





Two Exact Algorithms for the Packet Scheduling Problem

Fei Li¹(✉)  and Ningshi Yao² 

¹ Department of Computer Science, George Mason University,
Fairfax, VA 22030, USA
fli4@gmu.edu

² Electrical and Computer Engineering Department, George Mason University,
Fairfax, VA 22030, USA
nyao4@gmu.edu

Abstract. We consider a classic packet scheduling problem [7] and its variants. This packet scheduling problem has applications in the areas of logistics, road traffic, and more. There is a network and a set of unit-length packets are to be transmitted over the network from their respective sources to their respective destinations. Each packet is associated with a directed path on which it must travel along. Time is discrete. Initially, all the packets stay on the first edges of their respective paths. Packets are pending on the edges at any time. In each time step, a packet can move along its path by one edge, given that edge having no other packets move onto it in the same time step. The objective is to minimize *makespan* – the earliest time by which all the packets arrive at their respective destination edges. This problem was proved NP-hard [1] and it has been studied extensively in the past three decades. In this paper, we first provide a semi-online algorithm GRD and show that GRD is optimal for scheduling packets on arborescence and/or anti-arborescence forests. We then provide a parameterized algorithm PDP which finds an optimal makespan for the general case. PDP is a dynamic programming algorithm and its running time complexity depends on the *congestion* and *dilation* in the input instance. The algorithm PDP's idea is new and it is derived from an insightful lower bound construction for the general packet scheduling problem.

Keywords: Packet scheduling · exact algorithms · dynamic programming

1 Introduction

A packet scheduling problem [7] has been studied extensively in the past three decades. Consider a directed graph $G = (V, E)$ with a set of vertices V and a set of edges E , where $|V| = n$ and $|E| = m$. There are N packets that are

N. Yao's research is partially supported by NSF grant ECCS-2218517.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
W. Wu and J. Guo (Eds.): COCOA 2023, LNCS 14461, pp. 141–153, 2024.
https://doi.org/10.1007/978-3-031-49611-0_10

to be transmitted on G . Each packet $p \in \{1, 2, \dots, N\}$ has a *source* $s_p \in V$, a *destination* $t_p \in V$, and a directed *path* P_p with a *length* l_p denoting the path P_p 's number of edges $|P_p|$. A path P_p can be represented by an ordered set of edges $e_p(1), e_p(2), \dots, e_p(l_p)$ and each $e_p(i) \in E$ is an edge on P_p , $\forall i = 1, 2, \dots, l_p$. All the packet paths are simple ones and thus we have $l_p = |P_p| \leq |E| = m$, $\forall p$. The path P_p specifies the edges as well as the order that a packet p should travel along on G .

Time is discrete. Assume that any edge $e \in E$ represents a sufficiently-large-size buffer so that at any time, any number of packets can stay on the edge e . The vertices V act as switches so that in each time step, each edge e accepts at most one packet to be forwarded to the queue represented by e : In each time slot, for any outgoing edge $e = (v, w)$ incident to a vertex v , at most one packet staying on v 's incoming edges (u, v) and having e as their next step's edges can be moved onto e . Multiple packets can be forwarded simultaneously to their next edges in one single time step as long as these packets are not getting into the same queue after being forwarded. For ease of notation, we assume that all the packets p stay on their respective paths P_p 's first edges $e_p(1)$ at the end of time step 1. A packet p 's *duration* d_p is the time slot by which p arrives at the last edge $e_p(l_p)$ of its path P_p . Clearly, we have $d_p \geq l_p$, $\forall p$. We design a scheduling algorithm with the objective of minimizing *makespan*, defined as the maximum duration $\max_p d_p$ for all the packets $p \in \{1, 2, \dots, N\}$.

2 Related Work

This packet scheduling problem was proved NP-hard [1]. The authors in [13] showed that an optimal makespan cannot be approximated down to the ratio 1.2 unless $P = NP$, even for the case in which the graph is a tree (with bidirectional edges). Define the *dilation* D as the maximum path length, $D = \max_p l_p$, and the *congestion* C as the maximum number of paths having a single edge in common, $C = \max_{e \in E} |C(e)| = \max_{e \in E} |\{p | e \in P_p\}|$, where $C(e)$ is the set of packets having e in their paths. It is clear that any schedule's makespan has a lower bound of $\max(C, D) \geq \lceil \frac{C+D}{2} \rceil = \Omega(C + D)$ [7].

