Scheduling with Locality by Routing

Alison Hsiang-Hsuan Liu
Department of Information and Computing Sciences, Utrecht University, The Netherlands

Fu-Hong Liu
Department of Industrial Engineering and Engineering Management, National Tsing Hua University, Hsinchu, Taiwan

Abstract

This work examines a strongly NP-hard routing problem on trees, in which multiple servers need to serve a given set of requests (on vertices), where the routes of the servers start from a common source and end at their respective terminals. Each server can travel free of cost on its source-to-terminal path but has to pay for travel on other edges. The objective is to minimize the maximum cost over all servers. As the servers may pay different costs for traveling through a common edge, balancing the loads of the servers can be difficult. We propose a polynomial-time 4-approximation algorithm that applies the parametric pruning framework but consists of two phases. The first phase of the algorithm partitions the requests into packets, and the second phase of the algorithm assigns the packets to the servers. Unlike the standard parametric pruning techniques, the challenge of our algorithm design and analysis is to harmoniously relate the quality of the partition in the first phase, the balances of the servers' loads in the second phase, and the hypothetical optimal values of the framework. For the problem in general graphs, we show that there is no algorithm better than 2-approximate unless $P = NP$. The problem is a generalization of unrelated machine scheduling and other classic scheduling problems. It also models scheduling problems where the job processing times depend on the machine serving the job and the other jobs served by that machine. This modeling provides a framework that physicalizes scheduling problems through the graph's point of view.

2012 ACM Subject Classification Theory of computation → Routing and network design problems

Keywords and phrases Makespan minimization, Approximation algorithms, Routing problems, Parametric pruning framework

Digital Object Identifier 10.4230/LIPIcs.MFCS.2024.69

Acknowledgements The authors would like to thank Sheng-Yin Chen and Chung-Shou Liao for their discussions in the early stages.

1 Introduction

We propose a generalization of the makespan minimization problem that concerns jobs with *locality* and machines with *preferences*. Consider two jobs sharing some common preprocessing subroutines. The time for serving these jobs is reduced if they are assigned to the same machine since the outcome of the preprocessing can be shared (instead of executing the preprocessing step from scratch again). Jobs are considered to be "close" to each other if they share more preprocessing. In other words, the *locality* of jobs indicates how much acceleration a schedule can achieve when these jobs are assigned to one machine.

On the other hand, the machines have *preferences* on jobs. That is, a machine may be better at serving some of the jobs. If the jobs have common preprocessing routines, the machine that has a preference for the routine can further accelerate the total processing time of the batch of jobs.

For example, consider assigning teaching tasks to faculty members in a department. The teaching tasks can be classified systematically as a rooted tree structure. For instance, an internal node representing Algorithm Design And Analysis course may have two children, Approximation Algorithms and Randomized Algorithms courses. Every faculty member has their own research area and prefers teaching the courses that match their expertise, where each expertise is a path in the tree starting from the root. A course in an area further away from a teacher's research direction takes this teacher more time to prepare and also makes them less happy. The department's goal is to balance the workload people take to prepare for courses. That is, the objective of the assignment is to make the most unhappy colleague as happy as possible.

In a typical routing problem, there are servers that are able to move along paths on a given graph. A server may be required to stop at a terminal, which is a vertex that is associated with the server. Additionally, a server may also need to travel to some vertices to serve some requests or collect some packets located at the vertices.

To model the job localities and the machine preferences, we propose the *scheduling with locality* problem (SCHEDULING-WITH-LOCALITY) and describe it as a routing problem on a graph with server-sensitive edge weights. Given a weighted graph, the jobs are represented by *requests* located at (some of) the vertices, and the distance between requests indicates how close the corresponding jobs are. The machines are represented by *servers* initially at a *source vertex* and aim to travel to their *terminals*. Serving a request by a server is analogous to assigning the corresponding job (of the request) to the corresponding machine (of the server). The traveling distance for a server to serve a request is analogous to the time for processing the corresponding job on the corresponding machine. A server may have some *discount* on the weights of some particular edges. That is, it pays a smaller traveling cost when traveling through these edges. The discount of servers on those edges indicates the *preference* of the corresponding machines serving particular types of jobs. The goal is to compute a schedule for all servers, which is a set of walks (where vertices/edges may repeat), each for one server. A walk corresponding to a server starts from the common source vertex and ends at the terminal of the server. To feasibly serve all requests, the walks should cover all of them. The traveling cost of a server on its walk is the total load (that is, the total processing time) of the corresponding machine. To minimize the makespan of a schedule, we want to find the walks where the maximum traveling cost (with regard to the corresponding server) is minimized. The modeling aims to provide a framework that is capable of transforming scheduling problems into a routing problem that has the potential to capture more properties of the scheduling problems from the graph's point of view.

Formal problem definition. An instance of SCHEDULING-WITH-LOCALITY is given by an undirected graph $G = (V, E)$, a set of *servers* $\mathcal{S}$, a set of *requests* $\mathcal{R} \subseteq V$, and a *source* $\rho \in V$. Each server $s \in \mathcal{S}$ is associated with its *terminal* $\tau(s) \in V$. Each edge $e \in E$ is associated with server-sensitive weights $weight(e, s) \geq 0$ for $s \in \mathcal{S}$, where $weight(e, s)$ is the cost for server s traveling through edge e. In a feasible schedule, each server $s \in \mathcal{S}$ is assigned a walk[1] that starts from ρ and ends at $\tau(s)$, and the walks of all servers in $\mathcal{S}$ all together visit all requests $\mathcal{R}$. Let Z_s be the walk assigned to server s by schedule Z, the cost of a server s in this schedule, $cost(Z_s, s)$, is $\sum_{e \in Z_s} weight(e, s)$ by definition. The cost of a schedule Z is defined as the maximum cost of servers on their corresponding walk, $\max_{s \in \mathcal{S}}\{cost(Z_s, s)\}$. The objective is to find a schedule that minimizes the cost, i.e., the maximum traveling cost of server walks.

We consider a special case **Multi-server Routing with Free-paths (MRF) problem**, where the given graph is a weighted tree $\mathcal{T} = (V, E)$ rooted at the source vertex ρ. There is an integral weight function on edges $weight : E \to \mathbb{N}$. We call the path from ρ to $\tau(s)$ the

[1] A *walk* is defined by a sequence of vertices $W = \{v_0, v_1, v_2, \cdots, v_\ell\}$ where v_{i-1} and v_i are adjacent for any $1 \leq i \leq \ell$. It is possible that a vertex or an edge appears multiple times in a walk.

home path of server s, denoted as $P_H(s)$. A server can travel on the edges of its home path free of cost. Formally, $weight(e, s) = 0$ if $e \in P_H(s)$ and $weight(e, s) = weight(e)$ otherwise. We will generalize the edge weights from integers to real numbers later in the paper.

