has a solution of size B, then the pruned instance also has the same solution (we did not remove any demands and

did not make any previously non-overlapping demands overlap). If the pruned instance has a solution of size B then we want to show that this is also a valid solution to the original instance. This is not hard to see because the pruned instance's solution can only include at most two satisfied demands using the edge (u, v) in opposite directions. Because these demands have opposite directions they could not overlap in the part of the original graph that was pruned, or that would create a cycle. Since the rest of the instance is unchanged we are done.

The other reduction rules we will apply are simpler: first, if there is an arc without demands crossing it, cut it (notice that we allow the algorithm to cut an arc (u, v) even if the arc (v, u) is to be kept). If there is a pair of anti-parallel arcs between two vertices u, v such that at most one demand is using each arc, contract them, that is merge the vertices u, v into one vertex. If there are two demands d_1, d_2 and d_2 is using a superset of the edges of d_1, delete d_2. Finally, if a demand is overlapping no other demands remove it and all the arcs it's using from the graph and set $B := B - 1$. The correctness of these rules should be easy to see and of course they can be applied in polynomial time.

Suppose now that we apply these rules to an instance until it is no longer possible. Let G be the resulting tree and let G' be the tree we would get from G if we deleted all the leaves (we will call G' the frame of G). We will say that a demand in G is local if it has length 2 and both its endpoints are leaves. We will say that a vertex of G' is special if it has degree at least 3 (in G') or if there is a local demand in G touching it. Observe that all leaves of G' are special, since the edge connecting each leaf of G' to its parent is an edge connecting two internal vertices in G, so cutting it can not give an unbalanced cut. If there are at least B distinct vertices in the frame which are touched by local demands then we can clearly find a solution of size B and we are done. Assume then that this is not the case, so we can conclude that the frame G' has at most B leaves and therefore at most $2B$ special vertices (the average degree of a tree is at most 2). We will use a branching procedure whose depth will be bounded by the number of special vertices of the frame.

First, let's take an easy case. Consider a leaf of G', u and suppose its degree in G is at most B. Then, there are at most B^2 local demands touching u. Enumerate all 2^{B^2} subsets of them. For each subset, if it is feasible, include it in the solution decreasing B accordingly and remove all other local requests and all requests overlapping the ones selected. Since the optimal solution must agree with one of these subsets, one of these choices will certainly lead to the optimal. The important observation here is that now there are no more local demands touching u, so the graph can be simplified because the cut between u and its parent is unbalanced. Since u was special, this decreases the number of special vertices by one and notice that the reduction rules cannot possibly create new special vertices. In other words, the branching process is making progress even if no demands are selected. Since enumerating all local solutions takes time 2^{B^2} and the process will have depth at most $2B$ an easy upperbound on the total time now is $(2^{B^2})^{2B} = 2^{2B^3}$.

In general though we cannot assume that some leaf of G' will necessarily have a small number of leaves attached to it. Consider a leaf of the frame u and the leaves attached to it in G, $w_1, w_2, \ldots, w_d$. It is possible to compute a maximum-size feasible set of demands among the local demands with endpoints in the leaves attached to u using a maximum matching computation on a bipartite graph. One part of the graph contains a vertex for each arc (w_i, u) and the other part has a vertex for every arc (u, w_i). There is an edge between (w_i, u) and (u, w_j) iff there exists a demand from w_i to w_j. It should be easy to see that any matching on this bipartite graph corresponds to a set of non-overlapping local demands in G. Recall that we have applied reduction rules which delete single arcs, so it could be the case that the two parts of the bipartite graph do not have the same size. We will now use the bipartite graph for our analysis but it should be straightforward to translate between the matchings terminology to non-overlapping paths.

First, if the bipartite graph has a matching of size B or more we are done. So assume that the maximum matching has size at most $B-1$. We may assume that the bipartite graph has no isolated vertices. If some vertex, say (u, w_i) is isolated, then its arc has no local demands using it. The global demands using it are already overlapping in the arc that connects u to its parent. Therefore, moving the endpoint of these demands from w_i to u does not affect the instance and allows us to delete one arc.

Let L, R be the two parts of the bipartite graph with L containing the arcs directed towards u, and first consider the case where one of them, say L has small size, i.e. $|L| < B$. Suppose some vertex $(w_i, u) \in L$ has degree at most $B-1$ in the bipartite graph. There are B choices for how to use (w_i, u): either we will put one of its incident edges in the matching, in which case we delete it and the other endpoint of that edge, or leave it unmatched, in which case we can just delete it. Either way, by branching on B choices we have reduced the size of L, so if we could continue this way we would eliminate all of L in about $|L|^B < B^B$ time. The interesting case is therefore if all vertices of L have degree at least B.

In that case, it should be clear not only that the graph has a matching of size $|L|$, but also that this holds true even if we delete any of the vertices of R, since then all vertices of L would have degree at least $B-1$. Suppose that $(u, w_j) \in R$ and there is some demand coming from outside the star to w_j in G. We claim that it is possible to shorten this demand so that it ends at u without affecting the answer. To see this, consider an optimal solution of the new instance and suppose that extending the demand back to w_j is infeasible because the arc (u, w_j) is in use. It must be in use by some local demand, corresponding to an edge in the matching. But as we said, we can always find another matching of the same size without (u, w_j), thus freeing it for use by the external demand. Using this argument repeatedly makes sure that there are no external demands going into any leaves w_j (in the symmetric case where we start by assuming $|R| < B$ we would have no demands leaving a leaf to outside). Observe now that we are in a situation where we can satisfy $|L|$ local demands. Any solution

which satisfies an external demand touching a leaf must be satisfying a demand using an arc (w_i, u). This means that we can drop the external demand and include a local demand in its place. So without loss of generality we drop all external demands using some arc (w_i, u) and satisfy L local demands using the maximum matching. Now there are no local demands touching u so the graph can be reduced.

What is left now is to argue how to handle the general case where both $|L|$ and $|R|$ may be much larger than B. Compute a maximum matching of the graph and let $L_1 \subseteq L$ and $R_1 \subseteq R$ be the matched vertices (clearly, $|L_1|, |R_1| < B$). Let $L_2 = L \setminus L_1$, $R_2 = R \setminus R_1$. The graph induced by $L_1 \cup R_1$ can have at most B^2 edges. Enumerate all subsets of these and for each subsets S, include S in the solution if it is feasible and delete edges outside S (in G select the demands that correspond to S and delete the others, remove arcs and reduce B as necessary). After doing this for each subset of the edges connecting L_1 to R_1 we are left with a bipartite graph where no edge connect L_1 to R_1 but also no edges connect L_2 to R_2, since the existence of such an edge could have augmented our original matching. Therefore, the graph is disconnected into the components $L_1 \cup R_2, R_1 \cup L_2$. Observe that in each of these we have one part with size $< B$. We now apply the argument of this previously analyzed case to simplify the instance.

It should be clear from the above that from an instance where the frame has $P \leq 2B$ special vertices we can obtain by branching at most $f(B)$ instances where the frame has $P - 1$ or fewer special vertices, for some $f(B)$ which is at most in the order of $2^{O(B^3 \log B)}$. The $2^{O(B^2)}$ comes from brute forcing all the edges between matched vertices in the case where both parts of the bipartite graph are large. The $B^B = 2^{O(B \log B)}$ comes from handling cases where one part is small and some of its vertices have small degree. Since this branching will be repeated $2B$ times, the total running time is a huge $2^{O(B^4 \log B)}$.

□