A class of scheduling algorithms are called *greedy*, if an algorithm in such a class never leaves an outgoing edge $e = (u, v)$ *idle* (an idle edge does not accept packets) as long as there are packets waiting in the incoming edges of v having e as their next-step edges in the corresponding paths [9]. Any simple randomized algorithm with the greedy strategy achieves a makespan $O(C \cdot D)$. In [8], the authors gave a schedule of length $O(C + D)$ in time $O(L(\log \log L) \log L)$ with a probability at least $1 - L^\beta$, where $\beta < 0$ is a constant and $L = \sum_p l_p$. This randomized algorithm can be derandomized using the method of conditional probabilities [11] and it became the first constant approximation algorithm, against the lower bound $\lceil \frac{C+D}{2} \rceil$ of makespan. In [16], the author gave a simpler proof of the algorithm in [8]. The algorithm is an *offline algorithm* which is given its input including a complete description of the graph G and the packets' paths P_p , $\forall p$ in designing a schedule. For the algorithm in [8], the hidden constant in

$O(C + D)$ is high. The best known approximation ratio is 24 [14], against $C + D$ as well.

Some variants of this packet scheduling problem have been studied as well. In an *online setting*, an algorithm only uses the information that is available locally to a vertex v in order to determine which packet to be forwarded to the edge (v, w) , among the ones waiting on v 's incoming edges. For the class of *layered networks*, [6] gave a simple online randomized algorithm with a makespan $O(C + B + \log N)$, where $B (\leq D)$ is the number of layers of this network. For the case in which all the packets' paths P_p are assumed to be the shortest ones from the sources s_p to the destinations d_p (in terms of the number of edges), there was an online algorithm with a maximum duration bounded by $D + N - 1$ [10]. In [12], the authors improved the result to be $O(C + D + \log(N \cdot D))$, with a high probability, and in [18], the authors gave a simple online randomized algorithm with a duration $O(C + D + \log N)$, with a high probability. The most recent known work on online algorithms is [12], giving a universal deterministic $O(C + D + \log^{1+\epsilon} N)$ algorithm. This result is almost optimal. The problem whether there exists an online algorithm with competitive ratio bounded by $O(C + D)$ is still open. The algorithm in [10] is a greedy online algorithm. Some other variants in which the edges are bufferless (i.e., at most one packet is on an edge at any time) or the packets are allowed to wait (i.e., staying on each of such edges for more than 1 time slots) only on some predefined edges were discussed in [17]. Another line of research is to consider *packet scheduling* and *packet routing* (packet routing algorithms allow packets to choose paths to get to the destinations) together in order to minimize the makespan. The *competitive packet scheduling* problem is also studied. In this problem, the packets select their paths rationally and the makespan is the social welfare to be optimized [3]. The paper [9] gave a brief survey. More recent related work can be found from the work following [9].

Our Contributions. In this paper, we study exact algorithms for the packet scheduling problem. We design two algorithms. One is named GRD. GRD is a simple, fast semi-online algorithm and it optimizes the makespan in scheduling packets on arborescence and/or anti-arborescence forests. The other one is an exact algorithm, named PDP, for the general packet scheduling case and its running-time complexity depends on the parameters (congestion and dilation) of the input instances. In Sect. 3 and Sect. 4, we describe the algorithms GRD and PDP, along with their running-time analysis and performance analysis, respectively.

3 GRD: Scheduling Packets on Arborescence and Anti-Arborescence Forests

In the *semi-online setting*, an algorithm has no complete knowledge of the graph G and a packet has no information regarding to the other packets' status at any time. A semi-online algorithm may allow packets carry some information

on themselves — normally, such information is a constant value that cannot embed the whole graph information nor any information on the other packets. The values carried by the packets waiting on the incoming edges of a vertex can be used to make the decision of transmitting them.

In this section, we design a *semi-online* algorithms and name it GRD (standing for greedy). Each packet p has its path information P_p .

3.1 The Ideas

GRD is based on the following two greedy ideas: Consider a packet p at the beginning of a time step t .