Scheduling-with-Locality can model many scheduling problems, such as Unrelated machine scheduling [21], Machine scheduling with setup times [22], etc. (see the full version for details). On the other hand, although the preferences of servers are limited in the MRF problem, the problem captures the locality-aware properties of scheduling problems, i.e., the cost of a server serving a set of requests depends on the locality of the requests and the server. The problem still preserves servers' preferences for serving batches of requests and accommodates complicated inter-request localities by the tree structure. On the other hand, the routing is tricky due to different edge costs for different servers. In fact, one can show that without wisely partitioning the requests, any algorithm is at least r-approximate, where r is the number of requests.

Our results

For positive results, We propose a two-phase Partition-and-Balancing (PnB) algorithm that first partitions the requests into packets properly and then assigns the packets to the servers while balancing the serving costs of servers. More specifically, we apply the classic *parametric pruning* framework that has been exploited to solve the k-center problem [23]. Intuitively, the framework keeps guessing the optimal value. Given a parameter ϑ as the guessed optimal value, if an algorithm ALG guarantees to return a solution with a cost of $\alpha \cdot \vartheta$ as long as ϑ is a correct guess of the optimal cost, then ALG is α-approximate. Afterward, the framework concentrates on finding a correct guess.

In the two-phase PnB algorithm, we use the parameter ϑ as a hypothesized value of the optimal cost. In the first phase, we partition the requests into packets according to the value of ϑ. The partition ensures that for each packet, the cost of traveling to the packet and the cost of traveling within the packet (to serve the requests) are balanced for any server. Eventually, the cost of serving each packet falls between ϑ and 2ϑ, with a bounded number of packets with serving costs less than ϑ (Lemma 3). The second phase delicately assigns the packets, consulting the value of ϑ and the topology of the server terminals. In the end, with a large enough ϑ, the algorithm guarantees that a server only reaches packets within a distance bounded by a constant factor of ϑ, and the cost of any server is at most 4ϑ (Theorem 5). To make sure that if the algorithm fails to generate an assignment with the cost at most 4ϑ, then the value ϑ must be strictly smaller than the optimal cost, the algorithm is designed such that the information of the instance (that will be used in the second phase) when packing the requests into packets in the first phase is preserved (Theorem 8). Finally, the algorithm is adjusted to solve the problem with real number edge weights via runtime trade-off.

For negative results, we provide the 2-inapproximability for the general scheduling with locality problem (Scheduling-with-Locality) to complete the study (Theorem 15).

Related work

Makespan minimization has been studied in various settings where machines have different speeds with respect to different jobs [7, 12, 13, 15, 17, 20, 21], machines with status and setup costs [1, 10, 18], machines with reconfiguration overheads [19, 25], etc. On the other hand, the scheduling of jobs with locality has raised attention recently [14, 16, 25]. That is, reusing configurations among different jobs increases warm-starts and reduces cold-start overheads.

Jobs scheduling and load balancing. The closest related problem is the jobs scheduling problem with the objective of minimizing the makespan. Lenstra et al. proposed a classic LP-based polynomial time 2-approximation algorithm [21]. In contrast, Gairing et al. provided a combinatorial 2-approximation algorithm [15]. Later, Arad et al. used a parametric pruning style technique [2]. They showed that given values L and T, either there exists no schedule of mean machine completion time L and makespan T, or a schedule of makespan at most $T + L/h < 2T$ can be found in polynomial time, where $h \in (0, 1]$ is the feasibility factor of a given instance. An important special case of job scheduling problem is the restricted assignment problem, where processing times are of the form $p_{i,j} \in \{p_j, \infty\}$. The best known result of the problem to date is a $(\frac{33}{17} + \varepsilon)$-approximation by Svensson [26].

Multiple traveling salespeople problem. Another related problem is the *multiple traveling salespeople problem* [24]. Given a set C of n cities, k salespeople, and a depot $d \in C$, the multiple TSP problem aims to find k tours that start and end at the depot such that all of the cities in C must be visited by at least one salesperson. The objective is to minimize the maximum tour length over the k tours. In Euclidean metric, Monroe and Mount [24] showed that there exists a randomized algorithm for the multiple TSP problem that runs in the expected time $O(n((\frac{1}{\varepsilon})\log(\frac{n}{\varepsilon}))^{O(1/\varepsilon)})$, where $\varepsilon > 0$ is an approximation parameter. For the multiple distinct depots case, Xu and Rodrigues proposed a $\frac{3}{2}$-approximation algorithm [27]. For more approximation algorithms for a variety of vehicle routing problems within graphs, please see [3]. Note that the MRF problem should not be confused with the *k-delivery traveling salesman problem* [4, 6], which is a routing problem of limited-capacity vehicles with pickup and delivery locations.

Note that in these previous works, the number of servers is considered to be constant, while in the MRF problem, the number of servers is part of the input since each server has its own terminal. Moreover, in the MRF problem, the servers are allowed to travel through one edge multiple times. To decrease the overall cost, an optimal solution may send multiple servers through one edge. If the metric space in the multiple TSP problem is a tree, and the servers are allowed to travel through an edge more than once, the multiple TSP problem can be seen as a special case of our problem.

Other related work. MRF is similar to the *school bus problem* with regret minimization [8], in which an algorithm is additionally able to determine the locations of terminals. Bock et al. [8] provide a 13.5-approximation algorithm for the school bus problem. For more similar problems in vehicle routing, readers may refer to the survey [9]. On the other hand, the min-max objective considered in this paper also has a significant impact on other combinatorial optimization problems in vehicle routing [5] and efficiency of allocations [11].

Paper organization. In Section 2, we propose our algorithm PnB. In Section 3, we analyze the correctness and approximation ratio of the PnB algorithm. Finally, in Section 4, we provide complexity results of the SCHEDULING-WITH-LOCALITY problem. Due to the page limit, we skip most of the proofs. The complete version with proofs and pseudocodes is attached in the appendix.

2 Partition-and-Balancing Algorithm

We propose the Partition-and-Balancing (PnB) algorithm for the MRF problem, where the input graph is a weighted tree $\mathcal{T}$ rooted at the source vertex ρ, and $weight(e, s) = 0$ if the edge $e \in P_H(s)$ (that is, e is on the home path of s, from ρ to $\tau(s)$) and $weight(e, s) = weight(e)$ otherwise. We first specify the terminologies that will be used in the algorithm and analysis.

The input tree $\mathcal{T}$ can be partitioned into two components, *skeleton* (denoted by $\mathcal{T}_s$) and *request forest* (denoted by $\mathcal{F}_\mathcal{R}$) according to the servers' terminals and the set of requests $\mathcal{R}$. The skeleton is the union of the home paths of all servers, and the request forest is defined by $E(\mathcal{T}) \setminus \mathcal{T}_s$. Each connected component in $\mathcal{F}_\mathcal{R}$ is a tree rooted at some vertex on the skeleton. We call the trees *request trees* In other words, the request forest is a set of request trees (see Figure 1). We say that a server s is *based* at a (sub)tree T if $\tau(s) \in T$. We also say that an edge e is a *detour* of server s if $e \in P_H(s') \setminus P_H(s)$, where $s' \neq s$. Conceptually, a detour of server s is an edge that s travels at a non-free cost while there is another server that can travel through it free of cost. Note that the edges in request forest are not detours of any server, as all servers have to pay for travel in request forest.

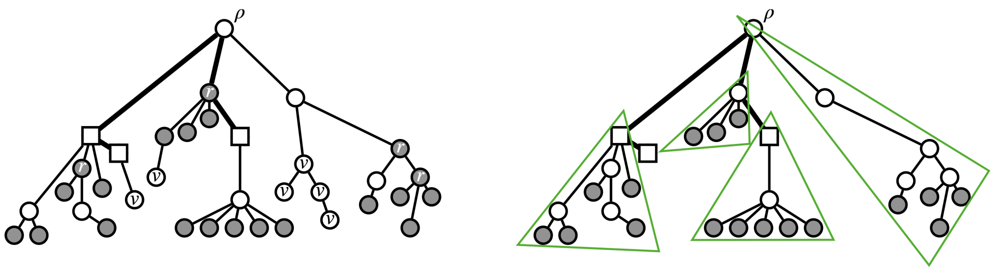
Given an edge-weighted graph, a *walk* is a sequence of vertices $W = \{v_0, v_1, v_2, \cdots, v_\ell\}$ such that v_{i-1} and v_i are adjacent for any $1 \leq i \leq \ell$. The walk W is called a *closed walk* if $v_0 = v_\ell$. The *cost* of walk W is the total weight of the edges and formally $\sum_{i=1}^{\ell} weight((v_{i-1}, v_i))$. Note that in a walk, the vertices or edges may be repeated, and a repeated edge contributes its weight multiple times in the cost of the walk. Given a weighted rooted tree $T = (V, E)$ with weight function $weight : E \to \mathbb{N}$, we denote the subtree rooted on vertex $v \in V$ by T_v. In T, the unique path between a pair of vertices u and v is called (u, v)-*path* and denoted by $P_{u,v}$. We further denote the distance between two vertices u and v by $dist(u, v)$, which is the sum of weights of the edges on $P_{u,v}$. For any connected component $C \subseteq E$, which can be a walk, a path, or even a subtree, we denote $weight(C)$ as the total weight of the edges in C.

For convenience, we define two operations, $\texttt{Walk}(P)$ and $\texttt{Merge}(C, W)$. The $\texttt{Walk}(P)$ operation makes the path P into a walk by going back and forth on the path. Formally, given that $P = [v_1, v_2, \cdots, v_\ell]$, $\texttt{Walk}(P)$ returns a walk $W = \{v_1, v_2, \cdots, v_{\ell-1}, v_\ell, v_{\ell-1}, v_{\ell-2}, \cdots, v_2, v_1\}$. The function $\texttt{Merge}(C, W)$ is to merge the closed walk W to the component C, which can be a path or a walk, by concatenating the two components properly. Note that the $\texttt{Merge}(C, W)$ function is feasible only when W is a closed walk, and there is at least a vertex that is in W and in C at the same time. More formally, assume that $C = \{v_1, v_2, \cdots, v_\ell\}$ is a walk and $W = \{u_1, u_2, \cdots, u_k, u_1\}$, where $v_x = u_y$ for some x and y, $\texttt{Merge}(C, W)$ returns a walk $\{v_1, v_2, \cdots, v_x = u_y, u_{y+1}, \cdots, u_k, u_1, \cdots, u_y = v_x, v_{x+1}, \cdots, v_\ell\}$. If there are multiple vertices that are in both C and W, we break ties arbitrarily.

Instance transformation. We first transform any input tree into a special form, where all requests are at the leaves, and the PnB algorithm works on the transformed instance. Formally, given the input tree $\mathcal{T}$, we first remove all vertices v such that there are no terminals or requests in T_v. Then, we remove the requests on any internal node of the resulting tree. (See Figure 1.)

One can prove by contradiction that in an optimal schedule, no server travels to any of the removed vertices. Moreover, since we only remove requests at the vertices with descendants that also contain requests or terminals, an optimal solution on the original input tree is a feasible schedule of the modified input tree. By a similar reasoning, we can argue that PnB($\mathcal{T}''$) is a feasible schedule of $\mathcal{T}$, where $\mathcal{T}''$ is the modified input tree. Therefore, we have the following lemma, and it is legal to restrict our discussion to the instance where all requests are at the leaves:

▶ **Lemma 1.** *If the PnB algorithm is α-approximate on the modified input $\mathcal{T}''$, PnB algorithm is α-approximate on input $\mathcal{T}$.*



(a) Original instance. **(b)** Instance after transformation.

Figure 1 Instance transformation. The squares are terminals, and the gray circles are requests. The thick edges indicate the skeleton $\mathcal{T}_s$. The vertices labeled by v and requests at the vertices labeled by r in 1a are removed in 1b. In 1b, each of the green triangles indicates a request tree.

2.1 PnB algorithm

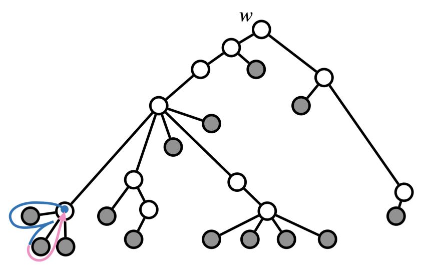
The PnB algorithm follows the parametric pruning framework and has two phases: `Partition` and `Assignment`. Taking parameter $\vartheta = 1, 2, 3, \cdots$, the algorithm PnB$(\mathcal{T}, \vartheta)$ first calls `Partition`$(\mathcal{T}, \vartheta)$. If `Partition`$(\mathcal{T}, \vartheta)$ does not fail, it returns an updated tree $\mathcal{T}'$ that only consists of the skeleton while each request tree T_w rooted at w is packed into packets stored at the vertex w. Next, the algorithm calls `Assignment`$(\mathcal{T}', \vartheta)$. If `Assignment`$(\mathcal{T}', \vartheta)$ does not fail, it returns a set of $|\mathcal{S}|$ walks, which is an assignment of the packets to the servers. Whenever a phase fails, we terminate the process of PnB$(\mathcal{T}, \vartheta)$ and move on to the next ϑ value.