1. The packet p greedily moves onto the next edge e as long as no other packets are competing for e in the same time step. Such a best-effort movement of p will not increase p 's duration and will not increase any delays to other packets.
2. Consider the case when there are more than one packets including p competing for an edge e . If p is not chosen to move onto e , then p 's duration is increased by 1. Therefore, in this time step t , the idea is to forward the packet whose duration's increase affects the algorithm's makespan the most. Recall that a semi-online algorithm has no global information on the graph G nor the information of the packets not-competing for e in the time step t , thus, a packet p 's duration is *estimated* as the sum of the length of its remaining path $(e_p(i+1), e_p(i+2), \dots, e_p(l_p))$ and the current time t , assuming $e = e_p(i+1)$. The packet with the largest number of time steps to reach its destination under the assumption of no future delays, among those pending packets for the edge e , is moved onto the edge e .

3.2 The Algorithm

We use c_p to denote the number of *remaining edges that the packet p should take in order to reach its destination*, assuming there are no delays along p 's remaining path. At the beginning of a time step t , p 's duration is estimated as $t + c_p$. When $t = 1$, c_p is initialized as l_p , and p should take the path P_p with l_p edges to its destination. In the algorithm GRD, the value c_p is updated by the packet p using a counter. At a time, given an edge e , GRD uses the value c_p to select the largest-value packet p to send to e . Since the value c_p may be updated over time, we use a function $c_p(t)$ to denote the value c_p at the end of time step t . The algorithm GRD is described in Algorithm 1.

Note that for each edge $e = (u, v)$, the decision of accepting a packet p or not by the edge e depends on the local packets' $c_p(t-1)$ values, hence GRD is a semi-online algorithm.

3.3 The Analysis

In the following, we analyze GRD. We first state two assumptions with which we do not lose generality. These two assumptions facilitate the analysis of GRD

Algorithm 1. GRD

- 1: For each packet p , associate p with a value $c_p(t - 1)$ to denote its remaining time slots needed to get p to its destination t_p , starting from time t and assuming no delay incurred in the future for forwarding p . Initially, $c_p(0) = l_p$.
 - 2: Forward a packet p to an edge e as long as no other packets are waiting for being forwarded to e or $c_p(t - 1)$ is the largest value for all such packets competing for the edge e . Ties are broken arbitrarily.
 - 3: Update $c_p(t) \leftarrow c_p(t - 1) - 1$, for each time of forwarding a packet p . Update $c_q(t) \leftarrow c_q(t - 1)$, for each time of not forwarding a packet q .
-

as well as the analysis of PDP which is introduced in Sect. 4. We then show GRD’s running time analysis and prove that it is optimal for scheduling packets on *anti-arborescence forests*.

Assumption 1. *Any edge in the graph G must belong to a packet’s path, say $\forall e \in E$, we have $e \in \bigcup_p P_p$.*

Assumption 1 holds since for any algorithm, it does not schedule a packet over an edge outside of the set of edges $\bigcup_p P_p$ and thus, such removals of edges do not hurt the algorithm in generating the makespan. A useful fact is that Assumption 1 implies $m \leq N$.

In the packet scheduling problem’s statement, we assume that all the paths are simple ones. Given an input instance with some packet paths having cycles, we can always convert the input instance to be one with simple paths only. Such conversion does not introduce a larger makespan.

Assumption 2. *All packets’ paths are simple ones.*

Consider a packet p and its path P_p that have cycles. We modify the path P_p and the graph G so that the modified path P_p has no cycles. Such cycles, if any, are removed one by one from the input instance as below. Let

$$P_p = \{s_p, v_1, v_2, \dots, v_{k-1}, v_k, v_{k+1}, \dots, v_w, v_k, v_{w+1}, \dots, t_p\}$$

and there is one simple cycle $v_k, v_{k+1}, \dots, v_w, v_k$. We create a new graph: having two vertices v'_k and v''_k so that all the edges having v_k as the heads originally now have v'_k as the heads. All the edges having v_k as the tails originally now have v''_k as the tails. We create a subpath $v'_k, v_{k+1}, \dots, v_w, v''_k$ to replace the subpath $v_k, v_{k+1}, \dots, v_w, v_k$. The vertex v_k is removed from the new graph and the new path is:

$$P_p = \{s_p, v_1, v_2, \dots, v_{k-1}, v'_k, v_{k+1}, \dots, v_w, v''_k, v_{w+1}, \dots, t_p\}$$

Recall here that though in the new graph we have two new edges (v_{k-1}, v'_k) and (v_w, v''_k) , these two new edges belong to the packet p ’s path only but not to any others. These two edges replaces the edges (v_{k-1}, v_k) and (v_w, v_k) . Having these two edges does not increase p ’s duration, nor any other packet’s duration. Any algorithm on the original graph G has the same makespan on the new graph.

Theorem 1. *GRD has a running time of $O(m \cdot D \log N)$.*

Proof. We are using a charging scheme to calculate GRD's running time complexity and will show that it is $O(\max(m, N)D \log N)$. With Assumption 1, we will have Theorem 1.

First, we show GRD's running time is $O(N \cdot D \log N)$. For each edge e in the graph G , we use a priority queue to maintain all the packets staying on the edge e at a time and the value c_p is used as the key. We charge GRD's running time on the priority queue operations on the packets during GRD's execution. For each edge e , it takes time $O(\log N)$ to get the packet with the largest c_p value. For each packet p , it incurs at most l_p times of getting into a new packet queue. Note that a packet queue is associated with each edge of the path P_p . For each such a packet transmission, it incurs queue-operation time $O(\log N)$. There are N packets. Thus, the total running time is $O(N \log N \max_p l_p) = O(N \log N \cdot D)$, where D is the dilation.

Second, we show that the running time can be calculated as $O(m \cdot D \log N)$ by charging the cost to each packet at a time. Label the edges as e_1, e_2, \dots, e_m . Note that among the packets $S(e_i)$ waiting to be sent to an edge e_i , only the max- c_p -value packet p experiences $\log |S(e_i)|$ time while the other packets in $S(e_i) \setminus \{p\}$ experiences search time 0. Consider a time step t and let $S(e_1), S(e_2), \dots, S(e_m)$ denote the m priority queues containing the N packets, with some queues being possibly empty. For this single time step t , the total search time incurred to those packets being sent is $\sum_i \log |S(e_i)|$ and the total search time incurred for those packets not being sent is 0. Note $S(e_i) \cap S(e_j) = \emptyset, \forall i \neq j$. We have the total search cost for a packet moving one step along its path (assuming $m \geq 2$):

$$\begin{aligned} \sum_i \log |S(e_i)| &= \log \prod_i |S(e_i)| \\ &\leq \log \left(\frac{\sum_i |S(e_i)|}{m} \right)^m \\ &= m \log \frac{N}{m} \leq m \log N - m \end{aligned} \tag{1}$$

Inequality 1 is based on Edwin Beckenbach and Richard Bellman's work presented in [2]. Recall that we only need to count the search time for a packet being sent in a time step, thus, the number of searches associated with a packet is its length, bounded by D . The total running cost of GRD is also bounded by $O(m \log N \cdot D)$. Theorem 1 is proved.

In the following, we analyze GRD's performance when the underlying graph G is an *arborescence and anti-arborescence forest*. An arborescence and anti-arborescence forest contains multiple arborescences and anti-arborescence. An *arborescence* [4] is a directed graph having a root so that there is exactly one directed path from the root to any vertex of this graph. An *anti-arborescence* [5] is one created by reversing all the directed edges of an arborescence, i.e. making them all point to the root rather than away from it.

Theorem 2. *GRD is optimal in scheduling packets on arborescence and anti-arborescence forests.*

Proof. In order to prove Theorem 2, we only need to show that GRD is optimal for packet scheduling on one arborescence and one anti-arborescence since each packet is scheduled only on one arborescence or one anti-arborescence. In the following, we prove that GRD is optimal in scheduling packets on an anti-arborescence. The analysis for GRD on arborescence is similar but easier. We leave it in our full journal paper.

We inductively prove Theorem 2 using an exchange argument. Let ADV denote an adversary. At the beginning of time step 1, ADV is the same as an optimal algorithm with the minimum makespan d^* . Consider an anti-arborescence T with a root r and label the depth of the edges as $1, 2, \dots$ based on the distances from the directed edges' tails to the root r . The 1-depth edges are the edges having r as their heads. An observation is that, given the graph being an anti-arborescence, a packet moves from an edge labelled as i to an edge labelled as $i - 1$ if the packet is transmitted in this time step. We are going to show that there exists an invariant maintained during the algorithm's execution. The invariant guarantees Theorem 2 since at the end of the schedule, we have $d^* = d$, where d is GRD's makespan.