2.1.1 `Partition`: Packing requests (Phase 1)

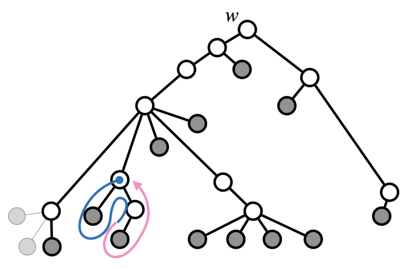
Given a value of ϑ, `Partition` deals with the request trees one by one and returns `fail` if there is a request tree with a depth (that is, the length of the longest path from w) larger than $\vartheta/2$. In a request tree T_w, the procedure keeps formulating packets of requests in a bottom-up manner, removing the walk that travels through the packed requests from the request tree, and finally, storing the formulated packets at the root of the request tree. After the `Partition` procedure, if it does not return `fail`, the updated tree $\mathcal{T}'$ has a special form, where the vertices in the request trees except the root are all deleted, and the vertices that were the roots of request trees now store packets. Moreover, each leaf is a terminal of some server in the updated tree $\mathcal{T}'$.

Checking the vertices in the request tree from bottom-top. For each request tree T_w rooted at w, the `Partition` procedure checks the vertices $v_1, v_2, \cdots, v_k$ in T_w in post order. If the subtree T_{v_i} has a weight of less than $\frac{\vartheta - dist(w,v_i)}{2}$, we finish checking v_i and move on to the next vertex v_{i+1} in T_w in the post order. Otherwise, T_{v_i} is "heavy" enough, and we proceed to the next step for partitioning the requests in T_{v_i}.

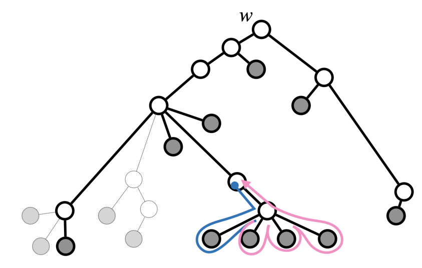
Partition the requests in a heavy-enough subtree T_{v_i} of request tree T_w. We travel T_{v_i} in the depth-first-search traversal, which is a walk $W_{v_i} = \{v_i, u_1, u_2, \cdots, u_{k_1}, v_i, u_{k_1+1}, u_{k_1+2}, \cdots, u_{k_2}, v_i, \cdots, v_i\}$. Let $\beta_{v_i} = \vartheta - 2 \cdot dist(w, v_i)$. We partition W_{v_i} into sub-walks, $B_1, B_2, \cdots, B_m$, such that each B_j starts and ends at v_i and is the walk with smallest total weight such that $weight(B_j) \geq \beta_{v_i}/2$. Then, each sub-walk B_j is merged to the walk `Walk`(P_{w,v_i}) and forms a packet. The involved vertices in B_j are then trimmed



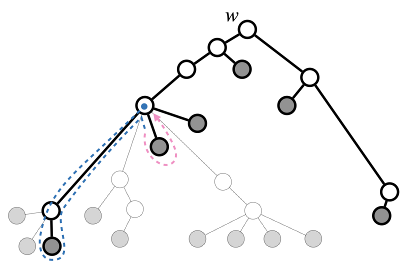
(a) Illustration of DFS packing. (1)



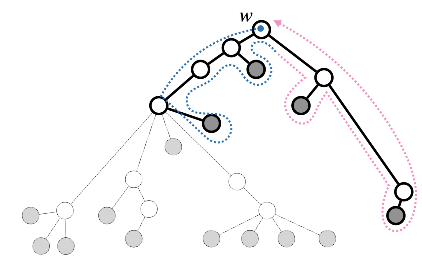
(b) Illustration of DFS packing. (2)



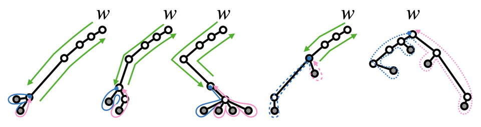
(c) Illustration of DFS packing. (3)



(d) Illustration of DFS packing. (4)



(e) Illustration of DFS packing. (5)



(f) The five corresponding packets stored at vertex w.

Figure 2 An illustration of `Partition`(ϑ) on T_w. In 2a–2e, the blue trajectories show the walks paid by the "budget" $\beta_\cdot = \vartheta - 2 \cdot dist(w, \cdot)$ (where $\cdot$ is the vertex the corresponding DFS starts at), and the red trajectories show the remaining walks back to the vertex $\cdot$. Figure 2f shows the five corresponding packets stored at the vertex w. The green trajectories are the walks `Walk`($P_{w,\cdot}$).

from T_w (except v_i, which is trimmed only when all requests in T_{v_i} are packed, and w, which is never trimmed). Note that the last partition of W_{v_i} may have a weight of less than $\beta_{v_i}/2$. In this case, this sub-walk will be pended and dealt with again when the next vertex v_{i+1} in T_w (in the post-order) is checked. Finally, if at the end of packing on T_w, there are still vertices left in the (trimmed) T_w, these left-over vertices are packed into one packet and stored at w. (See Figure 2.)

2.1.2 `Assignment`: Assigning packets (Phase 2)

After the procedure `Partition` in Section 2.1.1, the modified input tree $\mathcal{T}'$ is exactly the skeleton $\mathcal{T}_s$ with each request tree T_w rooted at the vertex $w \in \mathcal{T}_s$ packed into packets and stored at w. Moreover, every leaf in $\mathcal{T}'$ is a terminal of some server. Let $\mathcal{B}$ denote the set of all packets returned by `Partition`, and $\mathcal{B}_w$ denotes the set of packets stored at vertex

w ($\mathcal{B}_v$ is *empty* if there is no request tree rooted at v). By the process of `Partition`, the packets in $\mathcal{B}_w$ cover all requests in the request tree T_w. In the second phase of our algorithm, `Assignment`, we visit the vertices in $\mathcal{T}'$ in a post-order. When a vertex v is visited, we assign the packets stored in the subtree T_v but "far" enough from v to a server, which has its terminal nearby, and balance the load among the servers. The algorithm returns `fail` if it fails to balance the load of the servers.