- (Invariant): At the beginning of any time step t , ADV and GRD have the same configuration so that each edge holds the same set of packets.

At the beginning of time step 1, the invariant holds. Now, we consider the first time step t , in which, ADV and GRD sends different packets, say, q and p respectively, to an edge e . If such a time step t does not exist, then ADV and GRD are the same and therefore, $d^* = d$.

Consider the time step t . Recall that G is an anti-arborescence, thus, the fact that e is the edge that p and q plan to step onto in time step t implies that p and q have their paths overlap from time t till one packet reaches to its destination. GRD chooses p instead of q because of $c_p(t - 1) \geq c_q(t - 1)$, which implies that the remaining path for q is embedded in the remaining path for p . The modification on ADV is as below:

1. In time step t , we modify ADV so that ADV sends p instead of q in t .
2. In the remaining schedule, ADV switches the orders of scheduling packets p and q . In each time step that ADV originally schedules q , the packet p is available (considering that p is ahead of q on q 's remaining path and q 's remaining path is embedded in p 's remaining path) and p scheduled.
3. Similarly, in each time step ADV originally schedules p , q can be scheduled until q reaches to its destination.
4. The order and time slots of sending other packets than p and q are not changed.
5. For the possible case in which at some point t' in the future, the original ADV sends q instead of p making q is again before p on their shared subpath, then the modified ADV switches back and follows the original schedule starting from time t' .

Realize that such modification on ADV at time t does not increase the duration for p since p moves ahead of where it was in its original schedule. This modification does not make q 's duration more than p 's original duration, which is no more than the makespan d^* . This modification does not change any other packet's duration as well. Thus, d^* keeps the same after we modify ADV and the invariant holds.

For each edge that ADV and GRD schedule different packets, we apply the above procedure to modify ADV. The above modifications make sure that the modified ADV is with the same configuration as GRD. At the end of time step t , ADV and GRD are with the same configuration again. Inductively, the invariant is proved. Thus Theorem 2 holds.

4 An Optimal Algorithm for the General Case

In this section, we use the dynamic programming technique to design a parameterized optimal algorithm, named PDP, for the packet scheduling problem in the general case. This algorithm's idea is different from the ones in [14], which were based on the integer linear programming technique. Our algorithm catches some properties of the makespan lower bound construction for the packet scheduling problem and we hope that such properties can be used to design better approximation algorithms.

4.1 The Ideas

We introduce some concepts that will be used to describe our ideas. Consider a packet p in a given schedule. If p moves onto an edge e at time t , then we say that the edge e is *busy* at time t . Otherwise, we say that the edge e is *idle* at time t . The maximal interval $[t, t']$ in which an edge e is continuously busy (to accept different packets) is called a *busy interval*, and thus, the edge e is idle in time step $t - 1$ and time step $t' + 1$, if any. If at the beginning of a time step t , a packet p and a packet q have e as their next edges in their respective paths, then we call p and q the *competing packets for e* in time step t . The edge e is called a *congested edge*.

Consider a packet p at the beginning of a time step t . Assume p is on the edge $e_p(i)$ where $i \neq l_p$. The lower bound of time steps needed for p to arrive at its destination t_p is $l_p - i$ — In the lower bound case, all the edges in $e_p(i+1), e_p(i+2), \dots, e_p(l_p)$ should be busy for p . Assume p has the maximum duration d^* in an optimal algorithm. Our ideas in PDP is to make sure that p experiences not many delays along its path to its destination.

The first idea is as below: the packet p greedily moves onto the next edge e as long as no other packets are competing for e in the same time step. This idea is identical to one used by GRD. Let OPT denote an optimal algorithm. OPT forwards a packet as long as it can. Based on the above observation, we have the following lemma.

Lemma 1. *Assume that at the beginning of a time step t , there are k packets competing for an edge e . Then the edge e must be continuously busy from time t to time $t + k - 1$ in *OPT*.*

Proof. As *OPT* is a greedy algorithm, it schedules a packet onto e as long as e is not busy. The edge e accepts one packet at a time, then there are at least k packets available for e to accept in the time interval $[t, t + k - 1]$.

Another observation based on the first idea is as below: Let us category all the packets $1, 2, \dots, N$ into different groups, based on the needed number of transmissions to their destinations. We use $G(t, i)$ to denote the group containing the packets p which need i transmissions to their respectively destinations, starting from the beginning of a time step t . Initially, we have at most D groups:

$$G(1, 1), G(1, 2), \dots, G(1, D),$$

where D is the dilation of the input instance. Initially, a packet p has its path length l_p and thus, it belongs to the group $G(1, l_p)$. From the best-effort manner of forwarding packets, we have the following observation.

Lemma 2. *In *OPT*, we have*

$$\begin{aligned} G(t, i) &= G^1(t + 1, i - 1) \cup G^2(t + 1, i) \\ \emptyset &= G^1(t + 1, i - 1) \cap G^2(t + 1, i) \\ G(t, i + 1) &= G^1(t + 1, i) \cup G^2(t + 1, i + 1) \\ G(t + 1, i) &= G^1(t + 1, i) \cup G^2(t + 1, i) \end{aligned}$$

where $G^1(\cdot, \cdot)$ denotes the set of packets forwarded in the time step, $G^2(\cdot, \cdot)$ denotes the set of packets not being forwarded, and any one of them can be an empty set.

Proof. Lemma 2 holds due to the facts that a packet is either forwarded or kept stay in a single time step and a packet can be moved at most one step in one time slot. Consider the beginning of a time step t . For a packet $p \in G(t, i)$, if p is forwarded to the next edge of its path, then p is added to $G^1(t + 1, i - 1)$. If p stays on the edge in the time slot t , then p is added to $G^2(t + 1, i)$.

Though Lemma 2 is obvious, it provides us a way of constructing the dynamic program using the index i in $G(t, i)$. Lemma 2 implies that when t is increased by 1, the number of groups is not strictly increased.

In the following, we introduce some new observations and ideas that our algorithm needs.

Consider a packet p . For each edge e in the path P_p , the packet experiences at least one time step on an edge. We define

$$b(p, e) = \text{delayed time slots for the packet } p \text{ on the edge } e,$$

where $b(p, e) \geq 1$ and the packet p is on the edge $e_p(i+1)$ at time $t + b(p, e)$ given $e = e_p(i)$.

A packet p 's duration is the sum of its delays on the edges, say, $d_p = \sum_{e \in P_p} b(p, e)$. In order to calculate the values $b(p, e)$, we introduce the time slots to calculate $b(p, e)$. For any edge e , the packet p arrives at e at time

$$t_{in}(p, e) := \sum_{e'} b(p, e'), \quad (2)$$

where $e' \in e_p(1), e_p(2), \dots, e_p(i-1)$ given $e_p(i) = e$. Also, the packet p leaves the edge e at time

$$t_{out}(p, e) = \sum_{e'} b(p, e'), \quad (3)$$

where $e' \in e_p(1), e_p(2), \dots, e_p(i)$ given $e_p(i) = e$. $b(p, e)$ is calculated as below:

$$b(p, e) = t_{out}(p, e) - t_{in}(p, e). \quad (4)$$

Instead of assigning integer variables to the values $t_{in}(p, e)$ in Eq. (2) and $t_{out}(p, e)$ in Eq. (3), we come up with a new idea. We regard the packet p as a unit-length job, $t_{in}(p, e)$ as p 's *release time*, $t_{out}(p, e)$ as p 's *deadline*, and e as the machine processing p . The machine e processes at most one job at a time. In the following, we introduce the way of tuning up the values in Eq. (4) to make the job p successfully processed. The range $[t_{in}(p, e), t_{out}(p, e)]$ is the *interval* to schedule the packet p on e .

We want to guarantee that for each edge e , in the time ranges that the packets p are ready/competing to move on e , there are sufficient number of time steps to do so. Lemma 1 indicates that starting from a time t , the edge e is busy for at least k time slots given k competing packets for the edge e . In order to specify the interval to schedule a packet p on the edge e , we must guarantee that the *work load density* (the ratio of the number of packets and the number of time slots in any continuous range) for the edge e cannot exceed 1 [15]. That is, for any time range $[t, t']$, we have

$$\frac{|\{p | t \leq t_{in}(p, e) < t_{out}(p, e) \leq t'\}|}{t' - t + 1} \leq 1, \quad \forall e \quad (5)$$

Note that Inequality (5) is a lower bound construction for the general case's makespan t' for the edge e . When this inequality is tight, it is feasible to schedule all the packets successfully using the EDF (earliest-deadline-first) policy, where $t_{out}(p, e)$ denotes the deadline. Consider the maximal interval in which the edge e is busy. We have the following observation.