Initial assignment. In the round of visiting v, we check its children $c_1, c_2, \cdots, c_k$ one by one. When the child c_i is checked, for any vertex w that is in the subtree T_{c_i} and $dist(w, v) > \frac{\vartheta}{2}$, we *release* the packets in $\mathcal{B}_w$ and assign the packets to a server s based at w (that is, $\tau(s) \in T_w$)[2]. Ties are broken arbitrarily. Note that some servers may have already been assigned packets in previous rounds as we visit the vertices in $\mathcal{T}'$ in a post-order.

To describe how to reassign the packets among servers, we first define the terminologies. We call the total weight of the packets assigned to server s the *work* of s. A server is *heavy*, *light*, or *normal* if its work is strictly greater than 3ϑ, strictly smaller than ϑ, or otherwise, respectively. If a server s' is assigned to serve packets that are initially assigned to another server s, s' is called *helping* for s. Otherwise, a server that helps no other servers is *free*.

Re-assignment for load-balancing. After the initial assignment when checking a child c_i of v, the servers with terminals in T_{c_i} may be light, normal, or heavy. For a server s that becomes heavy in this round, we find some light servers based at T_{c_i} to help it with some of the packets as follows. We first check if there is a light server s' that is based at T_{c_i} and already helped s with some packets. If so, we re-assign some packets released in this round to s'. The re-assignment stops once s or s' becomes normal. If at this moment, s is still heavy, then we search for another light helping server for s based at T_{c_i} and repeat the re-assignment. If there is no light server that helped s before, we find a light free server with its terminal in T_{c_i} that has not helped any other server. If all light servers based at T_{c_i} are helping other servers, and s is still heavy, the procedure `Assignment` returns `fail`.

Wrap the packets into walks. Finally, let $W_s \subseteq V(\mathcal{T}')$ be the set of vertices that store packets assigned to server s and L_s be the set of vertices on the $(\rho, \tau(s))$-path. The packets assigned to s can be wrapped up into a walk from ρ to $\tau(s)$ by traversing the subtree induced by $W_s \cup L_s$.

Figure 3 is an example of the released packets when checking c and the servers' status after the initial assignment. The pseudocode of `Assignment` can be found in the full version.

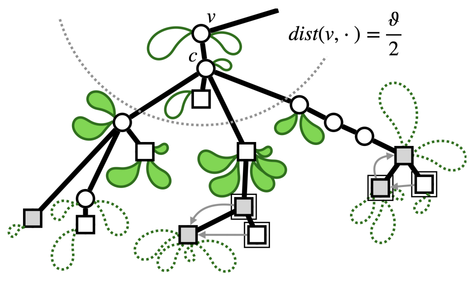
3 Analysis of PnB algorithm

Throughout the analysis, we use OPT to denote the optimal schedule. We also use OPT to denote the value of the optimal schedule when the context is clear.

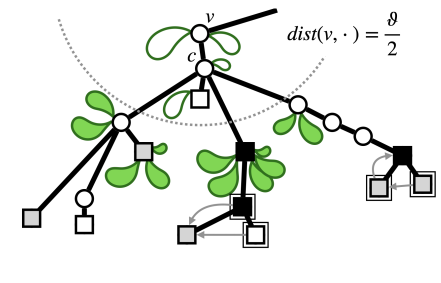
Sizes of the packets successfully returned by `Partition` are bounded

Recall that the PnB algorithm is run on a modified input tree $\mathcal{T}$, where each leaf is a terminal of some server. With parameter ϑ, the procedure `Partition` either returns `fail` or modifies the input tree $\mathcal{T}$ into $\mathcal{T}'$ that contains the same vertices as $\mathcal{T}'$, and the packets are stored at

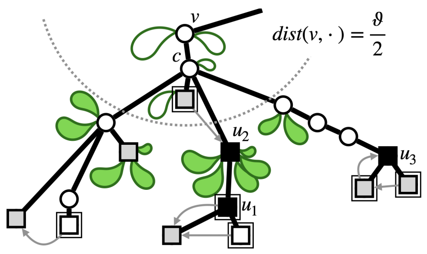
[2] Note that all these vertices on the path from v to c_i to w are all in $\mathcal{T}'$.



(a) The requests are released.



(b) After initial assignment.



(c) After re-assignment.

Figure 3 Illustration of `Assignment`(ϑ) when v is visited and in the round of checking c. Each bean-shaped object is a packet from `Partition`(ϑ). The solid bean-shaped green objects are released when c is checked, as they are outside the range of $\frac{\vartheta}{2}$ from v. The hollow and dashed-lined ones are packets released in previous rounds, and the hollow and solid-lined ones have not been released. The squares indicate the servers' terminals. The square is white/black/gray if its corresponding server is light/heavy/normal. When a terminal is a double-layered square, its corresponding server is helping. The gray arrows from $\tau(s')$ to $\tau(s)$ indicate that s' is helping s with some packets. Note that in this example, after checking c, `Assignment`(ϑ) must return `fail` as the servers with the terminals at u_1, u_2 and u_3 are heavy, but there are no free light servers in T_c.

some of these vertices. Consider any request tree T_w rooted at $w \in \mathcal{T}'$. By construction, the procedure `Partition` packed each request in T_w into exactly one packet that is stored at the vertex w, as long as `Partition`(ϑ) does not return `fail`.

`Partition` fails when there is a path $P_{w,\ell}$, where $w \in \mathcal{T}'$ and ℓ is a leaf in T_w, such that $dist(w,\ell) > \frac{\vartheta}{2}$. Since we work on the modified input tree, the only vertex that can be a terminal in T_w is 2. Thus, any server that serves the request on ℓ has cost at least $2 \cdot dist(w,\ell)$. Therefore, we have the following lemma:

▶ **Lemma 2.** *If* `Partition`(ϑ) *returns* `fail`, *then* $OPT > \vartheta$.

On the other hand, if `Partition` returns a partition regarding ϑ, each packet B stored at vertex $w \in \mathcal{T}'$ is a closed walk containing w. For any packet B packed when $v_i \in T_w$ is visited, it was packed using a budget of $\vartheta - 2 \cdot dist(w, v_i)$. Since we only pack heavy enough request trees, and the vertices in the request tree are packed from the bottom up, the trip back to v_i after the budget is off is bounded by a function of ϑ and the distance between w and v_i. By careful calculations, we have the following lemma:

▶ **Lemma 3.** *If* `Partition`(ϑ) *does not return* `fail`, *for any packet* B, $weight(B) \le 2\vartheta$.