Our dynamic programming algorithm is based on the formulation in Inequality (5). Given a makespan d for an edge e (for example, $d = t'$ in Inequality 5), we are looking at the earliest release time $t_{in}(p, e)$ for a packet p so that a schedule on an edge e ending at time d is feasible.

4.2 The Algorithm and Its Analysis

Along the way of describing our algorithm, we give the running time analysis as well as the correctness analysis. Some part of the correctness analysis has been given when we introduced the algorithm's ideas.

Denote $C(e)$ the set of packets having their paths P_p covering an edge e , $C(e) = \{p|e \in P_p\}$. Note that $\max_e C(e) = C$ where C denotes the congestion. Consider a packet p . Define $\psi(i, j), \forall i, j$, as the state of a packet i arrives at least the j -th edge $e_p(j)$ on its path P_i . Recall that $j \leq D$, where D is the dilation. Therefore, we have in total as most D^N different configurations to show the states of all the packets at a time. We index these configurations as $\Psi(1), \Psi(2), \dots, \Psi(Z)$, where $Z \leq D^N$. Our algorithmic contribution is to reduce the number of configurations needed. Our analysis below shows that the total number of configurations needed in the algorithm PDP is 2^N , much less than D^N where $D = \max_p l_p$.

Now, define

$$OPT(\Psi(k), t) = \begin{cases} 1, & \text{the configuration } \Psi(k) \text{ happens at the end of time step } t \\ 0, & \text{otherwise} \end{cases}$$

The Objective. The objective of minimizing the makespan d^* is to return the smallest value t so that $OPT(\Psi(k), t) = 1$ for the configuration $\Psi(k)$ when all the packets i arriving at least their destination edges with indexes $j (= l_i)$ in G .

The Base Case. The base case happens at the end of time step 1. We calculate $OPT(\Psi(k), 1)$ for all the indexes k . For each movement of a packet p , we list all the configurations that a greedy schedule moves packets. Consider each edge e and the competing packets $C(e)$. Define $C(e, 1) := \bigcup_p e_p(1)$ and $P(e, 1) := \{p|e_p(1) = e\}$. The total running time of the base case is thus to enumerate all the configurations $\Psi(k)$ and get the value $OPT(\Psi(k), 1)$.

$$\prod_{e \in C(e,1)} |P(e, 1)| \leq \left(\frac{N}{|C(e, 1)|} \right)^{|C(e,1)|} < \binom{N}{|C(e, 1)|} \leq \binom{N}{(N/2)} \approx \frac{2^N}{\sqrt{\pi \cdot N}} \tag{6}$$

since $|\bigcup_e P(e, 1)| = N$ and $P(e, 1) \cap P(e', 1) = \emptyset$, for all $e \neq e'$. This inequality holds due to Edwin Benckebach and Richard Bellman's formula, as well as the Stirling's approximation. We remark here that the parameterized running time $\prod_{e \in C(e,1)} |P(e, 1)|$ can be much less than the upper bound.

The Recursive Step. We consider the ways of calculating $OPT(\Psi(k), t)$. Due to the ideas introduced above, this configuration $\Psi(k)$ comes from the one step move for some packets and being idle for the remaining packets. For these N packets, we consider to partition them into two groups, the group of packets moving forward in a time step and the group of packets staying in the same time step. For each of such a partition, we transform from one configuration to another

configuration. These two configurations are called *neighboring configurations*. We have the following recursion:

$$OPT(\Psi(k), t) = \max_{k'} OPT(\Psi(k'), t - 1) \quad (7)$$

where $\Psi(k')$ at time $t - 1$ is a neighboring configuration of $\Psi(k)$ at time t . The correctness of the recurrence in Eq. 7 is based on the recursion discussed in Sect. 4.1.