`Assignment` feasibly returns a schedule with a cost of 4ϑ

After `Assignment`(ϑ), if it does not return `fail`, each server is assigned a set of packets, where each packet is a closed walk by the construction. Recall that a packet $B \in \mathcal{B}_w$ is assigned to a server s' either because B is assigned to s initially or s' is helping another server s with B. In the prior case, w is on $P_H(s') = P_{\rho,\tau(s')}$ and can be merged with $P_H(s')$ without incurring any detour edges. In the latter case, w is on $P_H(s) = P_{\rho,\tau(s)}$, and B was released when the algorithm visited some vertex $c \in P_{\rho,w}$, which is a common ancestor of $\tau(s')$ and $\tau(s)$. Thus, B can be merged with the walk `Walk`($P_{a,\tau(s)}$), and the merged walk can be merged with the path $P_H(s')$. Therefore, the packets assigned to server s' can be merged with $P_H(s')$ into a walk from the root to $\tau(s')$, and we have the following feasibility of the PnB algorithm (see the full version for a complete proof):

▶ **Theorem 4.** *Given instance $\mathcal{T}$, the transformed instance $\mathcal{T}'$, and a parameter of value ϑ, if* `Assignment`($\mathcal{T}', \vartheta$) *does not return* `fail`*,* PnB($\mathcal{T}, \vartheta$) *is feasible. Moreover, the cost of* PnB($\mathcal{T}, \vartheta$) *equals to the cost of* PnB($\mathcal{T}', \vartheta$)*.*

To bound the cost of any server, we first bound the cost spent on the server's detours. By the selection of helping servers, any server s' that helps another server s with packet $B \in \mathcal{B}_w$ is taking detours with a total distance of at most $2 \cdot dist(w, a)$, where a is the lowest common ancestor of $\tau(s')$ and $\tau(s)$. Since in each round, we only release "far-away" packets in terms of ϑ, $2 \cdot dist(w, a) \leq \vartheta$. Hence, the total detour length of s' is at most ϑ. Therefore, we have the following theorem:

▶ **Theorem 5.** *Given that* `Assignment`(ϑ) *does not return* `fail`*, the cost of any server on serving all packets assigned to it is at most 4ϑ.*

The lower bound of the optimal cost

There are two occasions that PnB returns `fail`; one is in the procedure `Partition`, and another one is in the procedure `Assignment`. In Lemma 2, we have shown that if `Partition` returns `fail`, the optimal cost must be larger than ϑ. Next, we consider the case when `Partition` successfully returns a partition of requests while `Assignment` returns `fail`.

We first argue that the partition returned by `Partition`(ϑ) is *sufficiently cost-efficient.* More specifically, if `Partition`(ϑ) does not return `fail`, and the optimal cost is at most ϑ, then total cost the optimal schedule has to pay is comparable to the total work of all packets.

▶ **Lemma 6.** *Given that* `Partition`(ϑ) *does not return* `fail`*, if $OPT \leq \vartheta$, then the sum of the cost of servers in the optimal schedule is at least $\sum_{packets\ B} weight(B)$. That is, $\sum_{s \in \mathcal{S}} cost^{OPT}(s) \geq \sum_{B \in \mathcal{B}} weight(B)$.*

Proof (Sketch). We denote n^*_e as the number of servers OPT sends over the edge e, and b_e as the number of packets constructed by `Partition`($\vartheta, \mathcal{T}$) that contain e. It is equivalent to prove that if OPT $\leq \vartheta$, $n^*_e \geq b_e$ for any edge $e \in \mathcal{F}_{\mathcal{R}}$. We prove this claim by induction on the edges with depths from the bottom up. Consider an edge $e = (u, v)$ in a request tree T_w, where u is v's parent. Denote $weight_{T_v}(B)$ the total weight of edges in packet B that are also in T_v. The total cost of servers in the optimal solution spent in the subtree T_v is $\sum_{e' \in T_v} n^*_{e'} \cdot weight(e') \geq \sum_{e' \in T_v} b_{e'} \cdot weight(e')$ by the inductive hypothesis. By construction, any packet that contains $e' \in T_v$ must contain the edge $e = (u, v)$. Therefore, the total weight $\sum_{e' \in T_v} b_{e'} \cdot weight(e') = \sum_{B : e \in B} weight_{T_v}(B)$.

Next, we bound $\sum_{B : e \in B} weight_{T_v}(B)$. By case distinction, any packet B containing the edge e that was packed when v or some descendent of v is visited has $weight_{T_v}(B) \geq \vartheta - 2 \cdot dist(w, v)$. Moreover, there is at most another packet containing the edge e that was packed when some ancestor of v is visited. Therefore, the total weight of the packets that contain e is at least $m \cdot (\vartheta - 2 \cdot dist(w,v)) + weight(\hat{B})$, where m is the total number of packets formed by visiting v or some descendent of v, and $\hat{B}$ is the packets packed when an ancestor of v is visited. Meanwhile, $b_e = m + \mathbb{1}[\text{there is at least one packet in } \hat{B}]$[3].

Finally, since OPT $\leq \vartheta$, and v is in the request tree T_w, any server that travels to v has to finish its route in T_v with a cost of at most $\vartheta - 2 \cdot dist(w,v)$. Therefore, an optimal solution has to send at least $\frac{\sum_{B:e\in B} weight_{T_v}(B)}{\vartheta - 2\cdot dist(w,v)}$ servers over the edge e to serve all requests in T_v. Hence, $n_e^* \geq m + \mathbb{1}[\text{there is at least one packet in } \hat{B}] = b_e$, since n_e^* is integral. ◀

Next, we show that if `Assignment`(ϑ) returns `fail`, the average cost of the servers in the (failed) `Assignment`(ϑ) assignment on serving the packets must be strictly larger than ϑ.

▶ **Lemma 7.** *Given that* `Assignment`(ϑ) *returns* `fail`*, on average, each server in the PnB assignment is assigned by packets with total work strictly larger than* ϑ*.*

Proof (Sketch). Assume that `Assignment`(ϑ) returns `fail` when visiting v and checking the child c. Consider the set $\mathcal{S}'$ of servers that an optimal solution uses to serve the requests in T_c. There are two cases of the optimal solution: 1) there exists at least one server in $\mathcal{S}'$ who has its terminal outside T_c, or 2) all servers in $\mathcal{S}'$ have their terminals in T_c. It is easy to see in case 1, the optimal cost is at least ϑ due to any packets inside T_c are at least $\vartheta/2$ away from c.