In the following, we calculate the running time of the recursive step. Though the total configuration number is up to D^N , in this recursion, we only consider the neighboring configurations so that if p is in $\Psi(k)$ given p being on at least the j -th edge of its path P_p , then p is on at least the $(j - 1)$ th edge in the configuration $\Psi(k)$ if j is forwarded in time step t , otherwise, p should be on at least the j th edge at the beginning of time step t . The total running time in this recursive step is therefore bounded by 2^N . As t is bounded by $O(C + D)$ [8], we have the following result.

Theorem 3. *PDP is an optimal algorithm for scheduling packets on a graph with a total running time $O(2^N(C + D))$.*

The instance-dependent running time has been provided above, as $\prod_{e \in C(e, 1)} |P(e, 1)|$ in Inequality 6.

5 Conclusions

In this paper, we present two exact algorithms for the packet scheduling problem. The solution to the general problem brings more insights on designing approximation algorithms. We expect these algorithmic techniques help with solving packet routing problems.

References

1. Clementi, A.E.F., Ianni, M.D.: On the hardness of approximating optimum schedule problems in store and forward networks. *IEEE/ACM Trans. Network.* **4**(2), 272–280 (1996)
2. Graham, D.L., Knuth, D.E., Patashnik, O.: *Concrete Mathematics*, A Foundation for Computer Science, 2nd edn. Addison-Wesley, Boston (1994)
3. Harks, T., Peis, B., Schmand, D., Tauer, B., Koch, L.V.: Competitive packet routing with priority lists. *ACM Trans. Econ. Comput.* (TEAC) (2018)
4. Kleinberg, J., va Tardos: *Algorithm Design*. Pearson, New York (2006)
5. Korte, B., Vygen, J.: *Combinatorial Optimization, Theory and Algorithms*. Springer, Heidelberg (2018). <https://doi.org/10.1007/978-3-662-56039-6>
6. Leighton, F.T., Maggs, B.M., Ranade, A.G., Rao, S.B.: Randomized routing and sorting on fixed-connection networks. *J. Algorithms* **14**(2), 167–180 (1994)
7. Leighton, F.T., Maggs, B.M., Rao, S.B.: Packet routing and job-shop scheduling in $O(\text{congestion} + \text{dilation})$ steps. *Combinatorica* **14**(2), 167–186 (1994)

8. Leighton, F.T., Maggs, B.M., Richa, A.W.: Fast algorithms for finding $O(\text{congestion} + \text{dilation})$ packet routing schedules. *Combinatorica* **19**(2), 375–401 (1999)
9. Maggs, B.M.: A survey of congestion + dilation results for packet scheduling. In: Proceedings of the 40th Annual Conference on Information Sciences and Systems (CISS) (2006)
10. Mansour, Y., Patt-Shamir, B.: Greedy packet scheduling on shortest paths. *J. Algorithms* **14**, 449–465 (1993)
11. Mitzenmacher, M., Upfal, E.: Probability and Computing, 2nd edn. Cambridge University Press, Cambridge (2017)
12. Ostrovsky, R., Rabani, Y.: Universal $O(\text{congestion} + \text{dilation} + \log^{1+\epsilon} n)$ local control packet switching algorithms. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC), pp. 644–653 (1997)
13. Peis, B., Skutella, M., Wiese, A.: Packet routing: complexity and algorithms. In: Bampis, E., Jansen, K. (eds.) WAOA 2009. LNCS, vol. 5893, pp. 217–228. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12450-1_20
14. Peis, B., Wiese, A.: Universal packet routing with arbitrary bandwidths and transit times. In: Proceedings of the International Conference on Integer Programming and Combinatorial Optimization (IPCO), pp. 362–375 (2011)
15. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 6th edn. Springer, New York (2022). <https://doi.org/10.1007/978-1-4614-2361-4>
16. Rothvob, T.: A simpler proof for $O(\text{congestion} + \text{dilation})$ packet routing. In: Proceedings of the 16th Annual Conference on Integer Programming and Combinatorial Optimization (IPCO), pp. 336–348 (2013)
17. Tauer, B., Fischer, D., Fuchs, J., Koch, L.V., Zieger, S.: Waiting for trains: complexity results. In: Proceedings of the 6th Annual International Conference on Algorithms and Discrete Applied Mathematics (CALDAM), pp. 282–303 (2020)
18. Vcking, F.M.B.: A packet routing protocol for arbitrary networks. In: Proceedings of the 28th Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 291–302 (1995)