For Case 2, we use a *helping forest* argument described below. Consider the servers' packets with their terminals in T_c at the moment when `Assignment`(ϑ) returns `fail`. We construct helping trees, which are rooted trees, where each node corresponds to a server by making s' a child of s if s' helps s with some packets.[4] In each helping tree, the root and the internal nodes are normal (or heavy if it is the server that triggers `Assignment`(ϑ) to return `fail`). Consider any internal node s and its children, which are sorted by the order that they help s. The servers s and all its children but the last one must be all normal. On the other hand, the last child can be normal or light. Furthermore, if the last child is light, it is a leaf, as it needs no one to help it. Hence, any internal node has at most one light leaf. The total work of an internal node s and its light leaf s_ℓ is at least 3ϑ, since $work(s) > 3\vartheta$ right before s_ℓ helps s. Therefore, the theorem is proven by the fact that the average work of the servers with terminals in T_c is strictly greater than $\frac{n\cdot\vartheta + \ell\cdot 3\vartheta}{n+2\ell} > \vartheta$, where ℓ is the number of internal node-light leaf pairs, and n is the number of other nodes in the helping forest. ◀

Now, we are ready to prove our main theorem.

▶ **Theorem 8.** *Given that* PnB($\mathcal{T}, \vartheta$) *returns* `fail`*,* $OPT > \vartheta$*.*

Proof. By the construction, `Partition`(ϑ) returns `fail`, the optimal cost must be strictly larger than ϑ. When `Assignment`(ϑ) returns `fail`, there are two cases of the optimal cost: 1) OPT $> \vartheta$, and 2) OPT $\leq \vartheta$. Suppose on contrary that `Assignment`(ϑ) returns `fail` and OPT $\leq \vartheta$. Then, OPT $\geq$ average cost of all servers $= \frac{\sum_{s\in\mathcal{S}} cost^{\text{OPT}}(s)}{|\mathcal{S}|} \geq \frac{\sum_{B\in\mathcal{B}} weight(B)}{|\mathcal{S}|}$, where the inequality is by Lemma 6. By Lemma 7, $\frac{\sum_{B\in\mathcal{B}} weight(B)}{|\mathcal{S}|} > \vartheta$. In result, OPT $> \vartheta$, which contradicts to the assumption that OPT $\leq \vartheta$. ◀

By Theorem 5, Theorem 8, and the parametric pruning framework, we get the approximation ratio:

▶ **Theorem 9.** *PnB is a* 4*-approximation.*

[3] The function $\mathbb{1}[X]$ is 1 if statement X is true and is 0 otherwise.
[4] By the construction, a server s helped by another server will not help other servers, while a helping server s' may be helped by other servers later.

4 Runtime and complexity

4.1 Runtime

Based on the pseudocode, the runtime of PnB depends on a polynomial function of the number of vertices, the number of requests, the number of servers and the value of the optimal schedule. We can slightly improve it as follows. Previously, we search for ϑ^* through a linear scan of ϑ starting from 1, 2, and so on. The scan can be accelerated by binary search.

More specifically, we make ϑ starting from 1, double its value until PnB outputs a feasible schedule, and then binary search within the range up to the ϑ of the feasible schedule. Therefore, for the part of the value of optimal schedule, the runtime reduces to a logarithmic function of the value.

▶ **Theorem 10.** *PnB runs in $O(|V|^3 \log OPT(\mathcal{T}))$ time where $|V|$ is the number of vertices in the input tree $\mathcal{T}$.*

▶ **Corollary 11.** *PnB runs in polynomial time.*

By the binary search method, we can generalize the edge weights from integers to real numbers.

▶ **Theorem 12.** *For real number edge weights, PnB is $(4+\epsilon)$-approximation with runtime $O(|V|^3(4/\epsilon - 1 + \log\lceil OPT(\mathcal{T})\rceil))$ where $|V|$ is the number of vertices in the input tree $\mathcal{T}$.*

4.2 Complexity results

First, by reducing from 3-Partition, we have the following theorem for MRF:

▶ **Theorem 13.** *MRF is strongly NP-hard even if the input is a star.*

Then, we move on to the complexity of Scheduling-with-Locality. We show that the unrelated machine scheduling problem can be reduced to a special case of the scheduling with locality problem. Thus, the problem is more general than the unrelated machine scheduling problem in the sense of approximation ratios.

▶ **Theorem 14.** *For $\alpha < 1.5$, there does not exist a polynomial-time α-approximation algorithm for Scheduling-with-Locality even if we restrict that a server only gets discounts on the edges of the path from the source vertex to its terminal, unless $P = NP$.*

For the rest of the section, we show the following theorem by a reduction from 3-dimensional matching.

▶ **Theorem 15.** *For $\alpha < 2$, there does not exist a polynomial-time α-approximation algorithm for Scheduling-with-Locality even if the edge weights consist of only 0 and 1, unless $P = NP$.*

3-Dimensional matching. In the problem, there are disjoint vertex sets A, B and C with $|A| = |B| = |C| = n$. There is also a set of triples T. Each $t \in T$ is (a_t, b_t, c_t) for some $a_t \in A$, $b_t \in B$ and $c_t \in C$. There is a matching if we can find n triples that cover $A \cup B \cup C$. It is known that 3-dimensional matching is NP-hard.

Construction of the reduction. If $|T| < n$, we construct a trivial no-instance. Thus in the following, we assume $|T| \geq n$. For ease of the description, there is no discounts on any edge for each server by default. For each vertex $x \in A \cup B \cup C$, there is a corresponding request vertex r_x. There are additionally $|T| - n$ dummy request vertices D. For each triple $t = (a, b, c) \in T$, there are four vertices v_t, x_t, y_t and z_t. Each triple has a corresponding server. Let s be the corresponding server of t. Vertex v_t is the terminal of s. There are edges (v_t, x_t), (x_t, y_t) and (y_t, z_t). The weights of the three edges is 1 and the discounts of (x_t, y_t) and (y_t, z_t) for server s are 0. There are additionally edges (x_t, r_a), (y_t, r_b) and (z_t, r_c) all with weight 0. In addition, v_t is adjacent to all the dummy requests D, and these edges all have weight 1. Finally, v_t is also adjacent to the source vertex. The edge has weight 1, but the corresponding server s has a discount 0.

The construction has the following properties.

▶ **Observation 16.** *Consider two different servers s and s', and their corresponding triples $t = (a, b, c) \in T$ and $t' = (a', b', c') \in T$ respectively. Any walk starting from the terminal of s and ending at x'_t incurs a cost of at least 2 for server s if $a \neq a'$. The reversed walk (from x'_t to the terminal of s) also incurs a cost of at least 2 for server s if $a \neq a'$. Both statements are also true if we replace the vertices (a, a', x'_t) by (b, b', y'_t) or replace (a, a', x'_t) by (c, c', z'_t).*

▶ **Lemma 17.** *For* Scheduling-with-Locality*, the question of deciding if there exists a feasible schedule with makespan at most 2 is NP-hard.*

Proof of Theorem 15. By the construction and Observation 16, it is easy to see that any feasible schedule with a makespan strictly larger than 2 has a makespan larger than 4. Thus, the theorem follows. ◀

References

1 Ali Allahverdi. The third comprehensive survey on scheduling problems with setup times/costs. *Eur. J. Oper. Res.*, 246(2):345–378, 2015. doi:10.1016/j.ejor.2015.04.004.

2 Dor Arad, Yael Mordechai, and Hadas Shachnai. Tighter bounds for makespan minimization on unrelated machines. *CoRR*, abs/1405.2530, 2014. arXiv:1405.2530.

3 Esther M. Arkin, Refael Hassin, and Asaf Levin. Approximations for minimum and min-max vehicle routing problems. *J. Algorithms*, 59(1):1–18, 2006. doi:10.1016/j.jalgor.2005.01.007.

4 Tetsuo Asano, Naoki Katoh, Hisao Tamaki, and Takeshi Tokuyama. Covering points in the plane by k-tours: Towards a polynomial time approximation scheme for general k. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 275–283. ACM, 1997. doi:10.1145/258533.258602.

5 Tolga Bektas and Adam N. Letchford. Using ℓ^p-norms for fairness in combinatorial optimisation. *Comput. Oper. Res.*, 120:104975, 2020. doi:10.1016/j.cor.2020.104975.

6 Binay K. Bhattacharya and Yuzhuang Hu. k-delivery traveling salesman problem on tree networks. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPIcs*, pages 325–336. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.FSTTCS.2012.325.

7 Marcin Bienkowski, Artur Kraska, and Hsiang-Hsuan Liu. Traveling repairperson, unrelated machines, and other stories about average completion times. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.28.

8 Adrian Bock, Elyot Grant, Jochen Könemann, and Laura Sanità. The school bus problem on trees. *Algorithmica*, 67(1):49–64, 2013.

9 Kris Braekers, Katrien Ramaekers, and Inneke Van Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. *Comput. Ind. Eng.*, 99:300–313, 2016.

10 John L. Bruno and Peter J. Downey. Complexity of task sequencing with deadlines, set-up times and changeover costs. *SIAM J. Comput.*, 7(4):393–404, 1978. doi:10.1137/0207031.

11 Ioannis Caragiannis, Christos Kaklamanis, Panagiotis Kanellopoulos, and Maria Kyropoulou. The efficiency of fair division. *Theory Comput. Syst.*, 50(4):589–610, 2012.

12 L. Chen, K. Jansen, and G. Zhang. On the optimality of approximation schemes for the classical scheduling problem. In *25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–668, 2014.

13 Shichuan Deng, Jian Li, and Yuval Rabani. Generalized unrelated machine scheduling problem. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 2898–2916. SIAM, 2023. doi:10.1137/1.9781611977554.ch110.

14 Alexander Fuerst and Prateek Sharma. Locality-aware load-balancing for serverless clusters. In Jon B. Weissman, Abhishek Chandra, Ada Gavrilovska, and Devesh Tiwari, editors, *HPDC '22: The 31st International Symposium on High-Performance Parallel and Distributed Computing, Minneapolis, MN, USA, 27 June 2022 – 1 July 2022*, pages 227–239. ACM, 2022. doi:10.1145/3502181.3531459.

15 Martin Gairing, Burkhard Monien, and Andreas Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. *Theor. Comput. Sci.*, 380(1-2):87–99, 2007. doi:10.1016/j.tcs.2007.02.056.

16 Maxime Gonthier, Loris Marchal, and Samuel Thibault. Locality-aware scheduling of independent tasks for runtime systems. In Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci, editors, *Euro-Par 2021: Parallel Processing Workshops – Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*, volume 13098 of *Lecture Notes in Computer Science*, pages 5–16. Springer, 2021. doi:10.1007/978-3-031-06156-1_1.

17 Sungjin Im and Shi Li. Improved approximations for unrelated machine scheduling. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 2917–2946. SIAM, 2023. doi:10.1137/1.9781611977554.ch111.

18 Klaus Jansen, Marten Maack, and Alexander Mäcker. Scheduling on (un-)related machines with setup times. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 145–154. IEEE, 2019. doi:10.1109/IPDPS.2019.00025.

19 Hessam Kooti and Eli Bozorgzadeh. Reconfiguration-aware task graph scheduling. In Eli Bozorgzadeh, João M. P. Cardoso, Rui Abreu, and Seda Ogrenci Memik, editors, *13th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013, Porto, Portugal, October 21-23, 2015*, pages 163–167. IEEE Computer Society, 2015. doi:10.1109/EUC.2015.33.

20 Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 217–224. IEEE Computer Society, 1987. doi:10.1109/SFCS.1987.8.

21 Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46:259–271, 1990. doi:10.1007/BF01585745.

22 Alexander Mäcker, Manuel Malatyali, Friedhelm Meyer auf der Heide, and Sören Riechers. Non-preemptive scheduling on machines with setup times. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures – 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 542–553. Springer, 2015. doi:10.1007/978-3-319-21840-3_45.

23 Jurij Mihelic and Borut Robic. Solving the k-center problem efficiently with a dominating set algorithm. *J. Comput. Inf. Technol.*, 13(3):225–234, 2005. `doi:10.2498/cit.2005.03.05`.

24 Mary Monroe and David M. Mount. A PTAS for the min-max euclidean multiple TSP. *CoRR*, abs/2112.04325, 2021. `arXiv:2112.04325`.

25 Javier Resano, Daniel Mozos, Diederik Verkest, Serge Vernalde, and Francky Catthoor. Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 585–594. Springer, 2003. `doi:10.1007/978-3-540-45234-8_57`.

26 Ola Svensson. Santa claus schedules jobs on unrelated machines. *SIAM J. Comput.*, 41(5):1318–1341, 2012. `doi:10.1137/110851201`.

27 Zhou Xu and Brian Rodrigues. A 3/2-approximation algorithm for multiple depot multiple traveling salesman problem. In Haim Kaplan, editor, *Algorithm Theory – SWAT 2010, 12th Scandinavian Symposium and Workshops on Algorithm Theory, Bergen, Norway, June 21-23, 2010. Proceedings*, volume 6139 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 2010. `doi:10.1007/978-3-642-13731-0_13`.